TimeWall: Enabling Time Partitioning for Real-Time Multicore+Accelerator Platforms*

Tanya Amert, Zelin Tong, Sergey Voronov, Joshua Bakita, F. Donelson Smith, and James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill

***Abstract*—Across a range of safety-critical domains, an evolution is underway to endow embedded systems with “thinking” capabilities by using artificial-intelligence (AI) techniques. This evolution is being fueled by the availability of high-performance embedded hardware, typically multicore machines augmented with accelerators. Unfortunately, existing software certification processes rely on *time partitioning* to isolate system components, and this sense of isolation can be broken by accelerator usage. To address this issue, this paper presents TimeWall, a time-partitioning framework for multicore+accelerator platforms. When applied alongside existing methods for alleviating spatial interference, TimeWall can help enable component-wise certification on multicore+accelerator platforms. The challenges in realizing a TimeWall implementation are discussed in detail in this paper. Additionally, the temporal isolation TimeWall affords is examined experimentally, including via a case study of a computer-vision perception application, on a real platform.**

***Index Terms*—Real-Time Systems, Heterogeneous Architectures, Graphics Processing Units**

I. Introduction

The use of artificial-intelligence (AI) techniques to endow embedded systems with “thinking” capabilities is transforming the role these systems play in our everyday lives. A decade ago, the goal of producing aircraft and automobiles at mass scales that can autonomously “think” may have seemed far fetched, yet we are closer than ever to this reality today.

Unfortunately, as the evolution towards increasing AI functionality moves forward, a major stumbling block is looming: some of the most compelling use cases for embedded AI—such as autonomous aircraft and automobiles—fall within safety-critical domains for which certification is essential. The AI-based workloads of relevance to these use cases rely on complex hardware—typically multicore machines augmented with accelerators (*e.g.*, graphics processing units (GPUs)). These complexities present new challenges for certification.

Time and space partitioning. Existing safety-critical software certification processes are rooted in time and space partitioning. In avionics, for example, the sharing of hardware between software components is specified by a real-time operating system (RTOS) design in ARINC 653 [42]. The fundamental RTOS concept in ARINC 653 is a *partition*, which encapsulates a set of software modules and affords them isolation in time and space.[1]

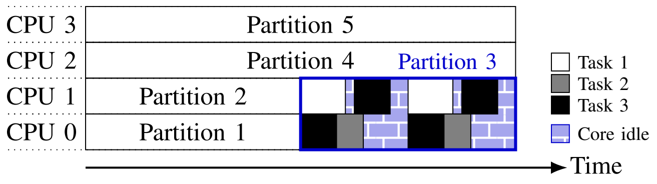


Fig. 1: Five partitions on four cores. Partition 3 executes on two cores; other partitions each execute on one core. In-partition scheduling is shown for Partition 3.

Broadly speaking, *time partitioning* means that at most one component may access each processing resource (*e.g.*, a CPU core) at any time, and *space partitioning* means that components cannot adversely interfere with each other in accessing non-processing resources, such as memory.

Space partitioning on multicore platforms can be provided by memory-protection functions (*e.g.*, page coloring). As depicted in Fig. 1, time partitioning in ARINC 653 is achieved via two-level scheduling: time slices are allocated to partitions, and for each partition, an in-partition scheduler allocates time to tasks of the contained component.

To evolve existing certification standards to embrace AI workloads, time-partitioning methods must enable accelerator usage. The key challenge here is that accelerators are typically invoked non-preemptively—this creates a hazard, as accelerator accesses too near the end of a time slice risk crossing the time-slice boundary, breaking cross-component isolation. Time-partitioning methods must prevent such scenarios—in essence enforcing a “wall in time” that accelerator accesses cannot cross.

Our focus. In this paper, we introduce TimeWall (Time-Isolated Multicore Execution With AcceLerator Locking), a framework for providing time partitioning on multicore+accelerator platforms. The design of TimeWall required new research contributions on several fronts. Before describing these contributions in detail, we briefly discuss prior work

*This work was supported by NSF grants CNS 1563845, CNS 1717589, CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

[1]We use *partition* to indicate an allocation by the OS and *component* when considering a portion of an application system that requires such an allocation. Each component executes within a unique partition.

related to resource sharing in component-based systems, as well as existing accelerator-access arbitration techniques.

Prior work. Component-based real-time systems have been investigated before [1], [4]–[11], [16], [20]–[23], [33], [34], [37], [38], [45], [52]. Some of this prior work considered shared resources, but only on uniprocessor platforms [5]–[7], [20] or multiprocessor platforms with each CPU dedicated to only one component [37], [38] (effectively, infinite time slices). To our knowledge, no prior work has explored the additional complexities introduced when components share accelerators, especially the challenges that arise when moving beyond theory to a practical implementation.

GPUs are probably the accelerator type of most relevance to AI workloads. GPU arbitration at the task level has been considered before, by using a real-time locking protocol [25], [31], [49], or via scheduling modifications in the GPU driver [18], [32]. However, to our knowledge, no prior work has considered GPU arbitration in component-based real-time systems.

Contributions. Prior work has not addressed the time partitioning of shared accelerators in component-based real-time systems. In this paper, we address this limitation by presenting TimeWall. Our contributions are threefold.

First, we present the design of TimeWall, which consists of three main parts: a table-driven scheduler that allocates time to partitions, in-partition global earliest-deadline-first (G-EDF) schedulers, and a specialized locking protocol that orchestrates accelerator accesses while respecting time-slice boundaries.

Second, we detail the challenges in moving from theory to practice when using GPUs as accelerators. In this setting, enforcing temporal isolation across components required significant effort due to unforeseen GPU-access edge cases. These edge cases, which we discuss in detail, highlight the need for *GPU budget enforcement*. We provide an approach for doing this, which we implemented in TimeWall.

Third, we present an experimental evaluation of TimeWall to demonstrate the temporal isolation it affords between GPU-using components. We explore the impact of various system partition configurations, and detail a case study of a computer-vision-based perception component in a LITMUS$^{\mathrm{RT}}$-based [12], [17] TimeWall implementation.

Organization. In the rest of this paper, we provide needed background (Sec. II), describe the design of TimeWall (Sec. III), detail the challenges in moving from theory to practice (Sec. IV), present our case-study evaluation (Sec. V), discuss our implementation in the broader context of response-time analysis for graph-based AI computations (Sec. VI), overview related work (Sec. VII), and conclude (Sec. VIII).

II. System Model and Background

Significant prior work has been directed at spatial isolation, most notably with respect to shared caches, buses, and DRAM banks (*e.g.*, see [43], [48], [57] and the references therein). We assume that techniques from this prior work are used alongside TimeWall to provide spatial partitioning for components (*e.g.*, that memory is partitioned and that data-movement costs are incorporated into timing analysis), and do not consider such techniques further.

In this paper, we focus our attention on ensuring temporal isolation between multiple components that require non-preemptive access to the same hardware accelerator; even if preemption is supported, such overheads are often prohibitively high. Time partitioning is violated if an accelerator access by one component extends beyond a time-slice boundary. Thus, a non-preemptive access must be postponed if it may cross a time-slice boundary.

Our solution to the time partitioning of accelerators involves using in-partition G-EDF schedulers alongside accelerator-access arbitration via a multiprocessor locking protocol. We now describe our system, task, and accelerator-request models, and the locking protocol we extended.

A. System Model

We consider a multicore+accelerator platform comprised of m identical CPUs alongside a set of accelerators. We allow different types of accelerators, *e.g.*, GPUs, digital signal processors (DSPs), and field-programmable gate arrays (FPGAs).

In this paper, we assume that all system components, and their assigned partition time slices, have been predetermined.[2] Associated with each component Γ is a set τ of tasks to be scheduled. During a time slice, Γ has exclusive access to a specified set Υ of CPU cores and accelerators. Note that some accelerators (*e.g.*, GPUs) can be broken into multiple virtual accelerators [39], [41]; we leave such sharing to future work.

B. Task Model

We assume that component Γ's task set τ is comprised of n implicit-deadline tasks. Each task τ_i releases a (potentially infinite) sequence of jobs: $J_{i,1}$, $J_{i,2}$, $\ldots$; we refer to an arbitrary job of task τ_i as J_i. We denote the period (and thus relative deadline) of a task τ_i as T_i, τ_i's worst-case execution time (WCET) as C_i, τ_i's utilization as $u_i = C_i/T_i$, and the total utilization of all tasks in Γ as $U = \sum_i u_i$.

In some AI applications, a task's utilization may exceed 1.0 [3]. Such tasks cannot be partitioned onto a single processor without over-utilizing the processor. Thus, for such tasks to be scheduled, CPU scheduling within a component must be done via global scheduling (*e.g.*, G-EDF), considering all jobs of all tasks in Γ together on the CPUs available to Γ, with consistent deadline tie breaking. Deadlines here define *priorities* rather than timing constraints—these deadlines may possibly be missed.[3] Thus, we consider a component Γ to be *schedulable* if, for each $\tau_i \in \tau$, we can compute a bound on the response time of τ_i (*i.e.*, the time between the release and subsequent completion of each job $J_{i,j}$). We leave the acceptability of such bounds as an application-level concern.

Task utilizations exceeding 1.0 also necessitate that some degree of intra-task parallelism be allowed. Prior work has

[2]In future work, we will explore issues arising when defining system components and time allocations. The results presented here are a necessary precursor to that work.

