COMBINING HARDWARE MANAGEMENT WITH MIXED-CRITICALITY PROVISIONING IN MULTICORE REAL-TIME SYSTEMS

Namhoon Kim

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2019

Approved by:

James H. Anderson

Sanjoy K. Baruah

F. Donelson Smith

Donald E. Porter

Rodolfo Pellizzoni

Frank Mueller

ABSTRACT

Namhoon Kim: Combining Hardware Management with Mixed-Criticality Provisioning
in Multicore Real-Time Systems
(Under the direction of James H. Anderson)

Safety-critical applications in cyber-physical domains such as avionics and automotive systems require strict timing constraints because loss of life or severe financial repercussions may occur if they fail to produce correct outputs at the right moment. We call such systems "real-time systems." When designing a real-time system, a multicore platform would be desirable to use because such platforms have advantages in size, weight, and power constraints, especially in embedded systems. However, the multicore revolution is having limited impact in safety-critical application domains. A key reason is the "one-out-of-m" problem: when validating real-time constraints on an m-core platform, excessive analysis pessimism can effectively negate the processing capacity of the additional m-1 cores so that only "one core's worth" of capacity is available. The root of this problem is that shared hardware resources are not predictably managed. Two approaches have been investigated previously to address this problem: mixed-criticality analysis, which provision less-critical software components less pessimistically, and hardware-management techniques, which make the underlying platform itself more predictable.

The goal of the research presented in this dissertation is to combine both approaches to reduce the capacity loss caused by contention for shared hardware resources in multicore platforms. Towards that goal, fundamentally new criticality-cognizant hardware-management tradeoffs must be explored. Such tradeoffs are investigated in the context of a new variant of a mixed-criticality framework, called MC^2, that supports configurable criticality-based hardware management. This framework allows specific DRAM banks and areas of the last-level cache to be allocated to certain groups of tasks to provide criticality-aware isolation. MC^2 is further extended to support the sharing of memory locations, which is required to realize the ability to support real-world workloads.

We evaluate the impact of combining mixed-criticality provisioning and hardware-management techniques with both micro-benchmark experiments and schedulability studies. In our micro-benchmark experiments,

we evaluate each hardware-management technique and consider tradeoffs that arise when applying them together. The effectiveness of the overall framework in resolving such tradeoffs is investigated via large-scale overhead-aware schedulability studies. Our results demonstrate that mixed-criticality analysis and hardware-management techniques can be much more effective when applied together instead of alone.

ACKNOWLEDGEMENTS

This dissertation could not have been accomplished without the help and support of many people. First and foremost, I wish to thank my advisor, Jim Anderson, for his guidance and patience during my time at Chapel Hill. I would also like to thank my dissertation committee, Sanjoy Baruah, Don Smith, Donald Porter, Rodolfo Pellizzoni, and Frank Mueller, for their guidance on my research and this dissertation.

I also wish to thank all of my co-authors, Micaiah Chisholm, Nathan Otterness, Stephen Tang, Bryan Ward, Glenn Elliott, Jeremy Erickson, who all helped me on many different works, and often stayed up all night revising a paper before a submission deadline. I am especially thankful for Micaiah Chisholm. The results of this work were obtained as part of a collaborative effort with Micaiah. I would like to also thank my wonderful colleagues from the Real-Time Systems Group at UNC, Mac Mollison, Catherine Nemitz, Ming Yang, Sergey Voronov, Tanya Amert, Sims Osborne, Kecheng Yang, Zhishan Guo, Bipasa Chattopadhyay, Joshua Bakita, Clara Hobbs, and Peter Tong, who provided helpful conversations and comments on ideas and presentations.

I wish to acknowledge the help provided by the staff of the UNC Computer Science Department. I owe special thanks to Mike Stone, John Sopko, and Bil Hays for keeping our machines running, and Jodie Turnbull, Denise Kenney, and Melissa Wood for keeping me out of paperwork trouble. I would also like to extend my thanks to the friends I have made throughout my time at UNC, including Young-woon Cha, Cheng-Yang Fu, Amos Wang, Chun-Wei Liu, Ilwoo Ryu, Tahsin Kabir, Junpyo Hong, Eunbyung Park, and Seulki Lee. Finally, I would like to thank my parents for their love and support during the years I was in Chapel Hill.

The funding for this research was provided by NSF grants CNS 1115284, CNS 1218693, CPS 1239135, CNS 1409175, CPS 1446631, CNS 1563845, and CNS 1717589; AFOSR grant FA9550-14-1-0161; ARO grant W911NF-14-1-0499 and W911NF-17-1-0294; and support from General Motors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACET	Average-Case Execution Time
API	Application Program Interface
BI	Bank Isolation
C-EDF	Clustered Earliest Deadline First
CAT	Cache Allocation Technology
CE	Concurrency Elimination
CI	Cache Isolation
CL	Last-Level Cache Locking
CPU	Central Processing Unit
CVA	Compliant Vector Analysis
CW	Cache Warm
DAG	Directed Acyclic Graph
DIS	Data Intensive Systems
DMA	Direct-Memory Access
DRAM	Dynamic Random-Access Memory
EDF	Earliest Deadline First
FIFO	First In, First Out
G-EDF	Global Earliest Deadline First
GFP	Get Free Page
HRT	Hard Real-Time
I/O	Input/Output
ICAS	Ideal Cache Allocation Size
IPC	Interprocess Communication
JLFP	Job-Level Fixed-Priority
L1	Level 1
L1-D	Level 1 Data

L1-I	Level 1 Instruction
L2	Level 2
L3	Level 3
L4	Level 4
LLC	Last-Level Cache
LP	Linear Programming
MMU	Memory Management Unit
NC	No Copy
OS	Operating System
P-EDF	Partitioned Earliest Deadline First
P-RES	Partitioned Uniprocessor Reservation
PCB	Producer/Consumer Buffer
PET	Provisioned Execution Time
PFN	Page Frame Number
RM	Rate-Monotonic
RSS	Resident Set Size
RTOS	Real-Time Operating System
SBP	Selective LLC Bypass
SPM	Scratchpad Memory
SRT	Soft Real-Time
SSD	Solid State Disk
SWaP	Size, Weight, and Power
TLB	Translation Lookaside Buffer
USB	Universal Serial Bus
WCET	Worst-Case Execution Time
WFB	Wait-Free Buffer
WS	Working Set

WSS Working Set Size

CHAPTER 1: INTRODUCTION

Applications in safety-critical domains, such as in the avionics and automotive industries, require not only logical correctness but also temporal correctness. These applications must produce correct outputs within precise time constraints. Such applications are called *real-time applications*.

To guarantee the temporal correctness of real-time applications, the system designer must be able to predict application behavior. An analytical model allows the designer to do this, and these models are used to design and build real-time systems. However, since *multicore* architectures have emerged as the dominant platform for embedded systems, this type of analysis requires very pessimistic assumptions on provisioning worst-case behaviors due to the nature of multicore platforms. Such analysis pessimism results in severe capacity loss in designing real-time systems on such platforms. A multicore processor is a single computing component with two or more cores for enhanced performance, reduced power consumption, and more efficient simultaneous processing of multiple tasks. These multicore platforms have the potential to enable a wealth of new computationally intensive features in safety-critical domains, such as the avionics and automotive industries. Safety-critical systems require *certification* because these systems may cause significant damage or loss of life if they are not operating properly. Government and international agencies have published standards, reports, and position papers to provide guidelines in designing safety-critical real-time applications. However, certifying the real-time correctness of a system running on m cores may require pessimistic analysis to guarantee correct behavior that entails over-provisioning the required computational capacity. Such pessimism may be so extreme, that the processing *capacity* of the additional m-1 cores is entirely negated. In effect, only "one core's worth" of capacity can be utilized even though m cores are available. In safety-critical application domains, such as avionics, this "one-out-of-m" problem has led to the common practice of simply disabling all but one core. This problem is the most serious unresolved obstacle in research work on real-time multicore resource allocation today. Certification difficulties in multicore platforms are extensively discussed in a recent position paper by the U.S. Federal Aviation Administration, CAST-32A (Certification Authorities Software Team, 2016).

The root of the one-out-of-m problem is attributable to shared hardware resources, such as caches, buses,

and memory banks, that are not predictably managed. Several programs may access these shared hardware resources at the same time, which causes contention for those resources. This contention for shared resources results in pessimistic *worst-case execution time (WCET)* assumptions due to cross-core interference and unpredictability in accessing such resources, and this pessimism entails the under-utilization of a system. In this dissertation, we address the one-out-of-m problem by mitigating such pessimism in order to reduce capacity loss for multicore platforms.

We begin this chapter with a brief introduction to real-time systems and mixed-criticality analysis techniques, which are used in the approach we investigate. We then discuss multicore architectures and challenges caused by sharing hardware resources. Next, we present our thesis and describe the contributions we make via the results of this dissertation. Finally, we outline the organization of this dissertation.

1.1 Real-Time Systems

The meaning of the term "real time" varies widely depending on the application domain. To clarify our definition of "real time," we assume that a real-time system describes a hardware and software system where computational *tasks* that comprise one or more real-time applications are subjected to timing constraints. A real-time system is said to be *correct* when all tasks in the system satisfy both logical and temporal correctness. In other words, all tasks in the system must produce correct output within a designated time interval. Each task releases recurring work, which is called a *job*, according to a predictable rate or time interval. The completion time of each job must occur by a specified *deadline*, which occurs within some interval of time after the job's release. In a *hard real-time (HRT)* system, all deadlines must be met. If a job misses a deadline, the system is considered to have failed. In a *soft real-time (SRT)* system, some deadline misses are acceptable, but the response time of a job should be bounded. A task system, a set of real-time tasks, is said to be *schedulable* if timing constraints are guaranteed to be satisfied under a given scheduling algorithm, which is responsible for allocating processor time to each job. In order to guarantee that all timing constraints will be met under a given scheduling algorithm, *scheduliability analysis* must be performed.

1.2 Multicore Architectures

Over the last decade, multicore platforms have become the dominant hardware platform for real-time applications because they have advantages in size, weight, and power consumption (SWaP), especially in

embedded systems. In a multicore platform, two or more processors are integrated in a single physical circuit. The advent of multicore technologies aroused interest in the analysis and design of multiprocessor real-time scheduling algorithms. Scheduling algorithms in a multicore platform can be generally classified into three categories: partitioned, global, or clustered. Under partitioned scheduling, tasks are statically assigned to processors. Under global scheduling, tasks may migrate across processors. Clustered scheduling is a hybrid of partitioned and global scheduling: processors are partitioned into clusters, and each task is assigned to one cluster and can migrate among the processors of that cluster.

While multicore platforms provide many advantages, there are many challenges in designing real-time systems to run on a multicore platform. Since two or more processors exist in a multicore platform, those processors share many hardware resources, such as the last-level cache (LLC), memory bus, memory controller, and memory banks. Contention for this shared hardware can significantly affect the response time of real-time tasks on multicore platforms. For example, when a processor loads its instructions and data into the LLC, the subsequent accesses hit in the cache. However, the loaded instructions and data subsequently could be evicted by other processors because the LLC is shared among processors on multicore platforms. The unpredictability of these cache evictions can seriously affect the timing behavior of real-time tasks. Also, when all processors are executing memory-intensive applications, the applications could cause contention with respect to the memory controller, bus, and memory banks. This contention also affects the timing behavior of real-time tasks. Due to this unpredictable cross-core interference, pessimistic execution-time provisioning is often used in the analysis of such systems. This analysis pessimism results in capacity loss on multicore platforms.

1.3 Mixed Criticality

A *mixed-criticality* system is a system that contains several real-time applications of different levels of criticality. A criticality is a designation of the level of assurance needed for certification. For example, in avionics, the failure of a flight-control task may cause loss of life or severe injuries, while the failure of other tasks, such as a temperature-control task, may merely cause degraded performance. Traditionally, tasks of different criticalities have been located on separated processors to ensure predictable behavior. Today, due to the advantages of multicore architectures, it is possible to locate tasks of different criticalities on the same platform, which yields SWaP and cost benefits.

When designing a mixed-criticality real-time system, a common approach is to assign higher priorities to higher-criticality tasks. However, due to cross-core interference from lower-criticality tasks on other processors, this approach requires pessimistic WCET provisioning for higher-criticality tasks. This results in severe under-utilization on multicore platforms.

To mitigate such pessimism due to the unpredictable nature of shared resources on multicore platforms, two orthogonal approaches to improve system utilization have been investigated. One approach is to manage shared hardware. Resource partitioning is a promising approach for managing hardware predictably. For example, a dedicated partition in cache for a safety-critical task prevents its instructions and data from unexpected cache replacements caused by non-safety-critical tasks. Alternatively, support could be introduced for scheduling accesses to shared resources. For example, scheduling memory accesses may guarantee exclusive accesses and reduce pessimism by allowing deterministic accesses to shared resources.

While hardware-management approaches seem promising, another promising solution is the application of mixed-criticality analysis assumptions. Vestal (2007) proposed a *mixed-criticality task model* in which multiple WCETs are specified for each task at different levels of assurance. Since higher-criticality tasks require greater levels of assurance, a WCET at a higher criticality level is typically greater than WCETs for the same task at lower criticality levels. As explained in Chapter 2, when checking timing constraints, these different WCETs are applied in a way that allows for increased platform utilization.

1.4 Thesis Statement

To mitigate analysis pessimism due to the unpredictable shared resources on multicore platforms, two orthogonal approaches to improve system utilization have been investigated, namely, hardware-management techniques and mixed-criticality analysis, as discussed above. These prior research efforts share a similar goal of improving platform utilization, but their orthogonal nature raises research questions pertaining to the combination of both approaches: 1) Can we achieve better platform utilization by introducing criticality-aware hardware management? 2) If so, how should resources be managed both within and across criticality levels? 3) If we can provide task isolation by managing hardware resources, can we support real-world workloads that must necessarily break isolation?

Addressing these questions requires delving into sharing and isolation tradeoffs that have not been considered before. For example, while higher-criticality tasks might require strong hardware-isolation

guarantees, more optimistically provisioned lower-criticality tasks might actually *benefit* from less restricted hardware sharing because shared hardware is often designed to improve average-case performance or throughput. In addition, in order to realize the ability to support real-world workloads, sharing between tasks must be supported. Data sharing directly breaks the isolation often provided for higher-criticality tasks. *Shared libraries* are another common source of sharing. *Statically linked libraries* can eliminate such sharing, but this solution comes with a cost because all needed libraries are replicated on a per-task basis. Furthermore, user tasks are not the only source of hardware interference. The operating system (OS) contends for hardware resources as well. The OS executes concurrently with user-level tasks when it handles interrupts or runs system services.

Motivated by these observations, we present the following thesis statement:

> *"The capacity loss of real-time systems on multicore mixed-criticality systems can be mitigated by combining mixed-criticality analysis and hardware-management techniques, particularly by managing the shared LLC and dynamic random-access memory (DRAM) memory. Such a combination approach can be designed to mitigate hardware contention caused by data sharing between tasks, shared libraries, and sharing between the kernel and user space in order to support practical real-time workloads."*

To support this thesis, we have designed and implemented hardware-management techniques in a framework called MC^2 (mixed-criticality on multicore) (Herman et al., 2012; Mollison et al., 2010; Ward et al., 2013), which has been the subject of continuing research by the Real-Time Systems Group at the University of North Carolina at Chapel Hill. We have added support for data sharing between tasks, shared libraries, and sharing between the kernel and user space. We also have evaluated the proposed framework with overhead-aware schedulability studies.

1.5 Contributions

We now present an overview of our contributions in this dissertation that support the thesis above.

1.5.1 Enabling Hardware Management in MC^2

Our main contribution in this dissertation is the design and implementation of a hardware management framework in MC^2. In Chapter 2, we discuss the original MC^2 framework. In Chapter 3, we present our

new framework that enables shared hardware management in MC^2. The new MC^2 framework reduces capacity loss by providing criticality-aware task isolation. The new MC^2 framework is highly configurable and supports fine-grained resource management. We consider various configurations and evaluate each one with respect to real-time schedulability with measured platform overheads considered, using workload assumptions based on trends observed in benchmark programs.

1.5.2 Support for Data Sharing Between Tasks

In Section 4.1, we extend our MC^2 framework to support data sharing among tasks. When hardware management is combined with mixed-criticality analysis assumptions, the pessimism in provisioning WCETs can be greatly reduced. However, the ability to support real-world workloads has not been realized. A key barrier is a lack of support for data sharing among tasks. Data sharing among tasks could break the task isolation provided by hardware management in MC^2. We devised and implemented techniques to manage shared buffers that can be used to transfer data between tasks (Chisholm et al., 2016). We show that our buffer-allocation techniques eliminate or reduce the deterioration caused by the introduction of sharing.

1.5.3 Techniques to Allow Shared Libraries

A shared library is a module that is intended to be shared by applications. Libraries used by an application are loaded into memory at run time. However, this read-only sharing breaks isolation properties because the code is loaded by the first program that uses the library, which means that the code can be loaded into any LLC partition. To eliminate such unpredictable code pages, statically linked libraries can be used where all needed libraries are replicated on a per-program basis. This resolves the issue of breaking isolation, but statically linked libraries require more memory space.

In Section 4.2, we consider main memory as a constrained resource. When memory is considered as a constrained resource, the wasteful practice of fully replicating shared libraries can degrade system utilization significantly. To remedy this situation, we propose an approach that ensures hardware isolation without degrading system utilization (Kim et al., 2017a).

1.5.4 Techniques to Reduce OS-Induced Interference

User-level programs are not the only source of hardware interference. The OS contends for hardware resources as well. When a program performs I/O operations with external devices, such as a camera or

secondary storage devices, an interaction between the OS and a user program is required to complete each I/O operation. In addition, the OS frequently executes interrupt service routines or system services. These activities may interfere with a real-time task by consuming CPU cycles or by accessing shared hardware resources.

In Section 4.3, we propose a fine-grained memory allocation system that optimizes I/O buffer allocations (Kim et al., 2018). We also propose buffer-management techniques to reduce interference caused by interprocess communication (IPC) and I/O operations via direct-memory access (DMA). In Section 4.1, we consider shared memory as a means to share data among tasks because it does not require OS involvement when transferring data. In Section 4.3, we extend this data sharing to other IPC mechanisms by managing memory buffers allocated by the OS. We demonstrate the effectiveness of our buffer-management techniques by evaluating schedulability.

1.6 Organization

This dissertation is organized as follows. In Chapter 2, we discuss several background topics relevant to the contents and contributions of this dissertation, including real-time scheduling, mixed-criticality scheduling, hardware-management techniques, and multicore platforms. In Chapter 3, we present the new MC^2 framework that supports hardware management. In Chapter 4, we describe the extended MC^2 framework that supports data sharing between tasks and shared libraries and mitigates OS-induced interference. Finally, we conclude in Chapter 5.

CHAPTER 2: BACKGROUND

In this chapter, we discuss background material and prior work on topics related to this dissertation. We begin with real-time task models, multicore scheduling, mixed-criticality scheduling, and schedulability. We then discuss the multicore hardware platform considered in this dissertation and related work on hardware-management techniques. Finally, we introduce the MC^2 framework and LITMUS$^{\mathrm{RT}}$, which is the OS considered in this work.

2.1 Real-Time Scheduling

Before we discuss hardware-management techniques and the MC^2 framework, we define formal models for real-time schedulability. We then discuss scheduling algorithms on multicore platforms and describe an analytical approach to evaluate schedulability.

2.1.1 Task Model

In this dissertation, we consider the well-studied *sporadic task model* (Mok, 1983) and *periodic task model* (Liu and Layland, 1973). The computational workload is represented as a *task system*. We specifically consider a task system $\tau = \{\tau_1, \ldots, \tau_n\}$, scheduled on m processors. A *job* is a recurrent invocation of work by a *task*, τ_i, and is denoted by $J_{i,j}$ where j indicates the j^{th} job of τ_i (we may omit the subscript j if the particular job invocation is inconsequential). Task τ_i is described by a tuple of three parameters: (C_i, T_i, D_i). Under the sporadic task model, the releases of jobs $J_{i,j}$ and $J_{i,j+1}$ have a minimum separation time described by the task's *period*, T_i. Under the periodic task model, the separation time between *every* consecutive pair of jobs of τ_i is *exactly* T_i time units. The *hyperperiod* of a given task system τ is the least common multiple of T_i, where $i = 1, \ldots, n$. Every task τ_i has a *relative deadline*, D_i, and a *WCET* given by C_i. If $D_i = T_i$ for all tasks, then the task system is called an *implicit deadline* system and the tasks are denoted using a two-tuple $\tau_i = (C_i, T_i)$ notation. Task systems with $D_i \leq T_i$ are called *constrained deadline* systems, and if no relationship between D_i and T_i is assumed, they are called *arbitrary deadline* systems. In this dissertation,

we consider implicit-deadline task systems only. The *utilization* of task τ_i, which quantifies the long-term processor share required by τ_i, is given by

$$u_i = C_i/T_i. \tag{2.1}$$

The *total utilization* of a task system is defined as

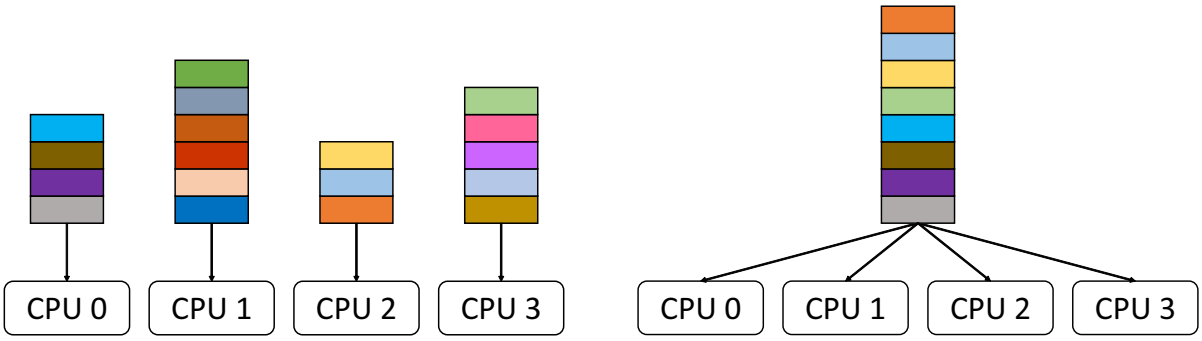
$$U = \sum_{i=1}^{n} u_i. \tag{2.2}$$

Job $J_{i,j}$'s release time and completion (*finish*) time are denoted $r_{i,j}$ and $f_{i,j}$, respectively. Each job $J_{i,j}$ has an *absolute deadline* $d_{i,j}$, by which the job $J_{i,j}$ should complete, D_i time units after its release time. The response time $R_{i,j}$ of a job $J_{i,j}$, the time between its release and completion, is defined by $f_{i,j} - r_{i,j}$, and the *tardiness* of a job $J_{i,j}$ is defined by $\max\{0, f_{i,j} - d_{i,j}\}$. Each job has a *precedence constraint* between successive jobs in that $J_{i,j}$ cannot be scheduled before $J_{i,j-1}$ completes.

2.1.2 Scheduling Algorithms

In this section, we discuss scheduling algorithms used in the MC^2 framework. We begin by defining the status of a job. A job is *ready* if it is released and available for execution. If the job is executing on a processor, then it is *scheduled*. A scheduled job is either *preemptible* or *non-preemptible*. In this dissertation, we consider preemptive scheduling algorithms, *i.e.*, a higher-priority job may preempt any lower-priority job that is running on a processor.

A scheduler is required to allocate ready jobs to a disjoint set of processors. Broadly speaking, there are two categories of schedulers: *static schedulers* and *dynamic schedulers*. Static schedulers, also known as clock-driven schedulers, make scheduling decisions by clock interrupts. The scheduling decisions are computed offline before the task system begins execution. The *cyclic executive* is a well-studied static scheduler (Baker and Shaw, 1989) for periodic tasks. It repeats a pre-computed scheduling table every *major cycle*. A major cycle is defined by the task-system hyperperiod. Cyclic executives are widely used in safety-critical applications because they are simple and straightforward to validate. Dynamic schedulers, also known as priority-driven schedulers, evaluate the priorities of ready jobs and select the highest-priority job for execution. In this dissertation, we specifically focus on *job-level fixed-priority (JLFP)* scheduling algorithms where fixed priorities are assigned to jobs. We categorize JLFP schedulers into two groups:



(a) Partitioned scheduling. (b) Global scheduling.

Figure 2.1: Examples of two scheduling approaches on a quad-core platform.

fixed- or *dynamic-priority*. Fixed-priority schedulers assign the same priority to all jobs of the same task. For example, the *rate-monotonic (RM)* scheduler is a fixed-priority scheduling algorithm where tasks are prioritized by periods, *i.e.*, a shorter period results in a higher priority. The *earliest-deadline-first (EDF)* scheduler is a dynamic-priority scheduling algorithm where each job is prioritized by its absolute deadline, with earlier deadlines having higher priority.

We now discuss scheduling approaches in multicore platforms. In this dissertation, we consider partitioned and global schedulers. Figure 2.1 illustrates these two scheduling approaches and ready queues for a system with four processors. A ready queue is a priority-ordered queue of ready jobs that are waiting to be scheduled on a processor. In this figure, a colored box represents a ready job in the queue. Under partitioned scheduling, each processor has a dedicated ready queue, while a single global ready queue is used under global scheduling.

Different partitioned and global schedulers are distinguished by how they prioritize jobs. For example, partitioned-EDF (P-EDF) adopts a partitioned approach and uses EDF priorities. Similarly, global-EDF (G-EDF) adopts a global approach and uses EDF priorities.

2.1.3 Schedulability

A schedule is *feasible* for a task system if all required timing constraints are satisfied in that schedule. A task system is *schedulable* under a given scheduling algorithm if the algorithm always produces a feasible schedule. The definition of feasibility and schedulability are with respect to timing constraints. For a HRT task, *all* deadlines must be met. For a SRT task, some deadline misses are acceptable, but its tardiness should be bounded by a constant. A scheduling algorithm is *optimal* if it always produces a feasible schedule when

one exists. A *schedulability test* must be performed to evaluate whether a task system is schedulable under a given scheduling algorithm.

Uniprocessor schedulability. We begin with uniprocessor schedulability. Liu and Layland (1973) showed that any implicit-deadline periodic task system scheduled by the uniprocessor RM scheduler is HRT-schedulable if

$$U \leq n(2^{\frac{1}{n}} - 1). \tag{2.3}$$

This test is only *sufficient* because task systems exist with utilizations satisfying $n(2^{\frac{1}{n}} - 1) < U \leq 1$ that are feasible to schedule that cannot be deemed unschedulable by the schedulability test in (2.3). An *exact* schedulability test always returns true if a task system can be correctly scheduled under a given scheduler. Liu and Layland (1973) also showed that any implicit-deadline periodic task system scheduled by the uniprocessor EDF scheduler is HRT-schedulable if and only if

$$U \leq 1. \tag{2.4}$$

Uniprocessor EDF is therefore optimal because it always produces a feasible schedule unless the processor is overutilized.

Schedulability on multicore platforms. We now discuss schedulability on multicore platforms consisting of m processors, in which $m > 1$. Under partitioned scheduling, as tasks are statically assigned to processors, we can apply a uniprocessor schedulability test to each processor. However, a task assignment problem arises when we evaluate the schedulability of a task system because tasks must be assigned to processors so that no processor is overutilized. This requires solving the bin-packing problem, which is known to be NP-hard in the strong sense (Dhall and Liu, 1978). For example, consider a task system with three identical tasks, $\tau_i = (2, 3)$ where $1 \leq i \leq 3$, on a system with two processors. The total utilization of this task system is two, which is equal to the number of processors. However, there is no partitioning of tasks that does not overutilize one processor. One processor must have two tasks, which is overutilized with a utilization of $\frac{4}{3}$. Therefore, this task system is not feasible under partitioned scheduling. Such a bin-packing problem results in capacity loss on multicore platforms. López et al. (2004) derived a utilization bound for P-EDF scheduling when using the *first-fit*, *best-fit*, or *worst-fit decreasing* task assignment heuristic. In these heuristics, tasks are

implicit-deadline periodic and ordered by decreasing utilization, that is, $u_1 \geq u_2 \geq \ldots \geq u_n$. The first-fit heuristic assigns each task τ_i to the first processor where it fits. The best-fit heuristic selects the processor with the lowest free capacity among those with enough capacity to hold the task. The worst-fit heuristic is the inverse of the best-fit heuristic, that is, each task τ_i is allocated to the processor with the highest free capacity.

Under global scheduling, the m highest-priority jobs are scheduled at any time. A single ready queue serves all processors in the system under global priority-driven schedulers. Thus, it does not need to solve the bin-packing problem, which causes capacity loss under partitioned scheduling. The schedulability of global scheduling has been investigated by many researchers, resulting in many schedulability tests. In this dissertation, we focus on SRT-schedulability under G-EDF for sporadic task systems as our MC^2 framework uses G-EDF for SRT systems. The details of the MC^2 framework will be discussed in Section 2.2. Devi and Anderson (2005) showed that G-EDF under SRT timing constraints has no capacity loss. In particular, an implicit-deadline sporadic task system under G-EDF is SRT-schedulable with bounded tardiness if the utilization constraints

$$U \leq m \tag{2.5}$$

and

$$\forall \tau_i : u_i \leq 1 \tag{2.6}$$

both hold. The tardiness x_i of any job in a sporadic task system τ under preemptive G-EDF is bounded by

$$x_i = \frac{(\sum_{\tau_j \in E(\tau)} C_j) - C_{min}}{m - \sum_{\tau_k \in T(\tau)} u_k} + C_i, \tag{2.7}$$

where C_{min} is the smallest C_i in the task system τ, $E(\tau)$ is the subset of $m-1$ tasks in τ with the largest C_j values, and $T(\tau)$ is the subset of $m-1$ tasks in τ with the largest u_k values.

Figure 2.2 shows the schedule of a task system with three identical periodic tasks, $\tau_i = (2, 3)$, under G-EDF on two processors. If there is a tie, we break it by task index. After time $t = 3$, the schedule settles into a steady pattern. The task system presented in Figure 2.2 is not schedulable under partitioned scheduling, while it is SRT-schedulable under G-EDF. This task system satisfies the task system utilization (Inequality (2.5)) and per-task utilization (Inequality (2.6)) constraints. The tardiness bound of this task system is two time units (Equation (2.7)). In Figure 2.2, we observe that the maximum tardiness of task τ_3 is one time unit. Thus, there exists a difference between the analytical bound and the observed bound due to

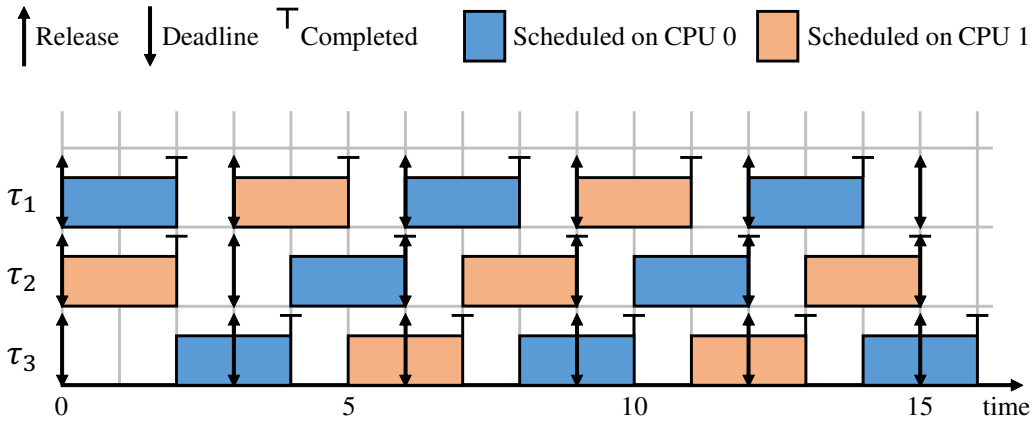


Figure 2.2: Example of bounded tardiness for a task system under G-EDF on two processors.

analytical pessimism. Erickson (2014) improved the analytical tardiness bound by introducing Compliant Vector Analysis (CVA), which uses pseudo-deadlines called *priority points*.

2.1.4 Mixed-Criticality Scheduling

The conventional approach to designing a safety-critical real-time system is to use pessimistic WCET provisioning for higher-criticality tasks, which results in the under-utilization of computational capacity. Vestal (2007) proposed a technique to mitigate such under-utilization of a system for uniprocessor platforms. He observed that the WCETs of higher-criticality tasks are needlessly pessimistic from the perspective of scheduling lower-criticality tasks. Thus, he proposed schedulability tests for mixed-criticality systems to reclaim capacity loss in practice. When checking the schedulability of lower-criticality tasks, less-pessimistic execution times are assumed. To achieve this, multiple WCET values are required for each task: one for its own criticality level and one for every other criticality level. More formally, in a system with L criticality levels, each task has a WCET specified at every level, and L system variants are analyzed: in the Level-l variant, the real-time requirements of all Level-l tasks are verified with Level-l WCETs assumed for *all* tasks of equal or higher-criticality level. The degree of pessimism in determining WCETs is level-dependent: if Level l is of higher criticality than Level l', then Level-l WCETs will generally be greater than Level-l' WCETs. For example, in the system considered by Vestal (2007), observed WCETs were used to determine WCETs for tasks at lower-criticality levels, and such execution times were inflated to determine WCETs at higher-criticality levels. The task model resulting from Vestal's work has come to be known as the *mixed-criticality task model*.

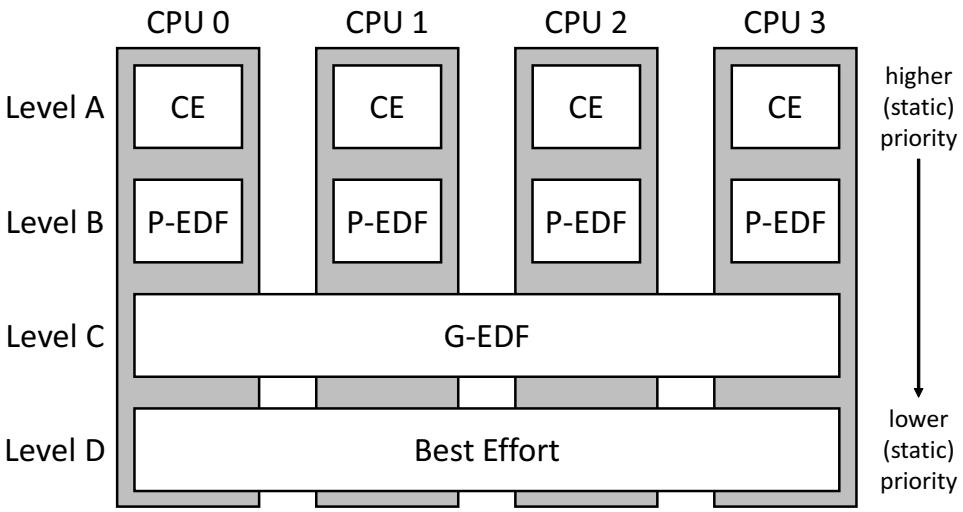


Figure 2.3: Scheduling in MC^2 on a quad-core machine.

2.2 MC^2

Vestal's work led to a significant body of follow-up work. Anderson et al. (2009) proposed a mixed-criticality scheduling approach for multicore platforms that uses a hierarchical scheduling framework called MC^2 (mixed-criticality on multicore), which provides isolation for tasks of different criticality levels (Mollison et al., 2010; Herman et al., 2012; Ward et al., 2013). MC^2 was the first mixed-criticality scheduling framework for multicore processors. MC^2 supports four criticality levels, denoted A (highest) through D (lowest), as shown in Figure 2.3. Higher-criticality tasks are statically prioritized over lower-criticality cones. In MC^2, tasks at each criticality level can be scheduled by different scheduling algorithms, allowing appropriate schedulers to be used per level. The selected scheduling algorithms for each level are illustrated in Figure 2.3.

Level-A tasks are partitioned and scheduled on each core using a time-triggered table-driven cyclic executive. Tasks are statically assigned to processors, and a dispatching table for each processor is used. As Level-A tasks are the highest-criticality tasks in the system, Level-A tasks are always scheduled when they become eligible. Level-B tasks are also partitioned but are scheduled using a P-EDF scheduler on each processor.[1] Level B has the second-highest priority, so any eligible Level-B task will be scheduled when no Level-A task is executing on the same processor. On each processor, the Level-A and -B tasks are required to be *simply periodic* (all tasks commence execution at time 0 and periods are harmonic), and the Level-B

[1]Other partitioned scheduling algorithms such as partitioned RM can be used for Level B.

task periods are integer multiples of the Level-A hyperperiod. Level-C tasks are scheduled via a G-EDF scheduler. The G-EDF scheduler is invoked on any processor whenever Level-C tasks are eligible but no higher-criticality tasks are eligible. Level-A and -B tasks are HRT, Level-C tasks are SRT, and Level-D tasks are non-real-time. In this dissertation, we assume that Level D is not present because it is afforded no real-time guarantees.

In MC^2, we adopt a measurement-based approach to determining *provisioned execution times (PETs)*[2] because work on static timing-analysis tools for multicore machines has not matured to the point of being directly applicable. Moreover, measurement-based processes for determining PETs are often used in practice. As in prior work on MC^2 (Mollison et al., 2010; Herman et al., 2012; Ward et al., 2013), we assume that Level-C PETs reflect measured *average-case execution times (ACETs)* (since Level C is SRT) and that Level-B PETs reflect measured WCETs (since Level B is HRT). Further, we assume that Level-A PETs are defined by applying an inflation factor to Level-B PETs (since Level A is of highest criticality).

MC^2 was originally designed in consultation with colleagues in the avionics industry. A major thesis underlying its design is that Levels A and B would be mostly comprised of quite deterministic "fly-weight" tasks with rather low utilizations; less-deterministic computationally intensive tasks of higher utilization would likely be assigned to Level C.

2.3 Multicore Platforms

In this section, we discuss multicore architectures and considerations in designing real-time systems on multicore platforms. We begin with a discussion of the key components of multicore architectures. We then discuss cross-core interference caused by shared hardware resources.

2.3.1 Processing Units

A multicore platform has two or more processors and hardware resources shared among the processors. A processor is a functional unit that consists of computation units and caches. The multicore platform improves overall performance by running multiple tasks in parallel, while a uniprocessor platform uses time slices to run multiple tasks. Multicore platforms could be implemented via two different configurations: heterogeneous platforms that use more than one kind of processor and homogeneous platforms that have identical processors.

[2]We use "PET" instead of "WCET" because under MC^2, some tasks are SRT, and hence may not be provisioned on a worst-case basis.

In this dissertation, we consider homogeneous multicore platforms in which all processors are running at the same clock speed.

2.3.2 Memory

When a processor runs a task, it needs to access instructions and data stored in memory. Historically, computer systems have hierarchical memory with multiple storage levels. *Registers* are located inside the processor and typically hold a word of data. Registers are the fastest storage in a system, but they are relatively few in number. *Cache* is an intermediate storage between registers and *main memory*. Cache was introduced to improve performance by copying frequently accessed instructions and data from main memory into the cache, which is faster but smaller than main memory. Caches are usually organized as multiple levels of caches. Typically, level 1 (L1) caches are split into instruction (L1-I) and data (L1-D) caches, while the level 2 (L2) cache is a unified cache that stores both instructions and data. Some hardware platforms have additional levels (L3 or L4).

Main memory, also known as *physical memory*, is connected to processors via a *memory bus* and managed by a *memory management unit (MMU)*, which translates *virtual memory addresses* to *physical memory addresses*. The mapping between virtual addresses and physical addresses is constructed by the OS and stored in *page tables*. A *page* is a fixed-length contiguous block of virtual memory. The MMU divides the virtual address space into pages: a page is the smallest unit of memory management conducted by the OS. A *page frame* is the fixed-length contiguous block of physical memory. Main memory consists of multiple DRAM banks that store page frames. In multicore platforms, DRAM banks and memory buses are shared among all processors and accesses to DRAM are controlled by the DRAM controller.

2.3.3 An Example Hardware Platform

We now describe the hardware platform considered in this dissertation. Our platform is the NXP i.MX6 quad-core ARM Cortex-A9 evaluation board. This platform provides two essential hardware features, cache lockdown and disabling bank interleaving, which are required for applying hardware-management techniques in MC^2. The basic architecture is illustrated in Figure 2.4. Each core on this machine is clocked at 800Mhz and has separate 32KB L1 instruction and data caches. Additionally, the L2 cache, which is the LLC on this machine, is a shared, unified 1MB 16-way set-associative cache. This platform contains 1GB of off-chip DRAM, which is divided into eight 128MB DRAM banks.

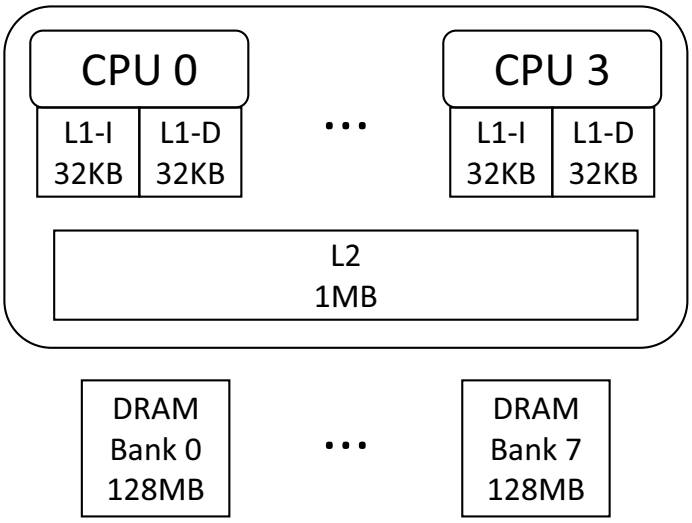


Figure 2.4: The architecture of quad-core ARM Cortex-A9.

2.4 Shared Hardware Resources

In this section, we discuss cross-core interference that arises from contention for shared hardware resources and previous research into mitigating such interference. We primarily focus on two key components of the memory hierarchy: caches and DRAM banks.

2.4.1 Caches

When a task needs to access a memory address, the processor checks whether the requested address exists in a cache. If the processor finds that the data is in the cache, a *cache hit* occurs; if not, then a *cache miss* occurs. In the case of a cache miss, the processor copies data from main memory to the cache in blocks of fixed size called *cache lines*. L1 caches are accessible only to the corresponding processor, so cross-core interference does not exist. However, the shared LLC can be concurrently accessed by multiple tasks executing on different processors. If two tasks on different processors request data that have the same mapping address in the LLC, one task may evict the previous cache line. Even though our platform has 16-way set-associative LLC in which each entry in main memory can be loaded into any one of 16 places in the LLC, cache evictions could occur depending on the replacement policy of the cache. Such cross-core evictions increase memory-access latency and may result in deadline misses of a job if the impact of cross-core evictions has not been properly accounted for. The cross-core interference in caches is highly unpredictable,

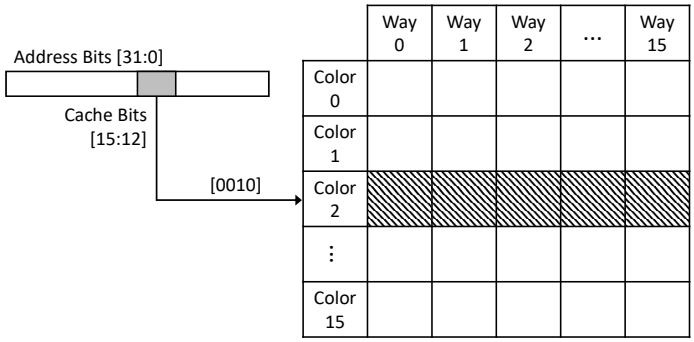


Figure 2.5: Set-based cache partitioning on the ARM Cortex-A9.

so the current state-of-the-art standard practice in using multicore platforms for safety-critical systems is to disable all but one core, which entails severely underutilizing a system, as mentioned in Chapter 1.

Set-based partitioning. To mitigate cross-core cache evictions, a *cache partitioning* approach has been proposed (Kirk, 1989; Bui et al., 2008; Altmeyer et al., 2014). Under cache partitioning, the shared LLC is divided into several partitions, and each task or processor is assigned to disjoint sets of cache lines so that no overlap exists in the LLC in order to eliminate cross-core evictions. Such allocations can be determined by the compiler (Mueller, 1995), or the OS can control the mapping by *page coloring* (Kessler and Hill, 1992). Under page coloring, colors are assigned to page frames and corresponding cache lines.[3] For example, imagine assigning the color "0" to the first page frame and corresponding cache lines. In a similar way, assign the color "1" to the next page frame and so on. Eventually, such color assignments will wrap, and we will sequence through the same colors again. The coloring process is based on physical memory addresses as depicted in Figure 2.5. In Figure 2.5, four bits are used to assign a color to a page as the LLC has 2,048 cache lines (16 colors) per way. This process ensures that differently colored pages map to different cache lines in the LLC. Thus, accesses to two differently colored pages cannot cause cache conflicts. By using page coloring, we can implement *set-based* cache partitioning.

Way-based partitioning. *Way-based* partitioning is also possible by using additional hardware features, such as *cache lockdown*, that are available on some platforms. Cache lockdown allows a processor to specify which ways of the cache are locked down. The locked cache ways are protected from being evicted. Way-based partitioning can be achieved by assigning disjoint sets of cache ways to each processor. In other

[3]Generally, the size of a page frame is greater than a cache line, so one page frame is mapped with a set of cache lines.

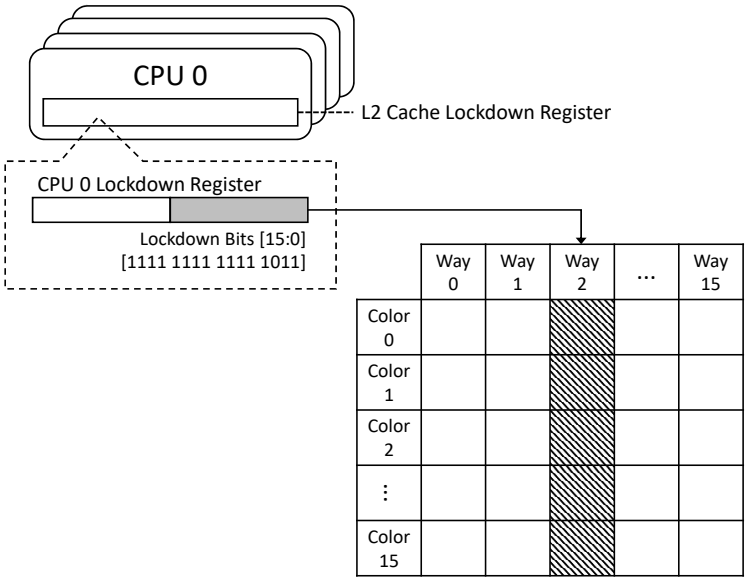


Figure 2.6: Way-based cache partitioning on the ARM Cortex-A9.

words, each processor has its own cache ways to eliminate cross-core evictions. In our platform, cache lockdown is supported by per-processor lockdown registers as depicted in Figure 2.6. In Figure 2.6, only Way 2 is unlocked from CPU 0's perspective, so CPU 0 can only use Way 2 for allocating cache lines. This can be achieved by setting all bits to 1 except the bit 2.

Prior work on managing caches. Page coloring and cache lockdown techniques are used in recent works to eliminate or mitigate cache evictions. Mancuso et al. (2013) proposed *colored lockdown* techniques that permanently lock the most commonly accessed pages into the LLC. Xu et al. (2016) proposed a dynamic cache allocation algorithm under global fixed-priority scheduling to reduce cache-related overheads. However, these papers do not consider mixed-criticality systems. Ward et al. (2013) proposed *cache locking* and *cache scheduling* techniques to improve system utilization for mixed-criticality systems. This paper considers a scheduling-based approach to LLC management for higher-criticality tasks only. In that work, the OS prefetches all potentially accessed pages before a higher-criticality job executes, and enforces that co-scheduled jobs do not conflict in the LLC. Lower-criticality jobs are allowed to execute in periods of high LLC contention that otherwise would have idled a processor. In this dissertation, we take a more holistic approach to criticality-aware LLC management, and consider hardware-management tradeoffs within and among all criticality levels.

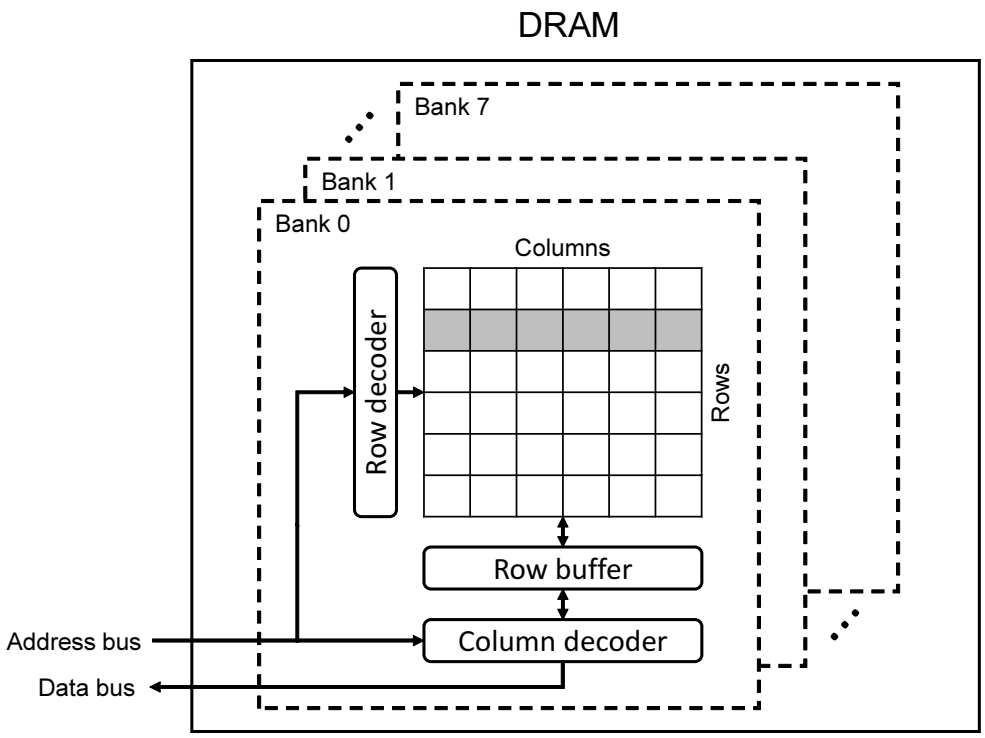


Figure 2.7: The architecture of a DRAM chip that has eight banks.

2.4.2 DRAM Banks

Memory references missed in caches result in a main memory access with longer latency. Figure 2.7 illustrates the architecture of a DRAM chip. A DRAM chip consists of multiple banks, and each bank consists of multiple rows and columns of memory cells. The *row buffer*, which stores accessed data, acts as a cache within DRAM. When a read or write operation is issued, the address is decoded by row and column decoders, and then the row buffer transfers data to the data bus. If the corresponding row is found in the row buffer, a *row buffer hit* occurs. Otherwise, a *row buffer miss* occurs, which increases memory-access latency. When row buffer miss occurs, the existing data in the row buffer must be copied back to the bank array, and then the new row is copied into the row buffer. Thus, memory references from different processors may interfere with each other in multicore platforms. Many techniques have been proposed to mitigate such unpredictability caused by row buffer conflicts.

Prior work on mitigating memory contention. One approach to mitigating DRAM bank interference due to row conflicts is to partition DRAM banks. Similar to cache partitioning, memory banks can be partitioned by page coloring, as a physical address of a page frame determines DRAM bank allocations. Yun et al. (2014)

proposed a bank-aware memory-allocation framework called PALLOC to mitigate bank-level interference by partitioning DRAM banks. In addition to bank partitioning, managing the DRAM controller has also been considered as a method to reduce interference due to bus contention. Yun et al. (2013) presented a framework called MemGuard to limit memory bandwidth by suspending tasks that use too much memory bandwidth within a given interval. However, these papers do not consider mixed-criticality systems.

2.4.3 Unmanaged Hardware Resources

The LLC and DRAM banks are not the only source of cross-core interference. Other hardware resources such as L1 caches, translation lookaside buffers (TLBs) (Panchamukhi and Mueller, 2015), memory controllers (Hassan et al., 2015; Guo and Pellizzoni, 2017), memory buses (Muench et al., 2014; Seetanadi et al., 2017), or cache-related registers (Valsan et al., 2016) can be contention sources. In this dissertation, we do not manage these shared hardware resources. However, we assume a measurement-based approach to determining PETs, so such uncontrolled resources are implicitly considered when determining PETs as we measure execution times under the presence of contention for such uncontrolled resources. We also assume that sufficient measurements are taken to cover the worst-case behavior of all tasks with respect to unmanaged resources.

2.5 LITMUS$^{\mathrm{RT}}$

The major contribution of this dissertation is to reduce capacity loss in multicore platforms. As we mentioned in Chapter 1, we propose to combine hardware management with mixed-criticality analysis to improve system utilization. To evaluate our proposed techniques, we conduct micro-benchmark experiments and schedulability studies with measured platform overheads. Unfortunately, most real-time operating systems (RTOSs) do not support mixed-criticality task models and hierarchical scheduling, which are required to implement the MC^2 framework. Further, these RTOSs lack support for fine-grained memory management. The MC^2 framework presented in Chapter 3 requires kernel modifications to support a mixed-criticality task model and mixed-criticality scheduling as shown in Figure 2.3. To implement and evaluate our hardware-management techniques, our MC^2 framework is implemented in the LITMUS$^{\mathrm{RT}}$ (LInux Testbed for MUltiprocessor Scheduling in Real-Time systems) kernel. LITMUS$^{\mathrm{RT}}$ is an open-source real-time extension to Linux, which has been developed by the Real-Time Systems Group at the University of North

Carolina at Chapel Hill and Max Planck Institute for Software Systems since 2006 (Calandrino et al., 2006; Brandenburg, 2011; LITMUS$^{\mathrm{RT}}$ Project, 2018). LITMUS$^{\mathrm{RT}}$ provides interfaces within the kernel that facilitate the prototyping of real-time scheduling algorithms, and it is a useful experimental platform for real-time systems

2.6 Summary

In this chapter, we described the real-time task model and the mixed-criticality task model that will be considered in this dissertation. We also discussed relevant concepts pertaining to real-time scheduling algorithms and schedulability. We then considered multicore platforms and shared hardware resources that cause cross-core interference. We closed by delineating prior work on mitigating contention for shared hardware resources.

CHAPTER 3: ENABLING HARDWARE MANAGEMENT IN MC^2 [4]

In this chapter, we present the design of hardware management that we added to the original MC^2 framework. Ward et al. (2013) proposed shared LLC management techniques in MC^2. However, the techniques proposed by them require prefetching every page of the next task and using synchronization protocols. They also did not support criticality-aware cache management. In this dissertation, we propose a holistic way to manage shared hardware resources, which does not require prefetching pages and using synchronization protocols to manage the shared LLC. We focus on managing the shared LLC and DRAM banks in the hardware platform depicted in Figure 2.4. The resulting MC^2 framework is highly configurable and allows us to explore tradeoffs between sharing and isolation in a criticality-cognizant way. The hardware-management techniques in MC^2 support DRAM bank partitioning, and fine-grained cache partitioning, which combines set-based and way-based cache partitioning. We present several resource allocation strategies and evaluate each strategy that we propose. We also evaluate the efficacy of our combination of hardware management and mixed-criticality provisioning by conducting a large-scale overhead-aware schedulability study.

This chapter is organized as follows. We begin by presenting the hardware-management techniques we employ. We then discuss resource allocation strategies. Following this discussion, we describe the details of our implementation in MC^2. We then present an evaluation of our techniques. Finally, we conclude this chapter.

3.1 Hardware Management Techniques

The goal of managing shared hardware is to reduce capacity loss on multicore platforms. This goal can be achieved by providing criticality-cognizant task isolation, which removes or mitigates cross-core

[4] Contents of this chapter previously appeared in preliminary form in the following papers:
Kim, N., Ward, B., Chisholm, M., Anderson, J., and Smith, F. D. (2017b). Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. *Real-Time Systems*, 53(5):709–759;
Kim, N., Ward, B., Chisholm, M., Fu, C., Anderson, J., and Smith, F. D. (2016). Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. In *Proceedings of the 22nd IEEE Real-Time Embedded Technology and Application's Symposium*, pages 1–12.

interference caused by contention for shared resources. There are important choices to make in providing criticality-cognizant task isolation. First, we must decide how to configure the shared LLC and DRAM banks for each criticality level. For example, we can assign a dedicated partition to each criticality level for strong task isolation or we may allow lower-criticality tasks to share the LLC and DRAM banks. Second, we must determine the size of each partition of the LLC and DRAM banks for better schedulability. Therefore, we design and implement our hardware management framework to support two abilities:

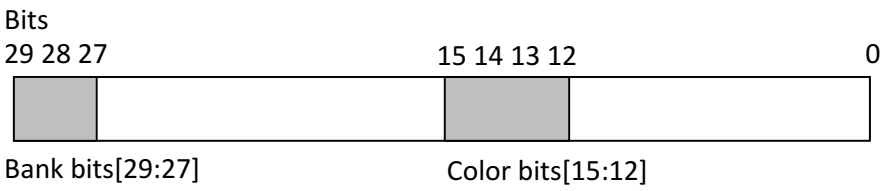
1. The framework can assign LLC and DRAM bank partitions at each criticality level independently.
2. The framework can freely change the size of each partition.

The first ability enables us to explore tradeoffs between sharing and isolation with respect to the LLC and DRAM banks. The second ability enables us to optimize the utilization of system resources in terms of schedulability. In order to provide these two properties, we designed and implemented a kernel module that partitions the shared LLC and DRAM banks and a user interface to configure each partition. Partitioning the shared LLC and DRAM banks can be achieved by managing page allocations in the OS. As we discussed in Chapter 2, we can assign distinct sets of pages to tasks in which two differently colored pages cannot cause conflicts in the shared LLC and DRAM banks.

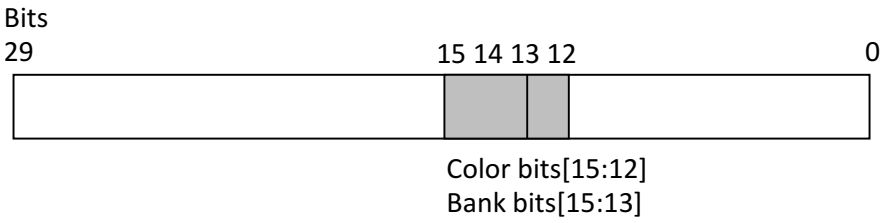
3.1.1 Cache Partitioning

We now describe the design of cache partitioning in detail. We begin by explaining way-based partitioning. We then describe set-based partitioning.

Way-based partitioning. For way-based partitioning, we require hardware features that manage way allocations in the shared LLC. Our platform provides a cache lockdown mechanism called *lockdown by master*. Each processor has a dedicated per-processor cache lockdown register. The lockdown register defines available cache ways from the corresponding processor's perspective. For example, we can configure the shared LLC so that Way 2 is unlocked for CPU 0 and locked for other processors as depicted in Figure 2.6. This configuration allows CPU 0 to allocate cache lines on Way 2 and cache lines on Way 2 cannot be evicted by a task running on other processors. Such an assignment can be easily changed at runtime by writing new values to the cache lockdown registers. We change the lockdown registers whenever the scheduler picks the



(a) Bank interleaving is off.



(b) Bank interleaving is on.

Figure 3.1: Physical address decoding of NXP i.MX6 processor.

next task to execute. Thus, we can achieve way-based cache partitioning according to the criticality of the next task.

Set-based partitioning. Set-based partitioning requires a hardware platform featuring the shared LLC organized as a physically tagged cache. Figure 3.1 shows the physical address decoding of our platform when bank interleaving is on and off. Details concerning bank interleaving will be discussed in Section 3.1.2. As discussed in Section 2.4.1, set-based partitioning can be achieved by page coloring. The address bits [15:12] determine the color of the shared LLC, which controls the mapping between cache lines and physical page frames. Once physical page frames are allocated to a task, the mapping between virtual addresses and physical addresses cannot be modified unless pages are swapped out to disk. In this dissertation, we do not allow swapping to disk at runtime, which means that all pages of a real-time task must be reside in memory until the task exits. Now, we can imagine two approaches to assign colored pages to tasks: 1) Allocate colored pages to a task when it is created. 2) Migrate pages before a task starts executing.

The former approach requires prior information about a task's criticality and processor assignment. However, it is not possible to know such information in LITMUS$^{\mathrm{RT}}$. A real-time task in LITMUS$^{\mathrm{RT}}$ is forked as a normal task when a new task is created. Then, real-time task parameters including the new task's criticality level are passed to the kernel via a system call. To implement the former approach, it is necessary to modify process-creation procedures in Linux. After a careful consideration of the two approaches, we

concluded that the former approach is not feasible due to difficulties in implementation effort, efficiency, and stability, while the latter approach is relatively easy to implement by adding a system call that migrates pages in the kernel. Thus, we chose to adopt the latter approach. We migrate original pages of all tasks in a task system to properly colored pages before the task system begins executing. However, some pages are not eligible to migrate. If pages are shared by other tasks or the kernel, they are not migratable. In this chapter, we assume that all pages are migratable, *i.e.*, no shared pages exist in tasks. We will discuss techniques to handle different types of shared pages in Chapter 4.

3.1.2 DRAM Bank Partitioning

The OS does not consider DRAM banks when allocating memory. In the kernel, memory is considered as a single resource. This could cause cross-core contention in DRAM banks if all processors are accessing the same bank at the same time. To make DRAM banks more predictable, we divide DRAM banks into multiple partitions and assign a dedicated partition to each processor so that memory accesses from different processors do not cause cross-core interference. To implement DRAM bank partitioning, the address mapping information between DRAM banks and physical page frames is required. Fortunately, the architecture manual of our platform provides the exact information about physical address decoding as depicted in Figure 3.1. As our platform has eight DRAM banks, three bits in the physical address determine the location of a particular DRAM bank. Since DRAM bank partitioning can be achieved in the same manner as set-based cache partitioning, we use the same mechanism, migrating pages, to partition DRAM banks.

Bank interleaving. Bank interleaving is a technique that parallelizes memory accesses to improve memory throughput by spreading contiguous pages across DRAM banks. As shown in Figure 3.1, the bank interleaving setting determines the location of bank address bits. As a result, our platform can be configured in two different memory layouts. Figures 3.2 and 3.3 illustrate page layouts in DRAM banks when bank interleaving is off and on, respectively. In both figures, a box with a *page frame number (PFN)* and a color in the shared LLC represents one physical page frame. When bank interleaving is off, the bank bits [29:27] and the color bits [15:12] do not overlap with each other, as shown in Figure 3.1 (a). As illustrated in Figure 3.2, each bank has pages of all 16 colors and 32,768 consecutive pages, which enables us to assign cache partitions and bank partitions independently. However, when bank interleaving is on, each bank has pages of only two colors because the bank bits [15:13] are overlapped with the color bits [15:12]. This is illustrated in Figures

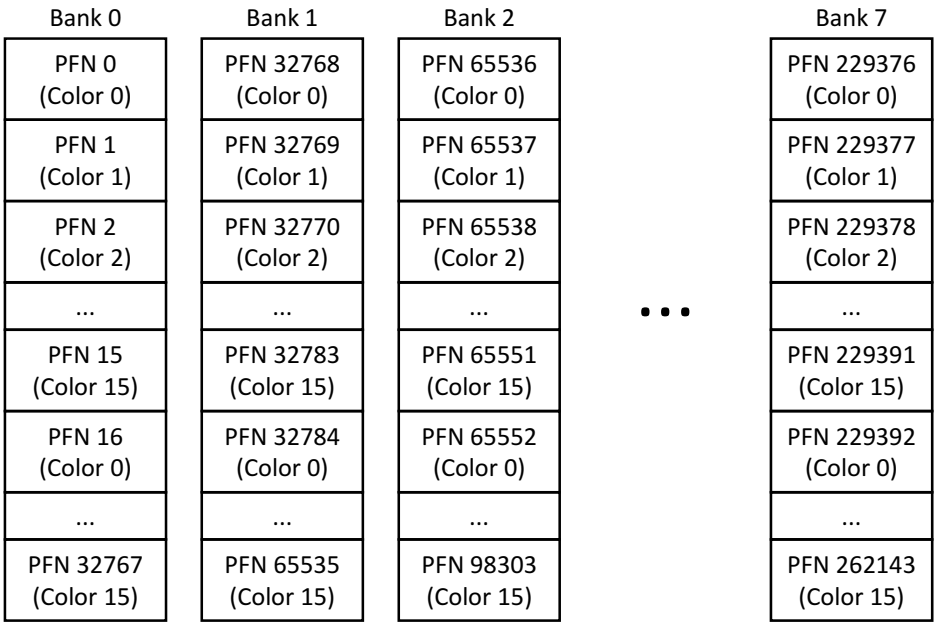


Figure 3.2: Page layout in DRAM banks when bank interleaving is off.

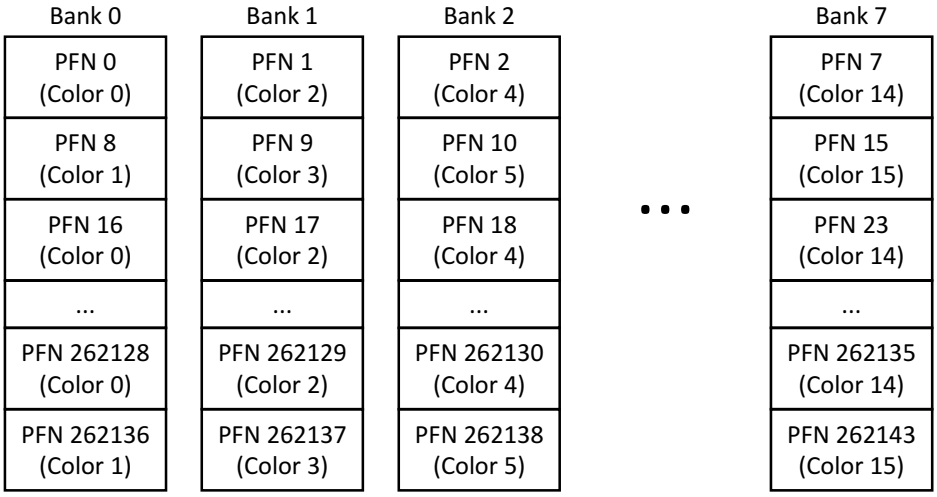


Figure 3.3: Page layout in DRAM banks when bank interleaving is on.

3.1 (b) and 3.3. In this dissertation, we disable bank interleaving because it permits more fine-grained control over page allocations. In addition, we can remove interference caused by accessing OS pages when bank interleaving is off. We will discuss this issue in Section 3.3.4.

3.2 Resource Allocation Strategy

In this section, we begin by describing our LLC allocation strategies considered in this chapter. We then describe DRAM allocation strategy and the benefits of disabling bank interleaving.

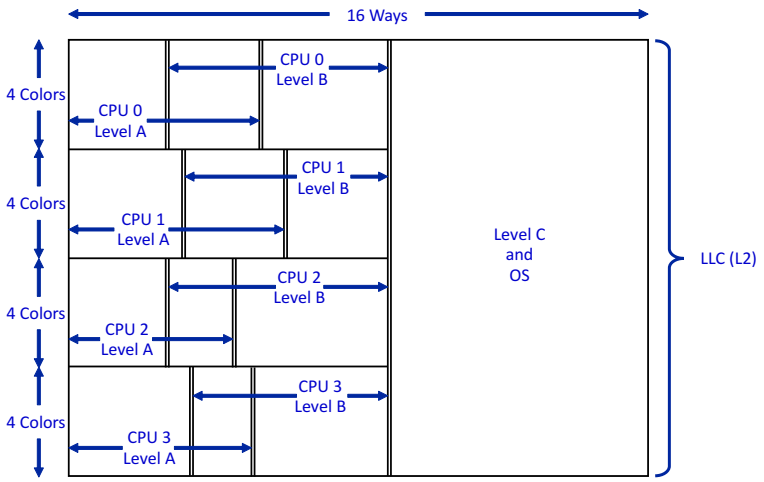
3.2.1 LLC Allocation

The LLC-allocation strategies studied in this dissertation are depicted in Figure 3.4. In this figure, LLC boundaries indicated by double lines are settable parameters. We categorize these allocation strategies into three separate *variants*.

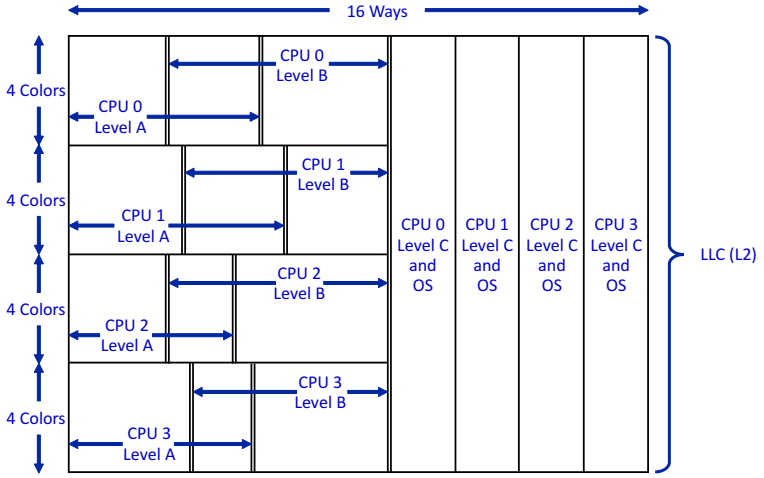
Variant 1, shown in Figure 3.4 (a), ensures strong isolation guarantees for Level-A and -B tasks while allowing for fairly permissive hardware sharing for Level-C tasks when used in combination with our DRAM-allocation strategy. As seen, Level C and the OS share a subsequence of the available LLC ways and all LLC colors because the OS pages may have more than four physically contiguous pages. As noted in Section 2.2, we assume that Level-C tasks (being SRT) are provisioned on an average-case basis. Under this assumption, LLC sharing with the OS should not be a major concern. The remaining LLC ways are partitioned among Level-A and -B tasks on a per-processor basis. That is, the Level-A and -B tasks on a given core share a partition. Each of these partitions is allocated a quarter of the available colors, as depicted. This scheme ensures that Level-A and -B tasks do not experience LLC interference due to tasks on other processors (spatial isolation). LLC interference between Level-A and -B tasks on the same processor may occur, but this interference only affects Level-B tasks, as Level-A tasks execute with higher priority. However, it may still be desirable to limit this interference. As a consequence, different LLC areas are allocated to Level A and B within a processor partition, but these areas may overlap.[5]

Variants 2 and 3, depicted in insets (b) and (c) of Figure 3.4, are analyzed in later chapters to characterize the advantages and disadvantages of isolation and sharing for each criticality level. In Variant 2, the LLC area allocated to Level C and the OS is partitioned by way on a per-processor basis. This variant provides stronger isolation guarantees to Level-C tasks but reduces the LLC area that such a task can utilize. In Variant 3, Level-A and -B tasks are not partitioned by processor, thus giving each Level-A and -B task access to all

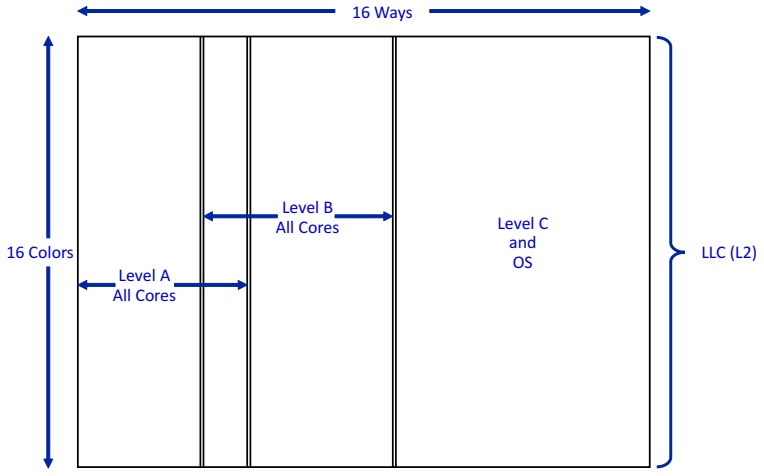
[5]We often use the term "area" instead of "partition" to describe these allocated LLC regions because of the potential for some regions to overlap.



(a) Variant 1 (HRT-isolated and SRT-shared LLC).



(b) Variant 2 (All-isolated LLC).



(c) Variant 3 (All-shared LLC).

Figure 3.4: LLC allocation variants.

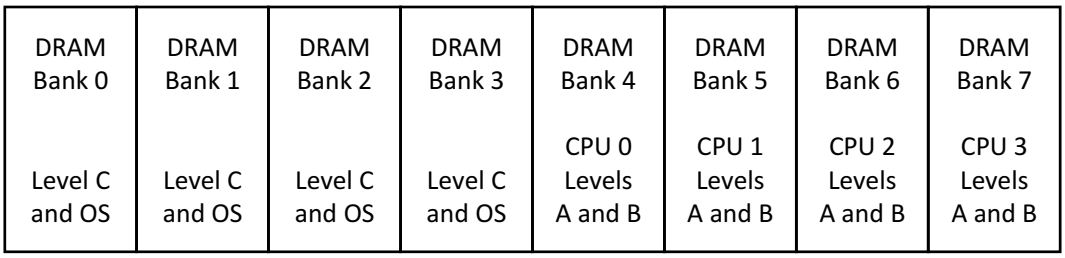


Figure 3.5: DRAM allocation strategy.

16 colors. This reduces isolation guarantees at Levels A and B, but increases the LLC area that Level-A and -B tasks can utilize.

The specific number of LLC ways assigned to each allocated LLC area is a tunable parameter that affects the execution times of tasks, as demonstrated later in Section 3.4. We determine values for these parameters on a per-task-system basis using optimization techniques proposed by Chisholm et al. (2015). The optimization techniques seek to minimize a task system's Level-C utilization while ensuring schedulability at all criticality levels.

3.2.2 DRAM Allocation and Bank Interleaving

Our DRAM allocation strategy is depicted in Figure 3.5. To provide strong isolation for HRT tasks, each processor has a dedicated DRAM bank for Level-A and -B tasks. Level-C tasks and the OS share the remaining four banks so that SRT tasks and the OS do not affect Level-A or -B tasks with respect to DRAM accesses. At memory initialization time, the OS reserves a range of physical pages used by the kernel and peripheral devices. The arrangement of the OS kernel code and data in our platform is presented in Figure 3.6. The reserved pages used by the kernel code and data are not movable and will never be swapped out. If we enable bank interleaving, such pages are spread across DRAM banks as shown in Figure 3.3, which may cause contention in Level-A and -B banks. However, with interleaving disabled, we can isolate the reserved pages from Level-A and -B tasks because the reserved pages are located in Bank 0.

3.3 Implementation

In this section, we discuss the implementation of the hardware management framework in MC^2. We have already discussed the basic hardware-management strategies in Section 3.2. We provide additional implementation details here. We begin by providing general information on the implementation of hardware

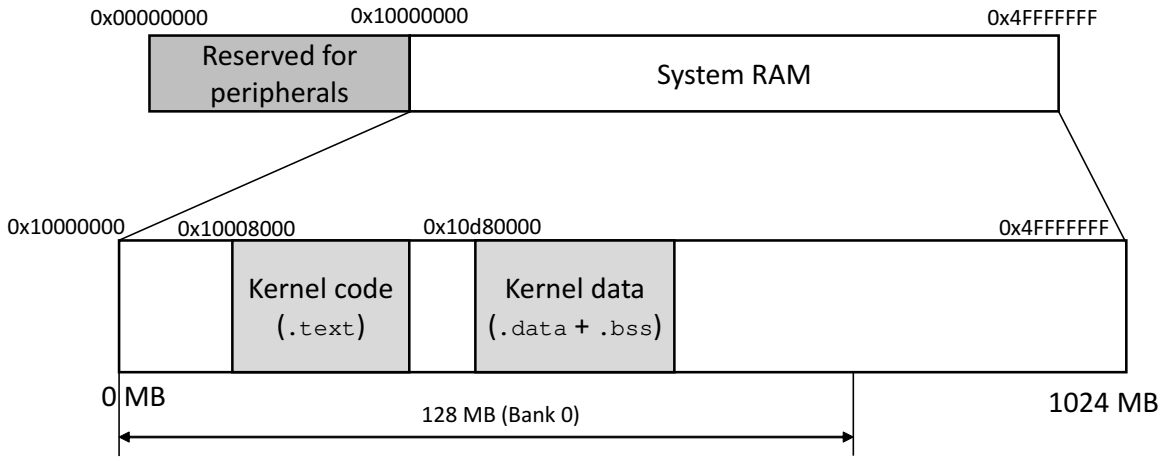


Figure 3.6: The arrangement of the kernel code and data in memory.

management in MC^2. We then discuss the MC^2 task scheduler and a system call that migrates pages in MC^2. We also discuss the coarse-grained OS isolation provided by our MC^2 framework.

3.3.1 General Information

The hardware management framework consists of two key components: a mixed-criticality task scheduler and a resource partitioning module. We implemented these components as an extension to LITMUS$^{\mathrm{RT}}$, version 2015.1, which is based upon the 4.1.3 Linux kernel.[6]

3.3.2 MC^2 Task Scheduler

Linux can use different scheduling algorithms to schedule different types of tasks by introducing *scheduler classes*. The base scheduler iterates over each scheduler class in order of priority. The highest priority scheduler class that has a runnable process picks the next task to run. LITMUS$^{\mathrm{RT}}$ introduces a new scheduler class, `SCHED_LITMUS`, to schedule real-time tasks. The `SCHED_LITMUS` class has the highest priority of all classes, so real-time tasks are guaranteed to have priority over normal Linux tasks. In LITMUS$^{\mathrm{RT}}$, a new scheduling algorithm can be added to `SCHED_LITMUS` as a scheduler plugin. We implemented our MC^2 scheduler as a plugin to LITMUS$^{\mathrm{RT}}$. We implemented three different scheduling policies to schedule mixed-criticality tasks because MC^2 has three criticality levels that require real-time constraints.[7]

[6]The code is available at `https://wiki.litmus-rt.org/litmus/Publications`.

[7]Level-D tasks are non-real-time. Thus, the MC^2 scheduler does not schedule Level-D tasks. Such tasks can be scheduled by Linux when a processor exists that has no eligible MC^2 tasks to execute.

LITMUS$^{\mathrm{RT}}$ provides several real-time scheduling algorithms as scheduler plugins. The *P-RES (partitioned uniprocessor reservation)* scheduler in LITMUS$^{\mathrm{RT}}$ supports the partitioned cyclic-executive and P-EDF scheduling algorithms. However, as MC^2 uses G-EDF for scheduling Level-C tasks, we extended the P-RES scheduler to support MC^2 tasks. In P-RES, a *reservation* is a schedulable entity that may have more than one task. Each reservation has a *budget* (*i.e.*, an OS-enforced execution time) and a *replenishment period*. This reservation structure is used to realize budget enforcement, which is required to implement MC^2 (Mollison et al., 2010; Herman et al., 2012).

In MC^2, we modified the structure of a reservation to have only one task. The budget of a reservation is equal to the associated task's PET at its own criticality level and the replenishment period is equal to the task's period. To statically prioritize each criticality level, we implemented a set of reservations, known as a *container* (in other literature, called a *server*). Each processor has a set of reservations, called a *local container*, that has Level-A and -B reservations for tasks assigned to that processor. In a local container, Level-A reservations are prioritized over Level-B reservations. Thus, the local container can serve as both a dispatching table for the cyclic executive and a ready queue for Level-B tasks. For Level-C tasks, we implemented a set of reservations, called a *global container*, that is shared among all m processors. Reservations in the global container are sorted in the order of deadlines. When the MC^2 scheduler is invoked to select the next task to run on the current processor, it first selects a container. Then, the scheduler selects the next task from the selected container, according to a scheduling algorithm associated with that particular container.

3.3.3 Allocating Colored Pages to Tasks

As we discussed in Section 3.1, a physical address of a page determines both its LLC color and DRAM bank. Therefore, we can achieve set-based partitioning and DRAM bank partitioning by modifying the memory-management system in Linux. We begin by describing the memory-management system in Linux. We then present our modifications to this system.

Memory management in Linux. In Linux, physical pages are managed and allocated by the *buddy allocator* (Knowlton, 1965; Knuth, 1968). Memory is divided into blocks of pages where the size of each

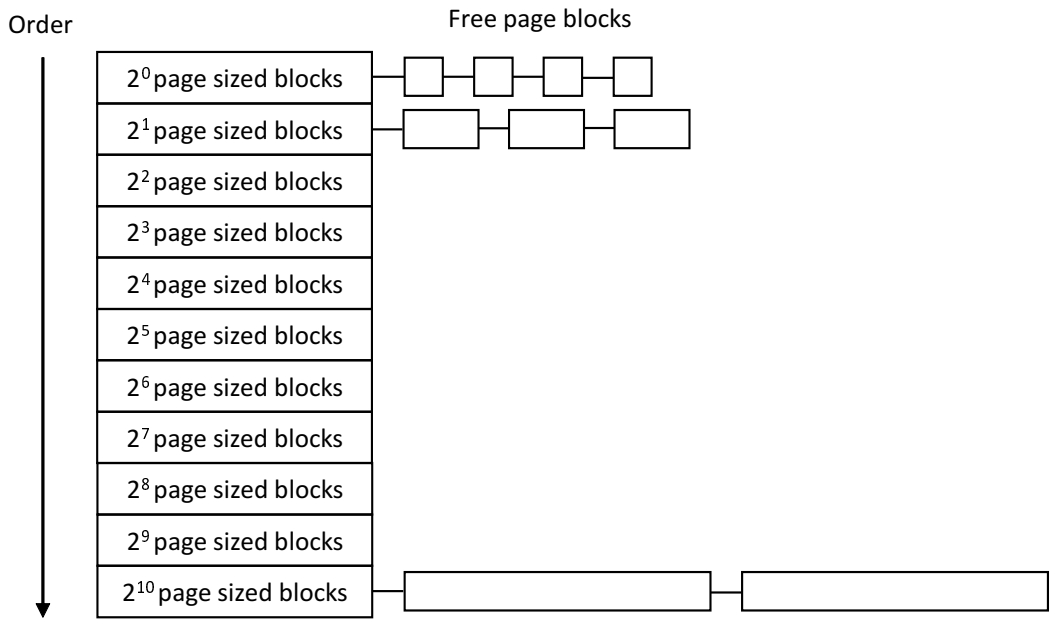


Figure 3.7: A list of free blocks managed by the buddy allocator.

block is 2^k pages for some k. The buddy allocator maintains a list of free blocks.[8] Figure 3.7 illustrates the management of free blocks in Linux. When the OS kernel allocates memory, the buddy allocator searches for an appropriately sized block. When a block of the requested size is not available, a larger block is divided into two half-sized blocks. These two blocks are buddies to each other. One half is used for the allocation and the other is added to the list of free blocks. When a block is freed, the buddy is examined and combined if it is free.

Our modifications to the buddy allocator. To properly allocate LLC colors and DRAM banks to tasks, we modified the buddy allocator in Linux. Our modification to the buddy allocator consists of replacing the single list of free pages with m+1 independent lists where m is the number of processors in the system. This is illustrated in Figure 3.8. These m lists hold free pages for the Level-A and -B tasks on each of the m processors. The additional list holds free pages for Level-C tasks and other non-real-time tasks. By default, we use Level-C/OS pages for all page allocations because a task does not have a criticality when it is created. When we migrate pages via a system call, a new page is allocated from the buddy allocator. As the

[8]The buddy allocator maintains a list of free blocks per a *zone*. A zone is a group of pages that have similar properties. The hardware platform considered in this dissertation has only one zone, `ZONE_NORMAL`, in the system. However, other architectures may have multiple zones such as `ZONE_DMA`, `ZONE_NORMAL`, and `ZONE_HIGHMEM`. In such platforms, the buddy allocator can have more than one list (Gorman, 2004).

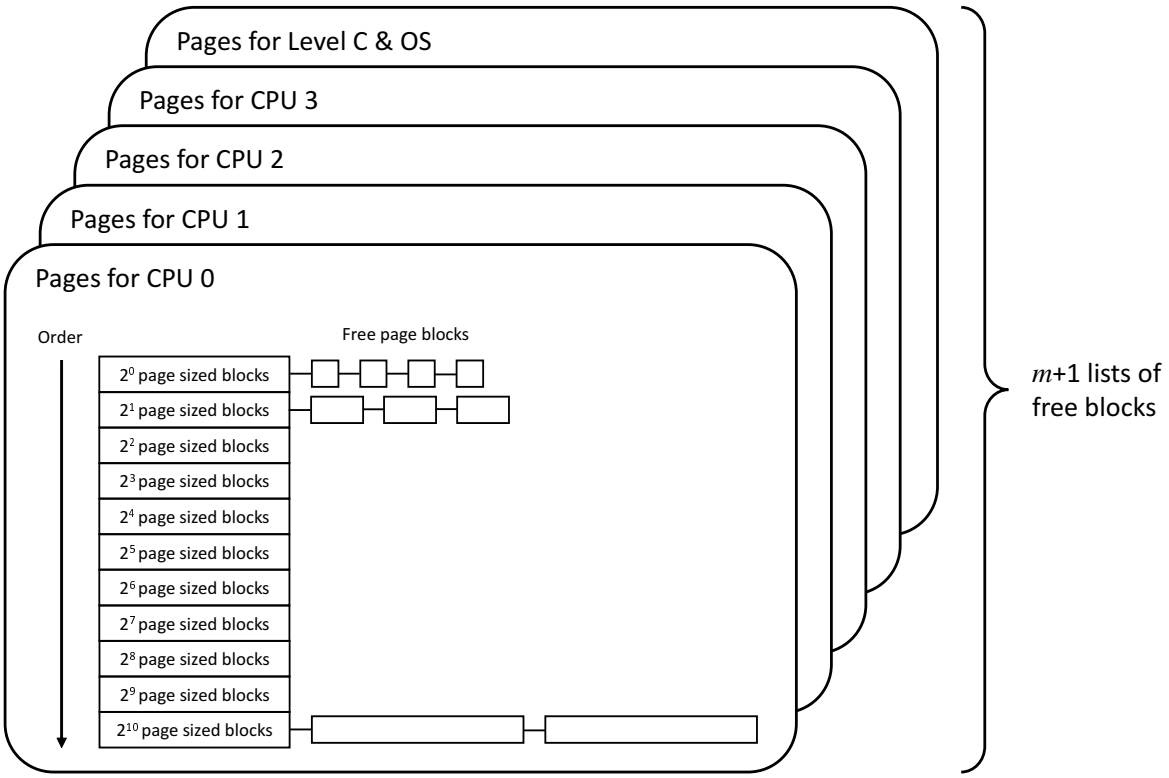


Figure 3.8: Modified m+1 lists of free blocks in the buddy allocator.

buddy allocator has information about the requesting task's criticality and assigned processor, we can allocate properly colored pages to tasks. After the new page is initialized, the contents of the old page are copied to the new page. Then, the old page is removed from the task, and the new page is inserted into the task. This procedure is repeated for every page of the task. Figure 3.9 shows a possible outcome of this migration for a task assigned Colors 0–3. In this figure, each box represents a page with the assigned color. The system call that migrates pages is performed before tasks commence execution, and as a result, tasks incur no runtime overheads due to page migrations.

3.3.4 Coarse-Grained OS Isolation

Our hardware management framework isolates the OS from Level-A and -B tasks in the LLC via way-based partitioning. Specifically, whenever the kernel begins executing on a processor as the result of an interrupt, exception, or a system call, we save the current value of that processor's lockdown register and then modify it so that the OS accesses only certain LLC ways in kernel mode. When exiting kernel mode, we restore the lockdown register using the saved value. Together with the DRAM bank partitioning, this

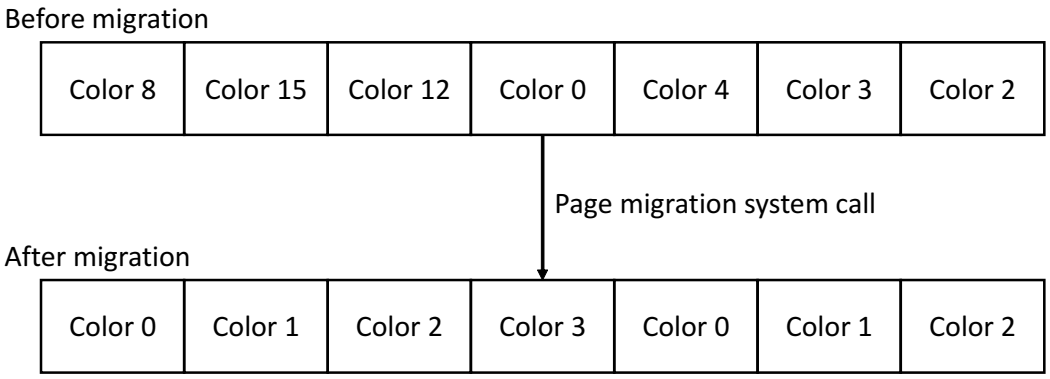


Figure 3.9: Migration of seven pages to a task assigned the first four LLC colors.

ensures that the OS only minimally interferes with Levels A and B. However, this coarse-grained OS isolation technique has a limitation. When the OS executes a system call on behalf of Level-A or -B tasks, the LLC ways for Level-C and OS will be used for Level-A and -B tasks. This may cause interference from Level-C tasks. In the experiments presented in this chapter, we avoid such interference by not allowing tasks to invoke system calls. We resolve this issue by fine-grained OS isolation, which will be discussed in Section 4.3.

3.4 Micro-Benchmark Experiments

We experimentally assessed the impact of combining mixed-criticality provisioning and hardware management. First, to assess the impact of hardware management, we collected extensive trace data for synthetic micro-benchmark programs and a benchmark program. Next, to assess the impact of interference caused by the OS, we examined the impact of providing coarse-grained OS isolation.

Evaluation process. We examined the impact of hardware management by collecting trace data for both synthetic micro-benchmark programs devised by us and a publicly available benchmark program. The micro-benchmark programs were designed as stress cases to demonstrate the upper limits of potential performance improvements made possible by LLC and DRAM-bank management. Each micro-benchmark program consists of a main loop that is repeated 500 times. During each loop iteration, a different randomly chosen sequence of unique word addresses is read, where each address aligns with the first word in a cache line (32 bytes on our hardware). Note that the word read at an address identifies the next address to read, eliminating the need to call a pseudo-random number generator during the benchmark's execution. Every available cache line is referenced once in an iteration. This access pattern has the effect of forcing each cache reference to

a random line and eliminating hits for successive references within a line (reducing spatial and temporal locality in references). Each micro-benchmark program has a specified *working set* (*WS*), which is the set of addresses used to reference data, and correspondingly a *working set size* (*WSS*). We used *Matrix* program from the *Data Intensive Systems (DIS) Stressmark Suite* (Musmanno, 2003), which was designed to reflect memory-usage patterns common in real-world use cases.

Quantifying cache-usage patterns is harder for real application code than micro-benchmark programs. However, there must naturally be a point of diminishing returns for larger and larger LLC allocations for any program (assuming it is executed in isolation). We call this point, where execution times do not substantially decrease given a larger LLC allocation, a program's *ideal cache allocation size* (*ICAS*). Note that it is possible for a program to have an ICAS larger than the LLC. In such cases, we define its ICAS to match the LLC size. Our micro-benchmark programs have a very small code footprint, so for them, ICAS is the same as WSS.

3.4.1 Impact of Providing Full Isolation at Levels A and B

In our first set of experiments, we examined the impact of providing full isolation to a task by giving it a dedicated LLC area and/or DRAM bank that are accessed by no other task. The LLC allocations for Level-A and -B tasks are depicted in insets (a) and (b) of Figure 3.4. When full isolation is provided, these experiments have implications when determining PETs for Level-A and -B tasks of Variants 1 and 2 in Figure 3.4. Such tasks can only experience interference from other tasks due to preemptions, and any execution-time increases due to preemptions are dealt with in overhead accounting.

To assess the impacts of providing full isolation, we ran experiments in which a measured program (either a micro-benchmark program or the Matrix program) was run alone on one core either in the presence of no other running programs—we call this the *idle scenario*—or along with *stress-inducing programs* running concurrently on the other three cores—we call this the *loaded scenario*. The loaded scenario was further factored into four cases: **(i)** *no cache or bank isolation*, **(ii)** *bank isolation but no cache isolation*, **(iii)** *cache isolation but no bank isolation*, and **(iv)** *both cache and bank isolation*. This yielded a total of five isolation configurations. The stress-inducing programs were configured to have a write ratio of 1/4 with a WSS of

1MB.[9] When bank isolation was not provided, these programs were configured to specifically target the DRAM bank used by the measured program.

For each measurement and isolation configuration, we divided the LLC into two partitions: one for the measured program and the other one for the stress-inducing programs. For the LLC partition for the measured program, we considered 256 possible LLC area sizes (given by 1 to 16 ways and 1 to 16 colors) for allocation to the measured program. The rest of the LLC space is shared by the stress-inducing programs. Each additional way or color increases the allocated LLC space by 4KB. This process yielded $256 \times 5 =$ 1,280 experiments per measured program. In each such experiment, we ran the measured program 100 times and recorded the observed WCET and ACET from the data collected. We are interested in both WCETs and ACETs because both are used in MC^2 provisioning, as discussed in Section 2.2.

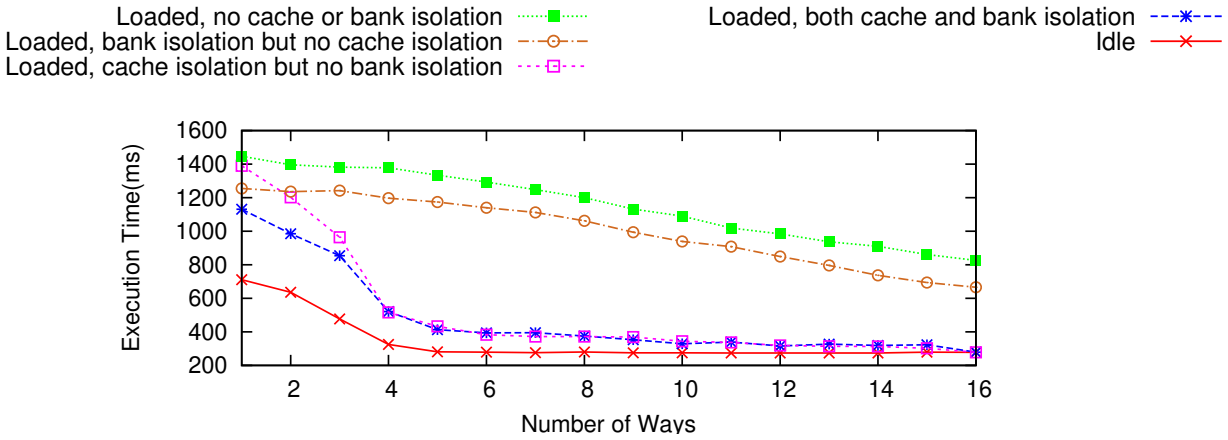
We collected trace data (8GB in total) for the Matrix program and for micro-benchmark-programs with WSSs in $\{32, 64, 256, 512\}$KB. (Recall that the L1 caches on our hardware platform are 32KB.) We now make several observations based on this data. We support these observations using the data in Figures. 3.10–3.12, which depict recorded WCETs and ACETs for the 256KB-WSS and 32KB-WSS variants of the micro-benchmark program and the Matrix program, given an allocated LLC area consisting of 16 colors and some number of ways, and 16 ways and some number of colors.

Observation 3.1. *Providing LLC isolation reduced WCETs by up to* 369% *for the micro-benchmark program and by up to* 142% *for the Matrix program.*

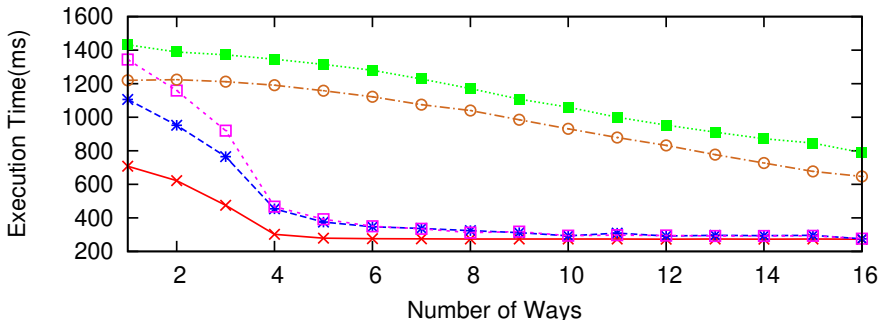
The noted 369% (respectively, 142%) reduction can be seen by comparing the curves in Figure 3.10 (c) (respectively, Figure 3.11 (a)) at the data points corresponding to five colors (respectively, twelve ways). As expected, our micro-benchmark tasks generally exhibited greater benefits from LLC isolation than the DIS programs. However, isolation comes at a cost. For example, if we choose to isolate Level-C tasks by way on a per-processor basis as shown in Figure 3.4 (b), then each Level-C task would only be able to access one quarter of the available LLC area size (assuming it is divisible by four) instead of sharing the entire area.

Observation 3.2. *LLC isolation had a greater impact on WCETs as LLC space approached the ICAS. Beyond this point, WCETs were only marginally greater than in an idle system, implying that unmanaged*

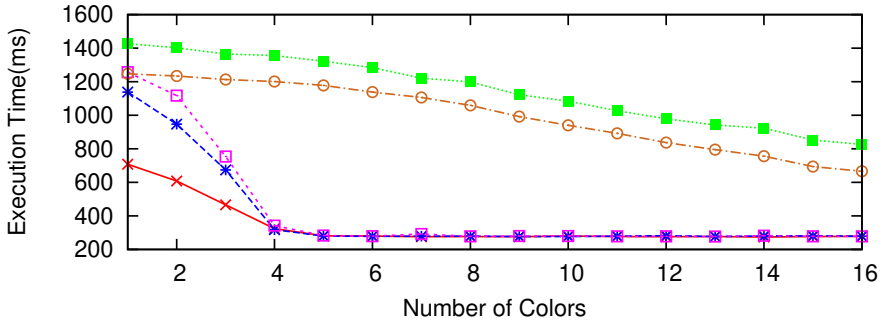
[9] We leveraged measurement methods presented in Brandenburg (2011). For our considered platform, we measured overheads of various write ratios ranging over $\{0, 1/4, 1/2, 1\}$, and a ratio of 1/4 showed the highest overheads.



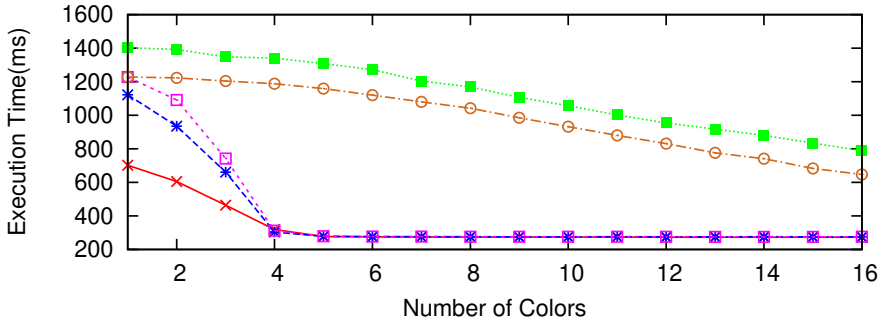
(a) WCET assuming 16 colors.



(b) ACET assuming 16 colors.

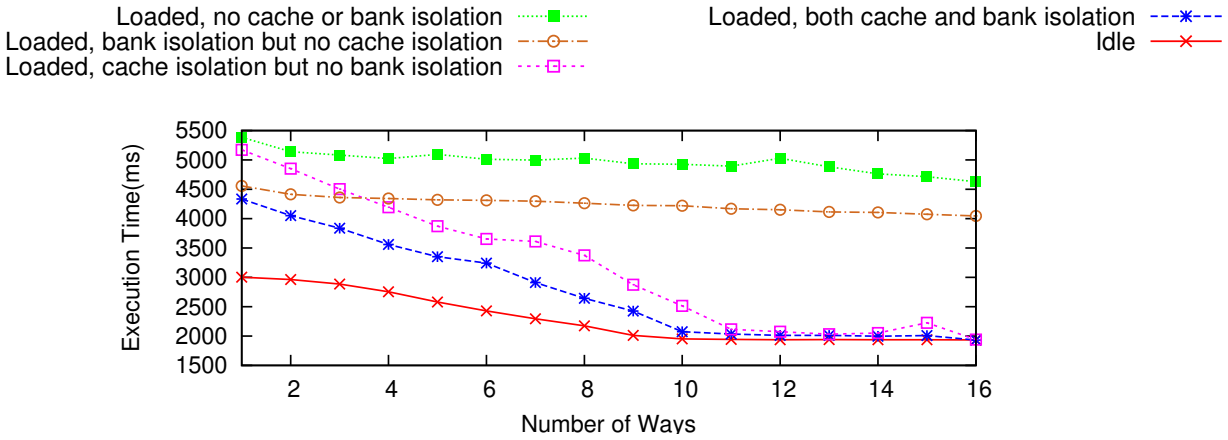


(c) WCET assuming 16 ways.

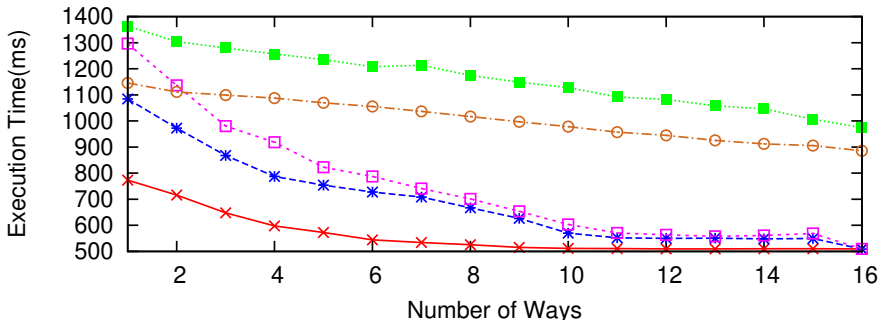


(d) ACET assuming 16 ways.

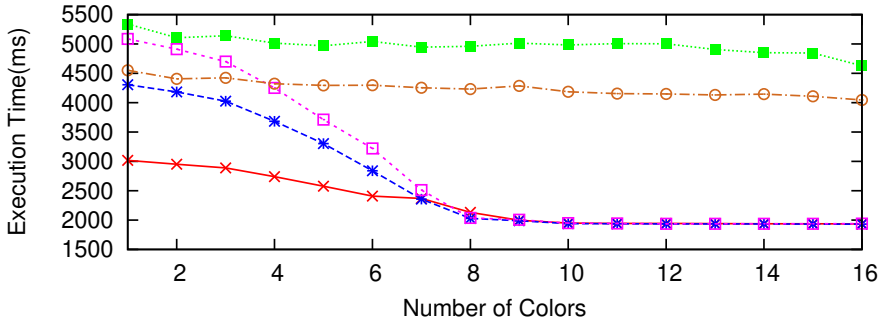
Figure 3.10: Execution-time data for the 256KB-WSS micro-benchmark program.



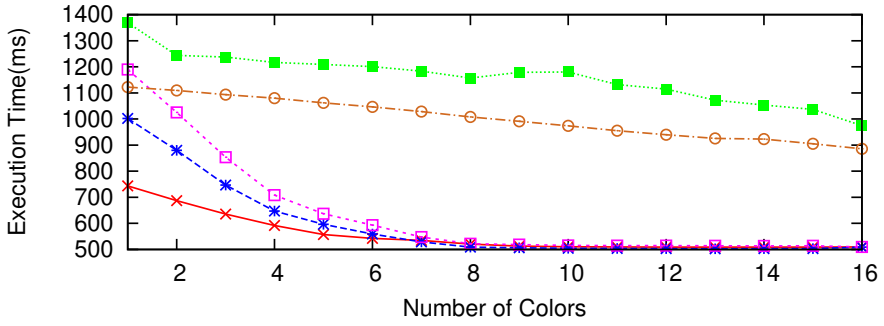
(a) WCET assuming 16 colors.



(b) ACET assuming 16 colors.



(c) WCET assuming 16 ways.



(d) ACET assuming 16 ways.

Figure 3.11: Execution-time data for the Matrix program.

hardware resources (*TLB, cache-related registers, memory bus, memory controllers—see Section 2.4.3*) *had only a small impact.*

In insets (a) and (c) of Figure 3.10, the WCET with LLC isolation becomes quite close to that in an ideal system at four colors, which yields an LLC area matching the micro-benchmark program's WSS, and therefore its ICAS. A similar trend can be seen in Figure 3.11 (a) at ten ways and in Figure 3.11 (c) at seven colors.

Observation 3.3. *Isolation with respect to both the LLC and DRAM banks improved WCETs over LLC isolation alone especially when the allocated LLC area is less than the ICAS.*

This effect can be seen in insets (a) and (c) of Figure 3.10 and in insets (a) and (c) of Figure 3.11. Note that, if the allocated LLC area is at least the given program's ICAS, then DRAM bank isolation has only a small impact. Recall from Section 2.1.4 that Level-A and -B tasks are included in the Level-C analysis, which assumes Level-C PETs for all tasks. Thus, ACETs are important to consider for all tasks.

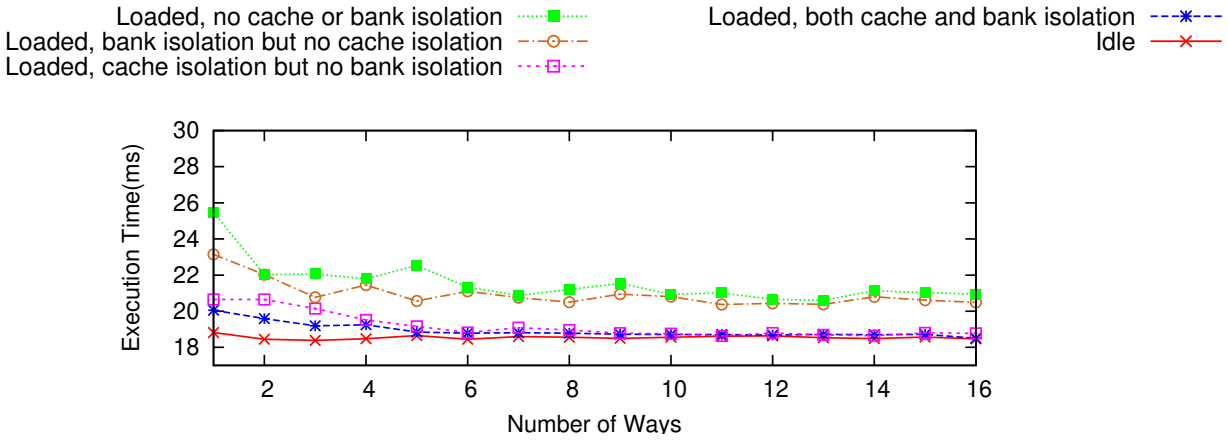
Observation 3.4. *The WCET trends noted in Observations 3.1–3.3 also apply to ACETs. ACETs were lower than WCETs by approximately 5-10% (respectively, 80%) for the micro-benchmark program (respectively, Matrix program).*

This can be seen by comparing insets (a) and (b) in Figure 3.10 and insets (a) and (b) in Figure 3.11. (The 5-10% reduction may be somewhat hard to see because of the scale.) The Matrix program exhibits a relatively lower ACET because it is less deterministic than the micro-benchmark program.

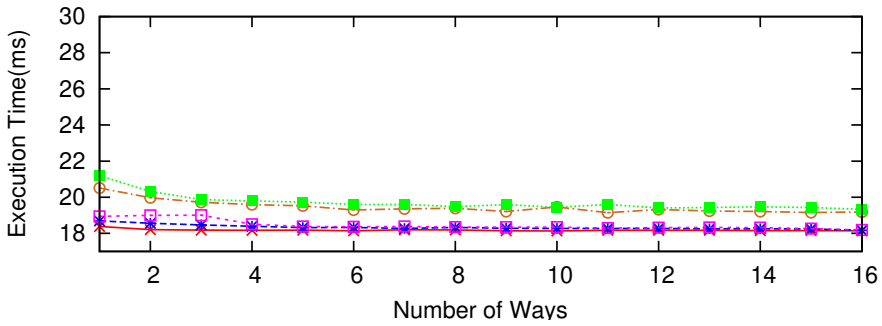
Observation 3.5. *The execution times of the 32KB-WSS micro-benchmark program were anomalously high for some allocated LLC area sizes.*

We originally expected that since this WSS matches the size of the L1 data cache, all memory references would hit in the L1 caches with the exception of compulsory misses. However, the results in insets (c) and (d) of Figure 3.12 do not support this hypothesis. Observe that, in these insets, some odd-numbered color allocations resulted in "spikes" in the execution times. The baseline execution time of $20ms$ corresponds to all references hitting in the L1 caches, as shown in insets (a) and (b) of Figure 3.12, so these spikes are due to L1 cache misses. To explain these spikes, we must carefully consider the cache structure.

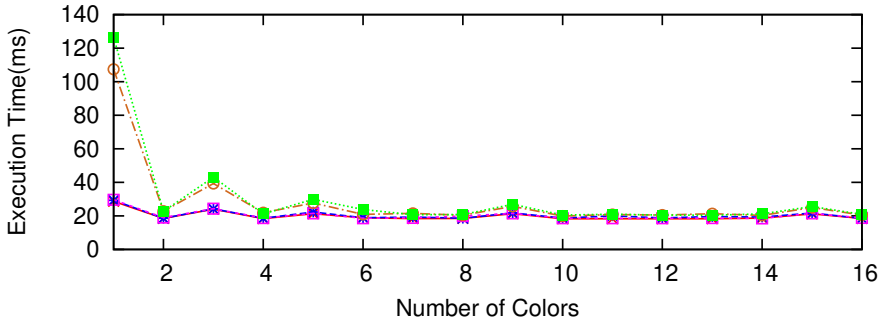
Our L1 data cache is a 32KB 4-way set-associative cache and has two colors, which we denote here as *X* and *Y* to distinguish them from the LLC colors. For each physical address, the least significant bit of its



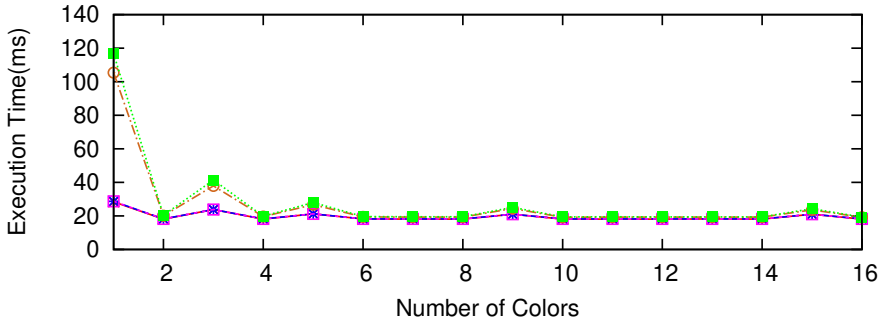
(a) WCET assuming 16 colors.



(b) ACET assuming 16 colors.



(c) WCET assuming 16 ways.



(d) ACET assuming 16 ways.

Figure 3.12: Execution-time data for the 32KB-WSS micro-benchmark program.

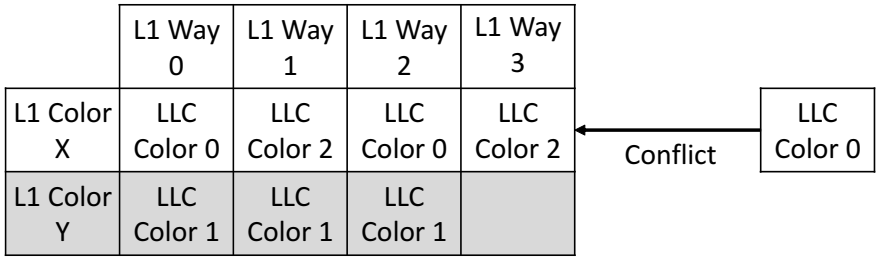
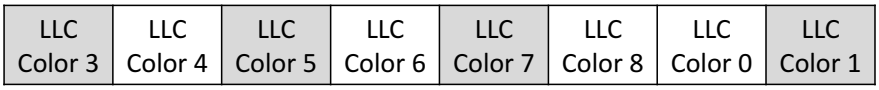
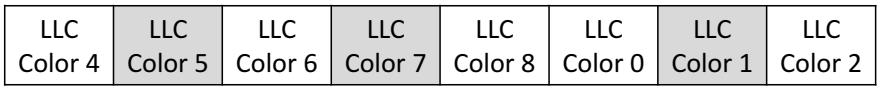


Figure 3.13: Conflict in the L1 cache.



(a) A case where the LLC color starts from 3.



(b) A case where the LLC color starts from 4.

Figure 3.14: Two potential mappings of eight data pages with nine LLC colors.

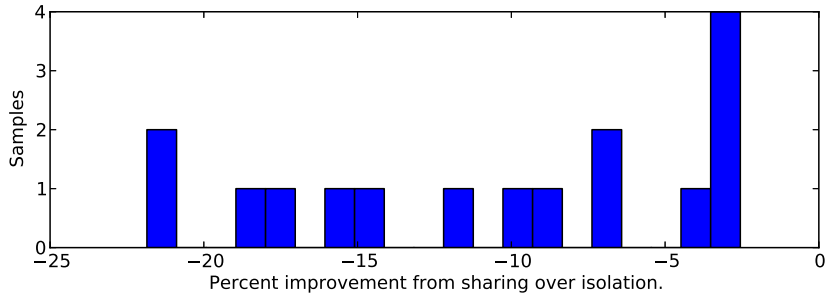
LLC color bits determines its L1-data-cache color. Even-numbered LLC colors are mapped to L1-Color X and odd-numbered LLC colors are mapped to L1-Color Y. The WSS of the considered micro-benchmark program is 32KB, so there are eight data pages to be colored. The spikes were observed when the required LLC colors were not equally distributed between the L1-Colors X and Y. For example, the eight pages of the micro-benchmark program could be allocated using the first three LLC colors as follows: three pages of Color 0, three pages of Color 1, and two pages of Color 2. In this case, as shown in Figure 3.13, five pages are mapped to the L1-Color X and three pages are mapped to the L1-Color Y. Since our L1 cache is only 4-way set associative, the five pages of L1-Color X compete for four ways, thereby causing evictions. This behavior was not observed for all odd-numbered color allocations.

For instance, suppose the micro-benchmark program is given an LLC area with Colors 0 through 8. Figure 3.14 shows two potential mappings of the program's eight pages to LLC colors. In Figure 3.14, white regions are mapped to L1-Color X while gray regions are mapped to L1-Color Y. In the first mapping, illustrated in Figure 3.14 (a), four pages are mapped to L1-Color X (denoted in white) and four pages are mapped to L1-Color Y (denoted in gray), and L1 evictions do not occur. In the second mapping, illustrated in Figure 3.14 (b), LLC-Color 3 is not used. As a result, there are five pages of L1-Color X and three pages

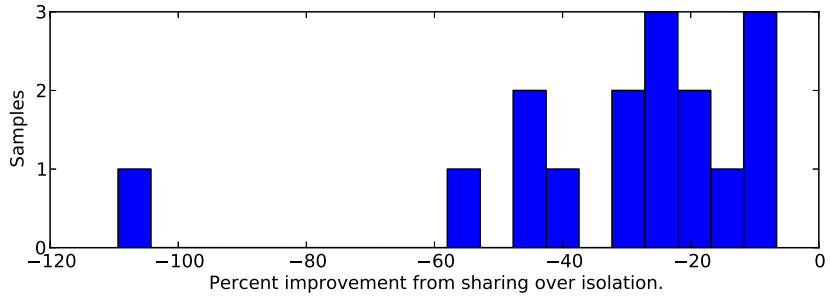
of L1-Color Y, which again causes evictions, explaining the execution-time spikes. The presence of these execution-time spikes where none was expected provides a cautionary note for designers concerned with provisioning systems. In particular, a complete understanding of the caching and memory characteristics of the considered platform is needed in order to avoid producing execution-time estimates that are problematic. With respect to the results discussed later in this dissertation, no special consideration of execution-time spikes was required because all of the allocation schemes we considered provide an even number of colors to all tasks.

3.4.2 Space Tradeoffs

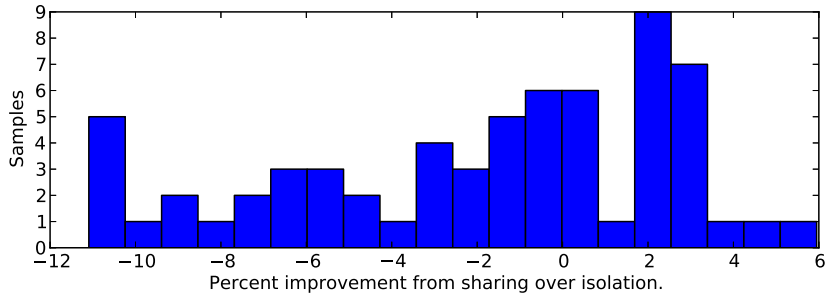
While the data discussed above shows clear execution-time benefits from isolation, this does not address the additional LLC space constraint imposed by partitioning. Here, we compare specific execution times associated with the different LLC allocation variants shown in Figure 3.4. For example, suppose we can allocate the entire LLC to Levels A and B under Variant 1 or 3. From the perspective of the micro-benchmark program considered in Figure 3.10 (c), these two allocation choices can be examined by comparing the curve for a loaded system with LLC and DRAM bank isolation at four colors to that for a loaded system with no LLC isolation, but DRAM bank isolation, at 16 colors. By comparing these two points, we see that the WCETs under these two allocations are $317ms$ and $665ms$, respectively, a 52% WCET improvement under isolation at Levels A and B. Similarly, suppose we can allocate the entire LLC to Level C under Variant 1 or 2. From the perspective of the micro-benchmark program considered in Figure 3.10 (b), these two allocation choices can be examined by comparing the curve for a loaded system with no LLC isolation at 16 ways to that for a loaded system with LLC isolation at four ways (we assume that bank isolation is not provided, as is the case for Level C). By comparing these two points, we see that the ACETs for these two options are $677ms$ and $466ms$, respectively, a 31% ACET improvement under isolation at Level C. To more clearly investigate the execution-time differences afforded by sharing versus isolation, we generated histograms, four of which are given in Figure 3.15. In Figure 3.15, the x-axis gives different percentages, and for a given percentage, the histogram shows the number of cases observed across all considered LLC-allocation sizes that exhibited that percentage improvement. Negative (respectively, positive) percentages indicate that isolation (respectively, sharing) produces lower execution times. WCETs are with respect to allocation variants (see Figure 3.4) at Levels A and B with bank isolation. ACETs are with respect to allocation variants at Level C without bank isolation.



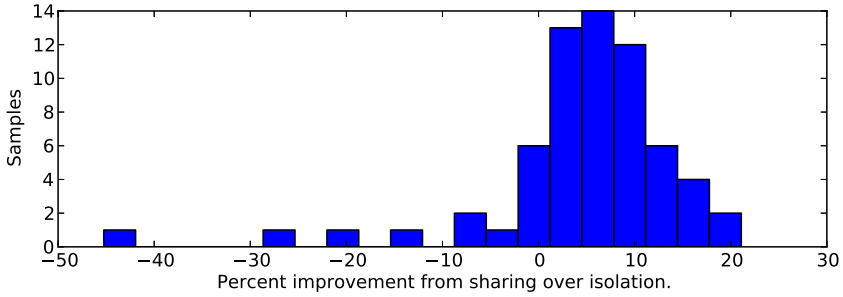
(a) Matrix WCETs.



(b) 256KB-WSS micro-benchmark WCETs.



(c) Matrix ACETs.



(d) 256KB-WSS micro-benchmark ACETs.

Figure 3.15: Histograms showing the percentage improvement in the ACETs and WCETs of 256KB-WSS micro-benchmark program and the Matrix program provided by sharing the program's allocated LLC area, instead of ensuring isolation.

Observation 3.6. *WCETs under per-processor partitioning at Levels A and B were almost always lower than WCETs under cross-core sharing of the available LLC space.*

Insets (a) and (b) of Figure 3.15 show that isolation improves WCETs in all cases for the Matrix and 256KB-WSS micro-benchmark programs. Similar results were found for other programs. Given that isolation is overwhelmingly preferable to sharing at Levels A and B, we chose not to consider Variant 3 of our general allocation strategy illustrated in Figure 3.4 (c) in the schedulability experiments presented later.[10]

Observation 3.7. *For a given LLC allocation, sharing an entire allocated LLC area at Level C without isolation and partitioning that area and providing isolation are incomparable with respect to ACETs. This exposes a tradeoff that is dependent upon the given task system to be supported, as well as the size and configuration of the LLC-allocation space under consideration.*

Figure 3.15 (c) shows that for the Matrix program, isolation tends to improve ACETs, while Figure 3.15 (d) shows that for the 256KB-WSS micro-benchmark program, the reverse is true. Also, there exist cases in which one approach may be substantially better, as demonstrated by the leftmost "outlier" in Figure 3.15 (d), where isolation yields a 45% improvement. Given this tradeoff, we opted to fully consider in the schedulability experiments presented later Variant 2 of our allocation strategy, shown in Figure 3.4 (b), in which the Level-C and OS LLC area is partitioned instead of shared.

3.4.3 Impact of Sharing at Level C: Additional Considerations

If hardware isolation is provided, then ACETs can be computed without much regard for the background workload. However, the situation is murkier for Level-C tasks assuming Variant 1 of the LLC-allocation strategy in Figure 3.4 since Level-C tasks share the same LLC area and DRAM banks. To better understand this issue, we conducted experiments in which a mixture of randomly selected DIS programs was executed at Level C and ACETs were determined for individual programs. The following observation follows from these experiments.

Observation 3.8. *Level-C ACETs for Level-C tasks increased when the allocated Level-C LLC area was reduced or when the utilization of the background Level-C workload was increased.*

[10] We considered all variants in Appendix A.

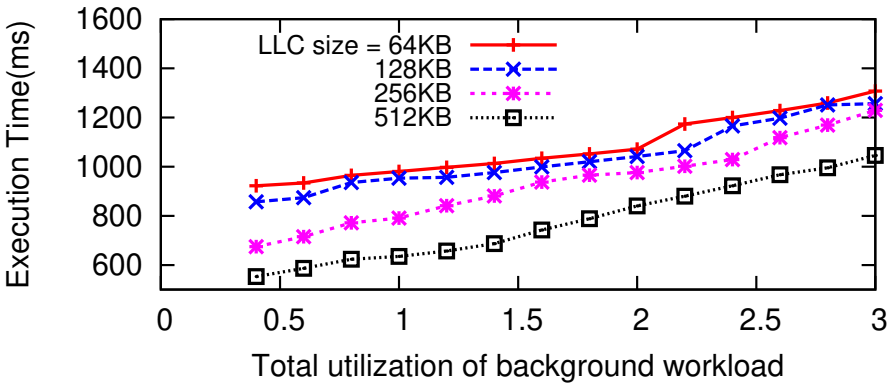


Figure 3.16: ACETs of the Matrix program with varying LLC sizes and background workloads.

Data supporting this observation is given in Figure 3.16, which gives ACETs for the Matrix program assuming various LLC area sizes and total background Level-C utilization. To determine Level-C PETs for Level-C tasks in practice, some domain knowledge would be required when defining an appropriate background workload. In our schedulability experiments, we determined such PETs by "indexing into" a graph similar to that in Figure 3.16, which is reflective of the assumption that the background workload is a mixture of DIS programs.

3.4.4 Impact of Coarse-Grained OS Isolation

Our MC^2 framework isolates the OS from Levels A and B with respect to the LLC and DRAM banks as discussed in Sections 3.2.2 and 3.3.4. As a result, execution within the OS does not cause any interference of Level-A and -B tasks with respect to the LLC or DRAM banks. We examined the impact of this feature by conducting an experiment in which a Level-B micro-benchmark program was executed with and without coarse-grained OS isolation. The micro-benchmark program was modified to invoke a dummy system call once per loop iteration that allocated and read 16 pages of memory. While such a system call may seem somewhat extreme, the point here is that if OS isolation is not provided, then predictability can be lost, unless the code paths the OS will take are known with high assurance. Figure 3.17 shows quartiles of measured execution times for 100 jobs of the micro-benchmark program, with and without OS isolation.

Observation 3.9. *OS isolation reduced the WCET and ACET of the micro-benchmark program considerably.*

The WCET (respectively, ACET) decreased by 24% (respectively, 16%) in Figure 3.17. The execution of the dummy system call can evict cache lines of 16 pages of the micro-benchmark program, which results in

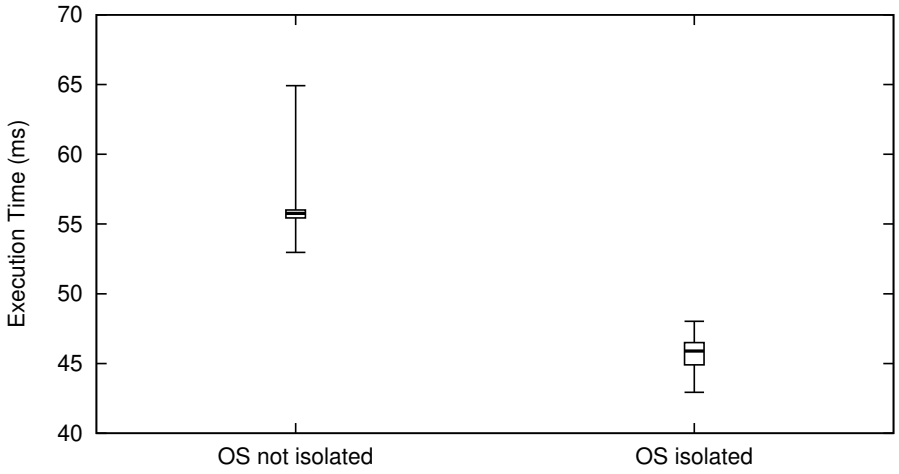


Figure 3.17: Execution times of a Level-B micro-benchmark program with and without coarse-grained OS isolation.

cache misses when user-mode execution resumes. OS isolation prevents evictions of user pages at Levels A and B in kernel mode since the OS only accesses the Level-C and OS partition.

Cross-core OS interference. In the experiment considered above, OS code executes on the same processor as the program invoking the system call. In addition to such same-core interference, the OS can also cause cross-core interference. In particular, because OS pages are spread over all cache colors, they can cause LLC evictions of tasks on any processor if not properly managed. To demonstrate the potential ill effects of cross-core OS interference, we executed two micro-benchmark programs concurrently at Level B on different processors. One program, referred to as the *caller*, issued dummy system calls after each loop iteration, while the other program, referred to as the *victim*, did not. The caller ran on CPU 0 while the victim ran on CPU 1. The two programs were allocated separate LLC partitions and DRAM banks. Recorded WCETs and ACETs for these programs, with and without coarse-grained OS isolation, are shown in Figure 3.18.

Observation 3.10. *OS isolation reduced the WCET and ACET of both the caller and victim by 20%.*

These results suggest that isolating user-space programs from each other with respect to the LLC and DRAM banks alone is not sufficient to mitigate execution-time interference; the OS must be isolated as well.

OS-related overheads can induce pessimism in schedulability analysis (Brandenburg, 2011), but Figures 3.17 and 3.18 suggest that per-overhead costs can be significantly reduced through OS isolation. One potential concern, however, is that restricting the OS to execute within a smaller LLC area might increase

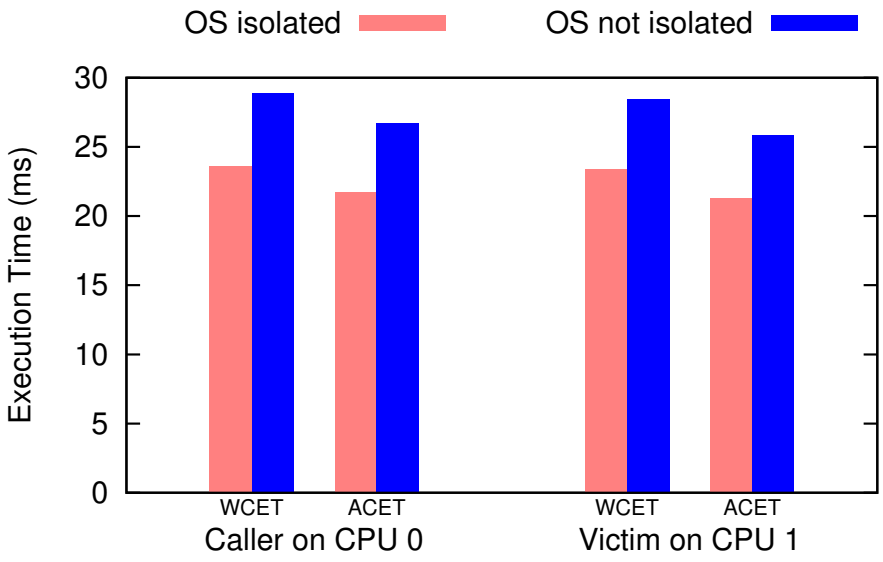


Figure 3.18: Cross-core OS interference.

its own execution times unacceptably. However, in additional experiments, we found that providing OS isolation increased system-call overheads by a mere $35ns$ in the worst case and by $15ns$ on average. Such small overheads are negligible in comparison with the noted reductions in WCETs and ACETs.

While our OS-isolation technique prevents the OS from evicting data from Level-A or -B tasks, it does not address the issue of data sharing or communication between a Level-A or -B task and the kernel itself. Consider, for example, the `read()` system call, in which the kernel copies data from kernel space into user space. When the kernel copies data into user-space pages, it will load the data into OS LLC ways, which are subject to interference from Level-C tasks and other kernel invocations. Consequently, such data cannot be assumed to be cached when subsequently accessed by the user-space task. Note, that when the user-space task reloads such data into the LLC, it will reside in the user-space ways, and therefore be isolated from further interference. We fully consider this issue and other issues that arise when tasks are permitted to share data or load data using system calls in Chapter 4. In Section 4.1, we considered a variant of MC^2 where data sharing is supported through shared memory pages. In Section 4.3, we discuss techniques to address the issue caused by executing system calls on behalf of Level-A and -B tasks and other issues that arise when Level-A and -B tasks perform I/O operations.

3.4.5 Optimizing LLC Partitions

In this section, we have provided recommendations for how to handle several tradeoffs such as tradeoffs between cross-core isolation and sharing in the LLC at Levels A and B, but other tradeoffs still require further analysis. For example, as seen in Figure 3.10, providing more LLC space to Level-A or -B tasks may reduce their execution times dramatically, but this may limit the available LLC space for Level C, which could increase Level-C execution times. How do we handle this tradeoff? Chisholm et al. (2015) showed that this tradeoff can be addressed by means of an optimization framework based on linear programming (LP). The LP-based optimization framework allows us to find an optimized LLC allocations for a specific task system. Although we have proposed hardware-management techniques and resource allocation strategies to minimize capacity loss, we still must provide evidence in support of the major thesis of this dissertation, namely, that combining mixed-criticality provisioning and hardware management is more effective in reducing capacity loss on multicore platforms than applying either technique alone. We provide such evidence in the form of a large-scale overhead-aware schedulability study in which the aforementioned optimization framework was applied.

3.5 Schedulability Study

In Section 3.4, we analyzed the effect of different hardware-management choices on the performance of individual benchmark and micro-benchmark programs. In this chapter, we analyze the impact of different hardware-management and scheduling choices on overall task-system schedulability. We conducted a large-scale overhead-aware schedulability study involving task systems randomly generated using a process based on our collected execution time data.

3.5.1 Overhead-Aware Schedulability Study

To quantify the gains afforded by criticality-aware isolation, we randomly generated millions of task systems and evaluated their schedulability with implementation-related overheads considered under the following scheduling- and resource-allocation schemes.[11]

[11] We measured implementation-related overheads of two schedulers, P-EDF and the MC^2 scheduler described in Section 3.3.2. We used the same measurement approaches presented in Brandenburg (2011).

- **MC^2-V1**: MC^2 under LLC-Allocation Variant 1 shown in Figure 3.4 (a) with DRAM-bank isolation at Levels A and B.
- **MC^2-V2**: MC^2 under LLC-Allocation Variant 2 shown in Figure 3.4 (b) with DRAM-bank isolation at Levels A and B. Differences in schedulability between MC^2-V1 and MC^2-V2 allow us to analyze the tradeoffs noted in Observation 3.7 at Level C.
- **MC^2**: MC^2 with no DRAM-bank or LLC isolation. This scheme provides the advantage of mixed-criticality analysis only.
- **PEDF-ISO**: P-EDF with DRAM-bank and LLC isolation. This scheme provides the advantage of hardware-management only, thus all tasks use Level-A PETs. P-EDF has been shown in previous work (Brandenburg, 2011) to be perhaps the most competitive known HRT scheduling algorithm for multiprocessors when considering implementation overheads.
- **PEDF**: P-EDF with no DRAM-bank or LLC isolation. This scheme provides neither mixed-criticality analysis nor hardware management.
- **EDF**: EDF on only one core. As stated in Chapter 1, this scheme represents the current industry best practice of disabling all but one core.

These six schemes allow us to independently investigate the gains afforded by isolation and mixed-criticality analysis, and fully quantify the value of combining both approaches.

Schedulability experimental framework. In our schedulability experiments, we assumed that PETs for each criticality level are defined as in Section 2.2. Task sets were randomly generated by using five uniform distributions to choose task and task-set parameters. The specific distributions used were selected from the per-distribution choices listed in Table 3.1. These distributions are defined with respect to the single-core EDF scheme. All combinations of these choices were considered. These distributions determine the criticality utilization ratio (*i.e.* the fraction of the overall utilization assigned to each criticality level), task periods, task utilizations, the maximum LLC reload time after a preemption or migration (specified as a fraction of overall task execution time), and per-task Level-A inflation factors (which are similar to those considered by Vestal (2007)).

At a high level, our overall experimental framework is as follows:

	Choice	Level A	Level B	Level C
Criticality Utilization Ratios	A-Heavy	[50, 70)	[10, 30)	[10, 30)
	B-Heavy	[10, 30)	[50, 70)	[10, 30)
	C-Heavy	[10, 30)	[10, 30)	[50, 70)
	AB-Moderate	[35, 45)	[35, 45)	[10, 30)
	AC-Moderate	[35, 45)	[10, 30)	[35, 45)
	BC-Moderate	[10, 30)	[35, 45)	[35, 45)
	All-Moderate	[35, 45)	[35, 45)	[35, 45)
Period (ms)	Short	$\{3, 6\}$	$\{6, 12\}$	[3, 33)
	Contrasting	$\{3, 6\}$	$\{96, 192\}$	[10, 100)
	Long	$\{48, 96\}$	$\{96, 192\}$	[50, 500)
Task Utilization	Light	[0.001, 0.03)	[0.001, 0.05)	[0.001, 0.1)
	Moderate	[0.02, 0.1)	[0.05, 0.2)	[0.1, 0.4)
	Heavy	[0.1, 0.3)	[0.2, 0.4)	[0.4, 0.6)
Max Reload Time	Light	[0.01, 0.1)	[0.01, 0.1)	[0.01, 0.1)
	Moderate	[0.1, 0.25)	[0.1, 0.25)	[0.1, 0.25)
	Heavy	[0.25, 0.5)	[0.25, 0.5)	[0.25, 0.5)
Level-A Inflation (%)	Constant	[0.5, 0.5)	[0.5, 0.5)	[0.5, 0.5)
	Small Variation	[0.3, 0.7)	[0.3, 0.7)	[0.3, 0.7)
	Large Variation	[0.1, 0.9)	[0.1, 0.9)	[0.1, 0.9)

Table 3.1: Task-set parameters and distributions.

Step 1 Select the specific five distributions used from among the choices listed in Table 3.1.

Step 2 Generate task and task-set parameters under the single-core EDF scheme using these distributions.

Step 3 Based on the generated EDF PETs, generate PETs for other isolation configurations and mixed-criticality provisioning assumptions (*e.g.*, PETs should be smaller in schemes that provide isolation compared to those that do not).

Step 4 Adjust task parameters to account for relevant overheads.

Step 5 Check the schedulability of the resulting task system under each considered scheme.

The adjustments to apply in the third and fourth steps were based upon measurement data. We collected 8GB of task execution-time data and 9GB of overhead data for the schedulability study.

In the third step, all PETs are defined based on EDF-scheme PETs, which correspond to A-inflated WCETs in an idle system with the full LLC allocated to the task in question. We denote this WCET parameters as C_i^0 for task τ_i. The C_i^0 values are obtained implicitly from the randomly generated tasks in the second step. All execution-time values used to obtain all other PETs for τ_i for different isolation and analysis assumptions

Exec. Time	WCET or ACET?	Idle or Load?	LLC Iso.?	DRAM Iso.?	LLC Area
C_i^0	A-infl. WCET	Idle	N/A	N/A	Entire LLC
C_i^1	A-infl. WCET	Load	Yes	Yes	Entire LLC
C_i^2	A-infl. WCET	Load	Yes	Yes	Any Area
C_i^3	A-infl. WCET	Load	Yes	No	Any Area
C_i^4	A-infl. WCET	Load	No	No	Any Area
C_i^5	WCET	Load	All Relevant Cases		Any Area
C_i^6	ACET	Load	Yes	Yes	Any Area
C_i^7	ACET	Load	Yes	No	Any Area
C_i^8	ACET	Load	No	No	Any Area

Table 3.2: Generated PET values.

Task Level	PET Level	MC^2-V1	MC^2-V2	MC^2	PEDF-ISO	PEDF	EDF
A	A/HRT	C_i^2	C_i^2	C_i^4	C_i^2	C_i^4	C_i^0
	B	C_i^5	C_i^5	C_i^5	N/A	N/A	N/A
	C	C_i^6	C_i^6	C_i^8	N/A	N/A	N/A
B	A/HRT	N/A	N/A	N/A	C_i^2	C_i^4	C_i^0
	B	C_i^5	C_i^5	C_i^5	N/A	N/A	N/A
	C	C_i^6	C_i^6	C_i^8	N/A	N/A	N/A
C	A/HRT	N/A	N/A	N/A	C_i^2	C_i^4	C_i^0
	B	N/A	N/A	N/A	N/A	N/A	N/A
	C	C_i^8	C_i^7	C_i^8	N/A	N/A	N/A

Table 3.3: Assignment of execution time parameters to PETs.

are listed in Table 3.2. Table 3.3 shows how these values are used to define all PETs under each scheme. The columns of Table 3.2 indicate how each execution-time value is defined (*i.e.*, whether the value is a Level-A-inflated WCET, a non-inflated WCET, or an ACET, whether the system is assumed to be under load or idle, *etc.*). Each of these values is generated by applying scaling factor(s) to the prior-listed execution-time values. We now present the details about how to generate PETs for each scheme here.

Step 3.1 Generate C_i^1 by scaling C_i^0 to account for interfering workload. We choose C_i^1 uniformly from $[120, 150)\%$ of C_i^0, based on WCET measurement data in idle and loaded systems with the full LLC allocation.

Step 3.2 Generate C_i^2 by scaling C_i^1 for different LLC allocations. Our C_i^0 values are defined from generated utilizations. The process for generating such utilizations was carefully defined to produce trends similar to those seen from measurement data. Since our C_i^1 values are simply scaled versions of our C_i^0 parameters, similar utilization trends will be seen when utilizations are defined in terms of C_i^1

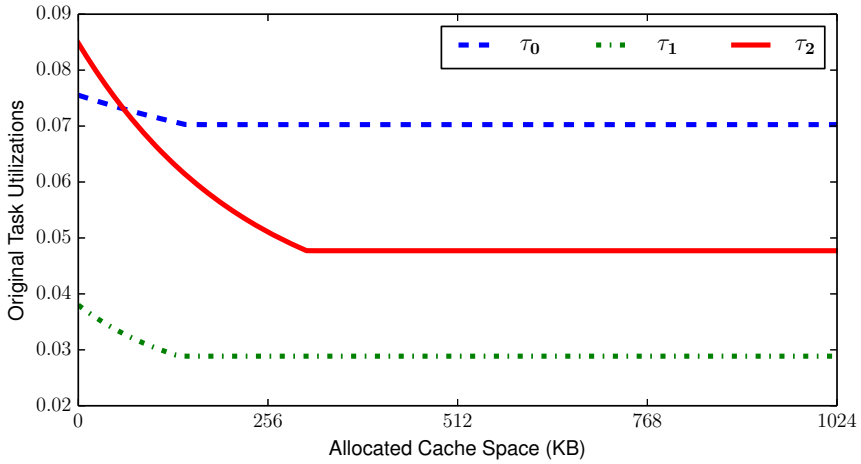


Figure 3.19: Utilizations generated under different LLC allocations for three example tasks.

values. Figure 3.19 illustrates typical generated utilizations. As seen in this figure, task utilizations monotonically decrease with increasing LLC space and converge at the ICAS. This is in accordance with Observation 3.2 on page 37. To reflect this, we obtain C_i^2 values for different LLC-allocation choices by applying a scaling factor to C_i^1 that exponentially increases with the minimum of the ICAS and LLC space. The actual scaling factors employed were selected to reflect measurement data.

Task ICASs were deduced using the Max Reload Time parameter in Table 3.1. The two both hinge on a task's cache footprint. Our Max Reload Time parameter was defined to reflect Observation 3.1, which showed that cache isolation can improve a task's WCET by up to 369%. For example, when the Light distribution is assumed, LLC isolation typically reduces WCETs by 20-50%, while when the Heavy distribution is assumed, the reduction is typically 200-500%. In addition, for all parameter combinations, tasks at Levels A and B tend to be more insensitive to LLC space than those at Level C. This reflects the underlying motivation for MC^2 that Level-A and -B tasks will tend to be rather deterministic fly-weight tasks and that Level-C tasks will tend to be more complex data-intensive tasks (see Section 2.2).

Step 3.3 Generate C_i^3 by scaling C_i^2 to account for shared DRAM banks. As seen in Figure 3.10, the impact of DRAM bank isolation on task execution times tended to range from imperceptible to 20% under small LLC allocations. Based on these results, we uniformly choose C_i^3 to be $[100, 130)\%$ of C_i^2 to account for the lack of bank isolation. Similar to the last step, this step is affected by the task ICAS and LLC allocation.

Step 3.4 Generate C_i^4 from C_i^3 based on known worst-case shared-cache behavior. When sharing a cache, cross-core interference may prevent a program from reusing any data in any shared cache blocks, thus eliminating any benefit from the LLC in the worst case. Therefore, we define C_i^4 to equal C_i^3 for the case when the allocated LLC space is zero.

Step 3.5 Generate all Level-B PETs from previously generated Level-A PETs. Using the Level-A Inflation factor in Table 3.1, all Level-B PETs can be computed from corresponding Level-A PETs. This gives us all C_i^5 values.

Step 3.6 Generate C_i^6 and C_i^7 to reflect expected ACET:WCET ratios under cache isolation and varying background workloads. ACET:WCET ratio trends will depend on the given background workload, *i.e.*, the total utilization of all competing tasks. Based on ACET:WCET ratio trends observed for benchmark and synthetic programs under different background workload utilizations, we identified an appropriate distribution from which to uniformly choose an ACET:WCET ratio for each task. For all tasks, these ratios were chosen uniformly among a range of percentages. For Level-C tasks, these ratios range over 20-40% for the lightest background workloads, and over 30-60% for the heaviest. For Level-A and -B tasks, these ratios range over 50-70% for the lightest background workloads, and 80-100% for the heaviest. This reflects our assumption that higher-criticality tasks tend to be more deterministic in their execution than Level-C tasks. Note that this process requires a means to calculate the Level-C utilization of a background workload, which is dependent on the ACETs that are generated. This entails an iterative process, such as that in Figure 3.20 (a). However, given the scale of our schedulability experiments, which involved millions of task systems, an iterative process was infeasible. As a result, we used the non-iterative process outlined in Figure 3.20 (b). We used the EDF-scheme utilization of the background workload as an upper bound on its average-case utilization.

Step 3.7 Generate C_i^8 to reflect differences between ACETs for a fully unmanaged system and ACETs for a cache-isolated system. From Figure 3.10 and Observation 3.8, we see that ACETs in an unmanaged cache gradually decline in a linear fashion as the allocated LLC space increases, even beyond the ICAS of the task. However, these ACETS generally remain higher than ACETs under cache isolation. When the LLC allocation is zero, both ACETs are the same, since LLC management does not affect tasks bypassing the LLC. To reflect this behavior, we generated C_i^8 as shown in Figure 3.21. On the right axis, we depict a scale showing the range of C_i^7's reduction in value as the allocated LLC space

Step 3.6.1a Assign all tasks Level-C PETs based on ACETs in an idle system.
Step 3.6.2a Based on assigned Level-C PETs, calculate the Level-C utilization of the background workload of each task.
Step 3.6.3a Assign new Level-C PETs based on ACETs under the background-workload utilizations calculated in Step 3.6.2a.
Step 3.6.4a Repeat Step 3.6.2a and 3.6.3a if any PETs increased in Step 3.6.3a.

(a) Iterative process.

Step 3.6.1b Based on the assigned EDF-scheme PET for each task, calculate the EDF-scheme utilization of the background workload of each task.
Step 3.6.2b Assign Level-C PETs based on ACETs under the background workload utilizations calculated in Step 3.6.1b.

(b) Non-iterative process.

Figure 3.20: Two methods for calculating ACETs in Step 3.6.

increases. On this scale, C_i^7 is at 0% reduction under zero allocated LLC space, and 100% under maximum allocated LLC space. C_i^8 at maximum allocated LLC space for the Matrix program would fall at approximately 50% on this scale. For each generated task, we choose a value from 30-70% on this scale for our generated C_i^8 at maximum LLC space. At zero allocated LLC space, C_i^8 matches C_i^7. For all other LLC allocation sizes, we interpolate values for C_i^8 linearly between values generated for zero allocated LLC space and maximum allocated LLC space.

From these steps, we now have all required PETs. We note once again that this process produces a *model* for producing PETs. As such, all claims resulting from our schedulability experiments apply only within the context provided by this model. Still, we have taken great pains to ensure that the range of PETs generated by this model encompass those that we have seen based on real measurement data, and that trends among related PETs for the same task correspond to those seen in our measurement data.

The distributions in Table 3.1 were defined to enable the systematic study of different factors impacting schedulability, such as mixed-criticality analysis, isolation, and overheads. We denote each combination of distribution choices using a tuple notation. For example, (C-Heavy, Long, Moderate, Heavy, Constant) denotes using the C-Heavy, Long, Moderate, *etc.*, distribution choices in Table 3.1. We call such a combination a *scenario*. We considered all possible such combinations, and for each one, we generated between 100 and

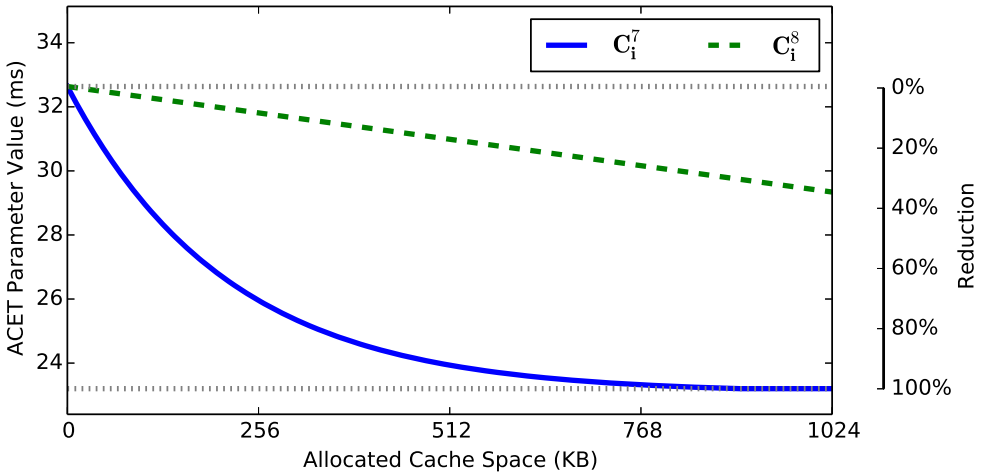


Figure 3.21: Comparison of C_i^7 and C_i^8 for a generated task.

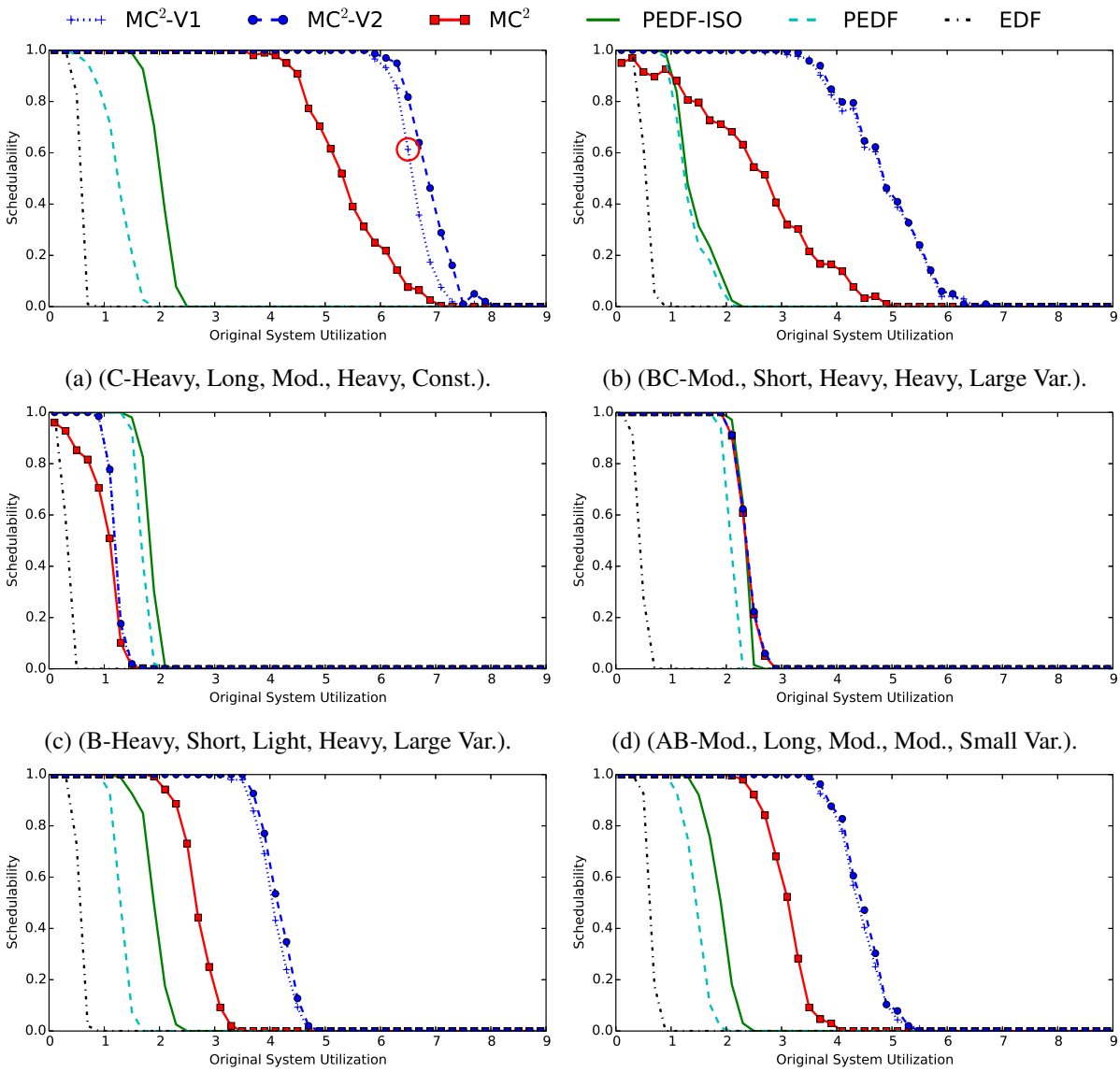
2,000 task sets, while ensuring that enough were generated to estimate the mean schedulability under a given combination to within ± 0.05 with 95% confidence.

For the schemes that support LLC isolation, we determined allocated LLC areas using the optimization methods in Chisholm et al. (2015) with small variations where necessary. Under the MC^2-V2 scheme, we divided the overall Level-C area into fourths (rounding as necessary) to give per-processor areas. For schemes requiring task partitioning, we used the worst-fit-decreasing bin-packing heuristic.

Schedulability results. In total, we evaluated the schedulability of approximately three million randomly generated task systems, which took roughly 18 CPU-days of computation. From this abundance of data, we generated over 500 schedulability plots, of which six representative plots are shown in Figure 3.22. The full set of plots is in Appendix A.

Each schedulability plot corresponds to a specific task-system category corresponding to a specific combination of the parameter distributions in Table 3.1. To understand how these plots are interpreted, consider Figure 3.22 (a). For this plot's task-system category, the circled point indicates that 61% of the generated task sets with EDF utilizations of 6.5 were schedulable under MC^2 with criticality-aware isolation. Note that, because the x-axis represents system utilizations under the single-core HRT EDF scheme, *it is possible under MC^2 to support systems with an EDF utilization exceeding four, as the mixed-criticality provisioning assumptions decrease PETs*.

We now state several observations that follow from the full set of collected schedulability data. We illustrate these observations using the data presented in Figure 3.22.



(a) (C-Heavy, Long, Mod., Heavy, Const.).

(b) (BC-Mod., Short, Heavy, Heavy, Large Var.).

(c) (B-Heavy, Short, Light, Heavy, Large Var.).

(d) (AB-Mod., Long, Mod., Mod., Small Var.).

(e) (B-Heavy, Long, Mod., Heavy, Const.).

(f) (A-Heavy, Long, Light, Mod., Const.).

Figure 3.22: Representative schedulability plots.

Observation 3.11. *MC² schemes were able to schedule at least one more processor's worth of utilization in comparison to the PEDF schemes in 64% of considered scenarios. Within this 64%, the MC² schemes were able to schedule 1.8 to 2.8 cores' worth of additional utilization on average.*

This observation is supported by insets (a), (b), (e) and (f) of Figure 3.22. The scenario in inset (a) corresponds to the industry-inspired motivation underlying the specification of MC² (Levels A and B would be mostly comprised of quite deterministic "fly-weight" tasks with rather low utilizations), as in this scenario, only tasks of rather light utilizations exist at Levels A and B. For some scenarios where loads are concentrated

at higher criticality levels, the MC^2 schemes also yielded substantial schedulability improvements. Inset (e) and (f) provide examples.

Observation 3.12. *LLC and DRAM-bank isolation allowed PEDF to only schedule 0.26 processors' worth of additional utilization on average. However, MC^2 was able to schedule 0.93 processors' worth of additional utilization on average under LLC and DRAM-bank isolation, sometimes scheduling two, and occasionally almost four, processors' worth of additional utilization.*

Insets (e) and (f) of Figure 3.22 give two examples of this for A-heavy and B-heavy task systems. In the MC^2-V1 scheme, higher-criticality tasks are afforded the benefits of the LLC while the unmanaged cases are not. However, isolation yields little improvement under PEDF. This is because, under PEDF-ISO, *all* tasks are assigned to processors and contend for LLC allocations, while in MC^2, Level-C tasks share LLC space apart from higher criticality levels. This shows that criticality-aware isolation can provide major schedulability benefits not seen in criticality-oblivious isolation.

Observation 3.13. *PEDF scheduled more task sets than MC^2 in 22% of the considered scenarios, particularly those most affected by overheads (short periods, light-utilization tasks, or both). In 9% of scenarios, schedulability under MC^2 and PEDF differed by less than 25%.*

Figure 3.22 (c) presents a scenario for which PEDF outperformed unmanaged MC^2 at certain utilizations, due to additional overheads for MC^2. Figure 3.22 (d) presents a scenario for which most schemes have nearly equivalent schedulability performance. However, for most scenarios, the benefits of mixed-criticality provisioning outweighed the disadvantages of additional overheads under MC^2.

Observation 3.14. *Adding LLC isolation for Level C in MC^2 resulted in less than 5% difference between MC^2-V1 and MC^2-V2 schedulability in 98% of the considered scenarios and resulted in a 3–7% gain in schedulability under MC^2-V2 in 10% of scenarios.*

In the specific scenarios in Figure 3.22, the impact of adding Level-C LLC isolation ranges from negligible (inset (c)) to slight (inset (e)) to moderate (inset (a)).

The schedulability study presented shows that, in many cases, MC^2 and hardware management significantly improve schedulability performance when combined. However, the conclusions drawn from this study are predicated on our provisioning assumptions, in particular the assumption that tasks can be reasonably provisioned based on *measured* WCETs and ACETs.

3.6 Summary

In this chapter, we presented a significant extension to the MC^2 framework that provides LLC and DRAM-bank isolation and that provides coarse-grained OS isolation from higher-criticality tasks. The MC^2 framework with hardware-management mechanisms is highly configurable and allows sharing and isolation tradeoffs to be studied in a criticality-aware way. We conducted extensive experiments in which the impact of the newly provided isolation mechanisms was assessed individually as well as collectively from a system-wide schedulability point of view.

CHAPTER 4: INTRODUCING SHARING IN MC^2 [12]

In Chapter 3, we showed that capacity loss can be significantly reduced when supporting real-time workloads on multicore platforms by combining hardware-management techniques and criticality-aware task provisioning. As in most other prior research on multicore hardware management, we considered a task system in which tasks do not share data with each other or any other entity because data sharing directly breaks the isolation techniques that underlie hardware management in MC^2. However, such a task system is not sufficient to support real-world workloads. In practice, tasks often communicate with other tasks or external devices to collect data. In this chapter, we examine the impact of introducing data sharing in MC^2 and extend our MC^2 framework to support different types of sharing while providing hardware isolation on multicore platforms.[13]

We begin by presenting techniques that support user-level data sharing between two tasks. We describe a new implementation of MC^2 that extends the prior one by allowing tasks to communicate through shared memory in Section 4.1. This new implementation allows producer/consumer buffers to be shared within a criticality level and wait-free buffers to be shared across criticality levels.

We then consider another common source of sharing, the usage of shared libraries. In previous sections, we assumed that all tasks are always statically linked to deal with shared libraries, meaning that all needed libraries are replicated on a per-task basis to eliminate any sharing. In the scheduliability studies that we conducted to evaluate the effects of hardware management in MC^2, this solution came for free because we did not consider memory to be a constrained resource when checking schedulability. In practice, however,

[12] Contents of this chapter previously appeared in preliminary form in the following papers:
Chisholm, M., Kim, N., Ward, B., Otterness, N., Anderson, J., and Smith, F. D. (2016). Reconciling the Tension Between Hardware Isolation and Data Sharing in Mixed-Criticality, Multicore Systems. In *Proceedings of the 37th IEEE International Real-Time Systems Symposium*, pages 57–68;
Kim, N., Chisholm, M., Otterness, N., Anderson, J., and Smith, F. D. (2017a). Allowing Shared Libraries While Supporting Hardware Isolation in Multicore Real-Time Systems. In *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 223–234;
Kim, N., Tang, S., Otterness, N., Anderson, J., Smith, F. D., and Porter, D. (2018). Supporting I/O and IPC via Fine-Grained OS Isolation for Mixed-Criticality Real-Time Tasks. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pages 191–201.

[13] The source code is available at `https://wiki.litmus-rt.org/litmus/Publications`.

the combined memory footprint of all tasks and the OS must fit within physical memory. Thus, the wasteful practice of fully replicating libraries can degrade schedulability significantly when memory is considered as a constrained resource. In Section 4.2, we propose an approach for supporting shared libraries in systems that ensure hardware isolation without degrading schedulability.

In Section 4.3, we address the sharing of data between the OS and user-level tasks that occurs when tasks invoke OS services for IPC or I/O. We investigate techniques for lessening the impact of OS-induced sharing by optimizing memory allocations. We also propose fine-grained OS isolation techniques that handle memory allocations issued by the kernel on behalf of Level-A or -B tasks, which resolves the coarse-grained OS isolation issue discussed in Section 3.3.4.

Finally, we conclude this chapter in Section 4.4.

4.1 User-Level Data Sharing

Even though we have shown that capacity loss can be reduced by combining hardware-management techniques and mixed-criticality provisioning, the ability to support real-world workloads has not been realized. A key reason is a lack of support for data sharing among tasks. As data sharing between two tasks directly breaks the isolation techniques provided in MC^2, we chose to disallow data sharing in Chapter 3. In this section, we examine the adverse impacts caused by data sharing and present methods for lessening them. We begin by presenting IPC mechanisms provided by the kernel to allow processes to communicate with each other on Linux.

4.1.1 IPCs

In order to transfer data between processes, Linux provides several IPC mechanisms to allow a user to coordinate shared data between processes. Linux processes can synchronize their actions and shared data via the IPC interfaces. When sharing data among processes, a proper synchronization protocol is required to ensure data integrity. Processes can share data via temporary files protected by locks. However, this approach has high overhead because it requires accessing the files on disk. To reduce the overheads incurred by disk accesses, the Linux kernel provides a set of system calls to support IPCs among processes without accessing the file system on the disk. While IPCs can be used for both synchronization and communication, we consider IPCs that are useful for data transfer in this section.

Pipe. A pipe is a unidirectional transfer mechanism that uses a byte stream buffer in the kernel. A pipe supports only sequential access of data and uses a pair of file descriptors that have no corresponding files on the disk. Since pipes use a kernel data buffer, for each pipe, the kernel creates an inode object and two file objects for reading and writing. Two processes can share data from a pipe via `read()` and `write()` system calls. The pipe is well suited for producer/consumer interactions between two processes, and it is not suitable for multiple readers/writers.

First in, first out (FIFO). The drawback of a pipe is that it is impossible to share data between two arbitrary processes in the system. Only the processes that have the same parent process can share a pipe because the file descriptors can be accessed only by related processes. To address such limitations, Linux introduces the FIFO, also called as *named pipe*, mechanism. Each FIFO has a file name in a file system, so the FIFO can be accessed by any processes in the system. FIFOs and pipes are almost identical but FIFOs support bidirectional communication.

Message queue. A message queue provides message-oriented communication where a receiver reads messages one at a time and partial reads of message are not allowed. Messages have priorities that decide the position of the message in the queue and a process can register to receive notification when a new message arrives. Unlike pipes, it is possible to share data with multiple readers and writers. Each message is created by a sender and it is sent to an *IPC message queue* in the kernel, where it remains until a receiver reads it.

Shared memory. The most useful IPC mechanism for data transfer is shared memory. The shared memory mechanism allows processes to share the same physical pages of memory. Therefore, communication among processes can be achieved by reading or writing the mapped address space. Each process that wants to share data requires adding the memory region that maps the shared page frames into its address space. This allows for lower-overhead IPC, as tasks may read and write buffers stored in shared pages without copying data into and out of kernel space while other mechanisms must invoke the kernel to read or write data, which results in two copies: from user space to the kernel and the kernel to user space. For this reason, we focus exclusively on shared-memory-based IPC mechanisms in this section. In Section 4.3, we consider the other IPCs that require kernel invocations.

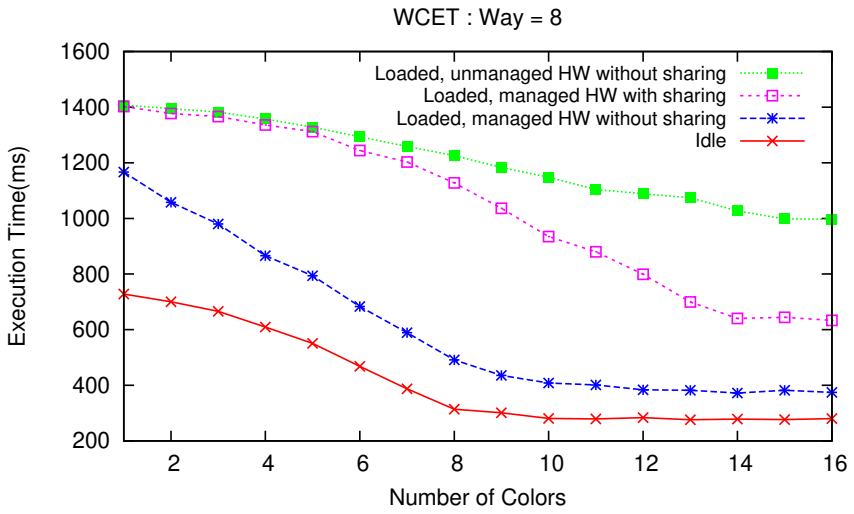


Figure 4.1: Measured WCET for the 256KB-WSS synthetic task considered in Chapter 3 assuming it is allocated eight LLC ways.

4.1.2 Problem Caused by User-Level Data Sharing

We augment the mixed-criticality task model described in Chapter 2 to allow any two tasks to communicate via buffers stored in shared memory. This modification explicitly breaks the isolation properties provided by the MC^2 implementation. For example, if two Level-B tasks assigned to different processors share a buffer, then the memory accesses of at least one of them cannot be entirely constrained to its assigned processor's DRAM bank. Also, memory accesses by one task can cause the other to experience LLC evictions.

To illustrate this lack of isolation, we conducted an experiment in which the WCET of a synthetic task with a WSS of 256KB was recorded under the MC^2 implementation described in Chapter 3 when executed either in an otherwise-*idle* system or in a *loaded* system that includes a stress-inducing background workload. We considered three loaded scenarios: no hardware management but also no data sharing, and hardware management both without and with data sharing. In the last scenario, each stress-inducing task was configured so that 12.5% of its memory accesses were to shared data. The synthetic task was allocated an LLC area consisting of eight ways and some number of colors. The obtained data, plotted in Figure 4.1, shows that with at least eight allocated colors and no data sharing, hardware management enabled a WCET near that of an idle system. However, the introduction of data sharing caused deterioration close to that of an unmanaged system.

4.1.3 Techniques for Mitigating Interference Due to Shared Memory

As seen in Figure 4.1, the introduction of data sharing can cause LLC and DRAM-bank interference. In this section, we propose to ameliorate such interference by applying the following three techniques in the sequence specified:

Selective LLC ByPass with Level-C DRAM Assignment (SBP): (i) Designate each buffer accessed by a Level-A or -B task as uncacheable and allocate it from the Level-C DRAM banks. (Note that such a buffer could be shared with a task at Level C.) This eliminates *unpredictable* LLC interference at Levels A and B, at the expense of higher average-case buffer access times due to the lack of caching. **(ii)** Designate each buffer shared exclusively by Level-C tasks as cacheable and allocate it from the Level-C DRAM banks. We consider the impacts of cross-core LLC and DRAM bank interference when provisioning Level-C tasks.

Concurrency Elimination (CE): If a buffer is shared by two tasks at Levels A and/or B, then assign both tasks to the same processor, and assign the buffer to that processor's designated DRAM bank and designate it as cacheable. (Since these techniques are being applied in sequence, this could "undo" a prior designation as uncacheable.) We call such a buffer a *core-local buffer*. This technique eliminates *concurrent* interference with respect to the considered buffer because a processor may only execute one task at a time. If a buffer is not core-local, we call it a *cross-core buffer*. Note that all buffers accessed by Level-C tasks are cross-core buffers because Level-C tasks can migrate across processors.

LLC Locking (CL): Permanently lock a buffer in the LLC to eliminate any DRAM-bank contention[14] or unintended LLC evictions. (As explained later, *portions* of a buffer actually can be locked.) This effectively reduces the LLC size for caching code and local data, exposing an interesting tradeoff that is explored in Section 4.1.7.

CL can be supported by using the lockdown registers mentioned in Chapter 2, using methods proposed by Mancuso et al. (2013), which entail prefetching the to-be-locked buffer into a way that is then permanently locked by all per-processor lockdown registers. Note that locked cache lines may still be read and written from any processor, just not evicted.

While it would be desirable to allow a cross-core buffer accessed by a Level-A or -B task to be *dynamically* cacheable, supporting this functionality using lockdown registers is not straightforward. For example, consider a buffer that is shared between Levels A and C. If that buffer were to be cached due to a reference by a

[14]Locked buffers are never evicted, so the write-back policy of our platform prevents any DRAM-bank contention for locked buffers.

Level-C task τ_i^C, then it would be cached in the Level-C LLC area, and thus could be potentially evicted by another Level-C task running on another processor. If that were to happen, followed by an immediate reference by a Level-A task, then the buffer would be cached in the Level-A LLC ways. This creates a situation wherein τ_i^C could potentially interfere with Level-A tasks by accessing Level-A ways. To avoid such situations, cross-core buffers accessed by Level-A or -B tasks are either permanently locked in the LLC or never cached in our framework.

4.1.4 Implementation

We implemented two shared-memory-based buffers: *producer/consumer buffers (PCBs)* and *wait-free buffers (WFBs)*. We use PCBs for data sharing among tasks of the same criticality and WFBs for data sharing among tasks of different criticalities. We assume wait-free sharing across levels so that multi-level blocking dependencies do not occur, which would greatly complicate mixed-criticality schedulability analysis. Each PCB or WFB is assumed to be written by one task and read by one task. For producer/consumer sharing, buffers must be replicated so that data can be produced without overwriting, and the consumer always reads the last-written value. For wait-free sharing, overwriting semantics is assumed, but buffer replication is still needed to ensure that read and write operations can occur without interfering with each other (Anderson and Holman, 2000).

Given the semantics of the considered WFBs proposed by Anderson and Holman (2000), read (respectively, write) operations must copy data to (respectively, from) such a buffer, *i.e.*, the data in the buffer cannot be accessed in place. A PCB can be accessed in place (without copying) if it is core-local, if it is entirely locked in the LLC,[15] or if it is only shared by Level-C tasks. All other PCBs must be copied. This is why cross-core buffers dealt with via SBP are stored in the Level-C banks. These copying rules support a major thesis underlying our implementation that requires the Level-A/B DRAM banks to be kept interference free.

User interfaces. We added a user-level interface to MC^2 where, via a character device, a task is able to request and map shared memory pages into its virtual address space. In our implementation, `mmap()` is used to specify the number of shared pages to map and their permissions, and two `ioctl()` commands are used

[15] As subsequently discussed, buffers may be partially locked in the LLC. A partially locked cross-core PCB shared with Levels A or B must be entirely copied, but part will be copied from the LLC.

to specify the DRAM bank and LLC colors of the mapped pages and specify the portions of a buffer that will be locked in the LLC.

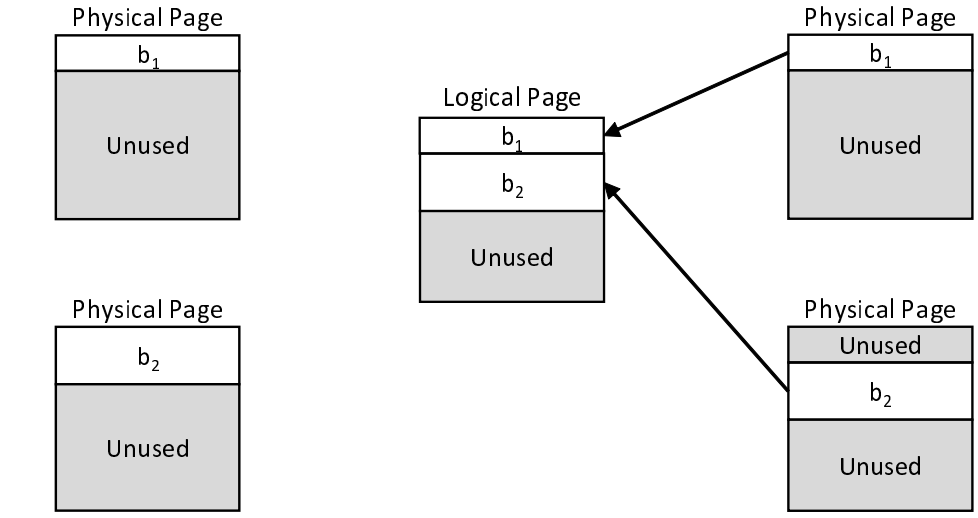
CL buffers. We support CL by locking *logical* pages into the LLC, where each such page occupies a single way. A logical page is defined by a set of *physical* pages of the same color. Note that same-colored physical pages will map to the same cache sets. We introduce the concept of logical pages because CL is actually applied at the granularity of *buffers*, not pages. To avoid unnecessarily wasting LLC space, which is a very constrained shared resource, we allow several such buffers to be stored in the same logical page by using an offset within a page. These buffers can even be allocated by different tasks. Clearly, storing buffers allocated by different tasks in the same physical page would be an egregious protection violation. To avoid this, we store them in different same-colored physical pages with appropriate offsets so that when they are mapped into the LLC, they map to disjoint cache sets.[16] The set of such pages is viewed as a single logical page. For example, consider two buffers, b_1 and b_2, allocated in different physical pages of the same color as illustrated in Figure 4.2. Without the concept of logical pages, both buffers start at the offset 0 in each physical page as shown in Figure 4.2 (a). Thus, we cannot lock both buffers in the LLC with a single way as they are mapped to the same cache sets. However, we can effectively use LLC space by introducing the concept of logical pages as illustrated in Figure 4.2 (b).

4.1.5 Micro-Benchmark Experiments

To compare SBP, CE, and CL, we ran micro-benchmark experiments in which a shared buffer was either read or written by a measured task. For SBP, we considered only the more-interesting option (i), *i.e.*, the considered page is designated as uncacheable. Under CE, we assumed the buffer was not locked in the LLC. To compare these techniques, we collected 10,000 samples for each choice of *buffer size* (2^NKB, $0 \leq N \leq 8$), *access type* (read or write), and *reference sequence* (random or sequential). For the purpose of measurement only, the measured task was instrumented to access the shared buffer in kernel space via a system call so that the `ktime_get()` API could be used to accurately measure execution times for small buffers.[17] Measurements were obtained in the presence of stress-inducing background tasks. Since we are

[16] Note that, with this implementation, if a misbehaving task accesses an address in a shared page that is external to the shared buffer it should be accessing (*e.g.*, via a buffer overflow), then this could cause a *temporal* fault for another task by evicting data in a cache line with the same color as the locked logical shared page. However, such a temporal fault cannot compromise logical correctness.

[17] In our new MC^2 implementation, tasks do not actually access buffers in kernel space. Such accesses were performed in kernel space in this experiment to enable more precise measurements.



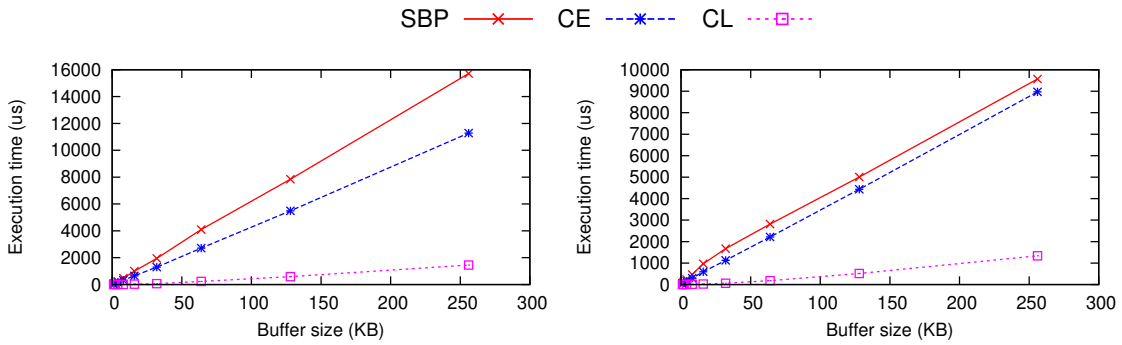
(a) Original page mapping. (b) Physical-to-logical page mapping.

Figure 4.2: Examples of page mappings for CL.

interested in systems in which the isolation techniques provided in MC^2 are employed, the background tasks were constrained to interfere only with the measured task's access of the shared buffer. The allowed interference depended on the technique being evaluated: the background tasks were configured to stress **(i)** the LLC and the specific DRAM bank used by the measured task under SBP; **(ii)** the LLC and the DRAM banks not used by the measured task under CE; and **(iii)** the non-locked LLC ways and the specific DRAM bank used by the measured task under CL. Figure 4.3 presents recorded WCETs and ACETs for accessing shared buffers.

Observation 4.1. *Buffer-accessing times were the lowest under CL and the highest under SBP. The WCET (respectively, ACET) for CL were typically 2 to 10% (respectively, 2 to 15%) of SBP accessing times. The WCET (respectively, ACET) for CE were typically 60 to 70% (respectively, 60 to 95%) of SBP accessing times.*

The worst-case CL accessing times were generally 2 to 3% of SBP accessing times for buffers that fit into the L1 cache, and nearer to 10% for other buffers. The average-case accessing times show similar trends. These results are expected, and are attributable to how the different techniques leverage resources within the memory hierarchy. Memory references will be satisfied from DRAM under SBP, from the LLC under CL, and from some combination of the two under CE, since a miss in the LLC causes a line of eight words to be cached.



(a) Measured WCETs for randomly writing.

(b) Measured ACETs for randomly writing.

(c) Measured WCETs for sequentially writing.

(d) Measured ACETs for sequentially writing.

(e) Measured WCETs for randomly reading.

(f) Measured ACETs for randomly reading.

(g) Measured WCETs for sequentially reading.

(h) Measured ACETs for sequentially reading.

Figure 4.3: Measured execution time data for accessing shared buffers.

4.1.6 Optimizations

The micro-benchmark results just described show that CL is clearly the technique of choice if it can be applied to a particular shared buffer. However, there is limited LLC space, so from a system-wide perspective, tradeoffs exist with respect to how the three techniques SBP, CE, and CL are applied. In this section, we show how such tradeoffs can be resolved. We begin by covering necessary additional background. Then, we describe a task-partitioning heuristic that resolves choices related to CE, and modifications to the cache-allocation optimization framework used in Chapter 3 that resolves choices related to CL.

Modeling buffers. As discussed in Chapter 2, we consider a set of implicit-deadline periodic tasks $\tau = \{\tau_1, \tau_2, \tau_3, ..., \tau_n\}$, split across Levels A–C in MC^2. Each task τ_i has a period, and PETs for its own criticality level and lower criticality levels.

To represent buffer sharing, we introduce a directed dependency graph, as illustrated in Figure 4.4. Each node represents a task, and each edge denotes a shared buffer and is directed from the (single) writer/producer of that buffer to the (single) reader/consumer. Each shared buffer is further specified by a *message size* (the amount of data read or written in one access), a *buffer size* (the message size times the number of message slots in the buffer—recall the discussion about buffer replication in Section 4.1.4), and its *type* (wait-free for cross-criticality sharing, and producer/consumer for same-criticality sharing). The introduction of producer/consumer buffers introduces precedence constraints (wait-free buffers do not—a read of a wait-free buffer simply obtains the most recently written value, whatever that value is). Because producer/consumer sharing occurs only within a criticality level, we have precedence constraints only within such a level. There is a considerable body of prior work on dealing with precedence constraints (Hsueh and Lin, 2001; Lakshmanan et al., 2010; Liu and Anderson, 2010, 2011; Saifullah et al., 2011), and it is usually assumed that the graph induced by such constraints is a directed acyclic graph (DAG). We assume that here. We also assume that all tasks in the same DAG have the same period.

Schedulability. Liu and Anderson (2010) have shown that tardiness bounds can be computed for a periodic or sporadic SRT task set with precedence constraints by converting to an "equivalent" independent task set that is then analyzed. Task utilizations are unaltered by this conversion. Since bounded tardiness is ensured at Level C by using utilization-based schedulability conditions (Mollison et al., 2010), Level-C precedence constraints can thus be supported with the same schedulability conditions as before.

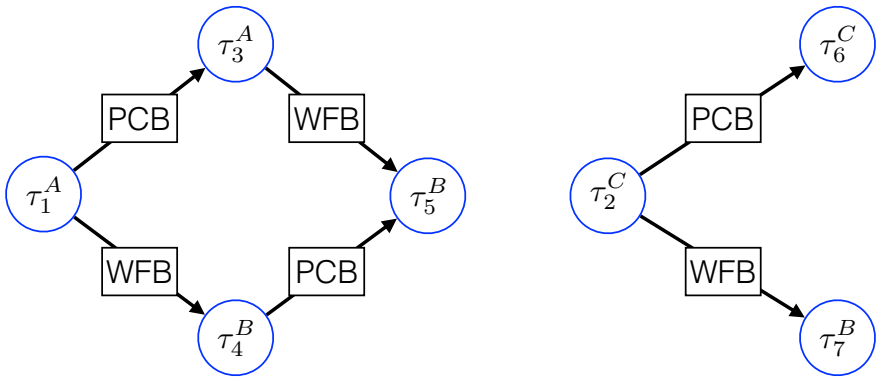


Figure 4.4: Example task system with PCBs and WFBs with tasks at different criticality levels (denoted by superscripts).

We handle precedence constraints at higher criticality levels by introducing release offsets to the task model that determine when a task initially commences execution. For example, in Figure 4.4, assuming the common period of τ_1^A and τ_3^A is 100 time units, τ_1^A and τ_3^A could be required to release their first jobs at times 0 and 100, respectively. The introduction of offsets does not impact the utilization of tasks, but can increase the end-to-end response time for a sequence of dependent jobs. We assume that any bounds that naturally follow from the use of offsets are acceptable provided individual tasks are schedulable. Since Level-A and -B conditions for checking task schedulability are utilization constraints (Mollison et al., 2010), these assumptions imply that we can also assess Level-A and -B schedulability with the same schedulability conditions as before.

With these assumptions, buffer sharing can impact schedulability only by increasing task execution times due to costs incurred in accessing buffers. We discuss this impact next assuming SBP, CE, and CL are applied. Note that, if shared buffers are not managed using any of our techniques, then execution times should be conservatively determined as in an unmanaged system, as the data depicted in Figure 4.1 suggests.

Execution-time impacts. We extended measurement-based methods used in Chapter 3 to deal with buffer sharing. We assume that each job consists of three phases. If a job copies a buffer it accesses, then this occurs in an initial *read phase*. The task then executes within an *execution phase* wherein only local data and isolated shared data are accessed. Finally, a *write phase* occurs, if buffers are being copied. Note that, according to the copying rules specified in Section 4.1.4, the read or write phases can be null if a buffer can be accessed in place. If either phase is non-null, then its execution cost can be determined via a measurement process. To take interference caused by unmanaged hardware resources into account, this process should

include a background workload that stresses the DRAM bank where the non-isolated data being copied to or from is stored.

The buffer copying rules ensure that the duration of a task's execution phase can be determined in the same manner as in Chapter 3, with one exception: applying CL can cause some data to be permanently locked in the LLC. It may seem surprising that marking a buffer as uncacheable by applying SBP has no impact. However, such a buffer must be a cross-core buffer shared by at least one task at Level A or B. (A buffer shared only at Level C is deemed cacheable by SBP, as is one shared at Levels A and B that is made core-local by CE.) The copying rules require a task to make a local copy of such a buffer before its execution phase.

As for the exception caused by CL, data locked in the LLC is guaranteed to be cache warm at all times, which can decrease the length of a task's execution phase. However, the actual benefits of locked data on a task's execution time are difficult to quantify because they depend on memory-access patterns. Therefore, we conservatively assess these benefits by only considering the time required to load such data once into the LLC. Specifically, we subtract this loading time from the execution time of a task's execution phase for any locked shared data that the task accesses.

Partitioning heuristics. To support CE, we used a two-step method proposed by Chisholm et al. (2016) to assign Level-A and -B tasks to processors. In the first step, a modified version of a greedy assignment heuristic proposed by Liu and Anderson (2011) is used that attempts to reduce schedulability-related data-sharing costs. The second step is applied only if the first step fails and attempts to find an assignment using the worst-fit decreasing heuristic, which more evenly balances utilization across processors, but is oblivious to data sharing. At a high level, Liu and Anderson's heuristic involves ranking shared buffers by utilization, where a buffer's utilization is based on the time to access it and the period of the accessing task. Modifications to this heuristic were required for our purposes because we have to take into account both Level-A and -B tasks and Level-A and -B PETs when applying the heuristic.

Optimizing buffer locking. We modified the prior optimization framework to optimize the size and use of the LLC locked-buffer area shown in Figure 4.5. Our new optimization framework determines how much space should be allocated to this area, and how much of each buffer should be locked into it as we allow buffers to be partially locked into the LLC. As a result, we added continuous variables to represent the

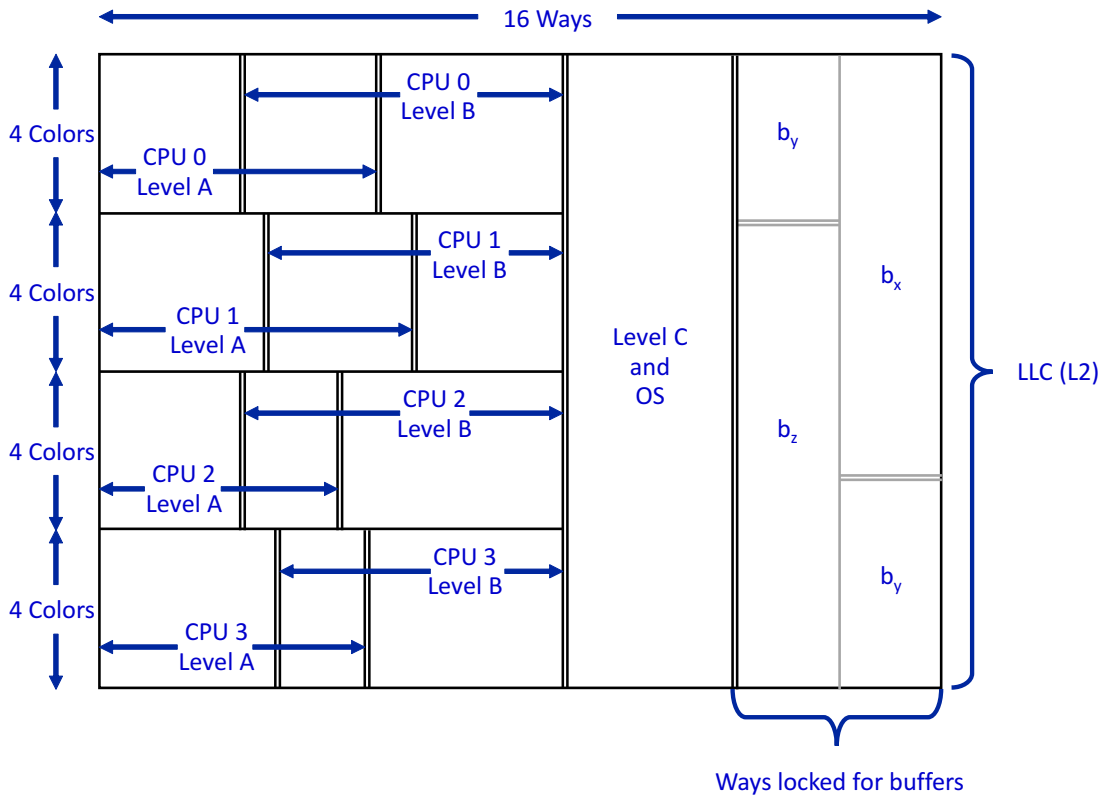


Figure 4.5: New LLC allocation that extends that in Figure 3.4 (a) by allowing several ways to be allocated for holding locked buffers. Several such buffers (b_x, b_y, b_z) are depicted.

amount of each buffer so locked and a new integer variable that represents the number of ways allocated to the locked-buffer area to the prior optimization framework. We also added execution times that must be considered pertaining to accessing shared buffers. These execution times can also be approximated linearly, as the micro-benchmark data shown in Figure 4.3 suggests.

4.1.7 Schedulability Study

To quantify the benefits of our shared-buffer management techniques, we conducted experiments in which the schedulability of millions of randomly generated task systems was assessed under several scheduling- and resource-management schemes. These schemes, which are summarized in Table 4.1, are as follows. Under U-EDF, all tasks are scheduled on a single processor using the P-EDF scheduler. This reflects current industry practice for eliminating shared-hardware interference by disabling all but one core. All other schemes use MC^2 scheduling and analysis. Under DSO (data-sharing oblivious), the MC^2 hardware-management techniques presented in Chapter 3 are used, but no special techniques are applied for shared buffers. In

Data Category	Benefit	U-EDF	DSO	SBP	SBP+CE	SBP+CE+CL	Ideal
Unshared at Levels A and B	CI	✓		✓	✓	✓	✓
	BI	✓		✓	✓	✓	✓
Shared within the same core	CI	✓			✓	✓	✓
	BI	✓			✓	✓	✓
	NC	✓	✓		✓	✓	✓
	CW					locked data only	✓
Shared with Level-A and -B tasks across cores	CI	N/A				locked data only	✓
	BI	N/A				locked data only	✓
	NC	N/A	✓			locked data only	✓
	CW	N/A				locked data only	✓
Shared at Level C only	CI	✓				locked data only	✓
	BI	✓				locked data only	✓
	CW					locked data only	✓

Table 4.1: Benefits accrued under different schemes. Benefits are cache isolation (CI), bank isolation (BI), cache warm (CW) during every access, and no copy (NC) phase required.

the next three schemes, the shared-buffer management techniques given in Section 4.1.3 are successively introduced: SBP introduces the SBP technique; SBP+CE then adds the CE technique, with task assignments being done via the assignment heuristic discussed in Section 4.1.6; finally, SBP+CE+CL adds the CL technique, where LLC locking decisions are made via the optimization algorithm discussed in Section 4.1.6. To upper bound the potential gains afforded by the use of our techniques, we also consider Ideal, in which an LLC of infinite capacity is assumed into which all shared buffers can be locked. (Code and local data are still assumed to be allocated assuming the actual LLC of finite size.)

Task-system generation. The task-system generation process we used extends that used in Chapter 3 by accounting for shared buffers. Task systems were randomly generated by using six uniform distributions to choose task and task-set parameters. The specific distributions used were selected from the per-distribution choices listed in Table 4.2. These distributions are defined with respect to the U-EDF scheme. All combinations of these choices were considered. These distributions determine the criticality utilization ratio (*i.e.* the fraction of the overall utilization assigned to each criticality level), task periods, task utilizations, the maximum LLC reload time after a preemption or migration (specified as a fraction of overall task execution time), and the maximum percent of a task's working set (WS)—the set of addresses used to reference data—dedicated to reading and writing shared buffers. For provisioning Level-A PETs, we applied an inflation factor of 50% to Level-B PETs. At a high level, our overall experimental framework is as follows:

Category	Choice	Level A	Level B	Level C
1: Criticality Utilization Ratios	A-Heavy	[50, 70)	[10, 30)	[10, 30)
	B-Heavy	[10, 30)	[50, 70)	[10, 30)
	C-Heavy	[10, 30)	[10, 30)	[50, 70)
	AB-Moderate	[35, 45)	[35, 45)	[10, 30)
	AC-Moderate	[35, 45)	[10, 30)	[35, 45)
	BC-Moderate	[10, 30)	[35, 45)	[35, 45)
	All-Moderate	[35, 45)	[35, 45)	[35, 45)
2: Period (ms)	Short	{3, 6}	{6, 12}	[3, 33)
	Contrasting	{3, 6}	{96, 192}	[10, 100)
	Long	{48, 96}	{96, 192}	[50, 500)
3: Task Utilization	Light	[0.001, 0.03)	[0.001, 0.05)	[0.001, 0.1)
	Moderate	[0.02, 0.1)	[0.05, 0.2)	[0.1, 0.4)
	Heavy	[0.1, 0.3)	[0.2, 0.4)	[0.4, 0.6)
4: Max Reload Time	Light	[0.01, 0.1)	[0.01, 0.1)	[0.01, 0.1)
	Moderate	[0.1, 0.25)	[0.1, 0.25)	[0.1, 0.25)
	Heavy	[0.25, 0.5)	[0.25, 0.5)	[0.25, 0.5)
5: Max % of WS Shared	Light	[0.001, 0.01)	[0.01, 0.1)	[0.05, 0.1)
	Heavy	[0.15, 0.3)	[0.2, 0.3)	[0.2, 0.7)
6: Tasks Per Graph Component		Task Count (levels not relevant below)		
	Small	{1, 2, 3, 4, 5}		
	Large	{11, 12, 13, 14, 15}		

Table 4.2: Task-set parameters and distributions.

Step 1 Select six specific distributions from among the distribution categories listed in Table 4.2.

Step 2 Using the selected distributions from the first four categories, generate task-set parameters under U-EDF.

Step 3 Based on the generated U-EDF PETs, generate PETs for the other schemes in Table 4.1. This process is informed by micro-benchmark data concerning task execution times, as discussed at length in Chapter 3. We augment this process by also considering buffer access times, as discussed in Section 4.1.6.

Step 4 Adjust the generated task parameters to account for relevant overheads as described in Chapter 3. The actual overhead values applied are based on measurement data.

Step 5 Generate a task dependency graph consisting of a collection of DAGs at each criticality level. The distribution selected from Category 6 in Table 4.2 determines the number of tasks in each DAG of the

graph. Techniques presented by Elliott et al. (2014) were used to ensure that a wide range of DAG topologies were generated.

Step 6 Assign wait-free dependencies to cross-criticality task pairs. To keep the parameter space from further exploding, we simply assumed that 1/6 of all dependencies were across criticalities. This reflects the hypothesis that cross-criticality sharing is less common.

Step 7 Generate an upper bound on the fraction of each task's WS that is shared using the distribution selected from Category 5. We determined actual buffer sizes subject to this upper bound to ensure that the buffer sizes are reasonable given the properties of the tasks that access them.

Step 8 Check the schedulability of the resulting task system under each considered scheme in Table 4.1.

We considered all possible combinations presented in Table 4.2, and for each utilization in each scenario, we generated enough task sets to estimate mean schedulability to within ± 0.05 with 95% confidence with at least 100 and at most 2,000 task systems.

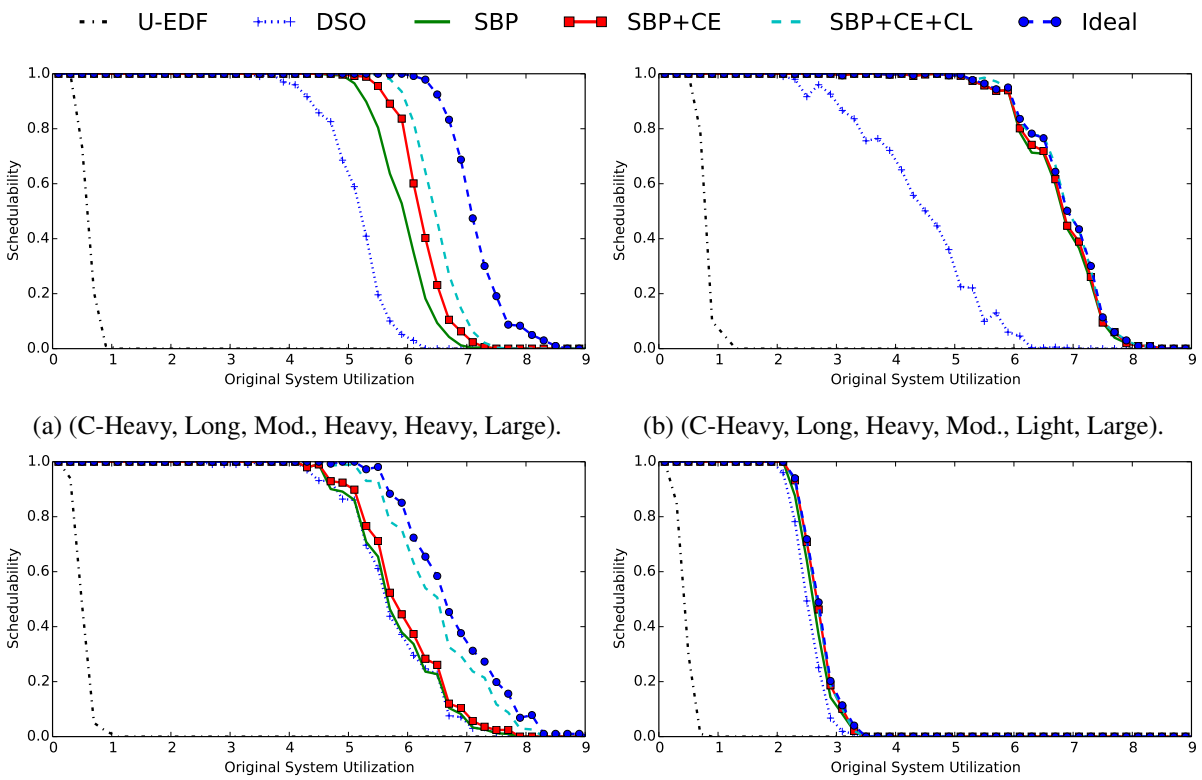
For schemes that do not lock buffers in the LLC, we used the LLC allocations illustrated in Figure 3.4 (a). For schemes not using our sharing-aware task-partitioning heuristic, we used the worst-fit-decreasing bin-packing heuristic.

Schedulability results. In total, we evaluated the schedulability of approximately nine million randomly generated task sets, which took roughly 27 CPU-days of computation. From this abundance of data, we generated over 700 schedulability plots, of which three representative plots are shown in Figure 4.6. The full set of plots is available in Appendix B.

We now state several observations that follow from the full set of collected schedulability data. We illustrate these observations using the data presented in Figure 4.6.

Observation 4.2. *SBP provided moderate schedulability benefits in approximately 60% of the considered scenarios. Moreover, in approximately 30% of cases, SBP provided schedulability near to that of Ideal.*

This observation is supported by insets (a) and (b) of Figure 4.6. For small-message-size scenarios, as in inset (b), copy-phases have little to no impact on task PETs. As a result, the isolation provided for local data under SBP eliminates the majority of sharing-related schedulability losses without significant schedulability



(a) (C-Heavy, Long, Mod., Heavy, Heavy, Large).

(b) (C-Heavy, Long, Heavy, Mod., Light, Large).

(c) (C-Heavy, Short, Heavy, Heavy, Heavy, Large).

(d) (A-Heavy, Contrast, Mod., Light, Heavy, Large).

Figure 4.6: Representative schedulability plots.

loss due to bypassing the cache or copying. For large-message-size scenarios, as in inset (a), greater copying overheads reduce the benefits of SBP. It is in these scenarios where CE and CL can be most beneficial.

Observation 4.3. *CE and CL provided mild improvements to schedulability in roughly 20% of considered scenarios.*

Figure 4.6 (a) gives one example of these improvements. In a majority of scenarios, copy-phase lengths are relatively small compared to execution-phase lengths, thus eliminating the need for techniques that primarily improve performance by reducing copying (CE and CL). In such scenarios, as in inset (b), the isolation provided by SBP is sufficient to achieve near-Ideal platform utilization.

Observation 4.4. *Across all considered scenarios, our combined techniques (SBP+CE+CL) provide 17% better schedulability on average than DSO. Moreover, this represents 84% of the improvement possible when compared to Ideal.*

Figure 4.6 (c) shows the benefits of our combined techniques (SBP+CE+CL). We observed similar trends in large-message-size scenarios.

Observation 4.5. *In roughly 40% of scenarios, schedulability under all MC^2 schemes, including DSO, was nearly equivalent.*

This observation is supported by Figure 4.6 (d). In scenarios with light WSs, there is little impact from cache use or shared-data copying. In such scenarios, breaking isolation in the LLC has little effect on task PETs. Similarly, copy-phase lengths have small impacts on PETs. As a result, all MC^2 configurations performed similarly in these scenarios.

Given the nature of our study, the observations above naturally hinge on our choice of hardware platform. The consideration of other platforms with different caching and memory characteristics might yield different conclusions. The Cortex-A9 platform used in this dissertation has a DRAM-to-cache latency ratio of 4-to-1. However, on many other platforms, the latency ratio may be an order of magnitude greater, which would dramatically increase read and write times for uncacheable buffers, and dramatically increase the effect of isolated cache space on PETs for Level-A and -B tasks. Platforms with longer DRAM bank access times will likely produce a much greater contrast in schedulability results between all MC^2 schemes in many of the considered scenarios. We may consider a dedicated DRAM bank only for shared data. However, the opportunity of a dedicated bank is not feasible on the hardware platform considered in this dissertation. Each processor requires one DRAM bank for Level-A and -B tasks, and the OS occupies more than two DRAM banks. Due to the limited number of available DRAM banks on our platform, we cannot reserve a DRAM bank for shared data.

4.2 Shared Libraries

In Section 4.1, we considered the implications of sharing as caused by the usage of shared data buffers. In this section, we consider another common source of sharing, the usage of shared libraries. Such sharing can be obviated by statically linked libraries, but this solution can degrade schedulability by exhausting memory capacity. In previous sections, we did not consider memory as a constrained resource. As a result, we assumed that tasks were always statically linked, and this solution did not affect previous schedulability studies. However, we must consider memory as a constrained resource when checking schedulability. The

memory footprint of all tasks and the OS must fit within physical memory in order to avoid unpredictable latency caused by handling page faults when executing real-time tasks.

We begin by describing static and shared libraries. We then present our solution to support shared libraries while ensuring hardware isolation. Finally, we consider the impact of introducing memory constraints and present a schedulability study to evaluate our proposed approach.

4.2.1 Libraries and Linking

In general, programs using both shared and non-shared libraries will not contain definitions of library-provided functions and data in their source code, but will instead refer to them by name. In a post-compilation procedure known as *linking*, function and variable names are then used to locate library-provided implementations, which are then used to produce the fully functioning final program. The value of a library lies in this reuse of the pre-implemented behavior. When a program invokes a library-provided function, it gains the behavior implemented inside that library without having to implement that behavior itself. Using libraries in this way can eliminate the need to recompile commonly used functions, as library code can be compiled once and left unchanged as a program is developed. The standard library for the C programming language is a classic example of a library in widespread use, and contains commonly used functions including `printf` and `strnlen`.

In Linux and other modern operating systems, libraries containing compiled code may be either *static libraries* or *shared libraries*. Static and shared libraries differ in both how they are allocated in memory and the time at which linking occurs.

Static libraries. Compiled code and data from static libraries are merged into a complete executable program file in a process called *static linking*. The advantage of static linking comes from its simplicity. Since all code is contained in a single executable file, they are easy to distribute and install and never experience dependency issues. On systems such as Linux, this means that page frames in physical memory are needed for both the base program and any content copied from static libraries. If the same library (*e.g.*, the C standard library) is used by many or all programs, *many copies of the library code are created, increasing physical memory consumption*.

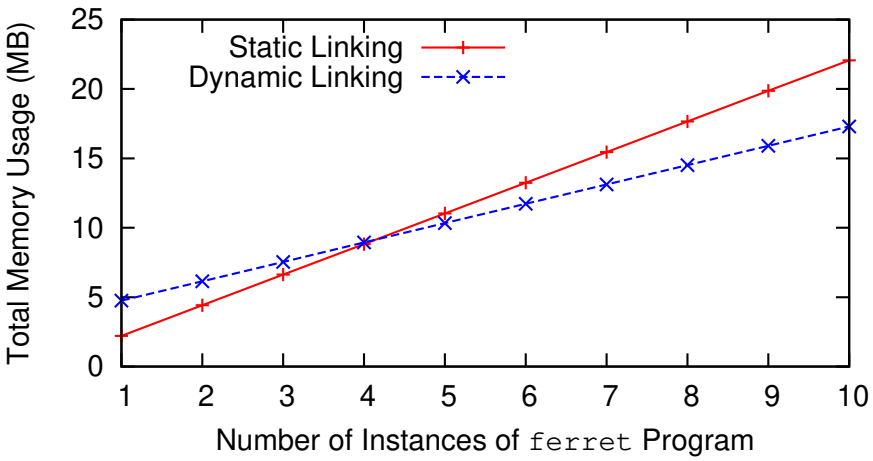


Figure 4.7: Memory usage comparison of static linking and dynamic linking.

Shared libraries. References to shared libraries are resolved using *dynamic linking*. Dynamic linking occurs either at load time, before a task (*i.e.*, process) begins execution, or at runtime when a library is first referenced. Dynamic linking requires the shared library to be mapped into the task's virtual address space. With dynamic linking, it is not known in advance which routine will be invoked by programs so the entire library is loaded into memory, which ostensibly occupies more memory than static linking. However, dynamic linking enables many programs to share a single copy of a shared library, so it reduces memory requirements of the overall system. This is one of the greatest advantages of dynamic linking.

Memory usage comparison. Even though tasks using static libraries may individually have smaller memory footprints, in aggregate the use of dynamic libraries typically saves memory. To illustrate the potential memory savings provided by shared libraries, we will refer to Figures 4.7 and 4.8. Figure 4.7 shows total memory usage of multiple instances of the `ferret` program from the PARSEC benchmark suite (Bienia, 2011), and Figure 4.8 shows memory regions reported by the `pmap` tool for the `ferret` program. The first column shows the virtual address of each allocated memory region, the second column shows the size of the allocated virtual memory in KB, and the third column shows the size of the allocated physical memory, called the *resident set size* (*RSS*), in KB. The fourth column shows access permissions: read, write, and execute. The last column indicates a file name for file-backed mapping, `[ anon ]` for non-file-backed ("anonymous") allocations, or `[ stack ]` for the program stack. Figure 4.8 (a) shows the simpler memory map for a statically linked version of `ferret`, and Figure 4.8 (b) shows the memory map when shared libraries are used instead. As seen in Figure 4.7, if five or more instances of the `ferret`

Address	KB	RSS	Mode	Mapping
00008000	960	960	r-x	ferret
000f8000	12	12	rw-	ferret
000fb000	500	500	rw-	[anon]
76f12000	528	528	rw-	[anon]
7ee0c000	620	256	rw-	[stack]
7efc1000	4	4	r-x	[anon]
ffff0000	4	0	r-x	[anon]
total	2628	2260		

(a) Static linking.

Address	KB	RSS	Mode	Mapping
00008000	136	136	r-x	ferret*
00031000	4	4	r--	ferret
00032000	4	4	rw-	ferret
00033000	488	488	rw-	[anon]
76b49000	520	520	rw-	[anon]
76bcb000	872	872	r-x	libc-2.19.so*
76cac000	8	8	r--	libc-2.19.so
76cae000	4	4	rw-	libc-2.19.so
76caf000	12	12	rw-	[anon]
76cb2000	100	100	r-x	libgcc_s.so.1*
76cd2000	4	4	rw-	libgcc_s.so.1
76cd3000	396	396	r-x	libm-2.19.so*
76d3d000	4	4	r--	libm-2.19.so
76d3e000	4	4	rw-	libm-2.19.so
76d3f000	296	296	r-x	libjpeg.so.7.0.0*
76d90000	4	4	r--	libjpeg.so.7.0.0
76d91000	4	4	rw-	libjpeg.so.7.0.0
76d9e000	152	152	r-x	libgslcblas.so.0.0.0*
76dcb000	4	4	r--	libgslcblas.so.0.0.0
76dcc000	4	4	rw-	libgslcblas.so.0.0.0
76dcd000	1396	1396	r-x	libgsl.so.0.17.0*
76f31000	8	8	r--	libgsl.so.0.17.0
76f33000	56	56	rw-	libgsl.so.0.17.0
76f41000	92	92	r-x	ld-2.19.so*
76f59000	28	28	rw-	[anon]
76f60000	4	4	r--	ld-2.19.so
76f61000	4	4	rw-	ld-2.19.so
7ecce000	620	256	rw-	[stack]
7eecf000	4	4	r-x	[anon]
ffff0000	4	0	r-x	[anon]
total	5236	4868		

* This mapping can be shared by several tasks.

(b) Dynamic linking.

Figure 4.8: Task memory map with **(a)** static vs. **(b)** dynamic linking.

program are running in a system, then dynamic linking saves memory by sharing the libraries listed in Figure 4.8 (b).

4.2.2 Techniques to Allow Shared Libraries

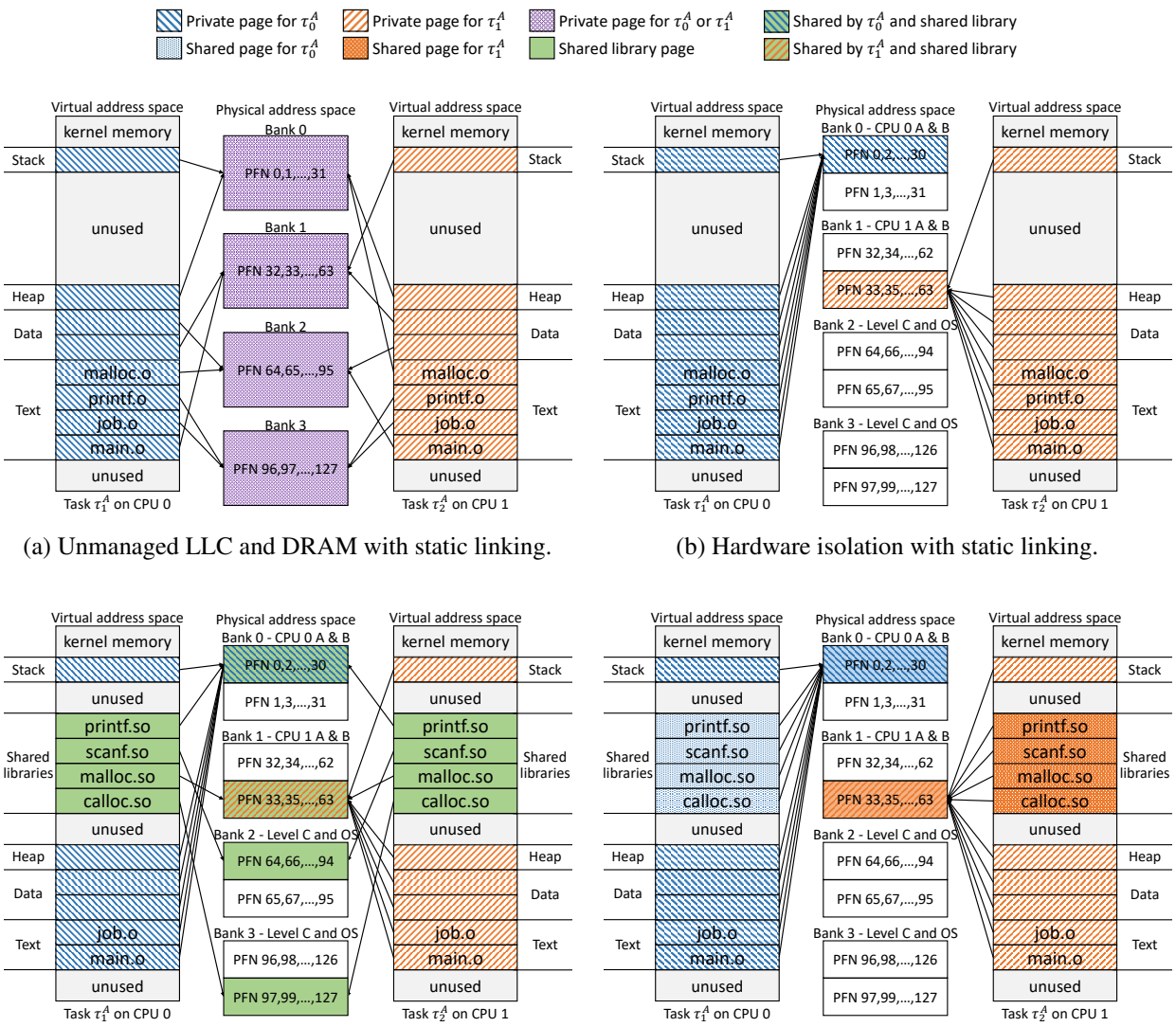
To reduce capacity loss due to statically linked libraries, we propose an alternative strategy, which we refer to as *selective sharing*, that enables us to take advantage of dynamic linking's memory savings. Informally, selective sharing requires regulating "who" may share "what" with "whom." In devising the selective-sharing approach, our major objective was to reduce the system's memory footprint while preserving all pre-existing isolation properties of MC^2.

We begin this section by describing how unrestricted shared library usage breaks hardware isolation. We follow this with a description of selective sharing and compare several implementation strategies. We note that any implementation of selective sharing will yield the same schedulability benefits, so the discussed approaches only differ in their relative ease-of-implementation.

Hardware isolation with shared libraries. To see why shared libraries are problematic when providing hardware isolation, consider Figure 4.9, which depicts the virtual address spaces for two Level-A tasks, τ_1^A and τ_2^A, on a dual-core machine with four DRAM banks, each of which contains 32 pages of physical memory. This is a simpler machine than our Cortex-A9 platform considered in this dissertation, but it is sufficient to illustrate the effects of hardware interference. In Figure 4.9, physical memory pages are referred to by PFNs, and the PFNs in each row of a DRAM bank correspond to a single color in the LLC.

Figure 4.9 (a) shows an example virtual-to-physical address mapping of statically linked programs before our isolation techniques are applied. The figure shows task τ_1^A executing on CPU 0 and task τ_2^A executing on CPU 1. In Figure 4.9 (a), these tasks can use arbitrary page frames, as represented by the lines connecting virtual-memory regions to arbitrary DRAM regions. This can result in LLC or bank interference if the two tasks attempt to access the same DRAM bank or pages mapped to the same LLC region.

In Chapter 3, we provided hardware isolation by migrating page frames. This remapping was realized by introducing a page-migrating system call to MC^2 that each real-time task invokes in order to migrate pages to colored pages. Figure 4.9 (b) illustrates this: task τ_1^A uses entirely separate DRAM banks and page colors from task τ_2^A. Now, the two tasks can run concurrently without having to guard against interferring with one another in the LLC or DRAM banks.



(a) Unmanaged LLC and DRAM with static linking.

(b) Hardware isolation with static linking.

(c) Hardware isolation with shared libraries.

(d) Hardware isolation with selective sharing.

Figure 4.9: Virtual address space and mappings to page frames and LLC under MC^2.

However, as we discussed in Section 4.2.1, this approach requires static linking and consumes more DRAM space, leading to degraded schedulability when memory is considered as a constrained resource. If we wish to save memory by allowing shared libraries, then the mechanisms used previously for providing hardware isolation with static linking are no longer sufficient. In particular, as illustrated in Figure 4.9 (c), private page frames can still be isolated, but pages belonging to shared libraries are accessible to any task that uses the library, even tasks on different processors. (*e.g.*, invocations of `printf()` by the two tasks under consideration map to the same DRAM bank). For this reason, we disallowed shared libraries in previous sections.

Selective sharing. Selective sharing preserves the notion of isolation provided by MC^2 while more efficiently utilizing memory by introducing *per-partition library replicas*. Specifically, up to m+1 replicas are created per library, where m is the number of processors: one per-processor copy is shared by all Level-A and -B tasks running on that processor, another copy is shared by Level-C tasks on all processors. We replicate only the shared read-only sections of a library, which hold both instructions and read-only data. The writable data sections of shared libraries are private, and therefore handled along with other private pages by the page-migrating system call from Chapter 3.

Figure 4.9 (d) illustrates the impact of selective sharing. In this particular example, a replica of each shared library exists for each CPU and is allocated in the CPU's DRAM partition. As seen in Figure 4.9 (d), a library loaded by task τ_1^A will be allocated in Bank 0, and a library loaded by task τ_2^A will be allocated in Bank 1. Furthemore, the allocations will use pages of different colors in order to map to different LLC regions. This ensures that MC^2's isolation properties are maintained.

Having described the concept of selective sharing, it remains to discuss whether it can be efficiently implemented. There are actually several possible implementation strategies, three of which we discuss next. These strategies vary in implementation complexity, but all require some kernel support for providing isolation in DRAM banks and the LLC.

Creating on-disk replicas of shared libraries. The simplest way to create per-partition library replicas is to use Linux's `LD_LIBRARY_PATH` environment variable, which adds a new list of directories the dynamic linker searches for shared libraries before it searches the default directory list. To use this approach, a user must create copies of shared libraries on disk and place them in separate directories. Then, before each real-time task is created, the user must set `LD_LIBRARY_PATH` for each specific task so that it refers to the set of shared libraries corresponding to its criticality level and task-to-processor assignment (if it is a Level-A or -B task). This approach has the advantage of requiring less kernel support (kernel support is still required to implement page coloring), but it requires nontrivial maintenance, disk usage, and configuration effort by the user. These disadvantages become more pronounced if a large number of shared libraries are in use.

Modifying the dynamic linker to provide selective sharing. A modified dynamic linker can potentially provide selective sharing while alleviating some of the maintenance difficulties associated with on-disk replicas. The dynamic linker is responsible for mapping shared libraries into each process's address space,

and typically does so by directly memory-mapping shared library files. A dynamic linker capable of selective sharing would instead need to allocate separate regions of shared memory for each library replica, based on the current task's criticality level and task-to-processor assignment (if it is a Level-A or -B task).

While this approach would be easier to use than on-disk replication, it poses greater implementation challenges. This approach would not only require modifying the code comprising the GNU dynamic linker,[18] but would still require more kernel support than on-disk replicas using `LD_LIBRARY_PATH`. Specifically, it would require modifications of the dynamic linker to request a specific page according to the task's criticality level. Due to this reason, we did not attempt to implement this approach. Even so, we believe that an implementation in this manner is possible.

A transparent kernel-level implementation. The final approach we discuss is transparent to users and is of similar difficulty to, or easier than, dynamic linker modification. This kernel-only approach requires extending the page-migrating system call discussed in Chapter 3 in order to specially handle shared-library pages. The final result is a system call which identifies, replicates, and migrates shared-library pages to the correct partitions in physical memory, in addition to maintaining previous MC^2 functionality.

The Linux kernel offers built-in page migration functions that are capable of safely relocating physical pages.[19] We used these functions to assist with coloring private pages, but unfortunately the default behavior of these functions makes them unusable for remapping shared pages. We worked around this limitation by tracking shared-library pages in a separate data structure and manually overriding the kernel's page-migration behavior when necessary. Beyond user space transparency and disk-space savings, this approach has the advantage of not changing existing system behavior: only programs that invoke the system call are affected.

4.2.3 Implementation

As discussed above, we chose to extend the page-migrating system call to handle shared pages. The procedures to handle shared pages consist of three phases: **(i)** identify shared-library pages, **(ii)** create per-partition library replicas, and **(iii)** adjust page tables. We now describe details of each phase.

[18]The dynamic linker used by most Linux distributions is provided by GNU libc, and consists of several thousand lines of code.

[19]`https://www.kernel.org/doc/Documentation/vm/page_migration`.

Identifying shared-library pages for replication. A task's address space has regions for the executable code and the initialized and uninitialized variables for each shared library used by it. Although library code is shared among many tasks, each task has its own copies of all per-library global and static variables because such pages are marked as writable. Therefore, we replicate the code (read-only) pages of shared libraries. The pages that can be replicated, the code pages of shared libraries, must satisfy the following conditions:

1. The access permissions are read and execute.
2. The reference count of the page is greater than two.
3. The page contents are loaded from a file.

A page contains only instructions if it has read and execute permissions. In Figure 4.8, the fourth column shows permissions of the memory area. A page is shared if it has a reference count exceeding two (in `struct page`); in contrast, a private (*i.e.*, *non-shared*) page has a reference count of two. If the reference count is two, the page is referenced by a task and a *page cache* [20] in the kernel. As shared libraries are loaded from external storage, the file pointer in memory area structure, `vm_file`, is non-null.

Creating shared library replicas. When a shared library is loaded for the first time, a unique set of pages for each shared library is allocated. We keep a "master" list of these initial pages. Each entry in the master list has m+1 pointers to each replica. After a page is classified as a shared-library page, we check whether the page is in the master list or not. If the page is found in the master list and the replicated page for the associated processor exists, we can skip the copying procedure in this phase. If the page is found in the master list, but the associated pointer is null, a replica page is created by allocating a new page from the appropriate DRAM region and the content of the initial page is copied into the new page. Then, the associated pointer in the master entry is set to refer the replica. If the page does not exist in the master list, both a new master entry and a replica are created and inserted to the master list.

Adjusting page tables. In this step, we replace the originally allocated page with a replica by altering the task's page tables to reference the replica pages instead of the original ones. The original page, no longer referenced by the task's page tables, is not de-allocated as it is kept in the page cache. The original page must

[20] The page cache is a disk cache that holds in-memory copies of pages from files on disk so that the kernel can quickly return the requested page.

be kept in the page cache because the kernel first looks up the page cache when a memory mapping is being established. Thus, it is guaranteed that the task has a mapping to a master page, not replicas before the task invokes the system call.

4.2.4 Introducing Memory Constraints

Understanding the impacts of supporting shared libraries in MC^2 requires an understanding of the limitations on available DRAM space on our studied platform. In this section, we begin by examining the impact of selective sharing. We then provide more detail regarding memory constraints. We also examine DRAM and LLC allocation techniques disregarded in Chapter 3. These techniques break certain isolation guarantees and thus were *pointless to consider earlier*. However, as we will show, these techniques provide advantages in DRAM space allocation and so *are worth considering now* since we are viewing the available DRAM space as a constrained resource now.

Memory footprint statistics. To illustrate the impact of selective sharing on DRAM use, we created a simple task system, using publicly available benchmark programs in the Data Intensive Systems (DIS) Stressmark Suite (Musmanno, 2003) and the PARSEC Benchmark Suite (Bienia, 2011) and analyzed DRAM consumption. Table 4.3 presents the RSS of the considered benchmark programs when libraries were non-shared (*i.e.*, linked statically) and selectively shared (*i.e.*, linked dynamically). This data was obtained with the smallest input data set assuming one instance of each of tasks $\tau_1, \ldots, \tau_{10}$ is assigned to Levels A and B on each processor and one instance of each of tasks τ_{11} and τ_{12} is assigned to Level C, on the hardware platform shown in Figure 2.4, which matches the execution times used in Section 4.2.5. This task system requires 11.7% less memory if selective sharing is used. The relative impact of selective sharing increases if a greater opportunity for sharing exists. For example, if we were to increase to five instances of each of $\tau_1, \ldots, \tau_{10}$ per processor and five instances of each of τ_{11} and τ_{12}, then memory consumption would be reduced by 22.9% compared to static linking.

While these results demonstrate that our techniques can improve efficiency in DRAM use, understanding the benefits in a holistic sense requires examining impacts on overall schedulability. To assess such impacts, we conducted a large-scale overhead-aware schedulability study. Before discussing the results of this study, we first delve into some issues that arose when introducing the effects of memory-capacity limits on schedulability.

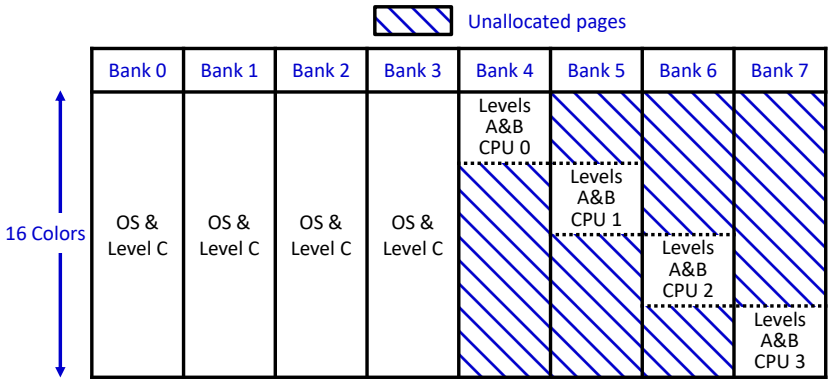
Task	Program Name	Non-Shared	Selectively Shared	
			Private Pages	Shared Pages
τ_1	field	1456 KB	1136 KB	4920 KB
τ_2	update	2284 KB	1964 KB	
τ_3	matrix	1936 KB	1600 KB	
τ_4	transitive	680 KB	360 KB	
τ_5	neighborhood	1208 KB	852 KB	
τ_6	pointer	2284 KB	1964 KB	
τ_7	blackscholes	700 KB	340 KB	
τ_8	ferret	2260 KB	1428 KB	
τ_9	swaptions	1064 KB	392 KB	
τ_{10}	x264	1368 KB	348 KB	
τ_{11}	fluidanimate	10328 KB	9720 KB	2144 KB
τ_{12}	freqmine	23824 KB	23188 KB	

Table 4.3: DRAM consumption of an example task system assuming non-shared and selectively shared libraries.

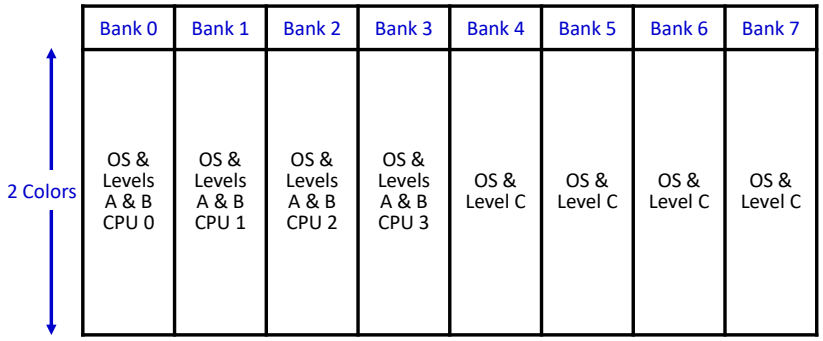
Impact of bank interleaving. Figure 4.10 (a) shows a more detailed view of the DRAM allocation scheme assumed in Chapter 3, as depicted earlier in Figure 3.5. Levels A and B are bank-partitioned from other processors to avoid cross-core contention in bank access at the highest criticality levels. Additionally, bank interleaving, which spreads contiguous pages across multiple banks, is disabled. This is because, under bank interleaving, each bank only contains two page colors, which creates dependencies between LLC allocation and bank allocation. When bank interleaving is disabled, each bank has pages of all 16 colors, which allows banks and colors to be allocated independently. As discussed in Section 3.1.2, we disabled bank interleaving in Chapter 3 because it permits more fine-grained control over page allocations.

Unfortunately, with DRAM now being viewed as a constrained resource, a disadvantage of disabling bank interleaving is exposed: because each Level-A and -B area of the LLC corresponds to one quarter of the available colors, only one quarter of the pages in each Level-A and -B bank can be allocated as illustrated in Figure 4.10 (a). One way to reclaim this lost DRAM space while maintaining LLC isolation is to enable bank interleaving and assign to the Level-A and -B subsystem on each processor an LLC area corresponding to the colors of the designated bank for that processor. Figure 4.10 (b) depicts the resulting DRAM allocations and Figure 4.11 shows the corresponding LLC layout. In this section, we consider this *interleaved approach* as an alternative to the *non-interleaved approach* considered in Chapter 3.

The new interleaved approach being considered here comes at a cost. As shown in Figure 4.10 (b), the OS is no longer restricted to DRAM banks shared with Level C. Specifically, the OS claims the first several



(a) Non-interleaved.



(b) Interleaved.

Figure 4.10: Page allocation in DRAM banks.

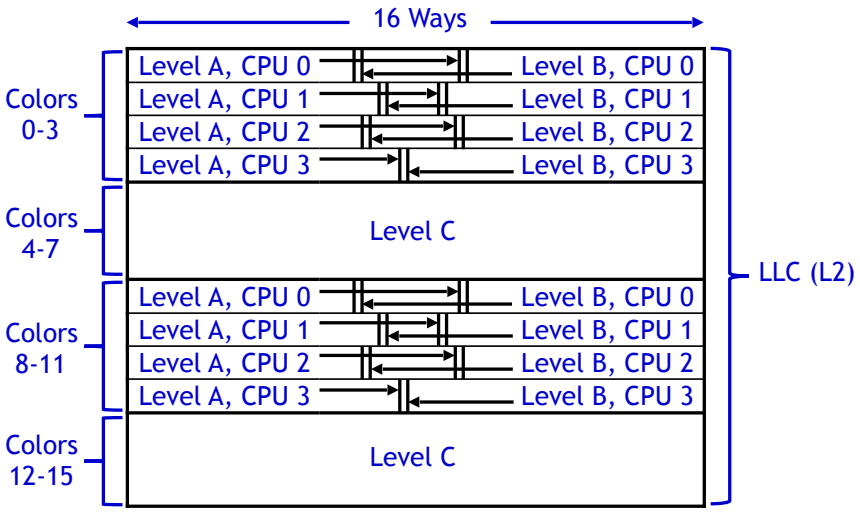


Figure 4.11: Interleaved LLC allocation.

hundred pages of physical memory at boot time. These pages are used for unmovable memory, including the kernel image and the area reserved for hardware devices. While these pages are relegated to Bank 0 when interleaving is disabled, they are spread across all banks when interleaving is enabled. As a result, interleaving eliminates bank isolation from the OS at Levels A and B. Additionally, because Level C is now color partitioned in the LLC as shown in Figure 4.11, the OS (which has access to all page colors) can no longer be restricted to the Level-C LLC partition. In order to ensure that Levels A and B are isolated from the OS in the LLC under the interleaved scheme, we assume that tasks do not invoke system calls during execution in this section.[21]

From the above discussion, it should be clear that both allocation schemes under consideration have advantages and disadvantages. To better understand the tradeoffs between them, we included both in our overhead-aware schedulability study, which is discussed next.

4.2.5 Evaluation

To assess the efficacy of selective sharing, we evaluated the schedulability of millions of randomly generated task systems under the MC^2 variants listed in Table 4.4. In denoting these variants, the prefix "I" (respectively, "NI") denotes that the interleaved (respectively, non-interleaved) memory-allocation scheme was used, as depicted in Figure 4.10 (b) and Figure 4.11 (respectively, Figure 4.10 (a) and Figure 3.5). The suffix used indicates how libraries were dealt with: "STC" denotes statically linked libraries, "SSH" denotes selectively shared libraries, and "IDL" (ideal) denotes viewing memory as an unconstrained resource as we did in Chapter 3. In addition to these MC^2 variants, we considered the HRT uniprocessor EDF scheduler, denoted U-EDF. This reflects current industry practice for eliminating shared-hardware interference by disabling all but one core. Under U-EDF, no DRAM capacity constraints were assumed.

Under the "STC" and "SSH" schemes, constraints on DRAM memory were considered. For these schemes, we calculated the total number of DRAM pages available to real-time tasks in MC^2 on our considered ARM platform after accounting for pages used by the OS. For the I-STC and I-SSH schemes, Levels A and B have approximately 25,000 pages allocated per processor, and Level C has approximately 100,000 pages allocated in total. For the NI-STC and NI-SSH schemes, Levels A and B have approximately

[21] We lift this assumption in Section 4.3 with the support for dynamic memory allocation in the kernel.

DRAM & Linking Assumptions	DRAM-Aware		DRAM-Oblivious
	Static	Sel. Shared	DRAM Constraints Ignored
Interleaved	I-STC	I-SSH	I-IDL
Non-Interleaved	NI-STC	NI-SSH	NI-IDL

Table 4.4: Considered MC^2 variants.

8,000 pages allocated per core[22], and Level C has approximately 73,000 pages allocated in total. Under the I-SSH and NI-SSH schemes, a task system's DRAM consumption was calculated assuming that all libraries are selectively shared.[23]

Task-system generation. We generated task systems by extending the process discussed in Chapter 3 to account for DRAM consumption. Task systems were randomly generated by using seven uniform distributions to choose task and task-system parameters. The specific distributions used were selected from the per-distribution choices listed in Table 4.5. All combinations of these choices were considered. At a high level, our overall experimental framework refines the following step-wise process used in Chapter 3:

Step 1 Select seven specific distributions from among the distribution categories listed in Table 4.5.

Step 2 Using the selected distributions from the first four categories, generate task-system parameters under U-EDF.

Step 3 Based on the generated U-EDF PETs, generate PETs for the MC^2 schemes. This process is informed by micro-benchmark data, as discussed at length in Chapter 3.

Step 4 Adjust the generated task parameters to account for relevant overheads. The actual overhead values applied are based on measured data.

Step 5 For each task, assign linked libraries from the list in Table 4.6. These are libraries used by the benchmark programs listed in Table 4.3. The first four libraries, which are shared by all benchmark programs under MC^2, were assigned to all tasks. Each Level-A and -B task has a 50% chance of

[22] Some pages from Level-A and -B banks are used by per-processor `kmem_cache` pages at boot time, which will be discussed in Section 4.3.

[23] To keep our study manageable, we examined the two extremes of always using static linking and always using selective sharing. In practice, some combination of the two might result in the best memory utilization. Static linking is often optimized to only link the portion of a library used by a linking task, so static linking occasionally consumes less DRAM than dynamic linking. For the best memory utilization, we could modify the optimization framework described in Chapter 3 to optimize for each individual library.

Category	Choice	Level A	Level B	Level C
1: Criticality Utilization Ratios	A-Heavy	[50, 70)	[10, 30)	[10, 30)
	B-Heavy	[10, 30)	[50, 70)	[10, 30)
	C-Heavy	[10, 30)	[10, 30)	[50, 70)
	AB-Mod.	[35, 45)	[35, 45)	[10, 30)
	AC-Mod.	[35, 45)	[10, 30)	[35, 45)
	BC-Mod.	[10, 30)	[35, 45)	[35, 45)
	All-Mod.	[35, 45)	[35, 45)	[35, 45)
2: Period (ms)	Short	$\{3, 6\}$	$\{6, 12\}$	[3, 33)
	Long	$\{48, 96\}$	$\{96, 192\}$	[50, 500)
3: Task Utilization	Light	[0.001, 0.03)	[0.001, 0.05)	[0.001, 0.1)
	Heavy	[0.1, 0.3)	[0.2, 0.4)	[0.4, 0.6)
4: Max Reload Time	Light	[0.01, 0.1)	[0.01, 0.1)	[0.01, 0.1)
	Heavy	[0.25, 0.5)	[0.25, 0.5)	[0.25, 0.5)
5: % DRAM Reserved for Task Set	Small	10		
	Moderate	40		
	Large	80		
6: Private Page Count (SSH)	Light	[50, 200)	[50, 200)	[100, 400): 0.75 [500, 1000): 0.25
	Heavy	[200, 500)	[200, 500)	[300, 600): 0.75 [1000, 7000): 0.25
7: Private Page Count Increase	Light	[80, 150)	[80, 150)	[100, 300)
	Heavy	[150, 250)	[150, 250)	[300, 500)

Table 4.5: Task-set parameters and distributions. In Category 6, last column, I:P denotes that interval I is selected with probability P.

having *Libm*. The remaining libraries are used by computationally intensive benchmark programs and were only assigned to Level-C tasks. To each Level-C task, we randomly assigned two to six libraries from the last six libraries listed.

Step 6 Assign Level-A and -B tasks to processors using a worst-fit-decreasing heuristic.

Step 7 Determine the DRAM consumption of the task system under the "STC" and "SSH" schemes. The distributions in the last two categories are used here. To determine the DRAM consumed by a task system at Level C and with respect to each Level-A and -B bank, we first must calculate the DRAM consumption for private pages of tasks. For each task, we choose a private-page count from the selected distribution under Category 6. This is the private-page count under the "SSH" schemes. The distributions chosen reflect private-page counts for the considered benchmark tasks in Table 4.3 when all libraries are selectively shared. In keeping with our thesis that Levels A and B consist mostly

Name	Size in KB	Assignment
libc	872	Assigned to all tasks
ld	92	
librt	20	
libpthread	64	
libm	396	Assigned to Level-A and -B tasks
libstdc++	608	Assigned to Level-C tasks
libjpeg	296	
libgslcblas	152	
libgsl	1396	
libgcc_s	100	

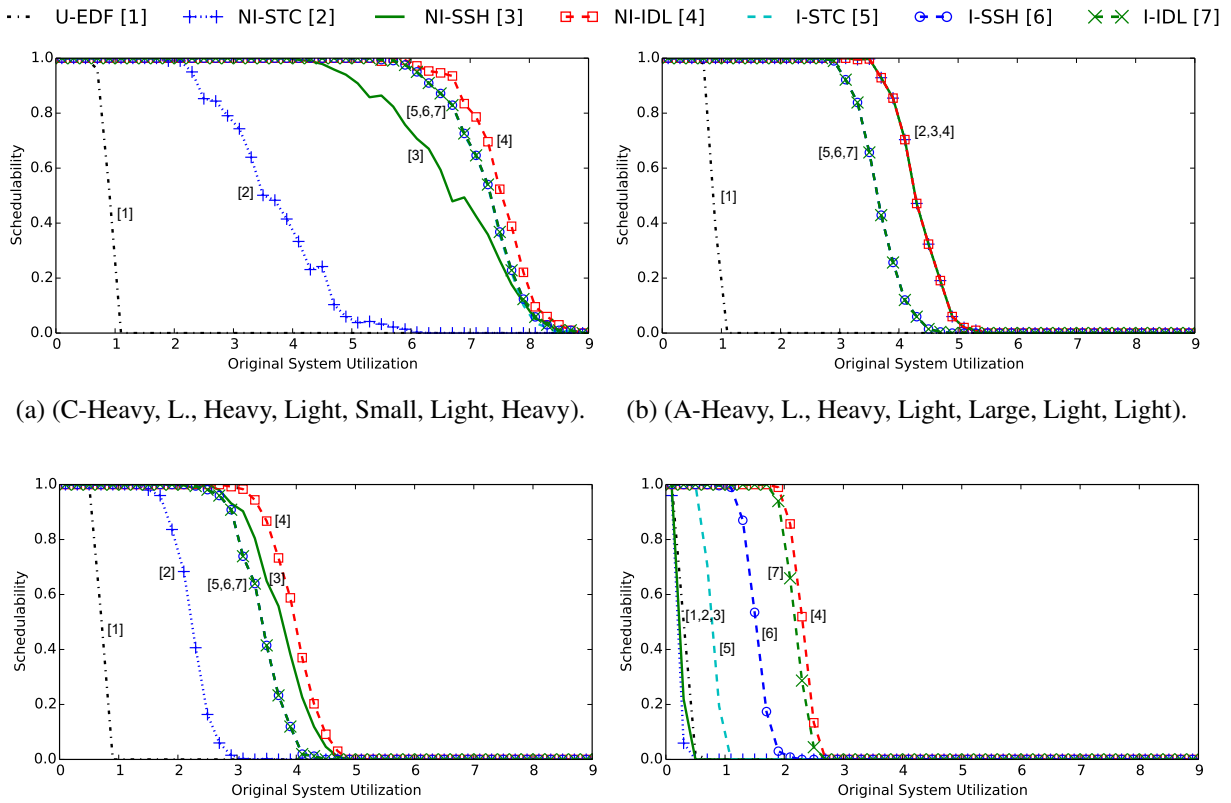
Table 4.6: The size of the code segments of considered libraries under selective sharing.

of "fly-weight" tasks (refer to Section 2.2), page-count distributions for Level-A and -B tasks yield smaller page counts than those for Level-C tasks.

We add an additional number of private pages under the "STC" schemes. The page-count increase for a task is chosen from the selected distribution in Category 7. Page-count increases under static linking for the benchmark programs in Table 4.3 are within the range [80, 260). The last category distributions were designed to encompass this range. Real-world systems may use more library functions than used in our benchmark programs, which could lead to greater static-linking overheads. To account for this possibility, we actually allow page-count increases up to 500. To ensure that the page-count increase for a task is reasonable, we adjust the increase to be no greater than 90% of the combined size of all libraries linked to the task.

Step 8 Check the schedulability of the resulting task system under U-EDF and each MC^2 scheme in Table 4.4. Category 5 is the portion of available DRAM that can be allocated to the task system. Some portion of DRAM may be unavailable due to space required to support non-real-time tasks or tasks specific to alternate *modes*.[24] The percentage of DRAM reserved is selected from Category 5 in Table 4.5. For example, if Moderate is selected, then under non-interleaved schemes, the DRAM consumption of generated Level-A and -B tasks on each processor must be at most 40% of the 8,000 pages. If the DRAM consumed in a Level-A and -B bank exceeds 40% of the 8,000 pages, then we deem the system as unschedulable under that scheme.

[24] Many safety-critical applications must support multiple functional modes, each defined by a distinct set of running tasks. For example, in an aircraft, different sets of running tasks may be required when taking off, at cruise altitude, or when some emergency condition occurs.



(a) (C-Heavy, L., Heavy, Light, Small, Light, Heavy).

(b) (A-Heavy, L., Heavy, Light, Large, Light, Light).

(c) (B-Heavy, L., Heavy, Heavy, Small, Light, Heavy).

(d) (A-Heavy, L., Light, Heavy, Small, Light, Light).

Figure 4.12: Representative schedulability plots.

Schedulability results. We generated enough task systems to estimate mean schedulability to within ± 0.05 with 95% confidence with at least 100 and at most 2,000 task systems. In total, we evaluated the schedulability of over five million randomly generated task systems, which took roughly 25 CPU-days of computation. From this abundance of data, we generated 672 schedulability plots, of which four representative plots are shown in Figure 4.12. The full set of plots is available in Appendix C.

Each schedulability plot corresponds to a single scenario, which represents a combination of distribution choices. We now state several observations that follow from the full set of collected schedulability data. We illustrate these observations using the data presented in Figure 4.12.

Observation 4.6. *Schedulability under* I-STC *was better than under* NI-STC *in 61% of cases. Conversely, schedulability under* NI-SSH *was better than under* I-SSH *in 54% of cases.*

This observation is supported by insets (a) and (c) of Figure 4.12. Under the "STC" schemes, DRAM is more of a limiting resource (since static linking wastes space), so the DRAM loss illustrated Figure 4.10 (a)

for non-interleaved memory becomes a liability. Under the "SSH" schemes, DRAM is a less-constraining resource, so the virtues of using non-interleaved memory usually win out (it is these virtues that caused us to only consider this possibility in previous sections).

Observation 4.7. *Schedulability loss under I-STC (respectively, NI-STC) was non-negligible (at least 10% of one core's capacity compared to an "IDL" allocation) in 27% (respectively, 61%) of scenarios.*

Figure 4.12 (b) shows a scenario with negligible loss. In scenarios with light memory consumption or large DRAM reservations, schedulability was rarely impacted by DRAM capacity.

Observation 4.8. *For scenarios with non-negligible schedulability loss under I-STC (respectively, NI-STC), I-SSH (respectively, NI-SSH) regained on average 43% (respectively, 36%) of schedulability lost under I-STC (respectively, NI-STC).*

Such regained schedulability can be seen in insets (a) and (c) of Figure 4.12. We note that it is unreasonable to expect most of the loss to be consistently regained, because the "IDL" schemes may deem systems to be schedulable whose memory footprints will not even fit into DRAM regardless of libraries. From these and similar scenarios, we conclude that selective sharing can result in significant schedulability gains.

Observation 4.9. *Interleaved schemes showed better schedulability when DRAM is the major constraining resource.*

Figure 4.12 (d) shows such a scenario. In scenarios with heavy memory consumption at Levels A and B, the disadvantage of disable bank interleaving was exposed. As shown in Figure 4.10, non-interleaved schemes can use only a quarter of pages in a Level-A and -B bank while interleaved schemes can use the entire DRAM space.

Given the nature of our study, the observations above naturally hinge on our experimental setup. However, we have taken great pains to ensure that a wide range of potential system configurations were considered. From the full set of collected schedulability data, our approach for dealing with shared libraries often improved schedulability significantly.

4.3 Sharing Between Kernel and User Space

In the preceding two sections, we addressed data-sharing among tasks using shared memory and read-only sharing through the usage of shared libraries. Hardware interference caused by such sharing arises from

interactions among user-level tasks. However, user-level tasks are not the only source of hardware interference. The OS generates contention for hardware components as well. The sharing of data between the OS and user-level tasks occurs when tasks invoke OS services for IPC or I/O. A task system considered in previous sections is not allowed to do any activity that requires involvement from the kernel, such as system calls, I/O activities, and IPCs other than shared memory, to avoid OS-induced interference. However, such a task system is not sufficient to support real-world workloads in practice.

In this section, we propose techniques for lessening the impact of OS-induced sharing. We extend MC^2 to enable *dynamic memory allocation* for supporting I/O and IPCs via fine-grained OS isolation. The prior MC^2 implementation provided no control over how the OS allocates memory regions used for sharing data with user-level tasks. In addition, we address the problem of way-based OS isolation discussed in Section 3.3.4 by using criticality-aware dynamic memory allocation.

We begin by describing how OS-induced sharing introduces interference. We then present techniques to mitigate such interference. Finally, we evaluate our techniques with micro-benchmark experiments and a schedulability study.

4.3.1 OS-Induced Sharing

By necessity, any RTOS must balance conflicting objectives. Judging from research papers, the primary objective is ostensibly support for predictable timing. The second objective, however, is likely more important: an RTOS *must carry out useful work*. MC^2 bridged the gap between these two demands by enabling user-level data sharing between tasks, but this did not go far enough for most real-world use cases. In particular, a system can only be useful if it *produces output*, and this fundamentally requires device I/O. Like IPC, device I/O is a type of data sharing, and, like all forms of data sharing, can cause interference. For example, when Level-A and -B tasks use IPC or I/O buffers in Level-C banks, they may suffer row buffer misses from Level-C activity. Previously, such interference has been considered in the context of temporal isolation (Pellizzoni et al., 2008; Pellizzoni and Caccamo, 2010; Kim et al., 2014; Muench et al., 2014), which is orthogonal to our approach. This section discusses how data sharing, including IPC and I/O, is at odds with the objective of hardware isolation. As mentioned earlier, we extend the data-sharing capabilities of MC^2 in two key areas: IPC and device I/O.

OS-supported IPC. We addressed data sharing in Section 4.1, but took a limited view of the solution: it only supported user-level IPC using shared memory. This limitation facilitates maximizing isolation, because it avoids the need for system calls or dynamic memory allocations. Shared-memory buffers can be allocated and isolated during task initialization, and can be accessed without involving the OS.

However, shared-memory IPC is insufficient in many cases. For example, even simple message-passing systems require in-memory synchronization primitives or wait-free data structures as discussed in Section 4.1. These shortcomings are addressed by other IPC mechanisms such as message queues or pipes, but these mechanisms break isolation because, in addition to sharing with other tasks, they require sharing data with the OS kernel. Involving the OS kernel is particularly problematic because *the OS is fundamentally shared among all tasks*.

Device I/O. Even if OS-supported IPC can be achieved without compromising isolation, IPC is still only one of many ways in which programs share data. A second major source of data sharing is device I/O. Modern device I/O is largely centered around DMA, where hardware peripherals can directly read from or write to the same DRAM as the CPU. So, to support device I/O, the first requirement is that a real-time system must *prevent unpredictable DRAM interference due to DMA*. However, device interaction does not end with raw data entering DRAM—it must also be processed by user-level tasks. Therefore, a second requirement is that a real-time system must *deliver I/O data to user-level tasks* (*i.e.*, the OS must place I/O data into user space in order to be accessible to user-level tasks). The second objective is similar to OS-supported IPC because low-level interaction with hardware is typically delegated to the OS kernel, even if I/O is initiated by user-level tasks.

Types of interference due to data sharing. The prior paragraphs established *why* data sharing can lead to hardware interference without describing the specifics of *how* the interference occurs. Fortunately, the types of interference we mitigate in our modifications to MC^2 can be simplified into two categories:

- **CPU-sourced interference.** CPU-sourced interference occurs when tasks suffer interference due to other tasks concurrently accessing a DRAM bank or cache region. Unmanaged IPC and interactions with the OS cause this type of interference.
- **DMA-sourced interference.** DMA-sourced interference occurs when tasks suffer interference due to DMA transfers. Such interference may be caused by concurrent DRAM accesses and contention

caused by DMA transfers in accessing unmanaged resources, such as L1 caches, TLBs, memory buses, memory controllers, and cache-related registers (recall Section 2.4.3). Because DMA bypasses the CPU entirely on our Cortex-A9 platform, DMA-sourced interference causes no cache interference. [25]

Our modifications to MC^2's DRAM and cache-aware memory-management system reduce both of these sources of interference. We note here that other kernel data structures such as task structures, page tables, and page caches are already isolated in Level-C DRAM banks as we described in Section 3.2.2.

4.3.2 Memory Interference from I/O Devices

To understand some of the difficulties with data sharing, one can observe the procedures for interacting with devices on a Linux-based system. I/O for different devices can differ surprisingly in terms of software complexity and the potential sources of memory interference. We illustrate this point using two devices: a secondary-storage disk and a USB video camera.[26]

Memory interference from zero-copy I/O. "Zero-copy" refers to I/O that does not require copying data between separate memory buffers. Figure 4.13 depicts one possible type of zero-copy I/O in Linux: reading from secondary storage.[27] Generally, a program reads from a disk by issuing a `read` system call and specifying a user-allocated memory buffer to receive the data. This is shown at the top of Figure 4.13. This prompts the kernel to determine the specific sector(s) of the disk to read, and to issue a request to the disk via the appropriate communication bus. The data transfer is then actually handled by the disk itself, which will use DMA to populate the user-space buffer. When the DMA transfer is complete, the disk will send an interrupt to the kernel, which returns control to the user task. Finally, the user task is free to operate on the received data.

The zero-copy example from Figure 4.13 illustrates two useful points. First, it illustrates both DMA-sourced interference when the device writes to the buffer, and CPU-sourced interference when the task

[25] Interference due to cache evictions does not occur. However, overhead due to the coherency protocol may exist. In the ARM Cortex-A9 processor, the cache-coherency protocol is implemented and managed by the Snoop Control Unit (SCU) hardware. The advantage of this hardware-based approach is that the broadcast of cache maintenance operations happens in hardware automatically with very low overhead (ARM Limited, 2009).

[26] USB devices may not be common in HRT systems; we use a USB camera only as an exemplar of devices where OS activity may cause memory interference.

[27] This actually is not the default disk-access behavior in Linux; zero-copy disk I/O requires passing the optional `O_DIRECT` flag to the `open` system call. The `O_DIRECT` flag promises that the kernel will avoid copying data from user space to kernel space, and will instead write it directly via DMA.

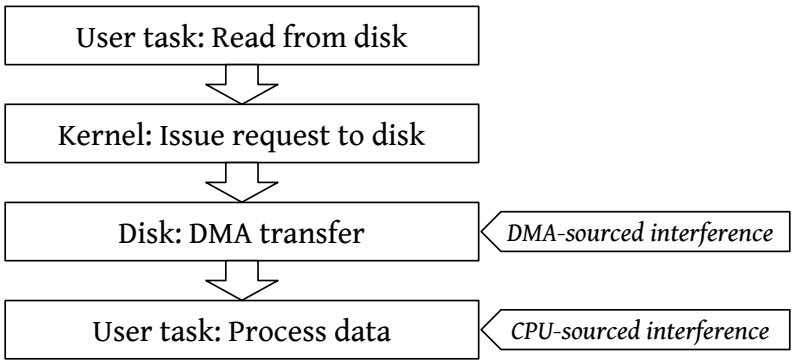


Figure 4.13: Simplified direct disk I/O data flow.

accesses it. Second, being zero-copy, the example involves only a single, user-allocated buffer. Thus, both sources of interference can be managed by properly provisioning a single region of memory. For example, if we allocate the buffer in a Level-A/B bank, the I/O-performing task does not experience CPU-sourced interference, but DMA-sourced interference due to bank conflicts could be suffered by other Level-A/B tasks. However, if we allocate the buffer in the Level-C banks, the I/O-performing task experiences only CPU-sourced interference.

Memory interference from USB I/O. In contrast to the zero-copy example, Figure 4.14 illustrates the flow of data when a user-level task attempts to read a frame from a USB camera on a Linux-based system like MC^2. Tasks using Linux's standard *Video for Linux version 2* `(v4l2)` API must first request one or more buffers to hold video frames, but this actual allocation is managed in kernel code. Next, the user task may issue a request to read a new video frame, which prompts the kernel to start receiving data from the camera. Despite using DMA to transfer USB packets, each USB packet only contains a small portion of the overall frame, which must be copied to the frame buffer. Finally, when the kernel finishes copying an entire frame, the user task is able to access the frame buffer.

Figure 4.14 demonstrates the counterintuitive fact that device complexity is not necessarily related to difficulty in preventing interference. In the zero-copy example, ensuring the isolation of a single user-allocated buffer is sufficient to prevent unpredictable CPU- and DMA-sourced interference. However, the seemingly simpler USB webcam driver uses intermediate buffers that can also be subject to CPU- and DMA-sourced interference. An RTOS must instrument allocation decisions by each device driver. Experiments presented in Section 4.3.6 indicate that the decision as to where to place these dynamic buffers introduces subtle tradeoffs.

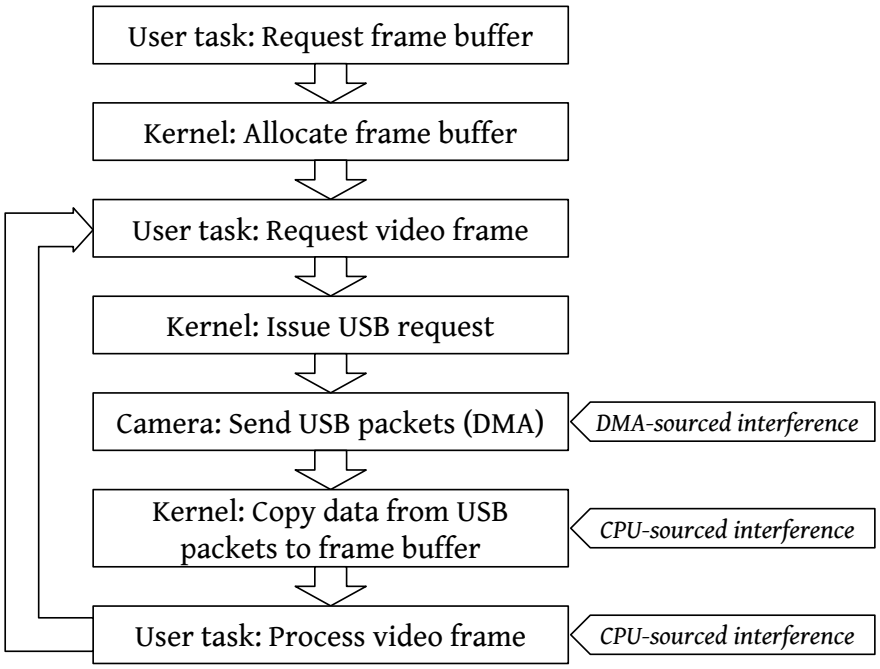


Figure 4.14: Simplified USB camera I/O data flow.

Non-data-related sources of interference. While the discussion above focuses on DRAM and cache interference due to *data transfers*, both IPC and I/O also involve other sources of interference, including interrupt overhead and interference due to instruction fetches. In practice, however, we found that these sources of interference are small enough to account for by inflating PETs.

To further reduce the impact of interrupt interference, we redirect all interrupts to a single processor. To account for interrupt handling time, we inflate the PET of any tasks assigned to this processor accordingly.[28] As MC^2 always places kernel code into the Level-C banks and LLC partition, the OS accesses Level-C DRAM and LLC areas when it executes interrupt handlers or invokes system calls. Therefore, Level-A and -B tasks assigned to the interrupt-handling processor do not experience interference with respect to the LLC and memory banks. However, when Level-A and -B tasks on this processor invoke system calls, they suffer small additional interference because the OS fetches instructions from the Level-C banks and LLC partition regardless of where IPC or I/O data buffers are located. This, unfortunately, is an unavoidable consequence of loading kernel code into Level-C banks at boot time—and the MC^2 kernel is too large to fit into a Level-A/B bank.

[28]This requires knowledge of worst-case interrupt interarrival and execution times. We assume that we operate in a "closed world," with *a priori* knowledge of interrupt types and maximum frequencies, as is typically assumed in real-time overhead accounting.

4.3.3 Implementation

The possibility for interference in both I/O and IPC stems from a single difficulty: the need to isolate MC^2's kernel-managed memory buffers. As discussed in Section 3.3.3, we modified the buddy allocator to support criticality-aware memory allocation for user-level tasks. This section discusses our modifications to MC^2's memory-management system to control dynamic-memory allocations in kernel space. We begin by revisiting the modified memory-management system in MC^2.

Memory-management system in MC^2. As discussed in Section 3.2, MC^2's primary means of reducing shared-hardware interference is cache partitioning and DRAM bank isolation. Level-C tasks may share a dedicated region of the LLC or certain DRAM banks, but Level-A and -B tasks are guaranteed to be free from unpredictable interference whenever possible. Each task is a Linux process, and must invoke a page-migrating system call after initialization. This system call prompts the MC^2 kernel to migrate the task's memory to appropriate physical locations.

This page-migrating approach has two shortcomings. First, *it only occurs once per task, after initialization*. Second, *it only migrates pages allocated in a task's own address space*. These problems preclude isolating IPC and device I/O, which use dynamically allocated kernel memory. We addressed both shortcomings by modifying the kernel's memory-allocation routines.

Isolating dynamic-memory allocations. Our modifications for isolating dynamic-memory allocations required changes to the memory management APIs in kernel. When the kernel allocates memory, either on its own or on behalf of a user-level task, the buddy allocator searches for an appropriately sized free chunk of contiguous physical memory, or a collection of non-contiguous pages if necessary. These pages are then removed from the free list and can be mapped into virtual memory. As discussed in Section 3.3.3, our modification to the buddy allocator consists of replacing the single list of free pages with m+1 independent lists. m of these lists hold free pages for the Level-A and -B tasks on each of the m processors. The additional list holds free pages for Level-C tasks. We extended Linux's core allocation function, `__alloc_pages()`, which is used by all the allocation API functions, to allow specifying which of the m+1 lists to allocate from; by default, we allocate from the Level-C list.

With this modification, all memory requests are served by the Level-C/OS DRAM banks unless an allocation request explicitly requires Level-A/B pages. To request the latter type pages explicitly, we extended

the *Get Free Page (GFP)* flags. These flags determines how the allocator will behave for the allocation. We added a bit in the GFP flags. If this bit is set, the buddy allocator determines which processor is making the request and allocates from that processor's free-page list for its Level-A/B bank.

Device drivers and IPCs request kernel memory via two simple interfaces, `vmalloc` and `kmalloc`. As `vmalloc` calls the `__alloc_pages` function, we can achieve criticality-aware dynamic allocations in kernel by extending the `__alloc_pages` function as mentioned earlier. The `kmalloc` function is used when small memory buffers are required. As the kernel manages the system's physical memory in page-sized chunks, the `kmalloc` function uses a different memory allocation technique. To handle such small memory allocations, Linux uses a set of pools of memory objects of fixed sizes; this functionality is provided by the `kmem_cache` allocator. We also extended this `kmem_cache` allocator to create per-processor memory pools with Level-A/B pages in order to serve small memory allocations on behalf of Level-A and -B tasks. The `kmem_cache` allocator creates memory pools of different sizes for `kmalloc` allocations at boot time by subdividing pages obtained from the buddy allocator.[29] Therefore, modifications to the buddy allocator ultimately affect *all* dynamic memory allocations, of both large and small buffers, and in both the kernel and user space.

Safe default behavior. A major benefit of the modifications outlined above is that Level-C tasks can dynamically allocate memory without further kernel modification. *Level-C memory allocations, issued by Level-C tasks or by the kernel on behalf of Level-C tasks, no longer cause unpredictable interference in Level-A and -B tasks*. Even though our new allocator is capable of obtaining isolated pages for Level-A and -B tasks, doing so requires explicitly modifying the driver or kernel code where pages are allocated. This is because if higher-criticality tasks use IPC or I/O buffers in Level-C banks, they may still suffer cache evictions from Level-C activity. We address such additional complications in the next section.

4.3.4 Optimizing Interference

With the modified memory-management system, we now discuss how we leveraged our modifications to reduce or prevent CPU- and DMA-sourced interference.

[29] This is the reason why we cannot use all 8,192 pages from a Level-A/B bank as described in Section 4.2.5.

Optimizing IPC-related interference. Even with the kernel modifications described in Section 4.3.3, optimized usage of the newly introduced features requires additional input from MC^2's offline optimization component. Usually, we want to minimize interference experienced by Level-A and -B tasks, which are HRT and are therefore most sensitive to additional overhead. Ideally, Level-A and -B tasks should *only experience interference in the bounded amount of time when they access shared buffers*. We considered this problem in Section 4.1 with statically allocated user-level buffers. Here, we apply the same proposed techniques to reduce interference in dynamically allocated memory. The techniques are:

- **Selective LLC Bypass (SBP)**: Allocate buffers from Level-C banks, but make them *uncacheable*. Even if tasks concurrently access these buffers, they will not cause other content to be evicted from the LLC, avoiding cache interference.
- **Concurrency Elimination (CE)**: If two communicating tasks are Level-A or -B tasks, assign both to the same CPU, and allocate the buffers from that processor's Level-A/B bank. Two tasks on the same processor cannot concurrently interfere with each other.
- **LLC Locking (CL)**: Lock a pre-allocated buffer into the LLC, so data can be shared without risking evictions or row-buffer conflicts. This approach reduces LLC space for other purposes, but eliminates both cache and DRAM bank interference.

Even though only one of these approaches may be applied to a single given buffer, different approaches can be in use for different buffers across the entire system. To this end, we modified MC^2's offline optimization component to choose the appropriate mechanism for each pair of communicating tasks. We evaluate the efficacy of this optimization in Sections 4.3.5 and 4.3.6.

Optimizing I/O-related interference. Because DMA-sourced interference only affects DRAM banks and not the LLC, we explore comparatively fewer management options for I/O buffers. Even so, with the ability to allocate kernel DMA buffers, we can handle DMA-sourced interference in two ways. The first, simpler, way is the default behavior described in Section 4.3.3: place DMA buffers in Level-C banks. The second way is to allocate DMA buffers in a Level-A/B bank if the corresponding device is being used by a Level-A or -B task.

These two approaches have different implications regarding how tasks are impacted by DMA-sourced interference. For example, if DMA buffers are in Level-C banks, then Level-A or -B tasks do not experience

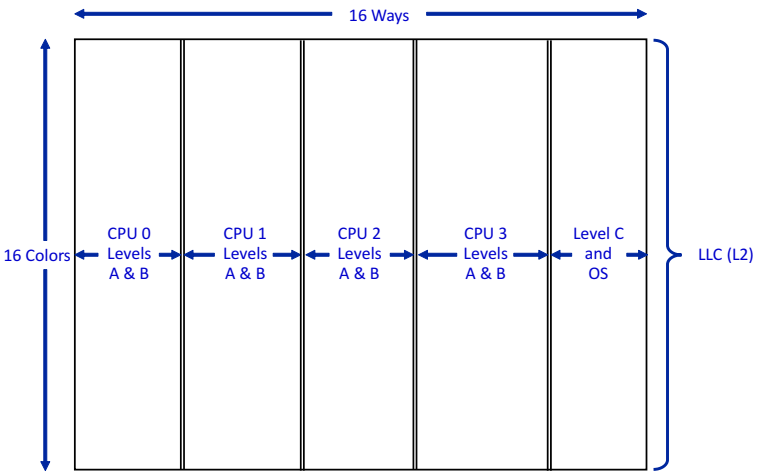
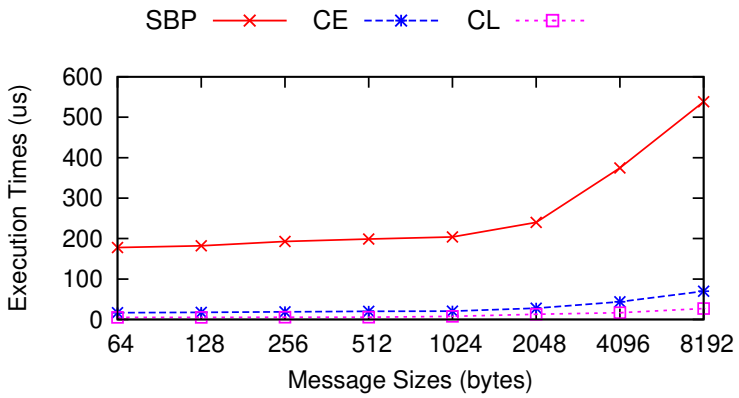


Figure 4.15: Variant 4 (Way-based-isolated LLC).

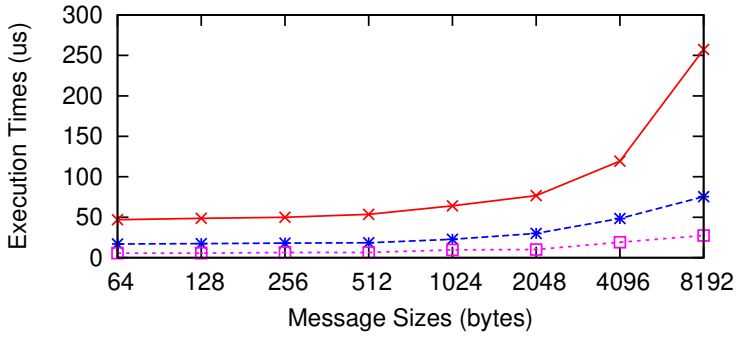
row-buffer conflicts as part of DMA-sourced interference. However, as discussed in Section 4.3.1, I/O data is useless without being accessed, and such accesses can give rise to CPU-sourced interference. Thus, while placing I/O buffers in Level-C banks reduces DMA-sourced interference in Level-A and -B tasks, it will increase CPU-sourced interference when Level-A and -B tasks access those buffers. Similar to the IPC-management options, we incorporated these two I/O-management options into our offline optimization component and evaluate their efficacy in Sections 4.3.5 and 4.3.6.

4.3.5 Micro-Benchmark Experiments

To assess the impacts of CPU- and DMA-sourced interference under different buffer-allocation options, we experimented with various micro-benchmark programs. We used the results of these experiments to inform the task-system generation process in the schedulability study presented in Section 4.3.6. In these micro-benchmark experiments, we investigated IPC impacts using Linux's System V message-queue implementation, and I/O impacts using a USB camera and a solid-state disk (SSD). We conducted our experiments using the LLC-allocation scheme in Figure 4.15. We did not consider set-based LLC-partitioning schemes in these experiments because device drivers or IPCs may request more than four physically contiguous pages (via `kmalloc`). The set-based LLC-partitioning schemes illustrated in insets (a) and (b) of Figure 3.4 cannot allocate more than four physically contiguous pages, but the Variant 4 shown in Figure 4.15 is not limited in this way.



(a) WCETs for `load_msg()`.



(b) WCETs for `do_msg_fill()`.

Figure 4.16: Measured WCETs for Sender and Receiver.

Impact of IPC-related interference. To evaluate the SBP, CE, and CL policies from Section 4.3.4, we implemented a *Sender* task, which sends 100 fixed-sized messages, and a *Receiver* task, which receives 100 messages. We ran these tasks at Level A concurrently with a background workload at Level C, with message sizes ranging from 64 to 8,192 bytes. The Receiver was set to start executing after the Sender completes, to guarantee immediate message availability. We designed the background workload to stress the Level-C partition in the LLC and DRAM banks. We measured execution times of `load_msg()`, which allocates a message buffer and copies a message from a user buffer, and `do_msg_fill()`, which copies a message to a user buffer and frees the message buffer. We collected 10,000 samples for each considered message size for each of SBP, CE, and CL. Figure 4.16 plots the measured WCET data collected for `load_msg()` and `do_msg_fill()`.

Observation 4.10. *Sending and receiving execution times were the lowest under CL and the highest under SBP.*

Program	Description
Matrix	DIS Stressmark Suite program. Solve the equation $Ax = b$.
Synthetic	Keep writing arbitrary data to a random memory address.
Framecopy	Copy image data from the frame buffer to user-space buffer.
Yuv2gray	Convert YUV formatted image to a grayscale image.

Table 4.7: Micro-benchmark programs.

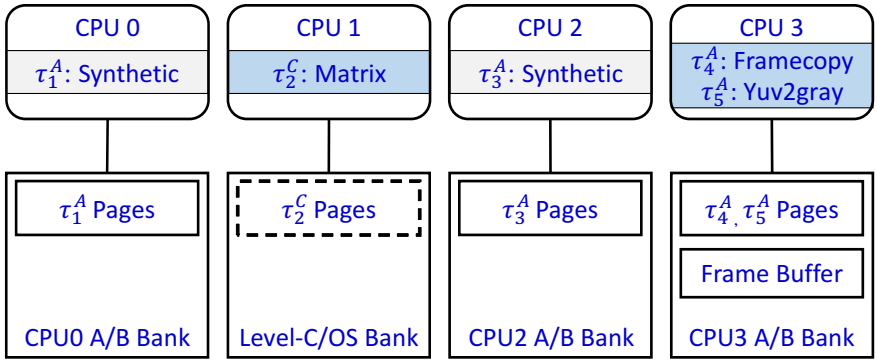
Observed CL sending times were between 2.7% and 5.4% of SBP sending times, and receiving times were 10.6% to 15.9% of SBP receiving times. Likewise, CE sending times were between 9.2% and 12.9%, and receiving times were between 29.2% and 40.5%, of the respective SBP times. These results are in accordance with results concerning user-level sharing presented in Section 4.1. CL is the best choice if the LLC is large enough to hold all message buffers. However, CL effectively reduces the LLC size for other purposes.

Impact of I/O-based CPU-sourced interference. While IPC only causes CPU-sourced interference, I/O can cause both CPU- and DMA-sourced interference. Of these, we first examine CPU-sourced interference.

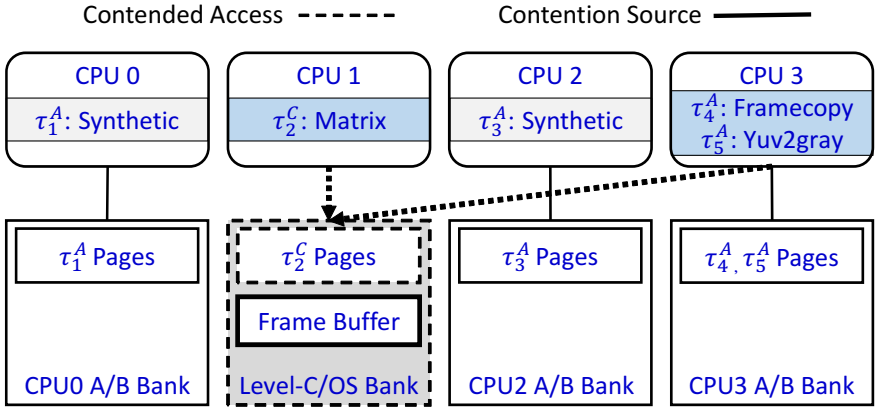
We assessed CPU-sourced interference using the micro-benchmark programs in Table 4.7. *Matrix* comes from the DIS Stressmark Suite (Musmanno, 2003). *Synthetic* continuously iterates a main loop that writes arbitrary data to randomly selected memory locations. It was designed to stress the LLC, DRAM, and other unmanaged resources. *Framecopy* and *Yuv2gray* are video-processing tasks. We modified the `v4l2` driver, which supports many USB cameras in Linux, to control where buffers are allocated for these tasks. Other non-shared data such as the matrix arrays of Matrix and the array used by Synthetic were statically allocated in accordance with our DRAM allocation strategy described in Figure 3.5.

To assess CPU-sourced interference, we considered three scenarios:

- Idle: Each micro-benchmark was run alone, with no interfering competing workload.
- Managed: The tasks were executed with the DRAM allocations shown in Figure 4.17 (a). (Each task's criticality level is denoted with a superscript in this figure.) The frame buffer was allocated in CPU 3's Level-A/B bank. This precludes task τ_4^A and τ_5^A from experiencing CPU-sourced interference.
- Unmanaged: The frame buffer was instead allocated in a Level-C bank, as shown in Figure 4.17 (b). This causes τ_4^A, τ_5^A, and τ_2^C to experience CPU-sourced interference. This scenario reflects the prior



(a) Managed scenario.



(b) Unmanaged scenario.

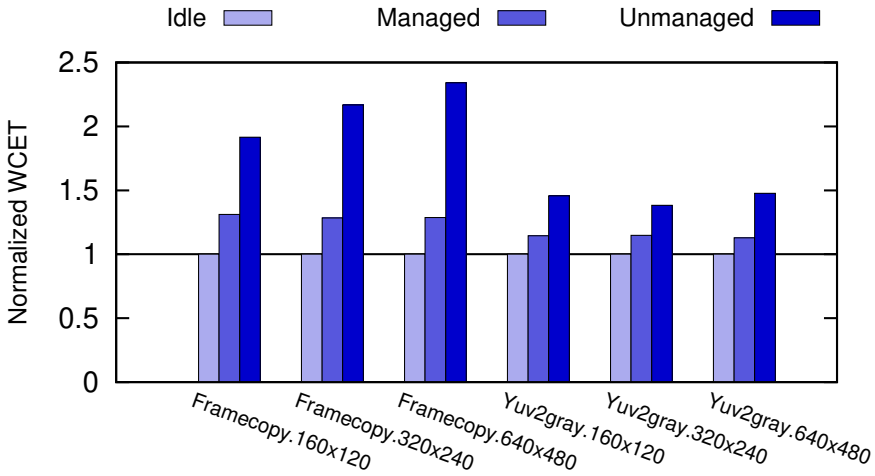
Figure 4.17: Micro-benchmark tasks and resource allocations.

MC^2 implementation, which provides no means for allocating buffers anywhere other than in Level-C banks.

