Building a Real-Time Multi-GPU Platform: Robust Real-Time Interrupt Handling Despite Closed-Source Drivers

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 klitirqd. The design of klitirqd 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 still had to be mapped to languages developed exclusively for graphics. Graphics hardware manufactures recognized the market opportunities for better support of general purpose computations on GPUs (GPGPU) and released more 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 also 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.

Prior Work. GPUs have received serious consideration in the real-time community only recently. On the theoretical side, Raravi et al. have developed methods for estimating worst-case execution time on GPUs [9] and scheduling algorithms for "two-type" heterogeneous multiprocessor platforms, with CPU/GPU platforms particularly in mind [10], [11]. On the more applied side, Kato et al. have developed quality-of-service techniques for graphical displays on fixed-priority systems [12], [13].

In our own work, we have investigated many of the challenges faced when augmenting multicore platforms with GPUs that have non-real-time, throughput-oriented, closed-source device drivers [14]. These drivers exhibit behaviors that are problematic. For example, the driver only allows one task to execute work non-preemptively on a GPU at a time.[2] Also, when a GPU comes under contention, blocked tasks wait on a spinlock. The resulting wasted CPU time can be significant: depending on the application, GPU accesses can commonly take tens of milliseconds up to several seconds [14]. More problematically, blocked tasks have no mechanism to change the priority of a GPU-holding task. Thus, real-time tasks may experience unbounded priority inversions.

The primary solution we presented in [14] 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} [15], UNC's real-time extension to Linux, and demonstrated improved real-time characteristics such as reduced CPU utilization and reduced deadline tardiness.

Contributions. One issue not addressed in our prior work is the effect asynchronous GPU interrupts have on real-time execution. Interrupts cause complications

[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.

[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.

in real-time systems by introducing increased system latencies, decreased schedulability, and additional complexity in real-time operating 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 nature, servicing techniques, and effects that interrupts 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-threaded sporadic task model. To the best of our knowledge, prior interrupt-related work has not 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 processes 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 klitirqd.

The rest of this paper is organized as follows. In Sec. II, we review the problems posed by interrupts in real-time systems and discuss interrupt processing in Linux (the foundation for LITMUS^{RT}). Then in Sec. III, we review prior work on real-time interrupt handling and describe our solution, klitirqd. In Sec. IV, we show that klitirqd can be applied to handle GPU interrupts by intercepting and rerouting the interrupt processing of the closed-source GPU driver. In Secs. V–VII, we evaluate klitirqd by examining its effects on priority inversions, response times, and overhead-aware schedulability analysis, respectively. We conclude in Sec. VIII.

Due to space limitations, we henceforth limit attention to GPU technologies from the manufacture NVIDIA. NVIDIA's CUDA [16] platform is widely accepted as the leading solution for GPGPU.

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. Only after an interrupt handler has completed execution may an interrupted CPU resume the execution of the previously scheduled task.

Interrupts require careful implementation and analysis in real-time systems. Interrupts may come periodically, sporadically, or at entirely unpredictable moments, depending upon the application. In uniprocessor and partitioned multiprocessor systems, one may be able model an interrupt source and handler as the highest-priority real-time task or as a blocking source [17], [18], though the unpredictable nature of interrupts in some applications may require conservative analysis. Such approaches can also be extended to multiprocessor systems where real-time tasks may migrate between CPUs [19]. 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 only performs the minimum amount of processing necessary to ensure proper functioning of hardware; any additional work that may need to be carried out in response to an interrupt is deferred for later processing. 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.

Interrupt Handling In Linux. We now review how split interrupt handling is done in Linux. We focus on Linux for three reasons. First, Linux is well supported by GPU manufactures. Second, it is the basis for LITMUS^{RT}. Third, despite its general-purpose origins, variants of Linux are widely used in supporting real-time workloads.

During the initialization of the Linux kernel, kernel components and device drivers (even closed-source ones) register interrupt handlers with the kernel's interrupt services layer. These registrations are essentially name-value pairs of the form `<interrupt identifier, interrupt service routine>`.

Upon receipt of an interrupt on a CPU, Linux immediately invokes the registered *interrupt service routine* (ISR). In terms of split interrupt handling, the ISR is the *top-half* of the interrupt handler. If an interrupt requires additional processing beyond what can be implemented in a minimal top-half, the top-half may issue deferred work to the Linux kernel in the form of *softirqs*. Softirqs are small units of work executed by the Linux kernel, and in split interrupt handling parlance, each invocation of a softirq is an ISR *bottom-half*. The sequence of steps taken by Linux to service an interrupt is illustrated in Fig. 1. 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.

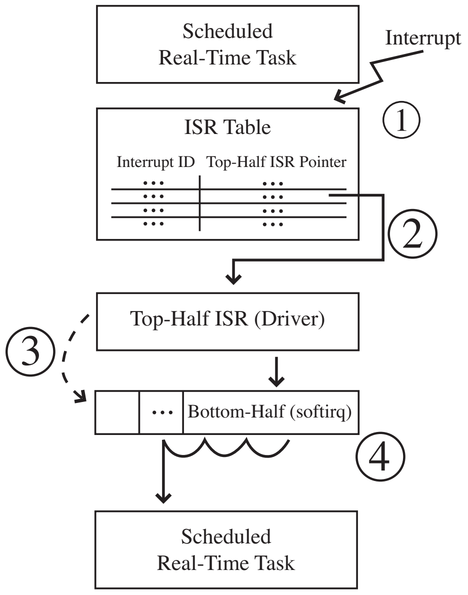


Figure 1. The interrupt handling in Linux. **(1)** An interrupt occurs and the currently scheduled task is suspended. **(2)** The ISR for the interrupt type is executed. **(3)** The ISR may issue deferred work as a tasklet. **(4)** Before resuming the interrupted task, up to ten softirqs are executed, possibly including tasklets issued in (3).

The Linux kernel executes softirqs using a heuristic. Immediately after executing a top-half, but before resuming execution of the interrupted task, the kernel executes up to ten bottom-halfs. Any pending softirqs remaining are dispatched to one of several (per-CPU) kernel threads dedicated to softirq processing; these are the "ksoftirq" daemons. The ksoftirq daemons are scheduled with high priority, but are schedulable and preemptible entities nonetheless. 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-half that yields a bottom-half, that bottom-half will subsequently be executed before interrupt processing completes, delaying the interrupted task. If a bottom-half 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 well-known PREEMPT_RT Linux kernel patch addresses this issue by processing all bottom-halfs (except the most critical, such as timers) with a pool of schedulable threads. Ideally, interrupt processing threads should be scheduled with the priority of the blocked client task using the I/O device. However, interrupt threads in PREEMPT_RT have only a single fixed priority, even if the associated device is shared by multiple client tasks of differing priorities. This can 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 rendezvouses with I/O results at a later point in time. Asynchronous I/O helps improve overall performance and is commonly used in GPU applications to mask bus latencies. Since synchronization with the I/O device is deferred in asynchronous I/O, it is possible for interrupts to be received, and corresponding bottom-halfs executed, while a client task is scheduled. In such a case, the client task essentially becomes temporarily multithreaded, which *breaks the assumption of single-threaded execution common in real-time task models such as the sporadic model.* A co-scheduled interrupt can be interpreted as causing a priority inversion. These issues caused by asynchronous I/O in multiprocessors are not merely limited to Linux variants. To the best of our knowledge, this problem has not been directly addressed in the real-time literature.

Neither standard Linux nor its PREEMPT_RT variant implement split interrupt handling in a way amenable to real-time schedulability analysis. In the next section, we propose such an implementation.

III. Interrupt Handling in LITMUS^{RT}

LITMUS^{RT}, a real-time extension to 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 very 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, one we begin 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 klitirqd.[3] This name is an abbreviation for "Litmus softirq daemon" and is prefixed with a 'k' to indicate that the daemon executes in kernel space. klitirqd 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.

klitirqd is designed to be extensible. Unlike the ksoftirq daemons, the system designer may create an arbitrary number of klitirqd threads to process tasklets from a single device, or a single klitirqd thread may be shared among many devices. The detailed implementation of klitirqd is as follows. Instead of using the standard Linux `tasklet_schedule()` function call to issue a tasklet to the kernel, an alternative function

[3]Source code available at http://www.cs.unc.edu/~anderson/litmus-rt

`litmus_tasklet_schedule()` is provided to issue a tasklet directly to a klitirqd thread. The caller (such as a device driver) must supply both an *owner* for the given tasklet as well as a klitirqd identifier that specifies which klitirqd daemon is to perform the processing. The owner of the tasklet may be a pointer to a real-time user process, such as one blocked for a particular I/O event, or even a bandwidth server used to limit the processing rate of a particular type of tasklet. An idle klitirqd thread suspends, waiting for a tasklet to process. Once a tasklet arrives, the klitirqd thread adopts the scheduling priority, including any inherited priority, of the tasklet owner.

The LITMUS^{RT} scheduler ensures that a klitirqd thread and its tasklet owner are never co-scheduled. This allows asynchronous I/O to be supported without violating the single-threaded task models commonly assumed.

We recognize that similar architectures for split interrupt handling have been proposed and implemented before. For instance, LynxOS [20] has supported priority-inheritance-based split interrupt handling for many years. In LynxOS, the interrupt processing daemon inherits the greatest priority of any task actively using the device that raised the interrupt. Steinberg et al. have also developed and implemented similar techniques based upon bandwidth inheritance to support interrupt processing in a modified L4 microkernel [21] and the NOVA microhypervisor [22]. While these approaches are similar to our own, there are several key differences. First, we support JLSP schedulers, while prior work has focused only on fixed-priority systems. Second, we support non-partitioned multiprocessor systems while maintaining single-threaded execution. LynxOS supports non-partitioned scheduling, but breaks the single-threaded model. Steinberg et al.'s methods are limited to uniprocessor and partitioned systems and require any tasks that share a resource to be within the same partition. Finally, the implementation of our solution in LITMUS^{RT} allows the use of unmodified Linux device drivers. At this time, native GPU drivers for LynxOS and L4 are unavailable.

More closely related to our approach is an implementation of real-time-scheduled interrupt handlers in Linux by Manica et al. [23]. Their approach grouped softirqs within bandwidth servers, similar to the techniques used by Steinberg et al., with the aim of constraining resource consumption by I/O-using tasks. However, each of their interrupt threads is pinned to an individual CPU, which limits applicability to partitioned scheduling.

IV. GPU Integration

In Sec. III, we described how interrupt handlers are to call the function `litmus_tasklet_schedule()` to dispatch klitirqd bottom-half tasklets. The caller must provide two parameters: (1) the tasklet *owner* (the real-time task that requires the bottom-half to execute to make progress) and (2) a *klitirqd identifier* for the daemon that is to execute the tasklet. While any LITMUS^{RT}-aware device driver could be easily modified to provide these parameters, how shall we accomplish this with a closed-source GPU driver that cannot even be modified to call `litmus_tasklet_schedule()`? We addressed this issue by focusing separately on tasklet interception, device identification, owner identification, and dispatch.

Tasklet Interception. Though the GPU driver is closed-source, it must still interface with an open source operating system kernel. The driver makes use of a variety of kernel services, including interrupt handler registration and tasklet scheduling. Though we cannot modify the GPU driver, we may still intercept the calls the driver makes to these OS services. In particular, we modify the standard internal Linux API function `tasklet_schedule()`.

When `tasklet_schedule()` is called by a kernel component, a callback function pointer must be provided that specifies the entry point for the execution of the deferred work. If we can identify callbacks to the closed-source driver, then we can identify and intercept all tasklets the driver schedules. Luckily, this is possible because the driver is loaded into Linux as a module (or kernel plugin). We leverage this fact to use various module-related features of Linux to inspect every callback function pointer of every tasklet scheduled in the system online.[4] Thus, we make modifications to `tasklet_schedule()` to catch 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. If a system has multiple GPUs, merely intercepting deferred GPU work is not enough; we must also determine which GPU in the system raised the initial interrupt. While we could have possibly performed this identification process at the lowest levels of interrupt handling, we opted for a simpler solution closer to the tasklet scheduling process. The GPU driver attaches to every tasklet a reference to a block of memory that provides input parameters to the tasklet callback. This block of data includes a device identifier (ranging from 0 to $g - 1$, where g is the number of system GPUs), which indicates which GPU raised the interrupt. However, accessing this data within the memory 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.

Because the internal APIs of Linux change frequently and many Linux users use custom kernel configurations, the NVIDIA driver is not distributed as a monolithic

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

precompiled binary. Instead, the driver is distributed in a partially compiled form, allowing it to support a changing kernel in varied configurations. The portions of the driver that NVIDIA wishes to keep closed are distributed in obfuscated precompiled object files. However, the distribution also includes plain source code for an OS/driver interface layer that bridges the internal Linux kernel interfaces with the precompiled object files. Through the visual inspection of this bridge code, we gained insight into the format of the tasklet memory block, and through a process of trial and error, determined the fixed address offset of the device identifier.

To this point, we have explained how to intercept and identify the source of tasklets the driver hands off to the kernel for later processing. What remains is to schedule the deferred work with the proper priority by identifying the user task that is using the associated GPU and then to dispatch the work to the appropriate klitirqd daemon.

Owner Identification. As mentioned in Sec. I, a closed-source GPU driver can exhibit behaviors that are detrimental to the predictability requirements of a real-time system. In [14], we presented methods for removing the GPU driver from resource arbitration decisions, thereby removing much of the associated uncertainty. The primary method we presented introduced a real-time semaphore to arbitrate access to GPUs. In particular, to manage a pool of k GPUs, a real-time k-exclusion protocol is used that can assign any available GPU to a GPU-requesting task. We can use such a protocol not only to arbitrate GPU access, but to also act as registry of tasks actively using GPUs. Whenever a GPU is allocated to a task by the protocol, an internal lookup table, called the *GPU ownership registry*, indexed by device identifier, is updated to record device ownership.

The arbitration protocol considered herein is a k-exclusion extension of the *flexible multiprocessor locking protocol* (FMLP) [24], which we call the k-FMLP.[5] Using the k-FMLP, GPU-using jobs merely issue requests for an available GPU, not a specific GPU. 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$^{\text{RT}}$ to support this work. In doing so, special consideration had to be paid to integrate with klitirqd. For example, the k-FMLP uses priority inheritance; a priority inherited by a GPU holder must be propagated transitively to any associated klitirqd tasklet.

With the device identifier extracted from the tasklet memory block and device registry table, determining the current GPU owner is straightforward. We now have gathered all required information to dispatch a GPU klitirqd tasklet; now we must determine which klitirqd instance will perform the processing.

[5]A full description of the k-FMLP is available in the online version of this paper at http://www.cs.unc.edu/~anderson/papers.html. A detailed discussion of some issues that arise when constructing a real-time k-exclusion protocol can be found in [25].

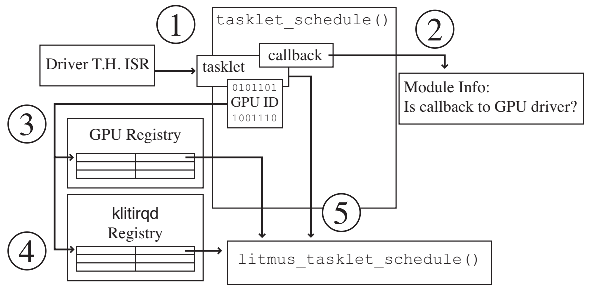


Figure 2. GPU tasklet redirection. **(1)** A tasklet from the GPU driver is passed to `tasklet_schedule()`. **(2)** The tasklet is intercepted if the callback points to the driver. **(3)** The GPU identifier is extracted from the memory block attached to the tasklet using a known address offset and the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

klitirqd Dispatch. The architecture of klitirqd is general enough to support any number of daemon instances, all scheduled by a JLSP real-time scheduler. In a system with g GPUs, there should be g klitirqd instances to ensure that all GPUs can be used simultaneously. Each klitirqd instance is assigned a specific GPU. This assignment is recorded in the *klitirqd assignment registry*.

The overall klitirqd architecture is summarized in Fig. 2. Using the device identifier extracted from the intercepted tasklet of the GPU driver, our modified `tasklet_schedule()` references the GPU ownership and klitirqd assignment registries and redirects all GPU tasklets to the proper klitirqd instance, with the proper priority, by calling `litmus_tasklet_schedule()`.

V. Evaluation of Priority Inversions

In this and the next two sections, we present an evaluation of klitirqd. Our focus in this section is determining the impact of priority inversions caused by interrupts.

Evaluation Platform. The platform used in all of our experiments is a dual-socket six-cores-per-socket Intel Xeon X5060 CPU platform, with a total of twelve cores running at 2.67GHz. This platform also includes eight Nvidia GTX-470 GPUs.

In all of our experiments, we use a clustered scheduler, with GPUs statically assigned to clusters, and a separate instance of the k-FMLP used within each cluster to manage the assigned GPUs. Clustered schedulers have been shown to be effective if bounded deadline tardiness is the real-time requirement of interest [26]. In this section, we consider only the clustered earliest-deadline-first (C-EDF) algorithm, though later we also consider the clustered rate-monotonic (C-RM) algorithm. In either case, clustering is split along the NUMA architecture of the system, yielding two clusters of six CPU cores and four GPUs apiece. 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 use CUDA 4.0 for our GPU runtime environment.

Experimental Setup. We assessed the impacts of priority inversions by generating sporadic task sets and then executing them in LITMUS^{RT}. Each generated task set included both CPU-only and 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. 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 [14], which are used frequently in image processing. Finally, the 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. In order to gauge the performance of our implementation with respect to system utilization, task sets were generated with system utilizations ranging from 7.5 to 11.5, in increments of 0.1, for a total of 41 task sets.

Each generated task set was executed in LITMUS^{RT} on the evaluation platform for two minutes. Tasks executed simple numerical code (on both CPUs and GPUs) for the configured execution durations. GPU requests were processed using asynchronous I/O. Every task set was executed twice, once in LITMUS^{RT} configured to use klitirqd and once in LITMUS^{RT} using standard Linux interrupt handling. Scheduling logs were recorded, from which we compared the performance of klitirqd and standard Linux interrupt handling in LITMUS^{RT} according to several metrics, as discussed next.

Metrics. Ideally, the system should conform to the sporadic task model and not suffer any priority inversions.

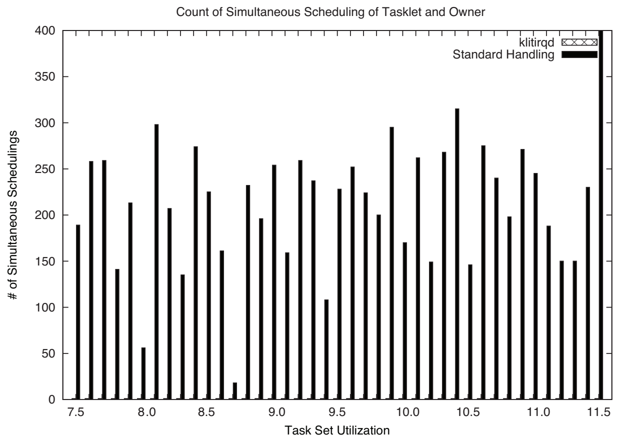


Figure 3. Histogram of concurrent execution events of a tasklet and its owner. No concurrent execution events were observed under klitirqd. There is no apparent trend with regard to task set utilization: the number of events observed differs greatly among the task sets.

We assessed deviance from this ideal by: **(i)** counting the number of *concurrent execution events*, where tasklets and owners are simultaneously scheduled in violation of the sporadic task model; **(ii)** determining the *distribution of priority inversion durations*; **(iii)** counting the *number of priority inversions*; and **(iv)** computing the *cumulative priority inversion length*.

Results. Fig. 3 shows our measurements for the number of concurrent execution events. No such events occurred when using klitirqd. However, as expected, they do occur under standard Linux. Most task sets experienced 100–200 concurrent execution events. While these numbers appear to be low, we found that it was most often the case that a GPU-using task had already blocked by the time its tasklet is scheduled for our simple test programs; we expect that this would not be the case with a more complex workload. The absence of concurrent execution events under klitirqd shows that it is effective at enforcing conformance to the sporadic task model.

While priority inversions cannot be totally eliminated, they should nevertheless be as short as possible. Fig. 4 shows a representative example of the cumulative distribution function of priority inversion length under both interrupt handling methods.[6] As seen, a typical priority inversion is much shorter under klitirqd than under Linux interrupt handling. For example, 90% of inversions under klitirqd are shorter than $9\mu s$, whereas the 90th percentile exceeds $30\mu s$ under Linux interrupt handling.

While priority inversions should be as short as possible, the number of inversions is also important because a system that suffers many short inversions may be disrupted by their cumulative effect. Fig. 5 depicts the

[6]Graphs for all tested task sets are available in the online version of this paper.

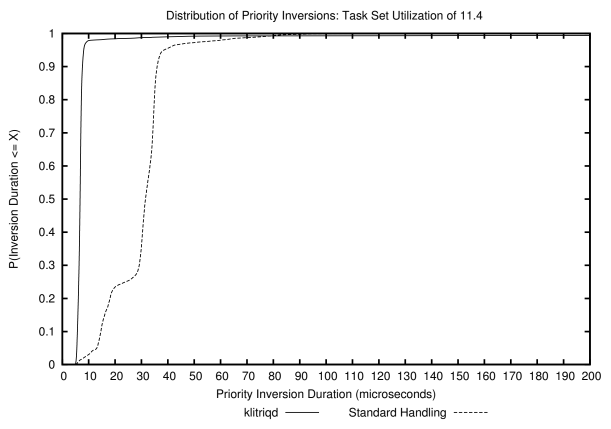


Figure 4. Cumulative distribution of priority inversion durations.

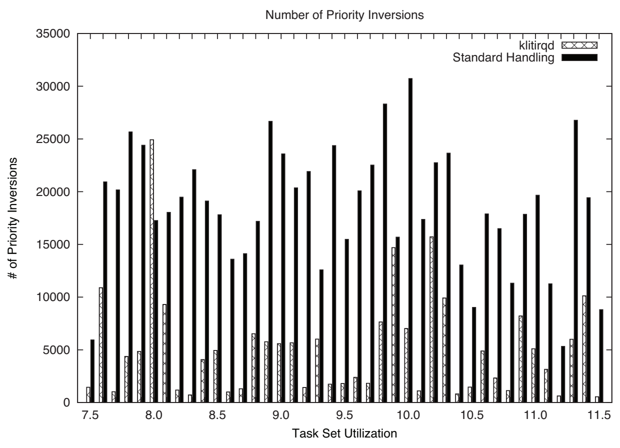


Figure 5. Histogram of detected inversions. There is no observable trend in task set utilization with this sample size, though variance among task sets is significant.

number of inversions caused by GPU tasklet processing under both interrupt handling methods. For all but one of the task sets, the use of klitirqd resulted in a significant reduction of priority inversions. The sole exception was the task set with a system utilization of 8.0. However, a closer examination reveals that the cumulative priority inversion length is in fact shorter under klitirqd, even for this exceptional case. Fig. 6 shows cumulative priority inversion length as a function of maximum priority inversion length for the task set with utilization 8.0. Observe in Fig. 6 that priority inversions of length up to $50\mu s$ have a cumulative duration of only 180,000μs under klitirqd, but more than 475,000μs under Linux interrupt handling. Even though the number of priority inversions is slightly greater under klitirqd in this particular case, the overall effect is much less because most inversions are indeed short. In all other cases where there were fewer priority inversions under klitirqd than under standard Linux interrupt handling, the cumulative priority inversion length was similarly (much) less.

In summary, our data shows that klitirqd outperforms standard Linux interrupt handling in each of the four evaluation metrics. Further, they demonstrate that *even closed-source drivers* can still be prevented from causing undue interference.

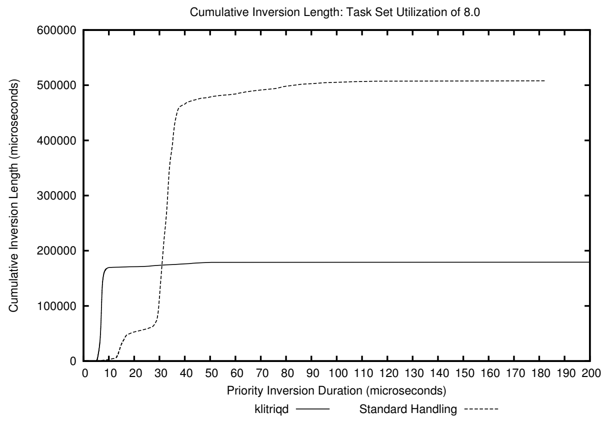


Figure 6. Cumulative priority inversion length as a function of maximum priority inversion length for the task set with utilization 8.0. The total duration of priority inversion is more than twice as large under standard Linux interrupt handling than klitirqd, despite an increased number of priority inversions under klitirqd in this case.

VI. System-Wide Evaluation of Interrupt Handling Methods

In this section, we examine system-wide effects of interrupt handling in terms of job response time under LITMUS^{RT}, both with and without klitirqd, the PREEMPT_RT real-time Linux patch, and standard Linux. Recall from Sec. II that Linux often executes interrupt bottom-halfs immediately after executing top-halfs, essentially nullifying the real-time benefits of split interrupt handling. While interrupt bottom-halfs are threaded under PREEMPT_RT, the interrupt threads that execute bottom-halfs must be assigned a *single* fixed priority, which may lead to undue priority inversions.

Experimental Setup. To demonstrate these real-time weaknesses in Linux and PREEMPT_RT, and show how klitirqd can successfully address them, 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} (both with and without klitirqd) against Linux and PREEMPT_RT, the workload was scheduled using the C-RM algorithm since Linux and PREEMPT_RT only support fixed real-time priorities. Counting semaphores were used to protect each pool of GPU resources in Linux and PREEMPT_RT in a manner similar to how the k-FMLP is used under LITMUS^{RT}. The workload consisted of 50 tasks: two GPU-using

Avg. Response Time as Percent of Period	C-RM					C-EDF
	PREEMPT_RT		Linux	LITMUS^{RT}		
	Low Prio. Interrupts (a)	GPU-Matching Prio. Interrupts (b)	(c)	w/o klitirqd (Linux) (d)	klitirqd (e)	klitirqd (f)
CPU-Only Tasks	429.03%	440.62%	100.98%	32.85%	32.83%	25.14%
GPU-Using Tasks	161.80%	71.63%	90.04%	26.70%	28.29%	18.13%

Table I
AVERAGE RESPONSE TIME OF CPU-ONLY AND GPU-USING TASKS EXPRESSED AS A PERCENTAGE OF PERIOD.

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 scheduling clusters. Unique priorities were assigned to each task within each cluster according to task period. Thus, the 20 CPU-only tasks in each cluster had priorities strictly between one GPU-using task with greater priority and four with lesser priority.

The workload was executed on six different platform configurations: (1) Standard Linux, 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 equal to that of the greatest GPU-using task;[7] (4) LITMUS^{RT} without klitirqd; (5) LITMUS^{RT} with klitirqd; and finally, (6) LITMUS^{RT} with klitirqd, scheduled under C-EDF (for the sake of comparison). This workload was executed 25 times for each system configuration for a duration of 60 seconds. Response times were recorded for all jobs consistently on each platform.

Results. Table I gives average response times, as percent of period, for CPU-only and GPU-using tasks under the various platform scenarios.

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 the average GPU-using job response time is over twice that in column (b).

Observation 2. *Standard Linux outperforms PREEMPT_RT (in this pathological case) due to lower interrupt handling overhead.* Under standard Linux, bottom-halfs are usually executed immediately after top-halfs; thus, bottom-halfs essentially execute with maximum priority (like column (b)), yet this is accomplished without the overhead of threaded interrupt handling. Increased CPU availability improves the response times of CPU-only jobs in column (c) in comparison to both columns (a) and (b), while GPU-using jobs still typically complete before their deadlines.

Observation 3. *klitirqd assigns priorities to interrupt threads at runtime, resulting in schedulable and analyzable real-time systems.* The average response time values in columns (d) and (e) indicate that jobs are typically completing well before their deadlines. While GPU-using tasks fare slightly worse in column (e), the heuristic-driven interrupt handling methods of standard Linux are not amenable to schedulability analysis.

Observation 4. *Overheads introduced by klitirqd into LITMUS^{RT} are largely negligible.* The nearly-equal response time values in columns (d) and (e) indicate that klitirqd overhead costs are not too great or are offset by a reduction in priority inversions.

Observation 5. *LITMUS^{RT} with klitirqd out-performs PREEMPT_RT.* A comparison of columns (b) and (e) suggests that, in this experiment, deadline misses were not significant under klitirqd but were common under PREEMPT_RT. Unfortunately, it is difficult to identify a single difference between PREEMPT_RT and LITMUS^{RT} causing 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 [27], but we mention it nonetheless. This is another indication that PREEMPT_RT may not be a desirable solution in all applications, especially in

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

soft real-time systems, since C-EDF is not supported.[8]

VII. Overhead-Aware Schedulability

As seen in Secs. V and VI, klitirqd reduces the frequency and duration of priority inversions at the expense of some additional scheduling overhead, while non-threaded interrupt handling incurs low system overheads due to the lack of scheduling, at the expense of potentially longer (and more numerous) priority inversions that may impact *every* task. How does this affect general task set schedulability? The answer to this question depends upon the actual system overheads and priority inversion durations associated with each approach. To address this question in the context of C-EDF, we conducted schedulability experiments in which actual measured overheads were considered using a methodology similar to that proposed in [28].

Using the same hardware platform described in Sec. V, we measured the following system overheads under C-EDF scheduling: 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 klitirqd, tasklet release queuing. We then randomly generated task sets with properties similar to those in Sec. V. Finally, we checked the schedulability of each generated task set by using a soft real-time (bounded tardiness) schedulability test for C-EDF scheduling augmented to account for overheads [29]. In doing the latter, average overhead values were used (as in [28] when analyzing soft real-time systems). Interrupts were accounted for using task-centric methods [19].[9]

A selection of our schedulability results is given in Fig. 7,[10] 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 $n \in \{1, 3, 6\}$, which specifies the number of GPU interrupts each GPU-using job may generate. In Fig. 7, schedulability is higher under klitirqd (threaded) interrupt handling than under standard Linux (non-threaded) interrupt handling for each choice of n, with a greater disparity between the two for larger values of n.

The primary motivation for utilizing GPUs in a real-time system is increased performance. The benefits of

[8]Prior work has blended the PREEMPT_RT with another patch in development, SCHED_DEADLINE, to achieve EDF-based interrupt handlers [23] in a partitioned EDF-scheduled system. However, this solution still suffers from utilization loss due to bin-packing-like problems even when used in a soft real-time system.

[9]Please see the appendix of the online version of the paper for a detailed description of interrupt and overhead accounting methods.

[10]Additional graphs are available in the online version of the paper.

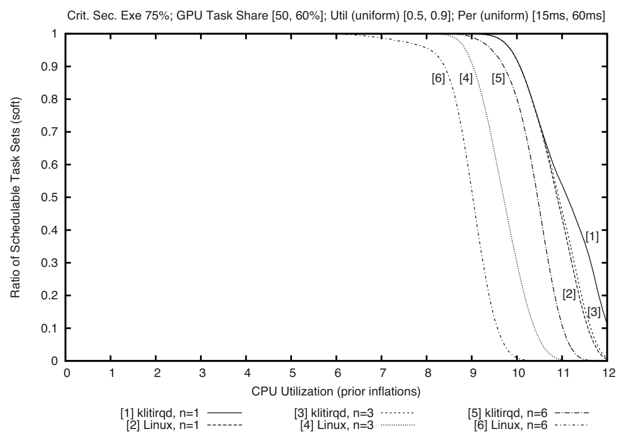


Figure 7. The percentage of schedulable task systems (y-axis) as a function of CPU utilization (x-axis) under klitirqd (threaded) interrupt handling and standard Linux (non-threaded) interrupt handling.

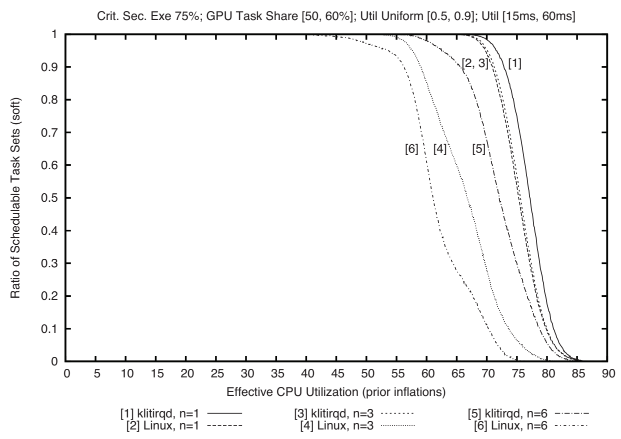


Figure 8. Schedulability results similar to Fig. 7 except that a GPU speedup of 16× is assumed.

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. 8 depicts the same schedulability results shown in Fig. 7, 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. 8 shows, the impact of using klitirqd is even greater if effective utilizations are considered. As seen, when $n = 6$, 90% of task sets with an effective utilization of 65.0 CPUs are schedulable under klitirqd. In contrast, effective utilization must be decreased to 55.0 CPUs to achieve the same degree of schedulability under Linux interrupt handling. klitirqd supports effective utilizations 10 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 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, and offers better results in terms of overall schedulability (with overheads considered).

This paper lays the groundwork for future investigations into real-time platforms using GPUs. In this paper, we have limited attention to clustered scheduling. In a future study, we intend to consider the 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 future 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 schedulability with overheads considered.

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] G. Raravi and B. Andersson, "Calculating an upper bound on the finishing time of a group of threads executing on a GPU: A preliminary case study," in *WiP session of 16th ECRTS*, 2010.

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

[11] 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, to appear.

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

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

[14] G. Elliott and J. Anderson, "Globally scheduled real-time systems with GPUs," in *18th RTNS*, 2010.

[15] 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.

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

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

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

[19] 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.

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

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

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

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

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

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

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

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

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

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

[30] 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) [24], 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 [30], [25]. 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.[11]

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 $\left(\sum_{x=1}^{k} wait_x(t)\right)/k$.*

[11]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 $\sum_{T_j \in T^R \backslash \{T_i\}} l_j$. Otherwise, it may be approximated by $\sum_{x=1}^{k} \max\{l\} \cdot |\text{FIFO}_x| = \max\{l\} \cdot \left|T_j \in T^R \backslash \{T_i\}\right|$, in the l-oblivious case.

On average, each queue has a worst-case wait time of $\left(\sum_{T_j \in T^R \backslash \{T_i\}} l_j\right)/k$ in the l-aware case, or $\left(\max\{l\} \cdot \left|T_j \in T^R \backslash \{T_i\}\right|\right)/k$ 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*

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

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*

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

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:

$$\begin{array}{lll} \textbf{Maximize} & b_i & \\ \textbf{subject to} & & \\ & \sum_{T_j \in T^R \backslash \{T_i\}} x_{j,q} \cdot l_q \geq b_i & \forall j \in \{1, \ldots, k\} \\ & \sum_{j=1}^{k} x_{j,q} = 1 & \forall T_q \in T^R \backslash \{T_i\} \\ & x_{j,q} \in \{0, 1\} & \end{array} \quad (3)$$

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 klitirqd is able to meet soft real-time bounded tardiness constraints for more task sets than standard non-threaded Linux interrupt handling, despite additional overheads associated with klitirqd'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 [28] 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 [28] 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 standard Linux interrupt handling. Thereafter, we alter this model to account for overheads of GPU interrupts under klitirqd. 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 [27]. Under this test, two conditions must hold:

$$e_i \leq p_i \quad (4)$$

$$\sum_{T_i \in T} e_i/p_i \leq m \quad (5)$$

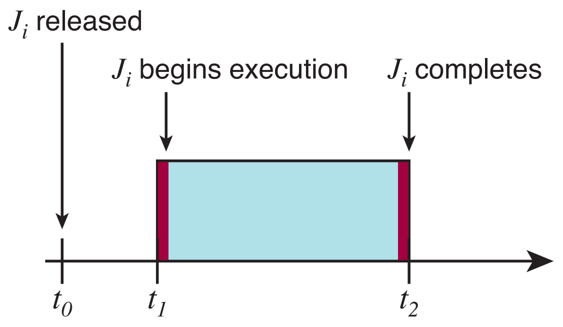


Figure 9. Basic accounting of overheads for simple CPU-only job, J_i. Job execution is framed by overheads to due scheduling decisions and context switches.

