Real-Time Handling of GPU Interrupts in LITMUS^{RT}

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

***Abstract*—Graphics processing units (GPUs) are becoming increasingly important in today's platforms as their increased generality allows for them to be used as powerful co-processors. However, unlike standard CPUs, GPUs are treated as I/O devices and require the use of interrupts to facilitate communication with the CPU. Interrupts cause delays in the execution of real-time tasks, challenges in real-time operating system implementation, and difficulties for formal analysis. We examine methods for designing proper real-time interrupt handling in multiprocessor systems and present our solution, klitirqd, an addition to LITMUS^{RT}, to address the challenges caused by interrupts in real-time systems. klitirqd is a flexible solution that improves upon prior approaches by supporting non-partitioned multiprocessor scheduling algorithms while respecting the single-threaded sporadic task model and also supporting asynchronous I/O. We use klitirqd to realize real-time GPU interrupt handling and overcome significant technical challenges of altering the interrupt processes of a closed-source GPU driver. This technique can be generalized to potentially support any closed-source device driver.**

I. Introduction

The parallel architecture of the graphics processing unit (GPU) often allows data parallel computations to be carried out at rates orders of magnitude greater than those offered by a traditional CPU. Enabled by increased programmability and single-precision floating-point support, the use of graphics hardware for solving non-graphical (general purpose) computational problems began gaining wide-spread popularity in the early part of the last decade [9], [13], [18]. However, early approaches were limited in scope and flexibility because non-graphical algorithms 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 language extensions and run-time environments,[1] eliminating many of the limitations found in early GPGPU solutions. Since the release of these second-generation GPGPU technologies, both graphics hardware and runtime environments have grown in generality, increasing the applicability of GPGPU to a breadth of domains. Today, GPUs can be found integrated on-chip in mobile devices and laptops [1], [4], [2], as discrete cards in higher-end consumer computers and workstations, and also within some of the world's fastest super-computers [17].

GPUs have applications in a number of real-time domains. For example, a GPU can efficiently carry out many digital signal processing operations such as multidimensional FFTs and convolution as well as matrix operations such as factorization on data sets of up to several gigabytes in size. These operations, coupled with other GPU-efficient algorithms, can be used in medical imaging and video processing, where real-time constraints are common. A particularly compelling application is that of driver-assisted and autonomous automobiles. In these platforms, multiple streams of data from video feeds and laser range sensors must be processed and correlated for localized navigation and obstacle avoidance [25]. GPUs are well suited to handle this type of workload since the sensors generate huge amounts of data and the algorithms used are often data parallel. Moreover, autonomous automobiles are clearly a safety-critical application where real-time constraints are important.

Prior Work: The real-time community has only recently begun investigating the use of GPUs in real-time systems. On the theoretical side, Raravi et al. have developed methods for estimating worst-case execution time on GPUs [21] and scheduling algorithms for "two-type" heterogeneous multiprocessor platforms, with CPU/GPU platforms particularly in mind [6], [22]. On the more applied side, Kato et al. have developed quality-of-service techniques for graphical displays on fixed-priority systems [15], [16].

In our own work, we have investigated and addressed many of the challenges faced when integrating GPUs that have non-real-time, throughput-oriented, closed-source device drivers into an actual real-time operating system [12] on a multiprocessor platform. The non-real-time-oriented drivers exhibit behaviors that are difficult to manage in a real-time system. For example, the driver only allows one task to execute work non-preemptively on a GPU at a time. When a GPU comes under contention, a blocked task waits on a spinlock, consuming CPU time, until it receives the GPU. This wastes CPU

[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

time that could be used to schedule other real-time tasks. To make this situation more difficult, blocked tasks have no mechanism to change the priority of a GPU-holding task. Thus, real-time tasks can be blocked, consuming CPU time, for an unbounded duration of time.

Additionally, a GPU-using task may self-suspend from CPU execution while it waits for computational results from a GPU. The non-preemptive execution on the GPU usually takes from 10s of milliseconds up to several seconds, depending upon the application [12]. Supposing that we could somehow bound the duration a blocked task spin-waits for a GPU, this bounded duration would still be quite long and a considerable amount of CPU time consumed. The primary solution we presented in [12] to address these issues is to treat a GPU as a shared resource, protected by a real-time suspension-based semaphore. The use of a real-time semaphore removes the GPU driver from resource arbitration decisions (since all arbitration decisions have already been made by the time a task invokes the driver) and priority inheritance mechanisms make it possible to bound blocking time. The fact that the semaphore is suspension-based (blocked jobs suspend from execution) increases the CPU availability to other real-time tasks. We validated this approach in experiments on UNC's real-time Linux-based operating system, LITMUS^{RT}, and demonstrated improved real-time characteristics such as reduced CPU utilization and reduced deadline tardiness.

