FC-GPU: Feedback Control GPU Scheduling for Real-time Embedded Systems

SRINIVASAN SUBRAMANIYAN, The Ohio State University, USA
XIAORUI WANG, The Ohio State University, USA

GPUs have recently been adopted in many real-time embedded systems. However, existing GPU scheduling solutions are mostly open-loop and rely on the estimation of worst-case execution time (WCET). Although adaptive solutions, such as feedback control scheduling, have been previously proposed to handle this challenge for CPU-based real-time tasks, they cannot be directly applied to GPU, because GPUs have different and more complex architectures and so schedulable utilization bounds cannot apply to GPUs yet. In this paper, we propose FC-GPU, the first Feedback Control GPU scheduling framework for real-time embedded systems. To model the GPU resource contention among tasks, we analytically derive a multi-input-multi-output (MIMO) system model that captures the impacts of task rate adaptation on the response times of different tasks. Building on this model, we design a MIMO controller that dynamically adjusts task rates based on measured response times. Our extensive hardware testbed results on an Nvidia RTX 3090 GPU and an AMD MI-100 GPU demonstrate that FC-GPU can provide better real-time performance even when the task execution times significantly increase at runtime.

CCS Concepts: • **Computer systems organization → Embedded systems**; **Real-time system architecture**.

Additional Key Words and Phrases: GPU Scheduling, Real-Time, Embedded Systems, Feedback Control

1 Introduction

In recent years, GPUs have been increasingly adopted in a wide variety of real-time embedded systems. For example, autonomous driving system (ADS) needs to run advanced machine learning (ML) algorithms for real-time object detection and path planning on on-board GPUs, in order to detect surrounding objects and plan accordingly for the car's near-future trajectory. Another example is immersive virtual reality (VR) headset, which often must finish the real-time processing and rendering of 3D videos, on its embedded GPUs, to avoid undesired delay during the interaction with the VR application (e.g., game or movie). Therefore, how to schedule real-time tasks on GPUs has become increasingly important for embedded systems and has recently received a lot of research attention (e.g., [3, 43, 48, 50, 52, 54, 59]). For example, in an ADS system, real-time GPU scheduling can coordinate various tasks, such as front and rear camera image recognition and path planning, so that all those tasks can meet their real-time deadlines [19, 25]. Similarly, embedded GPUs are used for real-time object detection in drone-based systems, allowing drones to recognize obstacles and adjust their flight path [60].

To date, most existing research on real-time GPU scheduling focuses on *open-loop* solutions, where the execution times of real-time tasks need to be carefully estimated, as the worst-case execution times (WCETs), in an offline manner to ensure runtime guarantees of timeliness. However, for many real-time systems, the task execution times can vary significantly on both CPUs and GPUs and so can be difficult to estimate precisely. Hence, any underestimation of WCETs can lead to undesired deadline misses, while overestimation can result in resource waste and higher system costs. For example, for CPU-based ADS tasks, the execution time of path tracking can vary significantly based on road conditions. For GPU-based tasks, the traffic light detector of Apollo [1], a well-known open autonomous driving platform, can have a 330% execution time increase depending on the number of detected traffic lights [2, 62]. Although it is indeed possible to overestimate the

Authors' Contact Information: Srinivasan Subramaniyan, subramaniyan.4@osu.edu, The Ohio State University, Columbus, Ohio, USA; Xiaorui Wang, wang.3596@osu.edu, The Ohio State University, Columbus, Ohio, USA.

WCETs of real-time tasks in a conservative fashion for runtime schedulability guarantees, doing so can cause a significant increase of needed CPUs and GPUs, and so is usually not desirable to system designers.

Adaptive real-time scheduling (e.g., [8, 9]) has been previously designed to address this challenge by dynamically adjusting the workloads of selected tasks, when the execution time of some tasks increases at runtime. Such scheduling solutions keep monitoring the system schedulability online and make necessary workload adaptation in a closed-loop manner to quickly react to execution time variations. To this end, an effective closed-loop scheduling solution is called Feedback Control Scheduling (FCS) [4, 12], which dynamically controls either the CPU utilization or the task deadline miss ratio by adjusting the invocation rates of periodic tasks within their allowed ranges. For example, the goal of CPU utilization control is to enforce appropriate utilization bound (e.g., the Rate Monotonic Scheduling (RMS) bound [6]) on a CPU, such that the real-time deadlines of all the tasks on the CPU can be guaranteed. FCS has been proven to be effective in handling unexpected workload variations and so extended to control distributed real-time system [13, 17, 18] and optimize system utilities [22, 24, 29, 30, 34].

Unfortunately, despite their effectiveness in controlling CPU-based *soft* real-time systems, existing FCS solutions *cannot* be directly applied to GPUs for several reasons. First, GPU utilization cannot be used directly as a control objective for FCS utilization control because RMS utilization bounds cannot be applied to GPUs. Second, previous FCS solution also controls the task deadline miss ratio to some set point (e.g., 1%). However, a constant 1% deadline miss ratio is commonly undesirable, even for soft real-time systems. Thus, a new control objective must be identified for real-time GPU tasks.

In this paper, we propose FC-GPU, a Feedback Control GPU scheduling framework for *soft* real-time embedded systems. Different from previous FCS solutions that control either CPU utilization or deadline miss ratio, FC-GPU is designed to control the response time of every GPU task to be shorter than its respective deadline, by adapting the invocation rate of each task. To model the GPU resource contention among tasks, we analytically derive a multi-input-multi-output (MIMO) system model that captures the impacts of task rate adaptation on the response times of different tasks. Then, based on the proposed MIMO model, FC-GPU is rigorously designed based on feedback control theory to analytically guarantee the stability and control performance of the system.

Specifically, this paper makes the following contributions:

- Unlike prior work that either relies on open-loop GPU scheduling with WCET assumptions or uses feedback control scheduling (FCS) only for CPUs (e.g., controlling utilization or miss ratio), we present the first closed-loop FCS algorithm for GPUs. Our approach continuously monitors task response times at runtime and adaptively adjusts task rates within allowed ranges to ensure real-time guarantees.
- We analytically derive a MIMO system model that captures the impacts of task rate adaptation on the response times of different tasks. We then design our controller based on the MIMO model and feedback control theory to outperform a baseline solution that controls each task separately.
- We present hardware testbed results on an Nvidia RTX 3090 GPU and an AMD MI-100 GPU to demonstrate the efficacy of the proposed FC-GPU algorithm. Extensive results show that FC-GPU can provide better real-time performance with 2% fewer missed deadlines for GPU tasks, even when the task execution times increase significantly at runtime.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 provides background information. Section 4 describes the design of FC-GPU. Section 5 presents the testbed evaluation, and Section 6 concludes the paper.

2 Related Work

Due to space limitations, we discuss only the most closely related work on GPU real-time scheduling and feedback performance control of computer systems.

GPU real-time scheduling: In the past decade, various scheduling methodologies have been proposed to improve GPU utilization and performance, allowing multiple tasks to execute currently on a GPU [35, 42, 45, 53, 58]. There is also a considerable amount of work designed to enable efficient GPU task preemption, such as PipeSwitch [57], EffiSha [46] and REEF [64]. In contrast to those studies, our paper focuses on embedded systems whose GPU tasks must meet explicit real-time deadlines. Some GPU real-time scheduling algorithms have been previously proposed [28, 31, 37, 40]. For example, Kato et al. introduce a priority-based scheduler [33] that assigns temporal budgets based on task priorities and periodically replenishes them. GPU-TDMh [51] uses time division multiplexing to effectively schedule hard real-time sporadic tasks on a single GPU. SBEET and SBEET-mg have designed real-time GPU scheduling frameworks to optimize the energy efficiency of embedded systems [3][66].

While most of the existing GPU scheduling solutions operate in an *open-loop* manner to configure the GPU based on offline-profiled parameters, our work explicitly monitors the task response times and accordingly adjusts the GPU task rates in a closed-loop manner to meet real-time deadlines. In addition, our work features a control-theoretic design foundation for analytically guaranteed control performance, such as control accuracy and system stability, without exhaustive iterations of tuning and testing.

Feedback performance control: As a theoretical tool for designing and analyzing adaptive systems, feedback control theory has been extensively applied to computer performance management [14]. For example, several control-theoretic techniques have been developed to control the CPU temperature or power consumption [11, 27, 32]. There are some projects that have applied control theory to real-time scheduling and applications. For example, Steere et al. have developed a feedback-based scheduler [9] that guarantees desired progress rates for real-time applications. Abeni et al. have presented control analysis of a reservation-based feedback scheduler [10]. The most closely related work is Feedback Control Scheduling (FCS) [4, 12], which dynamically adjusts the invocation rates of periodic real-time tasks to control either the CPU utilization or the task deadline miss ratio. For distributed real-time systems, end-to-end CPU utilization control [13, 17] and decentralized utilization control algorithms [18] have also been designed, respectively. In addition, FCS has been applied to various real-time applications, such as autonomous driving [65][55], information dissemination [20, 23], and smart grid [38]. Furthermore, FCS is extended to optimize system utilities, such as energy efficiency [22, 24, 29, 30, 34].

However, all the aforementioned studies *focus on CPU*. To our best knowledge, this paper presents the first study of applying control theory to GPU real-time scheduling.

3 Background

Most of today's commodity GPUs are still not open-source, so different scheduling methods [46, 64] have been reported for those GPUs. To design effective GPU control algorithms for real-time guarantees, it is essential to thoroughly understand the underlying GPU scheduling mechanism. Since we utilize both Nvidia and AMD GPUs, we analyze their scheduling mechanisms.

Nvidia GPUs have three compute modes: (1) exclusive, (2) prohibited, and (3) default. In the exclusive mode, only one task can access the GPU resources [40]. This restriction can cause real-time tasks to miss deadlines if multiple tasks exist in different GPU contexts. In the prohibited mode, no context executes on the GPU. Therefore, we choose not to use the exclusive and prohibited modes. We mainly focus on the default mode, which uses time-slicing, because it is common in most GPUs. We set the compute mode to default by running `sudo nvidia-smi -c 0`.

In addition to setting it to the default mode, tasks must be launched from separate processes. This approach ensures that they are *multiprogrammed*, meaning they run concurrently using time-slicing [56] [47]. When tasks are launched from separate processes at the device-driver level, each process is associated with a GPU context, which represents a virtual address space and other runtime states on the GPU. Any process accessing the GPU has a separate GPU context, regardless of whether it uses the CUDA library. Contexts from different processes are time-sliced to share the GPU hardware.

By default, AMD GPUs use *spatial sharing*. We configure the Linux driver on AMD GPUs to run in single-process time-slicing mode to enable time-slicing in AMD GPUs. Specifically, we modify the `amdgpu` driver by setting the following module parameter `hws_max_conc_proc=1`. This change shifts the driver's default multiprocess sharing mode to a single-process mode, prioritizing one process at a time. After this modification, we recompile the driver to enable these changes.

