Predictable GPU Arbitration for Fixed-Priority Real-Time Systems

Undergraduate Thesis

Submitted in partial fulfillment of the requirements of
BITS F421T Thesis

By

Pratyush Patel
ID No. 2013A7TS012G

Under the supervision of:

Dr. Ragunathan (Raj) Rajkumar
George Westinghouse Professor
Dept. of Electrical & Computer Engineering
Carnegie Mellon University

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, GOA CAMPUS
May 2017

Certificate

This is to certify that the thesis entitled, "*Predictable GPU Arbitration for Fixed-Priority Real-Time Systems*" and submitted by Pratyush Patel ID No. 2013A7TS012G in partial fulfillment of the requirements of BITS F421T Thesis embodies the work done by him under my supervision.

Supervisor
Dr. Ragunathan (Raj) Rajkumar
George Westinghouse Professor
Dept. of Electrical & Computer Engineering
Carnegie Mellon University
Date:

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, GOA CAMPUS

Abstract

Bachelor of Engineering (Hons.) Computer Science

Predictable GPU Arbitration for Fixed-Priority Real-Time Systems

by Pratyush Patel

Modern cyber-physical systems, such as autonomous vehicles, are growing increasingly complex and highly computation-intensive. This is because they must process sensor data from multiple sources in addition to performing several computation algorithms on such data. While multi-core platforms have satisfied this increasing need for computation capability to some extent, they are proving to be insufficient for several modern applications. These challenges have motivated hardware manufacturers to design accelerators such as GPUs and DSPs that can significantly address the heavy computation requirements of specific types of tasks.

Another dimension to cyber-physical systems is the guarantee of safety, as they are closely involved with human life and infrastructure. Often, safety is manifested as temporal correctness, which involves performing actuation in a timely manner. While this topic has already been explored to great detail in the context of uni-core processors, hardware accelerators have not received much attention as their demand has only risen recently.

Motivated by the aforementioned requirements, that is, maintaining temporal correctness while ensuring good performance, in this work, we explore how to make GPUs work in a predictable manner to ensure safety in real-time cyber-physical systems. To this end, we investigate the existing approaches to do so, identify their shortcomings, and propose a new, server-based approach that offers predictable CPU-GPU execution, while still maintaining efficiency. Although we focus solely on GPUs in this work, the same techniques can be also applied to other types of hardware accelerators.

To my parents.

Acknowledgements

Working towards my thesis has been a thrilling expedition; albeit, it is not one that I could have completed on my own. The insights, guidance, and open-minded attitude towards new ideas followed by rigorous and thoughtful discussion, from various people during this journey have been vital to its completion.

First of all, I would like to thank my advisor, Prof. Raj Rajkumar, for accepting me into his team, and for consistently supporting and guiding me through various roadblocks that I encountered during this time. Most importantly, his intuition and advice have refined my attitude to remain curious, yet, creative and scientific, which is indispensable for research. My thanks also go to my mentor, Dr. Björn Brandenburg, who helped kindle this mindset and spark my interest not only in research, but also in the art of communication and presentation, which I greatly value. I am grateful to Prof. Hyoseung Kim, for generously sharing his research experience, and being extremely open with his ideas, which inspired me to try and come up with more of my own.

I also wish to thank my teammates at the Real-Time and Multimedia Systems Laboratory: Anand Bhat, Sandeep D'souza, Shunsuke Aoki, Mengwen He and Iljoo Baek, for not only sparing a significant amount of their time for insightful discussions about research, but also for being wonderful friends that I could truly depend upon. I would particularly like to thank my project partner, Iljoo, whose dedication and trust in our work was a constant source of motivation. In addition, I would like express my gratitude to our collaborators from General Motors, Dr. Shige Wang and Dr. Unmesh Bordoloi, whose experience and feedback were invaluable in shaping our academic work to meet the practical requirements of the industry. I am also deeply thankful to all the friends I made during my time at CyLab and in Pittsburgh; this journey would not have been as enjoyable without you.

Back in India, I would like to thank BITS Pilani for providing me this amazing opportunity to experience an excellent research environment at Carnegie Mellon as an undergraduate student. I am grateful to Prof. Biju Raveendran for having helped nurture my interest in Operating Systems and Architecture, which I still continue to enjoy. I also want to thank my friends and collaborators back at home (and around the world) for their kindness and encouragement during various phases of my life. In particular, I am indebted to my three fellow conspirators at BITS for gleefully initiating our trend of healthy argumentation, thinking outside the box, and working towards things we genuinely enjoy, right from our freshman year, which literally defines a lot of what we (try to) do today.

Finally, I am deeply thankful to my parents for their boundless love, patience and their unwavering trust in the way I chose for myself. This journey would have been impossible without your encouragement, and the freedom with which you allowed me to pursue my interests.

Contents

List of Figures **vii**

List of Tables **viii**

Abbreviations **ix**

1 Introduction **1**
1.1 Thesis Statement 2
1.2 Contributions and Organization 3

2 Background and Prior Work **4**
2.1 Real-Time Scheduling 4
2.1.1 Task Model 4
2.1.2 Priority Assignment for Scheduling 5
2.1.3 Preemptive Multi-Processor Fixed-Priority Task Scheduling 6
2.1.4 Schedulability 7
2.2 GPU Fundamentals 8
2.2.1 GPU Execution Model 9
2.2.2 Synchronous and Asynchronous GPU Access. 9
2.3 Challenges in Real-Time GPU Computing 10
2.4 Prior Work on Predictable GPU Access 11
2.4.1 Schedulability-Unaware Approaches 11
2.4.2 Schedulability-Aware Approaches 12
2.5 Prior Work on Self-Suspension Analysis 12

3 A Server-Based Approach for Predictable GPU Arbitration **13**
3.1 System Model and Scheduling Assumptions 13
3.1.1 Hardware Model 13
3.1.2 Task Model 14
3.1.3 Scheduling Scheme 15
3.2 Limitations of Synchronization-Based GPU Arbitration 15
3.2.1 Overview 15
3.2.2 Schedulability Analysis 16
3.2.3 Limitations 18
3.3 Server-based GPU Arbitration 19
3.3.1 GPU Server Execution Model 19
3.3.2 Schedulability Analysis 22

4 Evaluation **25**
4.1 Implementation 25
4.1.1 GPU Driver and Task-Specific Threads 26
4.2 Practical Evaluation 26
4.2.1 Overheads 26
4.2.2 Case Study: Vision Application 28
4.3 Schedulability Experiments 29
4.3.1 Task Set Generation 29
4.3.2 Schedulability Comparisons 30

5 Conclusion **35**
5.1 Summary of Contributions 35
5.2 Future Work 35

Bibliography **37**

List of Figures

2.1 Execution pattern of a task accessing a GPU 9

3.1 Example schedule of GPU-using tasks under the synchronization-based approach with MPCP 18
3.2 GPU access procedure under our server-based approach 20
3.3 Example schedule under our server-based approach 21

4.1 MPCP lock overhead 27
4.2 Server task overheads 27
4.3 Task execution timeline during one hyperperiod (3,000 ms) 28
4.4 Schedulability results w.r.t. the accumulated GPU segment length 30
4.5 Schedulability results w.r.t. the percentage of GPU-using tasks 31
4.6 Schedulability results w.r.t. the number of tasks per core 31
4.7 Schedulability results w.r.t. the number of GPU segments per task 32
4.8 Schedulability results w.r.t. the task set utilization per core 32
4.9 Schedulability results w.r.t. the overhead of the GPU server (ϵ) 33
4.10 Schedulability results w.r.t. the ratio of misc. operations ($G^m_{i,j}/G^e_{i,j}$) 33

List of Tables

4.1 Tasks used in the case study 27
4.2 Base parameters for GPGPU access control experiments 29

Abbreviations

COTS	**C**ommercial **O**ff-**T**he-**S**helf
CPU	**C**entral **P**rocessing **U**nit
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
DMA	**D**irect **M**emory **A**ccess
DSP	**D**igital **S**ignal **P**rocessor
FMLP	**F**lexible **M**ulti-processor **L**ocking **P**rotocol
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
GP-GPU	**G**eneral-**P**urpose **G**raphics **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
MPCP	**M**ulti-processor **P**riority **C**eiling **P**rotocol
OpenCL	**Open C**omputing **L**anguage
RMS	**R**ate-**M**onotonic **S**cheduling
R2DGLP	**R**eplica-**R**equest **D**onation **G**lobal **L**ocking **P**rotocol
SOC	**S**ystem-**O**n-a-**C**hip
WCET	**W**orst-**C**ase **E**xecution **T**ime
WCRT	**W**orst-**C**ase **R**esponse **T**ime

Chapter 1

Introduction

Real-time systems are characterized by *logical* and *temporal* correctness [41]. While logical correctness is essential to most systems, temporal correctness is typical to systems that interact with the physical world, i.e., cyber-physical systems, where computation is tightly integrated with sensing and actuation at various levels. Common examples include safety-critical embedded applications such as in autonomous vehicles, health care, air traffic control and military defense. A violation of temporal correctness in these domains may therefore lead to compromising safety, which may imply the loss of life and infrastructure. As a result, it is imperative that such systems be designed and engineered for high timing predictability.

To ensure predictability, real-time systems are modeled using mathematical analysis to prove that temporal correctness is maintained. While this may make the system predictable to some degree, based how well the hardware and software components adhere to the analytical model, it is restrictive because these models often over-estimate computation requirements and over-provision resources significantly in order to guarantee timing requirements, which leads to lower system utilization. As numerous real-time systems involve resource-constrained embedded devices, insufficient utilization of available resources is a significant disadvantage.