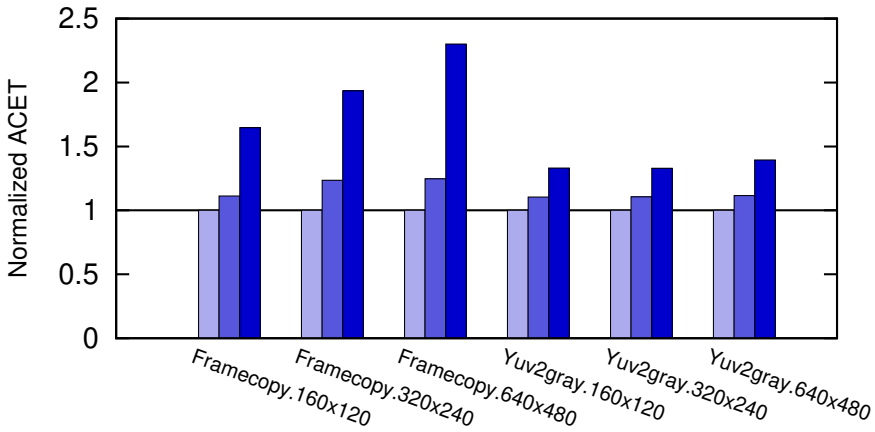
We collected 1,000 execution-time samples for each task in each scenario. Figure 4.18 presents normalized (relative to Idle) WCETs and ACETs obtained from this data for the copy and color-conversion functions of Framecopy (τ_4^A) and Yuv2gray (τ_5^A), respectively, for different frame sizes. We limit measurements to these functions as only these portions of tasks' execution times experience interference due to frame-buffer accesses. Figure 4.19 presents data showing the impact on Matrix (τ_2^C) of CPU-sourced interference caused by Yuv2gray (τ_5^A) as a function of the LLC region size allocated to Matrix (τ_2^C).

Observation 4.11. *CPU-sourced interference inflated the WCET (respectively, ACET) of copying by up to 81% (respectively, 85%), and of color conversion by up to 30% (respectively, 25%).*

This observation is supported by Figure 4.18 and confirms that DRAM interference due to I/O buffers can



(a) WCETs.



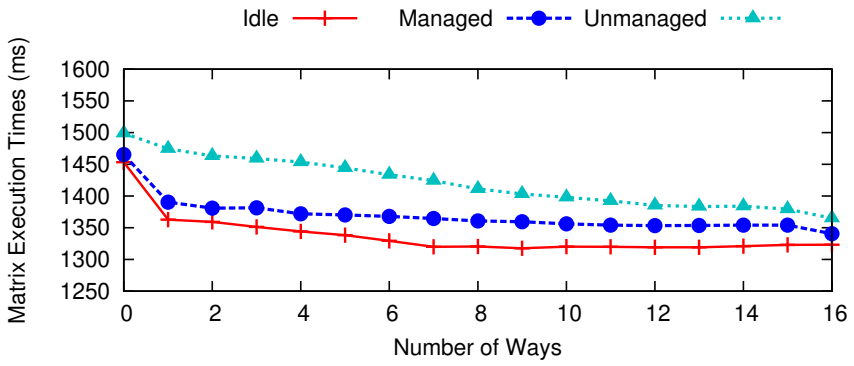
(b) ACETs.

Figure 4.18: Normalized execution times of Framecopy and Yuv2gray.

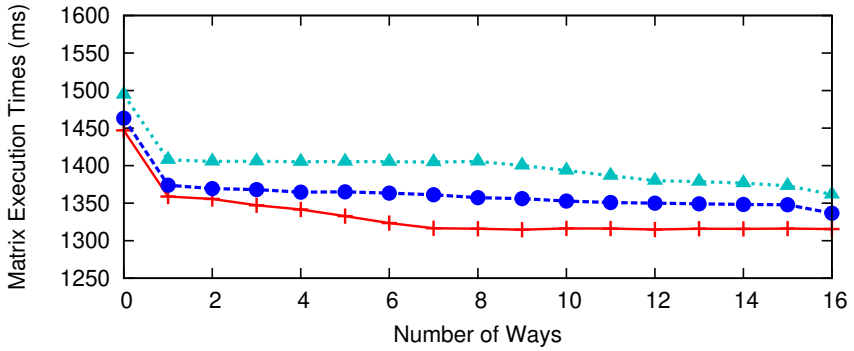
be problematic. Note that the gap between Idle and Managed is caused by interference from unmanaged resources (see Section 2.4.3). Under Managed, two instances of Synthetic (τ_1^A and τ_3^A) and the measuring task contend for unmanaged resources, while under Idle, no competing workload exists.

Observation 4.12. *CPU-sourced interference inflated Matrix's WCET (respectively, ACET) by up to 6% (respectively, 3.5%).*

This observation is supported by Figure 4.19. As expected, the WCET inflation decreases as the number of LLC ways allocated to Matrix increases. The ACET inflation does not change significantly regardless of LLC ways. Note that frame-buffer accesses do not break LLC isolation because CPUs 1 and 3 use different partitions in the LLC. The inflation seen here is due to DRAM bank conflicts.



(a) WCETs.



(b) ACETs.

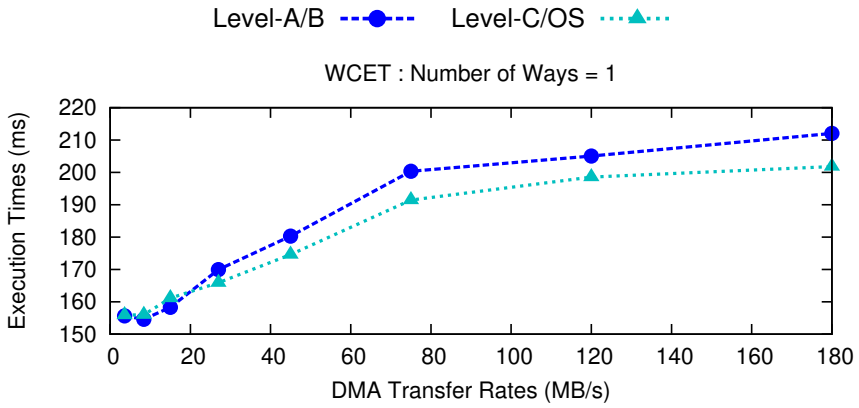
Figure 4.19: Matrix execution times as a function of allocated LLC space.

Impact of DMA-sourced interference. As discussed in Section 4.3.1, tasks can experience DMA-sourced interference whenever an I/O device sends or reads data. We conducted the following experiments to assess the impact of such interference.

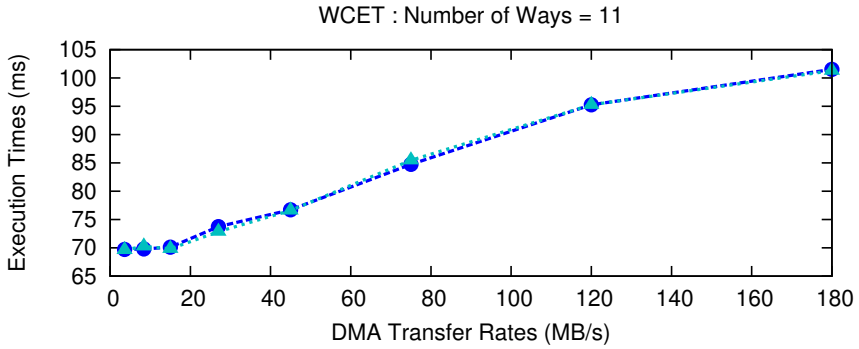
We used an SSD to generate DMA transactions at varying bandwidths. This was done by configuring a *Load-Generator* task to repeatedly read 400KB of data from the SSD at a fixed interval. Higher bandwidths were achieved by shrinking the duration of this interval. We configured Load-Generator to access the SSD using the `O_DIRECT` flag to enable DMA directly into user-allocated memory. To observe the impact of DMA-sourced interference, we measured the runtime of a variant of the Synthetic task considered earlier that repeatedly writes 256KB of arbitrary data to randomly selected memory locations.

We ran three instances of Load-Generator on separate processors, and one instance of Synthetic on the remaining processor, all at Level A. We collected 1,000 execution-time samples for Synthetic to determine how DMA-sourced interference affected it. This task system was evaluated under two scenarios:

- Level-A/B: All Load-Generator I/O buffers were allocated in Synthetic's Level-A/B bank. This



(a) Number of LLC ways = 1.



(b) Number of LLC ways = 11.

Figure 4.20: Synthetic WCETs under two allocation scenarios.

scenario represents the Managed scenario in the previous experiment, where an I/O-performing task does not experience CPU-sourced interference, but other tasks on the same processor could experience additional DMA-sourced interference caused by row-buffer conflicts.[30]

- Level-C/OS: All Load-Generator I/O buffers were allocated in the Level-C banks. This scenario represents the Unmanaged scenario in the previous experiment, where an I/O-performing task experiences CPU-sourced interference.

Figure 4.20 presents WCET data obtained for Synthetic under these two scenarios with a small (Figure 4.20 (a)) and large (Figure 4.20 (b)) LLC region allocated to it.

Observation 4.13. *Synthetic's WCET rose with increasing DMA bandwidth.*

[30]Note that the Load-Generator tasks were not actually executed on the same processor as Synthetic. Running them all on the same CPU makes obtaining accurate execution-time measurements more difficult. In the experiments considered here, we are mainly interested in the DMA-sourced interference with which Synthetic must contend, and not the processors from which that interference is induced.

This observation is supported by both insets of Figure 4.20. Because this slowdown happened under both scenarios, we can infer that it is due to DMA-sourced interference caused by unmanaged resources as described in Section 2.4.3.[31] We note that the impact of unmanaged resources increases as the DMA bandwidth increases.

Observation 4.14. *A large LLC region can hide WCET inflation due to DMA-sourced interference.*

This observation is supported by Figure 4.20 (b). As discussed in Section 4.3.1, DMA-sourced interference does not cause cache interference. Thus, if a task has sufficient cache size, row buffers misses caused by DMA do not affect the task's WCET.

Observation 4.15. *Allocating an I/O buffer in a Level-A/B bank can have a negative impact on other Level-A/B tasks that access that bank.*

Figure 4.20 (a) supports this observation. The difference between the two curves here is due to additional row-buffer conflicts that occur in the Level-A/B scenario. These conflicts become much less of an issue if a task is allocated sufficient LLC space, as seen in Figure 4.20 (b).

These results expose an interesting tradeoff. If an I/O-performing Level-A or -B task is allocated I/O buffers in its own Level-A/B bank, then it does not experience CPU-sourced interference but the resulting DMA transfers can cause other tasks that access that bank to experience additional row-buffer conflicts as part of DMA-sourced interference. These conflicts can be avoided by allocating I/O buffers in the Level-C banks, but then tasks experience CPU-sourced interference when accessing the buffers. In either case, tasks executing on any CPU can experience unmanaged resource contention as part of DMA-sourced interference. To sift through this tradeoff and other issues, we conducted a large-scale, overhead-aware schedulability study, which we discuss next.

4.3.6 Schedulability Study

In Section 4.3.5, we investigated the impact of our IPC and I/O management approaches on individual tasks, but high-level tradeoffs require investigating potential impacts on entire task systems. To assess such tradeoffs, we conducted a large-scale overhead-aware schedulability study. We begin by describing the buffer-management schemes that we considered in this study.

[31] In this experiment, the memory bandwidth was not overutilized because the maximum DRAM throughput of our platform is approximately 800MB/s. This value was obtained by running a memory profiling tool provided by NXP. This tool reported a bus load of 73% for a data-transfer rate of 602.06MB/s.

We considered five buffer-management schemes, which vary depending on how buffers for IPC and I/O are handled. To facilitate discussing these schemes, we denote them using the notation $x|y$, where x (respectively, y) indicates how IPC (respectively, I/O) buffers are handled. The possibilities for x are:

- R (random): IPC buffers are randomly assigned to DRAM banks with none of the optimizations in Section 4.3.4 applied;
- C (Level-C/OS banks used): all IPC buffers are allocated in Level-C/OS banks, again with no optimizations applied;
- O (optimized): IPC buffers are allocated using the optimization techniques in Section 4.3.4.

The possibilities for y are:

- R (random): I/O buffers are randomly assigned to banks;
- C (Level-C/OS banks used): all I/O buffers are allocated in Level-C/OS banks;
- C+A/B (all banks used): an I/O buffer used by a Level-A and -B task is assigned to its Level-A/B bank, and those used by Level-C tasks are assigned to Level-C/OS banks.

The five management schemes we considered are: R|R, C|C, O|C, C|C+A/B, and O|C+A/B. We considered R|R to illustrate the ill effects of paying no attention to OS buffer-assignment issues. C|C reflects the choice of partitioning the OS from HRT tasks in a coarse-grained way that prevents OS data structures from existing in a Level-A/B bank (a similar partitioning is possible on most RTOSs today). O|C, C|C+A/B, and O|C+A/B provide varying degrees of fine-grained partitioning in which certain OS data structures are allowed to exist in a Level-A/B bank.

Note that the two choices of C and C+A/B for I/O give two extremes in a spectrum of choices: in the former, all Level-A/B I/O buffers are allocated in Level-C banks, while in the latter, they are all allocated in a Level-A/B bank. It would have been interesting to allow *per-buffer* choices, *i.e.*, allow some Level-A/B I/O buffers to be allocated in Level-C banks and others in a Level-A/B bank. However, while allowing such choices in the context of a single task system is not problematic, doing so is quite impractical at the scale of our schedulability study. Moreover, this change would have greatly complicated explaining our results because it introduces an additional dimension (namely, the fraction of Level-A/B I/O buffers allocated in Level-C/OS vs. Level-A/B banks). Due to the scale of our schedulability study, we have no choice but to

avoid such complications. (Given this choice, the "best" schedulability curves we present actually *lower bound* the best that could be obtained from our allocation framework if per-buffer decisions were allowed.)

Modeling IPC and devices. The intent of our study is *not* to delve into complicated precedence-related schedulability issues but rather to demonstrate the effects of DRAM and LLC allocation policies. To avoid such complications, we assumed that tasks that communicate via IPC share a common period, like our user-level IPC mechanism discussed in Section 4.1. For similar reasons, we assumed that I/O-consuming Level-A and -B tasks simply poll for I/O data (polling is not necessary for Level-C tasks because they are sporadic).

We considered the disk and camera mentioned in Section 4.3.2 as exemplars of two categories of I/O devices. The former only causes interference when data is pushed by the device or accessed by the task, while the latter involves intermediate steps that cause additional interference. We assumed that intermediate buffers (USB packet buffers in the camera example) remain in Level-C/OS banks under C+A/B to prevent the OS from inducing CPU-sourced interference on Level-A and -B tasks while data is copied between buffers.

Task-system generation. We generated task systems by incorporating I/O sources into a procedure used in Chapter 3. Under this procedure, the following stepwise process is used to generate a task system, and each step is guided by measurement data presented in Section 4.3.5.

Step 1 Choose distributions from the first four categories in Table 4.8. The chosen distributions are used to generate a *preliminary task system*, which is modified below to introduce IPC and I/O.

Step 2 Select distributions from Category 5 in Table 4.8. Sample these distributions to determine the size of IPC buffers.

Step 3 Select a distribution (one for all criticality levels) from Category 6. Sample this distribution to determine the level of I/O bandwidth and assign buffers to tasks until this level is met.

Step 4 Select distributions from Category 7. Sample these distributions to determine the percentage of I/O tasks whose buffers directly receive data via DMA (like the SSD). The remaining I/O tasks perform intermediate copies (like the USB camera).

Note that the above procedure does not determine task execution costs. A task's execution cost is derived from its period and utilization and represents a bound on the time required for it to complete a job in an idle

Category	Choice	Level A	Level B	Level C
1: Criticality Utilization Ratios	C-Light	[35, 45)	[35, 45)	[10, 30)
	C-Heavy	[10, 30)	[10, 30)	[50, 70)
	All-Mod.	[28, 39)	[28, 39)	[28, 39)
2: Period (ms)	Short	{12, 24}	{24, 48}	[12, 100)
	Moderate	{20, 40}	{40, 80}	[20, 100)
	Long	{48, 96}	{96, 192}	[50, 500)
3: Task Utilization	Light	[0.001, 0.03)	[0.001, 0.05)	[0.001, 0.1)
	Medium	[0.02, 0.1)	[0.05, 0.2)	[0.1, 0.4)
	Heavy	[0.1, 0.2)	[0.2, 0.4)	[0.4, 0.6)
4: Max Reload Time	Light	[0.01, 0.1)	[0.01, 0.1)	[0.01, 0.1)
	Heavy	[0.25, 0.5)	[0.25, 0.5)	[0.25, 0.5)
5: IPC Size (bytes)	Small	{128, 256}	{128, 256}	{128, 256}
	Large	{4096, 8192}	{4096, 8192}	{4096, 8192}
6: I/O Bandwidth (MB/s)	Low	[0, 20]		
	Medium	[40, 60]		
	High	[180, 200]		
7: Direct I/O Tasks (%)	Few	[0, 30]	[0, 30]	[0, 30]
	Many	[70, 100]	[70, 100]	[70, 100]

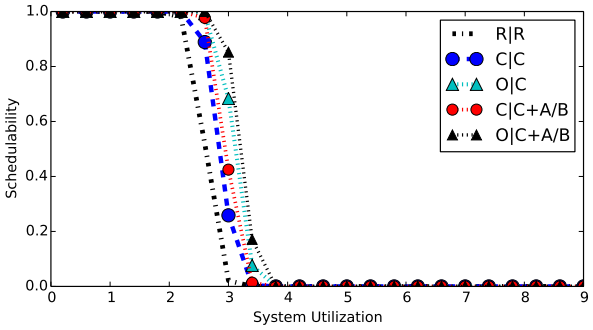
Table 4.8: Task-set parameters and distributions.

system with the full LLC available and no other competing work (including I/O). Execution costs under other management options and assumptions are determined using the idle-system cost and micro-benchmark data pertaining to way allocations, I/O buffer allocations, and I/O bandwidths.

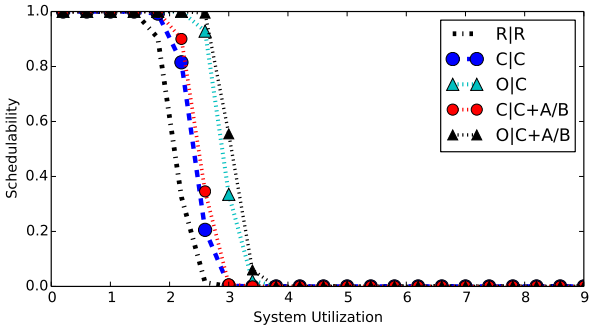
Overhead accounting. Any Level-A/B task that accesses a kernel data structure stored in Level-C banks requires additional execution time. The increases were informed by Figure 4.16 for IPC and Figure 4.18 for I/O buffers. For the latter, we assumed that each I/O-performing task begins with a system call that copies I/O data from kernel-managed buffers into local buffers. This assumption standardizes the interference to the duration of the copy rather than accounting for all possible buffer-access patterns. Additional CPU-sourced interference also occurs in the R|R scheme, which allows I/O buffers to be allocated in any Level-A/B bank.

We accounted for interrupts for all schemes by inflating PETs of Level-A/B tasks on CPU 0, where interrupts are handled. For Level-C tasks, we used interrupt-accounting techniques from Brandenburg et al. (2011), which are applicable under global scheduling.

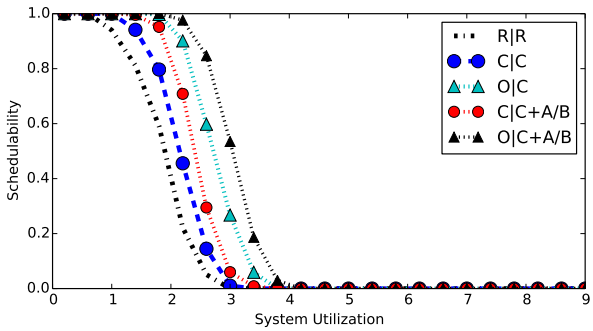
Schedulability results. We considered all possible combinations presented in Table 4.8. We generated sufficient task systems to estimate mean schedulability within ± 0.05 with 95% confidence with at least 100



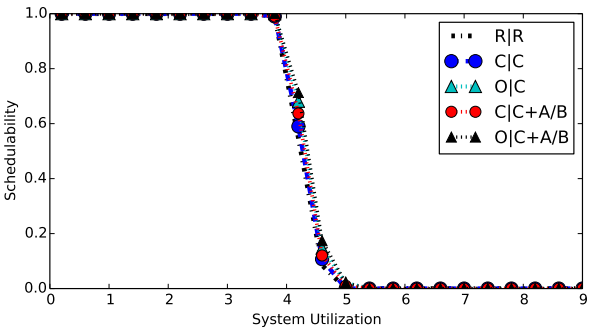
(a) (C-Light, Long, Light, Light, Small, High, Many).



(b) (C-Light, Long, Light, Light, Large, High, Many).



(c) (C-Light, Short, Med., Heavy, Large, High, Many).



(d) (All-Mod., Long, Heavy, Heavy, Small, Low, Many).

Figure 4.21: Representative schedulability plots.

and at most 2,000 task systems. In total, we evaluated 648 schedulability plots taking over 38 days of CPU time to compute. Figure 4.21 shows four representative plots. The full set of plots is presented in Appendix D.

Each schedulability plot corresponds to a single scenario, which represents a combination of distribution choices. We now state several observations that follow from the full set of plots. We illustrate these observations using the plots in Figure 4.21.

Observation 4.16. *Coarse I/O and IPC partitioning is beneficial.*

This can be observed qualitatively in insets (a), (b), and (c) of Figure 4.21, where the R|R curve is always below that for any other management scheme. Across all considered scenario, only one scenario presented in Figure 4.21 (d) showed no improvement to schedulability.

Observation 4.17. *IPC-buffer optimizations outperform coarse partitioning.*

This behavior can be observed by comparing the C|C and O|C curves in insets (a), (b), and (c) of Figure 4.21. As expected, the benefit of IPC-buffer optimizations is proportional to the amount of data shared with the OS.

Observation 4.18. *Reducing CPU-sourced I/O interference is more important than reducing DMA-sourced interference.*

Figure 4.21 (c) shows such a scenario that C|C+A/B is better than C|C. This observation addresses the tradeoff discussed in Section 4.3.4. This means that CPU-sourced interference from accessing I/O buffers in Level-C banks is usually worse than the DMA-sourced interference from placing I/O buffers in a Level-A/B bank. Unlike in IPC-buffer optimization, reducing CPU-sourced interference in I/O buffers leads to increased DMA-sourced interference.

Observation 4.19. *Used in conjunction, fine-grained I/O and IPC management outperform all other schemes in 99% of the considered scenarios.*

This observation is supported by insets (a), (b), and (c) of Figure 4.21, where O|C+A/B outperforms all other schemes. In 1% of considered scenarios, schedulability under O|C outperformed O|C+A/B by less than 1%.

From these observations, we can conclude that our extensions to MC^2 significantly improve the schedulability of systems requiring OS-supported IPC and device I/O. In some cases, such as Figure 4.21 (c), the improvement encompasses nearly an entire core's worth of additional computing capacity. This is even more impressive given that tasks spend only a relatively small proportion of their time accessing IPC and I/O buffers.

4.4 Summary

In this chapter, we discussed the impact of introducing sharing in MC^2. We covered three types of sharing: data sharing among tasks, the usage of shared libraries, and data sharing between the OS and user-level tasks. We presented techniques for mitigating capacity loss caused by the introduction of sharing. Our extension to MC^2 allows tasks to share data among tasks or between the OS and user-level tasks. In addition, the support for shared libraries in MC^2 brings the benefit of saving memory, which improves schedulability when considering memory as a constrained resource. We investigated the impact of such interference on individual tasks by conducting micro-benchmark experiments. We also conducted large-scale overhead-aware schedulability studies to evaluate various tradeoffs and system-wide impacts.

CHAPTER 5: CONCLUSION

The main objective of the research presented in this dissertation was to show that the capacity loss due to contention for shared hardware resources on multicore platforms can be mitigated by combining mixed-criticality analysis and hardware-management techniques. To demonstrate this, we designed and implemented several techniques in MC^2: criticality-aware resource allocation strategies, LLC and DRAM bank partitioning techniques, shared-buffer management techniques, per-partition library replicas to support shared libraries, and optimizing kernel buffers for I/O and IPC. We evaluated these techniques by conducting micro-benchmark experiments and large-scale overhead-aware schedulability studies.

The results of this dissertation were obtained as part of a collaborative effort with Micaiah Chisholm. His LP-based optimization framework, presented in Chisholm et al. (2015), was used extensively in the schedulability studies presented herein. He also contributed to the experimental framework (particularly the methods employed for generating task systems at random) used in these studies.

In the following, we summarize our results (Section 5.1), briefly discuss other contributions not included in this dissertation (Section 5.2), and discuss avenues for future work (Section 5.3).

5.1 Summary of Results

In Chapter 1, we presented the following thesis statement.

> *"The capacity loss of real-time systems on multicore mixed-criticality systems can be mitigated by combining mixed-criticality analysis and hardware-management techniques, particularly by managing the shared LLC and dynamic random-access memory (DRAM) memory. Such a combination approach can be designed to mitigate hardware contention caused by data sharing between tasks, shared libraries, and sharing between the kernel and user space in order to support practical real-time workloads."*

To support this thesis, we have developed hardware-management techniques in MC^2 and we have

investigated the efficacy of the developed techniques through micro-benchmark experiments and schedulability studies. We now review the contributions presented in this dissertation.

Enabling hardware management in MC^2. Adopting multicore platforms in safety-critical applications results in severe capacity loss due to analysis pessimism for certification. To mitigate such capacity loss, two orthogonal approaches have been proposed: hardware-management techniques and mixed-criticality analysis. To address the capacity loss of multicore platforms, we proposed the combination of both approaches in this dissertation. In Chapter 3, we presented a significant extension to the MC^2 framework that provides LLC and DRAM-bank isolation and that affords coarse-grained OS isolation to higher-criticality tasks. We implemented way-based and set-based LLC partitioning in MC^2, which can be combined to flexibly create LLC areas that can be designated for the sole use of certain tasks. We also implemented DRAM-bank isolation techniques by leveraging page coloring. These hardware-management techniques enable highly configurable resource allocations in MC^2, which allows us to explore sharing and isolation tradeoffs in a criticality-cognizant way. We also presented the results from micro-benchmark experiments and a large-scale overhead-aware schedulability study. We collected 8GB of trace data for micro-benchmark programs to evaluate our hardware-management techniques. We also generated millions of randomly generated task systems for the schedulability study, which took 18 CPU-days of computation. From this extensive evaluation, we showed that the combination of hardware-management techniques and mixed-criticality provisioning can reduce capacity loss significantly.

Support for data sharing among tasks. The MC^2 framework presented in Chapter 3 was not sufficient to support real-world workloads due to a lack of support data sharing. In Section 4.1, we extended our MC^2 framework to support data sharing among tasks. We showed how the introduction of data sharing can break the isolation properties provided in MC^2. To mitigate interference caused by the introduction of data sharing among tasks, we developed inter- and intra-criticality data-sharing mechanisms. We also implemented such data-sharing mechanisms via shared-memory buffers, and conducted a large-scale schedulability study to evaluate our techniques, which took roughly 27 CPU-days of computation. From the results of this schedulability study, we showed that our shared-buffer management techniques can effectively reclaim capacity loss due to the introduction of data sharing among tasks.

Techniques to allow shared libraries. Another common source of sharing is the usage of shared libraries. However, such sharing can be obviated by statically linking libraries even though statically linked tasks require more memory space. In our previous studies, we assumed that all tasks were always statically linked because we did not consider memory as a constrained resource. In practice, however, the combined memory footprint of all tasks and the OS must fit within the provided physical memory. In Section 4.2, we considered memory as a constrained resource, and we proposed per-partition library replicas to support shared libraries. We maintain up to m+1 library replicas per library, where m is the number of processors in a system: one per-processor copy is shared by all Level-A and -B tasks running on that processor, and another copy is shared by Level-C tasks on all processors. We discussed several approaches to implement per-partition library replicas, and we implemented a system call and relevant data structures in the kernel because this kernel-level approach has advantages over other approaches.

When considering memory as a constrained resource, a disadvantage of disabling bank interleaving is exposed: only a quarter of the pages in each Level-A/B bank is available. Thus, we considered the interleaved approach as an alternative to the non-interleaved approach. The new interleaved approach eliminates bank isolation from the OS at Levels A and B because the OS pages are spread across all banks when the bank interleaving is enabled. To evaluate the tradeoffs between these two approaches, we conducted a large-scale overhead-aware schedulability study, which took roughly 25 CPU-days of computation. From the results of this schedulability study, we observed that the use of static linking significantly degraded schedulability by wasting memory space. We also showed that our per-partition library replicas for dealing with shared libraries can improve schedulability.

Mitigating OS-induced interference. We considered data sharing among tasks in Section 4.1 and read-only sharing through the usage of shared libraries in Section 4.2. However, these are not the only sources of sharing in a system. The sharing of data between the OS and user-level tasks occurs when tasks invoke OS services for IPC or I/O. In Section 4.3, we discussed how OS-induced sharing introduces interference, and proposed approaches to optimize kernel-buffer allocations to mitigate such interference. To control kernel-buffer allocations, we extended our MC^2 framework to enable dynamic memory allocation.

We described two types of interference due to sharing between the OS and user-level tasks: CPU-sourced and DMA-sourced interference. CPU-sourced interference occurs when tasks suffer interference due to other tasks concurrently accessing a DRAM bank or the LLC. DMA-sourced interference occurs when a device

transfers data into DRAM banks directly by using DMA. To understand how such interference occurs in practice, we investigated the issue of data transfer using two devices as exemplars: a secondary storage disk that does not require copying data from the kernel to user space and a USB camera that copies frame data from the kernel to user space. To mitigate OS-induced interference, we used the buffer-management techniques presented in Section 4.1 for IPC buffers, and proposed fine-grained buffer-allocation techniques for device I/O buffers. We also evaluated the presented techniques by conducting micro-benchmark experiments and a large-scale overhead-aware schedulability study. Our micro-benchmark experiments showed that our approaches effectively reduced OS-induced interference. From the results of the schedulability study, which took roughly 38 CPU-days of computation, we showed that our kernel-buffer management techniques improved schedulability over unmanaged buffers.

5.2 Other Related Work

This section provides a brief summary of other related work done by the author during his graduate studies.

Supporting mode changes while providing hardware isolation.[32] The hardware-management techniques considered in this dissertation are limited to static task systems that never change at runtime. In reality, many safety-critical applications often transition among different functional modes as we mentioned in Section 4.2.5. Allowing multiple modes to exist can greatly complicate shared-hardware management. As tasks from inactive modes consume memory, DRAM allocations are the main issue when considering multi-mode systems. The DRAM region a task can access also determines the region of the LLC it accesses, which increases complexities in allocating DRAM and LLC space. In addition, tasks that are *shared* across multiple modes complicate the issue of supporting multiple modes. To further complicate matters, a shared task could potentially be of different criticalities in different modes. To address these issues, we proposed several resource-allocation approaches and evaluated these approaches via a large-scale overhead-aware schedulability study (Chisholm et al., 2017). We also added a mode-change protocol in the MC^2 task scheduler and conducted case studies to verify the mode-change protocol.

[32] Details of this contribution have been published in the following paper:
Chisholm, M., Kim, N., Tang, S., Otterness, N., Anderson, J., Smith, F. D., and Porter, D. (2017). Supporting mode changes while providing hardware isolation in mixed-criticality multicore systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 58–67.

Recovering from overload in multicore mixed-criticality systems.[33] When any job at Level C overruns its Level-C PET, the system at Level C may be *overloaded*, compromising timing guarantees at Level C. Under MC^2, a task may have its per-job response times permanently increased as a result of even one overload event, and multiple overload events could cause such increases to build up over time. To recover from transient overload conditions, we must alter scheduling decisions. We proposed a scheme that modifies scheduling decisions at Level C by scaling release times between subsequent jobs (Erickson et al., 2015). We leveraged a notion of *virtual time* to effectively reduce task utilizations. During normal operation of the system, actual time and virtual time progress at the same rate. However, after an overload occurs, the scheduler adjusts the speed of the virtual clock. As a result, virtual time progresses more slowly, and new releases of jobs are delayed. We implemented this scheme in MC^2 and evaluated it under several overload scenarios.

Minimizing response times of automotive dataflows on multicore platforms.[34] Graph-based software architectures, referred to as *dataflow* architectures, are common to applications that process streams of data or events. These architectures are prevalent in cyber-physical applications that require timing constraints such as sensing components of automotive systems. Typically, such applications are modeled as DAGs, with nodes denoting tasks and edges denoting producer/consumer relationships. We considered the problem of scheduling sporadic DAGs on multicore platforms (Elliott et al., 2014). We integrated dataflow analysis proposed by Liu and Anderson (2011) with job-splitting techniques proposed by Erickson and Anderson (2013) to lessen DAG end-to-end response time bounds. We also proposed a cache-aware heuristic to assign tasks to clusters that promotes cache reuse and reduces communication costs. We demonstrated that cache-aware techniques, coupled with clustered processor scheduling, can yield better timing properties than naive partitioned or global scheduling via a overhead-aware schedulability study and runtime experiments.

[33] Details of this contribution have been published in the following paper:
Erickson, J., Kim, N., and Anderson, J. (2015). Recovering from overload in multicore mixed-criticality systems. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium*, pages 775–785.

[34] Details of this contribution have been published in the following paper:
Elliott, G., Kim, N., Erickson, J., Liu, C., and Anderson, J. (2014). Minimizing Response Times of Automotive Dataflows on Multicore. In *Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

5.3 Future Work

We now discuss future work and promising directions of research that could improve or build upon the results presented in this dissertation.

5.3.1 Richer Data Sharing Models

In Section 4.1, we considered data sharing between one producer and one consumer. However, in practice, multiple producer/consumer relationships may exist. An interesting avenue for future work would be to remove this limitation.

5.3.2 Considering Other Platforms

We are interested in embedded platforms that are more recent than the ARM Cortex-A9. For embedded use cases, ARM, Intel, and PowerPC seem to be the major architectures, so we intend to explore three representative platforms in the future.

ARMv8 architecture. The ARM Cortex-A9 platform considered in this dissertation has a 32-bit addressing barrier that prevents extending beyond 4GB of physical memory. To lift this limitation, ARM released the ARMv8 architecture, its new 64-bit extension, in 2011. Within the ARMv8 family, a good candidate platform to consider is the i.MX8 platform, announced in 2017 by NXP. This platform was designed for safety-critical systems such as driver-information systems and unmanned aerial vehicles. The i.MX8 has a multi-cluster multicore architecture as shown in Figure 5.1.

On ARMv8 processors, lockdown registers are no longer available for cache management. As an alternative to cache management operations, scratchpad memory (SPM) is available on the i.MX8. SPM is low-latency memory under full programmer control that can be used without the unpredictability seen in caches. Recently, Tabish et al. (2016) proposed a technique to reduce shared-resource interference with SPM. A scratchpad memory could be used to implement the techniques proposed in this dissertation.

With respect to hardware management, the i.MX8 has advantages and disadvantages compared to the Cortex-A9. The i.MX8 provides significantly more LLC and DRAM space then the Cortex-A9, so we have more choices in allocating LLC and DRAM banks. For example, we could consider a dedicated bank approach for shared data as mentioned in Section 4.1. We can also provide more predictability for HRT

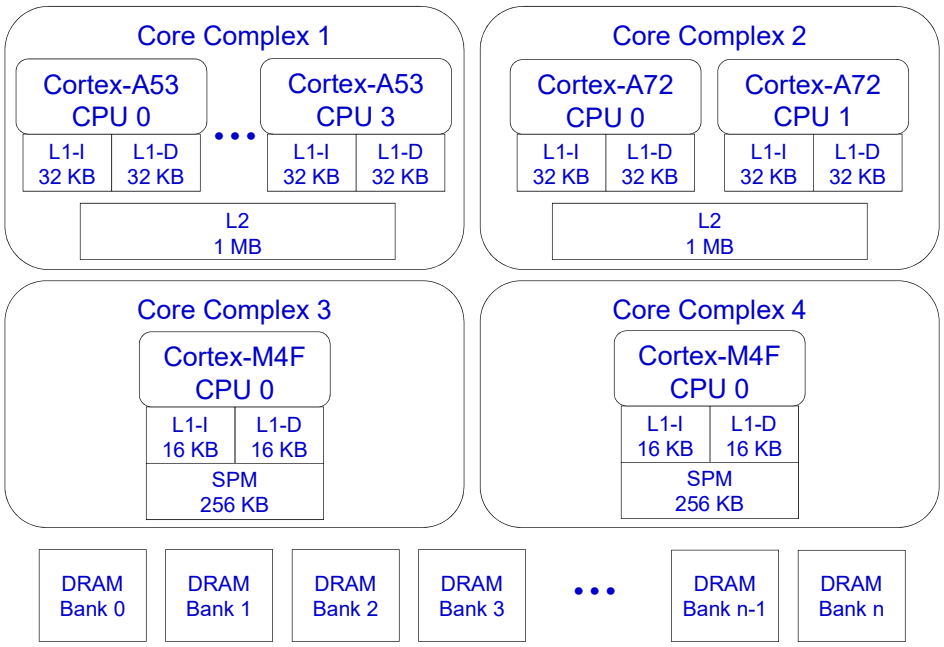


Figure 5.1: NXP i.MX8.

tasks using SPM. On the other hand, because processors are partitioned into multiple clusters on the i.MX8, L2 caches may be subject to additional delays due to cache coherency protocols. Also, on the Cortex-A9, the allocation of DRAM and LLC space had to be orchestrated assuming only one shared L2 cache. With multiple L2 caches on the i.MX8, such allocations may become much more complicated.

In addition, a multi-cluster architecture has never been considered before in work on mixed-criticality systems. To support Level-C tasks in MC^2 on a multi-cluster platform, we could continue to use G-EDF, but migrating tasks across clusters may create additional complications and overheads. An alternative is to use clustered EDF (C-EDF) instead. Additionally, if data sharing across clusters is required, then it is not clear how such sharing can be supported while providing a reasonable notion of isolation. The lack of a cache that is shared by all processors obviates using some of the techniques for supporting sharing proposed in this dissertation.

Intel-based platform. Recently, Intel introduced a feature called *cache allocation technology (CAT)* that can be used instead of cache lockdown registers. A particularly interesting platform to consider with CAT is the Intel E5-2658A v3, which is explicitly marketed for embedded systems. Figure 5.2 shows the cache and DRAM organization of this platform. The E5-2658A has 12 processors supporting two hardware threads each. Intel's CAT supports way-based cache allocation. On the E5-2658A, CAT is applied to the L3 cache,

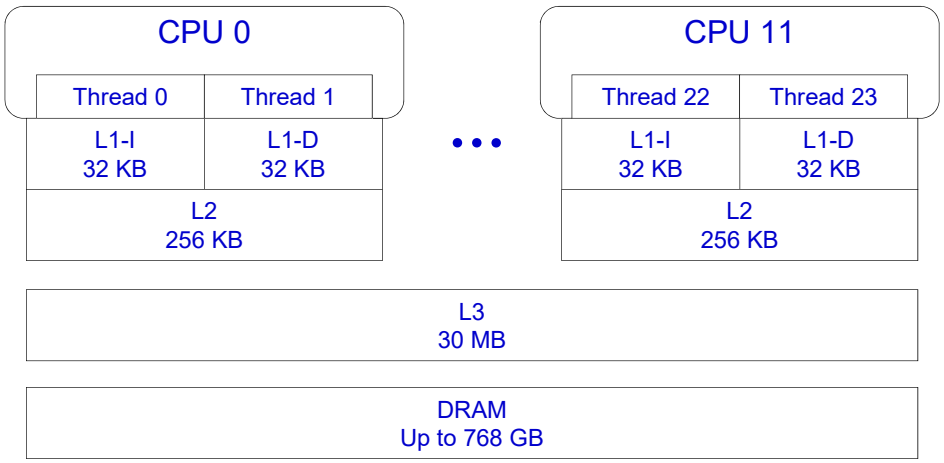


Figure 5.2: Intel E5-2658A v3.

which is a 20-way 30MB shared cache. This means that an Intel version of MC^2 can continue to support way-based partitioning. However, Intel uses a hash function to effectively randomize the mapping from memory addresses to DRAM banks and cache lines. This means that supporting DRAM bank isolation or set-based partitioning on an Intel platform is more challenging. One possible way forward is to reverse engineer the hash functions (Scolari et al., 2016).

One aspect of the E5-2658A's design that greatly interests us is its support for hardware threads. In the early days of work on real-time multiprocessor scheduling and schedulability analysis, hardware threading was nearly universally precluded, because threads that run concurrently on a processor can interfere with each other in ways that are difficult to predict. However, we believe that such interference may be acceptable for Level C on MC^2 because Level C is not afforded HRT guarantees.

PowerPC architecture. PowerPC architectures have long been used in embedded computing applications (*e.g.*, this architecture is used on the Curiosity Mars rover). A particularly interesting PowerPC platform to consider is the T4240, manufactured by NXP, the basic architecture of which is shown in Figure 5.3. The T4240 is a multi-cluster architecure like the i.MX8, with three clusters with four processors each, for a total of 12 processors. It also supports hardware threads like the E5-2658A. Moreover, it provides some hardware support for cache partitioning.

Research issues pertaining to multi-clustered architectures and the availability of hardware threads described above with respect to the i.MX8 and E5-2658A apply to the T4240 as well. The T4240 supports programmable registers for L2 cache partitioning for both sets and ways similar to the Cortex-A9, so much of

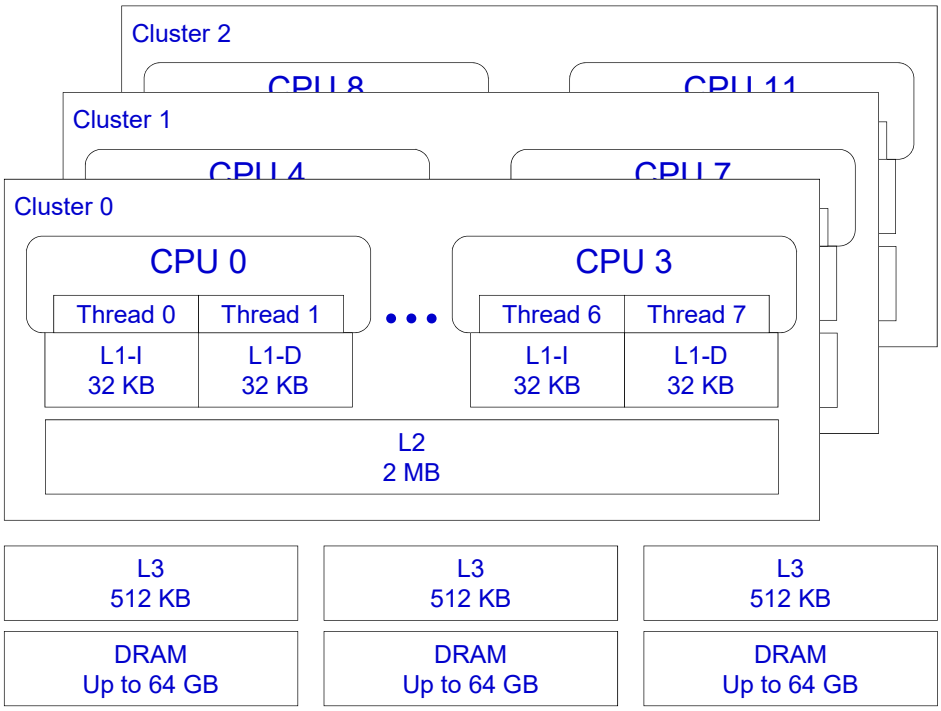


Figure 5.3: PowerPC T4240.

our previous cache partitioning techniques can be potentially carried over. However, further consideration is needed for the T4240 because of its unique L2 cache architecture. Each per-cluster L2 cache is divided into four banks to allow concurrent accesses by the processors within each cluster, but this also serves as an additional source of contention that we would like to evaluate. Another potential source of contention that we need to consider occurs when different clusters access each other's L2 caches. In particular, a request that results in a L2 cache miss for a cluster may be serviced by a different cluster's L2 cache. This incurs a latency penalty, but is still faster than going to L3 cache or DRAM. We can eliminate these sources of contention by partitioning the banks to processors (and caches to clusters), but this limits the maximum size of cache partitions. A schedulability study should be done to evaluate this tradeoff.

BIBLIOGRAPHY

Altmeyer, S., Douma, R., Lunniss, W., and Davis, R. (2014). Evaluation of cache partitioning for hard real-time systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, pages 15–26.

Anderson, J., Baruah, S., and Brandenburg, B. (2009). Multicore operating-system support for mixed criticality. In *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*.

Anderson, J. and Holman, P. (2000). Efficient pure-buffer algorithms for real-time systems. In *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications*, pages 57–64.

ARM Limited (2009). *Application Note 228: Implementing DMA on ARM SMP Systems*. `http://infocenter.arm.com/help/topic/com.arm.doc.dai0228a/DAI228A_DMA_on_SMP_systems.pdf`.

Baker, T. and Shaw, A. (1989). The cyclic executive model and Ada. *Real-Time Systems*, 1(1):7–25.

Bienia, C. (2011). *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, Princeton, NJ.

Brandenburg, B. (2011). *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC.

Brandenburg, B., Leontyev, H., and Anderson, J. (2011). An overview of interrupt accounting techniques for multiprocessor real-time systems. *Journal of Systems Architecture*, 57(6):638–654.

Bui, B., Caccamo, M., Sha, L., and Martinez, J. (2008). Impact of cache partitioning on multi-tasking real time embedded systems. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 101–110.

Calandrino, J., Leontyev, H., Block, A., Devi, U., and Anderson, J. (2006). LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 111–126.

Certification Authorities Software Team (2016). Position paper CAST-32A: Multi-core processors.

Chisholm, M., Kim, N., Tang, S., Otterness, N., Anderson, J., Smith, F., and Porter, D. (2017). Supporting mode changes while providing hardware isolation in mixed-criticality multicore systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 58–67.

Chisholm, M., Kim, N., Ward, B., Otterness, N., Anderson, J., and Smith, F. D. (2016). Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In *Proceedings of the 37th IEEE International Real-Time Systems Symposium*, pages 57–68.

Chisholm, M., Ward, B., Kim, N., and Anderson, J. (2015). Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *Proceedings of the 36th IEEE International Real-Time Systems Symposium*, pages 305–316.

Devi, U. and Anderson, J. (2005). Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 330–341.

Dhall, S. and Liu, C. (1978). On a real-time scheduling problem. *Operations Research*, 26(1):127–140.

Elliott, G., Kim, N., Erickson, J., Liu, C., and Anderson, J. (2014). Minimizing response times of automotive dataflows on multicore. In *Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

Erickson, J. (2014). *Managing Tardiness Bounds and Overload in Soft Real-Time Systems*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC.

Erickson, J. and Anderson, J. (2013). Reducing tardiness under global scheduling by splitting jobs. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 14–24.

Erickson, J., Kim, N., and Anderson, J. (2015). Recovering from overload in multicore mixed-criticality systems. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium*, pages 775–785.

Gorman, M. (2004). Describing physical memory. In *Understanding the Linux Virtual Memory Manager*, chapter 2. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Guo, D. and Pellizzoni, R. (2017). A requests bundling DRAM controller for mixed-criticality systems. In *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 247–258.

Hassan, M., Patel, H., and Pellizzoni, R. (2015). A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 307–316.

Herman, J., Kenna, C., Mollison, M., Anderson, J., and Johnson, D. (2012). RTOS support for multicore mixed-criticality systems. In *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 197–208.

Hsueh, C.-W. and Lin, K.-J. (2001). Scheduling real-time systems with end-to-end timing constraints using the distributed pinwheel model. *IEEE Transactions on Computers*, 50(1):51–66.

Kessler, R. and Hill, M. (1992). Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10:338–359.

Kim, J., Yoon, M., Bradford, R., and Sha, L. (2014). Integrated modular avionics (IMA) partition scheduling with conflict-free I/O for multicore avionics systems. In *Proceedings of the 38th IEEE Annual Computer, Software, and Applications Conference*, pages 321–331.

Kim, N., Chisholm, M., Otterness, N., Anderson, J., and Smith, F. D. (2017a). Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 223–234.

Kim, N., Tang, S., Otterness, N., Anderson, J., Smith, F. D., and Porter, D. (2018). Supporting I/O and IPC via fine-grained OS isolation for mixed-criticality real-time tasks. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pages 191–201.

Kim, N., Ward, B., Chisholm, M., Anderson, J., and Smith, F. D. (2017b). Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *Real-Time Systems*, 53(5):709–759.

Kim, N., Ward, B., Chisholm, M., Fu, C., Anderson, J., and Smith, F. D. (2016). Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *Proceedings of the 22nd IEEE Real-Time Embedded Technology and Application's Symposium*, pages 1–12.

Kirk, D. (1989). SMART (strategic memory allocation for real-time) cache design. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 229–237.

Knowlton, K. (1965). A fast storage allocator. *Communications of the ACM*, 8(10):623–624.

Knuth, D. (1968). *The Art of Computer Programming*. Addison-Wesley.

Lakshmanan, K., Kato, S., and Rajkumar, R. (2010). Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 259–268.

LITMUS$^{\mathrm{RT}}$ Project (2018). LITMUS$^{\mathrm{RT}}$: Linux testbed for multiprocessor scheduling in real-time systems. `http://www.litmus-rt.org/`.

Liu, C. and Anderson, J. (2010). Supporting soft real-time dag-based systems on multiprocessors with no utilization loss. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 3–13.

Liu, C. and Anderson, J. (2011). Supporting graph-based real-time applications in distributed systems. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 143–152.

Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61.

López, J., Díaz, J., and García, D. (2004). Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68.

Mancuso, R., Dudko, R., Betti, E., Cesati, M., Caccamo, M., and Pellizzoni, R. (2013). Real-time cache management framework for multi-core architectures. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 45–54.

Mok, A. (1983). *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA.

Mollison, M., Erickson, J., Anderson, J., Baruah, S., and Scoredos, J. (2010). Mixed criticality real-time scheduling for multicore systems. In *Proceedings of the 7th IEEE International Conference on Computer and Information Technology*, pages 1864–1871.

Mueller, F. (1995). Compiler support for software-based cache partitioning. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 125–133.

Muench, D., Paulitsch, M., and Herkersdorf, A. (2014). Temporal separation for hardware-based I/O virtualization for mixed-criticality embedded real-time systems using PCIe SR-IOV. In *Proceedings of the Workshop on Architecture of Computing Systems*, pages 1–7.

Musmanno, J. (2003). Data intensive systems (DIS) benchmark performance summary.

Panchamukhi, S. and Mueller, F. (2015). Providing task isolation via TLB coloring. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–13.

Pellizzoni, R., Bui, B., Caccamo, M., and Sha, L. (2008). Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 221–231.

Pellizzoni, R. and Caccamo, M. (2010). Impact of peripheral-processor interference on WCET analysis of real-time embedded systems. *IEEE Transactions on Computers*, 59(3):400–415.

Saifullah, A., Agrawal, K., Lu, C., and Gill, C. (2011). Multi-core real-time scheduling for generalized parallel task models. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 217–226.

Scolari, A., Bartolini, D. B., and Santambrogio, M. D. (2016). A software cache partitioning system for hash-based caches. *ACM Transactions on Architecture and Code Optimization*, 13(4):57:1–57:24.

Seetanadi, G., Camara, J., Almeida, L., Arzen, K., and Maggio, M. (2017). Event-driven bandwidth allocation with formal guarantees for camera networks. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 243–254.

Tabish, R., Mancuso, R., Wasly, S., Alhammad, A., Phatak, S., Pellizzoni, R., and Caccamo, M. (2016). A real-time scratchpad-centric OS for multi-core embedded systems. In *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 1–11.

Valsan, P., Yun, H., and Farshchi, F. (2016). Taming non-blocking caches to improve isolation in multicore real-time systems. In *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 1–12.

Vestal, S. (2007). Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 239–243.

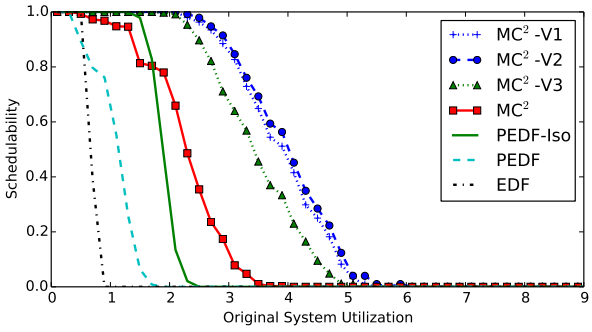
Ward, B., Herman, J., Kenna, C., and Anderson, J. (2013). Making shared caches more predictable on multicore platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 157–167.

Xu, M., Phan, L. T. X., Choi, H.-Y., and Lee, I. (2016). Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 1–12.

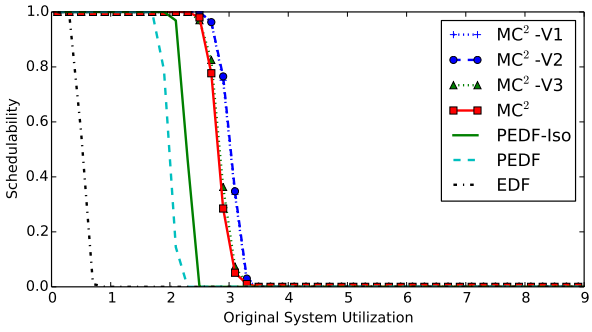
Yun, H., Mancuso, R., Wu, Z., and Pellizzoni, R. (2014). PALLOC: DRAM bank-aware memory allocator for performance isolation on multicoore platforms. In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 155–166.

Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., and Sha, L. (2013). MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 55–64.

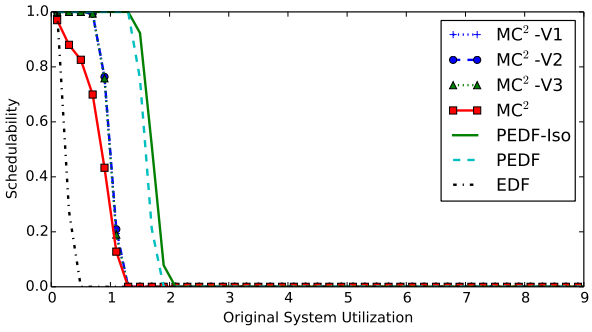
APPENDIX A: SCHEDULABILITY GRAPHS FOR THE STUDY DESCRIBED IN CHAPTER 3



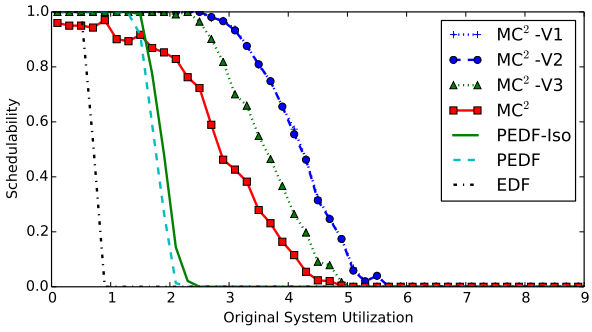
B-Heavy, Long, Heavy, Heavy, Large Var.



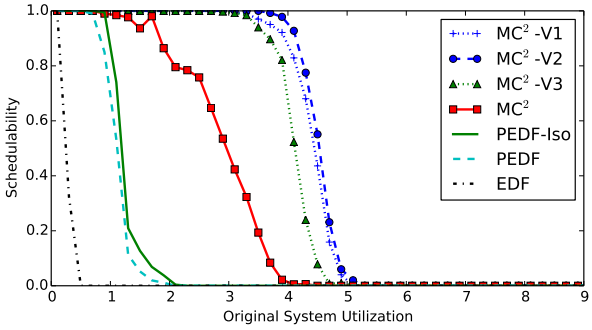
B-Heavy, Long, Light, Mod., Large Var.



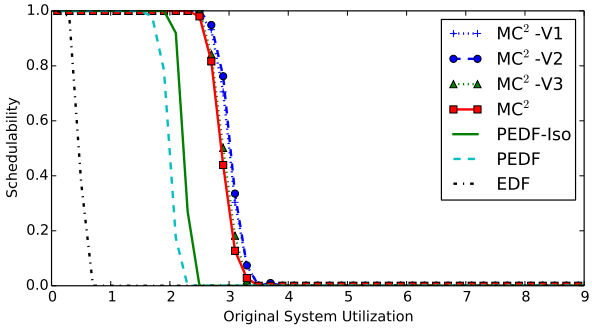
All-Mod., Short, Light, Heavy, Large Var.



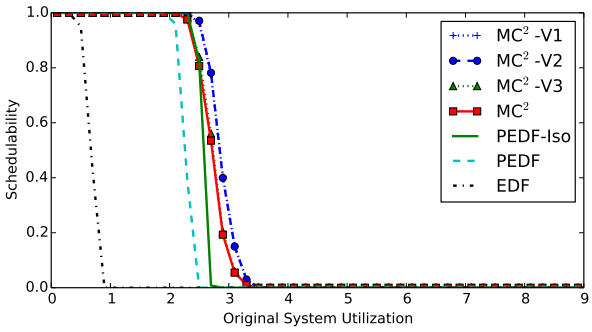
B-Heavy, Short, Heavy, Mod., Large Var.



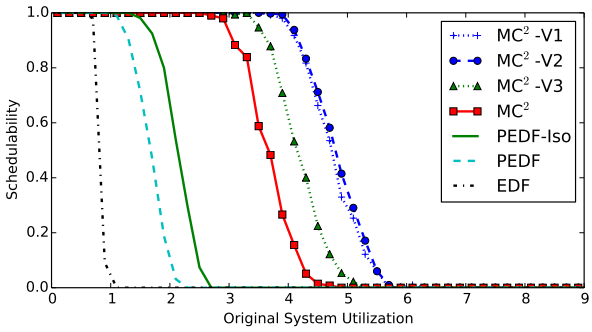
All-Mod., Cont., Heavy, Heavy, Const.



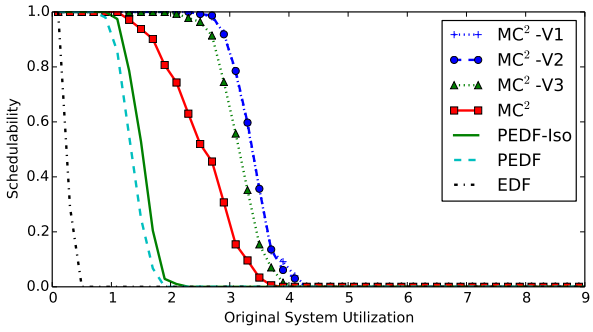
BC-Mod., Long, Light, Mod., Const.



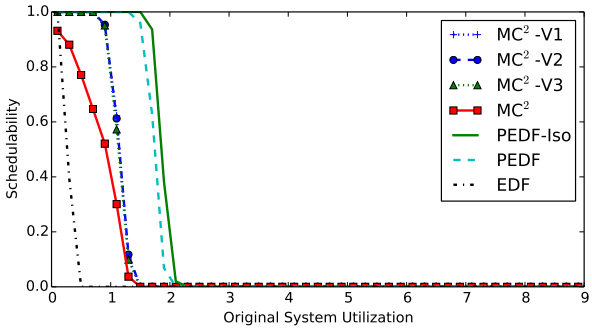
AB-Mod., Long, Light, Light, Large Var.



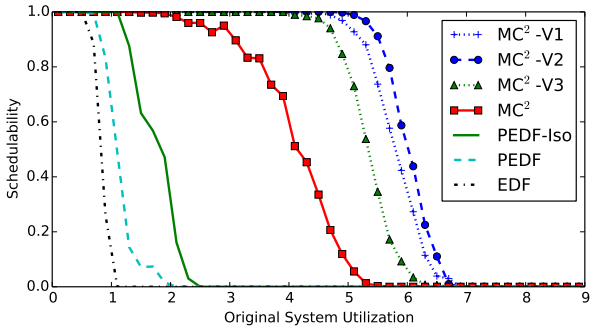
B-Heavy, Long, Mod., Light, Large Var.



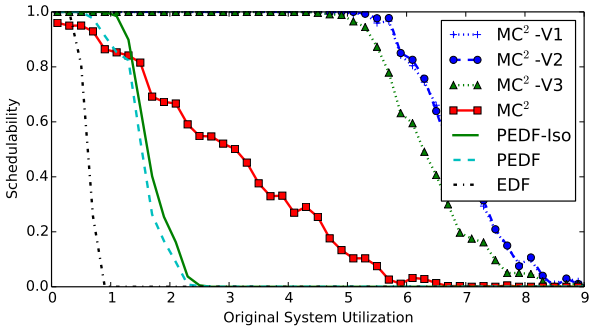
A-Heavy, Cont., Heavy, Heavy, Large Var.



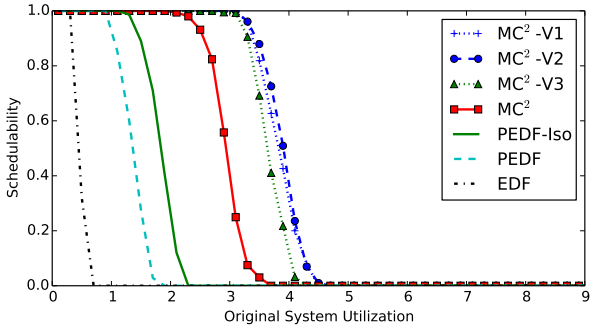
BC-Mod., Short, Light, Heavy, Const.



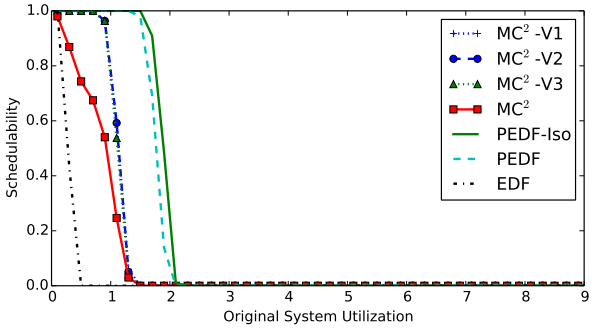
AC-Mod., Long, Heavy, Light, Const.



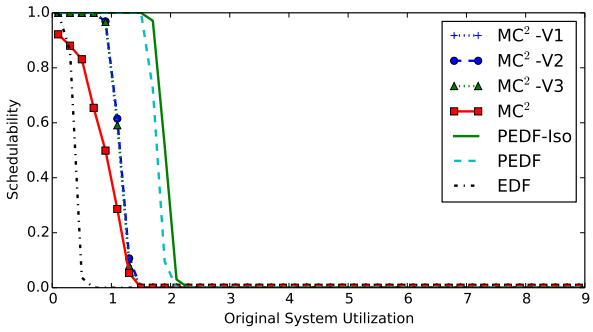
C-Heavy, Short, Heavy, Mod., Const.



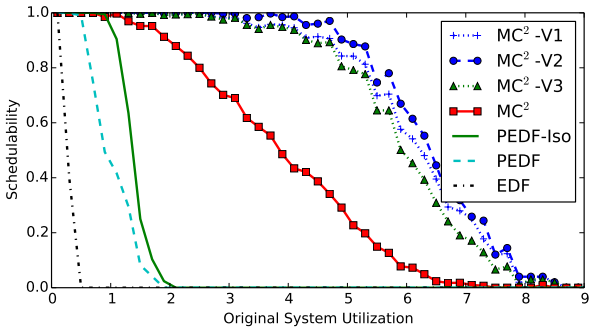
A-Heavy, Long, Mod., Heavy, Const.



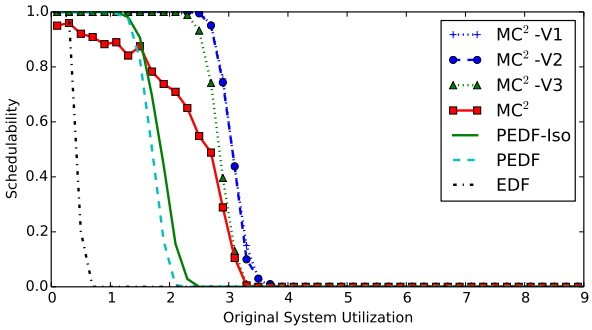
BC-Mod., Short, Light, Heavy, Large Var.



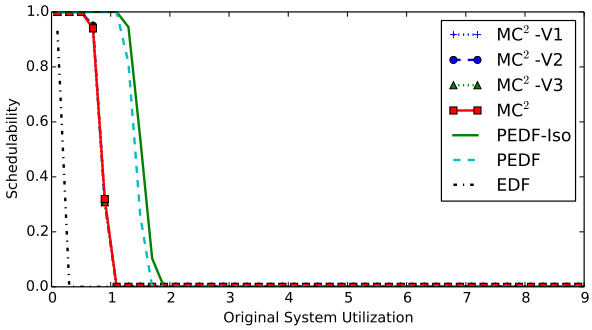
BC-Mod., Short, Light, Mod., Const.



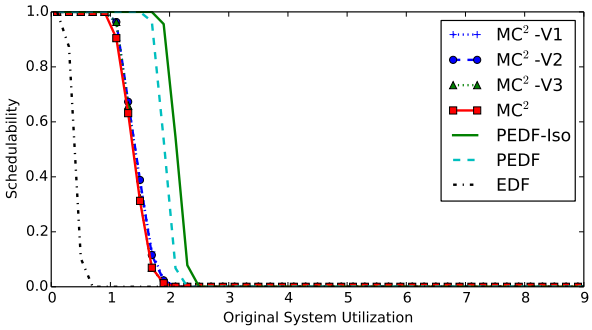
C-Heavy, Cont., Heavy, Heavy, Large Var.



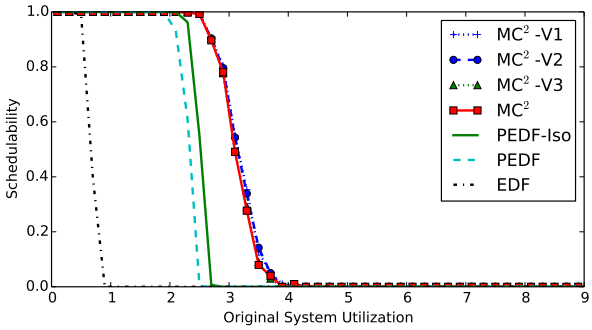
AB-Mod., Short, Mod., Heavy, Large Var.



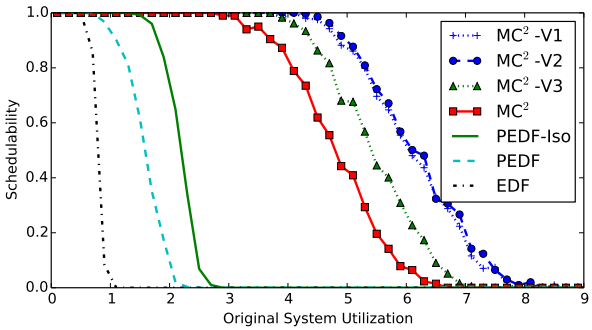
A-Heavy, Cont., Light, Mod., Large Var.



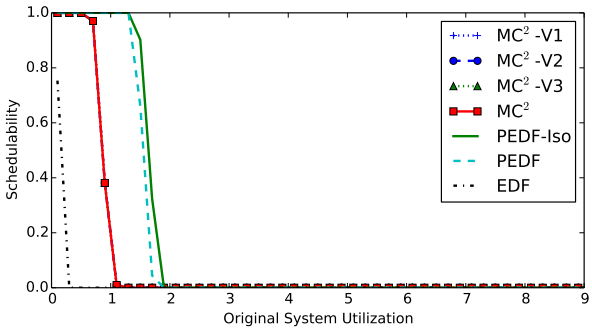
B-Heavy, Cont., Light, Light, Const.



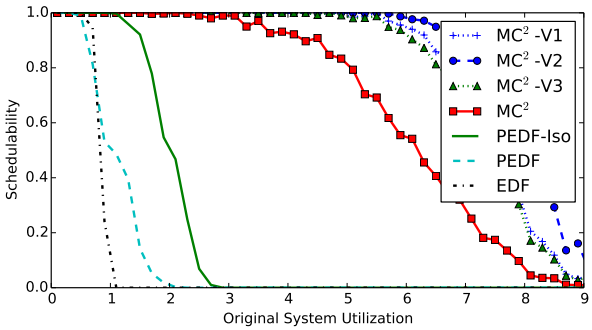
C-Heavy, Long, Light, Light, Const.



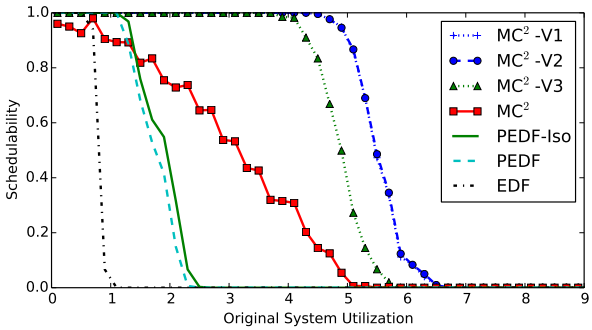
BC-Mod., Long, Mod., Light, Large Var.



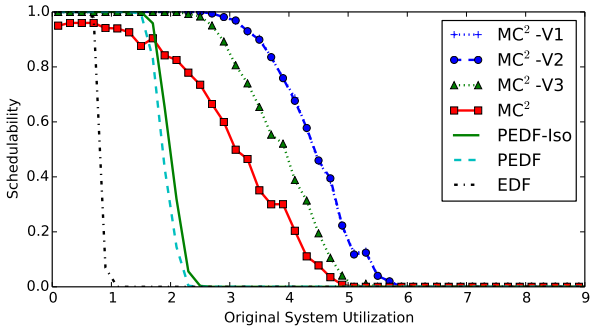
AC-Mod., Cont., Light, Heavy, Large Var.



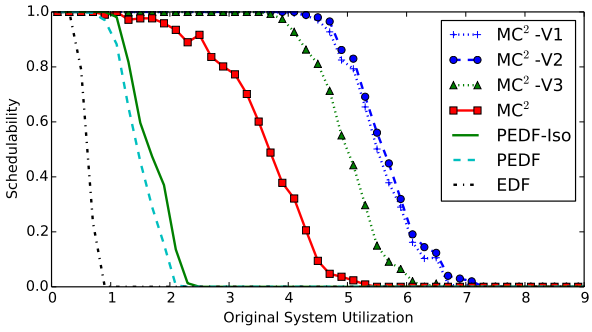
C-Heavy, Long, Heavy, Light, Large Var.



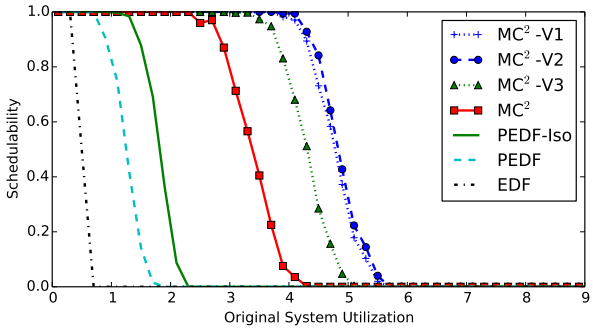
BC-Mod., Short, Heavy, Light, Const.



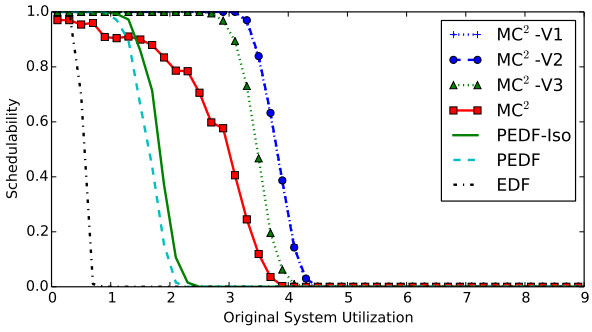
B-Heavy, Short, Heavy, Light, Large Var.



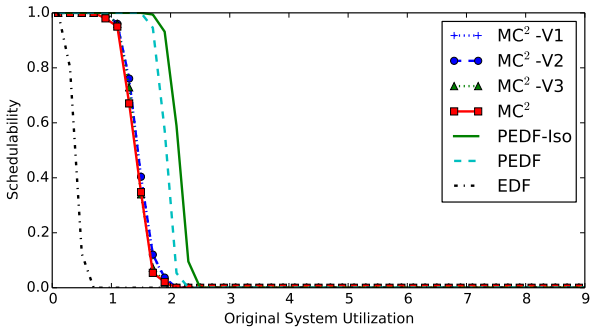
BC-Mod., Cont., Heavy, Light, Large Var.



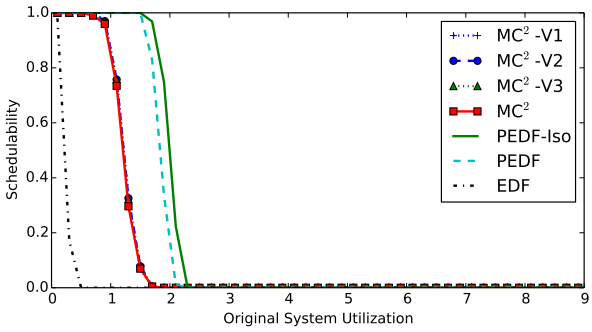
All-Mod., Long, Mod., Heavy, Large Var.



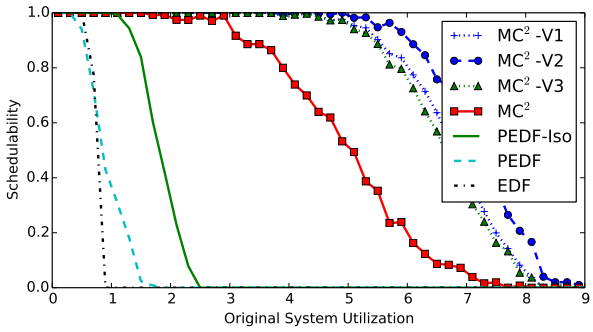
AB-Mod., Short, Heavy, Heavy, Const.



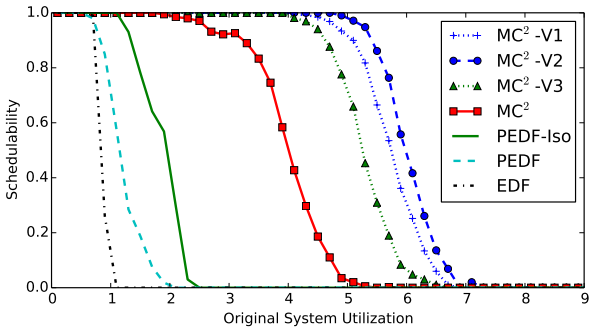
B-Heavy, Cont., Light, Light, Const.



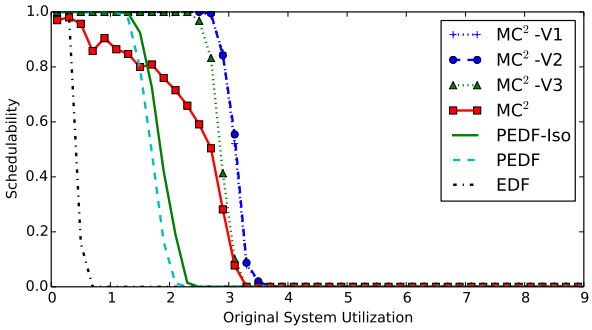
BC-Mod., Cont., Light, Mod., Large Var.



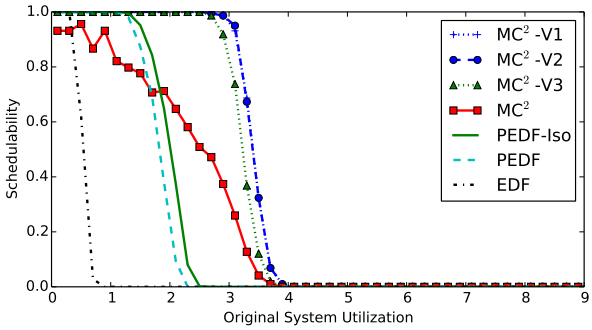
C-Heavy, Long, Heavy, Mod., Large Var.



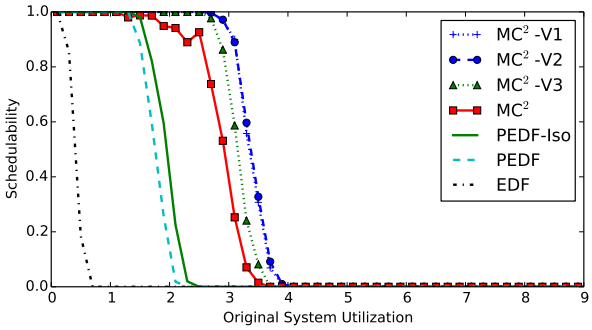
BC-Mod., Long, Heavy, Light, Const.



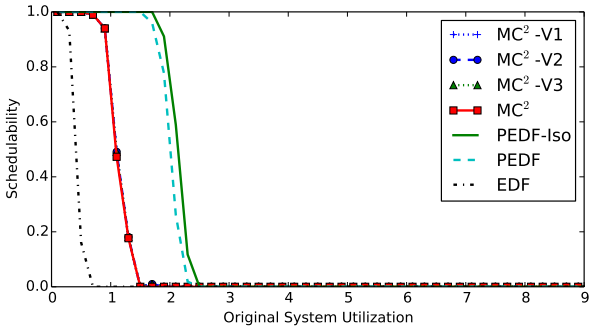
AB-Mod., Short, Mod., Heavy, Const.



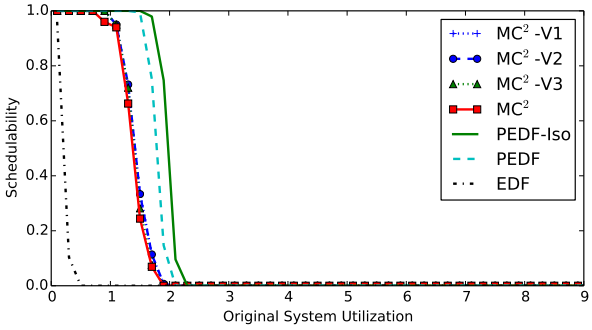
All-Mod., Short, Mod., Mod., Const.



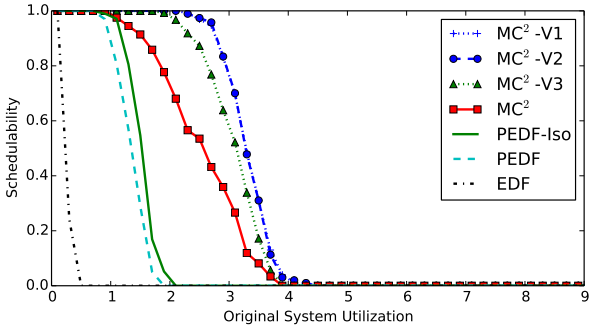
AB-Mod., Cont., Mod., Light, Const.



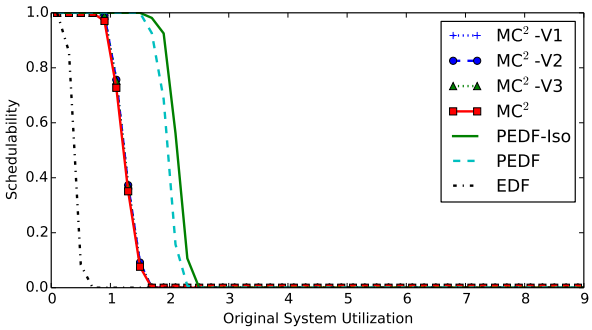
C-Heavy, Cont., Light, Light, Large Var.



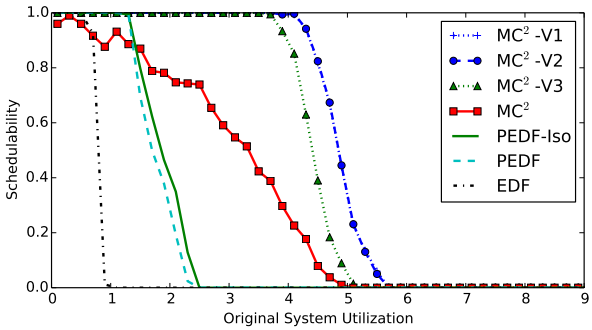
B-Heavy, Cont., Light, Mod., Const.



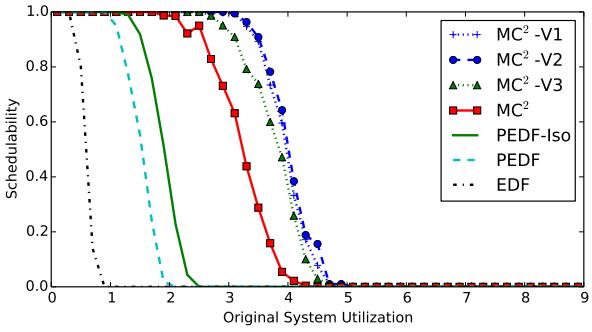
A-Heavy, Cont., Heavy, Heavy, Large Var.



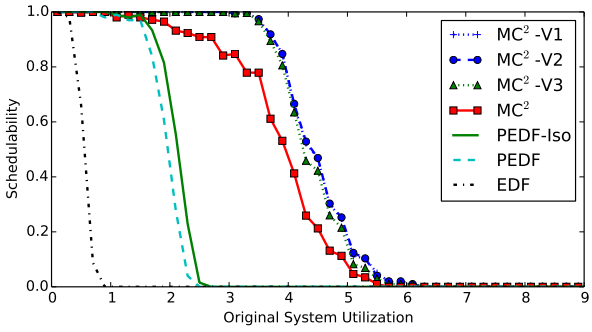
BC-Mod., Cont., Light, Light, Large Var.



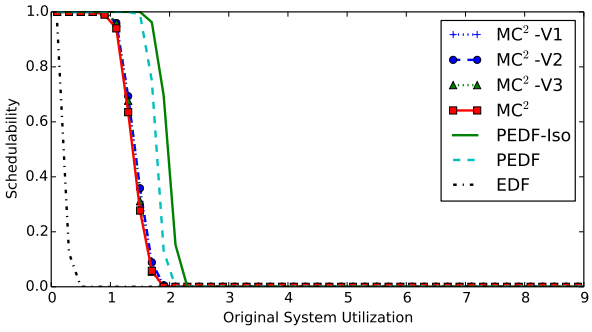
All-Mod., Short, Heavy, Light, Large Var.



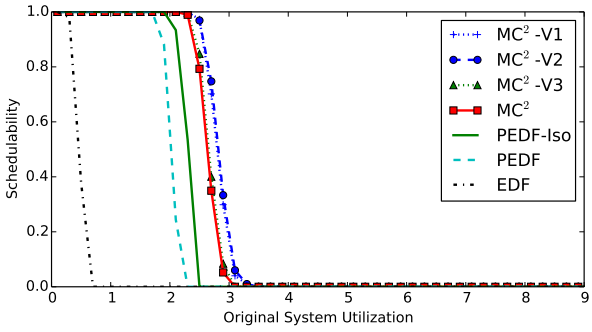
A-Heavy, Long, Mod., Mod., Large Var.



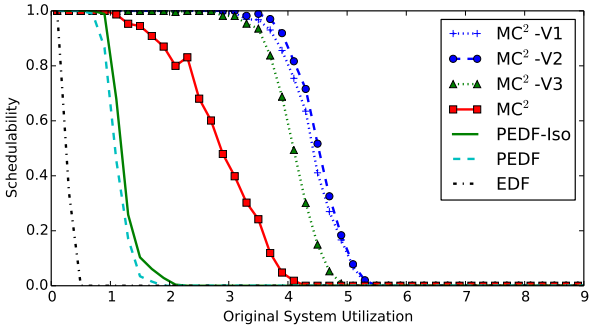
C-Heavy, Cont., Mod., Light, Large Var.



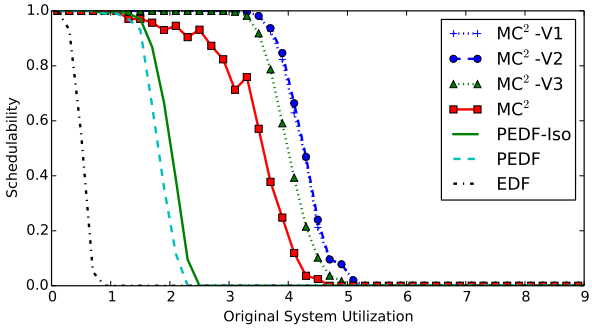
B-Heavy, Cont., Light, Mod., Large Var.



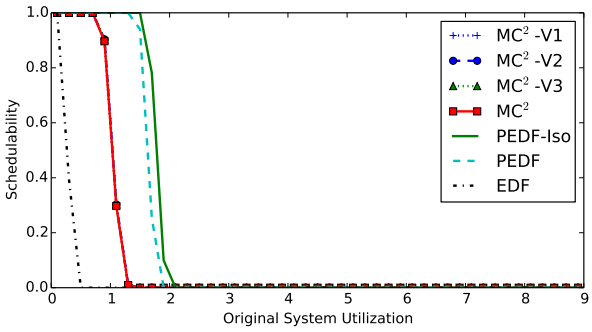
AB-Mod., Long, Light, Mod., Large Var.



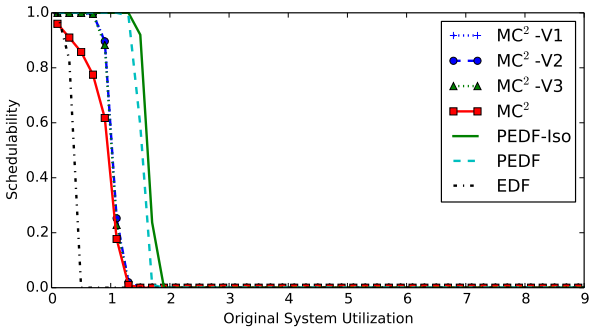
All-Mod., Cont., Heavy, Heavy, Large Var.



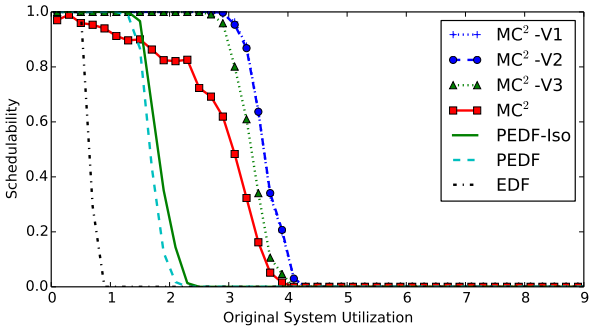
BC-Mod., Cont., Mod., Light, Large Var.



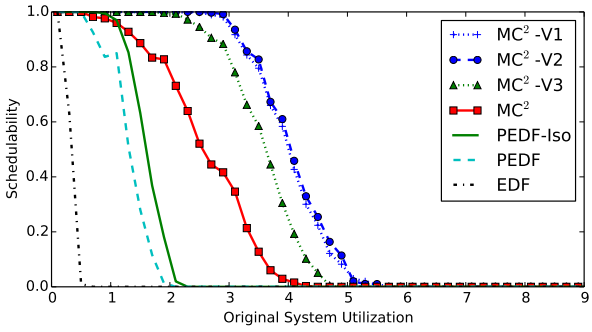
AB-Mod., Cont., Light, Light, Const.



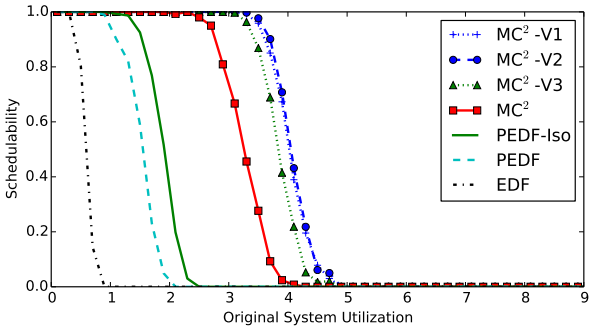
AB-Mod., Short, Light, Mod., Const.



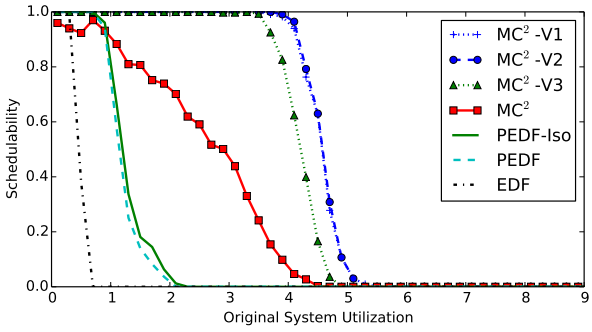
A-Heavy, Short, Heavy, Mod., Const.



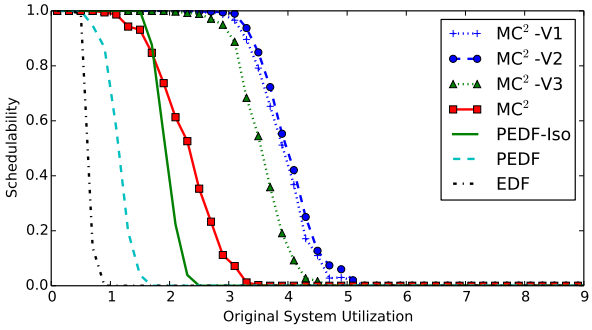
AB-Mod., Cont., Heavy, Mod., Large Var.



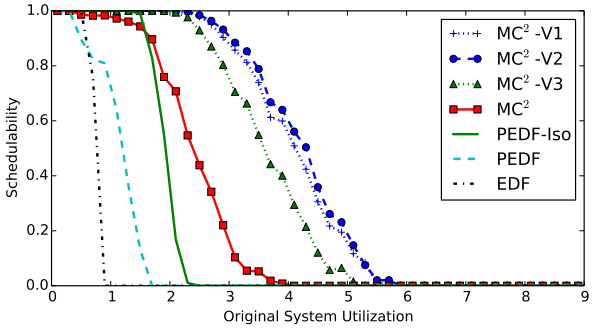
A-Heavy, Long, Mod., Mod., Large Var.



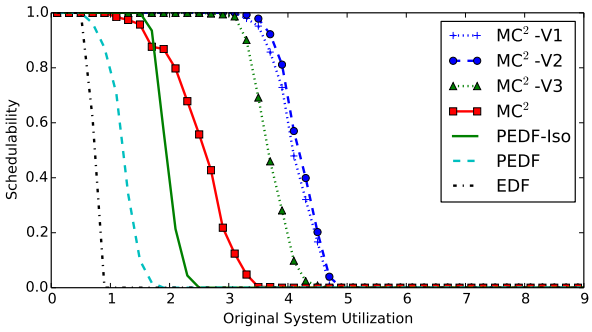
AC-Mod., Short, Heavy, Heavy, Const.



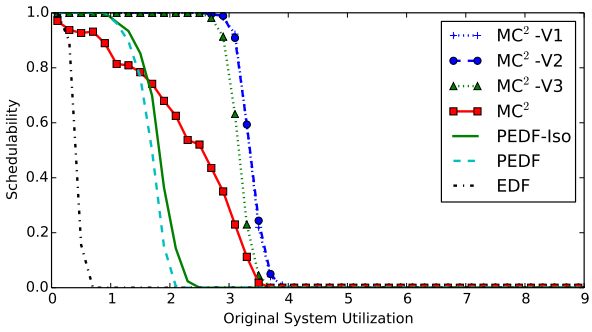
AB-Mod., Long, Heavy, Heavy, Const.



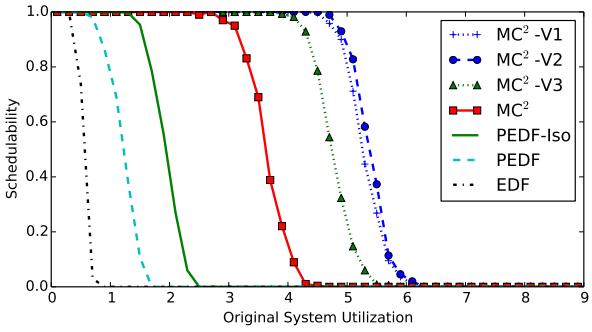
B-Heavy, Long, Heavy, Mod., Large Var.



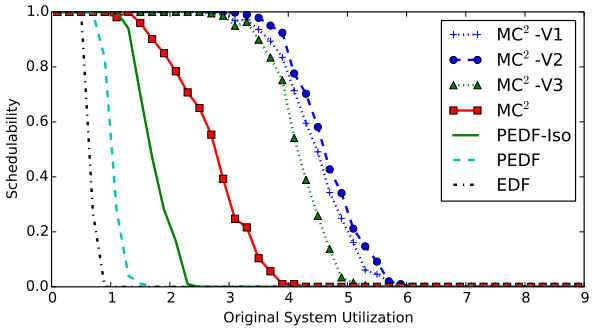
AB-Mod., Long, Heavy, Mod., Const.



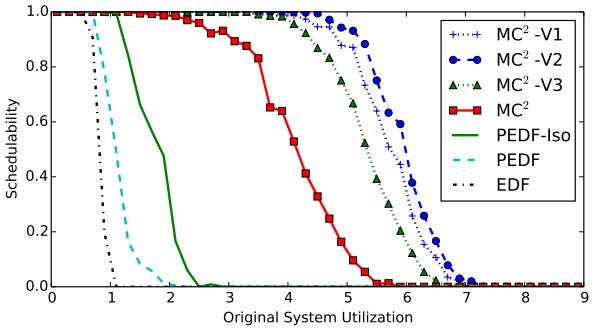
All-Mod., Short, Mod., Heavy, Const.



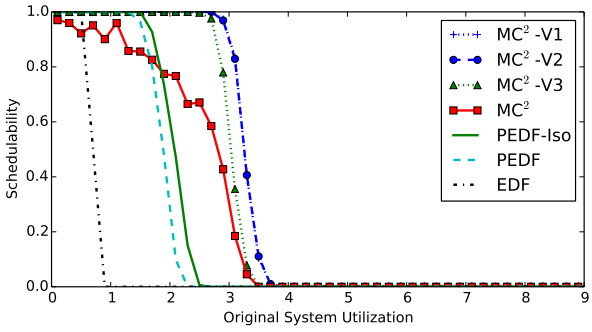
BC-Mod., Long, Mod., Heavy, Const.



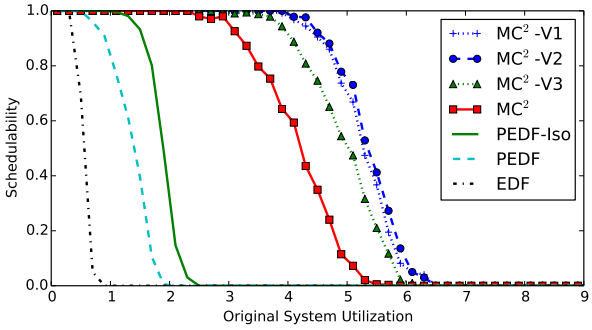
All-Mod., Long, Heavy, Heavy, Const.



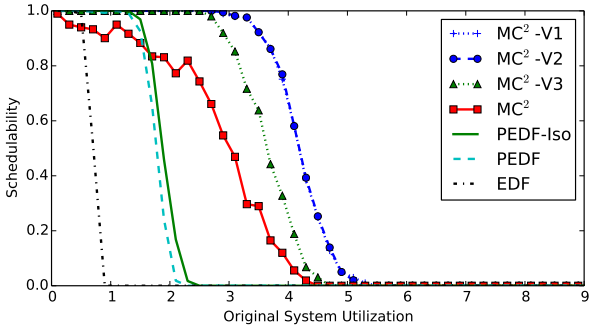
AC-Mod., Long, Heavy, Light, Const.



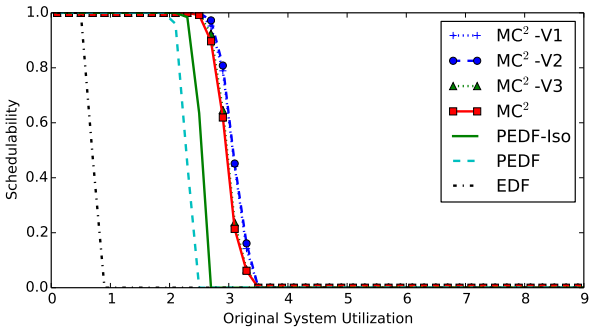
AB-Mod., Short, Mod., Light, Const.



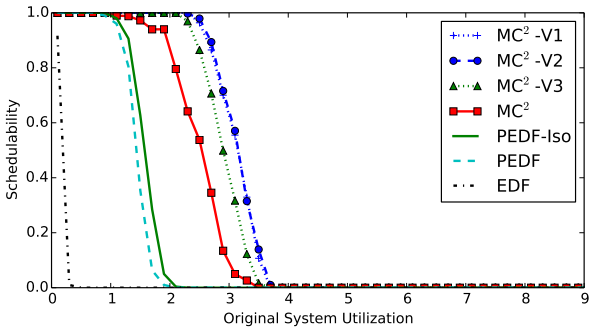
AC-Mod., Long, Mod., Mod., Large Var.



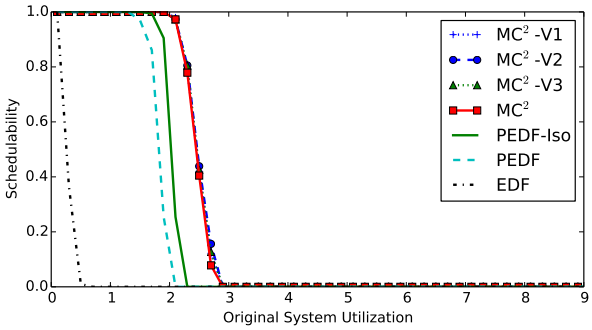
B-Heavy, Short, Heavy, Mod., Large Var.



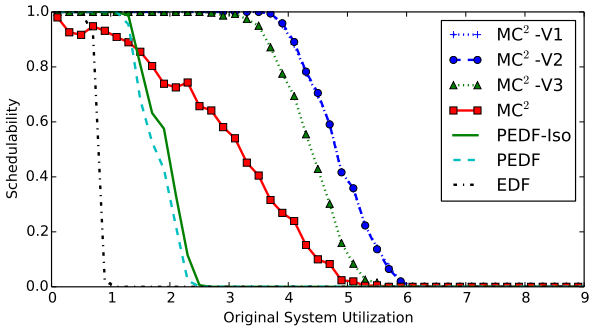
BC-Mod., Long, Light, Light, Const.



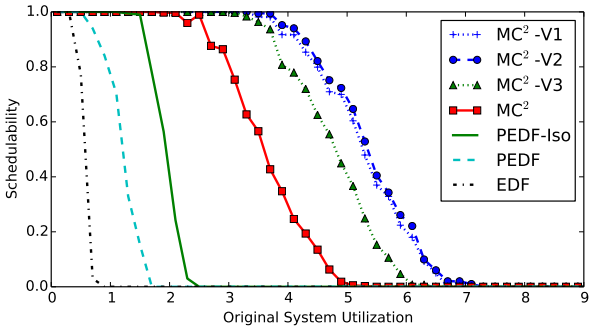
AB-Mod., Cont., Mod., Heavy, Large Var.



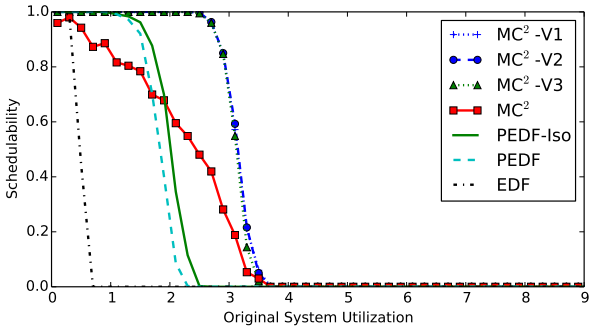
AC-Mod., Long, Light, Heavy, Const.



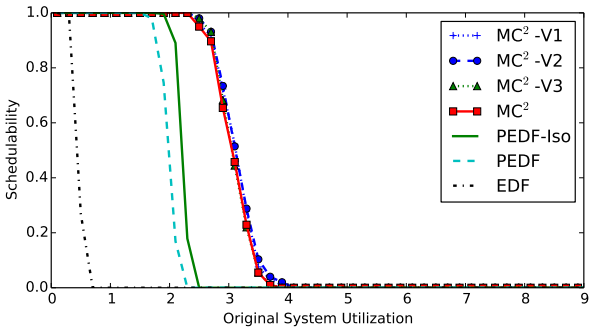
All-Mod., Short, Heavy, Light, Large Var.



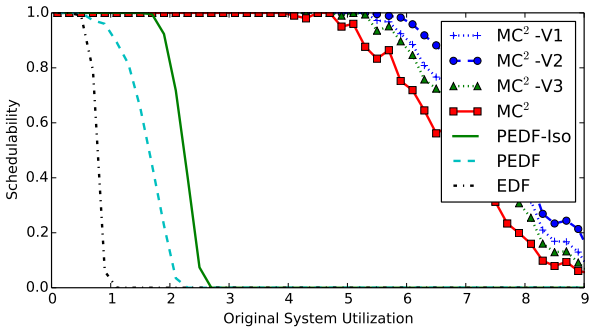
BC-Mod., Long, Mod., Heavy, Large Var.



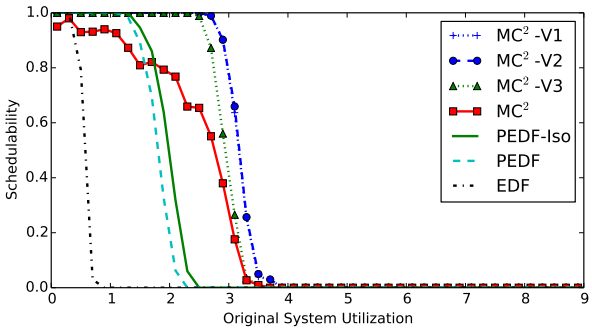
AC-Mod., Short, Mod., Mod., Const.



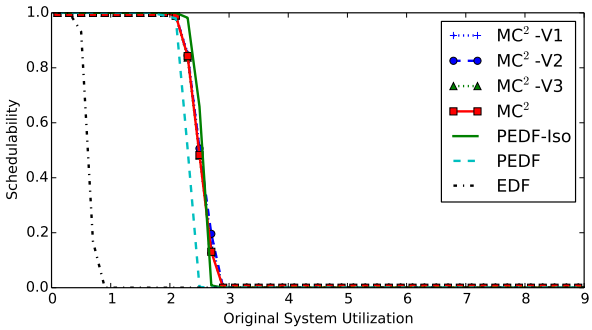
C-Heavy, Long, Light, Mod., Large Var.



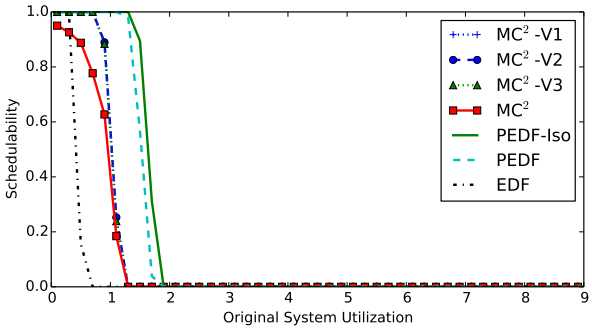
C-Heavy, Long, Mod., Light, Large Var.



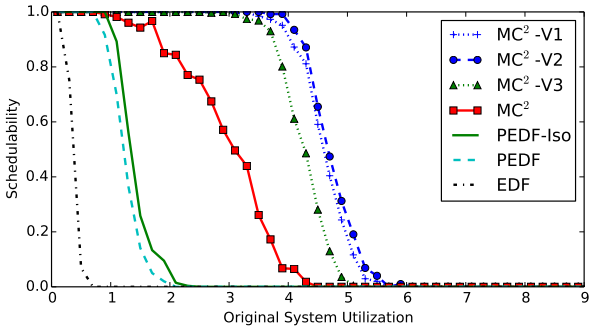
AB-Mod., Short, Mod., Mod., Large Var.



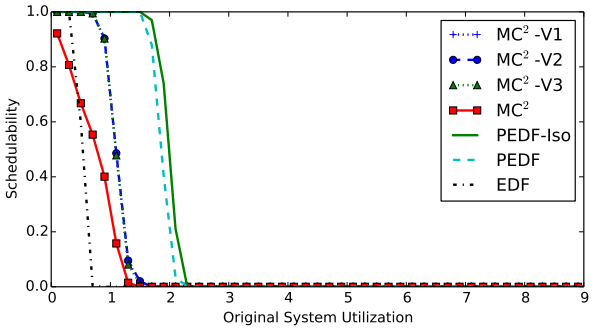
AC-Mod., Long, Light, Light, Large Var.



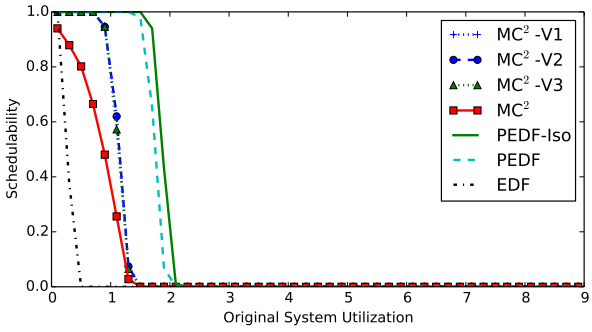
AB-Mod., Short, Light, Light, Large Var.



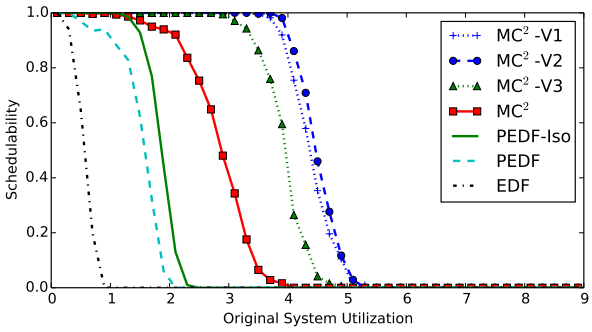
All-Mod., Cont., Heavy, Mod., Large Var.



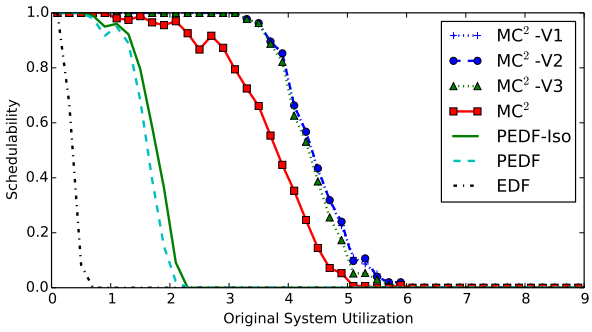
C-Heavy, Short, Light, Light, Large Var.



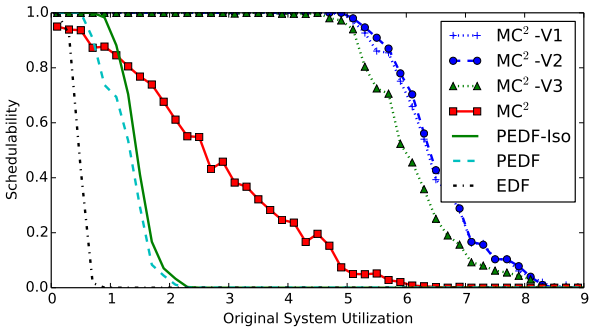
BC-Mod., Short, Light, Heavy, Large Var.



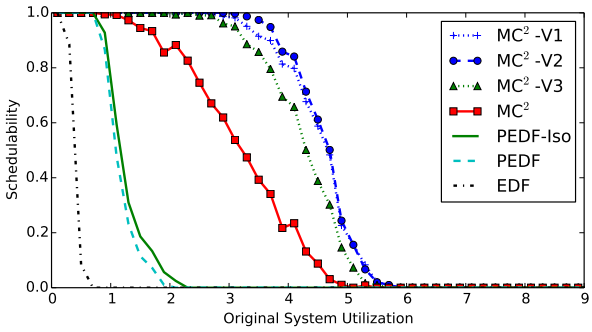
B-Heavy, Cont., Heavy, Light, Const.



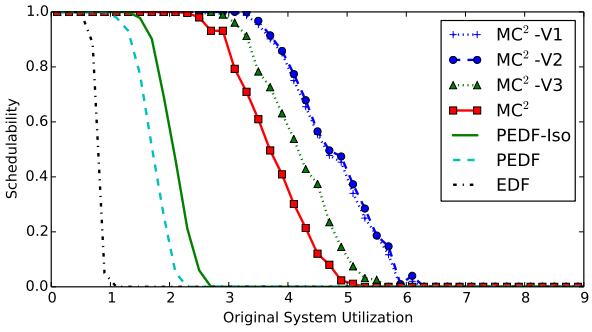
C-Heavy, Cont., Mod., Mod., Const.



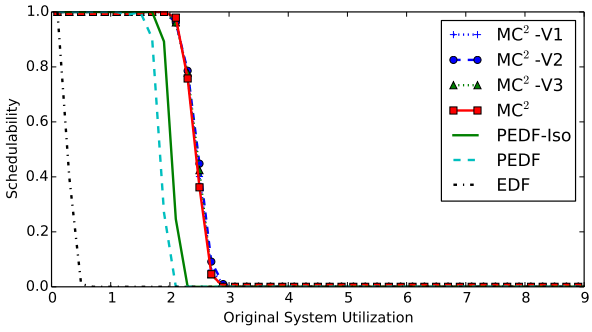
C-Heavy, Short, Heavy, Heavy, Const.



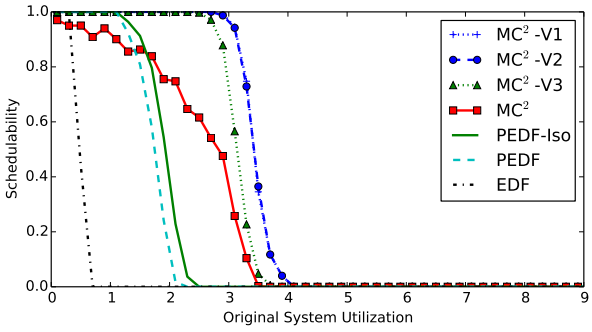
AC-Mod., Cont., Heavy, Mod., Large Var.



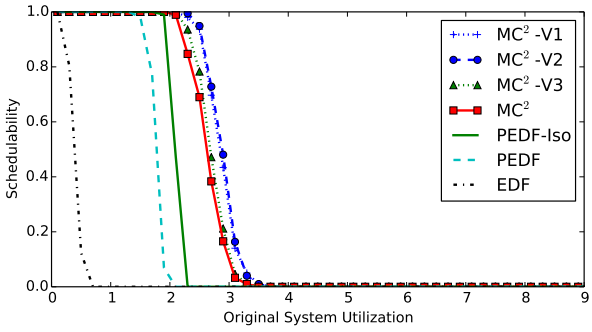
AB-Mod., Long, Mod., Light, Large Var.



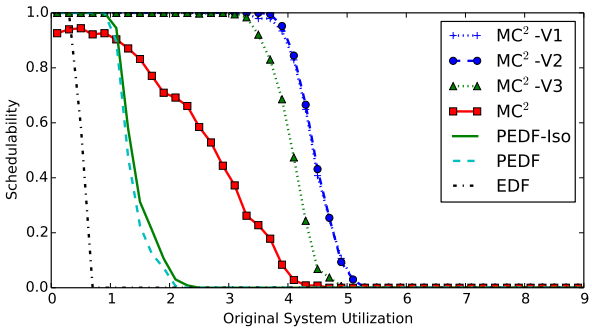
AC-Mod., Long, Light, Heavy, Large Var.



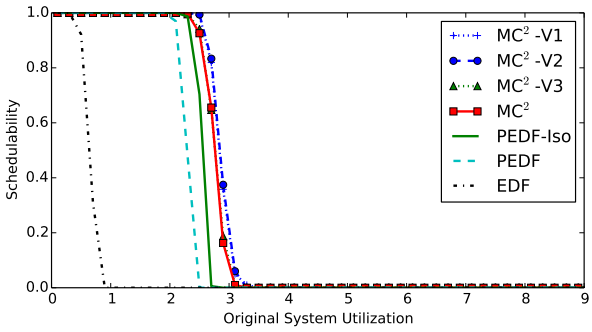
B-Heavy, Short, Mod., Heavy, Const.



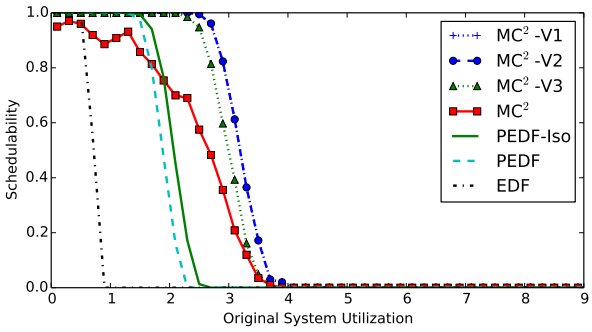
B-Heavy, Long, Light, Heavy, Large Var.



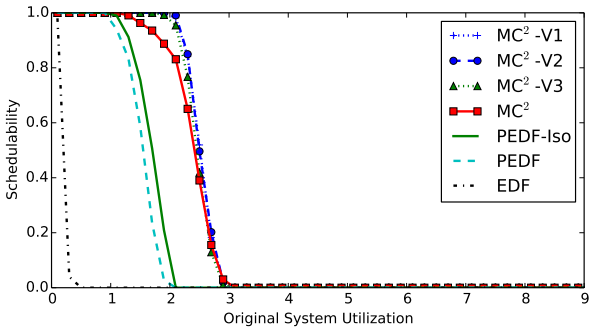
All-Mod., Short, Heavy, Heavy, Large Var.



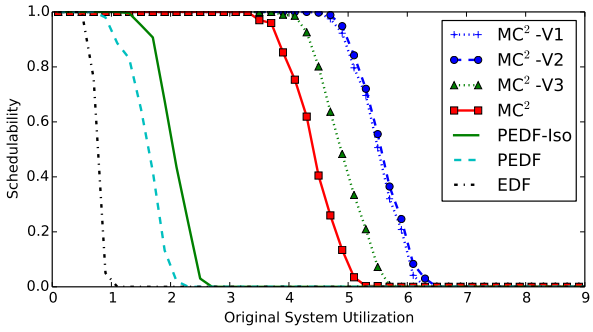
All-Mod., Long, Light, Light, Large Var.



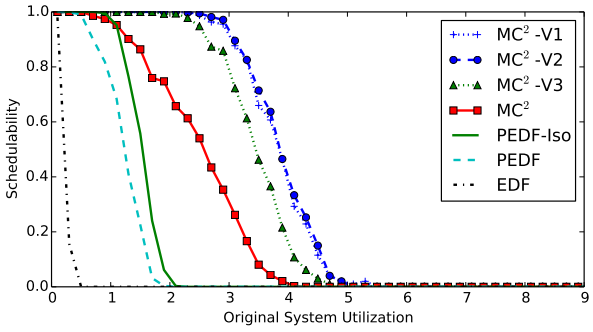
AB-Mod., Short, Mod., Light, Large Var.



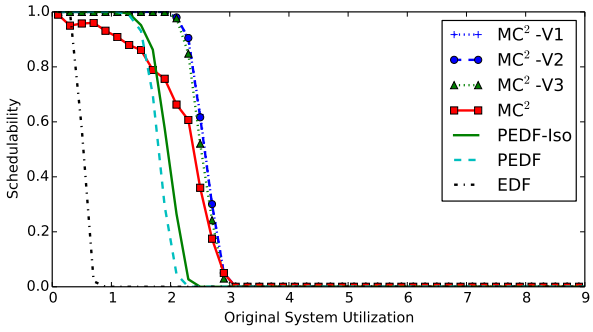
A-Heavy, Cont., Mod., Heavy, Large Var.



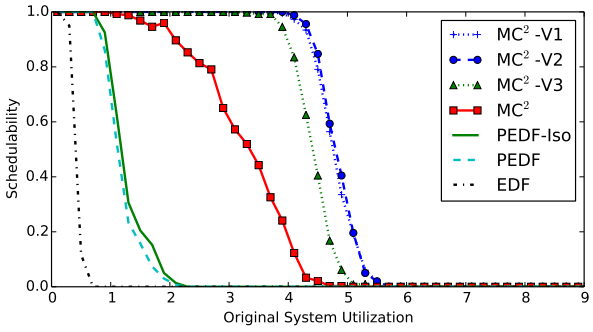
All-Mod., Long, Mod., Light, Large Var.



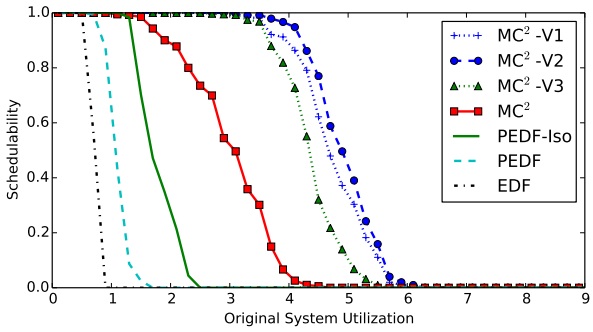
AB-Mod., Cont., Heavy, Heavy, Large Var.



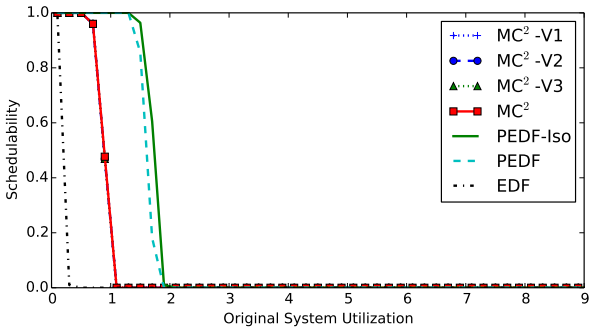
A-Heavy, Short, Mod., Mod., Large Var.



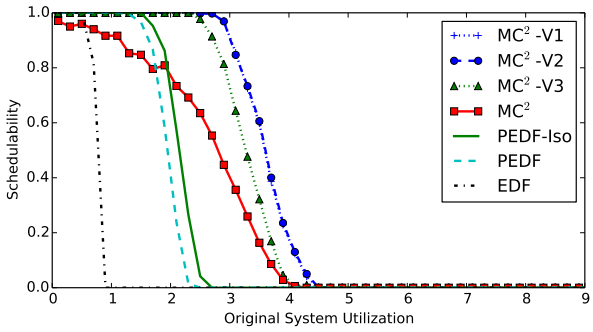
AC-Mod., Cont., Heavy, Mod., Const.



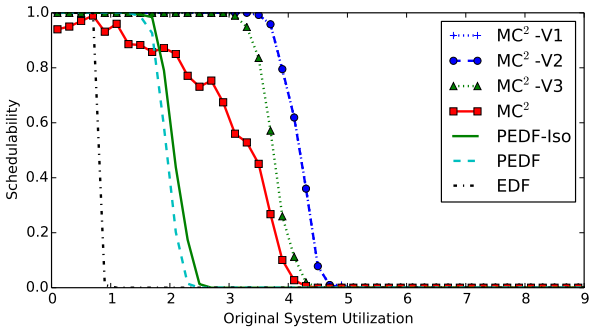
All-Mod., Long, Heavy, Mod., Const.



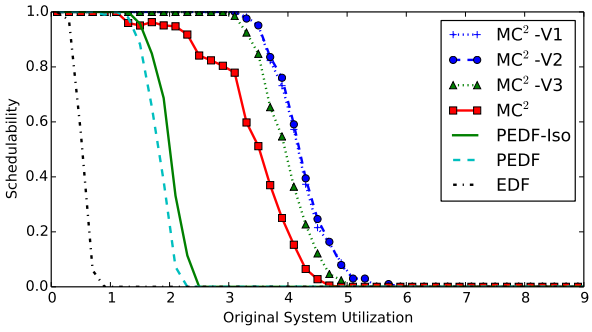
AC-Mod., Cont., Light, Mod., Const.



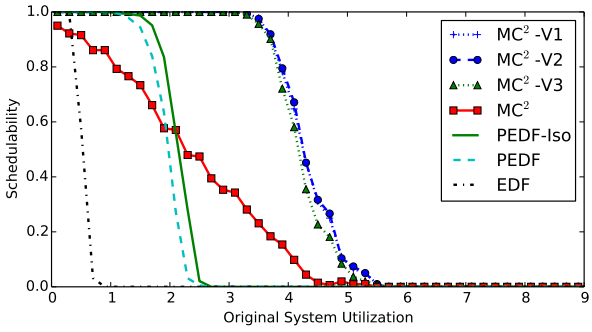
B-Heavy, Short, Mod., Light, Large Var.



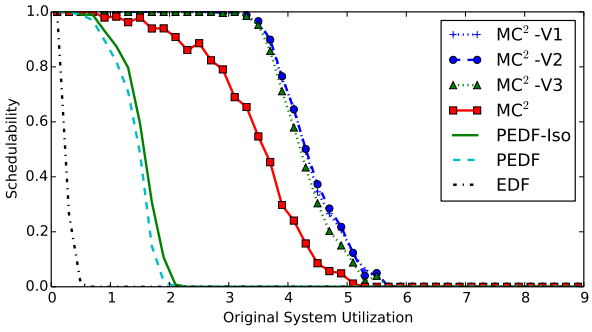
AB-Mod., Short, Heavy, Light, Const.



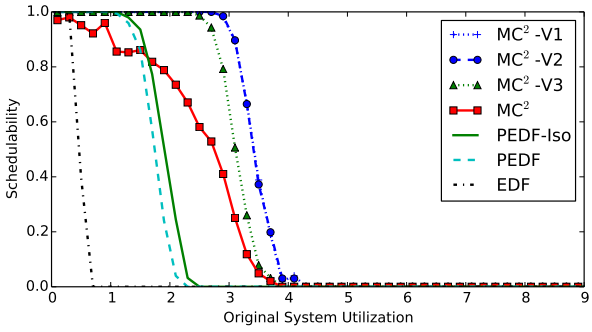
BC-Mod., Cont., Mod., Light, Large Var.



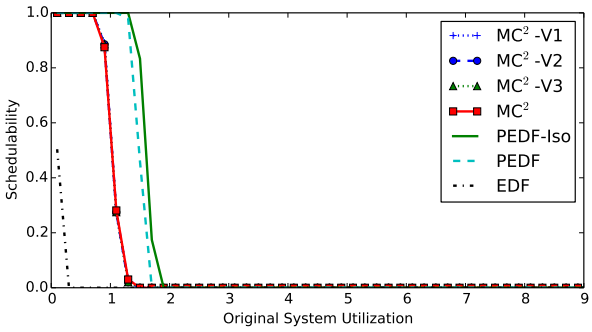
C-Heavy, Short, Mod., Mod., Const.



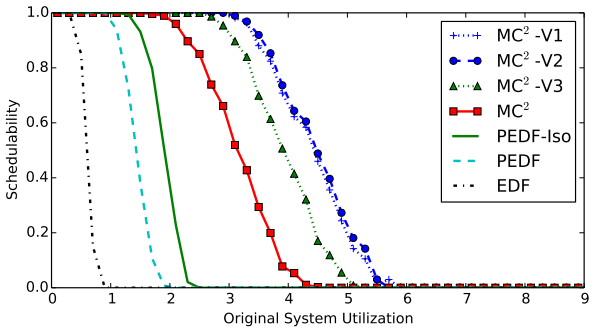
C-Heavy, Cont., Mod., Heavy, Large Var.



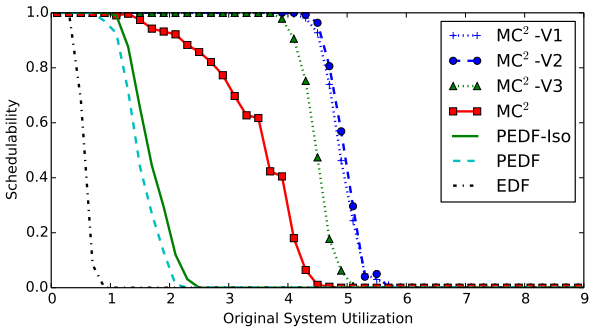
B-Heavy, Short, Mod., Heavy, Large Var.



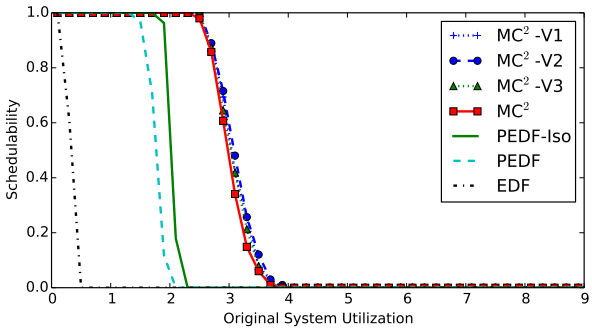
AB-Mod., Cont., Light, Heavy, Large Var.



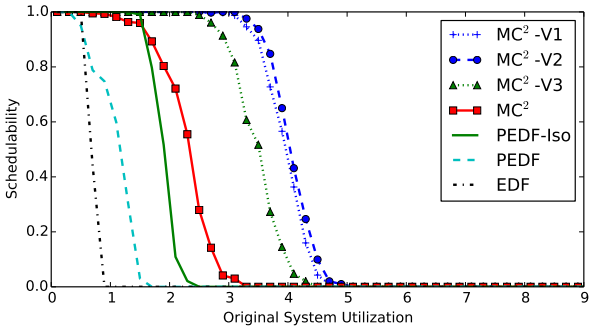
AB-Mod., Long, Mod., Mod., Large Var.



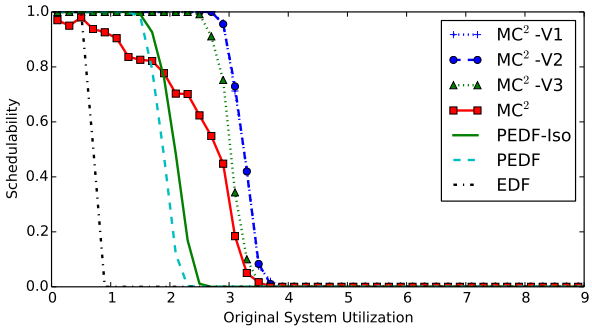
All-Mod., Cont., Heavy, Light, Const.



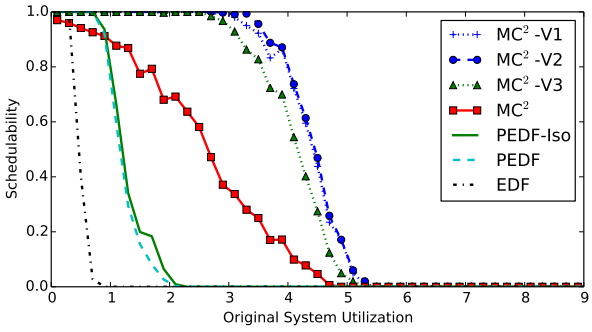
C-Heavy, Long, Light, Heavy, Large Var.



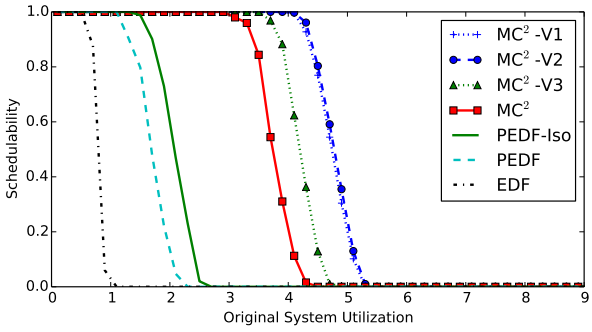
B-Heavy, Long, Heavy, Heavy, Const.



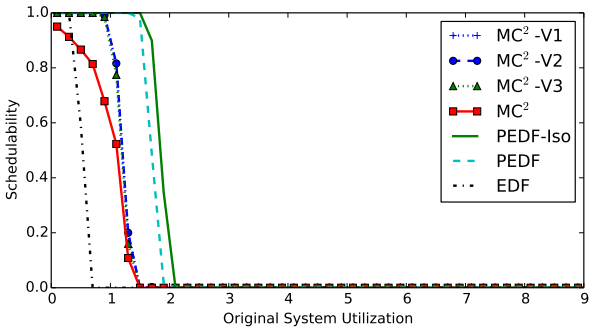
AB-Mod., Short, Mod., Light, Const.



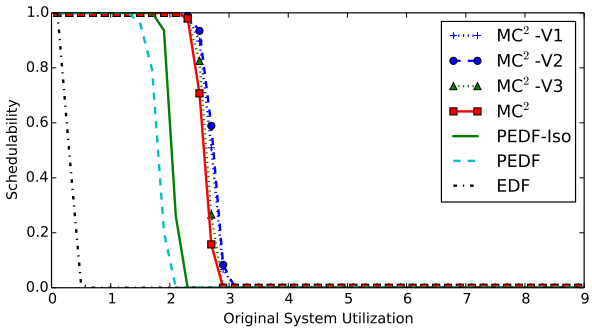
AC-Mod., Short, Heavy, Heavy, Large Var.



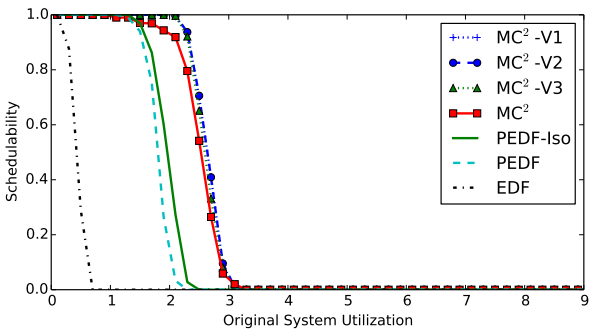
AB-Mod., Long, Mod., Light, Const.



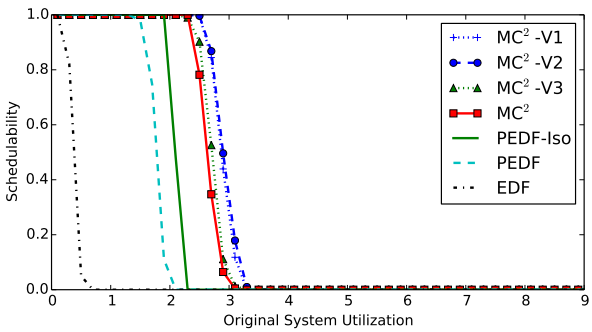
B-Heavy, Short, Light, Light, Const.



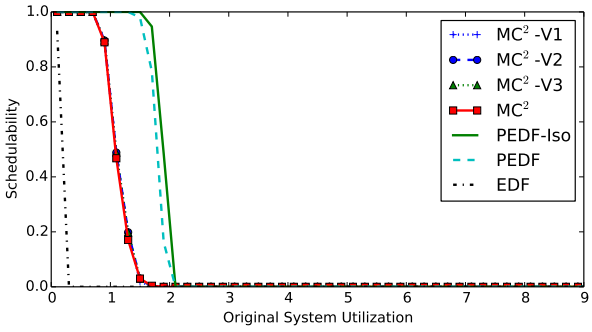
All-Mod., Long, Light, Heavy, Const.



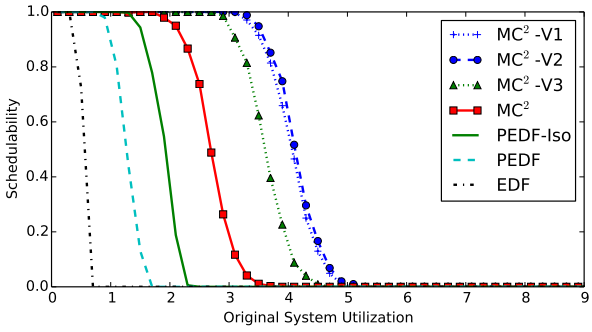
A-Heavy, Cont., Mod., Light, Large Var.



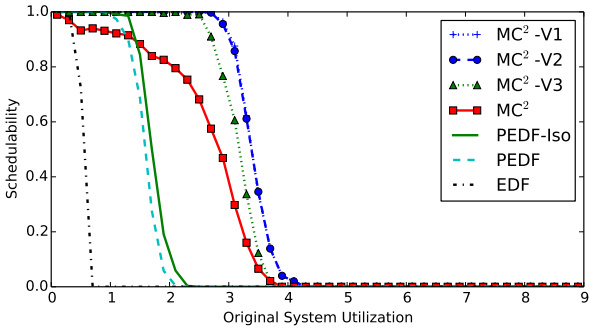
B-Heavy, Long, Light, Heavy, Large Var.



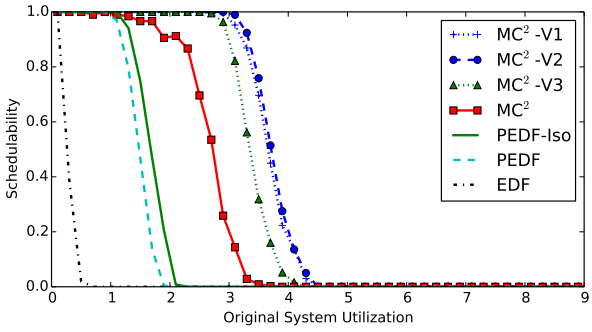
C-Heavy, Cont., Light, Heavy, Large Var.



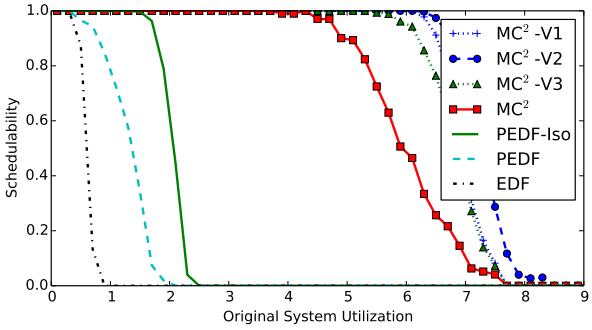
B-Heavy, Long, Mod., Heavy, Large Var.



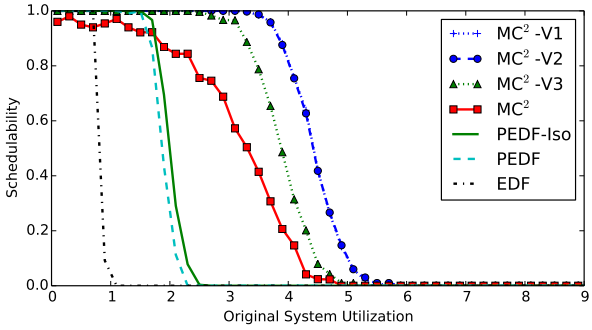
A-Heavy, Short, Heavy, Heavy, Const.



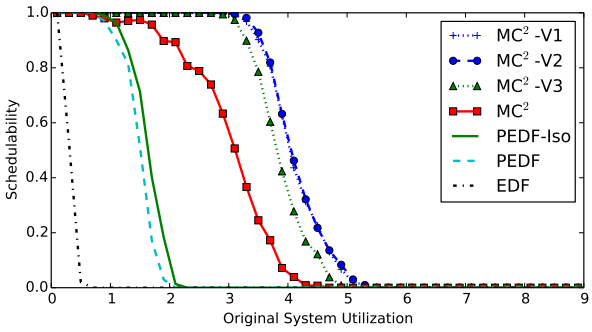
B-Heavy, Cont., Mod., Mod., Const.



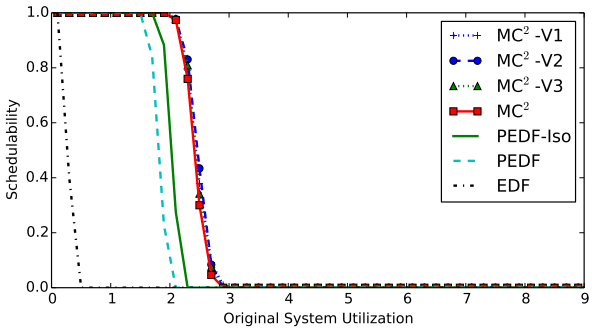
C-Heavy, Long, Mod., Mod., Const.



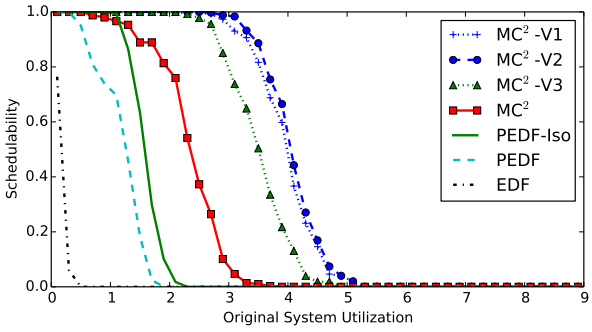
B-Heavy, Short, Heavy, Light, Const.



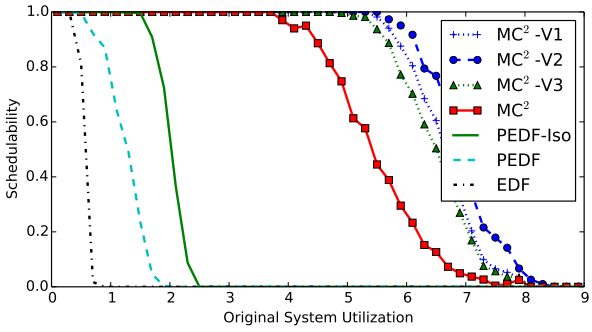
BC-Mod., Cont., Mod., Mod., Large Var.



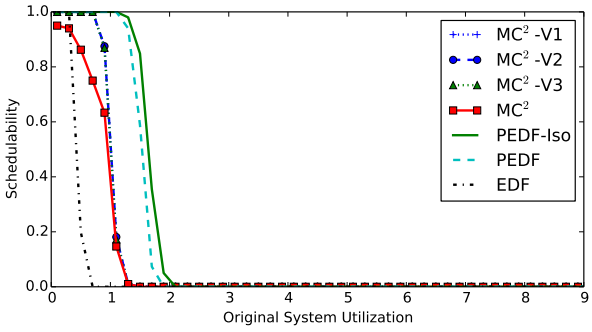
AC-Mod., Long, Light, Heavy, Large Var.



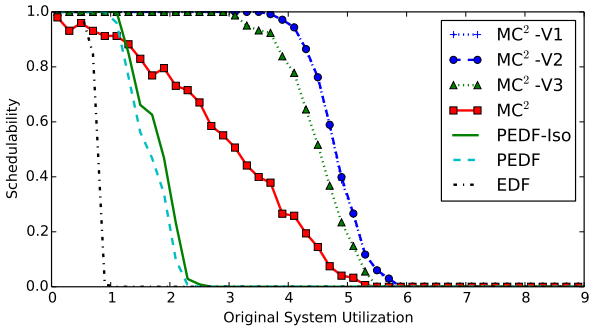
B-Heavy, Cont., Heavy, Heavy, Large Var.



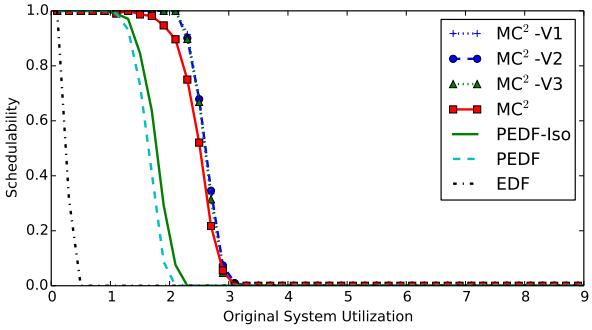
C-Heavy, Long, Mod., Heavy, Large Var.



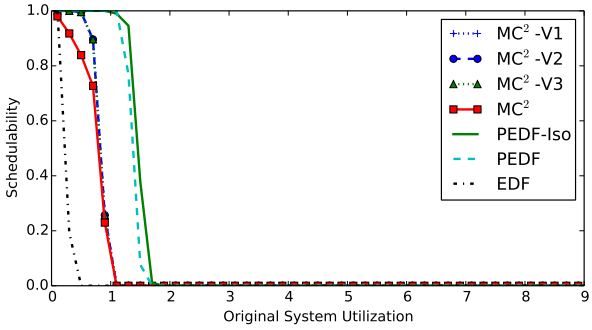
AB-Mod., Short, Light, Light, Large Var.



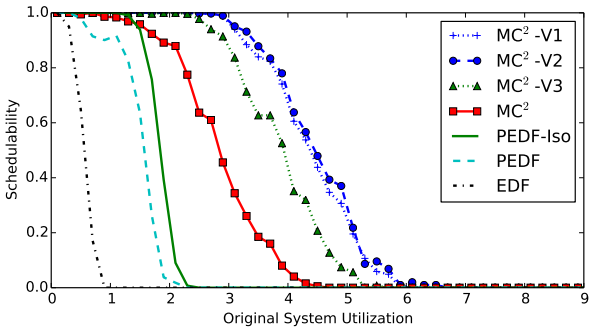
AC-Mod., Short, Heavy, Light, Large Var.



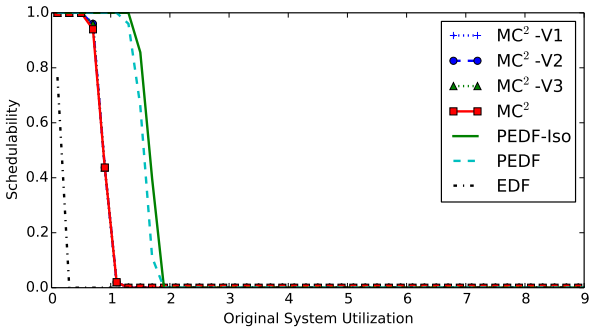
A-Heavy, Cont., Mod., Mod., Large Var.



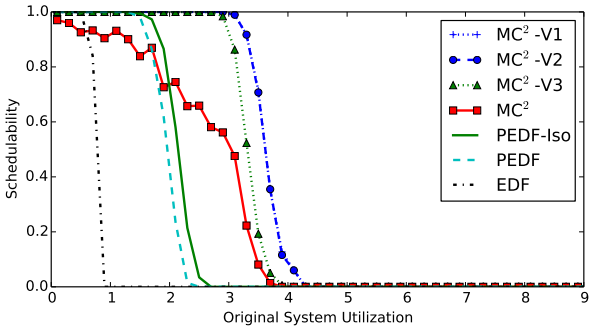
A-Heavy, Short, Light, Heavy, Large Var.



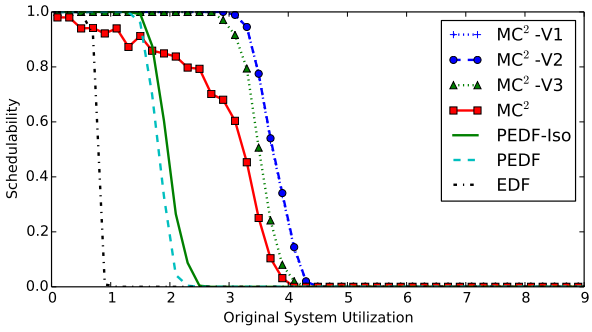
B-Heavy, Cont., Heavy, Light, Large Var.



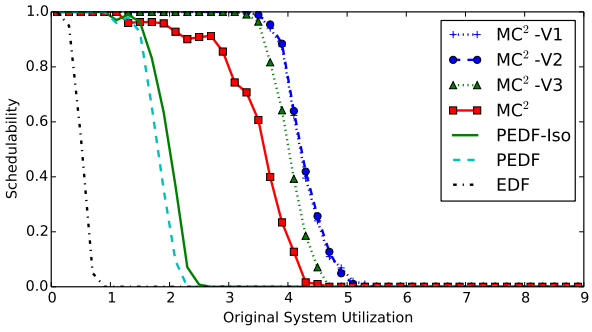
AC-Mod., Cont., Light, Heavy, Large Var.



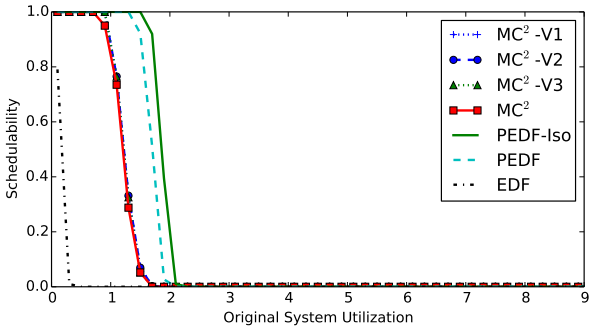
B-Heavy, Short, Mod., Light, Const.



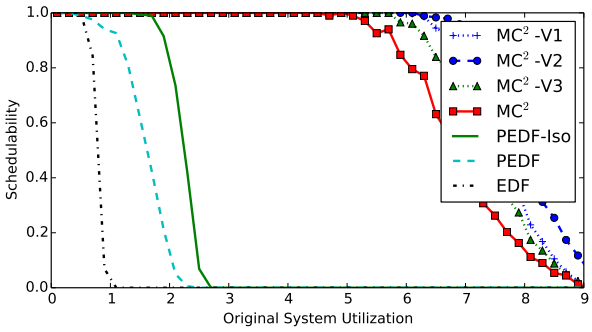
A-Heavy, Short, Heavy, Light, Const.



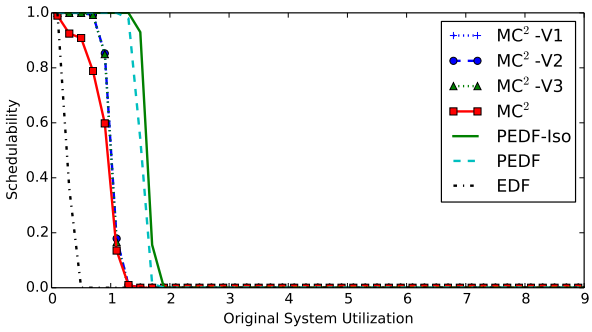
BC-Mod., Cont., Mod., Light, Const.



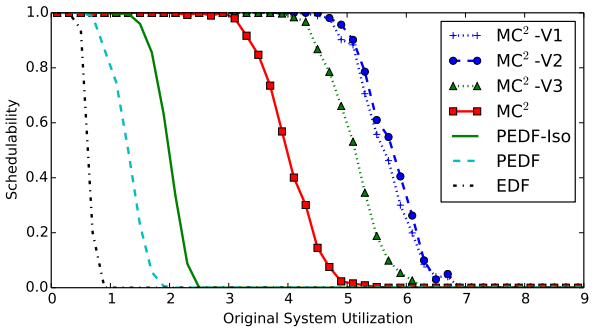
BC-Mod., Cont., Light, Heavy, Const.



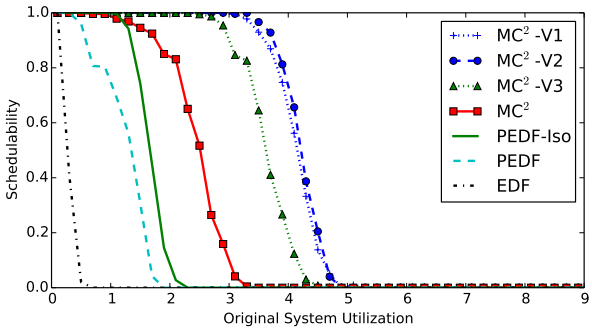
C-Heavy, Long, Mod., Light, Const.



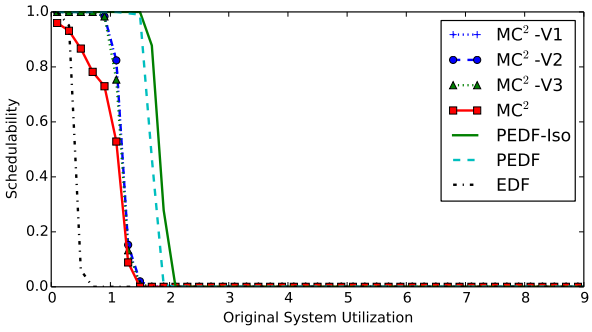
AB-Mod., Short, Light, Heavy, Const.



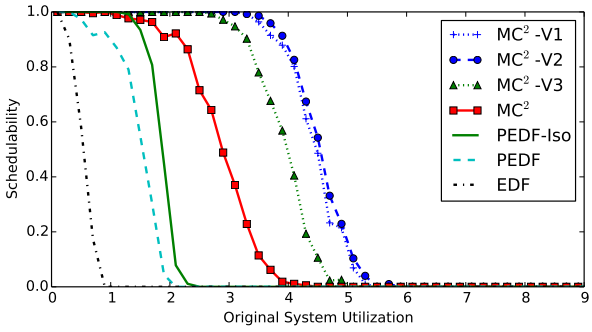
BC-Mod., Long, Mod., Mod., Const.



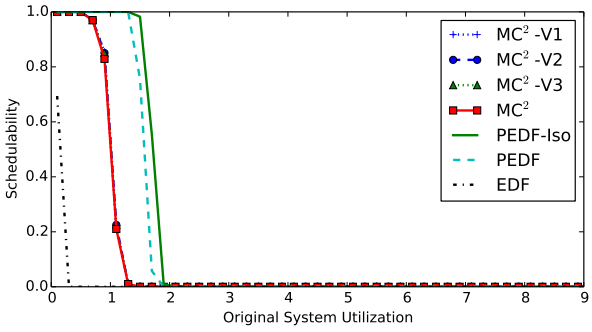
B-Heavy, Cont., Heavy, Mod., Const.



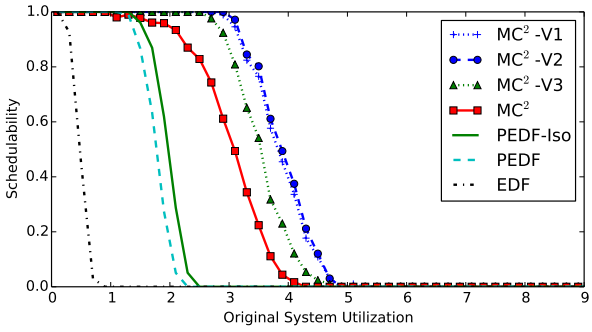
B-Heavy, Short, Light, Mod., Const.



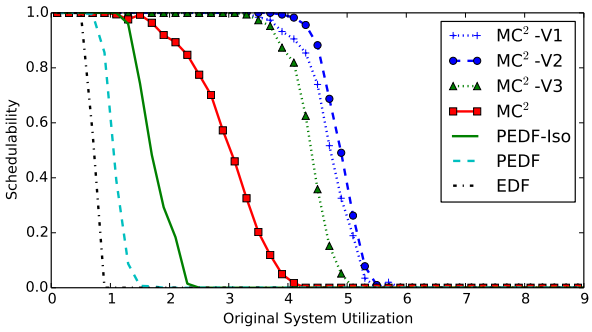
B-Heavy, Cont., Heavy, Light, Const.



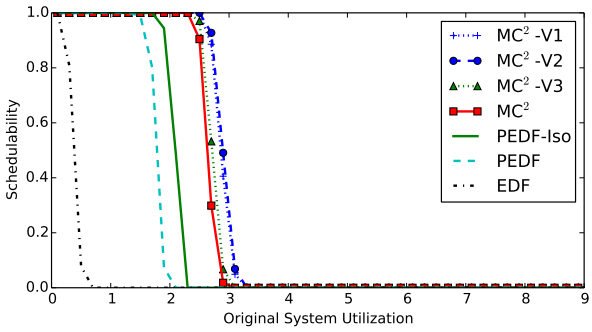
All-Mod., Cont., Light, Heavy, Const.



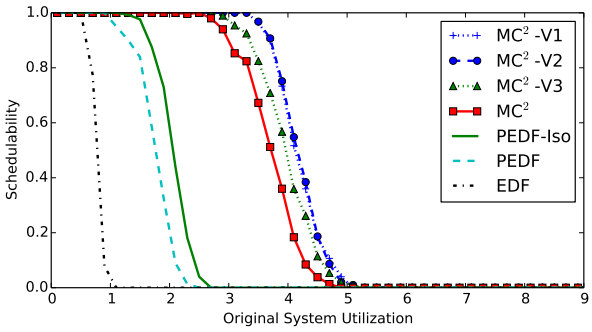
B-Heavy, Cont., Mod., Light, Large Var.



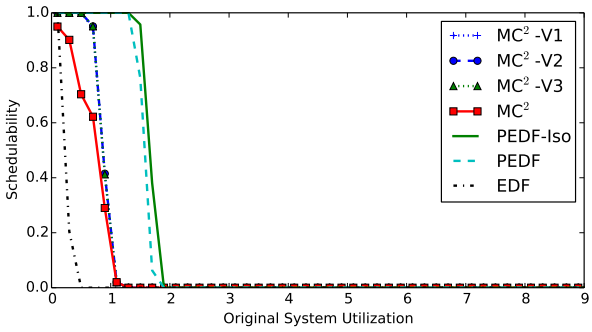
All-Mod., Long, Heavy, Mod., Const.



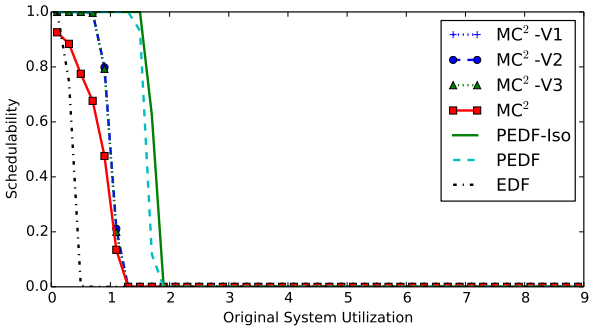
B-Heavy, Long, Light, Heavy, Const.



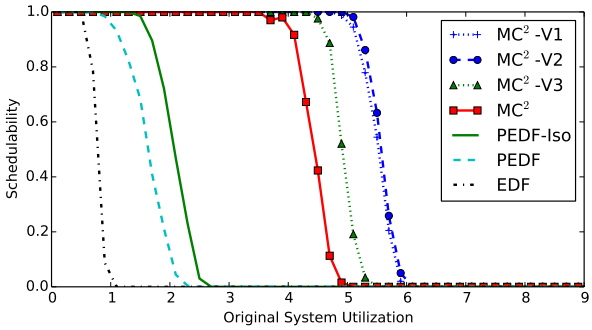
A-Heavy, Long, Mod., Light, Large Var.



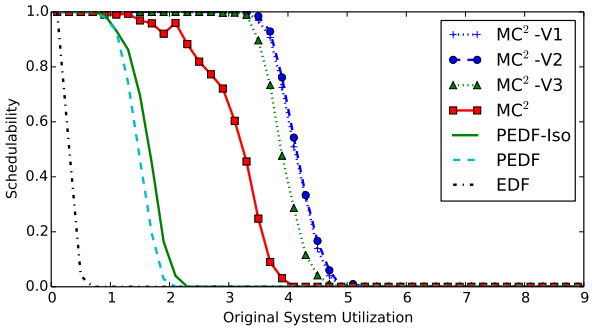
AC-Mod., Short, Light, Heavy, Const.



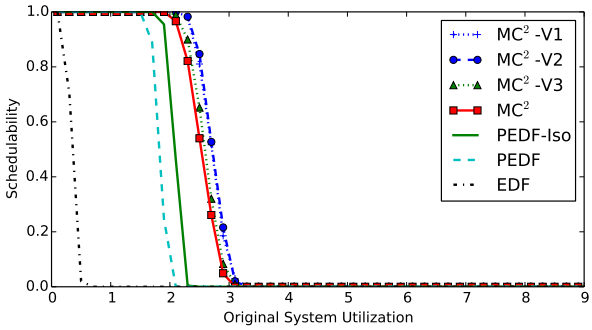
All-Mod., Short, Light, Mod., Const.



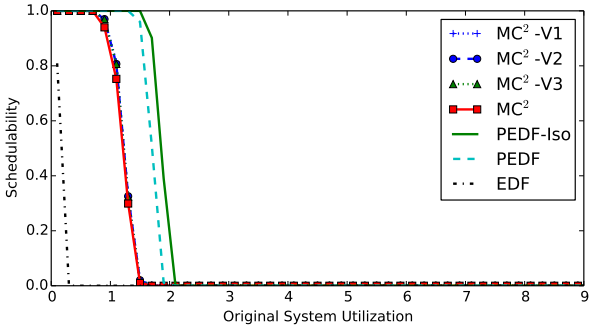
All-Mod., Long, Mod., Light, Const.



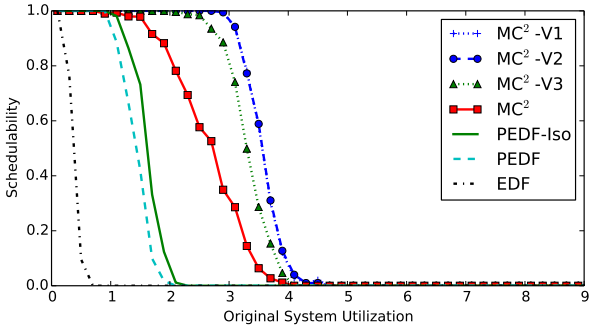
BC-Mod., Cont., Mod., Mod., Const.



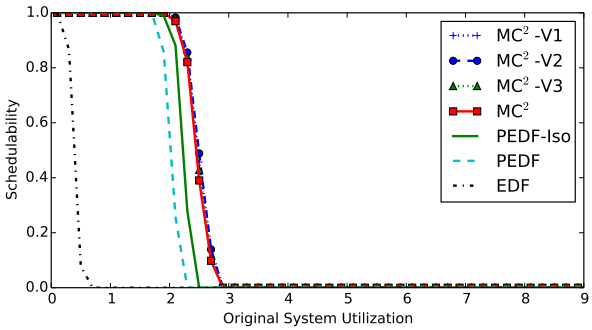
AB-Mod., Long, Light, Heavy, Large Var.



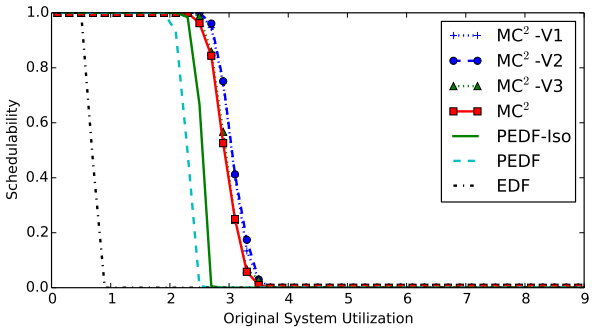
BC-Mod., Cont., Light, Heavy, Const.



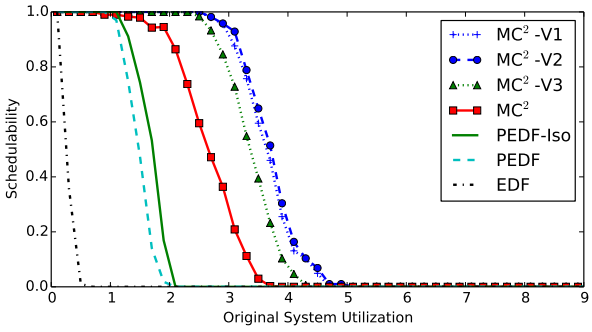
A-Heavy, Cont., Heavy, Mod., Const.



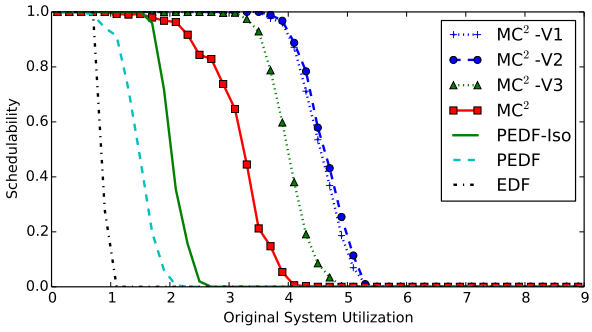
AC-Mod., Long, Light, Mod., Large Var.



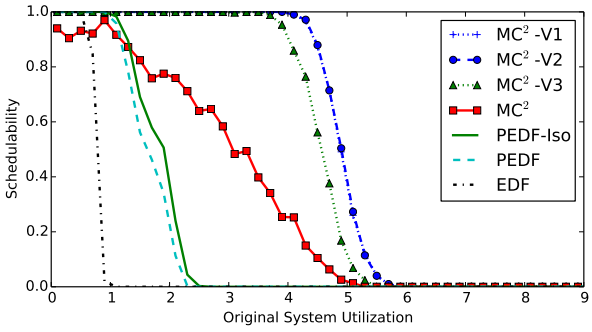
BC-Mod., Long, Light, Light, Large Var.



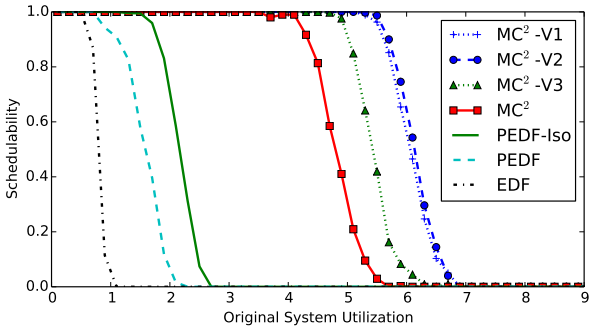
B-Heavy, Cont., Mod., Mod., Large Var.



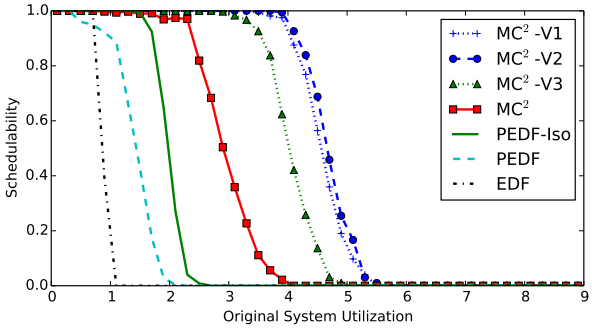
AB-Mod., Long, Heavy, Light, Const.



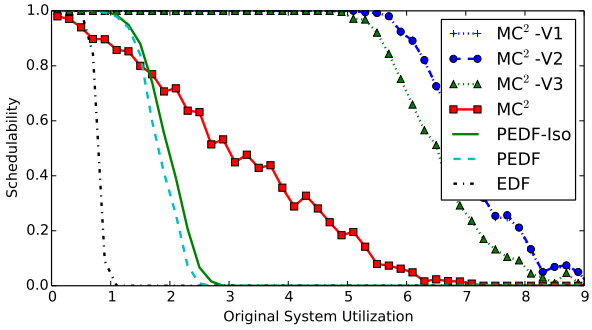
AC-Mod., Short, Heavy, Light, Const.



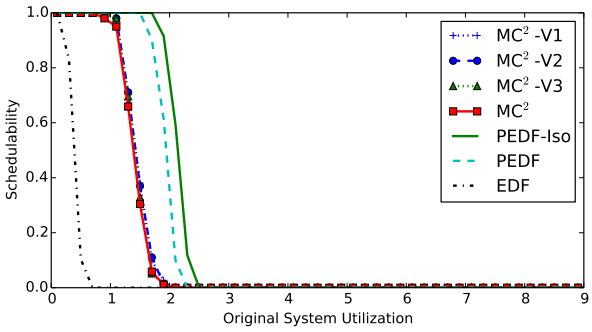
BC-Mod., Long, Mod., Light, Const.



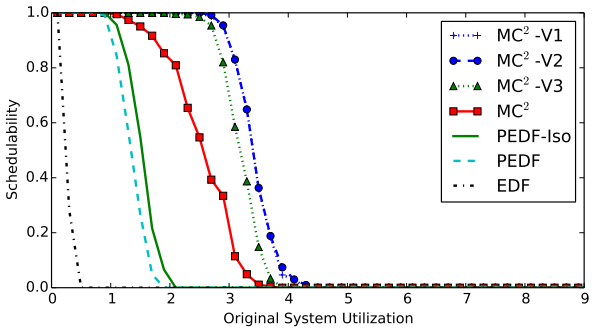
B-Heavy, Long, Heavy, Light, Const.



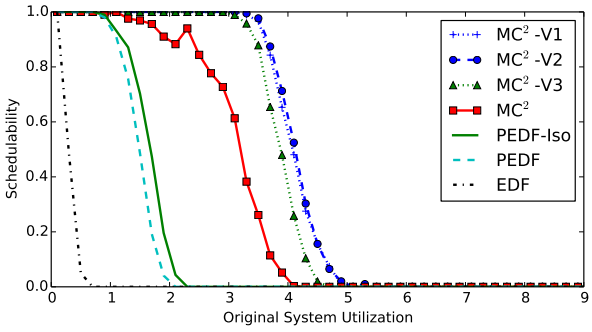
C-Heavy, Short, Heavy, Light, Const.



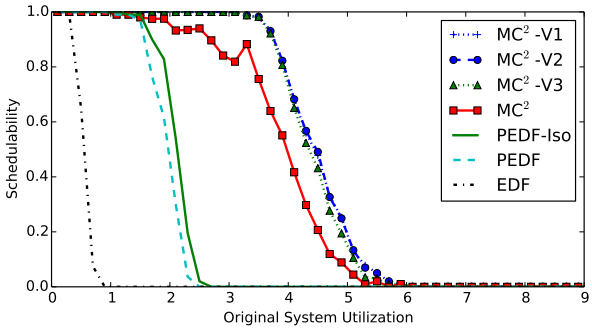
B-Heavy, Cont., Light, Light, Large Var.



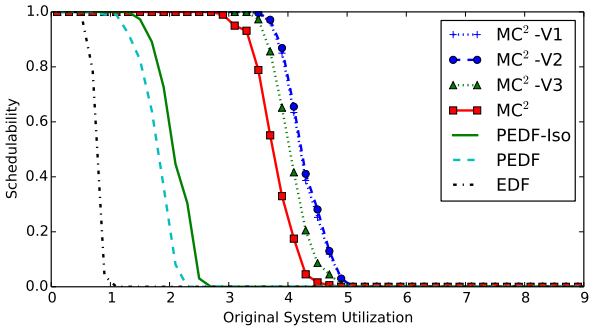
A-Heavy, Cont., Heavy, Heavy, Const.



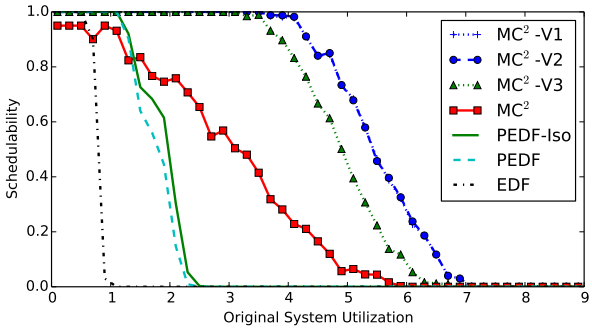
BC-Mod., Cont., Mod., Mod., Const.



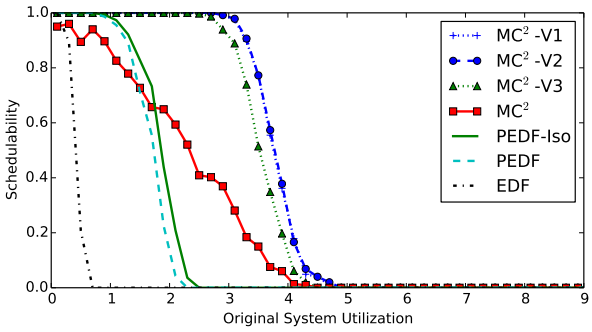
C-Heavy, Cont., Mod., Light, Const.



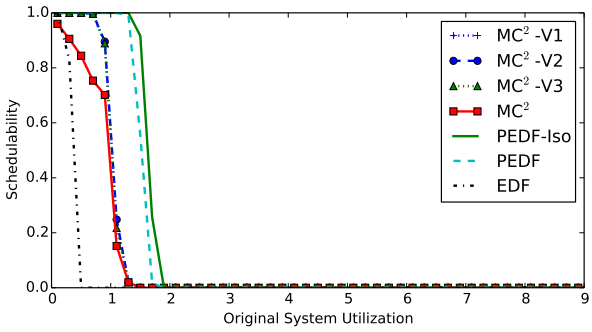
A-Heavy, Long, Mod., Light, Const.



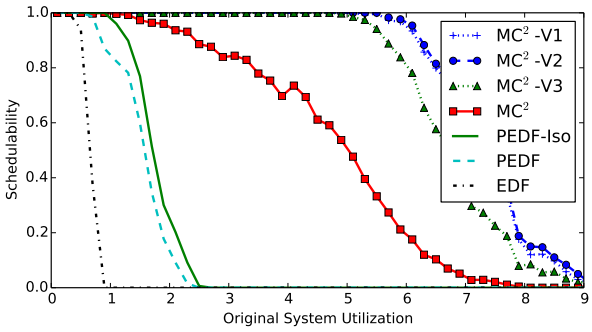
BC-Mod., Short, Heavy, Light, Large Var.



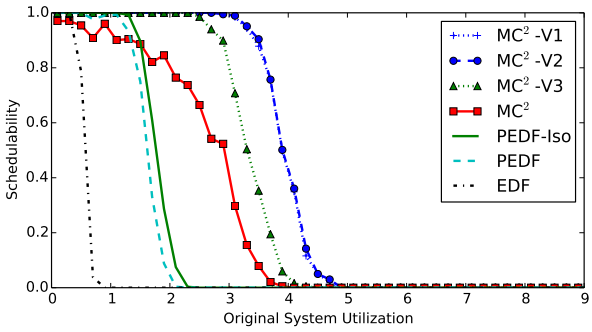
BC-Mod., Short, Mod., Heavy, Large Var.



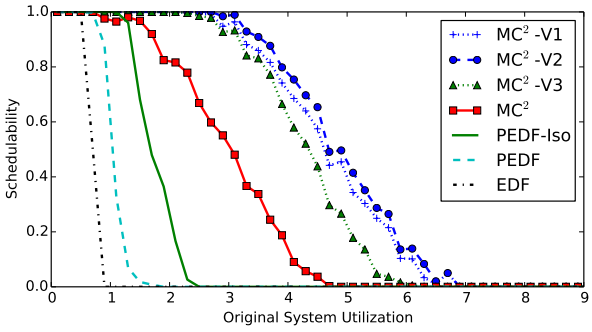
AB-Mod., Short, Light, Mod., Large Var.



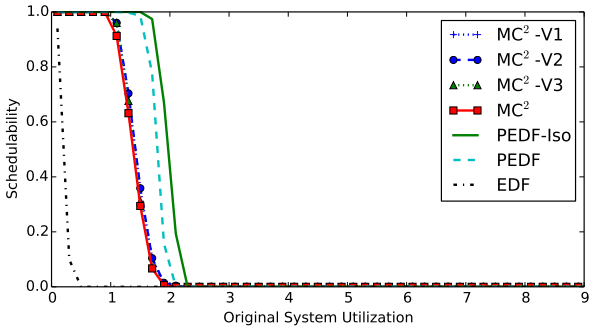
C-Heavy, Cont., Heavy, Light, Large Var.



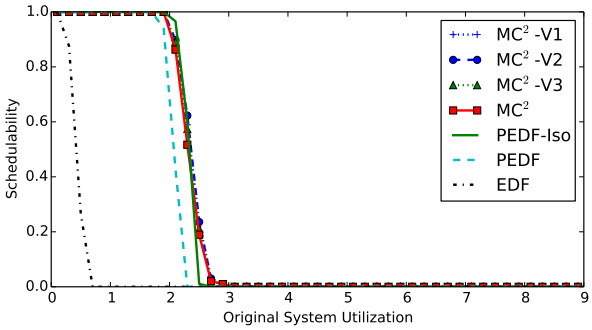
B-Heavy, Short, Heavy, Heavy, Const.



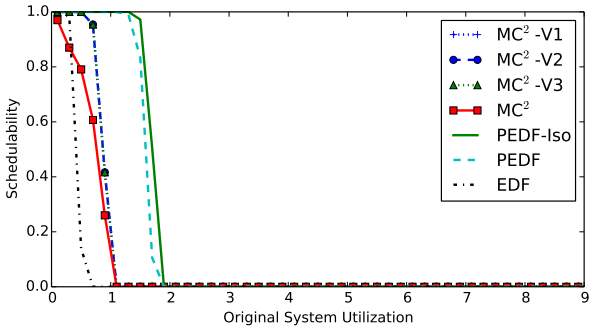
All-Mod., Long, Heavy, Mod., Large Var.



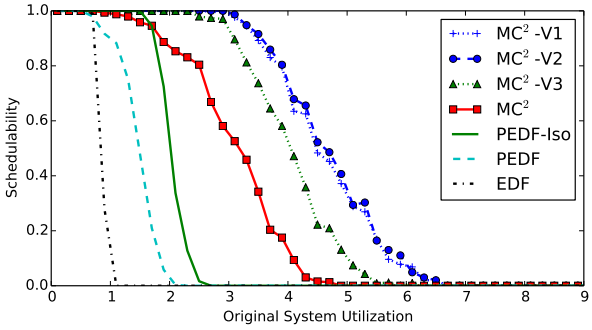
B-Heavy, Cont., Light, Mod., Large Var.



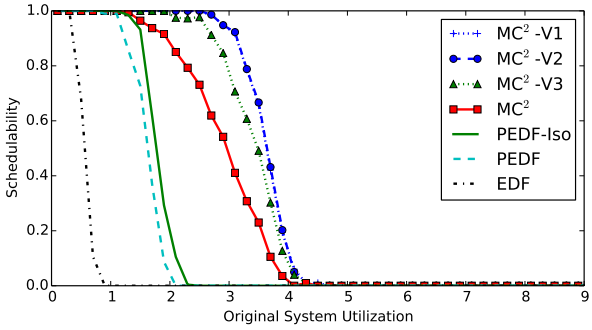
A-Heavy, Long, Light, Mod., Large Var.



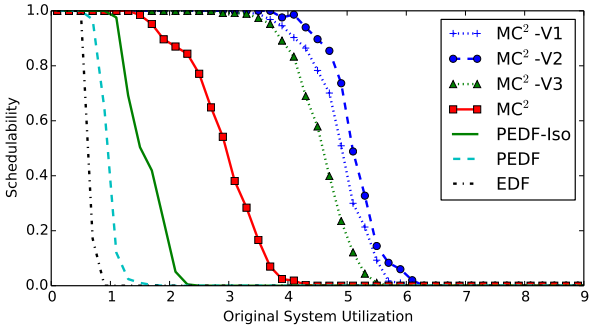
AC-Mod., Short, Light, Light, Const.



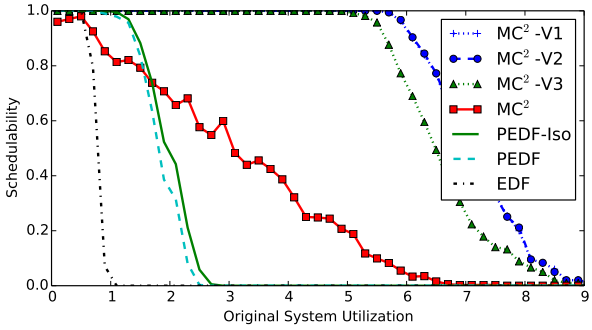
AB-Mod., Long, Heavy, Light, Large Var.



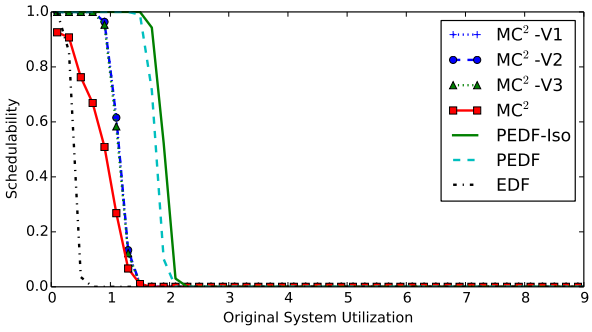
A-Heavy, Cont., Heavy, Light, Large Var.



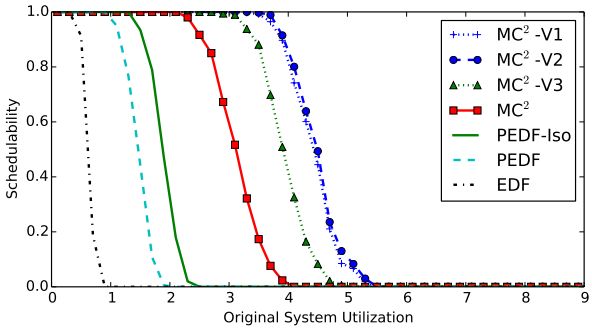
AC-Mod., Long, Heavy, Heavy, Const.



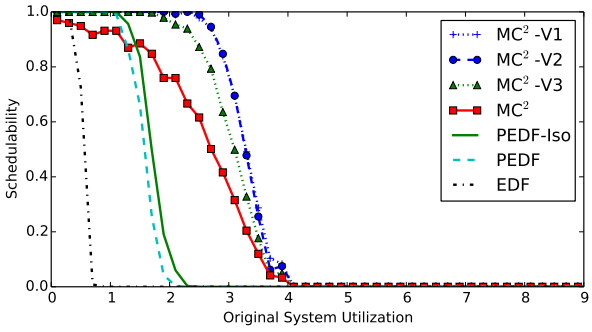
C-Heavy, Short, Heavy, Light, Const.



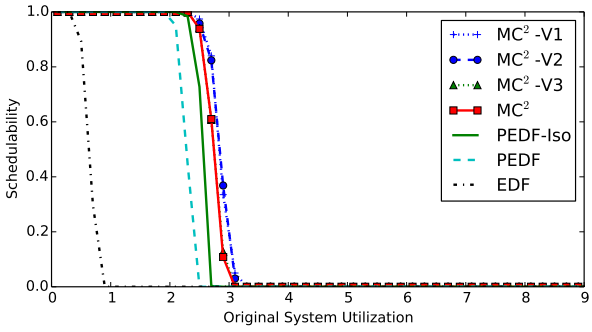
BC-Mod., Short, Light, Mod., Const.



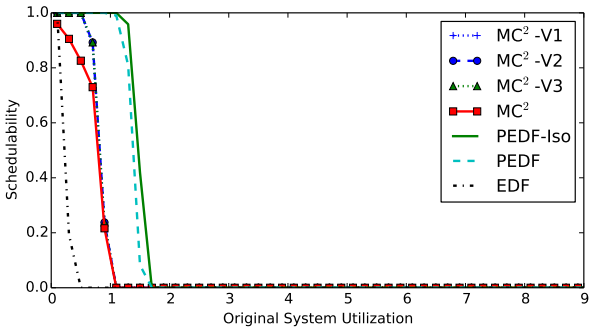
AB-Mod., Long, Mod., Mod., Const.



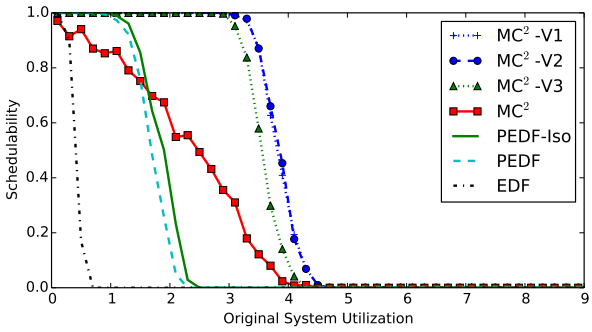
A-Heavy, Short, Heavy, Heavy, Large Var.



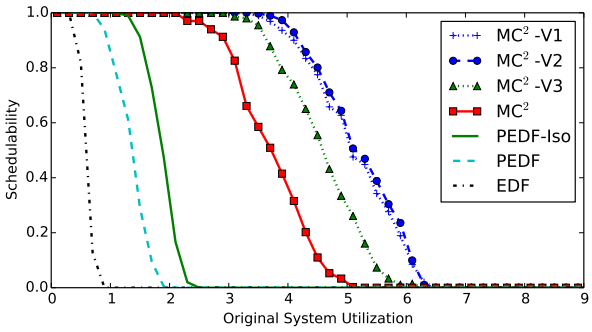
All-Mod., Long, Light, Light, Const.



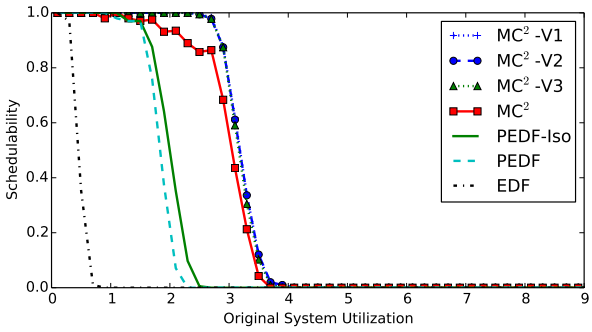
A-Heavy, Short, Light, Heavy, Const.



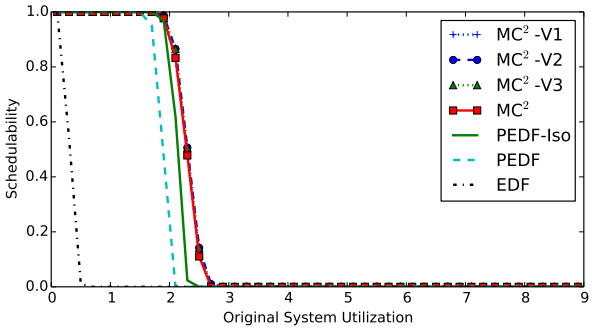
BC-Mod., Short, Mod., Heavy, Const.



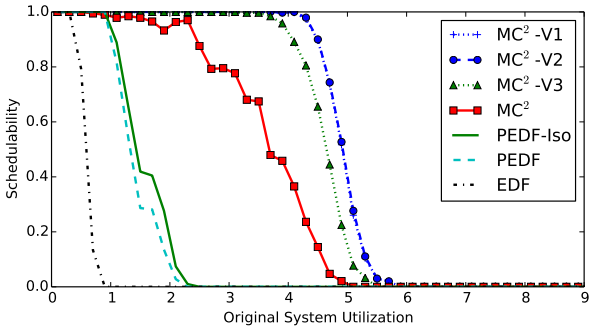
All-Mod., Long, Mod., Mod., Large Var.



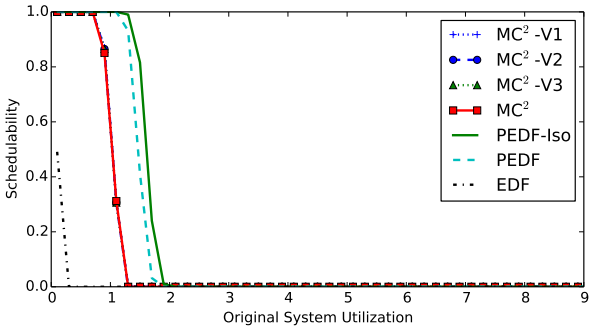
AC-Mod., Cont., Mod., Light, Large Var.



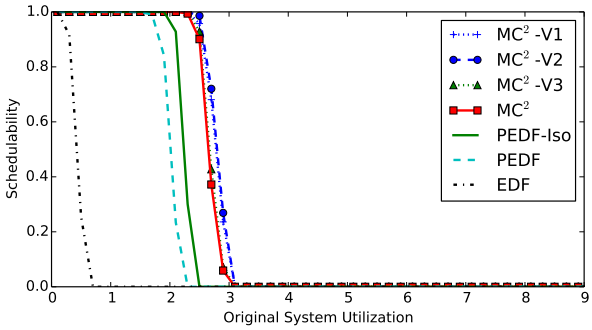
A-Heavy, Long, Light, Heavy, Const.



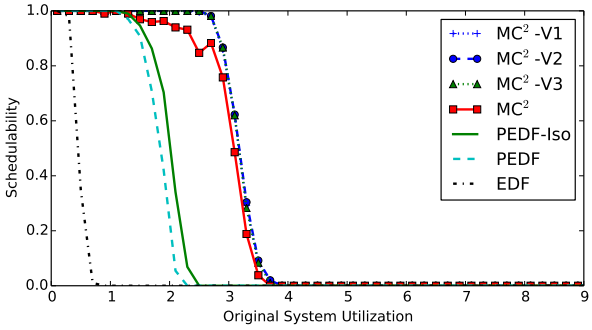
AC-Mod., Cont., Heavy, Light, Const.



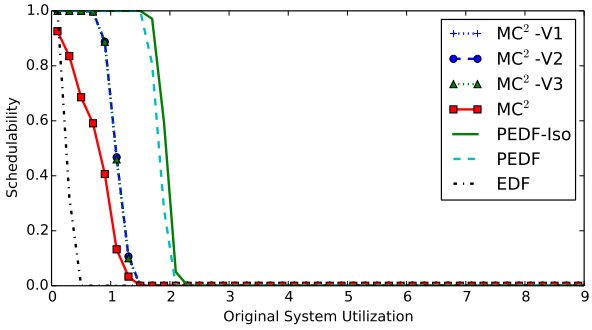
AB-Mod., Cont., Light, Heavy, Large Var.



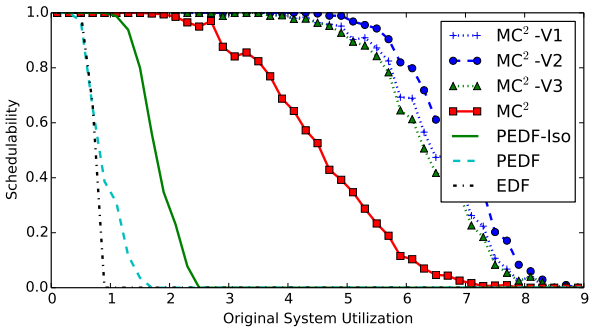
All-Mod., Long, Light, Mod., Const.



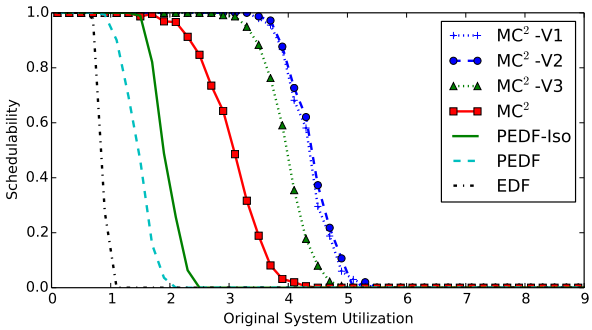
AC-Mod., Cont., Mod., Light, Const.



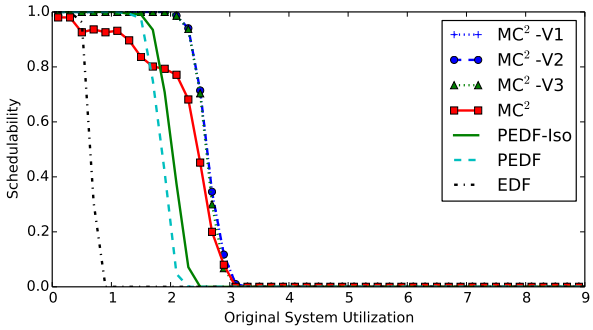
C-Heavy, Short, Light, Heavy, Const.



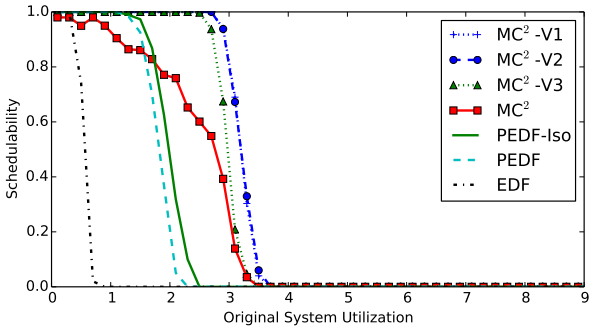
C-Heavy, Long, Heavy, Heavy, Const.



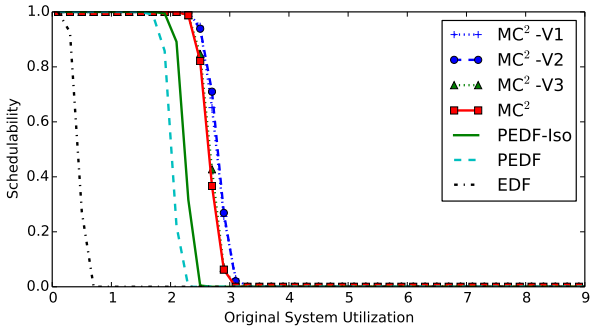
A-Heavy, Long, Heavy, Light, Const.



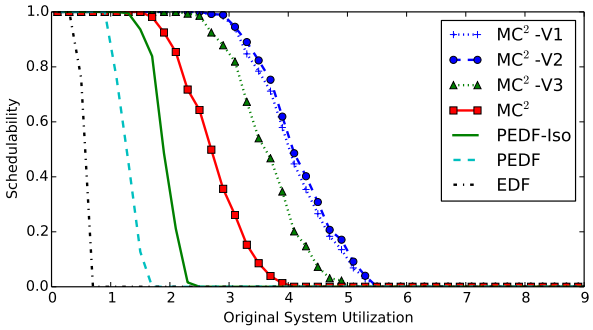
A-Heavy, Short, Mod., Light, Const.



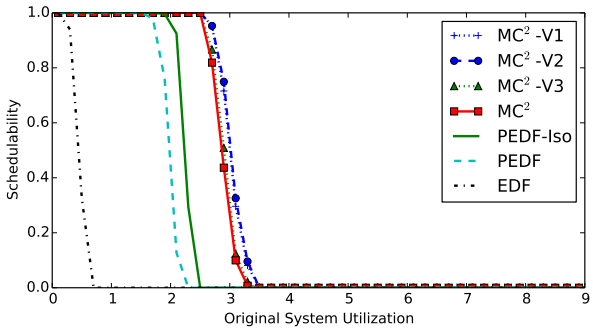
AB-Mod., Short, Mod., Mod., Const.



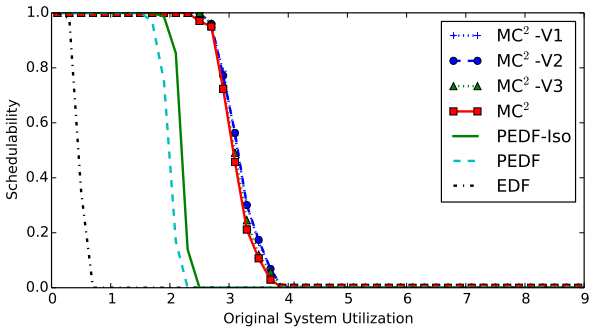
All-Mod., Long, Light, Mod., Const.



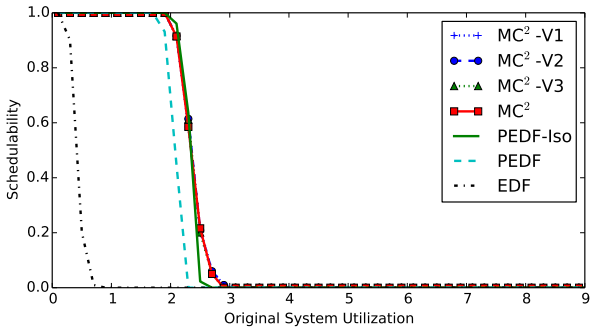
B-Heavy, Long, Mod., Heavy, Large Var.



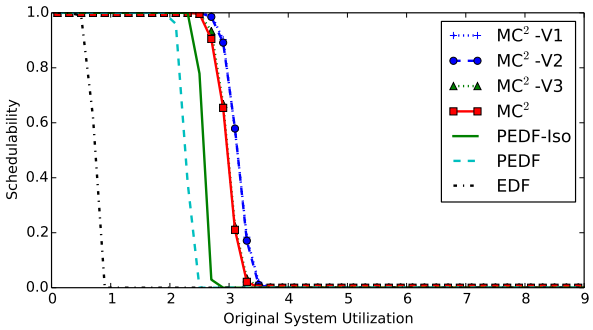
BC-Mod., Long, Light, Mod., Const.



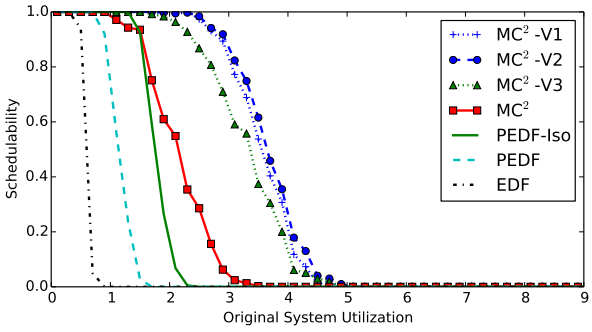
C-Heavy, Long, Light, Mod., Const.



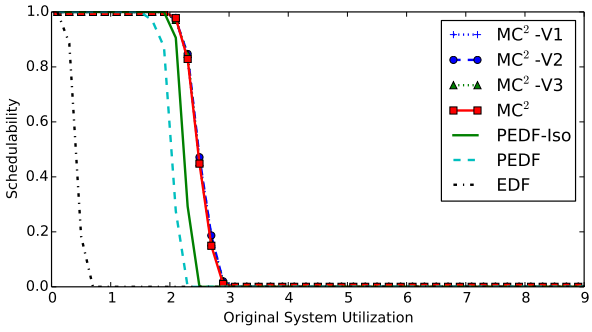
A-Heavy, Long, Light, Mod., Const.



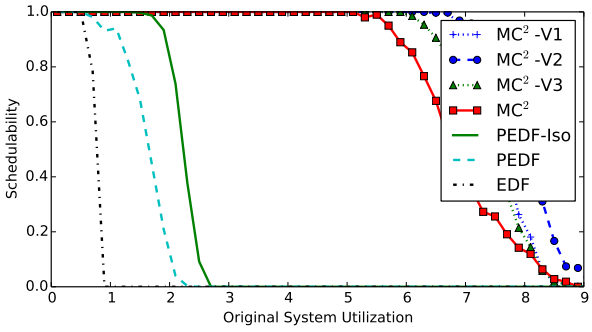
B-Heavy, Long, Light, Light, Const.



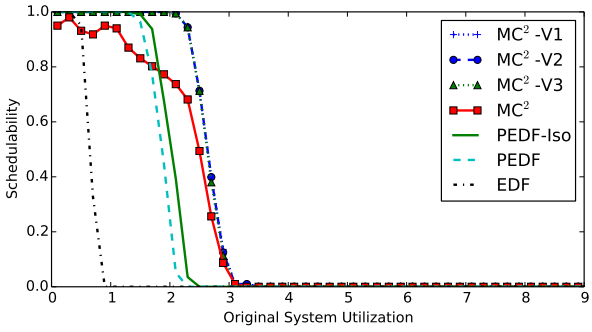
A-Heavy, Long, Heavy, Heavy, Large Var.



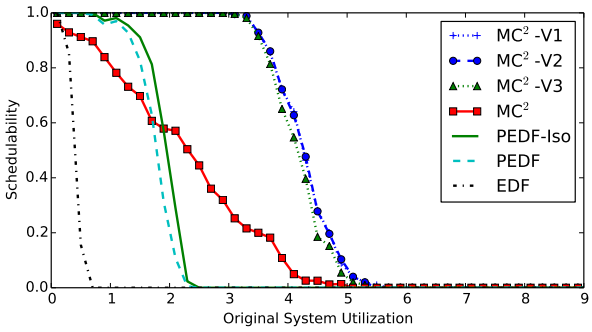
AC-Mod., Long, Light, Mod., Const.



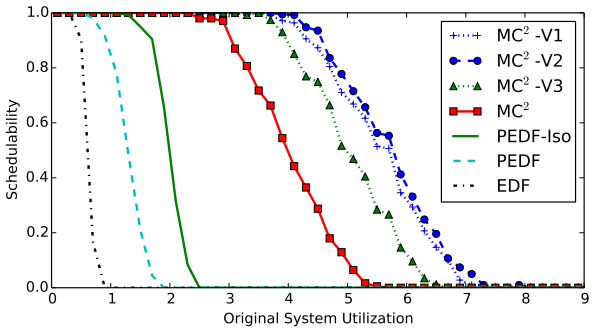
C-Heavy, Long, Mod., Light, Const.



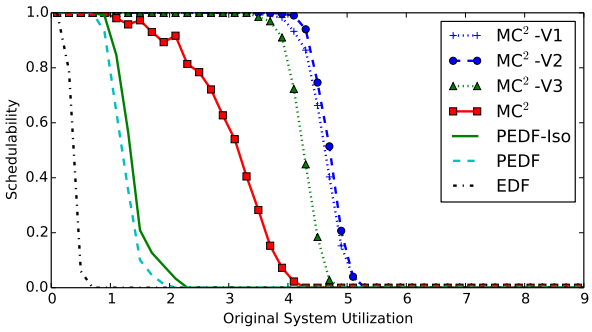
A-Heavy, Short, Mod., Light, Const.



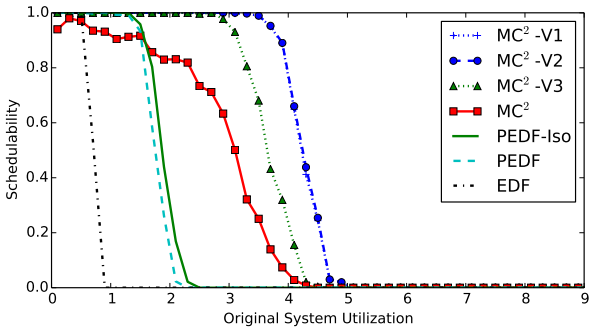
C-Heavy, Short, Mod., Heavy, Const.



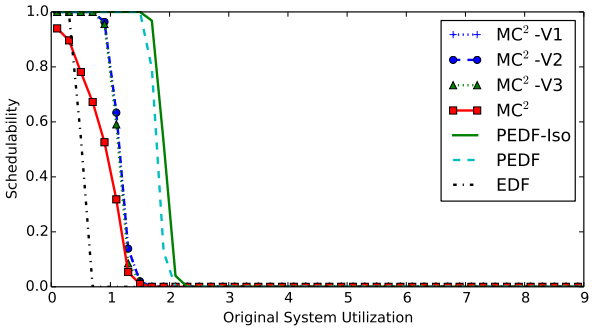
BC-Mod., Long, Mod., Mod., Large Var.



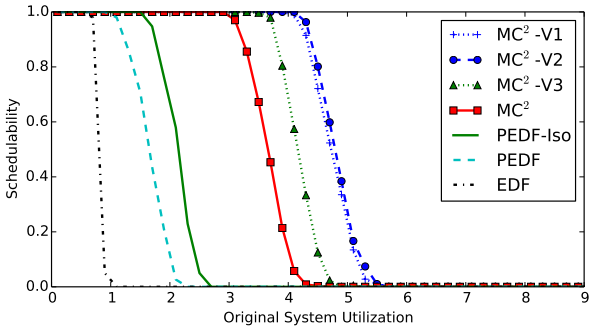
All-Mod., Cont., Heavy, Mod., Const.



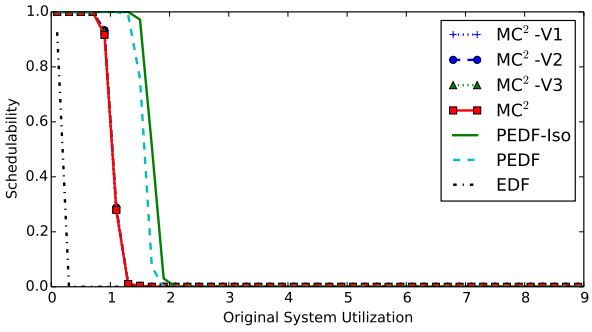
B-Heavy, Short, Heavy, Mod., Const.



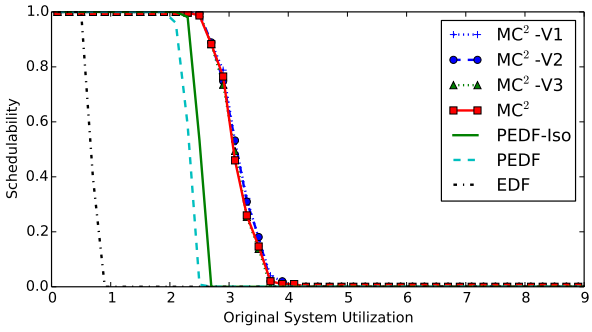
BC-Mod., Short, Light, Light, Const.



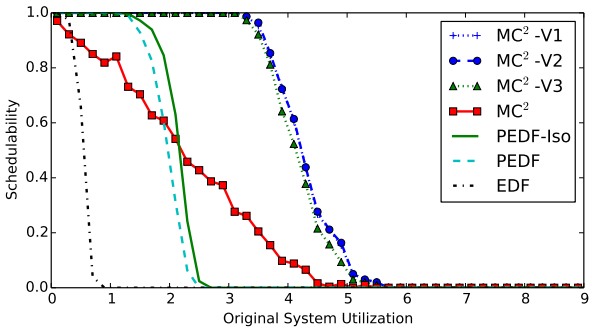
B-Heavy, Long, Mod., Light, Const.



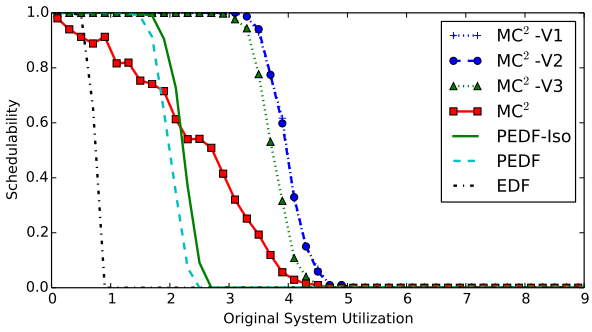
AB-Mod., Cont., Light, Mod., Const.



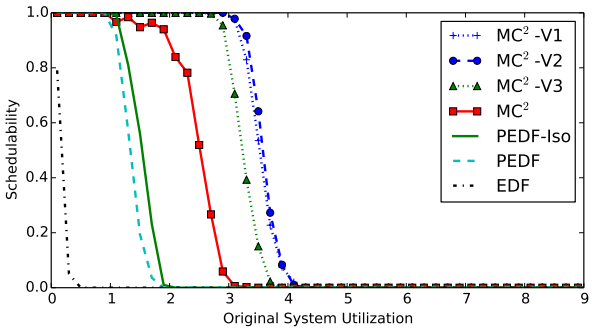
C-Heavy, Long, Light, Light, Const.



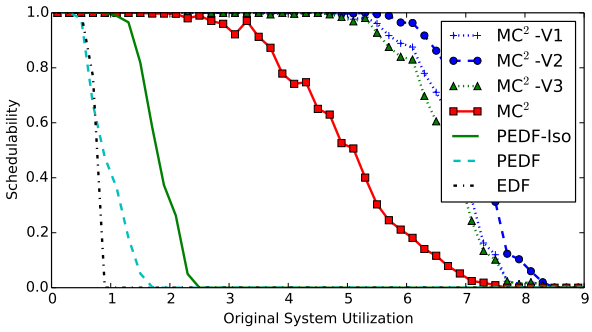
C-Heavy, Short, Mod., Mod., Large Var.



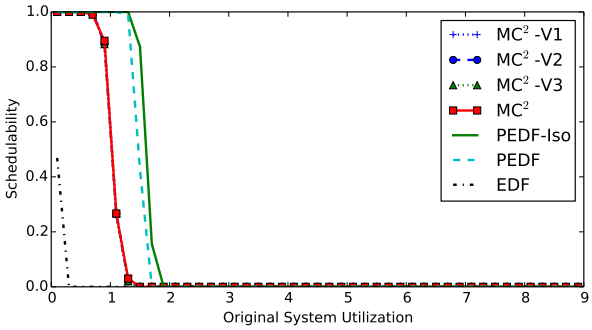
BC-Mod., Short, Mod., Light, Large Var.



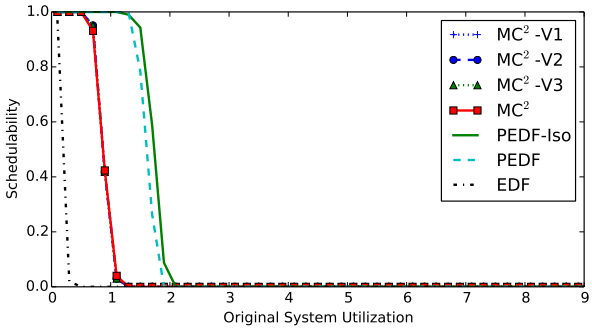
B-Heavy, Cont., Mod., Heavy, Const.



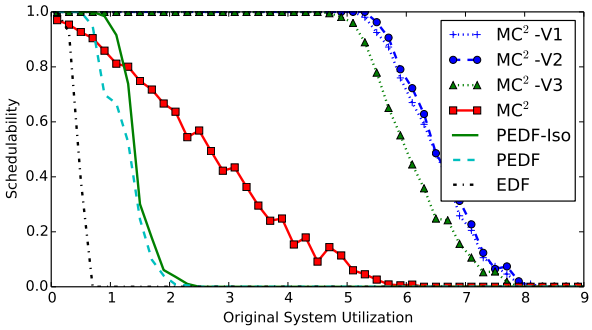
C-Heavy, Long, Heavy, Mod., Const.



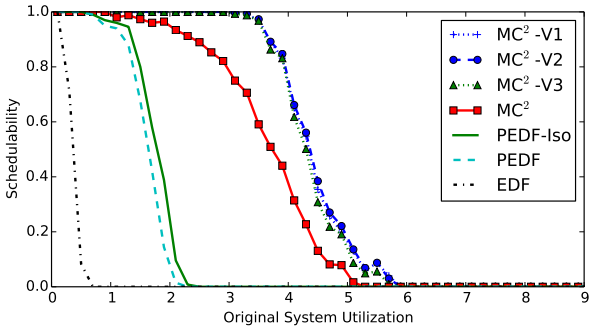
AB-Mod., Cont., Light, Heavy, Const.



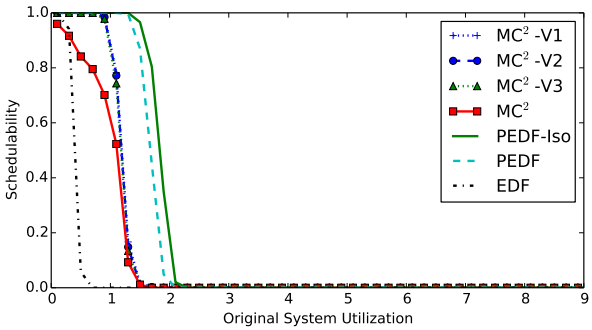
AC-Mod., Cont., Light, Mod., Large Var.



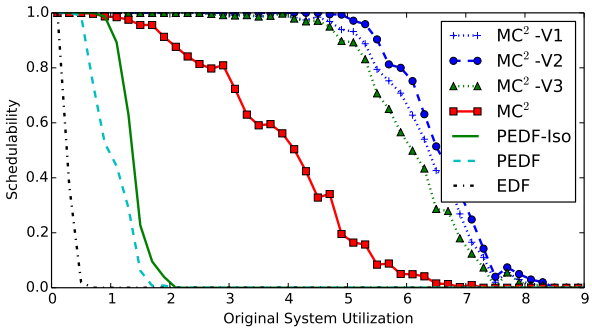
C-Heavy, Short, Heavy, Heavy, Const.



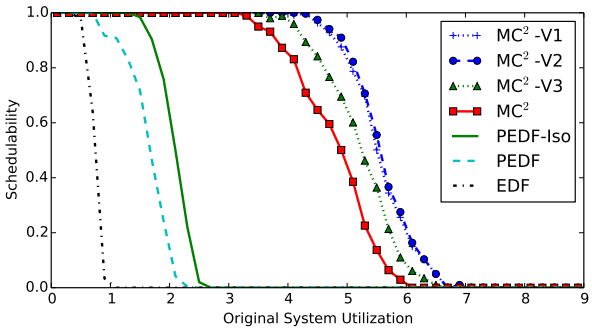
C-Heavy, Cont., Mod., Mod., Large Var.



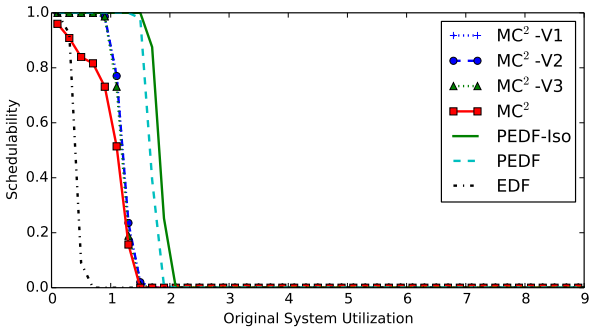
B-Heavy, Short, Light, Mod., Large Var.



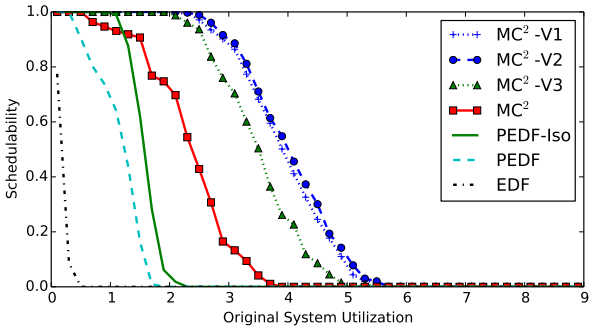
C-Heavy, Cont., Heavy, Heavy, Large Var.



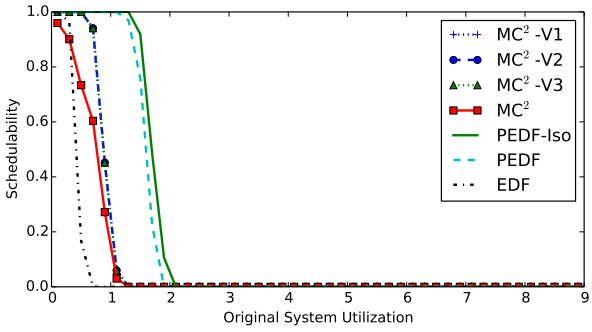
AC-Mod., Long, Mod., Light, Large Var.



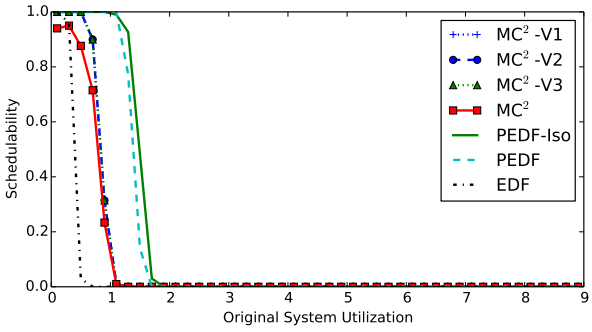
B-Heavy, Short, Light, Mod., Const.



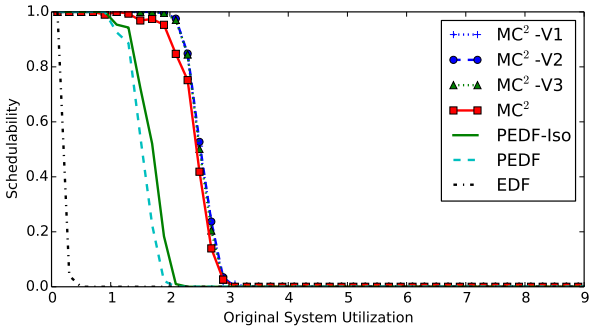
B-Heavy, Cont., Heavy, Heavy, Large Var.



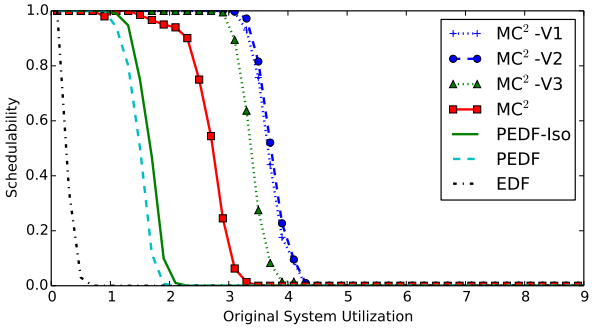
AC-Mod., Short, Light, Light, Large Var.



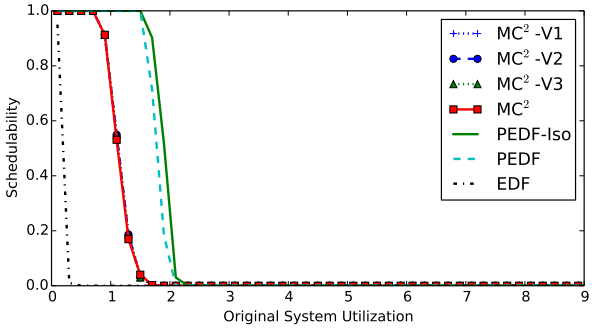
A-Heavy, Short, Light, Light, Large Var.



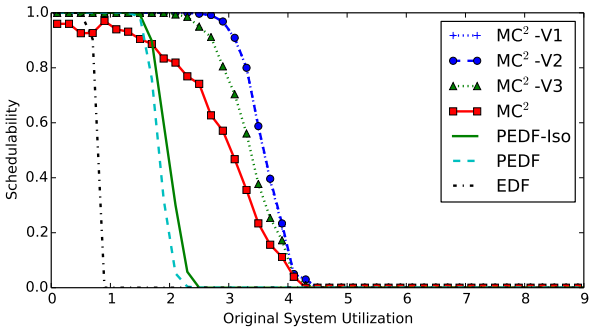
A-Heavy, Cont., Mod., Heavy, Large Var.



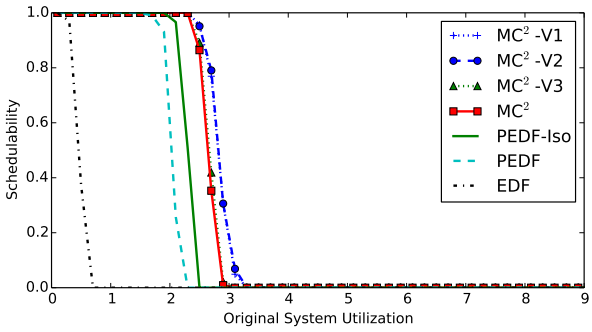
B-Heavy, Cont., Mod., Mod., Const.



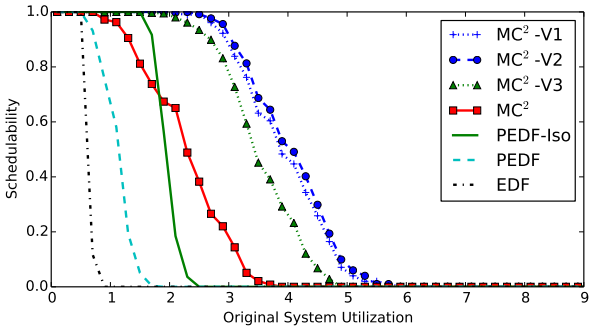
C-Heavy, Cont., Light, Heavy, Large Var.



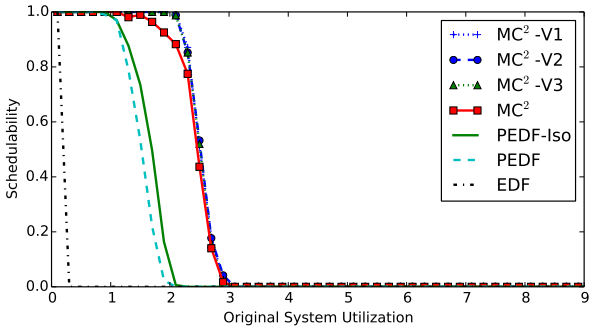
A-Heavy, Short, Heavy, Light, Large Var.



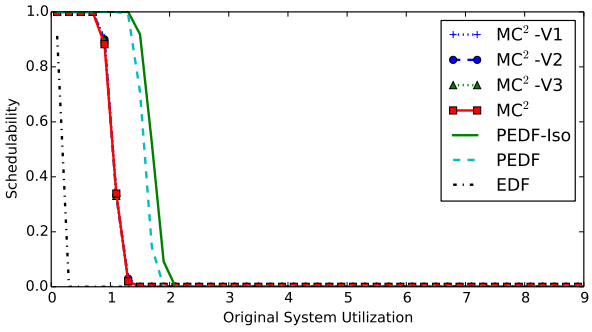
AB-Mod., Long, Light, Mod., Const.



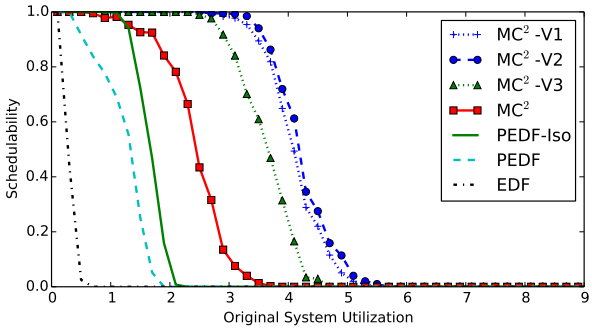
AB-Mod., Long, Heavy, Heavy, Large Var.



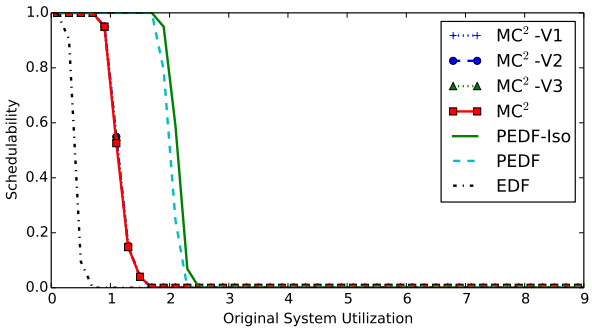
A-Heavy, Cont., Mod., Heavy, Const.



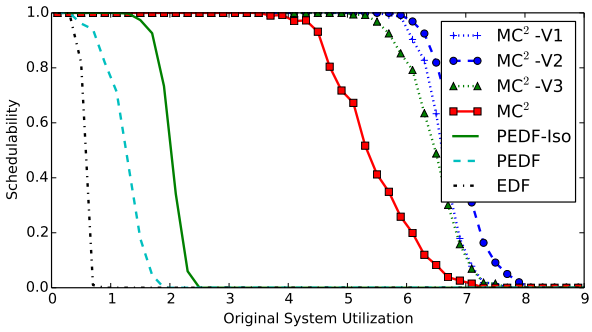
AB-Mod., Cont., Light, Mod., Large Var.



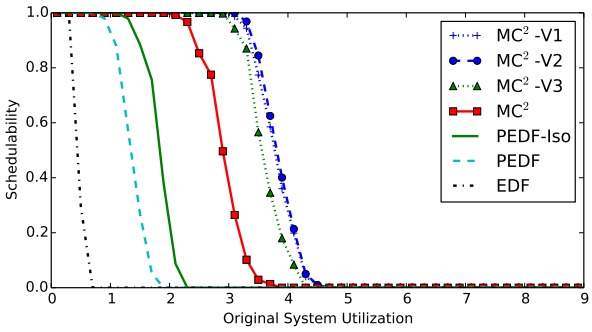
B-Heavy, Cont., Heavy, Mod., Large Var.



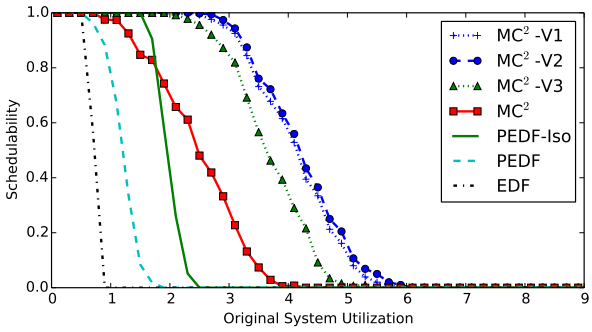
C-Heavy, Cont., Light, Light, Const.



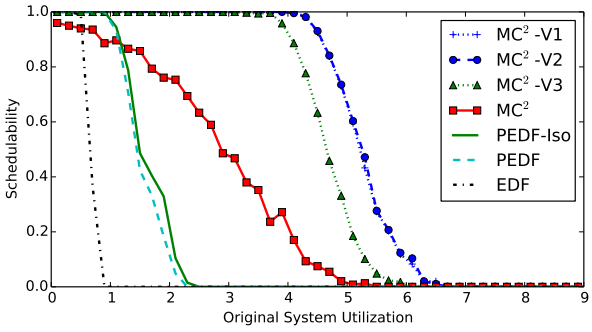
C-Heavy, Long, Mod., Heavy, Const.



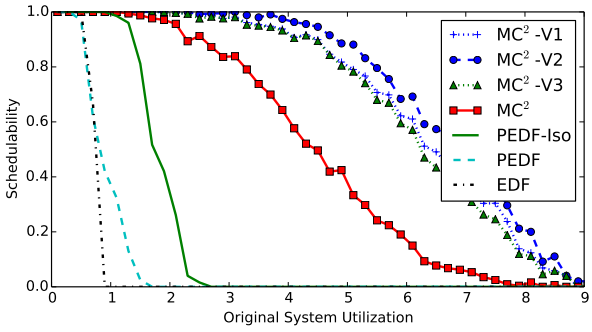
A-Heavy, Long, Mod., Heavy, Const.



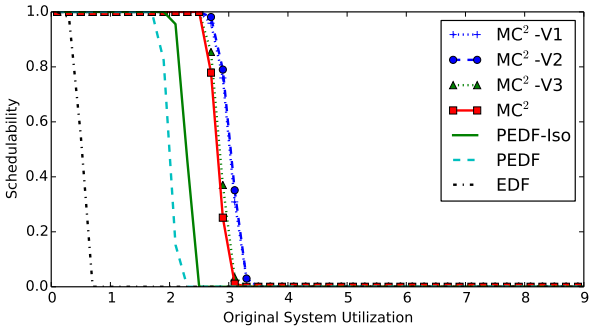
AB-Mod., Long, Heavy, Mod., Large Var.



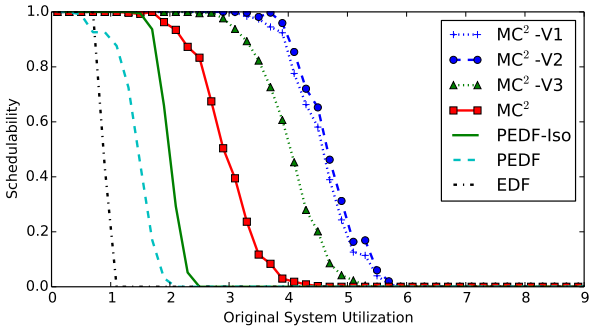
BC-Mod., Short, Heavy, Mod., Large Var.



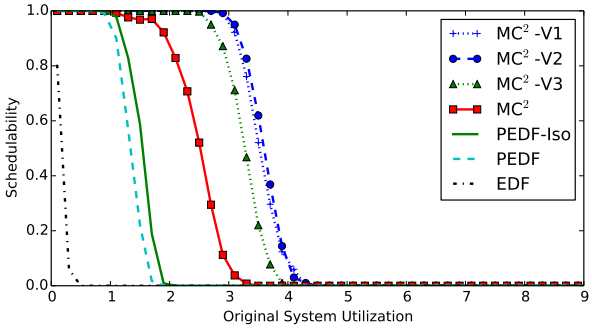
C-Heavy, Long, Heavy, Heavy, Large Var.



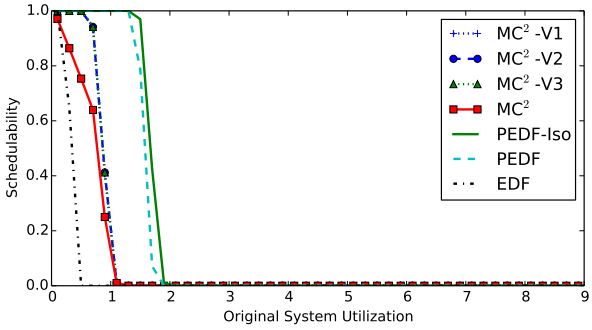
B-Heavy, Long, Light, Mod., Const.



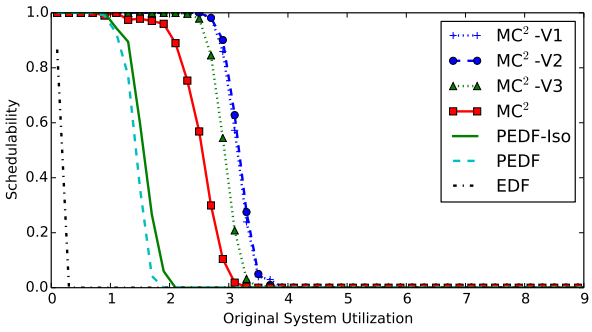
B-Heavy, Long, Heavy, Light, Const.



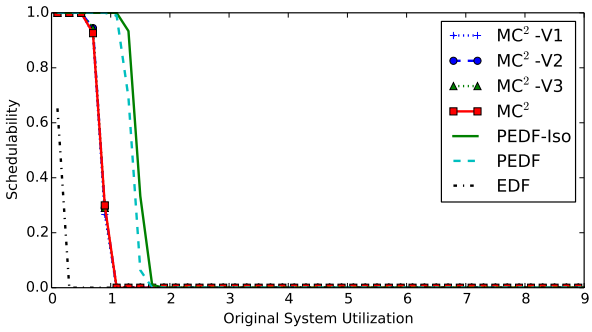
B-Heavy, Cont., Mod., Heavy, Const.



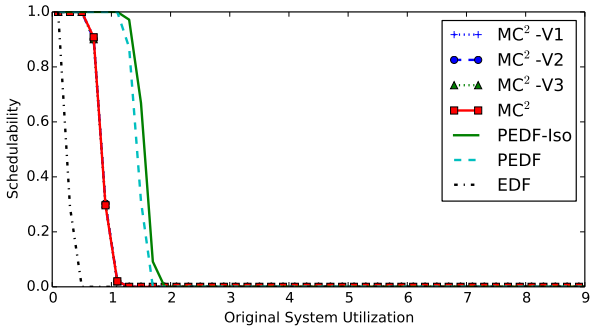
AC-Mod., Short, Light, Mod., Const.



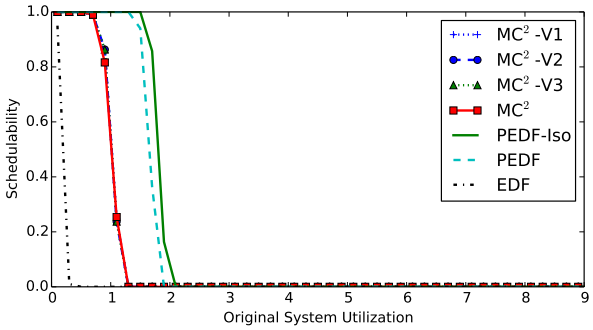
AB-Mod., Cont., Mod., Heavy, Const.



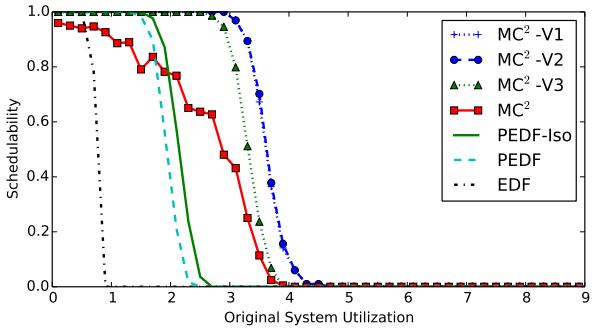
A-Heavy, Cont., Light, Heavy, Const.



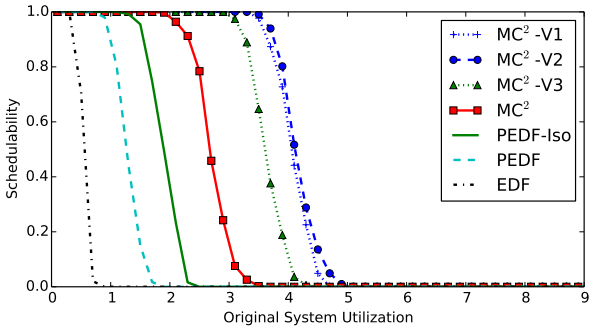
A-Heavy, Cont., Light, Light, Const.



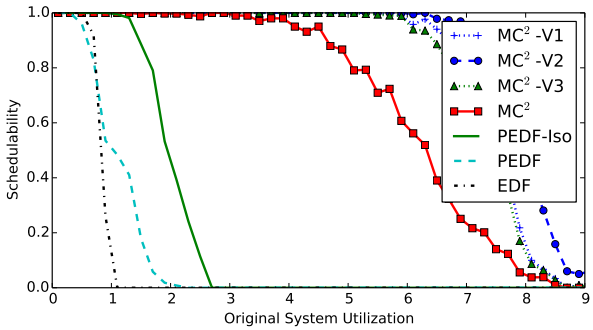
All-Mod., Cont., Light, Mod., Const.



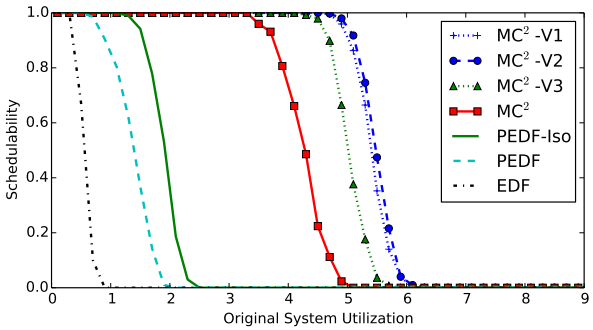
B-Heavy, Short, Mod., Light, Const.



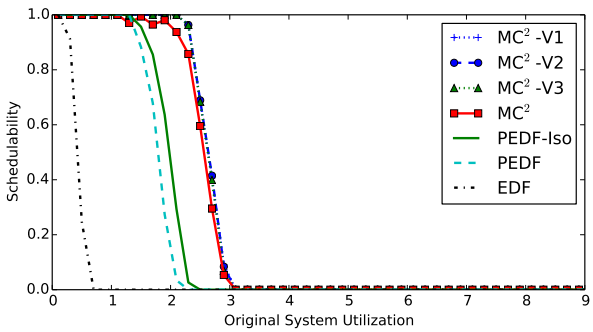
B-Heavy, Long, Mod., Heavy, Const.



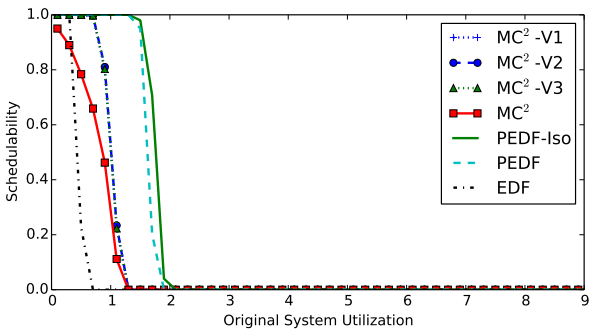
C-Heavy, Long, Heavy, Light, Const.



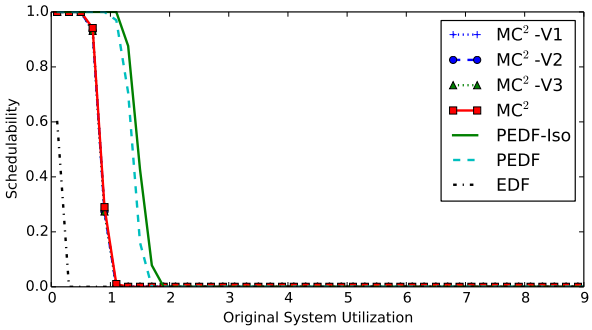
AC-Mod., Long, Mod., Mod., Const.



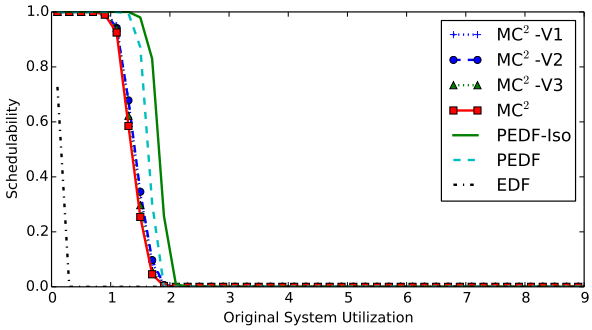
A-Heavy, Cont., Mod., Light, Const.



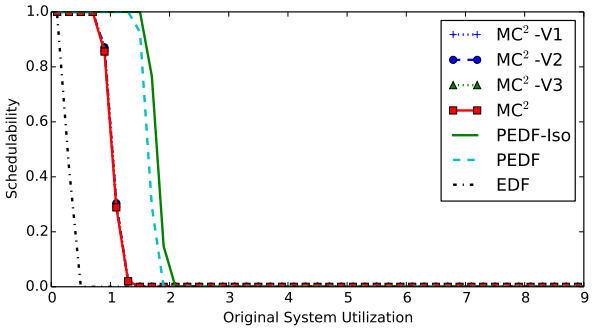
All-Mod., Short, Light, Light, Const.



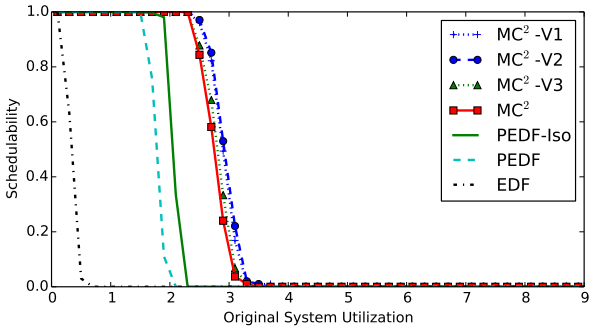
A-Heavy, Cont., Light, Heavy, Large Var.



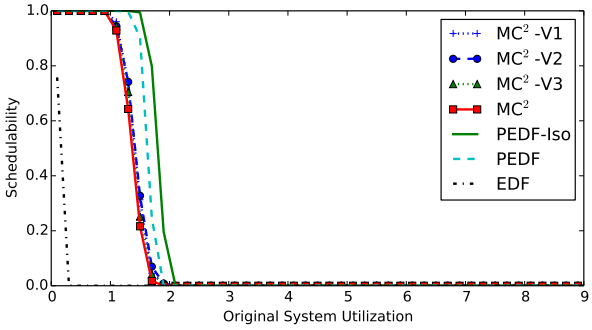
B-Heavy, Cont., Light, Heavy, Large Var.



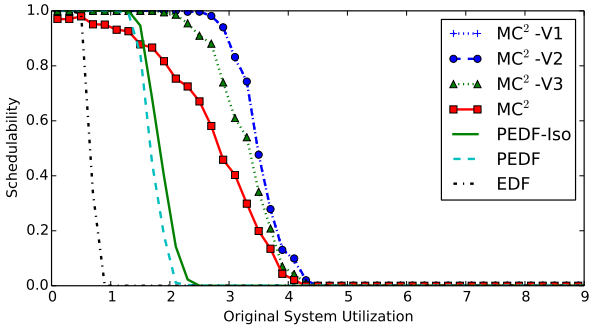
AB-Mod., Cont., Light, Light, Large Var.



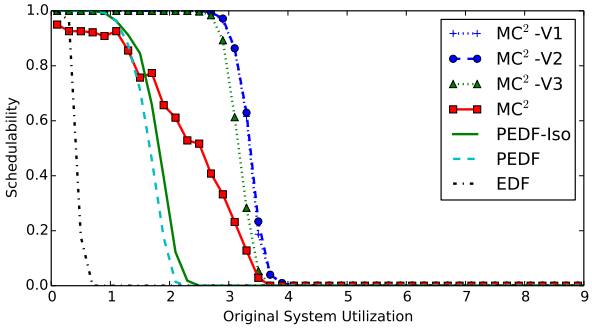
BC-Mod., Long, Light, Heavy, Large Var.



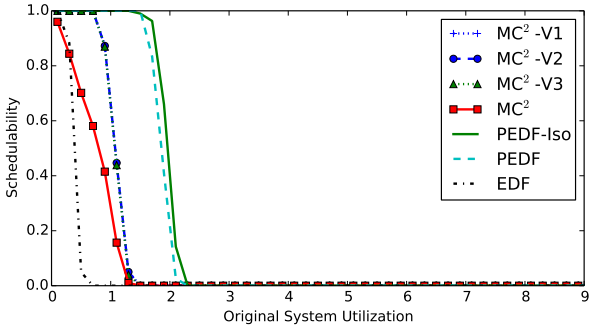
B-Heavy, Cont., Light, Heavy, Const.



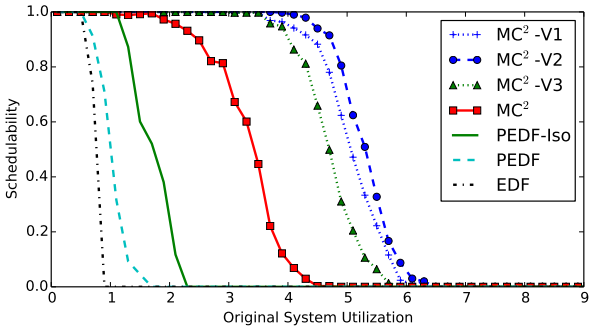
A-Heavy, Short, Heavy, Mod., Large Var.



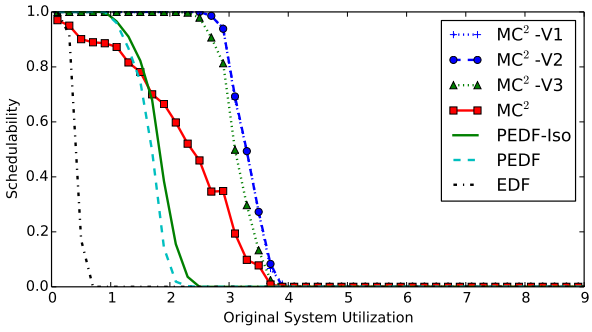
All-Mod., Short, Mod., Heavy, Large Var.



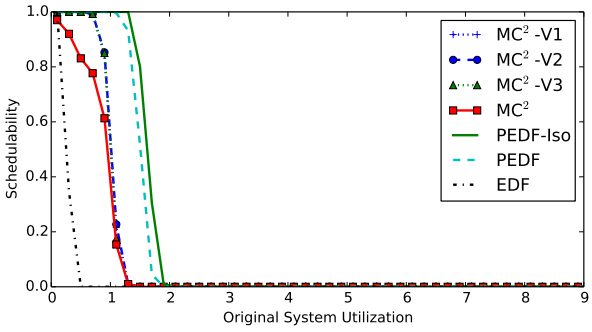
C-Heavy, Short, Light, Mod., Large Var.



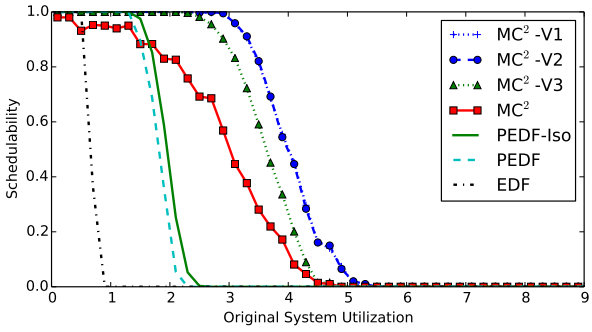
BC-Mod., Long, Heavy, Mod., Const.



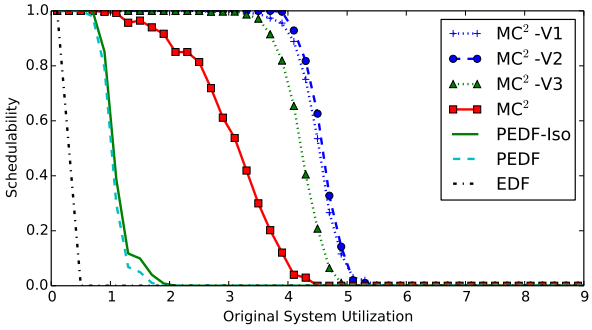
All-Mod., Short, Mod., Heavy, Large Var.



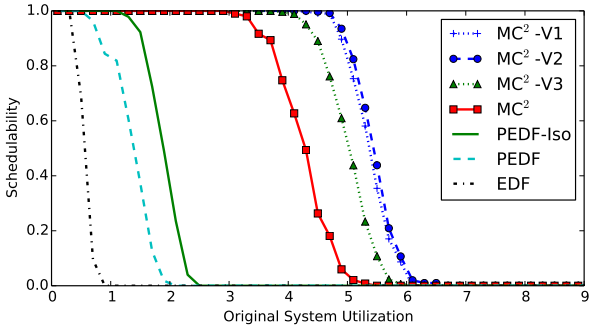
AB-Mod., Short, Light, Heavy, Large Var.



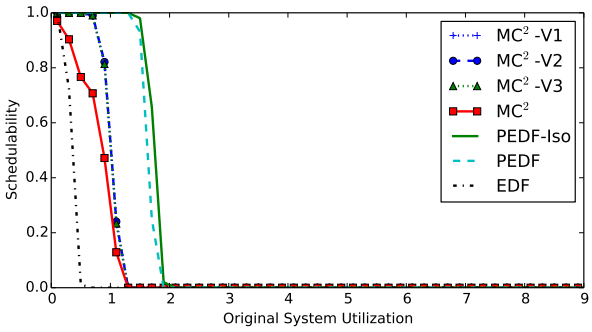
AB-Mod., Short, Heavy, Mod., Large Var.



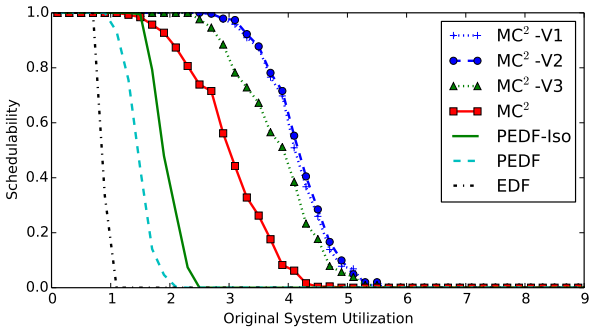
AC-Mod., Cont., Heavy, Heavy, Const.



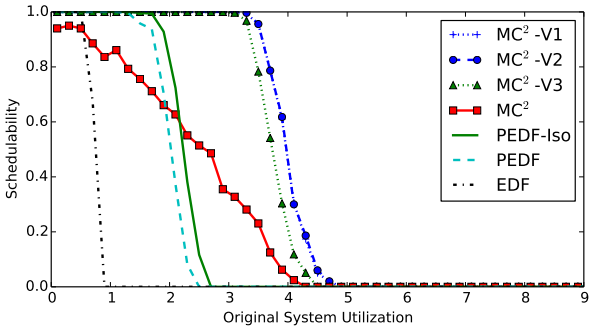
AC-Mod., Long, Mod., Mod., Const.



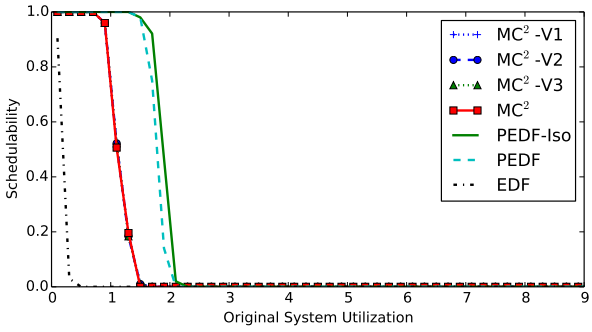
All-Mod., Short, Light, Mod., Const.



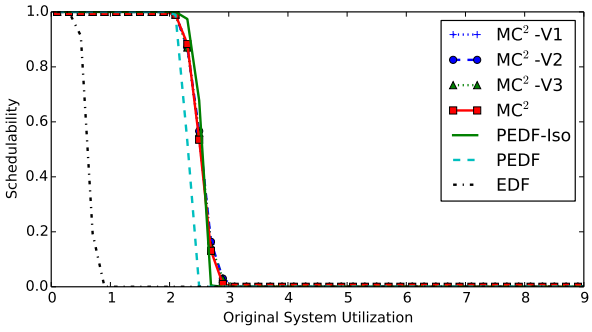
A-Heavy, Long, Heavy, Light, Large Var.



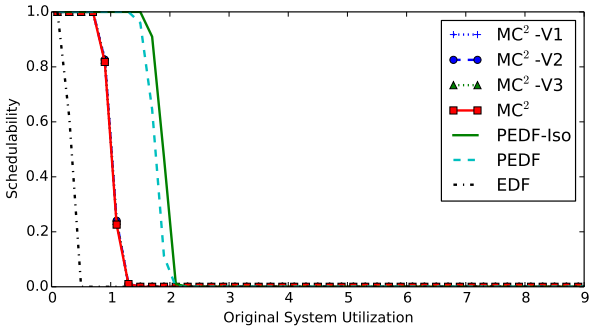
BC-Mod., Short, Mod., Light, Const.



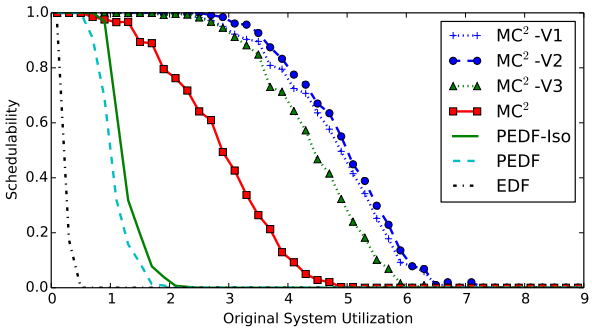
C-Heavy, Cont., Light, Heavy, Const.



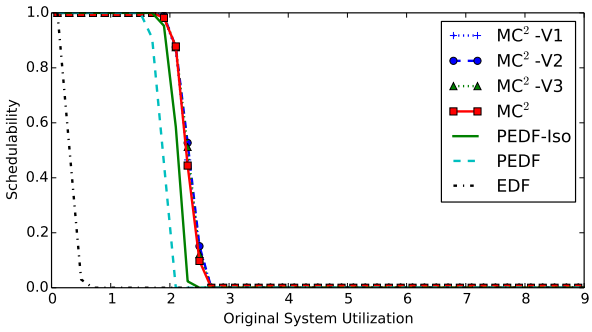
AC-Mod., Long, Light, Light, Const.



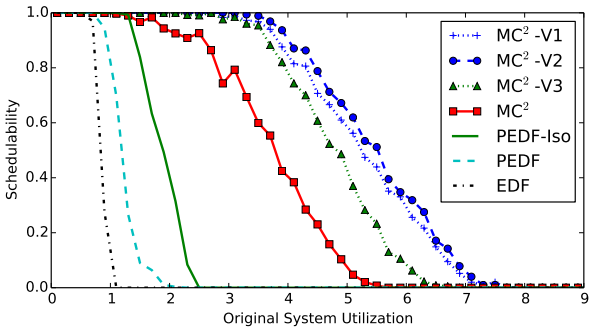
All-Mod., Cont., Light, Light, Large Var.



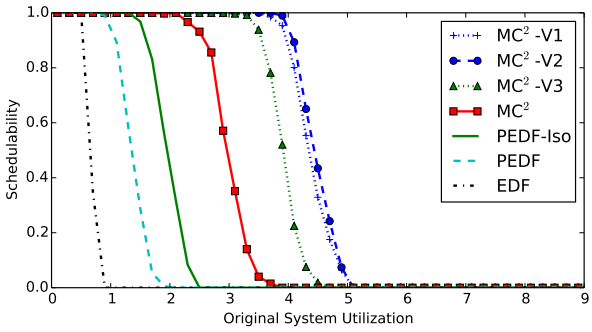
BC-Mod., Cont., Heavy, Heavy, Large Var.



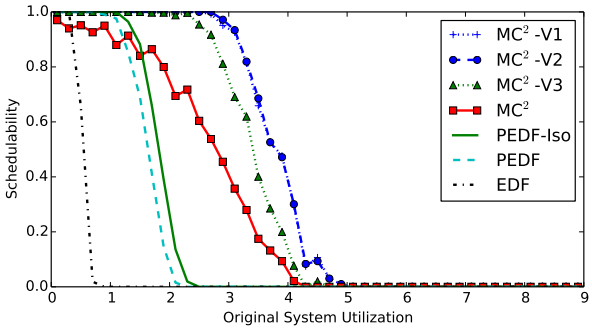
A-Heavy, Long, Light, Heavy, Large Var.



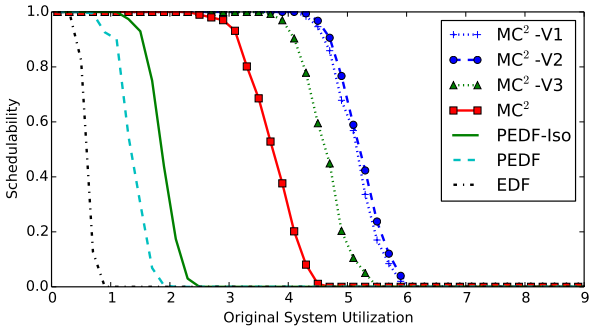
All-Mod., Long, Heavy, Light, Large Var.



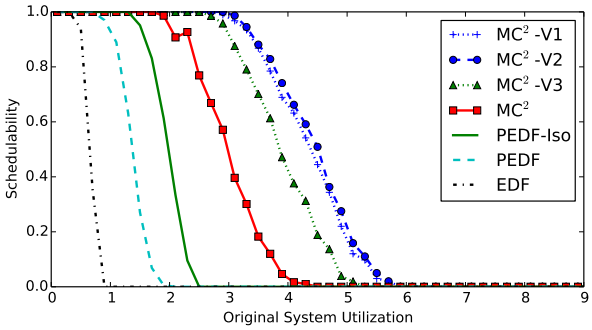
B-Heavy, Long, Mod., Mod., Const.



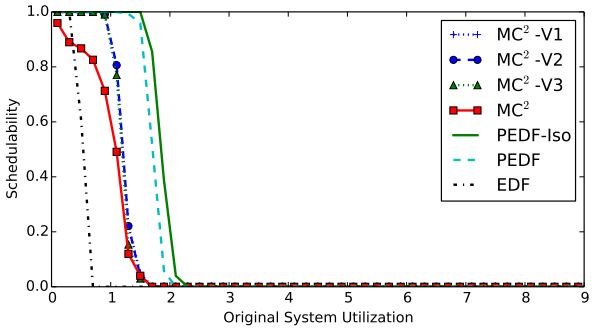
AB-Mod., Short, Heavy, Heavy, Large Var.



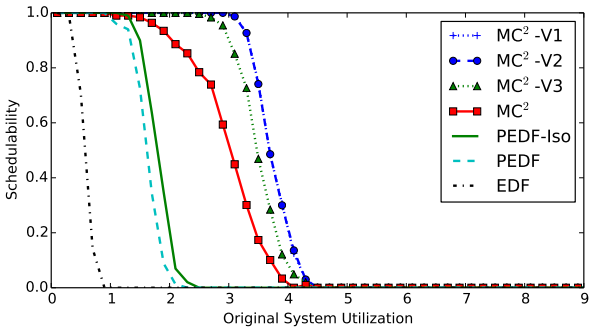
All-Mod., Long, Mod., Mod., Large Var.



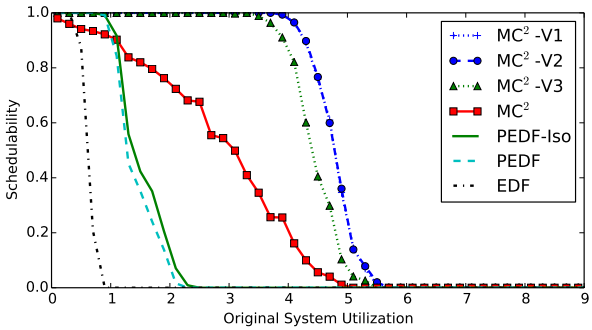
B-Heavy, Long, Mod., Mod., Large Var.



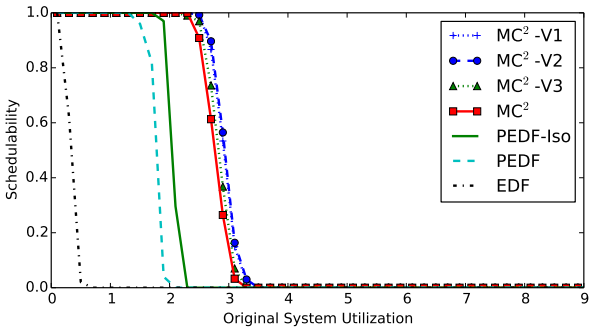
B-Heavy, Short, Light, Light, Large Var.



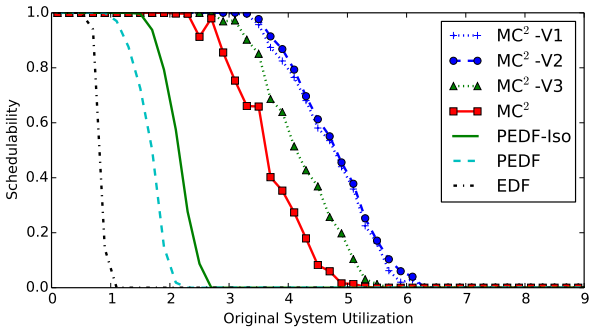
A-Heavy, Cont., Heavy, Light, Large Var.



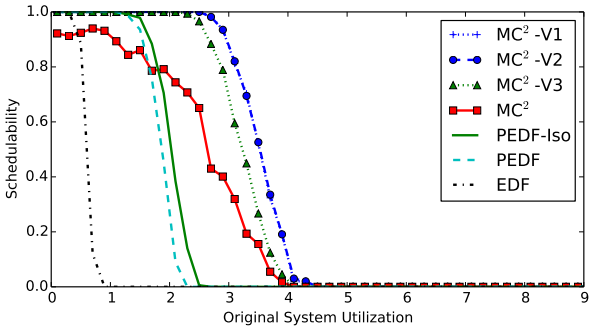
AC-Mod., Short, Heavy, Mod., Large Var.



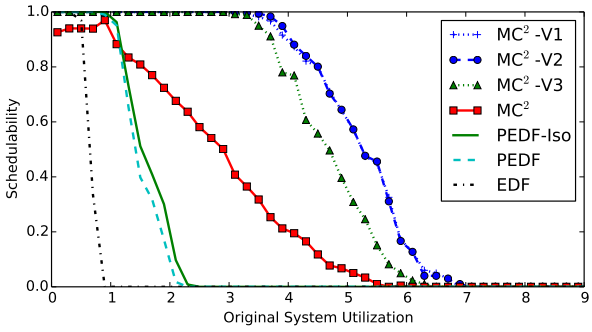
BC-Mod., Long, Light, Heavy, Large Var.



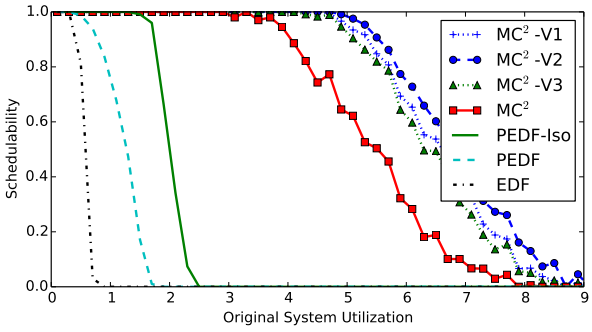
B-Heavy, Long, Mod., Light, Large Var.



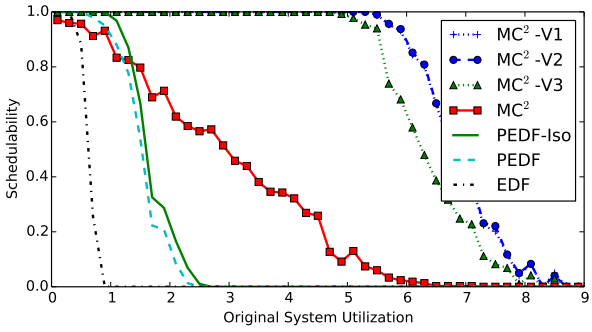
B-Heavy, Short, Mod., Mod., Large Var.



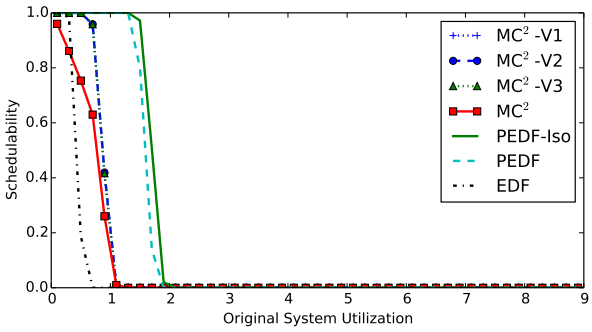
BC-Mod., Short, Heavy, Mod., Large Var.



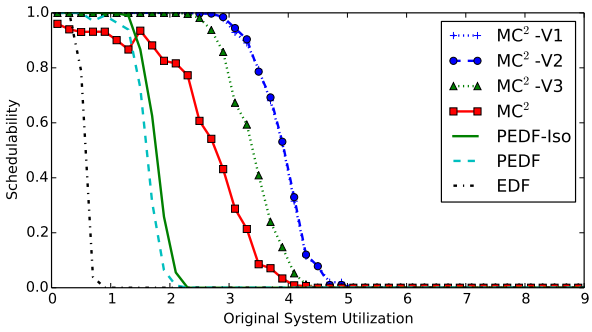
C-Heavy, Long, Mod., Heavy, Large Var.



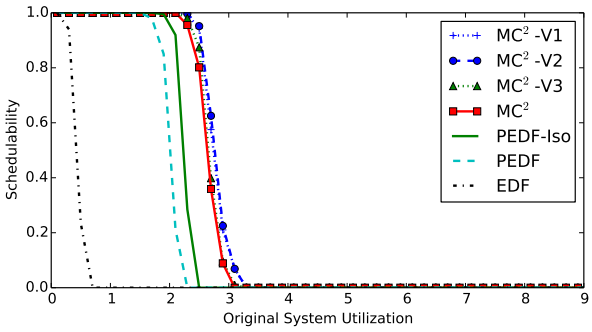
C-Heavy, Short, Heavy, Mod., Const.



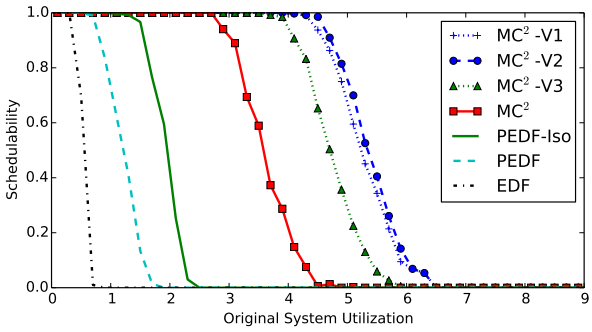
AC-Mod., Short, Light, Light, Const.



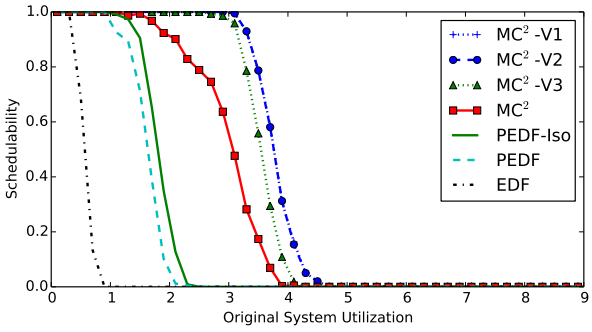
B-Heavy, Short, Heavy, Heavy, Const.



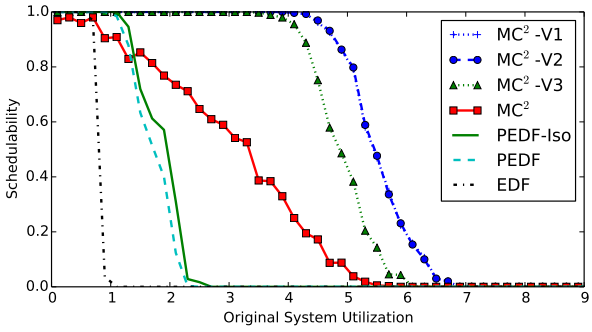
All-Mod., Long, Light, Mod., Large Var.



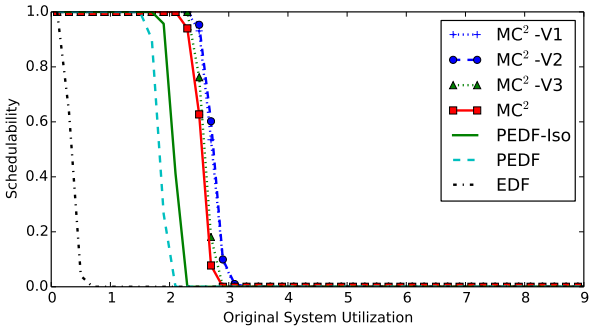
BC-Mod., Long, Mod., Heavy, Large Var.



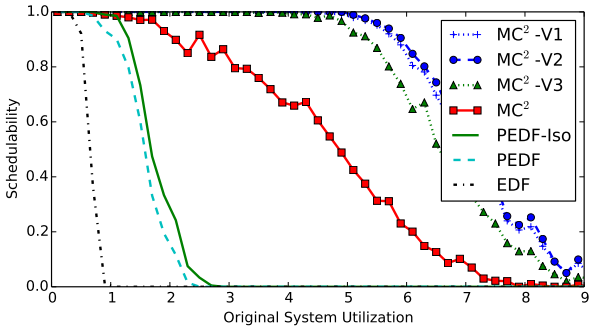
A-Heavy, Cont., Heavy, Light, Const.



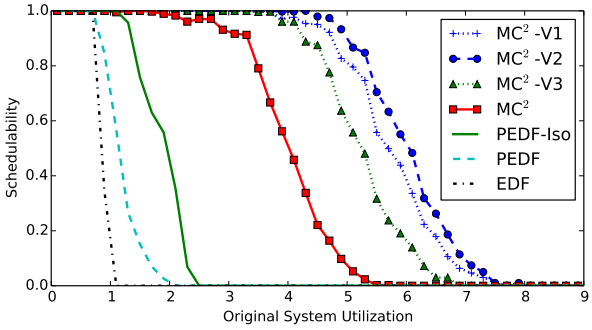
BC-Mod., Short, Heavy, Light, Const.



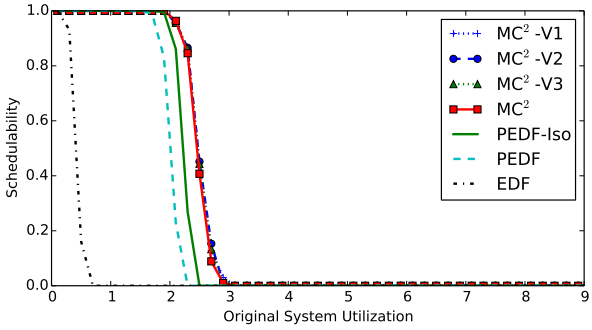
AB-Mod., Long, Light, Heavy, Const.



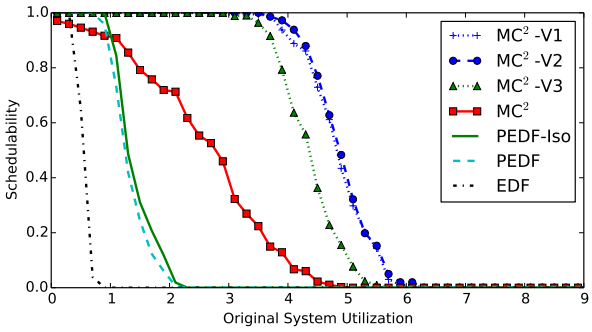
C-Heavy, Cont., Heavy, Light, Large Var.



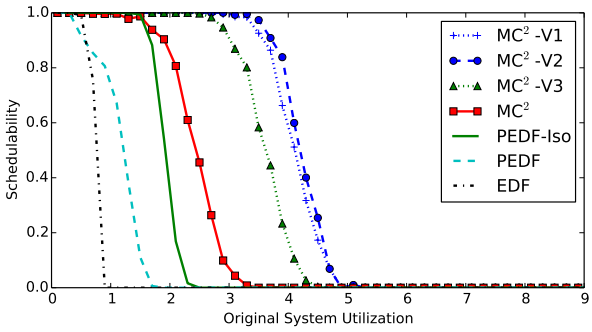
BC-Mod., Long, Heavy, Light, Large Var.



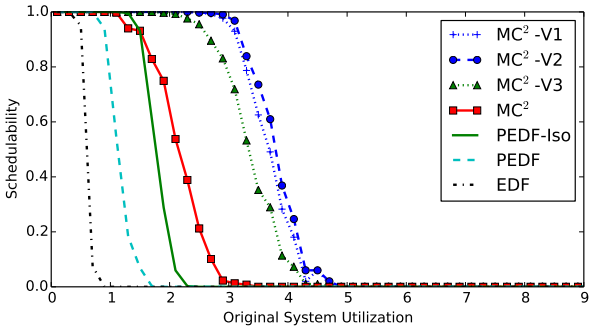
AC-Mod., Long, Light, Mod., Large Var.



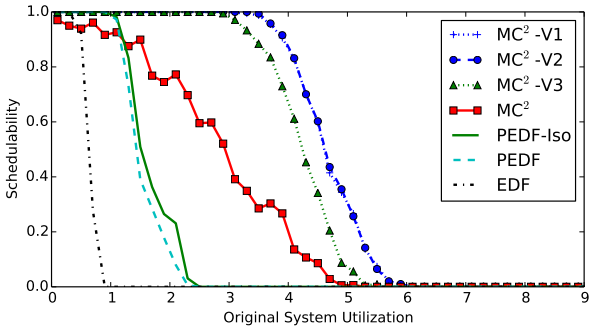
BC-Mod., Short, Heavy, Heavy, Large Var.



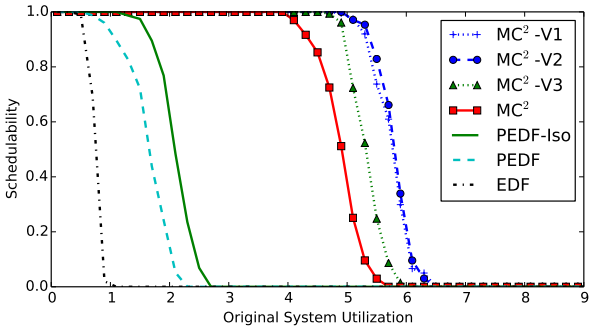
B-Heavy, Long, Heavy, Mod., Const.



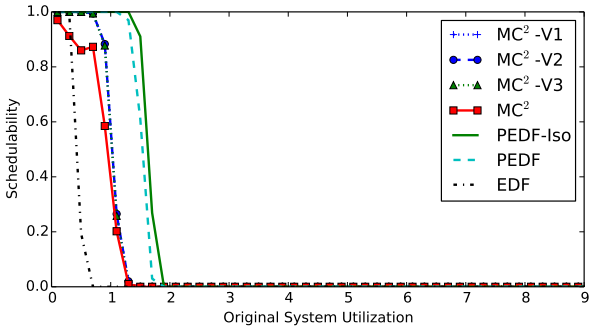
A-Heavy, Long, Heavy, Heavy, Const.



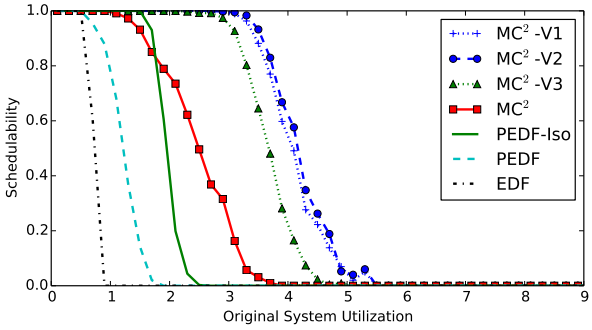
All-Mod., Short, Heavy, Mod., Large Var.



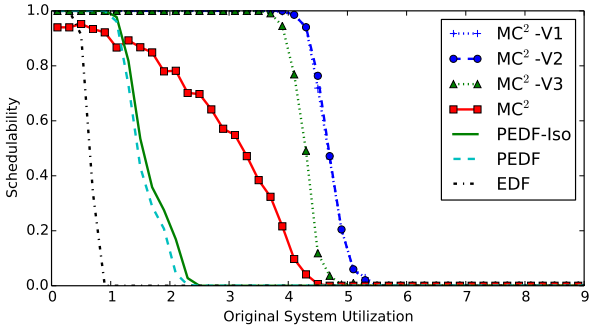
AC-Mod., Long, Mod., Light, Const.



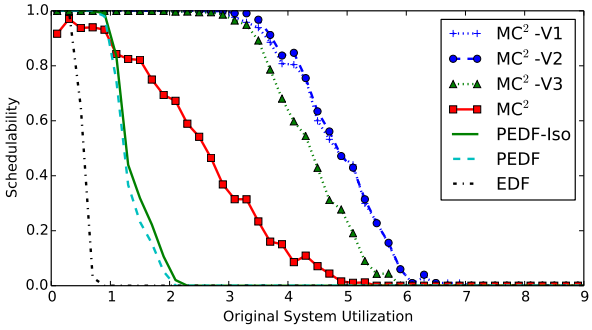
AB-Mod., Short, Light, Light, Const.



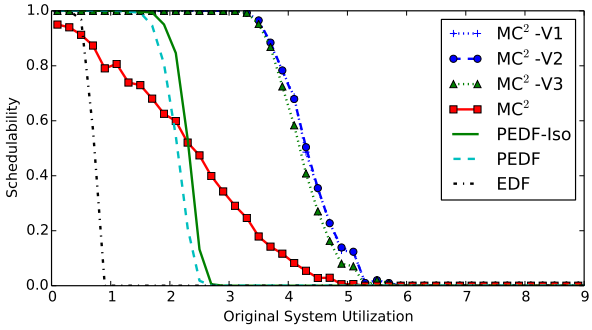
AB-Mod., Long, Heavy, Mod., Const.



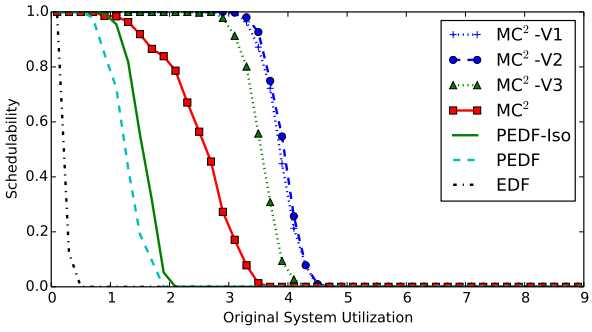
All-Mod., Short, Heavy, Mod., Const.



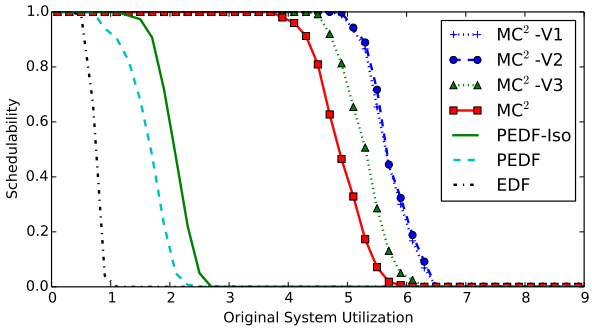
BC-Mod., Short, Heavy, Heavy, Large Var.



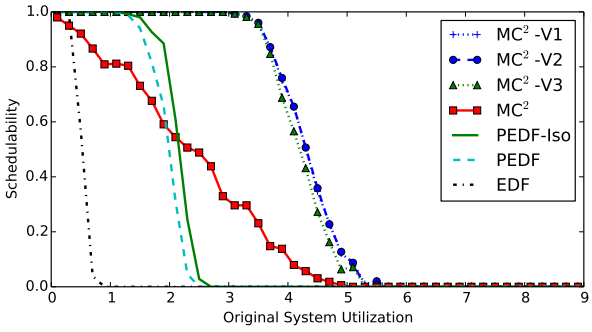
C-Heavy, Short, Mod., Light, Large Var.



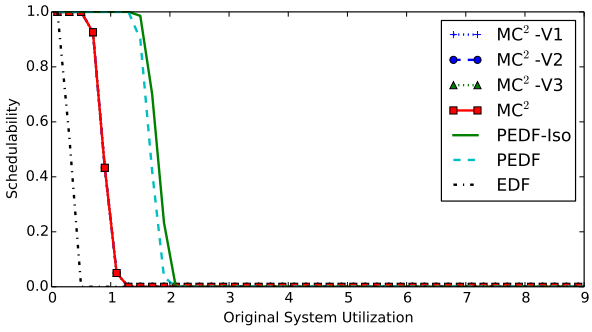
AB-Mod., Cont., Heavy, Heavy, Const.



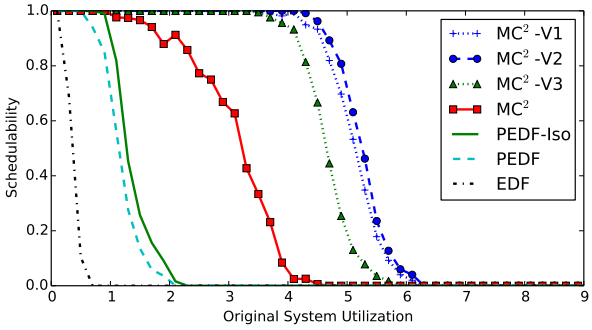
AC-Mod., Long, Mod., Light, Const.



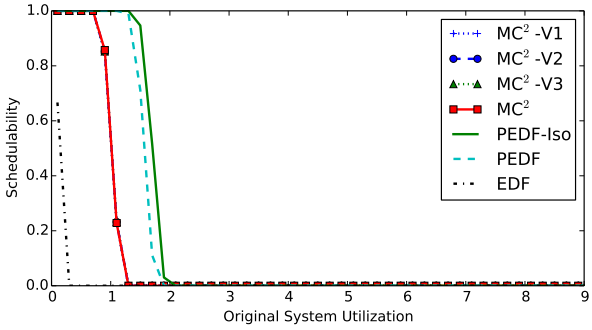
C-Heavy, Short, Mod., Mod., Const.