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 [29]. 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. 9 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:[12]

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

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-halfs or bottom-halfs*. Fig. 10 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

[12]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.

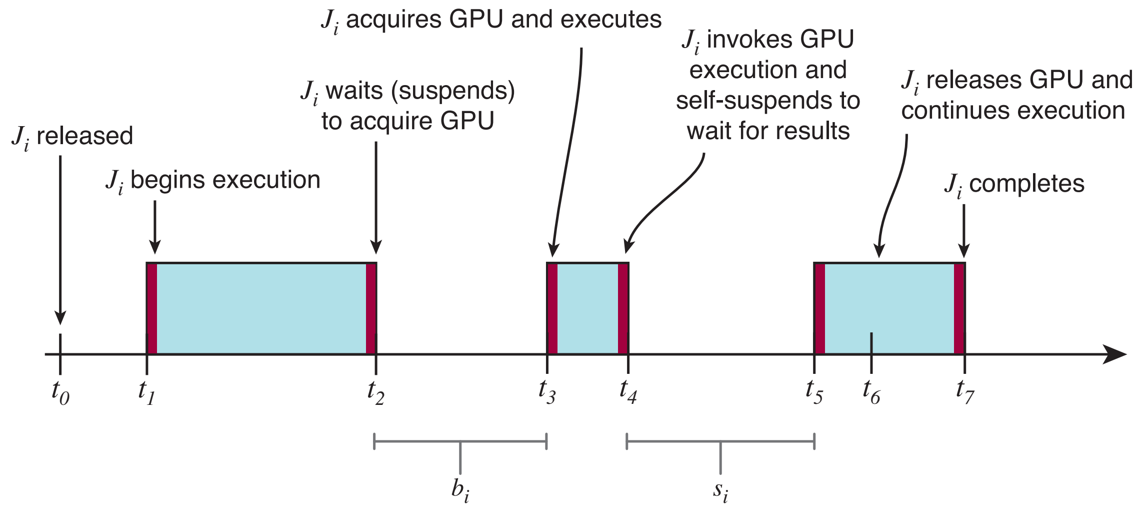

J_i acquires GPU and executes
J_i invokes GPU execution and self-suspends to wait for results
J_i waits (suspends) to acquire GPU
J_i releases GPU and continues execution
J_i released
J_i begins execution
J_i completes
t_0
t_1
t_2
t_3
t_4
t_5
t_6
t_7
b_i
s_i

Figure 10. 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.

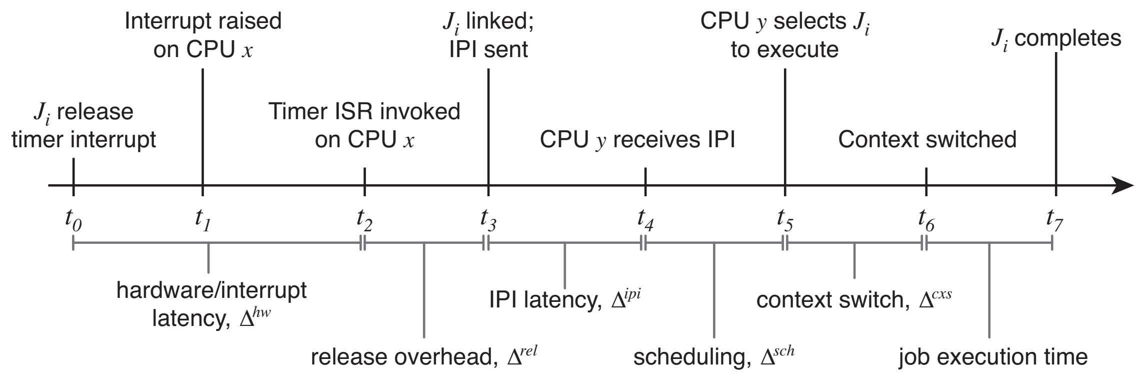

Interrupt raised on CPU x
J_i linked; IPI sent
CPU y selects J_i to execute
J_i completes
J_i release timer interrupt
Timer ISR invoked on CPU x
CPU y receives IPI
Context switched
t_0
t_1
t_2
t_3
t_4
t_5
t_6
t_7
hardware/interrupt latency, Δ^{hw}
IPI latency, Δ^{ipi}
context switch, Δ^{cxs}
release overhead, Δ^{rel}
scheduling, Δ^{sch}
job execution time

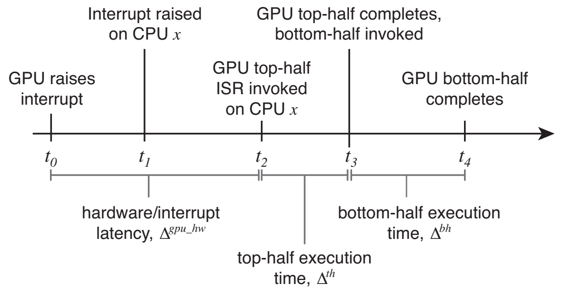
Figure 11. A newly released job experiences delays due to interrupt delivery latency and updates of ready queue data structures, in addition to scheduling and context switch overheads.

are delays which should be accounted for.

Fig. 11 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 klitirqd 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



Interrupt raised on CPU x
GPU top-half completes, bottom-half invoked
GPU raises interrupt
GPU top-half ISR invoked on CPU x
GPU bottom-half completes
t_0
t_1
t_2
t_3
t_4
hardware/interrupt latency, Δ^{gpu_hw}
bottom-half execution time, Δ^{bh}
top-half execution time, Δ^{th}

Figure 12. Interrupt handling overheads under standard Linux, with the liberal assumption that bottom-halfs are executed immediately and not preceded by other bottom-halfs 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 standard Linux interrupt handling and klitirqd.

Standard Linux Interrupts. Fig. 12 illustrates the sequence of events from a GPU interrupt, raised by the GPU device, to the completion of the interrupt bottom-half under standard Linux. 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-halfs 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 [28] 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 [29].

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:

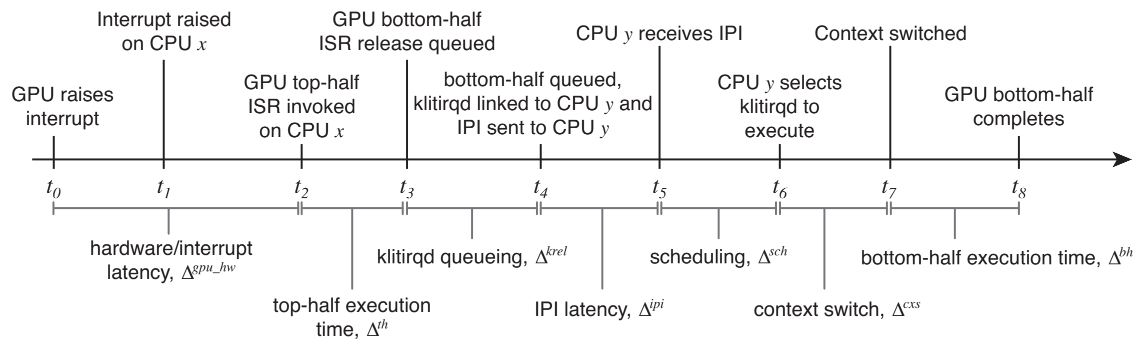
$$e_i^{(\text{cpu-only})_{12}} = e_i^{(\text{cpu-only})_9} + H_i(\Delta^{th} + \Delta^{bh}) \tag{12}$$

and

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

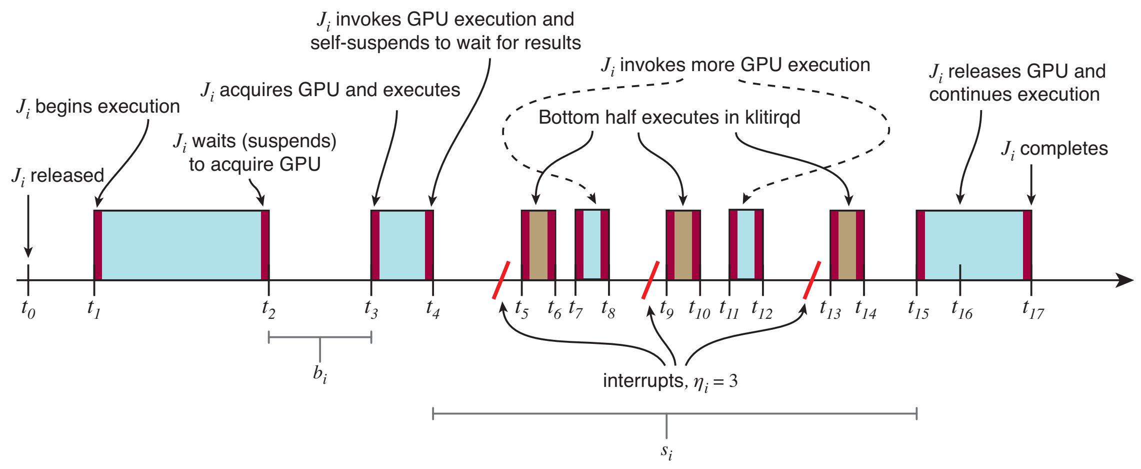
for CPU-only and GPU-using tasks, respectively.

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



Interrupt raised on CPU x
GPU bottom-half ISR release queued
CPU y receives IPI
Context switched
GPU raises interrupt
GPU top-half ISR invoked on CPU x
bottom-half queued, klitirqd linked to CPU y and IPI sent to CPU y
CPU y selects klitirqd to execute
GPU bottom-half completes
t_0
t_1
t_2
t_3
t_4
t_5
t_6
t_7
t_8
hardware/interrupt latency, Δ^{gpu_hw}
top-half execution time, Δ^{th}
klitirqd queueing, Δ^{krel}
IPI latency, Δ^{ipi}
scheduling, Δ^{sch}
context switch, Δ^{cxs}
bottom-half execution time, Δ^{bh}

Figure 13. Interrupt handling overheads under klitirqd.



J_i invokes GPU execution and self-suspends to wait for results
J_i invokes more GPU execution
J_i releases GPU and continues execution
J_i begins execution
J_i acquires GPU and executes
Bottom half executes in klitirqd
J_i waits (suspends) to acquire GPU
J_i completes
J_i released
t_0
t_1
t_2
t_3
t_4
t_5
t_6
t_7
t_8
t_9
t_{10}
t_{11}
t_{12}
t_{13}
t_{14}
t_{15}
t_{16}
t_{17}
b_i
interrupts, $\eta_i = 3$
s_i

Figure 14. Execution of a GPU-using job under klitirqd. By definition, s_i already incorporates Δ^{bh}.

Fig. 13 illustrates the sequence of events from a GPU interrupt, raised by the GPU device, to the completion of the interrupt bottom-half by klitirqd. Observe in Fig. 13 that immediately after the completion of an interrupt top-half at time t_3, the bottom-half is "released" (or queued) to klitirqd. 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 klitirqd. 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 klitirqd 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 klitirqd 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 klitirqd since s_i, by definition, already incorporates their execution time. This can be observed in Fig. 14.

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

Overhead	Duration
Δ^{sch}	$1.34\mu s$
Δ^{cxs}	$0.75\mu s$
Δ^{ipi}	$35.6\mu s$
Δ^{rel}	$1.52\mu s$
Δ^{th}	$15.79\mu s$
Δ^{bh}	$40.63\mu s$
Δ^{krel}	$3.78\mu s$
Δ^{tck}	$0.28\mu s$
Q	$1ms$

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

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 [28] and other sources, ticks can be accounted for with the following equation:

$$e_i'^{\text{tck}} = e_i' + \left\lceil \frac{p_i + x_i}{Q} \right\rceil \Delta^{tck}, \tag{17}$$

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 klitirqd 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-halfs. Thus, we merely group all types of bottom-halfs together and compute an average.

Table II 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} twice: once with klitirqd and once with standard Linux interrupt handling. Data on the frequency and duration of priority inversions was gathered. Figures for all of our gathered data are presented here.

Fig. 15 through Fig. 55 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. 56 through Fig. 96 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.

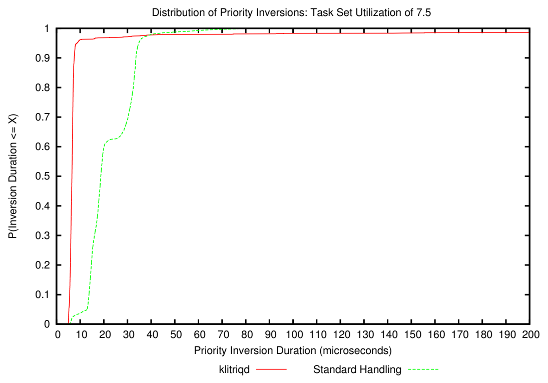


Figure 15. Task set utilization (prior inflation): 7.5. Cumulative distribtion of priority inversion durations. Higher curve is better.

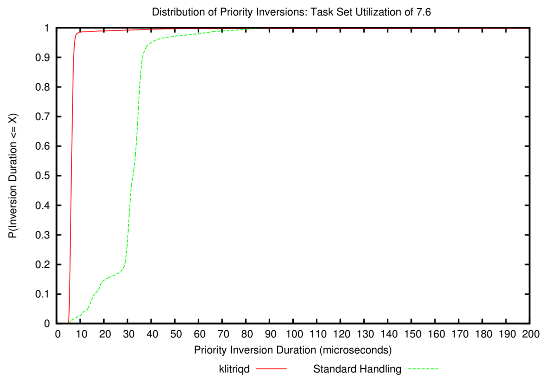


Figure 16. Task set utilization (prior inflation): 7.6. Cumulative distribtion of priority inversion durations. Higher curve is better.

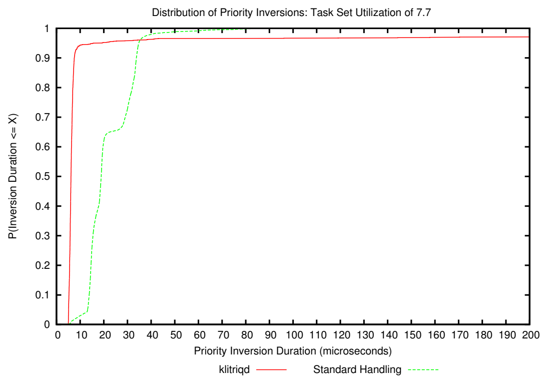


Figure 17. Task set utilization (prior inflation): 7.7. Cumulative distribtion of priority inversion durations. Higher curve is better.

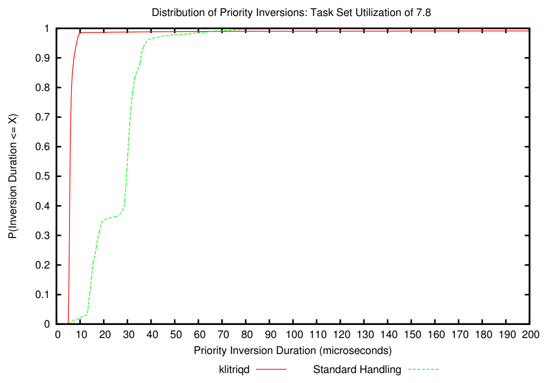


Figure 18. Task set utilization (prior inflation): 7.8. Cumulative distribtion of priority inversion durations. Higher curve is better.

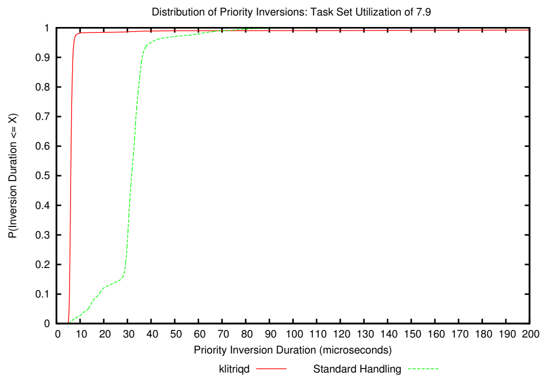


Figure 19. Task set utilization (prior inflation): 7.9. Cumulative distribtion of priority inversion durations. Higher curve is better.

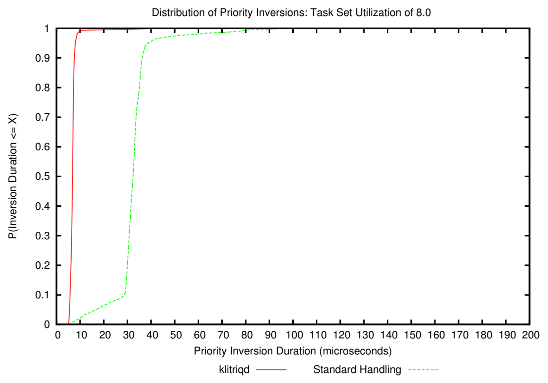


Figure 20. Task set utilization (prior inflation): 8.0. Cumulative distribtion of priority inversion durations. Higher curve is better.

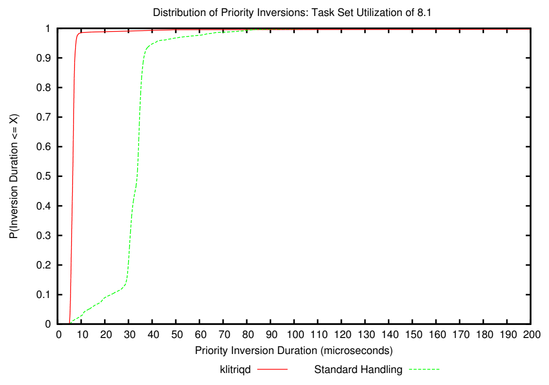


Figure 21. Task set utilization (prior inflation): 8.1. Cumulative distribtion of priority inversion durations. Higher curve is better.

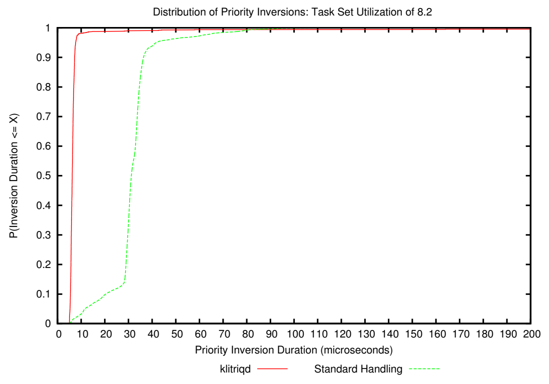


Figure 22. Task set utilization (prior inflation): 8.2. Cumulative distribtion of priority inversion durations. Higher curve is better.

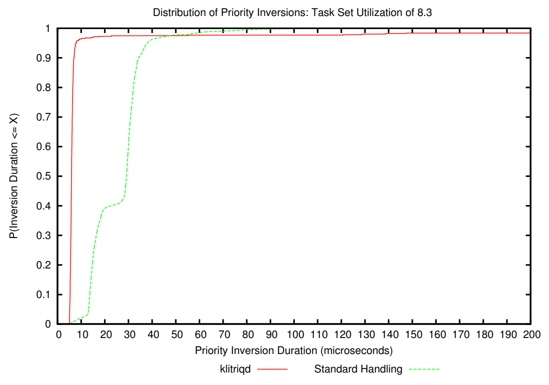


Figure 23. Task set utilization (prior inflation): 8.3. Cumulative distribtion of priority inversion durations. Higher curve is better.

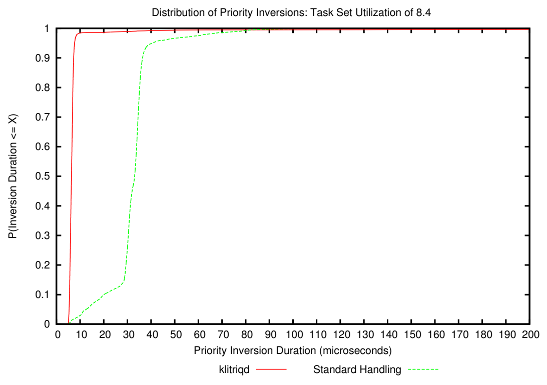


Figure 24. Task set utilization (prior inflation): 8.4. Cumulative distribtion of priority inversion durations. Higher curve is better.

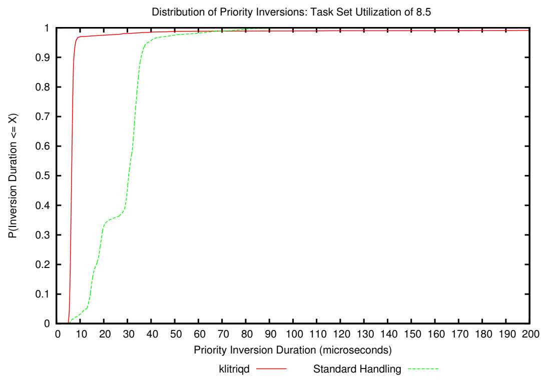


Figure 25. Task set utilization (prior inflation): 8.5. Cumulative distribtion of priority inversion durations. Higher curve is better.

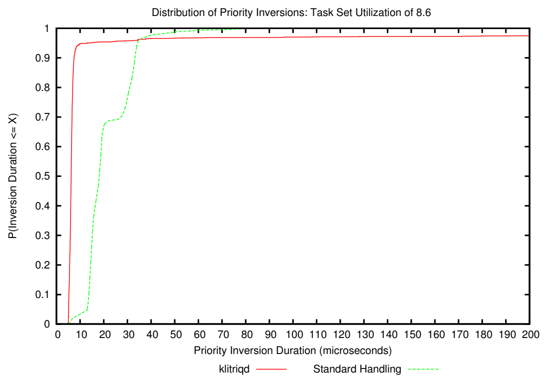


Figure 26. Task set utilization (prior inflation): 8.6. Cumulative distribtion of priority inversion durations. Higher curve is better.

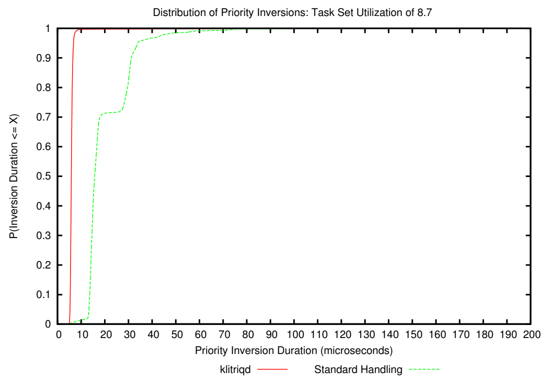


Figure 27. Task set utilization (prior inflation): 8.7. Cumulative distribtion of priority inversion durations. Higher curve is better.

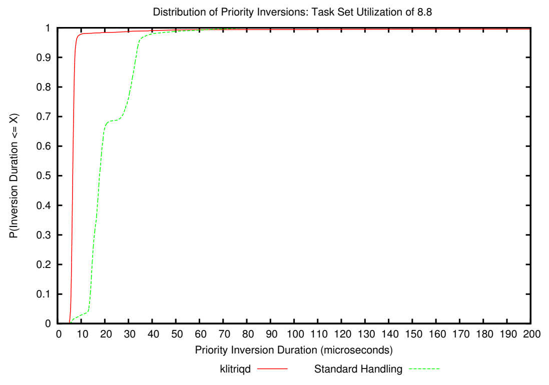


Figure 28. Task set utilization (prior inflation): 8.8. Cumulative distribtion of priority inversion durations. Higher curve is better.

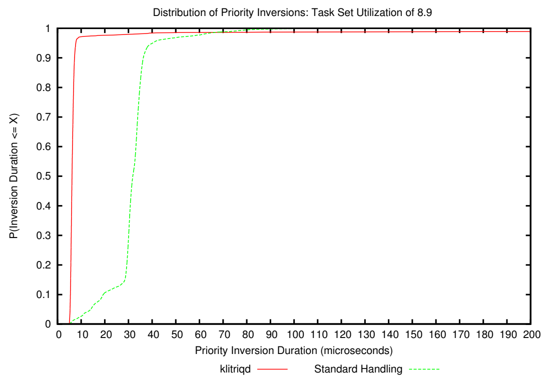


Figure 29. Task set utilization (prior inflation): 8.9. Cumulative distribtion of priority inversion durations. Higher curve is better.

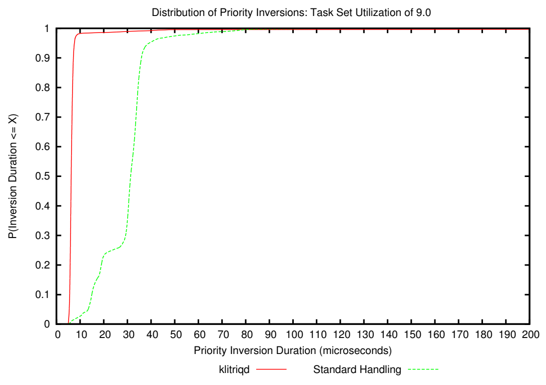


Figure 30. Task set utilization (prior inflation): 9.0. Cumulative distribtion of priority inversion durations. Higher curve is better.

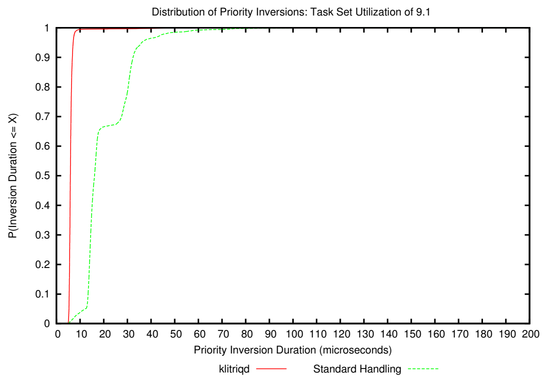


Figure 31. Task set utilization (prior inflation): 9.1. Cumulative distribtion of priority inversion durations. Higher curve is better.

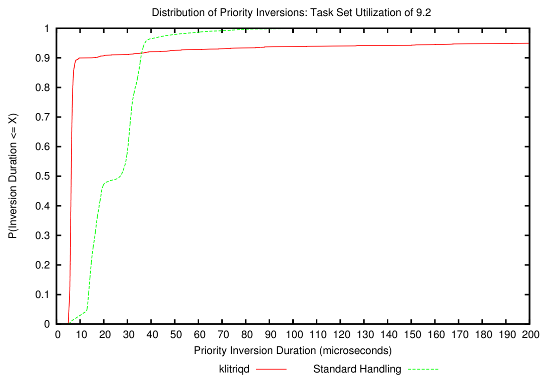


Figure 32. Task set utilization (prior inflation): 9.2. Cumulative distribtion of priority inversion durations. Higher curve is better.

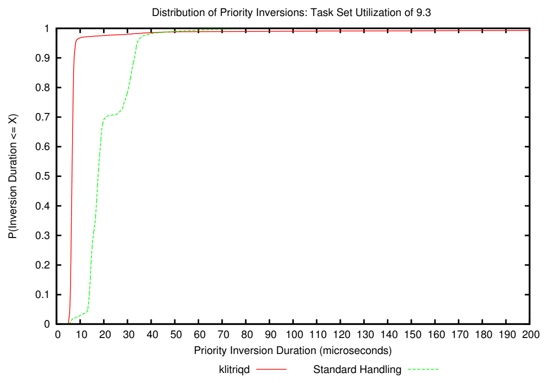


Figure 33. Task set utilization (prior inflation): 9.3. Cumulative distribtion of priority inversion durations. Higher curve is better.

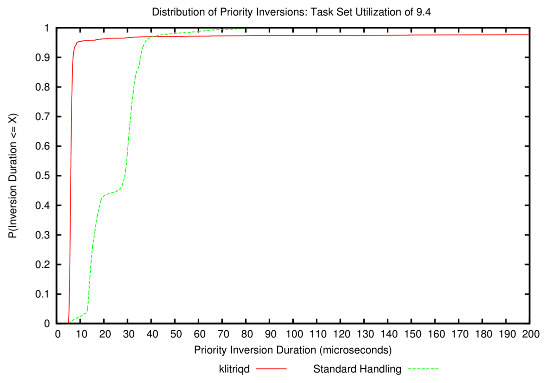


Figure 34. Task set utilization (prior inflation): 9.4. Cumulative distribtion of priority inversion durations. Higher curve is better.

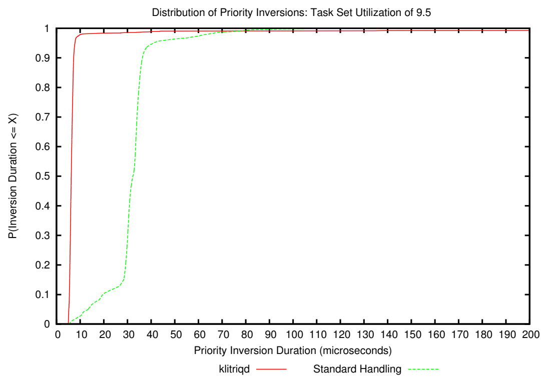


Figure 35. Task set utilization (prior inflation): 9.5. Cumulative distribtion of priority inversion durations. Higher curve is better.

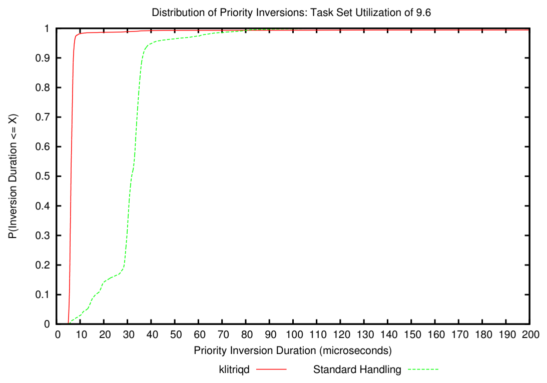


Figure 36. Task set utilization (prior inflation): 9.6. Cumulative distribtion of priority inversion durations. Higher curve is better.

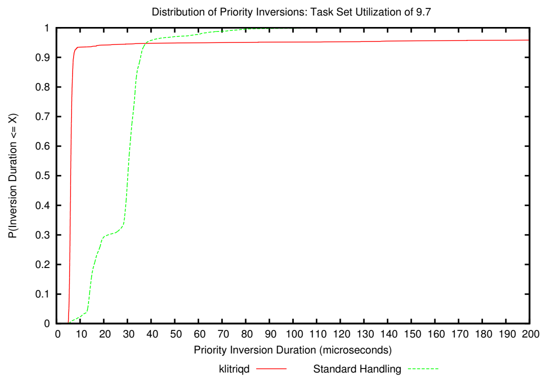


Figure 37. Task set utilization (prior inflation): 9.7. Cumulative distribtion of priority inversion durations. Higher curve is better.

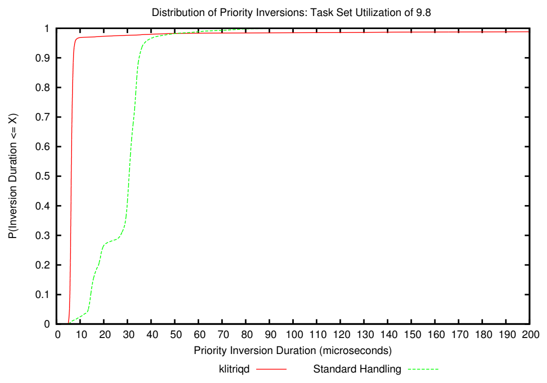


Figure 38. Task set utilization (prior inflation): 9.8. Cumulative distribtion of priority inversion durations. Higher curve is better.

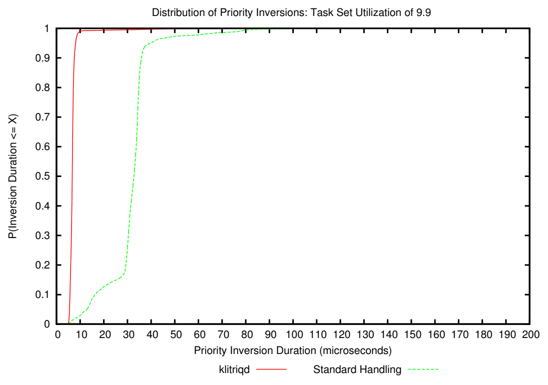


Figure 39. Task set utilization (prior inflation): 9.9. Cumulative distribtion of priority inversion durations. Higher curve is better.

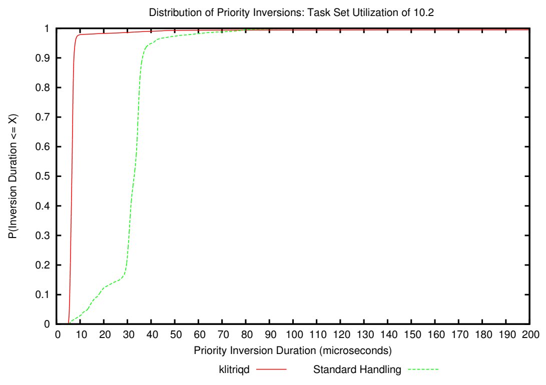


Figure 42. Task set utilization (prior inflation): 10.2. Cumulative distribtion of priority inversion durations. Higher curve is better.

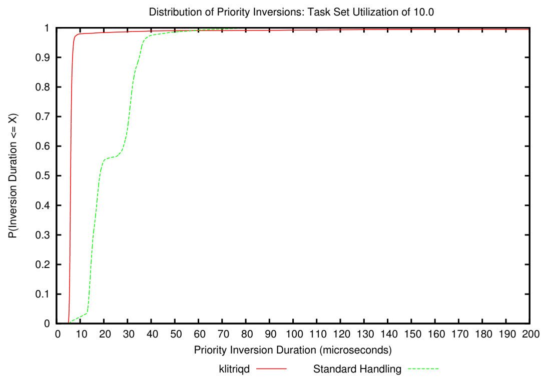


Figure 40. Task set utilization (prior inflation): 10.0. Cumulative distribtion of priority inversion durations. Higher curve is better.

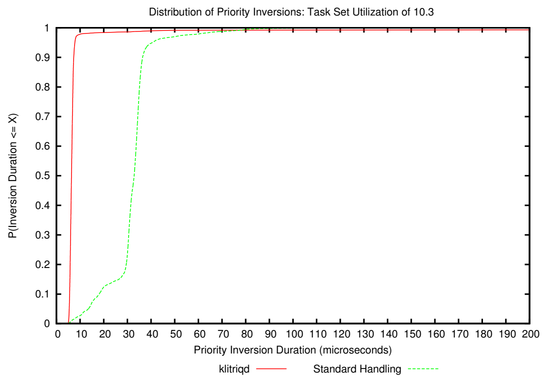


Figure 43. Task set utilization (prior inflation): 10.3. Cumulative distribtion of priority inversion durations. Higher curve is better.

