HAL
open science

Globally Scheduled Real-Time Multiprocessor Systems with GPUs

Glenn A. Elliott, James H. Anderson

To cite this version:

Glenn A. Elliott, James H. Anderson. Globally Scheduled Real-Time Multiprocessor Systems with GPUs. 18th International Conference on Real-Time and Network Systems, Nov 2010, Toulouse, France. pp.197-206. hal-00546957

HAL Id: hal-00546957

https://hal.science/hal-00546957v1

Submitted on 15 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Globally Scheduled Real-Time Multiprocessor Systems with GPUs*

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

Abstract

Graphics processing units, GPUs, are powerful processors that can offer significant performance advantages over traditional CPUs. The last decade has seen rapid advancement in GPU computational power and generality. Recent technologies make it possible to use GPUs as co-processors to the CPU. The performance advantages of GPUs can be great, often outperforming traditional CPUs by orders of magnitude. While the motivations for developing systems with GPUs are clear, little research in the real-time systems field has been done to integrate GPUs into real-time multiprocessor systems. We present two real-time analysis methods, addressing real-world platform constraints, for such an integration into a soft real-time multiprocessor system and show that a GPU can be exploited to achieve greater levels of total system performance.

1 Introduction

The computer hardware industry experienced a rapid growth in the graphics hardware market during this past decade, with fierce competition driving feature development and increased hardware performance. One important advancement during this time was the programmable graphics pipeline. Such pipelines allow program code, which is executed on graphics hardware, to interpret and render graphics data. Soon after its release, the generality of the programmable pipeline was quickly adapted to solve non-graphics-related problems. However, in early approaches, computations had to be transformed into graphics-like problems that a graphics processing unit (GPU) could understand. Recognizing the advantages of general purpose computing on a GPU, language extensions and runtime environments were released by major graphics hardware vendors and software producers to allow general purpose programs to be run on graphics hardware without transformation to graphics-like problems.[1]

Today, GPUs can be used to efficiently handle data-parallel compute-intensive problems and have been utilized in applications such as cryptology [18], supercomputing [3], finance [8], ray-tracing [10], medical imaging [24], video processing [22], and many others.

There are strong motivations for utilizing GPUs in real-time systems. Most importantly, their use can significantly increase computational performance. A review of published research shows that performance increases commonly range from 4x to 20x [4], though increases of up to 1000x are possible in some problem domains [9]. Tasks accelerated by GPUs may execute at higher frequencies or perform more computation per unit time, possibly improving system responsiveness or accuracy.

GPUs can also carry out computations at a fraction of the power needed by traditional CPUs. This is an ideal feature for embedded and cyber-physical systems. Further power efficiency improvements can be expected as processor manufacturers move to integrate GPUs in on-chip designs [1]. On-chip designs may also signify a fundamental architectural shift in commodity processors. Like the shift to multicore, it appears that the availability of a GPU may soon be as common as multicore is today. This further motivates us to investigate the use of GPUs in real-time systems.

A GPU that is used for computation is an additional processor that is interfaced to the host system as an I/O device, even in on-chip architectures. An I/O-interfaced accelerator co-processor, like a GPU or digital signal processor, when used in a real-time system, is unlike a non-accelerator I/O device. In work on real-time systems, the use of non-accelerator devices, such as disks or network interfaces, has been researched extensively [19], with issues such as contention resolution and I/O response time being the primary focus. While these are also concerns for GPUs, the role of the device in the system is different. A real-time system that reads a file from a disk or sends a packet out on a network uses these devices to perform a functional requirement of the system itself. Further, these actions merely cause delays in execution on the CPU; the operations themselves do not affect the actual amount of CPU computation that must be performed. This is not the case for a GPU co-processor as its use accelerates work that could have been carried out by a CPU and does not realize a new functional feature for the system. The performance of a real-time system with a GPU co-processor is dependent upon three inter-related design aspects: how traditional device issues (such as contention) are resolved; the

*Work supported by AT&T and IBM Corps.; NSF grants CNS 0834270 and CNS 0834132; ARO grant W911NF-09-1-0535; and AFOSR grant FA 9550-09-1-0549.

[1]Notable platforms include the Compute Unified Device Architecture (CUDA) from Nvidia [5], Stream from AMD/ATI [2], OpenCL from Apple and the Khronos Group [7], and DirectCompute from Microsoft [6].

extent to which the GPU is utilized; and the gains in CPU availability achieved by offloading work onto the GPU.

In this paper, we consider the use of GPUs in *soft* real-time multiprocessor systems, where processing deadlines may be missed but deadline tardiness must be bounded. Our focus on soft real-time systems is partially motivated by the prevalence of application domains where soft real-time processing is adequate. Such a focus is further motivated by fundamental limitations that negatively impact *hard* real-time system design on multiprocessors. In the multiprocessor case, effective timing analysis tools to compute worst-case execution times are lacking due to hardware complexities such as shared caches. Also, in the hard real-time case, the use of non-optimal scheduling algorithms can result in significant utilization loss when checking schedulability, while optimal algorithms have high runtime overheads. In contrast, many global scheduling algorithms are capable of ensuring bounded deadline tardiness in soft real-time systems with no utilization loss and with acceptable runtime overheads. One such algorithm is the *global earliest-deadline-first* (G-EDF) algorithm [16].

As G-EDF can be applied to ensure bounded tardiness with no utilization loss in systems without a GPU, we consider it as a candidate scheduler for GPU-enabled systems. We note however, that existing G-EDF analysis has its limitations. Specifically, most analysis is what we call *suspension-oblivious* in that it treats any self-suspension (be it blocking to obtain a lock or waiting time to complete an I/O transaction) as execution time on a CPU. This implies that the interval of time a task suspends from a CPU to execute on a GPU must also be charged as execution on a CPU. Under these conditions, it appears that a GPU may be useless if work cannot be offloaded from the CPUs. However, a GPU is an *accelerator* co-processor; it can perform more work per unit time than can be done by a CPU. Therefore, there may still be benefits to using a GPU even if CPU execution charges must mask suspensions. In this paper, we determine the usefulness of a GPU in a soft real-time multiprocessor system by answering the following question: How much faster than a CPU must a GPU be to overcome suspension-oblivious penalties and schedule more work than a CPU-only system?

