Exploring the Multitude of Real-Time Multi-GPU Configurations

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

***Abstract*—Motivated by computational capacity and power efficiency, techniques for integrating graphics processing units (GPUs) into real-time systems have become an active area of research. While much of this work has focused on single-GPU systems, multiple GPUs may be used for further benefits. Similar to CPUs in multiprocessor systems, GPUs in multi-GPU systems may be managed using partitioned, clustered, or global methods, independent of CPU organization. This gives rise to many combinations of CPU/GPU organizational methods that, when combined with additional GPU management options, results in thousands of "reasonable" configuration choices. In this paper, we explore real-time schedulability of several categories of configurations for multiprocessor, multi-GPU systems that are possible under GPUSync, a recently proposed highly configurable real-time GPU management framework. Our analysis includes the careful consideration of GPU-related overheads. We show system configuration strongly affects real-time schedulability. We also identify which configurations offer the best schedulability in order to guide the implementation of GPU-based real-time systems and future research.**

I. Introduction

It is quickly becoming standard practice to use graphics processing units (GPUs) to tackle general purpose, data parallel computational problems, due to the significant performance advantages GPUs have over traditional CPUs, both in terms of throughput and power efficiency. The ways in which GPUs are managed and scheduled differ greatly from CPUs. This has spurred research on supporting GPUs in real-time systems [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Still, few have explored multiprocessor, *multi-GPU* real-time systems.

CPUs in traditional multiprocessor scheduling can follow a partitioned, clustered, or global approach. Under clustered scheduling, a system's m CPUs are separated into clusters of c CPUs each, and each task is scheduled within a single cluster. Partitioned and global scheduling are special cases, where $c = 1$ and $c = m$, respectively. Similarly, GPUs can be organized by following a partitioned, clustered, or global approach. This categorization yields nine possible allocation categories, as illustrated in matrix form in Fig. 1. As we describe later, when combined with additional GPU management options, these nine choices multiply into many more. Which configurations are best for real-time predictability? Does configuration really matter? The answers to these basic questions are not immediately clear.

We began to answer some of these questions in prior work, where we explored technical implementation issues and the *run-time* performance of a subset of multi-GPU configurations [7]. Therein, we observed that clustered GPU scheduling can improve job response time. However, we did not investigate performance in terms of real-time schedulability. In this paper, we present an evaluation of several categories of multi-GPU configurations based on real-time schedulability. This evaluation carefully considers both general and GPU-specific runtime overheads. Although not exhaustive, our evaluation is broad (requiring over 40,000 CPU hours to complete). We investigate several possible configurations within each of the nine aforementioned high-level categories. We show that real-time guarantees differ greatly among configurations.

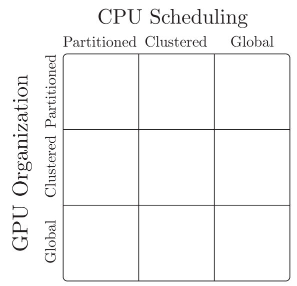

Figure 1: Matrix of CPU and GPU organization.

Scope and contributions. Our study was performed within the context of GPUSync, a highly configurable real-time GPU management framework developed by us that extends LITMUS^{RT}, a Linux-based real-time OS.[1] GPUSync takes a locking-protocol-based philosophy to GPU scheduling. Thus, schedulability tests incorporate lock-related blocking analysis, as well as overhead accounting.

We investigate real-time schedulability in terms of the existence of response-time bounds. This motivates us to use "fair-lateness" (FL) schedulers, which are earliest-deadline-first-like (EDF-like) schedulers that have provably smaller response-time bounds than standard EDF schedulers when $c > 1$ (FL is equivalent to EDF when $c = 1$) [11]. Under FL scheduling, *priority points* are defined to minimize worst-case response times. The FL scheduler uses these priority points as pseudo-deadlines, and thus is a job-level static-priority scheduler (a requirement of GPUSync).

The central contribution and intent of this paper is to *identify the most promising CPU/GPU configurations* by modeling real-world system behavior in overhead-aware schedulability tests (we do not seek to address other real-time issues such as precise bounds on worse-case execution time). We follow the overall empirical measurement, overhead accounting, and experimental process developed by Brandenburg [12], with additional techniques to quantify the effects that GPU operations have on overheads. We also devise a new ranking method to aid in comparing many CPU/GPU configurations under various assumptions and task set properties. Ultimately, we find that clustered CPU scheduling with partitioned GPUs offers the best real-time

[1]LITMUS^{RT} and GPUSync source code is shared at www.litmus-rt.org.

schedulability, overall. However, clustered GPU configurations are competitive in some situations. In these cases, a system designer may take advantage of the improved runtime performance demonstrated in [7] with minimal impact on response-time bounds.

Organization. In the rest of the paper, we provide needed background (Sec. II), discuss aspects of GPUSync relevant to blocking analysis (Sec. III), consider general and GPU-specific overheads and present empirical measurements (Sec. IV), and present the results from our schedulability experiments (Sec. V). We conclude with a summary of our findings and notes for future work (Sec. VI). Additional data and blocking analysis is presented as appendices.

II. Background

Current real-time GPU research falls within three general categories: (i) techniques for persistent low-latency tasks [1, 13], (ii) worst-case execution-time analysis of GPU program code [3, 4], or (iii) GPU resource scheduling [2, 5, 6, 7, 8, 9, 10]. In (i), a persistent task executes on a dedicated GPU, polling for and processing work. This research has focused on efficient data movement between a single GPU and the rest of the system. There is no need for *scheduling* data-movement or GPU computations since there is only a single dedicated GPU. Research on (ii) has focused on bounding the execution time of GPU program code, with no attention paid to scheduling or data-movement costs—it is assumed all data already resides on the GPU. In contrast to the first two categories, the techniques developed in (iii) seek to schedule both data movement and GPU computations on GPU(s) shared by competing jobs of different priorities. Only [5, 6, 7] have directly approached the topic of multi-GPU scheduling in real-time systems. This paper also falls within this last category. Specifically, we investigate the *analytical* real-time properties of GPUSync [7] ([7] focuses on *observed* real-time performance). However, before we can address this topic directly, we must first discuss system hardware specifics, examine how GPUs are used, and motivate our synchronization-based approach. We adapt some of the following information from [7] to suit our needs here.

System hardware. GPUs may be “discrete” or “integrated.” There are two distinguishing characteristics between these. First, integrated GPUs share main memory with CPUs, while discrete GPUs have local high-performance memory. Second, integrated GPUs are built with fewer transistors since they share silicon with CPUs and other system-on-chip components—this limits performance. We focus our attention on discrete GPUs due to their performance characteristics, but this introduces 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 EE consists of many parallel processors and performs all computation.

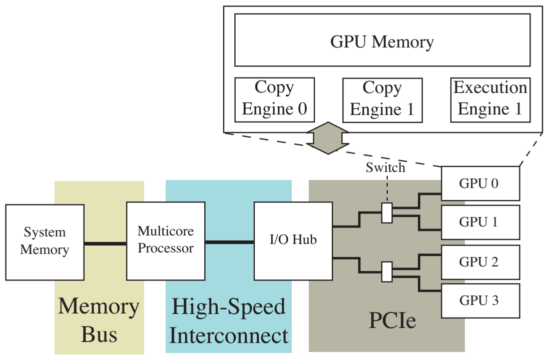

Figure 2: Example high-level architecture. On some multicore chips the I/O hub may be integrated.

The CEs transmit data between system memory and GPU memory.[2] GPUs commonly have only one CE and cannot send and receive data at the same time. However, high-end GPUs may have an additional independent CE, enabling simultaneous bi-directional transmissions. EEs and CEs perform operations *non-preemptively*.

Fig. 2 depicts a high-level architecture of a multicore, multi-GPU system. The CEs 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. Traffic is arbitrated at each switch using round-robin arbitration at the packet level. The structure depicted in Fig. 2 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.

GPU usage pattern. GPU-using programs execute on CPUs and invoke a sequence of *GPU operations*. There are two types of GPU operations. *Kernel* operations are programs executed by the GPU EE. *Memory copy* operations are data transfers to or from a GPU's local memory; these are processed by the CEs. A general execution sequence for a GPU-using program scheduled alone is depicted in Fig. 3. Observe that a program running on a CPU initiates GPU operations—the GPU does not initiate them independently. At time t_1, the 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 CEs. 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_6. Finally, the program no longer requires the GPU at time t_8. We call the duration from time t_1 to time t_8 a *GPU critical section* because the program expects its sequence of operations to be carried out

[2]GPUs support several data transmission methods [1, 14], but we focus on CEs due to high performance and ease of deterministic control.