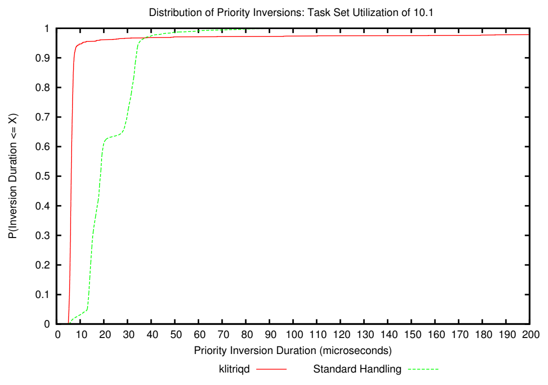


Figure 41. Task set utilization (prior inflation): 10.1. Cumulative distribtion of priority inversion durations. Higher curve is better.

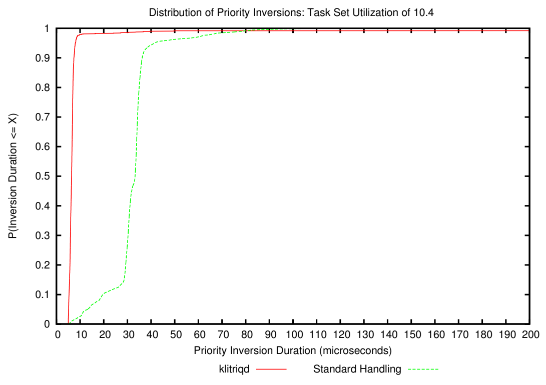


Figure 44. Task set utilization (prior inflation): 10.4. Cumulative distribtion of priority inversion durations. Higher curve is better.

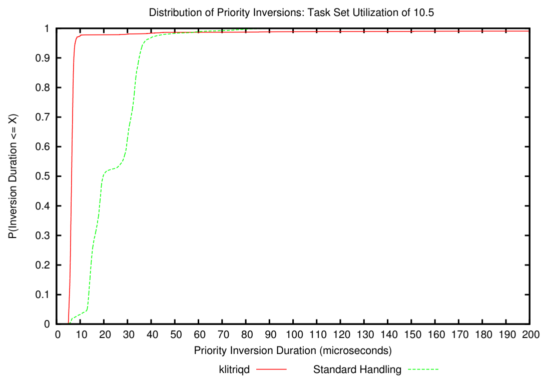


Figure 45. Task set utilization (prior inflation): 10.5. Cumulative distribtion of priority inversion durations. Higher curve is better.

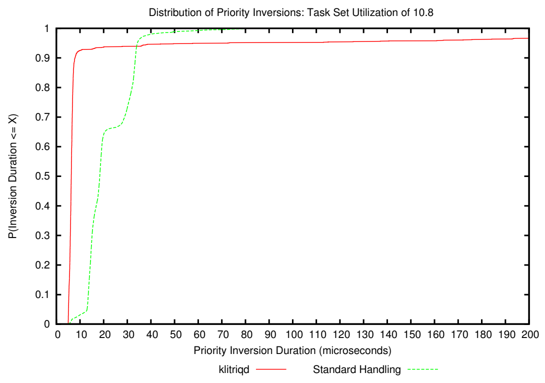


Figure 48. Task set utilization (prior inflation): 10.8. Cumulative distribtion of priority inversion durations. Higher curve is better.

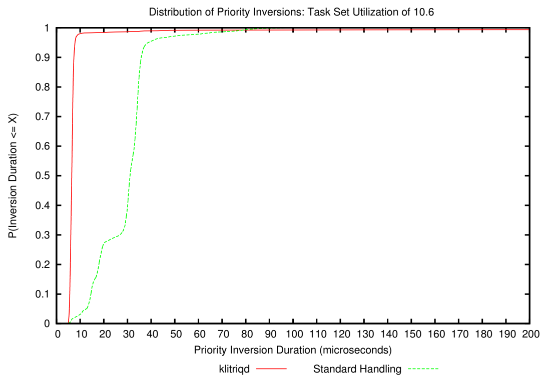


Figure 46. Task set utilization (prior inflation): 10.6. Cumulative distribtion of priority inversion durations. Higher curve is better.

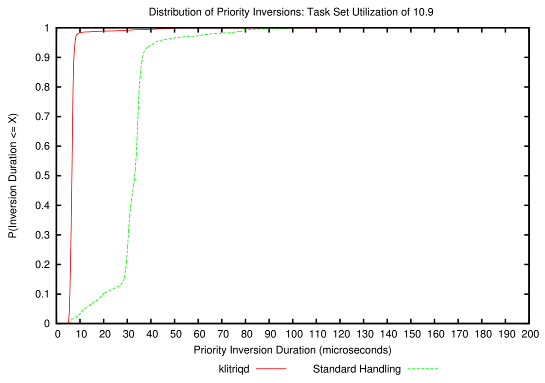


Figure 49. Task set utilization (prior inflation): 10.9. Cumulative distribtion of priority inversion durations. Higher curve is better.

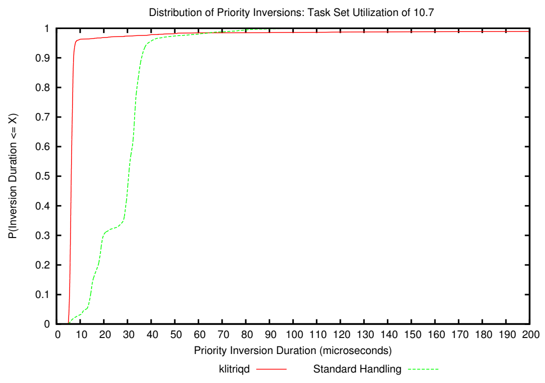


Figure 47. Task set utilization (prior inflation): 10.7. Cumulative distribtion of priority inversion durations. Higher curve is better.

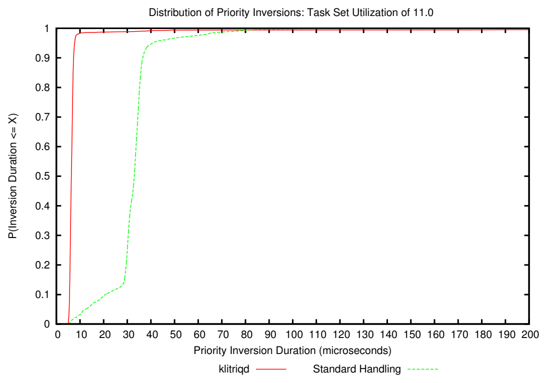


Figure 50. Task set utilization (prior inflation): 11.0. Cumulative distribtion of priority inversion durations. Higher curve is better.

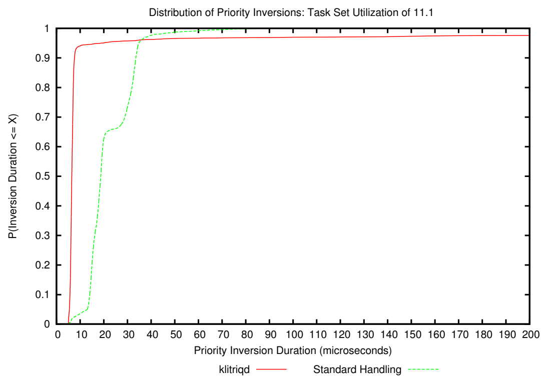


Figure 51. Task set utilization (prior inflation): 11.1. Cumulative distribtion of priority inversion durations. Higher curve is better.

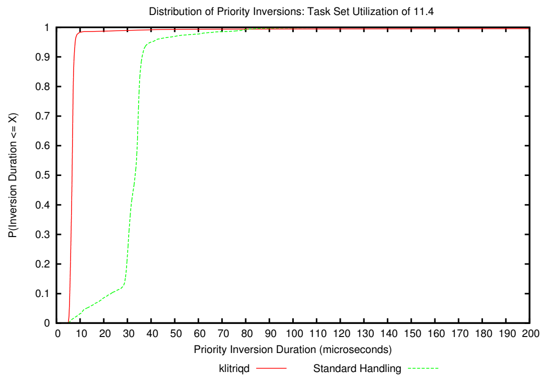


Figure 54. Task set utilization (prior inflation): 11.4. Cumulative distribtion of priority inversion durations. Higher curve is better.

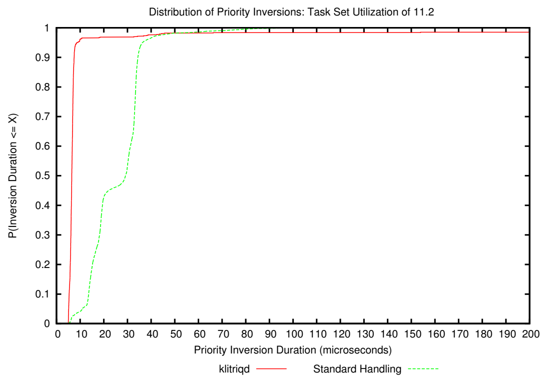


Figure 52. Task set utilization (prior inflation): 11.2. Cumulative distribtion of priority inversion durations. Higher curve is better.

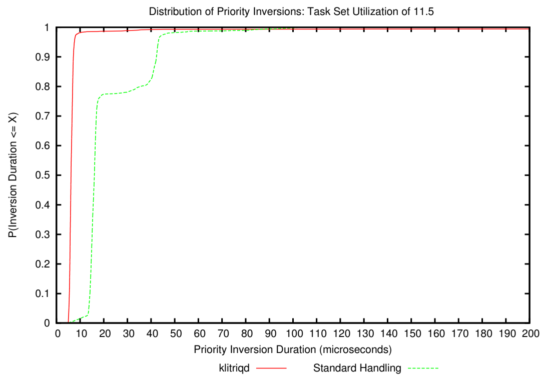


Figure 55. Task set utilization (prior inflation): 11.5. Cumulative distribtion of priority inversion durations. Higher curve is better.

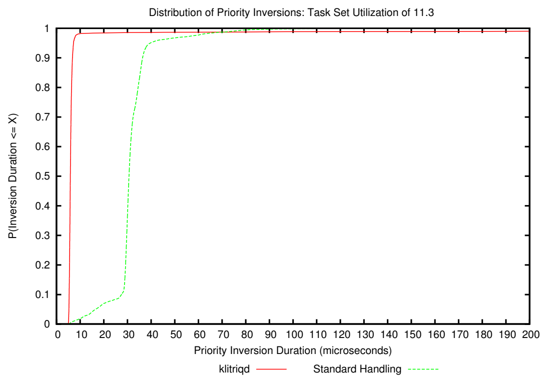


Figure 53. Task set utilization (prior inflation): 11.3. Cumulative distribtion of priority inversion durations. Higher curve is better.

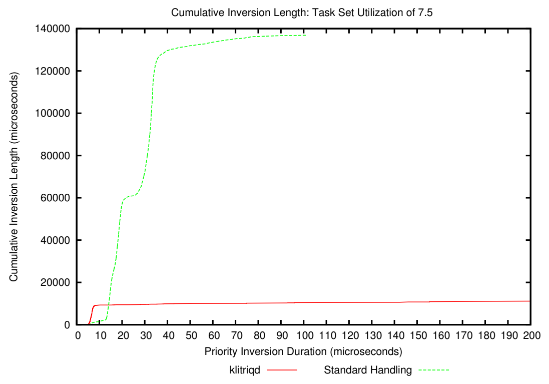


Figure 56. Task set utilization (prior inflation): 7.5. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

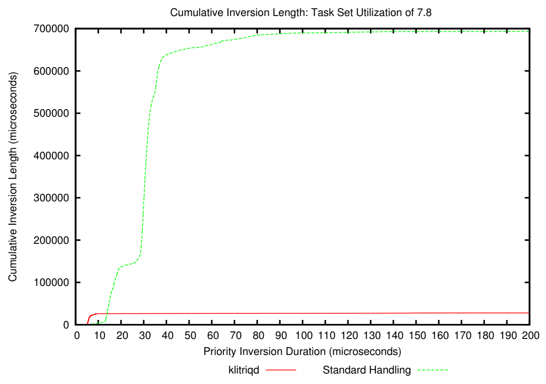


Figure 59. Task set utilization (prior inflation): 7.8. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

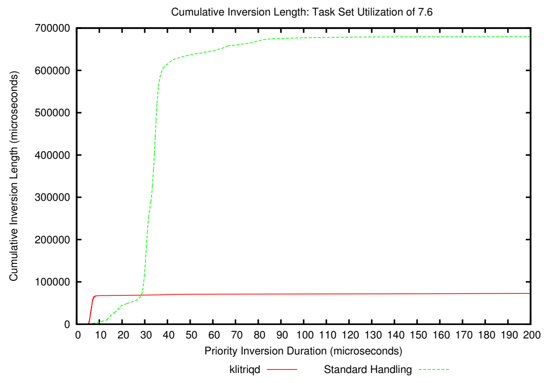


Figure 57. Task set utilization (prior inflation): 7.6. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

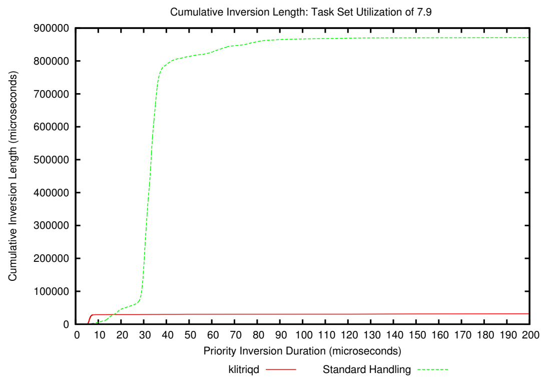


Figure 60. Task set utilization (prior inflation): 7.9. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

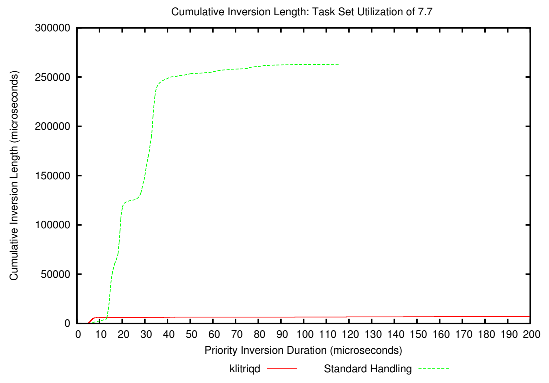


Figure 58. Task set utilization (prior inflation): 7.7. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

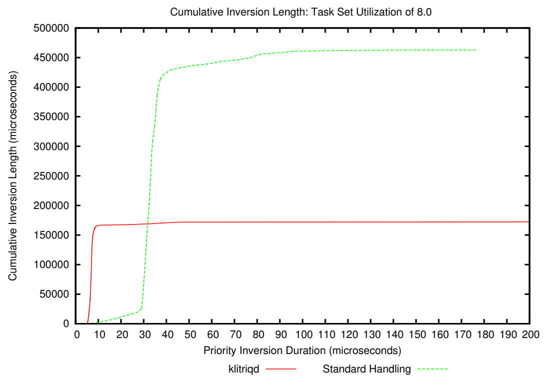


Figure 61. Task set utilization (prior inflation): 8.0. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

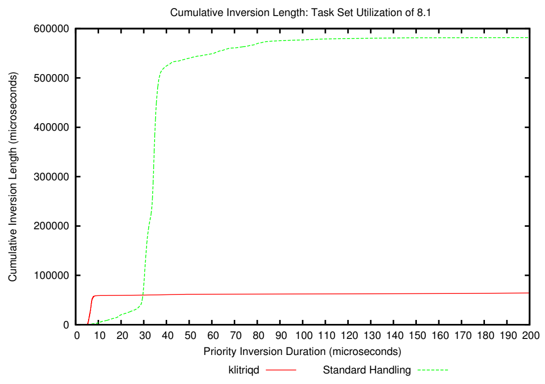


Figure 62. Task set utilization (prior inflation): 8.1. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

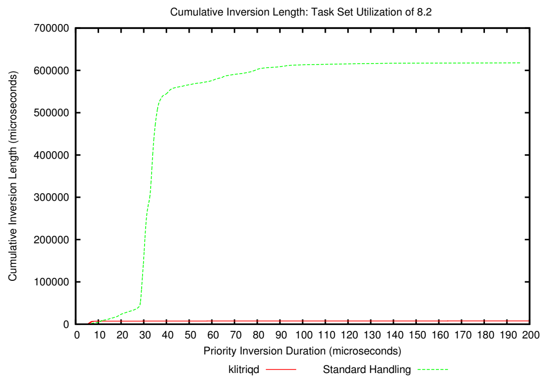


Figure 63. Task set utilization (prior inflation): 8.2. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

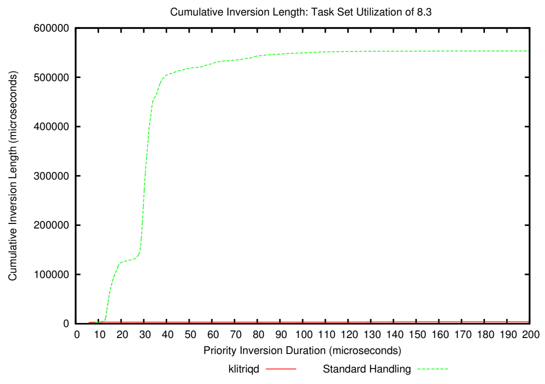


Figure 64. Task set utilization (prior inflation): 8.3. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

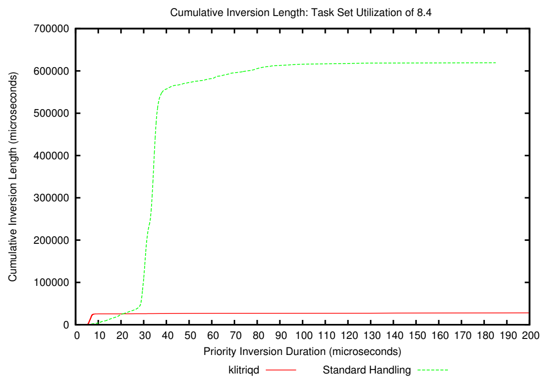


Figure 65. Task set utilization (prior inflation): 8.4. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

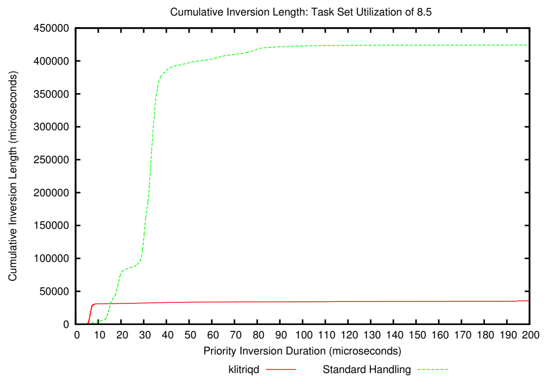


Figure 66. Task set utilization (prior inflation): 8.5. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

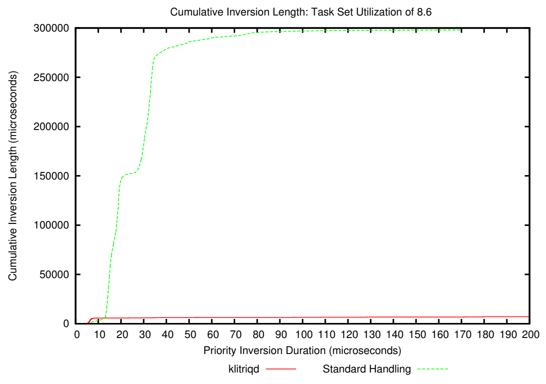


Figure 67. Task set utilization (prior inflation): 8.6. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

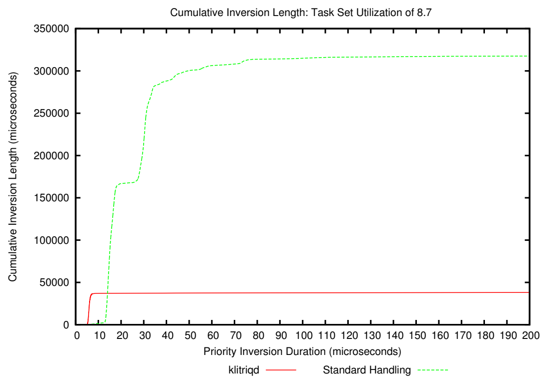


Figure 68. Task set utilization (prior inflation): 8.7. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

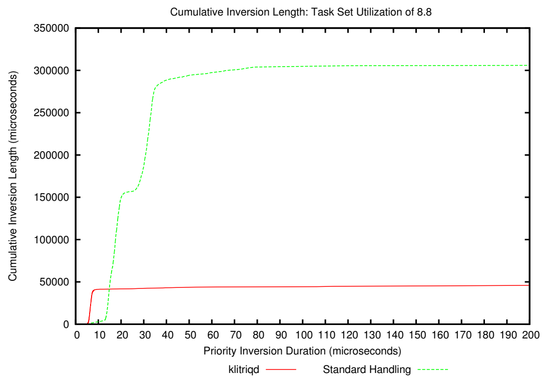


Figure 69. Task set utilization (prior inflation): 8.8. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

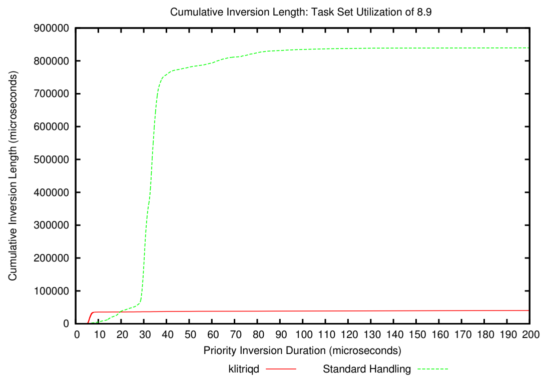


Figure 70. Task set utilization (prior inflation): 8.9. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

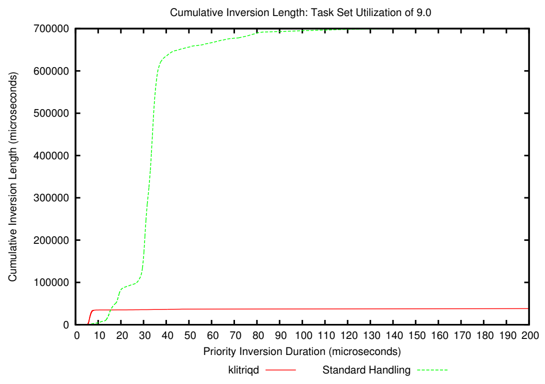


Figure 71. Task set utilization (prior inflation): 9.0. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

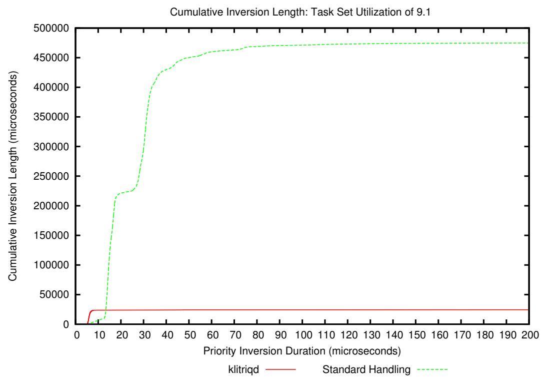


Figure 72. Task set utilization (prior inflation): 9.1. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

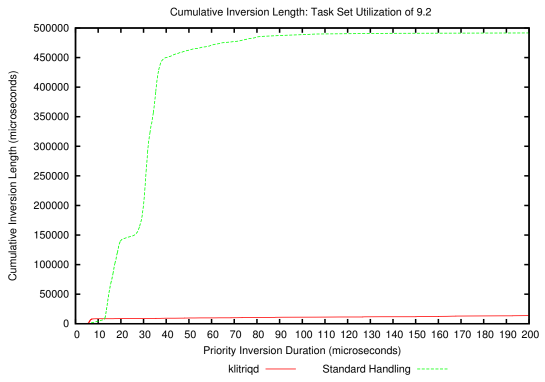


Figure 73. Task set utilization (prior inflation): 9.2. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

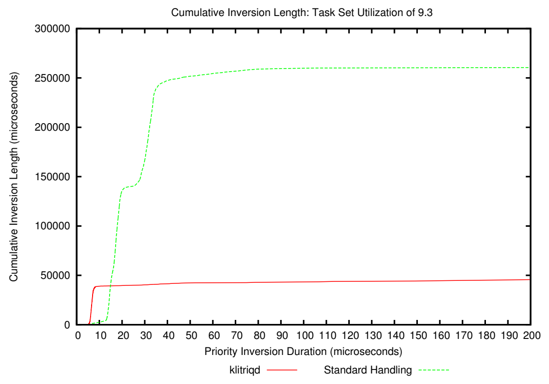


Figure 74. Task set utilization (prior inflation): 9.3. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

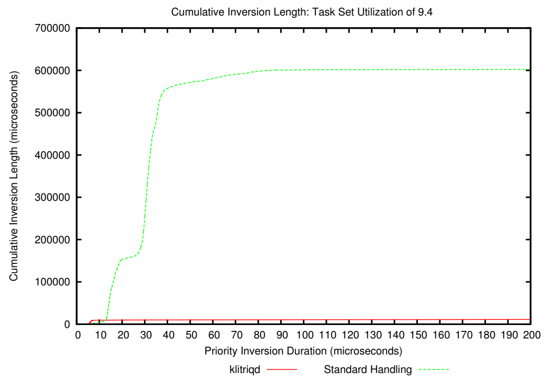


Figure 75. Task set utilization (prior inflation): 9.4. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

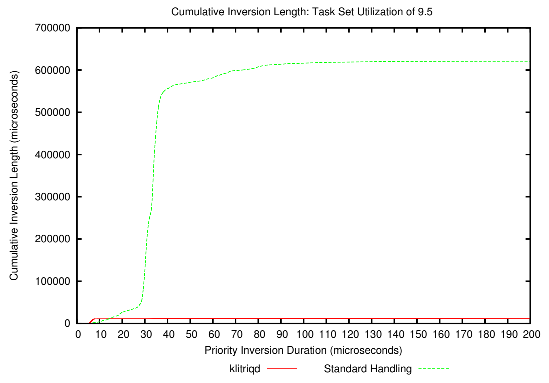


Figure 76. Task set utilization (prior inflation): 9.5. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

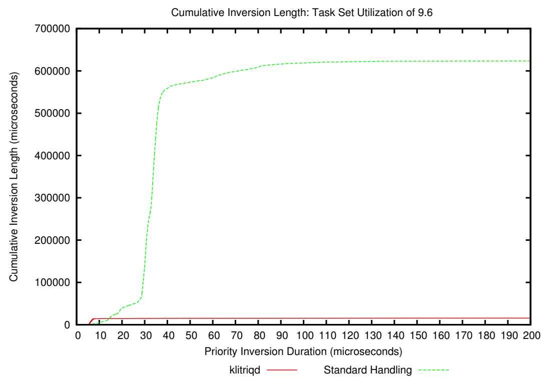


Figure 77. Task set utilization (prior inflation): 9.6. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

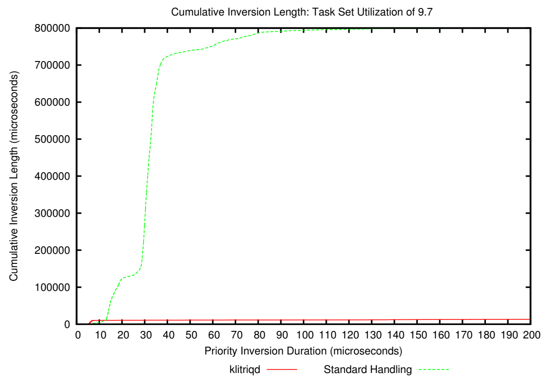


Figure 78. Task set utilization (prior inflation): 9.7. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

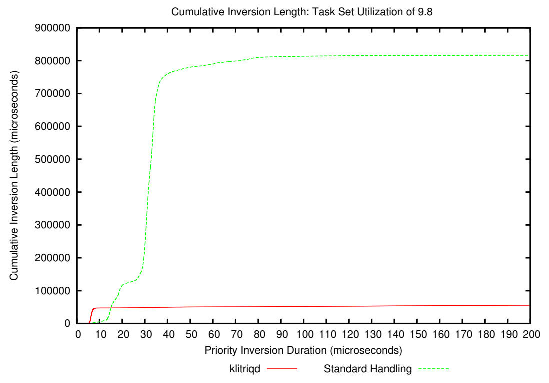


Figure 79. Task set utilization (prior inflation): 9.8. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

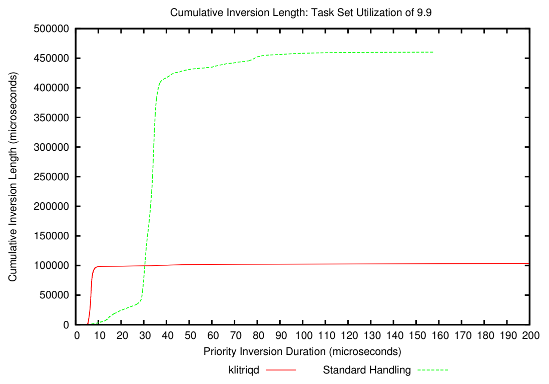


Figure 80. Task set utilization (prior inflation): 9.9. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

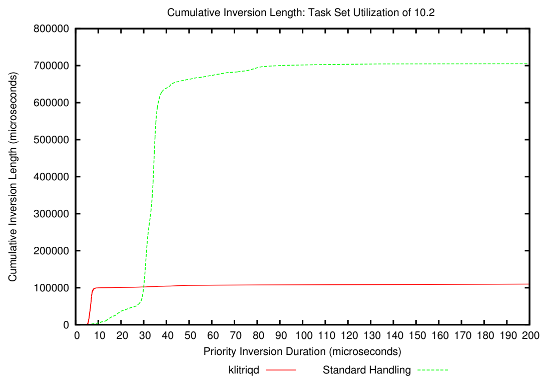


Figure 83. Task set utilization (prior inflation): 10.2. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

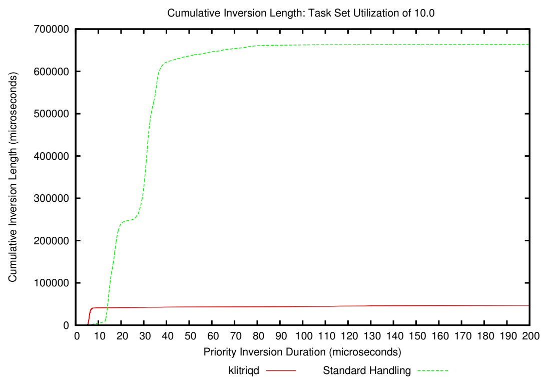


Figure 81. Task set utilization (prior inflation): 10.0. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

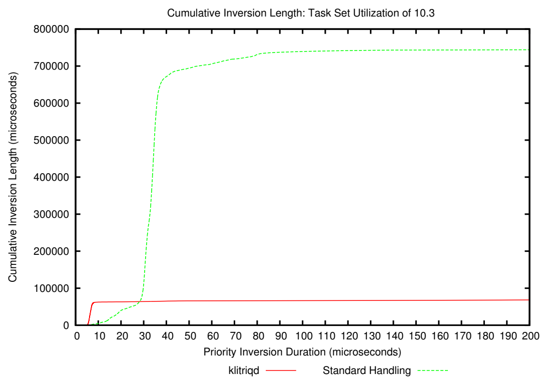


Figure 84. Task set utilization (prior inflation): 10.3. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

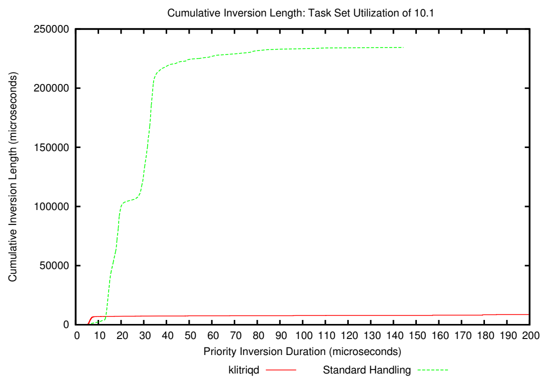


Figure 82. Task set utilization (prior inflation): 10.1. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

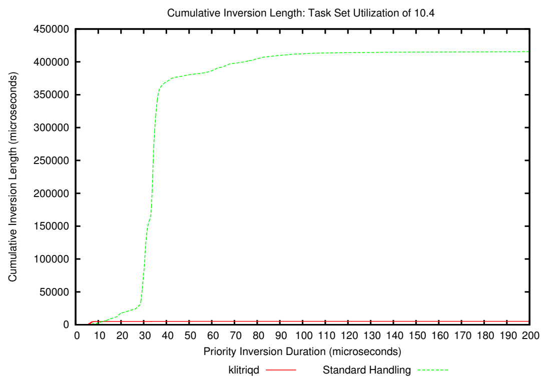


Figure 85. Task set utilization (prior inflation): 10.4. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

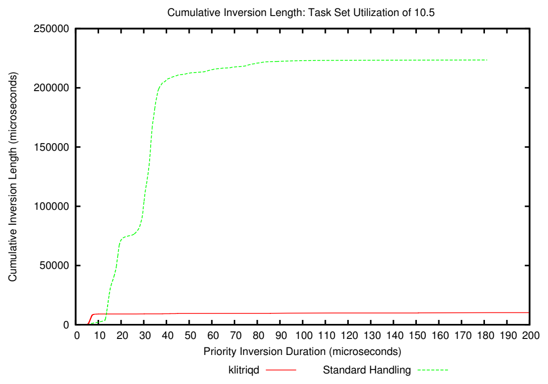


Figure 86. Task set utilization (prior inflation): 10.5. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

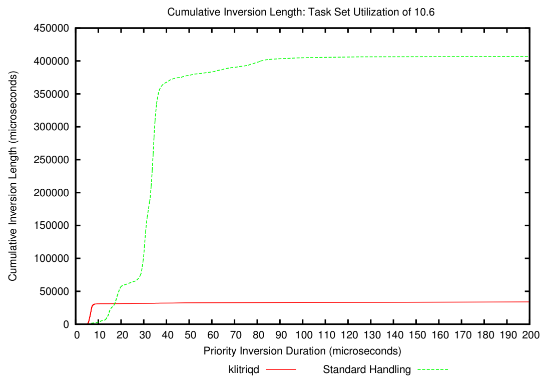


Figure 87. Task set utilization (prior inflation): 10.6. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

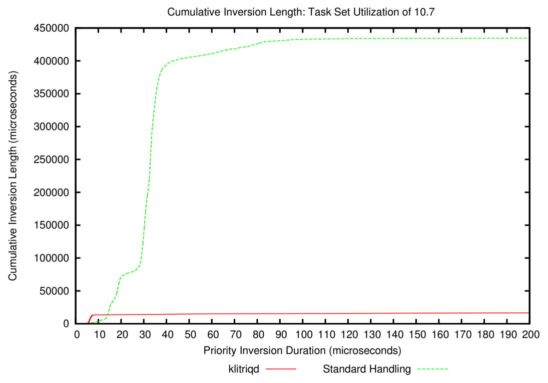


Figure 88. Task set utilization (prior inflation): 10.7. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

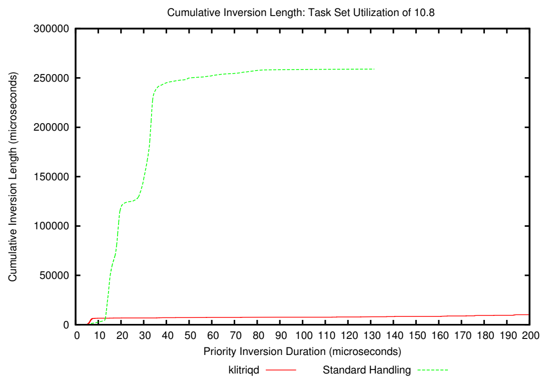


Figure 89. Task set utilization (prior inflation): 10.8. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

