Robust Real-Time Multiprocessor Interrupt Handling Motivated by GPUs

Glenn A. Elliott and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

***Abstract*—Architectures in which multicore chips are augmented with graphics processing units (GPUs) have great potential in many domains in which computationally intensive real-time workloads must be supported. However, unlike standard CPUs, GPUs are treated as I/O devices and require the use of interrupts to facilitate communication with CPUs. Given their disruptive nature, interrupts must be dealt with carefully in real-time systems. With GPU-driven interrupts, such disruptiveness is further compounded by the closed-source nature of GPU drivers. In this paper, such problems are considered and a solution is presented in the form of an extension to LITMUS^{RT} called klmirqd. The design of klmirqd targets systems with multiple CPUs and GPUs. In such settings, interrupt-related issues arise that have not been previously addressed.**

I. Introduction

Graphics processing units (GPUs) are capable of performing parallel computations at rates orders of magnitude greater than traditional CPUs. Driven both by this and by increased GPU programmability and single-precision floating-point support, the use of GPUs to solve non-graphical (general purpose) computational problems began gaining wide-spread popularity about ten years ago [1], [2], [3]. However, at that time, non-graphical algorithms had to be mapped to graphics-specific languages. GPU manufactures realized they could reach new markets by supporting general purpose computations on GPUs (GPGPU) and released flexible language extensions and runtime environments.[1] Since the release of these second-generation GPGPU technologies, both graphics hardware and runtime environments have grown in generality, enabling GPGPU across many domains. Today, GPUs can be found integrated on-chip in mobile devices and laptops [4], [5], [6], as discrete cards in higher-end consumer computers and workstations, and within many of the world's fastest supercomputers [7].

GPUs have applications in many real-time domains. For example, GPUs can efficiently perform multidimensional FFTs and convolutions, as used in signal processing, as well as matrix operations such as factorization on large data sets. Such operations are used in medical imaging and video processing, where real-time constraints are common. A particularly compelling use case is driver-assisted and autonomous automobiles, where multiple streams of video and sensor data must be processed and correlated in real time [8]. GPUs are well suited for this purpose.

[1]Notable platforms include the Compute Unified Device Architecture (CUDA) from Nvidia, Stream from AMD/ATI, OpenCL from Apple and the Khronos Group, and DirectCompute from Microsoft.

Prior Work. GPUs have received serious consideration in the real-time community only recently. Both theoretical work ([9], [10]) on partitioning and scheduling algorithms and applied work ([11], [12], [13]) on quality-of-service techniques and improved responsiveness has been done. Outside the real-time community, others have proposed operating system designs where GPUs are scheduled in much the same way as CPUs [14].

In our own work, we have investigated the challenges faced when augmenting multicore platforms with GPUs that have non-real-time, throughput-oriented, closed-source device drivers [15]. These drivers exhibit problematic behaviors for real-time systems. For example, tasks non-preemptively execute on a GPU in first-in-first-out order.[2] Also, GPU access is arbitrated by non-real-time spinlocks that lack priority-inheritance mechanisms. CPU time lost to spinning can be significant: GPU accesses commonly take tens of milliseconds up to several seconds [15]. Further, blocked tasks may experience unbounded priority inversions due to the lack of priority inheritance.[3]

The primary solution we presented in [15] to address these issues is to treat a GPU as a shared resource, protected by a real-time suspension-based semaphore. This removes the GPU driver from resource arbitration decisions and enables bounds on blocking time to be determined. We validated this approach in experiments on LITMUS^{RT} [16], UNC's real-time extension to Linux, and demonstrated that our methods reduced both CPU utilization and deadline tardiness.

Contributions. One issue not addressed in our prior work is the effect GPU interrupts have on real-time execution. Interrupts cause complications in the design and analysis of real-time systems. Ideally, interrupt handling should respect the priorities of executing real-time tasks. However, this is a non-trivial issue, especially for systems with shared I/O resources. In this paper, we examine the effects interrupt servicing techniques have on real-time execution on a multiprocessor, with GPU-related interrupts particularly in mind.

Our major contributions are threefold. First, we develop techniques that enable interrupts due to asynchronous I/O to be handled without violating the single-

[2]Newer GPUs allow some degree of concurrency, at the expense of introducing non-determinism due to conflicts within co-scheduled work. Further, execution remains non-preemptive in any case.

[3]A *priority inversion* occurs when a task has sufficient priority to be scheduled but it is not. Priority inversions can be introduced by a variety of sources, including locking protocols and interrupt services. These inversions must be bounded to ensure real-time guarantees.

threaded sporadic task model, improving schedulability analysis. Prior interrupt-related work has not directly addressed asynchronous I/O on multiprocessors. Second, we propose a technique to override the interrupt processing of closed-source drivers and apply this technique to a GPU driver. This required significant challenges to be overcome to alter the interrupt handling of the closed-source GPU driver. Third, we discuss an implementation of the proposed techniques and present an associated experimental evaluation. This implementation is given in the form of an extension to LITMUS^{RT} called klmirqd.

The rest of this paper is organized as follows. In Sec. II, we provide necessary background. In Sec. III, we review prior work on real-time interrupt handling and describe our solution, klmirqd. In Sec. IV, we show how GPU interrupt processing can be intercepted and rerouted, despite the use of a closed-source GPU driver. In Secs. V–VII, we present an experimental evaluation of klmirqd. We conclude in Sec. VIII.

Due to space limitations, we henceforth limit attention to GPU technologies from the manufacture NVIDIA, whose CUDA [17] platform is widely accepted as the leading GPGPU solution.

II. Interrupt Handling

An interrupt is a hardware signal issued from a system device to a system CPU. Upon receipt of an interrupt, a CPU halts its currently-executing task and invokes an interrupt handler, which is a segment of code responsible for taking the appropriate actions to process the interrupt. Each device driver registers a set of driver-specific interrupt handlers for all interrupts its associated device may raise. An interrupted task can only resume execution after the interrupt handler has completed.

Interrupts require careful implementation and analysis in real-time systems. In uniprocessor and partitioned multiprocessor systems, an interrupt handler can be modeled as the highest-priority real-time task [18], [19], though the unpredictable nature of interrupts in some applications may require conservative analysis. Such approaches can be extended to multiprocessor systems where tasks may migrate between CPUs [20]. However, in such systems, the subtle difference between an interruption and preemption creates an additional concern: an interrupted task cannot migrate to another CPU since the interrupt handler temporarily uses the interrupted task's program stack. As a result, conservative analysis must also be used when accounting for interrupts in these systems too. A real-time system, both in analysis and in practice, benefits greatly by minimizing interruption durations. Split interrupt handling is a common way of achieving this, even in non-real-time systems.

Under *split interrupt handling*, an interrupt handler performs the minimum amount of processing necessary to ensure proper functioning of hardware; additional work to be carried out in response to an interrupt is deferred. This deferred work may then be scheduled in a separate thread of execution with an appropriate priority. The duration of interruption is minimized and deferred work competes fairly with other tasks for CPU time.

GPU interrupt management is important to real-time systems for two reasons. First, we want to minimize the interference of GPU interrupts on all system tasks, including those that do not use a GPU. Second, we want a system that can be modeled analytically, so that schedulability can be assessed. Such analysis could be useful for systems such as driver-assisted automobiles. For example, response time bounds of GPU-based sensor processing may be required by an automatic collision avoidance component since response time equates directly to distance-travelled in a moving vehicle. Through analysis, we can determine the minimum system provisioning that meets the requirements of the collision avoidance component, as well as any other timing requirements of other components.

Interrupt Handling In Linux. We now review how Linux performs split interrupt handling. We focus on Linux for two reasons. First, despite its general-purpose origins, variants of Linux are widely used in supporting real-time workloads. Second, GPGPU is well supported on Linux and it is the *only* OS where we can make use of GPGPU and have the ability to modify OS source code. Other operating systems with reasonably robust GPGPU support (Windows and Mac OS X) are closed-source. Virtualization techniques that enable GPGPU in guest operating systems (ex. [21]) cannot be used because the GPGPU software, including the GPU driver, must still be hosted in a traditional OS environment, such as Linux.

During the initialization of the Linux kernel, device drivers (even closed-source ones) register interrupt handlers with the kernel's interrupt services layer, mapping interrupt signals to *interrupt service routines* (ISRs). Upon receipt of an interrupt on a CPU, Linux immediately invokes the registered ISR. The ISR is the *top-half* of the split interrupt handler. If an interrupt requires additional processing beyond what can be implemented in a minimal top-half, a deferrable *bottom-half* may be issued to the Linux kernel in the form of a *softirq*. There are several types of softirqs, but in this paper, we consider only *tasklets*, which are the type of softirq used by most I/O devices, including GPUs; we use the terms "softirq" and "tasklet" synonymously.

The Linux kernel executes tasklets using a heuristic. Immediately after executing a top-half, but before resuming execution of the interrupted task, the kernel executes up to ten tasklets. Any remaining tasklets are dispatched to one of several (per-CPU) kernel threads dedicated to tasklet processing; these are the "ksoftirq" daemons. The ksoftirq daemons are scheduled with high priority, but are preemptible. *The described heuristic can introduce long interrupt latencies, causing one to wonder if this can even be considered a split interrupt system.* In all likelihood, in a system experiencing few interrupts (though it may still be heavily utilized), for every top-

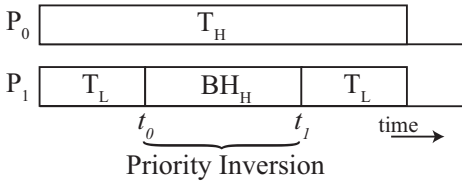

Figure 1. Priority inversion: T_L should be scheduled on processor P_1 at time t_0 since it is one of the top two highest-priority tasks.

half that yields a tasklet (bottom-half), that tasklet will subsequently be executed before the interrupted task is restored to the CPU. If a tasklet is deferred to a ksoftirq daemon, it is generally not possible to analytically bound the length of the deferral since these daemons are not scheduled with real-time priorities.

The PREEMPT_RT Linux kernel patch addresses this issue by using real-time schedulable worker threads to process all tasklets. Ideally, the scheduling priority of a worker thread should match that of the client task using the interrupt-raising device. However, these threads have a single fixed priority, even if an associated device is shared by multiple client tasks of differing priorities. This can easily lead to harmful priority inversions, as demonstrated in Sec. VI.

Priority inversions may also arise when asynchronous I/O is used. In asynchronous I/O, a task may issue a batch of I/O requests while continuing on to other processing. The task defers receipt of I/O results to a later time. This technique helps improve overall performance and is commonly used in GPU applications to mask bus latencies. Since synchronization with the I/O device is deferred, it is possible for interrupts to be received, and the corresponding bottom-halves to be executed, while a client task is scheduled. In such a case, the client task essentially becomes temporarily multithreaded, *breaking the assumption of single-threaded execution common in real-time task models such as the sporadic model.* A co-scheduled bottom-half can be interpreted as causing a priority inversion. This is illustrated in Fig. 1 for a two-processor system with tasks T_H and T_L, where T_H has higher priority. At time t_0, a bottom-half for T_H, BH_H, preempts T_L and is co-scheduled with T_H. Under a single-threaded task model, T_L should be scheduled at t_0, so a priority inversion occurs.

Priority inversions caused by asynchronous I/O in non-partitioned multiprocessors are not merely limited to Linux variants. Most methods in the real-time literature also have these shortcomings, and only a few techniques can avoid inversions in special cases.[4] To our knowledge, priority inversions caused by asynchronous I/O in non-partitioned multiprocessors have not been directly addressed in the real-time literature.

Neither standard Linux interrupt handling (SLIH) nor the PREEMPT_RT method implement split interrupt handling in a way amenable to real-time schedulability analysis. This is especially unfortunate since Linux-based systems are currently the *only* reasonable option for developing GPU-enabled real-time systems. In the next section, we propose a Linux-based solution that is amenable to analysis.

III. Interrupt Handling in LITMUS^{RT}

LITMUS^{RT}, a real-time extension of Linux, has been under continual development at UNC for over five years. To date, LITMUS^{RT} has largely been limited to workloads that are not I/O intensive, since LITMUS^{RT} has provided no mechanisms for real-time I/O. The implementation of real-time I/O is a considerable effort, and proper implementation of split interrupt handling is one critical aspect of this work. We begin this work here.

As discussed in Sec. II, current Linux-based operating systems use fixed-priority softirq daemons. In this paper, we introduce a new class of LITMUS^{RT}-aware daemons called klmirqd.[5] This name is an abbreviation for "Litmus softirq daemon" and is prefixed with a "k" to indicate that the daemon executes in kernel space. klmirqd daemons may function under any LITMUS^{RT}-supported job-level static-priority (JLSP) scheduling algorithm, including partitioned-, clustered-, and global-earliest-deadline-first and -fixed-priority schedulers.

klmirqd is designed to be extensible. Unlike the ksoftirq daemons, the system designer may create an arbitrary number of klmirqd threads to process tasklets from a single device, or a single klmirqd thread may be shared among many devices. A klmirqd thread may be configured in either a *dependent* or *independent* mode. In dependent mode, a klmirqd thread executes on behalf of a tasklet *owner* (the real-time client task using the interrupt-raising device), and in independent mode, a klmirqd thread runs as a bandwidth server. The latter mode is useful for constraining the utilization of *anonymous* tasklets (those with no owners), which is common to network traffic [22], [23].

Instead of using the standard Linux `tasklet_schedule()` function call to issue a tasklet to the kernel, an alternative function `litmus_tasklet_schedule()` is provided to issue a tasklet to klmirqd. *Owner* and *klmirqd thread identifier* parameters must be supplied by the caller of `litmus_tasklet_schedule()`, to dispatch work to a dependent-mode klmirqd thread. No owner parameter is needed for independent mode. Idle klmirqd threads suspend, waiting for tasklets to process. Dependent-mode klmirqd threads adopt the scheduling priority, including any inherited priority, of the tasklet owner when it executes. The LITMUS^{RT} scheduler

[4] These priority inversions may be avoided when bottom-halves are scheduled exclusively by bandwidth servers ([22], [23]), provided that real-time tasks are not dependent upon the completion of these bottom-halves. This is because bottom-halves are scheduled with a dedicated server's priority, not that of a task using the interrupt-raising device, and tasks in this case never block waiting for a bottom-halves to complete. Unique priorities among servers and tasks ensure that co-scheduling does not result in inversions. Further, inversions due to blocking are avoided since tasks never block.

[5] Source code available at http://www.litmus-rt.org.

	Owner-Based Bottom-Half Scheduling	Bandwidth Server Support	Threaded Interrupts	Non-Partitioned Multiprocessor Schedulers	Async I/O Without Inversions	JSLP Support
Linux (SLIH)				X		
PREEMPT_RT			X	X		
LynxOS [24]	X		X	X		
Steinberg et al. [25], [26]	X		X			
Lewandowski et al. [23]		X	X		X	
Manica et al. [22]		X	X		X	X
Zhang et al. (PAI) [27]	X					
klmirqd	X	X	X	X	X	X

Table I
SUMMARY OF SYSTEM INTERRUPT FEATURES, COMPARING KLMIRQD AGAINST NOTABLE PRIOR WORK.

also ensures that a dependent-mode klmirqd thread is *never co-scheduled* with its tasklet owner. This allows asynchronous I/O to be supported *without violating the single-threaded task models* commonly assumed.

klmirqd may be used for all system tasklets, both owned and anonymous. Unfortunately, applying klmirqd to all tasklets in LITMUS^{RT} (and by extension, Linux) is a very significant task. As such, we limit our focus to GPU applications in this paper.

Comparisons to prior work. We recognize that similar architectures for split interrupt handling have been proposed and implemented before. However, each approach does not support a full array of desired features.

Priority inheritance mechanisms for threaded interrupt handling have been used in LynxOS [24], the L4 microkernel [25], and the NOVA microhypervisor [26]. Bandwidth server techniques have also been used in Linux-based solutions [23], [22]. With the exception of LynxOS, all of these methods were originally developed for uniprocessor or partitioned multiprocessor platforms, and only [22] supports earliest-deadline-first scheduling. The use of partitioned solutions may constrain the allocation of shared resources, such as GPUs, between partitions. Schedulability analysis can also become overly pessimistic when these methods are extended to non-partitioned JLSP-scheduled multiprocessors.

The benefits of threaded interrupt handling comes at the cost of additional thread context-switch overheads. To address these concerns, Zhang et al. [27] developed a "process-aware interrupt" (PAI) method where arriving tasklets that do not have sufficient priority to be immediately scheduled are deferred, but not executed by dedicated interrupt threads. This is accomplished in the system's thread context switch code path. Prior to a context switch, the priority of the highest-priority deferred tasklet is compared against that of the next thread to be scheduled on the processor. The context switch is skipped if the tasklet has greater priority, and the tasklet is scheduled instead. The tasklet temporarily uses the program stack of the task that was scheduled prior to the aborted context switch. The resumption of this task can be delayed since it may not be rescheduled until the tasklet has completed, so the risk of priority inversions is not completely avoided. As shown in Sec. VII, this turns out to be a major analytical liability under non-partitioned multiprocessor scheduling. We have implemented a multiprocessor variant of Zhang's method that supports JLSP scheduling and compare it against klmirqd in later sections.[6]

A comparative summary of real-time interrupt handling alternatives is given in Table I.[7] klmirqd supports the greatest range of features.

IV. GPU INTEGRATION

In order to integrate GPU interrupt handling with klmirqd, we must first decide whether GPU klmirqd threads should run in dependent or independent mode. In the case of GPUs, the source device and ownership of every GPU tasklet can be determined by leveraging mechanisms already in place for real-time GPU management with additional reverse engineering of the closed-source GPU driver. Thus, we use dependent-mode klmirqd threads to avoid delays caused by budget exhaustion and analytical utilization loss due to budgetary overprovisioning, which can be introduced by bandwidth servers under independent mode.

In order to use klmirqd in dependent mode, for each tasklet we must identify: (1) the tasklet owner and (2) a target klmirqd thread to execute the tasklet. While an open source device driver could be modified to provide these parameters, how shall we accomplish this with a closed-source GPU driver that cannot be modified to call `litmus_tasklet_schedule()`? We addressed this issue by focusing separately on tasklet interception, device identification, owner identification, and dispatch. The approach taken to integrate GPUs into klmirqd is summarized in Fig. 2.

Tasklet Interception. The closed-source GPU driver must interface with the open source Linux kernel. We exploit this fact to intercept tasklets dispatched by the driver. This is done by modifying the standard internal Linux `tasklet_schedule()` function.

When `tasklet_schedule()` is called by a kernel component, the callback entry point of the deferred work is specified by a function pointer. We identify a tasklet as belonging to the closed-source GPU driver if this function pointer points to a memory region allocated to the

[6] PAI was originally designed for uniprocessor systems.

[7] The scheduling of interrupts can also be addressed orthogonally at the hardware level [28] and may be used to complement these software-based approaches.

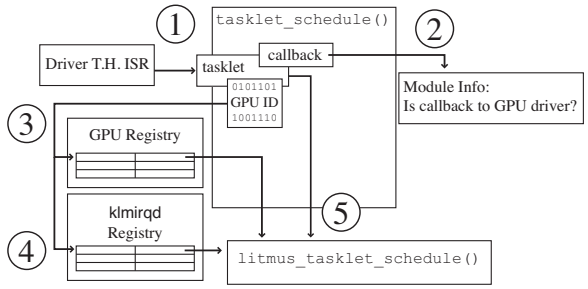

Figure 2. GPU tasklet redirection. **(1)** A tasklet from the GPU driver is passed to `tasklet_schedule()`. **(2)** The tasklet is intercepted if the callback points to the driver. **(3)** The GPU identifier is extracted from the data attached to the tasklet using a known address offset, and the GPU owner is found. **(4)** The GPU identifier is mapped to a klmirqd thread, and **(5)** the GPU tasklet is dispatched through `litmus_tasklet_schedule()`.

driver. Luckily, it is possible to make this determination since the driver is loaded as a module (or kernel plugin). We inspect every callback function pointer of every dispatched tasklet, online, using Linux's module-related routines.[8] Thus, we alter to `tasklet_schedule()` to intercept tasklets from the GPU driver and override their scheduling. It should be possible to use this technique to schedule tasklets of *any* closed-source driver in Linux, not just those from GPUs.

Device Identification. Merely intercepting GPU tasklets is not enough if a system has multiple GPUs; we must also identify which GPU raised the initial interrupt in order to determine tasklet ownership. While it may be possible to perform this identification process at the lowest levels of interrupt handling, we opt for a simpler solution closer to tasklet scheduling. The GPU driver attaches a memory reference to each tasklet, providing input parameters for the tasklet callback. This reference points to a data block that includes a device identifier indicating which GPU raised the interrupt. However, locating this identifier within the data block is challenging since it is packaged in a driver-specific format. Fortunately, the driver's links into the open source OS code allow us to locate the device identifier.

NVIDIA does not distribute its GPU driver as an entirely as a precompiled binary because the internal APIs of Linux change frequently and many users use custom configurations. To enable support for a changing kernel in varied configurations, the distribution includes plain source code for an OS/driver layer that bridges the kernel interfaces with closed-source precompiled object files. By visually inspecting this bridge code, we gained insight into the format of the tasklet data block, and determined the fixed address offset of the device identifier.

We can now intercept and identify the source of a GPU tasklet. What remains is to identify the tasklet owner and dispatch the tasklet to the appropriate klmirqd thread.

[8]This may sound like a costly operation, but it is actually quite a low-overhead process, as is shown in Sec. VI.

Owner Identification. As mentioned in Sec. I, we may arbitrate GPU access using a real-time suspension-based semaphore, thus preventing the GPU driver from exhibiting behaviors detrimental to real-time predictability [15]. Whenever a GPU is allocated to a task by the locking protocol, an internal lookup table, called the *GPU ownership registry*, indexed by device identifier, is updated to record device ownership.

To manage a pool of k GPUs, we may use a real-time k-exclusion protocol to assign any available GPU to a GPU-requesting task.[9] The arbitration protocol considered herein is a k-exclusion extension of the *flexible multiprocessor locking protocol* (FMLP) [29], which we call the k-FMLP.[10] The k-FMLP is particularly attractive because worst-case wait times scale inversely with the number of GPUs. The k-FMLP was implemented in LITMUS^{RT} to support this work. Special consideration had to be paid to integrate with klmirqd. Specifically, since the k-FMLP uses priority inheritance, a priority inherited by a GPU holder must be propagated immediately to any klmirqd thread executing on its behalf.

With the device identifier extracted from the tasklet and device registry table generated by the locking protocol, determining the current owner of a GPU is straightforward. We now have gathered all required information to dispatch a GPU tasklet to klmirqd; now we must determine which klmirqd thread will perform the processing.

klmirqd Dispatch. The architecture of klmirqd is general enough to support any number of daemon instances, all scheduled by a JLSP real-time scheduler. We create a single klmirqd thread per GPU to ensure that all GPUs can be used simultaneously. Each thread is assigned to a specific GPU, and the assignment is recorded in the *klmirqd assignment registry* for later lookup.

V. Evaluation of Priority Inversions

In this and the next two sections, we present a runtime evaluation of klmirqd. We first focus attention on the severity of priority inversions arising from various bottom-half scheduling methods. We compare the threaded interrupt handling of klmirqd to both Zhang et al.'s PAI [27] deferred bottom-half scheduler, as well as the heuristic-driven method of the standard Linux interrupt handler (SLIH).

Evaluation Platform. The platform used in all of our experiments is a dual-socket six-cores-per-socket Intel Xeon X5060 CPU system running at 2.67GHz that is equipped with eight NVIDIA GTX-470 GPUs. The platform has a NUMA architecture of two NUMA nodes, each with six CPU cores and four GPUs apiece.

[9]k-exclusion locking protocols can be used to arbitrate access to pools of similar or identical resources, such as communication channels or I/O buffers. k-exclusion extends ordinary mutual exclusion (mutex) by allowing up to k tasks to simultaneously hold locks (thus, mutual exclusion is equivalent to 1-exclusion).

[10] A full description of the k-FMLP is available in Appendix A. A detailed discussion of some issues that arise when constructing a real-time k-exclusion protocol can be found in [30].

In all of our experiments, we used a clustered scheduler, with GPUs statically assigned to clusters, and GPU access arbitrated by a separate k-FMLP instance within each cluster. Clustered window-constrained schedulers, such as clustered earliest-deadline-first (C-EDF), have been shown to be effective if bounded deadline tardiness is the real-time requirement of interest [31]. For this reason, we consider only C-EDF in this section since bounded deadline tardiness applies to many common real-time GPU applications [15]. Clustering is split along the NUMA architecture of the system, yielding two clusters. This configuration minimizes bus contention, given the memory and I/O bus architectures of the system. This is especially important for the I/O bus since contention can significantly affect data transmission rates between CPUs and GPUs. We used CUDA 4.0 for our GPU runtime environment.

Experimental Setup. LITMUS^{RT}, based upon Linux 2.6.36, was used as the testbed operating system for this evaluation since it enables fair comparisons among interrupt handling methods. We did not include unmodified Linux or PREEMPT_RT for this reason since they have fundamentally different scheduler architectures (though higher-level comparisons to these are made in Sec. VI).

We assessed the severity of priority inversions by generating sporadic task sets and executing them in LITMUS^{RT}. Each generated task set included both CPU-only and CPU-and-GPU-using (hereafter referred to as GPU-using) tasks. Individual task parameters were randomly generated as follows. The period of every task was randomly selected from the range $[15ms, 60ms]$; such a range is common for multimedia processing and sensor feeds such as video cameras. The utilization of each task was generated from an exponential distribution with mean 0.5 (tasks with utilizations greater than 1.0 were regenerated). This yields relatively large average per-task execution times.[11] We expect GPU-using tasks to have such execution times since current GPUs typically cannot efficiently process short GPU requests due to I/O bus latencies. Next, between 20% and 30% of tasks within each task set were selected as GPU-using tasks. Each GPU-using task had a GPU critical section length equal to 80% of its execution time. Of the critical section length, 20% was allocated to transmitting data to and from a GPU. This distribution of critical section length and data transmission time is common to many GPU applications, including FFTs and convolutions [15]. Finally, each task set was partitioned across the two clusters using a two-pass worst-fit partitioning algorithm that first assigns GPU-using tasks to clusters, followed by CPU-only tasks. This tends to evenly distribute GPU-using tasks between clusters. We generated task sets with high utilizations to put the system under heavy load, increasing the likelihood of priority inversions. Task set utilizations ranged from 7.5 to 11.5, in increments of 0.1, for a total of 41 task sets.

[11] A GPU-using task's execution time includes time spent executing on both CPUs and GPUs.

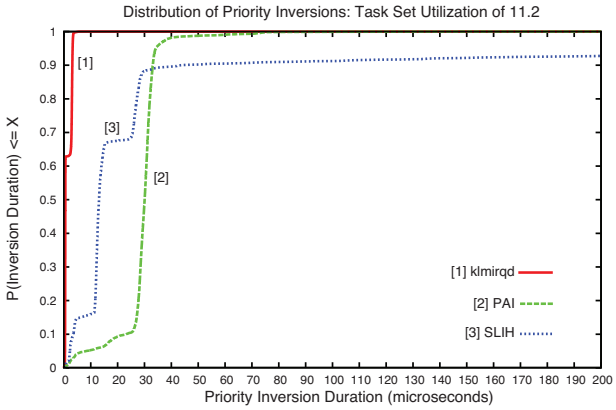

Figure 3. Cumulative distribution of priority inversion durations.

Tasks executed numerical code (on both CPUs and GPUs) for the configured durations. GPUs were accessed via asynchronous I/O. Every task set was executed once in LITMUS^{RT} for two minutes under each interrupt handling method. Scheduling logs were recorded, from which we compared the behaviors of each method.

Metrics. Ideally, a system should conform to the sporadic task model and not suffer any priority inversions. However, this is difficult to achieve in a real system. We assessed deviance from the ideal by: **(i)** determining the *distribution of priority inversion durations* and **(ii)** computing *cumulative priority inversions*.

Results. While priority inversions cannot be totally eliminated, they should nevertheless be as short as possible. Fig. 3 shows a representative example of the cumulative distribution function of priority inversion length induced by scheduled tasklets under the three interrupt handling methods for the task set with utilization 11.2.[12] As seen, a typical priority inversion is much shorter under klmirqd than under PAI or SLIH. For example, 90% of inversions under klmirqd are shorter than $5\mu s$, whereas the 90th percentile is roughly $35\mu s$ under PAI and exceeds $40\mu s$ under SLIH. The cumulative distribution function for SLIH also has a very long tail, extending out to $16ms$ (not depicted in Fig. 3 due to scale). While the performance of SLIH is not entirely surprising, the performance of PAI is, since PAI schedules bottom-halves by priority. However, if a bottom-half has the priority to be scheduled, then it is likely that the owner of the bottom-half, making use of asynchronous I/O, also has sufficient priority to be scheduled. In such cases, both the bottom-half and owner are co-scheduled, resulting in an inversion on a non-partitioned multiprocessor.

Although priority inversions should be as short as possible, the cumulative duration of inversions is also important because a system that suffers many short inversions may still be disrupted by their cumulative effect. Fig. 4 shows cumulative priority inversion length as a function of maximum priority inversion length for

[12] Graphs for all tested task sets are available in Appendix C.

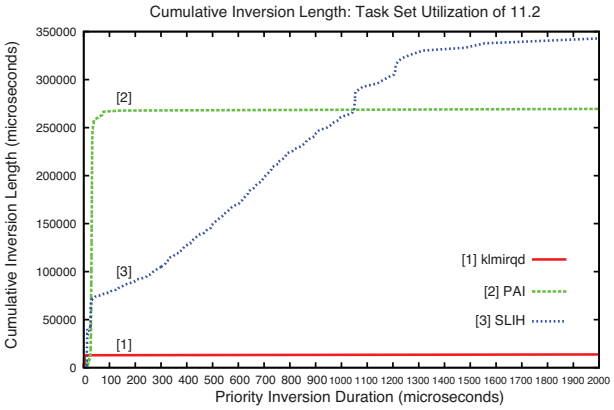

Figure 4. Cumulative priority inversion length as a function of maximum priority inversion length for the task set with utilization 11.2. The total duration of priority inversion is much less under klmirqd in comparison to PAI and SLIH. The curves for klmirqd and PAI plateau abruptly, indicating little variance in priority inversion durations.

the same task set shown in Fig. 3. Observe in Fig. 4 that the sum duration of priority inversions is roughly 12,000μs under klmirqd, about 260,000μs under PAI, and approximately 350,000μs under SLIH. The long tail distribution under SLIH in Fig. 3 is also exhibited here: the curve for SLIH in Fig. 4 increases slowly, while the curves for klmirqd and PAI abruptly plateau. The durations of priority inversions under klmirqd and PAI have little variance in contrast to SLIH. Minimized variance is desirable for predictable systems as it reduces jitter in response times.

While we may never be able to completely bound priority inversions caused by the *closed-source* GPU driver, the observed deterministic behaviors exhibited by klmirqd (in particular) and PAI are promising. It is clear that heuristic-driven SLIH performs poorly in comparison. In addition to offering better performance, klmirqd and PAI are also amenable to real-time analysis. The poor performance of SLIH is not unexpected, but it is important to note since a GPU-enabled real-time system can only feasibly be supported on Linux platforms today.

VI. System-Wide Evaluation of Interrupt Handling Methods

In this section, we examine system-wide effects of interrupt handling in terms under LITMUS^{RT}, using SLIH, PAI, and klmirqd methods; PREEMPT_RT; and unmodified Linux (which also uses SLIH).

Experimental Setup. To demonstrate the real-time weaknesses in unmodified Linux and PREEMPT_RT, and evaluate both klmirqd and PAI, we executed a workload of CPU-only and GPU-using tasks on the platform described in Sec. V. In order to fairly compare LITMUS^{RT} against unmodified Linux and PREEMPT_RT, the workload was scheduled using the clustered rate-monotonic (C-RM) algorithm since unmodified Linux and PREEMPT_RT only support fixed real-time priorities. Counting semaphores were used to protect each pool of GPU resources in unmodified Linux and PREEMPT_RT, similarly to how the k-FMLP is used under LITMUS^{RT}. The workload consisted of 50 tasks: two GPU-using tasks that consume $2ms$ of CPU time and $1ms$ of GPU time with a period of $19.9ms$; 40 CPU-only tasks that consume $5ms$ of CPU time with a period of $20ms$; and finally, eight GPU-using tasks that consume $2ms$ of CPU time and $1ms$ of GPU time with a period of $20.1ms$. The set of tasks was evenly partitioned between the two clusters. Unique priorities were assigned to each task within each cluster according to task period.

This task set represents a pathological case for fixed-priority threaded interrupt handling. Here, the highest and lowest priority tasks share GPUs and the interrupt handling threads, which each have a single fixed priority. Unrelated CPU-only tasks are sandwiched between these GPU-using tasks. If all tasks had equal priority, then under RM scheduling, priorities could be reassigned such that CPU-only tasks have priorities strictly greater or strictly less than those of GPU-using tasks. However, though task periods are close to being equal, it is not the case here.

The workload was executed on eight platform configurations: (1) Unmodified Linux, using SLIH, to provide a baseline of performance; (2) PREEMPT_RT, with GPU-interrupt priorities set below that of any other real-time task; (3) PREEMPT_RT with GPU-interrupt priorities greater than the greatest GPU-using task;[13] (4) LITMUS^{RT} using SLIH; (5) LITMUS^{RT} with klmirqd; (6) LITMUS^{RT} with PAI; and finally, under C-EDF (for the sake of comparison), both (7) LITMUS^{RT} with klmirqd, and (8) LITMUS^{RT} with PAI. The unmodified Linux and LITMUS^{RT} configurations were based upon Linux kernel version 2.6.36. PREEMPT_RT was based upon the 2.6.33 Linux kernel, which was the most recent kernel supported by PREEMPT_RT at the time of our evaluation. This workload was executed 25 times for each system configuration for a duration of 60 seconds each. Measurements were recorded consistently on each platform.

Results. Table II gives the average percentage of deadlines missed, as well as average response times (as percent of period), for CPU-only and GPU-using tasks under the various platform scenarios. The percentage of deadlines missed is useful for comparing schedulability. Response time measurements express the timeliness of job completions (or severity of a deadline misses).

A deadline miss occurs if a job does not complete within one period of its release time. We avoid penalizing the response time of a subsequent job following a missed deadline by shifting the job's release point to coincide with the tardy completion time of the prior job. However, since these tests execute for a fixed duration, frequently tardy tasks may not execute all their jobs within the allotted time; any jobs that have not completed (even

[13]This is a rational choice when an interrupt-generating device is shared by several tasks with differing priorities.

Scheduler:	C-RM						C-EDF	
Operating System:	PREEMPT_RT		Unmod. Linux	LITMUS^{RT}				
Interrupt Handling Method:	Low Prio. Interrupts (a)	High Prio. Interrupts (b)	SLIH (c)	SLIH (d)	klmirqd (e)	PAI (f)	klmirqd (g)	PAI (h)
Avg. % Miss Per Task								
CPU-Only Tasks	12.5%	12.5%	1.6%	10.0%	10.0%	9.9%	0%	0%
GPU-Using Tasks	10.1%	8.5%	6.8%	0%	0%	0%	0%	0%
Avg. Resp. Time as % Period								
CPU-Only Tasks	22474.5%	24061.0%	8992.1%	55.8%	55.8%	55.6%	55.4%	55.4%
GPU-Using Tasks	23066.1%	34263.5%	61131.7%	46.7%	49.6%	46.2%	46.2%	46.2%

Table II
AVERAGE NUMBER OF DEADLINE MISSES PER TASK AND AVERAGE JOB RESPONSE TIMES (EXPRESSED AS A PERCENTAGE OF PERIOD) FOR THE EXPERIMENTAL TASK SET EXECUTED UNDER SEVERAL SCHEDULING ALGORITHMS AND INTERRUPT HANDLING METHODS. THE EXPERIMENTAL TASK SET WAS EXECUTED 25 TIMES PER CATEGORY.

those not yet released) by the end of a test are considered to have missed deadlines, though these jobs are not included in response time measurements.

Observation 1. *There are no good options for selecting a* ***fixed*** *priority for interrupt threads shared by tasks of differing priorities.* The increase of GPU interrupt priority in column (b) causes all bottom-half thread execution to preempt CPU-only jobs, directly increasing their response times with respect to column (a), where interrupts have the lowest priority. In most cases under column (b), GPU interrupt execution is on behalf of lower-priority GPU-using jobs, thus causing CPU-only jobs to experience priority inversions. Priority inversions also occur if interrupt priority is too low, resulting in the starvation of GPU-using jobs. This is evident in column (a), where GPU-using tasks miss deadlines more often than in column (b).

Observation 2. *Unmodified Linux outperforms PREEMPT_RT* (*in this pathological case*) *due to lower interrupt handling overhead.* Under unmodified Linux, bottom-halves are usually executed immediately after top-halves; thus, bottom-halves essentially execute with maximum priority (like column (b)), yet this is accomplished without the overhead of threaded interrupt handling. Increased CPU availability greatly improves the response times of CPU-only jobs in column (c) in comparison to both columns (a) and (b), while deadline misses of GPU-using jobs are reduced.

Observation 3. *klmirqd dynamically assigns priorities to interrupt threads, resulting in schedulable and analyzable real-time systems.* The average response time values in columns (d) and (e) indicate that jobs typically complete well before their deadlines (they are less than 100%). While the response time of GPU-using tasks is slightly worse in column (e), the heuristic-driven nature of SLIH is not amenable to schedulability analysis.

Observation 4. *Overheads introduced by klmirqd into LITMUS^{RT} are largely negligible, in comparison to both SLIH and PAI.* The nearly-equal response times in columns (d), (e), and (f) indicate that klmirqd overhead costs are negligible. This indicates that techniques like those of PAI are unnecessary in the case of GPU interrupts. However, PAI does perform slightly better than klmirqd in most cases, though it suffers from significant analytical pessimism, as shown next in Sec. VII.

Observation 5. *LITMUS^{RT} with klmirqd or PAI outperforms PREEMPT_RT.* A comparison of column (b) to columns (e) and (f) shows that deadline misses were not significant under klmirqd or PAI, but were common under PREEMPT_RT. Unfortunately, it is difficult to identify a single difference between PREEMPT_RT and LITMUS^{RT} that causes this disparity in performance, as there are many core differences (in scheduler implementation, etc.) between the two. Additional investigation is merited. Nevertheless, these differences do not have bearing on the previously made observations.

Observation 6. *C-EDF scheduling is superior to C-RM in limiting deadline tardiness.* This is not surprising, in light of prior work [32], but we mention it nonetheless. This is another indication that PREEMPT_RT may not be a desirable solution in all applications, especially in soft real-time systems, since C-EDF is not supported.

VII. OVERHEAD-AWARE SCHEDULABILITY

As seen in Secs. V and VI, both klmirqd and PAI reduce the frequency and duration of priority inversions. klmirqd accomplishes this through some additional scheduling overheads. PAI, on the other hand, incurs lower system overheads due to the lack of scheduling, but does so through the sharing of program stacks with bottom-half execution. SLIH usually executes bottom-halves immediately following the top-half, incurring overheads less than even PAI. How do these trade-offs affect general task set schedulability? The answer to this question depends upon the actual system overheads and worst-case priority inversions that may occur in each approach.[14] To address this question in the context of C-EDF, we conducted schedulability experiments using a methodology similar to that proposed in [33] to integrate actual measured overheads.

Using the same hardware platform described in Sec. V, we measured the following system overheads: thread context switching, scheduling, job release queuing, inter-processor interrupt latency, CPU clock tick processing, both GPU interrupt top-half and bottom-half processing, and, in the case of klmirqd and PAI, tasklet

[14]We assume for analysis that SLIH *always* executes bottom-halves immediately following the top-halves.

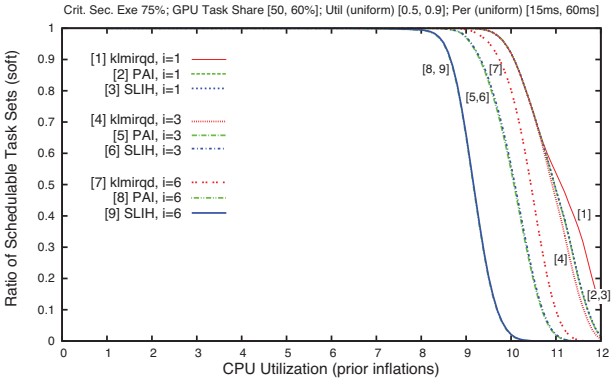

Figure 5. The percentage of schedulable task systems (y-axis) as a function of CPU utilization (x-axis) under klmirqd (threaded) interrupt handling, PAI (non-threaded) handling, and SLIH (non-threaded) interrupt handling.

