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

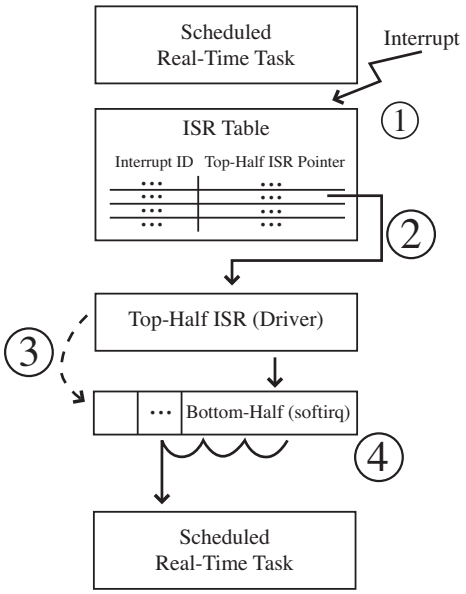

Figure 1. The interrupt handling in Linux. **(1)** An interrupt occurs and the currently scheduled task is suspended. **(2)** The ISR for the interrupt type is executed. **(3)** The ISR may issue deferred work as a tasklet. **(4)** Before resuming the interrupted task, up to ten softirqs are executed,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

softirq used by most I/O devices, including GPUs; we use the terms "softirq" and "tasklet" synonymously.

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, one thread dedicated to each I/O device. 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

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

standard Linux `tasklet_schedule()` function call to issue a tasklet to the kernel, an alternative function `litmus_tasklet_schedule()` is provided to issue a tasklet 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

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

and many Linux users use custom kernel configurations, the NVIDIA driver is not distributed as a monolithic 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^{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].

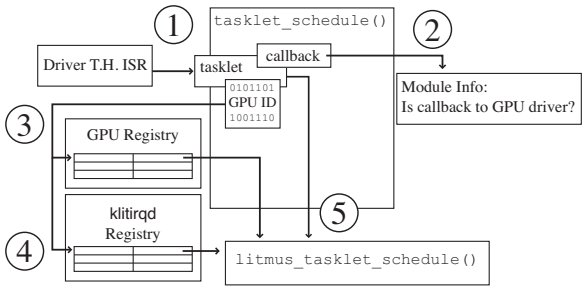

Figure 2. GPU tasklet redirection. **(1)** A tasklet from the GPU driver is passed to `tasklet_schedule()`. **(2)** The tasklet is intercepted if the callback points to the driver. **(3)** The GPU identifier is extracted from the memory block attached to the tasklet using a known address offset and the 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$^{\mathrm{RT}}$. 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$^{\mathrm{RT}}$ 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$^{\mathrm{RT}}$ configured to use klitirqd and once in LITMUS$^{\mathrm{RT}}$ using standard Linux interrupt handling. Scheduling logs were recorded, from which we compared the performance of klitirqd and standard Linux interrupt handling in LITMUS$^{\mathrm{RT}}$ according to several metrics, as discussed next.

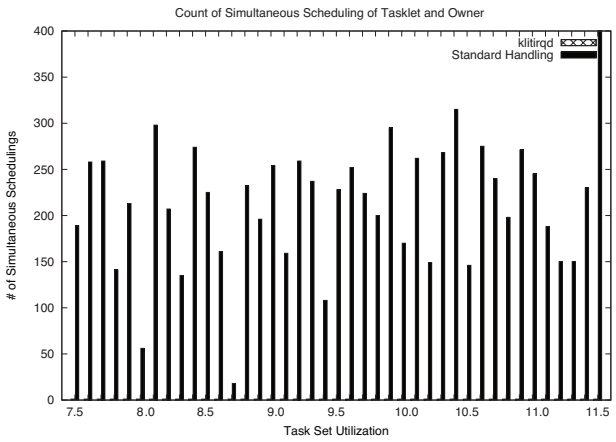

Figure 3. Histogram of concurrent execution events of a tasklet and its owner. No concurrent execution events were observed under klitirqd. There is no apparent trend with regard to task set utilization: the number of events observed differs greatly among the task sets.

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

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

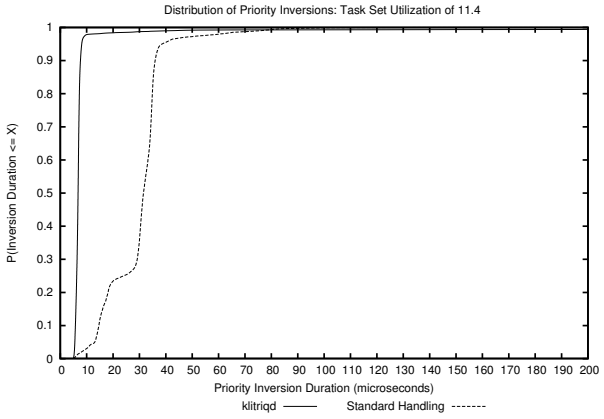

Figure 4. Cumulative distribution of priority inversion durations.