When a GPU switches from one task to another, it initiates a context switch. Due to architectural differences between GPUs and CPUs, GPU time-slicing operates differently from CPU time-slicing. The primary sources of overhead during GPU context switching include saving large register files and cache flushing [41]. To reduce these overheads, GPUs often use partial state-saving techniques rather than saving the entire context. For example, GPUs retain resources like shared memory, registers, and global memory across time slices, thus minimizing unnecessary data transfers [61].

Besides computational tasks, GPUs handle data transfer operations by loading data from the main memory into GPU memory, typically using specialized copy engines. The GPU driver splits data transfers into smaller segments, and preemption occurs at the boundaries of these segments [52]. Beginning with the Pascal architecture, Nvidia GPUs utilize demand paging for memory management. This approach prevents GPU memory belonging to a context from being swapped out during a context switch, regardless of whether the GPU is integrated or discrete [69].

4 Design of FC-GPU

In this section, we first describe the task model and have an overview of FC-GPU. Then, we formulate the control problem, discuss GPU system modeling, and design the controller based on the model to solve the formulated problem. Finally, we provide the detailed steps for stability analysis and present some discussion on overheads and other practical issues.

4.1 Task Model

In real-time systems, a task can be implemented as a process or thread that needs to be scheduled to run on the target GPU during specific time intervals. Each task can have multiple instances, known as jobs. A job's release time is when the job becomes ready to execute on the GPU and denotes the time the system can begin to schedule and run this job. In this paper, to simplify the design, we assume all tasks are periodic. We consider periodic tasks because real-time tasks in most embedded systems commonly run periodically to interact with the environment, such as sensors or cameras [7]. We assume each task's deadline is equal to its period because if a job cannot finish within its period, the next job is released, and the current job becomes outdated. For instance, in autonomous driving systems, tasks such as sensor data processing, decision-making, and vehicle control must be conducted periodically to ensure safe and efficient operation [50].

We now introduce the notation used throughout the paper. A system consists of N tasks (t_i, $1 <= i <= N$), with each task t_i defined by the following parameters:

- e_i: Execution time for the task in GPU.
- $r_i(k)$: Task rate of t_i in the k^{th} sampling period.
- θ_i: Context switching overhead for t_i.

- $\widetilde{e_i} = e_i + \theta_i$: Effective execution time for the task in GPU.
- $p_i(k) = 1/r_i(k)$: Task period in the k^{th} period.
- $d_i(k)$: Task deadline of t_i.
- j_{ik}: The k^{th} job of t_i in the k^{th} period.
- h_{ik}: Release time of job j_{ik}.
- f_{ik}: Finish time of job j_{ik}.
- $q_i(k)$: Response time of j_{ik}, i.e., $f_{ik} - h_{ik}$.
- z_i: RTR set point for t_i.
- $rtr_i(k)$: Response Time Ratio (RTR) of j_{ik}.

Our task model accommodates a wide range of tasks, defined as $t(e_i, p_i)$, where e_i is the GPU execution time and p_i is the task period. The execution times of these tasks can be known in advance, as they can be estimated with benchmarks. However, the execution time of a task may vary at runtime due to different sensor inputs or the need for additional computation. These runtime variations necessitate a closed-loop solution, because open-loop scheduling based on estimated execution times cannot handle them effectively. The response time for each job j_{ik} includes its execution time and any delays due to preemption by other jobs on the same GPU. Since q_i is unknown at design time due to runtime GPU scheduling, we measure it for each job as its finish time minus its release time. The Response Time Ratio (RTR) measures a task's proximity to its deadline and is expressed as:

$$rtr_i(k) = \frac{q_i(k)}{p_i(k)} = q_i(k) \times r_i(k) \tag{1}$$

An RTR of 1 indicates that the job of the task in that period completes exactly by its deadline; if it exceeds 1, the job misses its deadline. For example, with an RTR set point of 0.9 and a period of 2 s, every job of the task needs to finish by 1.8 s. However, if a job runs slightly longer than 1.8 s, it can still meet the deadline of 2 s.

4.2 Overview of FC-GPU

In this section, we provide a high-level description of the system architecture of FC-GPU. Given a GPU that runs multiple real-time tasks, the design objective is to ensure the response time of every task be shorter than its respective deadline by controlling the RTR to the desired set point (e.g., 0.9).

There can be different ways to adapt the tasks for controlling their response times. In this paper, we adopt task rate adaptation because it is widely used in real-time systems [12][21][4][18][13].

In rate adaptation, we assume that the rate at which a task operates can be dynamically adjusted within a specified range, denoted as $[r_{\min}, r_{\max}]$. Previous research has demonstrated that task rates often fall within this adjustable range in various fields, including real-time multimedia applications [44] and autonomous vehicle systems [55]. By dynamically altering task rates, embedded systems may optimize resource utilization, meet real-time deadlines, and maintain system stability in various operating environments. Also, the initial rates may not give us the desired utilization required for the system and ensure minimalistic deadline misses [13].

A key feature of our task model is its ability to control the system in an unpredictable environment where the task's response time is unknown to the scheduler. To perform rate adaptation, the FC-GPU architecture features a multi-input-multi-output (MIMO) feedback control loop that dynamically adjusts task rates to ensure that the task RTRs converge to their specified set points, as shown in Figure 1. This MIMO controller is located on the host CPU. A periodic feedback loop is invoked at every sampling instant and comprises a Response Time Monitor, a Controller, and a Task Rate Actuator.

The process within each control period is as follows:

(1) The monitor measures the response time ratio $rtr_i(k)$ for each task during the last control period.
(2) The controller collects all the $rtr_i(k)$ values and the input task rates $r_i(k)$ and computes the proportional constant (Δ) that has to be added to the task rates.
(3) After receiving the updated (Δ) from the controller, the Task Rate Actuator adjusts the $r_i(k)$ for all periodic tasks accordingly.

In this system, task rates (r_i) act as manipulated variables, and RTR (rtr_i) is our controlled variable. In the subsequent subsections, we present the design of the MIMO controller.

4.3 Problem Formulation

Based on the task model stated in Section 4.1, the FC-GPU control problem can be formulated as a constrained optimization problem, as follows:

$$\min_x f(x), \quad \text{where } f(x) = \sum_{i=1}^{N} \left(z_i - rtr_i(k)\right)^2, \quad x = \left[r_1(k), r_2(k), \ldots, r_N(k)\right]. \tag{2}$$

subject to:

$$r_{\min} \leq r_i(k) \leq r_{\max} \tag{3}$$

The design goal is to minimize the difference between the set point z_i and the observed RTR ($rtr_i(k)$) by adjusting the rate of each task $r_i(k)$ within the feasible range $[r_{\min}, r_{\max}]$. Formally, the scheduling objective is an online optimization problem.

Given a proper RTR set point (e.g., 0.9), this formulation ensures that when $f(x)$ is minimized, the tasks meet their deadlines. Instead of solving this constrained optimization in a classical offline manner, we use a *control-theoretic* approach with a *closed-loop feedback mechanism* to solve it online. At each sampling interval, the controller measures the current state of the system (that is, the RTR of each task), computes the error relative to the set points z_i, and adjusts the rates $r_i(k)$ accordingly. Over time, these adjustments converge to a stable operating point that minimizes $f(x)$ within the allowable limits, ensuring that the response time of every task remains within its deadline. This control-based method is suitable for real-time GPU scheduling because it reacts immediately to changes and remains robust under workload fluctuations or varying execution times.

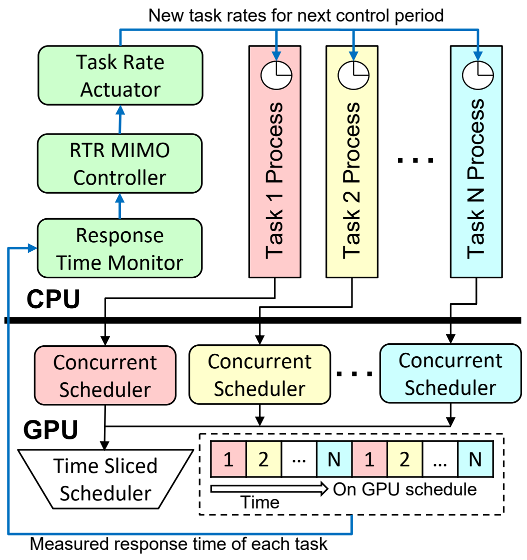


Fig. 1. Overall framework of the FC-GPU control loop that periodically monitors the response time ratio (RTR) of each task and accordingly adjusts the task rate to ensure that the task response times be shorter than their respective deadlines.

4.4 Modeling of GPU Real-time System

To effectively apply a control-theoretic methodology, we must first establish a dynamic model that characterizes the relationship between the manipulated variables $r_i(k)$ and the controlled variables

$rtr_i(k)$. Controlled variables are the system outputs we regulate or maintain at desired levels, such as RTR and system utilization. Manipulated variables are system attributes the controller can dynamically adjust to influence the controlled variables. Since $rtr_i(k) = q_i(k)/p_i(k)$, we first build a model between task rates $r_i(k)$ and the task response times $q_i(k)$.

To understand how the response time varies due to other concurrent tasks, we consider a simple workload (matrix multiplication) as an example [3]. The workload consists of a matrix multiplication kernel where each thread computes a single element of the output matrix by taking the dot product of a row from matrix A and a column from matrix B. The kernel is executed with a 2D grid and 2D thread blocks $(32, 32)$, where each block computes a 32×32 output tile. To optimize memory transfers, zero-copy pinned memory is used, allowing the GPU to directly access host memory. We launch a single matrix multiplication process and then launch two, three, and four processes of the same matrix multiplication task in parallel [49]. We measure their response time using the CUDA time stamper.

The variation in response time is shown in Figure 2. The response time follows a pattern proportional to $N \times e_i$, where e_i is the execution time for the task and N is the number of tasks. The observation is made from the following: When N tasks are present, the GPU switches between these N tasks. Assuming ideal time-slicing where the GPU perfectly divides its time between tasks without overhead, each task effectively gets $\frac{1}{N}$ of the GPU's time in each cycle. Consequently, the time to completion for each task becomes approximately $N \times e_i$, where e_i is the execution time with exclusive GPU access. This observation aligns with the time-slicing mechanism used in many Nvidia GPUs [49, 56, 63].

In a real-time embedded system, the response time of a task is typically modeled as the sum of its execution time (e_i) and contention time (c_i) [36]. The execution time (e_i) represents the duration a task takes to complete its operations if it has exclusive access to the GPU resources. In our design, we can easily estimate the execution time (e_i) in advance using benchmarks. However, the contention time (c_i) accounts for the additional delay introduced when multiple tasks compete for the same GPU resources. As GPUs often handle multiple tasks concurrently, they need to perform context switches, In addition to this, there can be contention for GPU resources. This contention can significantly increase the response times of the tasks. Essentially, the more frequently a task competes for GPU time, the higher the contention times of other co-scheduled tasks, thereby prolonging the response times for all tasks.

To build our model, we study how the response time of periodic tasks changes when they run alongside other tasks with different task rates. We list two simple examples shown below.