[3]The term "priority point" would be more appropriate than "deadline," but we use the latter to be consistent with the name "G-EDF."

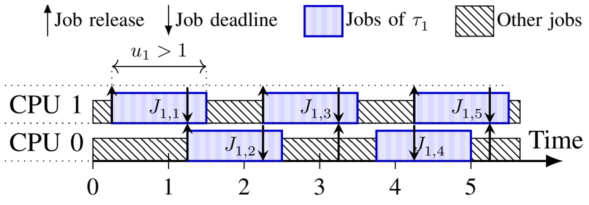


Fig. 2: An example schedule for an rp-sporadic task τ_1 with $u_1 = 1.25$ and $P_1 = 2$.

extended the traditional sporadic task model to allow full parallelism [26] or restricted parallelism [3], enabling the scheduling of tasks with utilizations exceeding 1.0. The *rp-sporadic task model* [3] introduces an additional task parameter, P_i, representing the number of jobs of task τ_i that may execute concurrently. We assume the rp-sporadic model in this paper, as it generalizes both the traditional sporadic task model ($\forall i : P_i = 1$) and the fully parallel model ($\forall i : P_i = m$).

Example 1. *An rp-sporadic task τ_1 with $u_1 = 1.25$ is depicted in Fig. 2. As $u_1 > 1$, τ_1 cannot be scheduled assuming the sequential sporadic task model, nor can it be partitioned to a single CPU. However, as $P_1 = 2$, up to two jobs of τ_1 may execute concurrently, e.g., at time 4.5.* ◊

C. Request Model

We consider accelerators to be shared resources that can be accessed by at most one job at a given time. Such accesses can be managed using a real-time mutual-exclusion locking protocol. A job J_i may issue one or more requests, $\mathcal{R}_i^1$, $\mathcal{R}_i^2, \ldots$, to the locking protocol; we let $\mathcal{R}_i$ denote an arbitrary request of a job of task τ_i. Once J_i is granted access, $\mathcal{R}_i$ is said to be *satisfied* until the job releases the lock. Request $\mathcal{R}_i$ is *active* from its issuance until J_i releases the lock; an active request is either waiting to acquire the lock or is satisfied.

D. Global OMLP

As discussed in Sec. II-B, allowing tasks with utilization exceeding 1.0 precludes partitioned scheduling, and we therefore require global scheduling (*e.g.*, G-EDF) and a multiprocessor mutual-exclusion locking protocol. One such protocol is the suspension-based global OMLP [14], which has been shown to have optimal priority-inversion blocking (pi-blocking) under suspension-oblivious analysis, which is the suspension-accounting method usually used under G-EDF.

When used on m processors, the global OMLP ensures $O(m)$ pi-blocking by utilizing a dual-queue structure, with an m-element FIFO queue fed into by a priority queue, as depicted in Fig. 3. Using the global OMLP, when a new request is issued, it is enqueued in the FIFO queue if fewer than m requests are already active, and in the priority queue otherwise. When the request at the head of the FIFO queue (*i.e.*, the lock holder) completes, it is dequeued, and the next request (if any) in the FIFO queue becomes satisfied; if the priority queue is not empty, the highest-priority request is moved from the priority queue to the tail of the FIFO queue.

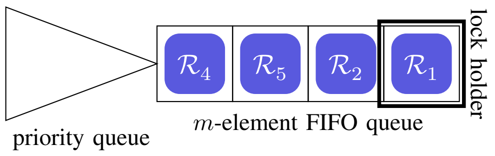


Fig. 3: The global OMLP structure for $m = 4$ CPUs.

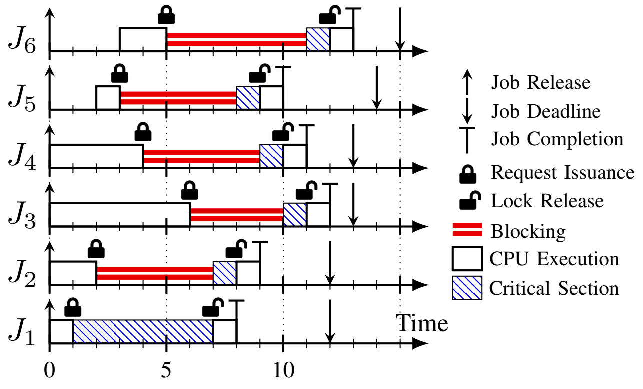


Fig. 4: Jobs issuing requests to the global OMLP with $m = 4$.

Example 2. *Fig. 4 depicts six jobs that issue requests to the global OMLP with $m = 4$. The global OMLP state shown in Fig. 3 corresponds to the set of active requests at time 4.5. The first four requests issued are enqueued directly in the FIFO queue in issuance order; thus, $\mathcal{R}_5$ is satisfied before $\mathcal{R}_4$, even though J_4 has higher priority than J_5.*

Requests $\mathcal{R}_6$ and $\mathcal{R}_3$ are enqueued in the priority queue upon issuance, as the FIFO queue is full. When J_1 releases the lock at time 7, $\mathcal{R}_2$ becomes satisfied (i.e., the head of the FIFO queue), and $\mathcal{R}_3$ is moved from the priority queue to the FIFO queue, as J_3 has higher priority than J_6. Thus, $\mathcal{R}_3$ is satisfied before $\mathcal{R}_6$, despite being issued later. ◊

E. Accelerator Access Model

We refer to the computations performed while a request is satisfied as its *critical section*. We assume that each job of a task τ_i may make any number of accelerator accesses and that successive accesses to the *same* accelerator may be grouped into the critical section of a single request. We denote by $Y_i^{k,1}$, $Y_i^{k,2}, \ldots$ the accelerator accesses occurring during the critical section of request $\mathcal{R}_i^k$.

Example 3. *Two request-issuing jobs are depicted in Fig. 5. Job J_2 makes two separate lock requests, whereas the two accesses by job J_1 are grouped into a single request.*

Request $\mathcal{R}_2^2$ is active in the interval $[5, 9)$: it is blocked by both accesses of $\mathcal{R}_1^1$ from time 5 to time 8 and then satisfied from time 8 to time 9 while $\mathcal{R}_2^2$'s critical section executes. ◊

III. TimeWall

Certification procedures tend to evolve slowly over time (with good reason!). Given this reality, our proposal for ensuring time partitioning on a multicore+accelerator platform is based on the current ARINC 653 time-slicing approach.

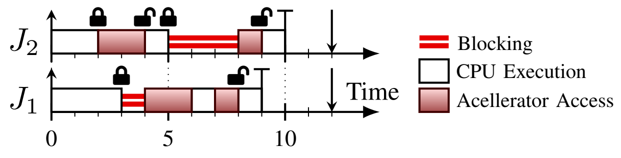


Fig. 5: Two jobs issuing requests to access the same accelerator.

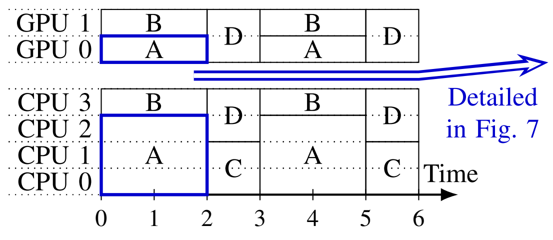


Fig. 6: Time-sliced schedule. Rectangles represent reservations.

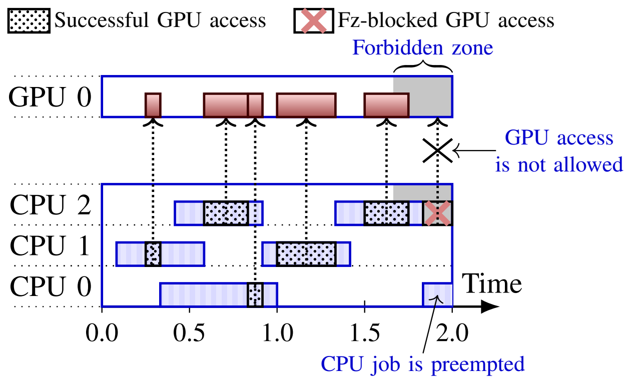


Fig. 7: Expanded view of the first time slice of Component A from Fig. 6. Rectangles represent jobs of tasks in Component A.

We abstract the idea of time slicing by ensuring that each component Γ is granted exclusive periodic access to a set Υ of computing resources (accelerators as well as M unit-speed CPUs, where $M \leq m$) by defining a *periodic component reservation* (PCR) for Γ (similarly to the Single Time Slot Periodic Partition model of Mok and Chen [36]).

The PCR for component Γ is defined as a three-tuple (Θ, Π, Υ), denoting that Γ receives exclusive access to the computing resources in Υ within continuous intervals of Θ time units that begin every Π time units ($\Theta \leq \Pi$). As an example, Fig. 6 shows the first few time slices for four components on a platform with four CPUs and two GPUs. In this example, Component A is specified by $(2, 3, \{\text{CPU } 0, \text{CPU } 1, \text{CPU } 2, \text{GPU } 0\})$.

We now introduce TimeWall, a framework to enable time partitioning on multicore+accelerator platforms.

