Predictable GPU Sharing in Component-Based Real-Time Systems

Syed W. Ali
Department of Computer Science, University of North Carolina at Chapel Hill, NC, USA

Zelin Tong
Department of Computer Science, University of North Carolina at Chapel Hill, NC, USA

Joseph Goh
Department of Computer Science, University of North Carolina at Chapel Hill, NC, USA

James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill, NC, USA

Abstract

This paper presents a real-time locking protocol whose design was motivated by the goal of enabling safe GPU sharing in time-sliced component-based systems. This locking protocol enables a GPU to be shared concurrently across, and utilized within, isolated components with predictable execution times. It relies on a novel resizing technique where GPU work is dimensioned on-the-fly to run on partitions of an NVIDIA GPU. This technique can be applied to any component that internally utilizes global CPU scheduling. The proposed locking protocol enables increased GPU parallelism and reduces GPU capacity loss with analytically provable benefits.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases GPU locking protocols, real-time locking protocols, priority-inversion blocking, component-based systems

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2024.15

Supplementary Material *Software (ECRTS 2024 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.10.1.1`

Funding Supported by NSF grants CPS 2038960, CPS 2038855, CNS 2151829, and CPS 2333120.

1 Introduction

Advances in computationally intensive AI and ML workloads require state-of-the-art hardware capable of providing significant parallelism. GPUs meet the demands of such work, which would otherwise execute too slow on general-purpose CPUs. When supporting such intensive workloads on computationally diverse hardware platforms composed of both multicore CPUs and GPUs, certifying real-time safety can be challenging. One way to ease this challenge is by breaking a large monolithic system into separate components that can be validated independently. However, such an approach requires some means for safe GPU sharing, both across components and by tasks within the same component.

Unfortunately, how a GPU can be safely shared in this way is not clear. Components require *isolation guarantees* that necessitate judicious management of compute resources, such as CPUs and GPUs. A shared GPU makes isolation guarantees difficult, as GPU work in one component may slow other concurrent GPU work in other components. Requiring each component to have a dedicated GPU largely solves interference concerns, but such a solution may require an excessive number of GPUs that will likely be underutilized. Efficient GPU usage is crucial as size, weight, and power, and cost (SWaP-C) constraints may inhibit the ability to add more hardware.

36th Euromicro Conference on Real-Time Systems (ECRTS 2024).
Editor: Rodolfo Pellizzoni; Article No. 15; pp. 15:1–15:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Additional concerns stem from how a component may internally utilize a GPU. Coarse-grain locking of the entire GPU enables predictable execution times but comes at the cost of inefficient utilization. This utilization loss is suffered when tasks *block* on one another when waiting for exclusive GPU access. Reliably and efficiently meeting the increasing computational demands of modern systems motivates designing a fine-grain real-time locking protocol that realizes concurrent GPU sharing and can be accommodated using existing real-time schedulability-analysis techniques.

Fine-Grain GPU Sharing. Existing methods for predictable GPU sharing require modifying the GPU-side workload to achieve temporally isolated access to the full GPU [13, 20, 15]. In such methods, GPU code, which is typically non-preemptive, must be modified to stop and resume within allotted access times. Other recently proposed methods allow concurrently scheduled GPU work by limiting what can be co-scheduled [16, 21]. In these methods, interference from co-scheduled work can be mitigated analytically by always assuming the worst observed interference.

Recent work on `libsmctrl` [3] enables GPU sharing by spatially partitioning the compute units on an NVIDIA GPU. That work enables the allocation of individual *streaming multiprocessors* (SMs). An SM consists of a set of processing elements (CUDA cores) and is the basic unit of hardware compute allocation on an NVIDIA GPU. Using `libsmctrl` [3] has the benefit of not requiring GPU workload modifications. Instead, only the manner in which CPU-side code launches GPU work changes. However, to enable the *safe* concurrent execution of work on a spatially partitioned GPU, some mechanism for allocating a GPU's internal processing elements is needed.

When a component is guaranteed access to a set of SMs, a locking protocol can be utilized within the component for fine-grain exclusive locking of individual SMs. However, current locking protocols are ill-suited for doing this. Coarse-grain locks on the entire GPU do not efficiently utilize the available compute capacity. Simple mutex-based locks on SMs do not allow work to run on any SM available. A k-exclusion lock, where a lock is placed on any k SMs, results in inefficient GPU utilization as a job must wait for its specified number of SMs to become available. In contrast, `libsmctrl` [3] can initiate work on any number of available SMs. To exploit this feature, a locking protocol is needed that allows GPU work to be dimensioned on-the-fly to fit on the available SMs.

This work proposes the *Streaming Multiprocessor Locking Protocol* (SMLP), a fine-grain SM locking protocol for systems implemented within time-sliced, isolated components. Shown in Fig. 1 is a component-based system where jobs (CPU-side work) in component A initiate GPU-side work on SMs allocated by the SMLP. Job J_1 utilizes all SMs available to A but cannot utilize SM2 as it is guaranteed to component B. Assume J_2 does not benefit from being allocated more than one SM. In this case, the SMLP allocates only one SM to J_2, efficiently matching its compute requirements. As shown by J_3, the SMLP may also delay GPU-side work if it cannot finish before the end of its component's current time slice.

This work extends TimeWall [1], a component-based scheduling framework, to enable efficient GPU sharing across components. Within a component, the SMLP employs a novel *resizing* technique where arriving GPU work is dynamically pinned to available SMs using `libsmctrl` [3]. We specifically analyze globally scheduled job-level fixed-priority (JLFP) task systems within a component. We analytically derive bounds on priority-inversion blocking (pi-blocking), *i.e.*, the duration for which a task is delayed by lower-priority tasks. We prove that pi-blocking under the SMLP is lower than existing coarse-grain GPU locking protocols.

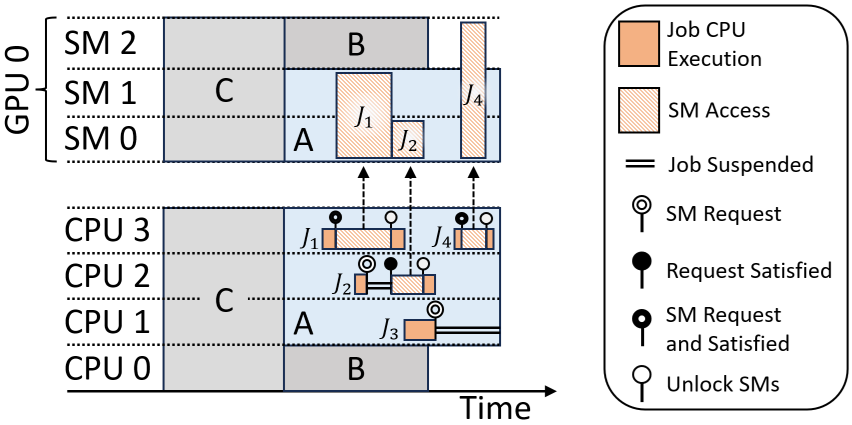

Figure 1 An example schedule with three components, A, B, and C. Components A and B run concurrently and are allocated distinct SMs within the same GPU. Shown within component A are jobs that utilize the allocated SMs. Efficiency is improved by allowing J_4 access to unallocated SMs.

Related Work. Brandenburg and Anderson [8] presented *suspension-oblivious* (s-oblivious) pi-blocking analysis where waiting realized by suspending is viewed as processor execution time in JLFP schedulability analysis. They also proved a per-lock-request pi-blocking lower bound of $\Omega(m)$, where m is the processor count. Therefore, $O(m)$ per request pi-blocking is asymptotically optimal. Minimizing pi-blocking durations is desirable as inflating execution times to account for pi-blocking may result in an unschedulable system.

Component-based systems are not new [2, 4, 5, 6], but due to space constraints, we limit our focus to the most relevant prior work. In avionics, ARINC 653 [17] details real-time operating system design specifications for enabling hardware sharing in component-based systems. It requires components to execute in *time slices*, uninterrupted durations during which isolated access to hardware resources is guaranteed. Amert *et al.* [1] proposed a component-based framework for multicore+accelerator platforms called TimeWall (Time-Isolated Multicore Execution With AcceLerator Locking), which is directed at time-sliced components as in ARINC 653. TimeWall does not allow concurrent cross-component GPU sharing and instead allocates entire GPUs to components. Voronov *et al.* [19] presented schedulability analysis for processing graphs (as often found in AI and ML workloads) within component-based frameworks like TimeWall. They also derived asymptotically optimal intra-component pi-blocking bounds for coarse-grain accelerator locking.

Bakita and Anderson [3] created `libsmctrl`, a library that allows GPU work to be launched on a subset of SMs in NVIDIA GPUs. Due to the highly parallel nature of GPU work, GPU execution time is a function of the number of SMs that service that work. However, the number of SMs that can be effectively used hinges on the extent of parallelism in said work, as shown in Sec. 6. As such, any greedy strategy that utilizes the maximum number of SMs available will not necessarily yield reduced GPU execution costs.

Contributions. We consider the problem of sharing GPUs across isolated components.

First, we extend TimeWall [1] to enable concurrent cross-component GPU sharing through SM reservations. Second, we present the *Streaming Multiprocessor Locking Protocol* (SMLP), a locking protocol that dynamically resizes GPU requests to enable concurrent GPU utilization. Third, we derive a pi-blocking bound for jobs within a component. Finally, we discuss the results of experiments conducted to assess the pi-blocking reductions enabled by the SMLP. Our experiments show that the SMLP improves worst-case pi-blocking by up to 50%.