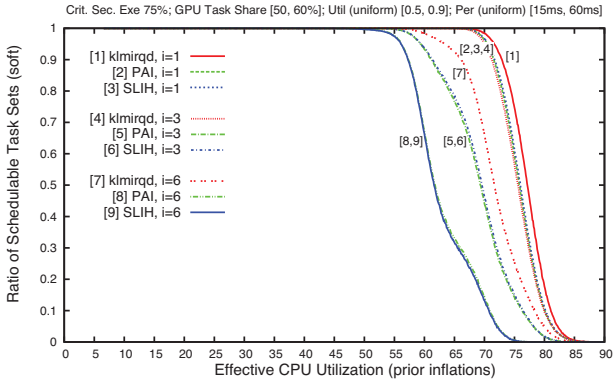

Figure 6. Schedulability results similar to Fig. 5 except that a GPU speedup of 16× is assumed.

release queuing. We then randomly generated task sets with properties similar to those in Sec. V. Schedulability of each generated task set was determined by using a soft real-time (bounded tardiness) schedulability test for C-EDF scheduling [34], augmented to account for overheads. Average overhead values were used (as in [33] when analyzing soft real-time systems). Interrupts were accounted for using task-centric methods [20].[15]

A selection of our schedulability results is given in Fig. 5,[16] which presents results for task sets in which: per-task utilizations vary uniformly over $[0.5, 0.9]$; GPU-using tasks use 75% of their execution time on the GPU; and 50% to 60% of tasks in each task set are GPU-using. Variance in GPU behavior was controlled by a parameter $i \in \{1, 3, 6\}$, which specifies the number of GPU interrupts each GPU-using job may generate. In Fig. 5, schedulability is higher under klmirqd than both PAI and SLIH for each choice of i, with a greater disparity between them for larger values of i.

Contrary to what we might expect from the results in Secs. V and VI, PAI performs worse than SLIH. This is because, while PAI attempts to schedule bottom-halves to avoid priority inversions, it does so at the expense of using the program stacks of other tasks. Under PAI, a bottom-half can preempt the lowest-priority scheduled job when the bottom-half arrives. However, on a non-partitioned multiprocessor, the relative priority of the preempted job may increase with respect to other jobs before the bottom-half completes (this may occur when higher-priority jobs on other processors complete). Unfortunately, the preempted job cannot resume execution until the preempting bottom-half completes, freeing up the task's program stack. The job is blocked, and this must be reflected in analysis. Accounting for bottom-halves under PAI results in formulas matching those for SLIH, except that PAI must also include additional overheads due to bottom-half scheduling.

[15]Please see Appendix B for theoretical analysis, which includes a detailed description of interrupt and overhead accounting methods.

[16]Additional graphs are available in Appendix D.

It is important to note that this scenario for PAI cannot occur on uniprocessors or partitioned multiprocessors since the preempted task never has a priority great enough to be scheduled before the preempting bottom-half completes.[17] Like the issues raised by co-scheduling bottom-halves and their owner tasks, this PAI limitation reinforces the notion that prior solutions for real-time interrupt handling may need reconsideration on multiprocessor platforms.

The primary motivation for utilizing GPUs in a real-time system is increased performance. The benefits of threaded GPU interrupt handling are even more clear when *effective utilization* is considered instead of actual CPU utilization. By supposing a GPU-to-CPU speed-up ratio of R, we may convert each GPU-using task into a functionally equivalent CPU-only task by viewing each time unit spent executing on a GPU as R times units spent executing on a CPU. We define effective utilization to be the utilization of a task set after such a conversion. Fig. 6 depicts the same schedulability results shown in Fig. 5, except that effective utilizations are considered, for the case $R = 16$ (i.e., a GPU is 16× faster than a CPU), a common speed-up.

As Fig. 6 shows, the impact of using klmirqd is even greater if effective utilizations are considered. When $i = 6$, 90% of task sets with an effective utilization of 65.0 CPUs are schedulable under klmirqd. In contrast, effective utilization must be decreased to 55.0 CPUs to achieve the same degree of schedulability under SLIH or PAI. klmirqd supports effective utilizations ten CPUs greater at 90% schedulability!

VIII. Conclusion

In this paper, we presented flexible real-time interrupt-handling techniques for multiprocessor platforms that are applicable to any JLSP-scheduler and that respect single-threaded task execution. We also reported on our efforts in implementing such techniques in LITMUS^{RT}, and showed that they can be successfully applied to even a closed-source GPU driver, thus allowing for improved real-time characteristics for real-time systems

[17]Barring priority inheritance mechanisms from locking protocols.

using GPUs. We presented an experimental evaluation of this implementation that shows that it reduces the interference caused by GPU interrupts in comparison to standard interrupt handling in Linux, outperforms fixed-priority interrupt handling methods, is competitive with other real-time methods, and offers better results in terms of overall schedulability (with overheads considered).

This paper lays the groundwork for future investigations into GPU-enabled real-time platforms. We limited attention to clustered scheduling in this paper. In a future study, we intend to consider a full gamut of partitioned, clustered, and global schedulers and different GPU-to-CPU assignment methods and GPU arbitration (i.e., locking) protocols. The goal of this study will be to identify the best combinations of scheduler, locking protocol, etc., for both soft and hard real-time systems, from the perspective of overhead-aware schedulability.

ACKNOWLEDGMENT

Work supported by NSF grants CNS 1016954 and CNS 1115284; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

REFERENCES

[1] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," in *SIGGRAPH '03*, 2003.

[2] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra, "Simulation of cloud dynamics on graphics hardware," in *SIGGRAPH '03*, 2003.

[3] J. Krüger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," in *SIGGRAPH '03*, 2003.

[4] AMD Fusion Family of APUs. [Online]. Available: http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf

[5] Intel details 2011 processor features, offers stunning visuals build-in. [Online]. Available: http://download.intel.com/newsroom/kits/idf/2010_fall/pdfs/Day1_IDF_SNB_Factsheet.pdf

[6] Bringing high-end graphics to handheld devices. [Online]. Available: http://www.nvidia.com/object/IO_90715.html

[7] V. Kindratenko and P. Trancoso, "Trends in high-performance computing," *Computing in Science Engineering*, vol. 13, no. 3, 2011.

[8] S. Thrun. (2010) GPU technology conference keynote, day 3. [Online]. Available: http://livesmooth.istreamplanet.com/nvidia100923/

[9] B. Andersson, G. Raravi, and K. Bletsas, "Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors," in *31st RTSS*, 2010.

[10] G. Raravi, B. Andersson, and K. Bletsas, "Provably good scheduling of sporadic tasks with resource sharing on two-type heterogeneous multiprocessor platform," in *15th OPODIS*, 2011.

[11] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Resource sharing in GPU-accelerated windowing systems," in *17th RTAS*, 2011.

[12] ——, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *USENIX Annual Technical Conference*, 2011.

[13] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "RGEM: A responsive GPGPU execution model for runtime engines," in *32nd RTSS*, 2011.

[14] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: operating system abstractions to manage GPUs as compute devices," in *23rd SOSP*, 2011.

[15] G. Elliott and J. Anderson, "Globally scheduled real-time multiprocessor systems with GPUs," *Real-Time Systems*, vol. 48, 2012.

[16] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *27th RTSS*, 2006.

[17] CUDA Zone. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[18] J. Liu, *Real-Time Systems*. Prentice Hall, 2000.

[19] K. Jeffay and D. Stone, "Accounting for interrupt handling costs in dynamic priority task systems," in *14th RTSS*, 1993.

[20] B. Brandenburg, H. Leontyev, and J. Anderson, "An overview of interrupt accounting techniques for multiprocessor real-time systems," *Journal of Systems Architecture*, vol. 57, no. 6, 2010.

[21] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-accelerated virtual machines," in *3rd HPCVirt*, 2009.

[22] N. Manica, L. Abeni, L. Palopoli, D. Faggioli, and C. Scordino, "Schedulable device drivers: Implementation and experimental results," in *6th OSPERT*, 2010.

[23] M. Lewandowski, M. J. Stanovich, T. P. Baker, K. Gopalan, and A. Wang, "Modeling device driver effects in real-time schedulability analysis: Study of a network driver," in *13th RTASS*, 2007.

[24] Writing device drivers for LynxOS. [Online]. Available: http://www.lynuxworks.com/support/lynxos/docs/lynxos4.2/0732-00-los42_writing_device_drivers.pdf

[25] U. Steinberg, J. Wolter, and H. Härtig, "Fast component interaction for real-time systems," in *17th ECRTS*, 2005.

[26] U. Steinberg, A. Böttcher, and B. Kauer, "Timeslice donation in component-based systems," in *6th OSPERT*, 2010.

[27] Y. Zhang and R. West, "Process-aware interrupt scheduling and accounting," in *27th RTSS*, 2006.

[28] L. E. L. del Foyo, P. Mejia-Alvarez, and D. de Niz, "Predictable interrupt management for real time kernels over conventional pc hardware," in *12th RTAS*, 2006.

[29] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, "A flexible real-time locking protocol for multiprocessors," in *13th ECRTS*, 2007.

[30] G. Elliott and J. Anderson, "An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems," in *19th RTNS*, 2011.

[31] A. Bastoni, B. Brandenburg, and J. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor real-time schedulers," in *31st RTSS*, 2010.

[32] U. Devi, "Soft real-time scheduling on multiprocessors," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2006.

[33] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2011.

[34] J. Erickson, N. Guan, and S. Baruah, "Tardiness bounds for global EDF with deadlines different from periods," in *14th OPODIS*, 2010.

[35] B. Brandenburg and J. Anderson, "Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks," in *11th EMSOFT*, 2011.

APPENDIX

A. k-FMLP

Inspired by the *Flexible Multiprocessor Locking Protocol* (FMLP) [29], the *k-Exclusion Flexible Multiprocessor Locking Protocol* (k-FMLP) is a simple and easy to understand protocol. The blocking experienced by a job waiting for a resource protected by the k-FMLP is $O(n/k)$ where n is the number of tasks using the lock and k is the size of the protected resource pool. Unlike the more recent *O(m) Locking Protocol* (OMLP) the k-FMLP is not optimal with respect to the number of m CPU processors. Achieving an $O(m/k)$ k-exclusion protocol (one might expect a $1/k$ term in any reasonable k-exclusion protocol) is actually a non-trivial [35], [30]. However, for the sake of the interrupt-handling focus of this paper, the k-FMLP will suffice. The k-FMLP is designed as follows.

Task Model. We consider the problem of scheduling a mixed task set of n sporadic tasks, $T = \{T_1, \ldots, T_n\}$, on m CPUs with one pool of k resources. A subset $T^R \subset T$ of the tasks require use of one of the system's k resources. We assume $k \leq m$. A *job* is a recurrent invocation of work by a task, T_i, and is denoted by $J_{i,j}$ where j indicates the j^{th} job of T_i (we may omit the subscript j if the particular job invocation is inconsequential). Each *task* T_i is described by the tuple $T_i(e_i, l_i, d_i, p_i)$. The *worst-case CPU execution time* of T_i, e_i, bounds the amount of CPU processing time a job of T_i must receive before completing. The *critical section length* of T_i, l_i, denotes the length of time task T_i holds one of the k resources. For tasks $T_i \notin T^R$, $l_i = 0$. The *deadline*, d_i, is the time after which a job is released by when that job must complete. Arbitrary deadlines are supported by the k-FMLP. The *period* of T_i, p_i, measures the minimum separation time between job invocations for task T_i.

We say that a job J_i is *pending* from the time of its release to the time it completes. A pending job J_i is *ready* if it may be scheduled for execution. Conversely, if J_i is not ready, then it is *suspend*.

A job $J_{i,j}$ (of a task $T_i \in T^R$) may issue a resource request $R_{i,j}$ for one of the k resources. Requests that have been allocated a resource (*resource holders*) are denoted by H_x, where x is the index of the particular resource (of the k) that has been allocated. Requests that have not yet been allocated a resource are *pending* requests. We assume that a job requests a resource at most once, though the analysis presented in this paper can be generalized to support multiple, non-nested, requests. We let b_i denote an upper bound on the duration a job may be blocked.

In the k-FMLP, a job J_i suspends if it issues a request R_i that cannot be immediately satisfied. We use *Priority inheritance* as a mechanism to bound worst-case blocking time. Under priority inheritance, a resource holder may temporarily assume the higher-priority of a blocked job that is waiting for the held resource. The priority of a job J_i in the absence of priority inheritance is the *base priority* of J_i. We call the priority with which J_i is scheduled the *effective priority* of J_i.

Structure. The k-FMLP uses k FIFO request queues, each queue assigned to one of the k protected resources. A job J_i enqueues a resource request R_i onto the queue FIFO_x when the job requires a resource. A job with a request at the head of its queue is considered the holder of the associated resource and is ready to run.

Rules. We define the *worst-case wait time* for a queue, FIFO_x, at time t with the formula $wait_x(t) = \sum_{R_j \in \text{FIFO}_x} l_j$. However, it may be too burdensome for an implementation to maintain critical section length values for each queued request. We may instead use the conservative approximation $wait_x(t) = \max\{l\} \cdot |\text{FIFO}_x|$, where $\max\{l\}$ is the longest duration any job may hold a resource and $|\text{FIFO}_x|$ denotes the number of requests in FIFO_x. We call an implementation which is informed of critical section lengths to be l-aware and those that are not as l-oblivious.

The k-FMLP is governed by the following rules.

R1 When J_i issues a request R_i at time t, R_i is appended to the queue with the minimum worst-case wait time, $\min_{1 \leq x \leq k}\{wait_x(t)\}$. J_i is granted ownership of the xth resource when R_i is at the head of FIFO_x.

R2 All jobs with queued request are suspended except for the resource holders, which are ready. H_x inherits the priority of the highest-priority blocked job in FIFO_x if that priority exceeds the base priority of H_x.

R3 When J_i frees resource x, R_i is dequeued from FIFO_x and the job with the next queued request in FIFO_x is granted the newly available resource. If there is no pending job in FIFO_x, then the highest-priority blocked job waiting for one of the k resources is "stolen" (removed from its queue) and granted the free resource.[18]

Blocking Analysis. The k-FMLP essentially load-balances resource requests amongst the k resources, similar to how patrons at a grocery store may organize themselves at several checkout stations, selecting the line they anticipate to have the shortest wait time.

Lemma 1. *When J_i issues a request for a resource at time t, there exists a resource queue with a worst-case wait time less than or equal to $\left(\sum_{x=1}^{k} wait_x(t)\right)/k$.*

[18]Request "stealing" does not affect worst-case blocking analysis, but is a useful to ensure an efficient, work-conserving, system.

Proof: Suppose all other resource-using jobs currently hold a resource or have a queued request issued before R_i. Therefore, all requests other than R_i have been partitioned into k groups. The sum total worst-case wait time at time t is $\sum_{x=1}^{k} wait_x(t)$. In the l-aware case, then the total worst-case wait time is equal to $\sum_{T_j \in T^R \setminus \{T_i\}} l_j$. Otherwise, it may be approximated by $\sum_{x=1}^{k} \max\{l\} \cdot |\text{FIFO}_x| = \max\{l\} \cdot \left|T_j \in T^R \setminus \{T_i\}\right|$, in the l-oblivious case.

On average, each queue has a worst-case wait time of $\left(\sum_{T_j \in T^R \setminus \{T_i\}} l_j\right)/k$ in the l-aware case, or $\left(\max\{l\} \cdot \left|T_j \in T^R \setminus \{T_i\}\right|\right)/k$ in the l-oblivious case. If one queue has a worst-case wait time greater than average, then another must have a worst-case wait time less than average. Thus, the average worst-case wait time upper-bounds the maximum wait time of the shortest queue at time t. ∎

Theorem 1. *The maximum time a job may be blocked under the k-FMLP in an l-aware implementation is bounded by the formula*

$$b_i \leq \left(\sum_{T_j \in T^R \setminus \{T_i\}} l_j \right) / k. \tag{1}$$

Proof: Follows from Lemma 1 and rule **R1**. ∎

Theorem 2. *The maximum time a job may be blocked under the k-FMLP in an l-oblivious implementation is bounded by the formula*

$$b_i \leq \left(\max\{l\} \cdot \left|T_j \in T^R \setminus \{T_i\}\right|\right) / k. \tag{2}$$

Proof: Follows from Lemma 1 and rule **R1**. ∎

Observe that both Eq. 1 and Eq. 2 are $O(n/k)$.

In the case of an l-aware implementation, we may derive an exact bound for b_i by determining the maximum worst-case wait time of the shortest FIFO queue. Unfortunately, the problem to compute an exact solution is a variant of the NP-hard (in the strong sense) job shop scheduling problem. Nevertheless, b_i may be computed exactly by solving the following 0-1 integer linear program for tasks in T^R:

$$\begin{aligned} &\textbf{Maximize} \quad b_i \\ &\textbf{subject to} \\ &\quad \textstyle\sum_{T_j \in T^R \setminus \{T_i\}} x_{j,q} \cdot l_q \geq b_i \quad \forall j \in \{1, \ldots, k\} \\ &\quad \textstyle\sum_{j=1}^{k} x_{j,q} = 1 \qquad\qquad\quad \forall T_q \in T^R \setminus \{T_i\} \\ &\quad x_{j,q} \in \{0, 1\} \end{aligned} \tag{3}$$

where $x_{j,q}$ is an indicator variable denoting the assignment of R_q to FIFO_j.

B. Overhead Accounting

In Sec. VII, we presented the results of overhead-aware schedulability tests. It was shown that threaded GPU interrupt processing in klmirqd is able to meet soft real-time bounded tardiness constraints for more task sets than standard Linux interrupt handling (SLIH), despite additional overheads associated with klmirqd's thread scheduling. Due to page constraints in Sec. VII, we were unable to present a detailed description of how overheads were accounted in our schedulability experiments. This is done here.

Traditional real-time schedulability tests usually model a theoretical system where scheduling decisions are instantaneous, with no execution overheads. However, overheads must be considered if we are to apply schedulability test methods to real systems in practice. The dissertation of B. Brandenburg [33] describes several methods for incorporating system overheads from various sources such as operating system ticks, timer interrupts, scheduling decisions, etc. into traditional schedulability tests, thus making them overhead-aware. These tests better reflect real-world performance than the traditional overhead-oblivious tests. One overhead accounting technique described in [33] is the "task-centric" method. We adapt the task-centric method in this work to account for interrupts caused by GPUs.

This section proceeds in six parts. First, we describe the schedulability test for bounded tardiness used in this paper and outline the general process of task execution inflation to account for overheads. Following, we update our schedulability test to account for basic overheads such as blocking due to locking protocols, self-suspensions, scheduling decisions, and context switches. Next, we further develop this model to liberally account for overheads of GPU interrupts under SLIH. Thereafter, we alter this model to account for overheads of GPU interrupts under klmirqd, as well as Zhang et al. [27]'s Process-Aware Interrupt (PAI) method. Then, we account for overheads due to operating system ticks. Finally, we present the values of the observed overheads from LITMUS$^{\text{RT}}$ running on our evaluation platform (platform described in Sec. V) used in our schedulability tests.

We express our overhead accounting methods using with the task model presented in Appx. A, with minor additions presented as needed.

Schedulability. The basic schedulability test for bounded tardiness in a soft real-time system scheduled under G-EDF is described in [32]. Under this test, two conditions must hold:

$$e_i \leq p_i \tag{4}$$

$$\sum_{T_i \in T} e_i / p_i \leq m \tag{5}$$

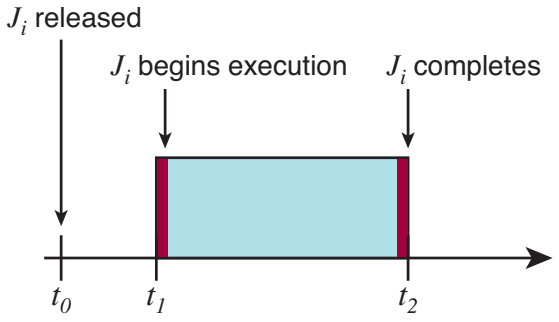

Figure 7. Basic accounting of overheads for simple CPU-only job, J_i. Job execution is framed by overheads to due scheduling decisions and context switches.

Condition Eq. (4) ensures that no individual task (or single CPU) is over utilized, and condition Eq. (5) ensures that the system as a whole (with m CPUs) is not over utilized. To test schedulability under C-EDF scheduling, we simply perform the G-EDF test on each cluster, scaling m accordingly to reflect the number of CPUs in each cluster.

To account for system overheads, such as scheduling decisions, we inflate e_i to include processing time for the appropriate operations. Accounting techniques, such as the task-centric method, determine what overheads should be charged against which tasks. Also, under suspension-oblivious analysis, we also inflate e_i to include suspension-based delays in execution. In this work, this includes the suspension of when a GPU-using job blocks waiting for an available GPU, as well as when a GPU-using job suspends while waiting for results from a GPU.

Fixed-point iterative schedulability tests must be used since the overhead accounting methods presented here depend upon the worst-case response time jobs. Fortunately, worst-case response times can be computed using bounded tardiness analysis [34]. However, this response time is likewise dependent upon the overheads under consideration. Thus, schedulability tests must be iteratively performed until tardiness bounds remain unchanged.

Basic Accounting. Let us begin accounting for basic overheads and suspension-based execution delays.

A system must make one scheduling decision for every job arrival and once again for every job completion. We account for scheduling decision overheads by charging every task the cost of two scheduling decisions. Similarly, the arrival of a job may trigger a preemption, causing a context switch. Another context switch occurs again when the preempting job completes, and the system switches back to the originally scheduled job. Thus, we charge each task for the cost of two context switches. Fig. 7 gives a basic depiction of execution of a CPU-only job where job execution is framed by scheduling decision and context switch costs. This gives the following basic inflation equation:[19]

$$e_i^{(\text{cpu-only})_6} = e_i + 2(\Delta^{sch} + \Delta^{cxs}), \tag{6}$$

where Δ^{sch} denotes the duration of a scheduling decision and Δ^{cxs} the time to perform a context switch.

We may also perform a basic accounting of overheads for GPU-using jobs. Unlike CPU-only jobs, GPU-using jobs must incur at least two suspensions in the worst-case. The first suspension occurs when a GPU-using job attempts to acquire a GPU when none are available; the job must suspend to wait for an available GPU. The next suspension occurs when the GPU-using job blocks to wait for the results from a GPU invocation. An additional suspension may be experienced for each additional use of the GPU by a job. Each suspension induces two additional scheduling decision overheads and context switch costs. Thus,

$$e_i^{(\text{gpu-using})_7} = e_i + (1 + 1 + \eta_i)(2(\Delta^{sch} + \Delta^{cxs})), \tag{7}$$

where η_i denotes the number of times the job J_i uses the GPU. Note that η_i also denotes the number of interrupts the GPU will send to the system to signal the completion of an invocation; this will be important later.

Due to our suspension-oblivious analysis, we must treat all durations of self-suspensions as additional execution time (i.e. CPU demand). To account for the suspension due to GPU acquisition, we must inflate e_i by b_i (Appx. A). To account for suspensions due to GPU use, we introduce a new term s_i. Let s_i denote the total time the GPU spends executing for J_i *as well as the execution time for any associated top-halves or bottom-halves*. Fig. 8 gives a depiction of a simple ($\eta_i = 1$) GPU-using job and associated overhead costs.

By simplifying Eq. (7) and incorporating b_i and s_i, we get:

$$e_i^{(\text{gpu-using})_8} = e_i + b_i + s_i + (2+\eta_i)(2(\Delta^{sch} + \Delta^{cxs})). \tag{8}$$

Release Overheads. In addition to scheduling and context switch overheads, there are also overheads associated with job releases. Consider a periodic task system where jobs are released by OS timers that trigger via timer interrupts. Suppose at time t_0 a timer interrupt is raised to release job J_i. In an ideal system, J_i would instantaneously appear in the ready queue of the appropriate CPU (or even immediately scheduled) at t_0. However, this is not the case in a real system, and there

[19] We will be progressively inflating execution costs and need a means of keeping track of our incremental steps. We use the super-script notation on e_i such that the super-script value matches the equation label where the inflated execution cost was defined.

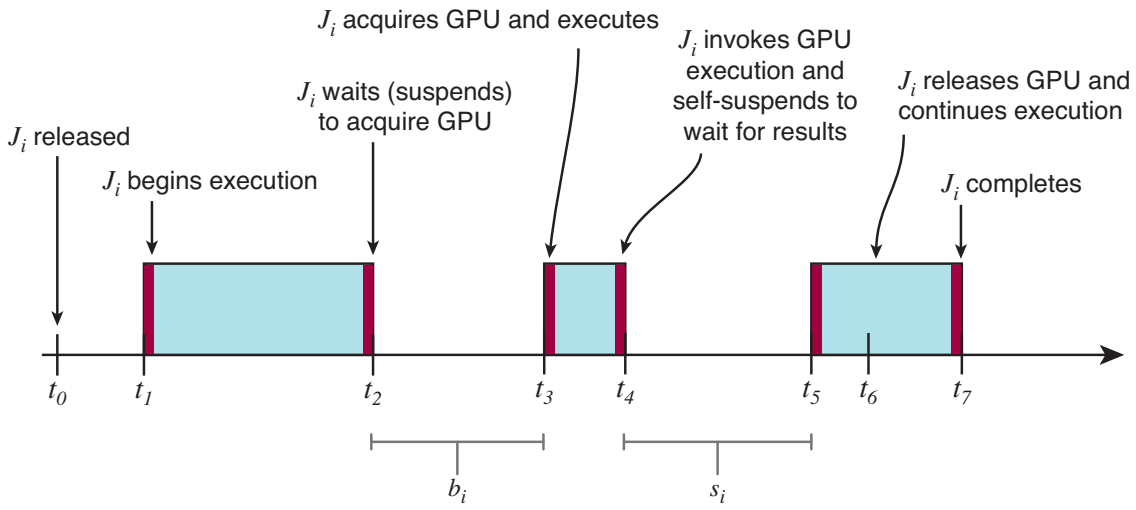

Figure 8. Basic accounting of overheads for simple GPU-using job, J_i. Suspensions incur additional scheduling and context switch overhead costs. Execution time must also be inflated by b_i and s_i under suspension-oblivious analysis.

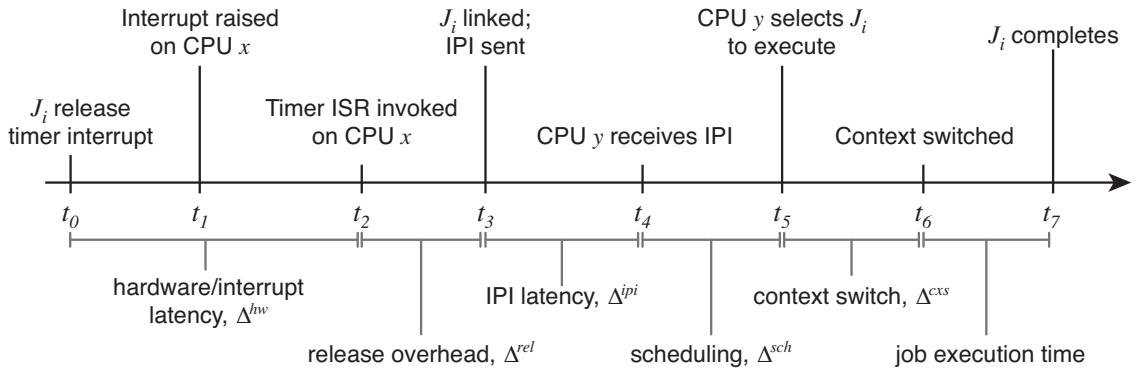

Figure 9. A newly released job experiences delays due to interrupt delivery latency and updates of ready queue data structures, in addition to scheduling and context switch overheads.

are delays which should be accounted for.

Fig. 9 depicts the event sequence of a job released by a timer. When the timer interrupt fires at time t_0, it is not received by CPU x until time t_1. It takes a moment for CPU x to respond to the interrupt, so the timer ISR is not invoked until time t_2. The interval $[t_0, t_2]$ may be modeled as delay due to hardware delays in interrupt delivery, denoted by Δ^{hw}. We ignore the effects of Δ^{hw} here since the goal of this paper is to compare the effects klmirqd interrupt handling and standard Linux interrupt handling have on schedulability. We presume Δ^{hw} affects both methods equally.

Continuing after t_2, the timer ISR must add J_i to the appropriate ready queue. Before the ready queue data structures can be updated, spinlocks must first be acquired to enforce safe concurrent access. Once these locks are acquired, J_i is added to the ready queues. These operations are not completed until time t_3. We denote the duration $[t_2, t_3]$ with Δ^{rel}.

It is possible that at time t_3, J_i should be immediately scheduled, but J_i should not run on CPU x according to the active scheduling algorithm. Instead, J_i must be scheduled on CPU y. To handle this case, CPU x updates data structures (*links*) at time t_3 to make J_i available to CPU y and then notifies CPU y of the need to schedule J_i using an inter-processor interrupt (IPI), which is received by CPU y at time t_4. The interval $[t_3, t_4]$ captures delays caused by IPI latencies and is

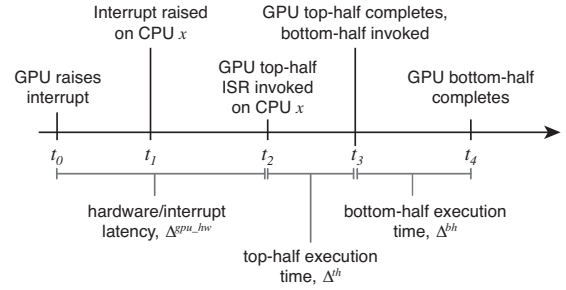

Figure 10. Interrupt handling overheads under standard Linux, with the liberal assumption that bottom-halves are executed immediately and not preceded by other bottom-halves from other sources or deferred to ksoftirq processing.

denoted by Δ^{ipi}.

Observe that Δ^{rel} and Δ^{ipi} overheads may delay execution whenever a job becomes ready to run, not just from timer-based releases, but also resumptions from self-suspensions. We must account for these overheads accordingly. CPU-only jobs only experience overheads Δ^{rel} and Δ^{ipi} once (per release), so

$$e_i^{(\text{cpu-only})_9} = e_i^{(\text{cpu-only})_6} + \Delta^{rel} + \Delta^{ipi}. \tag{9}$$

GPU-using jobs experience several self-suspensions, so

$$e_i^{(\text{gpu-using})_{10}} = e_i^{(\text{gpu-using})_8} + (2+\eta_i)(\Delta^{rel} + \Delta^{ipi}). \tag{10}$$

GPU Interrupts. We now present our accounting method for GPU interrupt overheads under SLIH, klmirqd, and PAI.

Standard Linux Interrupts. Fig. 10 illustrates the sequence of events from a GPU interrupt, raised by the GPU device, to the completion of the interrupt bottom-half under SLIH. At time t_0, the GPU raises an interrupt and the associated top-half commences execution at time t_2. This sequence is similar to the sequence already described for job release timer interrupts. Similarly, the interval $[t_0, t_2]$ models as delay due to hardware delays in GPU interrupt delivery; we denote this duration with the term Δ^{gpu_hw}. However, like Δ^{hw}, we also ignore Δ^{gpu_hw} for the same reasons.

At time t_2, the top-half of the GPU interrupt begins execution and completes at time t_3. This duration denoted by Δ^{th}. We make the liberal assumption that the GPU interrupt bottom-half begins execution immediately at time t_3 and completes at time t_4. This duration is denoted by Δ^{bh}. The response time of the interrupt handler, end-to-end, is $[t_0, t_4]$ (recall that the CPU cannot be preempted while it is processing the interrupt under standard Linux).

We say that this bound for response time is liberal for two important reasons: (1) it assumes no other bottom-halves for other system interrupts are queued ahead of the GPU interrupt bottom-half; and (2) it assumes the GPU interrupt bottom-half is never deferred to Linux's ksoftirqd daemon. *These are best-case assumptions for standard Linux interrupt handling.* If (1) fails to hold, then the job interrupted by the GPU interrupt is further delayed. If (2) fails to hold, then it is generally not possible to bound the response time of the bottom-half since ksoftirqd is not scheduled real-time priorities. If the response time of the bottom-half is not bounded, then the response time of the GPU-using job that depends upon the bottom-half's completion is also not bounded and no real-time guarantees for the job can be made!

Since the execution on the interval $[t_2, t_4] = \Delta^{th} + \Delta^{bh}$ is non-preemptive, GPU interrupt processing can induce a priority inversion if a higher-priority job is interrupted. This inversion is $(\Delta^{th} + \Delta^{bh})$ in length for every GPU interrupt in the worst-case. Under task-centric accounting when there is no CPU dedicated to interrupt handling (as is the case in this study), we cannot know which CPUs will be affected by GPU interrupts. As a result, we must model the processing for a single interrupt as occurring on *all* CPUs (within a given cluster) simultaneously (see [33] for a complete explanation). Furthermore, any job, both CPU-only and GPU-using, can be affected by these inversions.

To quantify the effect of GPU interrupts on job J_i, we must first determine the maximum number of GPU interrupts that may occur while J_i may be executing. Once the total number of interrupts is known, we multiply the number of interrupts by $(\Delta^{th} + \Delta^{bh})$ to make a total per-task accounting. The following formula computes the number of GPU interrupts that may, in the worst-case, delay job J_i in a soft real-time system with bounded tardiness (assuming n is the number of tasks within the cluster under consideration):

$$H_i = \sum_{i \neq j}^{n} \left(\left\lceil \frac{p_i + x_i + p_j + x_j}{p_j} \right\rceil \cdot \eta_j \right), \tag{11}$$

where x_i and x_j denote tardiness bounds, as computed by [34].

Since GPU interrupts affect both CPU-only and GPU-using jobs, we further inflate e_i with the same term according to the following formulas:

$$e_i^{(\text{cpu-only})_{12}} = e_i^{(\text{cpu-only})_9} + H_i(\Delta^{th} + \Delta^{bh}) \tag{12}$$

and

$$e_i^{(\text{gpu-using})_{13}} = e_i^{(\text{gpu-using})_{10}} + H_i(\Delta^{th} + \Delta^{bh}) \tag{13}$$

for CPU-only and GPU-using tasks, respectively.

klmirqd Interrupts. The use of klmirqd shortens the duration of non-preemptive execution of interrupt handling at the expense of additional thread-scheduling overheads.

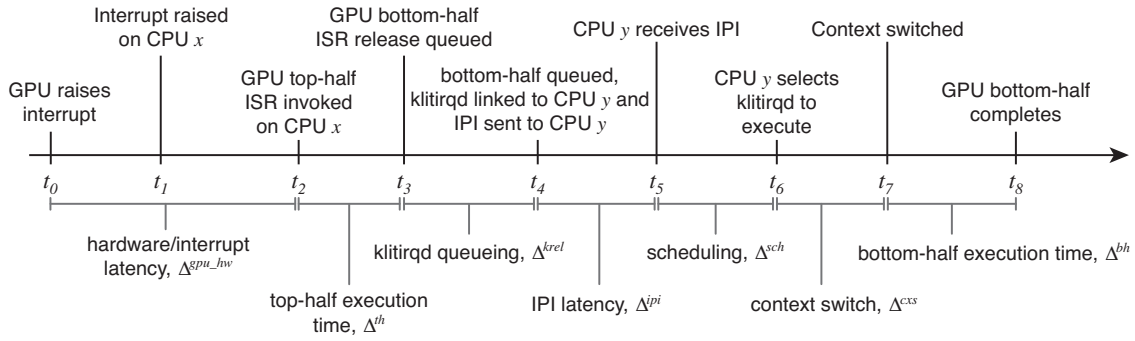

Figure 11. Interrupt handling overheads under klmirqd.

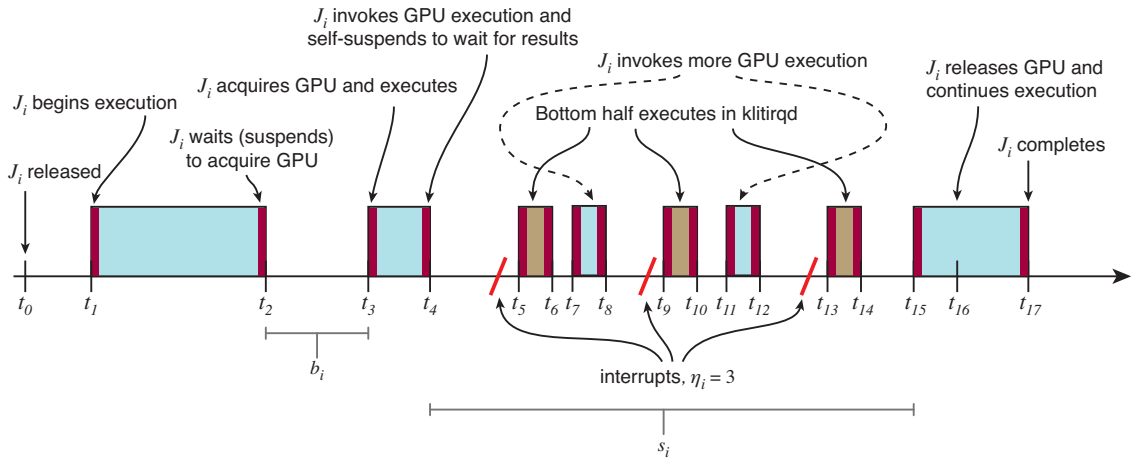

Figure 12. Execution of a GPU-using job under klmirqd. By definition, s_i already incorporates Δ^{bh}.

Fig. 11 illustrates the sequence of events from a GPU interrupt, raised by the GPU device, to the completion of the interrupt bottom-half by klmirqd. Observe in Fig. 11 that immediately after the completion of an interrupt top-half at time t_3, the bottom-half is "released" (or queued) to klmirqd. This operation completes at time t_4. Thus, the entire non-preemptive duration of interrupt handling lasts from $[t_2, t_4] = \Delta^{th} + \Delta^{krel}$. Thus, we can update Eqs. (12) and (13) to replace Δ^{bh} with Δ^{krel}. This gives us

$$\widehat{e}_i^{(\text{cpu-only})_{14}} = e_i^{(\text{cpu-only})_9} + H_i(\Delta^{th} + \Delta^{krel}) \qquad (14)$$

and

$$\widehat{e}_i^{(\text{gpu-using})_{15}} = e_i^{(\text{gpu-using})_{10}} + H_i(\Delta^{th} + \Delta^{krel}) \qquad (15)$$

for CPU-only and GPU-using tasks, respectively.

Eqs. (14) and (15) account for the effects of GPU interrupt top-half processing under klmirqd. However, we must still account for additional thread-scheduling overheads. These remaining overheads only affect GPU-using tasks, so Eq. (14) completes our accounting under klmirqd for CPU-only tasks (though tick interrupt accounting remains).

We must inflate every GPU-using job to include additional thread-scheduling costs. To schedule each bottom-half we charge costs for Δ^{sch}, Δ^{cxs}, and Δ^{ipi}. We isolate these charges to the single job that triggers them because the bottom-half is scheduled with the priority of the originating job. Charging GPU-using jobs for thread-scheduling overheads, we get the formula:

$$\widehat{e}_i^{(\text{gpu-using})_{16}} = \widehat{e}_i^{(\text{gpu-using})_{15}} + \eta_i(2(\Delta^{sch} + \Delta^{cxs}) + \Delta^{ipi}). \qquad (16)$$

Observe that we do not make a charge for Δ^{rel} because bottom-half releasing is already accounted for by Δ^{krel}. Furthermore, $\Delta^{krel} \ll \Delta^{rel}$ because klmirqd threads are already in a prepped idle state.

Due to suspension-oblivious analysis, we do not have to make any overhead charges for Δ^{bh} under klmirqd since s_i, by definition, already incorporates this overhead into execution time. This can be observed in Fig. 12.

This completes our accounting of GPU interrupt overheads under both standard Linux interrupt handling and klmirqd.

Process-Aware Interrupts.

The use of Zhang et al. [27]'s Process-Aware Interrupt (PAI) handling also shortens the duration of non-preemptive execution that occurs *immediately* after an interrupt is received. Interrupt accounting under PAI differs from both standard Linux and klmirqd interrupt handling methods.