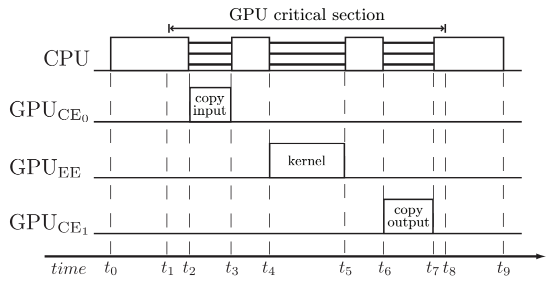
GPU critical section
CPU
$\mathrm{GPU}_{\mathrm{CE}_0}$
copy input
$\mathrm{GPU}_{\mathrm{EE}}$
kernel
$\mathrm{GPU}_{\mathrm{CE}_1}$
copy output
time t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9

Figure 3: GPU-using program execution sequence.

on the same GPU. Recall that GPU operations on the various engines are non-preemptive. For example, $\mathrm{GPU}_{\mathrm{CE}_0}$ cannot be preempted within $[t_2, t_3]$. However, the program running on the CPU is preemptive while waiting if it busy-waits (and is not running if it suspends). 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 CEs—it is actually up to the GPU to select which CE to use.

In addition to memory used for input and output, recurrent tasks may maintain inter-job *state* in GPU memory. State must be *migrated* from one GPU to another if such a task switches GPUs. Migration cost is the time taken to copy state data between these GPUs. This cost is partly dependent upon the *distance* between GPUs and 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. 2, the distance between GPUs 0 and 2 is two (one link to a switch, a second link to the 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 peer-to-peer (P2P) communication: data is copied directly from one GPU to another. P2P-based migrations are potentially more efficient, especially over short distances, due to proximity and reduced bus contention. However, this method requires coordination between the GPUs.

Synchronization-based philosophy. Significant difficulties arise when we attempt to develop schedulability tests for the system described above. A multiprocessor, multi-GPU system is a heterogenous multiprocessor system. However, GPU-using tasks are not fully preemptive, cannot execute wholly on one processor type or another, and may not arbitrarily migrate from one processor type to another. Thus, prior holistic schedulability tests for heterogenous multiprocessors do not apply because they assume tasks can be partitioned among heterogenous processors or execution is fully preemptive (e.g., [15, 16, 17]). Recently, Kim et al. presented schedulability analysis for uniprocessor, uni-GPU, systems under fixed-priority scheduling [10]. The main contribution of [10] is in managing self-suspensions that arise when a job suspends from a CPU to execute on a GPU. However, Kim et al.'s approach is not general enough for us to explore all the possibilities discussed in Sec. I. This lack of generality motivates us to consider alternative methods.

Real-time locking protocols impart a predictable access pattern to non-CPU resources. The locking protocol *itself* is a non-preemptive scheduler: it is non-preemptive since lock ownership cannot be arbitrarily revoked. Gai et al. observed this in [18], where they cast the problem of scheduling a uniprocessor with a single non-preemptive digital signal processor as a synchronization problem. We also took this perspective in developing GPUSync, a *single* synchronization-based framework that allows us to explore every high-level category described in Sec. I.

Although our current analysis is centered on locking protocols, it may be possible to analyze GPUSync with new holistic schedulability tests for heterogenous multiprocessors without having to change GPUSync itself. New tests would have to account for the complex relationship between EEs and CEs and task self-suspensions.

III. GPUSync

In this section, we discuss the synchronization-based framework implemented by GPUSync. We limit our attention to the aspects of GPUSync that relate to schedulability analysis. A treatment of technical aspects (such as interrupt handling and budgeting) can be found in [7]. We begin with a discussion of our assumed system model and proceed to describe GPUSync.

A. System Model

We consider a task system, T, comprised of n real-time tasks $T_1, \cdots, T_n$ that are scheduled on m CPUs, partitioned into clusters of c CPUs each. The subset $T^G \subseteq T$ includes all tasks that require GPU resources from the system's h GPUs, partitioned into clusters of g GPUs each. The subset $T^C \triangleq T \backslash T^G$ are tasks that do not use a GPU. We assume that the workload to be supported can be modeled as a traditional sporadic real-time task system. Every task has a *provisioned CPU execution time* of e_i^{cpu}, *period* p_i, and *relative deadline* d_i. Each task releases a (potentially infinite) series of *jobs* $T_{i,j}$ with a minimum separation time of p_i time units. Job $T_{i,j}$ is released (*arrives*) at time $a_{i,j}$ and completes (*finishes*) at time $f_{i,j}$. The *response time* of $T_{i,j}$ is $r_{i,j} \triangleq f_{i,j} - a_{i,j}$. The parameter e_i^{gpu} denotes T_i's *provisioned GPU execution time*. The parameter q_i^{cpu} denotes T_i's total CPU execution time requirements *within* its GPU critical section (note that q_i^{cpu} is included in e_i^{cpu}). Each $T_{i,j}$ sends (receives) z_i^I (z_i^O) bytes of data as *input* (*output*) to (from) GPU computations. The size of $T_{i,j}$'s state is denoted by z_i^S. For convenience, we define the function $\mathit{xmit}(z_i^I, z_i^O, z_i^S)$ to specify the total data transmission time required by $T_{i,j}$. As we discuss in Sec. IV, this can be computed given empirical measurement data. We assume that a job of $T_i \in T^G$ may use any

Parameter	Description
m	number of system CPUs
h	number of system GPUs
c	CPU cluster size
g	GPU cluster size
T^G	set of all GPU-using tasks
T^C	set of all CPU-only tasks
e_i^{cpu}	T_i's provisioned CPU execution time
e_i^{gpu}	T_i's provisioned GPU execution time
q_i^{cpu}	total CPU execution time within T_i's GPU critical section
z_i^I	size of T_i's GPU input data (bytes)
z_i^O	size of T_i's GPU output data (bytes)
z_i^S	size of T_i's inter-job GPU state data (bytes)
b_i	upperbound on blocking for T_i

Table I: Important notation.

one arbitrary GPU in its GPU cluster. e_i^{gpu}, z_i^I, z_i^O, and z_i^S are zero for $T_i \in T^C$. The term b_i denotes an upper-bound on the time $T_{i,j}$ may be blocked due to lock requests (for presentation simplicity, we assume tasks share no other resources, but this is not a GPUSync requirement). We derive values for b_i in Appendix B. Finally, T_i's *utilization* is given by $u_i \triangleq (e_i^{cpu} + e_i^{gpu} + xmit(z_i^I, z_i^O, z_i^S))/p_i$, and the task set utilization is $U \triangleq \sum_{i=1}^{n} u_i$.

We refer back to the parameters summarized in Table I.

Example. If we assume that the GPU usage pattern illustrated in Fig. 3 represents the entire execution sequence of a job $T_{i,j}$, then $e_i^{cpu} = (t_2 - t_0) + (t_4 - t_3) + (t_6 - t_5) + (t_9 - t_7)$, $e_i^{gpu} = t_5 - t_4$, $q_i^{cpu} = (t_2 - t_1) + (t_4 - t_3) + (t_6 - t_5) + (t_8 - t_7)$, and $xmit(z_i^I, z_i^O, z_i^S) = (t_3 - t_2) + (t_7 - t_6)$ (assuming $z_i^S = 0$, i.e. the job has no state to migrate between GPUs).