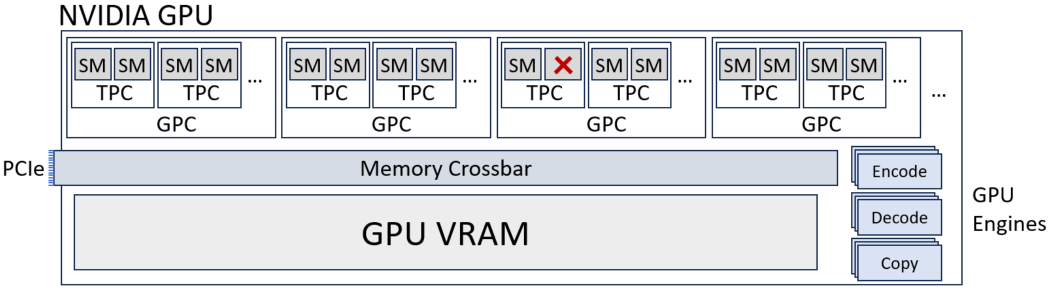

Figure 2 A simplified diagram of an NVIDIA GPU. Hardware manufacturing errors may result in disabled SMs in a TPC, shown by the crossed out SM.

Organization. In Sec. 2, we provide background information necessary to analyze the SMLP. In Sec. 3, we present rules describing the SMLP. In Sec. 4, we derive analytical pi-blocking bounds using the previously established rules. In Sec. 5, we detail SM allocation strategies that reduce execution time on the GPU when allocated relatively few SMs. In Sec. 6, we discuss the results of experiments conducted to evaluate pi-blocking under the SMLP.

2 System Model and Background

In this section, we present relevant background work and definitions needed to analyze the problem of locking SMs in a component-based system.

GPU Architecture. Fig. 2 depicts the high-level architectural layout of an NVIDIA GPU. General Processing Clusters (GPCs) contain Thread Processing Clusters (TPCs). Each TPC contains the SMs that execute a GPU *kernel*, code that runs on the GPU. An SM is a highly parallel processor composed of many concurrent threads of execution. The GPU's VRAM is the last level cache of memory shared by all GPCs. This memory is connected to the GPCs through the memory crossbar. The memory crossbar also connects the GPU to the rest of the system (*e.g.*, CPUs, storage, *etc*). Also shown in the figure are the encode, decode, and copy engines respectively used for video encoding, decoding, and asynchronous memory copying. Some NVIDIA GPUs do not group SMs into TPCs. Such architectures can be abstracted away by assuming that a TPC contains one SM.

SM Partitions. We partition an NVIDIA GPU along SM boundaries, where kernels may be pinned to TPCs as enabled by `libsmctrl` [3]. There may be hardware manufacturing errors where not all TPCs contain the same number of SMs, as shown in Fig. 2 where one SM is disabled. Therefore, we reason that GPU execution time is contingent upon the number of SMs that service a launched kernel rather than TPCs of varying compute capacity. As a simplifying assumption, we assume each TPC contains the same number of SMs. For example, most NVIDIA GPUs released in the last decade contain two SMs per TPC. The analysis presented herein allows for two TPCs with one SM each to act as one TPC, but we do not track such bookkeeping. Additionally, we show how unused, non-uniform TPCs can improve safety margins by reducing average execution time in Sec. 5. Currently, `libsmctrl` does not allow TPC allocations to be changed for a launched kernel. In our pi-blocking analysis, we assume the same non-preemptable limitation.

Partitioning a GPU along SM boundaries does not preclude the use of established analysis techniques for kernel execution times. When kernels utilize the encode, decode, or copy engines, Elliott [10] showed how a kernel's execution time can be analytically inflated by giving each concurrently running kernel round-robin use of the engine for one chunk of memory at a time. In this work we assume such GPU overheads, like data copying and decoding, takes zero time as a simplification. Also, VRAM access times can be made predictable through using spatial partitioning techniques and associated analysis [3, 10]. SM partitioning is orthogonal to the usage of these various other techniques, so we do not consider them further.

Component Model. The TimeWall [1] framework defines a component Γ as a tuple (Θ, Π, Υ). Component Γ requires access to the hardware resources specified in Υ. We require Υ to specify $\mathcal{M}$ and $\mathcal{H}$ such that Γ is guaranteed exclusive access to $\mathcal{M}$ specific CPUs[1] and $\mathcal{H}$ specific SMs. The parameters Θ and Π define a *periodic component reservation* (PCR) [1] where Γ is scheduled for an uninterrupted *time slice* duration of Θ, and each time slice begins every Π time units. We define Φ as the start time of Γ's most recent time slice. When Γ is not scheduled, the resources in Υ may be allocated to other components that also require exclusive, isolated access.

In-Partition Task Model. Within a component is its *in-partition scheduler*, which schedules work that utilizes the resources in Υ. We characterize this work as a system of n sporadic *tasks* $\tau_1, \tau_2, ..., \tau_n$ scheduled on $\mathcal{M}$ identical processors. Each task τ_i releases *jobs* $J_{i,1}, J_{i,2}, ...$ where an arbitrary job of τ_i is given by J_i. A task τ_i has parameters (C_i, T_i, D_i), which respectively denote its worst-case CPU execution cost, the minimum separation between its successive job releases, and the relative deadline by which each of its jobs must complete after release. We say that a job is *pending* from its release until it finishes execution. A pending job may be in one of two states: a *ready* job is available for execution, while a *suspended* job cannot be scheduled. Henceforth, all mentioned jobs are assumed to be in the same component Γ unless stated otherwise. Also, we assume time is discrete.

A job's *priority* determines whether it is scheduled in favor of other jobs in its component. We focus our analysis to *job-level fixed-priority* (JLFP) in-partition scheduling where a job's priority is fixed upon release. We say that job J_i *blocks* on J_k at time t when J_k holds mutually exclusive access to resources required by J_i. A locking protocol can modify a job's initial priority to achieve bounded blocking times. Under s-oblivious analysis [8], the *pi-blocking* of an arbitrary job J_i is the duration when J_i makes no progress towards completion, and the delay cannot be attributed to higher-priority work.

▶ **Definition 1.** *Under s-oblivious schedulability analysis, a job J_i incurs s-oblivious pi-blocking at time t if J_i is pending but not scheduled and fewer than $\mathcal{M}$ higher-priority jobs are pending.*

Under the SMLP, pi-blocking occurs when J_i has sufficient priority to be scheduled on its CPU but is prevented from launching GPU kernels. The pi-blocking duration is then analytically viewed as CPU execution time. By adding the worst-case GPU execution time and pi-blocking duration to a job's worst-case execution cost on the CPU, the analysis presented herein is compatible with years of established real-time schedulability analysis. As the overarching goal is to ensure that all jobs meet their deadline, any inflation to execution costs should be bounded.

[1] We use the term "CPU" somewhat interchangeably to mean a specific CPU core and the entire set of such cores. The intent should be clear from context.

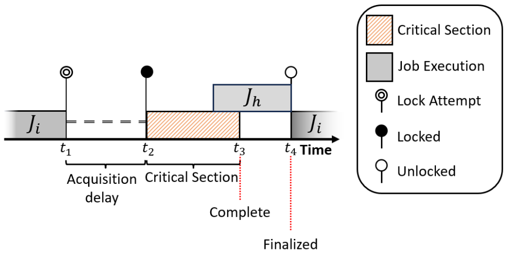

Figure 3 Request timeline.

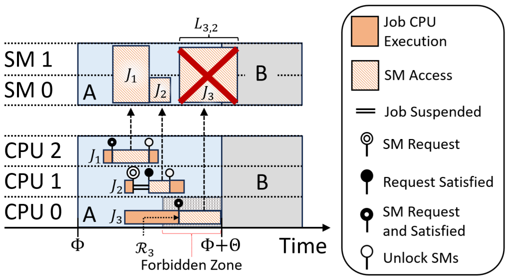

Figure 4 Forbidden-zone blocking where $\mathcal{R}_3$ would violate B's isolation guarantees if satisfied.

Resource Model. Let $\mathcal{L}$ be the set of $\mathcal{H}$ identical SMs allocated only to Γ during its time slice. An arbitrary job, J_i, may issue a request, $\mathcal{R}_i$, to launch a GPU kernel pinned to some subset of SMs in $\mathcal{L}$. The locking protocol may delay J_i's request to initiate GPU work, where this *acquisition delay* may contribute to pi-blocking. In the SMLP, jobs realize waiting by *suspending* and relinquishing any held CPU. The locking protocol *satisfies* $\mathcal{R}_i$ with exclusive access to a set of SMs upon which J_i may launch its GPU kernel. We only consider *synchronous kernels* that run uninterrupted on the GPU while the initiating CPU job suspends and is ineligible to be scheduled on the CPU until the kernel is complete.

Fig. 3 illustrates the timeline of a request $\mathcal{R}_i$. The *critical-section* duration is the GPU kernel's execution duration. The request $\mathcal{R}_i$ is *complete* when the GPU kernel completes. When complete, J_i may lack sufficient priority to be scheduled, thus being unable to execute an unlock procedure on the CPU. We consider $\mathcal{R}_i$ *finalized* the first moment when $\mathcal{R}_i$ is complete and J_i is scheduled. In the analysis presented in Sec. 4, we consider the critical-section duration as CPU time (in keeping with s-oblivious analysis). We define $L_{\max}$ as the largest critical-section duration of any GPU kernel launched by jobs in Γ.

Forbidden Zones. Shown in Fig. 4 is an example where a launched kernel's execution time exceeds the remaining time in its component's time slice. In this example, $\mathcal{R}_3$ in component A is satisfied and launches a non-preemptible kernel on SM0 and SM1 that executes when component B is scheduled. If jobs in component B access these SMs, the interference caused by non-exclusive SM access may result in missed deadlines.

