DySM: Dynamic Scaling of GPU Streaming Multiprocessor in Spatially Shared Real-Time Embedded GPU Systems

Srinivasan Subramaniyan
Electrical and Computer Engineering, The Ohio State University, Columbus, OH, USA

Xiaorui Wang
Electrical and Computer Engineering, The Ohio State University, Columbus, OH, USA

Abstract

Many of today's real-time embedded systems are increasingly relying on GPUs for AI-related computing. However, existing GPU scheduling solutions for spatially shared GPU systems are still mostly *open-loop* and rely on worst-case execution time (WCET) estimation for offline schedulability analysis, which cannot adapt to online workload variations. Although adaptive scheduling has been proposed to handle runtime execution time variations, prior approaches target either CPU or time-slicing GPUs, where only one task can execute on the GPU within a time slice. In contrast, *spatial sharing* enables concurrent kernel execution via Streaming Multiprocessor (SM) partitioning, allowing better GPU resource utilization. Therefore, new adaptive solutions must be designed for *spatially shared* GPU systems.

In this paper, we propose `DySM`, a *closed-loop* response time control algorithm for *spatially shared* GPUs in soft real-time systems. In face of runtime workload variations, `DySM` leverages dynamic SM scaling to control task response times with low runtime overhead. To model GPU resource contention among tasks, we analytically derive a multi-input-multi-output (MIMO) system model that captures the impact of SM scaling on the response times of different tasks. Based on this model, `DySM` is designed using feedback control theory for guaranteed system stability and control accuracy. Experimental results on an Nvidia GPU testbed demonstrate that `DySM` outperforms state-of-the-art solutions by providing runtime real-time guarantees. Compared to the best-performing baseline, `DySM` can reduce the deadline miss ratio by up to 90.93%.

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Real-time operating systems; Computing methodologies → Computational control theory

Keywords and phrases Real-time systems, GPU scheduling, response time ratio control, SM scaling, and feedback control

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2026.16

Supplementary Material *Software (Source Code)*: `https://github.com/srinivasans74/DySM`
archived at `swh:1:dir:40d31d1dbc6ff9c2018f4bdbb3b56dfed9c93dc4`
Software (ECRTS 2026 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.12.2.4`

Funding This work was supported, in part, by the U.S. NSF under Grant CNS-2344505.

Acknowledgements We would like to thank the anonymous reviewers for their valuable comments.

1 Introduction

Recent years have witnessed rapidly increasing applications of artificial intelligence (AI) and machine learning (ML) in real-time embedded systems [25]. As powerful accelerators, GPUs have enabled the development of high-performance and mission-critical autonomous systems [25] [28]. A well-known example of GPU-based embedded systems is self-driving

cars, which rely on onboard GPUs for real-time object detection and path planning [22, 58]. Other examples include immersive virtual reality (VR) headsets that use embedded GPUs for processing and rendering 3D video [30, 31] and autonomous drones that employ GPUs for recognizing obstacles and adjusting their flight path accordingly [41]. To facilitate those real-world applications, scheduling real-time tasks on GPUs has recently become an important area of research and received a lot of attention [5, 11, 28, 29, 42, 46, 49, 56, 59].

There are several different ways to schedule real-time tasks on GPUs. Earlier work has tried to model a GPU as a dedicated server that handles GPU requests from all the tasks in the system [27, 29]. By doing that, only one task can access the shared GPU at a time, and the task can finish all its GPU computing before other tasks can access the GPU. This methodology can ensure bounded GPU execution time and simplifies the design of GPU scheduling. However, it may result in degraded GPU schedulability because other tasks cannot access the GPU even when this task only needs a small portion of the available GPU resources. To improve GPU utilization by allowing concurrent GPU access, different solutions have been proposed, such as *time-slicing* and *spatial sharing*. For time-slicing, multiple real-time tasks can access the GPU at the same time as separate OS processes. The GPU time is divided as fine-grained time slices, so that only one task/process can run within a time slice and the tasks take turns to run on the GPU to their completion. Time-slicing is similar to real-time scheduling on a single CPU core, as it serializes the GPU kernel execution. In addition to time-slicing, modern GPUs also support spatial sharing, which enables concurrent execution by partitioning the GPU's streaming multiprocessors (SMs) among multiple tasks [2, 3, 42, 56]. Examples of spatial sharing include Nvidia's Multi-Process Service (MPS) [16] and AMD's CU Masking [38]. For example, MPS allows the compute kernels from different real-time tasks to execute on the same GPU at the same time, thus achieving efficient GPU sharing and better GPU utilization. Thus, spatial sharing is like running different real-time tasks concurrently on separate cores of a CPU, which can result in better schedulability.

While running real-time tasks in partitioned SMs can result in better GPU scheduling for *spatially shared* GPU systems, existing research is still mostly *open-loop*, as the task execution times on GPU need to be carefully estimated at compile time to ensure there are no deadline misses [42]. To be more specific, in such open-loop solutions, how many SMs of a GPU should be allocated to each real-time task is based on the task's estimated worst-case execution times (WCETs). Then, offline schedulability analysis is performed to ensure all tasks meet their deadlines if the estimated WCETs are accurate and the real task execution times would not vary significantly at runtime. Unfortunately, though open-loop solutions can be effective for hard real-time systems that run in closed environments, GPU-based embedded systems are often soft real-time systems that run in open environments, in which their execution times can have large variations due to their inputs (e.g., sensor readings). For example, the perception task in autonomous driving can experience execution-time increases of more than three times, depending on the number of detected objects and scene complexity, because the number of bounding boxes directly affects kernel execution time and memory traffic [4]. Hence, any underestimation of WCETs can lead to undesired deadline misses, while overestimation can result in GPU resource wastage.

Adaptive scheduling solutions have been previously proposed to solve runtime variations of task execution time for CPU-based soft real-time systems. Such scheduling solutions keep monitoring the task response time online and make necessary system or workload adaptation, in a closed-loop manner, to quickly react to execution time variations. However, those solutions cannot be directly applied to GPU-based real-time systems, because GPUs have

different and more complex architectures (e.g., SMs). In recent years, some studies have proposed adaptive scheduling methodologies for GPU-based real-time systems. For example, FC-GPU [49] has proposed to adjust the invocation rates of periodic tasks to meet their real-time deadlines. However, their work is designed for time-slicing GPUs, thus cannot be applied to spatially shared GPUs that allow better real-time scheduling. Also, the rates of some real-world tasks may not be adjustable in many real-time applications, because they are determined by physical sensors. Hence, a closed-loop scheduling solution must be designed for *spatially shared* GPU systems to fully utilize the GPU computing resources.

In this paper, we propose `DySM`, the first closed-loop, adaptive GPU scheduling algorithm for spatially shared GPUs in soft real-time systems. In sharp contrast to open-loop solutions that allocates SMs to different real-time tasks based on their WCETs in an offline manner, `DySM` leverages dynamic SM scaling to control the task response time at runtime. Specifically, `DySM` is designed to ensure that each task's response time is shorter than its deadline by adapting its SM allocation. For example, when the execution time of a task increases significantly, `DySM` can dynamically allocate more SMs to this task for faster GPU execution and thus a better chance of meeting its deadline. `DySM` is designed based on the observations that dynamic SM scaling incurs only negligible overhead and can have immediate effect on the task execution times. However, given the total number of SMs in a GPU is fixed, allocating more SMs to one task means fewer SMs to other tasks. Hence, to model such GPU resource contention among tasks, we analytically derive a multi-input-multi-output (MIMO) system model that captures the impact of SM scaling on the response times of different tasks. Then, using the proposed MIMO model, `DySM` is rigorously designed based on feedback control theory to guarantee the system's stability and control performance analytically.

Specifically, this paper makes the following contributions:

- Unlike prior work that relies on *open-loop* GPU scheduling with WCET assumptions, we design a closed-loop GPU scheduling algorithm, `DySM`, for spatially shared GPUs in soft real-time systems. Our approach continuously monitors task response times at runtime and adaptively adjusts the number of SMs assigned to each task to maintain real-time guarantees.
- We analytically derive a MIMO system model that captures the impact of SM allocation on the relative response times of multiple tasks. Based on this model and feedback control theory, we design a controller that explicitly accounts for cross-task coupling and outperforms baseline approaches that control each task independently.
- We present hardware testbed results on an Nvidia RTX 3090 GPU to demonstrate the efficacy of the proposed `DySM` algorithm. Extensive results show that `DySM` outperforms state-of-the-art solutions by providing runtime real-time guarantees. Compared to the best-performing baseline, `DySM` can reduce the deadline miss ratio by up to 90.93%.

The rest of the paper is organized as follows. Section 2 introduces the background. Section 3 formulates the control problem and discusses the system architecture. Section 4 presents the controller design. Section 5 describes the experimental setup, and the hardware experiments are presented in Section 6. Section 7 discusses the related work and Section 8 concludes the paper.

2 Background

In this section, we introduce the background on GPU scheduling and SM scaling.

2.1 GPU Scheduling

Nvidia GPUs offer various concurrency mechanisms that influence how tasks are scheduled [21]. Previous studies (e.g., [9,12,21]) indicate that task behavior is significantly affected by whether tasks are launched from separate processes or from a single process. When tasks are launched from separate processes, they are by default allocated time slices. However, when tasks are launched using Nvidia's Multi-Process Service (MPS) from separate processes, they are shared spatially [12, 46, 48]. MPS is not used for real-time scheduling because it introduces substantial runtime overhead. MPS reassigns streaming multiprocessors (SMs) only upon task termination, resulting in unpredictable execution delays and violating the timing determinism required in real-time environments [12, 17]. When tasks are launched from a single process, they share a common GPU context and are typically issued into CUDA streams. In this case, kernels may be spatially shared across available SMs, enabling concurrent execution. However, this *spatial sharing* is opportunistic and not strictly partitioned; kernels may still interfere with one another depending on scheduling behavior and resource contention [40]. To mitigate the contention, Bakita et al. have proposed a hardware-level compute partitioning mechanism that enables true spatial sharing of GPU cores [8]. This approach leverages undocumented mask fields in the kernel's task metadata (TMD) structure to restrict execution to a specified subset of compute units (e.g., TPCs/SMs). By activating these mask fields prior to kernel submission, the hardware scheduler is constrained to dispatch thread blocks only within the assigned partition. As a result, tasks assigned to separate partitions can execute simultaneously with bounded interference, improving predictability even in the presence of long-running or compute-saturating kernels.

