A Server-based Approach for Predictable GPU Access with Improved Analysis

Hyoseung Kim[1], Pratyush Patel[2], Shige Wang[3], and Ragunathan (Raj) Rajkumar[4]

[1]University of California, Riverside , hyoseung@ucr.edu

[2]Carnegie Mellon University , pratyusp@andrew.cmu.edu

[3]General Motors R&D , shige.wang@gm.com

[4]Carnegie Mellon University , rajkumar@cmu.edu

Abstract

We propose a server-based approach to manage a general-purpose graphics processing unit (GPU) in a predictable and efficient manner. Our proposed approach introduces a GPU server that is a dedicated task to handle GPU requests from other tasks on their behalf. The GPU server ensures bounded time to access the GPU, and allows other tasks to suspend during their GPU computation to save CPU cycles. By doing so, we address the two major limitations of the existing real-time synchronization-based GPU management approach: busy waiting within critical sections and long priority inversion. We have implemented a prototype of the server-based approach on a real embedded platform. This case study demonstrates the practicality and effectiveness of the server-based approach. Experimental results indicate that the server-based approach yields significant improvements in task schedulability over the existing synchronization-based approach in most practical settings. Although we focus on a GPU in this paper, the server-based approach can also be used for other types of computational accelerators.

1 Introduction

The high computational demands of complex algorithmic tasks used in recent embedded and cyber-physical systems pose substantial challenges in guaranteeing their timeliness. For example, a self-driving car [28, 40] executes perception and motion planning algorithms in addition to running tasks for data fusion from tens of sensors equipped within the vehicle. Since these tasks are computationally intensive, it becomes hard to satisfy their timing requirements when they execute on the same hardware platform. Fortunately, many of today's embedded multi-core processors, such as NXP i.MX6 [4] and NVIDIA TX1/TX2 [3], have an on-chip, general-purpose graphics processing

unit (GPU), which can greatly help in addressing the timing challenges of computation-intensive tasks by accelerating their execution.

The use of GPUs in a time predictable manner brings up several challenges. First, many of today's commercial-off-the-shelf (COTS) GPUs do not support a preemption mechanism, and GPU access requests from application tasks are handled in a sequential, non-preemptive manner. This is primarily due to the high overhead expected on GPU context switching [39]. Although some recent GPU architectures, such as NVIDIA Pascal [2], claim to offer GPU preemption, there is no documentation regarding their explicit behavior, and existing drivers (and GPU programming APIs) do not offer any programmer control over GPU preemption at the time of writing this paper. Second, COTS GPU device drivers do not respect task priorities and the scheduling policy used in the system. Hence, in the worst case, the GPU access request of the highest-priority task may be delayed by the requests of all lower-priority tasks in the system, which could possibly cause unbounded priority inversion.

The aforementioned issues have motivated the development of predictable GPU management techniques to ensure task timing constraints while achieving performance improvement [15, 16, 17, 20, 21, 27, 41]. Among them, the work in [15, 16, 17] introduces a *synchronization-based approach* that models GPUs as mutually-exclusive resources and uses real-time synchronization protocols to arbitrate GPU access. This approach has many benefits. First, it can schedule GPU requests from tasks in an analyzable manner, without making any change to GPU device drivers. Second, it allows the existing task schedulability analysis methods, originally developed for real-time synchronization protocols, to be easily applied to analyze tasks accessing GPUs. However, due to the underlying assumption on critical sections, this approach requires tasks to busy-wait during the entire GPU execution, thereby resulting in substantial CPU utilization loss. Note that semaphore-based locks also experience this problem as the busy waiting occurs *within* a critical section after acquiring the lock. Also, the use of real-time synchronization protocols for GPUs may unnecessarily delay the execution of high-priority tasks due to the priority-boosting mechanism employed in some protocols, such as MPCP [37] and FMLP+ [10]. We will review these issues in Section 4.2.

In this paper, we develop a *server-based approach* for predictable GPU access control to address the aforementioned limitations of the existing synchronization-based approach. Our proposed approach introduces a dedicated GPU server task that receives GPU access requests from other tasks and handles the requests on their behalf. Unlike the synchronization-based approach, the server-based approach allows tasks to suspend during GPU computation while preserving analyzability. This not only yields CPU utilization benefits, but also reduces task response times. We present the schedulability analysis of tasks under our server-based approach, which accounts for the overhead of the GPU server. Although we have focused on a GPU in this work, our approach can be used

for other types of computational accelerators, such as a digital signal processor (DSP).

We have implemented a prototype of our approach on a SABRE Lite embedded platform [1] equipped with four ARM Cortex-A9 CPUs and one Vivante GC2000 GPU. Our case study using this implementation with the workzone recognition algorithm [31] developed for a self-driving car demonstrates the practicality and effectiveness of our approach in improving CPU utilization and reducing response time. We have also conducted detailed experiments on task schedulability. Experimental results show that while our server-based approach does not dominate the synchronization-based approach, it outperforms the latter in most of the practical cases.

This paper is an extended version of our conference paper [25], with the following new contributions: (i) an improved analysis for GPU request handling time under the server-based approach, (ii) a discussion on task allocation with the GPU server, and (iii) new experimental results with the improved analysis.

The rest of this paper is organized as follows: Section 2 reviews relevant prior work. Section 3 describes our system model. Section 4 reviews the use of the synchronization-based approach for GPU access control and discusses its limitations. Section 5 presents our proposed server-based approach. Section 6 evaluates the approach using a practical case study and overhead measurements, along with detailed schedulabilty experiments. Section 7 concludes the paper.

2 Related Work

Many techniques have been developed to utilize a GPU as a predictable, shared computing resource. TimeGraph [21] is a real-time GPU scheduler that schedules GPU access requests from tasks with respect to task priorities. This is done by modifying an open-source GPU device driver and monitoring GPU commands at the driver level. RGEM [20] allows splitting a long data-copy operation into smaller chunks, reducing blocking time on data-copy operations. Gdev [22] provides common APIs to both user-level tasks and the OS kernel to use a GPU as a standard computing resource. GPES [41] is a software technique to break a long GPU execution segment into smaller sub-segments, allowing preemptions at the boundaries of sub-segments. While all these techniques can mitigate the limitations of today's GPU hardware and device drivers, they have not considered the schedulability of tasks using the GPU. In other words, they handle GPU requests from tasks in a predictable manner, but do not formally analyze the worst-case timing behavior of tasks on the CPU side, which is addressed in this paper.

Elliott et al. [15, 16, 17] modeled GPUs as mutually-exclusive resources and proposed the use of real-time synchronization protocols for accessing GPUs. Based on this, they developed GPUSync [17], a software framework for GPU management in multi-core real-time systems. GPUSync

supports both fixed- and dynamic-priority scheduling policies, and provides various features, such as budget enforcement, multi-GPU support, and clustered scheduling. It uses separate locks for copy and execution engines of GPUs to enable overlapping of GPU data transmission and computation. The pros and cons of the synchronization-based approach in general will be thoroughly discussed in Section 4.

Server-based software architectures for GPU access control have been studied in the context of GPU virtualization, such as rCUDA [14] and vCUDA [38]. They implement an RPC server to receive GPU acceleration requests from high-performance computing applications running on remote or virtual machines. The received requests are then handled by the server on a host machine equipped with a GPU. While our work uses a similar software architecture to these approaches, our work differs by providing guaranteed bounds on the worst-case GPU access time.

The self-suspension behavior of tasks has been studied in the context of real-time systems [7, 8, 13, 27]. This is motivated by the fact that tasks can suspend while accessing hardware accelerators like GPUs. Kim et al. [27] proposed *segment-fixed priority scheduling*, which assigns different priorities and phase offsets to each segment of tasks. They developed several heuristics for priority and offset assignment because finding the optimal solution for that assignment is NP-hard in the strong sense. Chen et al. [13] reported errors in existing self-suspension analyses and presented corrections for the errors. Those approaches assume that the duration of self-suspension is given as a fixed task parameter. However, this assumption does not comply with the case where a task accesses a shared GPU and the waiting time for the GPU is affected by other tasks in the system. In this work, we use the results in [8, 13] to take into account the effect of self-suspension in task schedulability, and propose techniques to bound the worst-case access time to a shared GPU.

Recently, there have been efforts to understand the details of GPU-internal scheduling on some NVIDIA GPUs. Otterness et al. [35] report that multiple GPU execution requests from different processes may be scheduled concurrently on a single NVIDIA TX1 GPU but the execution time of individual requests becomes longer and less predictive compared to when they run sequentially. Amert et al. [6] discuss the internal scheduling policy of NVIDIA TX2 for GPU execution requests made by threads sharing the same address space, i.e., those belonging to the same process. While these efforts have potential to improve GPU utilization and overall system efficiency, it is unclear whether their findings are valid on other GPU architectures and other types of accelerators. Although we do not consider these recently-found results in this work as our goal is to develop a broadly-applicable solution, we plan to consider them in our future work.

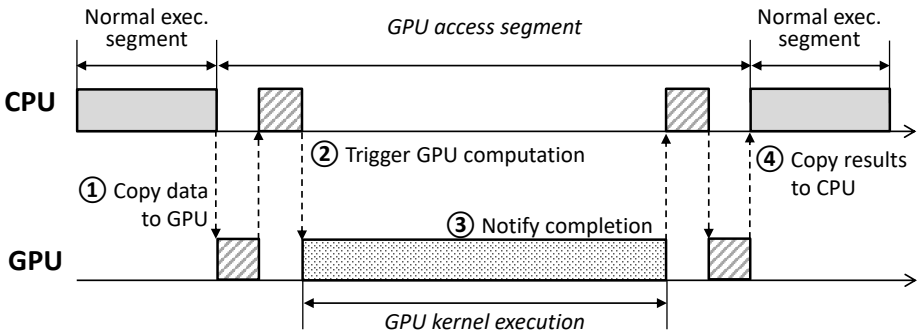


Figure 1: Execution pattern of a task accessing a GPU

3 System Model

The work in this paper assumes a multi-core platform equipped with a single general-purpose GPU device.[1] The GPU is shared among multiple tasks, and GPU requests from tasks are handled in a sequential, non-preemptive manner. The GPU has its own memory region, which is assumed to be sufficient enough for the tasks under consideration. We do not assume the concurrent execution of GPU requests from different tasks, called *GPU co-scheduling*, because recent work [35] reports that "co-scheduled GPU programs from different programs are not truly concurrent, but are multiprogrammed instead" and "this (co-scheduling) may lead to slower or less predictable total times in individual programs".