To date, little formal real-time analysis has been done to integrate graphics hardware into real-time systems, and this work, to our knowledge, is the first to investigate the integration of a GPU into a soft real-time multiprocessor environment. The contributions of this paper are as follows. We first profile common usage patterns for GPUs and explore the constraints imposed by both the graphics hardware architecture and the associated software drivers. We then present a real-time task model that is used to analyze the widely-available platform of a four-CPU, single-GPU system. With this model in mind, we propose two real-time analysis methods, which we call the *Shared Resource Method* and the *Container Method*, with the goal of providing predictable system behavior while maximizing processing capabilities and addressing real-world platform constraints. We compare these methods through schedulability experiments to determine when benefits are realized from using a GPU. Additionally, we present an implementation-oriented study that was conducted to confirm the necessity of real-time controls over a GPU in an actual real-time operating system environment. The paper concludes with a discussion of other avenues for possible real-time analysis methods and considers other problems presented by the integration of CPUs and GPUs.

Problem Type	**Average**
Eigenvalue Computation	
512x512	5.88ms
2048x2048	22.87ms
4096x4096	50.92ms
2D Convolution	
512x512	1.61ms
2048x2048	58.17ms
4096x4096	141.64ms
2D Fluid Simulation	
512x512	170μs
2048x2048	290μs
N-Body Simulation	
1024 particles	210μs

Table 1: Observed GPU kernel execution times on a GTX-285 Nvidia graphics card.

2 Usage Patterns and Platform Constraints

It is worthwhile to first examine the usage patterns of GPUs in general purpose applications as well as the constraints imposed by hardware and software architectures before developing any real-time analysis approach. As we shall see, these real-world characteristics cannot be ignored in a holistic system point-of-view. We begin by examining GPU execution environments.

The execution time of a GPU program, called a *kernel*, varies from application to application and can be relatively long. To determine likely execution-time ranges, we profiled sample programs from Nvidia's CUDA SDK on a GTX-285 Nvidia graphics card. We found that n-body simulations run on the order of 10–100μs per iteration on average while problems involving large matrix calculations (multiplication, eigenvalues, etc.) can take 10–200ms on average. Table 1 contains a summary of observed GPU execution times for several basic operations.

The results in Table 1 indicate that relatively long GPU access times are common. Additionally, the I/O-based interface to a GPU co-processor introduces several additional unique constraints that need to be considered. First, a GPU cannot access main memory directly, thus making the memory between the host and GPU non-coherent between synchronization points. Memory is transferred over the bus (PCIe) explicitly or through automated DMA to explicitly-allocated blocks of main memory (in integrated

graphics solutions, the GPU uses a partitioned section of main memory, but the architectural abstractions remain). Second, kernel execution on a GPU is non-preemptive: execution of the kernel must be run to completion before another kernel may begin. Third, kernels may not execute concurrently on a GPU even if many of the GPU's parallel sub-processors are unused.[2] Finally, a GPU is not a system programmable device in the sense that a general OS cannot schedule or otherwise control a GPU. Instead, a driver in the OS manages the GPU. This last constraint bears additional explanation.

At runtime, the host-side program sends data to the GPU, invokes a GPU program, and then waits for results. While this model looks much like a remote procedure call, unlike a remote RPC-accessible system, the GPU is unable to schedule its own workloads. Instead, the host-side driver manages all data transfers to and from the device, triggers kernel invocations, and handles the associated interrupts. Furthermore, this driver is closed-source since the vendor is unwilling to disclose proprietary information regarding the internal operations of the GPU. Also, driver properties may change from vendor to vendor, GPU to GPU, and even from driver version to version. Since even soft real-time systems require provable analysis, the uncertain behaviors of the driver force integration solutions to treat it as a black box.

Unknown driver behaviors are not merely speculative but are a real concern. For example, we found that a recent Nvidia CUDA driver may induce uncontrollable busy-waiting when the GPU is under contention, despite all runtime environment controls to the contrary.[3] Further complicating matters, the driver does not provide predictable real-time synchronization, an issue that receives more attention in Sec. 4.1. Serious behavioral deficiencies of the driver in real-time environments are further investigated in Sec. 6.

3 Task Model and Scheduling Algorithms

Real-time analysis offers several methods for describing the workload of a real-time system. This paper analyzes mixed task sets of CPU-only and GPU-using tasks with the synchronous implicit-deadline periodic task model as it adequately describes common workloads and has well-understood analytical properties.

A synchronous implicit-deadline periodic *task set*, T, consists of as a set of recurrent *tasks*, T_i, some of which may access a GPU. We let $G(T)$ denote the set of GPU-using tasks in T. Each task is described by three parameters: its *period*, p_i, which measures the separation between task recurrences (known as *jobs*); its *worst-case CPU execution time*, e_i, which bounds the amount of CPU processing time a job must receive before completing; and

[2]Nvidia's new Fermi architecture allows limited simultaneous execution of kernels as long as these kernels are sourced from the same host-side context/thread.

[3]See Appendix A for details in the extended online version of this paper found at `http://www.cs.unc.edu/~anderson/papers.html`.

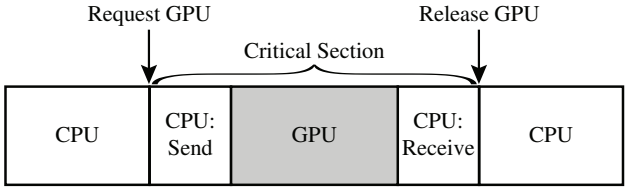

Figure 1: Execution phases of a GPU-using job.

its *worst-case GPU execution time*, s_i, which bounds the amount of GPU processing time required by one of its jobs. This last parameter captures the interval of time between a kernel invocation and the signaling of its completion to the driver. Like worst-case CPU execution, this parameter is unique to each task and is dominated by the GPU kernel execution time plus lesser communication latencies. Preliminary work [23] has been done to upper-bound GPU kernel execution time, though empirical tests are sufficient for many soft real-time systems. For tasks that do not use the GPU, $s_i = 0$. The *utilization* of task T_i is given by $u_i = e_i/p_i$ and the *system utilization* is given by $U = \sum u_i$.