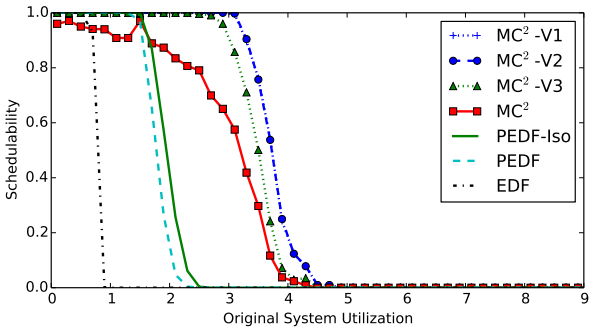
AC-Mod., Cont., Light, Light, Large Var.



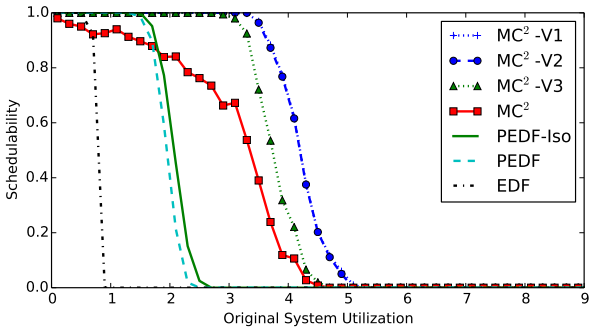
BC-Mod., Cont., Heavy, Mod., Const.



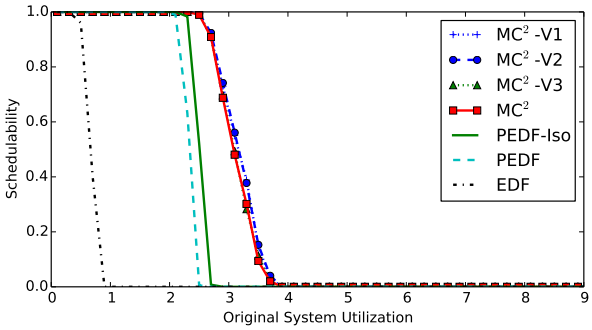
All-Mod., Cont., Light, Heavy, Large Var.



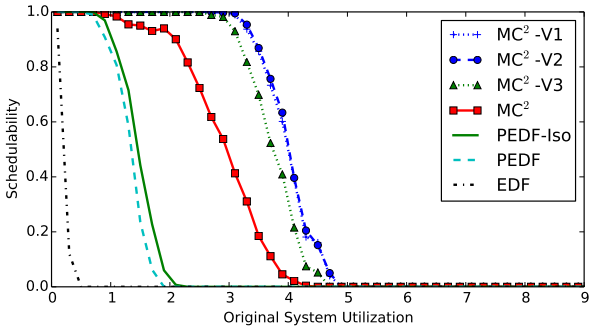
A-Heavy, Short, Heavy, Light, Large Var.



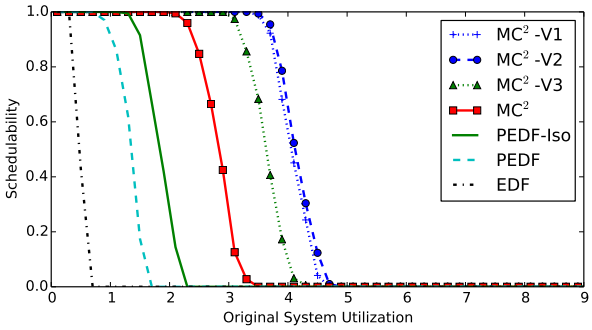
AB-Mod., Short, Heavy, Light, Large Var.



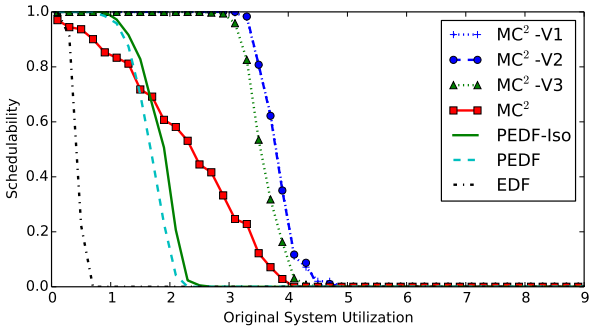
C-Heavy, Long, Light, Light, Large Var.



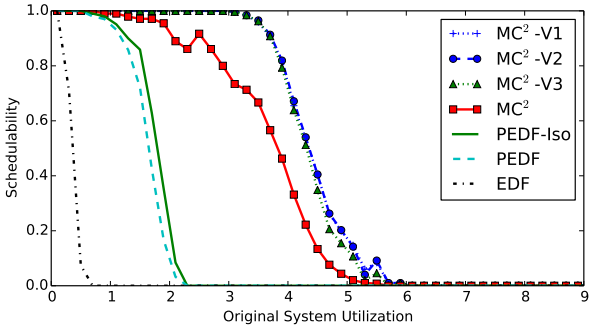
BC-Mod., Cont., Mod., Heavy, Large Var.



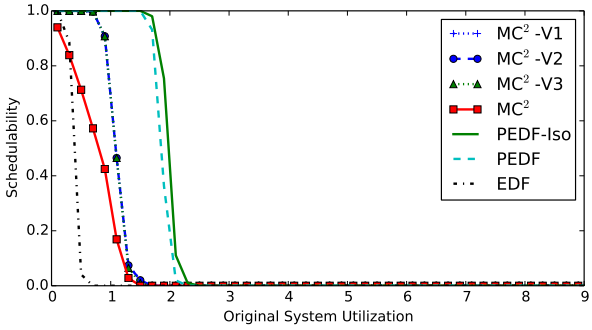
AB-Mod., Long, Mod., Heavy, Const.



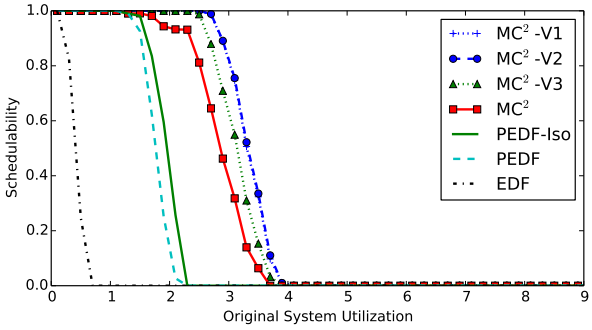
BC-Mod., Short, Mod., Heavy, Const.



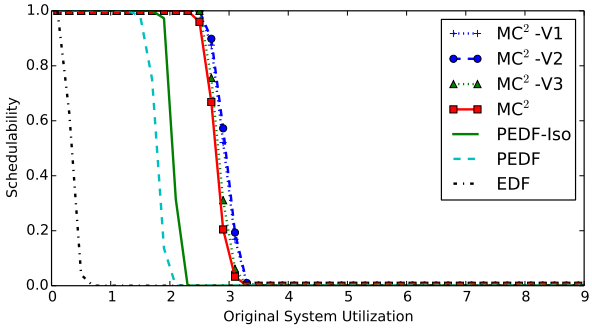
C-Heavy, Cont., Mod., Mod., Const.



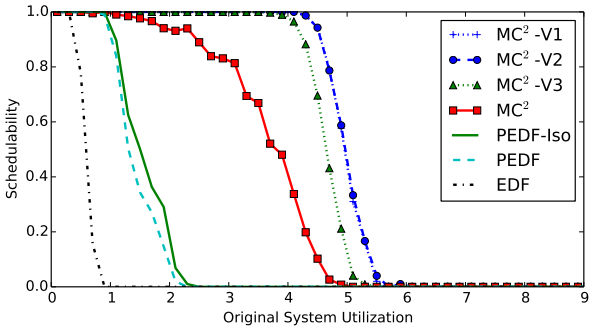
C-Heavy, Short, Light, Mod., Const.



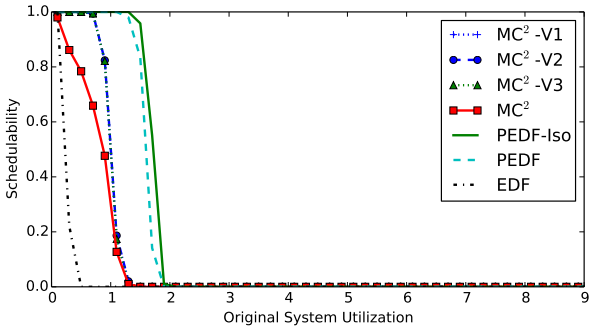
AB-Mod., Cont., Mod., Light, Large Var.



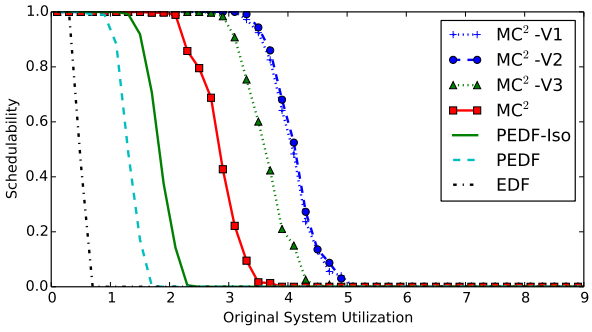
BC-Mod., Long, Light, Heavy, Const.



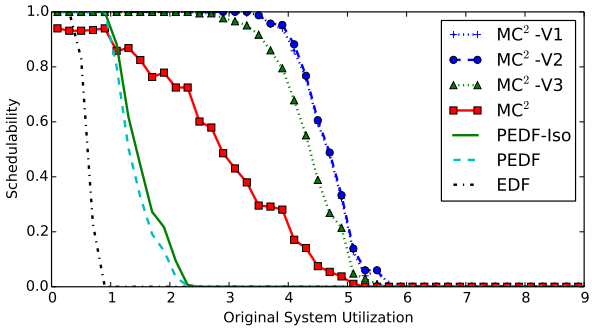
AC-Mod., Cont., Heavy, Light, Const.



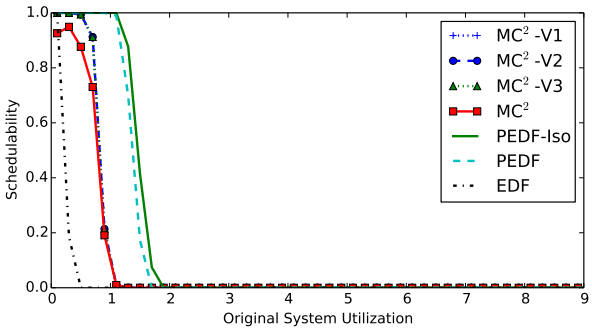
All-Mod., Short, Light, Heavy, Const.



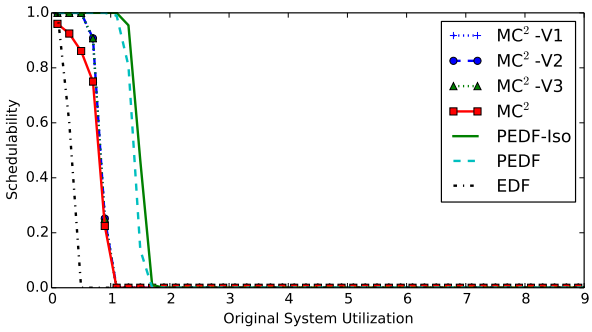
AB-Mod., Long, Mod., Heavy, Const.



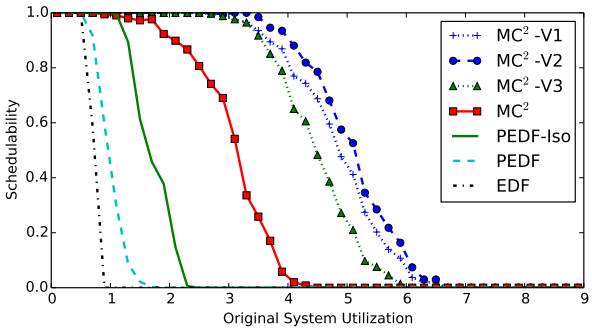
AC-Mod., Short, Heavy, Mod., Large Var.



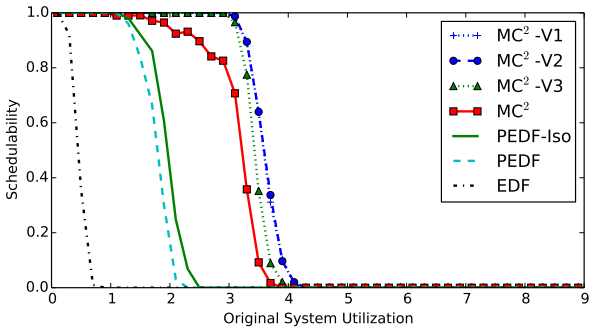
A-Heavy, Short, Light, Heavy, Large Var.



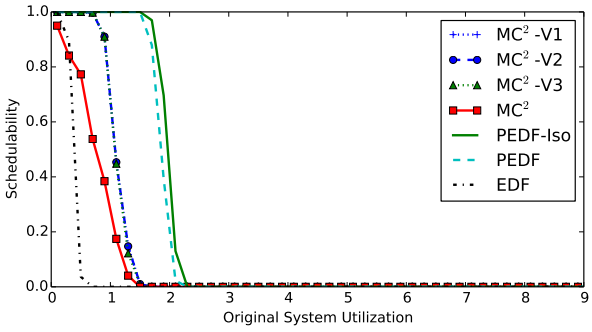
A-Heavy, Short, Light, Mod., Large Var.



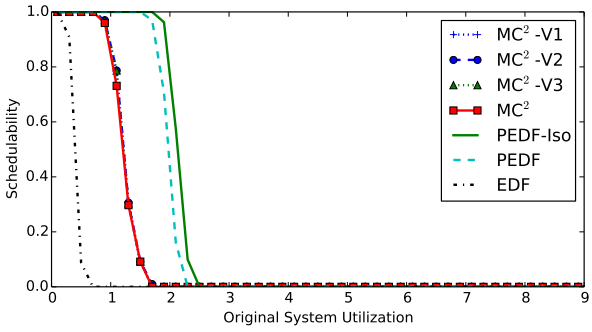
BC-Mod., Long, Heavy, Heavy, Large Var.



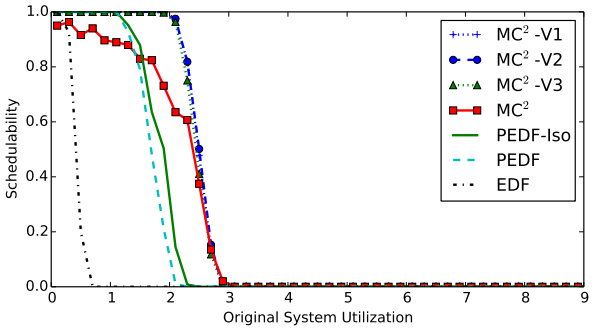
All-Mod., Cont., Mod., Light, Const.



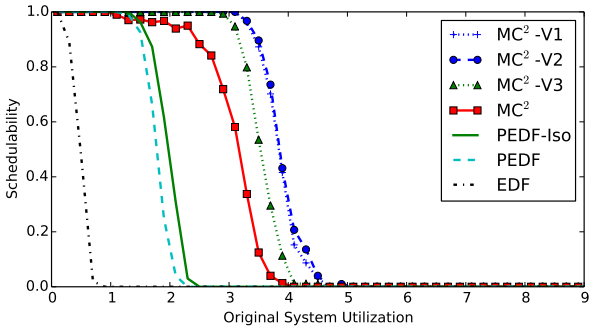
C-Heavy, Short, Light, Mod., Const.



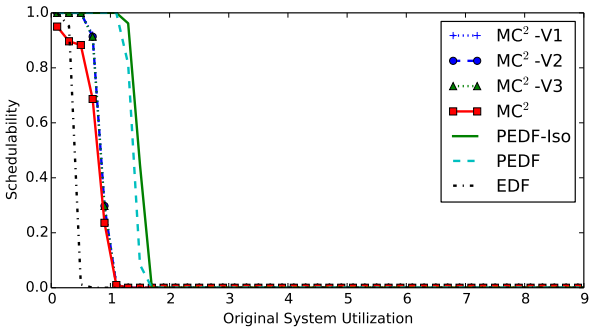
BC-Mod., Cont., Light, Light, Const.



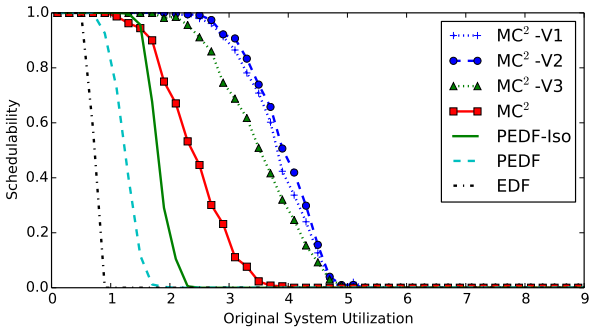
A-Heavy, Short, Mod., Heavy, Large Var.



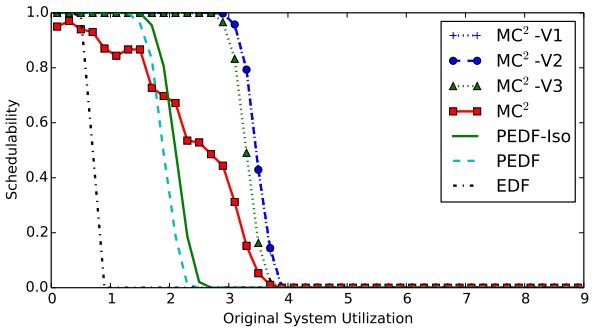
B-Heavy, Cont., Mod., Light, Large Var.



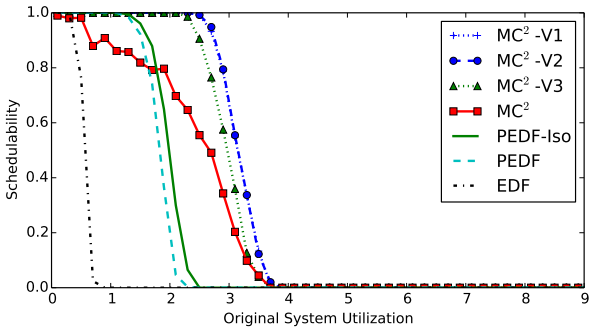
A-Heavy, Short, Light, Light, Const.



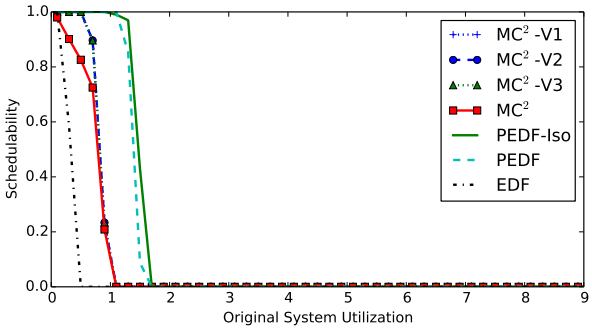
A-Heavy, Long, Heavy, Mod., Large Var.



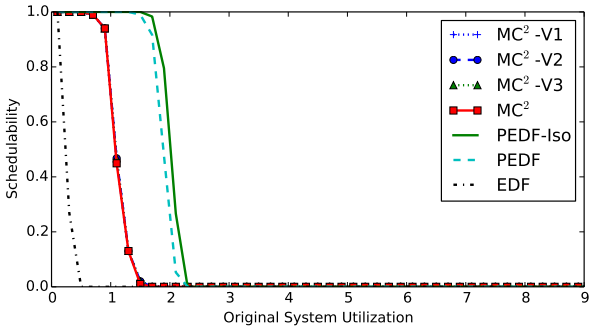
All-Mod., Short, Mod., Light, Const.



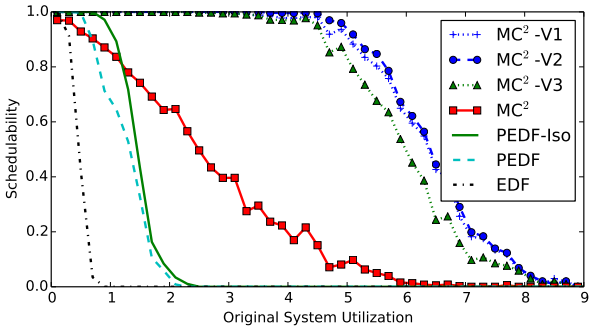
AB-Mod., Short, Mod., Mod., Large Var.



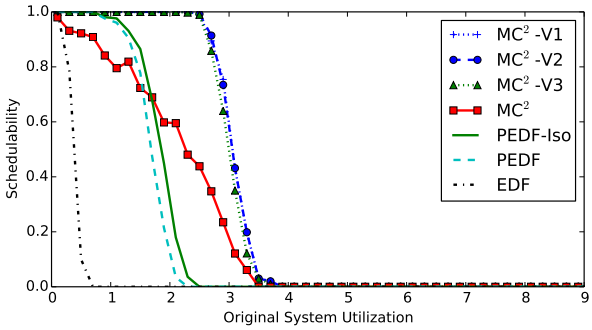
A-Heavy, Short, Light, Mod., Const.



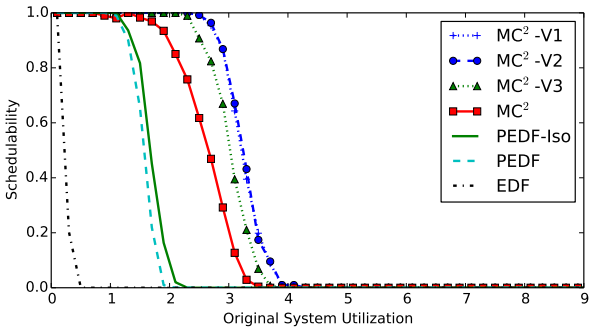
C-Heavy, Cont., Light, Mod., Const.



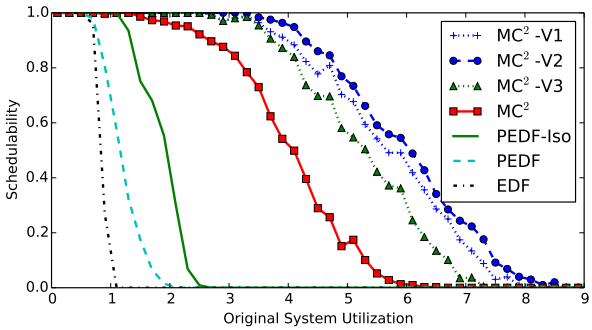
C-Heavy, Short, Heavy, Heavy, Large Var.



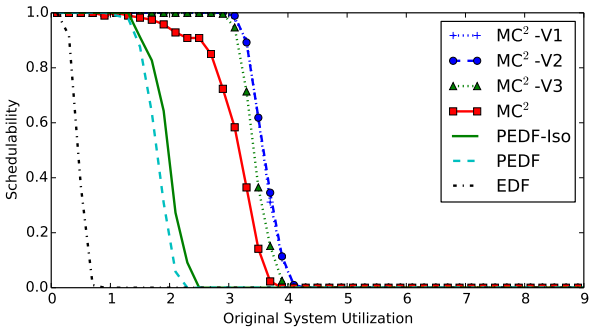
AC-Mod., Short, Mod., Heavy, Large Var.



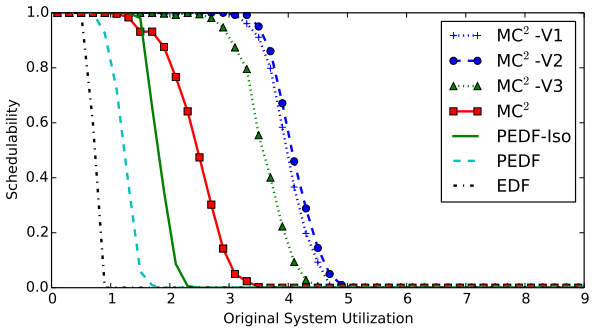
AB-Mod., Cont., Mod., Mod., Large Var.



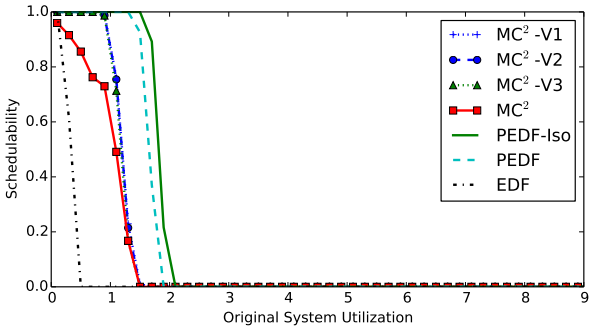
BC-Mod., Long, Heavy, Light, Large Var.



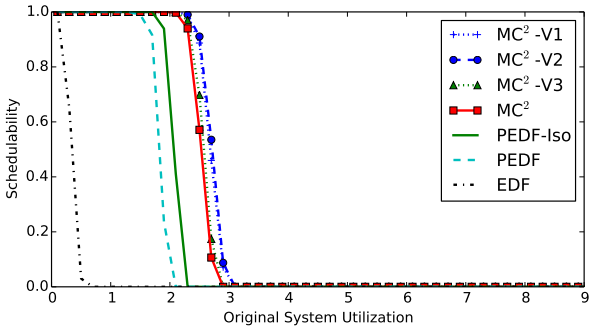
All-Mod., Cont., Mod., Light, Large Var.



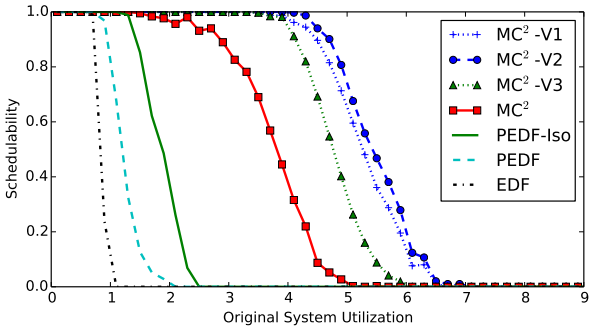
A-Heavy, Long, Heavy, Mod., Const.



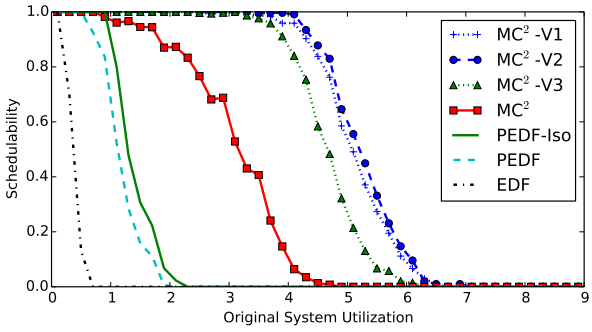
B-Heavy, Short, Light, Heavy, Const.



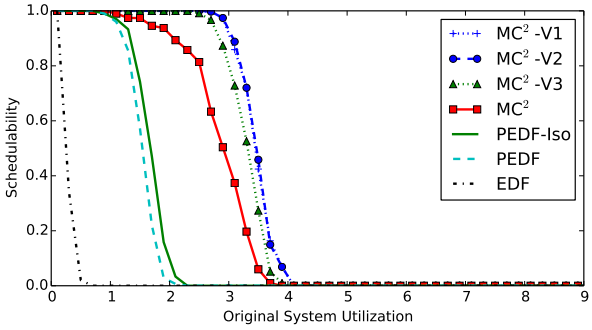
AB-Mod., Long, Light, Heavy, Const.



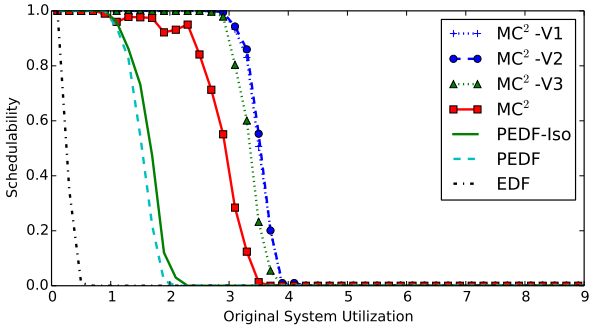
All-Mod., Long, Heavy, Light, Const.



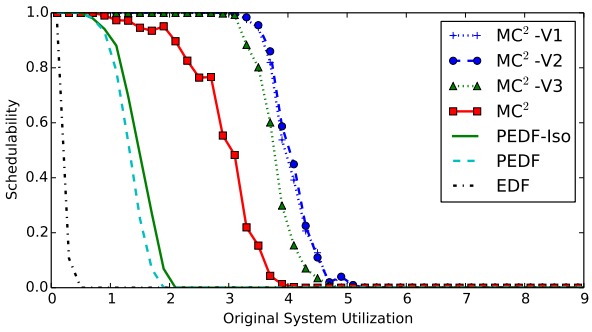
BC-Mod., Cont., Heavy, Mod., Const.



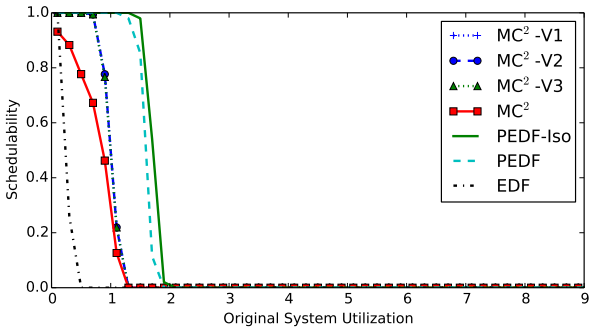
All-Mod., Cont., Mod., Mod., Large Var.



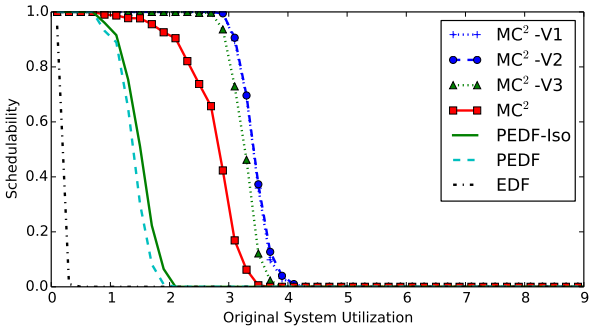
All-Mod., Cont., Mod., Mod., Large Var.



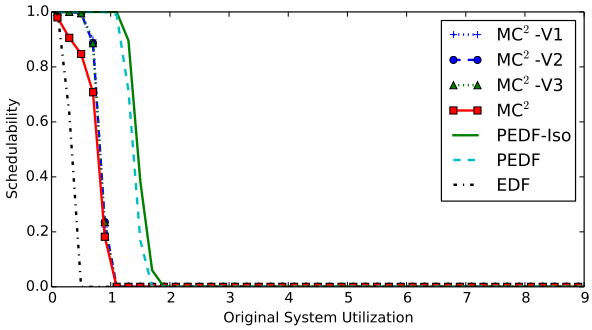
BC-Mod., Cont., Mod., Heavy, Large Var.



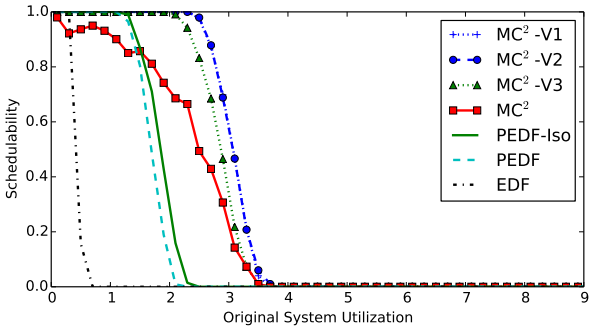
All-Mod., Short, Light, Heavy, Const.



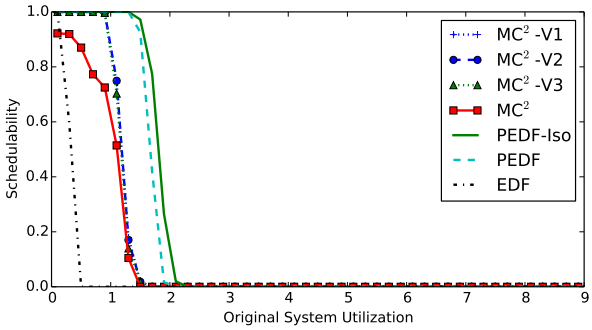
All-Mod., Cont., Mod., Heavy, Const.



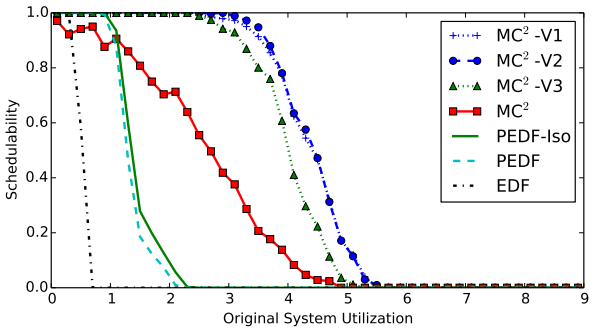
A-Heavy, Short, Light, Mod., Large Var.



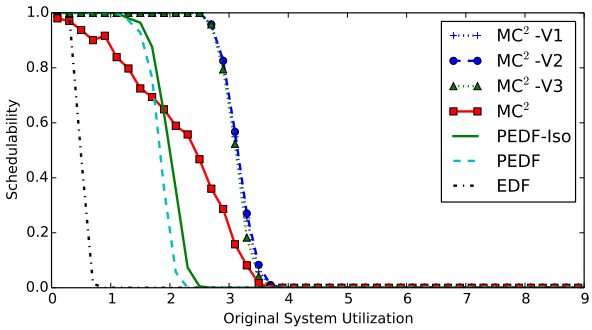
AB-Mod., Short, Mod., Heavy, Large Var.



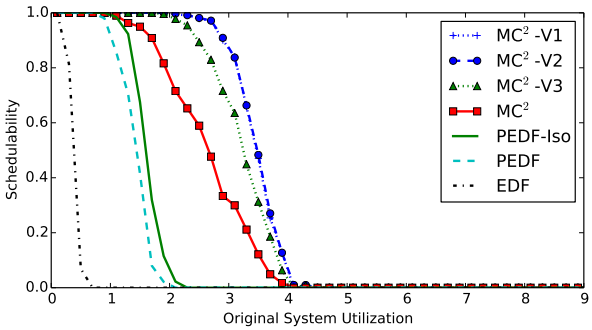
B-Heavy, Short, Light, Heavy, Large Var.



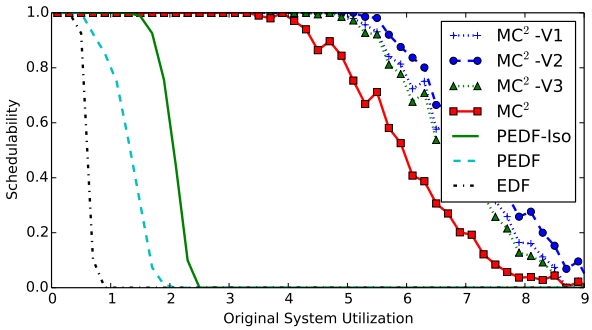
All-Mod., Short, Heavy, Heavy, Large Var.



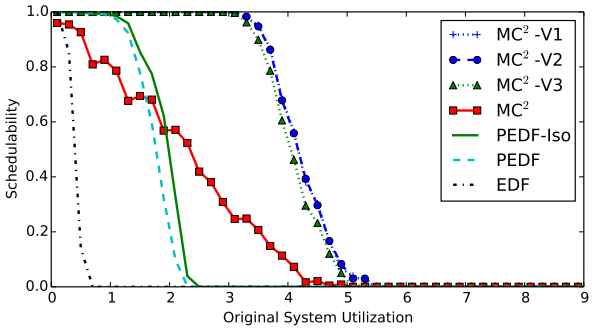
AC-Mod., Short, Mod., Mod., Large Var.



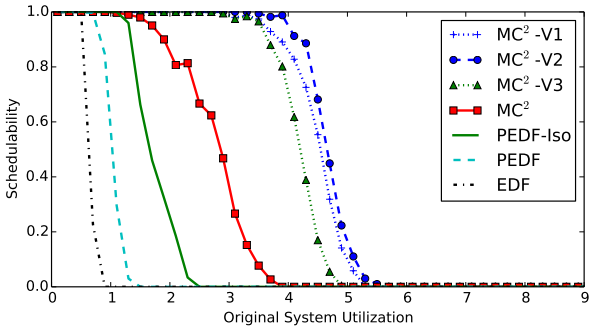
A-Heavy, Cont., Heavy, Mod., Large Var.



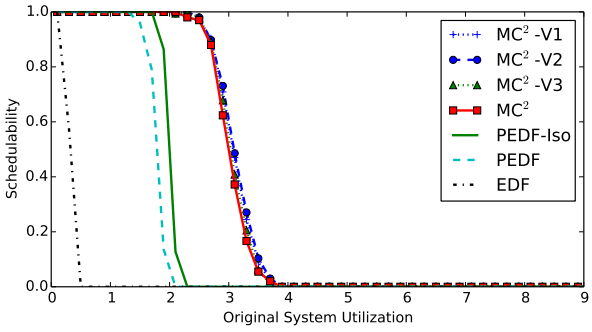
C-Heavy, Long, Mod., Mod., Large Var.



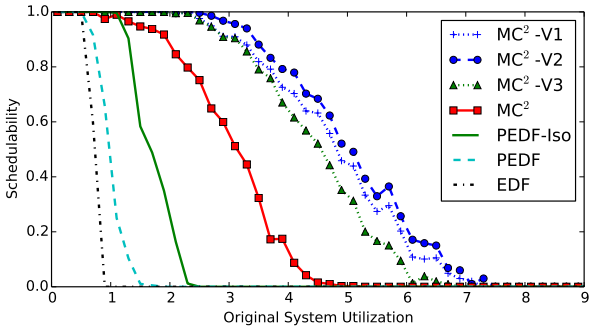
C-Heavy, Short, Mod., Heavy, Large Var.



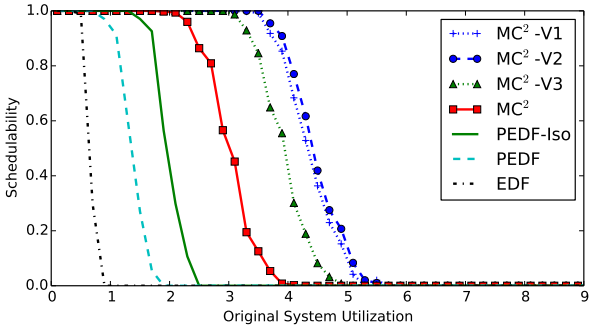
All-Mod., Long, Heavy, Heavy, Const.



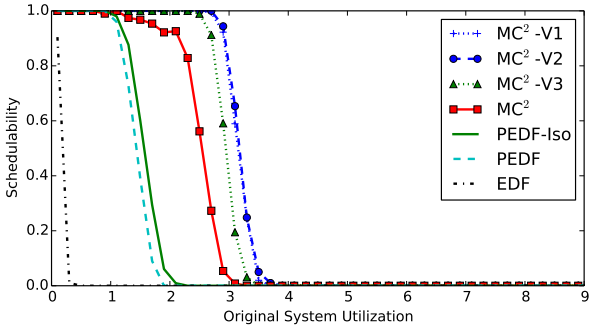
C-Heavy, Long, Light, Heavy, Const.



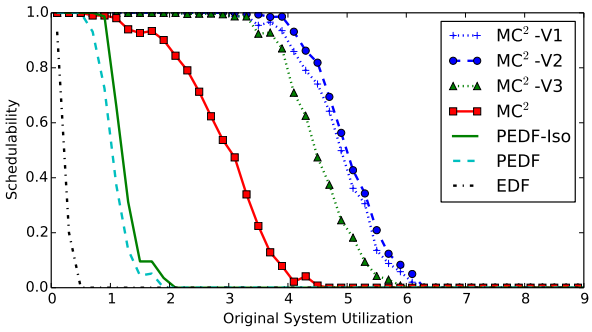
BC-Mod., Long, Heavy, Heavy, Large Var.



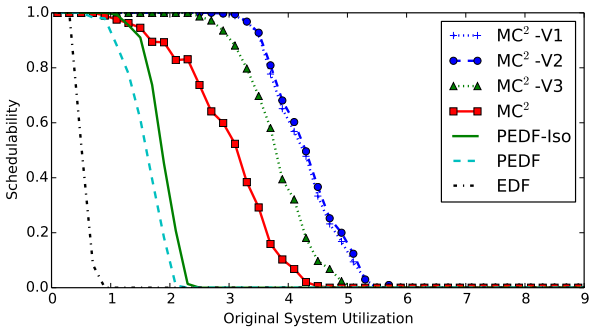
B-Heavy, Long, Mod., Mod., Large Var.



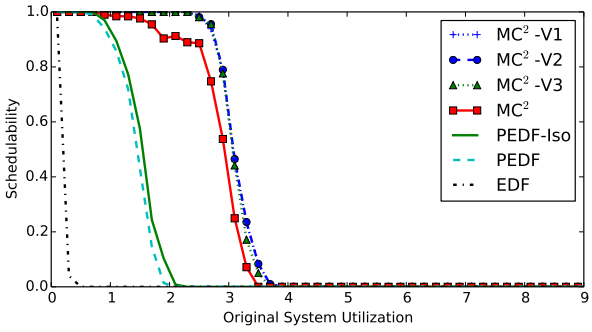
AB-Mod., Cont., Mod., Heavy, Const.



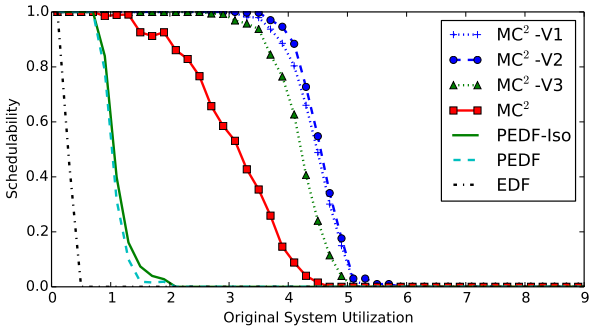
BC-Mod., Cont., Heavy, Heavy, Const.



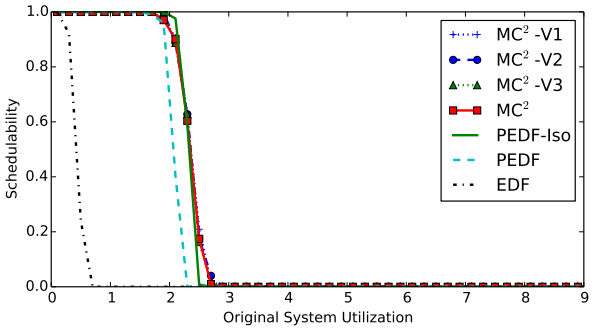
AB-Mod., Cont., Heavy, Light, Large Var.



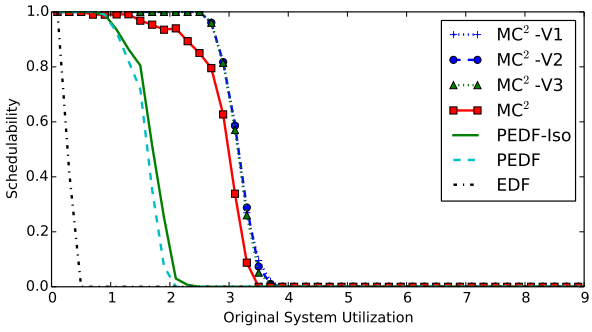
AC-Mod., Cont., Mod., Heavy, Const.



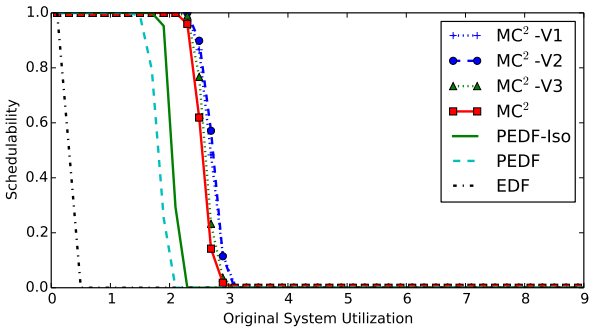
AC-Mod., Cont., Heavy, Heavy, Const.



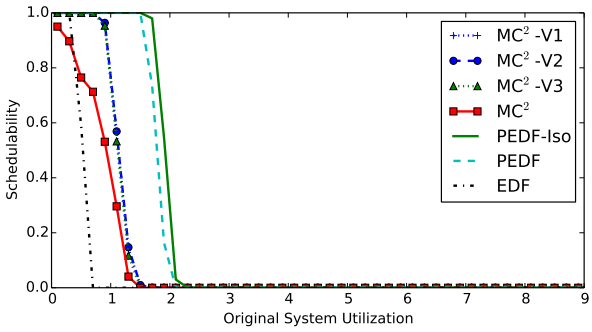
A-Heavy, Long, Light, Mod., Const.



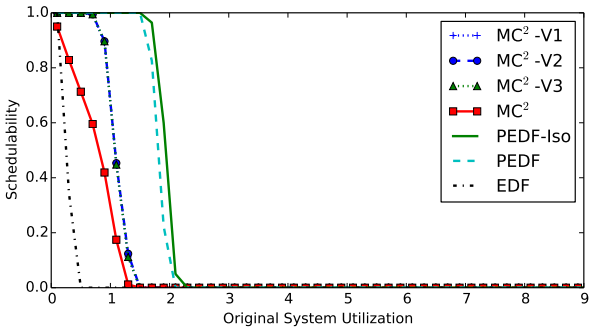
AC-Mod., Cont., Mod., Mod., Const.



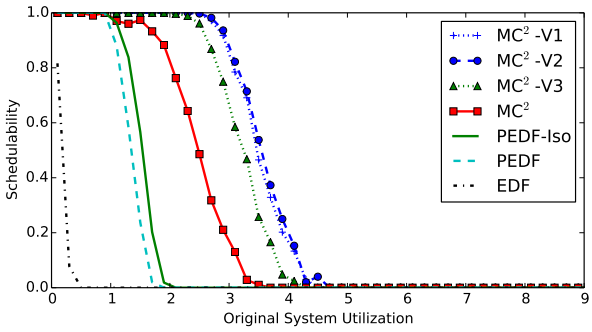
All-Mod., Long, Light, Heavy, Const.



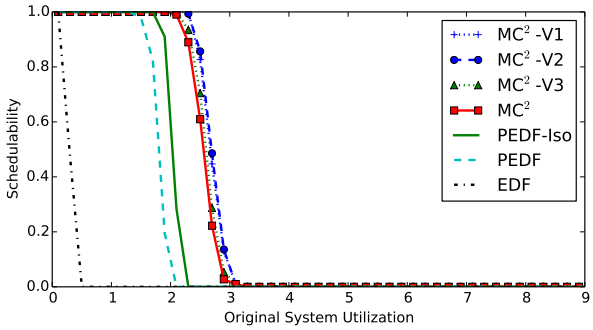
BC-Mod., Short, Light, Light, Const.



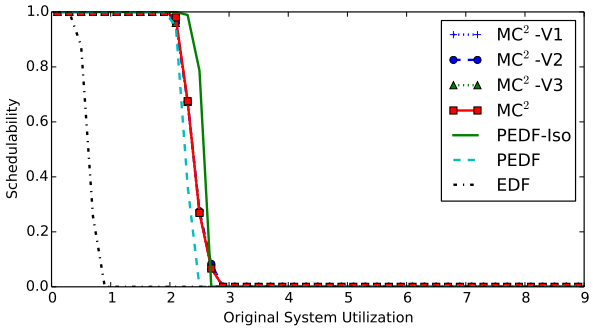
C-Heavy, Short, Light, Heavy, Const.



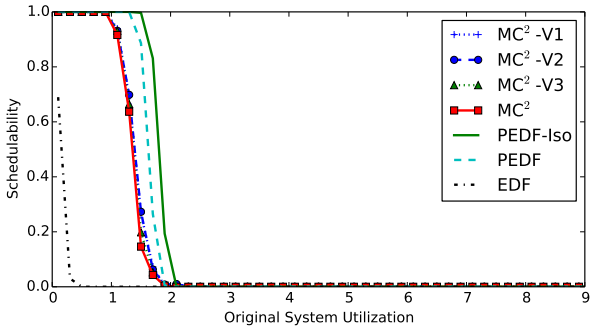
B-Heavy, Cont., Mod., Heavy, Large Var.



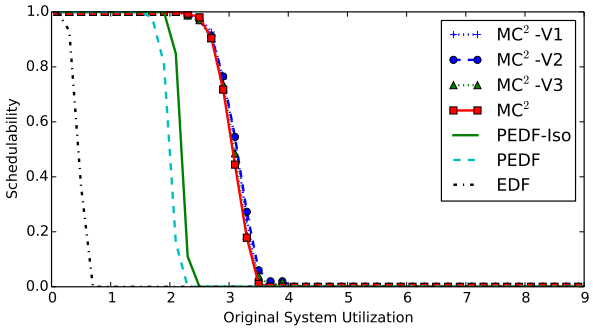
All-Mod., Long, Light, Heavy, Large Var.



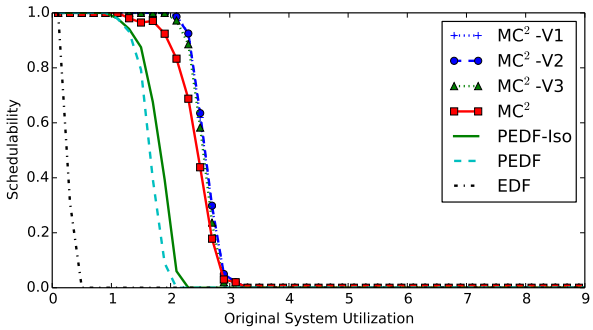
A-Heavy, Long, Light, Light, Const.



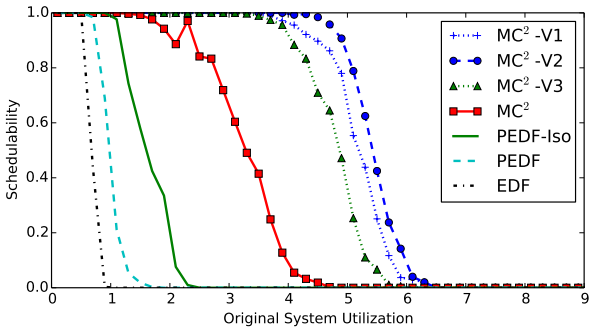
B-Heavy, Cont., Light, Heavy, Const.



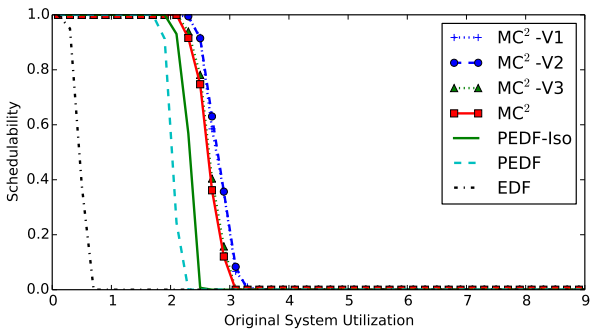
C-Heavy, Long, Light, Mod., Const.



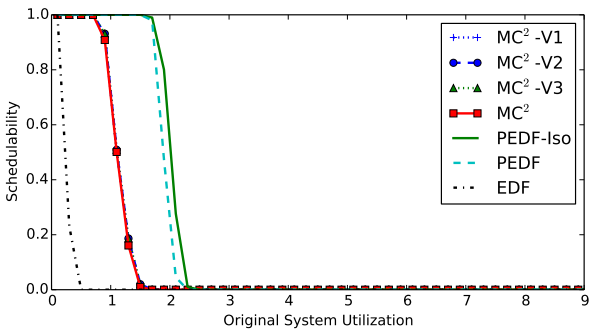
A-Heavy, Cont., Mod., Mod., Large Var.



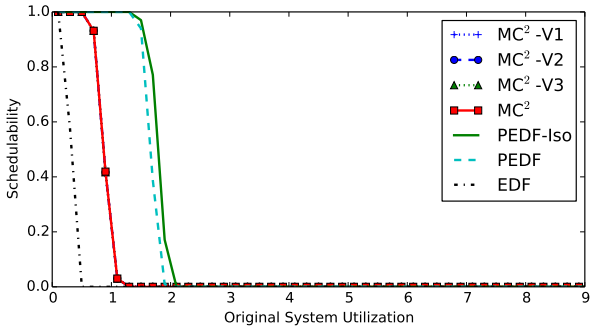
AC-Mod., Long, Heavy, Mod., Const.



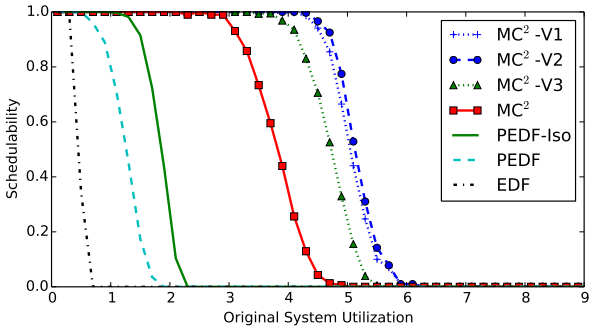
AB-Mod., Long, Light, Mod., Large Var.



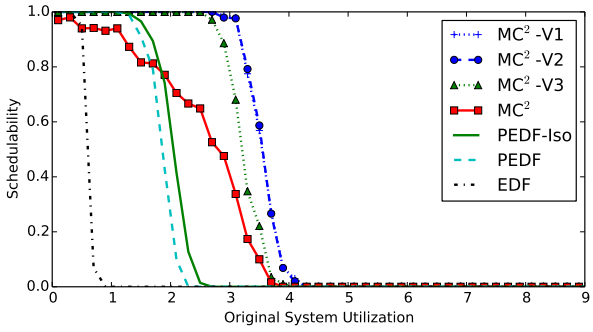
C-Heavy, Cont., Light, Mod., Const.



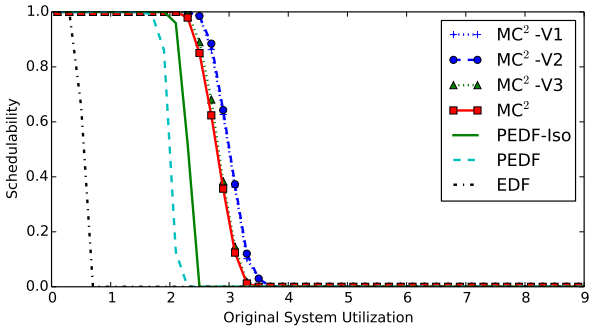
AC-Mod., Cont., Light, Light, Const.



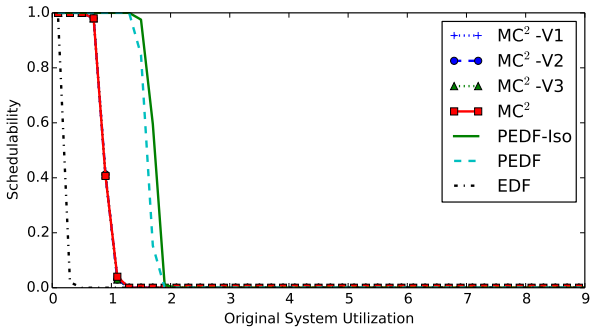
AC-Mod., Long, Mod., Heavy, Const.



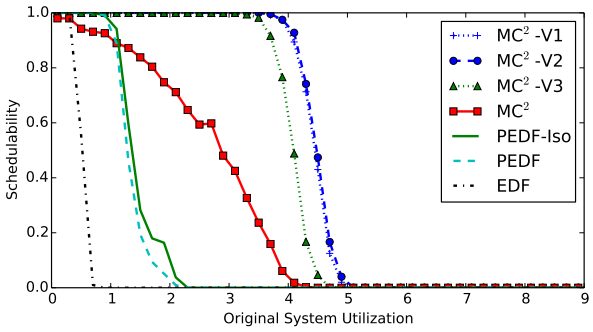
B-Heavy, Short, Mod., Mod., Const.



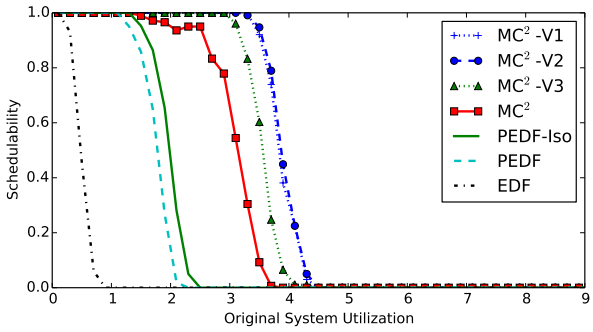
B-Heavy, Long, Light, Mod., Large Var.



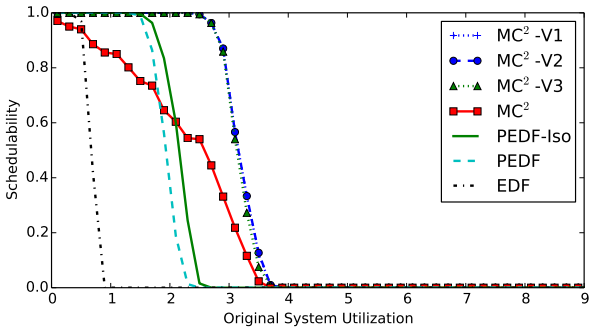
AC-Mod., Cont., Light, Mod., Const.



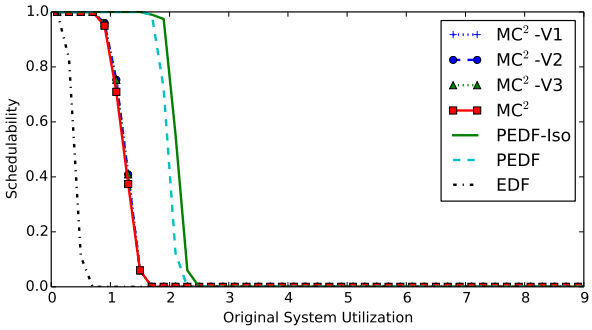
All-Mod., Short, Heavy, Heavy, Const.



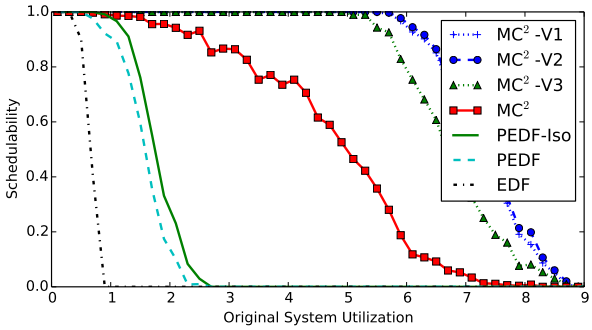
B-Heavy, Cont., Mod., Light, Const.



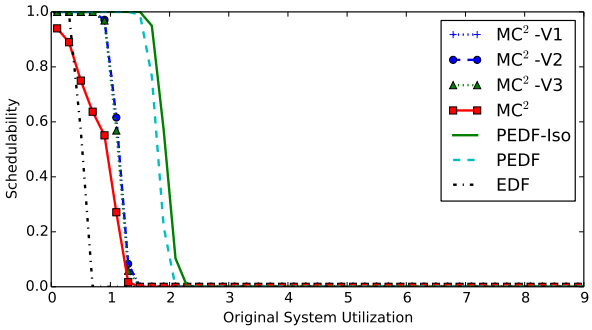
AC-Mod., Short, Mod., Light, Const.



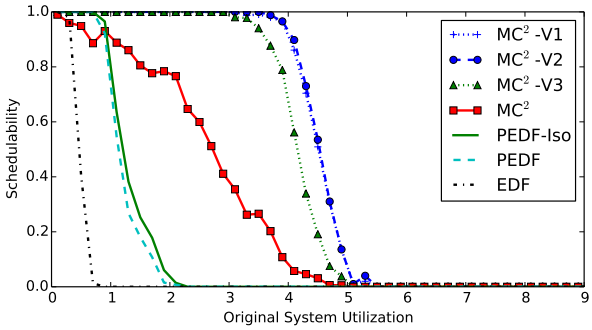
BC-Mod., Cont., Light, Light, Const.



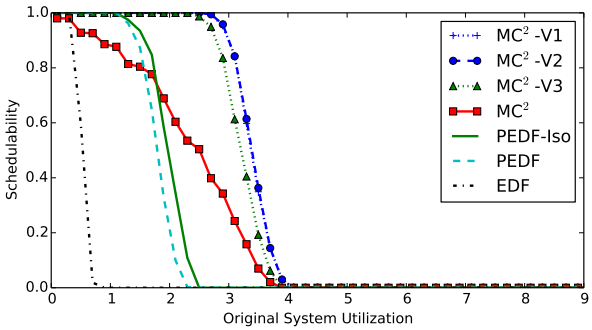
C-Heavy, Cont., Heavy, Light, Const.



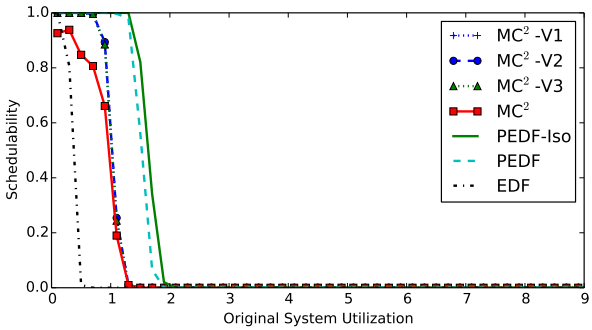
BC-Mod., Short, Light, Light, Large Var.



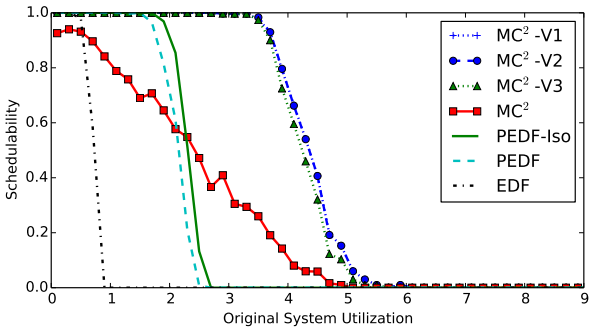
AC-Mod., Short, Heavy, Heavy, Large Var.



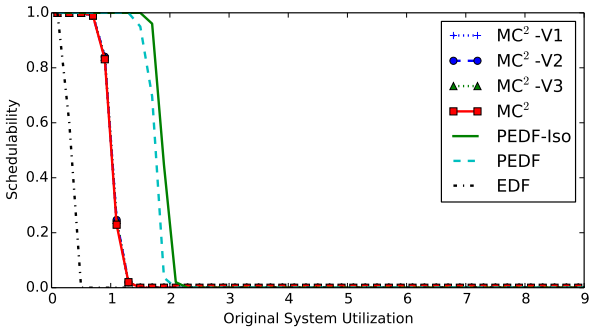
All-Mod., Short, Mod., Mod., Large Var.



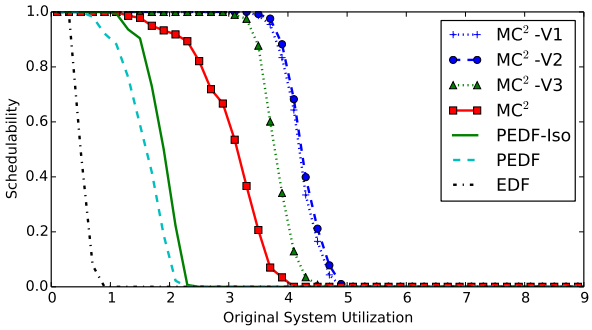
AB-Mod., Short, Light, Mod., Large Var.



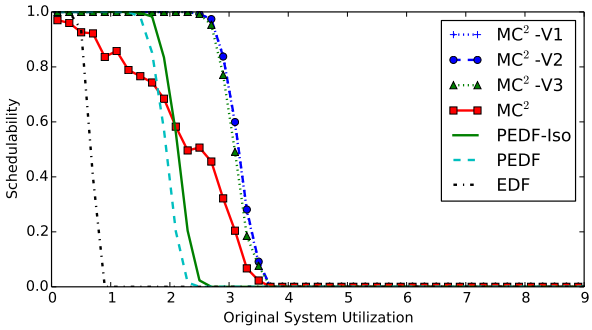
C-Heavy, Short, Mod., Light, Const.



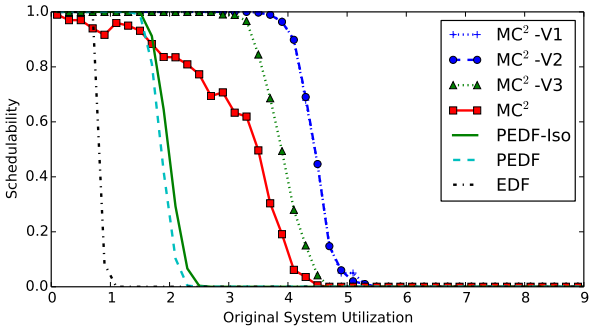
All-Mod., Cont., Light, Light, Const.



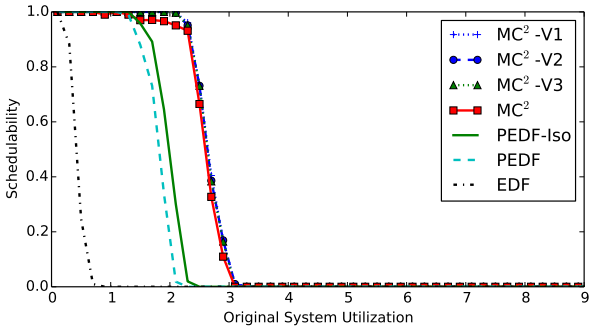
AB-Mod., Cont., Heavy, Light, Const.



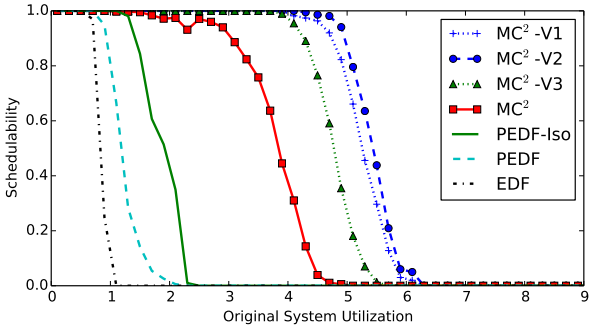
AC-Mod., Short, Mod., Light, Large Var.



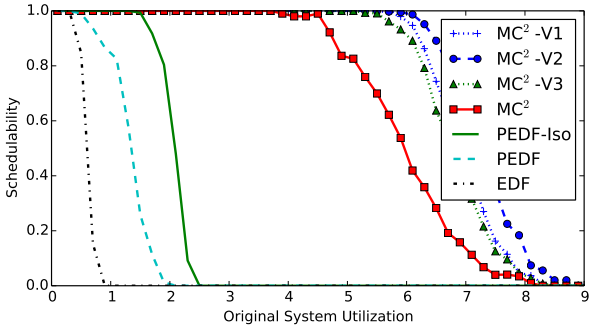
B-Heavy, Short, Heavy, Light, Const.



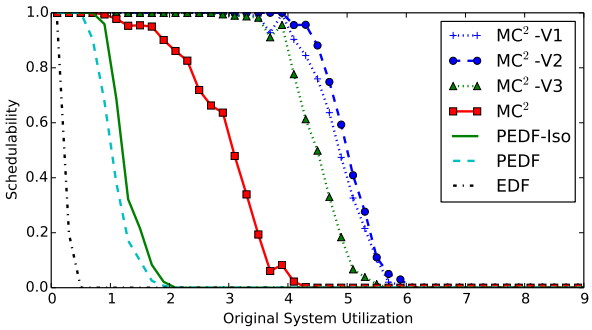
A-Heavy, Cont., Mod., Light, Const.



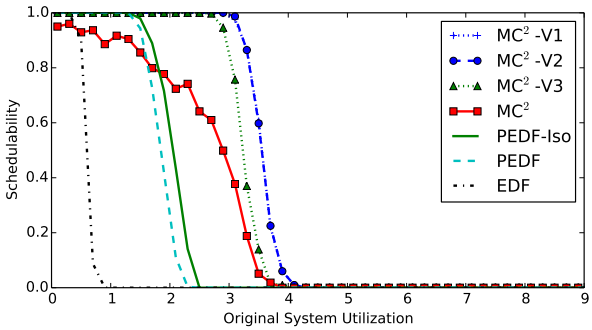
All-Mod., Long, Heavy, Light, Const.



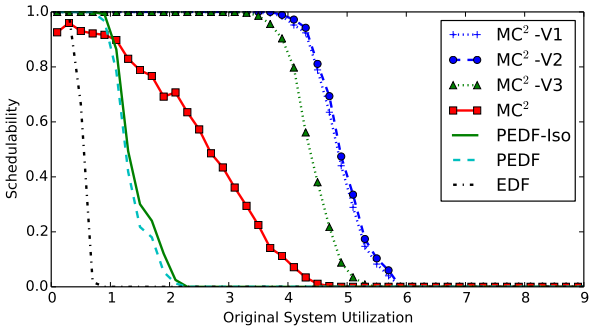
C-Heavy, Long, Mod., Mod., Const.



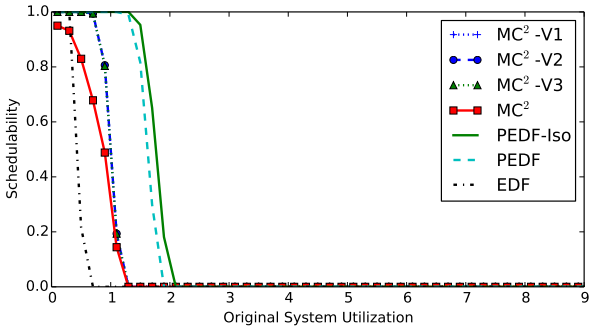
BC-Mod., Cont., Heavy, Heavy, Const.



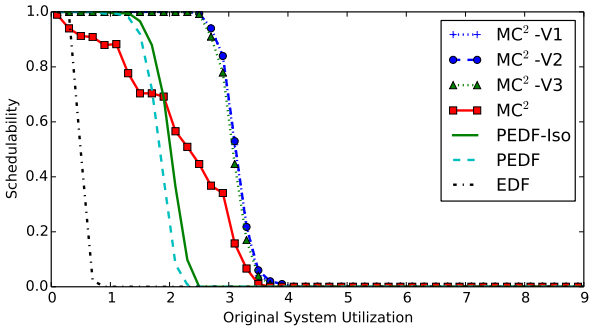
B-Heavy, Short, Mod., Mod., Const.



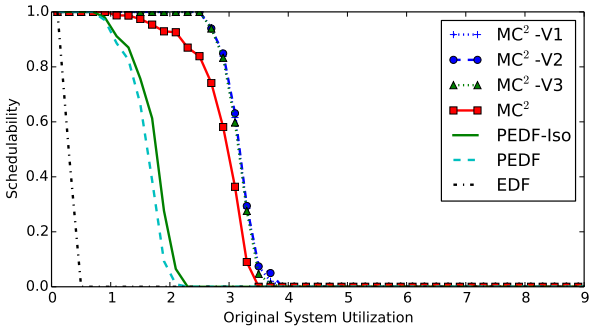
BC-Mod., Short, Heavy, Heavy, Const.



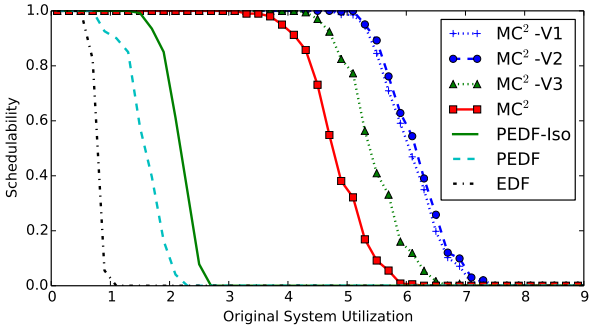
All-Mod., Short, Light, Light, Large Var.



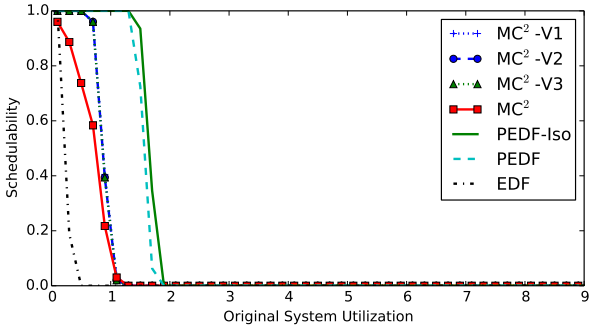
AC-Mod., Short, Mod., Mod., Large Var.



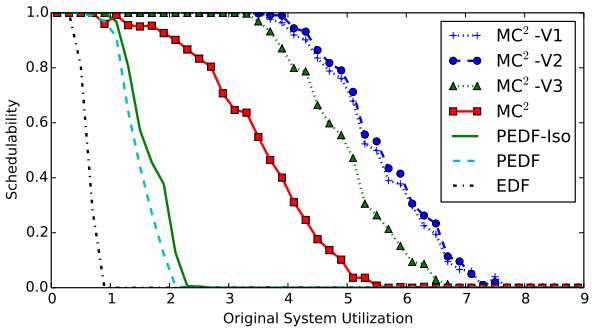
AC-Mod., Cont., Mod., Mod., Large Var.



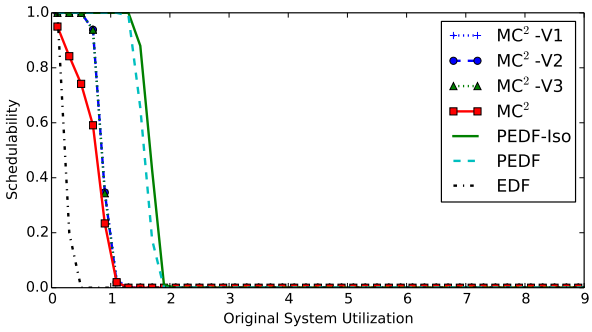
BC-Mod., Long, Mod., Light, Large Var.



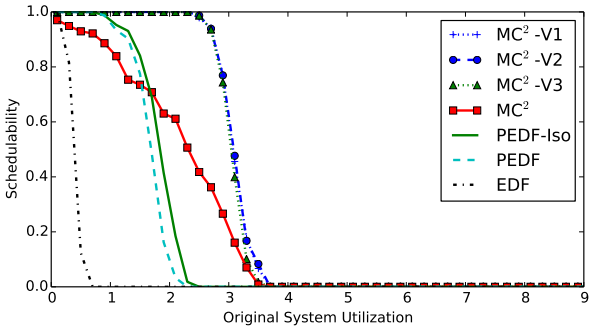
AC-Mod., Short, Light, Heavy, Large Var.



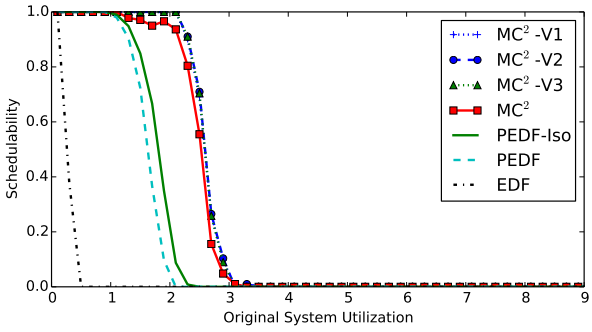
BC-Mod., Cont., Heavy, Light, Large Var.



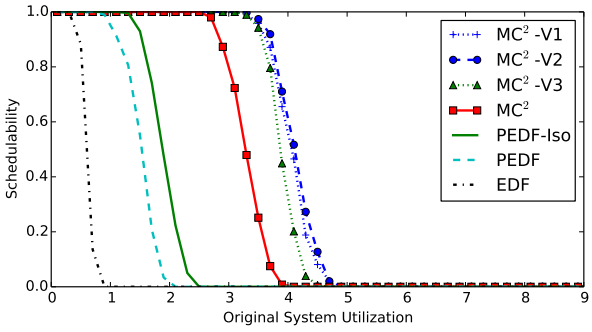
AC-Mod., Short, Light, Heavy, Large Var.



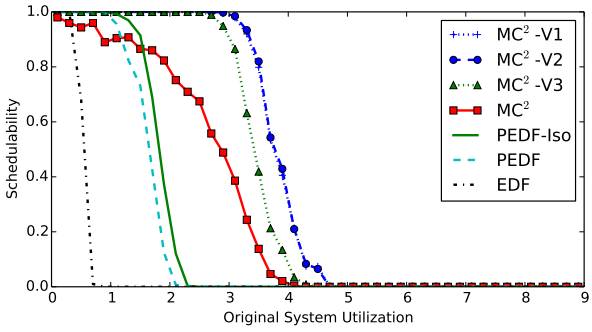
AC-Mod., Short, Mod., Heavy, Const.



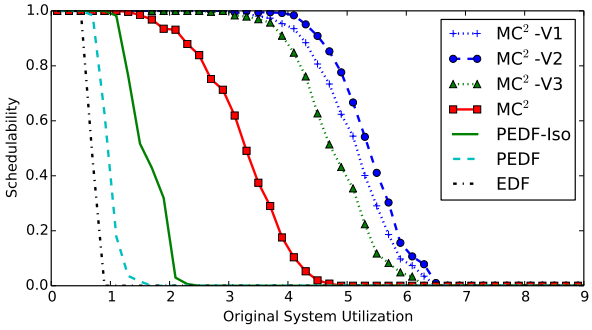
A-Heavy, Cont., Mod., Mod., Const.



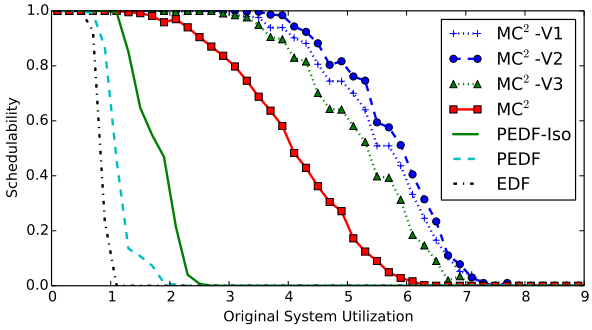
A-Heavy, Long, Mod., Mod., Const.



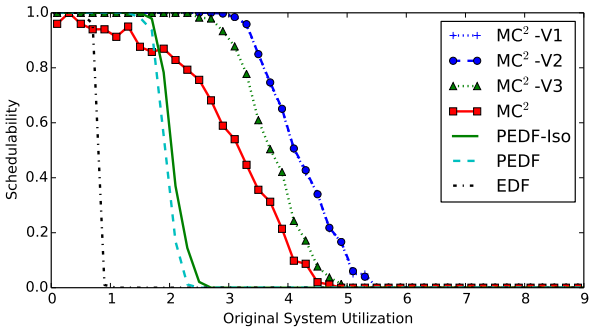
AB-Mod., Short, Heavy, Heavy, Const.



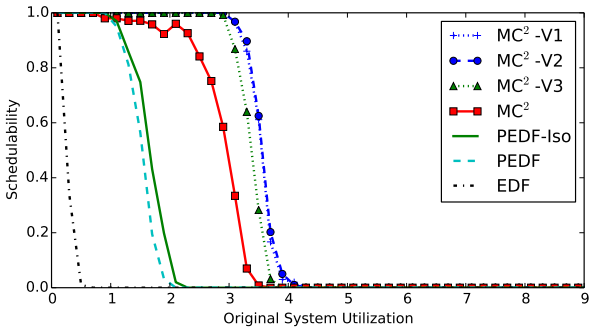
AC-Mod., Long, Heavy, Mod., Const.



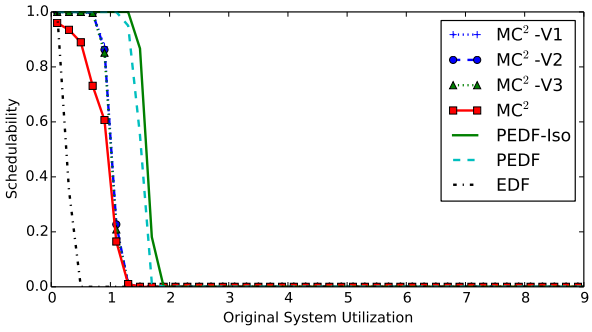
AC-Mod., Long, Heavy, Light, Large Var.



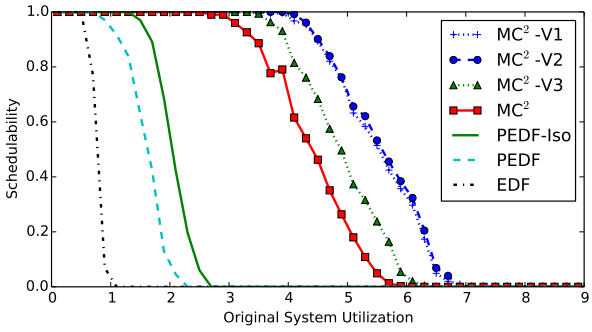
AB-Mod., Short, Heavy, Light, Large Var.



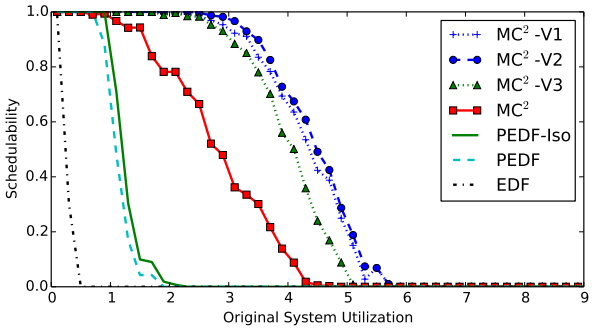
All-Mod., Cont., Mod., Mod., Const.



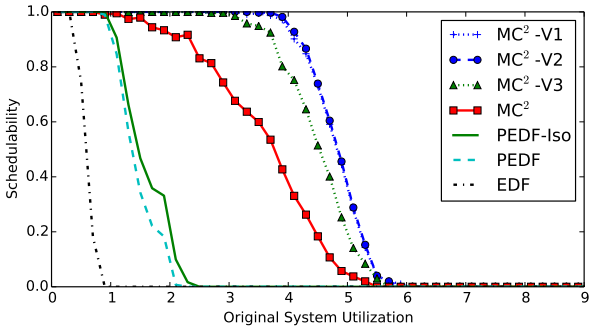
AB-Mod., Short, Light, Heavy, Const.



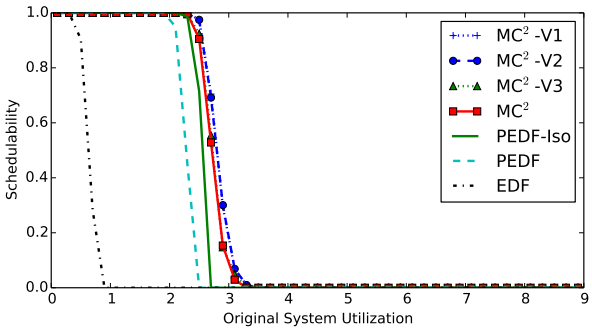
All-Mod., Long, Mod., Light, Large Var.



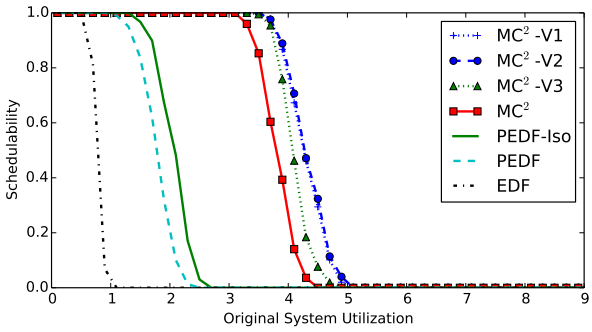
All-Mod., Cont., Heavy, Heavy, Large Var.



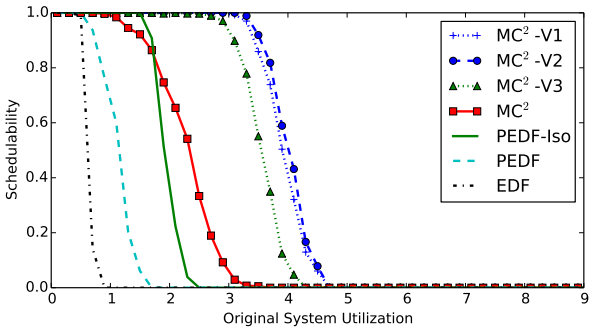
AC-Mod., Cont., Heavy, Light, Large Var.



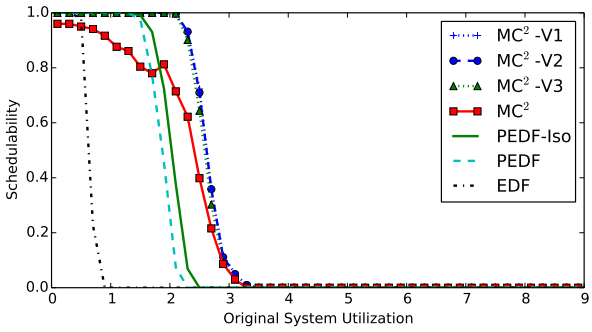
All-Mod., Long, Light, Light, Large Var.



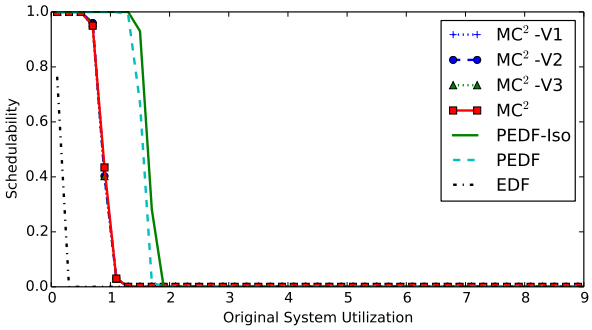
A-Heavy, Long, Mod., Light, Const.



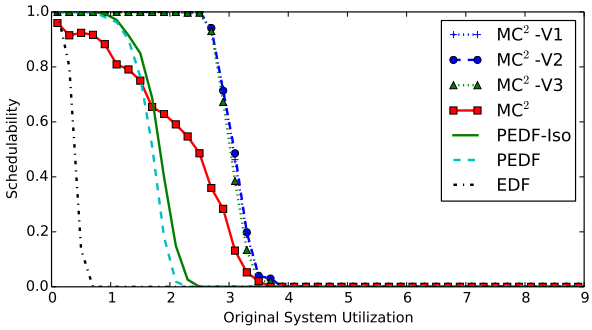
AB-Mod., Long, Heavy, Heavy, Const.



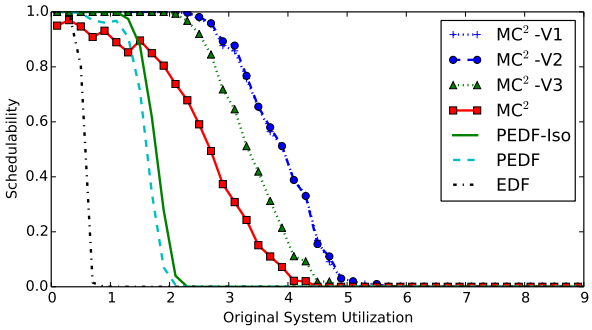
A-Heavy, Short, Mod., Light, Large Var.



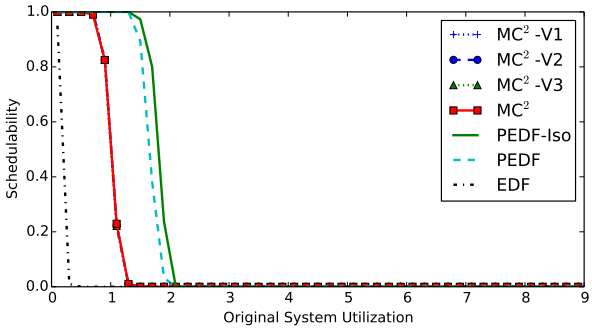
AC-Mod., Cont., Light, Heavy, Const.



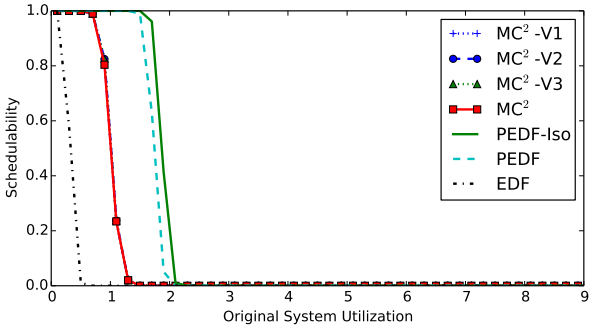
AC-Mod., Short, Mod., Heavy, Const.



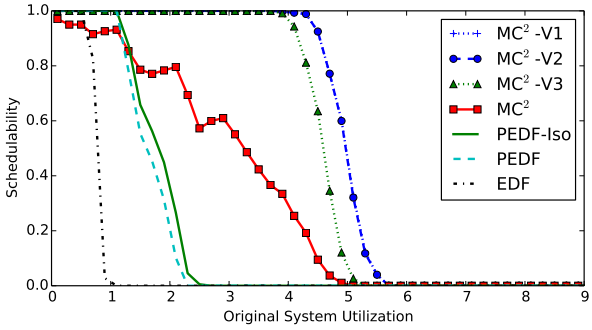
B-Heavy, Short, Heavy, Heavy, Large Var.



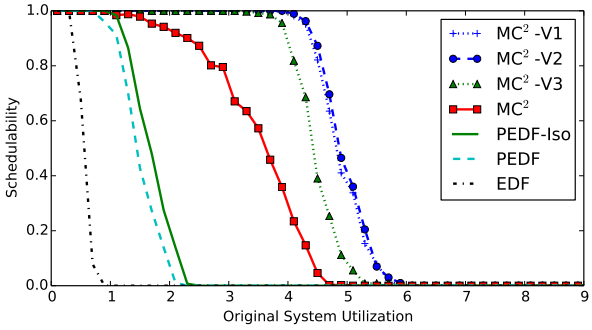
All-Mod., Cont., Light, Mod., Large Var.



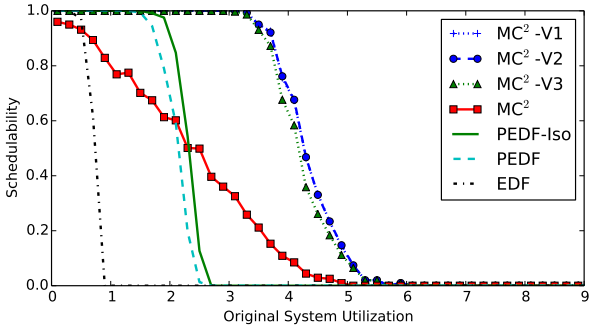
All-Mod., Cont., Light, Light, Const.



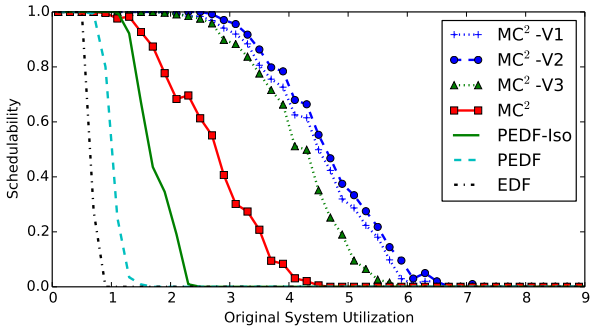
AC-Mod., Short, Heavy, Light, Const.



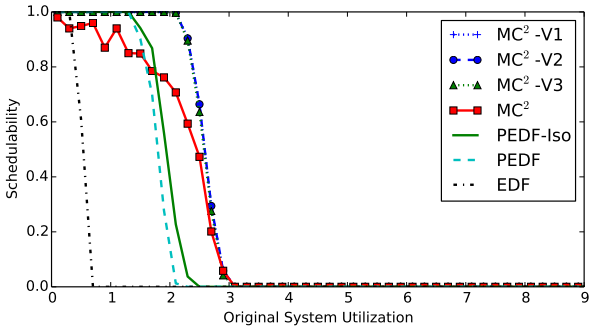
All-Mod., Cont., Heavy, Light, Const.



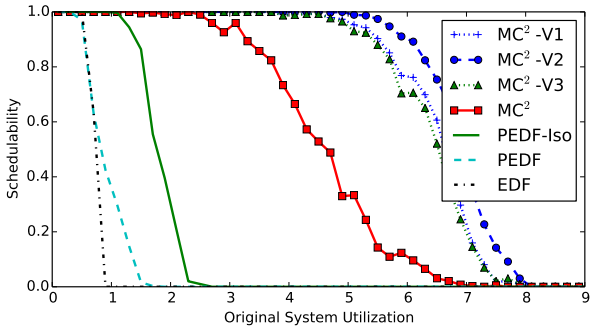
C-Heavy, Short, Mod., Light, Large Var.



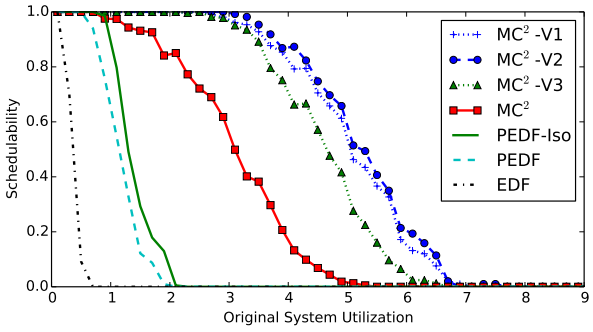
All-Mod., Long, Heavy, Heavy, Large Var.



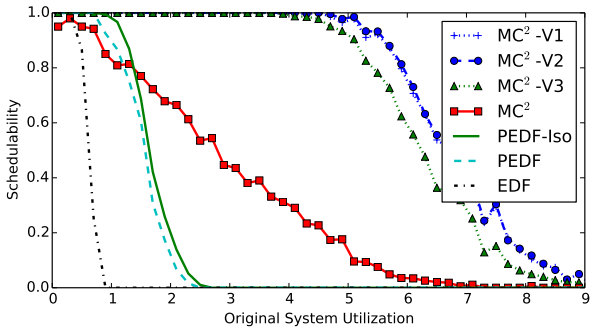
A-Heavy, Short, Mod., Mod., Large Var.



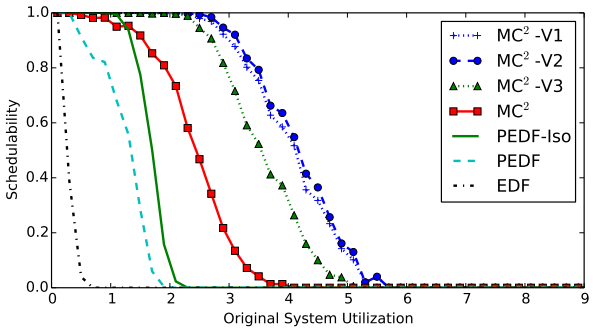
C-Heavy, Long, Heavy, Heavy, Const.



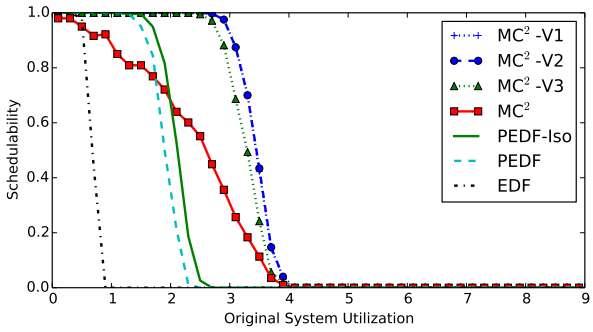
BC-Mod., Cont., Heavy, Mod., Large Var.



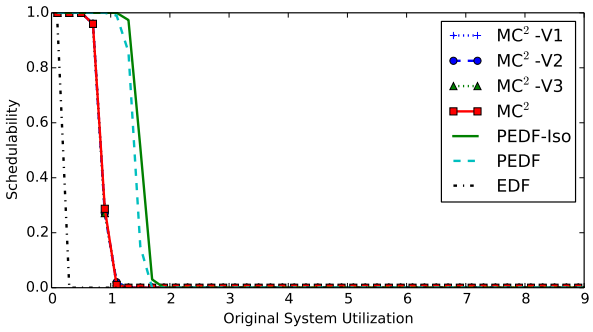
C-Heavy, Short, Heavy, Mod., Large Var.



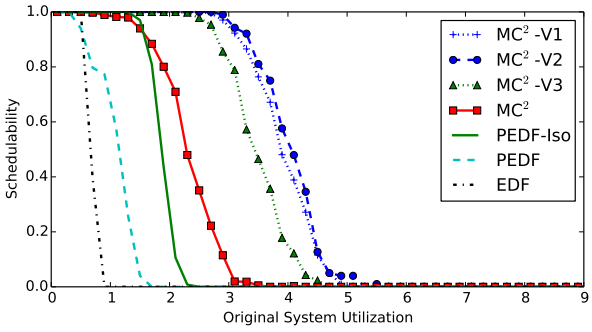
B-Heavy, Cont., Heavy, Mod., Large Var.



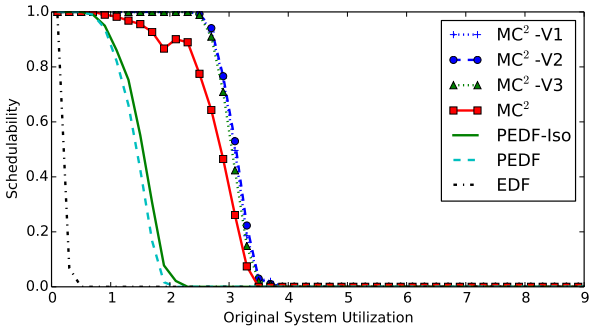
All-Mod., Short, Mod., Light, Large Var.



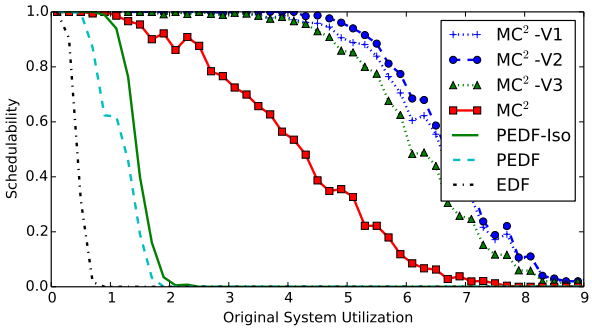
A-Heavy, Cont., Light, Mod., Const.



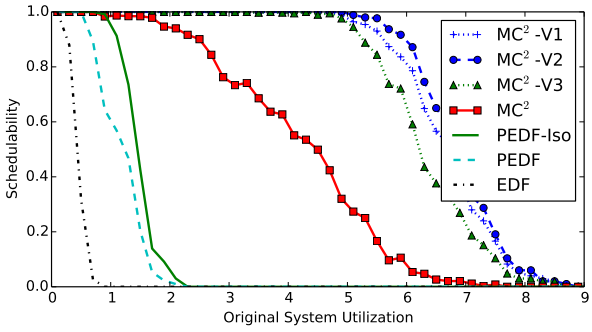
B-Heavy, Long, Heavy, Heavy, Const.



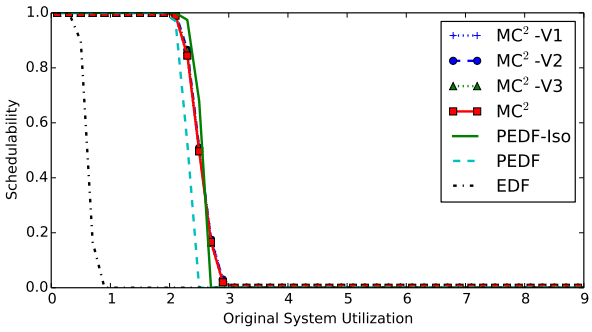
AC-Mod., Cont., Mod., Heavy, Large Var.



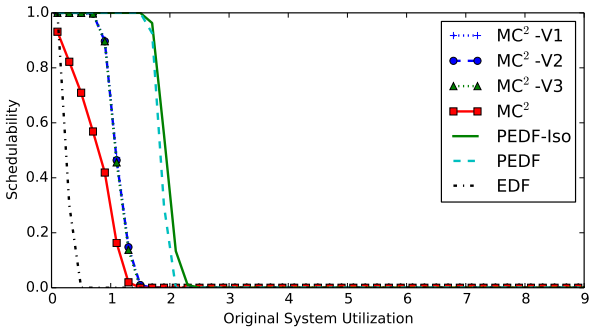
C-Heavy, Cont., Heavy, Mod., Large Var.



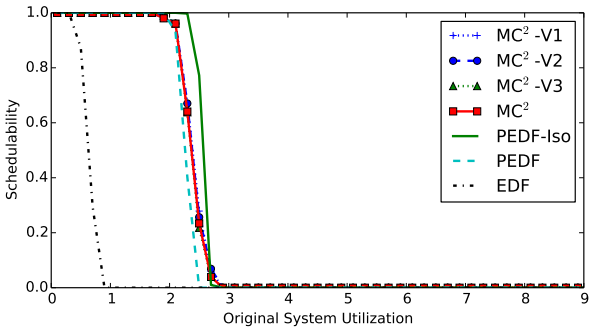
C-Heavy, Cont., Heavy, Mod., Large Var.



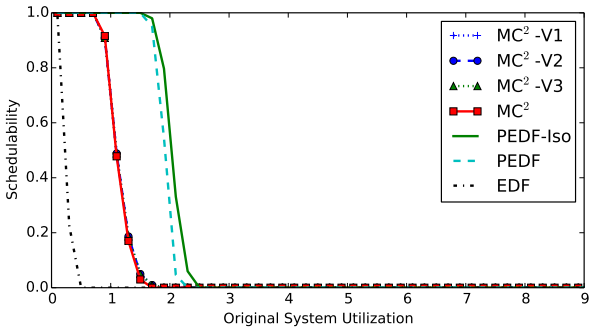
AC-Mod., Long, Light, Light, Const.



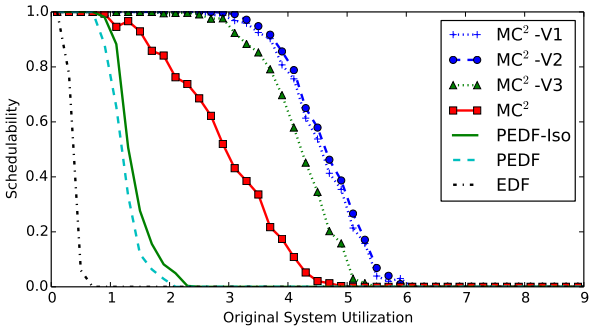
C-Heavy, Short, Light, Heavy, Large Var.



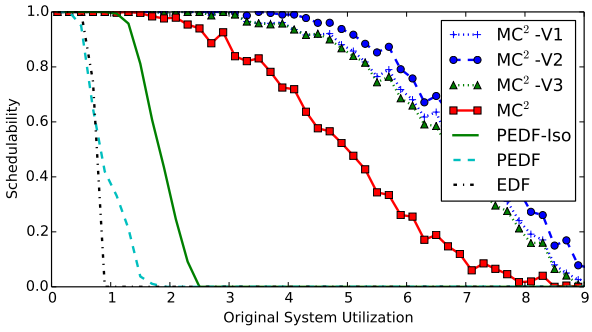
A-Heavy, Long, Light, Light, Large Var.



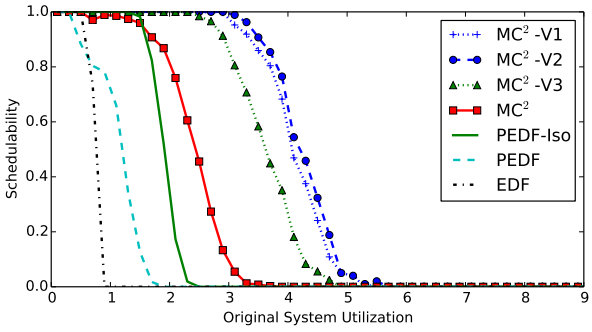
C-Heavy, Cont., Light, Mod., Large Var.



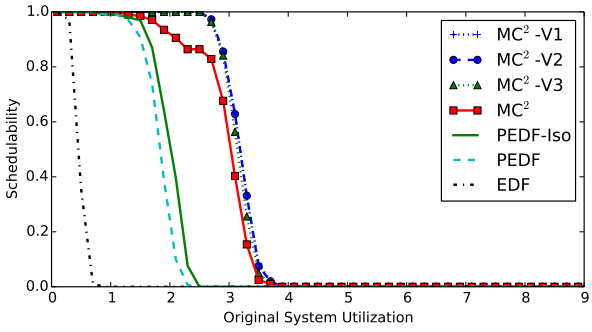
All-Mod., Cont., Heavy, Mod., Large Var.



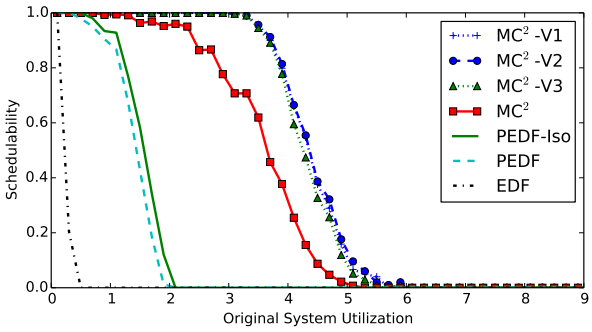
C-Heavy, Long, Heavy, Mod., Large Var.



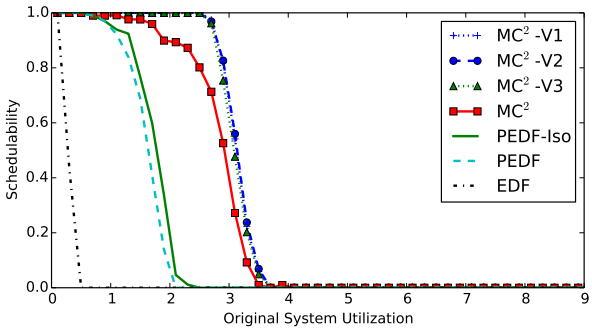
B-Heavy, Long, Heavy, Mod., Large Var.



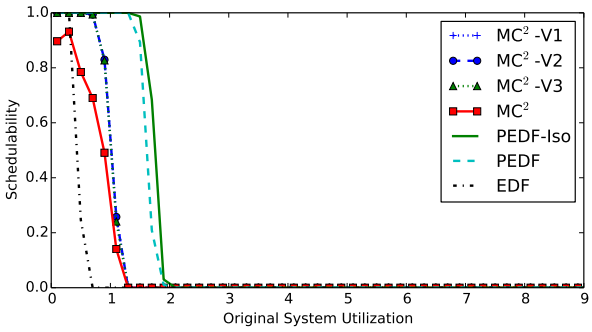
AC-Mod., Cont., Mod., Light, Large Var.



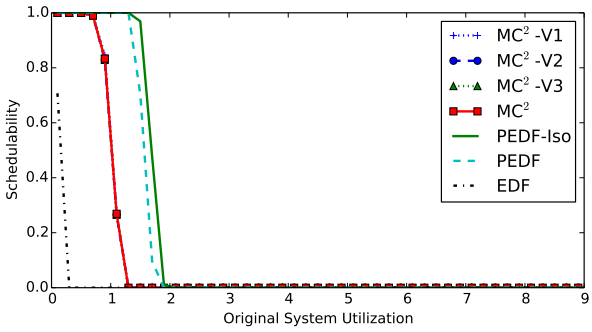
C-Heavy, Cont., Mod., Heavy, Large Var.



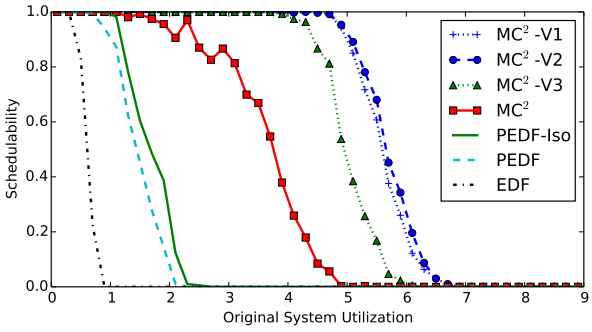
AC-Mod., Cont., Mod., Mod., Large Var.



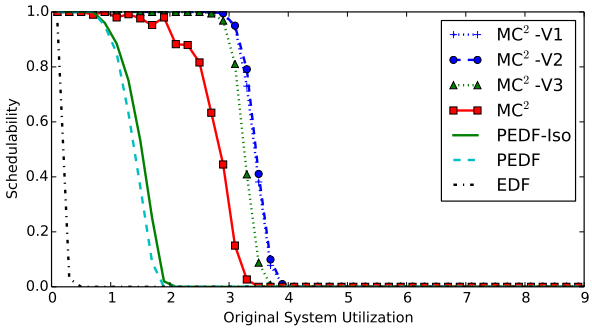
All-Mod., Short, Light, Light, Large Var.



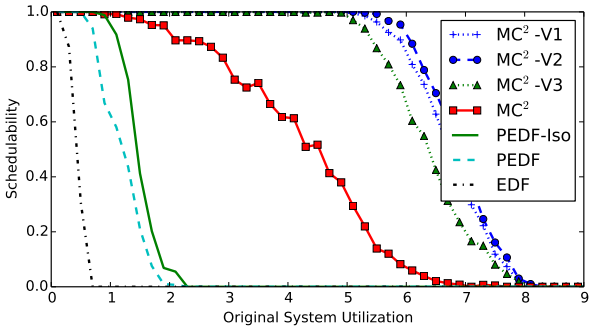
All-Mod., Cont., Light, Heavy, Large Var.



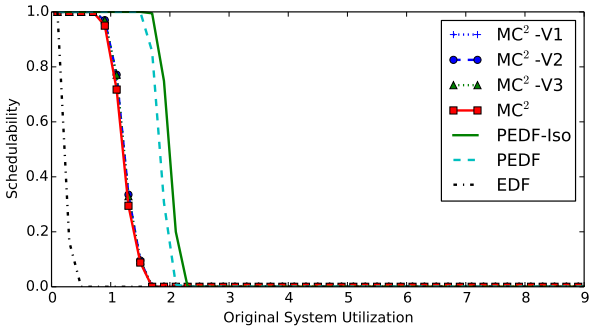
BC-Mod., Cont., Heavy, Light, Const.



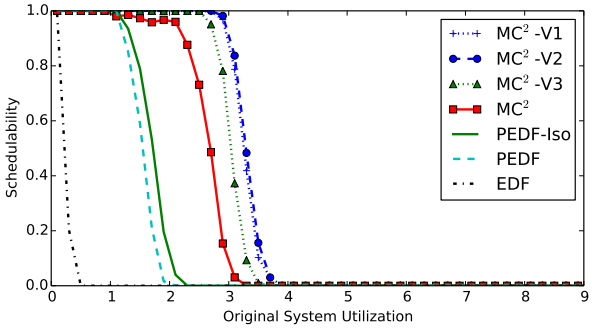
All-Mod., Cont., Mod., Heavy, Const.



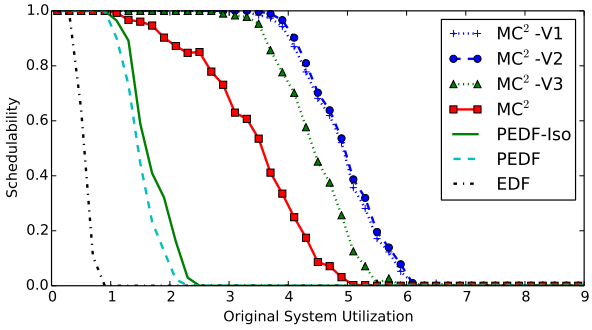
C-Heavy, Cont., Heavy, Mod., Const.



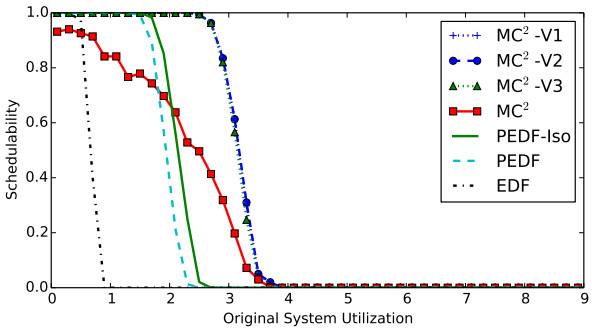
BC-Mod., Cont., Light, Mod., Const.



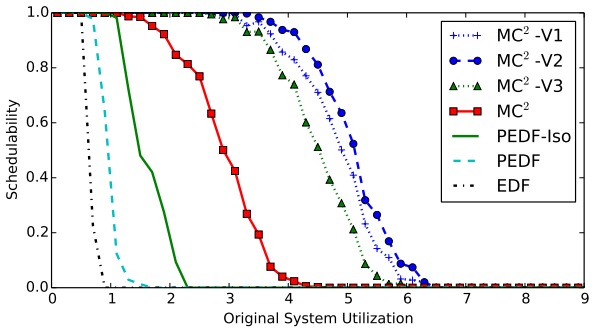
AB-Mod., Cont., Mod., Mod., Const.



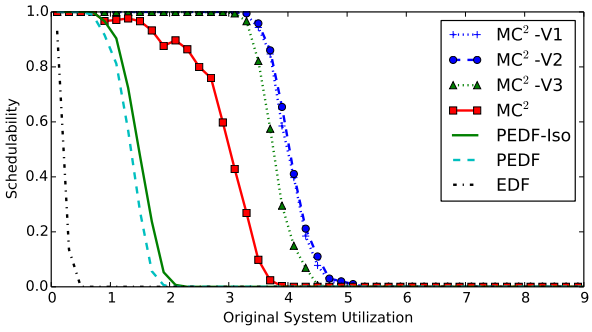
All-Mod., Cont., Heavy, Light, Large Var.



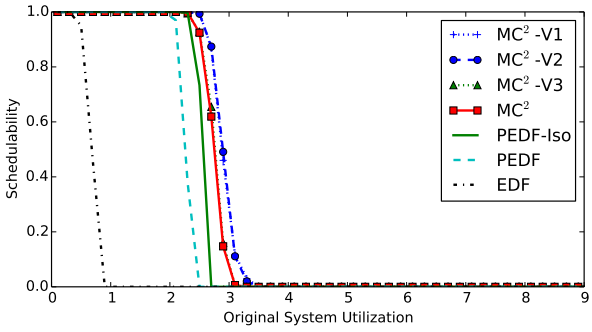
AC-Mod., Short, Mod., Light, Large Var.



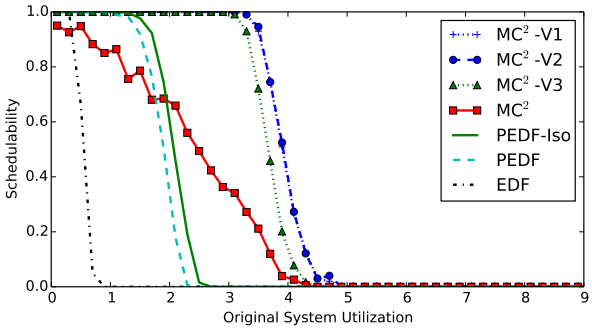
AC-Mod., Long, Heavy, Heavy, Large Var.



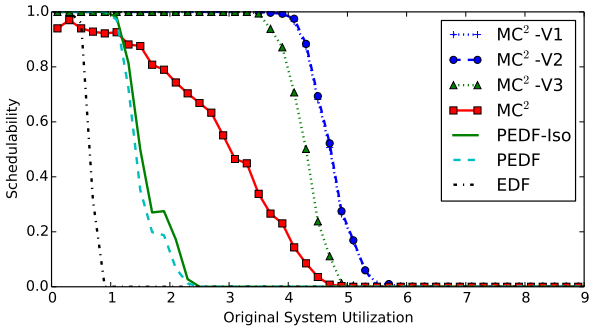
BC-Mod., Cont., Mod., Heavy, Const.



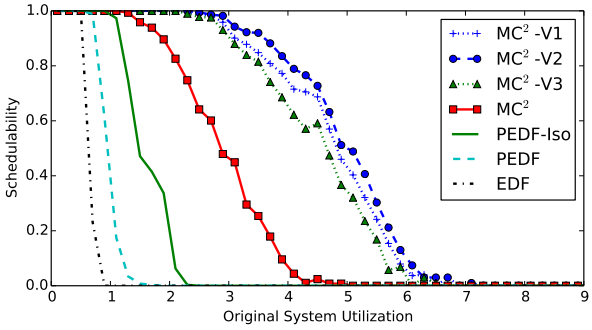
AB-Mod., Long, Light, Light, Const.



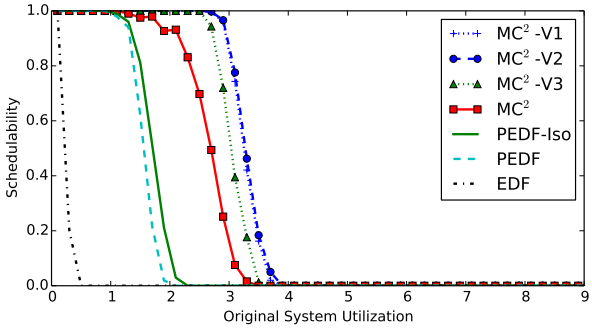
BC-Mod., Short, Mod., Mod., Const.



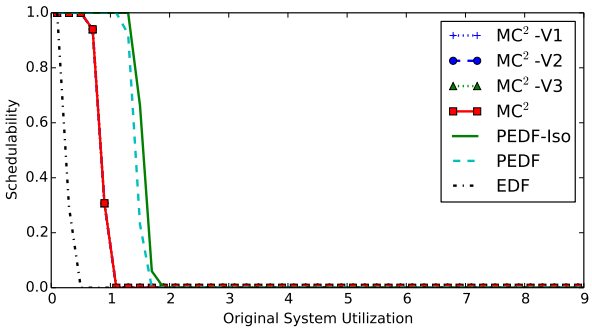
All-Mod., Short, Heavy, Mod., Const.



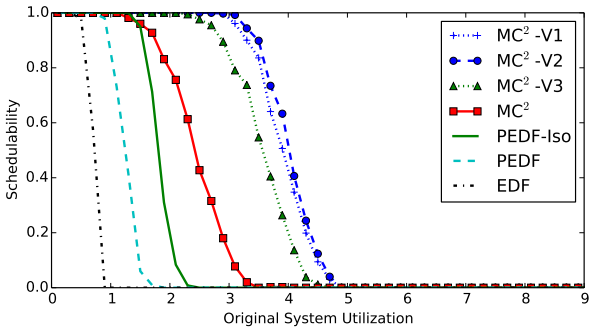
AC-Mod., Long, Heavy, Heavy, Large Var.



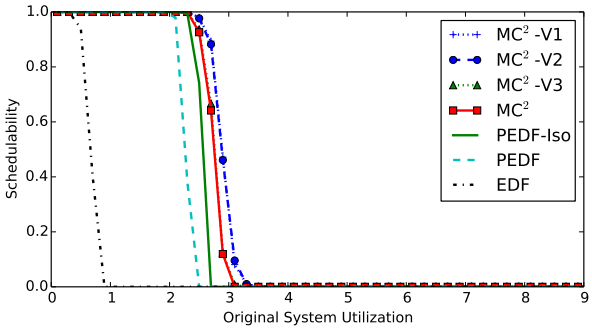
AB-Mod., Cont., Mod., Mod., Large Var.



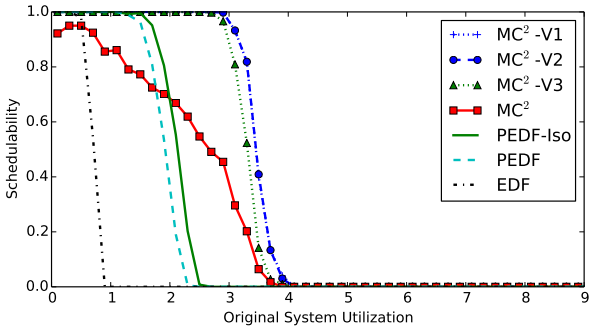
A-Heavy, Cont., Light, Light, Const.



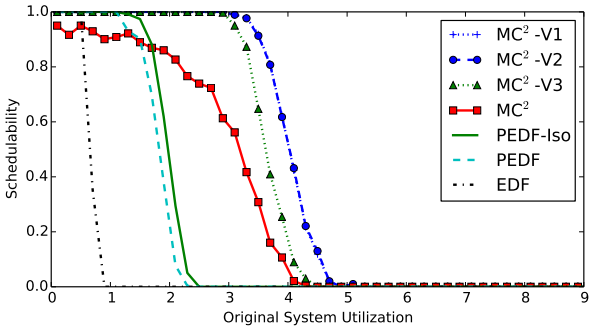
A-Heavy, Long, Heavy, Mod., Const.



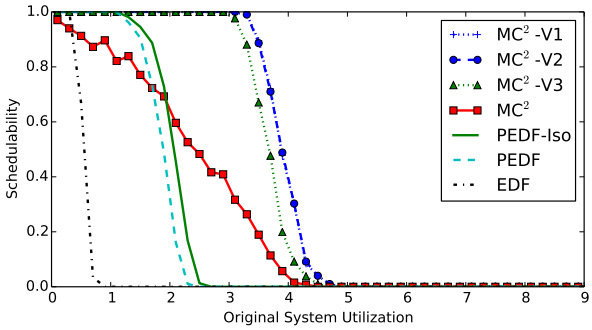
AB-Mod., Long, Light, Light, Const.



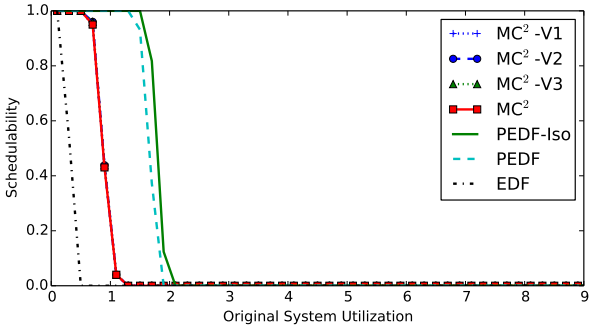
All-Mod., Short, Mod., Light, Const.



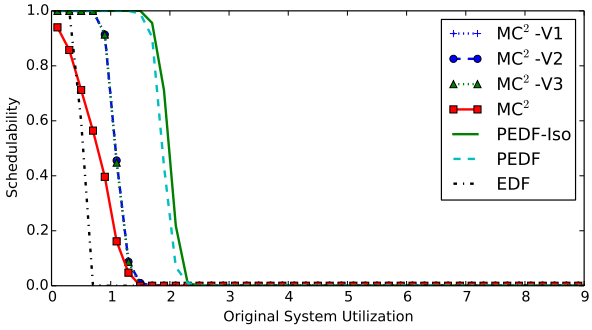
AB-Mod., Short, Heavy, Mod., Const.



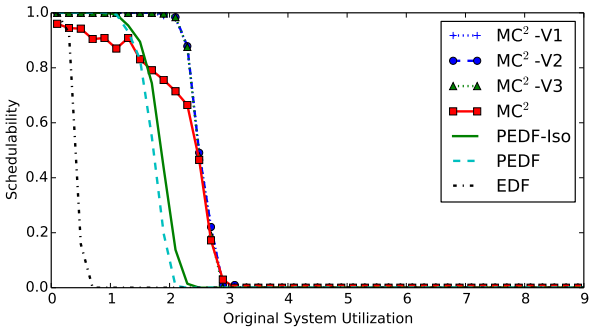
BC-Mod., Short, Mod., Mod., Const.



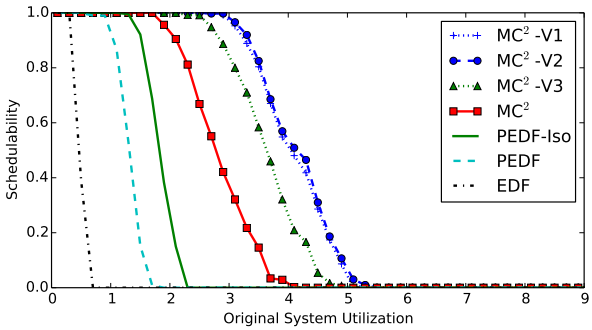
AC-Mod., Cont., Light, Light, Const.



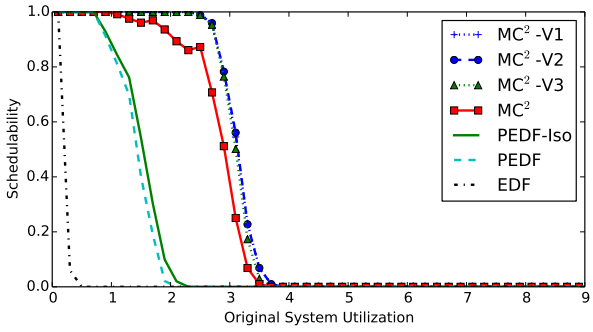
C-Heavy, Short, Light, Light, Large Var.



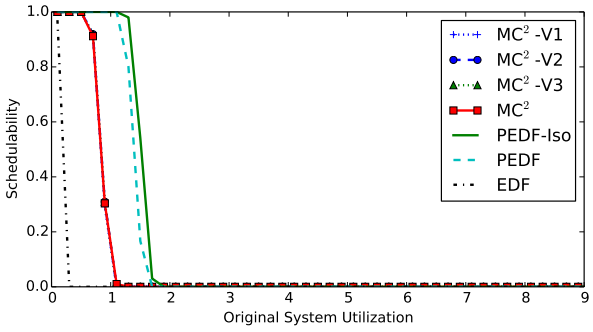
A-Heavy, Short, Mod., Heavy, Const.



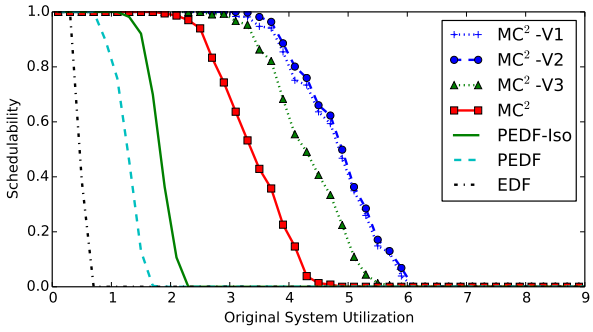
AB-Mod., Long, Mod., Heavy, Large Var.



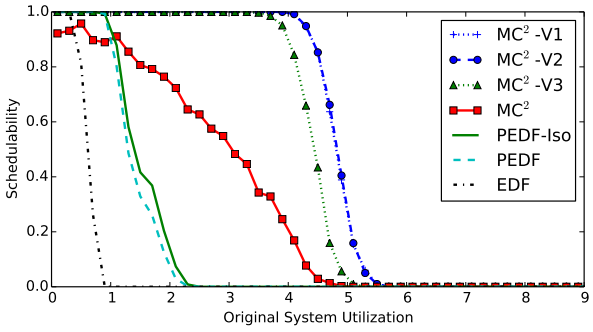
AC-Mod., Cont., Mod., Heavy, Large Var.



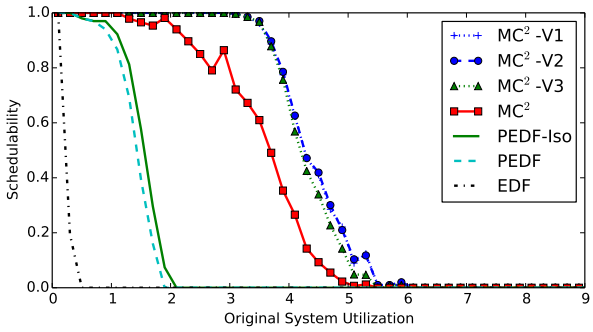
A-Heavy, Cont., Light, Mod., Large Var.



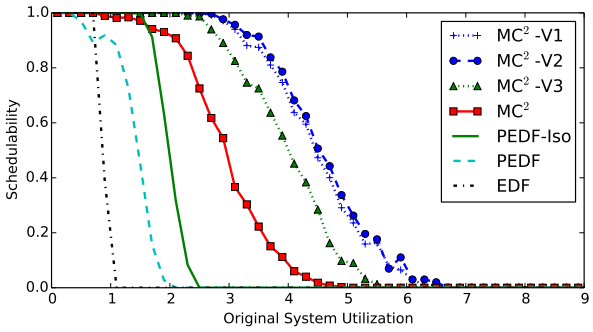
All-Mod., Long, Mod., Heavy, Large Var.



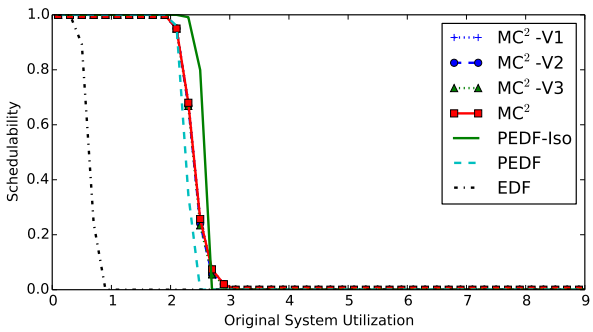
AC-Mod., Short, Heavy, Mod., Const.



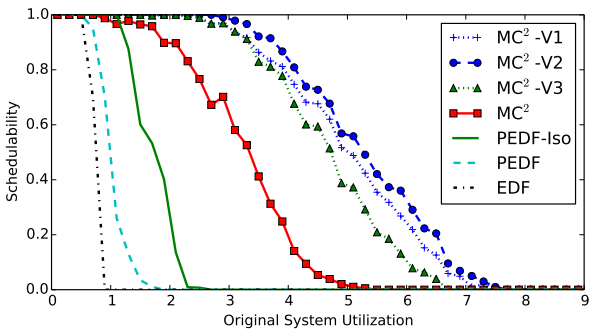
C-Heavy, Cont., Mod., Heavy, Const.



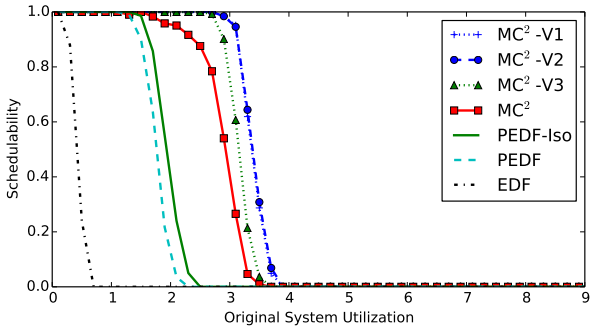
B-Heavy, Long, Heavy, Light, Large Var.



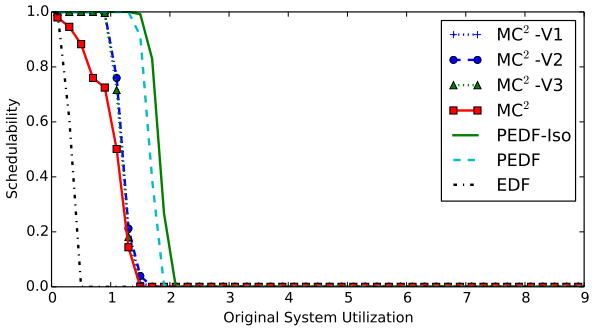
A-Heavy, Long, Light, Light, Large Var.



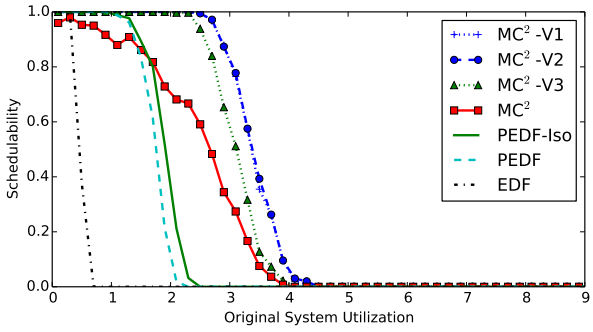
BC-Mod., Long, Heavy, Mod., Large Var.



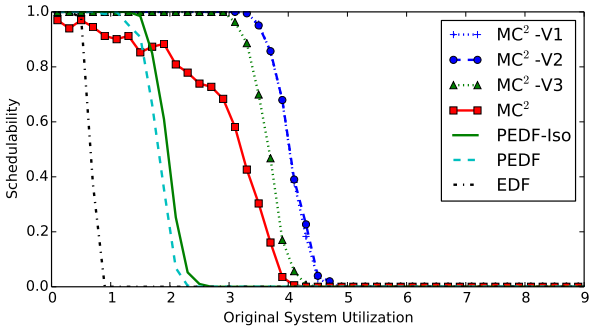
AB-Mod., Cont., Mod., Light, Const.



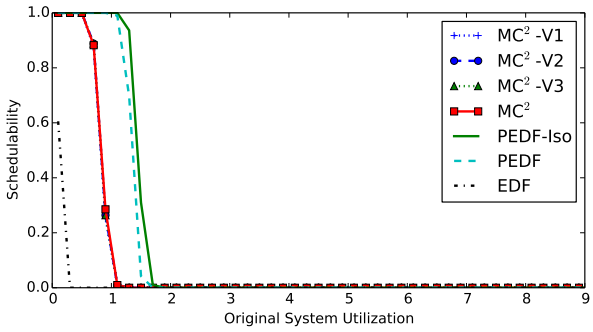
B-Heavy, Short, Light, Heavy, Const.



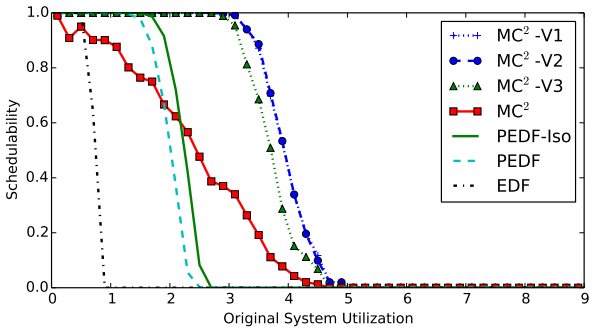
B-Heavy, Short, Mod., Heavy, Large Var.



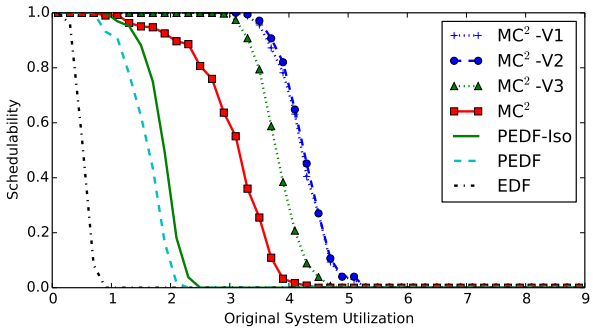
AB-Mod., Short, Heavy, Mod., Const.



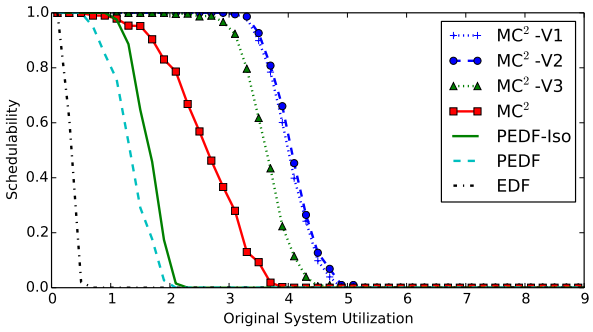
A-Heavy, Cont., Light, Heavy, Const.



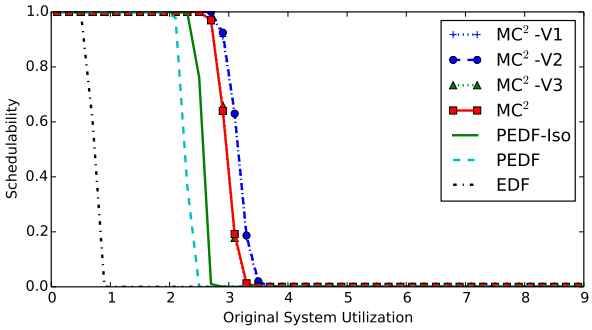
BC-Mod., Short, Mod., Light, Large Var.



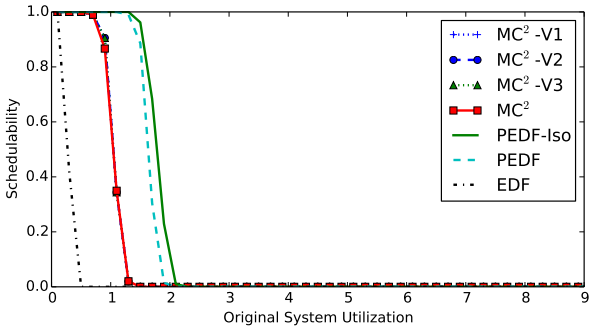
AB-Mod., Cont., Heavy, Light, Large Var.



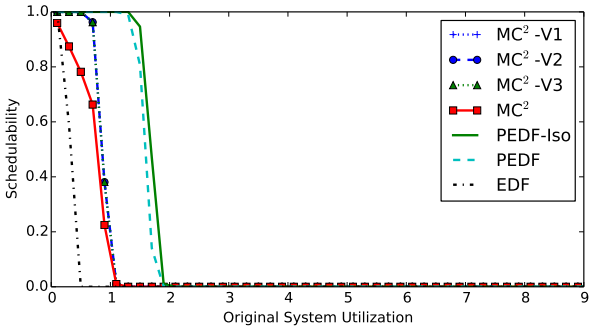
AB-Mod., Cont., Heavy, Mod., Large Var.



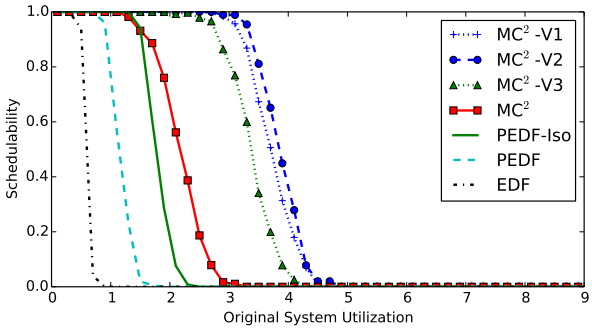
B-Heavy, Long, Light, Light, Const.



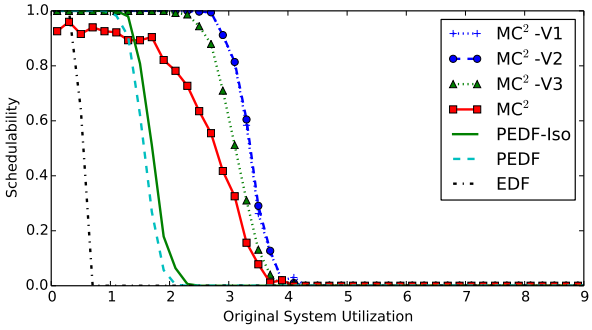
AB-Mod., Cont., Light, Light, Large Var.



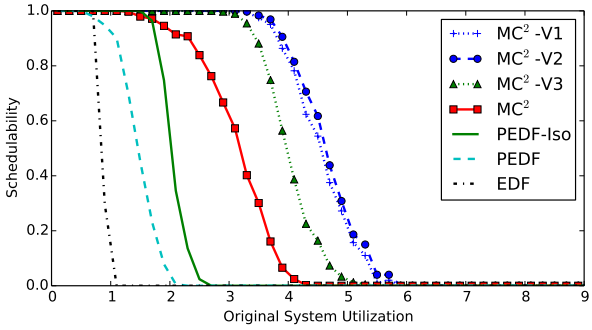
AC-Mod., Short, Light, Mod., Large Var.



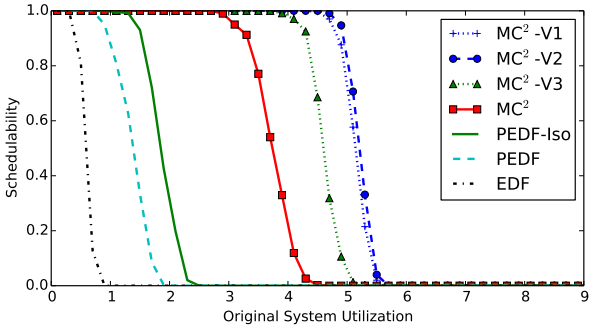
A-Heavy, Long, Heavy, Heavy, Const.



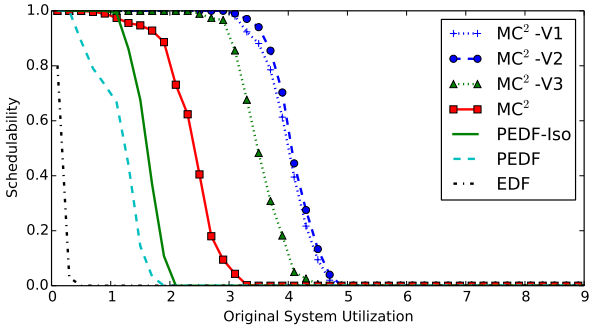
A-Heavy, Short, Heavy, Heavy, Const.



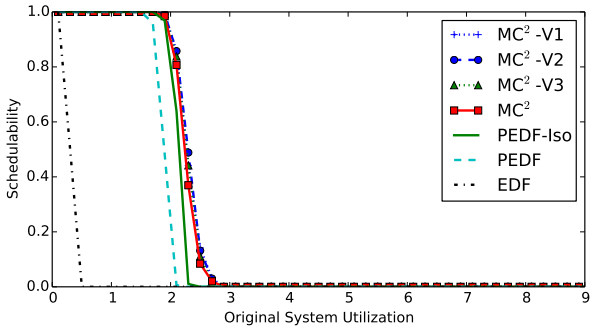
AB-Mod., Long, Heavy, Light, Const.



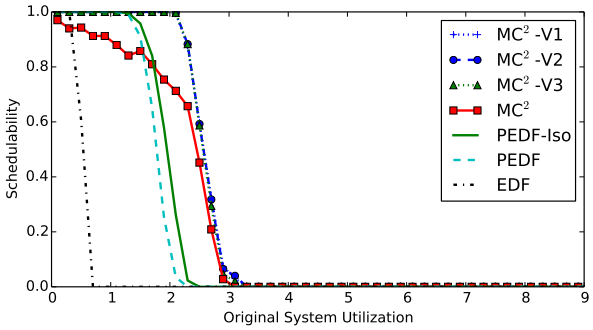
All-Mod., Long, Mod., Mod., Const.



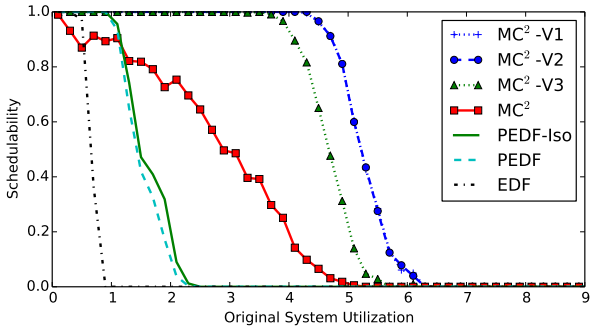
B-Heavy, Cont., Heavy, Heavy, Const.



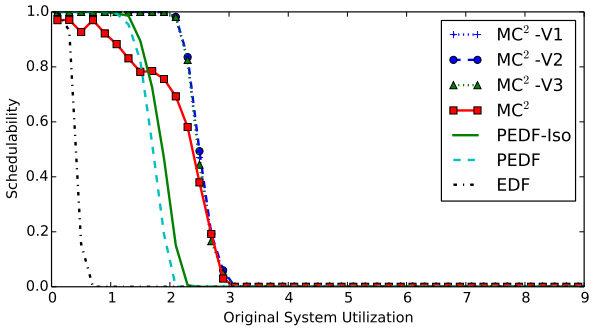
A-Heavy, Long, Light, Heavy, Large Var.



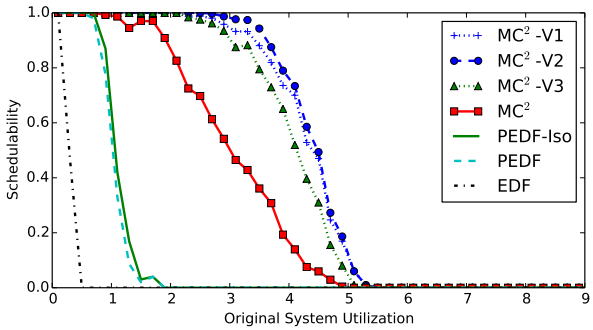
A-Heavy, Short, Mod., Mod., Const.



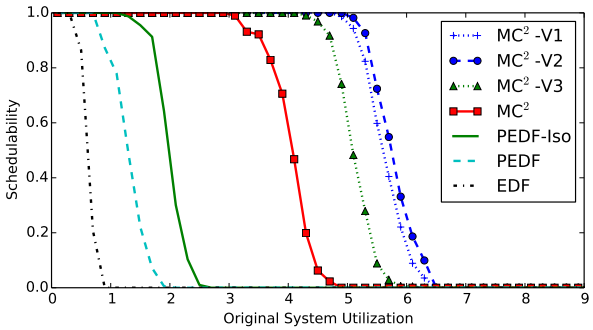
BC-Mod., Short, Heavy, Mod., Const.



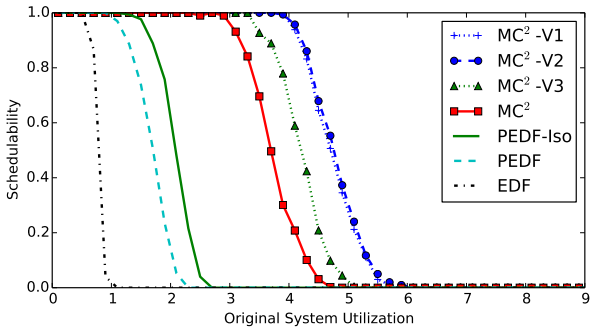
A-Heavy, Short, Mod., Heavy, Const.



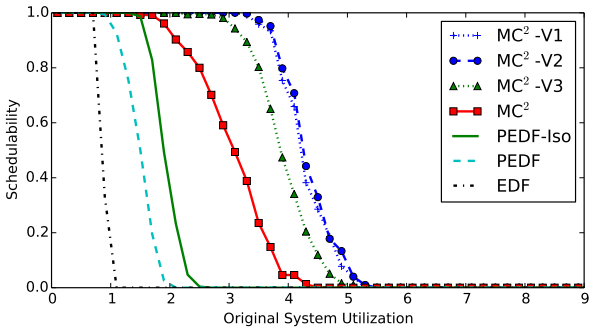
AC-Mod., Cont., Heavy, Heavy, Large Var.



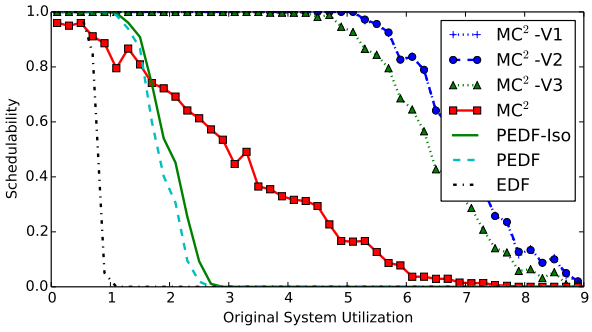
BC-Mod., Long, Mod., Mod., Const.



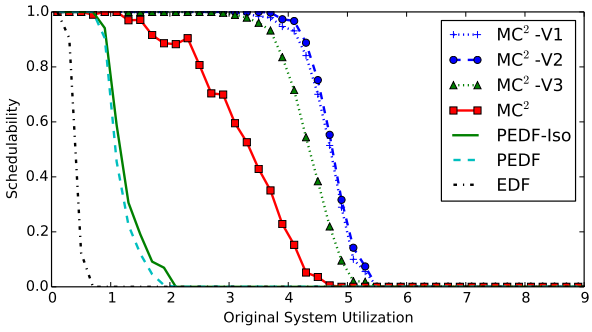
AB-Mod., Long, Mod., Light, Const.



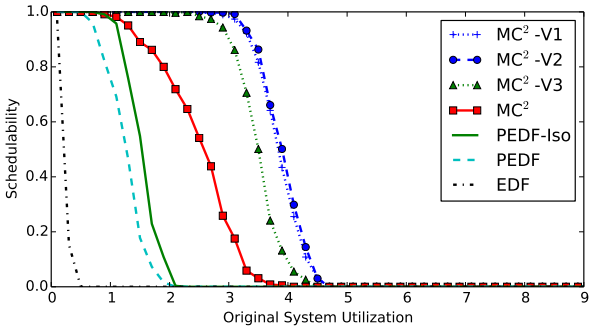
A-Heavy, Long, Heavy, Light, Const.



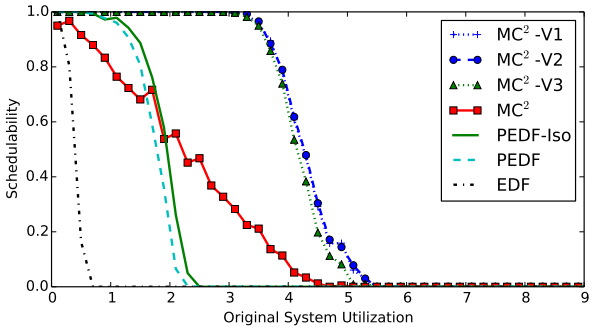
C-Heavy, Short, Heavy, Light, Large Var.



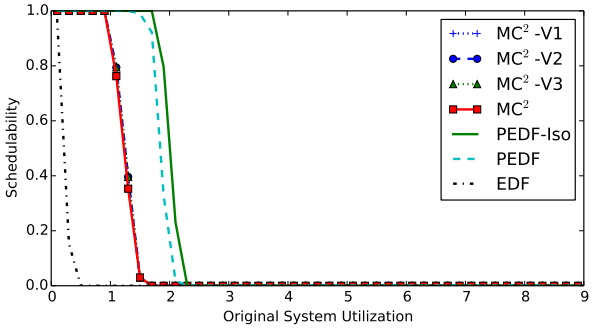
AC-Mod., Cont., Heavy, Mod., Large Var.



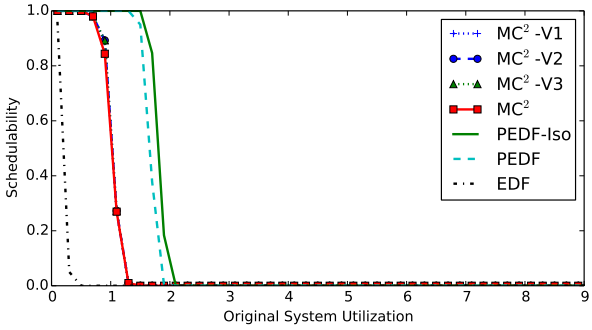
AB-Mod., Cont., Heavy, Heavy, Large Var.



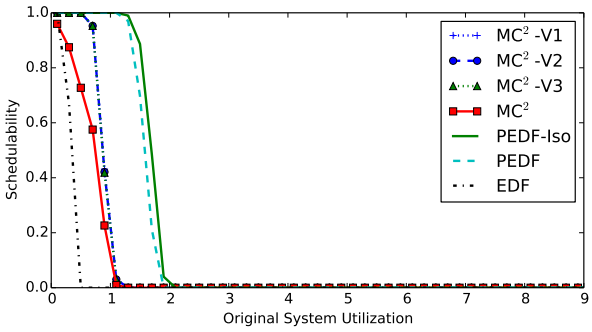
C-Heavy, Short, Mod., Heavy, Const.



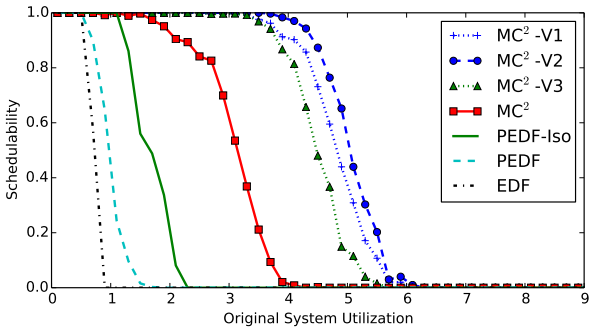
BC-Mod., Cont., Light, Mod., Const.



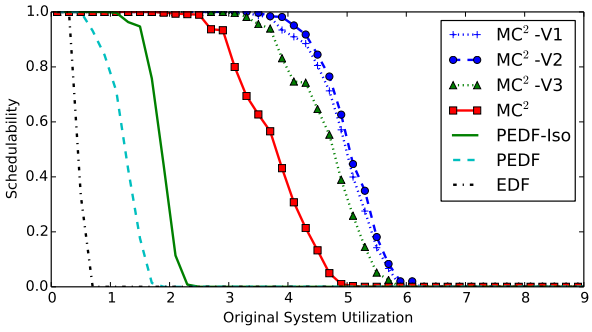
All-Mod., Cont., Light, Mod., Const.



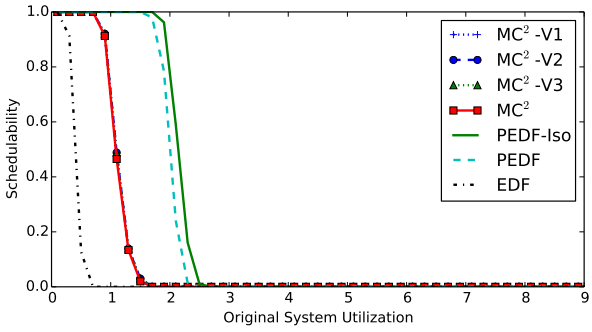
AC-Mod., Short, Light, Mod., Large Var.



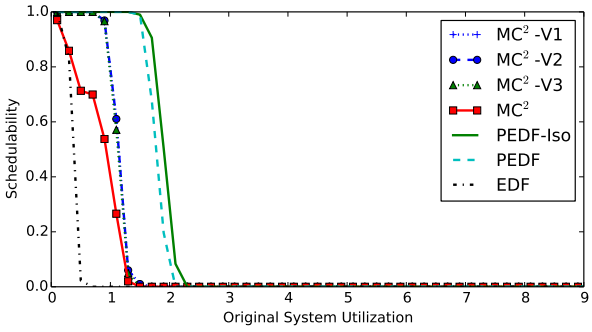
BC-Mod., Long, Heavy, Heavy, Const.



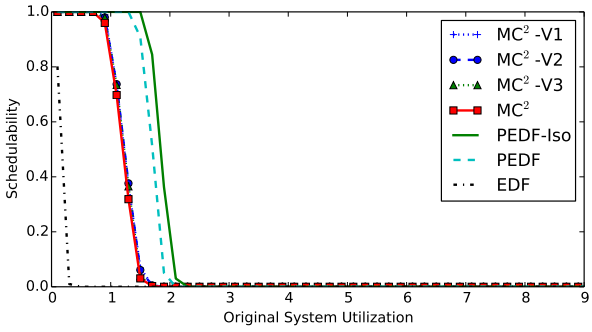
AC-Mod., Long, Mod., Heavy, Large Var.



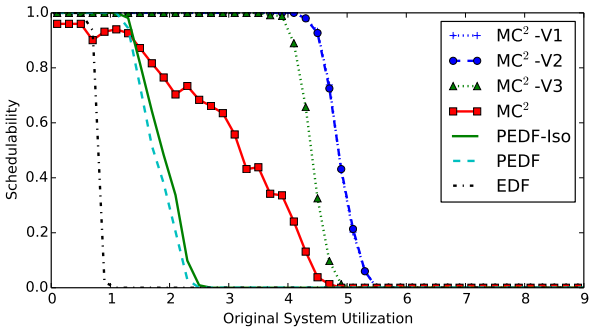
C-Heavy, Cont., Light, Light, Large Var.



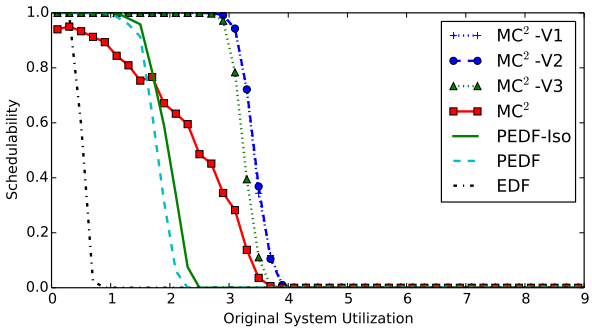
BC-Mod., Short, Light, Mod., Large Var.



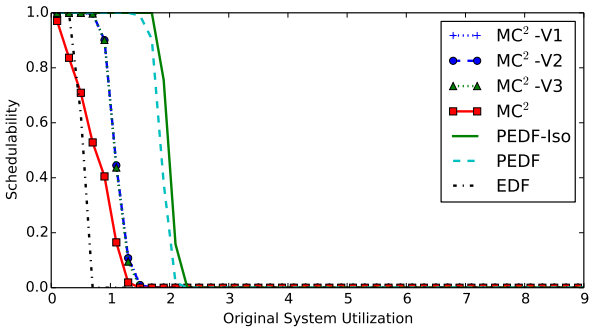
BC-Mod., Cont., Light, Heavy, Large Var.



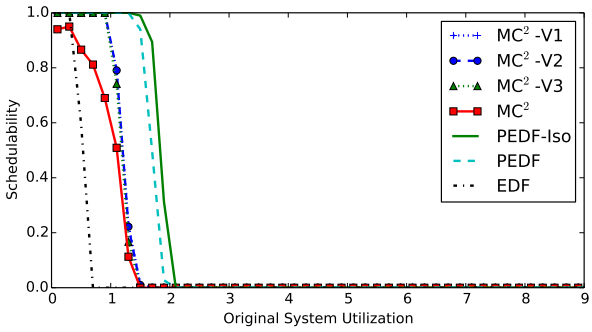
All-Mod., Short, Heavy, Light, Const.



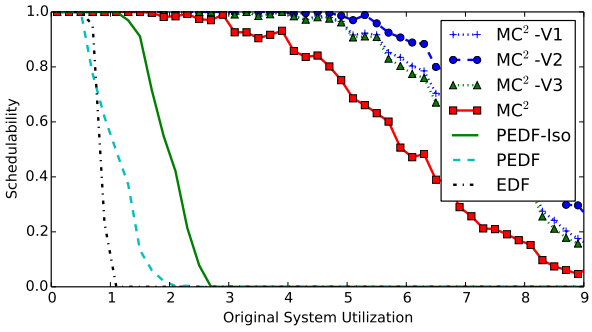
All-Mod., Short, Mod., Mod., Const.



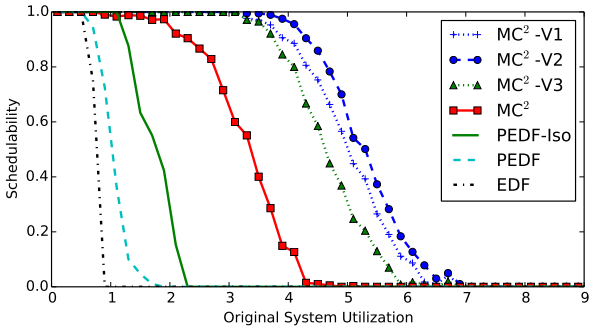
C-Heavy, Short, Light, Light, Const.



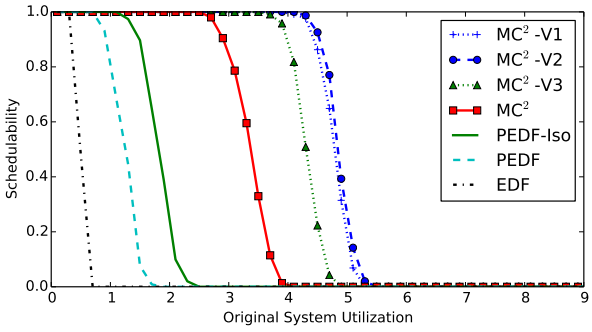
B-Heavy, Short, Light, Light, Const.



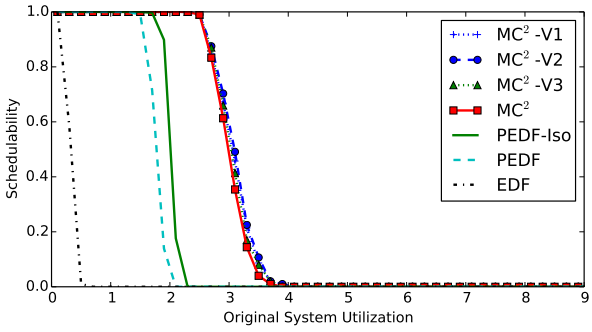
C-Heavy, Long, Heavy, Light, Large Var.



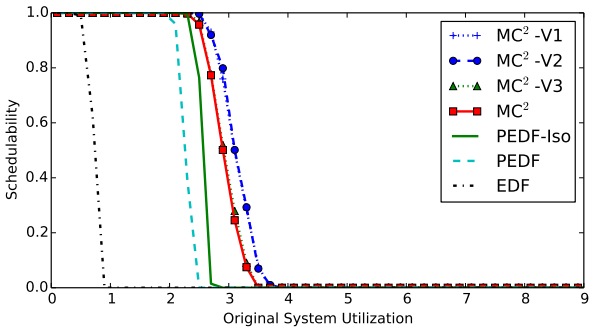
BC-Mod., Long, Heavy, Mod., Large Var.



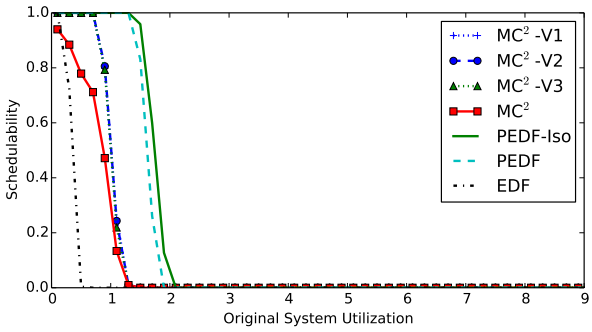
All-Mod., Long, Mod., Heavy, Const.



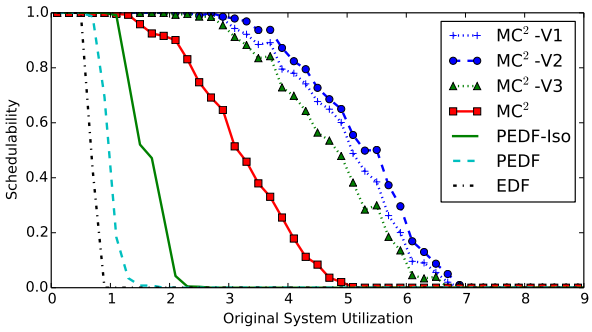
C-Heavy, Long, Light, Heavy, Const.



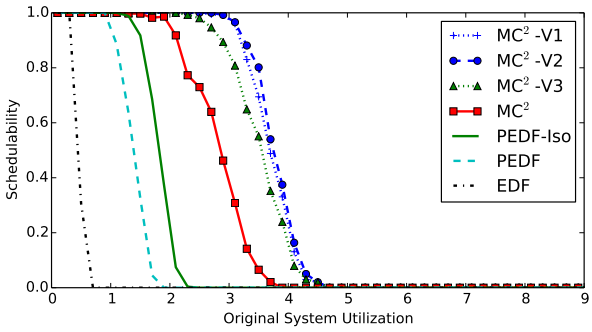
B-Heavy, Long, Light, Light, Large Var.



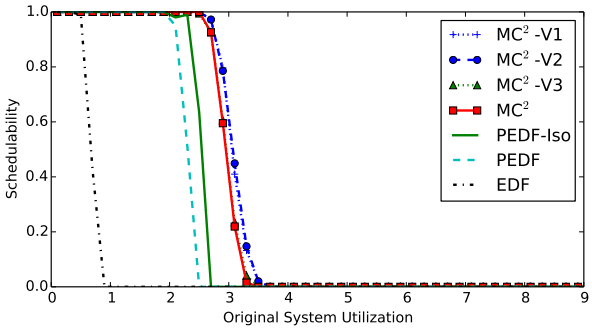
All-Mod., Short, Light, Mod., Large Var.



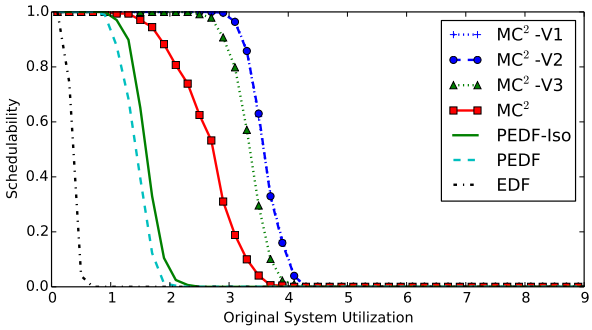
AC-Mod., Long, Heavy, Mod., Large Var.



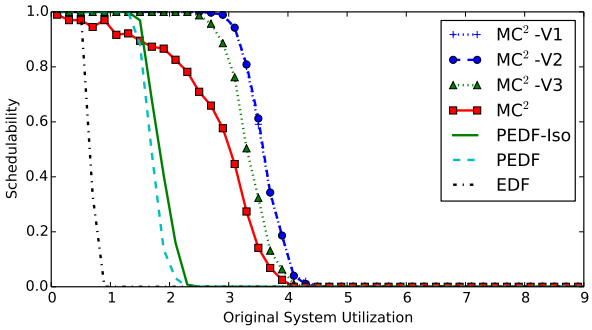
A-Heavy, Long, Mod., Heavy, Large Var.



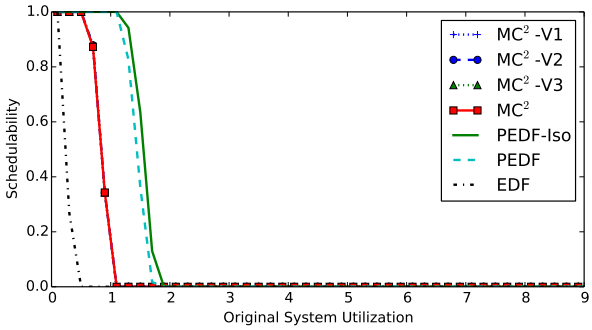
BC-Mod., Long, Light, Light, Large Var.



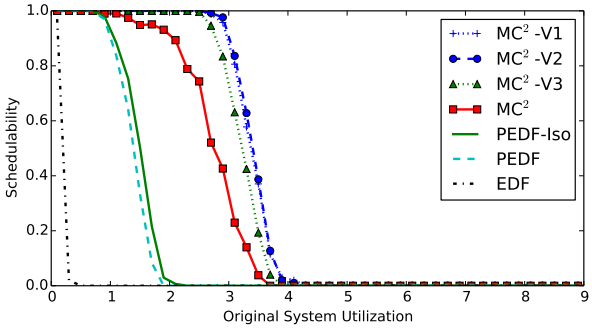
A-Heavy, Cont., Heavy, Mod., Const.



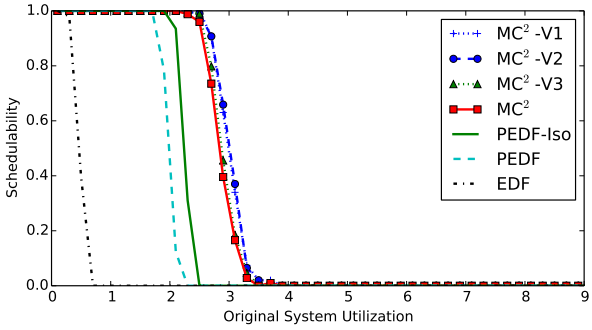
A-Heavy, Short, Heavy, Mod., Const.



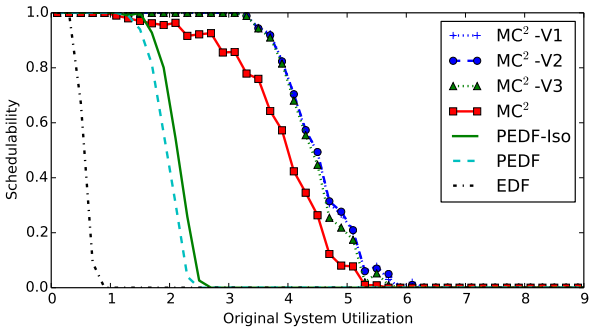
A-Heavy, Cont., Light, Light, Large Var.



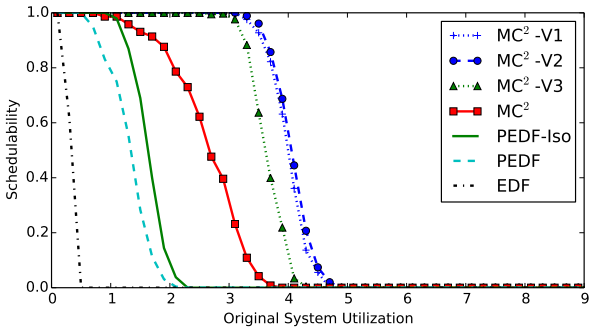
All-Mod., Cont., Mod., Heavy, Large Var.



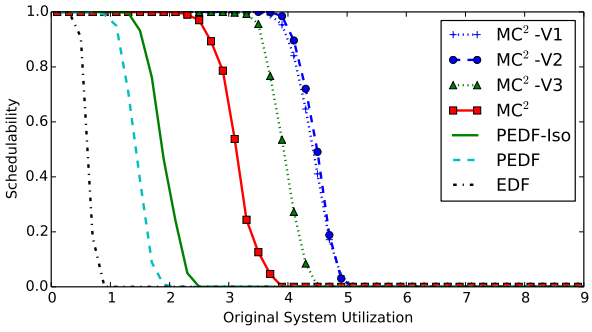
BC-Mod., Long, Light, Mod., Large Var.



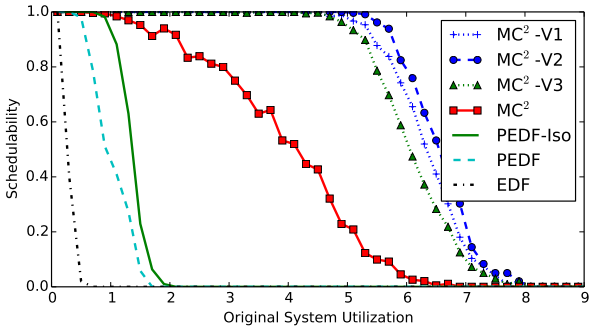
C-Heavy, Cont., Mod., Light, Const.



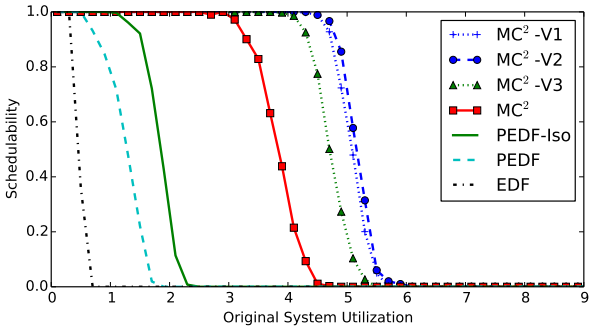
AB-Mod., Cont., Heavy, Mod., Const.



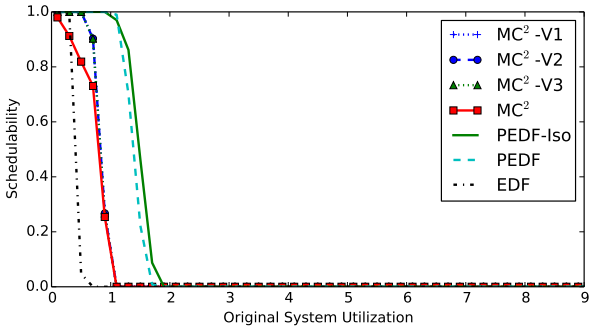
AB-Mod., Long, Mod., Mod., Const.



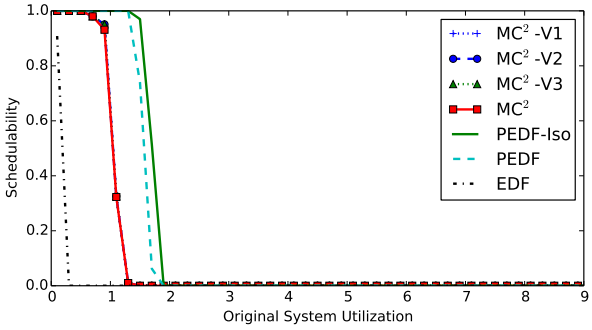
C-Heavy, Cont., Heavy, Heavy, Const.



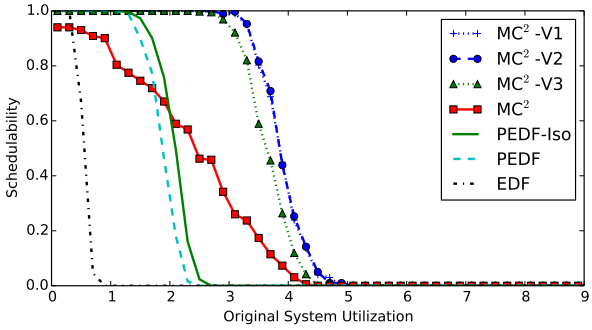
AC-Mod., Long, Mod., Heavy, Const.



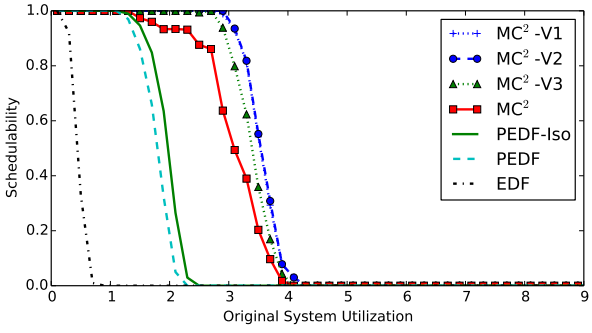
A-Heavy, Short, Light, Light, Large Var.



AB-Mod., Cont., Light, Mod., Const.

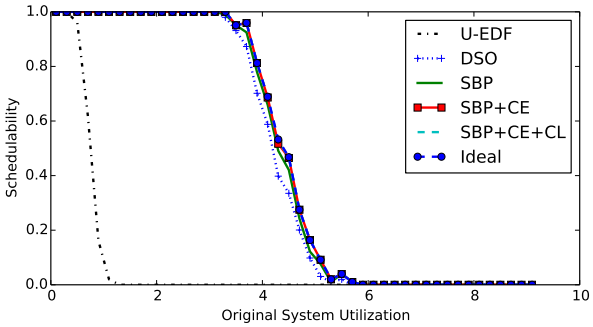


BC-Mod., Short, Mod., Mod., Large Var.

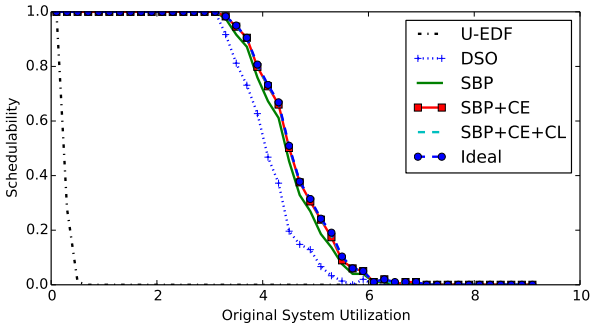


All-Mod., Cont., Mod., Light, Large Var.

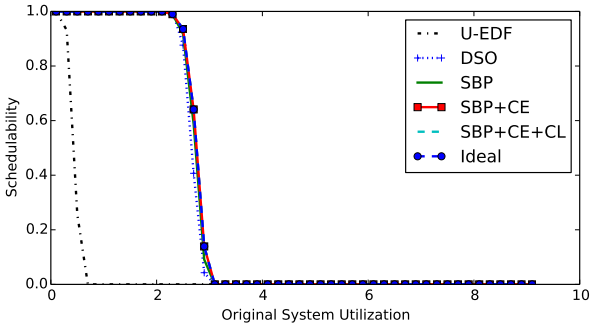
APPENDIX B: SCHEDULABILITY GRAPHS FOR THE STUDY DESCRIBED IN SECTION 4.1



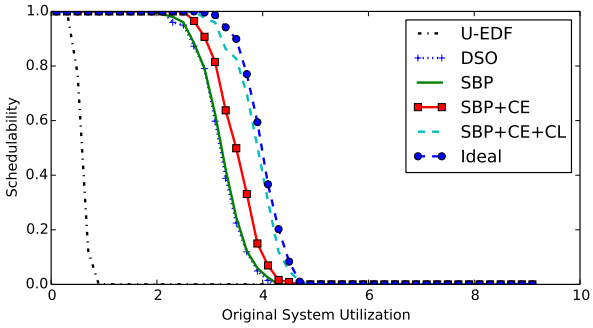
C-Heavy, Short, Mod., Light, Heavy, Small



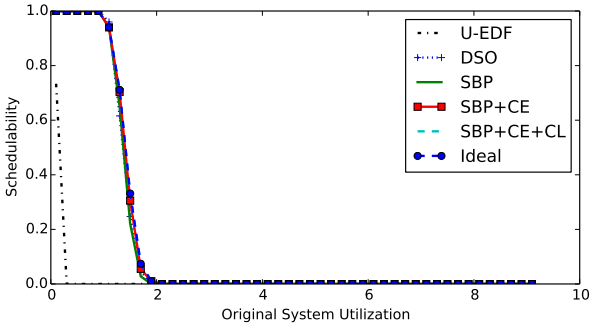
C-Heavy, Cont., Mod., Heavy, Light, Large



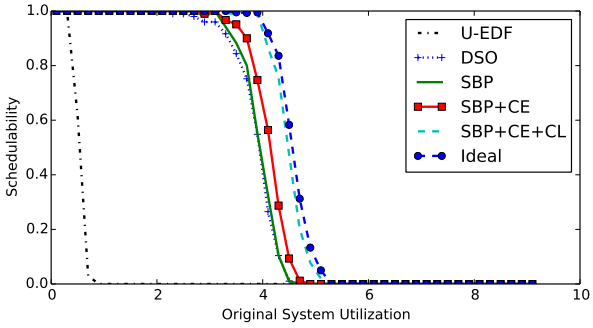
All-Mod., Long, Light, Mod., Heavy, Small



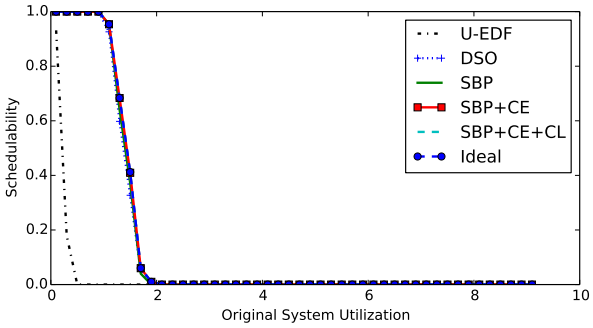
B-Heavy, Short, Heavy, Heavy, Heavy, Small



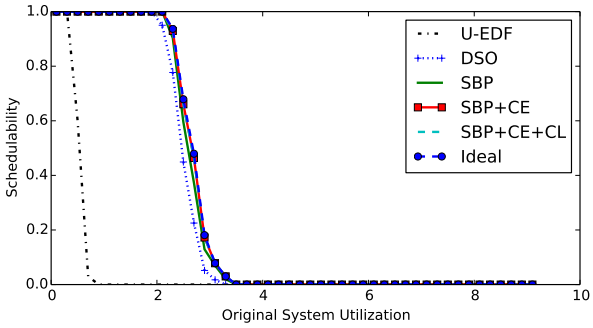
B-Heavy, Cont., Light, Heavy, Heavy, Small



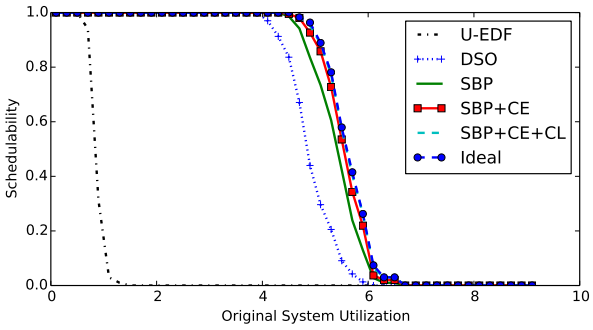
All-Mod., Short, Heavy, Heavy, Heavy, Small



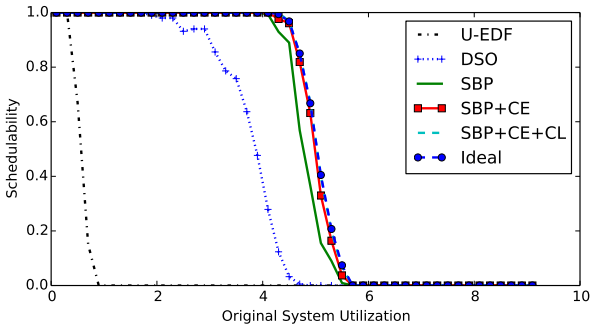
B-Heavy, Cont., Light, Mod., Light, Small



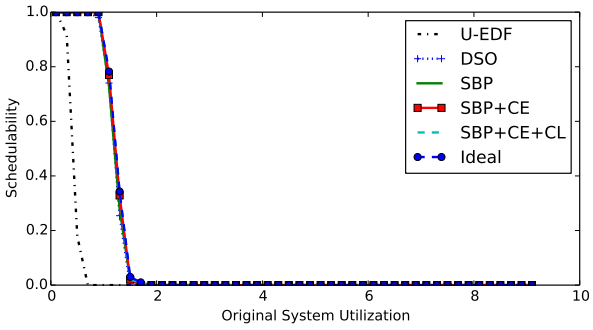
A-Heavy, Short, Mod., Mod., Light, Large



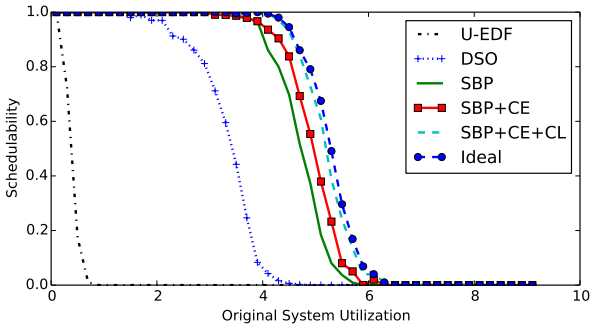
BC-Mod., Short, Heavy, Light, Heavy, Small



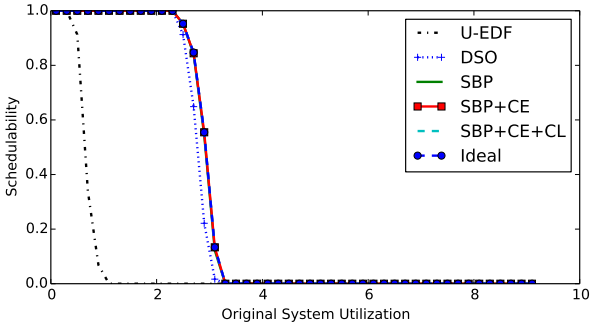
All-Mod., Cont., Heavy, Light, Heavy, Small



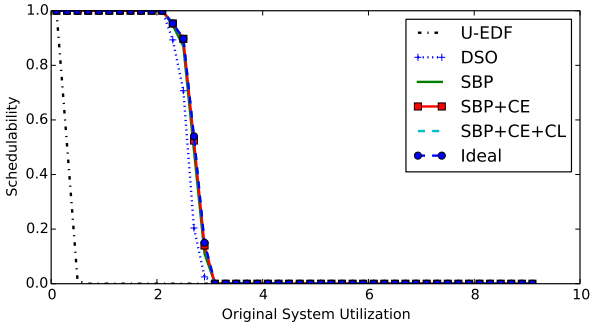
B-Heavy, Short, Light, Mod., Heavy, Large



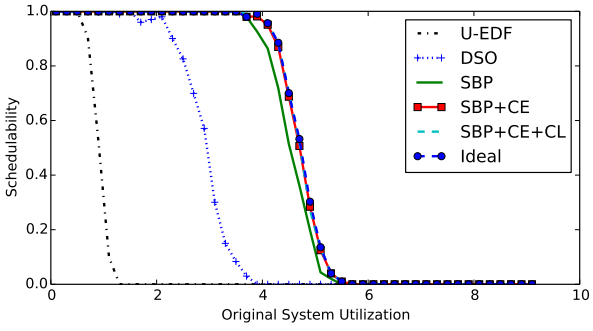
BC-Mod., Cont., Heavy, Mod., Heavy, Small



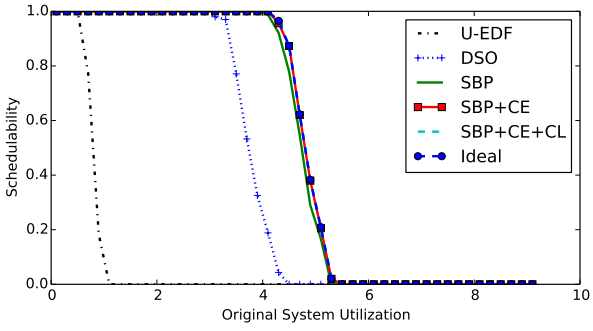
AB-Mod., Long, Light, Light, Light, Large



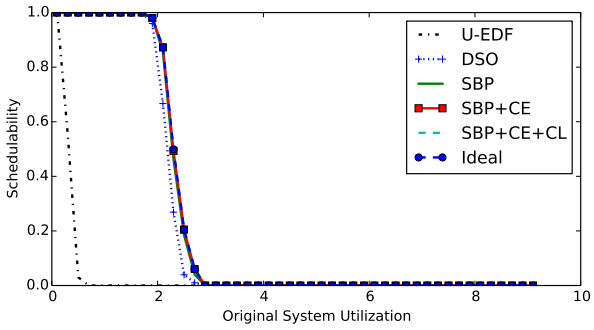
All-Mod., Long, Light, Heavy, Light, Large



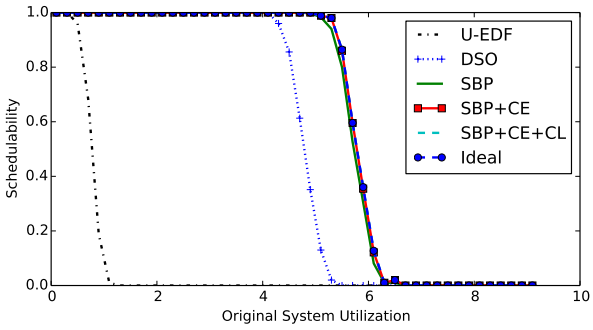
B-Heavy, Long, Heavy, Light, Light, Large



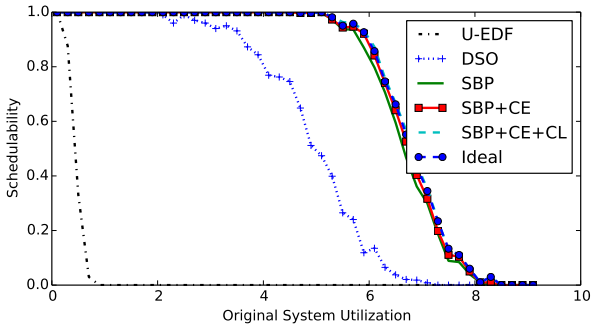
AB-Mod., Long, Mod., Light, Light, Large



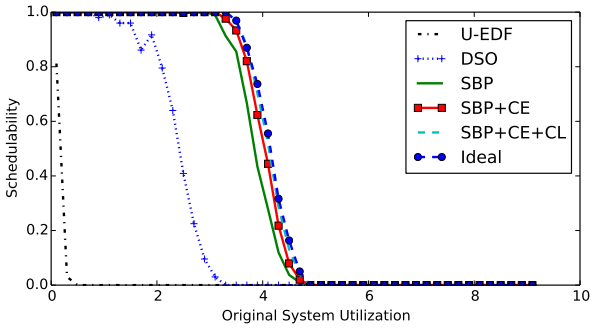
A-Heavy, Long, Light, Heavy, Light, Large



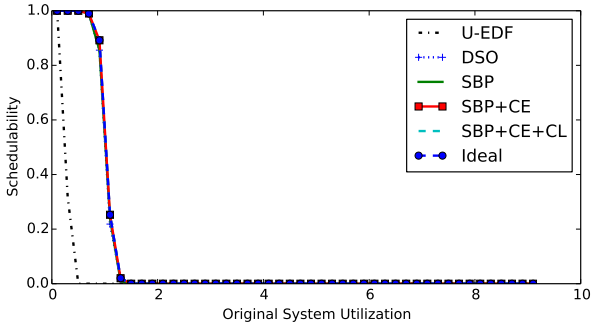
AC-Mod., Long, Mod., Light, Light, Large



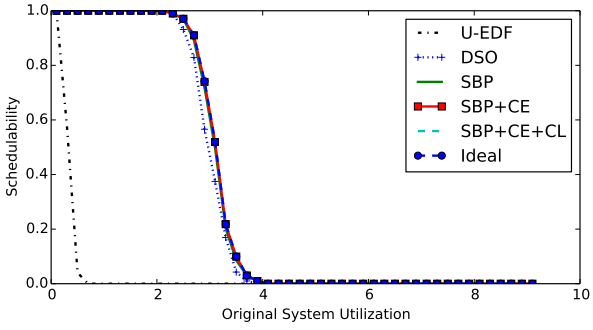
C-Heavy, Cont., Heavy, Mod., Light, Small



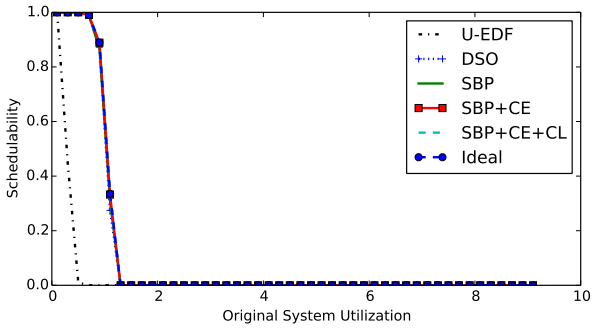
B-Heavy, Cont., Heavy, Heavy, Light, Large



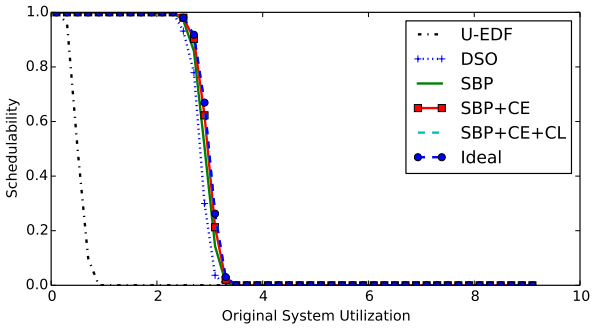
AB-Mod., Short, Light, Heavy, Light, Small



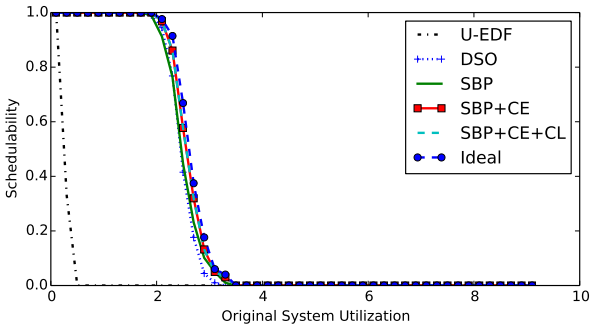
C-Heavy, Long, Light, Heavy, Light, Small



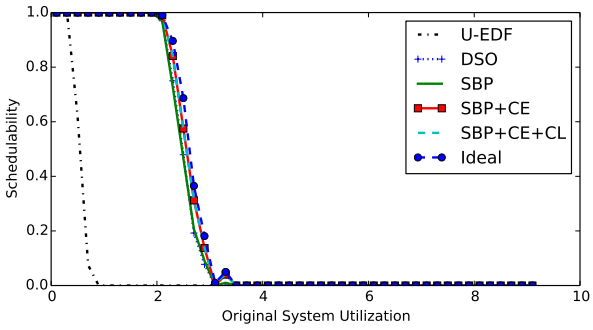
AB-Mod., Cont., Light, Light, Heavy, Small



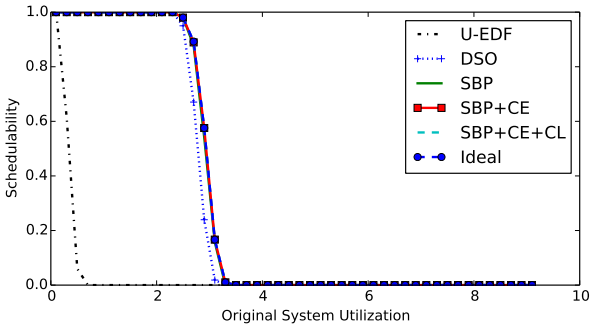
B-Heavy, Long, Light, Mod., Heavy, Large



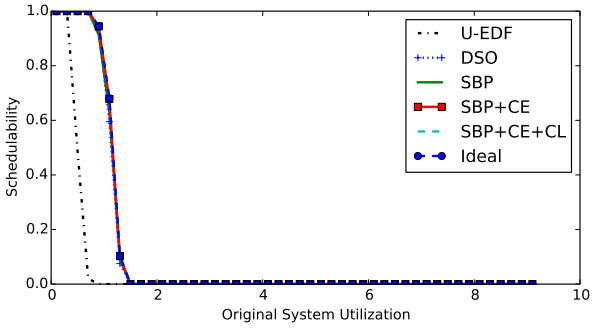
A-Heavy, Cont., Mod., Mod., Heavy, Large



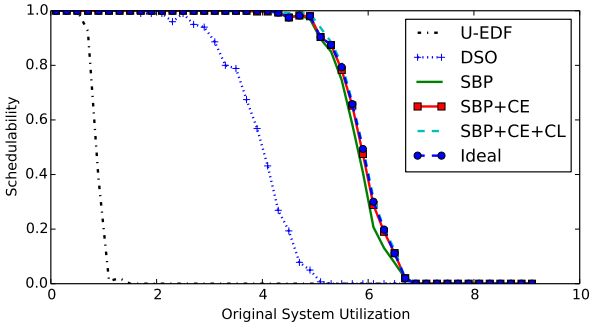
A-Heavy, Short, Mod., Mod., Heavy, Large



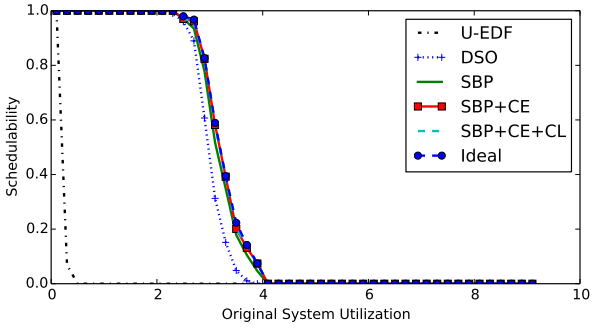
BC-Mod., Long, Light, Heavy, Light, Small



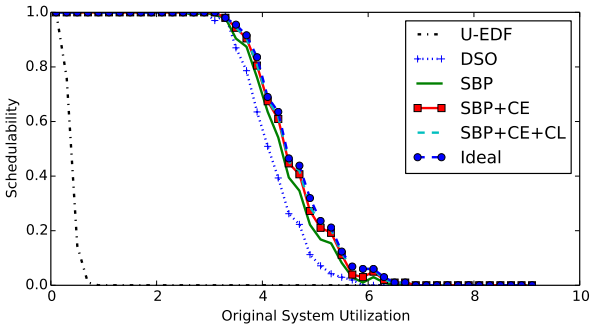
BC-Mod., Short, Light, Light, Light, Small



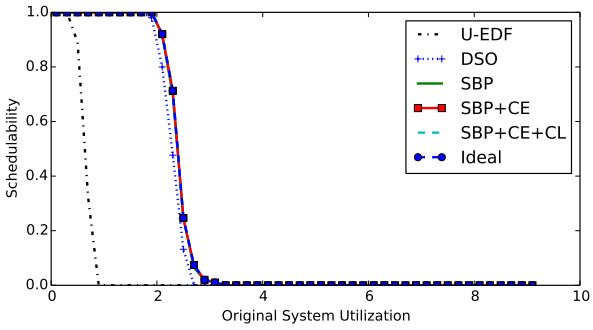
AC-Mod., Long, Heavy, Light, Light, Small



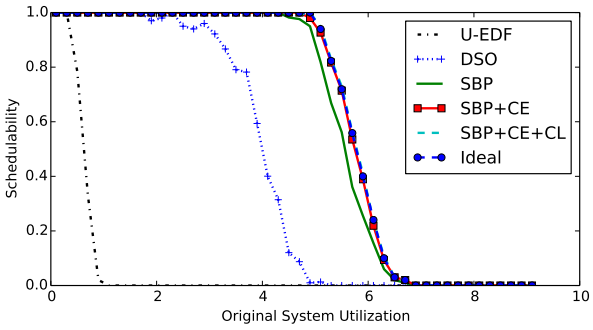
AC-Mod., Cont., Mod., Heavy, Light, Large



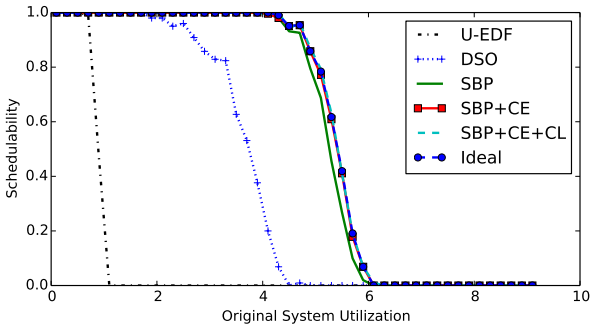
C-Heavy, Cont., Mod., Mod., Heavy, Large



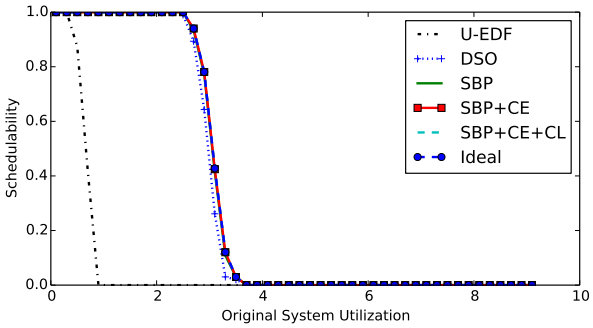
A-Heavy, Long, Light, Light, Light, Large



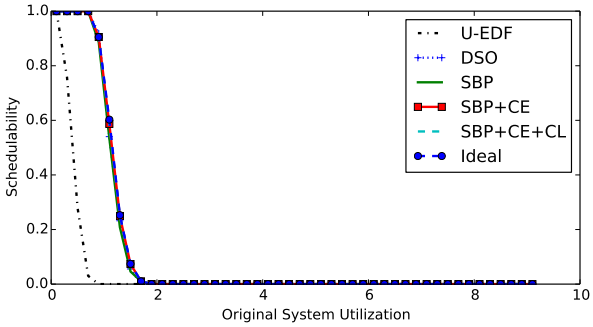
BC-Mod., Cont., Heavy, Light, Light, Small



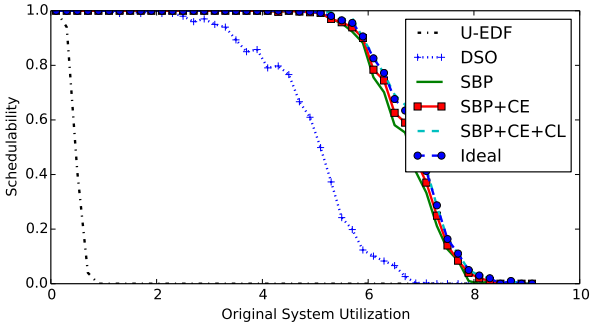
All-Mod., Long, Heavy, Light, Light, Small



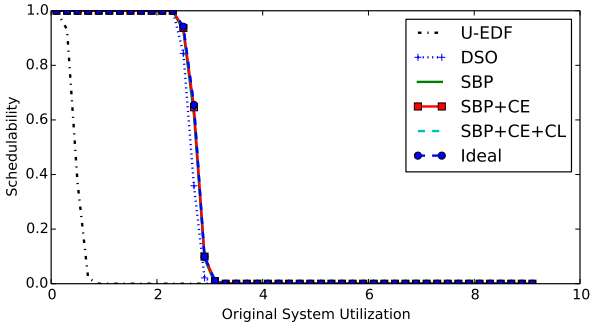
BC-Mod., Long, Light, Light, Heavy, Small



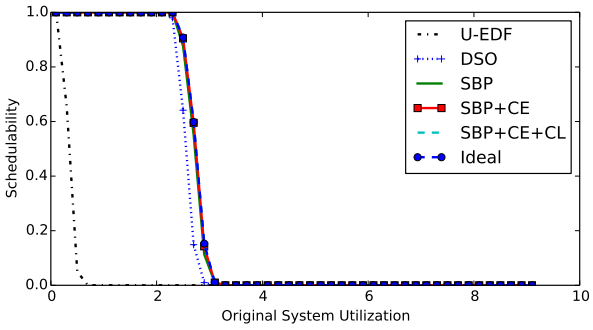
C-Heavy, Cont., Light, Light, Light, Large



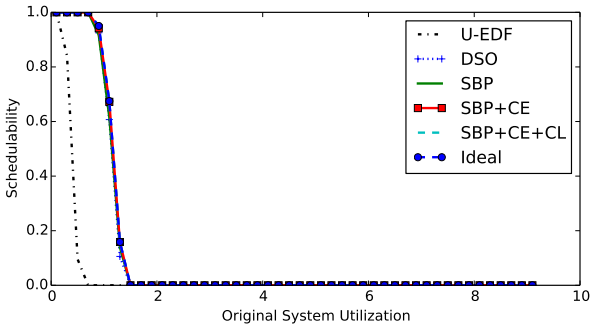
C-Heavy, Cont., Heavy, Mod., Light, Large



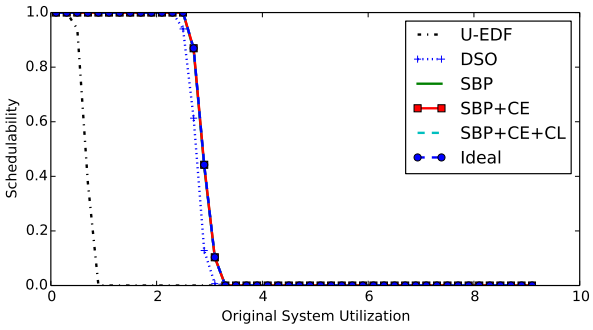
AB-Mod., Long, Light, Mod., Heavy, Small



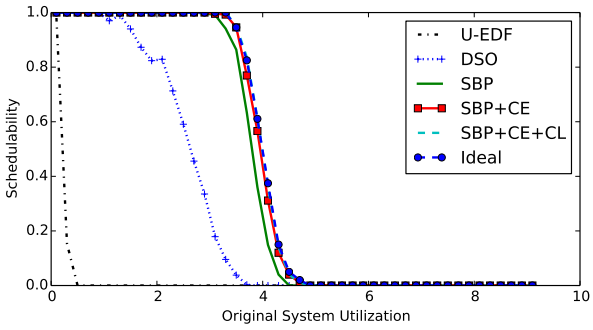
AB-Mod., Long, Light, Heavy, Light, Large



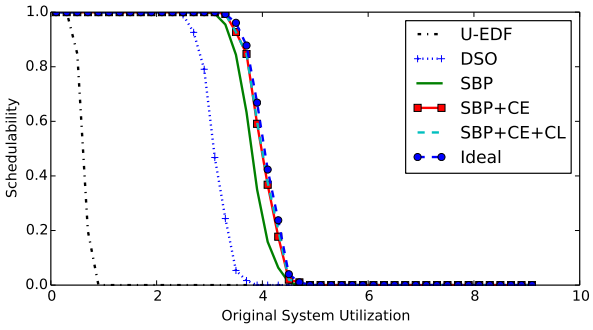
BC-Mod., Short, Light, Mod., Light, Small



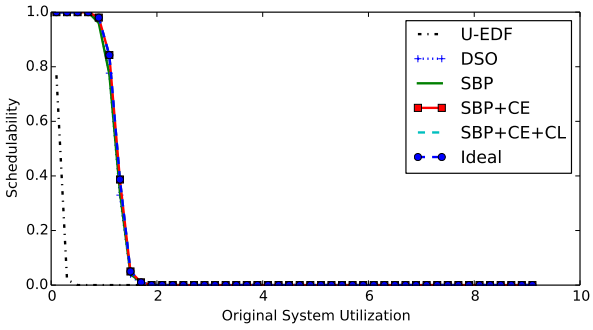
AB-Mod., Long, Light, Light, Heavy, Small



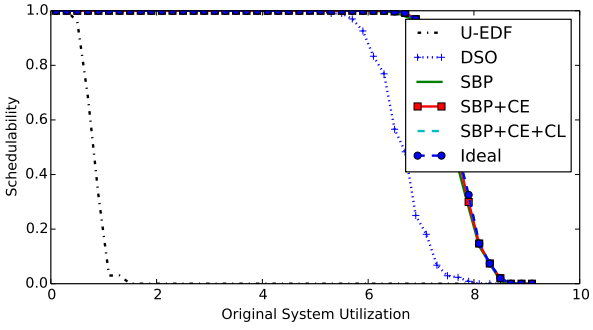
AB-Mod., Cont., Heavy, Heavy, Light, Small



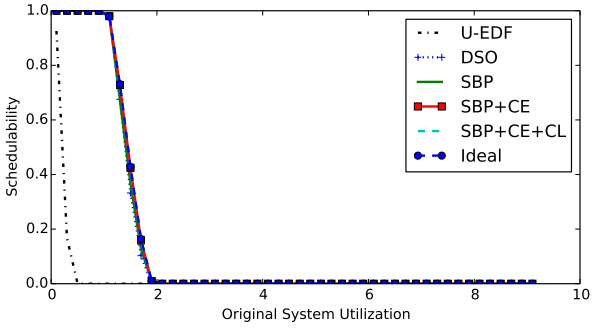
A-Heavy, Long, Mod., Mod., Heavy, Small



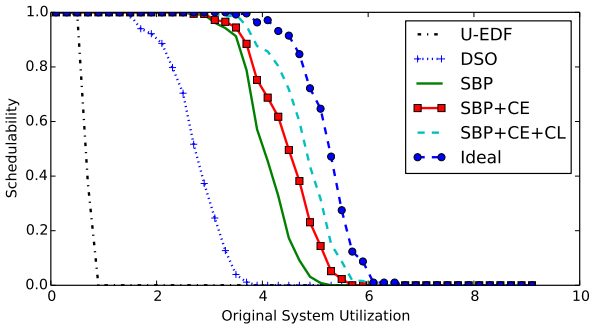
BC-Mod., Cont., Light, Heavy, Light, Small



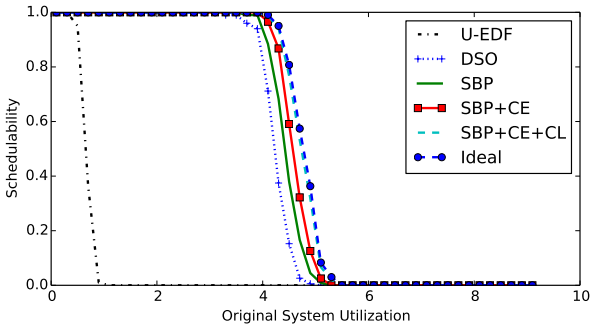
C-Heavy, Long, Mod., Light, Light, Large



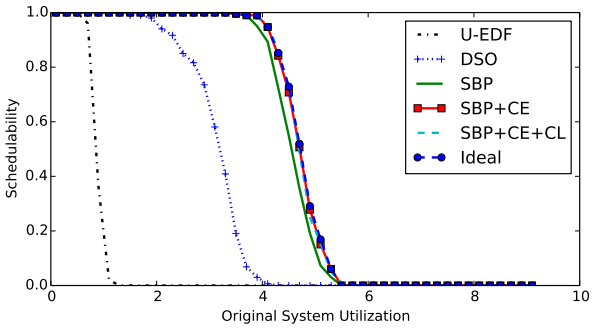
B-Heavy, Cont., Light, Mod., Heavy, Small



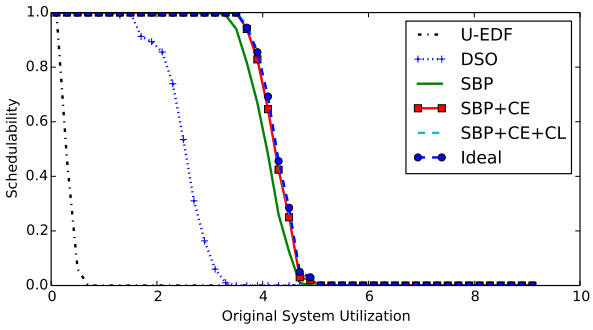
AC-Mod., Long, Heavy, Heavy, Heavy, Small



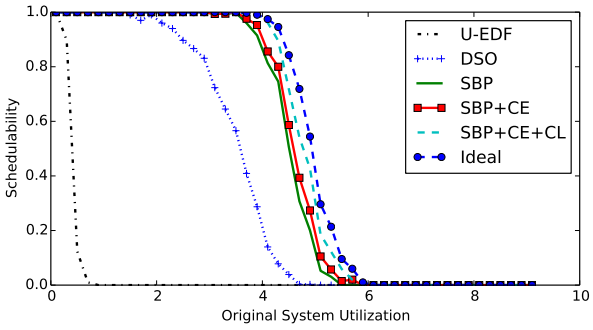
All-Mod., Short, Heavy, Mod., Heavy, Small



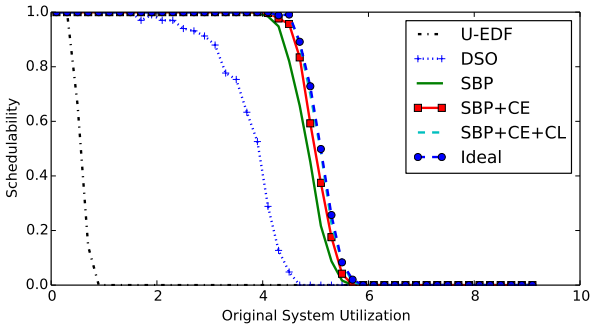
AB-Mod., Long, Heavy, Light, Light, Large



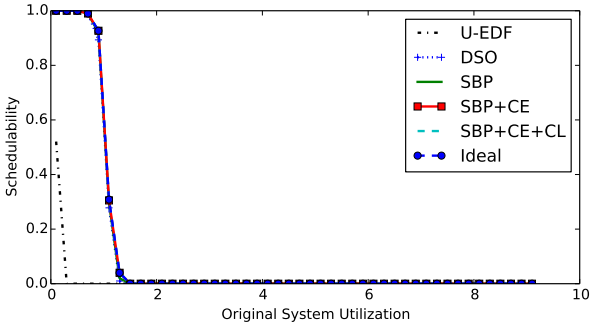
B-Heavy, Cont., Heavy, Mod., Light, Small



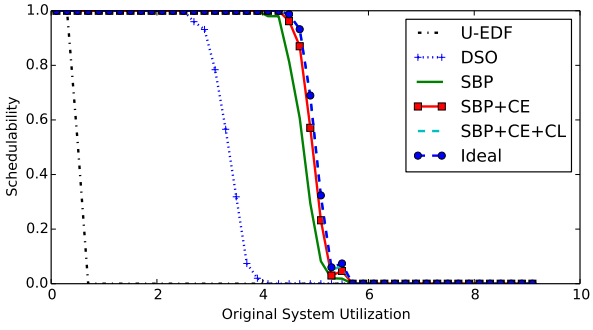
AC-Mod., Cont., Heavy, Mod., Heavy, Large



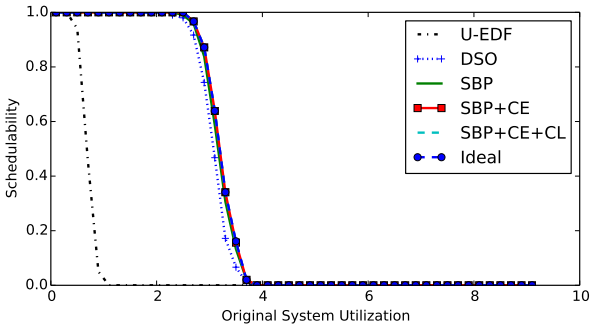
All-Mod., Cont., Heavy, Light, Heavy, Large



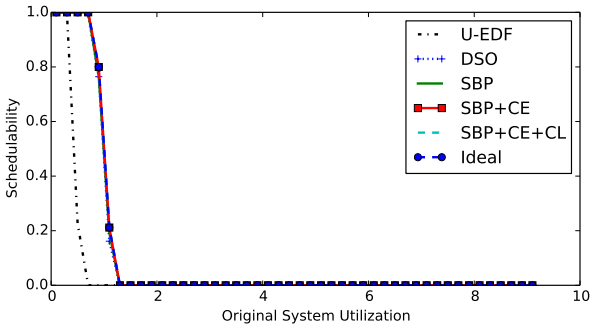
AB-Mod., Cont., Light, Heavy, Light, Small



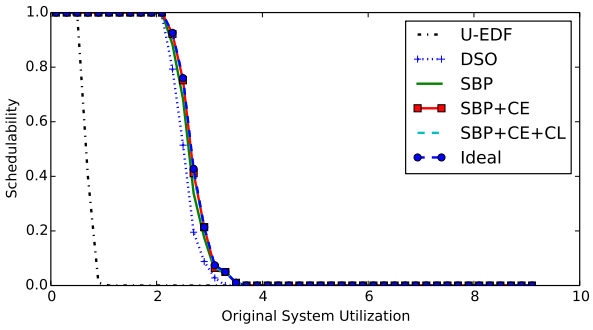
All-Mod., Long, Mod., Heavy, Light, Large



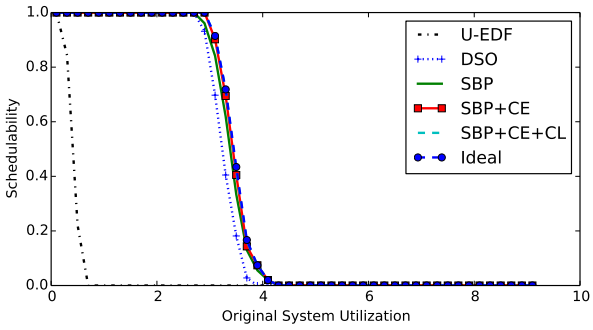
AC-Mod., Short, Mod., Light, Light, Small



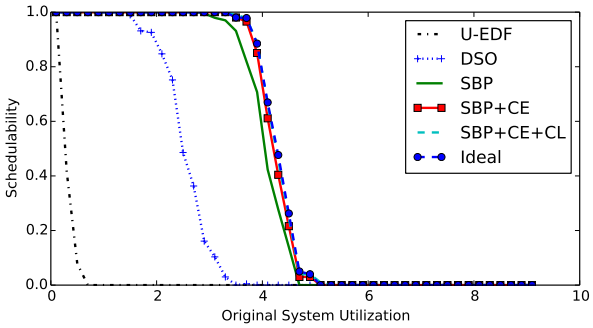
All-Mod., Short, Light, Light, Heavy, Small



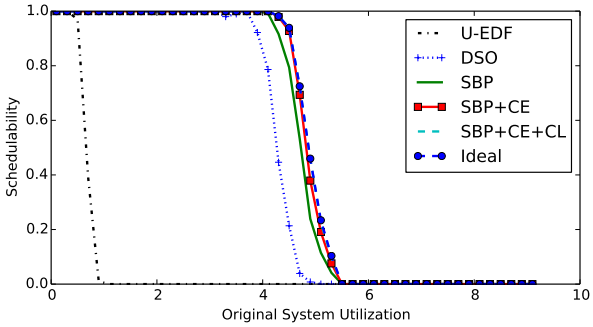
A-Heavy, Short, Mod., Light, Light, Large



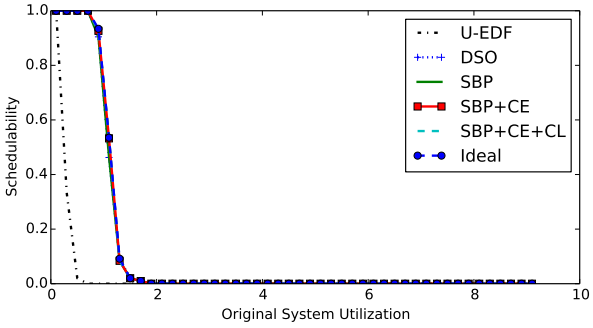
All-Mod., Short, Mod., Heavy, Light, Large



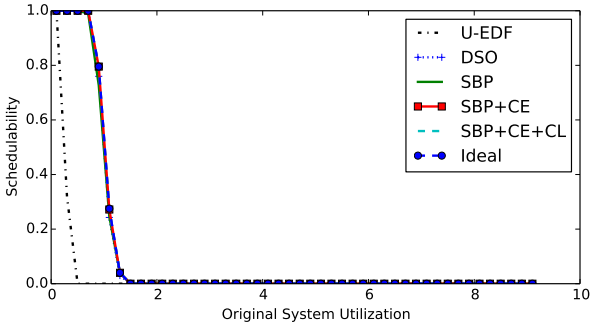
B-Heavy, Cont., Heavy, Mod., Light, Large



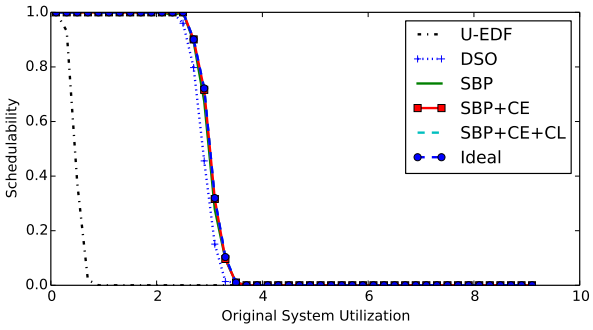
All-Mod., Short, Heavy, Mod., Light, Large



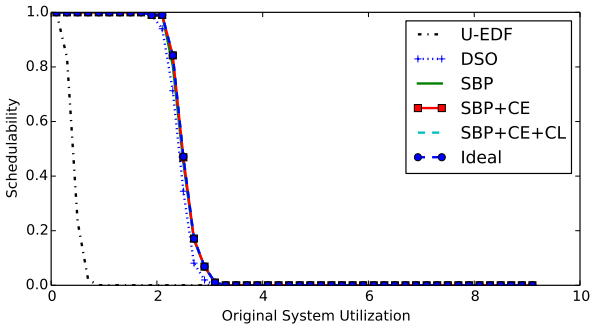
C-Heavy, Short, Light, Heavy, Heavy, Small



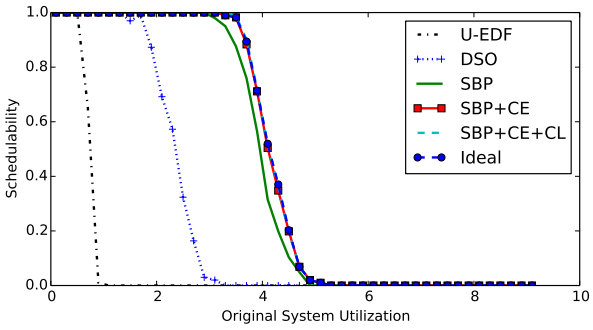
All-Mod., Short, Light, Heavy, Light, Large



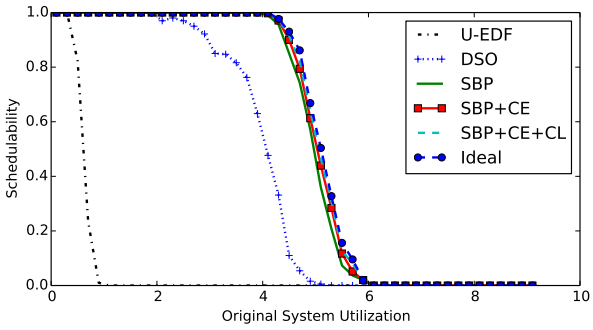
BC-Mod., Long, Light, Mod., Light, Large



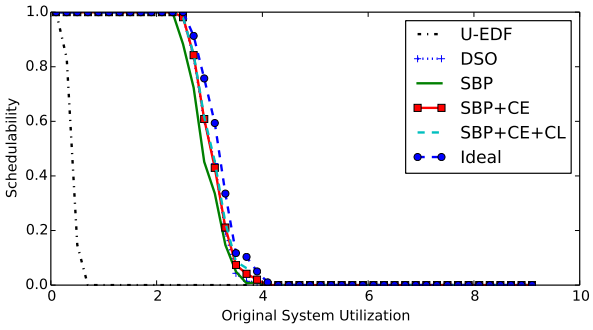
AC-Mod., Long, Light, Mod., Light, Large



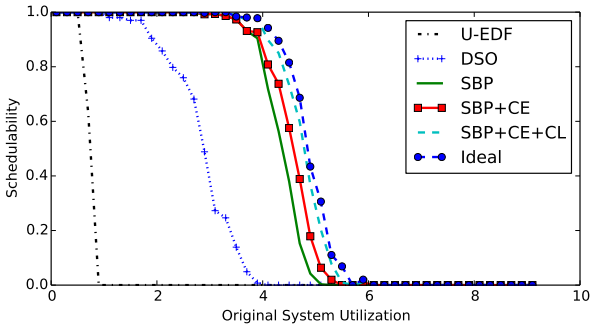
A-Heavy, Long, Heavy, Mod., Light, Small



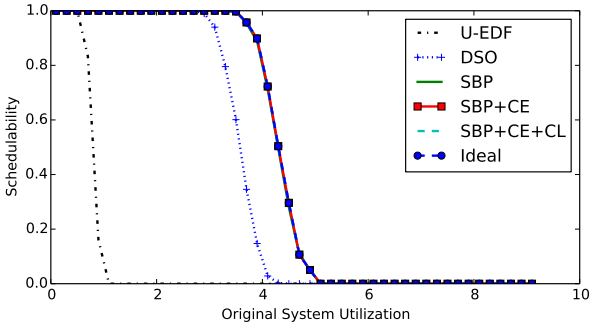
AC-Mod., Cont., Heavy, Light, Heavy, Large



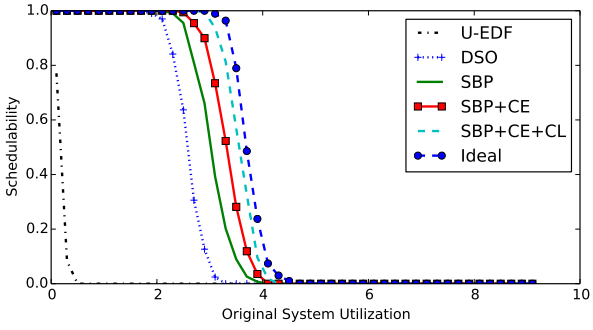
AC-Mod., Short, Mod., Heavy, Heavy, Large



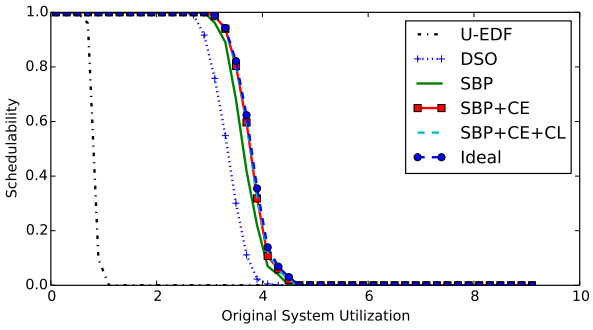
All-Mod., Long, Heavy, Mod., Heavy, Small



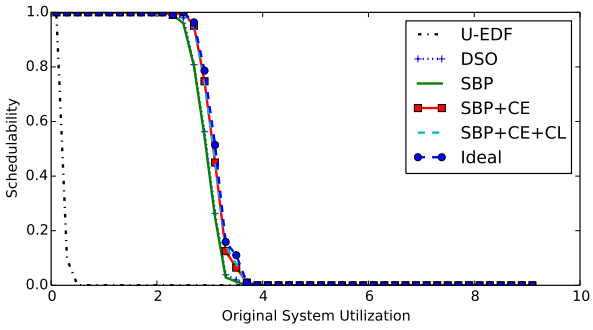
A-Heavy, Long, Mod., Light, Light, Small



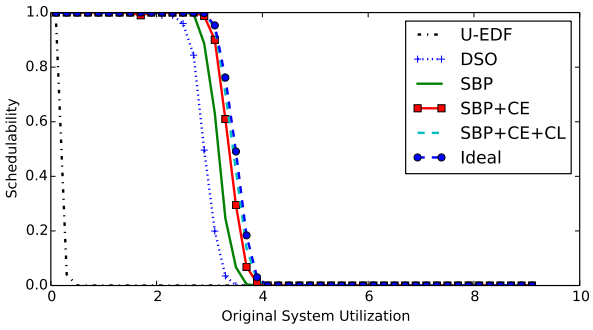
B-Heavy, Cont., Mod., Heavy, Heavy, Large



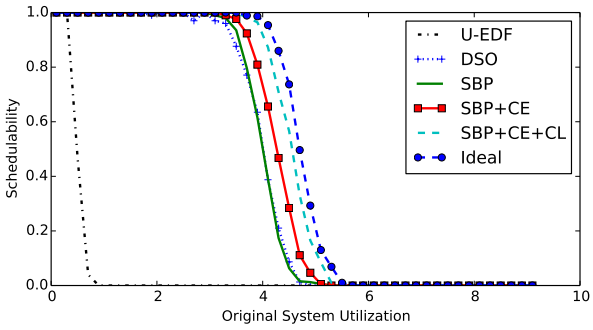
A-Heavy, Short, Heavy, Light, Heavy, Small



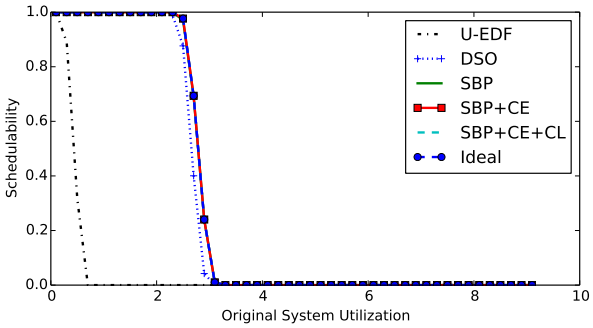
AC-Mod., Cont., Mod., Heavy, Heavy, Small



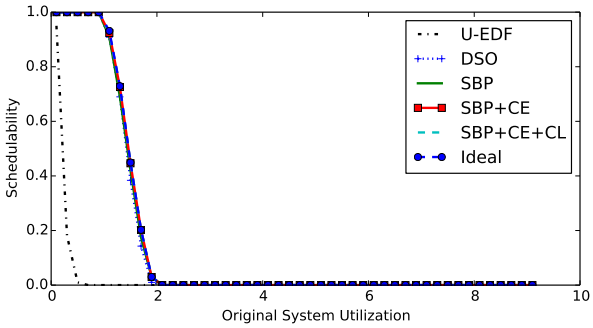
All-Mod., Cont., Mod., Heavy, Heavy, Small



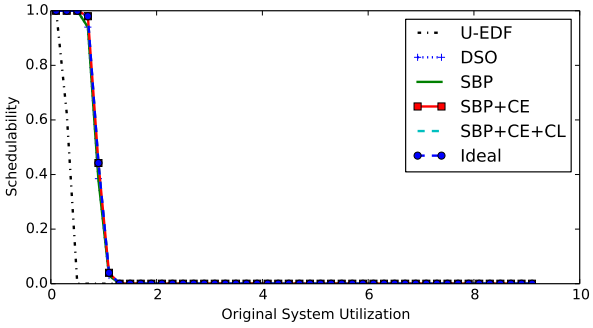
AC-Mod., Short, Heavy, Heavy, Heavy, Small



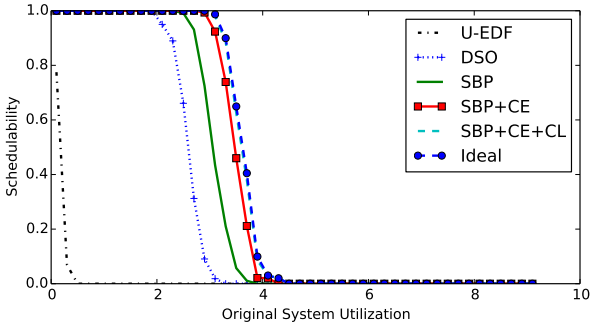
All-Mod., Long, Light, Mod., Light, Small



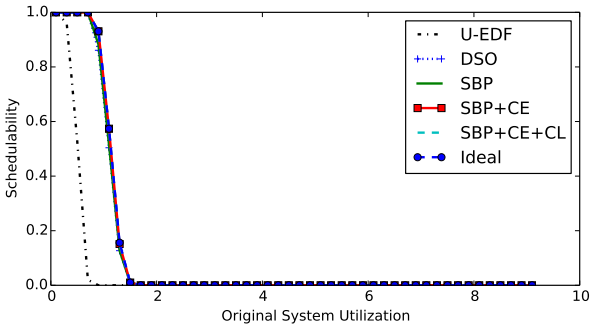
B-Heavy, Cont., Light, Mod., Light, Large



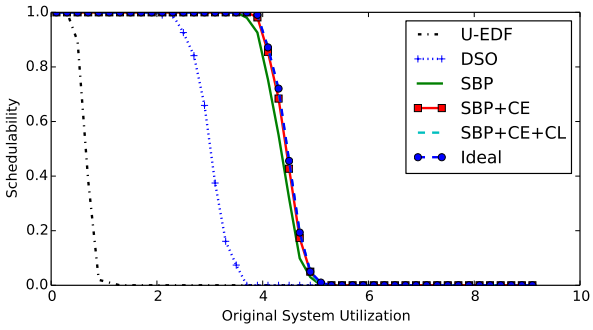
AC-Mod., Short, Light, Mod., Heavy, Small



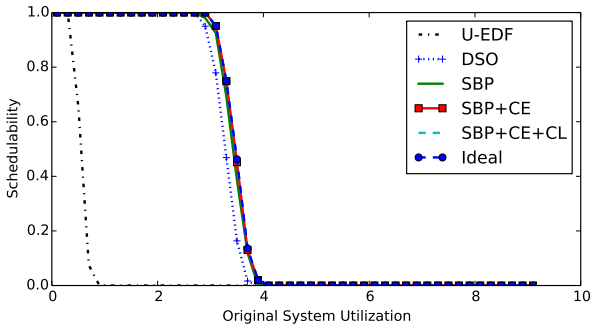
B-Heavy, Cont., Mod., Heavy, Heavy, Small



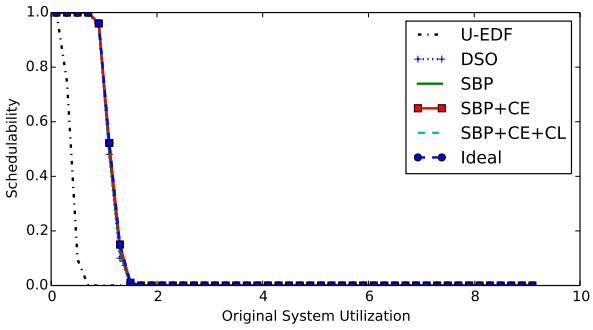
C-Heavy, Short, Light, Light, Light, Small



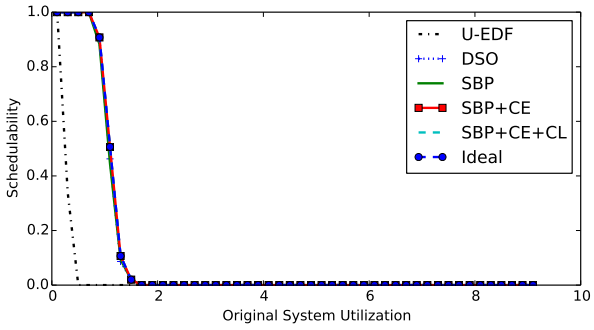
B-Heavy, Long, Mod., Mod., Light, Large



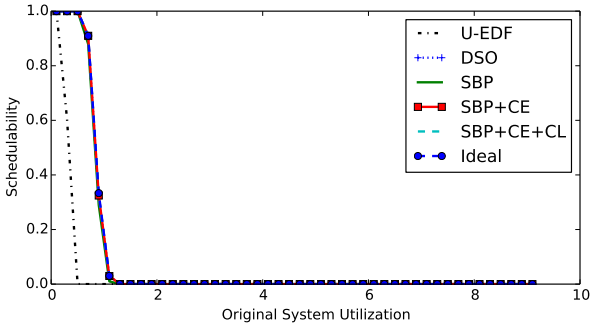
All-Mod., Short, Mod., Mod., Light, Small



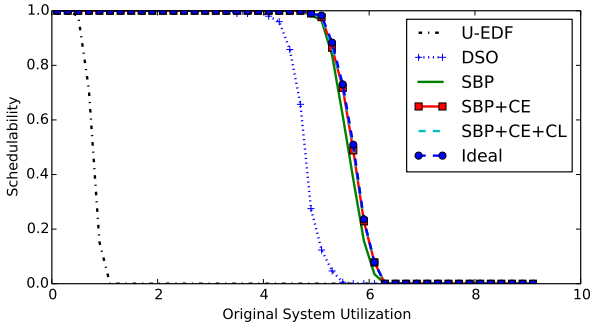
C-Heavy, Short, Light, Mod., Light, Small



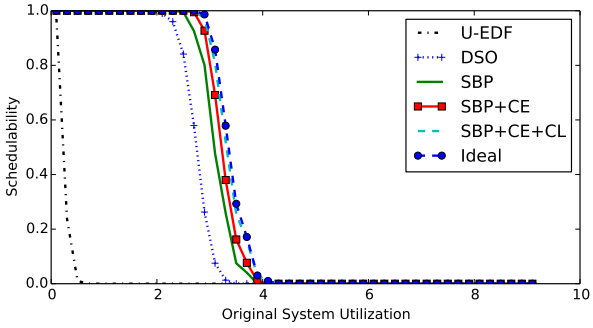
C-Heavy, Short, Light, Heavy, Light, Small



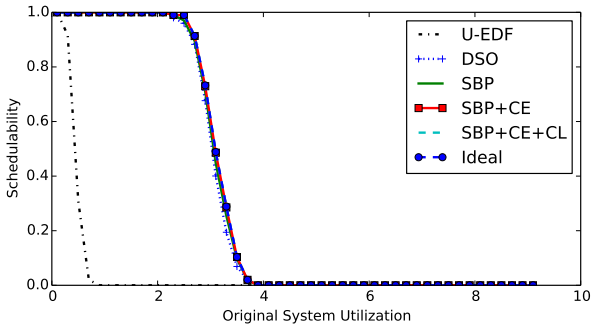
A-Heavy, Short, Light, Mod., Light, Large



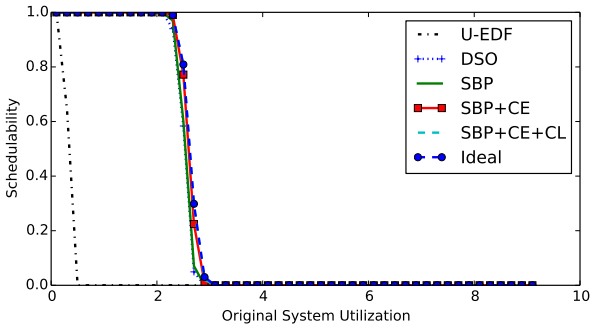
AC-Mod., Long, Mod., Light, Heavy, Small



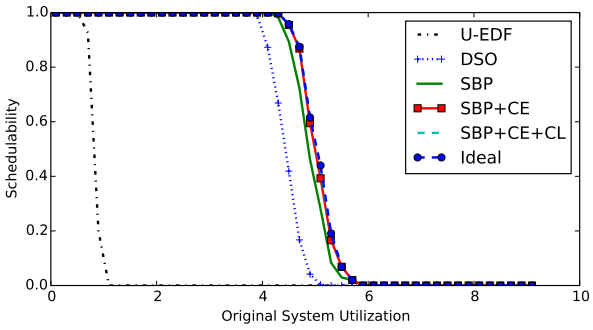
AB-Mod., Cont., Mod., Mod., Heavy, Large



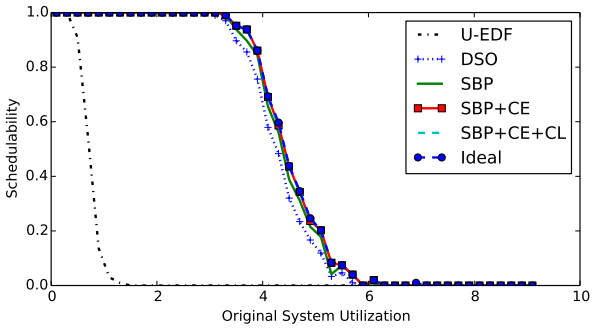
C-Heavy, Long, Light, Mod., Heavy, Small



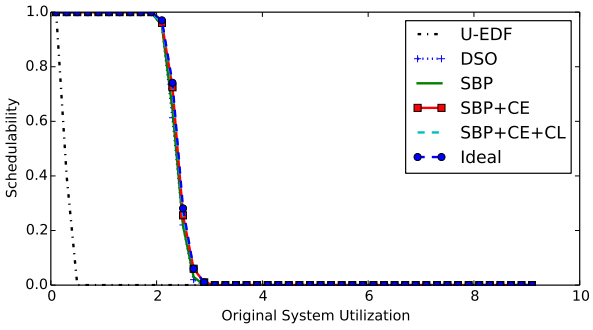
AB-Mod., Long, Light, Heavy, Heavy, Small



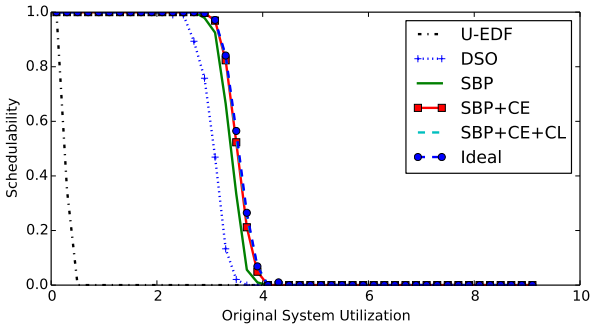
All-Mod., Short, Heavy, Light, Light, Large



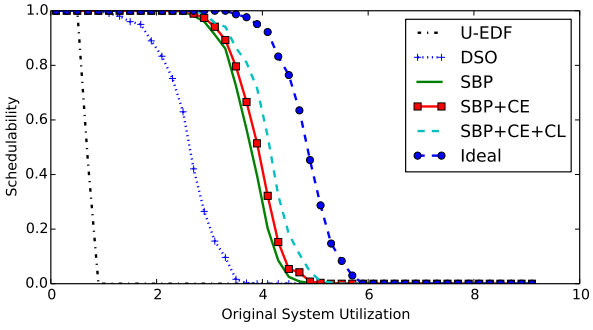
C-Heavy, Short, Mod., Light, Light, Large



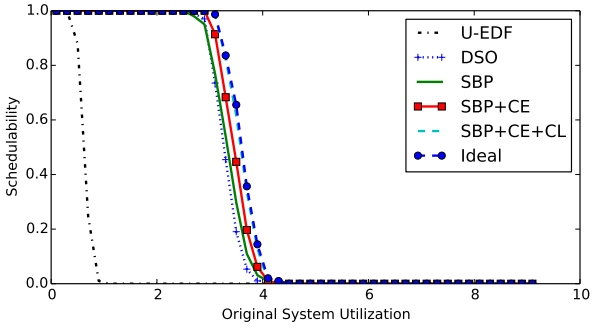
AC-Mod., Long, Light, Heavy, Heavy, Small



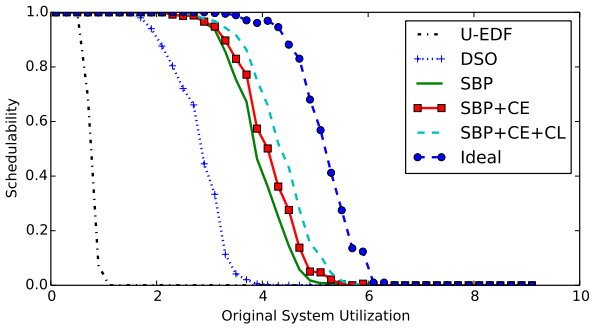
All-Mod., Cont., Mod., Mod., Heavy, Small



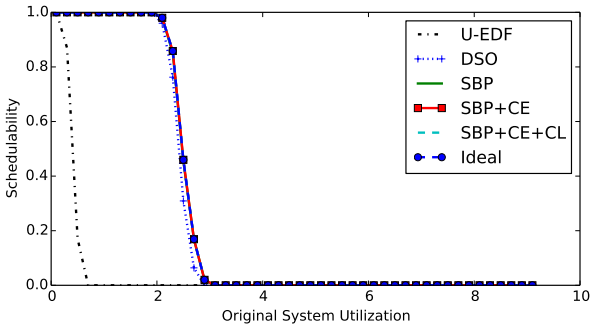
All-Mod., Long, Heavy, Heavy, Heavy, Large



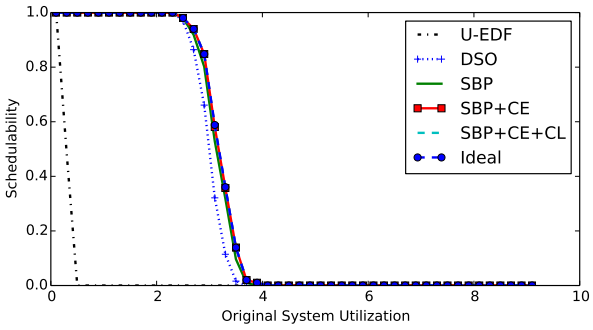
B-Heavy, Short, Mod., Mod., Heavy, Large



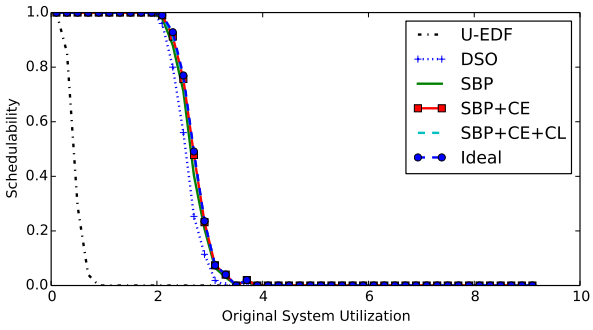
BC-Mod., Long, Heavy, Heavy, Heavy, Large



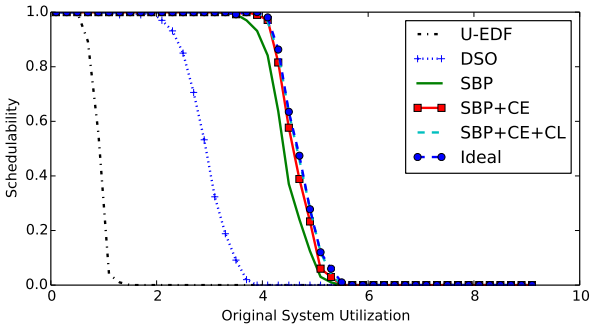
AC-Mod., Long, Light, Mod., Light, Small



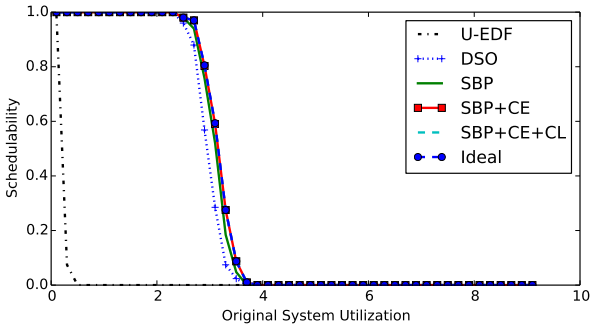
AC-Mod., Cont., Mod., Mod., Light, Small



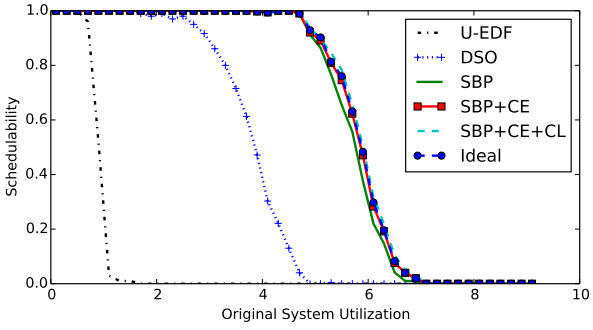
A-Heavy, Cont., Mod., Light, Light, Large



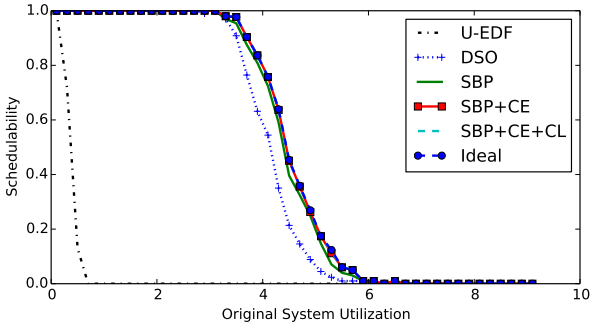
B-Heavy, Long, Heavy, Light, Heavy, Small



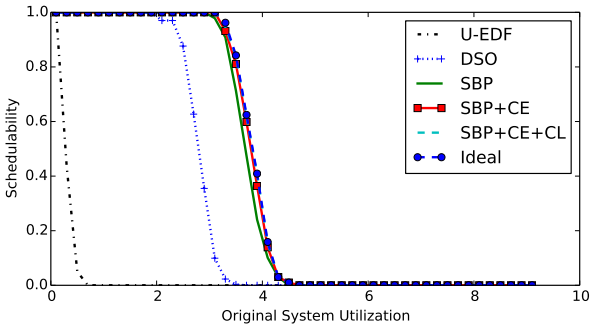
AC-Mod., Cont., Mod., Heavy, Light, Small



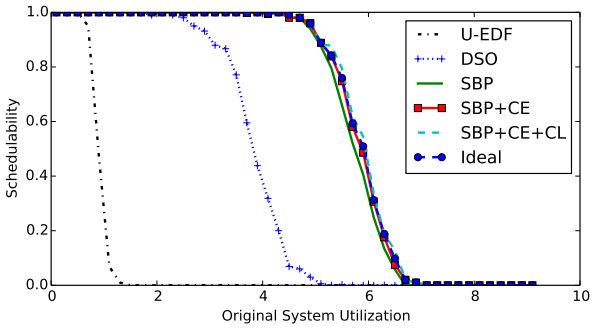
BC-Mod., Long, Heavy, Light, Light, Small



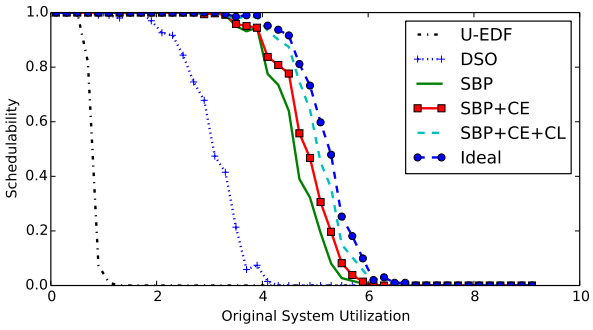
C-Heavy, Cont., Mod., Mod., Light, Small



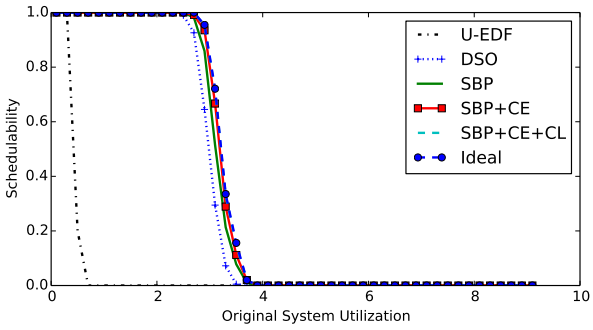
B-Heavy, Cont., Mod., Mod., Light, Large



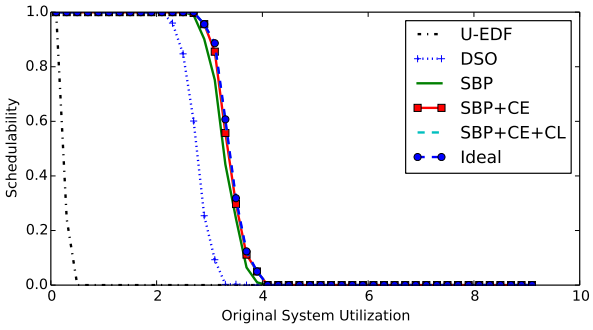
BC-Mod., Long, Heavy, Light, Light, Large



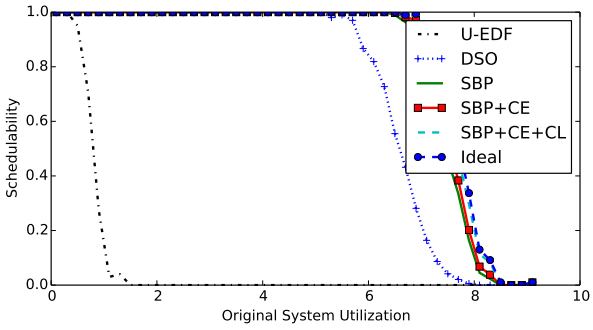
BC-Mod., Long, Heavy, Mod., Heavy, Small



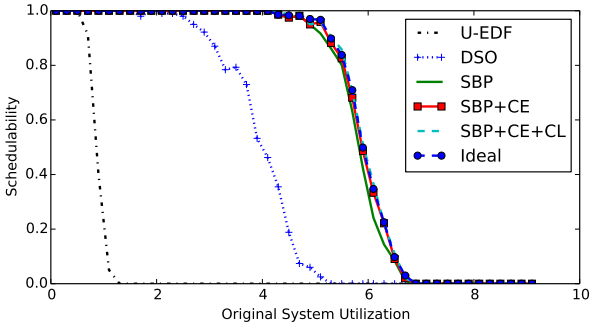
AB-Mod., Short, Mod., Heavy, Light, Large



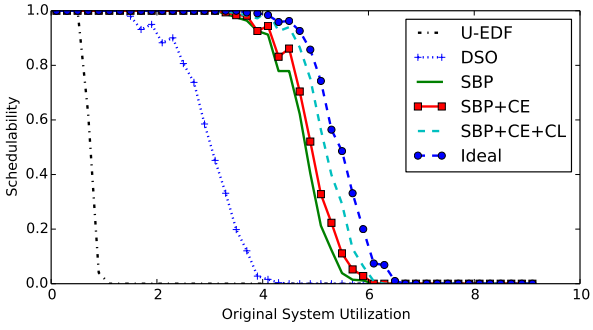
AB-Mod., Cont., Mod., Mod., Light, Large



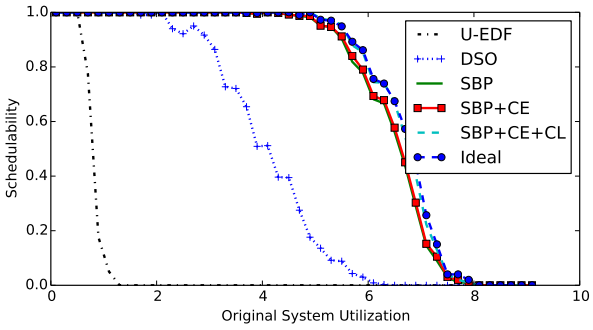
C-Heavy, Long, Mod., Light, Heavy, Large



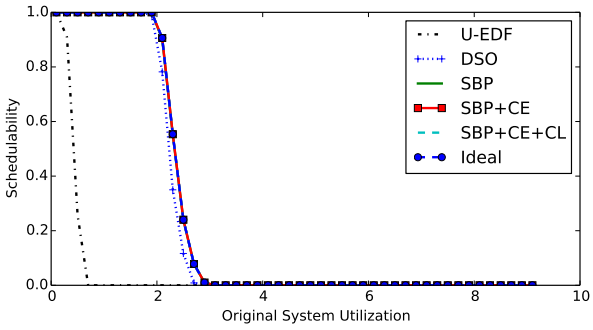
AC-Mod., Long, Heavy, Light, Light, Large



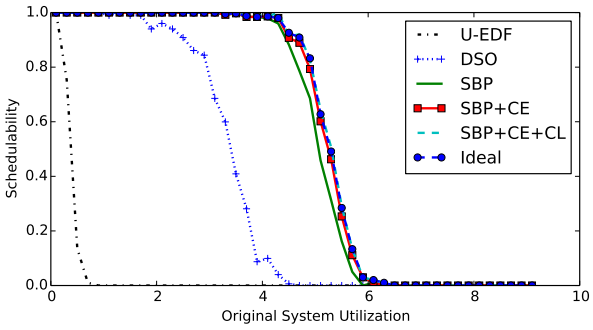
AC-Mod., Long, Heavy, Mod., Heavy, Large



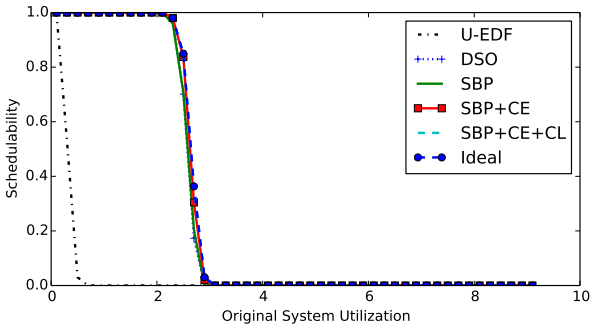
C-Heavy, Long, Heavy, Heavy, Light, Large



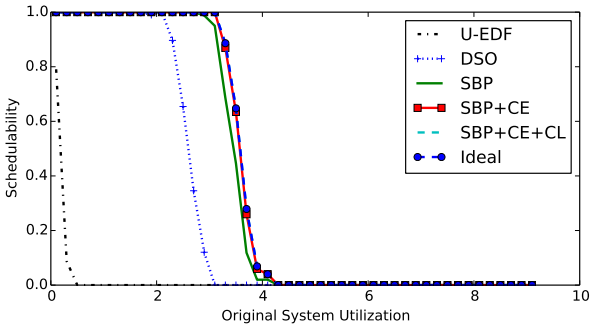
A-Heavy, Long, Light, Mod., Light, Small



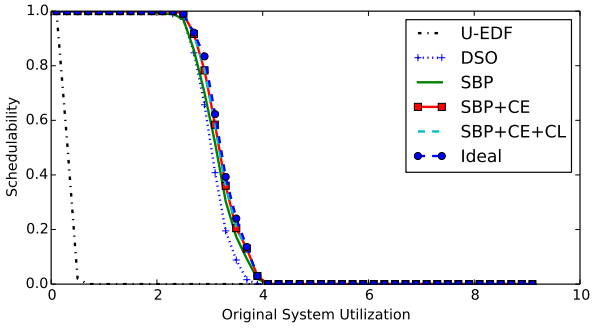
BC-Mod., Cont., Heavy, Mod., Light, Small



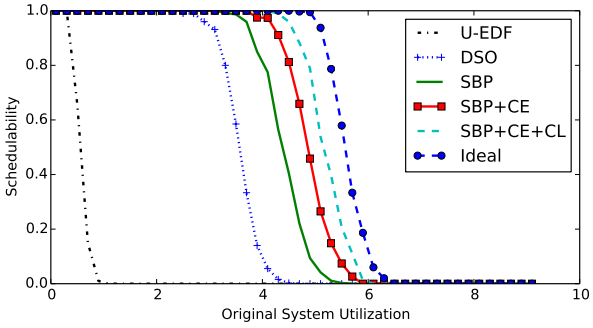
All-Mod., Long, Light, Heavy, Heavy, Small



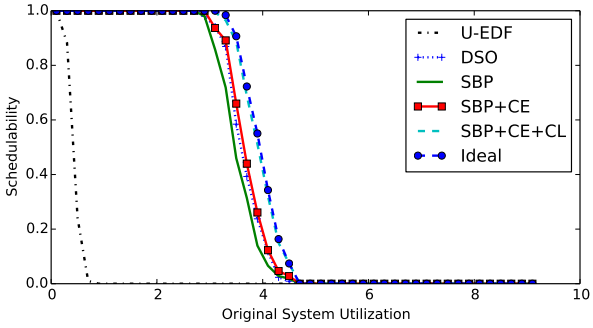
B-Heavy, Cont., Mod., Heavy, Light, Small



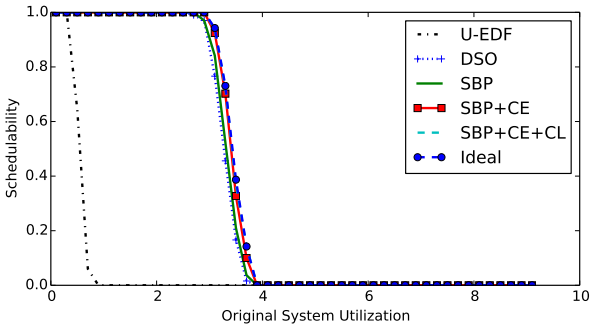
AC-Mod., Cont., Mod., Mod., Heavy, Large



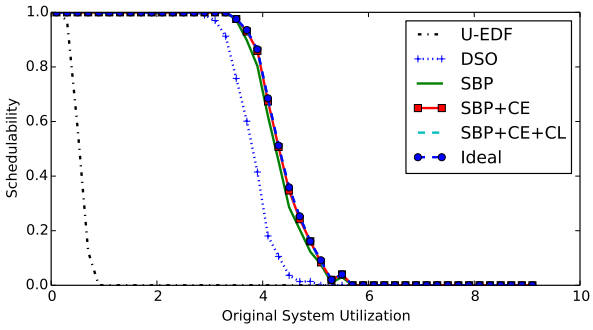
BC-Mod., Long, Mod., Heavy, Heavy, Large



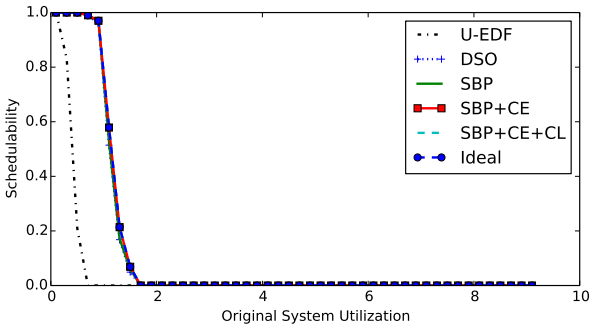
BC-Mod., Short, Mod., Heavy, Heavy, Large



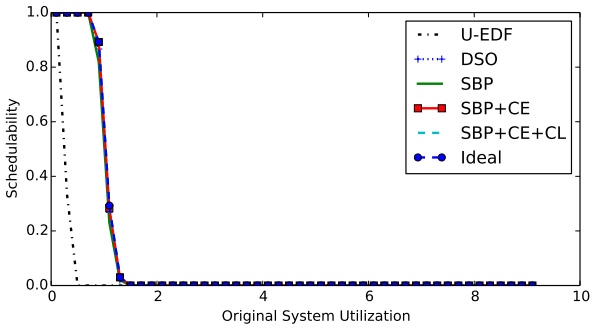
All-Mod., Short, Mod., Mod., Heavy, Small



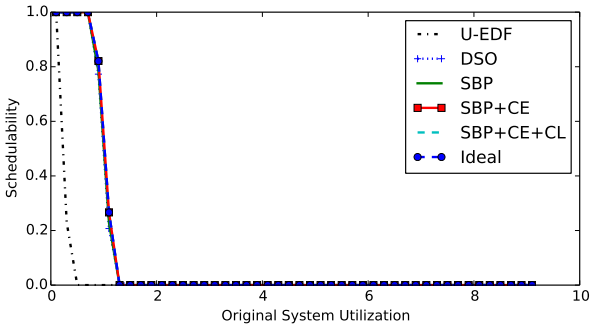
BC-Mod., Cont., Mod., Light, Light, Large



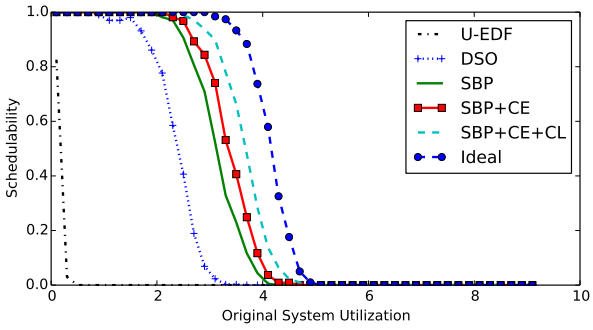
C-Heavy, Cont., Light, Light, Light, Small



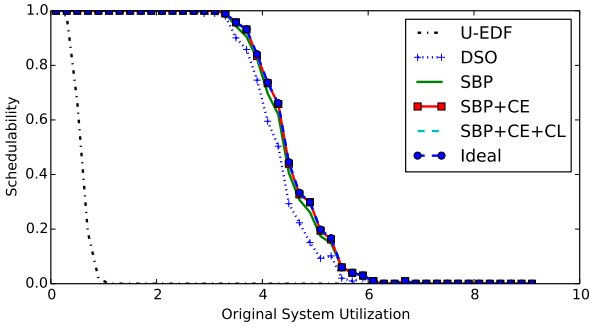
AB-Mod., Short, Light, Heavy, Heavy, Small



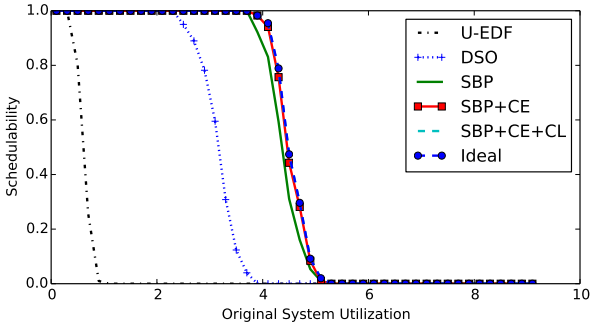
All-Mod., Short, Light, Heavy, Light, Small



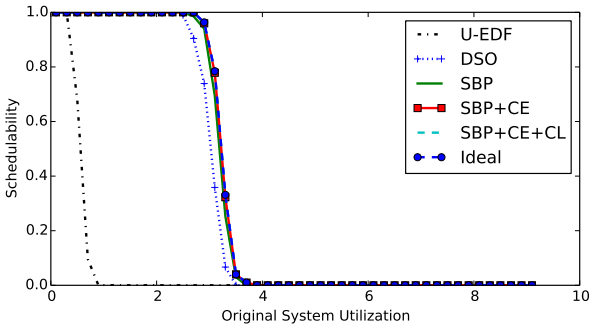
B-Heavy, Cont., Heavy, Heavy, Heavy, Large



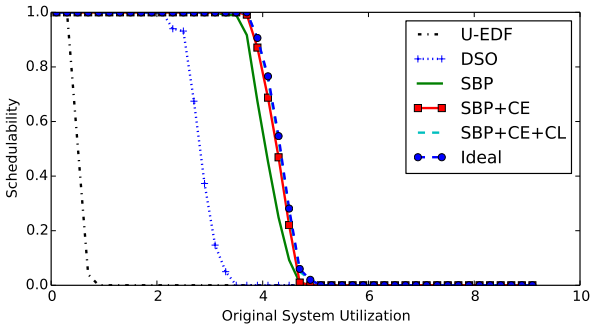
C-Heavy, Cont., Mod., Light, Heavy, Small



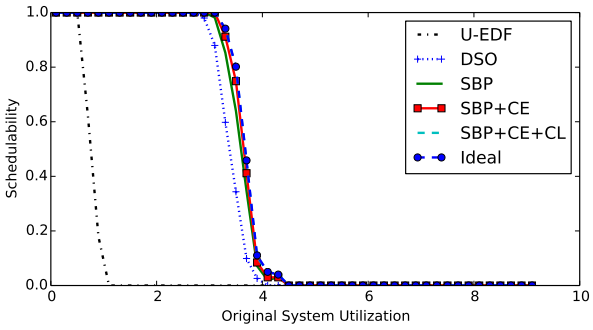
AB-Mod., Long, Mod., Mod., Light, Large



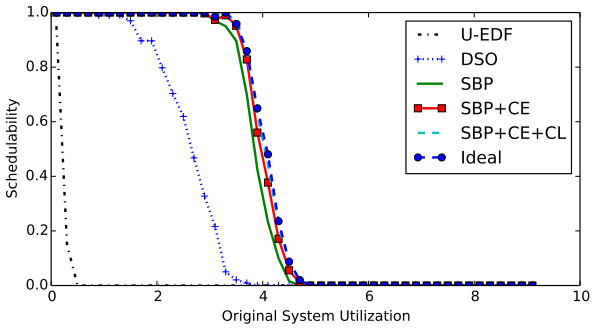
AB-Mod., Short, Mod., Mod., Light, Small



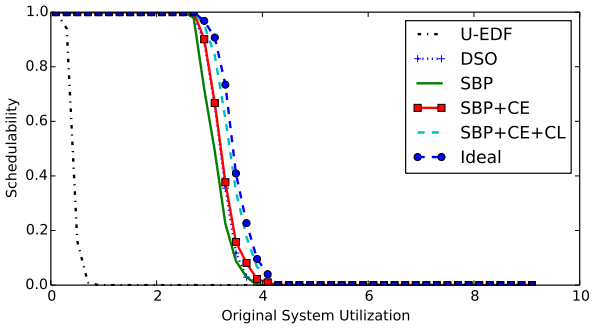
AB-Mod., Long, Mod., Heavy, Light, Large



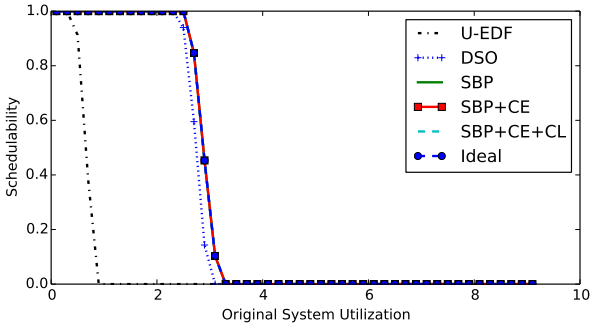
B-Heavy, Short, Mod., Light, Heavy, Large



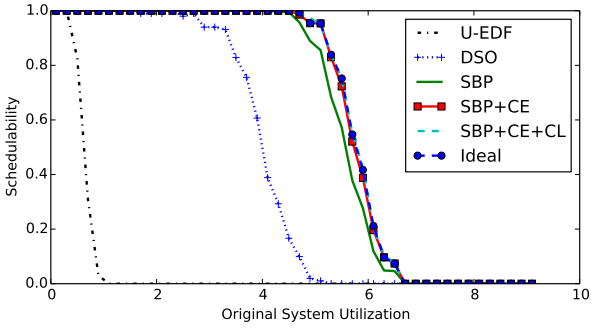
AB-Mod., Cont., Heavy, Heavy, Light, Large



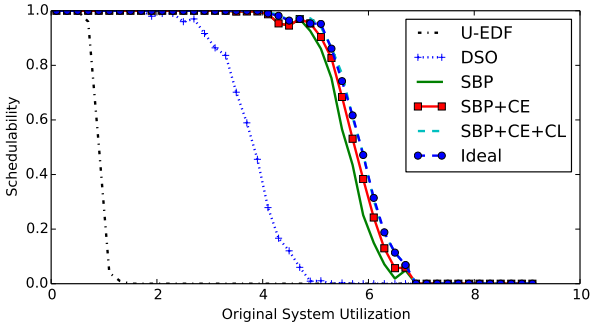
All-Mod., Short, Mod., Heavy, Heavy, Large



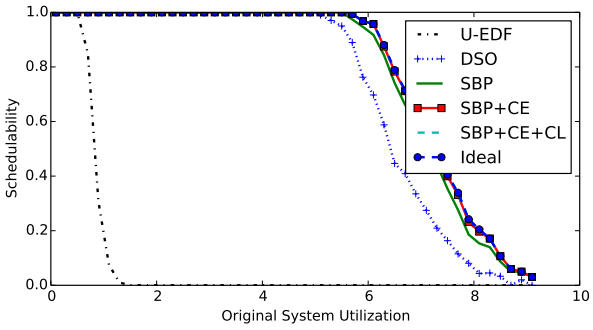
AB-Mod., Long, Light, Light, Light, Small



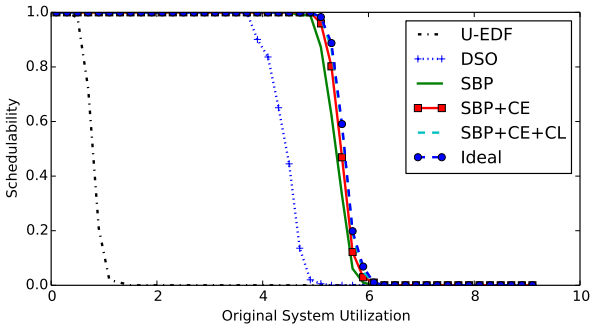
BC-Mod., Cont., Heavy, Light, Light, Large



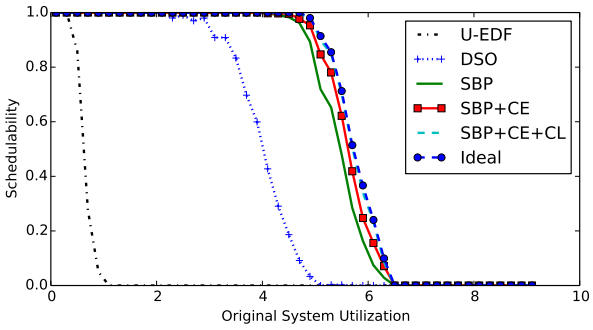
BC-Mod., Long, Heavy, Light, Heavy, Small



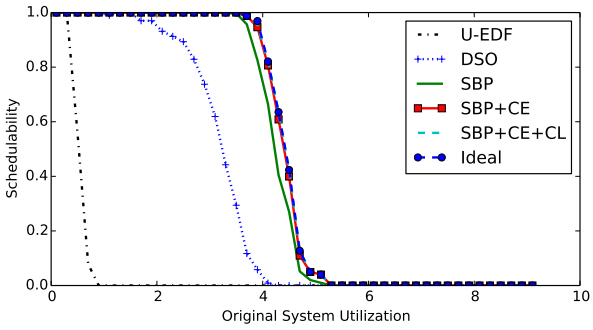
C-Heavy, Short, Heavy, Light, Light, Large



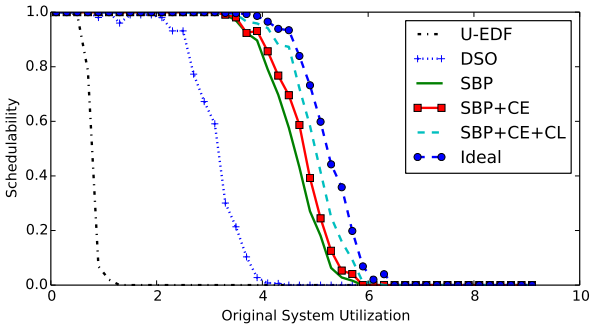
All-Mod., Long, Mod., Light, Heavy, Large



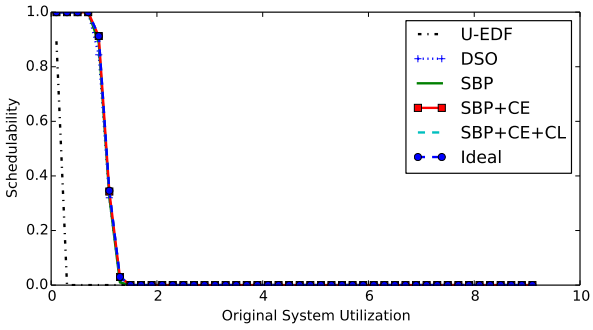
BC-Mod., Cont., Heavy, Light, Heavy, Large



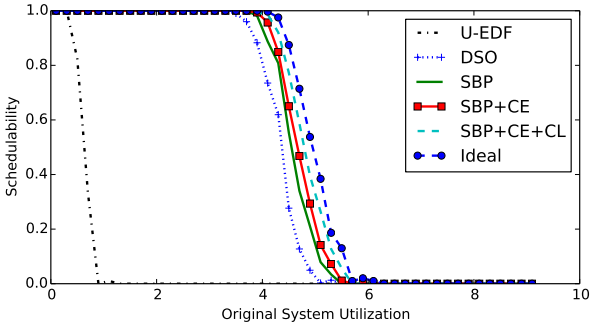
AB-Mod., Cont., Heavy, Light, Light, Large



