GPUSync: Architecture-Aware Management of GPUs for Predictable Multi-GPU Real-Time Systems *

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 was designed with the goal of increasing parallelism in mind. GPUSync can be applied in multi-GPU real-time systems, is cognizant of the system bus architecture and affinity among computational tasks and GPUs, and fully exposes the parallelism offered by modern GPUs, even when closed-source GPU drivers are used. In empirical evaluations presented herein involving real-world applications, GPUSync improved real-time response times by three times or more, on average, making previously unschedulable workloads schedulable.

1 Introduction

Graphics processing units (GPUs) are commonly used today to accelerate intensive general-purpose computations, a practice termed GPGPU. In many domains, the parallel processing architecture of GPUs can be exploited to speed up computations by orders of magnitude in comparison to even modern multicore processors. The breadth of such domains is now expanding to include many applications in which *real-time* constraints exist. In automotive systems, for example, compelling use cases for real-time GPUs include eye tracking [18], pedestrian detection [25], navigation [13], and obstacle avoidance [23]. If such features are consolidated onto a single platform, then multiple GPUs may be required. Unfortunately, ensuring real-time constraints while making full use of the available parallelism offered by GPUs is a challenge, particularly in a system with multiple GPUs. This paper is directed at addressing this challenge.

In GPU-enabled real-time systems, parallelism must be dealt with at many levels. For example, data must be transmitted to a GPU before computation begins; it would be desirable for GPU transmissions and computation to 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 carefully managed or *all* computations in the system may be slowed [21].

In a multi-GPU platform, opportunities for parallelism are increased. For example, on such a platform, it may be desirable to avoid statically assigning computational tasks to GPUs 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 the particular GPU. This state must be *migrated* each time a task uses a GPU different from the one it used previously. In addition to directly affecting the tasks involved, 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 addressed these parallelism-related issues only partially. Furthermore, issues unique to multi-GPU systems have received very little attention.

Contributions. In this paper, we present GPUSync, a complete real-time GPU management framework that is cognizant of system architecture and task GPU affinity, and fully exposes the parallelism offered by modern GPUs. In most prior work on real-time resource allocation for GPUs, the focus has been primarily on *scheduling* (e.g., see [2, 8, 15, 16] and the references therein). This reflects the viewpoint that GPUs are schedulable entities, like CPUs. Our viewpoint is different: we view the management of GPUs as a *synchronization* problem. This viewpoint reflects the reality that the scheduler within the operating system (OS) executes only on CPUs and any GPU is an additional processor that is treated as an I/O device (even in on-chip architectures).

GPUSync utilizes recently proposed optimal locking protocols [27] to manage GPU-related resources. This synchronization-based technique allows it to be "plugged in" to a variety of real-time schedulers that support the underlying locking protocols. Unlike some prior approaches

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

by others, GPUSync may be used with closed-source GPU drivers, which usually offer better performance and are more feature-rich than open-source alternatives. Our contributions include:

- A system-architecture-aware real-time multi-GPU management framework based on real-time locking protocols.
- Auto-tuning heuristics for maintaining GPU affinity that also reduce the costs of GPU migrations.
- Real-time support for direct GPU-to-GPU migrations that isolates bus traffic and improves overall performance.

We also discuss an implementation of GPUSync in UNC's Linux-based LITMUS$^{\mathrm{RT}}$ real-time OS [7], and present an evaluation of it involving computer vision feature tracking computations (such as those found in automotive applications). In these evaluations, GPUSync improved real-time response times by three times or more, on average.

The rest of this paper is organized as follows. In Sec. 2, we provide necessary background. In Sec. 3, we review relevant prior work. In Sec. 4, we describe GPUSync. In Sec. 5, we evaluate our implementation of GPUSync in LITMUS$^{\mathrm{RT}}$. We conclude in Sec. 6. Due to space limitations, we limit attention to GPU technologies from NVIDIA, whose CUDA [1] platform is widely accepted as the leading GPGPU solution. However, our techniques may also be used with GPUs from other manufactures.

2 System Architecture

We begin by describing the underlying hardware architecture of a large GPU-enabled system, which was used in the experiments in Sec. 5. Much of this paper focuses on efficient GPU memory management, so we pay special attention to the various interconnects spanning system memory and GPUs. While we focus on large GPU-enabled systems, architectures of smaller systems are generally subsets of larger systems.

Architectural Description. Fig. 1 depicts the high-level architecture of a large-scale GPU-enabled system. There are four major types of components: system memory, multicore processors,[1] I/O hubs, and GPUs. These components are connected through three different bus interconnects: the memory bus, the high-speed processor interconnect, and the PCIe bus.

CPUs in a multicore processor are connected to system memory by an on-chip memory controller that attaches directly to the memory bus. Though the memory bus may be made up of several independent channels, we must treat them together as a single bus because memory addresses are usually finely interleaved across them to facilitate parallelism. We do not believe there is a way to programmatically isolate traffic to individual channels with such fine interleaving.

The multicore processors are linked through high-speed interconnects.[2] These interconnects are full-duplex, so data may travel in both directions simultaneously at full speed. On the multicore 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 multicore 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 supports two independent links. Each link attaches to a PCIe switch that multiplexes two additional links to increase the number of supported GPUs. Thus, the PCIe bus is organized hierarchically. The GPUs attach to the switches.[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. Devices can also transmit directly to each other if they share a switch or I/O hub.

At the highest-level, components are partitioned into non-uniform memory access (NUMA) nodes. Depicted in Fig. 1, each NUMA node includes a pool of system memory, a multicore processor, an I/O hub, and a collection of GPUs.[5] Memory and device access across nodes is supported, but this requires an additional "hop" over the high-speed interconnect, increasing latency and reducing effective throughput since the additional link may carry other traffic. Further, the memory controller on each processor may allocate less bandwidth to remote requests [19]. Thus, memory traffic should be localized within a node whenever possible.

GPU Architecture. As seen in Fig. 1, 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 is made up 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. Most GPUs only have one copy engine, and thus may only send or receive data at a given instant, despite the fact that PCIe is dual-simplex. However, higher-end GPUs may include an

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

[2] Common interconnect types include QPI and HyperTransport for Intel and AMD systems, respectively.

