GPUSync: A Framework for Real-Time GPU Management *

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

Abstract

The integration of graphics processing units (GPUs) into real-time systems has recently become an active area of research. However, prior research on this topic has failed to produce real-time GPU allocation methods that fully exploit the available parallelism in GPU-enabled systems. In this paper, a GPU management framework called GPUSync is described that enables increased parallelism while providing predictable real-time behavior. GPUSync can be applied in multi-GPU real-time systems, is cognizant of the system bus architecture, and fully exposes the parallelism offered by modern GPUs, even when closed-source GPU drivers are used. Schedulability tests presented herein that incorporate empirically measured overheads, including those due to bus contention, demonstrate that GPUSync offers real-time predictability and performs well in practice.

1 Introduction

Graphics processing units (GPUs) are commonly used today to accelerate intensive general-purpose computations, a practice termed *GPGPU*. The breadth of application domains that can benefit from GPGPU includes many in which *real-time* constraints exist. In automotive systems, for example, GPUs can be utilized to perform eye tracking [18], pedestrian detection [25], navigation [14], and obstacle avoidance [23]. If a single platform consolidates such features, then it may require a multicore system with multiple GPUs. Such systems are the focus of this paper.

CPU scheduling in such a system can follow a partitioned, clustered, or global approach. Under clustered scheduling, a system's m CPUs are partitioned into clusters of c CPUs each, and each task is scheduled within a specific cluster. Partitioned and global scheduling are special cases, where $c = 1$ and $c = m$, respectively. Similarly, GPUs can be organized by following a partitioned, clustered, or global approach. This categorization yields nine possible allocation categories, as illustrated in matrix form in Fig. 1.

It is currently unclear which allocation categories in Fig. 1 should be preferred in practice. To determine this, we are currently engaged in a long-term study in which these categories will be compared on the basis of real-time schedulability, for both fixed-priority (FP) and earliest-deadline-first (EDF) task prioritizations—18 categories in total. For such a comparison to be meaningful, per-category methods are needed for utilizing the considerable parallelism available in multi-GPU systems. After all, GPU-enabled systems are architected with the goal of fostering parallelism in mind.

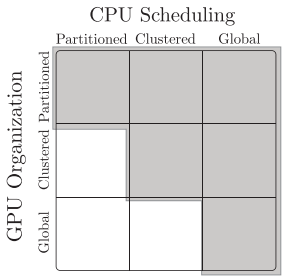

Figure 1: Matrix of CPU and GPU organization.

This paper is directed at such methods. Our goal is to determine how GPU-related parallelism can be dealt with within a *single* category in Fig. 1 when assessing schedulability (cross category comparisons are left for future work). Due to space constraints, we cannot consider all 18 alternatives. Thus, we limit attention to EDF scheduling under the shaded categories, as these sufficiently expose most interesting parallelism-related issues.

GPU-related parallelism. In any GPU-enabled multicore system, parallelism-related issues arise at several levels. For example, data must be transmitted to a GPU before computation begins; ideally, GPU transmissions and computation should overlap in time. Additionally, such transmissions result in increased traffic on shared buses, which are utilized in parallel by different tasks for different purposes. Bus traffic needs to be managed carefully or *all* computations in the system may be slowed [20]. If a system has multiple GPUs, as we assume, then further opportunities for parallelism exist. For example, on such a platform, it may be desirable to use a clustered or global GPU organization in order to avoid the utilization loss common to partitioned approaches. However, a GPU-using task may develop memory-based *affinity* for a particular GPU as it executes. In such cases, program state (data) is stored in GPU memory that is accessed each time the task executes on that particular GPU. This state must be *migrated* each time the task uses a GPU different from the one it used previously. Such migrations increase bus traffic and thus affect system-wide performance and predictability.

As explained in greater detail later, prior work on real-time GPU management has only partially addressed these parallelism-related issues. Furthermore, issues unique to multi-GPU systems have received very little attention.

Contributions. In this paper, we present a real-time GPU management framework called GPUSync, which is cognizant of system architecture and task GPU affinity and exploits the parallelism offered by modern GPUs. While the management of GPUs is viewed as a *scheduling* problem in

*Supported by NSF grants CNS 1016954, CNS 1115284, and CNS 1218693; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; AFRL grant FA8750-11-1-0033; and an NSF graduate fellowship.

most prior related efforts (e.g., see [2, 6, 15, 16]), we view such management as a *synchronization* problem. This viewpoint reflects the fact that GPUs are treated as I/O devices by the scheduler within the OS (even in on-chip architectures).

As noted above, we focus exclusively herein on EDF scheduling applied to the shaded categories in Fig. 1. However, variants of GPUSync that utilize different synchronization techniques can be applied to the other categories.

This paper's organization and our specific contributions are as follows. After addressing necessary preliminaries in Secs. 2 and 3, we describe the design of GPUSync in Sec. 4. GPUSync utilizes real-time locking protocols to manage GPU-related resources, so blocking bounds are needed for schedulability analysis—these we derive in Sec. 5. GPUSync is configurable in many respects. For example, the "distance" (with respect to the interconnection topology) that GPU data can migrate and the "chunking size" to use are configurable. Sec. 6 presents an assessment of tradeoffs concerning such configurable settings from both schedulability and runtime-performance perspectives. For the former purpose, we implemented GPUSync in UNC's Linux-based LITMUS$^{\mathrm{RT}}$ real-time OS [5], measured relevant overheads, and applied these overheads within overhead-aware schedulability studies. For the latter purpose, we conducted studies in which various real-time-related metrics were recorded for test workloads. Many of our proposed GPU management techniques proved able to either improve schedulability or improve actual performance without degrading schedulability. After discussing these results, we conclude in Sec. 7.

2 Target Platform

Our assumed target platform is a multicore system with one or more GPUs. The task systems we envision being hosted on such a platform are motivated by the automotive systems mentioned in Sec. 1. Specifically, we assume that such tasks are sporadic, have provisioned worst-case execution times that are rarely (if ever) exceeded, and are subject to the soft real-time (SRT) constraint of bounded deadline tardiness. In work on CPU scheduling, the clustered and global scheduling alternatives highlighted in Fig. 1 have been shown to be particularly effective in supporting such task systems [4].