Assuming a bottom-half does not have sufficient priority to be scheduled immediately after its top-half has completed, the bottom-half is enqueued into a bottom-half ready queue. Similar to the bottom-half "release" overhead Δ^{krel} in klmirqd, the overhead of bottom-half enqueueing under PAI is denoted by Δ^{pairel}. Δ^{pairel} differs from Δ^{krel} because while klmirqd operates on the process ready queues, PAI operates on a separate bottom-half-only ready queue. We account for Δ^{pairel} in the same fashion as we did Δ^{krel}, giving

$$\widetilde{e}_i^{(\text{cpu-only})_{17}} = e_i^{(\text{cpu-only})_9} + H_i(\Delta^{th} + \Delta^{pairel}) \quad (17)$$

and

$$\widetilde{e}_i^{(\text{gpu-using})_{18}} = e_i^{(\text{gpu-using})_{10}} + H_i(\Delta^{th} + \Delta^{pairel}) \quad (18)$$

for CPU-only and GPU-using tasks, respectively.

Eqs. (17) and (18) account for the effects of GPU interrupt top-half processing under PAI handling. However, we must still account for additional overheads due to multiprocessor scheduling. The bottom-half ready queue must be examined after every scheduling decision to see if a bottom-half should be scheduled instead of the job selected by the normal scheduler. Access to this ready queue must be synchronized across the globally scheduled cores, so minor overheads can be incurred. We account for this bottom-half scheduling under PAI with the term Δ^{paisch}, which is charged whenever a scheduling decision is made. Thus,

$$\widetilde{e}_i^{(\text{cpu-only})_{19}} = \widetilde{e}_i^{(\text{cpu-only})_{17}} + 2 \cdot \Delta^{paisch} \quad (19)$$

and

$$\widetilde{e}_i^{(\text{gpu-using})_{20}} = \widetilde{e}_i^{(\text{gpu-using})_{18}} + (2 + \eta_i)(2 \cdot \Delta^{paisch}) \quad (20)$$

for CPU-only and GPU-using tasks, respectively.

The timely execution of bottom-halves may also be delayed by interprocessor communication. Once a top-half has completed, the accompanying bottom-half may have sufficient priority to execute immediately, but not on the processor where the top-half has been executed. An IPI must be used to trigger the execution of the bottom-half on the remote processor (this scenario closely resembles the timer IPI latencies already discussed). We inflate every GPU-using job to include this additional IPI latency cost for each bottom-half. Thus, we have

$$\widetilde{e}_i^{(\text{gpu-using})_{21}} = \widetilde{e}_i^{(\text{gpu-using})_{20}} + \eta_i \cdot \Delta^{ipi}. \quad (21)$$

Observe the similarities and differences between Eq. (21) and Eq. (16). Whereas Eq. (16) of klmirqd includes a thread-scheduling cost for each of J_i's bottom-halves, Eq. (21) of PAI handling instead includes an IPI cost for each of J_i's bottom-halves.

Eqs. (19) and (21) account for bottom-half scheduling overheads, but not the cost of executing the bottom-halves themselves. These must be considered.

Recall that PAI attempts to schedule bottom-halves to avoid priority inversions, but it does so at the expense of using the program stacks of other tasks. Under PAI, a bottom-half can preempt the lowest-priority scheduled job when the bottom-half arrives. However, on a non-partitioned multiprocessor, the relative priority of the preempted job may increase with respect to other jobs before the bottom-half completes (this may occur when higher-priority jobs on other processors complete). Unfortunately, the preempted job cannot resume execution until the preempting bottom-half completes, freeing up the job's program stack. The job is blocked. Although PAI attempts to reduce priority inversions (and in practice does relatively well, as seen in Sec. V), it cannot eliminate inversions in the worst-case. In the worst-case, a job is blocked by *every* bottom-half that arrive between a job's release and completion. Furthermore, since any task can be preempted by a bottom-half, CPU-only and GPU-using tasks are affected alike. Thus,

$$\widetilde{e}_i^{(\text{cpu-only})_{22}} = \widetilde{e}_i^{(\text{cpu-only})_{19}} + H_i \cdot \Delta^{bh} \quad (22)$$

and

$$\widetilde{e}_i^{(\text{gpu-using})_{23}} = \widetilde{e}_i^{(\text{gpu-using})_{21}} + H_i \cdot \Delta^{bh} \quad (23)$$

for CPU-only and GPU-using tasks, respectively.

When we compare Eqs. (22) and (23) of PAI to Eqs. (12) and (13) of SLIH, we find that the equations are nearly the same. The only difference is that PAI includes additional overheads due to bottom-half scheduling.

It is important to note that this scenario for PAI cannot occur on uniprocessors or partitioned multiprocessors since the preempted task never has a priority great enough to be scheduled before the preempting bottom-half completes (barring priority inheritance mechanisms from locking protocols).

Tick Accounting. An operating system will periodically execute, once every quantum, a tick interrupt to perform periodic maintenance of internal bookkeeping data. Methods for accounting for operating system ticks are well known. As described in [33] and other sources, ticks

Overhead	Duration
Δ^{sch}	$0.63\mu s$
Δ^{cxs}	$0.36\mu s$
Δ^{ipi}	$0.60\mu s$
Δ^{rel}	$0.67\mu s$
Δ^{th}	$16.44\mu s$
Δ^{bh}	$29.90\mu s$
Δ^{krel}	$1.39\mu s$
Δ^{paisch}	$0.13\mu s$
Δ^{pairel}	$0.56\mu s$
Δ^{tck}	$0.86\mu s$
Q	$1ms$

Table III
OBSERVED OVERHEADS ON OUR EVALUATION PLATFORM. Q IS A COMPILE-TIME CONFIGURED VALUE.

can be accounted for with the following equation:

$$e_i'^{\text{tck}} = e_i' + \left\lceil \frac{p_i + x_i}{Q} \right\rceil \Delta^{tck}, \tag{24}$$

where Q denotes the tick quantum length and e_i' denotes an already-inflated task execution time. For our accounting, we account for tick overheads last, after having already inflated e_i for the various system overheads.

Observed Overheads. We executed task sets made up of both CPU-only and GPU-using tasks on our evaluation platform in LITMUS$^{\text{RT}}$ using both klmirqd and standard Linux interrupt handling and recorded logs of all scheduling events. Roughly 28 hours of execution was spent gathering these logs. From these logs we determined average-case values for each of the required overheads. We are only interested in average-case values since our system is soft real-time. With the exception of Δ^{th} and Δ^{bh}, outliers were removed by only considering values from the interquartile range (a standard statistical technique) before computing averages. We choose to not remove outliers in the averages for Δ^{th} and Δ^{bh} because, unlike the other overheads, the duration of each Δ^{th} and Δ^{bh} vary greatly. This is because each may perform a very different operation each invocation. For example, we have no visibility into the closed-source driver and are unable to identify different types of bottom-halves. Thus, we merely group all types of bottom-halves together and compute an average.

Table III displays all the relevant overheads discussed in this section. Note that Q is actually a compile-time configured variable in Linux and not an observed variable.

Please refer to Appx. D for all results to our overhead-aware schedulability experiments.

C. Priority Inversion Results

As described in Sec. V, 41 task sets were executed for two minutes in LITMUS^{RT} three times: once with klmirqd, once with PAI, and once with SLIH. Data on the frequency and duration of priority inversions was gathered. Figures for all of our gathered data are presented here.

Fig. 13 through Fig. 53 depict the probability that an observed priority inversion was less than a given value (x-axis). When comparing two curves in these graphs, a higher curve is generally better since this indicates that more priority inversions are likely to be shorter by comparison.

Fig. 54 through Fig. 94 depict the cumulative priority inversion duration as a function of maximum priority inversion duration (x-axis). In other words, for a given x value, the corresponding y value is the sum weight of all observed priority inversions with durations $\leq x$. When comparing two curves in these graphs, a lower curve for larger values of x is better since this indicates a lesser total duration of priority inversions. Stated more simply, a lower curve reflects a system that spends less time in an inversion state. *Note that with these figures, the line for klmirqd usually occupies the bottom-left corner of the graph and may be difficult to see.*

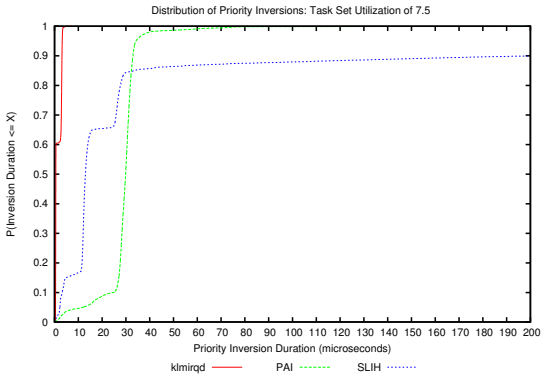

Figure 13. Task set utilization (prior inflation): 7.5. Cumulative distribution of priority inversion durations. Higher curve is better.

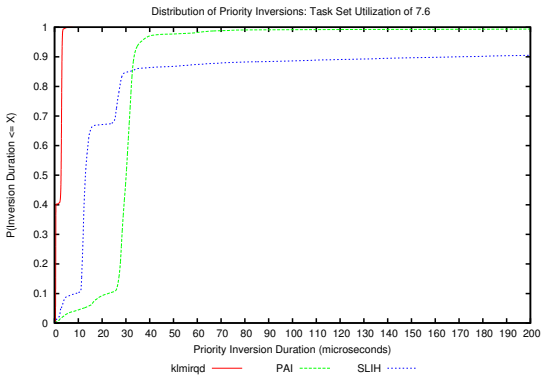

Figure 14. Task set utilization (prior inflation): 7.6. Cumulative distribution of priority inversion durations. Higher curve is better.

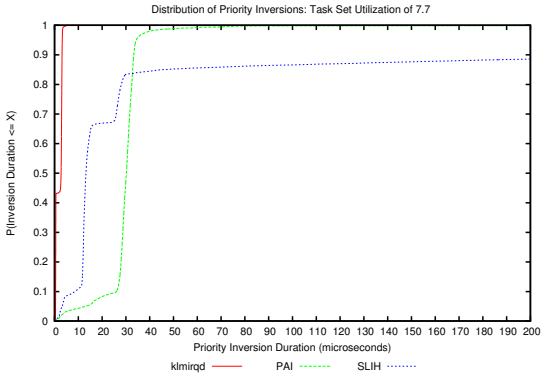

Figure 15. Task set utilization (prior inflation): 7.7. Cumulative distribution of priority inversion durations. Higher curve is better.

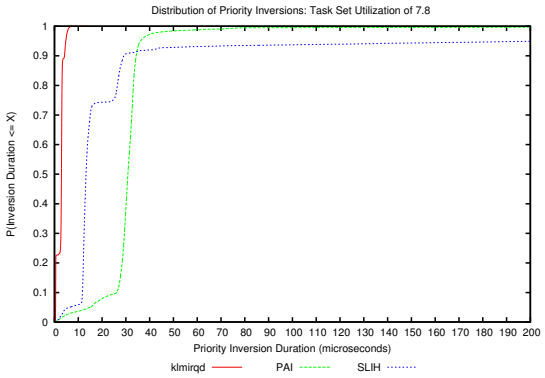

Figure 16. Task set utilization (prior inflation): 7.8. Cumulative distribution of priority inversion durations. Higher curve is better.

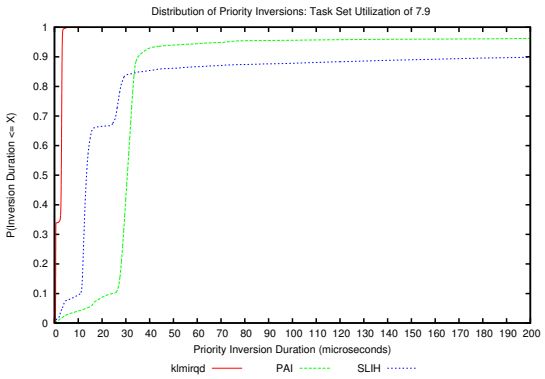

Figure 17. Task set utilization (prior inflation): 7.9. Cumulative distribution of priority inversion durations. Higher curve is better.

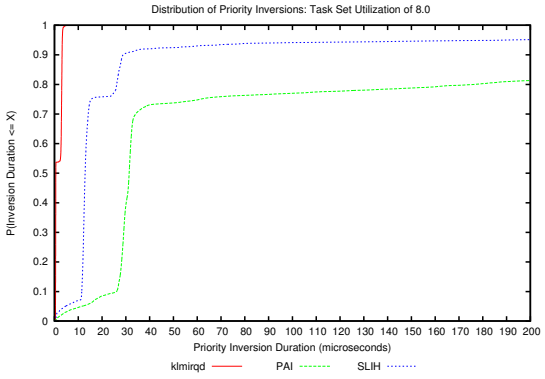

Figure 18. Task set utilization (prior inflation): 8.0. Cumulative distribution of priority inversion durations. Higher curve is better.

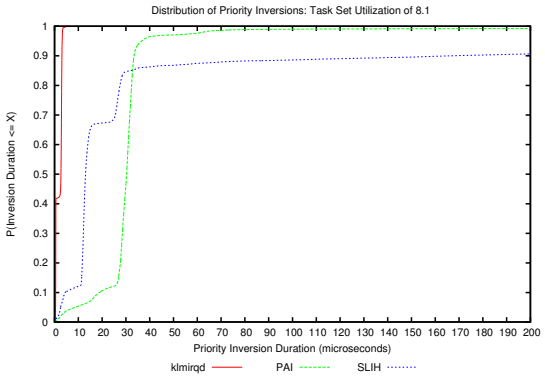

Figure 19. Task set utilization (prior inflation): 8.1. Cumulative distribution of priority inversion durations. Higher curve is better.

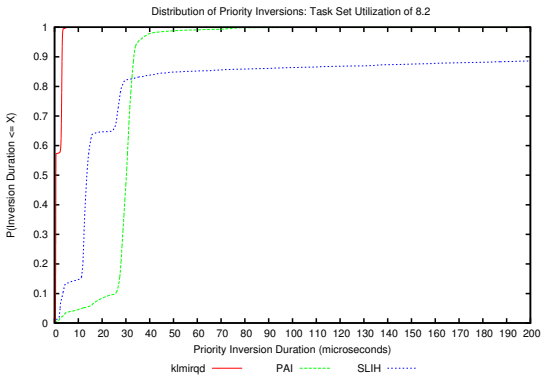

Figure 20. Task set utilization (prior inflation): 8.2. Cumulative distribution of priority inversion durations. Higher curve is better.

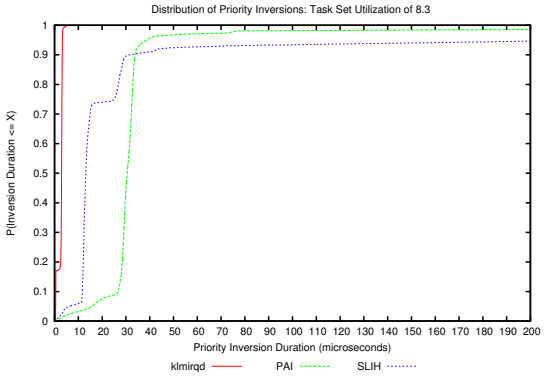

Figure 21. Task set utilization (prior inflation): 8.3. Cumulative distribution of priority inversion durations. Higher curve is better.

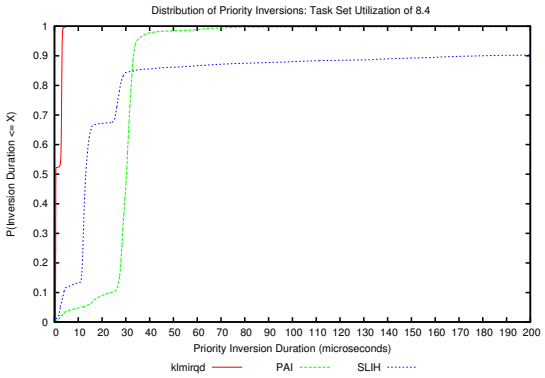

Figure 22. Task set utilization (prior inflation): 8.4. Cumulative distribution of priority inversion durations. Higher curve is better.

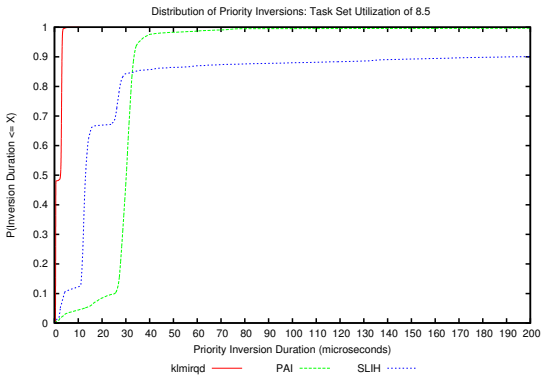

Figure 23. Task set utilization (prior inflation): 8.5. Cumulative distribution of priority inversion durations. Higher curve is better.

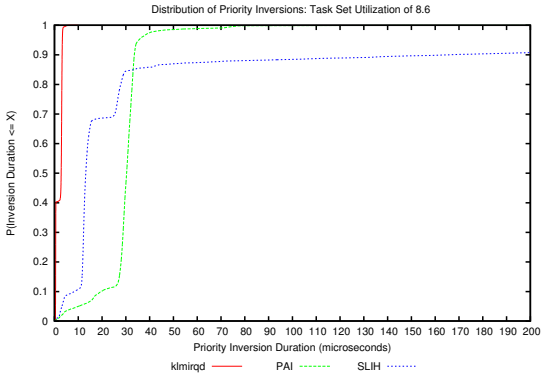

Figure 24. Task set utilization (prior inflation): 8.6. Cumulative distribution of priority inversion durations. Higher curve is better.

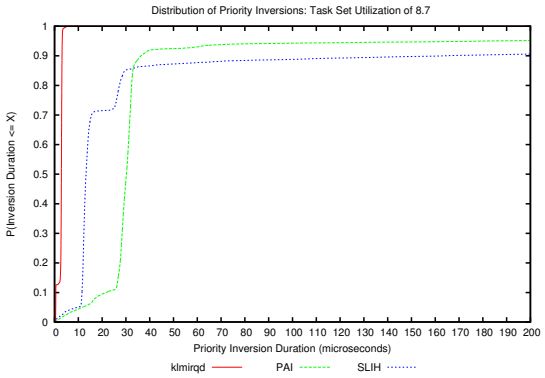

Figure 25. Task set utilization (prior inflation): 8.7. Cumulative distribution of priority inversion durations. Higher curve is better.

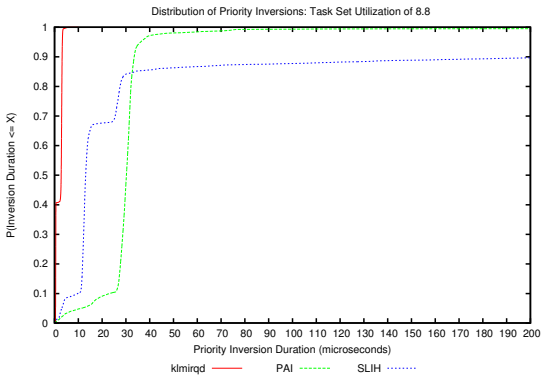

Figure 26. Task set utilization (prior inflation): 8.8. Cumulative distribution of priority inversion durations. Higher curve is better.

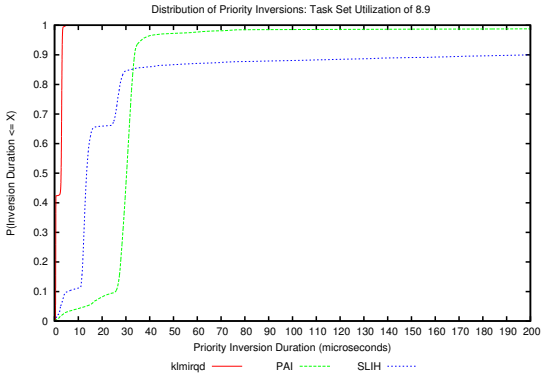

Figure 27. Task set utilization (prior inflation): 8.9. Cumulative distribution of priority inversion durations. Higher curve is better.

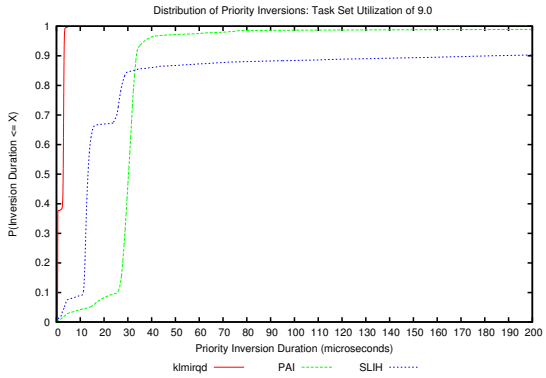

Figure 28. Task set utilization (prior inflation): 9.0. Cumulative distribution of priority inversion durations. Higher curve is better.

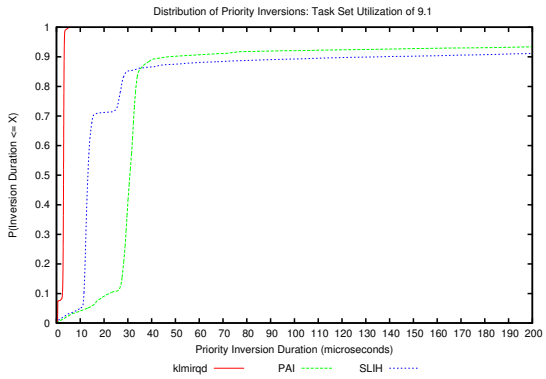

Figure 29. Task set utilization (prior inflation): 9.1. Cumulative distribution of priority inversion durations. Higher curve is better.

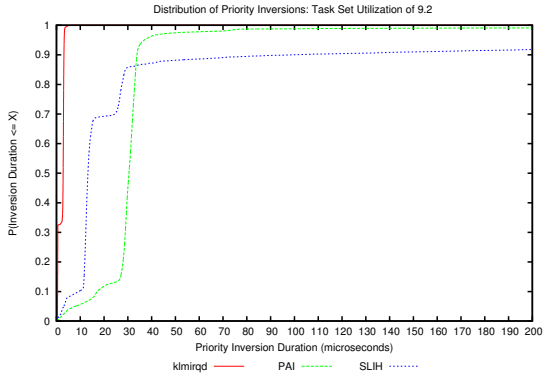

Figure 30. Task set utilization (prior inflation): 9.2. Cumulative distribution of priority inversion durations. Higher curve is better.

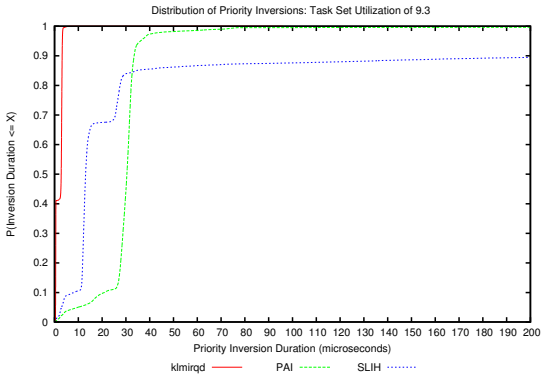

Figure 31. Task set utilization (prior inflation): 9.3. Cumulative distribution of priority inversion durations. Higher curve is better.

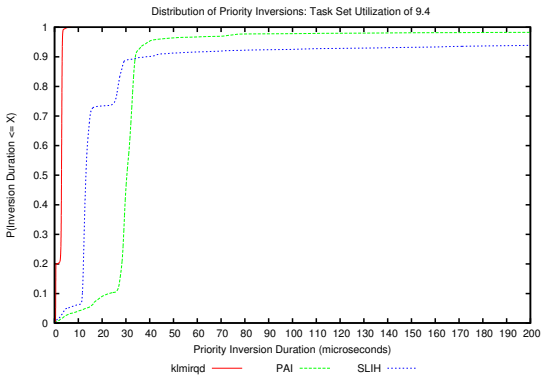

Figure 32. Task set utilization (prior inflation): 9.4. Cumulative distribution of priority inversion durations. Higher curve is better.

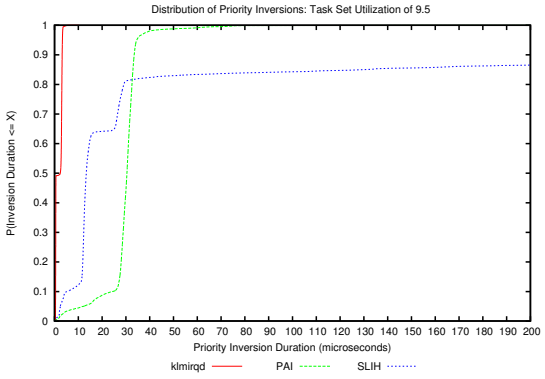

Figure 33. Task set utilization (prior inflation): 9.5. Cumulative distribution of priority inversion durations. Higher curve is better.

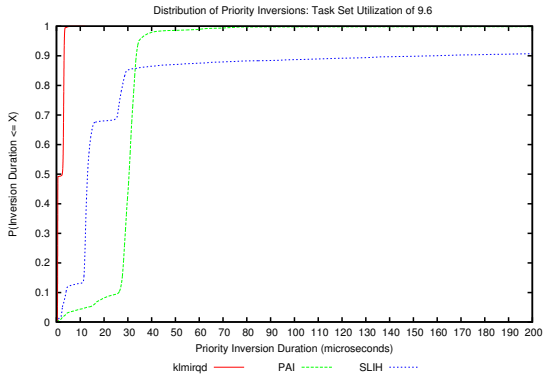

Figure 34. Task set utilization (prior inflation): 9.6. Cumulative distribution of priority inversion durations. Higher curve is better.

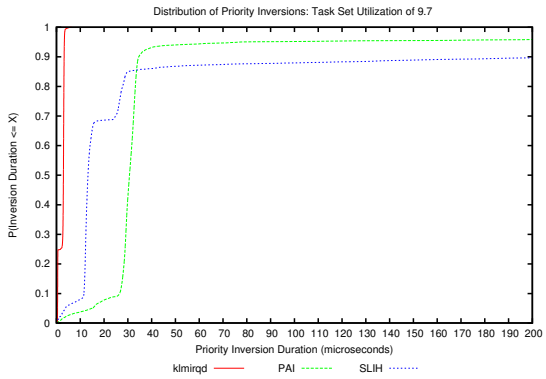

Figure 35. Task set utilization (prior inflation): 9.7. Cumulative distribution of priority inversion durations. Higher curve is better.

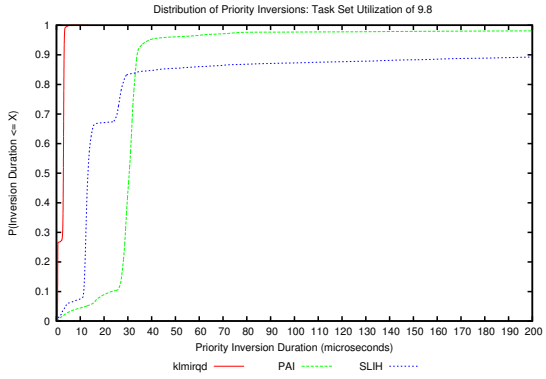

Figure 36. Task set utilization (prior inflation): 9.8. Cumulative distribution of priority inversion durations. Higher curve is better.

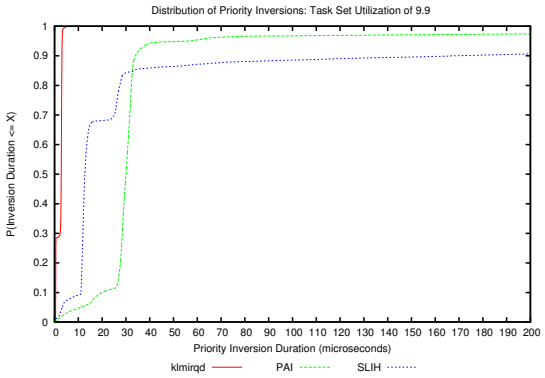

Figure 37. Task set utilization (prior inflation): 9.9. Cumulative distribution of priority inversion durations. Higher curve is better.

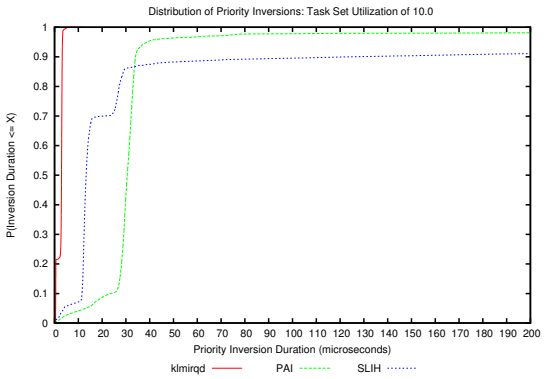

Figure 38. Task set utilization (prior inflation): 10.0. Cumulative distribution of priority inversion durations. Higher curve is better.

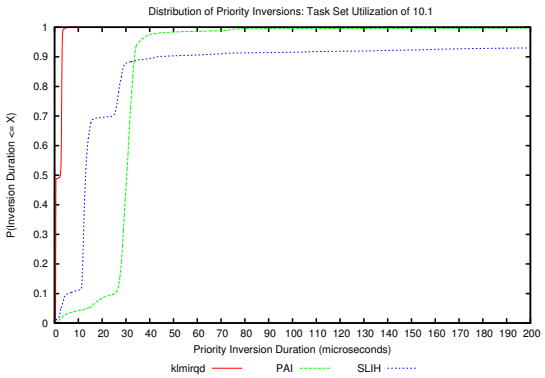

Figure 39. Task set utilization (prior inflation): 10.1. Cumulative distribution of priority inversion durations. Higher curve is better.

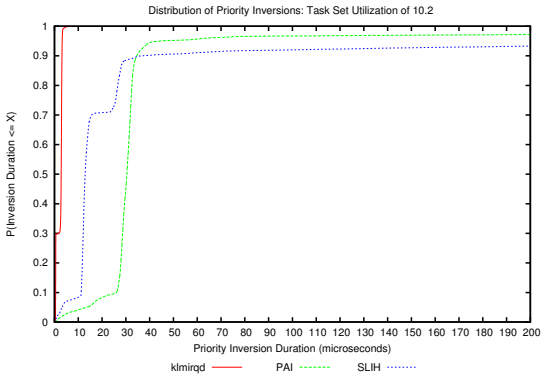

Figure 40. Task set utilization (prior inflation): 10.2. Cumulative distribution of priority inversion durations. Higher curve is better.

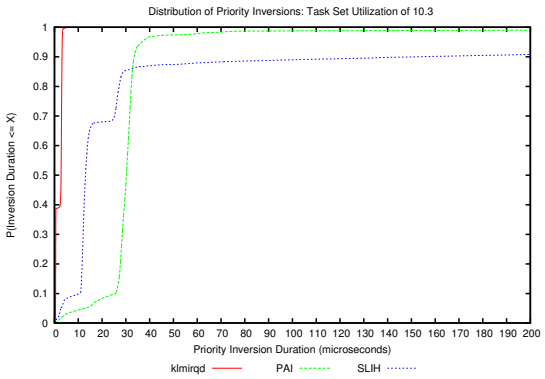

Figure 41. Task set utilization (prior inflation): 10.3. Cumulative distribution of priority inversion durations. Higher curve is better.

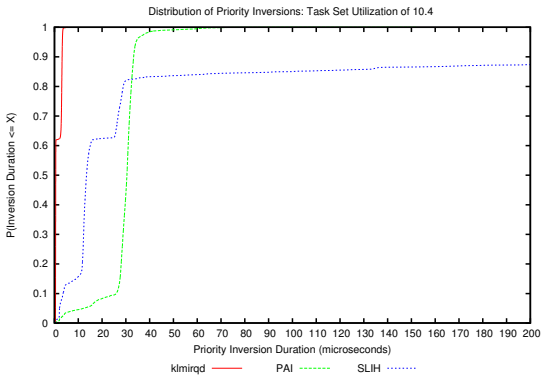

Figure 42. Task set utilization (prior inflation): 10.4. Cumulative distribution of priority inversion durations. Higher curve is better.

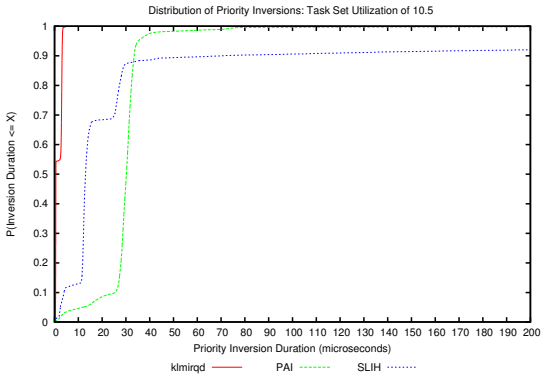

Figure 43. Task set utilization (prior inflation): 10.5. Cumulative distribution of priority inversion durations. Higher curve is better.

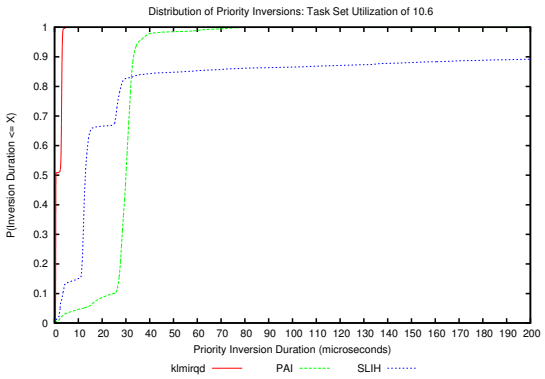

Figure 44. Task set utilization (prior inflation): 10.6. Cumulative distribution of priority inversion durations. Higher curve is better.

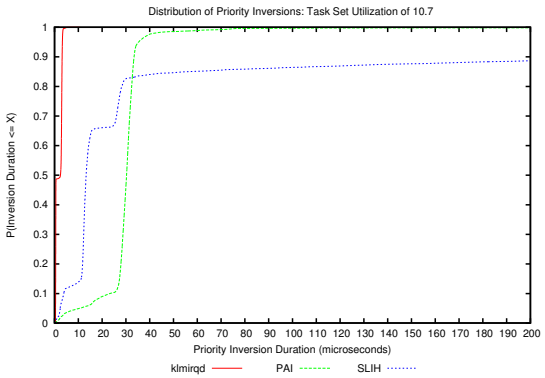

Figure 45. Task set utilization (prior inflation): 10.7. Cumulative distribution of priority inversion durations. Higher curve is better.

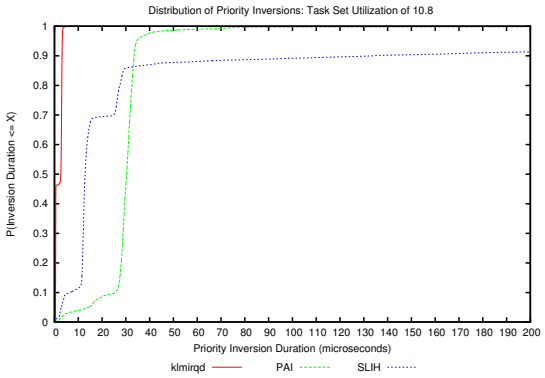

Figure 46. Task set utilization (prior inflation): 10.8. Cumulative distribution of priority inversion durations. Higher curve is better.

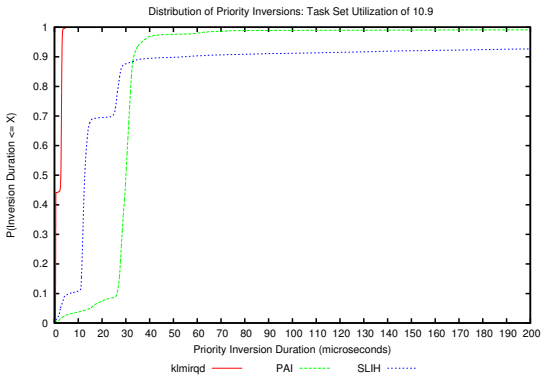

Figure 47. Task set utilization (prior inflation): 10.9. Cumulative distribution of priority inversion durations. Higher curve is better.

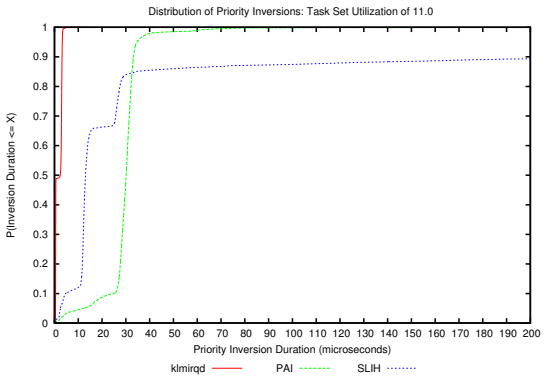

Figure 48. Task set utilization (prior inflation): 11.0. Cumulative distribution of priority inversion durations. Higher curve is better.

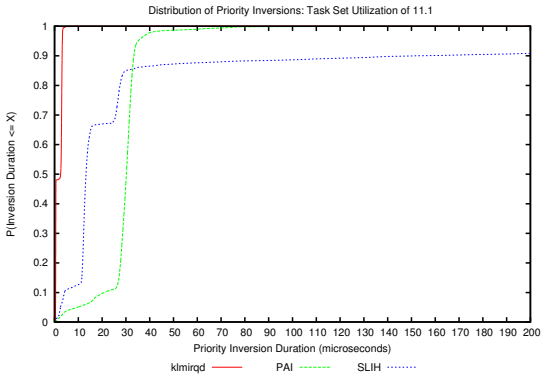

Figure 49. Task set utilization (prior inflation): 11.1. Cumulative distribution of priority inversion durations. Higher curve is better.

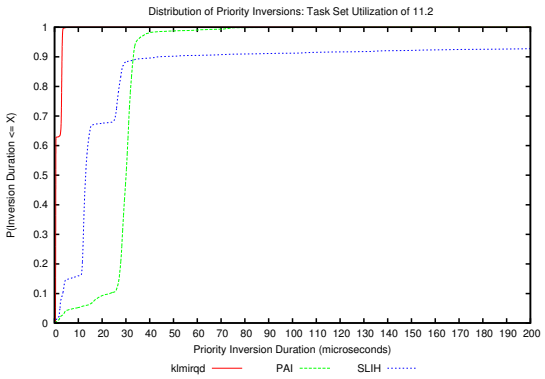

Figure 50. Task set utilization (prior inflation): 11.2. Cumulative distribution of priority inversion durations. Higher curve is better.

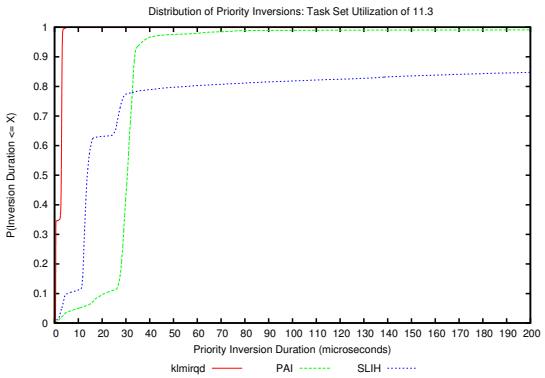

Figure 51. Task set utilization (prior inflation): 11.3. Cumulative distribution of priority inversion durations. Higher curve is better.

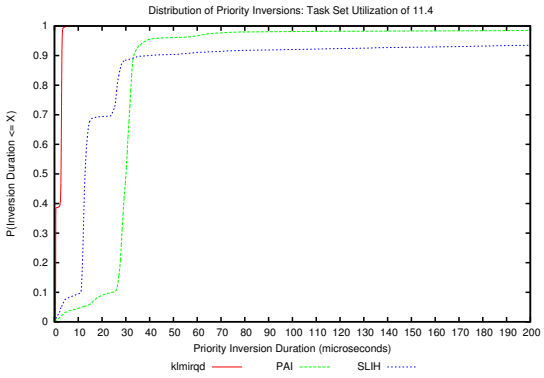

Figure 52. Task set utilization (prior inflation): 11.4. Cumulative distribution of priority inversion durations. Higher curve is better.

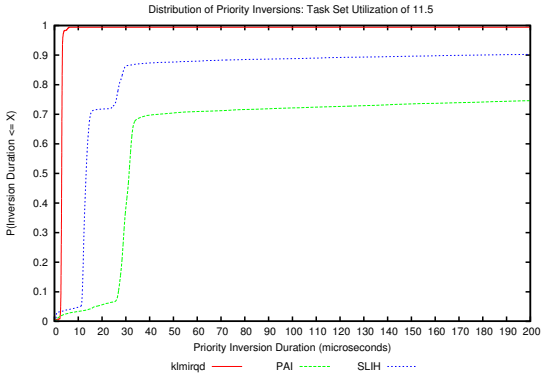

Figure 53. Task set utilization (prior inflation): 11.5. Cumulative distribution of priority inversion durations. Higher curve is better.