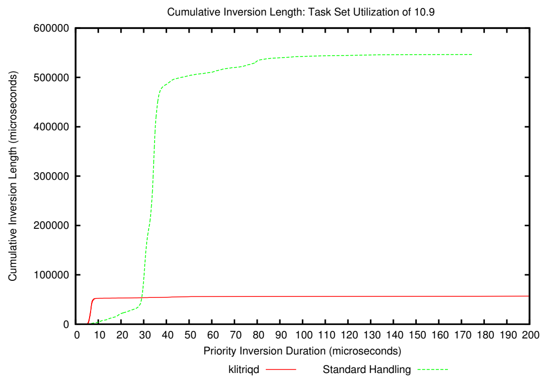


Figure 90. Task set utilization (prior inflation): 10.9. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

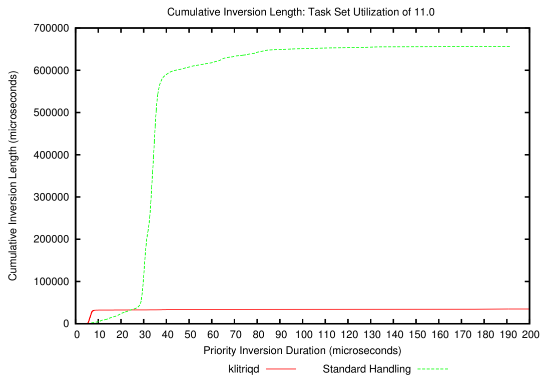


Figure 91. Task set utilization (prior inflation): 11.0. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

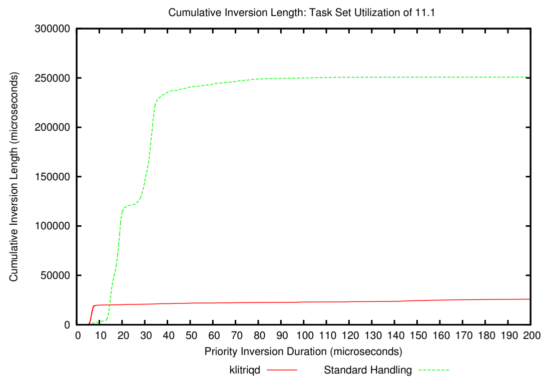


Figure 92. Task set utilization (prior inflation): 11.1. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

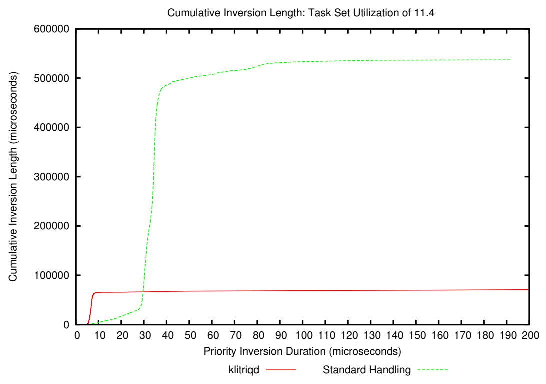


Figure 95. Task set utilization (prior inflation): 11.4. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

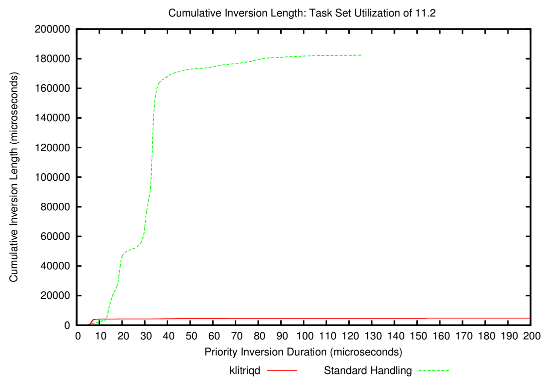


Figure 93. Task set utilization (prior inflation): 11.2. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

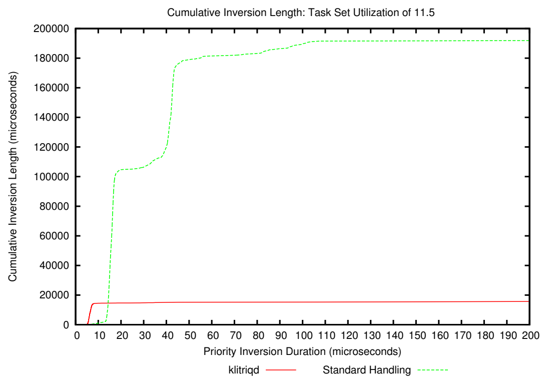


Figure 96. Task set utilization (prior inflation): 11.5. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

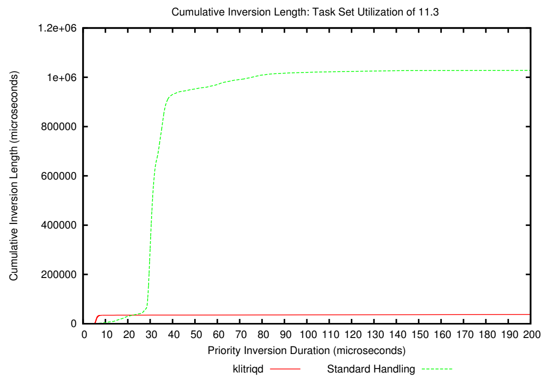


Figure 94. Task set utilization (prior inflation): 11.3. Cumulative priority inversion duration as a function of maximum priority inversion duration. Lower curve is better.

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 tests where per-task utilizations were *uniformly* selected at random from a given utilization range. These ranges were $[1\%, 10\%]$, $[10\%, 40\%]$, and $[50\%, 90\%]$. Within each given utilization range, the percentage share of GPU-using tasks was varied in 10% blocks (yielding ten graphs for each utilization range). 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 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.

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

Per-Task Utilization Distribution	x-axis	Per-Task Utilization Range or Average	Figures
Uniform	CPU Utilization, $R = 1$	$[1\%, 10\%]$	Fig. 97 to Fig. 106
		$[10\%, 40\%]$	Fig. 107 to Fig. 116
		$[50\%, 90\%]$	Fig. 117 to Fig. 126
	Effective Utilization, $R = 4$	$[1\%, 10\%]$	Fig. 127 to Fig. 136
		$[10\%, 40\%]$	Fig. 137 to Fig. 146
		$[50\%, 90\%]$	Fig. 147 to Fig. 156
	Effective Utilization, $R = 8$	$[1\%, 10\%]$	Fig. 157 to Fig. 166
		$[10\%, 40\%]$	Fig. 167 to Fig. 176
		$[50\%, 90\%]$	Fig. 177 to Fig. 186
	Effective Utilization, $R = 16$	$[1\%, 10\%]$	Fig. 187 to Fig. 196
		$[10\%, 40\%]$	Fig. 197 to Fig. 206
		$[50\%, 90\%]$	Fig. 207 to Fig. 216
Exponential	CPU Utilization, $R = 1$	10%	Fig. 217 to Fig. 226
		25%	Fig. 227 to Fig. 236
		50%	Fig. 237 to Fig. 246
	Effective Utilization, $R = 4$	10%	Fig. 247 to Fig. 256
		25%	Fig. 257 to Fig. 266
		50%	Fig. 267 to Fig. 276
	Effective Utilization, $R = 8$	10%	Fig. 277 to Fig. 286
		25%	Fig. 287 to Fig. 296
		50%	Fig. 297 to Fig. 306
	Effective Utilization, $R = 16$	10%	Fig. 307 to Fig. 316
		25%	Fig. 317 to Fig. 326
		50%	Fig. 327 to Fig. 336

Table III
ORGANIZATION OF FIGURED FOR SCHEDULABILITY EXPERIMENTS.