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

[4] Additional switches can be added to the PCIe network to support more I/O devices. For example, two GPUs may be packaged on a single daughterboard that includes an additional onboard PCIe switch.

[5] On larger systems, additional nodes may not have an I/O hub. On smaller systems, nodes may share a single I/O hub. Even smaller systems, such as a desktops and laptops, may have only a single node and a single GPU.

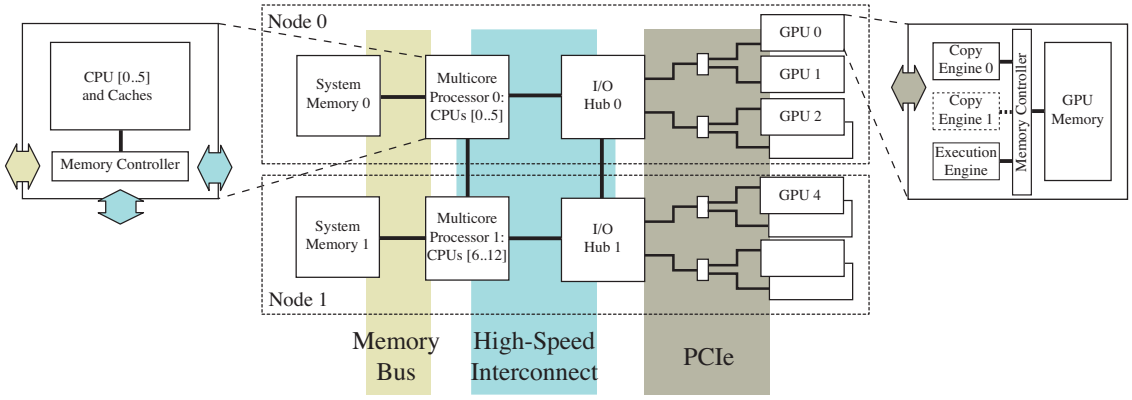

Figure 1: System architecture of a large-scale multi-GPU platform.

additional copy engine, enabling simultaneous bi-directional transmissions.

The execution engine and copy engines operate independently—a copy engine may transmit data while the execution engine performs computation. Also, some modern GPUs have the capability to transmit directly to each other, bypassing system memory. We leverage both of these aspects in developing GPUSync.

GPGPU Operations and State Migration. GPGPU programs execute on CPUs and issue commands to GPUs to perform operations. These operations primarily involve copying data between system and GPU memory and invoking GPU programs, called *kernels*, which run on a GPU. A GPGPU program typically performs the following sequence of operations per GPU access: (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 operation, 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. 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. 1, the distance between GPUs 0 and 2 is two since they share the I/O hub, but not switches. Migrations using direct GPU-to-GPU memory copies, especially over short distances, are fast due to proximity and potentially reduced bus contention since less distance is traveled. Unfortunately, direct GPU-to-GPU copies are not possible between I/O hubs (i.e., different nodes), so cross-node migrations must pass through system memory.

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. Observe in Fig. 1 that data copied between system and GPU memory, within the same NUMA node, traverses three interconnects, two memory controllers, and one PCIe switch. These elements are *shared* by concurrently executing operations. The speed of a data copy between system and GPU memory 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 [21]. 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, [21] does not address issues of GPU allocation and affinity. 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) fully expose 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

We now examine prior approaches to GPU resource management and consider how well each addresses issues relating to system architecture and real-time predictability. PTasks [22], developed by Rossbach et al., creates a new OS-level infrastructure of GPU management. PTasks uses dataflow graphs to reduce the amount of data that is transmitted between system and GPU memory. This allows PTasks to individually schedule execution and copy engines. PTasks also 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, the heavy use of parameter-tuned heuristics and migrations through system memory results in idle GPUs and increased bus contention. Finally, although PTasks is designed for interactive applications, it uses heuristics that are not amenable to real-time analysis.

RGEM is a user-space real-time GPU scheduler designed by Kato et al. [15]. Unlike PTasks, RGEM schedules GPU operations by static priority, so the system better maintains real-time predictability. RGEM also 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. However, RGEM does not individually schedule GPU execution and copy engines, so opportunities for parallelism are lost. Further, RGEM is designed to support only a single GPU.

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]. Unlike RGEM, Gdev separately schedules the execution and copy engines of a GPU. However, Gdev remains focused on single-GPU systems and provides no mechanisms for GPU allocation or migration support. Also, Gdev requires open-source software drivers, which can be a serious limitation since these drivers lag behind vendor-provided ones with respect to performance, available features, and support for the most recent GPUs.

In our own work, we have developed GPU-enabled real-time systems by viewing GPUs as shared resources, rather than directly schedulable processors, which is the perspective taken in the above work. In our approach, GPU access is arbitrated by a real-time locking protocol [12]. This has allowed us to integrate closed-source GPU drivers while still addressing low-level issues like real-time interrupt handling [11]. It has also allowed us to explore the properties of GPU-enabled real-time systems with schedulers such as clustered earliest-deadline-first (EDF) (work by others has focused on static priority scheduling). This has enabled support for soft real-time (SRT) systems that allow greater utilizations than would be possible with static priority schedulers [4]. Finally, our approach enables support for multi-GPU platforms through the use of k-exclusion locking protocols [9, 11].[6] However, our work thus far is not without its own limitations. Though we have supported multi-GPU platforms, no consideration for affinity has yet been made. Also, our prior work has treated GPU execution and copy engines as a single unit. We remedy these shortcomings in this paper.

4 GPUSync

GPUSync's design reflects several assumptions. First, the underlying multicore platform may have several GPUs. Second, the workload to be supported can be modeled as a traditional sporadic real-time task system, where real-time computations by a single task are recurrent; we call these recurrent invocations *jobs*. Third, jobs are scheduled in priority order and each job has an absolute deadline; our system supports job-level static priority schedulers, such as EDF. Fourth, we target SRT systems where the time by which a scheduling deadline is missed is bounded.[7] We schedule CPUs in clusters (one node in Fig. 1 per cluster) since this method has been shown to offer superior SRT capabilities [4]. Fifth, we assume that each job only needs to be *allocated* a GPU once. Finally, we assume that any changes in per-job execution times of the same task are gradual, facilitating execution time predictions based upon historical data.