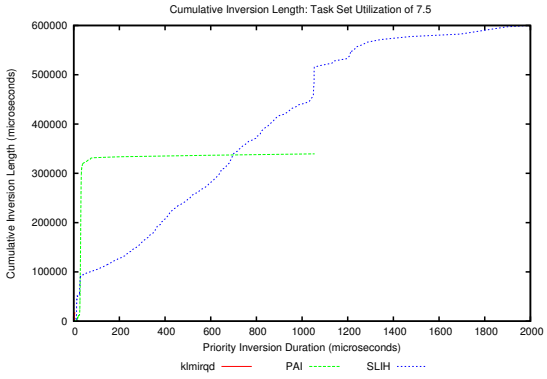

Figure 54. Task set utilization (prior inflation): 7.5. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

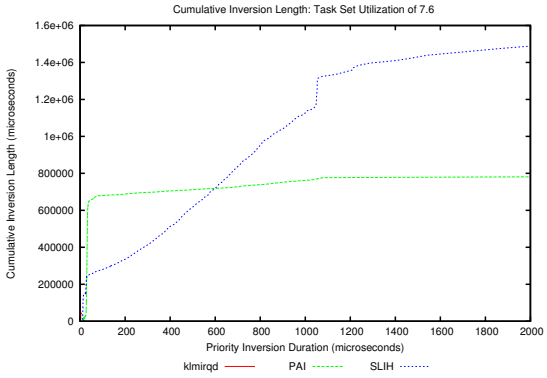

Figure 55. Task set utilization (prior inflation): 7.6. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

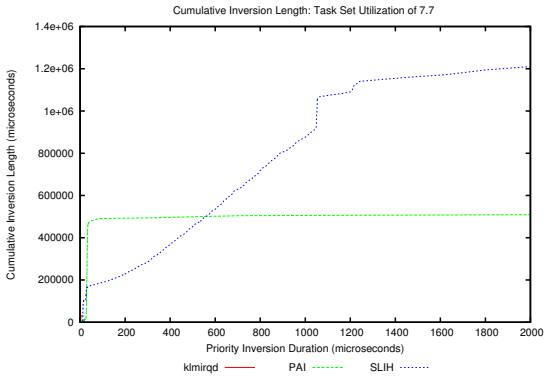

Figure 56. Task set utilization (prior inflation): 7.7. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

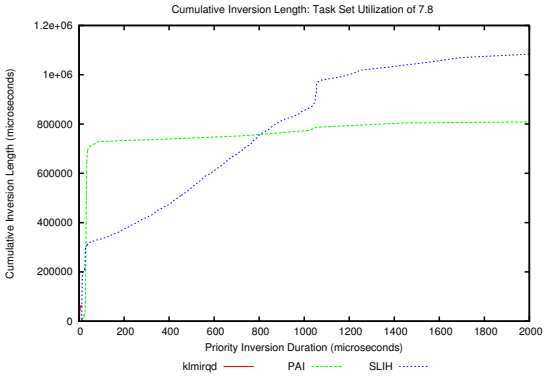

Figure 57. Task set utilization (prior inflation): 7.8. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

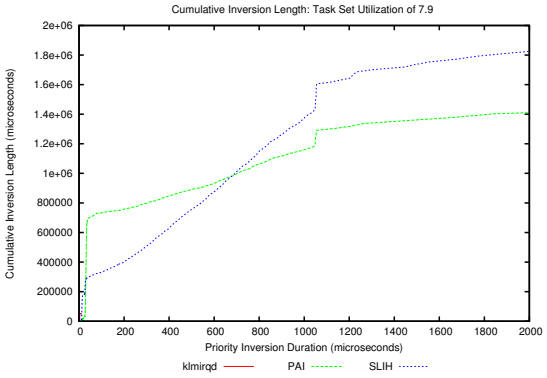

Figure 58. Task set utilization (prior inflation): 7.9. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

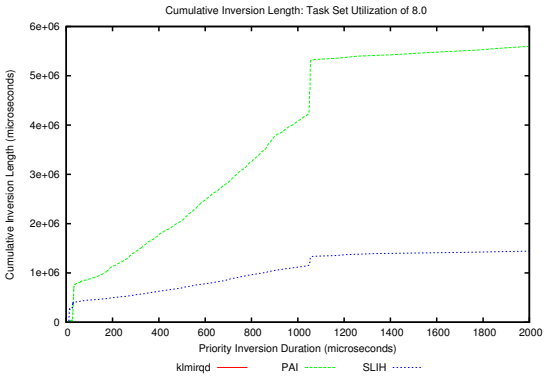

Figure 59. Task set utilization (prior inflation): 8.0. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

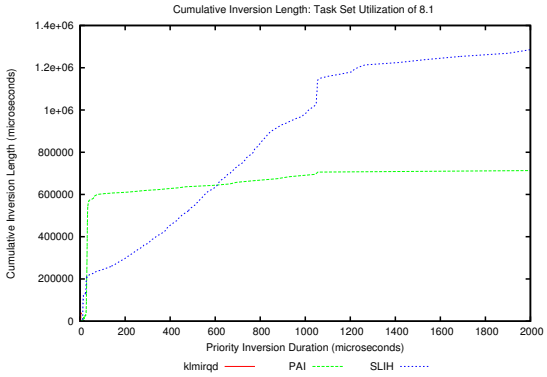

Figure 60. Task set utilization (prior inflation): 8.1. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

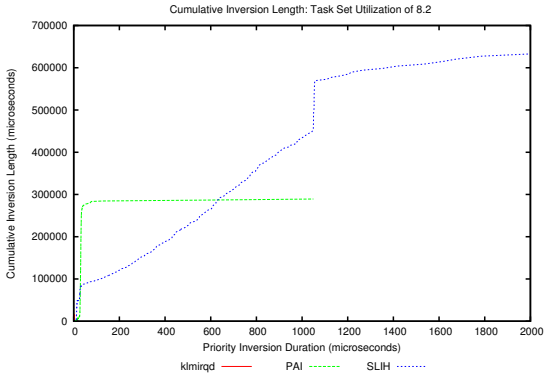

Figure 61. Task set utilization (prior inflation): 8.2. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

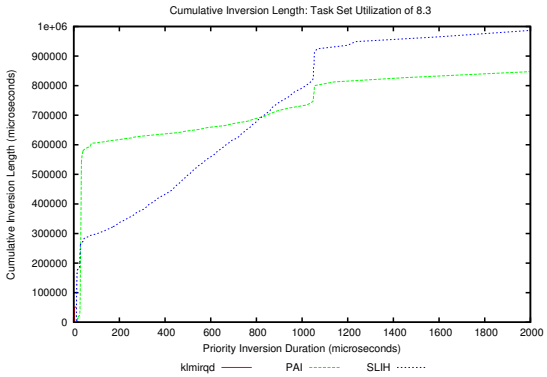

Figure 62. Task set utilization (prior inflation): 8.3. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

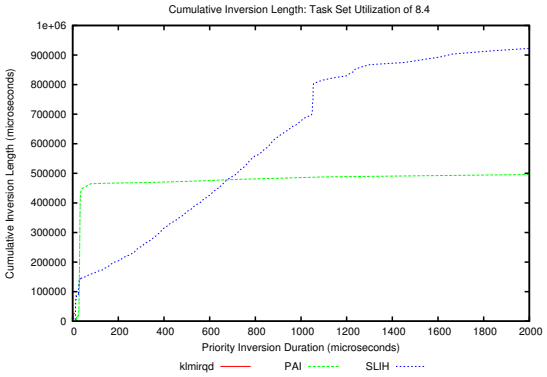

Figure 63. Task set utilization (prior inflation): 8.4. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

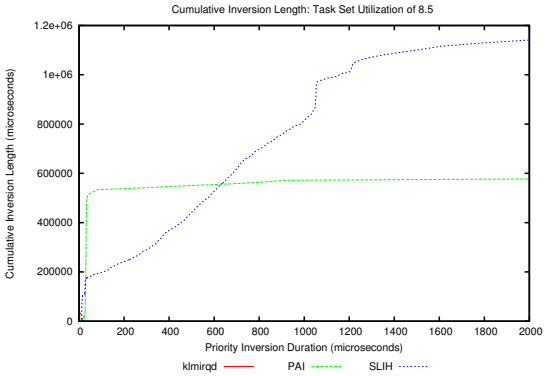

Figure 64. Task set utilization (prior inflation): 8.5. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

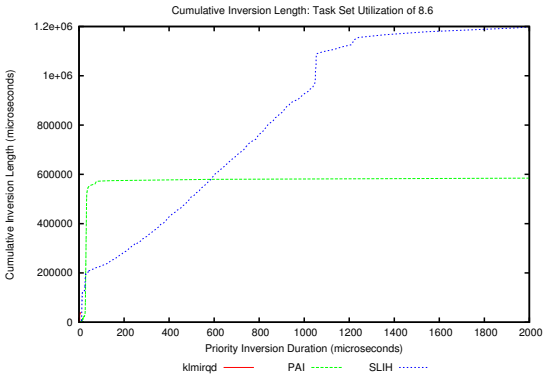

Figure 65. Task set utilization (prior inflation): 8.6. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

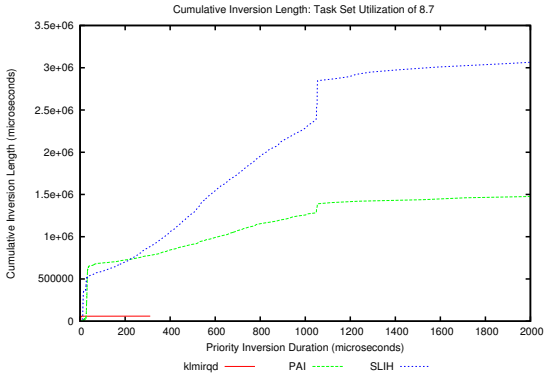

Figure 66. Task set utilization (prior inflation): 8.7. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

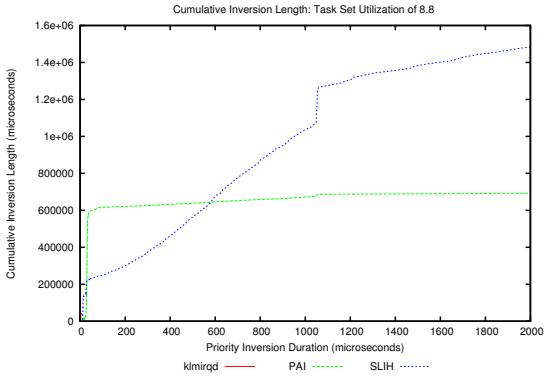

Figure 67. Task set utilization (prior inflation): 8.8. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

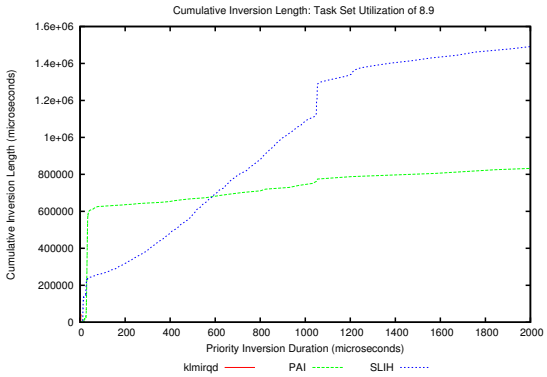

Figure 68. Task set utilization (prior inflation): 8.9. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

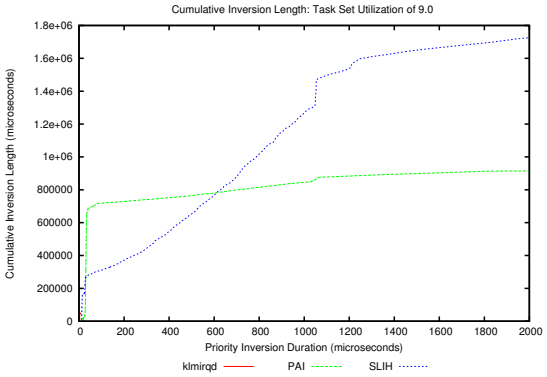

Figure 69. Task set utilization (prior inflation): 9.0. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

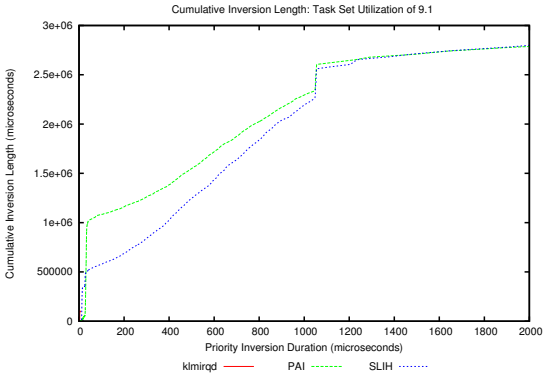

Figure 70. Task set utilization (prior inflation): 9.1. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

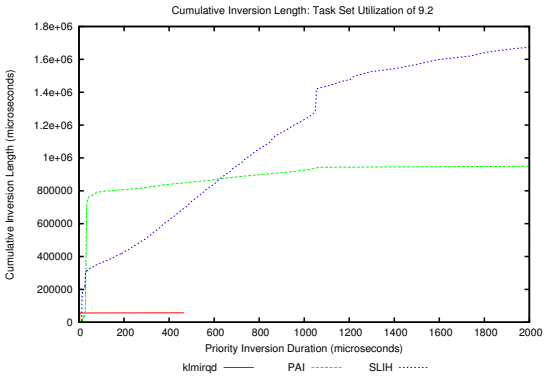

Figure 71. Task set utilization (prior inflation): 9.2. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

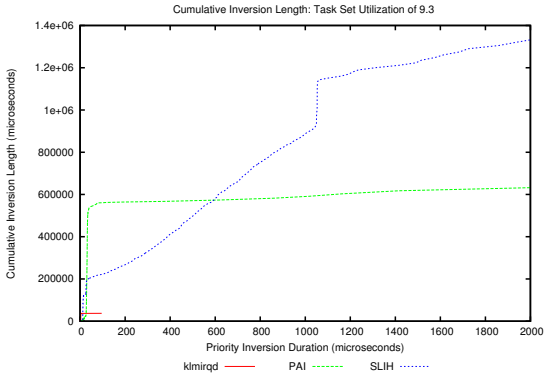

Figure 72. Task set utilization (prior inflation): 9.3. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

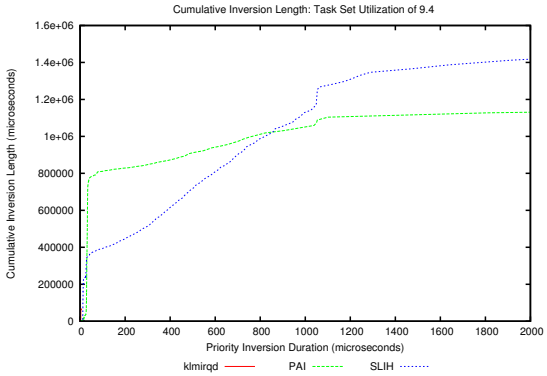

Figure 73. Task set utilization (prior inflation): 9.4. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

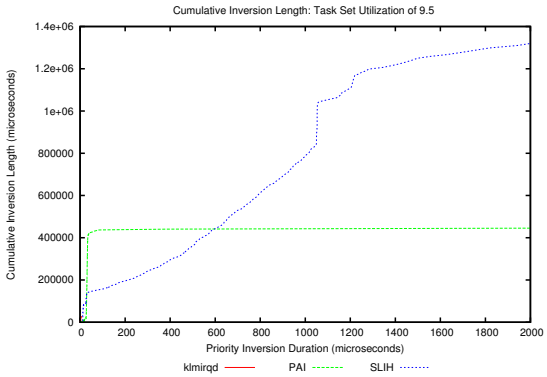

Figure 74. Task set utilization (prior inflation): 9.5. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

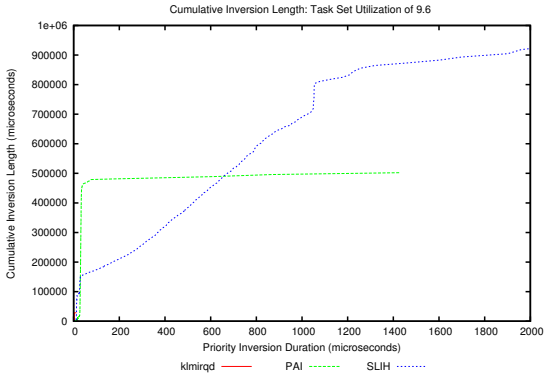

Figure 75. Task set utilization (prior inflation): 9.6. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

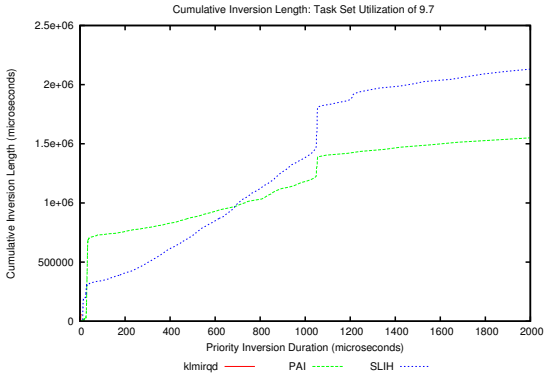

Figure 76. Task set utilization (prior inflation): 9.7. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

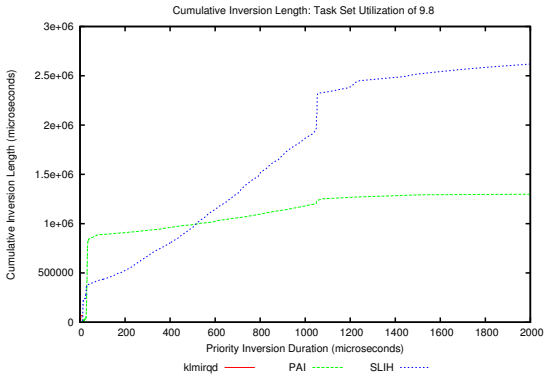

Figure 77. Task set utilization (prior inflation): 9.8. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

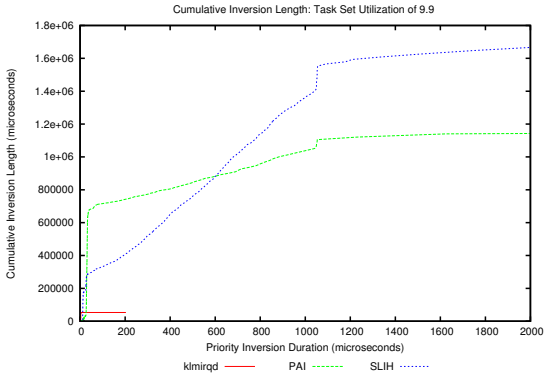

Figure 78. Task set utilization (prior inflation): 9.9. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

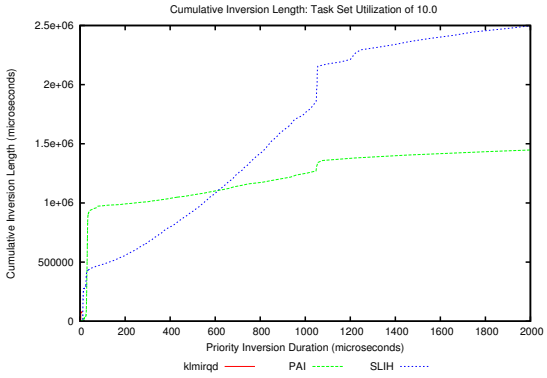

Figure 79. Task set utilization (prior inflation): 10.0. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

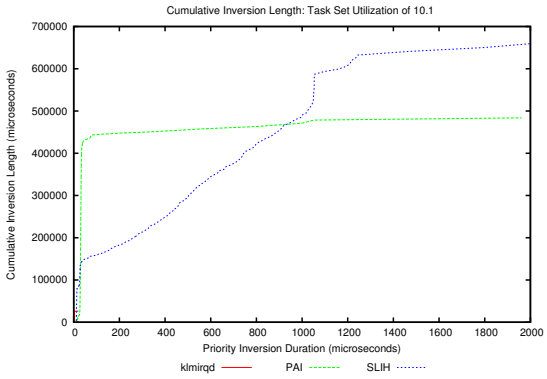

Figure 80. Task set utilization (prior inflation): 10.1. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

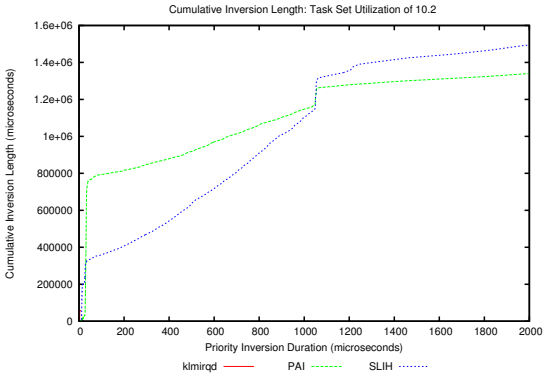

Figure 81. Task set utilization (prior inflation): 10.2. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

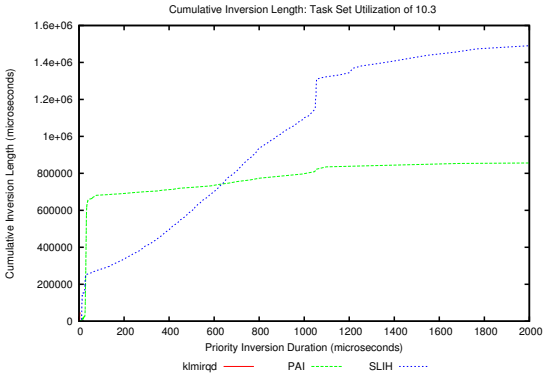

Figure 82. Task set utilization (prior inflation): 10.3. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

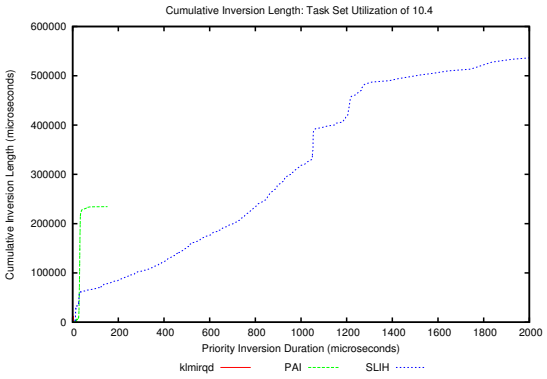

Figure 83. Task set utilization (prior inflation): 10.4. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

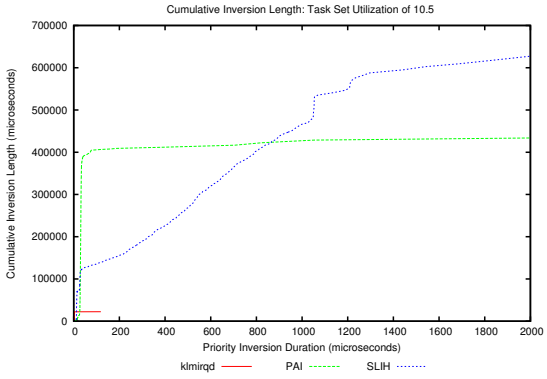

Figure 84. Task set utilization (prior inflation): 10.5. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

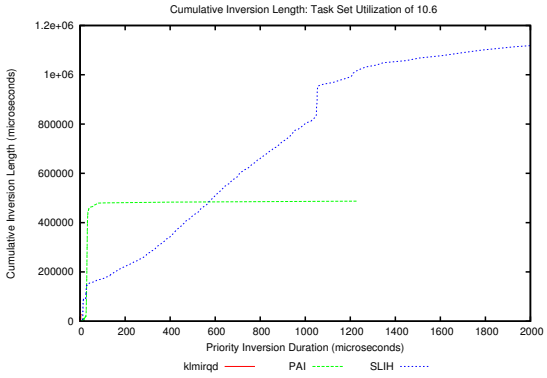

Figure 85. Task set utilization (prior inflation): 10.6. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

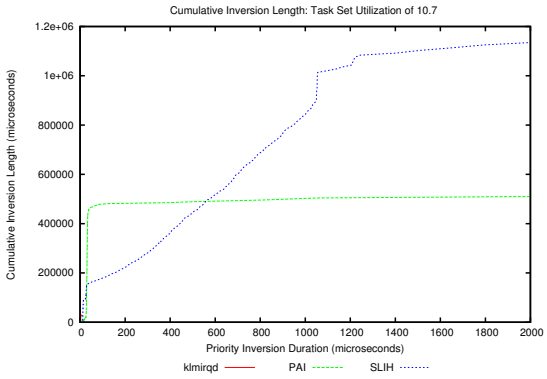

Figure 86. Task set utilization (prior inflation): 10.7. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

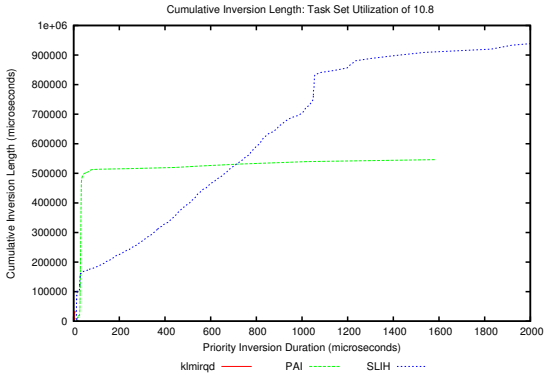

Figure 87. Task set utilization (prior inflation): 10.8. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

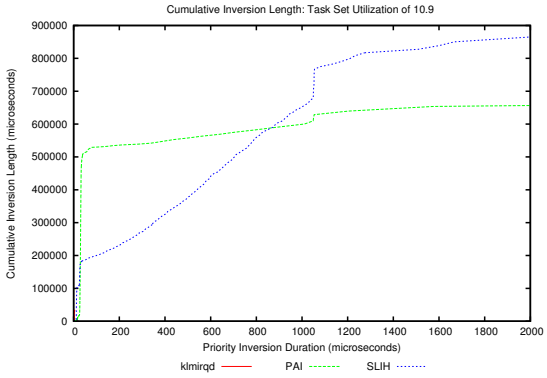

Figure 88. Task set utilization (prior inflation): 10.9. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

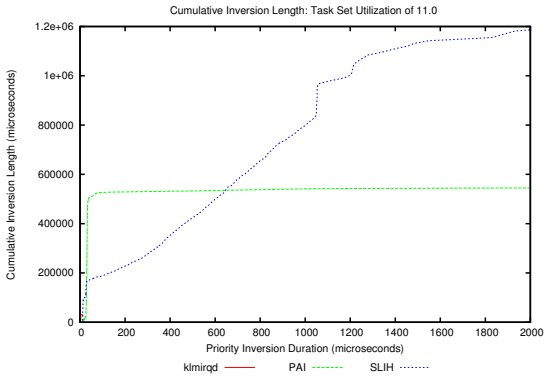

Figure 89. Task set utilization (prior inflation): 11.0. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

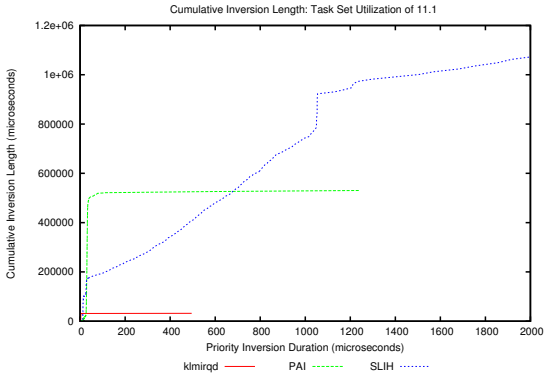

Figure 90. Task set utilization (prior inflation): 11.1. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

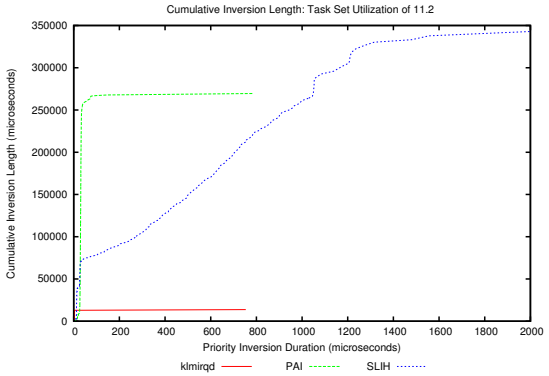

Figure 91. Task set utilization (prior inflation): 11.2. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

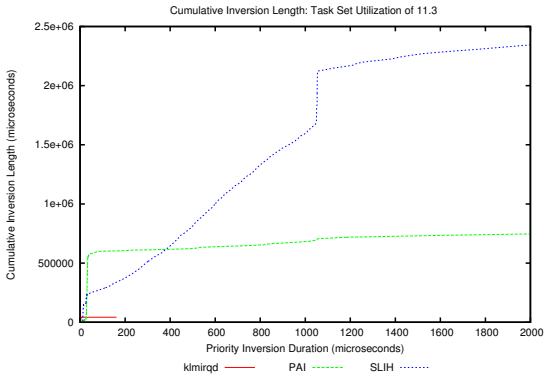

Figure 92. Task set utilization (prior inflation): 11.3. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

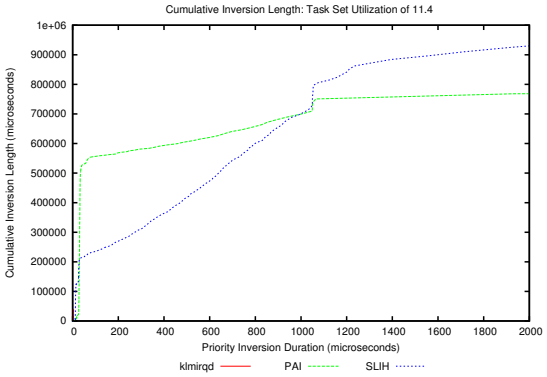

Figure 93. Task set utilization (prior inflation): 11.4. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

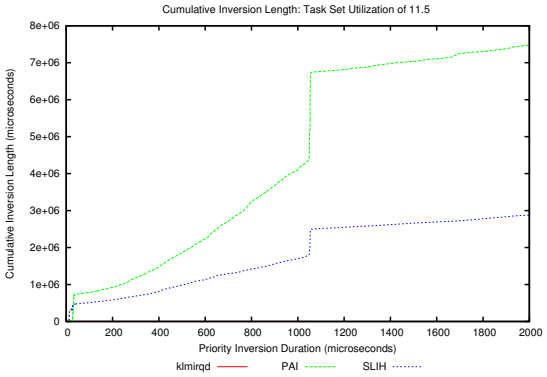

Figure 94. Task set utilization (prior inflation): 11.5. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

D. Schedulability Results

This section contains figures for all the schedulability experiments that were performed in Sec. VII. The figures are organized in the following manner. The first half of the figures are for the schedulability experiments where per-task utilizations were *uniformly* selected at random from a given utilization range. These ranges were $[1\%, 10\%]$, $[10\%, 40\%]$, and $[50\%, 90\%]$. The second half of the figures are for schedulability experiments where per-task utilizations were selected at random from a given exponential distribution. These distributions had average per-task utilizations set at 0.1, 0.25, and 0.5. For both types of experiments (uniform utilizations or exponential utilizations) the percentage share of GPU-using tasks was varied in 10% blocks (yielding ten graphs for each utilization range/distribution). These experiments were repeated four times, varying the GPU-to-CPU speed-up ratio such that $R \in \{1, 4, 8, 16\}$, as described in Sec. VII. This yields a total of 240 schedulability experiment graphs.

Note, when $R = 1$, the generated schedulability graphs are plotted against CPU utilization. When $R \neq 1$, then the generated schedulability graphs are plotted against the effective system utilization.

Please also note that the following plots may not be smooth at higher effective system utilizations when $R \neq 1$. This is because it was difficult to generate many task sets with these effective utilization. We could make these plot lines smoother, but it would require a significant increase in the number of tested task sets (each schedulability graph already represents the testing of several million task sets). As it is, it took over 24 hours to perform the schedulability experiments reflected in the following graphs. Furthermore, trends remain discernible despite non-smooth plot lines.

The organization of the following graphs are summarized in Table IV.

Per-Task Utilization Distribution	x-axis	Per-Task Utilization Range or Average	Figures
Uniform	CPU Utilization, $R = 1$	$[1\%, 10\%]$	Fig. 95 to Fig. 104
		$[10\%, 40\%]$	Fig. 105 to Fig. 114
		$[50\%, 90\%]$	Fig. 115 to Fig. 124
	Effective Utilization, $R = 4$	$[1\%, 10\%]$	Fig. 125 to Fig. 134
		$[10\%, 40\%]$	Fig. 135 to Fig. 144
		$[50\%, 90\%]$	Fig. 145 to Fig. 154
	Effective Utilization, $R = 8$	$[1\%, 10\%]$	Fig. 155 to Fig. 164
		$[10\%, 40\%]$	Fig. 165 to Fig. 174
		$[50\%, 90\%]$	Fig. 175 to Fig. 184
	Effective Utilization, $R = 16$	$[1\%, 10\%]$	Fig. 185 to Fig. 194
		$[10\%, 40\%]$	Fig. 195 to Fig. 204
		$[50\%, 90\%]$	Fig. 205 to Fig. 214
Exponential	CPU Utilization, $R = 1$	10%	Fig. 215 to Fig. 224
		25%	Fig. 225 to Fig. 234
		50%	Fig. 235 to Fig. 244
	Effective Utilization, $R = 4$	10%	Fig. 245 to Fig. 254
		25%	Fig. 255 to Fig. 264
		50%	Fig. 265 to Fig. 274
	Effective Utilization, $R = 8$	10%	Fig. 275 to Fig. 284
		25%	Fig. 285 to Fig. 294
		50%	Fig. 295 to Fig. 304
	Effective Utilization, $R = 16$	10%	Fig. 305 to Fig. 314
		25%	Fig. 315 to Fig. 324
		50%	Fig. 325 to Fig. 334

Table IV
ORGANIZATION OF FIGURED FOR SCHEDULABILITY EXPERIMENTS.