A. Scheduling Hierarchy

At the core of TimeWall is a two-level scheduling hierarchy. The top-level scheduler is the *partition allocator* (PA), which ensures that partitions are scheduled according to their PCRs. The PA is realized using a table-driven scheduling approach. We assume that the table is determined offline; optimizing the table creation is outside the scope of this paper.

The second-level scheduler is the *in-partition scheduler*. Conceptually, any multiprocessor scheduler could be used here. Our implementation uses G-EDF, but other G-EDF-like schedulers [27] could be applied similarly. In the rest of the paper, we focus on the allocations within a single arbitrary component, so we will simply refer to that component's in-partition scheduler as "the scheduler."

To realize this scheduling hierarchy, we extended the existing reservation-based scheduling mechanisms available in LITMUS$^{\text{RT}}$. In our implementation, each reservation is contained within a scheduling environment: the PA corresponds to an environment that schedules partition reservations, and each in-partition scheduler is associated with an environment that schedules task reservations. Using this hierarchical approach, a job can query the remaining budget for its partition, which is necessary to enforce time partitioning, as discussed next.

B. Time Partitioning via Forbidden Zones

As discussed in Sec. II-D, we require a multiprocessor locking protocol to ensure mutually exclusive access to each accelerator, and we chose the global OMLP for this purpose. We apply the global OMLP within a component, treating each accelerator in Υ as a separate resource protected by a unique global OMLP lock with an M-element FIFO queue. We leave the exploration of other protocols to future work.

The key challenge of time partitioning accelerators is preventing the continuation of a non-preemptive accelerator access past the end of a time slice. To prevent such time-slice overruns, we use a variant of a concept known as a *forbidden zone* [30].[4] Defined for each access to a non-preemptive accelerator, a forbidden zone is the time interval in which the access may not be *initiated*, otherwise it may not complete before the end of the component's time slice. Thus, the length of the forbidden zone for a given access is the worst-case duration of that access—*accelerator usage by other components has no impact on a given component's forbidden-zone lengths*. Note that the use of a forbidden zone requires that no accelerator access takes more than Θ time units.

Example 4. *Forbidden zones are illustrated in Fig. 7, which shows a detailed view of the execution of Component A from Fig. 6 within the time interval* $[0, 2)$. *The forbidden zone corresponding to the final GPU access by a job executing on CPU 2 is shown in grey, prior to the time-slice boundary.* $\Diamond$

The enforcement of forbidden zones in TimeWall is applied at the level of an *individual accelerator access*, rather than an entire critical section. To enable this fine-grained enforcement, we augmented the global OMLP to include an additional "forbidden-zone-check" mechanism in addition to the traditional "lock" and "unlock" functionality. Prior to initiating an accelerator access $Y_i^{k,\ell}$, the job J_i holding the lock must invoke the forbidden-zone check, which verifies that $Y_i^{k,\ell}$ is not within its forbidden zone, *i.e.*, that the time remaining

[4]A similar idea was later applied in a component-based setting [6], but that work focused on uniprocessor CPU platforms and did not allow for skipping ahead in a forbidden zone, which we discuss later.

in the time slice is at least the worst-case duration of $Y_i^{k,\ell}$. Otherwise, the forbidden-zone check suspends J_i until the next time slice of its containing component. We call such forbidden-zone-induced blocking *fz-blocking*.

Example 4 (cont'd)**.** *As shown in Fig. 7, the last job to execute on CPU 2 accesses GPU 0 twice. The first GPU access is initiated at time* 1.5*, before the start of its forbidden zone; this access is allowed to execute, as it will complete before the end of the time slice, by the definition of the forbidden-zone length. The second GPU access is fz-blocked, and cannot begin until Component A's next time slice. CPU execution, like that on CPU 0, is allowed and is preempted at time* 2.0*.* ◊

C. Performance Optimizations

The delays caused by fz-blocking can be mitigated slightly via two performance improvements, which we discuss now.

Skipping ahead. If the lock holder is fz-blocked due to an accelerator access in its forbidden zone, we can allow other requests to "skip ahead" of that access until the beginning of the next time slice. This corresponds to the *Skip Protocol* proposed previously [30], but requires some additional machinery due to the separate enforcement of forbidden zones and critical sections. Because allowing a request to skip ahead reorders the global OMLP's queues of requests, *critical-section lengths*[5] rather than individual access durations are compared to the time remaining in the slice to determine whether skipping ahead is allowed. A consequence is that individual accesses are not permitted to skip ahead; we leave to future work such access-level skipping, which would require additional mechanisms to ensure that the global OMLP remains starvation-free in the presence of time-slice boundaries.

Example 5. *Consider the set of active lock requests depicted in Fig. 8, in which accesses are represented by clouds and requests are depicted as rounded rectangles. Suppose that after $Y^{1,1}$ completes, the job becomes fz-blocked, i.e., that the worst-case duration of $Y^{1,2}$ is longer than the time remaining in the time slice.*

In this case, a later-enqueued request may be allowed to skip ahead. The next request that can be satisfied is $\mathcal{R}^5$, as its critical-section length is less than the worst-case duration of $Y^{1,2}$. Note that $\mathcal{R}^3$ is not eligible, even though its individual accesses are each of shorter duration than $Y^{1,2}$. ◊

Requests that skip ahead do so while another job is fz-blocked, and thus do not introduce any additional blocking.

Merging accelerator accesses. Each request for accelerator access incurs delays due to locking-protocol overhead as well as pi-blocking. Such blocking depends on the worst-case duration of any critical section [50].

As we allow multiple accelerator accesses within a critical section, a trade-off arises as to whether to merge successive accesses into a single critical section. If a job J_i acquires

[5]Skipping ahead requires that the `lock` call of the global OMLP be modified to take the critical-section length as an additional parameter.

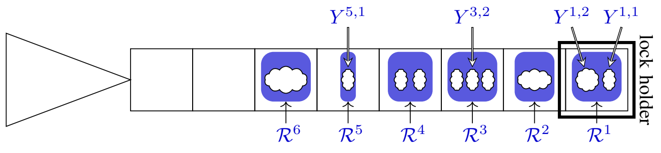


Fig. 8: The global OMLP with support for forbidden zones. Widths of rounded rectangles (requests) and clouds (accesses) indicate worst-case critical-section and access durations, respectively.

the lock for each access individually, the locking-protocol overhead and pi-blocking suffered by J_i are duplicated for each access. However, if the accesses are merged, then the single critical section has longer duration than if the lock were separately acquired for each access, increasing the blocking experienced by other jobs. We discuss the overhead associated with this trade-off for our implementation in Sec. V-A.

IV. Theory Meets Practice

In theory, forbidden zones ensure total isolation between accelerator-using components, *but does such isolation occur in practice?* To answer this question, we implemented the two-level scheduler and forbidden-zone-aware global OMLP comprising TimeWall within the 5.4.0-rc7 LITMUS$^{\mathrm{RT}}$ kernel [12], [17] and conducted experiments involving a GPU, perhaps the most commonly used type of accelerator in work on AI. These experiments led to several surprises, which we detail throughout this section.

Experimental setup. Our experimental platform contains two eight-core 2.10-GHz Intel Xeon Silver 4110 processors and one NVIDIA Titan V GPU. Each CPU core utilizes a 32-KB L1 instruction cache, a 32-KB L1 data cache, and a 1-MB L2 cache; all eight CPU cores on a socket share an 11-MB L3 cache. Only one socket was used to schedule components in our experiments. The CPUs and discrete GPU have separate DRAM, and we used platform-default mechanisms to manage concurrent bus access. To mitigate CPU spatial interference between components, we partitioned the L3 cache and main memory evenly between them. We also disabled hyperthreading and graphics output for all experiments.

Our case study, described in Sec. V, featured a GPU-enabled version of the pedestrian-detection algorithm *Histogram of Oriented Gradients (HOG)* [19]. HOG utilizes seven GPU operations: a CPU-to-GPU image copy-in, five GPU-local computations (termed *kernels*—not to be confused with an OS kernel), and one final GPU-to-CPU result copy-out.

To provision forbidden zones for the seven GPU operations in HOG, we needed the worst-case duration of each access. Due to the complexity of multicore platforms, the industry standard for WCET analysis on such platforms is measurement based [51]. Thus, for 25,000 video frames, we collected GPU- and CPU-based GPU-access-duration measurements, using NVIDIA's `nvprof` profiling tool and `clock_gettime()`, respectively. The results are listed in Table I, including the 99^{th}, 99.5^{th}, 99.9^{th}, 99.95^{th}, and 99.99^{th} CPU percentiles.

Perplexing edge cases. We expected the CPU-measured GPU-access durations to approximately match what we measured

TABLE I: Statistics for durations of the two copies and five kernels comprising the HOG case study in microseconds, as measured on the GPU using `nvprof` and on the CPU using `clock_gettime()`.