It may seem counterintuitive to allow deadline tardiness in a safety-critical automotive application. However, "bounded tardiness" does not necessarily equate to "unsafe." For example, the reaction time of an *alert* driver is about 700ms [13]. Thus, a pedestrian avoidance system may only need to react to events within a 700ms window to be a viably safe system. Furthermore, a system designer may only be able to implement basic safety features under hard real-time (HRT) constraints (no tardiness), yet more complex and robust systems may be possible if stringent requirements are relaxed.[1] These robust systems may indeed be *safer* than basic ones, despite bounded tardiness. This is especially true if empirical tests show that deadline misses are rare even when tardiness is merely bounded analytically.

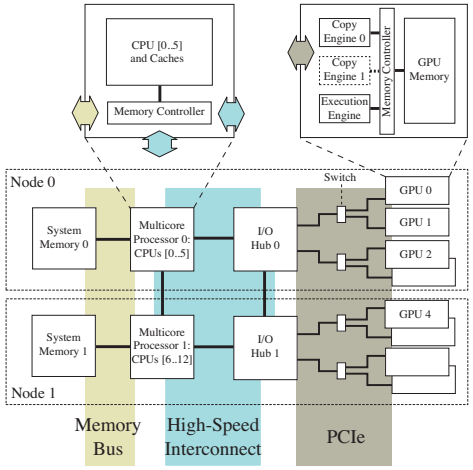

Figure 2: Assumed system architecture.

High-level architecture of assumed system. GPUSync should expose the parallelism offered by the underlying hardware to the extent possible while maintaining bounded tardiness. We now describe the canonical architecture assumed in this paper and the parallelism it offers. This canonical system is based on the lab machine used in our experiments. We note potentially interesting architectural alternatives found in other systems where appropriate. Due to space limitations, we limit attention to GPU technologies from NVIDIA, whose CUDA [1] platform is a popular GPGPU solution. However, our techniques may also be used with GPUs from other manufacturers.

Fig. 2 depicts the high-level architecture of the assumed system, which is a large-scale GPU-enabled system. There are four major types of components: system memory, multicore (MC) processors,[2] I/O hubs, and GPUs. These components are connected via three bus interconnects: the memory bus, high-speed processor interconnect, and PCIe bus.

The CPUs in each MC processor are connected to system memory by an on-chip memory controller attached directly to the memory bus. The MC processors are linked through full-duplex high-speed interconnects that allow data to travel in both directions simultaneously at full speed. On the MC processors, these interconnects communicate directly with the memory controller, allowing access to system memory. The I/O hubs also connect to each other and the MC processors using the same high-speed interconnect. Thus, I/O hubs can also access system memory.

The I/O hubs connect to GPUs using a packet-switched, dual-simplex, PCIe bus.[3] Each I/O hub has many independent "lanes" that are bundled in parallel to form "links." Each

[1] True HRT constraints are problematic in a GPU-enabled system anyway due to hardware complexity in such systems, the closed-source nature of GPU drivers, and the lack of timing analysis tools for such systems [8].

[2] We use the terms "CPU" and "core" interchangeably. We use the term "multicore processor" to refer an entire multicore chip.

[3] A dual-simplex bus is functionally equivalent to a full-duplex bus. The two only differ at the physical level.

link attaches to a PCIe *switch* that multiplexes additional links to increase the number of supported GPUs. Thus, the PCIe bus is organized hierarchically.[4] The GPUs attach to the links provided by the switches.[5] Devices can also *transmit directly to each other* if they share an I/O hub.

At the highest level, components are partitioned into non-uniform memory access (NUMA) nodes. As seen in Fig. 2, each NUMA node includes a pool of system memory, a MC processor, an I/O hub, and a collection of GPUs.[6] Memory and device access across nodes is supported, but requires an additional "hop" over the high-speed interconnect. Thus, memory traffic should be localized within a node if possible.

GPU architecture. As seen in Fig. 2, a GPU has four major components: an *execution engine*, one or more DMA *copy engines*, a memory controller, and specialized high-speed memory. The execution engine consists of many parallel processors and performs computations, similar to a CPU. The copy engines, connected to the PCIe bus, transmit data between system memory and GPU memory. Though the PCIe bus is dual-simplex, most GPUs only have one copy engine, and thus cannot send and receive data at the same time. However, high-end GPUs may have an additional copy engine, enabling simultaneous bi-directional transmissions.

The execution and copy engines operate independently—a copy engine may transmit data while the execution engine executes. Also, some modern GPUs have the capability to transmit directly to each other, bypassing system memory.

GPGPU operations and state migration. GPGPU programs execute on CPUs and invoke programs on GPUs called *kernels*. The typical execution sequence is as follows: (i) transmit input data for GPU kernels from system memory to GPU memory; (ii) execute kernels to operate on the input data, storing results in GPU memory; (iii) copy results from GPU memory back to system memory. A program may repeat this sequence several times before completing. Each action, unless explicitly broken up by the programmer, is performed non-preemptively by the GPU.

In addition to memory used for input and output, recurrent tasks may maintain *state* in GPU memory. For example, motion-tracking algorithms maintain information about the movement of objects between video frames. A task has *affinity* with the GPU that holds its most recent state. State must migrate with tasks from one GPU to another.[7] The *cost* of migration is the time it takes to move state from one GPU to another. Cost is partly dependent upon the method used to copy state between GPUs as well as the *distance* between GPUs. Distance is the number of links to the nearest common switch or I/O hub of two GPUs. For example, in Fig. 2, the distance between GPUs 0 and 2 is two (one link to a switch, a second link to a common I/O hub). Migrations using direct GPU-to-GPU memory copies, especially over short distances, are fast due to proximity and likely reduce bus contention since less distance is traveled.

Architectural implications. From a real-time perspective, we wish to constrain system utilization the least while maintaining predictable real-time performance. System utilization can be increased by exploiting parallelism. In the architecture above, GPU-related parallelism includes the dual-simplex PCIe bus and the independent operation of the execution and copy engines. Efficient GPU management techniques should allow the simultaneous use of these components. However, this is difficult due to bus contention issues.

GPGPU programs are data intensive and generate significant traffic between system and GPU memory. Within the same NUMA node in Fig. 2, data copied between system and GPU memory traverses the three buses, a system memory controller, and one PCIe switch. These elements are *shared* by concurrently executing operations. The speed of such a data copy is a function of contention, interconnect bandwidth, and bus arbitration protocols.