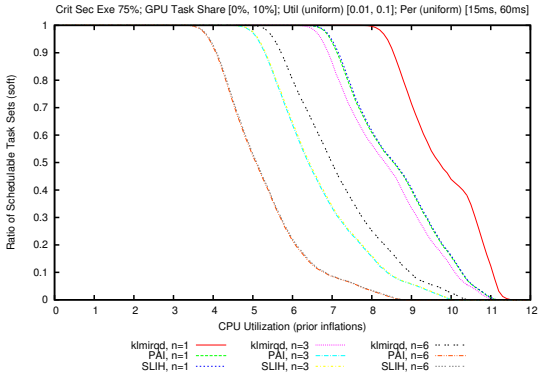

Figure 95. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

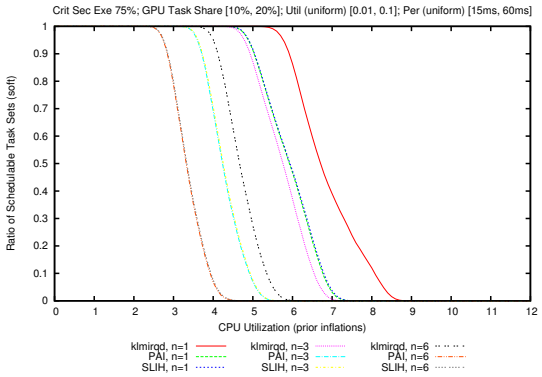

Figure 96. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

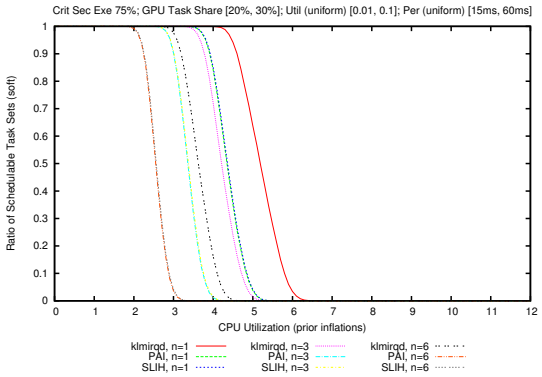

Figure 97. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

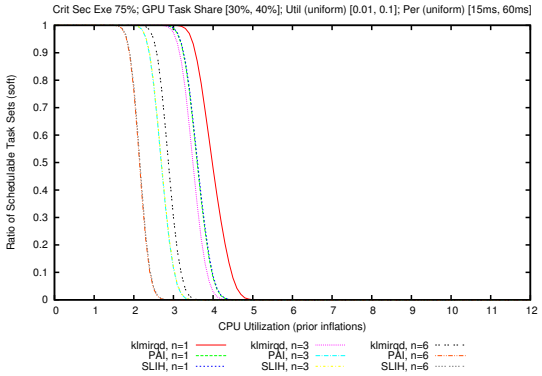

Figure 98. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

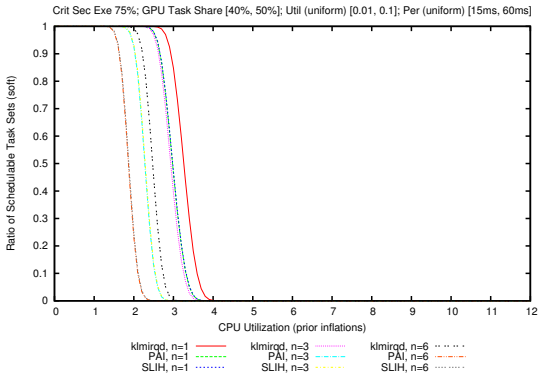

Figure 99. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

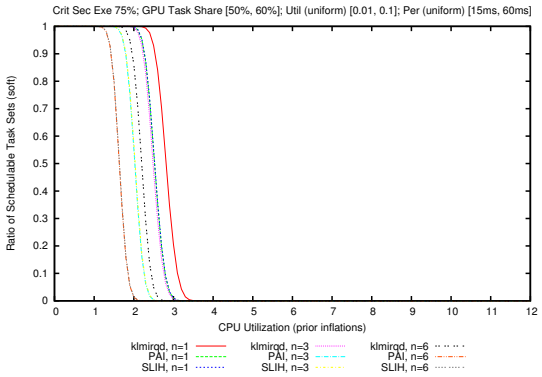

Figure 100. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

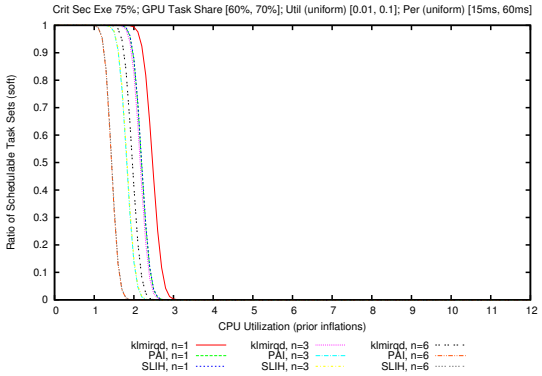

Figure 101. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

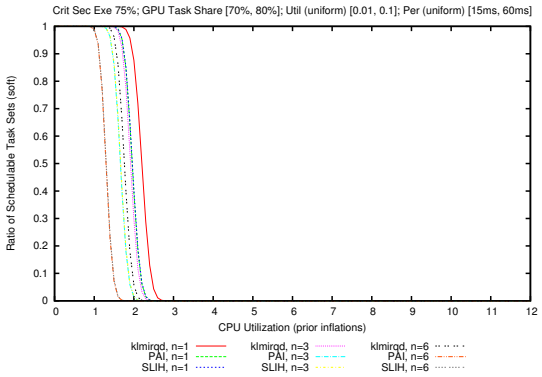

Figure 102. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

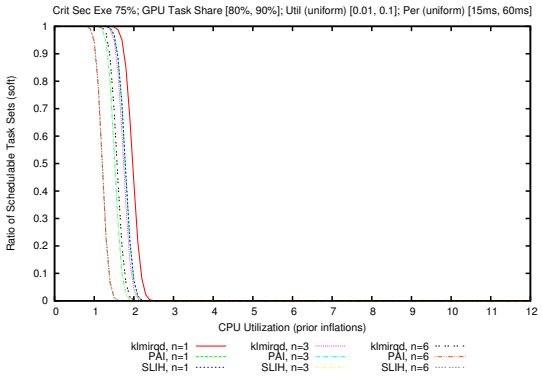

Figure 103. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

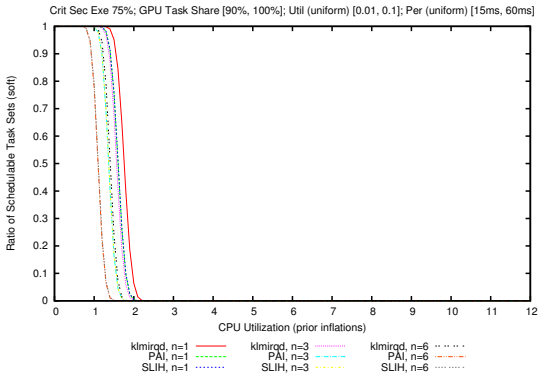

Figure 104. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

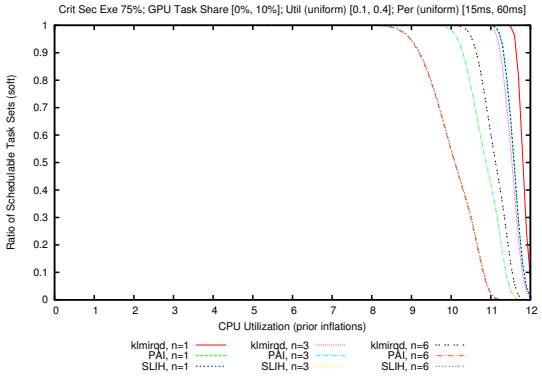

Figure 105. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

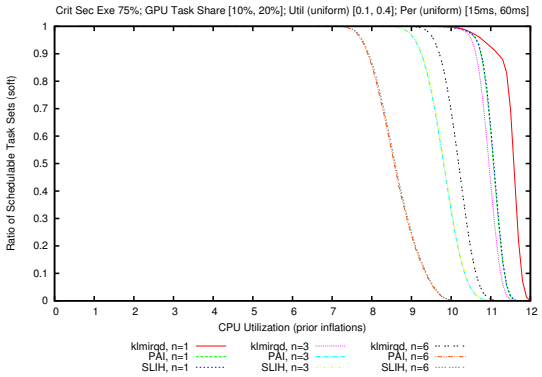

Figure 106. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

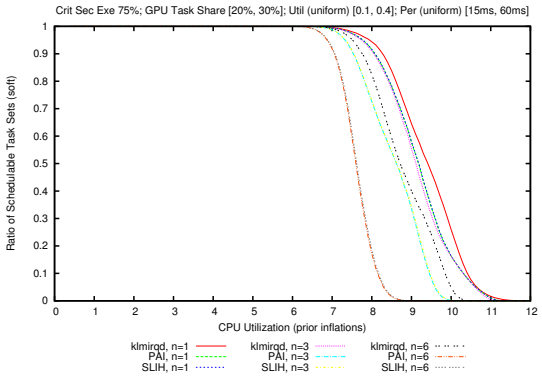

Figure 107. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

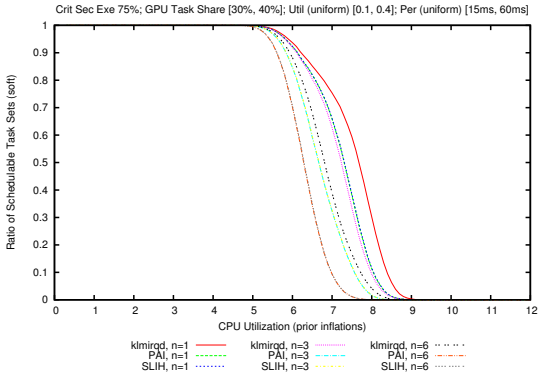

Figure 108. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

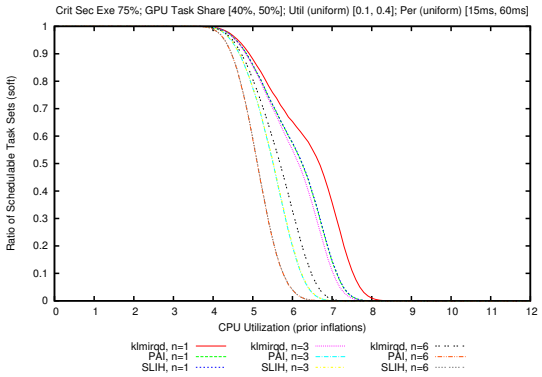

Figure 109. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

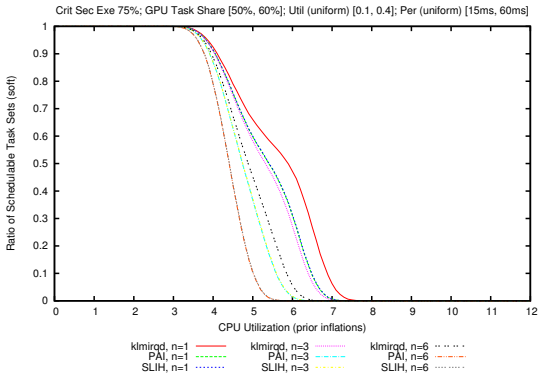

Figure 110. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

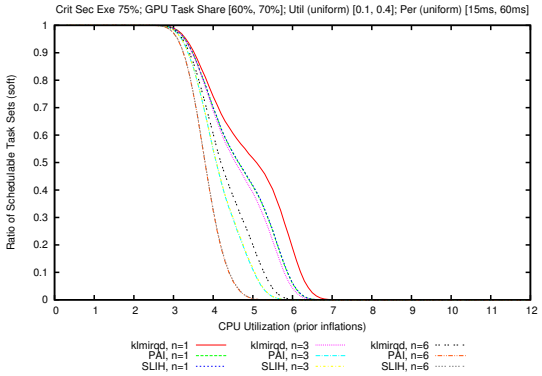

Figure 111. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

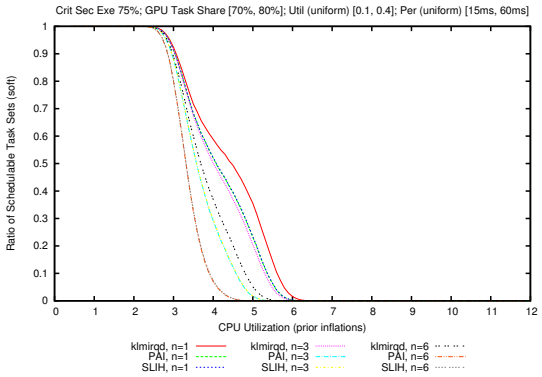

Figure 112. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

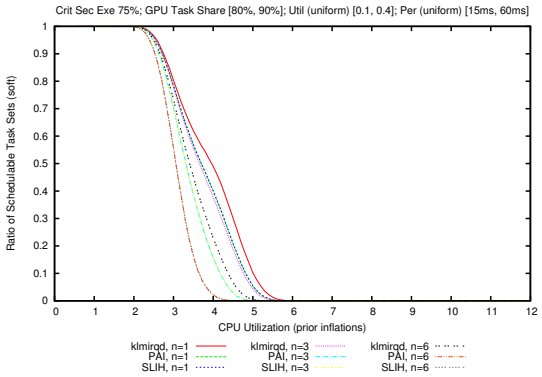

Figure 113. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

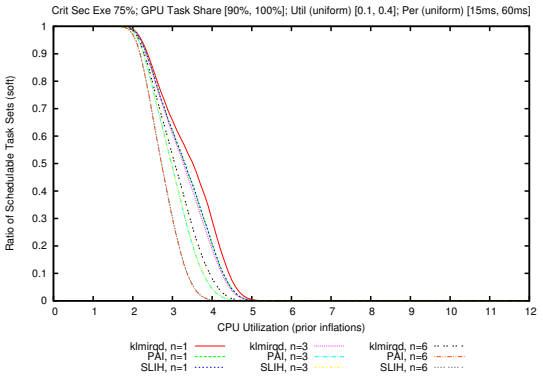

Figure 114. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

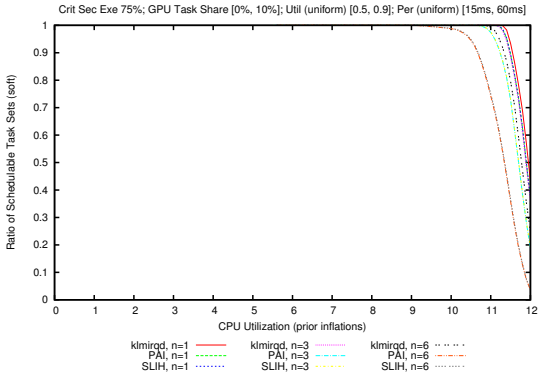

Figure 115. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

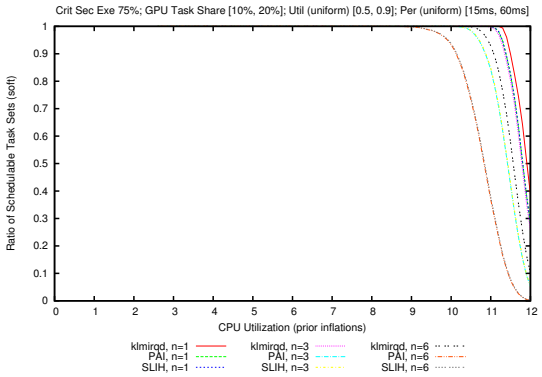

Figure 116. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

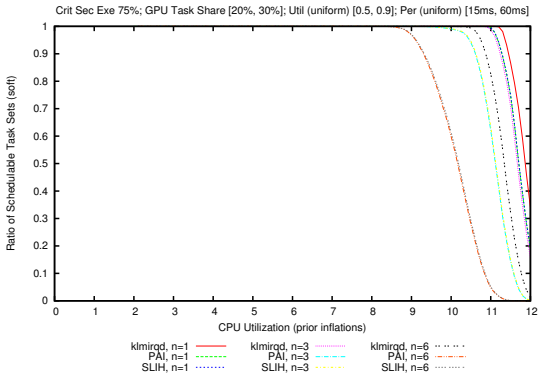

Figure 117. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

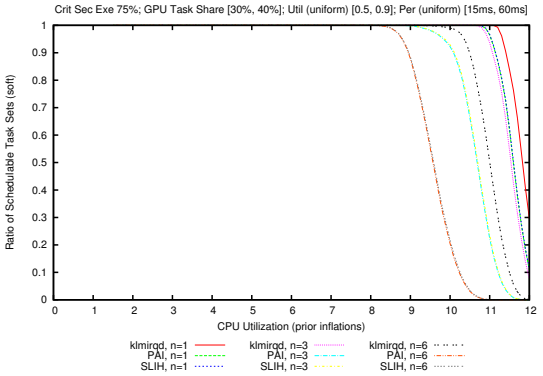

Figure 118. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

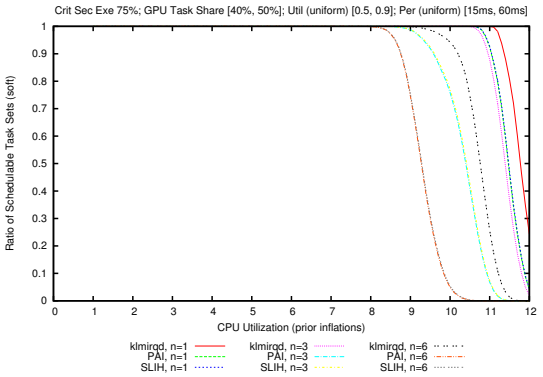

Figure 119. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

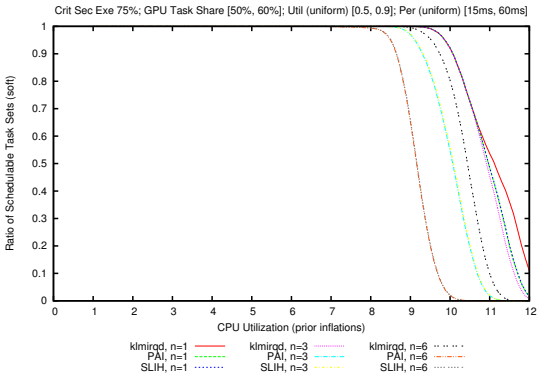

Figure 120. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

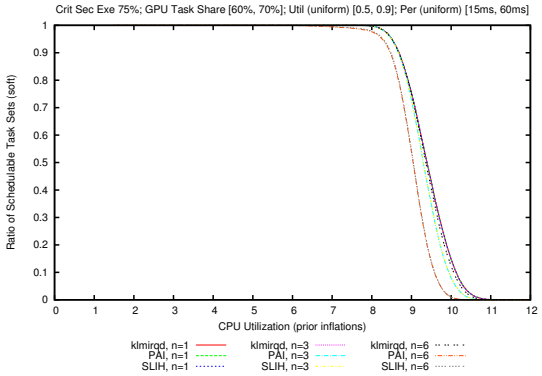

Figure 121. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

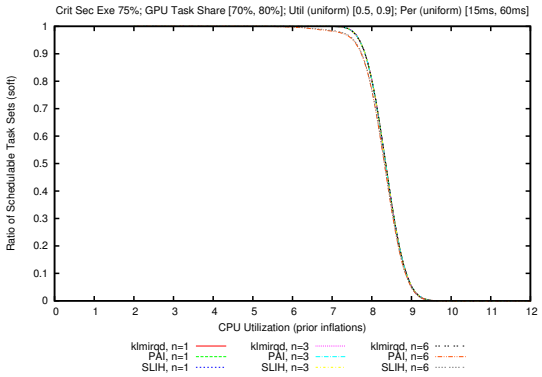

Figure 122. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

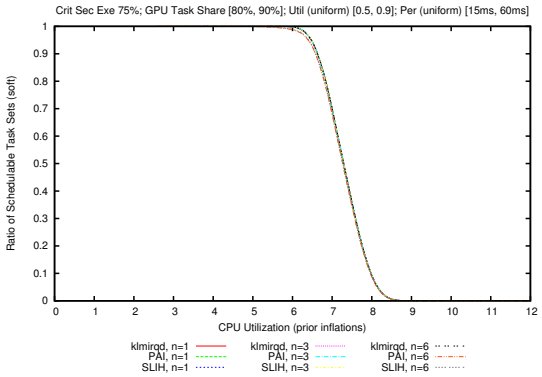

Figure 123. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

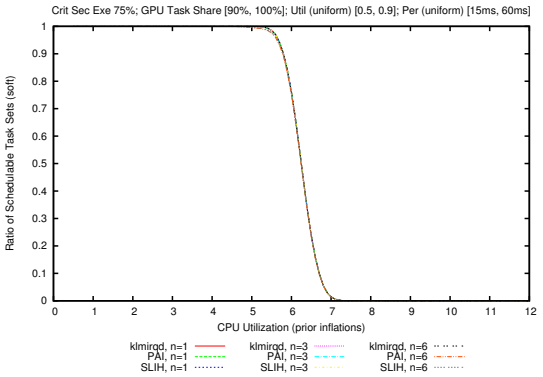

Figure 124. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

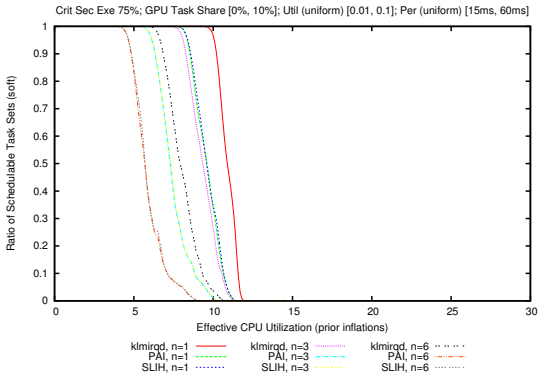

Figure 125. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

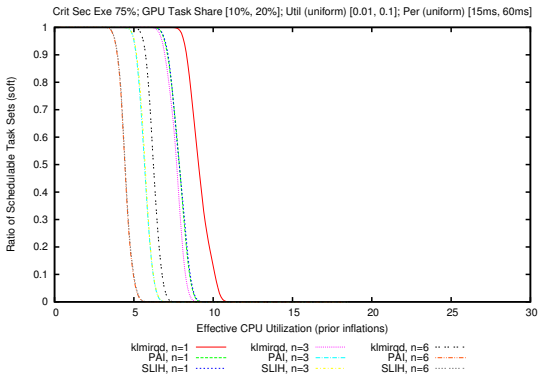

Figure 126. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

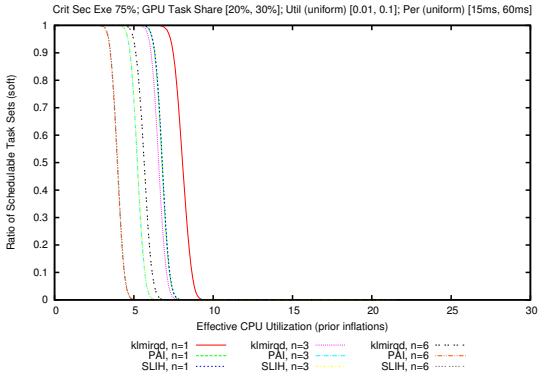

Figure 127. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

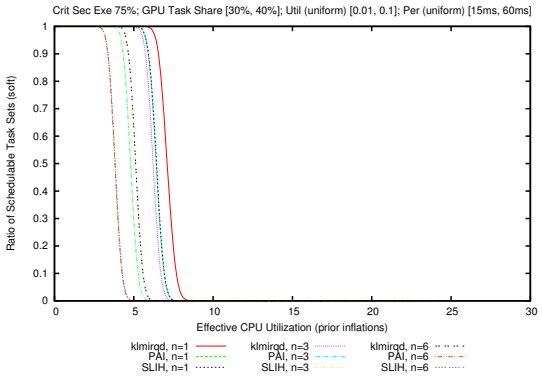

Figure 128. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

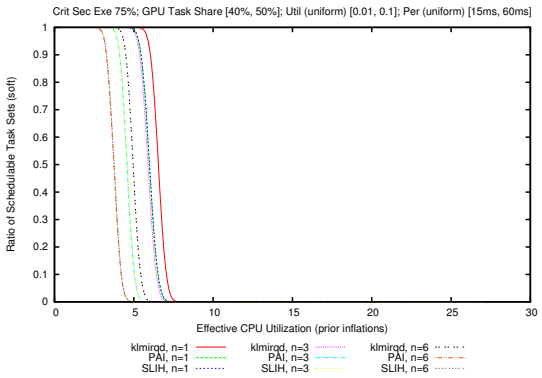

Figure 129. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

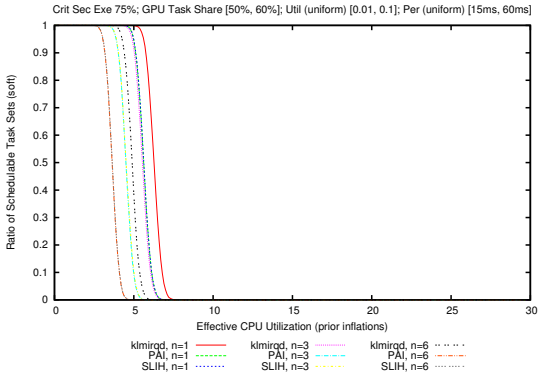

Figure 130. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

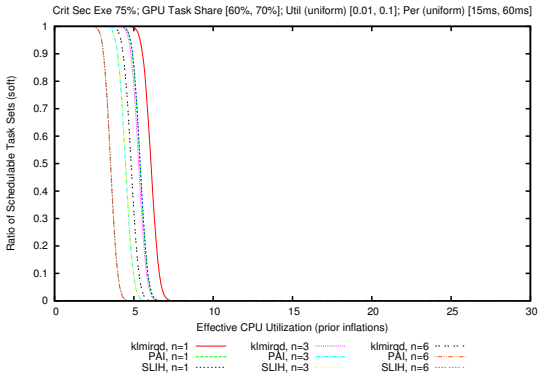

Figure 131. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

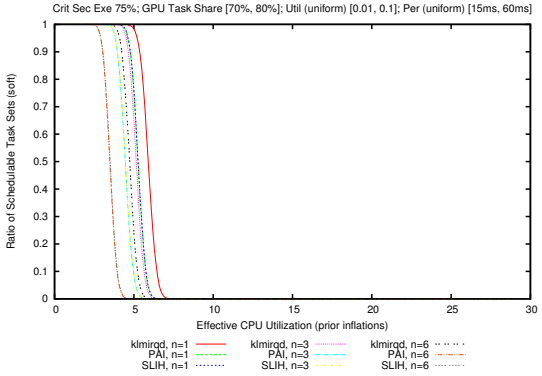

Figure 132. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

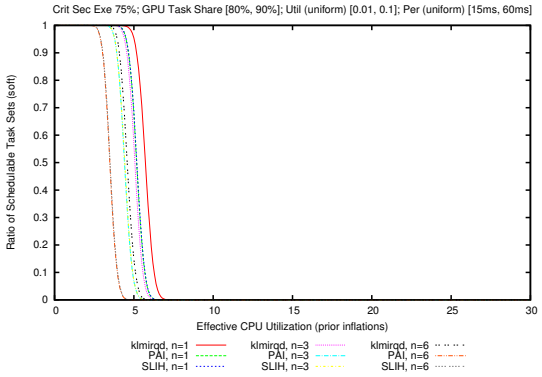

Figure 133. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

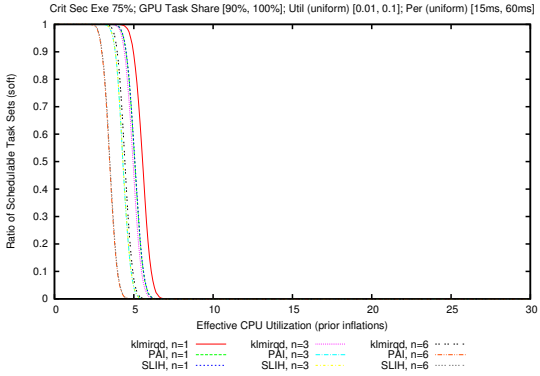

Figure 134. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

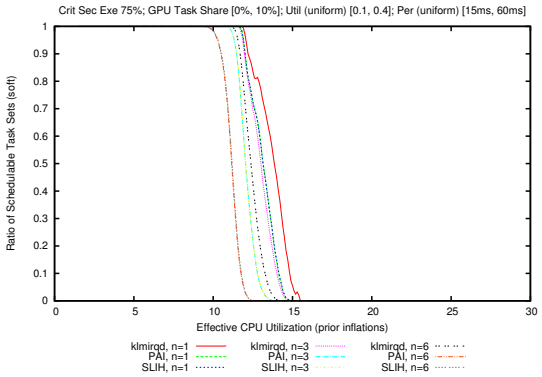

Figure 135. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

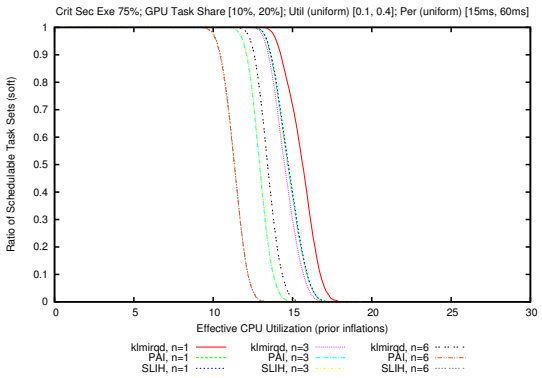

Figure 136. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

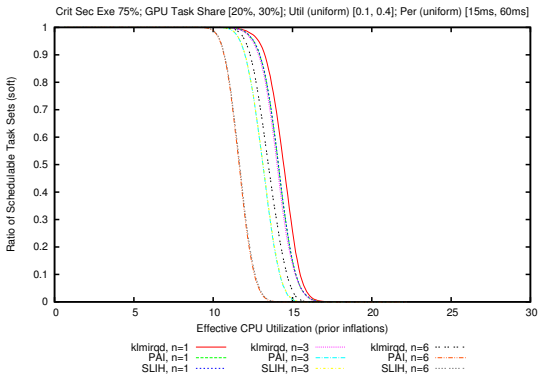

Figure 137. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

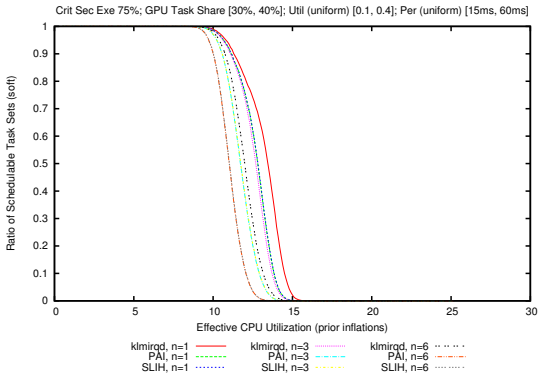

Figure 138. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

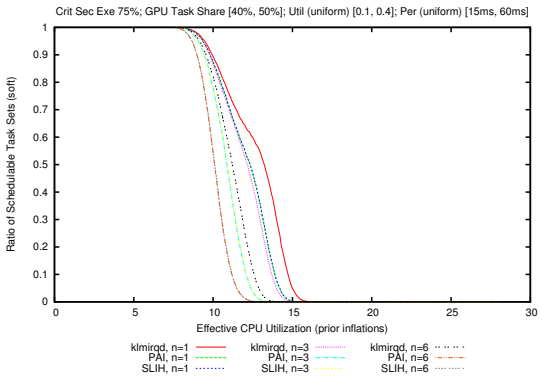

Figure 139. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

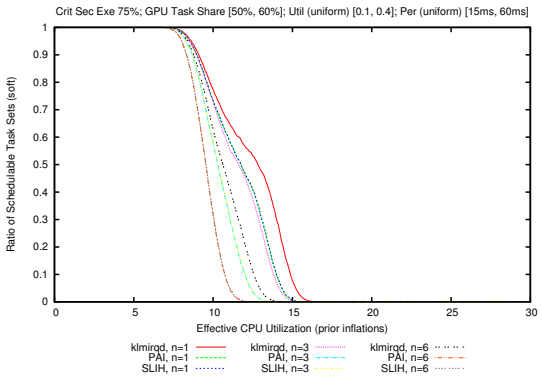

Figure 140. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

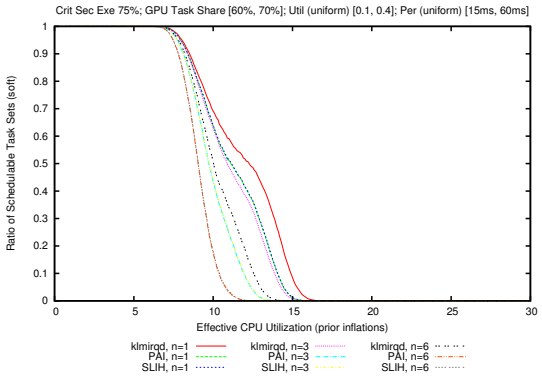

Figure 141. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

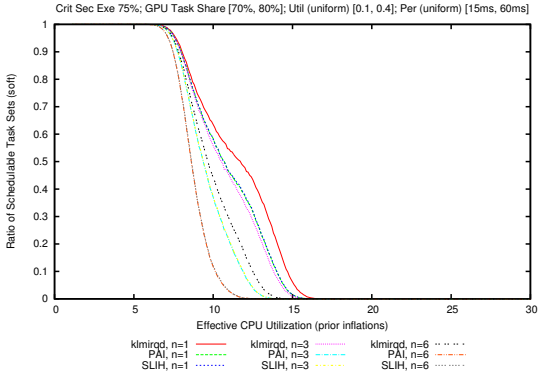

Figure 142. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

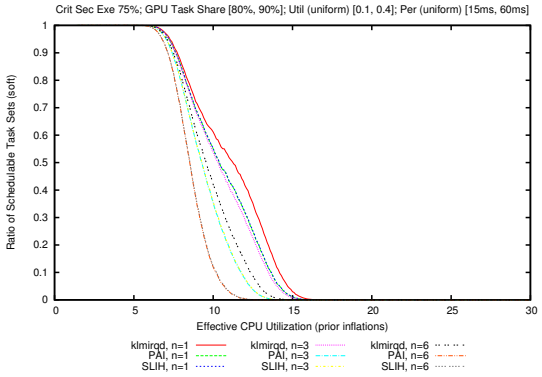

Figure 143. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

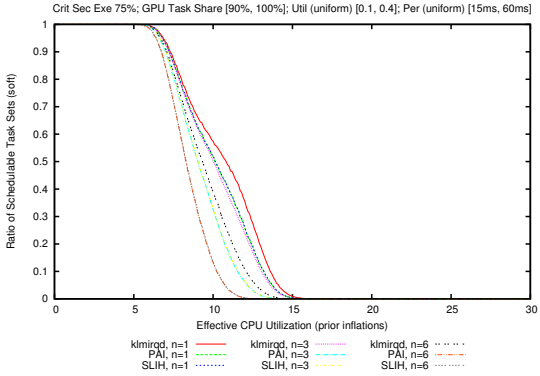

Figure 144. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

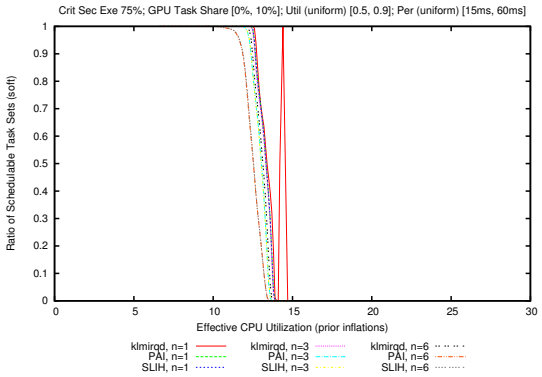

Figure 145. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

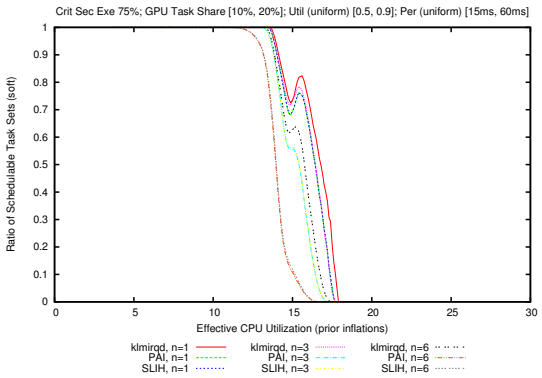

Figure 146. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

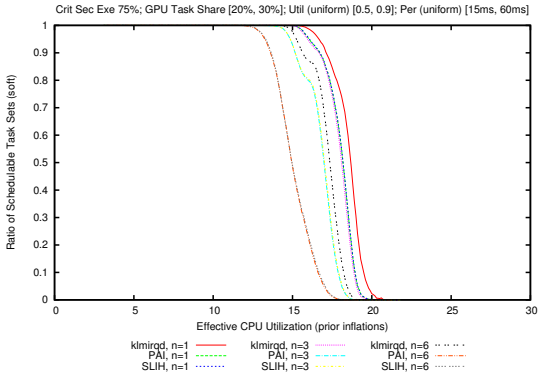

Figure 147. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

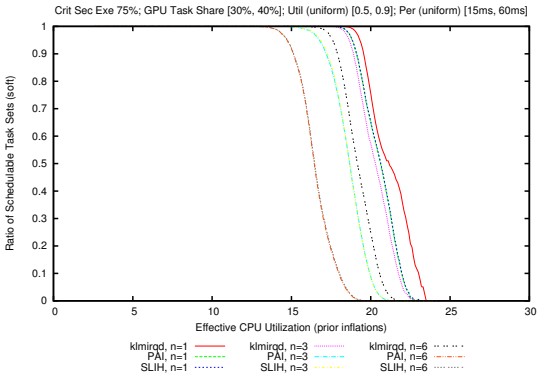

Figure 148. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

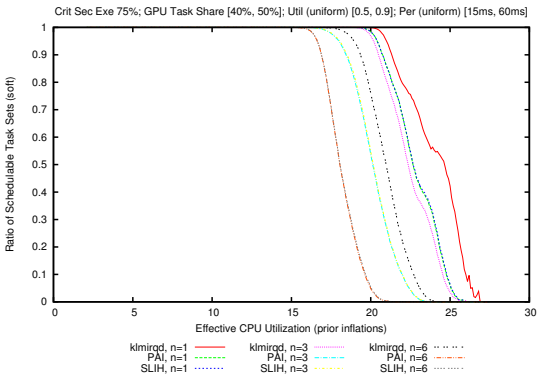

Figure 149. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

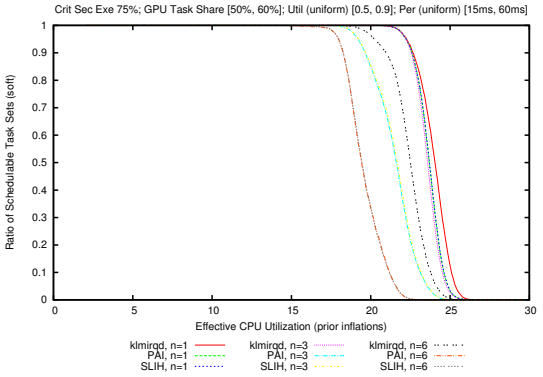

Figure 150. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

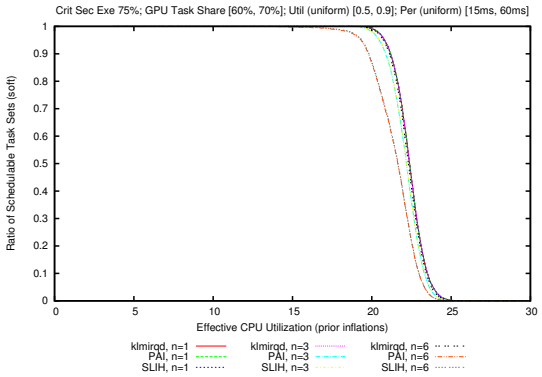

Figure 151. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

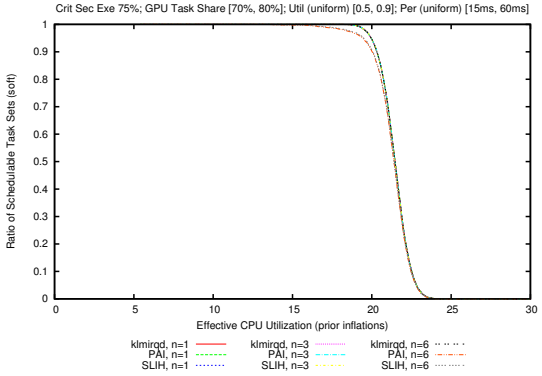

Figure 152. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

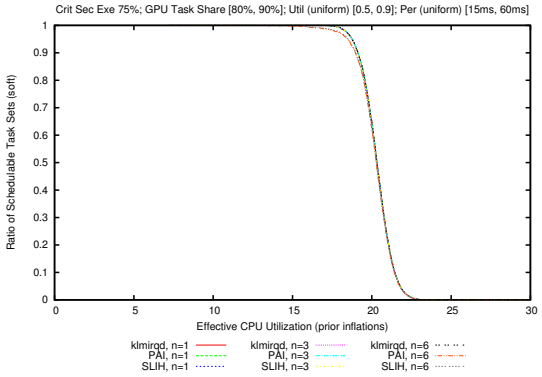

Figure 153. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

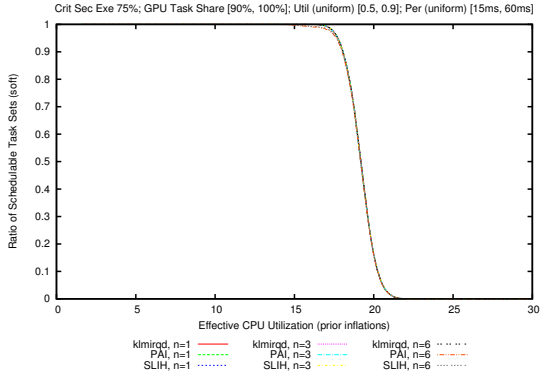

Figure 154. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