To further complicate matters, modern cyber-physical systems often collect sensor data from multiple sources and they may have to process each data source in order to perform proper actuation. For instance, a self-driving car [22, 45] executes perception and motion planning algorithms in addition to running tasks for data fusion from multiple sensors. Since these tasks are computationally intensive, it becomes exceedingly difficult to satisfy their timing requirements on CPU-only hardware platforms.

This problem of increasing processor demand has motivated hardware manufacturers to develop computing devices with hardware accelerators such as Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), etc., for faster computation of specific types of tasks (such as perception algorithms). Recent examples include embedded multi-core systems, such as the NXP i.MX6 [34] and the NVIDIA TX1 and TX2 [33], which have an on-chip, general-purpose GPU that can greatly help in addressing the timing challenges of computation intensive tasks by accelerating their execution.

While new technologies in hardware acceleration can reduce CPU computation requirements, using them in a predictable manner poses new challenges. Existing real-time system literature primarily deals with single-core, and to some extent, multi-core CPU systems [28, 40, 29, 6], which often assume complete control over the processor, including preemptive execution of processes with specified priorities. However, operating systems, including real-time operating systems [35, 27], typically treat hardware accelerators as I/O resources which introduces issues such as resource contention from multiple processes, inflexible execution control (such as non-preemptive execution and CPU busy-waiting during resource access), data-transfer delays, etc., which must all be considered while modeling such systems theoretically. Further, accelerators such as GPUs are often arbitrated using closed-source vendor-provided drivers (e.g., by NVIDIA), making it extremely difficult to support predictable behaviour.

Taking into consideration the aforementioned challenges, in this work, we explore and elaborate the advantages and disadvantages of some of the existing techniques to support hardware accelerators such as GPUs. To overcome the identified shortcomings, we then describe new techniques to make GPU-based computing systems more predictable, yet efficient. Our techniques provide a provable theoretical model to ensure predictability (under certain assumptions) and we provide a proof-of-concept implementation backed with numerous experiments to demonstrate improvements in efficiency over previous approaches. While we focus on a GPU throughout this work, similar methods can also be applied to other accelerators such as DSPs and FPGAs.

1.1 Thesis Statement

Modern cyber-physical systems that use hardware accelerator resources such as GPUs can be managed in an analytically predictable, and demonstrably efficient manner, by real-time scheduling *using a server-based approach, which arbitrates resource requests by relinquishing the CPU during resource access.*

1.2 Contributions and Organization

In the following, we briefly describe our contributions and the organization of the subsequent chapters of this document.

- In Chapter 2, we provide a brief overview of the existing work concerning real-time systems, multi-processor scheduling, GPU mechanics, and the challenges faced in using hardware accelerators such as GPUs in a time-predictable manner. We also review existing methodologies to achieve predictable GPU access, and briefly describe their scope and limitations.
- In Chapter 3, we begin by describing the system model on which our work is applicable, and illustrate (with examples) the limitations of the existing state-of-the-art synchronization-based scheme for GPU access. To overcome these limitations, we then propose our server-based approach and provide a theoretical analysis to guarantee its predictability.
- In Chapter 4, we present an evaluation comparing the synchronization-based and the server-based GPU arbitration mechanisms described in Chapter 3. In particular, we present a practical implementation of both approaches and use it to evaluate a vision-based embedded platform. We also measure practical overheads under both approaches. Finally, we use parameters from our measurements to perform extensive schedulability experiments to demonstrate the effectiveness of our server-based approach.
- In Chapter 5, we summarize our results and present directions for future work in this domain.

Chapter 2

Background and Prior Work

In this chapter, we present the essential background to real-time scheduling that we build upon in our work. Further, we provide an overview of the GPU execution model, programming interfaces, and the challenges faced in using GPUs predictably on real-time systems. Finally, we describe some existing approaches to make GPU access predictable.

2.1 Real-Time Scheduling

We begin by providing details as to how real-time systems can be modeled theoretically, and define the metric to verify their temporal correctness, i.e. schedulability. Additionally, we describe existing and relevant procedures to formally verify the schedulability of specific task models.

2.1.1 Task Model

A real-time system reacts to repetitive internal or external events, whose occurrence is serviced by a corresponding task denoted by τ_i. A collection of tasks executing on a system are grouped into a task set: $\Gamma = \{\tau_1, \tau_2, \cdots \tau_n\}$. Each of these tasks release a job on the processor at run time to handle each occurrence of their corresponding events. The characteristics of tasks are captured theoretically using task models. While many task models have been studied in real-time literature, we specifically consider the sporadic task model [30] throughout this work as it is well-studied and provides as reasonably accurate view of practical tasks. This section describes the essentials of the standard sporadic task model which we later extend to incorporate GPU-related task execution.

Under the sporadic task model, a task τ_i is characterized by the following parameters:

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

- C_i: the worst-case CPU execution time (WCET)[1] of each job 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.

Further, tasks are also characterized by a derived parameter, utilization, given by $U_i = C_i/T_i$, which indicates the extent to which a task utilizes the computing resources (we assume the maximum possible utilization for a single computing unit is 1). The time constraint for each task is specified by the deadline, D_i. In order to satisfy its temporal requirements, a job of a real-time task τ_i, released at time t, must complete before its absolute deadline, given by $t + D_i$.

Next, we describe the schemes using which tasks are scheduled on computing resources in order to meet deadline requirements.

2.1.2 Priority Assignment for Scheduling

In order to execute a task set on computing resources (such as uni-core or multi-core CPUs), we assume that its component tasks are arbitrated using a scheduler. Similar to general-purpose operating systems such as Linux, real-time operating systems also associate a notion of priority to tasks to determine the order in which they should be scheduled. We denote the priority of task τ_i by π_i.

The nature of priority assignment has led to the creation of two broad categories [28] of real-time scheduling, *(i)* Fixed-priority and *(ii)* Dynamic-priority scheduling. As the names suggest, task and job priorities do not change under fixed-priority scheduling (except when locks are involved

[1]A challenging component of real-time modeling and analysis is the estimation of the worst-case execution times of tasks, which is not trivial because tasks could potentially incur delays during their execution due to various sources (such as hardware interrupts). One popular method used to estimate WCET is to repeatedly execute the task on an otherwise idle platform and to then measure their response time for each iteration to obtain the *observed worst-case execution time.* This is then pessimistically multiplied by an *inflation factor* to account for unobserved delays. For most cases in practice, such a method provides sufficient WCET bounds, and we use the same method in order to estimate WCET of tasks in this work.

as discussed in Section 3.2.1), whereas dynamic-priority scheduling allows task and job priorities to change based on the policy used. We limit ourselves to fixed-priority scheduling throughout this document due to its ease of implementation and application in practical systems.

One method of priority assignment under fixed-priority scheduling is the Rate Monotonic (RM) scheme [28], where task priorities are assigned inverse to their periods. That is, tasks with shorter periods have a higher priority. This scheme is intuitive in the sense that a task with a shorter period would be released more frequently, and hence expect to be processed sooner (i.e., with a higher priority) than a task with a longer period. While other priority assignment schemes are also possible under fixed-priority scheduling, it has been proved that RM is an optimal prioritization scheme [28], making it lucrative for use in real-time systems.

Real-time scheduling was originally developed with uni-processors in mind [42, 16], and further considerations had to be taken into account for a multi-processor setting [40, 1, 17]. As our focus is on working with modern cyber-physical systems, which typically involve multi-core CPUs and hardware accelerators, in the next section, we provide background for multi-processor scheduling on real-time systems, and subsequently define schedulability, the metric which characterizes temporal constraints in real-time systems.

2.1.3 Preemptive Multi-Processor Fixed-Priority Task Scheduling

On a multi-processor[2], which we deal with throughout this work, fixed-priority scheduling can be further classified into two categories[3]: *(i)* Global and *(ii)* Partitioned. Under global scheduling, tasks may migrate across processors if they have a higher priority by preempting already executing lower-priority tasks or if there is an idle processor available. Under partitioned scheduling, tasks are pinned to specific processor cores and may not migrate even if other cores are idle.

While it may seem intuitive to always prefer global scheduling due to higher utilization benefits, previous work has shown that partitioned scheduling is much easier to analyze and can be used for most practical cases instead [9] because it effectively boils down to the uni-processor case for each core, allowing a large body of existing analyses to be applicable. On the contrary,

[2]Most modern processors and operating systems provide preemption support on CPUs, that is, higher-priority tasks can preempt lower-priority tasks during their execution. Throughout this work, we assume a preemptive execution model for CPUs.

[3]A third, hybrid category, known as *clustered scheduling* is also possible, which partitions processors into smaller clusters of processors that internally perform global scheduling.

while global scheduling offers fast average-case response times, it is relatively harder to analyze, has high overheads in practice [8, 3], and the worst-case performance remains comparable to partitioned scheduling, which makes it less lucrative for real-time systems. As a result, we primarily focus on partitioned fixed-priority task scheduling in our work.

An important consideration under partitioned scheduling is the manner in which tasks are partitioned across multiple computing resources. As this is proved to be a NP-hard problem, existing real-time literature suggests the use of heuristics such as first-fit decreasing, best-fit decreasing and worst-fit decreasing [1, 17] to allocate tasks on processor cores. Details of the effectiveness of various partitioning schemes with hardware accelerators is beyond the scope of this work, and we assume that the partitioning decisions are already made by the system designer, and we evaluate the resulting configuration for resource-aware scheduling, which is our primary focus.

2.1.4 Schedulability

A task set is considered schedulable under a scheduling scheme if all its component tasks meet their timing constraints using that scheme. This is typically determined by performing a schedulability test. Various methods such as utilization-based tests, integer linear programs (ILPs), etc., may be used to determine schedulability, but in this work we focus on schedulability tests based on response time analysis. Under response time analysis, the worst-case response time W_i, which is the amount of time that any job of a task takes to complete among all possible task activations (within the given task set), is computed analytically for each task and compared with the corresponding task deadline. If the worst-case response time of any task in the task set is greater than its deadline, then the task set is said to be unschedulable. If the worst-case response times of all tasks are lower than their corresponding deadlines, then the task set is considered schedulable.