Although *spatial sharing* can be achieved either through MPS enabled multi-process execution or through single-process stream-based concurrency, we adopt the single-process approach. This design choice mitigates overheads associated with MPS, removes reliance on hardware capabilities that are absent on a substantial fraction of edge GPUs [46, 56], and prevents latency introduced by SM reallocation occurring only upon task completion.

2.2 SM Scaling

When tasks are launched from the same process, spatial sharing can be explicitly controlled using two primary approaches: (1) persistent threads [60] [56] and (2) hardware-level SM masking (via `libsmctrl`) [8]. Persistent thread technique restructures kernels so that a fixed number of thread blocks remain resident on designated SMs and cooperatively schedules work in software, thereby emulating application-level partitioning. However, this approach requires kernel modification, introduces additional synchronization overhead, and reduces portability across different GPU architectures and programming frameworks.

In contrast, `libsmctrl` enables spatial partitioning by directly applying hardware-supported TPC masks at launch time, without modifying the application kernels. By configuring the TPC mask fields in the kernel's task metadata, the hardware scheduler is constrained to dispatch thread blocks only within a specified subset of compute units. Since TPCs represent the native hardware grouping of SM resources, masking at the TPC level aligns with the GPU's dispatch hierarchy and provides deterministic spatial isolation. This mechanism preserves CUDA scheduling semantics and avoids persistent-thread overhead. Beginning with the Pascal architecture, each Nvidia GPU contains two SMs per TPC; therefore, controlling the number of active TPCs provides slightly coarser-grained yet effective control over SM allocation. In this work, we leverage TPC masking using `libsmctrl` as the actuator for our scheduler. We use TPC masking to dynamically scale the number of SMs assigned to each task (referred to as `SM Scaling`).

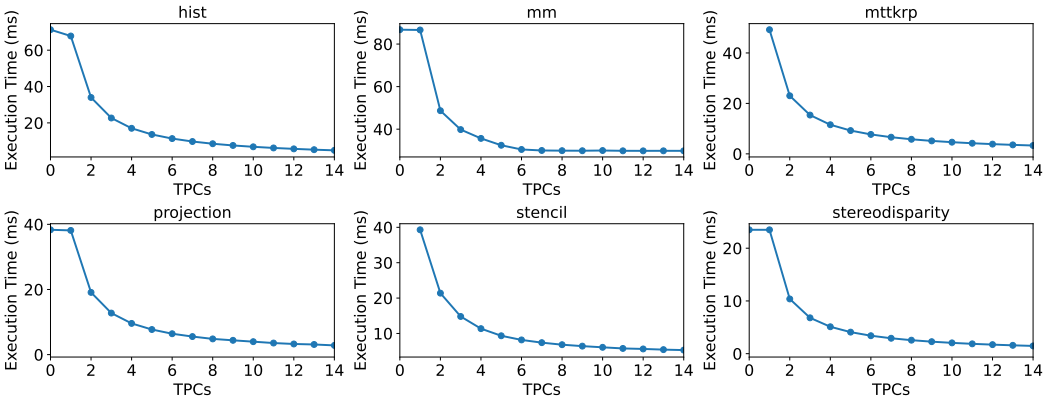


Figure 1 Impact of SM scaling on GPU execution time. Increasing the number of allocated TPCs (and hence SMs) reduces task execution time across all evaluated workloads.

Once a kernel corresponding to a task instance is launched in a stream with a fixed TPC mask, it executes non-preemptively on the assigned SMs until completion [24, 46]. The assigned partition configuration remains unchanged during kernel execution. Instead, SM reassignment is deferred until the subsequent job release and is implemented by updating the TPC mask before launching the next kernel instance. This reconfiguration incurs negligible overhead, as it only requires modifying mask fields in the kernel's launch metadata on the CPU side and does not involve reinitialization. When a kernel completes, all per-block state (e.g., registers, shared memory, and warp context) is automatically released by the hardware. Therefore, reassigning those TPCs to another task does not require state migration, context switching, or data movement. Consequently, partition resizing at task boundaries introduces minimal runtime overhead while preserving timing predictability.

To empirically demonstrate the effectiveness of SM scaling, we conduct an experiment on an RTX 3090 GPU. In this experiment, we vary the number of TPCs assigned to all workloads described in Section 5 and measure their execution times. These workloads (e.g., from Rodinia and CUDA samples) have been widely used in prior real-time and GPU scheduling research [8,39,49,55,57], and represent a diverse set of compute and memory-bound kernels that are representative of practical embedded and AI workloads.

As shown in Figure 1, increasing the number of allocated TPCs reduces execution time within the scalable region of each workload. However, beyond a certain point, performance flattens because the workload cannot be further parallelized. This saturation occurs due to limited parallelism (i.e., a non-zero serial fraction), finite thread-block availability, and memory bandwidth constraints. As a result, additional TPCs provide diminishing or negligible improvement once the parallel portion is fully exploited. We define the effective scaling region of a workload as the range of TPC allocations over which increasing the number of TPCs yields a non-negligible performance improvement. Specifically, we identify this region empirically as the set of allocations for which the marginal speedup exceeds a small threshold (e.g., 5–10%). Outside this region, additional TPCs provide diminishing returns, reducing the effectiveness of SM scaling for response-time control.

3 System Overview

In this section, we first describe the task model in Section 3.1. We then present the overall architecture of `DySM` in Section 3.2, followed by the control problem formulation in Section 3.3.

3.1 Task Model

In real-time systems, a task is implemented as a process or thread that is scheduled to run on the target GPU during specific time intervals. Each task has multiple instances, known as jobs. A job's release time is the point at which the job becomes ready to execute on the GPU, and the system can begin scheduling and running it. In this paper, to simplify the design, we assume all tasks are *periodic with implicit deadlines* (i.e., the relative deadline equals the period). This assumption is common in embedded and real-time systems, where periodic tasks interact with the environment, such as sensors or cameras [43]. If a job does not complete within its period, the next job is released, and the current job becomes outdated. For example, in autonomous driving systems, tasks such as sensor data processing, decision-making, and vehicle control run periodically to ensure safe and efficient operation [29].

We now introduce the notations used throughout the paper. A system consists of N tasks (t_i, $1 \leq i \leq N$), where each task t_i is defined by the following parameters:

- $e_i(k)$: Kernel execution time of task t_i in the k^{th} control period.
- e_{hdi}: Host-to-device copy time of task t_i.
- e_{dhi}: Device-to-host copy time of task t_i.
- s_{max}: Maximum number of SMs in the GPU.
- $s_i(k)$: The number of SMs assigned to every job of t_i in the k^{th} control period.
- $u_i(k) = s_i(k)/s_{max}$ = normalized allocation (fraction in (0,1]).
- p_i: The period of t_i, which is a constant without rate adaptation.
- d_i: The relative deadline of t_i (assuming $d_i = p_i$).
- $q_i(k)$ = Average response time of all the jobs of t_i that are finished in the k^{th} control period. A job's response time is its finish time minus its release time.
- $r_i(k) = q_i(k)/p_i(k)$: Relative response time (RRT) of t_i in the k^{th} control period.
- z_i: Relative response time set point for t_i.

Our task model accommodates a wide range of periodic tasks, represented as $t(et_i, p_i)$, where et_i is the estimated GPU execution time and p_i is the task period. The estimated execution time is $et_i(k) = \frac{e_i(k)}{s_i(k)} + e_{hdi} + e_{dhi}$, where $e_i(k)$ is the kernel execution time, and e_{hdi} and e_{dhi} are the transfer times from host to device and device to host. We assume that e_{hdi} and e_{dhi} depend only on the size of the data and remain constant across jobs and SM allocations. The kernel execution time scales inversely with the number of SM's. As the number of SM's decreases, the kernel execution time increases [8] [56]. We assume that the total computational demand of all tasks is within the available GPU capacity, i.e., the system is not persistently overloaded. This is a standard assumption in real-time scheduling, as no scheduling or control algorithm can guarantee deadline satisfaction when the aggregate demand exceeds available resources. Furthermore, tasks are assumed to execute independently, with no precedence constraints or shared data dependencies.

3.2 System Architecture

In this section, we provide a high-level description of the `DySM` system architecture. As discussed above, a significant concern in real-time GPU scheduling is the risk of deadline violations. Thus, existing approaches tend to 1) perform offline analysis, where the response time of the task is theoretically computed as an upper bound and schedulability analysis is performed [42,56,57,60], and 2) over-provision GPU resources for each task to ensure that the worst-case execution time (WCET) does not cause any deadline misses. Such practices result in low runtime adaptability [42], which is an issue in many real-time environments, where new tasks may arise from user applications, system events, or sensor inputs [6, 7]. For example, in

autonomous vehicles, when an unexpected obstacle, such as a traffic cone, suddenly appears on the road, the vehicle must react immediately to prevent a collision. This event may trigger additional perception or path planning workloads that must execute with tight latency constraints. When this occurs, the system must rapidly reallocate GPU compute resources to accommodate the new task while ensuring that existing safety-critical tasks continue to meet their deadlines. Offline schedulability analysis assumes a fixed and known task set with predetermined resource allocations. When new tasks are introduced dynamically, the original analysis no longer holds because the compute partitioning assumed during analysis may no longer be sufficient. Similarly, over-provisioning resources for worst-case execution leaves little flexibility for reallocating streaming multiprocessors at runtime when unexpected workloads arise. Without the ability to dynamically adjust the number of SMs assigned to each task, the system may either violate deadlines or waste substantial computational capacity. Consequently, rigid schedulability guarantees based on static allocation can break down in highly dynamic environments, motivating the need for adaptive GPU compute partitioning mechanisms.