Example 1. We use the following configurations: t_1 (1,2) and t_2 (1.5,3). We consider 3 jobs for t_1 and 2 jobs for t_2, as shown in Figure 3. At time 0, both the two tasks are released together and we see that j_{11} and j_{21} are time sliced. As soon as j_{11} finishes its execution at time 2, j_{12} is immediately released, and it competes again with j_{21}. Here, j_{21} runs to completion at time 3. At time 3, the second job j_{22} is released and competes with j_{12} and later with j_{13}. Here we compute the response time (q_i) for each job using the formulae $q_i = f_{ik} - h_{ik}$, where f_{ik} is the completion time and h_{ik} is the release time of the task. More details on the response time calculation can be found in this textbook [36]. The response times for j_{11}, j_{12} and j_{13} are 2 each and the response times for j_{21} and j_{22} are 3 each. The average response times of t_1 and t_2 are 2 and 3 seconds respectively.

Example 2. We consider two tasks using the configuration t_1(0.75,2) and t_2 (1,2.5) as shown in Figure 4. Here j_{11} and j_{21} are time sliced since they are released together. j_{11} finishes its execution by time 1.5 and j_{21} finishes its execution by 1.75, because after 1.5, j_{21} runs alone. At time 2, j_{12} is released and since there are no tasks present in the GPU, j_{12} has entire access to the GPU resources. At time 2.5, when j_{22} is released, it time slices with j_{12} and j_{12} finishes its execution at time 3 and j_{22} finishes it execution at time 3.75. The job j_{13} is released independently at time 4 and it runs to completion within 0.75 seconds. The response times for j_{11}, j_{12} and j_{13} are 1.5, 1 and 0.75, respectively. The response times for j_{21} and j_{22} are 1.75 and 1.25, respectively. The average response times of t_1 and t_2 are 1.08 and 1.5 seconds.

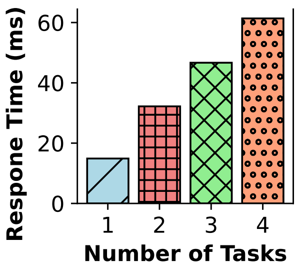


Fig. 2. Response time follows the pattern $N \times e_i$.

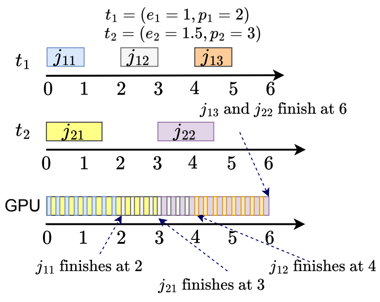


Fig. 3. Release time for each job for t_1 and t_2 in example 1.

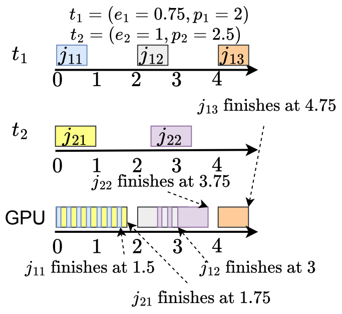

$t_1 = (e_1 = 0.75, p_1 = 2)$
$t_2 = (e_2 = 1, p_2 = 2.5)$
t_1
j_{11}
j_{12}
j_{13}
j_{13} finishes at 4.75
t_2
j_{21}
j_{22}
j_{22} finishes at 3.75
GPU
j_{11} finishes at 1.5
j_{12} finishes at 3
j_{21} finishes at 1.75

Fig. 4. Release time for each job for t_1 and t_2 in example 2.

We now try to derive our model by generalizing the examples. For two tasks t_1 (e_1, p_1) and t_2 (e_2, p_2), if t_1 operates with a period p_1 (where $p_1 = \frac{1}{r_1}$) close to e_1 and no other tasks are concurrently using the GPU, t_1 efficiently utilizes all available GPU time. Consequently, its response time q_1 equals e_1. However, when t_2 is active simultaneously, even with a longer period p_2 (where $p_2 = \frac{1}{r_2}$), both tasks compete for GPU time during overlapping periods. A higher r_2 value increases the frequency of overlapping periods. This increase in overlap intensifies resource competition and incurs frequent context switching. Consequently, it raises the response times for both t_1 and t_2. Conversely, a lower r_2 reduces t_1's response time. These principles also apply to t_2's response time relative to r_1. This demonstrates that the rate r_i (where $i \neq j$) of one task affects the response times of other tasks running on the GPU.

The response time of t_1 includes its execution time e_1 plus additional contention caused by t_2. This contention is influenced by r_2 and the context switching overhead θ_2 and can be computed as $(e_2 + \theta_2) \times r_2$. Previous work [69] reports that GPU context switching are long and can take between 50 and 750 μs, which noticeably increases task response times. Using this approach for measuring contention, the response time of t_1 can be modeled as shown in Equation (4), where e_1 and e_2 are the execution times of the two tasks.

$$q_1(k) = e_1 + (\theta_2 + e_2) \times r_2(k-1) \tag{4}$$

It is important to note that the model (4) shows the response time of one task $q_1(k)$ can be significantly affected by the rate/frequency of the other co-scheduled task (i.e., $r_2(k-1)$). Hence, we must design a MIMO controller that can effectively handle such interplay among different tasks. Any other control solutions that handle each GPU task independently from each other without considering such mutual impacts would not work well, as later demonstrated in our testbed evaluation.

To facilitate controller design later, we derive a difference equation based on the model in Equation (4) using the steps below for t_1.

$$q_1(k-1) = e_1 + (\theta_2 + e_2) \times r_2(k-2). \tag{5}$$

We obtain:

$$q_1(k) - q_1(k-1) = (\theta_2 + e_2) \times (r_2(k-1) - r_2(k-2)). \tag{6}$$

By defining the rate difference as:

$$\Delta r_2(k-1) = r_2(k-1) - r_2(k-2), \tag{7}$$

Equation (7) can be substituted to Equation (6) and we can obtain the general expression for a single task t_1 as shown in Equation (8).

$$q_1(k) = q_1(k-1) + (\theta_2 + e_2) \times \Delta r_2(k-1) \tag{8}$$

Equation (8) presents the difference equation for task t_1. If we consider task t_1 as the task competing for the resources of t_2, we can derive the difference equation for t_2 by following the same approach using Equations (5)–(8).

For a system comprising two tasks, the resultant expression is shown below:

$$\begin{aligned} q_1(k) &= q_1(k-1) + (\theta_2 + e_2) \times \Delta r_2(k-1), \\ q_2(k) &= q_2(k-1) + (\theta_1 + e_1) \times \Delta r_1(k-1). \end{aligned} \tag{9}$$

In Equation (9) $\theta_1 + e_1$ can be substituted as $\widetilde{e_1}$ and $\theta_2 + e_2$ can be substituted as $\widetilde{e_2}$ and the expression is given below.

$$\begin{aligned} q_1(k) &= q_1(k-1) + \widetilde{e_2} \times \Delta r_2(k-1), \\ q_2(k) &= q_2(k-1) + \widetilde{e_1} \times \Delta r_1(k-1). \end{aligned} \tag{10}$$

For the rest of the derivation, we use the model from Equation (10) instead of Equation (9) as the latter is the expanded form of the former

We now represent the model in the standard MIMO format, as shown in Equation (11).

$$\mathbf{q(k)} = \mathbf{q(k-1)} + \mathbf{A} \times \mathbf{\Delta r(k-1)} \tag{11}$$

where $\mathbf{A}$ is a square matrix with $A_{ii} = 0 (1 <= i <= N)$ and $A_{ij} = \widetilde{e_j} (i \neq j)$. The $\widetilde{e_j}$ values are the corresponding effective execution times of the tasks.

4.4.1 Extension to N tasks: For tasks t_i (e_i, p_i) where $i = 1, 2, \ldots, N$, if t_1 operates with a period p_1 (where $p_1 = \frac{1}{r_1}$) very close to its execution time e_1 and no other tasks are concurrently using the GPU, t_1 efficiently utilizes all available GPU resources, and its response time q_1 equals e_1. However, when other tasks t_j (where $j \neq 1$) are active simultaneously, even with longer periods $p_j = \frac{1}{r_j}$, all tasks contend for GPU resources during the overlapping periods.

A higher rate r_j for task t_j increases the frequency of overlapping periods, intensifying resource competition and thereby raising the response times for all tasks t_i. Conversely, a lower rate r_j reduces the response times of all tasks affected by t_j's competition for resources. These principles extend to the response time of any task t_i relative to the rates r_j of other tasks t_j $(j \neq i)$. This scenario clearly demonstrates that the rate r_j of one task affects the execution and response times of other tasks. Extending the model used for two tasks to N tasks for t_1 the contention time caused by other $(N-1)$ tasks can be represented as the product $\begin{bmatrix} r_2 & r_3 & \cdots & r_N \end{bmatrix} \odot \begin{bmatrix} e_2 + \theta_2 & e_3 + \theta_3 & \cdots & e_N + \theta_N \end{bmatrix}$. The response time for the task t_1 can be expressed in summation as

$$q_1(k) = e_1 + \sum_{i=2}^{N} (e_i + \theta_i) \cdot r_i(k-1) \tag{12}$$

$$q_1(k) = e_1 + \sum_{i=2}^{N} \widetilde{e_i} \cdot r_i(k-1) \tag{13}$$

We compute the difference equation for the above model and for a system comprising N tasks the response time for t_1 can be represented as

$$q_1(k) = q_1(k-1) + \sum_{i=2}^{N} \widetilde{e_i} \cdot \Delta r_i(k-1) \tag{14}$$

when expanded as shown below

$$q_1(k) = q_1(k-1) + \widetilde{e_2} * \Delta r_2(k-1) + \widetilde{e_3} * \Delta r_3(k-1) \ldots \widetilde{e_i} * \Delta r_N(k-1) \tag{15}$$

This formulation captures the idea that the response time for t_1 is influenced by the cumulative effect of being preempted by the system's other $N-1$ tasks. The generalized model for a system comprising N tasks is shown in the Equation (16). The actuation for the controller using this model is discussed in detail in the next subsection.

$$\begin{bmatrix} q_1(k) \\ q_2(k) \\ \vdots \\ q_N(k) \end{bmatrix} = \begin{bmatrix} q_1(k-1) \\ q_2(k-1) \\ \vdots \\ q_N(k-1) \end{bmatrix} + \begin{bmatrix} 0 & \widetilde{e_2} & \ldots & \widetilde{e_N} \\ \widetilde{e_1} & 0 & \ldots & \widetilde{e_N} \\ \vdots & \vdots & \ddots & \vdots \\ \widetilde{e_1} & \widetilde{e_2} & \ldots & 0 \end{bmatrix} \times \begin{bmatrix} \Delta r_1(k-1) \\ \Delta r_2(k-1) \\ \vdots \\ \Delta r_N(k-1) \end{bmatrix} \tag{16}$$

4.5 Controller Design and Analysis

We present the design and analysis of our proportional controller FC-GPU. 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.

