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

This paper describes GPUSync, which is a framework for managing graphics processing units (GPUs) in multi-GPU multicore real-time systems. GPUSync was designed with flexibility, predictability, and parallelism in mind. Specifically, it can be applied under either static- or dynamic-priority CPU scheduling; can allocate CPUs/GPUs on a partitioned, clustered, or global basis; provides flexible mechanisms for allocating GPUs to tasks; enables task state to be migrated among different GPUs, with the potential of breaking such state into smaller "chunks"; provides migration cost predictors that determine when migrations can be effective; enables a single GPU's different engines to be accessed in parallel; properly supports GPU-related interrupt and worker threads according to the sporadic task model, even when GPU drivers are closed-source; and provides budget policing to the extent possible, given that GPU access is non-preemptive. No prior real-time GPU management framework provides a comparable range of features.

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 where *real-time* constraints exist. In automotive systems, for example, GPUs have been used to perform eye tracking [18], pedestrian detection [26], navigation [11], and obstacle avoidance [25]. 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.

Although multiple tasks can share GPUs, stock operating system (OS) drivers from GPU manufacturers offer little in terms of scheduling policies. Contention for GPU resources is often resolved through undisclosed closed-source arbitration policies. These arbitration policies typically have no regard for task priority and may exhibit behaviors detrimental to multitasking on host CPUs [10]. Furthermore, a single task can dominate GPU resources by issuing many or long-running operations. Allocation methods are needed that eliminate or ameliorate these problems.

Such methods must address issues in three fundamental categories: *allocation*, *budgeting*, and *integration*. Allocation issues include task-to-GPU assignment, the scheduling of GPU memory transfers, and the scheduling of GPU computations. Budgeting issues arise when tasks utilize more GPU resources than allocated. Integration issues relate to the technical challenges of integrating GPU hardware (and closed-source software) into a real-time system.

In resolving such issues, careful attention should be paid to managing GPU-related parallelism. For example, modern GPUs can send data, receive data, and perform computations simultaneously. Ideally, these three operations should be allowed to overlap in time to maximize performance. Additionally, data transmissions result in increased traffic on shared buses used in parallel by other tasks. One must carefully manage bus traffic or *all* computations in a system (even non-GPU-related ones) may be slowed [21].

If a system has multiple GPUs, then parallelism-related issues arise when allocating GPUs. 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 and accessed by the task each time it 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 little attention.

Contributions. In this paper, we present a real-time GPU management framework called *GPUSync* that addresses the above issues. While GPU management is often viewed as a *scheduling* problem (e.g., see [2, 6, 14, 13, 15]), we instead view it as a *synchronization* problem.[1] Indeed, GPUSync is the culmination of several years of research by us on various aspects of synchronization-based GPU management [7, 8, 9, 10]. GPUSync extends this prior work in many ways, most notably by significantly enhancing the parallel control of GPU-related resources, by allowing GPU state to be migrated, by providing support for GPU budget enforcement, and by allowing the overall system to be configurable for use under different scheduling policies, prioritization schemes, etc. As in our prior work, we consider GPU management under job-level-static-priority (JLSP) schedulers, such as the rate-monotonic (RM) and earliest-deadline-first (EDF) schedulers, where the allocation of both CPUs and GPUs

[*]Work supported by NSF grants CNS 1016954, CNS 1115284, CNS 1218693, and CNS 1239135; ARO grant W911NF-09-1-0535; and an NSF graduate fellowship.

[1]Although scheduling-based approaches may use synchronization for thread-safety and event-signaling, our approach is explicitly designed, implemented, and analyzed from the perspective of real-time synchronization.

may be on a partitioned, clustered, or global basis.

Regarding parallelism, GPUSync extends our prior work by enabling *simultaneous* two-way data transmission and GPU computation. In the case of non-partitioned GPU approaches, we also provide support for predictable and efficient GPU-to-GPU (*peer-to-peer*, or *P2P*) state migration, as well as self-tuning heuristics that attempt to determine when migrations can be effective.

Because GPU accesses are non-preemptive, it is not possible to completely isolate tasks from the effects of budget overruns by other tasks. In GPUSync, such isolation is provided to the extent possible by monitoring execution budgets and utilizing various methods that penalize overrunning tasks by reducing their future allocations. Although it is not always possible to immediately take a GPU away from an overrunning task, GPUSync can notify a task of an overrun through signals. User code may leverage exception handling language features to gracefully relinquish a GPU in response to these signals. In a multi-GPU system, the effects of overruns are further mitigated by having multiple GPUs that tasks can utilize (this is yet another instance of replicated hardware ameliorating the effects of erroneous behaviors).

In our prior work, we showed how to support GPU-related interrupt handling in accordance with the sporadic task model [9]. GPUSync improves upon this earlier work by also properly managing user-space helper threads, which are utilized in the latest closed-source GPGPU runtimes. Our support for GPU-related interrupt handling is applicable to closed-source drivers. This is advantageous since closed-source drivers often offer better performance and support newer hardware than open-source alternatives.

To assess the efficacy of GPUSync, we implemented it in LITMUS^{RT}, a Linux-based real-time OS jointly developed by UNC and MPI.[2] Using this implementation, we conducted experiments that show that a synchronization-based approach can successfully meet the resource allocation needs of a real-time GPGPU system. We further show that execution behaviors can be predicted to promote task affinity in multi-GPU systems.

Organization. In the rest of the paper, we provide needed background (Sec. 2), describe GPUSync (Sec. 3), compare it to relevant prior work (Sec. 4), present our experimental results (Sec. 5), and conclude (Sec. 6). Due to space limitations, we limit attention to GPU technologies from NVIDIA, whose CUDA [1] platform is widely accepted as a leading GPGPU solution. However, the design of GPUSync does not rely upon specific NVIDIA technology.

2 Background

We begin by describing relevant aspects of GPGPU technology and challenges it poses to OS-level resource management. We then explain our reasoning for pursing a synchronization-based framework for realizing GPU-enabled real-time systems.

[2]Source code for LITMUS^{RT} and GPUSync is available at `www.litmus-rt.org`.

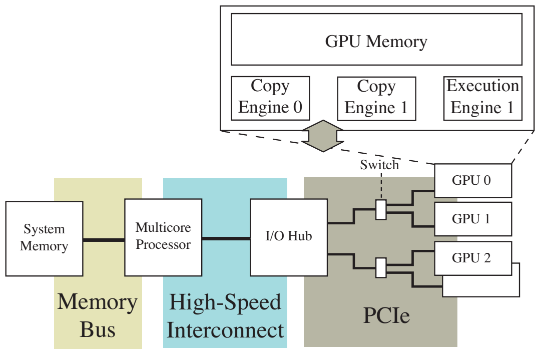


Figure 1: Example high-level architecture. On some multicore chips the I/O hub may be integrated.

GPU hardware. GPUs may be “discrete” or “integrated.” Discrete GPUs are those that plug into a host system as a daughter card. Integrated GPUs are on the same chip as system CPUs. In both cases, GPUs interface to the host system as I/O devices and are managed by OS drivers. Discrete GPUs differ from integrated GPUs in three ways: (i) they are much more computationally capable; (ii) they have local high-speed memory (integrated GPUs use system memory); and (iii) they operate most efficiently upon local memory, which requires copying data to and from system memory. We focus our attention on discrete GPUs for their performance characteristics and interesting challenges posed by memory management. However, our management techniques are still applicable to integrated GPUs, except that there is no need for GPU memory management.