To prevent this scenario, a job is *forbidden* to launch non-preemptible work that would violate isolation guarantees. This *forbidden-zone blocking* (fz-blocking) is not new and has been explored in many earlier works [1, 4, 5, 12, 19]. Component-based frameworks such as TimeWall [1] utilize fz-blocking to guarantee isolated access to non-preemptible hardware accelerators such as GPUs.

Priority Inheritance. Under global JLFP in-partition scheduling, a job J_i is scheduled on one of $\mathcal{M}$ processors when J_i is ready at that time and its priority is among the $\mathcal{M}$ highest priorities in Γ. If J_i issues a lock request and no SMs are available, then a *progress mechanism* ensures J_i makes progress towards acquiring SMs.

Suppose within Γ that the priority of J_i is within the top $\mathcal{M}$ priorities at time t, and J_i issues a request $\mathcal{R}_i$ for SMs when all SMs in $\mathcal{L}$ are allocated to other satisfied requests. Further, suppose that a lower-priority job, J_k, does not have a priority in the top $\mathcal{M}$ priorities at time t and is not scheduled but has a completed request that has not been finalized. The SMLP utilizes the *priority inheritance* [18] progress mechanism to allow J_k to inherit J_i's priority so that J_k can execute its CPU-side unlock code.

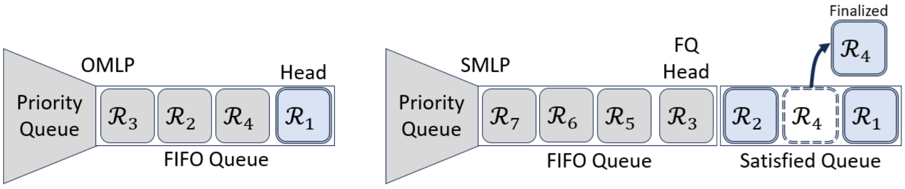
OMLP
Priority Queue
$\mathcal{R}_3$ $\mathcal{R}_2$ $\mathcal{R}_4$ $\mathcal{R}_1$
Head
FIFO Queue
SMLP
Priority Queue
$\mathcal{R}_7$ $\mathcal{R}_6$ $\mathcal{R}_5$ $\mathcal{R}_3$
FQ Head
FIFO Queue
$\mathcal{R}_2$ $\mathcal{R}_4$ $\mathcal{R}_1$
Satisfied Queue
Finalized
$\mathcal{R}_4$

Figure 5 Left: the global OMLP queues where a priority queue feeds into an $\mathcal{M}$-length FIFO queue. Right: the SMLP queues where the SQ (Satisfied Queue) is not a FIFO queue.

This work builds upon the global OMLP (the $O(\mathcal{M})$ locking protocol) [8] as requests arrive at a priority queue feeding into an $\mathcal{M}$-length FIFO queue. The OMLP has an optimal pi-blocking bound of $O(\mathcal{M})$ [8] when used in a component-based scheduler [19]. A request is only satisfied when at the head of the FIFO queue. The SMLP builds on the OMLP queues and allows multiple requests to be concurrently satisfied. Specifically, the SMLP satisfies requests earlier than a k-exclusion lock by launching GPU kernels on any available number of SMs rather than requiring requests to specify and wait for a specific number of SMs to become available. Fig. 5 provides a comparison of the OMLP and SMLP queues.

3 The SMLP

We now introduce the Streaming Multiprocessor Locking Protocol (SMLP). We first provide additional definitions and properties used to compose the rules of the SMLP. Then, we describe how the SMLP functions in a global JLFP system without time slicing. Lastly, we define rules that enable component-level isolation.

A job J_i in Γ may issue a lock request $\mathcal{R}_i$ to launch a GPU kernel. Much like the OMLP, $\mathcal{R}_i$ first arrives in the priority queue (PQ) that feeds into the $\mathcal{M}$-length FIFO queue (FQ). In the SMLP, the FQ is followed by another arrival-ordered queue called the Satisfied Queue (SQ). When $\mathcal{R}_i$ is satisfied with an SM allocation, J_i may launch its GPU kernel pinned to the allocated SMs and $\mathcal{R}_i$ moves to the SQ until finalized. When finalized, $\mathcal{R}_i$ is removed from the SQ and the held SMs are unlocked.

Satisfied Queue. The SQ is not a FIFO queue. Any request arriving in the SQ is ordered by its arrival time, where some consistent tie-breaking mechanism may be employed to ensure that no two requests occupy the same position. However, satisfied requests in the SQ may complete, finalize, and leave earlier than preceding requests. Thus, the SQ is first-in ordered, but is not first-out.

Analysis Assumptions. The lock and unlock procedures for the SMLP are assumed to take ε time, where strictly for analysis purposes, we assume that $\varepsilon \to 0$. We later show how the unlock code is subject to priority inheritance, such that, at any time after a job with a completed request incurs pi-blocking on another job, then the request is finalized in ε time.

▶ **Definition 2.** *Let each Γ define a positive integer h that divides $\mathcal{H}$. Each request in Γ will be satisfied with some multiple of h SMs.*

We show in Sec. 5 that h can be configured to reduce the critical-section duration of requests that exhibit lengthy execution times under small SM allocations.

▶ **Definition 3.** *Each request $\mathcal{R}_i$ defines an ordered set $L_i = \{L_{i,h}, L_{i,2h}, ..., L_{i,\mathcal{H}}\}$. Each $L_{i,k}$ represents the worst-case critical-section duration of $\mathcal{R}_i$ when $\mathcal{R}_i$ is satisfied with k SMs.*

When more SMs are allocated to a request, the critical-section duration is shorter. However, this trend ceases when an SM allocation exceeds the parallelism requirements of $\mathcal{R}_i$.

▶ **Example 4.** Consider a GPU kernel that performs vector addition on two vectors of length 2,048. Each SM contains 1,024 threads, and for the purpose of this example, each SM can perform 1,024 additions in parallel per cycle. When the kernel is allocated one SM, two cycles are required to perform the 2,048 additions. When the kernel is allocated two SMs, only one cycle is required to perform the additions as each SM performs addition on half of the vectors. However, when the kernel is allocated more than two SMs, the kernel still requires one cycle to complete, wasting the additional parallelism provided by the extra SMs. Thus, excessive SM allocations result in GPU capacity loss where those SMs could have been allocated to other requests.

To prevent allocating more SMs than necessary, we must impose some restrictions on the SM allocations used to satisfy requests. We first define a set that contains the SMs in Γ not allocated to any request at time t. Then, we apply an upper bound on how many of those SMs can be allocated to $\mathcal{R}_i$.

▶ **Definition 5.** *Let $\mathcal{I}(t)$ be the set of SMs not allocated to any job at time t, where $\mathcal{I}(t) \subseteq \mathcal{L}$.*

▶ **Definition 6.** *Let $z_i(t)$ be an upper bound on the number of allocable SMs to an unsatisfied request $\mathcal{R}_i$ at time t where $z_i(t) \leq |\mathcal{I}(t)|$.*

The goal of $z_i(t)$ is to prevent large SM allocations where additional SMs make no meaningful improvement in critical-section duration. One method of deriving $z_i(t)$ is to find the minimum number of SMs, j, where $L_{i,j} \approx L_{i,|\mathcal{I}(t)|}$. We leave such optimizations of $z_i(t)$ to the system designer to ensure that $\mathcal{R}_i$ is allocated at most the minimal number of SMs that appropriately satisfy its parallelism requirements.

3.1 SMLP Resizing Technique

We first define the SMLP with Rules 1–5 when there is no time slicing by assuming Γ has an infinite duration ($\Theta = \infty$). In such a component, according to Rule 1 below, when a request is *satisfiable*, it is provided with an SM allocation and moved to the SQ. Later, when we introduce time slicing in Sec. 3.2, we will see that a satisfiable request may not necessarily be immediately satisfied.

▶ **Rule 1.** A newly issued request $\mathcal{R}_i$ at time t is *immediately* satisfiable if $|\mathcal{I}(t)| \geq h$. If SMs are unavailable ($|\mathcal{I}(t)| < h$), then $\mathcal{R}_i$ is appended to the FQ if fewer than $\mathcal{M}$ total requests exist in the FQ. Otherwise, $\mathcal{R}_i$ is added to the PQ.

This rule implies that a newly arriving request can be immediately allocated SMs and moved to the SQ if h or more SMs are available. This can allow for more than $\mathcal{M}$ concurrent requests to be satisfied in the SQ if $\mathcal{H} > \mathcal{M}$.

The next rule describes the resizing technique of the SMLP to allow GPU kernels to execute on any number of SMs, as enabled by `libsmctrl` [3], while also avoiding excessive allocations that do not match the parallelism requirements of that request's kernel.

▶ **Rule 2.** When $\mathcal{R}_i$ enters the SQ at time t, it is satisfied with $z_i(t)$ SMs.

To reduce the GPU capacity loss of Ex. 4, the SMLP may satisfy $\mathcal{R}_i$ with fewer than $|\mathcal{I}(t)|$ SMs where $z_i(t)$ ensures the capacity allocated does not exceed $\mathcal{R}_i$'s parallelism needs.