We consider sporadic tasks with constrained deadlines. The execution time of a task using a GPU is decomposed into *normal execution segments* and *GPU access segments*. Normal execution segments run entirely on CPU cores and GPU access segments involve GPU operations. Figure 1 depicts an example of a task having a single GPU access segment. In the GPU access segment, the task first copies data needed for GPU computation, from CPU memory to GPU memory (Step ① in Figure 1). This is typically done using Direct Memory Access (DMA), which requires no (or minimal) CPU intervention. The task then triggers the actual GPU computation, also referred to as *GPU kernel execution*, and waits for the GPU computation to finish (Step ②). The task is notified when the GPU computation finishes (Step ③), and it copies the results back from the GPU to the CPU (Step ④). Finally, the task continues its normal execution segment. Note that during the time when CPU intervention is not required, e.g., during data copies with DMA and GPU kernel execution, the task may suspend or busy-wait, depending on the implementation of the GPU device driver and the configuration used.

Synchronous and Asynchronous GPU Access. The example in Figure 1 uses *synchronous mode* for GPU access, where each GPU command, such as memory copy and GPU kernel execution,

[1]This assumption reflects today's GPU-enabled embedded processors, e.g., NXP i.MX6 [4] and NVIDIA TX1/TX2 [3]. A multi-GPU platform would be used for future real-time embedded and cyber-physical systems, but extending our work to handle multiple GPUs remains as future work.

can be issued only after the prior GPU command has finished. However, many GPU programming interfaces, such as CUDA and OpenCL, also provide *asynchronous mode*, which allows a task to overlap CPU and GPU computations. For example, if a task sends all GPU commands in asynchronous mode at the beginning of the GPU access segment, then it can either perform other CPU operations within the same GPU access segment or simply wait until all the GPU commands complete. Hence, while the sequence of GPU access remains the same, the use of asynchronous mode can affect the amount of active CPU time in a GPU segment.

Task Model. A task τ_i is characterized as follows:

$$\tau_i := (C_i, T_i, D_i, G_i, \eta_i)$$

- C_i: the sum of the worst-case execution time (WCET) of all normal execution segments of task τ_i.
- T_i: the minimum inter-arrival time of each job of τ_i.
- D_i: the relative deadline of each job of τ_i ($D_i \leq T_i$).
- G_i: the maximum accumulated duration of all GPU segments of τ_i, when there is no other task competing for a GPU.
- η_i: the number of GPU access segments in each job of τ_i.

The utilization of a task τ_i is defined as $U_i = (C_i + G_i)/T_i$. Parameters C_i and G_i can be obtained by either measurement-based or static-analysis tools. If a measurement-based approach is used, C_i and G_i can be measured when τ_i executes alone in the system without any other interfering tasks and they need to be conservatively estimated. A task using a GPU has one or more GPU access segments. We use $G_{i,j}$ to denote the maximum duration of the j-th GPU access segment of τ_i, i.e., $G_i = \sum_{j=1}^{\eta_i} G_{i,j}$. Parameter $G_{i,j}$ can be decomposed as follows:

$$G_{i,j} := (G^e_{i,j}, G^m_{i,j})$$

- $G^e_{i,j}$: the WCET of pure GPU operations that do not require CPU intervention in the j-th GPU access segment of τ_i.
- $G^m_{i,j}$: the WCET of miscellaneous operations that require CPU intervention in the j-th GPU access segment of τ_i.

$G^e_{i,j}$ includes the time for GPU kernel execution, and $G^m_{i,j}$ includes the time for copying data, launching the kernel, notifying the completion of GPU commands, and executing other CPU operations.

The cost of triggering self-suspension during CPU-inactive time in a GPU segment is assumed to be taken into account by $G^m_{i,j}$. If data are copied to or from the GPU using DMA, only the time for issuing the copy command is included in $G^m_{i,j}$; the time for actual data transmission by DMA is modeled as part of $G^e_{i,j}$, as it does not require CPU intervention. Note that $G_{i,j} \leq G^e_{i,j} + G^m_{i,j}$ because $G^e_{i,j}$ and $G^m_{i,j}$ are not necessarily observed on the same control path and they may overlap in asynchronous mode.

CPU Scheduling. In this work, we focus on *partitioned fixed-priority preemptive task scheduling* due to the following reasons: *(i)* it is widely supported in many commercial real-time embedded OSs such as OKL4 [18] and QNX RTOS [5], and *(ii)* it does not introduce task migration costs. Thus, each task is statically assigned to a single CPU core. Any fixed-priority assignment, such as Rate-Monotonic [32] can be used for tasks. Each task τ_i is assumed to have a unique priority π_i. An arbitrary tie-breaking rule can be used to achieve this assumption under fixed-priority scheduling.

4 Limitations of Synchronization-based GPU Access Control

In this section, we characterize the limitations of using a real-time synchronization protocol for tasks accessing a GPU on a multi-core platform. We consider semaphore-based synchronization, where a task attempting to acquire a lock suspends if the lock is already held by another task. This is because semaphores are more efficient than spinlocks for long critical sections [29], which is the case for GPU access.

4.1 Overview

The synchronization-based approach models the GPU as a mutually-exclusive resource and the GPU access segments of tasks as critical sections. A single semaphore-based mutex is used for protecting such GPU critical sections. Hence, under the synchronization-based approach, a task should hold the GPU mutex to enter its GPU access segment. If the mutex is already held by another task, the task is inserted into the waiting list of the mutex and suspends until the mutex can be held by that task. Some implementations like GPUSync [17] use separate locks for internal resources of the GPU, e.g., copy and execution engines, to achieve parallelism in GPU access, but we focus on a single lock for the entire GPU for simplicity.

Among a variety of real-time synchronization protocols, we consider the Multiprocessor Priority Ceiling Protocol (MPCP) [36, 37] as a representative, because it works for partitioned fixed-priority scheduling that we use in our work and it has been widely referenced in the literature. We shall briefly review the definition of MPCP below. More details on MPCP can be found in [30, 36, 37].

1. When a task τ_i requests an access to a resource R_k, it can be granted to τ_i if it is not held by another task.

2. While a task τ_i is holding a resource that is shared among tasks assigned to different cores, the priority of τ_i is raised to $\pi_B + \pi_i$, where π_B is a base task-priority level greater than that of any task in the system, and π_i is the normal priority of τ_i. This "priority boosting" under MPCP is referred to as the *global priority ceiling* of τ_i.

3. When a task τ_i requests access to a resource R_k that is already held by another task, τ_i is inserted to the waiting list of the mutex for R_k and suspends.

4. When a resource R_k is released and the waiting list of the mutex for R_k is not empty, the highest-priority task in the waiting list is dequeued and granted R_k.

4.2 Limitations

As described in Section 3, each GPU access segment contains various operations, including data copies, notifications, and GPU computation. Specifically, a task may suspend when CPU intervention is not required, e.g., during GPU kernel execution, to save CPU cycles. However, under the synchronization-based approach, any task in its GPU access segment (i.e., running within a critical section after acquiring the lock) should "busy-wait" for any operation conducted on the GPU in order to ensure timing predictability. This is because most real-time synchronization protocols and their analyses, such as MPCP [37], FMLP [9], FMLP+ [10] and OMLP [12], assume that *(i)* a critical section is executed entirely on the CPU, and *(ii)* there is no suspension during the execution of the critical section. Hence, during its GPU kernel execution, the task is not allowed to suspend even when no CPU intervention is required.[2] For example, while the GPUSync implementation [17] can be configured to suspend instead of busy-wait during GPU kernel execution at runtime, its analysis uses a suspension-oblivious approach that does not incorporate benefits from self-suspension during GPU execution. As the time for GPU kernel execution and data transmission by DMA increases, the CPU time loss under the synchronization-based approach is therefore expected to increase. The analyses of those protocols could possibly be modified to allow suspension within critical sections, at the potential expense of increased pessimism.[3]

Some synchronization protocols, such as MPCP [37], FMLP [9] and FMLP+ [11], use priority boosting (either restricted or unrestricted) as a progress mechanism to prevent unbounded priority

[2]These protocols allow suspension while waiting for lock acquisition.

[3]The analyses for MPCP [29] and FMLP+ [10] under partitioned scheduling show that, whenever a task resumes from suspension, it may experience priority inversion from local tasks with higher boosted priorities. Hence, allowing self-suspension within critical sections under these protocols will likely increase chances of priority inversion and requires a change in the task model to limit the number of suspensions. Note that this however does not affect the asymptotic optimality of FMLP+, as reported in [11].