Our GPUs of interest each have an *execution engine* (EE) and one or two DMA *copy engines* (CEs). The execution engine consists of many parallel processors and performs computations, similar to a CPU. The copy engines transmit data between system memory and GPU memory. GPUs commonly have only one copy engine, and thus cannot send and receive data at the same time. However, high-end GPUs may have an additional independent copy engine, enabling simultaneous bi-directional transmissions. The execution and copy engines perform operations *non-preemptively*.

Fig. 1 depicts a high-level architecture of a multicore, multi-GPU system. The copy engines connect to the host system via a full-duplex PCIe bus. PCIe is a hierarchically organized packet-switched bus with an I/O hub at its root. Switches multiplex the bus to allow multiple devices to connect to the I/O hub. Unlike the older PCI bus, where only one device on a bus may transmit data at a time, PCIe devices can transmit data simultaneously. Traffic is arbitrated at each switch using round-robin arbitration at the packet level in case of contention.[3] The structure depicted in Fig. 1 may be replicated in large-scale NUMA platforms, with CPUs and I/O hubs connected by high-speed interconnects. However, only devices that share an I/O hub may communicate directly with each other as peers.

GPGPU operations and state migration. GPGPU programs execute on CPUs and invoke a sequence of *GPU op-*

[3]The PCIe specification allows for other arbitration schemes, but these appear to be rarely implemented [20].

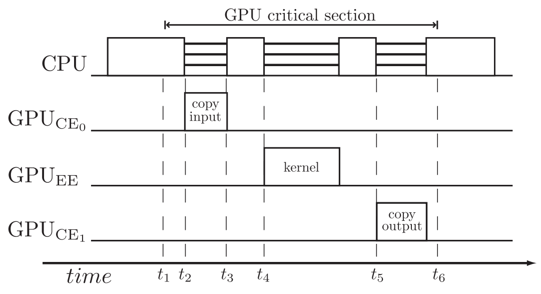


Figure 2: GPGPU program execution sequence.

erations. There are two types of GPU operations. *Kernel* operations are programs executed by the GPU execution engine. *Memory copy* operations are data transfers to or from a GPU's local memory; these are processed by the copy engines. A general execution sequence for a GPGPU program scheduled alone is depicted in Fig. 2. Observe that a program running on a CPU initiates GPU operations—the GPU does not initiate them independently. At time t_1, the GPGPU program selects a GPU to use. At time t_2, the program transmits input data for the GPU kernel from system memory to GPU memory. The memory copy is processed by one of the GPU's copy engines. The program waits (it may elect to either busy-wait or suspend) until the copy operation completes at time t_3. A kernel that operates on the input data is executed at time t_4—computational results are stored in GPU memory. The program copies the kernel output from the GPU at time t_5. Finally, the program no longer requires the GPU at time t_6. We call the duration from time t_1 to time t_6 a *GPU critical section* because the GPGPU program expects its sequence of operations to be carried out on the same GPU. Recall that GPU operations on the various engines are non-preemptive. For example, GPU_{CE_0} cannot be preempted during the interval $[t_2, t_3]$. However, the program running on the CPU is preemptive while waiting, if it busy-waits. There are two important things to note about this example. First, this is only a simple execution sequence. Any number of GPU operations may be issued within the GPU critical section. Second, we have depicted the input and output memory copies as processed by different copy engines—it is actually up to the GPU to select which copy engine to use.

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.[4] The *cost* of migration is the time it takes to move state from one GPU to another. Cost is partly dependent upon the *distance* between GPUs and the the method used to copy state between them. Distance is the number of links to the nearest common switch or I/O hub of two GPUs. For example, in Fig. 1, the distance between GPUs 0 and 2 is two (one link to a switch, a second link to a common I/O hub). One may use either of two methods to migrate state between GPUs. The first is a two-step process by way of a temporary buffer in system memory: data is copied to system memory from one GPU and then back out to another. The other method is a more efficient single-step approach using P2P communication: data is copied directly from one GPU to another. P2P-based migrations are more efficient, especially over short distances, due to proximity and reduced bus contention. However, this method requires coordination between the GPUs. Regardless of the method employed, current GPU technology requires that migrations must be carried out by the user application and is thus not transparent to the user.

[4]Memory must be pre-allocated on each GPU where a task may run to facilitate fast migrations. This is a reasonable constraint given that the memory footprints of real-time GPU applications are typically small (40MB in [16]) with respect to large GPU memories (2GB and greater).

OS concerns. A real-time OS (RTOS) provides timing guarantees for applications through prescribed resource allocation algorithms and the enforcement of allocation decisions. How does an RTOS achieve this for GPU-enabled systems? To answer this we must first consider other questions, such as: Which GPUs may a task access? Which GPU is best among multiple GPUs? When may a task access an assigned GPU? When may a task migrate between GPUs? How should contention for GPUs be arbitrated? How should allocation decisions be enforced?

The answers to these questions are not immediately apparent because the realities of GPU technology force us to use methods different from real-time CPU scheduling. There are three major differences that relate to allocation, budgeting, and integration.

First, allocation. GPUs have local memory that hold state and kernel input and output data. A task may have to move all this data between GPUs to support migration at arbitrary points (between non-preemptive operations) during execution. This is often prohibitively expensive, and not necessarily starvation-free, if migration decisions are not made carefully. This is not the case with modern shared-memory CPUs where data is always immediately accessible from any CPU and migration is transparent to the user. However, despite the cost of migration, opportunities to distribute work across GPUs may be lost if a scheduler is too migration-averse.

Second, budgeting. GPU operations are non-preemptive. Non-preemptive GPU scheduling may not be problematic to implement in and of itself, but it implies that budget policies must focus on recovery rather than strict enforcement since overrunning GPU operations cannot be halted. Furthermore, migration costs may motivate us to allow an overrunning task to execute to completion, rather than resume execution on a different GPU at a later time.

Finally, integration. CPU and GPU scheduling is interdependent. A GPU-using task requires CPU time in order to initiate GPU operations. Also, a GPU may block while waiting for a CPU to service device interrupts. Computing resources may be left idle if GPU operation initiation or interrupt servicing is delayed. Likewise, immediately handling these events may introduce unnecessary priority inversions and harm real-time schedulability.

Synchronization-based philosophy. In Sec. 1, we posited that GPU management is best viewed as a synchronization

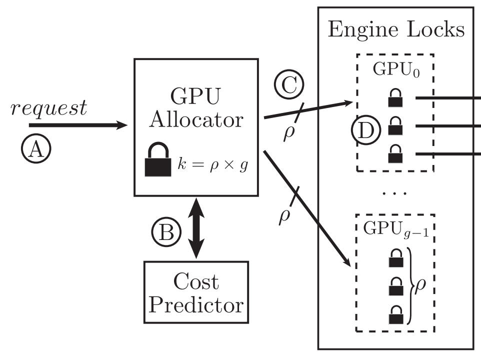


Figure 3: High-level design of GPUSync.

problem rather than one of scheduling. However, we wish to make clear that this distinction is somewhat blurred: the locking protocols we utilize *are* the GPU schedulers—these protocols prioritize ready GPU work and grant access accordingly. Nonetheless, as we shall see, a synchronization-based approach gives us established techniques to address problems relating to allocation, budgeting, and integration.

3 GPUSync

We consider a system with m CPUs, partitioned into clusters of c CPUs each, and h GPUs, partitioned into clusters of g GPUs each. Each GPU cluster is wholly assigned to a single CPU cluster in order to avoid complexities that arise due to the incomparability of priorities when resources are shared among CPU clusters.[5] We assume that the workload to be supported can be modeled as a traditional sporadic real-time task system, with jobs (task invocations) being scheduled by a JLSP scheduler. Furthermore, we assume that the scheduled system is a soft real-time (SRT) system for which bounded deadline tardiness is acceptable. While GPUSync's design does not inherently preclude use in hard real-time (HRT) systems, reliance upon closed-source software would make a claim of HRT support premature [8]. Thus, we focus on design strategies that improve predictability and average-case performance, while maintaining SRT guarantees. Finally, we assume that tasks can tolerate GPU migration at job boundaries, and that the per-job execution times of each task remain relatively consistent, with overruns of provisioned bounds being uncommon events.