4.5.1 State space representation: Based on the system model, a MIMO controller can be designed to ensure the RTR settles to its desired set point. The PID control approach used in earlier works on real-time feedback control scheduling [32] is suitable for real-time systems. Our design utilizes 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 apply the Proportional control (P) theory, as outlined in [14], to design our controller based on the system model described in Equation (11). The basic idea of a P-controller is to maintain system performance by continuously adjusting the control input based on the current error. The control objective is to minimize the error by dynamically adjusting the input task rate r_i in real-time. A P-controller achieves this by applying a control input $r(k)$ directly proportional to the current error. This straightforward approach continuously adjusts the input at each sampling period based on the instantaneous error, offering a computationally efficient solution. It is especially advantageous since it has a very minimal computational overhead. The State space representation of a MIMO controller is shown in Equation (17).

$$\begin{aligned} \dot{x}(k) &= Ax(k) + Bu(k) \\ y(k) &= Cx(k) + Du(k) \end{aligned} \tag{17}$$

Here- $x(k)$ is the state vector , $\dot{x}(k)$ is the derivative of the state vector , $u(k)$ is the input vector, $y(k)$ is the output vector, A, B, C, and D are matrices that define the system dynamics and output. For our system, We choose B as the matrix from our system model, A and C are Identity matrices, and D is 0. The final representation of our MIMO system is shown in Equation (18)

$$\begin{bmatrix} \dot{x}(k) \\ y(k) \end{bmatrix} = \begin{bmatrix} I & B \\ I & 0 \end{bmatrix} \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} \tag{18}$$

4.5.2 Controller Design: In our control problem, we aim to manage the controlled variables $rtr_i(k)$ by adjusting the manipulated variables $r_i(k)$. To design the controller, we compute RTR by multiplying the task rates by their corresponding response times, as stated before. For a system consisting of N tasks, the model can be generalized as shown in Equation (19).

$$\begin{bmatrix} rtr_1(k) \\ rtr_2(k) \\ \vdots \\ rtr_N(k) \end{bmatrix} = \begin{bmatrix} rtr_1(k-1) \\ rtr_2(k-1) \\ \vdots \\ rtr_N(k-1) \end{bmatrix} + \begin{bmatrix} 0 & a_{12} & \dots & a_{1N} \\ a_{21} & 0 & \dots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \dots & 0 \end{bmatrix} \times \begin{bmatrix} \Delta r_1(k-1) \\ \Delta r_2(k-1) \\ \vdots \\ \Delta r_N(k-1) \end{bmatrix} \odot \begin{bmatrix} \Delta r_1(k-1) \\ \Delta r_2(k-1) \\ \vdots \\ \Delta r_N(k-1) \end{bmatrix} \tag{19}$$

Equation (19) can be represented as

$$rtr_i(k) = rtr_i(k-1) + \sum_{\substack{j=1 \\ j\neq i}}^{N} a_{ij} \cdot \Delta r_i(k-1) \cdot \Delta r_j(k-1), \quad i = 1, 2, \dots, N. \tag{20}$$

Example: For a system with two tasks, the relationship is captured as:

$$\begin{aligned} rtr_1(k) &= rtr_1(k-1) + a_{12} \times \Delta r_1(k-1) \times \Delta r_2(k-1), \\ rtr_2(k) &= rtr_2(k-1) + a_{21} \times \Delta r_1(k-1) \times \Delta r_2(k-1). \end{aligned} \tag{21}$$

To transform the model in Equation (20) into a standard state space form, we first simplify the products of the task rate differences $\Delta r_i(k-1) \times \Delta r_j(k-1)$ by introducing new variables $\Delta r_{ij}(k-1)$ for each ordered pair (i, j) with $i < j$. This modified model is shown in Equation (22).

$$rtr_i(k) = rtr_i(k-1) + \sum_{\substack{j=1 \\ j\neq i}}^{N} a_{ij}\, \Delta r_{ij}(k-1), \quad i = 1, 2, \dots, N. \tag{22}$$

Example: The non-linear expression for two tasks in Equation (21) can be linearized as shown below

$$\begin{aligned} rtr_1(k) &= rtr_1(k-1) + a_{12}\, \Delta r_{12}(k-1), \\ rtr_2(k) &= rtr_2(k-1) + a_{21}\, \Delta r_{12}(k-1). \end{aligned} \tag{23}$$

We can represent our control problem using the state space equations given in Equation (24). In this representation, A and C are identity matrices, and B is a matrix that contains the effective execution times of the tasks. The term $\mathbf{\Delta r(k-1)}$ represents the control input vector, whose elements are $\Delta r_{ij}(k-1)$ for $i < j$.

$$\begin{bmatrix} \dot{x}(k) \\ y(k) \end{bmatrix} = \begin{bmatrix} I & B \\ I & 0 \end{bmatrix} \begin{bmatrix} x(k) \\ \mathbf{\Delta r(k-1)} \end{bmatrix} \tag{24}$$

In a MIMO controller, the system's performance is influenced by the inputs $u(k)$ and the internal state. The equation for a proportional MIMO controller is given by $\mathbf{u(k)} = -\mathbf{K_P} \times (\mathbf{z(k)} - \mathbf{y(k)})$, where $\mathbf{z(k)}$ is the reference vector representing the desired set points, and $\mathbf{y(k)}$ is the actual output of the system. After determining the desired system behavior, we use a technique called pole placement to find the optimal gain matrix for the system. This method adjusts the system's response by placing its poles in specific locations. We aim to place these poles within the unit circle to ensure stability and performance, as discussed in references [68] and [16]. We typically use Matlab to compute the gain matrix for the closed-loop system. In our MIMO controller, using the model in Equation (24), we can control RTR; but in our testbed, we can only directly adjust the task rates $r_1(k), r_2(k), \dots, r_n(k)$. To obtain $r_1(k)$ and $r_2(k), \dots, r_n(k)$ we transform $\mathbf{\Delta r}(k)$ into task rates $r_1(k)$ to $r_n(k)$ through a process called actuation. The inputs $r_1(k)$ to $r_n(k)$ through the actuations are provided in Equation (25).

$$r_i(k) = r_i(k-1) + \frac{u_i(k-1)}{\prod_{\substack{j=1 \\ j \neq i}}^{N} r_j(k-1)}, \tag{25}$$

Example: The equations for the input task rates for the two tasks using actuation are:

$$\begin{aligned} r_1(k) &= r_1(k-1) + \frac{u_1(k-1)}{r_2(k-1)}, \\ r_2(k) &= r_2(k-1) + \frac{u_2(k-1)}{r_1(k-1)}. \end{aligned} \tag{26}$$

4.6 Stability Analysis

A key advantage of control-theoretic design lies in its capacity to provide analytical assurance of system stability and control accuracy, even when the model used for controller design can be inaccurate or vary significantly at runtime. We now outline the steps of analyzing the stability of our MIMO controller.

(1) Derive the control inputs $u(k)$ based on the system model stated in Equation (17)
(2) Derive the closed-loop system by substituting the control inputs $u(k)$ derived in Step 1 into the actual system model.
(3) Derive the stability condition by computing the eigenvalues of the closed-loop system. If all the eigenvalues are inside the unit circle, the closed-loop system is stable.

We consider a system comprising N tasks, where the matrices in the state-space model (17) have the following dimensions: $A \in \mathbb{R}^{N \times N}$, $B \in \mathbb{R}^{N \times N}$, and $x(k), u(k) \in \mathbb{R}^N$. From the state-space model in Equation (24), we define the system matrices as follows: $A = I_{N \times N}$, $B \in \mathbb{R}^{N \times N}$, $C = I_{N \times N}$, and $D = 0$. We assume a scalar state-feedback control law of the form $u(k) = -k_p x(k)$, where $k_p \in \mathbb{R}$ is a scalar feedback gain applied identically across all tasks. The resulting closed-loop system can be represented in an augmented form as shown in Equation (27).

$$\begin{bmatrix} I_{N \times N} & -k_p B \\ I_{N \times N} & 0 \end{bmatrix} \tag{27}$$

In the next step, we compute the eigenvalues for the closed-loop system to derive the stability range [68].

$$M(\lambda) = \begin{bmatrix} I_N - \lambda I_N & -k_p B \\ I_N & -\lambda I_N \end{bmatrix} \tag{28}$$

Since $M(\lambda)$ is a block matrix and $-\lambda I_N$ is invertible for $\lambda \neq 0$, we apply the *Schur complement formula* to compute the determinant:

$$\det(M(\lambda)) = \det(-\lambda I_N) \cdot \det\left((1-\lambda) I_N - \frac{k_p}{\lambda} B\right) \tag{29}$$

Let μ_i be the eigenvalues of B. The eigenvalues of the matrix $(1-\lambda)I_N - \frac{k_p}{\lambda}B$ are:

$$(1-\lambda) - \frac{k_p}{\lambda}\mu_i \tag{30}$$

Setting the determinant to zero gives:

$$(1-\lambda) - \frac{k_p}{\lambda}\mu_i = 0 \tag{31}$$

Multiplying both sides by λ, we get:

$$\lambda(1-\lambda) = k_p\mu_i \tag{32}$$

This yields a quadratic equation in λ:

$$\lambda^2 - \lambda + k_p\mu_i = 0 \tag{33}$$

Solving for λ, we obtain:

$$\lambda = \frac{1 \pm \sqrt{1-4k_p\mu_i}}{2} \tag{34}$$

The system is stable if all eigenvalues satisfy $|\lambda| < 1$, which holds if:

$$0 < k_p\mu_i < \frac{1}{4} \tag{35}$$

Letting $\mu_{\max} = \rho(B)$ be the spectral radius of B, the sufficient condition for stability becomes:

$$0 < k_p < \frac{1}{4 \cdot \rho(B)} \tag{36}$$

Equation (36) has established a stability region in which the closed-loop system will remain stable if the gains of the actual system are within this region. In other words, the response times of the controlled real-time tasks can be controlled to their desired set points, if their execution time variations are within the derived range. In our stability analysis, we assume the chosen ranges are feasible, suggesting that a range of task rates within acceptable parameters may align the task RTR with its set point. We now use an example to show the stability analysis.