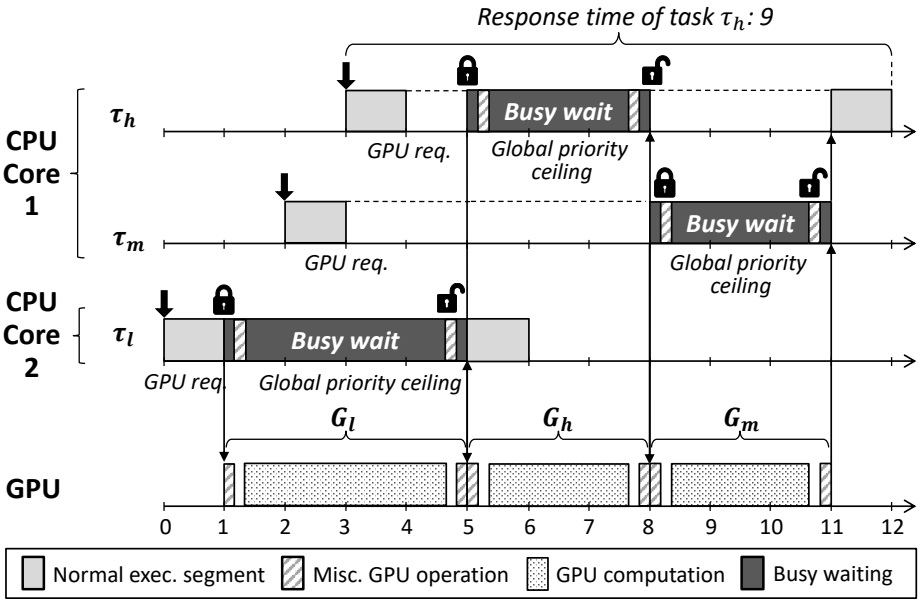


Figure 2: Example schedule of GPU-using tasks under the synchronization-based approach with MPCP

inversion. However, the use of priority boosting could cause another problem we call "long priority inversion". We describe this problem with the example in Figure 2. There are three tasks, τ_h, τ_m, and τ_l, that have high, medium, and low priorities, respectively. Each task has one GPU access segment that is protected by MPCP and executed between two normal execution segments. τ_h and τ_m are allocated to Core 1, and τ_l is allocated to Core 2.

Task τ_l is released at time 0 and makes a GPU request at time 1. Since there is no other task using the GPU at that point, τ_l acquires the mutex for the GPU and enters its GPU access segment. τ_l then executes with the global priority ceiling associated with the mutex (priority boosting). Note that while the GPU kernel of τ_l is executed, τ_l also consumes CPU cycles due to the busy-waiting requirement of the synchronization-based approach. Tasks τ_m and τ_h are released at time 2 and 3, respectively. They make GPU requests at time 3 and 4, but the GPU cannot be granted to either of them because it is already held by τ_l. Hence, τ_m and τ_h suspend. At time 5, τ_l releases the GPU mutex, and τ_h acquires the mutex next because it has higher priority than τ_m. At time 8, τ_h finishes its GPU segment, releases the mutex, and restores its normal priority. Next, τ_m acquires the mutex, and preempts the normal execution segment of τ_h because the global priority ceiling of the mutex is higher than τ_h's normal priority. Hence, although the majority of τ_m's GPU segment merely performs busy-waiting, the execution of the normal segment of τ_h is delayed until the GPU access segment of τ_m finishes. Finally, τ_h completes its normal execution segment at time 12, making the response time of τ_h 9 in this example.

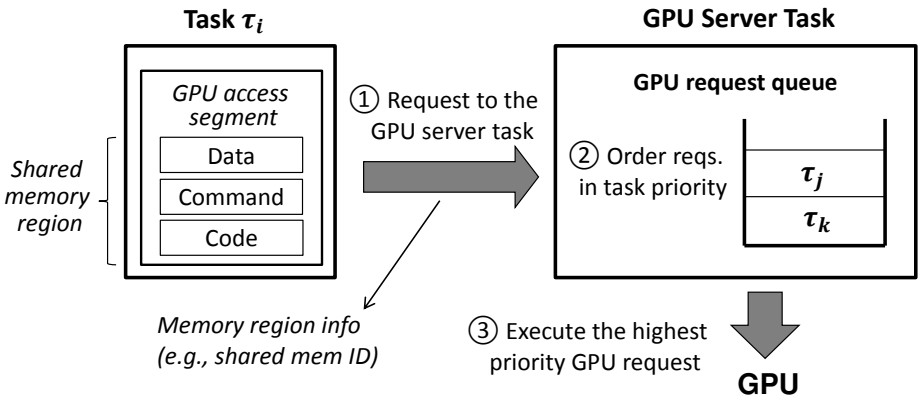


Figure 3: GPU access procedure under our server-based approach

5 Server-based GPU Access Control

We present our server-based approach for predictable GPU access control. This approach addresses the two main limitations of the synchronization-based approach, namely, busy waiting and long priority inversion.

5.1 GPU Server Design

Our server-based approach creates a *GPU server* task that handles GPU access requests from other tasks on their behalf. The GPU server is assigned the highest priority in the system, which is to prevent preemptions by other tasks. Figure 3 shows the sequence of GPU request handling under our server-based approach. First, when a task τ_i enters its GPU access segment, it makes a GPU access request to the GPU server, not to the GPU device driver. The request is made by sending the memory region information for the GPU access segment, including input/output data, commands and code for GPU kernel execution to the server task. This requires the memory regions to be configured as shared regions so that the GPU server can access them with their identifiers, e.g., `shmid`. After sending the request to the server, τ_i suspends, allowing other tasks to execute. Secondly, the server enqueues the received request into the GPU request queue, if the GPU is being used by another request. The GPU request queue is a priority queue, where elements are ordered in their task priorities. Thirdly, once the GPU becomes free, the server dequeues a request from the head of the queue and executes the corresponding GPU segment. During CPU-inactive time, e.g., data copy with DMA and GPU kernel execution, the server suspends to save CPU cycles in synchronous mode. In asynchronous mode, the server performs remaining miscellaneous CPU operations of the request being handled, i.e., asynchronous execution part, and suspends after completing them. This approach allows the GPU server to offer the same level of parallelism as

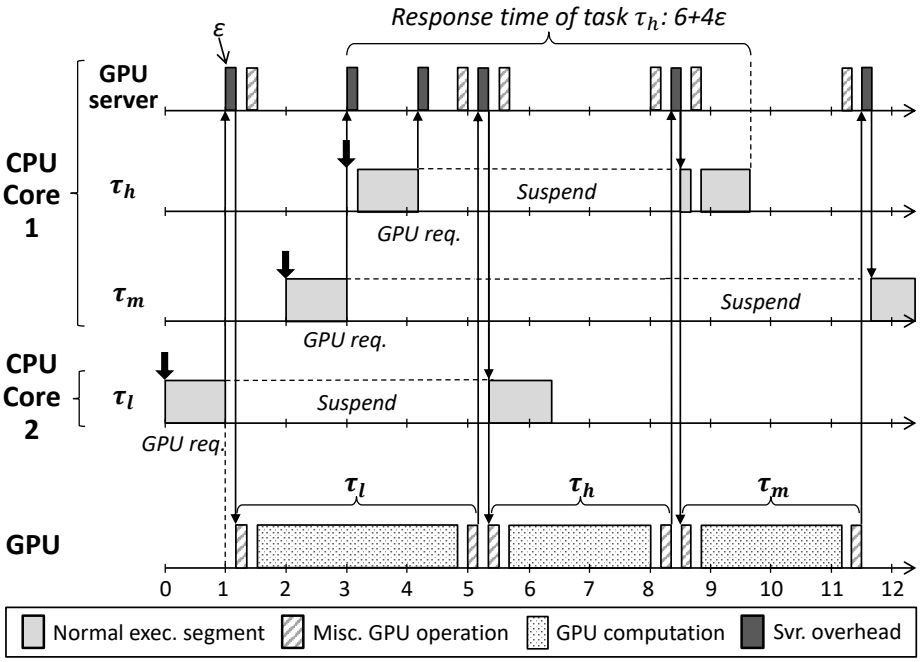


Figure 4: Example schedule under our server-based approach

in the asynchronous mode of the synchronization-based approach.[4] When the request finishes, the server notifies the completion of the request and wakes up the task τ_i. Finally, τ_i resumes its execution.

The use of the GPU server task inevitably introduces the following additional computational costs: *(i)* sending a GPU request to the server and waking up the server task, *(ii)* enqueueing the request and checking the request queue to find the highest-priority GPU request, and *(iii)* notifying the completion of the request to the corresponding task. We use the term ϵ to characterize the GPU server overhead that upper-bounds these computational costs.

Figure 4 shows an example of task scheduling under our server-based approach. This example has the same configuration as the one in Figure 2 and uses synchronous mode for GPU access.[5] The GPU server, which the server-based approach creates, is allocated to Core 1. Each GPU access segment has two sub-segments of miscellaneous operations, each of which is assumed to amount to ϵ.

At time 1, the task τ_l makes a GPU access request to the server task. The server receives the request and executes the corresponding GPU access segment at time $1 + \epsilon$, where ϵ represents the overhead of the GPU server. Since the server-based approach does not require tasks to busy-wait,

[4]Note that achieving parallelism beyond this level is not in the scope of this paper. For example, if one wants to develop a task utilizing more than one CPU cores during its GPU execution, the corresponding GPU segment needs to be programmed with multithreads, which is different from our task model and needs additional research efforts to guarantee real-time predictability.

[5]The GPU server supports both synchronous and asynchronous GPU access.