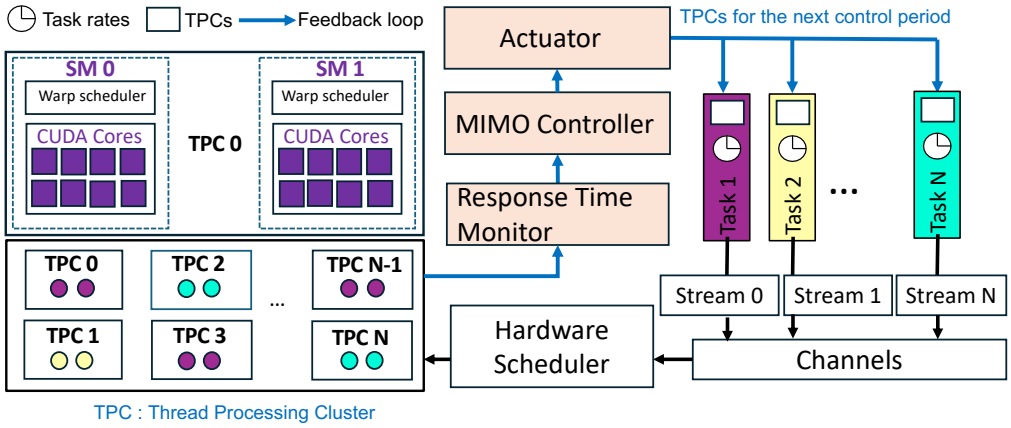


Figure 2 System architecture of DySM. Our solution employs a MIMO feedback control loop that periodically monitors each task's relative response time and dynamically scales the number of SMs assigned to each task to ensure that each task's response time does not exceed its deadline.

The overall architecture of DySM is shown in Figure 2. The framework employs a multi-input–multi-output (MIMO) feedback control loop that dynamically performs SM scaling to ensure that each task's relative response time converges to its specified set point. For a task, a relative response time of 1 indicates that the job finishes exactly at its deadline, whereas $r_i(k) > 1$ implies a deadline miss. For example, with a relative response time set point of 0.9 and a period of 2 s, each job should finish at 1.8 s.

A key feature of our task and system model is its ability to operate in unpredictable runtime environments, where task execution times and interference effects are not known a priori to the scheduler. Instead of relying on offline worst-case execution-time analysis, DySM adopts a closed-loop MIMO feedback control architecture that dynamically adapts GPU resource allocations at runtime to improve deadline adherence under runtime variability. As shown in Figure 2, each task submits its kernels to one or more CUDA streams for concurrent execution. Prior work [9] shows that Nvidia GPUs provide a limited number of compute channels per CUDA context (eight by default on Ampere GPUs). When the number of active streams exceeds the available channels, implicit serialization may occur due to channel sharing. Therefore, we limit the number of concurrent periodic tasks to the available compute

channels and assign each task to a dedicated CUDA stream to avoid unintended intra-context scheduling interference. We assume that each task is mapped to a single CUDA stream. Furthermore, we apply a consistent SM/TPC mask across all kernels within a task, ensuring a fixed resource partition throughout execution. Extending the approach to tasks using multiple streams would require additional coordination in the control interface and is left for future work.

In Figure 2, the control loop is invoked periodically at each sampling instant and consists of three main components: a `Response Time Monitor`, an `MIMO Controller`, and an `Actuator`. The control loop operates over a fixed control period, a configurable parameter that need not be an integer multiple of the task period. During each control period, the response-time monitor measures each task's response time $q_i(k)$ and computes the corresponding relative response time $r_i(k)$. To ensure stable and representative measurements, the control period is chosen to be sufficiently long to include multiple job completions per task. This averaging mitigates transient variations and provides a reliable estimate of task response time for feedback control. The MIMO controller then collects the $r_i(k)$ values of all tasks, compares them against their respective set points z_i, and determines the required control actions. Because the GPU's total SM budget is fixed, allocating additional SMs to one task necessarily reduces the resources available to others. This shared resource constraint introduces intrinsic coupling among tasks, as their response times depend on the same limited pool of SMs. Consequently, independent per-task controllers would ignore this interdependence and may lead to instability or inefficient allocation. Therefore, we adopt a MIMO controller to jointly regulate all task allocations and maintain system-level stability. Based on the controller's output, the actuator adjusts each task's compute allocation by updating the corresponding TPC masks (which quantize $s_i(k)$ to the hardware's TPC/granularity), thereby regulating the task's response time. In the following subsections, we formally formulate the control problem and present the detailed design of the MIMO controller.

3.3 Problem Formulation

The control problem can be formulated as a constrained optimization problem, as:

$$\begin{aligned} \min_{x} \quad & \sum_{i=1}^{N} \big(z_i - r_i(k)\big)^2, \qquad x = u(k) = \big[u_1(k), u_2(k), \ldots, u_N(k)\big]^{\mathsf{T}}, \\ \text{s.t.} \quad & \sum_{i=1}^{N} u_i(k) \leq 1, \qquad u_i(k) \in [u_{\min}, u_{\max}] \end{aligned} \tag{1}$$

where $u_i(k) \triangleq s_i(k)/s_{\max}$ and $u_{\min} = 1/s_{\max}$. The objective of the optimization is to minimize the deviation between the desired set point z_i and the relative response time $r_i(k)$. By adjusting the available SM allocations, the controller regulates GPU load and ensures that tasks meet timing constraints when an appropriate set point (e.g., $z_i = 0.9$) is selected. Rather than using a traditional offline method to solve this optimization problem, we employ a *control-theoretic approach* with a closed-loop feedback mechanism. Compared to offline optimization, a control-theoretic solution can dynamically adapt to runtime variations such as workload fluctuations, execution-time variability, and modeling inaccuracies. In this work, we assume the SM budget is typically tight; therefore, for controller derivation, we consider the constraint $\sum_{i=1}^{N} u_i(k) <= 1$. This reflects the normalized GPU resource budget, where $u_i(k)$ denotes the fraction of SMs allocated to task t_i. This constraint enforces that the total allocation does not exceed the available SM capacity and ensures that the controller redistributes a fixed resource budget among tasks. At each sampling interval, the

controller measures the system's current state and adjusts the number of SMs accordingly. This closed-loop design provides inherent robustness against performance degradation caused by unpredictable execution behavior (see Section 6).

4 Design

In this section, we model the control problem and present the detailed controller designs.

4.1 System Modeling

To apply a control-theoretic methodology, we establish a dynamic model that relates the *manipulated variables* (i.e., normalized SM allocation to every task) to the *controlled variables* (i.e., relative response time of every task). Our control objective is to manipulate SM allocations so that the relative response time of every task converges to its set point within a finite number of control periods. We define the manipulated variable for task t_i as

$$u_i(k) = \frac{s_i(k)}{s_{\max}}, \qquad u_i(k) \in [u_{\min}, u_{\max}], \tag{2}$$

where $s_i(k)$ is the number of SMs assigned to task t_i in control period k and $u_{\min} = 1/s_{\max}$ and we define $r_i(k)$ as the controlled variable.

We first model the response time of task t_i as $q_i(k)$. In Section 2.2, we have discussed that the response time of a task varies inversely with the number of SMs assigned to the task[1]. For a system comprising N tasks, we first obtain the response-time model for each task t_i via offline profiling within the operating region. We measure the response time of each task t_i under different SM allocations and fit the resultant data to an inverse model

$$q_i(k) = \frac{a_i}{s_i(k-1)} + b_i, \tag{3}$$

where $s_i(k-1)$ represents the number of SMs assigned to task t_i during the previous control period. The parameters a_i and b_i are determined through least-squares curve fitting. To create a generalized model, we fit the data across all workloads presented in Figure 1 and average the coefficients. This process yields generalized curve-fitting parameters (a_i and b_i) that represent the scalable component and the allocation-independent offset, respectively. Using the normalized allocation in Equation (2), we substitute $s_i(k) = u_i(k) \times s_{\max}$ to obtain

$$q_i(k) = \frac{a_i}{u_i(k-1)s_{\max}} + b_i. \tag{4}$$

Defining $e_i \triangleq \frac{a_i}{s_{\max}}$, the model becomes

$$q_i(k) = \frac{e_i}{u_i(k-1)} + b_i, \qquad \forall i \in \{1, \ldots, N\}. \tag{5}$$

In Section 3, we define the relative response time ($r_i = \frac{q_i}{p_i}$) of a task as the ratio of its response time to period. Equation (5) can be converted as follows

$$r_i(k) = \frac{e_i}{p_i \times u_i(k-1)} + b_i. \tag{6}$$

[1] In general, GPU kernels may exhibit non-linear speedup as compute resources increase (e.g., due to memory bandwidth limits, cache effects, or occupancy saturation) [56]. Our model, therefore, serves as a local approximation around the operating region; incorporating explicit non-linear speedup models is left for future work.

The SMs are shared across tasks and must satisfy the global budget constraint stated in the problem formulation in Section 3 ($\sum_{i=1}^{N} u_i(k) \leq 1$), where $u_i(k) = s_i(k)/s_{\max}$ denotes the normalized SM allocation for task t_i. This shared constraint induces coupling among tasks, as increasing the allocation of one task reduces the resources available to others.