Device	Statistic	Copy-In	K1	K2	K3	K4	K5	Copy-Out
GPU	max	77	27	42	56	28	49	29
CPU	99^{th}	154	144	138	73	30	64	46
	99.5^{th}	157	146	139	74	31	65	47
	99.9^{th}	180	154	146	79	45	70	52
	99.95^{th}	200	161	148	86	49	76	57
	99.99^{th}	1391	1342	163	1265	55	1236	69
	max	5247	1393	1388	1332	1286	1317	1300

on the GPU (with minor CPU-GPU communication overhead), yet this behavior only held up to the 99.95^{th} percentile of CPU measurements. In fact, the worst-case CPU-measured times were *two orders of magnitude* higher than those measured on the GPU. *What was causing such extreme edge cases, and why had prior work not noted these cases?*

Prior work seems to have obviated edge cases either by only timing GPU accesses on the GPU [29], [35], [54] or by using only a percentile of measured CPU times [53], [55], [56]. We can make no such simplification. Provisioning forbidden zones using lower percentiles would risk GPU accesses crossing time-slice boundaries, violating temporal isolation. However, as the average- and worst-case diverge, fz-blocking using worst-case measurements becomes exceedingly conservative, greatly reducing GPU utilization. Only one choice remained: we needed to identify the cause of the edge cases.

A. Investigating Potential Culprits

Prior work [24], [54] identified pitfalls in using GPUs in real-time systems, providing us with a starting point. After ruling out the pitfalls listed by Yang *et al.* [54], we suspected GPU interrupt handling as the edge-case culprit, as Elliott and Anderson [24] detailed priority inversions that could occur due to interrupt processing for CPU-GPU communication in prior kernel and GPU driver versions.

Interrupt processing. In Linux (upon which LITMUS$^{\mathrm{RT}}$ is based), interrupts are processed in two steps: "top halves," which typically execute immediately and non-preemptively to acknowledge the interrupt, and "bottom halves," which handle the interrupt processing itself. Despite recent changes in Linux interrupt handling [28], [44], bottom halves in LITMUS$^{\mathrm{RT}}$ may still execute at a lower priority than any LITMUS$^{\mathrm{RT}}$ task, leading to potential priority inversions.

We mitigated edge cases due to top halves by reserving one CPU per socket for top-half handling. To identify if bottom-half-related priority inversions were causing the edge cases, we used KUTrace [46], [47], a tracing tool that provides a timeline of all work on the system, including interrupts, syscalls, and page faults. The trace data surprisingly revealed that interrupts were not at fault; each edge case occurred entirely in userspace. However, this raised the question: what source could account for a userspace slowdown of this magnitude?

Power management. The trace data revealed an additional oddity, providing our next clue: just before a worst-case GPU access completed, all CPUs other than the one awaiting notification of a GPU-operation completion would exit a low-power state. Thus, we sought to disable power management, including both low-power states and CPU frequency scaling.

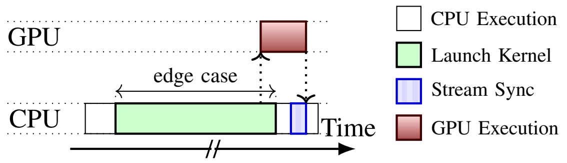
For low-power states, we disabled Linux's `cpuidle` mechanism [15]. Frequency scaling proved more challenging, and in fact, we discovered that it is impossible to completely disable frequency scaling in modern Intel processors. Thus, we implemented a monitor to periodically log CPU frequency. We correlated these measurements with our trace data, and observed no CPU-frequency changes during edge-case occurrences. To identify the userspace operations in which the edge cases occurred, we added KUTrace markers around each CPU-side operation occurring within our GPU-access-duration timing interval; we found that the edge cases occurred within library functions used to communicate with the GPU.

CUDA runtime library overhead. NVIDIA provides the CUDA runtime library for communicating with an NVIDIA GPU. For example, there are CUDA functions to submit a GPU kernel to the GPU (`cudaLaunchKernel`) and await its completion (`cudaStreamSynchronize`).

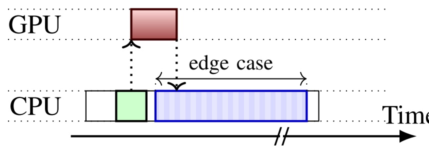
By aligning a GPU trace using NVIDIA's `nvprof` tracing tool with our KUTrace results, we observed two unexpected scenarios, shown in Fig. 9. Fig. 9a illustrates a scenario in which the asynchronous `cudaLaunchKernel` took multiple milliseconds (this call typically takes tens of microseconds). In the other scenario, depicted in Fig. 9b, the `cudaStreamSynchronize` call did not return until multiple milliseconds after the GPU execution had completed.[6]

Unfortunately, due to the closed-source nature of the NVIDIA ecosystem, discovering the root cause of these edge cases is exceedingly difficult. Furthermore, we may have observed only one of many possible edge cases in using such black-box GPUs. Thus, ***having to deal with edge cases is an unavoidable consequence of using an NVIDIA GPU***. Consequently, to support a robust real-time system, edge-case mitigation is a necessity.

[6]We use the CUDA option `cudaDeviceScheduleYield` to suspend on the CPU while waiting for the GPU operation to complete, but we observed the same results by instead spinning via `cudaDeviceScheduleSpin`.



(a) A long `cudaLaunchKernel` call.



(b) A long `cudaStreamSynchronize` call.

Fig. 9: Illustrations of two edge-case scenarios we observed using KUTrace and `nvprof`.

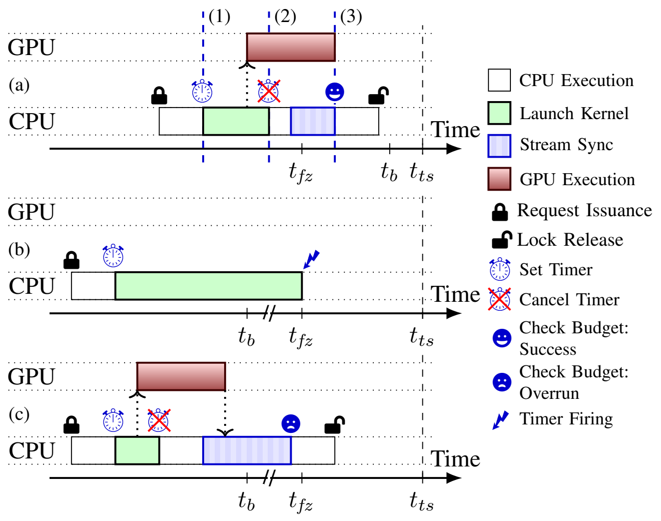


Fig. 10: The budget-enforcement mechanisms used in TimeWall, for (a) a "well-behaved" GPU access, (b) a GPU access for which the watchdog timer fires, and (c) a GPU access that exceeds its budget.

B. Mitigating Edge Cases through Budget Enforcement

The edge cases just discussed highlight our dual need to ensure that our provisioned GPU-related execution times are both respected and reasonable. The classic way of providing such assurance is by enforcing *budgets*, and we do that here. However, in the case of GPU operations, budget enforcement is trickier than for CPU-only tasks.

Our budget-enforcement solution follows a two-pronged approach, which we integrate into the forbidden-zone-aware global OMLP. To enforce forbidden zones we utilize a watchdog timer, and provision forbidden zones using worst-case GPU-measured access durations. To handle budget overruns, we monitor GPU access durations and cancel any additional job processing if a budget overrun occurs. We provision per-access budgets using 99.95^{th}-percentile CPU-measured access durations. To our knowledge, this is the first GPU budget-enforcement strategy to be proposed for real-time systems.

Enforcing forbidden zones. The watchdog timer is controlled via a pair of syscalls (as part of the forbidden-zone-aware global OMLP) for each accelerator access. Note that, like lock/unlock calls, these sycalls are performed by application code, and therefore forbidden-zone enforcement relies on an unenforced programming convention.

The first syscall takes two parameters: the access budget and the forbidden-zone length. If the time remaining in the component's time slice is less than the budget, the job is suspended until its component's next time slice. Otherwise, the timer is set to fire at the start of the forbidden zone (*i.e.*, the time-slice end minus the forbidden-zone length). After the syscall completes, the budget-expiration time is set within the LITMUS$^{\text{RT}}$ userspace libraries. The second syscall takes no parameters—it simply cancels the timer (if it has not already fired) after the operation launch has completed; a GPU operation initiated before its forbidden zone begins will complete before the time-slice boundary.

Example 6. *Our budget-enforcement mechanisms are illustrated for a single GPU access in Fig. 10. In each inset, the budget expires at time t_b, the forbidden zone begins at t_{fz}, and the time slice ends at t_{ts}.*

The "well-behaved" case is depicted in Fig. 10a. In this example, the first syscall occurs at time (1); the budget is less than the time remaining in the time slice, so the timer is set to fire at time t_{fz} (calculated as t_{ts} minus the forbidden-zone length), t_b is set (calculated as the current time plus the budget), and the access is allowed to proceed. After the kernel launch completes (time (2)), the watchdog timer is cancelled via the second syscall. As the access begins before t_{fz}, it is allowed to execute within its forbidden zone. Once the access completes, the budget is checked (time (3)); this check occurs before time t_b, so the budget has not expired. ◊

If the timer fires, the timer callback function immediately suspends the job, ensuring that the GPU operation will not be initiated in the current time slice.