τ_l suspends until the completion of its GPU request. The GPU request of τ_m at time 3 is enqueued into the request queue of the server. As the server executes with the highest priority in the system, it delays the execution of τ_h released at time 3 by ϵ. Hence, τ_h starts execution at time $3+\epsilon$ and makes a GPU request at time $4+\epsilon$. When the GPU access segment of τ_l completes, the server task is notified. The server then notifies the completion of the GPU request to τ_l and wakes it up, and subsequently executes the GPU access segment of τ_h at time $5+2\epsilon$. The task τ_h suspends until its GPU request finishes. The GPU access segment of τ_h finishes at time $8+2\epsilon$ and that of τ_m starts at time $8+3\epsilon$. Unlike the case under the synchronization-based approach, τ_h can continue to execute its normal execution segment from time $8+3\epsilon$, because τ_m suspends and the priority of τ_m is not boosted. The task τ_h finishes its normal execution segment at time $9+4\epsilon$, and hence, the response time of τ_h is $6+4\epsilon$. Recall that the response time of τ_h is 9 for the same taskset under the synchronization-based approach, as shown in Figure 2. Therefore, we can conclude that the server-based approach provides a shorter response time than the synchronization-based approach for this example taskset, if the value of ϵ is under 3/4 time units, which, as measured in Section 6.2, is a very pessimistic value for ϵ.

5.2 Schedulability Analysis

We analyze task schedulability under our server-based approach. Since all the GPU requests of tasks are handled by the GPU server, we first identify the GPU request handling time for the server. The following properties hold for the server-based approach:

Lemma 1. *The GPU server task imposes up to 2ϵ of extra CPU time on each GPU request.*

Proof. As the GPU server intervenes before and after the execution of each GPU segment, each GPU request can cause at most 2ϵ of overhead in the worst case. Note that the cost of issuing the GPU request as well as triggering self-suspension within the j-th GPU segment of τ_i is already taken into account by $G^m_{i,j}$, as described in Section 3. ■

When a task τ_i makes a GPU request, the GPU may be already handling a request from another task. Since the GPU executes in a non-preemptive manner, τ_i's request must wait for the completion of the currently-handled request. There may also be multiple pending requests that are prioritized over τ_i's request, and it has to wait for the completion of such requests. As a result, τ_i experiences *waiting time* for GPU access, which we define as follows:

Definition 1. *The waiting time for a GPU access segment of a task τ_i is the interval between the time when that segment is released and the time when it begins execution.*

Lemma 2. *The maximum handling time of all GPU requests of a single job of a task τ_i by the GPU server is given as follows:*

$$B_i^{gpu} = \begin{cases} B_i^w + G_i + 2\eta_i\epsilon & : \eta_i > 0 \\ 0 & : \eta_i = 0 \end{cases} \tag{1}$$

where η_i is the number of GPU access segments of τ_i and B_i^w is an upper bound on the total waiting time for all η_i GPU segments of τ_i.

Proof. If $\eta_i > 0$, the j-th GPU request of τ_i is handled after some waiting time, and then takes $G_{i,j} + 2\epsilon$ to complete the execution (by Lemma 1). Hence, the maximum handling time of all GPU requests of τ_i is $B_i^w + \sum_{j=1}^{\eta_i}(G_{i,j} + 2\epsilon) = B_i^w + G_i + 2\eta_i\epsilon$. If $\eta_i = 0$, B_i^{gpu} is obviously zero. ■

In order to tightly upper-bound the total waiting time B_i^w, we adopt the *double-bounding approach* used in [23, 24]. The double-bounding approach consists of two analysis methods: *request-driven* and *job-driven*. The request-driven analysis focuses on the worst-case waiting time that can be imposed on each request of τ_i, and computes an upper bound on the total waiting time by adding up all the per-request waiting time. On the other hand, the job-driven analysis focuses on the amount of waiting time that can be possibly given by other interfering tasks during τ_i's job execution, and takes the maximum of this value as an upper bound of the total waiting time. These two analyses do not dominate one another. Hence, the double-bounding approach takes the minimum of the two to provide a tighter upper bound. The total waiting time B_i^w for all GPU requests of a task τ_i is therefore:

$$B_i^w = \min(B_i^{rd}, B_i^{jd}) \tag{2}$$

where B_i^{rd} and B_i^{jd} are the results obtained by the request-driven and job-driven analyses, respectively.[6] Note that B_i^{rd} is the sum of the maximum waiting time for each GPU request of τ_i, i.e., $B_i^{rd} = \sum_{1 \le j \le \eta_i} B_{i,j}^{rd}$.

Lemma 3. [Request-driven analysis] *The maximum waiting time for the j-th GPU segment of a task τ_i under the server-based approach is bounded by the following recurrence:*

$$B_{i,j}^{rd,n+1} = \max_{\pi_l < \pi_i \wedge 1 \le k \le \eta_l} (G_{l,k} + \epsilon) + \sum_{\pi_h > \pi_i \wedge 1 \le k \le \eta_h} \left(\left\lceil \frac{B_{i,j}^{rd,n}}{T_h} \right\rceil + 1 \right) (G_{h,k} + \epsilon) \tag{3}$$

where $G_{l,k}$ is the maximum duration of the k-th GPU segment of τ_l and $B_{i,j}^{rd,0} = \max_{\pi_l < \pi_i \wedge 1 \le k \le \eta_l}(G_{l,k} + \epsilon)$ (the first term of the equation).

Proof. When τ_i, the task under analysis, makes a GPU request, the GPU may be handling a

[6] Our initial work in [25] uses only the request-driven analysis to bound the total waiting time, i.e., $B_i^w = B_i^{rd}$.

request from a lower-priority task, which τ_i has to wait for completion due to the non-preemptive nature of the GPU. Hence, the longest GPU access segment from all lower-priority tasks needs to be considered as the waiting time in the worst case. Here, only one ϵ of overhead is caused by the GPU server because other GPU requests will be immediately followed and the GPU server needs to be invoked only once between two consecutive GPU requests, as depicted in Figure 4. This is captured by the first term of the equation.

During the waiting time of $B_{i,j}^{rd}$, higher-priority tasks can make GPU requests to the server. As there can be at most one carry-in request from each higher-priority task τ_h during $B_{i,j}^{rd}$, the maximum number of GPU requests made by τ_h is bounded by $\sum_{u=1}^{\eta_h}(\lceil B_{i,j}^{rd}/T_h \rceil + 1)$. Multiplying each element of this summation by $G_{h,k} + \epsilon$ therefore gives the maximum waiting time caused by the GPU requests of τ_h, which is exactly used as the second term of the equation. ■

Lemma 4. [Job-driven analysis] *The maximum waiting time given by the GPU requests of other tasks during a single job execution of a task τ_i under the server-based approach is given by:*

$$B_i^{jd} = \eta_i \cdot \max_{\pi_l < \pi_i \wedge 1 \le k \le \eta_l} (G_{l,k} + \epsilon) + \sum_{\pi_h > \pi_i \wedge 1 \le k \le \eta_h} \left(\left\lceil \frac{W_i}{T_h} \right\rceil + 1 \right) (G_{h,k} + \epsilon) \tag{4}$$

where W_i is the response time of τ_i.

Proof. During the execution of any job of τ_i, in the worst case, every GPU request of τ_i may wait for the completion of the longest lower-priority GPU request, which is captured by the first term of the equation. In addition, due to the priority queue of the GPU server, higher-priority GPU requests made during the response time of τ_i (W_i) can introduce waiting time to the GPU requests of a single job of τ_i. The second term of the equation upper-bounds the maximum waiting time given by higher-priority GPU requests during W_i, and it can be proved in the same manner as in the proof of Lemma 3. ■

The response time of a task τ_i is affected by the presence of the GPU server on τ_i's core. If τ_i is allocated on a different core than the GPU server, the worst-case response time of τ_i under the server-based approach is given by:

$$W_i^{n+1} = C_i + B_i^{gpu} + \sum_{\tau_h \in \mathbb{P}(\tau_i) \wedge \pi_h > \pi_i} \left\lceil \frac{W_i^n + (W_h - C_h)}{T_h} \right\rceil C_h \tag{5}$$

where $\mathbb{P}(\tau_i)$ is the CPU core on which τ_i is allocated. The recurrence computation terminates when $W_i^{n+1} = W_i^n$, and τ_i is schedulable if $W_i^n \le D_i$. To compute B_i^{jd} at the n-th iteration, W_i^{n-1} can be used in Eq. (4). It is worth noting that, as captured in the third term, the GPU segments of higher-priority tasks do *not* cause any direct interference to the normal execution segments of τ_i because they are executed by the GPU server that runs on a different core.

If τ_i is allocated on the same core as the GPU server, the worst-case response time of τ_i is given by:

$$\begin{aligned} W_i^{n+1} =& C_i + B_i^{gpu} + \sum_{\tau_h \in \mathbb{P}(\tau_i) \wedge \pi_h > \pi_i} \left\lceil \frac{W_i^n + (W_h - C_h)}{T_h} \right\rceil C_h \\ &+ \sum_{\tau_j \neq \tau_i \wedge \eta_j > 0} \left\lceil \frac{W_i^n + \{D_j - (G_j^m + 2\eta_j \epsilon)\}}{T_j} \right\rceil (G_j^m + 2\eta_j \epsilon) \end{aligned} \tag{6}$$

where G_j^m is the sum of the WCETs of miscellaneous operations in τ_j's GPU access segments, i.e., $G_j^m = \sum_{k=1}^{\eta_j} G_{j,k}^m$.

Under the server-based approach, both, the GPU-using tasks, as well as the GPU server task, can self-suspend. Hence, we use the following lemma given by Bletsas et al. [8] to prove Eqs. (5) and (6):

Lemma 5. [from [8]] *The worst-case response time of a self-suspending task τ_i is upper-bounded by:*

$$W_i^{n+1} = C_i + \sum_{\tau_h \in \mathbb{P}(\tau_i) \wedge \pi_h > \pi_i} \left\lceil \frac{W_i^n + (W_h - C_h)}{T_h} \right\rceil C_h \tag{7}$$

Note that D_h can be used instead of W_h in the summation term of Eq. (7) [13].

Theorem 1. *The worst-case response time of a task τ_i under the server-based approach is given by Eqs.* (5) *and* (6).