GPUSync is made up of several components: a self-tuning execution *cost predictor*; a *GPU allocator*, based upon a real-time k-exclusion locking protocol, augmented with heuristics; and a set of real-time *engine locks*, one per GPU engine, to arbitrate access to the functional units of the GPU. GPUSync also provides *API routines* that integrate with our system to facilitate real-time direct GPU-to-GPU migrations and memory copies in order to reduce blocking times and maintain real-time predictability.

The high-level design of GPUSync is illustrated in Fig. 2. We describe the flow of a request for a GPU and its resources by a job to demonstrate how GPUSync functions. A request is issued to the GPU allocator in Step A. Utilizing the cost predictor (Step B) and internal heuristics, the GPU allocator determines which GPU should be allocated to 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 C. In Step D, the job competes with other jobs allocated the same GPU for

[6] k-exclusion extends ordinary mutual exclusion (mutex) by allowing up to k tasks to simultaneously hold a lock.

[7] We do not feel that current GPU technology is able to support strict hard real-time constraints (those were deadlines can never be missed under any circumstances) [10]. For example, synchronization mechanisms internal to closed-source software can conflict with the request deferral techniques of priority-ceiling-based locking protocols, such as the Stack Resource Policy [3], common to hard real-time systems. Note, however, this is only one of many challenges of integrating GPUs into hard real-time systems.

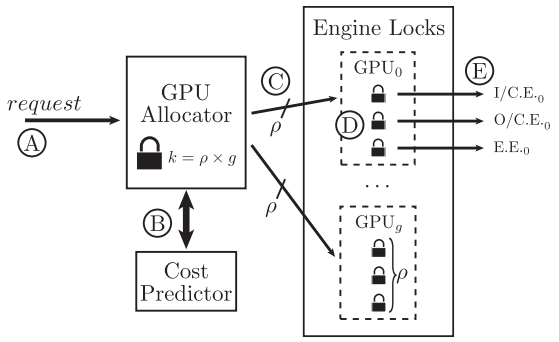

Figure 2: High-level design of GPUSync.

GPU engines; access is arbitrated by the engine locks. A job may finally access the engine on its assigned GPU once the corresponding engine lock has been acquired in Step E.

We now describe these components in more detail, as well as the APIs that integrate with GPUSync to improve performance, predictability, and ease programming.

4.1 Cost Predictor

GPUSync's GPU allocation heuristics require the ability to estimate costs due to migrations. The cost predictor is used to estimate the time a job will execute, including migration costs, if it is allocated a given GPU. The predictor uses online monitoring of job execution patterns to make estimates based upon past behavior.

The cost predictor begins measuring the accumulated execution time of a task once it has been allocated a GPU. Execution time is accumulated as the job is scheduled on a CPU. Execution delays due to preemption and blocking due to engine lock acquisition (explained later) are *not* included. However, CPU suspension durations due to GPU operations (memory copies, GPU kernels) *are* included in the accumulated time. When a job releases a GPU, this measurement gives a total job execution cost for both CPU and GPU operations and includes any delays due to migrations. This measurement is fed into a feedback control mechanism that is used to estimate future execution costs.

In order to make the cost predictor sensitive to migration costs, we actually use several feedback controllers per task. The predictor records the distance between the newly allocated GPU and the GPU that the task used previously (no record is made in the initial case) whenever a job is allocated a GPU. This distance *classifies* the type of migration that the job will experience. Each classification is given a unique feedback controller. Using the cost predictor, the GPU allocator's heuristics can estimate execution costs based upon migration distance and use this information to maintain GPU affinity and reduce bus contention.

4.2 GPU Allocation and Engine Access

As previously discussed, locking protocols can be employed to arbitrate access to GPUs. We develop a lock-based approach to GPU arbitration that allows the underlying architecture of the system to be more effectively utilized.

When a single k-exclusion lock is used to arbitrate access to g GPUs ($k = g$), a job implicitly has exclusive access to a GPU's execution and copy engines when it is allocated that GPU. In a finer-grained approach, if separate k-exclusion locks are used to allocate engines, then a job could be assigned engines within mismatched GPUs. This is a nonsensical assignment since the sequence of GPU operations described in Sec. 2 must be carried out on the same GPU. Consequently, a more sophisticated locking scheme is necessary to first perform GPU allocation in one phase, followed by real-time arbitrated engine access in another. This is exactly what we do: GPU allocation is performed using a k-exclusion locking protocol and engine access is arbitrated by mutual exclusion locks. GPU allocation is actually done on a per-node basis, i.e., tasks executing on a given CPU can only access GPUs that share the same I/O hub. Thus, we require a k-exclusion locking protocol for globally scheduled systems, where k is set to the number of GPUs in one of our clusters. The overall locking strategy we employ is derived from the *real-time nested locking protocol* (*RNLP*), which has been shown to be optimal for supporting nested resource requests in globally scheduled real-time systems [27].

GPU Allocator. We explain how GPU allocation is done by considering a single cluster (or effectively, a globally scheduled system) with g GPUs. GPU allocation is token-based. We associate ρ *GPU tokens* with each GPU. A job must acquire a GPU token before it can compete for access to GPU engines. All GPU tokens are pooled and managed by a *single* k-exclusion lock, $k = \rho \times g$. The number of jobs that may compete for GPU engines for any single access on a particular GPU is a function of ρ. We set ρ equal to the number of engines per GPU.[8]

We use a modified version of the Replica-Request Priority Donation 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 [28]. The R^2DGLP supports job-level static priority schedulers, so GPUSync may be "plugged in" to any scheduler that supports the R^2DGLP. Out modified R^2DGLP respects the priority inheritance rules of the original protocol, preserves real-time blocking correctness, and continues to support nested requests. Due to space constraints, we cannot fully describe the R^2DGLP and only give a brief overview here, in addition to our heuristic modifications.

[8]Greater values of ρ can increase parallelism on a single GPU, but this negatively affects real-time schedulability and lead to idle GPUs system-wide. Please see the appendix of the online version of this paper, located at `www.cs.unc.edu/~anderson/papers.html`, for analysis.