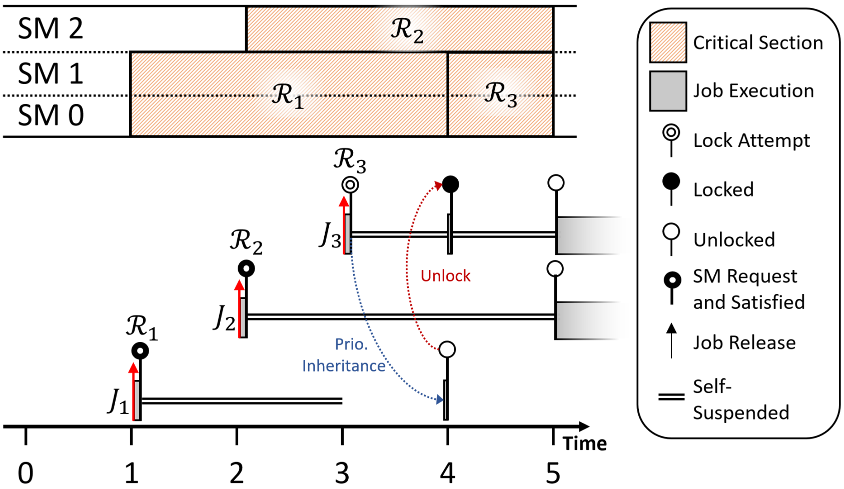

Figure 6 Timeline of events in Ex. 8 that illustrate both the necessity of priority inheritance and the operation of Rules 1–4.

Priority Inheritance. In the SMLP, we exploit the following property of non-preemptible GPU execution to ensure the progress of jobs with satisfied requests.

▶ **Property 7.** A satisfied, incomplete request $\mathcal{R}_i$ continues to execute on the GPU at any time t until completion, even if J_i's CPU priority is not among the top $\mathcal{M}$. As GPU time is analytically viewed as CPU time, J_i continues to make progress even when not scheduled on the CPU.

In conjunction with Prop. 7, we show how priority inheritance can be applied to ensure that jobs with completed kernels promptly execute their CPU-side unlock code.

▶ **Rule 3.** A job J_i with a request $\mathcal{R}_i$ is suspended on the CPU if $\mathcal{R}_i$ is in the PQ or FQ. If completed requests exist in the SQ, then let $\mathcal{R}_k$ denote the one with the earliest completion time, where ties are broken by SQ position. If J_k is not in the top $\mathcal{M}$ CPU priorities, then J_k inherits the CPU priority of the highest-priority suspended job in the SQ, FQ, or PQ.

A key property of this rule is that priority inheritance is only applied to one job whose request is in the SQ. This is done to expedite the execution of its CPU-side unlock code after its GPU kernel is complete. If the inherited priority is not in the top $\mathcal{M}$ priorities of Γ at that time, then none of the jobs with requests in any queue are experiencing pi-blocking.

Prop. 7 is sufficient to ensure that any satisfied request $\mathcal{R}_i$ makes progress towards completion on the GPU whether or not its job J_i is scheduled on the CPU. With such a progress property, why is priority inheritance needed in Rule 3? While Prop. 7 ensures that $\mathcal{R}_i$ makes progress, requests waiting on $\mathcal{R}_i$ can still be delayed after $\mathcal{R}_i$ completes without priority inheritance.

▶ **Example 8.** Consider Fig. 6 where for component Γ, $h = 1$, $\mathcal{M} = 2$, and $\mathcal{H} = 3$. At $t = 0$, $|\mathcal{I}(t)| = \mathcal{H}$, and there are no requests in Γ. At $t = 1$, job J_1 is released, and immediately issues $\mathcal{R}_1$, where $L_1 = \{5, 3, 3\}$. By Rule 1, $\mathcal{R}_1$ is appended to the SQ, and by Rule 2, it is satisfied with two SMs as $z_1(1) = 2$, which means that $\mathcal{R}_1$ cannot benefit from any additional parallelism. At $t = 2$, job J_2 is released with a higher priority than J_1, and immediately issues $\mathcal{R}_2$, where $L_2 = \{3, 2, 1\}$. By Rule 1, $\mathcal{R}_2$ is also appended to the SQ, and by Rule 2, it is satisfied with the only remaining SM. At $t = 3$, a job J_3 is released with a higher priority than J_1 and J_2, and immediately issues $\mathcal{R}_3$ where $L_3 = \{3, 1, 1\}$. By Rule 1, since $|\mathcal{I}(3)| = 0$ and the number of requests in the FQ is less than $\mathcal{M}$, $\mathcal{R}_3$ is appended to the FQ. At $t = 4$, $\mathcal{R}_1$ completes. If no progress mechanism is used, then J_1 does not have sufficient priority to be scheduled on the CPU to unlock the available SMs at time $t = 4$. In this case, SMs are idle and J_3 is among the $\mathcal{M}$ highest priority jobs, but $\mathcal{R}_3$ is not satisfied. Since $\mathcal{R}_3$ can be satisfied using the two available SMs at $t = 4$, J_3 thus experiences unnecessary pi-blocking. With Rule 3, J_1 inherits the priority of J_3 and has sufficient priority to be scheduled on the CPU to unlock the SMs needed by J_3, allowing $\mathcal{R}_3$ to be satisfied.

Priority inheritance in Rule 3 ensures that the job belonging to the first completed request in the SQ inherits the highest priority of all jobs in the SMLP queues, thus avoiding unnecessary pi-blocking. Should two requests in the SQ complete simultaneously, they can be consistently tie-broken by their position in the SQ. As such, the SMLP relies on the fact that the CPU-side unlock code inevitably has to run sequentially, one job at a time.

When a job J_i relinquishes its lock after its request $\mathcal{R}_i$ completes, we say that $\mathcal{R}_i$ is *finalized*. From Rule 3, we have the following property.

▶ **Property 9.** A request $\mathcal{R}_i$ is finalized and leaves the SQ at the first time instant after $\mathcal{R}_i$ is complete and J_i's base or inherited priority is among the $\mathcal{M}$ highest in Γ.

The next two rules describe how the SMLP queues function to ensure requests are satisfied without undue pi-blocking.

▶ **Rule 4.** At any time t when $|\mathcal{I}(t)| \geq h$, the head request in the FQ (if any) is satisfiable.

▶ **Rule 5.** At any time t, if there exist fewer than $\mathcal{M}$ requests in the FQ, then the head request in the PQ (if any) is enqueued in the FQ.

Note that Rules 4 and 5 are applied each time a request is finalized as SMs become available and the number of requests in the SQ changes. The operation of the above rules are demonstrated by the following example. This example is a continuation of Ex. 8.

▶ **Example 10.** Consider the same job arrivals and request parameters as Ex. 8. At $t = 4$, $\mathcal{R}_1$ completes, inherits a higher priority from Rule 3 and is finalized. By Rule 4 because $|\mathcal{I}(4)| = 2$, $\mathcal{R}_3$ is also satisfied and moved to the SQ. By Rule 2, $\mathcal{R}_3$ becomes satisfied with two SMs, thus removing the unnecessary pi-blocking.

From Ex. 10, we see that after a request $\mathcal{R}_i$ is finalized at time t, $|\mathcal{I}(t)|$ is at least h. Thus, Rules 4 and 5, in conjunction with Rule 2, guarantee that the request at the head of the FQ is satisfied when $|\mathcal{I}(t)| \geq h$ and Γ is not time sliced. As requests are resized by Rule 2 to "fit" in the available SMs at time t, we eliminate starvation by always satisfying requests when SMs are available.

3.2 SMLP with Time Slicing

Rules 1–5 describe the operation of the SMLP under a global JLFP in-partition scheduler. For the locking protocol to additionally maintain component-level isolation, a GPU kernel in a component must not execute past its component's time-slice boundaries. To provide such guarantees, we extend the concept of forbidden zones from TimeWall [1].

▶ **Definition 11.** *A request $\mathcal{R}_i$ in the FQ or PQ is* ***fz-blocked*** *at time t and cannot be satisfied when $t + L_{i,z_i(t)} > \Phi + \Theta$, where $z_i(t) \leq |\mathcal{I}(t)|$ from Def 6.*

In Fig. 4, a request in its forbidden zone is satisfied and violates component-level isolation guarantees. Def. 11 describes when such a request should be fz-blocked to prevent a non-preemptable GPU kernel from executing past the component's scheduled time slice. The next rule details how requests are satisfied and enter the SQ with respect to fz-blocking.

▶ **Rule 6.** A satisfiable request from Rule 1 or 4 is satisfied and moved to the SQ when not fz-blocked. Additionally, Rule 4 is applied when $t = \Phi$ to ensure queued requests are satisfied when Γ's time slice begins.

The next rule enables a skip-ahead mechanism where later-arriving or lower-priority requests may be satisfied earlier to reduce worst-case pi-blocking time.

▶ **Rule 7.** In Γ, if $\mathcal{R}_i$ is satisfiable at time t by Rules 1 or 4, and is fz-blocked, then the first request $\mathcal{R}_k$ in the FQ (or first by job priority in the PQ if no such request exists in the FQ) that is not fz-blocked is satisfied and moved to the SQ instead.

We now prove that the SMLP does not assign any two concurrent jobs the same SM.

▶ **Definition 12.** *Let $\mathcal{K}_i$ represent set of SMs allocated to $\mathcal{R}_i$ when satisfied, where $\mathcal{K}_i \subseteq \mathcal{L}$.*

▶ **Theorem 13.** *For any two concurrently satisfied, incomplete requests $\mathcal{R}_i$ and $\mathcal{R}_k$ where $\mathcal{R}_i \neq \mathcal{R}_k$, $\mathcal{K}_i \cap \mathcal{K}_k = \emptyset$ holds.*

Proof. Let t_i, t_k denote the time when $\mathcal{R}_i$ and $\mathcal{R}_k$ are satisfied respectively. Without loss of generality, assume $t_i < t_k$ where any ties are broken by position in the SQ. By Def. 5, $\mathcal{I}(t_k)$ cannot contain any SMs in $\mathcal{K}_i$ as $\mathcal{R}_i$ was allocated SMs earlier at time t_i and both requests are incomplete at time t_k. By Rule 2, $\mathcal{K}_k$ is assigned SMs in $\mathcal{I}(t_k)$, which, as shown, does not contain any SMs in $\mathcal{K}_i$. Thus, $\mathcal{K}_i \cap \mathcal{K}_k = \emptyset$. ◀

This theorem proves the necessary safety property of the SMLP where no two jobs are allocated the same SM. In the next section, we prove an upper bound on the worst-case pi-blocking duration of the SMLP.