Contributions: One aspect that we did not address in our prior work was the effect hardware interrupts from GPUs have on real-time execution. Interrupts for cause complications in real-time systems by introducing increased system latencies, decreased schedulability, and additional complexity in real-time operating systems. Proper real-time interrupt handling should ideally respect the priorities of executing real-time tasks. However, this is a non-trivial task, especially for systems with shared I/O resources. In this paper we examine the nature, servicing techniques, and effects general interrupts have on real-time execution. We present an approach for the proper real-time handling of interrupts and improve upon prior methods by supporting non-partitioned multiprocessor scheduling algorithms without violating the single-threaded sporadic task model, while also supporting asynchronous I/O. We apply our technique to the real-time scheduling of GPU interrupts, which also required us to overcome significant technical challenges to alter the interrupt processes of the closed-source GPU.

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 how interrupts are processed in Linux-based operating systems. In Sec. III, prior work in real-time interrupt handling is discussed, and we present our solution, klitirqd, for scheduled interrupt handling in LITMUS^{RT}. In Sec. IV we apply klitirqd to handle GPU interrupts, and explain in great detail how interrupt processing from the closed-source GPU driver must be intercepted, parameters decoded, and rerouted to klitirqd. Our implementation in LITMUS^{RT} for real-time GPU interrupt handling is evaluated in Sec. V and compared against standard Linux interrupt processing through experimentation on a multicore, multi-GPU, platform. Finally, we conclude with a summary of our approach and results in Sec. VI and also present future research enabled by this work.

Please note for the rest of this paper we will focus our attention on GPU technologies from the manufacture NVIDIA. NVIDIA's CUDA [3] platform is widely accepted as the leading solution for GPGPU.

II. Interrupt Handling

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

Interrupts are difficult to manage in a real-time system. Interrupts may come periodically, sporadically, or at entirely unpredictable moments, depending upon the application. Interrupts often cause disruptions in a real-time system since the CPU must temporarily halt the execution of the currently scheduled task. In uniprocessor and partitioned multiprocessor systems, one may be able model an interrupt source and handler as the highest-priority real-time task in a system or as a blocking source [19], [14], 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 [11]. 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. Stack corruption would occur if a task resumed execution before the interrupt handler completed. 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 the 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. This, in essence, describes proper interrupt handling in a real-time system. However, achieving this in practice is actually more complicated.

Interrupt Handling In Linux: We now review how split interrupt handling may be designed and implemented. While we will discuss real-time approaches, we will first examine interrupt handling in Linux.[2] We do this for several reasons. First, Linux provides the basis for LITMUS^{RT}, the platform considered in this work. Second, and much more importantly, high-performance GPU drivers are currently only available for general-purpose operating systems, specifically: Microsoft Windows, Mac OS X, Solaris, FreeBSD, and Linux. At the moment, any real-time system using GPUs built from available technology must be based upon FreeBSD or Linux since the other three listed operating systems are not real-time and are closed-source. Only these two open-source operating systems can be modified to support real-time scheduling and interrupt handling (they themselves are not real-time without modification). Between FreeBSD and Linux, Linux is the most likely choice due to its wide adoption, its ability to run on platforms ranging from embedded systems to supercomputers, and its vigorous community support. Further, FreeBSD is not currently supported by the major GPU manufacture AMD.

During the initialization of the Linux kernel, kernel components and device drivers (such as a GPU driver) 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>`. Interrupts may be classified as global or local. Local interrupts, as the name implies, are only handled by the processor on which they are raised. In contrast, global interrupts may be dispatched to any system CPU that is configured to receive the given type of interrupt. By default, all CPUs in Linux are configured to receive all global interrupts, though CPUs can be later "shielded" from doing so.

[2]Please note that what follows is only a short overview of Linux interrupt handling; please see [10] for a more complete description.