For fixed-priority scheduling on uni-processors, Liu and Layland [28] proposed the first version of the response time analysis. We build upon this in our work to further incorporate GPU related execution. Based on the analysis proposed in [28], the worst-case response time of a task τ_i is given by the following fixed-point iteration[4].

[4]Note that the computing system is assumed to solely execute the tasks in the task set and the analysis does not explicitly take into account any additional interference sources such as interrupts, OS processing, etc. In our discussion throughout this work, we assume that such interference sources are either not accounted for, or included in the estimated WCET of each task.

$$W_i^{n+1} = C_i + \sum_{\pi_h > \pi_i} \left\lceil \frac{W_i^n}{T_h} \right\rceil C_h \tag{2.1}$$

where $W_i^0 = C_i$. The fixed-point iteration terminates when $W_i^{n+1} = W_i^n$, or when the deadline is missed with $W_i^n > D_i$. The first term in the equation represents the time it takes for the task to execute on its own, and the second term captures the worst-case interference possible from all higher-priority tasks preempting the execution of the currently analyzed task. The task is said to meet its temporal requirements if $W_i \leq D_i$. If this condition holds for all tasks in the task set (i.e., for $i \in [1, n]$), then the task set is considered schedulable.

As multi-processor scheduling analysis is reduced to uni-processor scheduling (of each core) when dealing with partitioned scheduling (assuming no inter-task dependencies such as locks), the same analysis as Eq.((2.1)) can be applied, and the worst-case response time of a task τ_i under partitioned multi-processor fixed-priority scheduling is given by

$$W_i^{n+1} = C_i + \sum_{\tau_h \in \mathbb{P}(\tau_i) \wedge \pi_h > \pi_i} \left\lceil \frac{W_i^n}{T_h} \right\rceil C_h \tag{2.2}$$

where $\mathbb{P}(\tau_i)$ indicates the processor on which τ_i is scheduled. The same termination and schedulability conditions as the uni-processor case apply here as well.

2.2 GPU Fundamentals

GPUs are specialized, programmable processors that are typically used for graphics processing, and more recently, have been used for general-purpose computing[5]. The key advantage of GPUs as opposed to CPUs is their ability to efficiently perform parallel operations on multiple sets of data, which is highly relevant for image processing, neural networks and several other domains. However, for execution, GPUs are dependent on CPUs in the sense that they must be activated by the CPUs in order to perform useful operations. In this section, we cover the basics about the functioning of GPUs in order to analytically model them for our work.

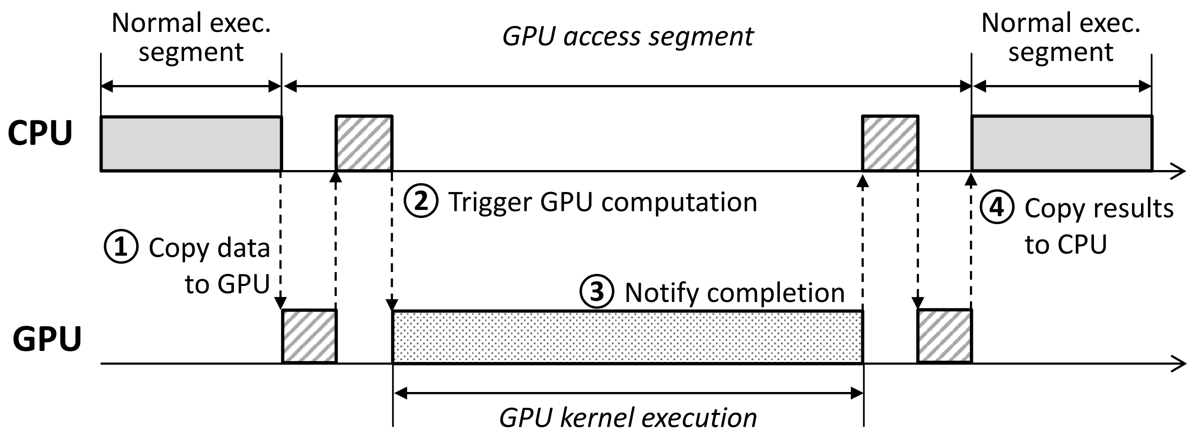


FIGURE 2.1: Execution pattern of a task accessing a GPU

2.2.1 GPU Execution Model

Tasks using the GPU are required to copy the input data and code to the GPU to initiate GPU execution, and are later expected to copy output data after GPU completion. 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 2.1 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 2.1). This is typically done using Direct Memory Access (DMA), which requires no (or minimal) CPU intervention. Then, the task 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 on the CPU, depending on the implementation of the GPU device driver and the configuration used.

2.2.2 Synchronous and Asynchronous GPU Access.

The example in Figure 2.1 uses *synchronous mode* for GPU access, where each GPU command, such as memory copy and GPU kernel execution, can be issued only after the prior GPU command has finished. However, many GPU programming interfaces, such as CUDA [31] and

[5]GPUs being used for general-purpose computing are specifically called GP-GPUs, but we refer to them as GPUs as well.

OpenCL [36], 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 during the execution of 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 access segment. Our work is applicable to both, synchronous and asynchronous modes of GPU execution.

2.3 Challenges in Real-Time GPU Computing

As described in the previous section, GPUs can be extremely useful to perform specific types of operations that require a high degree of parallelism. However, using them in predictable manner brings up some key challenges, which we describe below.

- **Lack of Preemption Support:** Many of the commercial-off-the-shelf (COTS) GPUs do not support a preemption mechanism, which is usually (although not always) assumed in real-time literature [28]. This means GPU access requests from application tasks can only be handled in a sequential, non-preemptive manner. Although recent research [43, 10] has explored the possibility of preemption on GPUs, these methods typically offer significantly higher preemption latency compared to CPUs, making it impractical for use in practice. New architectures such as NVIDIA Pascal [32] also claim to offer GPU preemption, but we have found no documentation available regarding its explicit behaviour[6].
- **Closed-Source GPU Drivers:** COTS GPU device drivers do not respect task priorities and the scheduling policies used in the system. Hence, in the worst case, the GPU access request of the highest-priority task may be delayed by requests of all lower-priority tasks in the system, which could be detrimental to real-time performance and cause the problem of unbounded priority inversion [39, 38]. While using open-source device drivers may be a solution, these drivers are inferior compared to their proprietary/closed-source counterparts, making them less lucrative to be used in actual systems.
- **Resource Contention During Co-Scheduling:** GPUs are usually shared among multiple tasks, all of which may access the GPU arbitrarily. When a task issues and executes a

[6]At the time of writing this document.

GPU request while another task is already using the GPU, these GPU executions are called co-scheduled. Co-scheduling of GPU requests may be problematic again as the manner in which tasks are executed by the device driver is unknown due to it being closed source. Often, the GPU hardware also has its own policy by which it arbitrates GPU requests and this is not disclosed by the hardware manufacturers.

2.4 Prior Work on Predictable GPU Access

The aforementioned issues have motivated the development of predictable GPU management techniques to ensure task timing constraints are met, while achieving a reasonable performance improvement [13, 15, 12, 20, 21, 23, 46]. We briefly describe the methodologies employed by prior work and their scope in this section.

2.4.1 Schedulability-Unaware Approaches

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. TimeGraph also provides a resource reservation mechanism that accounts for and enforces the GPU usage of each task, with posterior and apriori budget enforcement techniques. RGEM [20] allows splitting a long data-copy operation (to or from the GPU) into smaller chunks, reducing blocking time on data-copy operations. Gdev [19] provides common APIs to both, user-level tasks and the OS kernel, to use a GPU as a standard computing resource similar to a CPU. GPES [46] 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 some limitations of todays GPU hardware and device drivers, they do not consider the analytical schedulability of tasks using the GPU. In other words, they handle GPU requests from tasks in a priority-aware and predictable manner, but do not formally analyze the worst-case timing behavior of tasks on the CPU, which is addressed in our work.

2.4.2 Schedulability-Aware Approaches

Elliott et al. [13, 15, 12] modeled GPUs as mutually-exclusive resources and proposed the use of real-time synchronization protocols such as FMLP [5] and R2DGLP [44] for accessing GPUs in an isolated manner, while still using efficient, closed-source device drivers. Isolated execution can mitigate the aforementioned issues with regard to unpredictable behaviour during co-scheduling. Based on these ideas, Elliott et al. developed GPUSync [15], a software framework for multi-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 (similar to TimeGraph), 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. An important distinction is that GPUSync is a highly complicated software solution (about 20,000 lines of code), as it modifies the operating system to include various synchronization protocols. We instead develop a simpler mechanism that can be effectively used even in resource-constrained embedded cyber-physical systems. Further advantages and disadvantages of the synchronization-based approach in general will be thoroughly discussed in Section 3.2.

2.5 Prior Work on Self-Suspension Analysis

A task performing GPU computation need not necessarily continue to execute on the CPU (i.e., it can self-suspend during GPU execution), and as a result, we also explore existing literature to incorporate self-suspensions [2, 4, 11, 23]. Kim et al. [23] 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 a given assignment is NP-hard in the strong sense. Chen et al. [11] reported errors in existing self-suspension analyses and presented corrections for these errors. In our work, build upon results from [4, 11] to incorporate the effect of self-suspension during GPU execution in task schedulability. These analytical approaches are integrated into a server-based resource arbitration mechanism, similar to the alternate synchronization implementation proposed in [39], which we describe in detail in Section 3.3.