We now present the design of GPUSync by separately discussing allocation, budgeting, and integration.

3.1 Allocation

GPUSync utilizes a two-level nested locking structure, which we discuss at a high level before delving into details.

The high-level design of GPUSync's allocation mechanisms is illustrated in Fig. 3. There are several components: a self-tuning execution *cost predictor*; a *GPU allocator*, based upon a real-time k-exclusion locking protocol,[6] augmented with heuristics; and a set of real-time *engine locks*, one per GPU engine, to arbitrate access to GPU engines.

We now describe the general steps followed within GPUSync to allocate GPU resources. Recall from Sec. 2 that a GPU critical section is a region of code where a GPU-using task cannot tolerate a migration between GPUs during a sequence of GPU operations. A job must acquire one of ρ *tokens* associated with a particular GPU before entering a critical section that involves accessing that GPU. We call a GPU critical section protected by a token a *token critical section*. For the sake of simplicity in discussion, we assume each job has at most *one* token critical section, even though GPUSync supports jobs with multiple non-overlapping sections. As depicted in Fig. 3, a job requests a token from the GPU allocator in Step A (or time t_1 in Fig. 2). Utilizing the cost predictor in Step B and internal heuristics, the GPU allocator determines which token (and by extension, which GPU) should be allocated to the request. The requesting job is allowed access to the assigned GPU once it receives a token in Step C. In Step D, the job competes with other token-holding jobs for GPU engines; access is arbitrated by the engine locks. A job may only issue GPU operations on its assigned GPU once its needed engine locks have been acquired in Step E. For example, an engine lock must be acquired at times t_2, t_4, and t_5 in Fig. 2. With the exception of P2P migrations, a job cannot hold more than one engine lock at a time.

The general structure of GPUSync is straightforward: a GPU allocator assigns jobs to GPUs and engine locks arbitrate engine access. However, many questions remain. For example, how many tokens can each GPUs have? What queuing structures should be used to manage token and engine requests? How can we enable GPU migration, yet minimize associated overheads? We now answer such questions, and provide additional rationale for our design choices.

3.1.1 GPU Allocators

Each cluster of g GPUs is managed by one GPU allocator; as such, we henceforth consider GPU management only within a single GPU cluster. We associate ρ tokens (a configurable parameter) with each GPU. All GPU tokens are pooled and managed by the GPU allocator using a *single* k-exclusion lock, where $k = \rho \times g$.

The GPU allocator's k-exclusion lock uses a hybrid queuing structure consisting of several fixed-size FIFO queues FQ_i and a priority queue PQ, as depicted in Fig. 4. The FIFO queue FQ_i is associated with the ith token. Token-requesting jobs are enqueued in a load-balancing manner until every FQ is f jobs in length. (We will discuss rules for load balancing shortly.) The parameter f is configurable. Additional requests that cannot be placed in an FQ "overflow" into PQ. Jobs are moved from PQ into an FQ as space becomes available. A job enqueued in FQ_i suspends until it is at the head of FQ_i, in which case it is granted the ith token.

This structure is similar to that of the Replica-Request Donation Global Locking Protocol (R^2DGLP), a real-time k-exclusion locking protocol that is asymptotically optimal

[5]There is a large body of work that explores inter-cluster resource sharing that could be drawn upon for future extensions of GPUSync that allow the sharing GPU clusters among CPU clusters [5, 22, 23].

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

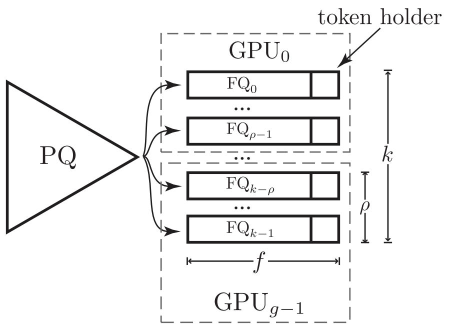


Figure 4: The structure of a GPU allocator lock.

for globally-scheduled systems [27]. Indeed, the GPU allocator lock utilizes the same inheritance rules as the R^{2}DGLP to ensure bounded blocking times. However, selecting different values for f allows the GPU allocator lock to take on different analytical properties. The GPU allocator functions exactly as the R^{2}DGLP when $f = \lceil c/k \rceil$, as the k-FMLP [9] when $f = \infty$, and as a purely priority-based protocol when $f = 1$. A system designer may tailor the GPU allocator to their specific task sets and schedulers. For instance, the k-FMLP can outperform the R^{2}DGLP for some tasks sets. The purely priority-based GPU allocator may be used with a task-level static priority scheduler such as the RM scheduler, when request interference on high-priority tasks must be minimized [4].

The value of ρ directly affects the maximum parallelism that can be achieved by a GPU since it controls the number of jobs that may directly compete for a GPU's engines—ρ must be at least the number of engines in a GPU if every engine is to ever be used in parallel. This implies that a large value should be used for ρ. However, the value of ρ also strongly affects GPU migrations. Too great a value may make the GPU allocator too migration-averse since tokens will likely be available for every job's preferred GPU, even those that are heavily utilized. Constraining ρ prevents GPUSync from overloading a GPU and promotes GPU migration to distribute load. We discuss this process next.

Maintaining GPU affinity. Real-time locking protocols are rarely designed to make use of online knowledge of critical section lengths, even though critical sections figure prominently in schedulability analysis and system provisioning. We augment the GPU allocator lock to incorporate knowledge of token critical section lengths into queueing decisions to reduce the frequency and cost of GPU migrations without preventing beneficial migrations.

The cost predictor returns an estimated token critical section length for a GPU request by a job J for each FQ where J's request may be potentially enqueued. This estimate includes the cost of migrating task state if J's task had previously executed on a different GPU. The total estimated critical section length for J's request for a particular FQ$_i$ is obtained by adding the execution time and any potential migration cost for the request to the total estimated waiting time for the request in that FQ$_i$. The latter can be obtained by summing the estimated execution times and migration costs of all requests currently enqueued in FQ$_i$. When the GPU allocator selects among several FQs for J's request, it can select the FQ that yields that shortest estimated token critical section length for J. This heuristic is sensitive to GPU migration costs and results in the desirable behavior of promoting only *beneficial* GPU migrations.

Cost predictor. To determine an estimated critical section length, the cost predictor must be able to determine the total execution time for a particular GPU request including any potential migration cost. GPUSync's cost predictor does this by measuring the accumulated CPU execution time of the requesting job once it has been allocated a GPU by acquiring a token. Execution delays due to preemption and blocking due to engine lock acquisition (explained later) are *not* included, but CPU suspension durations due to GPU operations (memory copies, GPU kernels) *are*. When a job releases its GPU token, this measurement gives a total request execution cost for both CPU and GPU operations and includes any delays due to migrations. However, this measurement is for only a single observation and provides a poor basis for predicting future behavior, especially since these measurements are strongly affected by PCIe bus congestion. Thus, we use a more refined process to drive a prediction model based upon statistical process control [17].