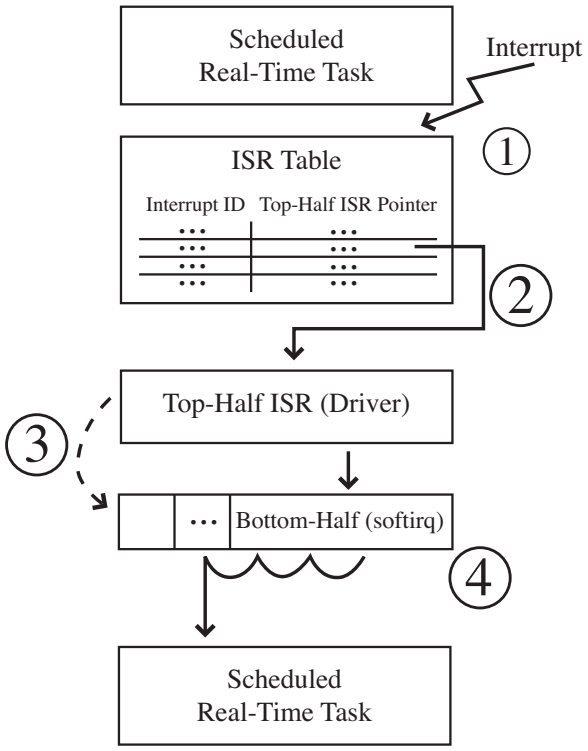

Figure 1. The interrupt handling sequence in Linux. **(1)** An interrupt occurs and the currently scheduled task is suspended. **(2)** The interrupt service routine for the interrupt type is executed. **(3)** The driver may schedule deferred work in the form of a tasklet. **(4)** Before resuming the interrupted task, up to ten softirqs are executed, possibly including tasklets scheduled in (3).

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 processing 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 are illustrated in Fig. 1. There are several types of softirqs, but for the scope of this paper, we focus on *tasklets*, which are the type of softirq used by most I/O devices, including GPUs; we will use the terms softirq and tasklet synonymously.

The standard Linux kernel executes softirqs using a heuristic. Immediately after executing an interrupt top-half, but before resuming execution of an interrupted

thread, the kernel will execute up to ten bottom-halves. Any pending softirqs remaining are dispatched to one of several (one for each CPU) kernel threads dedicated to softirq processing; these are the "ksoftirq" daemons. This thread is scheduled within Linux with a very high-priority, but is a schedulable and preemptible entity nonetheless. The mechanism of executing a fixed number of pending bottom-halves after every interrupt can introduce an unacceptably long latency into interrupt processing in a real-time system and causes one to wonder if this can even be considered a split interrupt system. In all likelihood, a system experiencing few interrupts (though it may still be heavily utilized) will execute top- and bottom-halves in pairs. That is, for every top-half that yields a bottom-half, that bottom-half will subsequently be executed before the interrupt processing completes. In the unlikely event that 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-halves (except the most critical, such as timers) with a worker-pool of schedulable threads. However, even PREEMPT_RT has its limitations. First, due to its POSIX real-time underpinnings, PREEMPT_RT is only suitable for use in static-priority real-time systems. Second, like the standard Linux kernel, softirq threads are pinned to individual CPUs and preclude their use in a globally-scheduled system. Finally, the scheduling priorities of the softirq threads are fixed (but may be changed manually). To illustrate the limitations inherent in assigning a fixed priority to a softirq thread, consider the situation where a real-time task blocks while waiting for an interrupt from an I/O device. One would hope that the softirq thread processing this interrupt would be scheduled with the priority of the blocked task. However, since the softirq cannot change the priority of its thread until it is scheduled, either the blocked task or higher-priority tasks may experience delays as the result of a priority inversion. If the softirq thread is scheduled with a lesser priority than the blocked task, then the blocked task is delayed. If the softirq thread is scheduled with a greater priority than the blocked task, then other tasks with lesser priorities than the softirq thread may be delayed. A system designer, through the careful use of interrupt shields, might be able to ensure that a given softirq thread executes on behalf of a single task, and thus be able to assign equal scheduling priorities to both. However, this precludes tasks of differing priorities to share a single device (and softirq thread), such as a GPU.

Neither the standard nor PREEMPT_RT variants of the Linux kernel are robust enough to implement proper real-time-schedulable split interrupt handling. The implementation of a complete real-time solution would require deep changes to many core Linux components, device drivers, and even user-level processes. Since Linux is primarily a general-purpose operating system, such changes are unlikely to occur in the near future. However, supposing Linux could implement proper split interrupt handling, how might it be done?

III. Interrupt Handling in LITMUS^{RT}

LITMUS^{RT} is a real-time Linux-based testbed that has been under continual development at UNC for over five years. Our research group has used LITMUS^{RT} to evaluate many real-time scheduling algorithms and locking protocols and has offered many valuable insights into the implementation constraints of real-time systems. 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 the previous section, current Linux-based operating systems use fixed-priority CPU-pinned softirq daemons. In this paper, we introduce a new class of LITMUS^{RT}-aware daemons called klitirqd. 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. At the moment klitirqd only supports tasklet processing (suitable in most I/O situations), though it should be relatively simple to extend support to all types of softirqs.

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 amongst many devices. The detailed implementation of klitirqd is as follows. Instead of using the standard Linux `tasklet_schedule()` function call to issue a tasklet to the kernel, an alternative function `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.