4 Suspension-Oblivious Pi-Blocking

In this section, we analytically derive an upper bound on the worst-case pi-blocking duration for any request in the SMLP. Prior work by Nemitz *et al.* [14] determined that computing a tight s-oblivious pi-blocking bound for the locking of multiple identical resource replicas is NP-hard. However, in their analysis, the number of replicas that satisfy a request cannot be determined without analyzing the underlying task set. By allowing the dynamic resizing of GPU requests to fit on available SMs with Rule 2, we can compute a reasonable upper bound on s-oblivious pi-blocking in polynomial time.

In Sec. 4.1, we prove a pi-blocking bound when Γ is not time sliced ($\Theta = \infty$). We first prove a bound on pi-blocking time incurred by a request in the FQ, then a bound on pi-blocking time incurred by a request in the PQ. In Sec. 4.2, we extend the analysis with Rules 6 and 7 when Γ is time sliced ($\Theta \neq \infty$).

4.1 SMLP Blocking Without Component Time Slicing

The results provided in this section pertain to a component Γ that is not time-sliced. In such components, satisfiable requests are immediately satisfied.

▶ **Lemma 14.** *Consider a request $\mathcal{R}_i$ that is issued at time t_0 and is satisfied at time t_1. If J_i is pi-blocked at time $t \in [t_0, t_1)$, then $|\mathcal{I}(t)| = 0$.*

Proof. If J_i is pi-blocked, then J_i's priority is in the top $\mathcal{M}$ priorities in Γ. By Rule 3, any completed requests can be finalized by inheriting J_i's priority. Thus, all available SMs have been unlocked by the CPU-side unlock procedure for any completed requests. If $|\mathcal{I}(t)| > 0$, then by Def. 2, $|\mathcal{I}(t)| \geq h$ as requests are only satisfied with a multiple of h SMs. If $|\mathcal{I}(t)| \geq h$, then by Rule 4, some request is satisfied, reducing available SMs until Rule 4 cannot be applied. If $\mathcal{R}_i$ is satisfied, then J_i is not pi-blocked. If no request is satisfied, then by Rule 4, it must be that $|\mathcal{I}(t)| = 0$. ◀

Lem. 14 ensures that whenever an unsatisfied request $\mathcal{R}_i$ incurs pi-blocking on J_i, all $\mathcal{H}$ SMs assigned to Γ are fully utilized. We now quantify the total SM utilization time of individual requests with the next two definitions.

▶ **Definition 15.** *Let $\mathcal{A}_{i,k}$ denote the* work *of $\mathcal{R}_i$ when allocated k SMs such that $\mathcal{A}_{i,k} = k \cdot L_{i,k}$*

▶ **Definition 16.** *Let $\mathcal{A}_i^{\max}$ denote the largest $\mathcal{A}_{i,k}$, where $h \leq k \leq \mathcal{H}$ and k SMs can be allocated by the function z_i.*

Intuitively, the work $\mathcal{A}_{i,k}$ of a request $\mathcal{R}_i$ captures the worst-case amount of computation required to complete $\mathcal{R}_i$ using k SMs. First, we determine an upper bound on the amount of work done on the GPU when J_i is pi-blocked. This upper bound on total work is then used to compute an upper bound on the total pi-blocking duration incurred by $\mathcal{R}_i$.

From Prop. 7, when $|\mathcal{I}(t)| = 0$, each SM completes one unit of work on requests in the SQ per unit of time. Thus, at each unit of time where $|\mathcal{I}(t)| = 0$, a total of $\mathcal{H}$ units of work are completed on requests in the SQ. Therefore, we have the following corollary of Lem. 14.

▶ **Corollary 17.** *For each unit of time that J_i is pi-blocked, the GPU completes $\mathcal{H}$ units of work on requests in the SQ.*

FQ Pi-Blocking. We first analyze the pi-blocking incurred by a request $\mathcal{R}_i$ in the FQ. When $\mathcal{R}_i$ is in the $\mathcal{M}$-length FQ, then there are at most $\mathcal{M} - 1$ other requests in the FQ. We require the following lemma to ensure this property.

▶ **Lemma 18.** *At any given time, at most $\mathcal{M}$ requests are enqueued in the FQ.*

Proof. By Rule 1, an issued request enters the FQ when less than $\mathcal{M}$ requests are in the FQ. By Rule 5, a request in the PQ enters the FQ under the same assumptions. ◀

We now provide some definitions necessary to the analysis of the SMLP.

▶ **Definition 19.** *Let $prec(\mathcal{R}_i)$ be the set consisting of all requests in the SQ and all requests that precede $\mathcal{R}_i$ in the FQ when $\mathcal{R}_i$ first enters the FQ.*

▶ **Definition 20.** *Let $top(k)$ be any set of k requests with the largest work values in Γ.*

▶ **Lemma 21.** *The total work incurred by $\mathcal{M} - 1$ requests is at most*

$$\sum_{\mathcal{R}_k \in top(\mathcal{M}-1)} \mathcal{A}_k^{\max}. \tag{1}$$

Proof. The total work of requests in a set of $\mathcal{M} - 1$ requests is maximized when those requests are the largest work-inducing requests, *i.e.*, the requests are in *top*$(\mathcal{M} - 1)$. ◀

From Lem. 21, we obtain the following corollary.

▶ **Corollary 22.** *If $|prec(\mathcal{R}_i)| \le \mathcal{M} - 1$, then the total work of requests in $prec(\mathcal{R}_i)$ is at most*

$$\sum_{\mathcal{R}_k \in prec(\mathcal{R}_i)} \mathcal{A}_k^{\max} \le \sum_{\mathcal{R}_k \in top(\mathcal{M}-1)} \mathcal{A}_k^{\max}.$$

We now upper-bound the pi-blocking duration incurred by a request $\mathcal{R}_i$ while in the FQ.

▶ **Definition 23.** *Let B_{FQ} be an upper bound on pi-blocking incurred by a request in the FQ.*

▶ **Lemma 24.** *A suitable value for B_{FQ} is*

$$B_{FQ} = L_{\max} + \sum_{\mathcal{R}_k \in top(\mathcal{M}-1)} \frac{\mathcal{A}_k^{\max}}{\mathcal{H}}.$$

Proof. Let $\mathcal{R}_i$ be a request in the FQ. By Rule 3, all completed requests are finalized, otherwise J_i is not pi-blocked. The SMLP allows for more than $\mathcal{M}$ concurrently satisfied requests. By Prop. 7, all satisfied requests in the SQ complete within $L_{\max}$ units of time. By Lem. 18, J_i is pi-blocked by at most $\mathcal{M} - 1$ preceding requests in the FQ. From Lem. 21 and Cor. 22, the total work incurred by the $\mathcal{M} - 1$ requests is upper bounded in (1). From Cor. 17, for each unit of time that a job J_i is pi-blocked, $\mathcal{H}$ units of work are completed. Thus, $\mathcal{R}_i$ in the FQ incurs pi-blocking of at most B_{FQ} units of time. ◀

PQ Pi-Blocking. We now analyze the pi-blocking incurred by a request in the PQ. First, we determine an upper bound on the number of requests that leave the PQ when J_i is pi-blocked.

▶ **Lemma 25.** *Let t_0 be the time $\mathcal{R}_i$ is enqueued at the PQ, and t_1 be the time $\mathcal{R}_i$ is enqueued in the FQ. For each time $t \in [t_0, t_1)$, let $entered(t)$ denote the number of requests that leave the PQ and enter the FQ in the interval $[t_0, t]$. If J_i is pi-blocked at time t, then $entered(t) < \mathcal{M}$.*

Proof. By contradiction. Suppose J_i is pi-blocked at time t (with a request $\mathcal{R}_i$ in the PQ) and $entered(t) \ge \mathcal{M}$. By Rule 5, because $\mathcal{R}_i$ has not entered the FQ, there must be $\mathcal{M}$ requests in the FQ. Because $entered(t) \ge \mathcal{M}$, at least $\mathcal{M}$ requests from higher-priority jobs are enqueued in the FQ in the interval $[t_0, t]$. By Def. 1, the presence of $\mathcal{M}$ higher-priority jobs with requests in the FQ imply that J_i is not pi-blocked. ◀

▶ **Lemma 26.** *For each request that is finalized when the PQ is non-empty, one request moves from the PQ to the FQ.*

Proof. By Rule 4, if SMs are available, then the head request in the FQ (if any) is satisfied and moves to the SQ. By Rule 6, if a request leaves the FQ, then the head request in the non-empty PQ moves to the FQ. ◀

▶ **Lemma 27.** *At time t, if no complete request is finalized, then no request incurs pi-blocking.*

Proof. By Rule 3, if the inherited priority, which is the highest priority of all jobs with an issued request, is not in the top $\mathcal{M}$ CPU priorities, then all jobs with issued requests are not in the top $\mathcal{M}$ priorities, and thus, are not pi-blocked at this time. ◀

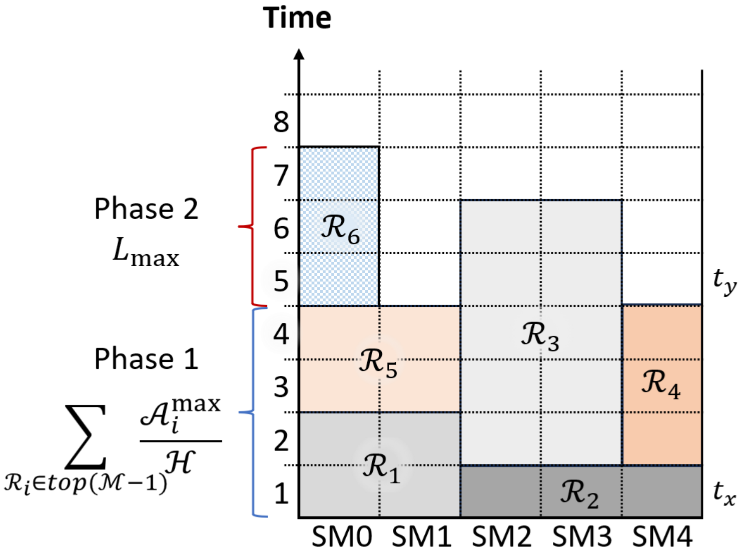

Figure 7 An example timeline for requests where $\mathcal{H} = 5$, $\mathcal{M} = 6$, and $h = 1$. The k^{th} row shows the allocation of SMs to satisfied requests at the k^{th} unit of time. In rows 1–4, requests are assigned all $\mathcal{H}$ SMs. In rows 5–7, requests are only assigned some of the SMs guaranteed to Γ.