As stated, our goal is to maximize system utilization while supporting soft real-time constraints. Unfortunately, due to a GPU's I/O-based interface, techniques for heterogeneous systems [11, 17, 12] do not immediately apply. However, as noted earlier, previous work [16] has shown that G-EDF can ensure bounded tardiness in ordinary multiprocessor systems (without a GPU) without system utilization constraints (provided the system is not overutilized). Thus, it is the primary scheduling algorithm considered in this paper.[4] G-EDF is a *global* scheduler, meaning that jobs share a single ready queue and can migrate between processors. G-EDF prioritizes work by job deadline, scheduling jobs with the earliest deadlines first.

4 Analysis Methods

We consider two methods for analyzing mixed task sets of CPU-only and GPU-using tasks on a multiprocessor system with a single GPU: the *Shared Resource Method* and the *Container Method*. Fundamental differences between these methods stem from how GPU execution time is modeled and how potential graphics hardware driver behaviors are managed.

4.1 Shared Resource Method

It is natural to view a GPU as a computational resource shared by the CPUs of a multiprocessor system. This is the approach taken by the Shared Resource Method (SRM), which treats the GPU as a globally-shared resource protected by a real-time semaphore.

The execution of a GPU-using job goes through several phases. In the first phase, the job executes purely on the

[4]Some have recently speculated [20] that the *earliest-deadline-zero-laxity* (EDZL) algorithm may be better suited to accounting for self-suspensions, though actionable results have yet to be presented, so better suspension accounting remains an open problem.

CPU. In the next phase, the job sends data to the GPU 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. Finally, the job resumes execution on the CPU and receives kernel execution results when signaled by the GPU. 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. 1.

We can remove the GPU driver from resource-arbitration decisions and create a suitable model for real-time analysis through the use of a real-time semaphore protocol. Contention for a GPU may occur when a job attempts to communicate with it. We resolve this contention with a synchronization point between the first and second phases to provide mutual exclusion through the end of the fourth phase; this interval is called a *critical section* and denoted for each task T_i by cs_i. This approach ensures that the driver only services one job at a time, which eliminates the need for knowing how the driver (which, again, is closed-source) manages concurrent GPU requests.

We may consider several real-time multiprocessor locking protocols to protect the GPU critical section. Such a protocol should have several properties. First, it must allow blocked jobs to suspend since critical-section lengths may be very long (recall Table 1). A spin lock would consume far too much CPU time. Second, the protocol must support priority inheritance so blocking times can be bounded. Finally, the protocol need not support critical-section nesting or deadlock prevention since GPU-using tasks only access one GPU. Both the "long" variant of the *Flexible Multiprocessor Locking Protocol* (FMLP-Long) [13] and the more recent global *O(m) Locking Protocol* (OMLP) [14] fit these requirements. Neither protocol is strictly better than the other for all task sets since priority-inversion-based blocking (per lock access), denoted by b_i, is *O(n)* under FMLP-Long and *O(m)* under the OMLP, where n is the number of tasks and m is the number of CPUs. Thus, we allow the SRM to use whichever protocol yields a schedulable task set.

The FMLP-Long uses a single FIFO job queue for each semaphore, and GPU requests are serviced in a first-come first-serve order. The job at the head of the FIFO queue is the lock holder. A job, J_i, of task $T_i \in G(T)$ may be blocked by one job from the remaining GPU-using tasks. Formally,

$$b_i = \sum_{G(T)\setminus\{T_i\}} cs_k. \tag{1}$$

The global OMLP uses two job queues for each semaphore: FQ, a FIFO queue of length at most m; and PQ, a priority queue (ordered by job priority). The lock holder is at the head of FQ. Blocked jobs enqueue on FQ if FQ is not full and on PQ, otherwise. Jobs are dequeued from PQ onto FQ as jobs leave FQ. Any job acquiring an OMLP lock may be blocked by at most $2(m-1)$ lower-

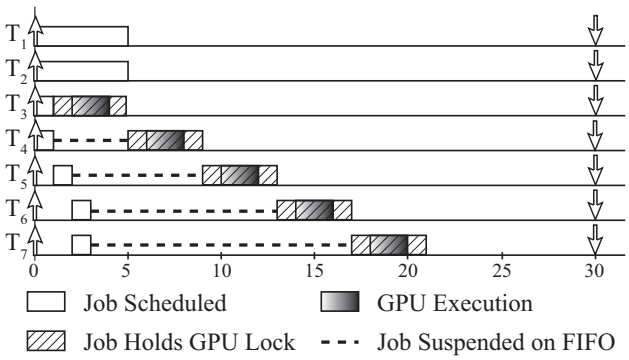

Figure 2: Schedule for the example task set under the SRM on a four-processor single-GPU system.

priority jobs. Let A be the set of jobs generated by any $T_k \in G(T) \setminus \{T_i\}$ that may contend with J_i for the GPU. Let A_{max} be the $2(m-1)$ jobs in A with the longest critical sections. The blocking time for task $T_i \in G(T)$ is given by the formula

$$b_i = \sum_{J_k \in A_{max}} cs_k. \tag{2}$$

Soft schedulability under the SRM is determined by the following two conditions. First,

$$e_i + s_i + b_i \leq p_i \tag{3}$$

is required by the tardiness analysis for G-EDF [16]. Second, the condition

$$U = \sum (e_i + s_i + b_i)/p_i \leq m \tag{4}$$

must hold. This is the soft G-EDF schedulability condition required by [16] to ensure bounded tardiness. Like all suspension-oblivious tests, we must analytically treat suspension due to both blocking and GPU execution as execution on the CPU. Note that no schedulability test is required for the GPU co-processor since a job's mutually exclusive GPU execution is masked by fictitious CPU execution. Still, the suspension-oblivious nature of this test is a limiting characteristic as is seen in Sec. 5.