To facilitate controller design, we next derive a linear discrete-time incremental model by linearizing the r_i expression around the current operating point. Specifically for a system comprising N tasks, we write the incremental form for each task as

$$r_i(k) = r_i(k-1) + \sum_{j=1}^{N} b_{ij}\, \Delta u_j(k), \tag{7}$$

where

$$b_{ij} = \begin{cases} a_i, & i = j, \\ -\dfrac{\gamma}{N-1}\, a_i, & i \neq j, \end{cases} \qquad a_i = \left.\frac{\partial r_i}{\partial u_i}\right|_{\text{op. point}}, \tag{8}$$

and $\gamma \in [0, 1]$ is a coupling coefficient that captures the strength of interaction induced by the shared SM budget. Stacking all tasks, we obtain the coupled MIMO model as:

$$\mathbf{r}(k) = \mathbf{r}(k-1) + \mathbf{B}\, \Delta \mathbf{u}(k), \tag{9}$$

where $\Delta \mathbf{u}(k) = [\Delta u_1(k), \ldots, \Delta u_N(k)]^{\mathsf{T}}$ and

$$\mathbf{B} = \begin{bmatrix} a_1 & -\frac{\gamma}{N-1}a_1 & \cdots & -\frac{\gamma}{N-1}a_1 \\ -\frac{\gamma}{N-1}a_2 & a_2 & \cdots & -\frac{\gamma}{N-1}a_2 \\ \vdots & \vdots & \ddots & \vdots \\ -\frac{\gamma}{N-1}a_N & -\frac{\gamma}{N-1}a_N & \cdots & a_N \end{bmatrix}. \tag{10}$$

Independent SISO controllers assume $r_i = f_i(u_i)$ and adjust u_i without accounting for the shared budget. As a result, multiple controllers may issue conflicting control actions, leading to oscillations, saturation, or inefficient resource allocation (see Section 6.1). Hence, the shared SM constraint renders independent SISO control inadequate, necessitating a MIMO formulation that jointly regulates the `RRT` of all tasks. The analytical model assumes non-overlapping spatial partitions and captures coupling induced by the shared SM budget. In practice, when the requested allocation is infeasible, the actuator may allow bounded overlap as a fallback to maintain progress; we treat this as an implementation-level feasibility mechanism and empirically evaluate its impact (See Section 5).

4.2 Controller Design and Analysis

We present the design and analysis of our MIMO controller `DySM`. We first derive the state representation of the model. Next, this formulation is transformed into a closed-loop solution, which allows us to design the control algorithm.

4.2.1 State Space Representation

Based on the system model, a MIMO controller can be designed to ensure that the `RRT` settles at its desired set point. The PID control approach used in earlier work on real-time feedback control scheduling [36, 49] is well-suited to real-time systems due to its negligible

time overhead. Our design employs MIMO controllers, which differ from single-input single-output (SISO) controllers. SISO controllers calculate the next state output based on a single set point and its gain, while MIMO controllers take into account all set points and their corresponding gains. This approach ensures minimal steady-state errors or oscillations and provides a robust design for task scheduling.

We design a discrete-time proportional `MIMO` controller based on the reduced-order coupled MIMO model in Equation (9). We define the state as the `RRT` vector $\mathbf{x}(k) = \mathbf{r}(k) \in \mathbb{R}^N$ and the output as $\mathbf{y}(k) = \mathbf{x}(k)$. The controller input is the allocation increment vector $\Delta\mathbf{u}(k) \in \mathbb{R}^N$, where $\Delta\mathbf{u}(k) = [\Delta u_1(k), \ldots, \Delta u_N(k)]^\mathsf{T}$.

The state-space form is

$$\begin{aligned} \mathbf{x}(k) &= \mathbf{x}(k-1) + \mathbf{B}\,\Delta\mathbf{u}(k), \\ \mathbf{y}(k) &= \mathbf{x}(k), \end{aligned} \tag{11}$$

where $\mathbf{B} \in \mathbb{R}^{N\times N}$ is given in Equation (10).

4.2.2 Controller Design

Let $\mathbf{z} \in \mathbb{R}^N$ denote the vector of `RRT` set points. `DySM` uses a proportional control law to compute the allocation increments for the independent inputs. The equation for a proportional MIMO controller is given by

$$\Delta\mathbf{u}(k) = \mathbf{K}_p\left(\mathbf{z} - \mathbf{y}(k-1)\right), \tag{12}$$

where $\mathbf{K}_p \in \mathbb{R}^{N\times N}$ is the proportional gain matrix, $\mathbf{z}$ is the reference vector representing the desired set points, and $\mathbf{y}(\mathbf{k-1})$ is the actual output of the system. After determining the desired system behavior, we use pole placement to determine the optimal gain matrix. This method adjusts the system's response by placing its poles in specific locations. We put these poles within the unit circle to ensure stability and performance, as discussed in references [49] and [52] [12].

Using the reduced-order plant model in Equation (11) and the proportional control law in Equation (12), `DySM` computes an update to the manipulated variables in the form of a normalized allocation increment vector $\Delta\mathbf{u}(k)$, where $u_i(k) = s_i(k)/s_{\max}$ denotes the fraction of SMs assigned to task t_i in control period k.

At each control period k, we update the normalized allocations for the tasks as

$$u_i(k) = \mathrm{sat}(u_i(k-1) + \Delta u_i(k)), \qquad i \in \{1, \ldots, N\}, \tag{13}$$

where $\mathrm{sat}(\cdot)$ clamps $u_i(k)$ to the admissible range $[u_{\min}, u_{\max}]$. However, on our testbed, the GPU cannot be directly assigned a fractional $u_i(k)$. Instead, allocations must be realized through *integer* hardware partitions; even when tasks are allowed to share the GPU, the masking mechanism enables or disables *entire* TPCs, so fractional allocations (e.g., 3.3 TPCs) are not supported. On Nvidia GPUs, `DySM` partitions the SMs spatially by configuring per-stream TPC masks, where each enabled TPC corresponds to a fixed set of SMs [46] [8]. As a result, the allocation resolution is quantized, and the real-valued target $u_i(k)$ must be converted to an integer number of enabled TPCs.

To bridge this gap while maintaining the desired long-term allocation, we use a first-order delta-sigma modulator [37] to convert the desired real-valued allocation into a sequence of integer TPC counts whose time average tracks $u_i(k)$ with negligible overhead. Concretely, at each control period we compute an integer target $\hat{s}_i(k)$ (or $\widehat{\mathrm{TPC}}_i(k)$) from $u_i(k)$, apply the

corresponding TPC mask to task t_i, and carry the quantization error forward in the modulator state to avoid systematic bias. For example, if the controller computes a fractional allocation (e.g., 13.1 SMs), the modulator alternates between the nearest feasible values. Since each TPC contains 2 SMs, it toggles between 12 and 14 SMs (e.g., $12, 12, 14, 14$), so the time-averaged allocation approaches the target [37]. This actuation strategy provides (i) bounded quantization error, (ii) stable average tracking of the desired allocation, (iii) compatibility with existing GPU partitioning primitives, and (iv) low overhead, because applying compute masks requires only lightweight metadata updates, as described in Section 2.2. The overhead test in Section 6.4 also shows that SM scaling incurs only negligible runtime overhead.

4.3 Stability Analysis

A key advantage of control-theoretic design is its ability to provide analytical guarantees on system stability and control accuracy, even in the presence of modeling errors or runtime variations. In this section, we analyze the stability of the proposed MIMO proportional controller based on the discrete-time `RRT` system model derived in Section 4. We list the steps below for performing stability analysis.

1. Compute the control input $\Delta\mathbf{u}(k)$ using the nominal system dynamics defined in Equation (11). Note that the nominal system model is used for our controller design and can be different from the actual system model due to modeling errors or variations.
2. Substitute the resulting control law into the actual system model (which may have different system parameters from actual measurements) to obtain the corresponding closed-loop model. This model represents the response of the actual system to the control input derived from the nominal system model.
3. Analyze the stability of the closed-loop system by evaluating its eigenvalues; the system is stable if all eigenvalues lie within the unit circle, despite modeling errors or variations.

As example, we consider a system comprising N tasks, where the matrices in the state-space model (11) have the following dimensions: $\mathbf{B} \in \mathbb{R}^{N\times N}$, and $\mathbf{x}(k) \in \mathbb{R}^N$, $\Delta\mathbf{u}(k) \in \mathbb{R}^N$. From Equation (9), the discrete-time system dynamics are given by

$$\mathbf{r}(k) = \mathbf{r}(k-1) + \mathbf{B}\,\Delta\mathbf{u}(k), \tag{14}$$

where $\mathbf{B} \in \mathbb{R}^{N\times N}$ is the gain matrix defined in Equation (10), and $\Delta\mathbf{u}(k)$ denotes the increment in the independent normalized SM allocation vector. We define the system state as $\mathbf{x}(k) = \mathbf{r}(k)$, and the system output as $\mathbf{y}(k) = \mathbf{x}(k)$. The control objective is to regulate $\mathbf{x}(k)$ to a desired reference vector $\mathbf{z}$. We employ a discrete-time proportional MIMO control law of the form

$$\Delta\mathbf{u}(k) = \mathbf{K}_p\,(\mathbf{z} - \mathbf{y}(k-1))\,, \tag{15}$$

where $\mathbf{K}_p \in \mathbb{R}^{N\times N}$ is the proportional gain matrix. Substituting Equation (15) into Equation (14) yields the closed-loop system

$$\mathbf{x}(k) = (\mathbf{I} - \mathbf{B}\mathbf{K}_p)\,\mathbf{x}(k-1) + \mathbf{B}\mathbf{K}_p\,\mathbf{z}. \tag{16}$$

The closed-loop state transition matrix is $\mathbf{F} = \mathbf{I} - \mathbf{B}\mathbf{K}_p$. As long as the proportional gain matrix $\mathbf{K}_p$ is chosen such that the eigenvalues of $\mathbf{I} - \mathbf{B}\mathbf{K}_p$ lie within the unit circle, the closed-loop system remains stable. In practice, this implies that the relative response times of all tasks converge to their respective set points without sustained oscillations or divergence, ensuring robust and bounded behavior even in the presence of modeling inaccuracies and runtime variations, without requiring an exact system model. Due to space limitations, the stability range can be derived by following the standard steps listed in [26, 35, 49].

5 Experimental Setup

Hardware Testbed. Our experimental platform consists of an Nvidia RTX 3090 GPU running Ubuntu 20.04 LTS. We use CUDA Toolkit 12.0. The `DySM` controller is implemented in C++, with a control period of 1 second. The control period is chosen to include multiple periods of each task, so that multiple jobs can be completed for computing their average response time that eliminates any outliers.