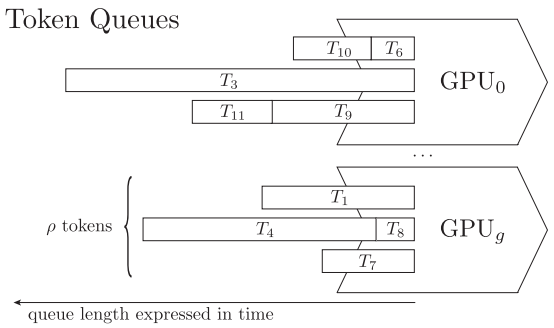

Figure 3: R^{2}DGLP token queues populated with requests. Queue length is measured in units of time.

Access to each token in the R^{2}DGLP is arbitrated by a per-token FIFO-ordered queue. When jobs issue token requests, they are enqueued in a token queue and suspend from execution until they are granted a token. The job at the head of each token queue holds the corresponding token. This is depicted in Fig. 3 where the token queues have been populated with requests. Observe that ρ tokens map to one GPU. The standard R^{2}DGLP specifies that a request is enqueued on the queue with the least *number* of requests. However, we modify the R^{2}DGLP to enqueue requests on the queue that gives the *earliest estimated completion time* of the request. This requires that queue length be measured in terms of *time*, not number of requests. Each enqueued request is weighted by the estimated time it will take for that request to complete on the GPU the queue is associated with. Thus, one token queue may hold more pending requests than another, but be shorter with respect to the estimated time it will take to execute pending requests. This is illustrated in Fig. 3, where the first token queue for GPU_0 contains two requests, but is shorter than the first token queue of GPU_g, which contains only one request.

The R^{2}DGLP uses the cost predictor to estimate the execution costs that will be incurred if a job with a GPU request is allocated to a particular GPU. When a job requests a GPU token, the job is not enqueued on the shortest queue, but rather on the queue that will allow the job to complete the soonest (which is a function of both queue lengths and expected migration costs).

An example of the heuristic is illustrated in Fig. 4. A task T_2 has affinity with GPU_0 and requests access to a GPU. The cost predictor estimates a cost for T_2 on GPU_0 and GPU_1; due to affinity, the cost is less on GPU_0. The R^{2}DGLP examines the current state of the token queues. The shortest queue for GPU_1 (the third queue for GPU_1) has an estimated length of t_1. The shortest queue for GPU_0 (the first queue for GPU_0) has an estimated length of t_2 (greater than t_1). However, if T_2 is enqueued on the queue of GPU_1, then the estimated completion time for T_2 is t_4, which is greater than t_3, the estimated completion time on GPU_0. T_2 is therefore enqueued on the token queue for GPU_0.

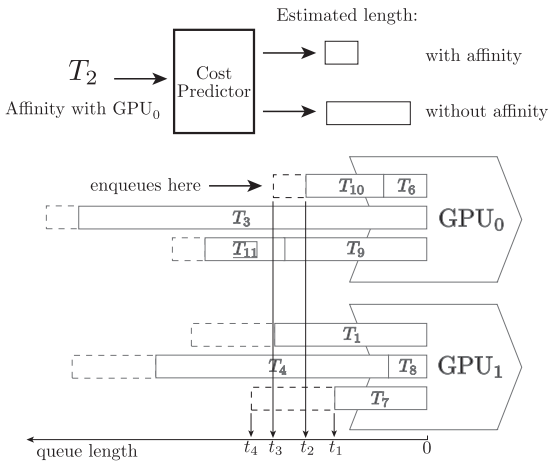

Figure 4: Request enqueues on the queue with the earliest estimated completion time.

This heuristic may cause a job to wait for an available GPU token even when one is available for another GPU for which migration costs are greater. However, this only occurs when the costs of executing on the free GPU is anticipated to lead to degraded performance.[9]

To ensure optimal blocking behaviors, the issuance of token requests may be deferred under the R^{2}DGLP to limit the total number of requests in the token queues. The rules governing these deferred requests are out of the scope of this paper. However, we note that we further modify the R^{2}DGLP with additional heuristics that promote affinity.

To summarize, through the use of heuristics, the R^{2}DGLP efficiently allocates GPUs to jobs while maintaining real-time blocking correctness and real-time predictability.

Engine Locks. A mutex is associated with each GPU copy and execute engine, as depicted in Fig. 2. For GPUs with *two* copy engines, one copy engine is designated as *inbound* and the other as *outbound*. A job acquires the inbound copy engine lock if data is to be transmitted to the GPU. The outbound copy engine lock is acquired if data is to be transmitted from the GPU. We differentiate inbound and outbound copy engines because we see little value in allowing two jobs to transmit data in the same direction on the same device since their effective bandwidth is halved due to PCIe bus arbitration protocols.[10] It is better to instead promote simultaneous full-speed transmissions on the dual-simplex PCIe bus. However, for a machine like our evaluation platform, which has *one* copy engine per GPU, we are not able to take advantage of this parallelism; for such a machine, the inbound and outbound locks are one in the same.

Each engine mutex prioritizes requests in FIFO order.

[9]This is unlike PTasks, which uses hard-coded parameters to make migration decisions without examining current system state.

[10]The PCIe *specification* supports real-time arbitration through static priorities [20]. However, these features are not implemented by even mainstream platforms [14].

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^2DGLP 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 complete. In order to reduce worst-case blocking, a job is allowed to hold at most *one* engine lock at a time, *except* during direct GPU-to-GPU migrations, which requires holding a copy engine lock for each GPU.

The use of engine locks enables the parallelism offered by GPUs to be utilized while simultaneously obviating the need for the GPU driver to make resource arbitration decisions, enhancing real-time predictability.

4.3 GPU-to-GPU Migrations

As stated earlier, we only allow a job to hold two engine locks when it must perform a direct GPU-to-GPU migration. When migrating from GPU_a to GPU_b, a job must hold the outbound copy engine lock of GPU_a and the inbound copy engine lock of GPU_b. We explore two methods of acquiring these locks: nested resource requests [28] and *dynamic group locks* (*DGLs*) [26]. Using nested resource requests, a job requests and acquires one copy engine lock before issuing a request for the other. Thus, a job may suspend twice, being blocked on each request, before acquiring both locks. These requests must also be issued according to a specified partial order to avoid deadlock. Under DGLs, a job atomically requests *both* copy engine locks and does not resume execution until it has acquired them. This method may reduce the number of context switches since fewer suspensions are possible. Also, requiring a specified partial ordering of requests is not necessary since requests for both locks are issued atomically.