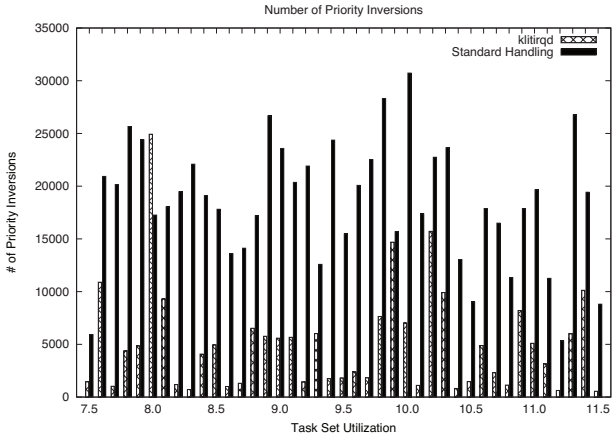

Figure 5. Histogram of detected inversions. There is no observable trend in task set utilization with this sample size, though variance among task sets is significant.

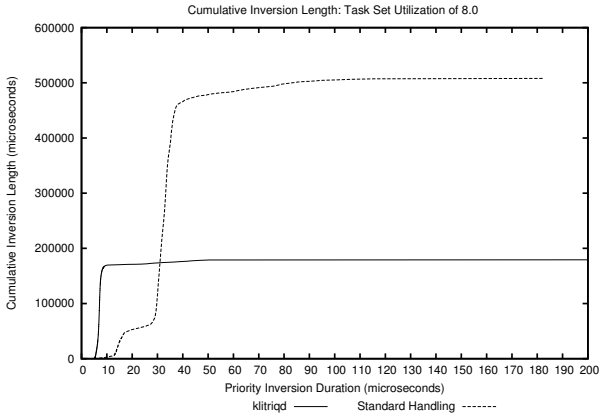

Figure 6. Cumulative priority inversion length as a function of maximum priority inversion length for the task set with utilization 8.0. The total duration of priority inversion is more than twice as large under standard Linux interrupt handling than klitirqd, despite an increased number of priority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

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 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.

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.

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.

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 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

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

be a desirable solution in all applications, especially in 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.

[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.

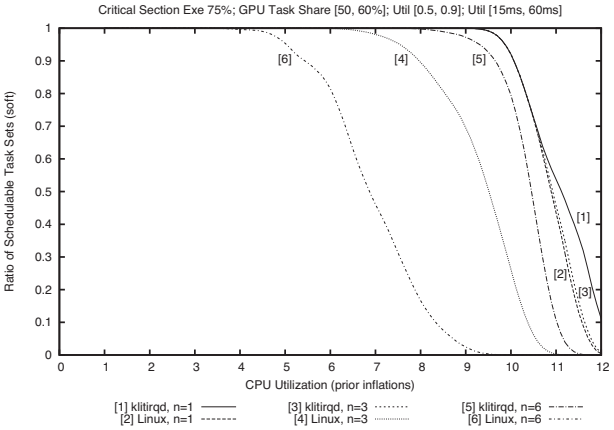

Figure 7. The percentage of schedulable task systems (y-axis) as a function of CPU utilization (x-axis) under klitirqd (threaded) interrupt handling and standard Linux (non-threaded) interrupt handling.

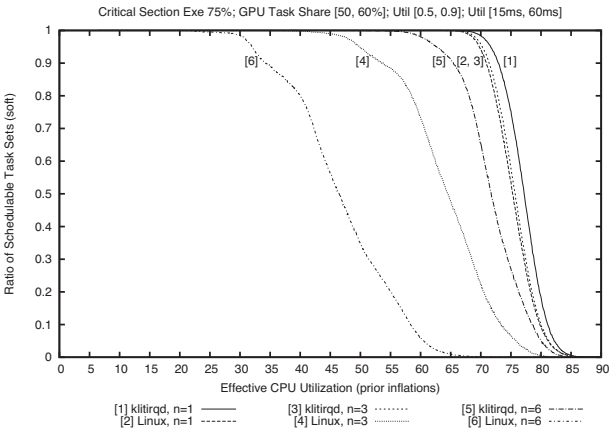

Figure 8. Schedulability results similar to Fig. 7 except that a GPU speedup of 16× is assumed.

The primary motivation for utilizing GPUs in a real-time system is increased performance. The benefits of threaded GPU interrupt handling are even more clear when *effective utilization* is considered instead of actual CPU utilization. By supposing a GPU-to-CPU speed-up ratio of R, we may convert each GPU-using task into a functionally equivalent CPU-only task by viewing each time unit spent executing on a GPU as R times units spent executing on a CPU. We define effective utilization to be the utilization of a task set after such a conversion. Fig. 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 35.0 CPUs to achieve the same degree of schedulability under Linux interrupt handling. This is nearly half the performance of klitirqd!

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.