SM Actuator. The *SM Actuator* manages SM scaling by adjusting the number of streaming multiprocessors (SMs) assigned to each task. We use the `libsmctrl` library for TPC control. The `nvdebug` kernel module [9] is loaded, and `libsmctrl` functions are invoked within the actuator. To enable this functionality, all tasks are launched from the same process. Specifically, we call `libsmctrl_set_stream_mask()` to assign TPCs. Because SMs cannot be adjusted directly, we instead adjust the number of TPCs, each of which contains two SMs. The target TPC values are periodically produced by the `DySM` MIMO controller and may be non-integer. Therefore, the modulator must locally map the controller's output to a sequence of supported integer TPC values as discussed in Section 4.2.

Partitioning policy. `DySM` uses spatial partitioning to allocate disjoint TPC/SM sets whenever feasible. If the sum of SMs assigned to all tasks exceeds the global budget, `DySM` iteratively removes the excess SMs during actuation from tasks whose RRT is below their set point (i.e., tasks with no deadline misses), while respecting each task's minimum SM allocation. This ensures that the total allocation fits within the global budget and preserves resources for tasks that miss their deadlines. If the requested allocations still cannot be realized simultaneously (e.g., due to TPC granularity, tight capacity, or transient controller demand), the actuator falls back to a controlled-sharing mode that allows limited overlap subject to a per-TPC sharing cap. This design ensures forward progress under overload while prioritizing isolation whenever possible.

Baselines. We compare our proposed approach against three baselines for scheduling tasks in spatial sharing. (1) `STGM` is a state-of-the-art *open-loop* solution that operates without any online control, following the static GPU management approach in [42]. The number of SMs assigned to each task is determined at compile time, and schedulability analysis is performed offline. (2) `SMshare` is another *open-loop* solution that assigns SMs statically at the start of execution, typically using an even split [2]. We compare with `STGM` and `SMshare` to show that open-loop solutions *cannot* adapt to online workload variations. (3) `Adhoc` represents an intuitive closed-loop way to do SM scaling by increasing/decreasing the number of SMs of a task by a fixed step size (instead of using control theory), if its measured response time is above/below the set point. This strategy has been adopted in some closed-loop GPU management approaches [3], though not designed directly for real-time systems or SM scaling. We refer to this strategy as `Adhoc`, where *each task has a separate controller* to adjust its SM count with no regard to other tasks. We compare with `Adhoc` to show the advantages of our MIMO control-theoretic design.

Workloads. We evaluate our approach using workloads drawn from the Nvidia CUDA Samples and the Rodinia Benchmark Suite. These benchmarks are widely used in the evaluation of many real-time systems [39,55–57]. From the Nvidia samples, we use `MatrixMul`, which performs matrix multiplication; `Stereodisparity`, which computes a stereo disparity

map; `Stencil`, which applies a stencil computation over a grid; and `Projection`, which implements high-quality compression. From the Rodinia suite, we use `Histogram`, a memory-intensive histogram computation; `Hotspot`, which estimates processor temperature using a stacked-layer packaging model; and `Particle Filter`, which implements a particle-filtering algorithm on the GPU. In addition, we use the `MTTKRP` kernel [47] to evaluate tensor computations.

Metrics. We use two metrics to compare `DySM` with the baselines. The first metric is `RRT`. The second metric is the `Deadline misses` for each control period. When the `RRT` exceeds 1, it indicates that a task's response time exceeds its deadline, which is also equal to its period. To calculate the `Deadline misses`, we monitor jobs whose response times surpass the task's period and classify these instances as deadline violations. This evaluation is repeated for each control period to determine the frequency of such violations. Experiments define acceptable performance as long as the `RRT` remains within ± 0.02 of the specified set point.

6 Experiments

In our evaluation, we try to answer the following questions:

- How does `DySM` compare against existing baseline approaches, when there are online workload changes? (Section 6.1)
- Why does TPC assignment across concurrent GPU tasks require an integrated (MIMO) control strategy? (Section 6.2)
- How effectively does `DySM` track different relative response time set points while avoiding deadline misses? (Section 6.3)
- What runtime overheads are introduced by the proposed framework components (i.e., monitoring, control, and actuation)? (Section 6.4)

6.1 Comparison with Baselines under Runtime Workload Variations

Real-time systems often experience workload fluctuations due to dynamic changes in the surrounding environment [22]. For example, in autonomous driving, an intersection with many traffic lights/signs and surrounding vehicles can significantly increase computational demand [23]. In this experiment, we emulate such real-world workload variations to compare `DySM` with the three baselines introduced in Section 5. We choose two periodic tasks `mm` ($29ms$, $45ms$) and `stereodisparity` ($2.4ms$, $21ms$). At the control period 80, we increase the input to `mm`. When the GPU's computational demands increase dynamically, the controller needs to play a crucial role in adjusting the per-task SM/TPC allocation to maintain consistent real-time performance. We discuss the result of each scheduling solution below.

1) `SMshare` is an *open-loop* baseline that assigns an equal number of SMs to all tasks and does not adapt allocations at runtime [2]. Specifically, if a system has N tasks and K SMs, each task is assigned $\frac{K}{N}$ SMs, and all K SMs are fully utilized under this equal partitioning. In this example, since there are two tasks `SMshare` assigns equal SMs to both tasks as shown in Figure 3(c). As shown in Figure 3(a), we observe that the `RRT` remains below 1 until control period 80. Moreover, the `RRT` is around 0.4, indicating that the system is over-provisioned (i.e., tasks complete well before their deadlines, leaving GPU resources unused). However, at control period 80, when the input to `mm` increases, the `RRT` exceeds 1, as shown in Figure 3(a), resulting in deadline misses. Because `SMshare` uses a fixed allocation and does not perform runtime adaptation, it cannot adjust the number

of SMs in response to changing workload conditions. Consequently, it fails to maintain deadlines under increased load, leading to the deadline misses observed in Figure 3(b). We next compare it with `STGM`.

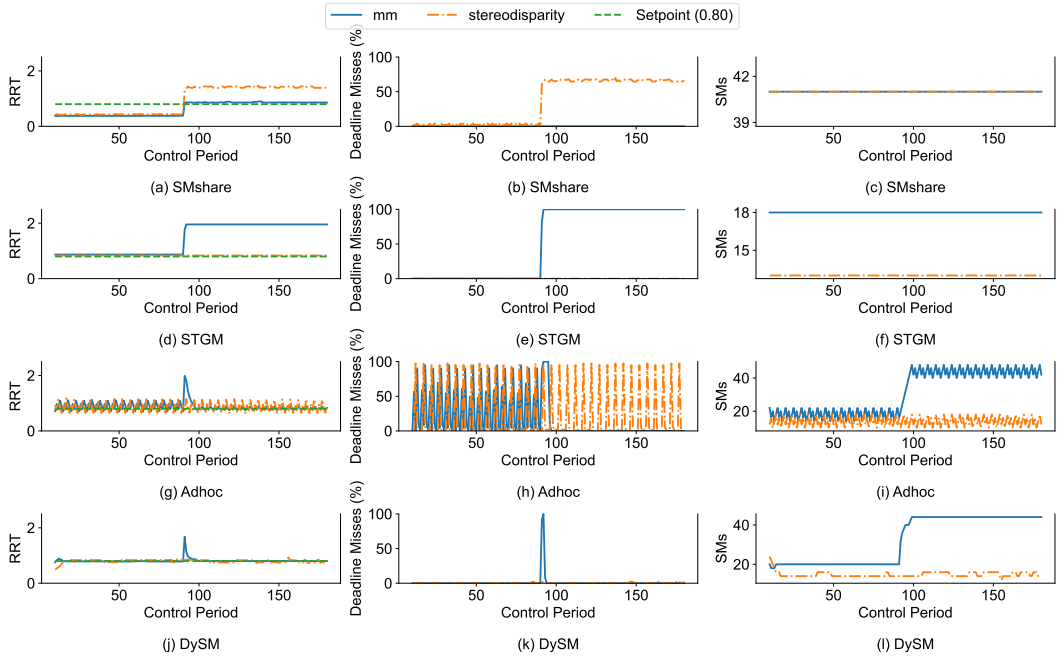


Figure 3 Comparison of relative response time (`RRT`), deadline misses, and number of SMs under different solutions. Figures (a), (b), and (c) are the results of SMshare. Figures (d), (e), and (f) are STGM. Both SMshare and STGM are state-of-the-art open-loop solutions without runtime adaptation; consequently, when the workload increases at control period 80, both approaches experience deadline misses. Figures (g), (h), and (i) show the results for Adhoc, which exhibits significant oscillations around the target set point. Figures (j), (k), and (l) are the results of DySM. During the control period 80, DySM adapts to runtime variation by dynamically adjusting the number of allocated SMs, thereby reducing deadline misses.