At first glance, one may assume that the klitirqd threads are scheduled with their adopted priorities by LITMUS^{RT} as determined by the active real-time scheduling algorithm.[3] However, this is only part of the solution. LITMUS^{RT} supports single threaded sporadic task systems on multiprocessor platforms. If a klitirqd thread were to service a tasklet in the manner described above, then it would be possible for klitirqd to execute simultaneously with the owner of that tasklet on a multiprocessor system. From the point of view of scheduling, this would essentially make the owner of a tasklet multithreaded. *This breaks any single-threaded real-time analysis.* While this may not be an issue for tasks that block on synchronous I/O (a blocked task cannot execute simultaneously with the klitirqd servicing its tasklet), this does become an issue for asynchronous I/O. In asynchronous I/O, a task may issue a batch of I/O requests while continuing on to other processing. It is only at some time later the this task rendezvouses with I/O results. Asynchronous I/O helps improve overall performance and is commonly used in GPU applications to mask bus latencies.

To address asynchronous I/O, we require that all tasks in LITMUS^{RT} that are ready to execute hold a per-task mutex. Regular real-time tasks acquire this mutex at every job release. If a task blocks on I/O, it releases this mutex. The klitirqd daemon that processes the corresponding tasklet must acquire the mutex of the tasklet owner before executing and the mutex is released upon tasklet completion. Thus, the scheduling between klitirqd and the tasklet owner is mutually exclusive. In the case of asynchronous I/O, the klitirqd daemons buffer pending tasklets and are only executed when the corresponding owner releases its mutex to rendezvous with I/O results. The sporadic task model is thus preserved.

We recognize that similar architectures for split interrupt handling have been proposed and implemented before. For instance, LynxOS [5] 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. However, LynxOS only supports fixed-priority scheduling. Steinberg et al. have also developed and implemented similar techniques based upon bandwidth inheritance to support interrupt processing in a modified L4 microkernel [24] and the NOVA microhypervisor [23]. While both of 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 the single-threaded sporadic task model. LynxOS supports non-partitioned scheduling, but breaks the single-threaded sporadic task model. Steinberg et al.'s methods are limited to uniprocessor and partitioned systems, which requires 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 the methodology we use here, Manica et al. presented an implementation of real-time scheduled interrupt handlers in Linux [20]. 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 handling threads were pinned to individual processors and did not use priority inheritance mechanisms suitable for non-partitioned multiprocessor scheduling.

IV. GPU Integration

Having given sufficient attention to proper real-time interrupt handling and our approach in LITMUS^{RT}, we apply these techniques to arrive at a complete real-time GPU interrupt handling infrastructure.

In Sec. III, we described how interrupt handlers are to call the function `litmus_tasklet_schedule()` to dispatch bottom-half tasklets to klitirqd. The caller must provide two parameters: (1) the *owner* of the tasklet (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. It should be relatively straightforward for a LITMUS^{RT}-aware device driver to provide these parameters, but how shall we accomplish this with a closed-source GPU driver that cannot even be modified to call `litmus_tasklet_schedule()`? This is a significant technical challenge and requires a detailed study of the interactions between the black-box driver and the operating system. We will accomplish this in four parts: tasklet interception, device identification, and 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,

[3] LITMUS^{RT} supports a variety of real-time schedulers through a plugin architecture.

we modify the standard internal Linux API function `tasklet_schedule()`.

When a kernel component calls `tasklet_schedule()`, it must provide a callback function pointer that specifies the entry point for the execution of the deferred work. If we can identify callbacks to the GPU 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 redirect the scheduling of the deferred work.

Device Identification: In a platform with 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 associated callback function. Included in this block of data is a device identifier (ranging from 0 to g, where g is the number of system GPUs), which indicates which GPU raised the interrupt. However, accessing this data within the memory block is challenging since it is packaged in a driver-specific format. Fortunately, the driver's links into the open source OS code allow us to locate the device identifier.