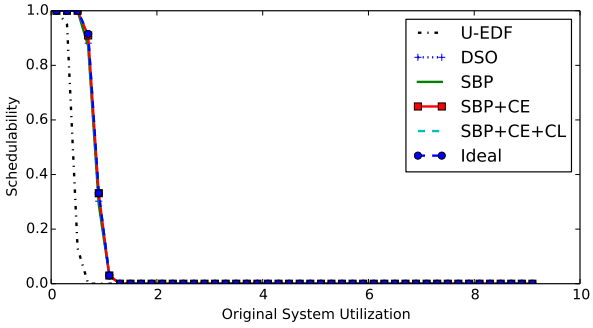
BC-Mod., Long, Heavy, Mod., Heavy, Large



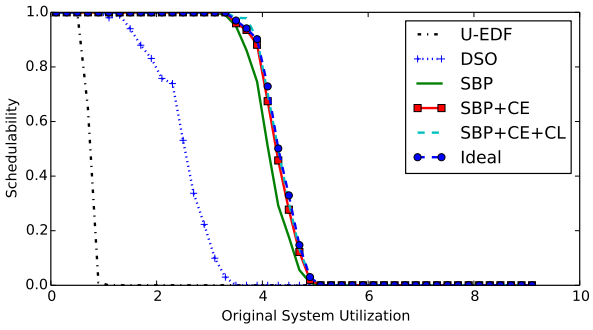
AB-Mod., Cont., Light, Mod., Light, Small



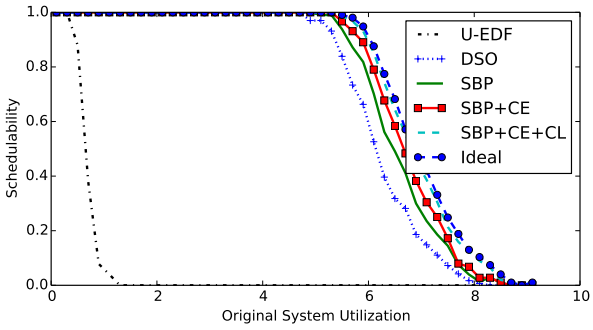
AC-Mod., Short, Heavy, Mod., Heavy, Large



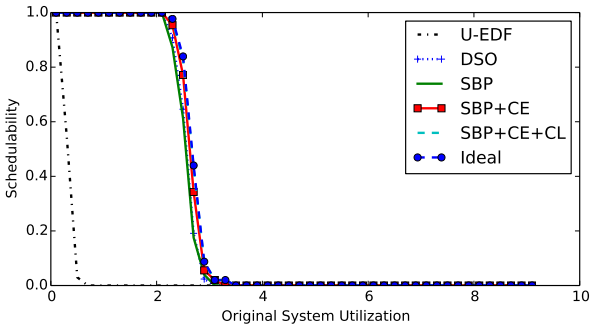
A-Heavy, Short, Light, Light, Light, Large



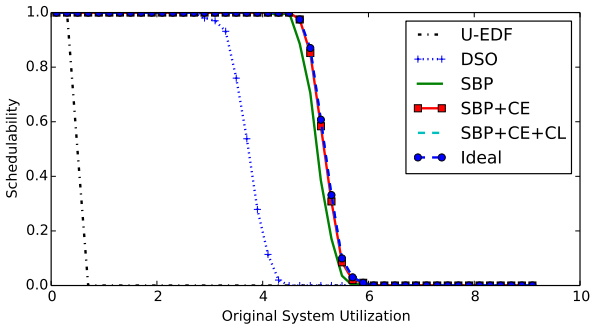
AB-Mod., Long, Heavy, Mod., Light, Large



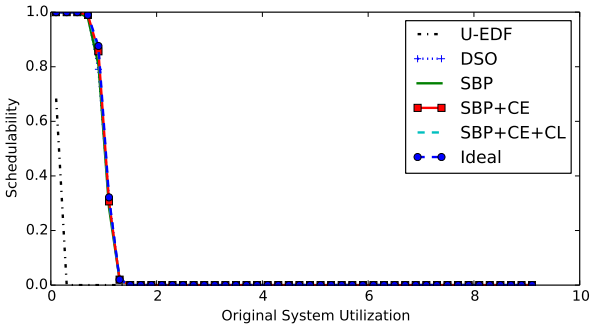
C-Heavy, Short, Heavy, Mod., Heavy, Small



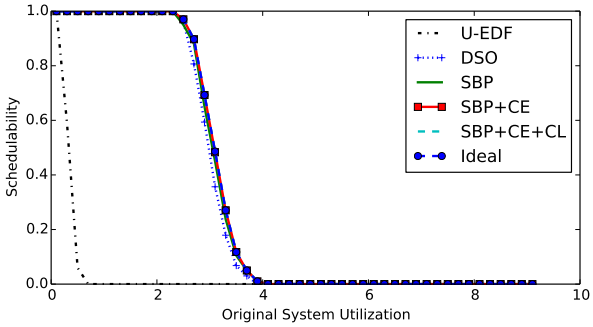
All-Mod., Long, Light, Heavy, Heavy, Large



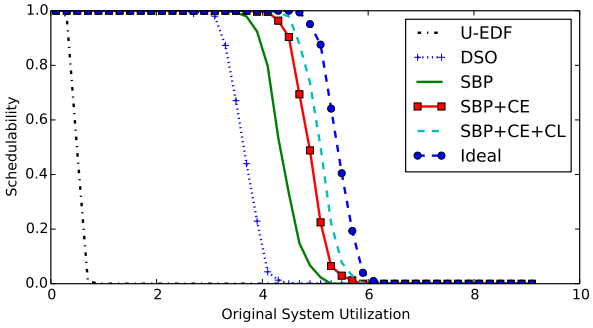
AC-Mod., Long, Mod., Heavy, Light, Small



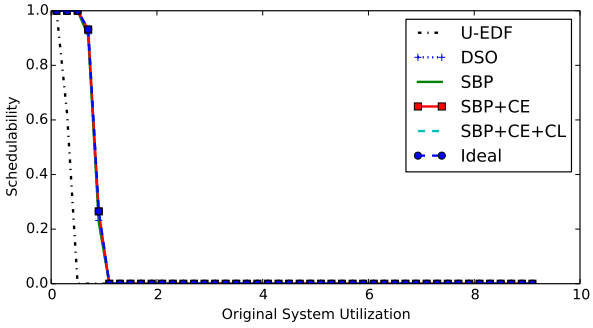
All-Mod., Cont., Light, Heavy, Heavy, Large



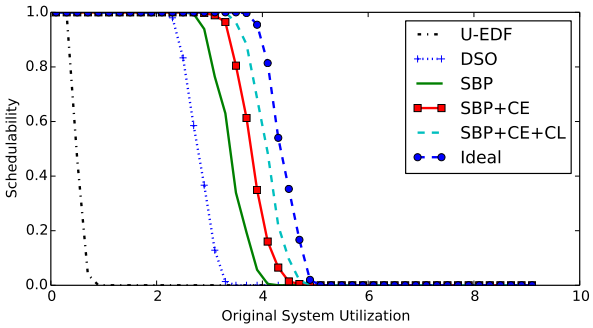
C-Heavy, Long, Light, Heavy, Light, Large



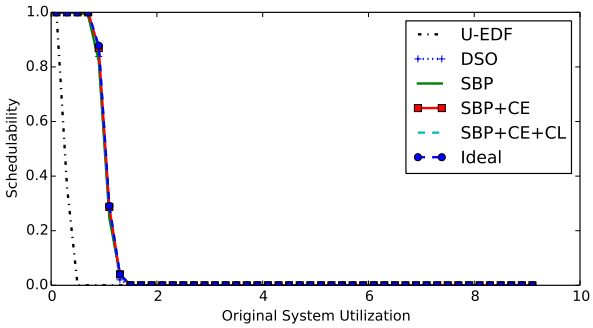
AC-Mod., Long, Mod., Heavy, Heavy, Large



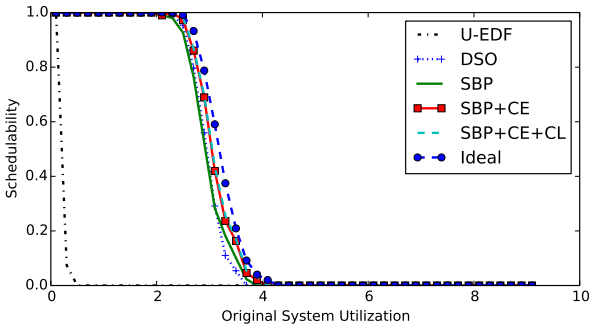
A-Heavy, Short, Light, Mod., Heavy, Small



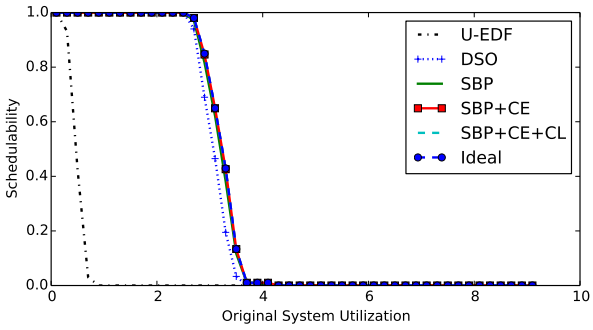
AB-Mod., Long, Mod., Heavy, Heavy, Large



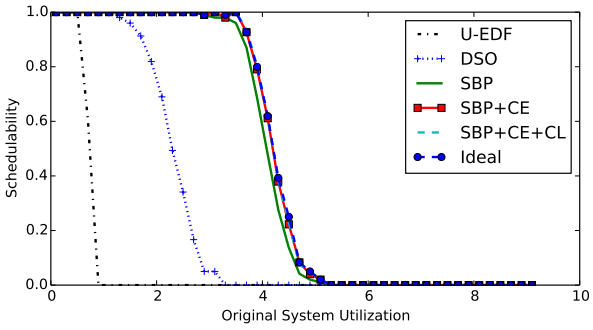
AB-Mod., Short, Light, Heavy, Light, Large



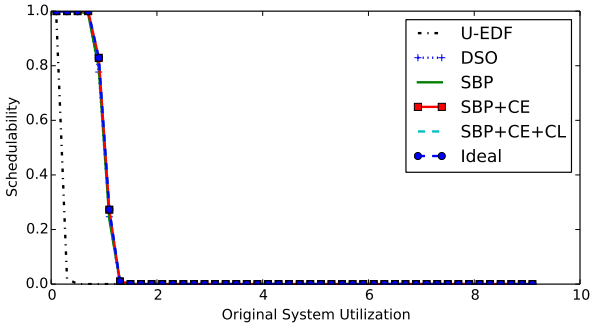
AC-Mod., Cont., Mod., Heavy, Heavy, Large



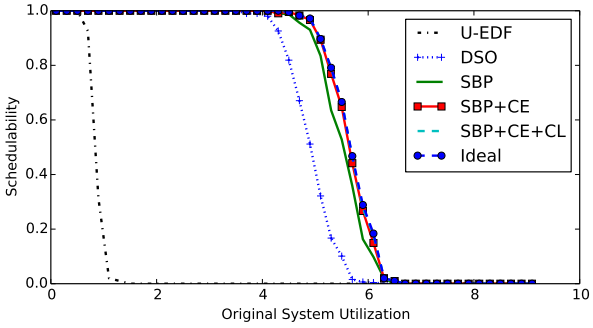
AC-Mod., Cont., Mod., Light, Light, Small



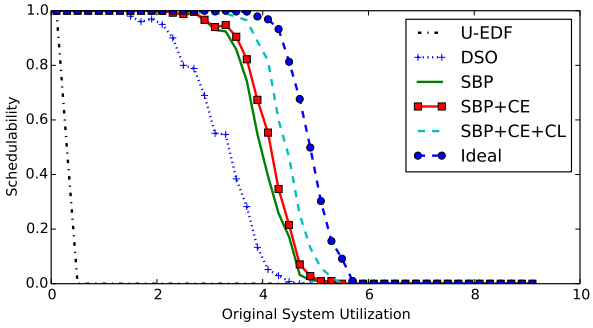
A-Heavy, Long, Heavy, Mod., Light, Large



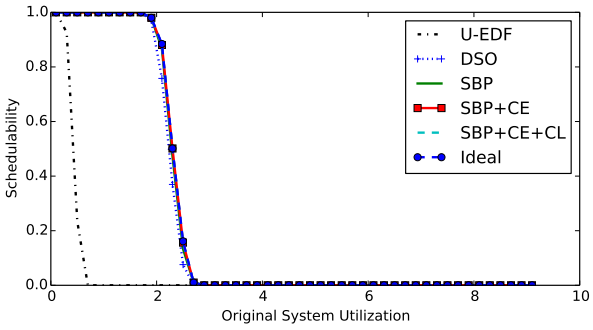
All-Mod., Cont., Light, Mod., Heavy, Small



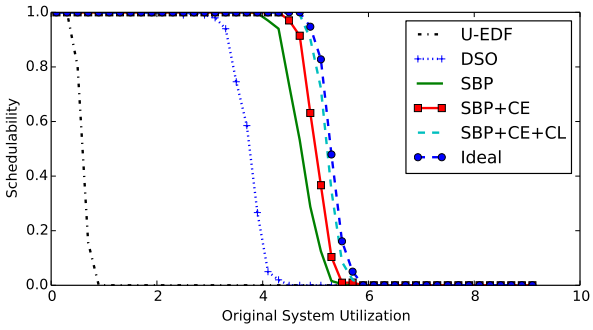
BC-Mod., Short, Heavy, Light, Light, Large



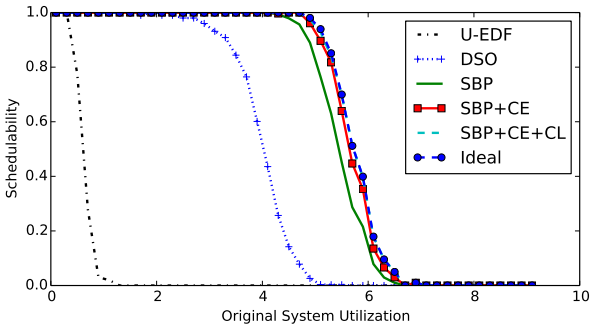
AC-Mod., Cont., Heavy, Heavy, Heavy, Large



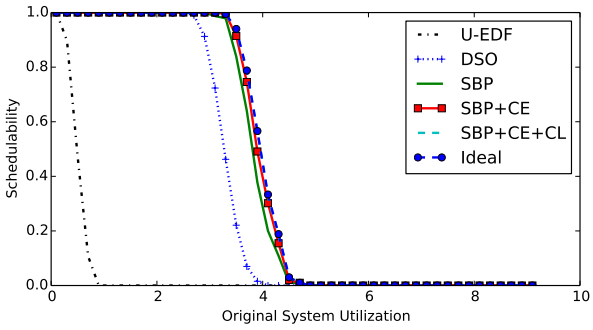
A-Heavy, Long, Light, Mod., Heavy, Small



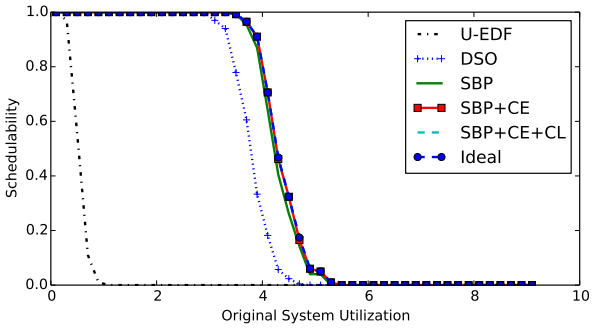
All-Mod., Long, Mod., Mod., Heavy, Large



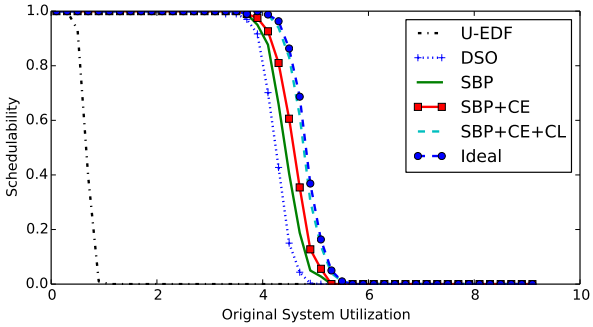
BC-Mod., Cont., Heavy, Light, Heavy, Small



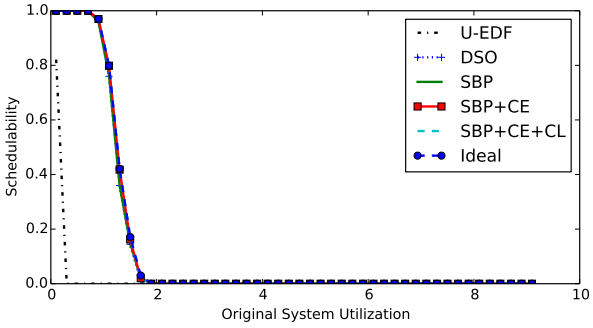
B-Heavy, Cont., Mod., Light, Heavy, Large



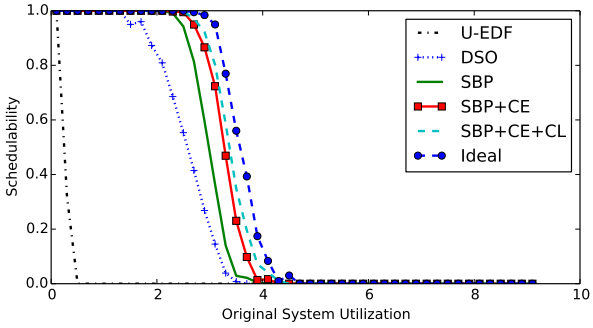
BC-Mod., Cont., Mod., Light, Light, Small



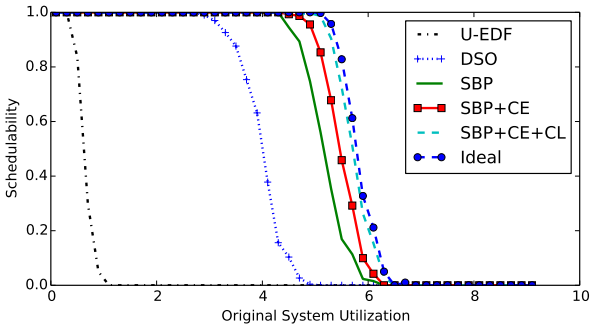
All-Mod., Short, Heavy, Mod., Heavy, Large



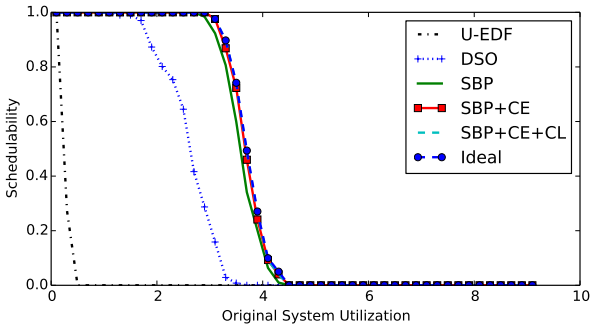
BC-Mod., Cont., Light, Heavy, Light, Large



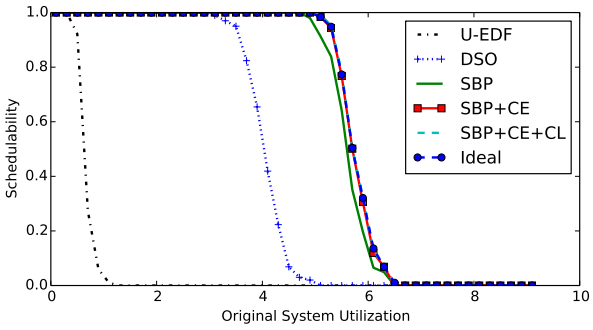
A-Heavy, Cont., Heavy, Heavy, Heavy, Small



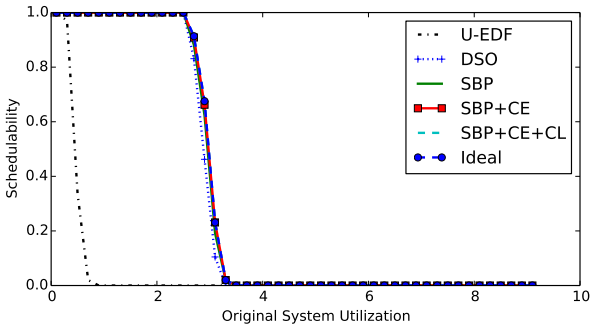
BC-Mod., Long, Mod., Mod., Heavy, Large



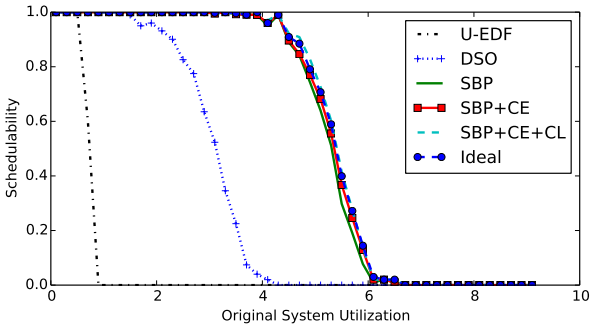
A-Heavy, Cont., Heavy, Heavy, Light, Large



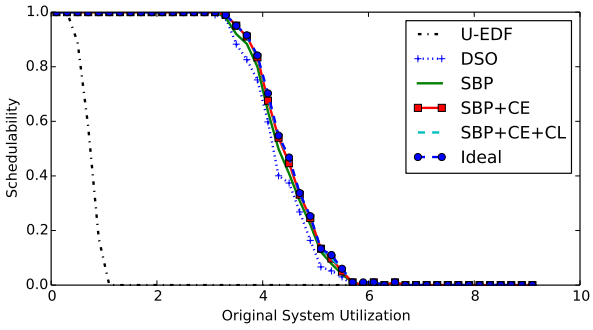
BC-Mod., Long, Mod., Mod., Light, Small



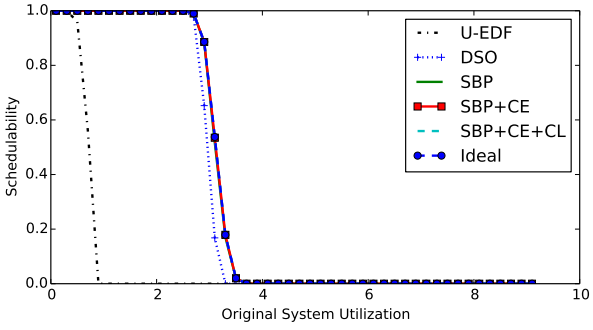
BC-Mod., Long, Light, Mod., Heavy, Small



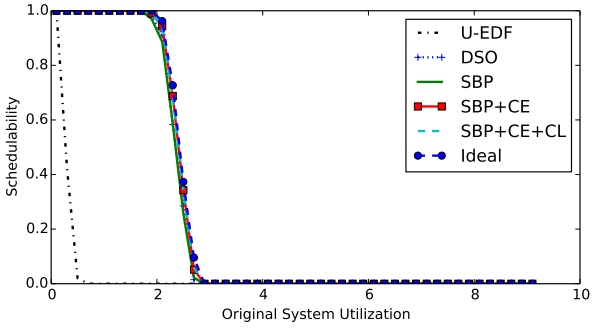
AC-Mod., Long, Heavy, Mod., Light, Large



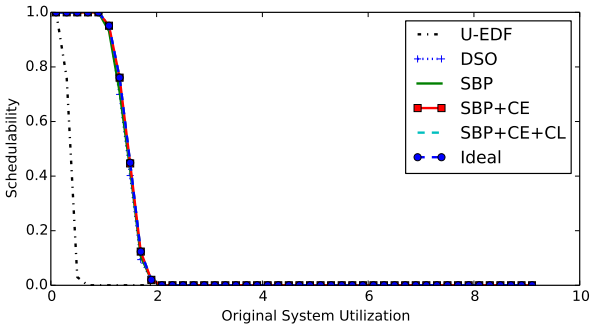
C-Heavy, Short, Mod., Light, Heavy, Large



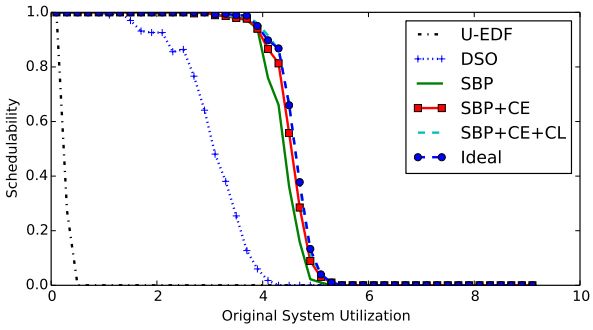
B-Heavy, Long, Light, Light, Heavy, Small



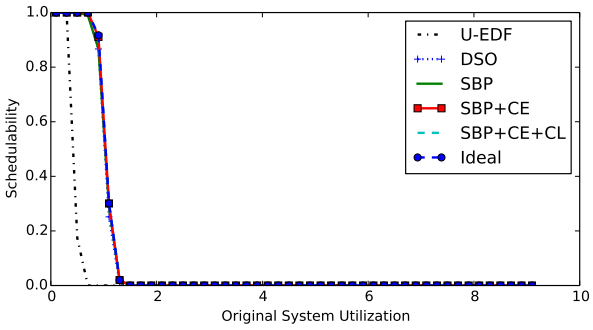
AC-Mod., Long, Light, Heavy, Heavy, Large



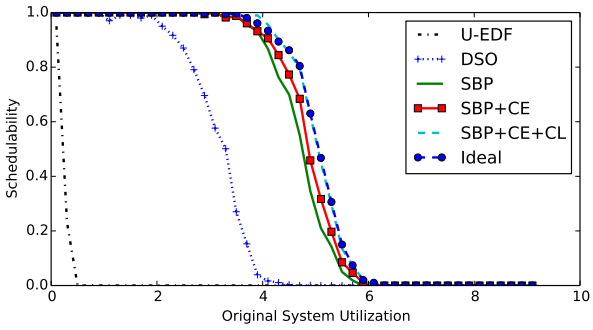
B-Heavy, Cont., Light, Light, Light, Small



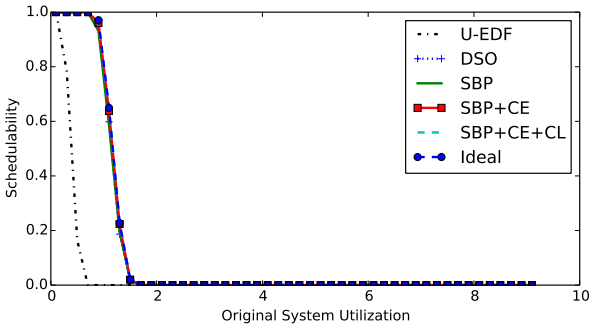
All-Mod., Cont., Heavy, Heavy, Light, Small



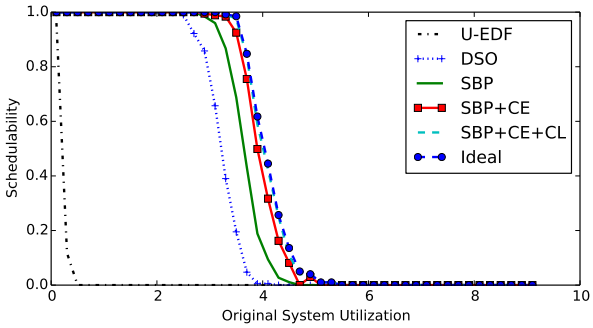
AB-Mod., Short, Light, Light, Light, Small



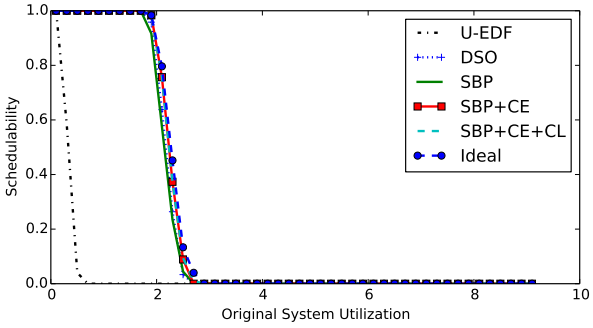
BC-Mod., Cont., Heavy, Heavy, Light, Large



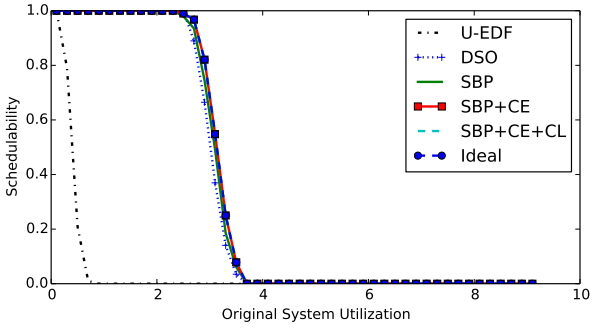
BC-Mod., Short, Light, Mod., Heavy, Large



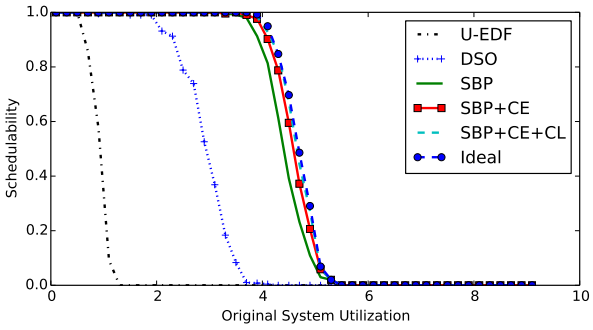
BC-Mod., Cont., Mod., Heavy, Heavy, Small



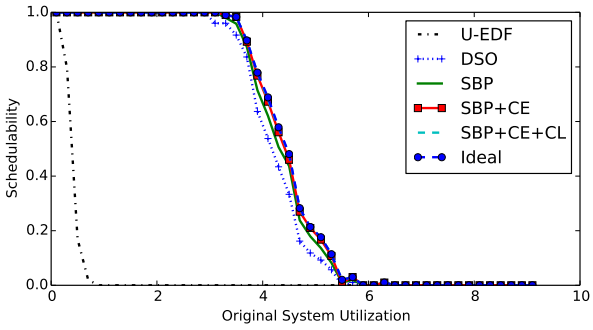
A-Heavy, Long, Light, Heavy, Heavy, Large



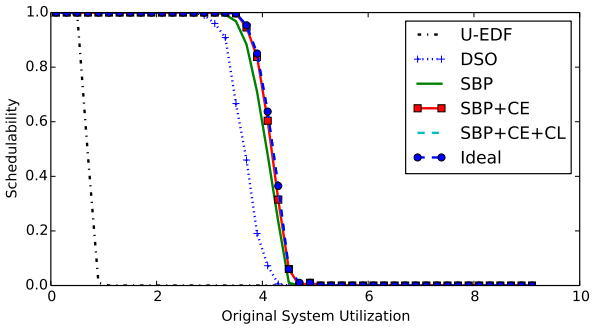
AC-Mod., Short, Mod., Heavy, Light, Small



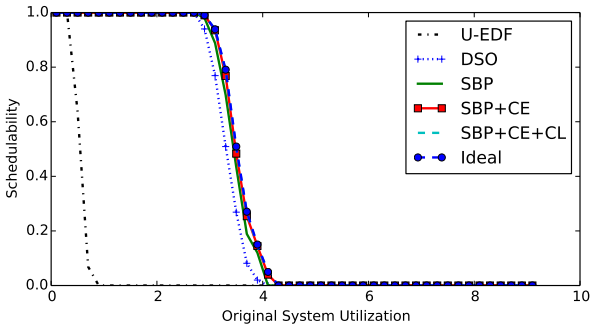
B-Heavy, Long, Heavy, Light, Heavy, Large



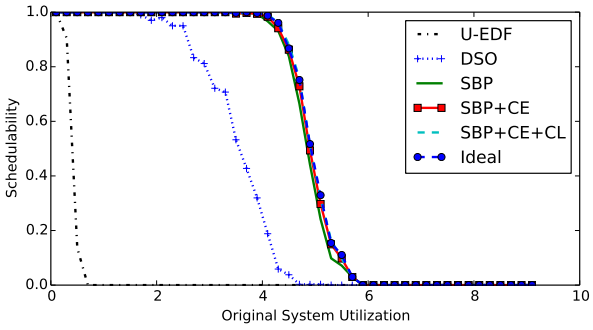
C-Heavy, Short, Mod., Heavy, Light, Large



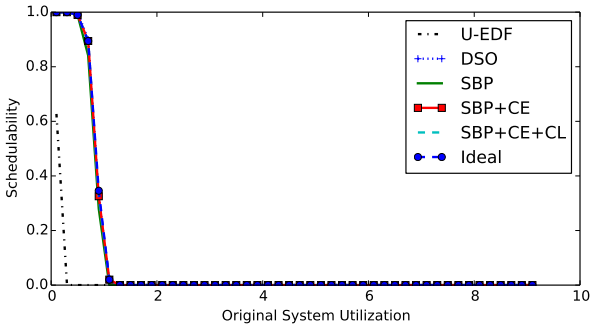
AB-Mod., Short, Heavy, Mod., Light, Large



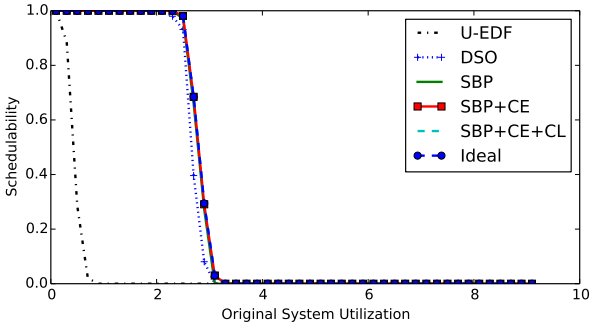
All-Mod., Short, Mod., Mod., Light, Large



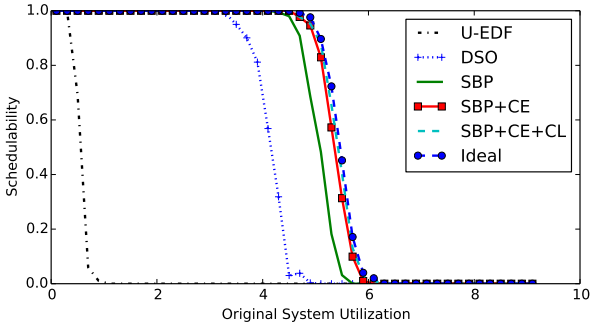
AC-Mod., Cont., Heavy, Mod., Light, Large



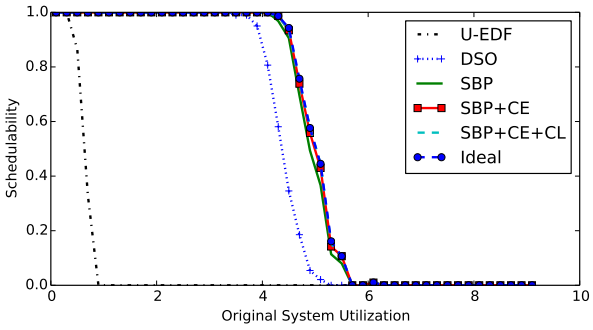
A-Heavy, Cont., Light, Heavy, Heavy, Large



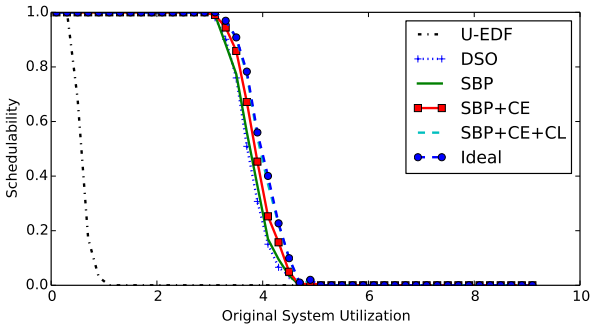
All-Mod., Long, Light, Mod., Light, Large



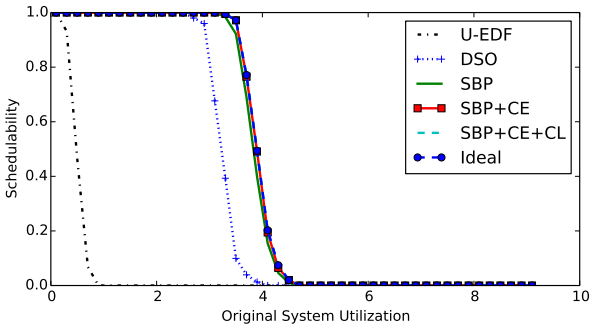
AC-Mod., Long, Mod., Mod., Heavy, Small



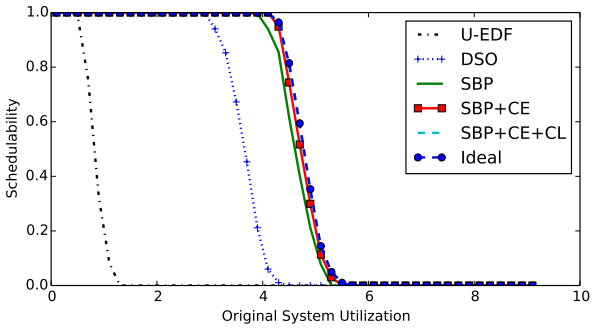
AC-Mod., Short, Heavy, Mod., Light, Large



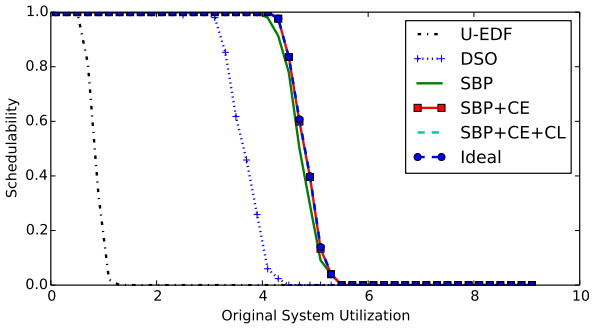
BC-Mod., Short, Mod., Mod., Heavy, Large



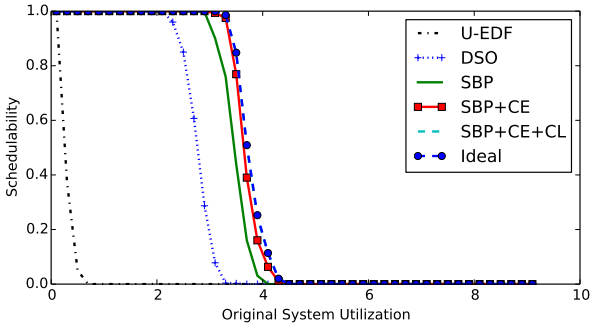
B-Heavy, Cont., Mod., Light, Light, Small



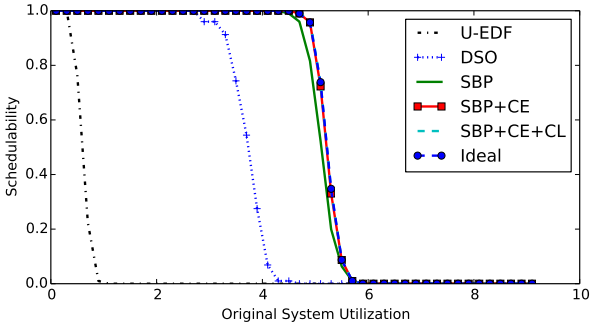
B-Heavy, Long, Mod., Light, Heavy, Large



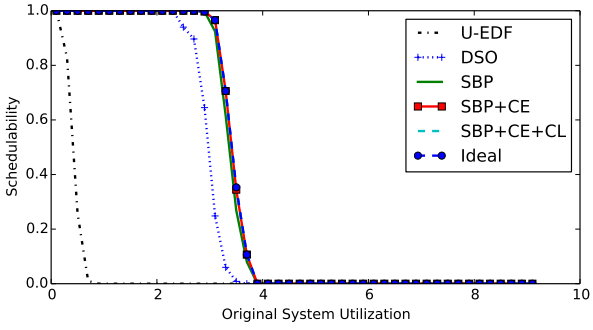
B-Heavy, Long, Mod., Light, Light, Small



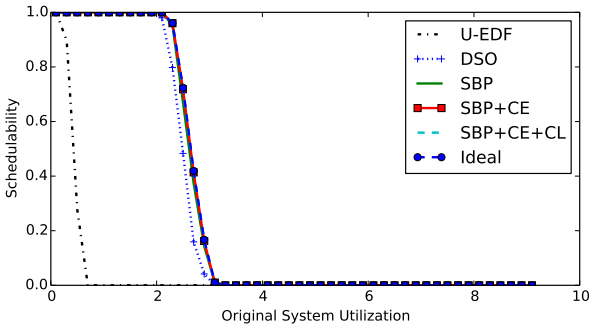
B-Heavy, Cont., Mod., Mod., Heavy, Small



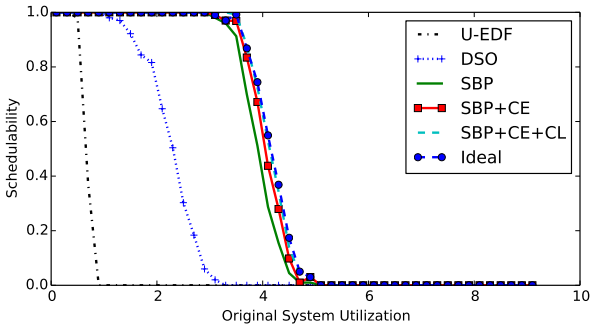
All-Mod., Long, Mod., Mod., Light, Small



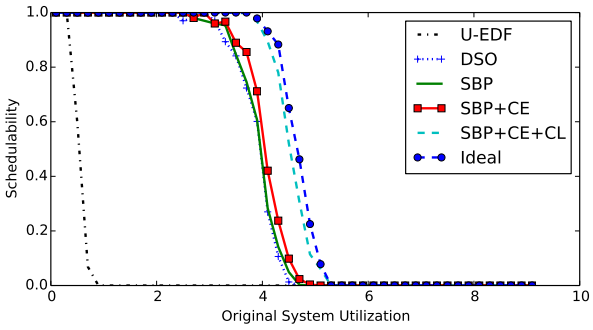
AB-Mod., Cont., Mod., Light, Light, Small



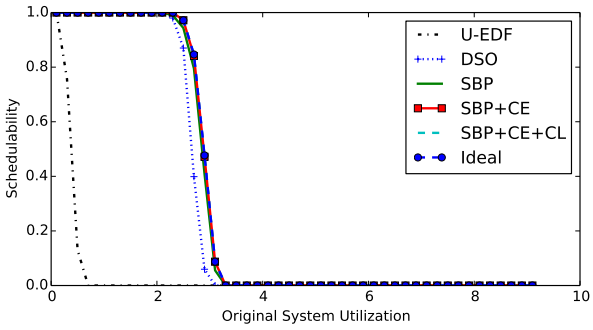
A-Heavy, Cont., Mod., Light, Light, Small



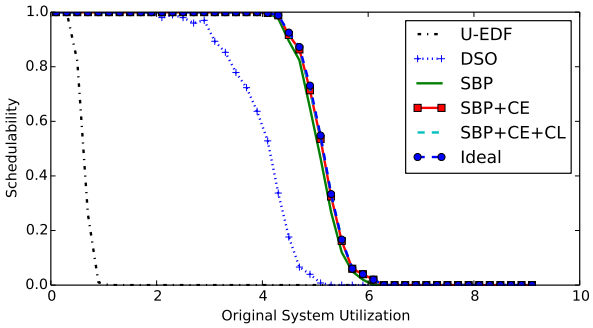
AB-Mod., Long, Heavy, Heavy, Light, Large



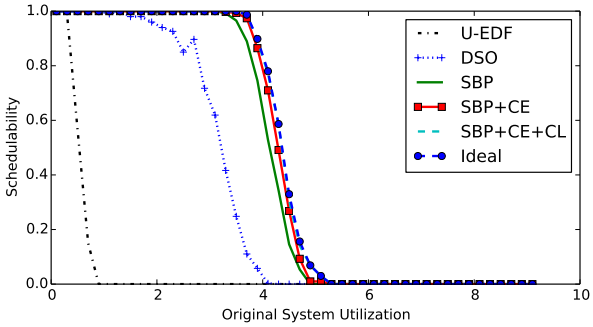
All-Mod., Short, Heavy, Heavy, Heavy, Large



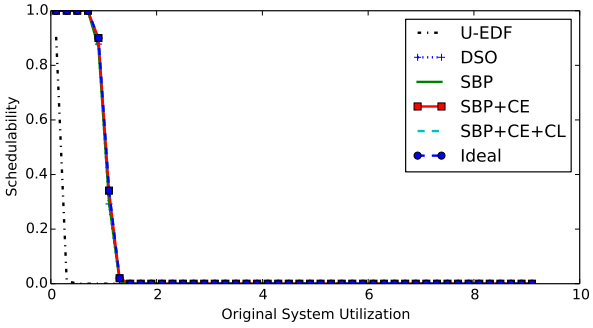
B-Heavy, Long, Light, Heavy, Light, Large



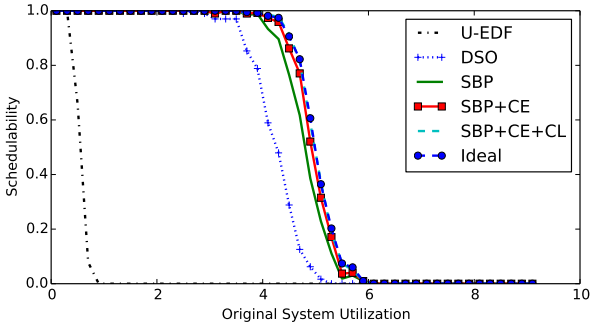
AC-Mod., Cont., Heavy, Light, Light, Large



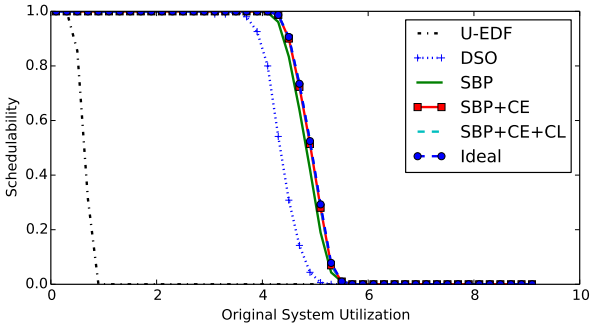
AB-Mod., Cont., Heavy, Light, Heavy, Large



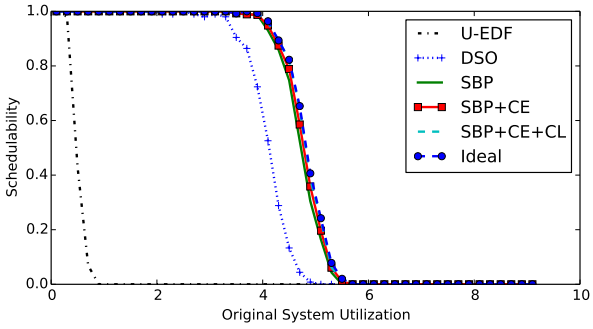
AB-Mod., Cont., Light, Mod., Heavy, Small



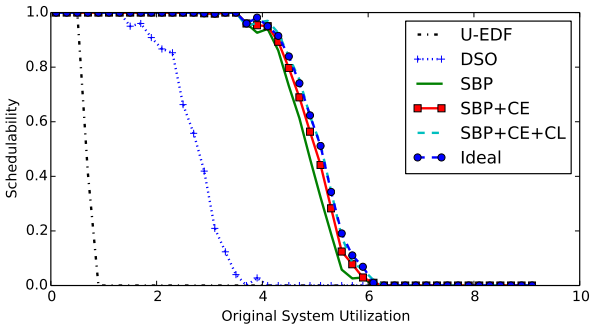
BC-Mod., Short, Heavy, Heavy, Light, Small



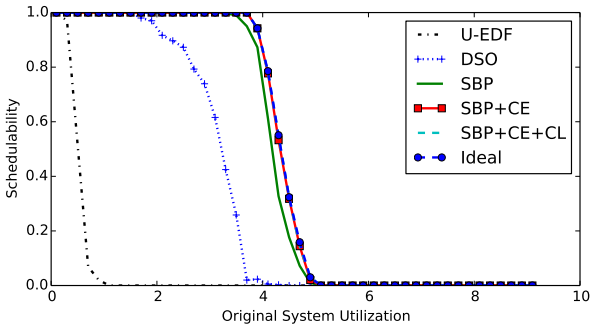
AC-Mod., Short, Heavy, Mod., Light, Small



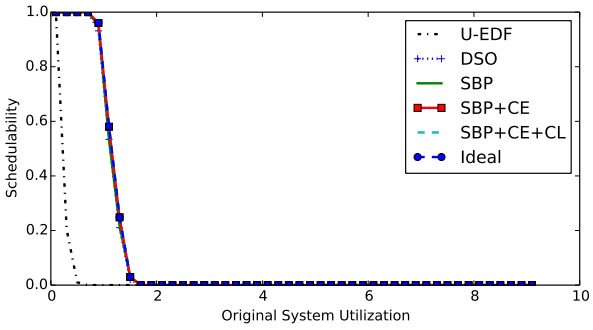
AC-Mod., Short, Heavy, Heavy, Light, Large



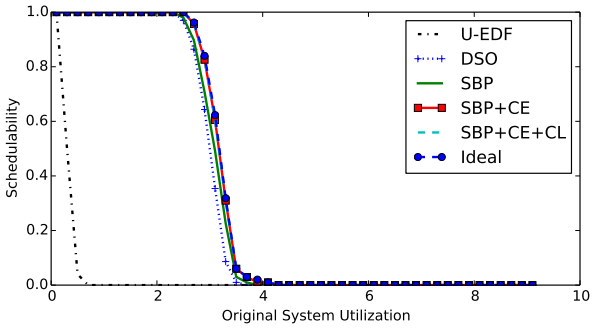
AC-Mod., Long, Heavy, Heavy, Light, Small



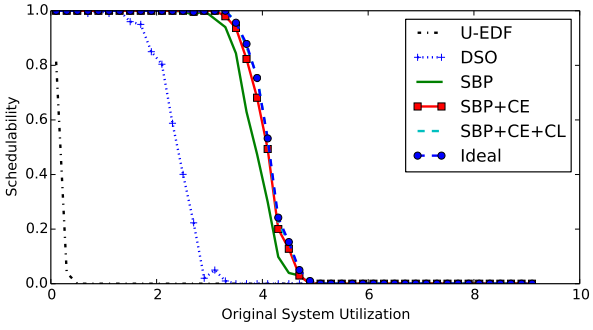
AB-Mod., Cont., Heavy, Light, Light, Small



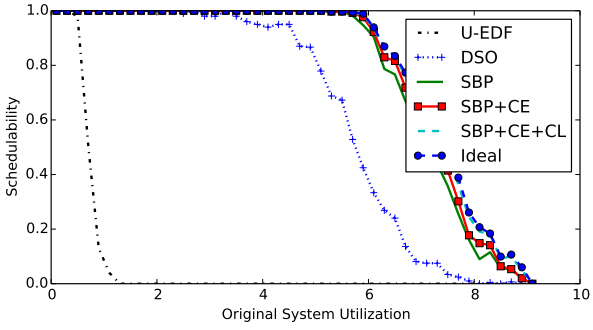
C-Heavy, Cont., Light, Mod., Heavy, Small



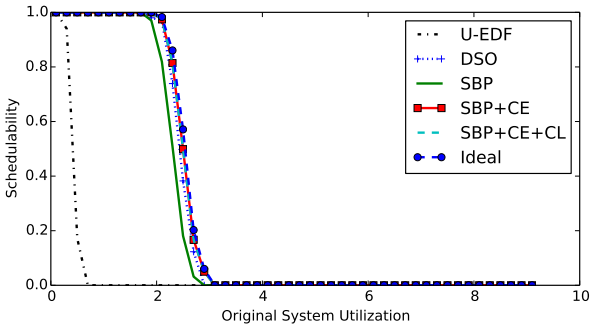
AC-Mod., Cont., Mod., Mod., Heavy, Small



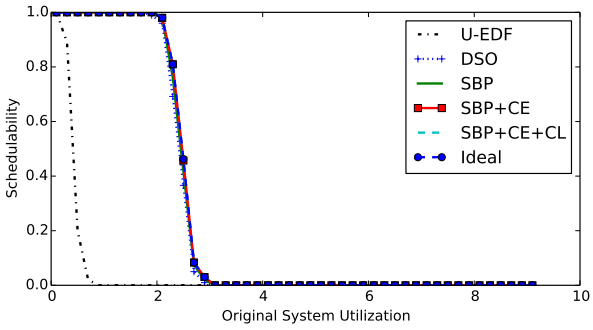
B-Heavy, Cont., Heavy, Heavy, Light, Small



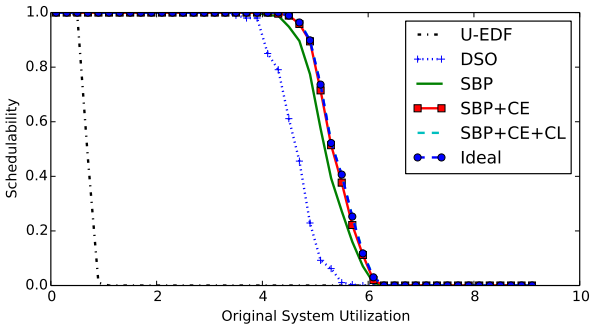
C-Heavy, Cont., Heavy, Light, Heavy, Large



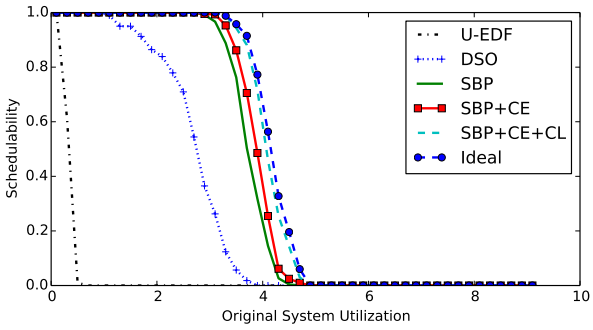
A-Heavy, Short, Mod., Heavy, Heavy, Small



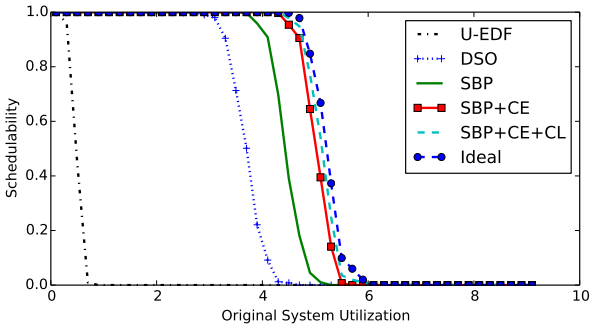
AC-Mod., Long, Light, Mod., Heavy, Small



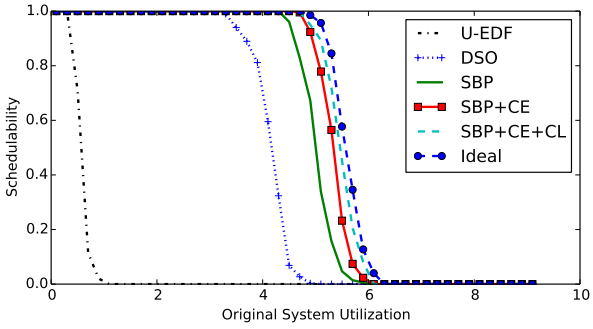
BC-Mod., Short, Heavy, Mod., Light, Small



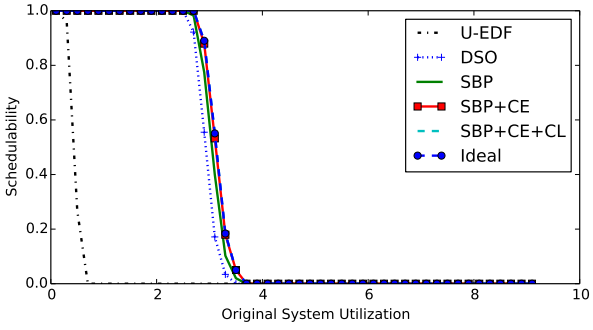
AB-Mod., Cont., Heavy, Mod., Heavy, Large



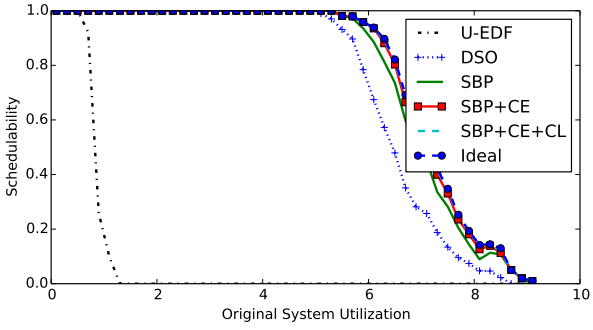
AC-Mod., Long, Mod., Heavy, Heavy, Small



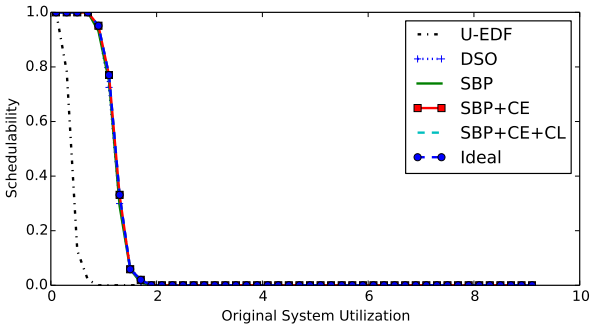
AC-Mod., Long, Mod., Mod., Heavy, Large



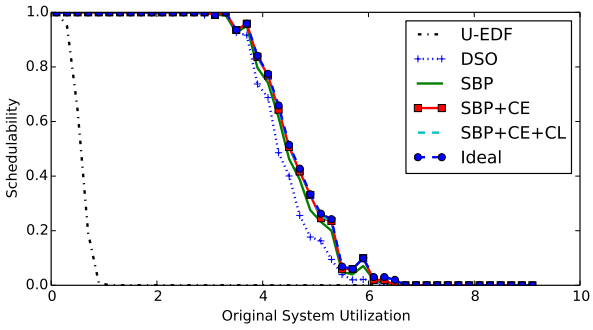
AB-Mod., Short, Mod., Heavy, Light, Small



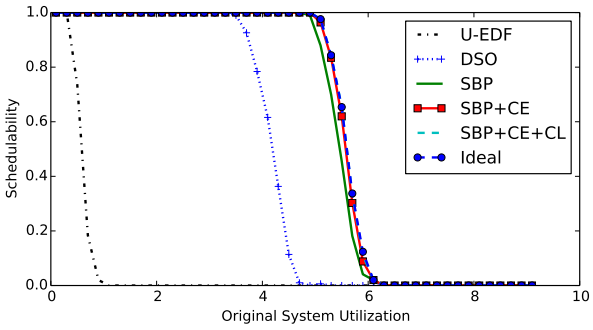
C-Heavy, Short, Heavy, Light, Heavy, Small



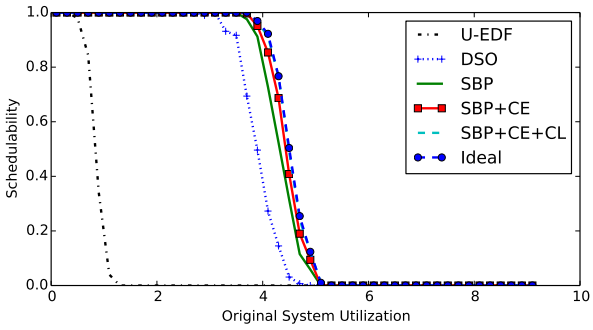
BC-Mod., Cont., Light, Light, Heavy, Small



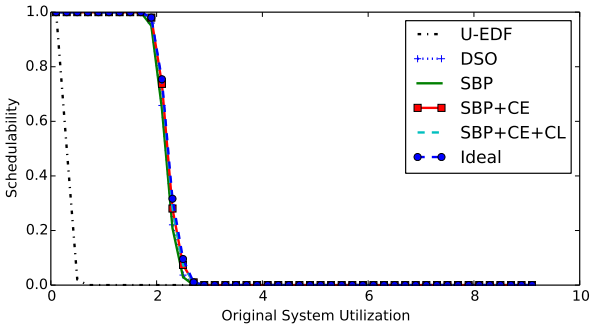
C-Heavy, Cont., Mod., Light, Heavy, Large



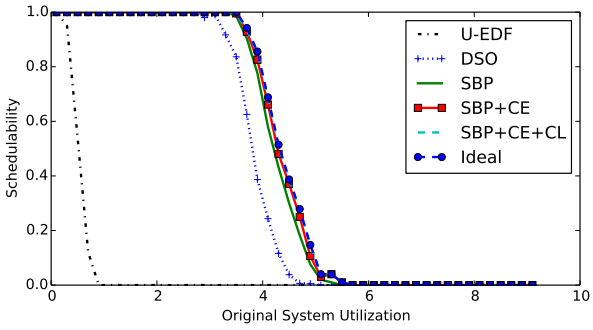
AC-Mod., Long, Mod., Mod., Light, Large



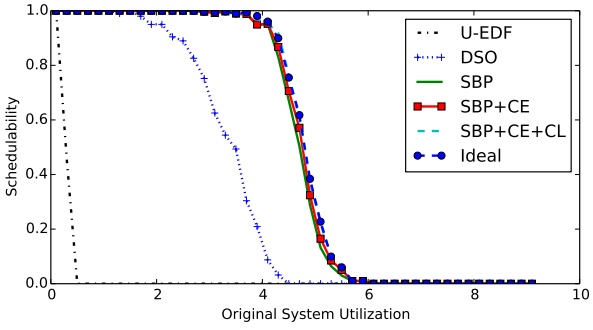
B-Heavy, Short, Heavy, Light, Heavy, Large



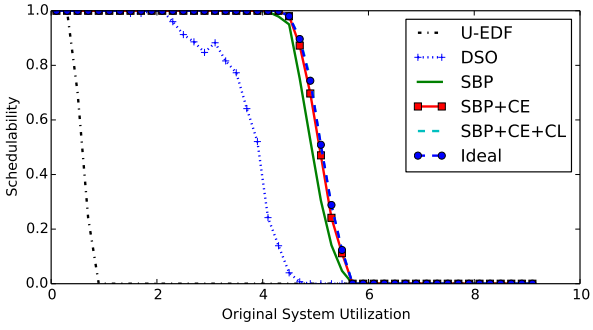
A-Heavy, Long, Light, Heavy, Heavy, Small



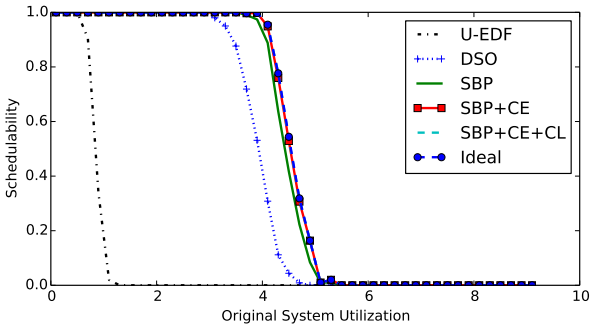
BC-Mod., Cont., Mod., Light, Heavy, Large



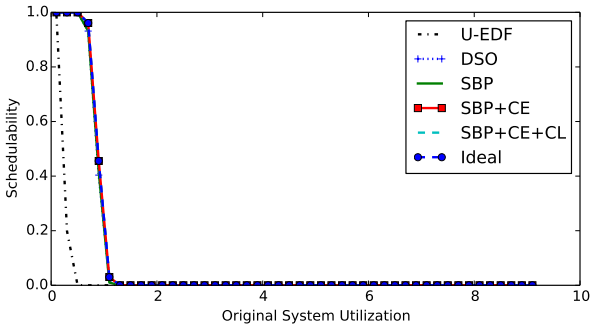
AC-Mod., Cont., Heavy, Heavy, Light, Large



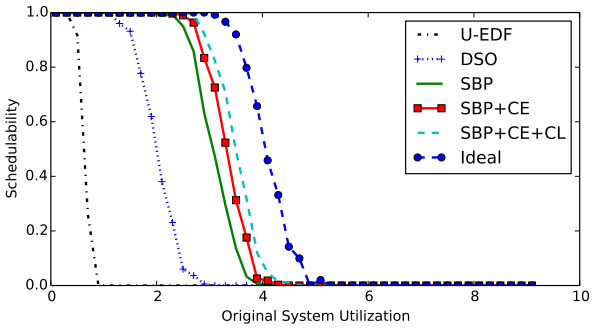
All-Mod., Cont., Heavy, Light, Light, Large



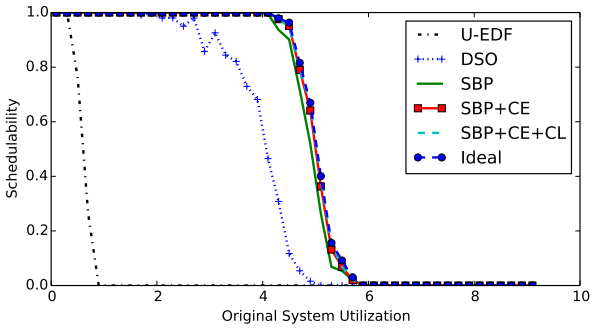
B-Heavy, Short, Heavy, Light, Light, Small



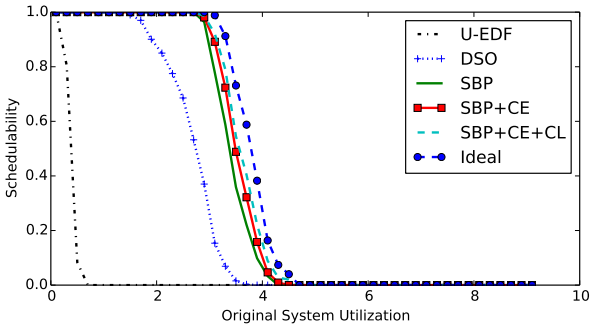
AC-Mod., Short, Light, Heavy, Light, Small



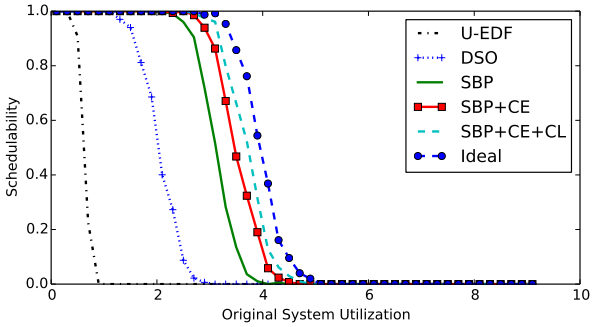
A-Heavy, Long, Heavy, Heavy, Heavy, Large



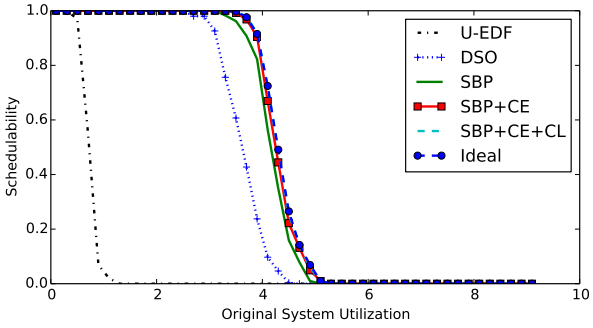
AC-Mod., Cont., Heavy, Light, Heavy, Small



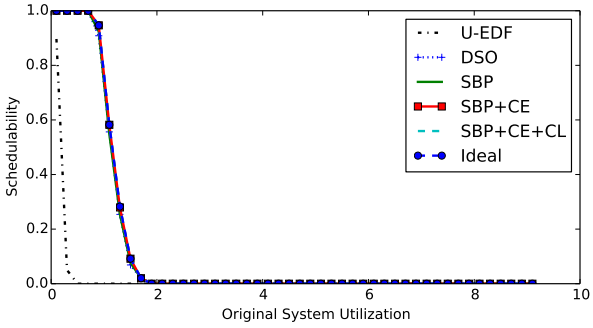
A-Heavy, Cont., Heavy, Mod., Heavy, Large



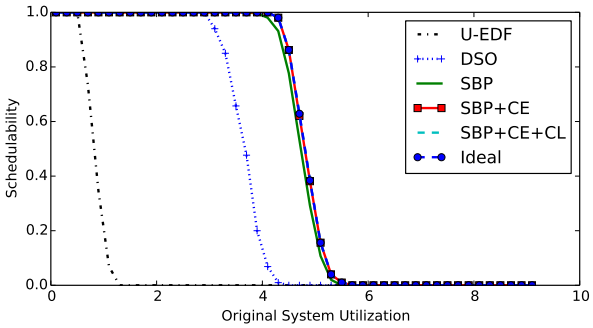
A-Heavy, Long, Heavy, Heavy, Heavy, Small



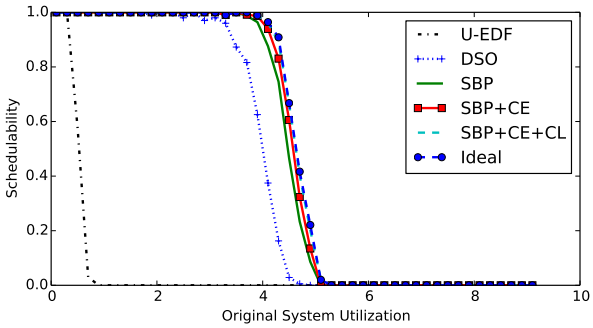
B-Heavy, Short, Heavy, Mod., Light, Large



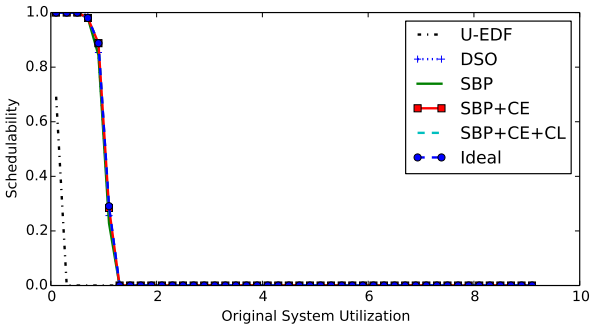
C-Heavy, Cont., Light, Heavy, Light, Large



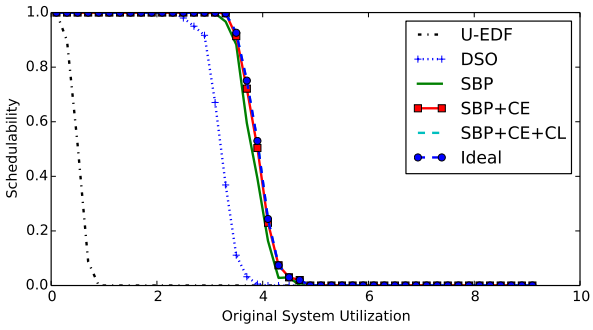
B-Heavy, Long, Mod., Light, Light, Large



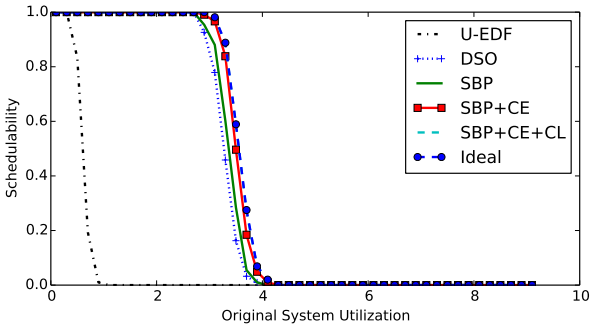
All-Mod., Short, Heavy, Heavy, Light, Large



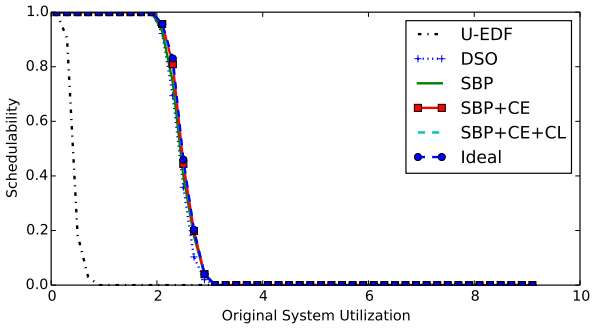
All-Mod., Cont., Light, Heavy, Heavy, Small



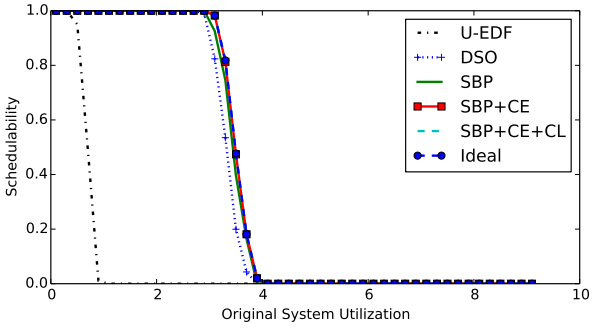
B-Heavy, Cont., Mod., Light, Heavy, Small



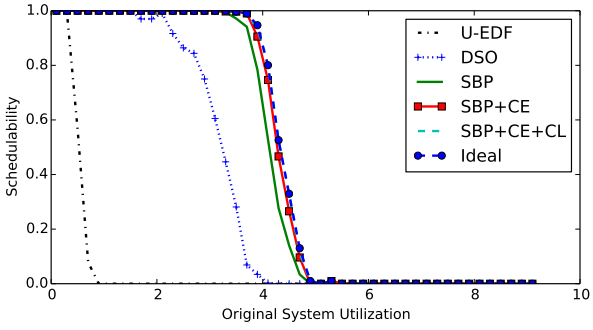
B-Heavy, Short, Mod., Mod., Heavy, Small



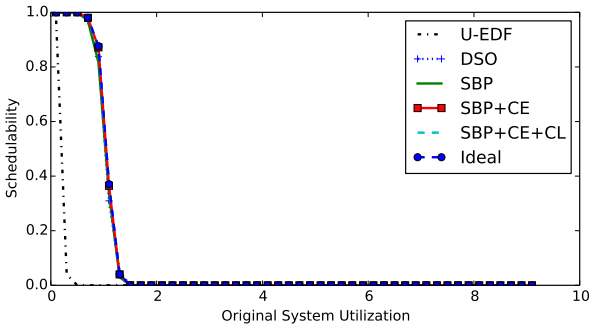
AC-Mod., Long, Light, Mod., Heavy, Large



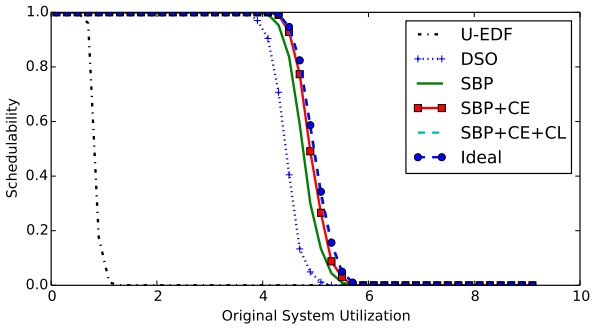
All-Mod., Short, Mod., Light, Heavy, Small



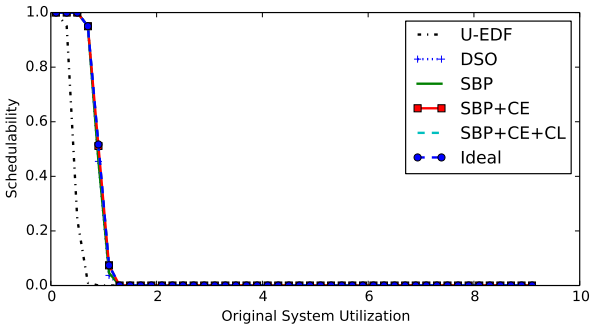
AB-Mod., Cont., Heavy, Light, Heavy, Small



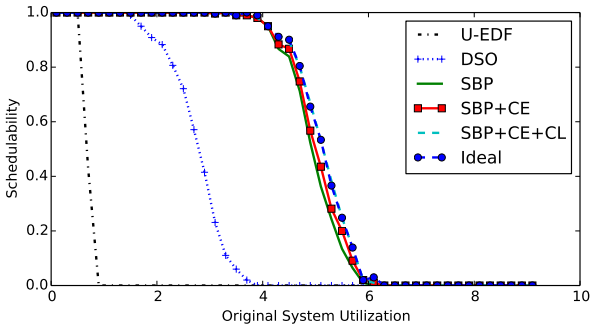
All-Mod., Cont., Light, Mod., Light, Large



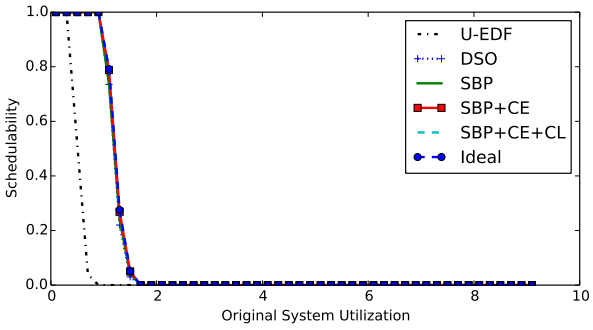
All-Mod., Short, Heavy, Light, Heavy, Large



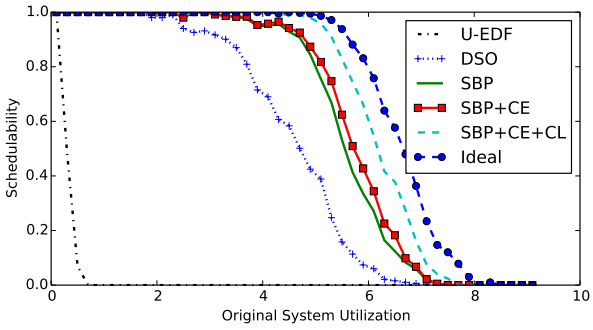
AC-Mod., Short, Light, Light, Light, Large



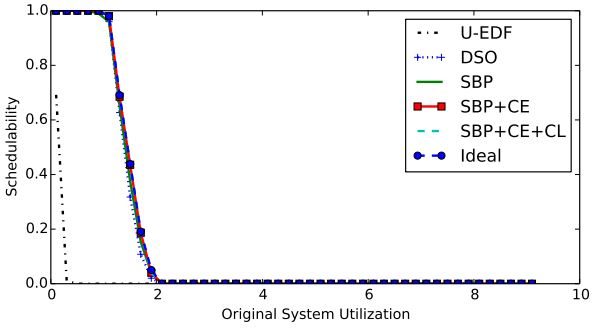
AC-Mod., Long, Heavy, Heavy, Light, Large



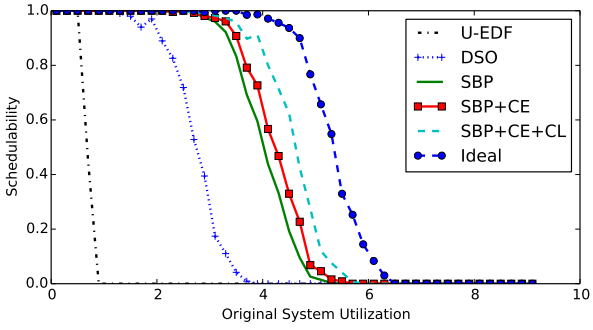
B-Heavy, Short, Light, Light, Heavy, Large



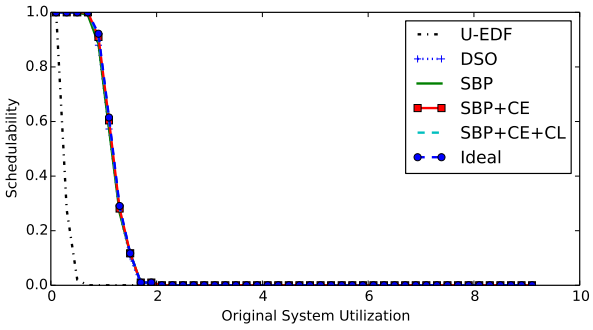
C-Heavy, Cont., Heavy, Heavy, Heavy, Small



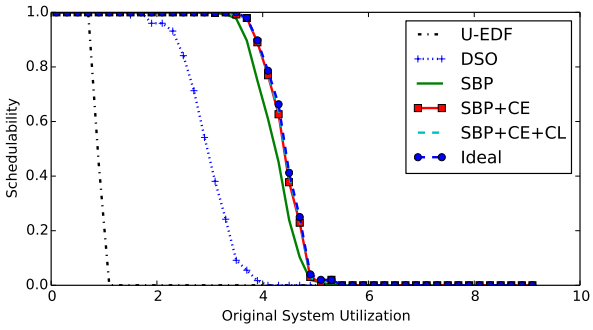
B-Heavy, Cont., Light, Heavy, Light, Large



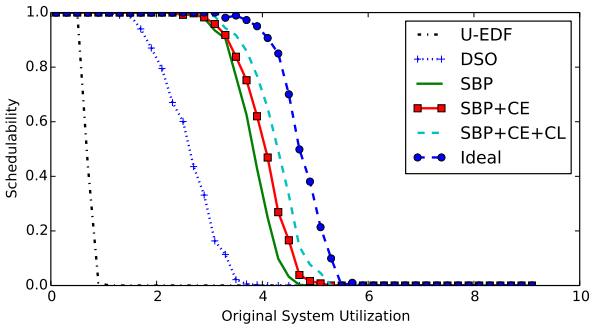
AC-Mod., Long, Heavy, Heavy, Heavy, Large



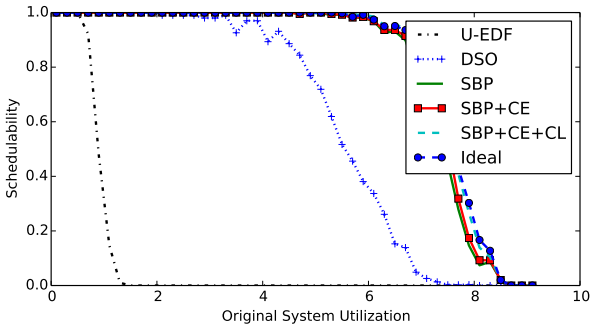
C-Heavy, Cont., Light, Mod., Heavy, Large



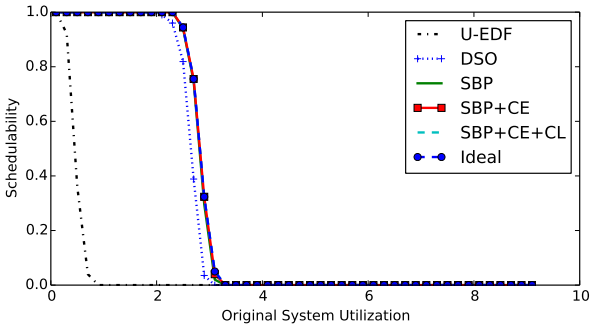
A-Heavy, Long, Heavy, Light, Heavy, Small



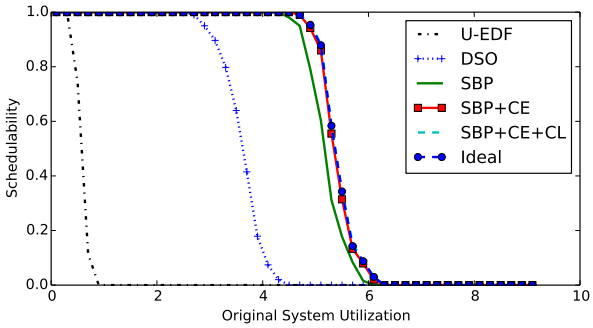
All-Mod., Long, Heavy, Heavy, Heavy, Small



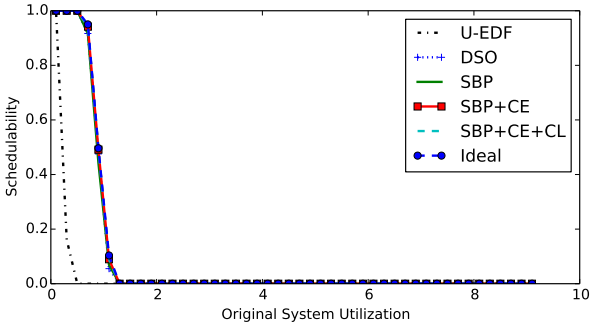
C-Heavy, Long, Heavy, Light, Heavy, Small



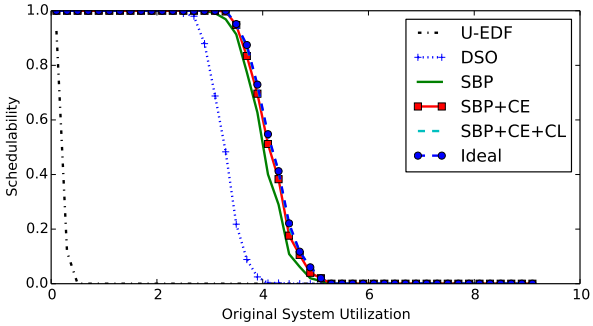
AB-Mod., Long, Light, Mod., Light, Large



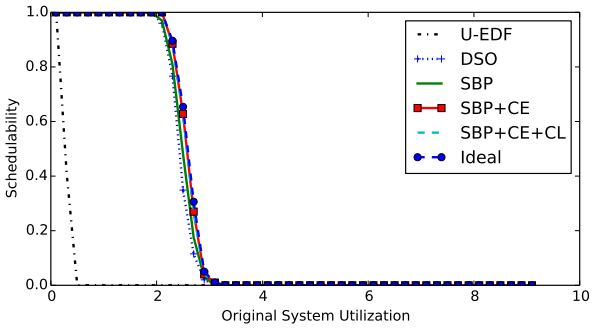
BC-Mod., Long, Mod., Heavy, Light, Small



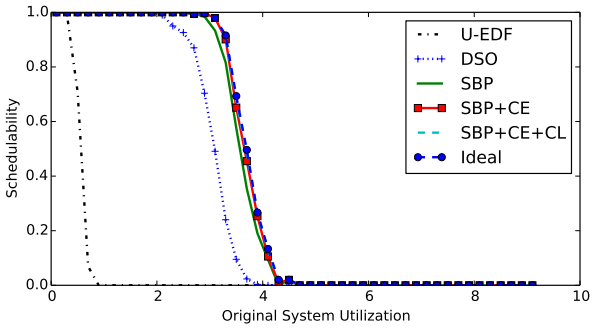
AC-Mod., Short, Light, Heavy, Light, Large



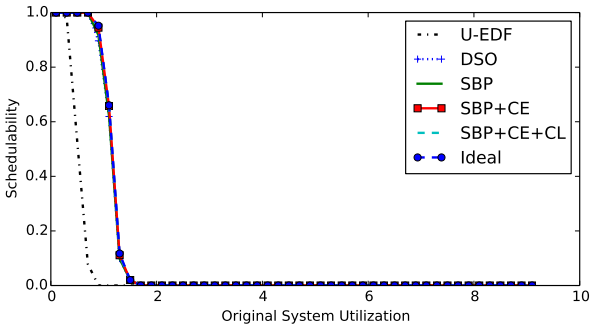
BC-Mod., Cont., Mod., Heavy, Light, Large



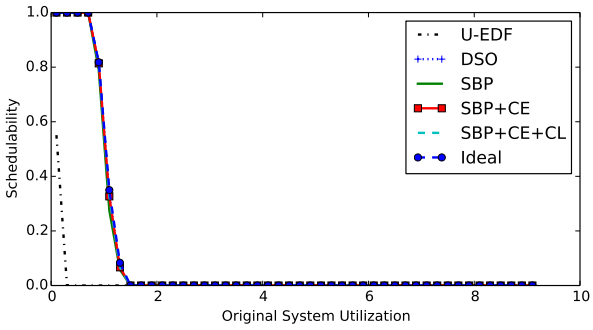
A-Heavy, Cont., Mod., Mod., Heavy, Small



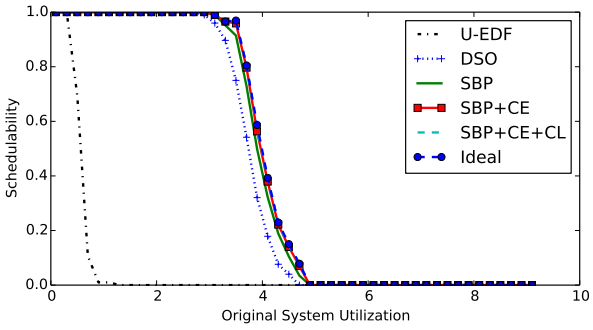
A-Heavy, Short, Heavy, Heavy, Light, Large



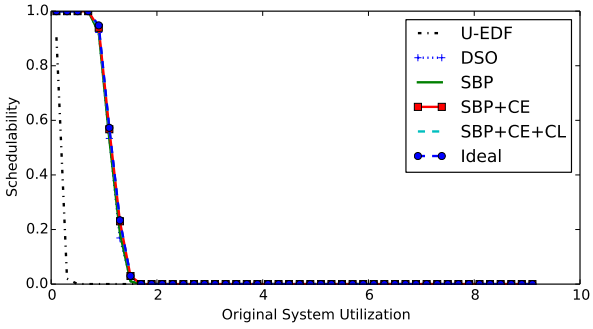
BC-Mod., Short, Light, Light, Heavy, Large



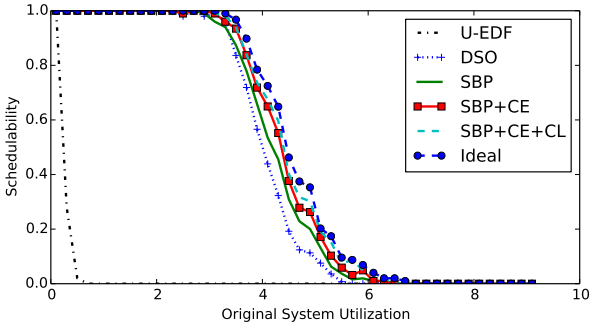
AB-Mod., Cont., Light, Heavy, Heavy, Large



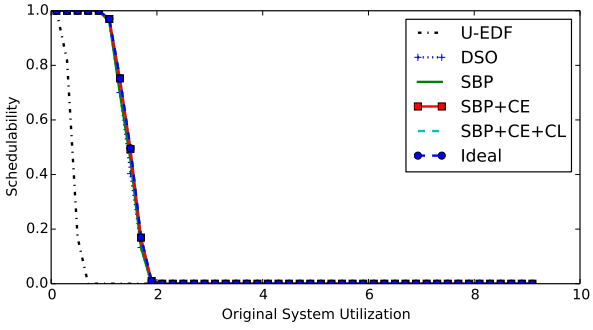
BC-Mod., Short, Mod., Mod., Light, Large



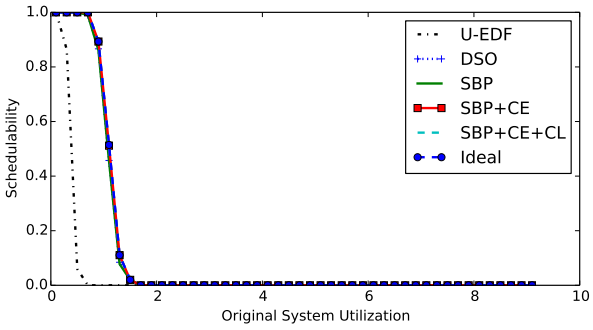
C-Heavy, Cont., Light, Heavy, Heavy, Small



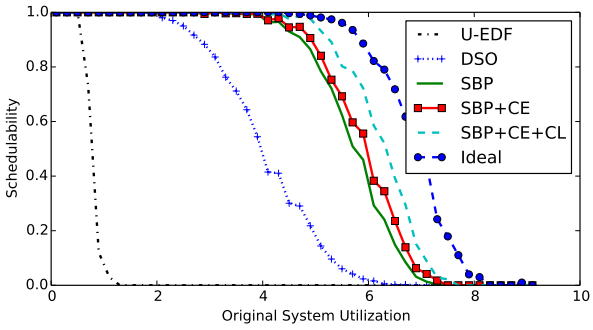
C-Heavy, Cont., Mod., Heavy, Heavy, Large



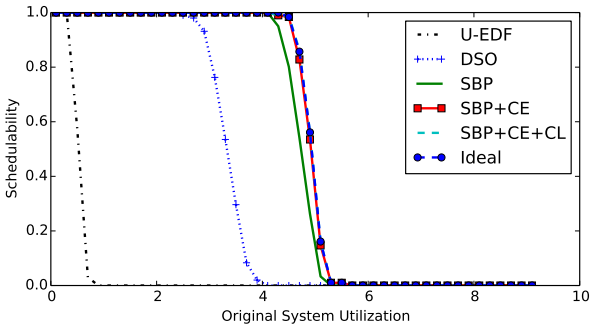
B-Heavy, Cont., Light, Light, Heavy, Small



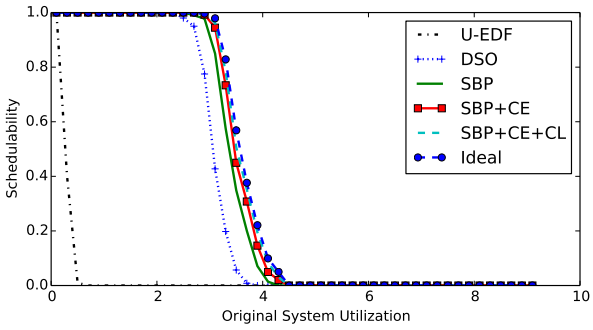
C-Heavy, Short, Light, Mod., Heavy, Small



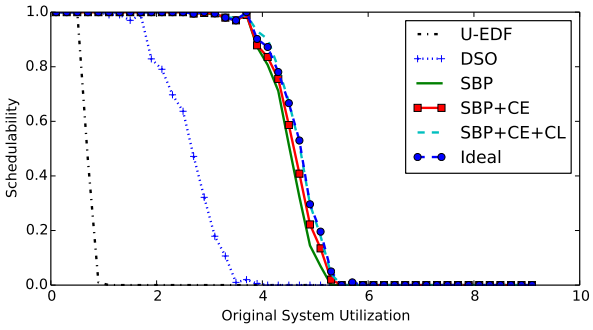
C-Heavy, Long, Heavy, Heavy, Heavy, Small



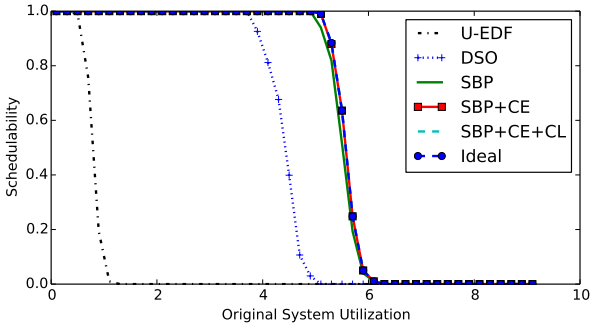
All-Mod., Long, Mod., Heavy, Light, Small



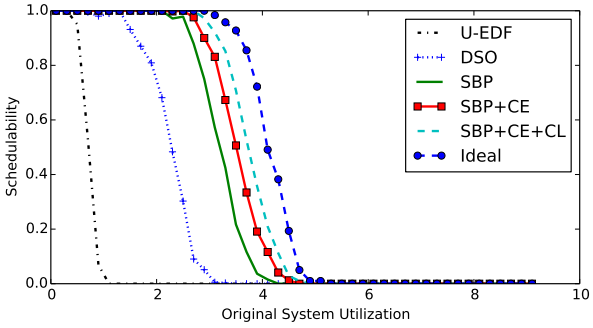
All-Mod., Cont., Mod., Mod., Heavy, Large



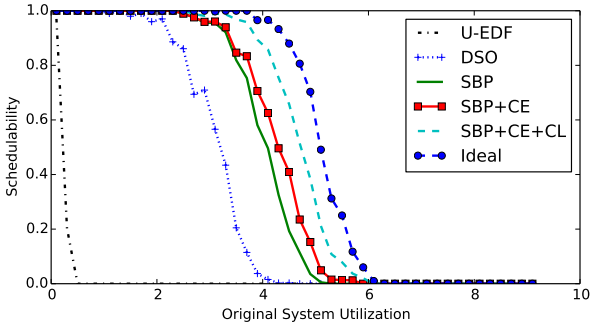
All-Mod., Long, Heavy, Heavy, Light, Large



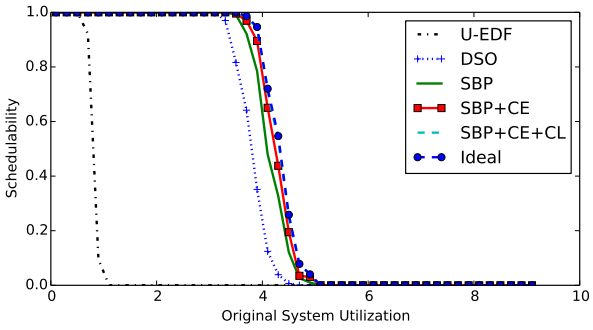
All-Mod., Long, Mod., Light, Light, Small



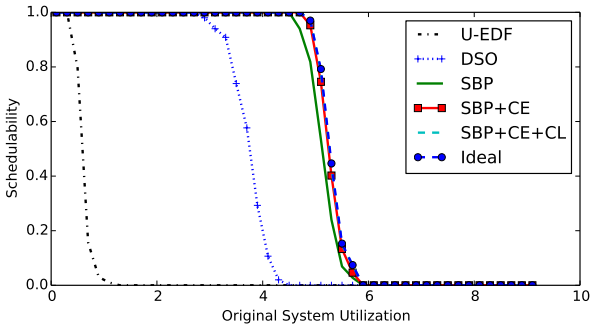
B-Heavy, Long, Heavy, Heavy, Heavy, Small



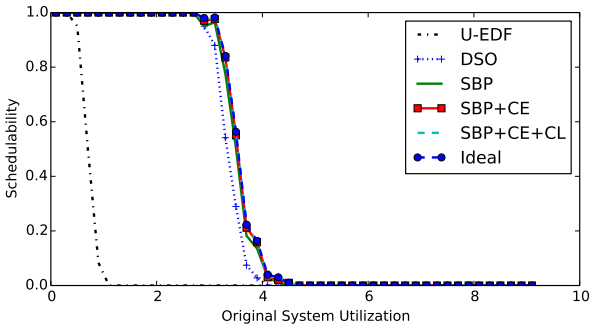
BC-Mod., Cont., Heavy, Heavy, Heavy, Small



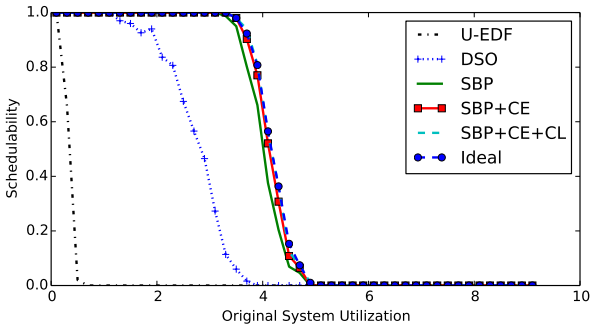
AB-Mod., Short, Heavy, Light, Heavy, Large



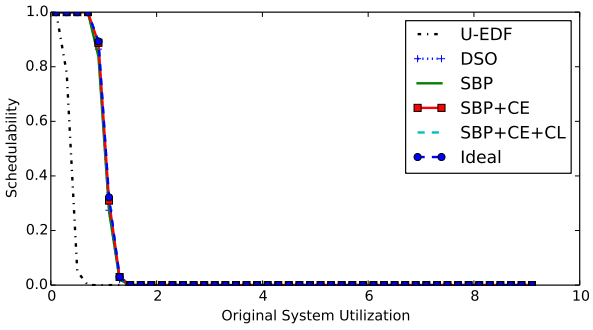
All-Mod., Long, Mod., Mod., Light, Large



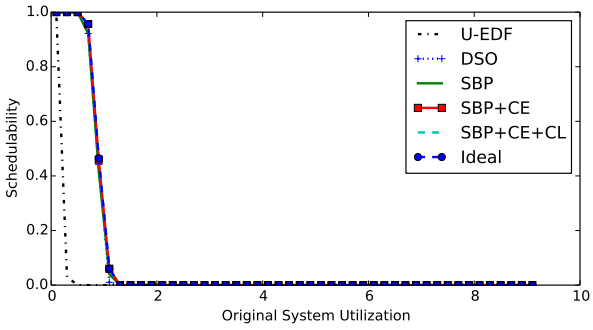
All-Mod., Short, Mod., Light, Light, Large



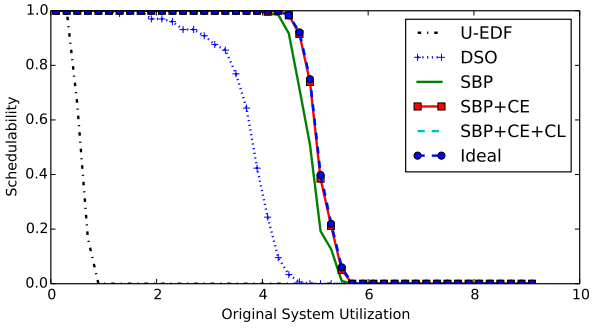
AB-Mod., Cont., Heavy, Mod., Light, Large



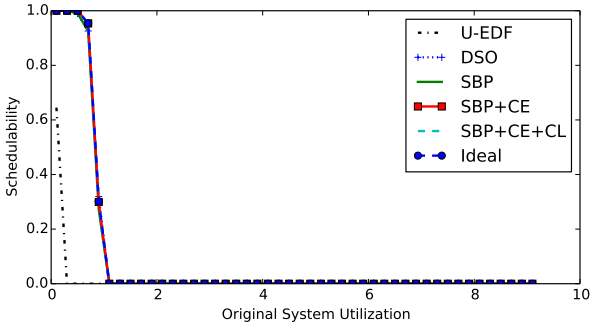
AB-Mod., Short, Light, Mod., Heavy, Large



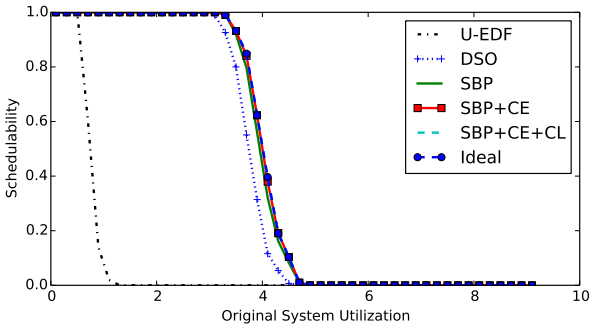
AC-Mod., Cont., Light, Mod., Light, Small



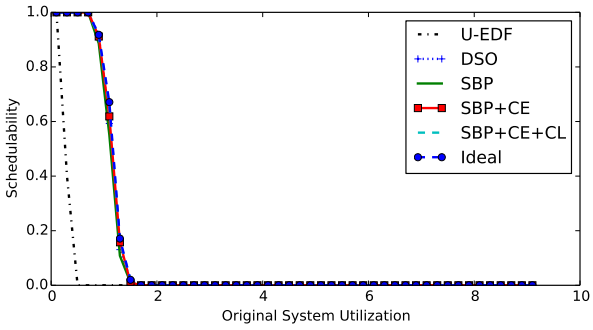
All-Mod., Cont., Heavy, Light, Light, Small



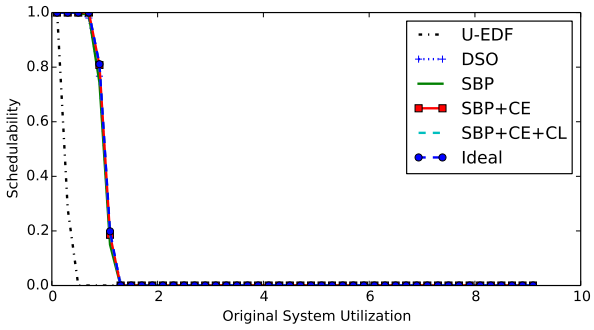
A-Heavy, Cont., Light, Heavy, Light, Small



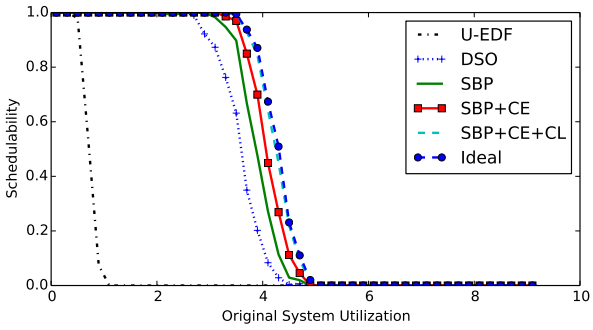
BC-Mod., Short, Mod., Light, Light, Small



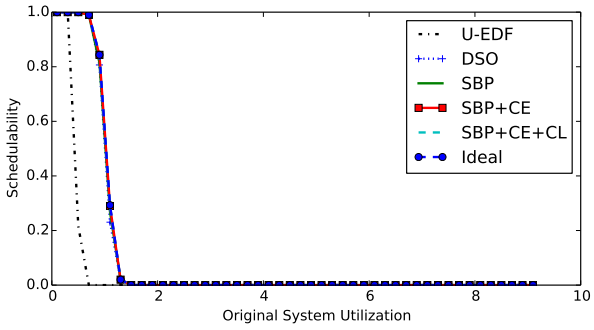
BC-Mod., Short, Light, Heavy, Heavy, Large



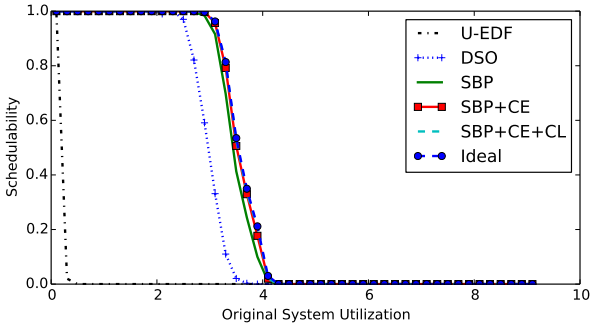
All-Mod., Short, Light, Heavy, Heavy, Small



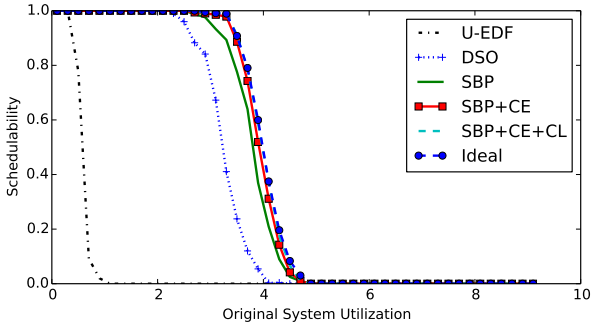
B-Heavy, Short, Heavy, Mod., Heavy, Small



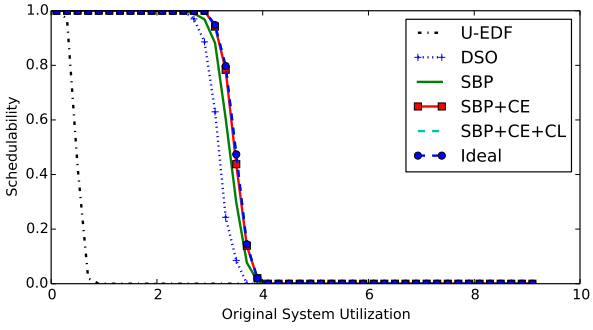
AB-Mod., Short, Light, Light, Heavy, Small



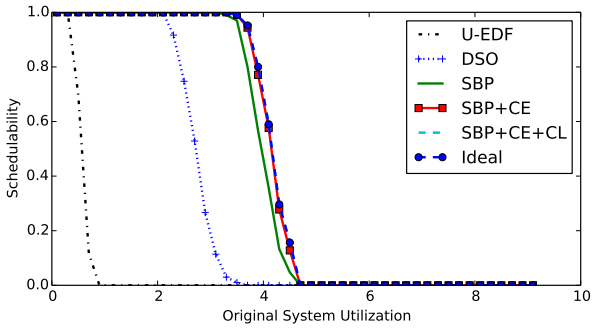
All-Mod., Cont., Mod., Heavy, Light, Large



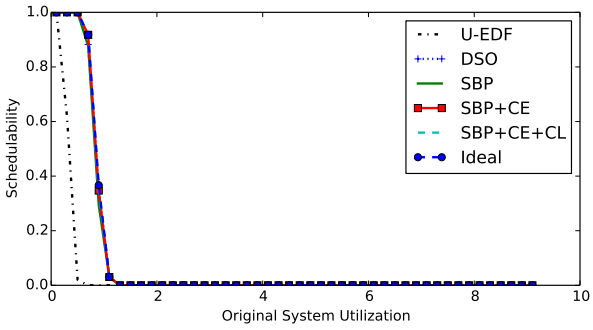
B-Heavy, Short, Heavy, Heavy, Light, Large



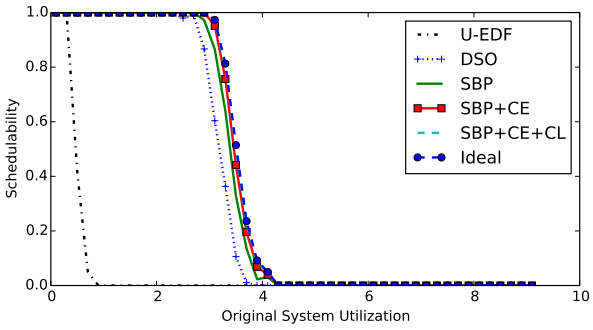
B-Heavy, Short, Mod., Heavy, Light, Small



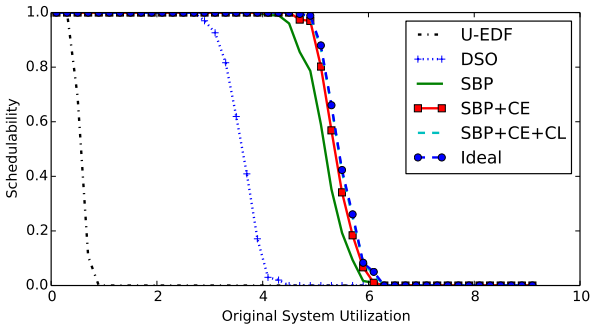
B-Heavy, Long, Mod., Heavy, Light, Small



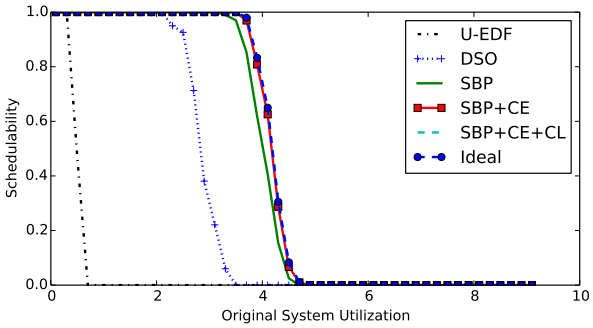
A-Heavy, Short, Light, Mod., Heavy, Large



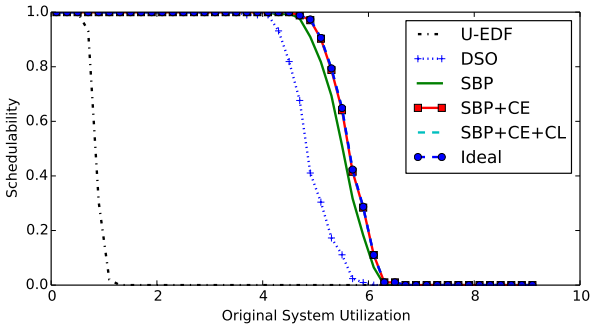
B-Heavy, Short, Mod., Heavy, Light, Large



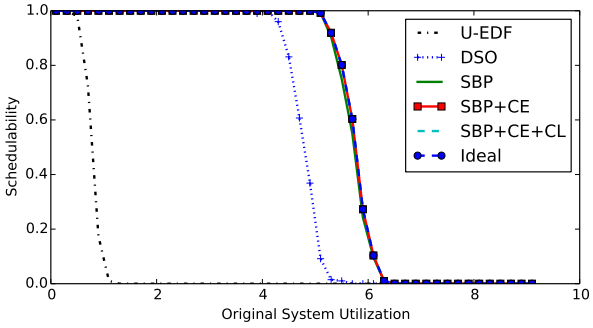
BC-Mod., Long, Mod., Heavy, Light, Large



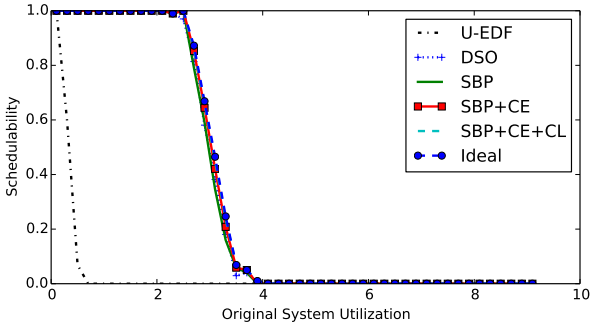
AB-Mod., Long, Mod., Heavy, Light, Small



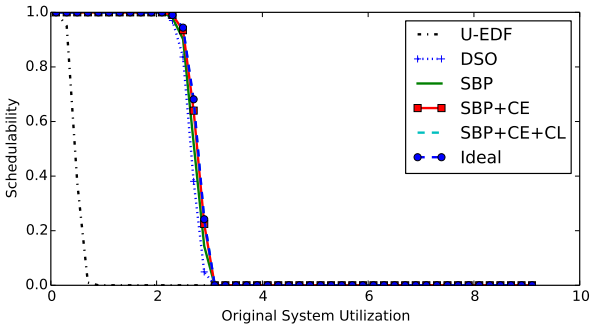
BC-Mod., Short, Heavy, Light, Light, Small



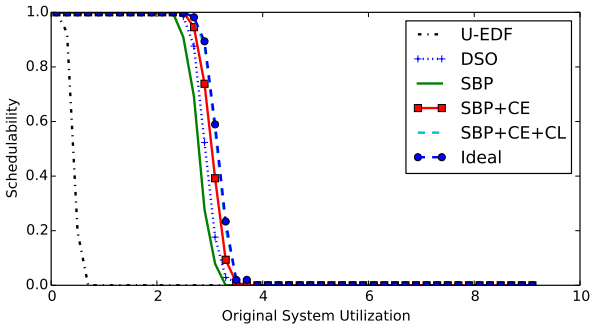
AC-Mod., Long, Mod., Light, Light, Small



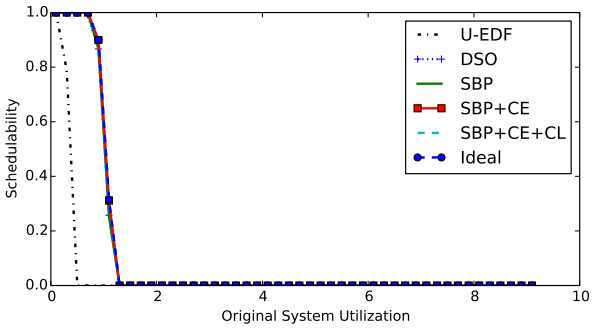
C-Heavy, Long, Light, Heavy, Heavy, Large



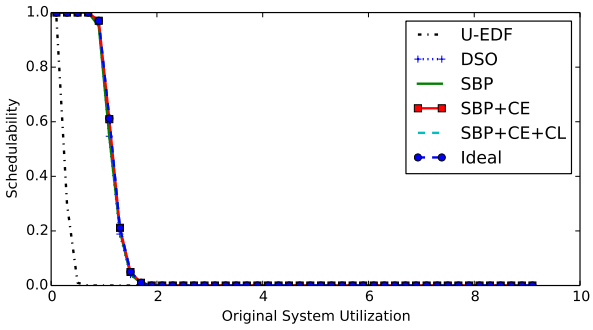
AB-Mod., Long, Light, Mod., Heavy, Large



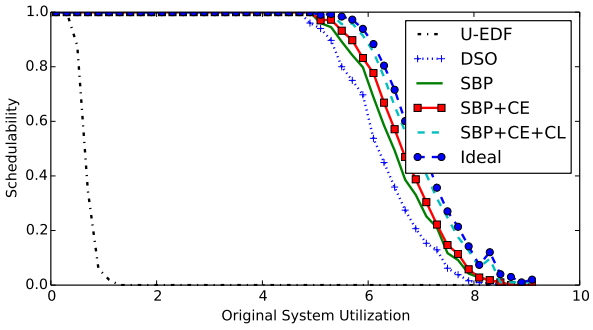
AB-Mod., Short, Mod., Heavy, Heavy, Small



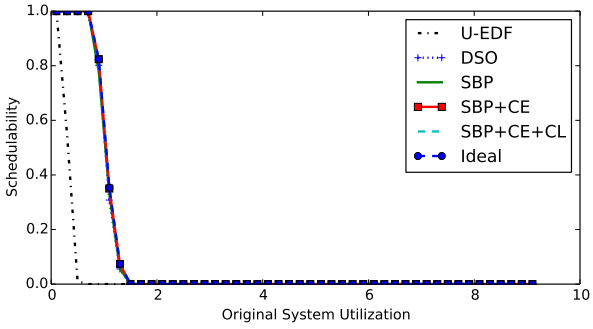
AB-Mod., Short, Light, Mod., Heavy, Small



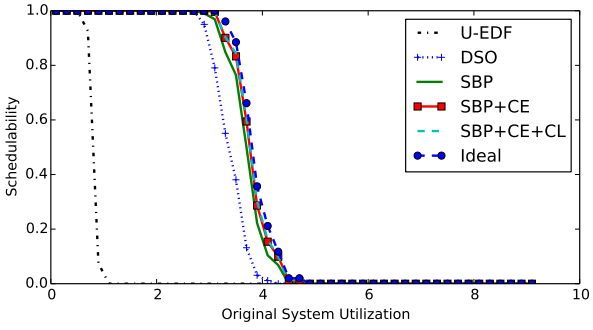
C-Heavy, Cont., Light, Mod., Light, Small



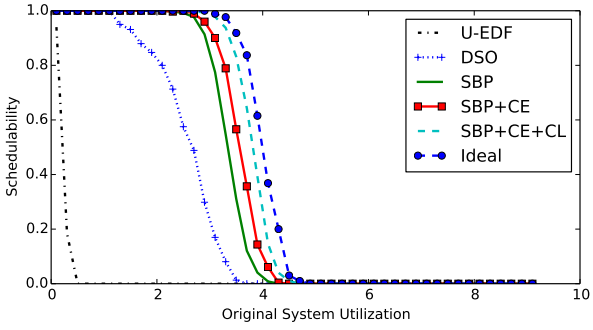
C-Heavy, Short, Heavy, Mod., Heavy, Large



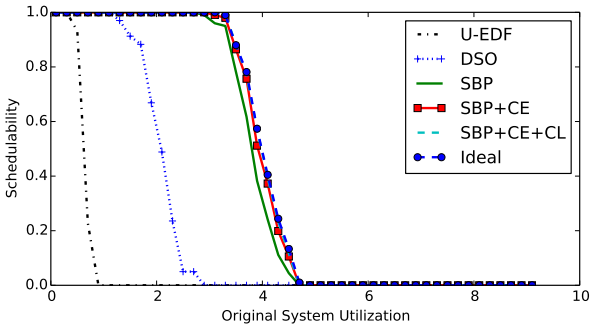
All-Mod., Cont., Light, Light, Light, Large



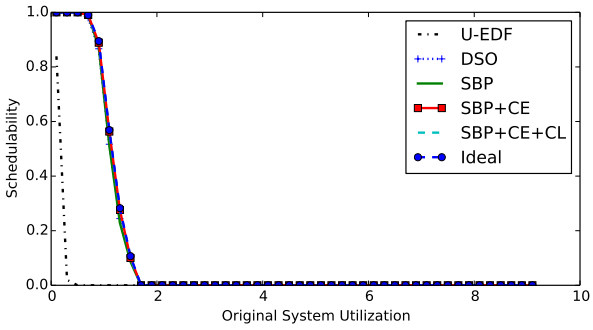
A-Heavy, Short, Heavy, Light, Heavy, Large



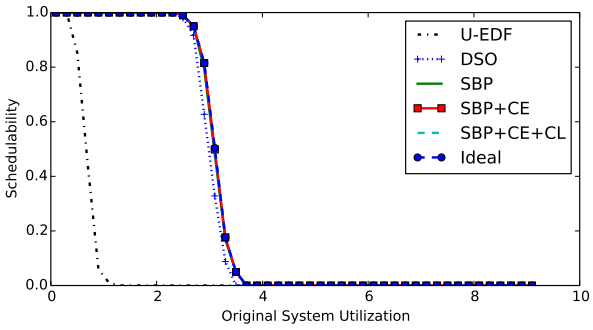
AB-Mod., Cont., Heavy, Heavy, Heavy, Small



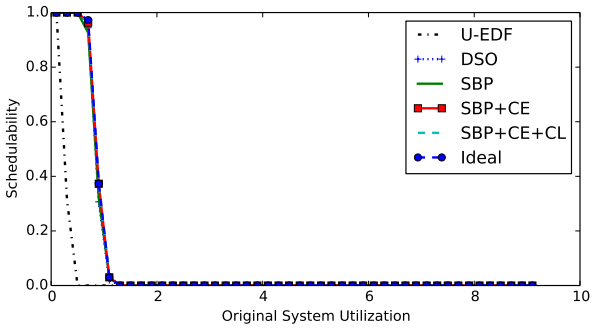
A-Heavy, Long, Heavy, Heavy, Light, Large



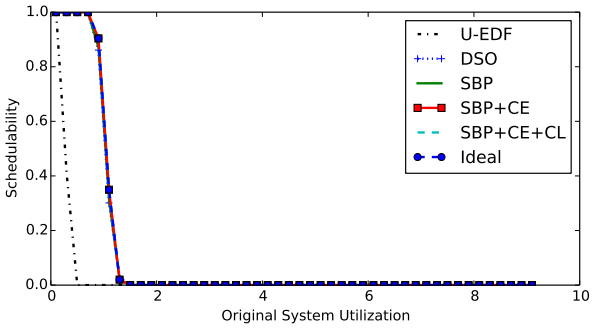
C-Heavy, Cont., Light, Heavy, Heavy, Large



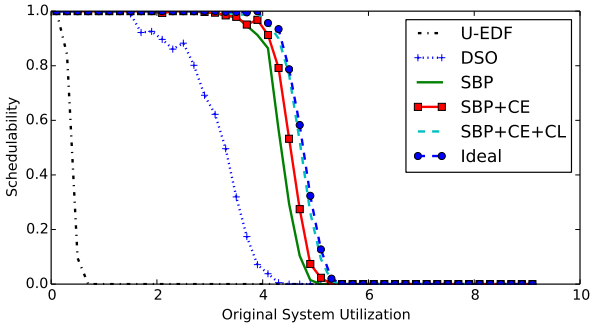
BC-Mod., Long, Light, Light, Light, Large



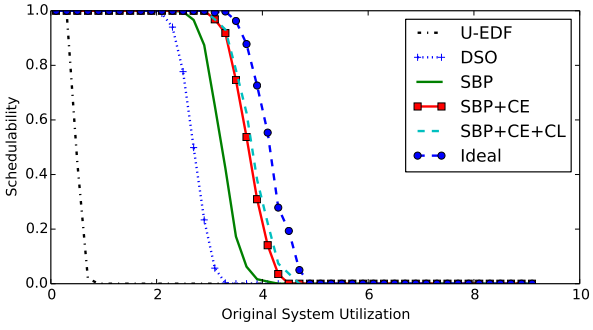
A-Heavy, Cont., Light, Light, Heavy, Large



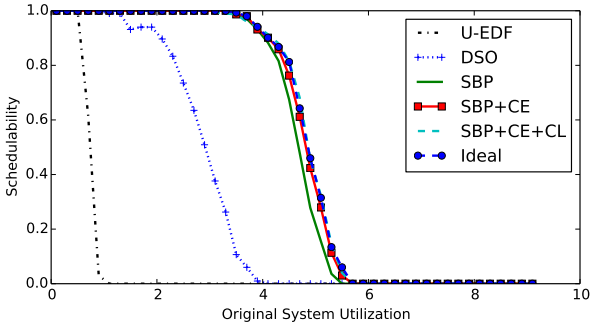
AB-Mod., Cont., Light, Light, Light, Small



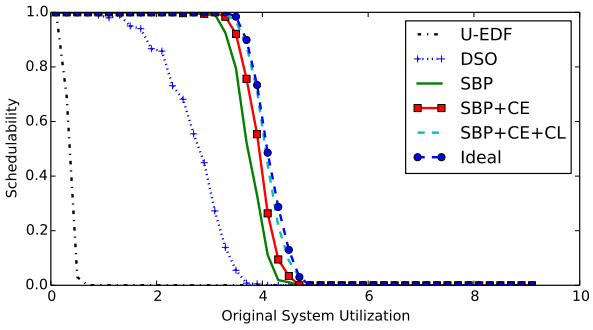
All-Mod., Cont., Heavy, Mod., Heavy, Small



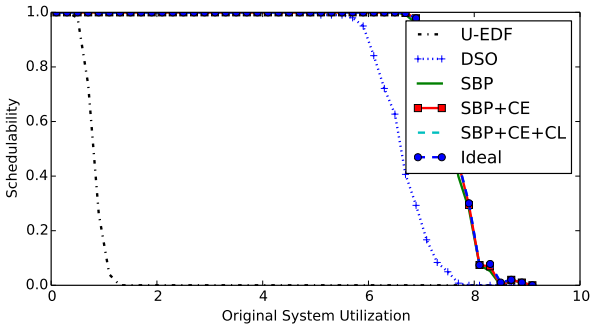
A-Heavy, Long, Mod., Heavy, Heavy, Large



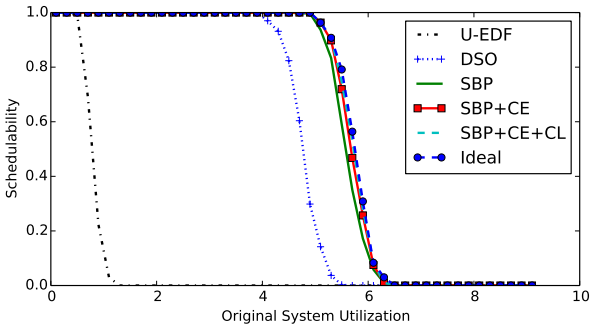
All-Mod., Long, Heavy, Mod., Light, Small



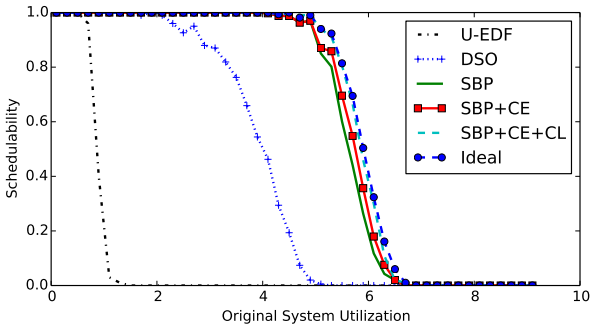
AB-Mod., Cont., Heavy, Mod., Heavy, Small



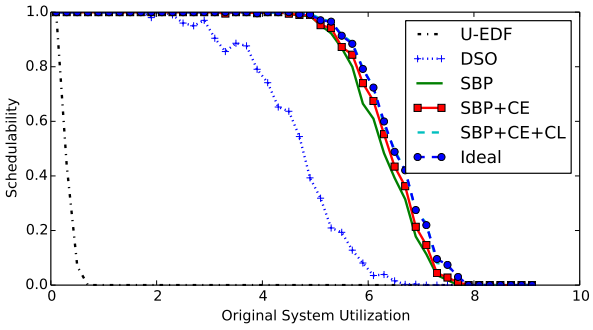
C-Heavy, Long, Mod., Light, Light, Small



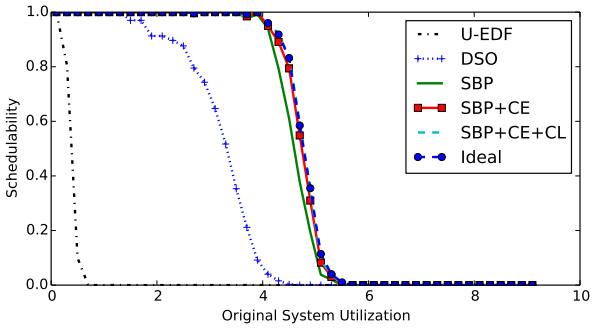
AC-Mod., Long, Mod., Light, Heavy, Large



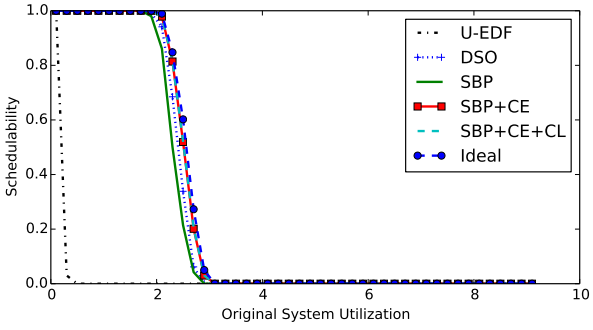
AC-Mod., Long, Heavy, Light, Heavy, Large



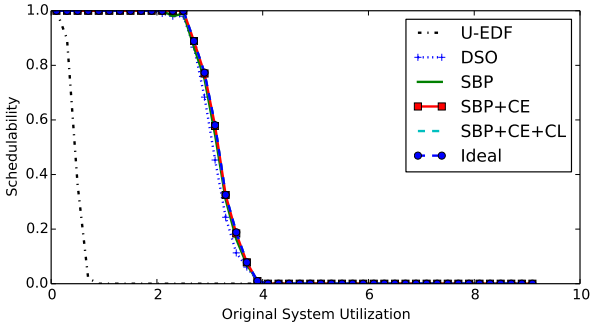
C-Heavy, Cont., Heavy, Heavy, Light, Small



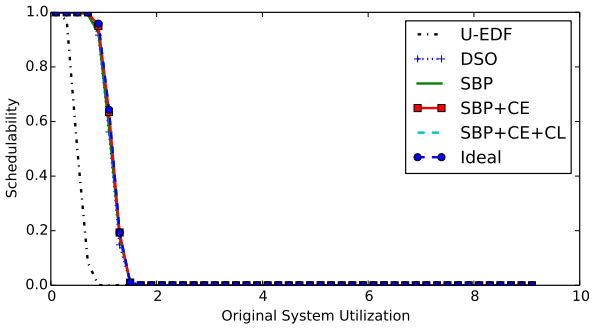
All-Mod., Cont., Heavy, Mod., Light, Small



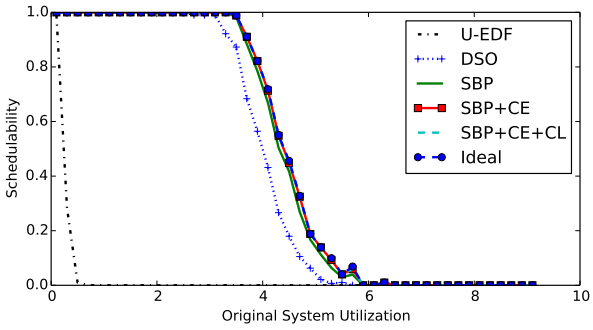
A-Heavy, Cont., Mod., Heavy, Heavy, Small



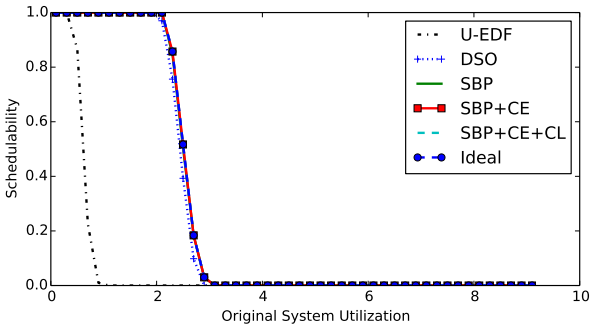
C-Heavy, Long, Light, Mod., Light, Large



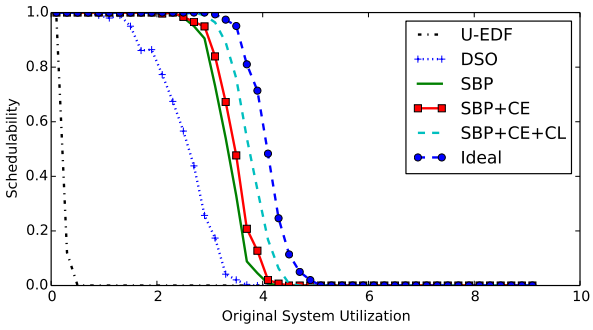
BC-Mod., Short, Light, Light, Light, Large



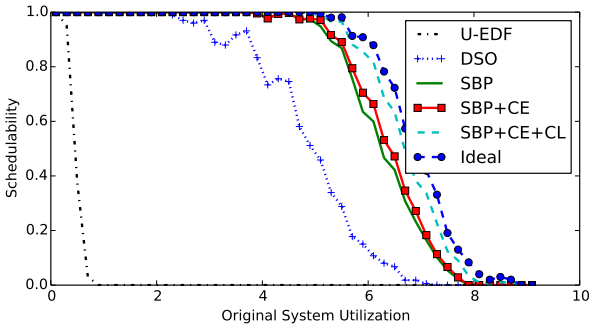
C-Heavy, Cont., Mod., Heavy, Light, Small



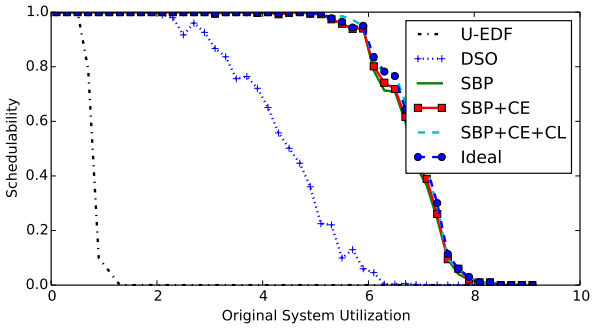
AC-Mod., Long, Light, Light, Heavy, Small



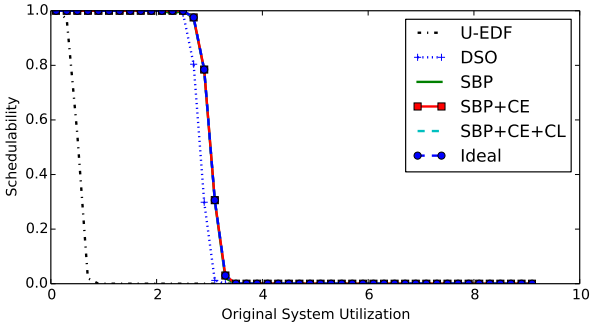
AB-Mod., Cont., Heavy, Heavy, Heavy, Large



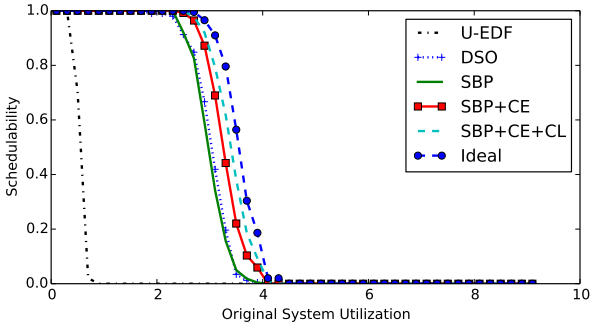
C-Heavy, Cont., Heavy, Mod., Heavy, Large



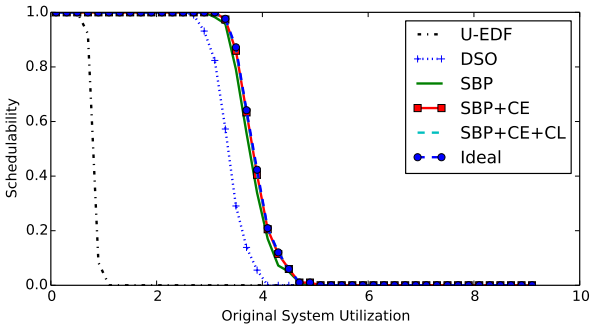
C-Heavy, Long, Heavy, Mod., Light, Large



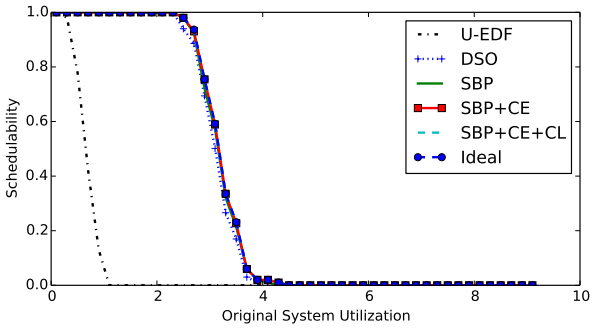
B-Heavy, Long, Light, Mod., Light, Small



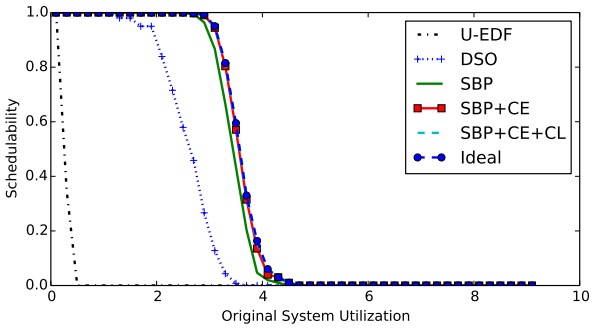
A-Heavy, Short, Heavy, Heavy, Heavy, Small



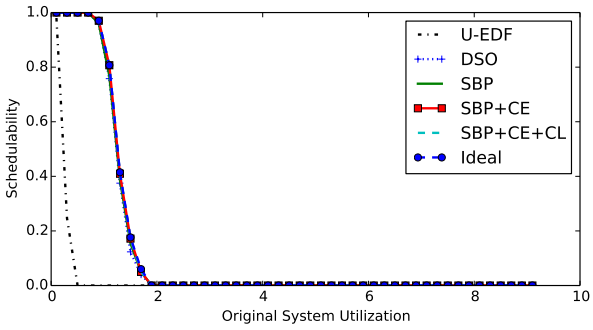
A-Heavy, Short, Heavy, Light, Light, Large



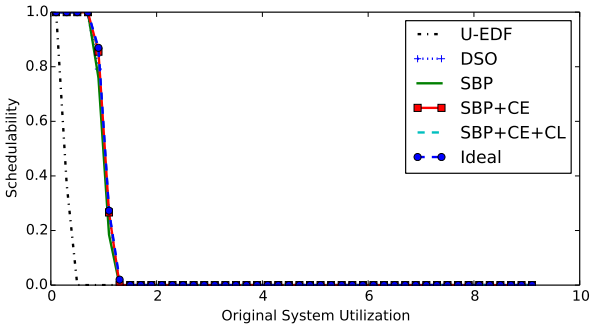
C-Heavy, Long, Light, Light, Heavy, Large



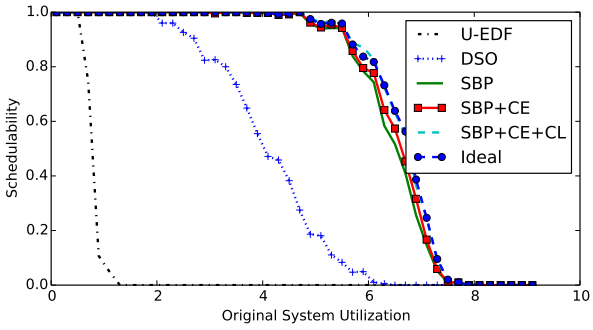
A-Heavy, Cont., Heavy, Heavy, Light, Small



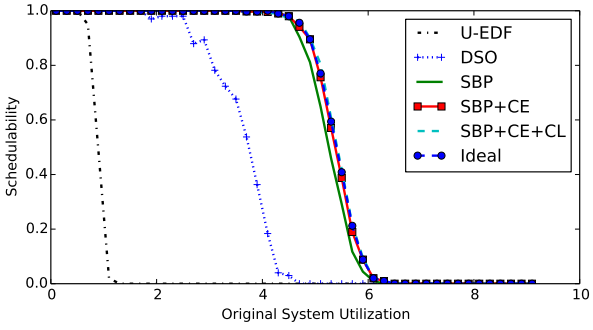
BC-Mod., Cont., Light, Mod., Light, Large



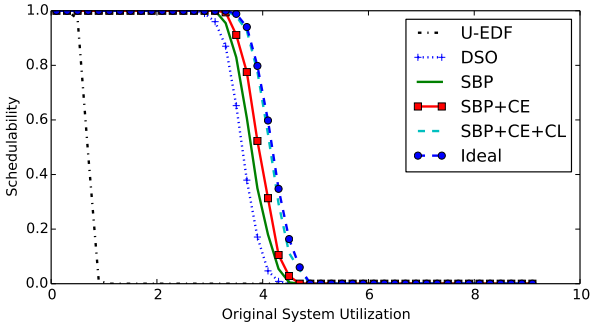
AB-Mod., Short, Light, Heavy, Heavy, Large



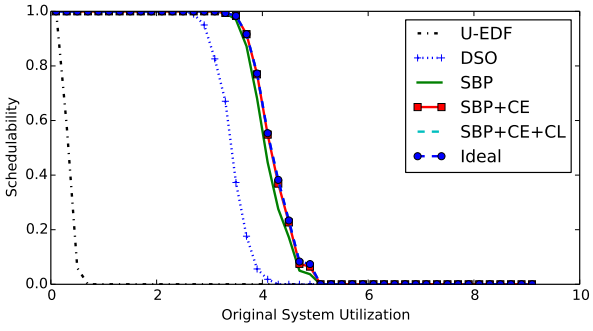
C-Heavy, Long, Heavy, Heavy, Light, Small



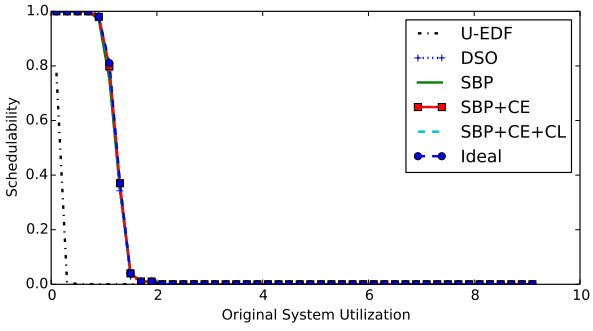
All-Mod., Long, Heavy, Light, Light, Large



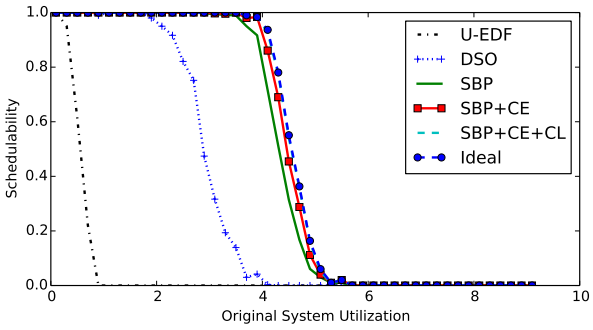
AB-Mod., Short, Heavy, Mod., Heavy, Large



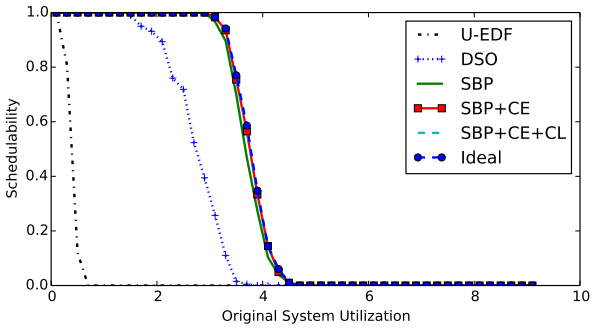
BC-Mod., Cont., Mod., Mod., Light, Small



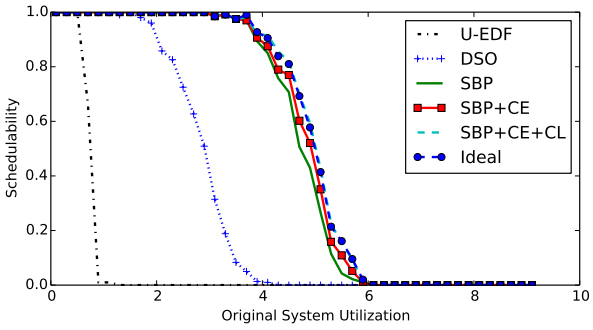
BC-Mod., Cont., Light, Heavy, Heavy, Small



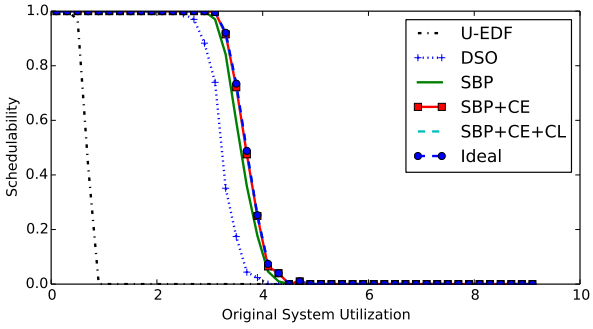
B-Heavy, Cont., Heavy, Light, Heavy, Large



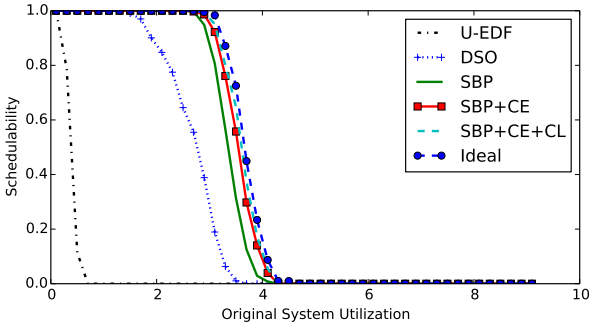
A-Heavy, Cont., Heavy, Mod., Light, Large



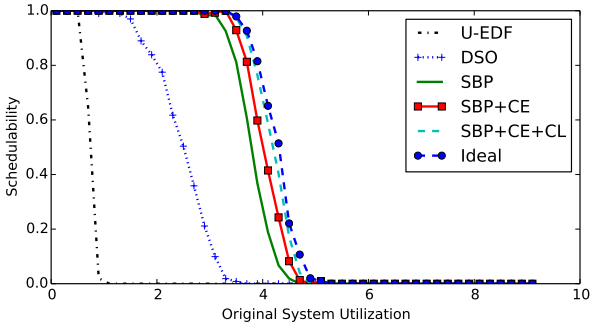
BC-Mod., Long, Heavy, Heavy, Light, Small



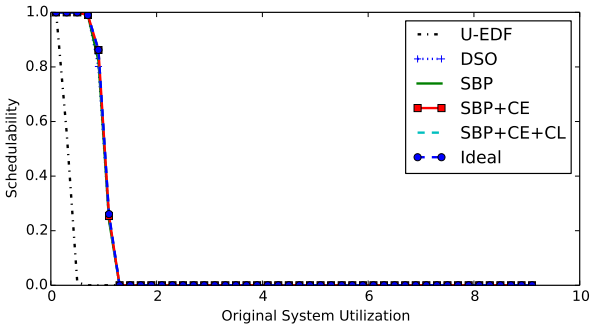
A-Heavy, Short, Heavy, Mod., Light, Small



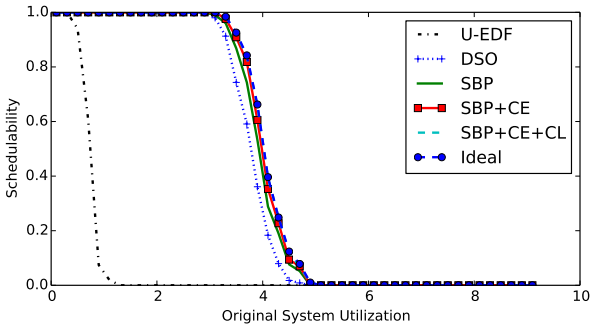
A-Heavy, Cont., Heavy, Mod., Heavy, Small



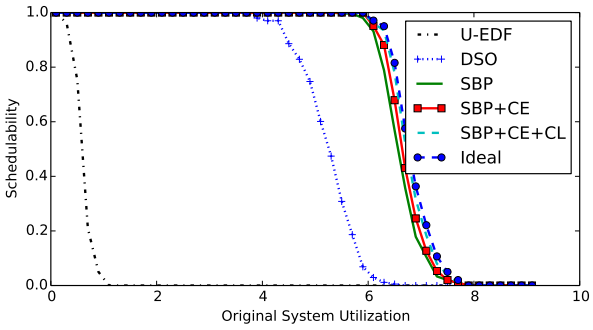
AB-Mod., Long, Heavy, Mod., Heavy, Small



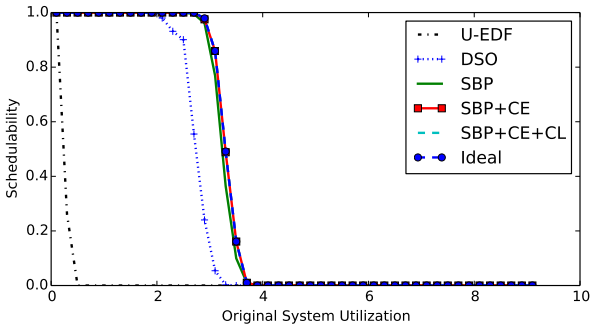
All-Mod., Cont., Light, Light, Light, Small



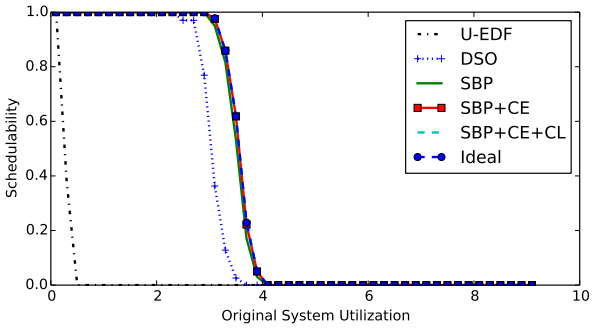
BC-Mod., Short, Mod., Light, Heavy, Large



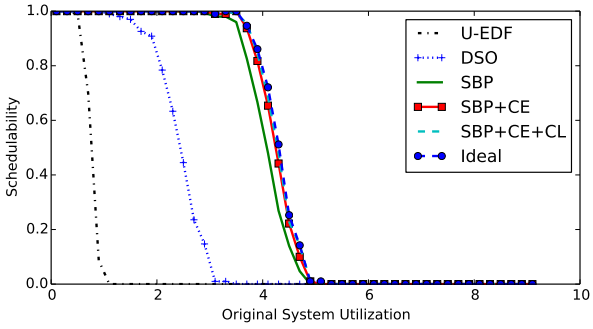
C-Heavy, Long, Mod., Heavy, Light, Large



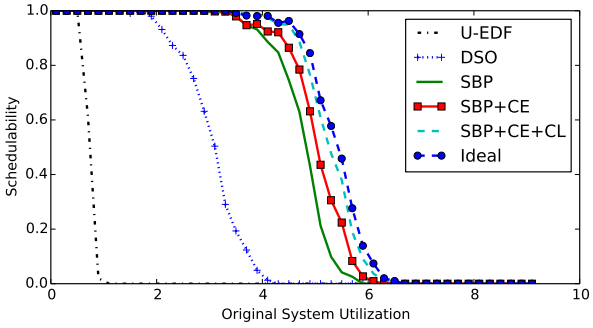
AB-Mod., Cont., Mod., Mod., Light, Small



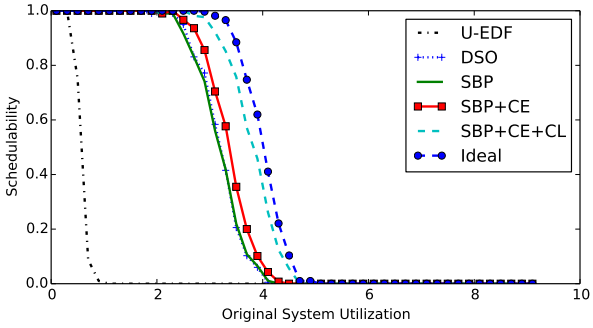
All-Mod., Cont., Mod., Mod., Light, Small



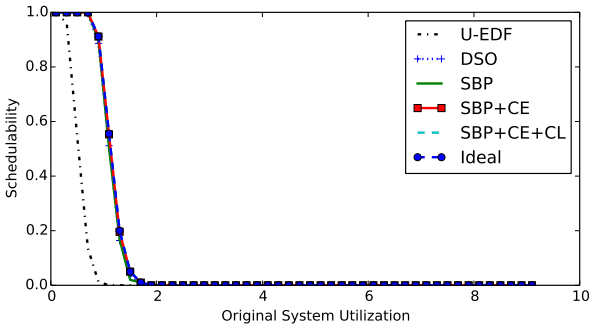
B-Heavy, Long, Heavy, Mod., Light, Large



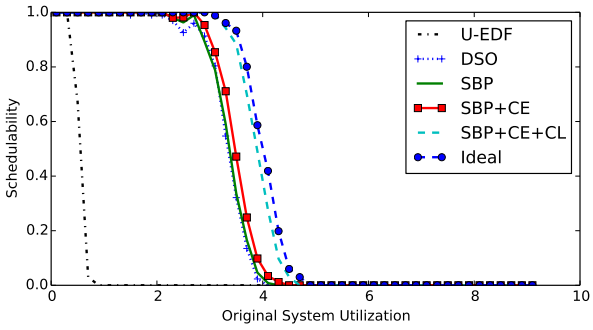
AC-Mod., Long, Heavy, Mod., Heavy, Small



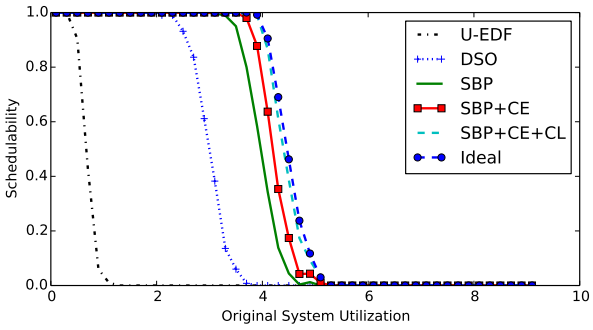
B-Heavy, Short, Heavy, Heavy, Heavy, Large



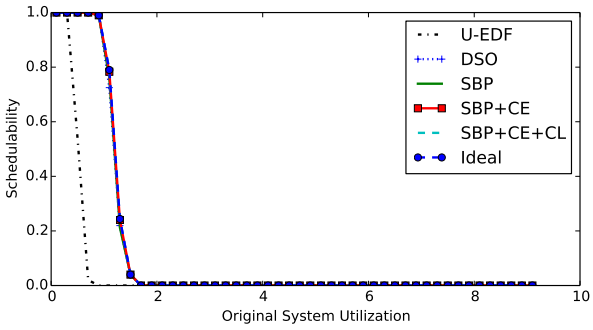
C-Heavy, Short, Light, Light, Heavy, Large



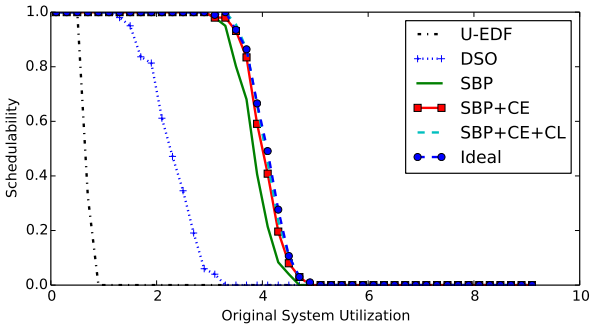
AB-Mod., Short, Heavy, Heavy, Heavy, Large



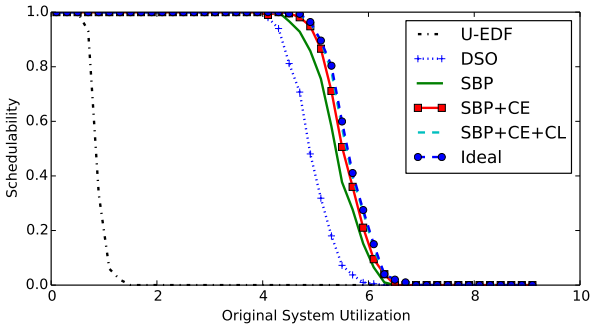
B-Heavy, Long, Mod., Mod., Heavy, Large



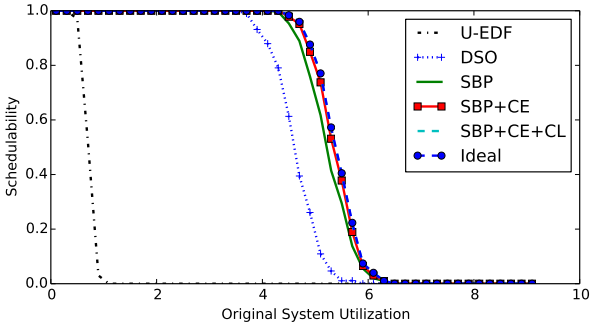
B-Heavy, Short, Light, Light, Heavy, Small



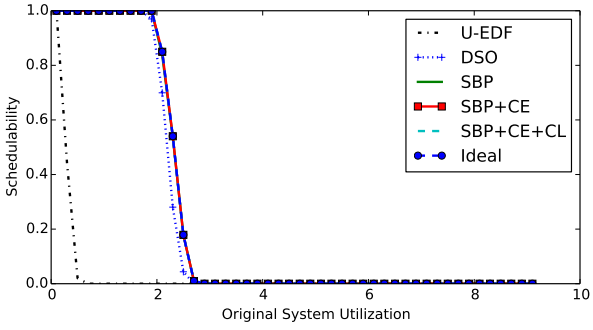
AB-Mod., Long, Heavy, Heavy, Light, Small



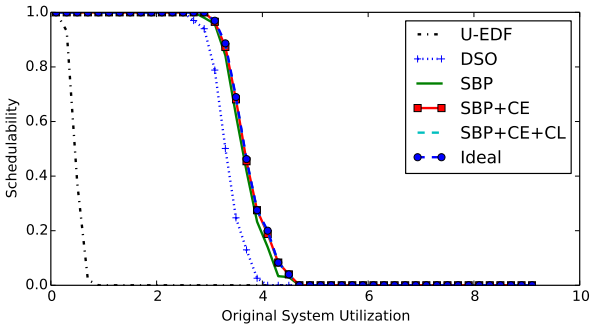
BC-Mod., Short, Heavy, Light, Heavy, Large



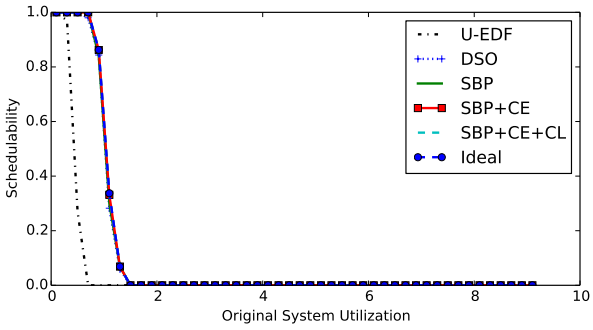
BC-Mod., Short, Heavy, Mod., Light, Large



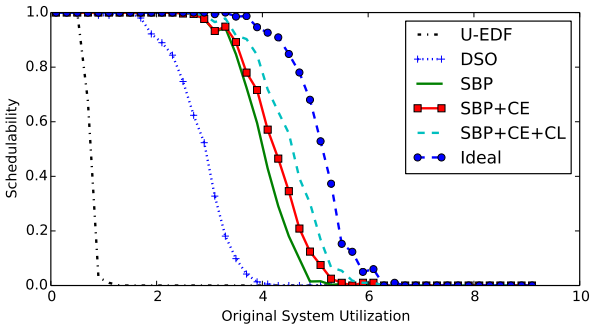
A-Heavy, Long, Light, Heavy, Light, Small



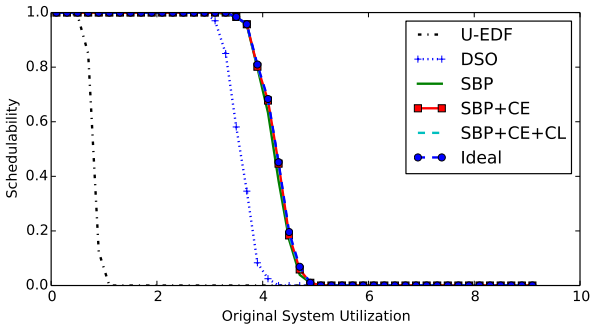
All-Mod., Cont., Mod., Light, Light, Large



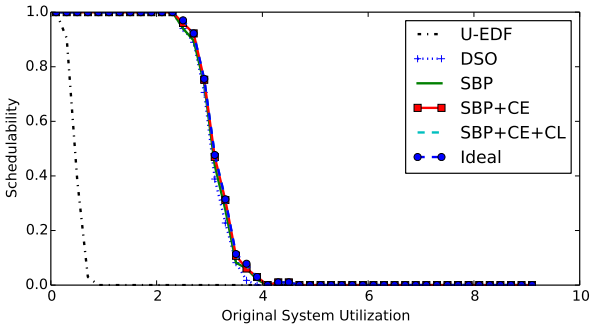
AB-Mod., Short, Light, Light, Heavy, Large



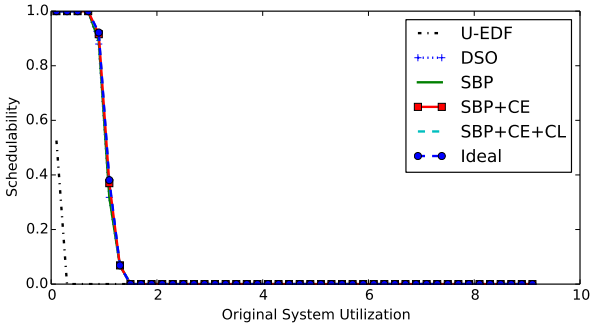
BC-Mod., Long, Heavy, Heavy, Heavy, Small



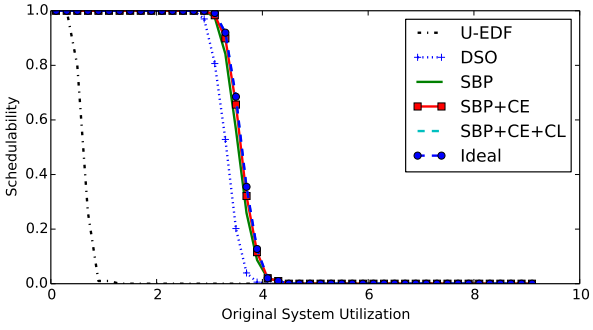
A-Heavy, Long, Mod., Light, Heavy, Small



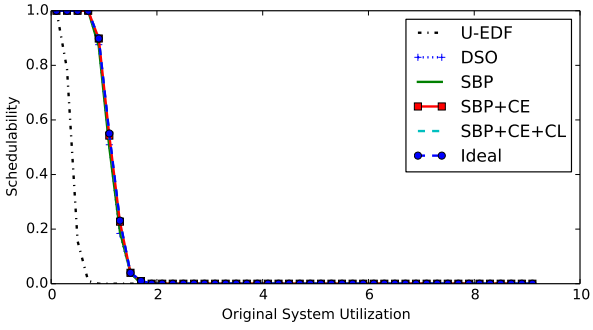
C-Heavy, Long, Light, Mod., Heavy, Large



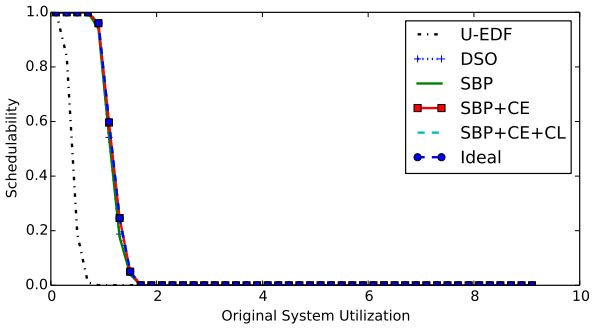
AB-Mod., Cont., Light, Heavy, Light, Large



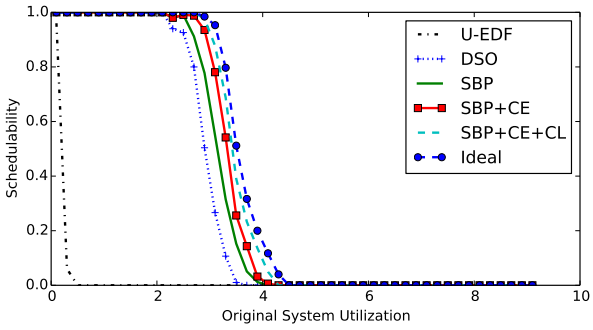
B-Heavy, Short, Mod., Mod., Light, Large



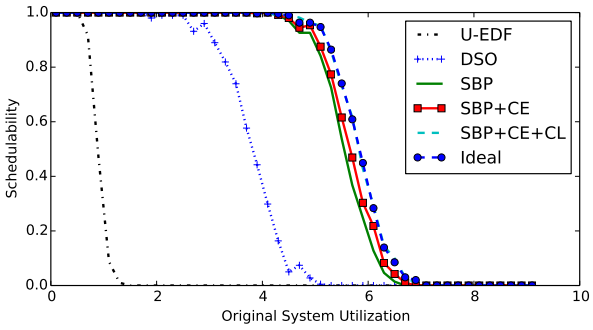
C-Heavy, Short, Light, Mod., Heavy, Large



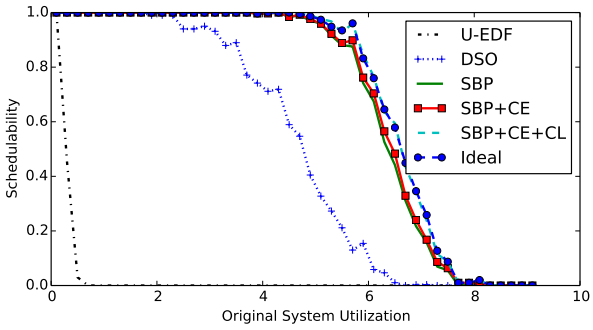
C-Heavy, Cont., Light, Light, Heavy, Small



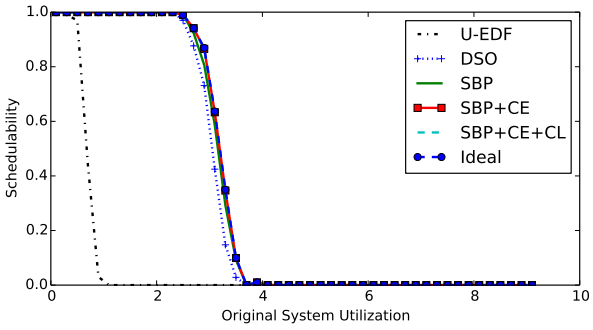
All-Mod., Cont., Mod., Heavy, Heavy, Large



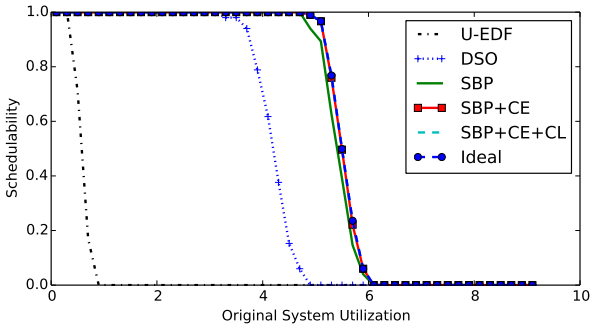
BC-Mod., Long, Heavy, Light, Heavy, Large



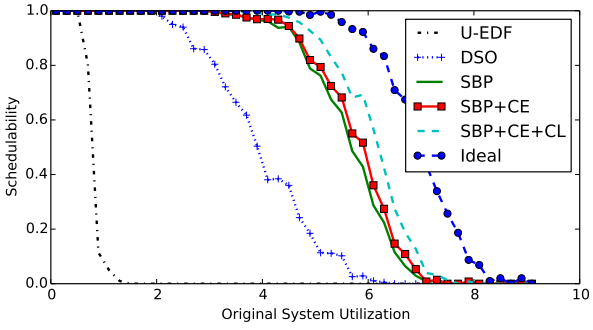
C-Heavy, Cont., Heavy, Heavy, Light, Large



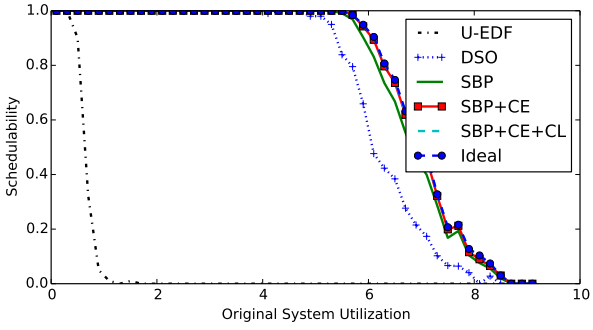
AC-Mod., Short, Mod., Light, Heavy, Small



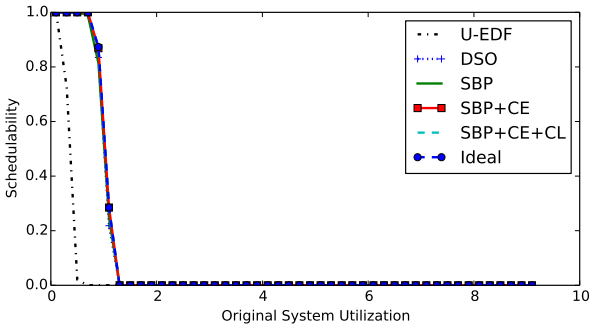
AC-Mod., Long, Mod., Mod., Light, Small



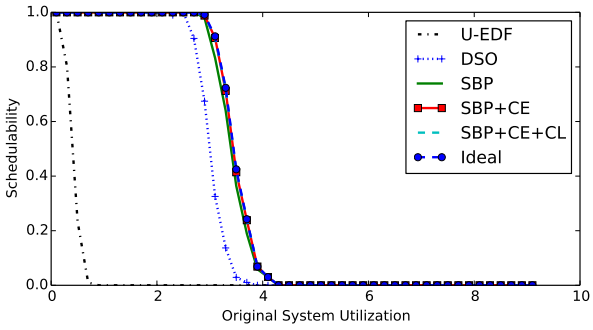
C-Heavy, Long, Heavy, Heavy, Heavy, Large



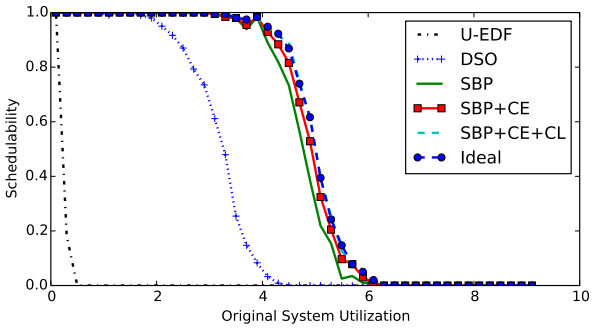
C-Heavy, Short, Heavy, Mod., Light, Small



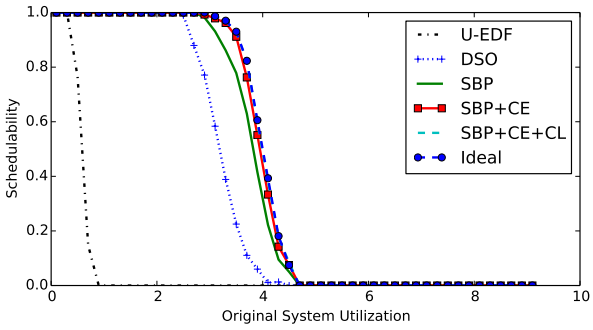
All-Mod., Short, Light, Mod., Light, Small



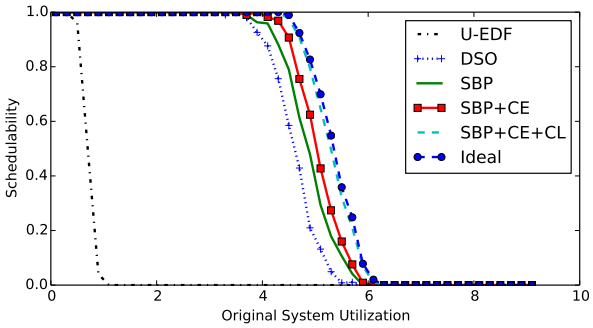
AB-Mod., Cont., Mod., Light, Light, Large



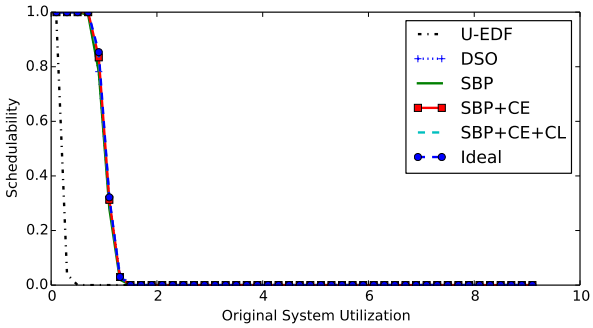
BC-Mod., Cont., Heavy, Heavy, Light, Small



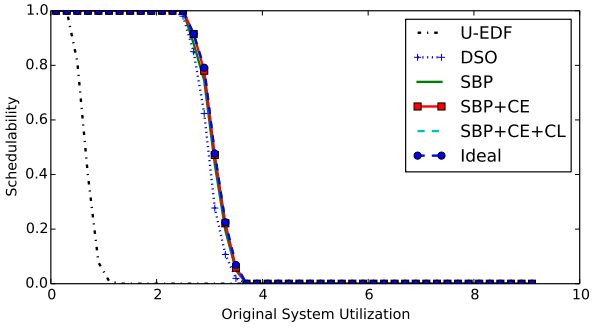
B-Heavy, Short, Heavy, Heavy, Light, Small



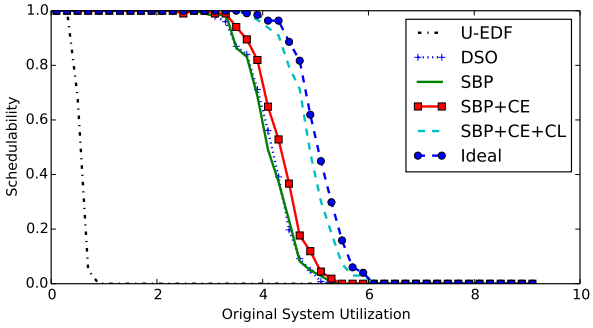
BC-Mod., Short, Heavy, Mod., Heavy, Large



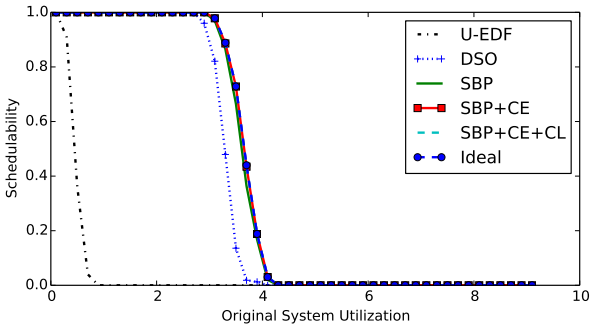
All-Mod., Cont., Light, Mod., Heavy, Large



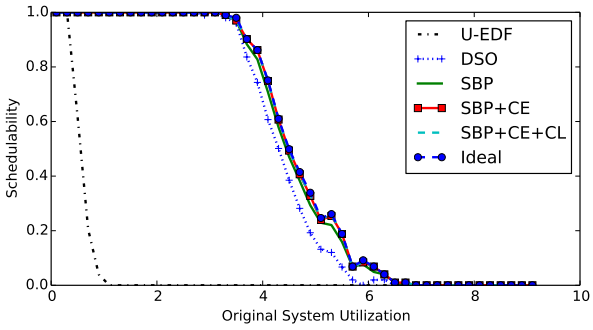
BC-Mod., Long, Light, Light, Heavy, Large



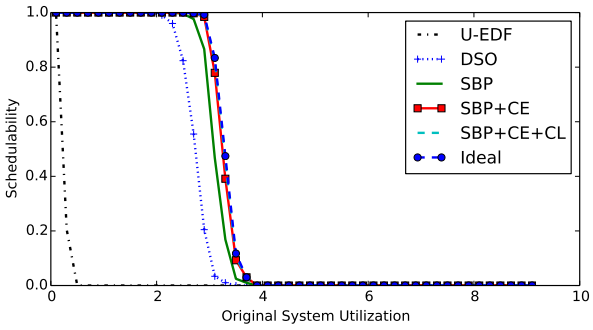
BC-Mod., Short, Heavy, Heavy, Heavy, Large



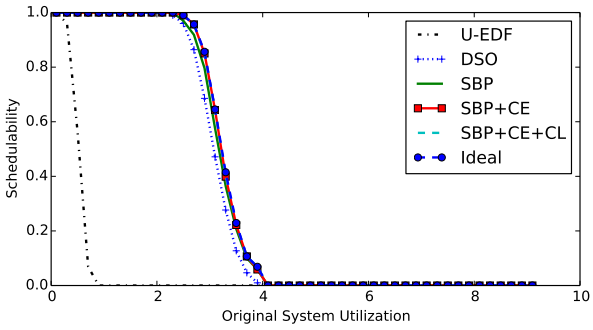
All-Mod., Cont., Mod., Light, Light, Small



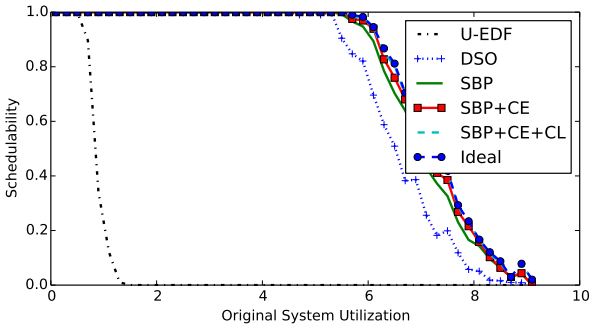
C-Heavy, Cont., Mod., Light, Light, Large



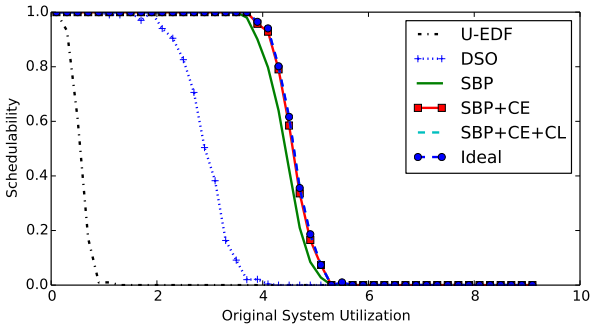
AB-Mod., Cont., Mod., Mod., Heavy, Small



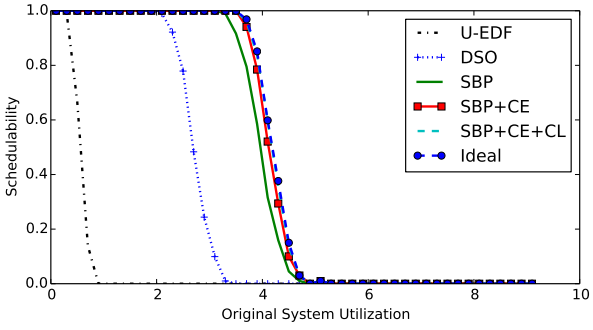
AC-Mod., Short, Mod., Mod., Light, Large



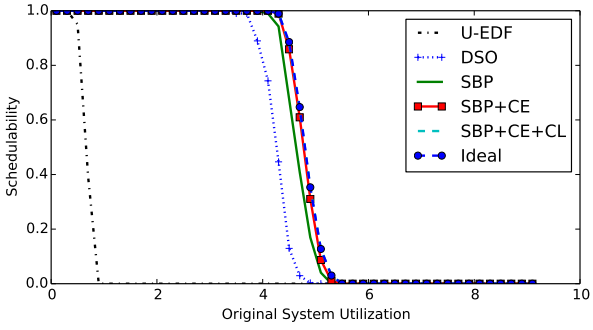
C-Heavy, Short, Heavy, Light, Heavy, Large



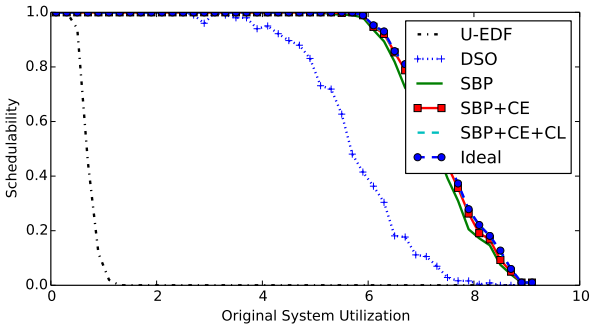
B-Heavy, Cont., Heavy, Light, Light, Large



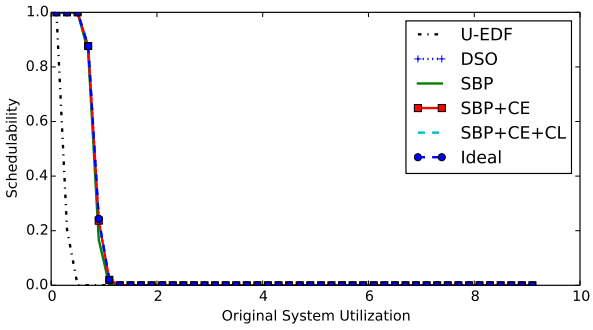
B-Heavy, Long, Mod., Heavy, Light, Large



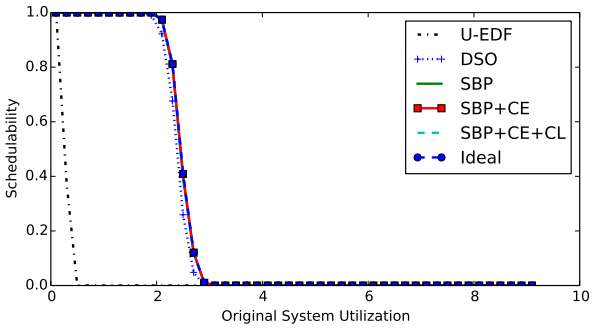
All-Mod., Short, Heavy, Mod., Light, Small



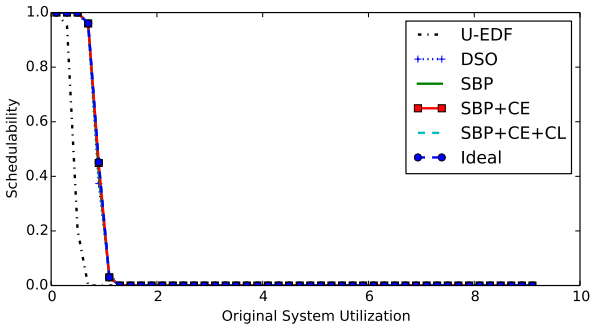
C-Heavy, Cont., Heavy, Light, Light, Large



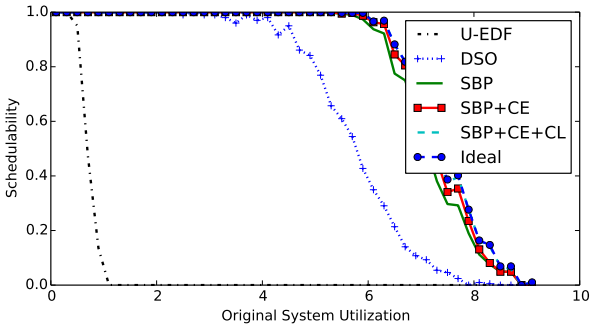
A-Heavy, Short, Light, Heavy, Heavy, Small



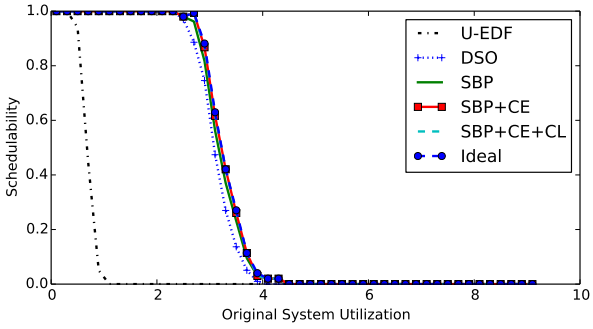
AC-Mod., Long, Light, Heavy, Light, Small



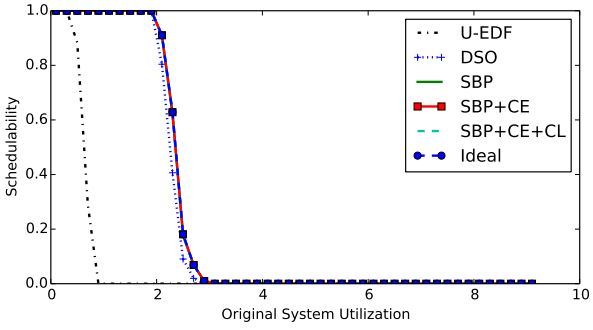
AC-Mod., Short, Light, Light, Light, Small



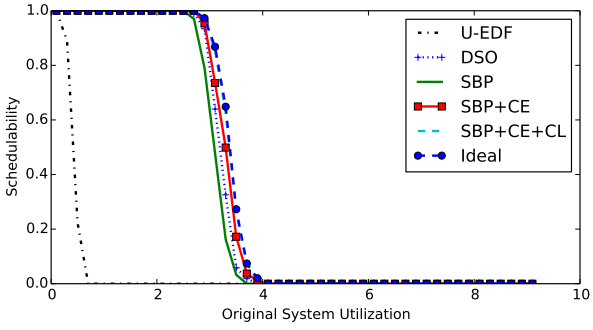
C-Heavy, Cont., Heavy, Light, Heavy, Small



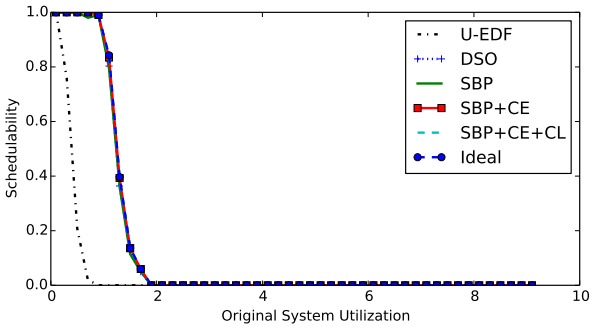
AC-Mod., Short, Mod., Light, Light, Large



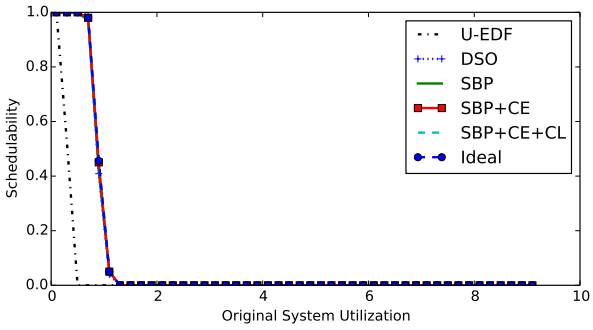
A-Heavy, Long, Light, Light, Heavy, Small



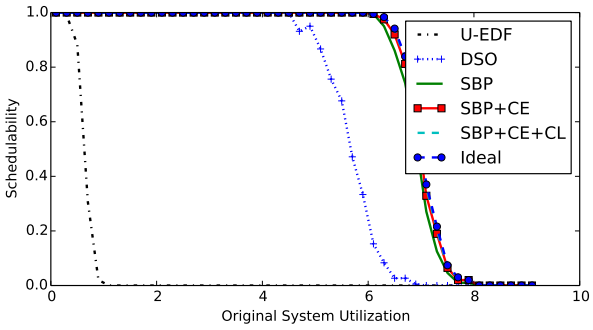
All-Mod., Short, Mod., Heavy, Heavy, Small



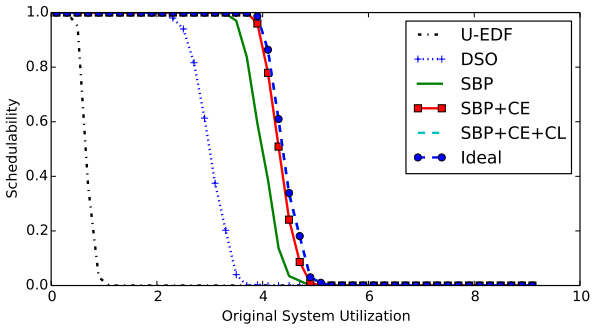
BC-Mod., Cont., Light, Light, Light, Large



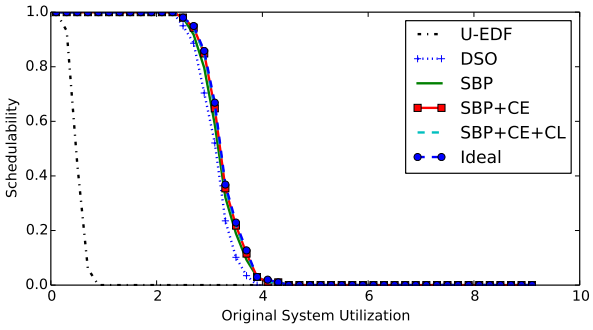
AC-Mod., Cont., Light, Light, Heavy, Small



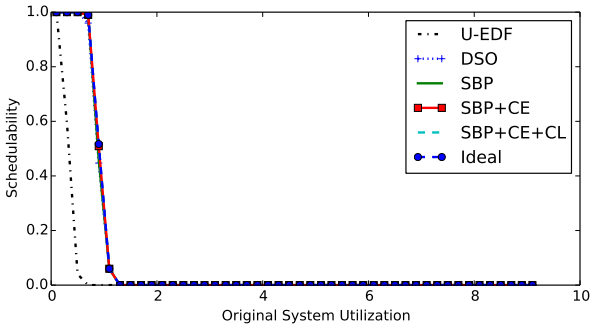
C-Heavy, Long, Mod., Mod., Light, Small



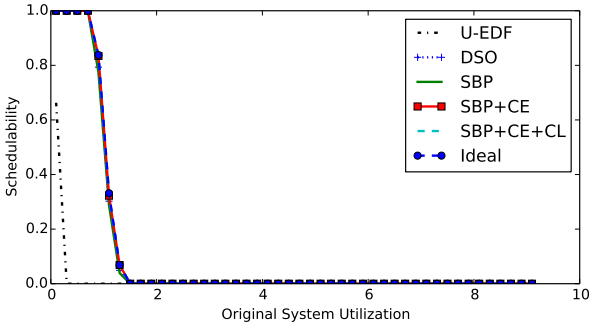
B-Heavy, Long, Mod., Mod., Heavy, Small



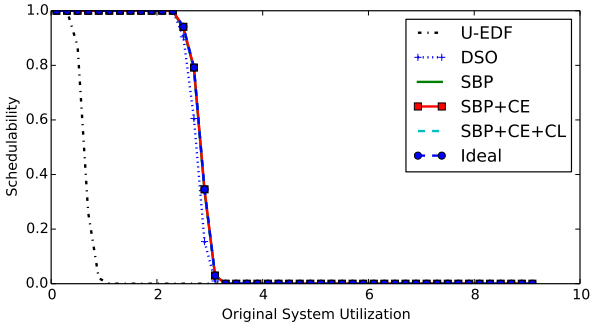
AC-Mod., Cont., Mod., Light, Heavy, Large



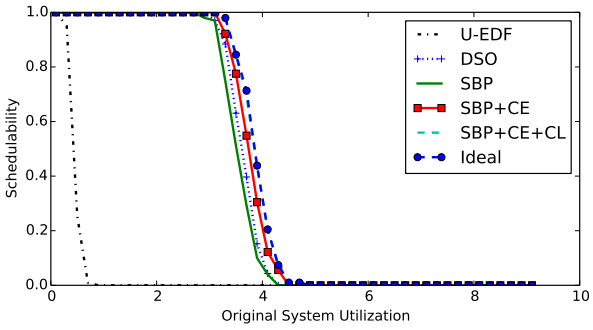
AC-Mod., Cont., Light, Light, Light, Large



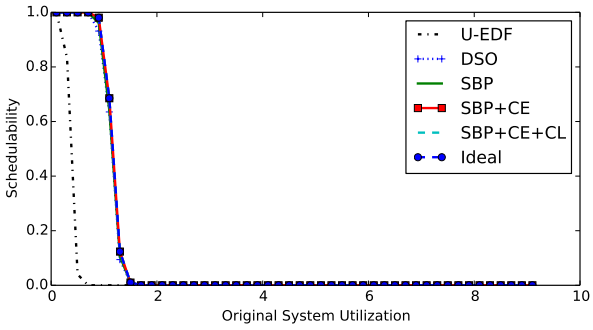
All-Mod., Cont., Light, Heavy, Light, Large



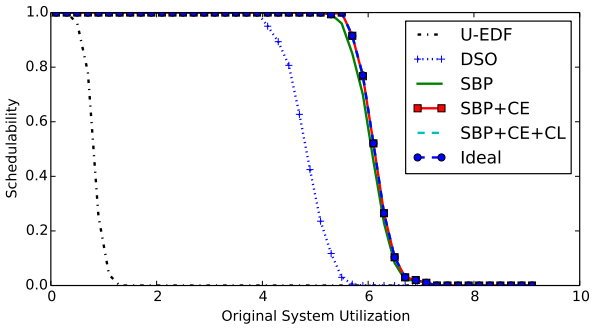
All-Mod., Long, Light, Light, Heavy, Small



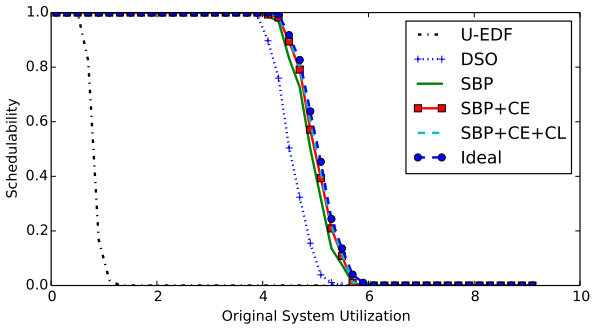
BC-Mod., Short, Mod., Heavy, Heavy, Small



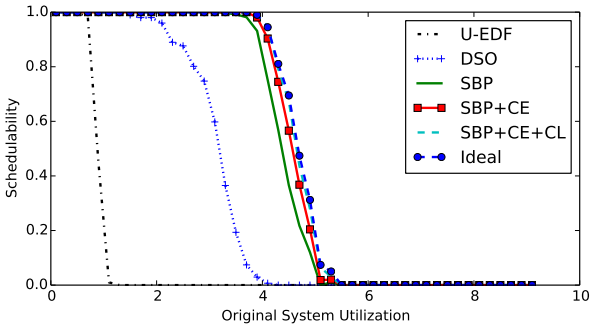
BC-Mod., Short, Light, Mod., Heavy, Small



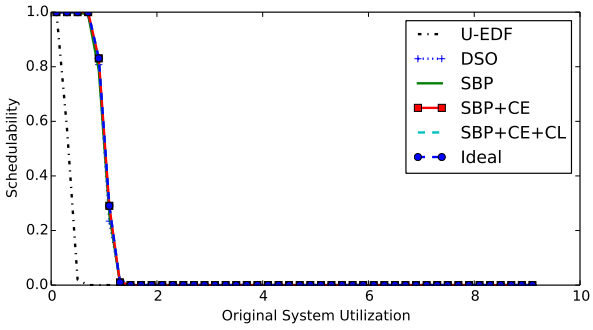
BC-Mod., Long, Mod., Light, Light, Small



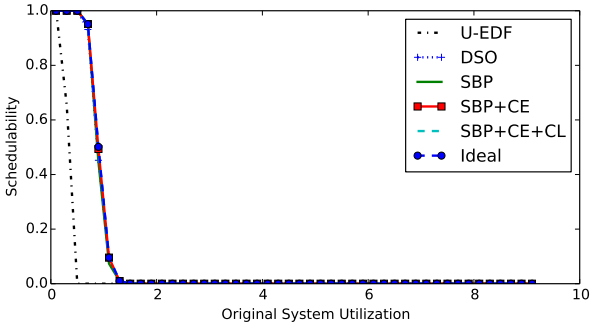
AC-Mod., Short, Heavy, Light, Heavy, Large



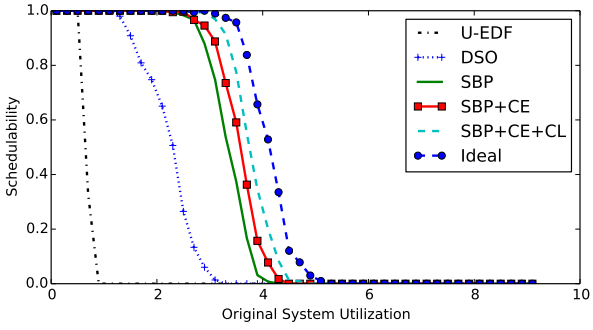
AB-Mod., Long, Heavy, Light, Heavy, Large



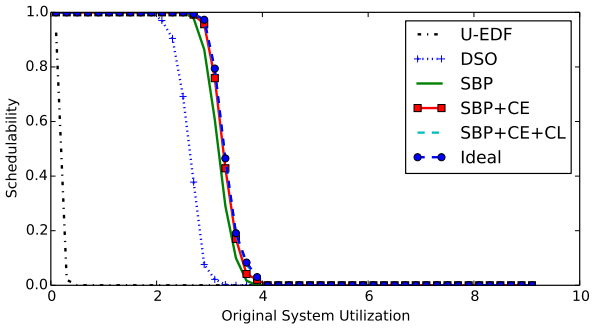
All-Mod., Cont., Light, Light, Heavy, Small



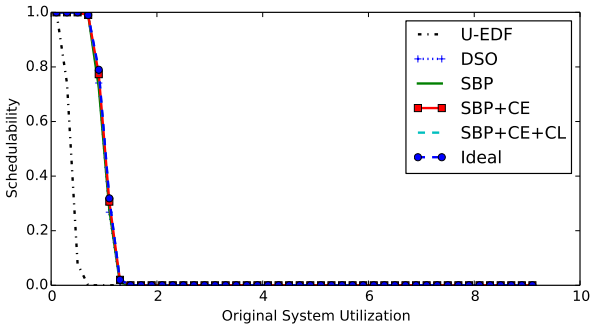
AC-Mod., Short, Light, Mod., Light, Large



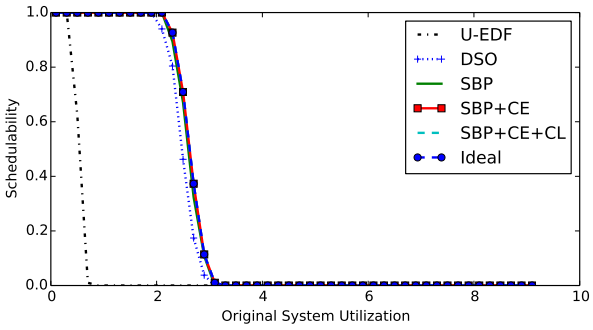
AB-Mod., Long, Heavy, Heavy, Heavy, Small



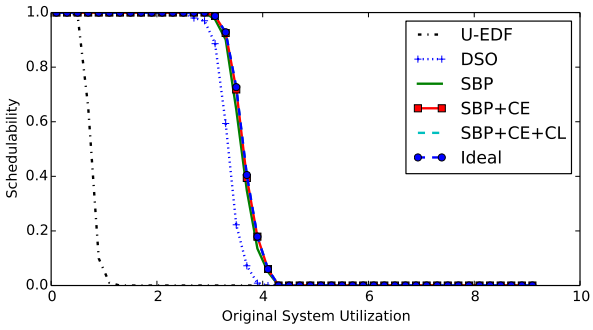
AB-Mod., Cont., Mod., Heavy, Light, Large



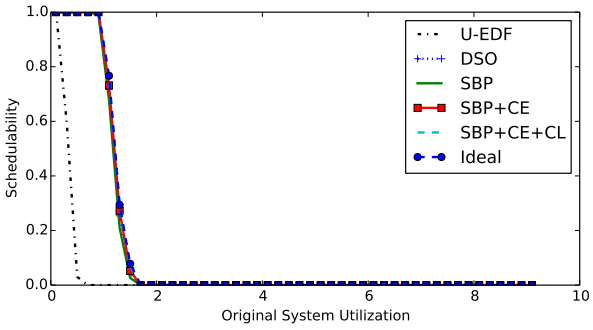
All-Mod., Short, Light, Mod., Heavy, Large



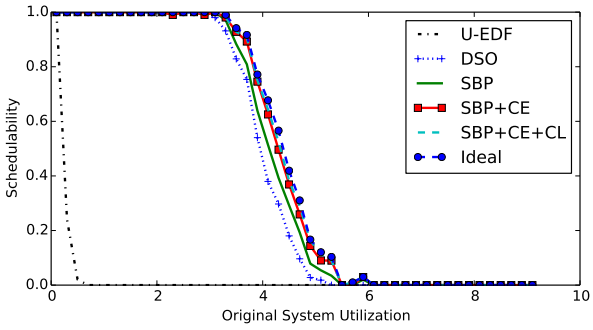
A-Heavy, Short, Mod., Mod., Light, Small



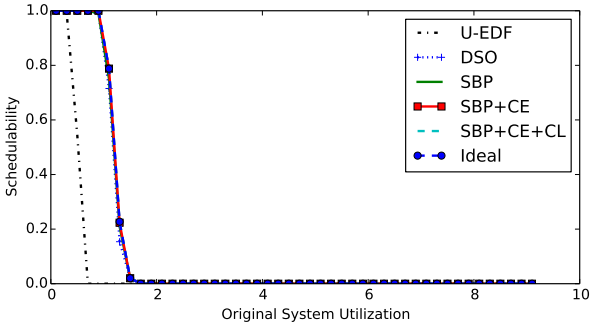
B-Heavy, Short, Mod., Light, Light, Small



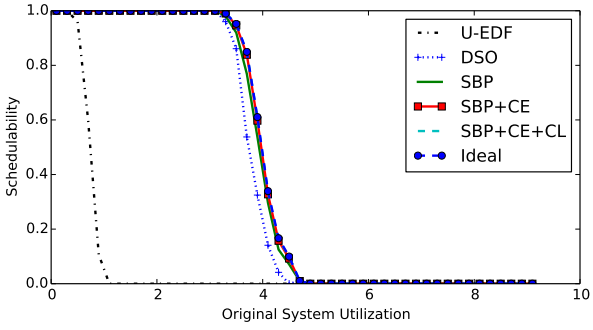
B-Heavy, Short, Light, Heavy, Heavy, Large



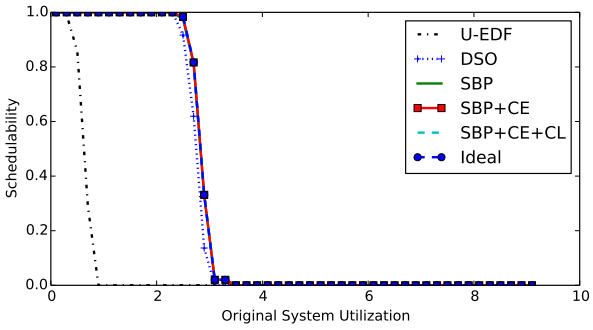
C-Heavy, Cont., Mod., Heavy, Heavy, Small



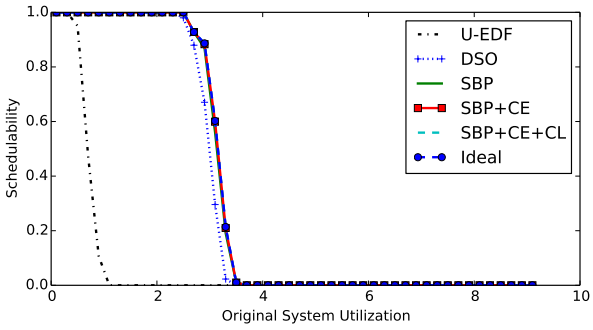
B-Heavy, Short, Light, Light, Light, Small



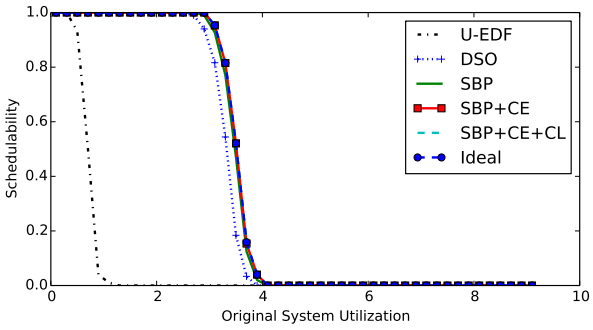
BC-Mod., Short, Mod., Light, Heavy, Small



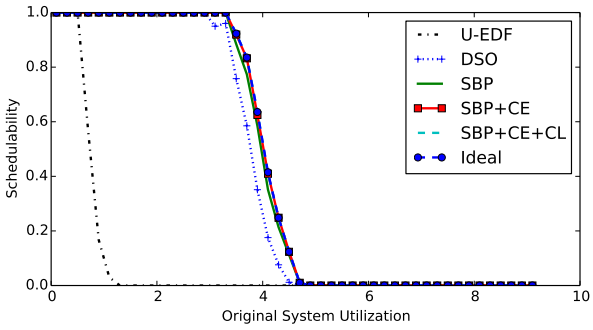
All-Mod., Long, Light, Light, Light, Small



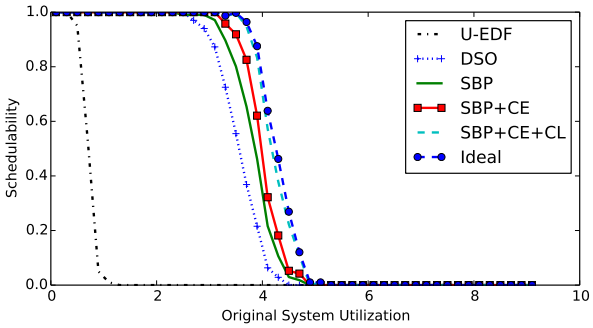
B-Heavy, Long, Light, Light, Heavy, Large



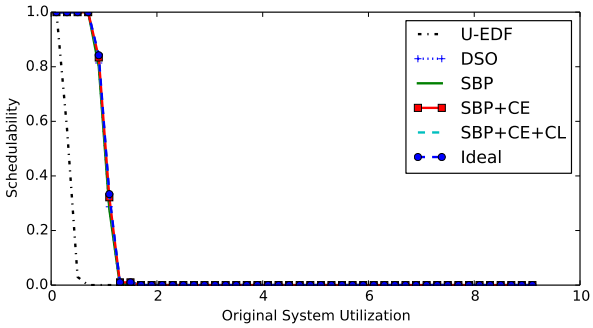
All-Mod., Short, Mod., Light, Light, Small



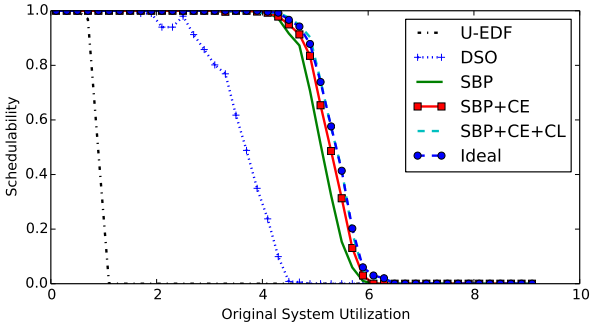
BC-Mod., Short, Mod., Light, Light, Large



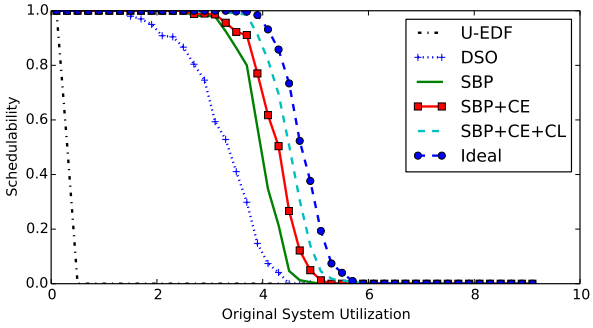
B-Heavy, Short, Heavy, Mod., Heavy, Large



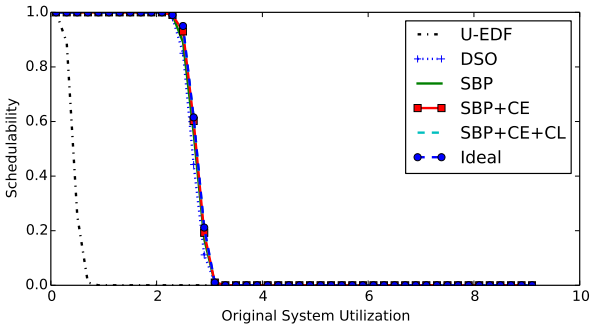
All-Mod., Cont., Light, Light, Heavy, Large



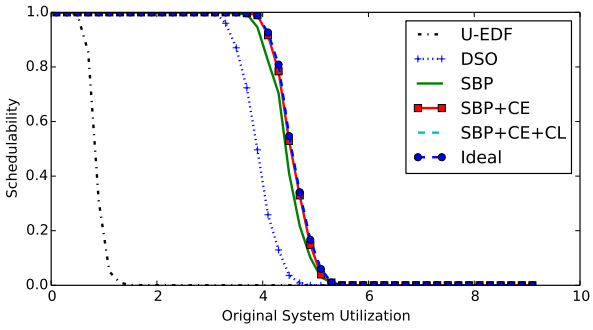
All-Mod., Long, Heavy, Light, Heavy, Small



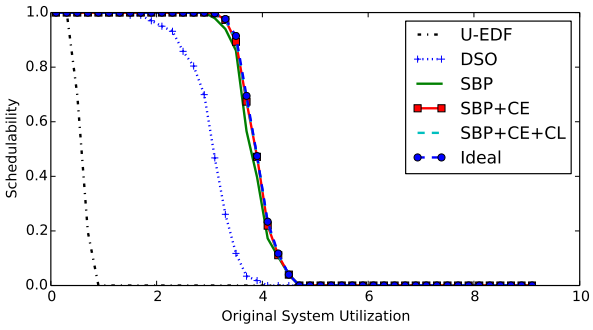
AC-Mod., Cont., Heavy, Heavy, Heavy, Small



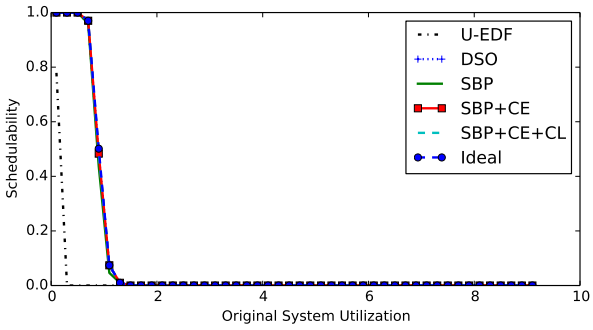
All-Mod., Long, Light, Mod., Heavy, Large



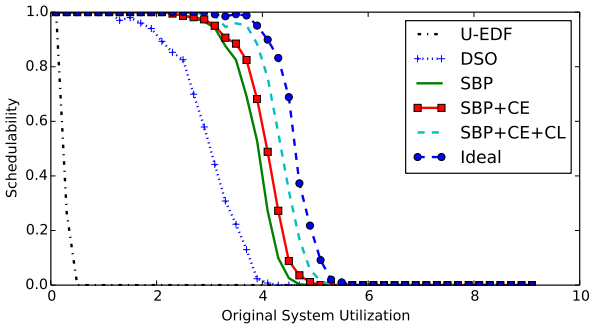
B-Heavy, Short, Heavy, Light, Light, Large



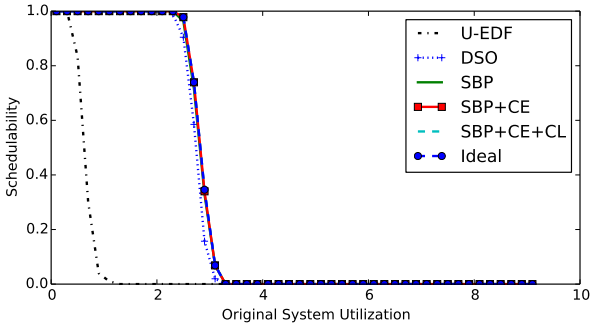
A-Heavy, Cont., Heavy, Light, Light, Large



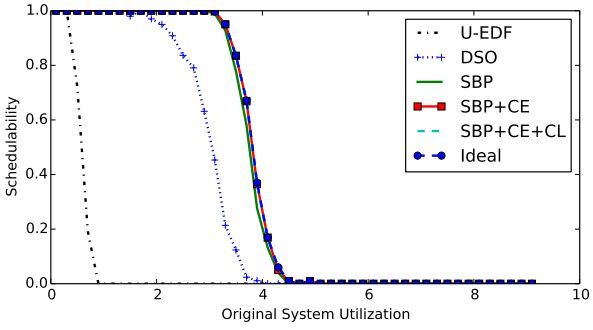
AC-Mod., Cont., Light, Heavy, Heavy, Large



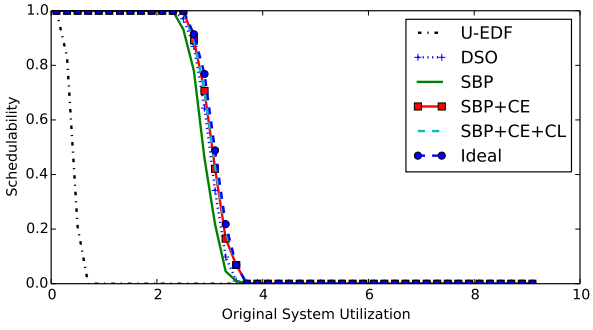
All-Mod., Cont., Heavy, Heavy, Heavy, Small



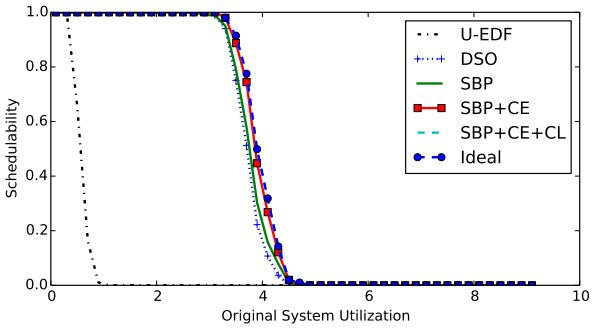
All-Mod., Long, Light, Light, Heavy, Large



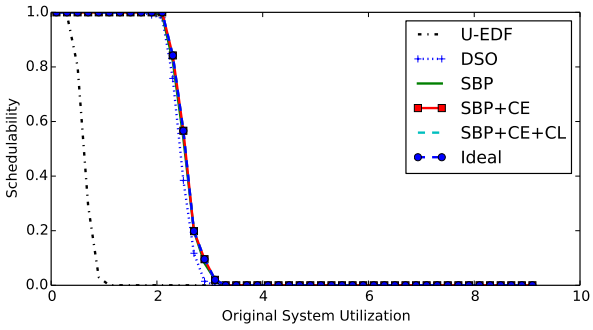
A-Heavy, Cont., Heavy, Light, Light, Small



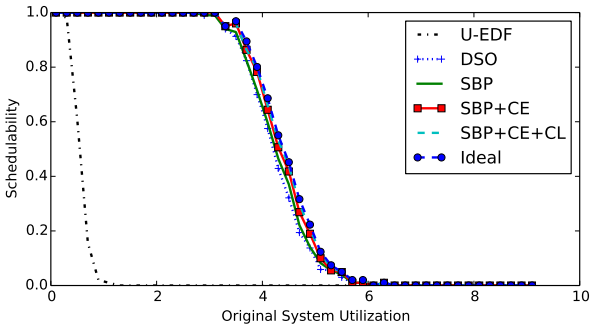
AC-Mod., Short, Mod., Heavy, Heavy, Small



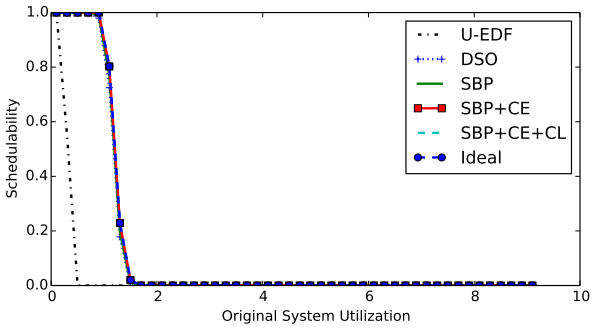
BC-Mod., Short, Mod., Mod., Heavy, Small



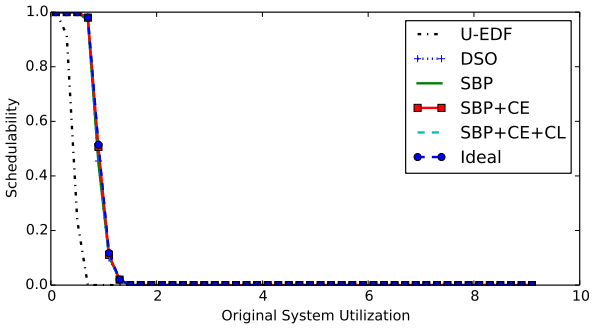
AC-Mod., Long, Light, Light, Light, Large



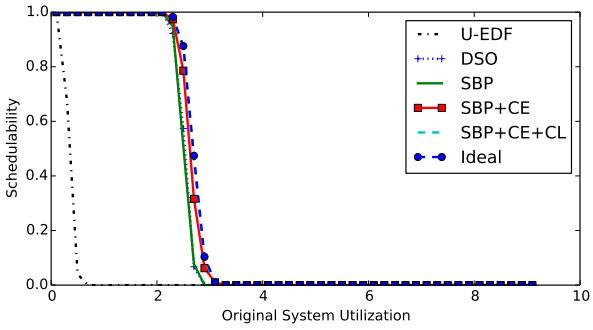
C-Heavy, Short, Mod., Mod., Heavy, Large



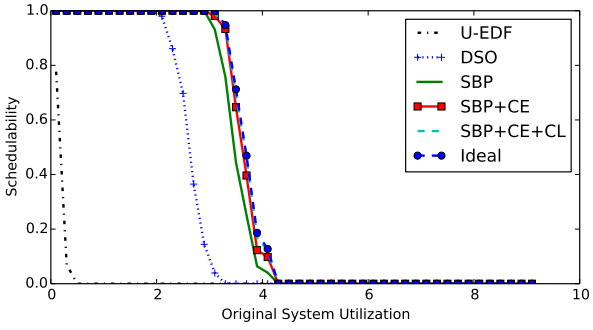
B-Heavy, Short, Light, Heavy, Light, Small



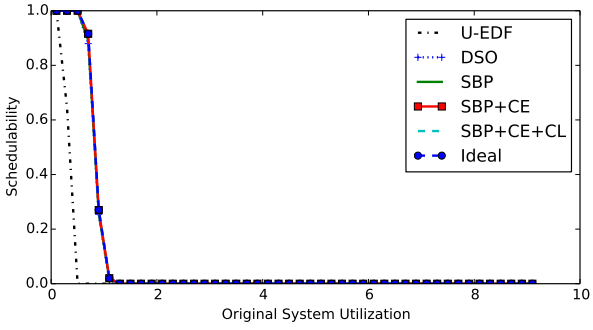
AC-Mod., Short, Light, Light, Heavy, Large



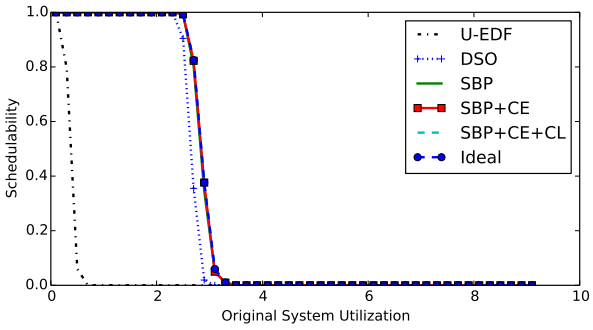
AB-Mod., Long, Light, Heavy, Heavy, Large



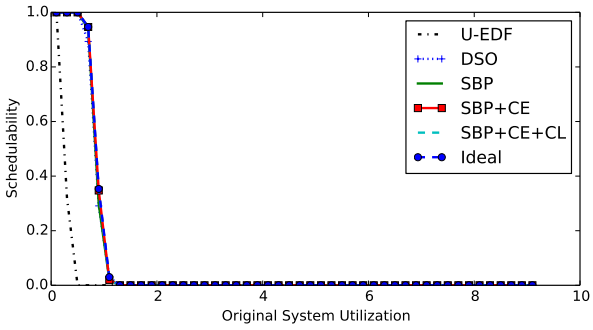
B-Heavy, Cont., Mod., Heavy, Light, Large



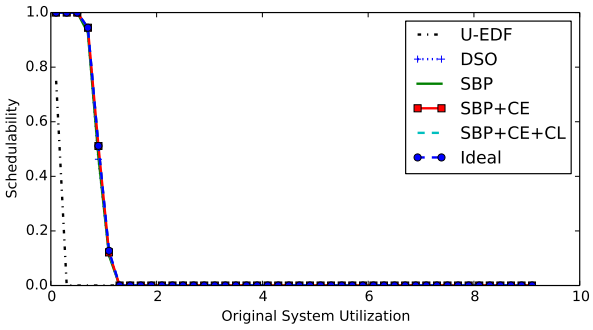
A-Heavy, Short, Light, Mod., Light, Small



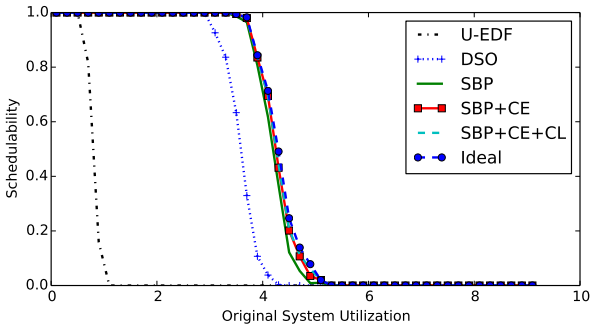
B-Heavy, Long, Light, Heavy, Light, Small



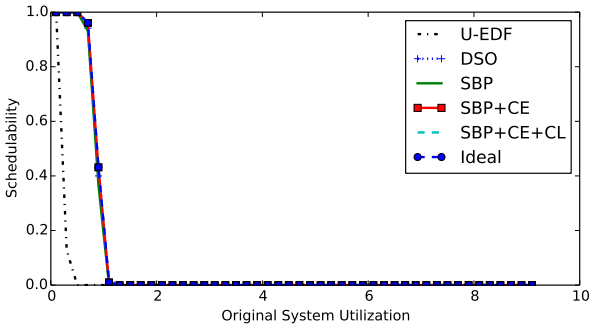
A-Heavy, Cont., Light, Light, Light, Large



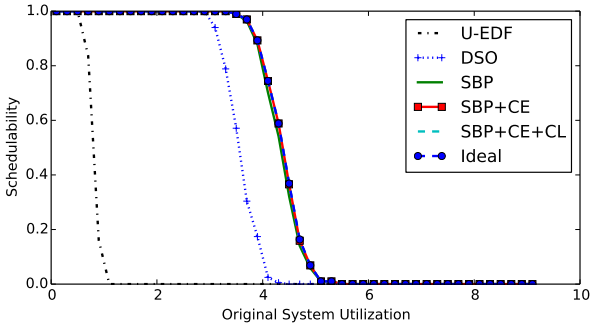
AC-Mod., Cont., Light, Heavy, Light, Large



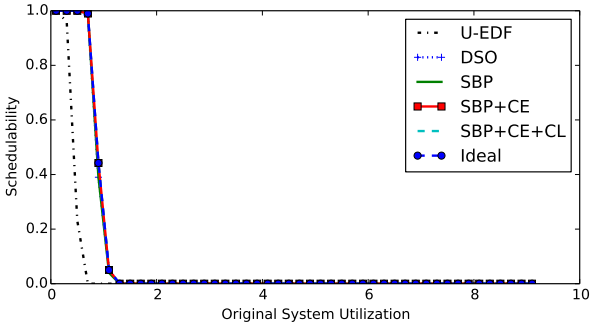
A-Heavy, Long, Mod., Light, Heavy, Large



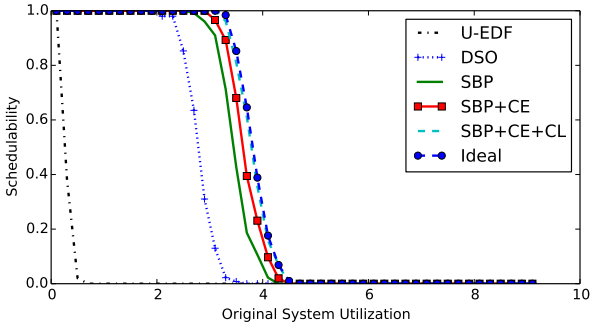
AC-Mod., Short, Light, Heavy, Heavy, Small



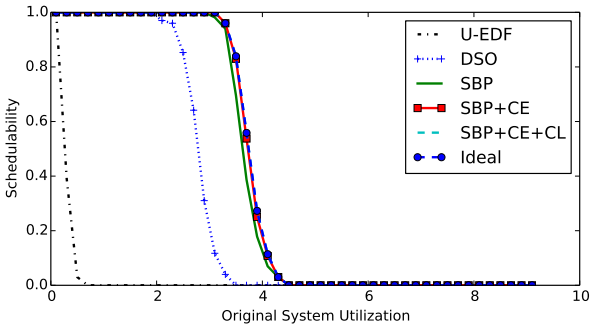
A-Heavy, Long, Mod., Light, Light, Large



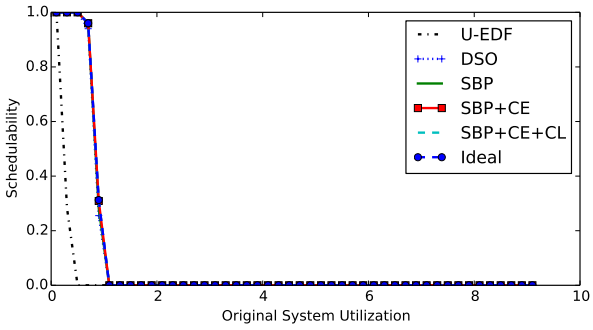
AC-Mod., Short, Light, Light, Heavy, Small



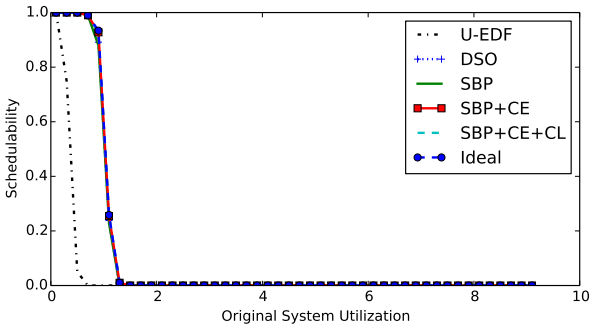
B-Heavy, Cont., Mod., Mod., Heavy, Large



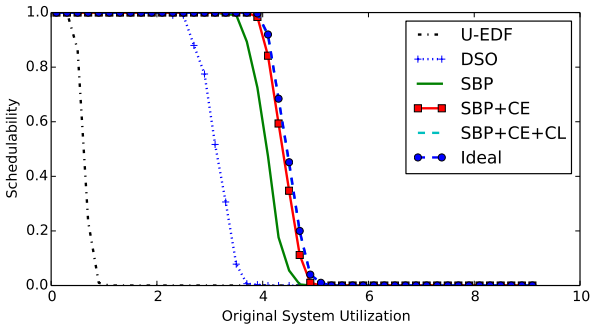
B-Heavy, Cont., Mod., Mod., Light, Small



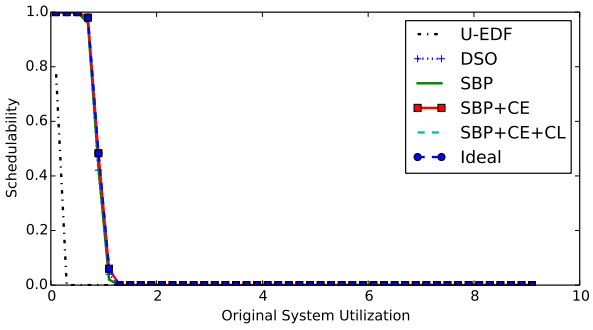
A-Heavy, Cont., Light, Light, Heavy, Small



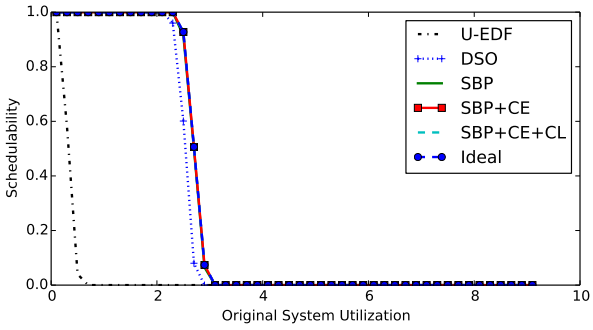
AB-Mod., Short, Light, Mod., Light, Large



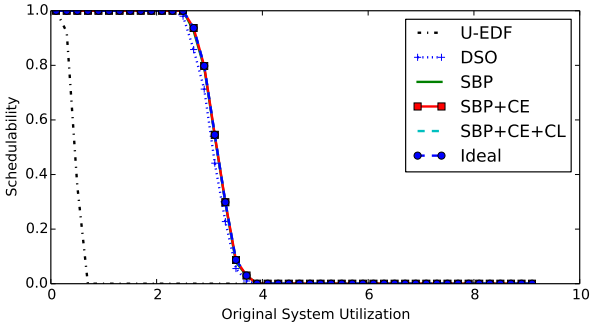
AB-Mod., Long, Mod., Mod., Heavy, Small



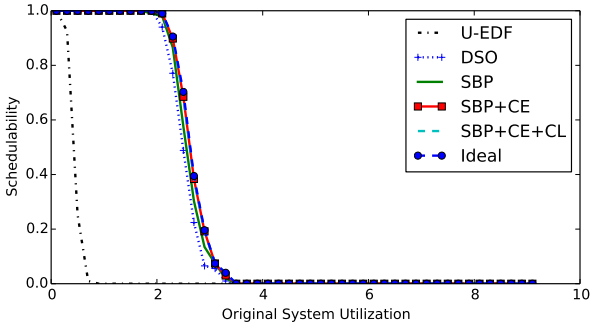
AC-Mod., Cont., Light, Heavy, Light, Small



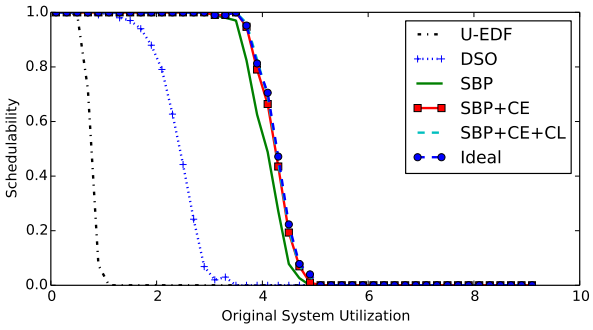
AB-Mod., Long, Light, Heavy, Light, Small



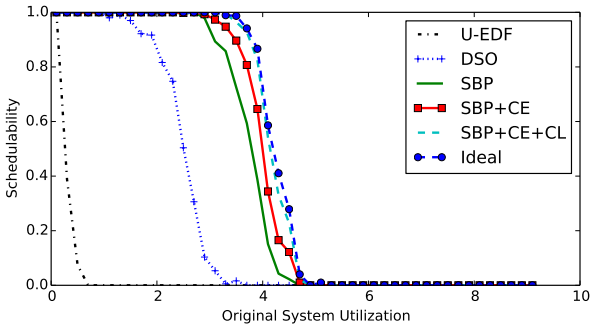
C-Heavy, Long, Light, Mod., Light, Small



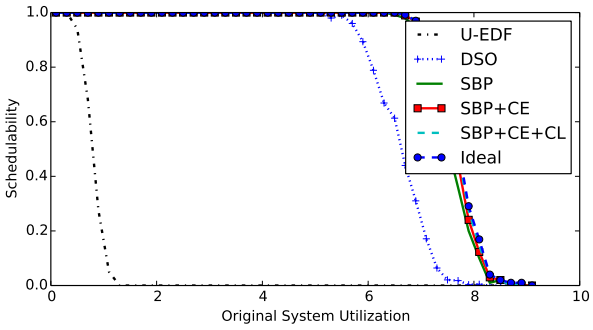
A-Heavy, Short, Mod., Heavy, Light, Large



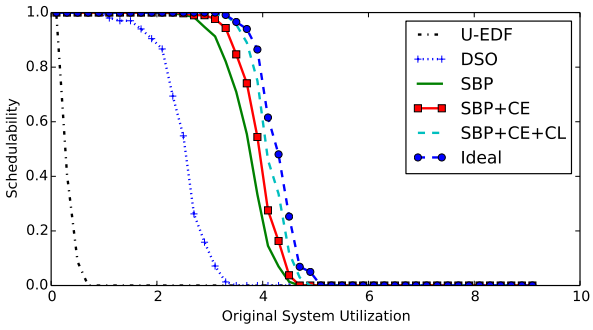
B-Heavy, Long, Heavy, Mod., Light, Small



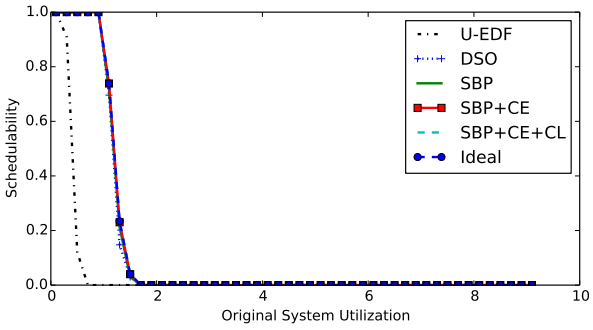
B-Heavy, Cont., Heavy, Mod., Heavy, Small



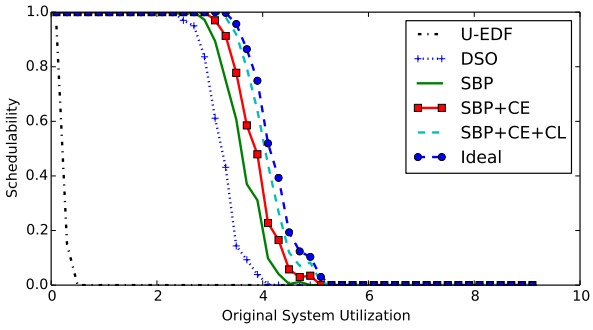
C-Heavy, Long, Mod., Light, Heavy, Small



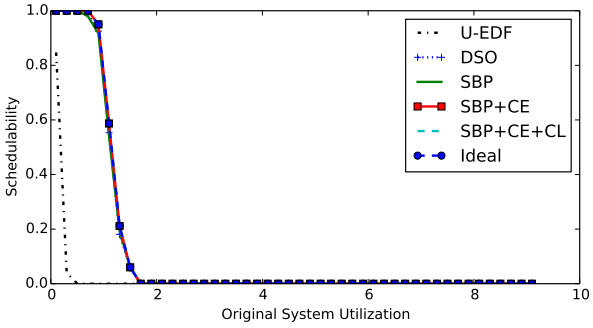
B-Heavy, Cont., Heavy, Mod., Heavy, Large



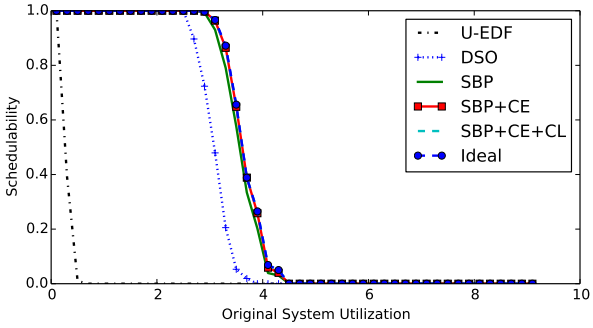
B-Heavy, Short, Light, Mod., Light, Small



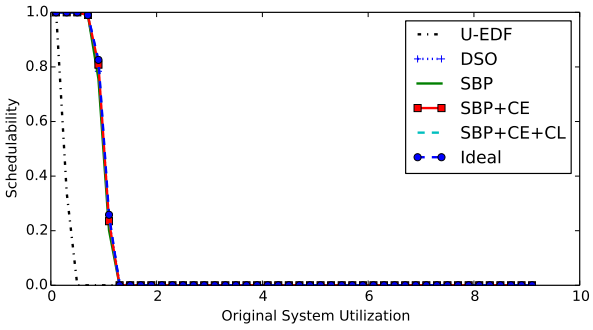
BC-Mod., Cont., Mod., Heavy, Heavy, Large



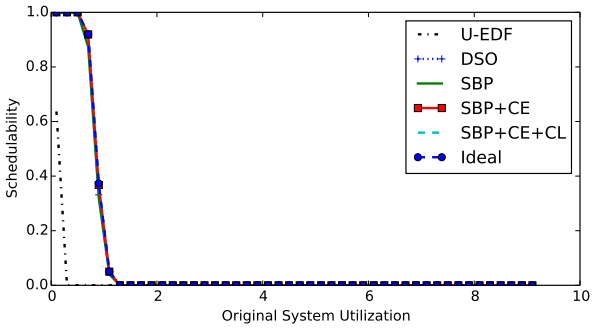
C-Heavy, Cont., Light, Heavy, Light, Small



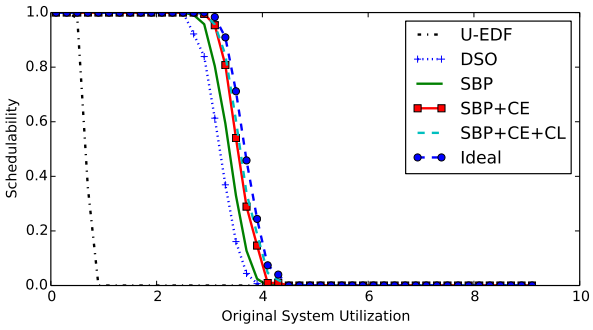
All-Mod., Cont., Mod., Mod., Light, Large



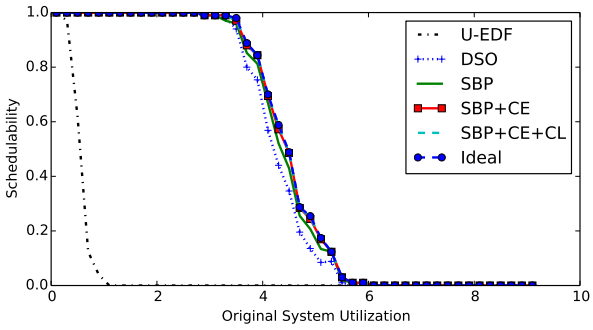
All-Mod., Short, Light, Heavy, Heavy, Large



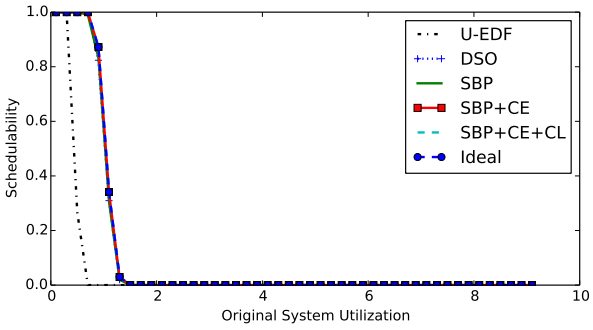
A-Heavy, Cont., Light, Heavy, Light, Large



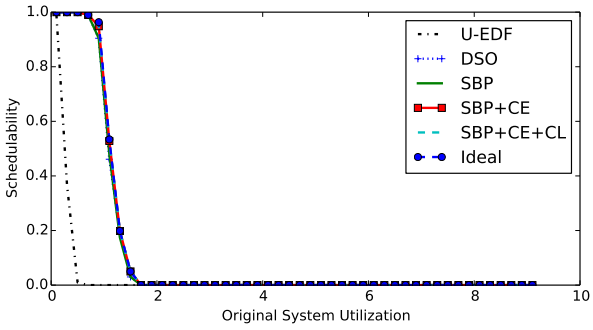
A-Heavy, Short, Heavy, Mod., Heavy, Small



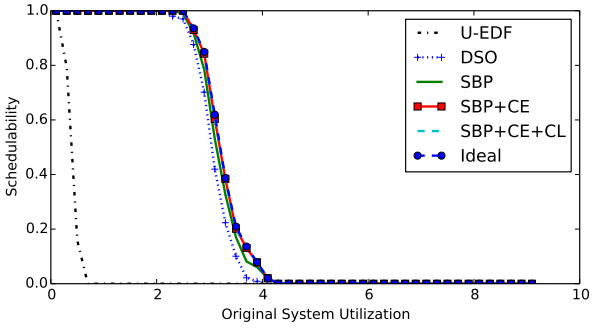
C-Heavy, Short, Mod., Mod., Light, Large



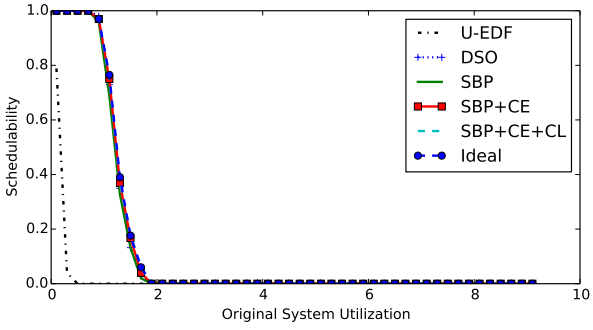
AB-Mod., Short, Light, Light, Light, Large



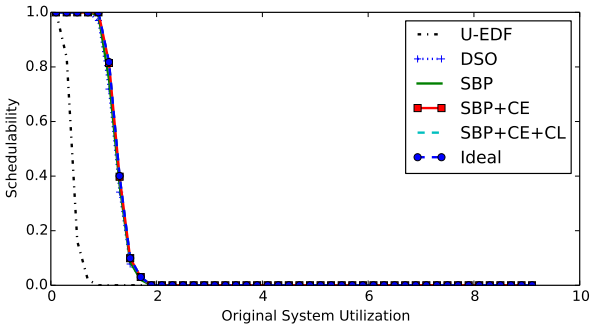
C-Heavy, Short, Light, Heavy, Light, Large



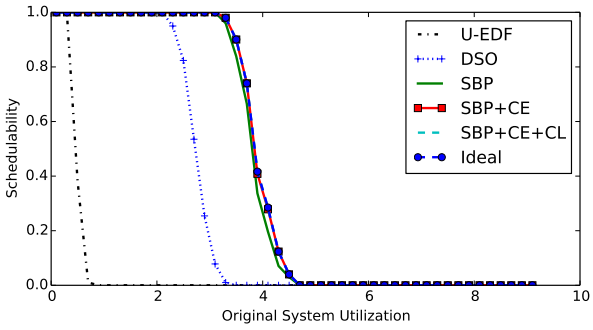
AC-Mod., Short, Mod., Heavy, Light, Large



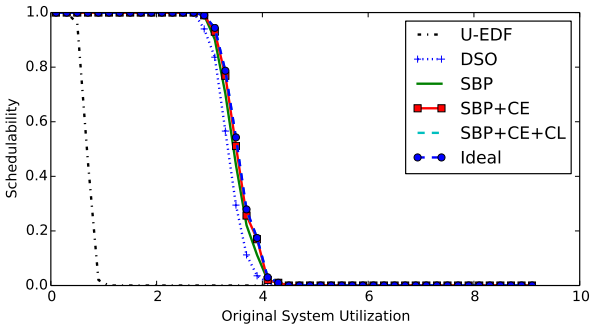
BC-Mod., Cont., Light, Heavy, Heavy, Large



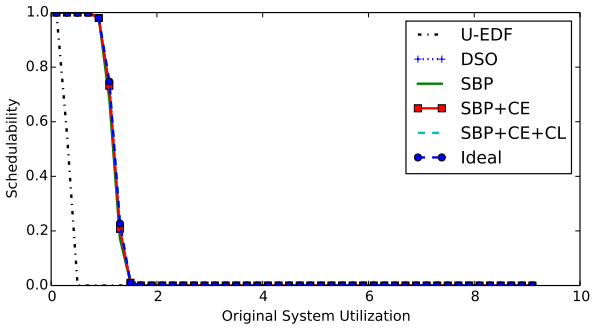
BC-Mod., Cont., Light, Light, Light, Small



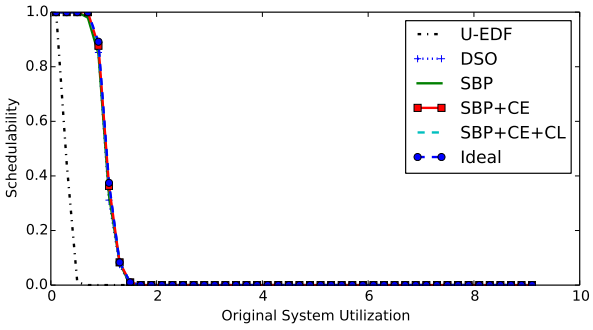
A-Heavy, Long, Mod., Heavy, Light, Small



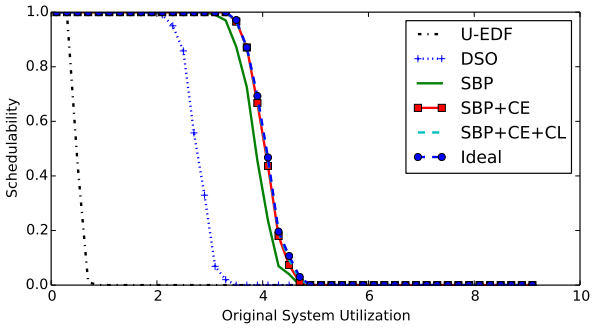
All-Mod., Short, Mod., Light, Heavy, Large



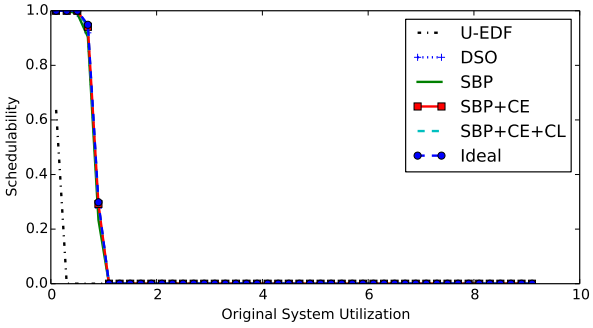
B-Heavy, Short, Light, Heavy, Heavy, Small



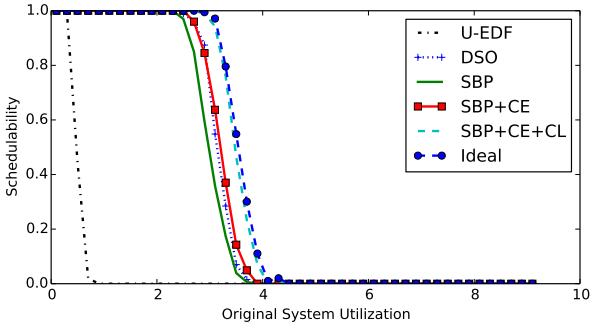
AB-Mod., Cont., Light, Light, Light, Large



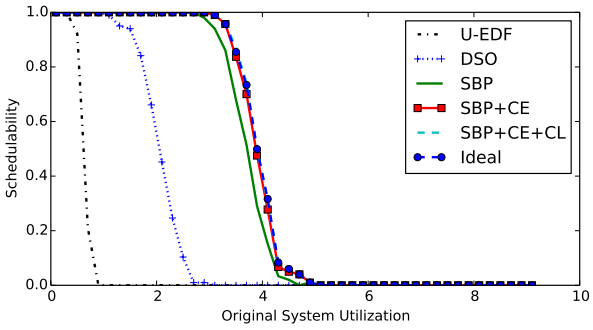
A-Heavy, Long, Mod., Heavy, Light, Large



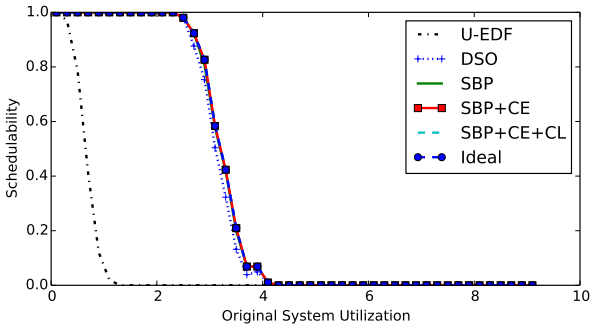
A-Heavy, Cont., Light, Heavy, Heavy, Small



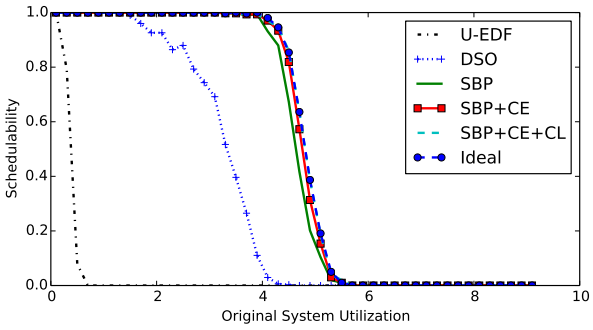
B-Heavy, Short, Mod., Heavy, Heavy, Large



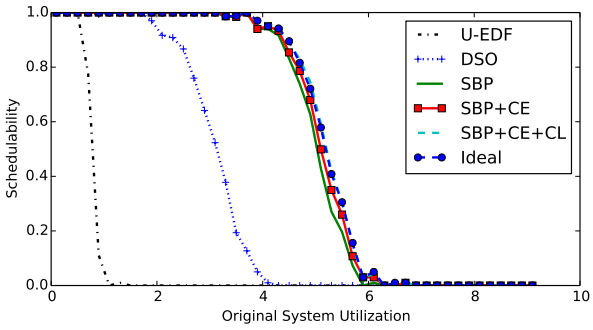
A-Heavy, Long, Heavy, Heavy, Light, Small



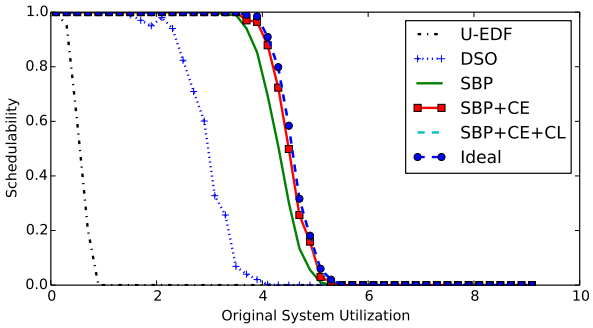
C-Heavy, Long, Light, Light, Light, Large



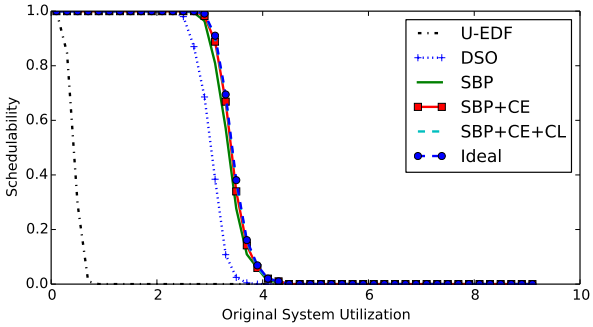
All-Mod., Cont., Heavy, Mod., Light, Large



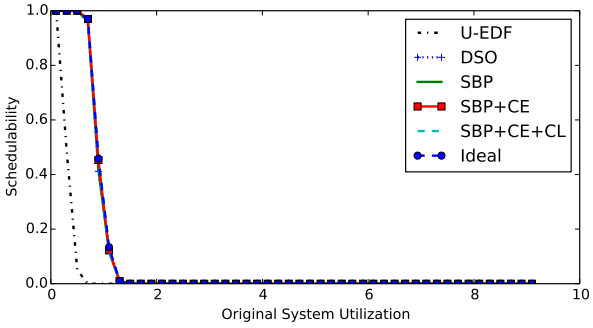
BC-Mod., Long, Heavy, Mod., Light, Large



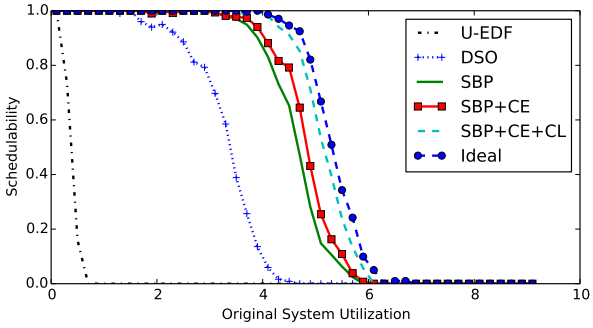
B-Heavy, Cont., Heavy, Light, Heavy, Small



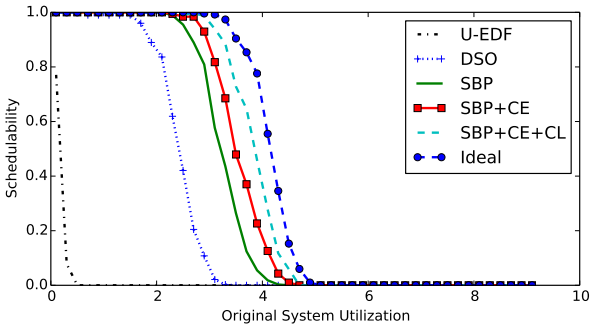
AB-Mod., Cont., Mod., Light, Heavy, Large



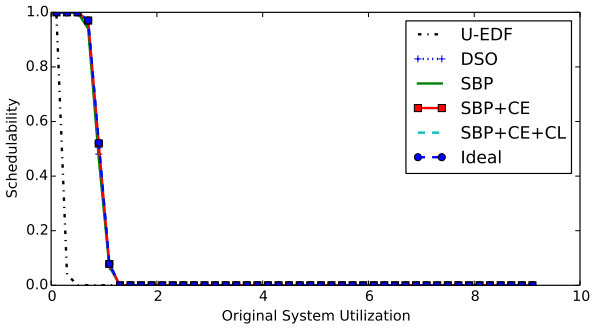
AC-Mod., Cont., Light, Light, Heavy, Large



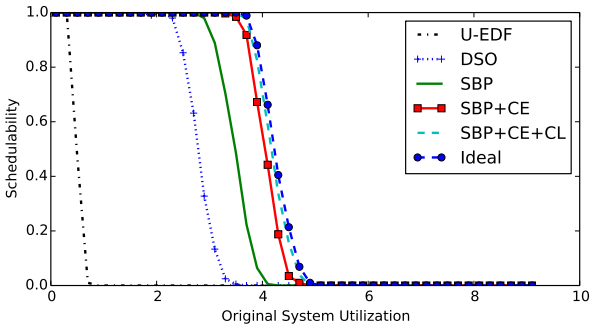
BC-Mod., Cont., Heavy, Mod., Heavy, Large



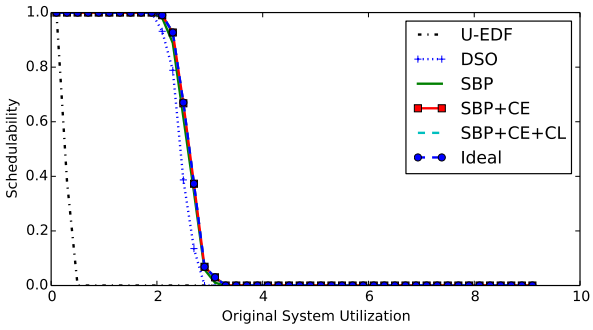
B-Heavy, Cont., Heavy, Heavy, Heavy, Small



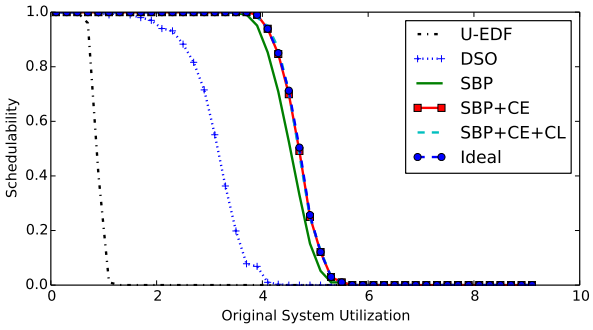
AC-Mod., Cont., Light, Mod., Heavy, Large



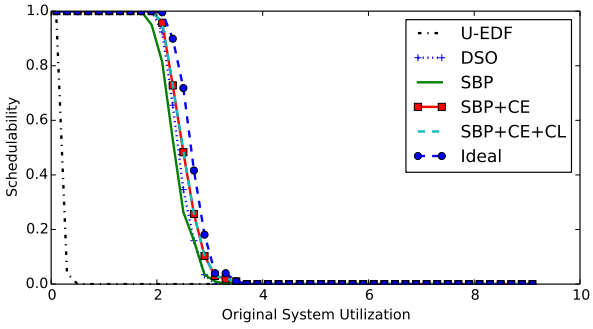
AB-Mod., Long, Mod., Heavy, Heavy, Small



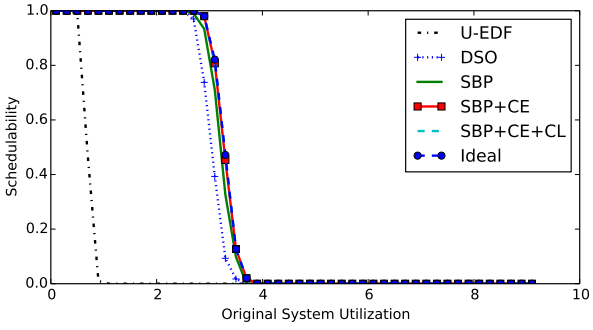
A-Heavy, Cont., Mod., Mod., Light, Small



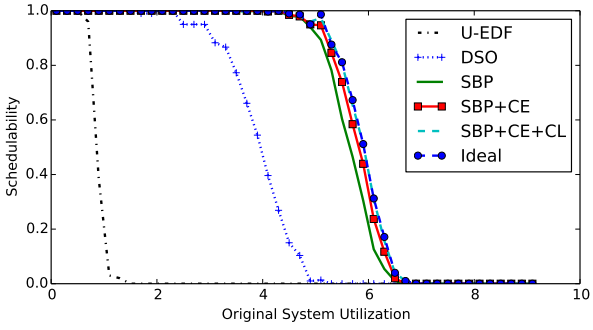
AB-Mod., Long, Heavy, Light, Light, Small



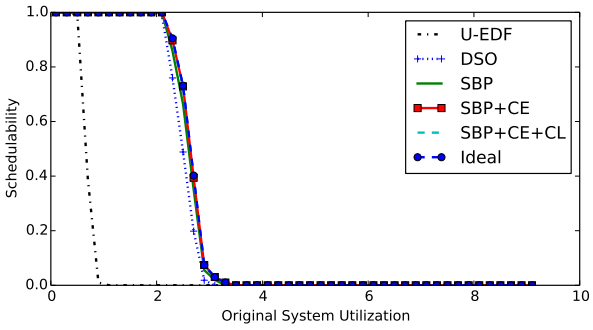
A-Heavy, Cont., Mod., Heavy, Heavy, Large



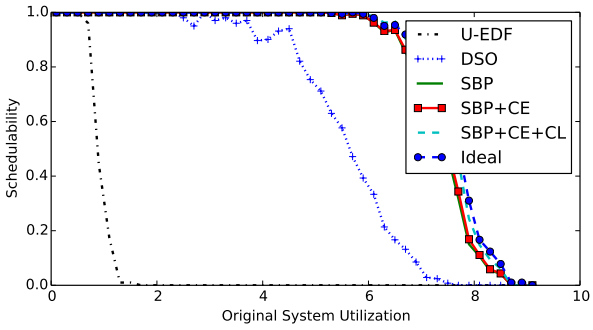
AB-Mod., Short, Mod., Light, Heavy, Small



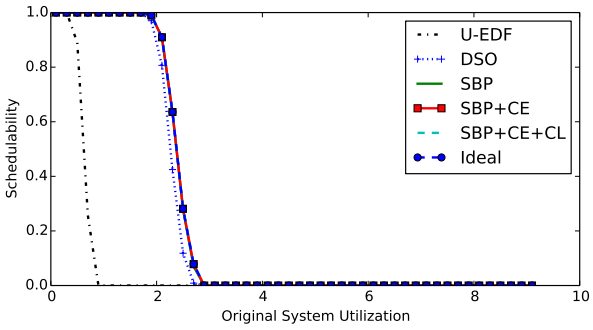
AC-Mod., Long, Heavy, Light, Heavy, Small



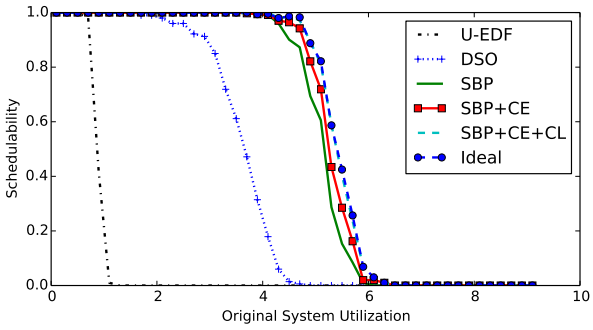
A-Heavy, Short, Mod., Light, Heavy, Small



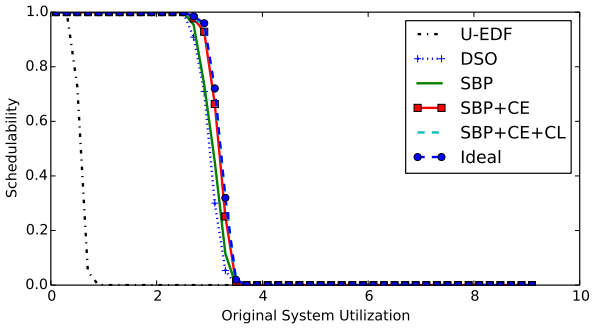
C-Heavy, Long, Heavy, Light, Heavy, Large



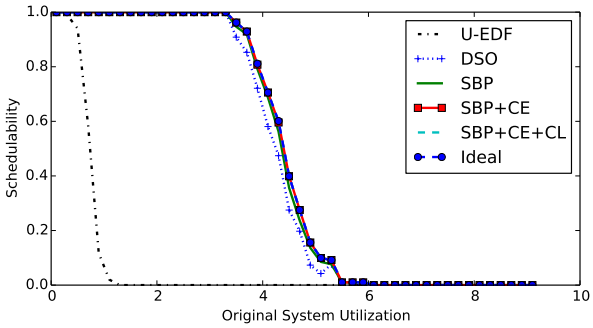
A-Heavy, Long, Light, Light, Light, Small



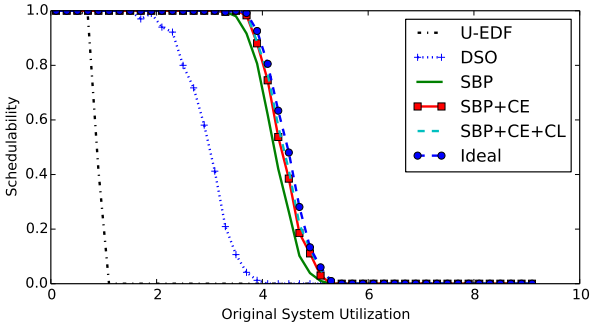
All-Mod., Long, Heavy, Light, Heavy, Large



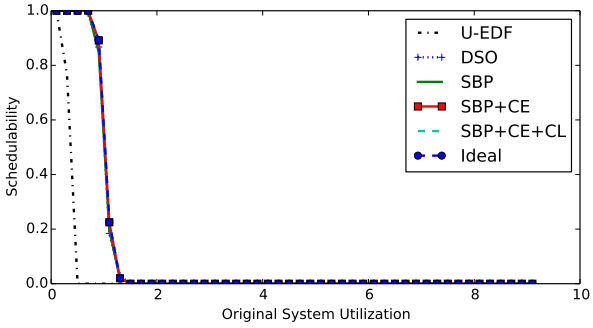
AB-Mod., Short, Mod., Mod., Heavy, Small



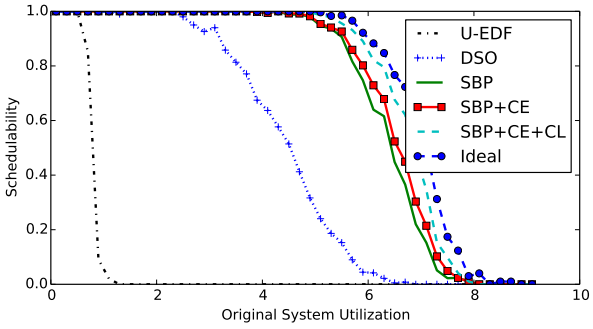
C-Heavy, Short, Mod., Light, Light, Small



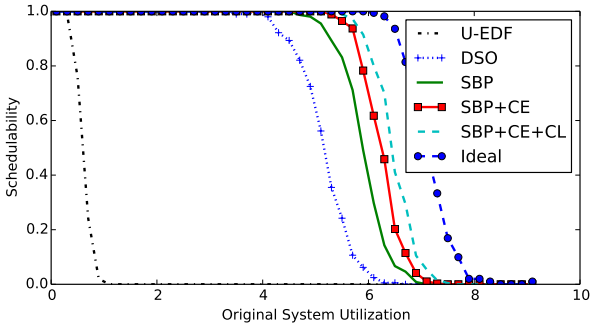
A-Heavy, Long, Heavy, Light, Heavy, Large



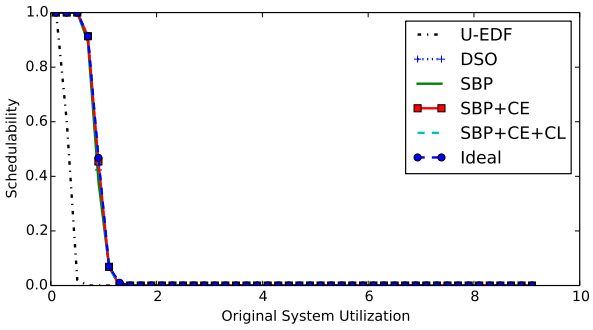
AB-Mod., Short, Light, Mod., Light, Small



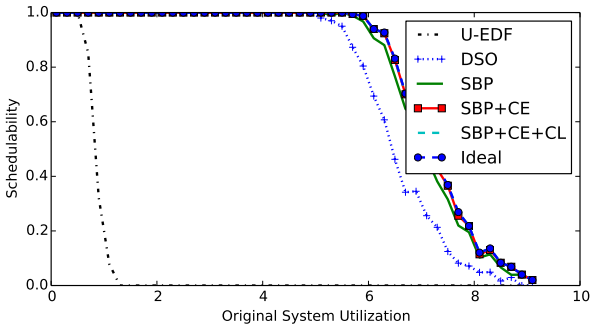
C-Heavy, Long, Heavy, Mod., Heavy, Small



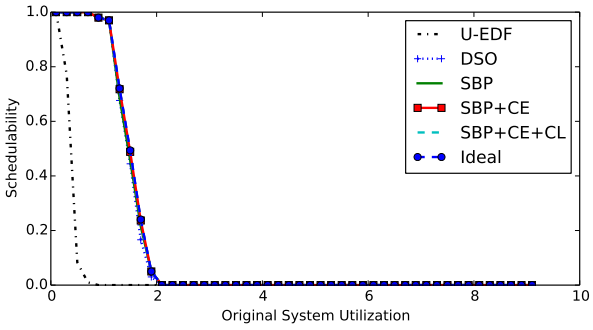
C-Heavy, Long, Mod., Heavy, Heavy, Large



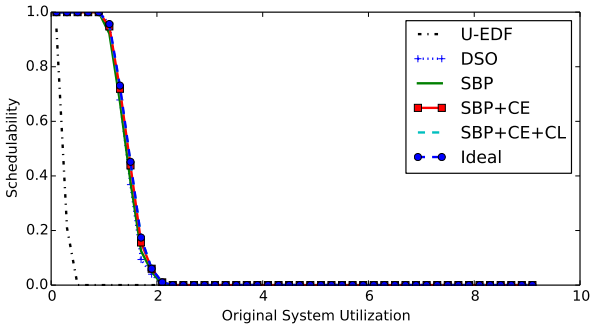
AC-Mod., Short, Light, Mod., Heavy, Large



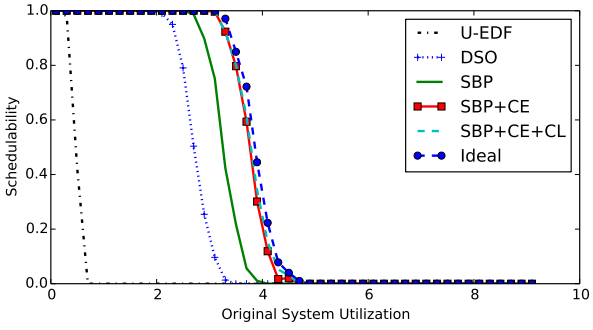
C-Heavy, Short, Heavy, Light, Light, Small



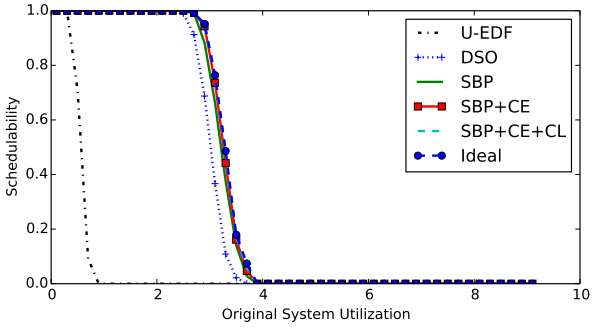
B-Heavy, Cont., Light, Light, Light, Large



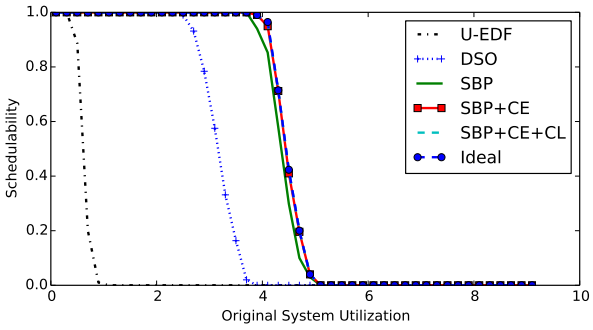
B-Heavy, Cont., Light, Mod., Heavy, Large



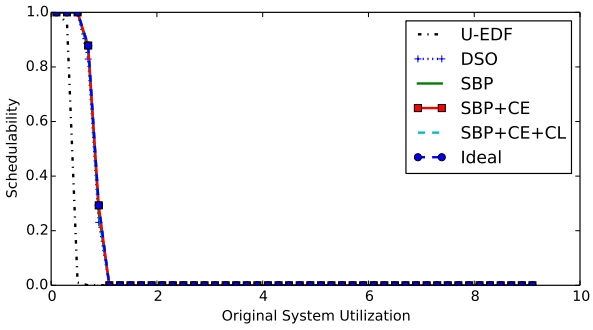
A-Heavy, Long, Mod., Heavy, Heavy, Small



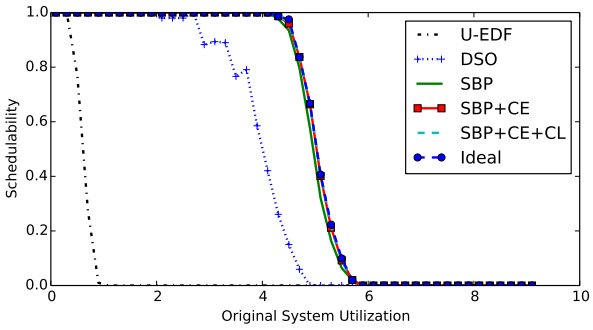
AB-Mod., Short, Mod., Mod., Light, Large



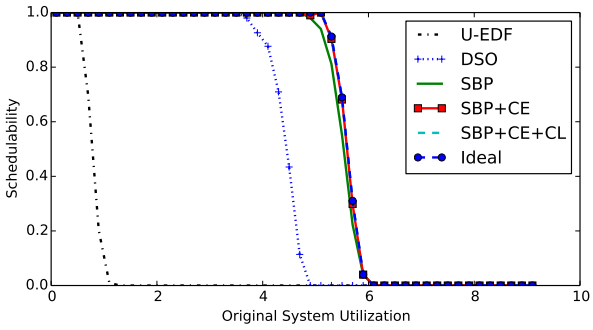
AB-Mod., Long, Mod., Mod., Light, Small



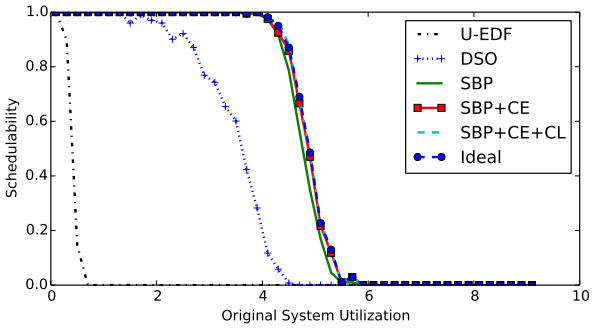
A-Heavy, Short, Light, Light, Light, Small