2) `STGM` is also a state-of-the-art *open-loop* baseline. We compute the number of SMs required for each task using the algorithm proposed in [42], based on the execution times of the tasks. Following this method, `STGM` allocates 18 SMs to `mm` and 13 SMs to `stereodisparity` as shown in Figure 3(f), ensuring that both tasks meet their deadlines. As shown in Figure 3(d), `STGM` maintains a `RRT` less than 1, resulting in zero deadline misses. Compared to `SMshare`, `STGM` assigns fewer SMs overall (i.e., it avoids equal partitioning across all available SMs) and instead provisions only the minimum number required to satisfy timing constraints. Consequently, the task response times are closer to their respective periods, ensuring tighter resource provisioning with reduced slack. However, because `STGM` does not adapt SM allocations at runtime, it cannot respond to workload increases. After control period 80, when the input to `mm` increases, the `RRT` exceeds 1, leading to 100% deadline misses, as shown in Figure 3(e). The results of `SMShare` and `STGM` demonstrate that open-loop GPU scheduling solutions cannot adapt to online workload variations. Thus, closed-loop adaptive solutions must be designed.
3) `Adhoc` is a heuristic-based closed-loop solution, as introduced before. In `Adhoc`, each task is managed by a separate controller, so there are N independent controllers for N tasks. At every control period, if the observed `RRT` is below the set point, the controller

of that task decreases the number of SMs allocated to the task by a fixed step size; if the RRT exceeds the set point, it increases the allocation by the same step size as shown in Figure 3(i). Because each task has its own independent Adhoc controller, actuation decisions are made locally without global coordination. As a result, when adjusting SM allocations, the controllers do not account for overlap in SM partitions across tasks. Consequently, multiple tasks may be assigned overlapping SM sets, leading to contention and unpredictable interference. While the Adhoc controller is simple to implement, its reliance on a fixed step size introduces inherent limitations. If the step size is too large (e.g., 4 TPCs or 8 SMs out of the total available SMs), the controller may overreact to small deviations in RRT, leading to oscillations around the set point. If it is too small (e.g., 1 TPC), convergence becomes slow and may fail to respond promptly to workload changes. Without control theory as a foundation, determining an appropriate step size typically requires extensive *offline profiling* for each task configuration and GPU platform. In this work, we have experimentally evaluated different step sizes and selected 5 SMs because it provides a good trade-off between convergence speed and oscillation. However, because the step size is fixed and not derived from system dynamics, it can remain suboptimal when workload characteristics or hardware behavior change, leading to performance variability. As shown in Figure 3(g), because the step size is mainly tuned to make mm have faster response when its input increases at control period 80, and then have small oscillation after 100, this step size causes stereodisparity to have much larger oscillations. As a result, Figure 3(h) shows that such oscillations lead to significant deadline misses for stereodisparity. A key limitation of solutions like Adhoc is that it is commonly hard to find a step size that works well in every case. In addition, the SM/TPC overlap caused by per-task independent controllers can also cause Adhoc to have longer response times. We analyze this reason in more detail in the next subsection.

4) DySM adapts promptly to changes in workload at runtime and maintains the RRT close to the desired set point, as shown in Figure 3(j). The precise control of DySM is mainly due to 1) its control-theoretic design provides the right (fractional) SM allocation, and 2) the delta-sigma modulator to approximate this fractional allocation, as discussed in Section 4.2. At control period 80, when the workload increases, DySM adjusts the number of SMs using the MIMO controller described in Section 4. During actuation, DySM also prevents SM/TPC overlap through coordinated control decisions, which results in fewer deadline misses. As shown in Figure 3(k), the deadline miss rate remains below 1%. DySM achieves this by dynamic SM scaling as shown in Figure 3(l) at runtime while preserving isolation among tasks. The overhead of dynamic SM scaling is minimal, as discussed in subsection 6.4.

6.2 Advantage of TPC-aware MIMO Control

DySM outperforms Adhoc for two major reasons. First, Adhoc relies on a fixed step size while DySM reallocates SMs based on how far the measured response times deviate from their set points, with control theory as a foundation. Second, DySM is a MIMO controller that integrates the constraint on total available SMs and TPCs in its model, but Adhoc uses a set of independent controllers that make isolated decisions, which can result in overlapping TPCs among tasks and thus GPU resource contention and longer response times.

In this experiment, we examine how overlapping TPC allocations across concurrent tasks can impact task response times. To that end, we run two periodic GPU tasks implemented as tensor-computation kernels (dense 3-way tensor operations). Each task is released every 30 ms from a dedicated CPU thread and launched on its own CUDA stream. Task 1 is fixed

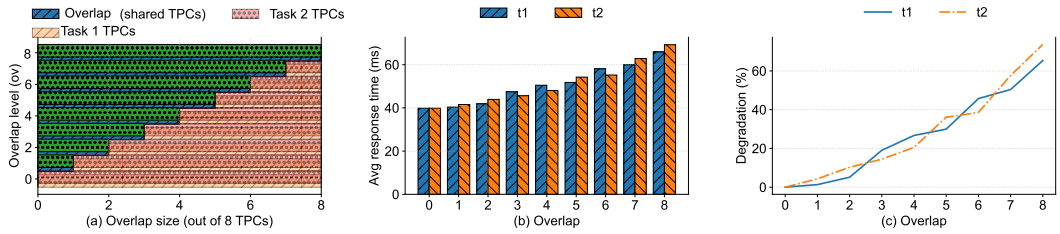


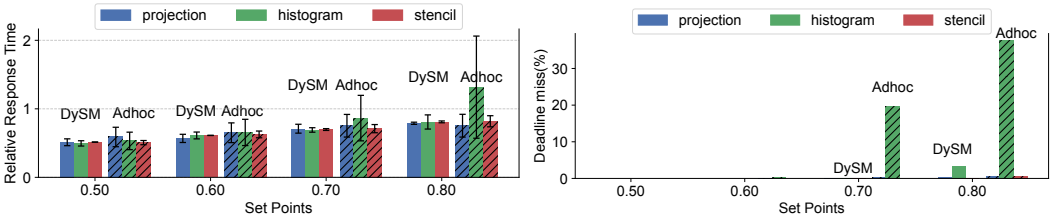
Figure 4 Impact of increasing TPC overlap between two concurrent GPU tasks. (a) Overlap pattern as Task 2 slides left while Task 1 remains fixed on TPCs $0-7$, increasing the number of shared TPCs from 0 to 8.(b) Average response time of both tasks as overlap increases. (c) Percentage degradation relative to the disjoint baseline (ov=0). As the number of shared TPCs increases, both tasks experience steadily rising response times, demonstrating the impact of compute contention under overlapping partition assignments.

to TPCs 0–7, while Task 2 initially executes on the disjoint range 8–15. We then gradually slide Task 2's allocation leftward, increasing the number of shared TPCs from 0 to 8. At full overlap, both tasks are restricted to the identical set 0–7. For each configuration, we execute 10 releases and report the average response time.

Figure 4(a) illustrates this progressive overlap pattern, and Figures 4(b)–(c) show the corresponding response times and degradation trends. Performance degradation is computed relative to the disjoint baseline ($ov = 0$), where the two tasks execute on non-overlapping TPC sets. Let $RT_i(0)$ denote the average response time of task i measured in this disjoint configuration. The percentage degradation for task i at overlap level ov is defined as $\text{Degradation}_i(\%) = \frac{RT_i(ov) - RT_i(0)}{RT_i(0)} \times 100$. As overlap increases, the response times of both tasks rise steadily. In the disjoint configuration ($ov = 0$), both tasks exhibit nearly identical response times of approximately 39.8 ms. When the overlap reaches its maximum ($ov = 8$), the response time increases to 65.9 ms for Task 1 and 69.2 ms for Task 2, corresponding to degradations of approximately 65% and 74%, respectively. Even modest overlap levels introduce noticeable slowdowns, and the degradation grows nonlinearly as the number of shared TPCs increases. This degradation occurs because overlapping TPC allocations introduce direct compute contention. When both kernels are eligible to execute on the same TPCs, the GPU hardware scheduler must multiplex thread blocks from both streams onto the shared compute partitions. Since CUDA kernels execute non-preemptively at the block level, blocks from competing kernels contend for the same SMs within overlapping TPCs, leading to increased queueing delays and extended kernel completion times. Similar effects have been observed in studies of hardware compute partitioning, where overlapping partitions lead to unpredictable scheduling behavior and inflated execution times due to contention at the TPC level [8]. Overall, these results demonstrate that even modest overlap in TPC assignments can substantially impact task response times, which is partially the reason for `Adhoc`'s inferior real-time performance.

6.3 Comparison under Different Set Points

Since open-loop solutions cannot handle runtime workload variations as discussed in Section 6.1, in this set of experiments, we evaluate the two closed-loop solutions, `DySM` and `Adhoc`, under set points ranging from 0.50 to 0.80. The task set consists of three tasks: `projection` (5.53ms, 39ms), `histogram` (4.5ms, 22ms), and `stencil` (5ms, 21ms). We measure the average `RRT` and standard deviation after the system reaches a steady state to mitigate the influence of any transient states at the beginning of each run.



(a) Relative Response Time. **(b)** Deadline misses (%).

Figure 5 Comparison between `Adhoc` and `DySM` under different set points.

Figure 5a shows that `Adhoc` cannot converge the tasks to their respective set points in many cases. This is because each task has its own controller that adjusts the number of SMs using a *step size* chosen offline. In addition, the overlapping TPCs caused by `Adhoc` increase task response times. As a result, `Adhoc` cannot precisely converge to the right set points. In contrast, `DySM` drives the `RRTs` of all tasks much closer to their specified set points, as shown in Figure 5a with significantly smaller deviations. Figure 5b shows the corresponding deadline misses for `DySM` and `Adhoc`. As the set point increases, response times increase closer to their deadlines, leading to a small number of deadline misses for both solutions, but `DySM` has much lower miss ratios than `Adhoc` due to its better control of response times. For example, at set points 0.70 and 0.80, the `Adhoc` deadline miss ratio for `histogram` is 19.73% and 37.6%. For `DySM`, the deadline miss ratio is just 0.12% and 3.41%, respectively. Hence, at set point 0.80, `DySM` reduces deadline misses by 90.93% relative to `Adhoc`. Compared to other tasks, `histogram` has a relatively higher deadline miss ratio because its higher runtime fluctuations, as shown in Figure 5a.

6.4 Overhead and Effectiveness of Dynamic SM Scaling

The `DySM` service necessarily introduces overhead. This overhead includes response-time monitoring, control computation, and enabling the TPC masks. `DySM` is viable only if this overhead remains sufficiently small relative to kernel execution time. To accurately measure overhead, we instrument the control service with timestamps at the entry and exit of each stage and report the corresponding execution times. As shown in prior studies [49], the overhead of the proportional MIMO controller varies linearly with the number of tasks; for a system with 10 tasks, the overhead is less than $20\mu s$, and for the response time monitor, it is less than $2\mu s$. Hence, here we focus mainly on the overhead of dynamic SM scaling.