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

This brings us half-way into our process of redirecting deferred work from the GPU driver to klitirqd processing. We can 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 [12], 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 exclusive access to the GPU. This mechanism was complemented by the fact that the programs that execute on the GPU, called *kernels*[5], do so non-preemptively—the GPU cannot be shared between tasks even without real-time mutual exclusion controls.[6]

The execution of a basic GPU-using job (the sporadic invocation of a GPU-using task) goes through several phases. In the first phase, the job executes purely on the CPU. In the next phase, the job sends data to the GPU, which is stored in device memory, for use by the GPU kernel. Next, the job suspends from the CPU when the kernel is invoked on a GPU. The GPU executes the kernel using many parallel threads, but kernel execution does not complete until after the last GPU-thread has completed. Once the kernel has completed, an interrupt is sent to the host CPU and handled by the device driver, which subsequently wakes the blocked job from its suspension. The job resumes execution on the CPU and copies back the kernel results from the GPU back to main-memory. Optionally, the job may continue executing on the CPU without using the GPU. Thus, a GPU-using job has five execution phases as depicted in Fig. 2. This model of a GPU-using job is a generalization of potentially more complex execution patterns. A more complex job may execute multiple kernels and may communicate with the GPU in between them.[7] However, this model this pattern with beginning and ending points. That is, when a job begins using the GPU and when it is finishes using the GPU; it is this interval that we protect with a real-time semaphore.

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

[5]This name is derived historically from "kernel" in mathematics. We will use "OS kernel" to refer to the operating system and "GPU kernel" to refer to a GPU program instance.

[6]It is possible for two tasks to share a GPU by overlapping data transmission of one task with the GPU execution of another. This is often done to mask transmission latencies. However, we leave an investigation into real-time solutions for this usage pattern for future work.

[7]For example, a `memset()` operation on device memory called from a host-side application is implemented as a small kernel which copies the given byte value to all bytes in the specified memory.

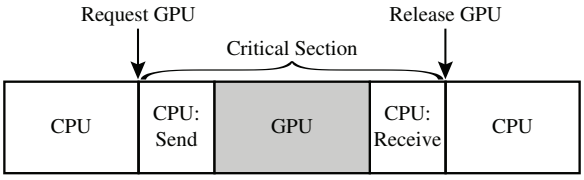

Figure 2. Execution phases of a GPU-using job.

We extend our prior solution to not only use a real-time semaphore to arbitrate GPU access, but to also act as registry for tasks actively using GPUs. Whenever a GPU is allocated to a task by the OS kernel, an internal lookup table indexed by device identifier (ranging from 0 to g) is updated to record device ownership. 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 tasklet to klitirqd; now we must determine which klitirqd instance will perform the processing.

klitirqd Dispatch: The architecture of klitirqd is general enough to support any number of daemon instances, all scheduled by a JLSP real-time scheduler. Since a system may have g GPUs, we should have at least g klitirqd instances to ensure that all GPUs can be used simultaneously. However, such an approach may cause implementation difficulties in schedulers where task migrations between CPUs are constrained. In order to remain flexible and avoid any dependencies on any particular scheduler implementation, for each scheduling domain (ready queue), we allocate one klitirqd daemon per GPU that may be accessed within that domain. For example, in a system with four GPUs where CPUs are globally scheduled, there are four klitirqd daemons for GPU processing. If that system where scheduled in clusters and GPUs were shared across clusters, then each cluster would have four klitirqd daemons. If GPUs were isolated to particular clusters, perhaps one each per CPU cluster, then each cluster would only have one klitirqd daemon.

Each klitirqd daemon used for GPU processing is assigned to a specific GPU within that scheduling domain. This assignment is recorded in another lookup table, which is referenced within our modified `tasklet_schedule()`. A tasklet from the GPU driver is intercepted, the source device is identified, the ownership of that device is found, and the proper klitirqd instance known. With this information, we modify `tasklet_schedule()` to redirect all GPU tasklets to klitirqd by calling `litmus_tasklet_schedule()`.

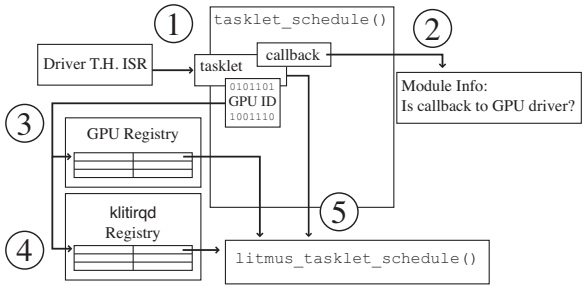

Figure 3. GPU tasklet redirection. **(1)** A tasklet from the GPU driver is passed to `tasklet_schedule()`. **(2)** The tasklet is intercepted if the callback points to the driver. **(3)** The GPU identifier is extracted from the memory block attached to the tasklet using a known address offset and the 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()`.