The management of bus contention in real-time systems has been explored extensively by Pellizzoni in his Ph.D. thesis [20]. However, his approach requires custom elements at every system level: specialized PCI hardware interposed between devices, OS modifications for memory and PCI bus scheduling, a custom compiler, and customized source code. While impressive in its scope, such an approach is currently infeasible in GPU-enabled real-time systems due to software complexity: the software stack that manages GPUs is exceedingly complex and often closed-source. Also, [20] does not address issues of GPU allocation and state migration. Instead, we endeavor to design efficient GPU resource management techniques that: (i) are cognizant of system architecture issues; (ii) aware of task GPU affinity; (iii) exploit the parallelism offered by modern GPUs; (iv) maintain real-time predictability; and (v) can be easily applied to existing systems and support a variety of real-time schedulers.

3 Prior GPU Approaches

Table 1 compares features of other notable GPU management frameworks related to GPUSync. PTasks [21], developed by Rossbach et al., creates a new OS-level infrastructure for GPU management. RGEM is a user-space real-time GPU scheduler designed by Kato et al. [15]. In more recent work, Kato et al. also developed Gdev, which extends many of the ideas developed in RGEM to the OS kernel space [16].

Other than GPUSync, only RGEM is designed for predictable real-time systems. RGEM schedules GPU operations by fixed priority. RGEM also addresses schedulabil-

[4] Unlike the older PCI bus where only one device on a bus may transmit data at a time, PCIe devices can transmit data simultaneously.

[5] Smaller systems may not be equipped with PCIe switches in which case GPUs attach directly to the I/O hubs.

[6] Nodes may share a single I/O hub in other configurations. Smaller systems may have only a single node and a single GPU.

[7] Memory must be pre-allocated on each GPU where a task may run to facilitate fast migrations. This is a reasonable constraint given that memory footprints of real-time GPU applications are relatively small. For example, the per-GPU memory footprint of an implementation of a complex computer vision algorithm that processes 640x480 resolution video streams is only 40MB [17]—much smaller than the 2GB and 6GB available in today's mid-range and high-end GPUs, respectively.

	Real-Time		Data/ Comp. Overlap	Auto. GPU Alloc.	P2P Migr.	Supports Clsd.-Src. Drivers
	FP	EDF				
PTasks			x	x		x
RGEM	x					x
Gdev			x			
GPUSync	x	x	x	x	x	x

Table 1: GPUSync vs. notable prior work.

ity problems caused by long non-preemptive copies between system and GPU memory by breaking large copies into smaller chunks, reducing the duration of priority inversions and thus improving schedulability. This chunking approach is also adopted by GPUSync.

PTasks and Gdev support the overlapping of GPU data transmissions and computation. PTasks uses data-flow graphs to describe flows of execution, and this is leveraged to individually schedule execution and copy engines. Gdev accomplishes overlapping by separately scheduling the execution and copy engines of a GPU. This is possible because Gdev is implemented in an open-source GPU device driver. This can be a limitation since these drivers lag behind vendor-provided ones with respect to performance, available features, and support for the most recent GPUs.

Excluding GPUSync, only PTasks supports automatic GPU allocation in multi-GPU systems. PTasks includes a data-aware GPU scheduler that attempts to greedily schedule GPU computations on the "best" available GPU at the time of an issued operation, where "best" is defined by GPU capabilities (such as speed) and affinity. However, data migration between GPUs must be performed by copying data to and from system memory. In the field of responsive or real-time GPGPU, no prior work has supported direct GPU-to-GPU, commonly referred to as *peer-to-peer* (*P2P*), migrations, let alone with real-time determinism.

Precursors to GPUSync use a synchronization-based approach for real-time GPU management in [10], with support for multiple GPUs added in [7] and [9]. GPUSync greatly expands upon these prior efforts by enabling fine-grained management of GPU resources.

4 GPUSync

We describe GPUSync's design by focusing on the platform depicted in Fig. 2, where EDF is used for CPU scheduling on a per-node basis, and the tasks assigned to a node can compete for any GPU within that node. This corresponds to the case of clustered CPU and GPU scheduling in Fig. 1. The other shaded regions in this figure can be seen as special-case instantiations of the case we consider.

GPUSync is made up of several components: a *GPU allocator* that utilizes a real-time k-exclusion locking protocol;[8] per-GPU real-time mutual exclusion (mutex) *engine locks*, one for each GPU copy and execution engine, to arbitrate access to those engines; and *API routines* that facilitate memory copies and P2P migrations.

[8] k-exclusion extends ordinary mutual exclusion by allowing up to k tasks to simultaneously hold a lock.

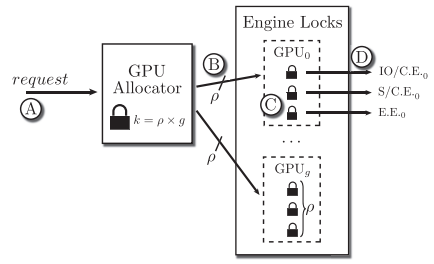

Figure 3: High-level design of GPUSync.

Fig. 3 illustrates how tasks acquire GPU-related resources. A request is issued to the GPU allocator in Step A. The GPU allocator determines the GPU to be allocated to satisfy the request, though the GPU may not be immediately available. The requesting job is allowed access to the assigned GPU once the GPU becomes available in Step B. In Step C, the job competes with other jobs allocated the same GPU for GPU engines; access is arbitrated by the engine locks. A job may access an engine on its assigned GPU after acquiring the corresponding engine lock in Step D.

We now describe GPUSync's individual components.

4.1 GPU Allocation and Engine Access

GPU allocation is performed using a k-exclusion locking protocol, and engine access is arbitrated by nested mutex locks. This overall locking strategy is derived from the *real-time nested locking protocol* (*RNLP*), which has been shown to be asymptotically optimal for supporting nested resource requests in multiprocessor real-time systems [27].

GPU allocator. We explain how GPU allocation is done by considering a single cluster with g GPUs (in Fig. 2, each node is a cluster and $g = 4$). GPU allocation is token-based. We associate ρ *GPU tokens* with each GPU. A job can only compete for access to a GPU's engines when it holds one of that GPU's tokens. The portion of a job's execution where a token must be held is called a *token critical section*. All GPU tokens are pooled and managed by a *single* k-exclusion lock, where $k = \rho \times g$. ρ is a configurable parameter. We explore the effect ρ has on schedulability further in Secs. 5 and 6.

We use a modified version of the Replica-Request Donation Global Locking Protocol (R^2DGLP), a recently proposed real-time k-exclusion locking protocol that is asymptotically optimal for globally-scheduled systems, to perform token allocation within a cluster [28]. Our modifications to the R^2DGLP are limited to load-balancing requests among GPUs and a few other technical details having to do with our particular usage of it. Due to space constraints, we cannot fully describe the R^2DGLP and only give a brief overview.