By Lem. 25 and 26, the pi-blocking duration incurred while $\mathcal{R}_i$ is in the PQ is at most the time to finalize $\mathcal{M}$ requests. By Lem. 27, any time when requests are completed but not finalized does not contribute towards pi-blocking. We now determine the total amount of work incurred by completing $\mathcal{M}$ requests to determine the PQ pi-blocking duration.

In Fig. 7, we visualize the time it takes for $\mathcal{M}$ requests to finalize in two phases where $\mathcal{M} = 6$. In the first phase, all $\mathcal{H}$ SMs are allocated to some subset of the $\mathcal{M}$ requests. In the second phase, fewer than $\mathcal{H}$ SMs are allocated to some subset of the $\mathcal{M}$ requests, as some requests have been finalized. If any requests are already satisfied when a request enters the PQ, then those satisfied requests will contribute to the $\mathcal{M}$ requests that will finalize. Thus, the time to finalize $\mathcal{M}$ requests is maximized when starting with $\mathcal{M}$ newly issued requests with no prior satisfied requests.

▶ **Definition 28.** *Let W be the worst-case sequence of $\mathcal{M}$ satisfied requests where the sequence results in the longest duration for all requests in W to be finalized. Let $\mathcal{R}_y$ denote the last request in W to be finalized. Additionally, let t_x denote the first time instant when each SM is utilized by one of the requests in W, and t_y denote the time that $\mathcal{R}_y$ is satisfied.*

▶ **Lemma 29.** *The latest time that t_y, the time that $\mathcal{R}_y$ is satisfied, can occur is*

$$t_x + \sum_{\mathcal{R}_k \in top(\mathcal{M}-1)} \frac{\mathcal{A}_k^{\max}}{\mathcal{H}}. \tag{2}$$

Proof. By contradiction. Suppose that t_y is greater than (2). By Lem. 14, all $\mathcal{H}$ SMs must have been executing the $\mathcal{M} - 1$ requests in W other than $\mathcal{R}_y$ during $[t_x, t_y)$. The amount of work completed during this interval is given by

$$\begin{aligned}\mathcal{H}\left(t_y - t_x\right) &> \mathcal{H} \sum_{\mathcal{R}_k \in top(\mathcal{M}-1)} \frac{\mathcal{A}_k^{\max}}{\mathcal{H}} \\ &= \sum_{\mathcal{R}_k \in top(\mathcal{M}-1)} \mathcal{A}_k^{\max}.\end{aligned}$$

This contradicts Lem. 21, as more work than that which can be incurred by $\mathcal{M} - 1$ requests must have executed during $[t_x, t_y)$. ◀

▶ **Lemma 30.** *In the sequence of $\mathcal{M}$ satisfied requests, the latest request to finalize, $\mathcal{R}_y$, is finalized at the latest by the time:*

$$t_x + L_{\max} + \sum_{\mathcal{R}_k \in top(\mathcal{M}-1)} \frac{\mathcal{A}_k^{\max}}{\mathcal{H}}. \tag{3}$$

Proof. By Lem. 29, the latest time $\mathcal{R}_y$ will be satisfied is $t_x + \sum_{\mathcal{R}_k \in top(\mathcal{M}-1)} \mathcal{A}_k^{\max}/\mathcal{H}$. By definition, the largest critical-section duration is $L_{\max}$. Thus, the request $\mathcal{R}_y$ will finalize at the latest by $L_{\max}$ time units after being satisfied. ◀

▶ **Definition 31.** *Let B_{PQ} be an upper bound on the pi-blocking duration incurred by $\mathcal{R}_i$ when in the PQ.*

▶ **Lemma 32.** *A suitable value for B_{PQ} is*

$$B_{PQ} = L_{\max} + \sum_{\mathcal{R}_k \in top(\mathcal{M}-1)} \frac{\mathcal{A}_k^{\max}}{\mathcal{H}}.$$

Proof. By Lem. 25 and 26, the total duration $\mathcal{R}_i$ incurs pi-blocking while in the PQ is at most the duration to finalize $\mathcal{M}$ requests. By Lem. 30, we obtain B_{PQ}. ◀

Total Pi-Blocking Without Time Slicing. We now derive a bound for the total amount of time a request incurs pi-blocking in the SMLP when $\Theta = \infty$.

▶ **Definition 33.** *Let X denote the worst-case pi-blocking time incurred by any request when Γ is not time sliced.*

▶ **Theorem 34.** *The pi-blocking time a request incurs under s-oblivious analysis is at most*

$$X = B_{FQ} + B_{PQ}.$$

Proof. Under the SMLP, a request can incur pi-blocking when it is in the PQ and the FQ. According to Lem. 24 and 32, the worst-case pi-blocking incurred by a request in the FQ and PQ is at most B_{FQ} and B_{PQ}, respectively. Thus, $X = B_{FQ} + B_{PQ}$. ◀

4.2 SMLP Blocking With Component Time-Slicing

We now modify the analysis in Sec. 4.1 to account for pi-blocking time incurred from time slicing Γ where $\Theta \neq \infty$. First, we establish the maximum number of forbidden zones encountered when $\mathcal{R}_i$ is incomplete and J_i is suspended.

▶ **Definition 35.** *Let $L_i^{\max}$ denote the largest critical-section duration in L_i.*

▶ **Lemma 36.** *The number of forbidden zones crossed by J_i with an incomplete request $\mathcal{R}_i$ is at most*

$$\left\lceil \frac{X + L_i^{\max}}{\Theta - L_i^{\max}} \right\rceil.$$

Proof. This proof is similar to the one presented by Vornov *et al.* [19]. From Rules 6, 7, and Def. 11, the worst-case forbidden-zone duration for $\mathcal{R}_i$ is $L_i^{\max}$. In each time slice of Γ, $\mathcal{R}_i$ is not fz-blocked for a duration at least $\Theta - L_i^{\max}$. The most time $\mathcal{R}_i$ spends pi-blocked without forbidden zones is given by X. Additionally, $\mathcal{R}_i$ may fz-block immediately when issued. This necessitates inflating the blocking time by an extra forbidden zone. Thus, the total number of boundaries crossed is $\lceil (X + L_i^{\max})/(\Theta - L_i^{\max}) \rceil$. ◀

With an upper bound on the number of forbidden zones that J_i crosses before $\mathcal{R}_i$ is satisfied, we can derive the worst-case total pi-blocking time.

▶ **Theorem 37.** *Within a component, the pi-blocking time incurred by $\mathcal{R}_i$ is at most*

$$X + \left\lceil \frac{X + L_i^{\max}}{\Theta - L_i^{\max}} \right\rceil L_i^{\max}.$$

Proof. By Thm. 34, the pi-blocking incurred by $\mathcal{R}_i$ without time slicing is X. Each time $\mathcal{R}_i$ is fz-blocked, it is blocked for a duration at most $L_i^{\max}$. From Lem. 36, this happens at most $\left\lceil \frac{X+L_i^{\max}}{\Theta-L_i^{\max}} \right\rceil$ times. Thus, the total pi-blocking is at most $X + \left\lceil \frac{X+L_i^{\max}}{\Theta-L_i^{\max}} \right\rceil L_i^{\max}$. ◀

▶ **Theorem 38.** *The total worst-case pi-blocking time incurred by J_i is b_i where*

$$b_i = \sum_{\mathcal{R}_k \in J_i} \left(X + \left\lceil \frac{X + L_k^{\max}}{\Theta - L_k^{\max}} \right\rceil L_k^{\max} \right).$$

Proof. This is a summation of all the pi-blocking incurred by every request issued by J_i and follows from Thm. 37. ◀

5 Blocking Optimization

Recall from Sec. 2 and illustrated in Fig. 2 that there may exist TPC partitions that contain disabled SMs. Because the SMLP only utilizes uniform SM allocations, this may result in "left-over" TPCs. For example, consider eight TPCs each with two SMs, and one TPC has a disabled SM for a total of 15 SMs when $h = 2$. The left-over TPC only contains one SM, and cannot be combined with another TPC to provide a comparable level of compute.

Additionally, consider the SMs that are not reserved by any component at a specific time. Shown in Fig. 1 is one such scenario where component B's scheduled time slice ends, but the next component that utilizes the SMs previously reserved by B is scheduled some time later. The SMLP can be augmented to utilize SMs that are not reserved by a component.

▶ **Definition 39.** *Let ℓ_i denote an arbitrary set of SMs partitioned by* `libsmctrl`.