Chapter 3

A Server-Based Approach for Predictable GPU Arbitration

In this chapter, we review the existing synchronization-based approach [13, 15, 12] and characterize the limitations of using a real-time synchronization protocol for tasks accessing a GPU on a multi-core platform. Next, to overcome these problems, we describe a new, server-based approach that permits efficient and predictable GPU access.

3.1 System Model and Scheduling Assumptions

We first describe the hardware configuration and task model based on which our solution is built, and followed by the assumptions that we make, which are applicable throughout this work.

3.1.1 Hardware Model

The hardware platform we consider is a multi-core system equipped with a single general-purpose GPU device.[1] This GPU is shared among multiple tasks, and GPU requests from tasks are

Note: This work is largely based on our conference paper which is under peer review at the 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2017.

[1]This assumption reflects today's GPU-enabled embedded processors, e.g., NXP i.MX6 [34] and NVIDIA TX1/TX2 [33]. 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.

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 [37] 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".

3.1.2 Task Model

We consider sporadic tasks with constrained deadlines ($D_i \leq T_i$), and extend the sporadic task model described in Section 2.1.1 to incorporate GPU execution. Under our GPU-aware sporadic model, a task τ_i is characterized by

- 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 length of the GPU access segments of τ_i.
- η_i: the number of GPU access segments in each job of τ_i.

The total utilization of a task τ_i is defined as $U_i = (C_i + G_i)/T_i$. The CPU and GPU utilizations of are given by $U_i^{cpu} = C_i/T_i$ and $U_i^{gpu} = G_i/T_i$, respectively.

As the task model indicates, a task using a GPU can have one or more GPU access segments. We use $G_{i,j}$ to denote the maximum length 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:

$$G_{i,j} := (G_{i,j}^e, G_{i,j}^m)$$

- $G_{i,j}^e$: the WCET of pure GPU operations that do not require CPU intervention in the j-th GPU access segment of τ_i.
- $G_{i,j}^m$: 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, issuing the kernel execution, notifying the completion of GPU commands, and executing other CPU operations. The cost of self-suspension during CPU-inactive time in a GPU segment is assumed to be taken into account by $G^m_{i,j}$. If data is copied to or from the GPU by 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 can be modeled as part of $G^e_{i,j}$ as it does not require CPU intervention.

3.1.3 Scheduling Scheme

We focus on partitioned, fixed-priority, preemptive task scheduling. Thus, each task is statically assigned to a single CPU core. Any fixed-priority assignment, such as Rate-Monotonic [28], can be used for tasks, but we assume that all tasks priorities are unique (ties are broken arbitrarily).

3.2 Limitations of Synchronization-Based GPU Arbitration

In this section, we provide details regarding the synchronization-based GPU arbitration mechanism mentioned in section 2.4.2, and explain two important shortcomings of using this approach, namely, busy-waiting and long priority-inversions.

3.2.1 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 mutex is used for protecting such GPU critical sections. Hence, under the synchronization-based approach, a task holds the mutex to enter its GPU access segment and it releases the mutex when it leaves the GPU access segment. A task can only enter its GPU access segment when the mutex is not held by any other task. If it is indeed already held by another task, the task is inserted into the waiting list of tasks associated with that mutex till it can be granted access to the resource. Some implementations like GPUSync [15] use separate locks for internal resources of the GPU, e.g., copy and execution engines, to achieve parallelism in GPU access, but we make use of a single lock for the entire GPU for simplicity.

Locks under the real-time domain must be governed by real-time locking protocols to prevent the problem of unbounded priority inversions [39], because conventional operating system locks may cause higher-priority tasks to wait on the completion of lock-holding lower-priority tasks for an unbounded amount of time [38], which is highly undesirable for scheduling. Among a variety of real-time synchronization protocols, we consider the Multiprocessor Priority Ceiling Protocol (MPCP) [38, 39] as a representative[2], because it can be implemented with relative ease for partitioned fixed-priority schedulers and has been widely understood in the literature. We shall briefly review the definition of MPCP below. More details on MPCP can be found in [25, 38, 39].

1. When a task τ_i requests 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, the task τ_i is inserted to the waiting list of the mutex for R_k.
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.

3.2.2 Schedulability Analysis

In this subsection, we review the existing task schedulability analysis under the synchronization-based approach with MPCP tailored to the GPU scenario. The analysis described here was originally developed by Lakshmanan et al. [24], with some corrections by Chen et al. [11].

The worst-case response time of a task τ_i under the synchronization-based approach with MPCP is given by the following recurrence equation:

$$\begin{aligned} W_i^{n+1} =& C_i + G_i + B_i^r + \sum_{\tau_h \in \mathbb{P}(\tau_i) \wedge \pi_h > \pi_i} \left\lceil \frac{W_i^n + \{W_h - (C_h + G_h)\}}{T_h} \right\rceil (C_h + G_h) \\ &+ (\eta_i + 1) \left(\sum_{\tau_l \in \mathbb{P}(\tau_i) \wedge \pi_l < \pi_i \wedge \eta_l > 0} \max_{1 \le u \le \eta_l} G_{l,u} \right) \end{aligned} \tag{3.1}$$

[2]GPUSync uses a combination of the R2DGLP and FMLP protocols, as mentioned in Section 2.4.2, but these use Integer Linear Programs (ILPs) for analysis which make them complicated for discussion.

where B_i^r is the remote blocking time for τ_i, $\mathbb{P}(\tau_i)$ is the CPU core where τ_i is allocated and π_i is the priority of τ_i. It terminates when $W_i^{n+1} = W_i^n$, and the task τ_i is schedulable if its response time does not exceed its deadline, i.e., $W_i^n \leq D_i$. Since the task τ_i has to busy-wait during its GPU access, the entire GPU access segment, G_i, is captured as the CPU usage of τ_i, along with its WCET C_i.

In the above equation, the blocking term (B_i^r) indicates the maximum amount of time a request-issuing task has to wait for the GPU to be released by other tasks accessing the GPU. The fourth (ceiling) term captures the execution interference due to higher-priority tasks similar to Eq. (2.2), but this has an additional $(W_h - (C_h + G_h))$ term to take into account the self-suspension behaviour of these tasks [11]. The last term indicates the preemptions caused due to priority boosting of critical sections of lower-priority, GPU-using tasks, on the same core as the analyzed task.

The remote blocking time B_i^r is given by $B_i^r = \sum_{1 \leq j \leq \eta_i} B_{i,j}^r$, where $B_{i,j}^r$ is the remote blocking time for the j-th GPU access segment of τ_i to acquire the GPU. The term $B_{i,j}^r$ is bounded by the following recurrence:

$$B_{i,j}^{r,n+1} = \max_{\pi_l < \pi_i \wedge 1 \leq u \leq \eta_l} W_{l,u}^{gpu} + \sum_{\pi_h > \pi_i \wedge 1 \leq u \leq \eta_h} \left(\left\lceil \frac{B_{i,j}^{r,n}}{T_h} \right\rceil + 1 \right) W_{h,u}^{gpu} \tag{3.2}$$

where $B_{i,j}^{r,0} = \max_{\pi_l < \pi_i \wedge 1 \leq u \leq \eta_l} W_{l,u}^{gpu}$ (the first term of the equation), and $W_{l,u}^{gpu}$ represents the worst-case response time of a GPU access segment $G_{l,u}$. The first term of Eq. (3.2) captures the time for a lower-priority task to finish its GPU access segment. The second term represents the time taken to process the GPU access segments of higher-priority tasks.

The worst-case response time of a GPU access segment $G_{l,u}$, namely $W_{l,u}^{gpu}$, is given by:

$$W_{l,u}^{gpu} = G_{l,u} + \sum_{\tau_x \in \mathbb{P}(\tau_l)} \max_{1 \leq y \leq \eta_x \wedge \pi_x > \pi_l} G_{x,y} \tag{3.3}$$

This equation captures the length of $G_{l,u}$ and the lengths of GPU access segments of higher-priority tasks on the same core. It considers only one GPU access segment from each task, because every GPU access segment is associated with a global priority ceiling and $G_{l,u}$ will never be preempted by normal execution segments.