Example. Consider a mixed task set with two CPU-only tasks with task parameters $(p_i = 30, e_i = 5, s_i = 0)$ and five GPU-using tasks with parameters $(p_i = 30, e_i = 3, s_i = 2, cs_i = 4)$ to be scheduled on a four-CPU system with a single GPU. The CPU-only tasks trivially satisfy Ineqs. (3). The FMLP-Long is best suited for this task set and the blocking term for every GPU-using task is $\sum cs_k = 16$ as computed by Eq. (1). Tasks in $G(T)$ satisfy Ineq. (3) since $3+2+16 = 21 \leq 30$. Ineq. (4) also holds since $U = 2 \cdot (5/30) + 5 \cdot ((3+2+16)/30) \approx 3.83 \leq 4$. Therefore, the task set is schedulable under the SRM.

A schedule for this task set is depicted in Fig. 2. T_1 and T_2 are the CPU-only tasks. Observe that the final job completes at time $t = 21$, well before its deadline. The computed system utilization of approximately 3.83 is quite

close to the upper bound of 4.0 used in Ineq. (4), which suggests a heavily-utilized system. However, the schedule in Fig. 2 shows that the suspension-oblivious analysis is quite pessimistic (mostly due to blocking-term accounting) given that the system is idle for much of the time. In fact, only one CPU is utilized after $t = 5$. The performance of the GPU must overcome these suspension-oblivious penalties if it is to be a worthwhile addition to a real-time multiprocessor system.

4.2 Container Method

The SRM may be overly pessimistic from a schedulability perspective due to heavy utilization penalties arising from the blocking terms introduced by the use of a multiprocessor locking protocol. Methods that lessen such penalties may offer tighter analysis. The Container Method (CM) is one such approach.

In many cases, a *single* GPU will limit the total actual CPU utilization (where suspension effects are ignored) of tasks in $G(T)$. For example, if all of the tasks in $G(T)$ perform most of their processing on the GPU, then the total actual CPU utilization of these tasks will be much less than 1.0 when the GPU is fully utilized. In this case, if we inflate each task's CPU execution time by its GPU execution time, as we do for suspension-oblivious analysis, then the actual GPU utilization and total suspension-oblivious CPU utilization will both be close to 1.0. This fact inspires the CM, which avoids heavy suspension-oblivious penalties by removing contention for the GPU resource through the isolation of $G(T)$ to a single (logical) processor.

It was shown in [21] that bandwidth reservations, or *containers*, may be used to support soft real-time guarantees in multiprocessor systems. In a container-based system, a task set is organized into a hierarchical collection of containers. Each container may hold tasks or child containers. A container C is assigned an *execution bandwidth*, $w(C)$, equal to the sum of the utilizations and bandwidths of its child tasks and containers, respectively.

For our GPU-enabled multiprocessor system, we place all CPU-only tasks in a root container, H, and all tasks of $G(T)$ in a child container G of H. As before, suspensions are treated as execution time and contribute to task utilizations. A container decomposition of the example task set given in Sec. 4.1 is shown in Fig. 3. Observe that the tasks in $G(T)$ are isolated in container G with a bandwidth of $5/6$, the total suspension-oblivious utilization of the tasks in $G(T)$. Container G and the CPU-only tasks are contained within H, which has a bandwidth of $7/6$.

Containers provide *temporal isolation* by hierarchically allocating execution time to contained tasks and containers. If each container schedules its contained tasks and containers using a *window-constrained* scheduling algorithm,[5] such as the G-EDF, then bounded tardiness can be ensured with no utilization loss [21]. The CM exploits both

[5] A *window-constrained* scheduling algorithm prioritizes a job by an arbitrary time point contained within an interval window that also contains the job's release and deadline.

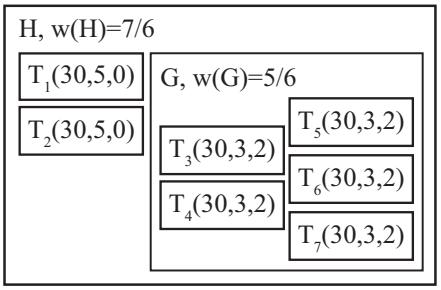

Figure 3: Container decomposition of an example mixed task set.

this and the ability to apply different schedulers to subsets of jobs.

We schedule the children of H with G-EDF and schedule the children of G with uniprocessor FIFO, which is a window-constrained algorithm that prioritizes jobs by release time. All GPU contention is avoided through the use of the FIFO scheduler, which eliminates preemptions, assuming jobs do not self-suspend or self-suspensions are analytically treated as CPU execution. This ensures that the GPU is always available to the highest-priority GPU-using job. Note, in implementation it is not necessary for G to suspend or idly consume CPU resources while the GPU is in use. Instead, G may schedule other contained jobs, provided that the GPU critical sections are protected by a simple release-ordered semaphore. This ensures that the highest-priority job may be scheduled immediately, without conflict, when it is ready to run. This work-conserving approach would reduce observed tardiness, though this is not captured by our analysis here.

Soft schedulability of a task set under the CM is determined by the following conditions. First,

$$w(G) \leq 1 \tag{5}$$

is required to ensure that G is schedulable with bounded tardiness on a uniprocessor. Second,

$$w(G) + \sum_{T_i \notin G(T)} u_i \leq m \tag{6}$$

must also hold. This condition ensures that the root container can be scheduled by G-EDF on m CPUs with bounded tardiness.