Example 6 (cont'd)**.** *The scenario depicted in Fig. 10b corresponds to the edge case in Fig. 9a. In this scenario, the first syscall sets the watchdog timer, but an edge case occurs during the kernel launch, so the timer fires at time t_{fz} before the second syscall occurs. The job is immediately suspended, so the GPU operation is not submitted in this time slice. Note that, due to this edge case, the budget check will also fail.* ◊

Handling budget overruns. Unfortunately, it is not simple to immediately kill a misbehaving GPU-using task, as this invalidates the CUDA context shared by any other tasks in the same process. Thus, we must wait until the `cudaStreamSynchronize` call completes to enforce GPU budgeting. If the access completes after the budget expires, the LITMUS$^{\text{RT}}$ userspace libraries send a `SIGSYS` signal to the application, which must handle the budget overrun.[7]

[7]Note that, like forbidden-zone enforcement, handling of the `SIGSYS` signal relies on unenforced programming conventions.

Example 6 (cont'd)**.** *In Fig. 10c, the edge case occurs after the GPU operation completes, as in Fig. 9b, so temporal isolation of the GPU is not violated. However, the* `cudaStreamSynchronize` *call completes after the access's budget expires at time t_b, so a* `SIGSYS` *signal is sent to the application.* ◊

In our HOG case study, the `SIGSYS` signal resulted in stopping all processing for the current video frame, *i.e.*, the frame was dropped—in the AI use cases that motivate this work, occasionally cancelling work (*e.g.*, dropping a video frame) is often deemed as acceptable. The choice of provisioning for GPU-access budgeting provides an interesting trade-off between job completion and system utilization; provisioning GPU-access budgeting on a lower percentile enables better utilization of the processors available to the partition at the cost of a higher number of jobs (*i.e.*, frames) being dropped. We explore this trade-off in Sec. V.

V. Experimental Evaluation of TimeWall

In this section, we present an experimental evaluation of our TimeWall implementation using both synthetic experiments and a case study featuring the HOG application. All experiments were performed on the platform described in Sec. IV.

A. Temporal Isolation and the Cost of Enforcement

We first discuss synthetic experiments we performed to verify temporal isolation and to quantify the overheads associated with forbidden-zone enforcement.

Verifying temporal isolation. We designed two GPU-using tasks to test temporal isolation. `GPU-LIGHT`, which accesses the GPU at the start of each time slice, is affected by any GPU interference: the `GPU-LIGHT` kernel executes for a given number of cycles, and if any GPU operation from another component overruns the time-slice boundary, the `GPU-LIGHT` task's response time will increase. `GPU-HEAVY`, on the other hand, submits GPU kernels near the end of each time slice, attempting to cause GPU interference.

We validated that our TimeWall implementation achieves temporal isolation and measured any context-switch costs due to alternating components sharing the GPU. We used two components, each with 16-ms time slices; we executed one `GPU-LIGHT` task in the first component, and measured its response time with different workloads in the second component. Our results are shown in Fig. 11, with the second component containing (a) nothing, one `GPU-LIGHT` task (b) with or (c) without forbidden-zone enforcement, or one `GPU-HEAVY` task (d) with or (e) without forbidden-zone enforcement.

We used the approach of Capodieci *et al.* [18] to calculate the theoretical context-switch cost using parameters of our Titan V GPU: with a GPU-internal bus bandwidth of 652.8 GB/s and 64K 4-byte registers per streaming multiprocessor (SM), 128 KB of L1 cache per SM, 4.5 MB of L2 cache, and 80 SMs, it takes about 51.65 μs to store or load the GPU state.

Observation 1. *Observed GPU-context-switch costs match our theoretical calculation.*

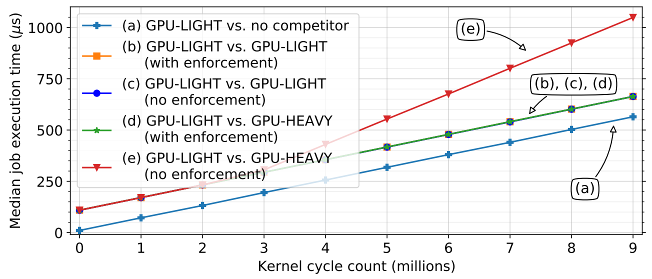


Fig. 11: Comparison of median `GPU-LIGHT` task execution times in the presence of alternating component workloads (the middle 50th percentile varied by less than 3%).

Curves (a) and (b)–(c) in Fig. 11 correspond to our `GPU-LIGHT` task running in isolation and against another `GPU-LIGHT` task, respectively; in neither case should GPU operations cross time-slice boundaries. Demonstrating this, the slopes of these curves are nearly identical, with a near constant offset between (a) and (b)–(c) ranging from 98.2 to 99.2 μs. A context switch requires both a store and a load, resulting in a cost of about 103.3 μs, which is in line with our measurements.

Observation 2. *TimeWall enforces temporal isolation between GPU-using components.*

The difference between curves (d) and (e) in Fig. 11 indicates the benefit of our watchdog timer. As curves (b), (c), and (d) are nearly identical, we can also observe that our forbidden-zone enforcement reduce any temporal interference to just the cost of GPU context switches.

Overhead of forbidden-zone enforcement. We used Feather-Trace [13] to measure the overheads[8] associated with enforcing forbidden zones for our TimeWall implementation. We performed a synthetic experiment with 30 GPU-using tasks in a component with $M = 5$ CPU cores; the overhead for each GPU access was 0.9 μs to set the watchdog timer, and an additional 1.2 μs if the timer fired.

To consider the trade-off associated with merging multiple GPU accesses into one critical section, we compare the timer overheads to the lock and unlock calls, for which we observed overhead costs of 0.7 μs and 4.7 μs, respectively. (The high unlock overhead is expected due to ensuring priority inheritance for the next lock holder.) These measurements suggest potential benefits of merging a few accesses into one critical section; *e.g.*, two individual requests incur a total of 12.6-μs overhead ($0.7 + 0.9 + 4.7 = 6.3$ μs each), but one request that comprises two accesses incurs only $0.7 + 0.9 + 0.9 + 4.7 = 7.2$-$\mu$s overhead. We plan to perform a full schedulability study in future work to explore this trade-off further.

B. Choosing a Time-Slice Length

The overhead and GPU context-switch costs discussed in Sec. V-A become increasingly relevant for shorter time slices.

[8]We used 99th-percentile measurements. LITMUS^{RT} was developed for academic research purposes to investigate functionality that *could be* fielded in an RTOS for safety-critical contexts. We take the 99th-percentile as representative of achievable worst-case overheads in a well-honed RTOS.

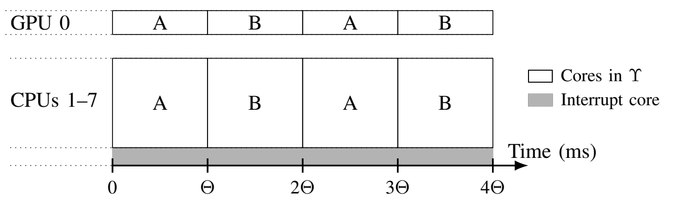


Fig. 12: Reservations for the two components in our time-slice experiments. We measured response times of tasks in Component A.

However, long time slices can result in high response times due to long intervals when jobs are released but not scheduled, *e.g.*, if $\Theta = 100$ ms, $\Pi = 200$ ms, and $T_i = 25$ ms for some task τ_i, then four jobs of τ_i may be released but not scheduled during each time interval that τ_i's component does not execute. To explore the "sweet spot" between these two time-slice-length extremes, we performed experiments using randomly generated synthetic tasks.

Components and partitions. Our time-slice experiments used two components, scheduled as shown in Fig. 12. Components A and B alternately utilized CPUs 1–7 and the GPU. We reserved CPU 0 for interrupt handling, and the other socket (CPUs 8–15) for non-real-time tasks. We used Intel's Resource Directory Technology (via the `resctl` command) to partition the L3 cache and DRAM between components.

Component A was comprised of GPU-using tasks with $C_i = 4$ ms; each job spun on the CPU for its WCET, and accessed the GPU once at a random point in its execution, with $Y_i^{1,1}$ uniformly chosen from $[0.02, 0.04]$ ms ("short"), $[0.2, 0.4]$ ms ("medium"), or $[2, 4]$ ms ("long"). For each of these three access-duration ranges, we generated a task set for Component A by adding randomly generated tasks until we reached utilization restrictions [50]. In Component B, we executed a cache-thrashing workload designed to evict the contents of the per-core L1 and L2 caches.

We varied Θ and Π while maintaining the ratio $\Theta/\Pi = 0.5$, and measured the response times of the tasks in Component A for two minutes. We set Θ to be powers of two from 0.5 ms to 256 ms. To observe the impact of time-slice-aligned job releases, we separately performed experiments in which periods of tasks in Component A were uniformly chosen to be either 32 ms or 64 ms ("aligned") or 25 ms or 50 ms ("unaligned"). We assumed periodic tasks for these experiments, so releases being aligned with time slices means that for $\Pi \geq T_i$, a job was released at the start of each time slice; it is possible that multiple jobs were released during a time slice, *e.g.*, if $\Theta = 64$ ms and $T_i = 32$ ms.

Job releases aligned with time slices. The results of our aligned-releases experiments are shown in Fig. 13 for each range of GPU-access durations. Note that for a task system to be schedulable, we require $\Theta \geq Y_i^{k,\ell}$, so we do not include response-time curves for long GPU-access durations (up to 4 ms) with $\Theta < 4$ ms.