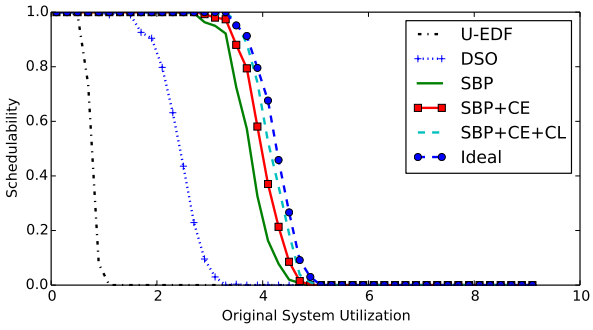
AC-Mod., Cont., Heavy, Light, Light, Small



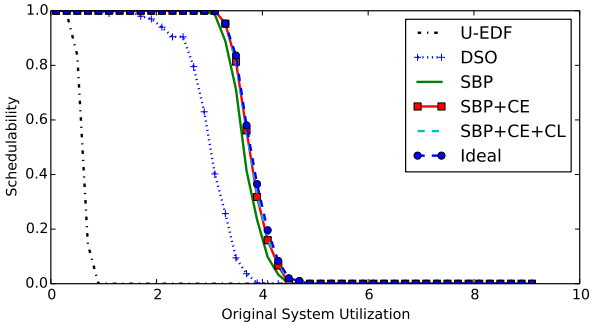
All-Mod., Long, Mod., Light, Light, Large



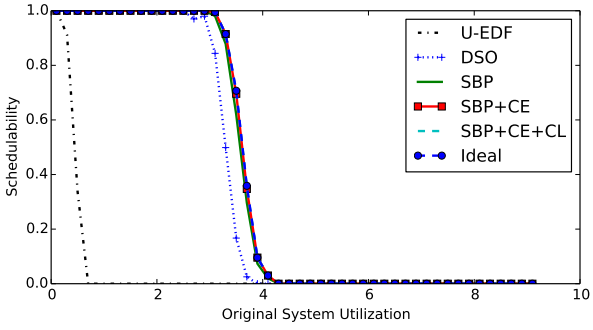
AC-Mod., Cont., Heavy, Mod., Light, Small



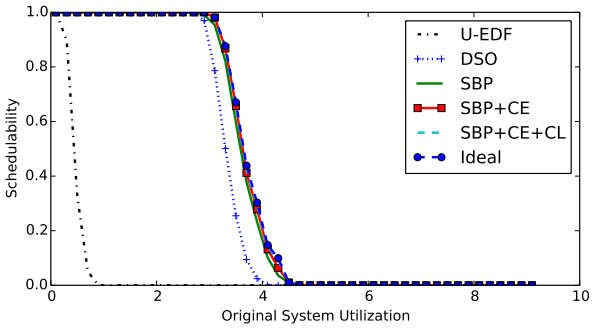
B-Heavy, Long, Heavy, Mod., Heavy, Small



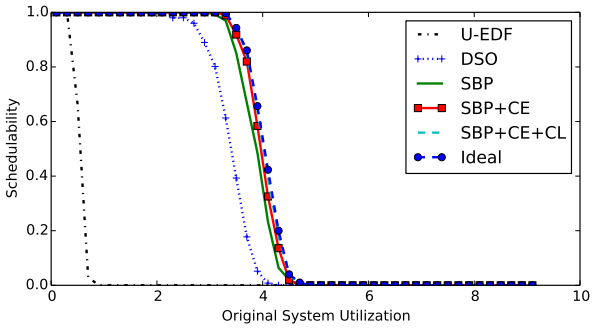
A-Heavy, Cont., Heavy, Light, Heavy, Small



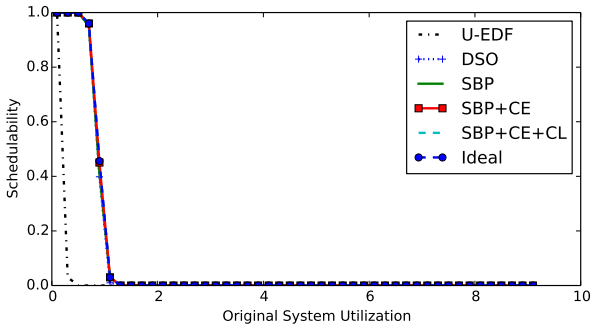
All-Mod., Cont., Mod., Light, Heavy, Small



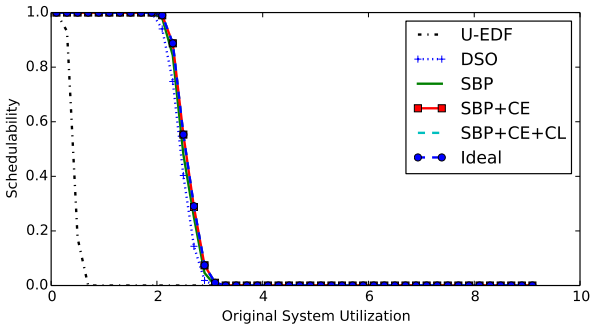
All-Mod., Cont., Mod., Light, Heavy, Large



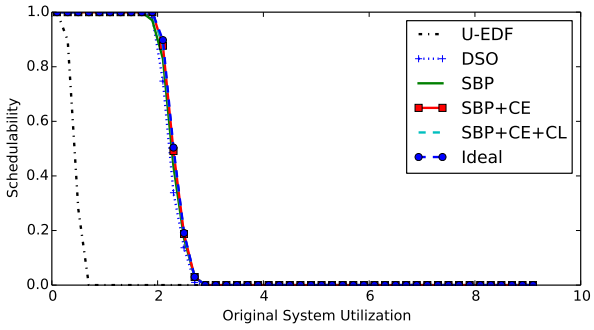
AB-Mod., Short, Heavy, Heavy, Light, Large



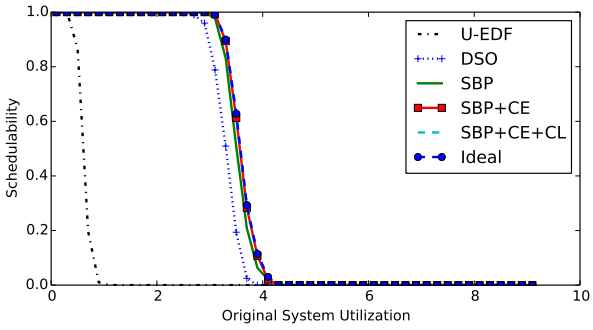
AC-Mod., Cont., Light, Mod., Heavy, Small



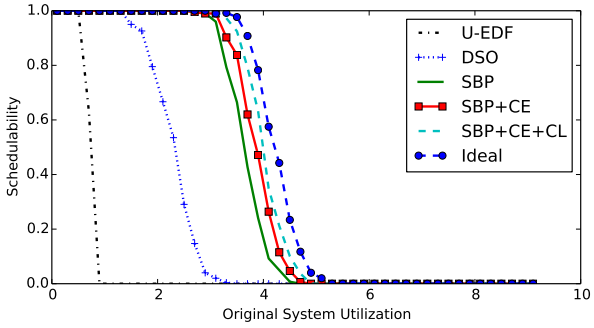
A-Heavy, Short, Mod., Heavy, Light, Small



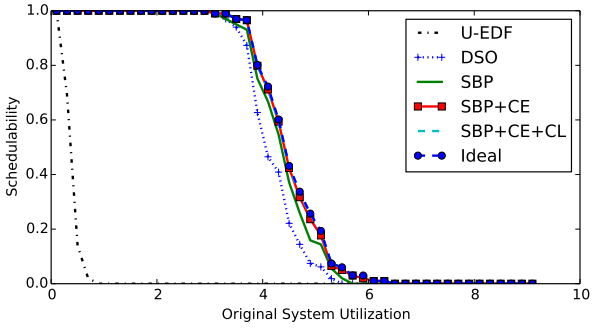
A-Heavy, Long, Light, Mod., Heavy, Large



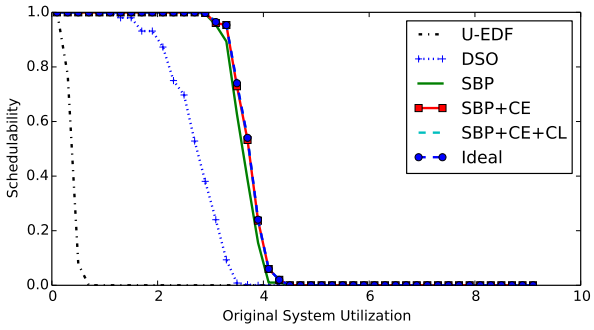
B-Heavy, Short, Mod., Mod., Light, Small



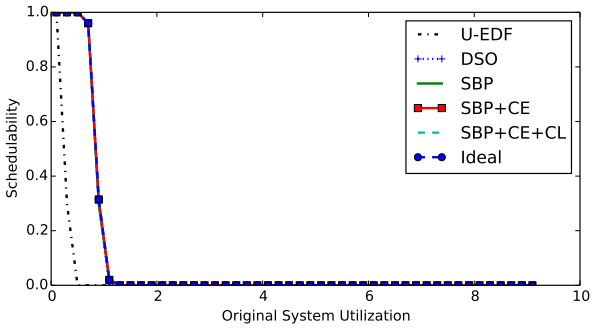
A-Heavy, Long, Heavy, Mod., Heavy, Large



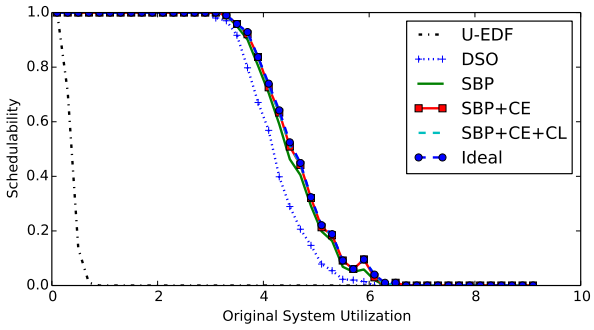
C-Heavy, Cont., Mod., Mod., Heavy, Small



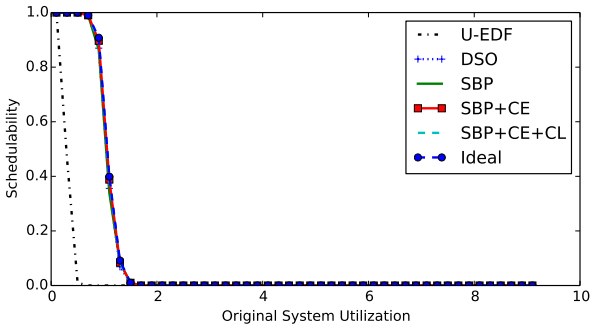
A-Heavy, Cont., Heavy, Mod., Light, Small



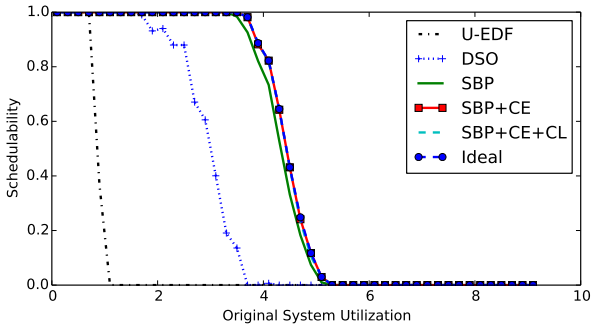
A-Heavy, Cont., Light, Light, Light, Small



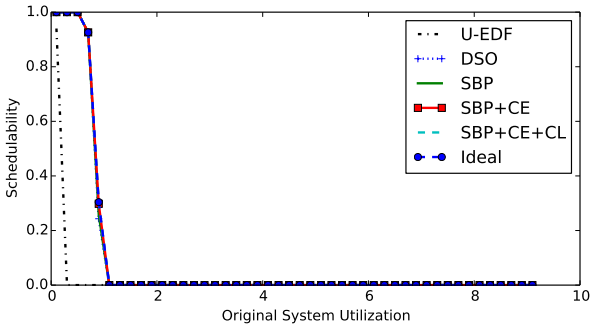
C-Heavy, Cont., Mod., Mod., Light, Large



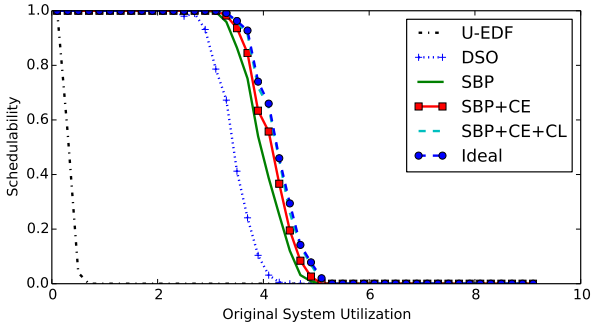
AB-Mod., Cont., Light, Light, Heavy, Large



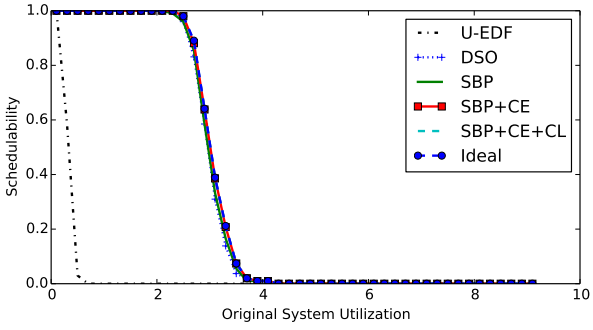
A-Heavy, Long, Heavy, Light, Light, Small



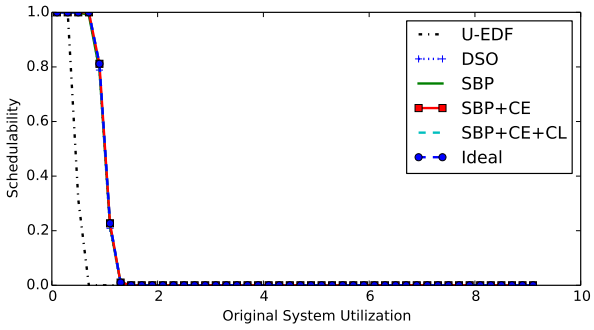
A-Heavy, Cont., Light, Mod., Heavy, Small



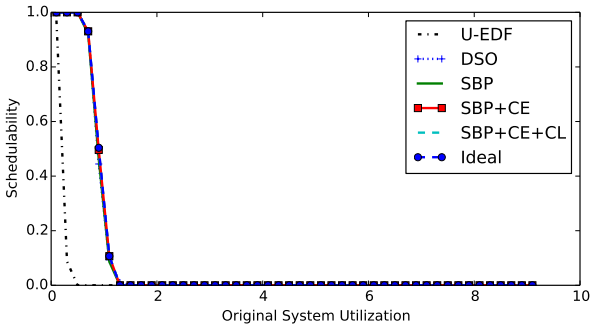
BC-Mod., Cont., Mod., Mod., Heavy, Large



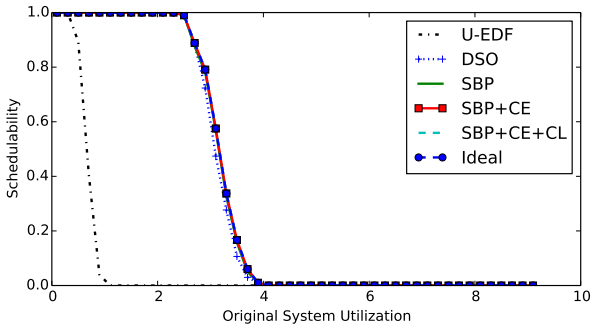
C-Heavy, Long, Light, Heavy, Heavy, Small



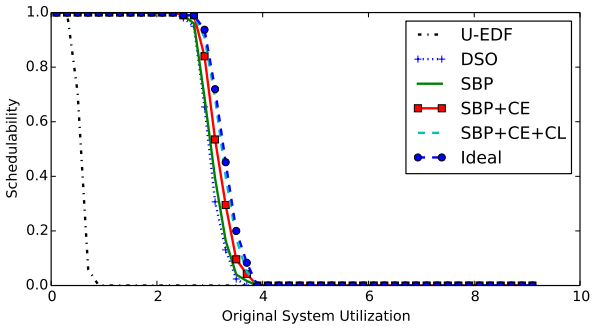
All-Mod., Short, Light, Light, Light, Small



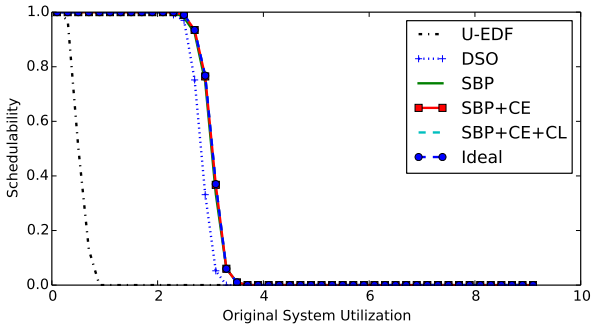
AC-Mod., Cont., Light, Mod., Light, Large



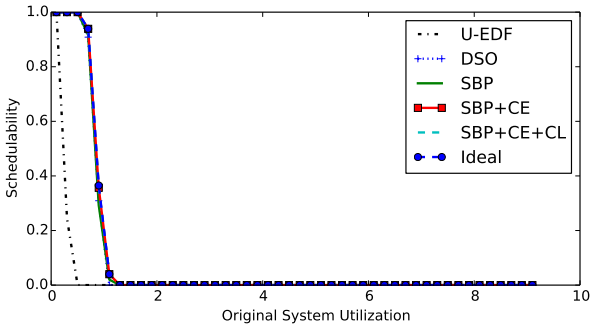
C-Heavy, Long, Light, Light, Light, Small



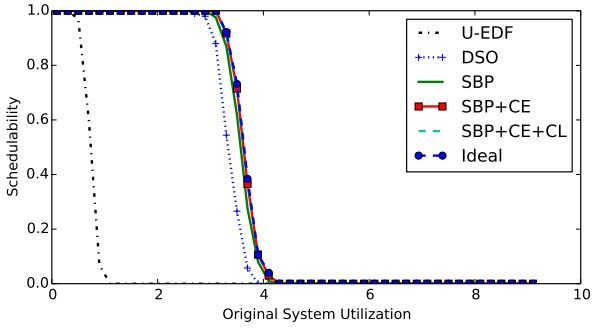
AB-Mod., Short, Mod., Mod., Heavy, Large



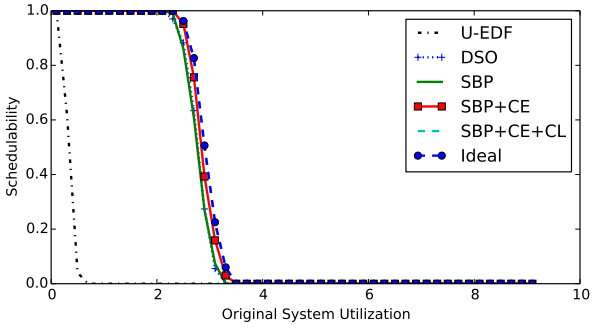
B-Heavy, Long, Light, Mod., Light, Large



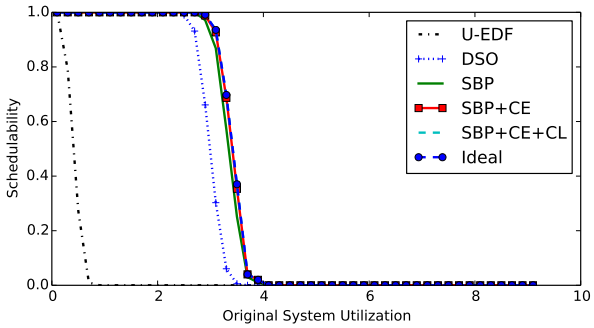
A-Heavy, Short, Light, Heavy, Light, Large



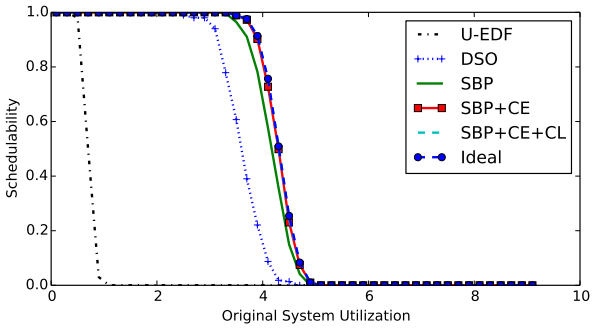
B-Heavy, Short, Mod., Light, Heavy, Small



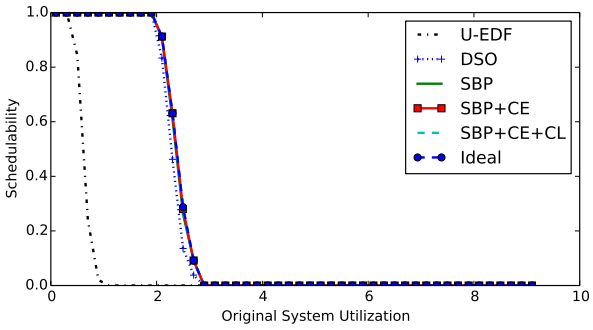
BC-Mod., Long, Light, Heavy, Heavy, Large



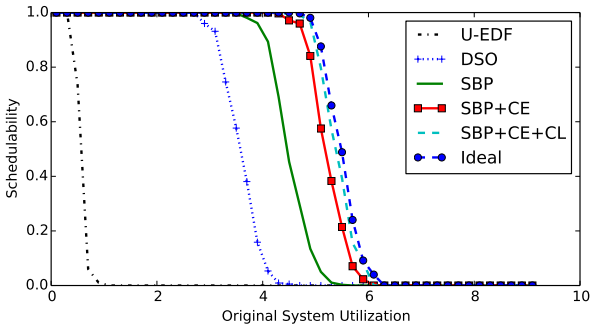
AB-Mod., Cont., Mod., Light, Heavy, Small



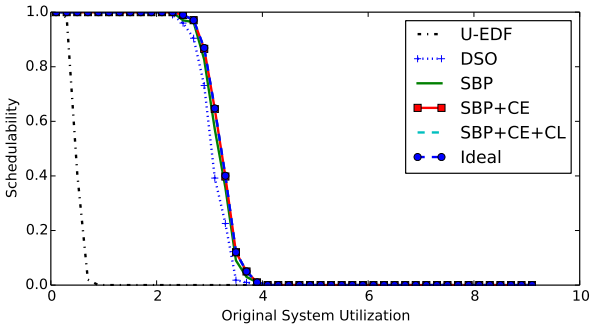
B-Heavy, Short, Heavy, Mod., Light, Small



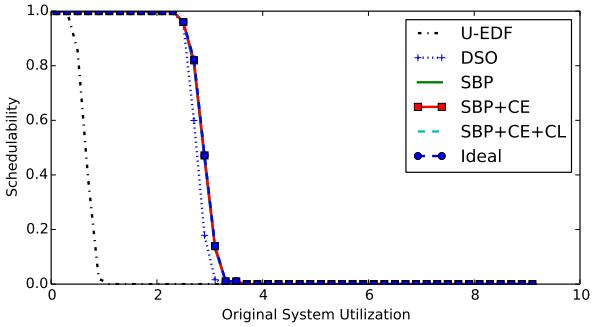
A-Heavy, Long, Light, Light, Heavy, Large



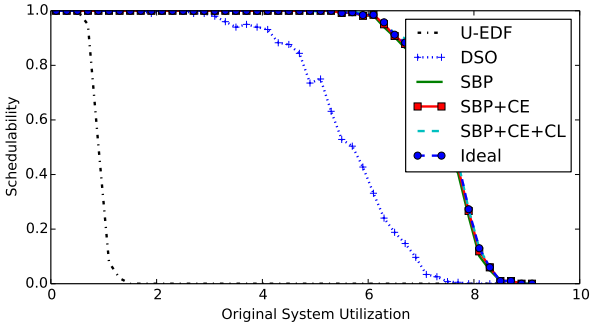
BC-Mod., Long, Mod., Heavy, Heavy, Small



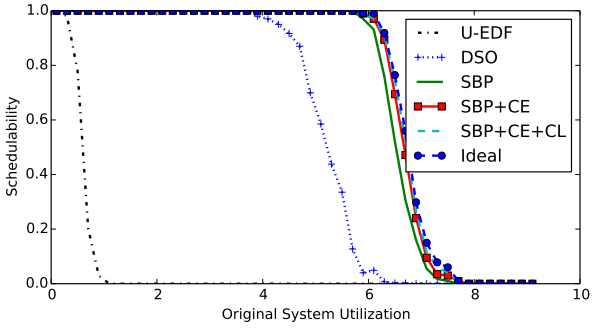
AC-Mod., Cont., Mod., Light, Heavy, Small



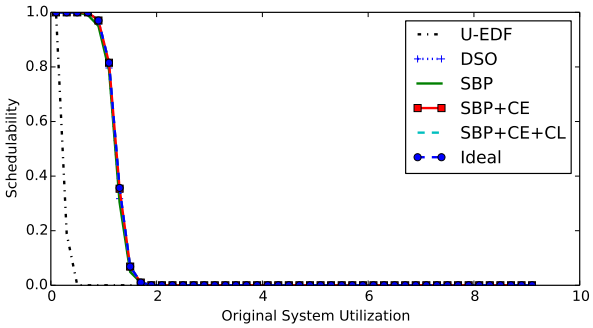
AB-Mod., Long, Light, Light, Heavy, Large



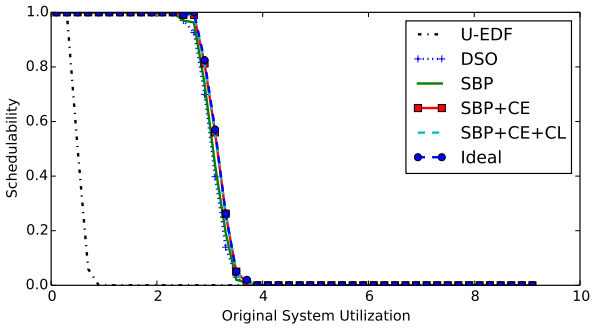
C-Heavy, Long, Heavy, Light, Light, Small



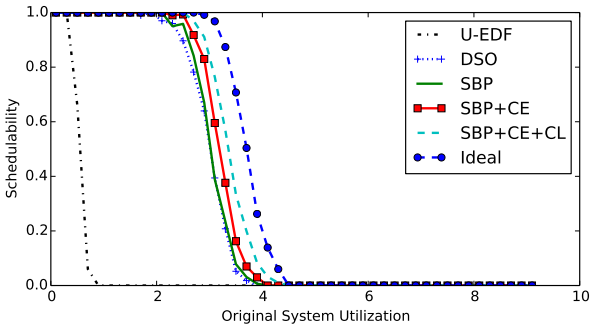
C-Heavy, Long, Mod., Heavy, Light, Small



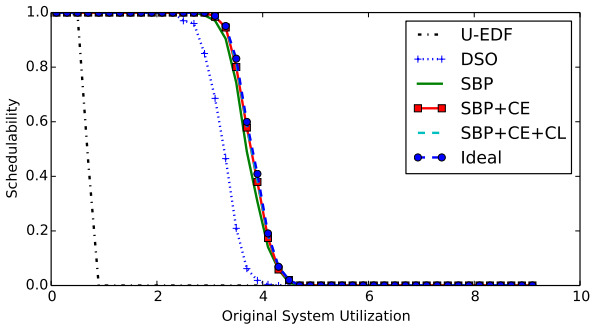
BC-Mod., Cont., Light, Mod., Heavy, Small



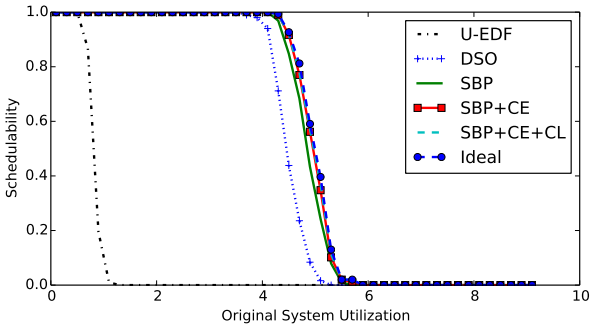
AC-Mod., Short, Mod., Mod., Heavy, Small



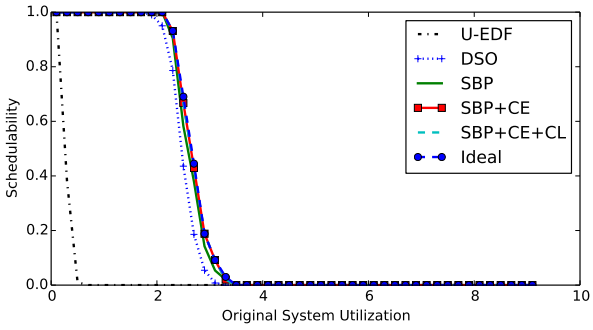
A-Heavy, Short, Heavy, Heavy, Heavy, Large



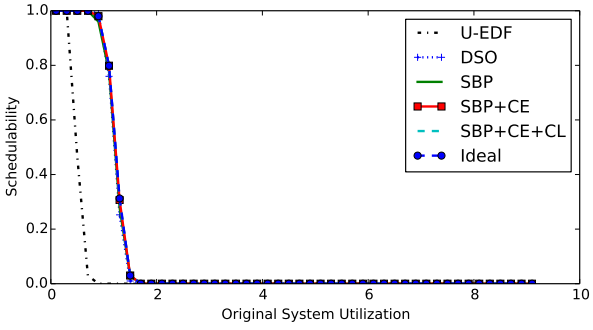
A-Heavy, Short, Heavy, Mod., Light, Large



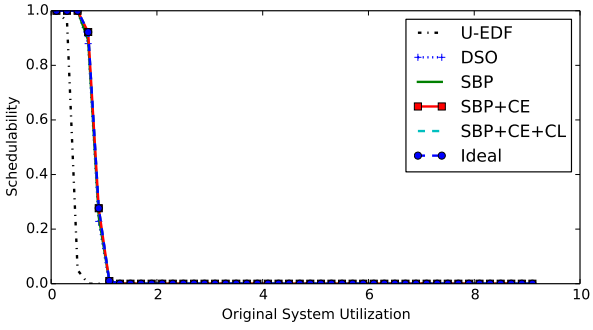
AC-Mod., Short, Heavy, Light, Heavy, Small



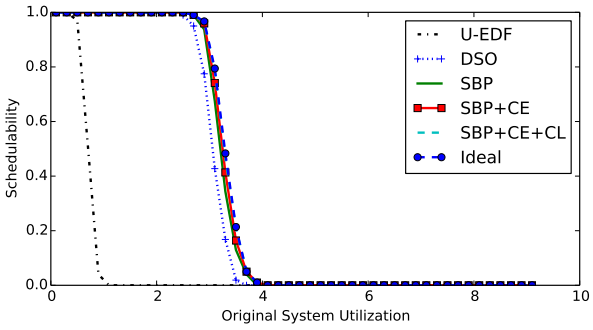
A-Heavy, Cont., Mod., Mod., Light, Large



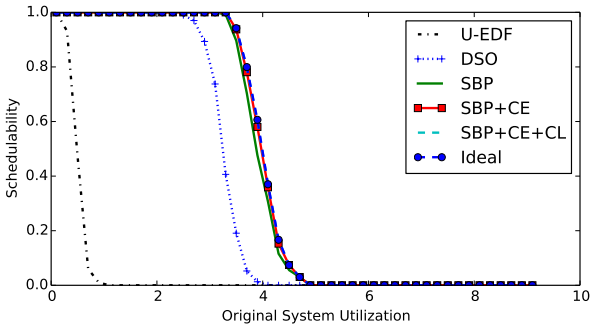
B-Heavy, Short, Light, Light, Light, Large



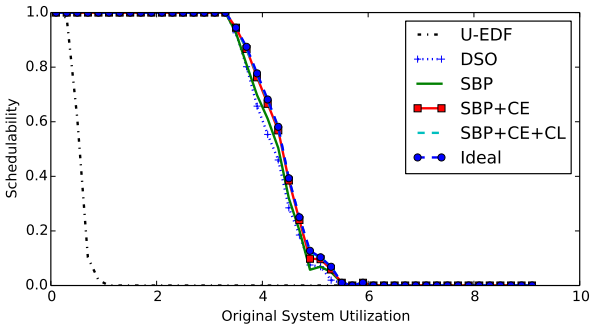
A-Heavy, Short, Light, Light, Heavy, Small



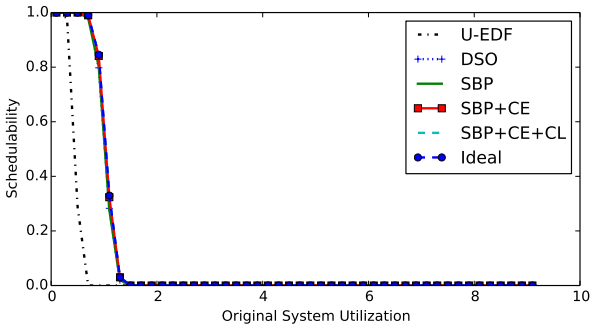
AB-Mod., Short, Mod., Light, Heavy, Large



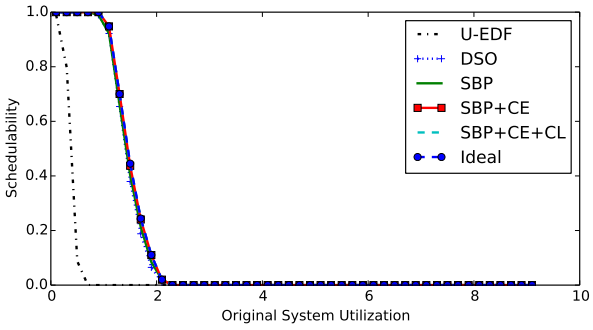
B-Heavy, Cont., Mod., Light, Light, Large



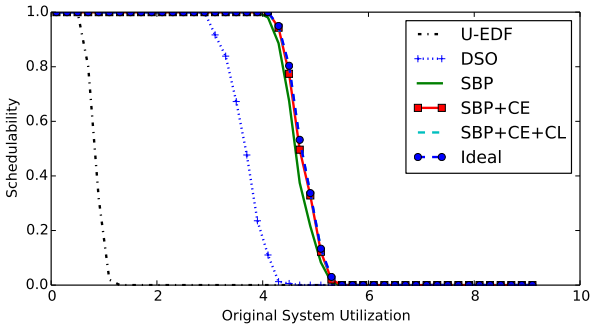
C-Heavy, Short, Mod., Mod., Heavy, Small



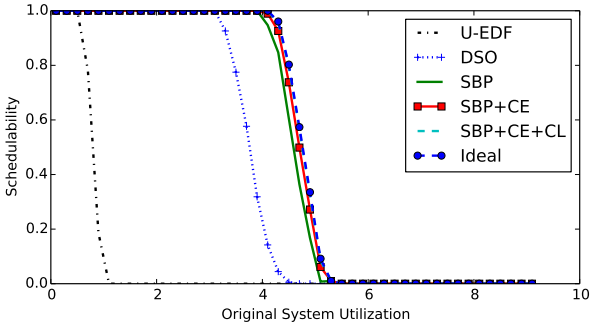
All-Mod., Short, Light, Light, Heavy, Large



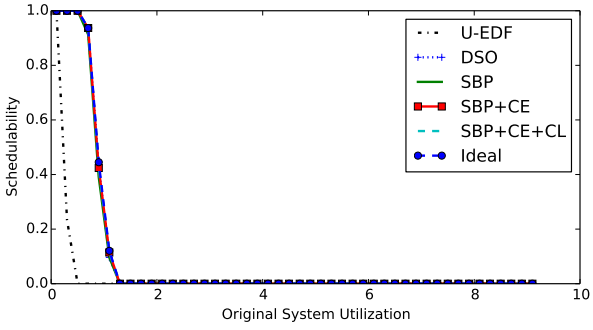
B-Heavy, Cont., Light, Light, Heavy, Large



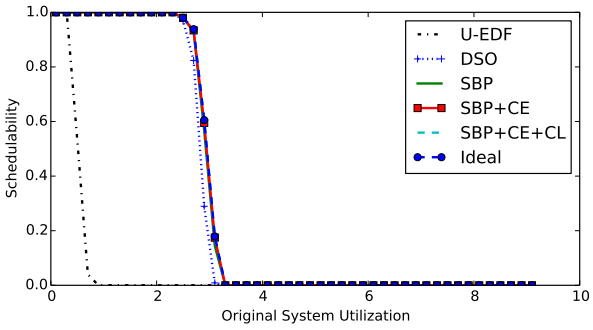
B-Heavy, Long, Mod., Light, Heavy, Small



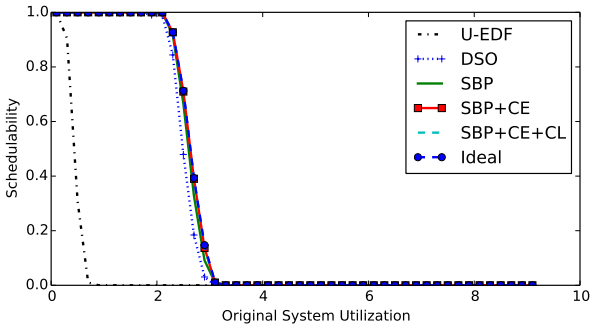
AB-Mod., Long, Mod., Light, Heavy, Large



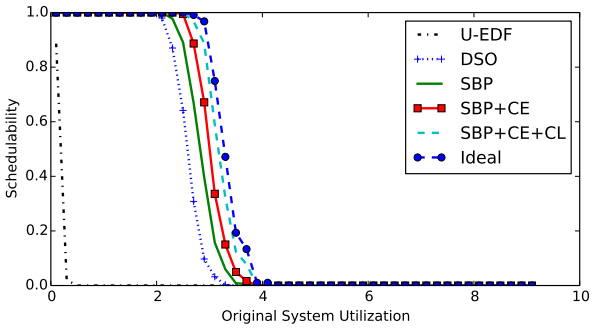
AC-Mod., Short, Light, Heavy, Heavy, Large



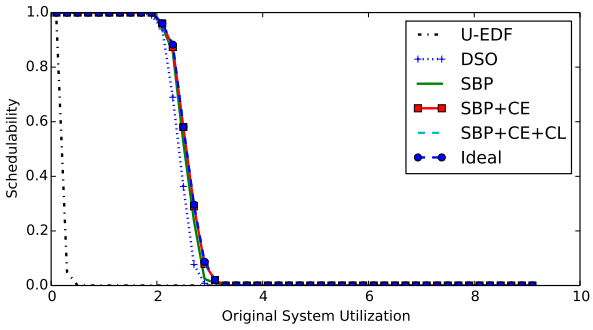
B-Heavy, Long, Light, Mod., Heavy, Small



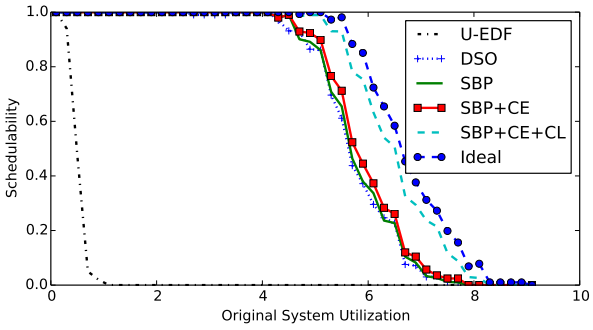
A-Heavy, Cont., Mod., Light, Heavy, Small



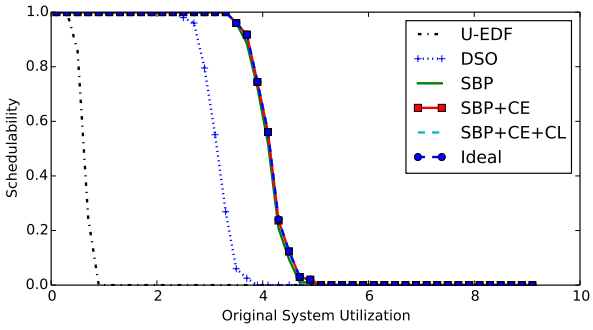
AB-Mod., Cont., Mod., Heavy, Heavy, Large



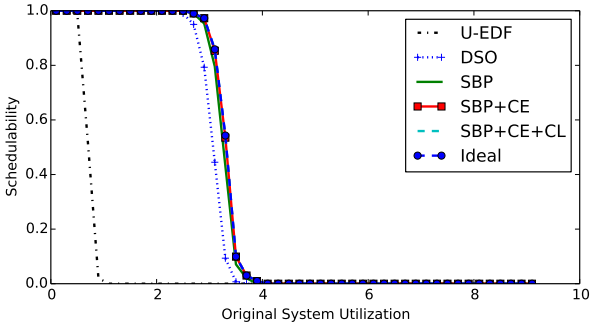
A-Heavy, Cont., Mod., Heavy, Light, Small



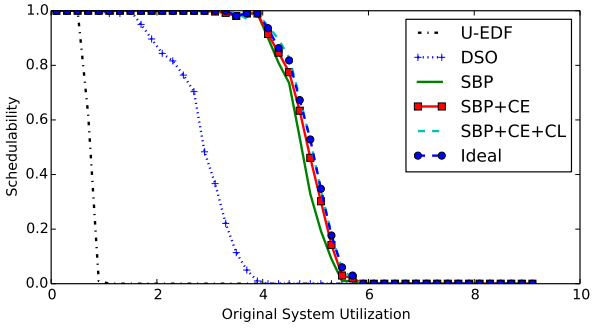
C-Heavy, Short, Heavy, Heavy, Heavy, Large



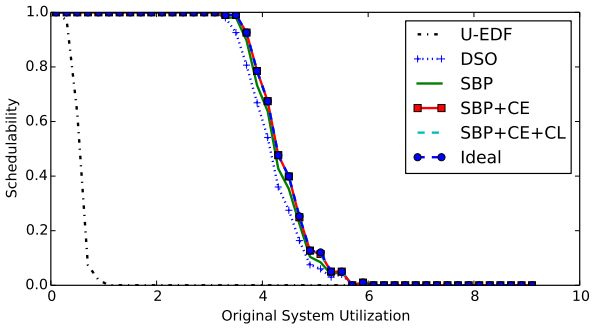
A-Heavy, Long, Mod., Mod., Light, Small



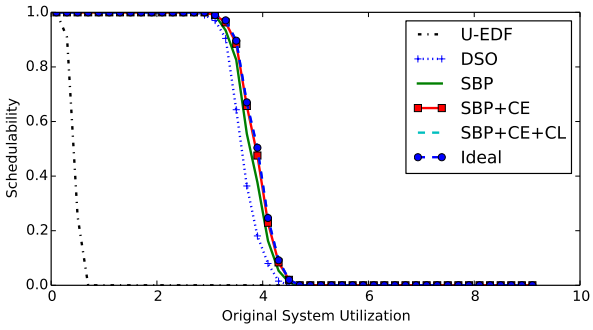
AB-Mod., Short, Mod., Light, Light, Small



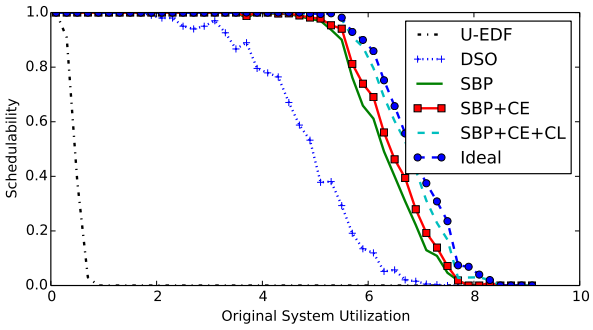
All-Mod., Long, Heavy, Mod., Light, Large



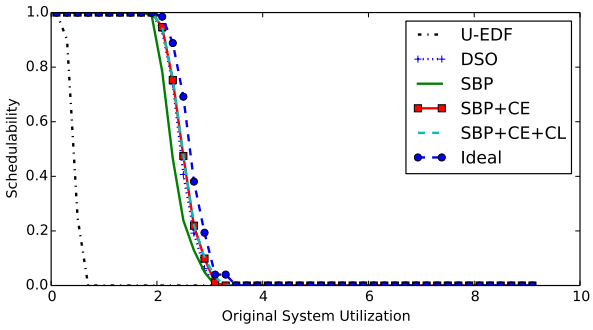
C-Heavy, Short, Mod., Mod., Light, Small



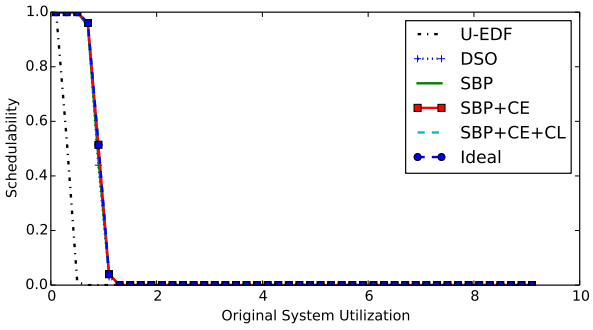
BC-Mod., Short, Mod., Heavy, Light, Small



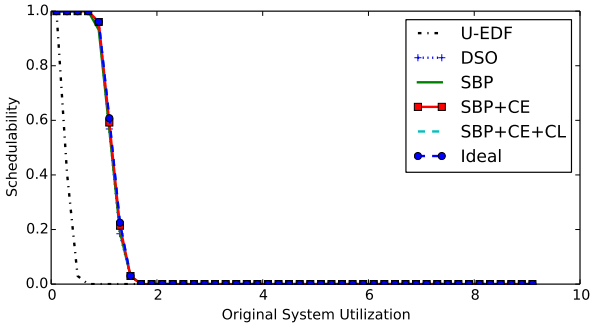
C-Heavy, Cont., Heavy, Mod., Heavy, Small



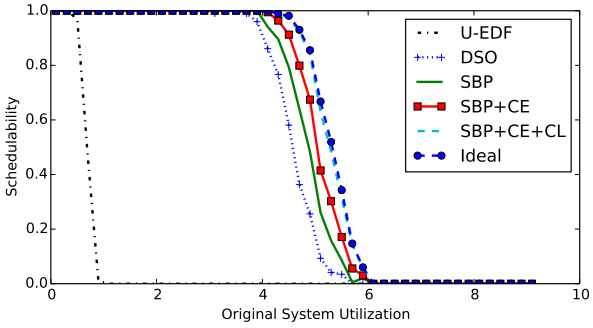
A-Heavy, Short, Mod., Heavy, Heavy, Large



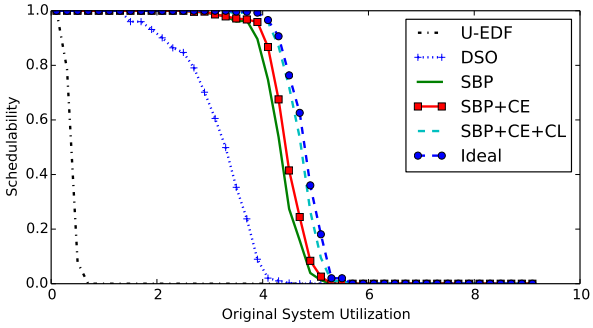
AC-Mod., Cont., Light, Light, Light, Small



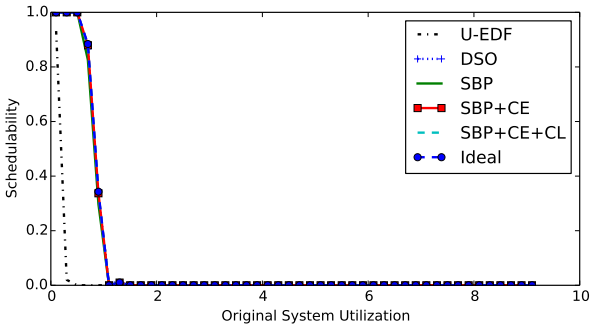
BC-Mod., Short, Light, Heavy, Light, Large



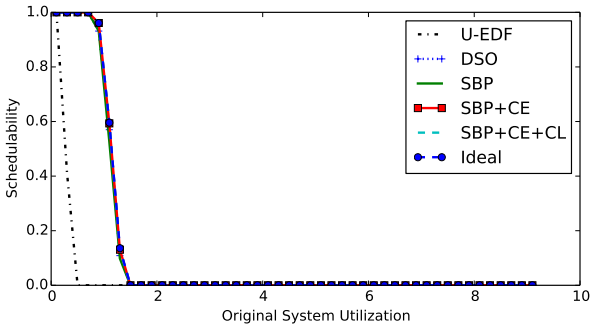
BC-Mod., Short, Heavy, Mod., Heavy, Small



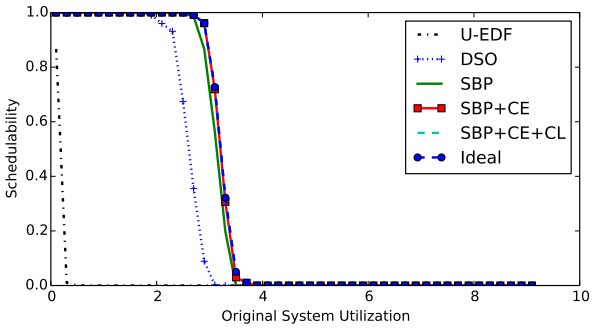
All-Mod., Cont., Heavy, Mod., Heavy, Large



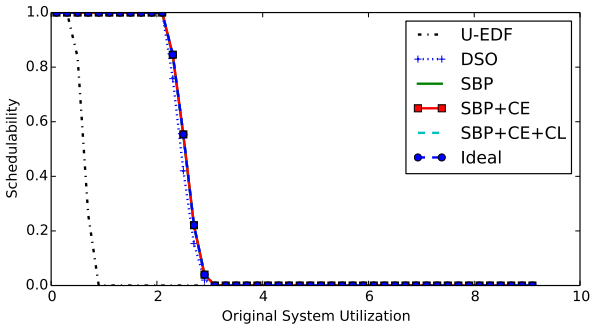
A-Heavy, Cont., Light, Mod., Heavy, Large



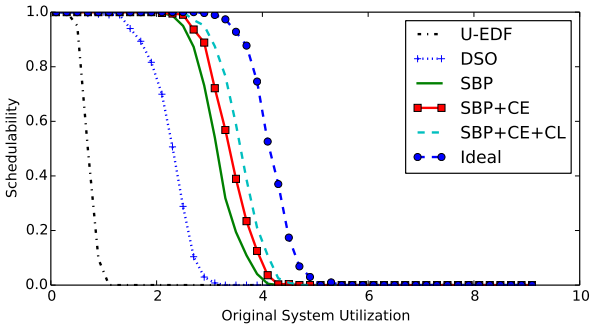
BC-Mod., Short, Light, Heavy, Heavy, Small



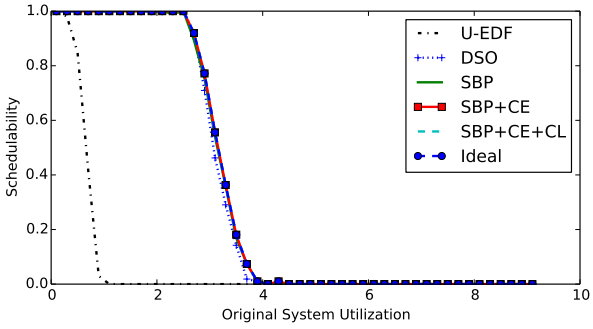
AB-Mod., Cont., Mod., Heavy, Light, Small



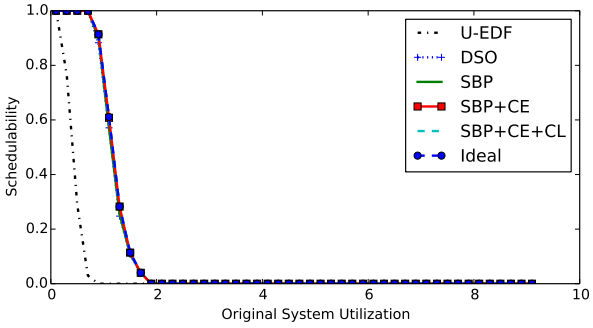
AC-Mod., Long, Light, Light, Light, Small



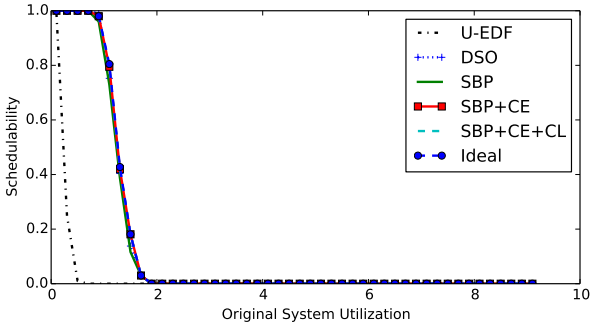
B-Heavy, Long, Heavy, Heavy, Heavy, Large



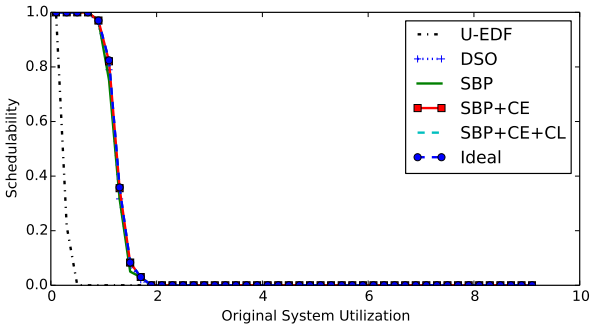
C-Heavy, Long, Light, Light, Heavy, Small



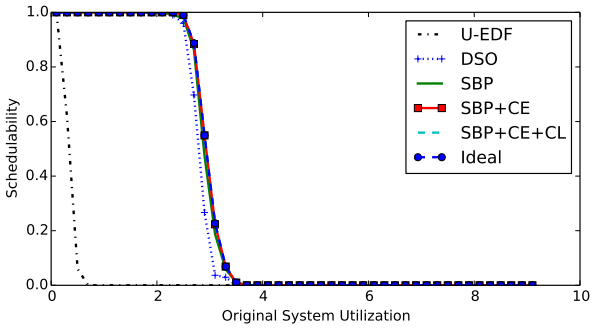
C-Heavy, Cont., Light, Light, Heavy, Large



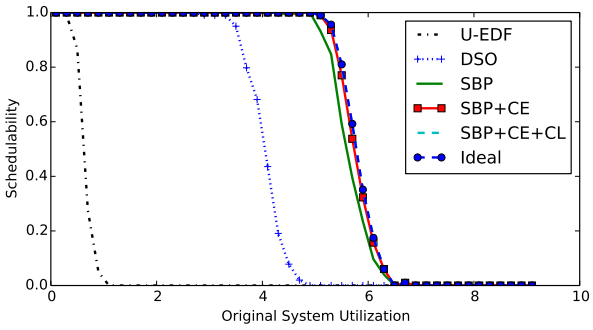
BC-Mod., Cont., Light, Mod., Heavy, Large



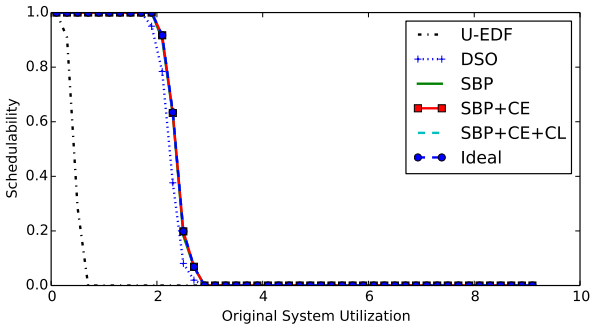
BC-Mod., Cont., Light, Mod., Light, Small



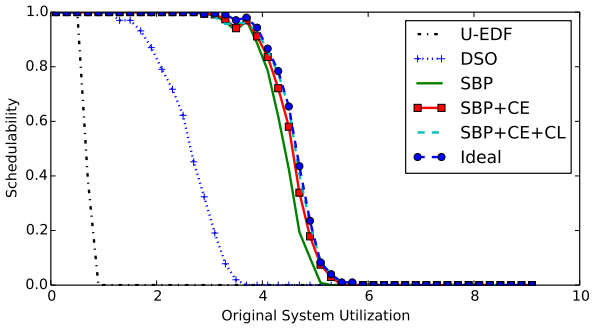
BC-Mod., Long, Light, Heavy, Light, Large



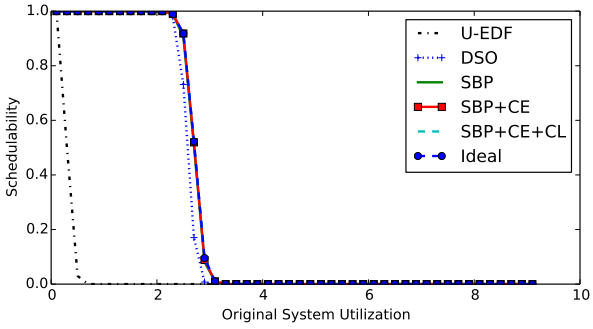
BC-Mod., Long, Mod., Mod., Light, Large



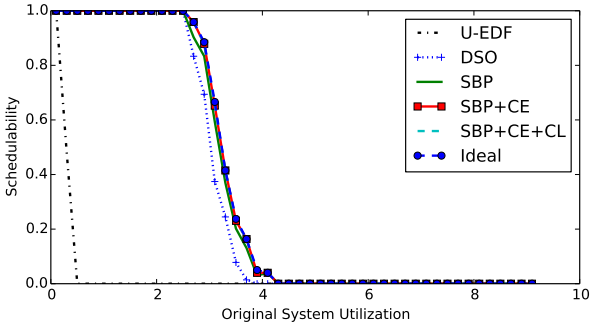
A-Heavy, Long, Light, Mod., Light, Large



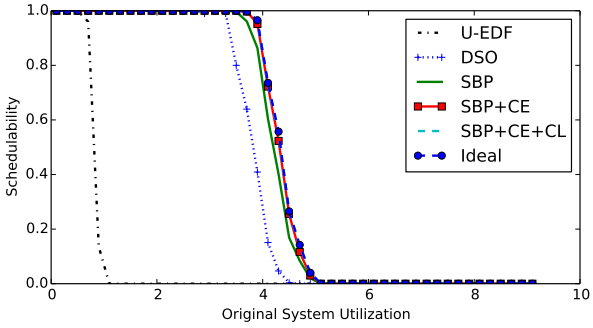
All-Mod., Long, Heavy, Heavy, Light, Small



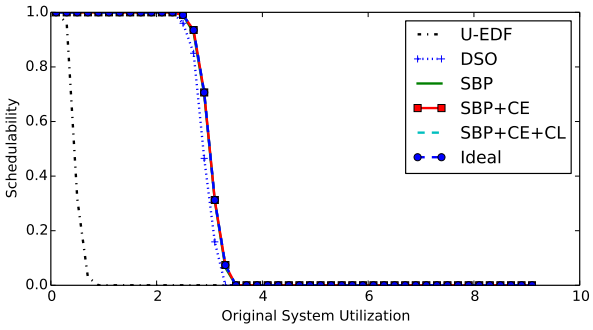
All-Mod., Long, Light, Heavy, Light, Small



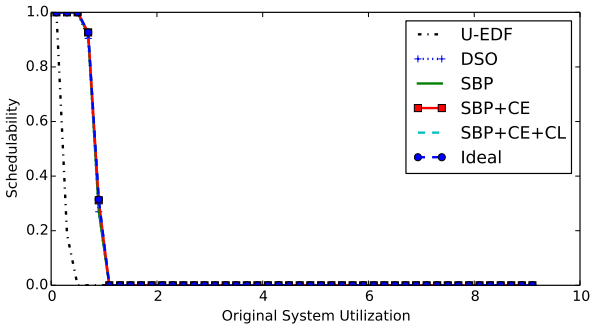
AC-Mod., Cont., Mod., Mod., Light, Large



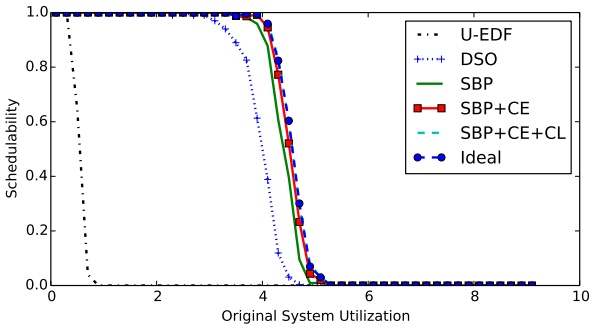
AB-Mod., Short, Heavy, Light, Light, Large



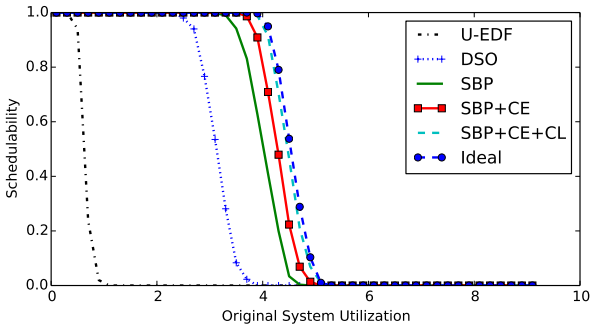
BC-Mod., Long, Light, Mod., Light, Small



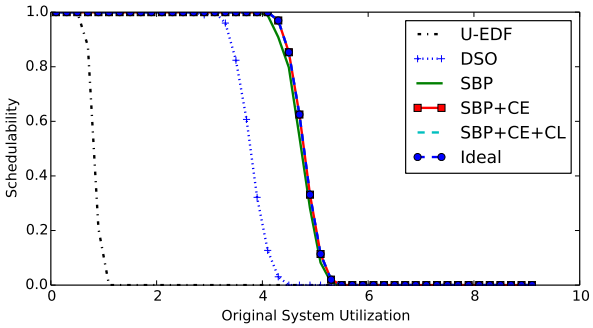
A-Heavy, Short, Light, Heavy, Light, Small



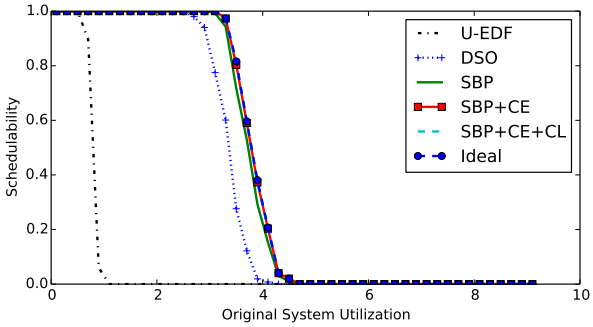
All-Mod., Short, Heavy, Heavy, Light, Small



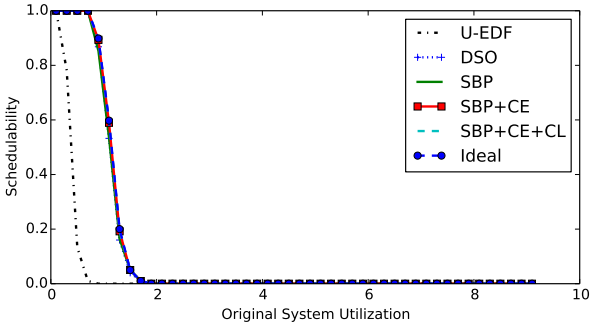
AB-Mod., Long, Mod., Mod., Heavy, Large



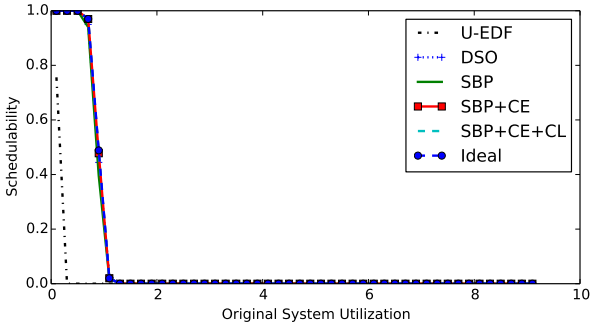
AB-Mod., Long, Mod., Light, Light, Small



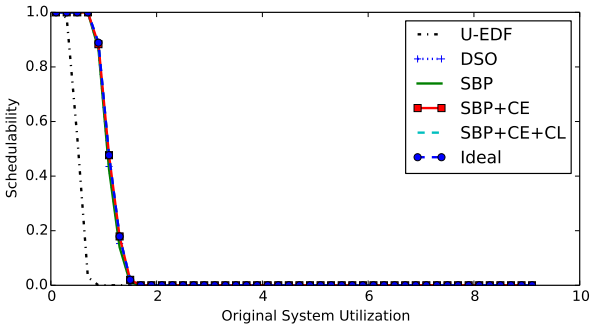
A-Heavy, Short, Heavy, Light, Light, Small



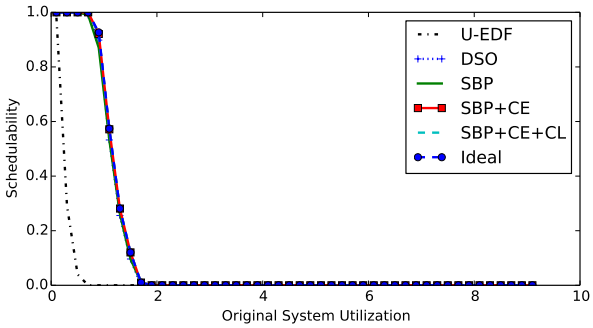
C-Heavy, Short, Light, Mod., Light, Large



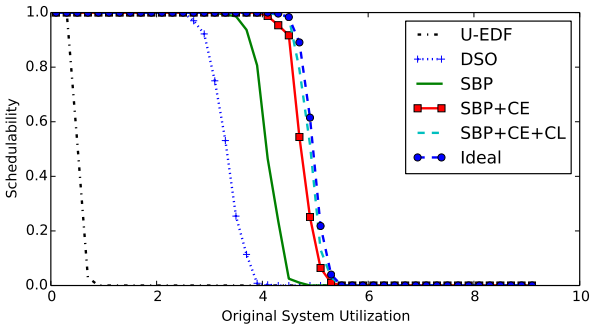
AC-Mod., Cont., Light, Heavy, Heavy, Small



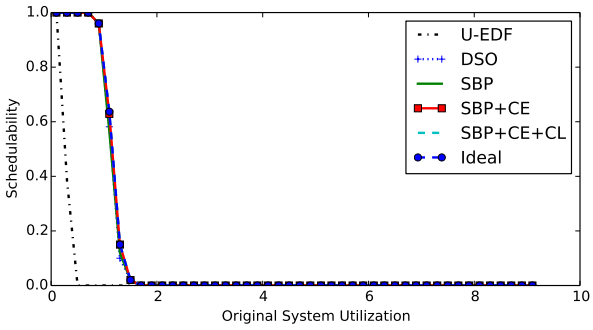
C-Heavy, Short, Light, Light, Heavy, Small



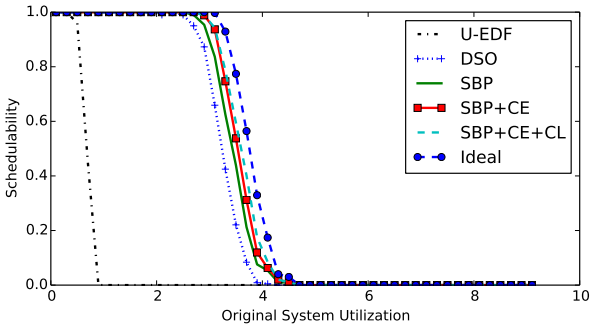
C-Heavy, Cont., Light, Mod., Light, Large



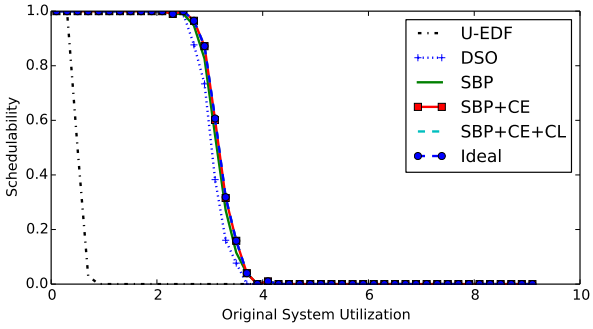
All-Mod., Long, Mod., Heavy, Heavy, Small



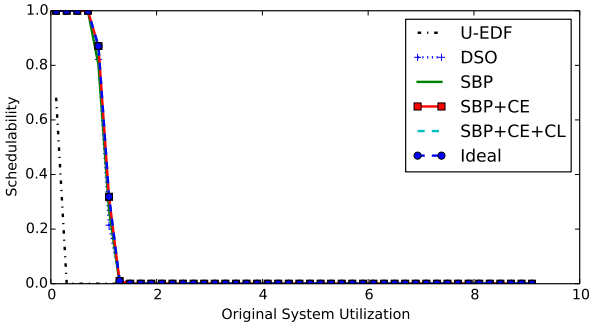
BC-Mod., Short, Light, Heavy, Light, Small



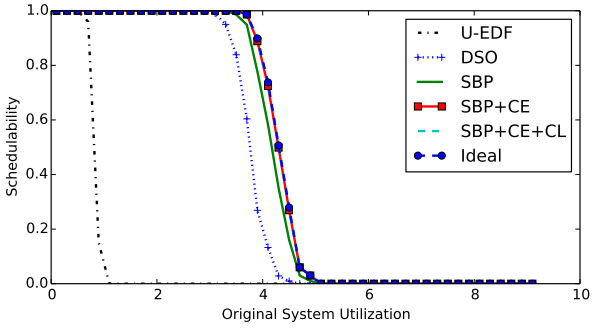
A-Heavy, Short, Heavy, Mod., Heavy, Large



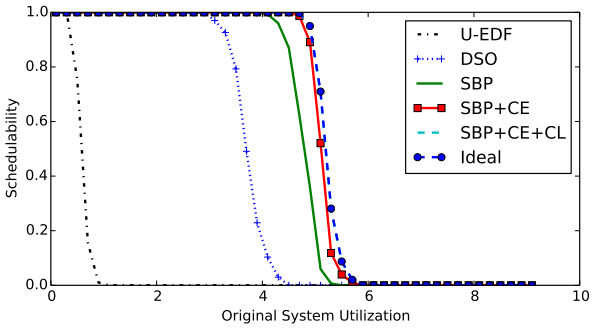
AC-Mod., Short, Mod., Mod., Light, Small



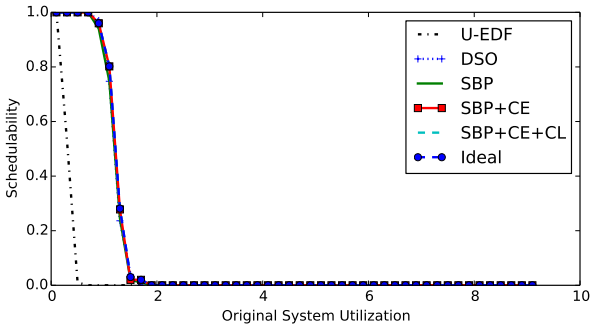
All-Mod., Cont., Light, Heavy, Light, Small



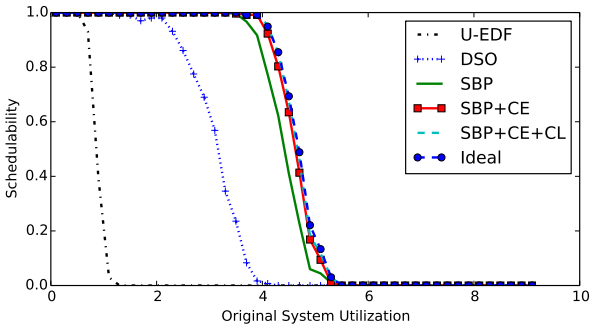
AB-Mod., Short, Heavy, Light, Light, Small



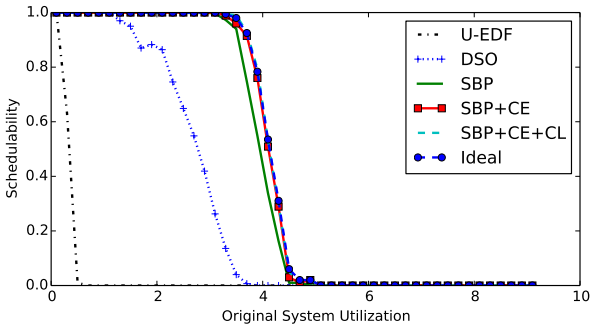
All-Mod., Long, Mod., Mod., Heavy, Small



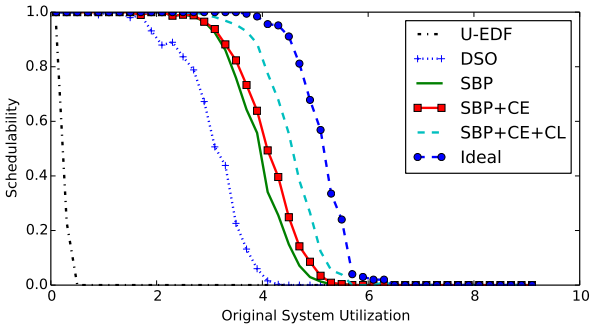
B-Heavy, Short, Light, Heavy, Light, Large



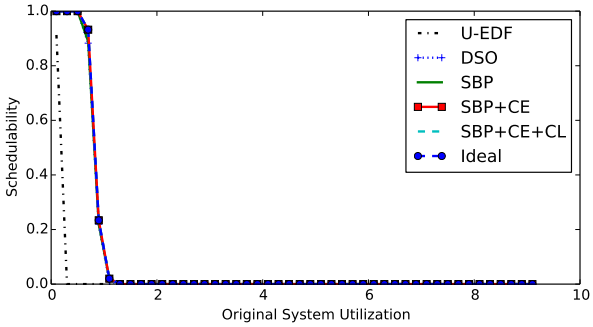
AB-Mod., Long, Heavy, Light, Heavy, Small



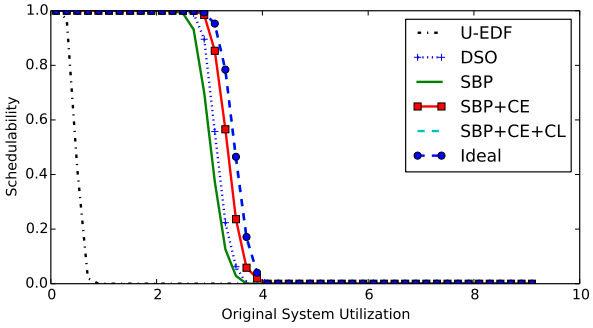
AB-Mod., Cont., Heavy, Mod., Light, Small



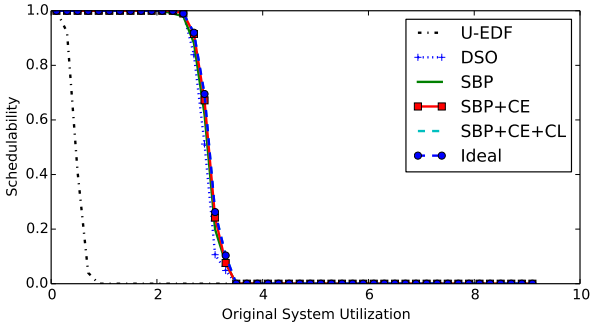
BC-Mod., Cont., Heavy, Heavy, Heavy, Large



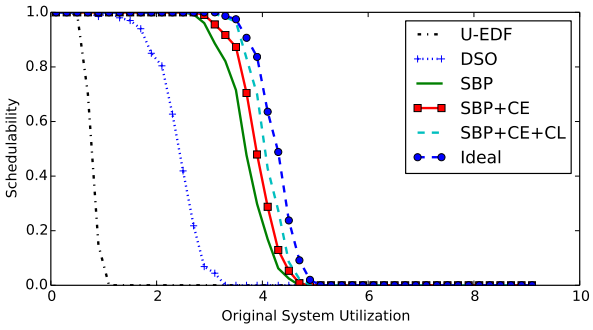
A-Heavy, Cont., Light, Mod., Light, Small



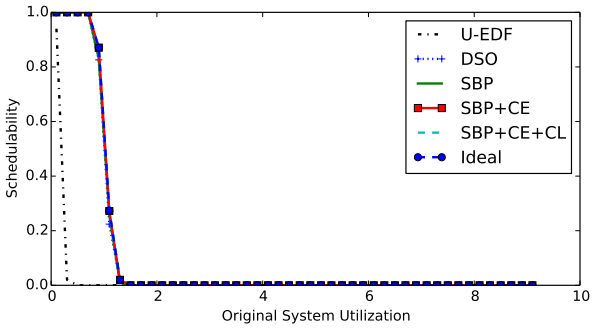
B-Heavy, Short, Mod., Heavy, Heavy, Small



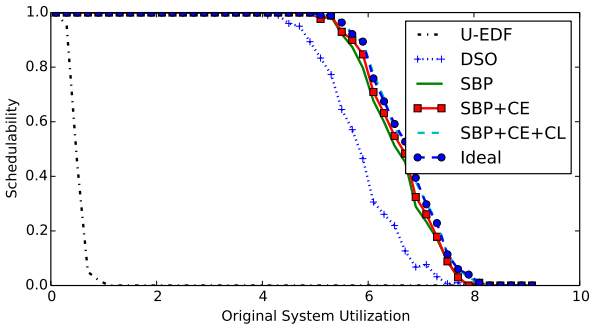
BC-Mod., Long, Light, Mod., Heavy, Large



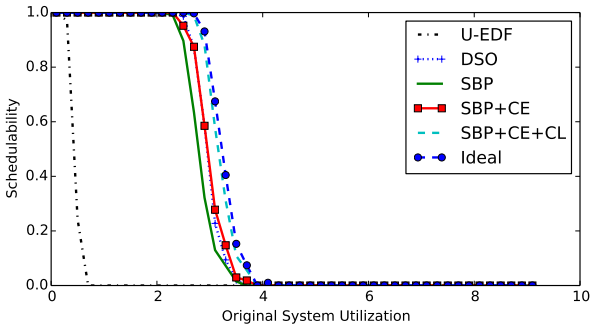
B-Heavy, Long, Heavy, Mod., Heavy, Large



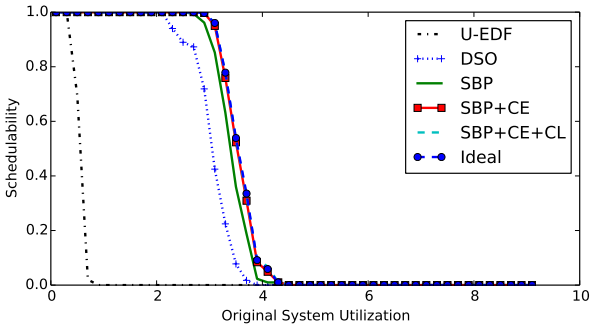
All-Mod., Cont., Light, Mod., Light, Small



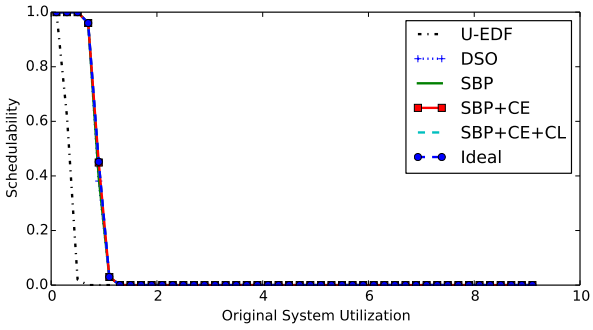
C-Heavy, Short, Heavy, Heavy, Light, Large



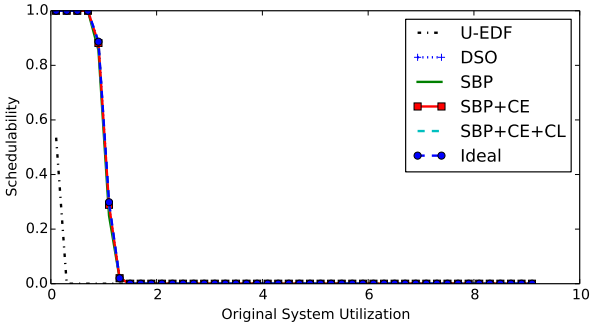
AB-Mod., Short, Mod., Heavy, Heavy, Large



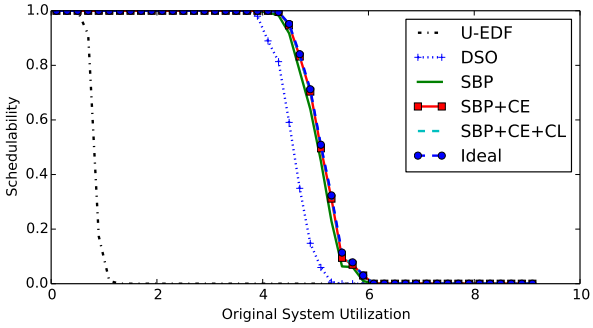
A-Heavy, Short, Heavy, Heavy, Light, Small



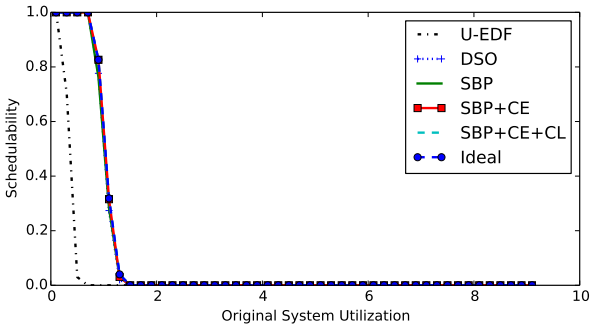
AC-Mod., Short, Light, Mod., Light, Small



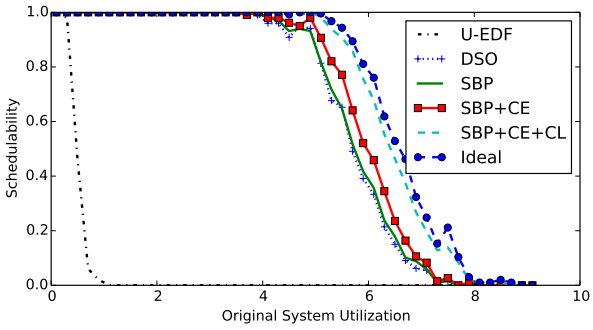
AB-Mod., Cont., Light, Heavy, Heavy, Small



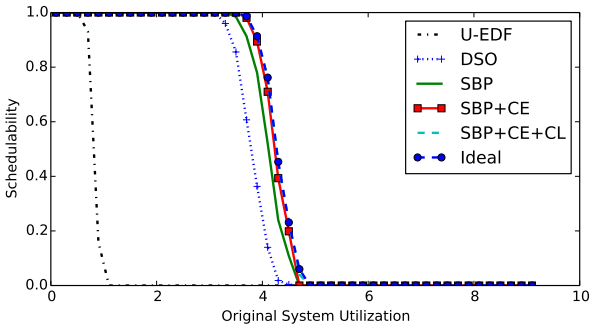
AC-Mod., Short, Heavy, Light, Light, Large



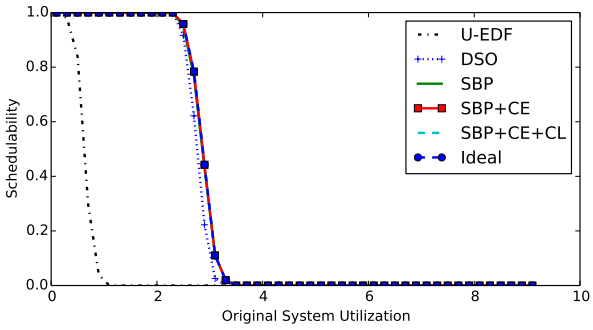
All-Mod., Short, Light, Mod., Light, Large



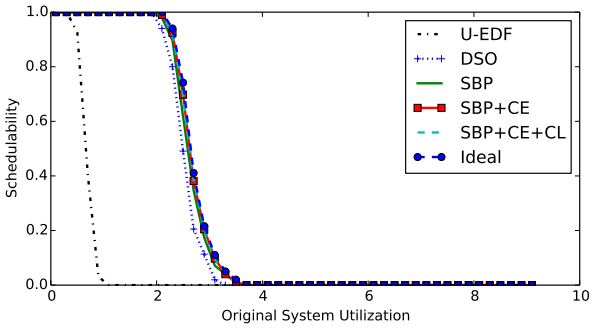
C-Heavy, Short, Heavy, Heavy, Heavy, Small



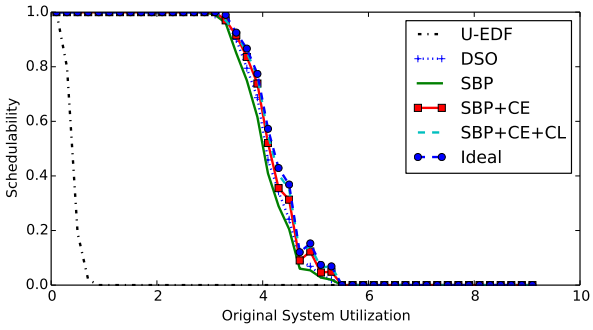
AB-Mod., Short, Heavy, Light, Heavy, Small



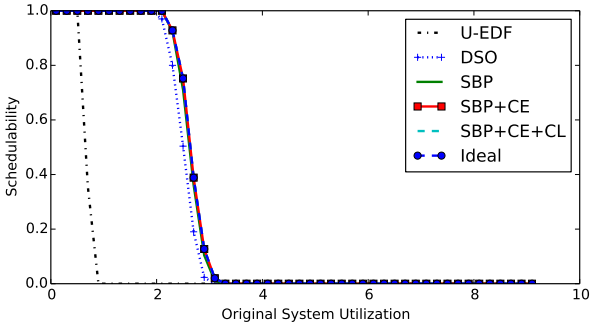
All-Mod., Long, Light, Light, Light, Large



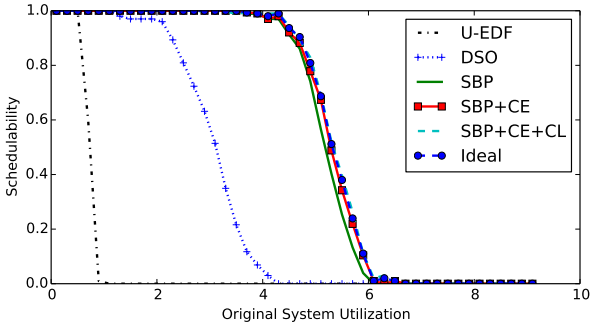
A-Heavy, Short, Mod., Light, Heavy, Large



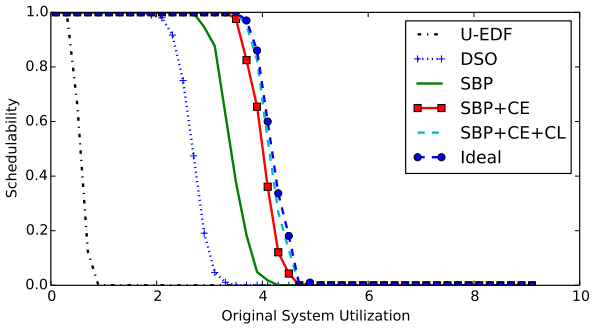
C-Heavy, Short, Mod., Heavy, Heavy, Small



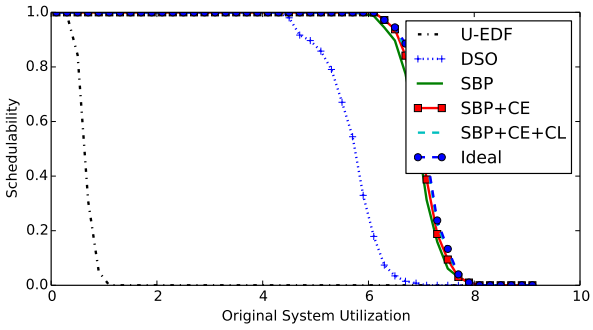
A-Heavy, Short, Mod., Light, Light, Small



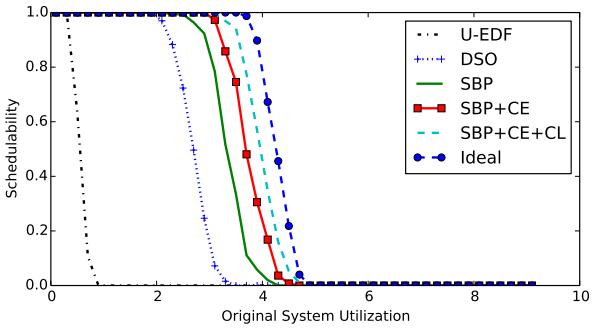
AC-Mod., Long, Heavy, Mod., Light, Small



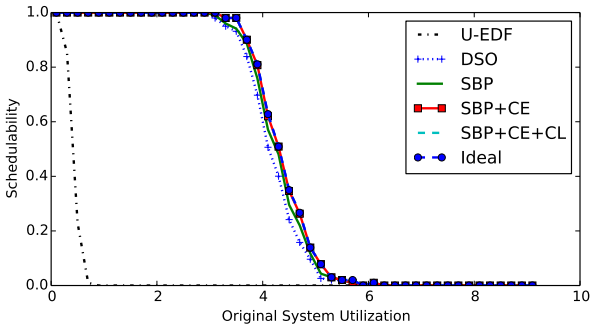
B-Heavy, Long, Mod., Heavy, Heavy, Small



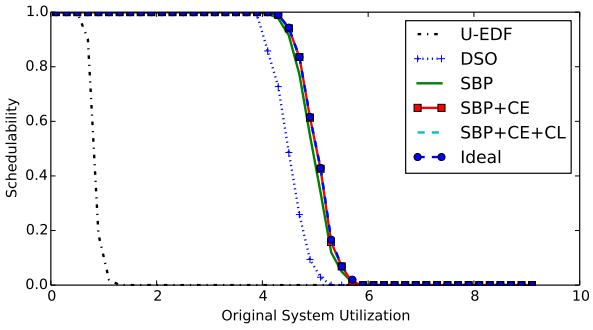
C-Heavy, Long, Mod., Mod., Light, Large



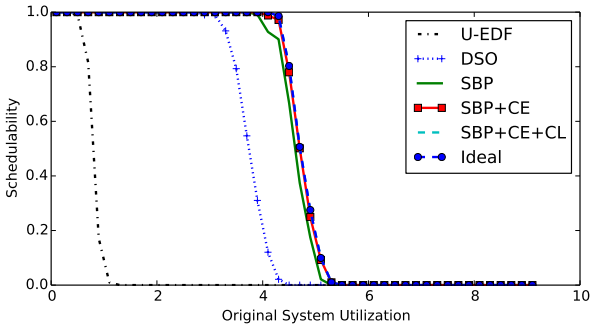
B-Heavy, Long, Mod., Heavy, Heavy, Large



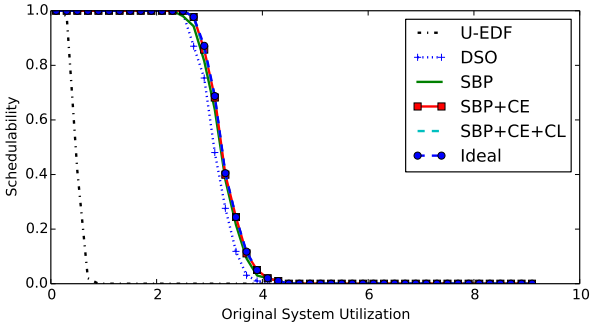
C-Heavy, Short, Mod., Heavy, Light, Small



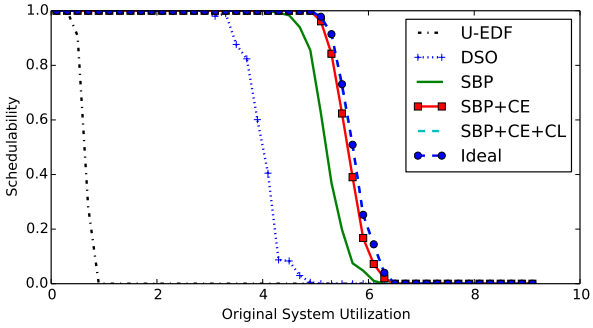
AC-Mod., Short, Heavy, Light, Light, Small



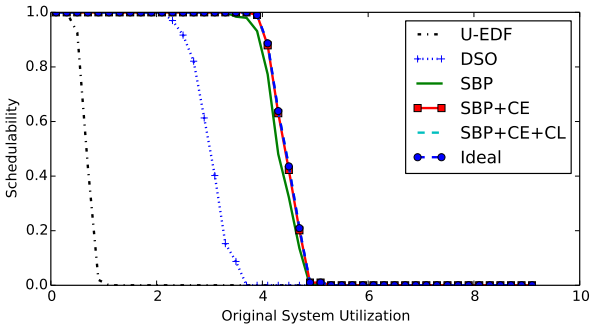
AB-Mod., Long, Mod., Light, Heavy, Small



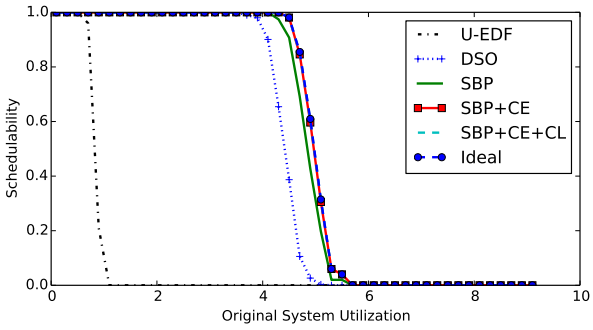
AC-Mod., Cont., Mod., Light, Light, Large



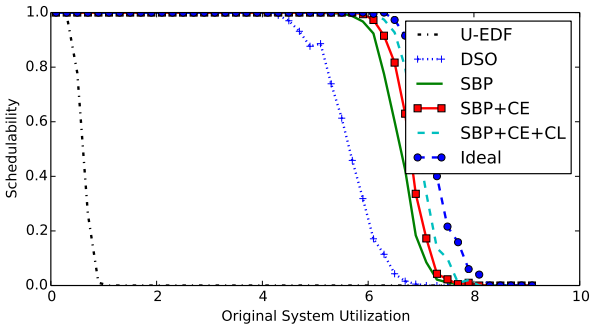
BC-Mod., Long, Mod., Mod., Heavy, Small



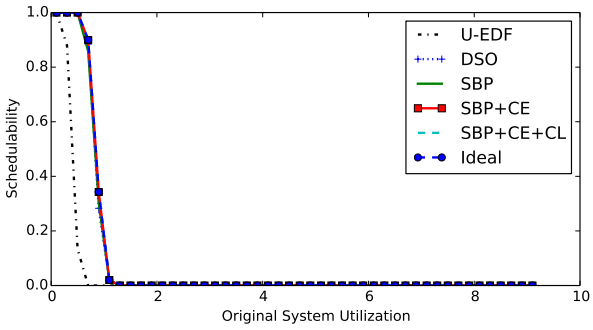
B-Heavy, Long, Mod., Mod., Light, Small



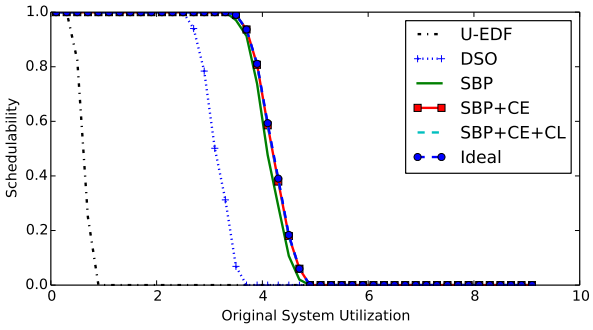
All-Mod., Short, Heavy, Light, Light, Small



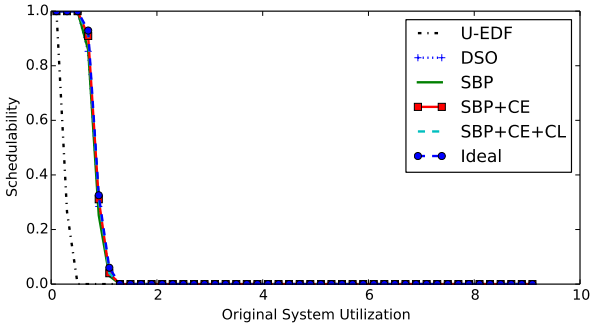
C-Heavy, Long, Mod., Mod., Heavy, Large



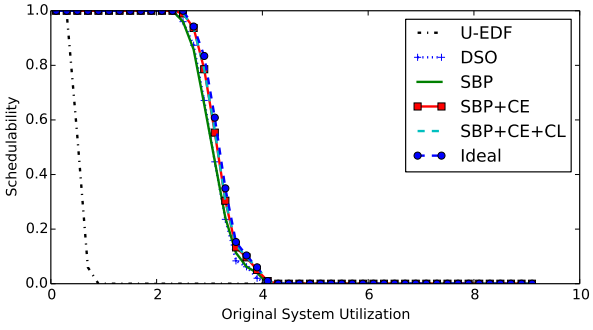
A-Heavy, Short, Light, Light, Heavy, Large



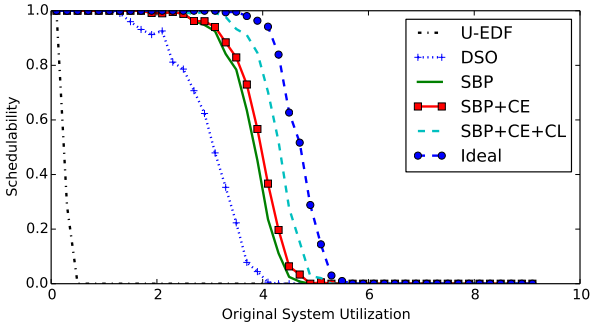
A-Heavy, Long, Mod., Mod., Light, Large



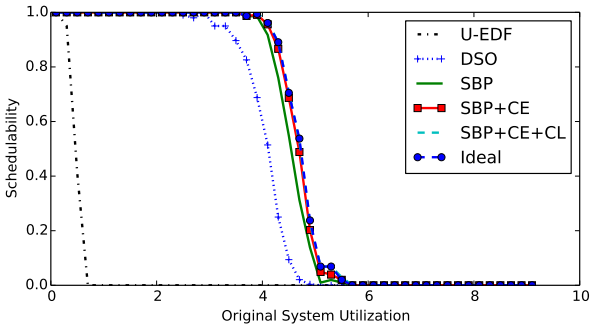
A-Heavy, Short, Light, Heavy, Heavy, Large



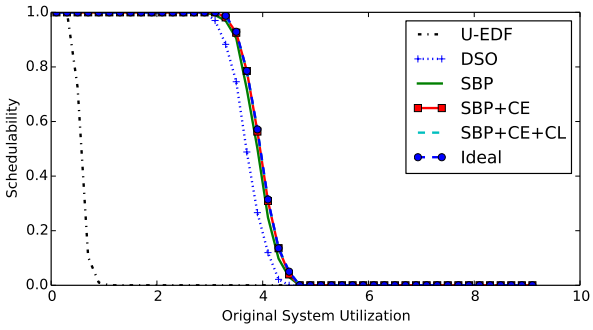
AC-Mod., Short, Mod., Mod., Heavy, Large



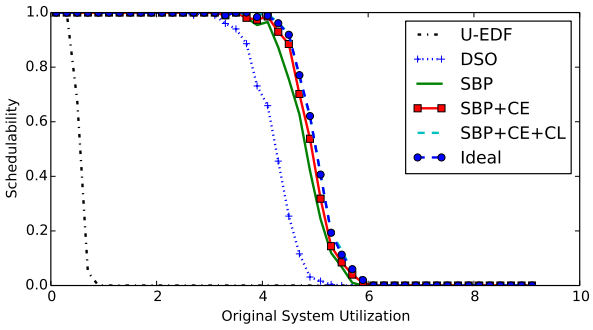
All-Mod., Cont., Heavy, Heavy, Heavy, Large



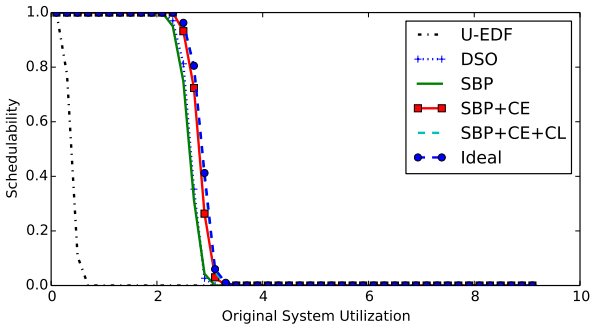
AC-Mod., Short, Heavy, Heavy, Light, Small



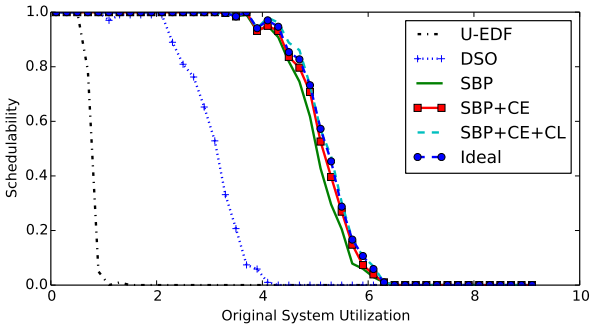
BC-Mod., Short, Mod., Mod., Light, Small



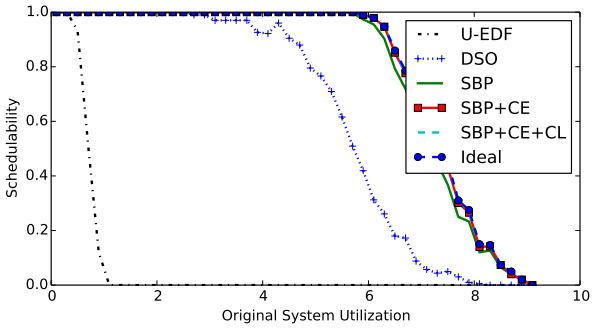
BC-Mod., Short, Heavy, Heavy, Light, Large



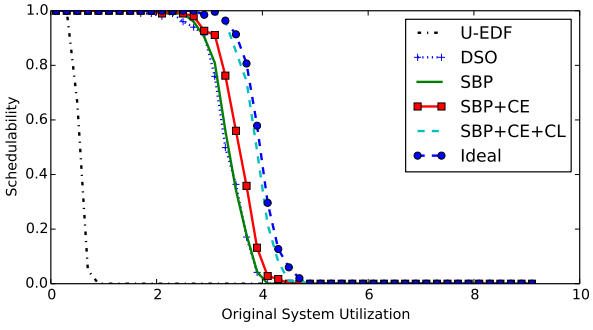
B-Heavy, Long, Light, Heavy, Heavy, Large



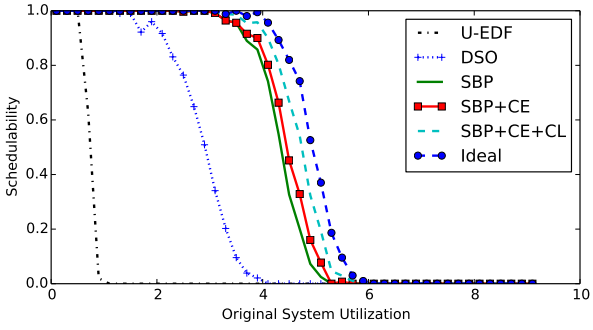
BC-Mod., Long, Heavy, Mod., Light, Small



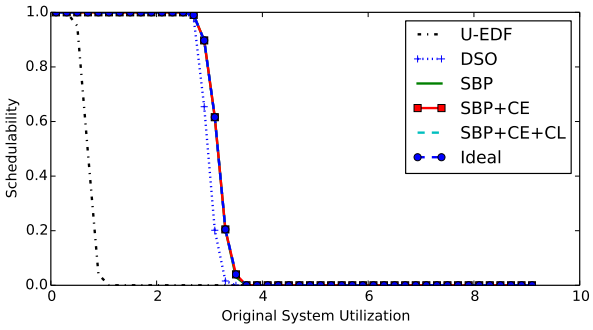
C-Heavy, Cont., Heavy, Light, Light, Small



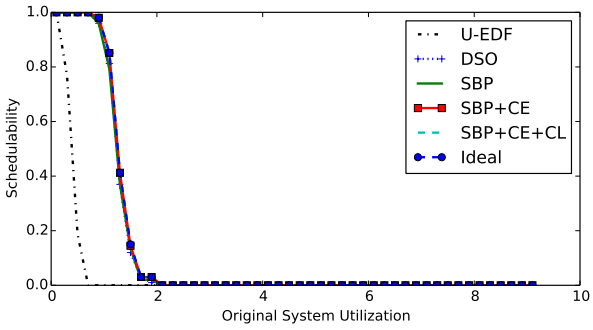
AB-Mod., Short, Heavy, Heavy, Heavy, Small



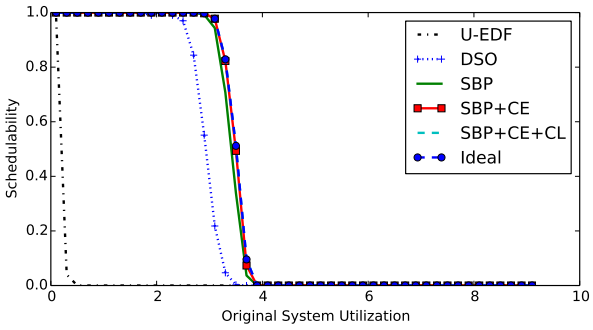
All-Mod., Long, Heavy, Mod., Heavy, Large



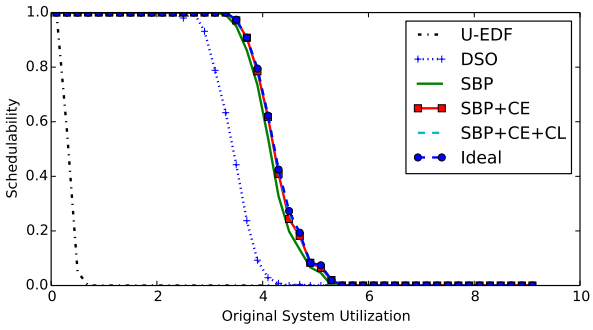
B-Heavy, Long, Light, Light, Light, Small



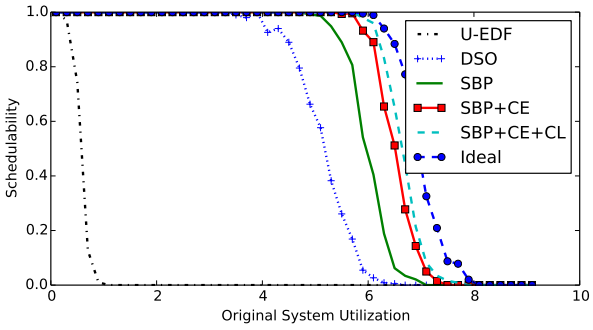
BC-Mod., Cont., Light, Light, Heavy, Large



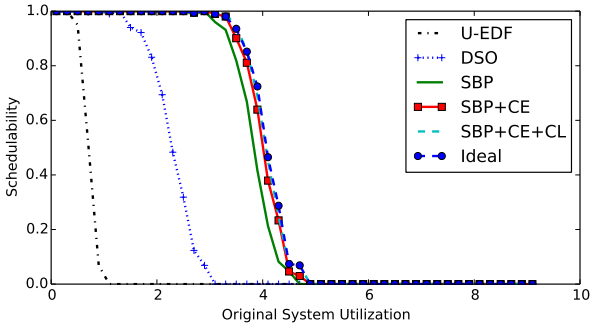
All-Mod., Cont., Mod., Heavy, Light, Small



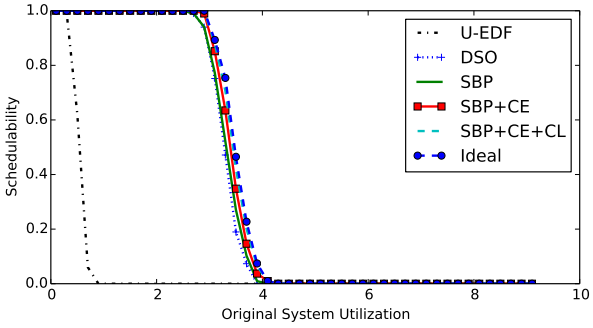
BC-Mod., Cont., Mod., Mod., Light, Large



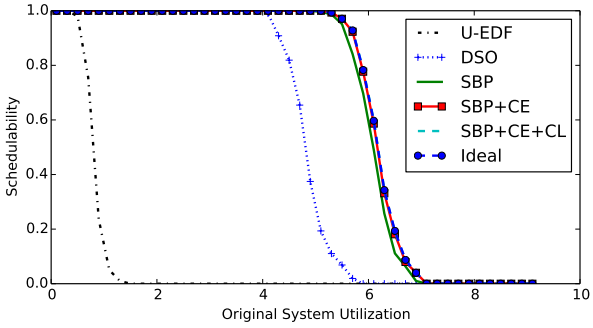
C-Heavy, Long, Mod., Heavy, Heavy, Small



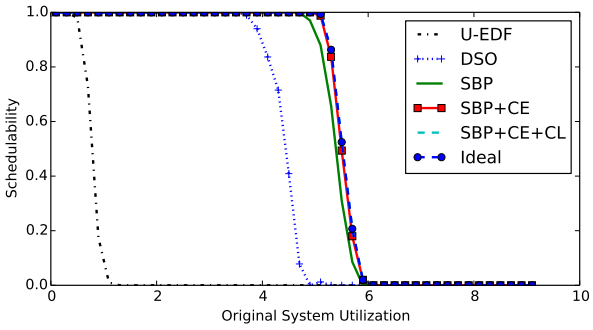
B-Heavy, Long, Heavy, Heavy, Light, Small



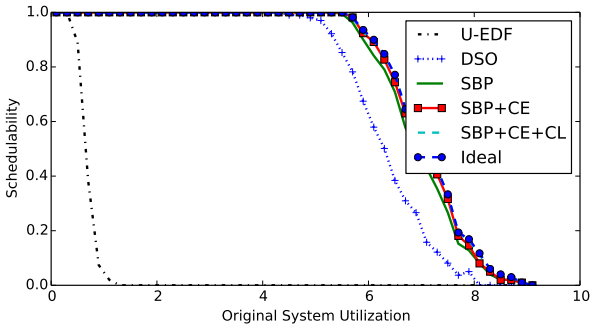
All-Mod., Short, Mod., Mod., Heavy, Large



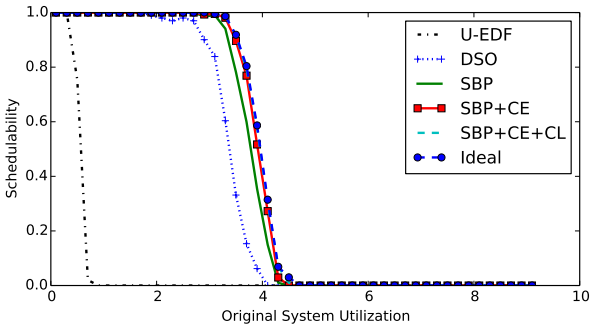
BC-Mod., Long, Mod., Light, Light, Large



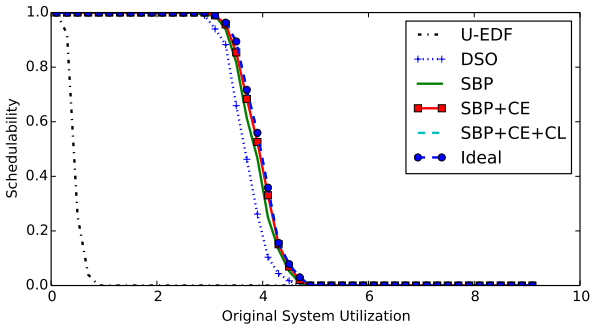
All-Mod., Long, Mod., Light, Heavy, Small



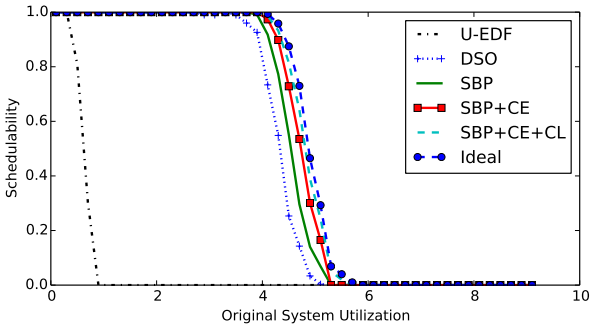
C-Heavy, Short, Heavy, Mod., Light, Large



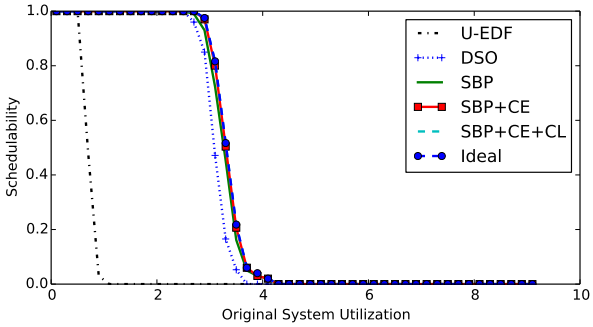
AB-Mod., Short, Heavy, Heavy, Light, Small



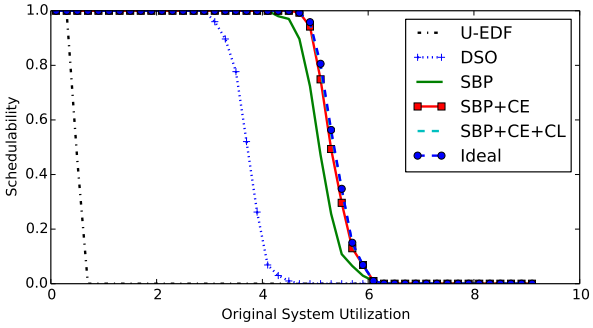
BC-Mod., Short, Mod., Heavy, Light, Large



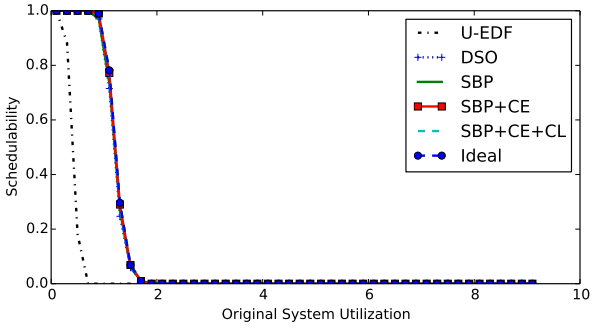
AC-Mod., Short, Heavy, Mod., Heavy, Small



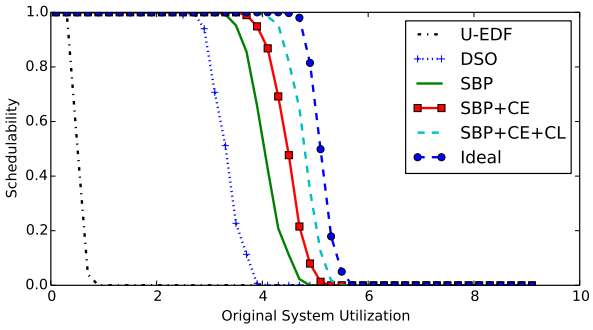
AB-Mod., Short, Mod., Light, Light, Large



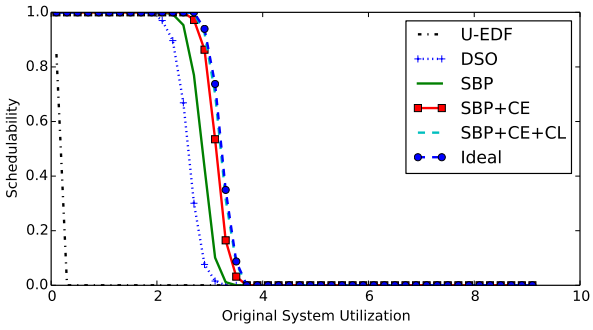
AC-Mod., Long, Mod., Heavy, Light, Large



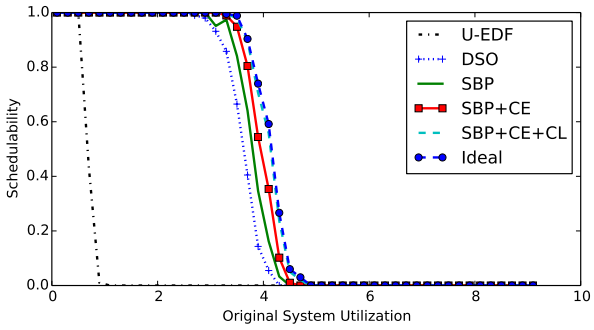
B-Heavy, Short, Light, Mod., Light, Large



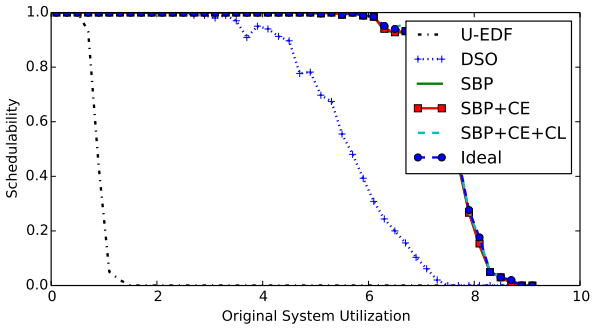
All-Mod., Long, Mod., Heavy, Heavy, Large



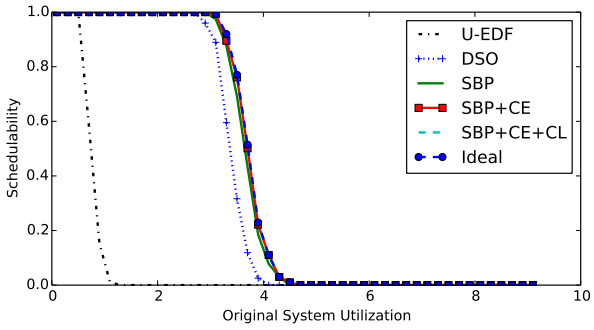
AB-Mod., Cont., Mod., Heavy, Heavy, Small



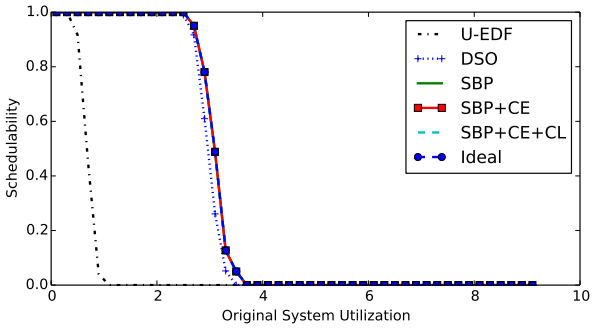
AB-Mod., Short, Heavy, Mod., Heavy, Small



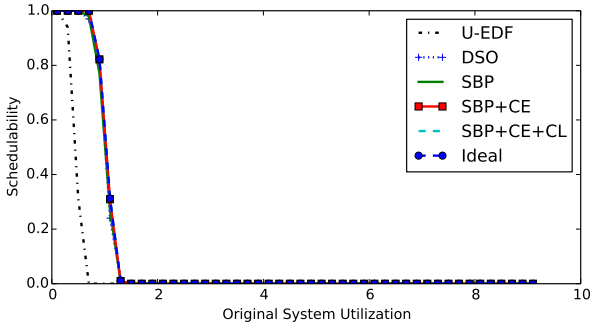
C-Heavy, Long, Heavy, Light, Light, Large



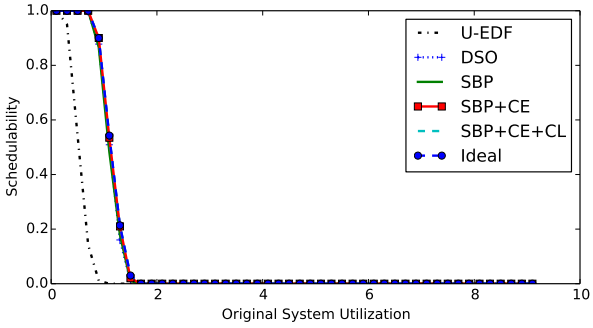
B-Heavy, Short, Mod., Light, Light, Large



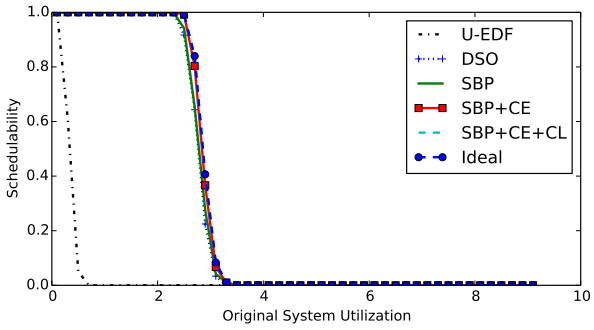
BC-Mod., Long, Light, Light, Light, Small



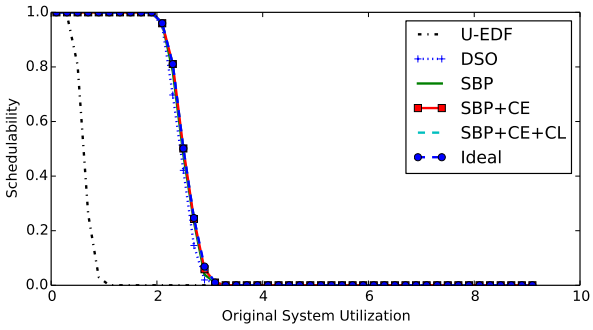
All-Mod., Short, Light, Light, Light, Large



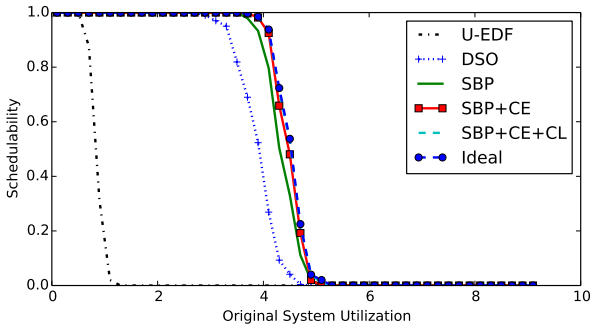
C-Heavy, Short, Light, Light, Light, Large



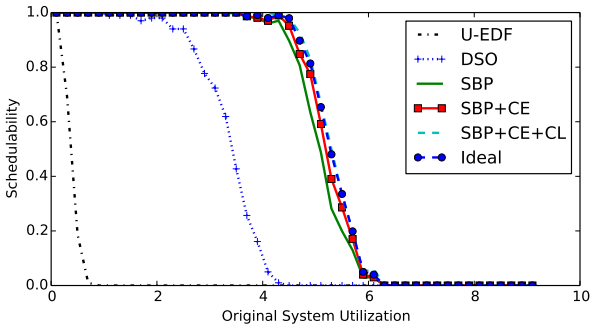
BC-Mod., Long, Light, Heavy, Heavy, Small



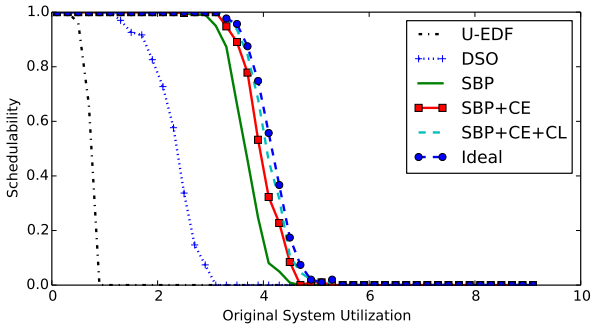
AC-Mod., Long, Light, Light, Heavy, Large



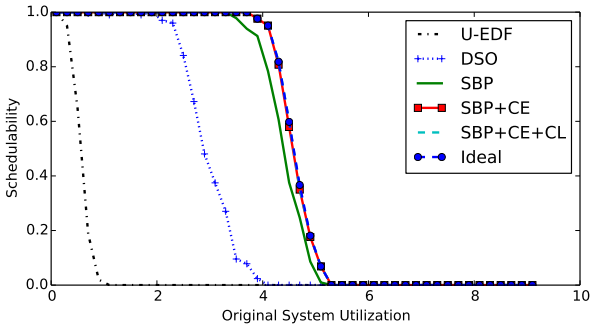
B-Heavy, Short, Heavy, Light, Heavy, Small



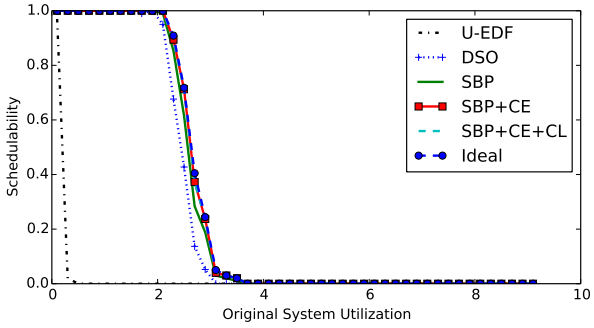
BC-Mod., Cont., Heavy, Mod., Light, Large



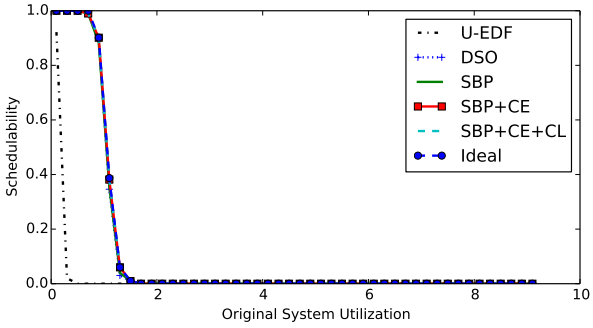
A-Heavy, Long, Heavy, Mod., Heavy, Small



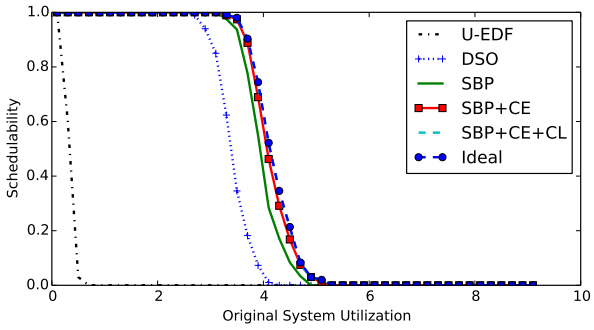
B-Heavy, Cont., Heavy, Light, Light, Small



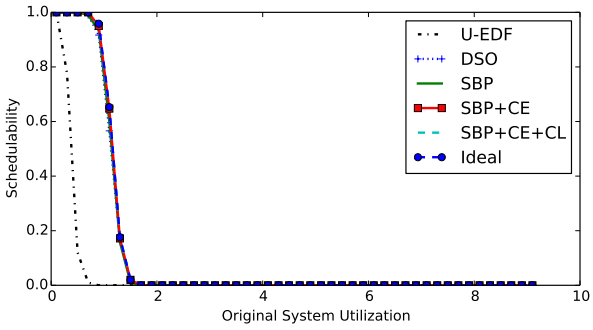
A-Heavy, Cont., Mod., Heavy, Light, Large



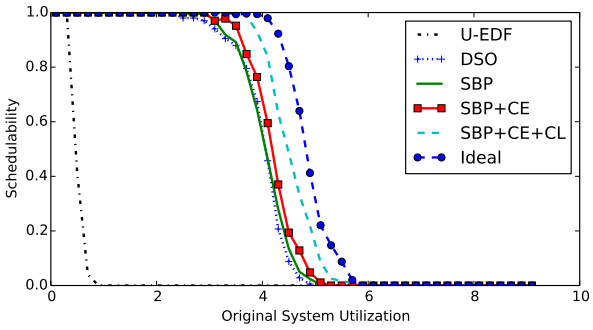
AB-Mod., Cont., Light, Mod., Light, Large



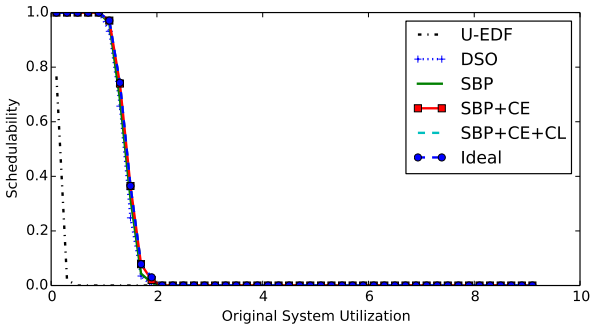
BC-Mod., Cont., Mod., Mod., Heavy, Small



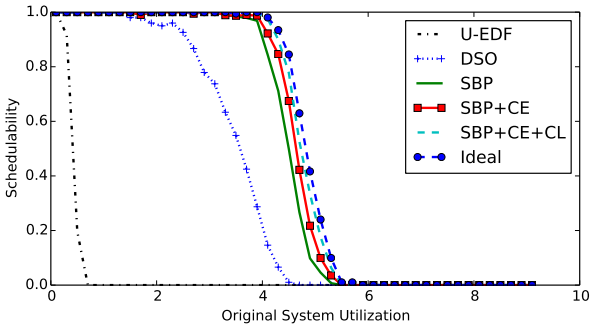
BC-Mod., Short, Light, Mod., Light, Large



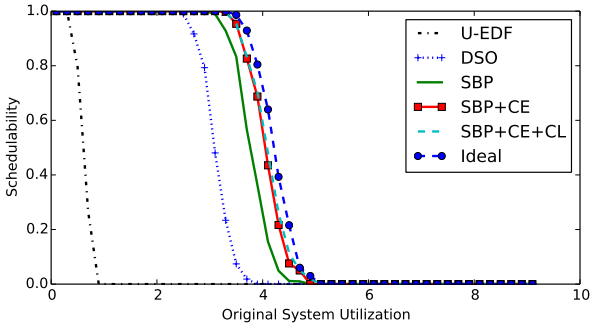
AC-Mod., Short, Heavy, Heavy, Heavy, Large



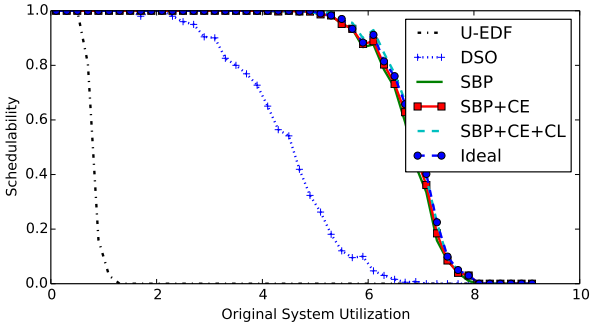
B-Heavy, Cont., Light, Heavy, Light, Small



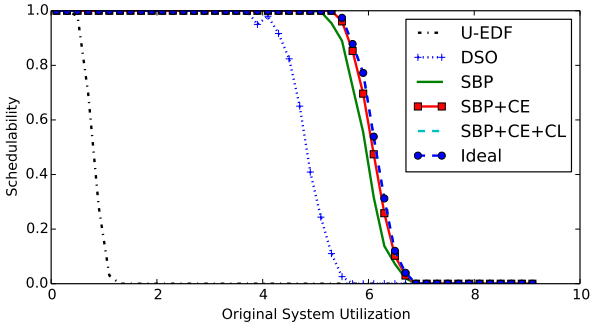
AC-Mod., Cont., Heavy, Mod., Heavy, Small



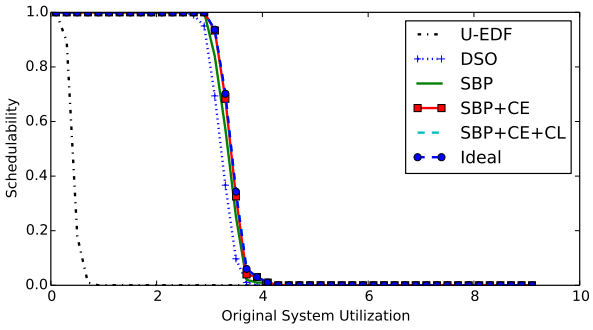
A-Heavy, Long, Mod., Mod., Heavy, Large



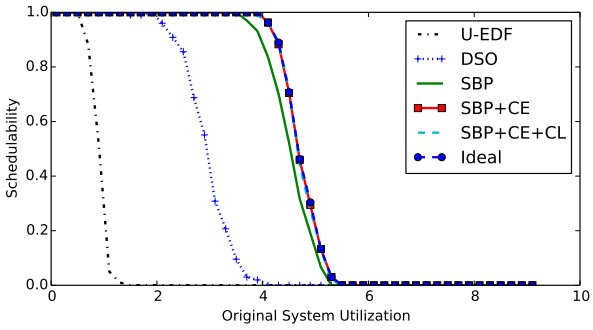
C-Heavy, Long, Heavy, Mod., Light, Small



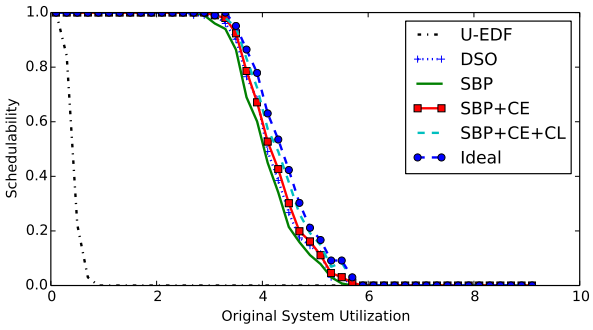
BC-Mod., Long, Mod., Light, Heavy, Large



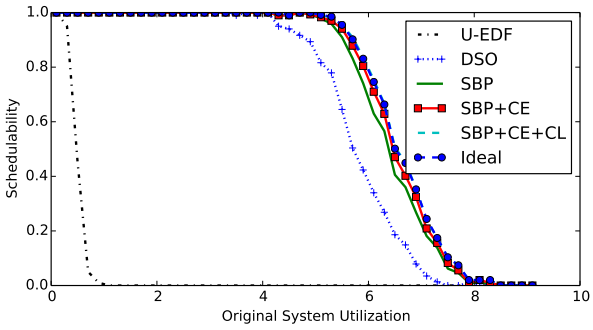
All-Mod., Short, Mod., Heavy, Light, Small



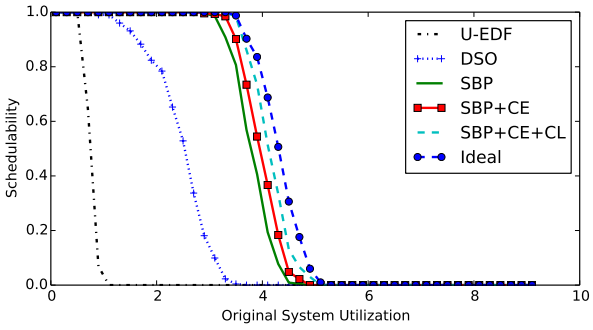
B-Heavy, Long, Heavy, Light, Light, Small



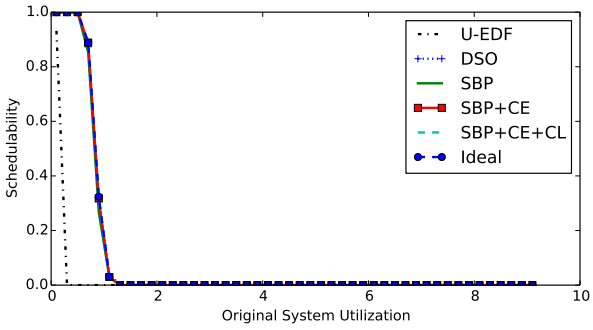
C-Heavy, Short, Mod., Heavy, Heavy, Large



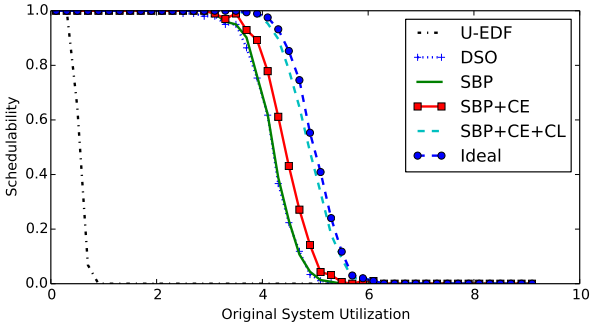
C-Heavy, Short, Heavy, Heavy, Light, Small



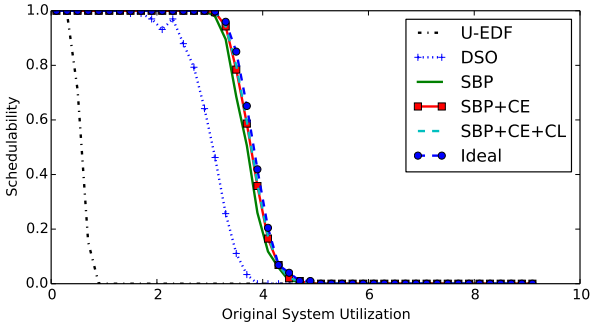
AB-Mod., Long, Heavy, Mod., Heavy, Large



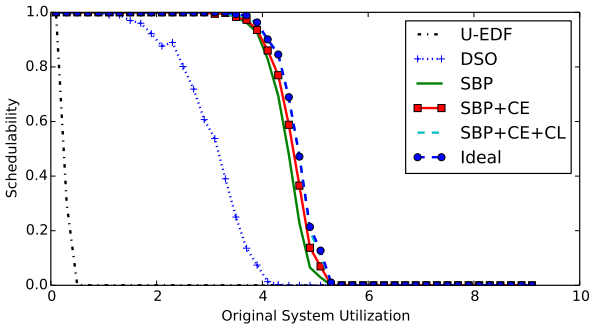
A-Heavy, Cont., Light, Mod., Light, Large



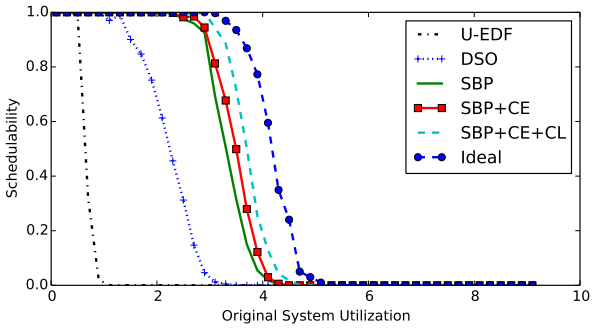
BC-Mod., Short, Heavy, Heavy, Heavy, Small



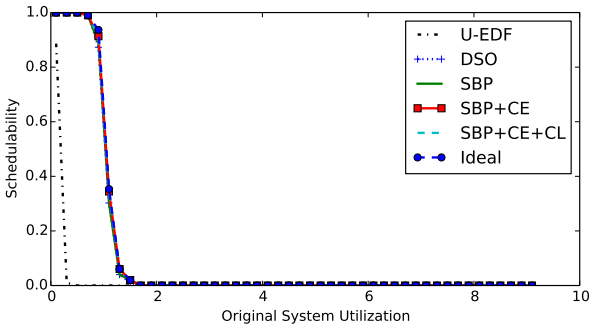
A-Heavy, Cont., Heavy, Light, Heavy, Large



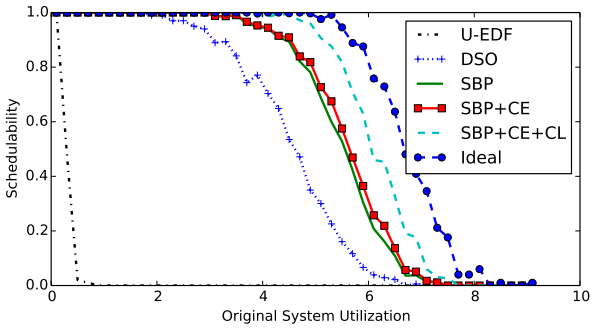
All-Mod., Cont., Heavy, Heavy, Light, Large



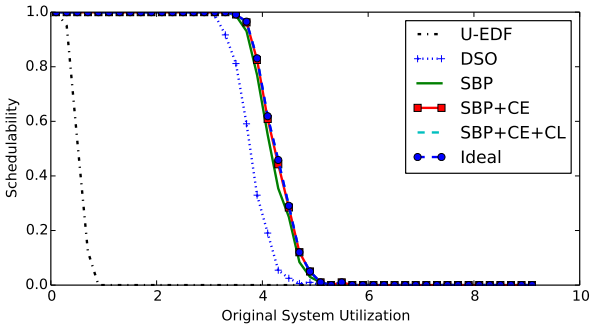
AB-Mod., Long, Heavy, Heavy, Heavy, Large



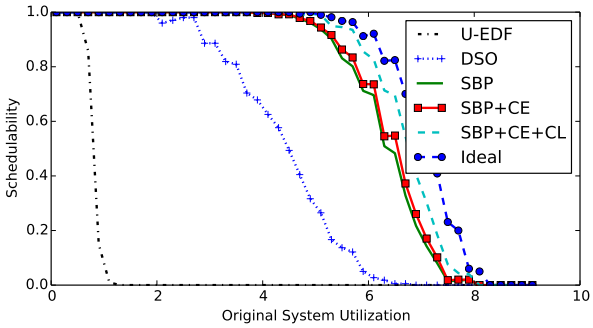
AB-Mod., Cont., Light, Mod., Heavy, Large



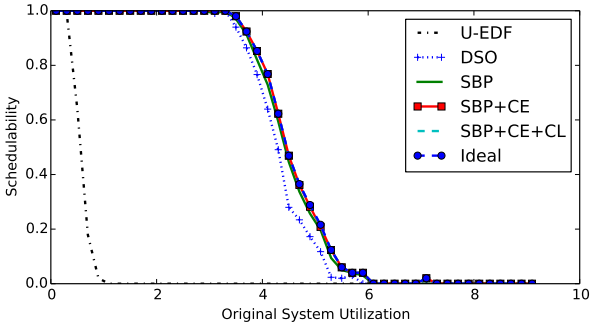
C-Heavy, Cont., Heavy, Heavy, Heavy, Large



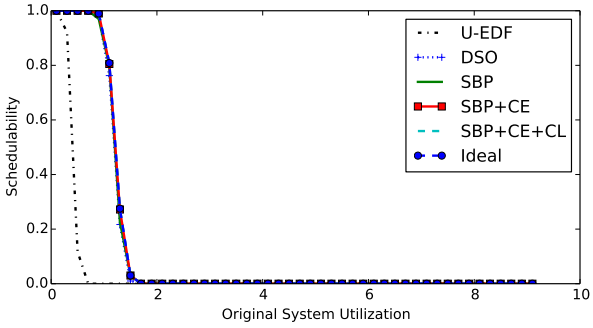
BC-Mod., Cont., Mod., Light, Heavy, Small



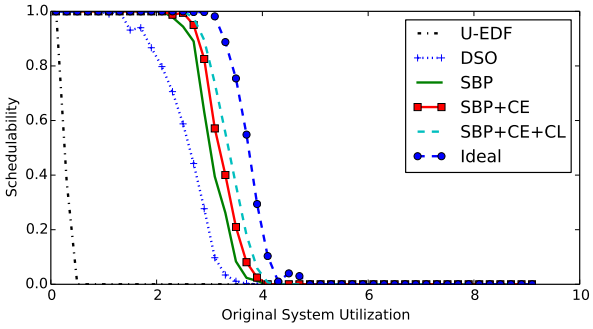
C-Heavy, Long, Heavy, Mod., Heavy, Large



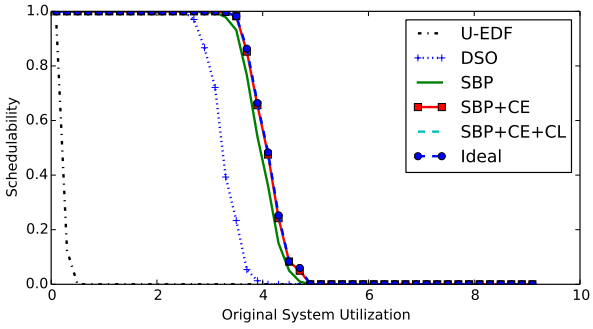
C-Heavy, Cont., Mod., Light, Light, Small



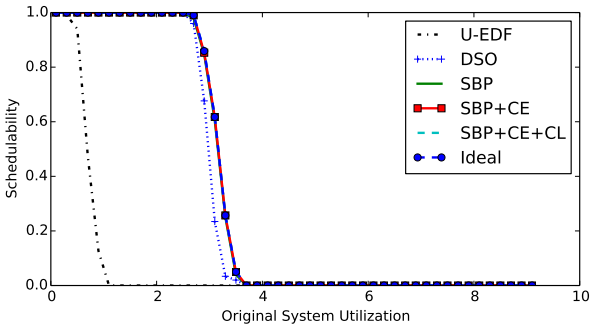
B-Heavy, Short, Light, Mod., Heavy, Small



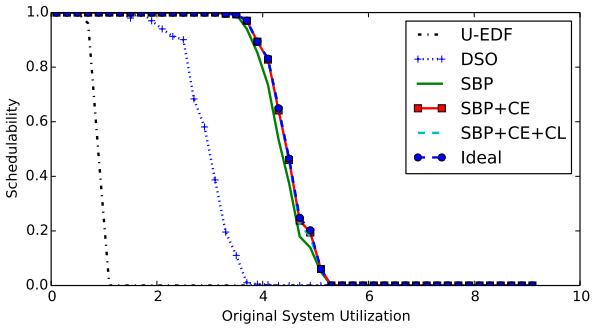
A-Heavy, Cont., Heavy, Heavy, Heavy, Large



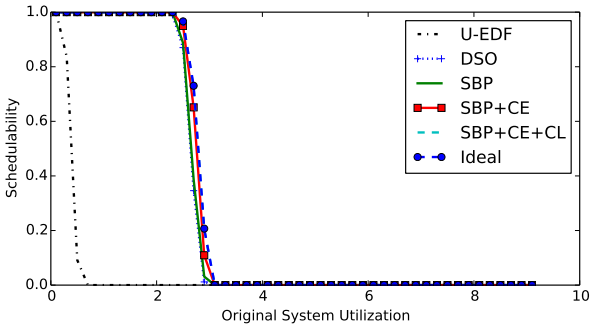
BC-Mod., Cont., Mod., Heavy, Light, Small



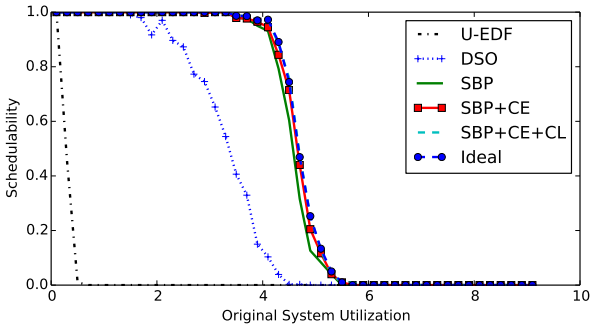
B-Heavy, Long, Light, Light, Light, Large



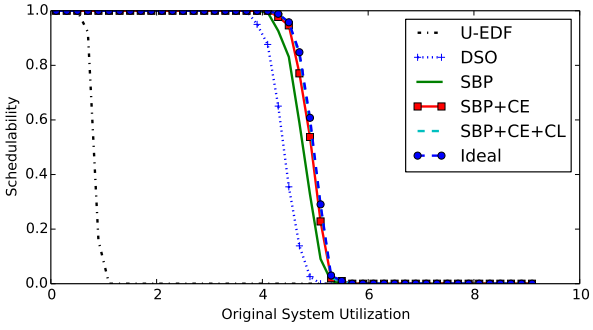
A-Heavy, Long, Heavy, Light, Light, Large



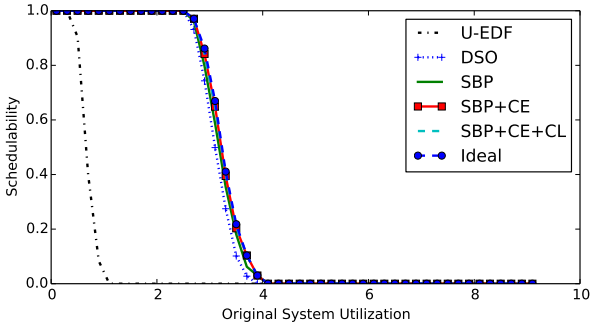
B-Heavy, Long, Light, Heavy, Heavy, Small



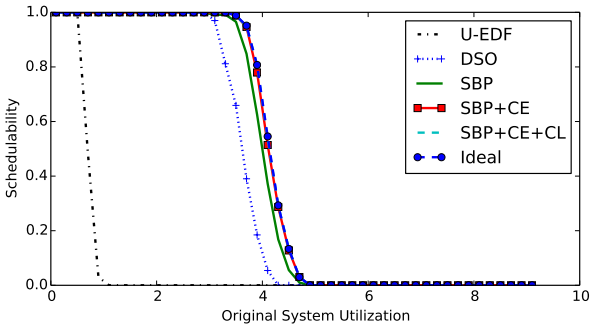
AC-Mod., Cont., Heavy, Heavy, Light, Small



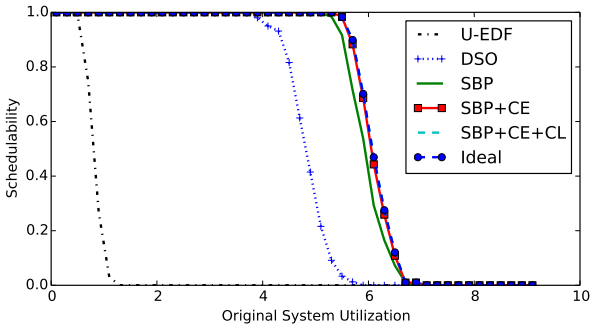
All-Mod., Short, Heavy, Light, Heavy, Small



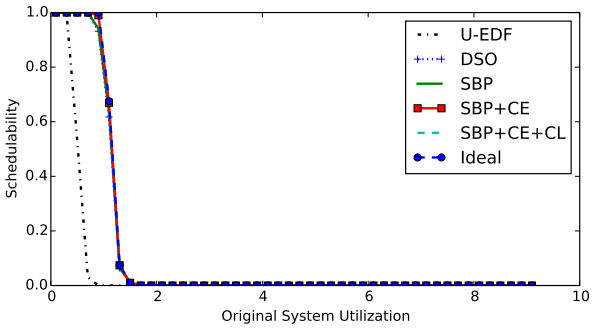
AC-Mod., Short, Mod., Light, Heavy, Large



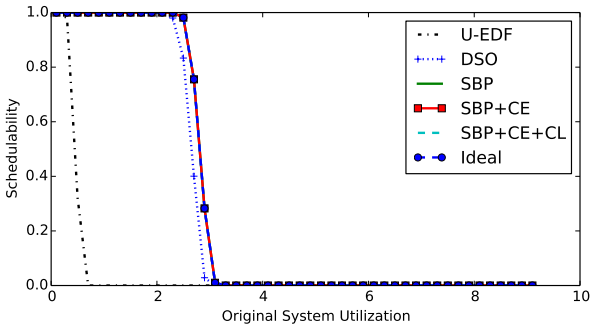
AB-Mod., Short, Heavy, Mod., Light, Small



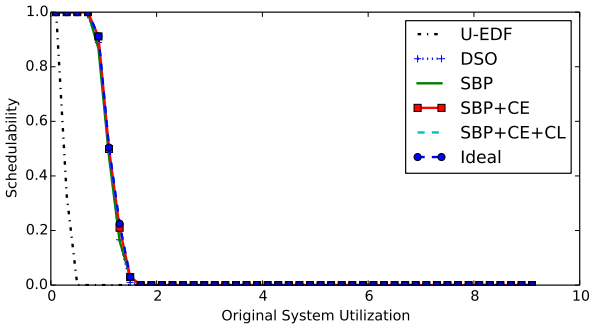
BC-Mod., Long, Mod., Light, Heavy, Small



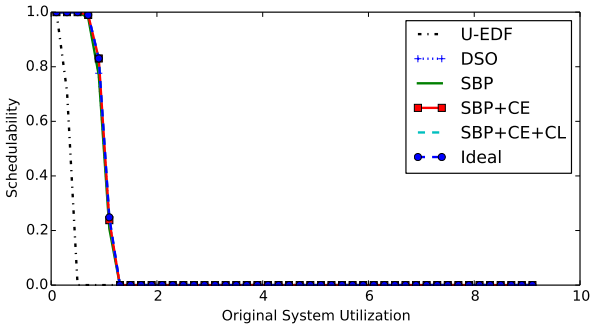
BC-Mod., Short, Light, Light, Heavy, Small



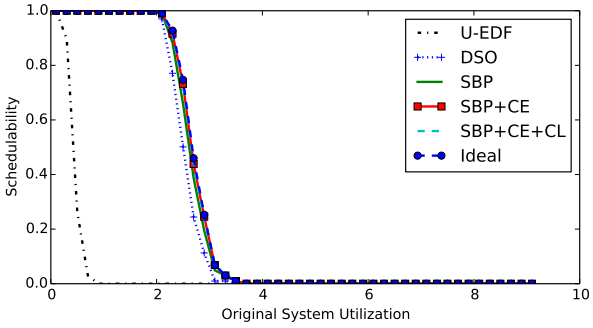
AB-Mod., Long, Light, Mod., Light, Small



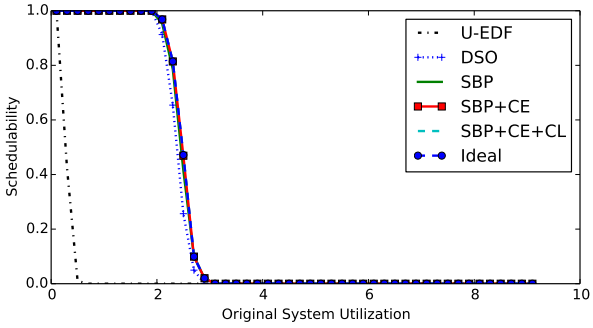
C-Heavy, Short, Light, Heavy, Heavy, Large



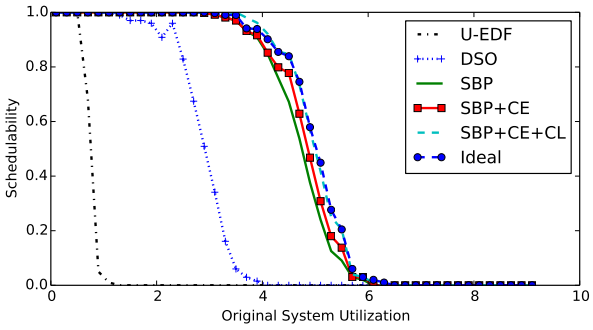
All-Mod., Short, Light, Mod., Heavy, Small



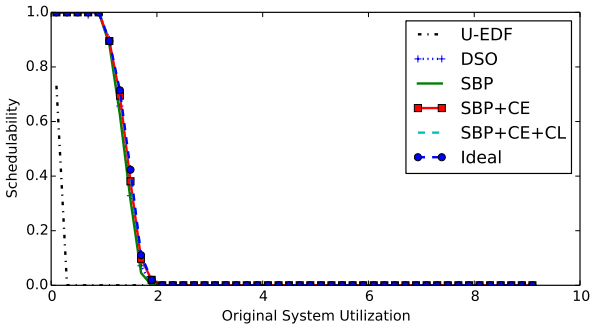
A-Heavy, Cont., Mod., Light, Heavy, Large



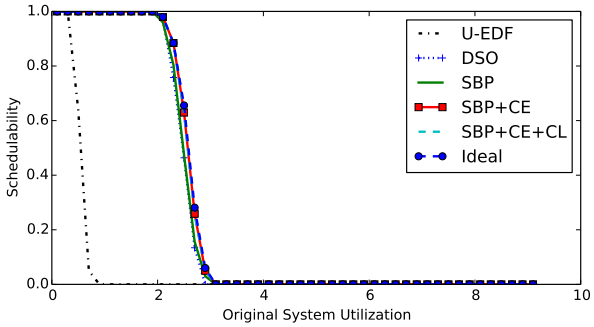
AC-Mod., Long, Light, Heavy, Light, Large



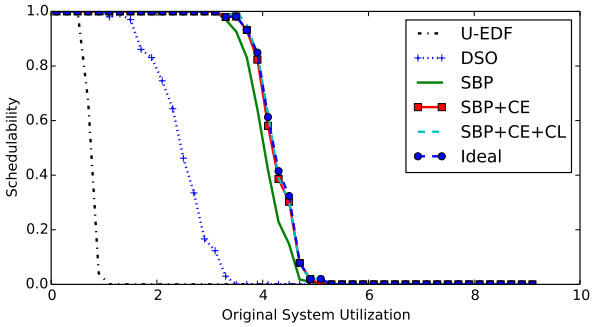
BC-Mod., Long, Heavy, Heavy, Light, Large



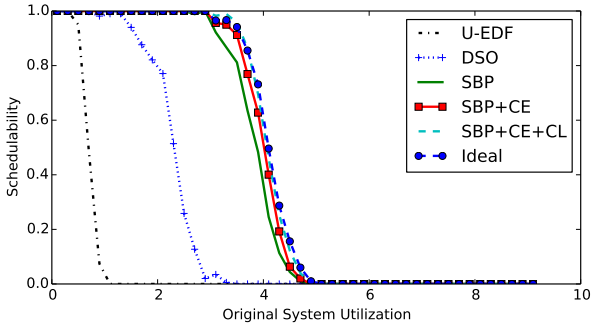
B-Heavy, Cont., Light, Heavy, Heavy, Large



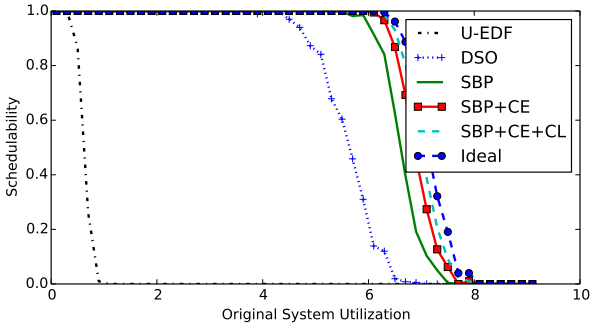
A-Heavy, Short, Mod., Mod., Heavy, Small



AB-Mod., Long, Heavy, Mod., Light, Small

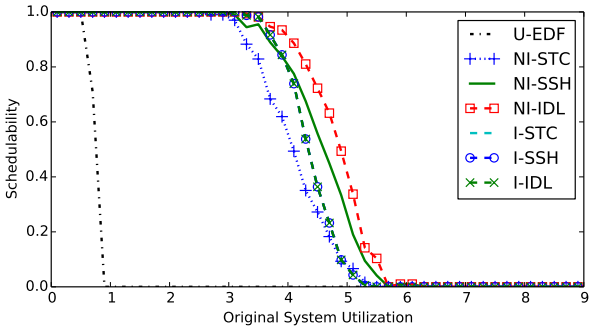


B-Heavy, Long, Heavy, Heavy, Light, Large

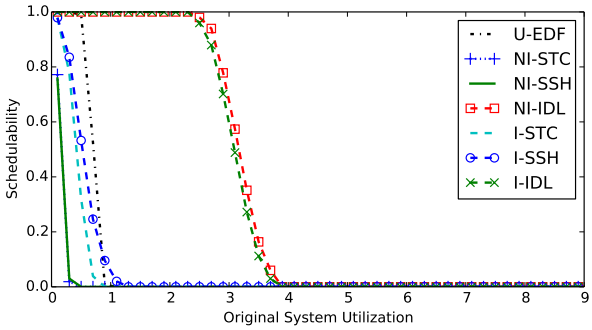


C-Heavy, Long, Mod., Mod., Heavy, Small

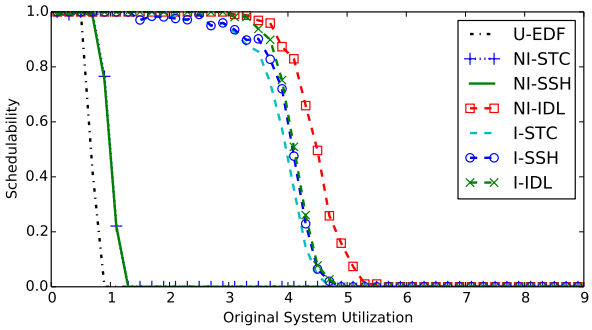
APPENDIX C: SCHEDULABILITY GRAPHS FOR THE STUDY DESCRIBED IN SECTION 4.2



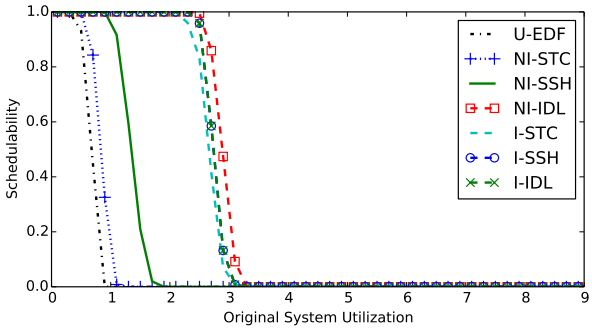
BC-Mod., Long, Heavy, Heavy, Small, Light, Light



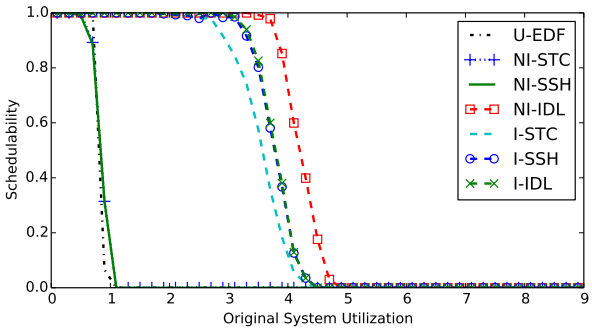
C-Heavy, Long, Light, Light, Small, Heavy, Heavy



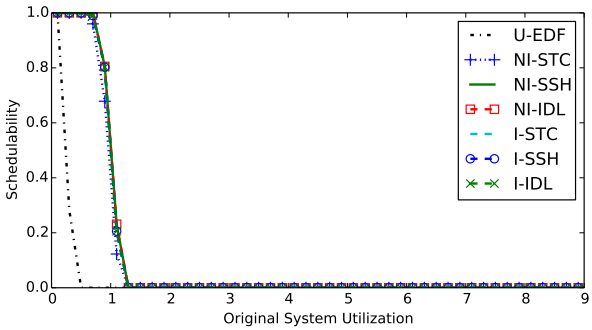
All-Mod., Long, Heavy, Heavy, Small, Heavy, Heavy



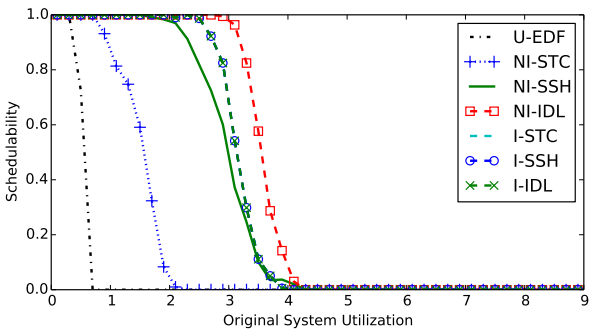
AB-Mod., Long, Light, Light, Mod., Heavy, Heavy



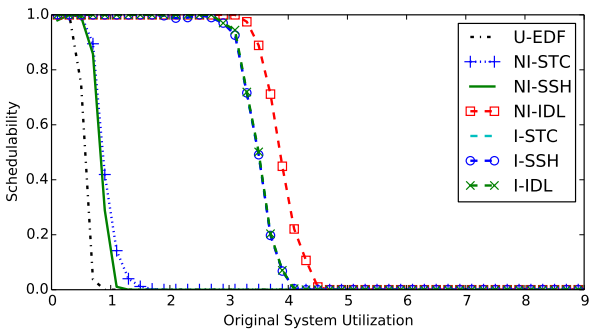
AB-Mod., Short, Heavy, Light, Small, Heavy, Heavy



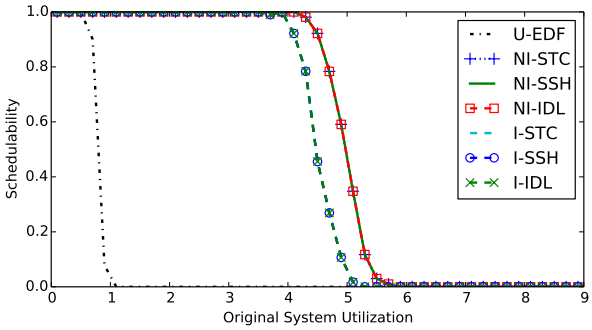
All-Mod., Short, Light, Heavy, Mod., Heavy, Heavy



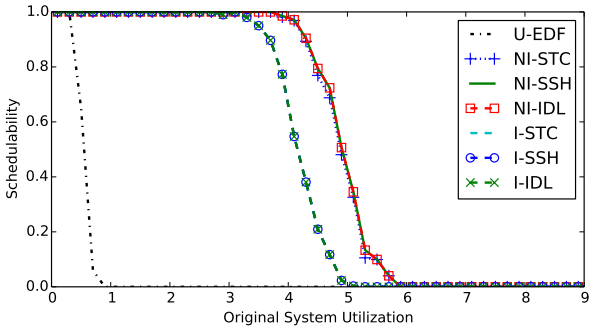
A-Heavy, Short, Heavy, Heavy, Small, Light, Heavy



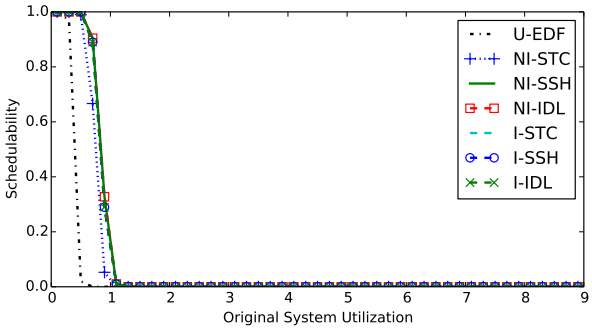
AB-Mod., Short, Heavy, Heavy, Small, Heavy, Light



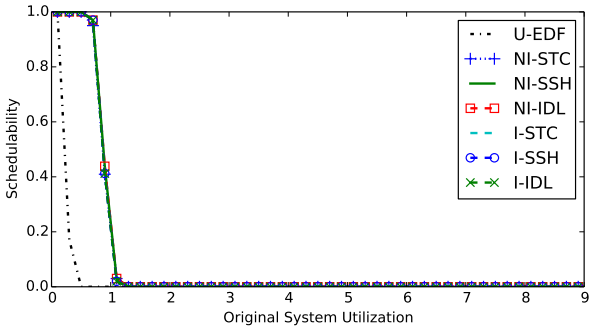
AC-Mod., Short, Heavy, Light, Large, Light, Light



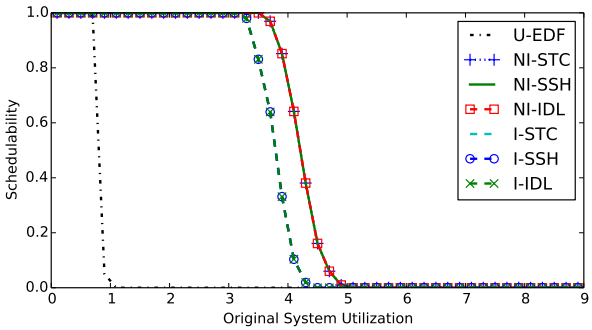
BC-Mod., Short, Heavy, Heavy, Large, Heavy, Heavy



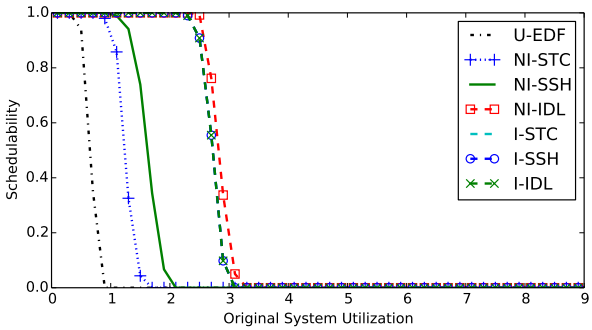
A-Heavy, Short, Light, Light, Mod., Heavy, Heavy



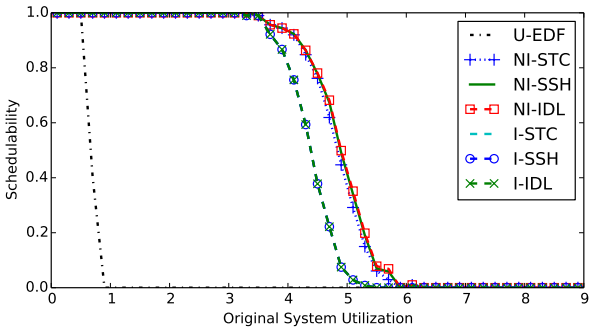
AC-Mod., Short, Light, Heavy, Mod., Heavy, Heavy



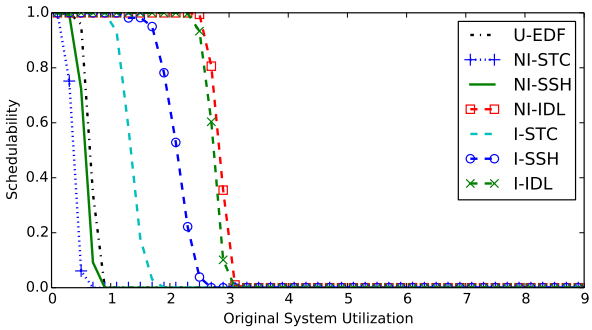
AB-Mod., Short, Heavy, Light, Mod., Heavy, Heavy



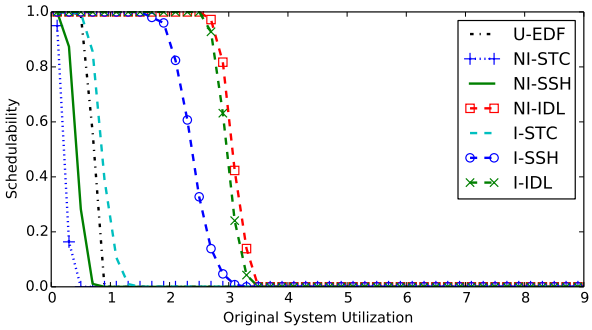
All-Mod., Long, Light, Light, Mod., Heavy, Light



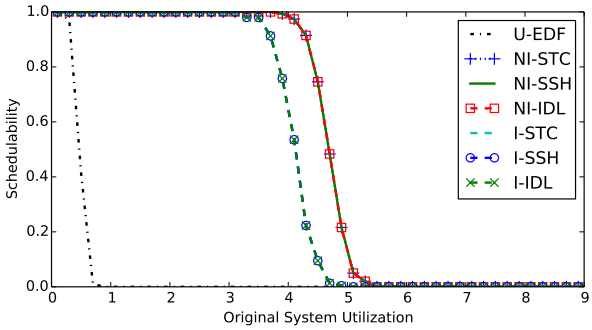
AC-Mod., Long, Heavy, Heavy, Large, Heavy, Light



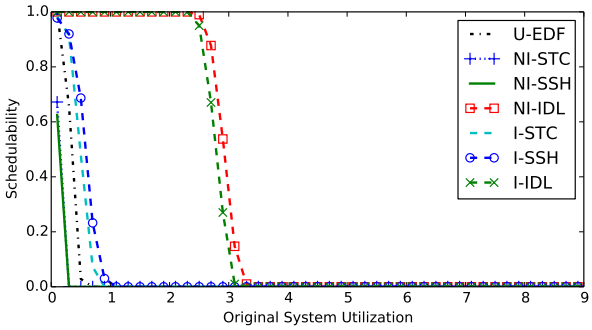
All-Mod., Long, Light, Light, Large, Heavy, Heavy



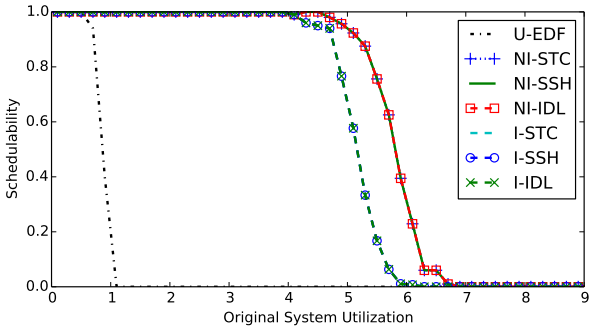
BC-Mod., Long, Light, Light, Small, Light, Heavy



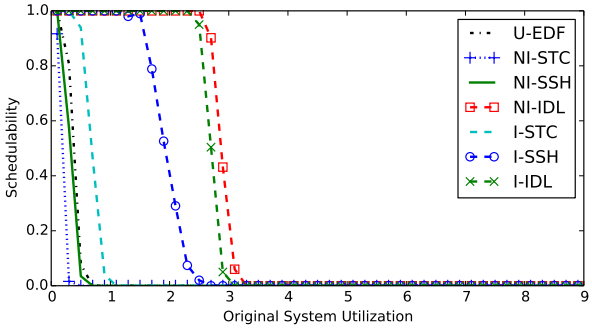
AC-Mod., Short, Heavy, Heavy, Mod., Light, Light



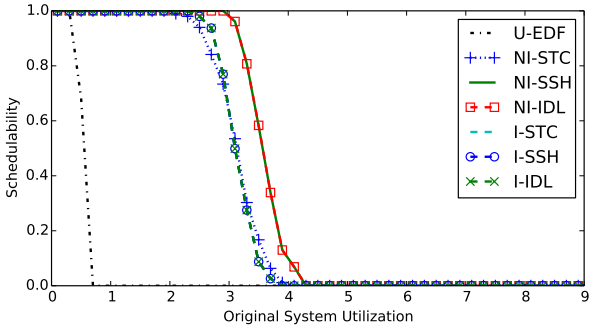
BC-Mod., Long, Light, Heavy, Small, Heavy, Light



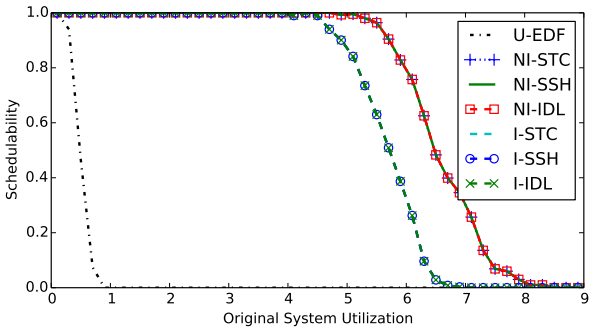
AC-Mod., Long, Heavy, Light, Mod., Heavy, Light



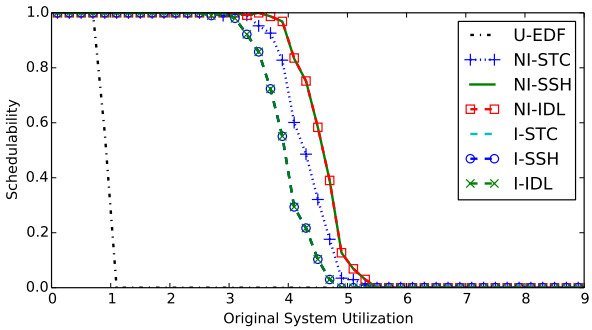
B-Heavy, Long, Light, Heavy, Small, Light, Heavy



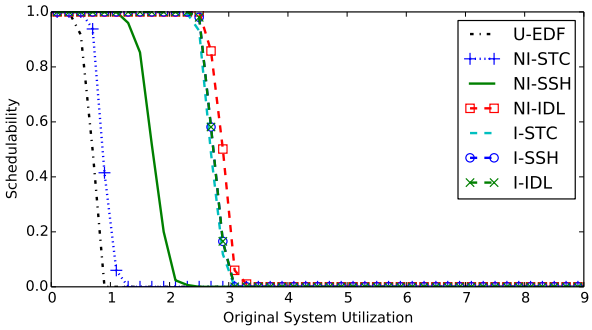
A-Heavy, Short, Heavy, Heavy, Large, Heavy, Heavy



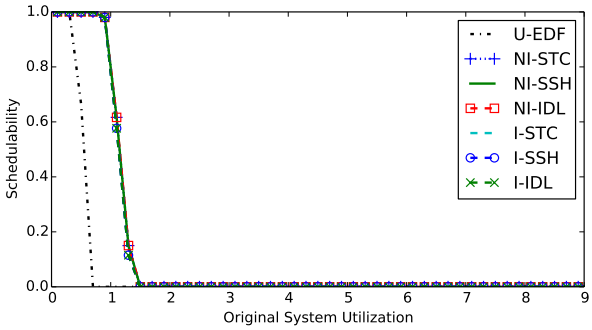
C-Heavy, Short, Heavy, Heavy, Large, Light, Light



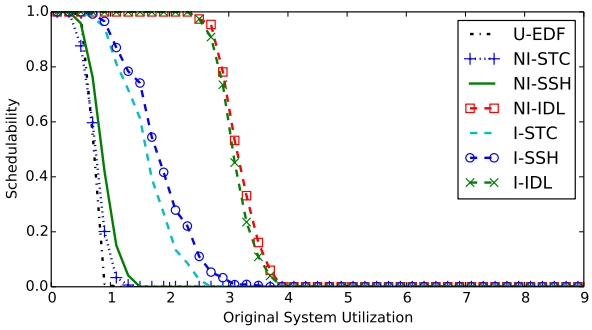
B-Heavy, Long, Heavy, Light, Large, Heavy, Heavy



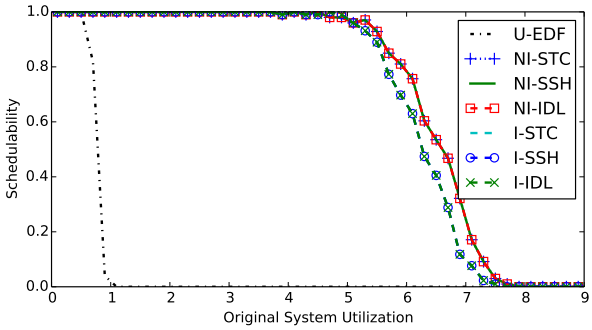
AB-Mod., Long, Light, Light, Large, Light, Light



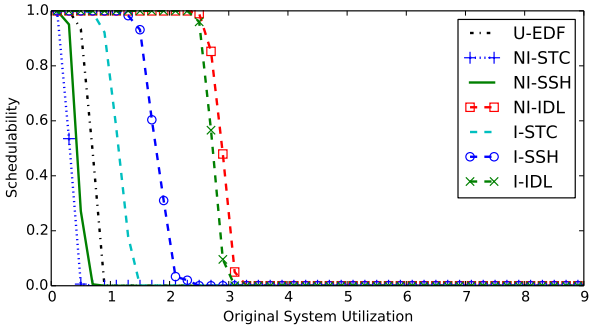
BC-Mod., Short, Light, Light, Mod., Light, Light



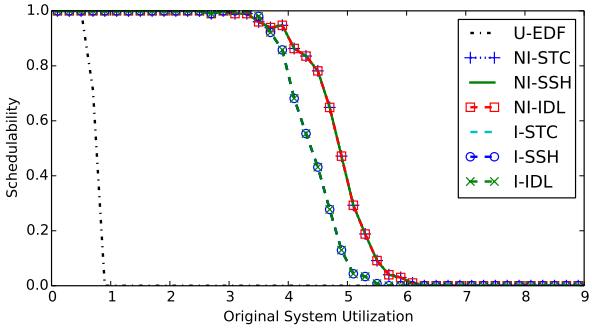
C-Heavy, Long, Light, Light, Large, Heavy, Light



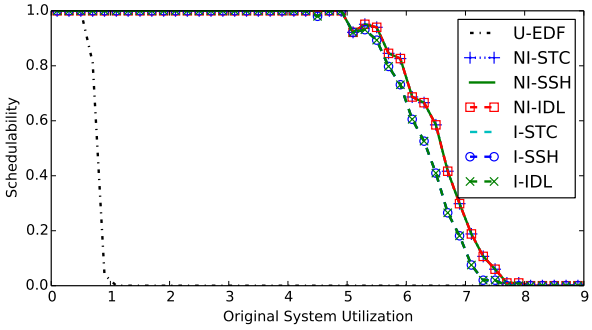
C-Heavy, Long, Heavy, Heavy, Large, Light, Light



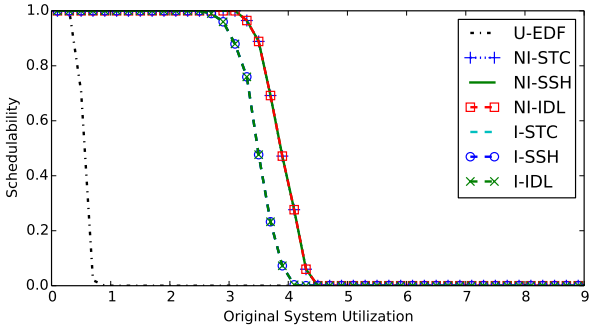
AB-Mod., Long, Light, Light, Large, Heavy, Heavy



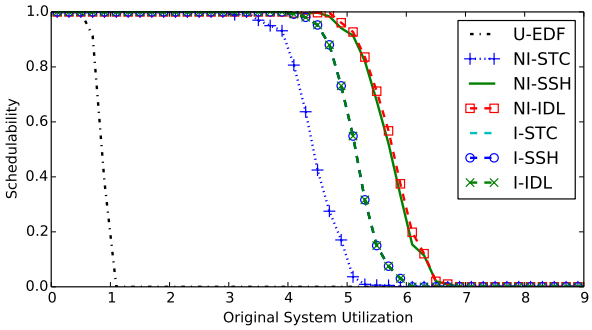
BC-Mod., Long, Heavy, Heavy, Mod., Heavy, Light



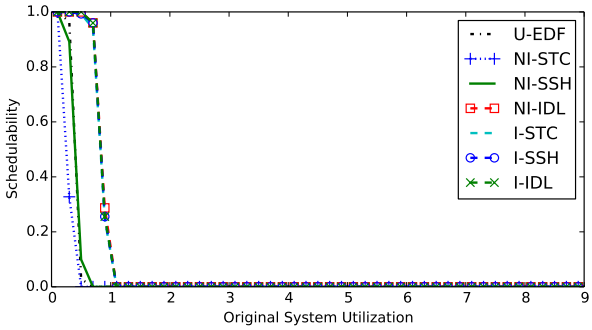
C-Heavy, Long, Heavy, Heavy, Mod., Light, Heavy



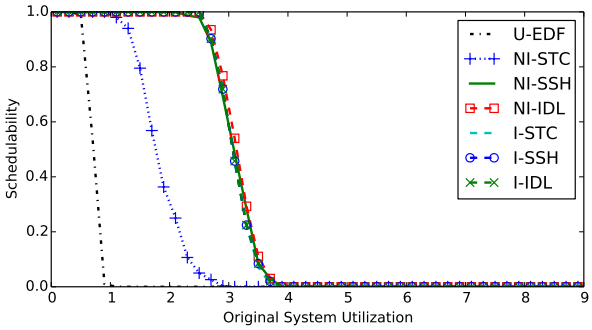
AB-Mod., Short, Heavy, Heavy, Mod., Light, Light



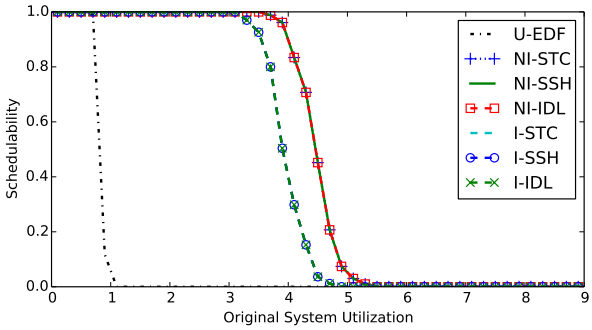
AC-Mod., Long, Heavy, Light, Large, Heavy, Heavy



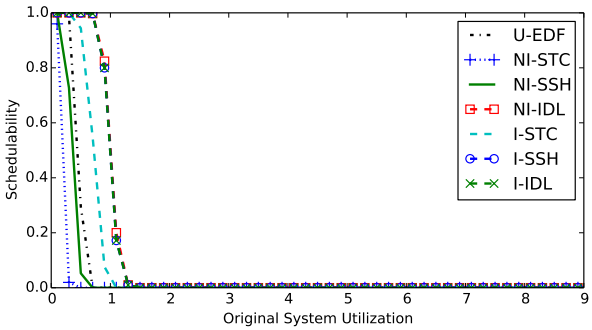
A-Heavy, Short, Light, Light, Large, Heavy, Heavy



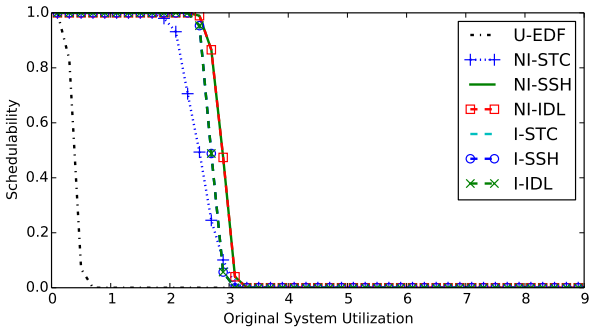
C-Heavy, Long, Light, Light, Large, Light, Light



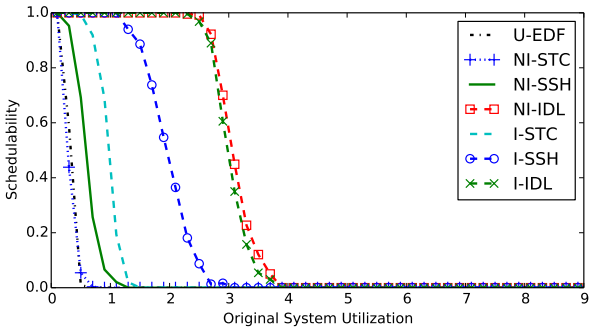
B-Heavy, Short, Heavy, Light, Mod., Heavy, Heavy



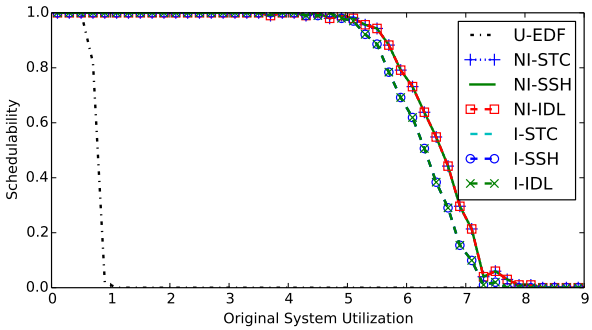
All-Mod., Short, Light, Light, Small, Light, Heavy



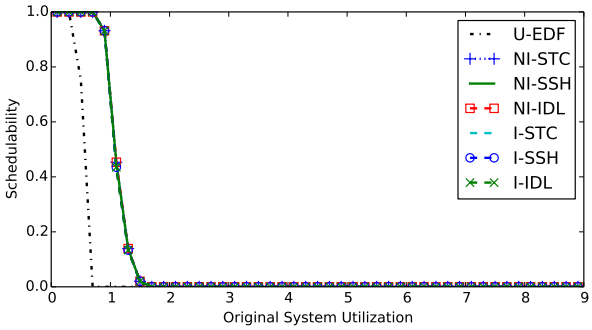
B-Heavy, Long, Light, Heavy, Mod., Light, Light



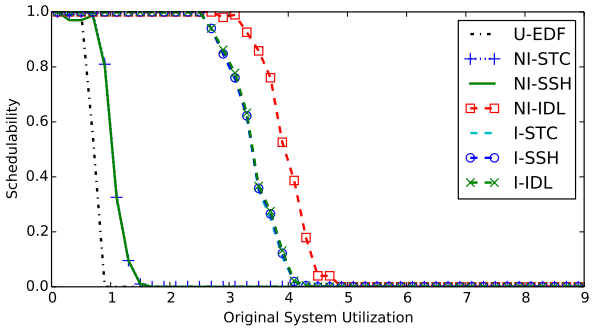
C-Heavy, Long, Light, Heavy, Small, Light, Heavy



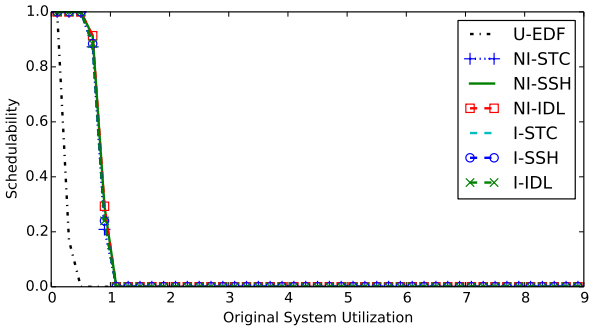
C-Heavy, Long, Heavy, Heavy, Mod., Light, Light



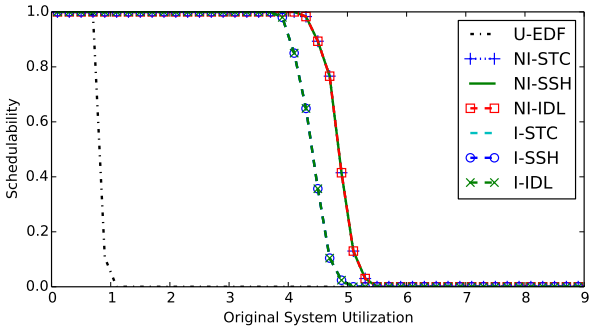
C-Heavy, Short, Light, Light, Mod., Light, Heavy



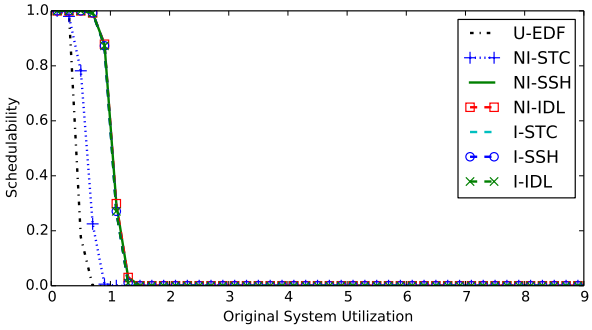
B-Heavy, Long, Heavy, Heavy, Small, Heavy, Heavy



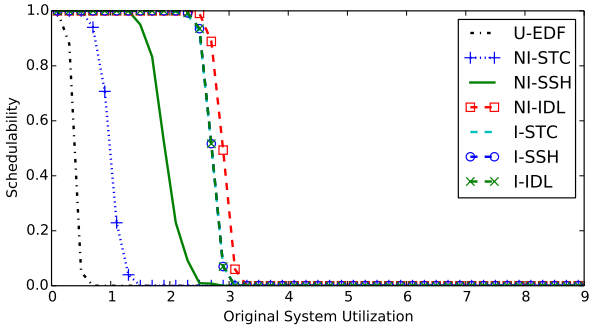
A-Heavy, Short, Light, Heavy, Mod., Heavy, Light



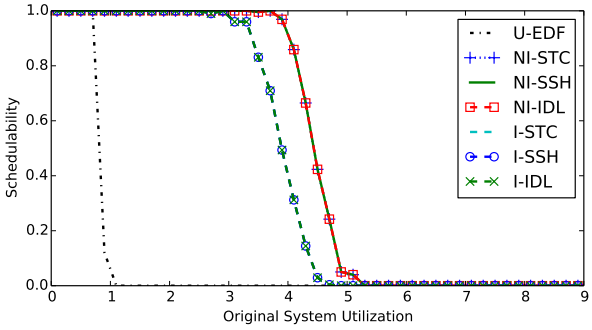
All-Mod., Short, Heavy, Light, Mod., Light, Light



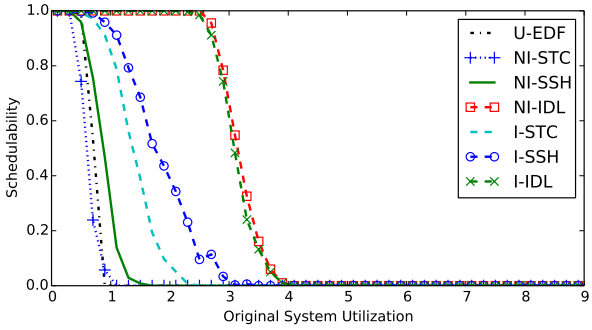
AB-Mod., Short, Light, Light, Large, Light, Heavy



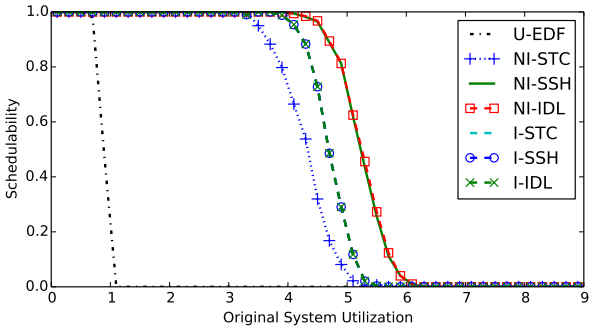
B-Heavy, Long, Light, Heavy, Large, Light, Light



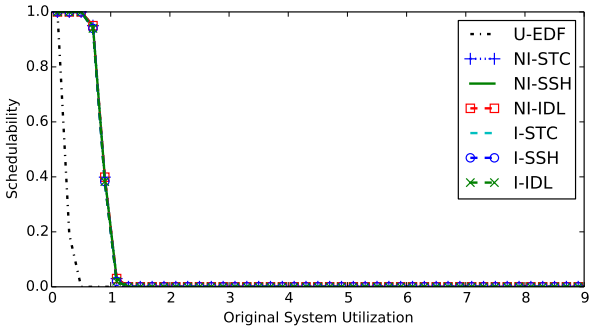
B-Heavy, Short, Heavy, Light, Large, Light, Heavy



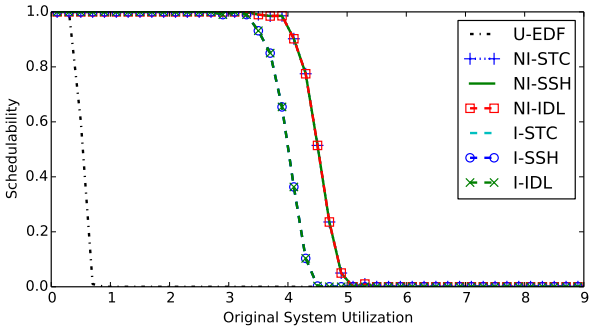
C-Heavy, Long, Light, Light, Large, Heavy, Heavy



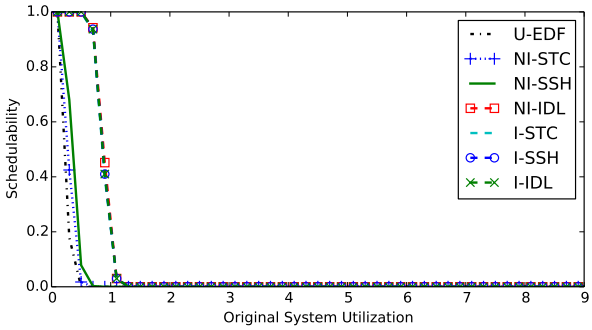
All-Mod., Long, Heavy, Light, Large, Heavy, Heavy



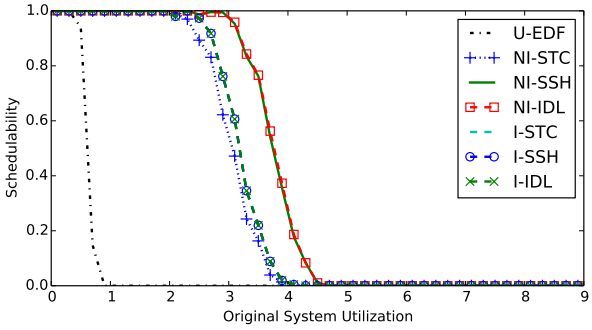
AC-Mod., Short, Light, Heavy, Mod., Heavy, Light



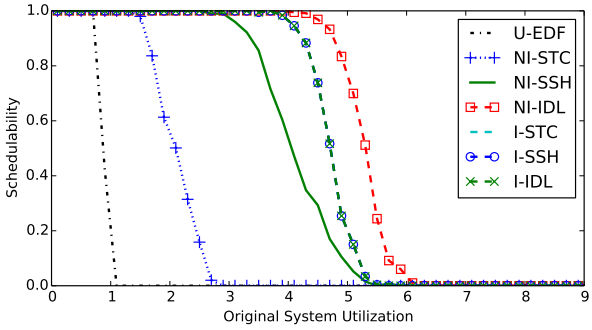
All-Mod., Short, Heavy, Heavy, Mod., Light, Heavy



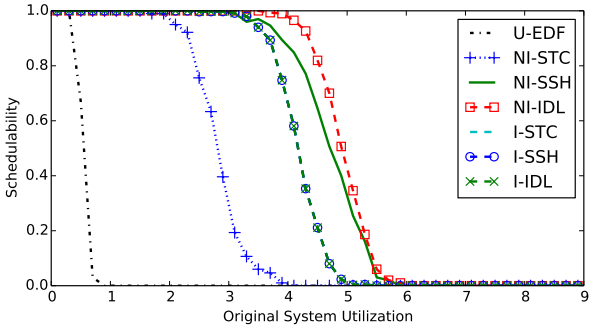
AC-Mod., Short, Light, Heavy, Small, Light, Light



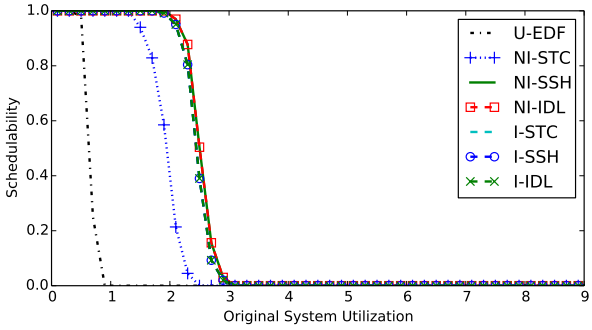
A-Heavy, Long, Heavy, Heavy, Large, Heavy, Heavy



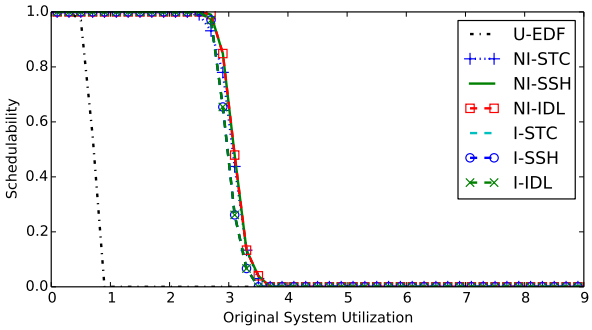
All-Mod., Long, Heavy, Light, Small, Light, Heavy



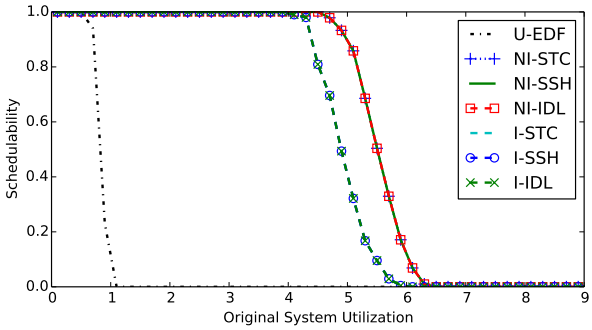
BC-Mod., Short, Heavy, Heavy, Small, Light, Heavy



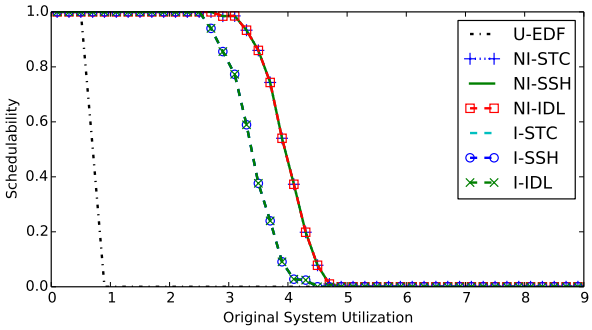
AC-Mod., Long, Light, Light, Mod., Light, Heavy



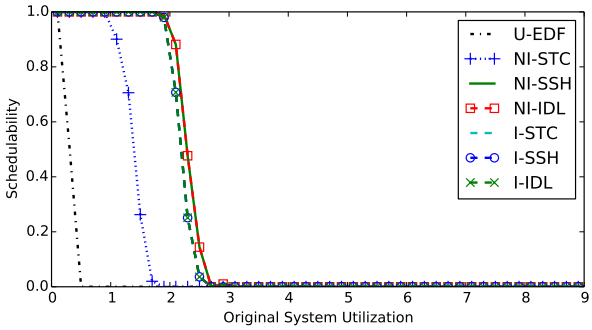
BC-Mod., Long, Light, Light, Mod., Light, Light



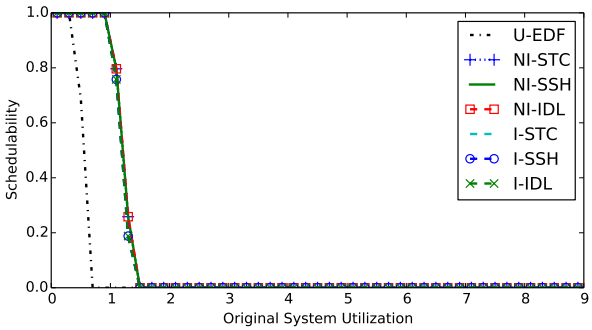
BC-Mod., Short, Heavy, Light, Mod., Light, Light



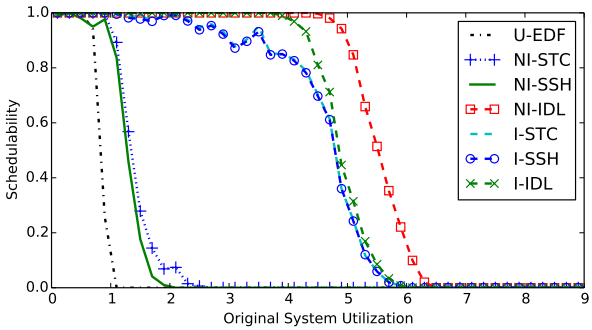
B-Heavy, Long, Heavy, Heavy, Mod., Heavy, Light



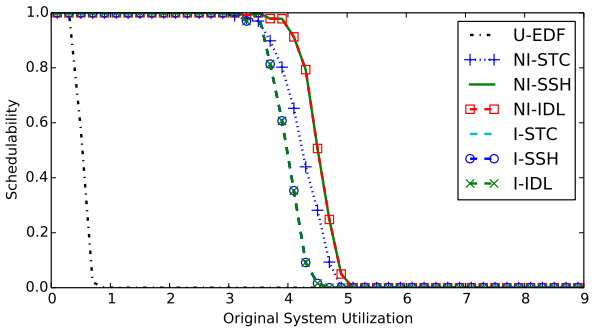
A-Heavy, Long, Light, Heavy, Mod., Light, Heavy



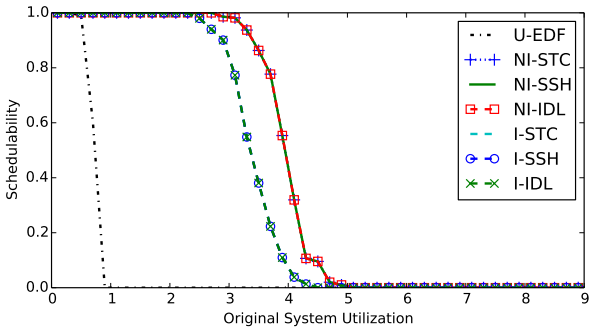
B-Heavy, Short, Light, Light, Mod., Light, Heavy



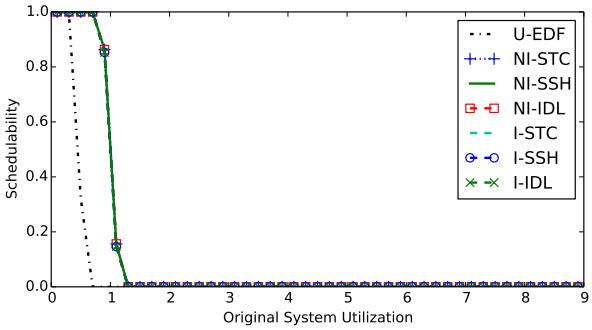
BC-Mod., Short, Heavy, Light, Small, Heavy, Light



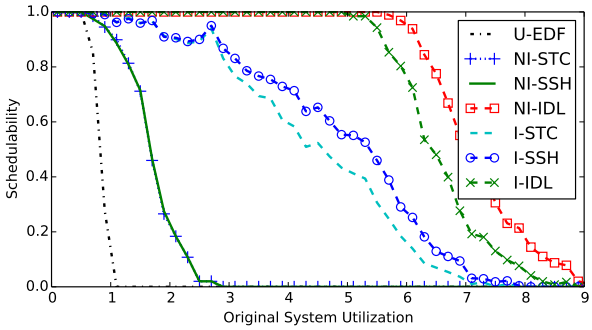
All-Mod., Short, Heavy, Heavy, Large, Heavy, Heavy



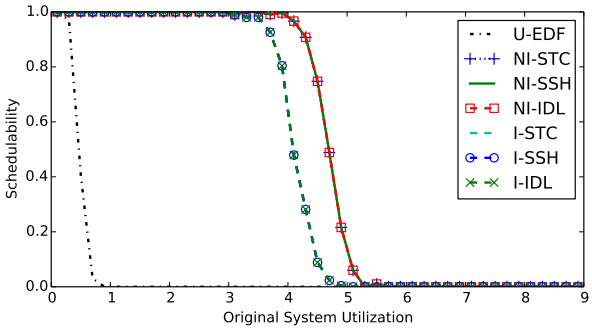
B-Heavy, Long, Heavy, Heavy, Large, Light, Light



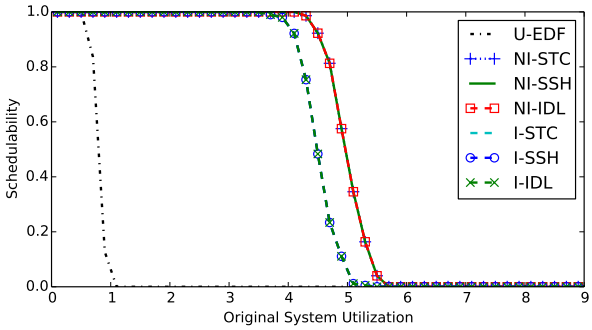
All-Mod., Short, Light, Light, Mod., Light, Light



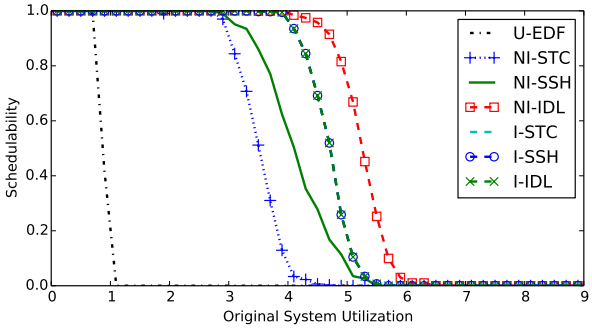
C-Heavy, Short, Heavy, Light, Small, Heavy, Heavy



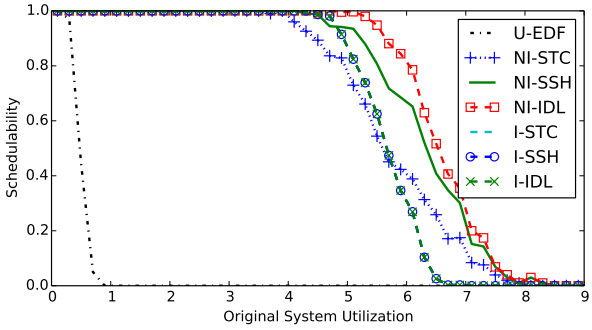
AC-Mod., Short, Heavy, Heavy, Large, Light, Heavy



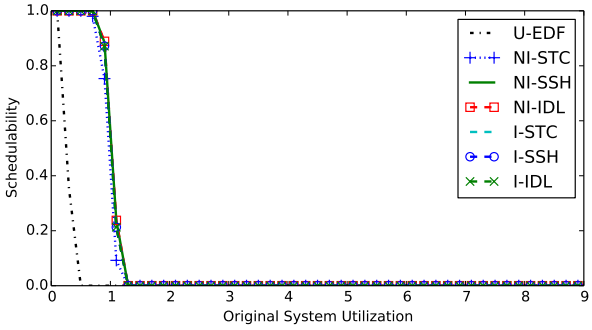
AC-Mod., Short, Heavy, Light, Mod., Heavy, Heavy



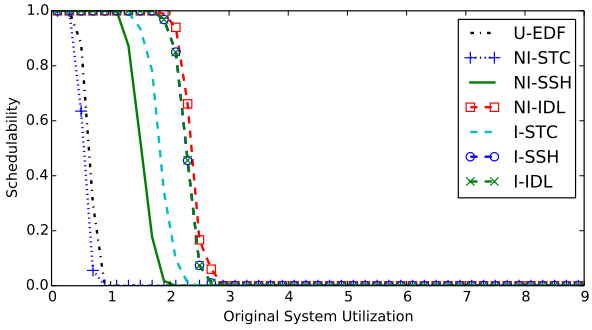
All-Mod., Long, Heavy, Light, Small, Light, Light



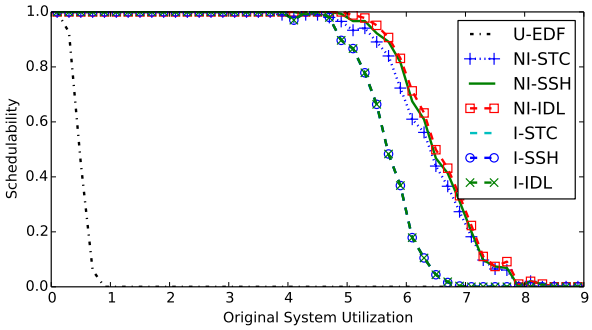
C-Heavy, Short, Heavy, Heavy, Small, Light, Light



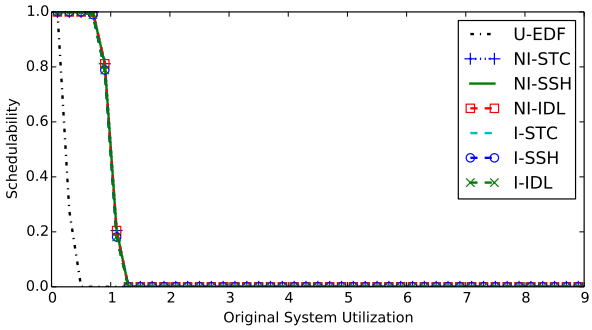
AB-Mod., Short, Light, Heavy, Mod., Heavy, Light



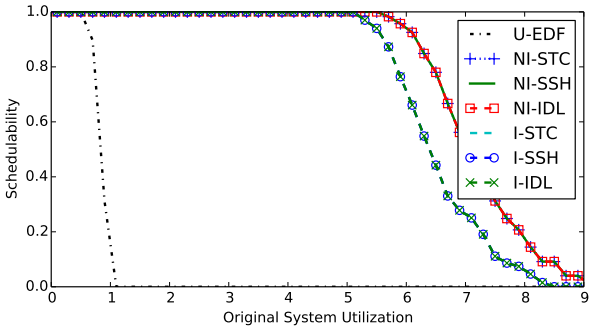
A-Heavy, Long, Light, Light, Large, Light, Heavy



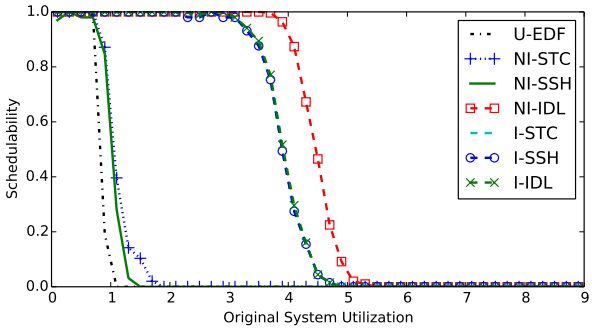
C-Heavy, Short, Heavy, Heavy, Large, Heavy, Heavy



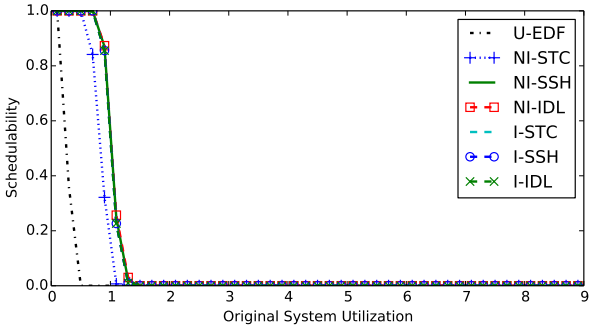
All-Mod., Short, Light, Heavy, Mod., Light, Light



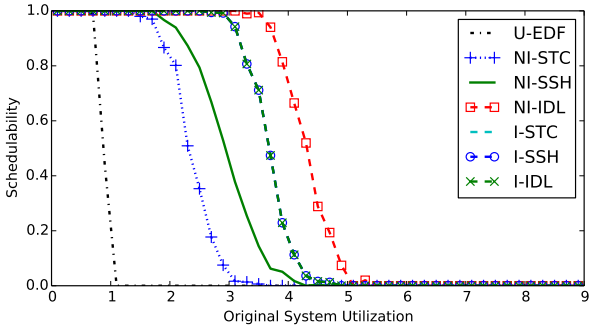
C-Heavy, Short, Heavy, Light, Mod., Heavy, Light



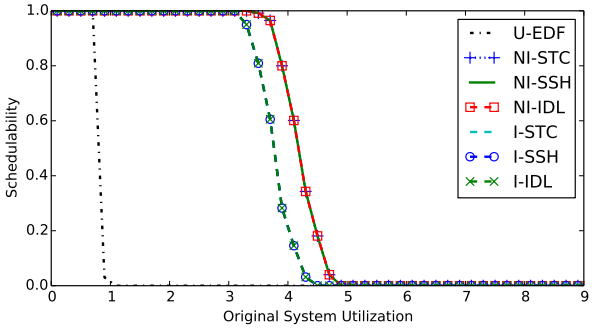
B-Heavy, Short, Heavy, Light, Small, Heavy, Light



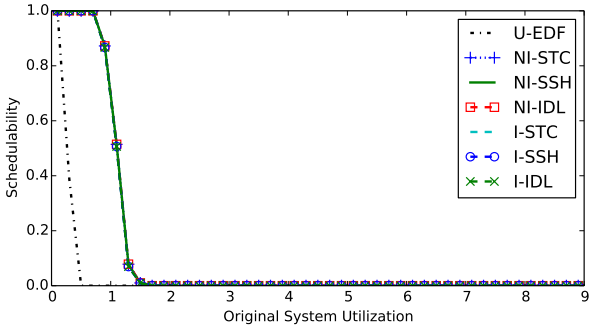
AB-Mod., Short, Light, Heavy, Mod., Heavy, Heavy



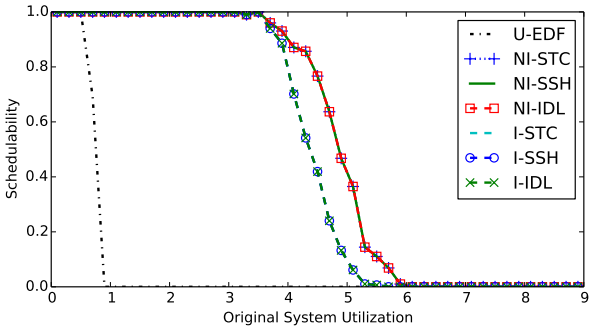
A-Heavy, Long, Heavy, Light, Small, Light, Light



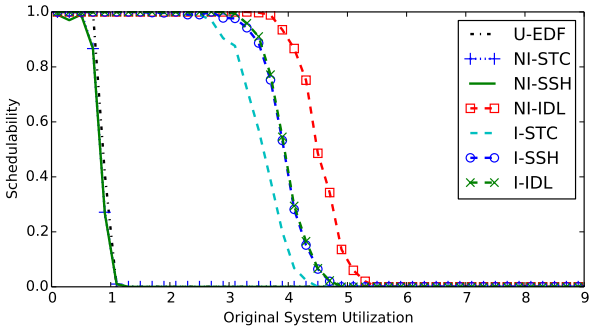
AB-Mod., Short, Heavy, Light, Large, Light, Heavy



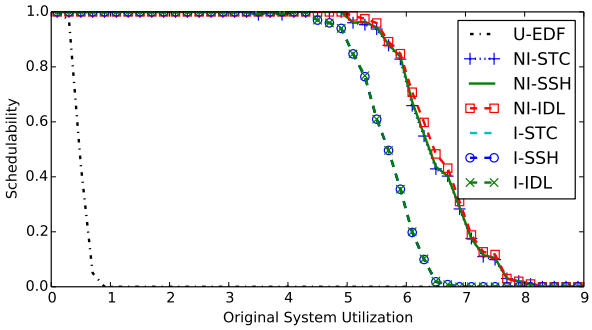
C-Heavy, Short, Light, Heavy, Large, Light, Light



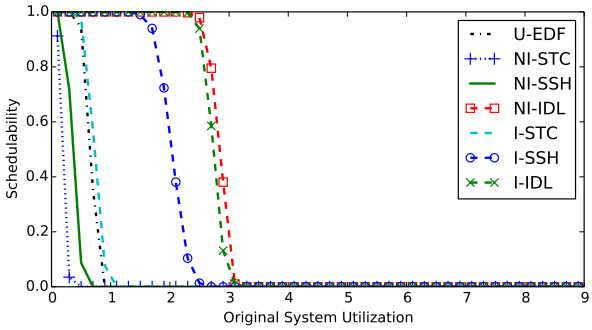
BC-Mod., Long, Heavy, Heavy, Large, Light, Light



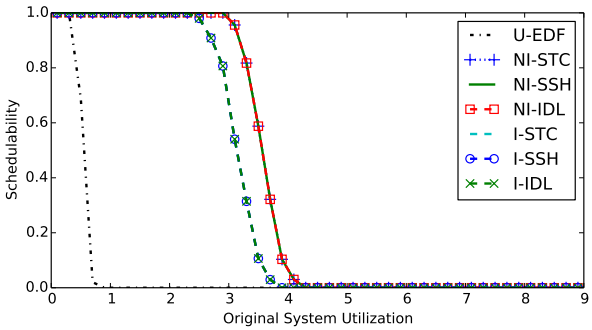
AB-Mod., Long, Heavy, Light, Small, Heavy, Heavy



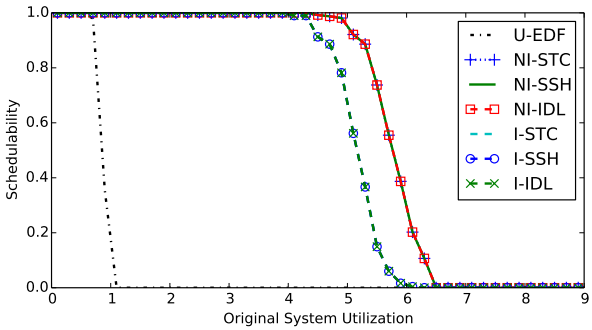
C-Heavy, Short, Heavy, Heavy, Large, Heavy, Light



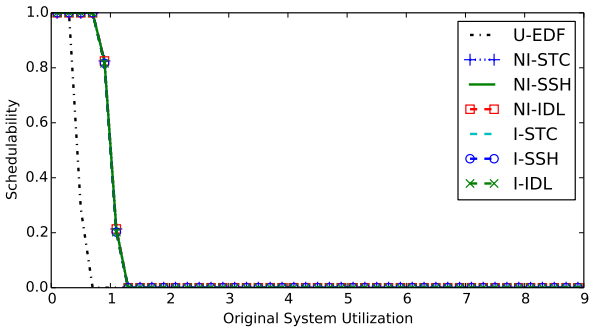
All-Mod., Long, Light, Light, Small, Light, Heavy



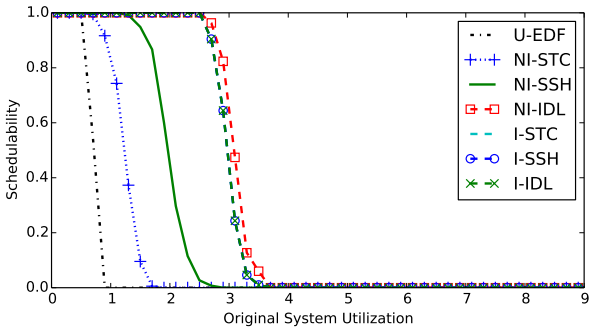
A-Heavy, Short, Heavy, Heavy, Large, Light, Heavy



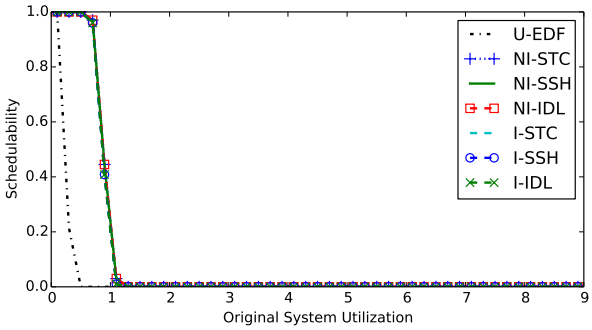
AC-Mod., Long, Heavy, Light, Large, Light, Heavy



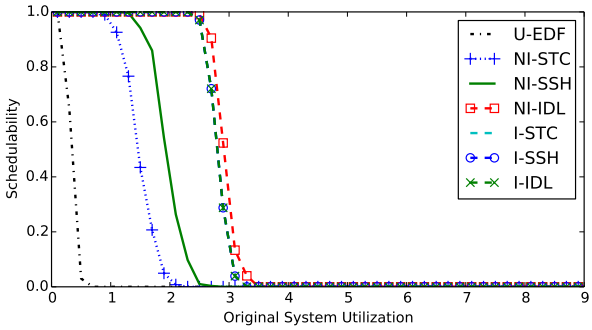
All-Mod., Short, Light, Light, Mod., Light, Heavy



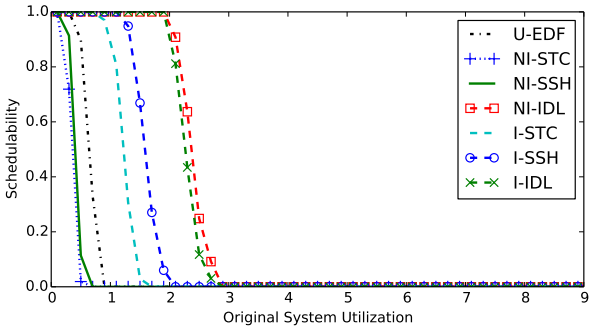
BC-Mod., Long, Light, Light, Mod., Heavy, Heavy



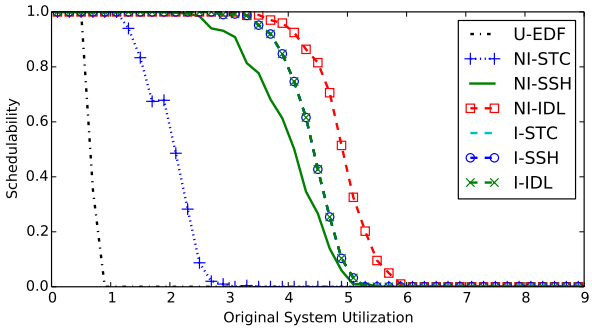
AC-Mod., Short, Light, Heavy, Mod., Light, Heavy



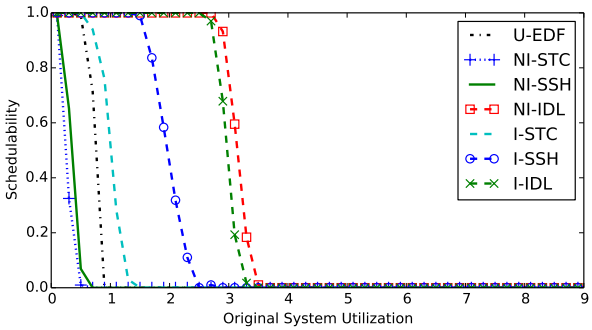
BC-Mod., Long, Light, Heavy, Mod., Heavy, Light



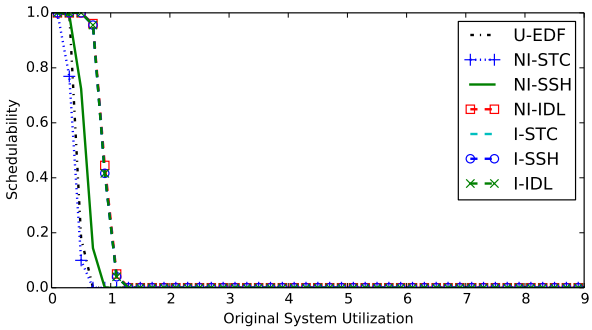
A-Heavy, Long, Light, Light, Large, Heavy, Light



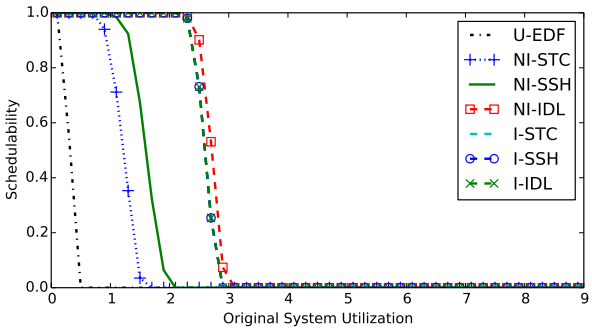
AC-Mod., Long, Heavy, Heavy, Small, Light, Heavy



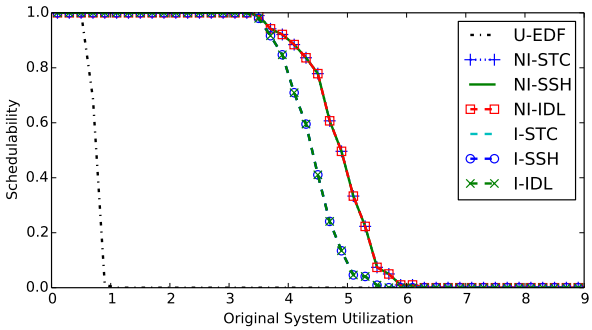
B-Heavy, Long, Light, Light, Small, Light, Light



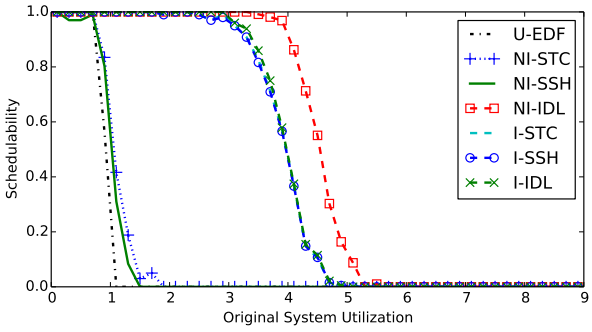
AC-Mod., Short, Light, Light, Large, Heavy, Heavy



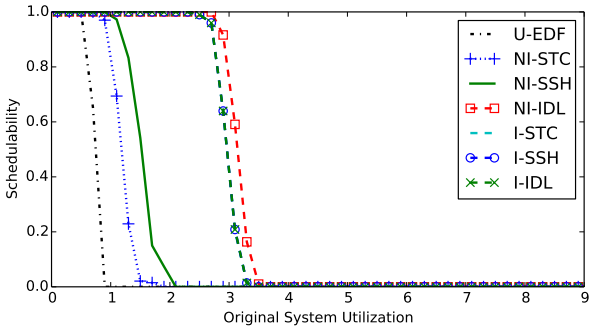
All-Mod., Long, Light, Heavy, Mod., Heavy, Light



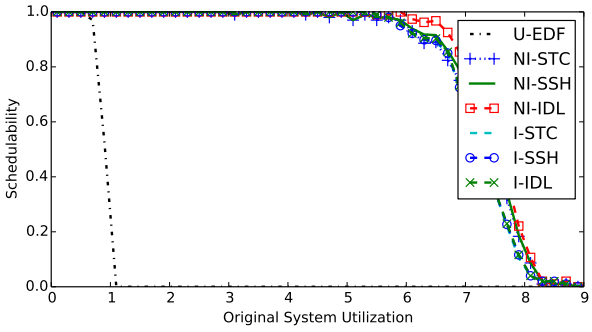
BC-Mod., Long, Heavy, Heavy, Large, Light, Heavy



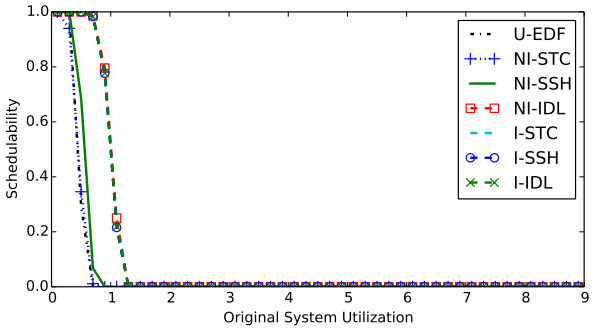
B-Heavy, Long, Heavy, Light, Small, Heavy, Light



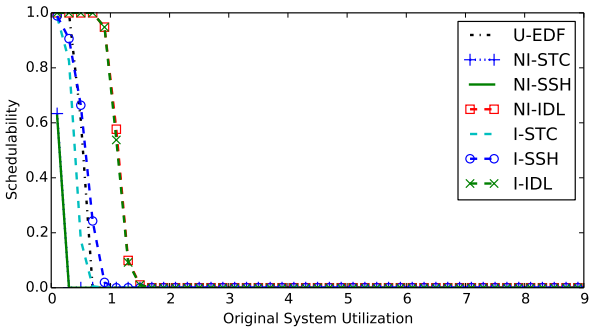
B-Heavy, Long, Light, Light, Mod., Heavy, Light



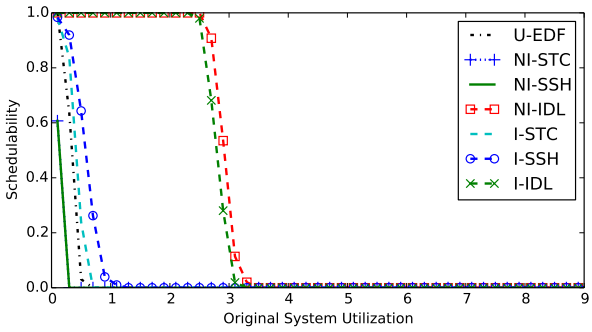
C-Heavy, Long, Heavy, Light, Large, Heavy, Light



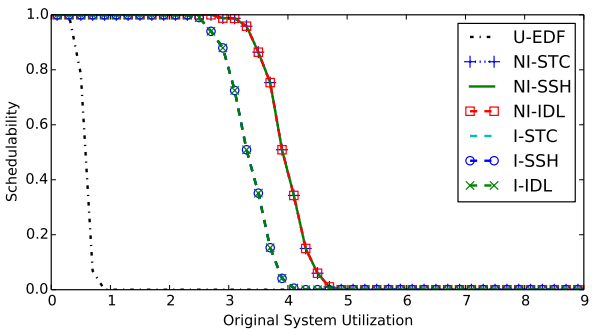
All-Mod., Short, Light, Light, Large, Heavy, Light



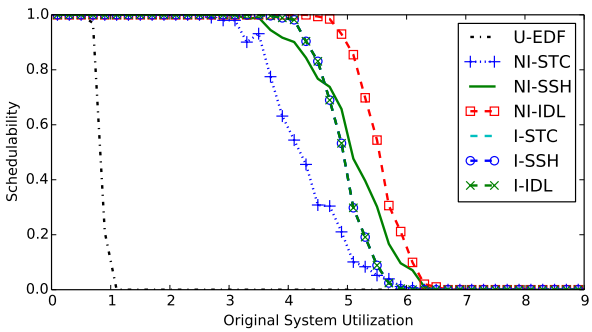
BC-Mod., Short, Light, Light, Small, Heavy, Heavy



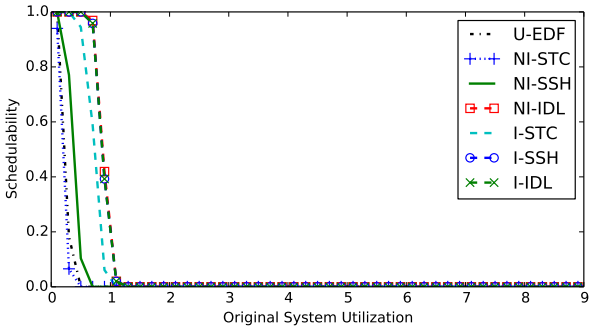
BC-Mod., Long, Light, Heavy, Small, Heavy, Heavy



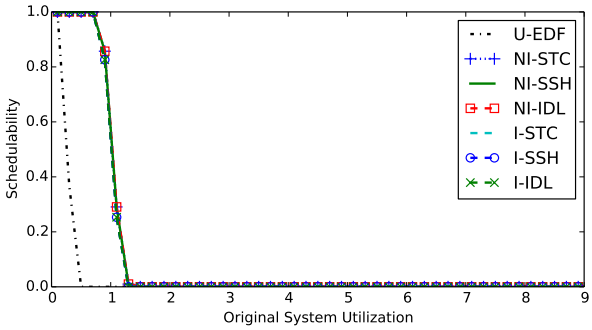
B-Heavy, Short, Heavy, Heavy, Large, Light, Heavy



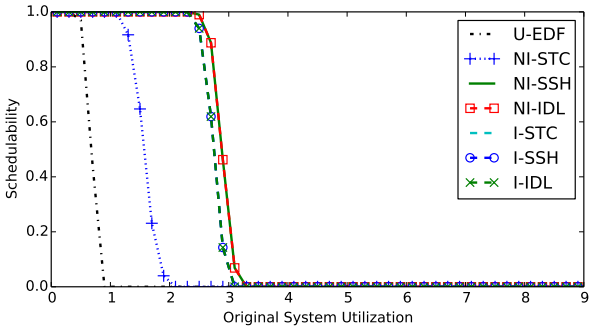
BC-Mod., Short, Heavy, Light, Small, Light, Light



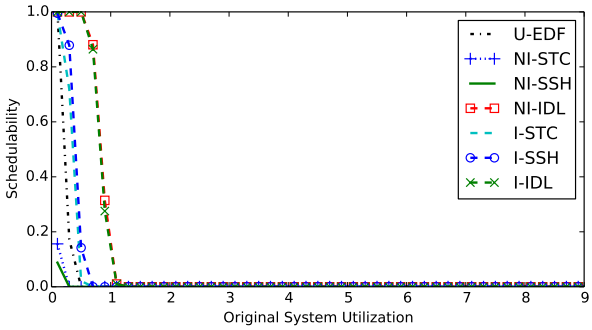
AC-Mod., Short, Light, Heavy, Small, Light, Heavy



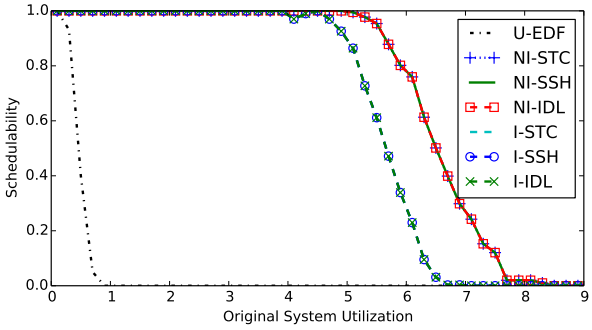
AB-Mod., Short, Light, Heavy, Mod., Light, Light



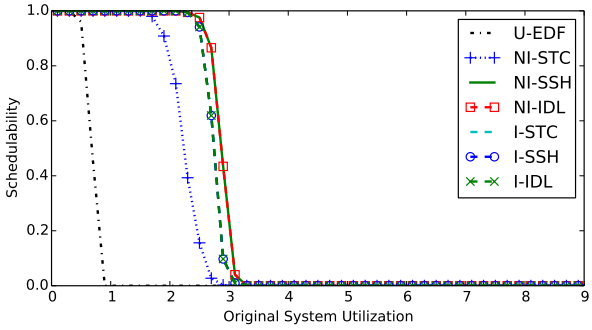
AB-Mod., Long, Light, Light, Mod., Light, Heavy



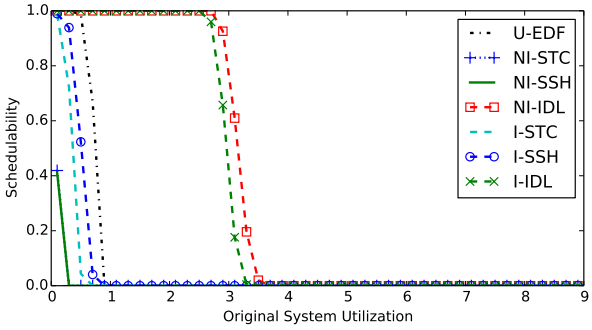
A-Heavy, Short, Light, Heavy, Small, Heavy, Light



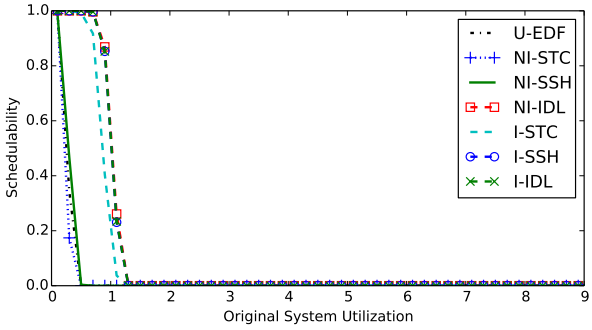
C-Heavy, Short, Heavy, Heavy, Mod., Heavy, Light



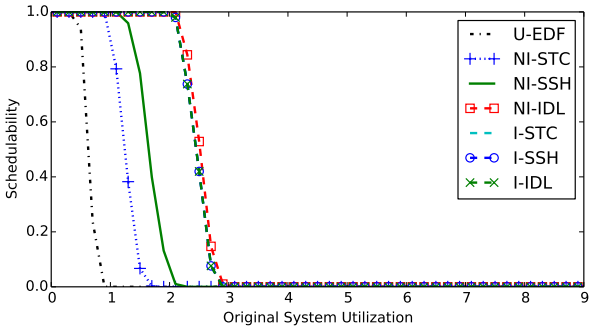
AB-Mod., Long, Light, Light, Mod., Light, Light



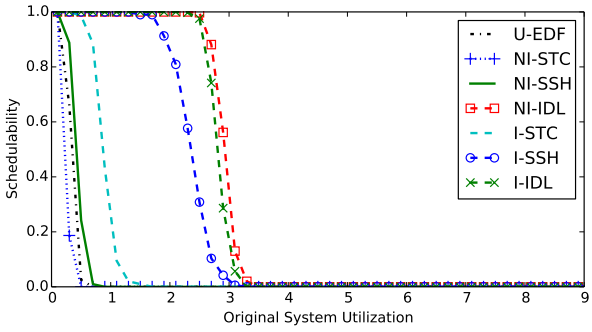
B-Heavy, Long, Light, Light, Small, Heavy, Heavy



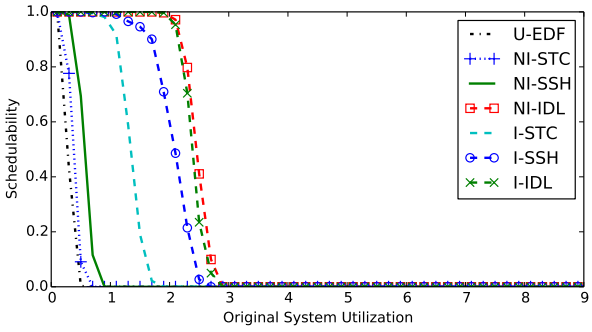
AB-Mod., Short, Light, Heavy, Small, Light, Light



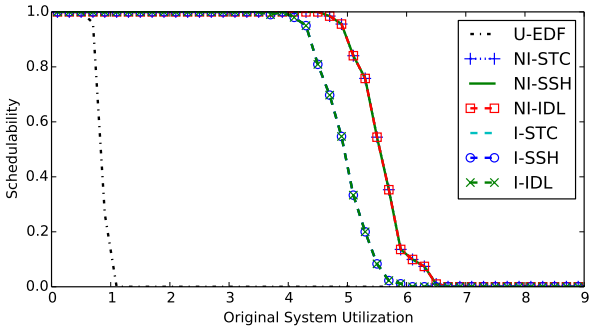
AC-Mod., Long, Light, Light, Mod., Heavy, Light



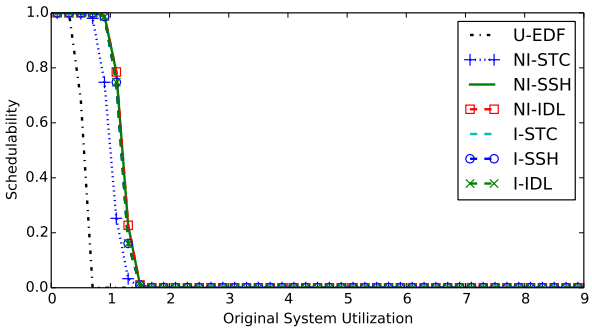
BC-Mod., Long, Light, Heavy, Small, Light, Heavy



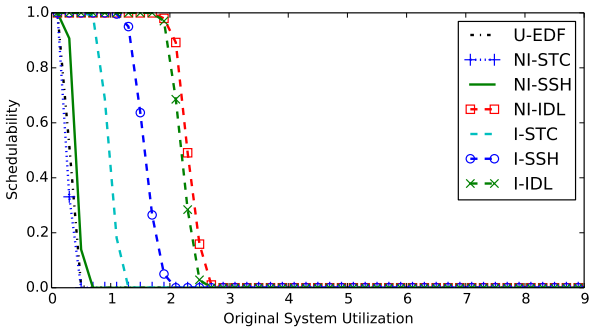
AC-Mod., Long, Light, Heavy, Large, Heavy, Heavy



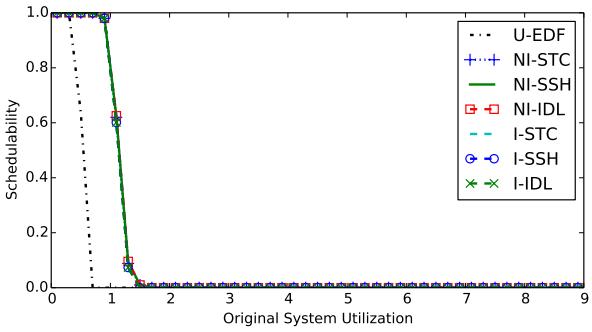
BC-Mod., Short, Heavy, Light, Large, Light, Light



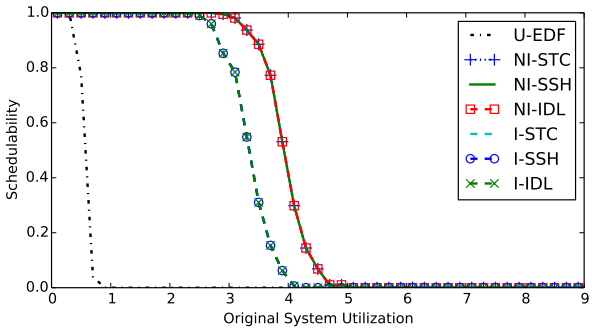
B-Heavy, Short, Light, Light, Large, Light, Light



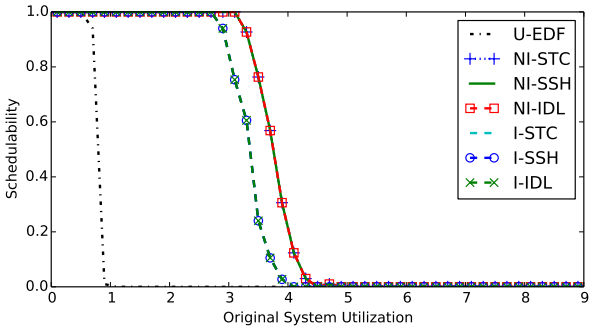
A-Heavy, Long, Light, Heavy, Large, Heavy, Heavy



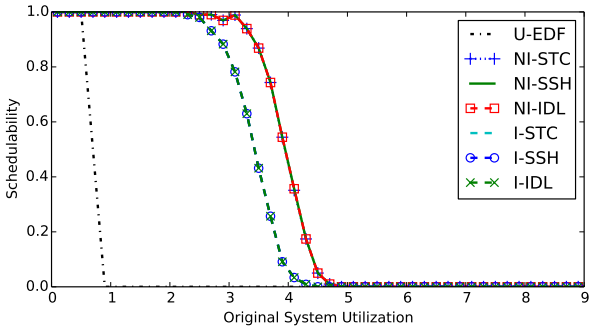
BC-Mod., Short, Light, Light, Mod., Heavy, Light



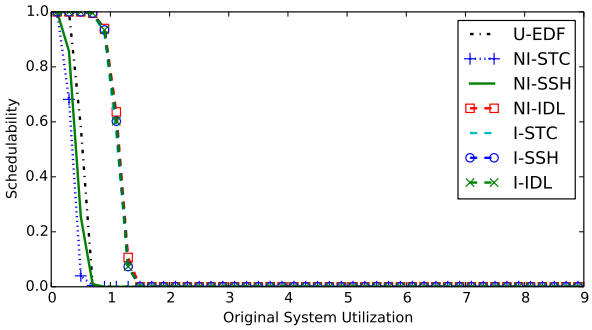
B-Heavy, Short, Heavy, Heavy, Mod., Light, Light



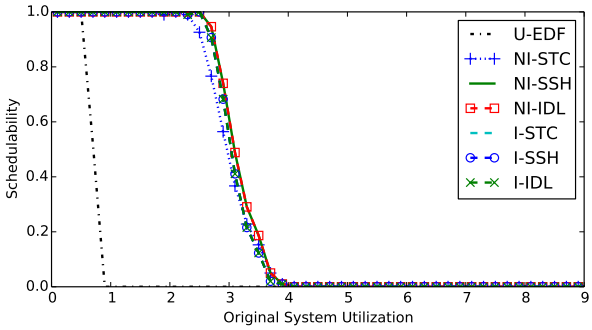
A-Heavy, Short, Heavy, Light, Mod., Light, Light



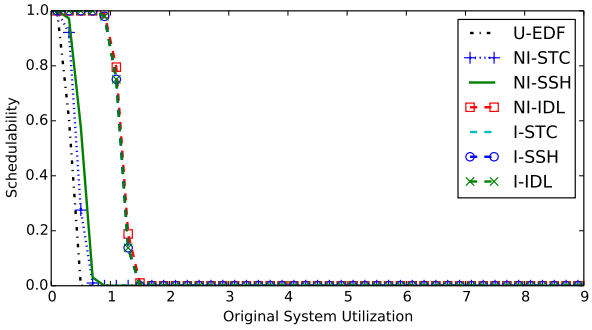
B-Heavy, Long, Heavy, Heavy, Large, Heavy, Light



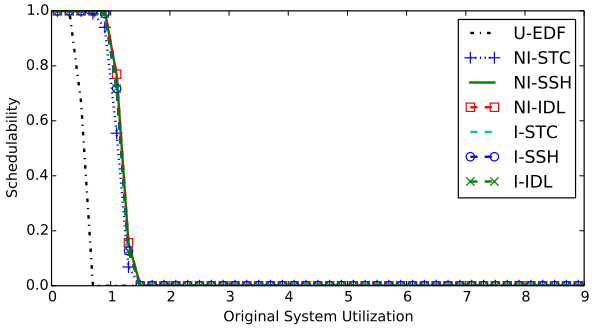
BC-Mod., Short, Light, Light, Small, Light, Light



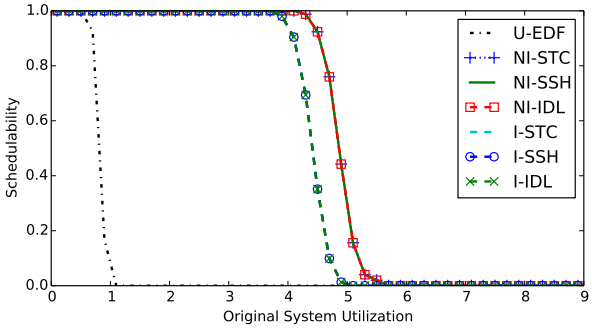
C-Heavy, Long, Light, Light, Mod., Light, Heavy



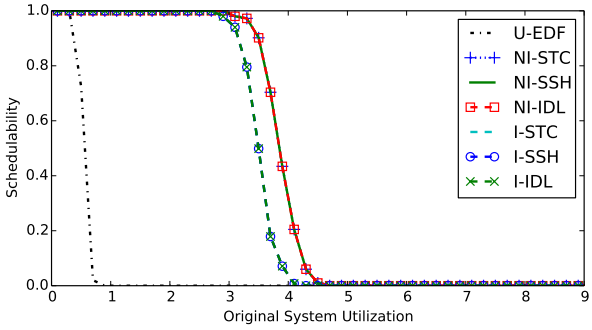
B-Heavy, Short, Light, Heavy, Large, Heavy, Light



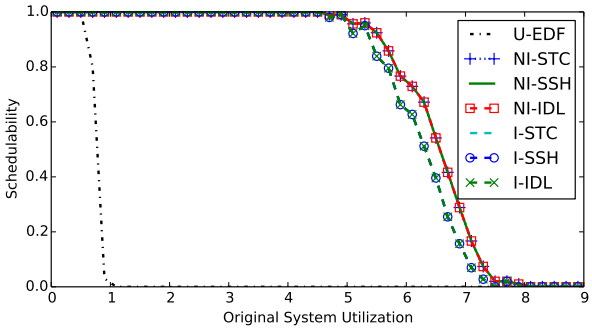
B-Heavy, Short, Light, Light, Mod., Heavy, Light



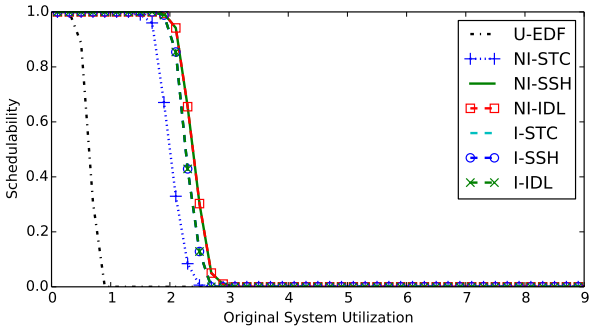
All-Mod., Short, Heavy, Light, Mod., Light, Heavy



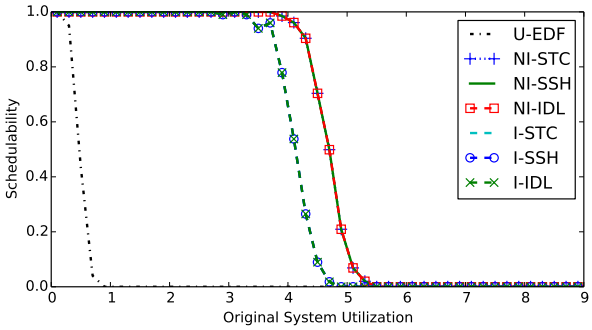
AB-Mod., Short, Heavy, Heavy, Mod., Heavy, Heavy



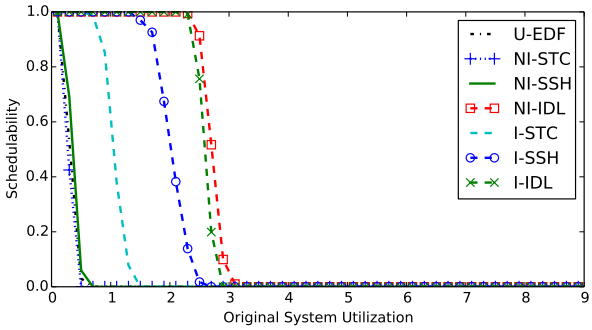
C-Heavy, Long, Heavy, Heavy, Mod., Heavy, Light



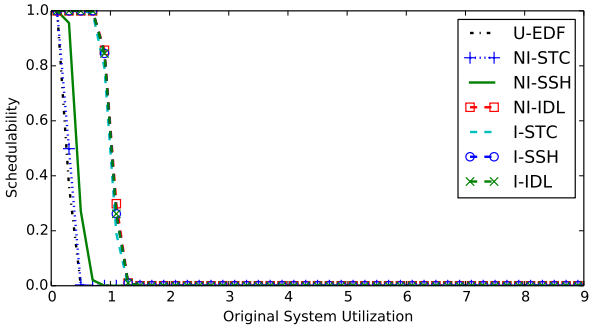
A-Heavy, Long, Light, Light, Mod., Light, Light



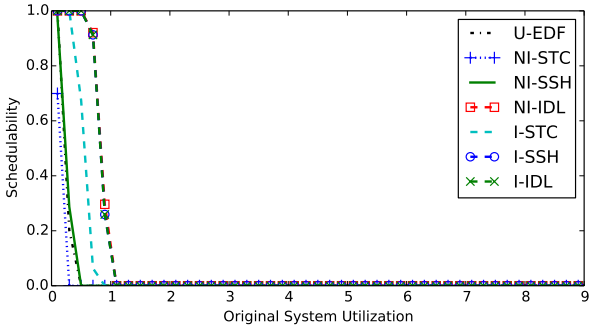
AC-Mod., Short, Heavy, Heavy, Large, Light, Light



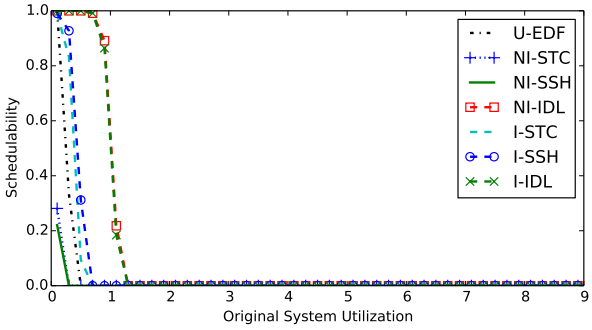
All-Mod., Long, Light, Heavy, Small, Light, Light



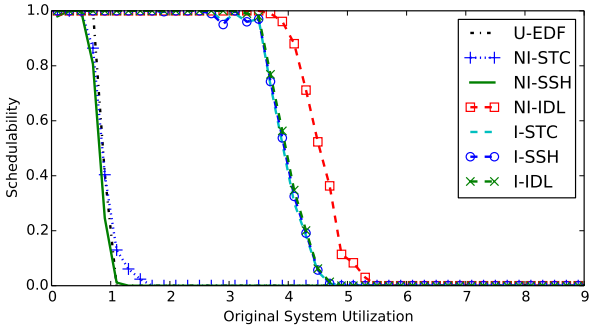
AB-Mod., Short, Light, Heavy, Large, Heavy, Heavy



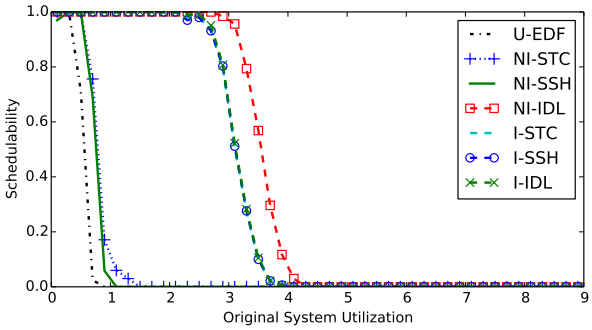
A-Heavy, Short, Light, Heavy, Small, Light, Heavy



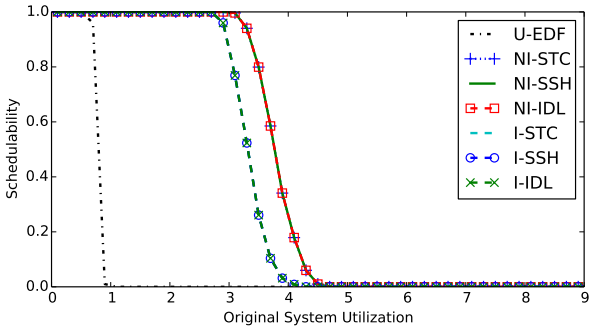
AB-Mod., Short, Light, Heavy, Small, Heavy, Light



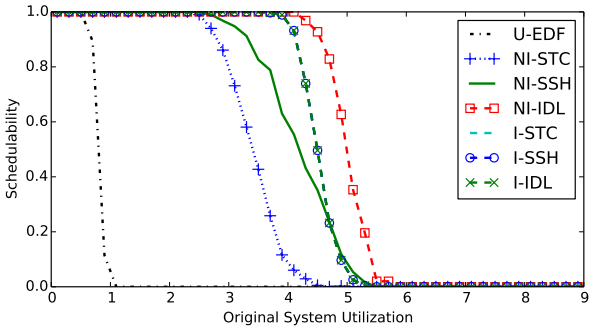
AB-Mod., Long, Heavy, Light, Small, Heavy, Light



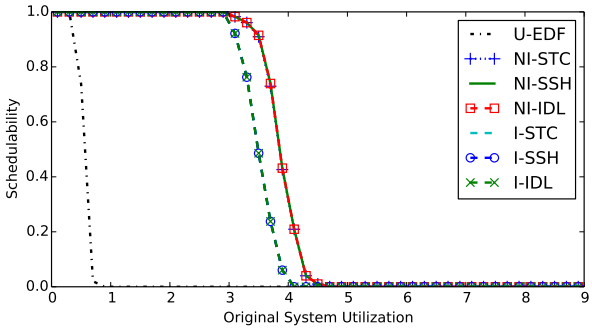
A-Heavy, Short, Heavy, Heavy, Small, Heavy, Light



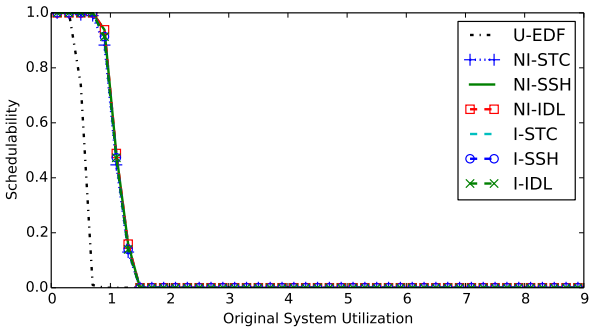
A-Heavy, Short, Heavy, Light, Large, Light, Light



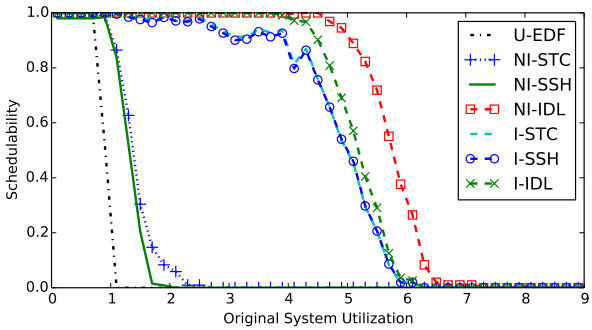
AC-Mod., Short, Heavy, Light, Small, Light, Light



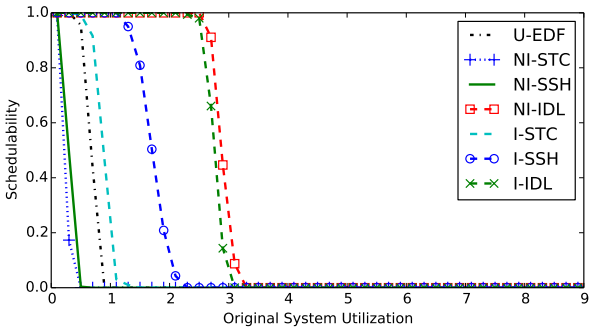
AB-Mod., Short, Heavy, Heavy, Large, Heavy, Light



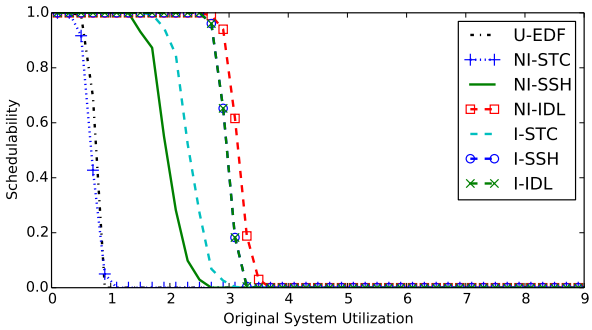
C-Heavy, Short, Light, Light, Large, Light, Heavy



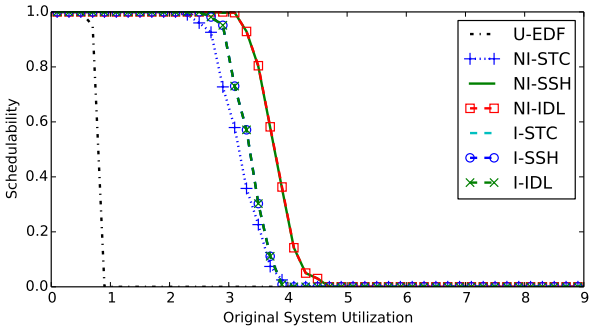
BC-Mod., Long, Heavy, Light, Small, Heavy, Light



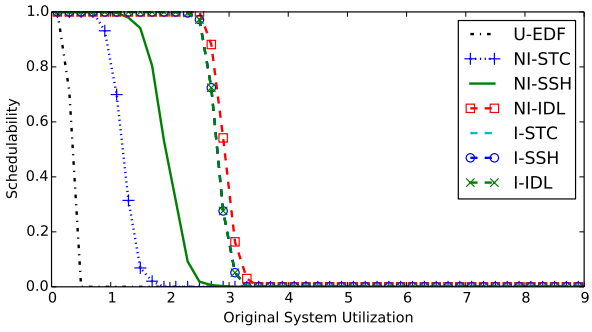
AB-Mod., Long, Light, Light, Small, Light, Light



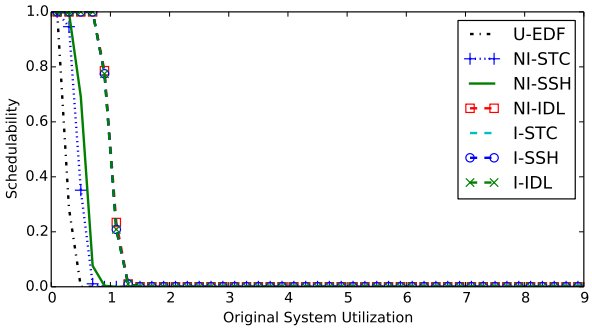
B-Heavy, Long, Light, Light, Large, Light, Heavy



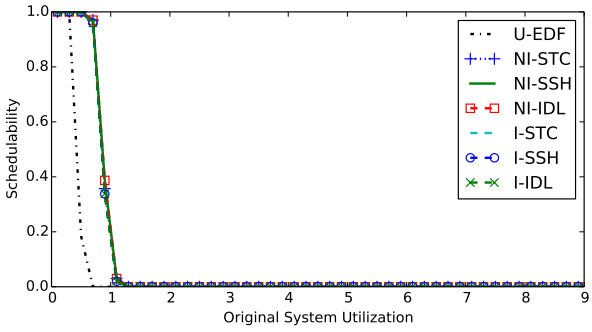
A-Heavy, Short, Heavy, Light, Large, Heavy, Heavy



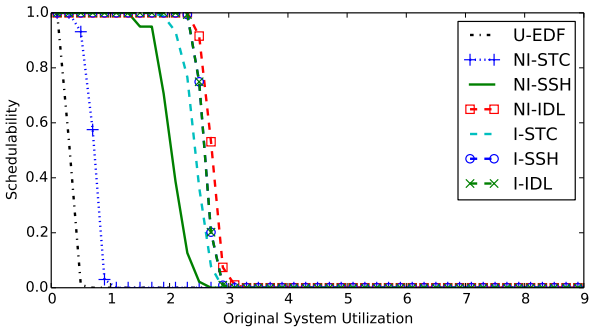
BC-Mod., Long, Light, Heavy, Mod., Heavy, Heavy



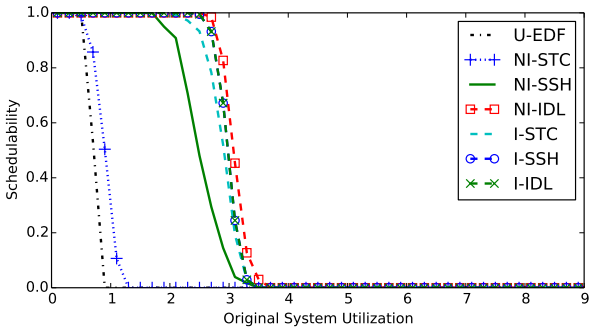
All-Mod., Short, Light, Heavy, Large, Heavy, Light



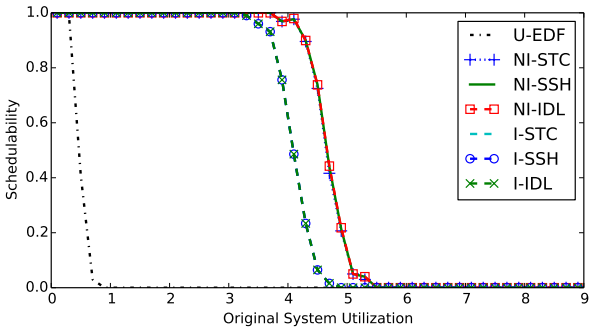
AC-Mod., Short, Light, Light, Large, Light, Light



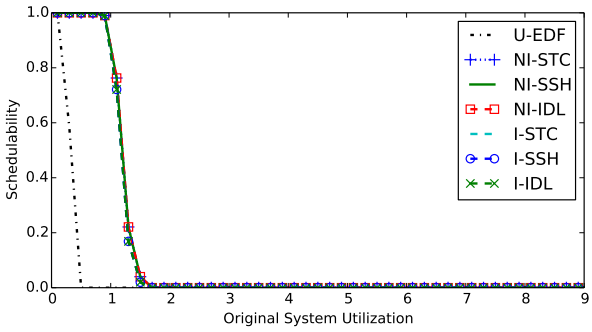
All-Mod., Long, Light, Heavy, Large, Light, Heavy



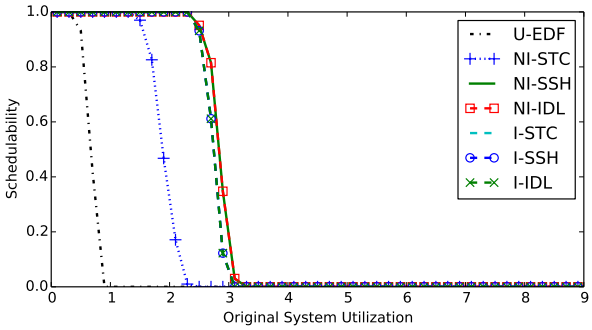
BC-Mod., Long, Light, Light, Large, Light, Heavy



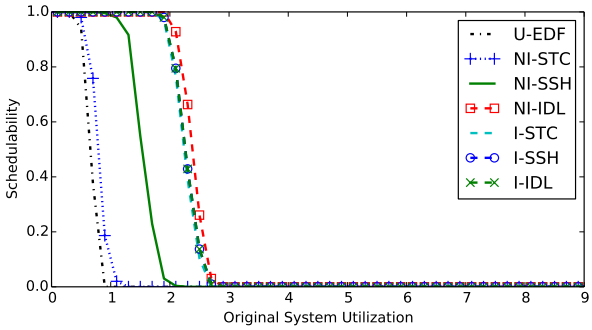
AC-Mod., Short, Heavy, Heavy, Large, Heavy, Light



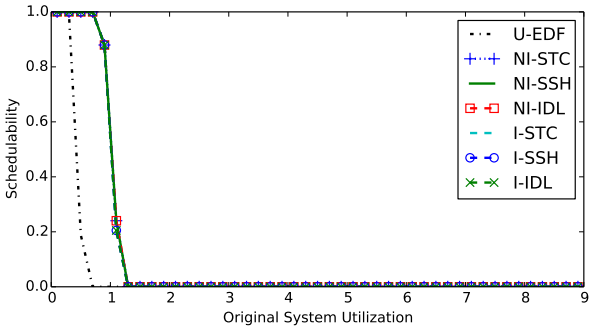
B-Heavy, Short, Light, Heavy, Mod., Light, Light



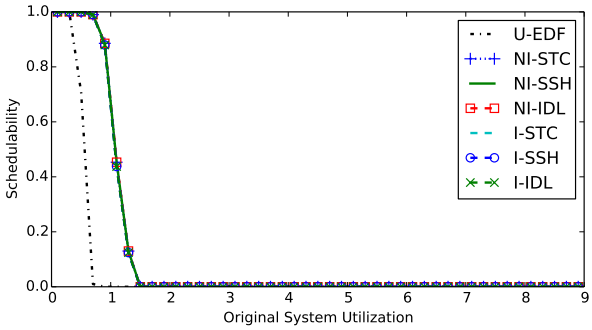
All-Mod., Long, Light, Light, Mod., Light, Heavy



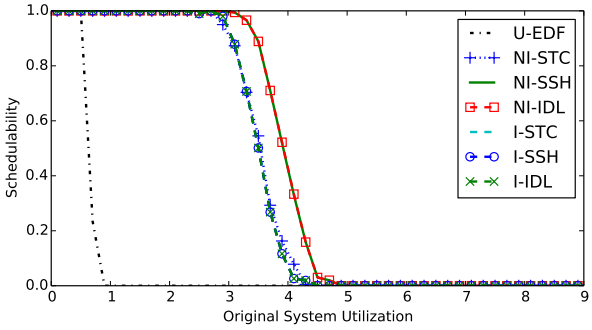
A-Heavy, Long, Light, Light, Large, Light, Light



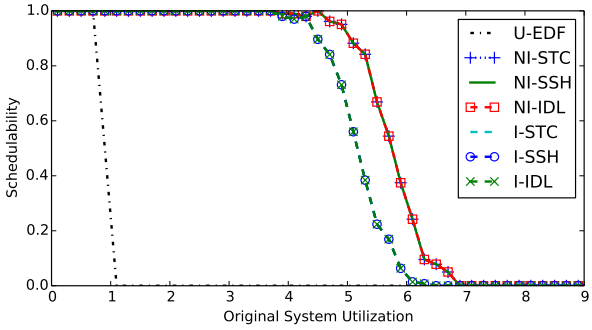
AB-Mod., Short, Light, Light, Mod., Light, Heavy



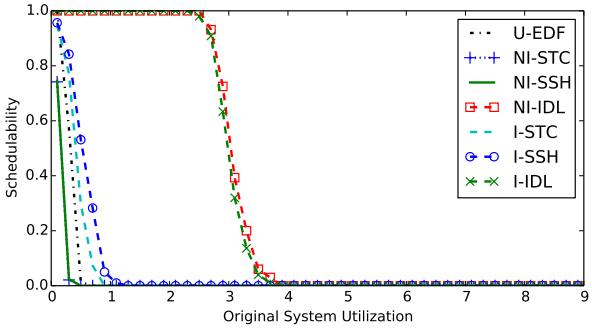
C-Heavy, Short, Light, Light, Large, Light, Light



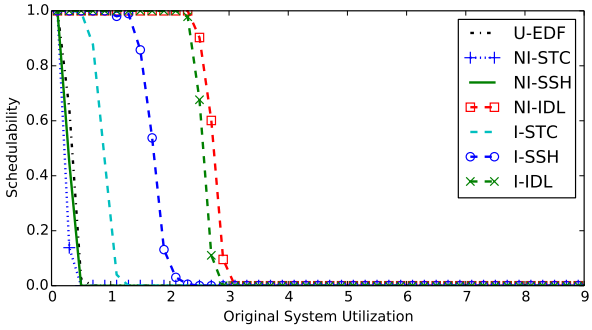
AB-Mod., Long, Heavy, Heavy, Large, Heavy, Heavy



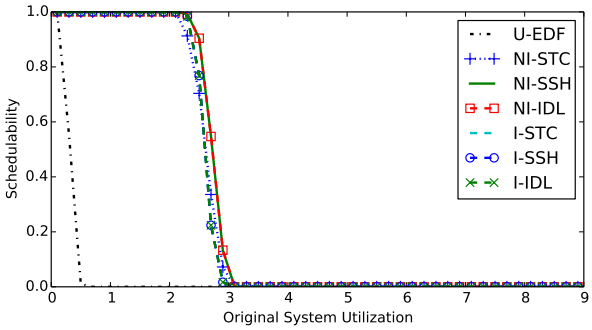
BC-Mod., Long, Heavy, Light, Mod., Light, Heavy