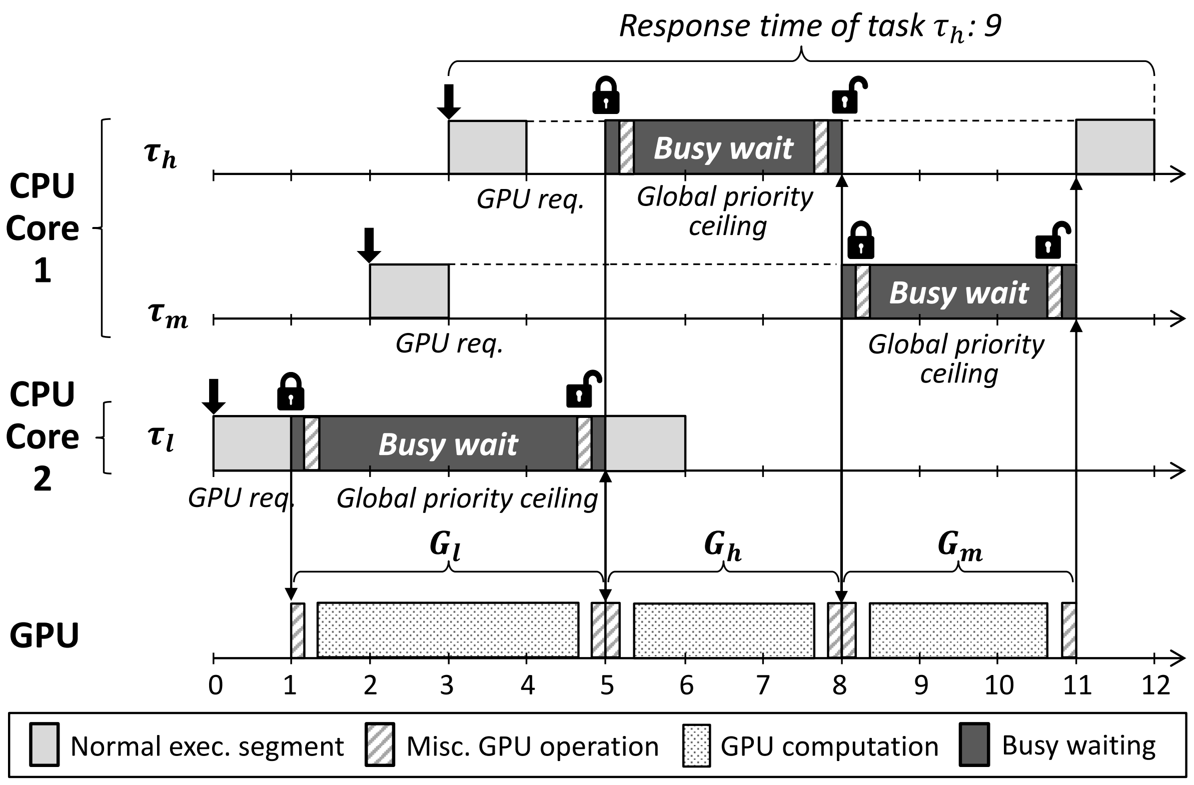


FIGURE 3.1: Example schedule of GPU-using tasks under the synchronization-based approach with MPCP

3.2.3 Limitations

As described in Section 2.2.1, 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., GPU kernel execution, to save CPU cycles. However, under the synchronization-based MPCP approach, any task in its GPU access segment should "busy-wait" for any operation conducted on the GPU in order to ensure timing predictability. This is because real-time synchronization protocols and their analyses, such as MPCP [39], FMLP [5], and OMLP [7], commonly 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.[3] 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 expense of increased pessimism.

[3]The GPUSync implementation [15] can be configured to suspend instead of busy-wait during the GPU kernel execution, but it uses suspension oblivious analysis, and this does not take into account the effect of self-suspension within critical sections.

Some synchronization protocols, such as MPCP [39] and FMLP [5], use priority boosting as a progress mechanism to prevent unbounded priority inversion. However, the use of priority boosting could cause another problem we call "long priority inversion". We describe this problem with the example illustrated in Figure 3.1. 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 executed between two normal execution segments. Tasks 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. 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. At time 5, τ_l releases the GPU mutex and τ_h acquires the GPU mutex next because it has higher priority than τ_m. At time 8, τ_h finishes its GPU access segment and releases the mutex. Next, the task τ_m acquires the GPU mutex and enters its GPU access segment with its global priority ceiling. This makes τ_m preempt the normal execution segment of τ_h. Hence, although the majority of τ_m's GPU access 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.

3.3 Server-based GPU Arbitration

In order to overcome the aforementioned limitations of the synchronization-based approach, we present our server-based mechanism for predictable GPU access control in this section.

3.3.1 GPU Server Execution Model

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 3.2 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

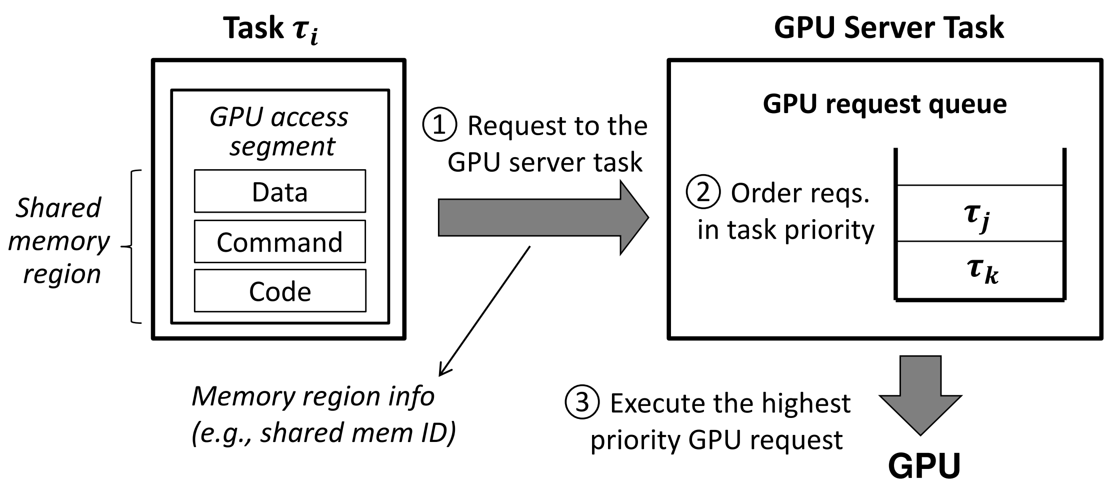

Task τ_i
GPU access segment
Data
Command
Code
Shared memory region
① Request to the GPU server task
Memory region info (e.g., shared mem ID)
GPU Server Task
GPU request queue
② Order reqs. in task priority
τ_j
τ_k
③ Execute the highest priority GPU request
GPU

FIGURE 3.2: GPU access procedure under our server-based approach

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, the task τ_i suspends, allowing other tasks to execute. Second, 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. Third, once the GPU becomes free, the server dequeues a request from the head of the queue and executes the corresponding GPU access segment. During CPU-inactive time intervals, e.g., data copy with DMA and GPU kernel execution, the server also suspends to save CPU cycles. Finally, 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 3.3 shows an example of task scheduling under our server-based approach. This example has the same configuration as the one in Figure 3.1. The GPU server, which the server-based approach creates, is allocated to Core 1. The GPU server overhead ϵ is assumed to be 1/6 time

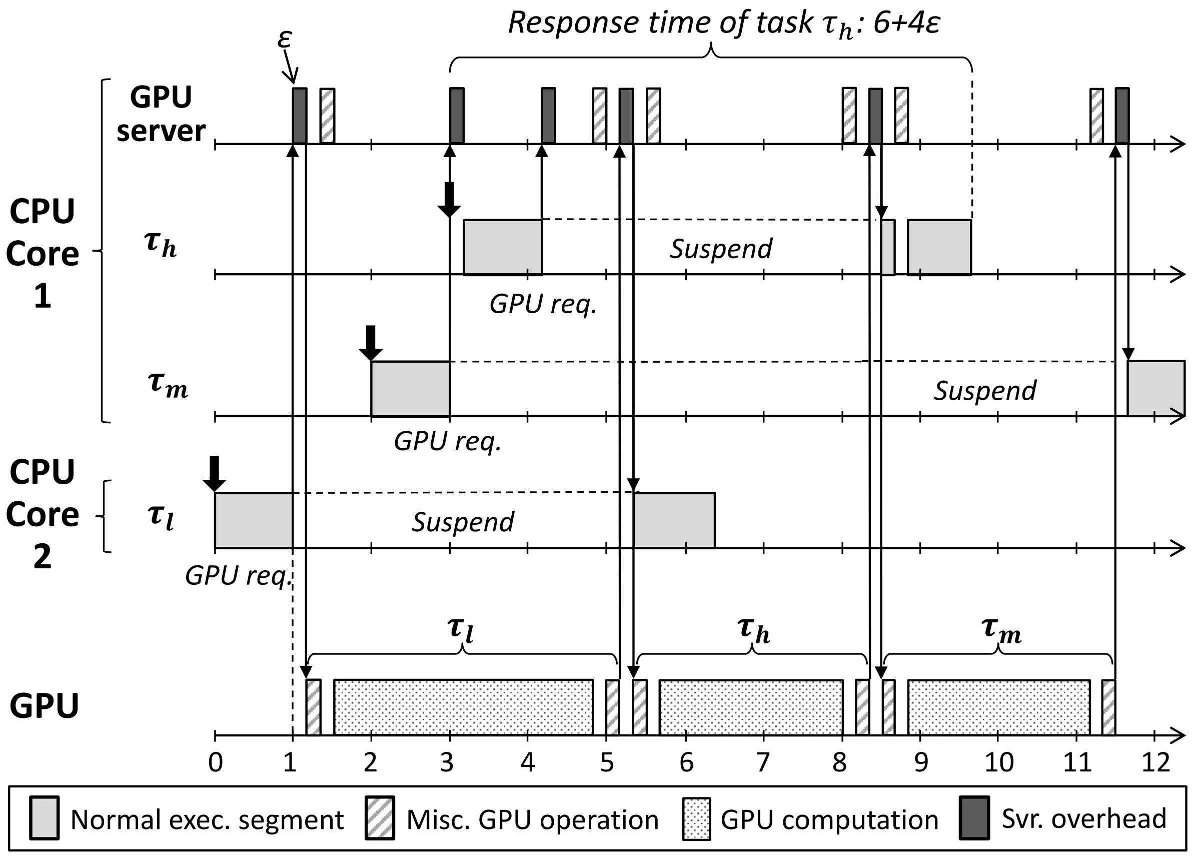


FIGURE 3.3: Example schedule under our server-based approach

units. 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 $1+\epsilon$. Since the server-based approach does not require tasks to busy-wait, 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 task set under the synchronization-based approach, as shown in Figure 3.1. Therefore, we can conclude that the server-based approach provides a shorter response time than the synchronization-based approach for this example task set, if the value of ϵ is under 3/4 time units, which, as measured in Section 4.1, is a very pessimistic value for ϵ.