Specifically, for each task and GPU migration distance pair, we maintain an average and standard deviation over a window of recent observations (we found a window of twenty was suitable in our implementation). The average and standard deviation is recomputed after every new observation unless the observation falls more than two standard deviations away from the average and at least ten prior observations have been made. This filtering prevents unusual observations from overly-influencing future predictions.

One may be concerned that the heuristic nature of GPUSync's cost predictor does not lend itself to real-time predictability. However, the cost predictor has been designed to not violate analytical worst-case behavior. Predictability is preserved provided system provisioning does not rely upon the cost predictor's optimizations.

3.1.2 Engine Locks

A mutex is associated with each GPU copy and execution engine, was 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 copy engine used both for inbound and outbound data.

GPUSync can be configured to satisfy engine lock requests in either FIFO or priority order (all GPUs within the same GPU cluster must use the same engine lock order). Blocked jobs suspend while waiting for an engine. A job that holds an engine lock may inherit the *effective* priority of any job it blocks. We stress effective priority, because the blocked job may itself inherit a priority from a job waiting for a token. 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.

Engine locks should be held for as little time as possible in order to prevent excessive blocking times that degrade overall schedulability. Minimizing the hold time of execution engine locks requires application-specific solutions to break kernels into small operations. However, a generic approach is possible for copy engine locks. *Chunking* is a technique where large memory copies are broken up into smaller copies. The effectiveness of this technique is demonstrated in [13]. GPUSync supports chunking of both regular and P2P memory copies by way of a user-space library. Chunk size is fully configurable. The library also transparently handles copy engine locking.

Bus scheduling concerns. The copy engine locks indirectly impose a schedule on the PCIe bus if we assume that bus traffic from non-GPU devices is negligible. This is because these locks grant permission to send/receive data to/from a particular GPU to one task at a time. It is true that traffic of other GPUs will cause bus congestion and slow down a single memory transmission. However, since the PCIe bus is packetized, it is shared fluidly among copy engine lock holders. Thus, we can bound the effect of maximum PCIe bus congestion and incorporate it into real-time analysis. Deeper analysis or explicit bus scheduling techniques may be necessary if very strict timing guarantees are required, but such approaches are beyond the scope of this paper.

Migrations. GPUSync supports both P2P and system memory migrations, which are handled differently.

For a P2P migration from one GPU to another, a job must hold copy engine locks for both GPUs. Requests for both copy engine locks are issued together atomically to avoid deadlock.[7] The job may issue memory copies to carry out the migration once both engine locks are held. P2P migration isolates 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. However, gains from fast P2P migrations may be offset by higher lock contention: unlikely scenarios exist where every token holder in a GPU cluster may request the same copy engine lock simultaneously.

If GPUSync is configured to perform migrations through system memory, then such migrations are performed *conservatively*, i.e., they 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 corresponding task for its subsequent job if a different GPU is allocated. An advantage of this approach over P2P migration is that a job never needs to hold two copy engine locks at once.

3.2 Budgeting

Budget enforcement policies are necessary to ensure that a task's resource utilization remains within its provisioned budget. However, the non-preemptivity of GPU operations makes budget enforcement problematic. Even if limited preemption is provided by breaking a single GPU request into multiple non-preemptive regions, data on a GPU may be in a transient state at a preemption point and thus be too difficult (or too costly) to migrate to another GPU to resume at a later time. This motivates us to focus on budget enforcement based on overrun recovery rather than strict enforcement that absolutely prevents overruns. GPUSync provides three budget enforcement options: signaled overruns, early budget releasing, and a bandwidth inheritance-based method.

Pseudocode 1 Example of budget signal handling.

```
▷ Enabled/disabled on try-block enter/exit.
function BudgetSignalHandler()
    throw BudgetException();
end function

procedure DoJob()
    $t \leftarrow$ GetToken();
    $gpu \leftarrow$ MapTokenToGpu($t$);
    try:
        DoGpuWork($gpu$);                ▷ Main job computation.
    catch BudgetException:
        CleanUpGpu($gpu$);               ▷ Gracefully cleans up state.
    finally:
        FreeToken($t$);
end procedure
```

Signaled overruns. Under the signaled overrun policy, jobs are provisioned with a single budget equal to a maximum CPU execution time, plus a maximum total GPU operation time. A job's budget is drained when it is scheduled on a CPU or GPU engine. The OS delivers a signal to the job if it exhausts its budget. The signal triggers the job to execute an application-defined signal handler. The handler is application-defined since appropriate responses may vary at different points of execution. As depicted in Pseudocode 1, one way an application may respond to a signal is to unwind the job's stack (either by throwing an exception[8] or a call to the standard function `longjmp()`) and execute clean-up code before releasing its token lock.

Early releasing. The early releasing policy extends the signaled overrun policy by early-releasing a task's next job upon budget exhaustion to continue the execution of the overrunning job. This refreshes the task's budget, but its current deadline is shifted (postponed) to that of its next job. In essence, the next job has been sacrificed in order to complete the overrunning one. This policy penalizes an overrunning task by forcing it to consume its own future budget allocations. This prevents the system from being overutilized in the long-term. Under deadline schedulers, deadline postponement also helps to prevent the system from being overutilized in the short-term. We note that deadline postponement is challenging to implement since it requires priority inheritance relations established by locking protocols to be re-evaluated.

[7] GPUSync also grants requests atomically to avoid deadlock if priority-ordered engine locks are used.

[8] Throwing exceptions from signal handlers may require special compiler options.

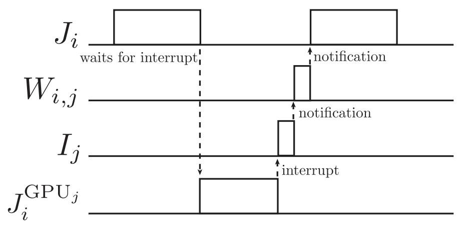


Figure 5: Schedule of worker threads and dependencies.

Bandwidth inheritance. As shown in [19], a job that overruns its budget while holding a shared resource can negatively affect other jobs, even non-resource-using ones. Bandwidth inheritance (BWI) limits the effects of such overruns to resource-sharing tasks. This is accomplished by draining the budget of a *blocked* job whose priority is being inherited by a scheduled job in place of the scheduled job's own budget. This can cause a blocked job to be penalized for the overrun of another, but improves temporal isolation from non-resource-using tasks. GPUSync allows the early releasing policy to be applied with the variant of BWI described in [3] when deadline scheduling is used. However, jobs must be provisioned with additional budget derived from analytical blocking bounds. This policy may be desirable in systems where CPU-only tasks require protection from GPU-using tasks in order to meet strict timing constraints.

3.3 Integration

Our desire to support closed-source drivers and GPGPU runtimes requires GPUSync to adapt to aspects of software that has not necessarily been designed with real-time predicability in mind. We have already addressed many of these aspects through synchronization and budgeting. A remaining issue to resolve is that of worker threads. There are two types of worker threads: (i) OS kernel-level threads, one per GPU, that process GPU interrupts, and (ii) GPGPU runtime user-level threads, g per task, that mediate communication between application code and the GPU driver. The primary purpose of the work executed by these threads is notification delivery. Real-time jobs are dependent upon these notifications, as illustrated by the schedule in Fig. 5. Here, a job J_i initiates work on GPU$_j$ (the engine irrelevant) and it suspends waiting for an operation to complete; GPU$_j$ signals completion with an interrupt. The interrupt is processed by interrupt thread I_j, which then notifies the user-space worker thread $W_{i,j}$, which in turn notifies J_i. Although these worker threads actually perform very little work, unbounded priority inversions can result if they are not properly scheduled.