Proof. To account for the maximum GPU request handling time of τ_i, B_i^{gpu} is added in both Eqs. (5) and (6). In the ceiling function of the third term of both equations, $(W_h - C_h)$ accounts for the self-suspending effect of higher-priority GPU-using tasks (by Lemma 5). With these, Eq.(5) upper-bounds the worst-case response time of τ_i when it is allocated on a different core than the GPU server.

The main difference between Eq. (6) and Eq. (5) is the last term, which captures the worst-case interference from the GPU server task. The execution time of the GPU server task is bounded by summing up the worst-case miscellaneous operations and the server overhead caused by GPU requests from all other tasks ($G_j^m + 2\eta_j \epsilon$). Since the GPU server self-suspends during CPU-inactive time intervals, adding $\{D_j - (G_j^m + 2\eta_j \epsilon)\}$ to W_i^n in the ceiling function captures the worst-case self-suspending effect (by Lemma 5). These factors are exactly captured by the last term of Eq. (6). Hence, it upper-bounds task response time in the presence of the GPU server. ■

5.3 Task Allocation with the GPU Server

As shown in Eqs. (6) and (5), the response time of a task τ_i with the server-based approach is affected by the presence of the GPU server on τ_i's core. This implies that, in order to achieve better schedulability, the GPU server should be considered when allocating tasks to cores.

Under partitioned scheduling, finding the optimal task allocation can be modeled as the bin-packing problem, which is known to be NP-complete [19]. Hence, bin-packing heuristics, such as first-fit decreasing and worst-fit decreasing, have been widely used as practical solutions to the task allocation problem. Such heuristics first sort tasks in decreasing order of utilization and then allocate them to cores. In case of the GPU server, its utilization depends on the frequency and length of miscellaneous operations of GPU access segments. Specifically, the utilization of the GPU server is given by:

$$U_{server} = \sum_{\forall \tau_i : \eta_i > 0} \frac{G_i^m + 2\eta_i \epsilon}{T_i} \tag{8}$$

With this utilization, the GPU server can be sorted and allocated together with other regular tasks using conventional bin-packing heuristics. We will use this approach for schedulability experiments in Section 6.3.

6 Evaluation

This section provides our experimental evaluation of the two different approaches for GPU access control. We first present details about our implementation and describe case study results on a real embedded platform. Next, we explore the impact of these approaches on task schedulability with randomly-generated tasksets, by using parameters based on the practical overheads measured from our implementation.

6.1 Implementation

We implemented prototypes of the synchronization-based and the server-based approaches on a SABRE Lite board [1]. The board is equipped with an NXP i.MX6 Quad SoC that has four ARM Cortex-A9 cores and one Vivante GC2000 GPU. We ran an NXP Embedded Linux kernel version 3.14.52 patched with Linux/RK [34] version 1.6[7], and used the Vivante v5.0.11p7.4 GPU driver along with OpenCL 1.1 (Embedded Profile) for general-purpose GPU programming. We also configured each core to run at its maximum frequency, 1 GHz.

Linux/RK provides a kernel-level implementation of MPCP which we used to implement the

[7]Linux/RK is available at `http://rtml.ece.cmu.edu/redmine/projects/rk/`.

synchronization-based approach. Under our implementation, each GPU-using task first acquires an MPCP-based lock, issues memory copy and GPU kernel execution requests in an asynchronous manner, and uses OpenCL events to busy-wait on the CPU till the GPU operation completes, before finally releasing the lock.

To implement the server-based approach, we set up shared memory regions between the server task and each GPU-using task, which are used to share GPU input/output data. POSIX signals are used by the GPU-using tasks and the server to notify GPU requests and completions, respectively. The server has an initialization phase, during which, it initializes shared memory regions and obtains GPU kernels from the GPU binaries (or source code) of each task. Subsequently, the server uses these GPU kernels whenever the corresponding task issues a GPU request. As the GPU driver allows suspensions during GPU requests, the server task issues memory copy and GPU kernel execution requests in an asynchronous manner, and suspends by calling the `clFinish()` API function provided by OpenCL.

GPU Driver and Task-Specific Threads. The OpenCL implementation on the i.MX6 platform spawns user-level threads in order to handle GPU requests and to notify completions. Under the synchronization-based approach, OpenCL spawns multiple such threads for each GPU-using task, whereas under the server-based approach, such threads are spawned only for the server task. To eliminate possible scheduling interference, the GPU driver process, as well as the spawned OpenCL threads, are configured to run at the highest real-time priority in all our experiments.

6.2 Practical Evaluation

Overheads. We measured the practical worst-case overheads for both the approaches in order to perform schedulability analysis. Each worst-case overhead measurement involved examining 100,000 readings of the respective operations measured on the i.MX6 platform. Figure 5 shows the mean and 99.9th percentile of the MPCP lock operations and Figure 6 shows the same for server related overheads.

Under the synchronization-based approach with MPCP, overhead occurs while acquiring and releasing the GPU lock. Under the server-based approach, overheads involve waking up the server task, performing priority queue operations (i.e., server execution delay), and notifying completion to wake up the GPU-using task after finishing GPU computation.

To safely take into consideration the worst-case overheads, we use the 99.9th percentile measurements for each source of delay in our experiments. This amounts to a total of 14.0 μs lock-related delay under the synchronization-based approach, and a total of 44.97 μs delay for the server task under the server-based approach.

Case Study. We present a case study motivated by the software system of the self-driving car devel-

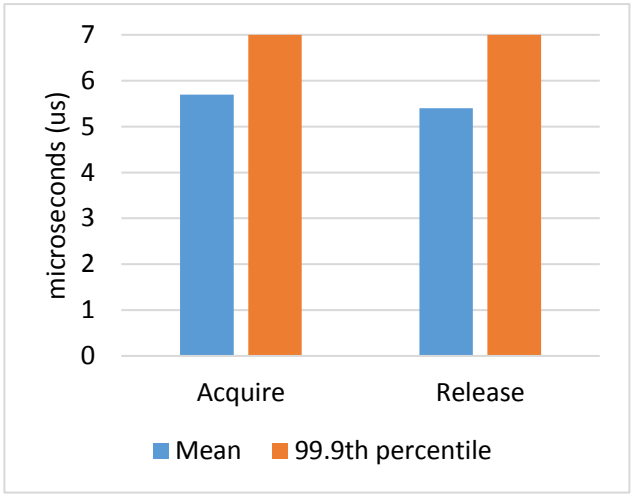


Figure 5: MPCP lock overhead

Table 1: Tasks used in the case study

Task τ_i	**Task name**	C_i **(in ms)**	η_i	G_i **(in ms)**	$T_i = D_i$ **(in ms)**	**Core**	**Priority**
τ_1	`workzone`	20	2	$G_{1,1} = 95$, $G_{1,2} = 47$	300	0	70
τ_2	`cpu_matmul1`	215	0	0	750	0	67
τ_3	`cpu_matmul2`	102	0	0	300	1	69
τ_4	`gpu_matmul1`	0.15	1	$G_{4,1} = 19$	600	1	68
τ_5	`gpu_matmul2`	0.15	1	$G_{5,1} = 38$	1000	1	66

oped at CMU [40]. Among various algorithmic tasks of the car, we chose a GPU accelerated version of the workzone recognition algorithm [31] (`workzone`) that periodically processes images collected from a camera. Two GPU-based matrix multiplication tasks (`gpu_matmul1` and `gpu_matmul2`), and two CPU-bound tasks (`cpu_matmul1` and `cpu_matmul2`) are also used to represent a subset of other tasks of the car. Unique task priorities are assigned based on the Rate-Monotonic policy [32] and each task is pinned to a specific core as described in Table 1. Of these, the `workzone` task has two GPU segments per job whereas both the GPU-based matrix multiplication tasks have a single GPU segment. Under the server-based approach, the `server` task is pinned to CPU Core 1 and is run with real-time priority 80. In order to avoid unnecessary interference while recording traces, the task-specific OpenCL threads are pinned on a separate CPU core for both approaches. All tasks are released at the same time using Linux/RK APIs. CPU scheduling is performed using the `SCHED_FIFO` policy, and CPU execution traces are collected for one hyperperiod (3,000 ms) as shown in Figure 7.

The CPU-execution traces for the synchronization and server-based approaches are shown in Figure 7(a) and Figure 7(b), respectively. Tasks executing on Core 0 are shown in blue whereas tasks executing on Core 1 are shown in red. It is immediately clear that the server-based approach allows suspension of tasks while they are waiting for the GPU request to complete. In particular,

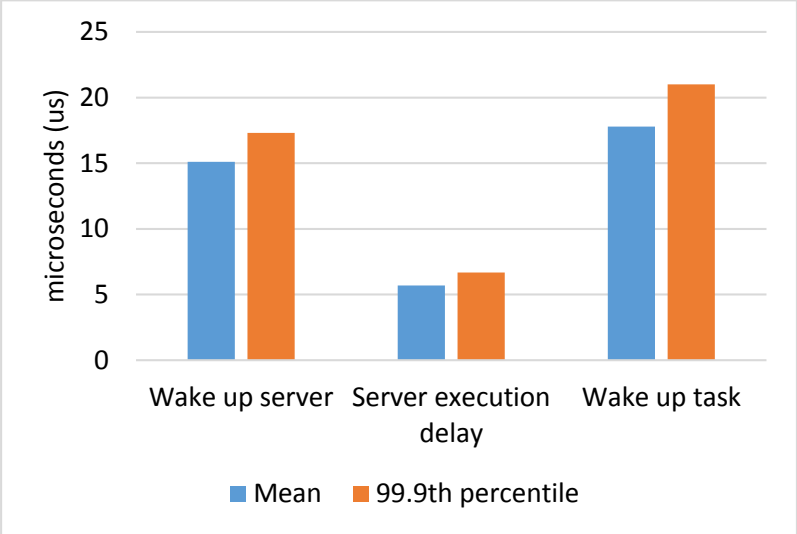


Figure 6: Server task overheads

Table 2: Base parameters for taskset generation

Parameters	**Values**
Number of CPU cores (N_P)	4, 8
Number of tasks (n)	[$2N_P$, $5N_P$]
Task utilization (U_i)	[0.05, 0.2]
Task period and deadline ($T_i = D_i$)	[30, 500] ms
Percentage of GPU-using tasks	[10, 30] %
Ratio of GPU segment len. to normal WCET (G_i/C_i)	[10, 30] %
Number of GPU segments per task (η_i)	[1, 3]
Ratio of misc. operations in $G_{i,j}$ ($G^m_{i,j}/G_{i,j}$)	[10, 20] %
GPU server overhead (ϵ)	50 μs