3.3.2 Schedulability Analysis

We formally 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 3.1. *The GPU server task imposes up to 2ϵ of extra CPU time on each GPU request.*

Proof. As the GPU server task intervenes before and after the execution of each GPU access segment, each GPU request is expected to cause at most 2ϵ of overhead in the worst case. Note that the cost of issuing the GPU request as well as self-suspension is already taken into account by $G^m_{i,j}$, as described in Section 3.1. ■

Lemma 3.2. *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 relation:*

$$B^{w,n+1}_{i,j} = \max_{\pi_l<\pi_i \wedge 1\le u\le \eta_l} (G_{l,u} + \epsilon) + \sum_{\pi_h>\pi_i \wedge 1\le u\le \eta_h} \left(\left\lceil \frac{B^{w,n}_{i,j}}{T_h} \right\rceil + 1 \right) (G_{h,u} + \epsilon) \tag{3.4}$$

where $B^{w,0}_{i,j} = \max_{\pi_l<\pi_i \wedge 1\le u\le \eta_l}(G_{l,u} + \epsilon)$ (the first term of the equation).

Proof. When τ_i, the task under analysis, makes a GPU request, the GPU may already be handling a request from a lower-priority task. As the GPU executes in a non-preemptive manner, τ_i must wait for the completion of the lower-priority GPU request, and as a result, 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 3.3. This is captured by the first term of the equation.

During the waiting time of $B^w_{i,j}$, 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 $B^w_{i,j}$, the maximum number of GPU requests made by τ_h is bounded by $\sum_{u=1}^{\eta_h}(\lceil B^w_{i,j}/T_h\rceil + 1)$. Multiplying each element of this summation by $G_{h,u} + \epsilon$ 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 3.3. *The maximum handling time of all GPU requests of a task τ_i by the GPU server is given by:*

$$B_i^{gpu} = \begin{cases} B_i^w + G_i + 2\eta_i\epsilon & : \eta_i > 0 \\ 0 & : \eta_i = 0 \end{cases} \tag{3.5}$$

where B_i^w is the sum of the maximum waiting time for each GPU access segment of τ_i, i.e., $B_i^w = \sum_{1\le j\le \eta_i} B^w_{i,j}$.

Proof. If $\eta_i > 0$, the j-th GPU request of τ_i is handled after $B^w_{i,j}$ of waiting time, and takes $G_{i,j} + 2\epsilon$ to complete (by Lemma 3.1). Hence, the maximum handling time of all GPU requests of τ_i is $\sum_{j=1}^{\eta_i}(B^w_{i,j} + G_{i,j} + 2\epsilon) = B_i^w + G_i + 2\eta_i\epsilon$. If $\eta_i = 0$, B_i^{gpu} is obviously zero. ■

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:

$$W_i^{n+1} = C_i + B_i^{gpu} + \sum_{\tau_h\in\mathbb{P}(\tau_i)\wedge\pi_h>\pi_i} \left\lceil \frac{W_i^n + (W_h - C_h)}{T_h} \right\rceil C_h \tag{3.6}$$

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 $W_i^n \le D_i$. It is worth noting that, as captured in the third term, the GPU access segments of higher-priority tasks do not cause interference to 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:

$$\begin{aligned} W_i^{n+1} = & C_i + B_i^{gpu} + \sum_{\tau_h\in\mathbb{P}(\tau_i)\wedge\pi_h>\pi_i} \left\lceil \frac{W_i^n + (W_h - C_h)}{T_h} \right\rceil C_h \\ & + \sum_{\tau_j\ne\tau_i\wedge\eta_j>0} \left\lceil \frac{W_i^n + \{D_j - (G_j^m + 2\eta_j\epsilon)\}}{T_j} \right\rceil (G_j^m + 2\eta_j\epsilon) \end{aligned} \tag{3.7}$$

where G_j^m is the sum of the WCETs of miscellaneous operations in τ_i'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 to prove Eqs. (3.6) and (3.7):

Lemma 3.4 (from [4]). *The worst-case response time of a self-suspending task τ_i is upper-bounded by:*

$$W_i^{n+1} = C_i + \sum_{\tau_h \in \mathbb{P}(\tau_i) \wedge \pi_h > \pi_i} \left\lceil \frac{W_i^n + (W_h - C_h)}{T_h} \right\rceil C_h \tag{3.8}$$

Note that D_h can be used instead of W_h in the summation term of Eq. (3.8) [11].

Theorem 3.5. *The worst-case response time of a task τ_i under the server-based approach is given by Eqs. (3.6) and (3.7).*

Proof. To account for the maximum GPU request handling time of τ_i, B_i^{gpu} is added in both Eqs. (3.6) and (3.7). 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 3.4). With these, Eq.(3.6) 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. (3.7) and Eq. (3.6) 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 3.4). These factors are exactly captured by the last term of Eq. (3.7), and hence, it upper bounds task response time in the presence of the GPU server. ■

Chapter 4

Evaluation

In this section, we provide 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 task sets, by using parameters based on practical overheads measured from our implementation.

4.1 Implementation

We implemented prototypes of the synchronization-based and the server-based approaches on a SABRE Lite board [18]. 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 [35] version 1.6[1], 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 of 1 GHz.

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

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

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.

4.1.1 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.

4.2 Practical Evaluation

4.2.1 Overheads

We measured the practical worst-case overheads for both the approaches in order to perform realistic schedulability analysis. Each worst-case overhead measurement involved examining 100,000 readings of the respective operations measured on the i.MX6 platform. Figure 4.1 shows the mean and 99.9th percentile of the MPCP lock operations and Figure 4.2 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.

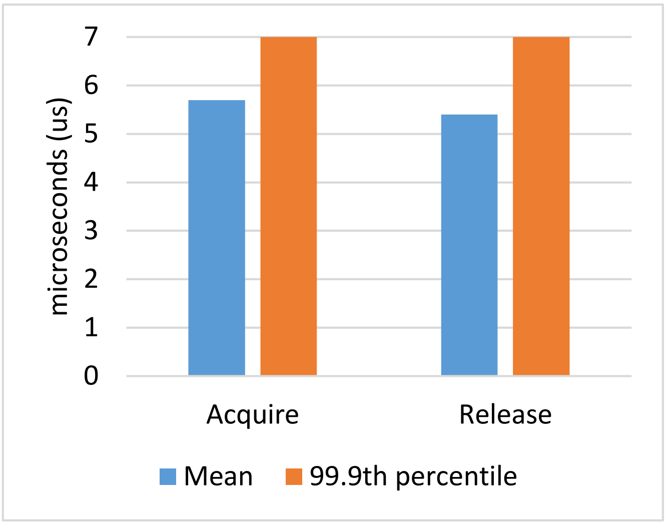


FIGURE 4.1: MPCP lock overhead

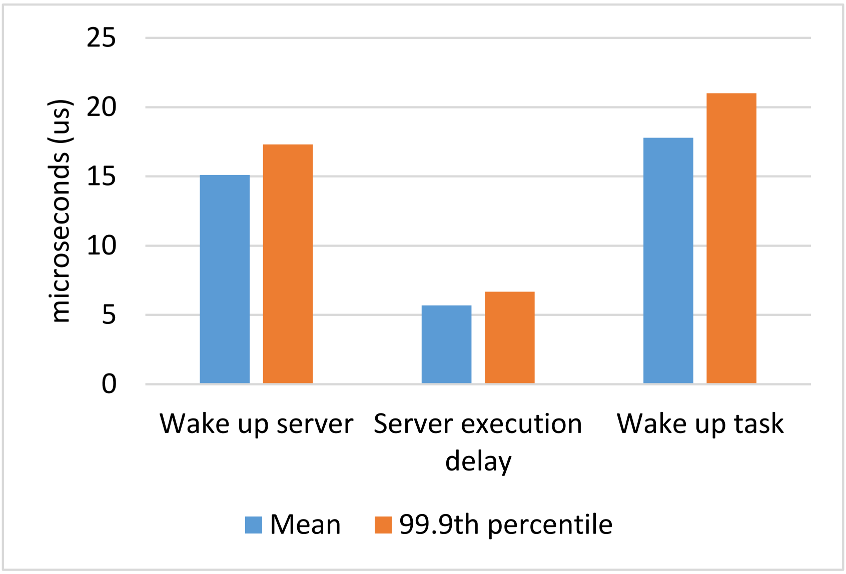
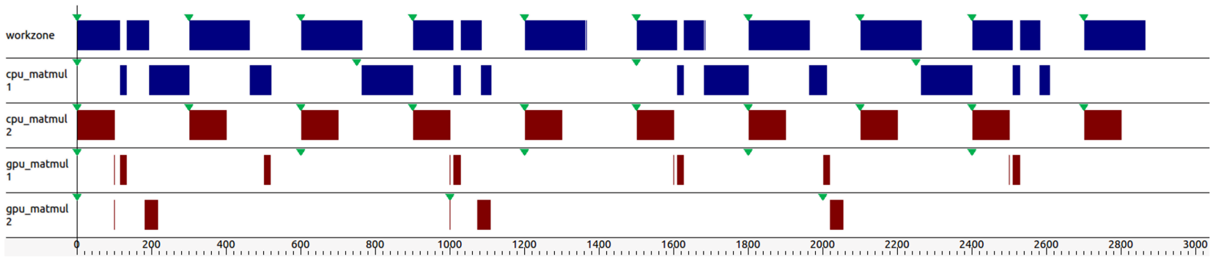


FIGURE 4.2: Server task overheads

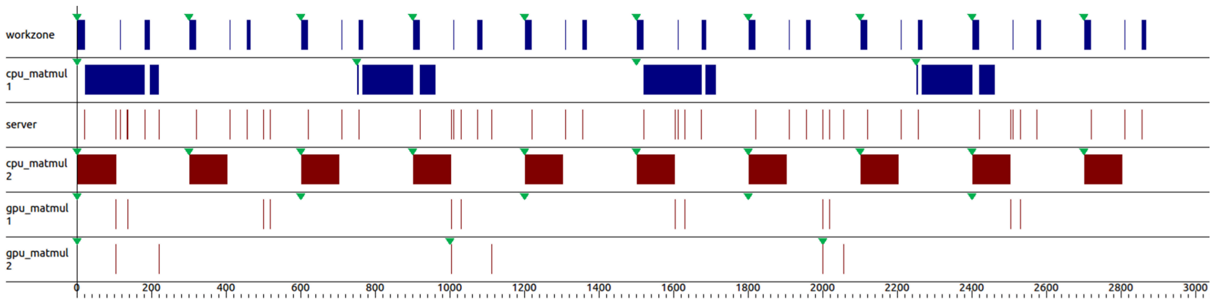
Task τ_i	**Task name**	C_i **(in ms)**	η_i	G_i **(in ms)**	$T_i = D_i$ **(in ms)**	**CPU Core**	**Priority**
τ_1	`workzone`	20	2	$G_{1,1} = 95$, $G_{1,2} = 47$	300	0	70
τ_2	`cpu_matmul1`	215	0	0	750	0	67
τ_3	`cpu_matmul2`	102	0	0	300	1	69
τ_4	`gpu_matmul1`	0.15	1	$G_{4,1} = 19$	600	1	68
τ_5	`gpu_matmul2`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

TABLE 4.1: Tasks used in the case study

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.