▶ **Definition 40.** *Let $v_i(t)$ denote the remaining time from t until an unallocated ℓ_i is reserved by a component. If ℓ_i is allocated to a request or reserved by a component, then $v_i(t) = 0$.*

Average-case GPU utilization can be improved by allocating these SMs to newly satisfied requests, provided that the SM usage time is less than $v_i(t)$. Shown in Fig. 8 is an example of such an allocation where $\mathcal{R}_4$ utilizes SMs beyond the $\mathcal{H}$ guaranteed SMs. So long as requests are not entirely satisfied with "extra" SMs, the SMLP allows such an allocation without needing any change in the blocking analysis. We show in the experimental evaluation that the addition of even a single SM may halve the response time of some GPU kernels, thus, it may be beneficial to only allocate these extra SMs when requests are otherwise satisfied with their minimum h SMs. By improving the GPU kernel execution time on average, we improve safety margins by ensuring the worst-case kernel execution times are less likely to occur.

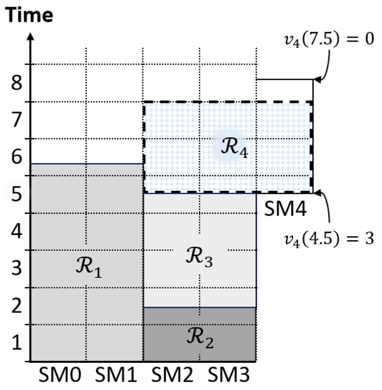

Figure 8 Example timeline where a component requires $\mathcal{H} = 4$ SMs, but may utilize an extra SM not initially promised to Γ, if it reduces $\mathcal{R}_4$'s completion time.

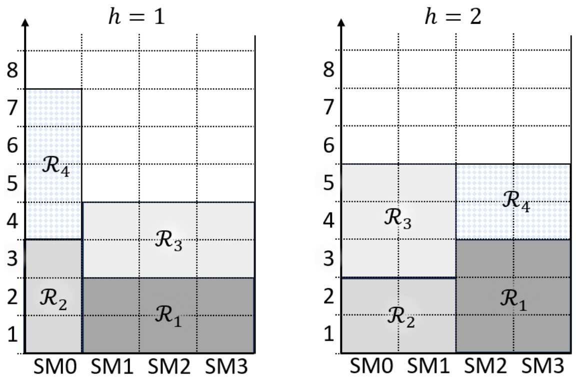

Figure 9 Example request satisfaction timelines with values of h configured as labeled. By increasing h from 1 to 2, $\mathcal{R}_4$ can complete faster, resulting in lower values for $L_{\max}$ and reducing the cost inflation b_i.

Pi-Blocking Optimizations. A GPU kernel may be forced to execute on as few as h SMs. Consider a request $\mathcal{R}_i$ to launch a highly parallelizable GPU kernel where the minimal SM allocation, $L_{i,h}$ is disproportionately larger than when satisfied with many SMs. Because the pi-blocking bound under the SMLP is dependent on $L_{\max}$ from Lem. 32, reducing the value of $L_{\max}$ may greatly reduce the pi-blocking bound of requests under the SMLP.

Fig. 9 shows how increasing h can decrease $L_{\max}$, thereby improving the pi-blocking bound of requests in SMLP. Such optimization can only be obtained by analyzing the underlying task set, as increasing h also results in all requests being satisfied with more SMs, and thus, fewer requests may run simultaneously.

6 Experimental Evaluation

In this section, we evaluate the SMLP via two sets of experiments. First, we present the results of a simulation study in which the worst-observed pi-blocking is compared among the SMLP and OMLP [8]. Second, we observe critical-section durations when utilizing `libsmctrl` [3] as measured on an NVIDIA RTX 4080, and use those observations to further elucidate our simulation results.

Task Generation. We consider *periodic* tasks where each task's period, T_i, is sampled uniformly from small, medium, and large ranges given by $(3, 33)$ms, $(10, 100)$ms, and $(50, 200)$ms, respectively. For each T_i sample range, we consider components with a processor count of $\mathcal{M} \in \{4, 8, 12, 16\}$ processors. For each processor count, we chose the number of tasks, n, uniformly, where $n \sim U(2\mathcal{M}, 150)$. We then consider *normalized utilizations* $U \in \{0.2, 0.3, 0.4, ..., 0.9\}$, where $U = (\sum_{i=1}^{n} {}^{u_i}/_{\mathcal{M}})$ and each task has a utilization $u_i = C_i/T_i$. For each normalized utilization, we generate values for the worst-case execution time, C_i, for each task τ_i as described in [11]. For each generated task set, we observe pi-blocking when $\mathcal{H} \in \{8, 16, 24, 32, 40, 48, 56, 64\}$ and $\Theta \in \{1.5, 2.0, 2.5, 3.0, \infty\}$. For each combination of T_i sample range, $\mathcal{M}$, U, Θ, and $\mathcal{H}$, we generate 1,000 task sets and determine the worst-observed pi-blocking time when each task set utilizes the SMLP and the OMLP. We call each combination a *scenario*.

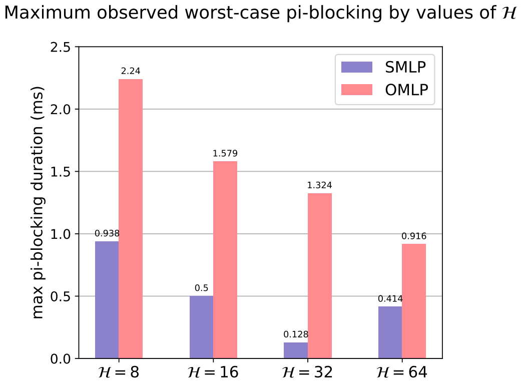

Figure 10 Shown are the largest observed pi-blocking durations across values of $\mathcal{H}$ when $\mathcal{M}=8$, $\Theta=2.5$, $T_i \in [10,100]$, $p_{req}=0.5$, and $U=0.6$.

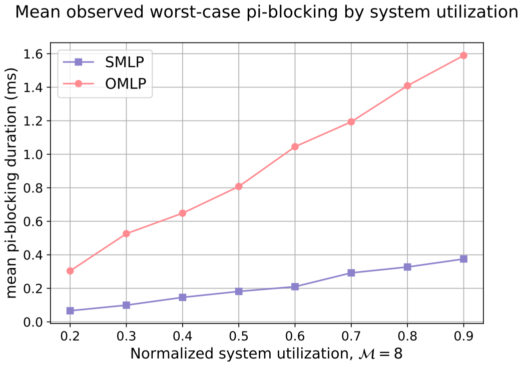

Figure 11 Shown are mean pi-blocking times across normalized utilization values when $\mathcal{H}=16$, $\mathcal{M}=8$, $\Theta=2.5$, $T_i \in [10,100]$, and $p_{req}=0.5$.

Request Generation. For every generated task set, we select a task τ_i with probability p_{req} where the jobs of a selected τ_i issue a request $\mathcal{R}_i$ when released. We repeat the above task generation for each $p_{\text{req}} \in \{0.1, 0.25, 0.5\}$. For each $\mathcal{R}_i$, we determine the worst-possible critical-section duration where $L_i^{\max} \sim U(0, C_i)$. We then uniformly sample a maximum parallelism amount, $\rho_i^{\max}$ in the range $[1, \mathcal{H}]$, where any reduction in critical-section duration ceases when allocated above $\rho_i^{\max}$ SMs. In our second set of experiments, we observe that additional SM allocations do not scale linearly. Thus, we pick the values for each $L_{i,j}$:

$$L_{i,j} = \frac{L_i^{\max}}{\rho_i^{\max}} \cdot \max\left(\left\lceil \frac{\rho_i^{\max} - j + 1}{j} \right\rceil, 1\right).$$

The maximum parallelism amount simulates requests that cannot utilize all $\mathcal{H}$ SMs. We justify the step-graph generated by the $L_{i,j}$ equation in the second set of experiments.

For the SMLP, we define $z_i(t)$ to be the smallest number of SMs, j, where $L_{i,j} \leq L_{i,|\mathcal{I}(t)|}$. For the OMLP, we define $z_i(t) = \mathcal{H}$ where the entire GPU is locked.

Experiment 1. We present a representative sample of our results in Fig. 10 and 11. Each locking protocol was scheduled under global EDF scheduling [9]. For each scenario, we determined the largest observed pi-blocking.

Shown in Fig. 10, we see the largest observed pi-blocking for the SMLP is significantly lower than that of the OMLP. With small values for $\mathcal{H}$, where the entire GPU is utilized by most requests, the SMLP still performs well when compared to the OMLP. With large values for $\mathcal{H}$, there is relatively low pi-blocking as there are plenty of available SMs for both locking protocols. We also observed that with small values for Θ, most of the pi-blocking incurred is from fz-blocking.

Shown in Fig. 11, when the normalized utilization is low, then the GPU is lightly utilized, and as such, the SMLP performs relatively similarly to the OMLP. Unsurprisingly, larger utilizations show both locking protocols incurring more pi-blocking as more tasks request GPU access. In higher utilizations, where it is more likely to have lengthy requests with small SM allocations, we observed that the SMLP still obtained lower pi-blocking times due to the increased parallelism afforded. Under most scenarios, we see pi-blocking improvements by up to 50% when compared to the OMLP.