This process is summarized in Fig. 3.[8]

V. Evaluation

Evaluation Platform: Motivated by the autonomous automotive application discussed in Sec. I, we validated our GPU interrupt handling approach on a multicore platform with several GPUs. Encouraged by results from [7], we will limit our attention to a multiprocessor system scheduled by the clustered earliest-deadline-first algorithm, with GPUs statically assigned to clusters. As described in the previous sections, the allocation of GPUs to tasks can be performed through the treatment of system GPUs as shared resources with access arbitrated by real-time locking protocols. In this paper, we use a simple k-exclusion locking protocol that extends the *Flexible Multiprocessor Locking Protocol* (FMLP) [8] so that k resources (our GPUs) can be simultaneously managed. This k-exclusion protocol, which we term the k-FMLP, was implemented in LITMUS^{RT} to support this work.[9] Special considerations had to be paid to integrate with klitirqd. Specifically, the priority inherited by a GPU holder must also be propagated to the associated klitirqd.

Our evaluation platform is as follows. We use a dual-socket six-core Intel Xeon X5060 CPU platform, for a total of twelve cores running at 2.67GHz, scheduled with the clustered earliest-deadline-first algorithm. Clustering is split along the NUMA architecture of the system, yielding six cores per cluster. Incidentally, an L3 cache is shared within each cluster. Our platform also includes eight Nvidia GTX-470 GPUs. Though any CPU core can

[8]GPU tasklets may spawn additional deferred work using Linux work queues, which are dedicated to process work that is "more deferrable" than tasklets. We replicate our GPU tasklet solution to perform this work to avoid any priority inversions.

[9]Please see Appendix A of the online version of this paper at http://www.cs.unc.edu/~anderson/papers.html for a more complete description of the k-FMLP.

access any GPU, we statically assign four GPUs to each cluster and a k-FMLP semaphore (where $k = 4$) protects each GPU pool. This GPU assignment complements the I/O bus architecture of the system, which is also split along NUMA boundaries. This clustering of the CPUs and GPUs attempts to minimize bus contention which can significantly affect data transmission rates between CPUs and GPUs. Finally, four klitirqd daemon instances are run in each cluster, one for each GPU.

We use CUDA 4.0 Release Candidate 2 for our GPU runtime environment. We opted to use this release candidate environment because CUDA 4.0 offers significant improvements for multi-GPU platforms. We encountered no stability issues with the release candidate drivers.

Experimental Setup: In these experiments, we paid considerations towards real application characteristics inspired by the autonomous automobile application, as discussed in the introduction. This has implications for task period, execution time, and CPU/GPU data transmission size. Task sets composed of both CPU-only and GPU-using tasks were randomly generated with these implications in mind. The period of every task was randomly selected from the range $[15ms, 60ms]$, periods common to 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 are regenerated). This yields relatively long average per-task execution times, but we expect GPU-using tasks to have such execution time since current GPUs typically cannot efficiently process short GPU kernels due to I/O bus latencies. Next, between 20% and 30% of tasks within each task set were selected as GPU-using tasks. We have found that this ratio maximized the utilization of both CPUs and GPUs on our platform. For each GPU-using task, a GPU critical section of 80% the length of task execution time was assigned. Of the critical section length, 20% was allocated to transmitting data between the CPU and GPU. This distribution of critical section length and data transmission time is common to many GPU applications, including FFTs and convolutions [12], 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 which first assigned 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.

Generated task sets were executed in LITMUS$^{\text{RT}}$ for a duration of two minutes. Every task set was executed twice, once in LITMUS$^{\text{RT}}$ configured to use klitirqd and once in LITMUS$^{\text{RT}}$ using standard Linux interrupt handling. Time-stamped execution trace logs of scheduling and interrupt events were made from within the kernel using the low-overhead tracing facilities in LITMUS$^{\text{RT}}$. From this data we compared the performance of klitirqd and standard Linux interrupt handling in LITMUS$^{\text{RT}}$ according to the following metrics.

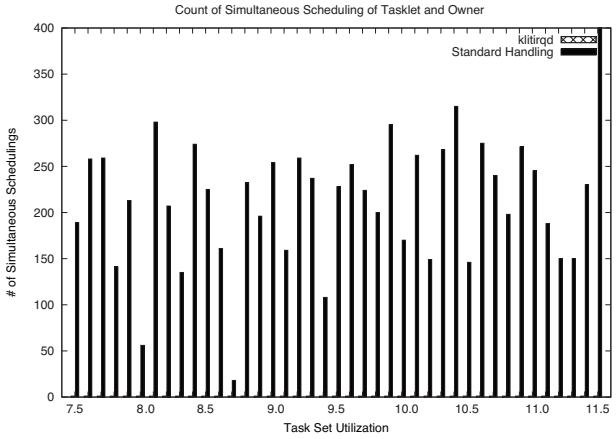

Figure 4. Histogram of concurrent execution events of a tasklet and its owner. No concurrent execution events were observed under klitirqd. There is no apparent trend with regard to task set utilization, though the number of events observed differs greatly among the task sets.