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

4.4 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 with the introduction of copy engine locks.

Chunking is implemented through user-space memory copy APIs. The caller specifies the appropriate data pointers, the amount of data to be copied, the chunk size to use, and type of copy to be performed (system-to-GPU, GPU-to-system, or GPU-to-GPU). Our APIs perform copies incrementally in appropriately sized chunks. The necessary copy engine locks (determined from the source and destination of the copy) are acquired automatically before each chunk is transmitted. For GPU-to-GPU copies, locks are acquired either using nested requests or DGLs, depending upon system configuration. Copy engine locks are released after the chunked copy has completed. Thus, a job will repetitively acquire and release copy engine locks when performing copies greater than the specified chunk size.

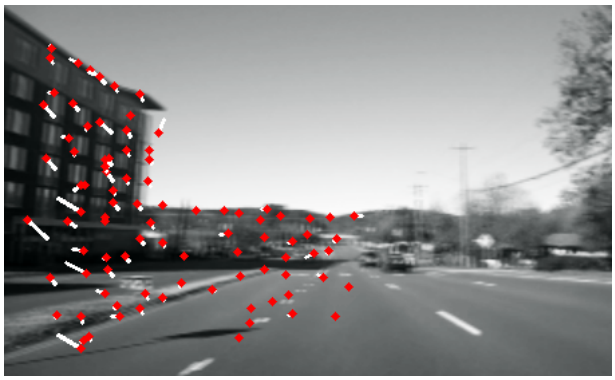

Figure 5: Graphical depiction of feature tracker output.

In theory, GPUSync facilities efficient and predictable utilization of system resources. However, an evaluation is necessary to see if there are real benefits to the approach. This is examined next.

5 Evaluation

In this section, we evaluate our implementation of GPUSync in LITMUS^{RT}, currently based upon Linux 3.0.0. Motivated by the automotive applications given in Sec. 1, we evaluate our system with the real-world computer vision application of feature tracking. Feature tracking can provide input to higher-level applications such as object tracking.

Feature Tracking. We adapted a freely available CUDA-based implementation of a Kanade-Lucas-Tomasi (KLT) feature tracker [24]. Fig. 5 gives a graphical representation of a tracker's output: salient features (points) in video are identified and their motion (lines) is tracked between frames. This implementation incorporates inertial sensor data [17] to enhance results; this is particularly attractive in systems where video cameras are mounted on moving vehicles. We made infrastructural enhancements to the original implementation to support multi-GPU platforms (i.e., migrations) and integrate with LITMUS^{RT} and GPUSync.[11] The tracker represents a scheduling challenge since it utilizes

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

both CPUs and GPUs to carry out its computations. Though the KLT tracker represents only one GPGPU application, its image processing operations are emblematic of many others.

Evaluation Platform. Our test platform is a NUMA system like that depicted in Fig. 1. It has two Xeon X5060 processors with six 2.67GHz cores each and is equipped with eight NVIDIA GTX-470 GPUs. We used clustered EDF scheduling for reasons discussed earlier in Sec. 4. We used CUDA 4.0 for our GPGPU runtime environment.[12]

Experimental Setup. We performed feature tracking on 20 independent video streams simultaneously. Eight video streams were processed at 20 frames-per-second and 12 streams were processed at a slower rate of ten frames-per-second (in a vehicle setting, different video streams may execute at different rates, depending upon importance). Each video stream was handled by one task, with each frame being processed by one job. Jobs were scheduled with implicit deadlines commensurate with stream frame rate. Tasks were evenly partitioned between the two NUMA nodes of the system (ten streams apiece).

Lacking a real vehicle testbed, we processed pre-recorded video streams that were 320x240 in resolution. Frames were preloaded into memory before processing commenced in order to avoid disk latencies (there would not be such latencies with real video cameras). All data was page-locked in system memory to facilitate fast and deterministic memory operations (including system memory-based migrations). Memory copies for input and output data between system and GPU memory was roughly 1MB per frame, combined. However, task state data was about 6.5MB in size.

The video streams were processed under various system configurations determined by a combination of the following parameters: with and without GPU affinity allocation heuristics; nested engine lock requests or DGLs; $\rho \in \{1, 2\}$ (our GPUs have only one copy engine); migrations via system memory or through GPU-to-GPU copies; and chunk sizes of 2MB, 4MB, and with chunking disabled. Each meaningful combination of parameters resulted in a test scenario (e.g., using engine locks when $\rho = 1$ is a meaningless test).

In addition to these GPUSync tests, we tested scenarios that relied upon the throughput-oriented, closed-source, GPU driver to arbitrate GPU access. A simple load-balancer was used to perform GPU allocation. Thus, jobs were allocated GPUs without blocking and then competed for engine access within the driver with no real-time guarantees. Chunk sizes and migration methods were tested under these scenarios.

Each task processed eight minutes worth of video. Measurements of job response times were made within a *five minute window* starting after 30 seconds of execution (this delay allows the cost predictor to "warm up"). We measured each job's response time starting from the latter of its release time and its predecessor job's completion. (If a job's predecessor is tardy, then its own release is not altered; however, jobs of the same task must execute in sequence.) Response times, as a *percent of period*, were averaged for each scenario.

[12]Unfortunately, the current version of LITMUS$^{\mathrm{RT}}$ cannot guarantee real-time delivery of messages from the OS to the CUDA runtime in user space with CUDA 4.1, the most recent version of CUDA, due to subtle changes in its implementation. We are currently extending LITMUS$^{\mathrm{RT}}$ to accommodate these changes.