Example: We consider a system comprising of two tasks. We choose two control inputs $u(k)$ to regulate the system's behavior. Let $A = I_{2\times 2}$ be the identity matrix I, and let $B \in \mathbb{R}^{2\times 2}$ be a matrix representing the system's input dynamics. We calculate the closed-loop representation for the system to perform stability analysis. For simplicity, we assume k_p is a scalar. The closed-loop expression is given by:

$$\begin{bmatrix} A & -k_p \cdot B \\ I & D \end{bmatrix} \tag{37}$$

Since D is a zero matrix, we find the eigenvalues λ for the matrix in Equation (37) by substituting $A = I$ and $C = I$.

$$\begin{bmatrix} I - \lambda I & -k_p \cdot B \\ I & 0 - \lambda I \end{bmatrix} \tag{38}$$

Considering a system of two tasks, where I and B are 2×2 matrices, the full matrix expands to:

$$M = \begin{bmatrix} (1-\lambda) & 0 & 0 & -k_p \times b_1 \\ 0 & (1-\lambda) & -k_p \times b_2 & 0 \\ 1 & 0 & -\lambda & 0 \\ 0 & 1 & 0 & -\lambda \end{bmatrix} \tag{39}$$

To simplify the matrix and facilitate the calculation of the determinant, we swap rows 2 and 3 and columns 2 and 3. We then compute the determinant of M' and set it equal to zero.

$$M' = \begin{bmatrix} (1-\lambda) & 0 & 0 & -k_p \times b_1 \\ 1 & -\lambda & 0 & 0 \\ 0 & -k_p \times b_2 & (1-\lambda) & 0 \\ 0 & 0 & 1 & -\lambda \end{bmatrix} \tag{40}$$

We compute

$$\det(M') = 0 \tag{41}$$

This yields the expression:

$$\lambda \left((1-\lambda)^2 \lambda - k_p^2 b_1 b_2 \right) = 0 \tag{42}$$

$$\lambda\,(1-\lambda)^2 \;=\; k_p^2\, b_1\, b_2. \tag{43}$$

Letting $\gamma = k_p \sqrt{b_1 b_2}$, we get:

$$\lambda(1-\lambda)^2 = \gamma^2 \tag{44}$$

This leads to two quadratic equations:

$$\lambda^2 - \lambda + \gamma = 0 \quad \text{and} \quad \lambda^2 - \lambda - \gamma = 0 \tag{45}$$

For the first equation, the discriminant is $D = 1 - 4\gamma$. For real roots, $D \geq 0$, so:

$$\gamma \leq \frac{1}{4} \tag{46}$$

The roots are:

$$\lambda = \frac{1 \pm \sqrt{1 - 4\gamma}}{2} \tag{47}$$

For stability ($|\lambda| < 1$), we require:

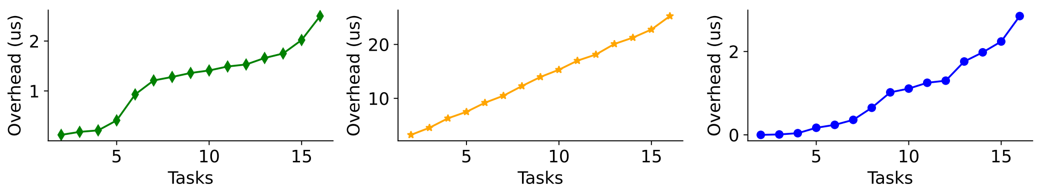
$$0 < \gamma < \frac{1}{4} \tag{48}$$

Substituting back $\gamma = k_p \sqrt{b_1 b_2}$ and assuming $b_1 b_2 > 0$, we find:

$$0 < k_p < \frac{1}{4\sqrt{b_1 b_2}} \tag{49}$$

4.7 Discussion

Design consideration. Our design assumes that the number of tasks, N, is known, which is reasonable because most embedded systems have fixed numbers of tasks. When the number of tasks changes, a different system model must be used to design a different controller. For systems whose tasks may change at runtime, we can have multiple system models for different task numbers and switch the designed controllers upon dynamic task arrival or departure. We assume a context-switch overhead of 750 μs, an upper bound reported in [69]. In real-time GPU scheduling, a Proportional controller imposes minimal computational overhead and is straightforward to implement, whereas MPC must solve an optimization problem at every step, leading to significant CPU and memory usage. We operate the proportional gains within the proven stability region, ensuring no overshoot or instability under changing workloads. By contrast, MPC's high overhead and complexity can violate the tight timing constraints of embedded platforms. For our experiments, the acceptable system performance is when the RTR (i.e., $rtr_i(k)$) should fall within ±0.02 of the set point.



(a) Response time Measurement (b) Controller (c) Rate adaptation

Fig. 5. (a) The overhead for monitoring the response times. (b) The overhead for the proportional controller. (c) The overhead for the task actuator involves updating the task rates.

Overhead Analysis. The RTR control service necessarily introduces overhead. This overhead includes response time monitoring, control computation, and rate adaptation. FC-GPU is viable only if the overhead is sufficiently low. To accurately measure the overhead, we add time stamps at the starting and finishing points of each segment of the control service code to get the execution time. Since FC-GPU runs at the highest Linux real-time priority, the code segment between two timestamps will not be preempted during its execution. Hence, the time-stamped result accurately reflects the real execution overhead. To achieve fine-grained measurements, we put each code segment of the control service (monitoring, control, and rate adaptation) into a loop to run the tested code 100 times for an averaged result by minimizing system noise.

Figures 5(a) and 5(c) show that the overheads of monitoring response times and rate adaptation for FC-GPU increase from less than one microsecond to about 3 microseconds, when the number of tasks in the system increases from 2 to 16. The overheads are negligible because both parts have only 10 or fewer lines of code. The controller has a slightly higher overhead that ranges from 2 to 28 microseconds, as shown in Figure 5(b). This is because the controller has several operations, such as the feedback control law and computing the deviation from the set points. In summary, such low overheads of FC-GPU are acceptable to most systems. In our implementation, the FC-GPU controller is running on a dedicated CPU core to communicate with the GPU tasks via share memory.

Aperiodic or Sporadic Tasks. FC-GPU is *not* limited to periodic tasks, even though we focus mainly on periodic tasks in this paper. FC-GPU can be integrated with classical approaches that handle aperiodic or sporadic tasks. For example, in real-time scheduling, a polling server or a sporadic server [15] can be used to convert aperiodic or sporadic tasks into a periodic framework. Specifically, for each aperiodic task, a polling server is created with a period and execution time budget. Upon the arrival of an aperiodic task, it will be buffered into a queue. In each period, the polling server will 1) process the aperiodic task in the queue and run until its budgeted execution time is exhausted, or 2) suspend itself if there is no task in the queue [15]. FC-GPU can be integrated with polling server to handle aperiodic tasks. Due to its feedback nature, FC-GPU adapts the rates of all tasks (including the polling server) to control the task RTRs at the desired set point. Upon the arrival of an aperiodic task that will be processed by the polling server, the RTRs of all tasks may increase due to the competition of this aperiodic task for GPU resource. As a result, FC-GPU can adjust the task rates to run them slower, so that the task RTRs can be controlled to the set points again. This is similar to the case when a periodic task (the polling server) has unexpected execution time increase (from 0 to the budgeted execution time). This case has been tested in Sections 5.1 and 5.2 to show that FC-GPU can quickly adapt to such runtime execution time variations.

5 Testbed Evaluation

Hardware Testbed. Our testbed includes an Nvidia RTX 3090 GPU and an AMD MI-100 GPU. We run Ubuntu 20.04 LTS as the operating system, along with CUDA Toolkit 12.0 on the Nvidia GPU and ROCm 5.3.0 on the AMD GPU. We implement our controller in C++. To reduce redundant data copies during each control period, on the Nvidia GPU we allocate host memory with `cudaMallocHost` and for the AMD GPU we use the ROCm counterpart, `hipHostMalloc`, which provides pinned (page-locked) host memory and enables more efficient data transfers between the CPU and the GPU.

Workload Description. We use workloads from Nvidia Samples [5] and the Rodinia Benchmark [26]. These benchmarks are used in work done on real-time systems [3][69][49][66]. The Nvidia Samples suite includes two compute-intensive tasks: *dxtc*, which performs texture compression, and *stereodisparity*, which processes stereo vision data. The Rodinia Benchmark suite provides four tasks: *srad*, which reduces speckles via anisotropic diffusion; *hotspot*, which simulates heat dispersion on a 2D grid and is memory-intensive; *floyd-warshall*, which computes all-pairs shortest paths in a compute-intensive manner; and *stencil*, which performs memory-intensive computations on 3D grids. The control period for all experiments is 4 seconds. In this section, all the task durations are in milliseconds.

Metrics. We use four metrics to compare FC-GPU with the baselines. The first metric is RTR (rtr) as shown in Equation (1). The second metric is the Deadline Miss Ratio (DMR) in every control period. The deadline-miss ratio is an important metric for real-time systems. When the system's RTR exceeds 1, it results in missed deadlines, indicating that the task's response time exceeds its deadline, which is also its period. To calculate DMR, we monitor jobs whose response time is longer than the task's period and mark these as deadline violations. This process repeats for each control period to determine the frequency of such violations. The third metric is task execution jitter. Task execution jitter denotes the largest difference between the execution times of a task's jobs, expressed as a percentage of the mean execution time. The fourth metric is the variance in response time, which indicates how much the response times fluctuate around their mean.

In this section, we present the results of five sets of experiments on our testbed. Section 5.1 shows the limitations of open-loop GPU scheduling approaches. Section 5.2 compares FC-GPU with a state-of-the-art solution. Section 5.3 compares FC-GPU with two well-designed closed-loop baselines. Section 5.4 examines the accuracy of FC-GPU's control at different set points across different GPUs. Section 5.5 demonstrates how the system adapts to online workload variations.

5.1 Limitation of Traditional Open-loop Scheduling

One of the primary motivations for proposing a closed-loop solution is that existing open-loop scheduling solutions [57] do not adapt the task rates when there are fluctuations in the workloads. In order to underscore this limitation, we illustrate how an open-loop solution that lacks any feedback or control logic cannot maintain the RTR of each task at its intended set point. For this illustration, we choose two workloads, mm $(10, 20)$ and $stencil$ $(200, 400)$. We carefully tune their task rates to ensure that each workload initially converges close to its respective set point without inducing large fluctuations. We select the set points for both workloads to be 0.90. Under the initial workload conditions, both mm and $stencil$ have their RTR at their set points as shown in Figure 6(a). Figure 6(b) depict the tasks' periods remain constant throughout the experiment.

Real-time systems commonly undergo workload fluctuations at runtime due to external disturbances, environmental variability, or sporadic task arrivals. These variations can significantly affect performance and responsiveness, potentially causing tasks to exceed their deadlines. For example, the traffic light detector of Apollo [1] , a well-known open autonomous driving platform, can have

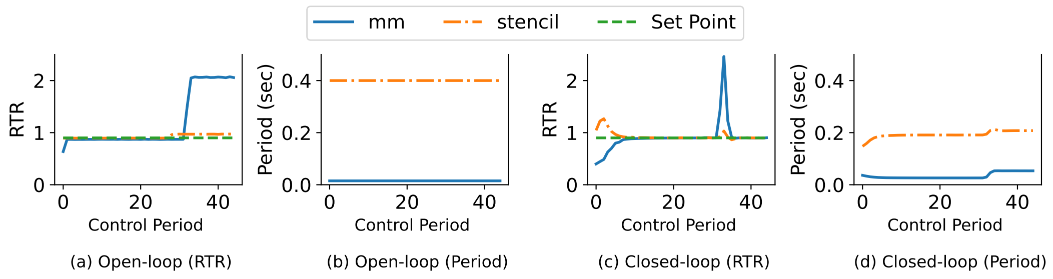


Fig. 6. Open-loop vs. closed-loop scheduling. Figure (a) shows that with no rate adaptation, both tasks deviate from their set points since the response times of the tasks increase and miss deadlines when the workload increases at the 30th control period. Figure (b) shows the variation in task period for the open-loop experiment. Figure (c) shows that the closed-loop solution ensures the tasks meet their set points. Figure (d) reveals how a closed-loop controller continuously updates the task periods.

a 330% execution time increase depending on the number of detected traffic lights [2, 62]. We increase the computation for the two workloads in the control period 30 to evaluate how FC-GPU handles such conditions. To do this, we increase the input size of *mm* from 512k to 1024k, effectively doubling its execution time. Simultaneously, we increase the iteration count of *stencil* from 375 to 400.