Metrics: Ideally the system should conform to the sporadic task model and not suffer any priority inversions. However, due to implementation limitations in a real operating system this cannot be entirely achieved. To characterize the degree of deviation from this ideal, we assessed each interrupt handling method 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*, which should be mostly short; **(iii)** *number of priority inversions*, which should be low; and **(iv)** *cumulative priority inversion length*, which should also be low.

Results: Fig. 4 shows our measurements for the number of concurrent execution events. No violations of the model were detected when using klitirqd. However, as expected, violations are possible under standard Linux. Most task sets experienced over 100 concurrent execution events during two minutes of execution and up to 200 concurrent execution events occurred frequently. These numbers in themselves may appear to be low, but consider that, though GPU-using tasks operate asynchronously, they do rendezvous with I/O results quickly in our experiments. It is most often the case that a GPU-using task is already blocked by the time its tasklet is

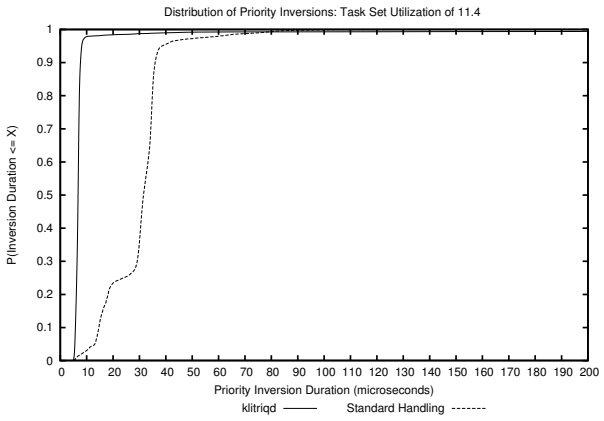

Figure 5. The cumulative distribution of priority inversion durations show that the typical inversion is much shorter under klitirqd than without.

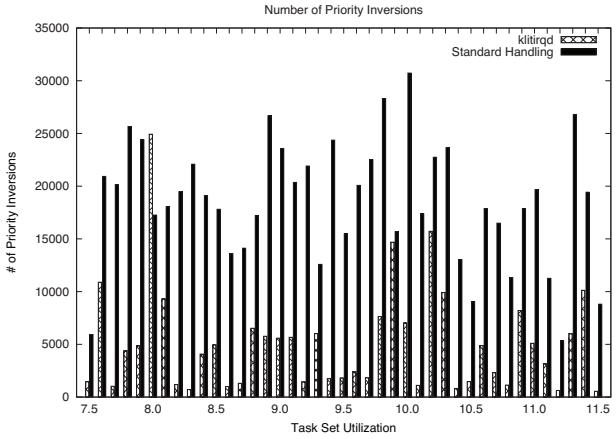

Figure 6. Histogram of detected inversions in two minutes of execution for each task set under both klitirqd and standard Linux interrupt handling. There is no observable trend in task set utilization with this sample size, though variance between task sets is significant.

scheduled. We would expect to see a greater number of concurrent execution events with a greater degree of asynchronous I/O operations. The absence of concurrent execution events under klitirqd shows that is is effective at enforcing conformance to the sporadic task model.

With regard to (ii), while priority inversions cannot be totally eliminated, nevertheless they should be as short as possible. Fig. 5 shows a representative example of the cumulative distribution function (CDF) of priority inversion length under both interrupt handling methods.[10] It shows that a typical priority inversion is much shorter under klitirqd than under Linux interrupt handling. For example, 90% of inversions under klitirqd are shorter than $9\mu s$, whereas the 90th percentile exceeds $30\mu s$ under Linux interrupt handling.

While priority inversions should be as short as possible, the number of inversions is also important because a system that suffers many short inversions may be disrupted by their cumulative effect. Fig. 6 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. This can be seen in Fig. 7, which shows cumulative priority inversion length as a function of maximum priority inversion length. That is, a point (x, y) on a curve in the graph implies that all priority inversion up to length x have a cumulative duration of y. For example, all priority inversion of length up to $x = 50\mu s$ have a cumulative

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

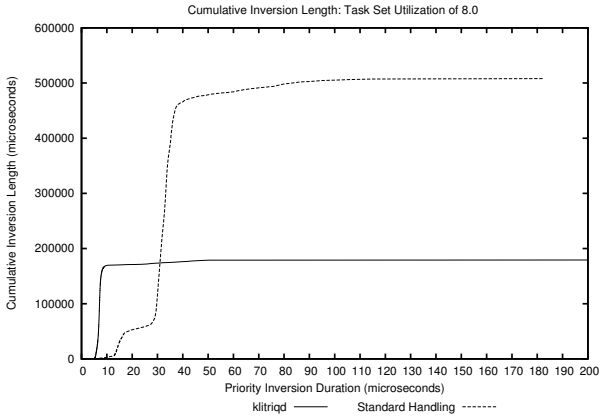

Figure 7. Cumulative priority inversion length as a function of maximum priority inversion length. The total duration of priority inversion is more than twice as large under standard Linux interrupt handling than klitirqd.