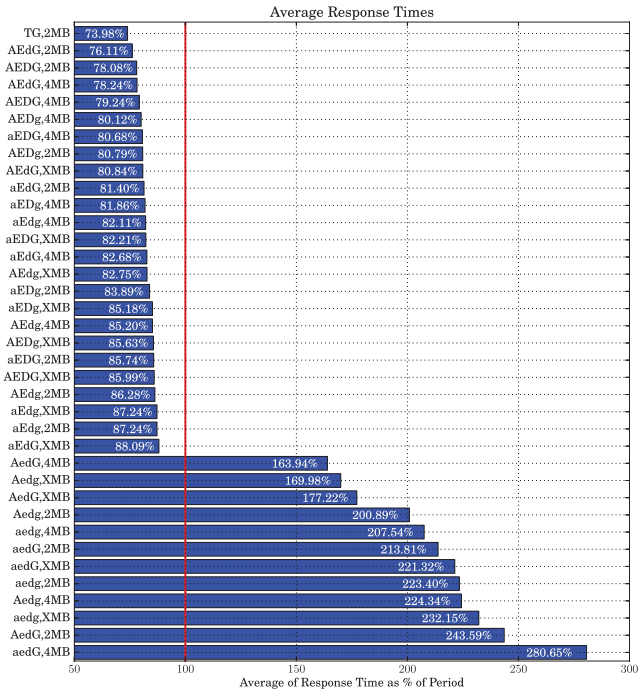

Figure 6: Average response time as percent of period under each configuration.

Results. A large number of different scenarios were tested. We denote scenarios in the following manner: the letter A (a) denotes affinity was enabled (disabled); E (e) denotes that engine locks were (not) used; D (d) denotes that DGLs were (not) used; G (g) denotes that migrations were performed via GPU-to-GPU (system) memory copies. Finally, each scenario designation is followed by a chunk size (X denotes that chunking was disabled). Thus, AEDg,2MB denotes the scenario where affinity heuristics and engine locks with DGLs were used, with migrations passing through system memory, and all memory copies were broken into 2MB chunks.

Fig. 6 depicts the average job response time, as a percent of period, under each GPUSync scenario, sorted from shortest to greatest. The best performing scenario utilizing the throughput-oriented driver (GPU-to-GPU migrations with 2MB chunking) is also included and labeled as TG,2MB. Any scenario with an average left of the vertical line at 100% on the x-axis is said to be SRT schedule.

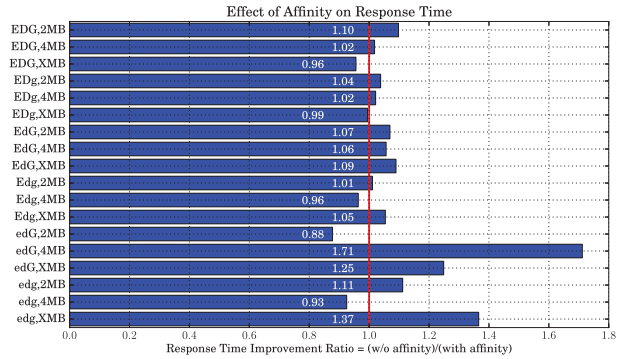

Figure 7: Effect of affinity heuristics under each scenario.

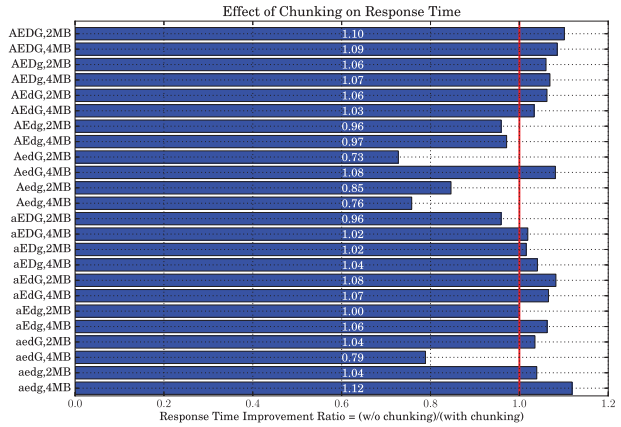

Figure 8: Effect of chunking under each scenario.

Observation 1. *The use of affinity heuristics, engine locks, direct GPU-to-GPU migrations, and chunking improved average response times by three times or more.*

The scenario AEdG,2MB gave the best average response time of 76.11%. In contrast, aedg,XMB, which is roughly equivalent to our prior approaches, had an average response time of 232.15%.

Observation 2. *The simultaneous use of execution and copy engines dramatically improved performance. A scenario was schedulable iff engine locks were used.*

There is a clear break in the response times in Fig. 6. The worst performing schedulable scenario was aEdG,XMB with an average response time of 88.09%. The next best scenario was AedG,4MB with an average response time of 163.94%. All scenarios before aEdG,XMB used engine locks. All scenarios after AedG,4MB did not.

Observation 3. *Real-time performance of GPUSync is competitive with the throughput-oriented driver.*

The best performing throughput-oriented scenario, TG,2MB, had an average response time of 73.98% in comparison to 76.11% for AEdG,2MB—a difference of about 2%. GPUSync is competitive with the driver while also providing guarantees on real-time performance. We believe the driver performed well because tasks largely shared periods, and thus deadlines, so the risk for priority inversions was low (there was little need for a priority inheritance mechanism).

We note that prior real-time approaches have also been compared against the throughput-oriented driver [15, 16]. Through these comparisons, we presume that we are competitive with these real-time approaches as well, though direct comparisons are necessary to draw concrete conclusions.

Observation 4. *Affinity heuristics usually improved response times, usually between 1 and 10%.*

The average response time for scenario AEdG,2MB was 76.11%, but was only 81.40% with affinity heuristics disabled (aEdG,2MB); this is an improvement ratio (without-affinity divided by with-affinity) of about 1.06. Improvement ratios due to affinity are more clearly shown in Fig. 7 where any ratio above 1.0 indicates improved performance. Observe, however, that affinity heuristics failed to improve performance in a few cases. We speculate that these instances could be due to poor performance by the cost predictor. In our implementation, the cost predictor uses the same feedback control parameters under every scenario. It is possible that performance could be improved with scenario-tailored parameters; further investigation is merited.

Observation 5. *Chunking usually improved performance.*

The average response times for scenarios AEdG,2MB, AEdG,4MB, and AEdG,XMB were 76.11%, 78.24%, and 80.84%, respectively, as seen in Fig. 6. Chunking usually improved performance under each scenario. This is observable in Fig. 8, which depicts the ratio of response time improvement under each chunked scenario in comparison to its "unchunked" counterpart. Ratios above 1.0 indicate improved average response times. As with affinity, chunking appears to hurt performance in a few scenarios. We cannot directly explain why this is, but we note that these are the same scenarios where affinity heuristics also hurt performance. Further investigation is necessary.

