Robust Real-Time Multiprocessor Interrupt Handling Motivated by GPUs

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

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

I. Introduction

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

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

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

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

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

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

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

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

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

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

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

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

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

II. Interrupt Handling

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

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

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

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

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

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

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

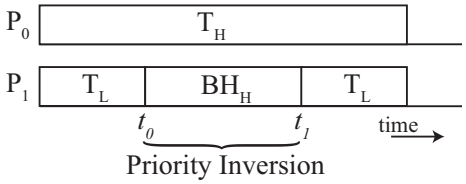

Figure 1. Priority inversion: T_L should be scheduled on processor P_1 at time t_0 since it is one of the top two highest-priority tasks.

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

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

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

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

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

III. Interrupt Handling in LITMUS^{RT}

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

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

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

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

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

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

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

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

never co-scheduled with its tasklet owner. This allows asynchronous I/O to be supported *without violating the single-threaded task models* commonly assumed.

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

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

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

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

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

IV. GPU INTEGRATION

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

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

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

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

[6] PAI was originally designed for uniprocessor systems.

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

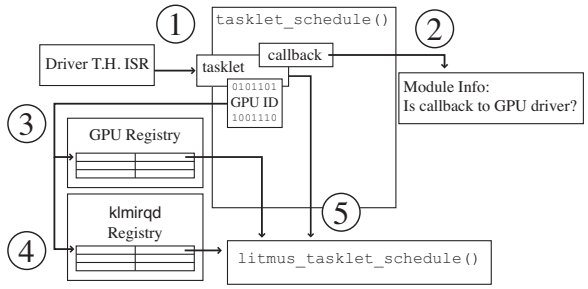

Figure 2. GPU tasklet redirection. **(1)** A tasklet from the GPU driver is passed to `tasklet_schedule()`. **(2)** The tasklet is intercepted if the callback points to the driver. **(3)** The GPU identifier is extracted from the data attached to the tasklet using a known address offset, and the GPU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

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

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

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

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

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

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

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

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

V. Evaluation of Priority Inversions

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

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

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

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

[10] A full description of the k-FMLP is available in Appendix A of 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 [30].

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

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

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

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

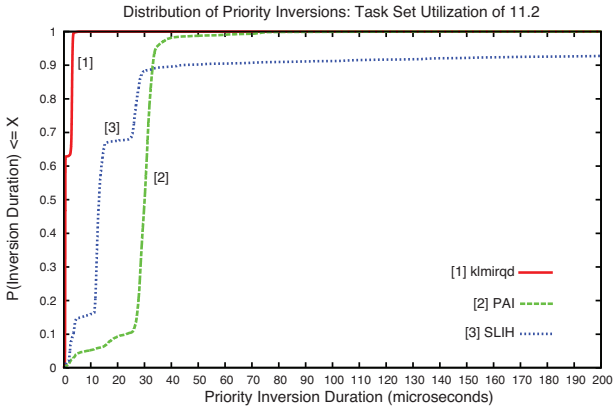

Figure 3. Cumulative distribution of priority inversion durations.

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

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

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

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

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

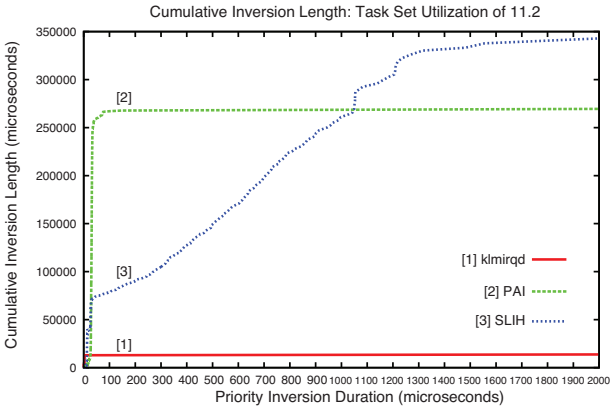

Figure 4. Cumulative priority inversion length as a function of maximum priority inversion length for the task set with utilization 11.2. The total duration of priority inversion is much less under klmirqd in comparison to PAI and SLIH. The curves for klmirqd and PAI plateau abruptly, indicating little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

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

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

VI. System-Wide Evaluation of Interrupt Handling Methods

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

VII. OVERHEAD-AWARE SCHEDULABILITY

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

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

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

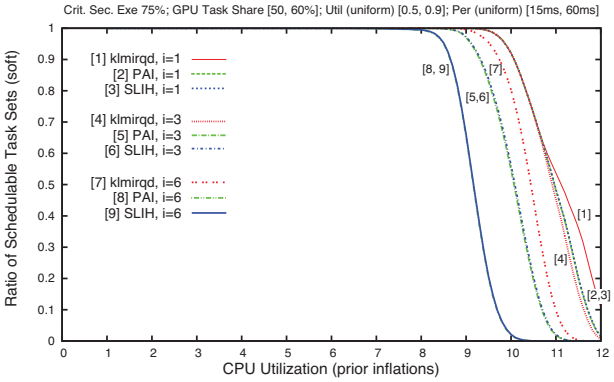

Figure 5. The percentage of schedulable task systems (y-axis) as a function of CPU utilization (x-axis) under klmirqd (threaded) interrupt handling, PAI (non-threaded) handling, and SLIH (non-threaded) interrupt handling.

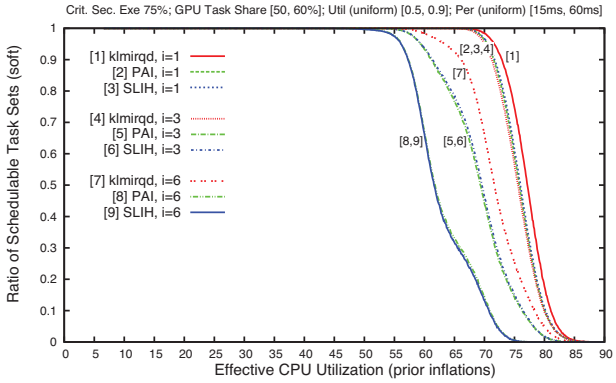

Figure 6. Schedulability results similar to Fig. 5 except that a GPU speedup of 16× is assumed.

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

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

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

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

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

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

VIII. Conclusion

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

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

[16] Additional graphs are available in Appendix D of the online version of this paper.

[17] Barring priority inheritance mechanisms from locking protocols.

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, is competitive with other real-time methods, and offers better results in terms of overall schedulability (with overheads considered).

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

ACKNOWLEDGMENT

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

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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