Access to each token in the R^2DGLP is arbitrated by a per-token FIFO-ordered queue, as seen in Fig 4. Jobs are enqueued on the shortest token queue upon token request. If

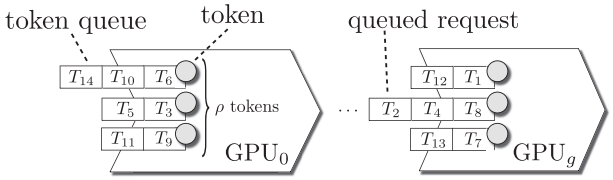

Figure 4: R^{2}DGLP token queues populated with token requests, identified by the task identifier, T_i, of the requesting task.

$k > 1$, then requests are load-balanced across queues.[9] An enqueued job suspends until it is at the head of its queue, in which case it is granted the corresponding token.

Engine locks. A mutex is associated with each GPU copy and execution engine, as seen in Fig. 3. For GPUs with *two* copy engines, there is some flexibility in how copy engines may be used. For example, one copy engine may be reserved only for P2P operations with the remaining engine used both for inbound and outbound data. For a machine like our evaluation platform, which has *one* copy engine per GPU, we are not able to take advantage of separate copy engines.

Each engine mutex prioritizes requests in FIFO order. Blocked jobs suspend while waiting for an engine. A job that holds an engine lock may inherit the priority of any job it blocks. Priority inheritance relations from the R^{2}DGLP may propagate to an engine holder to ensure timely real-time scheduling. A job releases an engine lock once all of its engine-related operations (e.g., GPU kernel execution or memory copy) complete. In order to reduce worst-case blocking, a job is allowed to hold at most *one* engine lock at a time, *except* during P2P migrations. Engine locks enable the parallelism offered by GPUs to be utilized while simultaneously obviating the need for the (unpredictable) GPU driver to make resource arbitration decisions.

4.2 Migrations

GPUSync supports both P2P and system memory migrations. The rules governing each type of migration differ.

P2P migrations. When migrating from GPU$_a$ to GPU$_b$, a job must hold copy engine locks for both GPU$_a$ and GPU$_b$. As shown in the full version of this paper [11], if these locks are acquired separately, then worst-case blocking grows quadratically with respect to the total number of GPU tokens. We avoid such excessive blocking by instead using *dynamic group locks* (*DGLs*) [26]. Using DGLs, a job atomically requests *both* copy engine locks simultaneously.

A job may issue memory copies to carry out migration once both engine locks are held. This isolates migration traffic to the PCIe bus: copied data does not traverse the high-speed processor interconnect or system memory buses—computations utilizing these interconnects are not disturbed.

[9] It is possible to use affinity-aware heuristics to estimate queue length in terms of time instead of number of requests. Such heuristics can improve average-case performance while maintaining real-time predictability. Due to space constraints, we do not consider such heuristics further.

System memory migrations. Migrations through system memory are also supported by GPUSync. Such migrations are performed *speculatively*, i.e., migrations are always assumed to be necessary. Thus, state data is aggregated with input and output data. State is always copied off of a GPU after per-job GPU computations have completed. State is then copied back to the next GPU used by the task for the subsequent job if a different GPU is allocated. An advantage of this approach over P2P migrations is that a job never has to hold two copy engine locks at once. This reduces lock contention and may improve blocking bounds, depending upon system and task set parameters.

Speculative migrations may seem heavy handed, especially when migrations between GPUs may not always be necessary. Instead, an "on demand" approach could be taken where each migration forces data to be copied off of the previously-allocated GPU to system memory and then to the newly-allocated GPU. However, this method offers no analytical real-time benefits over P2P migrations and would likely increase interconnect contention.

4.3 Memory Copy Routines

Recall from Sec. 2 that GPU copy engines perform operations non-preemptively, and large memory transactions can cause tasks to experience long periods of blocking. Prior work has shown that response times can be improved if these large copy operations are broken up, or chunked, into smaller ones [15, 16]. We adopt this method. Chunking is implemented via user-space memory copy APIs. The programmer specifies the source, destination, and amount of memory to be copied. The copy is carried out incrementally in configurable chunk sizes. The API also automatically acquires the necessary copy engine locks before copying each chunk.

5 Analysis

For completeness, we now present a coarse analysis of blocking under GPUSync. Our main purpose here is to expose analytical differences among various configurations of GPUSync. Fine-grained blocking analysis can be found in [11]. We make the simplifying assumption that each job of any task competes for a GPU token at most once.

We assume that the tasks utilizing GPUSync execute globally on m CPUs (perhaps a cluster or the entire system) and compete for access to g GPUs. Each task T_i is an implicit-deadline sporadic task specified by a tuple $(e_i, c_i, p_i, N_i^I, N_i^O, N_i^S, N_i^K, K_i)$. p_i is the minimum job separation parameter for T_i. The other terms are defined by focusing on an arbitrary job J_i of T_i. e_i (c_i) bounds the amount of CPU execution time J_i receives outside (inside[10]) a token critical section. N_i^I, N_i^O, and N_i^S bound the number of data chunks making up J_i's GPU input, output, and state, respectively. N_i^K bounds the number of GPU kernels issued by J_i, each of which executes for at most K_i time (the CPU execu-

[10] The CPU may execute control logic during GPU computations.

tion time required to dispatch a kernel assumed negligible).

We now present various blocking-related notation. Let C_i denote the maximum token critical section length of T_i, b_i^C denote the maximum time J_i may be blocked while waiting for a token, and b_i^E denote the maximum time J_i may be blocked *within* a token critical section for *all* engine locks. Then, the maximum time a job may be blocked accessing locks *and* tokens is give by $b_i \triangleq b_i^C + b_i^E$. Let X^I, X^O, and X^{P2P} denote the maximum time it takes to transmit a chunk of GPU data for input, output, and P2P migration, respectively, and let X^{max} denote the maximum of X^I, X^O, and X^{P2P}. Also, let S_i denote the maximum time to perform a GPU migration. For P2P migrations, $S_i = X^{P2P} N_i^S$. For migrations through system memory, $S_i = X^I N_i^S + X^O N_i^S$. $S_i = 0$ when GPUs are partitioned (no migrations). Let K^{max} denote the longest duration an execution engine lock is held by any other task, and let C^{max} denote the longest token critical section among all tasks.