Observation 6. *The use of DGLs led to minor degradations in performance.*

Under AEDG,2MB, the average response time was 78.08% in comparison to 76.11% for AEdG,2MB—a difference of about 2%. Similar trends can be observed between other scenario pairs. Under DGLs, it is possible for a task to be blocked by several jobs in one queue, while it is also at the head of another queue. In such a situation, one copy engine goes idle, even if there are other requests to be satisfied. Under nested resource requests however, such requests would be satisfied. While DGLs have resulted in increased average response times, their worst-case blocking bounds are improved over that of nested resource requests, unless the nested requests are implemented as described in [27]. So while nested locking may be favorable for workloads with

looser SRT constraints, our results demonstrate that DGLs, which have improved blocking bounds, can be applied to real-time applications with only a minor reduction in response times.

6 Conclusion

In this paper, we presented GPUSync, a lock-based GPU management framework for real-time multi-GPU systems. Unlike prior research, GPUSync is cognizant of the system bus architecture and 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 application-based empirical evaluations, that our approach can significantly improve the efficiency of real-time multi-GPU systems. Average response time improvements by a factor of three or more were observed in our evaluations, allowing previously unschedulable workloads to be schedulable.

There are many opportunities for further research. More work can be done to develop more advanced GPU assignment heuristics. For example, it should be straightforward to extend support to systems with GPUs of differing performance characteristics, i.e., heterogeneous GPUs. This extension would also add support for integrated GPU graphics hardware (GPUs embedded on-chip with multicore processors that only use system memory) into GPUSync. Also, our heuristics currently assume that tasks exhibit relatively consistent execution behaviors. Our feedback control systems, as currently designed, would be unable to provide good estimations for tasks with erratic execution behaviors. However, it may be possible to improve estimates with additional information from the user program. Finally, it would be interesting to quantify the benefits of increases parallelism offered by GPUs that support GPUs that support simultaneous bi-directional memory copies.

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 Real-Time Systems Symposium*, pages 239–248, 2010.

[3] T. P. Baker. A stack-based resource allocation policy for real-time processes. In *11th Real-Time Systems Symposium*, pages 191–200, 1990.

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

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

[6] B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *11th International Conference on Embedded Software*, 2011.

[7] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *27th Real-Time Systems Symposium*, pages 111–126, 2006.

[8] 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*, pages 84–97, 2009.

[9] G. Elliott and J. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. In *19th Conference on Real-Time and Network Systems*, pages 15–24, 2011.

[10] G. Elliott and J. Anderson. Real-world constraints of GPUs in real-time systems. In *17th Conference on Embedded and Real-Time Computing Systems and Applications*, volume 2, pages 48–54, 2011.

[11] G. Elliott and J. Anderson. Building a real-time multi-GPU platform: Robust real-time interrupt handling despite closed-source drivers. In *24th Euromicro Conference on Real-Time Systems*, 2012. to appear.

[12] G. Elliott and J. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1):34–74, 2012.

[13] 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*, pages 1006 –1013, 2010.

[14] Intel. *Intel 5520 Chipset and Intel 5500 Chipset Datasheet*, 2009.

[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 Real-Time Systems Symposium*, pages 57–66, 2011.

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

[17] J. S. Kim. IMU-aided affine-photometric KLT feature tracker in CUDA framework. http://www.cs.cmu.edu/~myung/IMU_KLT/KLT_gpu_cuda.html.

[18] M. Lalonde, D. Byrns, L. Gagnon, N. Teasdale, and D. Laurendeau. Real-time eye blink detection with GPU-based SIFT tracking. In *Canadian Conference on Computer and Robot Vision*, pages 481–487, 2007.

[19] Z. Majo and T. R. Gross. Memory system performance in a NUMA multicore multiprocessor. In *4th Systems and Storage Conference*, pages 1–10, 2011.

[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, University of Illinois at Urbana Champaign, 2010.

[22] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *23rd Symposium on Operating Systems Principles*, pages 233–248, 2011.

[23] S. Thrun. GPU Technology Conference 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. *Journal of Computer Vision*, 9(2):137–154, 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*, pages 229–238. Springer US, 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 Euromicro Conference on Real-Time Systems*, 2012. to appear.

[28] B. Ward, G. Elliott, and J. Anderson. Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *18th Conference on Embedded and Real-Time Computing Systems and Applications*, 2012. to appear.

A Blocking Analysis

In this appendix, we investigate the blocking complexity of GPUSync, and explore the effect of different values of ρ on the blocking behavior. The blocking analysis is broken down into two phases, blocking associated with waiting for a GPU token, and blocking waiting for individual engines. The total blocking for each GPU request is then equal to the sum of the blocking in each of these phases. In all subsequent analysis, we assume *suspension-oblivious* analysis, in which suspensions are analytically considered computation time [5, 6]. Note that our analysis of the blocking complexity of GPUSync parallels the blocking analysis of R^{2}DGLP.

In the following analysis, we let m be the number of processors, and n be the number of GPU-using tasks, denoted $T_1, \ldots, T_n$. Assume, without loss of generality, that each GPU-using task engages in GPUSync at most once per job. Within this operation, T_i may issue e_i execution-engine requests. Associated with each job of a task T_i is at most d_i bytes of data (state, input, and output) that must be transferred to or from the GPU as part of the GPU request, and the d_i bytes are broken up into chunks of size c (as described in Sec. 4). Let $L^C_{\max}$ be the maximum amount of time it takes to transfer C bytes to or from a GPU and $L^E_{\max}$ be the maximum amount of time any job executes on a GPU execution engine. Finally, the amount of time a task executes on the CPU to manage this workflow is given by $L^P_{\max}$.

Engine-lock Blocking. Recall from the design of GPUSync that there are at most ρ tokens associated with each GPU. Thus, when direct GPU-to-GPU migrations are *not* used, there can be at most ρ tasks contending for any engine at a time. Using this property, we can analyze blocking associated with waiting on each individual engine. We thus have the following lemma.