To enable the real-time scheduling of worker threads, we map these threads to a particular GPU and register them with GPUSync before task set execution begins. This process can be tricky with closed-source software, but it is possible—we share the technical details in the online appendix of this paper.[9] When J_i *suspends* waiting for a GPU operation to complete, the user-level and interrupt threads for that GPU become eligible to run and may *inherit the priority* of J_i.[10] Thus, these threads are scheduled according to priority when signals need to be delivered, so J_i does not experience any priority inversions. Also, jobs with priorities greater than J_i are unaffected.

[9]Available at `www.cs.unc.edu/~anderson/papers.html`.

	Real-Time		Budg. Enfrced	Data/ Comp. Ovlp.	Auto. GPU Alloc.	P2P Migr.	Multi GPU Aware	Clsd. Src. Compat.
	FP	EDF						
PTasks			x	x	x		x	x
TimeGraph	x		x					
RGEM	x			x				x
Gdev			x	x				
GPUSync	x	x	x	x	x	x	x	x

Table 1: GPUSync vs. notable prior work.

User-level threads may only inherit priority from a single source since each thread is associated exclusively with a single task. However, the interrupt thread is shared among the engine lock holders of the interrupt thread's GPU. Under GPUSync, and similar to [29], the interrupt thread inherits the greatest priority among all suspended engine lock holders of its GPU. This may cause an interrupt to be processed at too great a priority since the interrupt thread may ultimately signal a lower-priority engine lock holder. However, this is unavoidable since we cannot know which of the suspended engine lock holders will receive the notification generated by the interrupt a priori.

4 Prior GPU Approaches

Table 1 compares features of other notable GPU management frameworks related to GPUSync. PTasks [24], developed by Rossbach et al., creates a new OS-level infrastructure for GPU management.

Kato et al. designed and implemented TimeGraph [14], RGEM [13], and Gdev [15, 12]. TimeGraph is a user-space/kernel-space real-time GPU scheduler designed for computer graphics. RGEM is implemented entirely in user-space and supports real-time GPGPU. Gdev extends many of the ideas developed in RGEM to the kernel-space.

Other than GPUSync, only TimeGraph and RGEM are designed for predictable real-time systems. Both TimeGraph and RGEM schedule GPU operations by fixed priority, and schedule GPU operations in priority order. GPUSync can be configured to subsume this functionality.

RGEM addresses schedulability 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 supported by GPUSync.

Gdev allows GPUs to access main system memory directly and has been shown to be beneficial in supporting low-latency applications [12]. This is also supported by closed-source GPGPU runtimes and may be used under GPUSync. However, direct system memory access is incompatible with GPUSync's copy engine management, so making use of such features under GPUSync requires re-

[10]We also schedule worker threads with a base-priority statically lower than any normal real-time task on idle CPUs. This has the potential to increase performance when a job's GPU operation has completed, but the job has yet to block for results.

served or exclusively-allocated GPUs for these tasks (which appears necessary in Gdev as well).

TimeGraph uses a global OS kernel-space GPU scheduler in conjunction with an open-source GPU device driver. Dependence on an open-source driver 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. RGEM, PTasks, and Gdev support the overlapping of GPU data transmissions and computation. RGEM utilizes a user-space GPU scheduler, separately scheduling the execution and copy engines of a GPU. PTasks uses data-flow graphs to describe flows of execution, and this is leveraged to individually schedule execution and copy engines. Gdev's design is similar to RGEM, but is implemented as a kernel-space open-source GPU driver.

All approaches, but RGEM, include budget enforcement. TimeGraph supports two budgetary methods: posterior and apriori. Posterior enforcement is similar GPUSync's early releasing policy, but it does not include deadline postponement. A job is scheduled under apriori enforcement if the job is predicted to not overrun its budget. Being non-real-time, PTasks and Gdev focus on quality-of-service and fairness.

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 P2P migrations, let alone with real-time determinism.

Precursors of GPUSync use a synchronization-based approach for real-time GPU management in [10], with support for multiple GPUs added in [7] and [9]. Support for real-time interrupt handling is also supported in [9], but a GPU could only be used by one task at a time. GPUSync greatly expands upon these prior efforts by enabling fine-grained management of GPU resources and also supports GPGPU runtimes that make use of user-level worker threads, such as in CUDA 4.2 and later.

5 Evaluation

In this section, we evaluate several performance characteristics in an implementation of GPUSync.

GPUSync was implemented as an extension to LITMUS^{RT}, a real-time kernel extension to Linux. GPUSync adds approximately 20,000 lines of code to LITMUS^{RT}.[11] Contributions to this total by category are: GPU allocator and locking protocols, 35%; scheduler enhancements, budgeting, and nested inheritance, 35%; GPU interrupt and thread management, 20%; miscellaneous infrastructural changes, 10%.

Our implementation of GPUSync was run on a test platform much like that in Fig. 1. This platform has two NUMA "nodes," each with one Xeon X5060 processor with six 2.67GHz cores and four NVIDIA Quadro K5000 "Kepler" GPUs, for a total of twelve CPUs and eight GPUs. The K5000 is a high-end GPU with two copy engines. We used CUDA 5.0 for our GPGPU runtime environment with the NVIDIA 304.54 driver.

We present our evaluation in two parts. First, we examine the effectiveness GPUSync's budgeting mechanisms, cost predictor, and affinity-aware GPU allocator. Second, we assess the effectiveness of clustered GPU management and P2P migrations through the scheduling of real-world computer vision workloads.

5.1 Budgeting, Cost Prediction, and Affinity

We used a mixed task set of CPU-only and GPU-using implicit-deadline periodic tasks to evaluate budget enforcement and the cost predictor. Numerical code was executed by tasks on both CPUs and GPUs to simulate real applications. Task execution time was tightly controlled through the use of processor cycle counters on both CPUs and GPUs. GPU-using tasks also transmitted data on the PCIe bus. Task periods ranged from 10 to 75 milliseconds, reflecting sensor sampling rates found in advanced vehicle prototypes [11]—one motivating application we discussed in Sec. 1.

The task set consisted of 28 CPU-only tasks and 34 GPU-using tasks. Each task was assigned a utilization based upon the combined processor time (CPU and GPU engines) a task's job must receive before completing. Of the CPU-only tasks, twelve had a utilization of 0.1, eight had a utilization of 0.2, and two had a utilization of 0.3. Of the GPU-using tasks, fourteen had a utilization of 0.1, fourteen more had a utilization of 0.25, and six had a utilization of 0.5. 90% of each GPU-using task's utilization was devoted to GPU operations, with 75% of that towards GPU kernels, and the remaining 25% towards memory copies. The amount of memory to copy was determined based upon desired copy time and worst-case bus congestion bandwidth derived from empirical measurements. Memory copies were evenly split between input and output data. Task state size was set to twice the combined size of input and output data. Additionally, data was transmitted in 2MB chunks.

We configured our system to run as a cluster along NUMA boundaries. Thus, there were two clusters of six CPUs and four GPUs apiece. The above task set was evenly partitioned between the two clusters. The task set was scheduled under clustered EDF (C-EDF) and clustered RM (C-RM) schedulers. Under C-EDF, the GPU allocator was configured to use the R^{2}DGLP as its token lock, and engine locks were FIFO ordered. Under C-RM, the GPU allocator was configured to be priority-ordered, and priority-ordered engine locks were also used. Three tokens ($\rho = 3$) were allocated to each GPU in order to allow all GPU engines to be used simultaneously. P2P migrations were also used. These configurations were selected because they both offer good analytical schedulability under their respective C-EDF and C-RM schedulers.

The task set was scheduled under two execution-behavior

[11]For comparison, the LITMUS^{RT} patch to the Linux 3.0 kernel is roughly 15,000 lines of code.