Example. Consider the same mixed task set from Sec. 4.1. Ineq. (5) is satisfied since the container bandwidth is $w(G) = 5 \cdot (5/30) \approx 0.83 \leq 1$. Ineq. (6) also holds as $U = 2 \cdot (5/30) + 5 \cdot (5/30) \approx 1.16 \leq 4$. Therefore, the task set is schedulable under the CM. A schedule for this task set is depicted in Fig. 4. T_1 and T_2 are the CPU-only tasks. Note that the final job completes at time $t = 25$.

The SRM enforces more permissive constraints on the GPU while the CM enforces more permissive constraints on the CPUs. This trade-off is reflected in both the schedulability tests and example schedules of these methods. Due

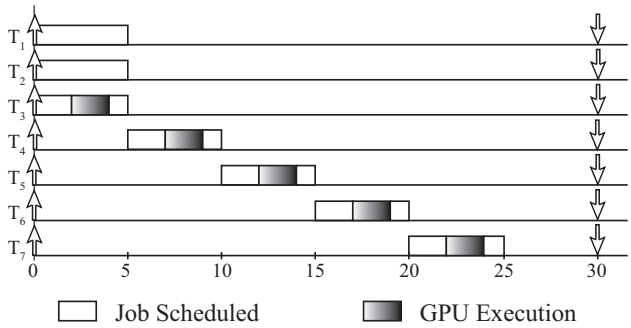

Figure 4: Schedule for the example task set under the CM on a four-processor single-GPU system.

to the mutually exclusive ownership of the GPU, there may exist only one job within its critical section ready to be scheduled on any CPU at any given time. This implies that system GPU utilization under the SRM can be bounded by the formula

$$\sum_{T_i \in G(T)} cs_i/p_i \leq 1 \tag{7}$$

for schedulable task sets. This measure includes CPU execution time within critical sections since entire critical sections must execute in sequence. Comparing the SRM's and the CM's measures of GPU utilization for the previous example, we find the SRM's GPU utilization (Ineq. (7)) is approximately 0.67 while the CM's (Ineq. (5)) is approximately 0.83; the SRM's CPU constraint (Ineq. (4)) is approximately 3.83 while the CM's (Ineq. (6)) is approximately 1.16. Such trade-offs are not merely limited to the tightness of analytical bounds, but are actually reflected in task set schedules, as can be observed in Figs. 2 and 4. While the CM enforces more permissive CPU utilization constraints, the GPU-using jobs complete later under the CM. This corresponds directly to the CM's higher measure of GPU utilization.

5 Evaluation

We carried out SRM- and CM-related schedulability experiments to answer the question raised at the beginning of this paper: How much faster than a CPU must a GPU be to overcome suspension-oblivious penalties and schedule more work than a CPU-only system?

5.1 Experimental Setup

To better understand the schedulability of mixed task sets, we randomly generated task sets using three task utilization intervals, three period intervals, three GPU usage patterns, and ten GPU task percentages. *Utilization intervals* determine the range of utilization for individual tasks and were $[0.01, 0.1]$ (*light*), $[0.1, 0.4]$ (*medium*), and $[0.5, 0.9]$ (*heavy*). *Period intervals* determine the range of task periods for individual tasks and were $[3ms, 33ms]$, $[15ms, 60ms]$, and $[50ms, 250ms]$. The *GPU usage pattern* determines how much of the execution time of each GPU-using task is GPU execution time; 25%, 50%, and 75% were used, in line with common CPU/GPU workload distributions.[6] Finally, the *GPU task percentage* is the ratio of GPU-using tasks to the total number of tasks; increments of 10% were used to test GPU task percentages from 0% to 100%. A schedulability experiment scenario was defined by any permutation of these four parameters, yielding a total of 270 scenarios.

We generated random task sets for each scenario in the following manner. First, we selected a total system utilization cap uniformly in the interval $(0, 4]$, capturing the possible system utilizations of a platform with four CPUs and a single GPU when suspension-oblivious analysis is used. We then generated tasks by making selections uniformly from the utilization interval and period interval according to the given scenario. We derived execution times from these selections. We added these tasks to a task set until the set's total utilization exceeded the utilization cap, at which point the last-generated task was discarded. Next, we selected tasks for $G(T)$ from the task set; we determined the number of GPU-using tasks by the GPU task percentage of the scenario. We then assigned the same GPU usage pattern to each task in $G(T)$ according to the scenario. Additionally, we assumed that time spent communicating with the GPU increases with execution time and assessed a GPU communication cost, ε, of 5% of task execution. Thus, a GPU-using task T_i's critical-section length is $(x+\varepsilon) \cdot e_i$ where x denotes the GPU usage pattern. We made cursory tests of CPU and GPU utilization to ensure that the CPUs and GPU were not implicitly overutilized. Finally, we discarded task sets with only one GPU-using task since this case is uninteresting as the GPU does not require resource arbitration. We tested a total of 1,000,000 task sets for each scenario.

We also generated equivalent CPU-only task sets to help answer the question of when a multiprocessor system can benefit from a GPU co-processor. We transformed the mixed task sets into CPU-only equivalents by modifying the execution time of the tasks in $G(T)$ with the formula $e'_i = e_i + c \cdot s_i - \varepsilon$ where c is a positive constant scaling factor denoting the GPU speed-up over the CPU. Each task set was tested with c equal to two, four, eight, and 16. A transformed CPU-only task set is schedulable (has bounded tardiness) if $e'_i \leq p_i$ for all tasks and $U' \leq m$, where U' is the system utilization of the modified task set.

We tested the SRM and the CM according to the schedulability conditions already described in Sec. 4.

5.2 Results

A representative subset of graphs resulting from our schedulability experiments is presented in this section to show the schedulability properties of the SRM and the CM

[6] Common workload profiles were solicited from research groups at UNC that frequently make use of CUDA. A poll was also informally taken at the Nvidia CUDA online forums.