A job must first acquire a token from the GPU allocator before it can begin using a GPU. This allocator uses the R^{2}DGLP to control access to ρg tokens. Using the blocking analysis presented in [28] for the R^{2}DGLP, a token-requesting job is blocked by at most $2\lceil m/(\rho g) \rceil - 1$ token critical sections of other jobs. Thus, the total duration of blocking while waiting for a token is bounded by $b_i^C = C^{max}(2\lceil m/(\rho g) \rceil - 1)$. Bounds on C^{max} must be computed since tasks may block while acquiring engine locks. By construction, the token critical section length for T_i is $C_i = c_i + b_i^E + X^I N_i^I + X^O N_i^O + K_i N_i^k + S_i$. All parameters for this equation have been derived, except for b_i^E.

b_i^E is the sum of all blocking experienced within the token critical section. Let b_i^K denote J_i's maximum total blocking time for the execution engine lock, let $b_i^{I/O}$ denote its maximum total blocking time while waiting to transmit input and output chunks, and let b_i^{P2P} denote its maximum total blocking time while waiting for copy engines locks to perform a P2P migration. Then, $b_i^E = b_i^K + b_i^{I/O} + b_i^{P2P}$.

A job may be blocked for every GPU kernel it executes when acquiring the execution engine lock of its allocated GPU. At most $\rho - 1$ other jobs at a time may compete for this lock for a given request. Since requests are FIFO ordered, the resulting blocking is bounded by $b_i^K = (\rho - 1)K^{max}$.

Bounds for $b_i^{I/O}$ and b_i^{P2P} depend on whether migrations are P2P or through system memory and on the number of copy engines per GPU. In our analysis, we assume that all migrations are supported using the same mechanism, though in practice GPUSync could support both types in the same system. Due to space constraints, we only present blocking analysis for GPUs with *one* copy engine here. (Analysis for GPUs with two copy engines is available in [11].)

Copy engine blocking with P2P. Under P2P migrations, any task holding a GPU token may request the copy engine lock of the GPU it used in its prior job in order to perform a migration. There are ρg such tasks. In the worst-case, they may all attempt to access the same copy engine lock at the same instant. Thus, *any* request for a copy engine lock may be blocked by $\rho g - 1$ other requests. From the blocking analysis of DGLs [26], the total number of blocking requests for the copy engine is at most $(\rho g - 1)$. Since no task requires more than X^{max} time to complete $b_i^{I/O} = ((\rho g - 1)X^{max})(N_i^I + N_i^O)$ and $b_i^{P2P} = ((\rho g - 1)X^{max})N_i^S$.

Copy engine blocking with system memory migration. In this case, copy engines are only accessed by tasks that have been given a token for an allocated GPU, so at most $\rho - 1$ other jobs may compete for the copy engine lock at a given instant. Recall that state is aggregated with input and output data. Thus, $b_i^{P2P} = 0$. However, now $b_i^{I/O} = ((\rho - 1)X^{max})(N_i^I + N_i^O + 2N_i^S)$ since state data must be handled twice.

Note the analytical differences between P2P migrations and system memory migrations. In the case of P2P migrations, copy engine lock contention is $O(\rho g)$, which results in $O(m)$ blocking total when including token blocking, while in the case of system memory migrations, copy engine lock contention is is $O(\rho)$, which results in $O(m/g)$ total blocking. Despite its inferior order of complexity, P2P migration may still result in better analytical bounds if the advantages of fewer and faster memory copies can be exploited (it is faster because state is not copied to memory). Furthermore, there are benefits to P2P migrations that cannot be captured in the above analysis, namely, isolation from the system memory bus and improved average-case performance.

Number of per-GPU tokens. We conclude this section by discussing trade-offs in selecting ρ. If ρ is small, then b_i^C is large and b_i^E is small. The converse is true if ρ is large. What value for ρ should be used? Is there a balance between b_i^C and b_i^E to be made? Analytically speaking, the ρ term often cancels out between b_i^C and b_i^E. However, this is not always the case due to the ceiling taken when computing b_i^C. In practice, ρ should only be great enough to keep GPU resources busy while maintaining acceptable blocking bounds.

6 Experimental Results

In this section, we assess trade-offs in the design choices affecting GPUSync by presenting the results of overhead-aware schedulability studies. We also discuss experiments involving real-time computer vision workloads. To obtain the overhead information needed in our schedulability study, we implemented GPUSync in LITMUS$^{\mathrm{RT}}$ [5][11] and measured various overheads while executing workloads devised to expose worst-case scenarios. We then incorporated these overheads into schedulability experiments. We also used our LITMUS$^{\mathrm{RT}}$ implementation for the computer vision study. These experimental efforts are discussed in detail next.

6.1 Overhead Measurements

Our implementation of GPUSync was run on a test platform much like that in Fig. 2. This platform has two "nodes,"

[11] Source code is available at `www.litmus-rt.org`.

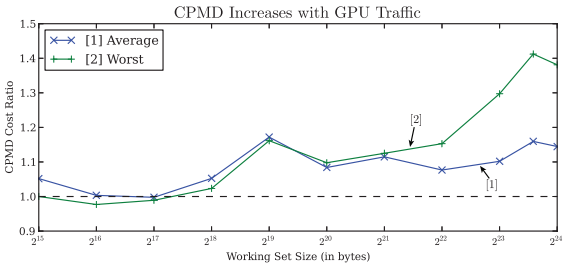

Figure 5: CPMD cost increase with GPU traffic.

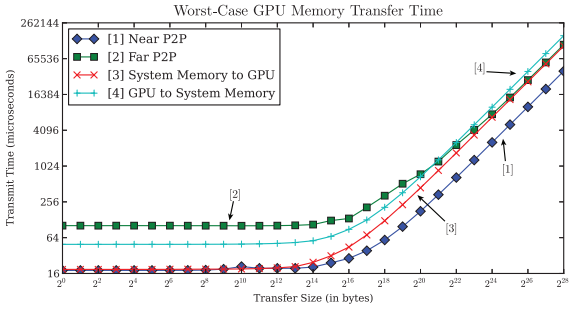

Figure 6: Worst-case GPU memory transmission times.

each with one Xeon X5060 processor with six 2.67GHz cores and four closely connected NVIDIA GTX-470 GPUs. Within each node, we used EDF scheduling globally and either pooled GPUs or statically assigned them to cores. We used a CUDA 5.0 for our GPGPU runtime environment.