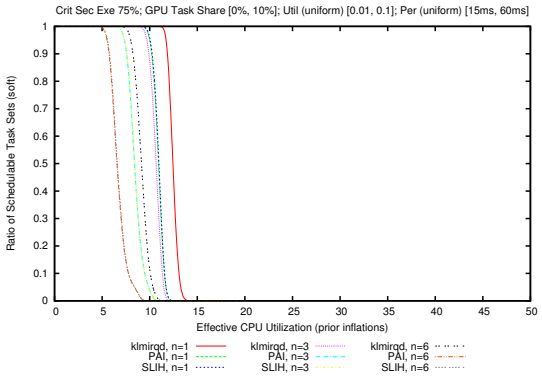

Figure 155. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

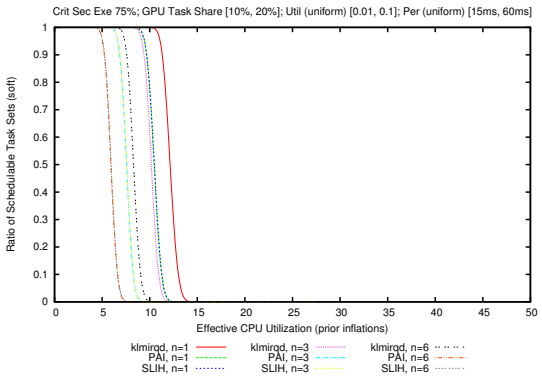

Figure 156. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

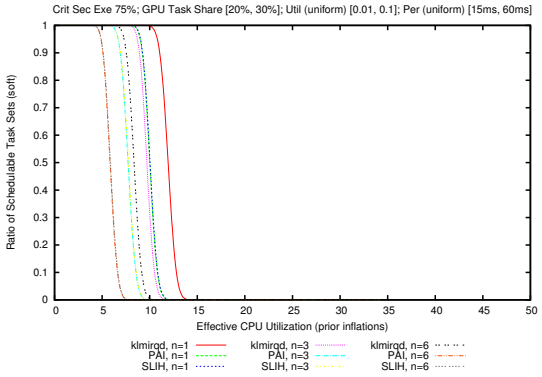

Figure 157. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

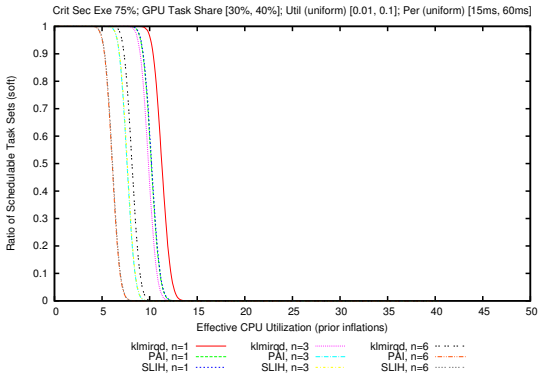

Figure 158. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

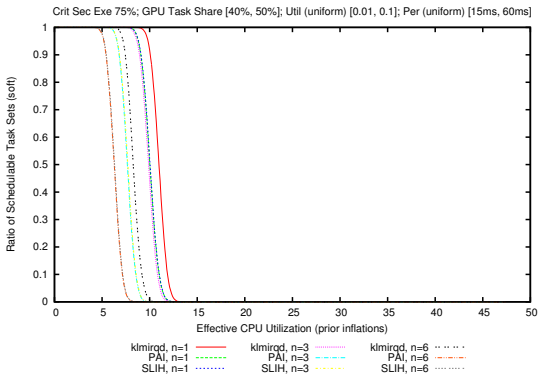

Figure 159. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

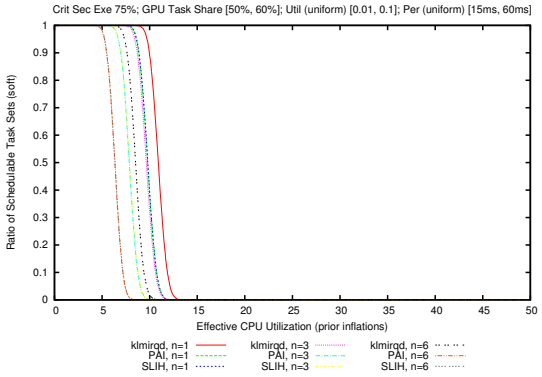

Figure 160. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

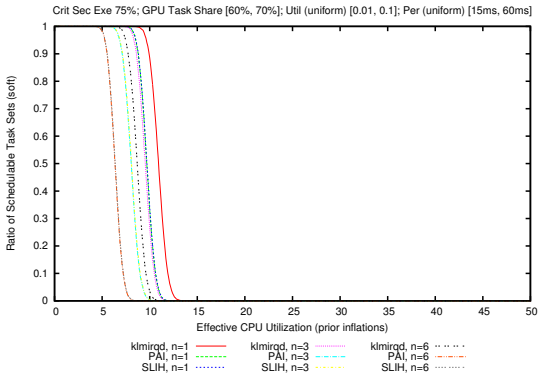

Figure 161. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

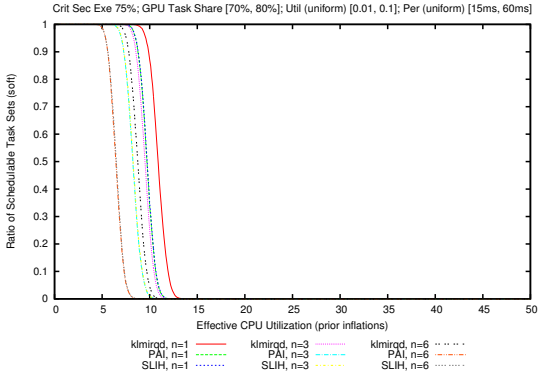

Figure 162. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

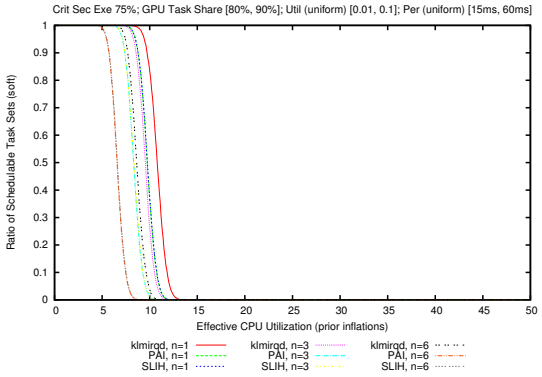

Figure 163. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

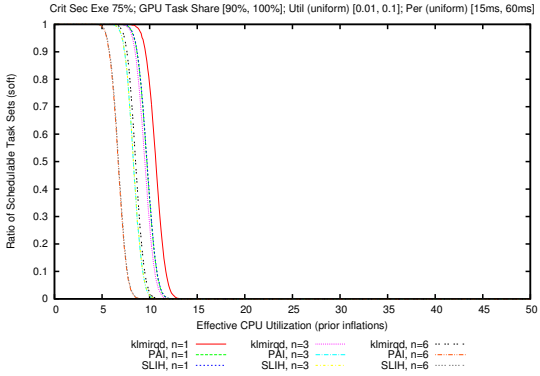

Figure 164. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

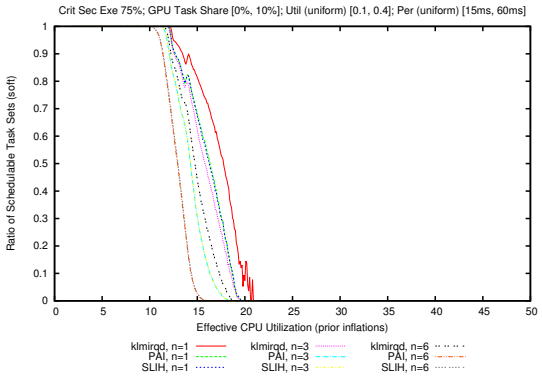

Figure 165. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

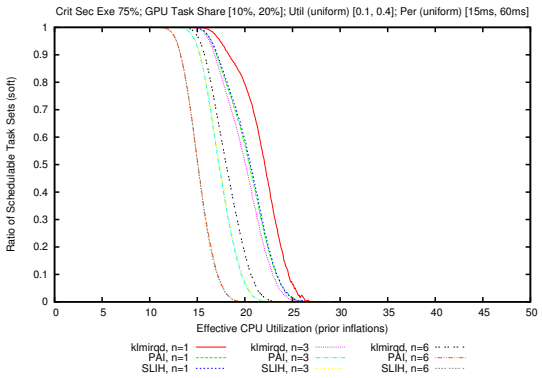

Figure 166. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

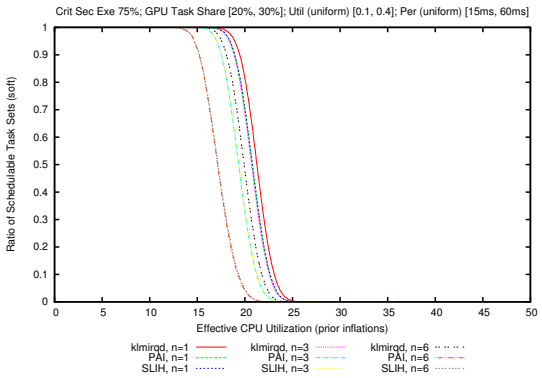

Figure 167. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

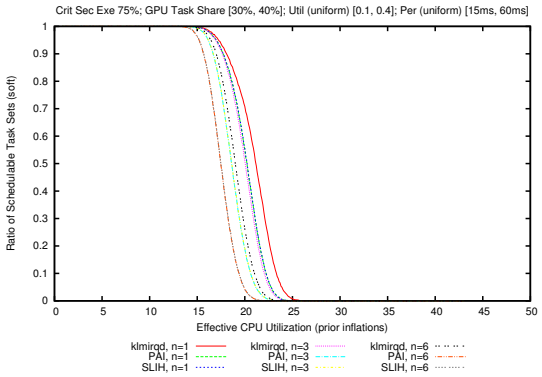

Figure 168. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

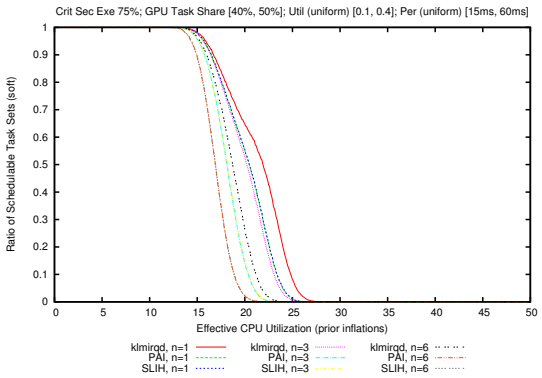

Figure 169. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

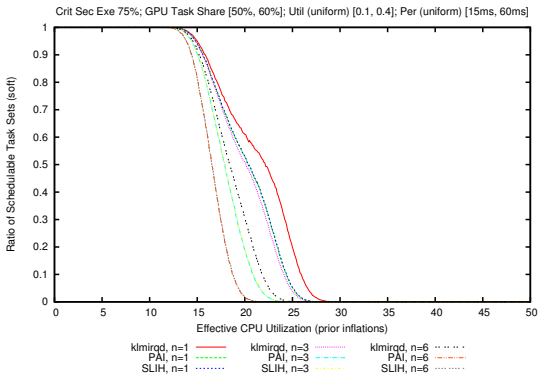

Figure 170. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

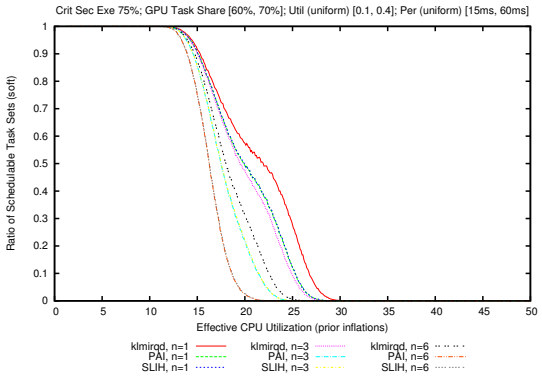

Figure 171. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

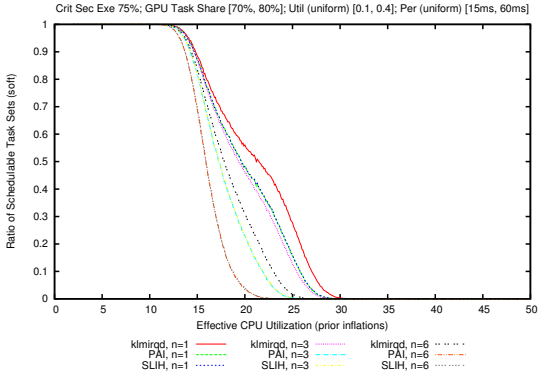

Figure 172. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

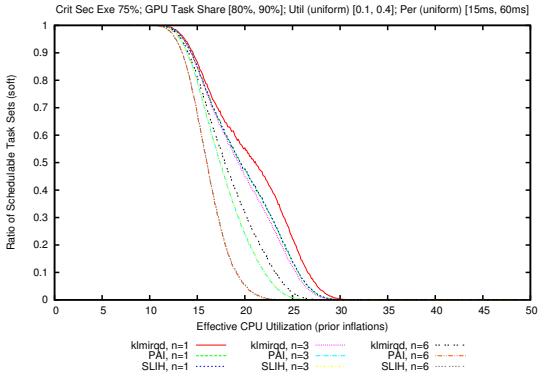

Figure 173. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

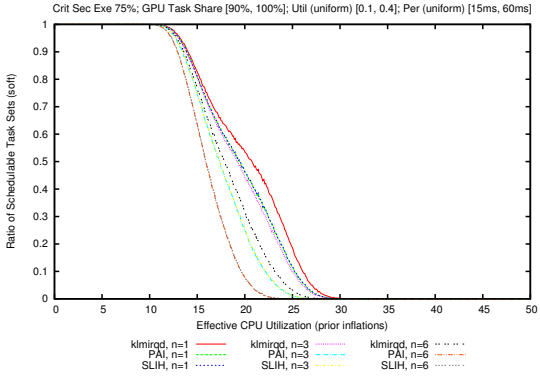

Figure 174. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

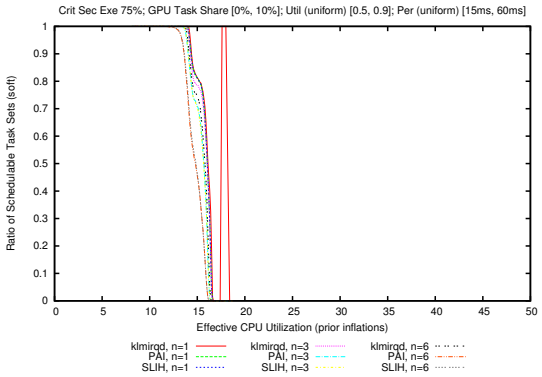

Figure 175. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

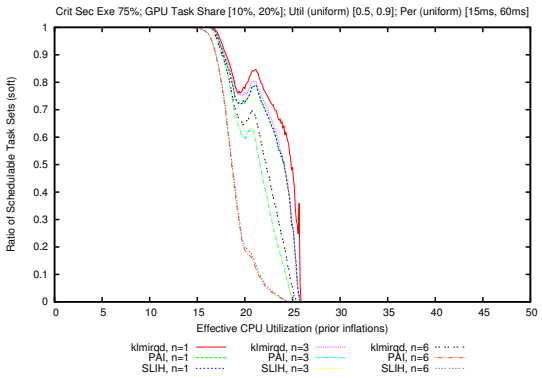

Figure 176. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

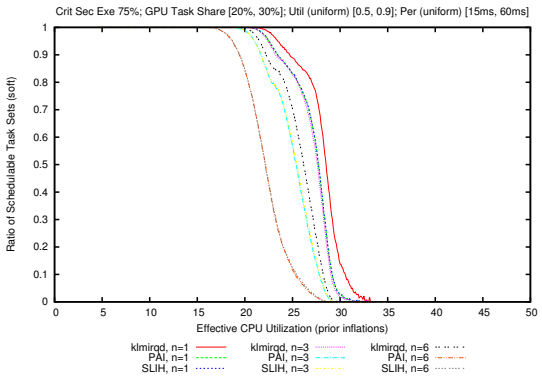

Figure 177. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

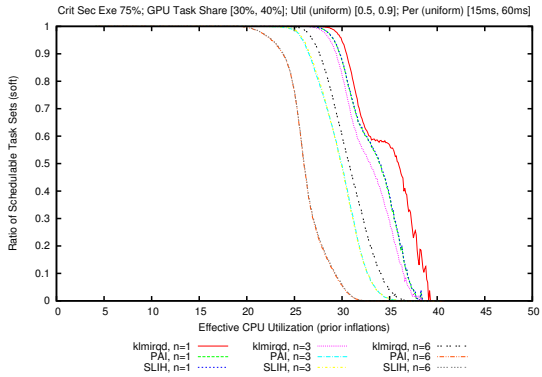

Figure 178. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

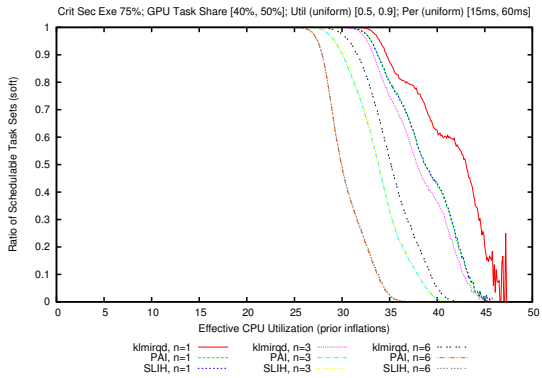

Figure 179. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

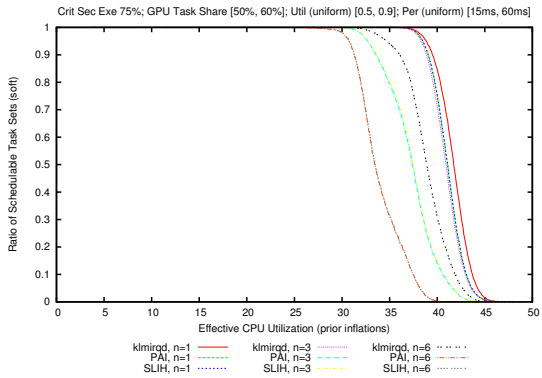

Figure 180. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

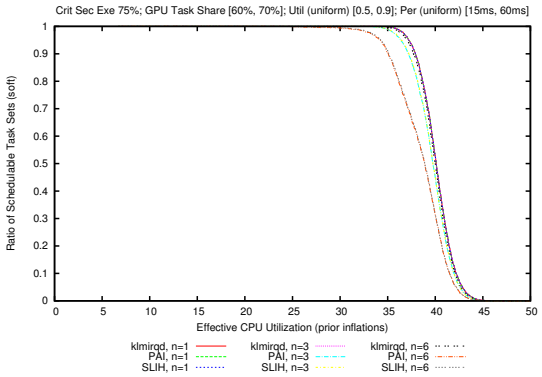

Figure 181. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

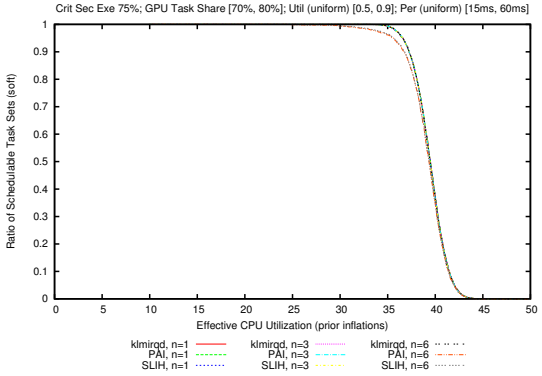

Figure 182. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

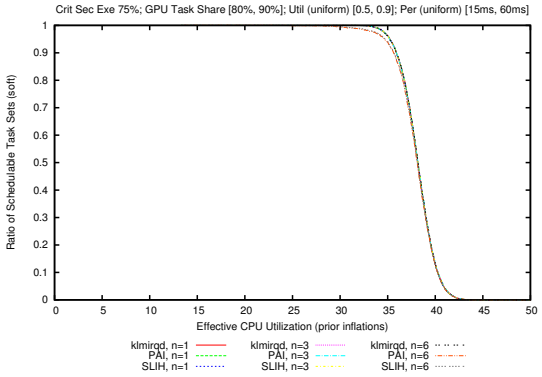

Figure 183. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

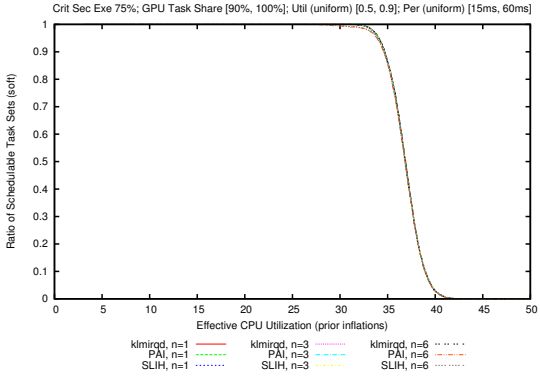

Figure 184. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

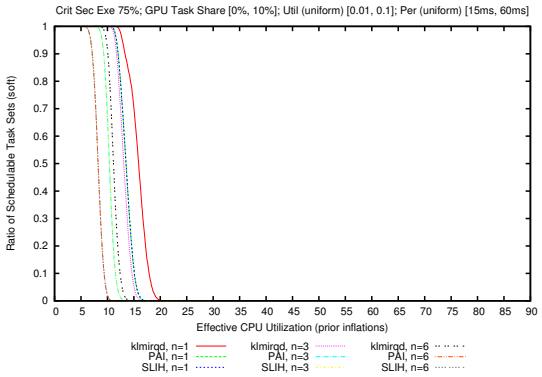

Figure 185. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

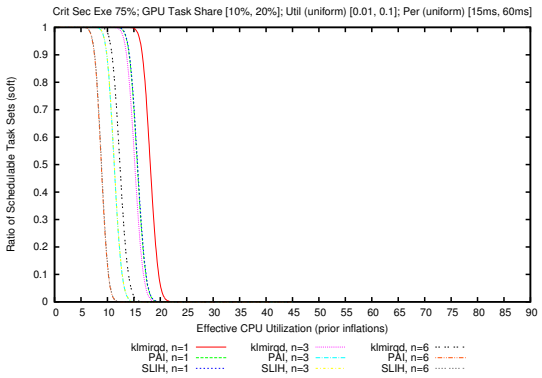

Figure 186. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

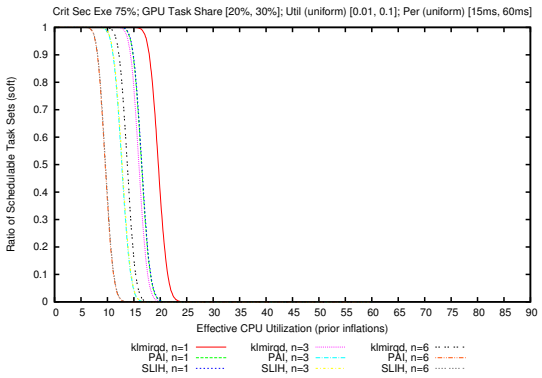

Figure 187. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

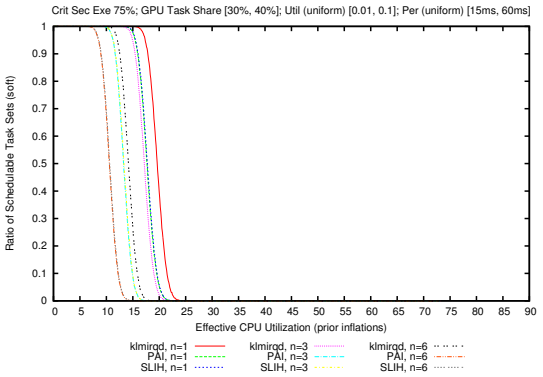

Figure 188. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

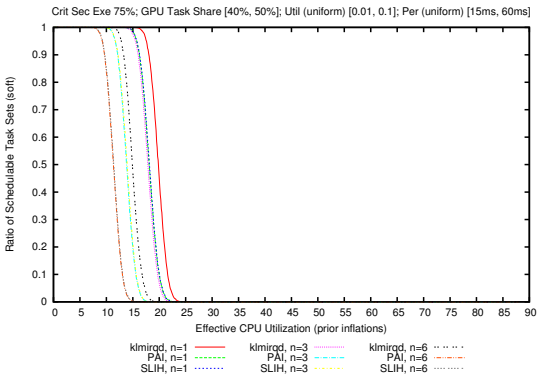

Figure 189. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

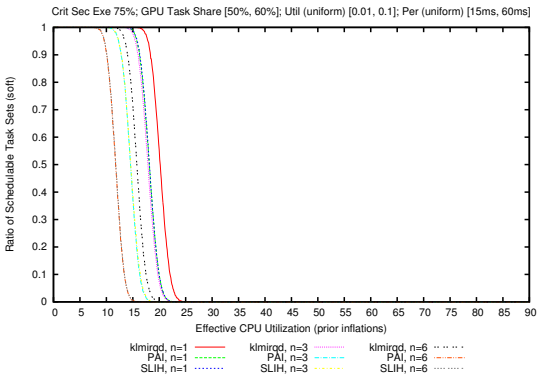

Figure 190. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

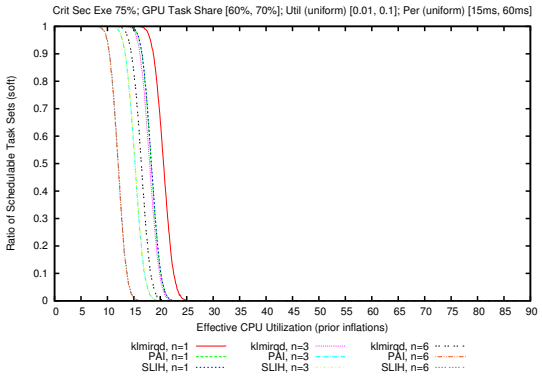

Figure 191. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

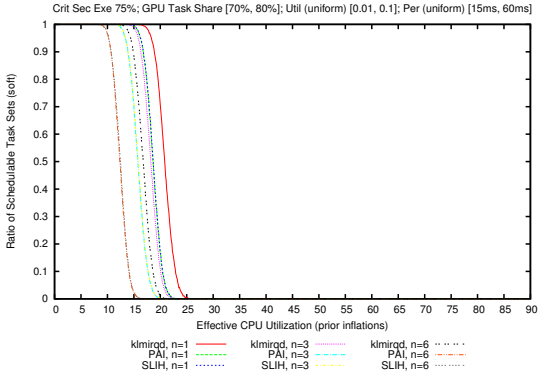

Figure 192. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

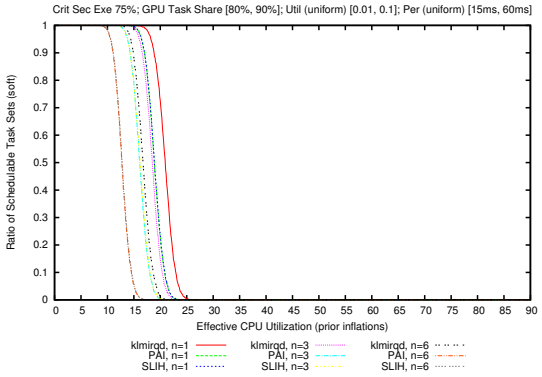

Figure 193. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

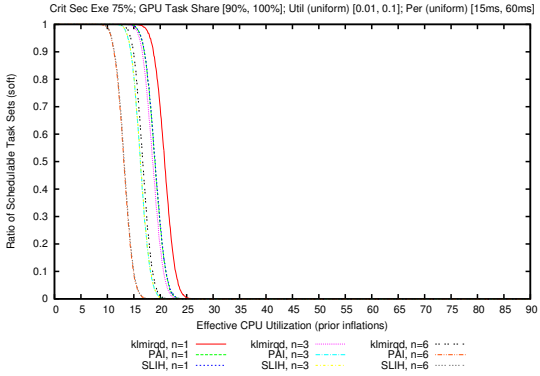

Figure 194. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

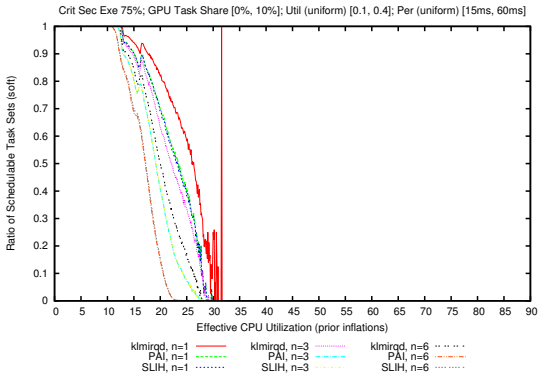

Figure 195. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

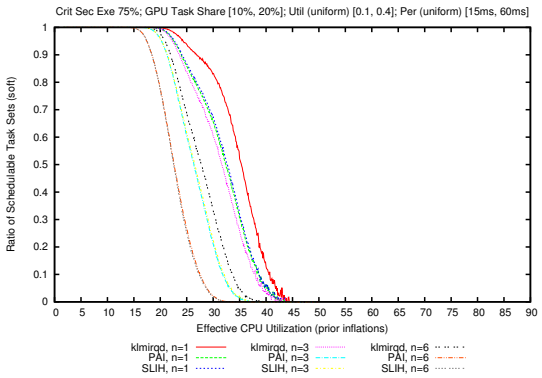

Figure 196. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

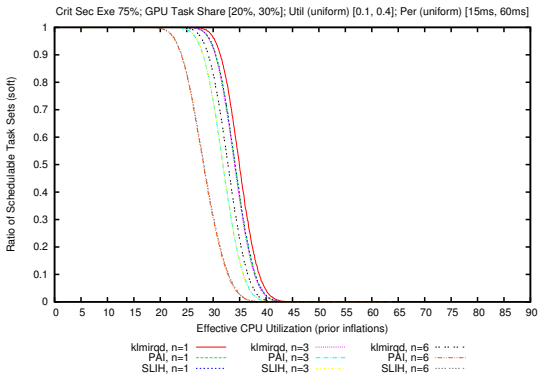

Figure 197. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

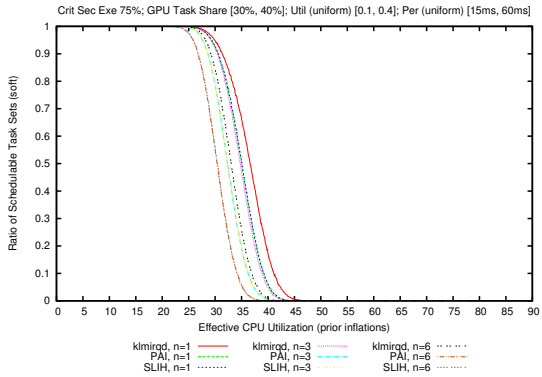

Figure 198. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

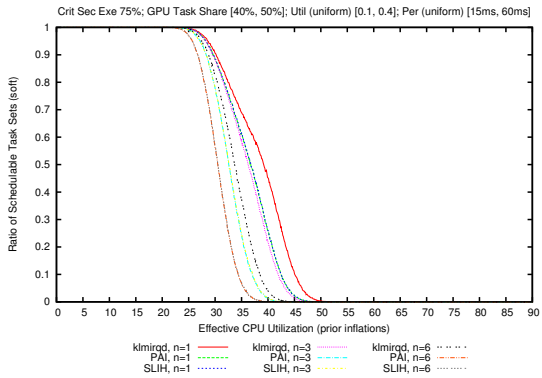

Figure 199. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

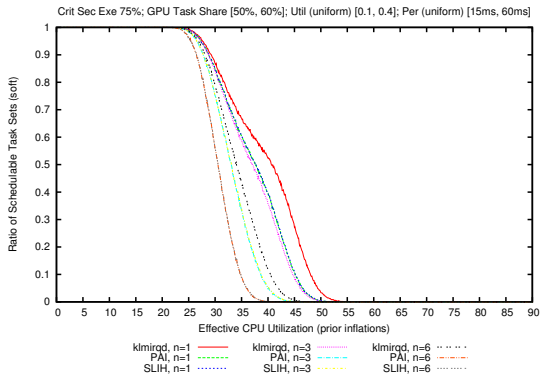

Figure 200. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

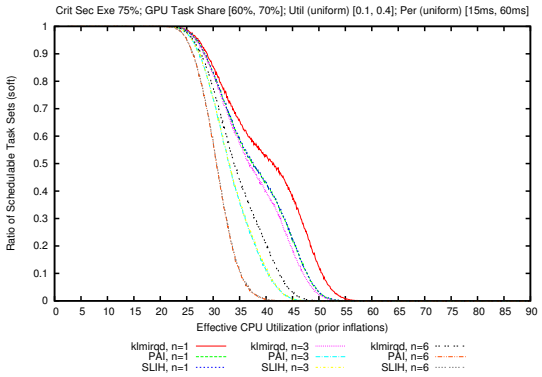

Figure 201. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

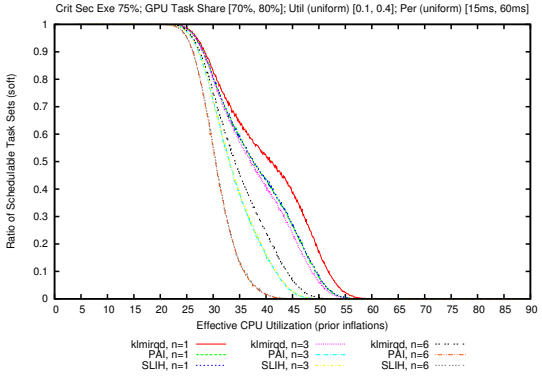

Figure 202. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

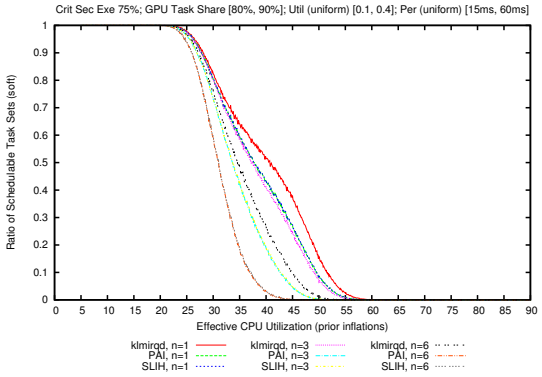

Figure 203. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

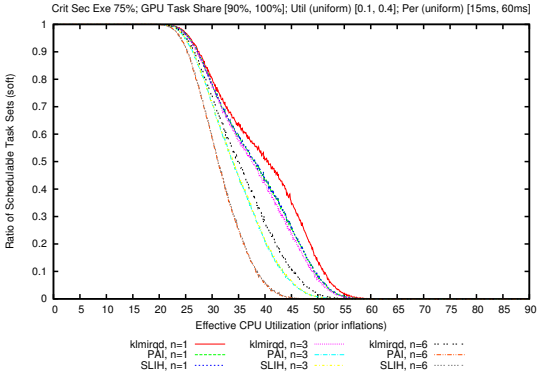

Figure 204. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

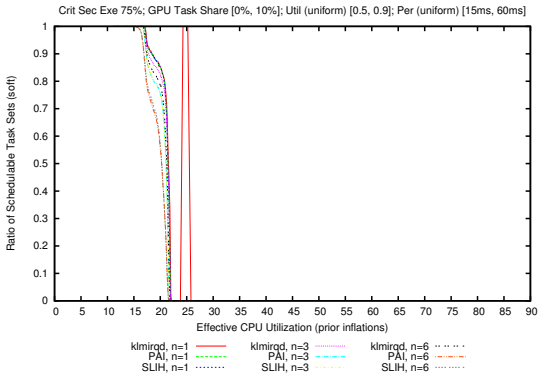

Figure 205. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

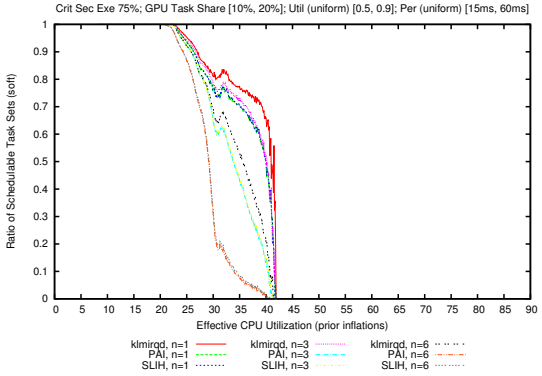

Figure 206. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

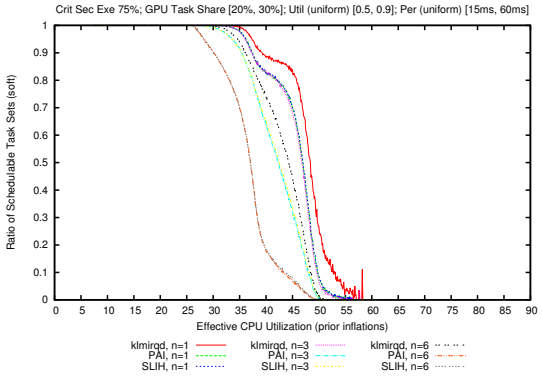

Figure 207. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

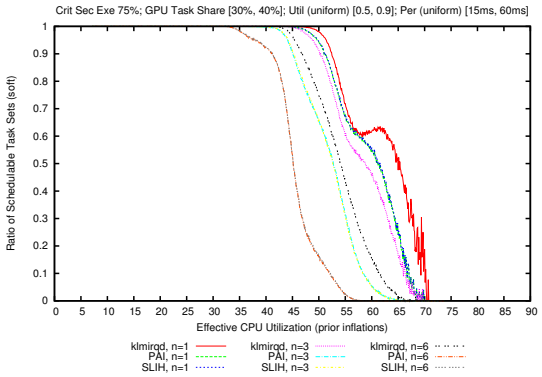

Figure 208. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

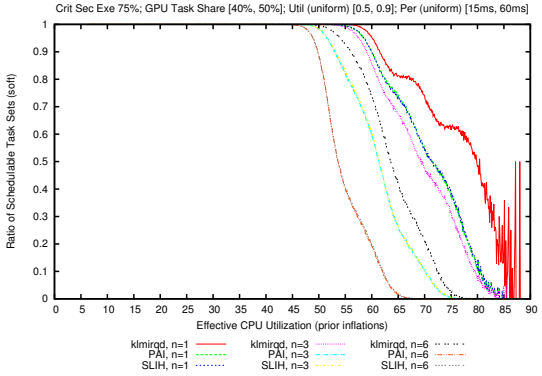

Figure 209. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

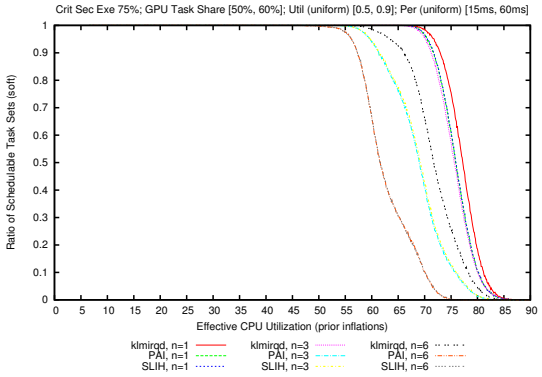

Figure 210. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

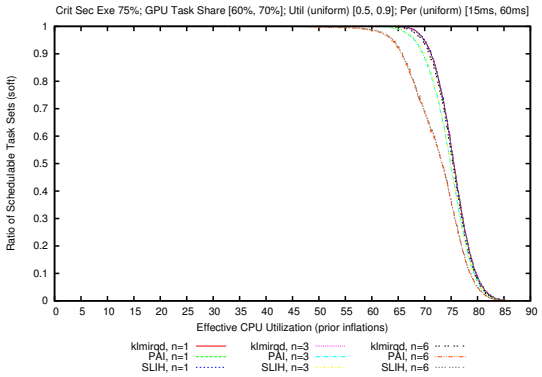

Figure 211. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

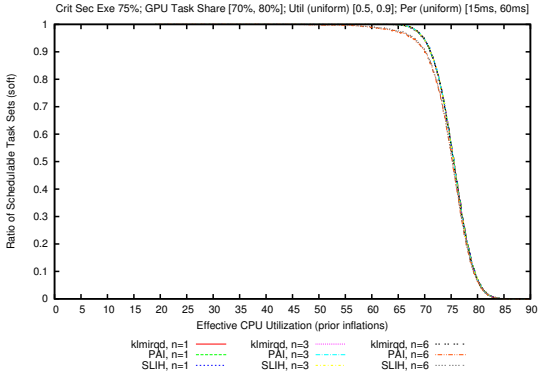

Figure 212. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

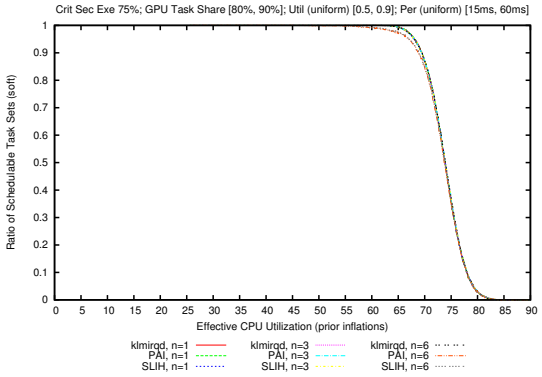

Figure 213. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

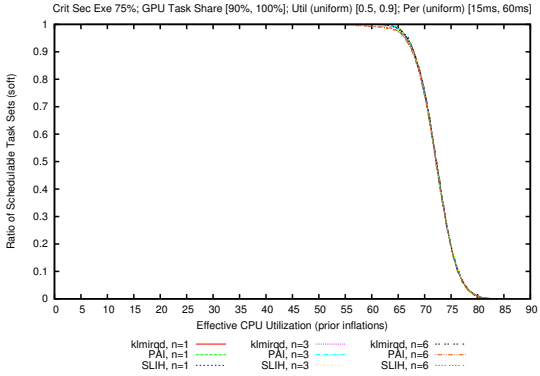

Figure 214. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