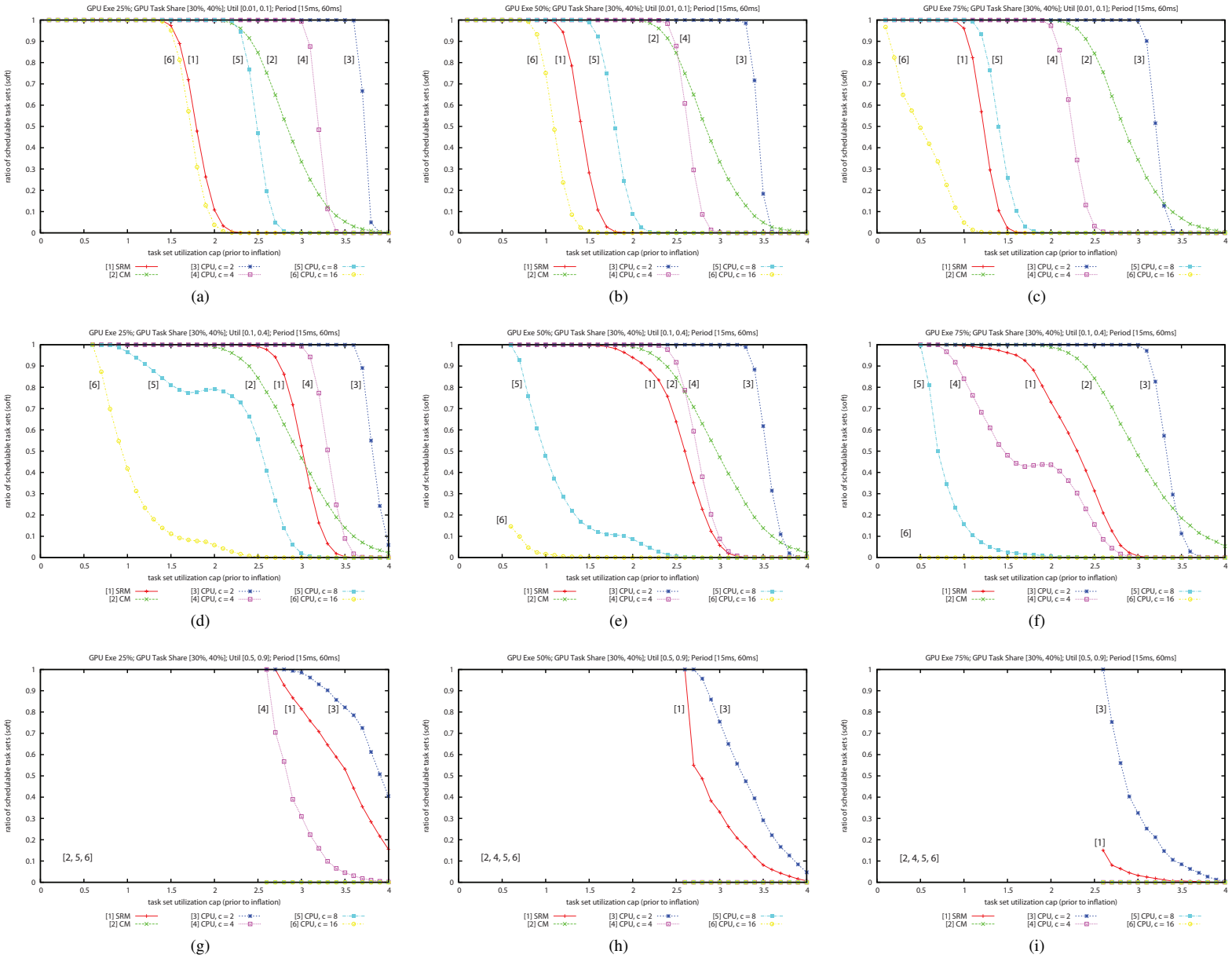

Figure 5: Per-task utilization increase from top to bottom. GPU execution percentage increases from left to right.

and to demonstrate their advantages over pure CPU-only systems.[7] We are limited by page constraints from presenting results for all 270 scenarios, though additional graphs are available in an extended version of the paper online. The presented subset of scenarios was selected because they best utilized both the GPU and the CPUs, illustrating seen trends more broadly.

Schedulability results for task sets with a GPU task percentage ranging from 30% to 40% are shown in Fig. 5. The graphs are organized to show trends as functions of per-task utilization (down the columns) and GPU usage pattern (across the rows). The rows correspond to light, medium, and heavy per-task utilization intervals. Likewise, the columns correspond to GPU usage patterns of 25%, 50%, and 75%.

Observation 1. *A GPU usually allows a four-CPU system to schedule more work than CPU equivalents if the GPU is four times faster than the CPU.* Both the SRM and the CM outperform the CPU equivalents when $c \geq 8$ in all cases. A speed-up of four is all that is necessary in many cases as seen in insets (b), (c), (e), and (f) of Fig. 5. This suggests that the practical use of a GPU in a four-CPU real-time system is possible since speed-ups greater than eight are common. Indeed, this answers (in the context of this experimental framework) our original question of how fast a GPU must be to overcome penalties from suspension-oblivious analysis.

Observation 2. *The Container Method frequently offers better schedulability than the Shared Resource Method.* The CM can often schedule task sets the SRM cannot as illustrated by the large differences in the schedulability curves seen in insets (a), (b), (c), (e), and (f) of Fig. 5. In the SRM, each task in $G(T)$ incurs an execution penalty up to the length of six critical sections (recall that Eq. (2) includes up to $2(m-1)$ terms). If the constraint given by Ineq. (3) is not violated, then there is still a good chance that the constraint of Ineq. (4) will be, especially at higher system utilizations. The CM clearly benefits from avoiding the inclusion of blocking terms in its schedulability analysis, despite the fact that its GPU utilization condition (Ineq. (5)) includes more CPU execution time.