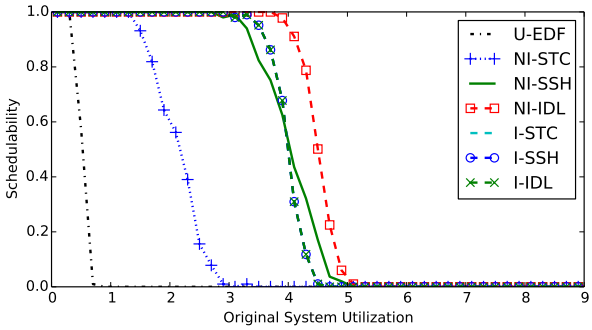
C-Heavy, Long, Light, Heavy, Small, Heavy, Heavy



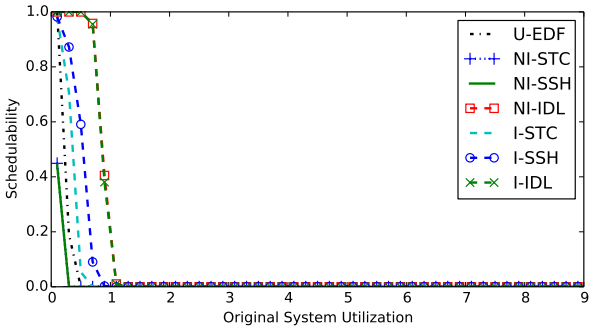
AB-Mod., Long, Light, Heavy, Small, Light, Light



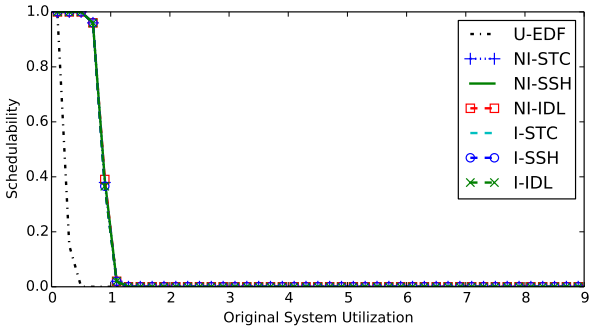
All-Mod., Long, Light, Heavy, Mod., Light, Light



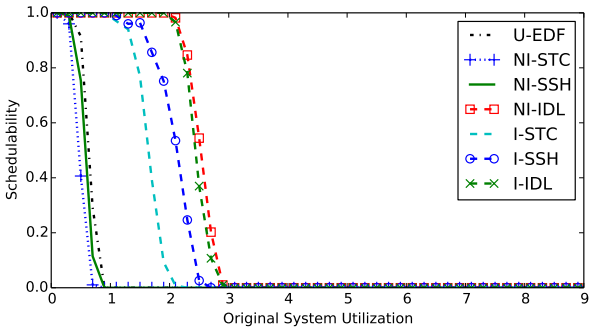
All-Mod., Short, Heavy, Heavy, Small, Light, Heavy



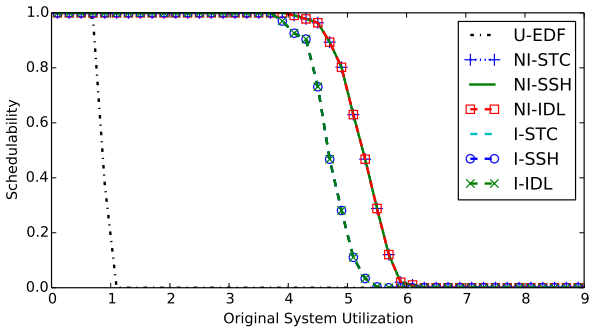
AC-Mod., Short, Light, Heavy, Small, Heavy, Heavy



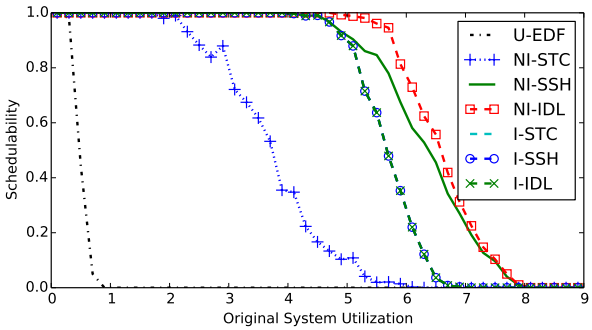
AC-Mod., Short, Light, Heavy, Large, Light, Light



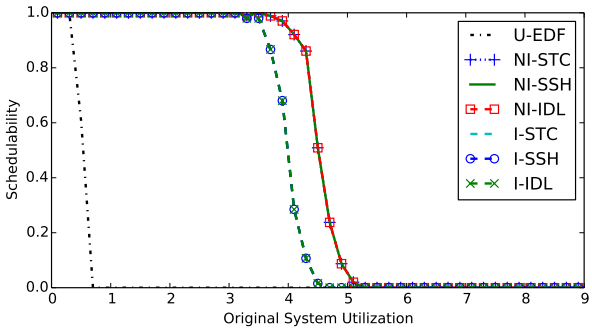
AC-Mod., Long, Light, Light, Large, Heavy, Light



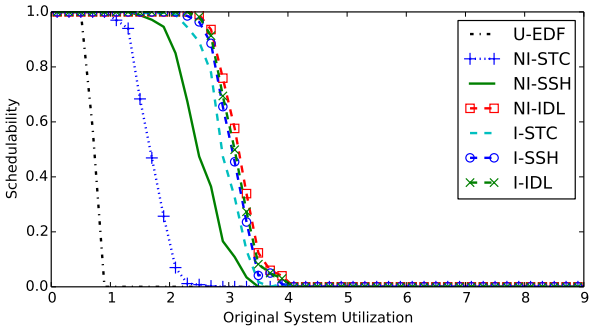
All-Mod., Long, Heavy, Light, Large, Light, Heavy



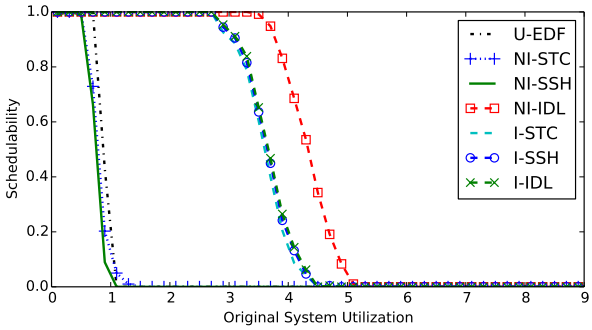
C-Heavy, Short, Heavy, Heavy, Small, Light, Heavy



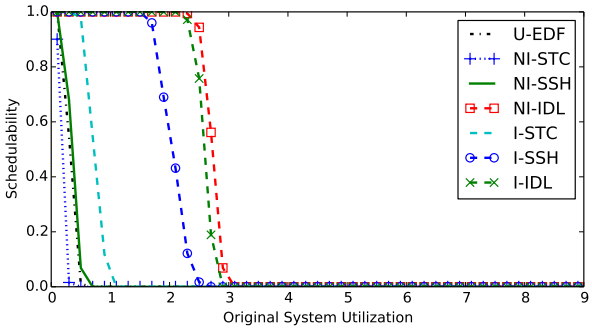
All-Mod., Short, Heavy, Heavy, Mod., Heavy, Light



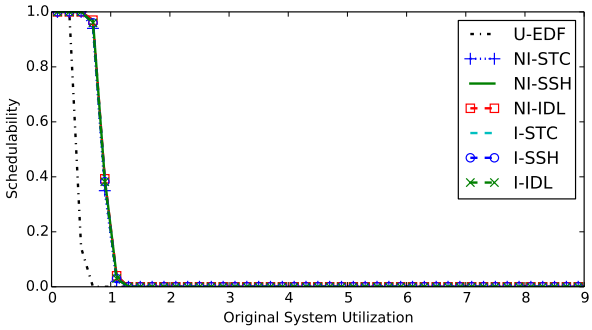
C-Heavy, Long, Light, Light, Mod., Heavy, Heavy



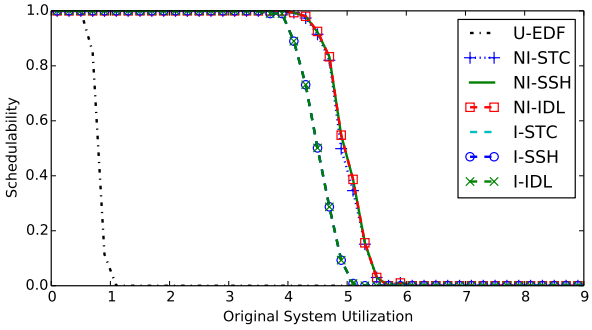
A-Heavy, Long, Heavy, Light, Small, Heavy, Light



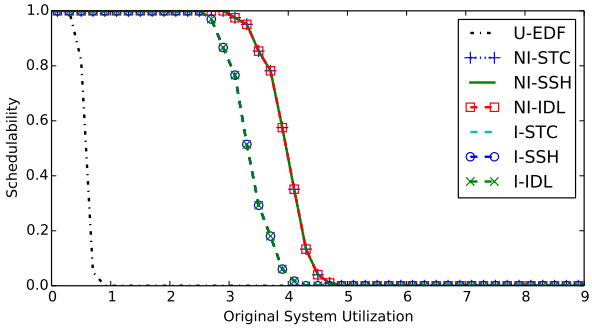
All-Mod., Long, Light, Heavy, Small, Light, Heavy



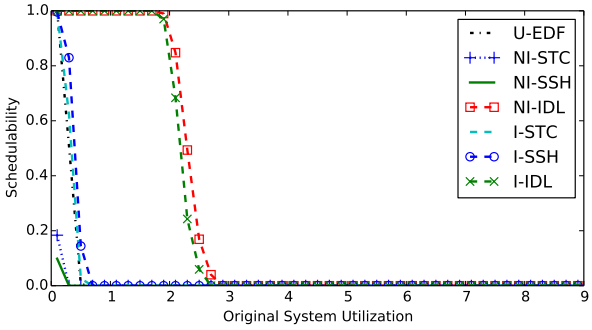
AC-Mod., Short, Light, Light, Mod., Heavy, Heavy



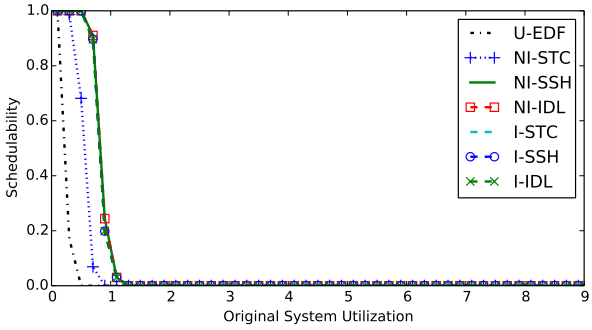
AC-Mod., Short, Heavy, Light, Large, Heavy, Light



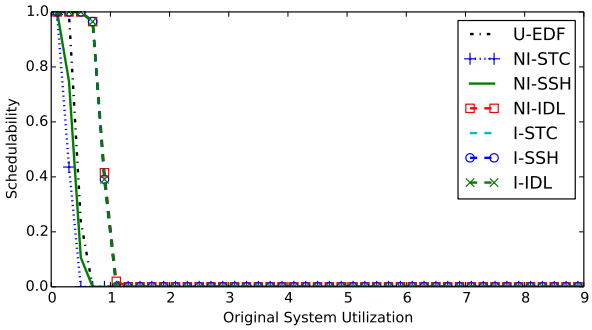
B-Heavy, Short, Heavy, Heavy, Mod., Heavy, Heavy



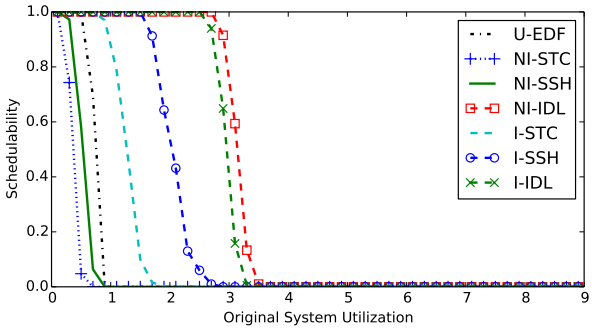
A-Heavy, Long, Light, Heavy, Small, Heavy, Light



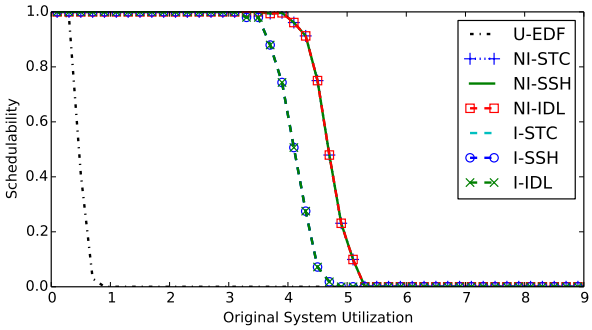
A-Heavy, Short, Light, Heavy, Large, Light, Heavy



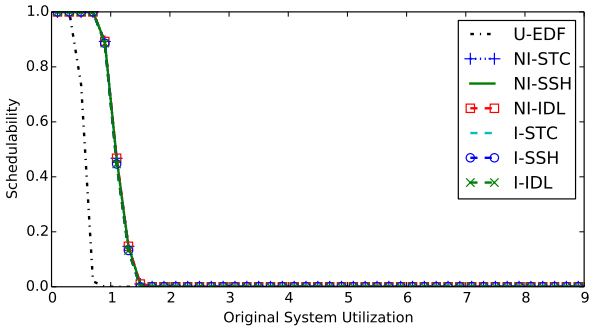
AC-Mod., Short, Light, Light, Small, Light, Light



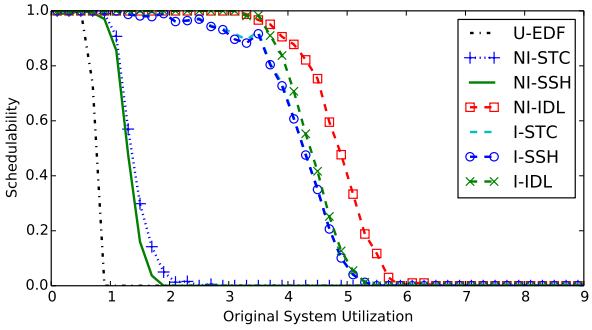
B-Heavy, Long, Light, Light, Large, Heavy, Heavy



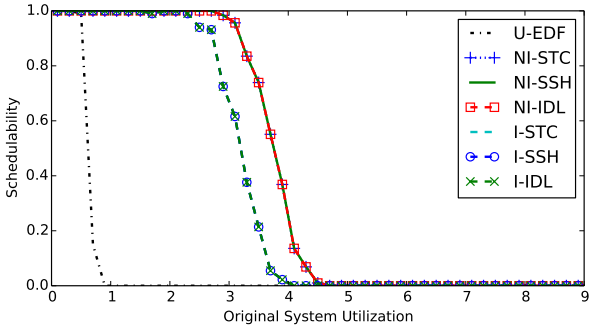
AC-Mod., Short, Heavy, Heavy, Mod., Heavy, Light



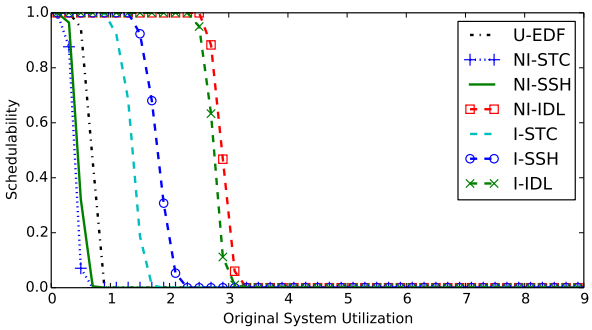
C-Heavy, Short, Light, Light, Mod., Heavy, Light



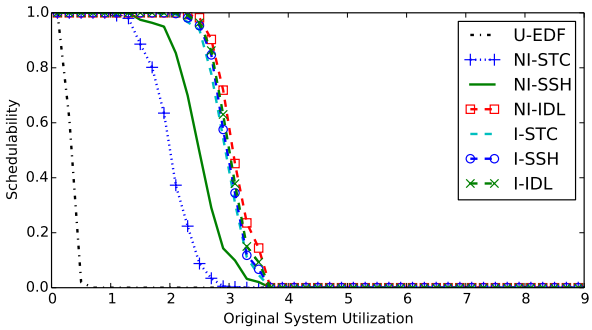
BC-Mod., Long, Heavy, Heavy, Small, Heavy, Light



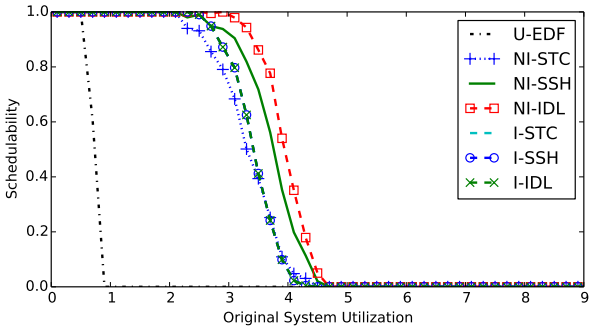
A-Heavy, Long, Heavy, Heavy, Mod., Light, Heavy



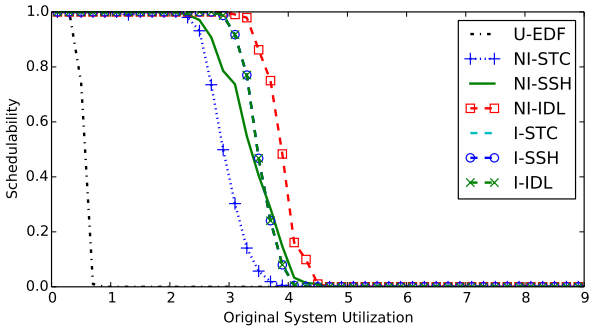
AB-Mod., Long, Light, Light, Large, Heavy, Light



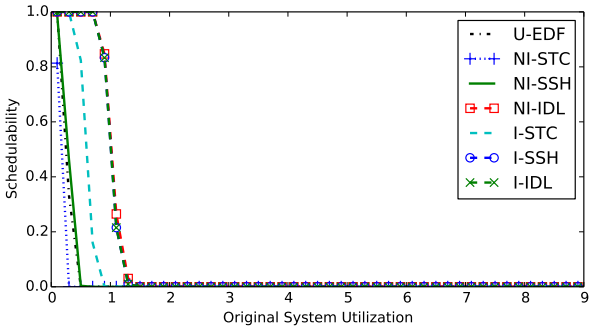
C-Heavy, Long, Light, Heavy, Mod., Heavy, Light



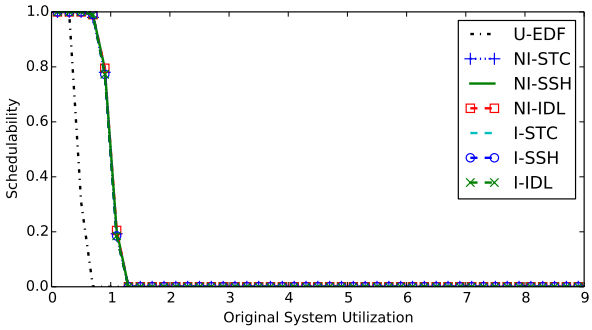
B-Heavy, Long, Heavy, Heavy, Small, Light, Light



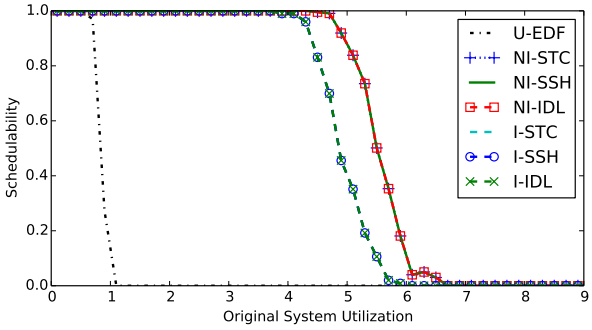
AB-Mod., Short, Heavy, Heavy, Small, Light, Light



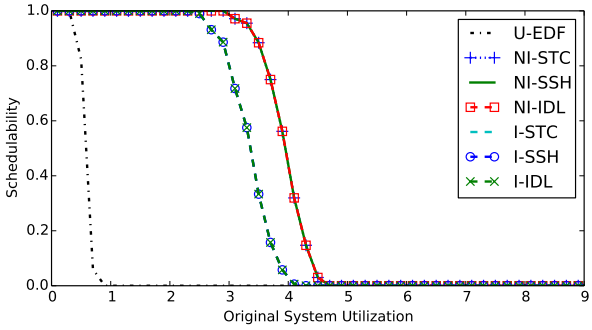
AB-Mod., Short, Light, Heavy, Small, Light, Heavy



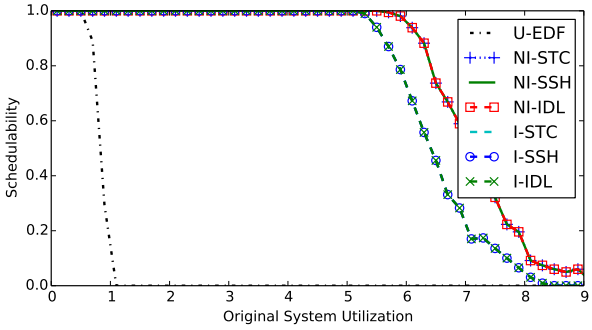
All-Mod., Short, Light, Light, Mod., Heavy, Light



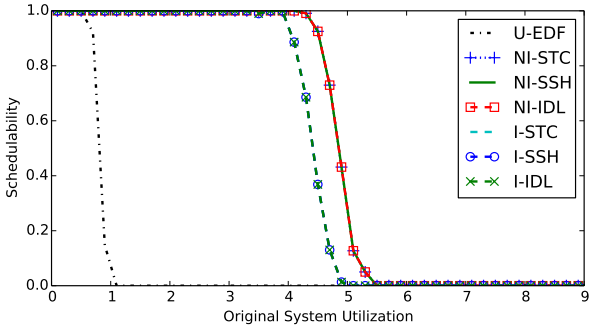
BC-Mod., Short, Heavy, Light, Large, Light, Heavy



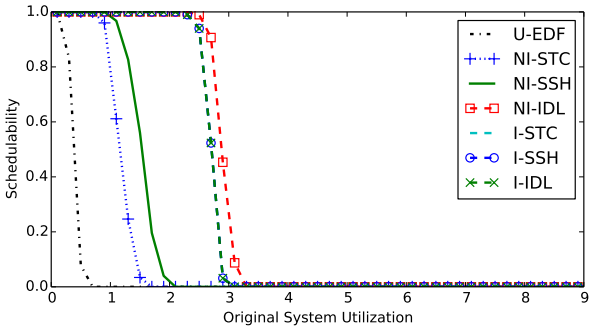
B-Heavy, Short, Heavy, Heavy, Large, Light, Light



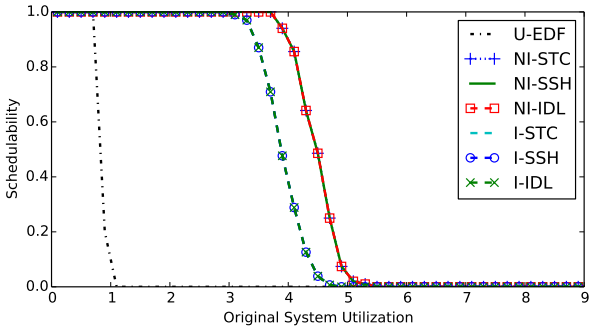
C-Heavy, Short, Heavy, Light, Large, Light, Light



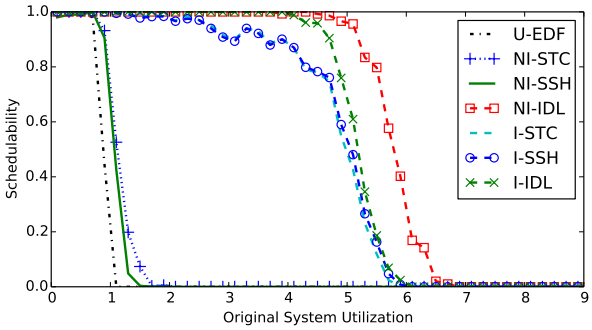
All-Mod., Short, Heavy, Light, Large, Light, Heavy



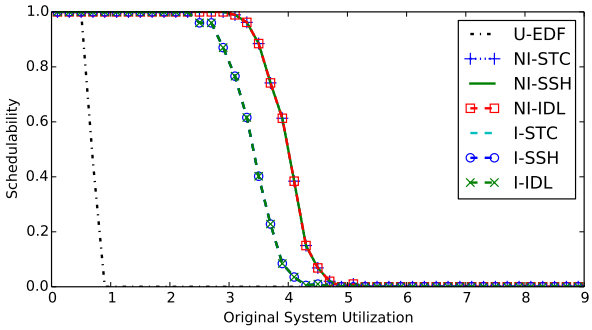
B-Heavy, Long, Light, Heavy, Mod., Heavy, Light



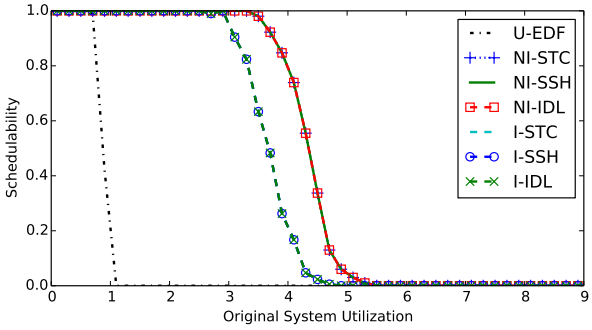
B-Heavy, Short, Heavy, Light, Mod., Heavy, Light



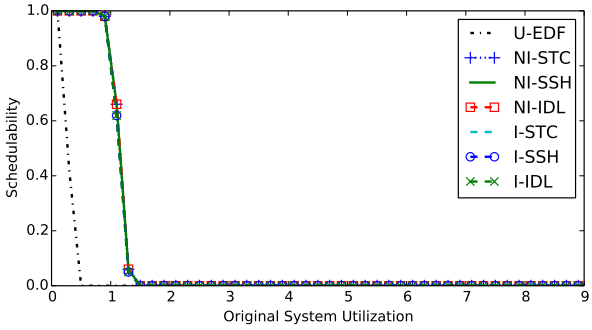
AC-Mod., Long, Heavy, Light, Small, Heavy, Light



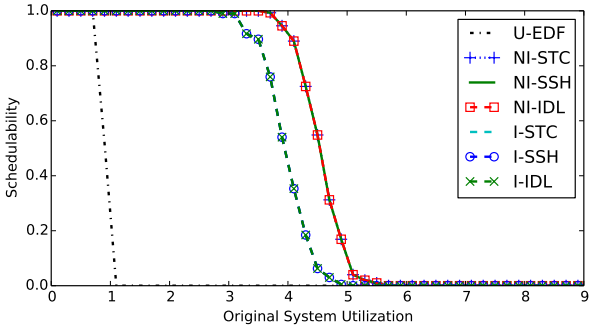
B-Heavy, Long, Heavy, Heavy, Mod., Light, Heavy



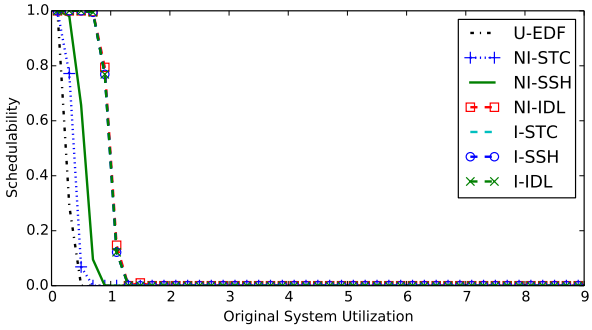
A-Heavy, Long, Heavy, Light, Mod., Light, Heavy



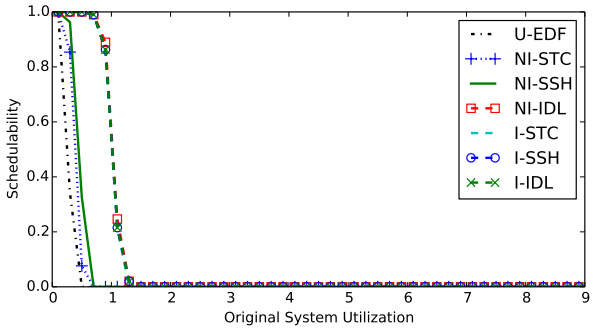
BC-Mod., Short, Light, Heavy, Mod., Light, Light



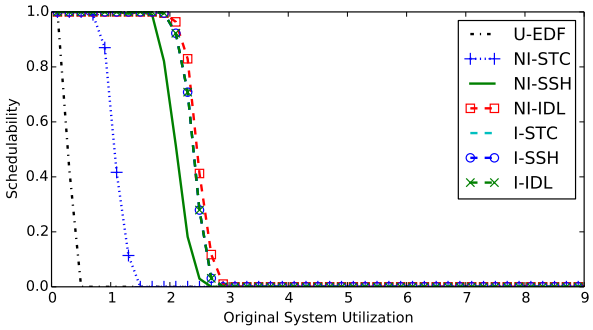
B-Heavy, Long, Heavy, Light, Mod., Heavy, Light



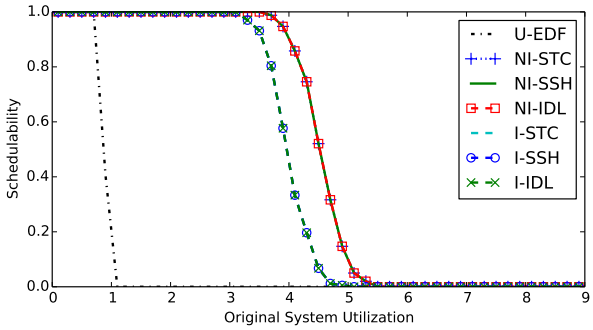
All-Mod., Short, Light, Heavy, Large, Heavy, Heavy



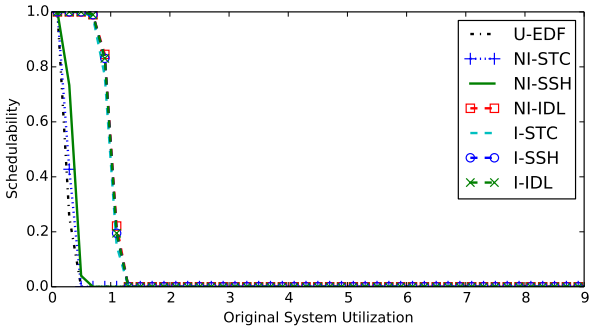
AB-Mod., Short, Light, Heavy, Large, Heavy, Light



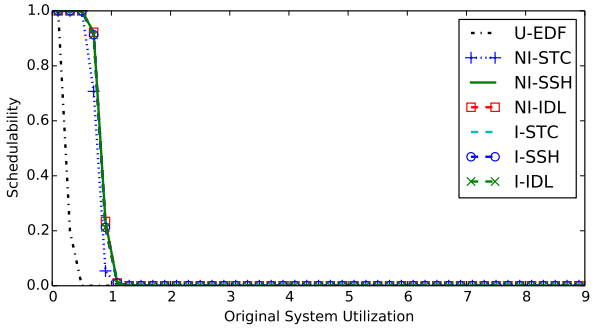
AC-Mod., Long, Light, Heavy, Large, Light, Light



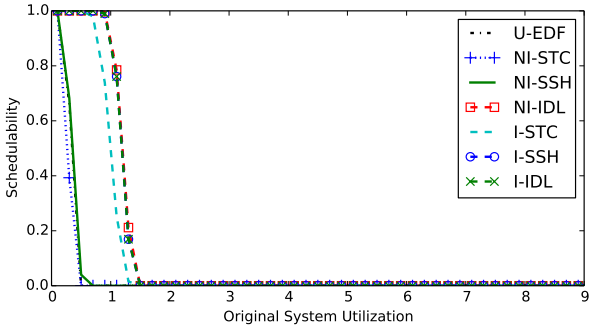
AB-Mod., Long, Heavy, Light, Large, Light, Light



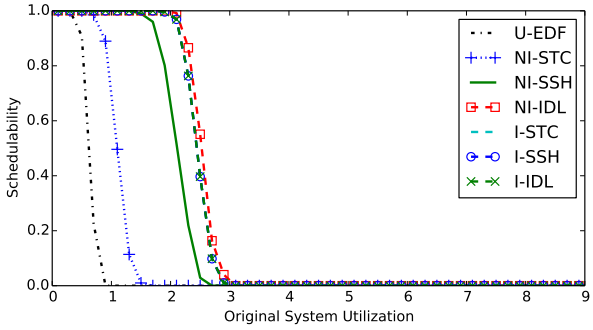
All-Mod., Short, Light, Heavy, Small, Light, Light



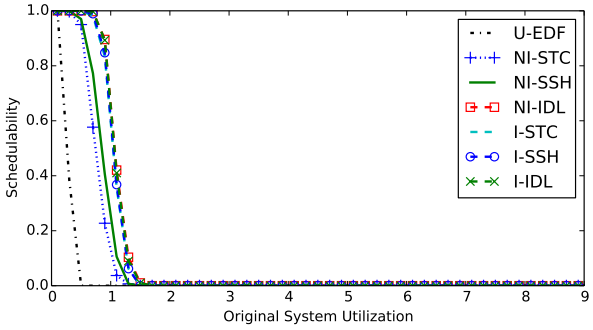
A-Heavy, Short, Light, Heavy, Large, Light, Light



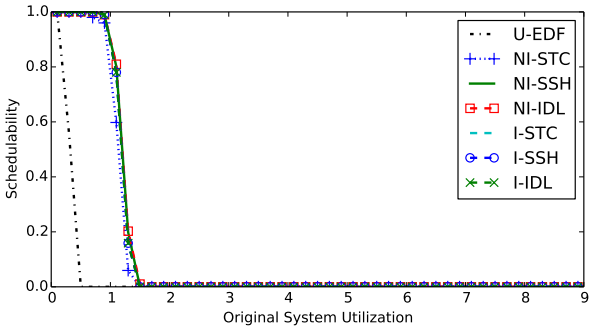
B-Heavy, Short, Light, Heavy, Small, Light, Light



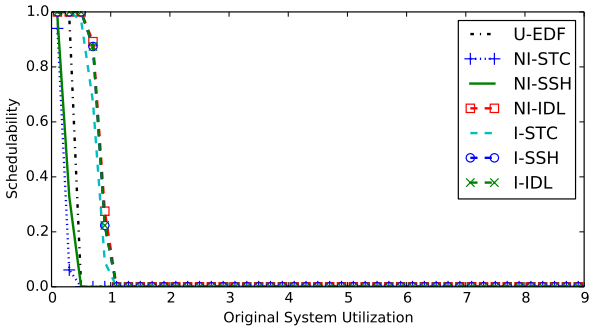
AC-Mod., Long, Light, Light, Large, Light, Light



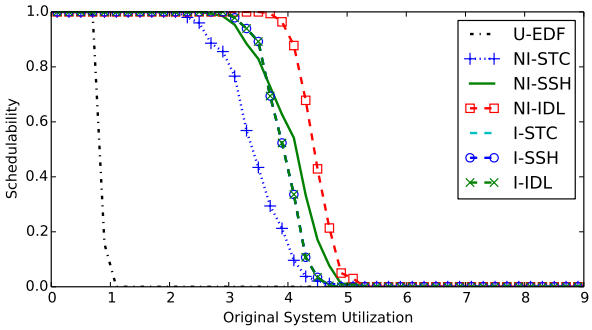
C-Heavy, Short, Light, Heavy, Large, Heavy, Light



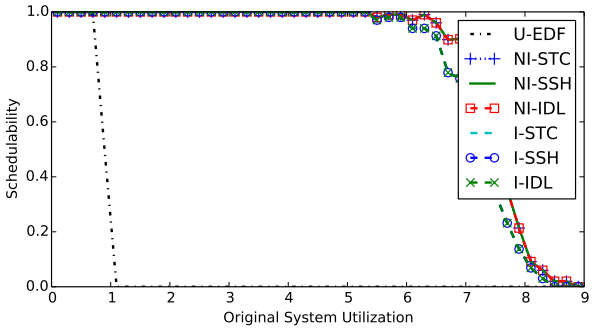
B-Heavy, Short, Light, Heavy, Mod., Heavy, Light



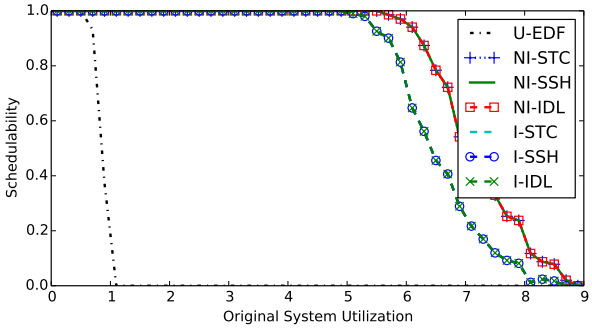
A-Heavy, Short, Light, Light, Small, Light, Light



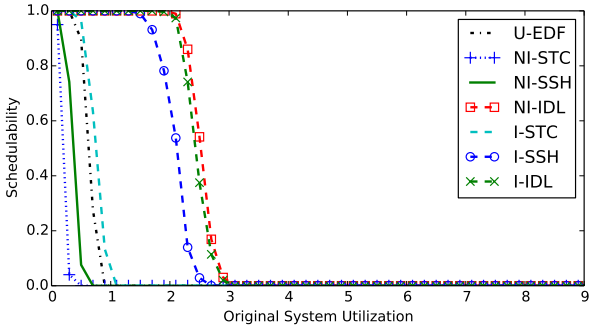
B-Heavy, Short, Heavy, Light, Small, Light, Light



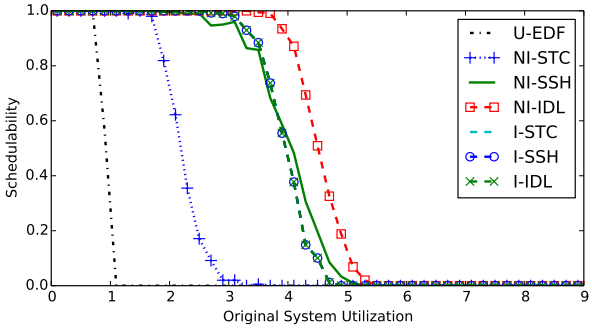
C-Heavy, Long, Heavy, Light, Mod., Light, Heavy



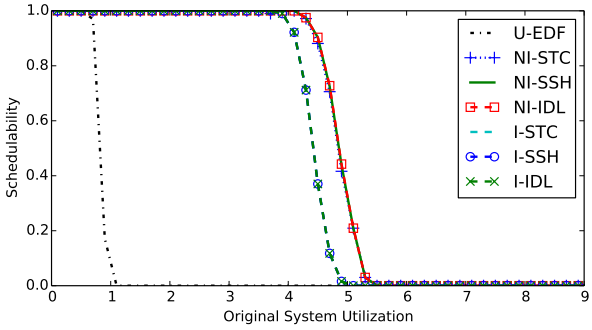
C-Heavy, Short, Heavy, Light, Mod., Heavy, Heavy



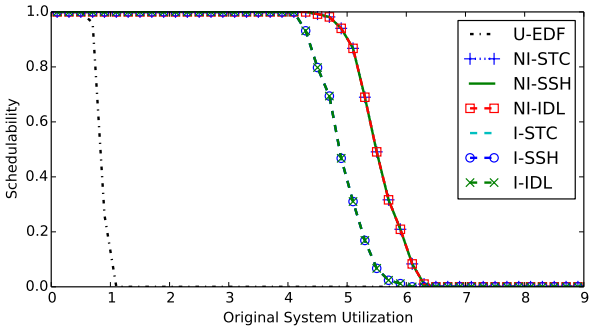
AC-Mod., Long, Light, Light, Small, Light, Heavy



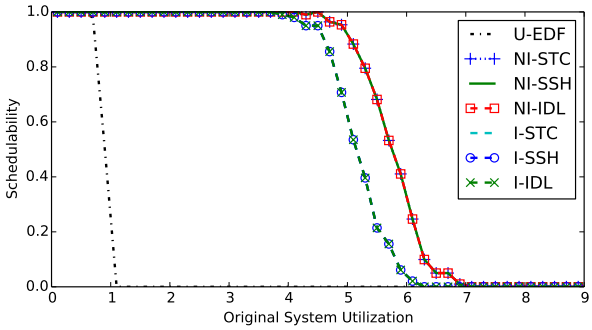
B-Heavy, Long, Heavy, Light, Small, Light, Heavy



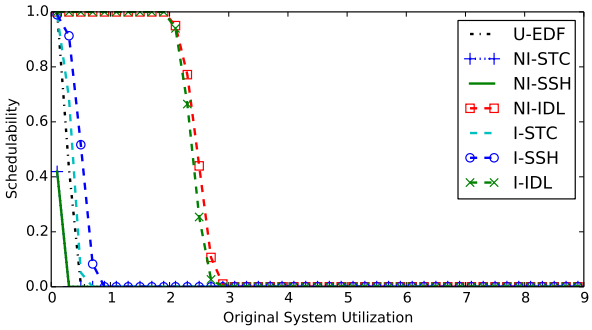
All-Mod., Short, Heavy, Light, Large, Heavy, Light



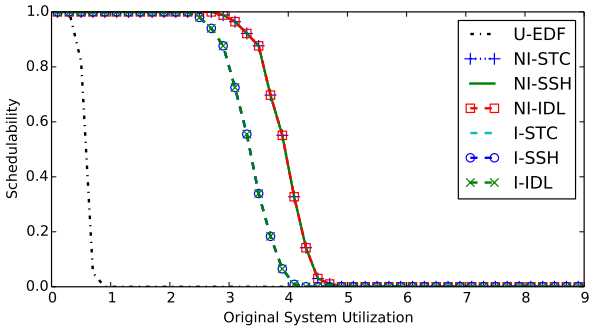
BC-Mod., Short, Heavy, Light, Mod., Heavy, Light



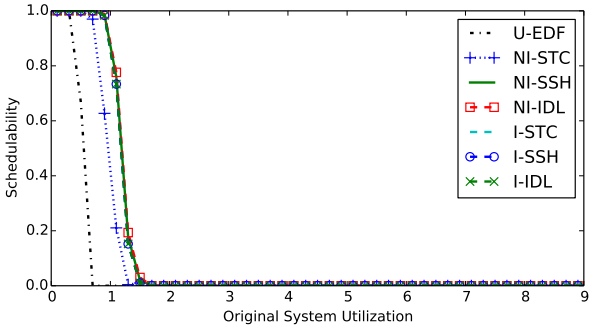
BC-Mod., Long, Heavy, Light, Mod., Light, Light



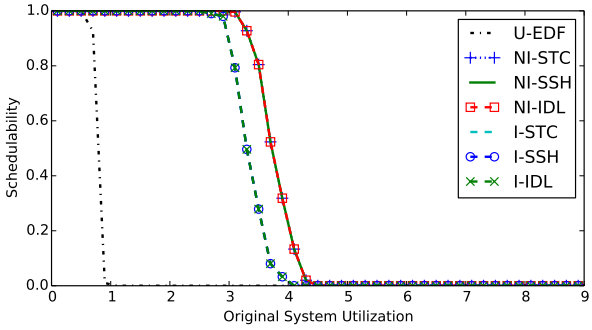
AC-Mod., Long, Light, Heavy, Small, Heavy, Heavy



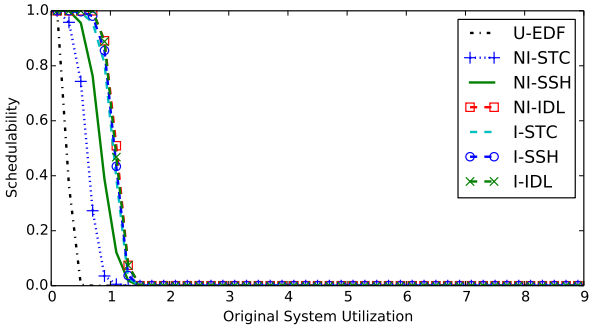
B-Heavy, Short, Heavy, Heavy, Mod., Heavy, Light



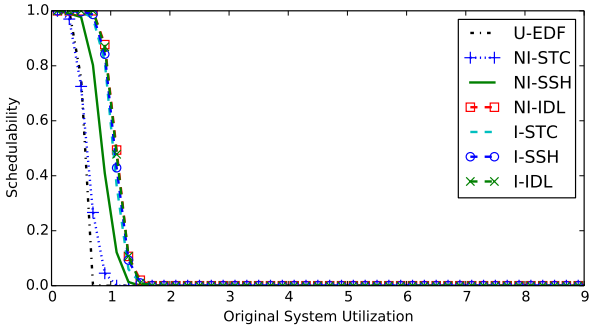
B-Heavy, Short, Light, Light, Mod., Heavy, Heavy



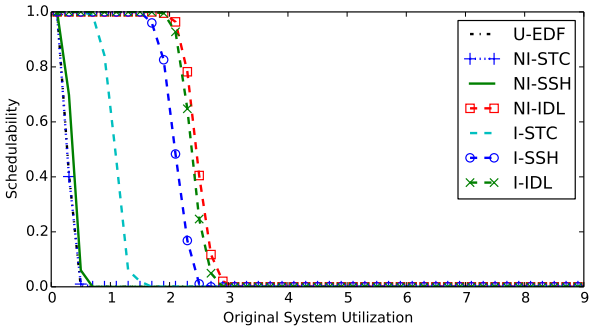
A-Heavy, Short, Heavy, Light, Mod., Heavy, Heavy



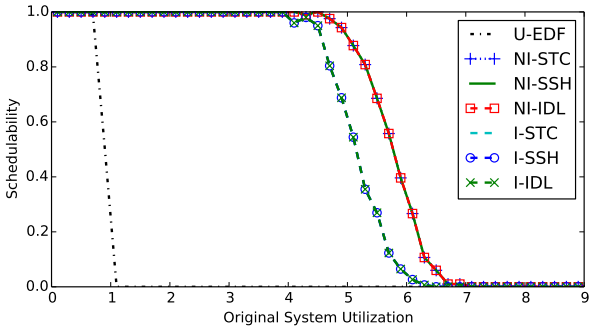
C-Heavy, Short, Light, Heavy, Large, Heavy, Heavy



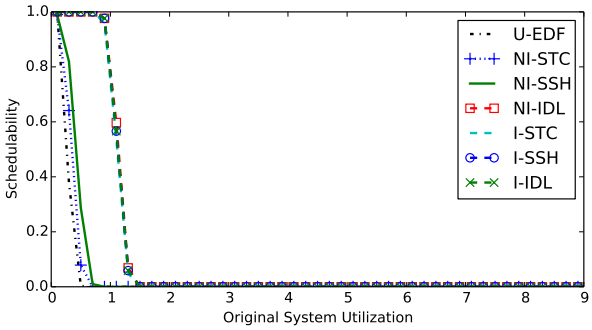
C-Heavy, Short, Light, Light, Large, Heavy, Heavy



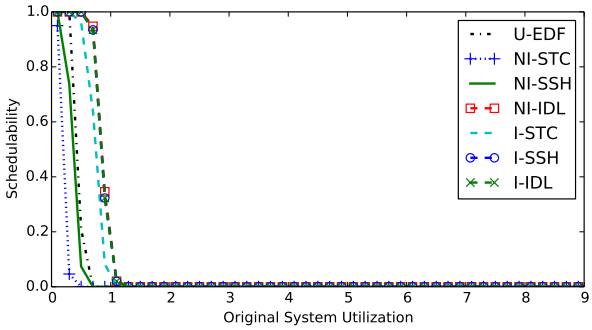
AC-Mod., Long, Light, Heavy, Small, Light, Light



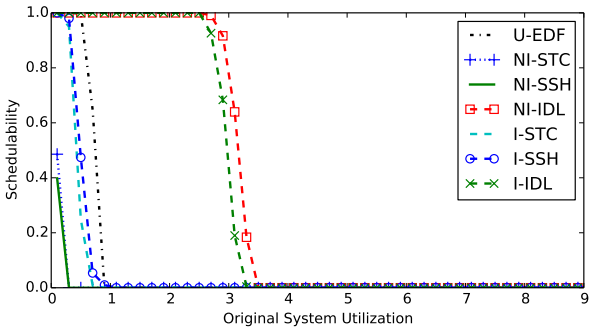
BC-Mod., Long, Heavy, Light, Large, Light, Heavy



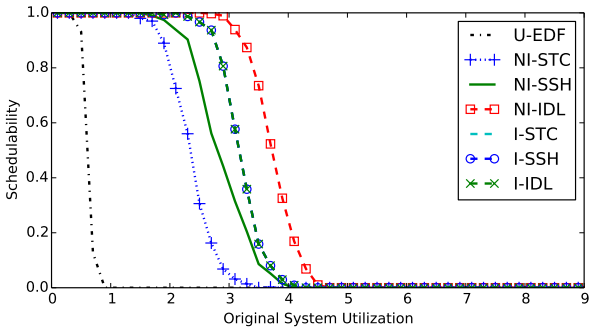
BC-Mod., Short, Light, Heavy, Small, Light, Light



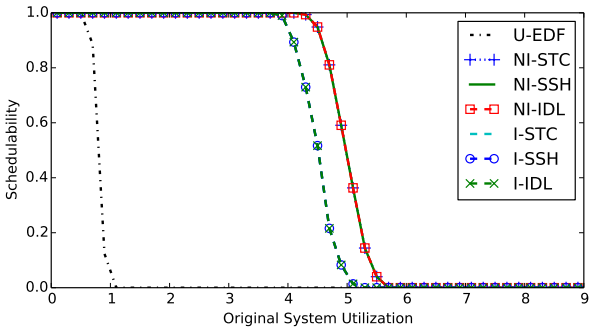
AC-Mod., Short, Light, Light, Small, Light, Heavy



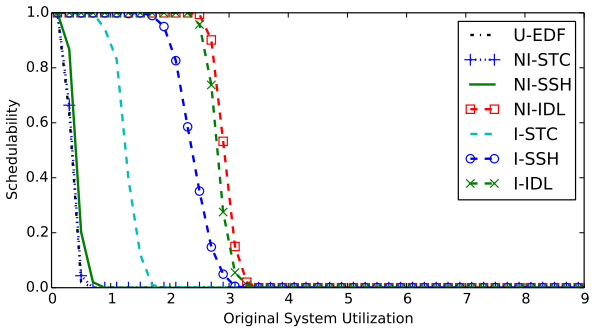
B-Heavy, Long, Light, Light, Small, Heavy, Light



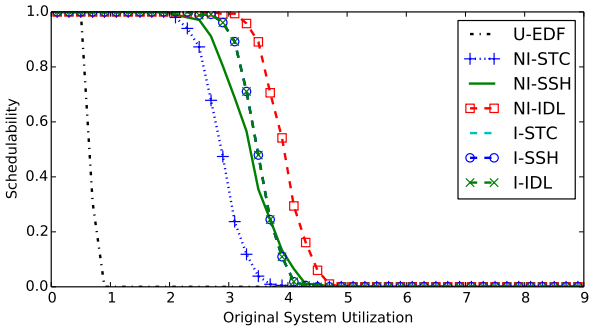
A-Heavy, Long, Heavy, Heavy, Small, Light, Light



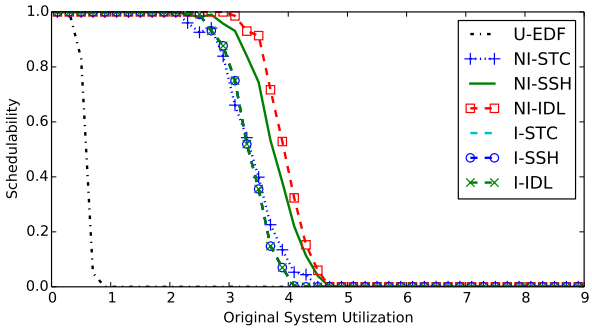
AC-Mod., Short, Heavy, Light, Mod., Light, Heavy



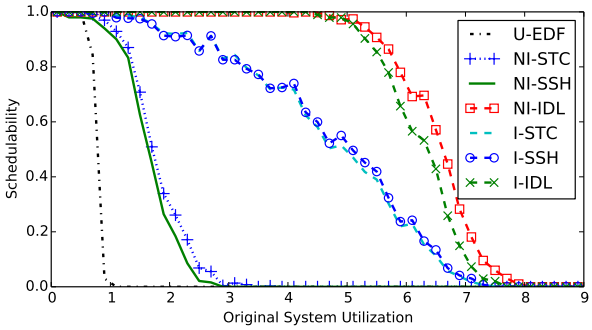
BC-Mod., Long, Light, Heavy, Small, Light, Light



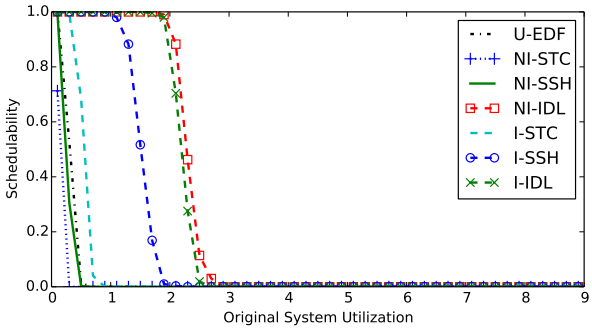
AB-Mod., Long, Heavy, Heavy, Small, Light, Light



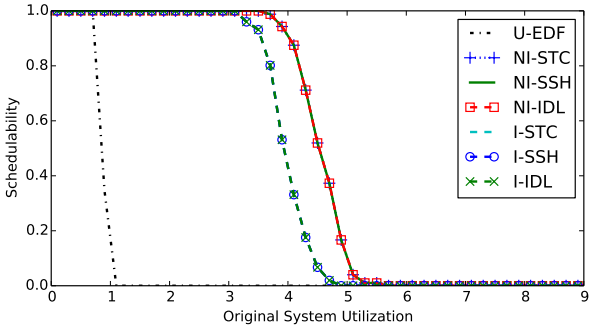
B-Heavy, Short, Heavy, Heavy, Small, Light, Light



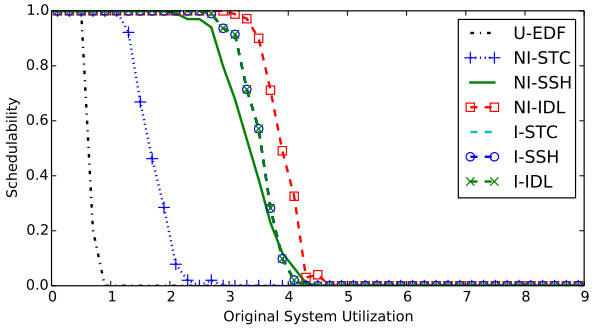
C-Heavy, Long, Heavy, Heavy, Small, Heavy, Light



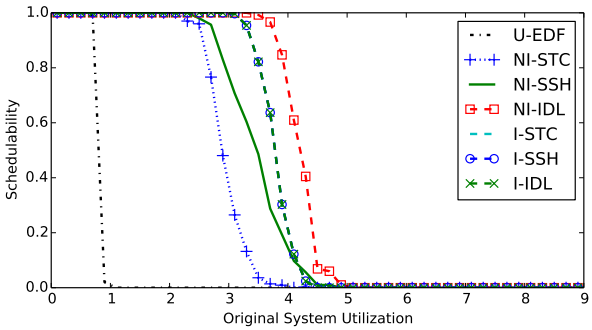
A-Heavy, Long, Light, Heavy, Small, Light, Heavy



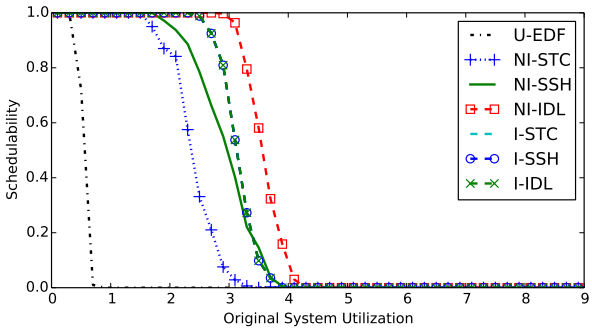
AB-Mod., Long, Heavy, Light, Mod., Heavy, Heavy



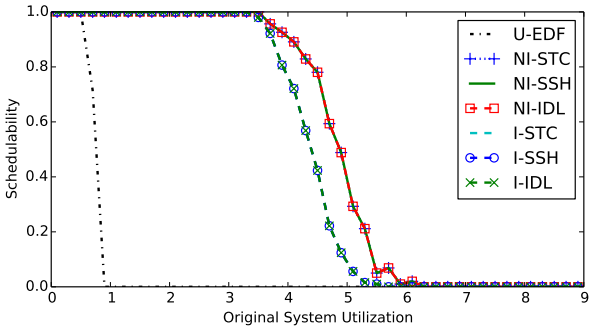
AB-Mod., Long, Heavy, Heavy, Small, Light, Heavy



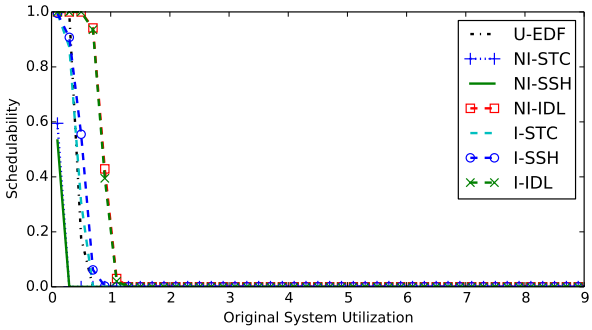
AB-Mod., Short, Heavy, Light, Small, Light, Light



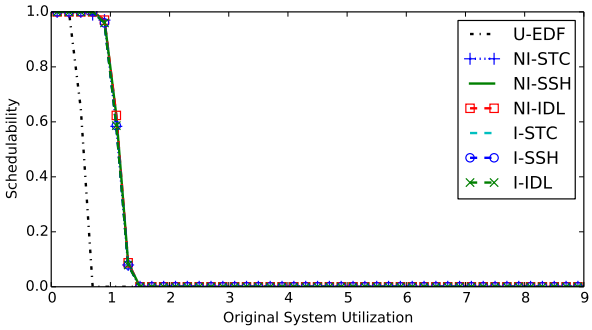
A-Heavy, Short, Heavy, Heavy, Small, Light, Light



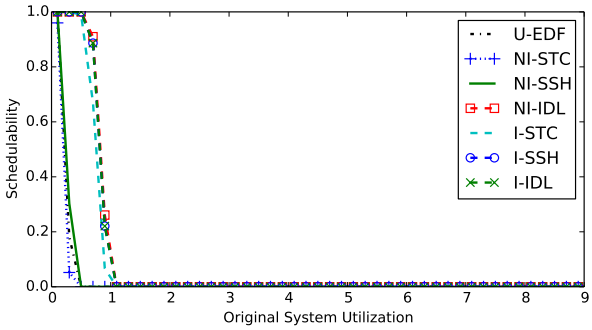
BC-Mod., Long, Heavy, Heavy, Mod., Light, Heavy



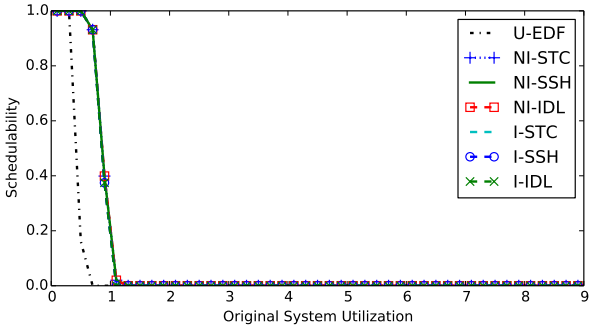
AC-Mod., Short, Light, Light, Small, Heavy, Light



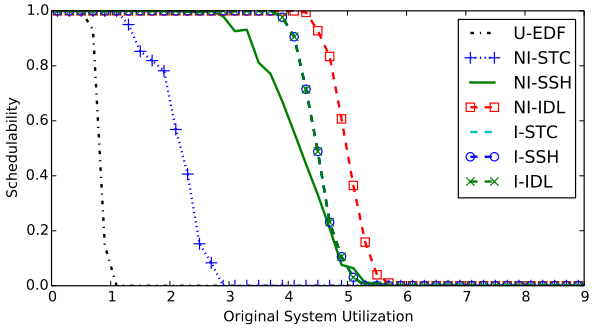
BC-Mod., Short, Light, Light, Large, Light, Light



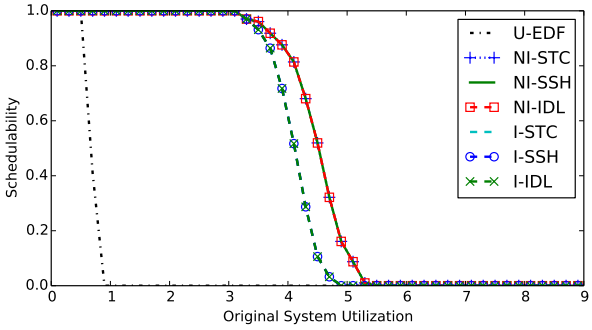
A-Heavy, Short, Light, Heavy, Small, Light, Light



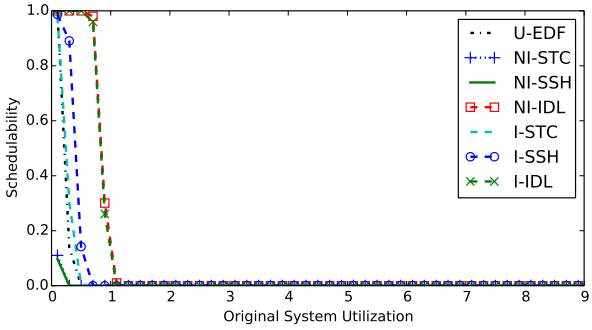
AC-Mod., Short, Light, Light, Mod., Heavy, Light



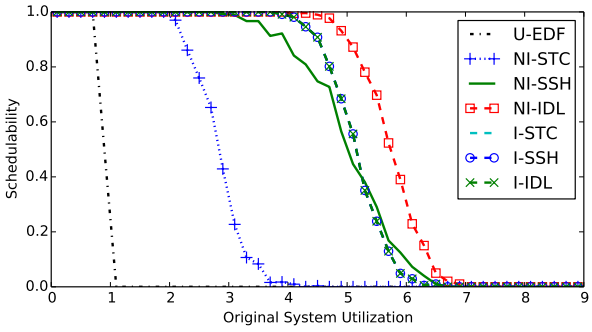
AC-Mod., Short, Heavy, Light, Small, Light, Heavy



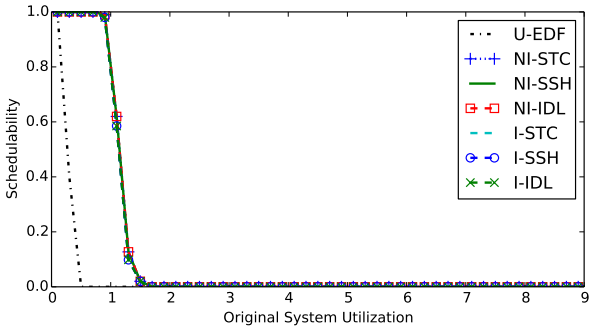
All-Mod., Long, Heavy, Heavy, Mod., Heavy, Light



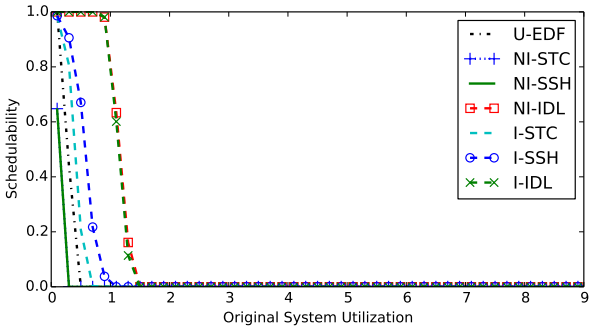
A-Heavy, Short, Light, Heavy, Small, Heavy, Heavy



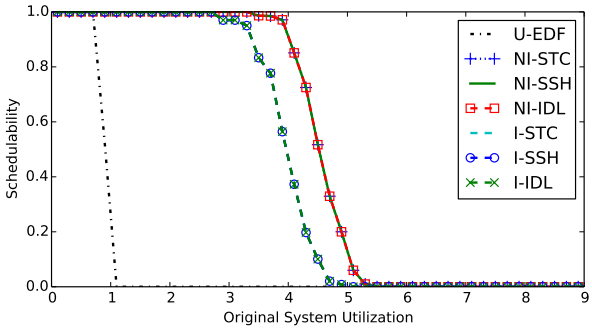
BC-Mod., Long, Heavy, Light, Small, Light, Heavy



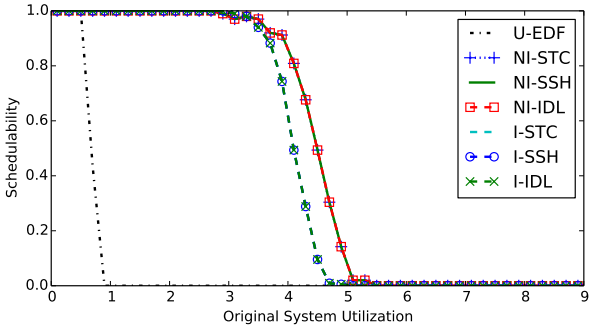
BC-Mod., Short, Light, Heavy, Mod., Light, Heavy



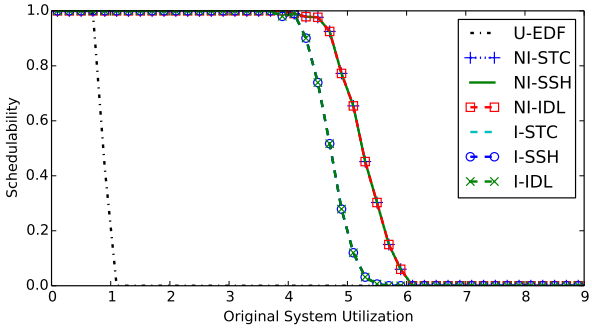
BC-Mod., Short, Light, Heavy, Small, Heavy, Heavy



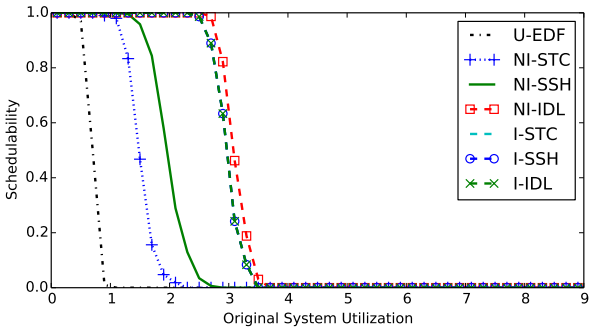
B-Heavy, Long, Heavy, Light, Mod., Heavy, Heavy



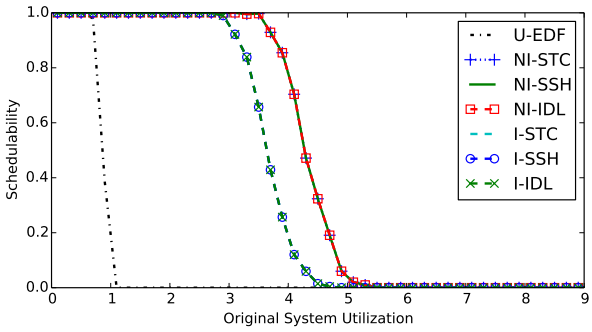
All-Mod., Long, Heavy, Heavy, Mod., Heavy, Heavy



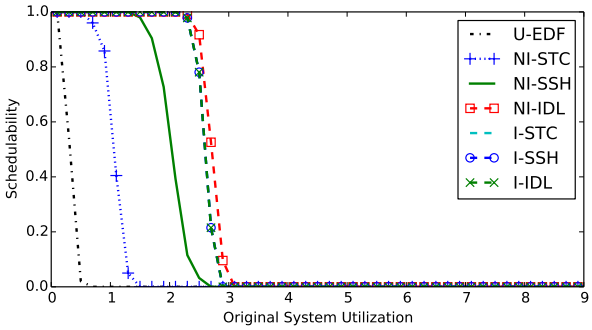
All-Mod., Long, Heavy, Light, Mod., Light, Light



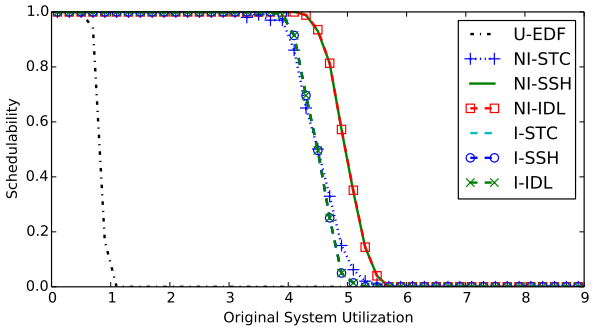
BC-Mod., Long, Light, Light, Mod., Heavy, Light



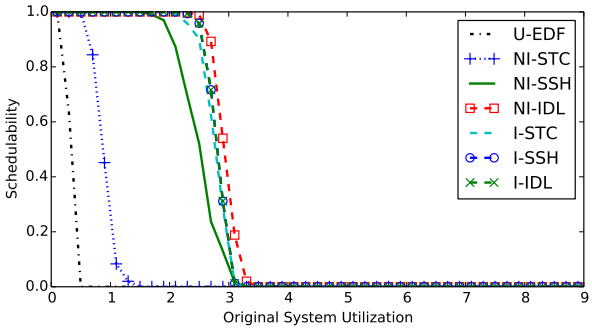
A-Heavy, Long, Heavy, Light, Mod., Light, Light



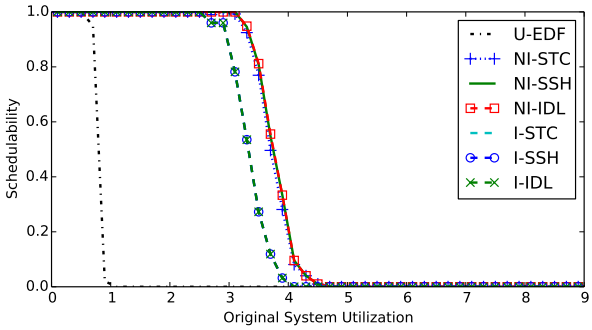
All-Mod., Long, Light, Heavy, Large, Light, Light



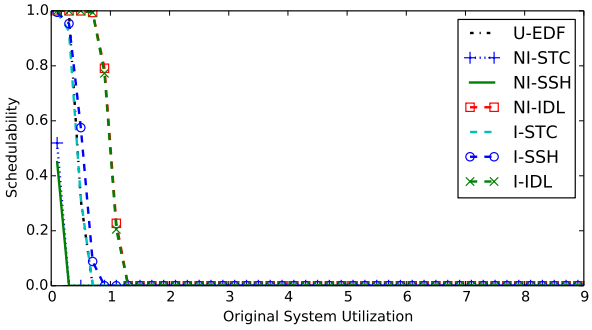
AC-Mod., Short, Heavy, Light, Large, Heavy, Heavy



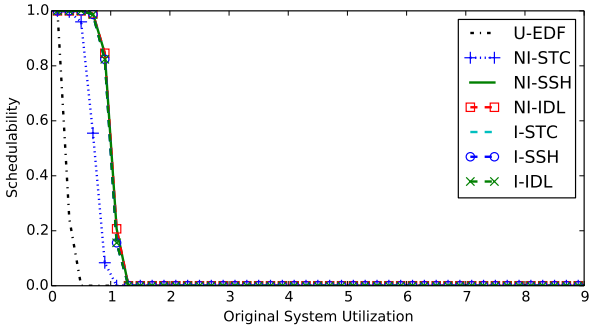
BC-Mod., Long, Light, Heavy, Large, Light, Heavy



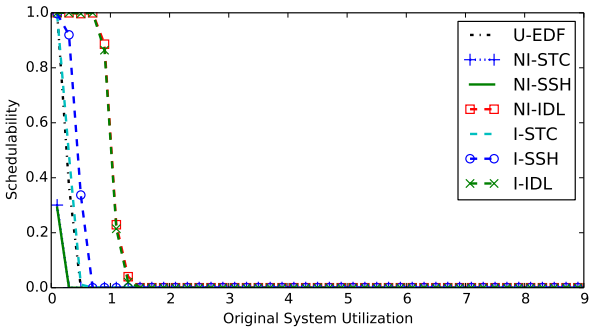
A-Heavy, Short, Heavy, Light, Large, Heavy, Light



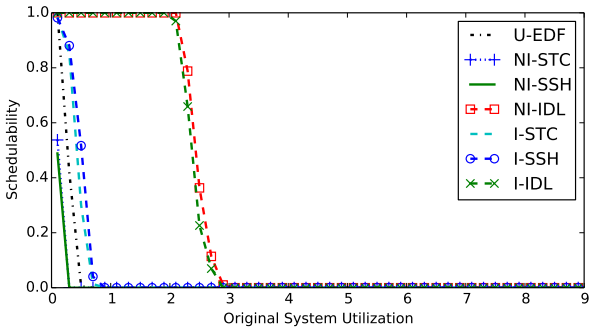
All-Mod., Short, Light, Light, Small, Heavy, Light



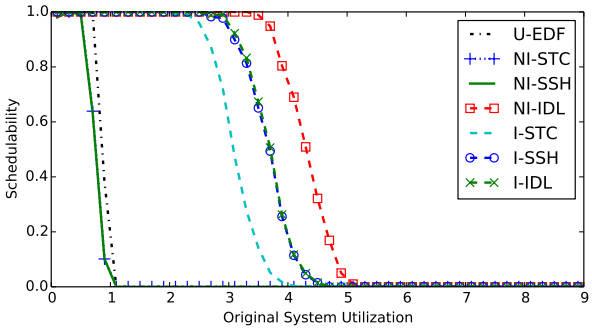
All-Mod., Short, Light, Heavy, Large, Light, Heavy



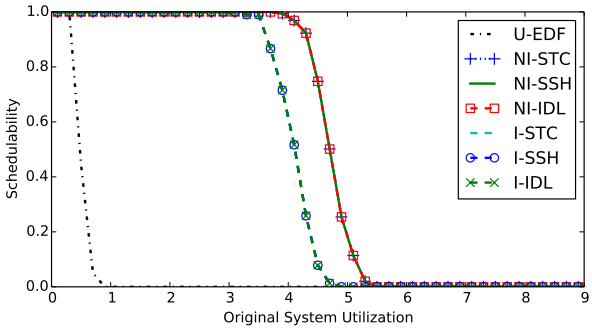
AB-Mod., Short, Light, Heavy, Small, Heavy, Heavy



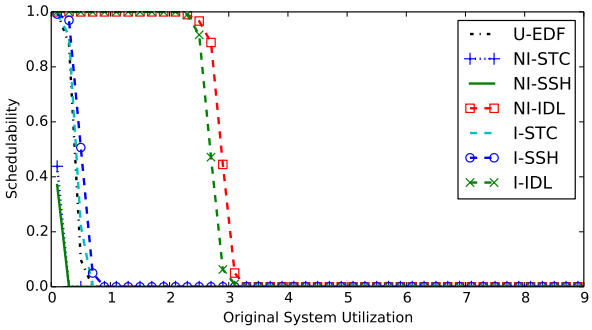
AC-Mod., Long, Light, Heavy, Small, Heavy, Light



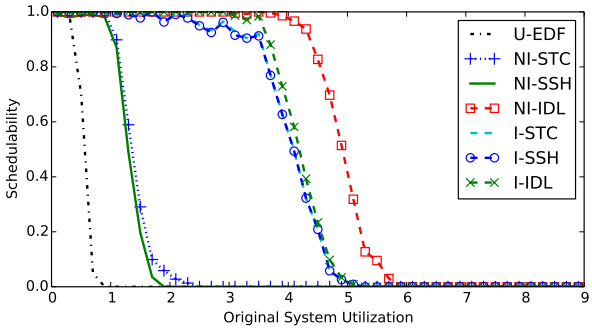
A-Heavy, Long, Heavy, Light, Small, Heavy, Heavy



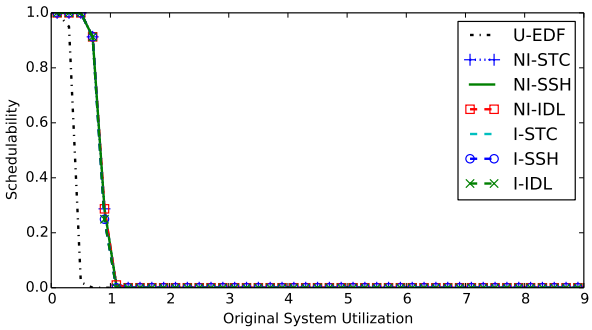
AC-Mod., Short, Heavy, Heavy, Mod., Light, Heavy



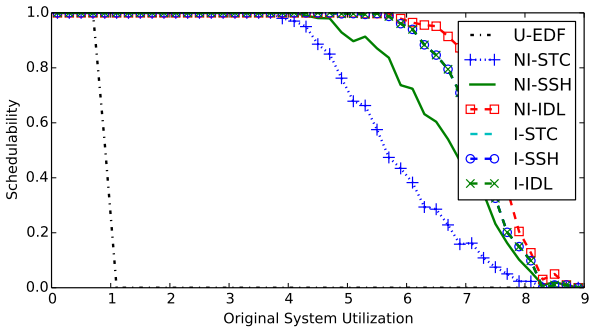
B-Heavy, Long, Light, Heavy, Small, Heavy, Light



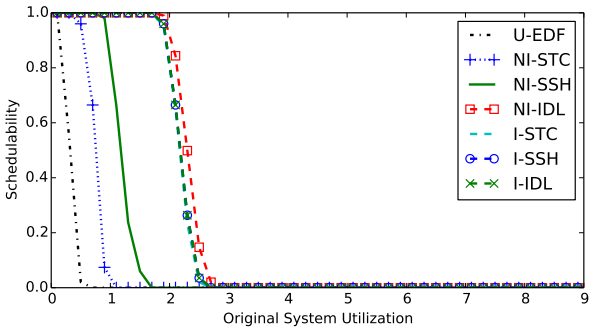
BC-Mod., Short, Heavy, Heavy, Small, Heavy, Light



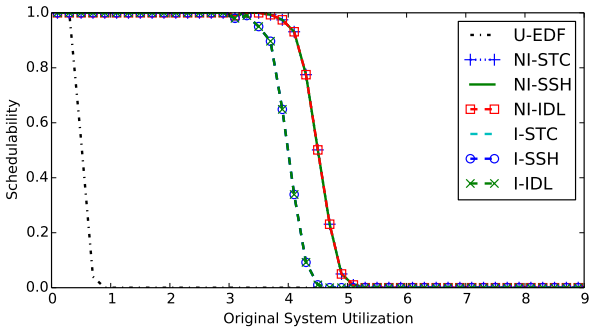
A-Heavy, Short, Light, Light, Mod., Light, Heavy



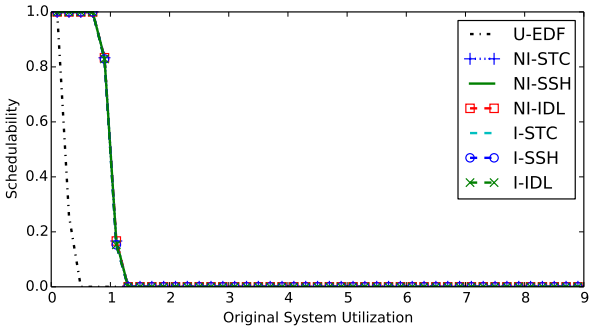
C-Heavy, Long, Heavy, Light, Small, Light, Light



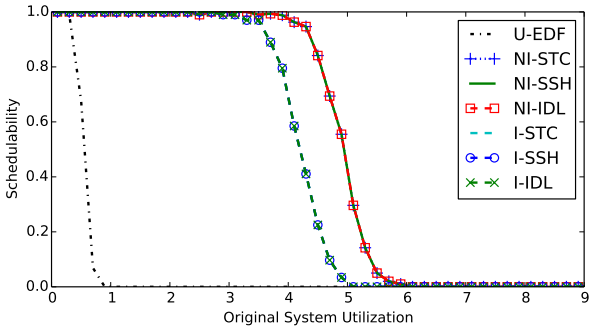
A-Heavy, Long, Light, Heavy, Mod., Heavy, Heavy



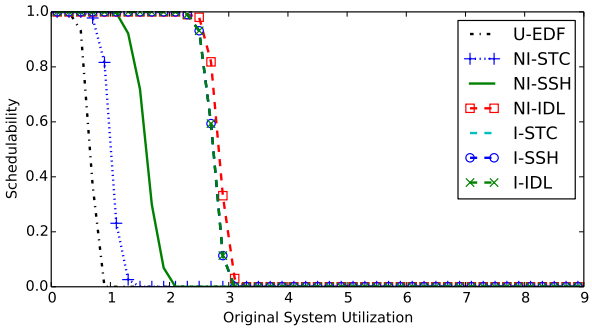
All-Mod., Short, Heavy, Heavy, Mod., Heavy, Heavy



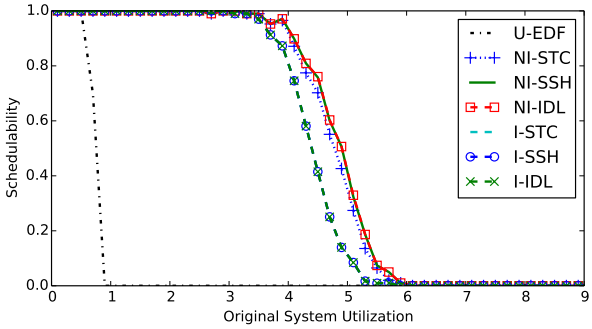
All-Mod., Short, Light, Heavy, Mod., Light, Heavy



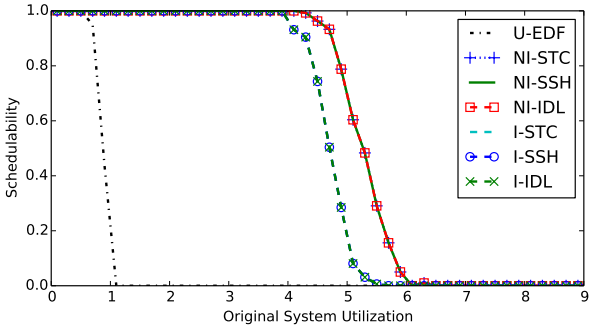
BC-Mod., Short, Heavy, Heavy, Mod., Heavy, Light



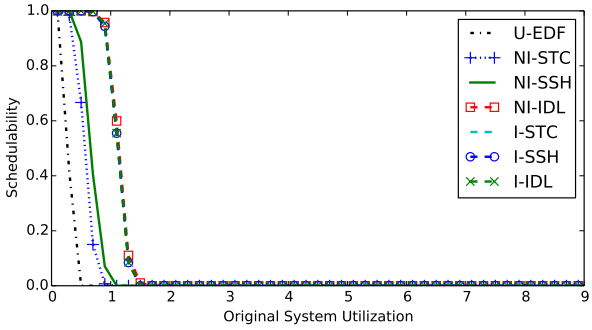
All-Mod., Long, Light, Light, Mod., Heavy, Heavy



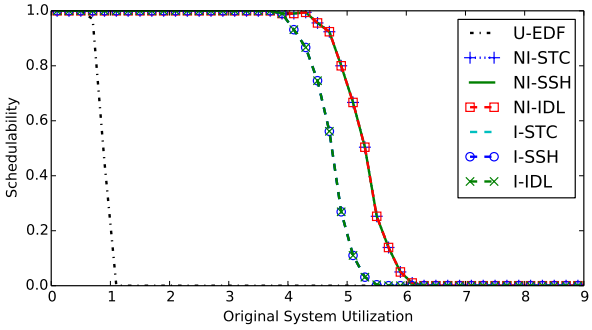
BC-Mod., Long, Heavy, Heavy, Large, Heavy, Heavy



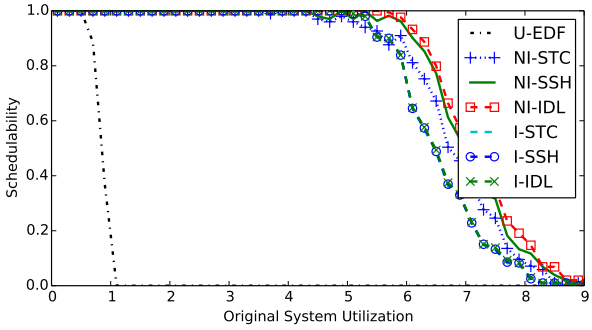
All-Mod., Long, Heavy, Light, Mod., Heavy, Heavy



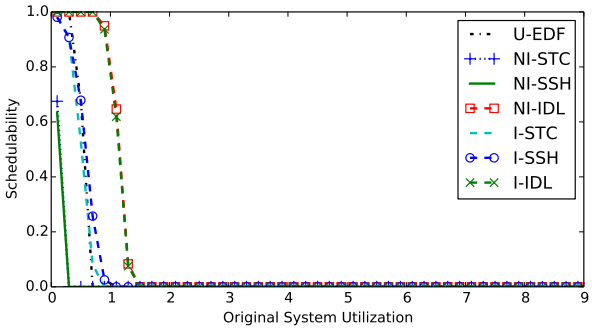
BC-Mod., Short, Light, Heavy, Large, Heavy, Light



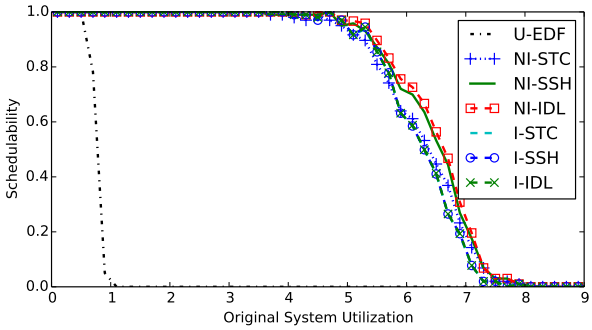
All-Mod., Long, Heavy, Light, Mod., Heavy, Light



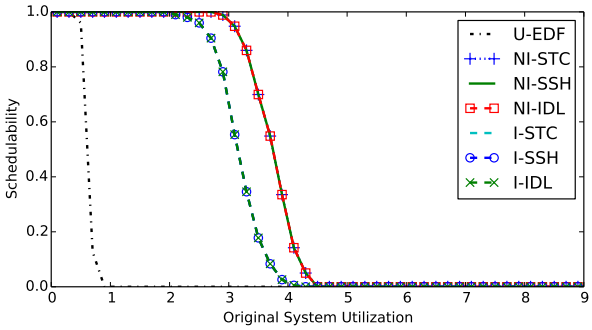
C-Heavy, Short, Heavy, Light, Large, Heavy, Heavy



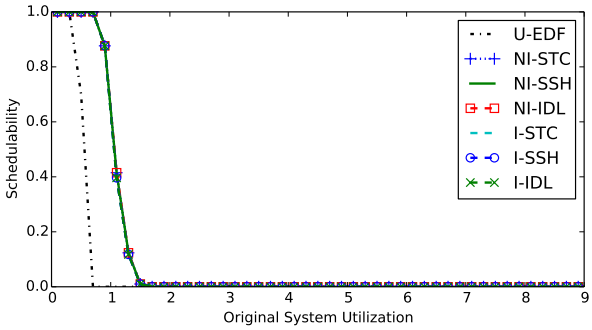
BC-Mod., Short, Light, Light, Small, Heavy, Light



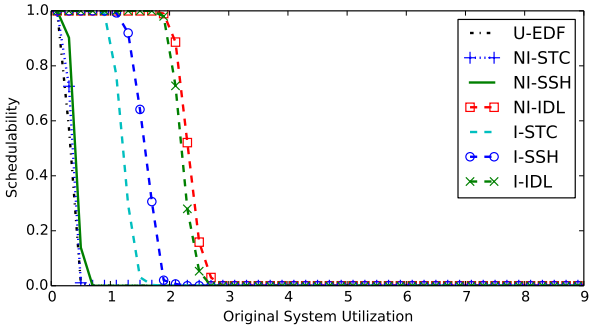
C-Heavy, Long, Heavy, Heavy, Large, Heavy, Heavy



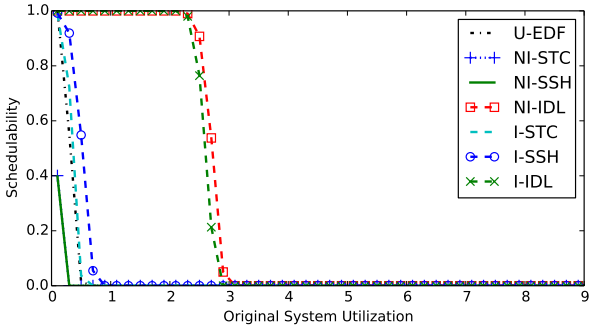
A-Heavy, Long, Heavy, Heavy, Mod., Heavy, Heavy



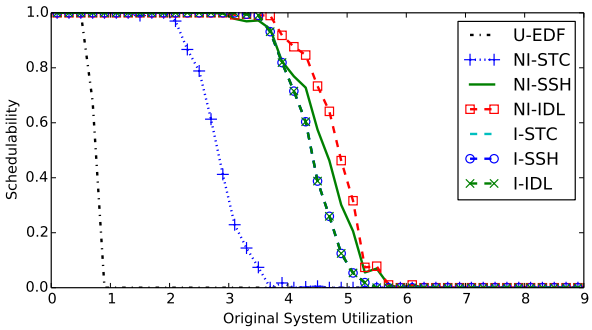
C-Heavy, Short, Light, Light, Mod., Light, Light



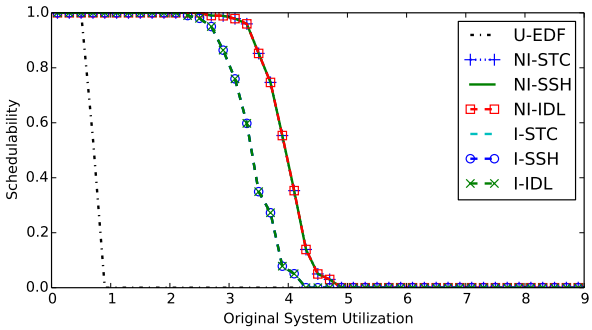
A-Heavy, Long, Light, Heavy, Large, Heavy, Light



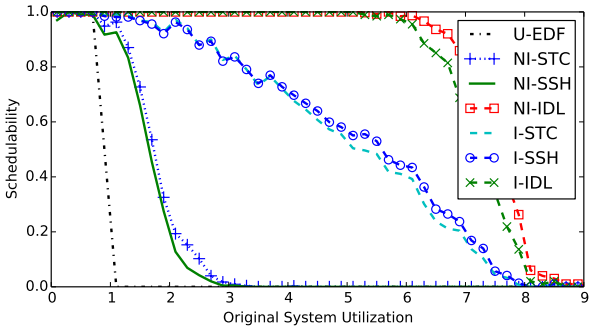
All-Mod., Long, Light, Heavy, Small, Heavy, Heavy



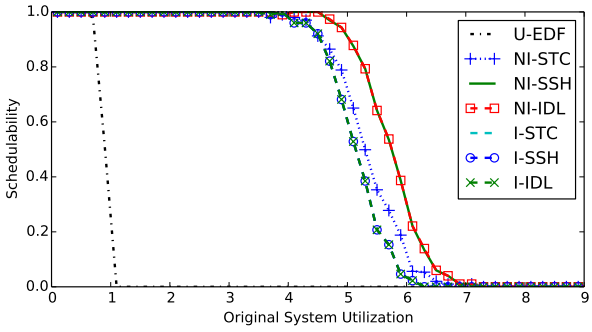
BC-Mod., Long, Heavy, Heavy, Small, Light, Heavy



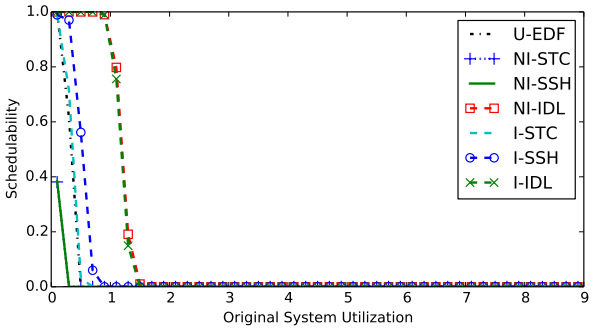
B-Heavy, Long, Heavy, Heavy, Large, Light, Heavy



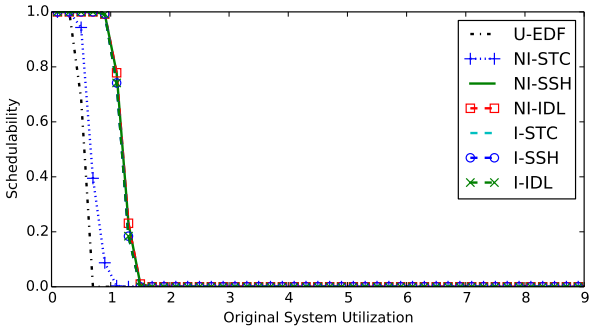
C-Heavy, Long, Heavy, Light, Small, Heavy, Light



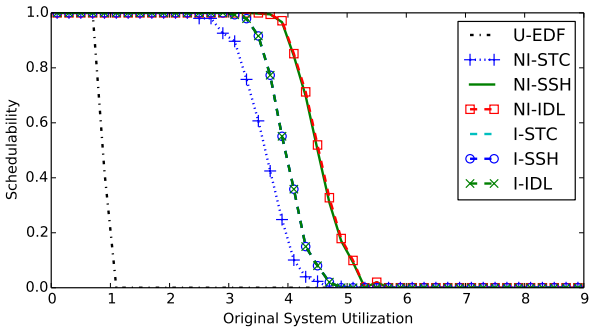
BC-Mod., Long, Heavy, Light, Large, Heavy, Heavy



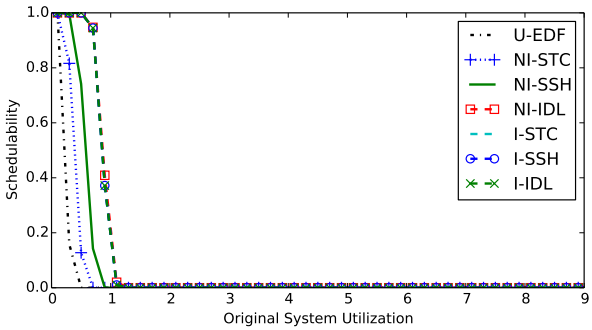
B-Heavy, Short, Light, Heavy, Small, Heavy, Heavy



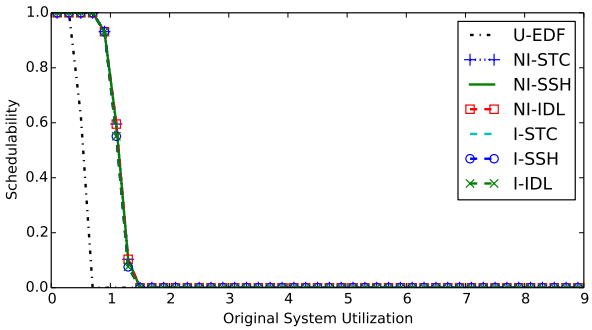
B-Heavy, Short, Light, Light, Large, Light, Heavy



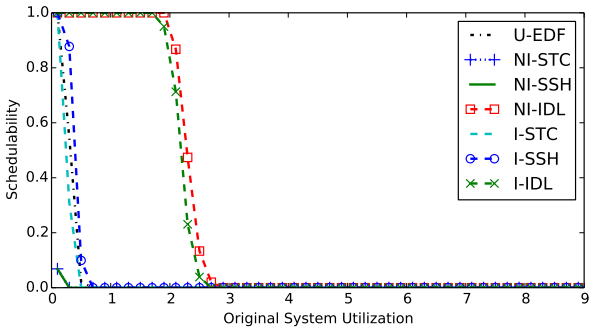
AB-Mod., Long, Heavy, Light, Large, Heavy, Heavy



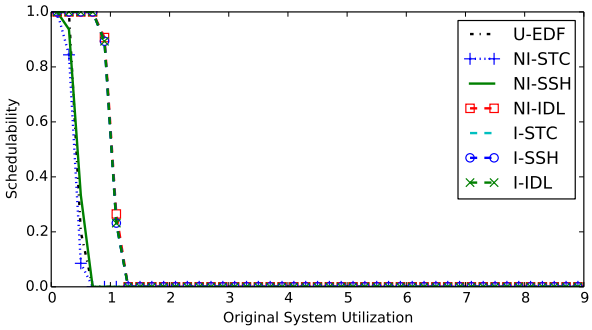
AC-Mod., Short, Light, Heavy, Large, Heavy, Heavy



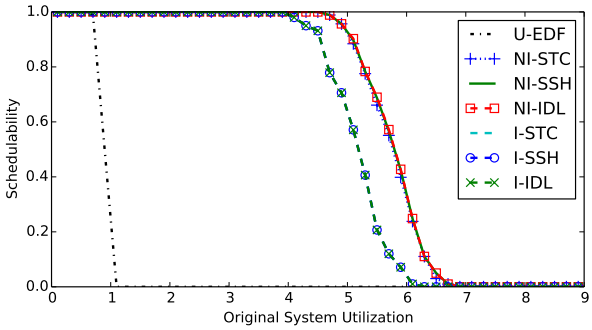
BC-Mod., Short, Light, Light, Mod., Light, Heavy



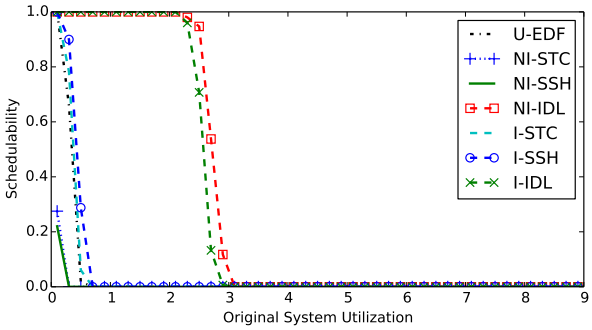
A-Heavy, Long, Light, Heavy, Small, Heavy, Heavy



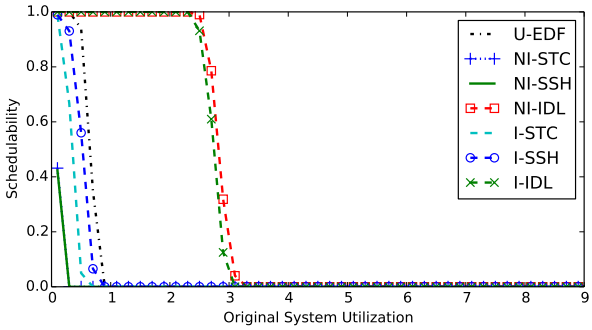
AB-Mod., Short, Light, Light, Large, Heavy, Light



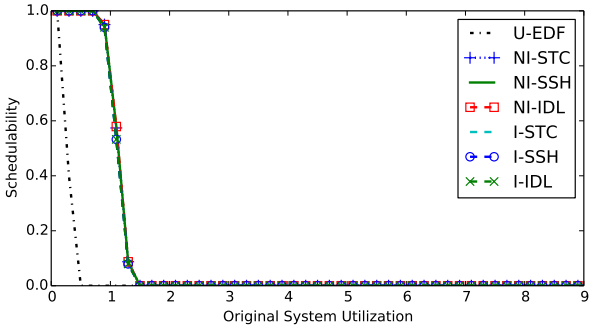
BC-Mod., Long, Heavy, Light, Large, Heavy, Light



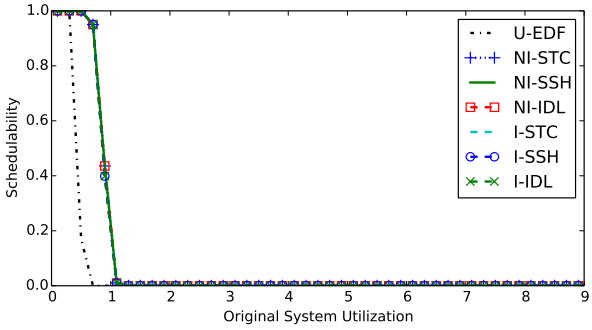
AB-Mod., Long, Light, Heavy, Small, Heavy, Light



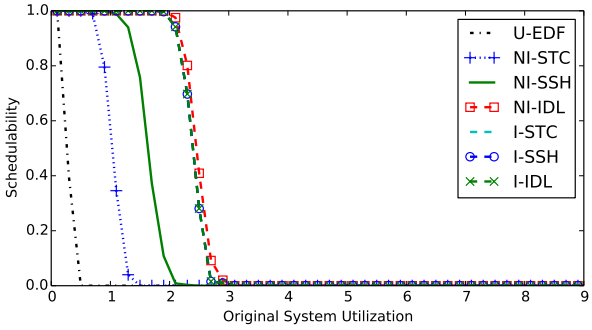
All-Mod., Long, Light, Light, Small, Heavy, Heavy



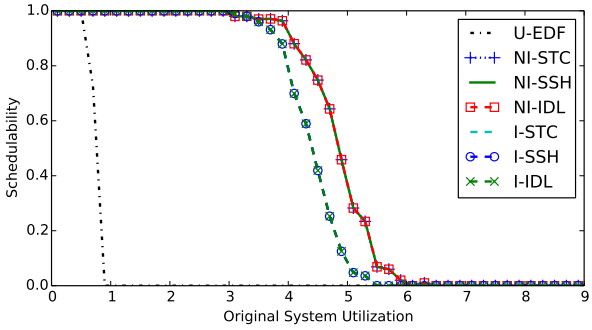
BC-Mod., Short, Light, Heavy, Mod., Heavy, Light



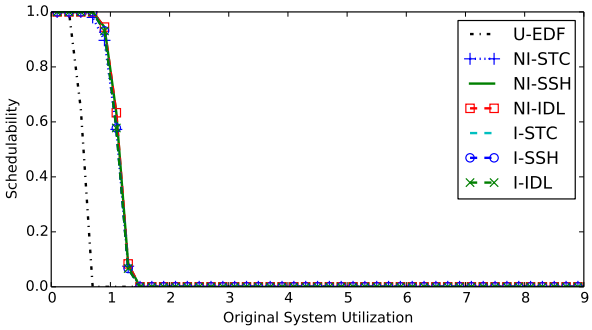
AC-Mod., Short, Light, Light, Mod., Light, Light



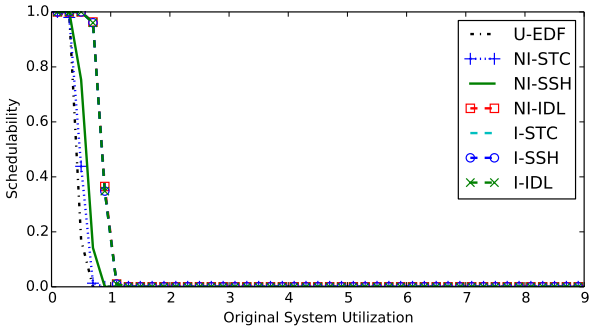
AC-Mod., Long, Light, Heavy, Mod., Heavy, Heavy



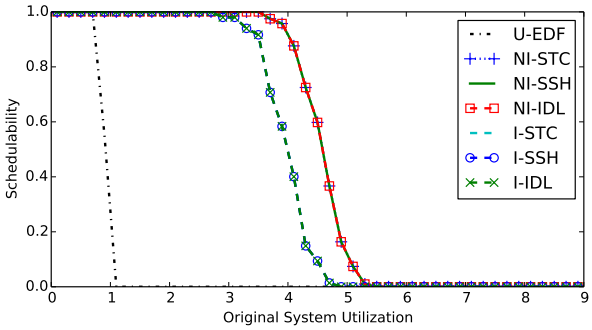
BC-Mod., Long, Heavy, Heavy, Mod., Light, Light



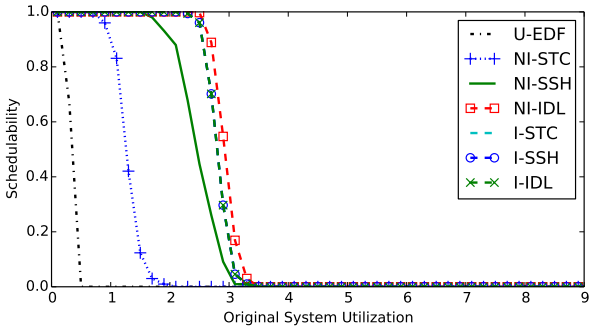
BC-Mod., Short, Light, Light, Mod., Heavy, Heavy



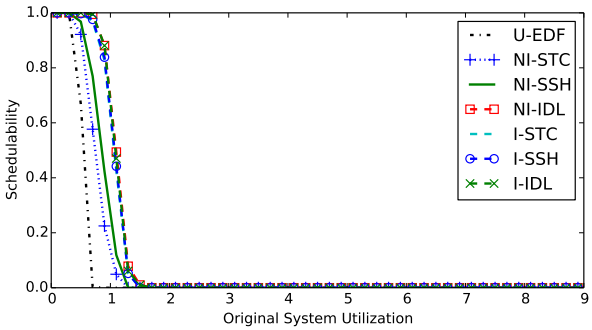
AC-Mod., Short, Light, Light, Large, Heavy, Light



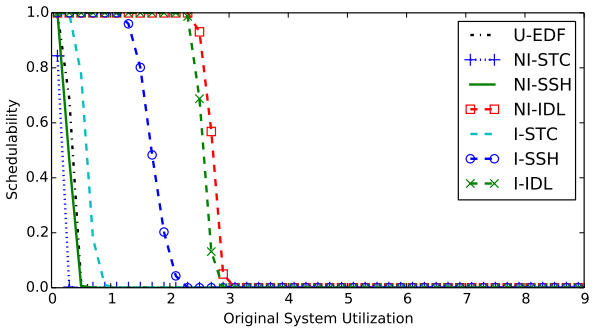
B-Heavy, Long, Heavy, Light, Mod., Light, Heavy



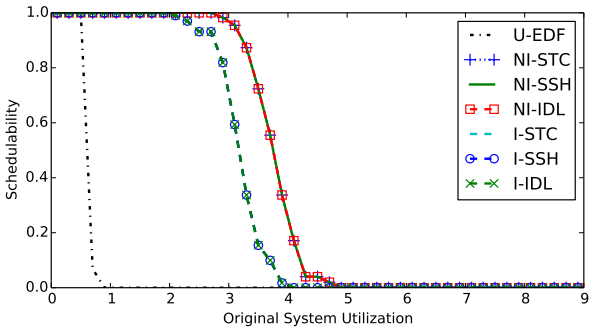
BC-Mod., Long, Light, Heavy, Large, Light, Light



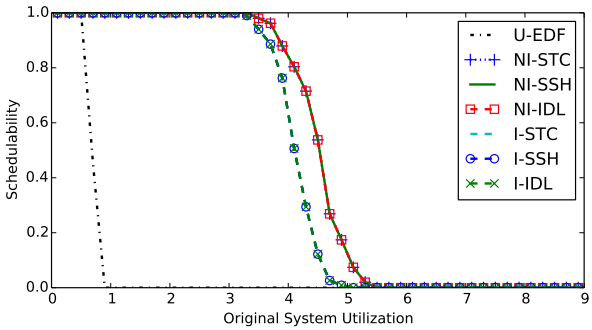
C-Heavy, Short, Light, Light, Large, Heavy, Light



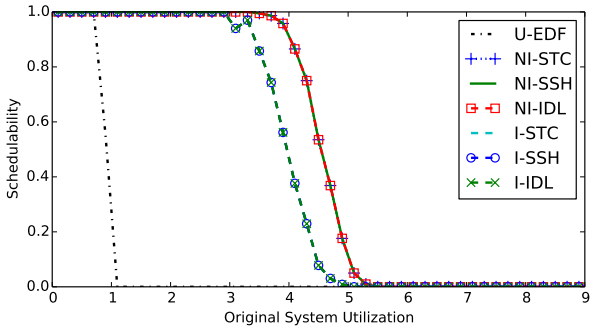
AB-Mod., Long, Light, Heavy, Small, Light, Heavy



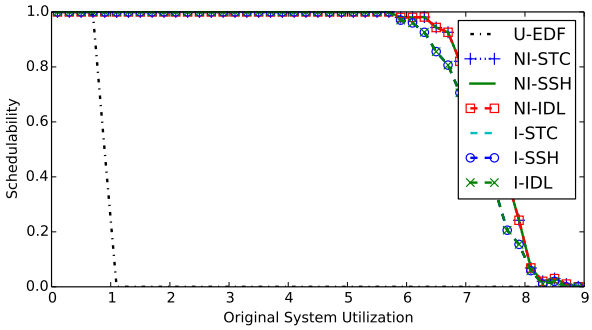
A-Heavy, Long, Heavy, Heavy, Large, Light, Heavy



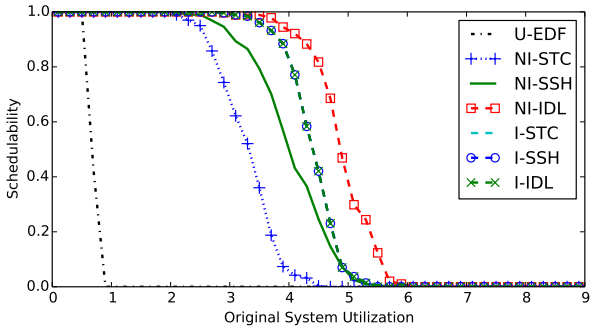
All-Mod., Long, Heavy, Heavy, Mod., Light, Light



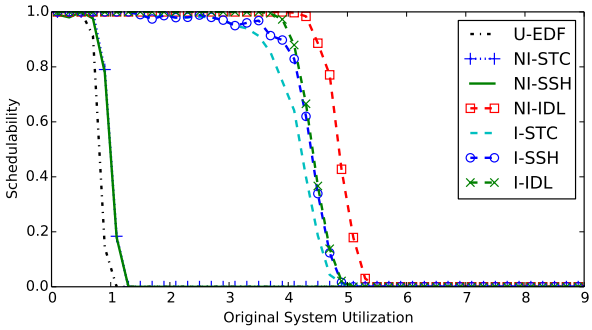
B-Heavy, Long, Heavy, Light, Large, Light, Heavy



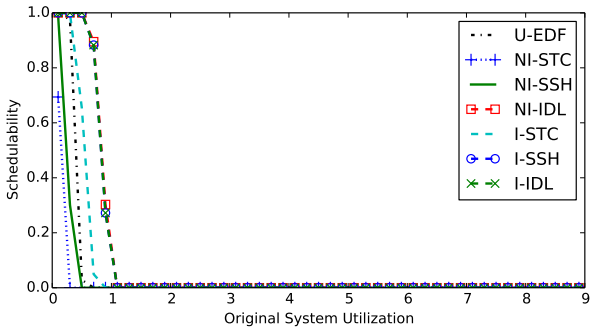
C-Heavy, Long, Heavy, Light, Mod., Heavy, Heavy



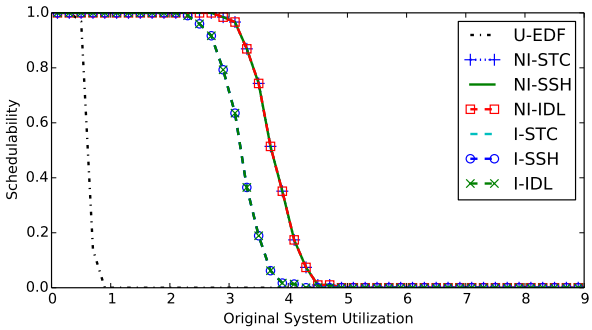
AC-Mod., Long, Heavy, Heavy, Small, Light, Light



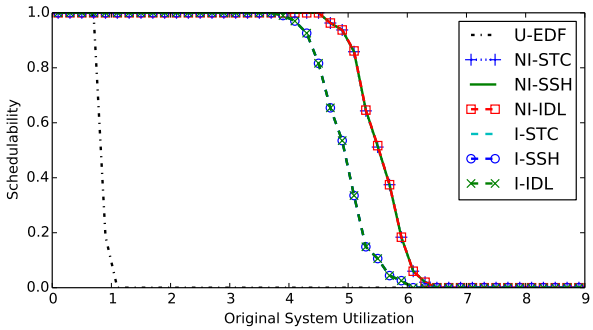
All-Mod., Short, Heavy, Light, Small, Heavy, Heavy



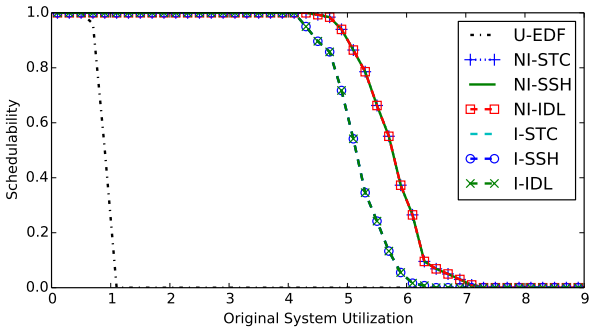
A-Heavy, Short, Light, Light, Small, Light, Heavy



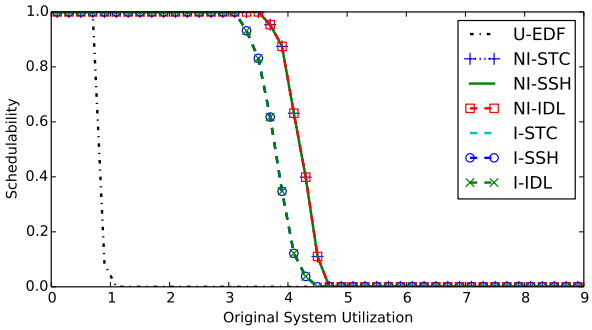
A-Heavy, Long, Heavy, Heavy, Large, Light, Light



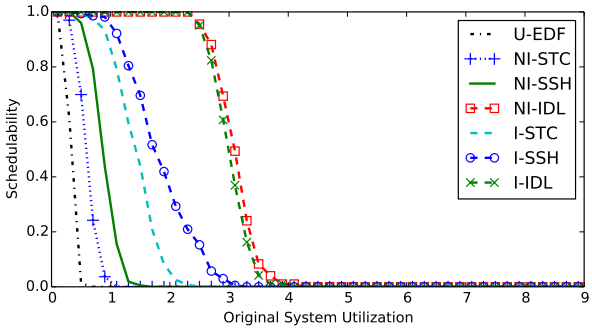
BC-Mod., Short, Heavy, Light, Large, Heavy, Light



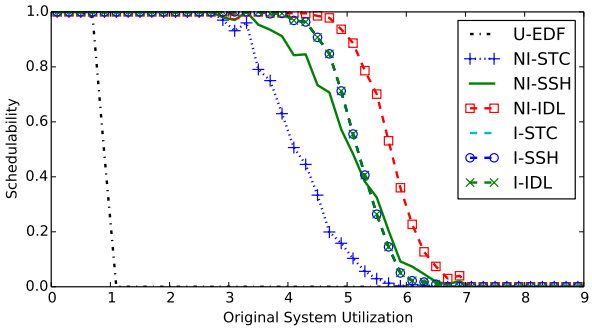
BC-Mod., Long, Heavy, Light, Mod., Heavy, Heavy



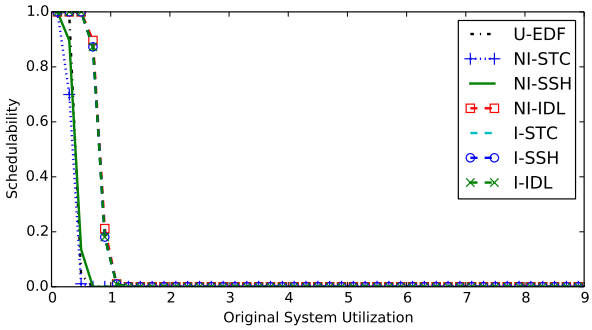
AB-Mod., Short, Heavy, Light, Mod., Light, Heavy



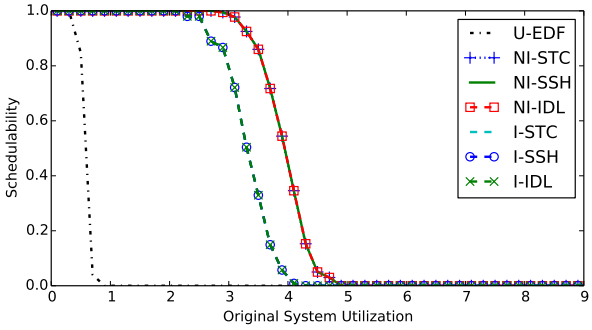
C-Heavy, Long, Light, Heavy, Large, Heavy, Heavy



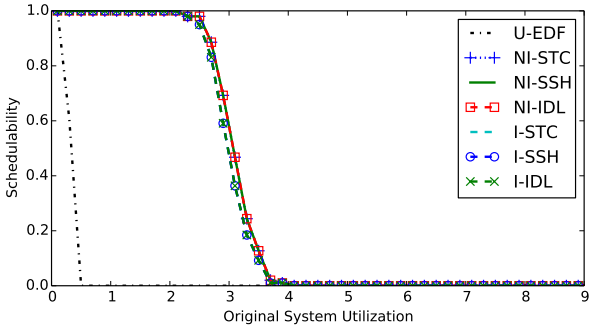
BC-Mod., Long, Heavy, Light, Small, Light, Light



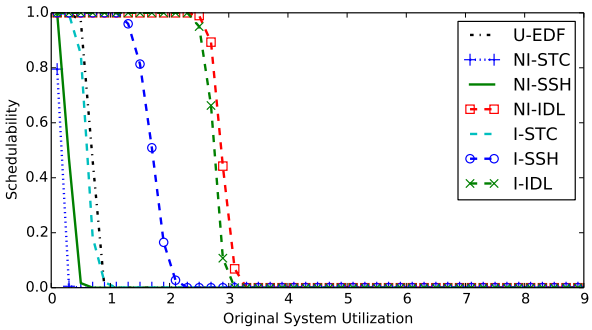
A-Heavy, Short, Light, Light, Large, Heavy, Light



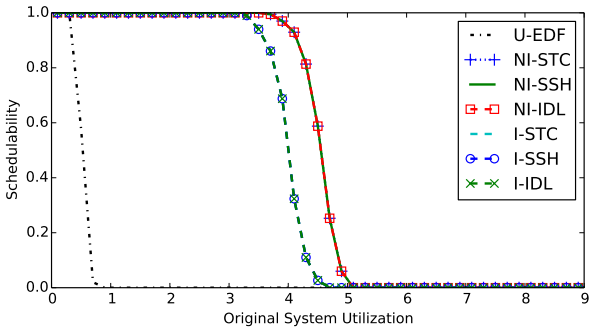
B-Heavy, Short, Heavy, Heavy, Mod., Light, Heavy



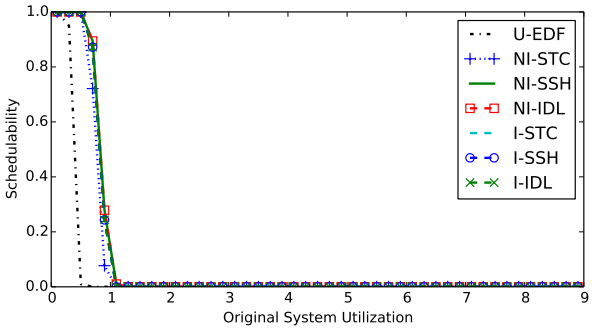
C-Heavy, Long, Light, Heavy, Mod., Light, Light



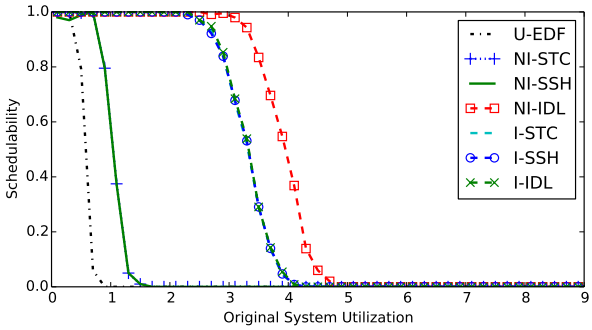
AB-Mod., Long, Light, Light, Small, Light, Heavy



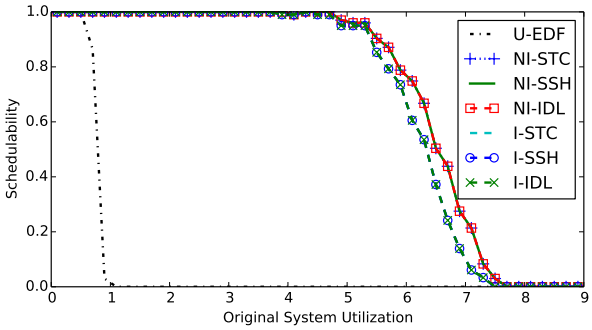
All-Mod., Short, Heavy, Heavy, Large, Light, Heavy



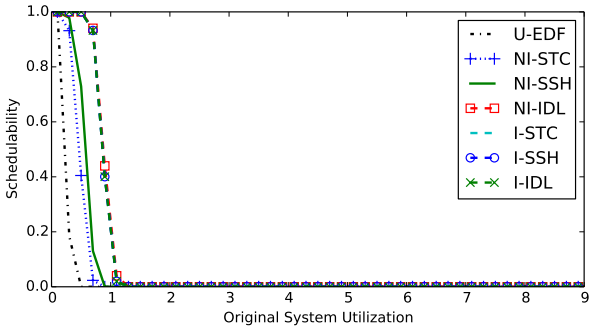
A-Heavy, Short, Light, Light, Large, Light, Light



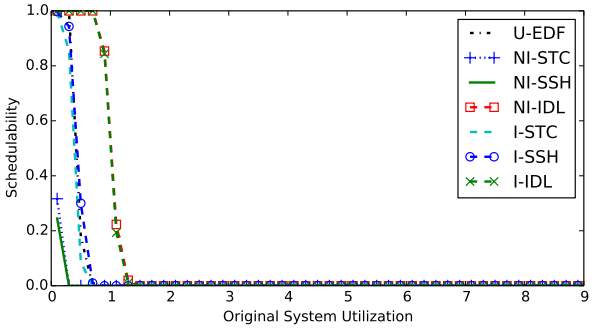
B-Heavy, Short, Heavy, Heavy, Small, Heavy, Heavy



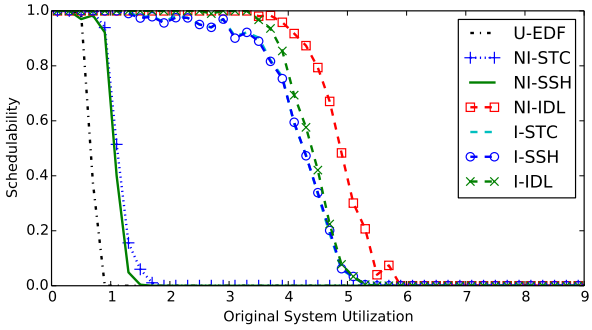
C-Heavy, Long, Heavy, Heavy, Mod., Heavy, Heavy



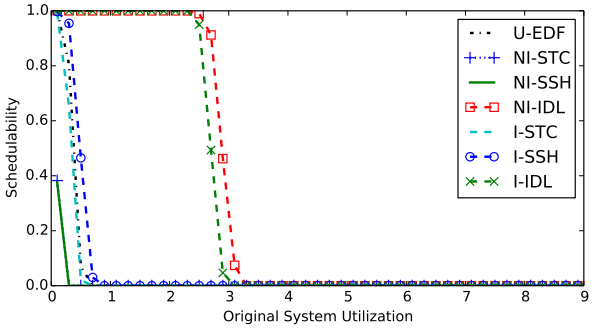
AC-Mod., Short, Light, Heavy, Large, Heavy, Light



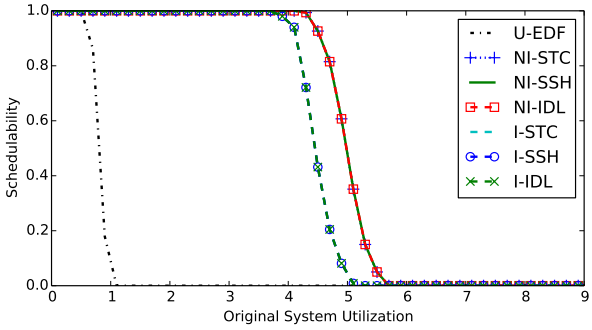
AB-Mod., Short, Light, Light, Small, Heavy, Light



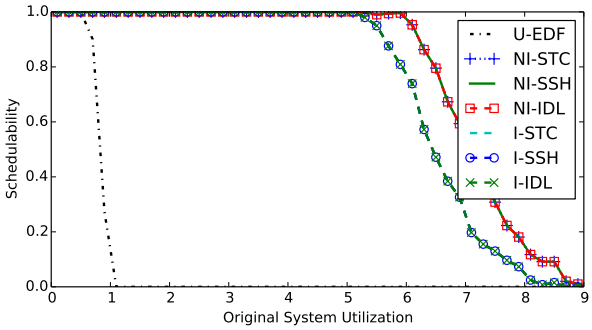
AC-Mod., Long, Heavy, Heavy, Small, Heavy, Light



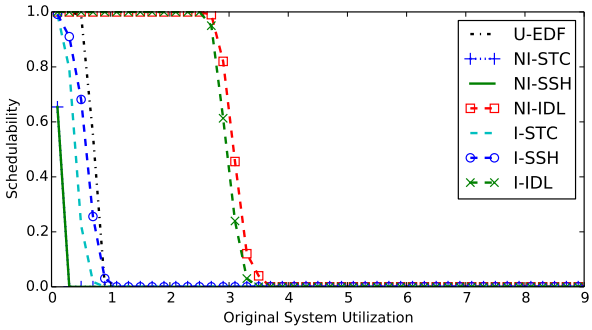
B-Heavy, Long, Light, Heavy, Small, Heavy, Heavy



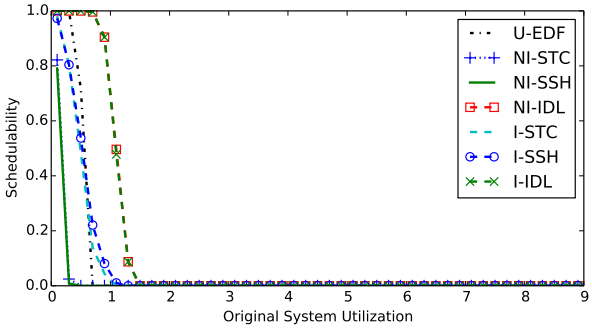
AC-Mod., Short, Heavy, Light, Large, Light, Heavy



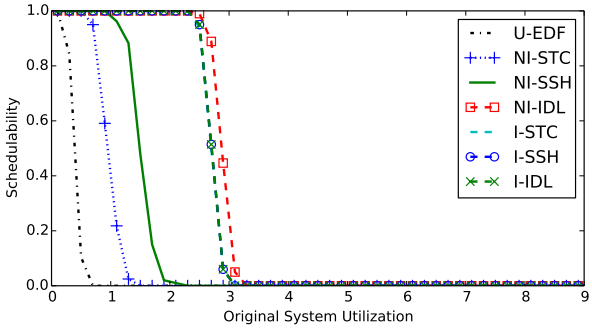
C-Heavy, Short, Heavy, Light, Mod., Light, Heavy



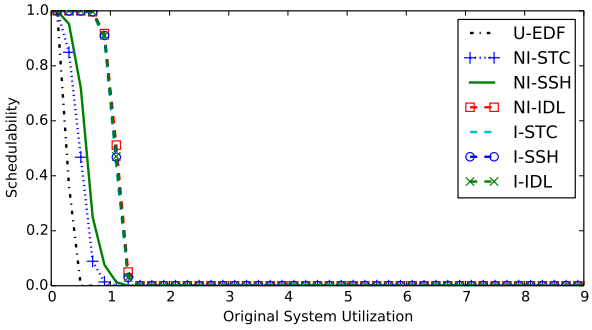
BC-Mod., Long, Light, Light, Small, Heavy, Heavy



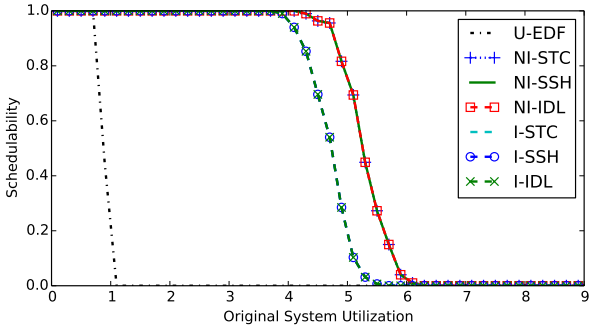
C-Heavy, Short, Light, Light, Small, Heavy, Light



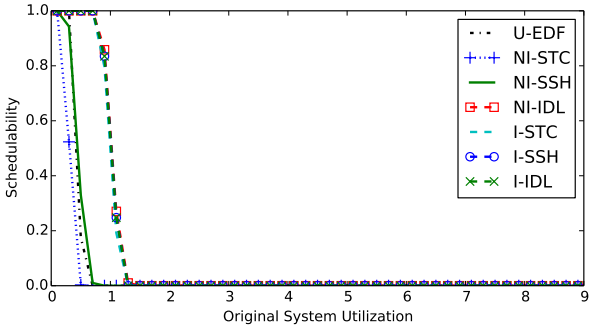
B-Heavy, Long, Light, Heavy, Mod., Heavy, Heavy



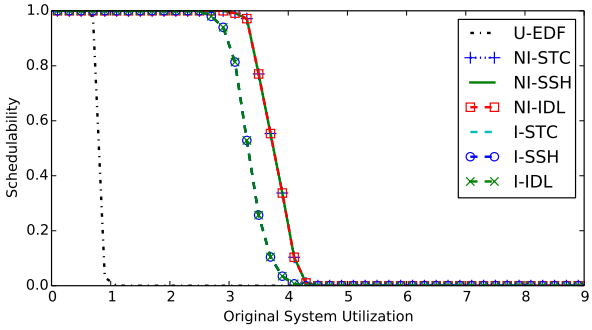
C-Heavy, Short, Light, Heavy, Small, Light, Light



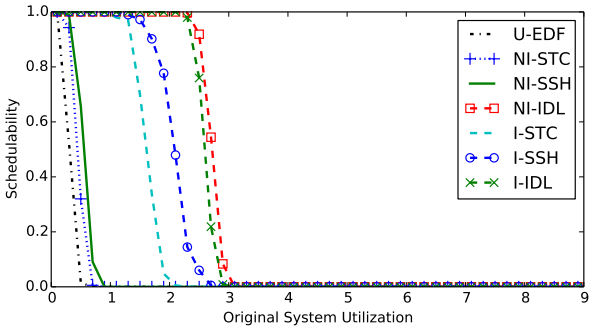
All-Mod., Long, Heavy, Light, Mod., Light, Heavy



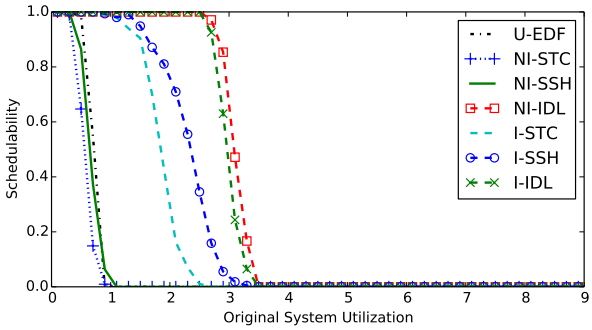
AB-Mod., Short, Light, Light, Large, Heavy, Heavy



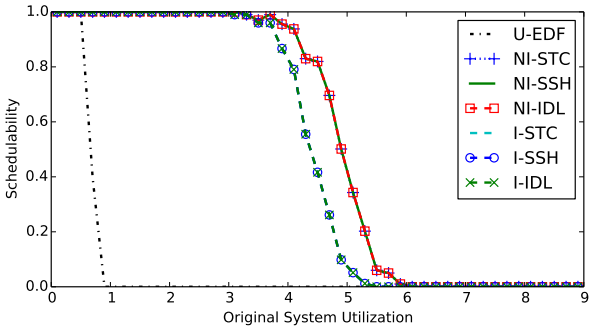
A-Heavy, Short, Heavy, Light, Mod., Light, Heavy



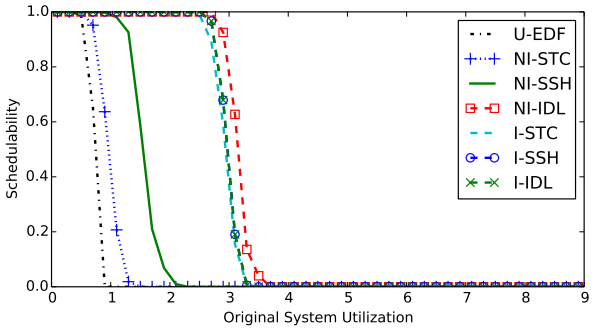
All-Mod., Long, Light, Heavy, Large, Heavy, Light



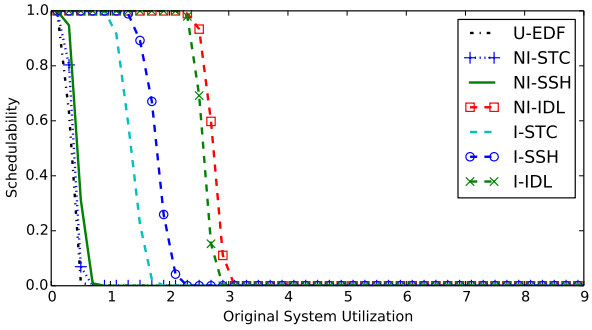
BC-Mod., Long, Light, Light, Large, Heavy, Light



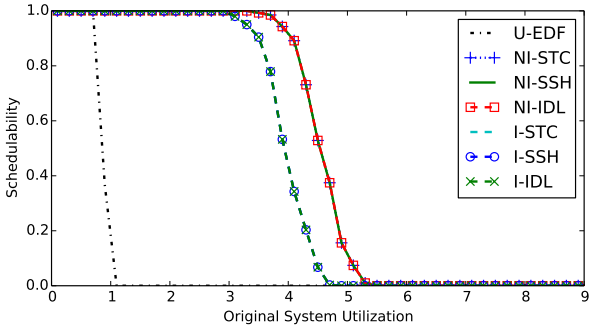
AC-Mod., Long, Heavy, Heavy, Mod., Light, Heavy



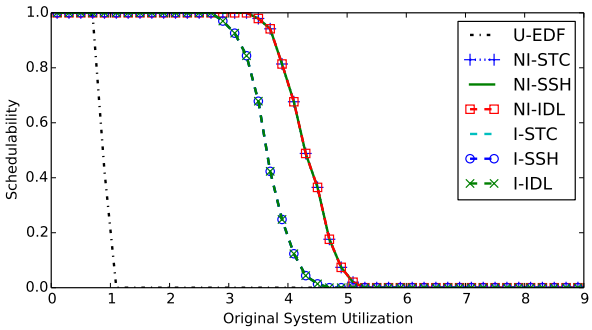
B-Heavy, Long, Light, Light, Mod., Heavy, Heavy



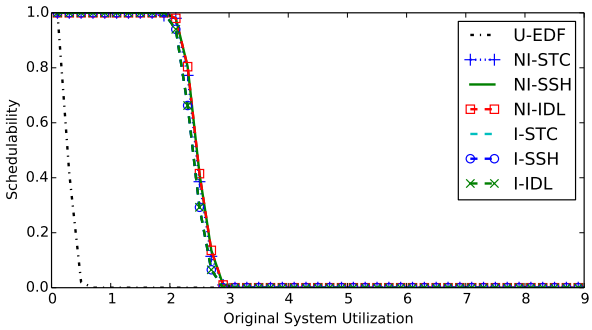
AB-Mod., Long, Light, Heavy, Large, Heavy, Light



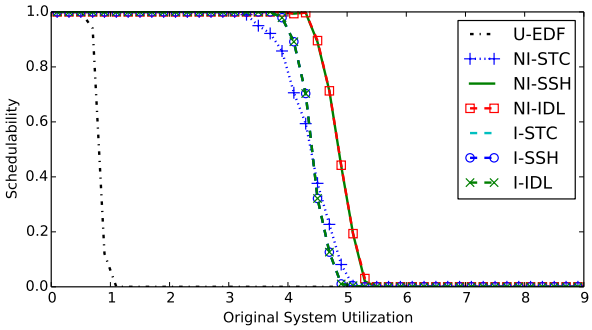
AB-Mod., Long, Heavy, Light, Mod., Light, Light



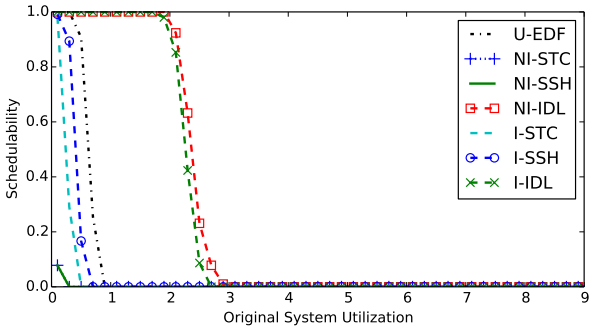
A-Heavy, Long, Heavy, Light, Large, Light, Light



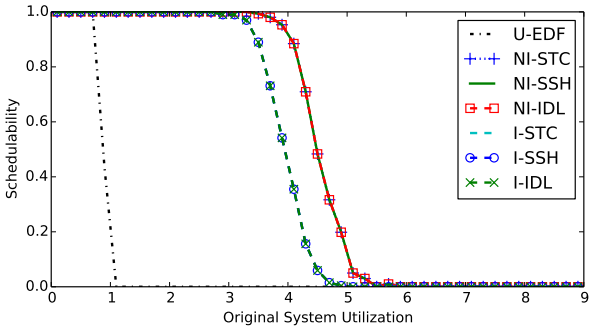
AC-Mod., Long, Light, Heavy, Mod., Light, Light



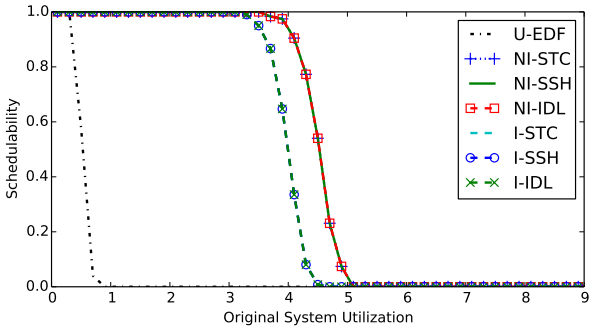
All-Mod., Short, Heavy, Light, Large, Heavy, Heavy



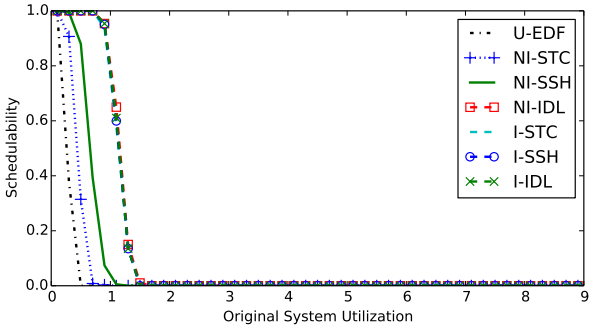
A-Heavy, Long, Light, Light, Small, Heavy, Heavy



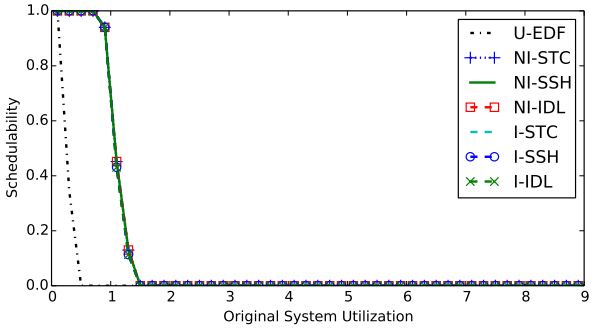
AB-Mod., Long, Heavy, Light, Mod., Heavy, Light



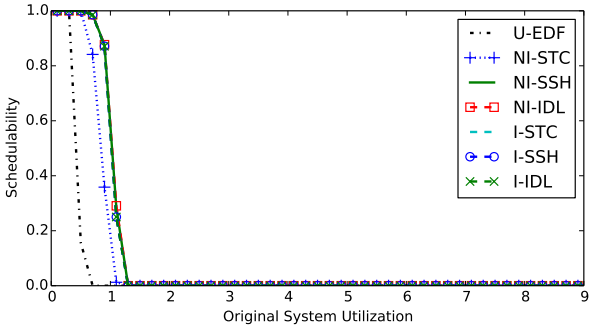
All-Mod., Short, Heavy, Heavy, Large, Light, Light



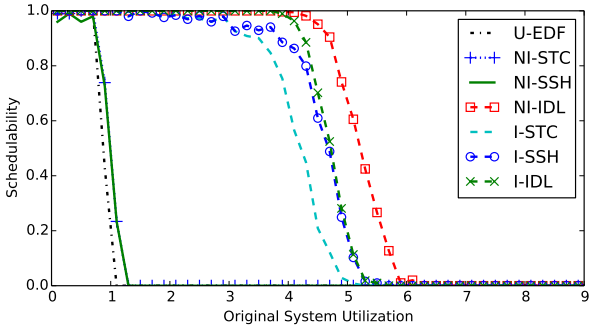
BC-Mod., Short, Light, Heavy, Large, Heavy, Heavy



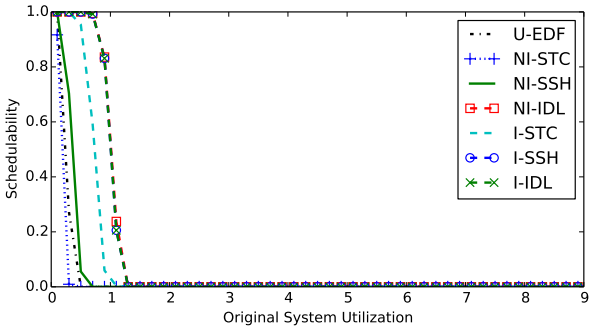
C-Heavy, Short, Light, Heavy, Mod., Heavy, Light



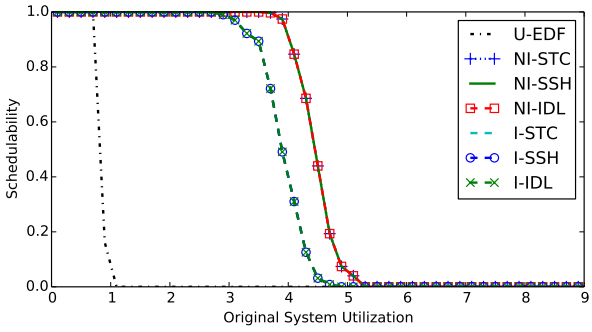
AB-Mod., Short, Light, Light, Mod., Heavy, Heavy



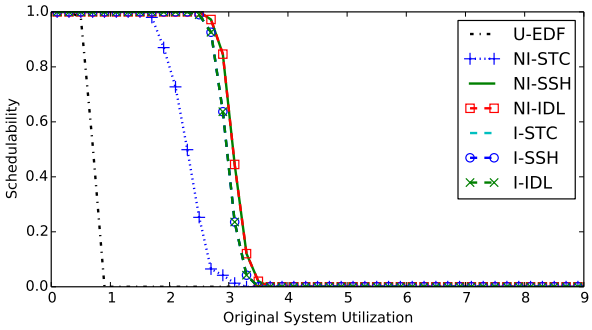
All-Mod., Long, Heavy, Light, Small, Heavy, Heavy



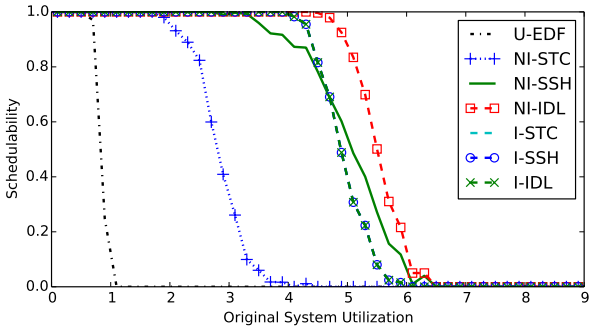
All-Mod., Short, Light, Heavy, Small, Light, Heavy



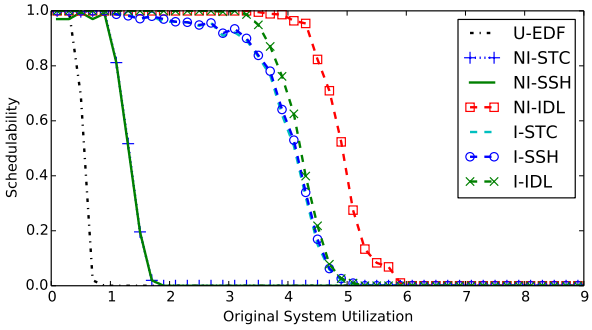
B-Heavy, Short, Heavy, Light, Mod., Light, Light



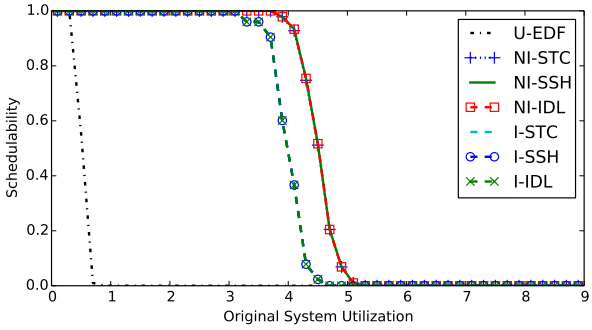
BC-Mod., Long, Light, Light, Mod., Light, Heavy



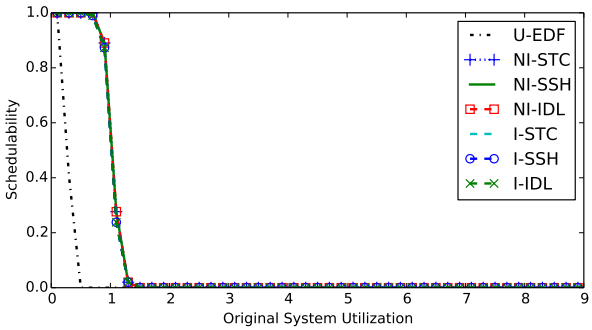
BC-Mod., Short, Heavy, Light, Small, Light, Heavy



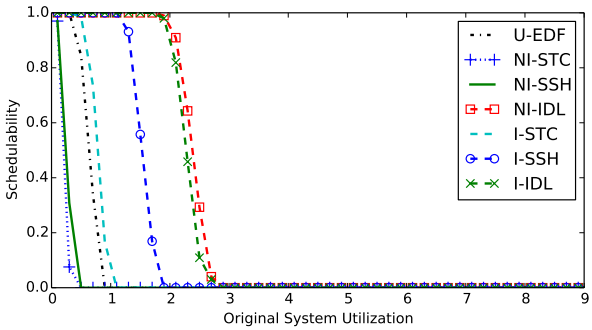
BC-Mod., Short, Heavy, Heavy, Small, Heavy, Heavy



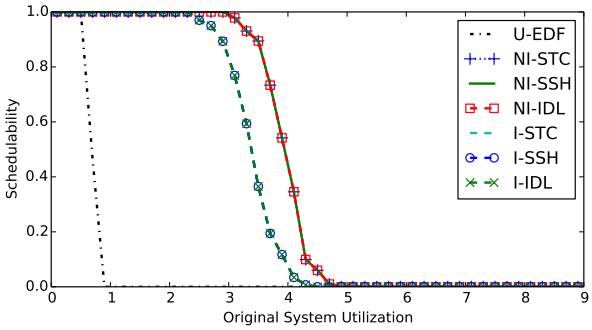
All-Mod., Short, Heavy, Heavy, Large, Heavy, Light



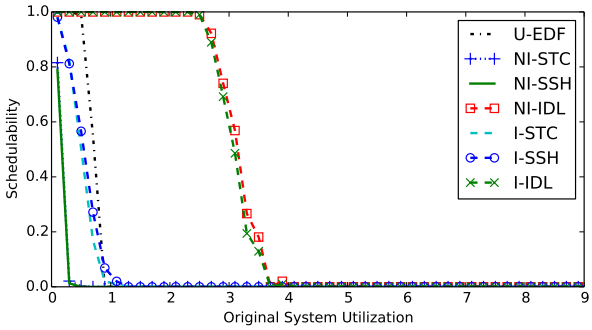
AB-Mod., Short, Light, Heavy, Mod., Light, Heavy



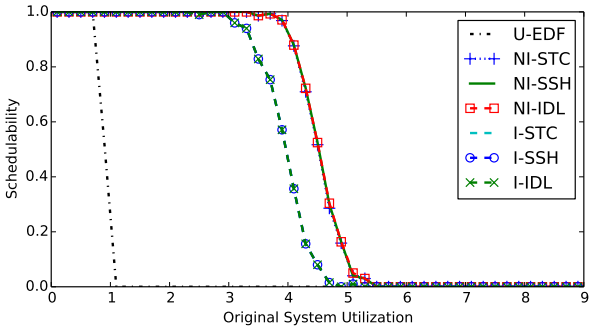
A-Heavy, Long, Light, Light, Small, Light, Light



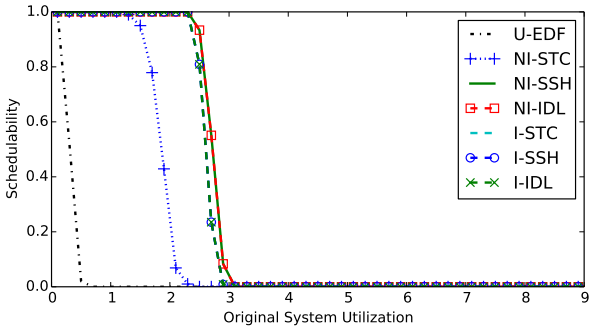
B-Heavy, Long, Heavy, Heavy, Mod., Heavy, Heavy



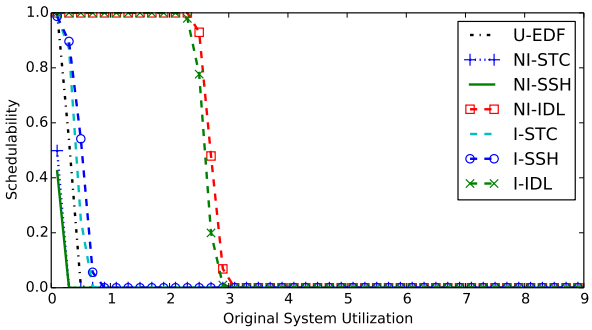
C-Heavy, Long, Light, Light, Small, Heavy, Light



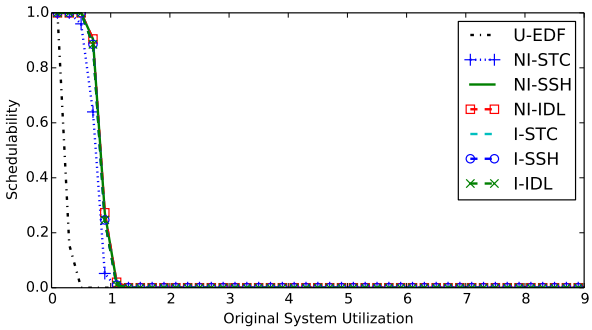
B-Heavy, Long, Heavy, Light, Large, Heavy, Light



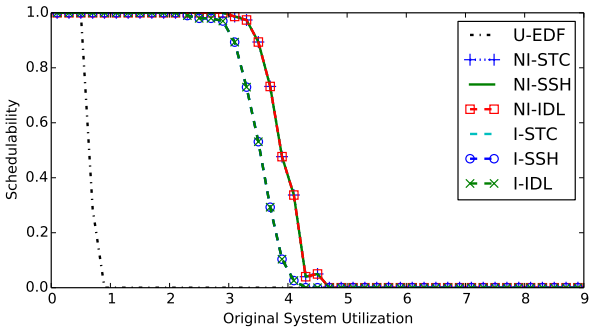
All-Mod., Long, Light, Heavy, Mod., Light, Heavy



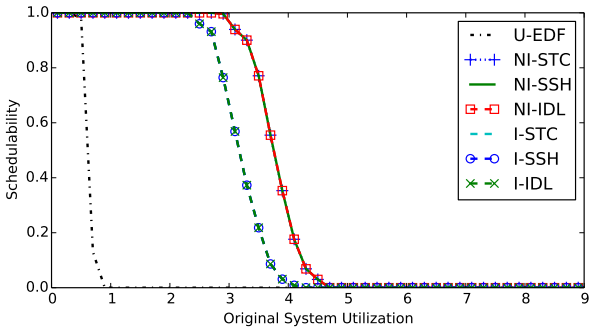
All-Mod., Long, Light, Heavy, Small, Heavy, Light



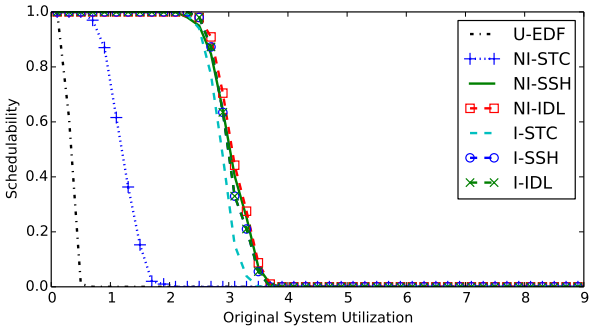
A-Heavy, Short, Light, Heavy, Mod., Heavy, Heavy



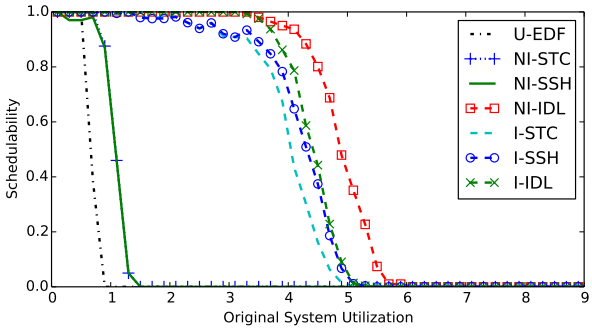
AB-Mod., Long, Heavy, Heavy, Large, Light, Light



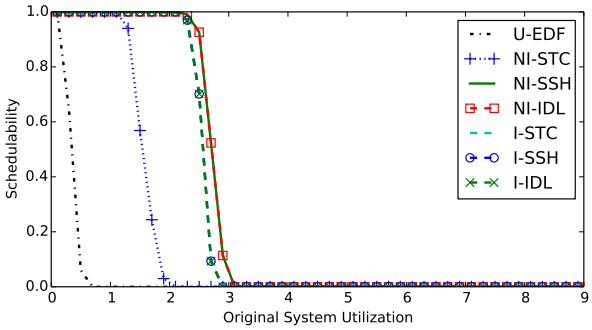
A-Heavy, Long, Heavy, Heavy, Mod., Light, Light



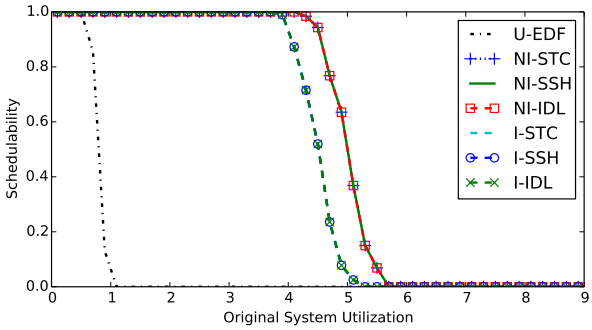
C-Heavy, Long, Light, Heavy, Large, Light, Heavy



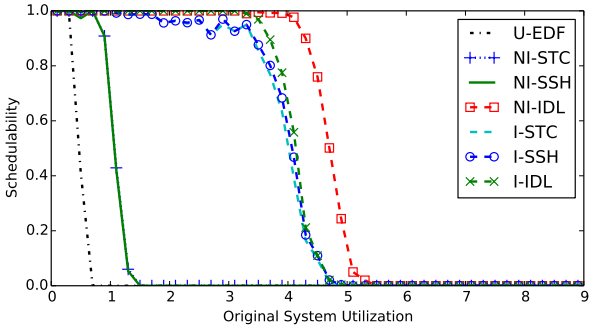
AC-Mod., Long, Heavy, Heavy, Small, Heavy, Heavy



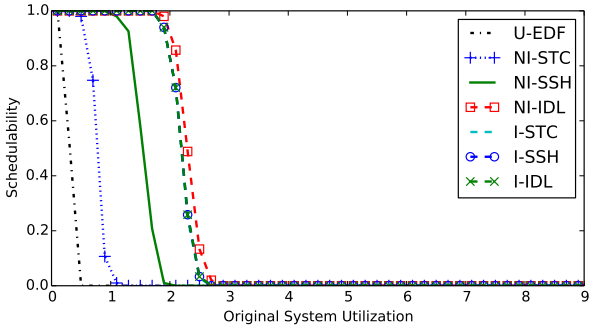
AB-Mod., Long, Light, Heavy, Mod., Light, Heavy



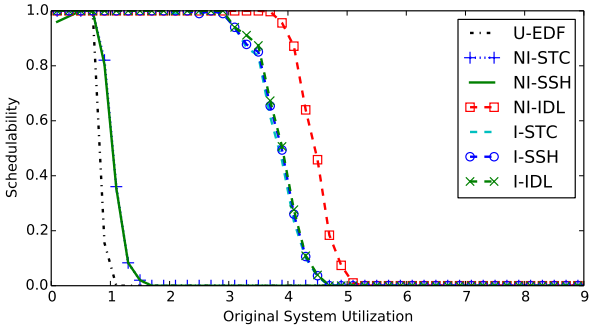
AC-Mod., Short, Heavy, Light, Mod., Light, Light



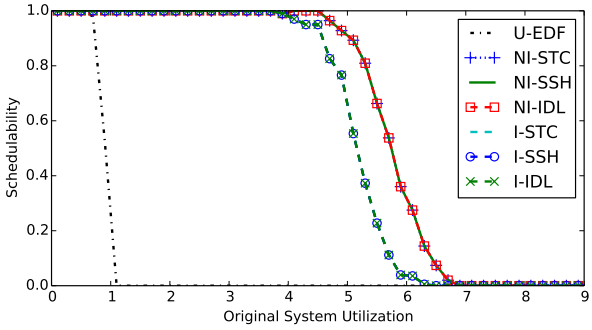
AC-Mod., Short, Heavy, Heavy, Small, Heavy, Heavy



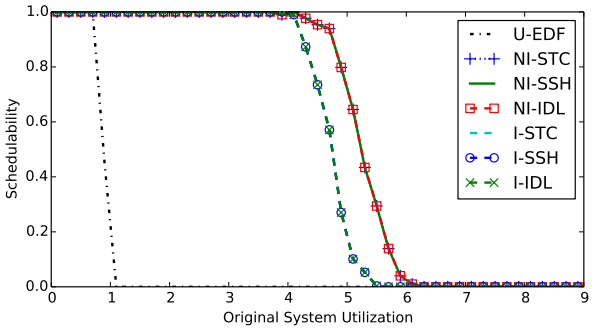
A-Heavy, Long, Light, Heavy, Large, Light, Light



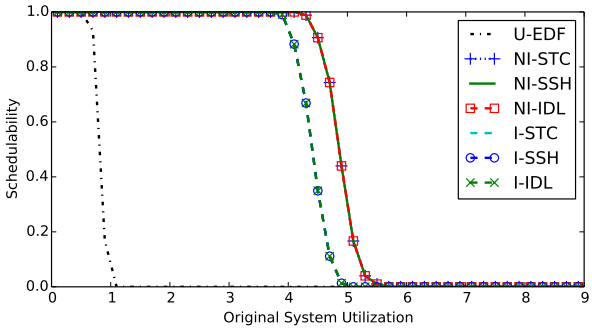
B-Heavy, Short, Heavy, Light, Small, Heavy, Heavy



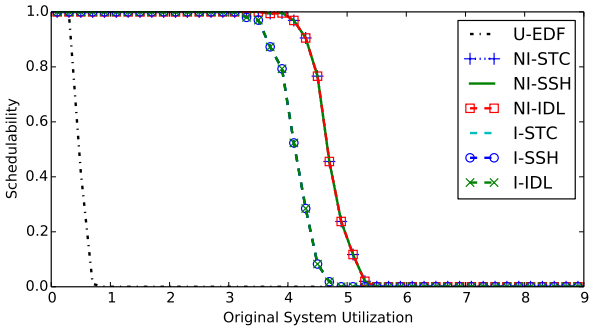
BC-Mod., Long, Heavy, Light, Large, Light, Light



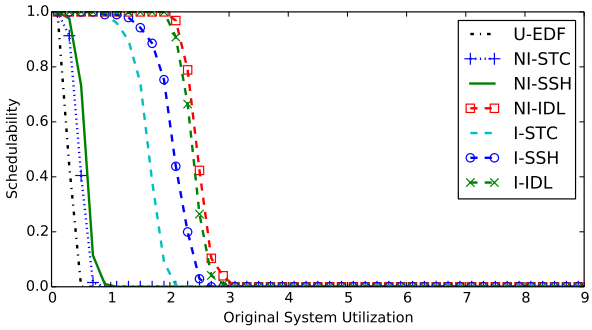
All-Mod., Long, Heavy, Light, Large, Light, Light



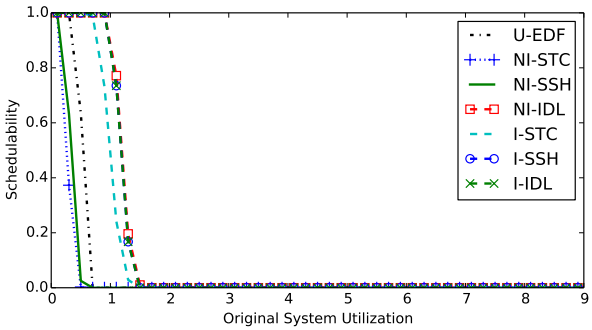
All-Mod., Short, Heavy, Light, Large, Light, Light



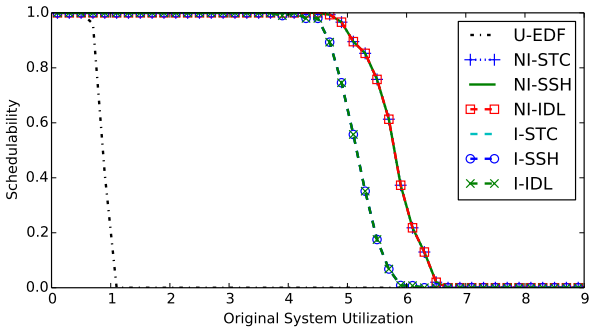
AC-Mod., Short, Heavy, Heavy, Mod., Heavy, Heavy



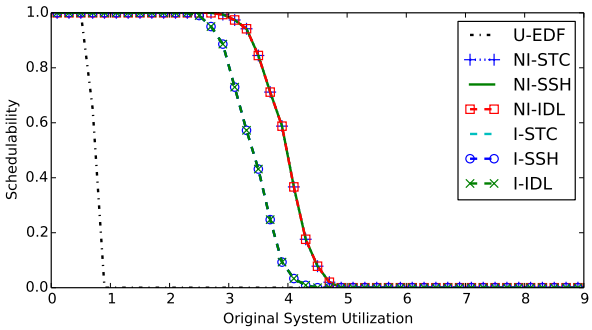
AC-Mod., Long, Light, Heavy, Large, Heavy, Light



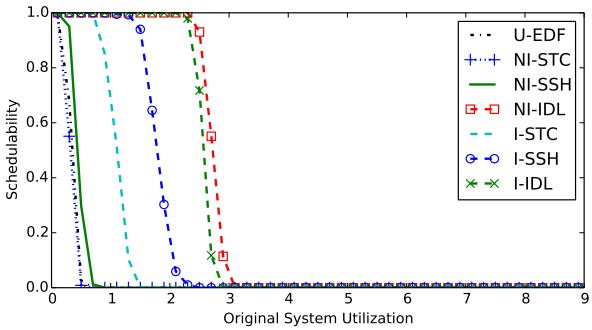
B-Heavy, Short, Light, Light, Small, Light, Light



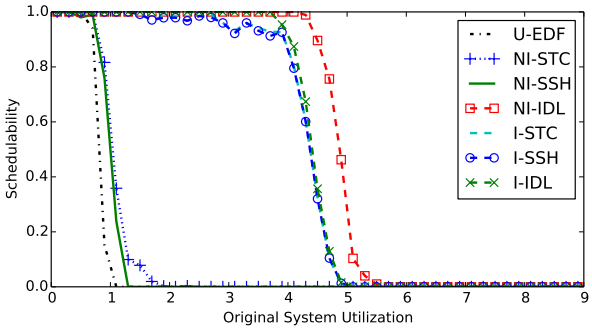
AC-Mod., Long, Heavy, Light, Mod., Light, Light



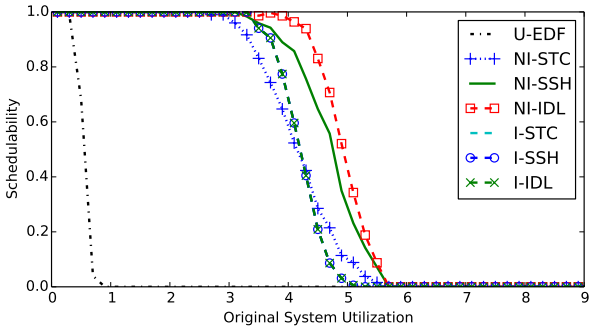
B-Heavy, Long, Heavy, Heavy, Mod., Light, Light



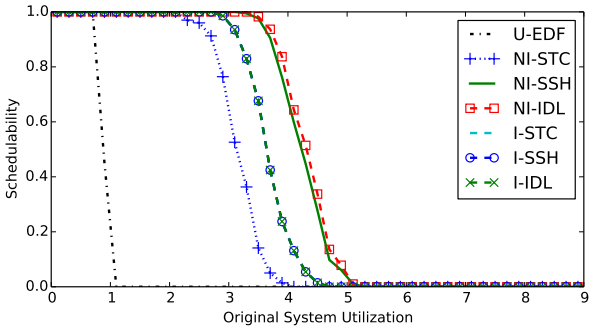
AB-Mod., Long, Light, Heavy, Large, Heavy, Heavy



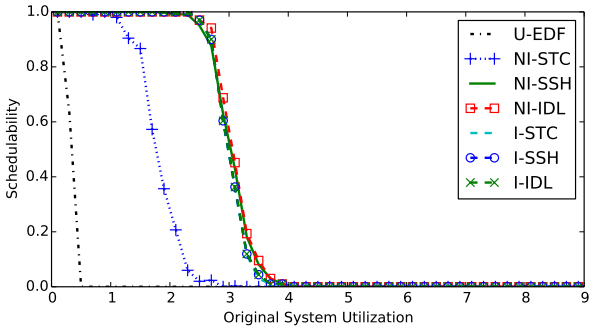
All-Mod., Short, Heavy, Light, Small, Heavy, Light



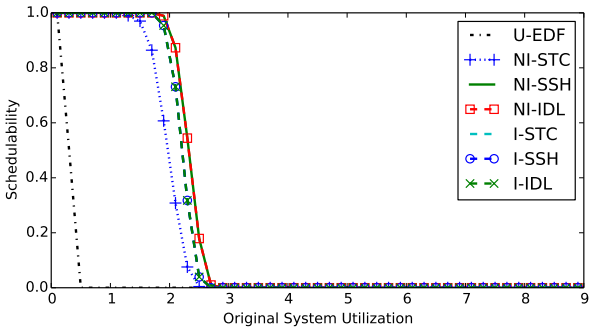
BC-Mod., Short, Heavy, Heavy, Small, Light, Light



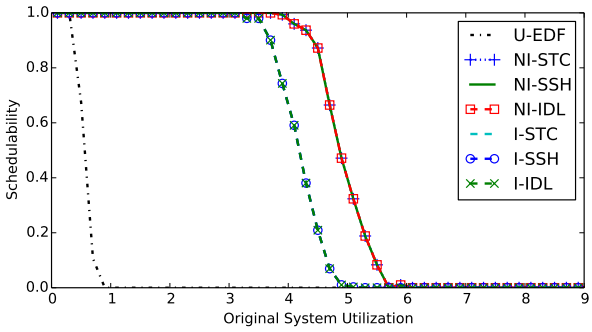
A-Heavy, Long, Heavy, Light, Large, Heavy, Heavy



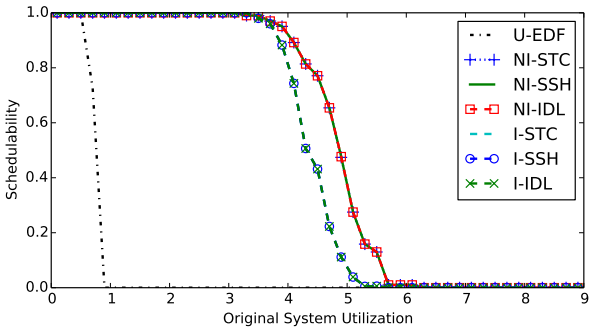
C-Heavy, Long, Light, Heavy, Large, Light, Light



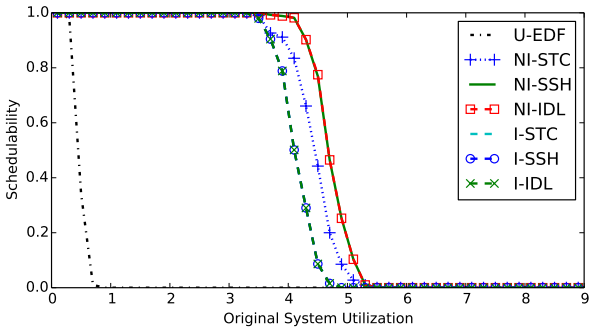
A-Heavy, Long, Light, Heavy, Mod., Light, Light



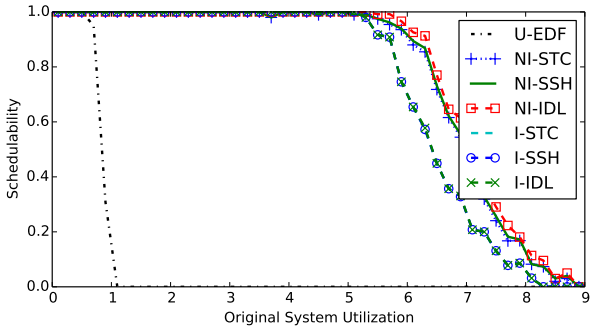
BC-Mod., Short, Heavy, Heavy, Large, Light, Heavy



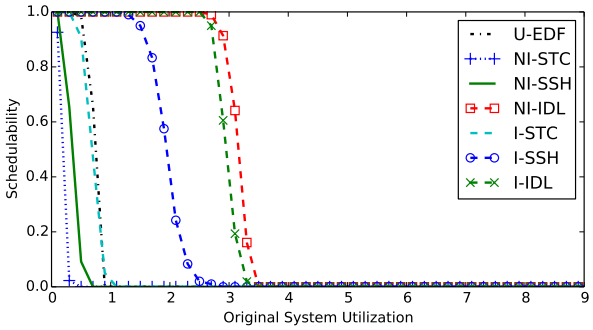
BC-Mod., Long, Heavy, Heavy, Large, Heavy, Light



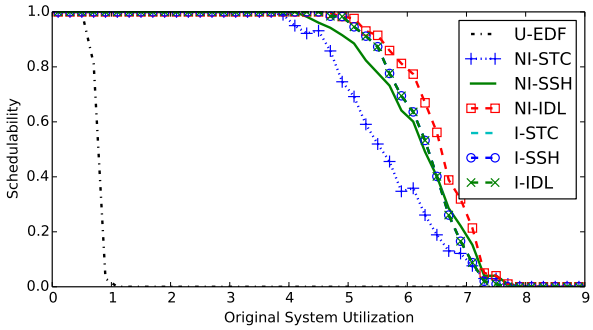
AC-Mod., Short, Heavy, Heavy, Large, Heavy, Heavy



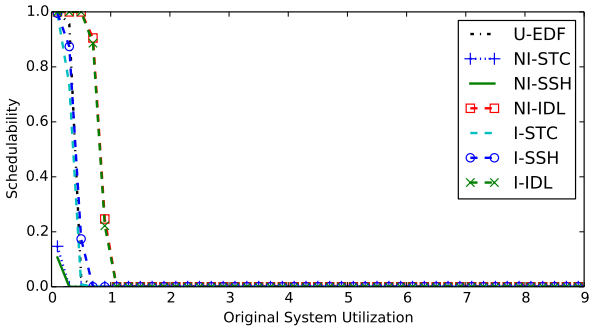
C-Heavy, Short, Heavy, Light, Large, Heavy, Light



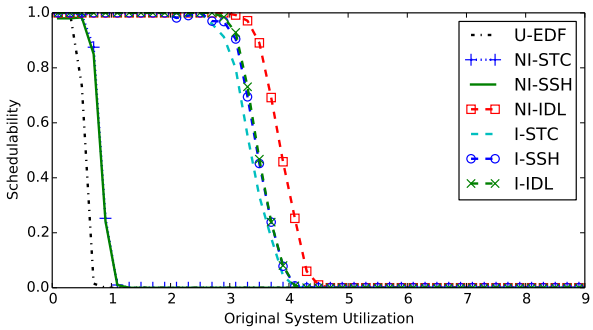
B-Heavy, Long, Light, Light, Small, Light, Heavy



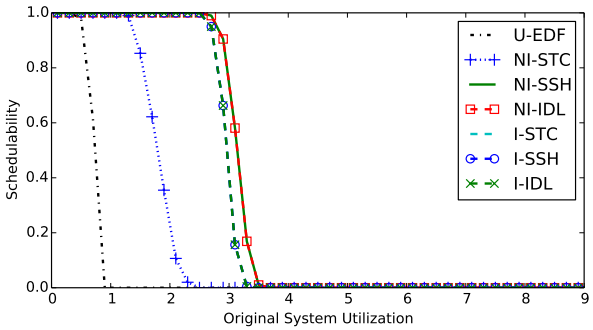
C-Heavy, Long, Heavy, Heavy, Small, Light, Light



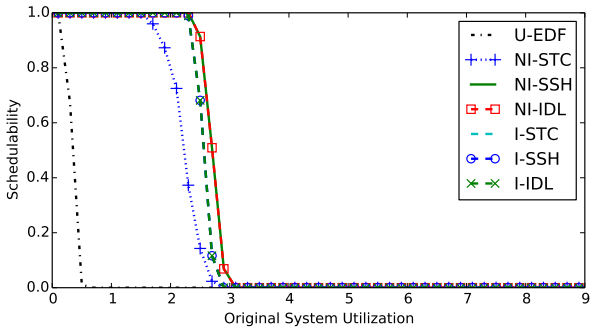
A-Heavy, Short, Light, Light, Small, Heavy, Light



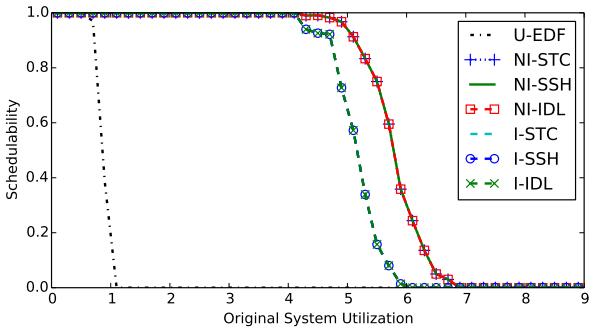
AB-Mod., Short, Heavy, Heavy, Small, Heavy, Heavy



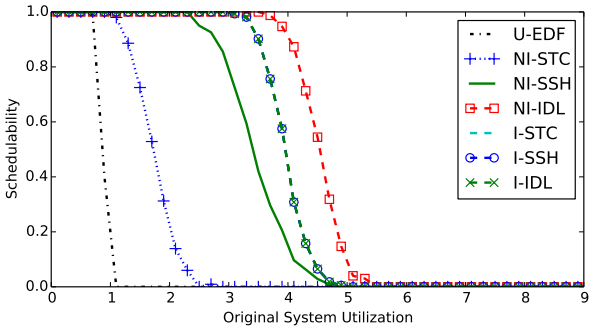
B-Heavy, Long, Light, Light, Mod., Light, Heavy



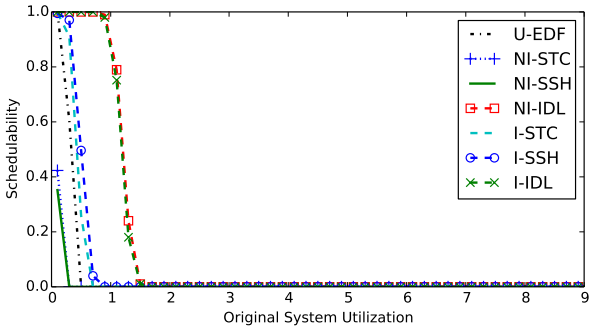
AB-Mod., Long, Light, Heavy, Mod., Light, Light



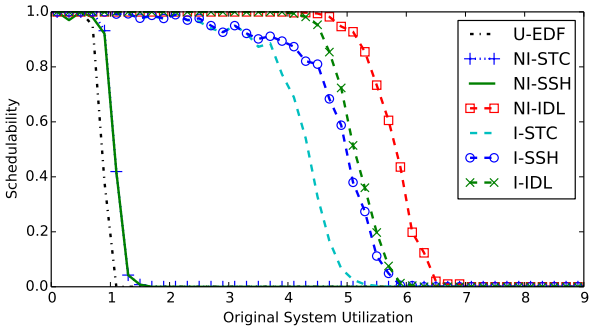
AC-Mod., Long, Heavy, Light, Large, Light, Light



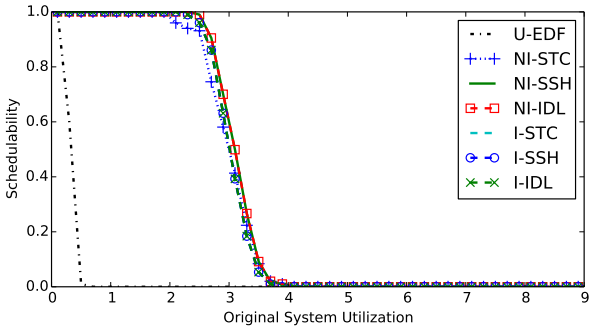
AB-Mod., Long, Heavy, Light, Small, Light, Heavy



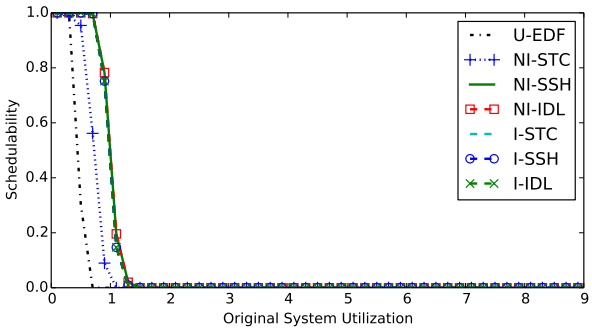
B-Heavy, Short, Light, Heavy, Small, Heavy, Light



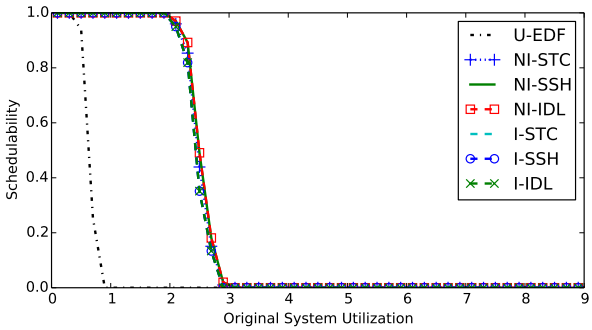
AC-Mod., Long, Heavy, Light, Small, Heavy, Heavy



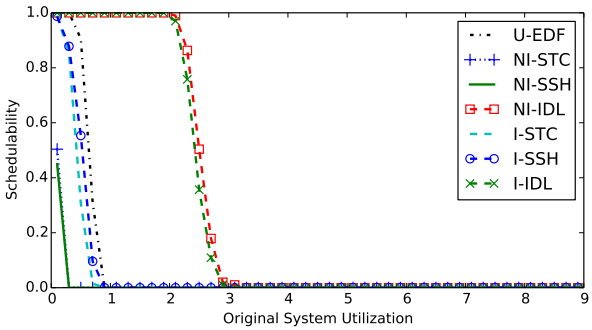
C-Heavy, Long, Light, Heavy, Mod., Light, Heavy



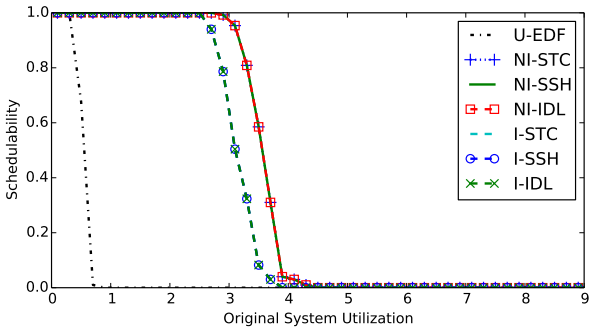
All-Mod., Short, Light, Light, Large, Light, Heavy



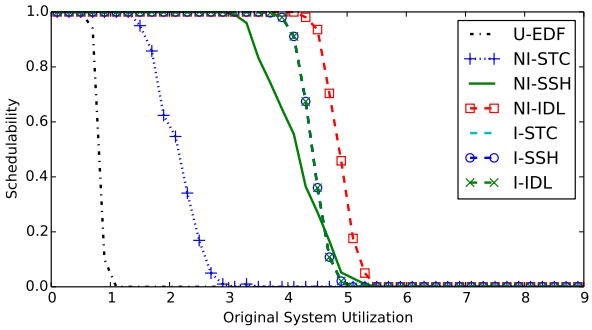
AC-Mod., Long, Light, Light, Mod., Light, Light



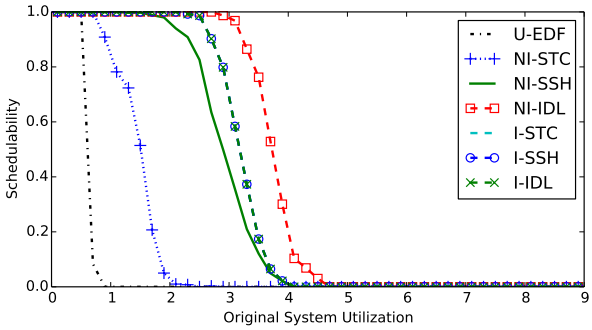
AC-Mod., Long, Light, Light, Small, Heavy, Light



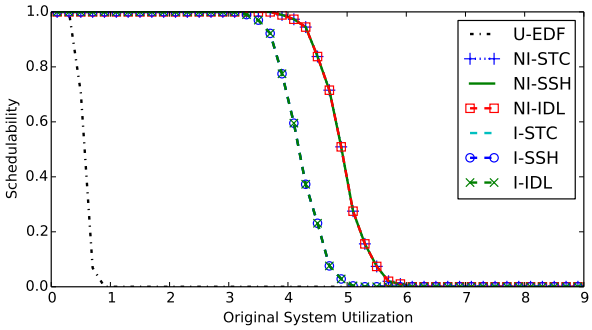
A-Heavy, Short, Heavy, Heavy, Mod., Light, Heavy



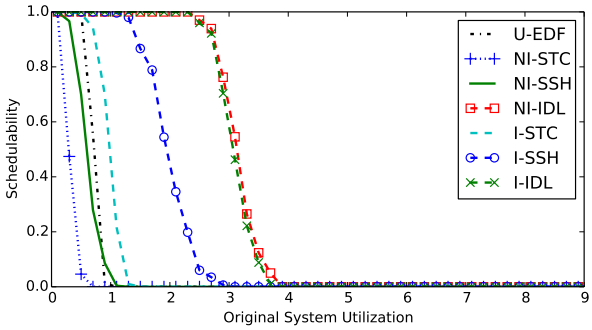
All-Mod., Short, Heavy, Light, Small, Light, Heavy



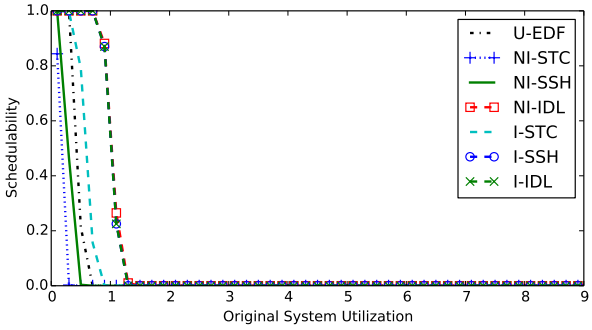
A-Heavy, Long, Heavy, Heavy, Small, Light, Heavy



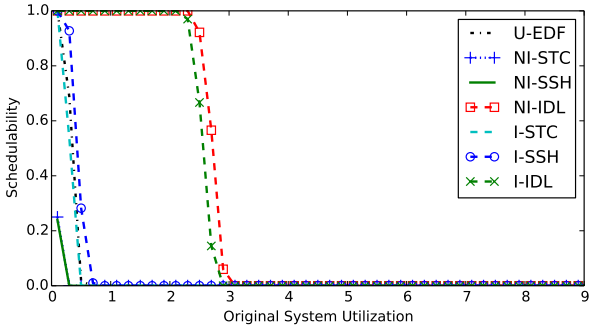
BC-Mod., Short, Heavy, Heavy, Mod., Light, Heavy



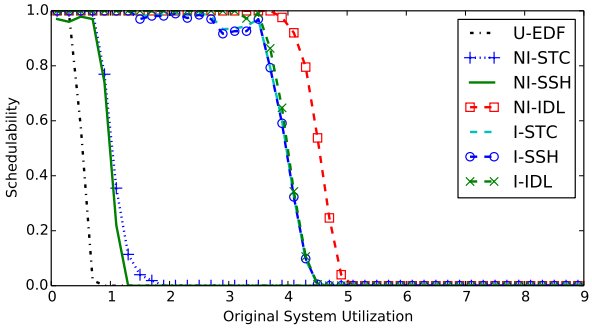
C-Heavy, Long, Light, Light, Small, Light, Heavy



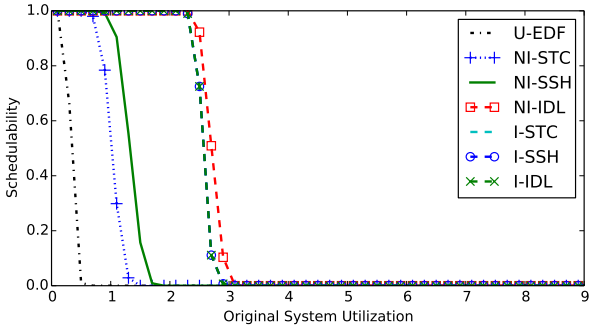
AB-Mod., Short, Light, Light, Small, Light, Heavy



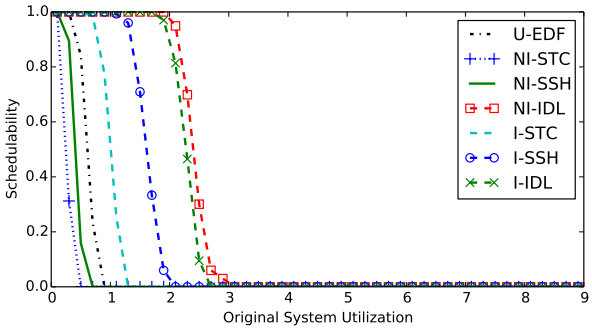
AB-Mod., Long, Light, Heavy, Small, Heavy, Heavy



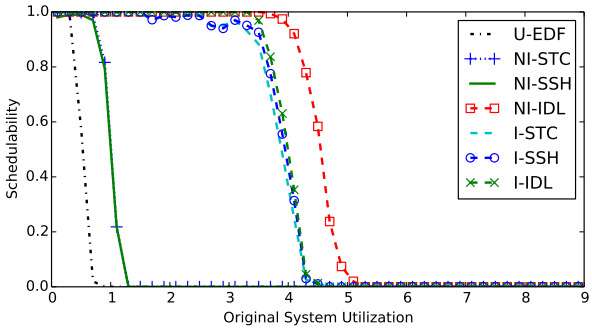
All-Mod., Short, Heavy, Heavy, Small, Heavy, Light



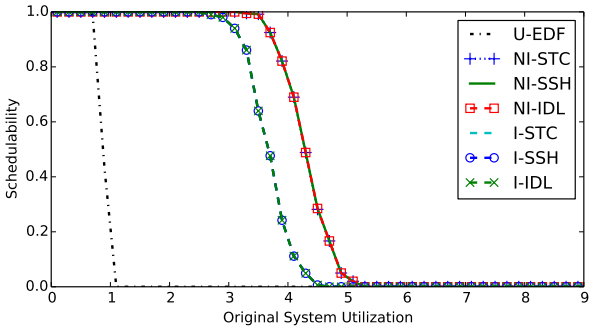
AB-Mod., Long, Light, Heavy, Mod., Heavy, Light



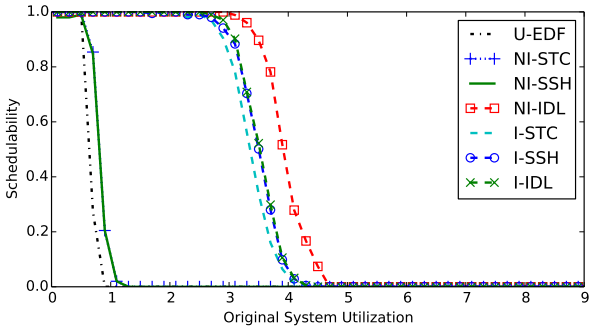
A-Heavy, Long, Light, Light, Large, Heavy, Heavy



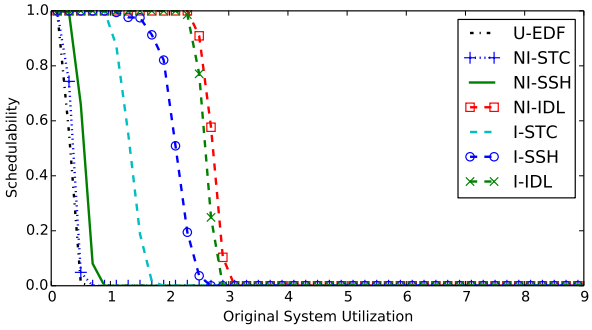
All-Mod., Short, Heavy, Heavy, Small, Heavy, Heavy



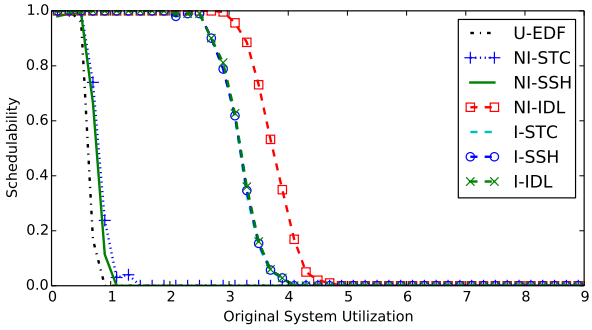
A-Heavy, Long, Heavy, Light, Large, Light, Heavy



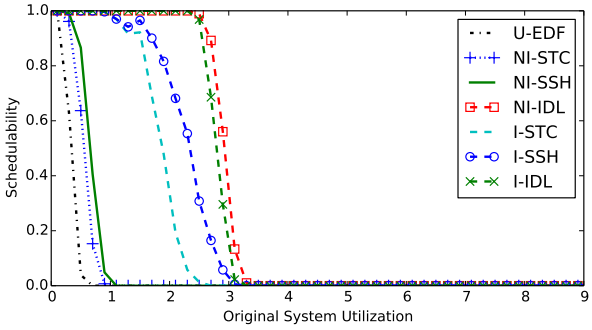
AB-Mod., Long, Heavy, Heavy, Small, Heavy, Heavy



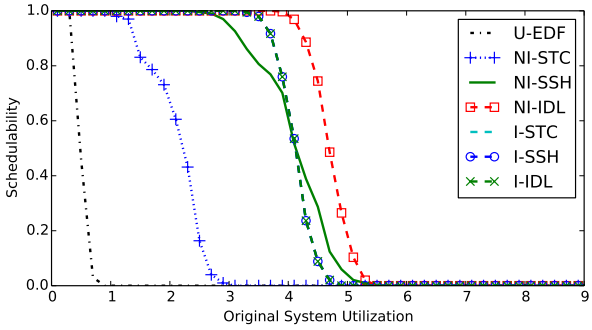
All-Mod., Long, Light, Heavy, Large, Heavy, Heavy



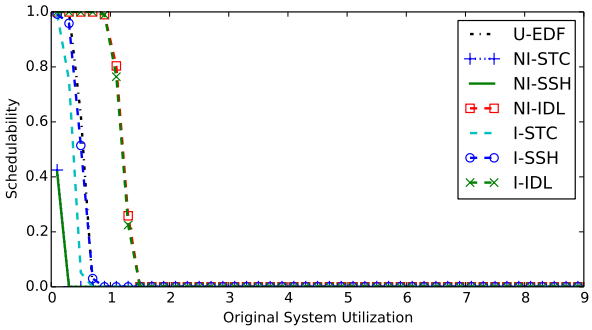
A-Heavy, Long, Heavy, Heavy, Small, Heavy, Light



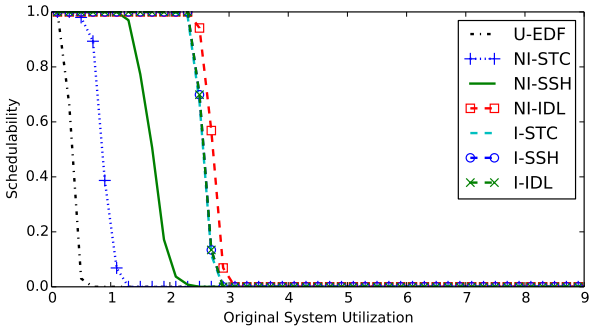
BC-Mod., Long, Light, Heavy, Large, Heavy, Light



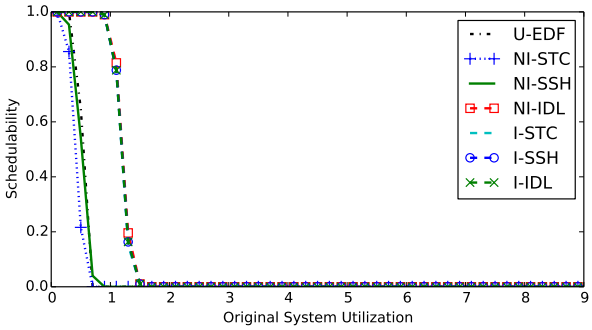
AC-Mod., Short, Heavy, Heavy, Small, Light, Heavy



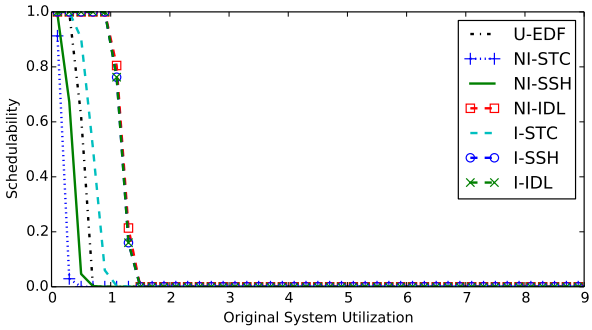
B-Heavy, Short, Light, Light, Small, Heavy, Heavy



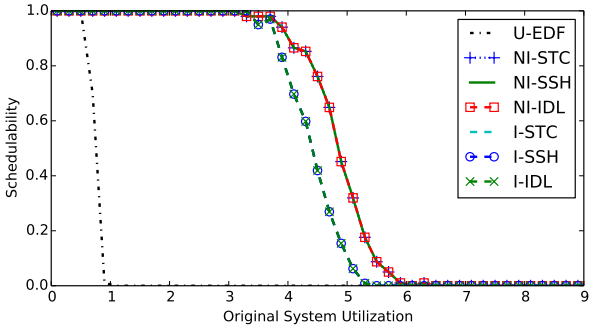
AB-Mod., Long, Light, Heavy, Large, Light, Light



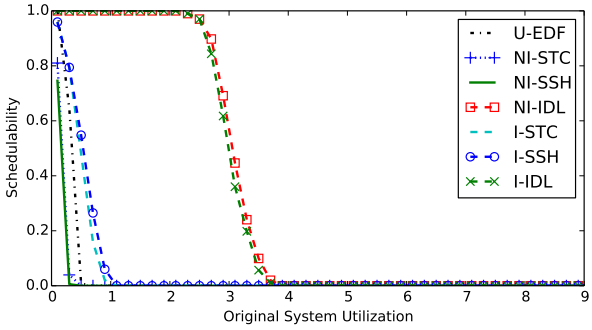
B-Heavy, Short, Light, Light, Large, Heavy, Light



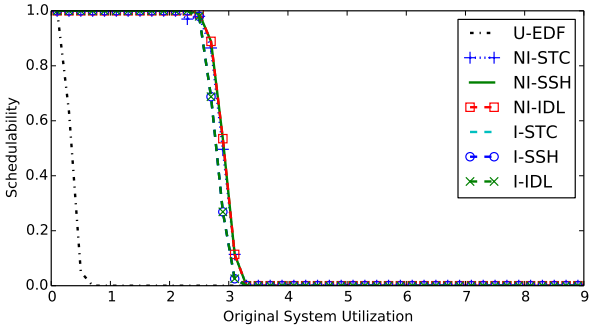
B-Heavy, Short, Light, Light, Small, Light, Heavy



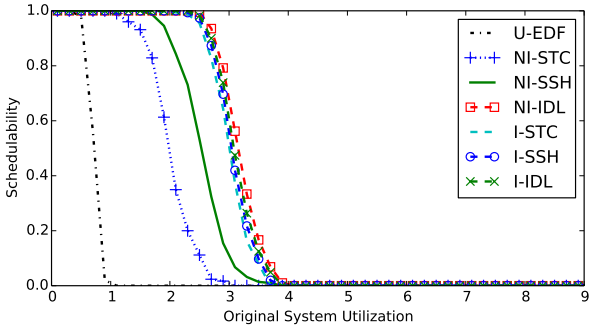
BC-Mod., Long, Heavy, Heavy, Mod., Heavy, Heavy



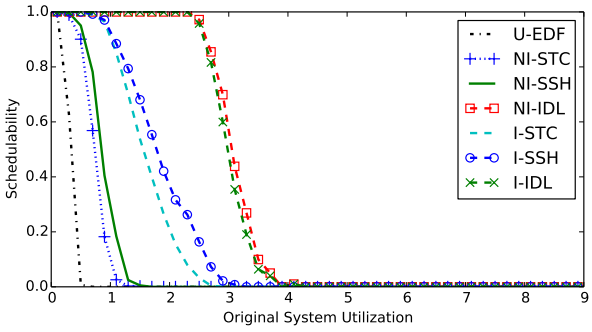
C-Heavy, Long, Light, Heavy, Small, Heavy, Light



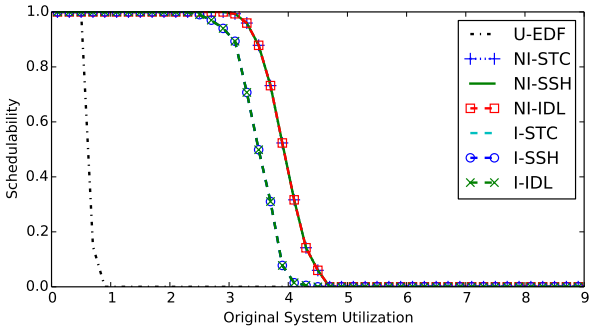
BC-Mod., Long, Light, Heavy, Mod., Light, Light



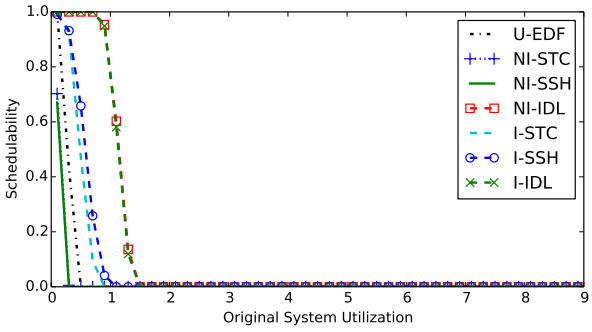
C-Heavy, Long, Light, Light, Mod., Heavy, Light



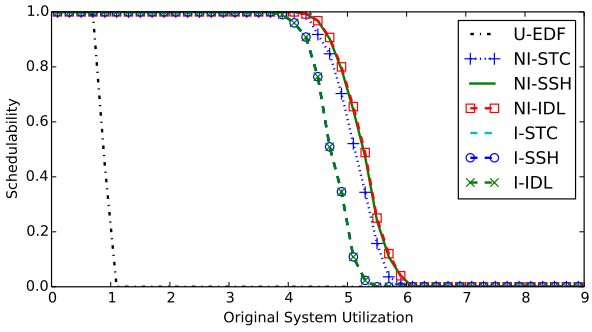
C-Heavy, Long, Light, Heavy, Large, Heavy, Light



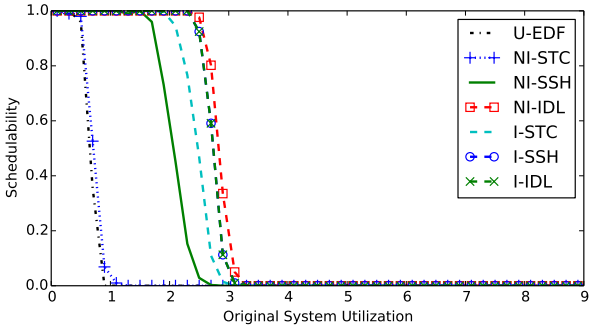
AB-Mod., Long, Heavy, Heavy, Mod., Light, Heavy



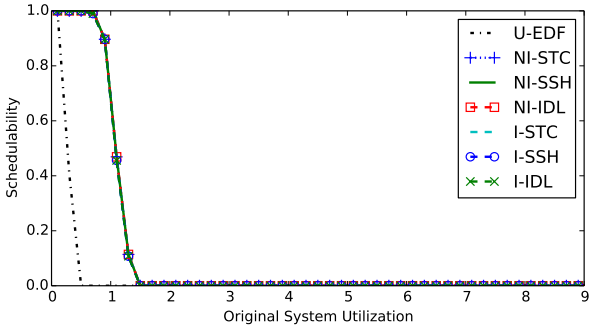
BC-Mod., Short, Light, Heavy, Small, Heavy, Light



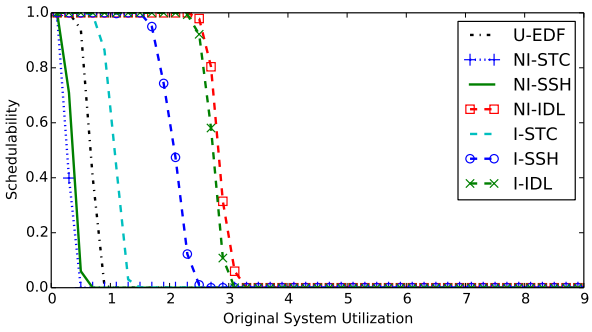
All-Mod., Long, Heavy, Light, Large, Heavy, Light



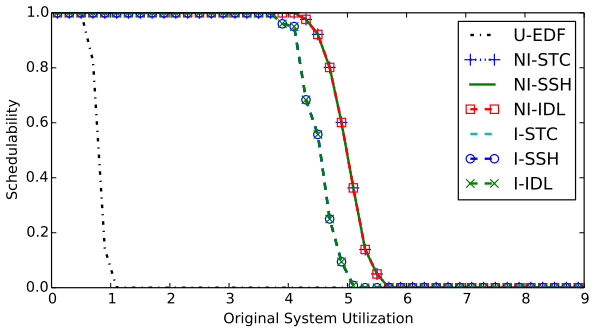
All-Mod., Long, Light, Light, Large, Light, Heavy



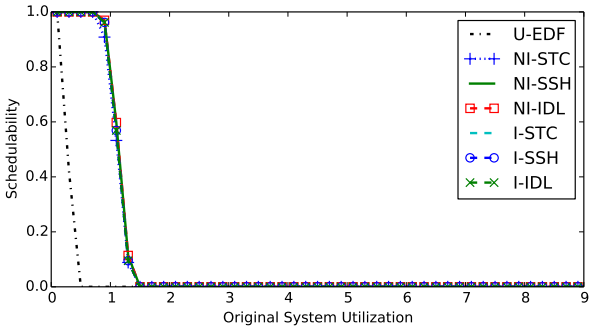
C-Heavy, Short, Light, Heavy, Mod., Heavy, Heavy



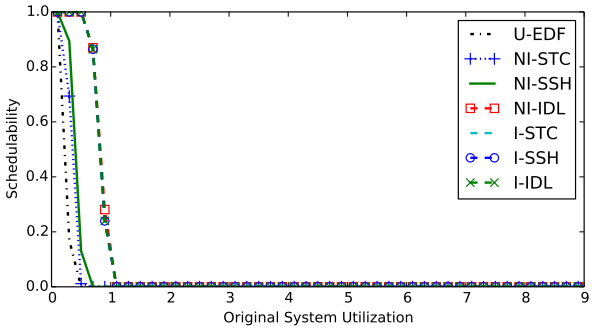
All-Mod., Long, Light, Light, Small, Light, Light



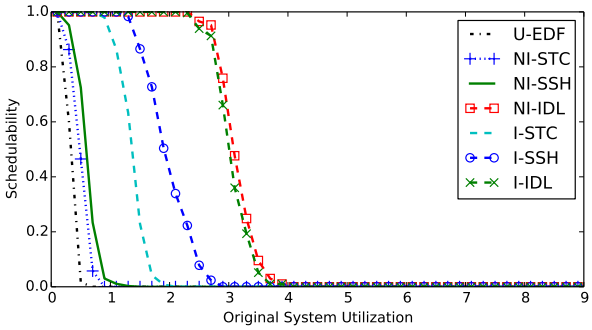
AC-Mod., Short, Heavy, Light, Mod., Heavy, Light



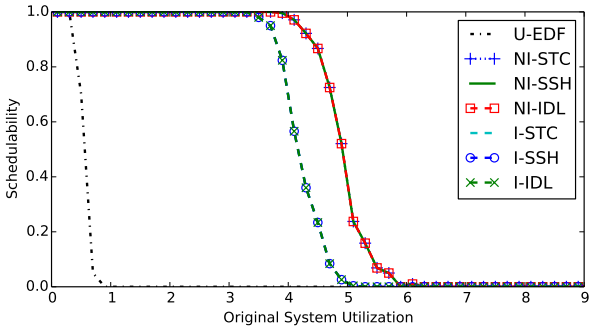
BC-Mod., Short, Light, Heavy, Mod., Heavy, Heavy



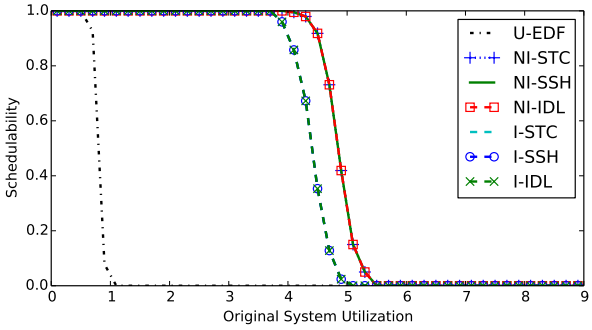
A-Heavy, Short, Light, Heavy, Large, Heavy, Light



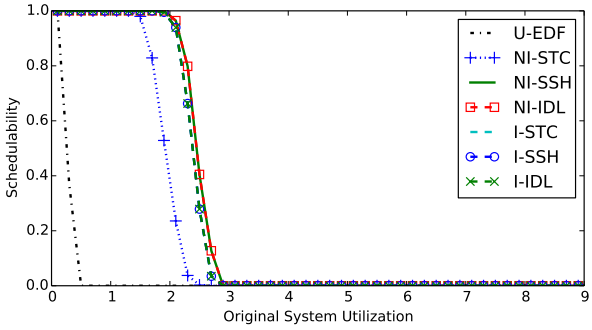
C-Heavy, Long, Light, Heavy, Small, Light, Light



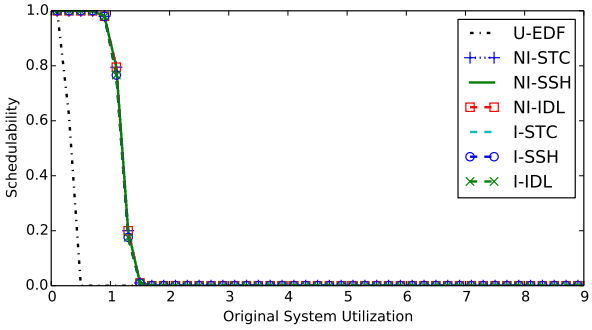
BC-Mod., Short, Heavy, Heavy, Mod., Heavy, Heavy



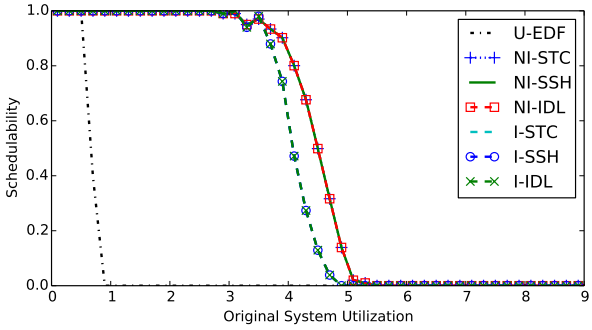
All-Mod., Short, Heavy, Light, Mod., Heavy, Light



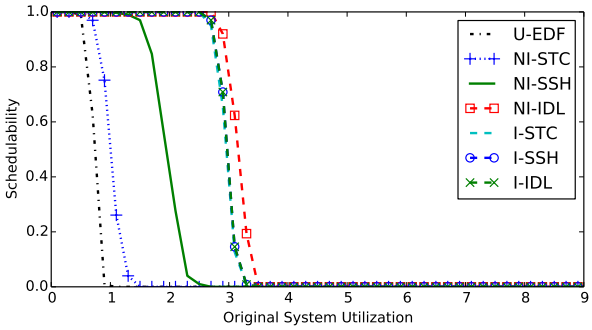
AC-Mod., Long, Light, Heavy, Mod., Light, Heavy



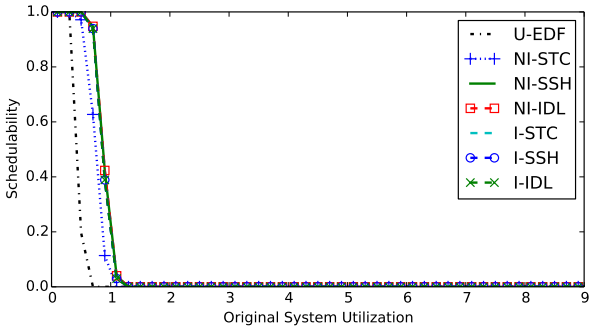
B-Heavy, Short, Light, Heavy, Mod., Light, Heavy



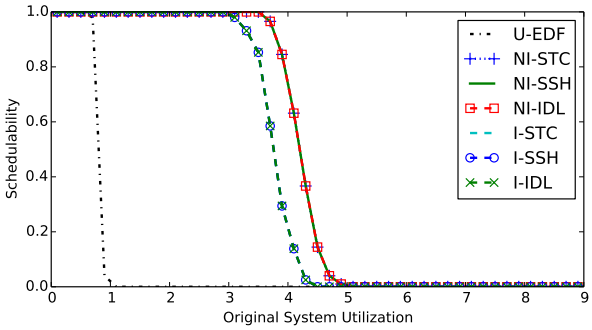
All-Mod., Long, Heavy, Heavy, Large, Light, Heavy



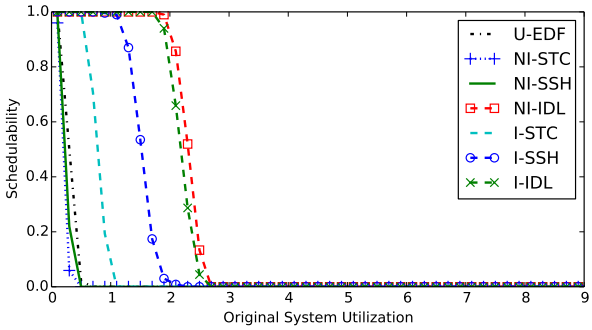
B-Heavy, Long, Light, Light, Large, Light, Light



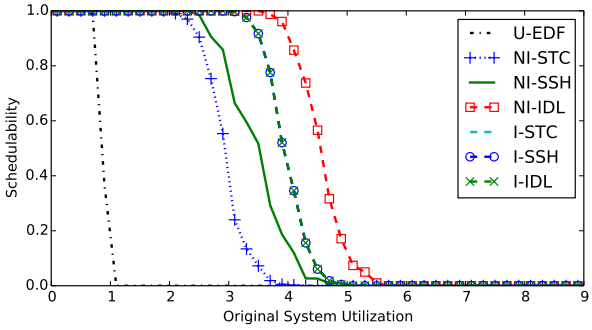
AC-Mod., Short, Light, Light, Large, Light, Heavy



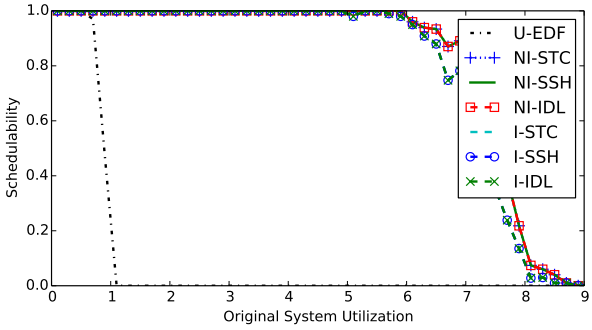
AB-Mod., Short, Heavy, Light, Mod., Light, Light



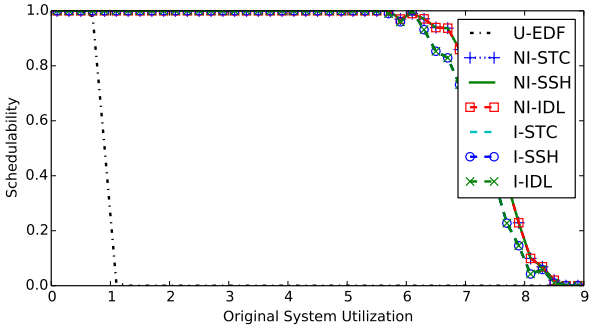
A-Heavy, Long, Light, Heavy, Small, Light, Light



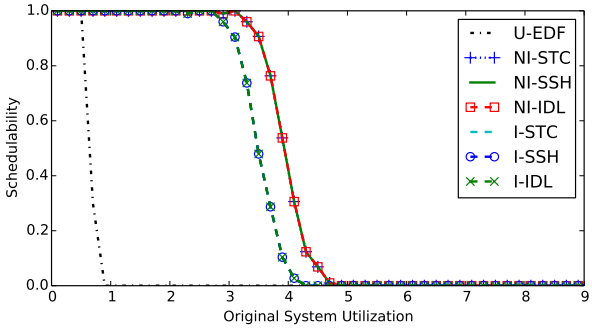
AB-Mod., Long, Heavy, Light, Small, Light, Light



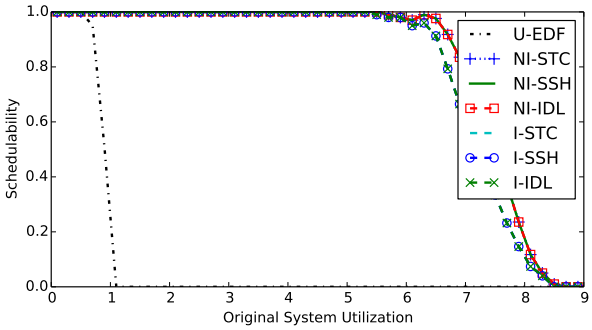
C-Heavy, Long, Heavy, Light, Mod., Light, Light



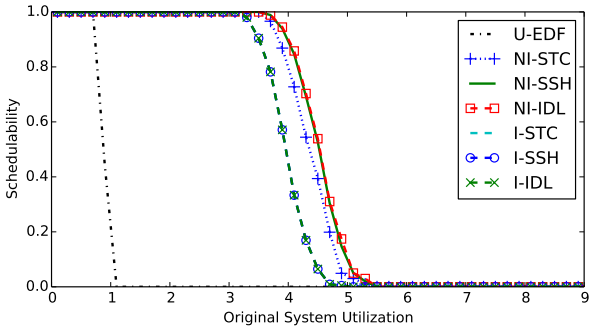
C-Heavy, Long, Heavy, Light, Large, Light, Light



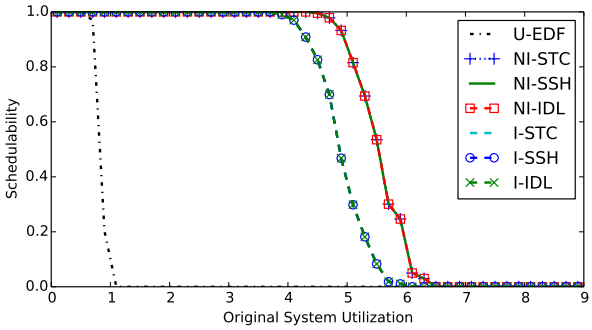
AB-Mod., Long, Heavy, Heavy, Mod., Light, Light



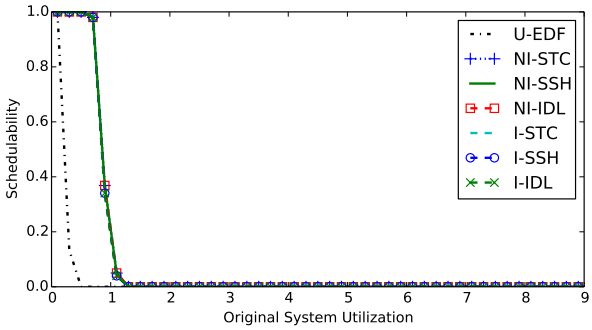
C-Heavy, Long, Heavy, Light, Mod., Heavy, Light



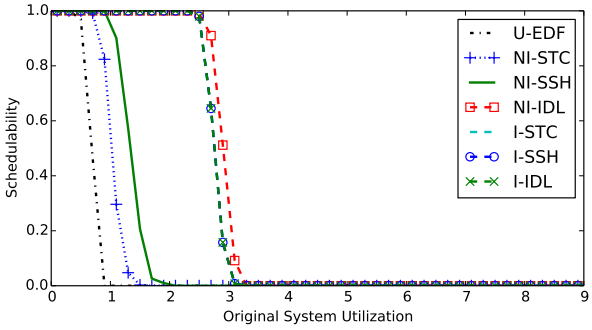
AB-Mod., Long, Heavy, Light, Large, Heavy, Light



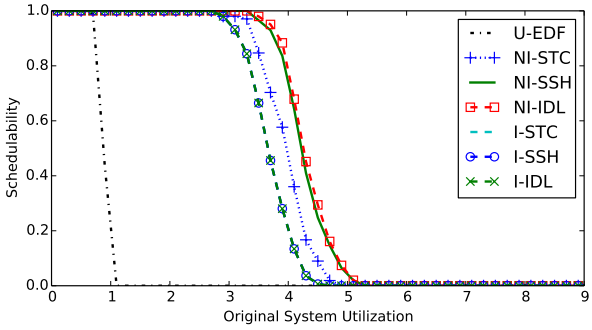
BC-Mod., Short, Heavy, Light, Mod., Heavy, Heavy



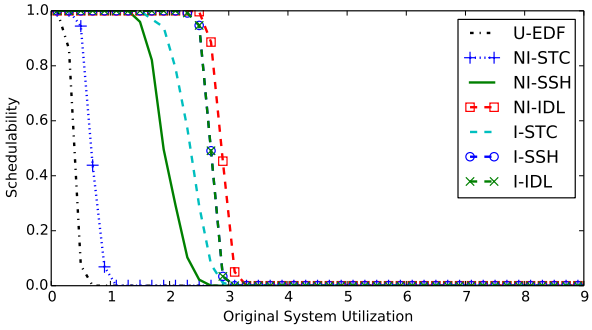
AC-Mod., Short, Light, Heavy, Mod., Light, Light



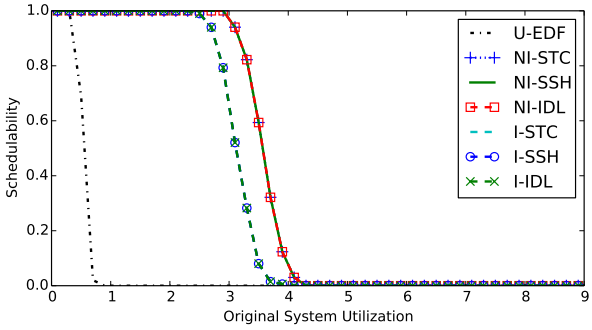
AB-Mod., Long, Light, Light, Mod., Heavy, Light



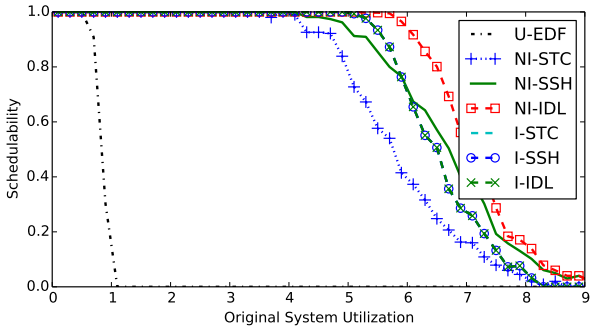
A-Heavy, Long, Heavy, Light, Large, Heavy, Light



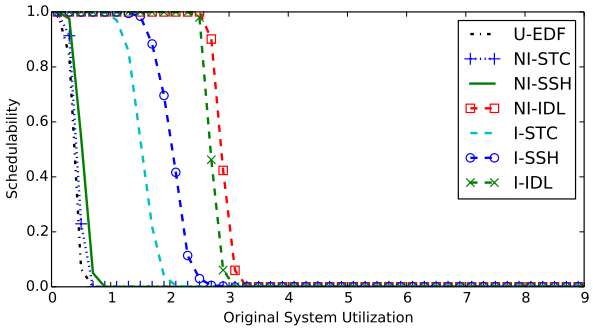
B-Heavy, Long, Light, Heavy, Large, Light, Heavy



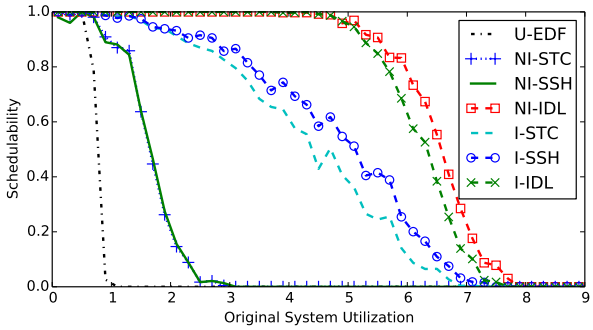
A-Heavy, Short, Heavy, Heavy, Large, Light, Light



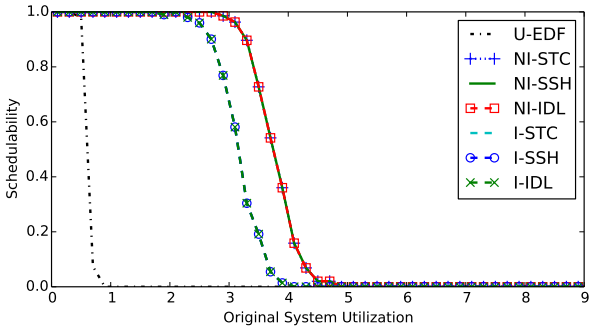
C-Heavy, Short, Heavy, Light, Small, Light, Light



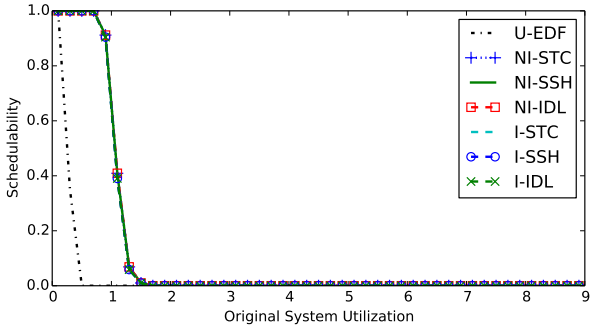
B-Heavy, Long, Light, Heavy, Large, Heavy, Light



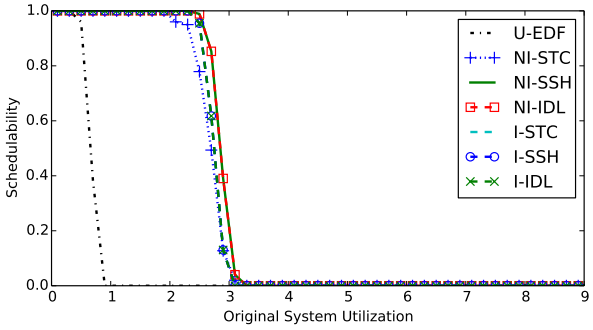
C-Heavy, Long, Heavy, Heavy, Small, Heavy, Heavy



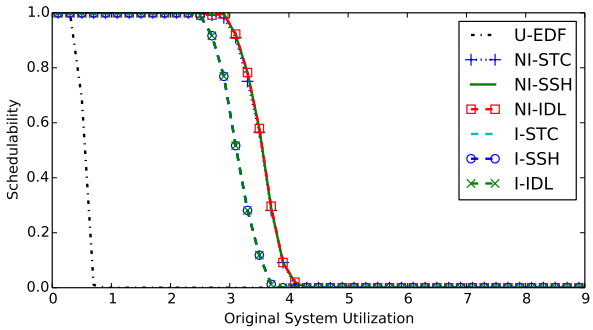
A-Heavy, Long, Heavy, Heavy, Mod., Heavy, Light



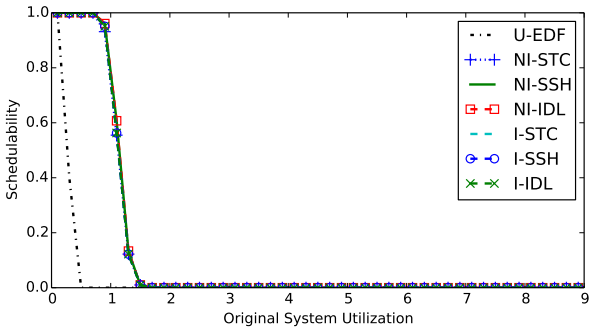
C-Heavy, Short, Light, Heavy, Mod., Light, Heavy



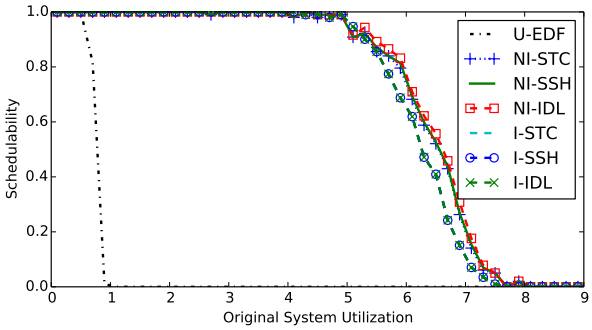
All-Mod., Long, Light, Light, Mod., Light, Light



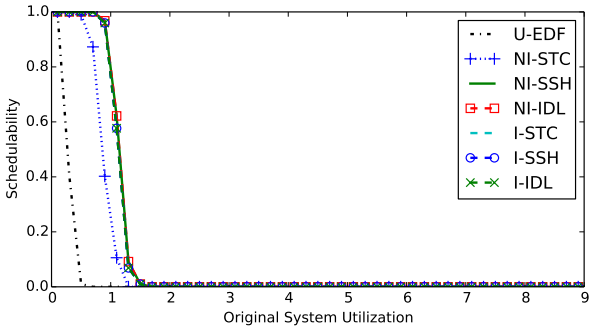
A-Heavy, Short, Heavy, Heavy, Large, Heavy, Light



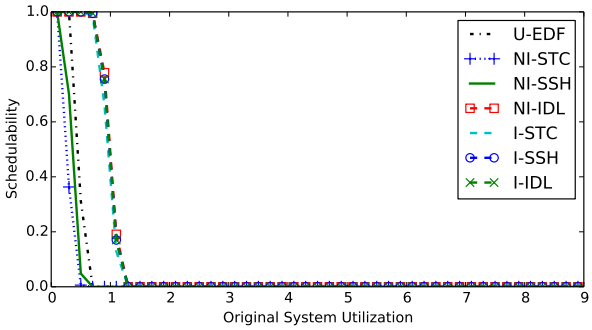
BC-Mod., Short, Light, Heavy, Large, Light, Light



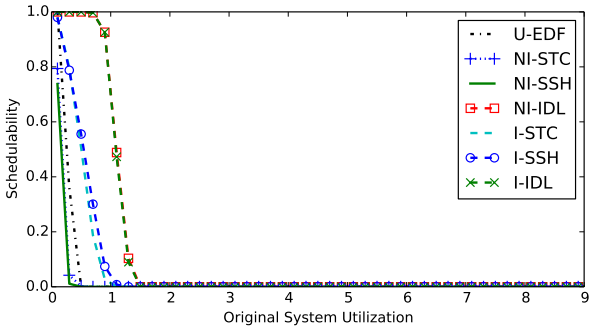
C-Heavy, Long, Heavy, Heavy, Large, Heavy, Light



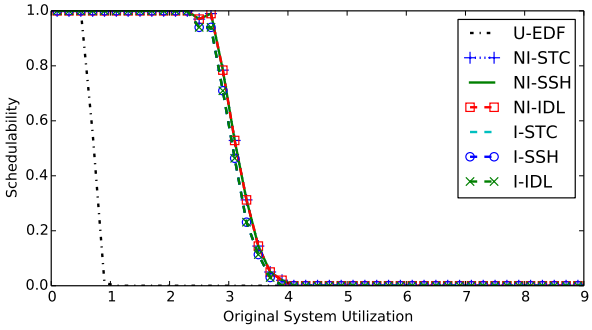
BC-Mod., Short, Light, Heavy, Large, Light, Heavy



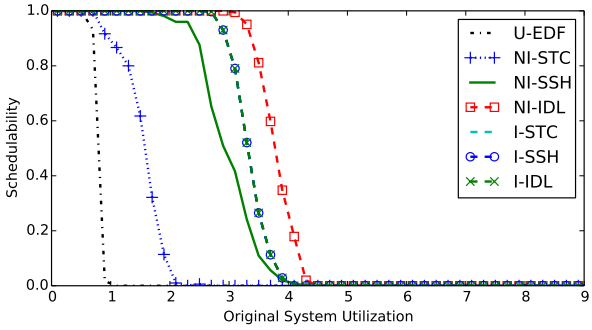
All-Mod., Short, Light, Light, Small, Light, Light



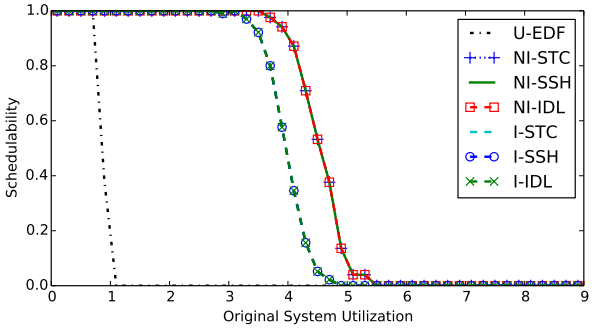
C-Heavy, Short, Light, Heavy, Small, Heavy, Light



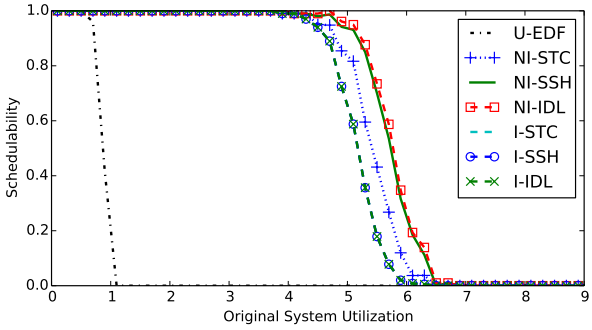
C-Heavy, Long, Light, Light, Mod., Light, Light



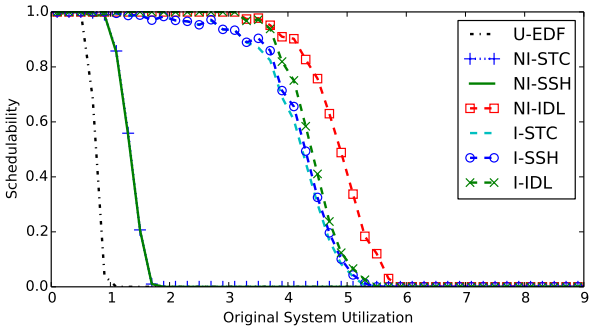
A-Heavy, Short, Heavy, Light, Small, Light, Heavy



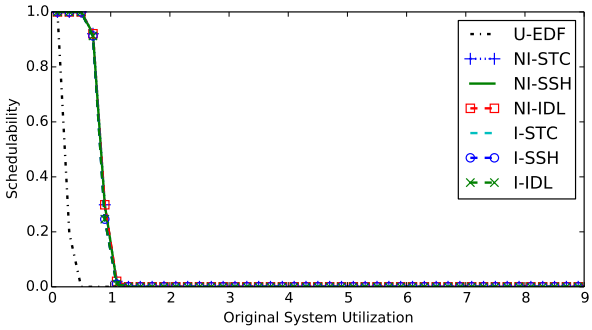
AB-Mod., Long, Heavy, Light, Large, Light, Heavy



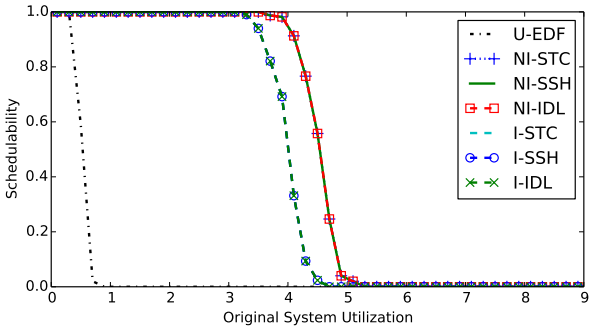
AC-Mod., Long, Heavy, Light, Large, Heavy, Light



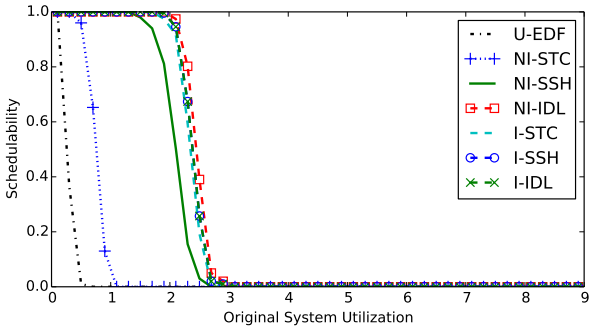
BC-Mod., Long, Heavy, Heavy, Small, Heavy, Heavy



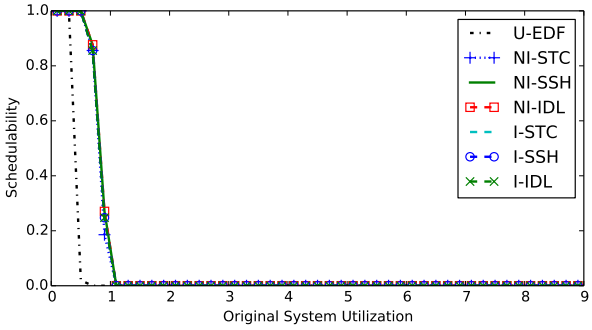
A-Heavy, Short, Light, Heavy, Mod., Light, Light



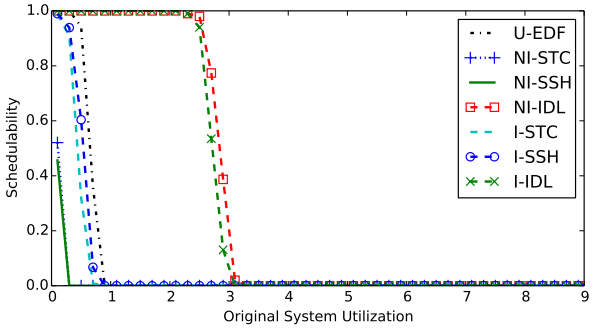
All-Mod., Short, Heavy, Heavy, Mod., Light, Light



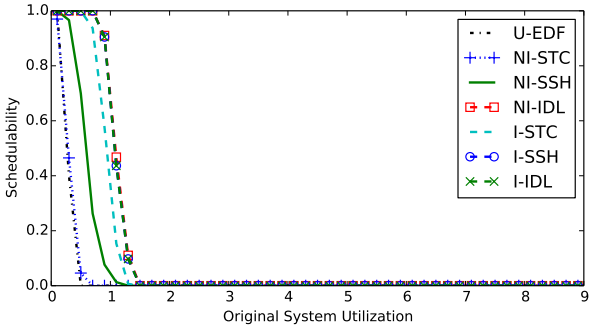
AC-Mod., Long, Light, Heavy, Large, Light, Heavy



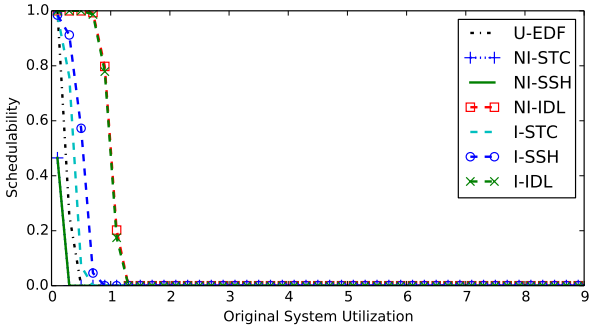
A-Heavy, Short, Light, Light, Mod., Heavy, Light



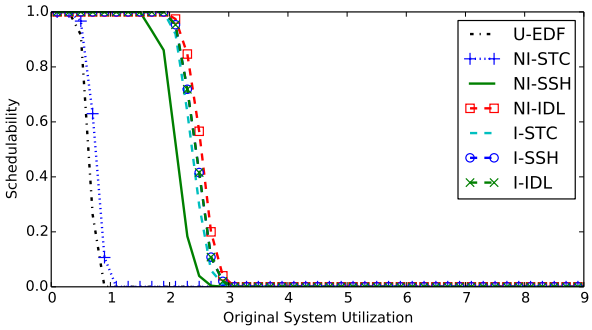
All-Mod., Long, Light, Light, Small, Heavy, Light



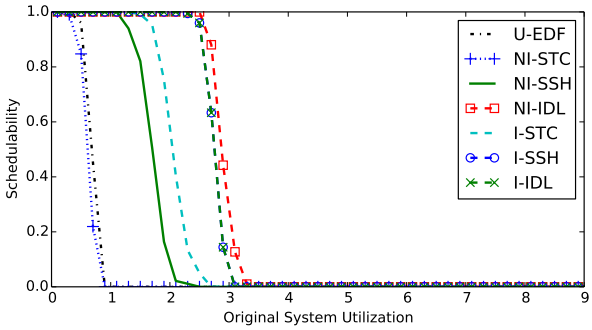
C-Heavy, Short, Light, Heavy, Small, Light, Heavy



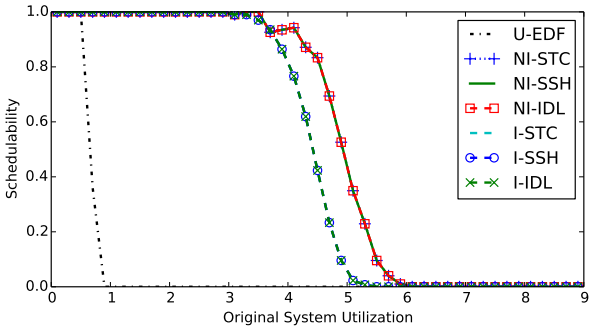
All-Mod., Short, Light, Heavy, Small, Heavy, Heavy



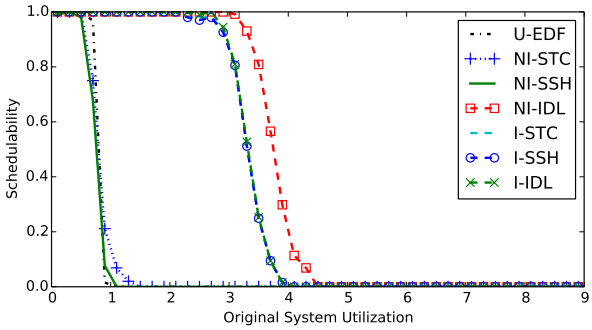
AC-Mod., Long, Light, Light, Large, Light, Heavy



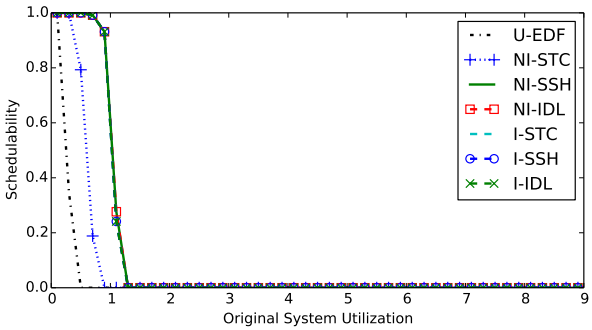
AB-Mod., Long, Light, Light, Large, Light, Heavy



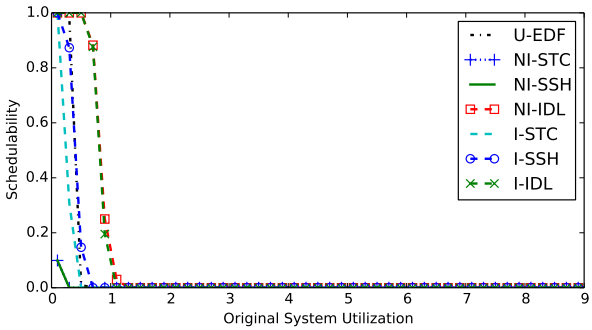
AC-Mod., Long, Heavy, Heavy, Mod., Light, Light



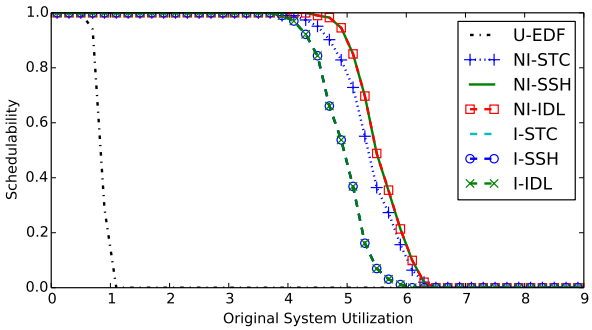
A-Heavy, Short, Heavy, Light, Small, Heavy, Light



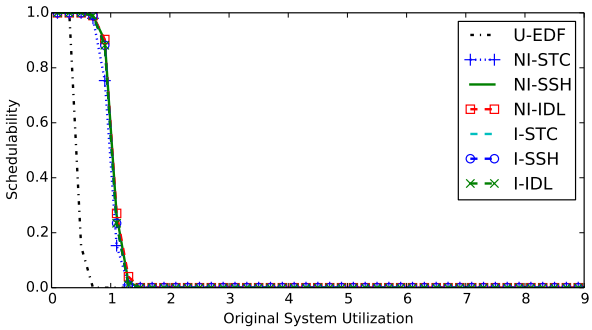
AB-Mod., Short, Light, Heavy, Large, Light, Heavy



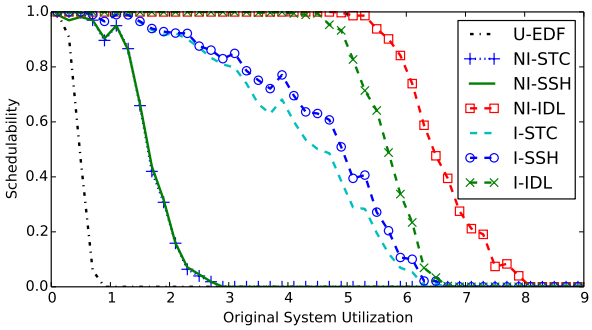
A-Heavy, Short, Light, Light, Small, Heavy, Heavy



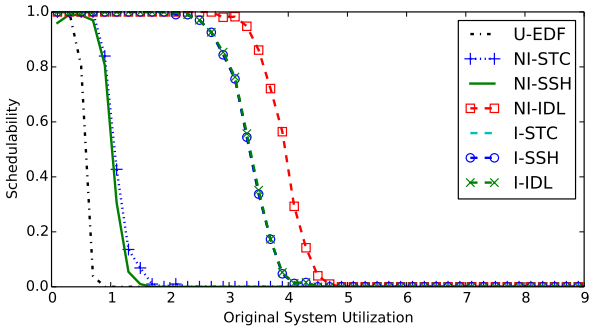
BC-Mod., Short, Heavy, Light, Large, Heavy, Heavy



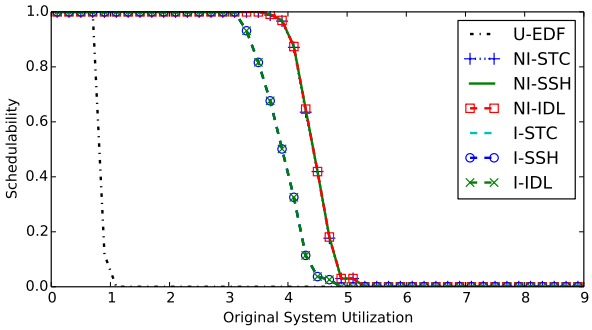
AB-Mod., Short, Light, Light, Mod., Heavy, Light



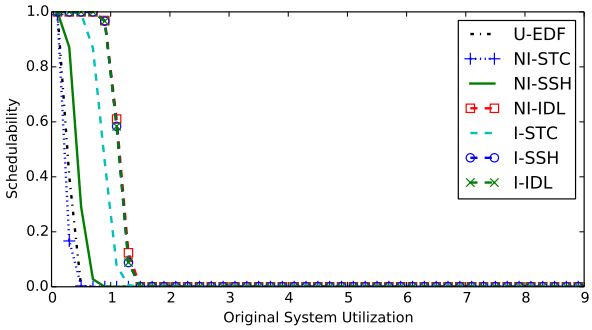
C-Heavy, Short, Heavy, Heavy, Small, Heavy, Heavy



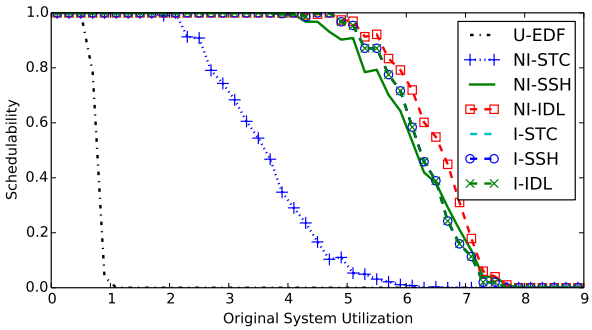
B-Heavy, Short, Heavy, Heavy, Small, Heavy, Light



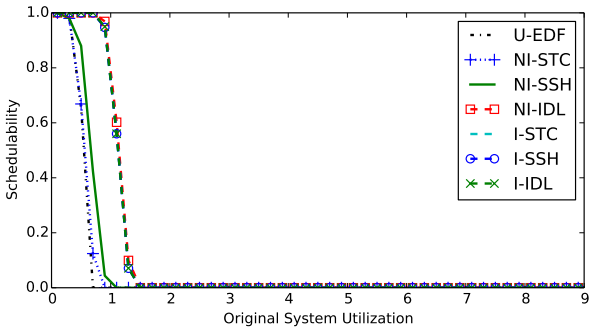
B-Heavy, Short, Heavy, Light, Large, Heavy, Light



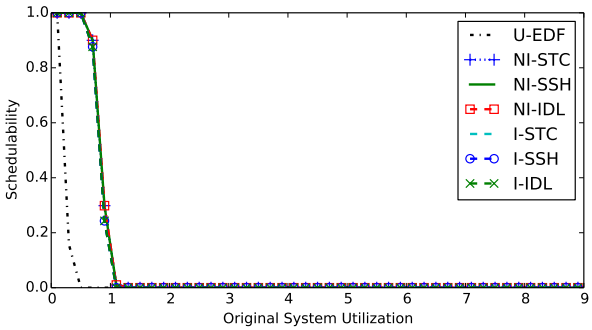
BC-Mod., Short, Light, Heavy, Small, Light, Heavy



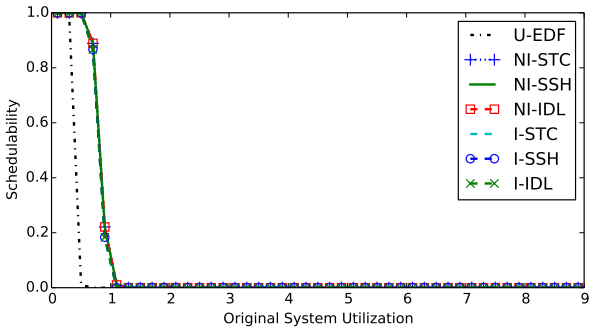
C-Heavy, Long, Heavy, Heavy, Small, Light, Heavy



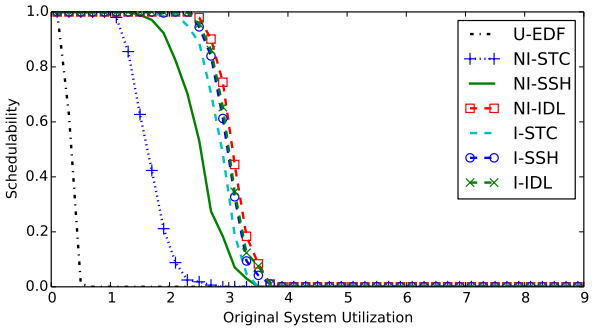
BC-Mod., Short, Light, Light, Large, Heavy, Light



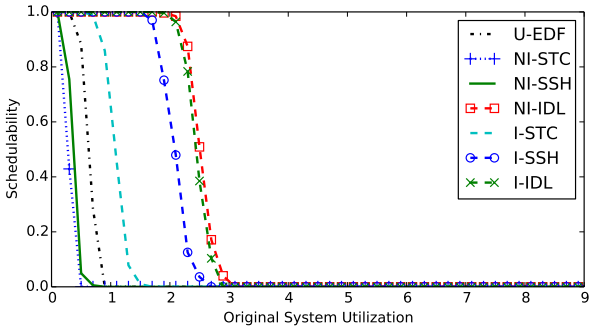
A-Heavy, Short, Light, Heavy, Mod., Light, Heavy



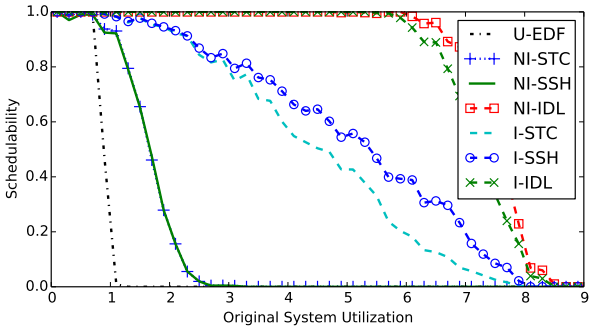
A-Heavy, Short, Light, Light, Mod., Light, Light



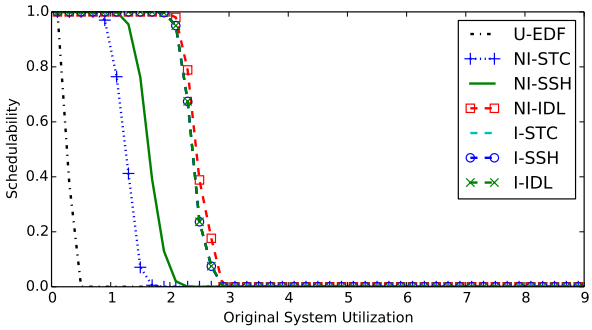
C-Heavy, Long, Light, Heavy, Mod., Heavy, Heavy



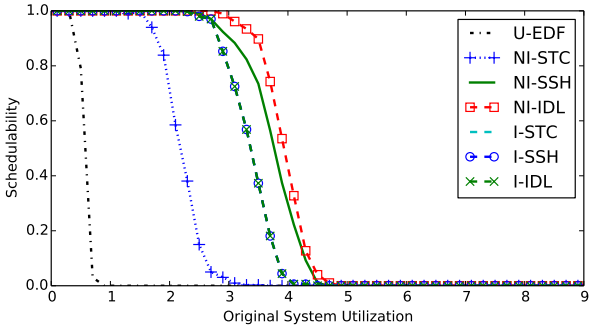
AC-Mod., Long, Light, Light, Small, Light, Light



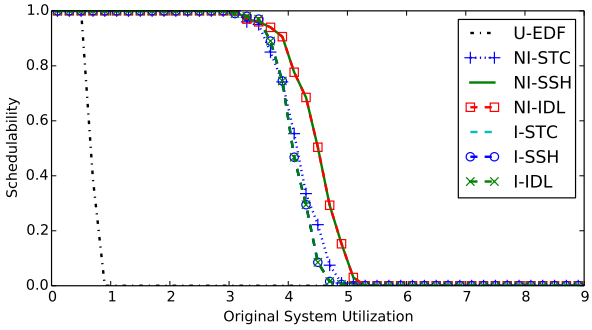
C-Heavy, Long, Heavy, Light, Small, Heavy, Heavy



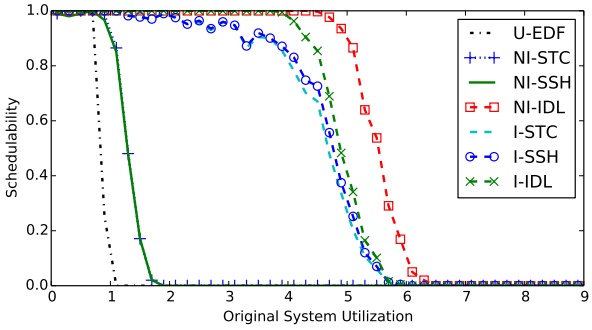
AC-Mod., Long, Light, Heavy, Mod., Heavy, Light



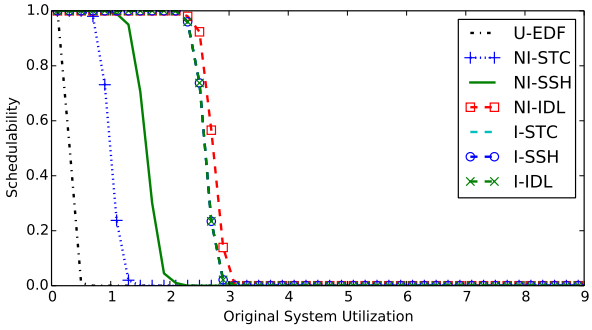
B-Heavy, Short, Heavy, Heavy, Small, Light, Heavy



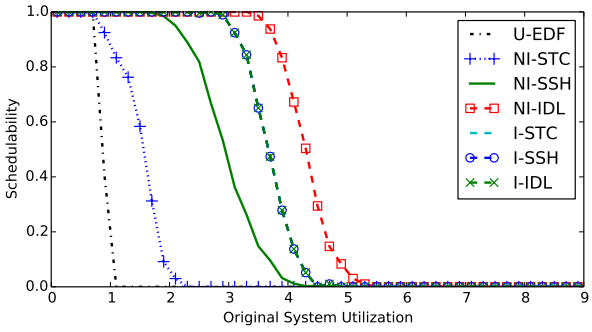
All-Mod., Long, Heavy, Heavy, Large, Heavy, Heavy



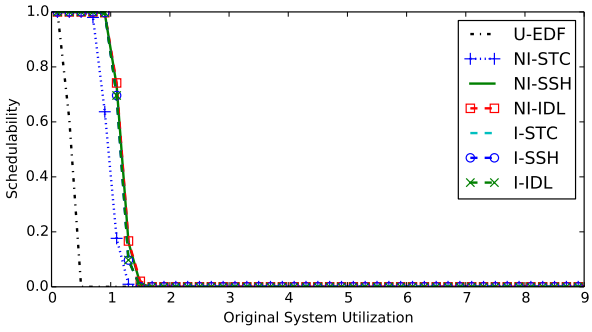
BC-Mod., Short, Heavy, Light, Small, Heavy, Heavy



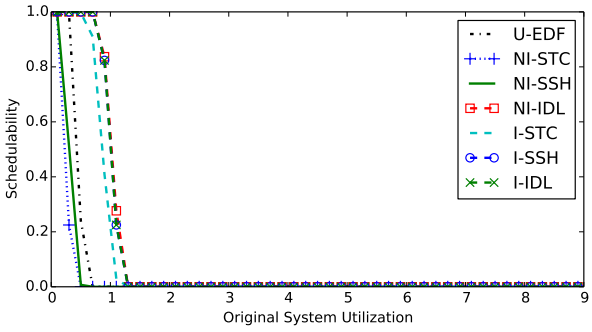
All-Mod., Long, Light, Heavy, Mod., Heavy, Heavy



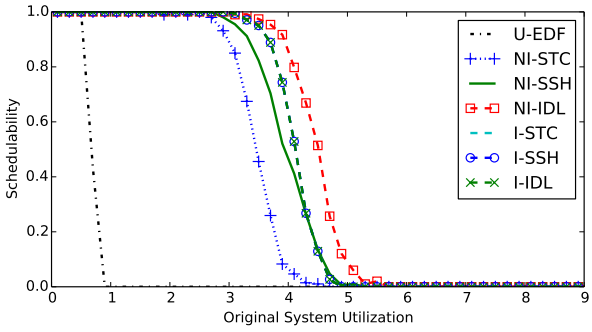
A-Heavy, Long, Heavy, Light, Small, Light, Heavy



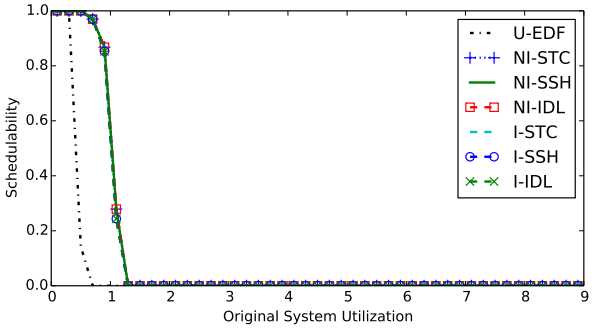
B-Heavy, Short, Light, Heavy, Mod., Heavy, Heavy



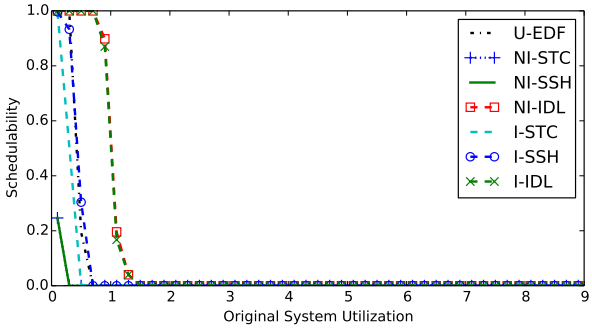
AB-Mod., Short, Light, Light, Small, Light, Light



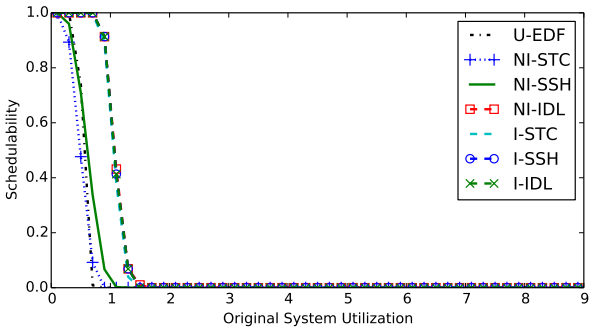
All-Mod., Long, Heavy, Heavy, Small, Light, Light



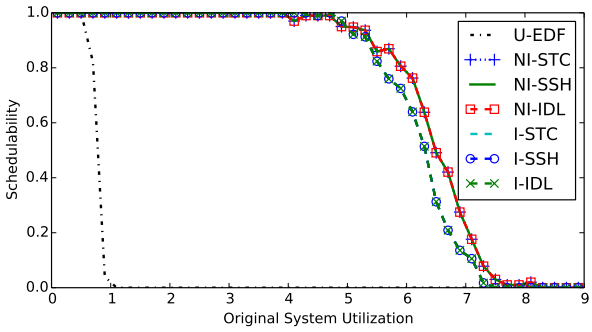
AB-Mod., Short, Light, Light, Mod., Light, Light



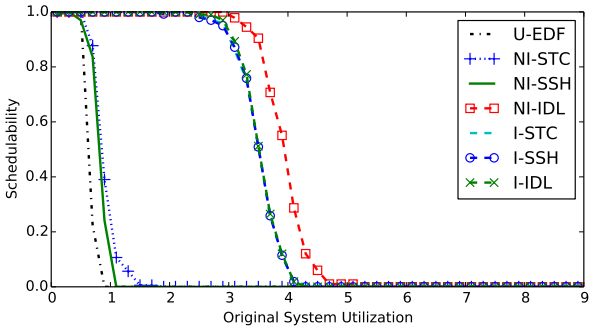
AB-Mod., Short, Light, Light, Small, Heavy, Heavy



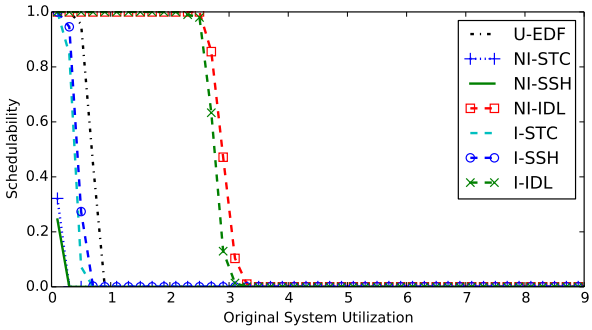
C-Heavy, Short, Light, Light, Small, Light, Light



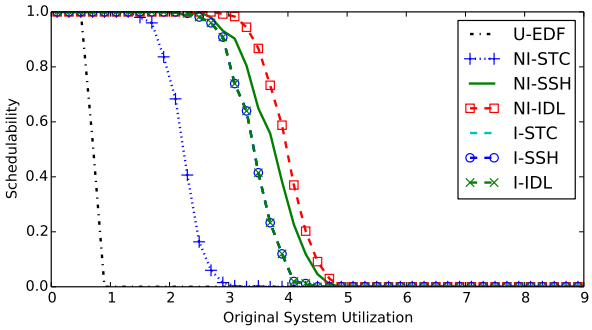
C-Heavy, Long, Heavy, Heavy, Large, Light, Heavy



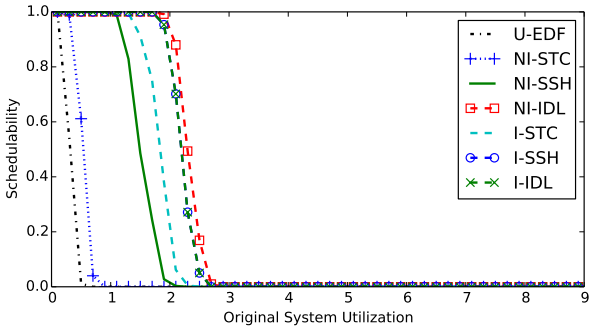
AB-Mod., Long, Heavy, Heavy, Small, Heavy, Light



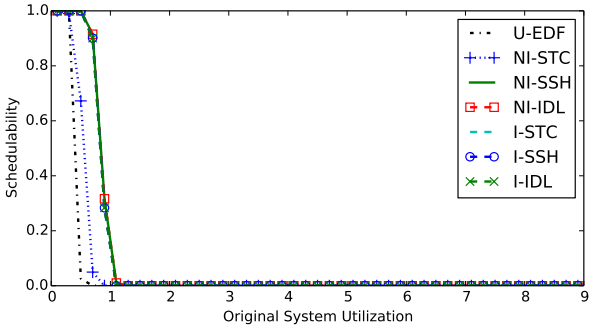
AB-Mod., Long, Light, Light, Small, Heavy, Light



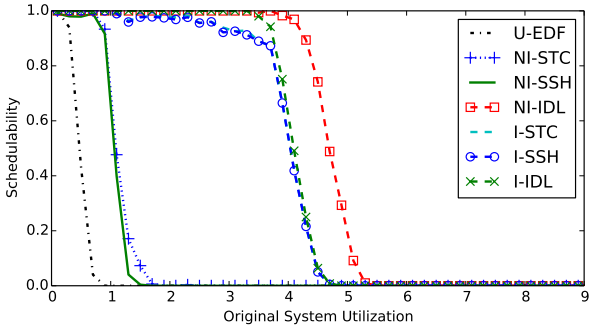
B-Heavy, Long, Heavy, Heavy, Small, Light, Heavy



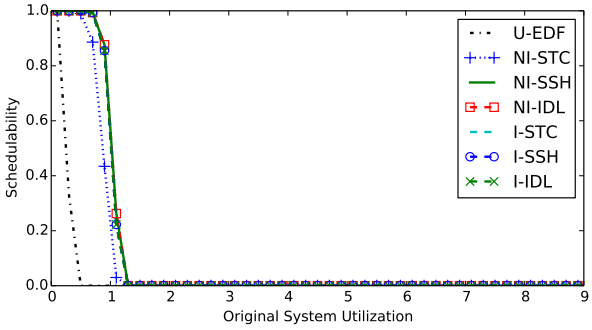
A-Heavy, Long, Light, Heavy, Large, Light, Heavy



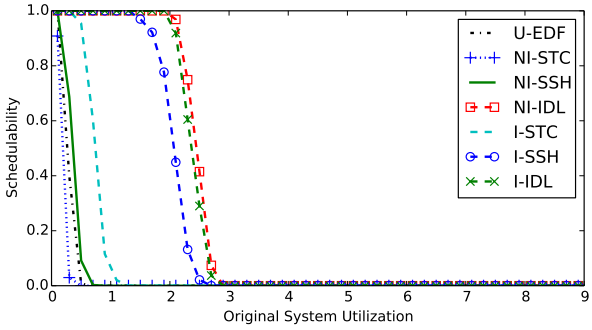
A-Heavy, Short, Light, Light, Large, Light, Heavy



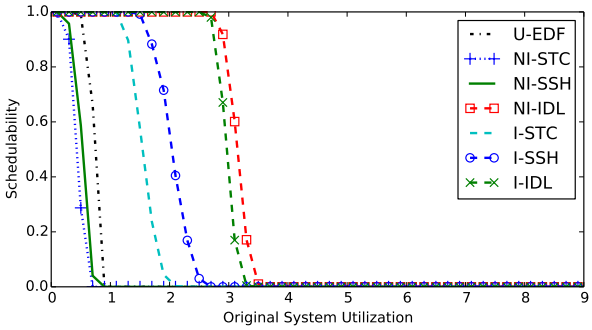
AC-Mod., Short, Heavy, Heavy, Small, Heavy, Light



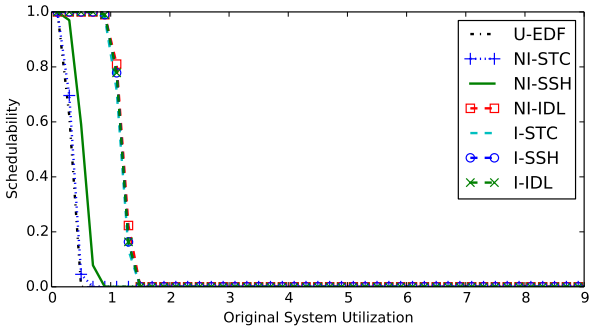
AB-Mod., Short, Light, Heavy, Large, Light, Light



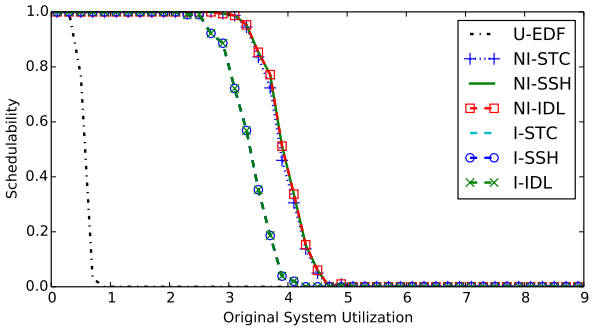
AC-Mod., Long, Light, Heavy, Small, Light, Heavy



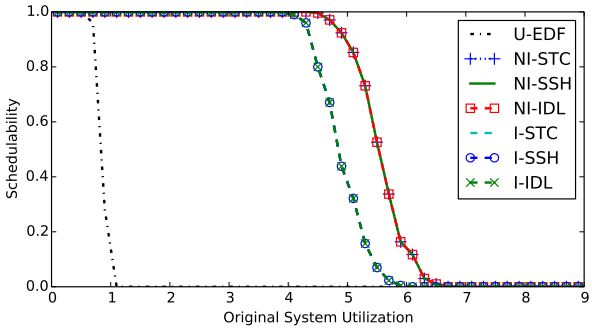
B-Heavy, Long, Light, Light, Large, Heavy, Light



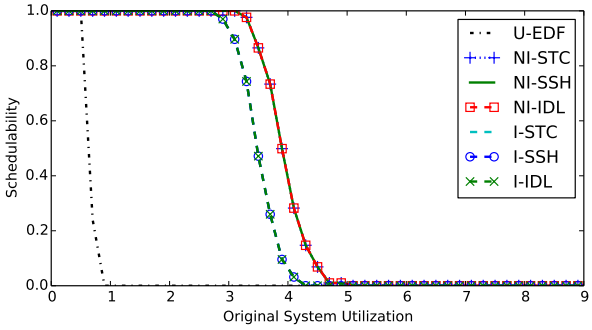
B-Heavy, Short, Light, Heavy, Large, Heavy, Heavy



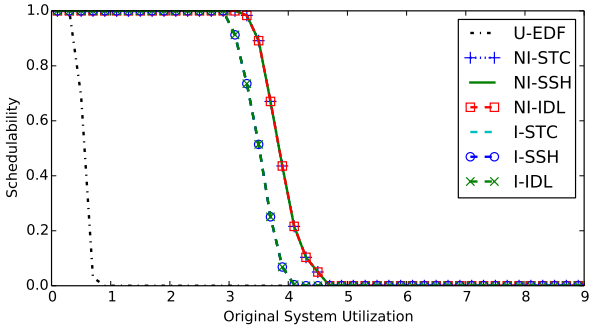
B-Heavy, Short, Heavy, Heavy, Large, Heavy, Heavy



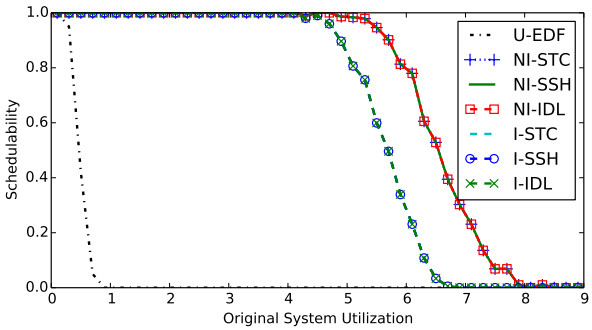
BC-Mod., Short, Heavy, Light, Mod., Light, Heavy



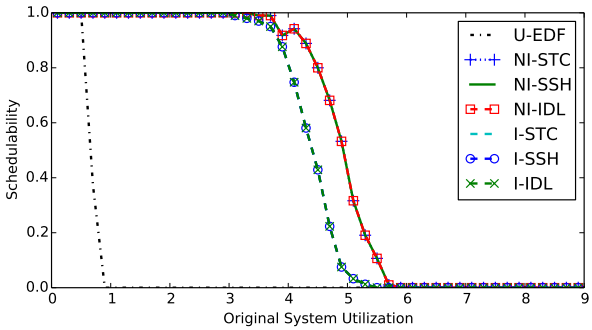
AB-Mod., Long, Heavy, Heavy, Large, Light, Heavy



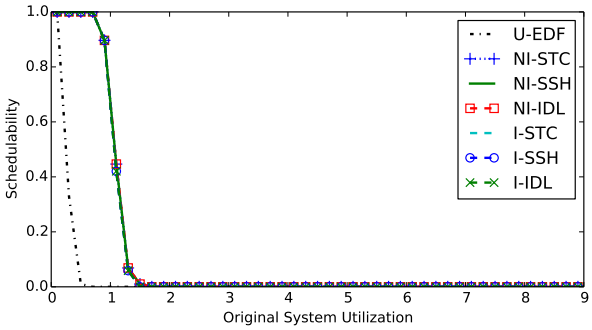
AB-Mod., Short, Heavy, Heavy, Large, Light, Heavy