Relevant overheads include those of an algorithmic nature and those related to memory accesses. *Algorithmic overheads* include: thread context switching, scheduling, job release queuing, inter-processor interrupt latency, CPU clock tick processing, and GPU interrupt processing. We measured these overheads by using light-weight tracing techniques while executing workloads that stress the various hardware components managed by GPUSync.[12]

Memory overheads reflect lost capacity due to cache affinity loss and GPU memory transfer rates. These overheads are dependent on the interleaving of memory accesses by different tasks and thus are more difficult to measure. Cache affinity loss manifests as *cache preemption/migration delays* (*CPMDs*) [3, 4], which reflect the increased execution time of a task needed to repopulate caches after it is preempted or migrated. Overheads due to GPU memory traffic, called *GPU memory traffic delays* (*GMTDs*), affect the rate at which GPU memory copies can be executed. CPMDs and GMTDs are interrelated because CPUs and GPUs share the system memory bus. This interrelation has not been fully considered in prior work in which CPMDs or GMTDs were measured [3, 4, 15]. We account for this interrelation by stressing buses with respect to both GPU and CPU traffic simultaneously. We now describe our CPMD and GMTD measurement process in more detail and discuss a subset of the measurement results presented in full in [11].

Measuring CPMDs. We measured CPMDs using methods described in [3, 4], extended to include the effects of GPU memory traffic. Specifically, a test thread that performs memory operations was executed in the presence of "cache-polluting" threads that cause cache evictions and heavy system memory bus load, and additional threads that copy data between GPUs and system memory. The test thread periodically sleeps and migrates among CPUs to simulate preemptions and migrations. CPMDs were computed by comparing the execution time of a sequence of read/write operations upon data of a given *working set size* (*WSS*) in the presence and absence of preemptions and migrations. CPMDs are sensitive to read:write ratios; to keep our study tractable, we limited attention to a read:write ratio of 3:1. Many thousands of samples are required for good measurements.

Fig. 5 shows the relative increase in CPMDs we observed in the presence of GPU traffic on the system memory bus. The cost increase is depicted as a ratio of CPMDs with GPU traffic to CPMDs without GPU traffic. CPMDs increased between 10% and 20% for average-case measurements, and up to 40% for worst-case measurements.

Measuring GMTDs. GMTDs were measured in the presence of system memory bus load caused by CPU cache polluters. GPU bus polluters were also executed to stress the PCIe bus. A test thread then performed GPU memory copies of various sizes. Memory pages were pinned so that memory copies would be exempt from page faults and be as fast as possible. Measurements for sending, receiving, and performing P2P memory copies were made.

Fig. 6 illustrates the measured times; note that both axes are on a base-2 logarithmic scale. *Far* (*near*) P2P copies are between GPUs with a distance of two (one). Note that far P2P copies take only slightly longer than copies between system memory and GPUs. However, near P2P copies take less than half the time of any other copy method for copies 256KB or greater in size—reduced bus contention is a clear win. Note also that small memory copies (4K or less) take roughly the same time within (but not among) each copy method, indicating that fixed-cost overheads dominate for small copies, though these overheads differ for each type of copy. These fixed costs gradually have less of an effect between 8K to 256K, after which transmit times scale linearly with data size. This implies that chunk sizes should at least be 256K to minimize overhead-related utilization loss.

Fig. 7(a) shows the relative increase in *worst-case* GPU memory copy times we observed in the presence of CPU traffic on the system memory bus. There are some surprising results. First, there appears to be an anomaly for near P2P copies of 512B, where worst-case transmissions are actually faster with CPU traffic. Another surprise is that the worst-case system-memory-to-GPU transfer time for *small* transfers is also faster with CPU traffic. Both anomalies per-

[12] As is done in [3, 4] and suggested by the US National Institute of Standards and Technology [22], we used an interquartile range filter to remove outliers in all measurements that can be affected by non-deterministic events such as interrupt handling since these events introduce measurement errors.

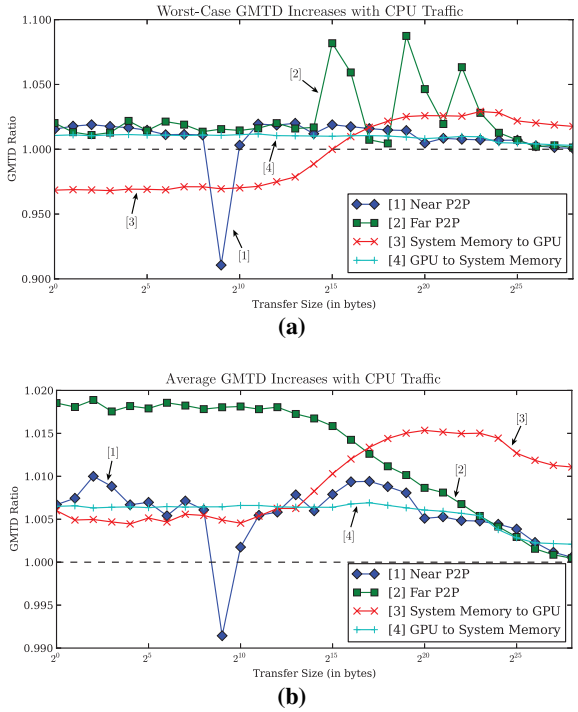

Figure 7: **(a)** Worst-case and **(b)** average GMTD cost increase with CPU system bus traffic.

sisted across many trials, so additional investigation is merited. These anomalies may be due to behaviors within the closed-source GPU driver that are difficult to discern. In any event, the time-scale of these transfer times is less than 32µs so they are of little practical impact. Also, when we examine the relative increase in *average* GPU memory copy times in Fig. 7(b), we see that system-memory-to-GPU copy times do increase, as expected. Despite these anomalies, our data indicates that GMTDs rarely increase by more than 5% in the worst case, and less than 2% in the average case. Thus, this data suggests that GMTDs, unlike CPMDs, are not strongly affected by system memory traffic from the CPUs.

6.2 Schedulability Experiments

We evaluated GPUSync with overhead-aware schedulability tests to better understand the real-time properties of GPUSync and trade-offs in design choices. We randomly generated task sets of varying characteristics and tested them for SRT schedulability using the methods described in [12]. We now describe the experimental process we used.