B. GPUSync Structure

It helps to refer to concrete system configurations in describing GPUSync, so let us define several such configurations. Fig. 4 depicts a matrix of several high-level CPU/GPU configurations for a 12-CPU, 8-GPU system, which we also use in Secs. IV and V. We refer to each cell in Fig. 4 using a column-major tuple, with the indices P, C, and G denoting partition, clustered, and global choices, respectively. The tuple (P, P) refers to the top-left corner—a configuration with partitioned CPUs and GPUs. Likewise, (G, C) indicates the right-most middle cell—globally scheduled CPUs with clustered GPUs. We use the wildcard $*$ to refer to an entire row or column: e.g., $(P, *)$ refers to the left-most column—all configurations with partitioned CPUs. Within each cell, individual CPUs and GPUs are shown on the left and right, respectively. Dashed boxes delineate CPU and GPU clusters (no boxes are used in partitioned cases). The solid lines depict the association between CPUs and GPUs. For example, the solid lines in (C, C) indicate that two GPU clusters are wholly assigned to each CPU cluster. Finally, the horizontal dashed line across each cell denotes the NUMA boundary of the system. Offline, tasks are assigned to CPU and GPU clusters in accordance with the desired

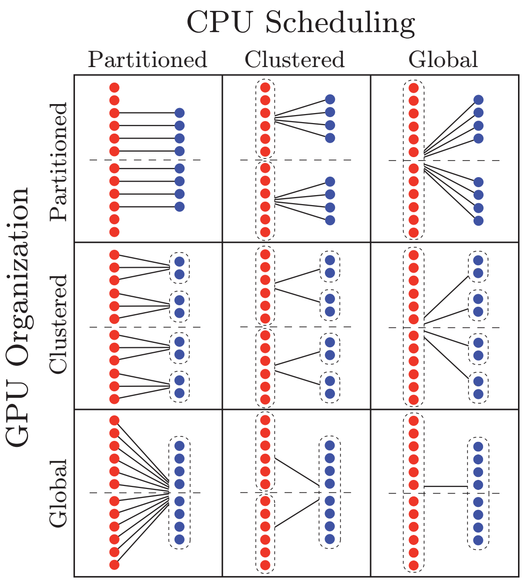

Figure 4: Concrete configurations.

configuration.

GPUSync uses a two-level nested locking structure: an outermost *token lock* to allocate GPUs to jobs and innermost *engine locks* to arbitrate access to GPU engines. This is depicted in Fig. 5. In Step A (or time t_1 in Fig. 3), the job requests a *token* from the GPU allocator responsible for managing the GPUs in the job's GPU cluster. The GPU allocator determines which token—and by extension, which GPU—should be allocated to the request. The requesting job may access the assigned GPU once it receives a *token* in Step B. In Step C, 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 after acquiring its needed engine locks in Step D. For example, an engine lock must be acquired at times t_2, t_4, and t_6 in Fig. 3. With the exception of P2P migrations, a job cannot hold more than one engine lock at a time.

GPUSync can be configured to use different locking protocols to manage tokens and engines. In this paper, we configure GPUSync to use protocols known to offer *asymptotically optimal* blocking bounds under FL scheduling. We now describe the two locking levels in more detail. We provide blocking analysis in Appendix B.

Token lock. Each cluster of g GPUs is managed by one GPU allocator. We associate ρ tokens (a configurable

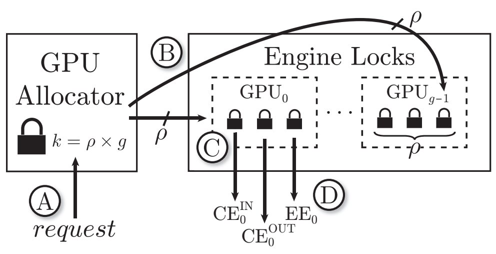

Figure 5: High-level design of GPUSync.

parameter) with each GPU. All GPU tokens are pooled and managed by the per-cluster GPU allocator using a *single* k-exclusion lock, where $k = \rho g$. Jobs with a pending token request suspend until assigned a token.

We employ one of two k-exclusion locking protocols to allocate tokens, depending upon how GPU clusters are shared among CPU clusters. We use the Replica-Request Donation Global Locking Protocol (R^{2}DGLP) [19] in cases where GPU clusters are wholly assigned to a single CPU cluster. This is because the R^{2}DGLP's progress mechanism (i.e., priority inheritance) ensures predictable behavior without directly affecting CPU-only tasks. However, a stronger mechanism (i.e., priority donation) is required when GPU clusters are *shared* among CPU clusters. Hence, we employ the Clustered k-exclusion O(m) Locking Protocol (CK-OMLP) [20] for cases (P, C), (P, G) and (C, G). Priority donation may cause CPU-only tasks to experience blocking at release-time. We use the R^{2}DGLP and CK-OMLP because these protocols offer asymptotically optimal blocking bounds with the associated CPU/GPU cluster configurations under FL scheduling. To improve runtime performance, we augment both protocols with heuristics that bias token selection towards certain GPUs over others in order to reduce the frequency and cost of GPU migrations within a cluster. We showed that this can substantially improve observed performance in [7]. Moreover, these heuristics do not affect the analytical properties of the locking protocol.

The value of ρ affects both migration frequency and GPU parallelism. A small value of ρ, such as $\rho = 1$, will trigger frequent migrations since tokens are scarce, incurring migration overheads. A larger value of ρ, such as $\rho = 3$, will allow the engines of a GPU to be used by separate jobs simultaneously. However, if tokens are *too* plentiful, then work may "pile up" on few GPUs, while others are underutilized. There is a balance to strike: the value of ρ should promote GPU parallelism while distributing work equitably across GPUs.

Engine locks. A mutex is associated with every GPU EE and CE. For GPUs with *two* CEs, one engine is dedicated to inbound data copies, and the other to outbound data copies. Although CEs are capable of copying data both to and from the GPU, on our test platform, we were unable to coax a dual-CE GPU (NVIDIA Quadro K5000) to perform two copies simultaneously in the same direction. Thus, we are unable to reserve one CE for copies to and from system memory and the other for P2P migrations (such a configuration may offer favorable analytical properties).

Each engine mutex prioritizes requests in FIFO order. Blocked jobs suspend while waiting for an engine. A job that holds an engine lock may inherit the priority of any job it blocks. Priority inheritance relations from the token lock may propagate to an engine holder to ensure timely real-time scheduling. A job releases an engine lock once its engine-related operation (e.g., GPU kernel execution or memory copy) completes. 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.

Parameter	Choices
CPU Scheduling	P, C, G
GPU Organization	P, C, G
Tokens per GPU (ρ)	$\mathbb{N}_1$
Migration Method	None, System Memory, P2P

Table II: GPUSync configuration parameters studied in this paper.

Schedulability is best when engine locks are held for short durations, so each individual GPU operation should be protected by separate engine-lock critical sections. To that end, GPUSync provides convenience routines to break large memory copies into chucks. Others have used this chunking technique [1, 8].

Migrations. GPUSync supports both P2P and system memory migrations. The rules governing each method differ.

Under P2P migration, when migrating from GPU$_a$ to GPU$_b$, a job must hold the appropriate CE locks for both GPU$_a$ and GPU$_b$. As shown in [21], worst-case blocking grows quadratically with respect to the total number of GPU tokens if these locks are acquired separately. We avoid such excessive blocking by instead using *dynamic group locks* (*DGLs*) [22]. Using DGLs, a job *atomically* requests *both* CE locks *simultaneously*, instead requesting of one after the other. As a result, worst-case blocking grows linearly instead of quadratically. A job may issue memory copies to carry out migration once both engine locks are held. P2P migrations are usually only possible between GPUs within the same NUMA node, so P2P migrations are restricted to cases $(*, C)$ of Fig. 4.

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

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

Table II summarizes the GPUSync configuration options we study in this paper. Not every combination is valid, e.g., the migration method "None" is only valid for $(*, P)$.

IV. Overheads

In this section, we discuss the methodology we used to gather general and GPU-related system overheads for the

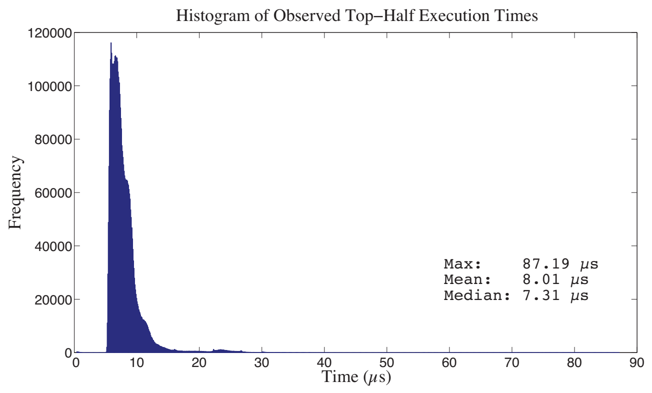

Figure 6: Histogram of observed top-half execution times.

test platform upon which our evaluation in Sec. V is based. Brandenburg demonstrated in [12] the importance (and challenges) of incorporating such overheads into schedulability analysis—sometimes schedulers that perform well in theory, do poorly in practice. We replicate and extend Brandenburg's methods for gathering and accounting for these overheads. We classify overheads as either algorithmic overheads or memory overheads. Before discussing these further, we first frame the context in which our overheads were gathered.