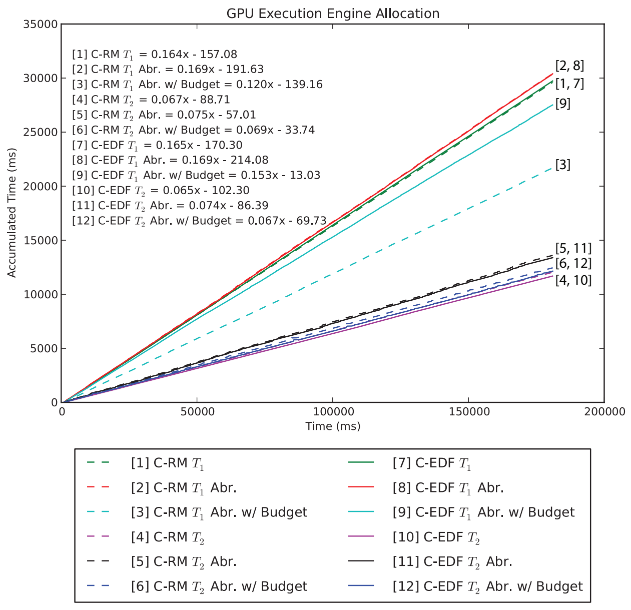


Figure 6: Allocated execution engine time.

scenarios. Under the first scenario, tasks adhered to their prescribed execution times as closely as possible. Under the second scenario, however, eight GPU-using tasks in each cluster were configured to exhibit aberrant behaviors by executing for *ten times* their normal execution time at random moments (roughly spaced by five seconds for each task). These scenarios were scheduled for 180 seconds under both C-EDF and C-RM schedulers and measurements were taken.

Budget performance. We analyze the ability of GPUSync to manage budgets (and penalize overrunning tasks) by examining the long-term utilization of the execution engines. Here, we test GPUSync under the no-budget-enforcement and the early releasing policies described in Sec. 3. We measure execution engine utilization (kernel time divided by period) with respect to the hold time of execution engine locks. This is an effective measure, even if the engine may idle while the engine lock is held, since all other tasks are blocked from using the engine.

From the task set described above, we focus our attention on two GPU-using tasks, T_1 and T_2. T_1 has a period of 15ms and a utilization of 0.25. T_1's ideal execution-engine utilization is 0.169, and it sends and receives 512KB to and from the GPU. T_2 has a period of 75ms, a utilization of 0.1, an ideal execution-engine utilization of 0.068, and it sends and receive 1024KB to and from the GPU. We focus on these tasks because of their short and long periods, respectively.

Fig. 6 depicts the accumulated time tasks T_1 and T_2 hold an execution engine lock over the duration of their execution. Equations characterizing each line are also given. We make several observations.

Obs. 1. A synchronization-based approach to GPU scheduling is effective at supplying GPU-using tasks with provisioned execution time.

Ideally, the slope of the lines in Fig. 6 should be equal to the task's execution-engine utilization. With the exception of line [3], the slopes of all the lines are very close to the desired utilization. For example, when T_2 is well-behaved under C-EDF scheduling (line [10]), the slope is 0.065, this is commensurate with the assigned utilization of 0.068.

Obs. 2. Budget enforcement can penalize aberrant tasks by allocating less execution time.

This may be observed in lines [3] and [9] for the aberrant task T_1 under both C-RM and C-EDF scheduling in Fig. 6. As shown by line [3] ([9]), T_1's utilization is 0.12 (0.153)—30% (10%) less than the provisioned 0.169, for C-RM and C-EDF, respectively. This loss of utilization is a result of the early releasing budget policy where any surplus from an early-released budget is discarded after an overrunning job completes.

Obs. 3. C-RM and C-EDF can both perform well.

For this particular experiment, we observe that both C-RM and C-EDF are able to supply the needed GPU execution time. This is an important result because it empowers system designers to select the scheduler that suits their needs. This is promising for SRT applications, where bounded tardiness can be guaranteed even for relatively high utilization levels.

Cost predictor accuracy. We measured the overheads related to the cost predictor because of our concern that computing averages and standard deviations can be computationally expensive. However, we found these worries to be unfounded. Updating the cost predictor estimate took $0.294\mu s$ on average, and $2.335\mu s$ in the (observed) worst-case.

We now discuss the accuracy of the cost predictor. Fig. 7 plots cumulative distribution functions (CDFs) for the percentage error of the cost predictor for different migration distances under the aberrant behavior scenario, without budget enforcement. Migration distance is denoted by d; $d = 0$ reflects no migration, $d = 1$ denotes migrations to neighboring GPUs, and $d = 2$ reflects migrations to distant GPUs.

Obs. 4. The cost predictor is generally accurate at predicating token hold time, despite aberrant task behavior.

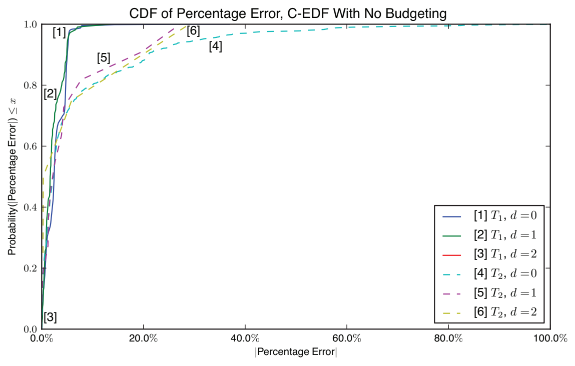
As seen in Fig. 7, under C-EDF (inset (a)), roughly 95% of all predictions for task T_1 have a percentage error of 5% or less (only one sample exists for $T_1, d = 2$). Accuracy is even better for T_1 under C-RM. This is expected since T_1 has a statically high priority due to its short period, and it is thus able to avoid more interference from other tasks than under C-EDF.

Obs. 5. The cost predictor is less accurate for tasks with longer execution times and periods.

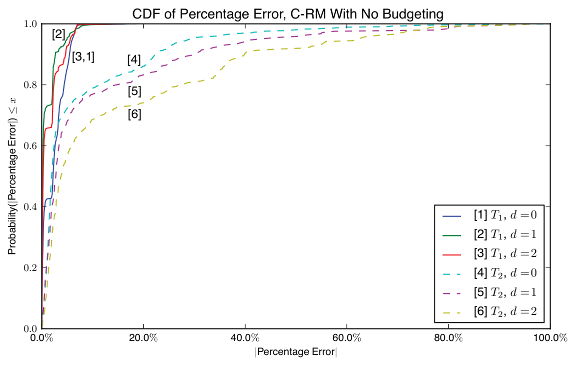
Both insets (a) and (b) of Fig. 7 show that the cost predictor is less accurate for T_2 than T_1 in this experiment. This is because T_2 has a longer execution time and period than most other tasks in the task set. Thus, T_2 is more likely to experience interference from aberrant tasks.

Obs. 6. The cost predictor is moderately less accurate under C-RM than C-EDF.

This can be seen by comparing insets (a) and (b) of Fig. 7. For example, about 90% of predictions for T_2 with $d = 1$



(a) C-EDF



(b) C-RM

Figure 7: Cumulative distribution functions of percentage error.

under C-EDF (inset (a)) have a percentage error no greater than 20%. Compare this to C-RM (inset (b)), where only 80% of $d = 1$ predictions for T_2 have the same degree of accuracy.

Obs. 7. The cost predictor is generally less accurate for longer migration distances.