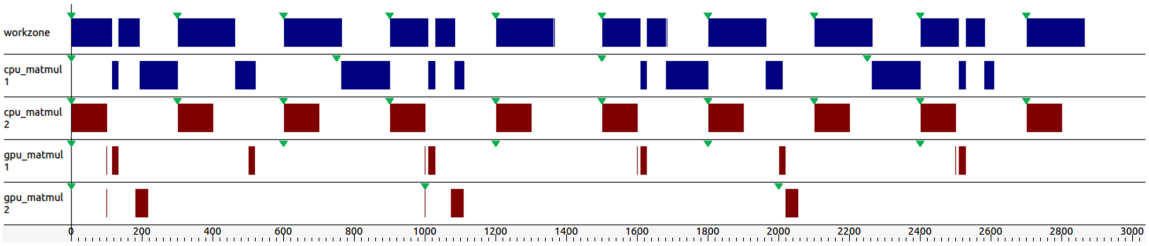
we make the following key observations from the case study:

1. Under the synchronization-based approach, tasks suspend when they wait for the GPU lock to be released, but they do not suspend while using the GPU. On the contrary, under the server-based approach, tasks suspend even when their GPU segments are being executed. The results show that our proposed server-based approach can be successfully implemented and used on a real platform.

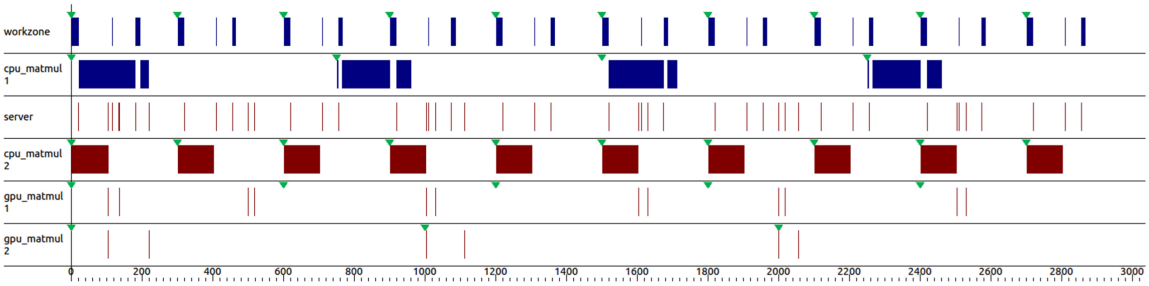
2. The response time of `cpu_matmul1` under the synchronization-based approach is significantly larger than that under the server-based approach, i.e., 520.68 ms vs. 219.09 ms in the worst case, because of the busy-waiting problem discussed in Section 4.2.

6.3 Schedulability Experiments

Taskset Generation. We used 10,000 randomly-generated tasksets for each experimental setting. Unless otherwise mentioned, the base parameters given in Table 2 are used for taskset generation.



(a) Synchronization-based Approach (MPCP)



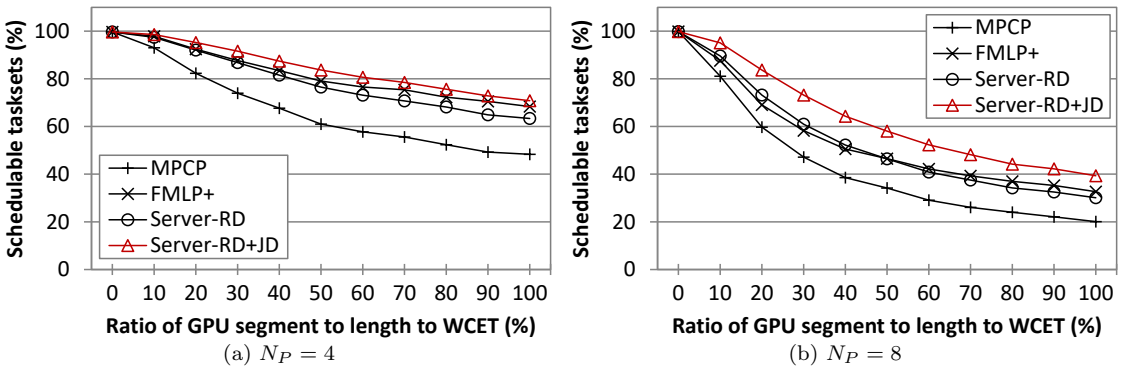
(b) Server-based Approach

Figure 7: Task execution timeline during one hyperperiod (3,000 ms)

The GPU-related parameters are inspired from the observations from our case study and the GPU workloads used in prior work [21, 22]. Other task parameters are similar to those used in the literature. Systems with four and eight CPU cores ($N_P = \{4, 8\}$) are considered. For each taskset, n tasks are generated where n is uniformly distributed over $[2N_P, 5N_P]$. A subset of the generated tasks is chosen at random, corresponding to the specified percentage of GPU-using tasks, to include GPU segments. Task period T_i and utilization U_i are randomly selected from the defined minimum and maximum period ranges. Task deadline D_i is set equal to T_i. Recall that the utilization of a task τ_i is defined as $U_i = (C_i + G_i)/T_i$. If τ_i is a CPU-only task, C_i is set to $U_i \cdot T_i$ and G_i is set to zero. If τ_i is a GPU-using task, the given ratio of the accumulated GPU segment length to the WCET of normal segments is used to determine the values of C_i and G_i. G_i is then split into η_i random-sized pieces, where η_i is the number of τ_i's GPU segments chosen randomly from the specified range. For each GPU segment $G_{i,j}$, the values of $G^e_{i,j}$ and $G^m_{i,j}$ are determined by the ratio of miscellaneous operations given in Table 2, assuming $G_{i,j} = G^e_{i,j} + G^m_{i,j}$. Finally, task priorities are assigned by the Rate-Monotonic policy [32], with arbitrary tie-breaking.

Results. We captured the percentage of schedulable tasksets where all tasks met their deadlines. For the synchronization-based approach, two multiprocessor locking protocols are used: `MPCP` and `FMLP+`. Schedulability under MPCP and FMLP+ is tested using the analysis developed by Lakshmanan et al. [29] and Brandenburg [10][8], respectively. Both analyses are appropriately modified based on the correction by Chen et al. [13]. We considered both zero and non-zero locking over-

[8]We used the FMLP+ analysis for preemptive partitioned fixed-priority scheduling given in Section 6.4.3 of [10].



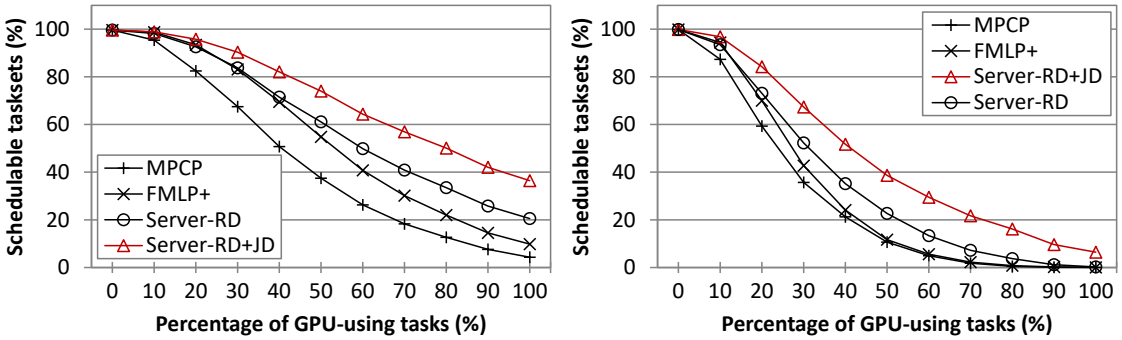
(a) $N_P = 4$ (b) $N_P = 8$

Figure 8: Schedulability w.r.t. the GPU segment length

head under the synchronization-based approach, but there was no appreciable difference between them. Hence, only the results with zero overhead are presented in the paper. Tasks are allocated to cores by using the worst-fit decreasing (WFD) heuristic in order to balance the load across cores. Under the server-based approach, the GPU server is allocated along with other tasks by WFD. Eqs. (5) and (6) are used for task schedulability tests. We set the GPU server overhead ϵ to 50 μs, which is slightly larger than the measured overheads from our implementation presented in Section 6.2. Two versions of the GPU server are considered to show the analytical improvement of this work: `Server-RD`, which is our initial work [25] using only the request-driven approach, and `Server-RD+JD`, which is the one newly proposed in this paper. Unless otherwise mentioned, we will limit our focus to Server-RD+JD and call it as the server-based approach.