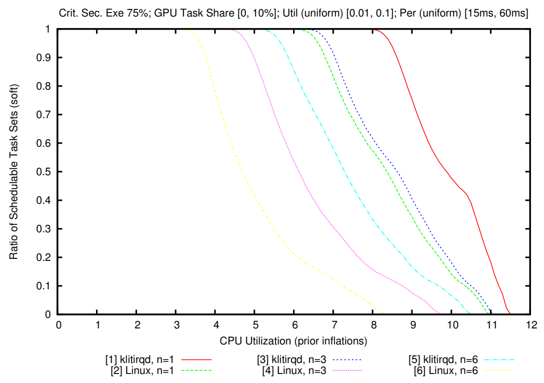


Figure 97. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

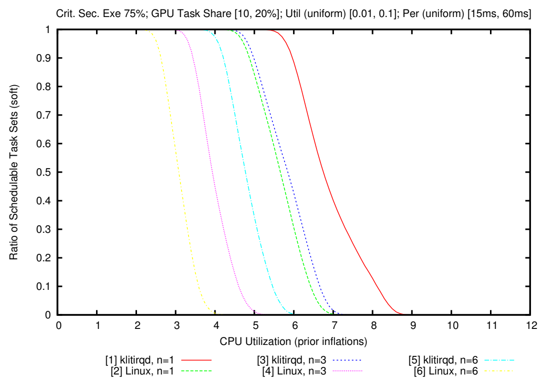


Figure 98. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

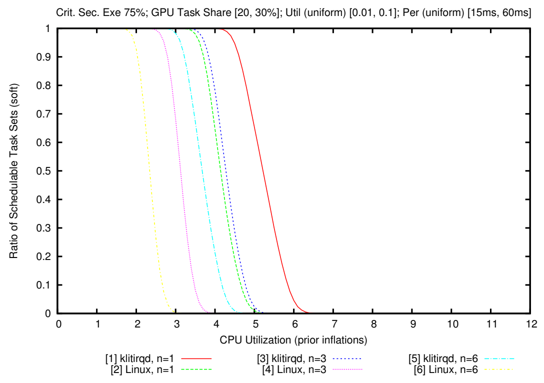


Figure 99. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

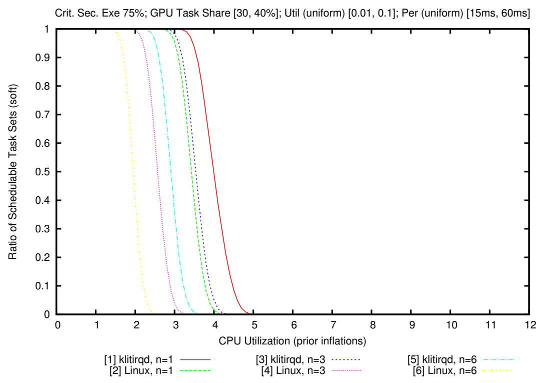


Figure 100. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

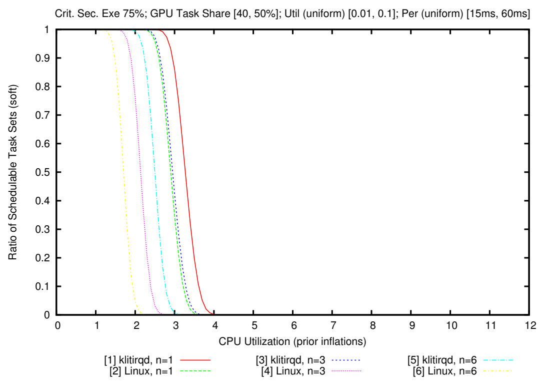


Figure 101. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

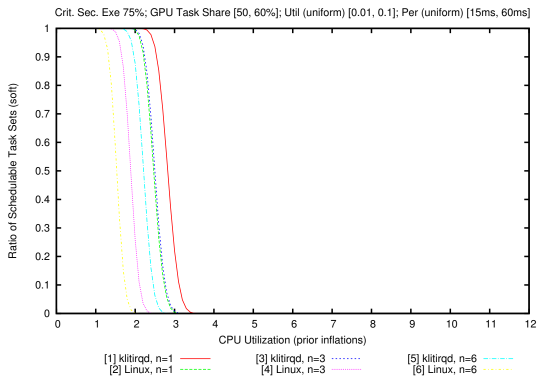


Figure 102. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

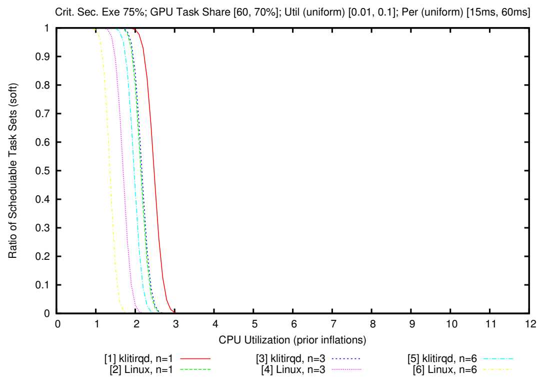


Figure 103. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

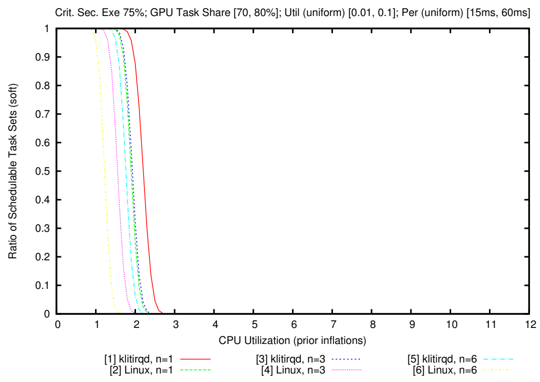


Figure 104. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

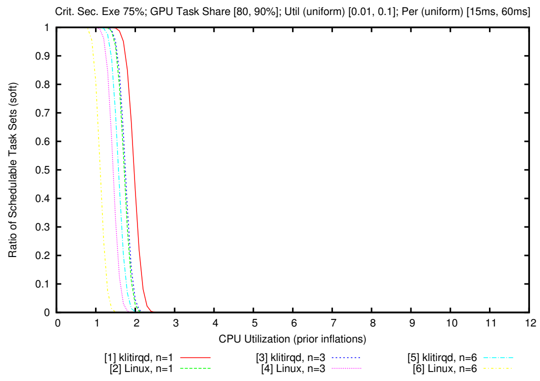


Figure 105. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

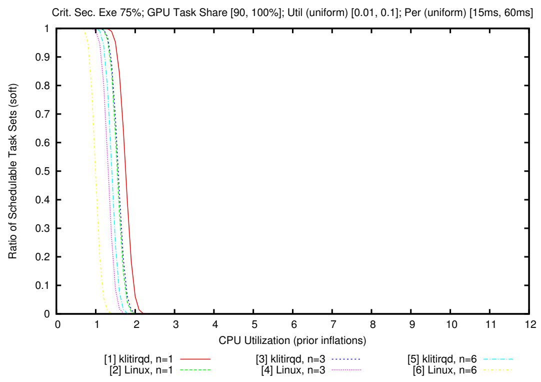


Figure 106. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

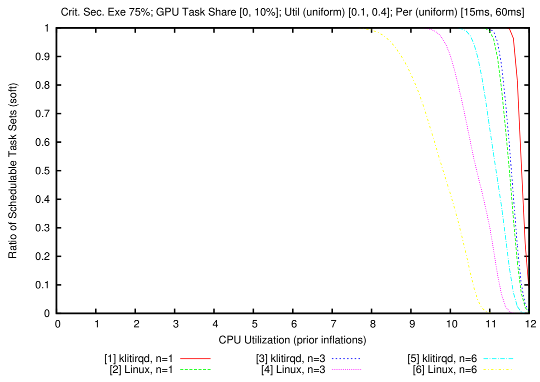


Figure 107. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

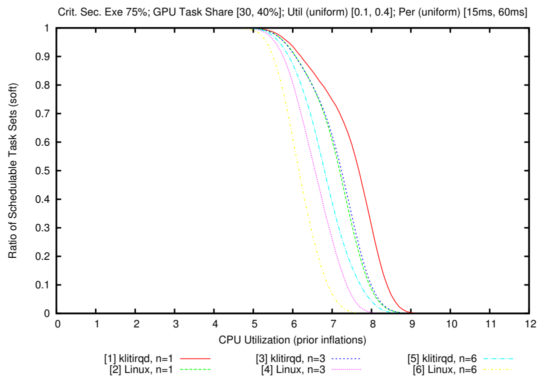


Figure 110. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

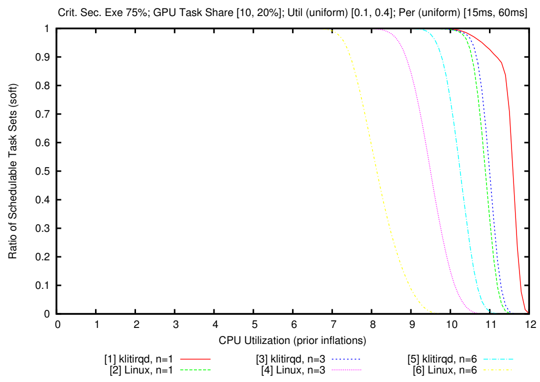


Figure 108. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

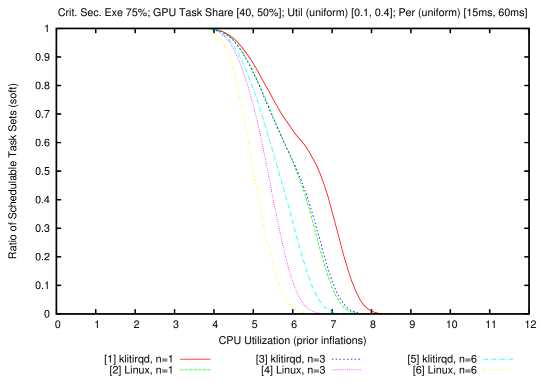


Figure 111. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

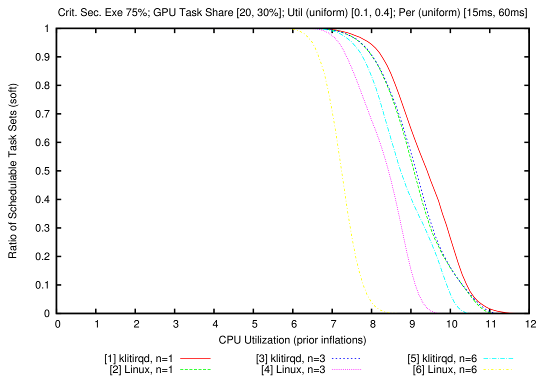


Figure 109. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

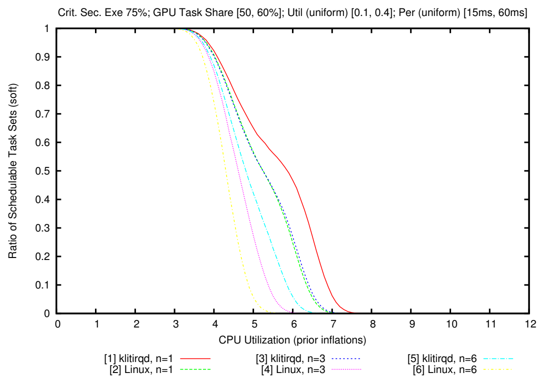


Figure 112. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

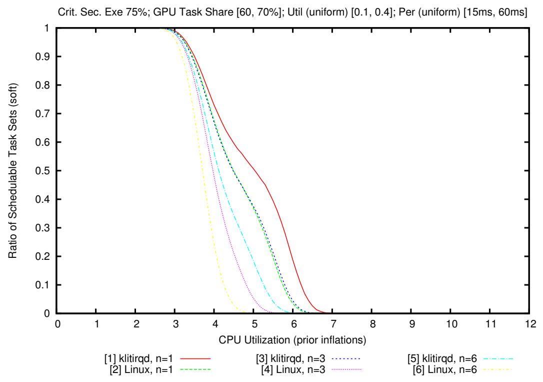


Figure 113. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

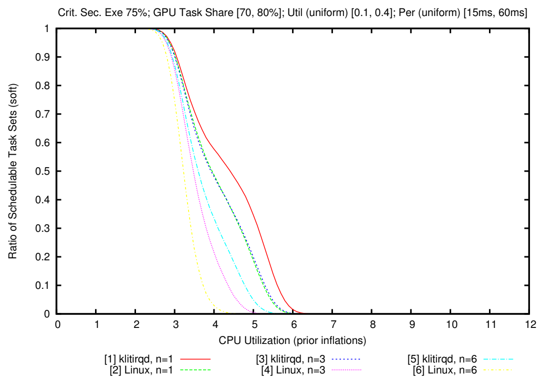


Figure 114. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

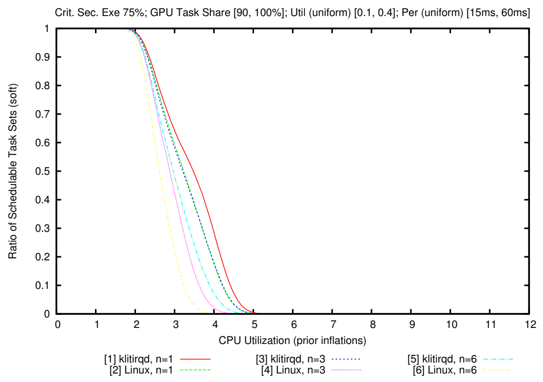


Figure 116. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

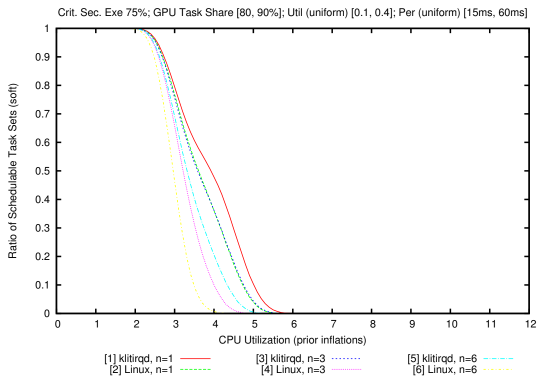


Figure 115. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

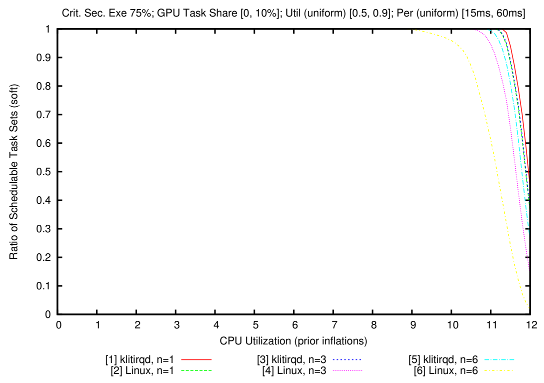


Figure 117. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

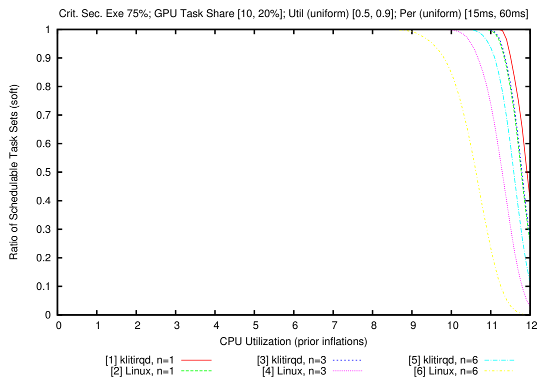


Figure 118. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

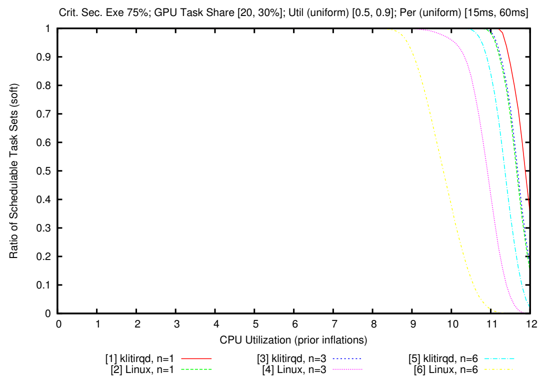


Figure 119. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

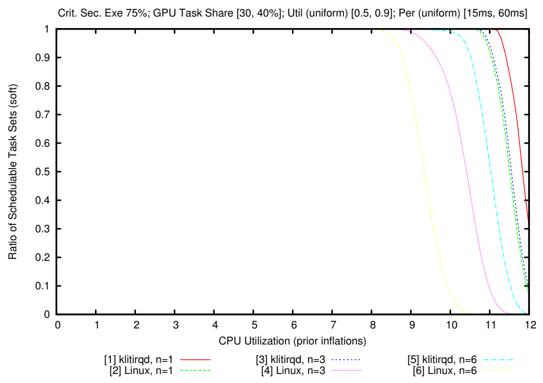


Figure 120. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

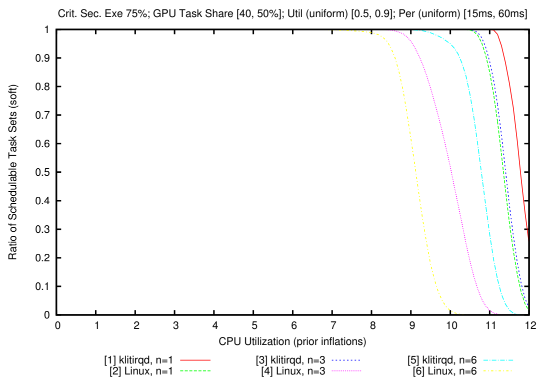


Figure 121. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

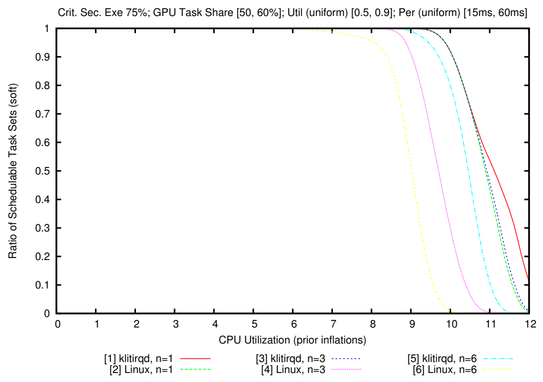


Figure 122. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

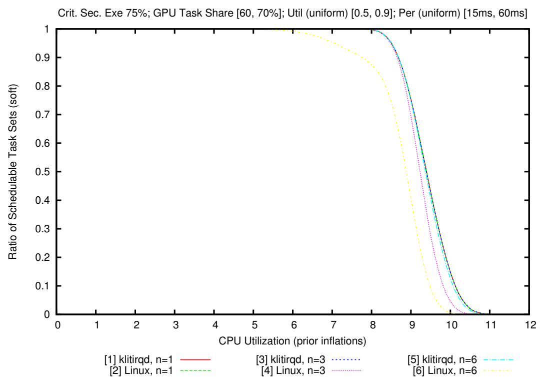


Figure 123. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

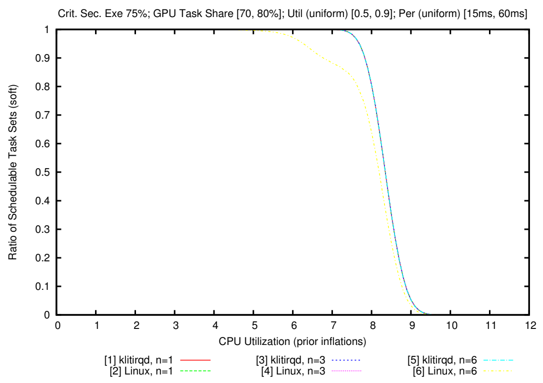


Figure 124. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

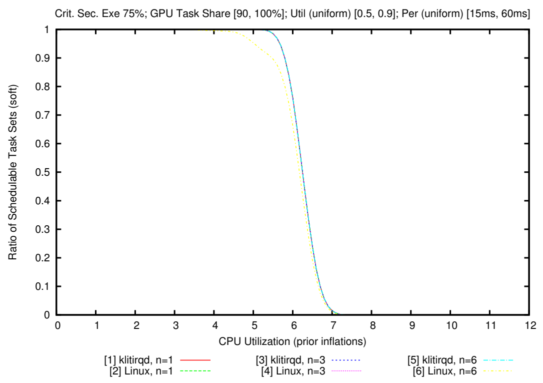


Figure 126. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

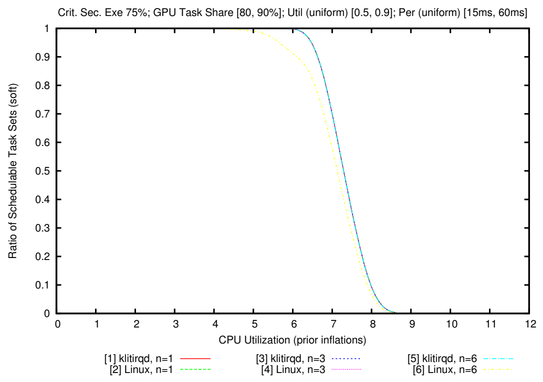


Figure 125. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

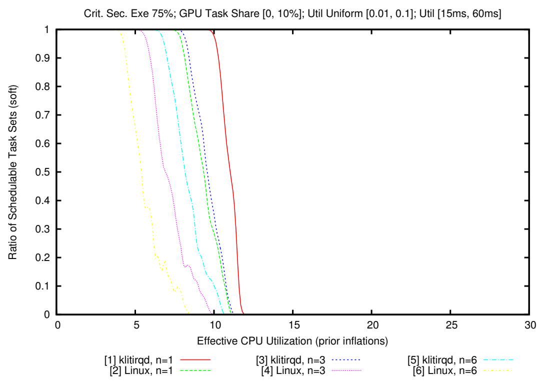


Figure 127. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

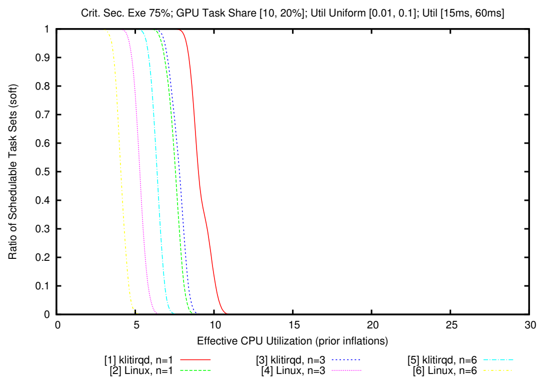


Figure 128. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

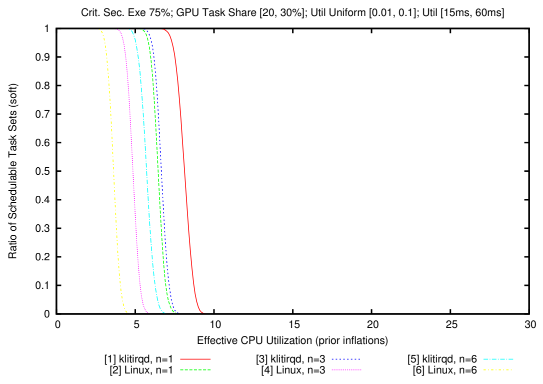


Figure 129. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

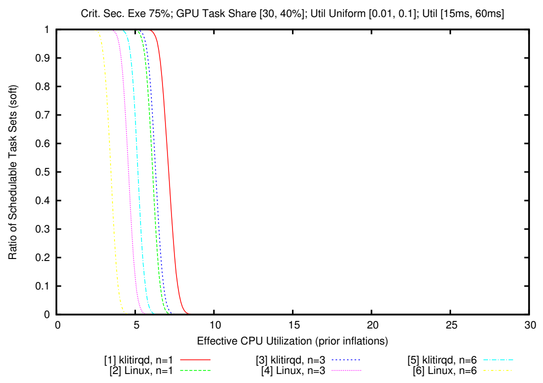


Figure 130. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

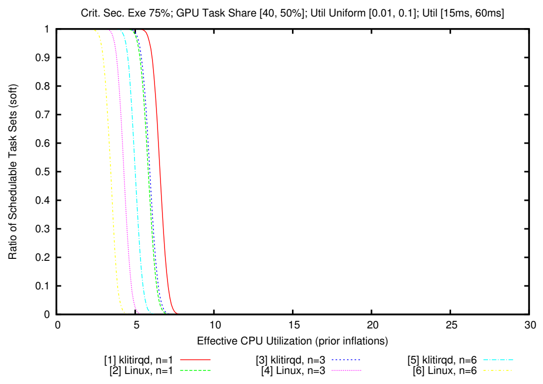


Figure 131. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

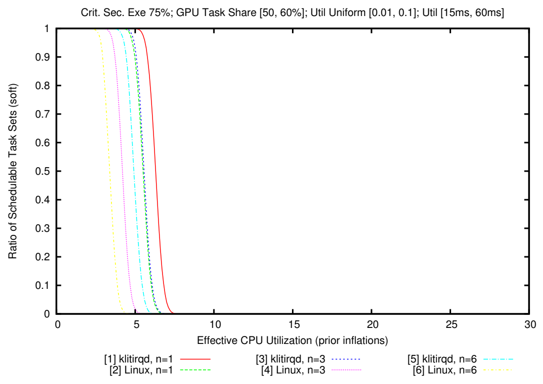


Figure 132. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

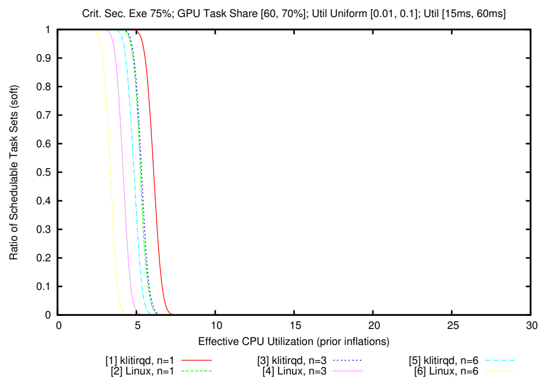


Figure 133. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

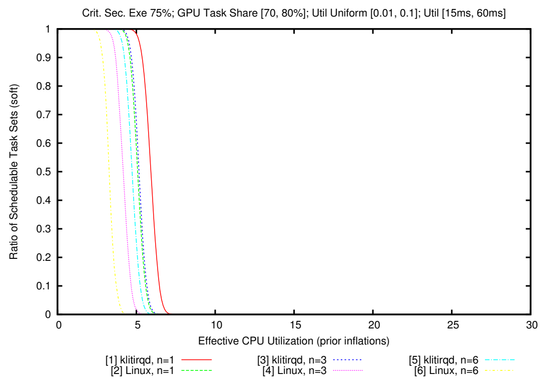


Figure 134. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

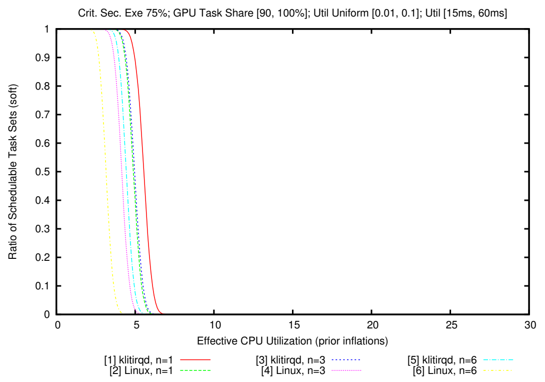


Figure 136. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

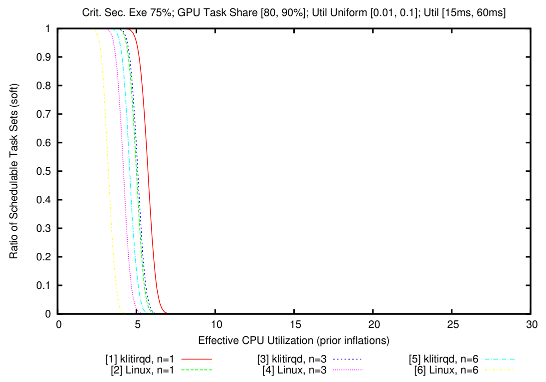


Figure 135. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

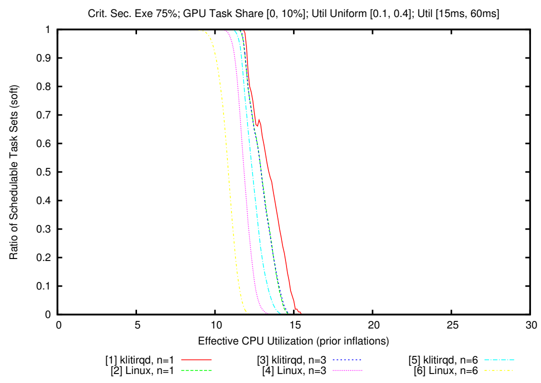


Figure 137. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

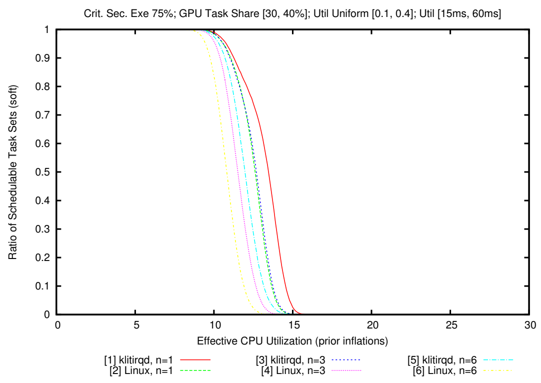


Figure 140. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

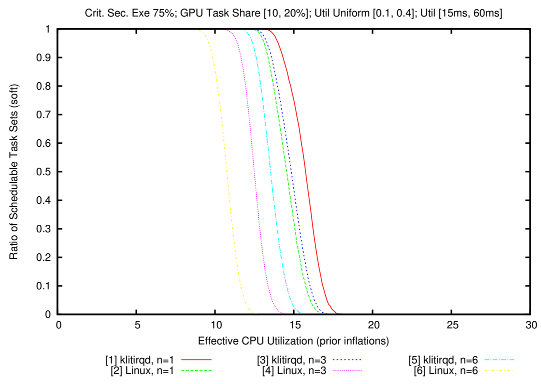


Figure 138. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

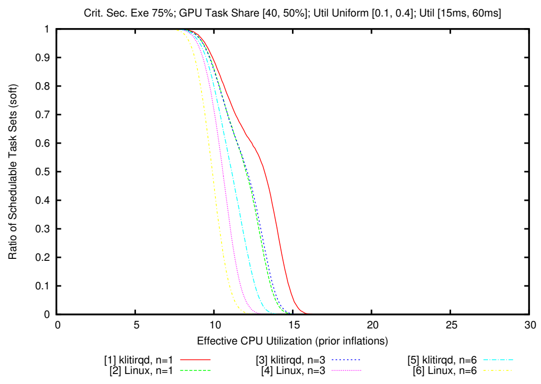


Figure 141. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

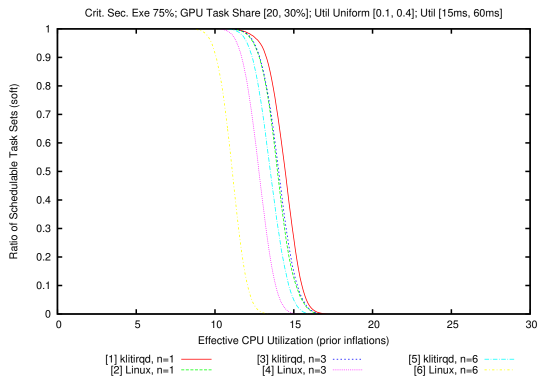


Figure 139. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

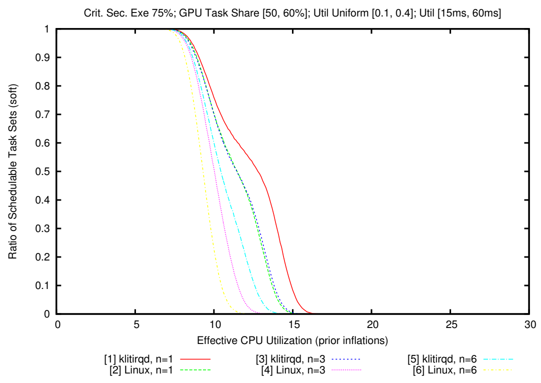


Figure 142. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

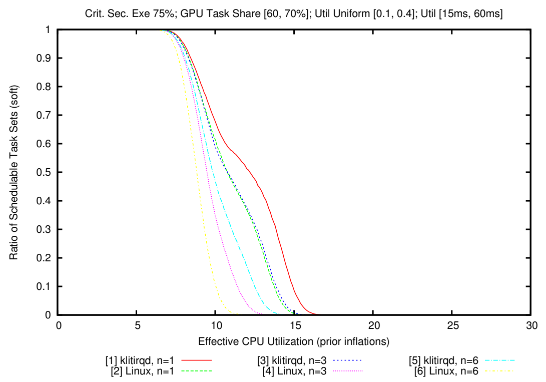


Figure 143. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

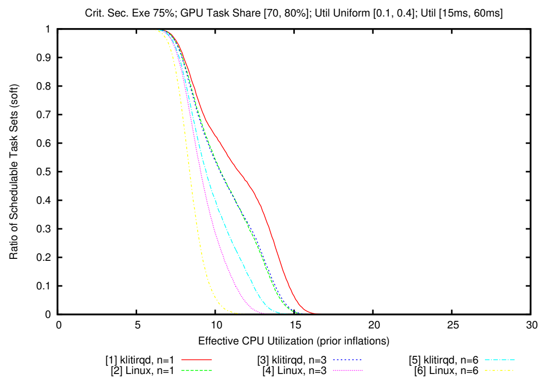


Figure 144. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

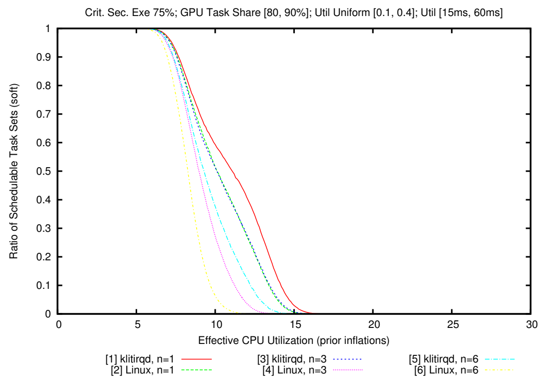


Figure 145. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

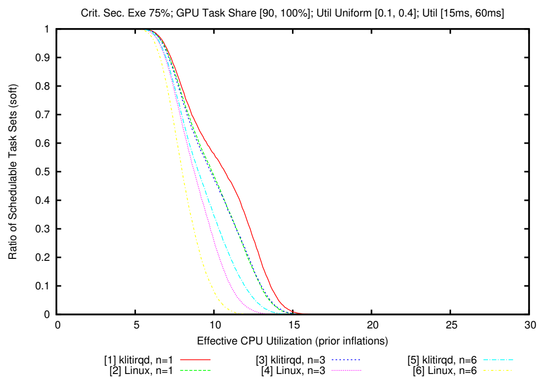


Figure 146. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

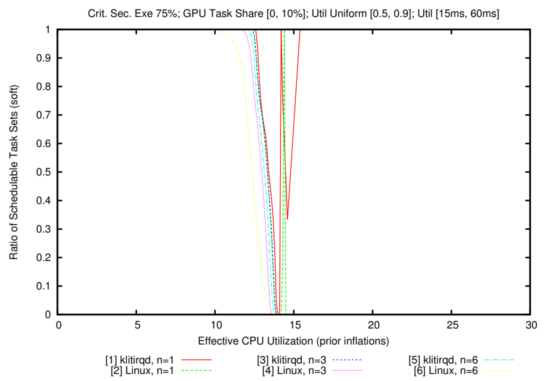


Figure 147. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

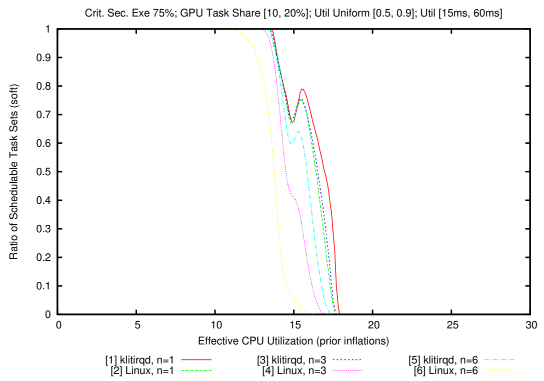


Figure 148. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

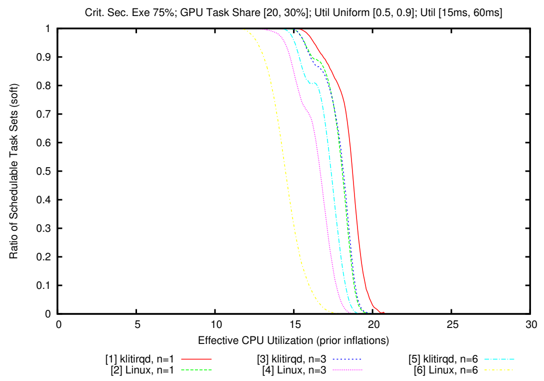


Figure 149. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

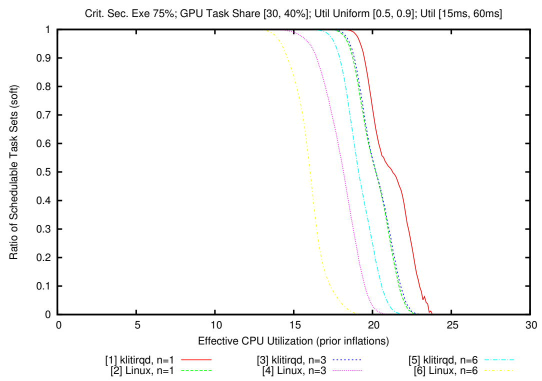


Figure 150. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

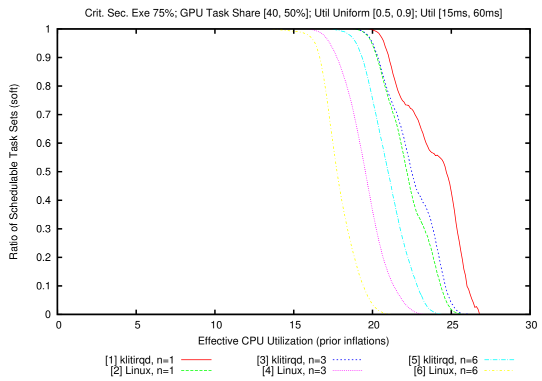


Figure 151. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

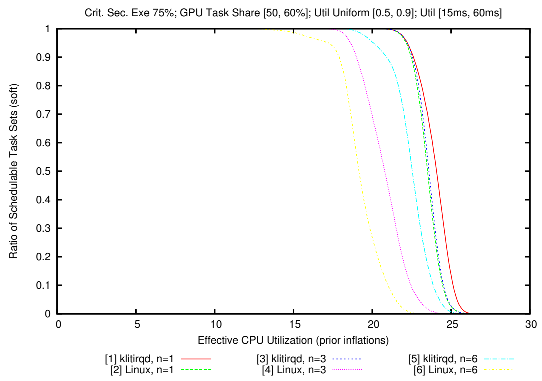


Figure 152. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

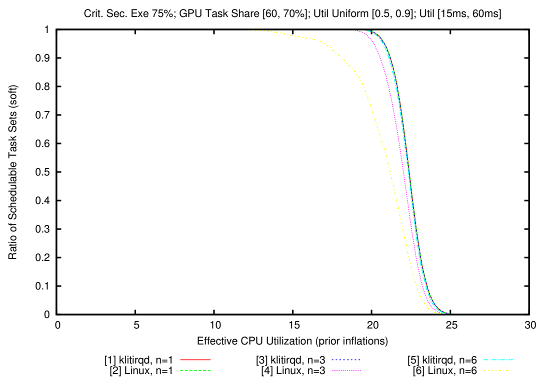


Figure 153. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

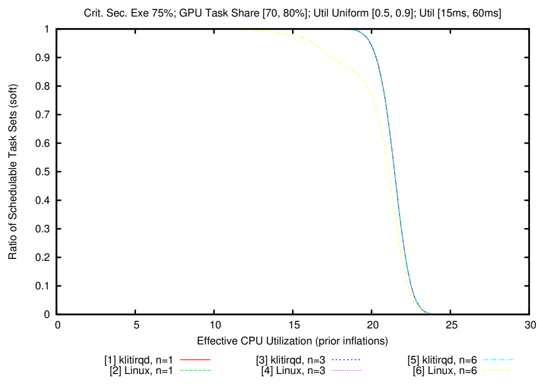


Figure 154. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

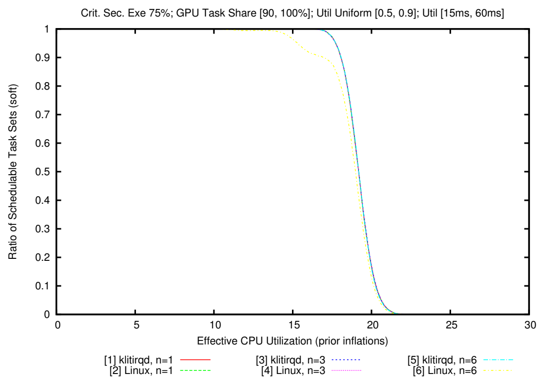


Figure 156. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

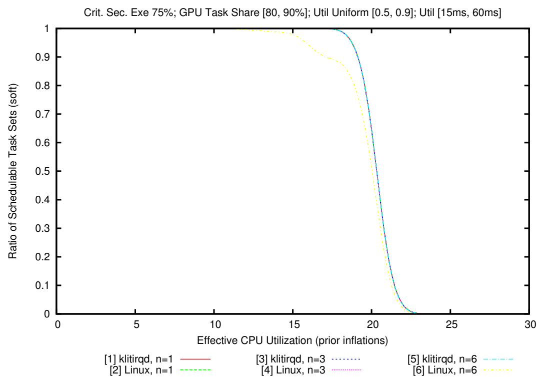


Figure 155. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

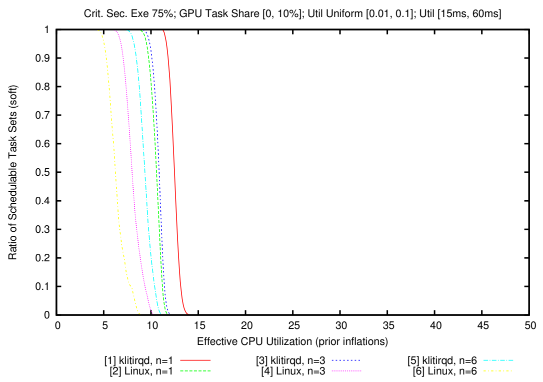