A migrating job of a task must acquire additional copy engine locks and do more work than non-migrating jobs. This introduces variability into token hold times predicted by the cost predictor. We see that this generally has a negative effect on the cost predictor. This is clearly demonstrated in Fig. 7 for C-RM, where each CDF generally upper-bounds the CDF of the same task at the next migration distance. We note that this does not always hold under C-EDF. For example, predictions for T_2 with $d = 1$ are more accurate than $d = 0$. However, as we discuss shortly, this may be due to a smaller sample size of observations.

Migration frequency. The cost predictors are an important component of GPUSync, as they influence the behavior of the migration heuristics. Table 2 gives the total number of migrations observed under the aberrant scenario, with and without budget enforcement.

Obs. 8. Affinity-aware GPU assignment helps maintain affinity.

Under all scenarios, we see that tasks are significantly more likely to be assigned the GPU with which they have affinity. For instance, task T_1, under C-EDF with no budget enforcement, maintained affinity 11887 times, migrated to a neighboring GPU 110 times, and never migrated to a distant GPU. Similar trends are observed for all tasks. Without an affinity-aware method, migrations would be more frequent than reassignment since any task would have a 75% chance (with a GPU-cluster size of four) of being assigned to a different GPU.

	C-EDF				C-RM			
	No Budgeting		Budgeting		No Budgeting		Budgeting	
Distance	T_1	T_2	T_1	T_2	T_1	T_2	T_1	T_2
0	11887	2382	8866	2374	11255	1826	8586	1915
1	110	11	130	8	374	245	386	142
2	0	4	0	13	368	326	173	340
Total	11997	2397	8996	2395	11997	2397	9145	2397

Table 2: Observed migration counts and distances for T_1 and T_2.

Obs. 9. GPUSync is more successful at maintaining affinity under the C-EDF configuration.

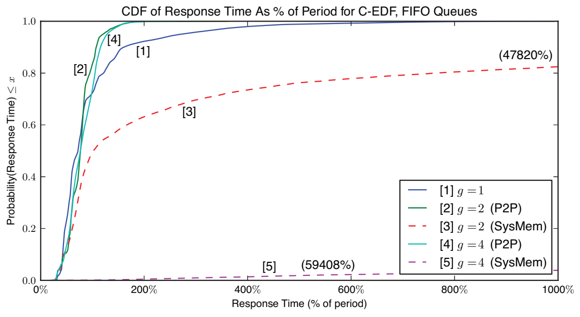
Observe in Table 2 that migrations are more frequent under C-RM than C-EDF. Indeed, distant migrations are practically eliminated under C-EDF. This behavior is attributable to the decreased cost-predictor accuracy under C-RM (recall Obs. 6). Inaccurate estimates can cause migrations to seem faster than waiting for the GPU for which the job has affinity. This observation demonstrates the importance of the cost predictor with respect to the runtime behavior.

5.2 Real-Time Vision Workloads

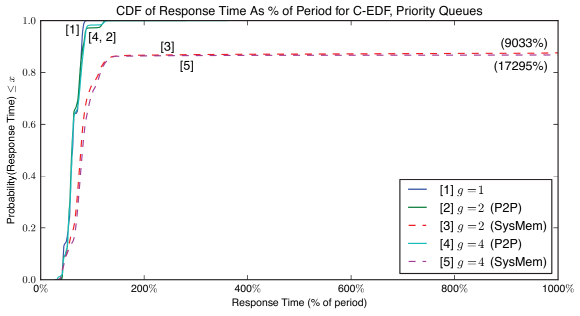
We now describe the vision-related experiments mentioned earlier. In these experiments, we adapted a freely-available CUDA-based feature tracking program to GPUSync on LITMUS$^{\mathrm{RT}}$ [16]. Feature tracking is an important application in automotive systems that sense and monitor the environment. 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 our evaluation platform by applying feature tracking to thirty independent video streams simultaneously. Each video stream is handled by one task, with each frame being processed by one job. The video streams have different execution-rate requirements: two high-rate streams run at 30 frames per second (FPS), ten medium-rate streams run at 20 FPS, and eighteen low-rate streams run at 10 FPS. The high- and medium-rate streams operate on video frames with a resolution of 320x240 pixels. Each job of these streams has roughly 1MB of combined input and output data and a state size of about 6.5MB. The low-rate streams process larger video frames with a resolution of 640x480. A low-rate stream requires four times as much memory as a higher-rate stream. Video 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.

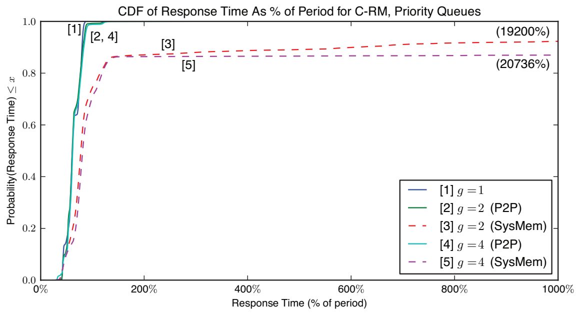
We tested the same configurations as in the prior experiments, with the addition C-EDF configured with a priority-ordered GPU allocator and engine locks. The token count was also reduced to $\rho = 2$ for all configurations—we found



(a) C-EDF, FIFO Queues



(b) C-EDF, Priority Queues



(c) C-RM, Priority Queues

Figure 8: Cumulative distribution functions of job response times as percent of period. Graphs are truncated for visual clarity.

this worked best for this data-heavy workload. The video streams were scheduled on three different GPU clustering configurations: eight GPU partitions ($g = 1$), four small GPU clusters of two GPUs ($g = 2$), and two large GPU clusters of four GPUs ($g = 4$). We tested the clustered configurations with both P2P and system memory migration. Tasks were partitioned evenly among the CPU and GPU clusters. Our video-stream workload was scheduled under each configuration for 120 seconds and measurements were taken.

Results. We analyze each GPUSync configuration by inspecting the response time of each job. We classify a configuration as resulting in an "unschedulable" system if any task consistently exhibits growth in response time (a sign of unbounded deadline tardiness).[12] We assume a schedulable system with bounded deadline tardiness, otherwise.

Fig. 8 plots the CDF of job response times under each configuration, expressed as a *percent of period*. Unschedulable configurations are denoted by dashed lines. We truncate the plots in order to show more detail for shorter response times. Truncated lines are annotated with the value (in parentheses) where the associated CDF reaches 1.0 (i.e., the observed worst-case response time). We make several observations from these results.

[12]We plan on investigating formal schedulability analysis in future work.

Obs. 10. Priority-ordered queues often provide improved observed response times.

This trend can be seen by comparing the curves in inset (a) to insets (b) and (c). The priority-ordered queues decrease the average-case blocking experienced by high-priority requests, i.e., those with urgent deadlines or short periods under C-EDF and mboxC-RM, repetitively. However, Wieder and Brandenburg [28] have recently shown that FIFO-ordered queues often improve schedulability. While schedulability analysis is outside the scope of this paper, we believe that GPUSync can be configured to use FIFO-ordered queues to optimize for schedulability, or priority-ordered queues to optimize for runtime behavior, highlighting GPUSync's flexibility.

Obs. 11. Clustered GPU management can outperform a partitioned approach.

This can be observed in lines [2] and [4] in inset (a) of Fig. 8. The CDFs of these clustered GPU configurations with P2P migration show that a job is more likely to have a shorter response time than jobs in the other FIFO-ordered configurations. Despite the extra load imposed by memory migrations, clustered GPU approaches can still outperform a partitioned GPU approach (line [1]). This is a positive result in light of the benefits FIFO ordering can have on schedulability—clustering may provide a viable FIFO-based alternative to a poorer performing partitioned approach.