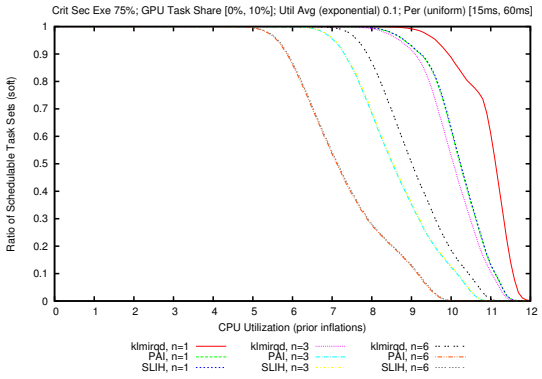

Figure 215. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

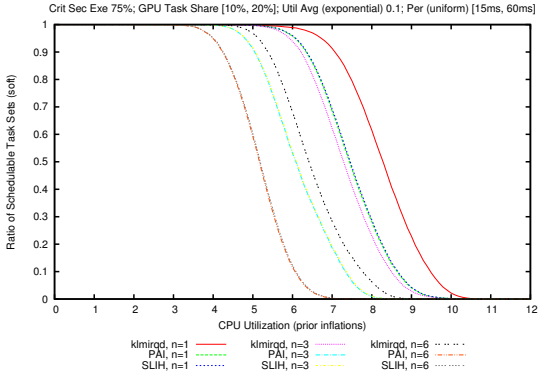

Figure 216. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

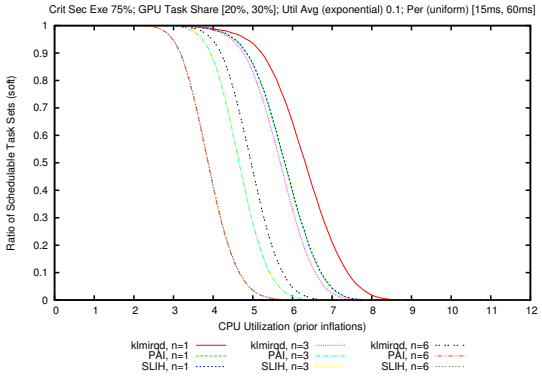

Figure 217. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

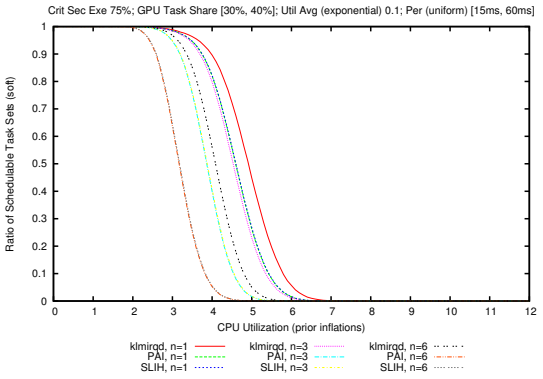

Figure 218. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

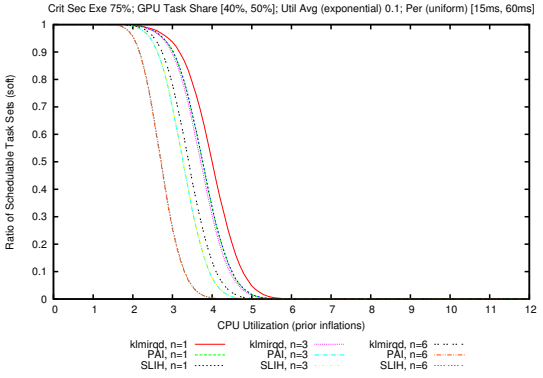

Figure 219. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

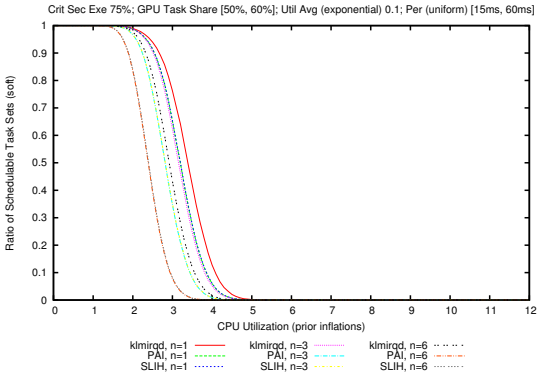

Figure 220. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

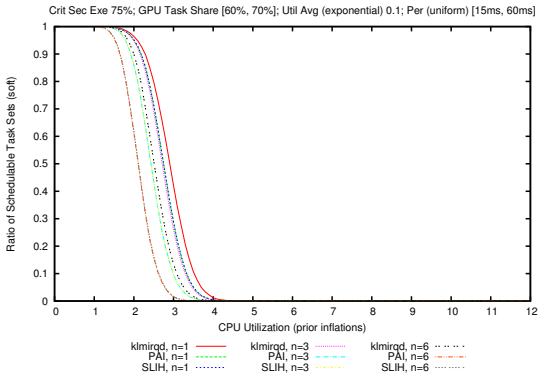

Figure 221. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

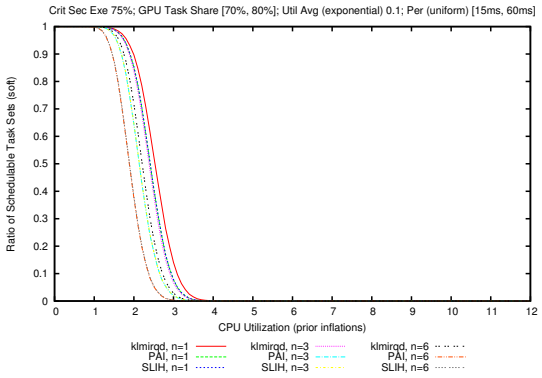

Figure 222. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

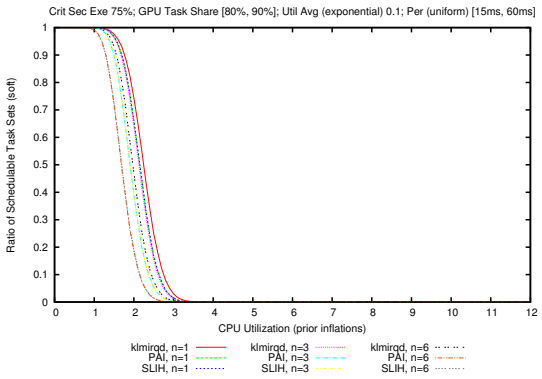

Figure 223. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

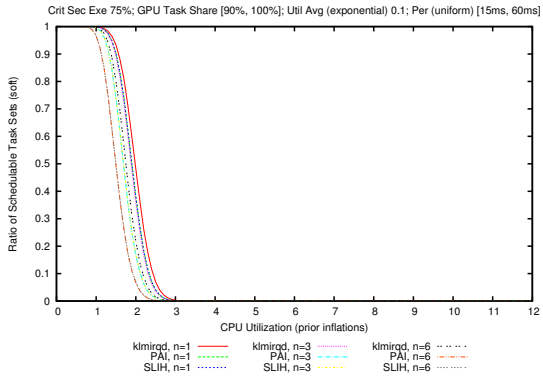

Figure 224. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

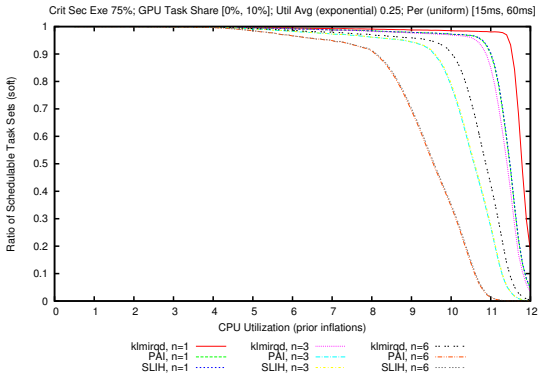

Figure 225. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

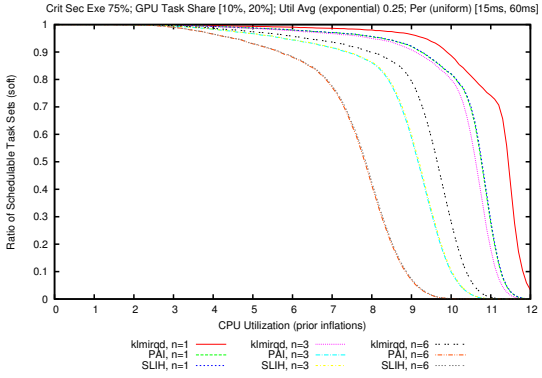

Figure 226. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

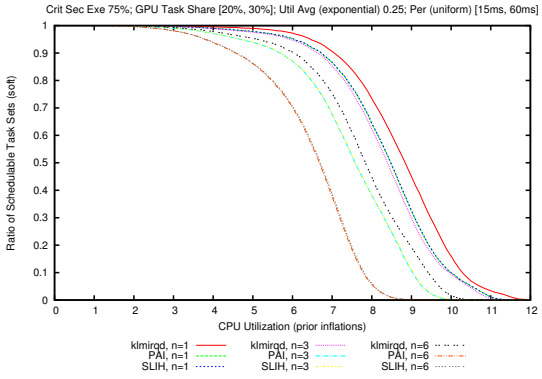

Figure 227. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

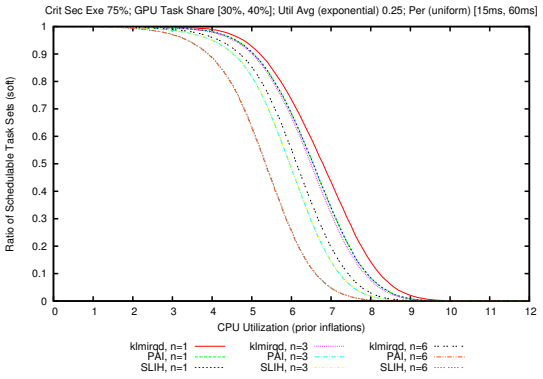

Figure 228. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

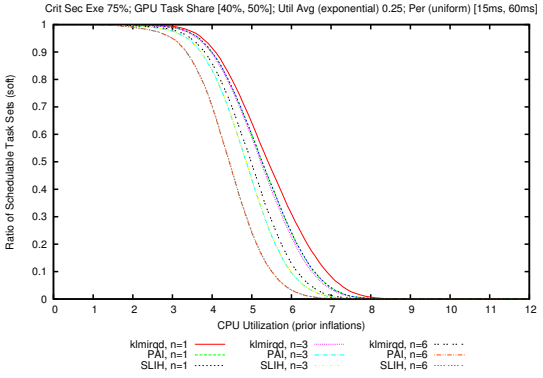

Figure 229. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

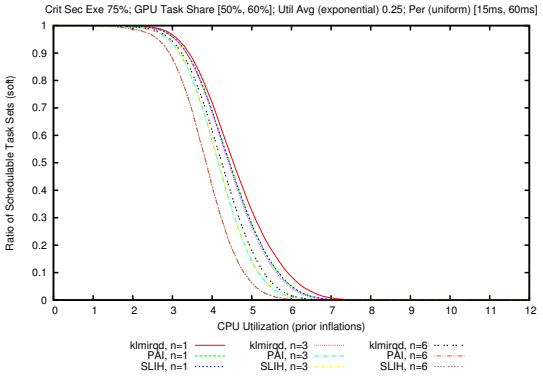

Figure 230. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

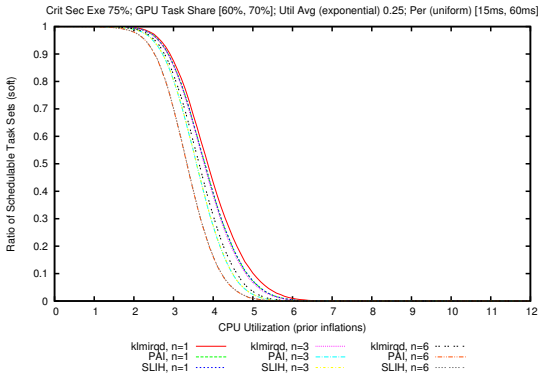

Figure 231. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

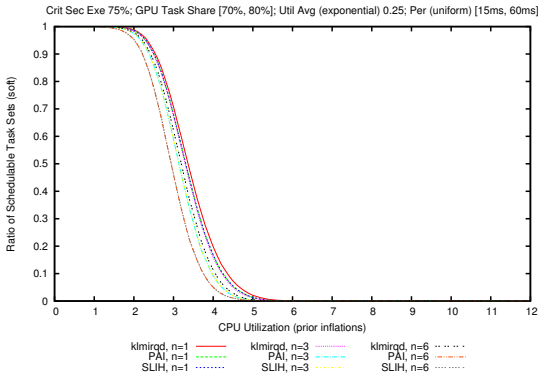

Figure 232. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

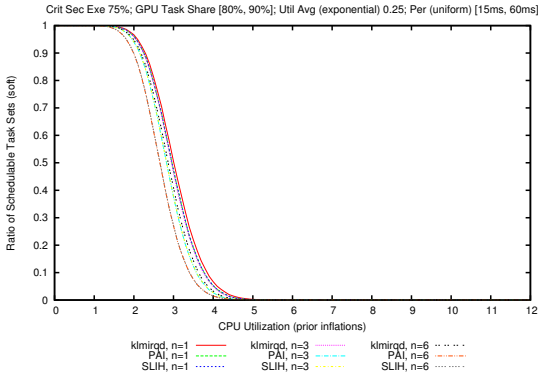

Figure 233. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

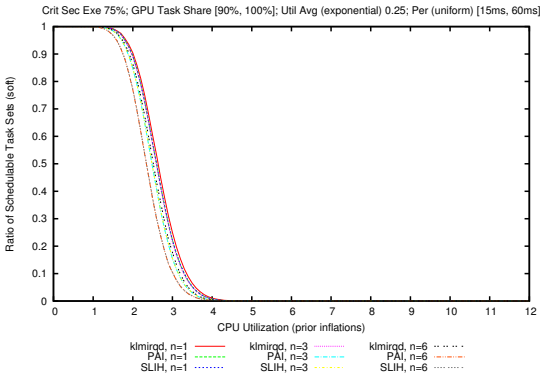

Figure 234. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

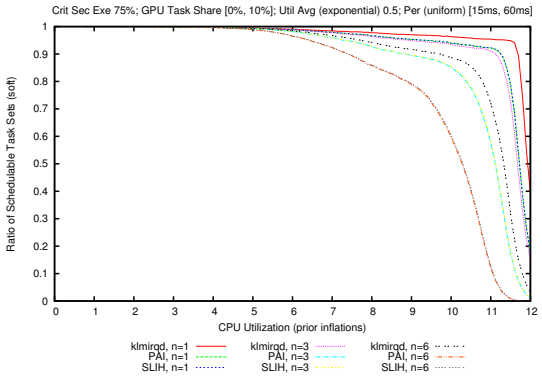

Figure 235. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

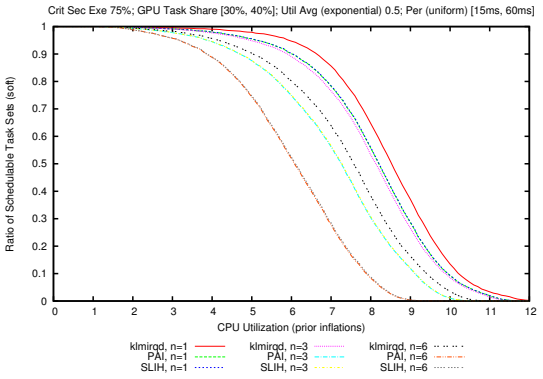

Figure 238. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

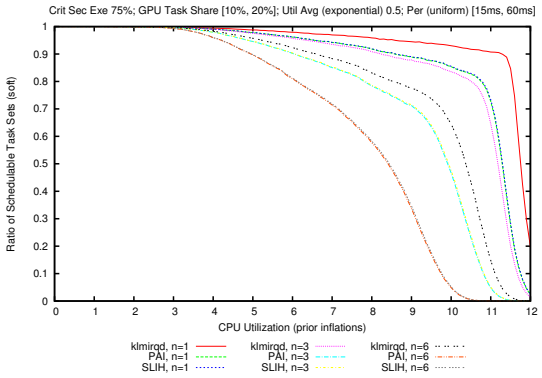

Figure 236. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

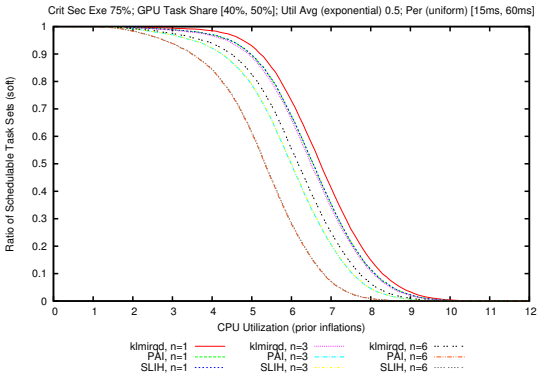

Figure 239. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

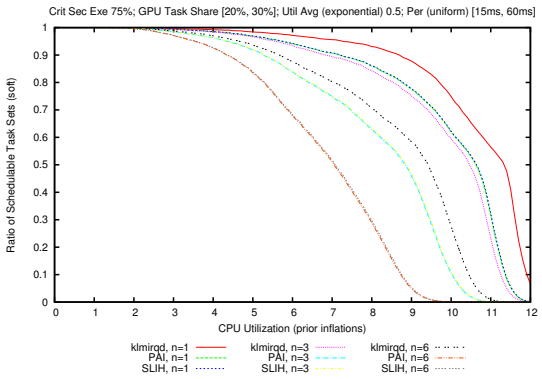

Figure 237. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

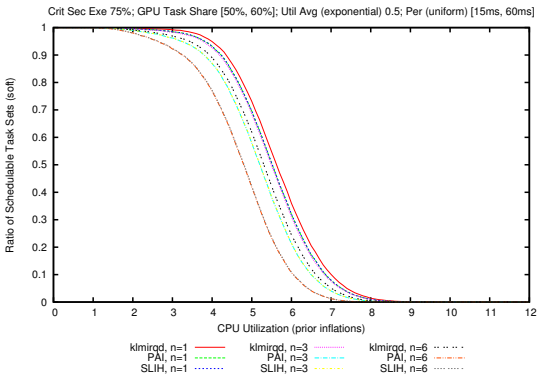

Figure 240. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

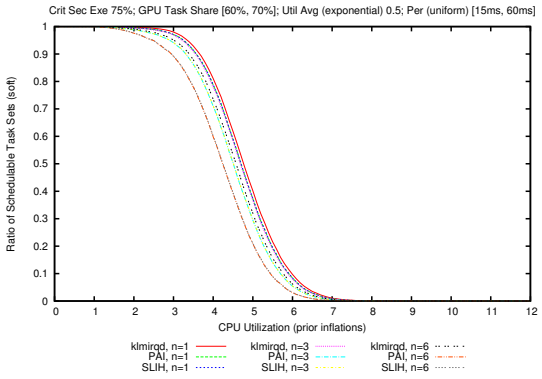

Figure 241. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

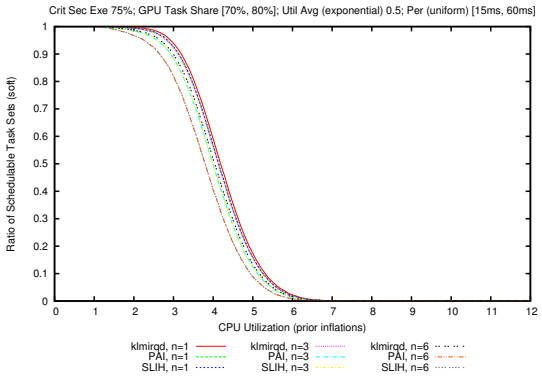

Figure 242. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

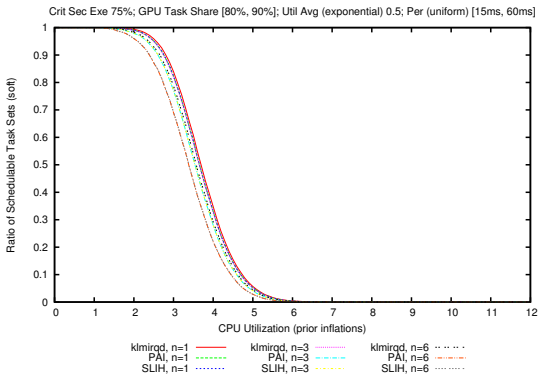

Figure 243. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

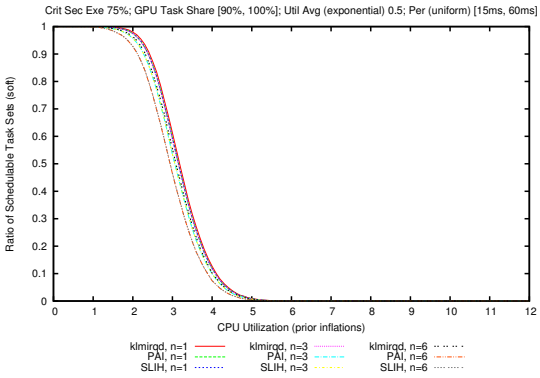

Figure 244. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

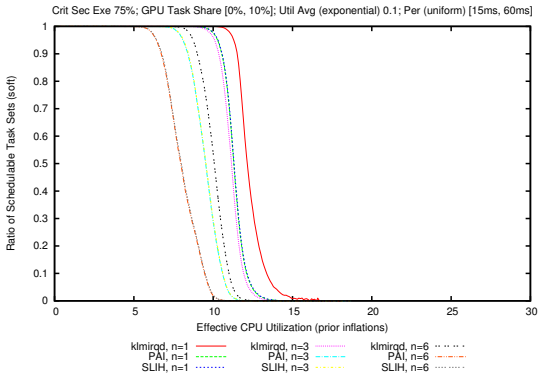

Figure 245. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

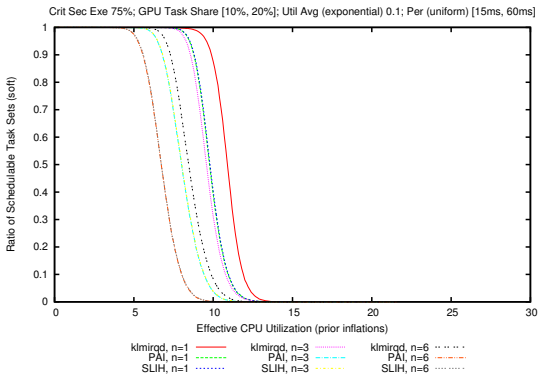

Figure 246. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

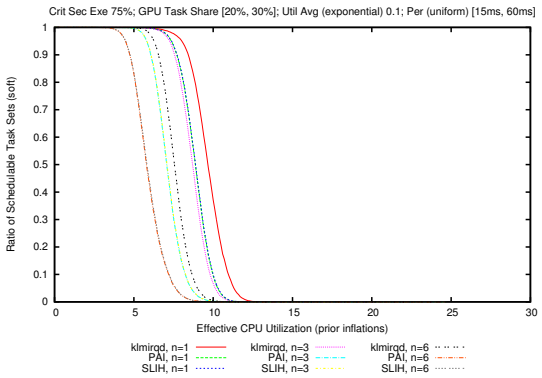

Figure 247. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

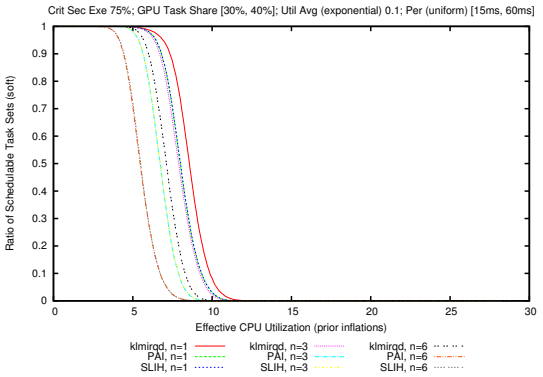

Figure 248. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

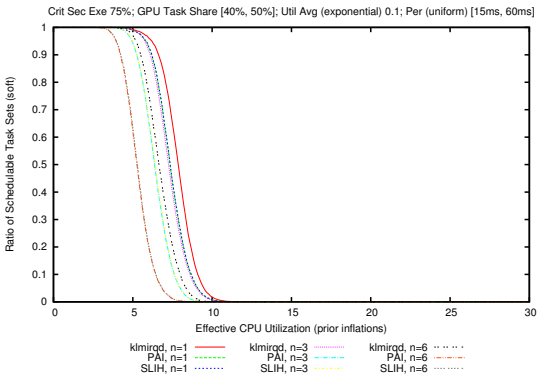

Figure 249. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

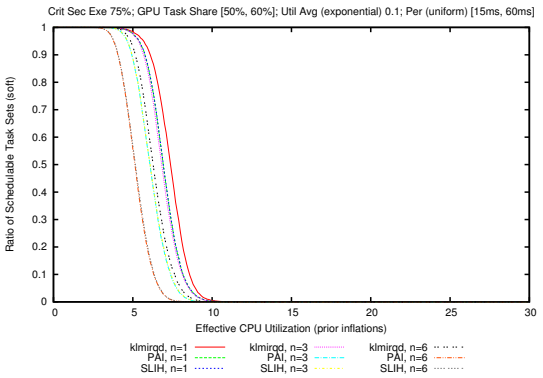

Figure 250. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

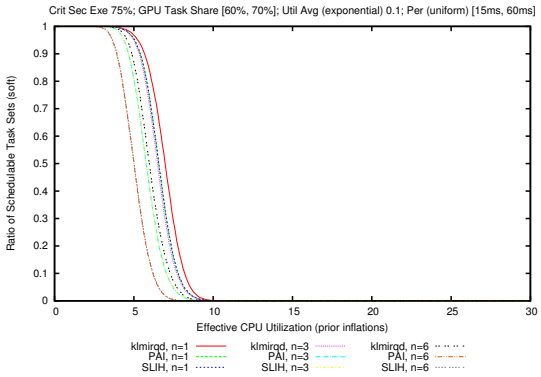

Figure 251. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

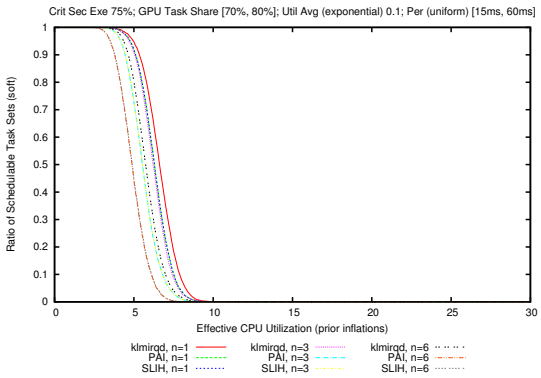

Figure 252. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

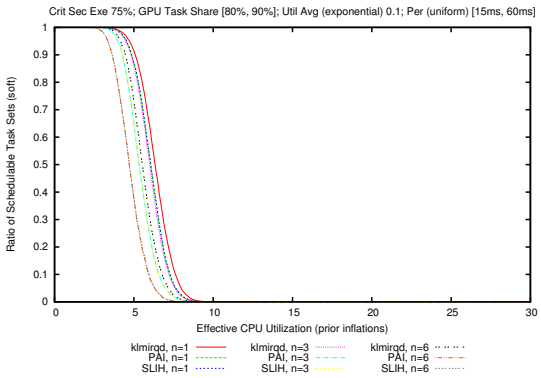

Figure 253. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

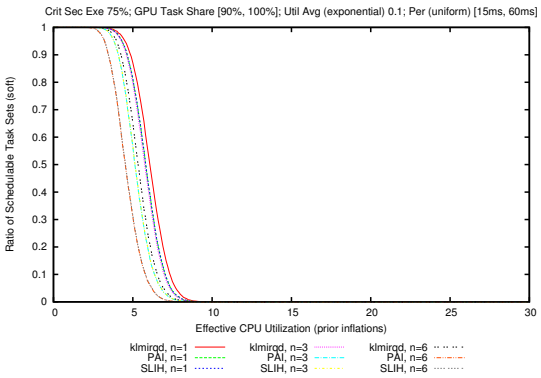

Figure 254. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

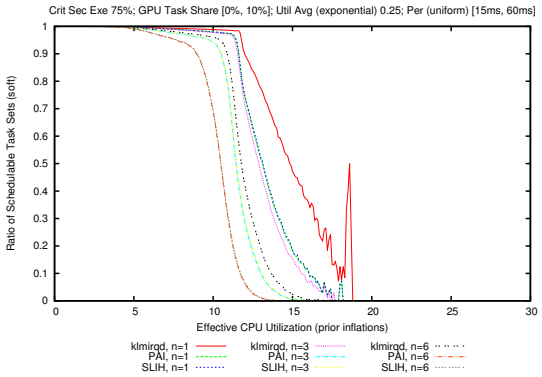

Figure 255. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

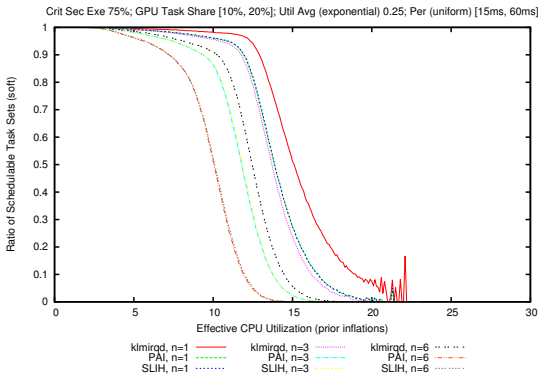

Figure 256. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

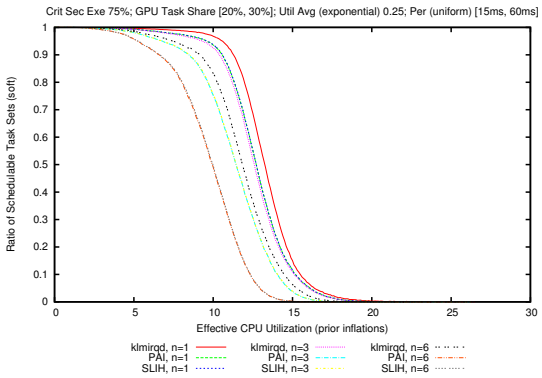

Figure 257. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

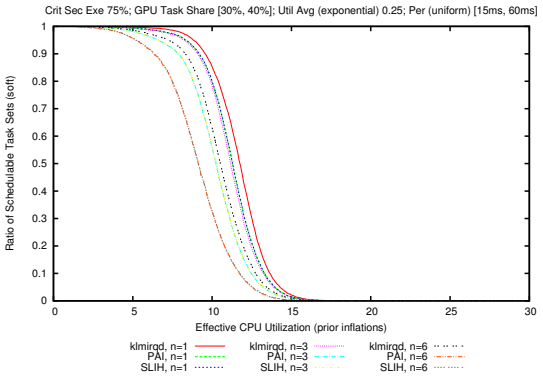

Figure 258. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

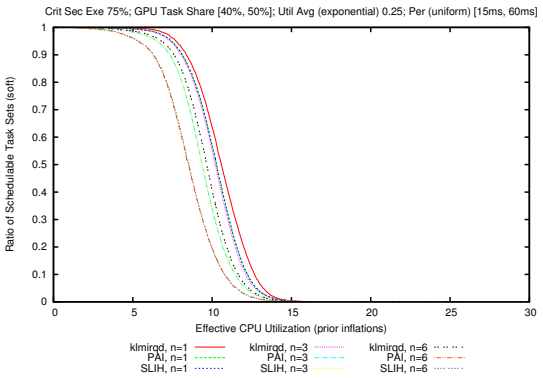

Figure 259. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

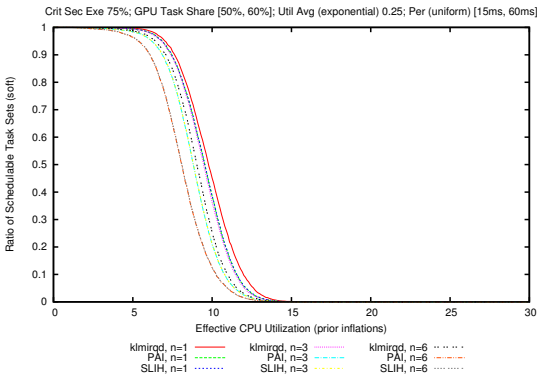

Figure 260. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

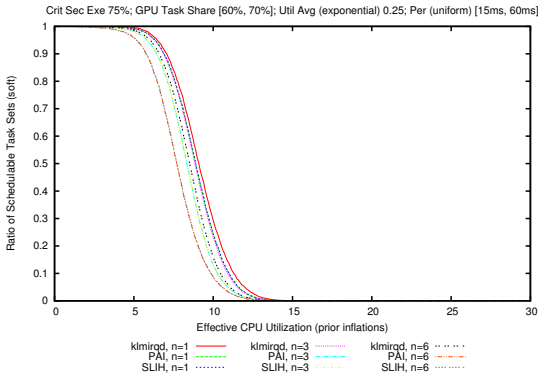

Figure 261. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

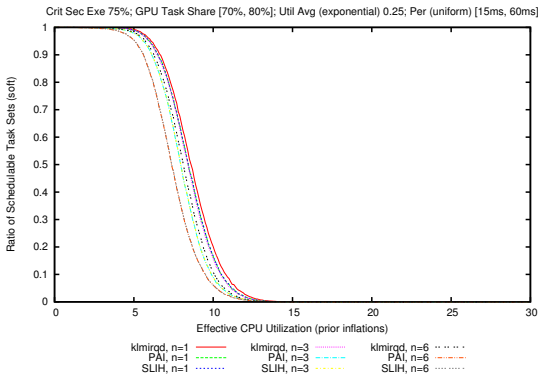

Figure 262. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

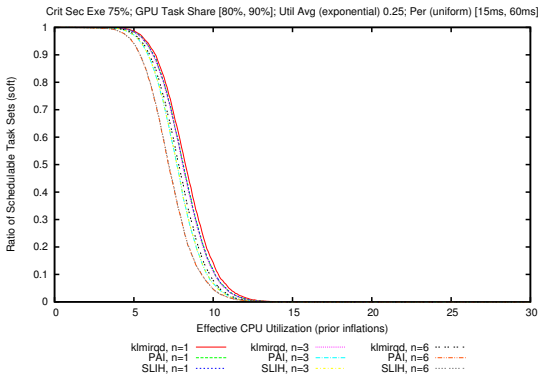

Figure 263. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

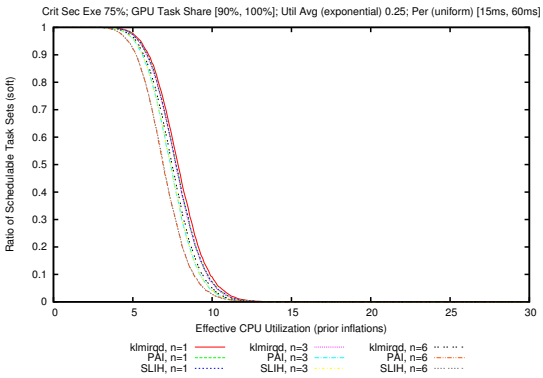

Figure 264. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

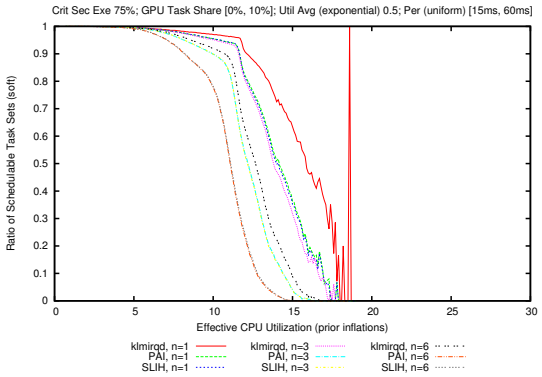

Figure 265. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

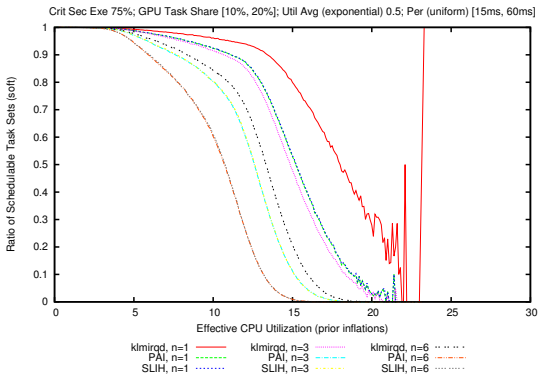

Figure 266. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

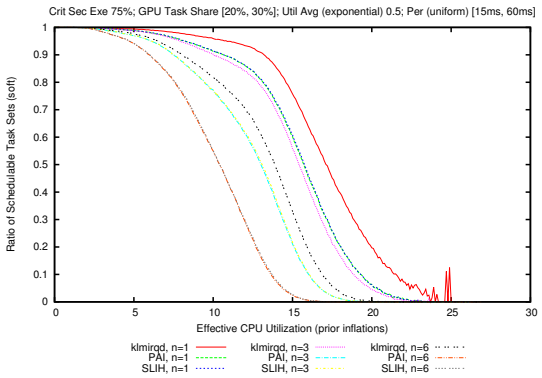

Figure 267. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

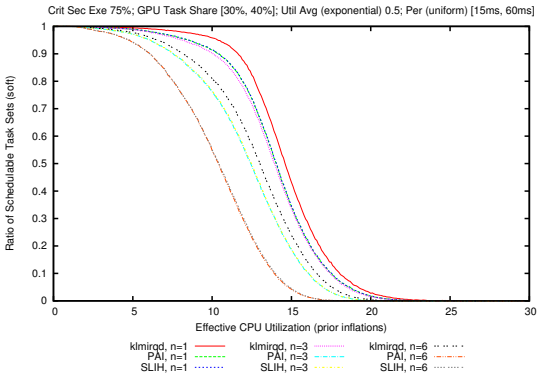

Figure 268. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

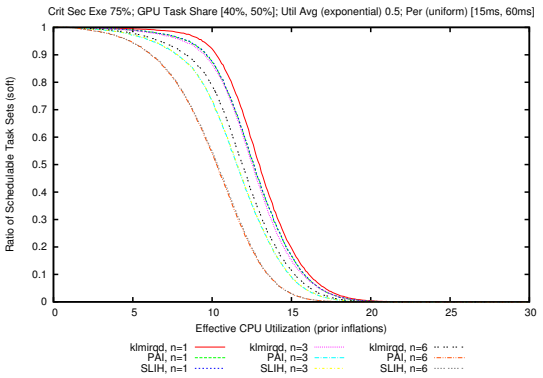

Figure 269. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

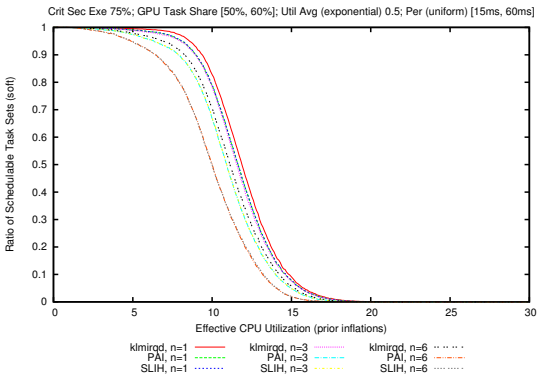

Figure 270. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

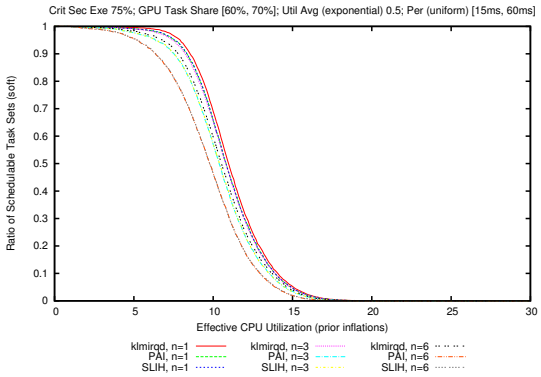

Figure 271. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

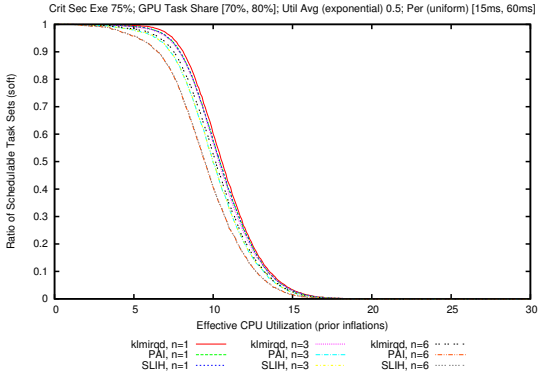

Figure 272. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

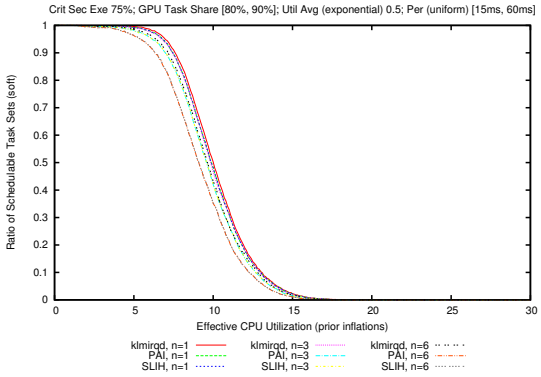

Figure 273. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

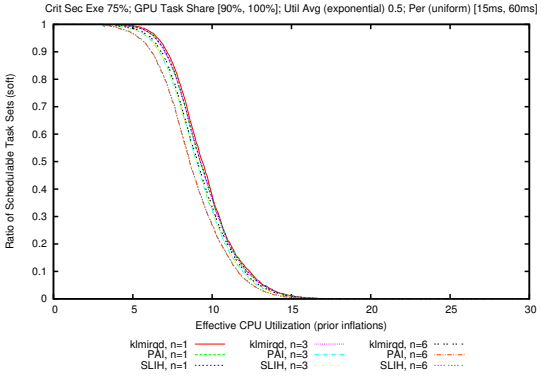

Figure 274. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