Evaluation platform. Our analysis and overheads are within the context of our implementation of GPUSync in LITMUS$^{\text{RT}}$ (based on the 3.10.5 Linux kernel) running on our test platform. This platform has two NUMA nodes, each like the system depicted in Fig. 2. Each NUMA node is equipped with one Xeon X5060 processor with six 2.67GHz cores, and four NVIDIA K5000 Quadro GPUs. Overheads were gathered under partitioned, clustered, and global FL scheduling. We used CUDA 6.0 as our GPU runtime environment and the NVIDIA Linux driver 331.62.

A. Algorithmic Overheads

We followed Brandenburg's methods to gather *algorithmic overheads*. These include: thread context switching, scheduling, job release queuing, inter-processor interrupt latency, CPU clock tick processing, and GPU interrupt processing. We measured these overheads using the light-weight tracing facilities of LITMUS$^{\text{RT}}$ while executing workloads that stress the various hardware components managed by GPUSync. These overheads were measured under different CPU and GPU cluster configurations, as well as with task sets of varying sizes (in order to capture overhead trends dependent upon the number of tasks). Over 11GB of trace data was recorded (a single trace event is only 16 bytes in size). We distilled this data into average and worst-case overheads, and incorporate them into schedulability analysis using Brandenburg's "preemption-centric" method [12]. Most of these algorithmic overheads have been studied in prior work, so we do not discuss them further. However, we do discuss algorithmic overheads related to GPUs.

GPUs interact with the host system primarily through I/O interrupts. Interrupt processing is split into "top" and "bottom" halves. Due to page constraints, we defer a lengthy description of our GPU interrupt accounting method to [21]. Nevertheless, this accounting requires that we quantify the execution cost of top and bottom halves of GPU interrupt processing. Fig. 6 shows a histogram of observed top-half execution times recorded over a 20 minute period, where 30 tasks executing at rates between 10 and 30 frames per second, used GPUs managed by GPUSync to perform computer vision calculations.[3] The most striking aspect of this data is the outliers: the maximum value is $87.19\mu s$, yet the mean and median are only $8.01\mu s$ and $7.31\mu s$, respectively. We observed similar characteristics for bottom-halves: the maximum observed bottom-half execution time was $1ms$, while the mean and median are $66.14\mu s$ and $54.68\mu s$, respectively. Although these overheads are greater, GPUSync schedules these bottom-halves according to real-time priorities, so this significantly mitigates their impact [5]. We must be mindful of the discrepancies between worst-case and average measurements as we model schedulability and determine system provisioning in practice.

B. Memory Overheads

Although algorithmic overheads are important, those related to memory access are more so in a real-time GPU system. As pointed out by Pellizzoni et al. [23], I/O memory bus traffic can significantly impact the performance of tasks executing on CPUs due to system memory bus contention. Moreover, in multi-GPU systems, there is also contention for the PCIe bus. We seek to quantify two memory-related overheads. First, we want to determine the impact GPU memory traffic has on cache preemption/migration delays (CPMDs) [12]. Second, we seek to determine the speed at which data can be transmitted to and from system memory and directly between GPUs. We incorporate the former into schedulability analysis. The latter is used to compute task execution requirements on GPU CEs—critical to real-time GPU schedulability. CPMD and GPU memory copy costs have been studied in prior work ([12] and [14], respectively), so we refrain from repeating such work here. Instead, we focus on cost *increases* due to GPU memory traffic under worst-case scenarios.

Increase in CPMDs. To assess CPMDs, we used an experimental method modeled after the "synthetic method" described in [12]. A non-preemptive instrumented process records the time taken to read a prescribed amount (a "working set size") of sequential data from a "hot" cache. The process suspends for a short duration, resumes on a random processor, and rereads said data from the now "cold" cache. A cost is determined by subtracting the hot measurement from the cold.

We are concerned with two memory configurations since our test platform is a NUMA system. Under partitioned and clustered CPU scheduling (when clusters reside entirely

[3]This is the same test scenario used in our evaluations in [7].

within a NUMA node), memory can be allocated locally to increase performance and reduce interference from NUMA-remote tasks. However, under global CPU scheduling, one may *interleave* memory pages across the NUMA nodes in order to obtain good average case performance. We require overhead data for both configurations in order to accurately model each CPU/GPU configuration described in Sec. III.

Under both *local* and *interleaved* configurations, we collected three CPMD datasets: (i) an "idle" dataset where the instrumented process runs alone, (ii) a "loaded" dataset where "cold" measurements are taken in the presence of cache-trashing processes that introduce contention for both caches and memory bus, and (iii) a "loaded-gpu" dataset where additional load is created by GPU-using processes, one for each CE, that fully loads the bidirectional PCIe bus with a constant stream of 512MB DMA memory transfers to and from pinned pages in system memory. We observed that GPU traffic increased local CPMD costs by a factor between two and four for working set sizes larger than 32KB. Interleaved CPMDs were affected to a lesser degree, with increases by a factor between 1.1 and 1.9. However, interleaved CPMDs *without* GPU traffic are nearly as great as local CPMDs *with* GPU traffic.

GPU memory copy costs. We performed similar experiments to determine GPU memory copy costs. An instrumented process performed memory copies to and from system memory and P2P memory copies between neighboring GPUs. We tested both local and interleaved configurations under idle and loaded scenarios. In addition to loading every GPU CE, we also executed memory-heavy GPU kernels on the EE of GPUs used by the instrumented process in order to stress the GPU's *own* local memory bus. Table III shows the worst-case GPU transmission times with and without load for 1MB memory copies. There are no results for P2P copies in the interleaved case since P2P copies are isolated from system memory. We see that load greatly increases memory transmission time. For example, system-to-GPU memory copies increased by factors of 6.6 and 9.6 for the local and interleaved cases, respectively. Interestingly, P2P transmissions were hardly affected. This is because P2P copies were performed between *neighboring* GPUs, so there was no contention for the PCIe bus linking them. This result also shows that the EE did not strongly affect DMA performance. This is due to the GPU's architecture: the EE has a very high-bandwidth connection to GPU memory in comparison to a CE.

V. Experimental Results

In this section, we assess trade-offs among many configuration options supported by GPUSync by presenting the results of overhead-aware schedulability studies. We randomly generated task sets of varying characteristics and tested them for schedulability using the methods described in [11]. We now describe the experimental process we used.

1MB	Idle	Loaded	Factor
Local			
System-to-GPU	179.6ms	1181.3ms	6.6
GPU-to-System	169.2ms	1214.3ms	7.2
P2P	167.9ms	175.68ms	1.05
Interleaved			
System-to-GPU	187.1ms	1802.4ms	9.6
GPU-to-System	204.6ms	1733.7ms	8.4

Table III: Worst-case GPU memory copy cost for 1MB.

Experimental setup. There is a wide space of system configuration and taskset parameters to explore. We evaluated each high-level configuration illustrated in Fig. 4. These configurations are not exhaustive, but we feel they are they simplest and realistic configurations within each cell. For instance, in (P, P), four partitioned CPUs have no attached GPU; these CPUs may only schedule tasks of T^C. Such a configuration is a natural extension of existing uniprocessor, uni-GPU methods. We also only explore GPU clusters of size two in $(*, C)$. This is because we found no runtime benefit to larger sizes in [7]. The configurations of $(*, C)$ were also tested under system memory and P2P migration methods (we denote the P2P cases as $(*, C^{P2P})$). Every cell was tested with several values of ρ: $\rho = 1$ to examine schedulability under exclusive GPU allocation; $\rho = 3$ to explore schedulability when all GPU engines (one EE and two CEs) are given the opportunity to operate simultaneously; and $\rho = 2$ to see if there is a balance to strike between $\rho = 1$ and $\rho = 3$. $(*, P)$ was also tested with $\rho = \infty$ since ρ's role in facilitating migrations is moot. Finally, we assumed a data copy chunk size of 1MB.