Figure 157. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

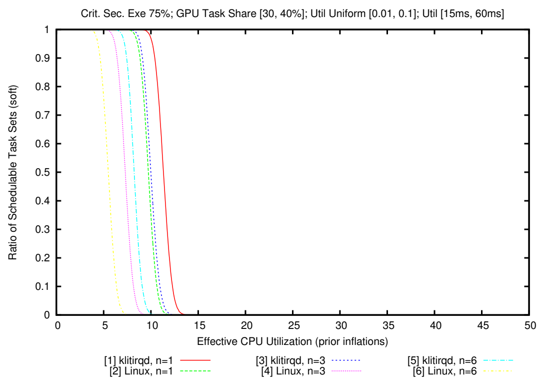


Figure 160. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

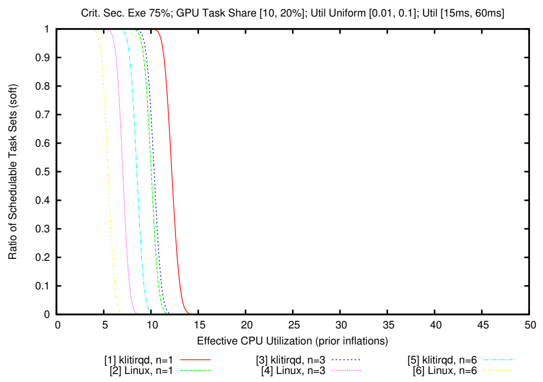


Figure 158. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

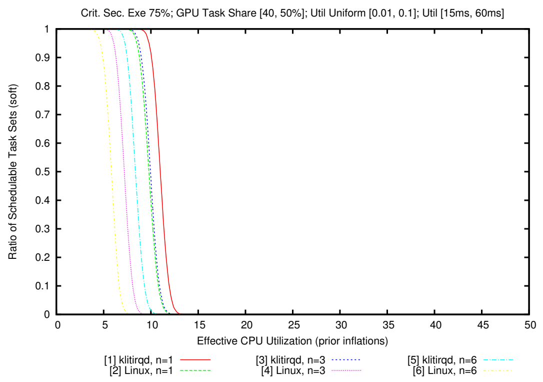


Figure 161. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

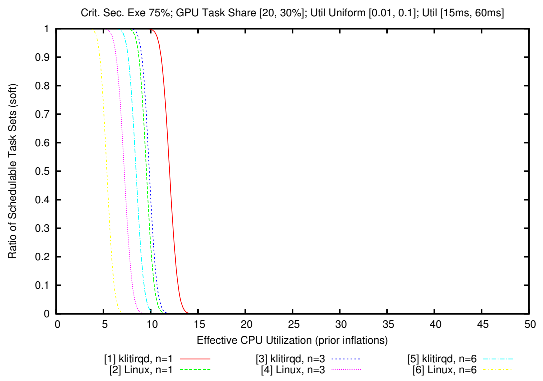


Figure 159. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

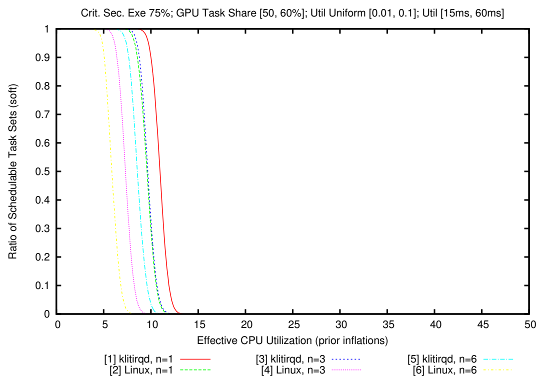


Figure 162. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

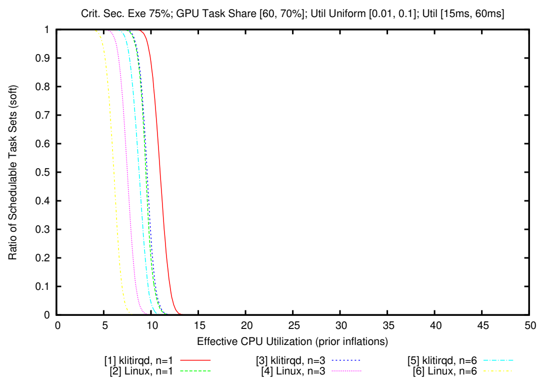


Figure 163. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

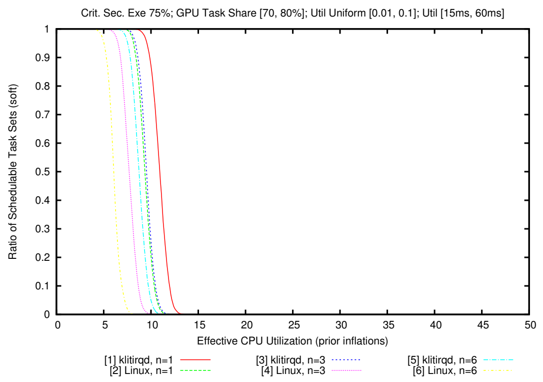


Figure 164. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

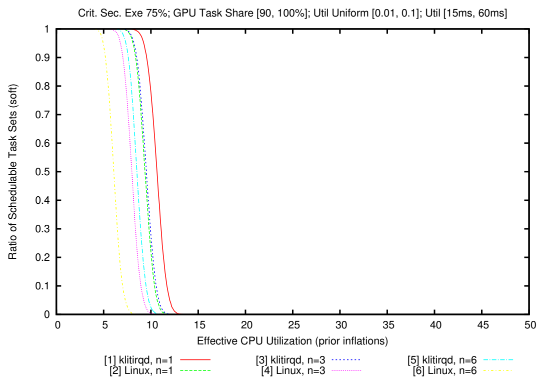


Figure 166. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

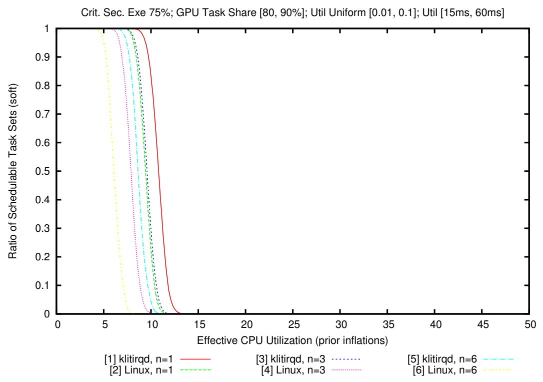


Figure 165. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

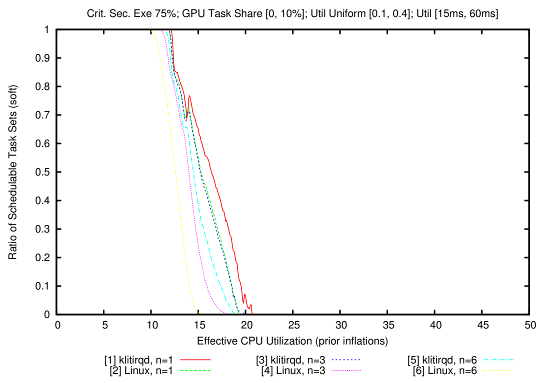


Figure 167. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

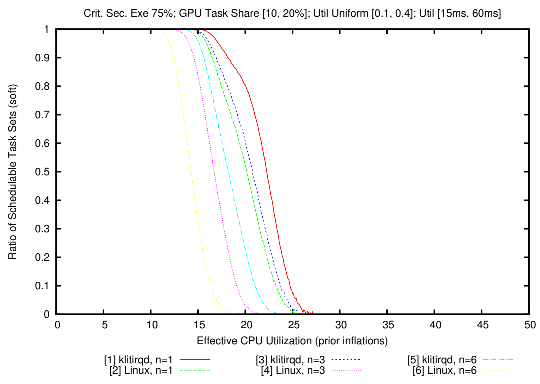


Figure 168. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

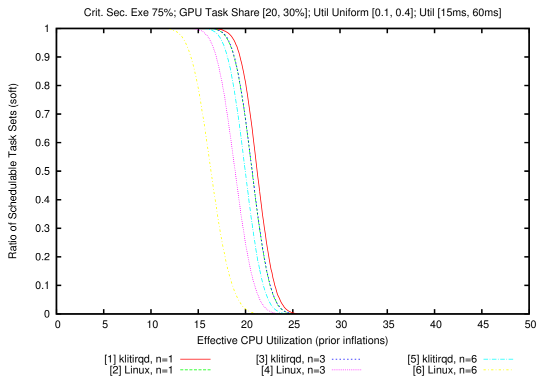


Figure 169. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

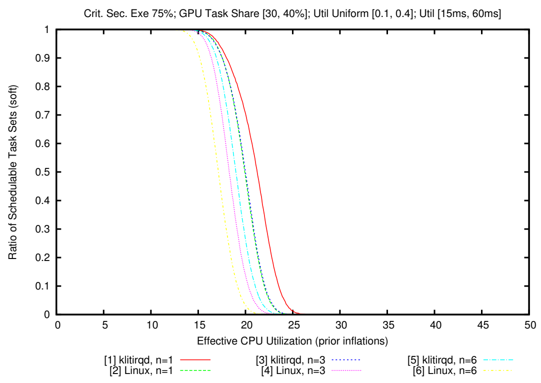


Figure 170. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

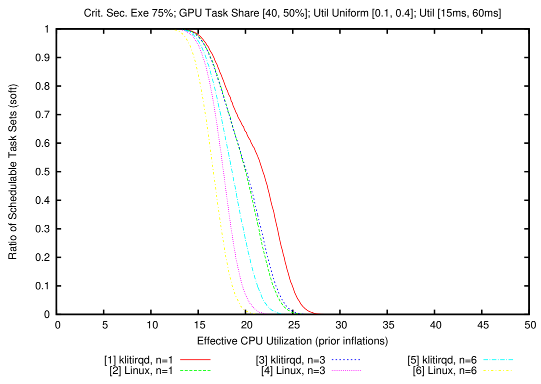


Figure 171. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

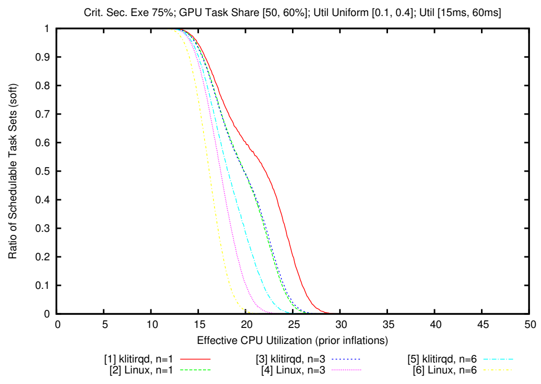


Figure 172. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

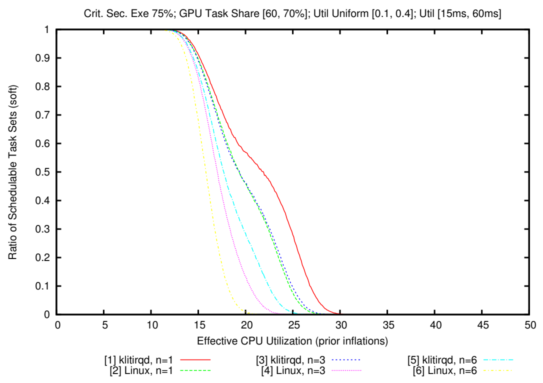


Figure 173. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

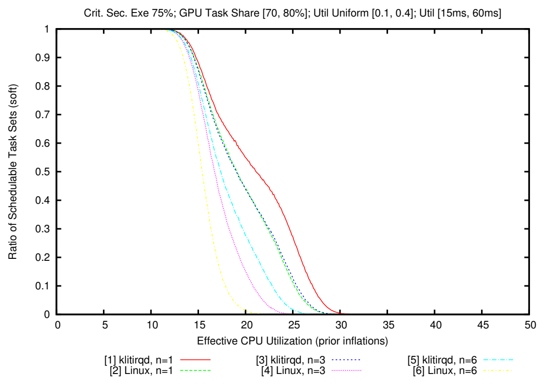


Figure 174. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

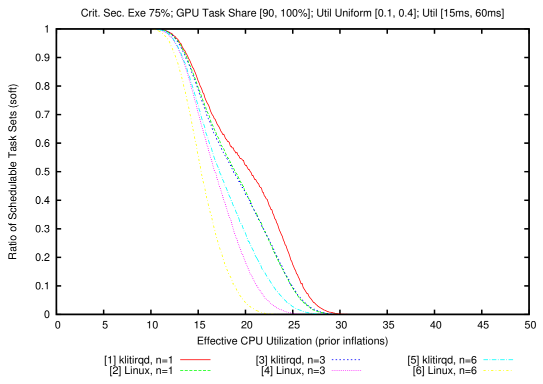


Figure 176. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

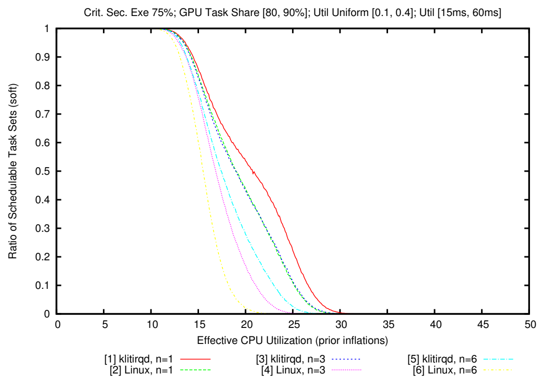


Figure 175. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

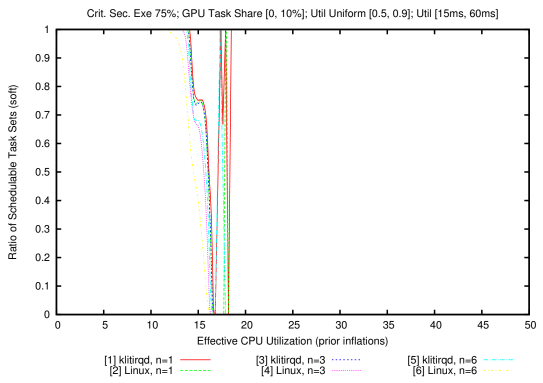


Figure 177. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

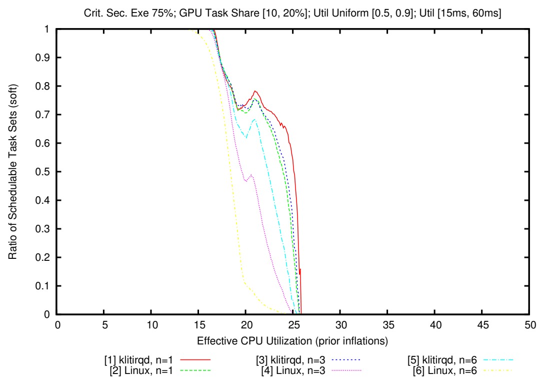


Figure 178. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

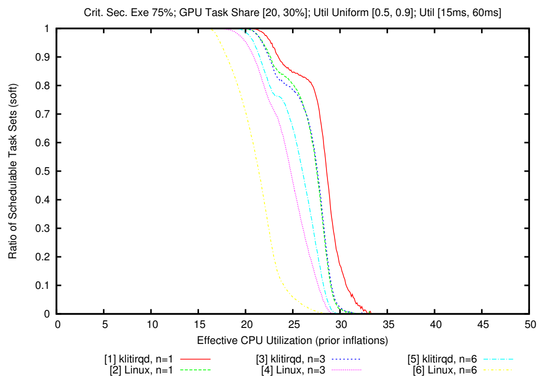


Figure 179. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

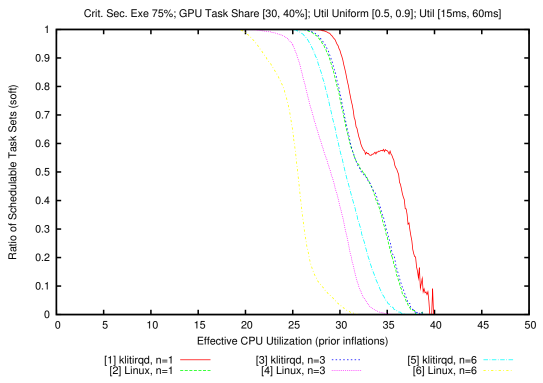


Figure 180. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

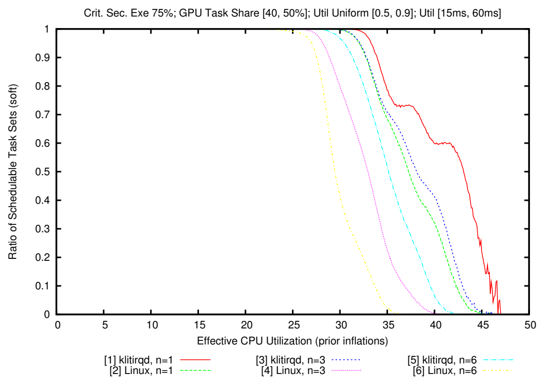


Figure 181. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

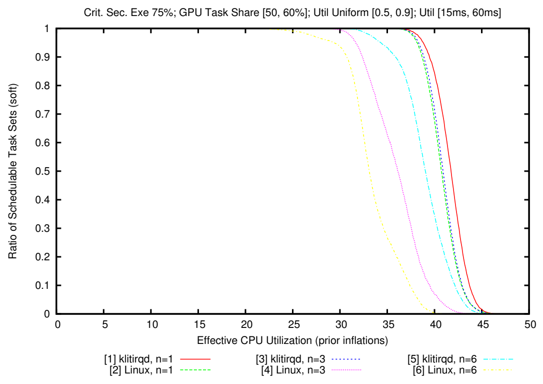


Figure 182. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

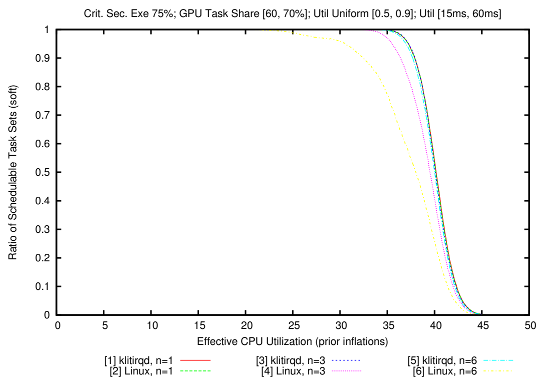


Figure 183. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

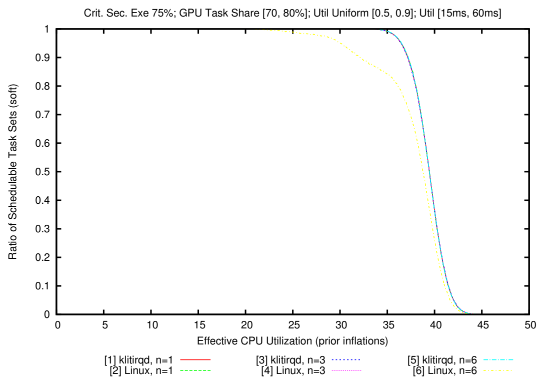


Figure 184. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

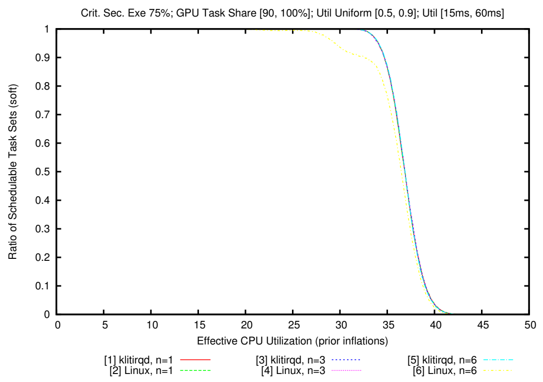


Figure 186. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

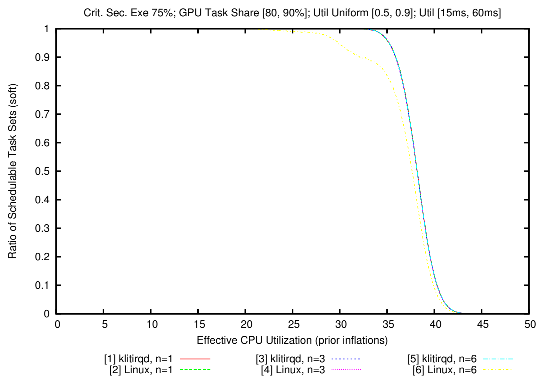


Figure 185. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

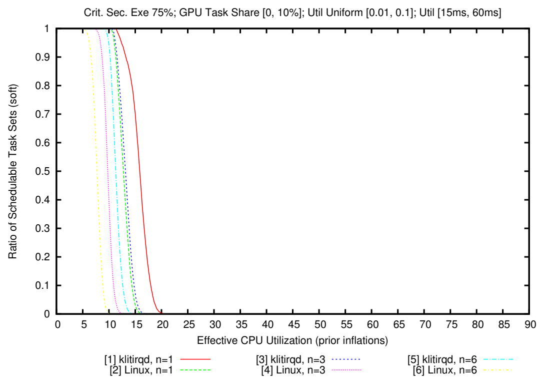


Figure 187. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

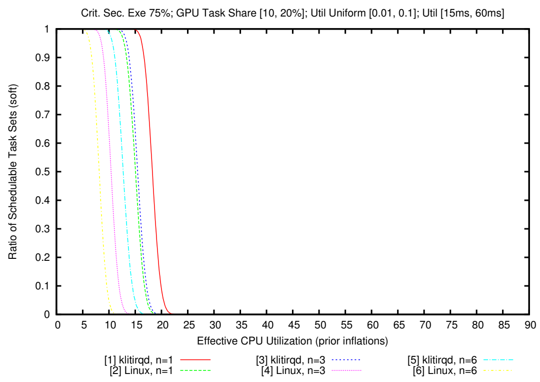


Figure 188. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

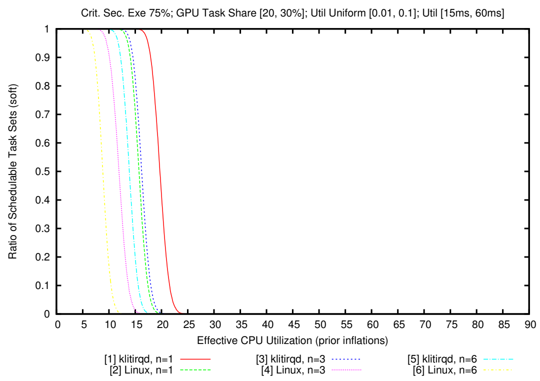


Figure 189. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

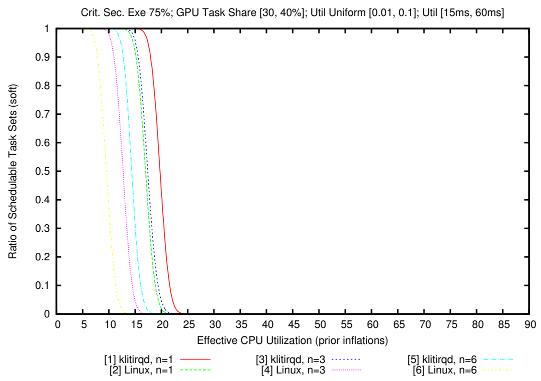


Figure 190. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

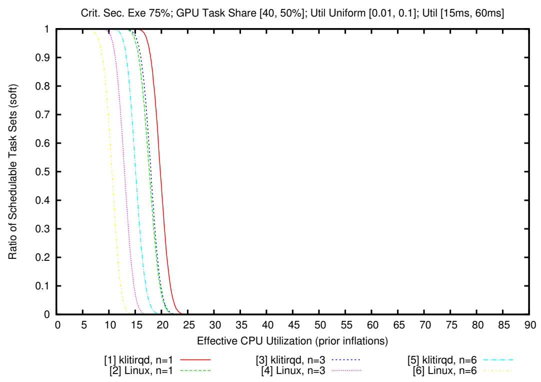


Figure 191. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

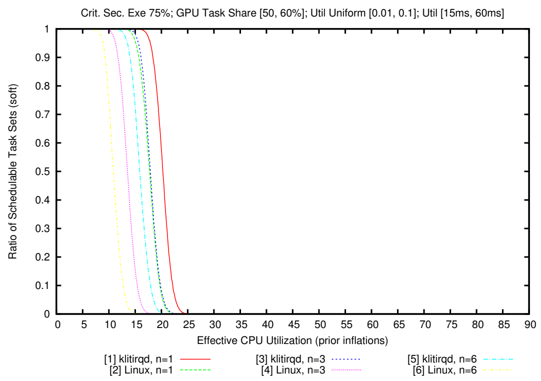


Figure 192. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

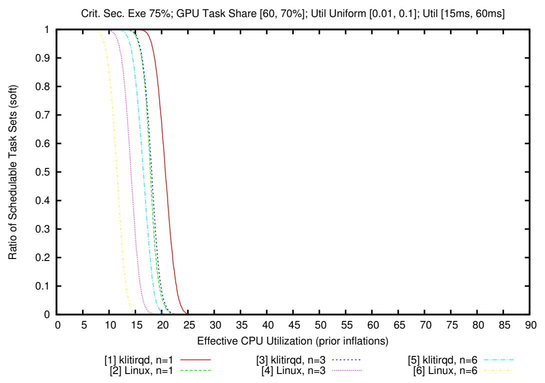


Figure 193. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

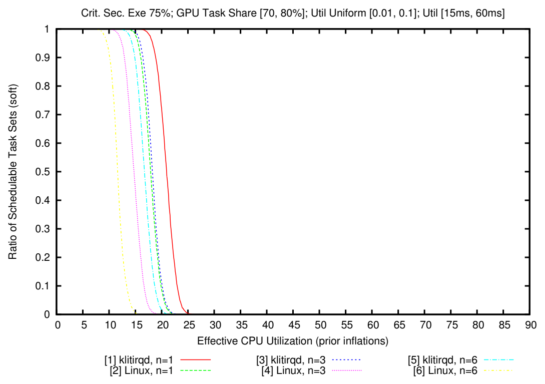


Figure 194. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

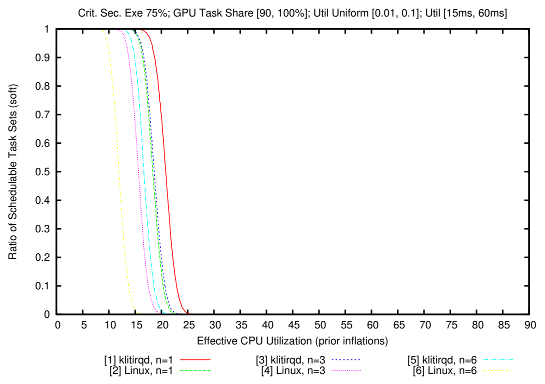


Figure 196. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

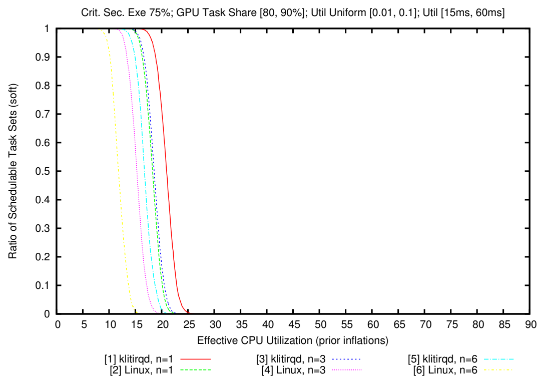


Figure 195. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [1%, 10%].

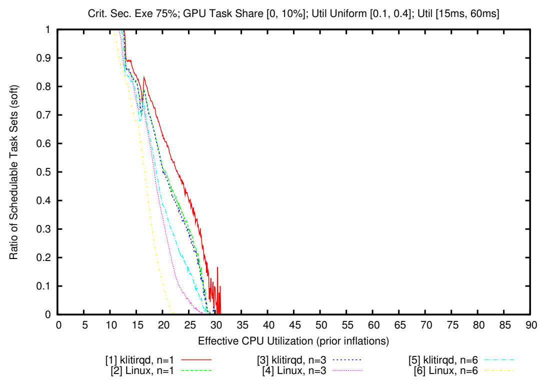


Figure 197. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

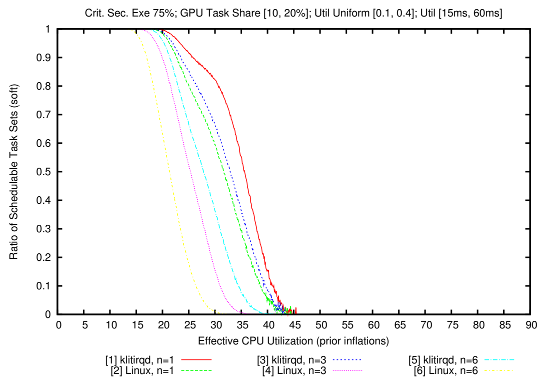


Figure 198. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

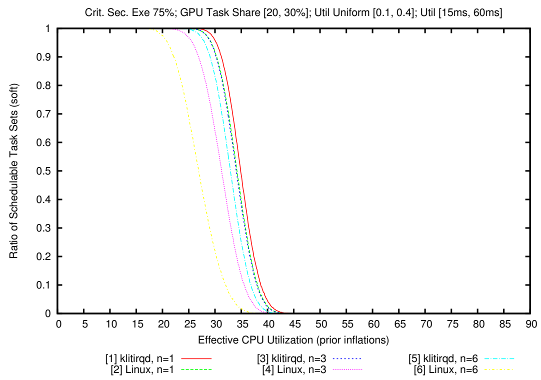


Figure 199. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

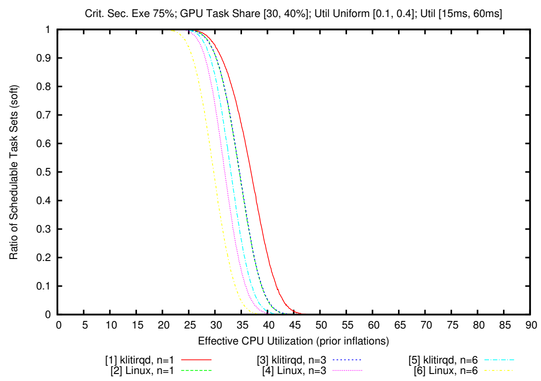


Figure 200. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

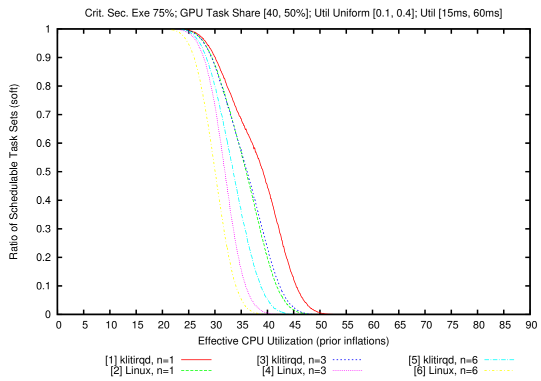


Figure 201. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

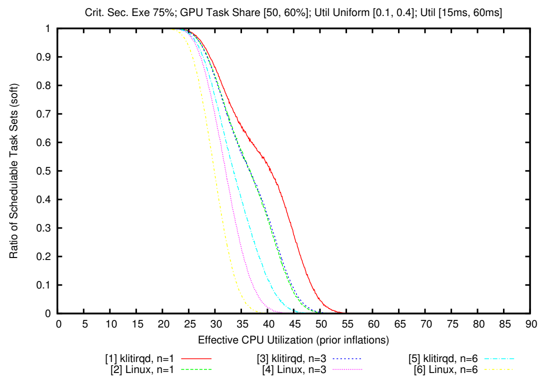


Figure 202. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

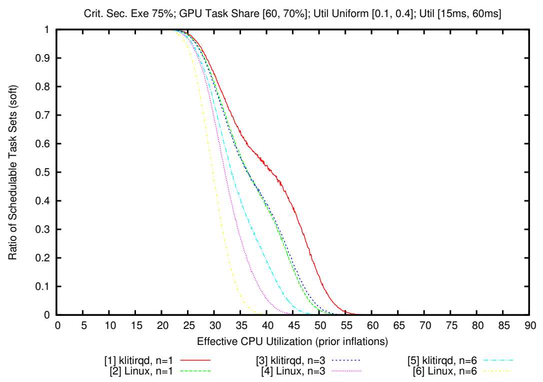


Figure 203. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

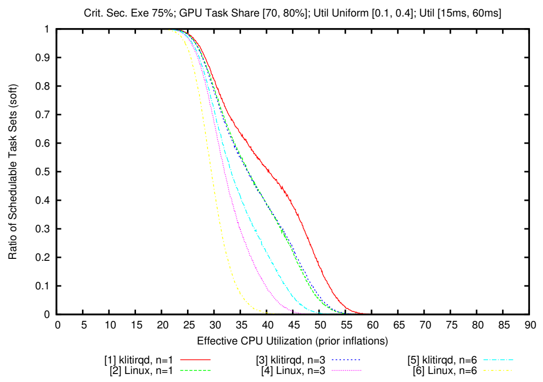


Figure 204. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

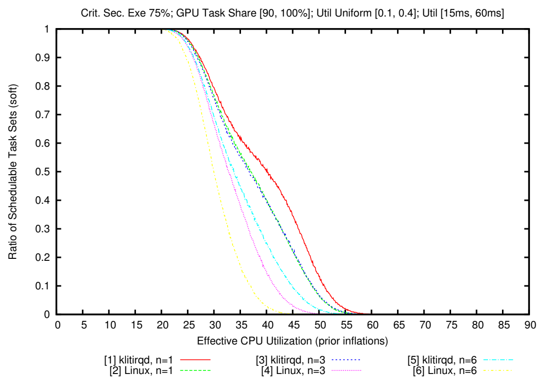


Figure 206. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

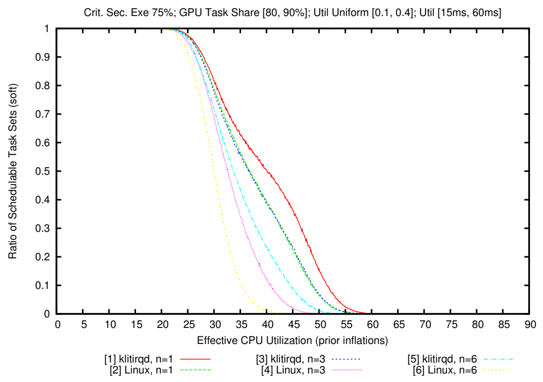


Figure 205. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [10%, 40%].

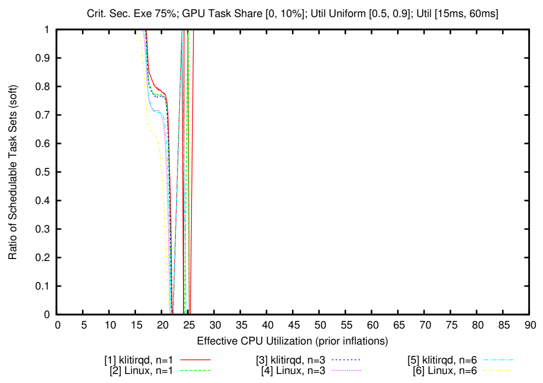


Figure 207. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

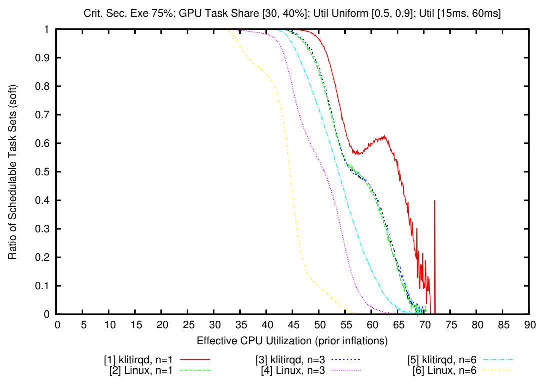


Figure 210. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

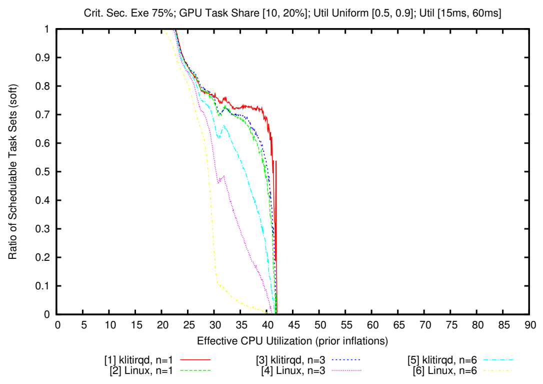


Figure 208. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

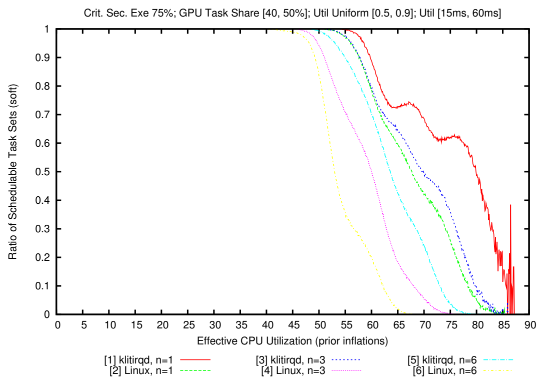


Figure 211. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

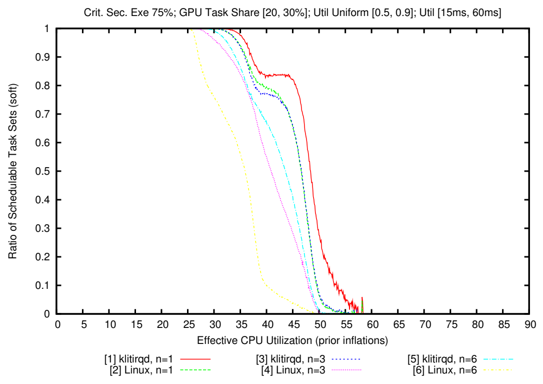


Figure 209. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

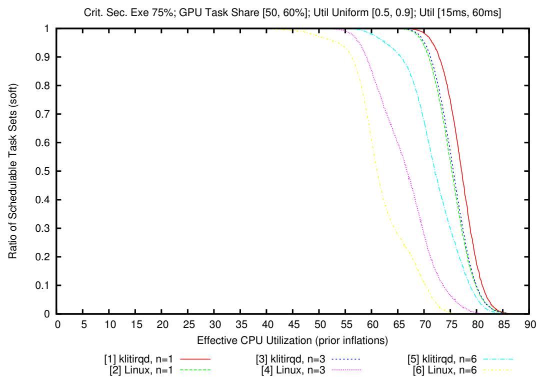


Figure 212. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

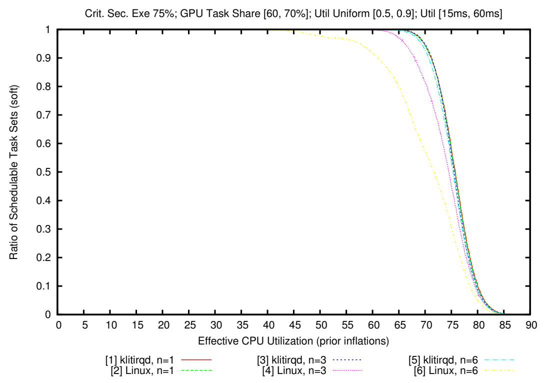


Figure 213. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

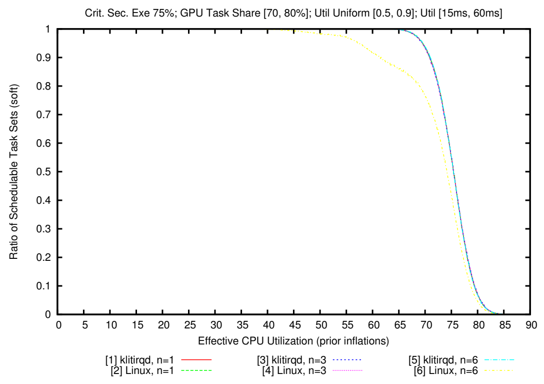


Figure 214. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

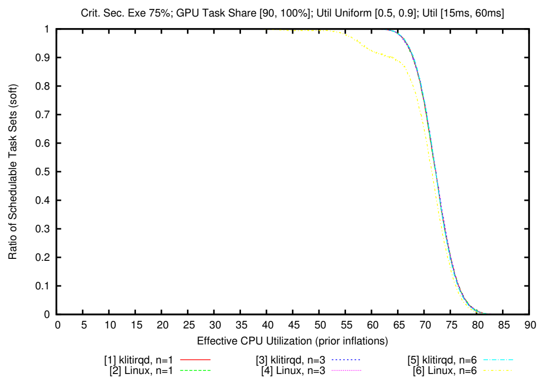


Figure 216. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

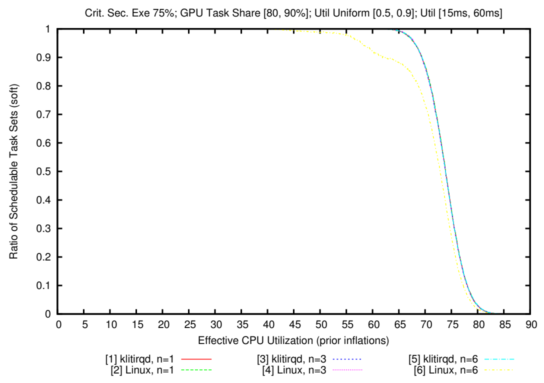


Figure 215. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected uniformly from the range [50%, 90%].