Observation 3. *Extremely short time slices can cause response times to be unbounded.*

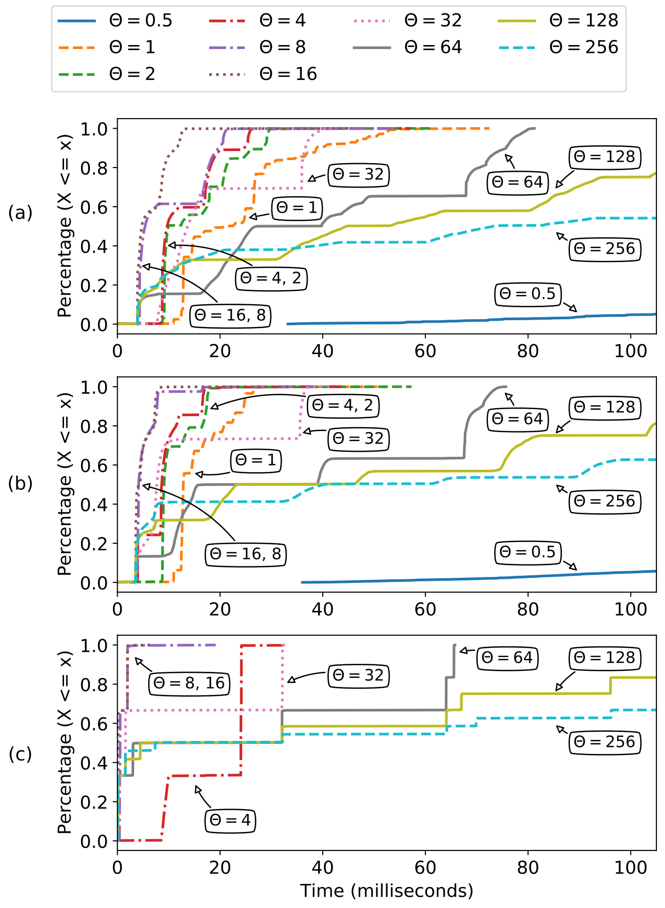


Fig. 13: CDFs of job response times with releases aligned to time slices, for (a) short, (b) medium, and (c) long GPU accesses.

This can be observed for the 0.5-ms curve in Fig. 13a. For such short time slices, forbidden zones, scheduling and locking overhead, and context-switch costs took a larger proportion of the capacity available to the component. This was even more pronounced for the medium GPU-access durations in Fig. 13b; forbidden zones took up to 80% of the 0.5-ms time slices.

Observation 4. *Extremely long time slices can result in prohibitively large response times.*

For $\Theta \geq 64$ ms and $T_i = 32$ or 64 ms, multiple jobs may have been released while the component was not scheduled, resulting in high response times in all scenarios in Fig. 13.

Observation 5. *The lowest response times occurred for $\Theta > C_i$ and $\Pi \leq T_i$.*

In all three scenarios, the lowest response times occurred for $\Theta = 16$ ms ($\Pi = 32$ ms) and $\Theta = 8$ ms ($\Pi = 16$ ms). Thus, the best reservation had time slices longer than the WCET of any task and a period at most that of any task (*i.e.*, a job was released no more frequently than every time slice of Component A).

Job releases unaligned with time slices. The results of our unaligned-releases experiments are shown in Fig. 14. In these experiments, we observed that $\Theta = 16$ ms and $\Theta = 8$ ms also resulted in the lowest response times for tasks with unaligned

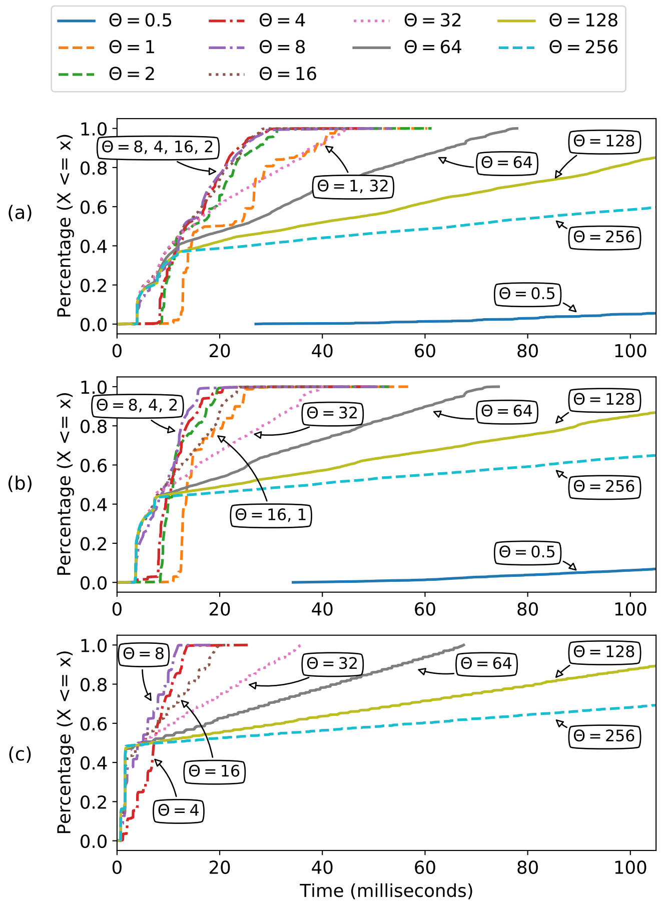


Fig. 14: CDFs of job response times with releases unaligned to time slices, for (a) short, (b) medium, and (c) long GPU accesses.

releases. This indicates that these time-slice budgets were not artifacts of aligned job releases, but rather, for our task-system parameters, provided the best balance of the negative effects of overhead and fz-blocking with the amount of time a component was not scheduled.

C. Case-Study Evaluation

We now describe the components present in our case-study evaluation, and then present our evaluation of TimeWall using a real workload.

Components and partitions. Our case study used the same partitions depicted in Fig. 12, with different workloads running in both Components A and B. We again partitioned the DRAM and the L3 cache between components.

Component A was comprised of three instances of HOG, each with a period of 40 ms (corresponding to 25 frames per second) and a parallelization level of $P_i = 2$. Component B contained seven cache-thrashing and GPU-using tasks designed to evict the contents of the per-core L1 and L2 caches, cause GPU context switches, and stress the watchdog timer. These tasks had a period of 160 ms and each job accessed the GPU for 5 ms.

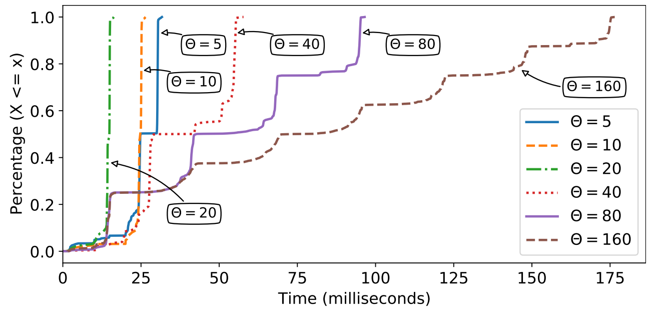


Fig. 15: CDFs of HOG response times with varying Θ.

The HOG algorithm. As discussed in Sec. IV, the HOG algorithm processes a video frame by copying the image to the GPU, detecting pedestrians at multiple image-scale levels (we used 13 levels in our experiments), and then copying the results for each level back from the GPU. Thus, processing each frame takes 78 GPU operations: one copy-in, 64 kernels (kernels K2–K5 operate on all scale levels, and K1 operates on all but the first), and 13 copy-out operations on the GPU. We configured each HOG task to process one frame per job.

Choosing time-slice lengths for HOG. We performed experiments similar to those in Fig. 13 for the HOG tasks in Component A, with each instance processing 5,000 video frames. We aligned job releases with time slices (in practice, real-time systems typically have harmonic periods), and chose Θ values ranging from 5 ms to 160 ms. The results of our HOG time-slice experiments are depicted in Fig. 15.

Observation 6. *The lowest response times for HOG tasks occurred for* $\Theta = 20$ *ms.*

This matches what we expect from Obs. 5. The HOG tasks had a period of 40 ms, so for $\Theta = 20$ ms, each task released a job each time slice of Component A. The worst-case response time we observed was 16.31 ms, which corresponds to completing a job within the time slice in which it was released. All remaining experiments discussed in this section thus used $\Theta = 20$ ms.

Frame dropping due to GPU-budget enforcement. The provisioning choice for budgeting GPU accesses greatly impacts the number of frames dropped. In our implementation, if any of HOG's 78 GPU accesses exceeds its budget, the frame is dropped. We treat each access as an independent random variable with probability ρ of not exceeding its budget. Thus, ρ^{78} is the probability that all 78 accesses for a given frame do not exceed their budgets, and the probability of dropping a frame in HOG is given by $1 - \rho^{78}$. We chose a value of ρ to provision GPU accesses in HOG as described below.

This expected frame-drop rate of $1 - p^{78}$ is plotted as the "Theoretical" curve in Fig. 16. The other curves correspond to the observed frame-drop rates for different values of a safety-margin multiplier we apply to CPU-based measurements of GPU access times for provisioning budgets. To measure these frame-drop rates, we provisioned GPU-access budgeting using these different multipliers, and ran three instances of HOG for 25,000 frames each, counting the number of frames dropped.