After the control period 30, we observe that *mm*'s RTR exceeds its set point. Concurrently, *stencil* also experiences an elevated RTR as the two tasks compete for GPU resources. This causes both tasks to violate their respective RTR constraints, and *mm* ultimately misses its deadline. These observations confirm that an open-loop scheduling framework is inadequate. It does not track changes in workload or respond to rising computational demands.

We propose our closed-loop feedback control solution (FC-GPU) to overcome this limitation. In FC-GPU, the rate of each task is recomputed at every period interval based on the observed RTR of all the tasks. In Figures 6(c) We demonstrate how this closed-loop approach drives *mm* and *stencil* to their respective RTR set points by adjusting the periods shown in Figure 6(d). In the control period 30, when the computation increases, the RTR increases for both tasks, but the controller ensures that the tasks meet their RTR set points in subsequent control periods.

5.2 Comparison with the State-of-the-Art Open-loop Solutions

In this experiment, we compare FC-GPU with a state-of-the-art solution, GPU-TDMh [51], because both FC-GPU and GPU-TDMh are based on time slicing. As discussed in Section 2, GPU-TDMh is an *open-loop* solution that cannot adapt the system at runtime. Hence, it can be expected that GPU-TDMh shares the same problem with the open-loop solution in the last section. Specifically, GPU-TDMh conducts offline profiling to find a fixed slice length for each GPU task so that every job's slices can be finished before its deadline. At runtime, a lightweight reservation server wakes up every period to check each task. If a task has work pending, it launches one pre-sized sub-kernel corresponding to its slice, then waits for the next period.

To perform the experiment, we use the matrix multiplication benchmark used in [51]. We choose two matrix multiplication workloads $mm1$ and $mm2$. At the beginning of the experiment, we carefully select the optimal kernel slice configuration for GPU-TDMh to ensure that it meets all the task deadlines. From the control periods $0 - 25$, we can observe that both tasks are controlled to the RTR set point as shown in Figure 7(a). At the control period 25, we increase the input size for both the workloads by 1.5×, and we observe that the RTR exceeds 1 and there are constant deadline

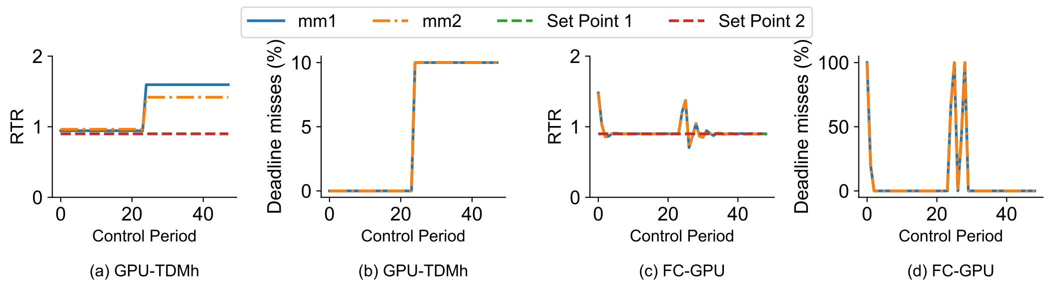


Fig. 7. GPU-TDMh vs. FC-GPU scheduling. (a) shows that GPU-TDMh initially chooses the appropriate kernel slices to ensure no deadline misses. At the control period 25, both tasks have their response times deviating from their set points due to workload increase. (b) shows that GPU-TDMh has 100% deadline misses after the workload increase in the 25th control period. (c) shows that FC-GPU ensures the tasks meet their set points despite the same workload increase. Figure (d) shows that FC-GPU reduces the deadline misses to 0% after the workload increase.

misses as shown in Figure 7(b). One of the significant limitations of GPU-TDMh, as an open-loop solution, is that it is designed offline and cannot adapt to online changes, such as the arrival of a new task or when workloads change. For FC-GPU, at control period 25 when the workloads increase, it adjusts the task rates using the designed MIMO controller and so the task RTRs can be controlled to their set points. As a result, the deadline miss ratio drops back to 0%, as shown in Figures 7(c) and 7(d). Experiments 5.1 and 5.2 show that open-loop solutions cannot adapt to runtime changes, and hence in the next subsequent section, we compare with the two baselines that use closed-loop control.

5.3 Comparison with Baseline Closed-loop Solutions

In this experiment, we compare FC-GPU with two well-designed closed-loop solutions: (1) Adhoc and (2) SISO. We launch both tasks concurrently as separate processes. We choose a set point of 0.80 for this experiment. We choose two workloads for this experiment, *mm* $(367, 800)$ and *stencil* $(200, 400)$.

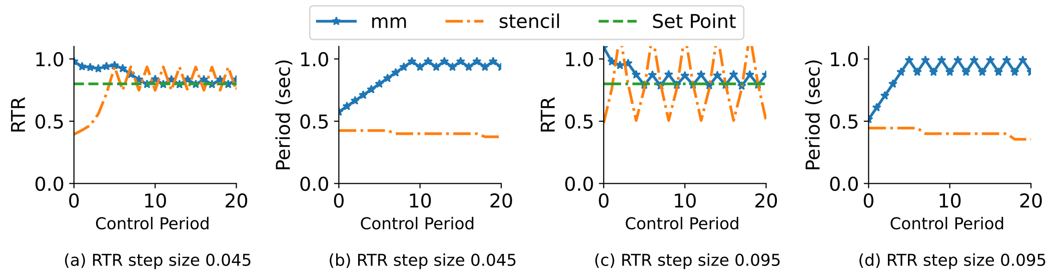


Fig. 8. Performance of the Ad-hoc controller for two different step sizes. (a) With a small step size, the controller output exhibits minimal oscillations and closely approaches the set point; (b) the task period remains relatively stable. In contrast, (c) a larger step size induces large oscillations around the set point, and (d) the corresponding task period shows greater variation.

Adhoc. Some recent research efforts on GPU scheduling (e.g. [3]) propose heuristic-based adaptation mechanisms. We refer to these heuristic-based solutions collectively as "Adhoc." For

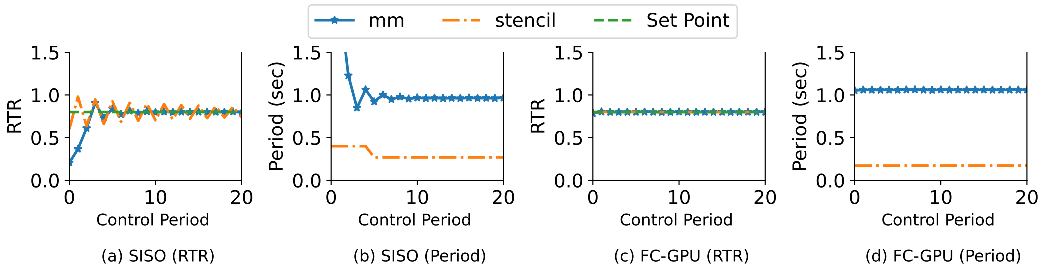


Fig. 9. Comparison of (a) SISO and (c) FC-GPU. (a) SISO exhibits small oscillations near the set points. (b) and (d) illustrate the task periods of both SISO and FC-GPU. (c) FC-GPU's MIMO control ensures stable and accurate convergence for both tasks without requiring extensive offline tuning.

the Adhoc controller, if the current RTR in a control period is below the set point, the task rate is increased by a fixed step size; if the RTR exceeds the set point, the task rate is decreased by that same step size. Figures 8(a) and 8(b) illustrate the Adhoc controller's behavior for a step size of 0.045. For this step size, the system remains relatively close to the set point with comparatively limited oscillations. However, using a larger step size of 0.095 in Figures 8(c) induces oscillations. Although the Adhoc controller is simple, extensive offline profiling is required to discover an appropriate step size for every task configuration and GPU platform. Moreover, if hardware or application characteristics deviate from their profiled values, the predetermined step size can become suboptimal, resulting in significant performance variance.

SISO. The single-input-single-output (SISO) controller is a commonly used baseline in control systems. It is designed using a linear model [14]. The SISO controller independently measures the system's output for each task and compares it to the set point. Then it adjusts each task's period based on these comparisons. This adjustment reduces the error between the measured output and the set point, keeping the system's performance within the desired range.

The variation in the RTR and the task periods for the two tasks is shown in Figures 9(a) and Figures 9(b). In Figures 9(a), we observe that SISO exhibits small oscillations near its set points. This is because each SISO controller aims exclusively to stabilize the RTR of a single task and does not account for contention on shared GPU resources. However, since each SISO controller is designed to manage the RTR of a single task, it requires precise tuning. Even minor deviations can result in significant oscillations.

FC-GPU. The proposed solution utilizes a multi-input-multi-output (MIMO) feedback controller to dynamically adjust the task rates for both tasks, explicitly accounting for mutual contention. In contrast to the Adhoc and SISO controllers, FC-GPU takes mutual interference among functions into account. Figures 9(c) and 9(d) display the RTR and periods for the same pair of tasks under FC-GPU's MIMO control. Both tasks converge to their desired set points with minimal oscillations. Table 1 shows that FC-GPU has the smallest task execution jitter and variance in response time. This is because FC-GPU performs coordinated control using a MIMO controller to adjust task rates. As shown in Equation (26), the rate of one task is adjusted based on the deviation observed in another. When multiple tasks run concurrently on the GPU, resource contention occurs. The controller mitigates this by coordinating the task rates, reducing overlap, and minimizing contention. As a result, kernel execution times and response times become more stable compared to the baselines.

5.4 Control Accuracy under Different Set Points

In this section, we focus on evaluating the performance of FC-GPU across a range of set points on the Nvidia and AMD GPUs. Our primary objective is to demonstrate our solution's real-time

Table 1. Execution jitter and response time variance for all the baselines and FC-GPU. Here Δ_1 refers to the step size 0.045 and Δ_2 refers to the step size 0.095.