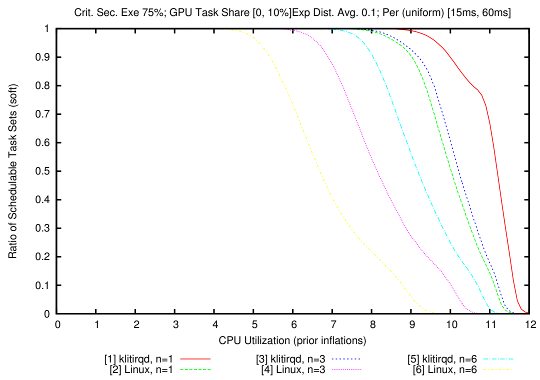


Figure 217. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

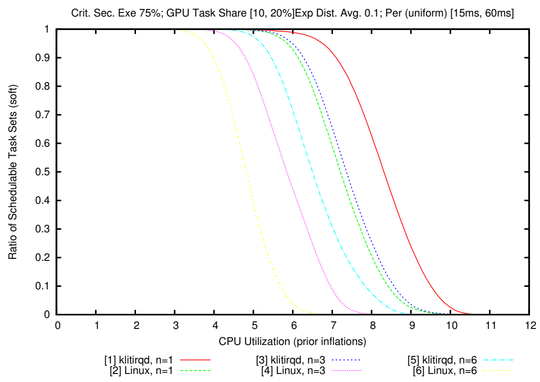


Figure 218. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

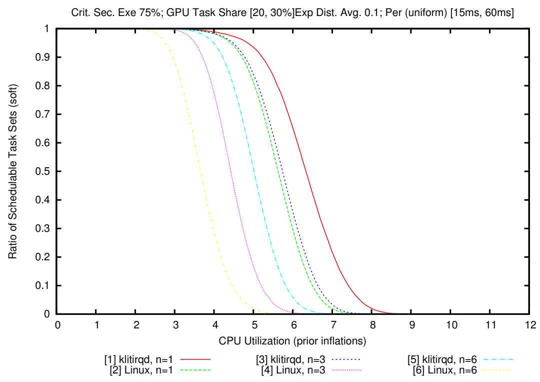


Figure 219. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

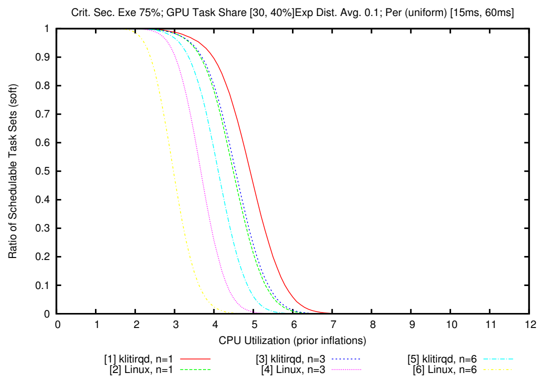


Figure 220. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

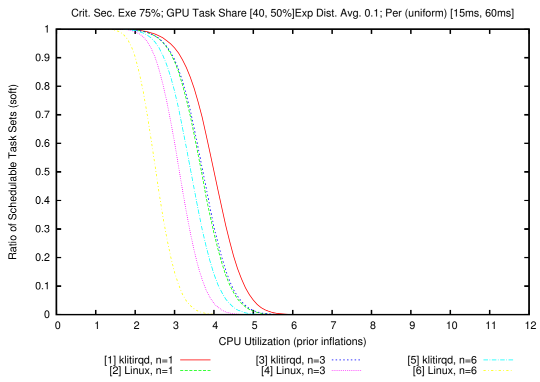


Figure 221. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

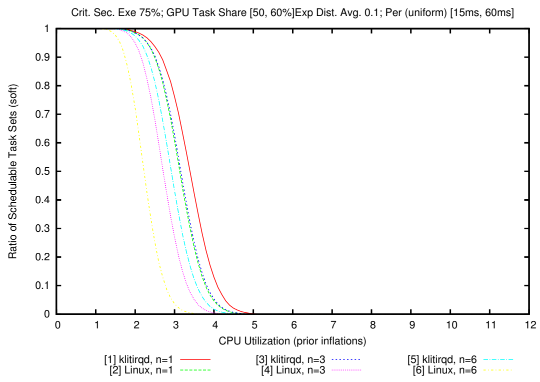


Figure 222. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

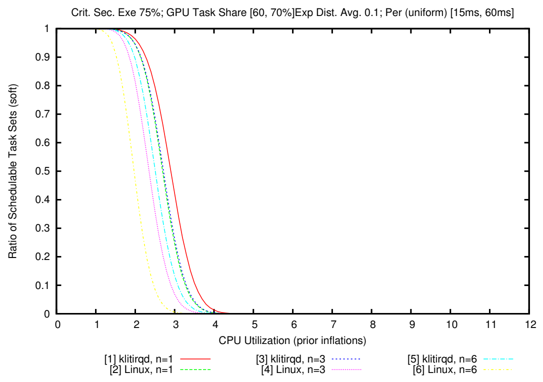


Figure 223. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

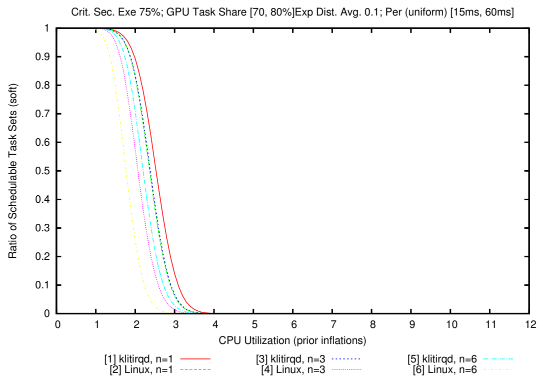


Figure 224. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

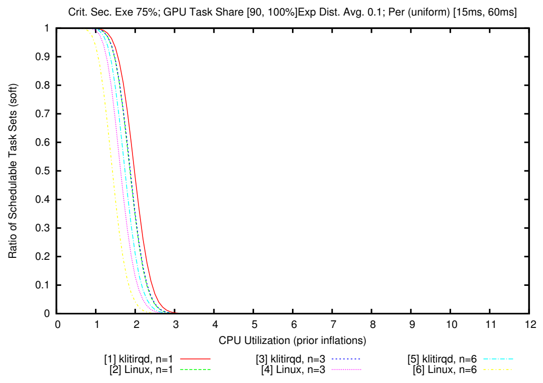


Figure 226. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

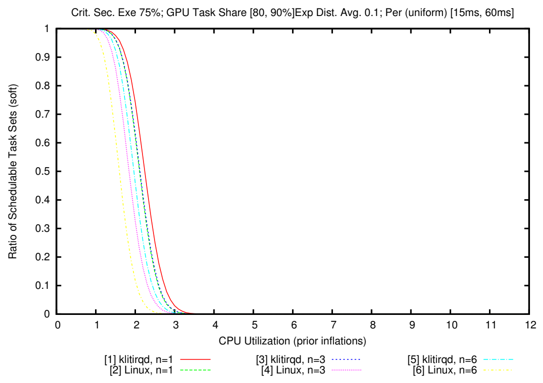


Figure 225. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

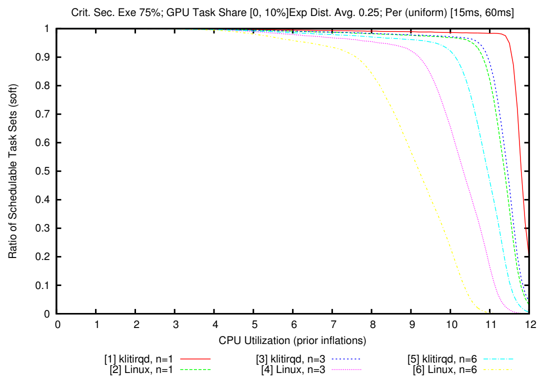


Figure 227. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

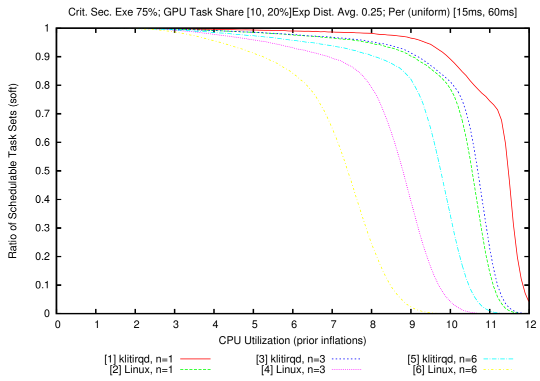


Figure 228. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

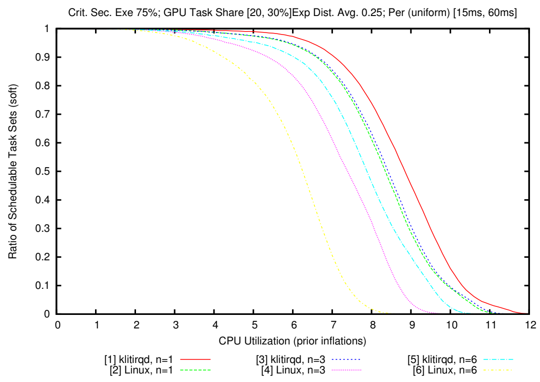


Figure 229. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

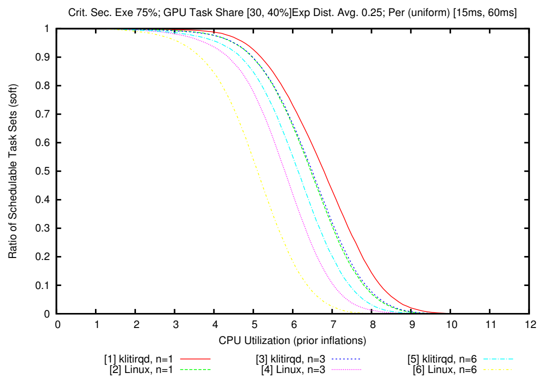


Figure 230. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

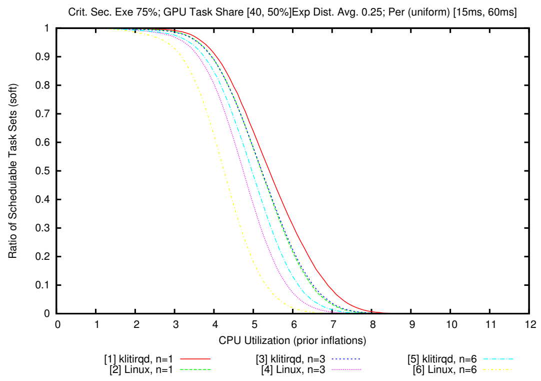


Figure 231. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

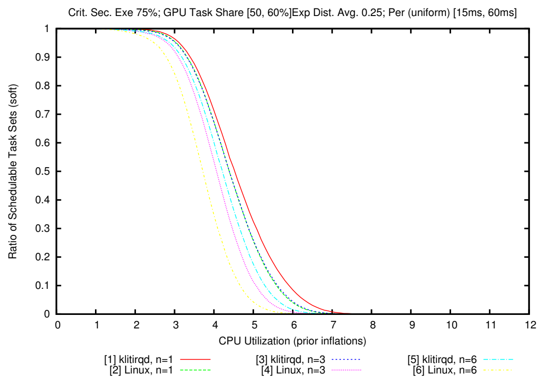


Figure 232. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

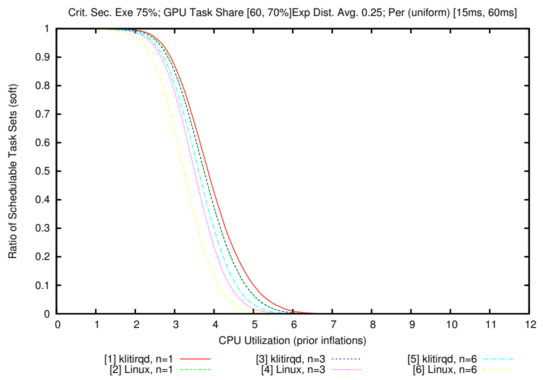


Figure 233. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

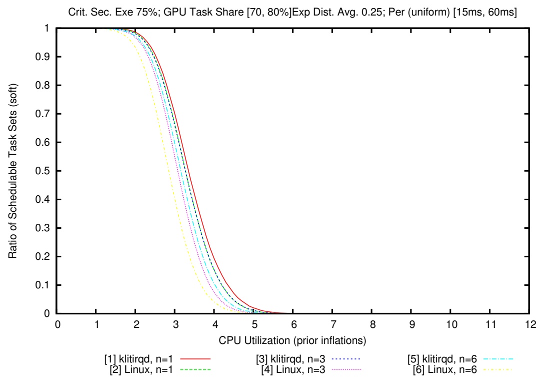


Figure 234. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

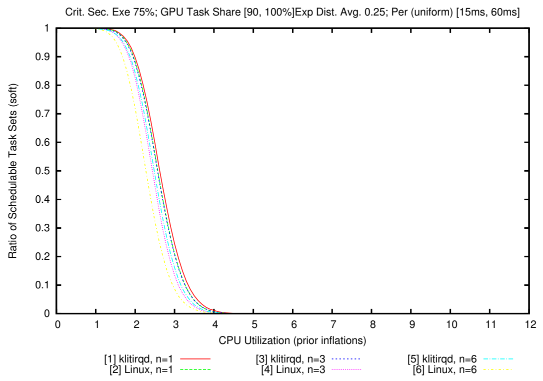


Figure 236. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

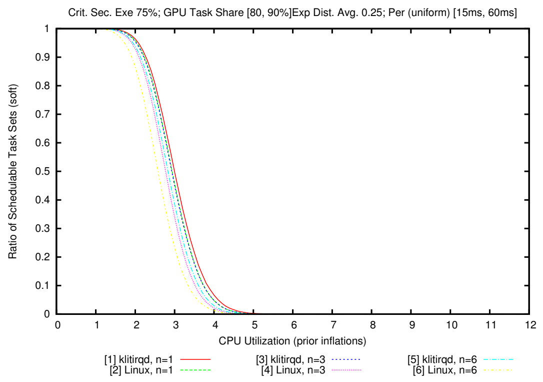


Figure 235. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

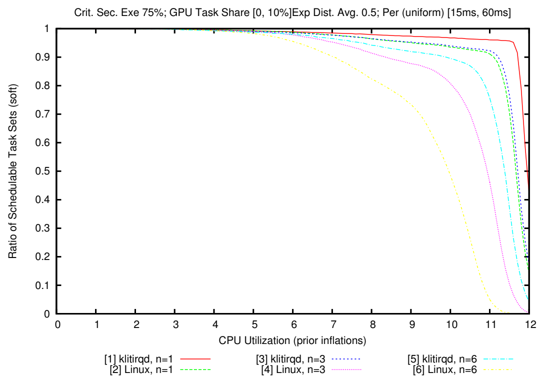


Figure 237. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

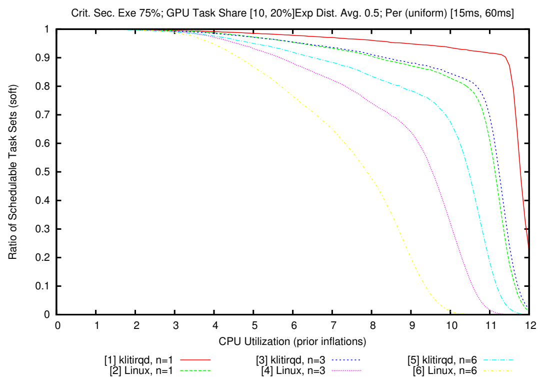


Figure 238. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

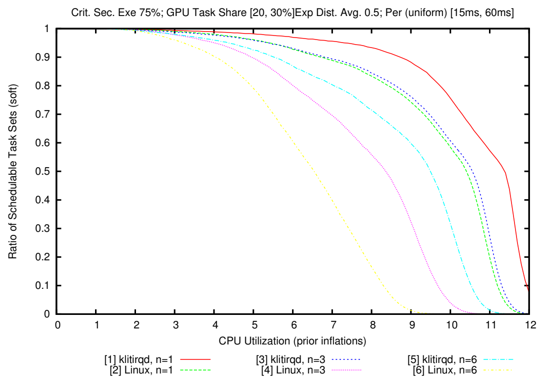


Figure 239. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

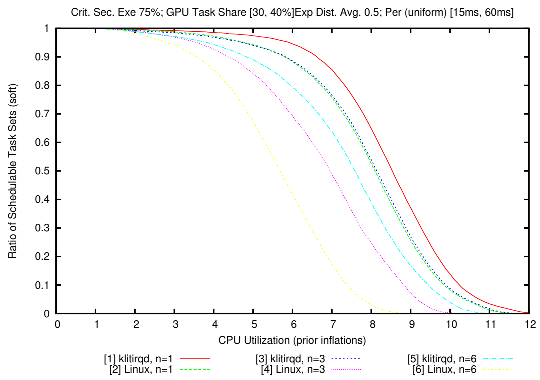


Figure 240. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

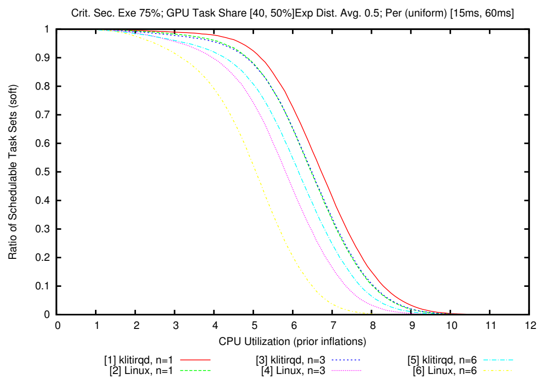


Figure 241. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

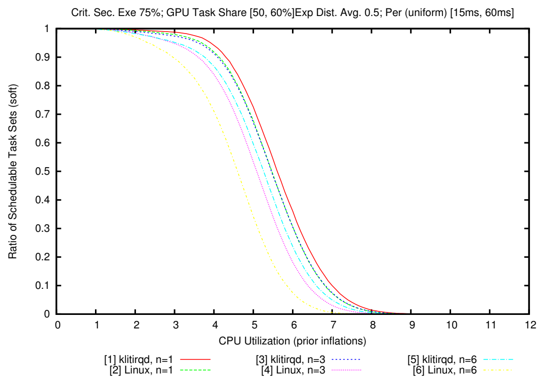


Figure 242. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

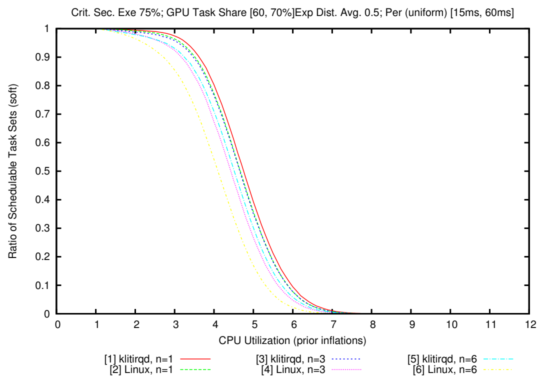


Figure 243. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

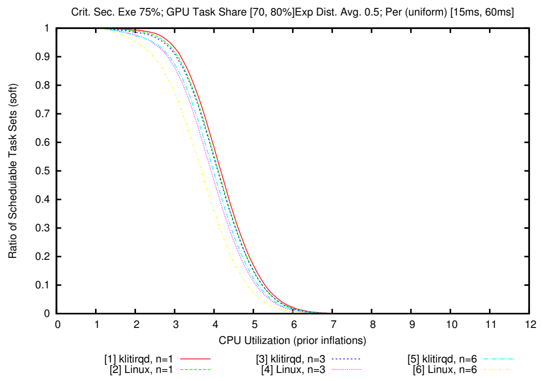


Figure 244. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

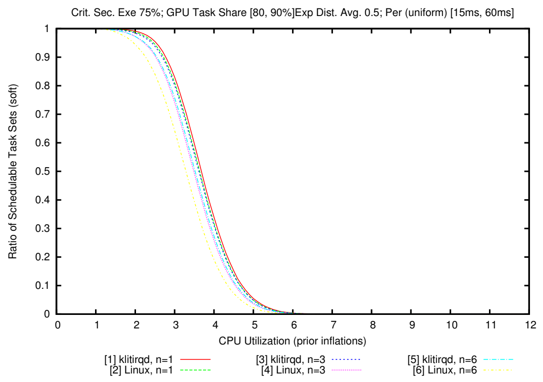


Figure 245. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

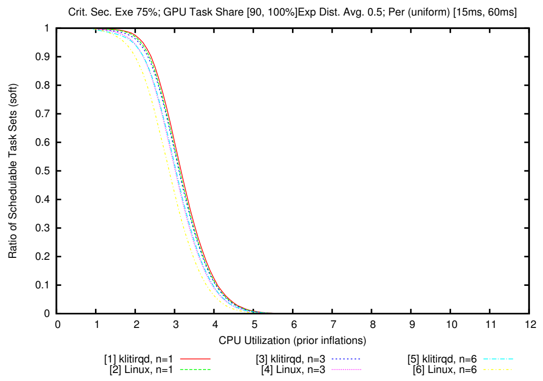


Figure 246. The percentage of schedulable task sets as a function of CPU utilization. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

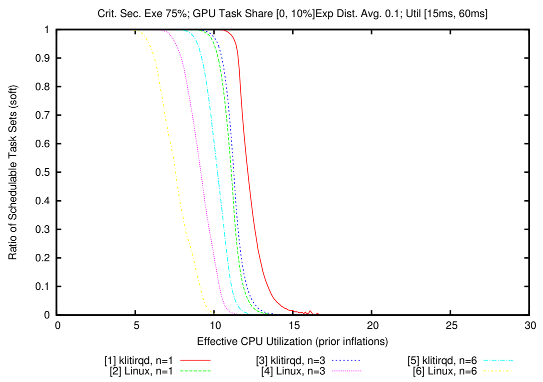


Figure 247. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

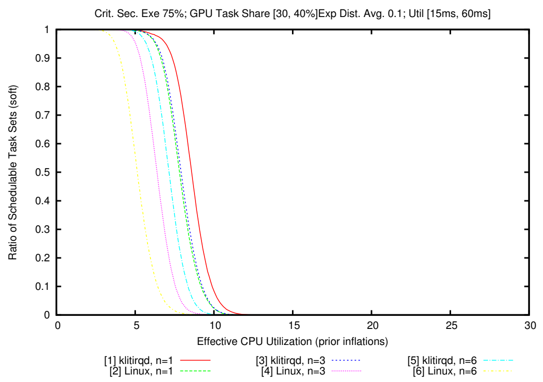


Figure 250. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

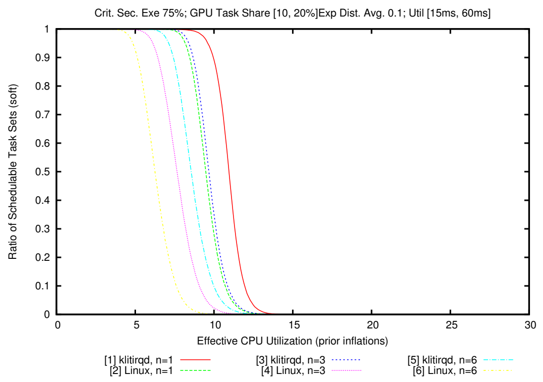


Figure 248. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

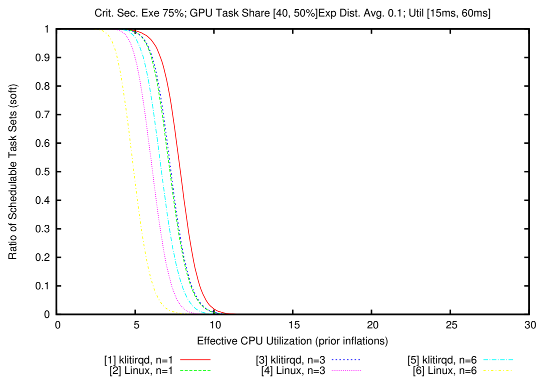


Figure 251. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

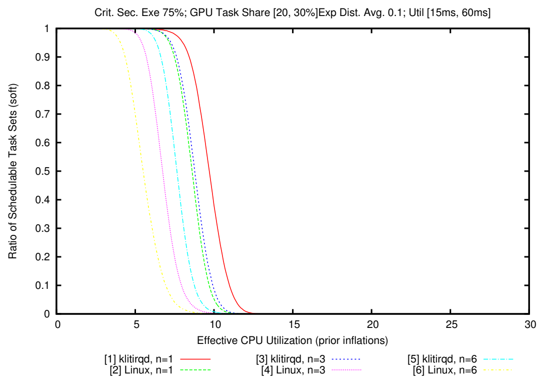


Figure 249. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

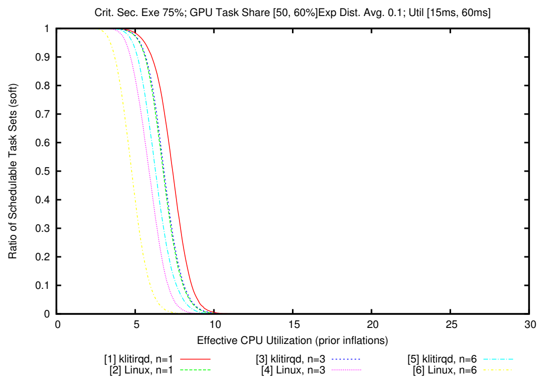


Figure 252. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

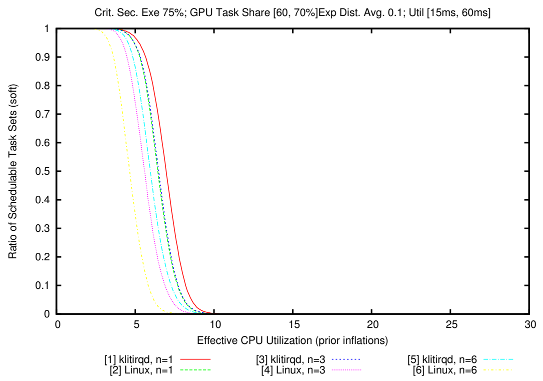


Figure 253. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

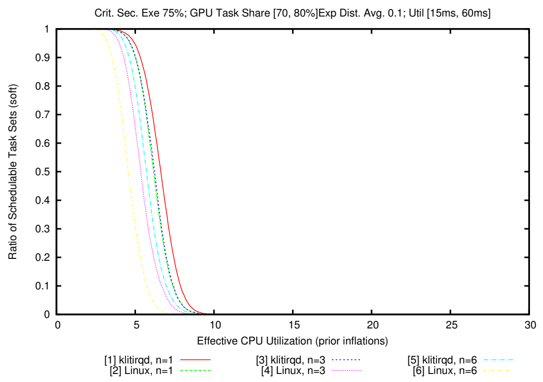


Figure 254. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

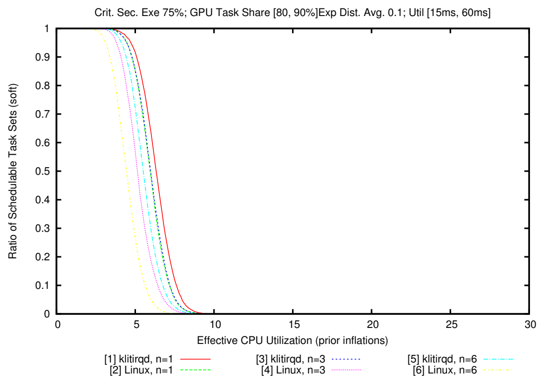


Figure 255. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

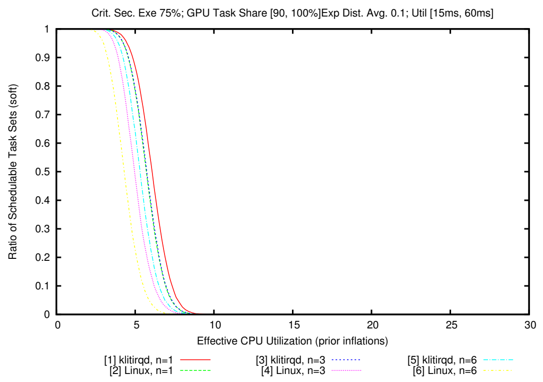


Figure 256. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

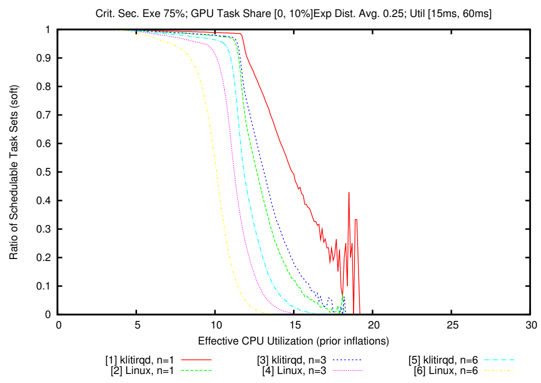


Figure 257. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

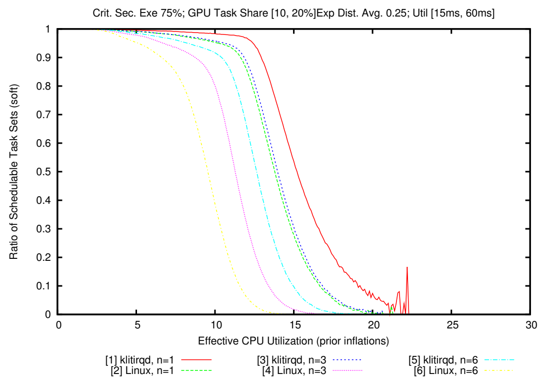


Figure 258. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

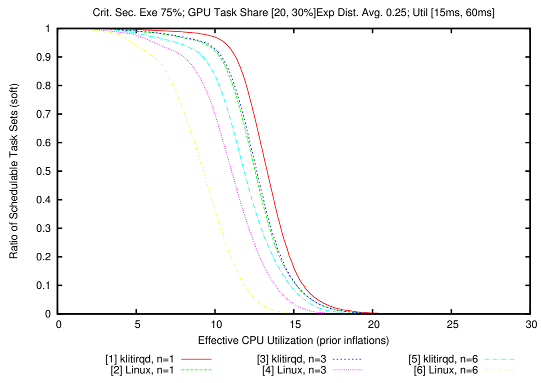


Figure 259. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

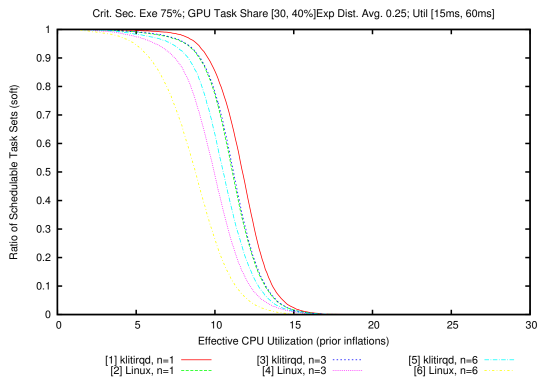


Figure 260. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

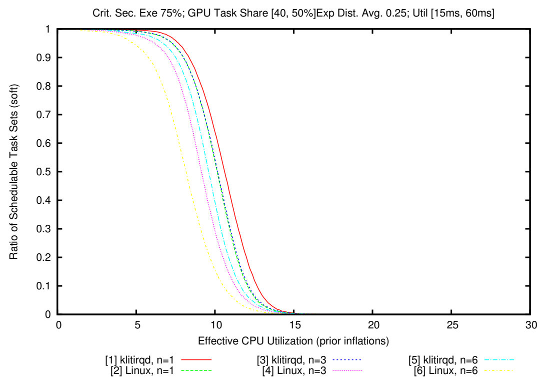


Figure 261. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

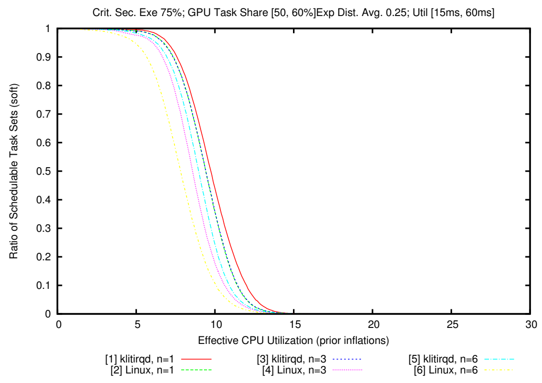


Figure 262. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

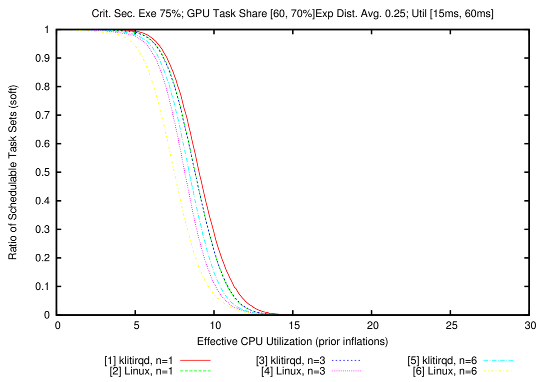


Figure 263. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

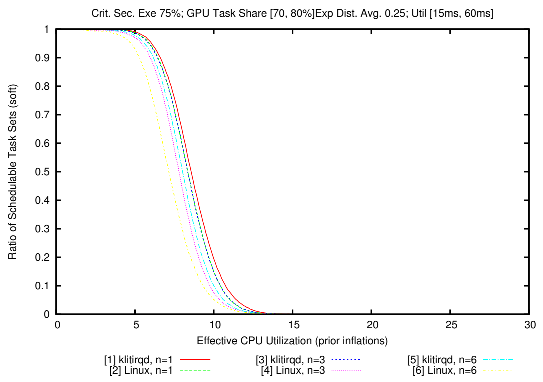


Figure 264. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

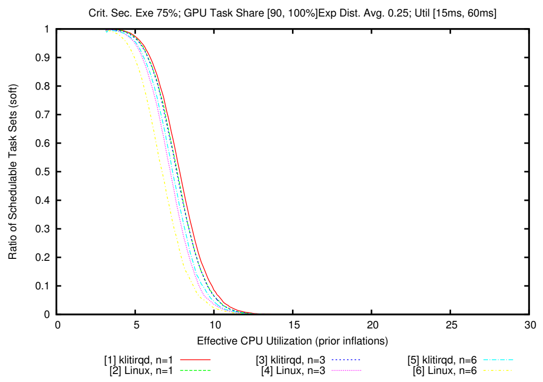


Figure 266. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

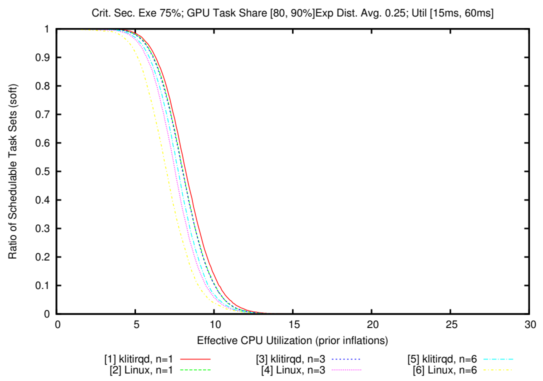


Figure 265. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

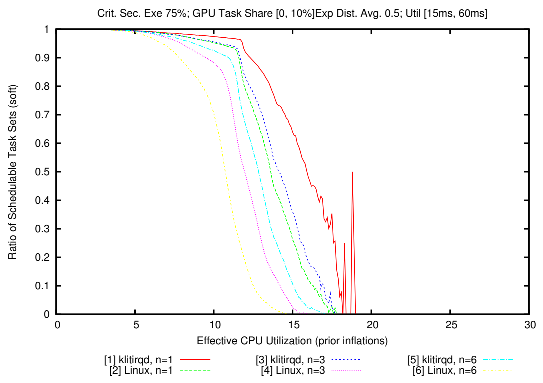


Figure 267. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

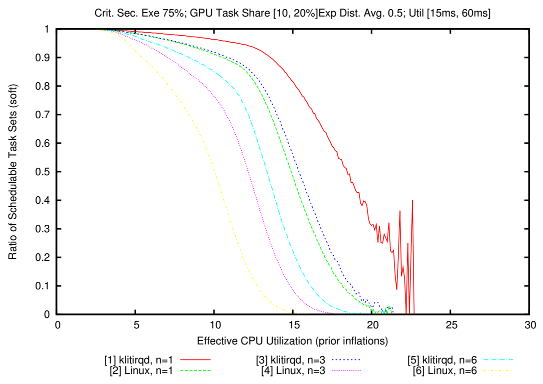


Figure 268. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

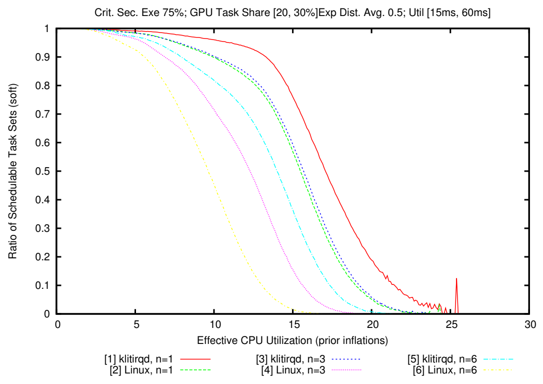


Figure 269. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

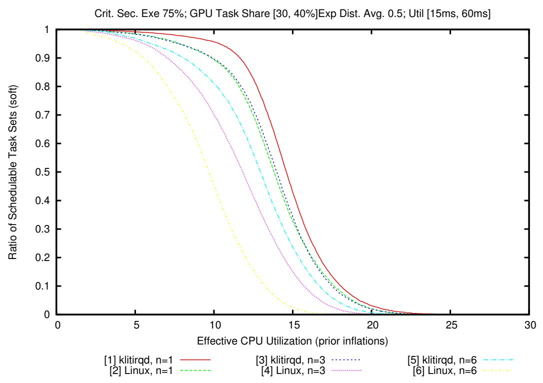


Figure 270. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

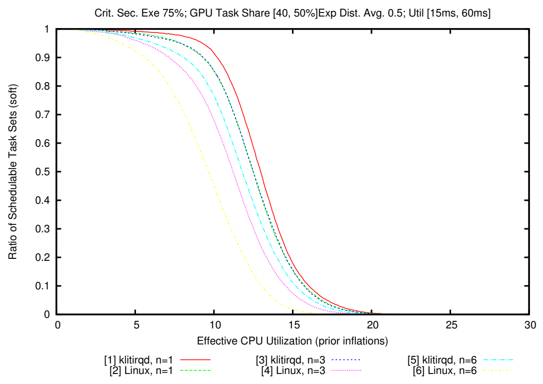


Figure 271. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

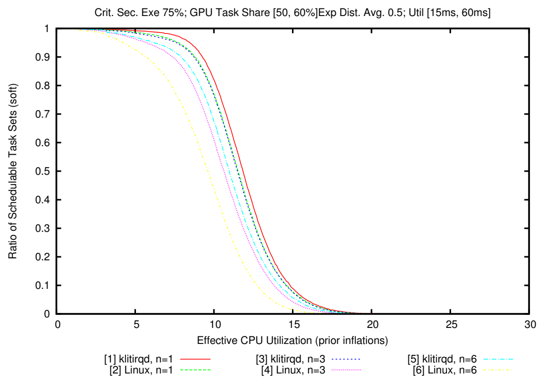


Figure 272. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

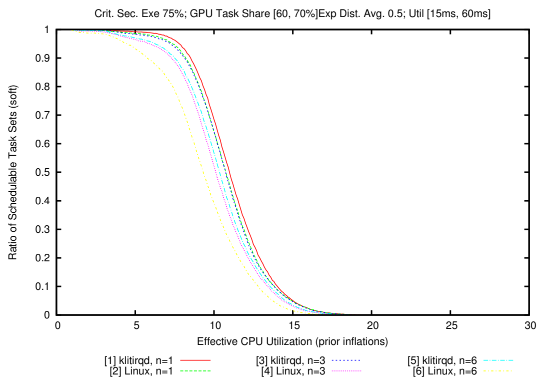


Figure 273. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

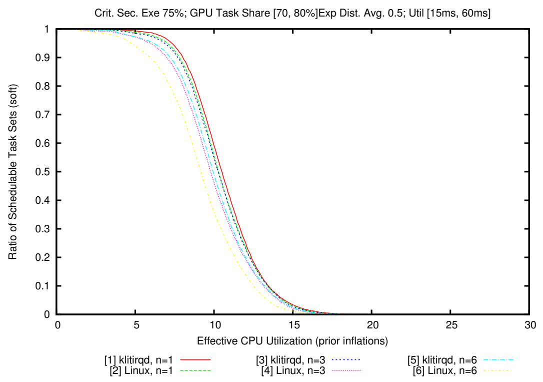


Figure 274. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

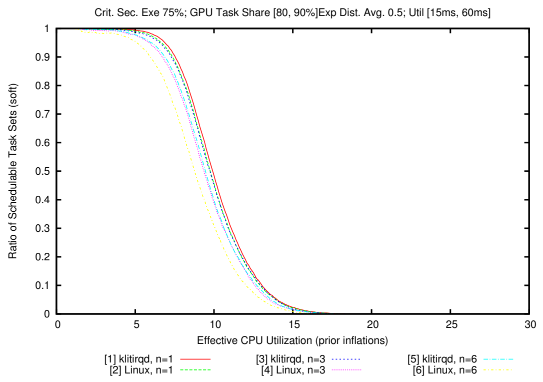


Figure 275. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

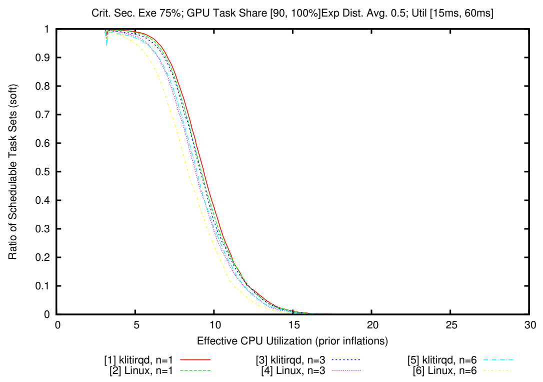


Figure 276. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 4×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