Experimental setup. There is a wide space of task set and system configuration parameters to explore. Task set parameters include: task set utilization distribution, task set period distribution, CPU cache WSS distribution, number and duration of GPU kernels, size of per-job GPU input and output data, size of task state stored in GPU memory, GPU memory transmission chunk size, the critical section length of the GPUSync token, and the ratio of GPU-using tasks to CPU-only tasks. System configuration parameters include: the number of GPU tokens, migration method used, and GPU cluster configuration. In order to keep our study tractable, we kept several of these parameters constant and allowed others to vary between (what we believe to be) realistic real-world bounds for the automotive systems discussed earlier. For example, task periods were uniformly distributed in the range [33ms, 100ms], reflecting the sensor rates of video cameras and LIDAR detectors found in advanced vehicle prototypes [14]. Due to page constraints, we cannot fully describe the full range of task set parameters here and refer the reader to [11] for details. We merely state that each unique permutation of parameters yields a single *task set scenario*, and that there were 360 task set scenarios in total.

Each task set scenario was combined with several *system configurations*, assuming a platform like our test platform (as illustrated in Fig. 2). System configurations varied in ρ, chunk size, and CPU/GPU organization. GPU management methods using GPUSync with P2P migrations, GPUSync with migrations through system memory, and exclusive GPU allocation through a R^{2}DGLP k-exclusion lock were also tested. Each permutation of these parameters resulted in a unique system configuration, for a total of 54 system configurations. Task set scenarios and system configurations were combined and then tested under both average- and worst-case overhead assumptions, for a total of 38,880 experiments. We generated task sets for each experiment with task set utilizations ranging from (0, 12], capturing the possible system utilizations on our test platform. Each task set was partitioned using a two-pass, worst-fit, heuristic that first partitioned GPU-using tasks among GPU clusters and associated CPU clusters in one pass, and then partitioned CPU-only tasks in another. Schedulability for bounded tardiness was then tested. Blocking terms were computed using *fine-grained* analysis from [11] (in contrast to the coarse-grained analysis in Sec. 5). Enough task sets were tested for each experiment to generate smooth schedulability curves; this came to roughly 400,000 task sets per experiment.

Results. Our schedulability results are given in full in [11]. Key observations following from these results are stated below and supported using representative graphs that compare different GPUSync settings in terms of *schedulability* (the fraction of generated task sets deemed schedulable). In the presented graphs, "P2P" denotes P2P migrations, "System memory" denotes system memory migrations, and "Exclusive alloc." denotes exclusive allocation using the R^{2}DGLP to arbitrate access to the GPU as a whole.

Obs. 1. When tasks' state sizes are large, P2P migrations offer better schedulability, despite greater worst-case copy engine contention. When the state size is small, such contention leads to worse schedulability.

In Fig. 8, we observe that when each task's state is 64MB (curves 3 and 4), GPUSync with P2P migrations has better schedulability than exclusive GPU allocation. However, when the state size is only 4MB, GPUSync with memory migrations (curves 1 and 2) has better schedulability

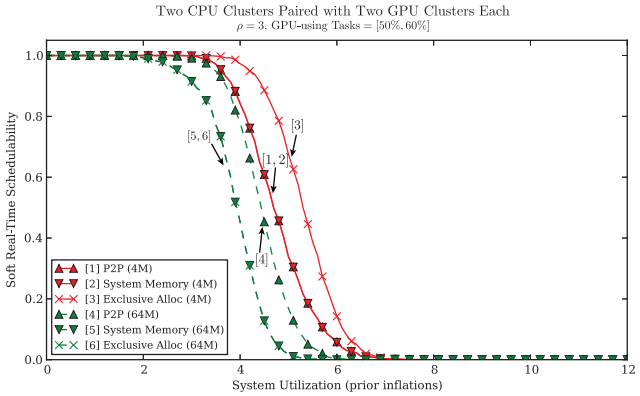

Figure 8: Effect of state size on schedulability.

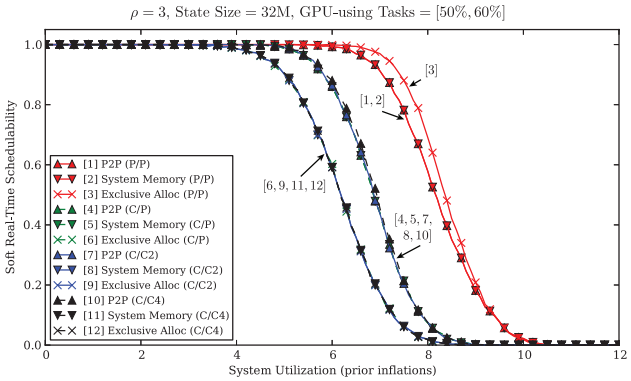

Figure 9: Effect of organizational method on schedulability.

than P2P migrations. Recall from Sec. 5 that P2P migrations cause $O(m)$ blocking, while memory migrations only cause $O(m/g)$ blocking. The increased asymptotic blocking in the case of P2P migrations is offset by the improved bus contention and therefore transfer times of migrations. However, when the state size is small, such gains do not offset increased blocking bounds.

Obs. 2. Schedulability of fine-grained locking meets or exceeds that of coarse-grained locking when state sizes are large. However, the parallelism afforded by this approach can have a negative effect on schedulability when state size is small. In either case, we expect improved average-case response times in practice.

This can also be seen in Fig. 8, where the curve for system memory migration (5) and exclusive GPU locking (6) overlap. Given that coarse- and fine-grained locking are comparable for applications with larger state sizes, from a schedulability point of view, it makes sense to employ fine-grained locking in practice to improve average-case response times, as will be discussed later.

Obs. 3. ρ does not significantly impact schedulability, but larger values of ρ support more average-case parallelism.

Due to space constraints, we omit a graph demonstrating this observation since schedulability curves are nearly indistinguishable among tested values of ρ. However, such graphs are available in [11]. In our experiments, we only tested $\rho \in \{1, 2, 3\}$ because larger values negatively impact blocking when applying fine-grained analysis. Our results do however suggest that ρ can be set to the number of engines per GPU. This allows copy and execute engines to be used concurrently, which can improve average-case response times.

While comparing different organizational methods is not the focus of this paper, we note the following.

Obs. 4. Regarding the clustering of GPUs, schedulability is generally better with a cluster size of one (partitioning) and is similar for other cluster sizes.

This observation is supported by Fig. 9. In this figure, curves 4, 5, 6; 7, 8, 9; and 10, 11, 12 depict a system in which the six CPUs on one node form a cluster and the four GPUs on that node are organized, respectively, in four clusters of size one (partitioned), two clusters of size two, and one cluster of size four. Curves 1, 2, and 3 depict a partitioned system, in which both GPUs and CPUs are partitioned; each GPU is mapped to a single processor, and tasks cannot migrate among processors. In the partitioned case, four processors host tasks that cannot access any GPU.