Metric	Adhoc				SISO		FC-GPU	
	mm Δ_1	mm Δ_2	stencil Δ_1	stencil Δ_2	mm	stencil	mm	stencil
Execution Jitter (%)	25.97	11.58	0.31	0.36	24.74	0.48	5.15	0.14
Response Time Var (ms^2)	3453.91	362.29	803.47	131.01	4499.55	381.57	0.45	0.01

adaptability and responsiveness. As observed in previous experiments, we compute the average RTR and their standard deviation after the system reaches a steady state to mitigate the influence of any transient states at the beginning of each run. We test different RTR set points ranging from 0.75 to 0.90 with an interval of 0.05. We also measure the Deadline Miss Ratio (DMR) in every control period. The deadline-miss ratio is an important metric for real-time systems. When the system's RTR exceeds 1, it results in missed deadlines, indicating that the task's response time exceeds its deadline, which is also its period. To calculate the deadline miss ratio, we monitor jobs whose response time is longer than the task's period and mark these as deadline violations. This process repeats for each control period to determine the frequency of such violations.

Nvidia GPU. We test four tasks using the following configurations: `dxtc` (55.3, 600), `mm` (20, 200), `stereodisparity` (25.97, 200), and `hotspot` (68, 200). Table 2 shows the average RTR variation across all set points from 0.75 to 0.90 for FC-GPU. As shown in Table 2, all tasks on the Nvidia GPU converge to their respective set points, as indicated by the mean RTR values closely matching the target values across all benchmarks. The minimal standard deviations highlight the stability of the response, indicating negligible oscillations. This showcases the effectiveness of FC-GPU in accommodating diverse set points while ensuring reliable and consistent convergence for each task without significant runtime variation. As RTR is controlled to the set point in every case, there are no missed deadlines across all set points.

Table 2. Mean and standard deviation of RTR across set points for each task on Nvidia and AMD GPUs.

GPU	Nvidia GPU								AMD GPU							
Benchmark	dxtc		stereo		mm		hotspot		mm		flyod		stencil		srad	
Set point	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std
0.75	0.75	0.00	0.75	0.00	0.75	0.00	0.75	0.00	0.75	0.01	0.75	0.01	0.75	0.01	0.75	0.01
0.80	0.80	0.00	0.80	0.00	0.80	0.00	0.80	0.00	0.80	0.01	0.80	0.01	0.80	0.01	0.80	0.01
0.85	0.85	0.00	0.85	0.00	0.85	0.00	0.85	0.00	0.85	0.01	0.85	0.01	0.85	0.00	0.85	0.01
0.90	0.90	0.00	0.90	0.00	0.90	0.00	0.90	0.00	0.90	0.00	0.90	0.01	0.90	0.01	0.90	0.01

AMD GPU. To ensure that our solution is applicable across various GPUs, we validate our proposed solution on an AMD MI-100 GPU. In this experiment, we apply the RTR control for the AMD MI-100 while reusing the model coefficients used initially for the Nvidia RTX 3090 GPU. This demonstrates the robustness of our solution, showing that it can achieve the desired set point even without performing pole placement for FC-GPU on the MI-100. We choose the four tasks for this experiment by excluding workloads from the Nvidia benchmark suite. The four tasks are `mm` (25, 75), `srad` (25.97, 100), `floyd` (12.6, 100), and `stencil` (51, 60). The RTR variation across all four tasks, with set points ranging from 0.75 to 0.90, is illustrated in Table 2. From the Table, it is evident that FC-GPU exhibits the average RTR at the respective set points. The standard deviations are small, indicating minimal oscillation from the set point.

Additionally, we compute the deadline miss ratio (DMR) for each set point. Noticeably, the deadline misses gradually increase as the set points exceed 0.85 but remain less than 3% as shown

Table 3. Deadline miss ratio (%) for all the tasks under different set points on the AMD GPU.

Benchmark	0.75	0.80	0.85	0.90
mm	0.00	0.00	0.18	0.17
floyd	0.00	0.40	3.04	3.20
srad	0.00	0.00	0.00	0.00
stencil	0.24	0.47	0.36	0.36

in Table 3. A closer inspection of the raw data from each control period on the AMD GPU shows that deadline misses occasionally occur around the mid-range of the run (e.g., between periods 40 and 60). These arise from cache misses and data-loading delays, increasing the overall response time in those intervals. However, in subsequent periods, the controller adjusts the task rates so that all tasks meet their RTR set points, ensuring that the DMR returns to 0%.

5.5 Online Workload Variation

Real-time systems commonly undergo workload fluctuations at runtime due to external disturbances, environmental variability, or dynamic task arrivals. These variations can significantly affect performance and responsiveness, potentially causing tasks to exceed their deadlines. To evaluate how FC-GPU handles such conditions, we conduct two experiments: one with runtime execution time variations and one with the dynamic arrival of a new task. Both experiments demonstrate that FC-GPU promptly reduces deadline violations and quickly adapts to set points.

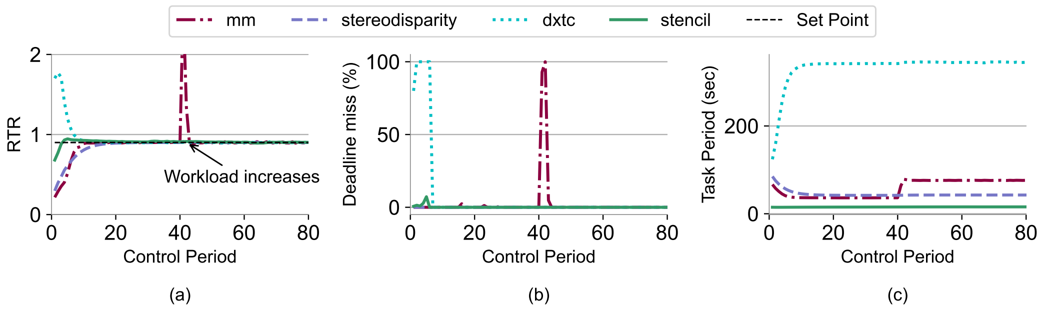


Fig. 10. Online experiments showcasing FC-GPU's responsiveness to dynamic conditions. Figure (a) shows that at control period 40, the input to the first workload increases and the controller quickly adjusts the RTR to its set point. Figure (b) presents the deadline misses for all tasks, with values remaining below 2%. Figure (c) illustrates the variation in the task periods.

5.5.1 Varying Execution Times. In this experiment, we assess the system's performance as execution times fluctuate dynamically during runtime. We focus on the soft real-time video processing system utilized for surveillance purposes [39]. Four tasks, `mm` $(25, 80)$, `stereodisparity` $(25.97, 100)$, `stencil` $(51, 100)$, and `dxtc` $(55.3, 15)$, are released simultaneously at time 0. At control period 60, we increase the input size for `mm`, which incurs additional memory operations. During this memory copy, the GPU effectively runs only three tasks, causing a drop in the RTR. In the next control period, all four tasks resume competition for GPU resources, resulting in a temporary overshoot of `mm`'s RTR beyond 1.5, as displayed in Figure 10(a). Figure 10(b) shows the deadline misses for each task. While `dxtc` initially exceeds the RTR, FC-GPU quickly readjusts their rates to bring the RTR below the set point thresholds. The change in task period during each control period is shown in Figure 10(c). Although increased workload in the control period 40 causes short-term deadline violations, the controller mitigates these violations in subsequent control periods. After stabilization, the deadline-miss ratio remains under 2%.

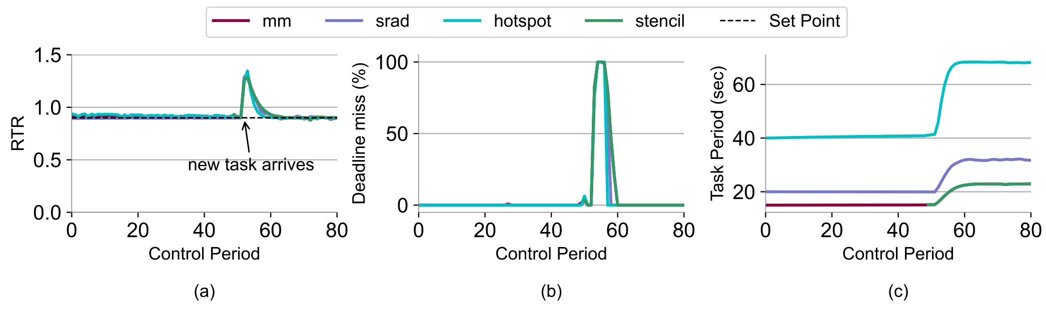


Fig. 11. Online experiments showcasing FC-GPU's responsiveness to the dynamic arrival of new tasks. Figure (a) shows that a new task arrives in the system during the control period, 50, and the controller quickly adjusts the RTR to its set point. Figure (b) presents the deadline misses for all tasks, with values remaining below 2%. Figure (c) illustrates the variation in the task periods.

5.5.2 Dynamic Arrival of Tasks. In real-time systems new tasks can originate from user applications, system events, or processing sensor inputs [67]. For example, in autonomous vehicles, when an unexpected obstacle, such as a traffic cone, appears on the road, a vehicle must react swiftly to prevent collision. Thus, the vehicle may promptly schedule a new task to address obstacles and ensure safe navigation. When this situation occurs, the controller needs to adjust the task rates for all the tasks within a small number of control periods and ensure there are no significant deadline misses. As discussed before, FC-GPU can handle dynamic task arrival or departure by having controller models for different numbers of tasks, and then switch between those controller models at runtime.

We initially run three tasks, `mm` $(25, 50)$, `srad` $(25.97, 285)$, and `hotspot` $(12.6, 21)$, at time 0. So FC-GPU is initially employing the controller designed for 3 tasks to control their RTR. In control period 50, a new task arrives, *stencil* $(51, 51)$. When the new task arrives, we send an interrupt signal to the FC-GPU controller. The controller captures the interrupt signal and reloads the model parameters designed for the total number of tasks in the system. The controller uses gang scheduling [14] to quickly reload the model parameters and begin to control the RTR. As Figure 11(e) reveals, the sudden arrival of `stencil` briefly inflates the RTRs, followed by rapid convergence toward their respective set points. Figure 11(f) shows a similar pattern in deadline misses: Once the fourth task arrives, the DMR increases to 100%, but the controller adjusts the task rates within several control periods, eventually driving the overall DMR below 2%.

6 Conclusion

Existing GPU real-time scheduling solutions are mostly open-loop and rely on WCET. Traditional adaptive solutions, such as feedback control scheduling, are designed for CPU-based real-time tasks. Thus, they cannot be directly applied to GPU, because GPUs have different and more complex architectures and so schedulable utilization bounds cannot apply to GPUs yet. In this paper, we have presented FC-GPU, the first Feedback Control GPU scheduling framework for soft real-time embedded systems. To model the GPU resource contention among tasks, we derive a multi-input-multi-output (MIMO) system model that captures the impacts of task rate adaptation on the response times of different tasks. Our extensive hardware testbed results on both Nvidia and AMD GPUs demonstrate that FC-GPU can provide better real-time performance with much less deadline miss for GPU tasks, even when the task execution times significantly increase at runtime.

7 Acknowledgements

This work was supported, in part, by the U.S. NSF under Grant CNS-2344505. We would also like to thank the anonymous reviewers for their valuable comments.

References

[1] apollo [n. d.]. Apollo 3.0 Software Architecture. https://tinyurl.com/mhd6dfka.