We quantify the overhead of dynamically updating the set of active TPCs/SMs at runtime using `libsmctrl`, and verify that such updates correctly constrain which SMs execute subsequent work. Since these CUDA kernels are not preemptible, a mask update cannot affect a kernel that is already executing; instead, it only influences kernels launched after the update. In our system, each kernel execution lasts on the order of a few milliseconds, while the control period is 1 second. Each task releases one job per control interval, and its execution completes well before the next control decision is applied. Once a kernel corresponding to a task instance is launched in a stream with a fixed TPC mask, it executes non-preemptively on the assigned SMs until completion [24, 46]. The assigned partition configuration remains unchanged during kernel execution. SM reassignment is deferred to the subsequent job release and is implemented by updating the TPC mask before launching the next kernel instance. Since kernel execution time (milliseconds) is much shorter than the

control period (1 second), mask updates naturally take effect at job boundaries. As discussed in Section 2.2, the reconfiguration incurs negligible overhead, as it only modifies mask fields in the kernel launch metadata on the CPU side and does not require reinitialization.

Accordingly, our evaluation focuses on (i) the CPU-side overhead of issuing a mask update and (ii) the minimal delay required to observe its effect at the kernel dispatch boundary. We evaluate mask enforcement using a lightweight verification kernel. Each thread block reads its SM identifier (`%smid`) once and atomically increments a per-SM counter in global memory. The kernel has minimal computation, so that the measured latency is dominated by the mask update and kernel dispatch mechanisms rather than application-level computation. We launch at least 4096 blocks to ensure sufficient parallelism to use all enabled SMs. After kernel completion, the per-SM counters are copied back to the host, and the number of active SMs is determined by counting non-zero entries. We define a *flip index* as a control step at which the active SM mask is changed. At each flip index, `DySM` constructs a new bitmask of length N_{SM}, where N_{SM} is the total number of SMs on the GPU (e.g., 82 SMs on the Nvidia RTX3090). A bit value of 0 enables the corresponding SM for subsequent kernel launches, while a value of 1 disables it. The mask is applied via `libsmctrl` prior to kernel dispatch. By varying the mask at successive flip indices, we explicitly control the number of SMs available to the workload and observe the resulting dispatch behavior.

In this experiment, we alternate between two configurations at successive flip indices. Specifically, the even-numbered flip indices activate 60 SMs, while the odd-numbered flip indices activate 20 SMs. By switching between these two configurations before each new kernel launch, we can explicitly control the number of SMs available to the workload and observe the resulting dispatch behavior. Figure 6 summarizes the overhead and effectiveness of runtime SM mask updates. Figure 6(a) shows the CPU-side latency of issuing a TPC/SM mask update via `libsmctrl` across successive flip indices. The update cost remains consistently in the sub-nanosecond range, demonstrating that mask reconfiguration introduces negligible control overhead. Minor variation across flip indices is attributable to normal system noise rather than the mask operation itself. Figure 6(b) reports the kernel execution latency required to observe the effect of a mask update at the next dispatch boundary. As expected, the update takes effect only for kernels launched after the flip, and the observed latency remains stable across flips, indicating that the mask update does not introduce additional scheduling or dispatch delays beyond the unavoidable kernel launch boundary. Figure 6(c) validates correct enforcement of the SM mask. The number of SMs that execute at least one thread block closely matches the requested allocation at each flip index, confirming that `libsmctrl` reliably constrains kernel execution to the enabled subset of SMs. Together, these results demonstrate that runtime SM masking can be applied with negligible overhead while providing precise control over SM-level resource allocation.

7 Related Work

The related work of this paper can be categorized into two categories: 1) Real-time scheduling in spatial shared GPU systems and 2) feedback control for computer systems.

Real-time scheduling in spatial shared GPU systems. Spatial multitasking, also known as spatial resource sharing or GPU partitioning, emphasizes that multiple tasks can execute simultaneously on different subsets of computing units within the GPU [2, 12, 21, 39, 46, 48, 50, 56]. Adriaens et al. [2] present one of the earliest studies of spatial multitasking on GPUs, in which streaming multiprocessors (SMs) are partitioned among applications to enable concurrent kernel execution. Their approach assigns each application a fixed subset of

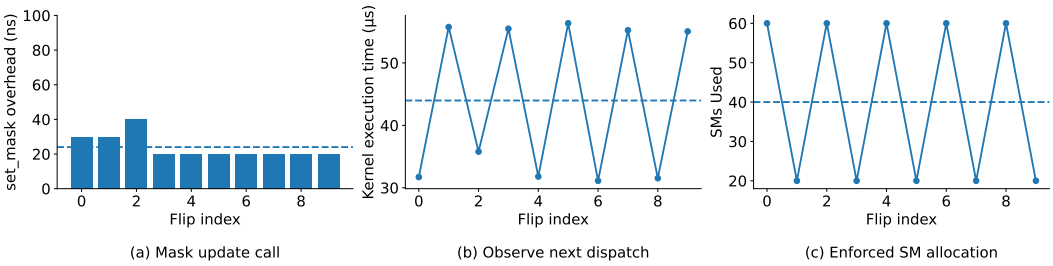


Figure 6 Overhead and effectiveness of runtime SM mask updates. (a) CPU-side execution time for issuing a TPC/SM mask update using `libsmctrl` across successive flip indices. (b) Kernel execution latency observed as the effect of a mask update at the dispatch boundary. (c) Number of SMs used by the verification kernel after each mask flip, confirming that the enforced SM allocation matches the requested mask.

SMs, and schedules the application's thread blocks exclusively within its assigned partition using static and profile-guided partitioning heuristics. Aguilera et al. [3] introduce a runtime mechanism for dynamically reallocating SMs based on application performance; their notion of QoS is defined solely in terms of average IPC and does not incorporate any real-time semantics. In particular, their approach does not model task deadlines, response times, or deadline misses, and transient QoS violations are considered acceptable, whereas our proposed solution explicitly regulates `relative response time` with respect to deadlines and significantly reduces deadline misses under runtime variability. Saha et al. [42] have proposed `STGM`, an offline algorithm to combine spatial partitioning and temporal scheduling to improve task-set schedulability under the Rate Monotonic (RM) policy. For `Nvidia GPUs`, Bakita and Anderson [8] have studied `libsmctrl`, a user-space library that enables transparent, hardware-based spatial partitioning of GPU compute resources by controlling which TPCs/SMs a kernel may execute on. However, all the aforementioned studies focus only on open-loop schedulability analysis, which cannot adapt to runtime workload variations commonly faced by GPU-based real-time systems.

Feedback control. Feedback control theory serves as a theoretical framework for designing and analyzing adaptive systems and has been widely utilized in computer performance management [26]. For instance, various control-theoretic techniques have been developed to regulate CPU temperature and power consumption [36, 44, 54]. There are projects that have implemented control theory in real-time scheduling and applications. One notable example is the feedback-based scheduler created by Steere et al. [45], which ensures that real-time applications achieve their desired progress rates. Additionally, Abeni et al. have provided a control analysis of a reservation-based feedback scheduler [1]. Another line of related work is Feedback Control Scheduling [32, 33], which adjusts the invocation rates of periodic real-time tasks to manage either CPU utilization or task deadline miss ratio. This methodology has been extended to distributed real-time systems [34, 35, 53]. Furthermore, feedback control has been applied to various real-time applications, including autonomous driving [6] [7], information dissemination [13, 15], and smart grid management [14], as well as energy efficiency [10, 18–20, 51]. Recently, feedback control has been applied to real-time GPU scheduling under time-slicing [49]. However, existing research relies exclusively on task-rate adaptation as its control knob and does not adjust the number of SMs allocated to each task, which is a critical limitation in spatially shared GPUs. Hence, rate adaptation alone cannot fully exploit available GPU parallelism nor provide predictable isolation among concurrently executing tasks.

8 Conclusion

Existing real-time GPU scheduling solutions for spatial sharing are mostly *open-loop* and rely on WCET estimation. Prior feedback control solutions for GPUs work only for time-slicing GPUs, and so cannot fully exploit spatial sharing for higher GPU resource utilization and better real-time performance. In this paper, we have presented `DySM`, the first closed-loop response time control algorithm for spatially shared GPU systems. `DySM` leverages dynamic SM scaling to regulate task response times with low runtime overhead while preserving spatial isolation. To model GPU resource contention among tasks, we analytically derive a multi-input-multi-output (MIMO) system model that captures the impact of SM scaling on the response times of different tasks. Based on this model, `DySM` is designed using feedback control theory for guaranteed system stability and control accuracy. Experimental results on an Nvidia GPU testbed demonstrate that `DySM` outperforms state-of-the-art solutions by providing runtime real-time guarantees. Compared to the best-performing baseline (`Adhoc`), `DySM` can reduce the deadline miss ratio by up to 90.93% (Section 6.3).

References

1 Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. Analysis of a reservation-based feedback scheduler. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, December 2002.

2 Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. The case for gpgpu spatial multitasking. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.

3 Paula Aguilera, Katherine Morrow, and Nam Sung Kim. Qos-aware dynamic resource allocation for spatial-multitasking gpus. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 726–731. IEEE, 2014. doi:10.1109/ASPDAC.2014.6742976.

4 Nusaybah M Alahdal, Felwa Abukhodair, Leila Haj Meftah, and Asma Cherif. Real-time object detection in autonomous vehicles with yolo. *Procedia Computer Science*, 246:2792–2801, 2024. doi:10.1016/J.PROCS.2024.09.392.

5 Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115. IEEE, 2017. doi:10.1109/RTSS.2017.00017.

6 Yunhao Bai, Li Li, Zejiang Wang, Xiaorui Wang, and Junmin Wang. Performance optimization of autonomous driving control under end-to-end deadlines. *Real-Time Systems*, 58(4):509–547, 2022. doi:10.1007/S11241-022-09379-6.

7 Yunhao Bai, Zejiang Wang, Xiaorui Wang, and Junmin Wang. Autoe2e: End-to-end real-time middleware for autonomous driving control. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1101–1111. IEEE, 2020. doi:10.1109/ICDCS47774.2020.00092.

8 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. IEEE, 2023. doi:10.1109/RTAS58335.2023.00012.

9 Joshua Bakita and James H Anderson. Demystifying nvidia gpu internals to enable reliable gpu management. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 294–305. IEEE, 2024. doi:10.1109/RTAS61025.2024.00031.