We believe that different non-unit cluster sizes exhibit similar schedulability because migration data is only part of a job's use of the copy engines—there is also input and output data. Additionally, as is seen in Fig. 7, the difference in transfer times between near and far migrations is small. GPU partitioning avoids these migration costs. However, partitioning approaches are subject to capacity loss on account of bin-packing-related issues, and may not be as responsive as systems in which migrations are allowed.

6.3 Real-Time Vision Workloads

We now describe the vision-related experiments mentioned earlier. In these experiments, we adapted a freely-available CUDA-based feature tracking program to GPUSync on LITMUS$^{\mathrm{RT}}$ [24].[13] Feature tracking is an important application in automotive systems (our motivating application) that sense and monitors the environment. The tracker represents a scheduling challenge since it utilizes both CPUs and GPUs to carry out its computations. Though feature tracking is only one GPGPU application, its image processing operations are emblematic of many others.

We stressed the system by applying featuring tracking to many independent video streams simultaneously, each with 320x240 resolution. Each video stream was handled by one task, with each frame being processed by one job. Tasks were partitioned between the two NUMA nodes of our system.

Frames were preloaded into memory in order to avoid disk latencies (such latencies would be non-existent with real video cameras). All data was page-locked in system memory to facilitate fast and deterministic memory operations. Memory copies for input and output data between system

[13] Source code available at `www.litmus-rt.org`.

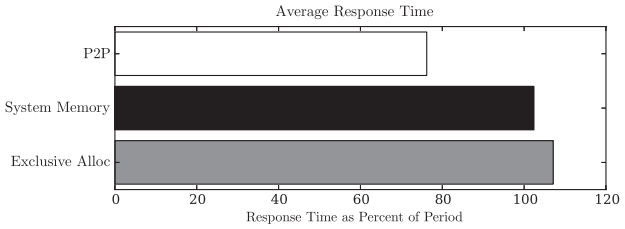

Figure 10: Observed response times under GPUSync

and GPU memory was roughly 1MB per frame, combined. However, task state data was about 6.5MB in size.

We let $\rho = 2$, and tested P2P and system memory migrations and exclusive GPU arbitration. We used a chunk size of 2MB for the P2P and system memory migrations (chunking is not necessary under exclusive GPU arbitration).

Measurements of job response times were made within a five minute execution window. Response times, as a *percent of period*, were averaged and are plotted in Fig. 10. Note that P2P migrations improve average-case response times rather significantly over either exclusive GPU arbitration, or system memory migrations. These results, together with those of the prior section, suggest that our techniques for managing parallelism can improve average-case responsiveness without compromising predictablility.

7 Conclusion

We have presented GPUSync, a GPU management framework that can be flexibly applied under different real-time schedulers to manage GPU-related resources in multi-GPU muticore systems. GPUSync's design leverages recently developed asymptotically optimal multiprocessor real-time locking protocols to manage GPU-related resources that can be accessed in parallel. While we have focused on certain scheduler and synchronization-protocol choices, variants of GPUSync can be applied assuming other choices. In fact, GPUSync enables schedulability to be assessed in a parallelism-cognizant way for all of the CPU/GPU allocation categories reflected in Fig. 1. Using these results, we next intend to conduct an overhead-aware schedulability study in which all of these categories are compared.

References

[1] CUDA Zone. `http://www.nvidia.com/object/cuda_home_new.html`.

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

[3] A. Bastoni. *Towards the Integration of Theory and Practice in Multiprocessor Real-Time Scheduling*. PhD thesis, Univ. of Rome, 2011.

[4] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, Univ. of North Carolina at Chapel Hill, 2011.

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

[6] J. Correa, M. Skutella, and J. Verschae. The power of preemption on unrelated machines and applications to scheduling orders. In *12th APPROX and 13th RANDOM*, 2009.

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

[8] G. Elliott and J. Anderson. Real-world constraints of GPUs in real-time systems. *17th RTCSA*, 2, 2011.

[9] G. Elliott and J. Anderson. Building a real-time multi-GPU platform: Robust real-time interrupt handling despite closed-source drivers. In *24th ECRTS*, 2012.

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

[11] G. Elliott, B. Ward, and J. Anderson. GPUSync: A framework for real-time GPU management (full version). `http://www.cs.unc.edu/~anderson/papers.html`, 2012.

[12] J. Erickson, U. Devi, and J. Anderson. Improved tardiness bounds for Global EDF. In *22nd ECRTS*, 2010.

[13] M. Green. "How long does it take to stop?" Methodological analysis of driver perception-brake times. *Transportation Human Factors*, 2(3), 2000.

[14] F. Homm, N. Kaempchen, J. Ota, and D. Burschka. Efficient occupancy grid computation on the GPU with LIDAR and radar for road boundary detection. In *Intelligent Vehicles Symposium*, 2010.

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

[16] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class GPU resource management in the operating system. In *USENIX ATC*, 2012.

[17] J. S. Kim, M. Hwangbo, and T. Kanade. Realtime affine-photometric KLT feature tracker on GPU in CUDA framework. In *12th ICCV Workshop*, 2009.

[18] M. Lalonde, D. Byrns, L. Gagnon, N. Teasdale, and D. Laurendeau. Real-time eye blink detection with GPU-based SIFT tracking. In *Canadian Conf. on Computer and Robot Vision*, 2007.

[19] PCI-SIG. *PCIe Base 3.0 Specification*, 2010.

[20] R. Pellizzoni. *Predictable and Monitored Execution for COTS-based Real-Time Embedded Systems*. PhD thesis, Univ. of Illinois at Urbana Champaign, 2010.

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

[22] S. Thrun. e-handbook of statistical methods. `http://www.itl.nist.gov/div898/handbook/`, 2010.

[23] S. Thrun. GPU Technology Conf. Keynote, Day 3. `http://livesmooth.istreamplanet.com/nvidia100923`, 2010.

[24] C. Tomasi and T. Kanade. Shape and motion from image streams under orthography: A factorization method. *J. of Computer Vision*, 9(2), 1992.

[25] Y. Wang and J. Kato. Integrated pedestrian detection and localization using stereo cameras. In *Digital Signal Processing for In-Vehicle Systems and Safety*. 2012.

[26] B. Ward and J. Anderson. Nested multiprocessor real-time locking with improved blocking. in submission.

[27] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *24th ECRTS*, 2012.

[28] B. Ward, G. Elliott, and J. Anderson. Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *18th RTCSA*, 2012.