Random task sets for schedulability experiments were generated according to several parameters in a multistep process. Task utilizations were generated using three uniform distributions: $[0.01, 0.1]$ *(light)*, $[0.1, 0.4]$ *(medium)*, and $[0.5, 0.9]$ *(heavy)*. Task periods were generated using two uniform distributions with ranges [33ms, 100ms] *(moderate)*, [200ms, 1000ms] *(long)*.[4] Tasks were generated by selecting a utilization and period until reaching a desired task set utilization. The task set was then randomly subdivided into T^G and T^C. The number of tasks in T^G was set to be: 33%, 66%, or 100% of the task set size. For tasks in T^G, kernel execution times were generated using three uniform distributions with ranges $[10\%, 25\%]$, $[25\%, 75\%]$, and $[75\%, 95\%]$ of task execution time (a corresponding amount of time was subtracted from CPU execution time). Input/output data sizes were generated using three values: 256KB (*light*), 2MB (*medium*), and 8MB (*heavy*). A selected data size was evenly split between z_i^I and z_i^O. Task GPU state size was generated using three values: 0%, 25%, and 100% of T_i's combined input/output data size. In order to keep our study tractable, all tasks were assigned a CPU cache working set size of 4KB. For tasks in T^G, 5% of its

[4]These periods are inspired by the sensor streams GPUs may process. Moderate periods represent video-based sensors. Long periods model slower sensors such as LIDAR.

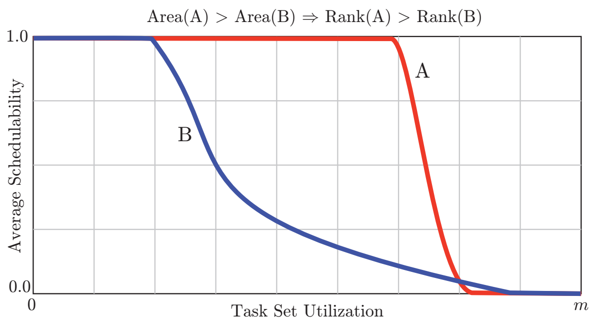

Figure 7: Illustrative ranking.

CPU execution time was determined to be within the task's *single* GPU critical section. Overheads and data transmission times were taken from four data sets: average-case (AC) observations in an idle system (AC/I); AC observations in a loaded system (AC/L); worst-case (WC) observations in an idle system (WC/I); and WC observations in a loaded system (WC/L).

A unique combination of the above system configurations and taskset parameters defined a set of experiment settings, $75,816$ in all. Under each set of experiment parameters, for each 0.25 increment in system utilization range $(0, 12]$ (reflecting the range of system utilizations supported by our twelve-core test platform), we generated between 500 and $4,000$ task sets.[5] Task sets were partitioned to the CPU/GPU clusters in three phases:

Phase 1: T^G was partitioned among the GPU clusters, using the worst-fit heuristic in decreasing GPU utilization order, where $u_i^{gpu} \triangleq (e_i^{gpu} + q_i^{cpu} + xmit(z_i^I, z_i^O, z_i^S))/p_i$.

Phase 2: T^G was then partitioned among CPU clusters, in accordance with experiment parameters, using the worst-fit heuristic in decreasing utilization (as defined in Sec. III) order. Blocking terms were calculated and incorporated into each CPU cluster's (estimated) total utilization.

Phase 3: T^C was then partitioned among the CPU clusters using the worst-fit heuristic in decreasing utilization order.

Task sets were then tested for bounded response time, incorporating the overheads discussed in Sec. IV; blocking terms were calculated using the *fine-grain* analysis presented in [21]. Approximately two billion task sets were tested. We used a university compute cluster to perform our experiments, consuming over $42,000$ CPU hours. Our experiment tools were implemented on top of the schedulability test toolkit SchedCAT [24].

Results. With over $75,000$ experiments, it is infeasible to compare different system configurations by examining individual schedulability curves alone. Since our primary goal is to compare the effectiveness of each configuration, we devised the following ranking method to collapse our results into something more manageable. For every unique combination of task set parameters, we determined a "sub-rank" for each system configuration from first to last place. These sub-rankings were determined by comparing the area under each system configuration's schedulability curve. A larger area under the curve indicates better schedulability. An illustrative example is shown in Fig. 7. In this example with two system configurations A and B, configuration A has a first-place sub-rank since the area under A's curve is greater (i.e., more task sets were schedulable under A). A *final* rank for each system configuration was determined by computing for each configuration, the median, average, and standard deviation of its sub-ranks. We then ranked system configurations according to median sub-rank, tie-breaking by average sub-rank. This ranking approach was applied separately to results from each of our four overhead datasets.

Rankings Under Worst-Case Overheads, Loaded

Rank	Median	Avg	Std	(CPU,GPU,ρ)	WC/I	AC/I	AC/L
1	3	5.75	5.35	(C, P, ∞)	1	1	1
2	4	6.16	5.24	$(C, P, 3)$	3	4	2
3	5	6.05	4.97	$(C, P, 2)$	2	5	3
4	6	7.27	5.91	(P, P, ∞)	5	7	9
5	6	7.75	6.52	$(P, P, 2)$	4	12	12
6	6	7.91	6.24	$(P, P, 3)$	6	13	11
7	10	10.82	4.48	$(C, C, 2)$	8	9	7
8	10	11.02	7.93	$(P, P, 1)$	9	22	21
9	11	10.95	4.45	$(C, P, 1)$	7	16	14
10	11	11.40	4.49	$(C, C^{P2P}, 2)$	10	10	8
11	13	13.18	4.44	$(C, C, 3)$	11	15	15
12	13	13.39	4.59	$(C, C^{P2P}, 3)$	13	20	16
13	13	14.02	8.60	(G, P, ∞)	14	2	4
14	13	14.25	8.35	$(G, P, 3)$	15	3	5
15	13	14.69	8.26	$(G, P, 2)$	12	6	6
16	18	18.57	5.61	$(G, P, 1)$	16	17	17
17	18	18.88	5.59	$(G, C, 2)$	17	8	10
18	19	19.22	5.49	$(G, C^{P2P}, 2)$	18	11	13
19	21	19.06	5.26	$(C, C^{P2P}, 1)$	21	26	23
20	21	19.44	5.81	$(C, C, 1)$	20	25	24
21	21	20.51	5.15	$(G, C, 3)$	19	14	18
22	21	20.77	5.16	$(G, C^{P2P}, 3)$	22	19	19
23	24	24.83	6.54	$(G, G, 2)$	23	18	20
24	25	24.79	4.10	$(G, C^{P2P}, 1)$	25	24	25
25	25	24.88	4.07	$(G, C, 1)$	26	23	26
26	26	25.95	6.56	$(G, G, 3)$	24	21	22
27	27	22.35	9.22	$(P, C, 1)$	27	29	28
28	27	22.47	9.17	$(P, C^{P2P}, 1)$	28	28	29
29	28	23.35	9.48	$(P, C, 2)$	29	30	30
30	29	23.82	9.46	$(P, C^{P2P}, 2)$	30	31	31
31	29	24.43	9.96	$(P, C, 3)$	32	32	32
32	30	24.84	10.18	$(P, C^{P2P}, 3)$	33	33	33
33	31	29.04	6.29	$(G, G, 1)$	31	27	27
34	35	35.09	1.47	$(P, G, 1)$	34	34	34
35	35	35.25	1.40	$(P, G, 2)$	35	36	35
36	35	35.32	1.44	$(P, G, 3)$	36	37	37
37	37	35.73	1.64	$(C, G, 1)$	37	35	36
38	38	38.03	1.33	$(C, G, 3)$	39	39	39
39	38	38.09	1.38	$(C, G, 2)$	38	38	38

Table IV: Configuration rankings under WC/L.

Table IV shows configuration rankings assuming worst-case, loaded system overheads (WC/L). Rankings under other overhead assumptions are given in Appendix A. The

[5]After testing a minimum of 500 task sets, additional task sets were generated until average schedulability fell within a three percentage-point interval with 95% confidence, or until $4,000$ task sets had been tested.

column labeled "Rank" gives each configuration's final rank. Observe that the table is sorted according to this column. The next three columns give the median, average, and standard deviation of each configuration's sub-ranks. Entries in the column labeled "(CPU,GPU,ρ)" identify the ranked system configuration. Here, we extend the tuple-notation from Sec. III to include ρ. The last three columns give the final rank of a configuration under the *other* overhead data sets. For a given row, we may compare the values of these columns against each other, and the value in the "Rank" column, to discern how a system configuration's ranking changes under different overhead conditions. We make the following observations.