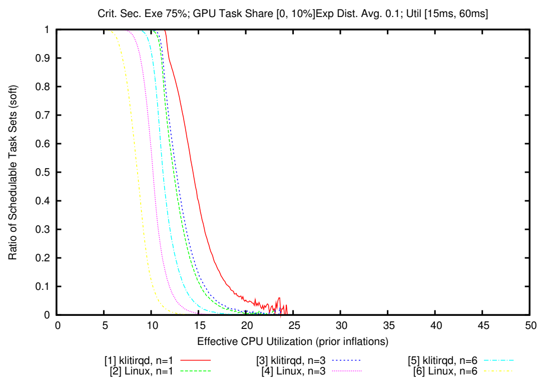


Figure 277. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

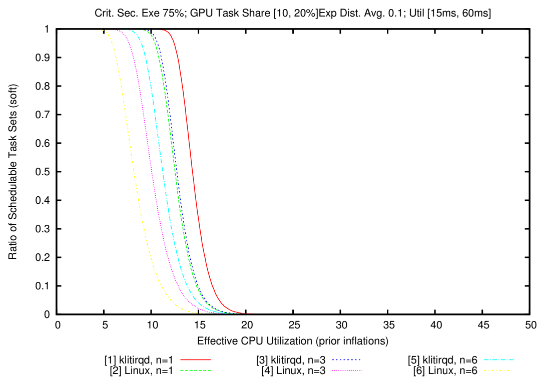


Figure 278. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

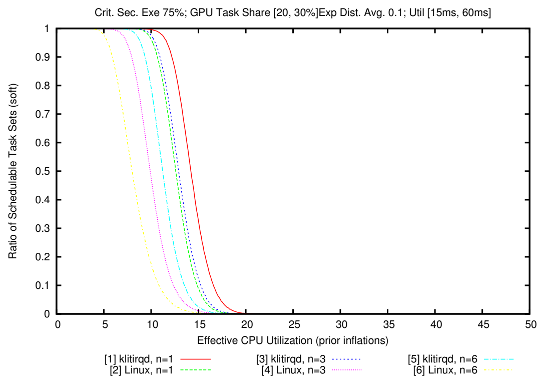


Figure 279. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

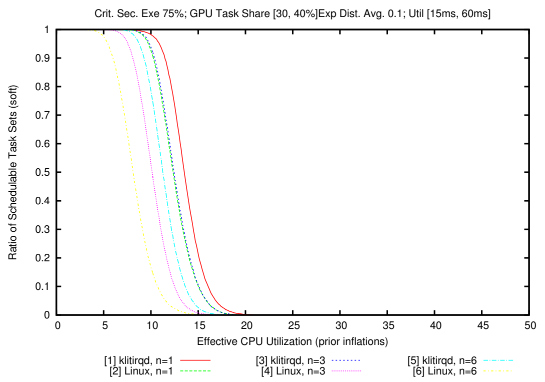


Figure 280. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

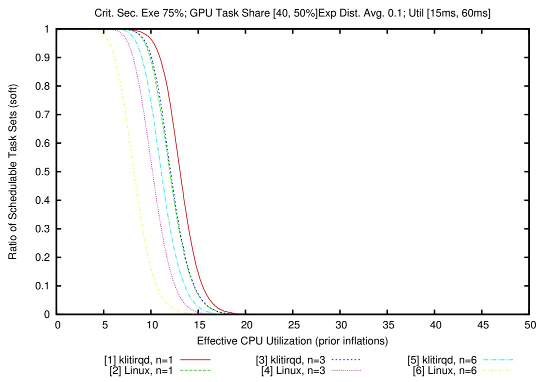


Figure 281. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

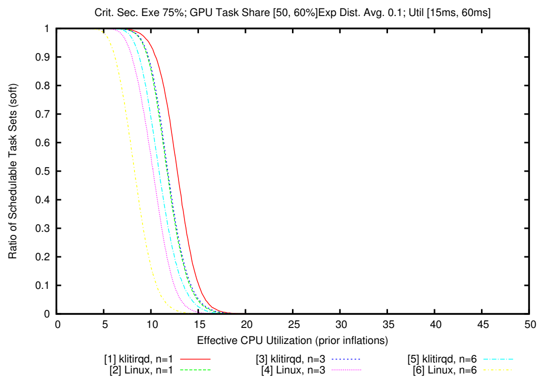


Figure 282. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

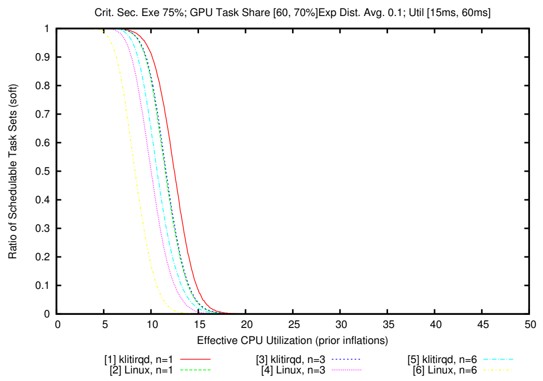


Figure 283. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

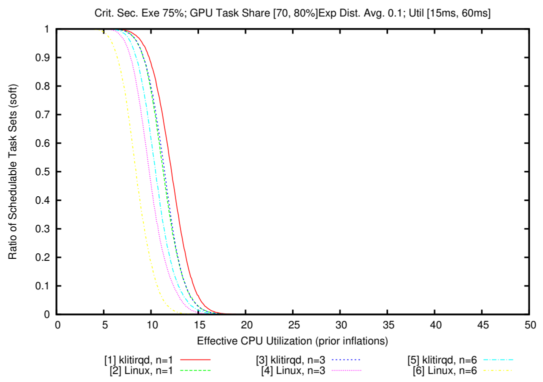


Figure 284. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

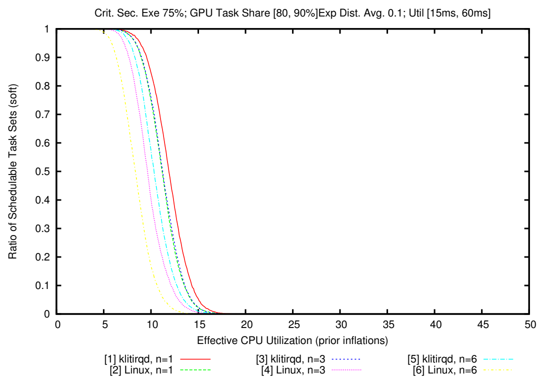


Figure 285. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

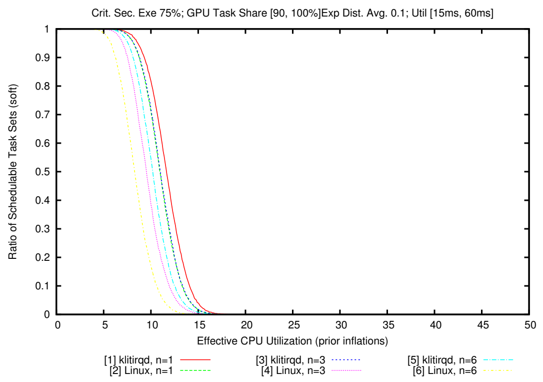


Figure 286. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

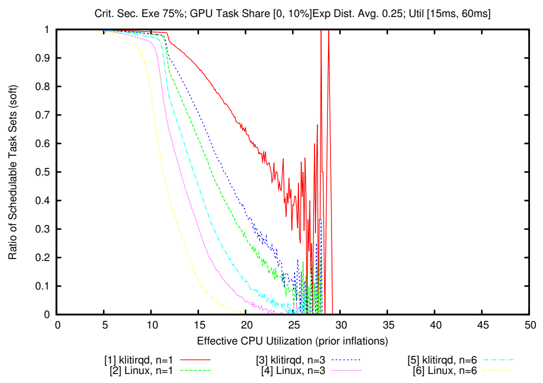


Figure 287. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

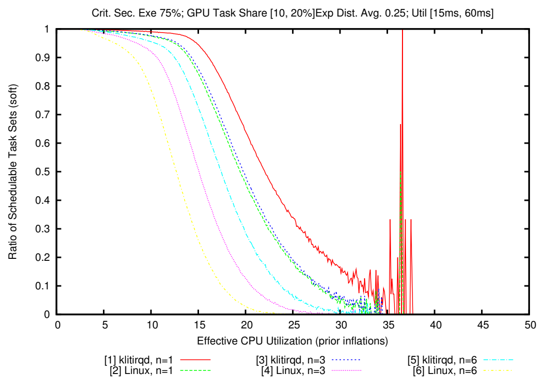


Figure 288. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

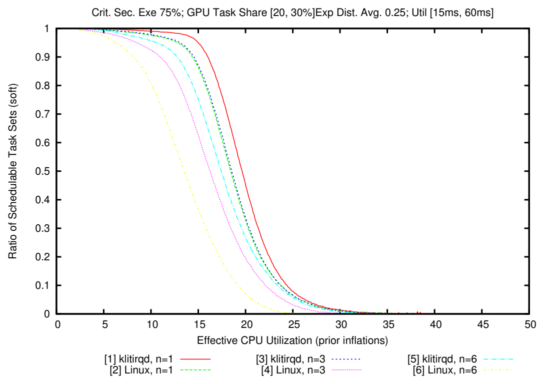


Figure 289. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

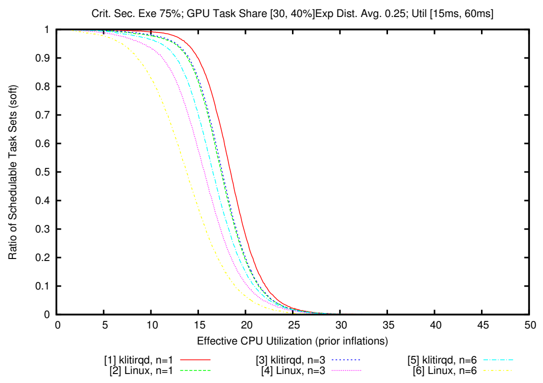


Figure 290. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

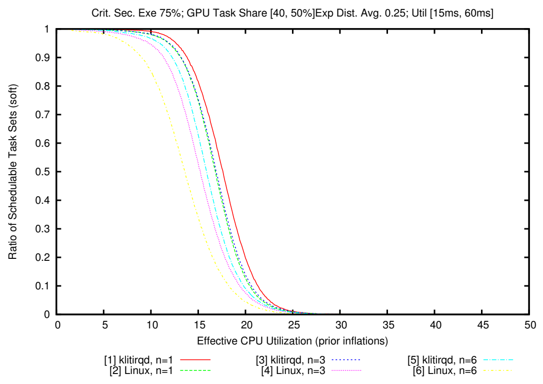


Figure 291. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

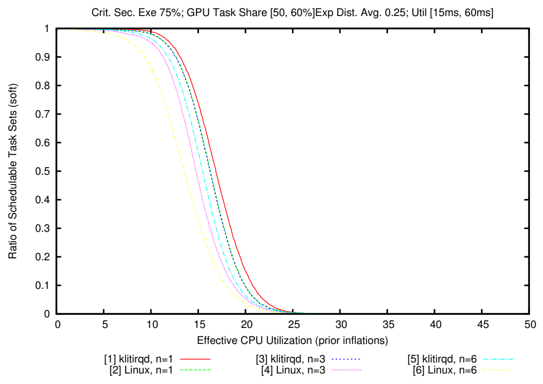


Figure 292. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

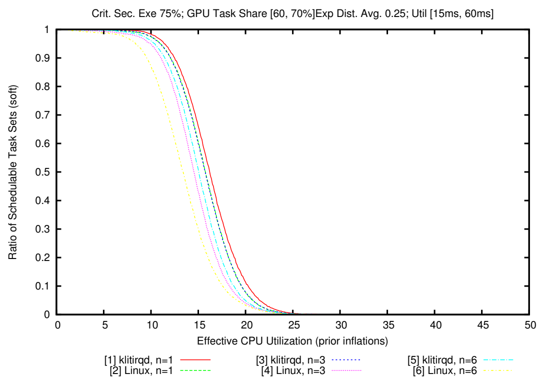


Figure 293. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

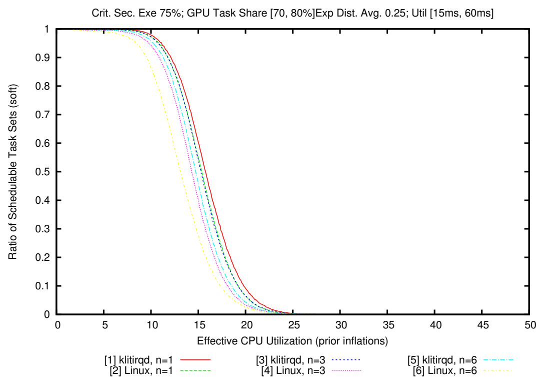


Figure 294. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

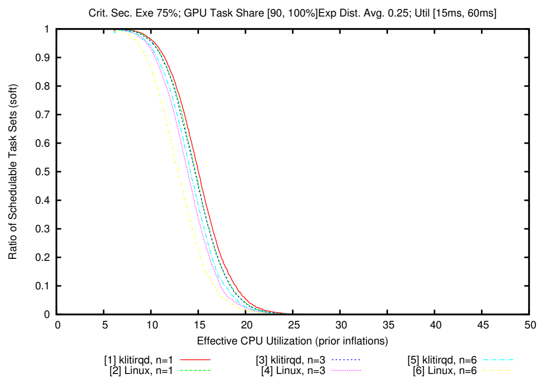


Figure 296. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

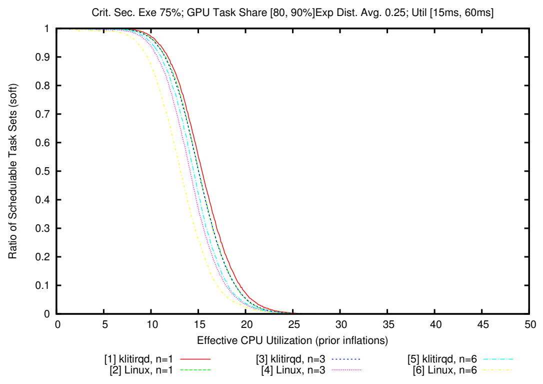


Figure 295. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

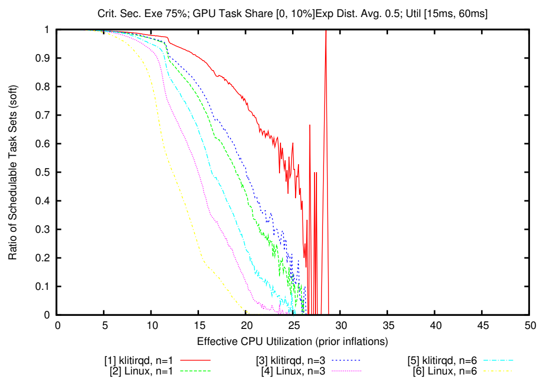


Figure 297. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

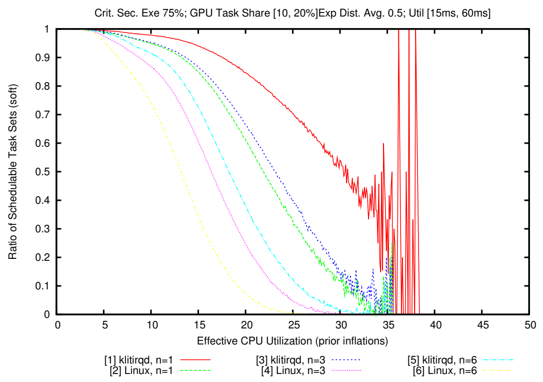


Figure 298. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

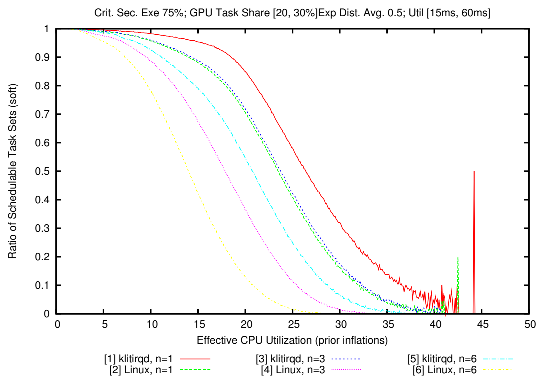


Figure 299. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

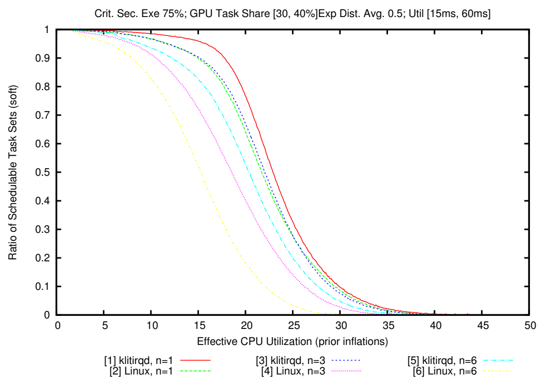


Figure 300. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

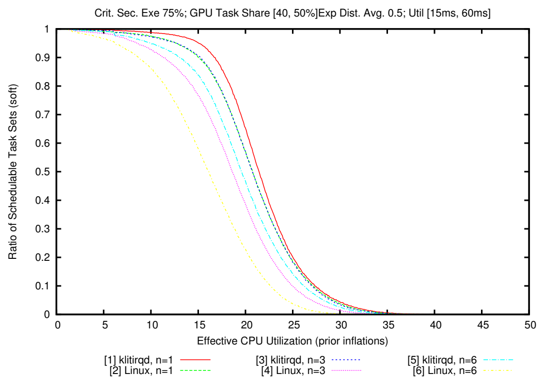


Figure 301. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

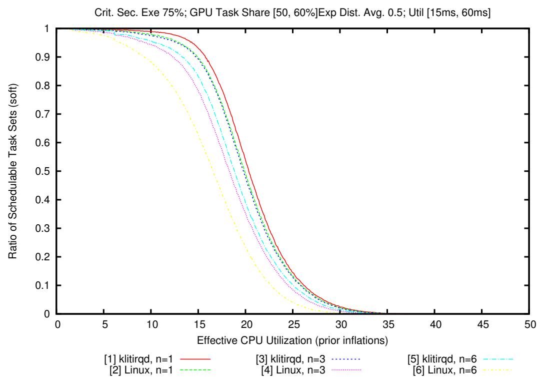


Figure 302. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

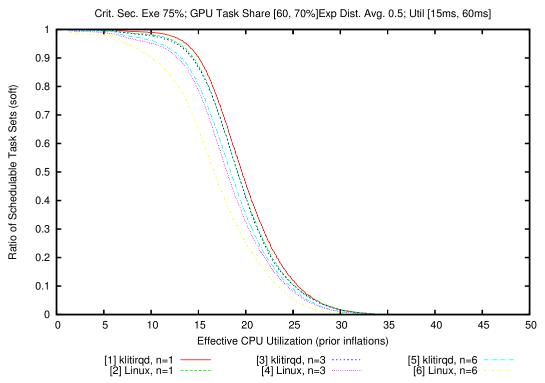


Figure 303. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

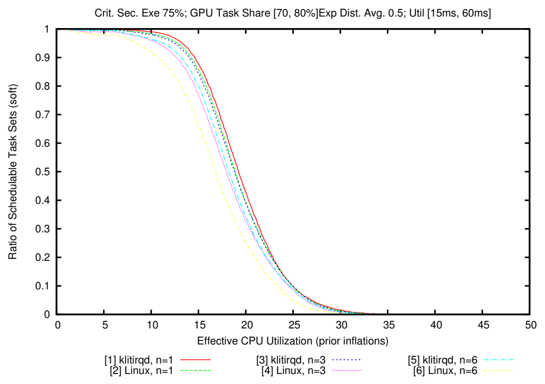


Figure 304. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

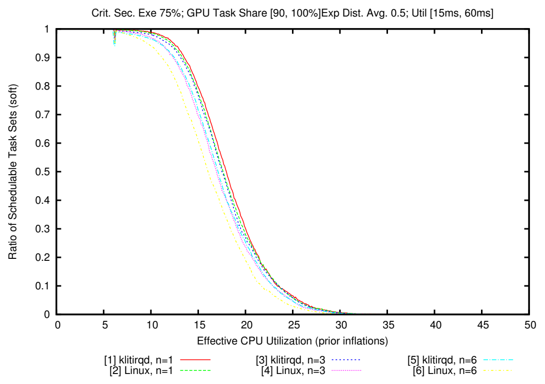


Figure 306. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

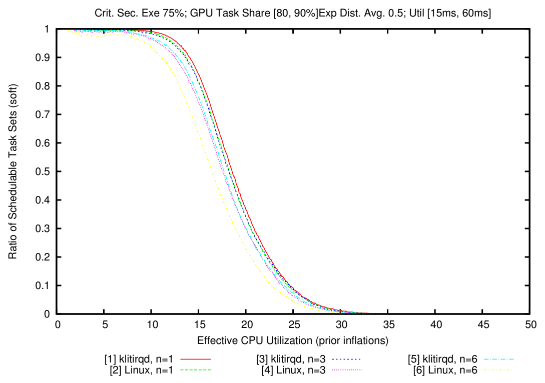


Figure 305. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 8×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

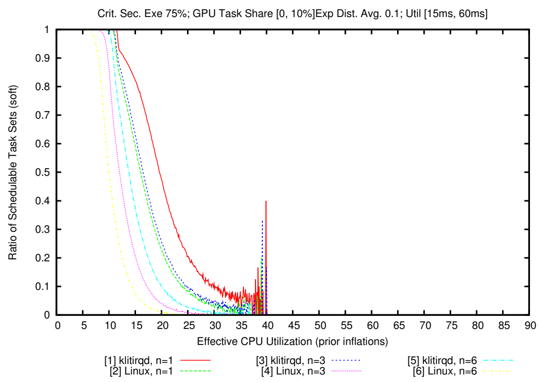


Figure 307. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

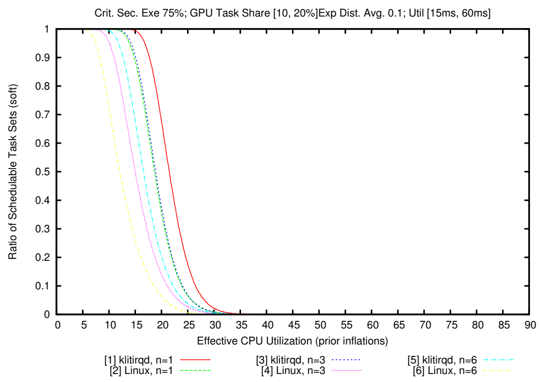


Figure 308. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

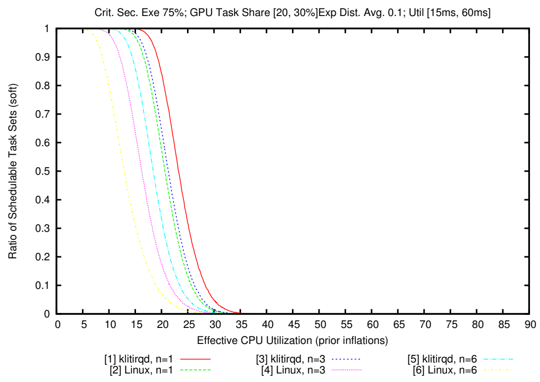


Figure 309. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

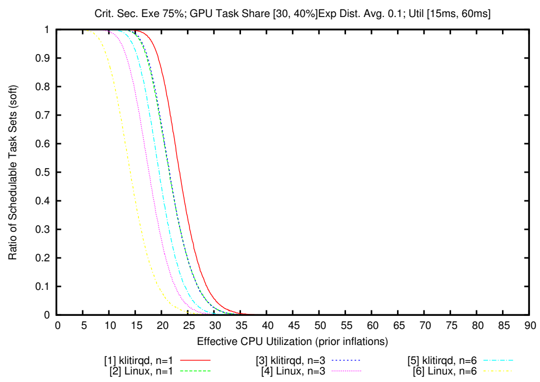


Figure 310. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

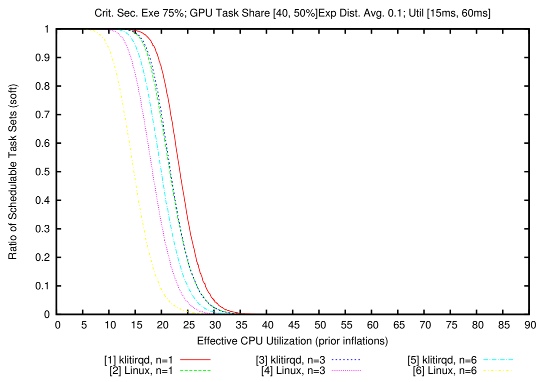


Figure 311. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

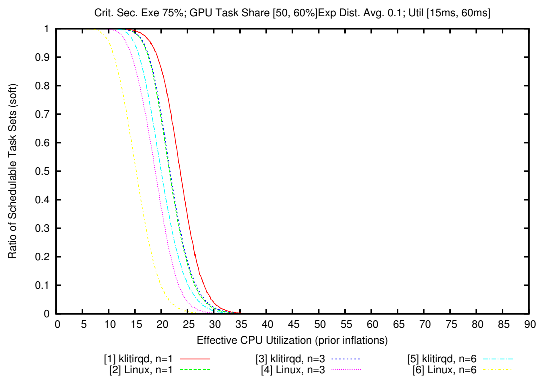


Figure 312. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

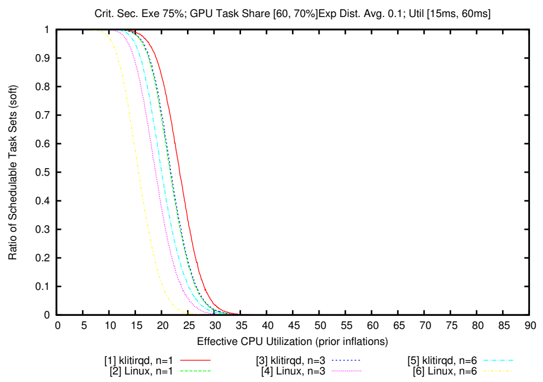


Figure 313. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

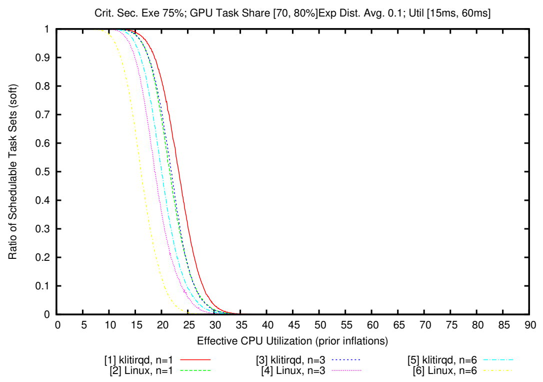


Figure 314. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

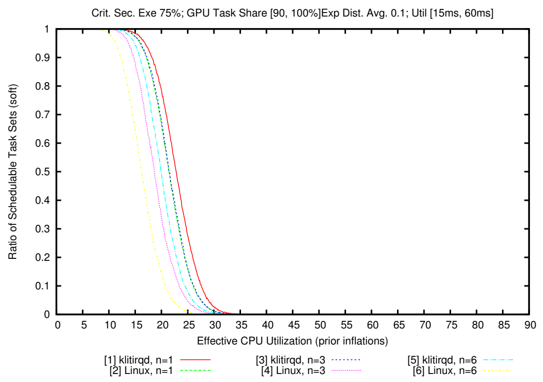


Figure 316. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

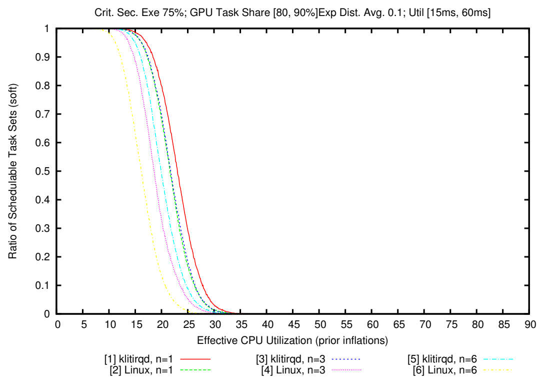


Figure 315. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 10%.

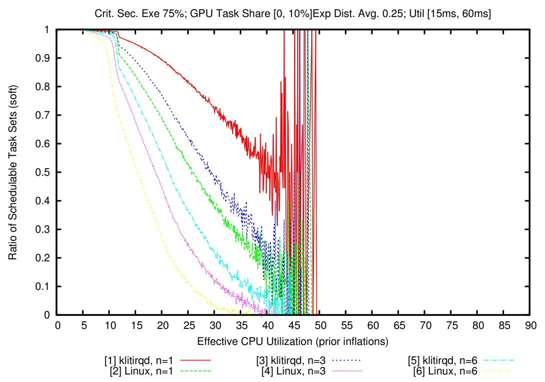


Figure 317. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

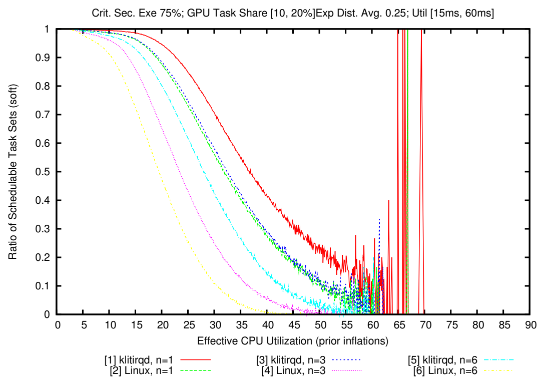


Figure 318. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

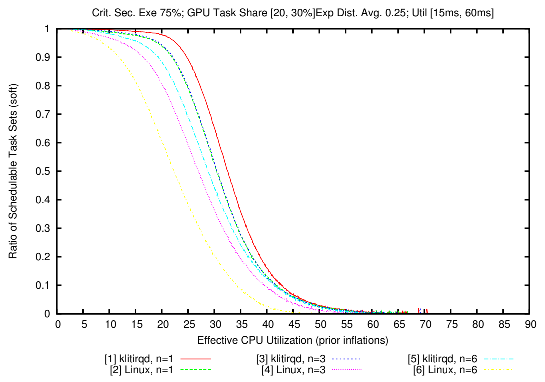


Figure 319. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

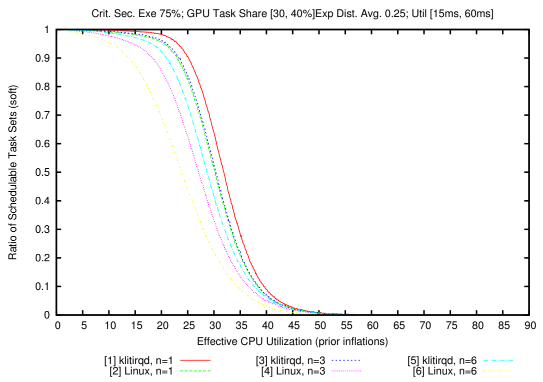


Figure 320. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

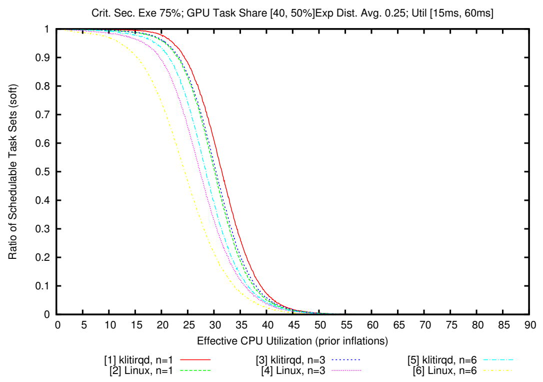


Figure 321. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

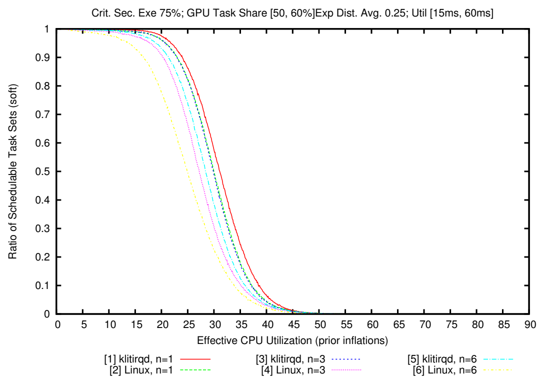


Figure 322. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

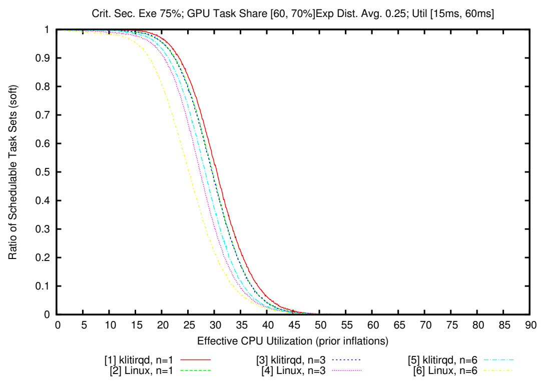


Figure 323. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

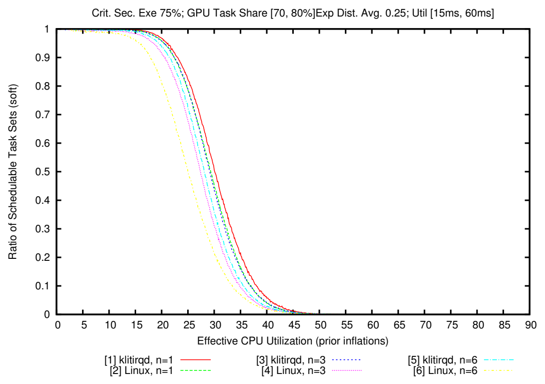


Figure 324. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

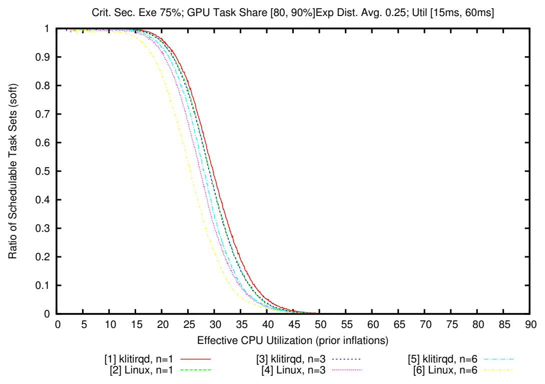


Figure 325. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

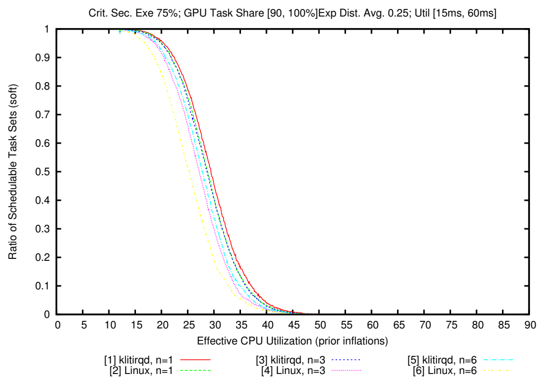


Figure 326. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 25%.

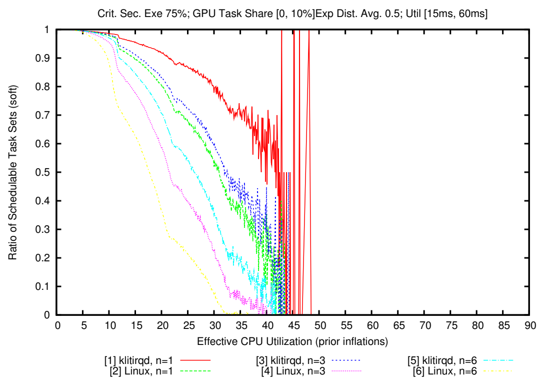


Figure 327. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [1%, 10%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

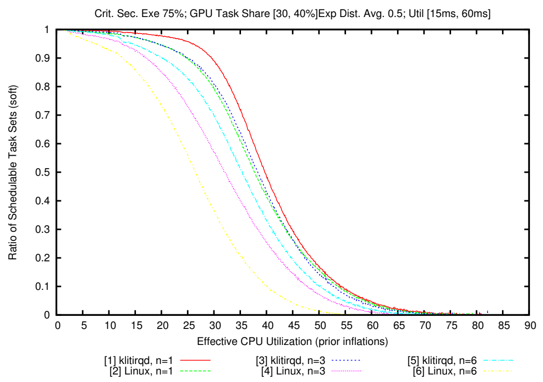


Figure 330. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [30%, 40%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

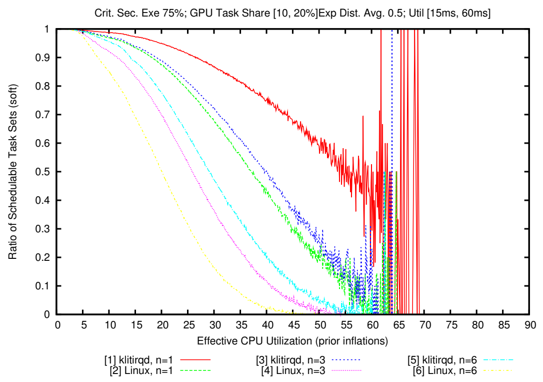


Figure 328. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [10%, 20%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

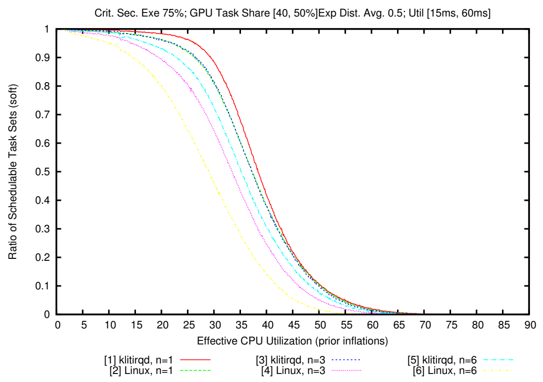


Figure 331. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [40%, 50%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

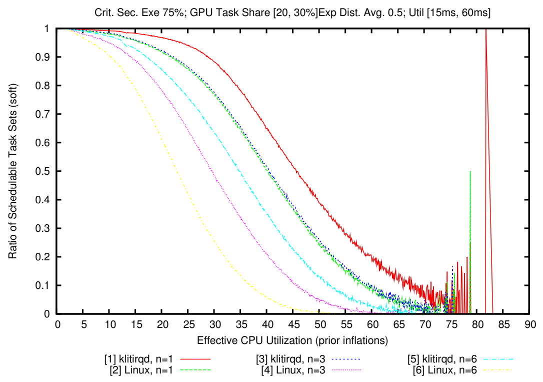


Figure 329. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [20%, 30%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

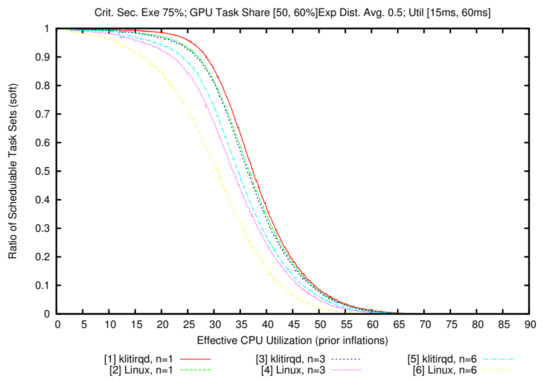


Figure 332. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [50%, 60%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

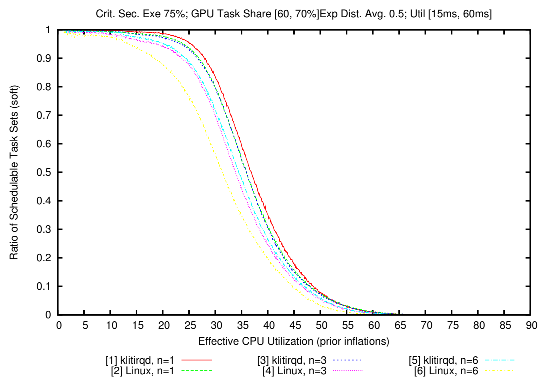


Figure 333. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [60%, 70%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

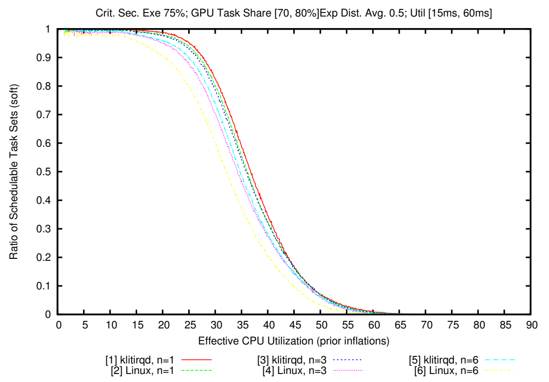


Figure 334. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [70%, 80%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

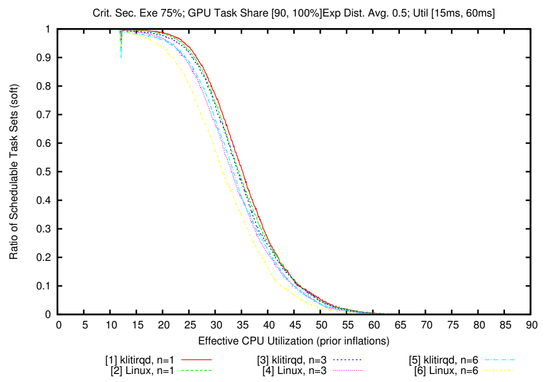


Figure 336. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [90%, 100%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.

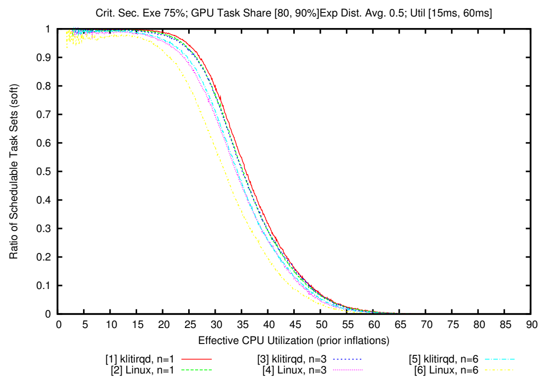


Figure 335. The percentage of schedulable task sets as a function of effective system utilization with a GPU speed-up of 16×. The percentage number of GPU-using tasks per task set is in the range [80%, 90%]. Suspension-oblivious per task utilization selected from the exponential distribution with an average utilization of 50%.