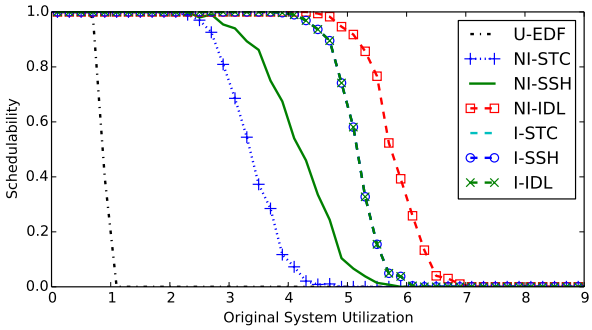
C-Heavy, Short, Heavy, Heavy, Mod., Light, Light



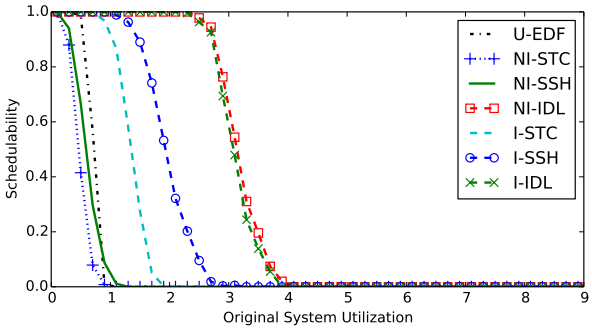
AC-Mod., Long, Heavy, Heavy, Mod., Heavy, Heavy



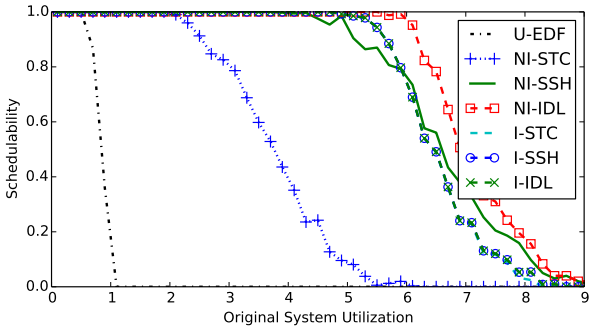
C-Heavy, Short, Light, Heavy, Mod., Light, Light



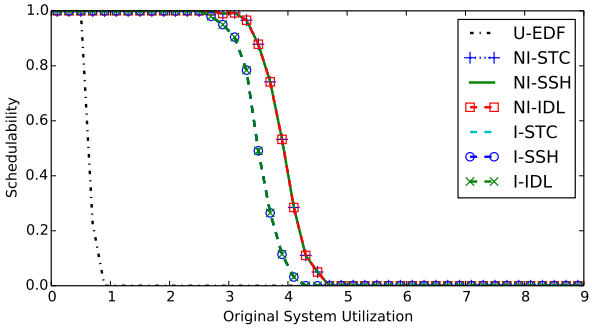
AC-Mod., Long, Heavy, Light, Small, Light, Light



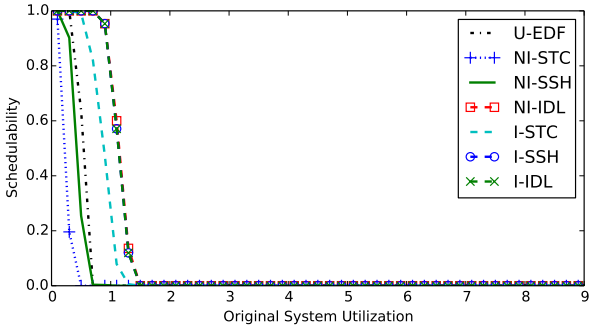
C-Heavy, Long, Light, Light, Small, Light, Light



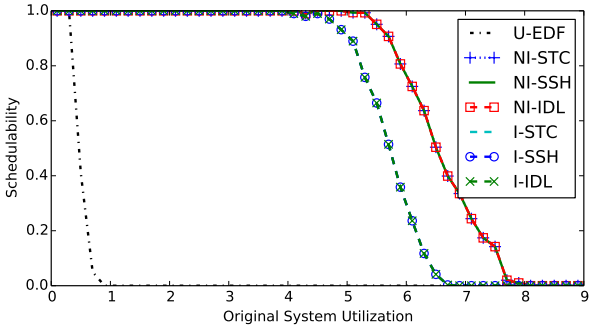
C-Heavy, Short, Heavy, Light, Small, Light, Heavy



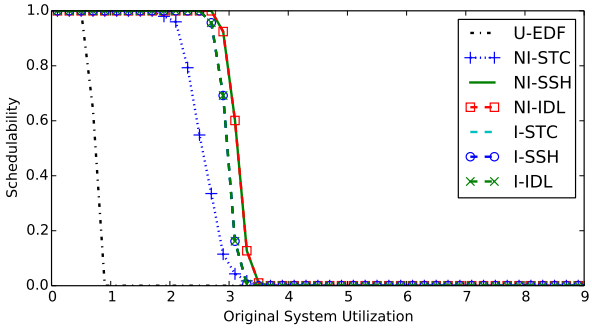
AB-Mod., Long, Heavy, Heavy, Mod., Heavy, Light



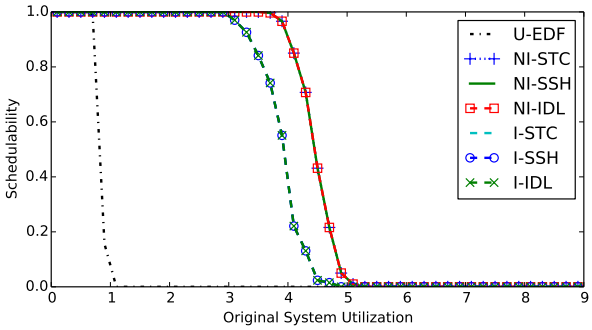
BC-Mod., Short, Light, Light, Small, Light, Heavy



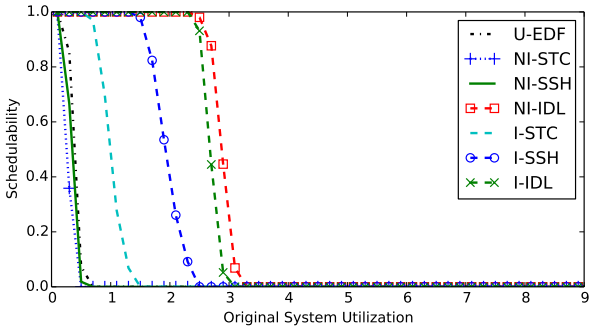
C-Heavy, Short, Heavy, Heavy, Large, Light, Heavy



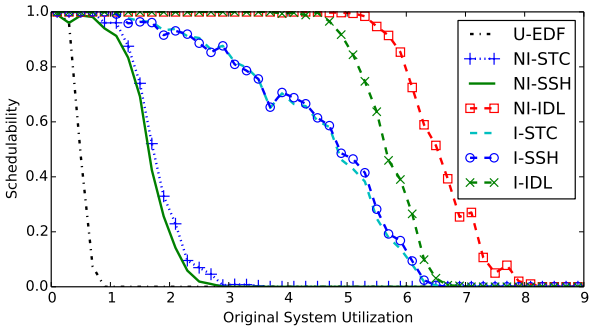
B-Heavy, Long, Light, Light, Mod., Light, Light



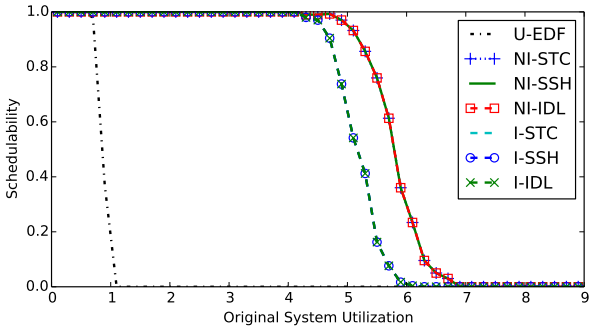
B-Heavy, Short, Heavy, Light, Mod., Light, Heavy



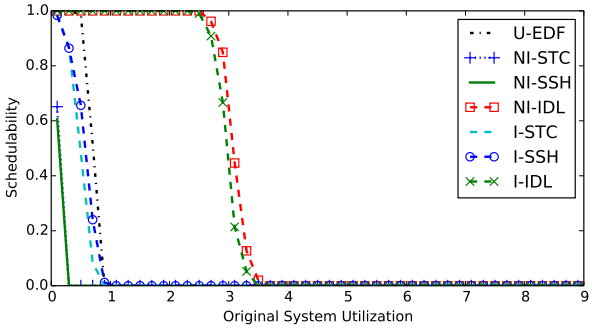
B-Heavy, Long, Light, Heavy, Small, Light, Light



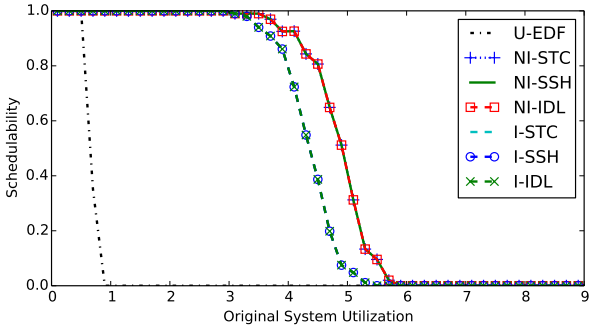
C-Heavy, Short, Heavy, Heavy, Small, Heavy, Light



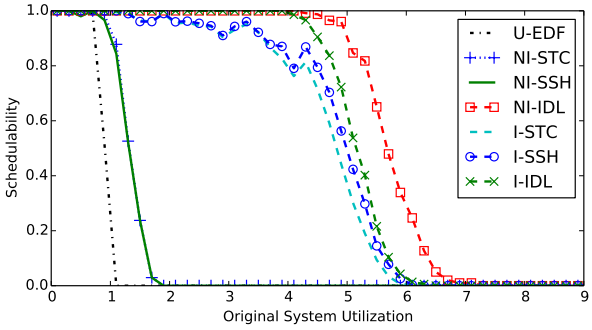
AC-Mod., Long, Heavy, Light, Mod., Light, Heavy



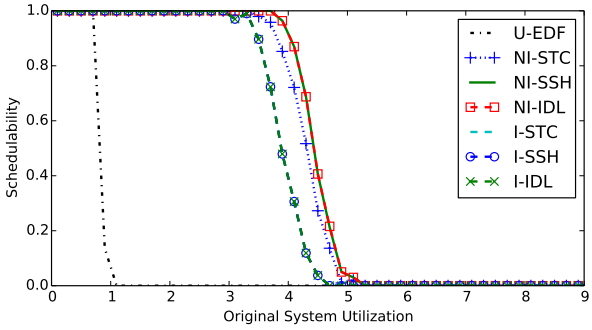
BC-Mod., Long, Light, Light, Small, Heavy, Light



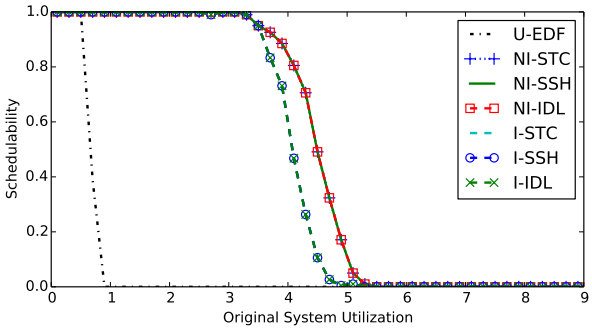
AC-Mod., Long, Heavy, Heavy, Large, Light, Heavy



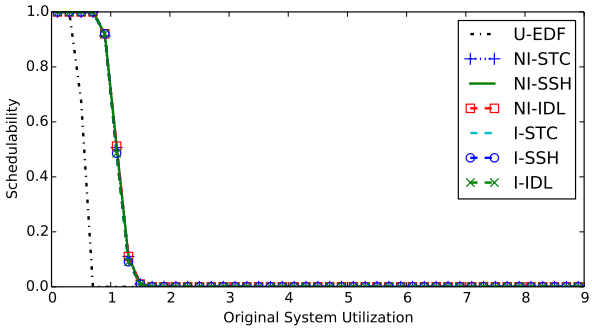
BC-Mod., Long, Heavy, Light, Small, Heavy, Heavy



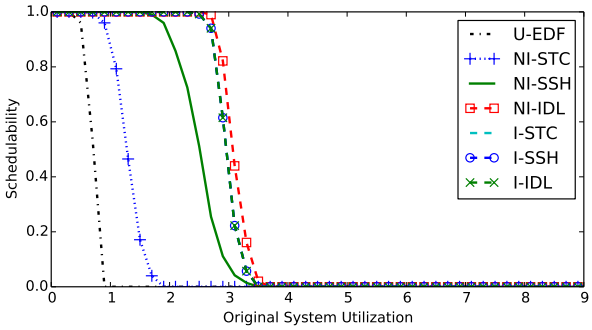
B-Heavy, Short, Heavy, Light, Large, Heavy, Heavy



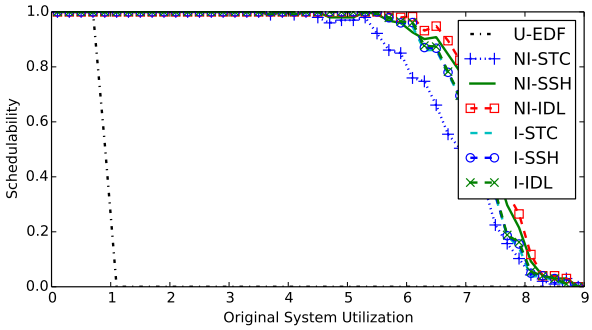
All-Mod., Long, Heavy, Heavy, Mod., Light, Heavy



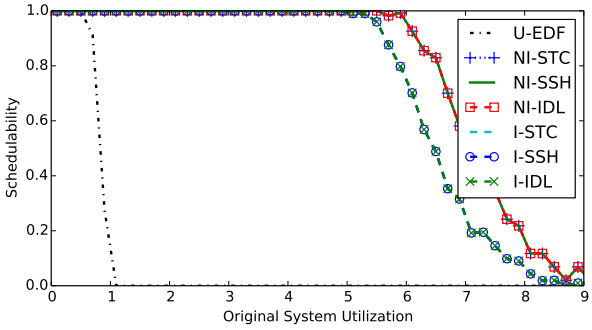
C-Heavy, Short, Light, Light, Mod., Heavy, Heavy



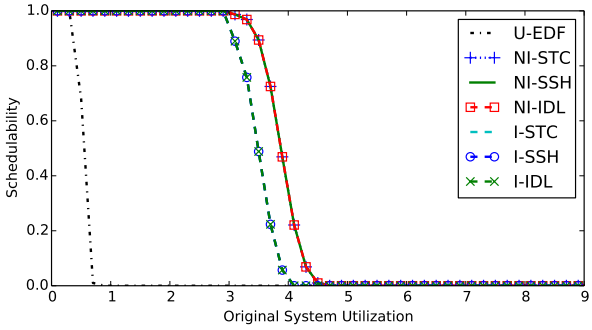
BC-Mod., Long, Light, Light, Large, Light, Light



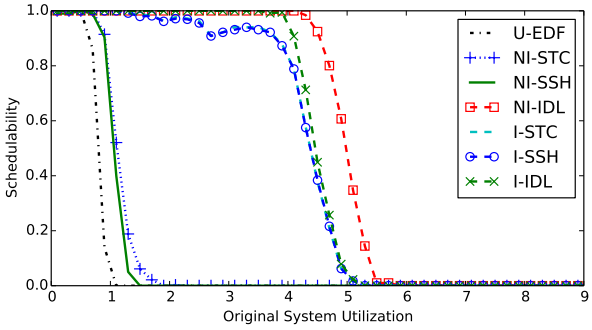
C-Heavy, Long, Heavy, Light, Large, Heavy, Heavy



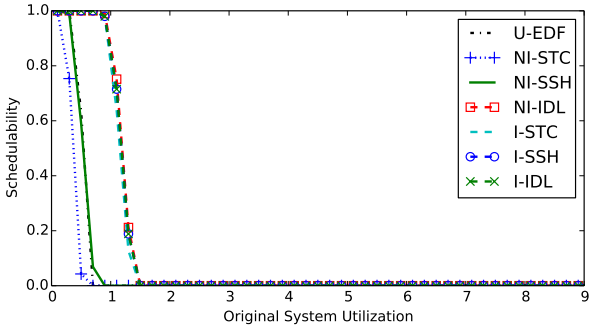
C-Heavy, Short, Heavy, Light, Mod., Light, Light



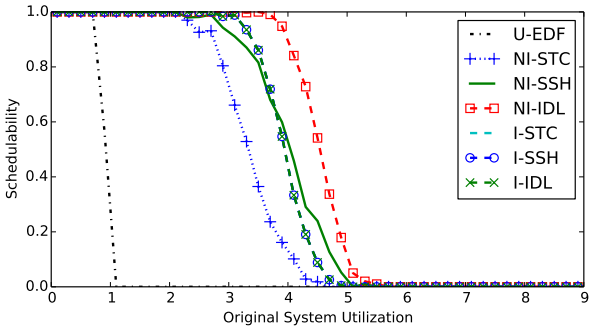
AB-Mod., Short, Heavy, Heavy, Mod., Heavy, Light



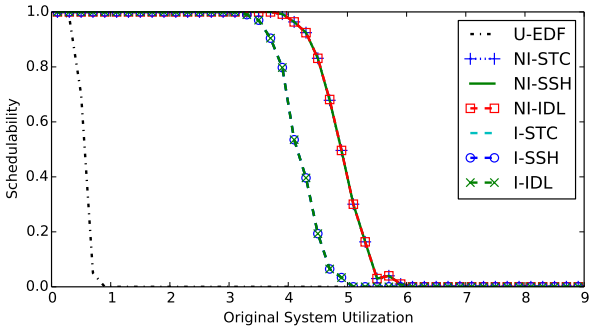
AC-Mod., Short, Heavy, Light, Small, Heavy, Light



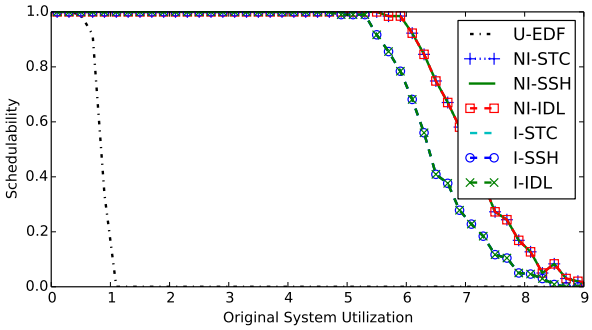
B-Heavy, Short, Light, Light, Large, Heavy, Heavy



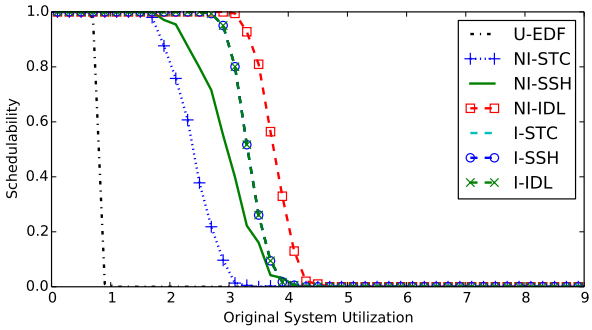
B-Heavy, Long, Heavy, Light, Small, Light, Light



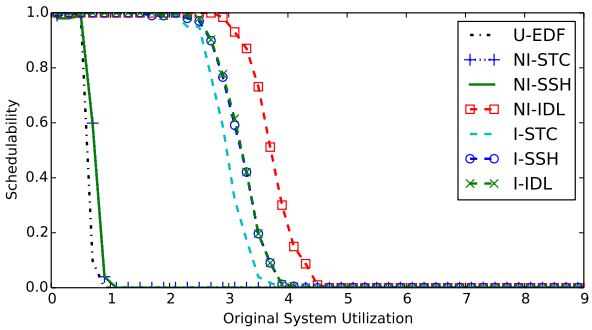
BC-Mod., Short, Heavy, Heavy, Large, Heavy, Light



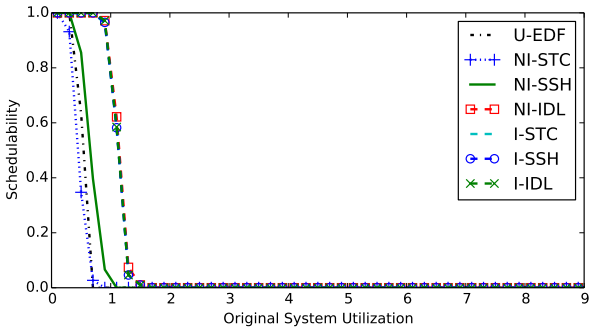
C-Heavy, Short, Heavy, Light, Large, Light, Heavy



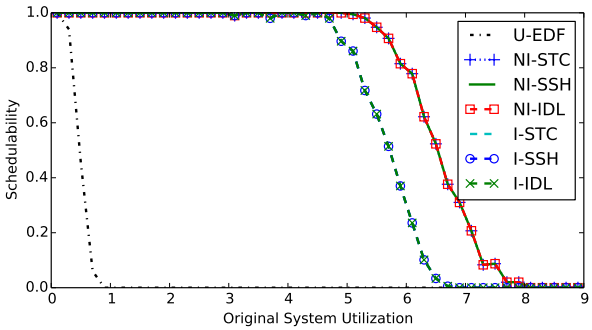
A-Heavy, Short, Heavy, Light, Small, Light, Light



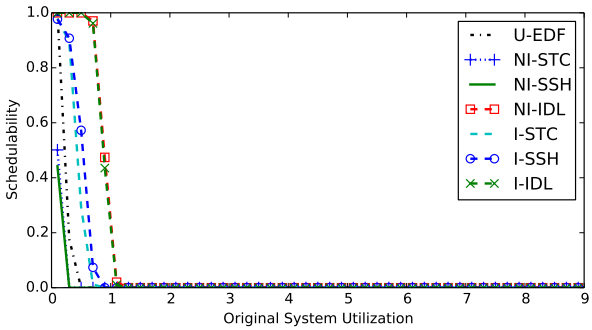
A-Heavy, Long, Heavy, Heavy, Small, Heavy, Heavy



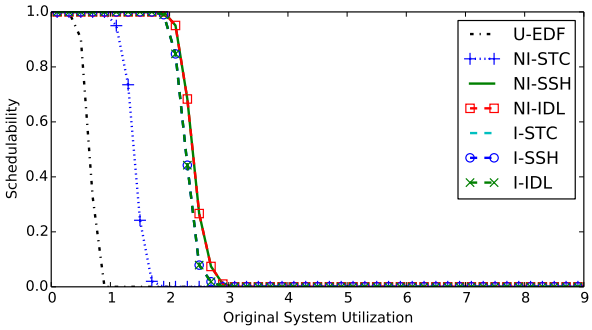
BC-Mod., Short, Light, Light, Large, Heavy, Heavy



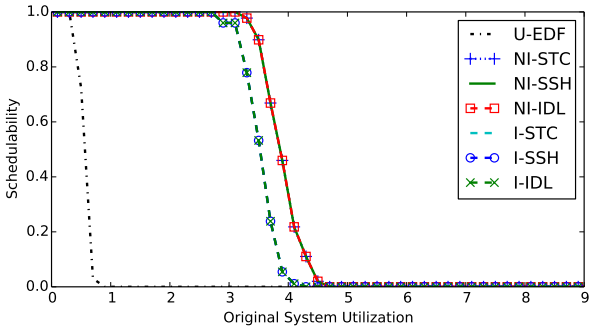
C-Heavy, Short, Heavy, Heavy, Mod., Heavy, Heavy



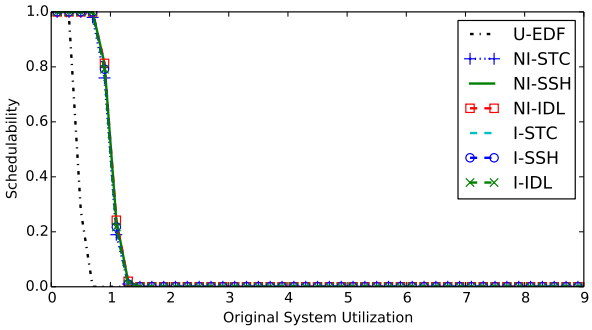
AC-Mod., Short, Light, Heavy, Small, Heavy, Light



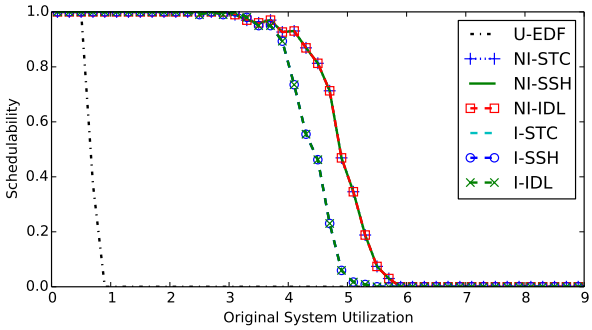
A-Heavy, Long, Light, Light, Mod., Light, Heavy



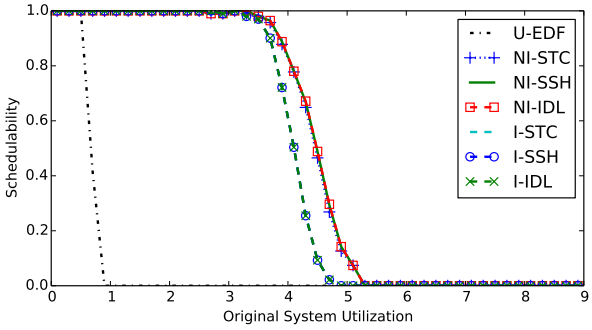
AB-Mod., Short, Heavy, Heavy, Large, Light, Light



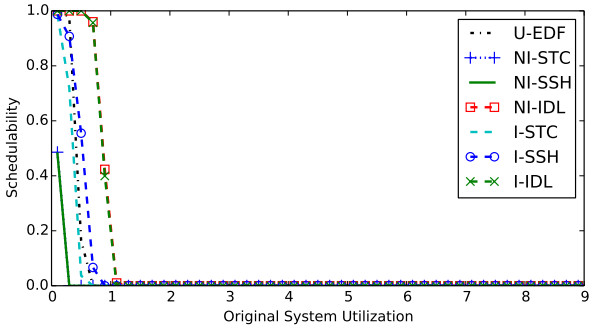
All-Mod., Short, Light, Light, Large, Light, Light



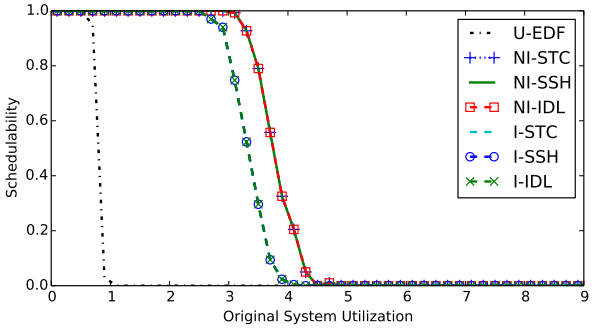
AC-Mod., Long, Heavy, Heavy, Mod., Heavy, Light



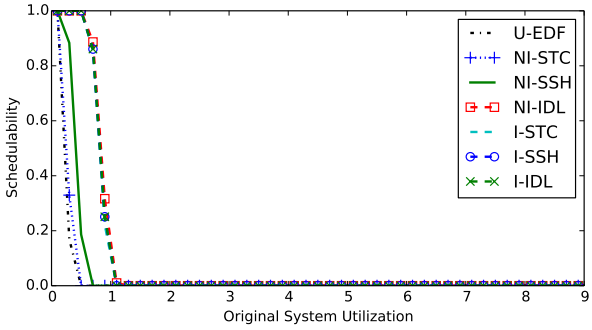
All-Mod., Long, Heavy, Heavy, Large, Heavy, Light



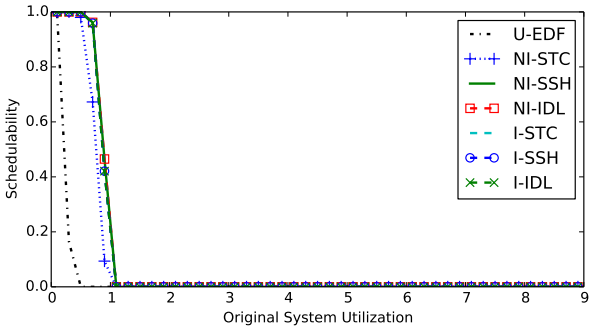
AC-Mod., Short, Light, Light, Small, Heavy, Heavy



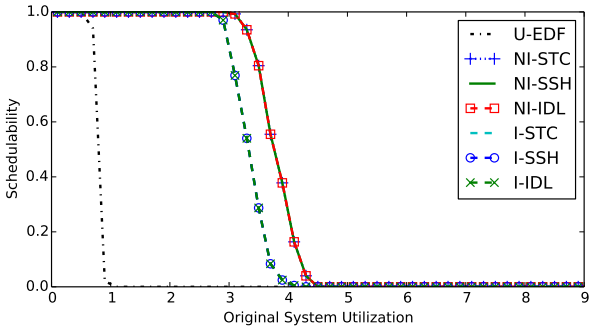
A-Heavy, Short, Heavy, Light, Large, Light, Heavy



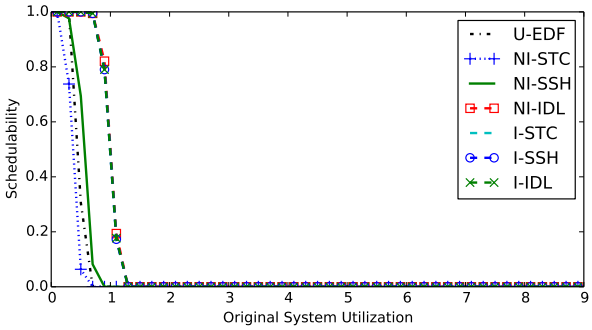
A-Heavy, Short, Light, Heavy, Large, Heavy, Heavy



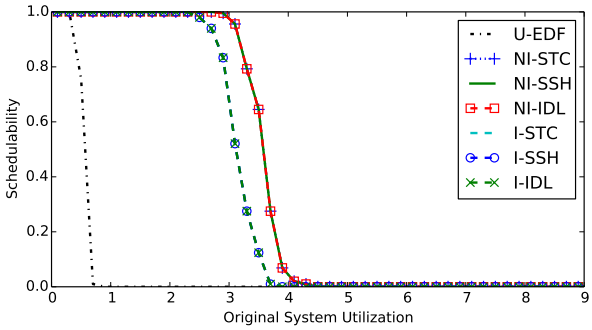
AC-Mod., Short, Light, Heavy, Large, Light, Heavy



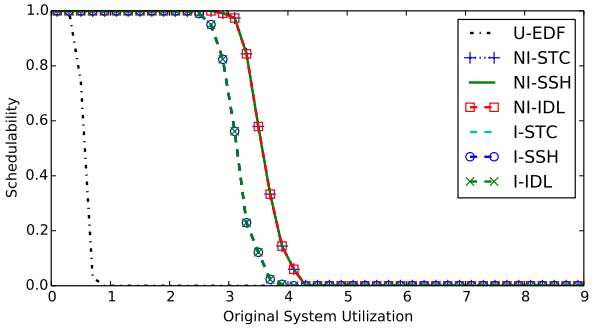
A-Heavy, Short, Heavy, Light, Mod., Heavy, Light



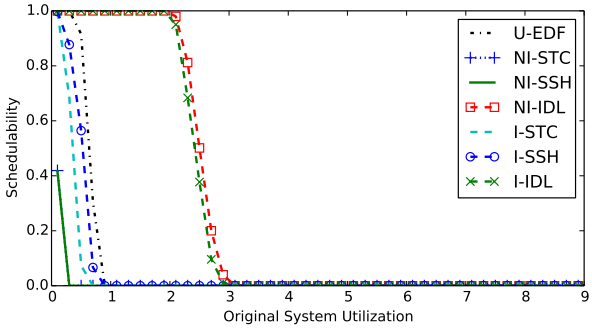
All-Mod., Short, Light, Light, Large, Heavy, Heavy



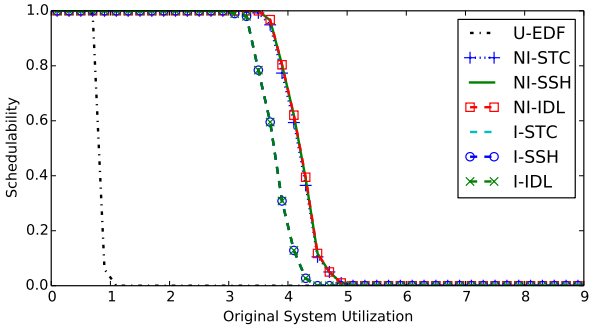
A-Heavy, Short, Heavy, Heavy, Mod., Light, Light



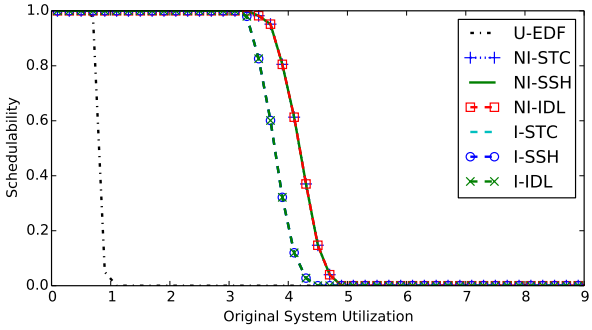
A-Heavy, Short, Heavy, Heavy, Mod., Heavy, Heavy



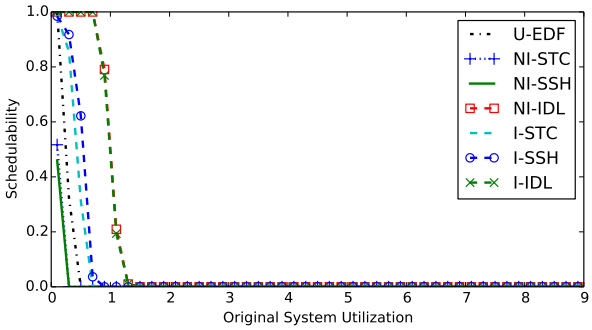
AC-Mod., Long, Light, Light, Small, Heavy, Heavy



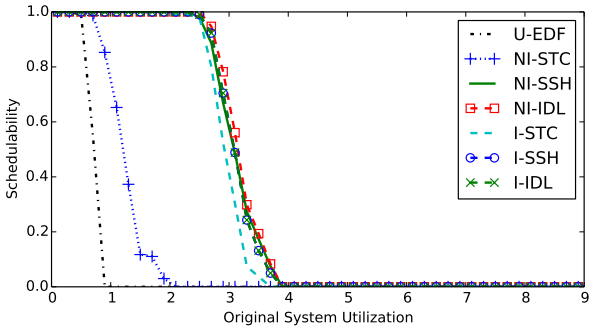
AB-Mod., Short, Heavy, Light, Large, Heavy, Light



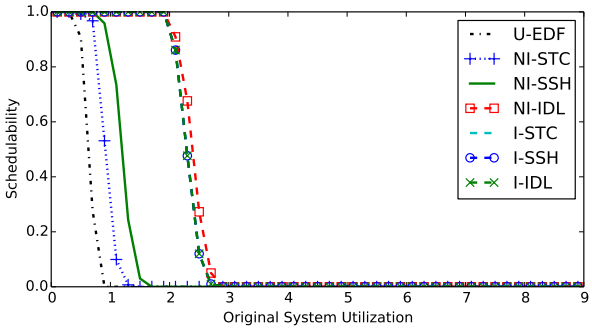
AB-Mod., Short, Heavy, Light, Mod., Heavy, Light



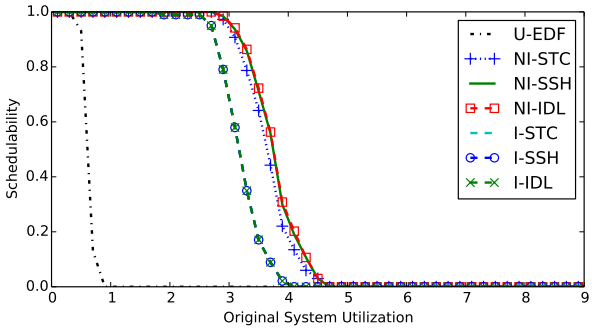
All-Mod., Short, Light, Heavy, Small, Heavy, Light



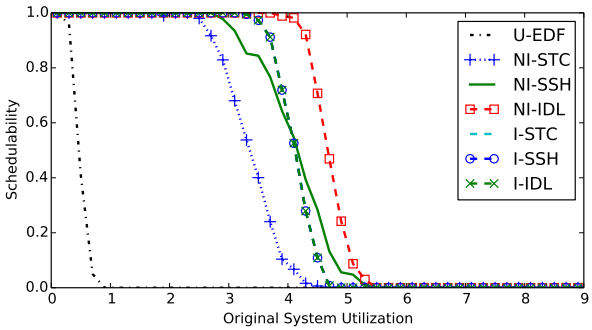
C-Heavy, Long, Light, Light, Large, Light, Heavy



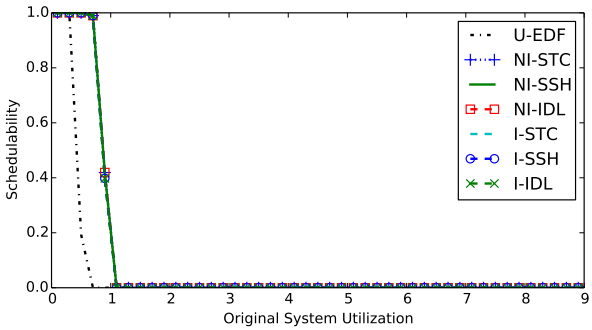
A-Heavy, Long, Light, Light, Mod., Heavy, Light



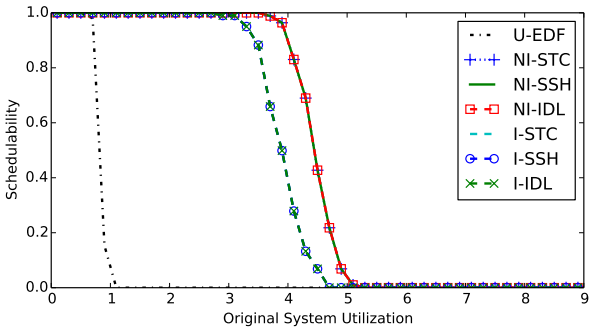
A-Heavy, Long, Heavy, Heavy, Large, Heavy, Light



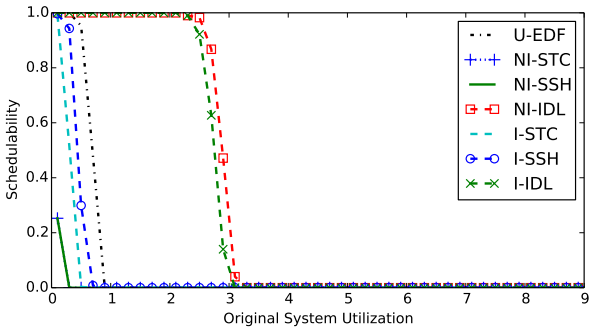
AC-Mod., Short, Heavy, Heavy, Small, Light, Light



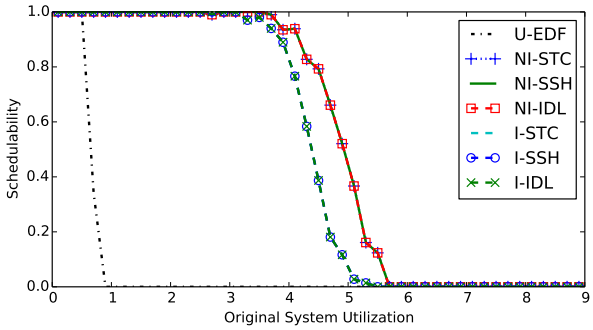
AC-Mod., Short, Light, Light, Mod., Light, Heavy



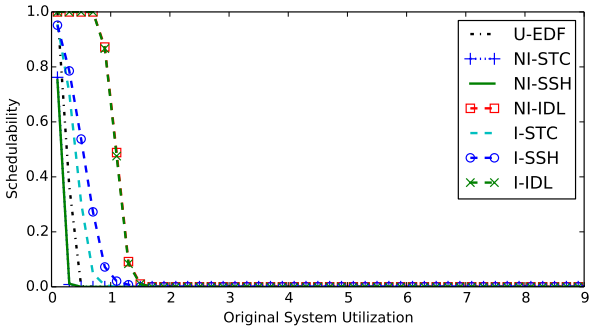
B-Heavy, Short, Heavy, Light, Large, Light, Light



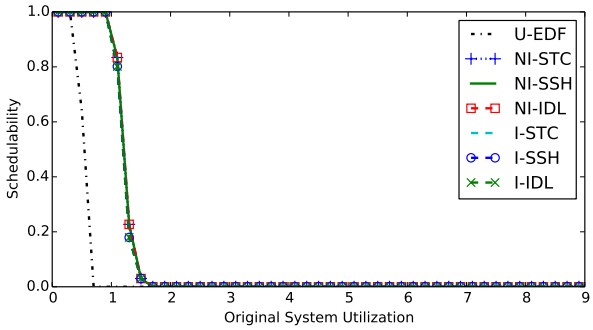
AB-Mod., Long, Light, Light, Small, Heavy, Heavy



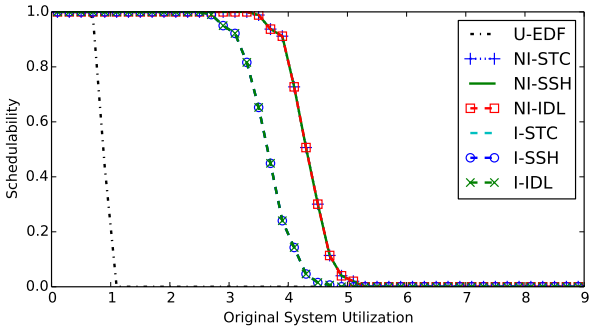
AC-Mod., Long, Heavy, Heavy, Large, Light, Light



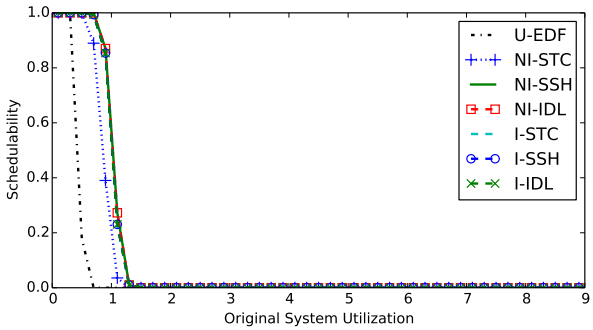
C-Heavy, Short, Light, Heavy, Small, Heavy, Heavy



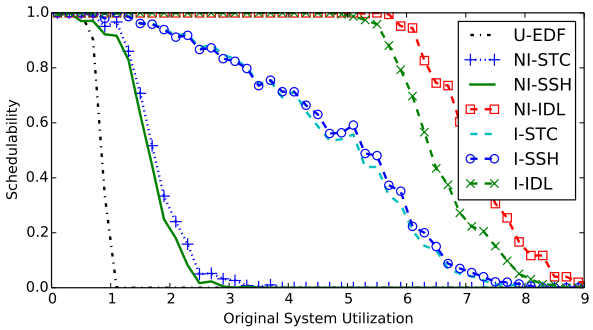
B-Heavy, Short, Light, Light, Mod., Light, Light



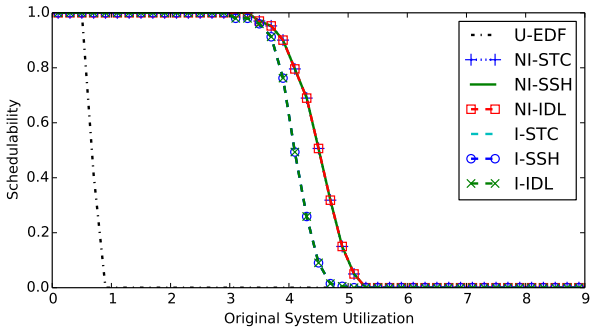
A-Heavy, Long, Heavy, Light, Mod., Heavy, Light



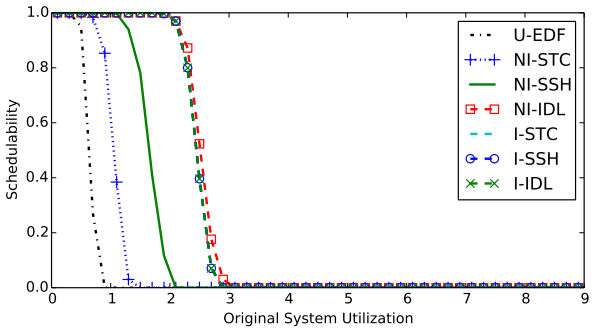
AB-Mod., Short, Light, Light, Large, Light, Light



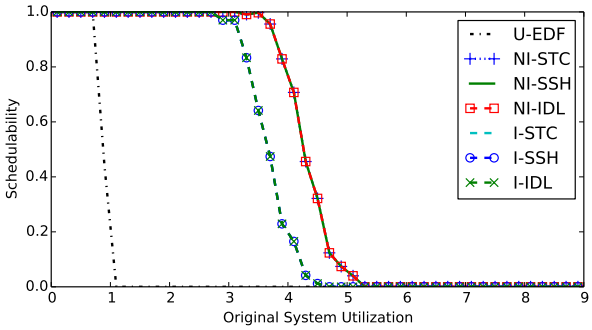
C-Heavy, Short, Heavy, Light, Small, Heavy, Light



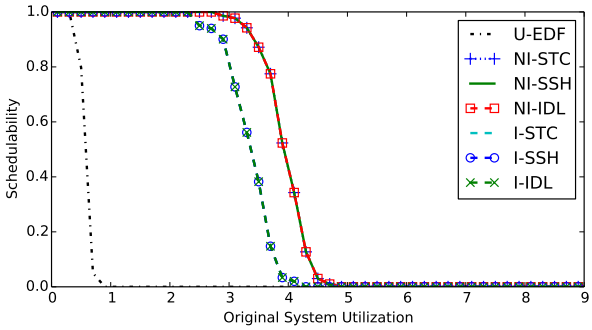
All-Mod., Long, Heavy, Heavy, Large, Light, Light



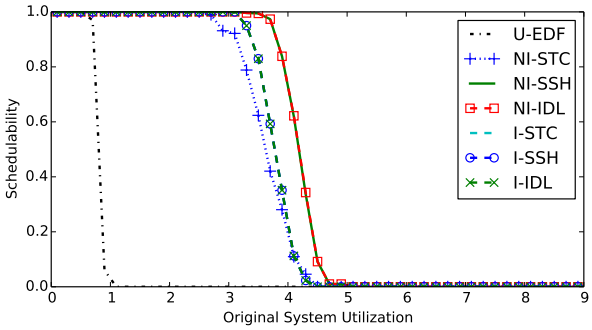
AC-Mod., Long, Light, Light, Mod., Heavy, Heavy



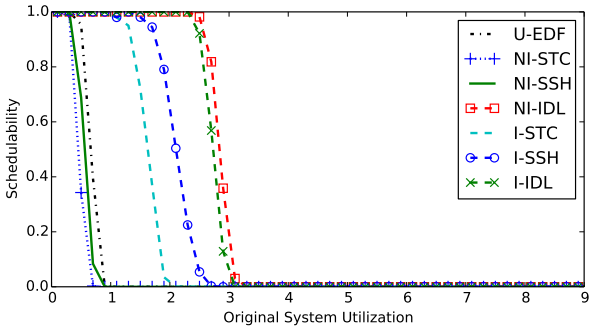
A-Heavy, Long, Heavy, Light, Mod., Heavy, Heavy



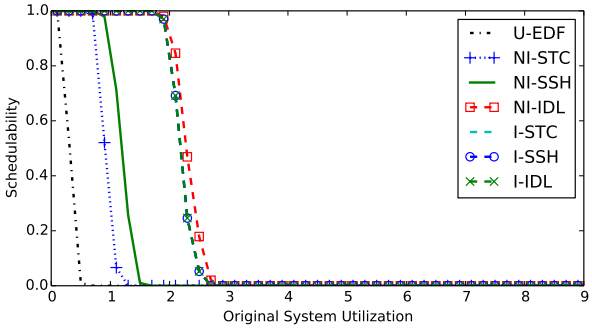
B-Heavy, Short, Heavy, Heavy, Large, Heavy, Light



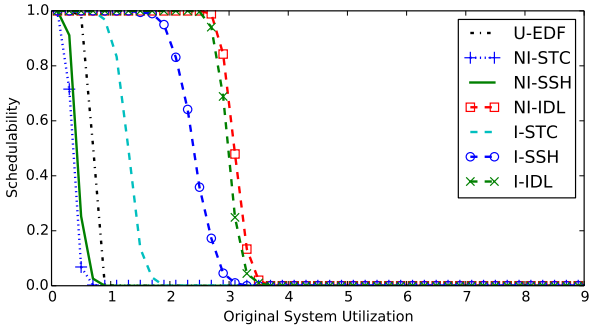
AB-Mod., Short, Heavy, Light, Large, Heavy, Heavy



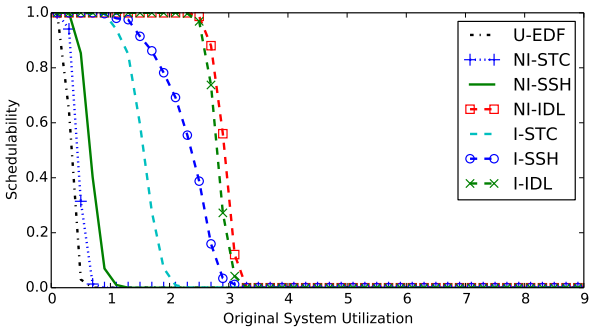
All-Mod., Long, Light, Light, Large, Heavy, Light



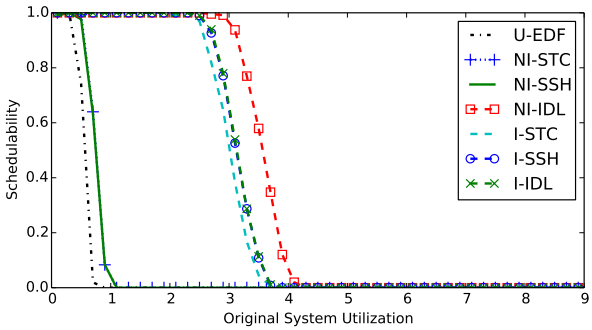
A-Heavy, Long, Light, Heavy, Mod., Heavy, Light



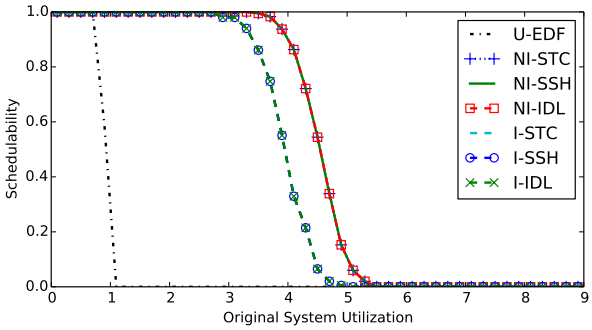
BC-Mod., Long, Light, Light, Small, Light, Light



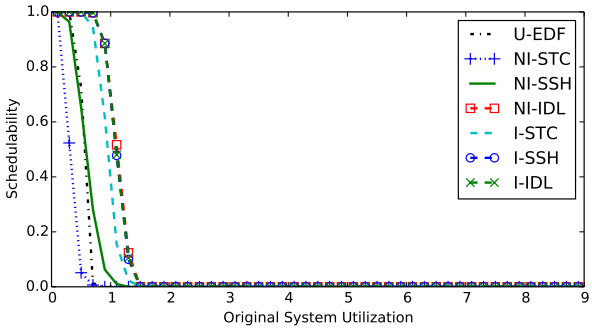
BC-Mod., Long, Light, Heavy, Large, Heavy, Heavy



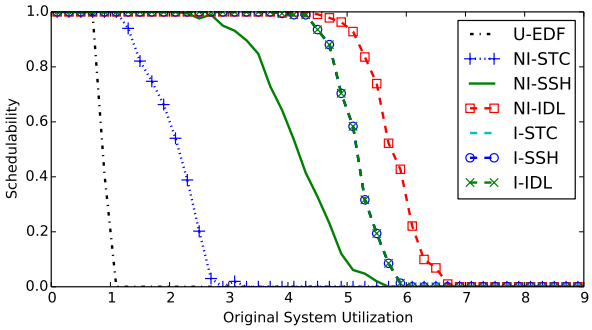
A-Heavy, Short, Heavy, Heavy, Small, Heavy, Heavy



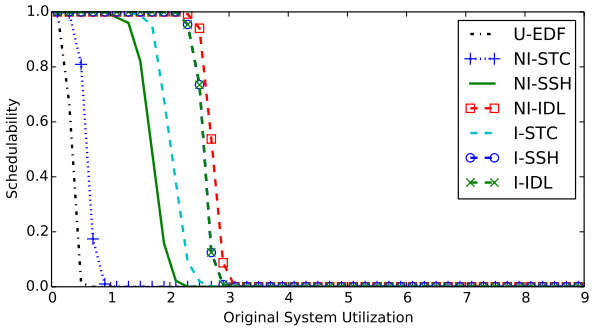
B-Heavy, Long, Heavy, Light, Large, Light, Light



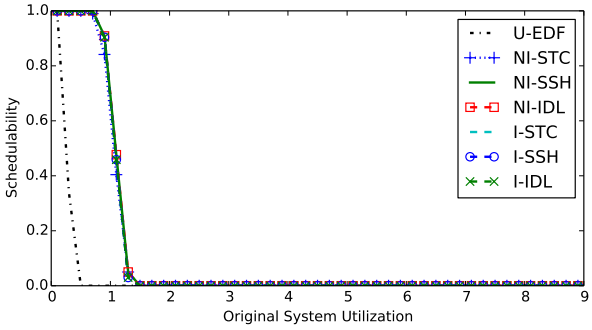
C-Heavy, Short, Light, Light, Small, Light, Heavy



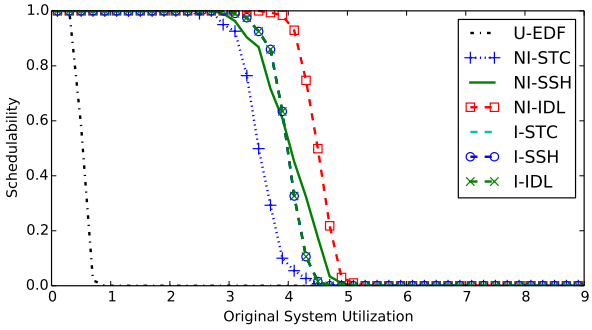
AC-Mod., Long, Heavy, Light, Small, Light, Heavy



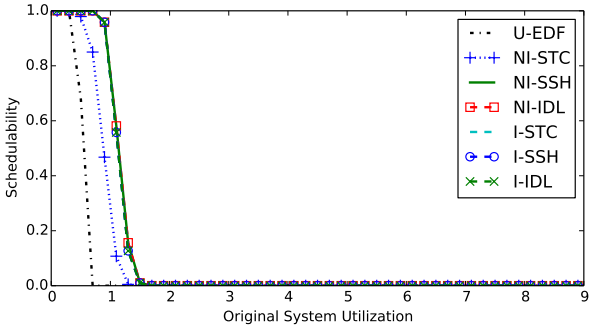
AB-Mod., Long, Light, Heavy, Large, Light, Heavy



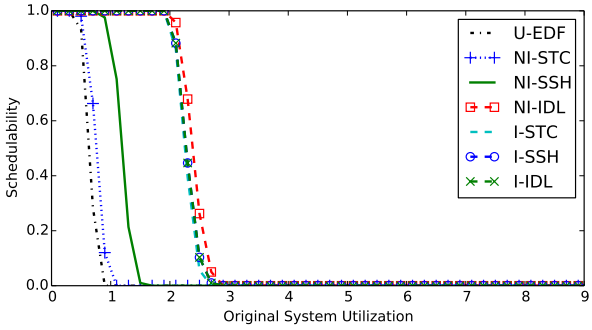
C-Heavy, Short, Light, Heavy, Large, Light, Heavy



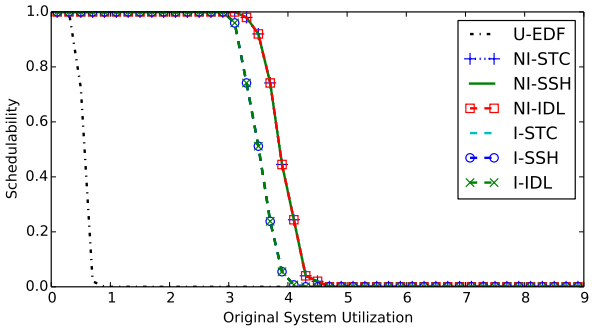
All-Mod., Short, Heavy, Heavy, Small, Light, Light



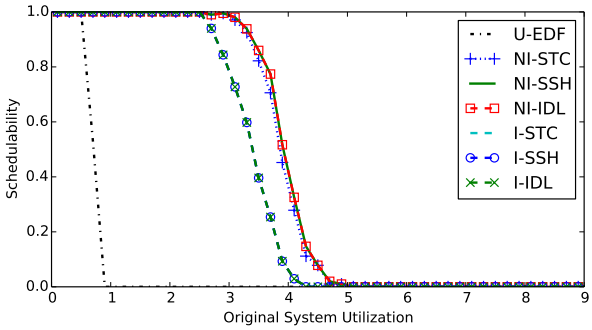
BC-Mod., Short, Light, Light, Large, Light, Heavy



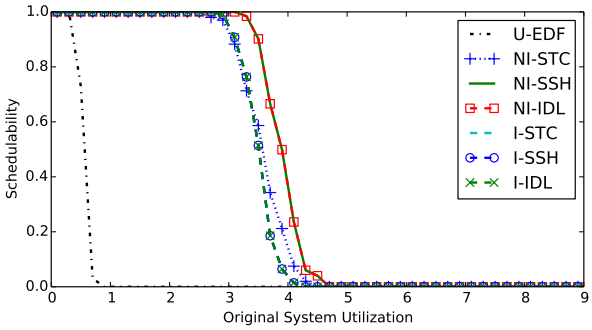
A-Heavy, Long, Light, Light, Mod., Heavy, Heavy



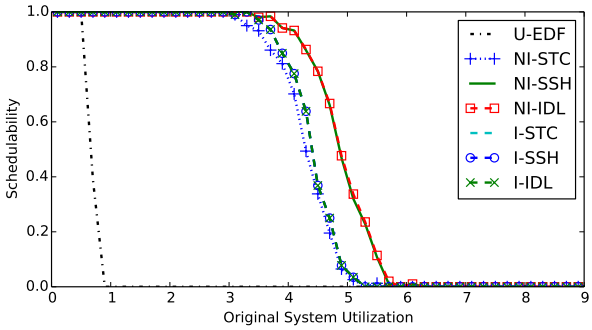
AB-Mod., Short, Heavy, Heavy, Mod., Light, Heavy



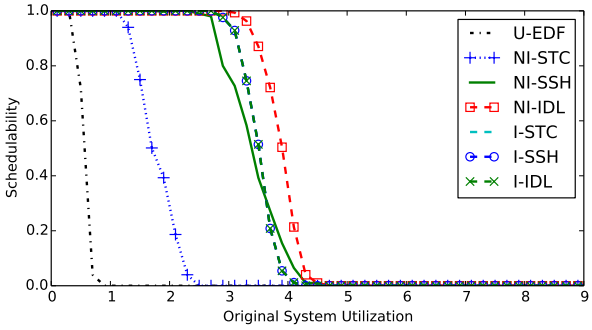
B-Heavy, Long, Heavy, Heavy, Large, Heavy, Heavy



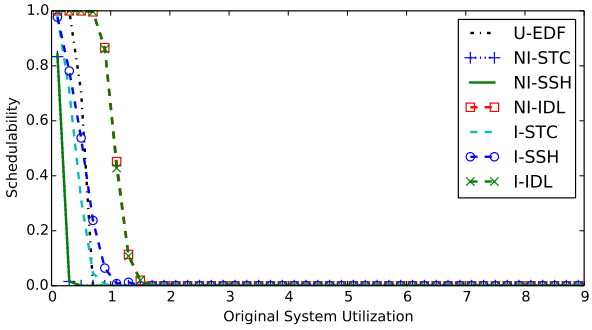
AB-Mod., Short, Heavy, Heavy, Large, Heavy, Heavy



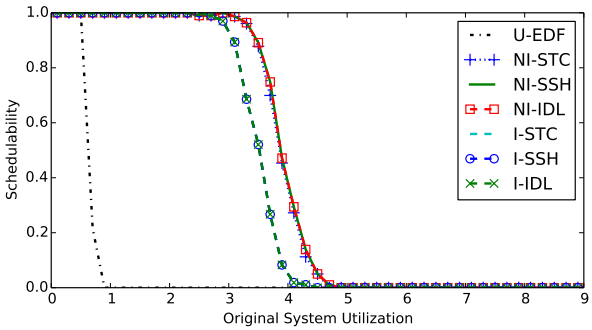
AC-Mod., Long, Heavy, Heavy, Large, Heavy, Heavy



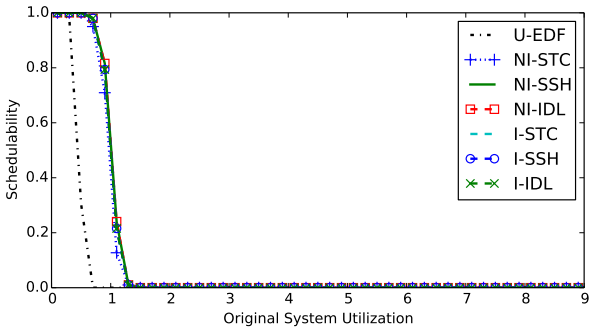
AB-Mod., Short, Heavy, Heavy, Small, Light, Heavy



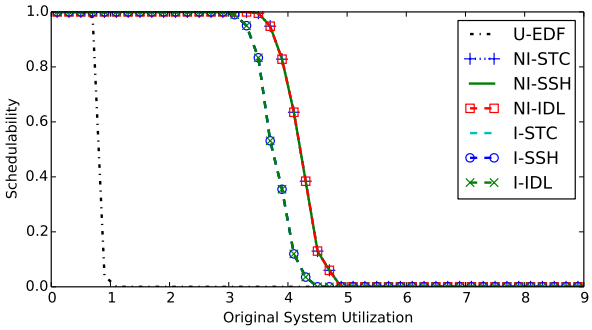
C-Heavy, Short, Light, Light, Small, Heavy, Heavy



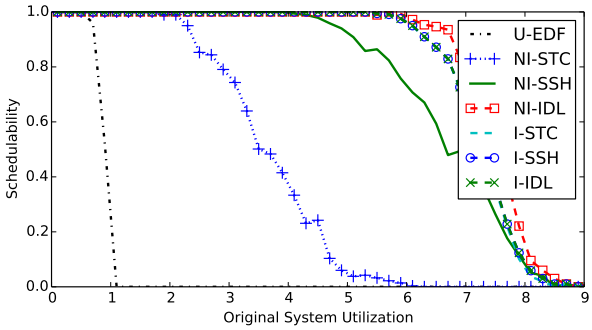
AB-Mod., Long, Heavy, Heavy, Large, Heavy, Light



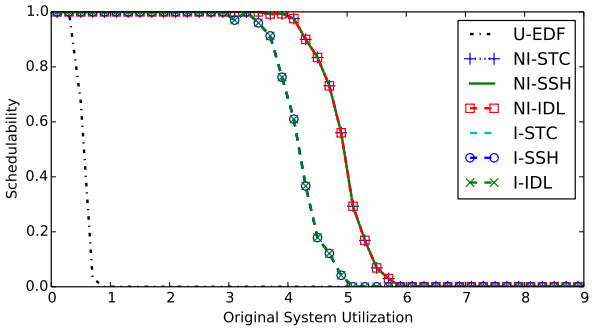
All-Mod., Short, Light, Light, Mod., Heavy, Heavy



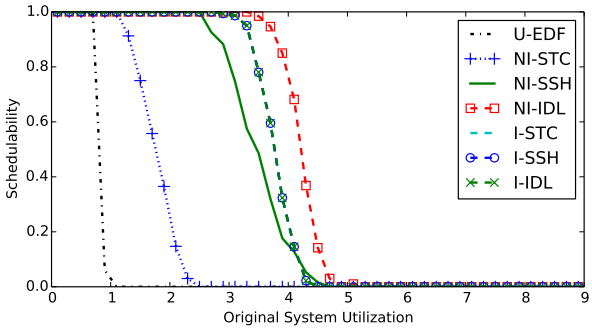
AB-Mod., Short, Heavy, Light, Large, Light, Light



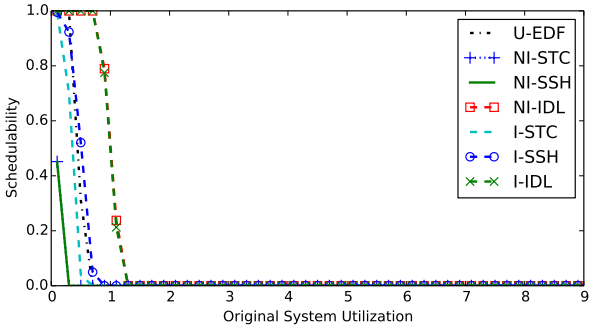
C-Heavy, Long, Heavy, Light, Small, Light, Heavy



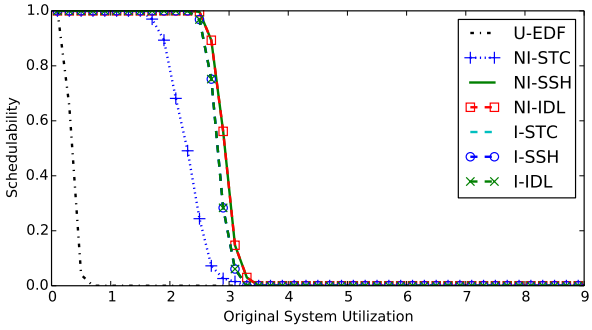
BC-Mod., Short, Heavy, Heavy, Mod., Light, Light



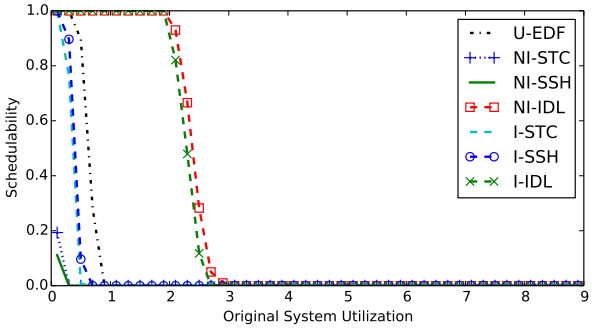
AB-Mod., Short, Heavy, Light, Small, Light, Heavy



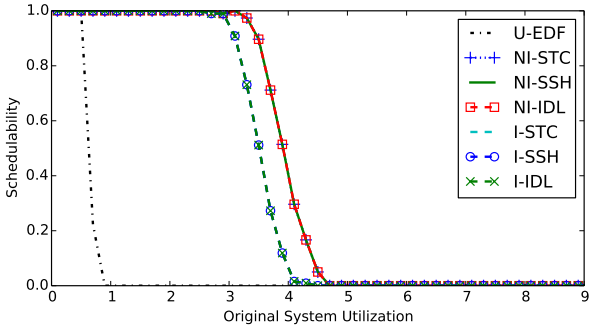
All-Mod., Short, Light, Light, Small, Heavy, Heavy



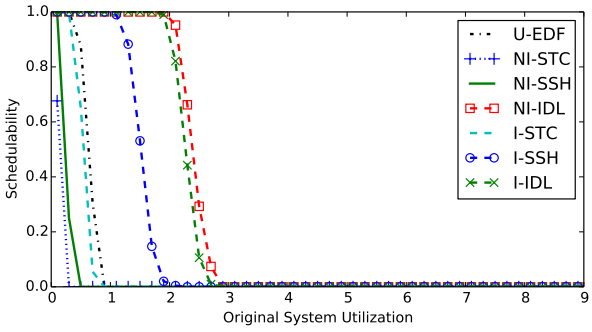
BC-Mod., Long, Light, Heavy, Mod., Light, Heavy



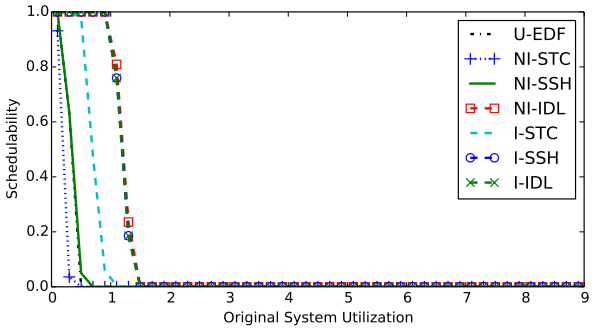
A-Heavy, Long, Light, Light, Small, Heavy, Light



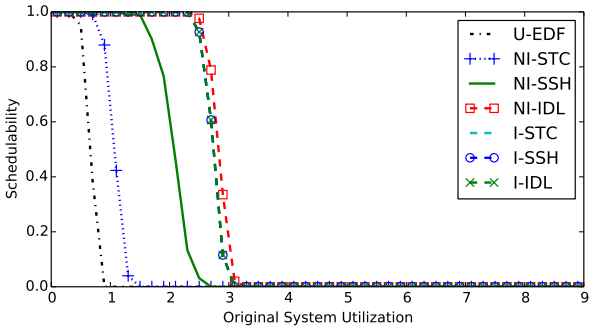
AB-Mod., Long, Heavy, Heavy, Mod., Heavy, Heavy



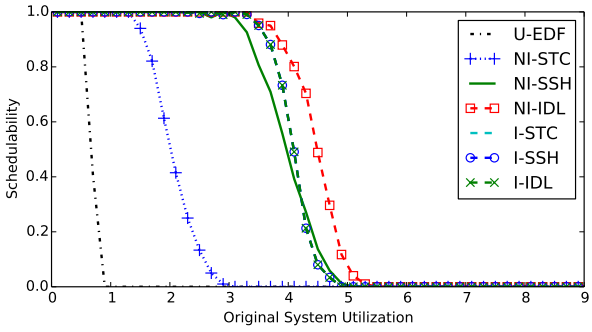
A-Heavy, Long, Light, Light, Small, Light, Heavy



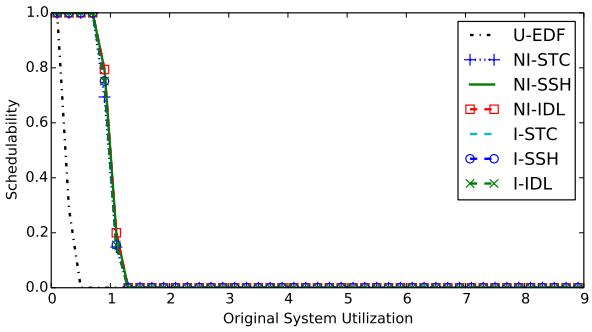
B-Heavy, Short, Light, Heavy, Small, Light, Heavy



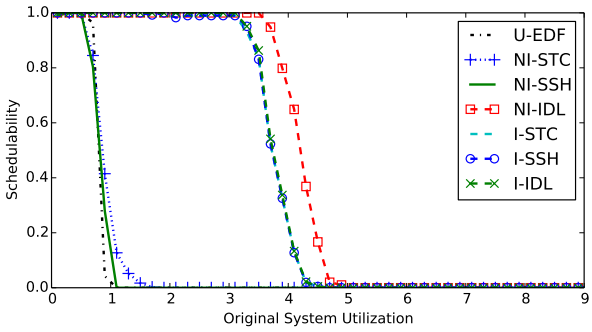
All-Mod., Long, Light, Light, Large, Light, Light



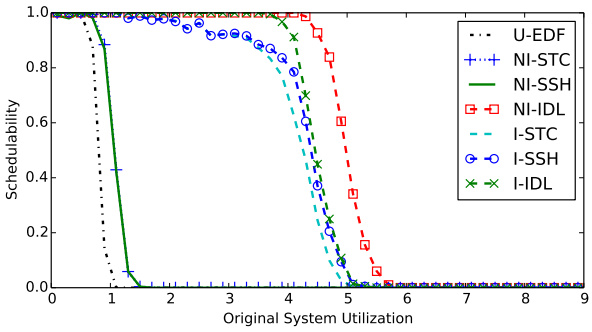
All-Mod., Long, Heavy, Heavy, Small, Light, Heavy



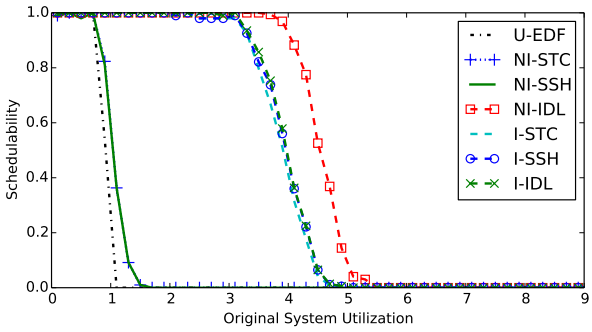
All-Mod., Short, Light, Heavy, Large, Light, Light



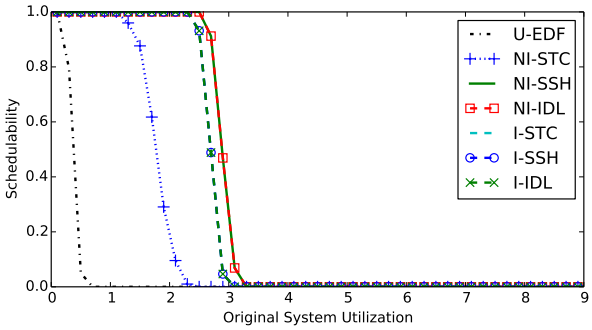
AB-Mod., Short, Heavy, Light, Small, Heavy, Light



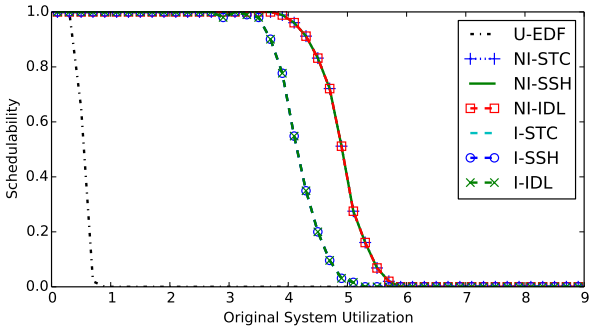
AC-Mod., Short, Heavy, Light, Small, Heavy, Heavy



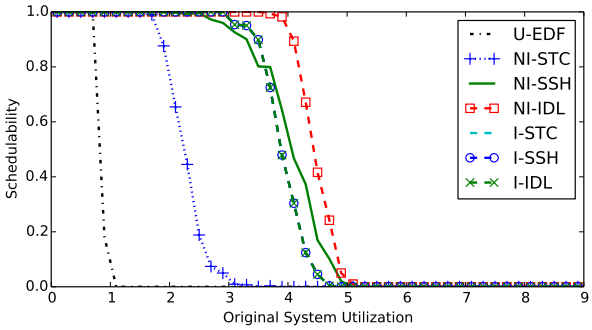
B-Heavy, Long, Heavy, Light, Small, Heavy, Heavy



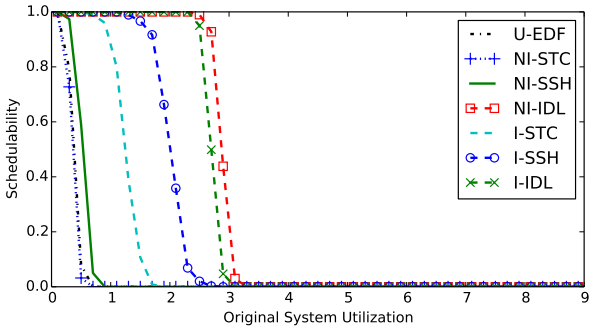
B-Heavy, Long, Light, Heavy, Mod., Light, Heavy



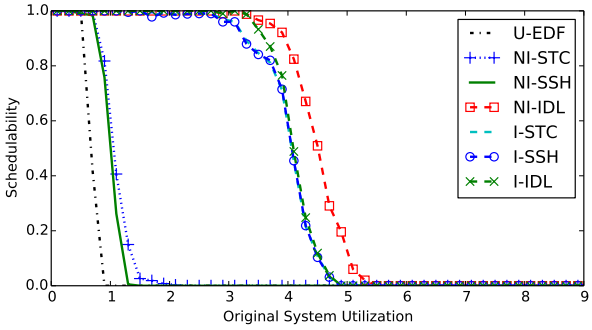
BC-Mod., Short, Heavy, Heavy, Large, Light, Light



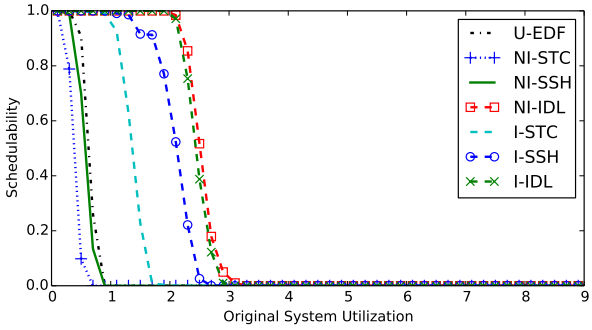
B-Heavy, Short, Heavy, Light, Small, Light, Heavy



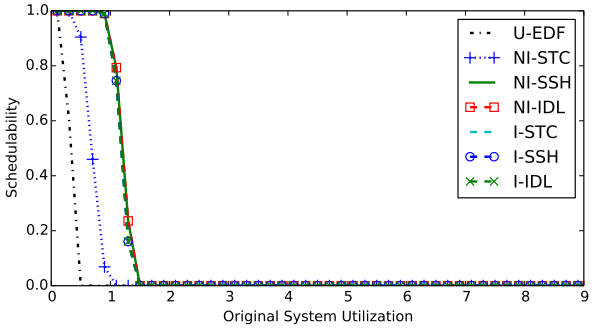
B-Heavy, Long, Light, Heavy, Large, Heavy, Heavy



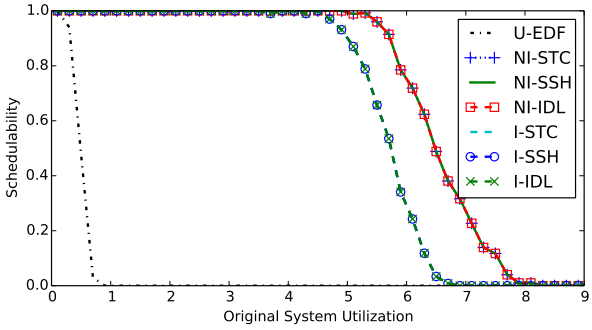
All-Mod., Long, Heavy, Heavy, Small, Heavy, Light



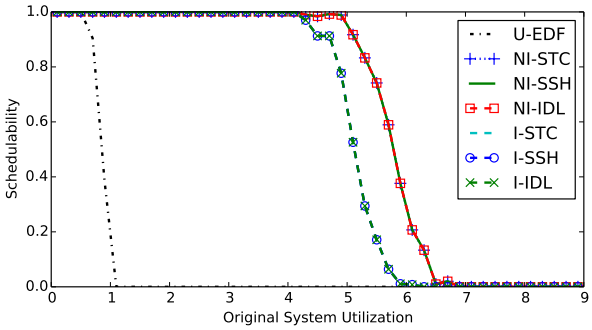
AC-Mod., Long, Light, Light, Large, Heavy, Heavy



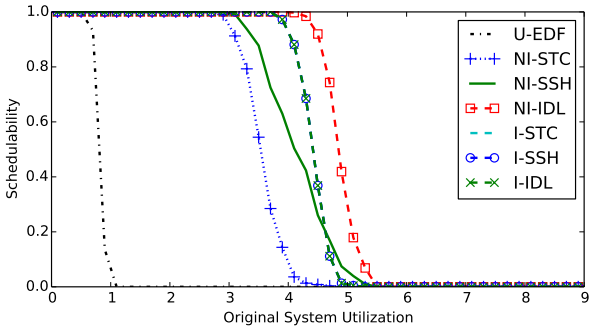
B-Heavy, Short, Light, Heavy, Large, Light, Heavy



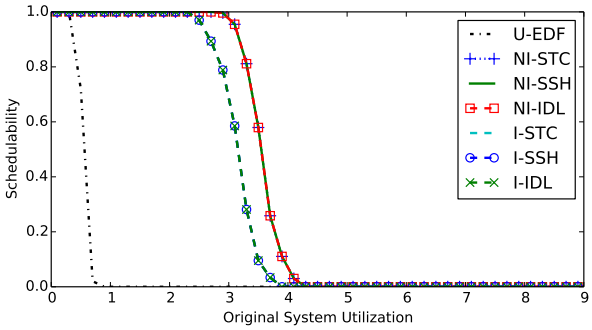
C-Heavy, Short, Heavy, Heavy, Mod., Light, Heavy



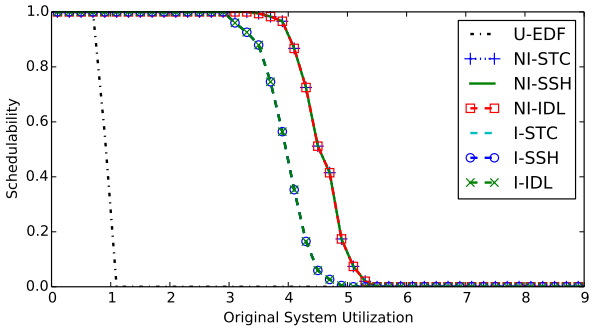
AC-Mod., Long, Heavy, Light, Mod., Heavy, Heavy



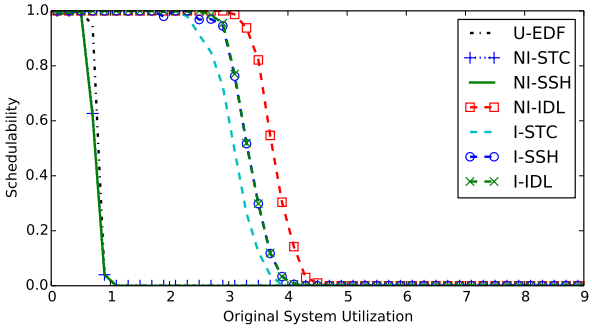
All-Mod., Short, Heavy, Light, Small, Light, Light



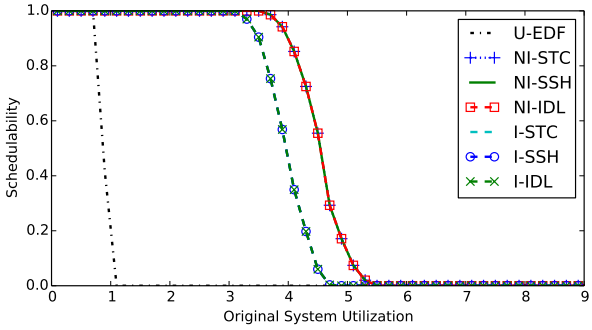
A-Heavy, Short, Heavy, Heavy, Mod., Heavy, Light



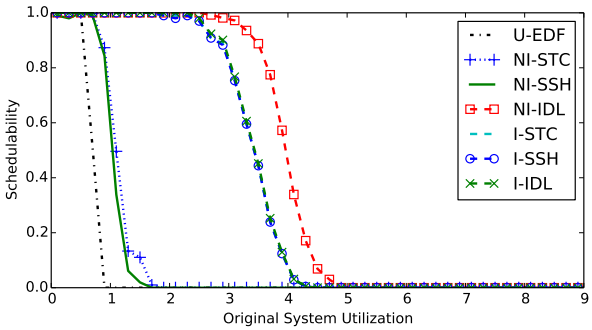
B-Heavy, Long, Heavy, Light, Mod., Light, Light



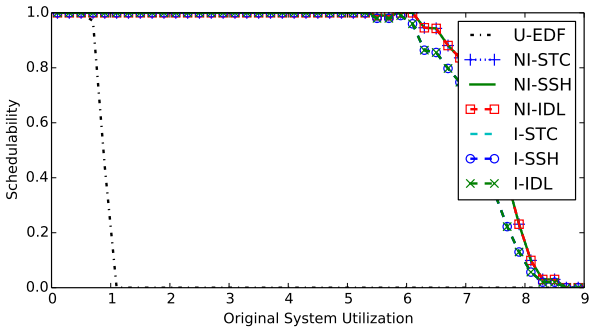
A-Heavy, Short, Heavy, Light, Small, Heavy, Heavy



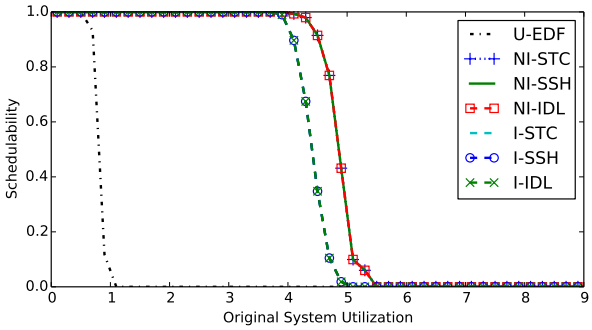
AB-Mod., Long, Heavy, Light, Mod., Light, Heavy



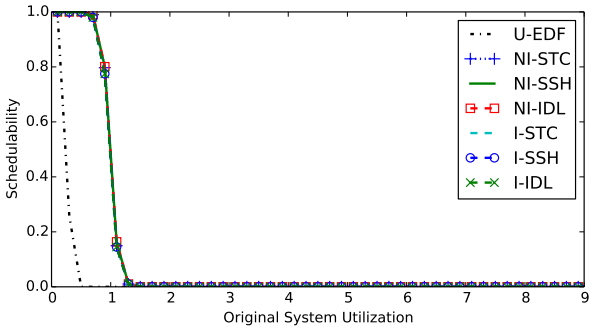
B-Heavy, Long, Heavy, Heavy, Small, Heavy, Light



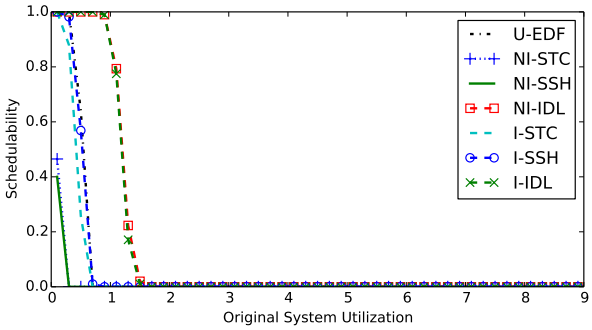
C-Heavy, Long, Heavy, Light, Large, Light, Heavy



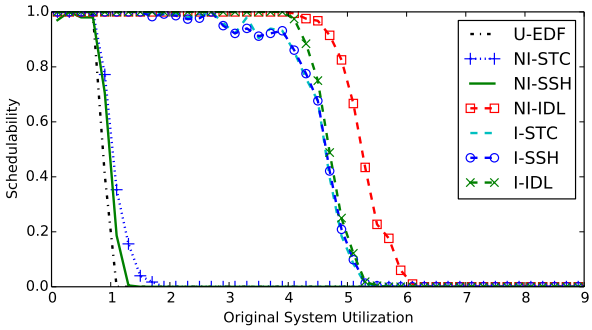
All-Mod., Short, Heavy, Light, Mod., Heavy, Heavy



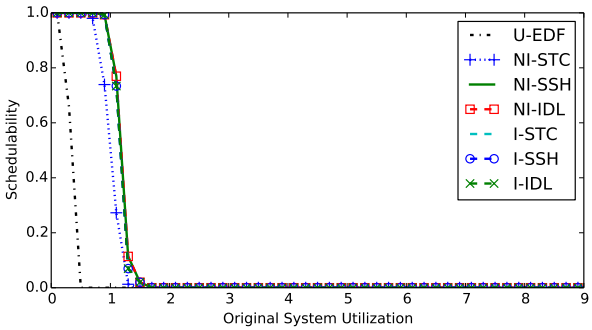
All-Mod., Short, Light, Heavy, Mod., Heavy, Light



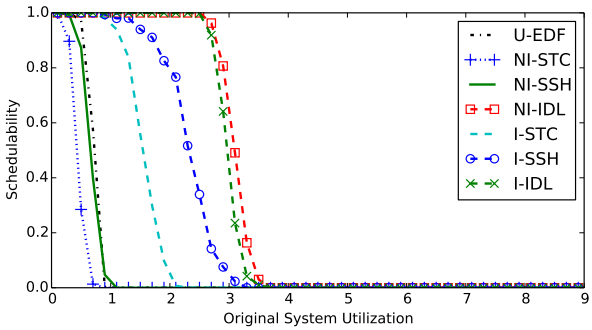
B-Heavy, Short, Light, Light, Small, Heavy, Light



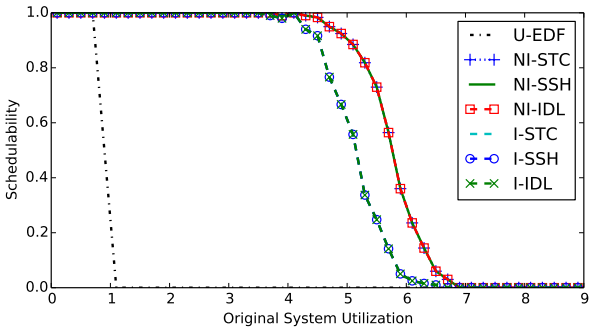
All-Mod., Long, Heavy, Light, Small, Heavy, Light



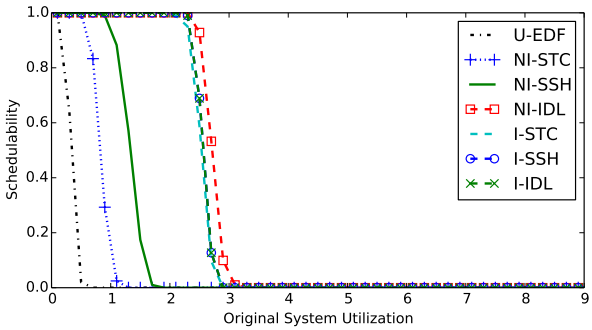
B-Heavy, Short, Light, Heavy, Large, Light, Light



BC-Mod., Long, Light, Light, Large, Heavy, Heavy

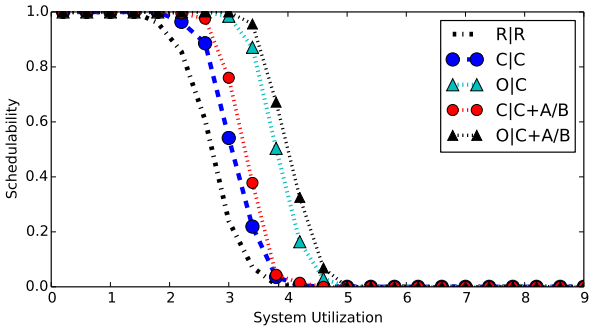


BC-Mod., Long, Heavy, Light, Mod., Heavy, Light

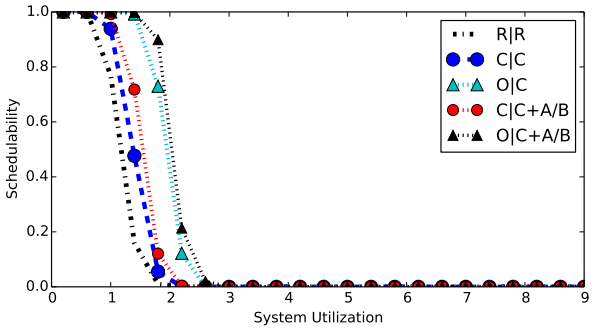


AB-Mod., Long, Light, Heavy, Mod., Heavy, Heavy

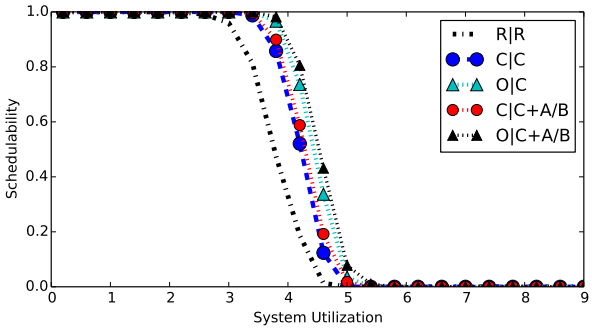
APPENDIX D: SCHEDULABILITY GRAPHS FOR THE STUDY DESCRIBED IN SECTION 4.3



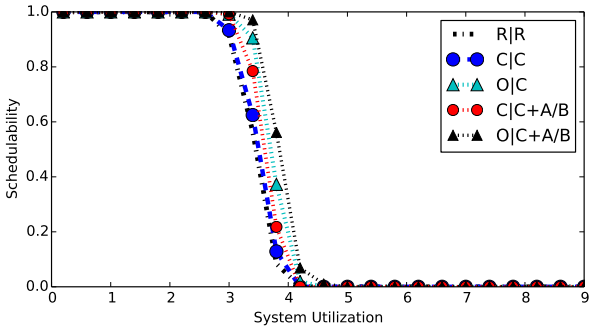
C-Light, Mod., Med., Heavy, Small, Low, Few



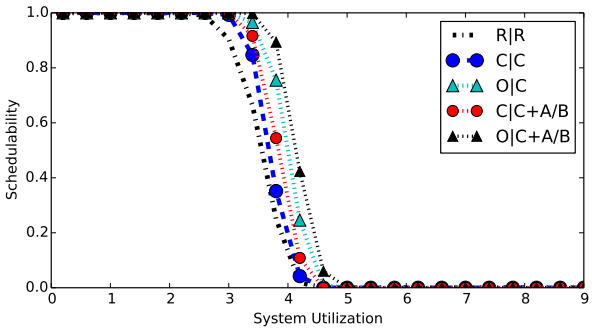
All-Mod., Short, Light, Heavy, Large, High, Few



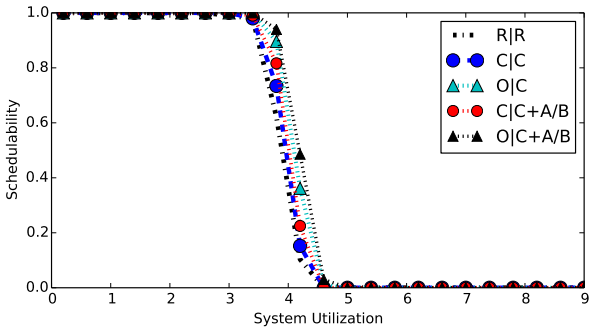
C-Light, Long, Med., Heavy, Small, Med., Many



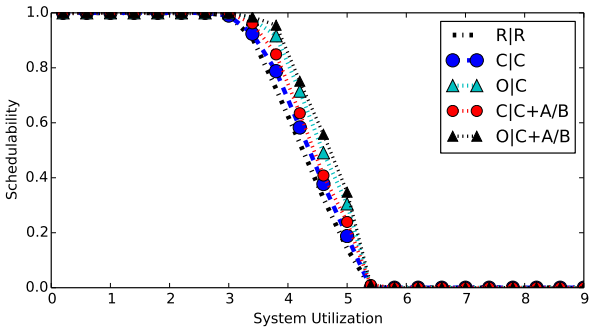
All-Mod., Mod., Heavy, Light, Small, Med., Many



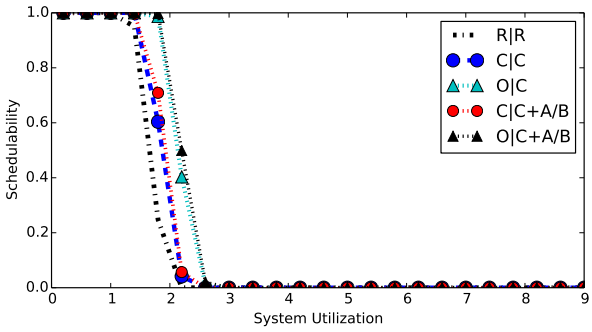
C-Light, Mod., Heavy, Light, Small, Low, Few



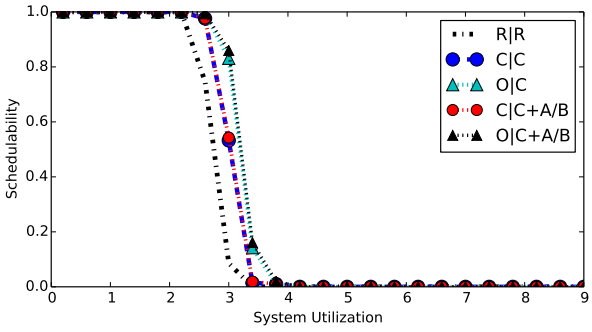
C-Light, Short, Heavy, Light, Small, High, Many



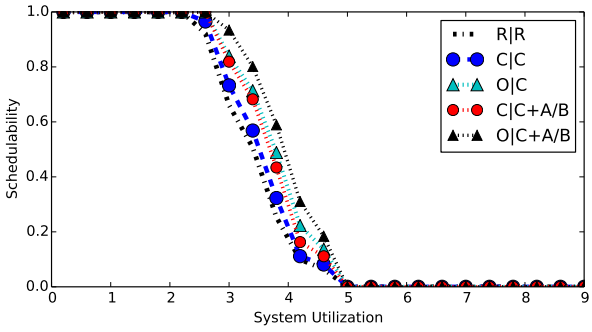
C-Heavy, Short, Heavy, Light, Large, Med., Few



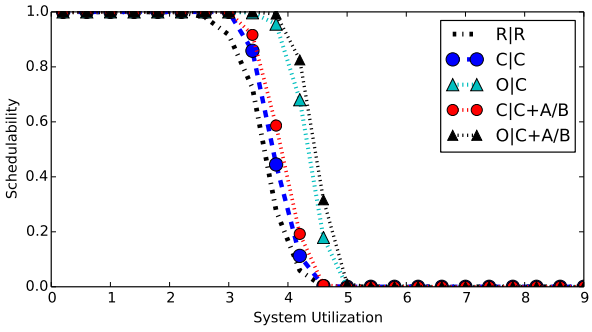
C-Light, Short, Light, Heavy, Small, Med., Few



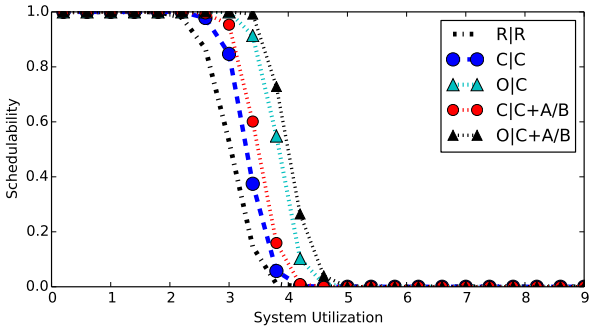
All-Mod., Long, Light, Light, Small, Low, Few



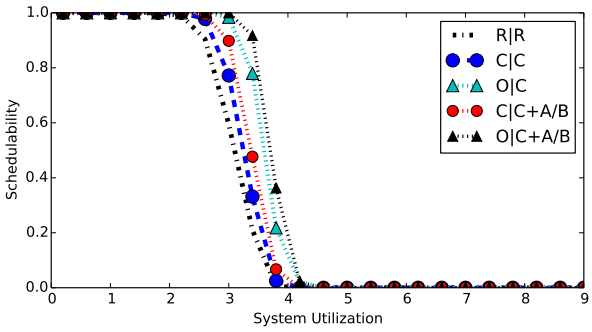
C-Heavy, Mod., Heavy, Light, Large, High, Few



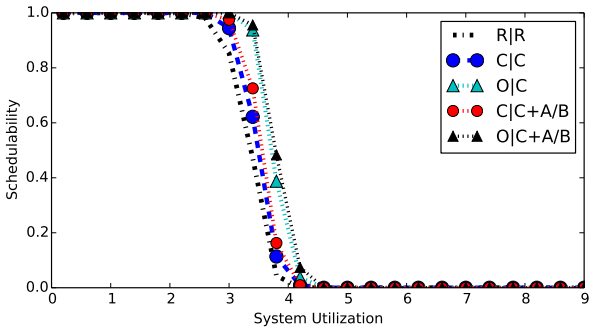
All-Mod., Short, Med., Heavy, Small, Med., Few



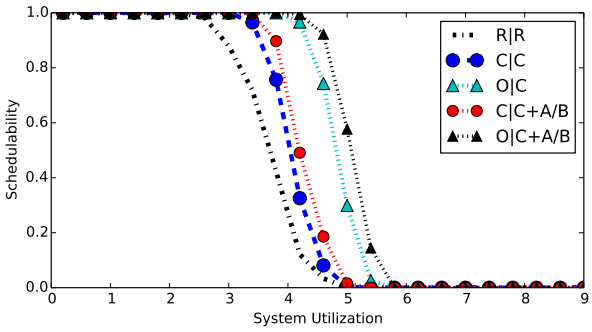
C-Light, Short, Med., Heavy, Small, High, Many



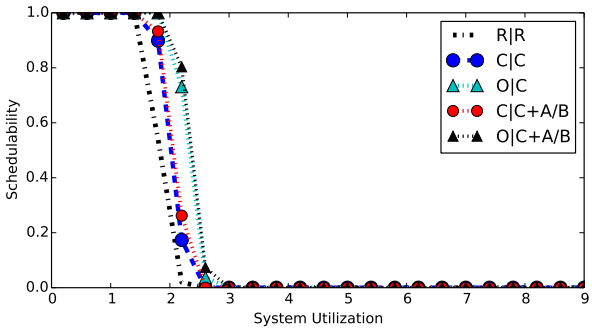
C-Light, Mod., Heavy, Heavy, Small, Med., Few



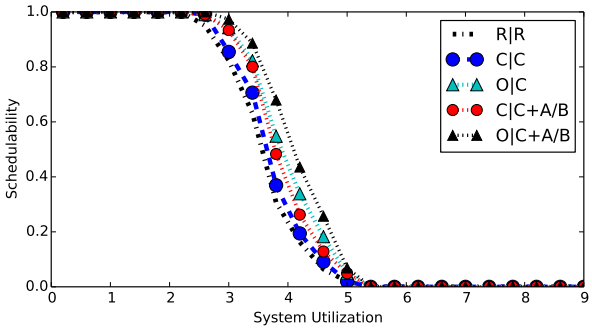
C-Light, Short, Heavy, Heavy, Small, High, Many



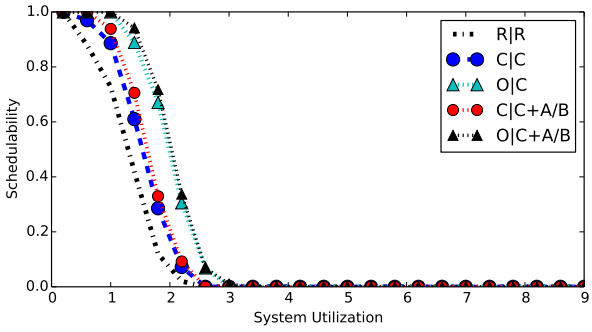
All-Mod., Mod., Med., Light, Small, Med., Many



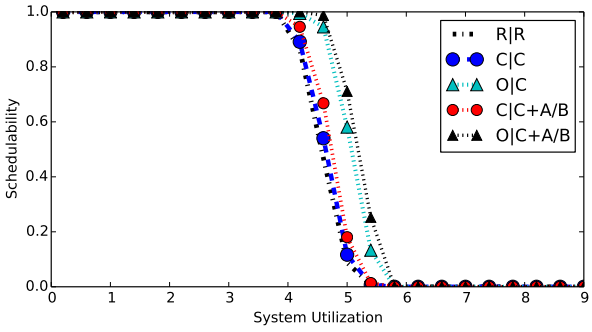
C-Light, Short, Light, Light, Small, Med., Few



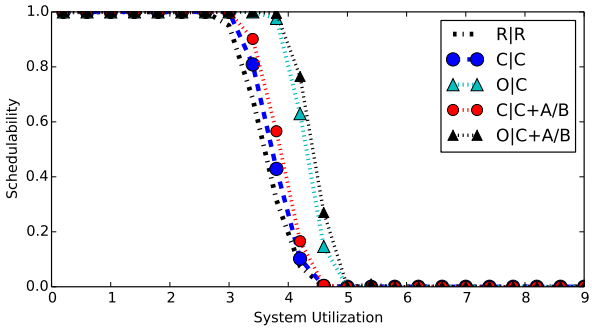
C-Heavy, Mod., Heavy, Light, Large, Med., Many



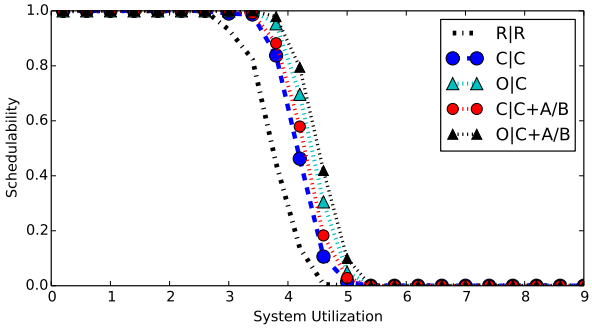
C-Heavy, Mod., Light, Light, Large, Low, Few



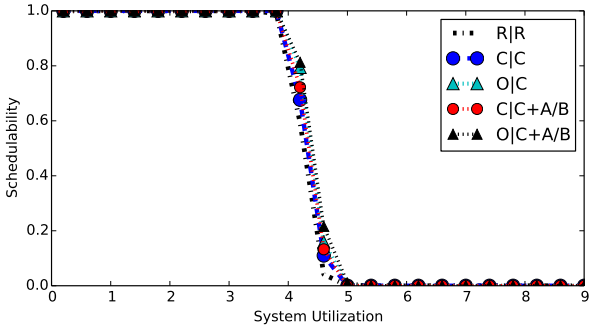
All-Mod., Long, Med., Light, Large, High, Few



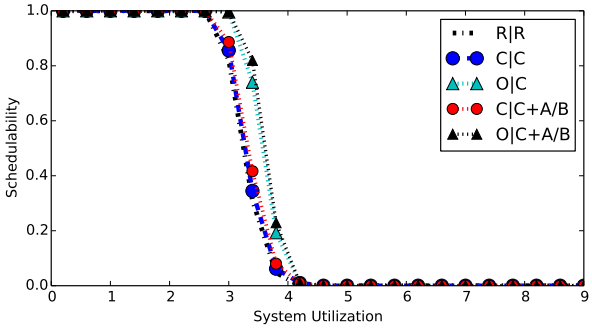
All-Mod., Short, Med., Heavy, Small, Med., Many



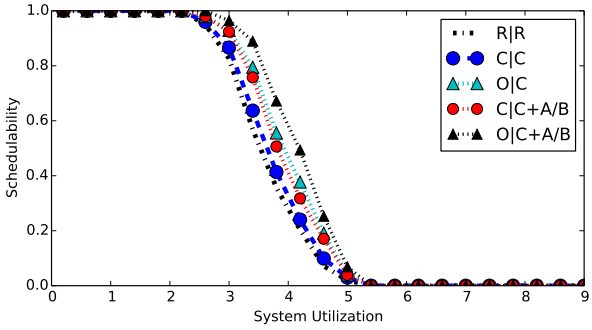
C-Light, Long, Med., Heavy, Small, High, Few



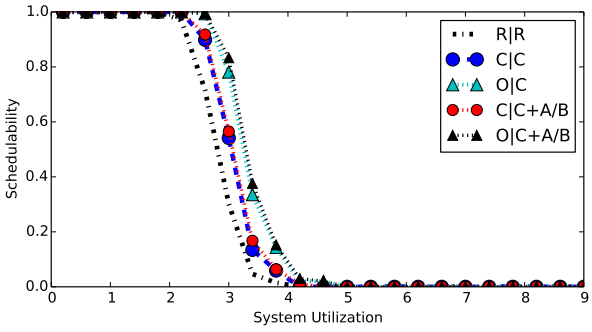
C-Light, Long, Heavy, Light, Small, Med., Few



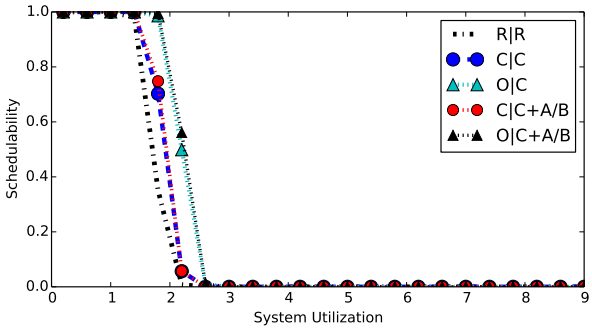
All-Mod., Short, Heavy, Heavy, Small, High, Few



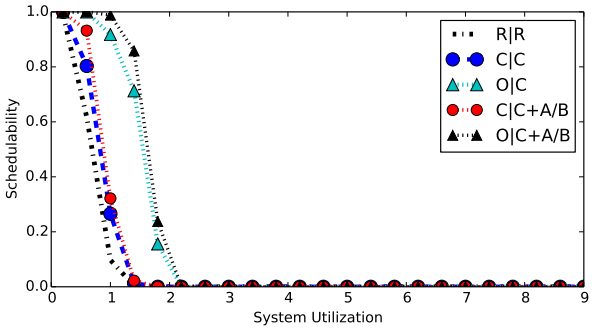
C-Heavy, Mod., Heavy, Light, Large, Med., Few



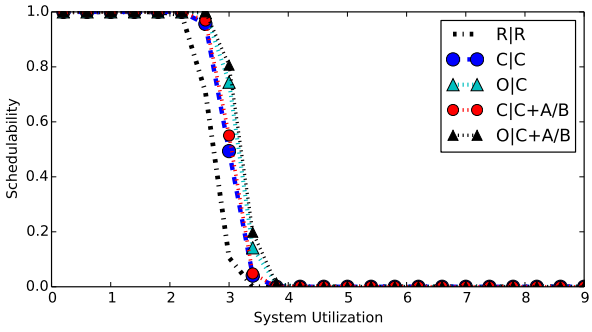
C-Heavy, Long, Light, Heavy, Large, Med., Few



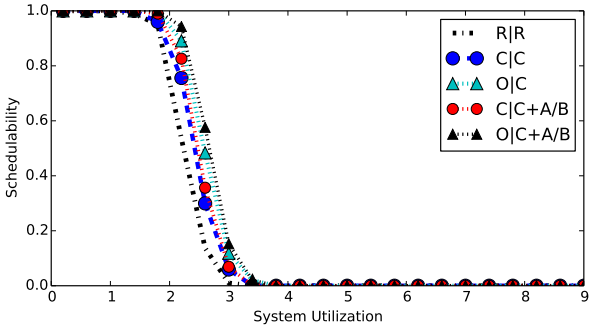
C-Light, Short, Light, Heavy, Small, Low, Few



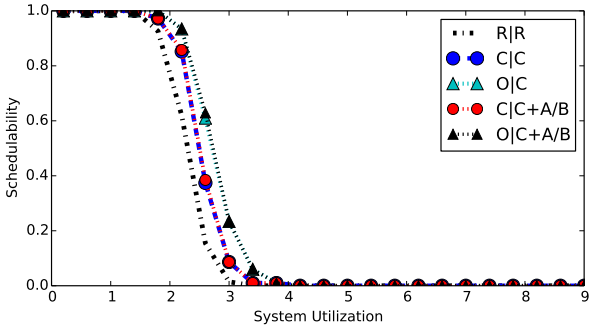
C-Light, Mod., Light, Heavy, Large, Med., Few



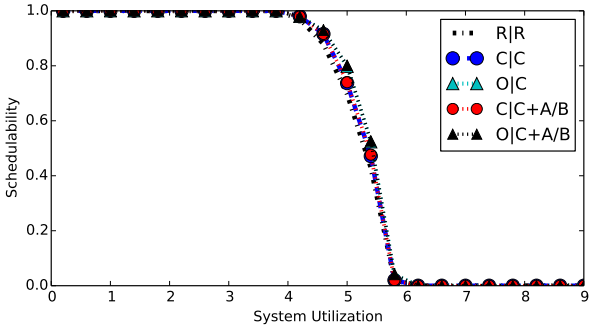
All-Mod., Long, Light, Light, Small, Med., Few



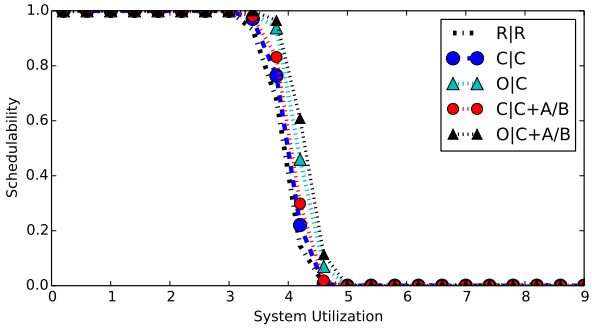
C-Heavy, Short, Light, Light, Small, High, Few



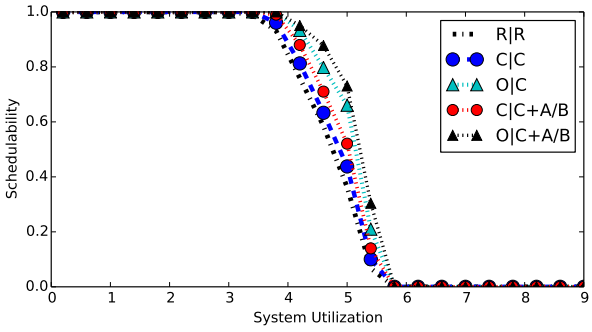
C-Heavy, Short, Light, Light, Small, Low, Many



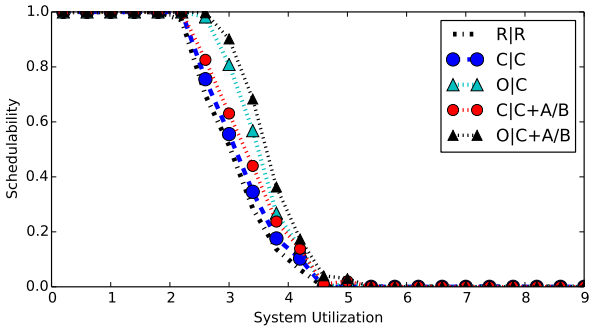
C-Heavy, Long, Heavy, Light, Large, High, Few



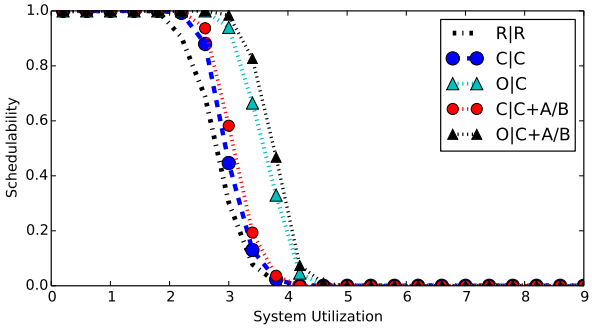
C-Light, Short, Heavy, Light, Small, Low, Many



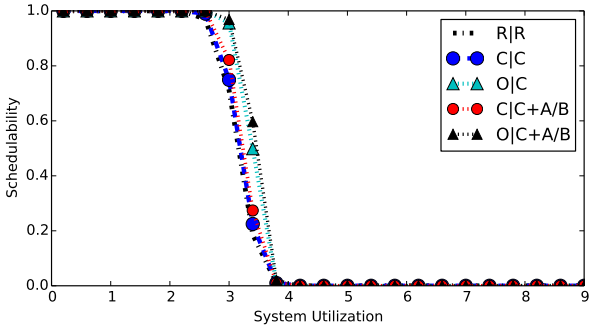
C-Heavy, Short, Heavy, Light, Small, Low, Many



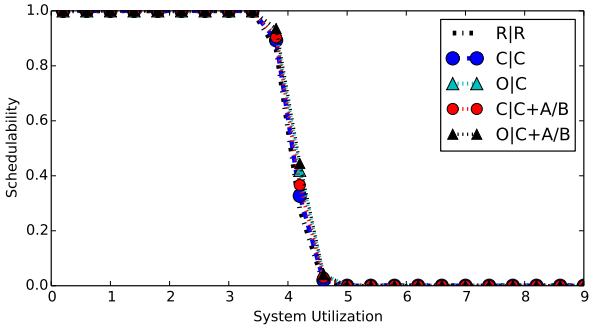
C-Heavy, Mod., Heavy, Heavy, Large, High, Few



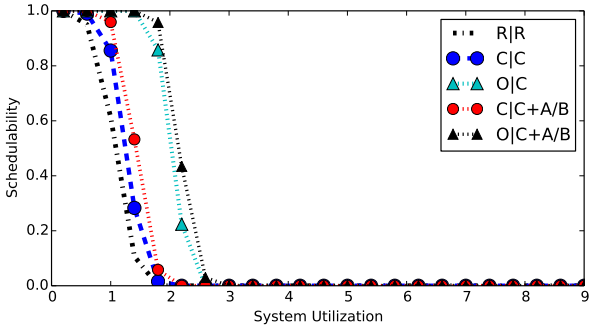
All-Mod., Short, Med., Heavy, Large, High, Few



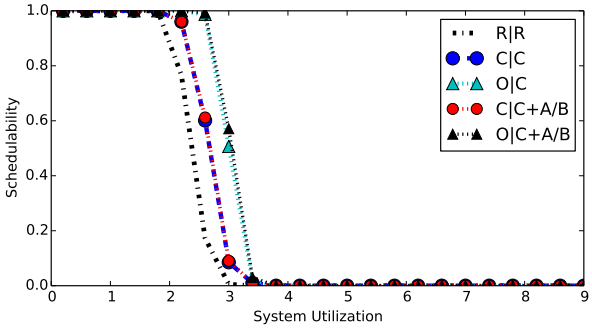
All-Mod., Short, Heavy, Heavy, Large, Low, Many



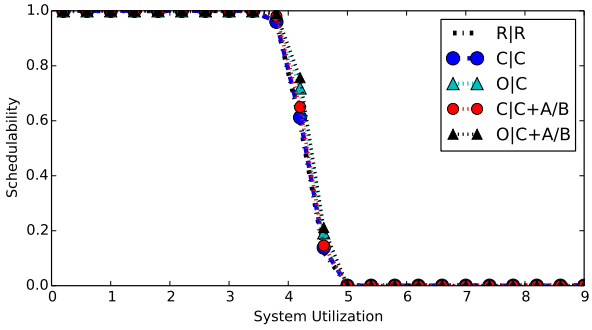
All-Mod., Long, Heavy, Heavy, Large, Low, Many



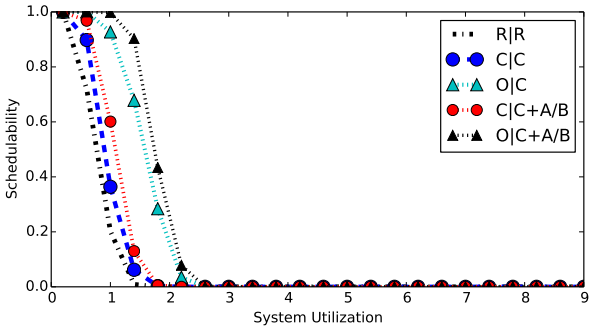
C-Light, Short, Light, Light, Large, High, Many



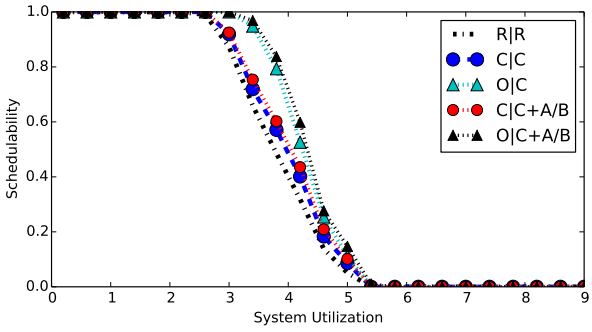
All-Mod., Long, Light, Light, Large, Low, Many



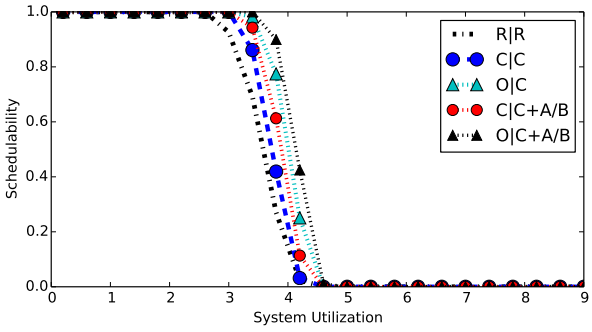
All-Mod., Long, Heavy, Heavy, Small, Low, Few



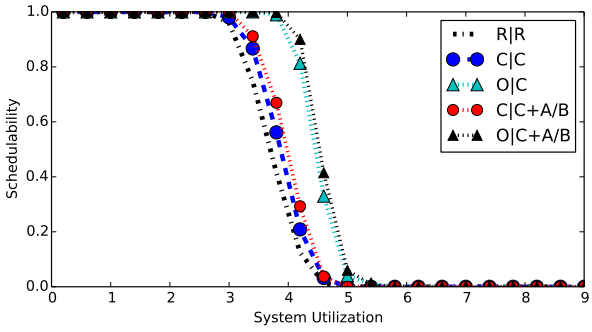
All-Mod., Mod., Light, Light, Large, High, Few



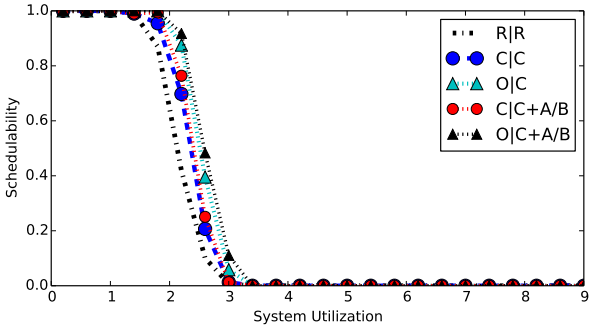
C-Heavy, Short, Heavy, Heavy, Small, High, Many



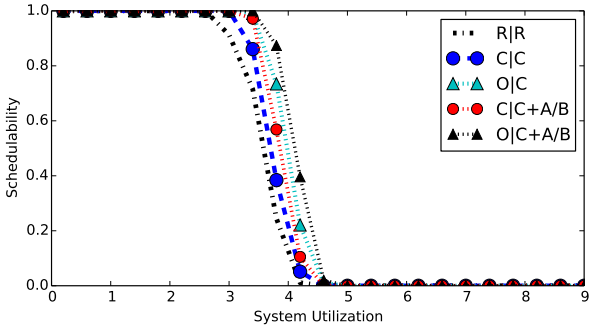
C-Light, Mod., Heavy, Light, Small, Low, Many



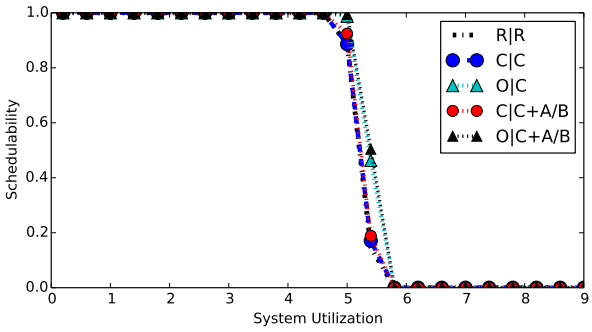
All-Mod., Short, Med., Heavy, Small, Low, Many



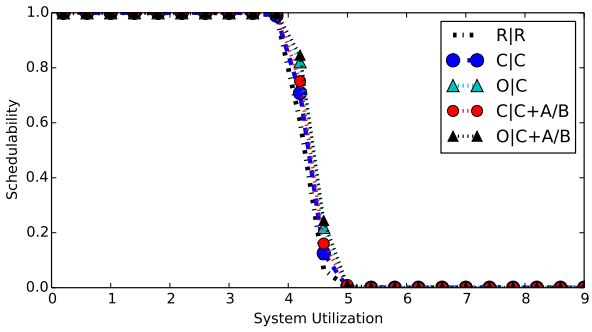
C-Heavy, Short, Light, Heavy, Small, High, Many



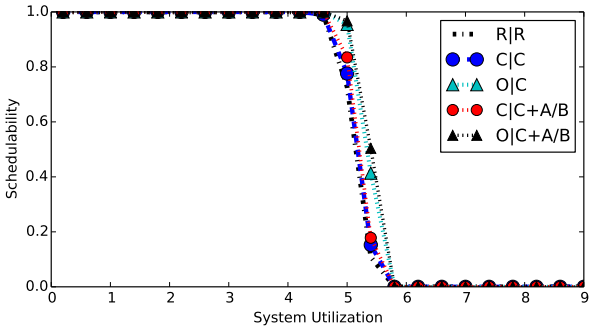
C-Light, Mod., Heavy, Light, Small, Med., Few



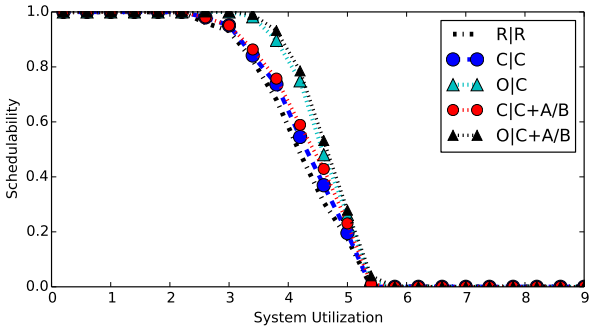
C-Heavy, Long, Med., Light, Large, Low, Many



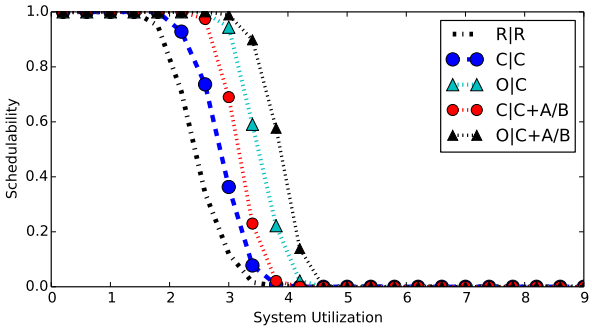
C-Light, Long, Heavy, Light, Small, High, Many



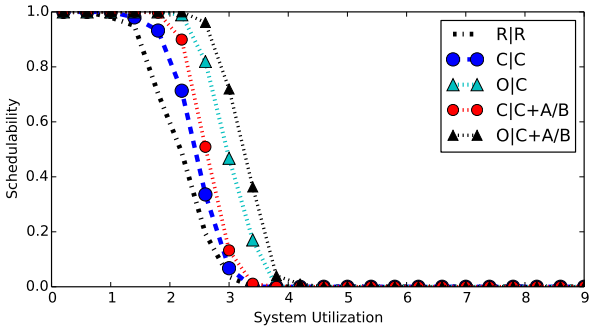
C-Heavy, Long, Med., Heavy, Large, Med., Few



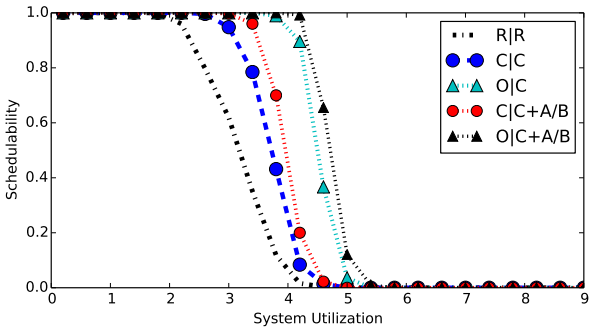
C-Heavy, Short, Heavy, Heavy, Small, Med., Few



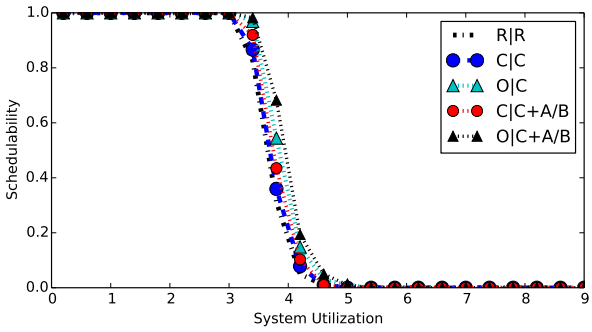
C-Light, Mod., Med., Heavy, Small, High, Many



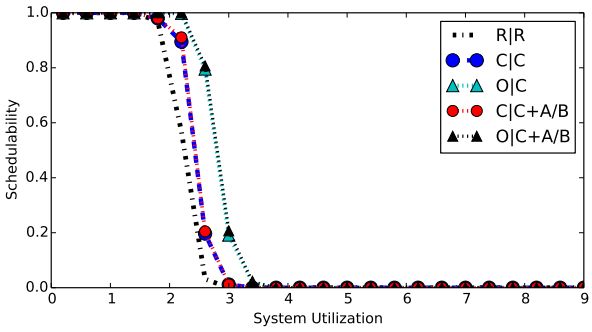
All-Mod., Mod., Med., Heavy, Large, High, Few



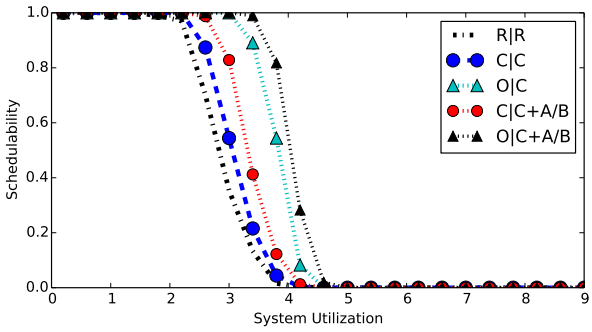
C-Light, Mod., Med., Light, Small, Low, Many



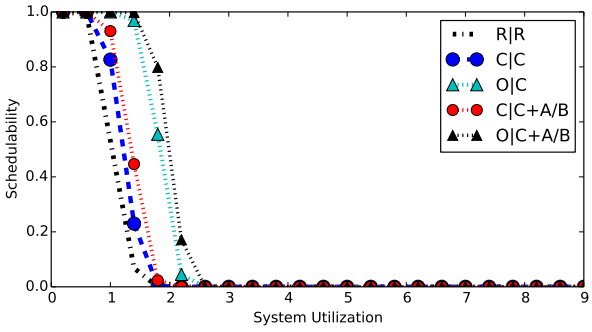
C-Heavy, Short, Med., Heavy, Large, Med., Many



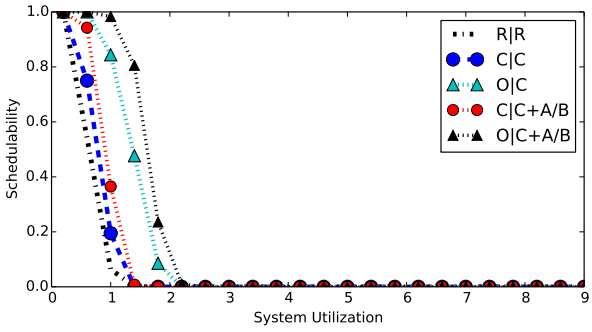
All-Mod., Long, Light, Heavy, Large, Low, Few



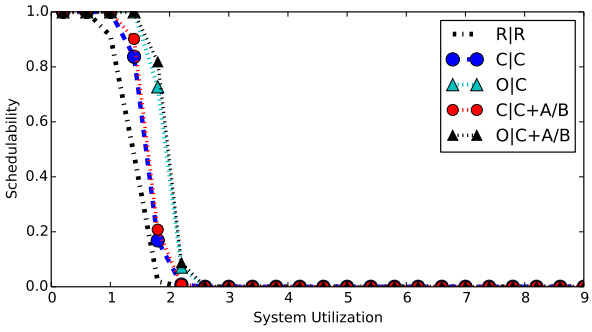
All-Mod., Mod., Med., Heavy, Small, High, Few



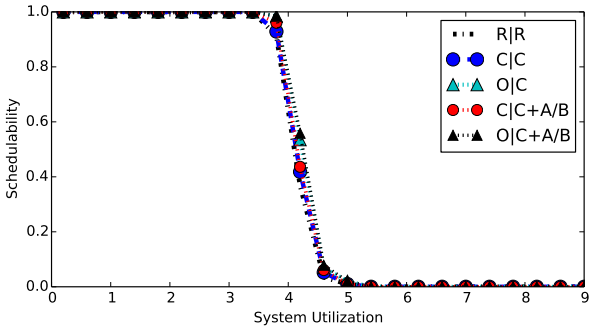
C-Light, Short, Light, Heavy, Large, High, Few



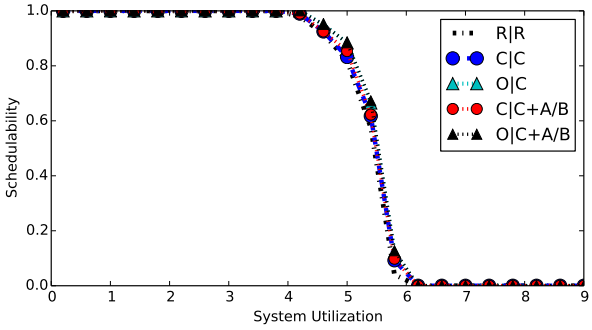
C-Light, Mod., Light, Heavy, Large, High, Few



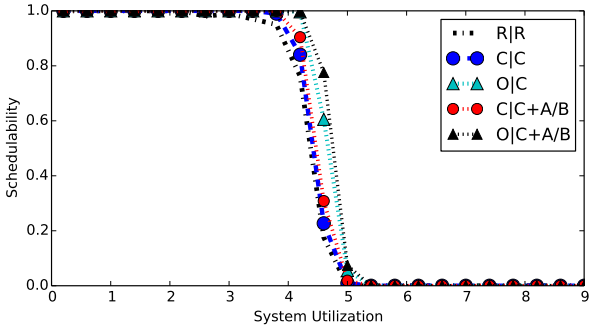
C-Light, Mod., Light, Light, Small, Low, Few



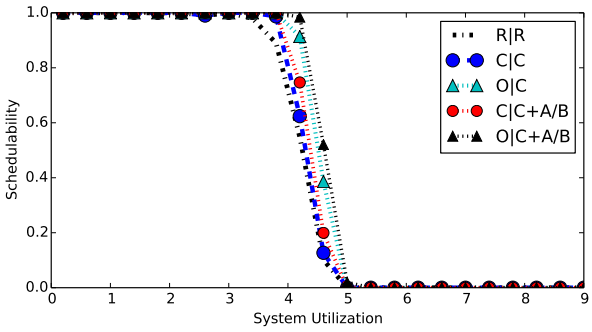
All-Mod., Long, Heavy, Light, Small, High, Few



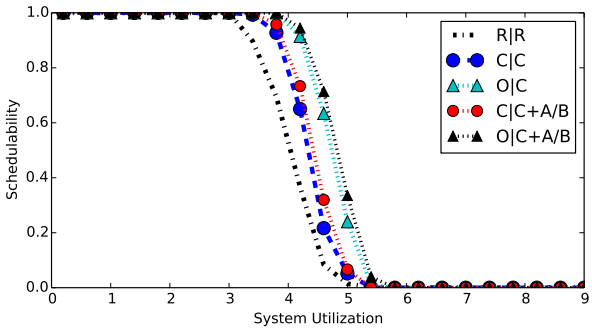
C-Heavy, Long, Heavy, Light, Small, Med., Few



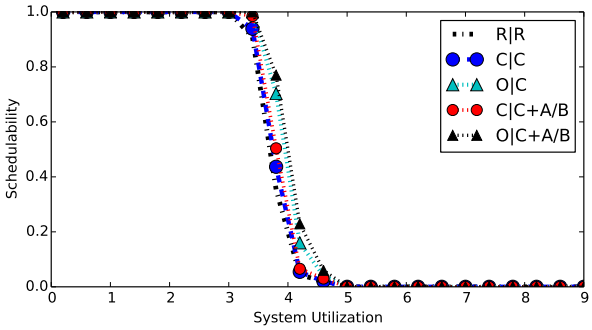
C-Heavy, Short, Med., Light, Large, Med., Few



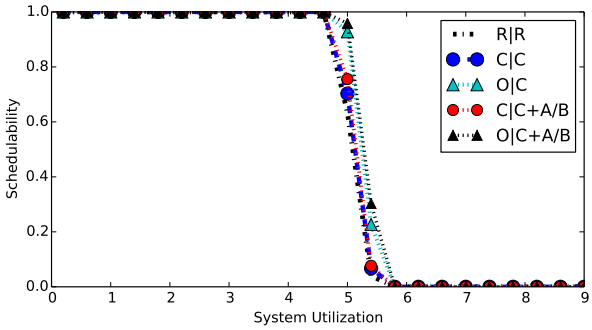
C-Heavy, Short, Med., Light, Large, High, Few



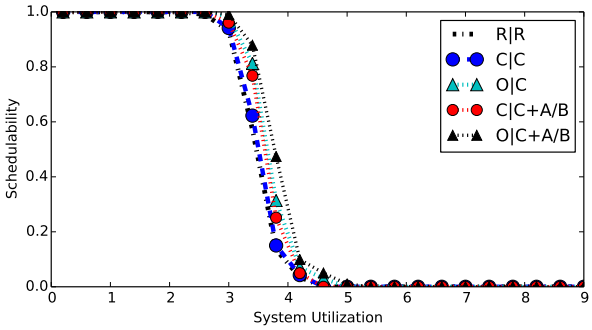
All-Mod., Long, Med., Heavy, Large, Med., Few



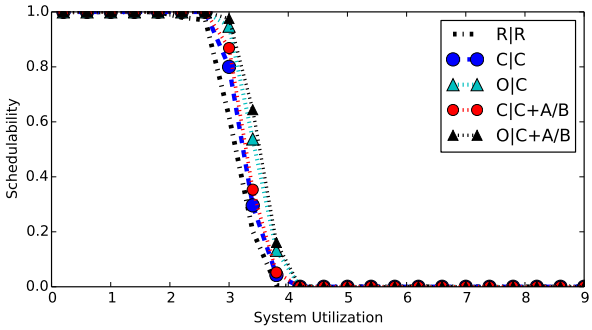
All-Mod., Short, Heavy, Light, Small, Med., Few



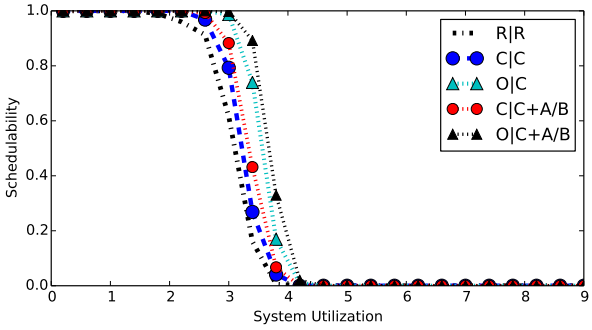
C-Heavy, Long, Med., Heavy, Large, High, Few



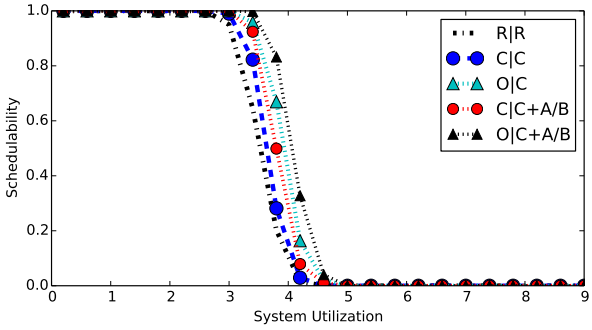
C-Heavy, Mod., Med., Heavy, Small, High, Few



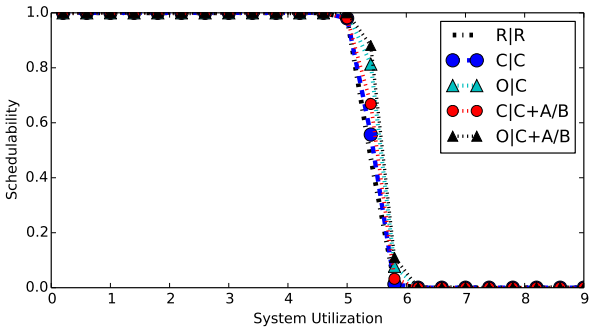
C-Light, Short, Heavy, Heavy, Large, Med., Few



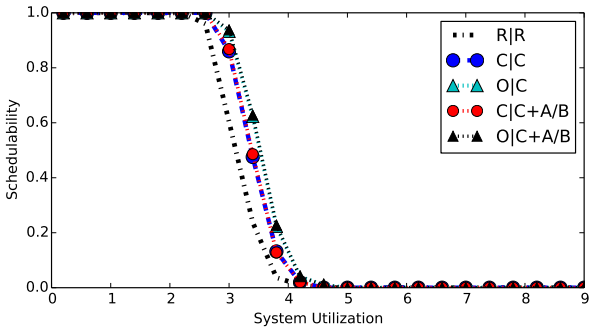
C-Light, Mod., Heavy, Heavy, Small, High, Few



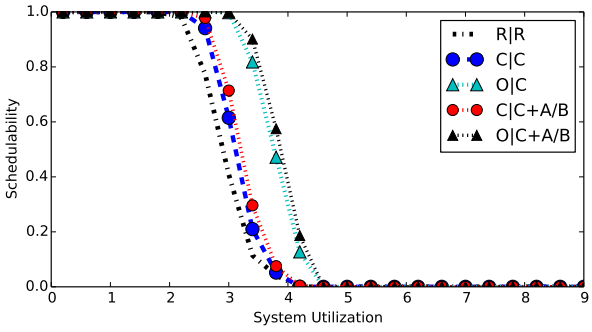
C-Light, Mod., Heavy, Light, Small, High, Many



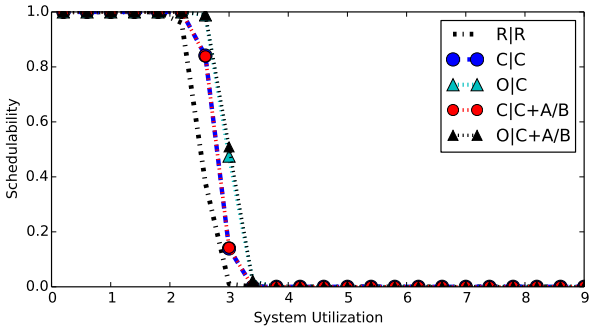
C-Heavy, Long, Med., Heavy, Small, Med., Few



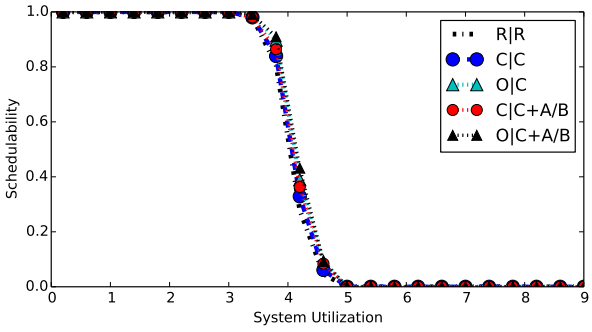
C-Heavy, Long, Light, Heavy, Small, Low, Few



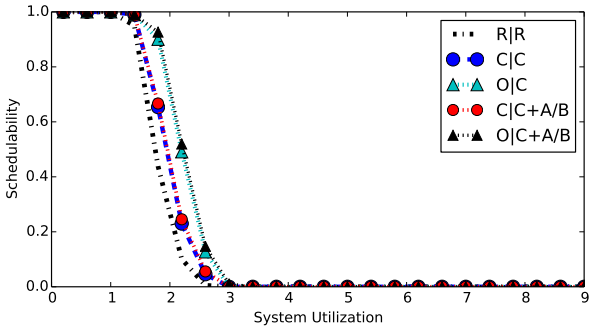
All-Mod., Short, Med., Heavy, Large, Med., Many



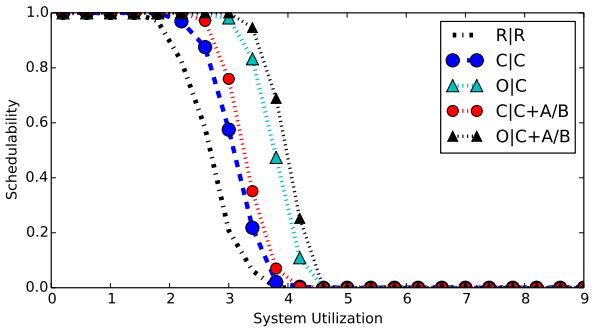
All-Mod., Long, Light, Heavy, Small, Low, Many



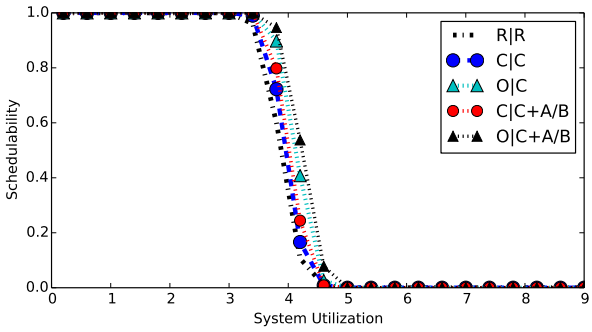
All-Mod., Long, Heavy, Heavy, Large, High, Few



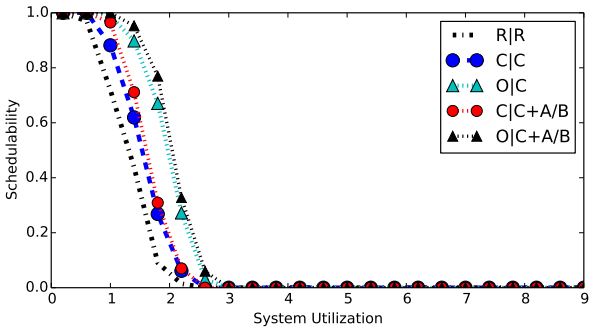
C-Heavy, Mod., Light, Heavy, Small, Med., Many



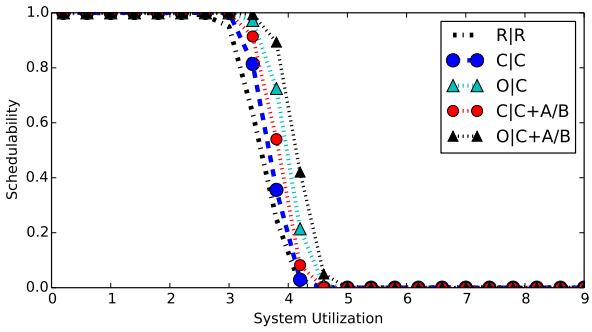
C-Light, Mod., Med., Heavy, Small, Med., Few



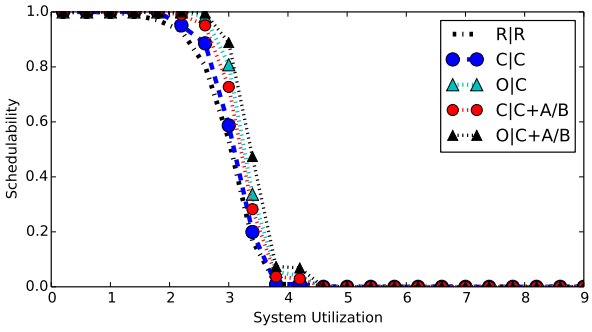
C-Light, Short, Heavy, Light, Small, Med., Many



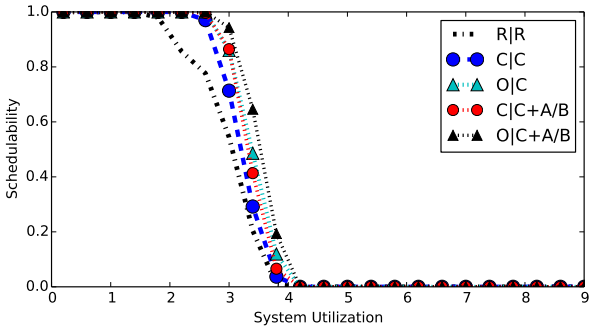
C-Heavy, Mod., Light, Light, Large, Med., Few



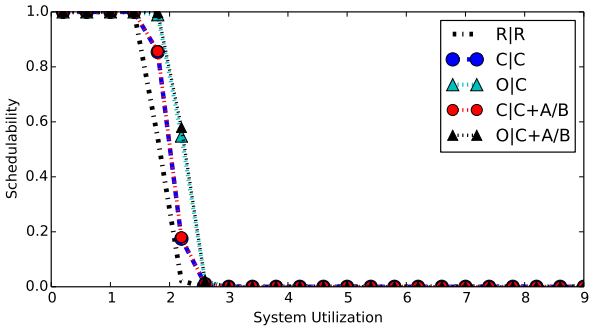
C-Light, Mod., Heavy, Light, Small, Med., Many



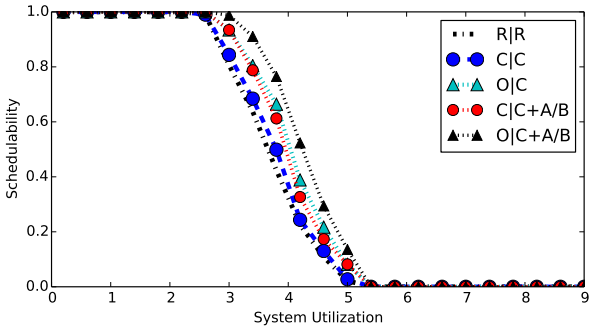
C-Heavy, Mod., Med., Heavy, Large, High, Many



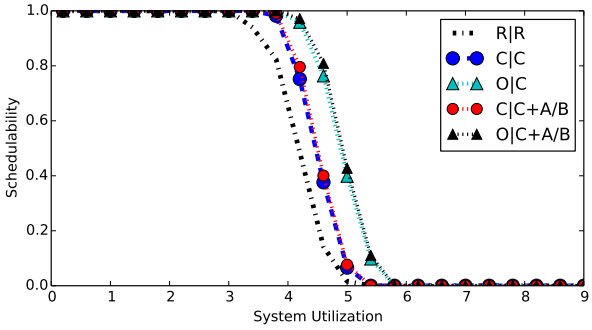
C-Light, Mod., Heavy, Light, Large, Med., Few



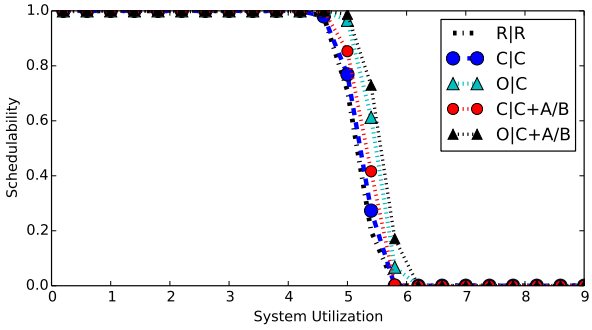
All-Mod., Short, Light, Heavy, Small, Low, Few



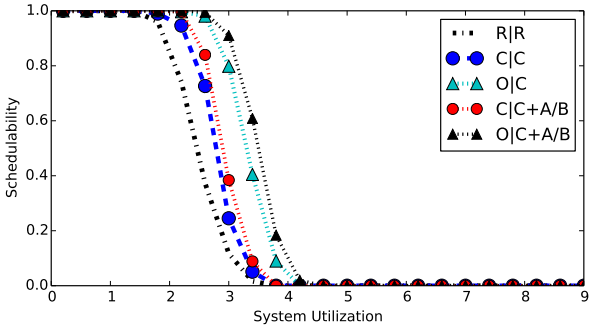
C-Heavy, Mod., Heavy, Light, Large, Low, Many



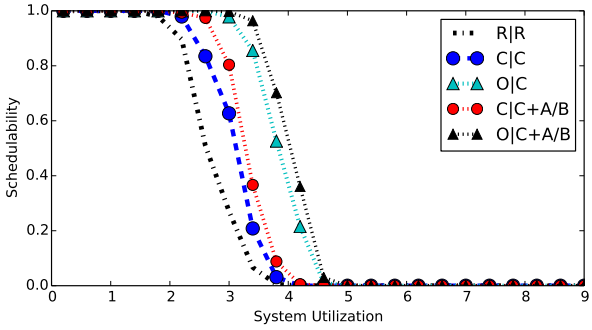
All-Mod., Long, Med., Heavy, Large, Low, Many



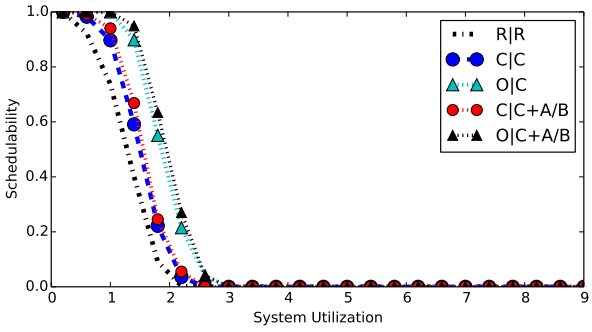
All-Mod., Long, Med., Light, Small, Med., Few



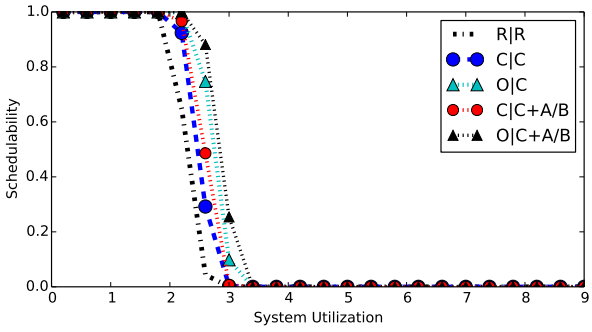
C-Light, Short, Med., Heavy, Large, High, Many



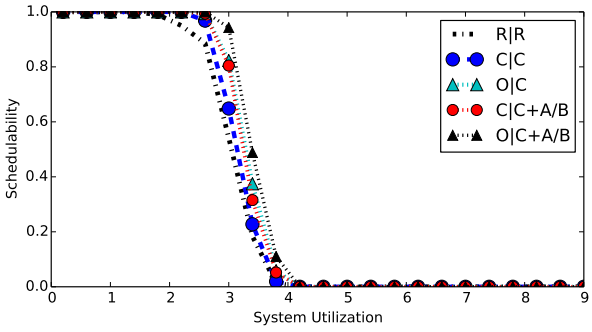
C-Light, Mod., Med., Heavy, Small, Low, Many



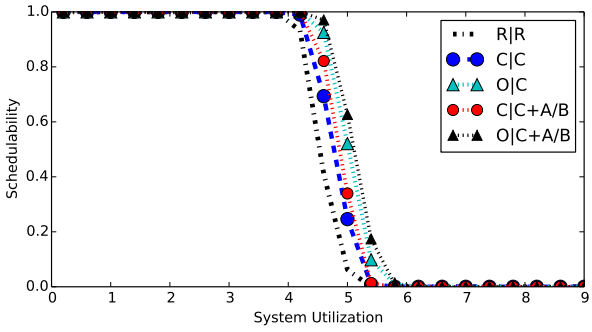
C-Heavy, Mod., Light, Heavy, Large, Low, Few



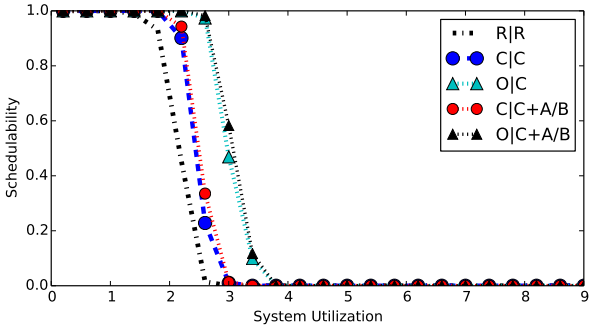
C-Light, Long, Light, Heavy, Small, High, Many



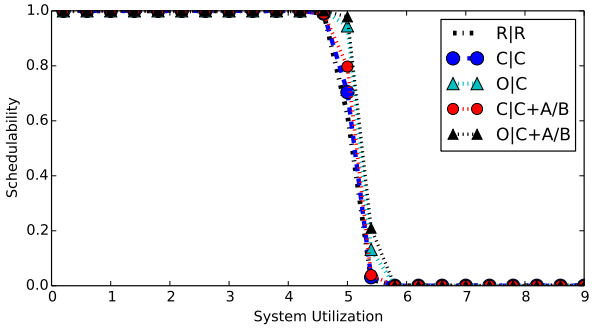
All-Mod., Mod., Heavy, Light, Large, Low, Many



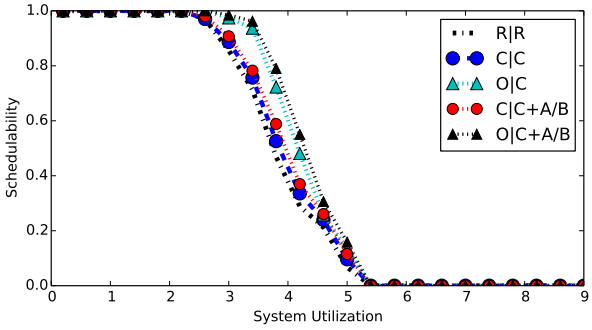
C-Light, Long, Med., Light, Small, Med., Many



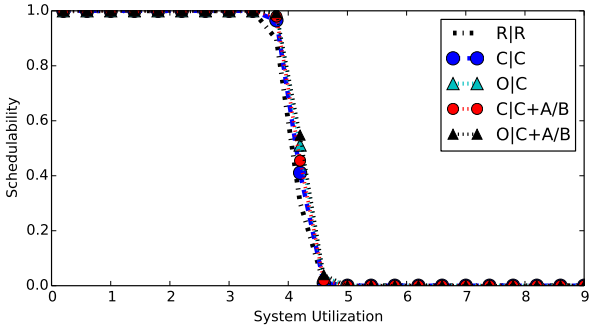
C-Light, Long, Light, Light, Large, Med., Few



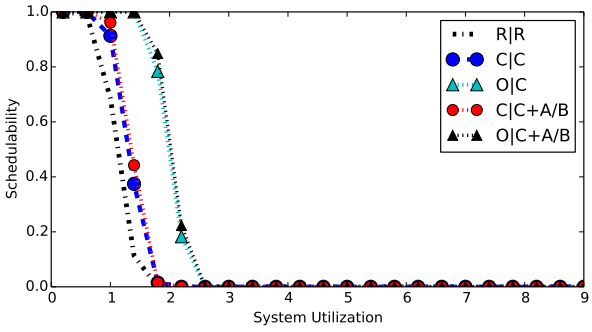
C-Heavy, Long, Med., Light, Large, Med., Few



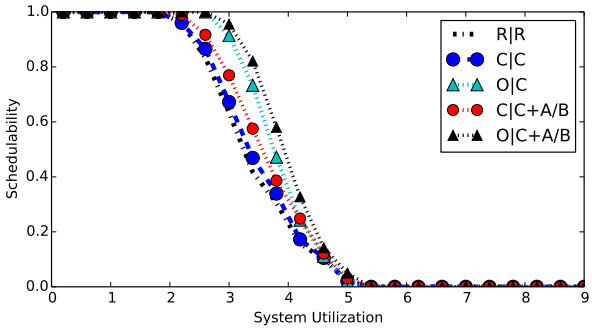
C-Heavy, Short, Heavy, Heavy, Large, Med., Many



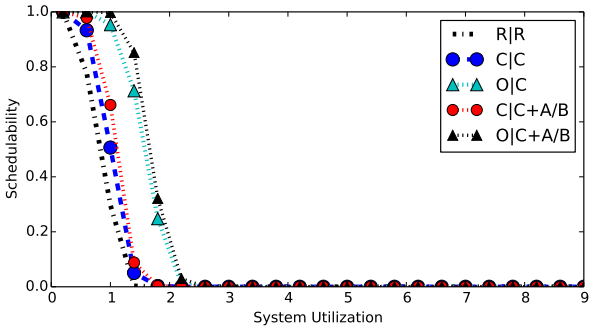
C-Light, Long, Heavy, Light, Large, Med., Many



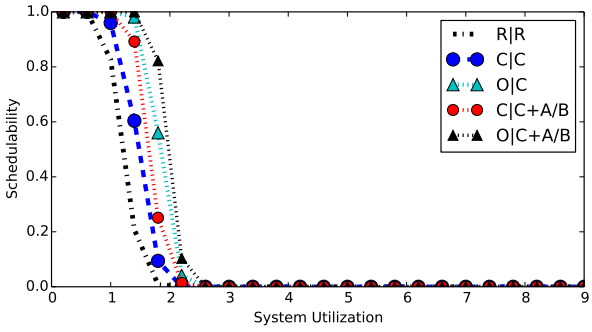
C-Light, Short, Light, Heavy, Large, Low, Many



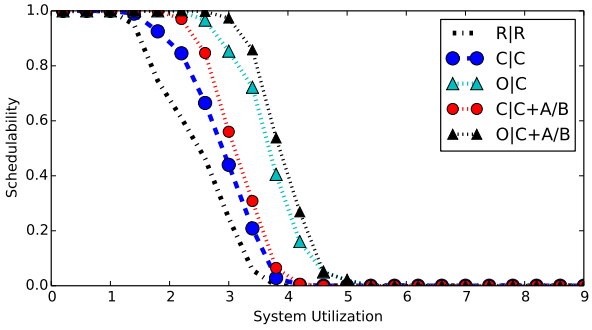
C-Heavy, Mod., Heavy, Heavy, Large, Low, Few



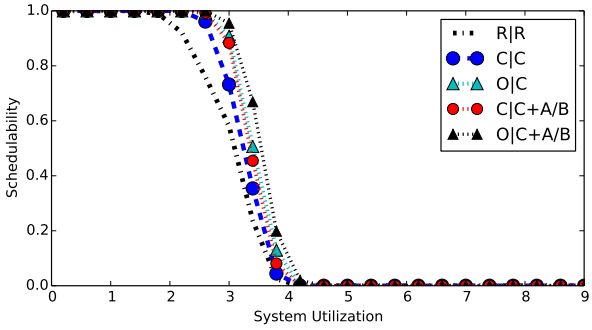
All-Mod., Mod., Light, Heavy, Large, Med., Few



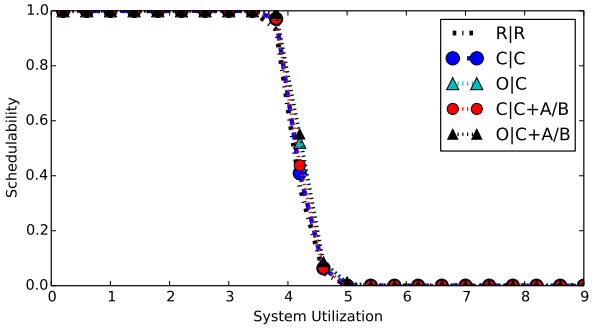
C-Light, Mod., Light, Light, Small, High, Few



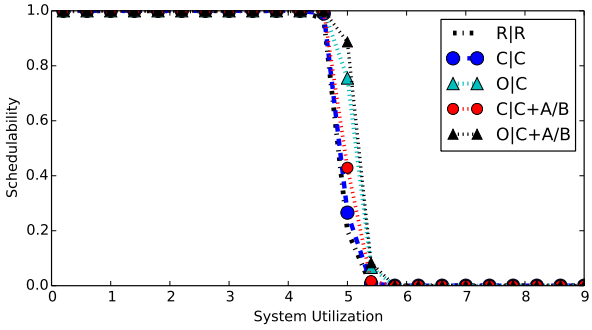
All-Mod., Mod., Med., Light, Large, Low, Few



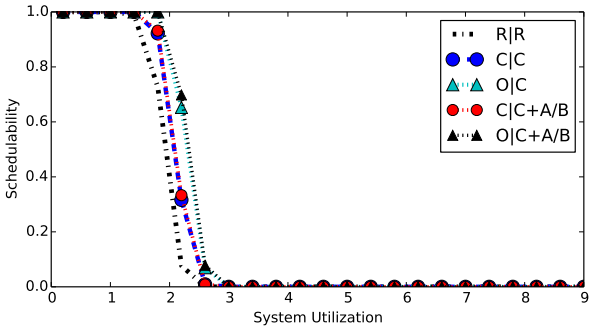
C-Light, Mod., Heavy, Light, Large, High, Many



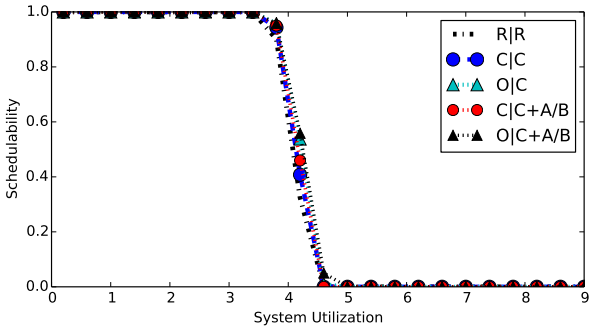
All-Mod., Long, Heavy, Light, Small, Med., Many



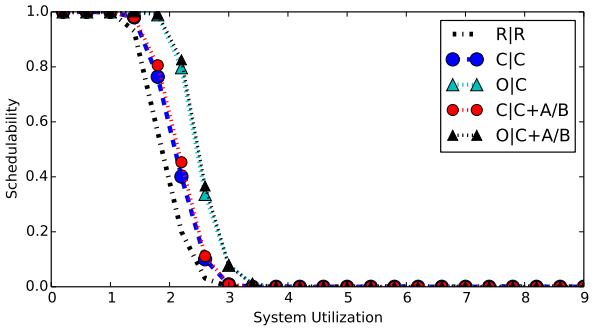
C-Heavy, Short, Med., Light, Small, Med., Many



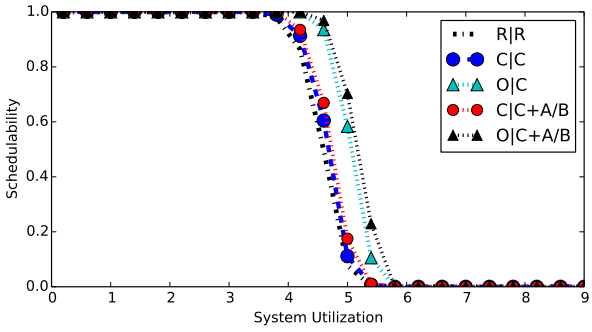
All-Mod., Short, Light, Light, Small, Low, Few



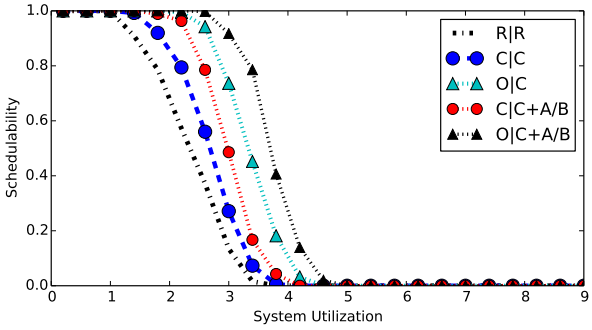
C-Light, Long, Heavy, Light, Large, Med., Few



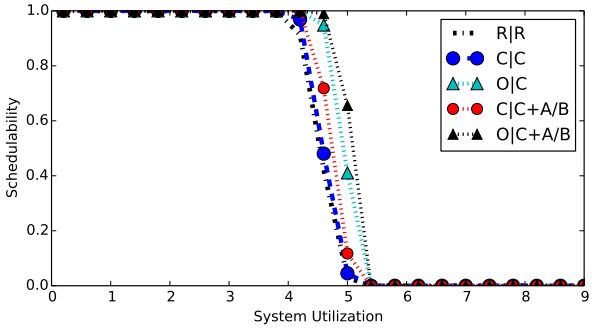
C-Heavy, Short, Light, Light, Large, Med., Few



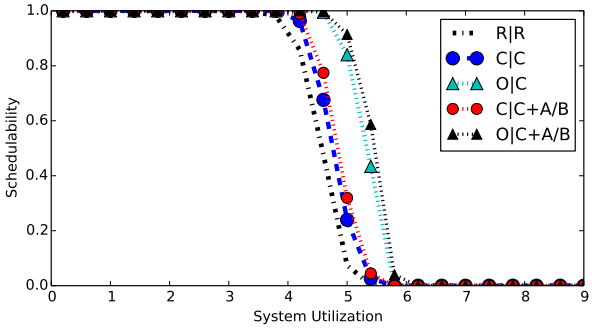
All-Mod., Long, Med., Light, Large, High, Many



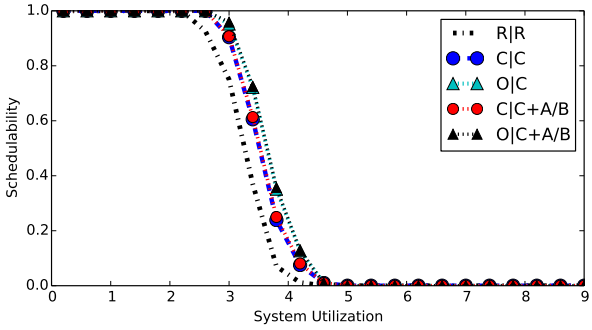
All-Mod., Mod., Med., Light, Large, High, Few



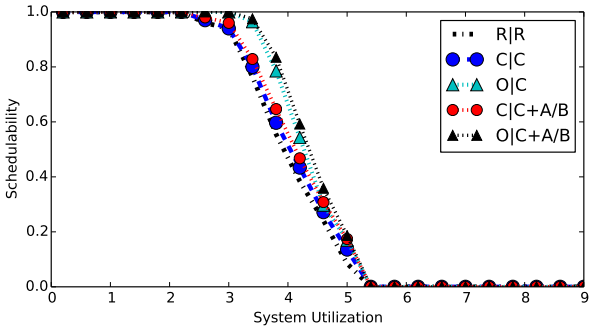
C-Heavy, Mod., Med., Light, Small, Low, Few



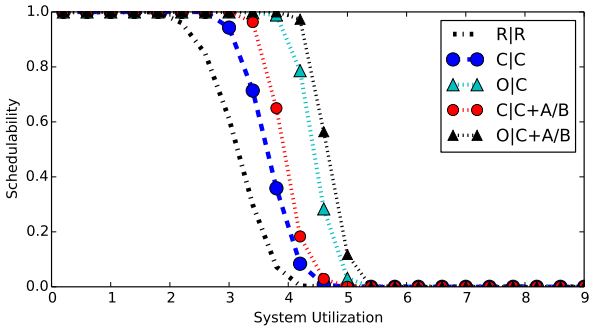
All-Mod., Short, Med., Light, Small, Low, Few



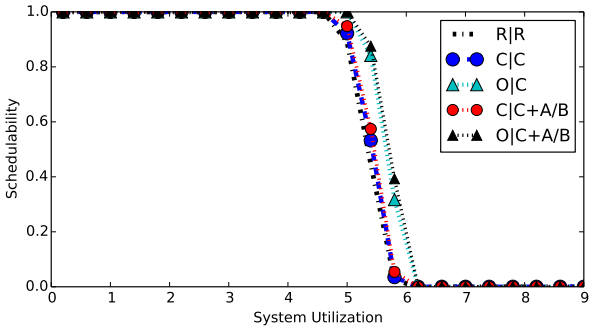
C-Heavy, Long, Light, Light, Small, Low, Many



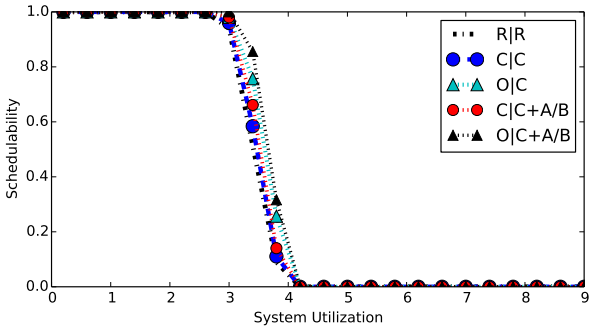
C-Heavy, Short, Heavy, Heavy, Large, Low, Many



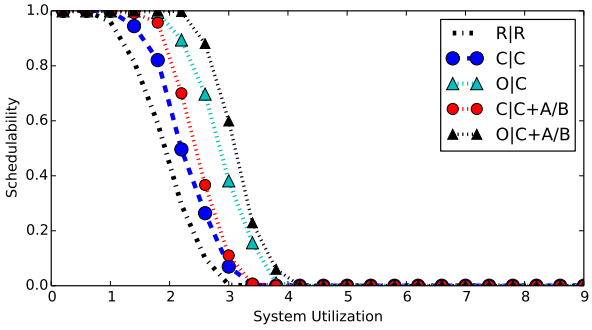
C-Light, Mod., Med., Light, Small, Med., Few



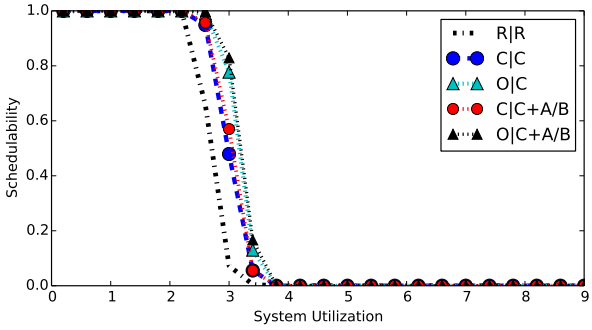
All-Mod., Long, Med., Light, Small, Low, Many



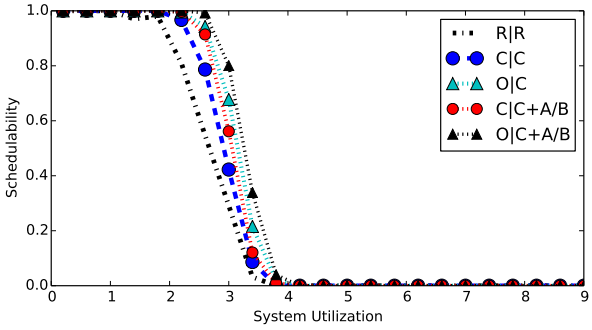
All-Mod., Short, Heavy, Light, Large, Med., Many



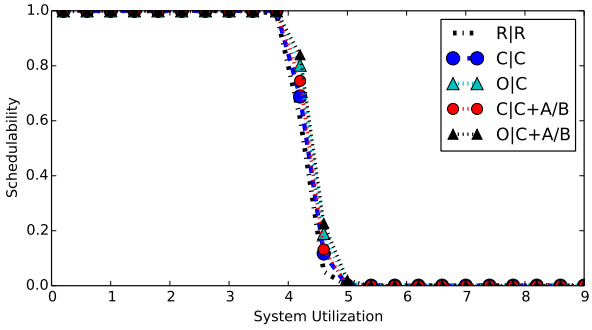
C-Light, Mod., Med., Heavy, Large, Low, Many



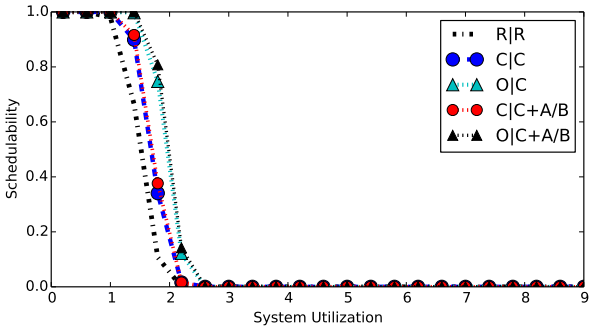
All-Mod., Long, Light, Light, Small, Med., Many



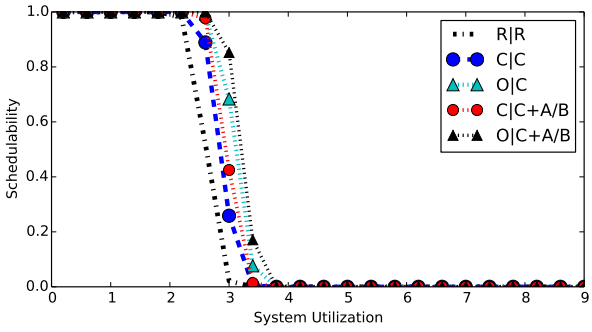
C-Light, Mod., Heavy, Heavy, Large, Low, Few



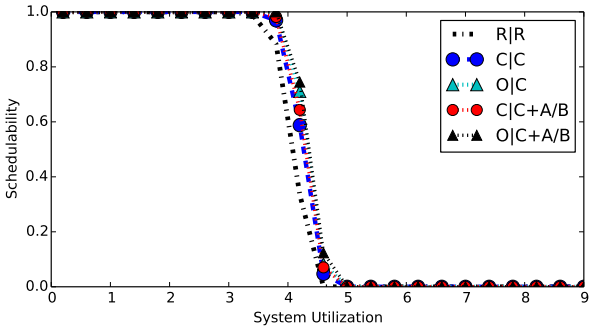
C-Light, Long, Heavy, Light, Small, High, Few



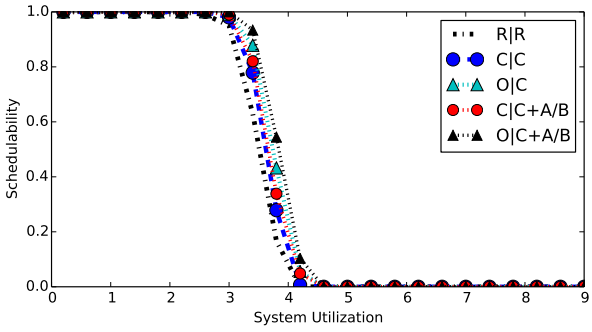
All-Mod., Mod., Light, Light, Small, Low, Many



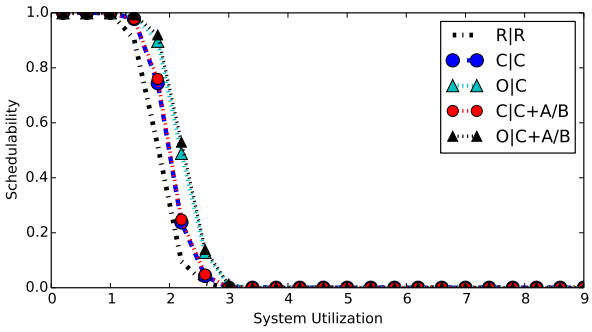
C-Light, Long, Light, Light, Small, High, Many



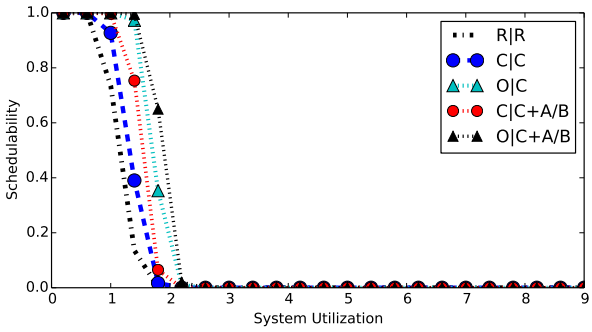
C-Light, Long, Heavy, Heavy, Small, Med., Many



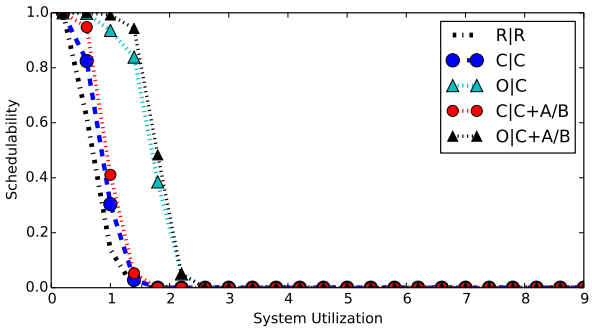
C-Light, Short, Heavy, Light, Large, Med., Few



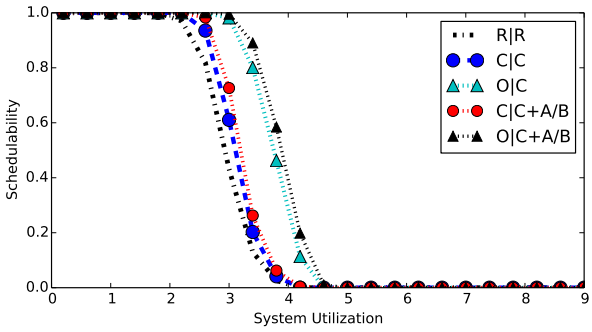
C-Heavy, Mod., Light, Heavy, Small, Low, Many



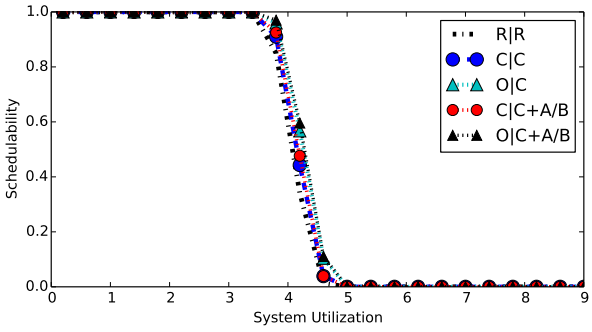
C-Light, Mod., Light, Heavy, Small, High, Few



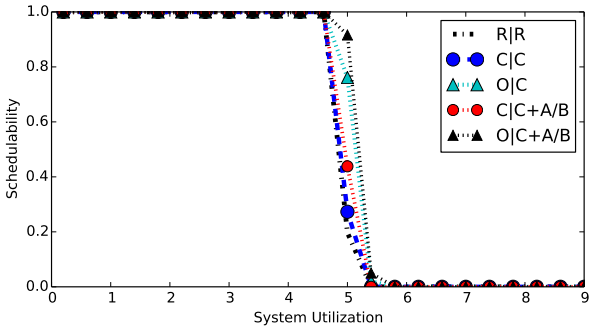
C-Light, Mod., Light, Light, Large, Med., Many



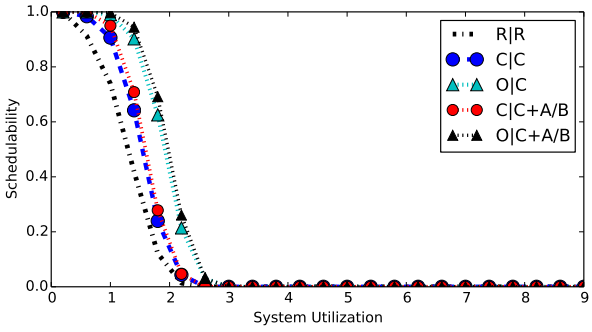
All-Mod., Short, Med., Heavy, Large, Med., Few



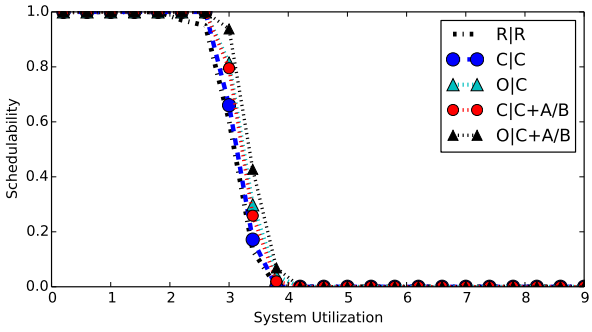
C-Light, Long, Heavy, Light, Large, Low, Few



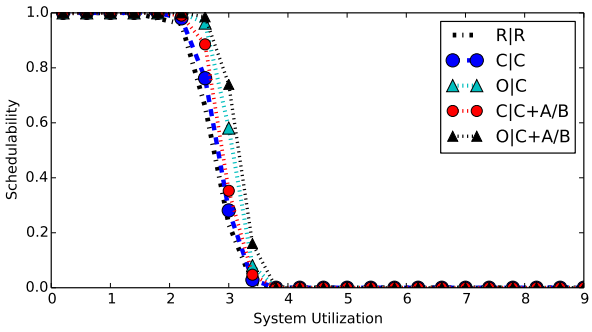
C-Heavy, Short, Med., Light, Small, Med., Few



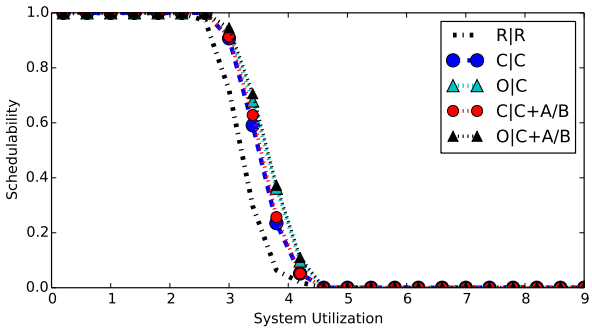
C-Heavy, Mod., Light, Heavy, Large, Low, Many



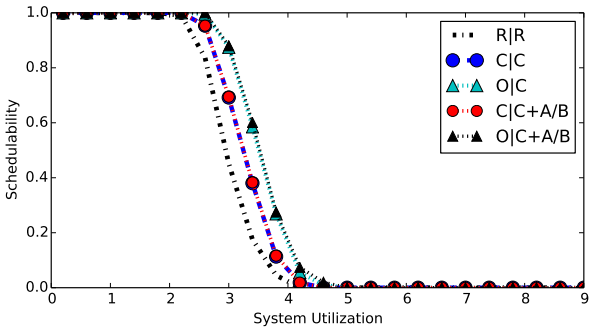
All-Mod., Mod., Heavy, Light, Large, Med., Few



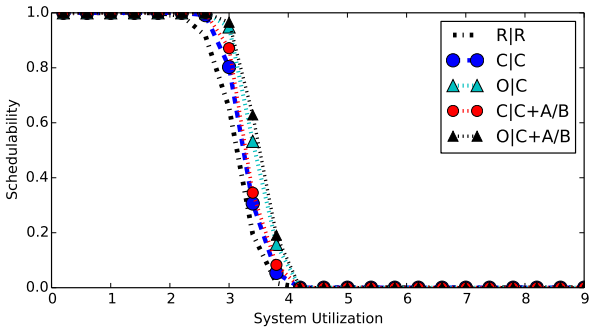
All-Mod., Mod., Heavy, Heavy, Large, High, Many



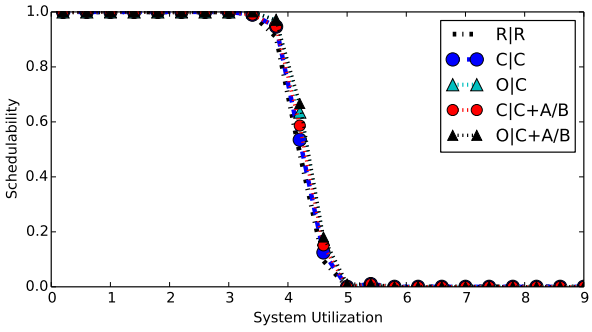
C-Heavy, Long, Light, Light, Small, Med., Few



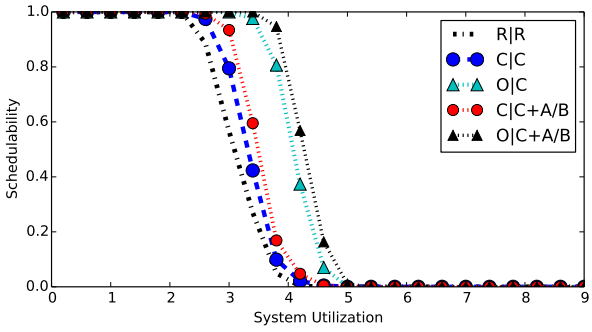
C-Heavy, Long, Light, Light, Large, Low, Many



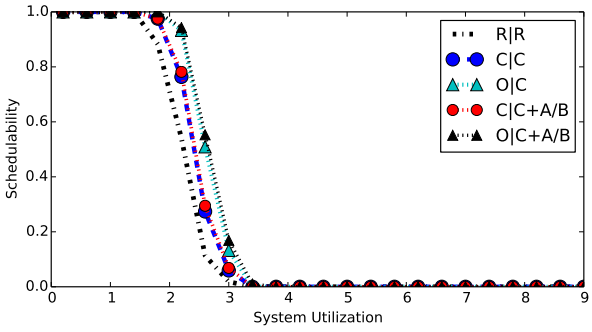
C-Light, Short, Heavy, Heavy, Large, Low, Many



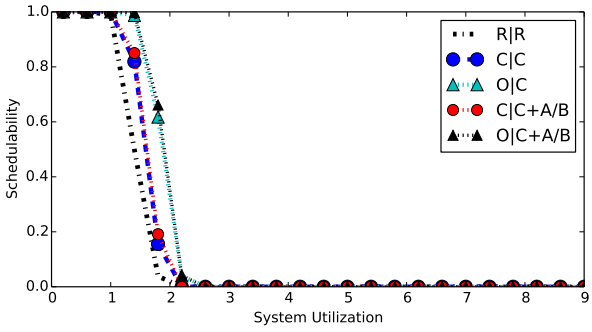
All-Mod., Long, Heavy, Heavy, Small, Med., Few



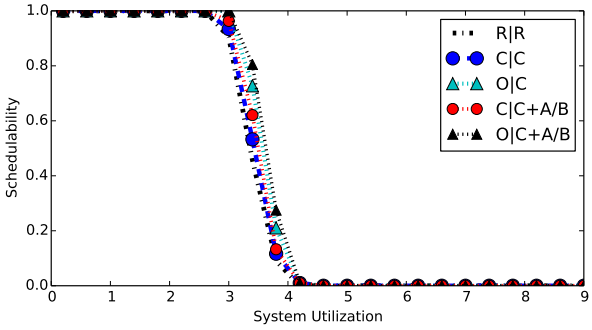
All-Mod., Mod., Med., Heavy, Small, Med., Many



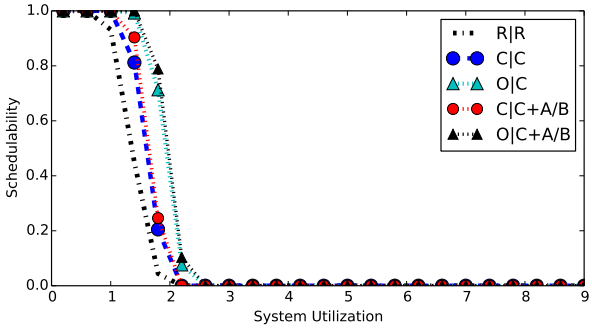
C-Heavy, Short, Light, Heavy, Small, Med., Few



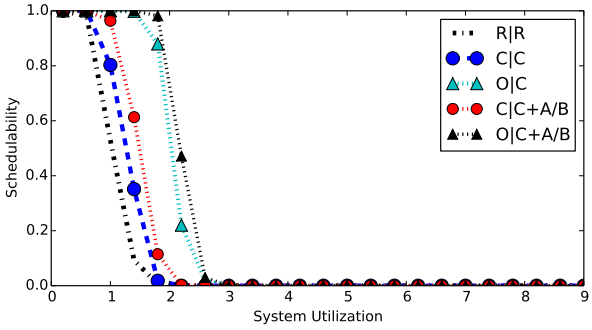
All-Mod., Mod., Light, Heavy, Small, Low, Few



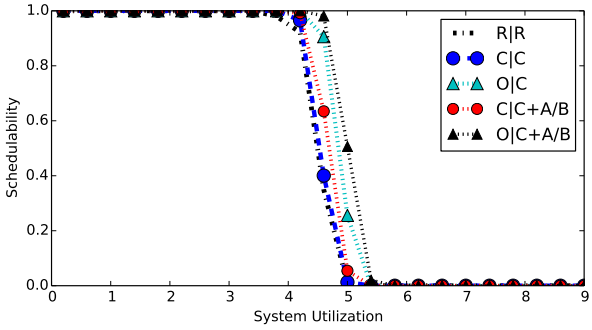
All-Mod., Short, Heavy, Light, Large, High, Few



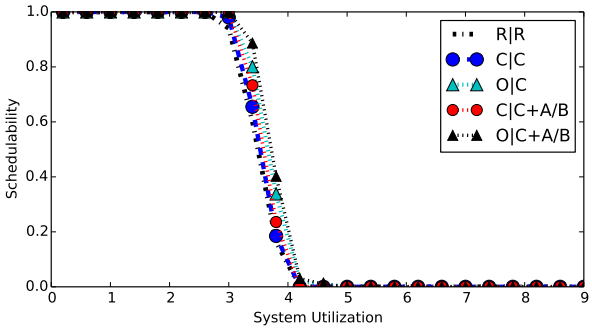
C-Light, Mod., Light, Light, Small, Med., Many



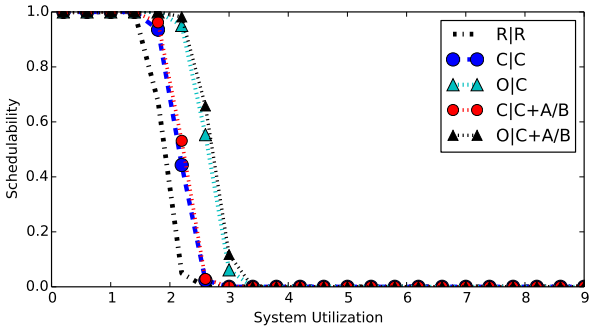
C-Light, Short, Light, Light, Large, High, Few



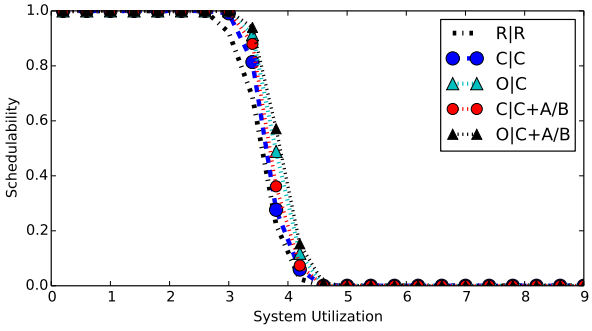
C-Heavy, Mod., Med., Light, Small, Med., Few



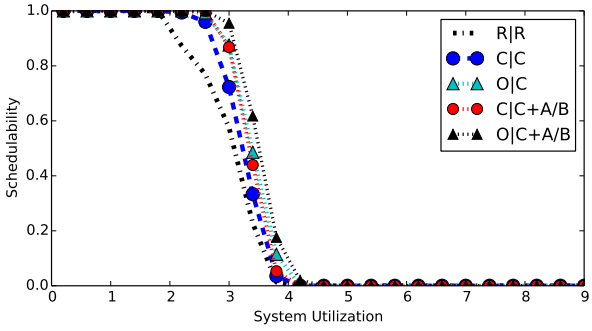
All-Mod., Short, Heavy, Light, Large, Low, Many



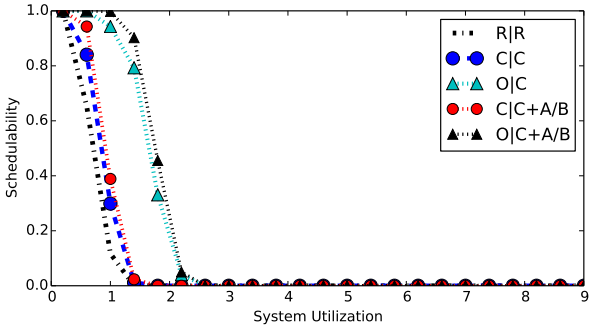
C-Light, Long, Light, Heavy, Large, Med., Few



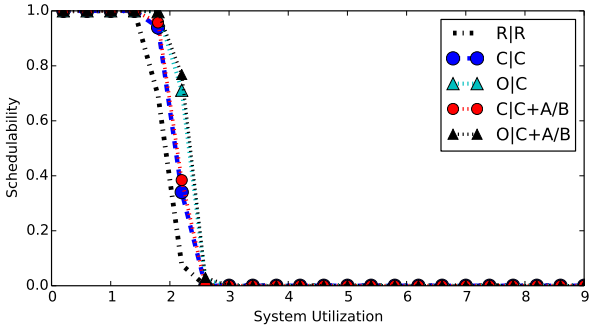
C-Light, Short, Heavy, Light, Large, Low, Many



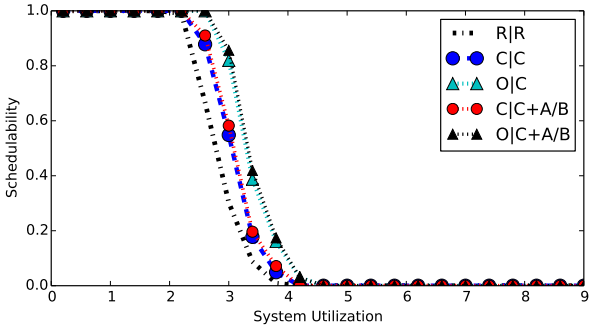
C-Light, Mod., Heavy, Light, Large, Low, Many



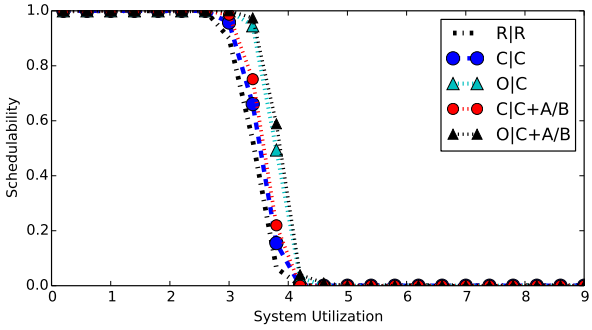
C-Light, Mod., Light, Light, Large, Med., Few



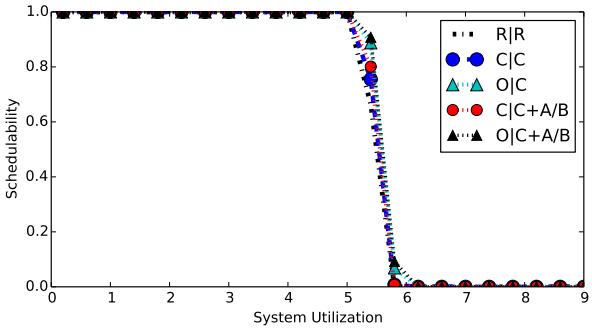
All-Mod., Short, Light, Light, Small, Med., Many



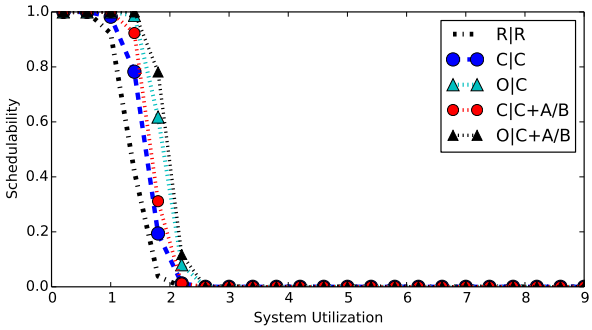
C-Heavy, Long, Light, Heavy, Large, Med., Many



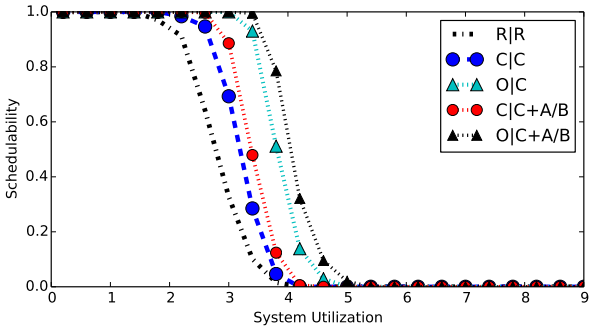
C-Light, Short, Heavy, Heavy, Small, Low, Few



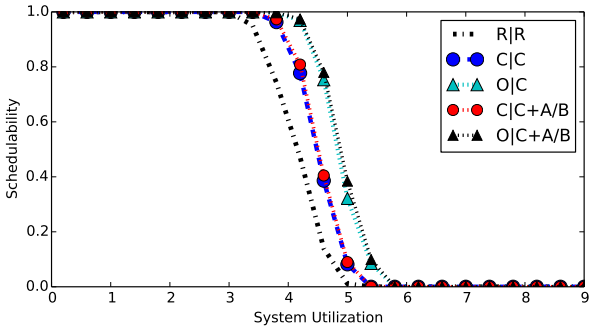
C-Heavy, Long, Med., Light, Small, Low, Few



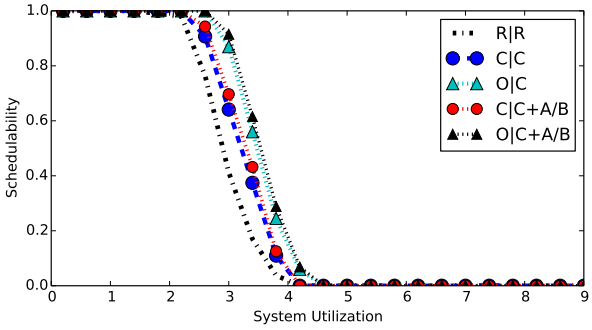
All-Mod., Mod., Light, Light, Small, High, Many



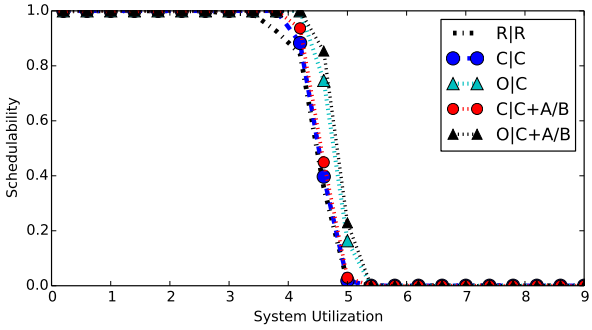
C-Light, Short, Med., Light, Large, High, Many



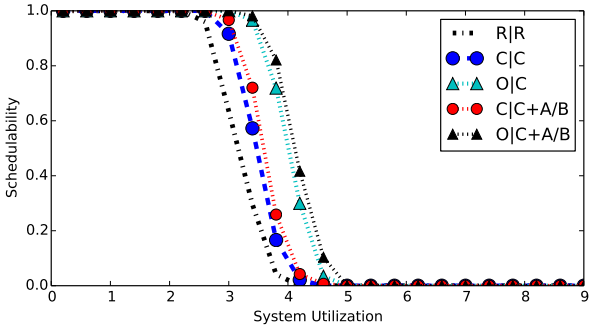
All-Mod., Long, Med., Heavy, Large, Low, Few



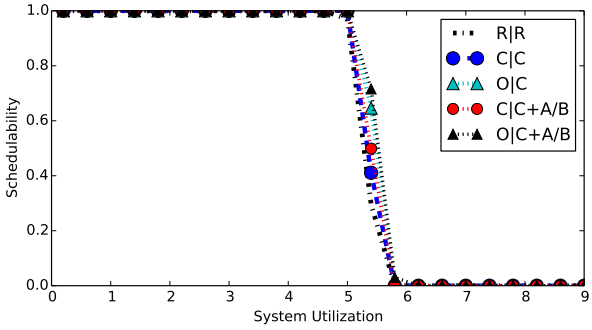
C-Heavy, Long, Light, Light, Large, High, Few



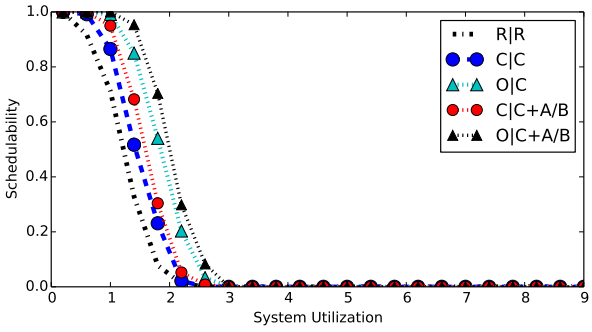
C-Heavy, Short, Med., Light, Large, Low, Few



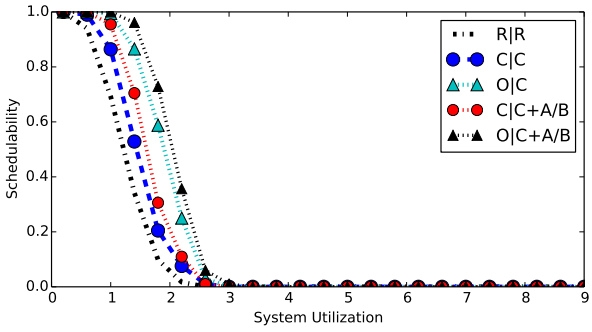
C-Light, Short, Med., Heavy, Small, Med., Few



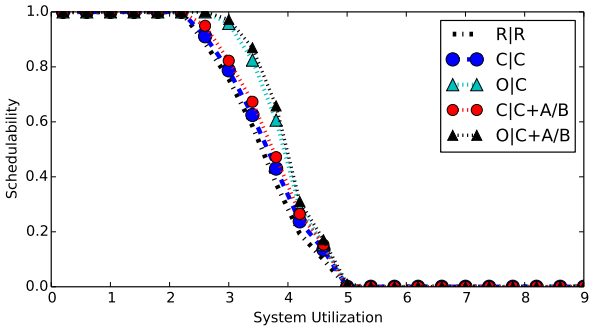
C-Heavy, Long, Med., Light, Small, Med., Many



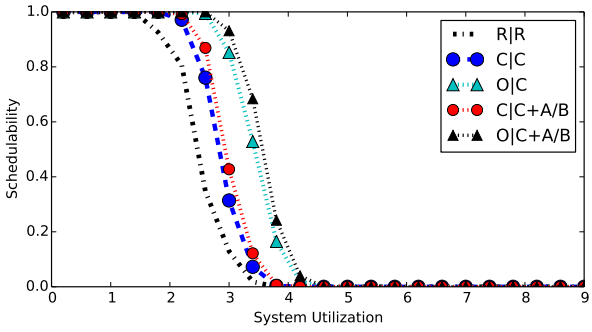
C-Heavy, Mod., Light, Light, Large, High, Few



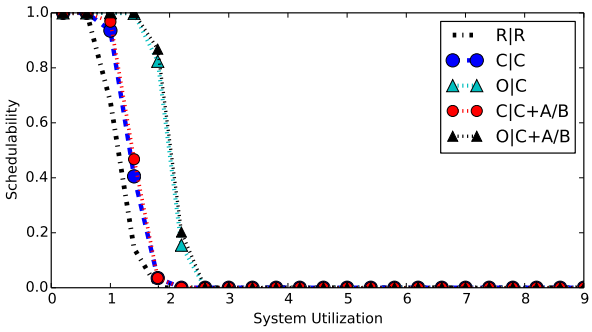
C-Heavy, Mod., Light, Light, Large, High, Many



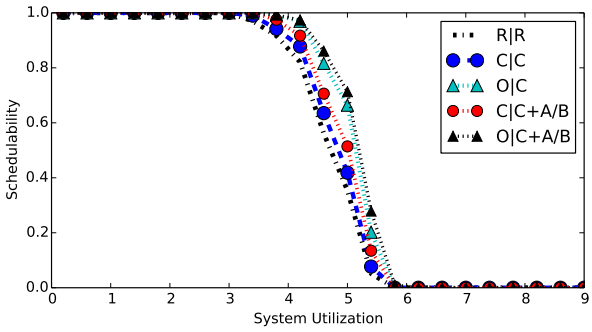
C-Heavy, Short, Heavy, Heavy, Large, High, Few



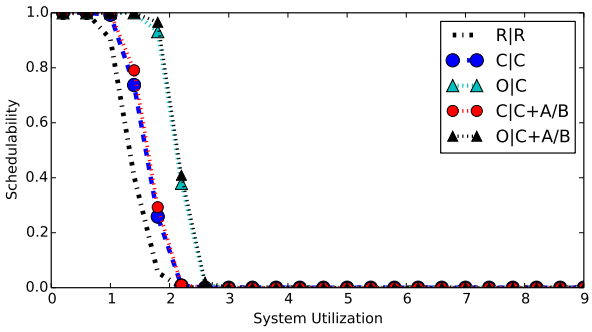
C-Light, Short, Med., Heavy, Large, Med., Many



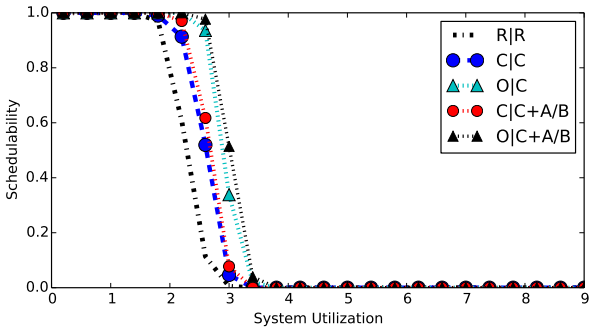
C-Light, Short, Light, Heavy, Large, Low, Few



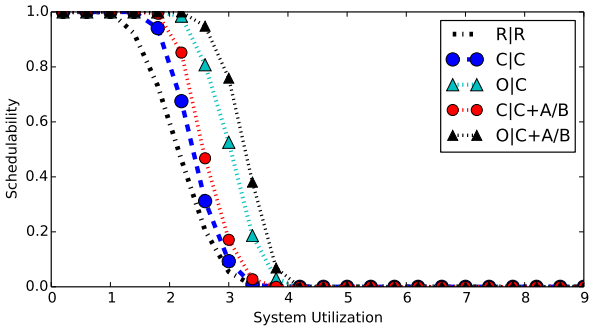
C-Heavy, Short, Heavy, Light, Small, Low, Few



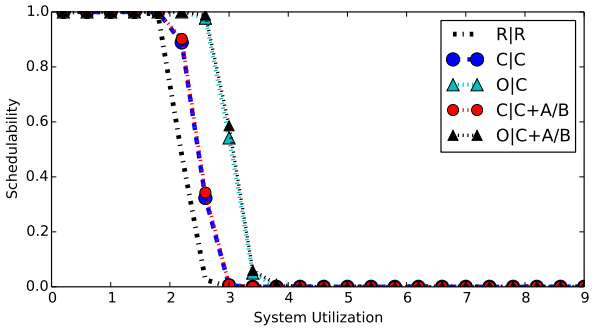
All-Mod., Short, Light, Light, Large, Low, Few



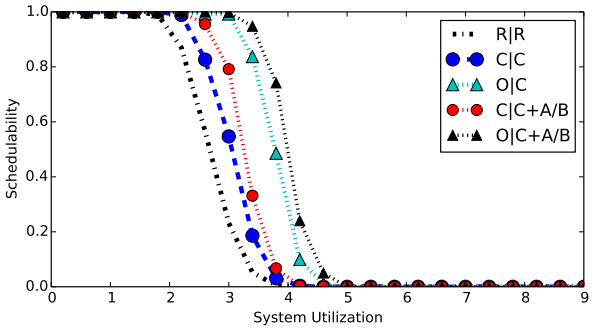
All-Mod., Long, Light, Light, Large, High, Many



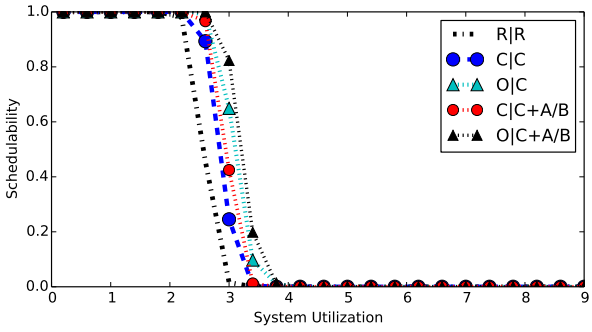
All-Mod., Mod., Med., Heavy, Large, High, Many



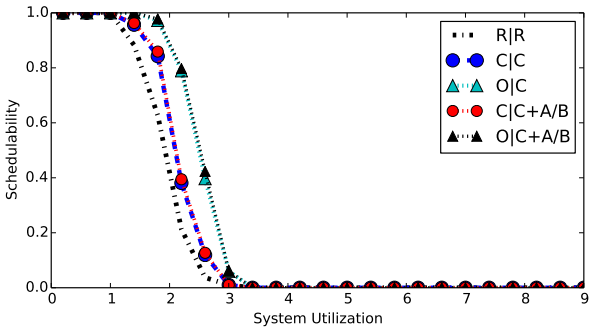
C-Light, Long, Light, Light, Large, Low, Few



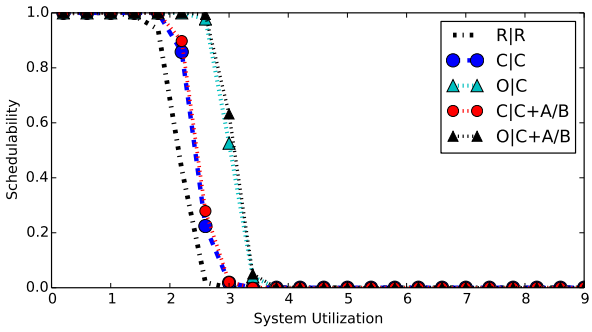
C-Light, Mod., Med., Heavy, Small, Med., Many



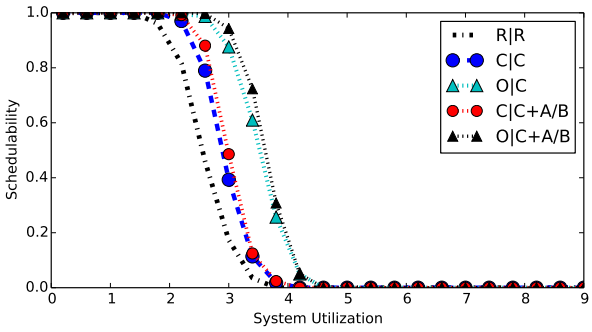
C-Light, Long, Light, Light, Small, High, Few



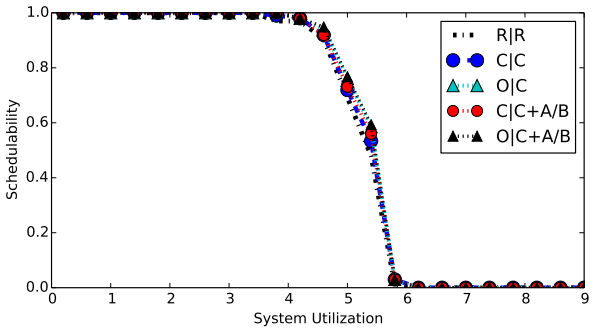
C-Heavy, Short, Light, Light, Large, Low, Many



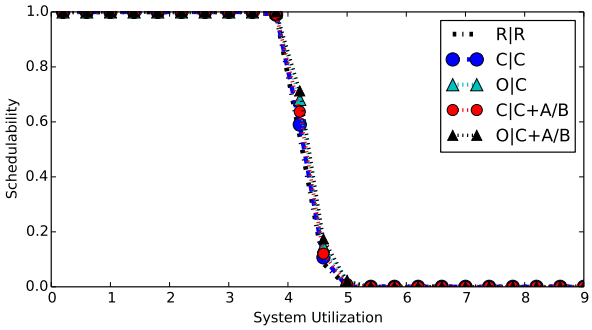
C-Light, Long, Light, Light, Large, Med., Many



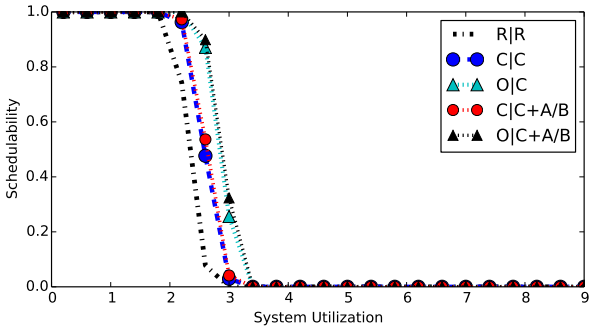
C-Light, Short, Med., Heavy, Large, Low, Many



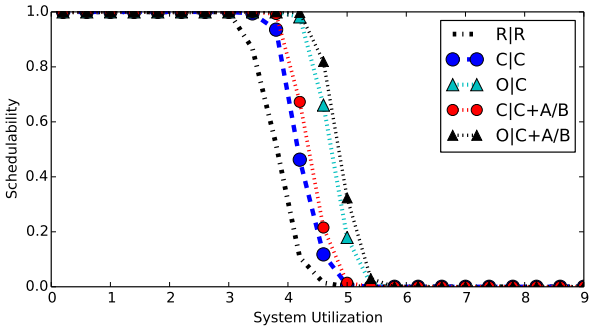
C-Heavy, Long, Heavy, Light, Large, Med., Few



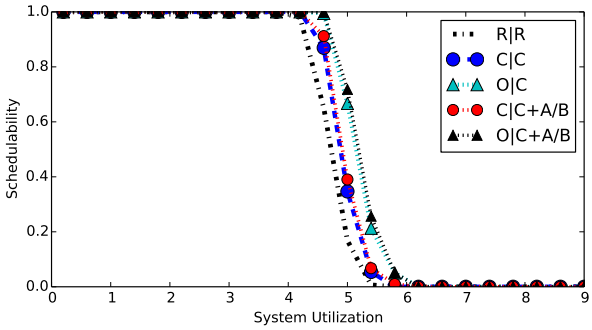
All-Mod., Long, Heavy, Heavy, Small, Low, Many



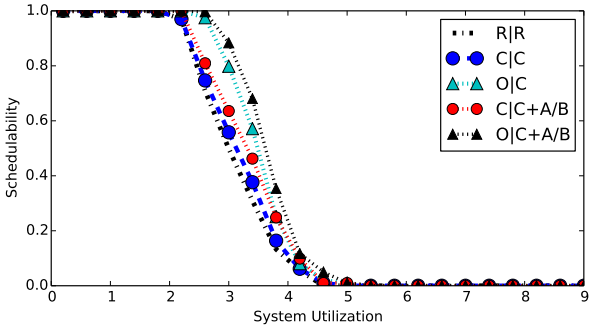
C-Light, Long, Light, Heavy, Small, Med., Few



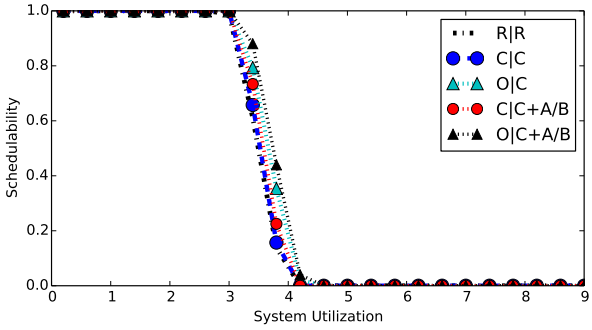
C-Light, Short, Med., Light, Small, Med., Many



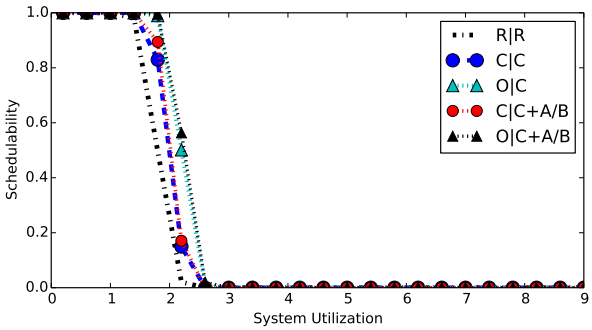
C-Light, Long, Med., Light, Small, Low, Many



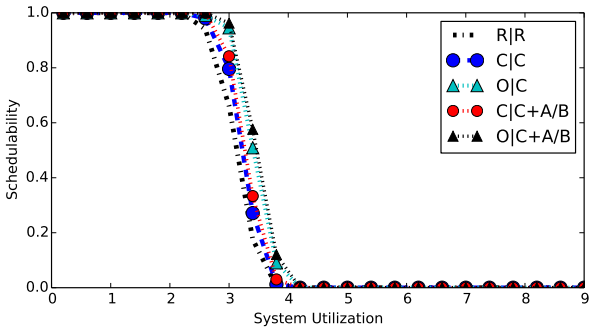
C-Heavy, Mod., Heavy, Heavy, Large, High, Many



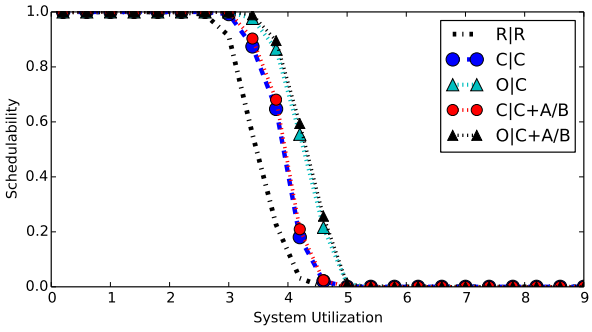
All-Mod., Short, Heavy, Light, Large, Low, Few



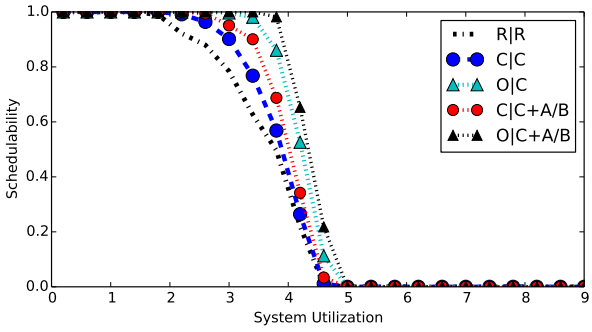
All-Mod., Short, Light, Heavy, Small, Med., Few



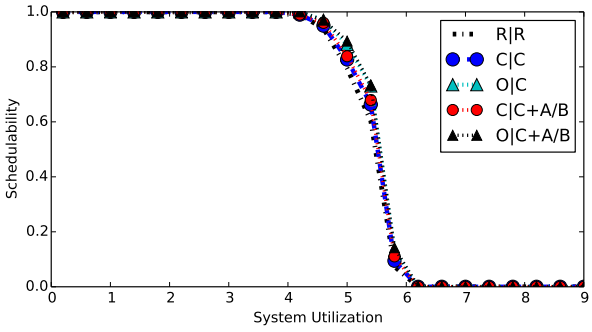
C-Light, Short, Heavy, Heavy, Large, High, Many



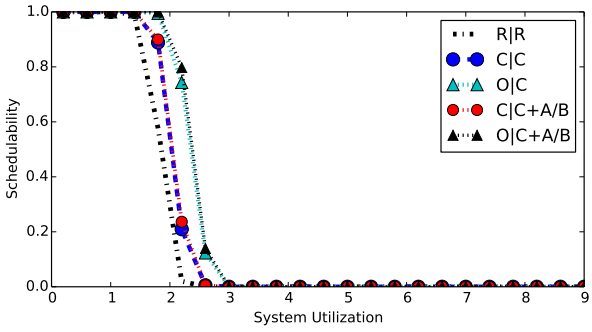
C-Light, Long, Med., Heavy, Large, Low, Few



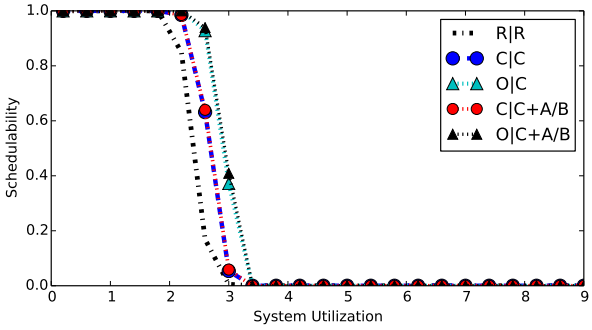
C-Heavy, Mod., Med., Light, Large, Med., Few



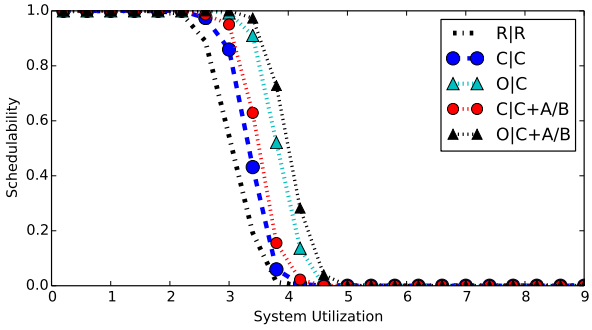
C-Heavy, Long, Heavy, Light, Small, Med., Many



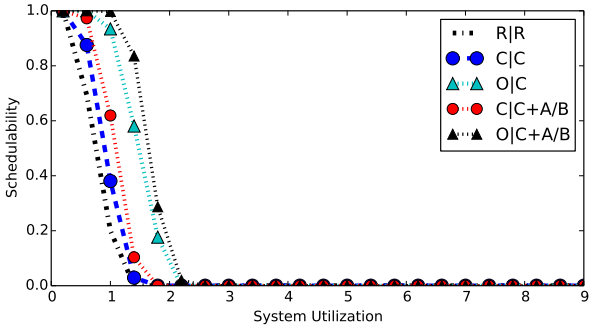
C-Light, Short, Light, Light, Small, Low, Few



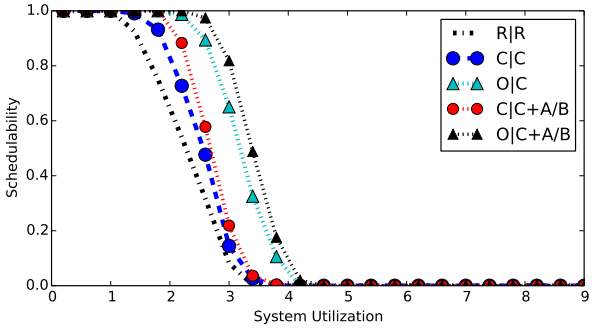
C-Light, Long, Light, Heavy, Small, Low, Many



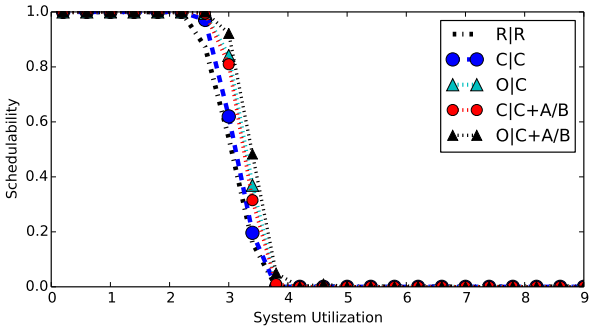
C-Light, Short, Med., Heavy, Small, High, Few



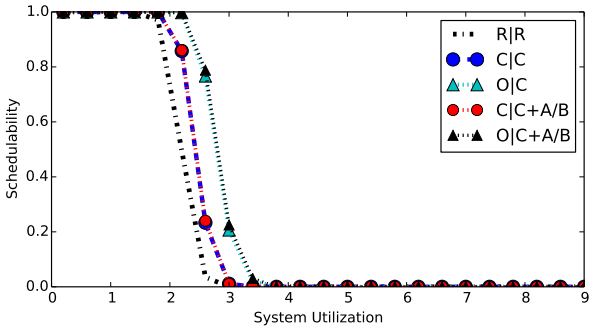
All-Mod., Mod., Light, Heavy, Large, High, Many



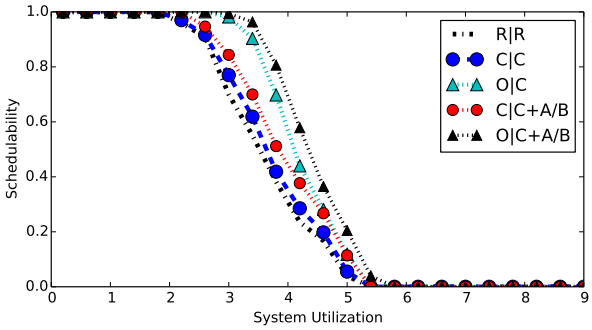
All-Mod., Mod., Med., Heavy, Large, Med., Few



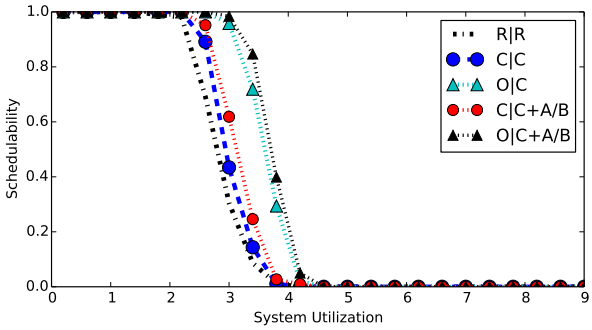
All-Mod., Mod., Heavy, Light, Large, Low, Few



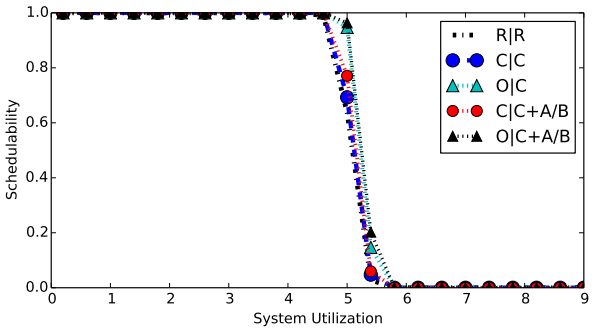
All-Mod., Long, Light, Heavy, Large, Low, Many



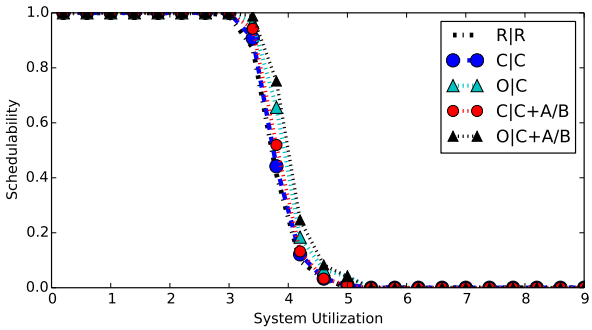
C-Heavy, Mod., Heavy, Heavy, Small, Med., Few



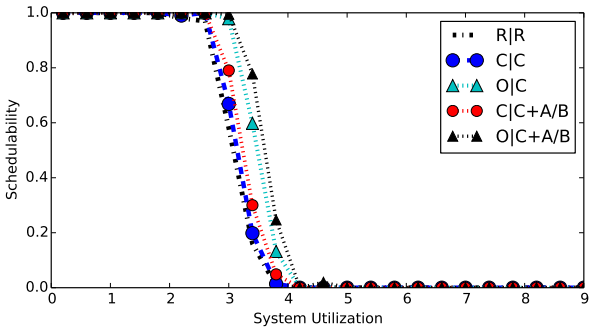
All-Mod., Short, Med., Heavy, Large, High, Many



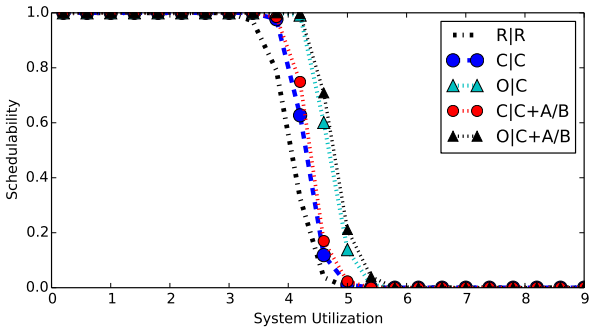
C-Heavy, Long, Med., Light, Large, Med., Many



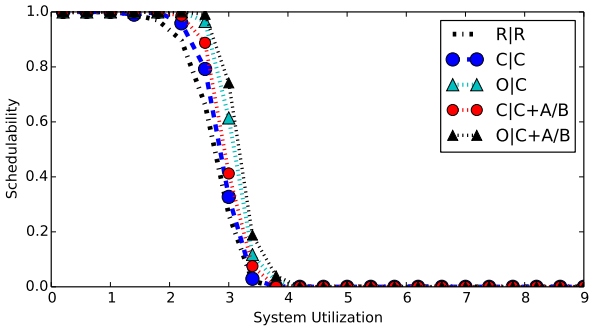
C-Heavy, Short, Med., Heavy, Large, Low, Few



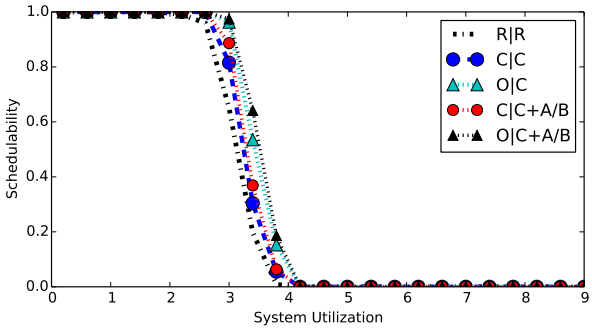
All-Mod., Mod., Heavy, Heavy, Small, Low, Few



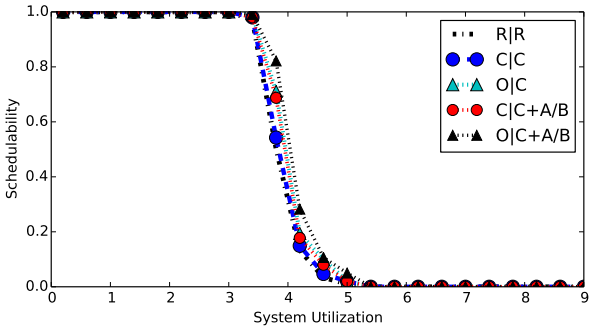
C-Light, Long, Med., Light, Large, Med., Many



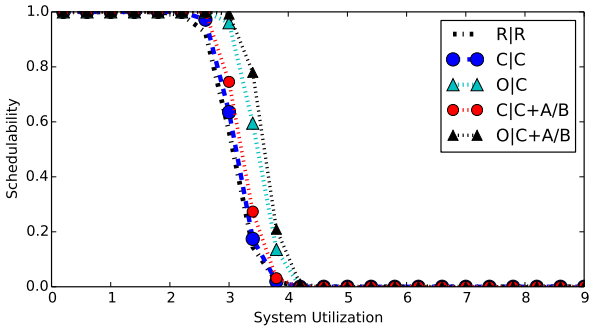
All-Mod., Mod., Heavy, Heavy, Large, Med., Few



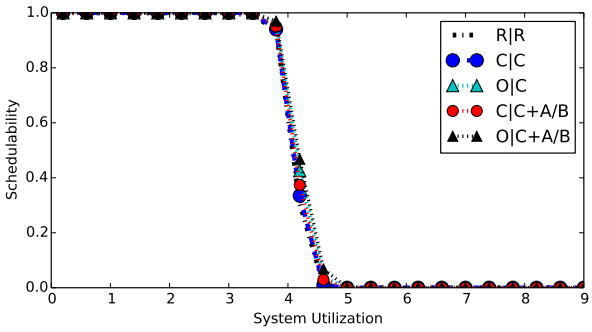
C-Light, Short, Heavy, Heavy, Large, Low, Few



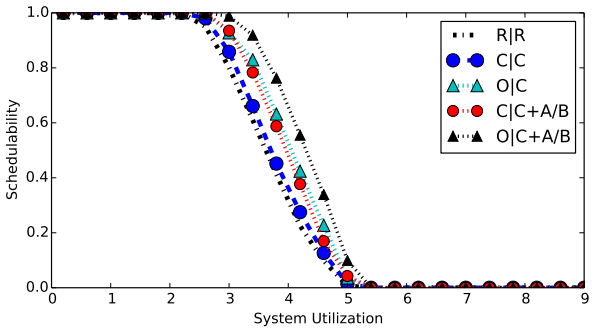
C-Heavy, Short, Med., Heavy, Small, High, Many



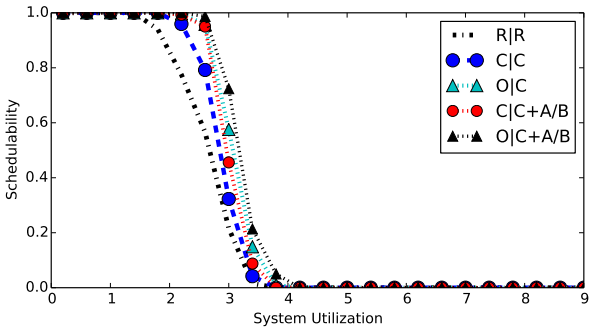
All-Mod., Mod., Heavy, Heavy, Small, Med., Few



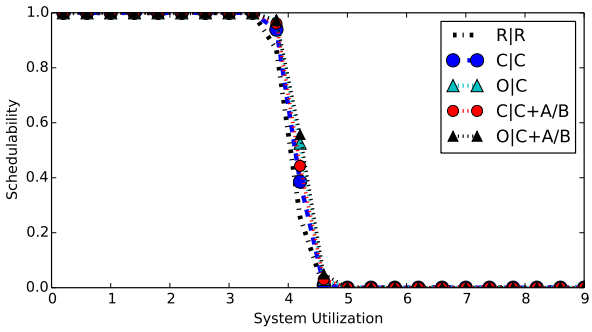
All-Mod., Long, Heavy, Heavy, Large, High, Many



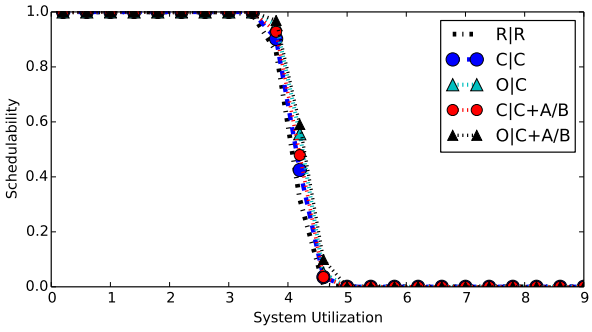
C-Heavy, Mod., Heavy, Light, Large, Low, Few



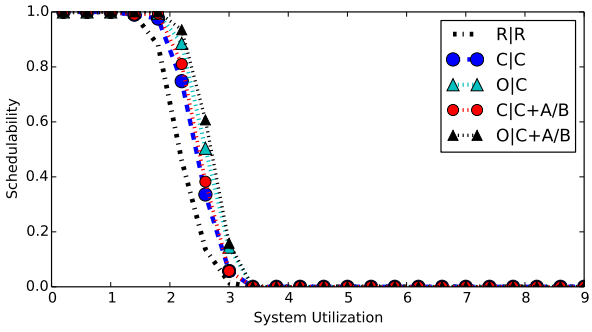
C-Light, Mod., Heavy, Heavy, Large, High, Few



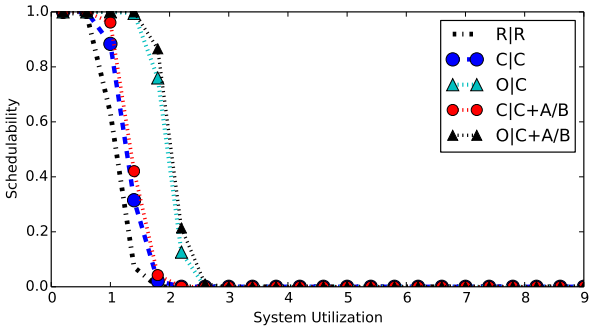
C-Light, Long, Heavy, Light, Large, High, Many



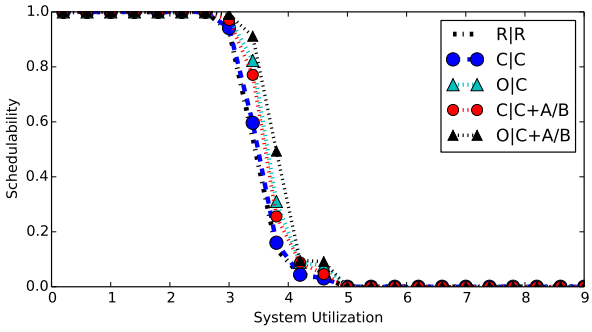
C-Light, Long, Heavy, Light, Large, High, Few



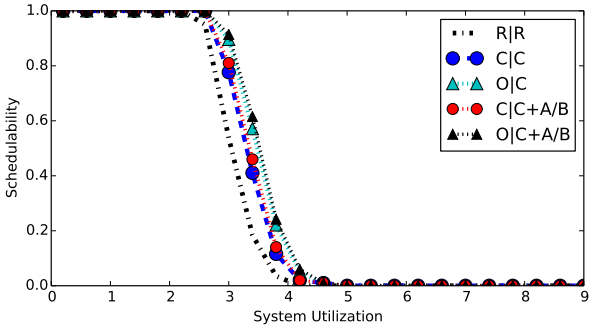
C-Heavy, Short, Light, Light, Small, High, Many



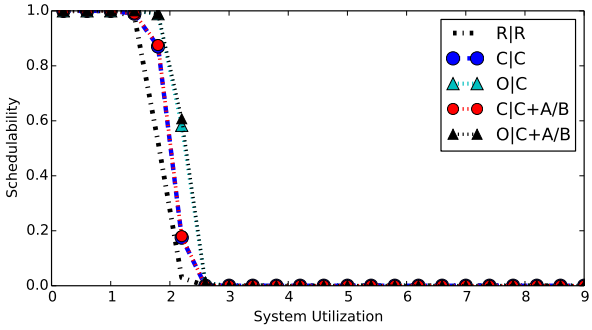
C-Light, Short, Light, Heavy, Large, Med., Many



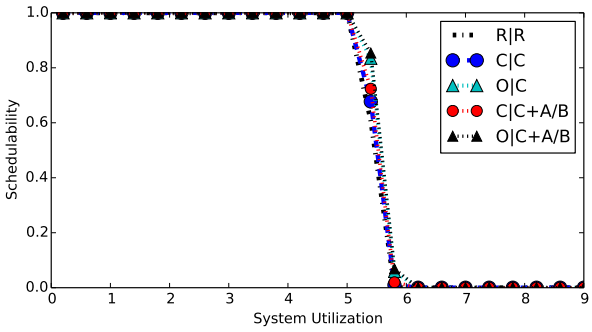
C-Heavy, Mod., Med., Heavy, Small, High, Many



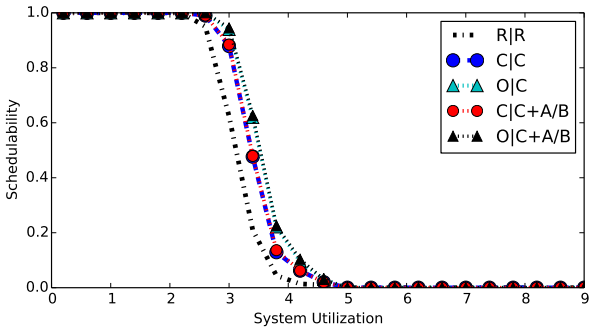
C-Heavy, Long, Light, Heavy, Small, Med., Many



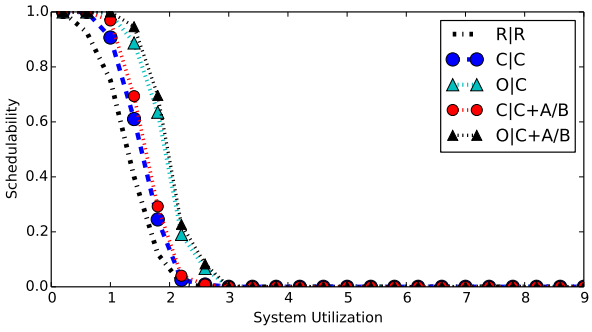
All-Mod., Short, Light, Heavy, Small, Low, Many



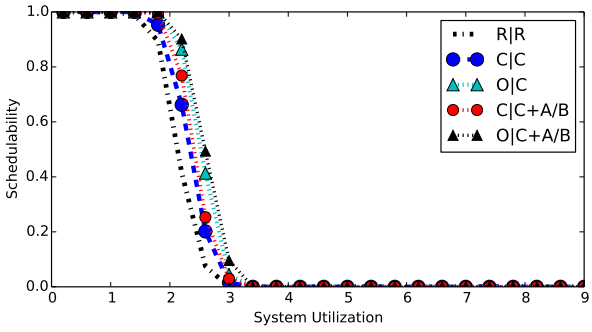
C-Heavy, Long, Med., Light, Small, Low, Many



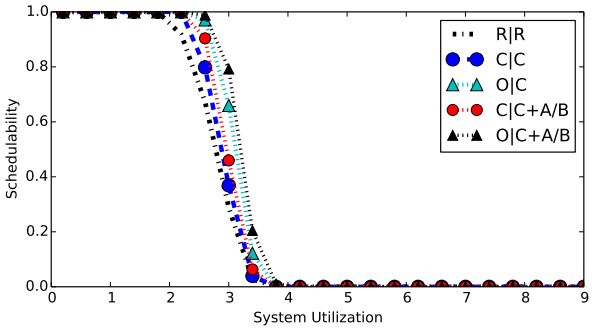
C-Heavy, Long, Light, Heavy, Small, Low, Many



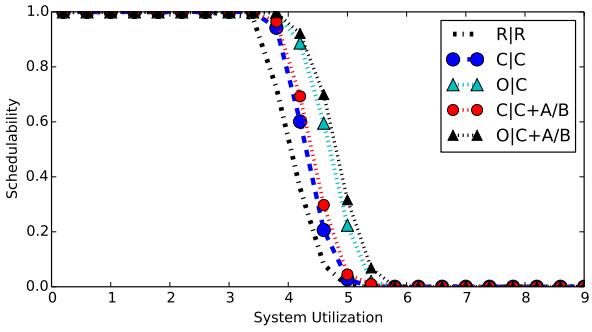
C-Heavy, Mod., Light, Heavy, Large, Med., Many



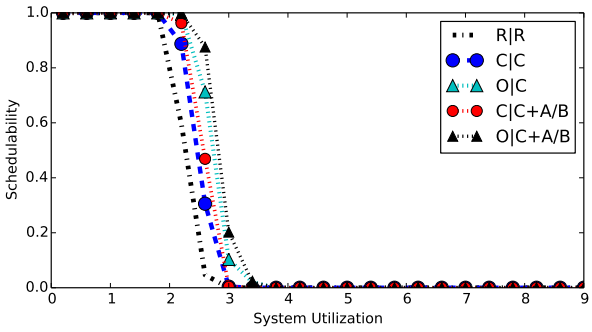
C-Heavy, Short, Light, Heavy, Small, High, Few



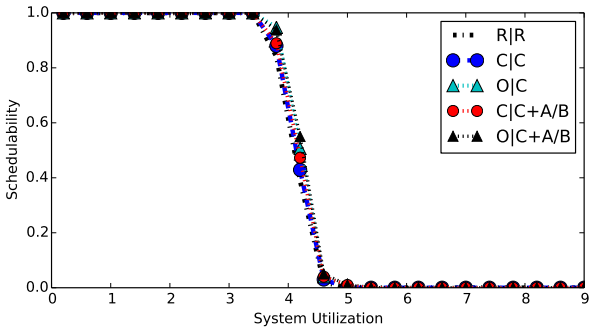
All-Mod., Mod., Heavy, Heavy, Large, Low, Many



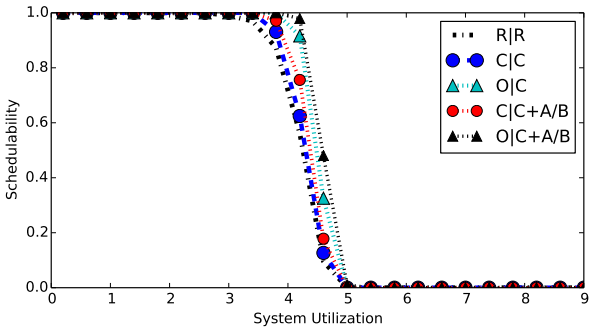
All-Mod., Long, Med., Heavy, Large, Med., Many



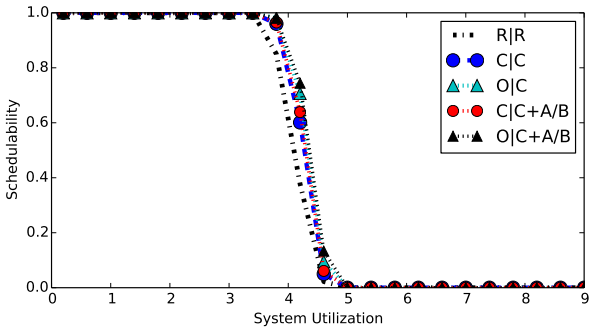
C-Light, Long, Light, Heavy, Small, High, Few



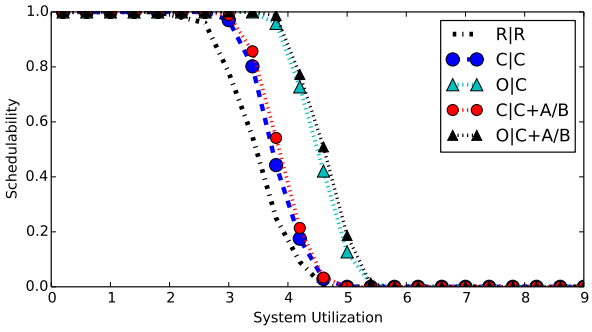
All-Mod., Long, Heavy, Heavy, Large, Low, Few



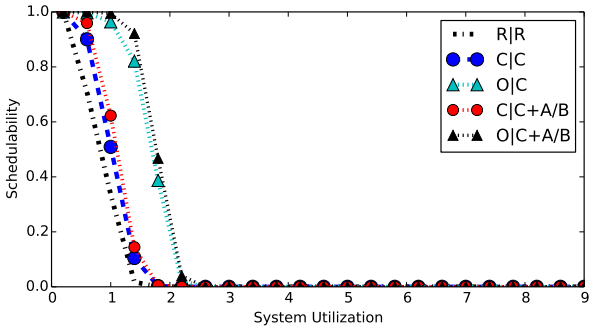
C-Heavy, Short, Med., Light, Large, High, Many



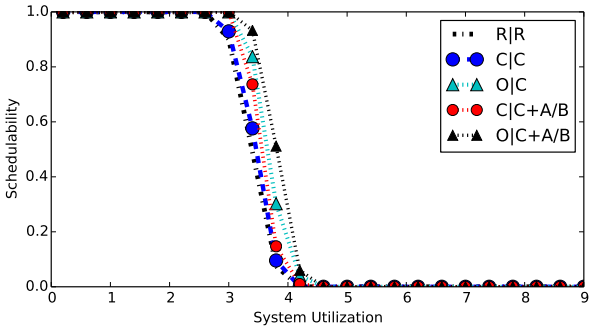
C-Light, Long, Heavy, Heavy, Small, High, Few



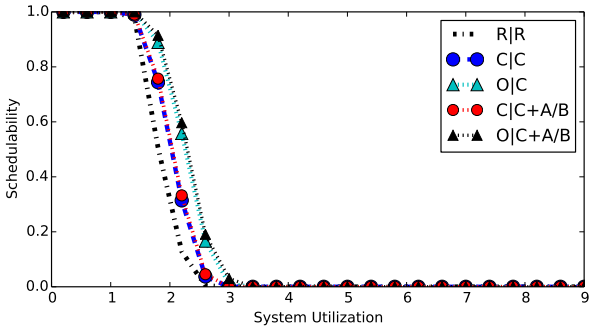
All-Mod., Short, Med., Light, Large, Low, Few



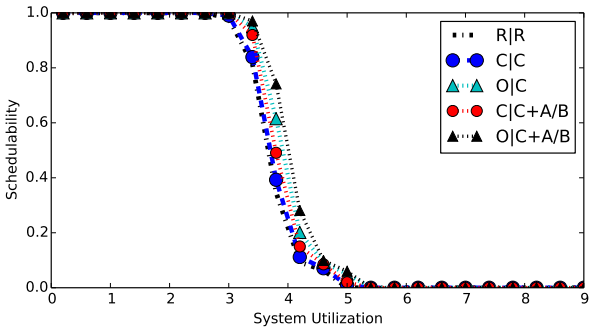
All-Mod., Mod., Light, Light, Large, Med., Few



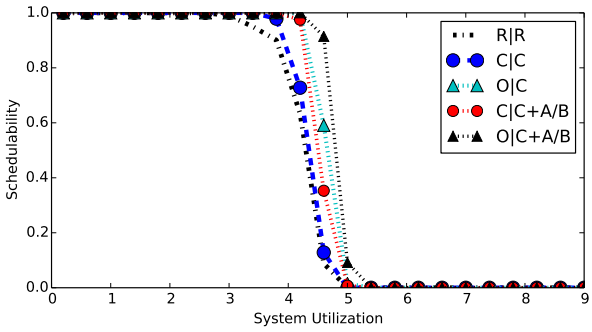
All-Mod., Mod., Heavy, Light, Small, High, Many



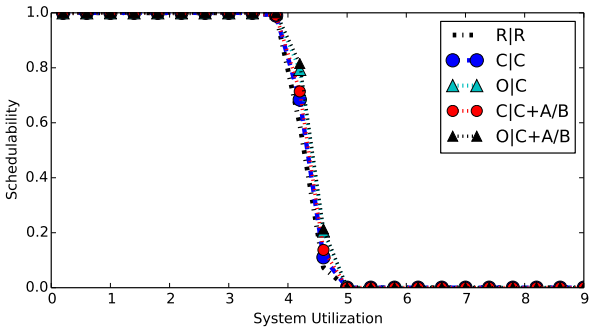
C-Heavy, Mod., Light, Light, Small, Med., Few



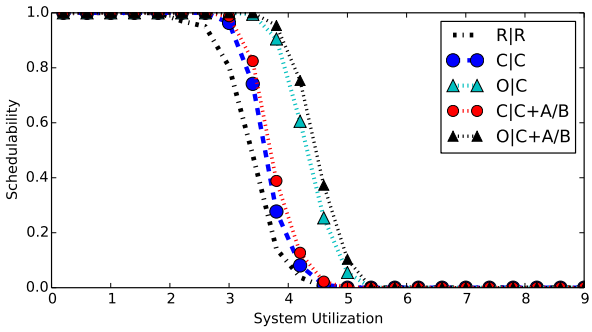
C-Heavy, Mod., Med., Heavy, Small, Low, Few



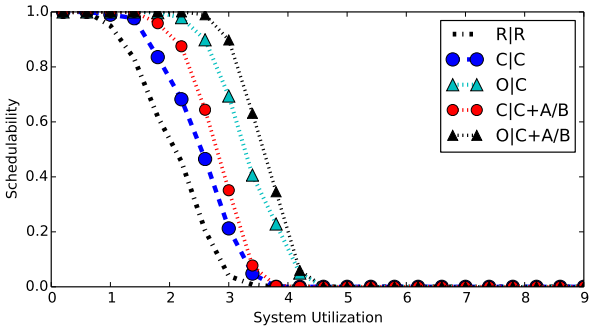
C-Heavy, Mod., Med., Light, Small, High, Many



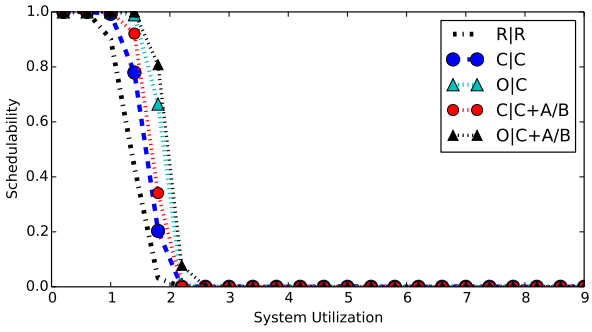
C-Light, Long, Heavy, Light, Small, Low, Few



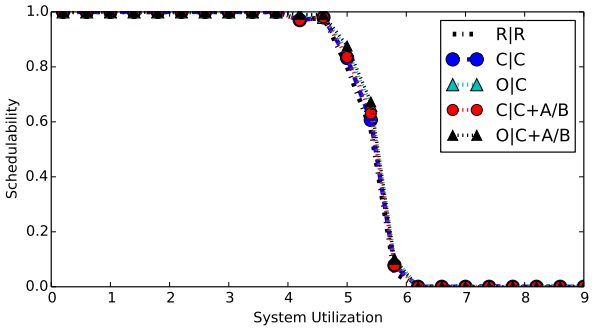
All-Mod., Short, Med., Light, Large, Med., Few



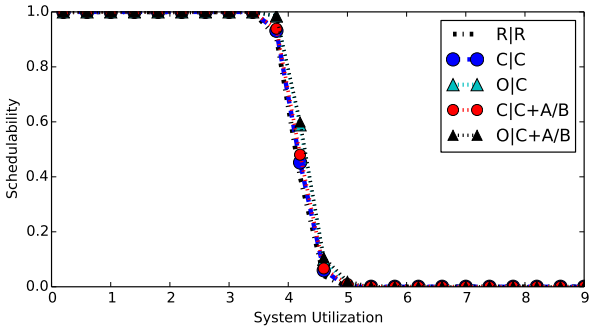
C-Light, Mod., Med., Light, Large, Low, Many



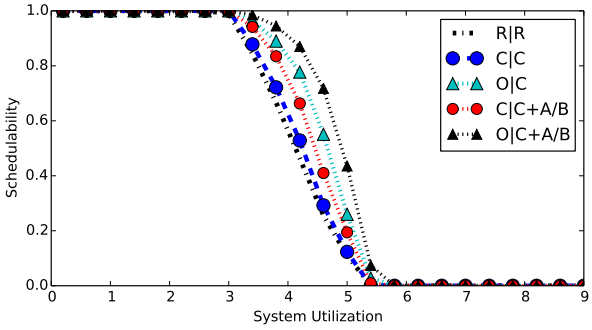
All-Mod., Mod., Light, Light, Small, High, Few



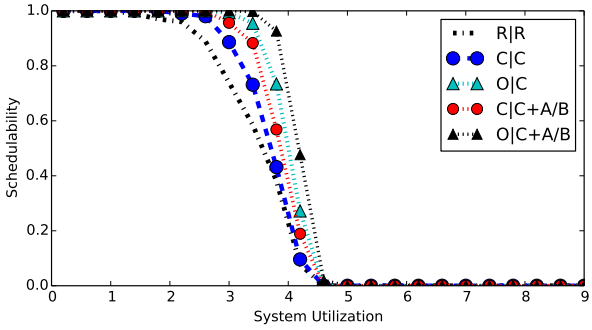
C-Heavy, Long, Heavy, Light, Large, Low, Few



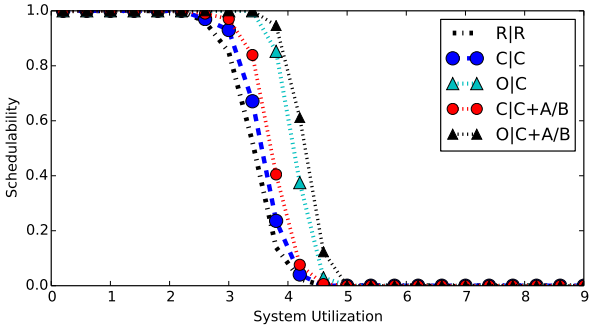
All-Mod., Long, Heavy, Light, Small, High, Many



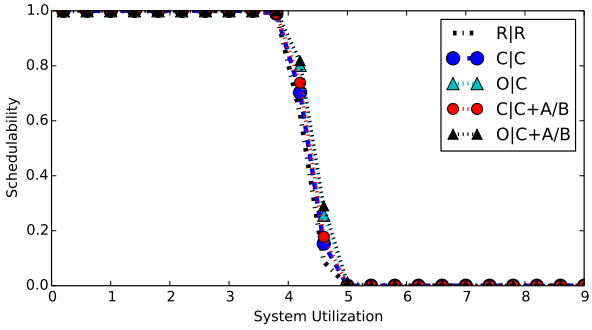
C-Heavy, Mod., Heavy, Light, Small, Low, Few



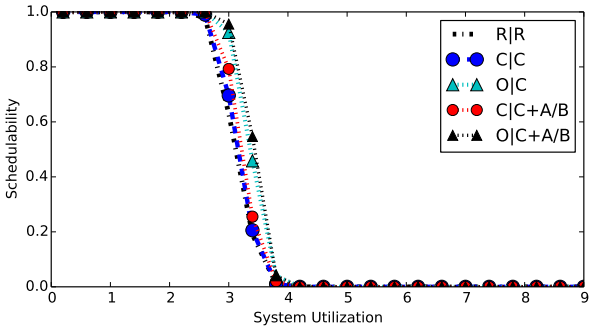
C-Heavy, Mod., Med., Light, Large, High, Many



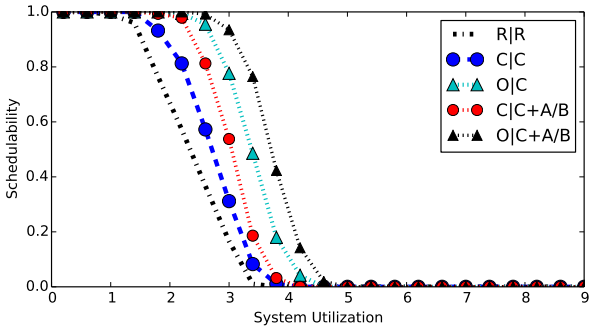
All-Mod., Short, Med., Heavy, Small, High, Many



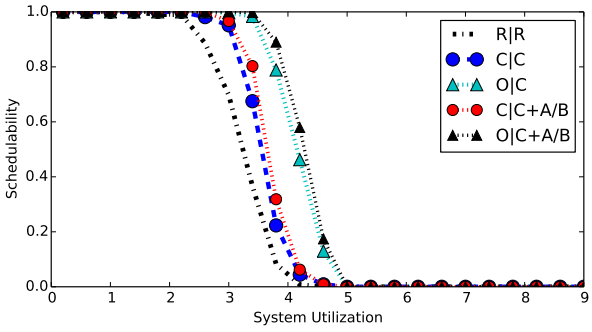
C-Light, Long, Heavy, Light, Small, Low, Many



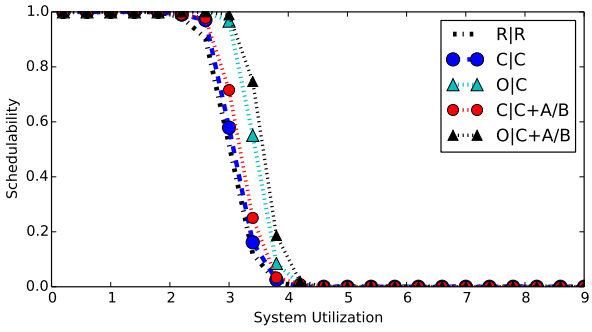
All-Mod., Short, Heavy, Heavy, Large, Med., Many



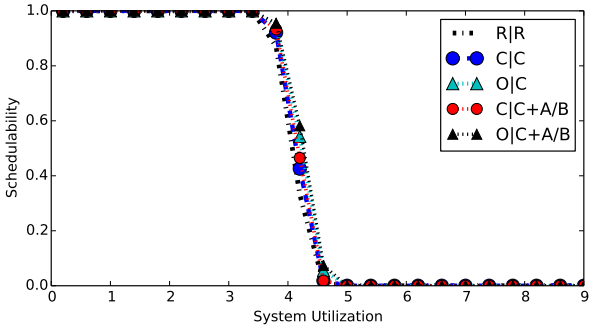
All-Mod., Mod., Med., Light, Large, High, Many



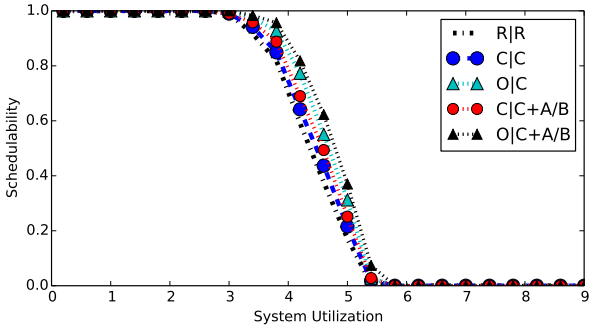
C-Light, Short, Med., Heavy, Small, Low, Many



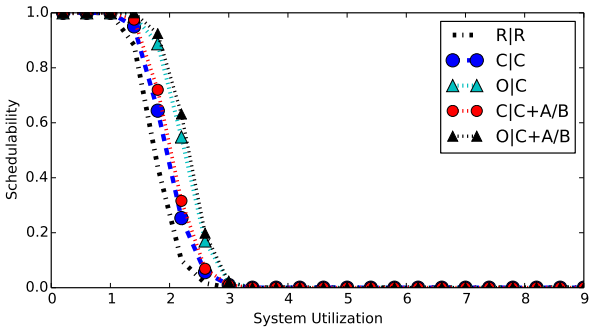
All-Mod., Mod., Heavy, Heavy, Small, Med., Many



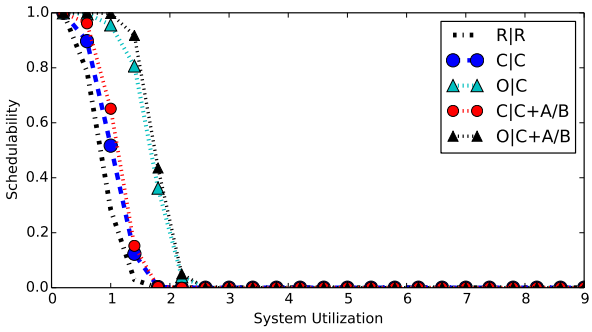
C-Light, Long, Heavy, Light, Large, Low, Many



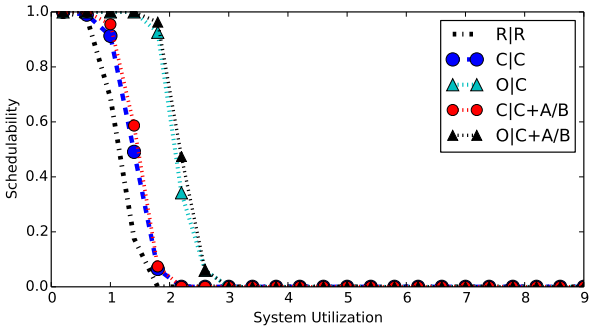
C-Heavy, Short, Heavy, Light, Large, Low, Few



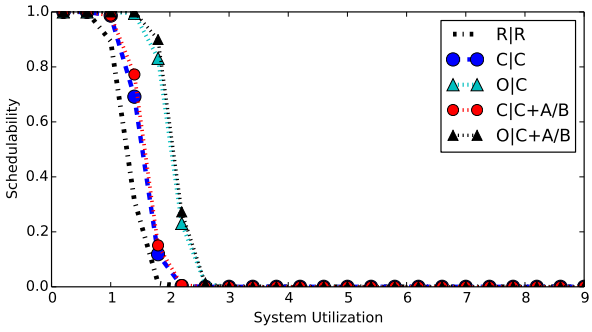
C-Heavy, Mod., Light, Light, Small, High, Many



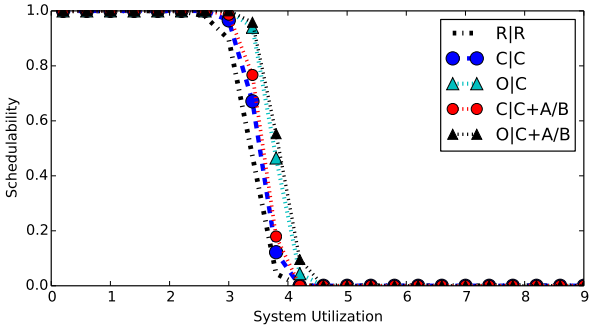
All-Mod., Mod., Light, Light, Large, Low, Few



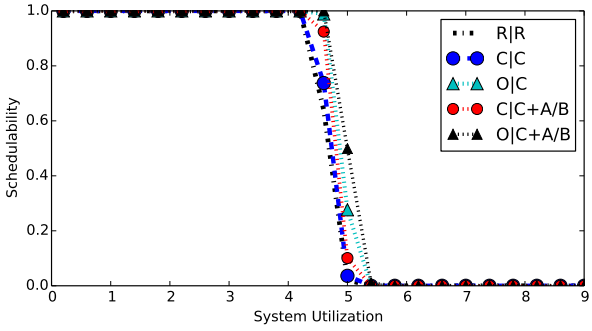
C-Light, Short, Light, Light, Large, Med., Many



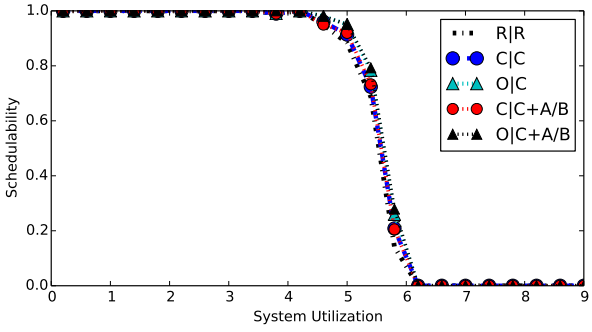
All-Mod., Short, Light, Heavy, Large, Low, Few



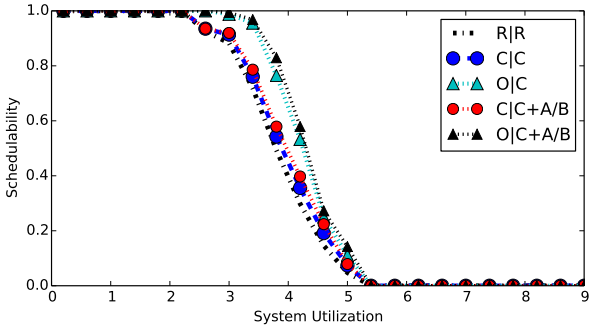
C-Light, Short, Heavy, Heavy, Small, Med., Few



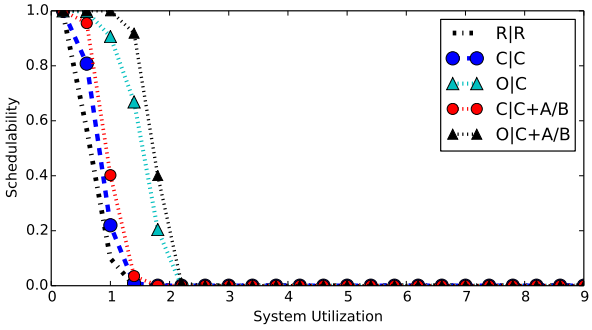
C-Heavy, Short, Med., Light, Small, High, Few



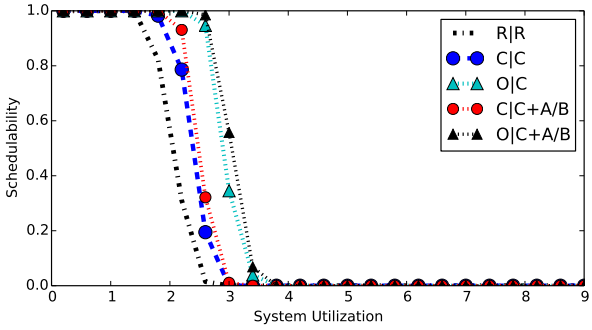
C-Heavy, Long, Heavy, Light, Small, Low, Many



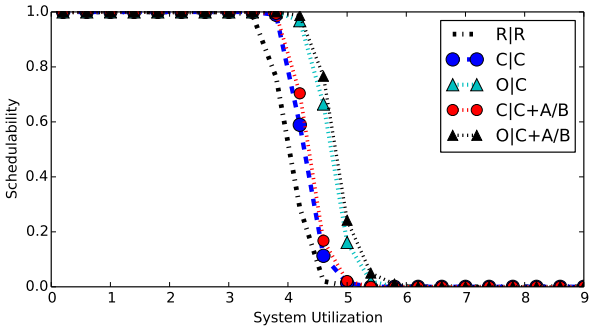
C-Heavy, Short, Heavy, Heavy, Small, High, Few



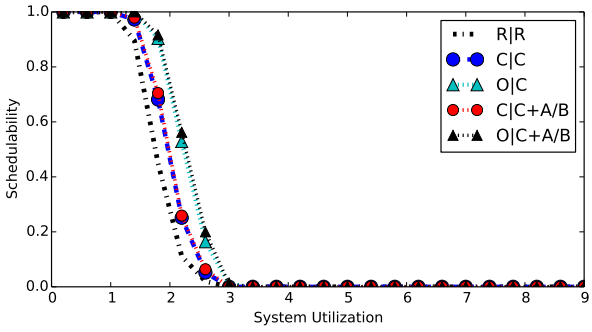
C-Light, Mod., Light, Light, Large, High, Few



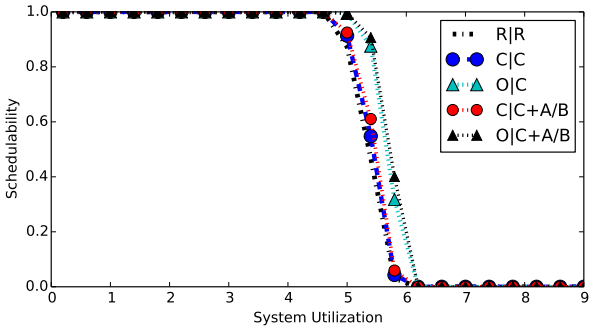
C-Light, Long, Light, Light, Large, High, Few



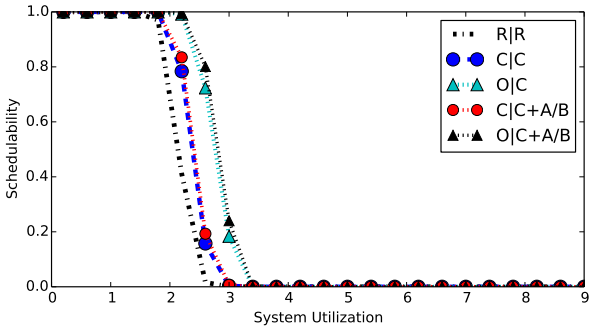
C-Light, Long, Med., Light, Large, Med., Few