Obs. 1. Clustered CPU scheduling with partitioned GPUs and $\rho = \infty$ had the highest rank under all overhead conditions.

We may observe this in the first row of Table IV by comparing the values for the Rank column against columns WC/I, AC/I, and AC/L—all have a first-place rank.

Obs. 2. Clustered CPU scheduling with partitioned GPUs and $\rho = \infty$ was not *always* the best configuration.

To see this, compare the Median and Average sub-rank values of the first row for (C, P, ∞). If (C, P, ∞) always had the highest rank, then Median and Average would both have a value of "1." They do not.

Obs. 3. Under partitioned GPUs, schedulability tends to be maximized when ρ is large. Namely, when $\rho = \infty$.

We may observe this by scanning the system configuration column, picking out entries matching $(*, P, *)$. Observe that entires that only differ by ρ tend to be ranked in decreasing ρ-order. For instance, (C, P, ∞), $(C, P, 3)$, $(C, P, 2)$, and $(C, P, 1)$ are ranked first, second, third, and ninth, respectively. This pattern is repeated for $(G, P, *)$ for ranks 13 through 16. The rankings for $(P, P, 2)$ (ranked fifth) and $(P, P, 3)$ (ranked sixth) are an exception to this trend. However, their average sub-ranks are very close: 7.75 and 7.91, respectively.

Obs. 4. Under clustered GPUs, schedulability tends to be maximized when $\rho = 2$.

Locate the $(*, C, 2)$ and $(*, C^{P2P}, 2)$ entries in Table IV. Observe that each entry, with one exception, has the highest rank among similar configurations that only differ by ρ. For example, $(C, C, 2)$ is ranked seventh while $(C, C, 3)$ and $(C, C, 1)$ are ranked 11*th* and 20*th*, respectively. The only exception to this trend is with the $(P, C, *)$ configurations. Here, $(P, C, 1)$ is ranked 27*th* and $(P, C, 2)$ is ranked 29*th*. It is highly likely that this exceptional behavior is due to the CK-OMLP, which is used to distribute GPU tokens in this case, but not the others. These same trends can be observed for rankings in the WC/L, AC/I, and AC/L columns, as well.

Obs. 5. Schedulability is comparably poor under the CK-OMLP.

We observe in Table IV that configurations $(P, G, *)$, $(P, C, *)$, $(P, C^{P2P}, *)$, and $(C, G, *)$ make up twelve of the thirteen *last* rankings under WC/L. This similarly holds under the other overhead data sets.

Obs. 6. System memory migrations offer better schedulability than P2P migrations.

Every clustered GPU configuration where P2P migrations were used was ranked lower than the similar configuration where system memory migrations were used. In most cases, the P2P-variant ranks closely below the other. For instance, $(C, C, 2)$ is ranked seventh and $(C, C^{P2P}, 2)$ is ranked tenth. The difference is similar under the other overhead data sets.

This is a disappointing result since we observed superior runtime performance under P2P migrations in [7]. The blocking complexity under P2P migrations is greater (refer to Appendix B for details), and this may result in more pessimistic blocking bounds.[6] Despite this disappointment, we observe that the rankings for system memory and P2P migration methods remain close enough that a system designer may opt to use P2P configuration for the sake of better runtime performance.

Obs. 7. Global GPU scheduling performed poorly.

$(G, G, 2)$ was the best global GPU configuration and it ranked 18*th* under AC/I, and in the twenties for the other data sets. Since global CPU scheduling performs comparatively well for $(G, P, *)$ (ranks 13*th*, 14*th*, and 15*th*), we do not believe this poor performance is due to greater scheduling overheads. Instead, it is most likely due to the additional GPU memory overheads incurred by using memory pages interleaved across the NUMA nodes, as we discussed in Sec. IV.

Obs. 8. Global CPU scheduling with partitioned GPUs performs well under AC overheads.

Locate the $(G, P, *)$ entries in Table IV at rankings 13 through 16. These rankings are relatively low. However, compare these to their rankings under AC/I and AC/L overhead data sets. $(G, P, *)$ does much better. Indeed (G, P, ∞) ranks second under AC/I instead of thirteenth under WC/L.

This completes our high-level comparisons of the various system configurations. We now take a deeper look at some of our results. Fig. 8 plots schedulability curves for the highest-ranked configuration of each high-level configuration under AC/L overheads. Tasks had a *medium* utilization, *moderate* period, *medium* data requirement, and no state. GPU kernel execution times were determined by the $[10\%, 25\%]$ uniform distribution. Finally, 66% of the tasks in each task set used a GPU. We make the following observations.

Obs. 9. Clustered GPU scheduling can be competitive with partitioned GPU scheduling.

Observe the curve for $(C, C, 2)$ (line 3) in Fig. 8 and compare it to curves for (C, P, ∞) and (G, P, ∞) (lines 1

[6]We discuss the challenge of determining tighter blocking bounds for P2P migrations in [21].

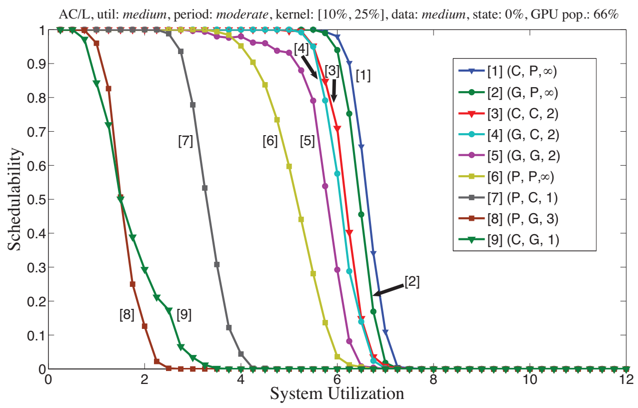

Figure 8: Detailed schedulability result.

and 2, respectively). Although schedulability is not as good, it is close. Moreover, $(C, C, 2)$ beats (P, P, ∞). This is interesting since (P, P, ∞) represents the natural extension of existing uniprocessor, uni-GPU methods. This result promotes $(C, C, 2)$ as an attractive choice in cases such as this because, in practice, it offers better resilience to GPU failure (or misbehaving GPU-using tasks) since a failed/locked-up GPU does not halt all GPU service to tasks within its cluster.

Obs. 10. Global CPU scheduling can perform well.

We see that (G, P, ∞) (line 2) and $(G, C, 2)$ (line 4) are competitive with their corresponding clustered-CPU configurations, (C, P, ∞) (line 1) and $(C, C, 2)$ (line 2). This supports the trend we identified in Obs. 8 through high-level observations. For example, (G, P, ∞) ranks fourth for AC/L in Table IV.

VI. Conclusion

GPUs have been advocated as accelerators in many settings where real-time constraints exist. However, to deploy GPUs in such settings, an understanding of schedulability-related tradeoffs is needed. In this paper, we have presented the first ever comprehensive study of such tradeoffs. Our study has focused particularly on multicore platforms augmented with potentially several GPUs. Multi-GPU multicore systems are an attractive platform for providing the necessary computational capacity for real-time applications that require intensive data-parallel processing, while adhering to stringent size, weight, and power envelopes.

The study reported in this paper was a significant undertaking, having examined nearly two billion different task systems and 156 different system configurations via experiments that took over 40,000 CPU hours to complete. Despite the extensive nature of this study, it had to be necessarily constrained. Specifically, we focused only on examining whether response-time bounds could be ensured. This motivated us to focus on one particular class of schedulers, namely the FL schedulers. In future work, we intend to expand this study to consider other real-time constraints and other classes of schedulers. This paper lays the groundwork for how such studies should be carried out.

Dealing with the vast amount of data produced by our study led to a secondary contribution: the development of a ranking method (inspired by the notion of weighted schedulability [12]) as a way of collapsing many thousands of schedulability graphs into a form that is much more succinct and allows trends to be more easily seen. While such a collapsing was necessary due to space constraints, any aggregation of data runs the risk of hiding important information. To guard against this, we have made all of our data available online in the form of a SQLite database file.

References

[1] J. Aumiller, S. Brandt, S. Kato, and N. Rath, “Supporting low-latency CPS using GPUs and direct I/O schemes,” in *18th RTCSA*, 2012.

[2] C. Basaran and K.-D. Kang, “Supporting preemptive task executions and memory copies in GPGPUs,” in *24th ECRTS*, 2012.