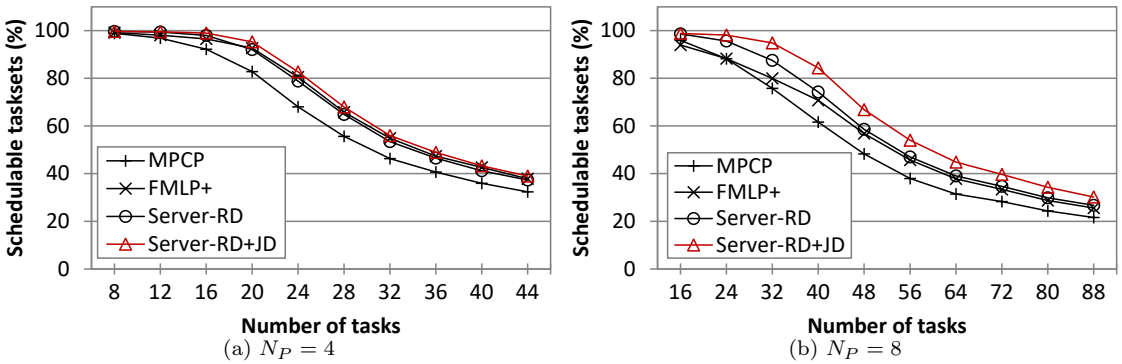
Figure 8 shows the percentage of schedulable tasksets as the ratio of the accumulated GPU segment length (G_i) increases. In general, the percentage of schedulable tasksets is higher when $N_P = 4$, compared to when $N_P = 8$. This is because the GPU is contended for by more tasks as the number of cores increases. The server-based approach outperforms both of the synchronization-based approaches (MPCP and FMLP+) in all cases of Figure 8. This is mainly due to the fact that the server-based approach allows other tasks to use the CPU while the GPU is being used. While MPCP generally performs worse than FMLP+ in our experiments, we suspect that this is due to the pessimism in the analysis [29] that computes an upper bound by the sum of the maximum per-request delay, similarly to the request-driven analysis shown in Eq. 3. If it is extended with the job-driven analysis like Eq. 4, its performance will likely be improved.

Figure 9 shows the percentage of schedulable tasksets as the percentage of GPU-using tasks increases. The left-most point on the x-axis represents that all tasks are CPU-only tasks, and the right-most point represents that all tasks access the GPU. Under all approaches, the percentage of schedulable tasksets reduces as the percentage of GPU-using tasks increases. However, the server-based approach significantly outperforms the other approaches, with as much as 38% and 27%



(a) $N_P = 4$ (b) $N_P = 8$

Figure 9: Schedulability w.r.t. the percentage of GPU-using tasks



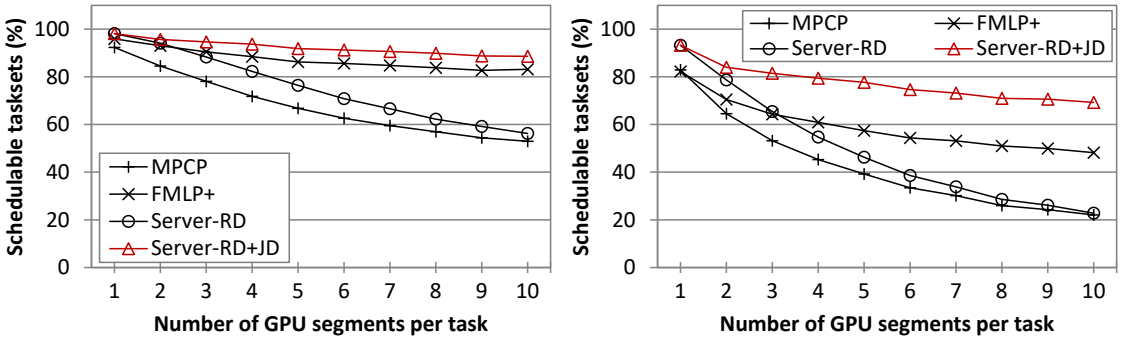
(a) $N_P = 4$ (b) $N_P = 8$

Figure 10: Schedulability w.r.t. the number of tasks

more tasksets being schedulable than MPCP and FMLP+, respectively, when the percentage of GPU-using tasks is 70% and $N_P = 4$.

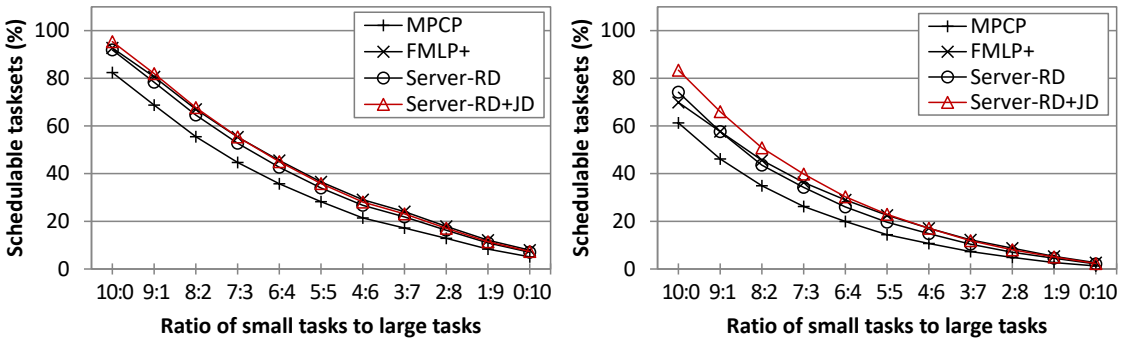
The benefit of the server-based approach is also observed with changes in other task parameters. In Figure 10, the percentage of schedulable tasksets is illustrated as the number of tasks increases. While the difference in schedulability between the server and FMLP+ is rather small when $N_P = 4$, it becomes larger when $N_P = 8$. This happens because the total amount of the CPU-inactive time in GPU segments increases as more tasks are considered. A similar observation is also seen in Figure 11, where the number of GPU access segments per task varies. Here, the different is much noticeable as more GPU segments create larger CPU-inactive time, which is favorable to the server-based approach.

The base task utilization given in Table 2 follows a uniform distribution. In Figure 12, we evaluate with bimodal distributions where the utilization of small tasks is chosen from $[0.05, 0.2]$ and that of large tasks is chosen from $[0.2, 0.5]$. The x-axis of each sub-figure in Figure 12 shows the ratio of small tasks to large tasks in each taskset. While the server-based approach has higher schedulability for fewer large tasks, the gap gets smaller and the percentage of schedulable tasksets



(a) $N_P = 4$ (b) $N_P = 8$

Figure 11: Schedulability w.r.t. the number of GPU segments



(a) $N_P = 4$ (b) $N_P = 8$

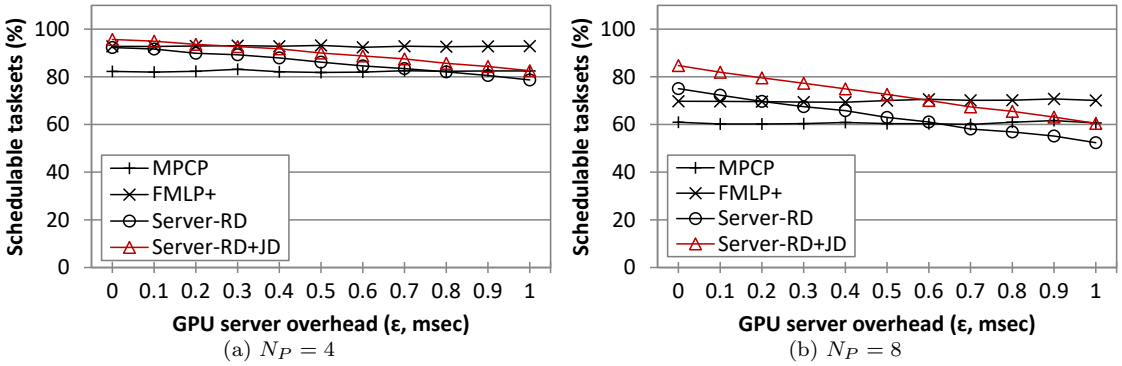
Figure 12: Schedulability w.r.t. the ratio of small tasks to large tasks

under all approaches goes down to zero as the ratio of large tasks increases, due to the high resulting utilization of generated tasksets.

We next investigate the factors that negatively impact the performance of the server-based approach. The GPU server overhead ϵ is obviously one such factor. Although an ϵ of 50 μs that we used in prior experiments is sufficient enough to upper-bound the GPU server overhead in most practical systems, we further investigate with larger ϵ values. Figure 13 shows the percentage of schedulable tasksets as the GPU server overhead ϵ increases. Since ϵ exists only under the server-based approach, the performance of MPCP and FMLP+ is unaffected by this factor.[9] On the other hand, the performance of the server-based approach deteriorates as the overhead increases.

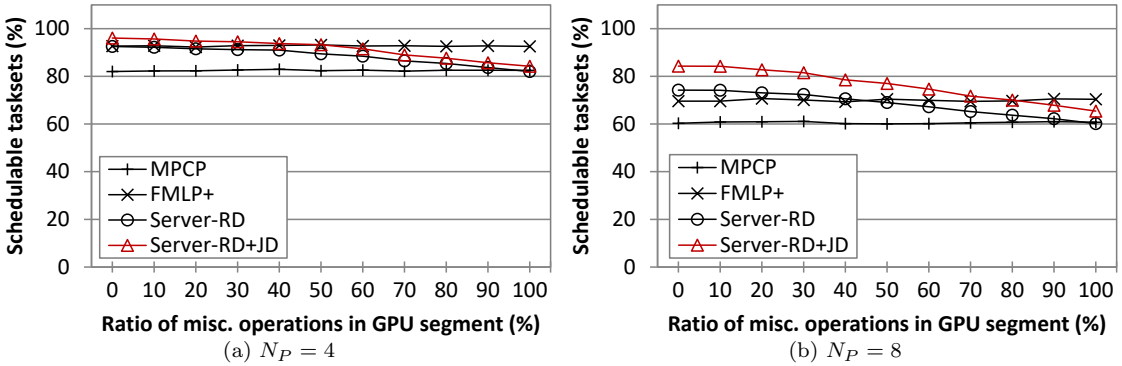
The length of miscellaneous operations in GPU access segments is another factor degrading the performance of the server-based approach because miscellaneous operations require the GPU server to consume a longer CPU time. Figure 14 shows the percentage of schedulable tasksets as the ratio of miscellaneous operations in GPU access segments increases. As the analyses of MPCP

[9]Only a small fluctuation in schedulability of synchronization-based approaches ($< 1\%$) exists due to the nature of random parameters.