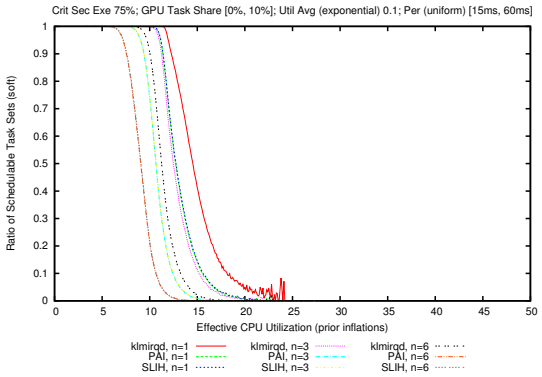

Figure 275. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

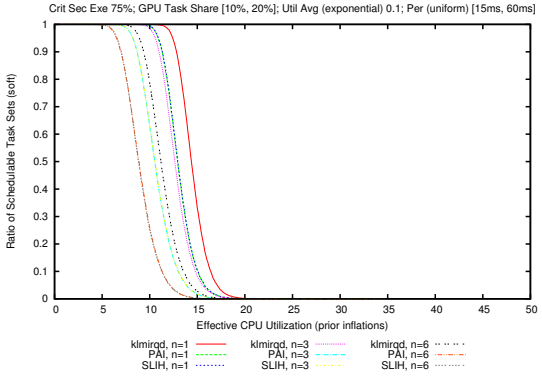

Figure 276. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

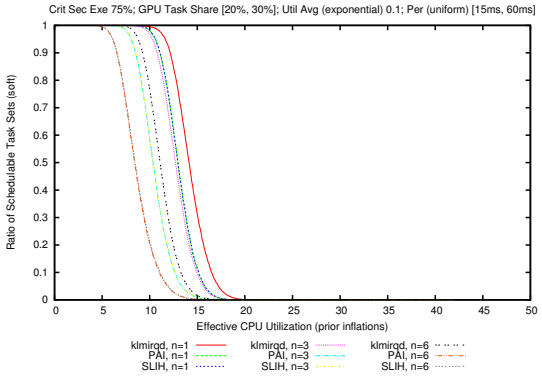

Figure 277. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

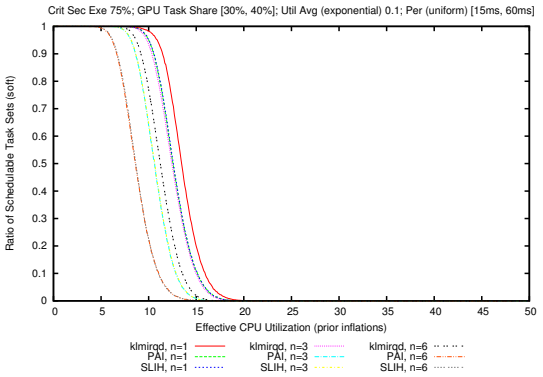

Figure 278. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

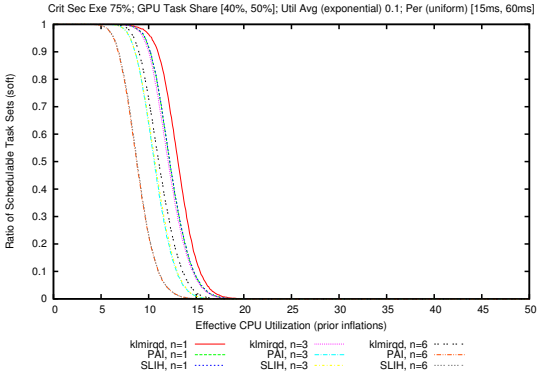

Figure 279. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

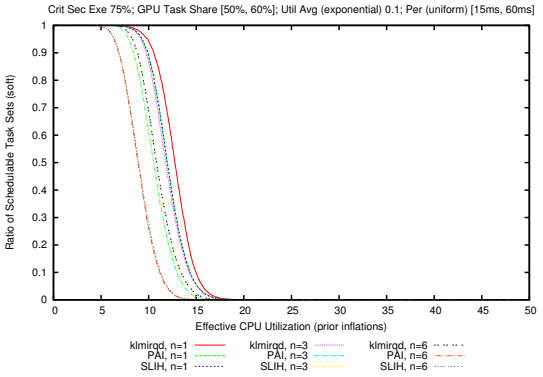

Figure 280. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

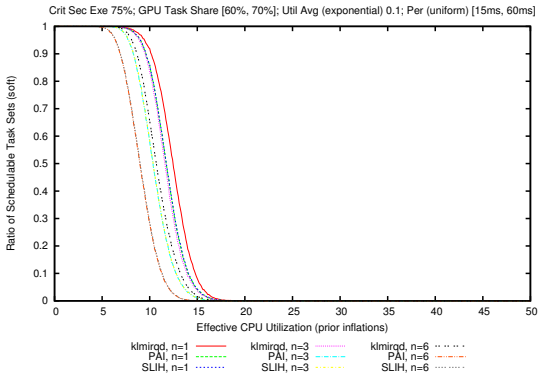

Figure 281. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

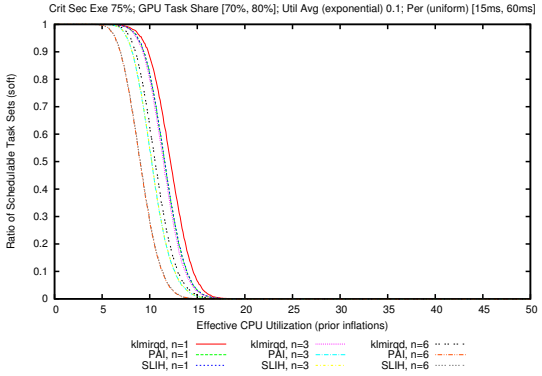

Figure 282. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

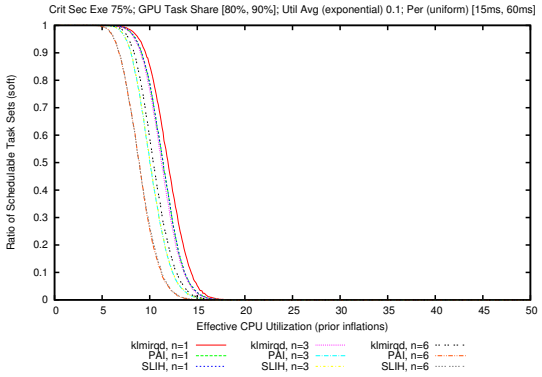

Figure 283. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

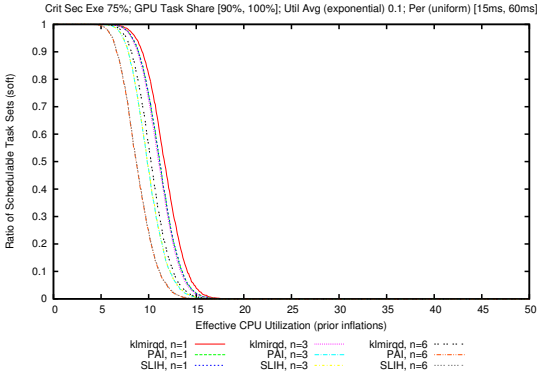

Figure 284. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

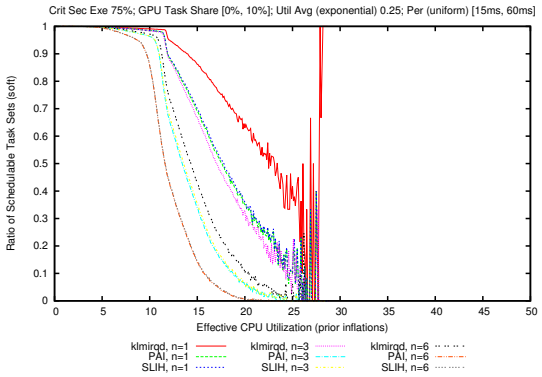

Figure 285. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

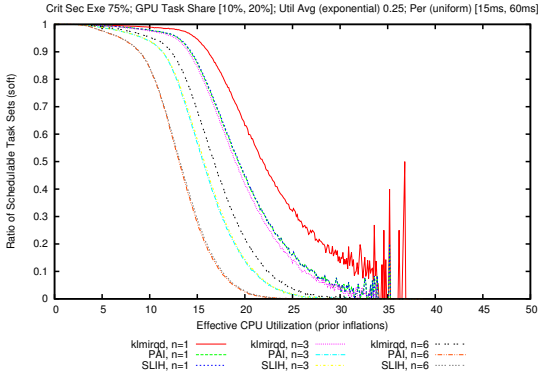

Figure 286. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

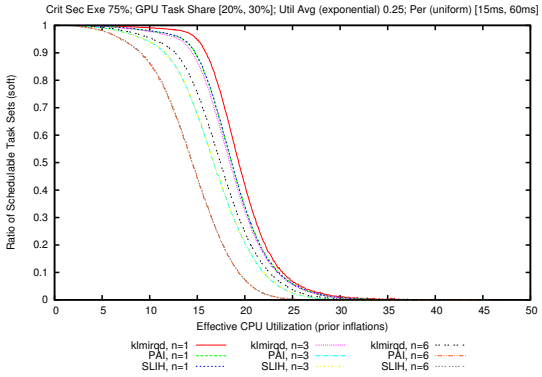

Figure 287. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

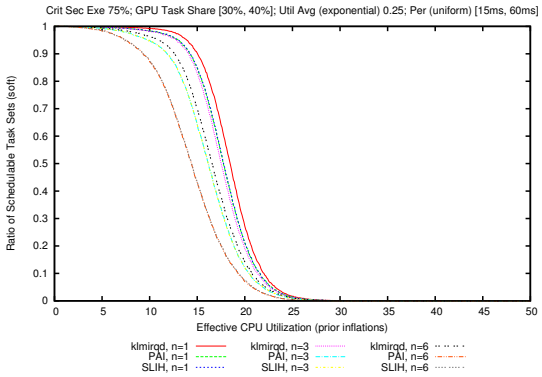

Figure 288. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

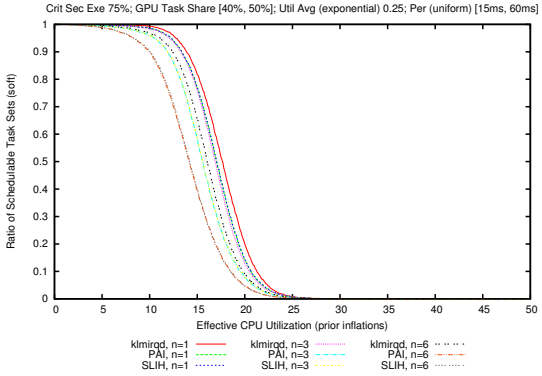

Figure 289. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

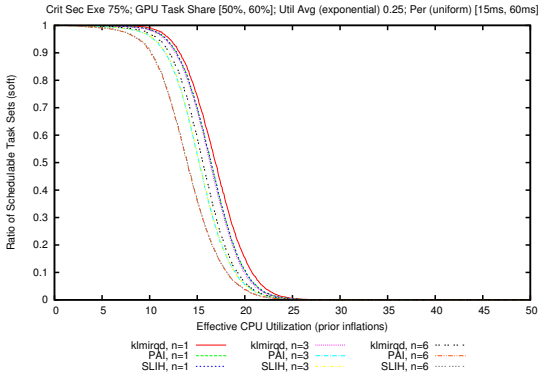

Figure 290. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

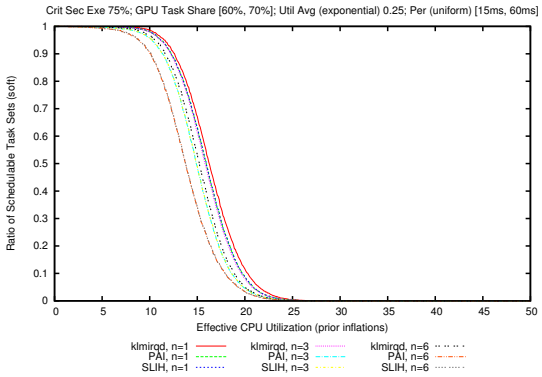

Figure 291. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

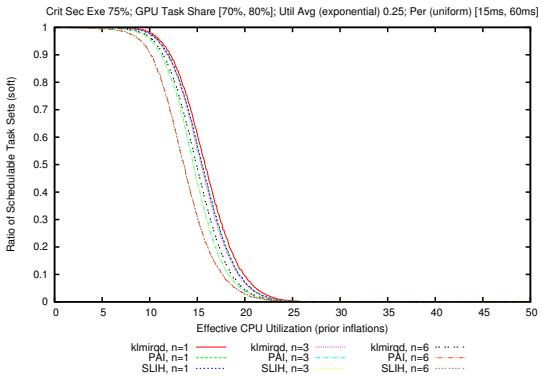

Figure 292. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

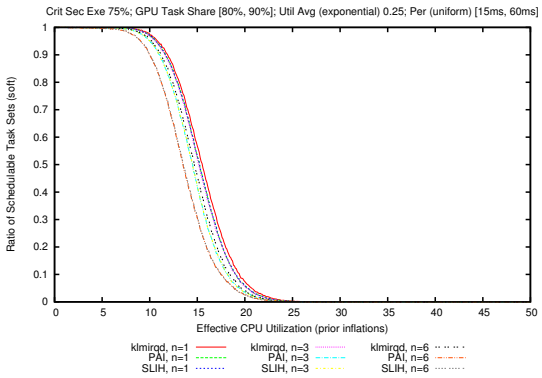

Figure 293. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

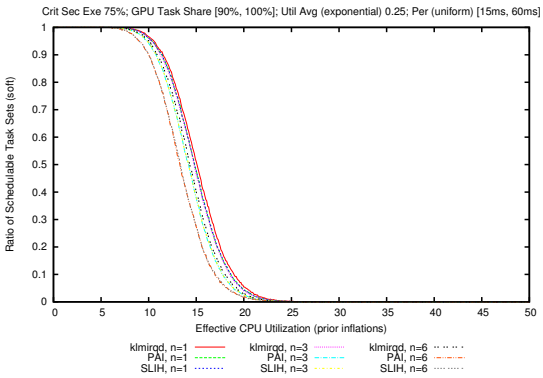

Figure 294. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

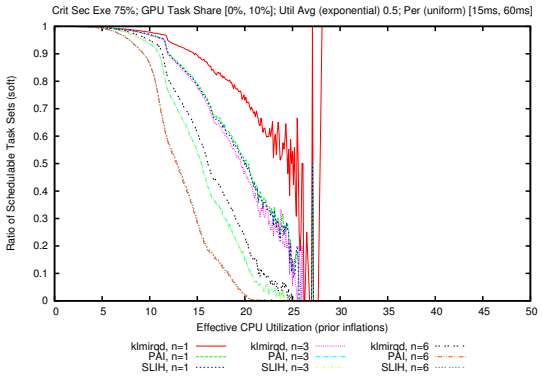

Figure 295. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

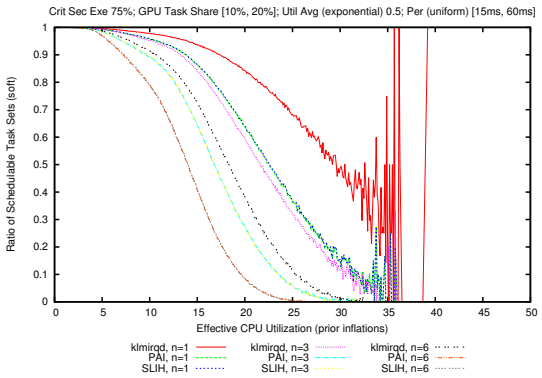

Figure 296. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

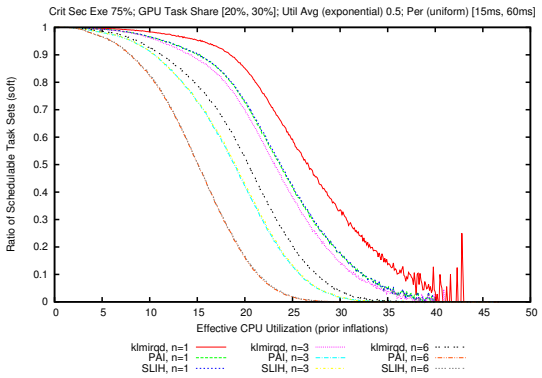

Figure 297. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

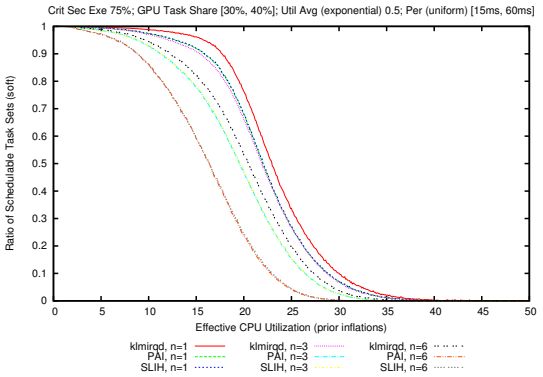

Figure 298. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

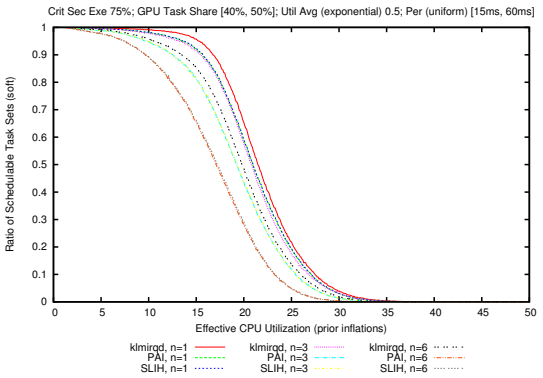

Figure 299. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

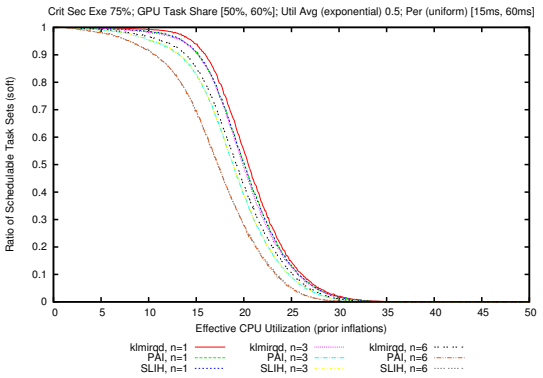

Figure 300. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

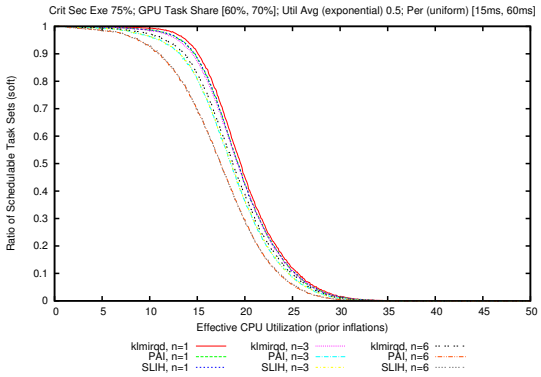

Figure 301. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

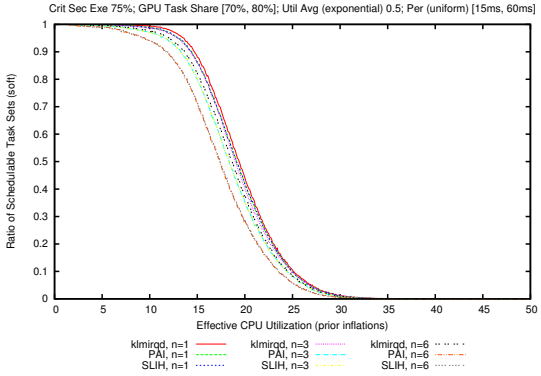

Figure 302. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

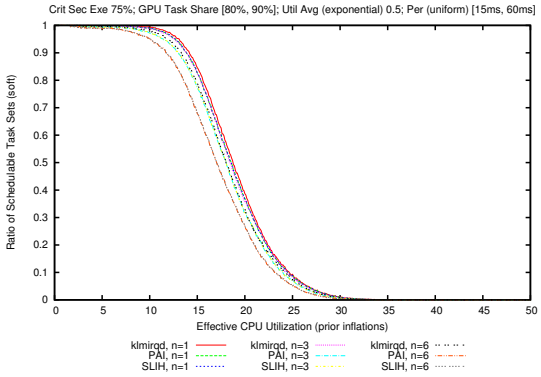

Figure 303. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

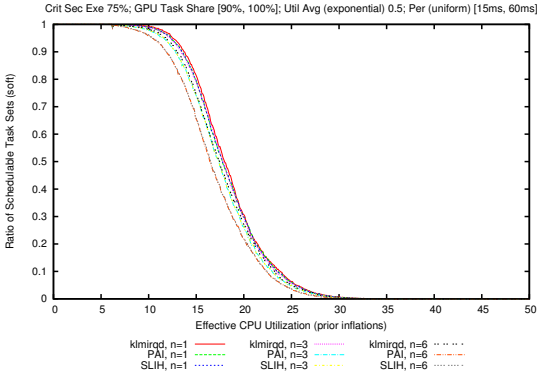

Figure 304. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

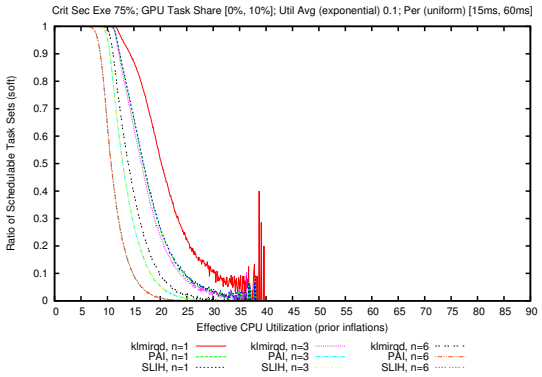

Figure 305. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

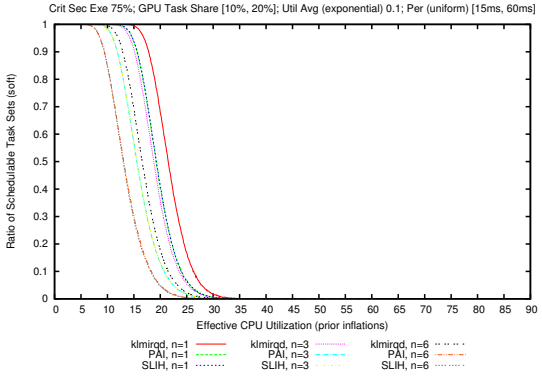

Figure 306. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

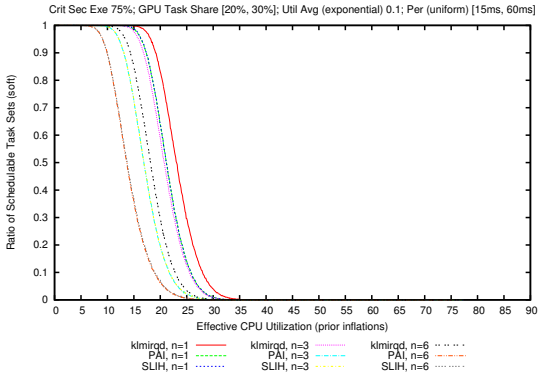

Figure 307. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

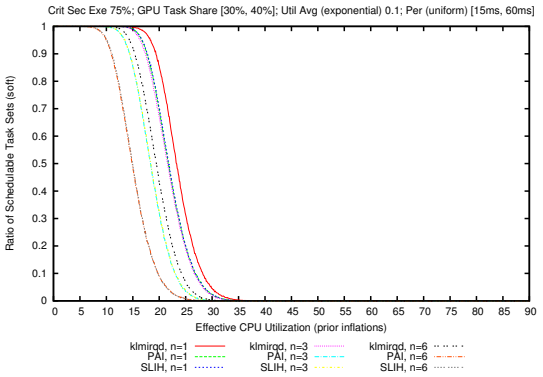

Figure 308. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

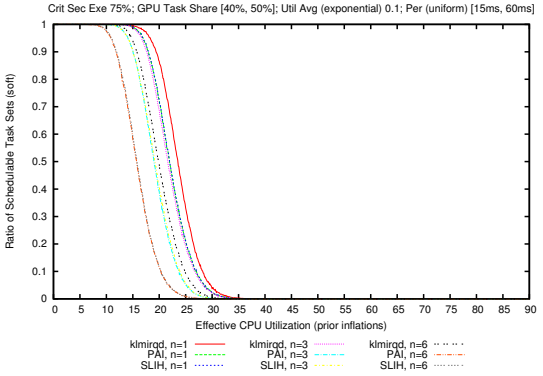

Figure 309. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

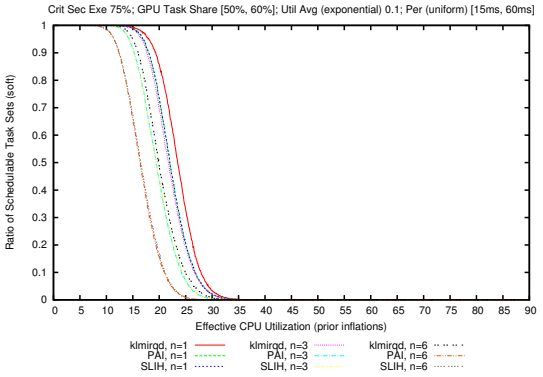

Figure 310. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

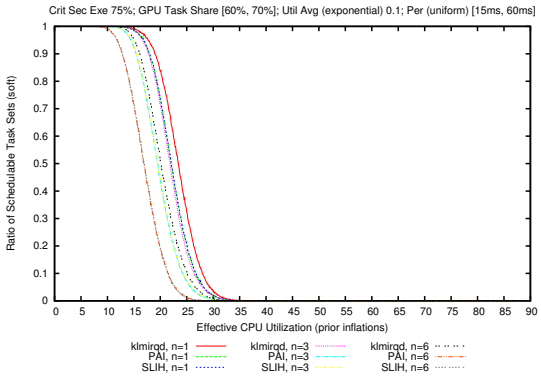

Figure 311. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

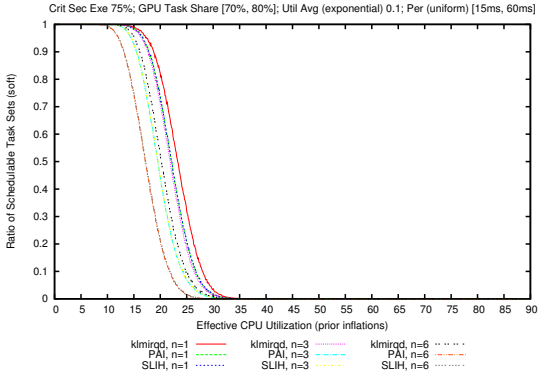

Figure 312. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

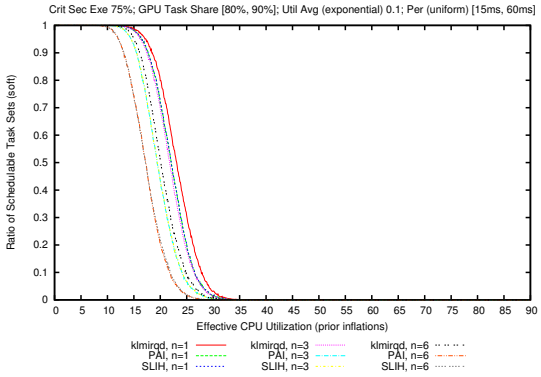

Figure 313. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

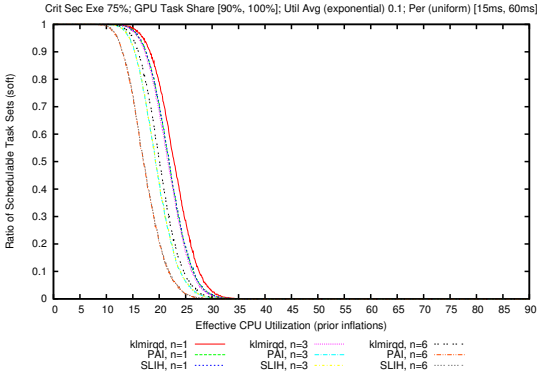

Figure 314. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

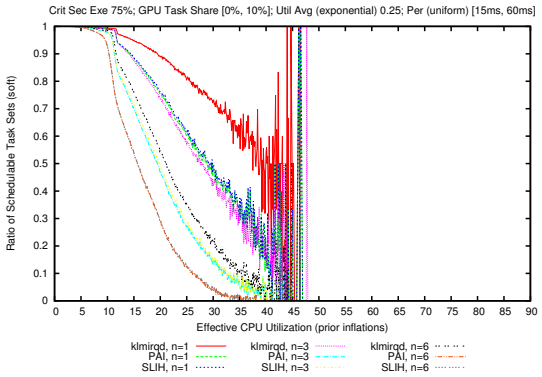

Figure 315. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

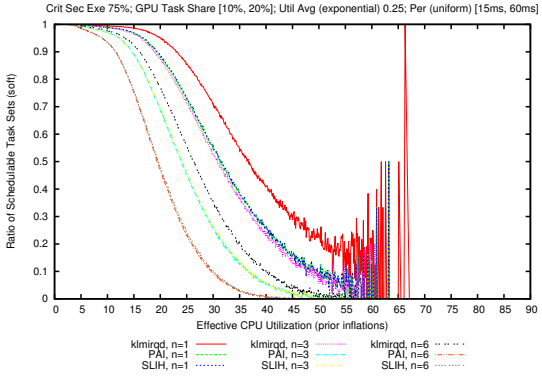

Figure 316. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

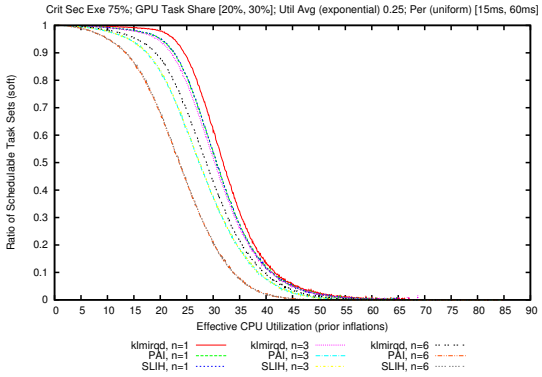

Figure 317. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

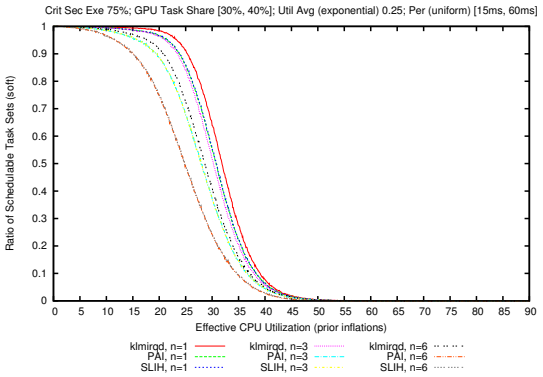

Figure 318. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

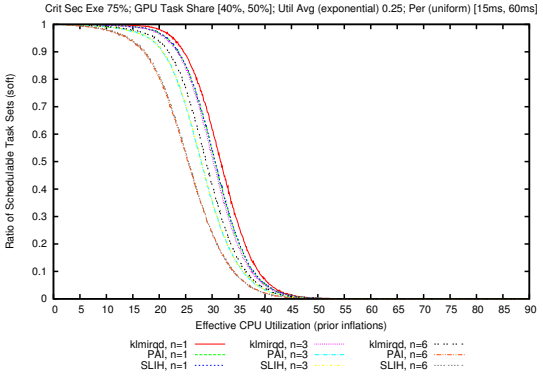

Figure 319. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

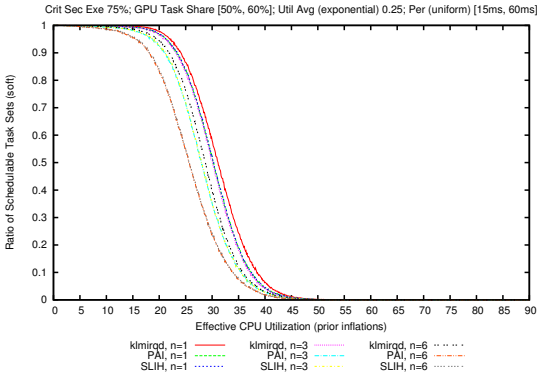

Figure 320. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

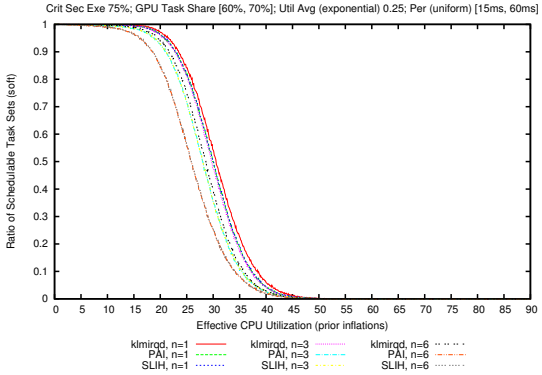

Figure 321. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

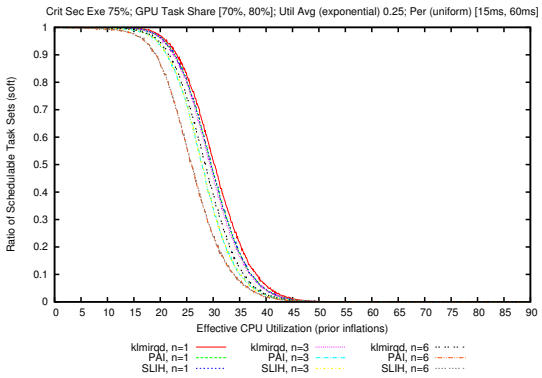

Figure 322. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

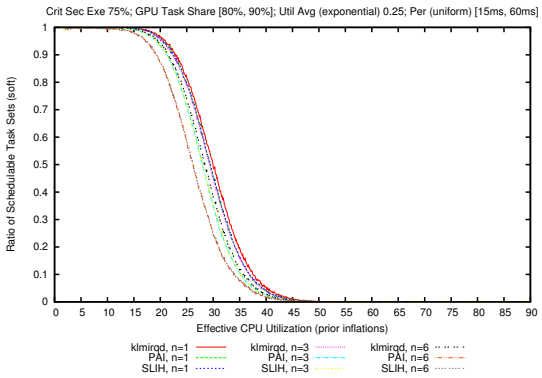

Figure 323. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

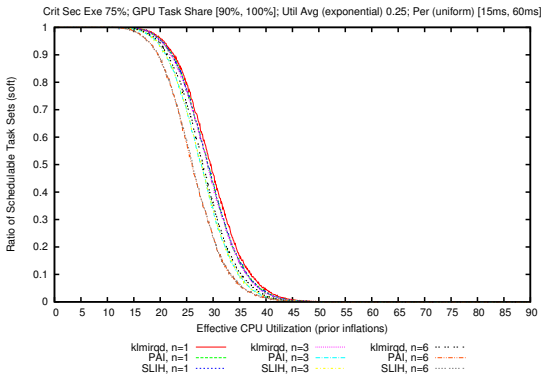

Figure 324. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

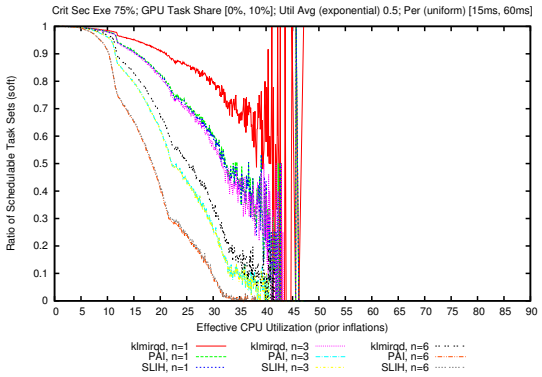

Figure 325. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

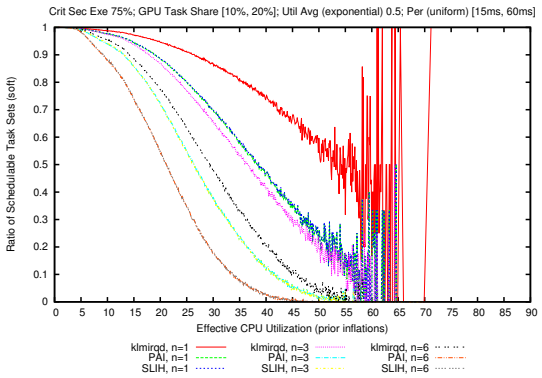

Figure 326. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

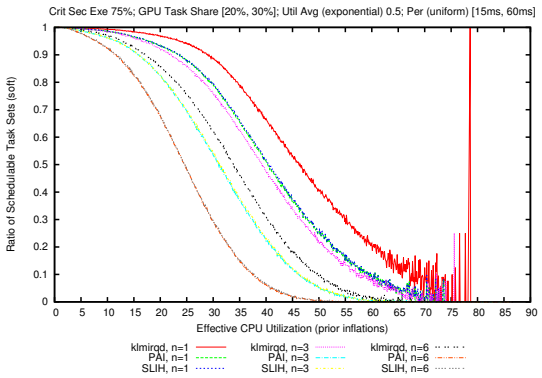

Figure 327. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

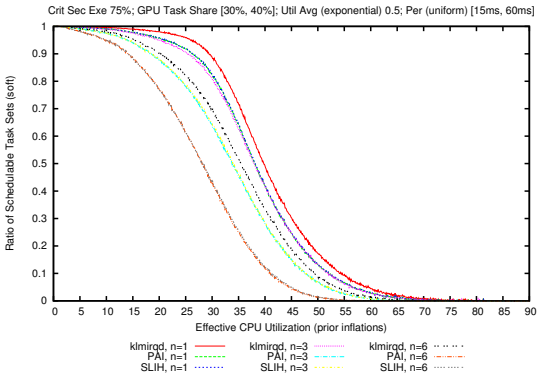

Figure 328. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

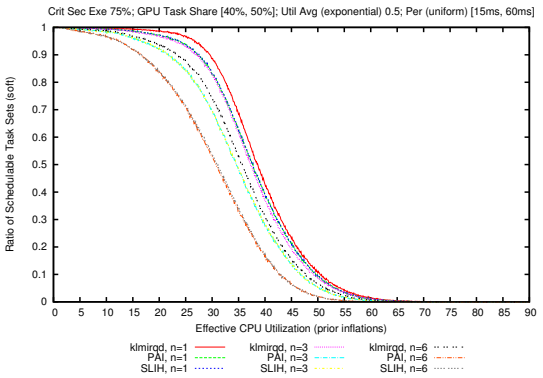

Figure 329. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

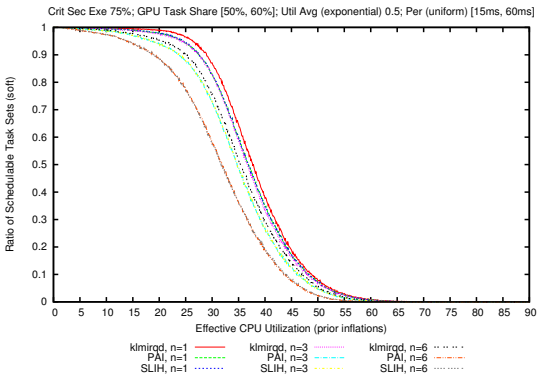

Figure 330. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

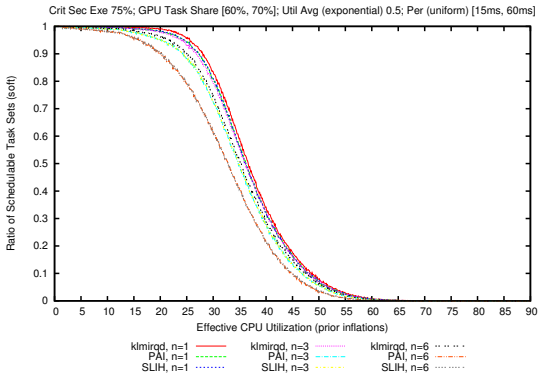

Figure 331. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

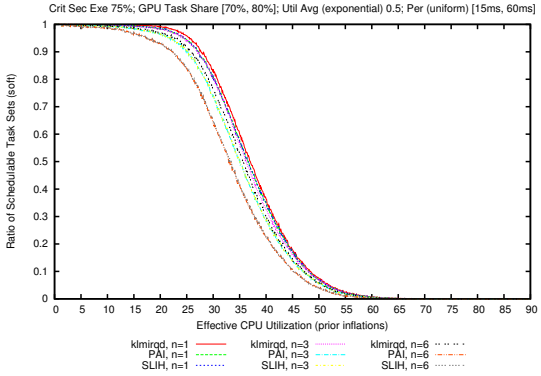

Figure 332. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

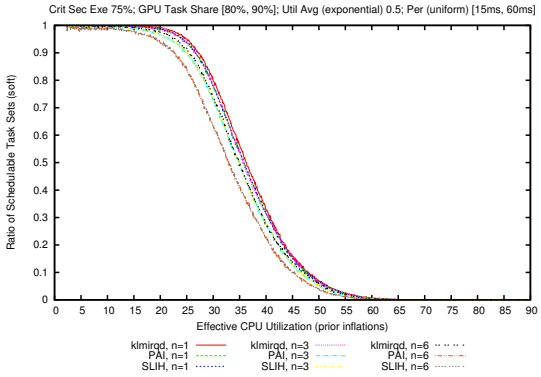

Figure 333. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

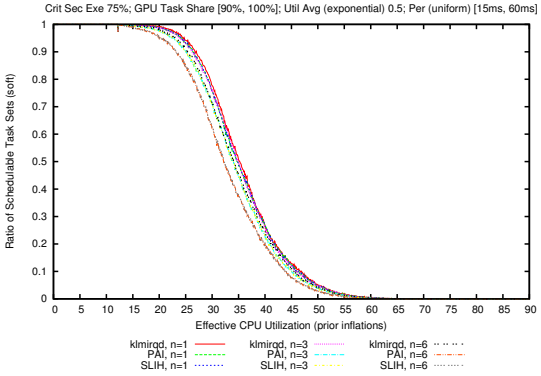

Figure 334. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.