[3] K. Berezovskyi, K. Bletsas, and B. Andersson, “Makespan computation for GPU threads running on a single streaming multiprocessor,” in *24th ECRTS*, 2012.

[4] A. Betts and A. Donaldson, “Estimating the WCET of GPU-accelerated applications using hybrid analysis,” in *25th ECRTS*, 2013.

[5] 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.

[6] ——, “An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems,” *Real-Time Systems*, vol. 49, no. 2, 2013.

[7] G. Elliott, B. Ward, and J. Anderson, “GPUSync: A framework for real-time GPU management,” in *34th RTSS*, 2013.

[8] 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.

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

[10] J. Kim, B. Andersson, D. Niz, and R. Rajkumar, “Segment-fixed priority scheduling for self-suspending real-time tasks,” in *34th RTSS*, 2013.

[11] J. Erickson, J. Anderson, and B. Ward, “Fair lateness scheduling: reducing maximum lateness in G-EDF-like scheduling,” *Real-Time Sys.*, vol. 50, no. 1, 2014.

[12] B. Brandenburg, “Scheduling and locking in multiprocessor real-time operating systems,” Ph.D. dissertation, Univ. of North Carolina at Chapel Hill, 2011.

[13] N. Rath, J. Bialek, P. Byrne, B. DeBono, J. Levesque, B. Li, M. Mauel, D. Maurer, G. Navratil, and D. Shiraki, “High-speed, multi-input, multi-output control using GPU processing in the HBT-EP tokamak,” *Fusion Engineering and Design*, vol. 87, no. 12, 2012.

[14] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, “Data transfer matters for GPU computing,” in *19th ICPADS*, 2013.

[15] S. Baruah, “Scheduling periodic tasks on uniform multiprocessors,” *Information Processing Letters*, vol. 80, no. 2, 2001.

[16] ——, “Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms,” in *25th RTSS*, 2004.

[17] S. Funk, J. Goossens, and S. Baruah, “On-line scheduling on uniform multiprocessors,” in *22nd RTSS*, 2001.

[18] P. Gai, L. Abeni, and G. Buttazzo, “Multiprocessor DSP scheduling in system-on-a-chip architectures,” in *14th ECRTS*, 2002.

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

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

[21] G. Elliott and J. Anderson, "Appendix to exploring the multitude of real-time multi-GPU configurations," http://cs.unc.edu/~anderson/papers.html, December 2014.

[22] B. Ward and J. Anderson, "Fine-grained multiprocessor real-time locking with improved blocking," in *21st RTNS*, 2013.

[23] R. Pellizzoni and M. Caccamo, "Impact of peripheral-processor interference on WCET analysis of real-time embedded systems," *IEEE Transactions on Computers*, vol. 59, no. 3, 2010.

[24] "SchedCAT," http://mpi-sws.org/~bbb/projects/schedcat.

Appendix A.
Ranking Data

In this appendix, we present ranking data for the AC/I, AC/L, and WC/I experiments. This data is given in Tables V–VII.

Appendix B.
Analysis

In this appendix, we present a coarse analysis of blocking under GPUSync. Our main purpose here is to expose analytical differences among various configurations of GPUSync. Fine-grained blocking analysis can be found in [21]. We make the simplifying assumption that each job of any task competes for a GPU token at most once.

Due to our FL scheduler and locking protocols, we are constrained to "suspension-oblivious" schedulability analysis, where all task self-suspensions due to GPU operations and blocking are treated as CPU demand [20]. This is a safe, albeit pessimistic, technique. Nevertheless, utilization loss due to this pessimism can be easily offset by the performance gains offered by GPUs, which are often an order of magnitude faster than a CPU.

We must introduce some additional blocking-related notation. Let K_i denote the maximum token critical section length of T_i, b_i^K denote the maximum time $T_{i,j}$ may be blocked due to the token lock, and b_i^E denote the maximum time $T_{i,j}$ may be blocked *within* a token critical section for *all* engine locks. Then, the maximum time a job may be blocked accessing locks *and* tokens is given by $b_i \triangleq b_i^K + b_i^E$. Let N_i^I, N_i^O and N_i^S denote the number of chunks required to copy z_i^I, z_i^O, and z_i^S, respectively. Let X^I, X^O, and X^{P2P} denote the maximum time it takes to transmit a chunk of GPU data for input, output, and P2P migration, respectively, and let X^{max} denote the maximum of X^I, X^O, and X^{P2P}. Also, let S_i denote the maximum time to perform a GPU migration. For P2P migrations, $S_i = X^{P2P} N_i^S$. For migrations through system memory, $S_i = X^I N_i^S + X^O N_i^S$. For $(*, P)$ configurations, $S_i = 0$. Let E^{max} denote the longest duration an EE lock is held by any other task, and let K^{max} denote the longest token critical section among all tasks.

A job must acquire a token from the GPU allocator before it can begin using a GPU. When the R^{2}DGLP is in use, a token-requesting job is blocked by at most $2\lceil c/(\rho g)\rceil - 1$ token critical sections of other jobs [19]. Thus, the total duration of blocking while waiting for a token is bounded by $b_i^K = K^{max}(2\lceil c/(\rho g)\rceil - 1)$. Let M denote the number of CPUs that share a given GPU cluster. Under the CK-OMLP, a token-requesting job is blocked by at most $\lceil M/(\rho g)\rceil - 1$ token critical sections of other jobs [20]. However, all tasks, including those in T^C, may experience up to K^{max} blocking at release-time due to priority donation. $b_i^K = K^{max}\lceil M/(\rho g)\rceil$ when $T_i \in T^G$ and $b_i^K = K^{max}$ when $T_i \in T^C$. Bounds on K^{max} must be computed since tasks may block while acquiring engine locks. By construction, the token critical-section length for T_i is $K_i = q_i^{cpu} + e_i^{gpu} + b_i^E + X^I N_i^I + X^O N_i^O + S_i$. All these parameters have been derived, excepting b_i^E.

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

Rankings Under Average-Case Overheads, Idle							
Rank	Median	Avg	Std	(CPU,GPU,ρ)	WC/I	AC/L	WC/L
1	2	3.66	4.04	(C, P, ∞)	1	1	1
2	3	3.48	2.54	(G, P, ∞)	14	4	13
3	4	4.51	2.79	$(G, P, 3)$	15	5	14
4	4	4.94	3.96	$(C, P, 3)$	3	2	2
5	5	5.07	3.39	$(C, P, 2)$	2	3	3
6	5	5.21	2.53	$(G, P, 2)$	12	6	15
7	11	11.52	5.78	(P, P, ∞)	5	9	4
8	11	11.78	5.00	$(G, C, 2)$	17	10	17
9	11	12.02	4.70	$(C, C, 2)$	8	7	7
10	11	12.83	4.79	$(C, C^{P2P}, 2)$	10	8	10
11	12	12.65	5.14	$(G, C^{P2P}, 2)$	18	13	18
12	13	13.19	6.20	$(P, P, 2)$	4	12	5
13	13	13.81	5.37	$(P, P, 3)$	6	11	6
14	15	15.01	5.73	$(G, C, 3)$	19	18	21
15	15	15.07	5.17	$(C, C, 3)$	11	15	11
16	15	15.07	4.32	$(C, P, 1)$	7	14	9
17	15	14.30	5.66	$(G, P, 1)$	16	17	16
18	16	15.20	6.08	$(G, G, 2)$	23	20	23
19	16	15.26	5.44	$(G, C^{P2P}, 3)$	22	19	22
20	16	15.55	4.89	$(C, C^{P2P}, 3)$	13	16	12
21	18	17.58	6.64	$(G, G, 3)$	24	22	26
22	21	18.52	4.75	$(P, P, 1)$	9	21	8
23	24	22.22	6.11	$(G, C, 1)$	26	26	25
24	24	22.28	6.10	$(G, C^{P2P}, 1)$	25	25	24
25	24	23.16	3.43	$(C, C, 1)$	20	24	20
26	24	23.25	3.32	$(C, C^{P2P}, 1)$	21	23	19
27	27	24.70	9.90	$(G, G, 1)$	31	27	33
28	28	28.54	2.24	$(P, C^{P2P}, 1)$	28	29	28
29	28	28.58	2.25	$(P, C, 1)$	27	28	27
30	30	29.67	2.26	$(P, C, 2)$	29	30	29
31	30	29.70	2.27	$(P, C^{P2P}, 2)$	30	31	30
32	31	30.55	2.26	$(P, C, 3)$	32	32	31
33	31	30.78	2.25	$(P, C^{P2P}, 3)$	33	33	32
34	35	34.74	2.00	$(P, G, 1)$	34	34	34
35	35	35.10	2.12	$(C, G, 1)$	37	36	37
36	35	35.14	2.03	$(P, G, 2)$	35	35	35
37	36	35.32	2.03	$(P, G, 3)$	36	37	36
38	38	37.93	1.67	$(C, G, 2)$	38	38	39
39	38	37.99	1.73	$(C, G, 3)$	39	39	38

Table V: Configuration rankings under AC/I.