Observation 3. *The Shared Resource Method improves as per-task utilization increases.* Observe how the schedulability curve for the SRM improves from inset (a) to (d) to (g) in Fig. 5. For example, roughly 50% of task sets are schedulable at system utilizations 2.0, 3.0, and 3.5 in insets (a), (d), and (g), respectively. The SRM benefits from increased per-task utilization since it reduces the total number of tasks in a given task set and hence also reduces the number of tasks in $G(T)$. This improves schedulability since fewer GPU-using tasks result in smaller cumulative blocking-term penalties.

[7] Please note that some graphs appear to be missing data points at lower and upper system utilization ranges. This is caused by the occasional inability to generate task sets meeting particular scenario constraints.

Observation 4. *The Container Method is largely unaffected by either GPU utilization or per-task utilization.* The schedulability curves for the CM in Fig. 5 are all very similar. This is due to two aspects of the CM. First, container bandwidth is a function of the cumulative execution time of each task in $G(T)$, or $e_i + s_i$. As s_i increases across the rows of Fig. 5, e_i decreases by an equal amount, so container bandwidth remains constant. The CM is also resistive to changes in per-task utilization as its schedulability, unlike the SRM, is not dependent on the number of tasks in $G(T)$, but only on the total suspension-oblivious utilization of $G(T)$.

Observation 5. *The Container Method cannot schedule task sets with per-task utilizations greater than 0.5.* The CM cannot schedule any heavy task set due to its strict container bandwidth constraints as seen in the insets (g), (h), and (i) of Fig. 5. Recall that the condition given by Ineq. (5) must be met for a task set to be schedulable under the CM. A heavy task set is schedulable under the CM only if $G(T)$ contains two tasks with utilizations equal to 0.5. However, the occurrence of this case is highly improbable since utilizations are chosen at random.

Observation 6. *The Container Method is best suited for systems with medium or light per-task utilizations.* While schedulability may vary across the CM's gradually sloping curves, it frequently offers better schedulability than the SRM (Obs. 2). Further, in medium and light cases where the SRM offers better schedulability than the CM (Fig. 5 (inset (d))), the CM is still competitive.

6 Implementation

We implemented the SRM with the OMLP in LITMUS^{RT} (described in detail in [15]), a UNC-produced Linux-based testbed for real-time schedulers. We did this to both evaluate the practical performance characteristics of our solution and, more importantly, to show that *unguarded GPU device driver access is not viable for a real-time system*—some real-time control is necessary.

We generated synthetic workloads in the same fashion as in Sec. 5.1 and ran them on our test platform, an Intel Core i7 quad-core system with a Nvidia GTX-295 graphics card.[8] The system CPUs operated at 2.67GHz with an 8MB shared cache. The Nvidia 190.53 64-bit Linux proprietary driver was used on the platform without modification. Nvidia's CUDA 2.3 SDK provided the CUDA runtime environment. In all tests, no display of any kind was used. Thus, the GPU was used exclusively for CUDA computations without interference from other applications.

We executed synthetic task sets with G-EDF scheduling for a duration of 2.5 minutes under two scenarios: one with the SRM and one without. We made measurements for response time and tardiness. A total of 400 task sets were tested under each scenario.

[8] The GTX-295 actually provides two independent GPUs on a single card, though only one GPU was used in this work.

	Avg. Resp. Time		Avg. Tardiness	
Category	SRM	Driver	SRM	Driver
Easy	25.00%	24.95%	0.02%	0.00%
Difficult	29.33%	34.89%	0.17%	4.64%
Unable	92.79%	134.50%	91.97%	133.50%

Table 2: Response time and tardiness statistics for the SRM and unguarded driver. Smaller values are better.

A summary of our findings for medium-utilization task sets is shown in Table 2. A large amount of data was generated in these tests and cannot be presented in detail due to page constraints, so only high-level statistics are shown. We use the ratios *response time/task period* and *tardiness/task period* to interpret our data as this allows measurements involving task sets with different period ranges to be compared.

The executed task sets are organized into *easy*-, *difficult*-, and *unable-to-schedule* categories. Easy-to-schedule task sets are those that are deemed schedulable by the theoretical analysis of Sec. 4.1. Difficult-to-schedule task sets are those for which theoretical analysis was unable to determine schedulability, but the observed tardiness of any job of task T_i never exceeded p_i. While 2.5 minutes of execution cannot *prove* schedulability, it indicates that the task set may be schedulable. We make this assumption here. Unable-to-schedule task sets are those that could not be successfully scheduled (tardiness exceeded p_i) by the implementation—no unable-to-schedule task sets were ever deemed schedulable by the theoretical analysis.

Observation 1. *The Shared Resource Method offers real-time guarantees with little or no observed cost.* The SRM allows GPU-using tasks to be scheduled with real-time guarantees through the use of predictable locking mechanisms, though performance is slightly hindered for easy-to-schedule task sets (as seen in Table 2, average response time and average tardiness are slightly better for the driver in this case). The marginally better performance of the unguarded driver for easy-to-schedule task sets comes at the expense of significantly increased CPU utilization since the driver reduces latency through busy-wait spinning. It is likely that the CPUs could potentially handle additional processing in such cases. Nevertheless, the driver's spinning and lack of priority inheritance becomes a liability in task sets where resources are more taxed as seen in the greater ratios of the difficult- and unable-to-schedule categories.

Observation 2. *The Shared Resource Method is superior at controlling job tardiness.* G-EDF scheduling distributes tardiness relatively equally across all tasks in both the SRM and unguarded driver scenarios. However, tardiness growth is much better controlled under the SRM as can be observed in Fig. 6, which depicts the growth in tardiness for a task from an unable-to-schedule task set. This control over tardiness is also exhibited in nine difficult-to-schedule task sets the SRM was able to schedule (keeping tardiness close to zero) that the unprotected driver could not. A real-time mechanism such as the SRM is necessary in a GPU-enabled real-time system that may occasionally become overutilized.