Obs. 12. Small GPU clusters may be preferable over large GPU clusters.

In Fig. 8, we observe that small GPU clusters (lines [2] and [3]) generally yield shorter response times than large GPU clusters (lines [4] and [5]) in this experiment. For example, in inset (a) of Fig. 8, a cluster size of two with P2P migrations (line [2]) has the shorter response times than a cluster size of four (line [4]). This difference is attributable to the higher cost of migration between distant GPUs in the larger clusters. In each inset of Fig. 8, a cluster size of two with system memory migrations (line [3]) has shorter response times than a cluster size of four (line [5]), despite an equal migration distance to system memory. Large clusters perform poorly due to an increased chance of migration.

Obs. 13. GPU state migration through system memory is often too costly.

We observe from Fig. 8 that no configuration that used system memory migration resulted in a schedulable system in our experiment. Indeed, partitioned GPU management is preferable to all such system-memory-migration configurations. This is clearly reflected by lines [3] and [5] in each inset, where the observed worst-case response times are over 9,000% of task period.

6 Conclusion

In this paper, we presented GPUSync, a lock-based GPU management framework for real-time multi-GPU systems. Unlike prior research, GPUSync takes a synchronization-based approach to GPU scheduling that simultaneously promotes affinity among computational tasks and GPUs, and fully exposes the parallelism offered by modern GPUs. While others have approached GPU management from a scheduling perspective, GPUSync's synchronization-based techniques allows it to be "plugged in" to a variety of real-time schedulers. We reported on our efforts of implementing GPUSync in LITMUS$^{\mathrm{RT}}$, and have given evidence, through empirical evaluations, that GPUSync can successfully meet the resource allocation needs of a real-time GPGPU system. We further show that execution behaviors can be predicted to promote task affinity in multi-GPU systems. We also demonstrated that efficient clustered GPU management, enabled by peer-to-peer communication between GPUs, can outperform partitioned approaches in some GPUSync configurations.

This paper has focused on the design and implementation of GPUSync. In a future paper, we will analytically model the real-time predicability of GPUSync and examine the trade-offs of different GPUSync configurations in terms of schedulability analysis. This work is already underway and nearing completion.

Considering the broader area of research, GPUSync does not support systems with different types of GPUs, i.e., heterogenous GPUs. A particularly interesting scenario to study would be the simultaneous support of both integrated and discrete GPUs. If these GPUs were to be managed in a clustered fashion, then online mechanisms would have to balance the relatively weak computational power of integrated GPUs against the data communication cost of discrete GPUs. Optimizing the solution for power consumption would add yet another dimension of complexity.

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] B. Brandenburg. Virtually exclusive resources. Technical Report MPI-SWS-2012-005, Max Planck Institute for Software Systems, 2012.

[4] B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *19th RTAS*, 2013.

[5] B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, 2012.

[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] 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.

[12] S. Kato, J. Aumiller, and S. Brandt. Zero-copy I/O processing for low-latency GPU computing. In *4th ICCPS*, 2013.

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

[14] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX ATC*, 2011.

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

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

[17] M. Kim and B. Noble. Mobile network estimation. In *7th MobiCom*, 2001.

[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] G. Lamastra, G. Lipari, and L. Abeni. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *22nd RTSS*, 2001.

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

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

[22] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *10th Distributed Computing Systems*, 1990.

[23] R. Rajkumar, S. Lui, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *9th RTSS*, 1988.

[24] 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.

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

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

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

[28] A. Wieder and B. Brandenburg. On spinlocks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *34th RTSS*, 2013.

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

A Scheduling Closed-Source Threads

At the end of Sec. 3, we discussed several thread dependancies that can exist in today's GPGPU systems. A GPU-using real-time task can be dependent upon the timely execution of both GPU interrupt handling threads and user-level threads created by a closed-source GPGPU runtime. A detailed discussion of our general approach to GPU interrupt thread scheduling is available in [9], with necessary enhancements to support GPUSync described in Sec. 3 of this paper. However, we wish to provide more detail on how user-level threads found in a closed-source GPGPU runtime can be scheduled with real-time priorities. We limit the specifics of our discussion to the CUDA GPGPU runtime from NVIDIA, but we believe out general techniques use can work with other runtimes as well.

Starting in CUDA 4.2, the CUDA runtime creates one user-level worker thread for each GPU used by each program instance (process). NVIDIA does not disclose what computations their worker threads perform, but through experimentation, we have found that they are responsible for waking GPU-using threads that suspend while waiting for GPU operations to complete.[13] Clearly, these worker threads must be scheduled with an appropriate priority to avoid priority inversions. However, applying a scheduling policy to these tasks is challenging since they are created by closed source software.

We make no assumption of *when* the CUDA runtime creases worker threads. We also assume that we do not know which worker threads are associated with which GPUs. However, for our discussion here, let us assume that there is only one real-time task per process, and it is represented by the process's *main thread*. As described in Sec. 3, the base priority of a worker thread is statically less than any normal real-time task in the system. However, the worker threads within a process may inherit the effective priority of process's main thread when the main thread suspends while holding an engine lock. This effective priority may be the base priority of the main thread or even a priority inherited through locking protocols. The single threaded sporadic task model is preserved in this implementation, since a real-time task may only use one GPU at a time—only one worker thread may have work to do at any given moment. Thus, no two worker threads inheriting the same priority will execute simultaneously.

The scheduling policy of worker threads in LITMUS^{RT} is controlled through two systems calls: `enable_aux_rt_tasks()` and `disable_aux_rt_tasks()`. LITMUS^{RT} discovers and tracks CUDA's worker threads through these system calls that are called by regular real-time tasks. `enable_aux_rt_tasks()` may take a combination of two flags as parameters: `CURRENT` and `FUTURE`. When `enable_aux_rt_tasks(CURRENT)` is called, LITMUS^{RT} iterates over the list of all threads within the process of the thread that issued the system call. The scheduling policy of all *non*-real-time threads found within the process are changed to an "auxiliary real-time task" policy. Auxiliary real-time tasks follow the scheduling policy described in the paragraph above. When `enable_aux_rt_tasks(FUTURE)` is called, then any new threads created within the process in the future automatically assume the auxiliary real-time task policy. `CURRENT` and `FUTURE` may be combined in a single call to `enable_aux_rt_tasks()`. `disable_aux_rt_tasks()` also takes the flags `CURRENT` and `FUTURE`. `disable_aux_rt_tasks(CURRENT)` restores the default Linux scheduling policy to all current auxiliary tasks. `disable_aux_rt_tasks(FUTURE)` disables the behavior that automatically applies the auxiliary real-time task policy to threads created in the future. `CURRENT` and `FUTURE` also may be combined in a single call to `disnable_aux_rt_tasks()`.

Earlier we assumed that each process only contains one real-time task, represented by the main thread. This is not strictly necessary in actuality. LITMUS^{RT} allows more than one real-time task to execute within a process. This may be desirable in some real-time applications where a shared address space aids in efficiency. This is also supported by our implementation of GPUSync: In a process that contains more than one real-time task, auxiliary threads inherit the *maximum* priority of any suspended engine-holding real-time task within their process. This ensures that no priority inversions due to auxiliary tasks are *unbounded*. However, it remains possible for an auxiliary task to ultimately perform work for a lower-priority real-time task while the auxiliary task inherits a higher priority from another. This is a bounded priority inversion that should be accounted for in schedulability analysis. This may be done using the same techniques developed for interrupt accounting [9].

[13]These worker threads are also used to execute stream callbacks in CUDA 5.0. (See CUDA's `cudaStreamAddCallback().`)