Rankings Under Average-Case Overheads, Loaded							
Rank	Median	Avg	Std	(CPU,GPU,ρ)	WC/I	AC/I	WC/L
1	2	3.15	3.59	(C, P, ∞)	1	1	1
2	3	4.01	3.32	$(C, P, 3)$	3	4	2
3	3	4.62	3.46	$(C, P, 2)$	2	5	3
4	4	6.08	6.39	(G, P, ∞)	14	2	13
5	5	7.01	6.09	$(G, P, 3)$	15	3	14
6	6	7.62	5.81	$(G, P, 2)$	12	6	15
7	9	10.77	4.48	$(C, C, 2)$	8	9	7
8	10	10.95	4.69	$(C, C^{P2P}, 2)$	10	10	10
9	11	12.02	5.49	(P, P, ∞)	5	7	4
10	12	13.96	5.80	$(G, C, 2)$	17	8	17
11	13	13.53	5.26	$(P, P, 3)$	6	13	6
12	13	13.91	5.70	$(P, P, 2)$	4	12	5
13	13	14.30	5.45	$(G, C^{P2P}, 2)$	18	11	18
14	14	12.84	5.02	$(C, P, 1)$	7	16	9
15	14	13.27	5.10	$(C, C, 3)$	11	15	11
16	15	13.59	5.05	$(C, C^{P2P}, 3)$	13	20	12
17	16	15.47	5.17	$(G, P, 1)$	16	17	16
18	17	16.31	5.93	$(G, C, 3)$	19	14	21
19	17	16.38	5.79	$(G, C^{P2P}, 3)$	22	19	22
20	20	20.07	6.69	$(G, G, 2)$	23	18	23
21	21	18.11	5.74	$(P, P, 1)$	9	22	8
22	22	21.04	6.86	$(G, G, 3)$	24	21	26
23	23	20.16	6.26	$(C, C^{P2P}, 1)$	21	26	19
24	23	20.84	6.28	$(C, C, 1)$	20	25	20
25	24	22.88	4.76	$(G, C^{P2P}, 1)$	25	24	24
26	24	23.45	4.88	$(G, C, 1)$	26	23	25
27	27	26.99	7.46	$(G, G, 1)$	31	27	33
28	28	27.98	4.25	$(P, C, 1)$	27	29	27
29	28	27.99	4.53	$(P, C^{P2P}, 1)$	28	28	28
30	30	28.61	4.18	$(P, C, 2)$	29	30	29
31	30	28.79	4.62	$(P, C^{P2P}, 2)$	30	31	30
32	31	29.41	4.54	$(P, C, 3)$	32	32	31
33	31	29.77	4.21	$(P, C^{P2P}, 3)$	33	33	32
34	35	35.22	1.63	$(P, G, 1)$	34	34	34
35	35	35.42	1.64	$(P, G, 2)$	35	36	35
36	36	35.46	1.79	$(C, G, 1)$	37	35	37
37	36	35.47	1.63	$(P, G, 3)$	36	37	36
38	38	37.97	1.39	$(C, G, 2)$	38	38	39
39	38	38.03	1.40	$(C, G, 3)$	39	39	38

Table VI: Configuration rankings under AC/L.

Rankings Under Worst-Case Overheads, Idle							
Rank	Median	Avg	Std	(CPU,GPU,ρ)	AC/I	AC/L	WC/L
1	3	4.84	4.35	(C, P, ∞)	1	1	1
2	3	5.12	4.35	$(C, P, 2)$	5	3	3
3	4	5.30	4.28	$(C, P, 3)$	4	2	2
4	6	7.82	6.60	$(P, P, 2)$	12	12	5
5	6	7.92	6.18	(P, P, ∞)	7	9	4
6	7	8.71	6.50	$(P, P, 3)$	13	11	6
7	9	9.97	4.53	$(C, P, 1)$	16	14	9
8	9	10.31	4.13	$(C, C, 2)$	9	7	7
9	10	10.66	8.00	$(P, P, 1)$	22	21	8
10	10	10.96	4.29	$(C, C^{P2P}, 2)$	10	8	10
11	12	12.57	4.64	$(C, C, 3)$	15	15	11
12	12	13.96	8.16	$(G, P, 2)$	6	6	15
13	12	12.97	4.60	$(C, C^{P2P}, 3)$	20	16	12
14	13	13.71	8.82	(G, P, ∞)	2	4	13
15	13	14.02	8.47	$(G, P, 3)$	3	5	14
16	18	18.40	5.81	$(G, P, 1)$	17	17	16
17	18	18.71	6.48	$(G, C, 2)$	8	10	17
18	18	19.01	6.16	$(G, C^{P2P}, 2)$	11	13	18
19	20	20.63	5.56	$(G, C, 3)$	14	18	21
20	21	18.69	5.99	$(C, C, 1)$	25	24	20
21	21	19.03	5.63	$(C, C^{P2P}, 1)$	26	23	19
22	21	20.64	5.45	$(G, C^{P2P}, 3)$	19	19	22
23	21	21.66	6.32	$(G, G, 2)$	18	20	23
24	24	23.44	6.61	$(G, G, 3)$	21	22	26
25	26	24.94	4.80	$(G, C^{P2P}, 1)$	24	25	24
26	26	24.99	4.81	$(G, C, 1)$	23	26	25
27	27	24.07	7.61	$(P, C, 1)$	29	28	27
28	27	24.42	7.48	$(P, C^{P2P}, 1)$	28	29	28
29	29	26.05	7.16	$(P, C, 2)$	30	30	29
30	29	26.37	6.92	$(P, C^{P2P}, 2)$	31	31	30
31	30	27.40	7.63	$(G, G, 1)$	27	27	33
32	31	27.53	7.04	$(P, C, 3)$	32	32	31
33	31	27.65	7.03	$(P, C^{P2P}, 3)$	33	33	32
34	35	34.55	2.50	$(P, G, 1)$	34	34	34
35	35	35.12	2.48	$(P, G, 2)$	36	35	35
36	36	35.15	2.63	$(P, G, 3)$	37	37	36
37	36	35.28	2.89	$(C, G, 1)$	35	36	37
38	38	38.28	0.85	$(C, G, 2)$	38	38	39
39	38	38.33	0.79	$(C, G, 3)$	39	39	38

Table VII: Configuration rankings under WC/I.

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

Bounds for $b_i^{I/O}$ and b_i^{P2P} depend on whether migrations are P2P or through system memory and on the number of CEs per GPU. In our analysis, we assume that all migrations are performed using the same method, though GPUSync could support both types in the same system.

CE blocking with P2P. Under P2P migrations, any task holding a GPU token may request the CE lock of the GPU it used in its prior job in order to perform a migration. There are at most ρg such tasks. In the worst case, they may all attempt to access the same CE lock at the same instant. Thus, *any* request for a CE lock may be blocked by $\rho g - 1$ other requests. From the blocking analysis of DGLs [22], the total number of blocking requests for a CE is at most $\rho g - 1$. Since no task requires more than X^{max} time to complete $b_i^{I/O} = X^{max}(N_i^I + N_i^O)(\rho g - 1)$ and $b_i^{P2P} = X^{max} N_i^S(\rho g - 1)$.

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

Analytical bounds for P2P and system memory migrations differ. CE lock contention is $O(\rho g)$ and $O(\rho)$ under P2P and system memory migrations, respectively. Despite its inferior order of complexity, P2P migration may still result in better analytical bounds if the advantages of fewer and faster memory copies can be exploited (it is faster because state is not copied to memory). Also, there are benefits to P2P migrations that cannot be captured in the above analysis, namely, isolation from the system memory bus and rarity of migrations due to the GPU allocator's heuristics.