Lemma 1. *A task holding GPU token a can be blocked by at most $N^E = \rho - 1$ other requests for the execution engine of GPU$_a$ per execution-engine request.*

Similarly, if tasks never requires more than one copy engine at a time (i.e., they migrate through system memory), they will never be blocked by requests for other copy engines. We therefore have the following lemma.

Lemma 2. *If tasks require at most one copy engine at a time, a task holding GPU token a can be blocked by at most $N^C_{sys} = \rho - 1$ other requests for a copy engine of GPU$_a$ per copy engine request.*

However, if GPU-to-GPU copies are satisfied through DGLs,[13] it is possible for tasks to be blocked by requests for other copy engines.

Lemma 3. *If tasks may require multiple copy engines concurrently, and such requests are satisfied via DGLs, a job holding GPU token a can be blocked by at most $N^C_{dgl} = \rho g - 1$ copy engine requests per copy engine request.*

Proof. By construction, there can be at most $\rho g - 1$ other jobs holding tokens (possibly for other GPUs) at any time. Because each copy-engine lock is FIFO ordered, a request can only be blocked by a subset of the $\rho g - 1$ other requests with tokens at the time of request. □

Under DGLs, because jobs are enqueued waiting for multiple copy engines, *transitive blocking* is possible, and thus tasks can be blocked waiting for requests for engines they do not require. For example, if T_1 requires the copy engines of GPU$_a$ and GPU$_b$, and T_2 requires the copy engines of GPU$_b$ and GPU$_c$, then T_1 is transitively blocked by any request T_2 is blocked on while waiting for a copy engine of GPU$_c$.[14] It is therefore possible, for a task to be blocked by at most $\rho g - 1$ requests. When nested requests are used instead of DGLs to support GPU-to-GPU copies, it is possible that

[13] Nested requests as defined by [27] have the same blocking bound as DGLs.

[14] Note that deadlock is avoided since DGL requests are enqueued atomically on the constituent FIFO queues.

T_i's request is blocked by requests that were issued after T_i's outermost request. This property leads to more pessimistic analysis.

Lemma 4. *If tasks may require multiple copy engines concurrently, and such requests are satisfied via nested requests, a job holding GPU token a can be blocked by at most $N^C_{nest} = (\rho g - 1)(\rho g - 2)/2$ requests per outermost copy engine request.*

Proof. Consider the pathological worst case scenario in which a request is enqueued behind $\rho g - 1$ other requests for a copy engine of GPU$_a$. Because there can be at most ρg total requests for any same copy engine, if the i^{th} request in the queue for a copy engine of GPU$_a$ issues a nested request, there may be at most $i - 1$ other requests enqueued ahead of it (directly or transitively) after it issues its nested request. Thus the total number of blocking requests is $N^C_{nest} = \sum_{i=1}^{\rho g}(i-1) = (\rho g - 1)(\rho g - 2)/2$. □

Theorem 1. *The total duration of blocking while waiting for engines is given by $\lceil d_i/c \rceil N^C L^C_{\max} + N^E e_i L^E_{\max}$, where N^C is the number of blocking copy-engine requests given the system configuration.*

Proof. T_i must transmit $\lceil d_i/c \rceil$ chunks of size c to send d_i bytes and for each chunk, T_i may be blocked by N^C other copy-engine requests, each of which also takes at most $L^C_{\max}$ time to transmit. By definition, T_i uses the execution engine e_i, and thus the total blocking for the execution engine is $N^E e_i L^E_{\max}$. The total engine blocking is bounded as claimed. □

Token Blocking. To bound the duration of token blocking, we first must quantify the duration of time that a job may hold a token.

Lemma 5. *The maximum token hold time, THT^c_i, of a task T_i with chunk size c is given by $\lceil d_i/c \rceil (N^C + 1)L^C_{\max} + (N^E + 1)L^E_{\max} + L^P_{\max}$.*

Proof. The maximum hold time of a GPU token is the total duration of blocking while waiting for GPU engines, in addition to the duration of execution on each engine. The total duration of blocking is given by Theorem 1, and the total duration of execution on each of the engines is given by $\lceil d_i/c \rceil L^C_{\max} + L^E_{\max}$. Finally, the task may also hold the token for at most $L^P_{\max}$ time while executing on the CPU. The bound is then as claimed. □

Theorem 2. *The maximum duration of blocking while waiting for a token using chunk size c is given by $(2\lceil \frac{m}{\rho g} \rceil - 1)THT^c_{\max}$ where $THT^c_{\max} = \max THT^c_i$.*

Proof. Follows from the blocking bound for the R^{2}DGLP, which is used to arbitrate access to the GPU tokens, and the fact that a critical section with respect to the R^{2}DGLP is the token hold time. □

Theorem 3. *The total blocking for a GPU request under GPUSync is given by $(2\lceil \frac{m}{\rho g} \rceil - 1)THT^c_{\max} + \lceil d_i/c \rceil N^C L^C_{\max} + N^E L^E_{\max}$.*

Proof. Follows from Theorems 1 and 2. □

Discussion. Note that, asymptotically speaking, when $N^c = \rho - 1$, GPUSync is $O(m/g)$, while when $N^c = \rho g - 1$, GPUSync is $O(m)$. However, GPU-to-GPU migrations should not be immediately discounted due to increased asymptotic blocking, because, as we have demonstrated, average-case response times are improved. Furthermore, finer-grained analysis of this case may be possible to reflect the improved performance of GPU-to-GPU migrations. Besides beneficial system-wide side-effects due to reduced contention for main memory, GPU-to-GPU memory copies can usually be performed at a higher transfer rate since there is typically much less bus contention. However, quantifying such effects analytically begins to require in-depth timing analysis, which is outside of the scope of this paper.

When considering the blocking behavior with respect to constant factors, the choice of ρ can have an effect on blocking bounds. If ρ is large, priority donation will never be used, and the resulting protocol is $O(n/g)$.[15] While this may be favorable if a small number of tasks access GPUs, if a large number of tasks access GPUs this may be a poor choice. In such a case, choosing ρ to minimize $\lceil \frac{m}{\rho g} \rceil N^c$ is the best choice, with respect to blocking analysis.

[15] Such a bound is also applicable under suspension-aware analysis, and furthermore is optimal in such case.