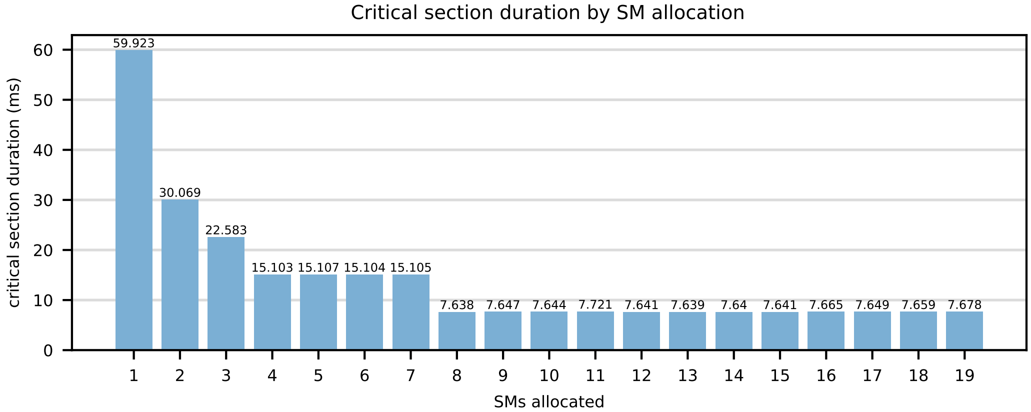

Figure 12 Shown are critical-section durations when a kernel is optimally satisfied with 8 SMs.

Experiment 2. In our second set of experiments, we determine GPU kernel critical-section durations when executed on an NVIDIA 4080 using `libsmctrl` [3]. This experiment seeks to validate the assumption that GPU kernel execution durations can be reduced when the kernel can utilize the additional parallelism afforded by increasing the number of SMs. Importantly, we also validate that SMs beyond the parallelism requirements of the GPU kernel provide no meaningful benefit in execution duration. In this experiment, we perform vector math on vectors that are sized to utilize 8 SMs (needing 16,384 simultaneous operations). Any additional SMs afford no benefit as the vectors have only so much data that needs processing.

Shown in Fig. 12, we test the execution time when allocated between 1 and 19 SMs. We observe that additional SM allocations beyond the parallelism requirements of a GPU kernel afford no benefit. As such, this justifies why pi-blocking and request response times can be reduced significantly when compared to locking the entire GPU.

Additionally, the step-graph in Fig. 12 justifies the step-graph generated for L_i in the first experiment. We see that GPU kernels have an optimal number of SMs, $\rho_i^{\max}$, where additional SMs provide no meaningful reduction in execution cost. In this experiment, the optimal allocation is $\rho_i^{\max} = 8$. This also justifies the definition of $z_i(t)$ where certain ranges of SM allocations provide no meaningful benefit over a lower amount. Consider when $|\mathcal{I}(t)| = 6$, then $z_i(t)$ for this GPU kernel should return 4 as there is no meaningful benefit to allocating 5 or 6 SMs.

7 Conclusion

We have presented the SMLP, a multiprocessor real-time locking protocol that enables safe GPU sharing in time-sliced component-based systems. The SMLP utilizes a novel resizing technique where GPU kernels are resized to fit on available SMs. We also provided an upper bound on pi-blocking time that can be computed in linear time. While computing a tight pi-blocking time is known to be an NP-hard problem [14] [7], the GPU request satisfaction techniques of the SMLP allow for a reasonable upper bound approximation. This approximation is made possible by allowing the SMLP analysis to upper-bound how requests are sized rather than by analyzing the underlying task set. In conclusion, the results of this paper show how a GPU can be shared across components in such a way that the worst-case pi-blocking time is easily computable and lower than the coarse-grain locking of the OMLP.

Future Work. This work creates a platform for sharing partitioned GPUs in component-based systems. We would like to extend the SMLP to support additional in-component JLFP schedulers such as clustered or partitioned scheduling. Such schedulers are not compatible with priority inheritance, and thus, would require additional novel blocking analysis. Similarly, the selection of an optimal value for minimum allocations (h) and the number of SMs optimally assigned to a component ($\mathcal{H}$) is left to future work. We also intend to explore how to size a component's periodic reservation (Θ, Π) when GPUs can be shared.

References

1 Tanya Amert, Zelin Tong, Sergey Voronov, Joshua Bakita, F. Donelson Smith, and James H. Anderson. Timewall: Enabling time partitioning for real-time multicore+accelerator platforms. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 455–468, 2021. `doi:10.1109/RTSS52674.2021.00048`.

2 Madhukar Anand, Arvind Easwaran, Sebastian Fischmeister, and Insup Lee. Compositional feasibility analysis of conditional real-time task models. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 391–398, 2008. `doi:10.1109/ISORC.2008.47`.

3 Joshua Bakita and James H. Anderson. Hardware compute partitioning on nvidia gpus*. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 54–66, 2023. `doi:10.1109/RTAS58335.2023.00012`.

4 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, 2010.

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

6 Alessandro Biondi, Giorgio Buttazzo, and Marko Bertogna. A design flow for supporting component-based software development in multiprocessor real-time systems. *Real-Time Systems*, 54, October 2018. `doi:10.1007/s11241-018-9301-3`.

7 Bjorn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, USA, 2011. AAI3502550.

8 Bjorn B. Brandenburg and James H. Anderson. Optimality results for multiprocessor real-time locking. In *2010 31st IEEE Real-Time Systems Symposium*, pages 49–60, 2010. `doi:10.1109/RTSS.2010.17`.

9 U.M.C. Devi and J.H. Anderson. Tardiness bounds under global edf scheduling on a multiprocessor. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 12 pp.–341, 2005. `doi:10.1109/RTSS.2005.39`.

10 Glenn A. Elliott. *Real-Time Scheduling for GPUs with Applications in Advanced Automotive Systems*. PhD thesis, University of North Carolina at Chapel Hill, USA, 2015. URL: `https://api.semanticscholar.org/CorpusID:63240444`.

11 P. Emberson, R. Stafford, and R.I. Davis. Techniques for the synthesis of multiprocessor tasksets. *WATERS'10*, January 2010.

12 Philip Holman and James H. Anderson. Locking under pfair scheduling. *ACM Trans. Comput. Syst.*, 24:140–174, 2006. URL: `https://api.semanticscholar.org/CorpusID:7948325`.

13 Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. Fractional gpus: Software-based compute and memory bandwidth reservation for gpus. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 29–41, 2019. `doi:10.1109/RTAS.2019.00011`.

14 Catherine E. Nemitz, Kecheng Yang, Ming Yang, Pontus Ekberg, and James H. Anderson. Multiprocessor real-time locking protocols for replicated resources. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 50–60, 2016. `doi:10.1109/ECRTS.2016.29`.

15 Feddal Nordine, Houssam-Eddine Zahaf, and Giuseppe Lipari. Toward Precise Real-Time Scheduling on NVidia GPUs. In *15th Junior Researcher Workshop on Real-Time Computing 2022*, Paris, France, June 2022. URL: https://hal.science/hal-03834181.

16 Nathan Otterness, Vance Miller, Ming Yang, James H. Anderson, F. Donelson Smith, and Shige Wang. Gpu sharing for image processing in embedded real-time systems. In *2016 12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2016. URL: https://api.semanticscholar.org/CorpusID:14892240.

17 Paul J. Prisaznuk. Arinc 653 role in integrated modular avionics (ima). In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 1.E.5–1–1.E.5–10, 2008. doi:10.1109/DASC.2008.4702770.

18 Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990. doi:10.1109/12.57058.

19 Sergey Voronov, Stephen Tang, Tanya Amert, and James H. Anderson. Ai meets real-time: Addressing real-world complexities in graph response-time analysis. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 82–96, 2021. doi:10.1109/RTSS52674.2021.00019.

20 Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 119–130, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2751205.2751213.

21 Tyler Yandrofski, Jingyuan Chen, Nathan Otterness, James H. Anderson, and F. Donelson Smith. Making powerful enemies on nvidia gpus. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 383–395, 2022. doi:10.1109/RTSS55097.2022.00040.

A Summary of Notation

Given below is the summary of the notation used throughout this work. All definitions are in the context of Γ, the current component, unless otherwise stated.

Table 1 Summary of Notation.

Γ	The component in question
Θ	The uninterrupted execution duration of Γ per time slice
Π	The period of Γ
Φ	The start time of Γ's most recent time slice
Υ	Set of resources required by Γ
$\mathcal{M}$	The number of required processors
$\mathcal{H}$	The number of required SMs
$\mathcal{L}$	The set of $\mathcal{H}$ identical SMs
τ_i	An arbitrary task where $\tau_i = (C_i, T_i, D_i)$
τ	The set of all tasks where $\tau_i \in \tau$
C_i	The worst-case CPU execution cost of τ_i
T_i	The minimum separation between job releases of τ_i
D_i	The relative deadline by which each job of τ_i must complete after releasing
J_i	An arbitrary job of τ_i
$\mathcal{R}_i$	An arbitrary GPU request issued by J_i, contains L_i
h	A multiple of h SMs may satisfy $\mathcal{R}_i$
L_i	An ordered set of critical-section durations $\{L_{i,h}, L_{i,2h}, ..., L_{i,\mathcal{H}}\}$
$L_{i,k}$	The critical-section duration of $\mathcal{R}_i$ when allocated k SMs
$L_i^{\max}$	The largest critical-section duration in L_i
$L_{\max}$	The largest critical-section duration of any GPU kernel
$\mathcal{A}_{i,k}$	The work of $\mathcal{R}_i$ when allocated k SMs where $\mathcal{A}_{i,k} = k \cdot L_{i,k}$
$\mathcal{A}_i^{\max}$	The largest $\mathcal{A}_{i,k}$ for $\mathcal{R}_i$ when k SMs can be allocated by z_i
$\mathcal{I}(t)$	The set of SMs not allocated to any job at time t, where $\mathcal{I}(t) \subseteq \mathcal{L}$
$z_i(t)$	An upper bound on the number of allocable SMs to request $\mathcal{R}_i$ at time t