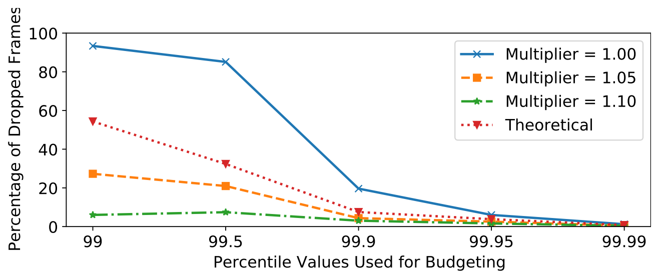


Fig. 16: Measured frame-drop rates for HOG depending on multiplier used for provisioned GPU-access budgeting values, compared to the theoretical frame-drop rate.

Observation 7. *Theoretical frame-drop rates increase rapidly when provisioning on less than the* 99.9^{th} *percentile.*

For 99.9^{th}-percentile measurements, the theoretical frame-drop rate is 7.5%. However, this rises to 32.4% and 54.3% for the 99.5^{th}- and 99^{th}-percentile measurements, respectively.

Observation 8. *Using a multiplier of* 1.0 *can result in highly conservative budgeting.*

This can be seen by comparing the 1.0-multiplier and "Theoretical" curves in Fig. 16. For example, provisioning using exactly the 99.9^{th} percentile resulted in almost a $3\times$ increase in the frame-drop rate over the expected value.

Observation 9. *Increasing budget provisioning by a small multiplier can cause a sizeable reduction in frame-drop rates.*

Provisioning on the 99.95^{th} percentile with multiplier 1.1 resulted in a 1.6% frame-drop rate, compared to the expected rate of 3.8%. By using a slightly higher multiplier, we can use lower GPU-access measurements, enabling better platform utilization. Thus, for the remaining HOG experiments in this subsection, we used a multiplier of 1.1 and provisioned budgets at the 99.95^{th} percentile.

The cost of enforcing time partitioning. Next, we sought to understand the trade-offs associated with using TimeWall with a real workload. For this, we measured the response times of the HOG tasks in Component A with no workload in Component B (*i.e.*, during Component B's time slices, the CPUs and GPU were idle), and varied the method of enforcing temporal isolation between GPU-using components. We compared no GPU management (*i.e.*, no guarantee of isolation), employing the global OMLP (but without support for forbidden zones), and using TimeWall (including forbidden-zone enforcement and CPU budgeting) to ensure temporal isolation.

To also consider the effects of interference between HOG tasks, we performed experiments with three or one HOG task executing in Component A. Our results are shown in Fig. 17.

Observation 10. *Using TimeWall resulted in only a slight increase in average-case response times for HOG tasks.*

This difference is visible for three HOG tasks in Fig. 17a. This increase is most likely the result of enforcing forbidden zones. When only the global OMLP is employed, accesses are allowed to begin within the forbidden zone, which can result in temporal-isolation violations, but increases the amount of work that may complete within a given time slice.

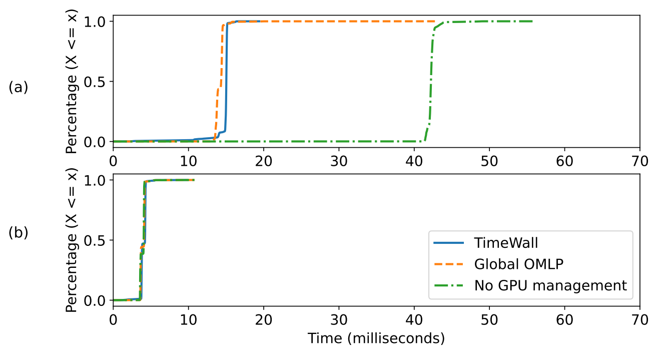


Fig. 17: CDFs of HOG response times with different GPU-management approaches, with either (a) three or (b) one HOG task(s) in Component A.

Observation 11. *Using TimeWall resulted in a lower worst-case response time than using the global OMLP.*

The worst-case response times for HOG using either the global OMLP or TimeWall were 42.9 ms and 19.4 ms, respectively. This difference is likely due to GPU-access budgeting employed by TimeWall, which terminates a job when it experiences a GPU-access timing anomaly. Using just the global OMLP allows for work to continue after a timing anomaly, which can extend a job's execution long enough to cross a time-slice boundary (recall that we chose $\Theta = 20$ ms).

Observation 12. *When GPU accesses were not managed, GPU contention between multiple HOG tasks greatly increased response times.*

This can be observed by comparing the "No GPU management" curves in Fig. 17. In our experiments, each HOG task was executed as a separate process in Linux with its own CUDA context. When only a single HOG task was present in Component A and GPU accesses were not managed (as in inset (b)), the average and worst-case response times were 3.9 ms and 11.0 ms, respectively. However, when executing three HOG tasks without using any locking protocol (as in inset (a)), the average and worst-case response times were 42.3 ms and 55.9 ms, respectively; almost every job took more than a single time slice, and thus was delayed an additional 20 ms while Component A was not active. This increase in response times was most likely due to multiprogramming effects on the GPU [2], which were not present when only one HOG task executed in Component A.

Putting it all together. We executed the three HOG tasks in Component A for a total of 75,000 frames (25,000 each), either alone or alongside our cache-thrashing and GPU-using workload in Component B, and measured the observed response time for each frame and the overall number of frames dropped. These values are listed in Table II. We also plot the cumulative distribution function of the observed response times in Fig. 18.

Observation 13. *HOG tasks in Component A were not significantly affected by the presence of tasks in Component B.*

This is supported by the observed response times in Table II and Fig. 18 and the number of frames dropped in Table II.

TABLE II: Observed response times across 75,000 frames for HOG instances in Component A without or with tasks executing in Component B.

	A	A+B
Observed Maximum Response Time (ms)	19.40	16.54
Observed Average Response Time (ms)	14.81	14.61
Number of Dropped Frames	1212	1238

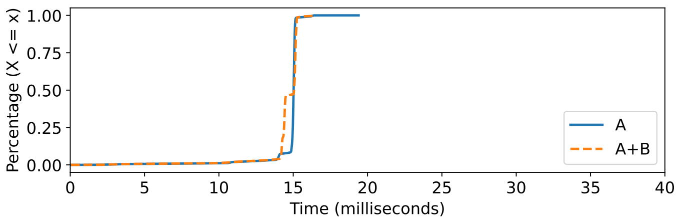


Fig. 18: CDFs of observed response times for the HOG tasks in Component A without and with a workload in Component B.

Note that for the scenario in which we executed tasks in both Components A and B, there were slightly more dropped frames. This freed up system capacity, reducing the response times of other HOG jobs in Component A, thereby lowering the maximum response time for the "A+B" curve in Fig. 18.

Observation 14. *The frame-drop rate of HOG matched the expected rate for our budget provisioning choice.*

The frame-drop rates were 1.6% and 1.7% when only Component A executed work and when both components executed work, respectively. This matches our expectation of 1.6% given our provisioned GPU-access budgets. (Recall the discussion following Obs. 9.)

VI. Discussion

While we have focused here on experimental trade-offs, the theoretical underpinnings of TimeWall's time-partitioning approach have been investigated in a companion paper [50]. That work more broadly considers AI computations specified as processing graphs, presents blocking analysis associated with forbidden zones, and shows how to compute (and optimize) end-to-end graph response-time bounds. Graphs are scheduled in that work by treating their individual nodes as rp-sporadic tasks. In this paper, we chose to consider such tasks only, as a full consideration of graphs was not possible due to space constraints. Nonetheless, all of our conclusions related to forbidden zones, time isolation, *etc.*, apply to graphs, and our TimeWall implementation fully supports such graphs.

VII. Related Work

Component-based multiprocessor real-time systems have been investigated before [1], [4]–[11], [16], [20]–[23], [33], [34], [37], [38], [45], [52]. Of prior work that considered shared resources, most focused on resource sharing between components on a uniprocessor platform [5]–[7], [20] or when each core of a multiprocessor platform is dedicated to only one component [37], [38]. Biondi *et al.* [10] considered the higher-level problem of how to partition applications into components, with shared resources considered, and Xu *et al.* [52] explored the impact of cache interference between components on the same platform. Notions related to the forbidden-zone concept, which was proposed earlier [30], were used in some of this work [6], [38], but not for managing accelerator accesses.

GPU accesses are typically managed in one of two ways. Accesses can be arbitrated via a real-time locking protocol or other middleware [25], [31], [49]. For example, GPUSync [25] provides mutually exclusive accesses for systems with multiple GPUs. Another approach is to either utilize the existing GPU-scheduling rules [2], [40], or to modify the GPU driver scheduling policies [18], [32]. Capodieci *et al.* explored implementing EDF scheduling on an NVIDIA GPU, whereas Kato *et al.* presented TimeGraph [32], which uses non-preemptive fixed-priority scheduling. However, arbitration of GPU accesses has not been previously explored in component-based real-time systems.

VIII. Conclusion

We have presented TimeWall, a time-partitioning approach for real-time systems deployed on multicore+accelerator platforms. TimeWall's design was motivated by the need to enable component-wise certification of AI-based embedded applications in safety-critical settings. It utilizes a modified multiprocessor locking protocol to maintain the invariant that each time-partitioned component always has exclusive access to all processing resources allocated to it. We discussed the specific challenges associated with realizing a full scheduler implementation of TimeWall for a multicore+GPU platform, and demonstrated the isolation properties afforded by TimeWall via experiments on actual hardware involving a synthetic tasks as well as a computer-vision application.