10 Can Basaran, Mehmet H. Suzer, Kyoung-Don Kang, and Xue Liu. Robust fuzzy cpu utilization control for dynamic workloads. *Journal of Systems and Software*, 83(7):1192–1204, 2010. doi:10.1016/J.JSS.2010.01.031.

11 Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for gpu with preemption support. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 119–130. IEEE, 2018. doi:10.1109/RTSS.2018.00021.

12 Guoyu Chen, Srinivasan Subramaniyan, and Xiaorui Wang. Latency-guaranteed co-location of inference and training for reducing data center expenses. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, pages 473–484. IEEE, 2024. doi:10.1109/ICDCS60910.2024.00051.

13 Ming Chen, Xiaorui Wang, Raghul Gunasekaran, Hairong Qi, and Mallikarjun Shankar. Control-based real-time metadata matching for information dissemination. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

14 Ming Chen, Xiaorui Wang, and Clinton Nolan. Hierarchical cpu utilization control for real-time guarantees in power grid computing. *Journal of Real-Time Systems*, 48(2):198–221, January 2012. doi:10.1007/S11241-011-9141-X.

15 Ming Chen, Xiaorui Wang, and Ben Taylor. Integrated control of matching delay and cpu utilization in information dissemination systems. In *Proceedings of the 17th IEEE International Workshop on Quality of Service (IWQoS)*, 2009.

16 NVIDIA Corporation. *Multi-Process Service*, n.d. Available at https://docs.nvidia.com/deploy/mps/index.html. URL: https://docs.nvidia.com/deploy/mps/index.html.

17 Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 492–506, 2020.

18 Xing Fu, Khairul Kabir, and Xiaorui Wang. Cache-aware utilization control for energy efficiency in multi-core real-time systems. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, 2011.

19 Xing Fu and Xiaorui Wang. Utilization-controlled task consolidation for power optimization in multi-core real-time systems. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2011.

20 Xing Fu, Xiaorui Wang, and Eric Puster. Dynamic thermal and timeliness guarantees for distributed real-time embedded systems. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2009.

21 Guin Gilman and Robert J Walls. Characterizing concurrency mechanisms for nvidia gpus under deep learning workloads. *ACM SIGMETRICS Performance Evaluation Review*, 49(3):32–34, 2022.

22 Ionel Gog, Sukrit Kalra, Peter Schafhalter, Joseph E Gonzalez, and Ion Stoica. D3: a dynamic deadline-driven approach for building autonomous vehicles. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 453–471, 2022. doi:10.1145/3492321.3519576.

23 Ionel Gog, Sukrit Kalra, Peter Schafhalter, Joseph E. Gonzalez, and Ion Stoica. D3: A dynamic deadline-driven approach for building autonomous vehicles. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, 2022.

24 Lixiang Han, Zimu Zhou, and Zhenjiang Li. Pantheon: Preemptible multi-dnn inference on mobile edge gpus. In *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*, pages 465–478, 2024. doi:10.1145/3643832.3661878.

25 Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022. URL: https://www.usenix.org/conference/osdi22/presentation/han.

26 Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M Tilbury. *Feedback control of computing systems*. John Wiley & Sons, 2004.

27 Seyedmehdi Hosseinimotlagh and Hyoseung Kim. Thermal-aware servers for real-time tasks on multi-core gpu-integrated embedded systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

28 Woosung Kang, Kilho Lee, Jinkyu Lee, Insik Shin, and Hoon Sung Chwa. Lalarand: Flexible layer-by-layer cpu/gpu scheduling for real-time dnn tasks. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 329–341. IEEE, 2021.

29 Hyoseung Kim, Pratyush Patel, Shige Wang, and Ragunathan Raj Rajkumar. A server-based approach for predictable gpu access control. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2017. doi:10.1109/RTCSA.2017.8046309.

30 Ze Li and Marco Brocanelli. An edge-based reinforcement learning approach for augmented reality apps in dynamic contexts. In *Proceedings of the Tenth ACM/IEEE Symposium on Edge Computing*, pages 1–16, 2025. doi:10.1145/3769102.3770627.

31 Ze Li, Niloofar Didar, and Marco Brocanelli. Offloading-aware control of ai task allocation and virtual object quality in mar apps. *IEEE Transactions on Mobile Computing*, 2025.

32 C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Journal of Real-Time Systems, Special Issue on Control-Theoretical Approaches to Real-Time Computing*, 23(1/2):85–126, July 2002. doi:10.1023/A:1015398403337.

33 Chenyang Lu, Xiaorui Wang, and Christopher Gill. Feedback control real-time scheduling in ORB middleware. In *Proceedings of 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2003.

34 Chenyang Lu, Xiaorui Wang, and Xenofon Koutsoukos. End-to-end utilization control in distributed real-time systems. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, 2004.

35 Chenyang Lu, Xiaorui Wang, and Xenofon Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE transactions on parallel and distributed systems*, 16(6):550–561, 2005. doi:10.1109/TPDS.2005.73.

36 Kai Ma, Xue Li, Ming Chen, and Xiaorui Wang. Scalable power control for many-core architectures running multi-threaded applications. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 449–460, 2011. doi:10.1145/2000064.2000117.

37 Yuan Ma, Srinivasan Subramaniyan, and Xiaorui Wang. Power capping of gpu servers for machine learning inference optimization. In *Proceedings of the 54th International Conference on Parallel Processing*, pages 449–459, 2025. doi:10.1145/3754598.3754670.

38 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.

39 Nathan Otterness, Ming Yang, Tanya Amert, James Anderson, and F Donelson Smith. Inferring the scheduling policies of an embedded cuda gpu. *OSPERT'17*, 2017.

40 Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H Anderson, F Donelson Smith, Alex Berg, and Shige Wang. An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 353–364. IEEE, 2017.

41 Pedram Amini Rad, Danny Hofmann, Sergio Andres Pertuz Mendez, and Diana Goehringer. Optimized deep learning object recognition for drones using embedded gpu. In *26th IEEE International Conference on Emerging Technologies and Factory Automation*, 2021.

42 Sujan Kumar Saha, Yecheng Xiang, and Hyoseung Kim. Stgm: Spatio-temporal gpu management for real-time tasks. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–6. IEEE, 2019. doi:10.1109/RTCSA.2019.8864564.

43 Lui Sha, Ragunathan Rajkumar, and Shirish S Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1):68–82, 1994. doi:10.1109/5.259427.

44 Kevin Skadron, Tarek Abdelzaher, and Mircea R. Stan. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*, 2002.

45 David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd Operating Systems Design and Implementation (OSDI)*, 1999.

46 Srinivasan Subramaniyan, Rudra Joshi, Xiaorui Wang, and Marco Brocanelli. Seeb-gpu: Early-exit aware scheduling and batching for edge gpu inference. In *Proceedings of the 2025 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2025.

47 Srinivasan Subramaniyan and Xiaorui Wang. Opticpd: Optimization for the canonical polyadic decomposition algorithm on gpus. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 403–412. IEEE, 2023. doi:10.1109/IPDPSW59300.2023.00071.

48 Srinivasan Subramaniyan and Xiaorui Wang. Exploiting ml task correlation in the minimization of capital expense for gpu data centers. In *2025 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2025. doi:10.1109/IPCCC66453.2025.11304653.

49 Srinivasan Subramaniyan and Xiaorui Wang. Fc-gpu: Feedback control gpu scheduling for real-time embedded systems. In *Proceedings of the ACM SIGBED International Conference on Embedded Software (EMSOFT 2025)*, Taipei, Taiwan, September 2025.

50 Srinivasan Subramaniyan and Xiaorui Wang. Cats: Correlation-aware task scheduling for gpu power optimization in ai data centers. In *Proceedings of the International Conference on Supercomputing (ICS '26)*, Belfast, United Kingdom, 2026. Association for Computing Machinery.

51 Xiaorui Wang, Xing Fu, Xue Liu, and Zonghua Gu. Power-aware cpu utilization control for distributed real-time systems. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2009.

52 Xiaorui Wang, Dong Jia, Chenyang Lu, and Xenofon Koutsoukos. Decentralized utilization control in distributed real-time systems. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10–pp. IEEE, 2005.

53 Xiaorui Wang, Dong Jia, Chenyang Lu, and Xenofon Koutsoukos. DEUCON: Decentralized end-to-end utilization control for distributed real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(7):996–1009, 2007. doi:10.1109/TPDS.2007.1051.

54 Yefu Wang, Kai Ma, and Xiaorui Wang. Temperature-constrained power control for chip multiprocessors with online model estimation. *ACM SIGARCH computer architecture news*, 37(3):314–324, 2009. doi:10.1145/1555754.1555794.

55 Yidi Wang, Mohsen Karimi, and Hyoseung Kim. Towards energy-efficient real-time scheduling of heterogeneous multi-gpu systems. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 409–421. IEEE, 2022. doi:10.1109/RTSS55097.2022.00042.

56 Yidi Wang, Mohsen Karimi, Yecheng Xiang, and Hyoseung Kim. Balancing energy efficiency and real-time performance in gpu scheduling. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 110–122. IEEE, 2021. doi:10.1109/RTSS52674.2021.00021.

57 Yidi Wang, Cong Liu, Daniel Wong, and Hyoseung Kim. Gcaps: Gpu context-aware preemptive priority-based scheduling for real-time tasks. In *36th Euromicro Conference on Real-Time Systems (ECRTS 2024)*, pages 14:1–14:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.ECRTS.2024.14.

58 Ming Yang. Avoiding pitfalls when using nvidia gpus for real-time tasks in autonomous systems. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, 2018.

59 Husheng Zhou, Guangmo Tong, and Cong Liu. Gpes: A preemptive execution system for gpgpu computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 87–97. IEEE, 2015. doi:10.1109/RTAS.2015.7108420.

60 An Zou, Jing Li, Christopher D Gill, and Xuan Zhang. Rtgpu: Real-time gpu scheduling of hard deadline parallel tasks with fine-grain utilization. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1450–1465, 2023. doi:10.1109/TPDS.2023.3235439.