(a) $N_P = 4$ (b) $N_P = 8$

Figure 13: Schedulability w.r.t. the GPU server overhead (ϵ)

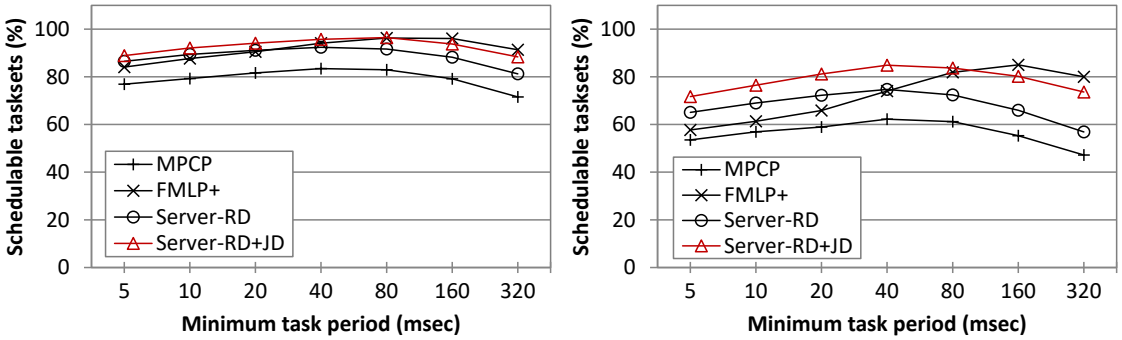


(a) $N_P = 4$ (b) $N_P = 8$

Figure 14: Schedulability w.r.t. the ratio of miscellaneous operations in GPU access segments

and FMLP+ require tasks to busy-wait during their entire GPU access, their performance remains unaffected. On the other hand, as expected, the performance of the server-based approach degrades as the ratio of miscellaneous operations increases. Specifically, starting from the ratio of 60% at $N_P = 4$ and 90% at $N_P = 8$, the server-based approach underperforms FMLP+. However, we expect that such a high ratio of miscellaneous operations in GPU segments is hardly observable in practical GPU applications because memory copy is typically done by DMA and GPU kernel execution takes the majority time of GPU segments.

Lastly, we evaluate the impact of task periods in schedulability. Recall that task periods are uniformly chosen from $[T_{min}, T_{max}]$, where $T_{min} = 20$ ms and $T_{max} = 500$ ms in our base parameters. In this experiment, we fix T_{max} to 500 ms and vary the value of T_{min}. The results are shown in Figure 15. Interestingly, the server-based approach outperforms FMLP+ until T_{min} reaches 80ms at $N_P = 4$ and 160 ms at $N_P = 8$, but it starts to underperform afterwards. We suspect that this is mainly due to the rate-monotonic (RM) priority assignment used in our experiments and the request ordering policies of the GPU server and FMLP+. The priority-based ordering of the GPU server favors higher-priority tasks and the FIFO ordering of FMLP+ ensures fairness in



(a) $N_P = 4$ (b) $N_P = 8$

Figure 15: Schedulability w.r.t. the minimum task period

waiting time. If there is a large difference in task periods, it is better to give shorter waiting time to higher-priority tasks as they have shorter periods (and implicit deadlines) under RM. On the other hand, if the difference in task periods is relatively small, fair waiting time for tasks is likely to improve overall schedulability. There has been a long debate on priority-based and FIFO ordering in the literature. Further discussion on this issue is beyond the scope of the paper, and we leave the extension of the GPU server with FIFO ordering as part of future work.

In summary, the server-based approach outperforms the synchronization-based approach using MPCP and FMLP+ in most cases where realistic parameters are used. Specifically, the benefit of the server-based approach is significant when the percentage of GPU-using tasks is high, the number of GPU segments per tasks is large. However, we do find that the server-based approach does not dominate the synchronization-based approach. The synchronization-based approach may result in better schedulability than the server-based approach when the GPU server overhead or the ratio of miscellaneous operations in GPU segments is beyond the range of practical values (under MPCP and FMLP+) and the range of task period is relatively small (under FMLP+).

7 Conclusions

In this paper, we have presented a server-based approach for predictable CPU access control in real-time embedded and cyber-physical systems. It is motivated by the limitations of the synchronization-based approach, namely busy-waiting and long priority inversion. By introducing a dedicated server task for GPU request handling, the server-based approach addresses those limitations, while ensuring the predictability and analyzability of tasks. The implementation and case study results on an NXP i.MX6 embedded platform indicate that the server-based approach can be implemented with acceptable overhead and performs as expected with a combination of CPU-only and GPU-using tasks. Experimental results show that the server-based approach yields

significant improvements in task schedulability over the synchronization-based approach in most cases. Other types of accelerators such as DSPs and FPGAs can also benefit from our approach.

Our proposed server-based approach offers several interesting directions for future work. First, while we focus on a single GPU in this work, the server-based approach can be extended to multi-GPU and multi-accelerator systems. One possible way is to create a GPU server for each of the GPUs and accelerators, and allocating a subset of GPU-using tasks to each server. The optimization and dynamic adjustment of servers considering task allocation, overhead, and access duration is worthy of investigation. Secondly, the server-based approach can facilitate an efficient co-scheduling of GPU kernels. For instance, the latest NVIDIA GPU architectures can schedule multiple GPU kernels concurrently only if they belong to the same address space [2], and the use of the GPU server satisfies this requirement, just as MPS [33] does. Thirdly, as the GPU server has a central knowledge of all GPU requests, other features like GPU fault tolerance and power management can be developed. Lastly, the server-based approach can be extended to a virtualization environment and compared against virtualization-aware synchronization protocols [26]. We plan to explore these topics in the future.

References

[1] i.MX6 Sabre Lite by Boundary Devices. `https://boundarydevices.com/`.

[2] NVIDIA GP100 Pascal Whitepaper. `http://www.nvidia.com`.

[3] NVIDIA Jetson TX1/TX2 Embedded Platforms. `http://www.nvidia.com`.

[4] NXP i.MX6 Processors. `http://www.nxp.com`.

[5] QNX RTOS. `http://www.qnx.com`.

[6] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[7] N. Audsley and K. Bletsas. Realistic analysis of limited parallel software/hardware implementations. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2004.

[8] K. Bletsas, N. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. Technical Report CISTER-TR-150713, CISTER, 2015.

[9] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2007.

[10] B. Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, University of North Carolina, Chapel Hill, 2011.

[11] B. Brandenburg. The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[12] B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, 17(2):277–342, 2013.

[13] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, and D. Niz. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. Technical Report 854, Department of Computer Science, TU Dortmund, 2016.

[14] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 224–231, 2010.

[15] G. Elliott and J. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1):34–74, 2012.

[16] G. Elliott and J. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. *Real-Time Systems*, 49(2):140–170, 2013.

[17] G. Elliott, B. C. Ward, and J. H. Anderson. GPUSync: A framework for real-time GPU management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[18] General Dynamics. OKL4 Microvisor. `http://ok-labs.com`.

[19] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, 1974.

[20] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[21] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX Annual Technical Conference (ATC)*, 2011.

[22] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt. Gdev: First-class GPU resource management in the operating system. In *USENIX Annual Technical Conference (ATC)*, 2012.

[23] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding and reducing memory interference in COTS-based multi-core systems. *Real-Time Systems*, 52(3):356–395, 2016.

[24] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2014.

[25] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar. A server-based approach for predictable GPU access control. In *IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017.

[26] H. Kim, S. Wang, and R. Rajkumar. vMPCP: A synchronization framework for multi-core virtual machines. In *IEEE Real-Time Systems Symposium (RTSS)*, 2014.

[27] J. Kim, B. Andersson, D. de Niz, and R. Rajkumar. Segment-fixed priority scheduling for self-suspending real-time tasks. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[28] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

[29] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[30] K. Lakshmanan, R. Rajkumar, and J. P. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2009.

[31] J. Lee, Y.-W. Seo, W. Zhang, and D. Wettergreen. Kernel-based traffic sign tracking to improve highway workzone recognition for reliable autonomous driving. In *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, 2013.

[32] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[33] NVIDIA. Sharing a GPU between MPI processes: Multi-process service (MPS) overview. `http://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`.

[34] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in Linux. In *IEEE Real-Time Systems Symposium (RTSS) Work-In-Progress*, 1998.

[35] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2017.

[36] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *International Conference on Distributed Computing Systems (ICDCS)*, 1990.

[37] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, 1988.

[38] L. Shi, H. Chen, J. Sun, and K. Li. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*, 61(6):804–816, 2012.

[39] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *International Symposium on Computer Architecture (ISCA)*, 2014.

[40] J. Wei, J. M. Snider, J. Kim, J. M. Dolan, R. Rajkumar, and B. Litkouhi. Towards a viable autonomous driving research platform. In *IEEE Intelligent Vehicles Symposium (IV)*, 2013.

[41] H. Zhou, G. Tong, and C. Liu. GPES: a preemptive execution system for GPGPU computing. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2015.