(A) Synchronization-based Approach with Busy-Waiting (MPCP)



(B) Server-based Approach

FIGURE 4.3: Task execution timeline during one hyperperiod (3,000 ms)

4.2.2 Case Study: Vision Application

We present a case study motivated by the software system of the self-driving car developed at Carnegie Mellon University [45]. Among various algorithmic tasks of the car, we chose a GPU accelerated version of the workzone recognition algorithm [26] (`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 [28] and each task is pinned to a specific core as described in Table 4.1. 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 4.3.

The CPU-execution traces for the synchronization and server-based approaches are shown in Figure a and Figure 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

Parameters	**Values**
Number of CPU cores (N_P)	4, 8
Number of tasks per core	[3, 5]
Percentage of GPU-using tasks	[10, 30] %
Task period and deadline ($T_i = D_i$)	[100, 500] ms
Task set utilization per core	[30, 50] %
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 $G_{i,j}$ ($G^m_{i,j}$)	[10, 20] %
GPU server overhead (ϵ)	50 μs

Table 4.2: Base parameters for task set generation

allows suspension of tasks while they are waiting for the GPU request to complete. In particular, 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 3.2.3.

4.3 Schedulability Experiments

4.3.1 Task Set Generation

We used 10,000 randomly-generated task sets based on the parameters given in Table 4.2 for each experimental setting. The parameters are inspired from the observations from our case study and the GPU workloads used in prior work [21, 19]. Systems with four and eight CPU cores ($N_P = \{4, 8\}$) are considered. To generate each task set, the number of CPU cores in the system and the number of tasks for each core are first created based on the parameters in Table 4.2. Next, 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 is randomly selected within the defined minimum and maximum task period range. Task deadline D_i is set equal to T_i. On each core, the task set utilization is split into k random-sized pieces, where

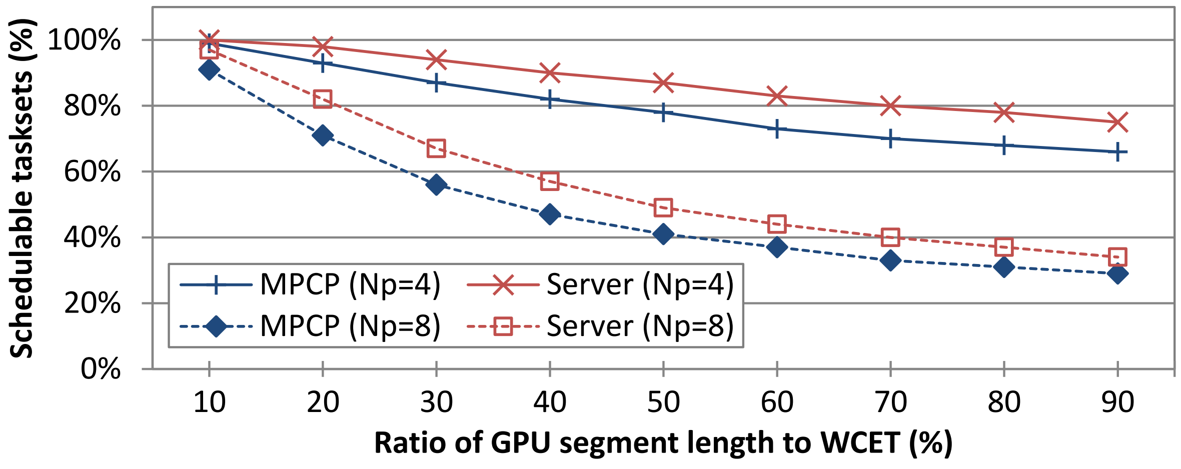


FIGURE 4.4: Schedulability results w.r.t. the accumulated GPU segment length

k is the number of tasks per core. The size of each piece represents the utilization U_i of the corresponding task τ_i, i.e., $U_i = (C_i + G_i)/T_i$. 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 4.2, assuming $G_{i,j} = G^e_{i,j} + G^m_{i,j}$. Finally, task priorities are assigned by the Rate-Monotonic policy [28], with arbitrary tie-breaking.

4.3.2 Schedulability Comparisons

We capture the percentage of schedulable task sets where all tasks meet their deadlines. For the synchronization-based approach, MPCP [39] is used with the task schedulability test developed by Lakshmanan et al. [24] and the correction given by Chen et al. [11]. We considered both zero and non-zero locking overhead under MPCP, but there was no noticeable difference between them. Hence, we only present the results with zero overhead. Under the server-based approach, the GPU server is randomly allocated on one of the cores in the system. Eqs. (3.6) and (3.7) 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 4.1.

Figure 4.4 shows the percentage of schedulable task sets as the ratio of the accumulated GPU segment length (G_i) increases. The solid lines denote the results with $N_P = 4$ and the dotted lines denote results with $N_P = 8$. In general, the percentage of schedulable task sets is higher when $N_P = 4$, compared to when $N_P = 8$. This is because the GPU is contended for by more tasks as

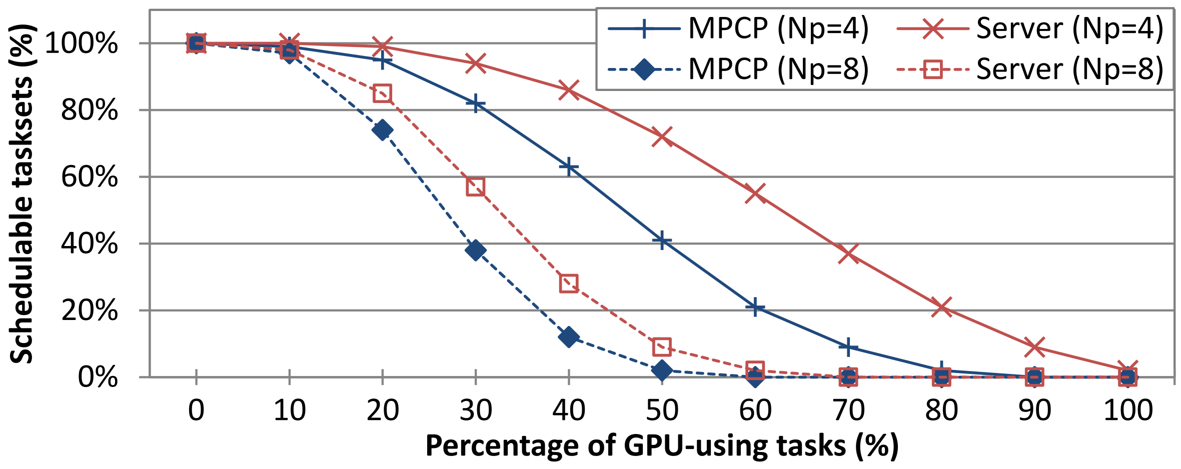


FIGURE 4.5: Schedulability results w.r.t. the percentage of GPU-using tasks

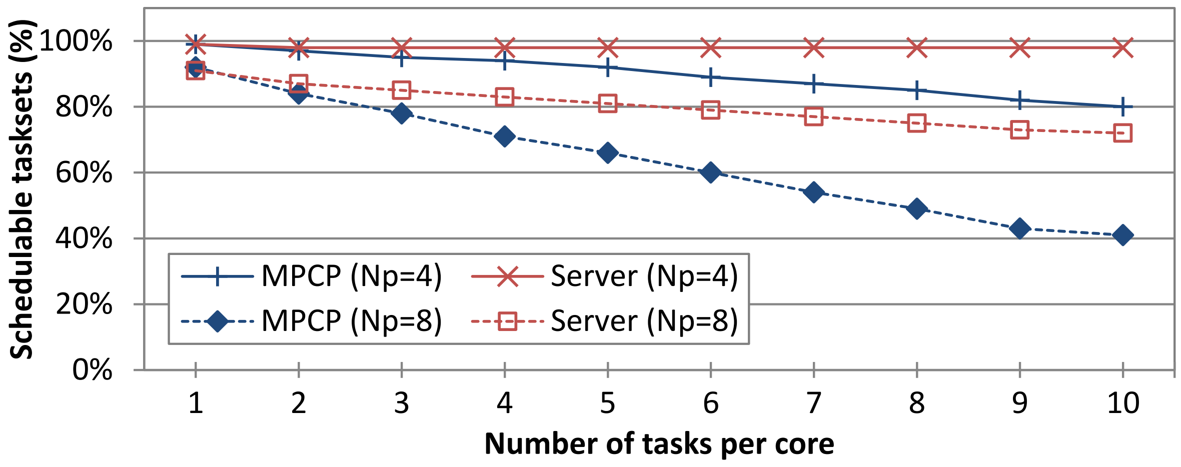


FIGURE 4.6: Schedulability results w.r.t. the number of tasks per core

the number of cores increases. The server-based approach outperforms the synchronization-based approach in all cases of this figure. 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.