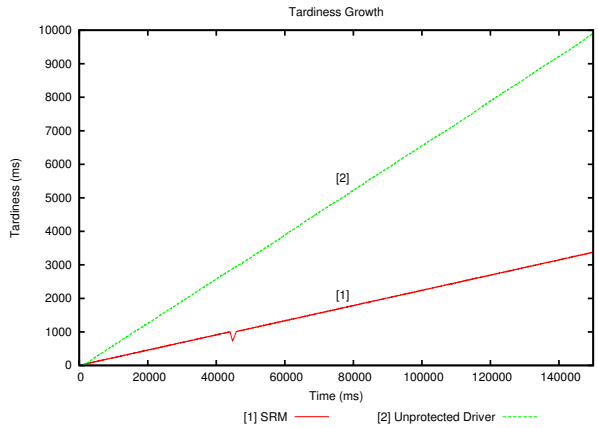

Figure 6: Growth of tardiness growth of a task from an unable-to-schedule task set. The SRM exhibits superior control over tardiness.

7 Future Work

In future work, we intend to investigate how the SRM may be improved to support the exploitation of asynchronous memory transfers. Discrete graphics cards support the ability for graphics hardware to send and receive data to one task while the GPU itself performs computations for another. This allows for the masking of communication latencies in a pipelined manner. The current treatment of critical sections precludes the use of such a mechanism.

Another direction we may pursue is support for multi-GPU platforms. Platforms with many GPUs (sometimes heterogeneous) are already available at consumer prices. It is feasible to design a system that could dynamically choose to execute a particular task or job on one of multiple CPUs or on a variety of GPUs. If a SRM-like approach is taken, not only could the locks protecting GPUs become k-exclusion locks,[9] thus adding an extra dimension of complexity, but execution times of tasks could vary depending upon where it is scheduled if GPUs with varying capabilities are used.

Finally, we plan to perform in-depth empirical analysis to determine the gap between the theoretical schedulability results presented in this paper and apparent schedulability in a real system. Rigorous empirical tests should further clarify when a GPU is beneficial in "real world" real-time systems.

[9] k-exclusion locks protect a resource or resource pool, allowing up to k simultaneous accesses.

8 Conclusion

Recent advances in graphics hardware are enabling the acceleration of computations traditionally carried out on CPUs. The use of such hardware in a real-time system may allow workloads to be supported that are too computationally intensive for CPU-only systems, while also benefiting from reduced power consumption. Through the consideration of current architectural constraints, this paper has presented two methods for integrating GPUs into soft real-time multiprocessor systems: the Shared Resource Method, and the Container Method. Schedulability experiments were presented that assess the schedulability characteristics of each. Both solutions were able to schedule greater computational workloads than pure CPU systems in common cases. The Shared Resource Method was also evaluated through implementation and exhibited superior runtime characteristics in terms of schedulability and efficient resource utilization in comparison to a similar system that is oblivious to GPU hardware and device driver behaviors.

References

[1] AMD Fusion Family of APUs. Available from: `http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf` [cited September 28, 2010].

[2] ATI Stream Technology. Available from: `http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx` [cited September 28, 2010].

[3] China's new nebulae supercomputer is no. 2. Available from: `http://www.top500.org/lists/2010/06/press-release` [cited September 28, 2010].

[4] CUDA community showcase. Available from: `http://www.nvidia.com/object/cuda_apps_flash_new.html` [cited September 28, 2010].

[5] CUDA Zone. Available from: `http://www.nvidia.com/object/cuda_home_new.html` [cited September 28, 2010].

[6] Microsoft DirectX. Available from: `http://www.microsoft.com/games/en-US/aboutGFW/pages/directx.aspx` [cited September 28, 2010].

[7] OpenCL. Available from: `http://www.khronos.org/opencl/` [cited September 28, 2010].

[8] Parallel computing with SciFinance. Available from: `http://www.scicomp.com/parallel_computing/SciComp_NVIDIA_CUDA_OpenMP.pdf` [cited September 28, 2010].

[9] G. Abhijeet and T. Ioane Muni. GPU based sparse grid technique for solving multidimensional options pricing PDEs. In *Proceedings of the 2nd Workshop on High Performance Computational Finance*, pages 1–9, November 2009.

[10] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics*, pages 145–149, August 2009.

[11] S. Baruah. Scheduling periodic tasks on uniform processors. In *Proceedings of the EuroMicro Conference on Real-time Systems*, pages 7–14, June 2000.

[12] S. Baruah. Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms. In *Proceedings of the 25th IEEE Real-Time Systems Symposium*, pages 37–46, December 2004.

[13] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57, August 2007.

[14] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, December 2010. To appear.

[15] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS^{RT}: A status report. In *Proceedings of the 9th Real-Time Linux Workshop*, pages 107–123, November 2007.

[16] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *Real-Time Systems*, volume 38, pages 133–189, February 2008.

[17] S. Funk, J. Goossens, and S. Baruah. On-line scheduling on uniform multiprocessors. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, December 2001.

[18] O. Harrison and J. Waldron. Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th Conference on Security Symposium*, pages 195–209, July 2008.

[19] W. Kang, S. H. Son, J. A. Stankovic, and M. Amirijoo. I/O-aware deadline miss ratio management in real-time embedded databases. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 277–287, December 2007.

[20] K. Lakshmanan, S. Kato, and R. Rajkumar. Open problems in scheduling self-suspending tasks. In *Proceedings of the 1st International Real-Time Scheduling Open Problems Seminar*, pages 12–13, July 2010.

[21] H. Leontyev and J. Anderson. A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. *Real-Time Systems*, 43(1):60–92, September 2009.

[22] B. Pieters, C. F. Hollemeersch, P. Lambert, and R. Van de Walle. Motion estimation for H.264/AVC on multiple GPUs using Nvidia CUDA. In *Applications of Digital Image Processing XXII*, volume 7443, page 74430X, September 2009.

[23] 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 the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 5–8, August 2010.

[24] Y. Watanabe and T. Itagaki. Real-time display on Fourier domain optical coherence tomography system using a graphics processing unit. In *Journal of Biomedical Optics*, volume 14, page 060506, December 2009.