[2] apollolight [n. d.]. Apollo's Traffic Light Perception. https://github.com/ApolloAuto/apollo /blob/master/docs/specs/traffic_light.md.

[3] Yidi Wang, Mohsen Karimi, Yecheng Xiang, and Hyoseung Kim. [n. d.]. Balancing energy efficiency and real-time performance in GPU scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*. IEEE.

[4] Chenyang Lu, Xiaorui Wang, and Christopher Gill. [n. d.]. Feedback Control Real-Time Scheduling in ORB Middleware. In *Proceedings of 9th IEEE Real-Time and Embedded Technology and Applications Symposium.*

[5] https://github.com/nvidia/cuda samples. [n. d.]. Nvidia Benchmark suite. ([n. d.]).

[6] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* (1973).

[7] Lui Sha, Ragunathan Rajkumar, and Shirish S Sathaye. 1994. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proc. IEEE* (1994).

[8] Giorgio C Buttazzo, Giuseppe Lipari, and Luca Abeni. 1998. Elastic task model for adaptive rate control. In *Proceedings of the IEEE Real-Time Systems Symposium.*

[9] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. 1999. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the 3rd Operating Systems Design and Implementation.*

[10] Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. 2002. Analysis of a Reservation-based Feedback Scheduler. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium.*

[11] Kevin Skadron, Tarek Abdelzaher, and Mircea R. Stan. 2002. 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.*

[12] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. 2002. 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 (July 2002), 85–126.

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

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

[15] Giorgio C. Buttazzo. 2004. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series).* Springer-Verlag TELOS, Santa Clara, CA, USA.

[16] Xiaorui Wang, Dong Jia, Chenyang Lu, and Xenofon Koutsoukos. 2005. Decentralized utilization control in distributed real-time systems. In *26th IEEE International Real-Time Systems Symposium.*

[17] Chenyang Lu, Xiaorui Wang, and Xenofon Koutsoukos. 2005. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Transactions on Parallel and Distributed Systems* (2005).

[18] Xiaorui Wang, Dong Jia, Chenyang Lu, and Xenofon Koutsoukos. 2007. DEUCON: Decentralized End-to-End Utilization Control for Distributed Real-Time Systems. *IEEE Transactions on Parallel and Distributed Systems* (2007).

[19] Marc Lalonde, David Byrns, Langis Gagnon, Normand Teasdale, and Denis Laurendeau. 2007. Real-time eye blink detection with GPU-based SIFT tracking. In *Fourth Canadian Conference on Computer and Robot Vision.* IEEE.

[20] Ming Chen, Xiaorui Wang, Raghul Gunasekaran, Hairong Qi, and Mallikarjun Shankar. 2008. 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.*

[21] Jianguo Yao, Xue Liu, Mingxuan Yuan, and Zonghua Gu. 2008. Online adaptive utilization control for real-time embedded multiprocessor systems. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis.*

[22] Xing Fu, Xiaorui Wang, and Eric Puster. 2009. 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.*

[23] Ming Chen, Xiaorui Wang, and Ben Taylor. 2009. Integrated Control of Matching Delay and CPU Utilization in Information Dissemination Systems. In *Proceedings of the 17th IEEE International Workshop on Quality of Service.*

[24] Xiaorui Wang, Xing Fu, Xue Liu, and Zonghua Gu. 2009. Power-Aware CPU Utilization Control for Distributed Real-Time Systems. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*.

[25] Jun-Sik Kim, Myung Hwangbo, and Takeo Kanade. 2009. Realtime affine-photometric KLT feature tracker on GPU in CUDA framework. In *IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*.

[26] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE.

[27] Yefu Wang, Kai Ma, and Xiaorui Wang. 2009. Temperature-constrained power control for chip multiprocessors with online model estimation. *ACM SIGARCH computer architecture news* (2009).

[28] Sunpyo Hong and Hyesoon Kim. 2010. An integrated GPU power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*.

[29] Can Basaran, Mehmet H. Suzer, Kyoung-Don Kang, and Xue Liu. 2010. Robust fuzzy CPU utilization control for dynamic workloads. *Journal of Systems and Software* (2010).

[30] Xing Fu, Khairul Kabir, and Xiaorui Wang. 2011. Cache-Aware Utilization Control for Energy Efficiency in Multi-Core Real-Time Systems. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*.

[31] Po-Han Wang, Chia-Lin Yang, Yen-Ming Chen, and Yu-Jung Cheng. 2011. Power gating strategies on GPUs. *ACM Transactions on Architecture and Code Optimization* (2011).

[32] Kai Ma, Xue Li, Ming Chen, and Xiaorui Wang. 2011. Scalable power control for many-core architectures running multi-threaded applications. In *Proceedings of the 38th annual international symposium on Computer architecture*.

[33] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, Yutaka Ishikawa, et al. 2011. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *USENIX Annual Technical Conference*.

[34] Xing Fu and Xiaorui Wang. 2011. 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*.

[35] Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. 2012. The case for GPGPU spatial multitasking. In *IEEE International Symposium on High-Performance Comp Architecture*.

[36] Marilyn Wolf. 2012. *Computers as components: principles of embedded computing system design*. Elsevier.

[37] Glenn A Elliott and James H Anderson. 2012. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems* (2012).

[38] Ming Chen, Xiaorui Wang, and Clinton Nolan. 2012. Hierarchical CPU Utilization Control for Real-Time Guarantees in Power Grid Computing. *Journal of Real-Time Systems* 48, 2 (Jan. 2012), 198–221.

[39] Yu Wang and Jien Kato. 2012. Integrated pedestrian detection and localization using stereo cameras. *Digital Signal Processing for In-Vehicle Systems and Safety* (2012).

[40] Glenn A Elliott, Bryan C Ward, and James H Anderson. 2013. GPUSync: A framework for real-time GPU management. In *IEEE 34th Real-Time Systems Symposium*.

[41] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling preemptive multiprogramming on GPUs. *ACM SIGARCH Computer Architecture News* (2014).

[42] Paula Aguilera, Katherine Morrow, and Nam Sung Kim. 2014. QoS-aware dynamic resource allocation for spatial-multitasking GPUs. In *19th Asia and South Pacific Design Automation Conference*. IEEE.

[43] Husheng Zhou, Guangmo Tong, and Cong Liu. 2015. GPES: A preemptive execution system for GPGPU computing. In *IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE.

[44] Hassan Gomaa. 2016. *Real-time software design for embedded systems*. Cambridge University Press.

[45] Haeseung Lee and Mohammad Abdullah Al Faruque. 2016. Run-time scheduling framework for event-driven applications on a GPU-based embedded system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016).

[46] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. Effisha: A software framework for enabling effficient preemptive scheduling of gpu. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[47] Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H Anderson, F Donelson Smith, Alex Berg, and Shige Wang. 2017. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *Real-Time and Embedded Technology and Applications Symposium*. IEEE.

[48] Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. 2017. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *IEEE Real-Time Systems Symposium (RTSS)*. IEEE.

[49] Nathan Otterness, Ming Yang, Tanya Amert, James Anderson, and F Donelson Smith. 2017. Inferring the scheduling policies of an embedded CUDA GPU. In *Workshop on Operating Systems Platforms for Embedded Real Time Systems Applications (OSPERT)*.

[50] Hyoseung Kim, Pratyush Patel, Shige Wang, and Ragunathan Raj Rajkumar. 2017. A server-based approach for predictable GPU access control. In *IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE.

[51] Vladislav Golyanik, Mitra Nasri, and Didier Stricker. 2017. Towards scheduling hard real-time image processing tasks on a single GPU. In *IEEE International Conference on Image Processing (ICIP)*. IEEE.

[52] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. 2018. Deadline-based scheduling for GPU with preemption support. In *IEEE Real-Time Systems Symposium (RTSS)*. IEEE.

[53] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. 2019. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *IEEE Real-Time and Embedded Technology and Applications Symposium*.

[54] Sujan Kumar Saha, Yecheng Xiang, and Hyoseung Kim. 2019. STGM: Spatio-temporal GPU management for real-time tasks. In *International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE.

[55] Yunhao Bai, Zejiang Wang, Xiaorui Wang, and Junmin Wang. 2020. AutoE2E: End-to-end real-time middleware for autonomous driving control. In *IEEE 40th International Conference on Distributed Computing Systems*.

[56] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. 2020. Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[57] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast pipelined context switching for deep learning applications. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*.

[58] Jinghao Sun, Jing Li, Zhishan Guo, An Zou, Xuan Zhang, Kunal Agrawal, and Sanjoy Baruah. 2020. Real-time scheduling upon a host-centric acceleration architecture with data offloading. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[59] Woosung Kang, Kilho Lee, Jinkyu Lee, Insik Shin, and Hoon Sung Chwa. 2021. LaLaRAND: Flexible Layer-by-Layer CPU/GPU Scheduling for Real-Time DNN Tasks. In *IEEE Real-Time Systems Symposium (RTSS)*. IEEE.

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

[61] Guin Gilman and Robert J Walls. 2022. Characterizing concurrency mechanisms for NVIDIA GPUs under deep learning workloads. *ACM SIGMETRICS Performance Evaluation Review* (2022).

[62] Ionel Gog, Sukrit Kalra, Peter Schafhalter, Joseph E. Gonzalez, and Ion Stoica. 2022. D3: A Dynamic Deadline-Driven Approach for Building Autonomous Vehicles. In *Proceedings of the Seventeenth European Conference on Computer Systems*.

[63] Wei Gao, Qinghao Hu, Zhisheng Ye, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. 2022. Deep Learning Workload Scheduling in GPU Datacenters: Taxonomy, Challenges and Vision. *arXiv preprint arXiv:2205.11913* (2022).

[64] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*.

[65] Yunhao Bai, Li Li, Zejiang Wang, Xiaorui Wang, and Junmin Wang. 2022. Performance optimization of autonomous driving control under end-to-end deadlines. *Real-Time Systems* (2022).

[66] Yidi Wang, Mohsen Karimi, and Hyoseung Kim. 2022. Towards energy-efficient real-time scheduling of heterogeneous multi-gpu systems. In *IEEE Real-Time Systems Symposium*.

[67] Jinghao Sun, Kailu Duan, Xisheng Li, Nan Guan, Zhishan Guo, Qingxu Deng, and Guozhen Tan. 2023. Real-Time Scheduling of Autonomous Driving System with Guaranteed Timing Correctness. In *IEEE 29th Real-Time and Embedded Technology and Applications Symposium*.

[68] Guoyu Chen, Srinivasan Subramaniyan, and Xiaorui Wang. 2024. Enabling Latency-guaranteed Co-location of Inference and Training for Reducing Data Center Expenses. In *44th International Conference on Distributed Computing Systems*. IEEE.

[69] Yidi Wang, Cong Liu, Daniel Wong, and Hyoseung Kim. 2024. GCAPS: GPU Context-Aware Preemptive Priority-Based Scheduling for Real-Time Tasks. In *36th Euromicro Conference on Real-Time Systems*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.