duration of only 180,000μs under klitirqd, but more than 475,000μs under Linux interrupt handling. If priority inversions of all lengths are considered, i.e., if $x = 200$ in Fig. 7, then the cumulative inversion length still does not exceed 200,000μs under klitirqd, whereas the cumulative inversion length under standard Linux interrupt handling exceeds 500,000$\mu s \approx 0.5s$. This shows that 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 outperformed

standard Linux interrupt handling in each of the four evaluation metrics. Further, these results demonstrate that *even closed-source drivers*, be they for GPUs or other devices, can still be prevented from causing undue interference. The klitirqd approach to interrupt handling thus lays the foundation for the integration of GPUs into predictable real-time systems.

VI. Conclusion

In this paper we presented a flexible real-time interrupt architecture able to support any JLSP-scheduled non-partitioned multiprocessor platform while also respecting the sporadic task model. This solution was implemented in our real-time Linux-based operating system, LITMUS^{RT}, and we showed that it can be successfully applied to even a closed-source GPU driver, thus allowing for improved real-time characteristics for real-time systems using GPUs. Our implementation was tested through empirical experimentation and shown to greatly reduce the interference caused by GPU interrupts in comparison to standard interrupt handling in Linux.

This paper lays the groundwork for future investigations into real-time platforms using GPUs. In this study, we limited our examination to a cluster-scheduled system. We are preparing to perform a wider study where we will consider the full gamut of partitioned, clustered, and global schedulers, GPU assignment methods, and various locking protocols. A decision in each category affects the available options in the others. For example, the choice of a partitioned scheduler limits which locking protocols may be used for GPU allocation. Such decisions also affect the tightness of real-time analysis techniques. It is not immediately clear which system configurations offer the best real-time properties. This wider study will also include an examination of partitioning heuristics for CPU/GPU platforms in addition to the development of a new k-exclusion protocol for globally scheduled multiprocessors.

References

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

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

[3] CUDA Zone [online]. Available from: http://www.nvidia.com/object/cuda_home_new.html.

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

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

[6] B. Andersson, G. Raravi, and K. Bletsas. Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors. In *31st RTSS*, pages 239–248, 2010.

[7] A. Bastoni, B. Brandenburg, and J. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor real-time schedulers. In *31st RTSS*, pages 14–24, 2010.

[8] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *13th ECRTS*, pages 47–57, 2007.

[9] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *SIGGRAPH '03*, pages 917–924, 2003.

[10] D. Bovet and M Cesati. *Understanding the Linux Kernel*. O'Reilly, 3rd edition, 2006.

[11] B. Brandenburg, H. Leontyev, and J. Anderson. An overview of interrupt accounting techniques for multiprocessor real-time systems. *Journal of Systems Architecture*, 2010.

[12] G. Elliott and J. Anderson. Globally scheduled real-time systems with gpus. In *18th RTNS*, 2010.

[13] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *SIGGRAPH '03*, pages 92–101, 2003.

[14] K. Jeffay and D. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *14th RTSS*, pages 212–221, 1993.

[15] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Resource sharing in GPU-accelerated windowing systems. In *17th RTAS*, pages 191–200, 2011.

[16] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX Annual Technical Conference*, 2011. To appear.

[17] V. Kindratenko and P. Trancoso. Trends in high-performance computing. *Computing in Science Engineering*, 13(3):92–95, 2011.

[18] J. Krüger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH '03*, pages 908–916, 2003.

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

[20] N. Manica, L. Abeni, L. Palopoli, D. Faggioli, and C. Scordino. Schedulable device drivers: Implementation and experimental results. In *6th OSPRT*, 2010.

[21] 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 *Work-in-progress session of 16th ECRTS*, pages 5–8, 2010.

[22] G. Raravi and B. Andersson. Provably good multiprocessor scheduling of implicit-deadline sporadic tasks with resource sharing on two-type heterogeneous platform. Technical Report HURRAY-TR-110106, CISTER-ISEP Research Center, Polytechnic Institute of Porto, 2011.

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

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

[25] S. Thrun. GPU technology conference keynote, day 3, 2010. Available from: http://livesmooth.istreamplanet.com/nvidia100923/.