In future work, we plan to consider locking-protocol options other than the global OMLP used in our current TimeWall prototype. We also intend to explore accelerators that support multiple virtual accelerators. Finally, we plan to further examine higher-level issues that arise when breaking a system into certifiable components, such as how to divide a set of applications into individual components.

References

[1] Kunal Agrawal, Alan Burns, Abhishek Singh, and Sanjoy Baruah. Minimizing execution duration in the presence of learning-enabled components. In *Proceedings of the 24th Design, Automation and Test in Europe Conference and Exhibition*, pages 1644–1649, 2020.

[2] Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 104–115, 2017.

[3] Tanya Amert, Sergey Voronov, and James H Anderson. OpenVX and real-time certification: The troublesome history. In *Proceedings of the 40th IEEE Real-Time Systems Symposium*, pages 312–325, 2019.

[4] Madhukar Anand, Arvind Easwaran, Sebastian Fischmeister, and Insup Lee. Compositional feasibility analysis of conditional real-time task models. In *Proceedings of the 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 391–398, 2008.

[5] Moris Behnam, Thomas Nolte, Mikael Sjodin, and Insik Shin. Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems. *IEEE Transactions on Industrial Informatics*, 6(1):93–104, 2009.

[6] Moris Behnam, Insik Shin, Thomas Nolte, and Mikael Nolin. SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *Proceedings of the 7th ACM and IEEE International Conference on Embedded Software*, pages 279–288, 2007.

[7] Marko Bertogna, Nathan Fisher, and Sanjoy Baruah. Resource-sharing servers for open environments. *IEEE Transactions on Industrial Informatics*, 5(3):202–219, 2009.

[8] Enrico Bini, Marko Bertogna, and Sanjoy Baruah. Virtual multiprocessor platforms: Specification and use. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 437–446, 2009.

[9] Enrico Bini, Giorgio Buttazzo, and Marko Bertogna. The multi supply function abstraction for multiprocessors. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 294–302, 2009.

[10] Alessandro Biondi, Giorgio Buttazzo, and Marko Bertogna. A design flow for supporting component-based software development in multi-processor real-time systems. *Real-Time Systems*, 54(4):800–829, 2018.

[11] Jalil Boudjadar, Jin Hyun Kim, Linh Thi Xuan Phan, Insup Lee, Kim Larsen, and Ulrik Nyman. Generic formal framework for compositional analysis of hierarchical scheduling systems. In *Proceedings of the 21st IEEE International Symposium on Real-Time Distributed Computing*, pages 51–58, 2018.

[12] Björn B Brandenburg. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 2011.

[13] Björn B Brandenburg and James H Anderson. Feather-trace: A lightweight event tracing toolkit. In *Proceedings of the 3rd International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 20–27, 2007.

[14] Björn B Brandenburg and James H Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 49–60, 2010.

[15] Neil Brown. Improvements in CPU frequency management. *LWN.net*, April 2016. URL: https://lwn.net/Articles/682391/.

[16] Artem Burmyakov, Enrico Bini, and Eduardo Tovar. Compositional multiprocessor scheduling: the GMPR interface. *Real-Time Systems*, 50(3):342–376, 2014.

[17] John Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–126, 2006.

[18] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for GPU with preemption support. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, pages 119–130, 2018.

[19] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Proceedings of the IEEE Computer Vision and Pattern Recognition Conference*, pages 886–893, 2005.

[20] Robert I Davis and Alan Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 257–270, 2006.

[21] Zhong Deng and Jane W-S Liu. Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 308–319, 1997.

[22] Arvind Easwaran, Madhukar Anand, and Insup Lee. Compositional analysis framework using EDP resource models. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 129–138, 2007.

[23] Arvind Easwaran, Insik Shin, and Insup Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, 43(1):25–59, 2009.

[24] Glenn A Elliott and James H Anderson. Robust real-time multiprocessor interrupt handling motivated by GPUs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 267–276, 2012.

[25] Glenn A Elliott, Bryan Ward, and James H Anderson. GPUSync: a framework for real-time GPU management. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, pages 33–44, 2013.

[26] Jeremy P Erickson and James H Anderson. Response time bounds for G-EDF without intra-task precedence constraints. In *Proceedings of the 15th International Conference On Principles Of Distributed Systems*, pages 128–142, 2011.

[27] Jeremy P Erickson and James H Anderson. Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 3–11, 2012.

[28] Thomas Gleixner. genirq: Forced threaded interrupt handlers, February 2011. Message to the Linux kernel mailing list.

[29] Seonyeong Heo, Sungjun Cho, Youngsok Kim, and Hanjun Kim. Real-time object detection system with multi-path neural networks. In *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 174–187, 2020.

[30] Philip Holman and James H Anderson. Locking under Pfair scheduling. *ACM Transactions on Computer Systems*, 24(2):140–174, 2006. (an earlier version appeared at RTSS 2002).

[31] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Raj Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 57–66, 2011.

[32] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the USENIX Annual Technical Conference*, pages 17–30, 2011.

[33] Nima Khalilzad, Moris Behnam, and Thomas Nolte. On component-based software development for multiprocessor real-time systems. In *Proceedings of the 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 132–140, 2015.

[34] Hennadiy Leontyev and James H Anderson. A hierarchical multi-processor bandwidth reservation scheme with timing guarantees. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 191–200, 2008.

[35] Xinxin Mei, Xiaowen Chu, Hai Liu, Yiu-Wing Leung, and Zongpeng Li. Energy efficient real-time task scheduling on CPU-GPU hybrid clusters. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications*, pages 1–9, 2017.

[36] Aloysius K Mok, Xiang Feng, and Deji Chen. Resource partition for real-time systems. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium*, pages 75–84, 2001.

[37] Farhang Nemati, Moris Behnam, and Thomas Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 251–261, 2011.

[38] Farhang Nemati and Thomas Nolte. Resource sharing among real-time components under multiprocessor clustered scheduling. *Real-Time Systems*, 49(5):580–613, 2013.

[39] NVIDIA. Multi-process service. Online at https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2020. Version R450.

[40] Ignacio Sañudo Olmedo, Nicola Capodieci, Jorge Luis Martinez, Andrea Marongiu, and Marko Bertogna. Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective. In *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 213–225, 2020.

[41] Nathan Otterness and James H Anderson. Exploring AMD GPU scheduling details by experimenting with "worst practices". In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, pages 24–34, 2021.

[42] Paul J Prisaznuk. ARINC 653 role in integrated modular avionics (IMA). In *Proceedings of the 27th IEEE/AIAA Digital Avionics Systems Conference*, pages 1–E, 2008.

[43] Shahin Roozkhosh and Renato Mancuso. The potential of programmable logic in the middle: cache bleaching. In *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 296–309, 2020.

[44] Marta Rybczyńska. Modernizing the tasklet API. *LWN.net*, September 2020. URL: https://lwn.net/Articles/830964/.

[45] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 181–190, 2008.

[46] Richard Sites. Benchmarking "hello, world!": Six different views of the execution of "hello, world!" show what is often missing in today's tools. *ACM Queue*, 16(5):54–80, October 2018. doi:10.1145/3291276.3291278.

[47] Richard Sites. *Understanding Software Dynamics*. Addison-Wesley Professional, 2021.

[48] Muhammad R Soliman and Rodolfo Pellizzoni. PREM-based optimal task segmentation under fixed priority scheduling. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, pages 4:1–4:23, 2019.

[49] Uri Verner, Avi Mendelson, and Assaf Schuster. Scheduling periodic real-time communication in multi-GPU systems. In *Proceedings of the 23rd International Conference on Computer Communication and Networks*, pages 1–8, 2014.

[50] Sergey Voronov, Stephen Tang, Tanya Amert, and James H Anderson. AI meets real-time: Addressing real-world complexities in graph response-time analysis. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, 2021.

[51] Reinhard Wilhelm. Real time spent on real time. In *Proceedings of the 41st IEEE Real-Time Systems Symposium*, pages 1–2, 2020.

[52] Meng Xu, Linh Thi Xuan Phan, Oleg Sokolsky, Sisu Xi, Chenyang Lu, Christopher Gill, and Insup Lee. Cache-aware compositional analysis of real-time multicore virtualization platforms. *Real-Time Systems*, 51(6):675–723, 2015.

[53] Ming Yang, Tanya Amert, Kecheng Yang, Nathan Otterness, James H Anderson, F Donelson Smith, and Shige Wang. Making OpenVX really “real time”. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 80–93, 2018.

[54] Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H Anderson, and F Donelson Smith. Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, pages 20:1–20:21, 2018.

[55] Ming Yang, Shige Wang, Joshua Bakita, Thanh Vu, F Donelson Smith, James H Anderson, and Jan-Michael Frahm. Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 305–317, 2019.

[56] Shuochao Yao, Yifan Hao, Yiran Zhao, Huajie Shao, Dongxin Liu, Shengzhong Liu, Tianshi Wang, Jinyang Li, and Tarek Abdelzaher. Scheduling real-time deep learning services as imprecise computations. In *Proceedings of the 26th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, 2020.

[57] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 155–166, 2014.