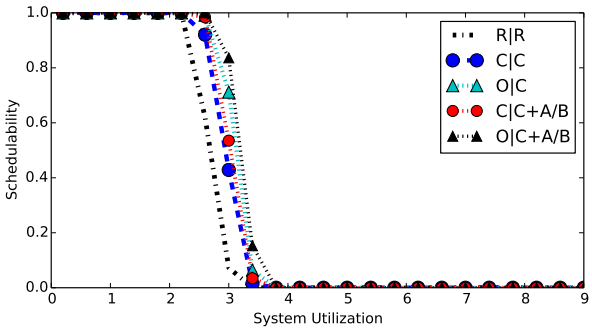
C-Heavy, Mod., Light, Heavy, Small, Med., Few



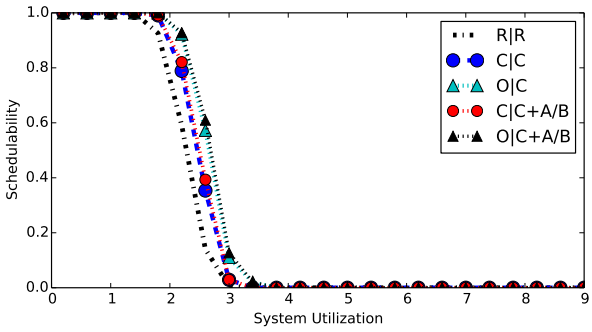
All-Mod., Long, Med., Light, Small, Low, Few



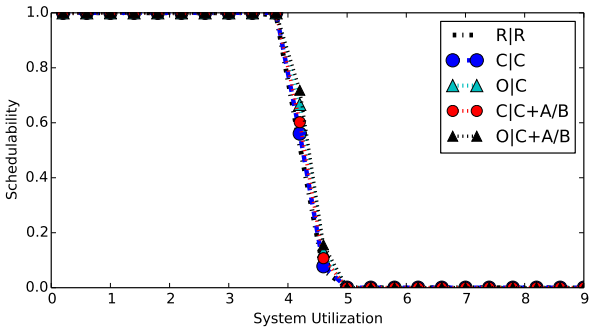
All-Mod., Long, Light, Heavy, Large, Med., Many



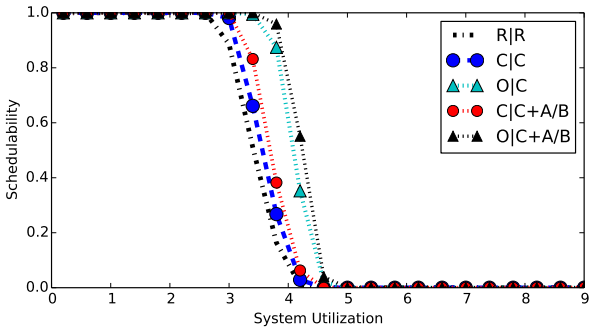
All-Mod., Long, Light, Light, Small, High, Many



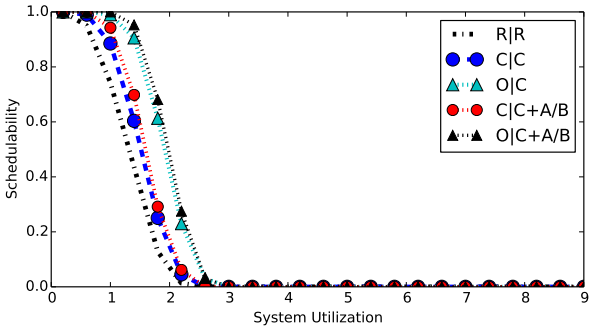
C-Heavy, Short, Light, Light, Small, Med., Many



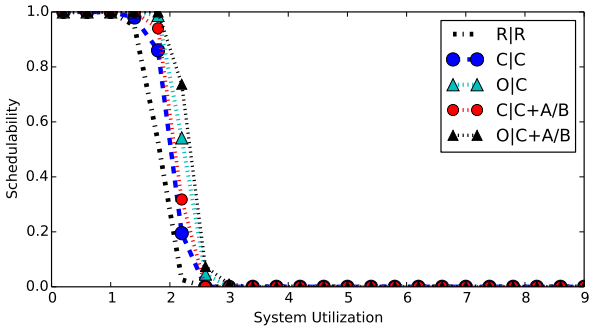
All-Mod., Long, Heavy, Heavy, Small, High, Few



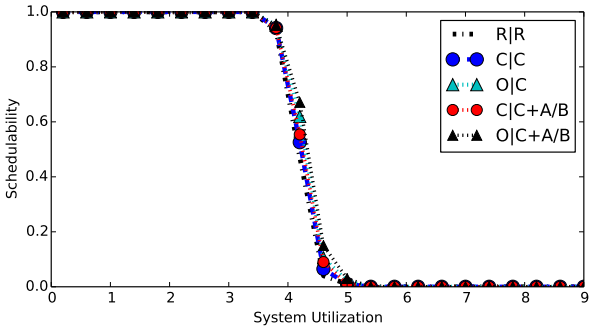
All-Mod., Short, Med., Heavy, Small, High, Few



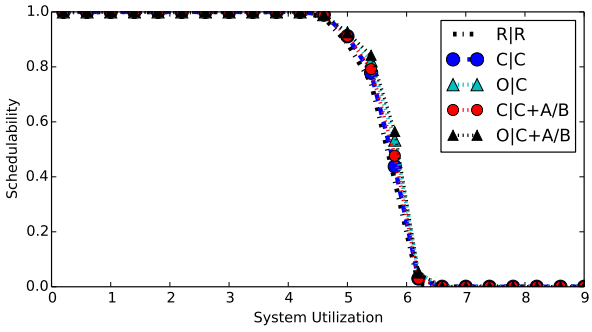
C-Heavy, Mod., Light, Heavy, Large, Med., Few



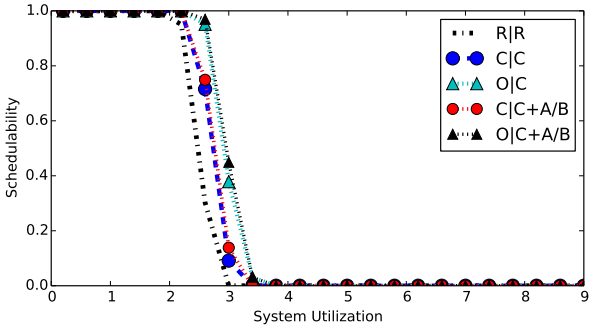
All-Mod., Short, Light, Light, Small, High, Many



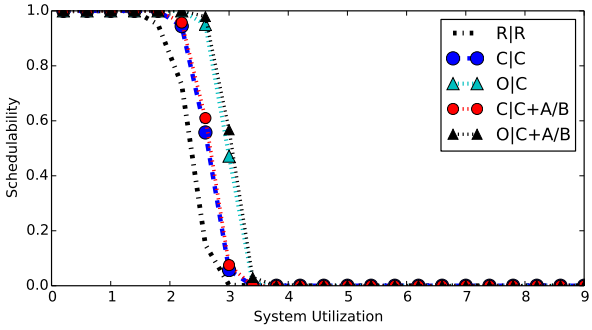
All-Mod., Long, Heavy, Heavy, Small, Med., Many



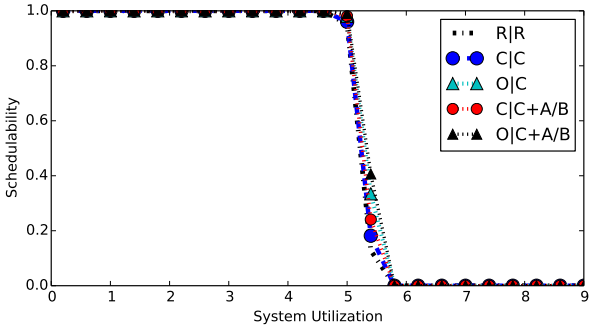
C-Heavy, Long, Heavy, Heavy, Small, Med., Few



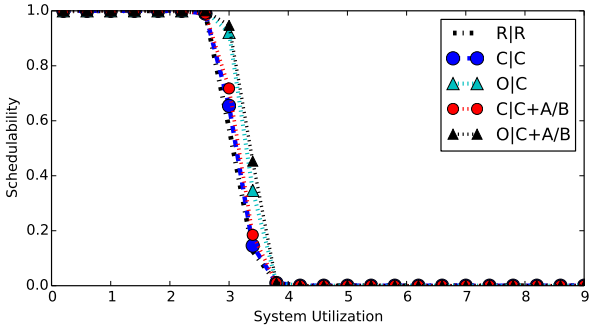
All-Mod., Long, Light, Heavy, Small, Med., Many



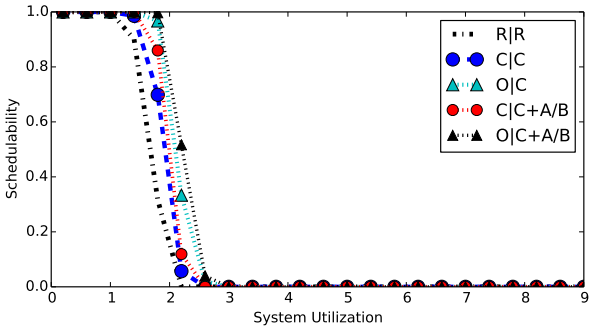
All-Mod., Long, Light, Light, Large, Med., Many



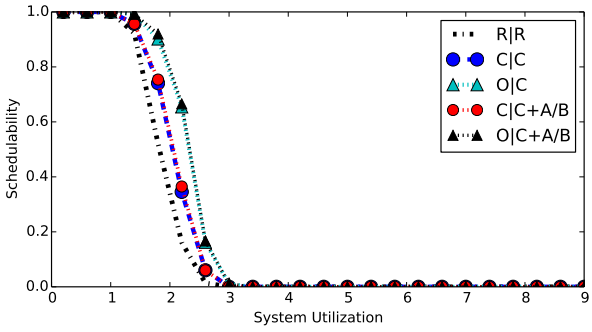
C-Heavy, Long, Med., Light, Small, High, Many



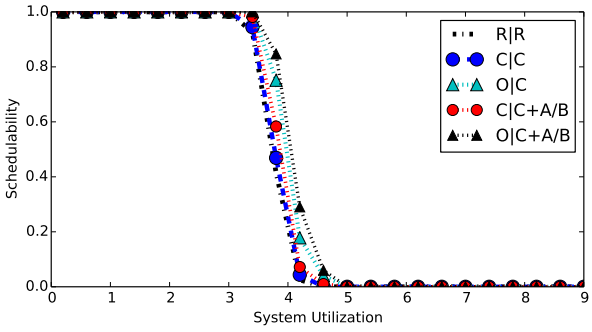
All-Mod., Short, Heavy, Heavy, Large, High, Few



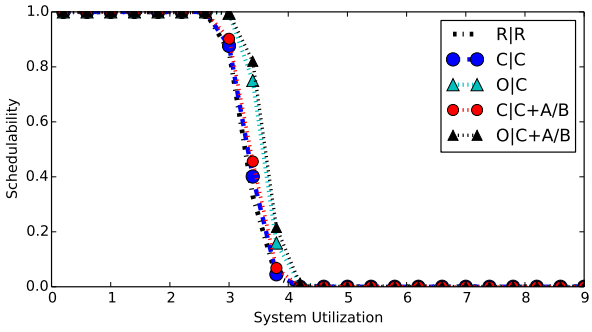
All-Mod., Short, Light, Heavy, Small, High, Few



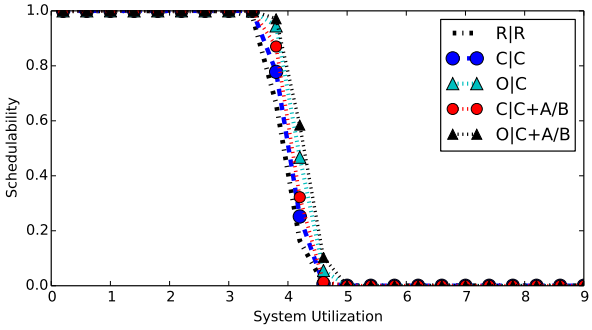
C-Heavy, Mod., Light, Light, Small, Low, Many



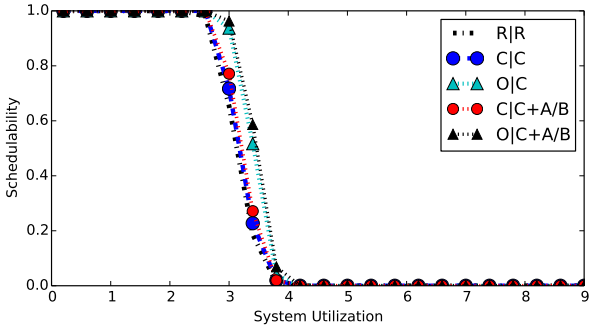
All-Mod., Short, Heavy, Light, Small, Low, Few



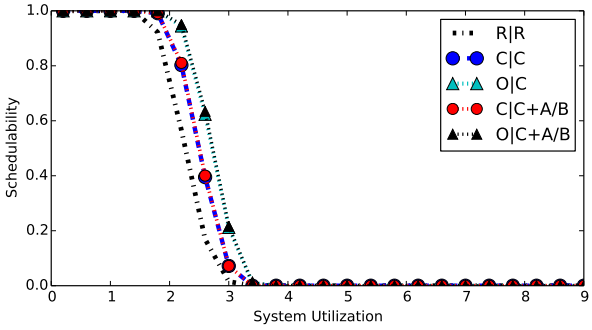
All-Mod., Short, Heavy, Heavy, Small, High, Many



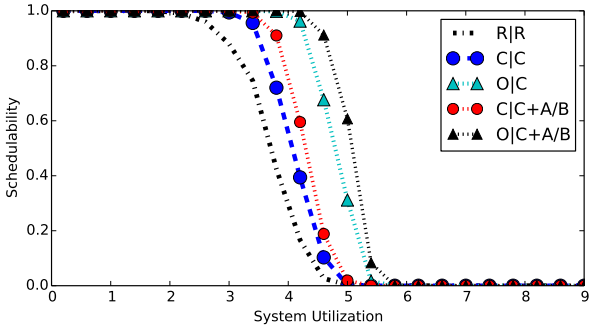
C-Light, Short, Heavy, Light, Small, Low, Few



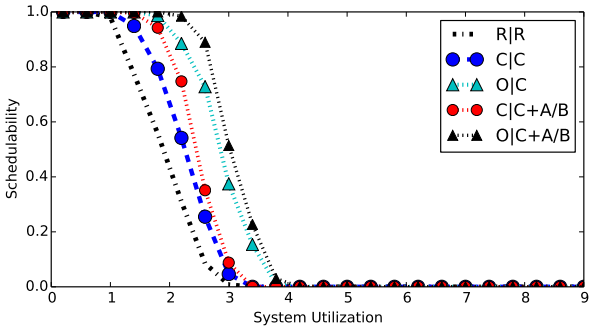
All-Mod., Short, Heavy, Heavy, Large, Low, Few



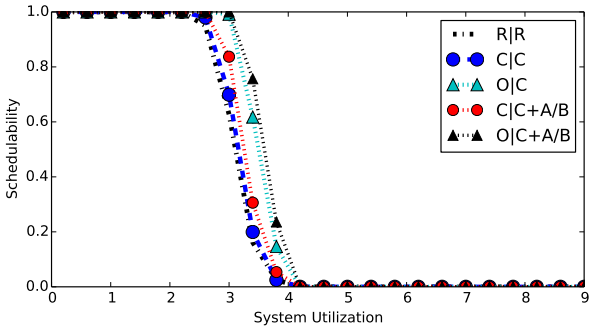
C-Heavy, Short, Light, Light, Small, Low, Few



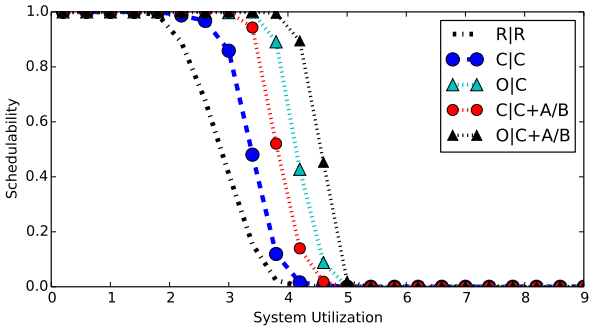
All-Mod., Mod., Med., Light, Small, Med., Few



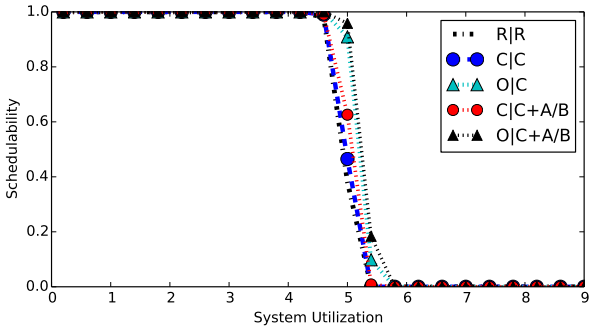
C-Light, Mod., Med., Heavy, Large, Med., Few



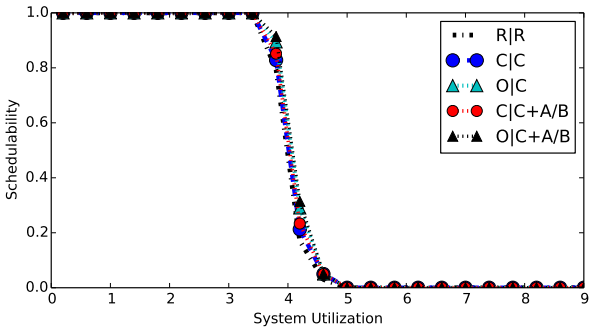
All-Mod., Mod., Heavy, Heavy, Small, Low, Many



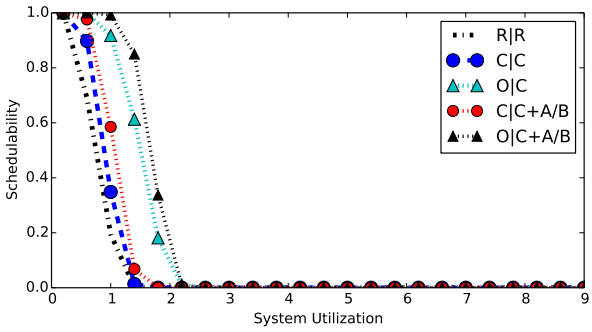
C-Light, Mod., Med., Light, Small, High, Many



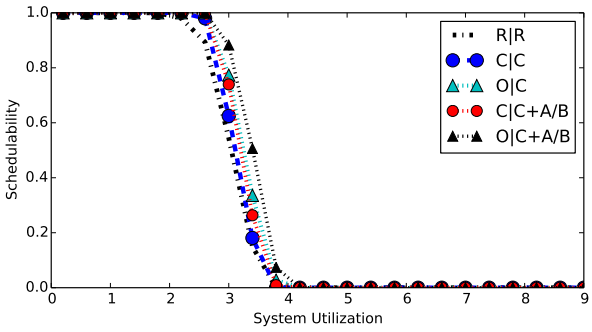
C-Heavy, Short, Med., Light, Small, Low, Many



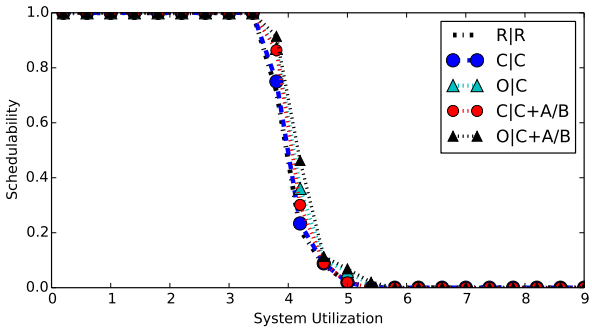
All-Mod., Long, Heavy, Light, Large, Med., Many



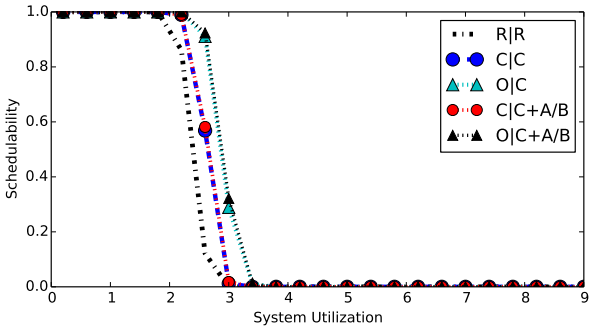
All-Mod., Mod., Light, Heavy, Large, High, Few



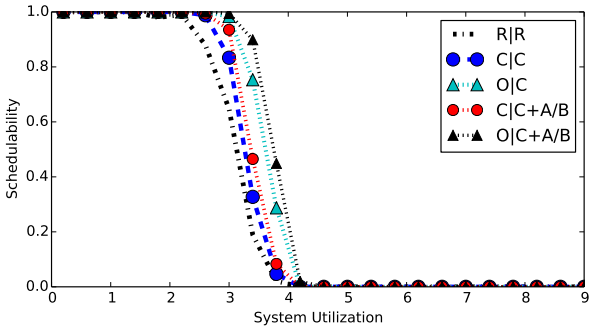
All-Mod., Mod., Heavy, Light, Large, Med., Many



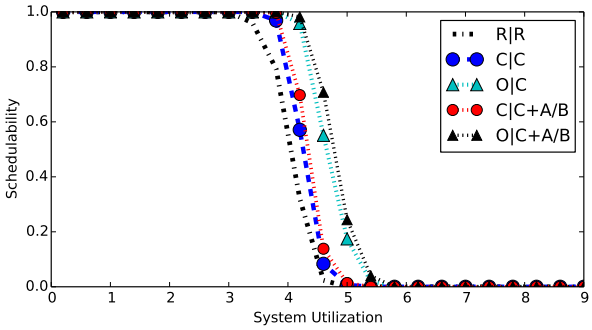
C-Heavy, Short, Med., Heavy, Small, Med., Few



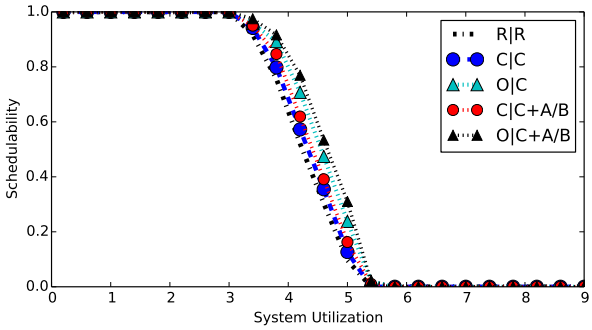
C-Light, Long, Light, Heavy, Small, Low, Few



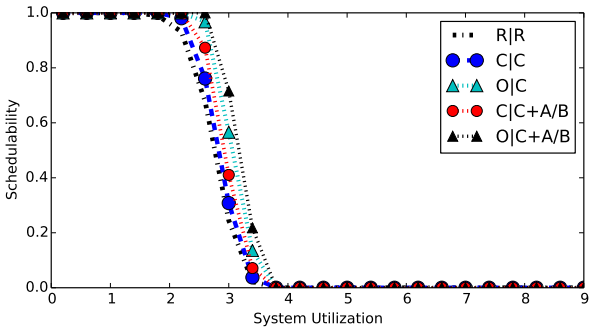
C-Light, Mod., Heavy, Heavy, Small, Low, Few



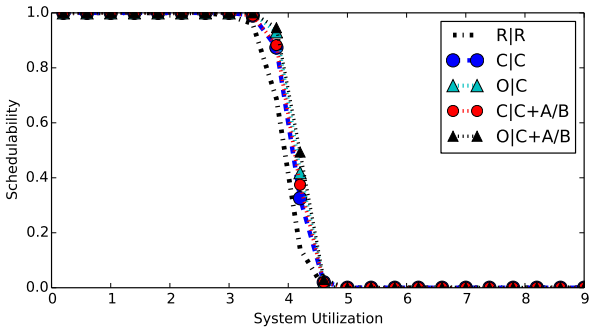
C-Light, Long, Med., Light, Large, High, Few



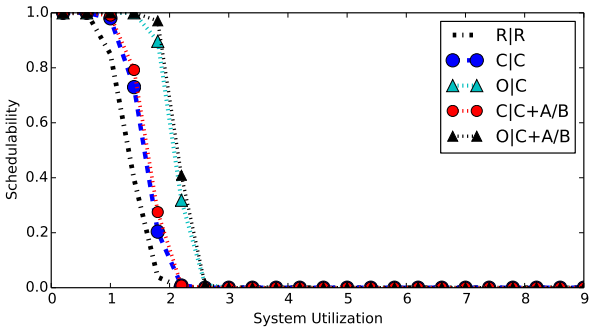
C-Heavy, Short, Heavy, Light, Large, Med., Many



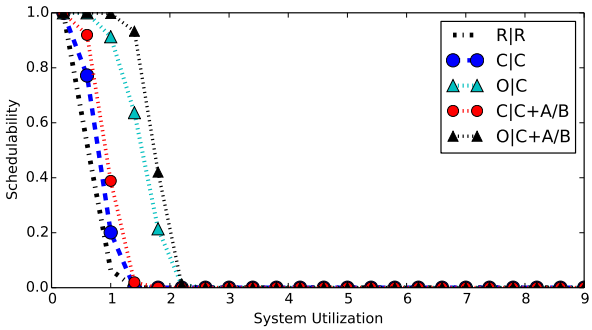
All-Mod., Mod., Heavy, Heavy, Large, Med., Many



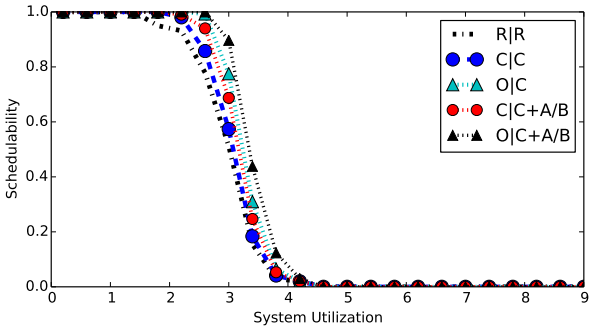
C-Light, Long, Heavy, Heavy, Large, High, Few



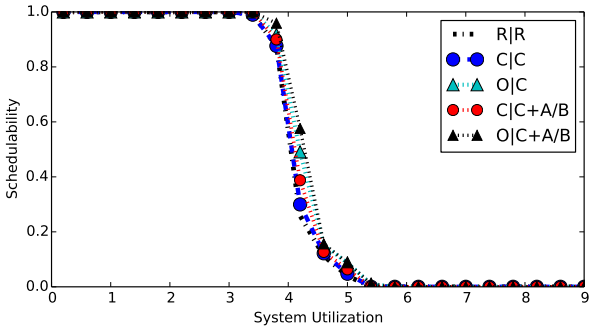
All-Mod., Short, Light, Light, Large, Med., Many



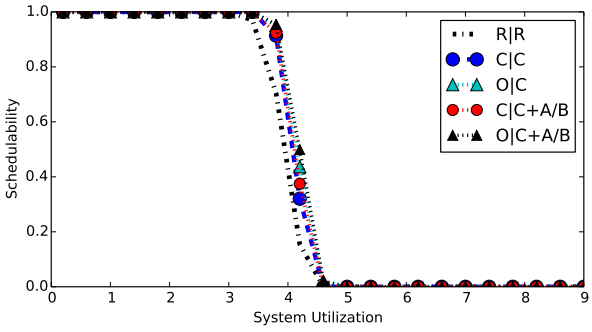
C-Light, Mod., Light, Light, Large, High, Many



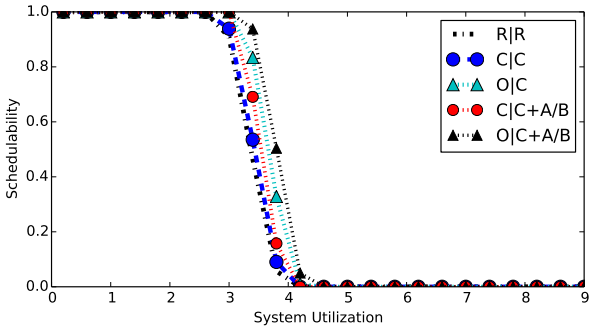
C-Heavy, Mod., Med., Heavy, Large, High, Few



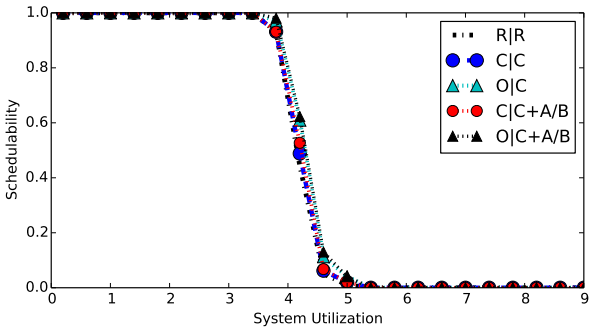
C-Heavy, Short, Med., Heavy, Small, Low, Many



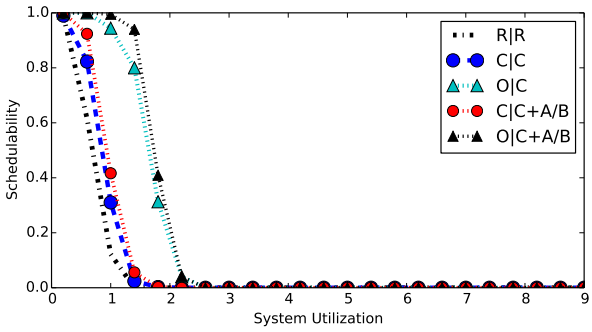
C-Light, Long, Heavy, Heavy, Large, High, Many



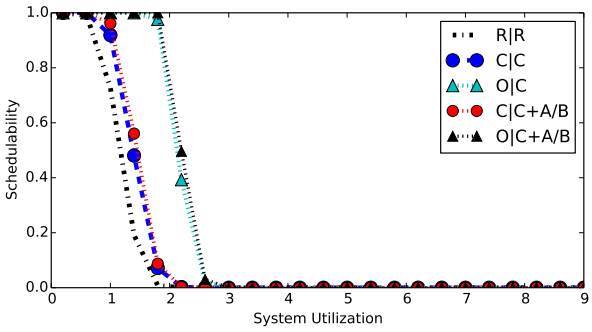
All-Mod., Mod., Heavy, Light, Small, High, Few



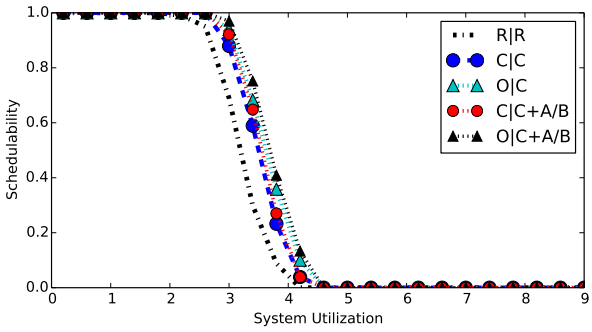
All-Mod., Long, Heavy, Light, Small, Med., Few



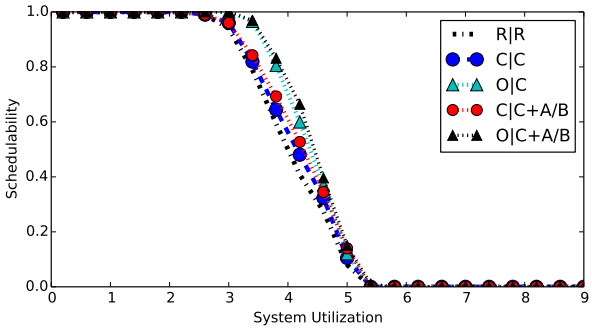
C-Light, Mod., Light, Light, Large, Low, Few



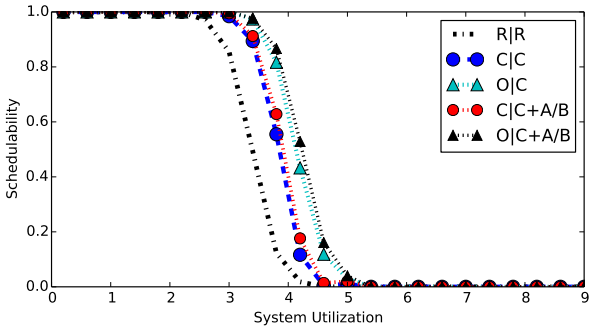
C-Light, Short, Light, Light, Large, Low, Many



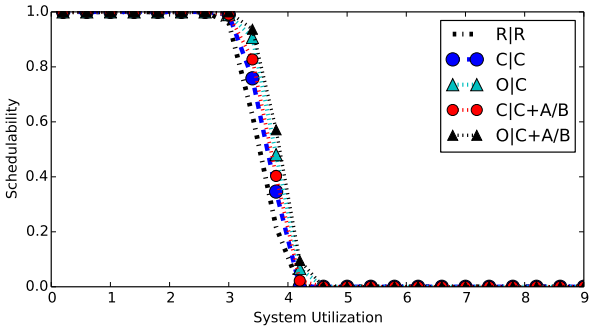
C-Heavy, Long, Light, Light, Small, High, Few



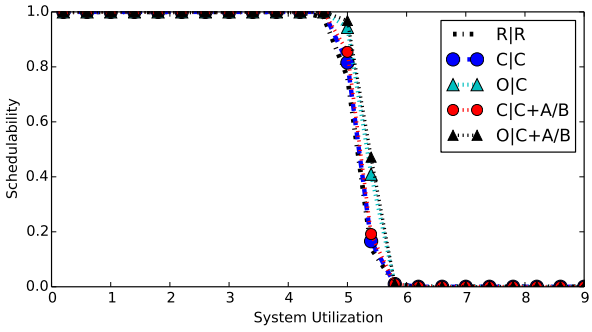
C-Heavy, Short, Heavy, Heavy, Large, Low, Few



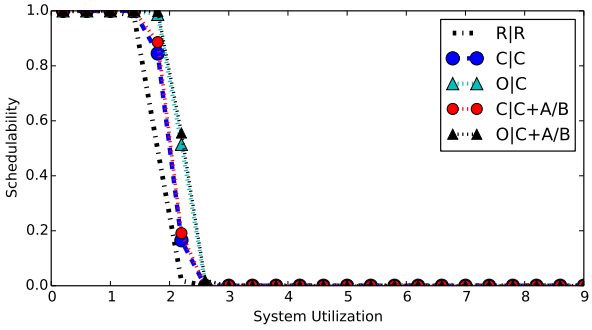
C-Light, Long, Med., Heavy, Large, Med., Few



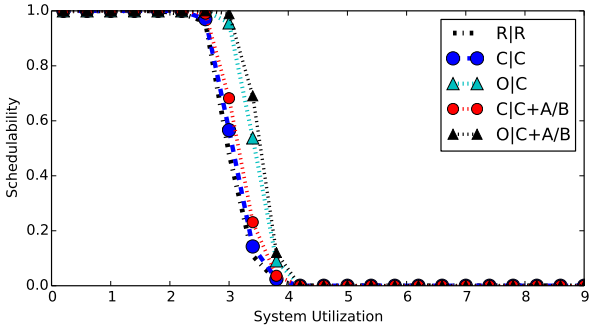
C-Light, Short, Heavy, Light, Large, Low, Few



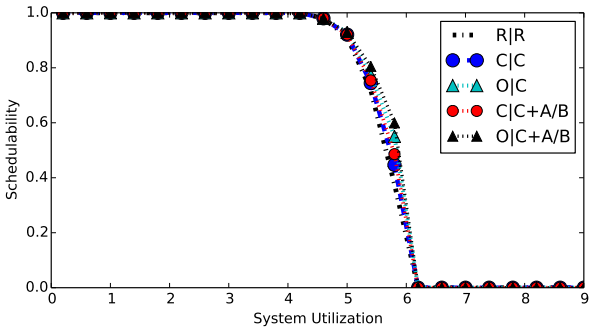
C-Heavy, Long, Med., Heavy, Large, Med., Many



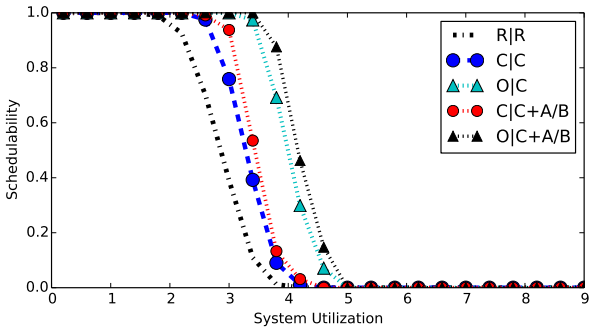
All-Mod., Short, Light, Heavy, Small, Med., Many



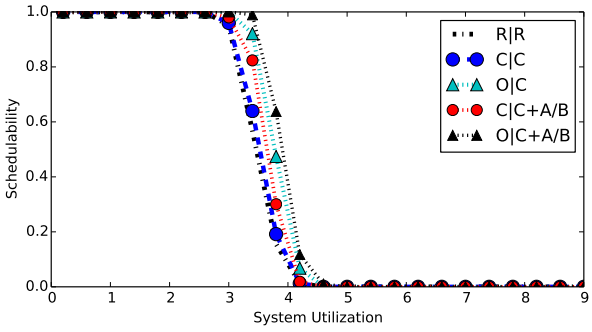
All-Mod., Mod., Heavy, Heavy, Small, High, Many



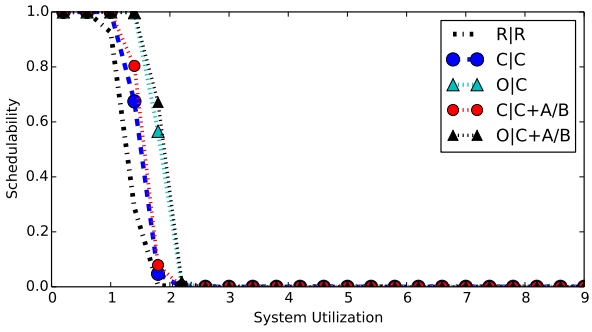
C-Heavy, Long, Heavy, Heavy, Small, Med., Many



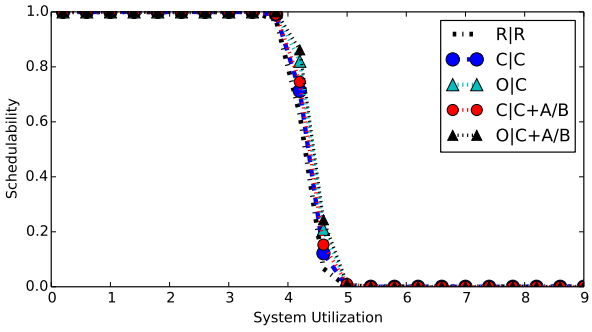
C-Light, Short, Med., Light, Large, Med., Many



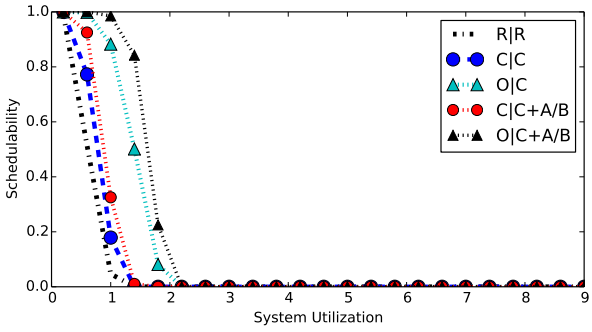
All-Mod., Mod., Heavy, Light, Small, Low, Many



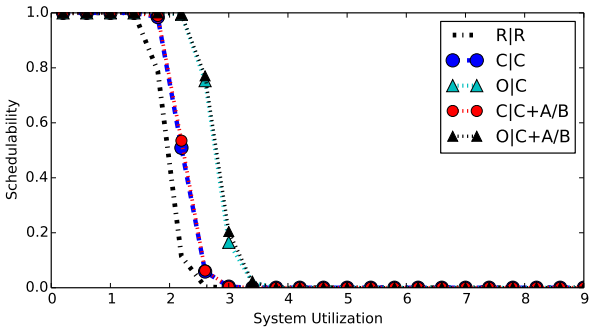
C-Light, Mod., Light, Heavy, Small, Med., Many



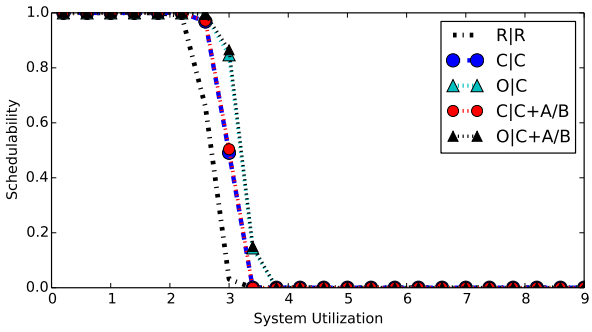
C-Light, Long, Heavy, Light, Small, Med., Many



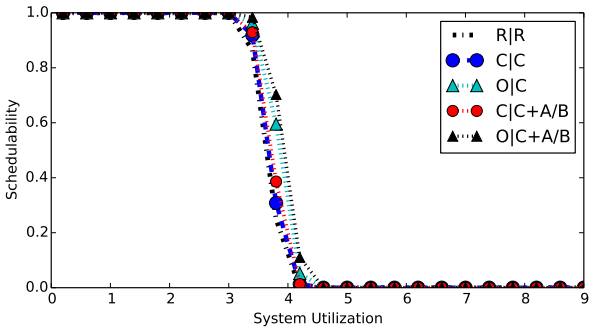
C-Light, Mod., Light, Heavy, Large, High, Many



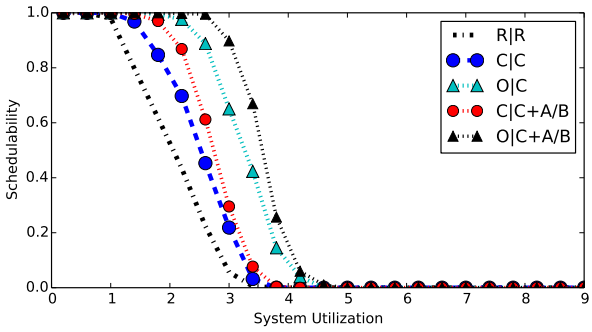
C-Light, Long, Light, Heavy, Large, Low, Many



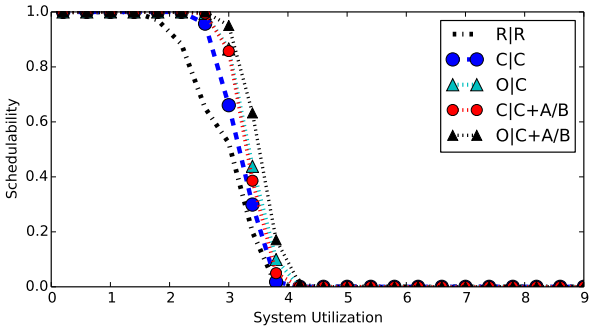
C-Light, Long, Light, Light, Small, Low, Few



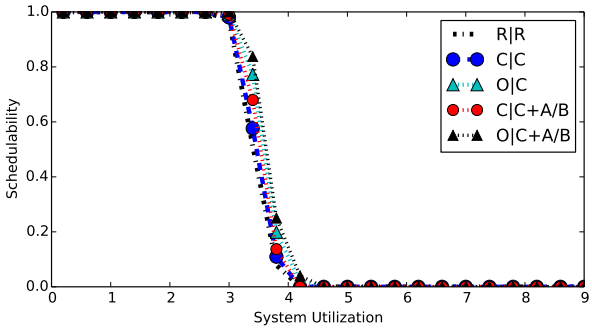
All-Mod., Short, Heavy, Light, Small, High, Many



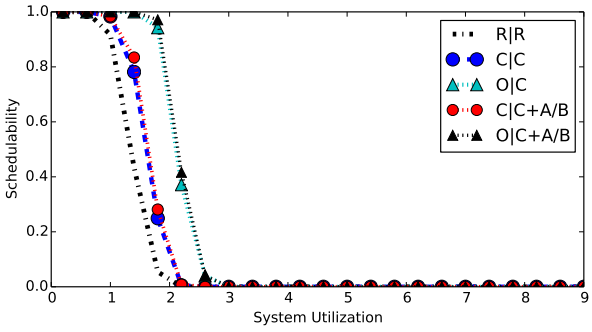
C-Light, Mod., Med., Light, Large, Med., Many



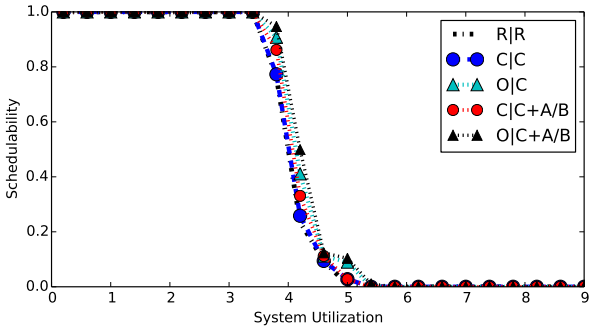
C-Light, Mod., Heavy, Light, Large, High, Few



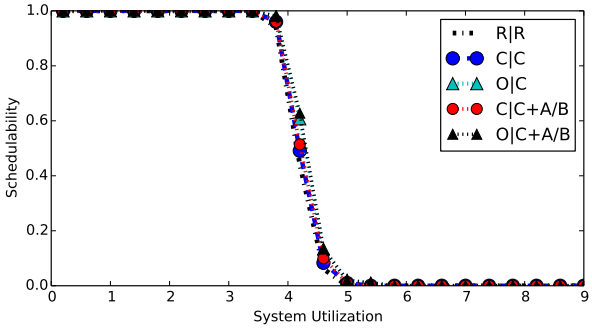
All-Mod., Short, Heavy, Light, Large, High, Many



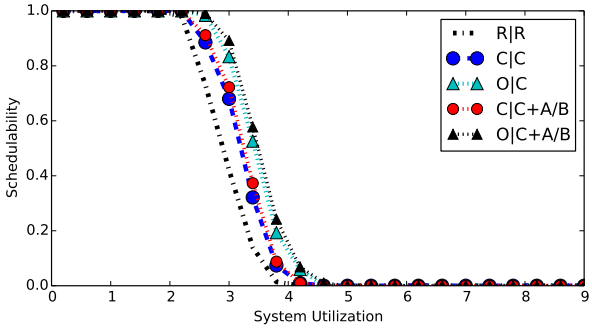
All-Mod., Short, Light, Light, Large, Low, Many



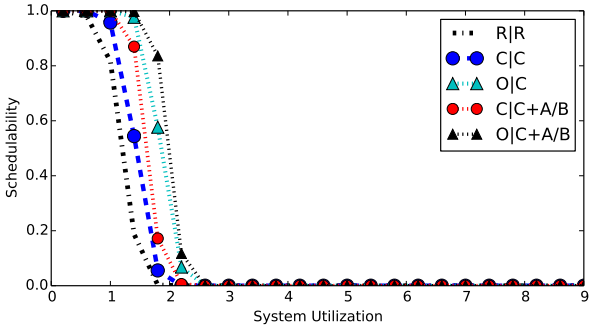
C-Heavy, Short, Med., Heavy, Small, Med., Many



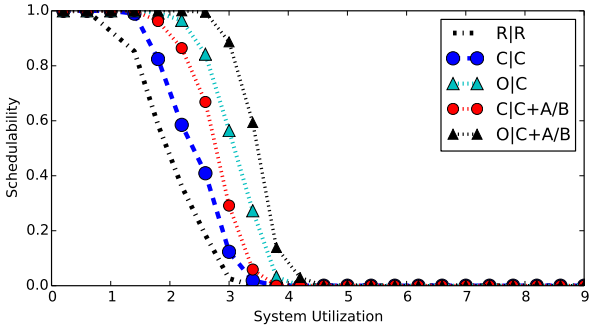
All-Mod., Long, Heavy, Light, Small, Low, Many



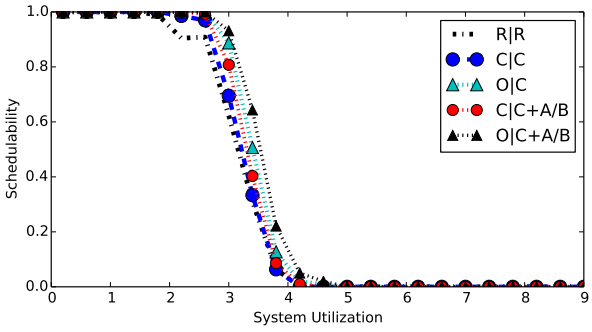
C-Heavy, Long, Light, Light, Large, High, Many



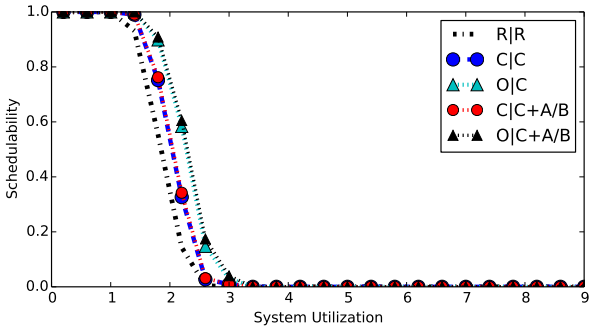
C-Light, Mod., Light, Light, Small, High, Many



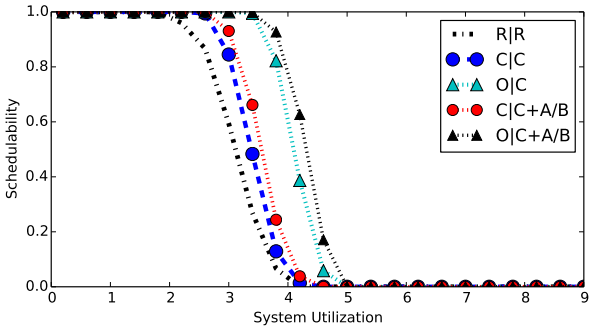
C-Light, Mod., Med., Light, Large, High, Many



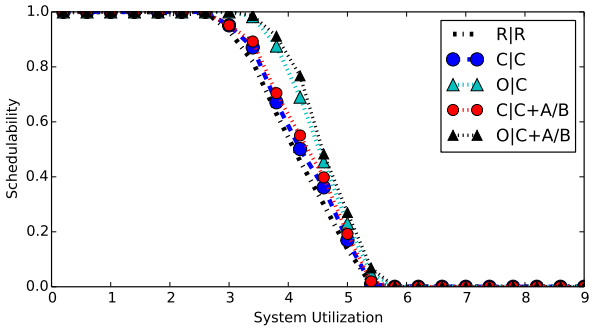
C-Heavy, Mod., Med., Heavy, Large, Low, Many



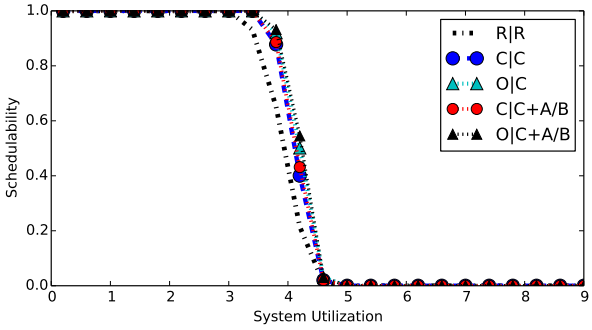
C-Heavy, Mod., Light, Light, Small, Low, Few



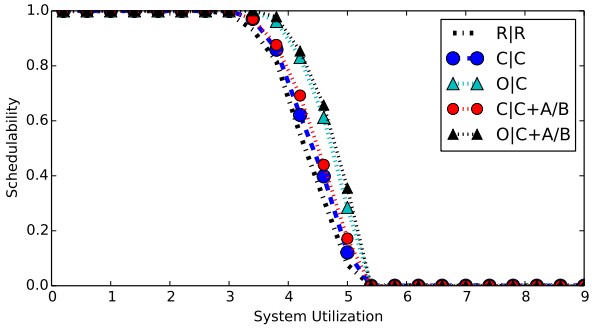
All-Mod., Mod., Med., Heavy, Small, Low, Few



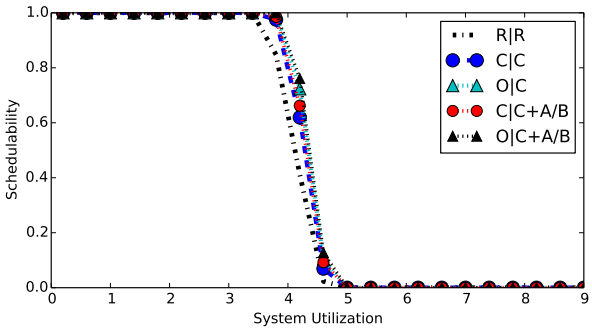
C-Heavy, Short, Heavy, Heavy, Small, Med., Many



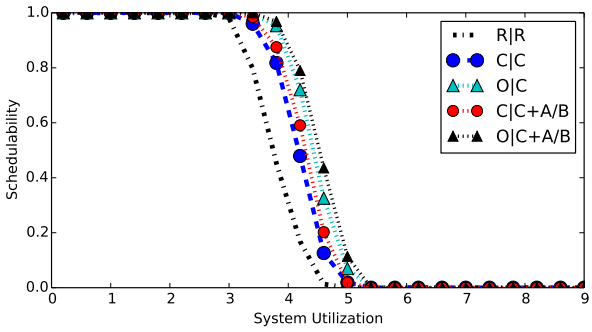
C-Light, Long, Heavy, Heavy, Large, Low, Many



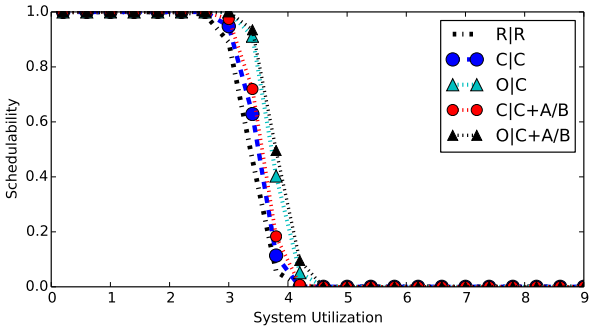
C-Heavy, Short, Heavy, Light, Small, High, Few



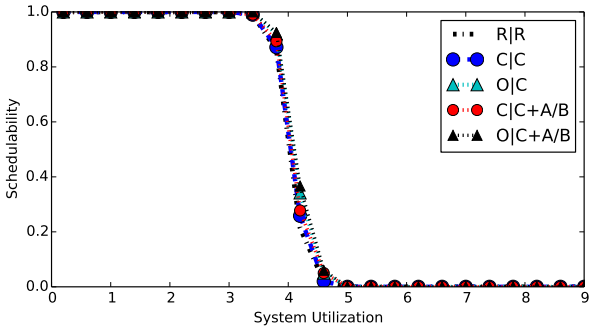
C-Light, Long, Heavy, Heavy, Small, High, Many



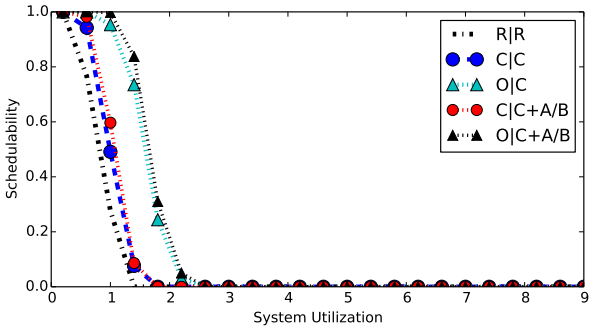
C-Light, Long, Med., Heavy, Small, High, Many



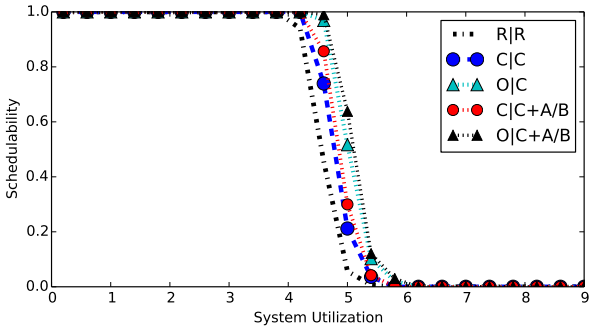
C-Light, Short, Heavy, Heavy, Small, High, Few



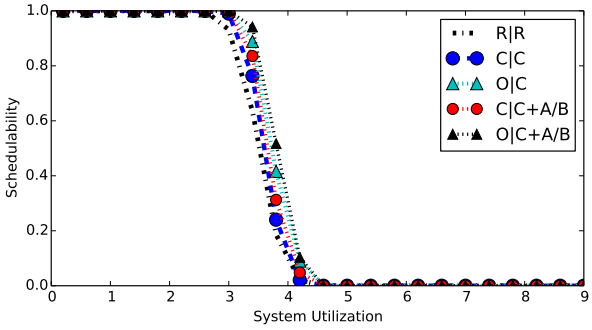
All-Mod., Long, Heavy, Light, Large, Low, Many



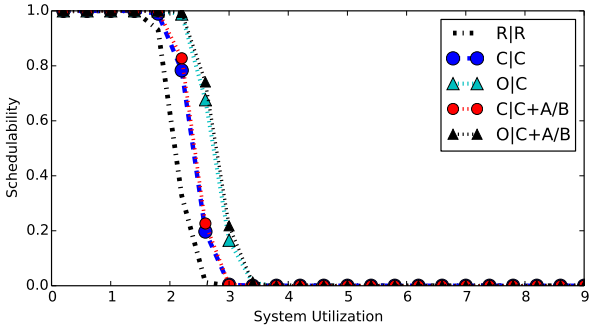
All-Mod., Mod., Light, Heavy, Large, Low, Few



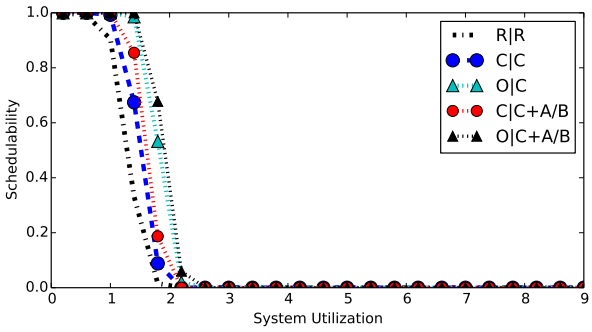
C-Light, Long, Med., Light, Small, Med., Few



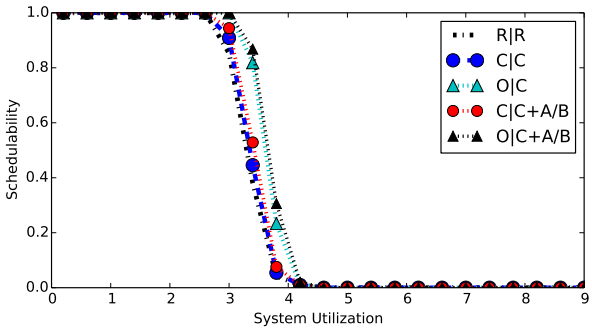
C-Light, Short, Heavy, Light, Large, Med., Many



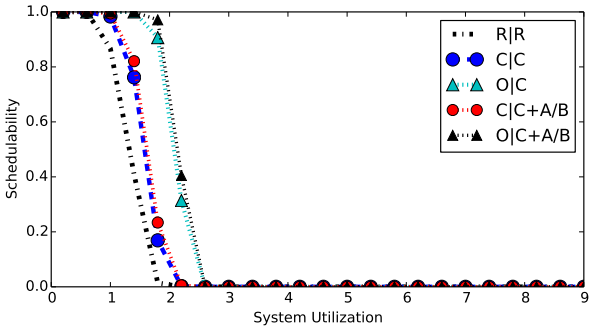
All-Mod., Long, Light, Heavy, Large, Med., Few



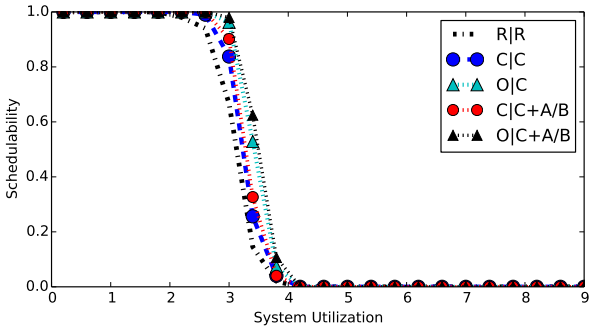
All-Mod., Mod., Light, Heavy, Small, High, Many



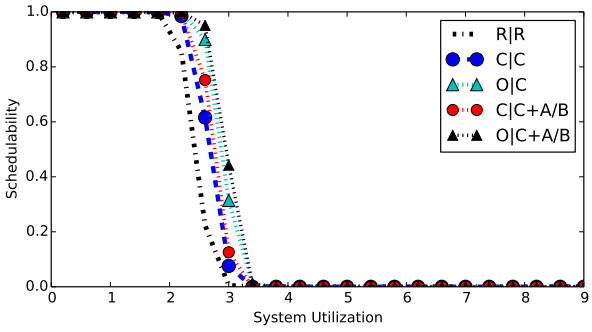
All-Mod., Short, Heavy, Heavy, Small, Med., Few



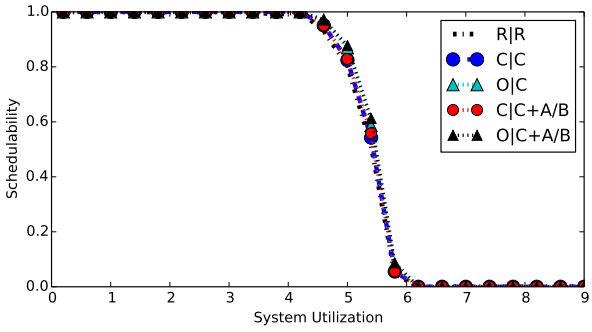
All-Mod., Short, Light, Light, Large, Med., Few



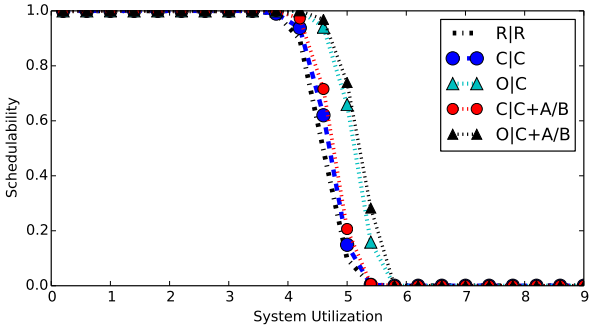
C-Light, Short, Heavy, Heavy, Large, Med., Many



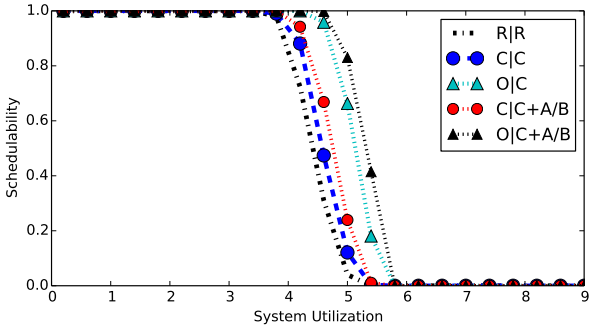
All-Mod., Long, Light, Heavy, Small, High, Few



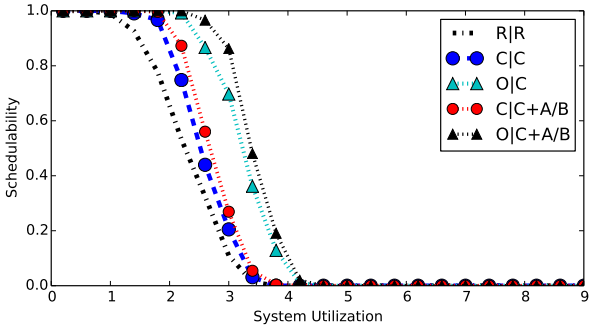
C-Heavy, Long, Heavy, Light, Large, Low, Many



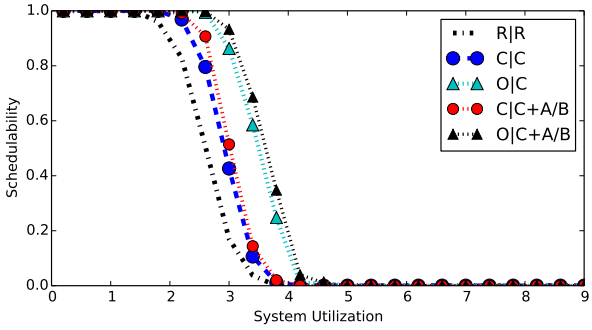
All-Mod., Long, Med., Light, Large, Med., Many



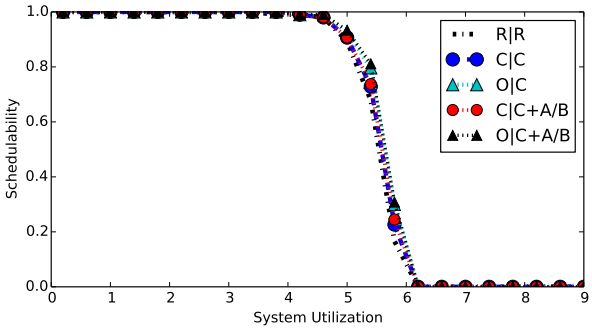
All-Mod., Short, Med., Light, Small, Med., Few



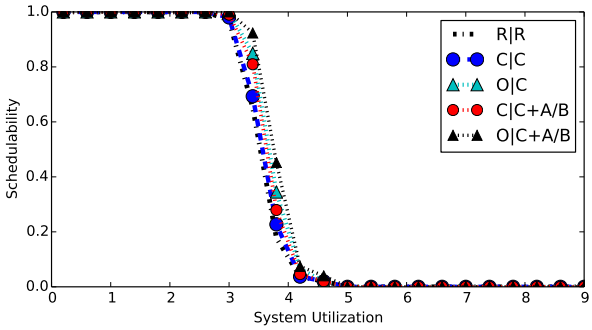
All-Mod., Mod., Med., Heavy, Large, Low, Few



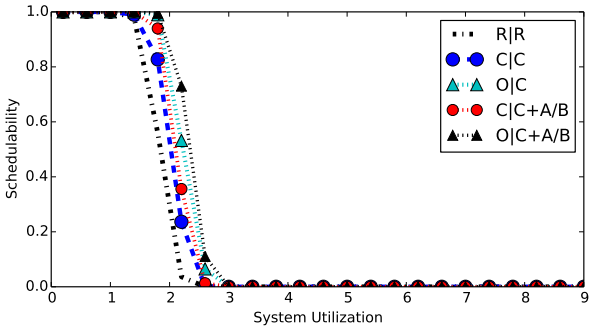
C-Light, Short, Med., Heavy, Large, Low, Few



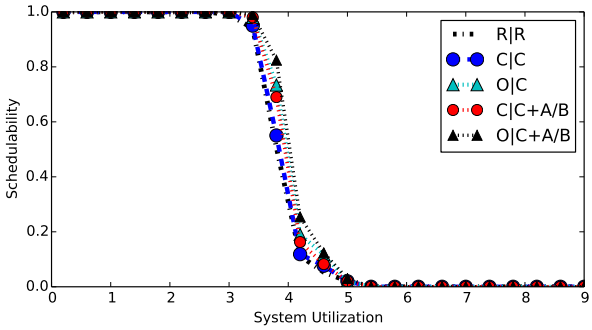
C-Heavy, Long, Heavy, Light, Small, Low, Few



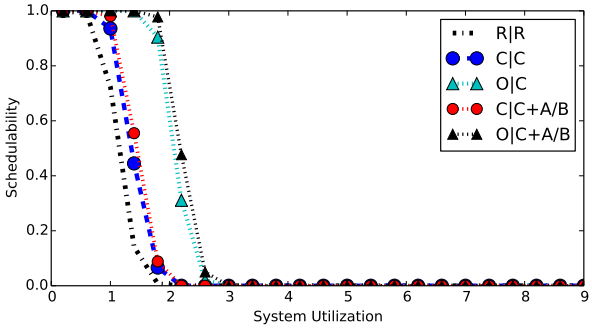
C-Heavy, Short, Med., Heavy, Large, High, Many



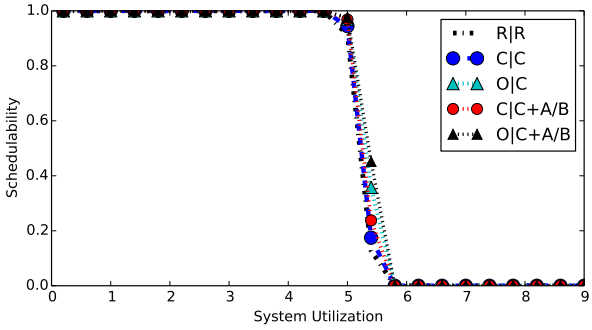
All-Mod., Short, Light, Light, Small, High, Few



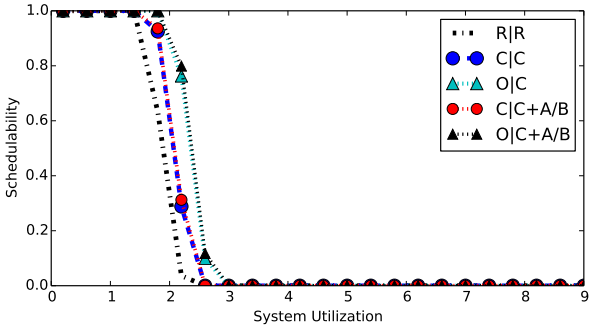
C-Heavy, Short, Med., Heavy, Small, High, Few



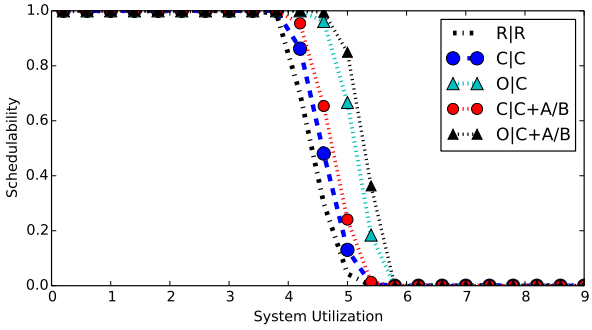
C-Light, Short, Light, Light, Large, Med., Few



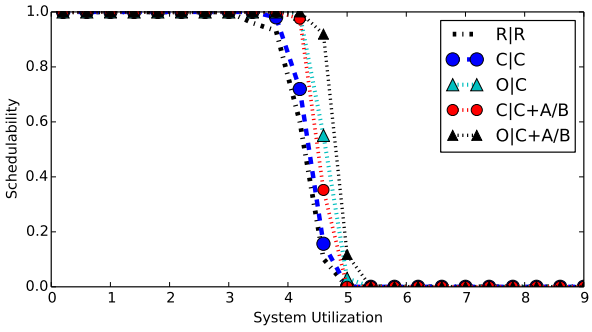
C-Heavy, Long, Med., Light, Small, High, Few



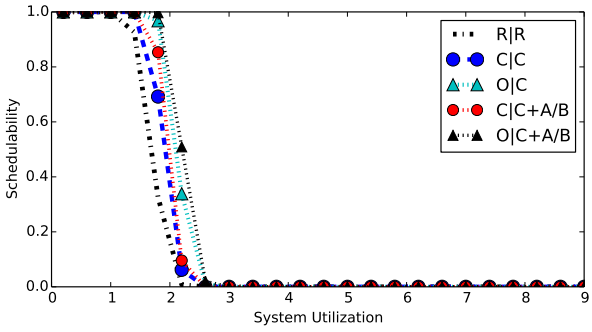
C-Light, Short, Light, Light, Small, Low, Many



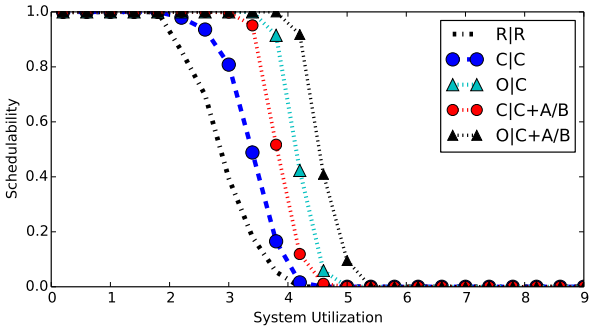
All-Mod., Short, Med., Light, Small, Med., Many



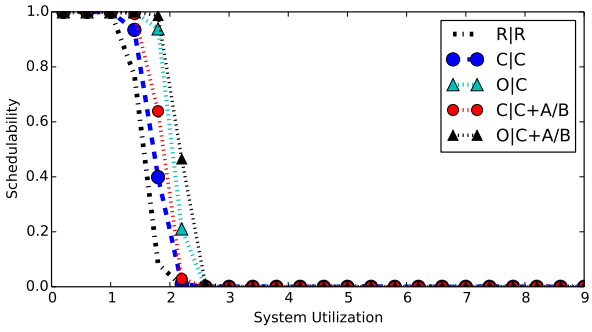
C-Heavy, Mod., Med., Light, Small, High, Few



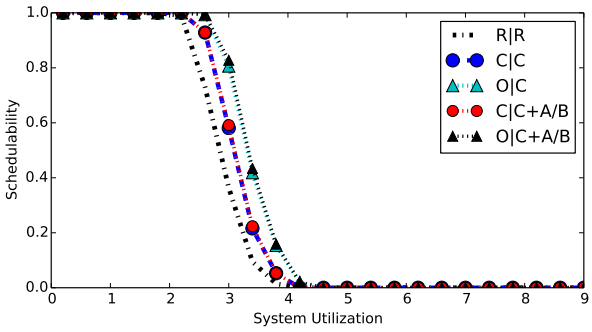
All-Mod., Short, Light, Heavy, Small, High, Many



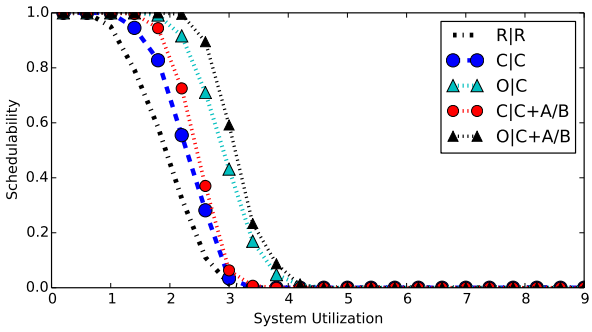
C-Light, Mod., Med., Light, Small, High, Few



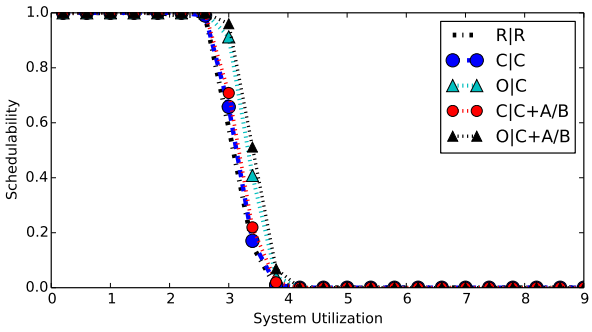
C-Light, Short, Light, Heavy, Small, High, Many



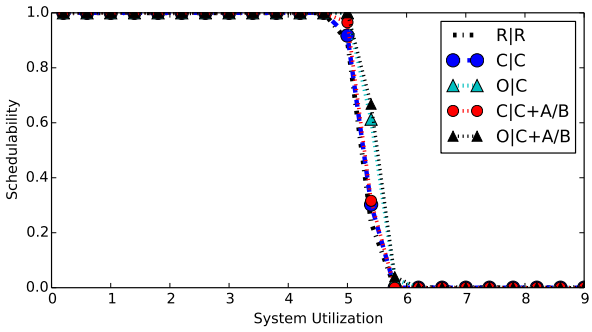
C-Heavy, Long, Light, Heavy, Large, Low, Many



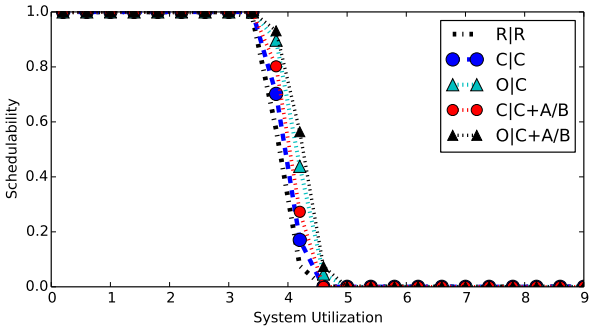
C-Light, Mod., Med., Heavy, Large, Low, Few



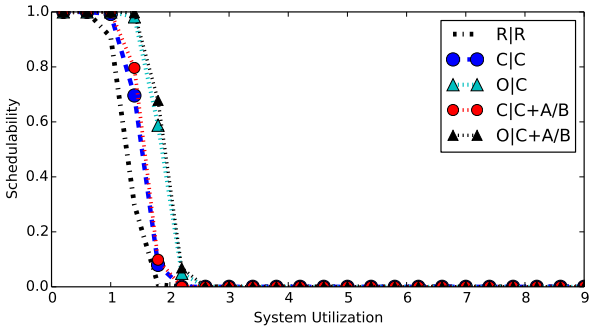
All-Mod., Short, Heavy, Heavy, Large, Med., Few



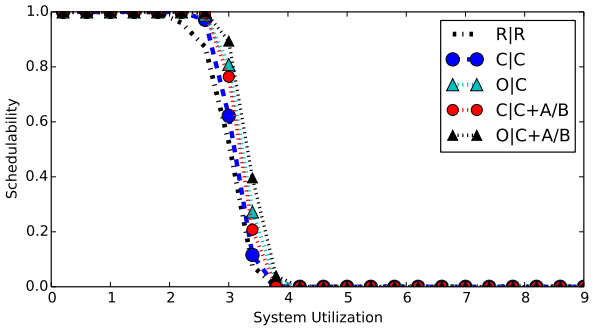
C-Heavy, Long, Med., Heavy, Large, Low, Few



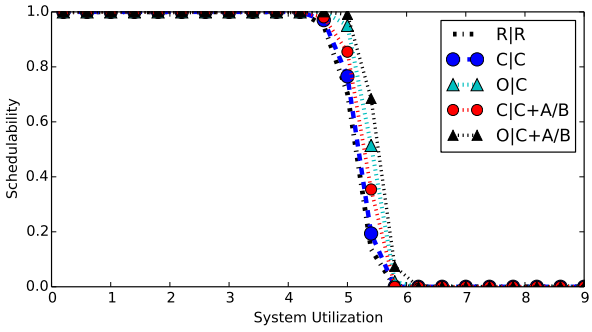
C-Light, Short, Heavy, Light, Small, Med., Few



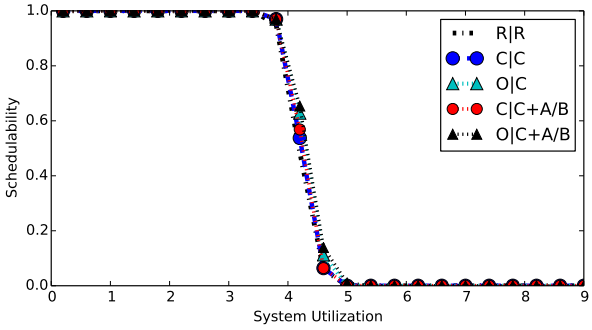
C-Light, Mod., Light, Heavy, Small, Low, Many



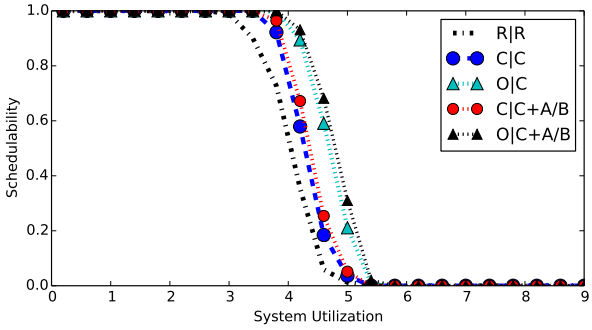
All-Mod., Mod., Heavy, Light, Large, High, Few



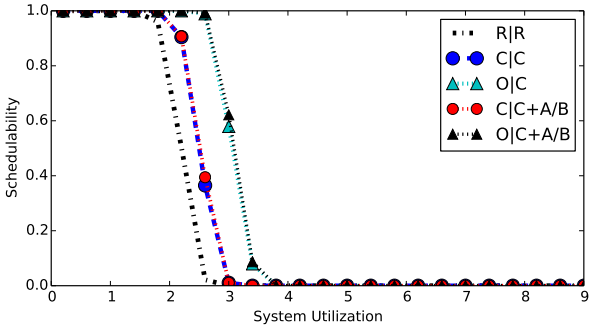
All-Mod., Long, Med., Light, Small, High, Many



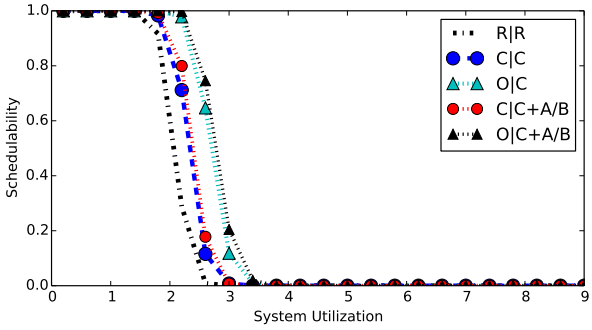
All-Mod., Long, Heavy, Heavy, Small, High, Many



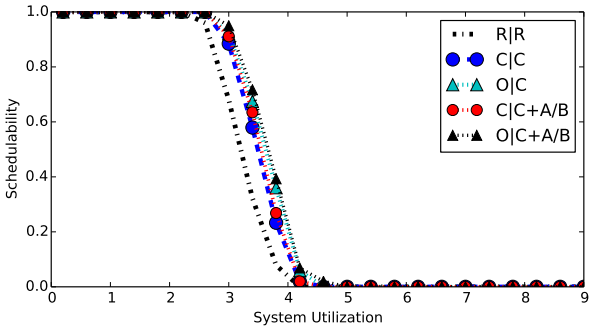
All-Mod., Long, Med., Heavy, Large, High, Many



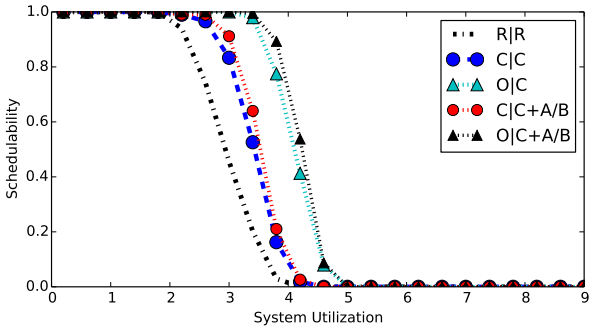
C-Light, Long, Light, Light, Large, Low, Many



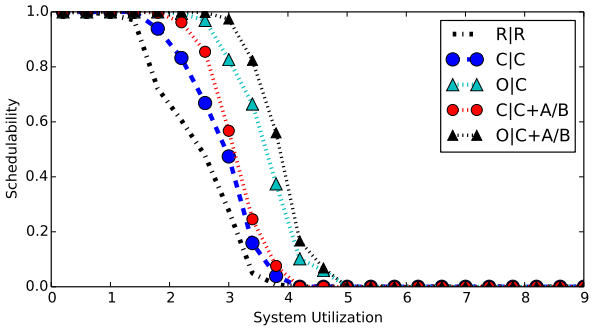
All-Mod., Long, Light, Heavy, Large, High, Few



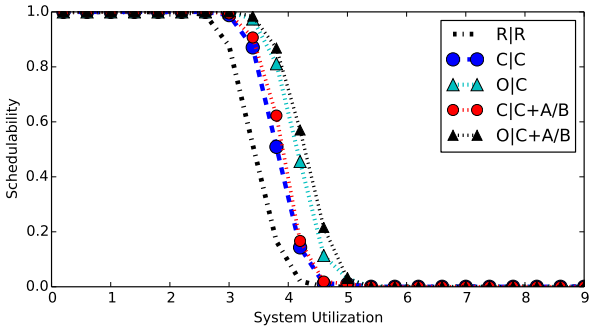
C-Heavy, Long, Light, Light, Small, High, Many



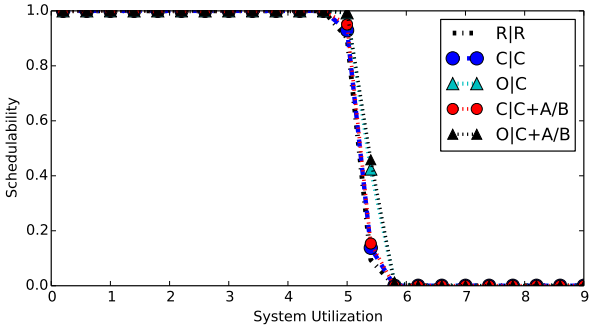
C-Light, Short, Med., Light, Large, Low, Many



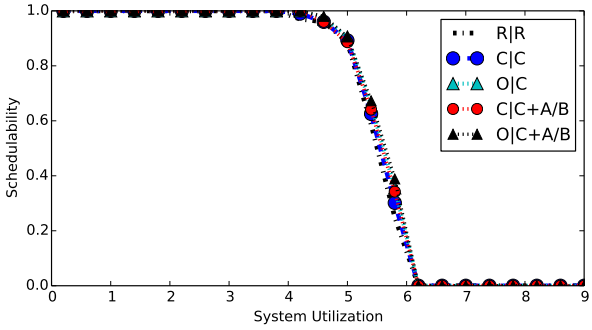
All-Mod., Mod., Med., Light, Large, Low, Many



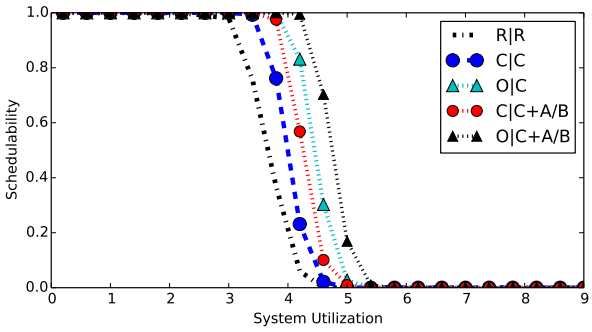
C-Light, Long, Med., Heavy, Large, High, Many



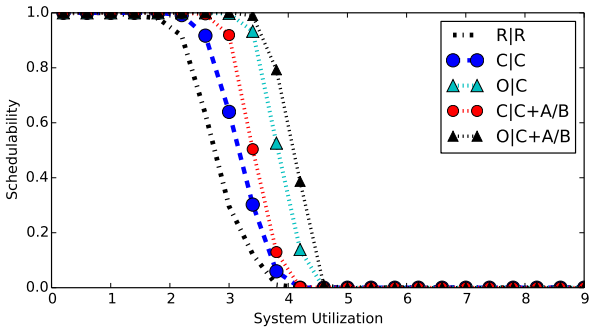
C-Heavy, Long, Med., Light, Large, Low, Few



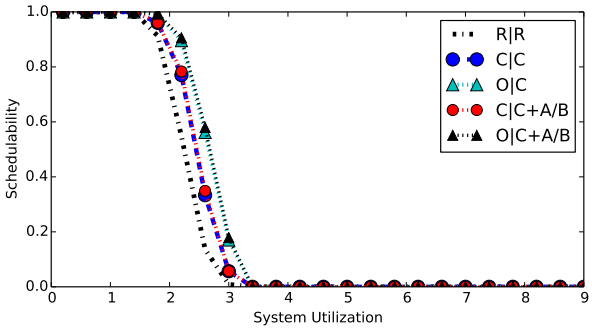
C-Heavy, Long, Heavy, Heavy, Large, Med., Many



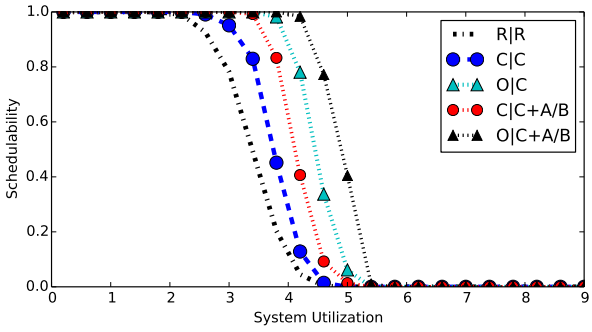
C-Light, Short, Med., Light, Small, High, Few



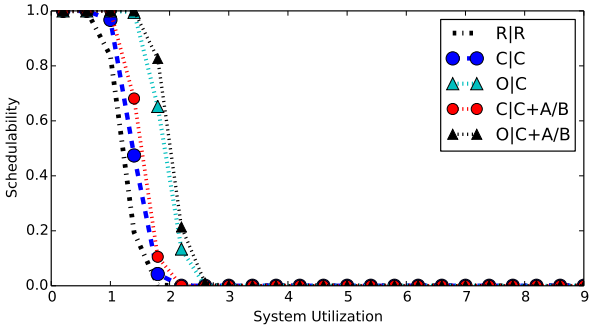
C-Light, Short, Med., Light, Large, High, Few



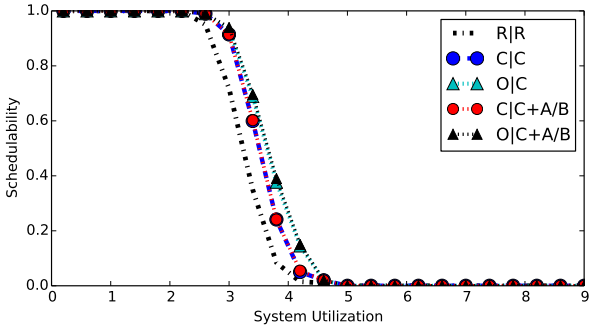
C-Heavy, Short, Light, Heavy, Small, Low, Many



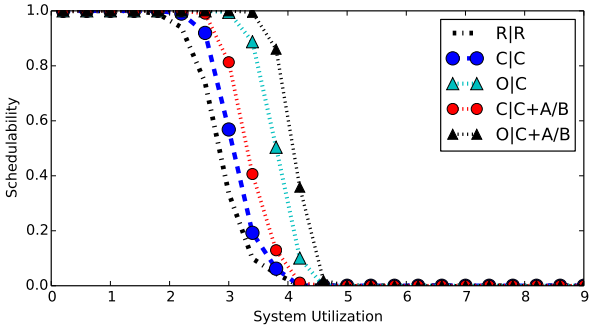
All-Mod., Mod., Med., Light, Small, High, Many



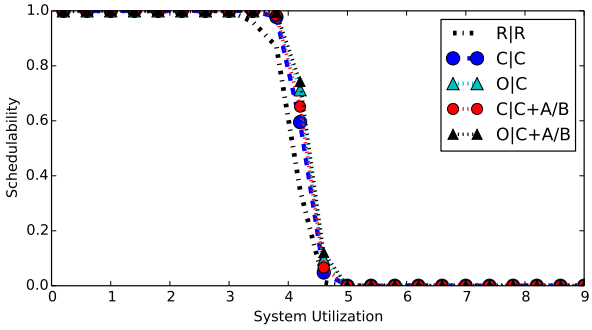
All-Mod., Short, Light, Heavy, Large, High, Many



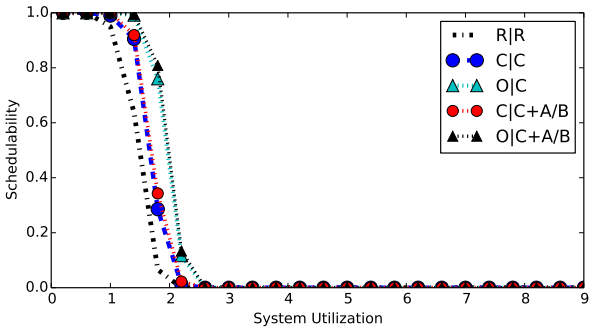
C-Heavy, Long, Light, Light, Small, Low, Few



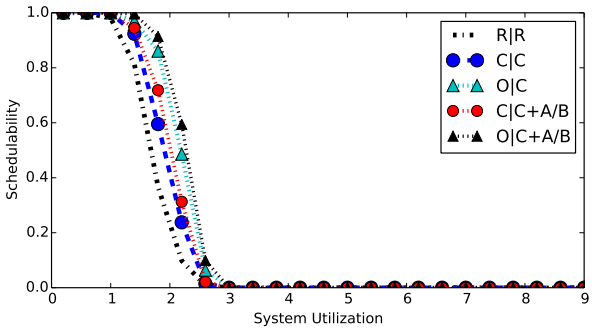
All-Mod., Mod., Med., Heavy, Small, High, Many



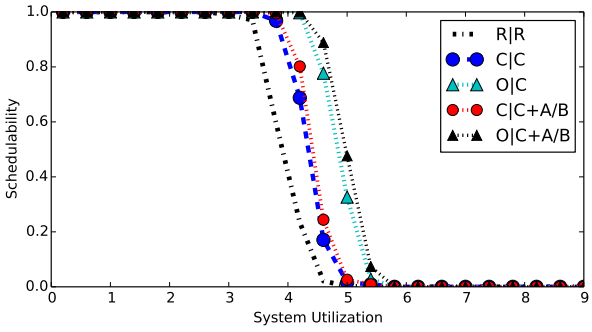
C-Light, Long, Heavy, Heavy, Small, Med., Few



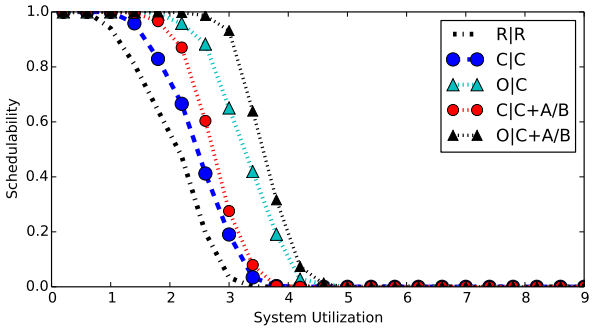
All-Mod., Mod., Light, Light, Small, Med., Many



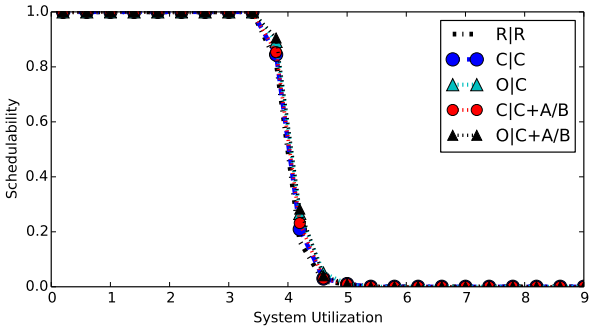
C-Heavy, Mod., Light, Heavy, Small, High, Many



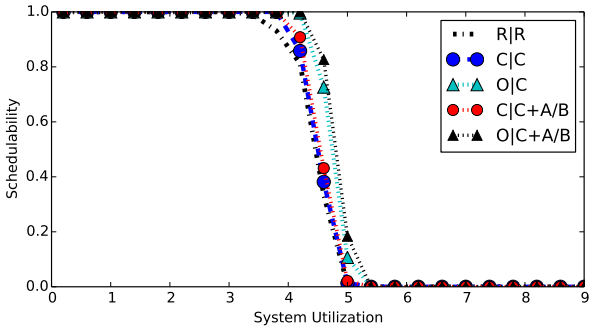
C-Light, Short, Med., Light, Small, Low, Few



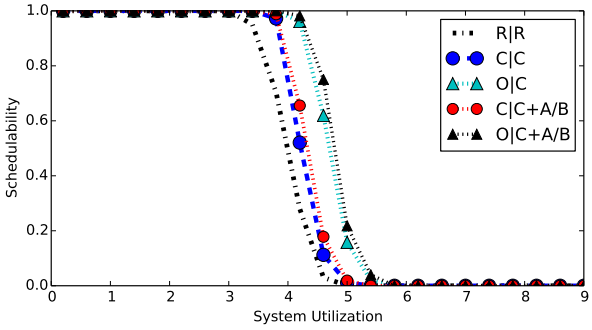
C-Light, Mod., Med., Light, Large, Low, Few



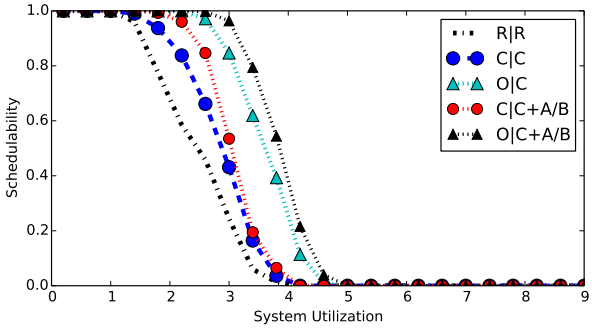
All-Mod., Long, Heavy, Light, Large, High, Few



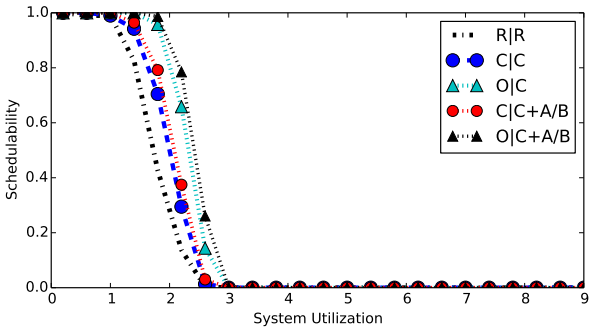
C-Heavy, Short, Med., Light, Large, Low, Many



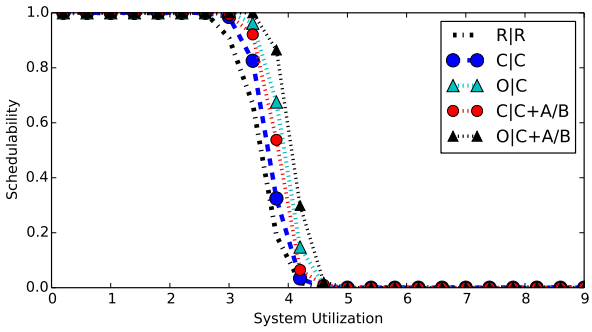
C-Light, Long, Med., Light, Large, High, Many



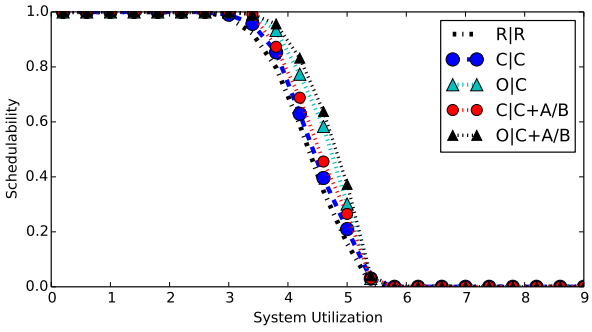
All-Mod., Mod., Med., Light, Large, Med., Many



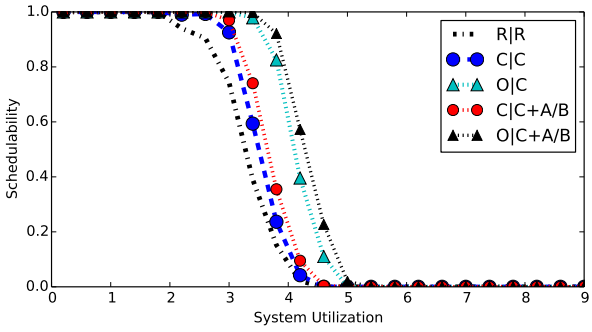
C-Heavy, Short, Light, Heavy, Large, High, Few



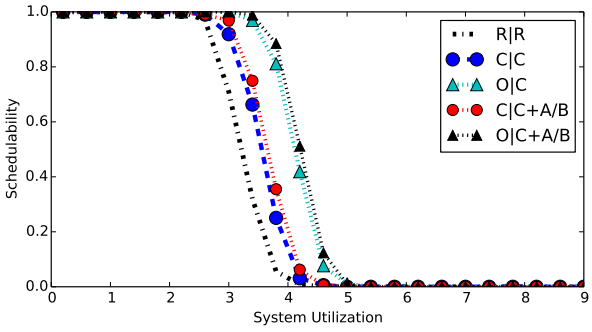
C-Light, Mod., Heavy, Light, Small, High, Few



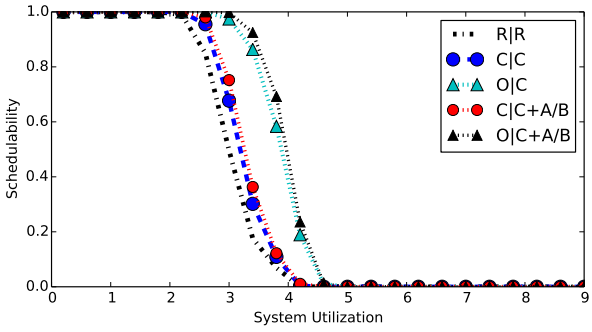
C-Heavy, Short, Heavy, Light, Large, Low, Many



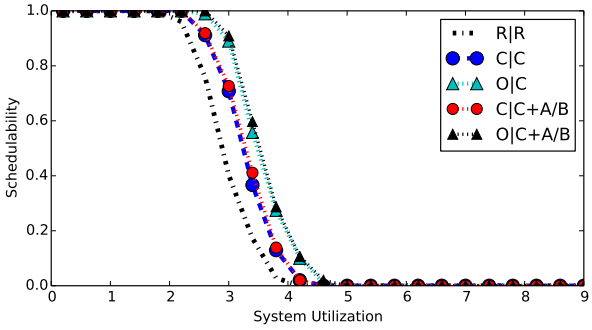
All-Mod., Short, Med., Light, Large, High, Many



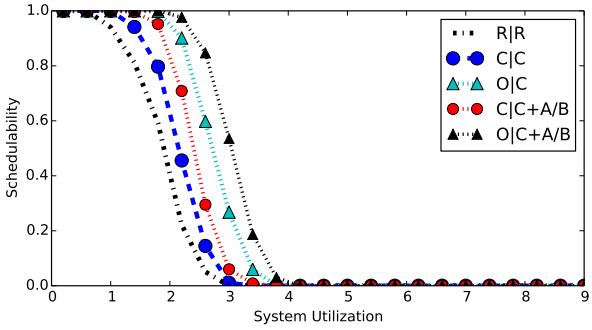
C-Light, Short, Med., Heavy, Small, Low, Few



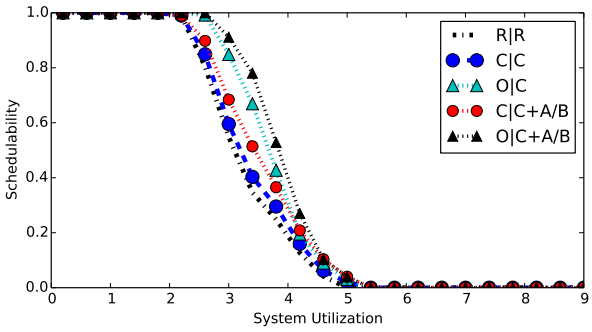
All-Mod., Short, Med., Heavy, Large, Low, Few



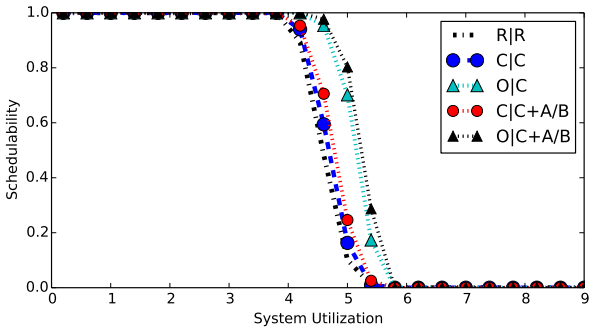
C-Heavy, Long, Light, Light, Large, Med., Few



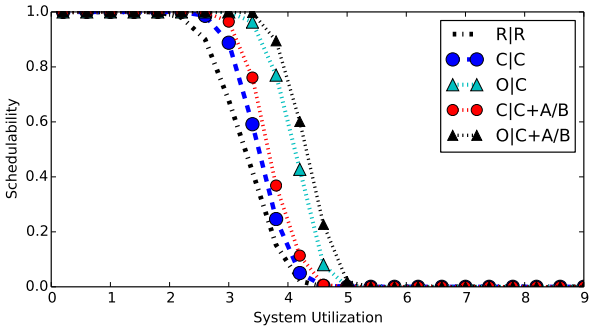
C-Light, Mod., Med., Heavy, Large, High, Many



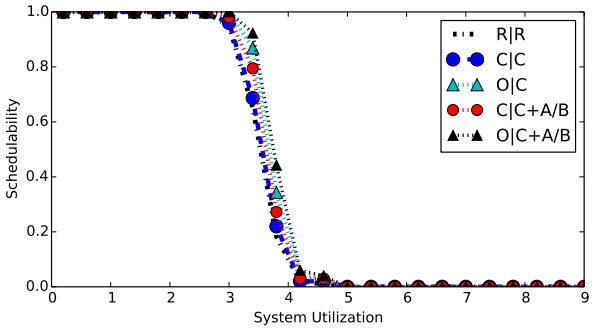
C-Heavy, Mod., Heavy, Heavy, Large, Med., Few



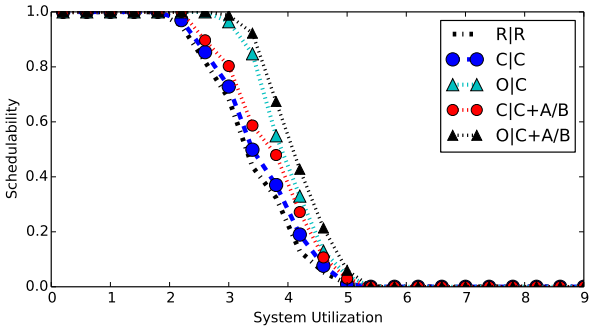
All-Mod., Long, Med., Light, Large, Med., Few



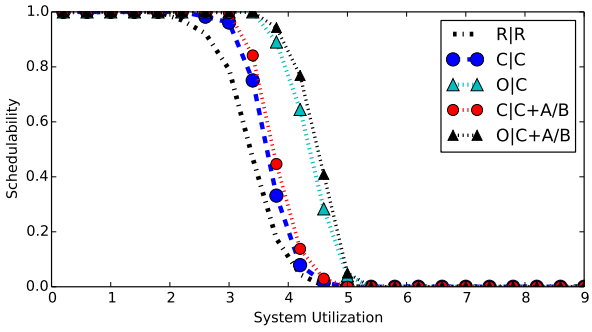
All-Mod., Short, Med., Light, Large, High, Few



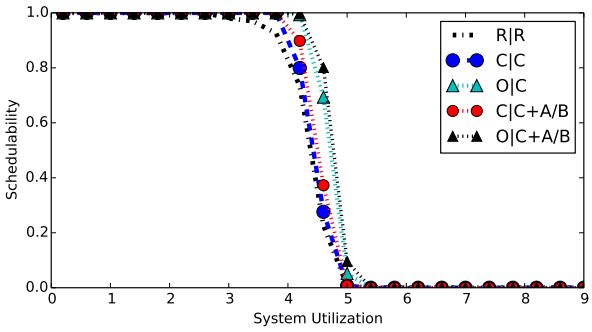
C-Heavy, Short, Med., Heavy, Large, High, Few



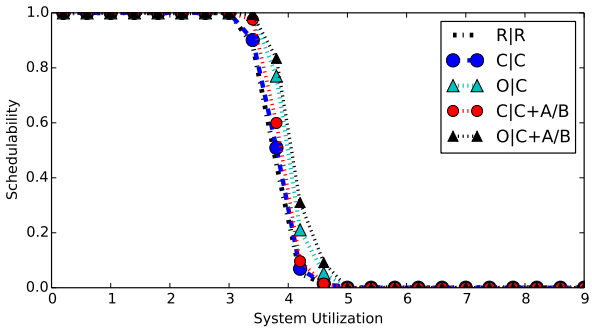
C-Heavy, Mod., Heavy, Heavy, Small, High, Few



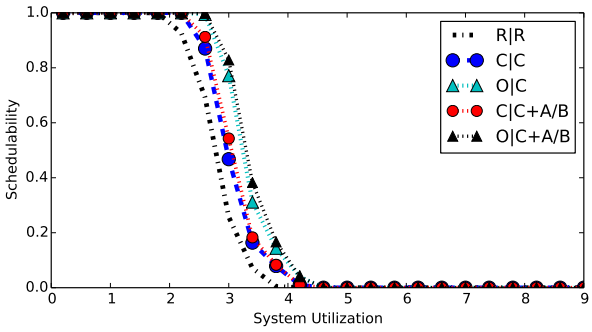
All-Mod., Short, Med., Light, Large, Med., Many



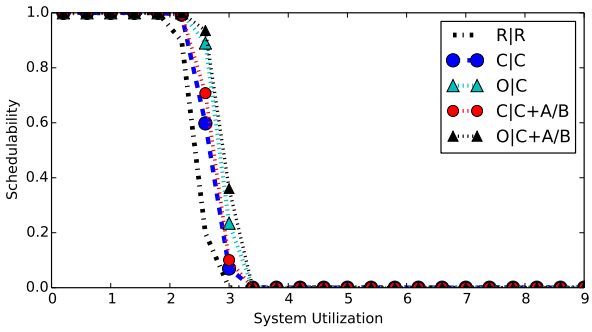
C-Heavy, Short, Med., Light, Large, Med., Many



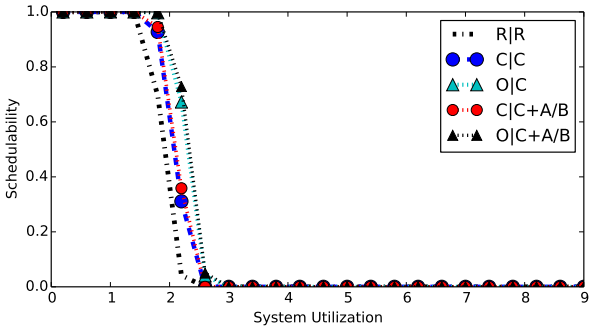
All-Mod., Short, Heavy, Light, Small, Low, Many



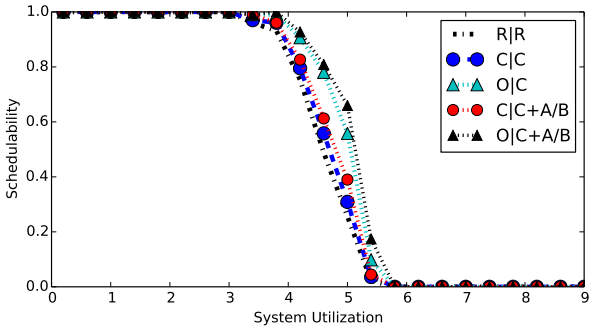
C-Heavy, Long, Light, Heavy, Large, High, Many



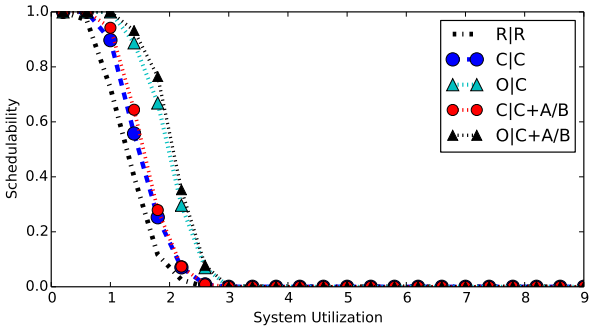
All-Mod., Long, Light, Heavy, Small, High, Many



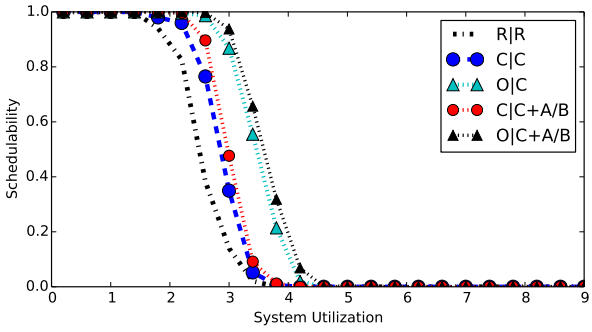
All-Mod., Short, Light, Light, Small, Med., Few



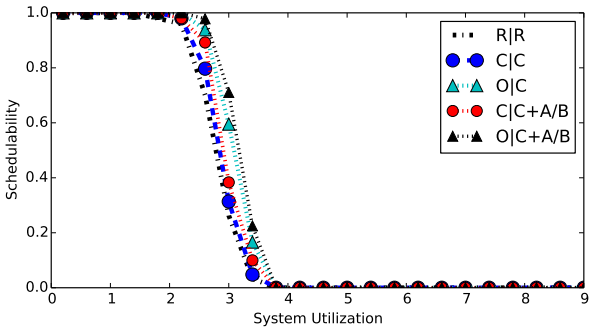
C-Heavy, Short, Heavy, Light, Small, Med., Few



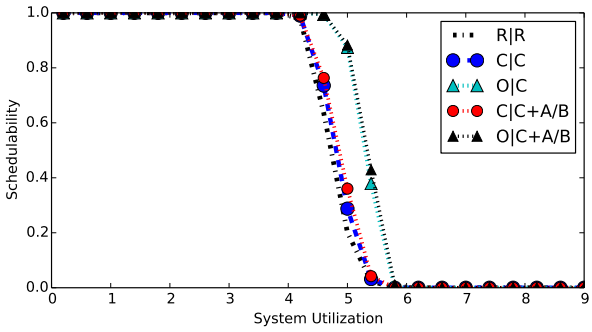
C-Heavy, Mod., Light, Light, Large, Low, Many



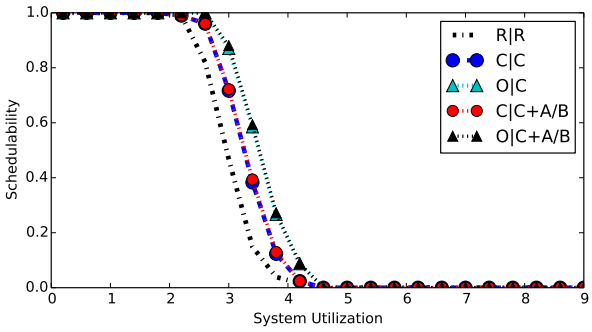
C-Light, Short, Med., Heavy, Large, Med., Few



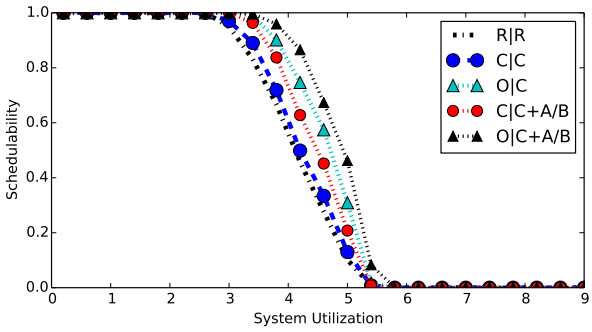
All-Mod., Mod., Heavy, Heavy, Large, High, Few



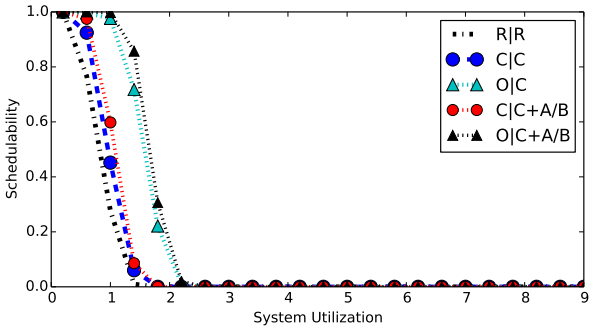
All-Mod., Long, Med., Light, Large, Low, Many



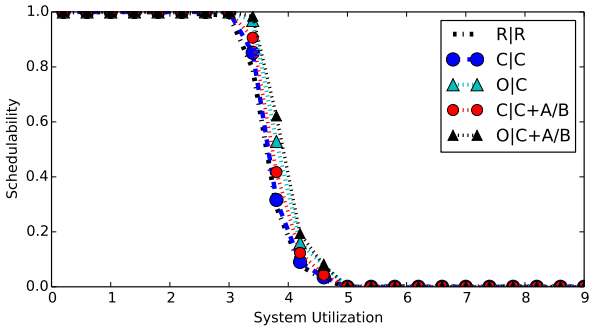
C-Heavy, Long, Light, Light, Large, Low, Few



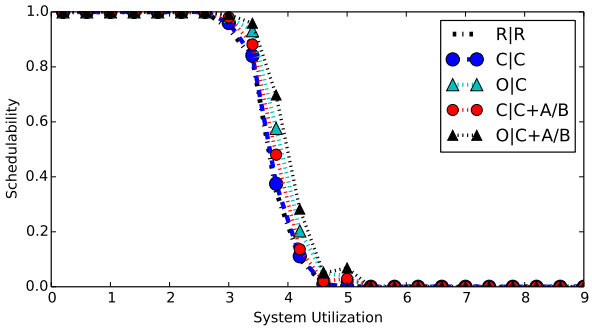
C-Heavy, Mod., Heavy, Light, Small, Low, Many



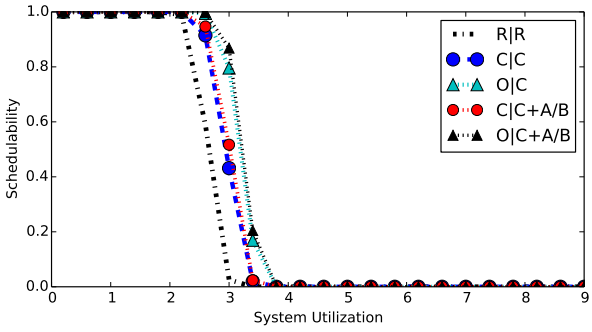
All-Mod., Mod., Light, Heavy, Large, Med., Many



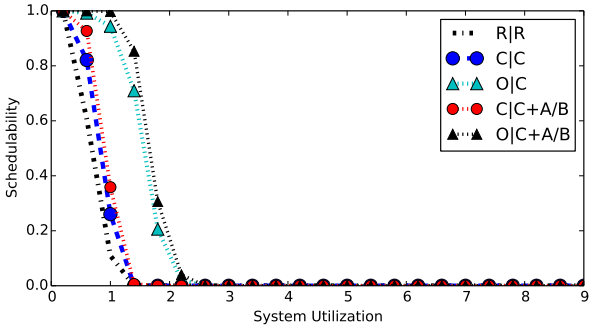
C-Heavy, Short, Med., Heavy, Large, Med., Few



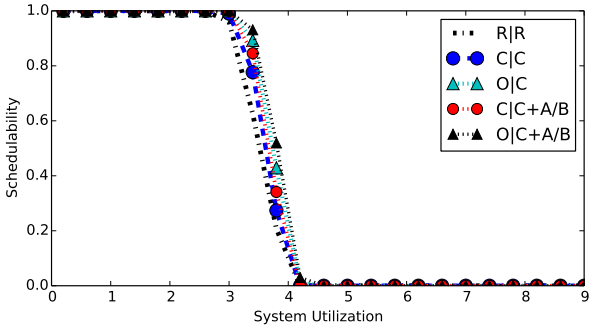
C-Heavy, Mod., Med., Heavy, Small, Low, Many



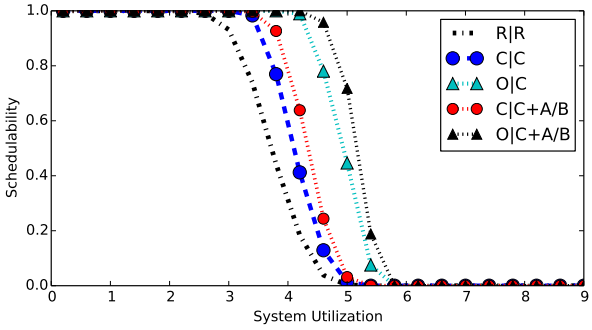
C-Light, Long, Light, Light, Small, Med., Many



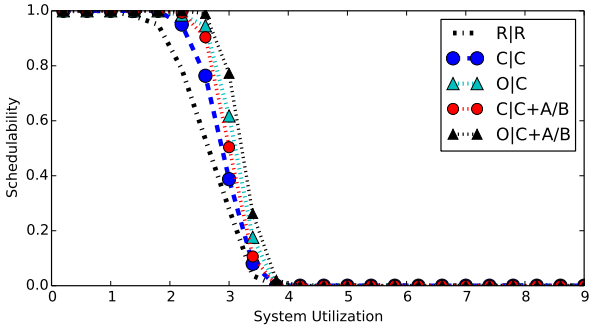
C-Light, Mod., Light, Heavy, Large, Low, Many



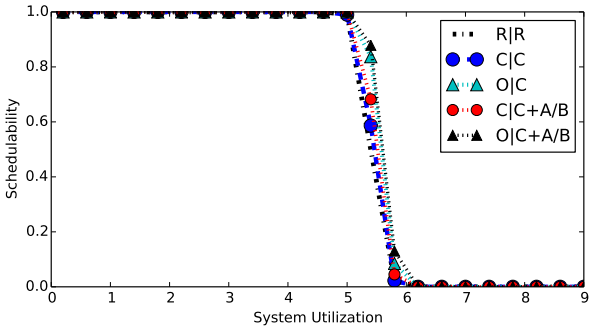
C-Light, Short, Heavy, Light, Large, High, Many



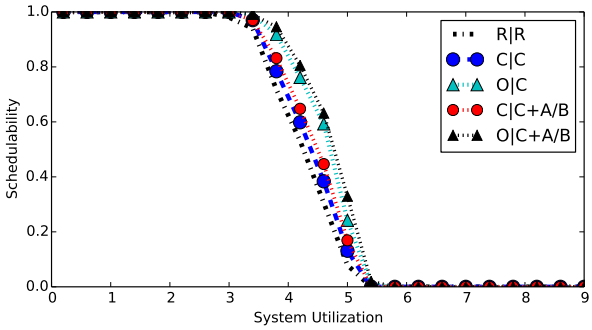
All-Mod., Mod., Med., Light, Small, Low, Few



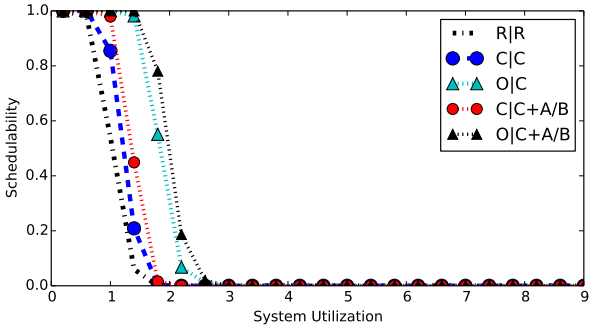
C-Light, Mod., Heavy, Heavy, Large, High, Many



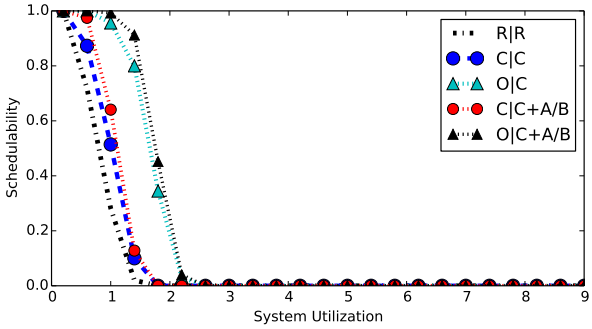
C-Heavy, Long, Med., Heavy, Small, Med., Many



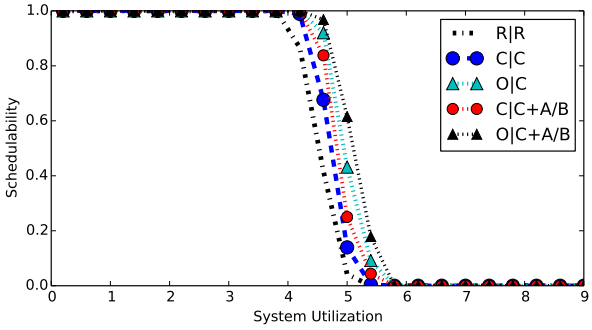
C-Heavy, Short, Heavy, Light, Small, High, Many



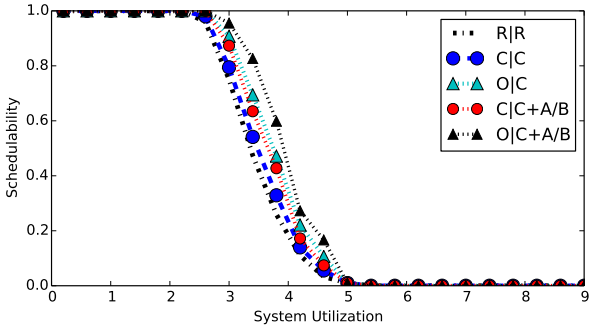
C-Light, Short, Light, Heavy, Large, High, Many



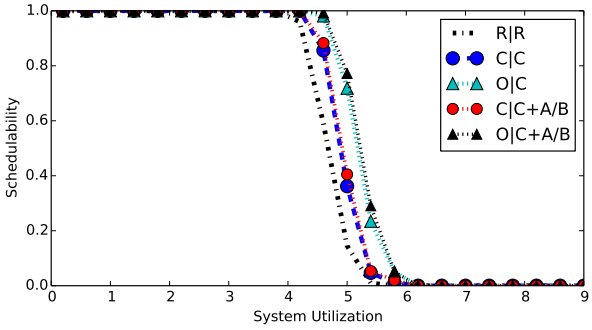
All-Mod., Mod., Light, Light, Large, Med., Many



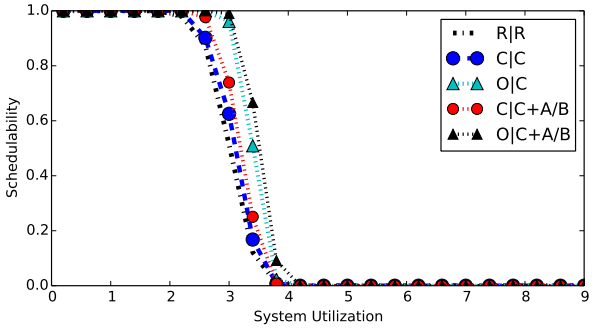
C-Light, Long, Med., Light, Small, High, Few



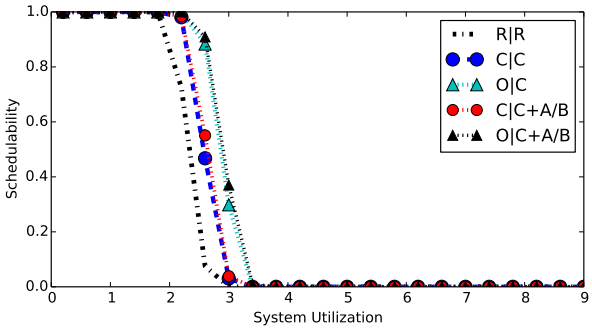
C-Heavy, Mod., Heavy, Light, Large, High, Many



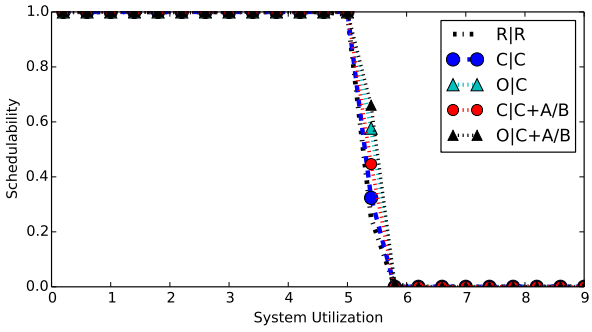
C-Light, Long, Med., Light, Small, Low, Few



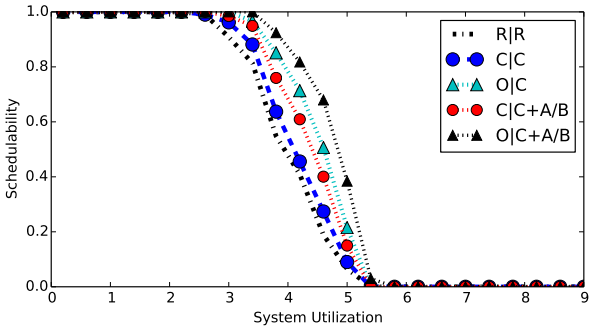
All-Mod., Mod., Heavy, Heavy, Small, High, Few



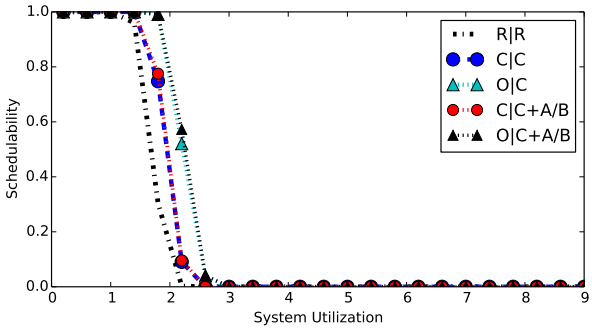
C-Light, Long, Light, Heavy, Small, Med., Many



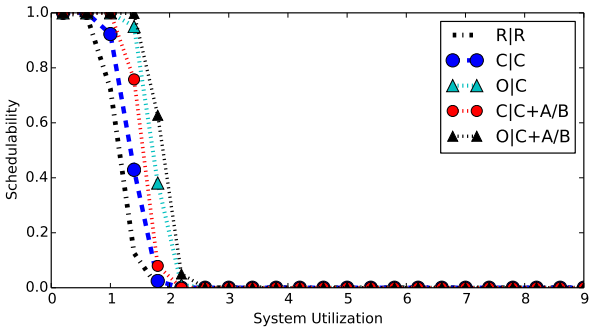
C-Heavy, Long, Med., Light, Small, Med., Few



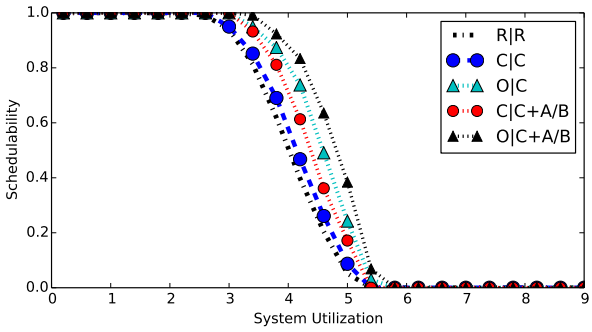
C-Heavy, Mod., Heavy, Light, Small, Med., Few



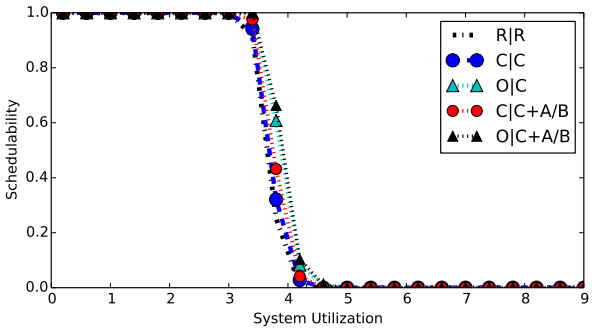
C-Light, Short, Light, Heavy, Small, Low, Many



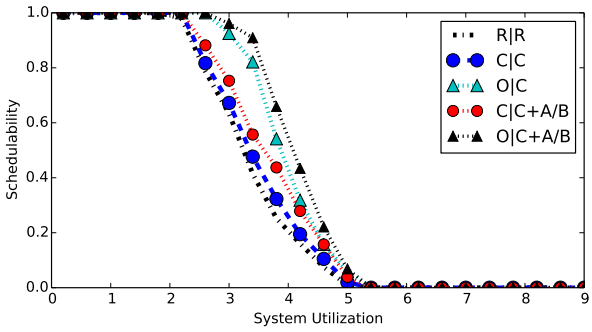
C-Light, Mod., Light, Heavy, Small, High, Many



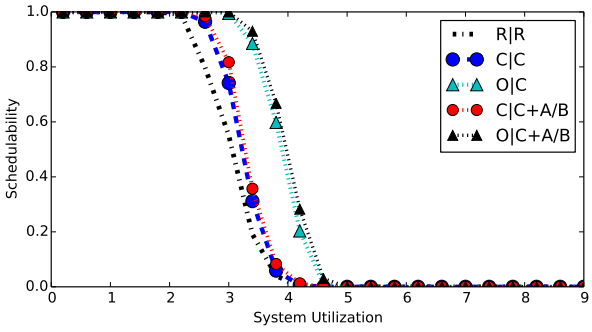
C-Heavy, Mod., Heavy, Light, Small, Med., Many



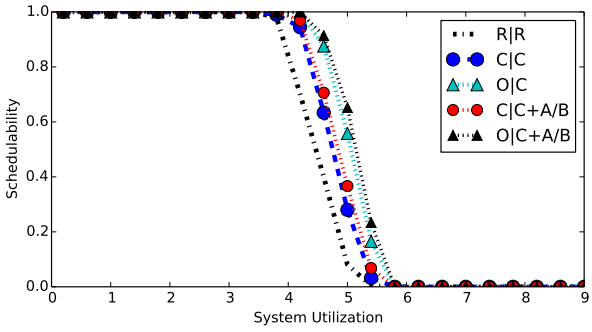
All-Mod., Short, Heavy, Light, Small, High, Few



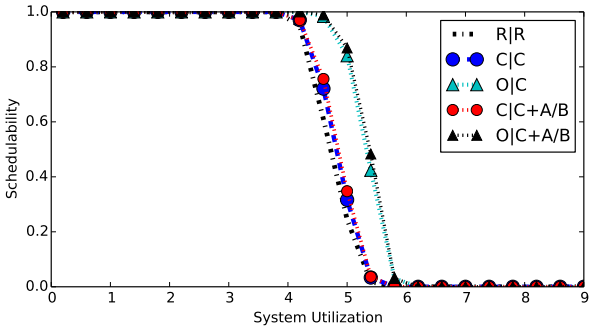
C-Heavy, Mod., Heavy, Heavy, Small, High, Many



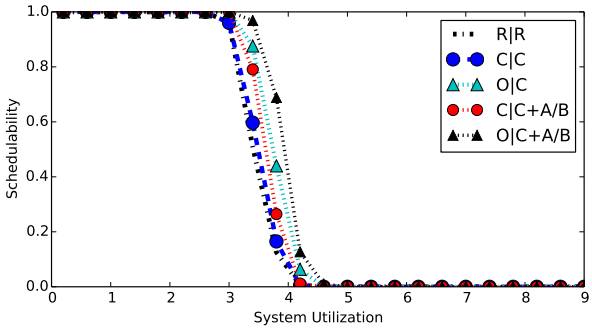
All-Mod., Short, Med., Heavy, Large, Low, Many



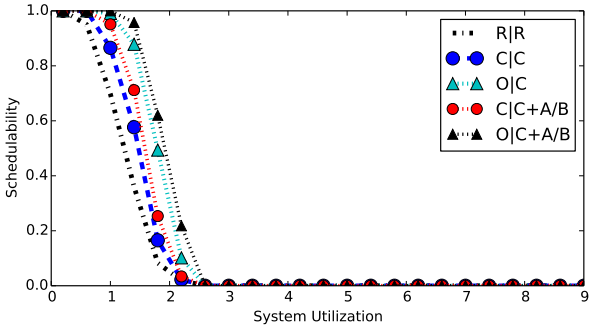
All-Mod., Long, Med., Heavy, Small, Med., Many



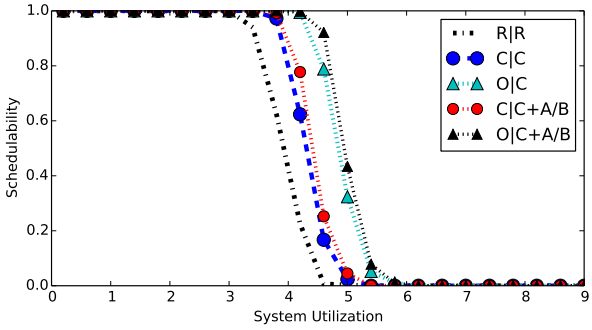
All-Mod., Long, Med., Light, Large, Low, Few



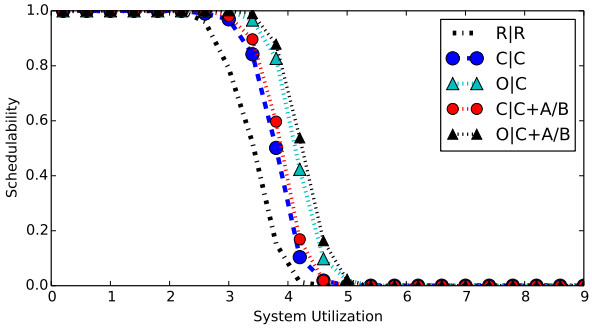
All-Mod., Mod., Heavy, Light, Small, Low, Few



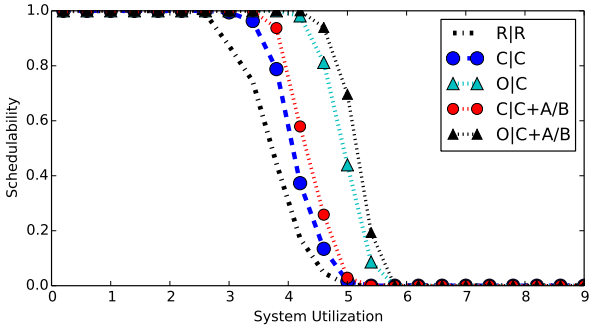
C-Heavy, Mod., Light, Heavy, Large, High, Many



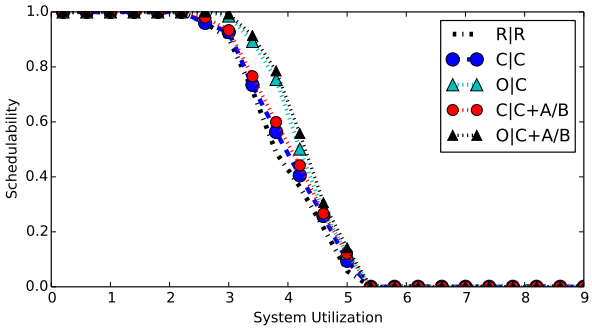
C-Light, Short, Med., Light, Small, Low, Many



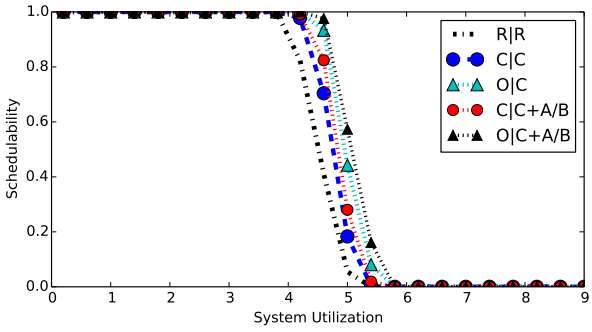
C-Light, Long, Med., Heavy, Large, High, Few



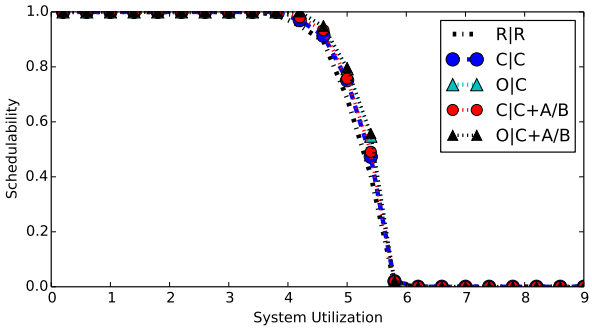
All-Mod., Mod., Med., Light, Small, Low, Many



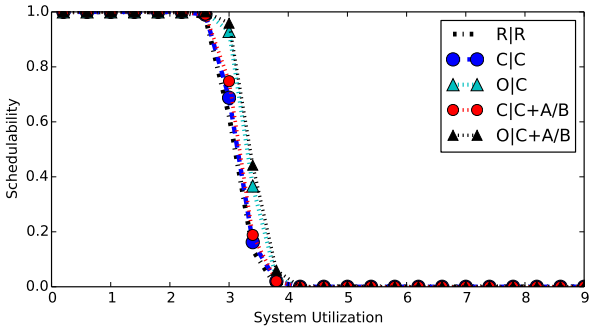
C-Heavy, Short, Heavy, Heavy, Large, Med., Few



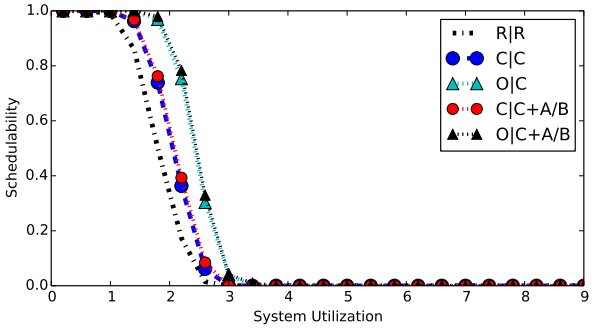
C-Light, Long, Med., Light, Small, High, Many



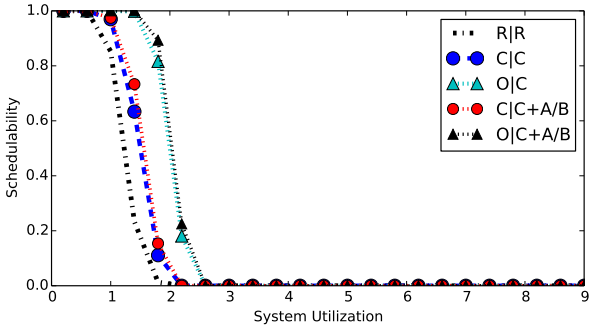
C-Heavy, Long, Heavy, Light, Large, Med., Many



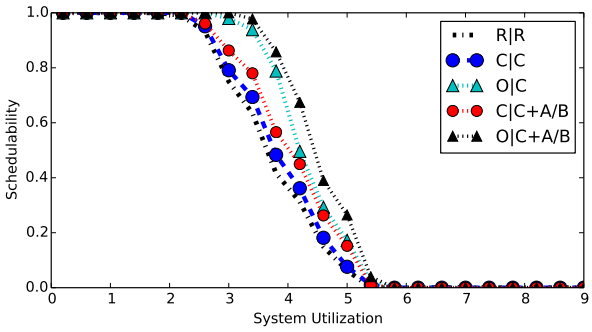
All-Mod., Short, Heavy, Heavy, Large, High, Many



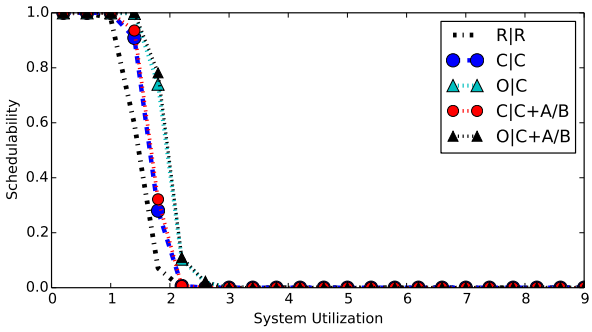
C-Heavy, Short, Light, Heavy, Large, Low, Few



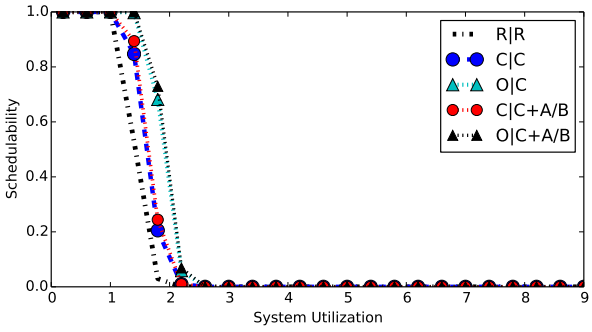
All-Mod., Short, Light, Heavy, Large, Med., Few



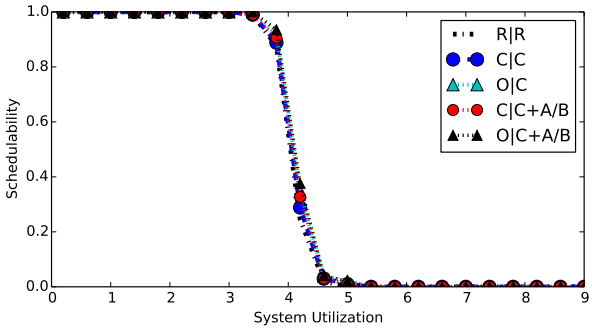
C-Heavy, Mod., Heavy, Heavy, Small, Low, Many



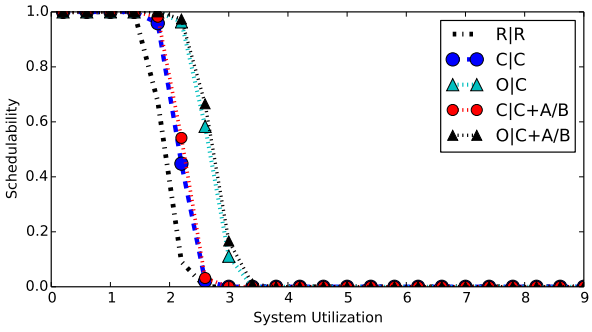
All-Mod., Mod., Light, Light, Small, Low, Few



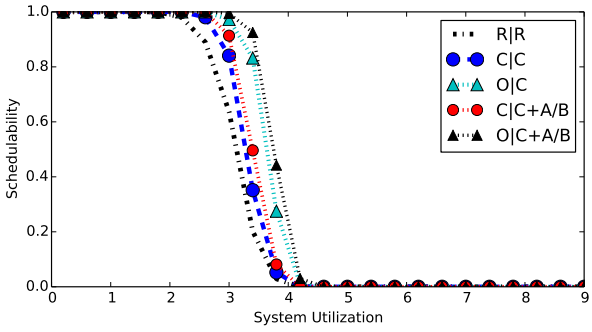
All-Mod., Mod., Light, Heavy, Small, Med., Many



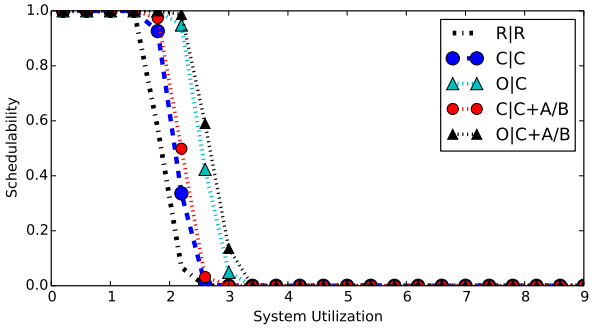
All-Mod., Long, Heavy, Heavy, Large, Med., Many



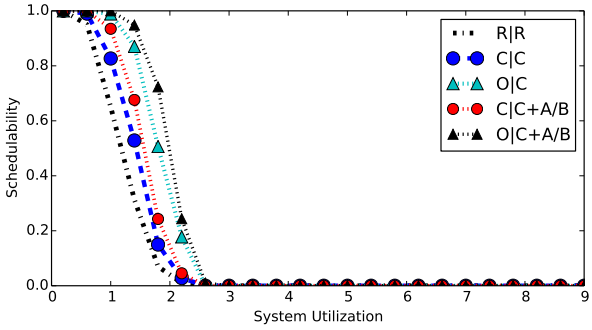
C-Light, Long, Light, Heavy, Large, Med., Many



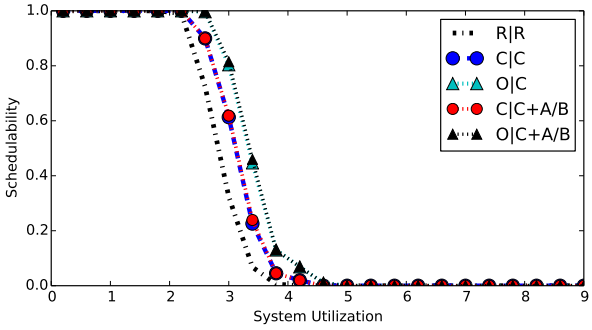
C-Light, Mod., Heavy, Heavy, Small, Low, Many



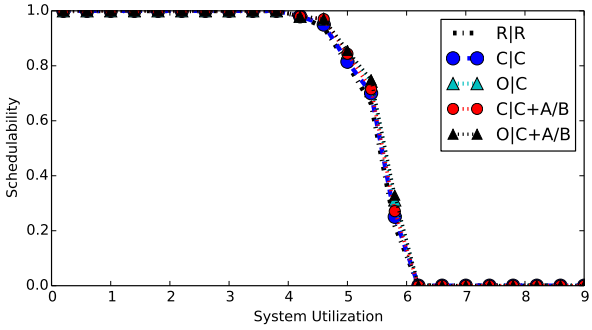
C-Light, Long, Light, Heavy, Large, High, Many



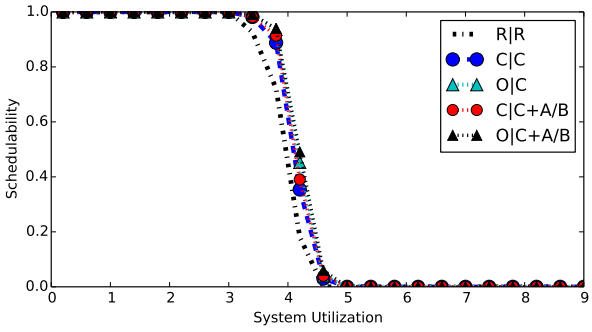
C-Heavy, Mod., Light, Heavy, Large, High, Few



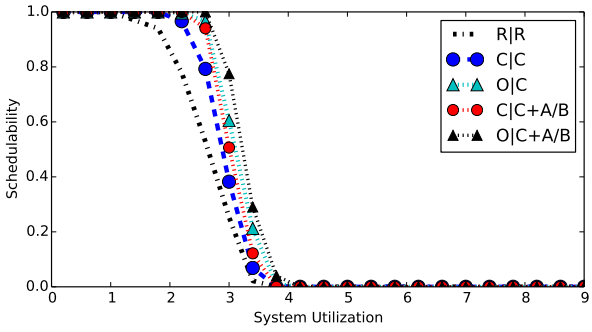
C-Heavy, Long, Light, Heavy, Large, Low, Few



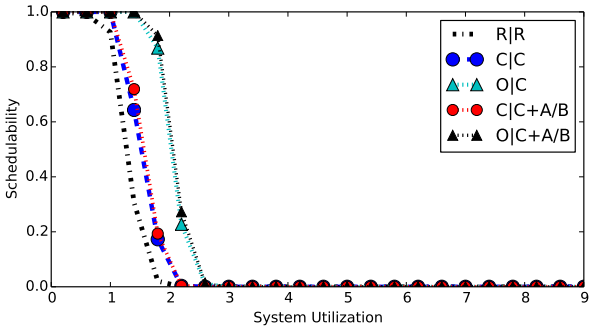
C-Heavy, Long, Heavy, Heavy, Large, High, Few



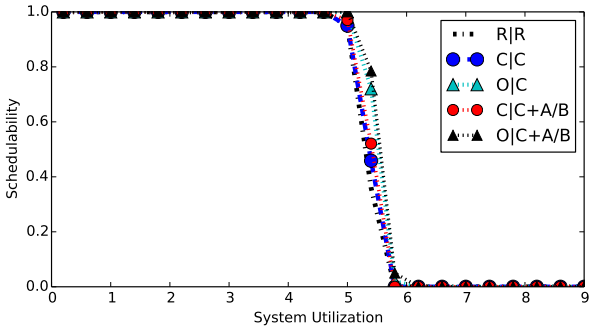
C-Light, Long, Heavy, Heavy, Large, Low, Few



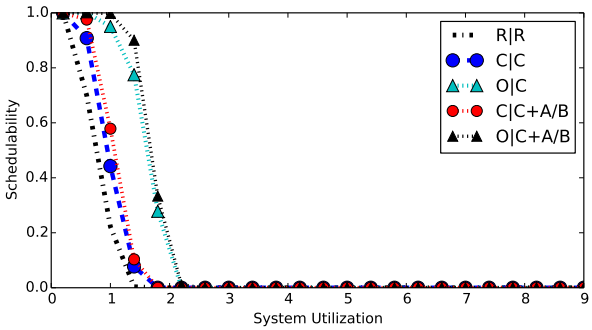
C-Light, Mod., Heavy, Heavy, Large, Med., Many



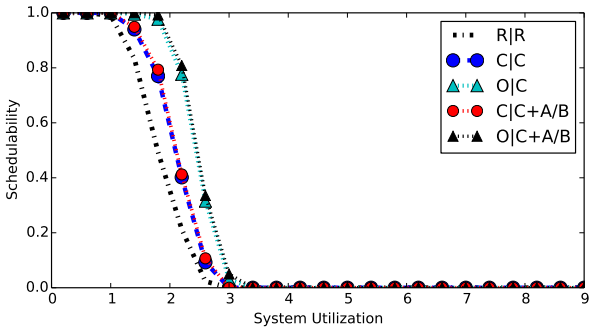
All-Mod., Short, Light, Heavy, Large, Low, Many



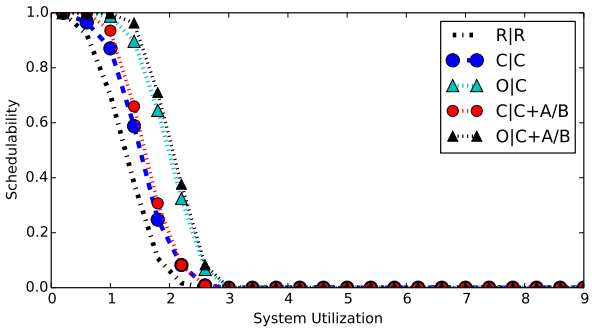
C-Heavy, Long, Med., Heavy, Small, High, Many



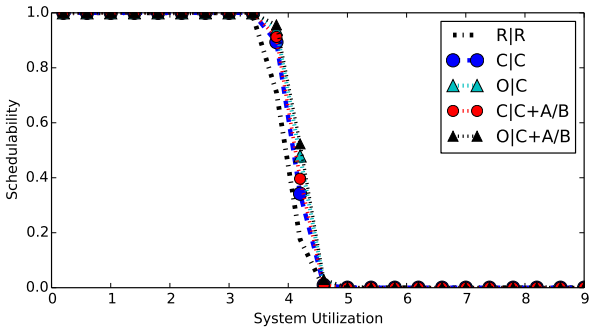
All-Mod., Mod., Light, Heavy, Large, Low, Many



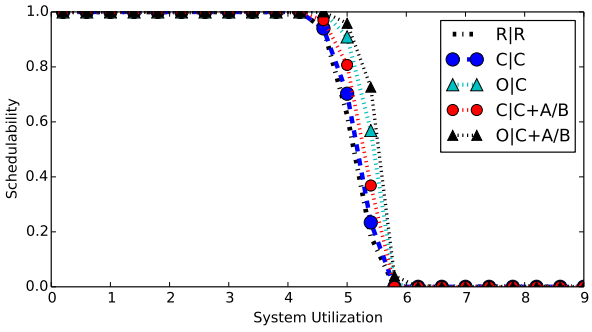
C-Heavy, Short, Light, Heavy, Large, Low, Many



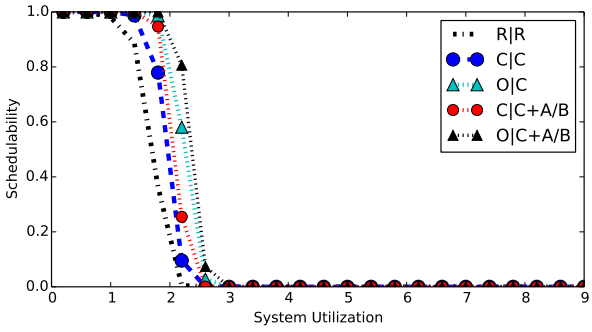
C-Heavy, Mod., Light, Light, Large, Med., Many



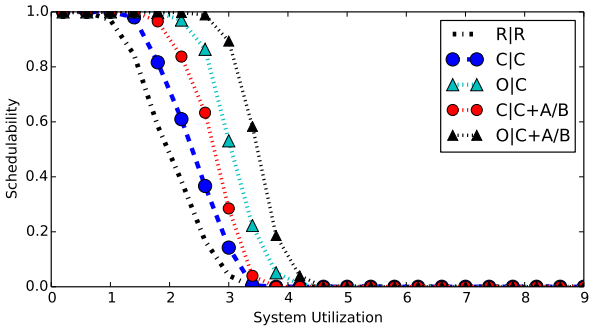
C-Light, Long, Heavy, Heavy, Large, Med., Few



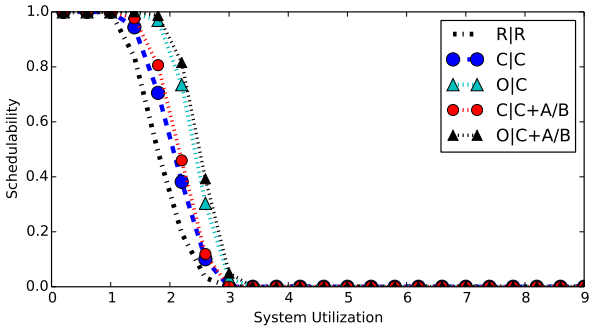
All-Mod., Long, Med., Light, Small, High, Few



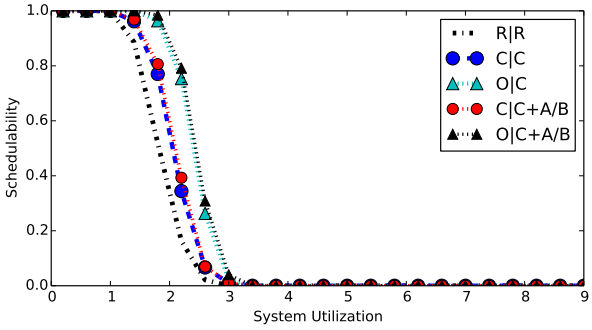
C-Light, Short, Light, Light, Small, High, Many



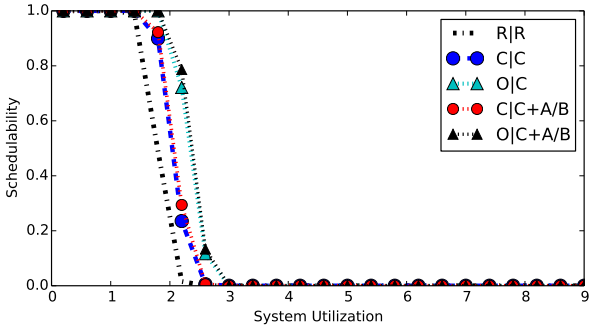
C-Light, Mod., Med., Light, Large, High, Few



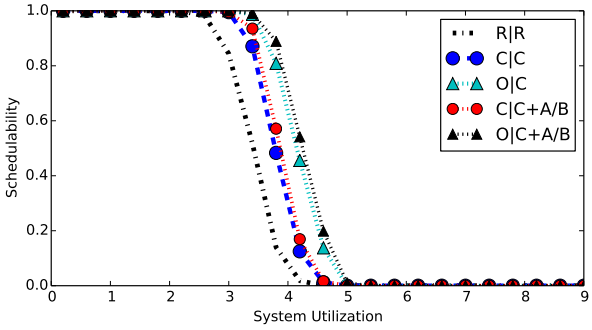
C-Heavy, Short, Light, Light, Large, High, Few



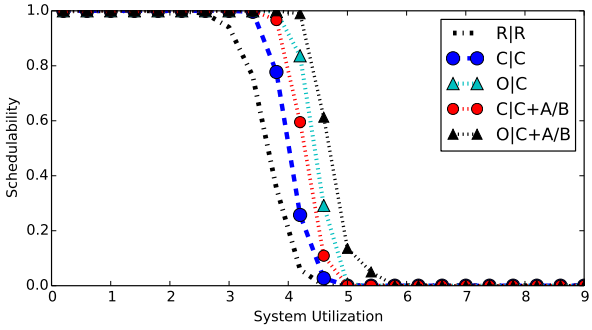
C-Heavy, Short, Light, Heavy, Large, Med., Many



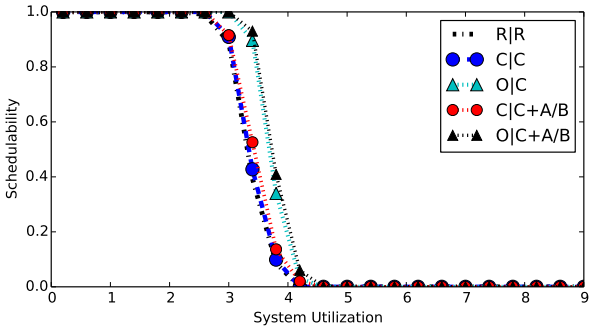
C-Light, Short, Light, Light, Small, Med., Many



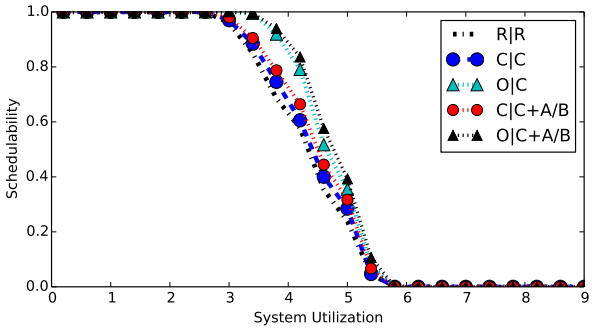
C-Light, Long, Med., Heavy, Large, Med., Many



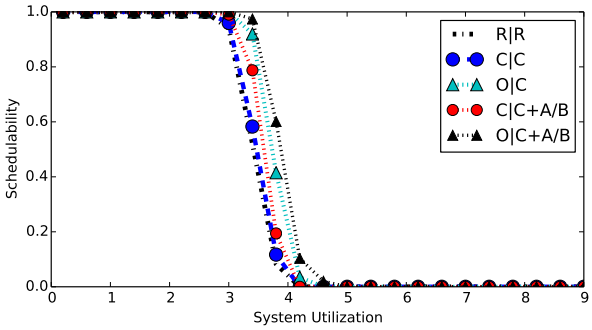
C-Light, Short, Med., Light, Small, High, Many



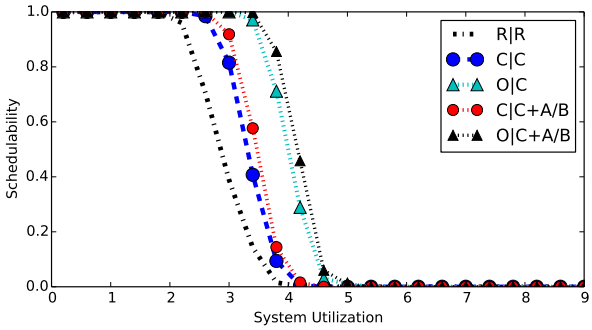
All-Mod., Short, Heavy, Heavy, Small, Low, Many



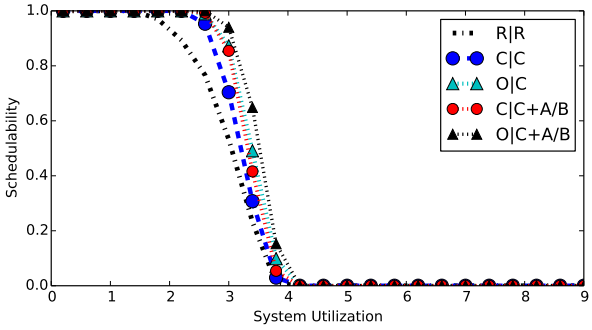
C-Heavy, Short, Heavy, Heavy, Small, Low, Few



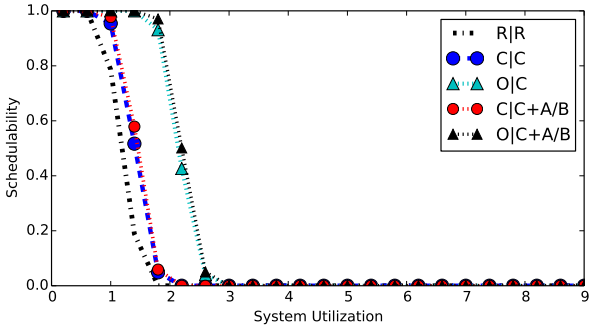
All-Mod., Mod., Heavy, Light, Small, Med., Few



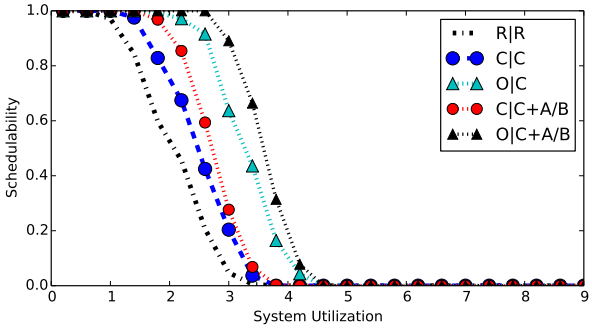
C-Light, Short, Med., Light, Large, Med., Few



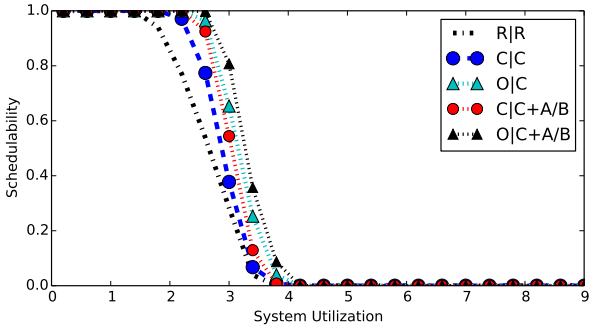
C-Light, Mod., Heavy, Light, Large, Med., Many



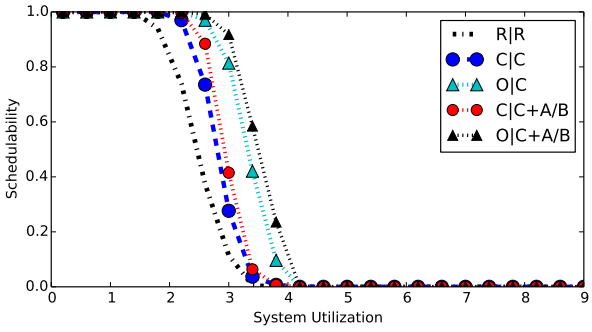
C-Light, Short, Light, Light, Large, Low, Few



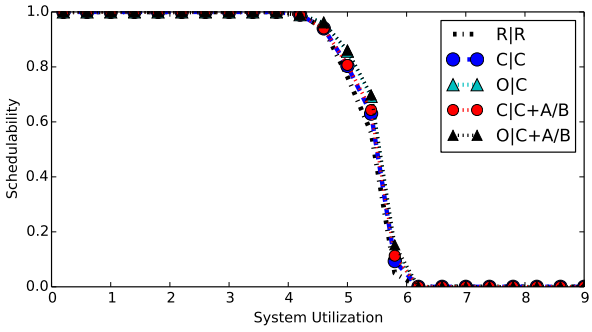
C-Light, Mod., Med., Light, Large, Med., Few



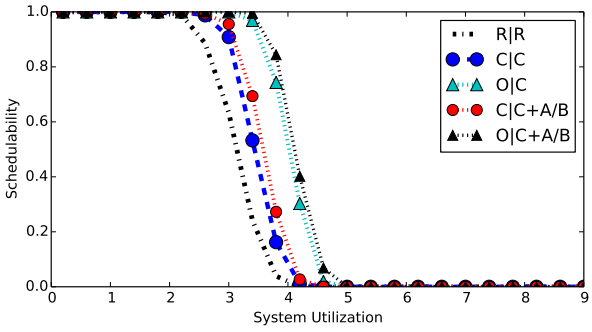
C-Light, Mod., Heavy, Heavy, Large, Low, Many



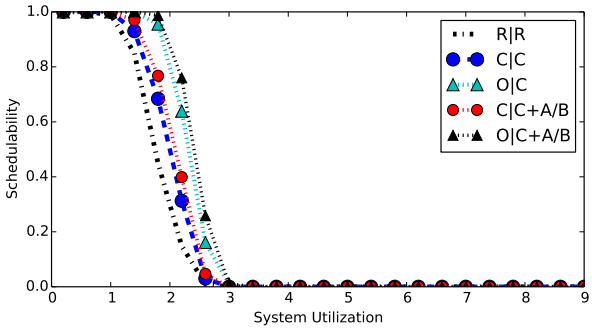
C-Light, Short, Med., Heavy, Large, High, Few



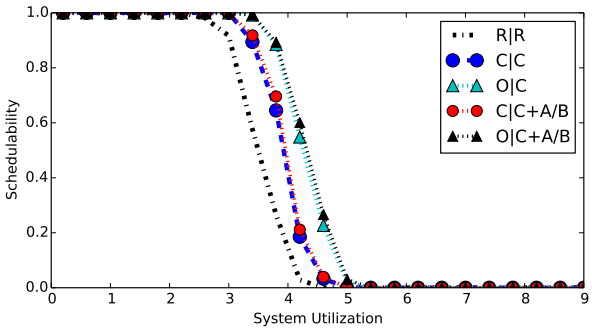
C-Heavy, Long, Heavy, Light, Small, High, Many



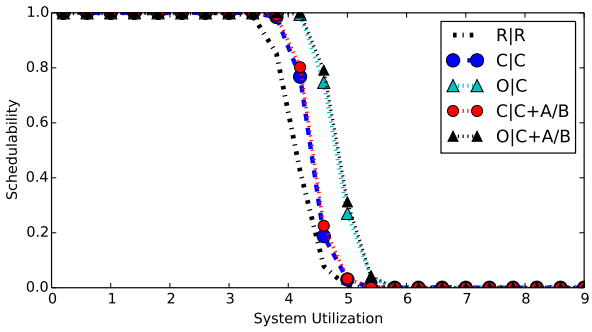
C-Light, Short, Med., Heavy, Small, Med., Many



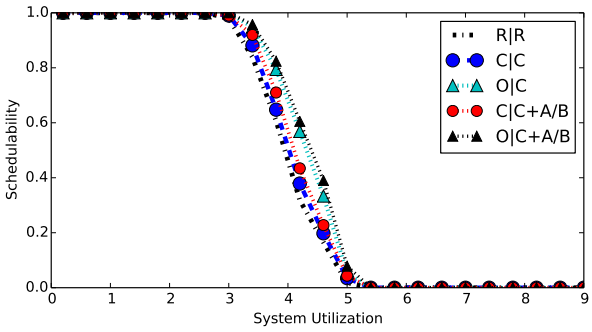
C-Heavy, Short, Light, Heavy, Large, High, Many



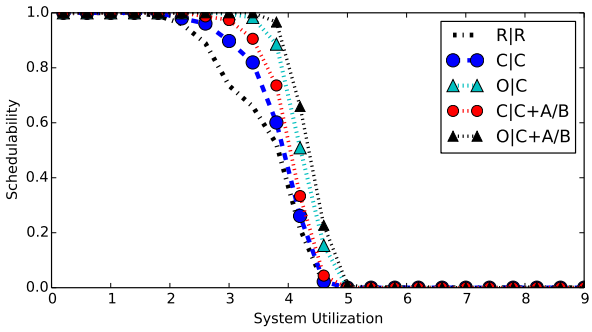
C-Light, Long, Med., Heavy, Large, Low, Many



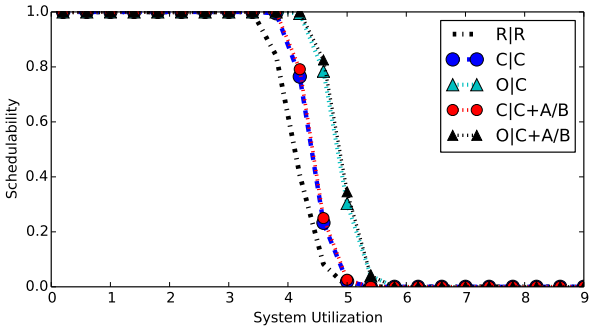
C-Light, Long, Med., Light, Large, Low, Many



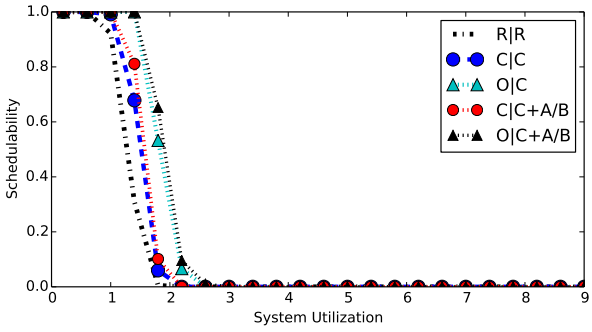
C-Heavy, Short, Heavy, Light, Large, High, Few



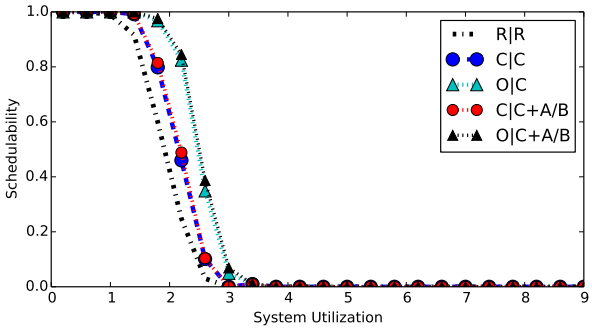
C-Heavy, Mod., Med., Light, Large, Low, Few



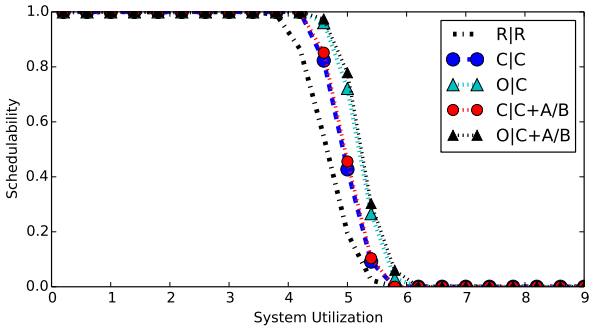
C-Light, Long, Med., Light, Large, Low, Few



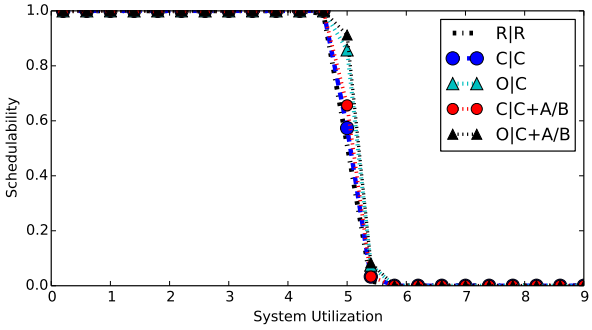
C-Light, Mod., Light, Heavy, Small, Med., Few



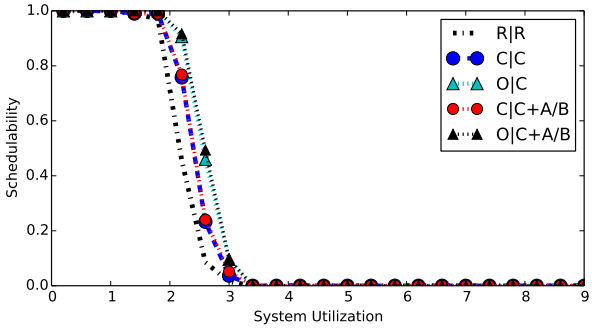
C-Heavy, Short, Light, Light, Large, Low, Few



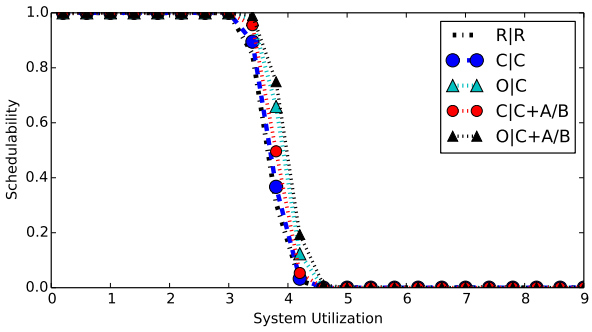
All-Mod., Long, Med., Heavy, Small, Low, Few



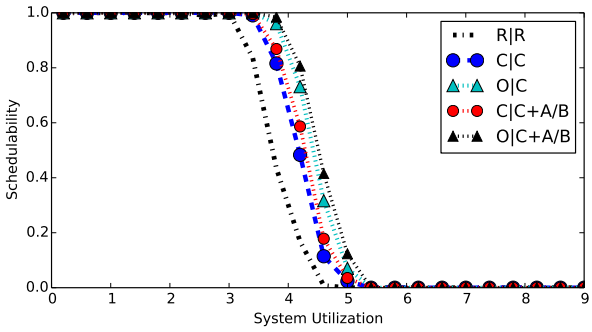
C-Heavy, Long, Med., Light, Large, High, Many



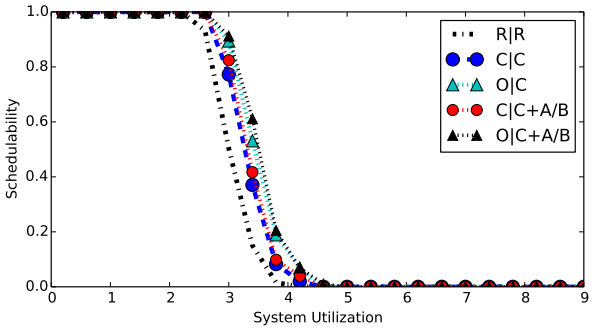
C-Heavy, Short, Light, Heavy, Small, Low, Few



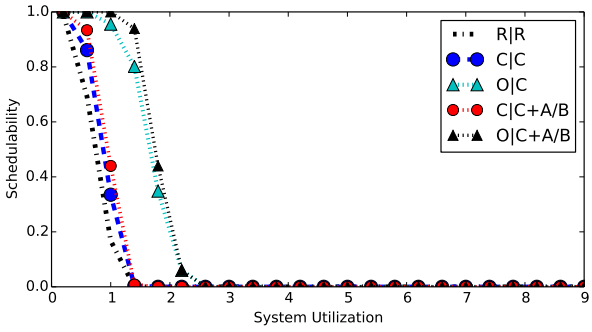
All-Mod., Short, Heavy, Light, Small, Med., Many



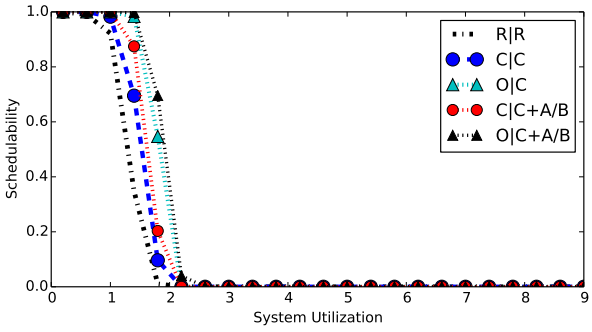
C-Light, Long, Med., Heavy, Small, Med., Few



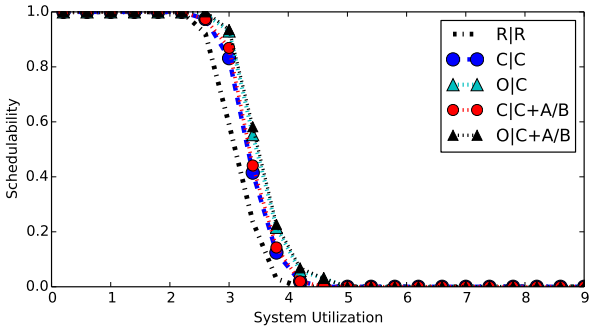
C-Heavy, Long, Light, Heavy, Small, High, Few



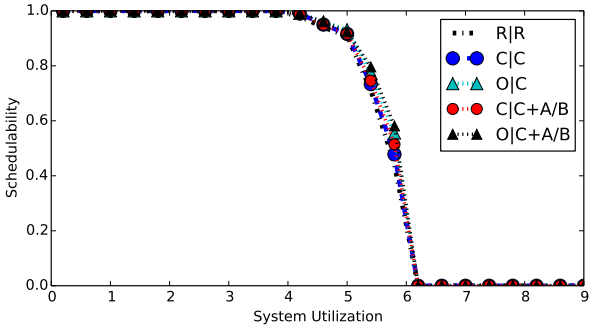
C-Light, Mod., Light, Light, Large, Low, Many



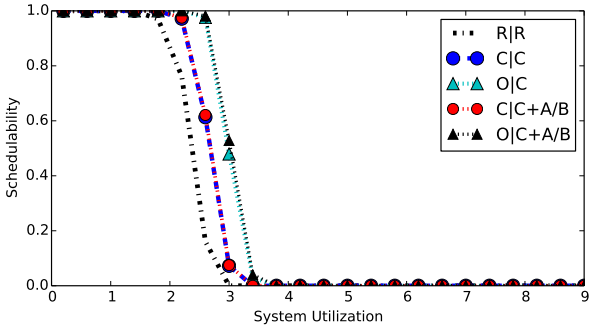
All-Mod., Mod., Light, Heavy, Small, High, Few



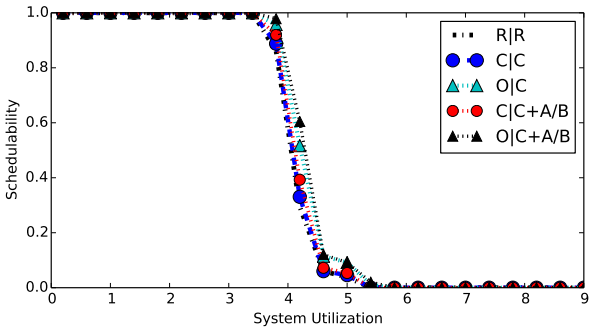
C-Heavy, Long, Light, Heavy, Small, Med., Few



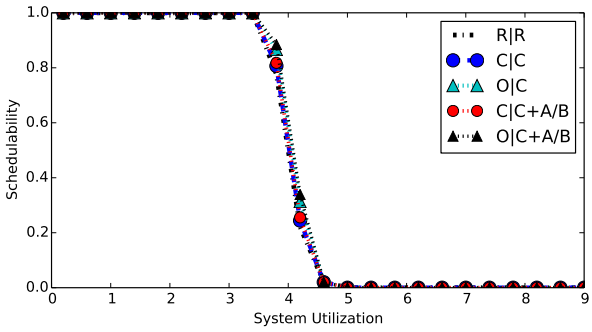
C-Heavy, Long, Heavy, Heavy, Small, High, Many



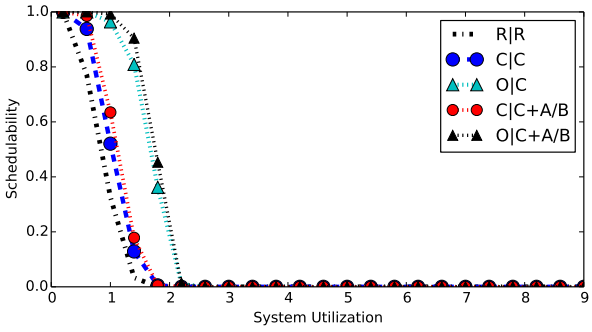
All-Mod., Long, Light, Light, Large, Low, Few



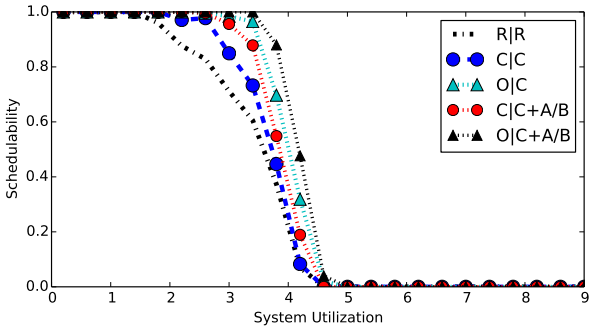
C-Heavy, Short, Med., Heavy, Small, Low, Few



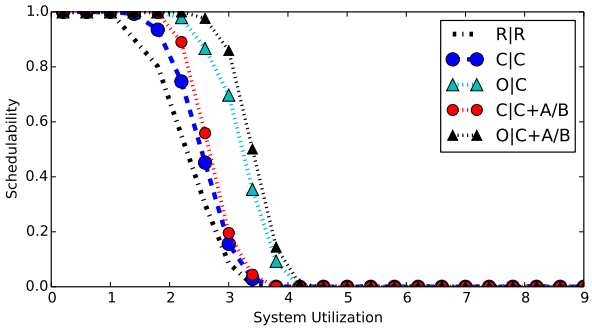
All-Mod., Long, Heavy, Light, Large, Med., Few



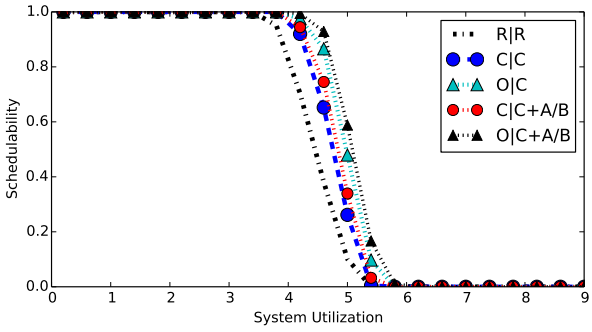
All-Mod., Mod., Light, Light, Large, Low, Many



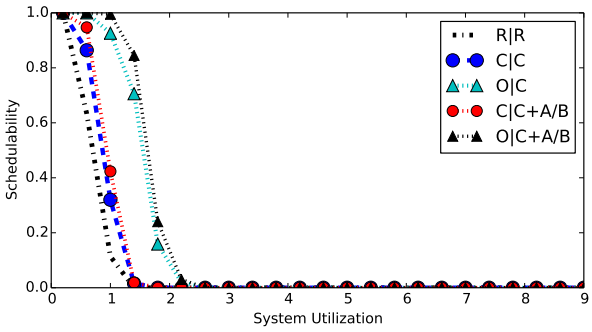
C-Heavy, Mod., Med., Light, Large, High, Few



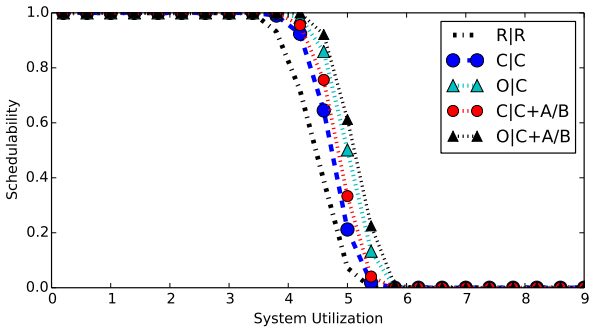
All-Mod., Mod., Med., Heavy, Large, Med., Many



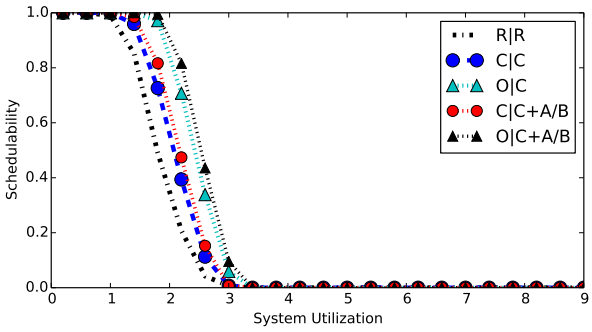
All-Mod., Long, Med., Heavy, Small, High, Many



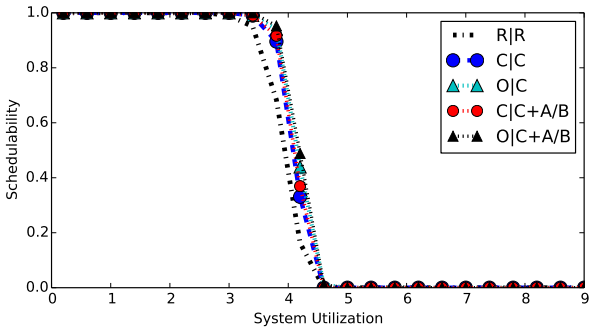
C-Light, Mod., Light, Heavy, Large, Med., Many



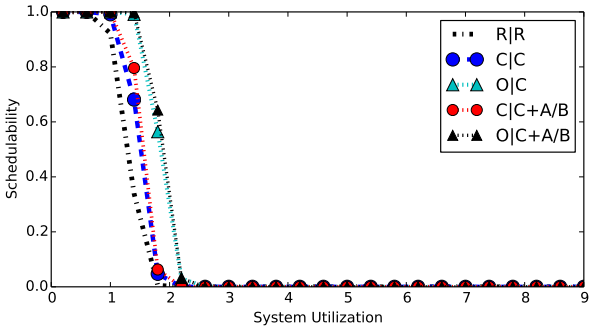
All-Mod., Long, Med., Heavy, Small, High, Few



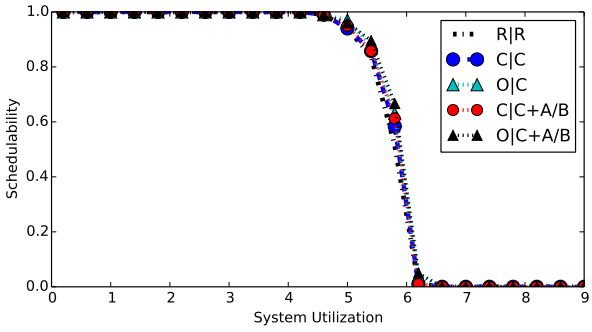
C-Heavy, Short, Light, Light, Large, High, Many



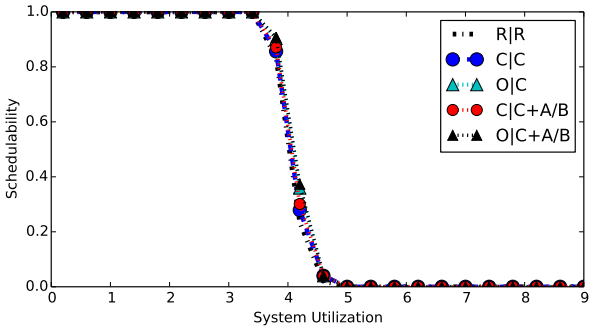
C-Light, Long, Heavy, Heavy, Large, Med., Many



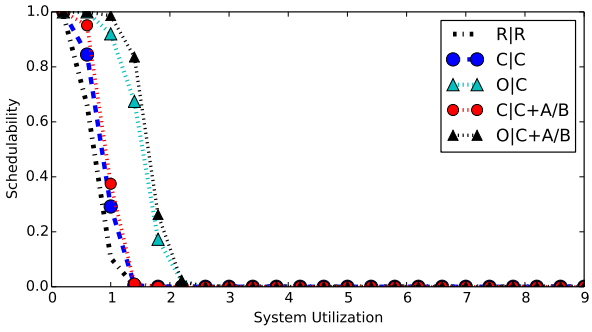
C-Light, Mod., Light, Heavy, Small, Low, Few



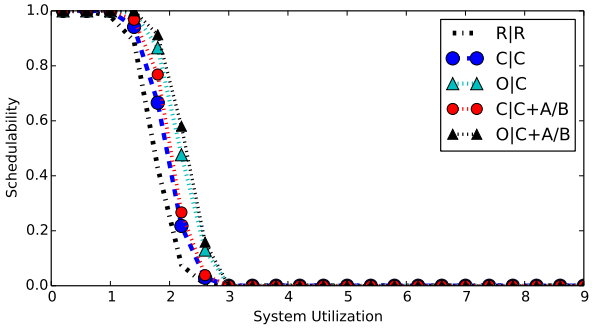
C-Heavy, Long, Heavy, Heavy, Small, Low, Many



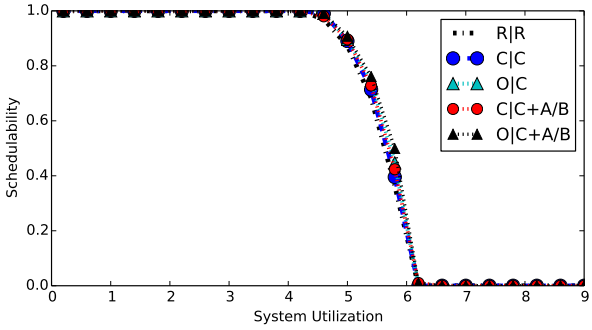
All-Mod., Long, Heavy, Light, Large, Low, Few



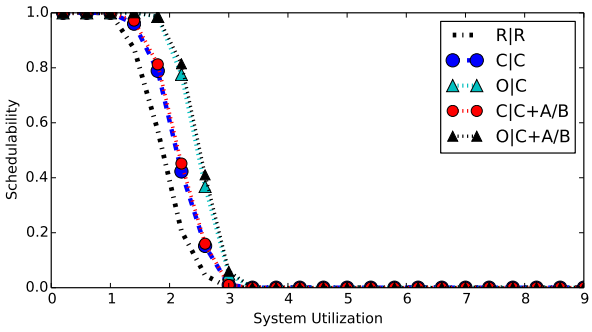
C-Light, Mod., Light, Heavy, Large, Low, Few



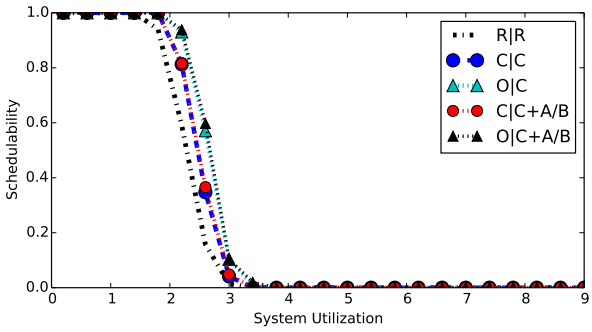
C-Heavy, Mod., Light, Light, Small, High, Few



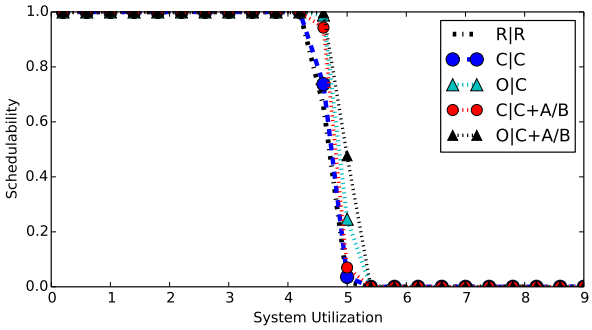
C-Heavy, Long, Heavy, Heavy, Large, Low, Few



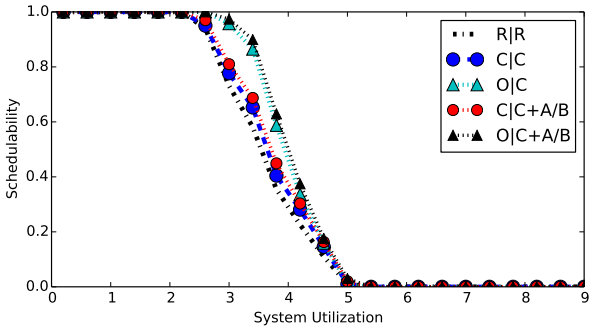
C-Heavy, Short, Light, Light, Large, Med., Many



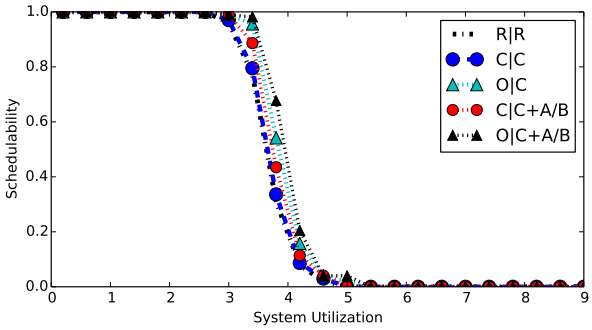
C-Heavy, Short, Light, Light, Small, Med., Few



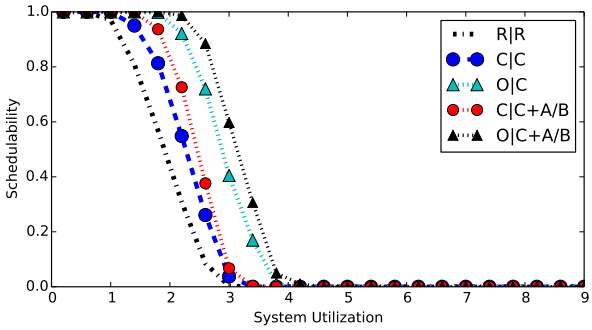
C-Heavy, Short, Med., Light, Small, High, Many



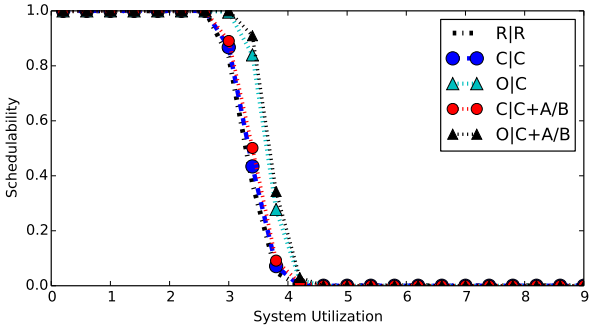
C-Heavy, Short, Heavy, Heavy, Large, High, Many



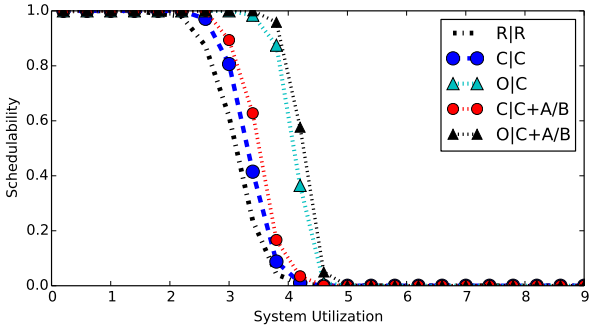
C-Heavy, Mod., Med., Heavy, Small, Med., Many



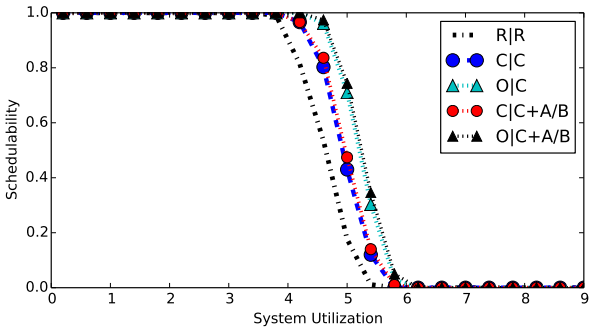
C-Light, Mod., Med., Heavy, Large, Med., Many



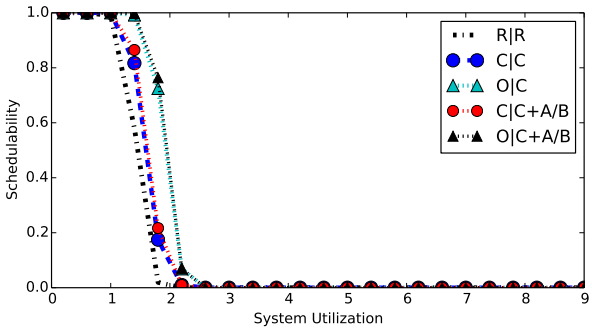
All-Mod., Short, Heavy, Heavy, Small, Med., Many



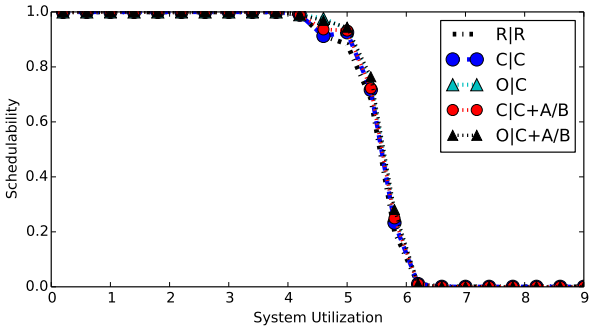
All-Mod., Mod., Med., Heavy, Small, Med., Few



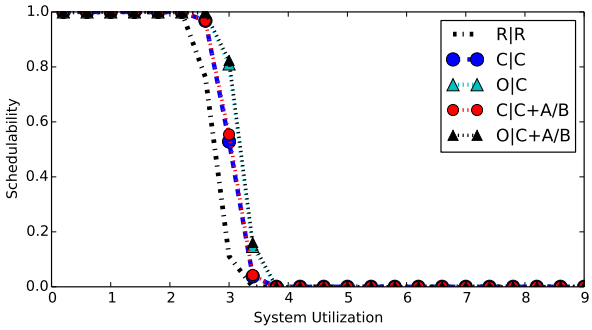
All-Mod., Long, Med., Heavy, Small, Low, Many



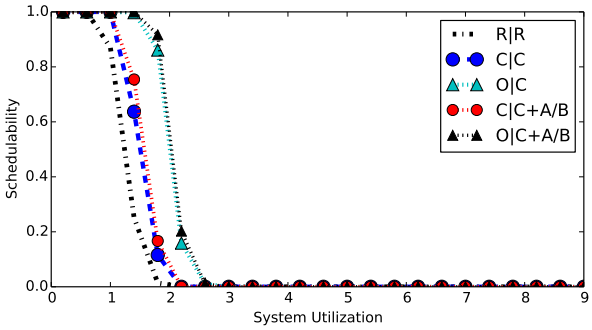
All-Mod., Mod., Light, Heavy, Small, Low, Many



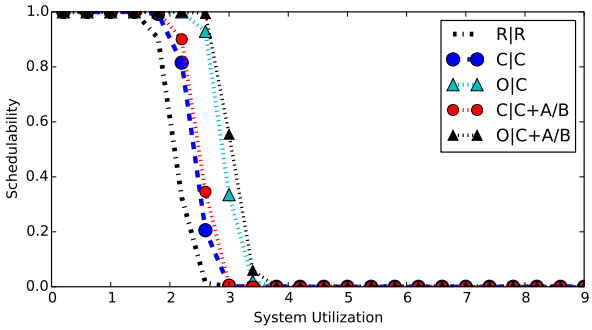
C-Heavy, Long, Heavy, Heavy, Large, High, Many



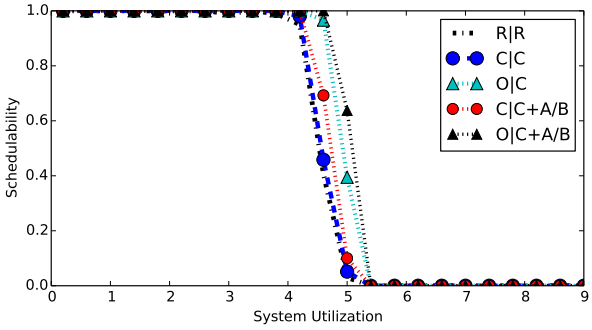
All-Mod., Long, Light, Light, Small, Low, Many



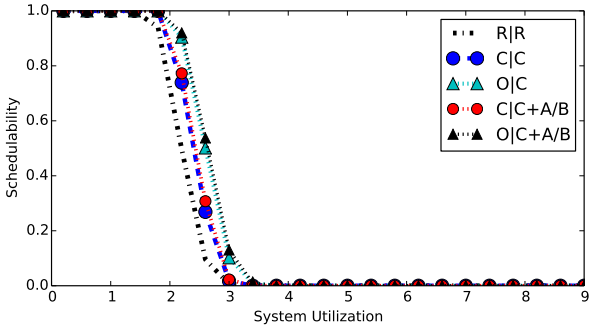
All-Mod., Short, Light, Heavy, Large, Med., Many



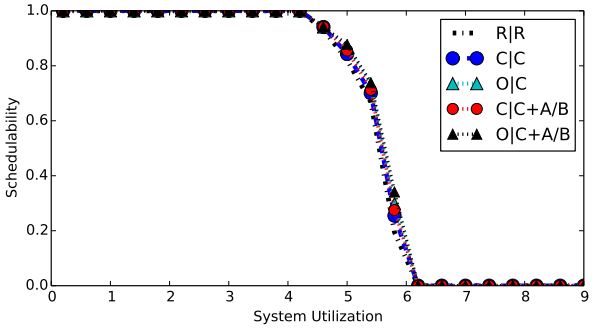
C-Light, Long, Light, Light, Large, High, Many



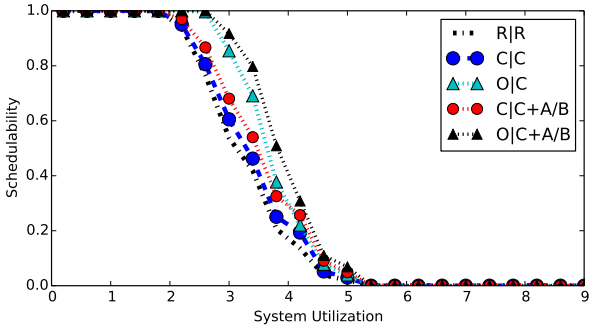
C-Heavy, Mod., Med., Light, Small, Low, Many



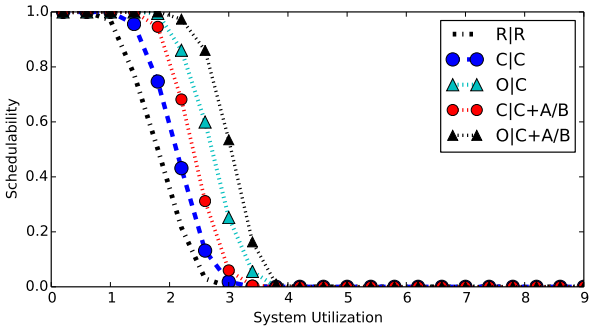
C-Heavy, Short, Light, Heavy, Small, Med., Many



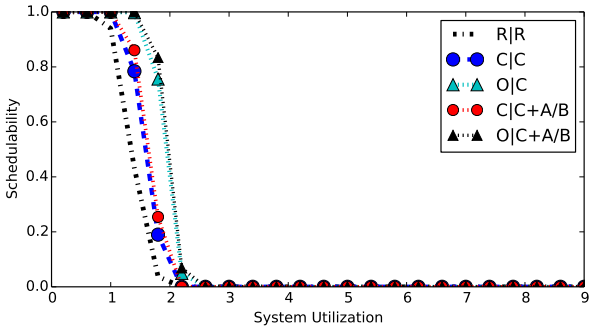
C-Heavy, Long, Heavy, Heavy, Large, Med., Few



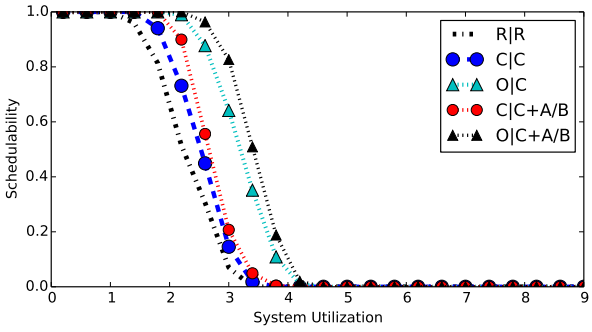
C-Heavy, Mod., Heavy, Heavy, Large, Med., Many



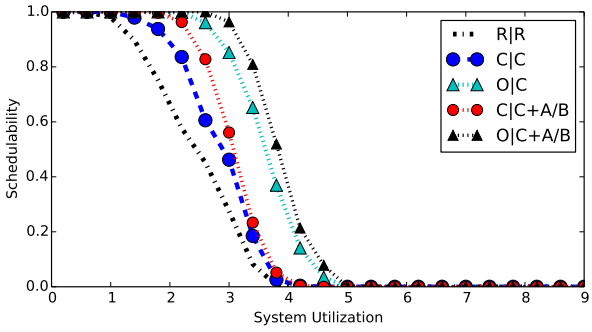
C-Light, Mod., Med., Heavy, Large, High, Few



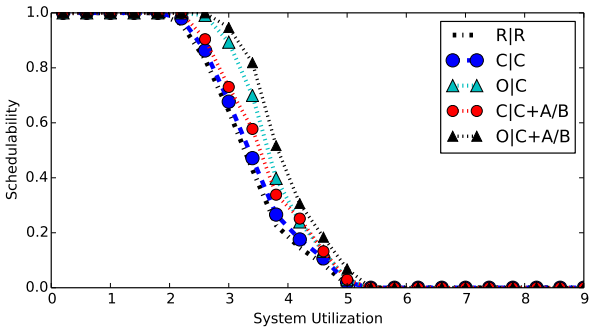
C-Light, Mod., Light, Light, Small, Low, Many



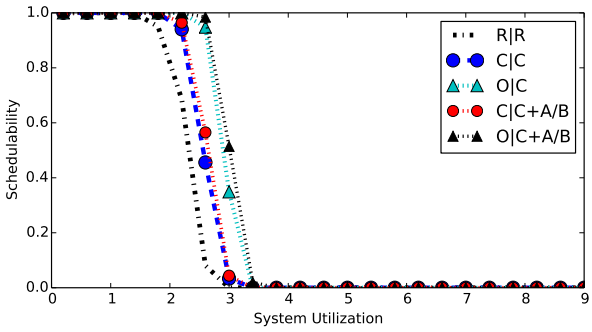
All-Mod., Mod., Med., Heavy, Large, Low, Many



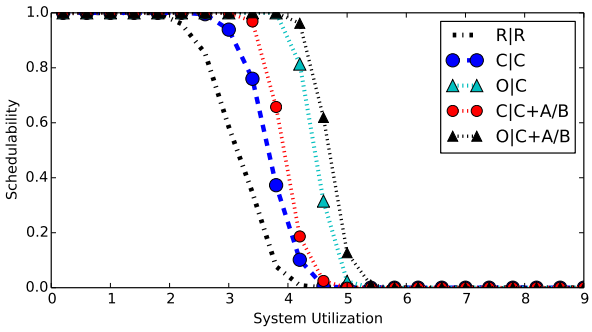
All-Mod., Mod., Med., Light, Large, Med., Few



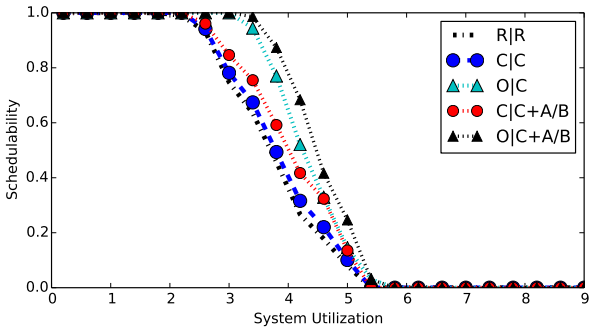
C-Heavy, Mod., Heavy, Heavy, Large, Low, Many



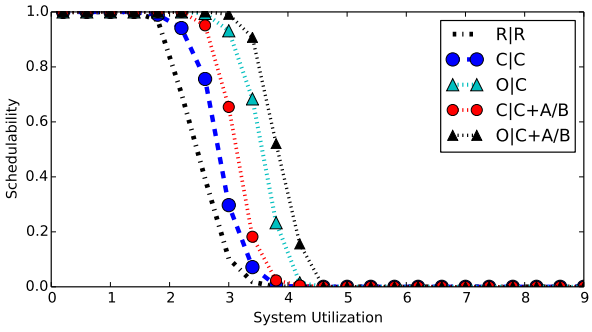
All-Mod., Long, Light, Light, Large, High, Few



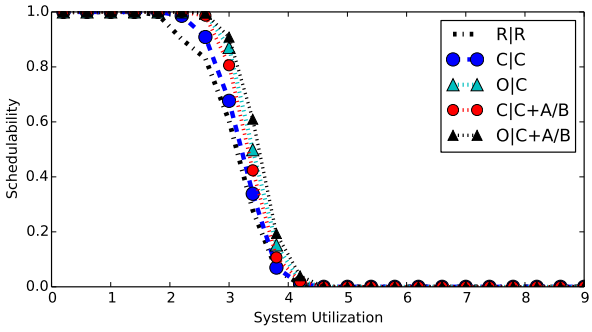
C-Light, Mod., Med., Light, Small, Med., Many



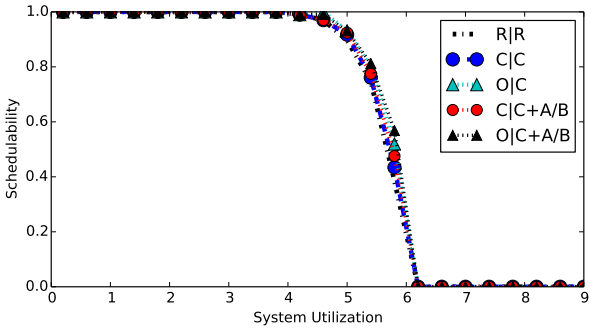
C-Heavy, Mod., Heavy, Heavy, Small, Low, Few



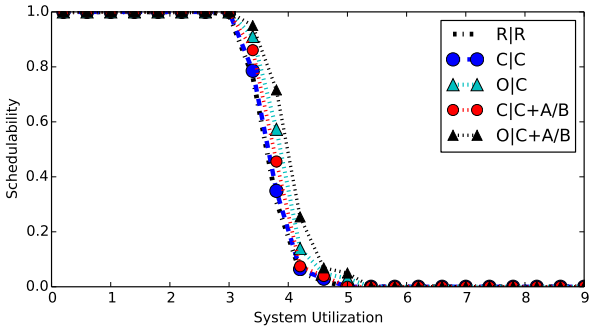
C-Light, Mod., Med., Heavy, Small, High, Few



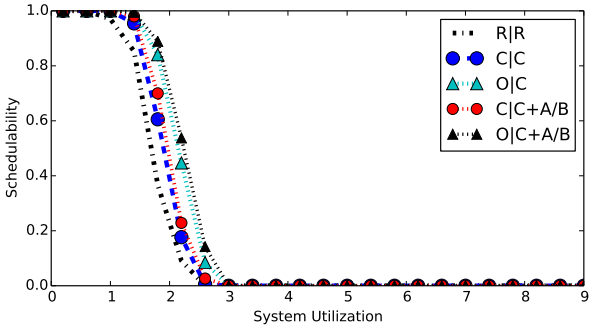
C-Heavy, Mod., Med., Heavy, Large, Med., Many



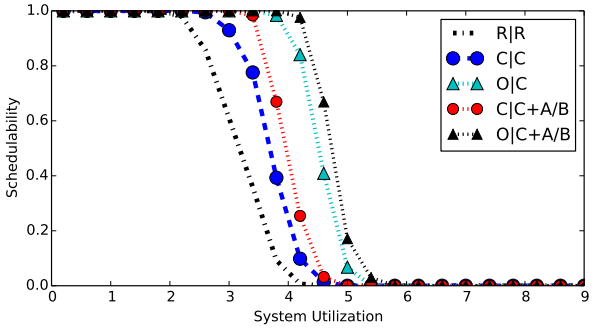
C-Heavy, Long, Heavy, Heavy, Small, High, Few



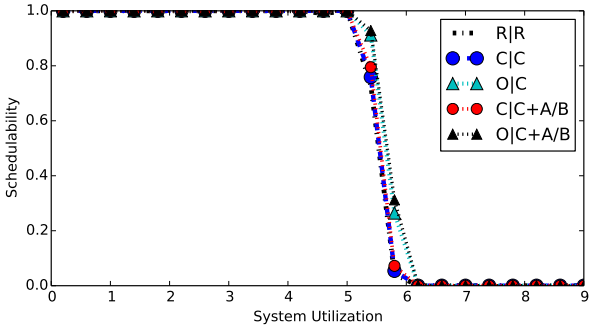
C-Heavy, Mod., Med., Heavy, Small, Med., Few



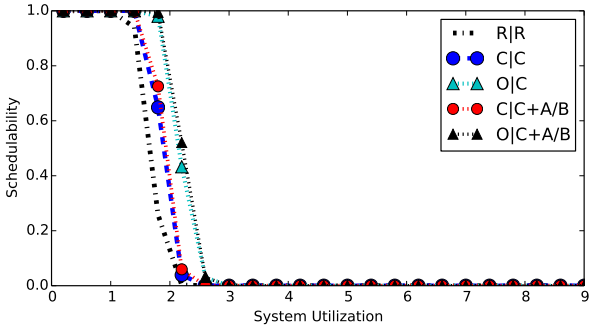
C-Heavy, Mod., Light, Heavy, Small, High, Few



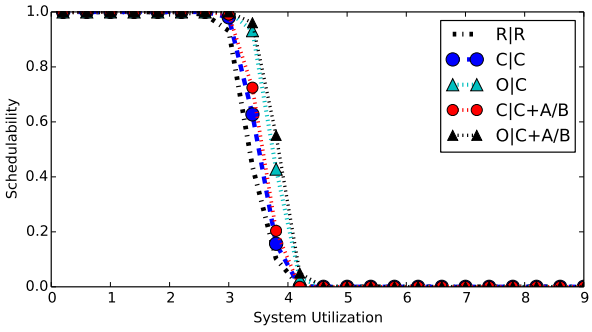
C-Light, Mod., Med., Light, Small, Low, Few



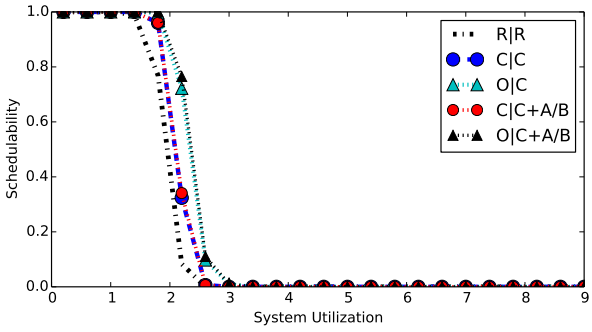
C-Heavy, Long, Med., Heavy, Small, Low, Few



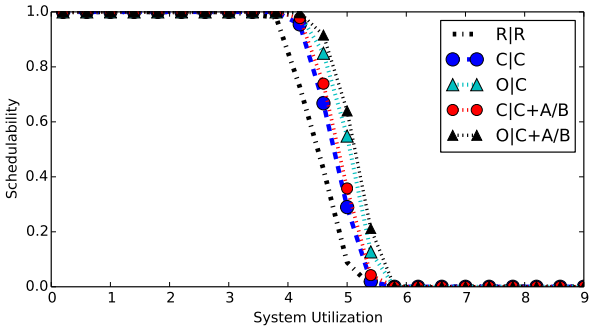
C-Light, Short, Light, Heavy, Small, Med., Many



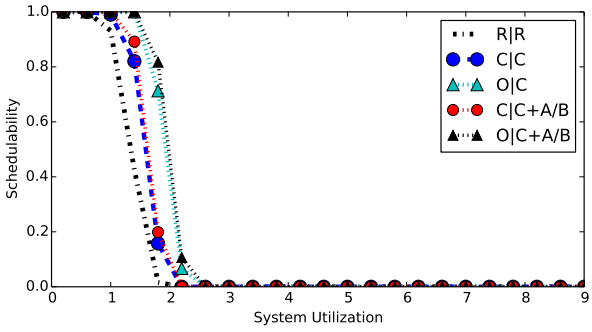
C-Light, Short, Heavy, Heavy, Small, Med., Many



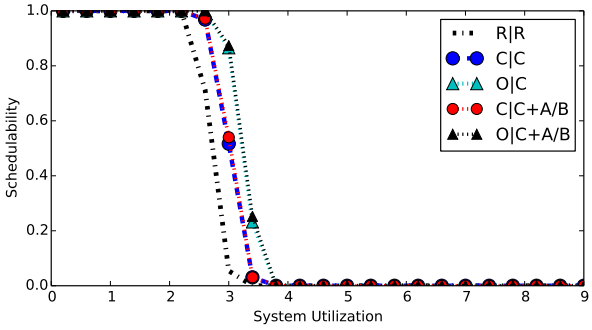
All-Mod., Short, Light, Light, Small, Low, Many



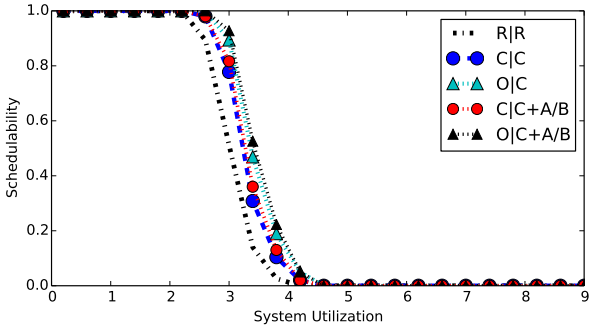
All-Mod., Long, Med., Heavy, Small, Med., Few



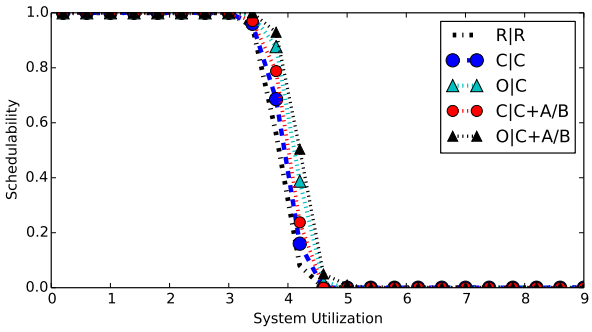
C-Light, Mod., Light, Light, Small, Med., Few



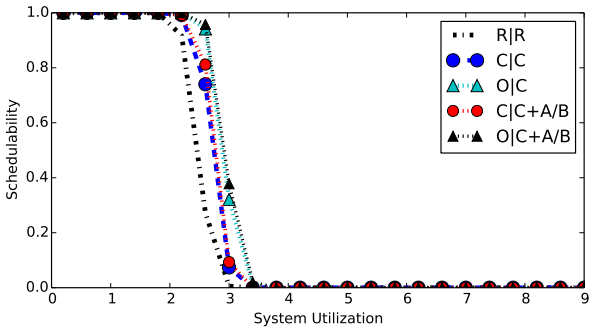
C-Light, Long, Light, Light, Small, Low, Many



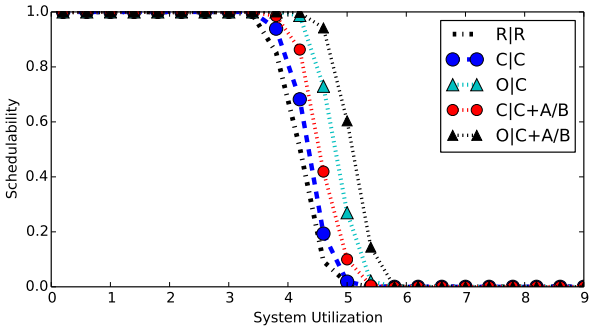
C-Heavy, Long, Light, Heavy, Small, High, Many



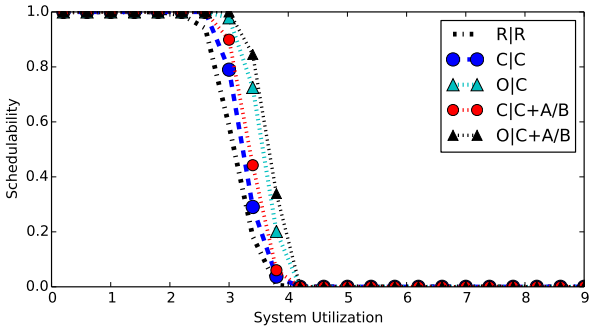
C-Light, Short, Heavy, Light, Small, High, Few



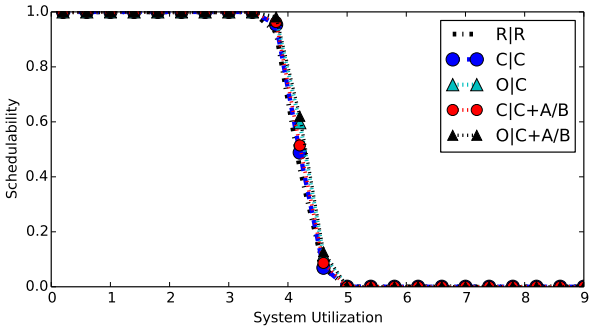
All-Mod., Long, Light, Heavy, Small, Med., Few



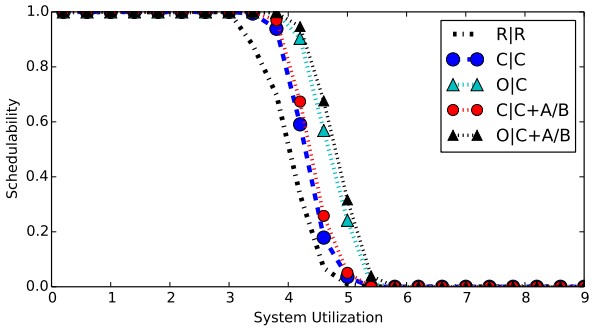
All-Mod., Short, Med., Light, Small, High, Few



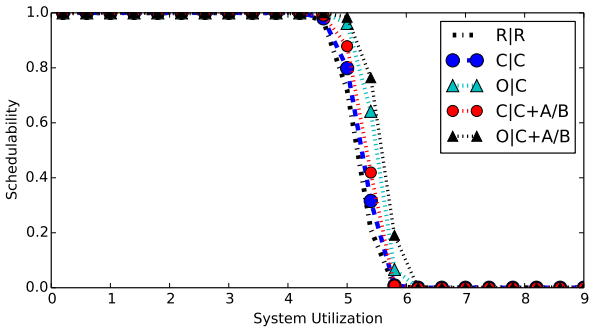
C-Light, Mod., Heavy, Heavy, Small, High, Many



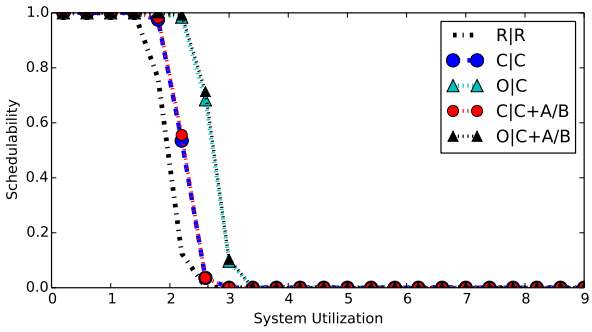
All-Mod., Long, Heavy, Light, Small, Low, Few



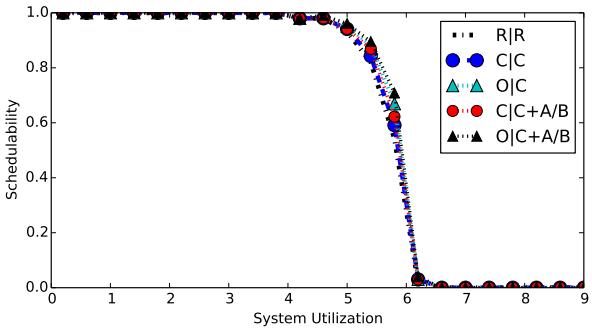
All-Mod., Long, Med., Heavy, Large, High, Few



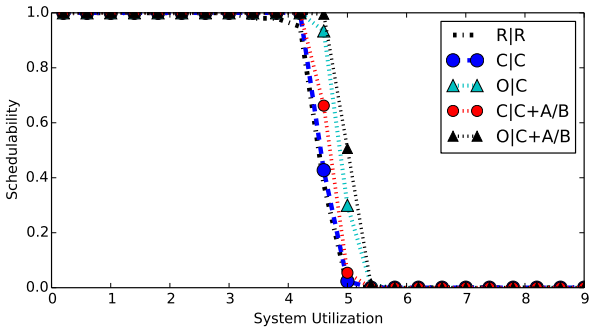
All-Mod., Long, Med., Light, Small, Med., Many



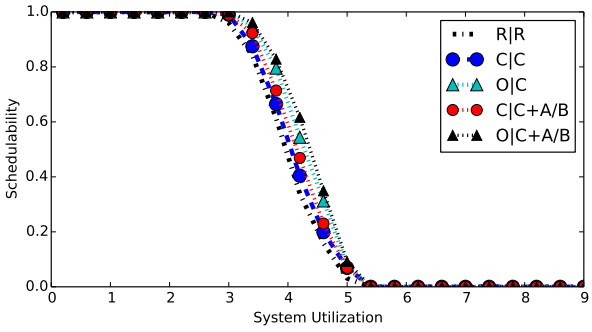
C-Light, Long, Light, Heavy, Large, Low, Few



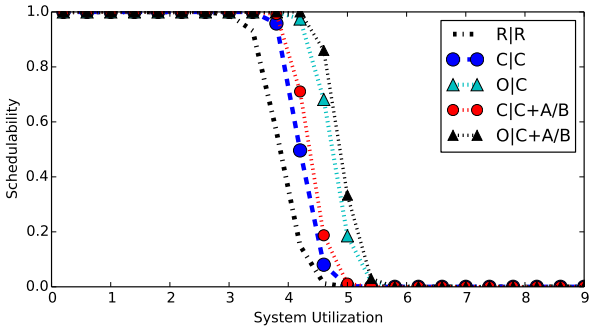
C-Heavy, Long, Heavy, Heavy, Small, Low, Few



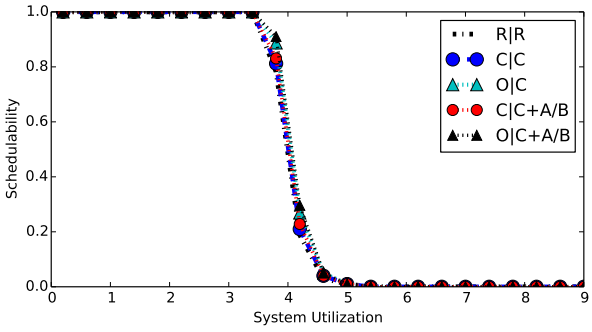
C-Heavy, Mod., Med., Light, Small, Med., Many



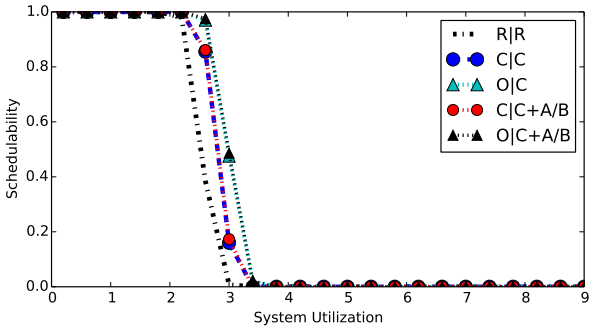
C-Heavy, Short, Heavy, Light, Large, High, Many



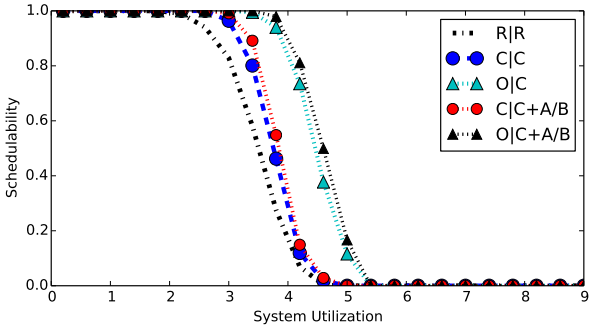
C-Light, Short, Med., Light, Small, Med., Few



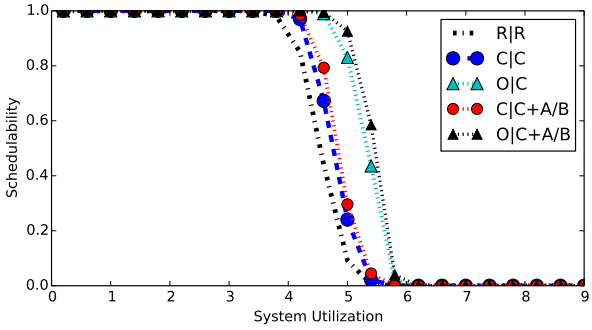
All-Mod., Long, Heavy, Light, Large, High, Many



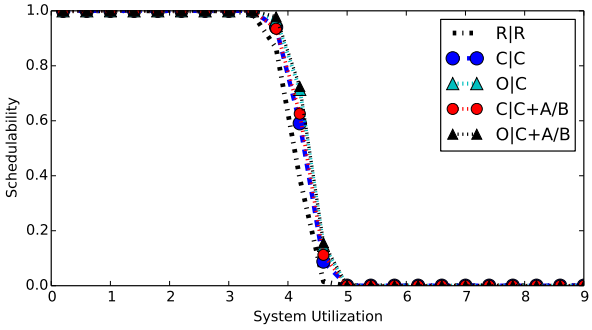
All-Mod., Long, Light, Heavy, Small, Low, Few



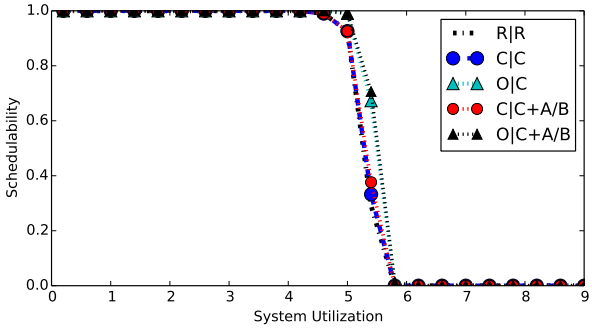
All-Mod., Short, Med., Light, Large, Low, Many



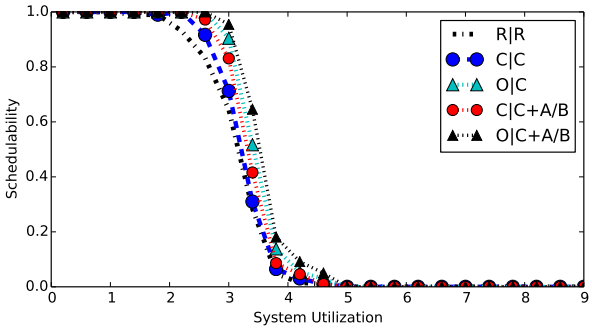
All-Mod., Short, Med., Light, Small, Low, Many



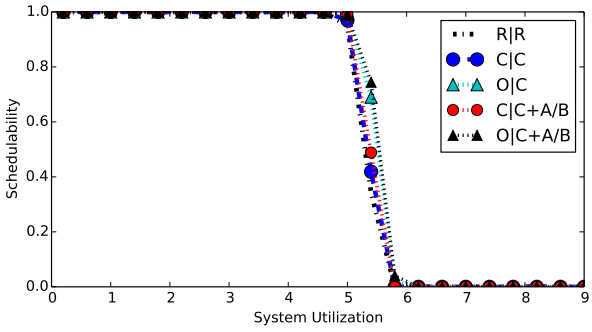
C-Light, Long, Heavy, Heavy, Small, Low, Many



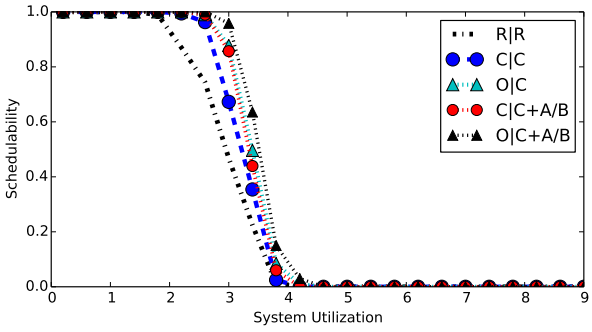
C-Heavy, Long, Med., Heavy, Large, Low, Many



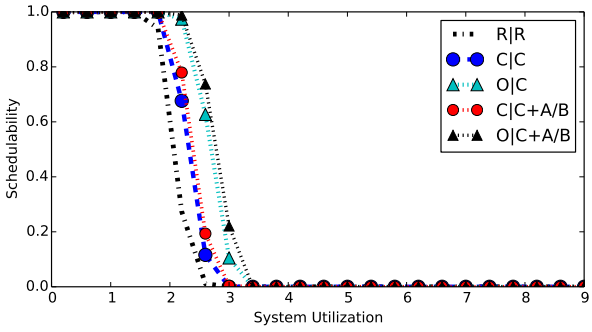
C-Heavy, Mod., Med., Heavy, Large, Low, Few



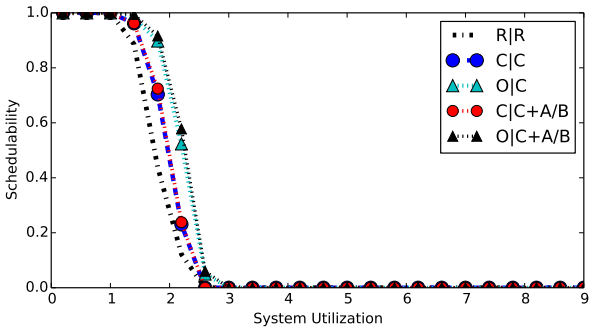
C-Heavy, Long, Med., Heavy, Small, High, Few



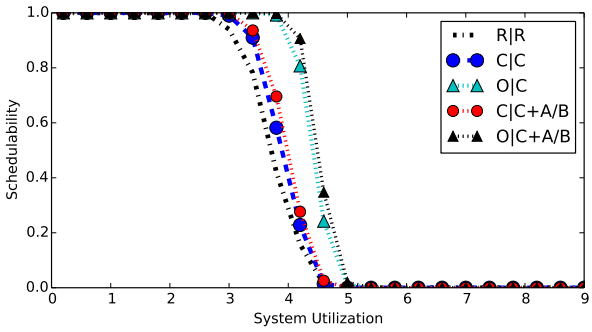
C-Light, Mod., Heavy, Light, Large, Low, Few



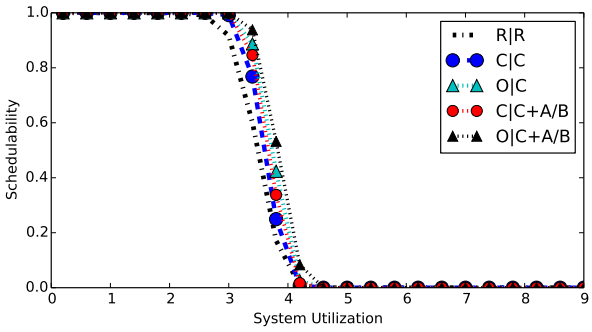
All-Mod., Long, Light, Heavy, Large, High, Many



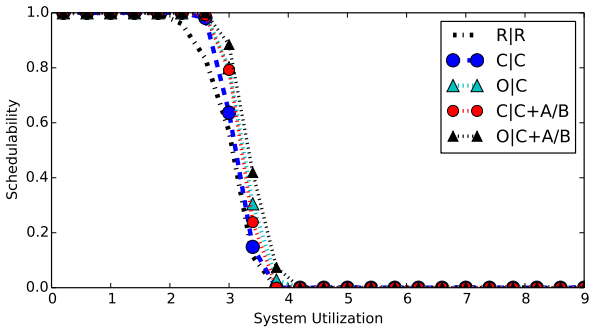
C-Heavy, Mod., Light, Heavy, Small, Low, Few



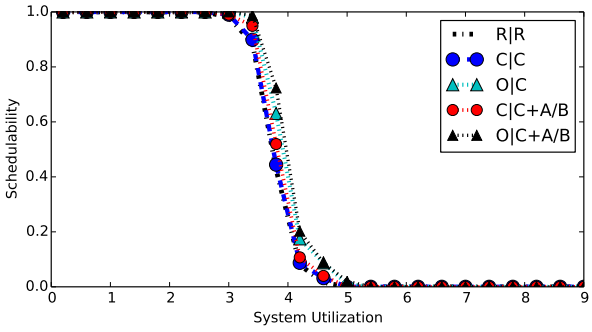
All-Mod., Short, Med., Heavy, Small, Low, Few



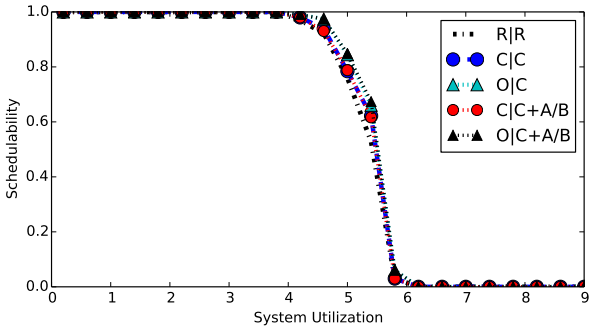
C-Light, Short, Heavy, Light, Large, High, Few



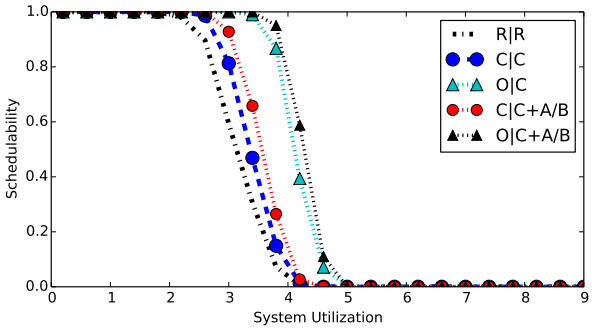
All-Mod., Mod., Heavy, Light, Large, High, Many



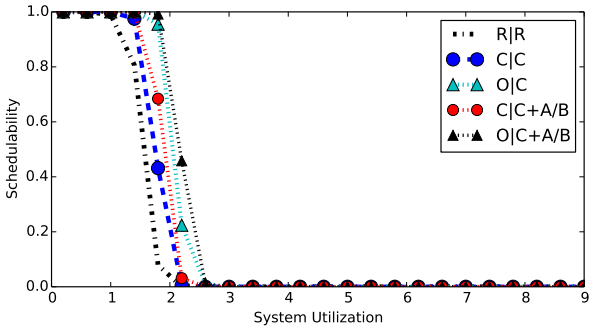
C-Heavy, Short, Med., Heavy, Large, Low, Many



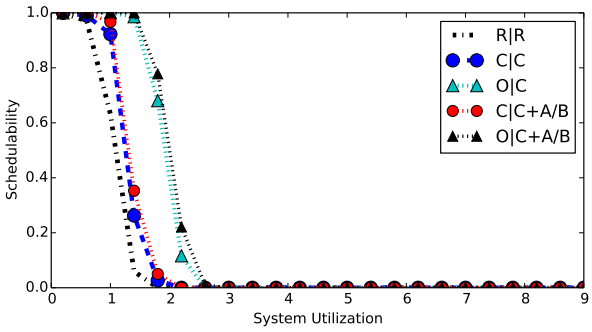
C-Heavy, Long, Heavy, Light, Small, High, Few



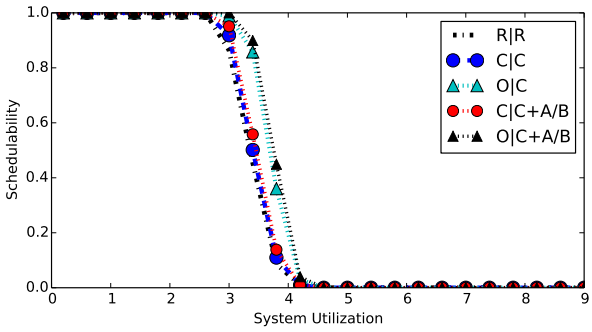
All-Mod., Mod., Med., Heavy, Small, Low, Many



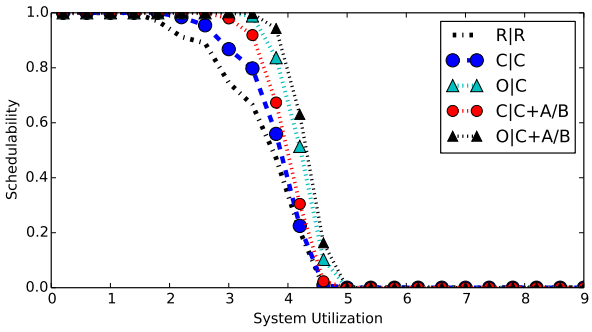
C-Light, Short, Light, Heavy, Small, High, Few



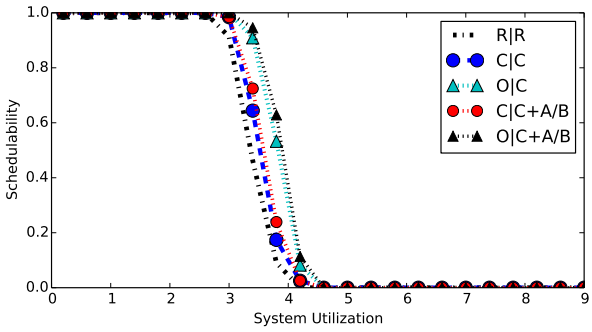
C-Light, Short, Light, Heavy, Large, Med., Few



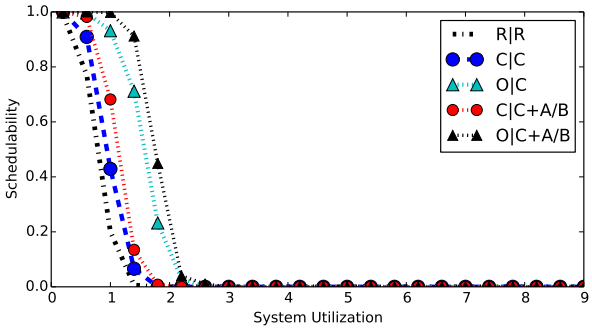
All-Mod., Short, Heavy, Heavy, Small, Low, Few



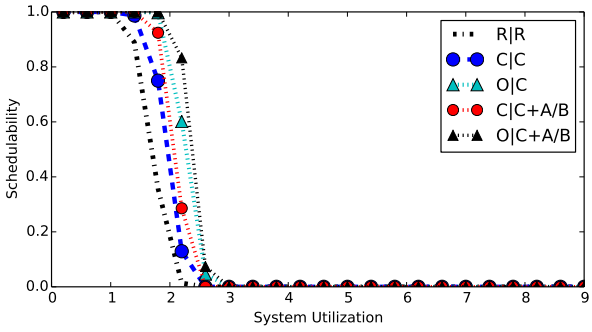
C-Heavy, Mod., Med., Light, Large, Med., Many



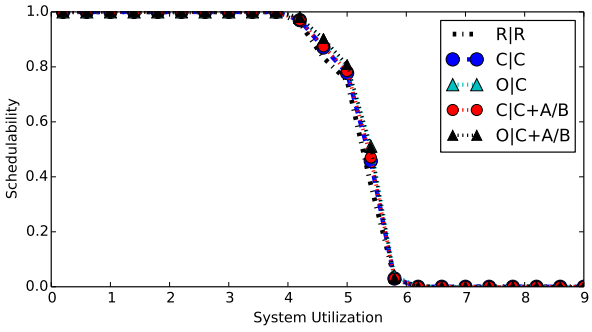
C-Light, Short, Heavy, Heavy, Small, Low, Many



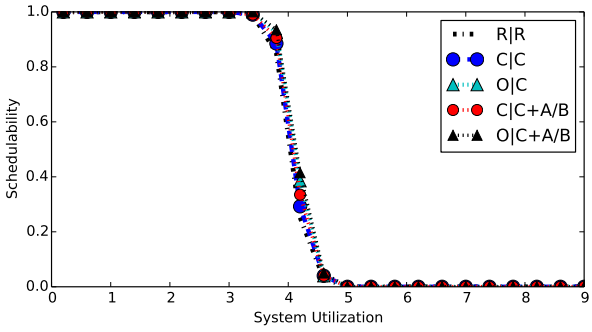
All-Mod., Mod., Light, Light, Large, High, Many



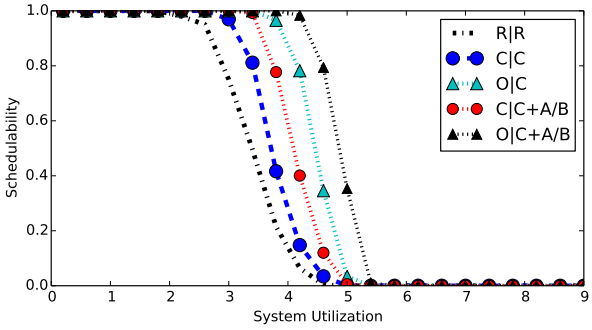
C-Light, Short, Light, Light, Small, High, Few



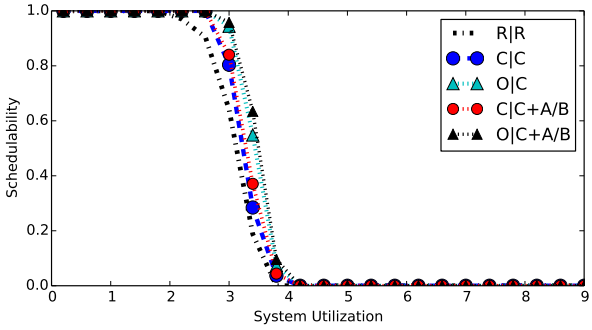
C-Heavy, Long, Heavy, Light, Large, High, Many



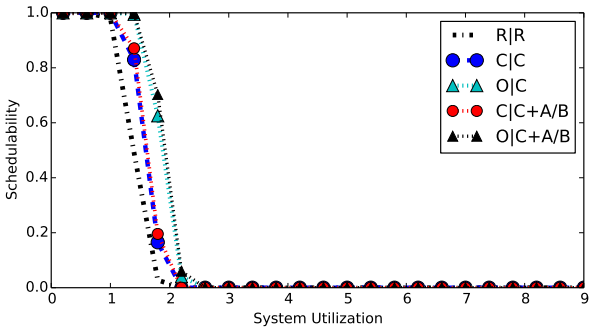
All-Mod., Long, Heavy, Heavy, Large, Med., Few



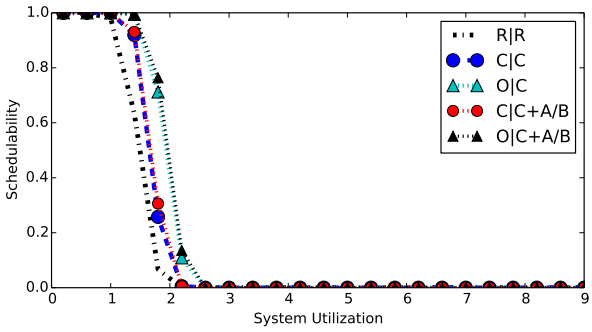
All-Mod., Mod., Med., Light, Small, High, Few



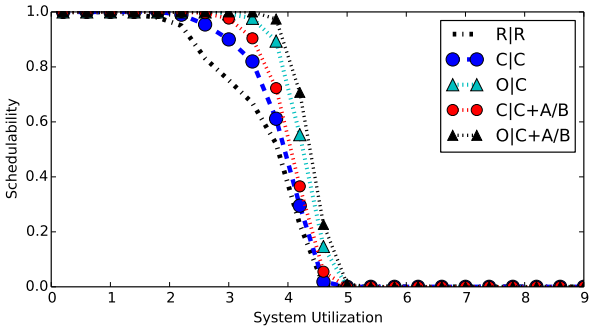
C-Light, Short, Heavy, Heavy, Large, High, Few



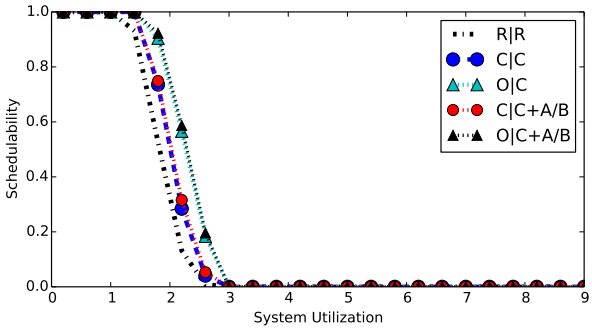
All-Mod., Mod., Light, Heavy, Small, Med., Few



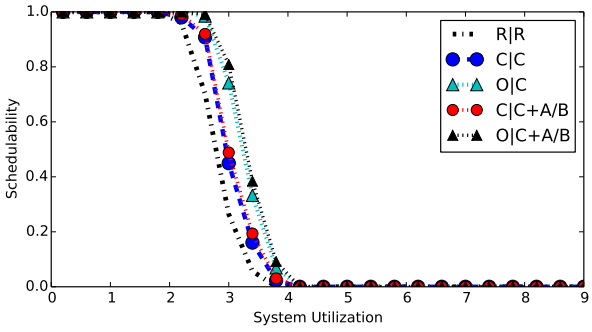
All-Mod., Mod., Light, Light, Small, Med., Few



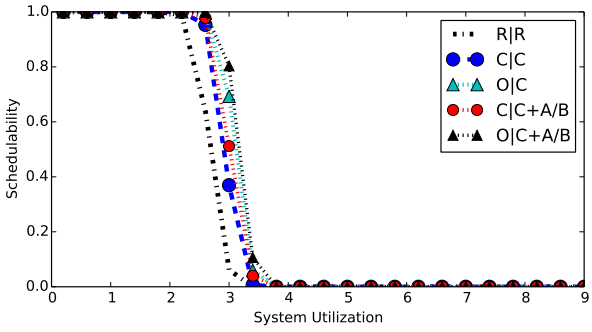
C-Heavy, Mod., Med., Light, Large, Low, Many



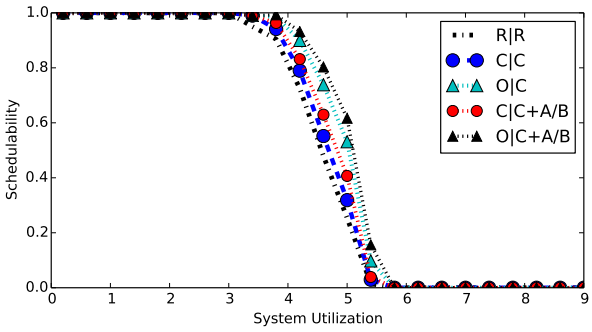
C-Heavy, Mod., Light, Light, Small, Med., Many



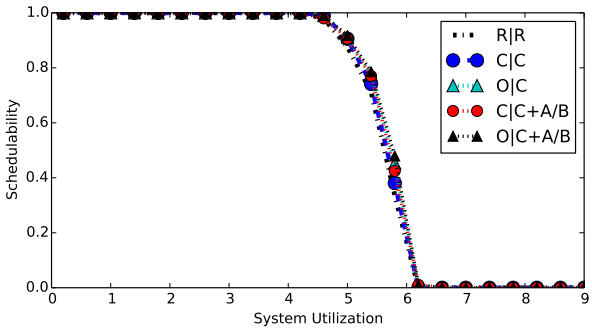
C-Heavy, Long, Light, Heavy, Large, High, Few



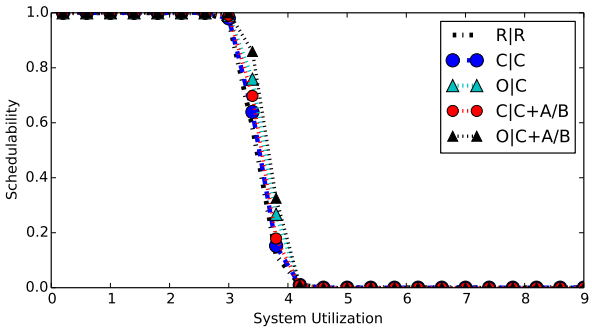
All-Mod., Long, Light, Light, Small, High, Few



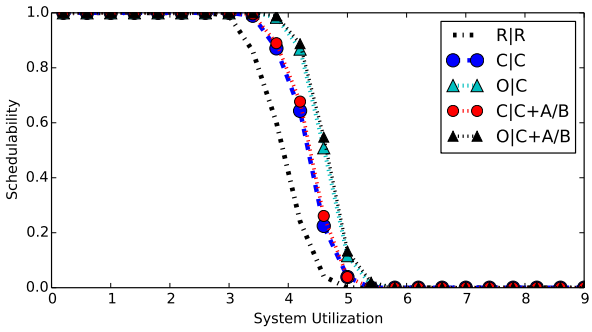
C-Heavy, Short, Heavy, Light, Small, Med., Many



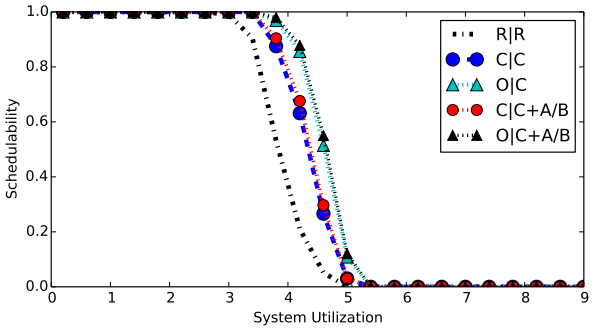
C-Heavy, Long, Heavy, Heavy, Large, Low, Many



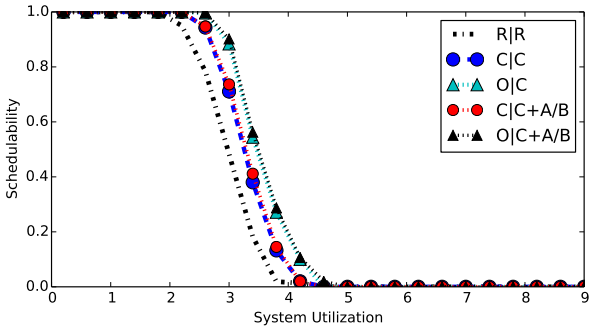
All-Mod., Short, Heavy, Light, Large, Med., Few



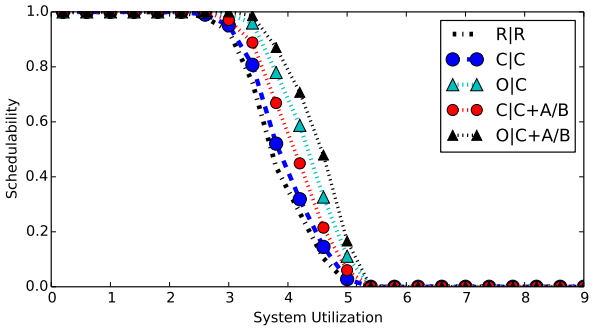
C-Light, Long, Med., Heavy, Small, Low, Few



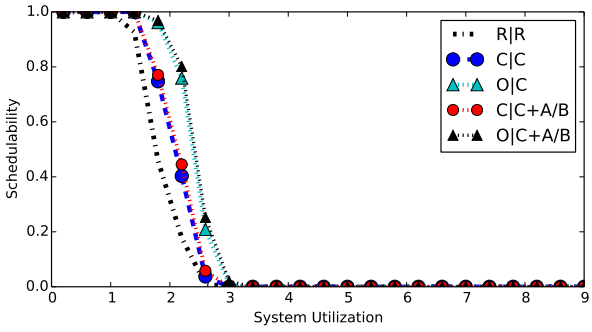
C-Light, Long, Med., Heavy, Small, Low, Many



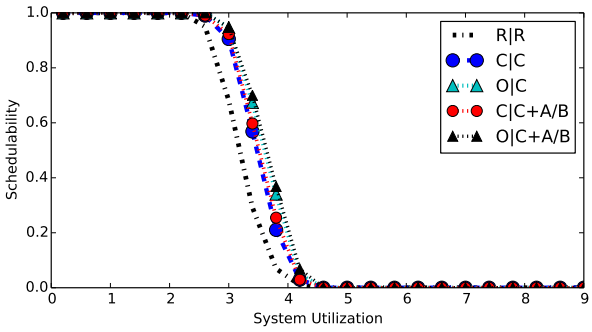
C-Heavy, Long, Light, Light, Large, Med., Many



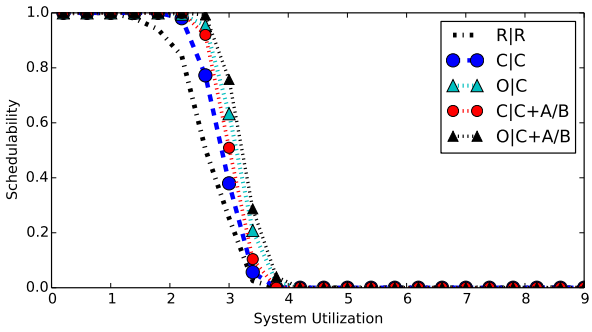
C-Heavy, Mod., Heavy, Light, Small, High, Few



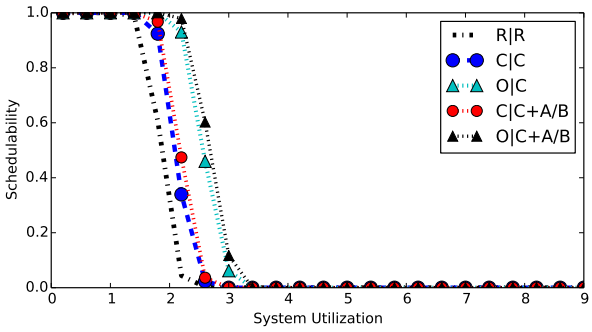
C-Heavy, Short, Light, Heavy, Large, Med., Few



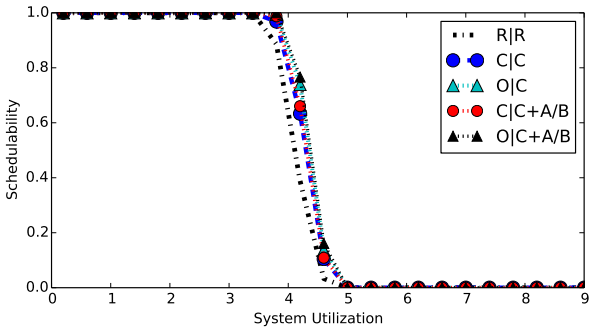
C-Heavy, Long, Light, Light, Small, Med., Many



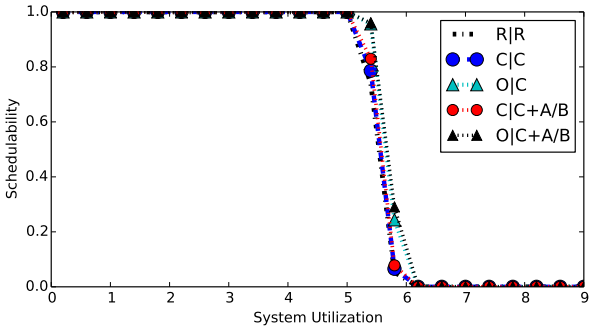
C-Light, Mod., Heavy, Heavy, Large, Med., Few



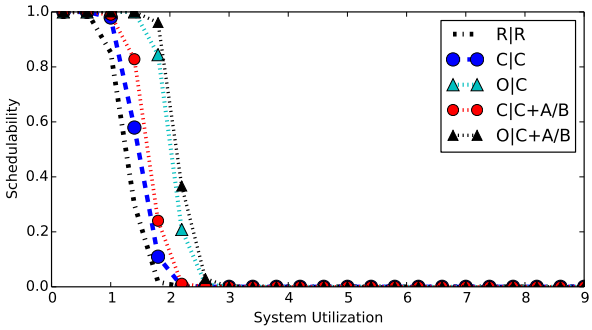
C-Light, Long, Light, Heavy, Large, High, Few



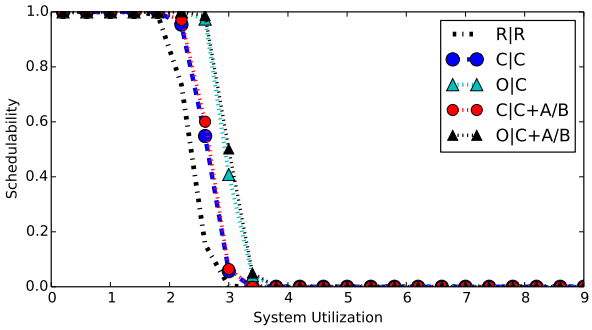
C-Light, Long, Heavy, Heavy, Small, Low, Few



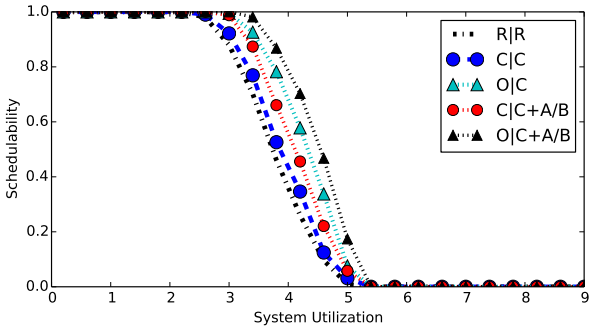
C-Heavy, Long, Med., Heavy, Small, Low, Many



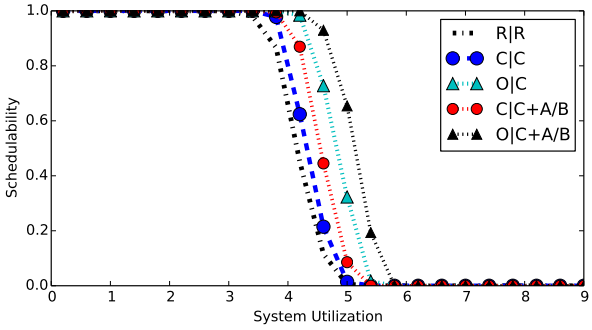
All-Mod., Short, Light, Light, Large, High, Many



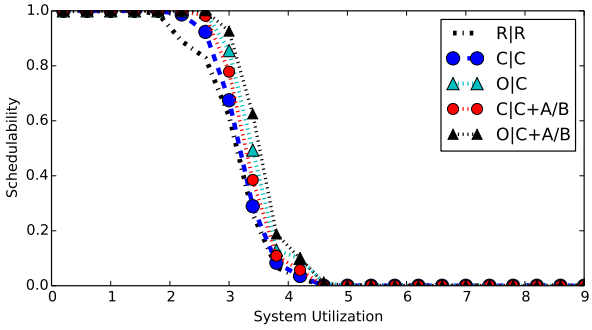
All-Mod., Long, Light, Light, Large, Med., Few



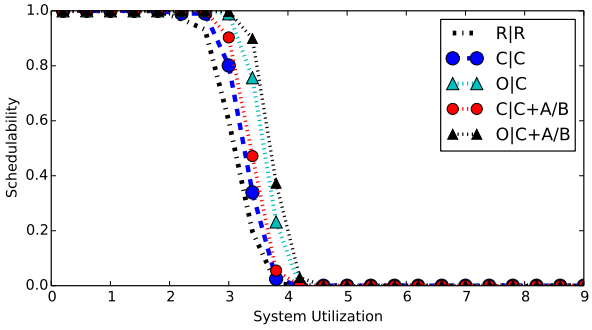
C-Heavy, Mod., Heavy, Light, Small, High, Many



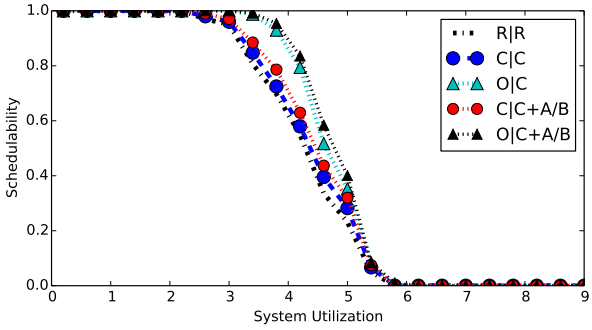
All-Mod., Short, Med., Light, Small, High, Many



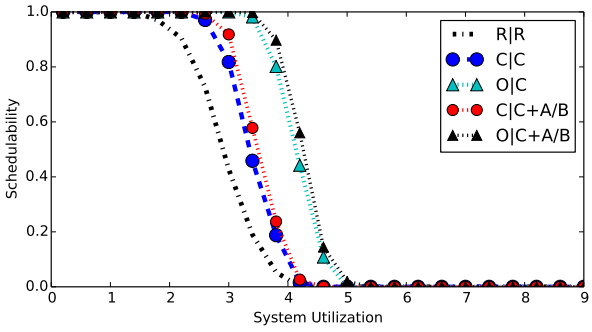
C-Heavy, Mod., Med., Heavy, Large, Med., Few



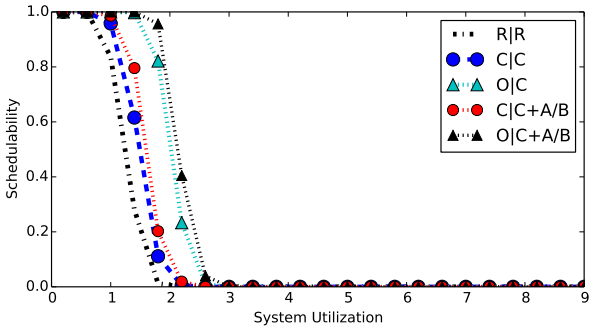
C-Light, Mod., Heavy, Heavy, Small, Med., Many



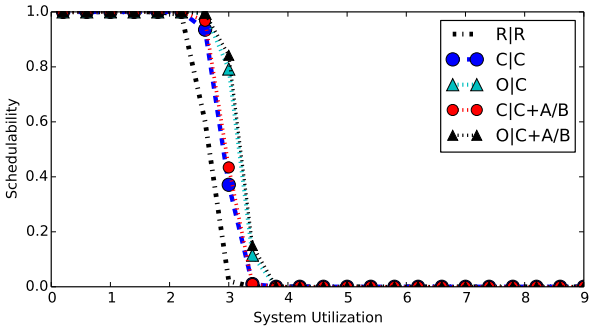
C-Heavy, Short, Heavy, Heavy, Small, Low, Many



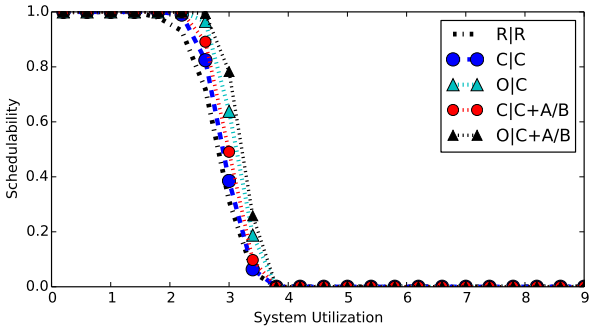
C-Light, Short, Med., Light, Large, Low, Few



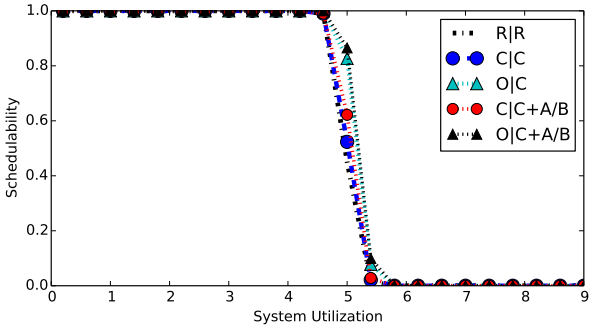
All-Mod., Short, Light, Light, Large, High, Few



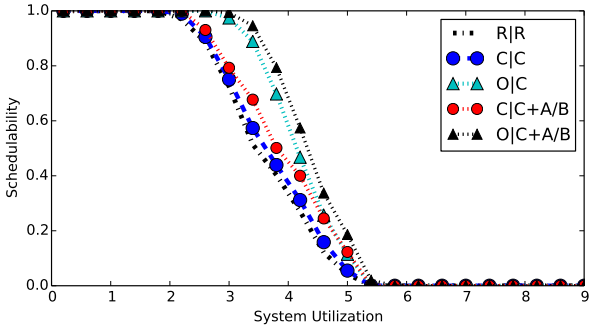
C-Light, Long, Light, Light, Small, Med., Few



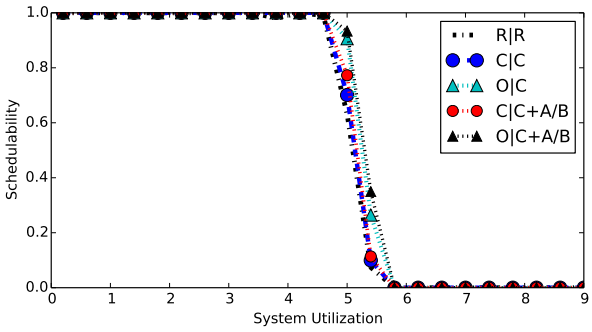
All-Mod., Mod., Heavy, Heavy, Large, Low, Few



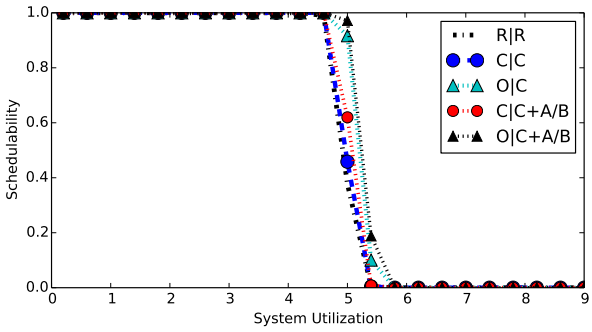
C-Heavy, Long, Med., Light, Large, High, Few



C-Heavy, Mod., Heavy, Heavy, Small, Med., Many



C-Heavy, Long, Med., Heavy, Large, High, Many



C-Heavy, Short, Med., Light, Small, Low, Few