Figure 4.5 shows the percentage of schedulable task sets 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 both approaches, the percentage of schedulable task sets reduces as the percentage of GPU-using tasks increases. However, the server-based approach significantly outperforms MPCP, with as much as 34% more task sets being schedulable when the percentage of GPU-using tasks is 60% and $N_P = 4$.

The benefit of the server-based approach is also observed with changes in other task parameters. In Figure 4.6, the percentage of schedulable task sets is illustrated as the number of tasks per core increases. The server-based approach is affected less by the increase in task counts,

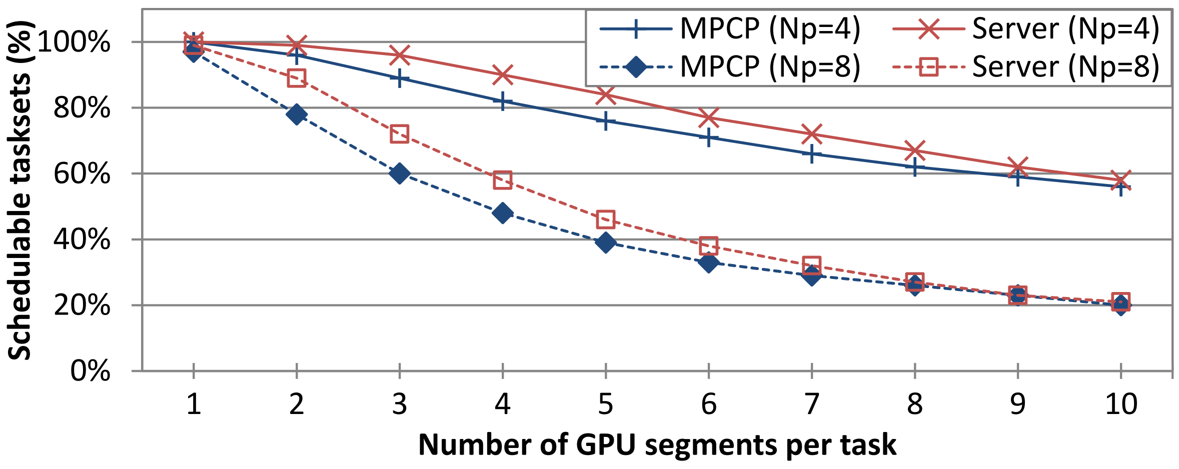


FIGURE 4.7: Schedulability results w.r.t. the number of GPU segments per task

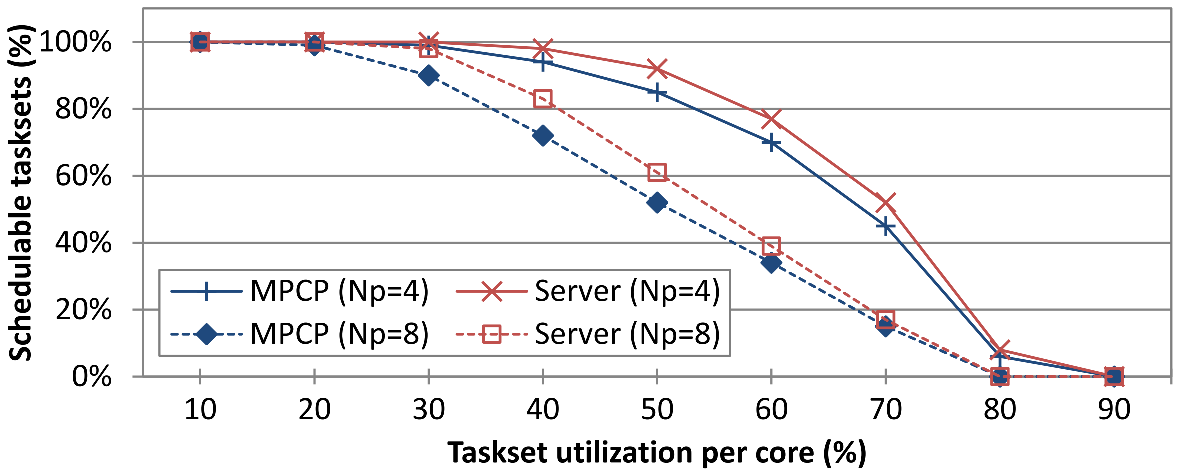


FIGURE 4.8: Schedulability results w.r.t. the task set utilization per core

compared to the synchronization-based approach. The difference in schedulability between the two approaches grows larger as the number of tasks per core increases. This is because as more tasks exist, the total amount of the CPU-inactive time in GPU segments also increases. Figure 4.7 shows the experimental results as the number of GPU segments per task increases. While the server-based approach has higher schedulability for fewer GPU segments per task, it gets closer and then coincides with the synchronization-based approach eventually, because it faces excess server overhead (of 2ϵ per request, by Lemma 3.1) with an increasing number of GPU segments. Figure 4.8 shows the results as task set utilization per core increases. The server-based approach provides higher schedulability than the synchronization-based approach, but as the total task set utilization gets closer to 90%, the gap diminishes and the percentage of schedulable task sets under both approaches goes down to zero.

Next, we investigate the factors that negatively impact the performance of the server-based

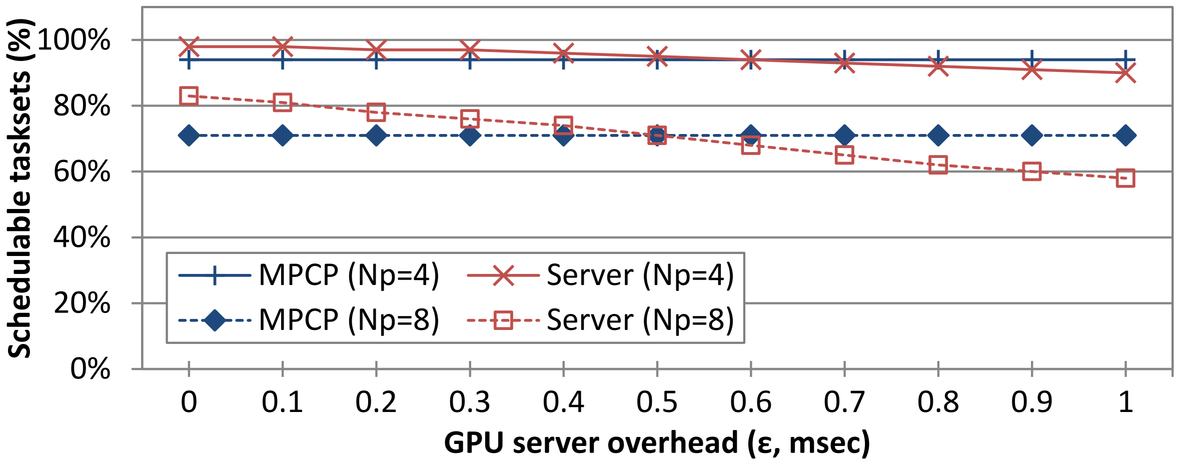


FIGURE 4.9: Schedulability results w.r.t. the overhead of the GPU server (ϵ)

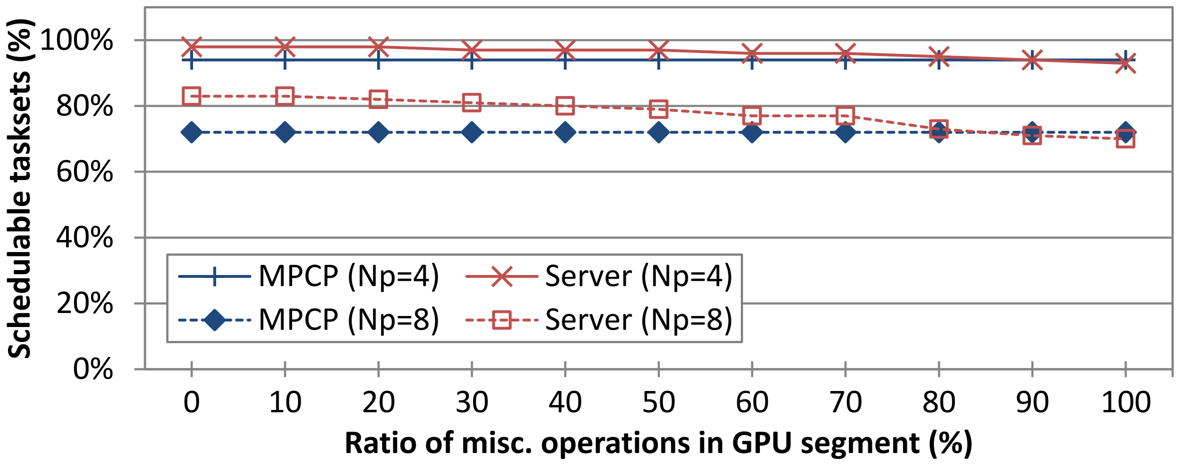


FIGURE 4.10: Schedulability results w.r.t. the ratio of misc. operations ($G^m_{i,j}/G^e_{i,j}$)

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 4.9 shows the percentage of schedulable task sets as the GPU server overhead ϵ increases. Since ϵ exists only under the server-based approach, the performance of MPCP is unaffected by this factor. On the other hand, the performance of the server-based approach deteriorates as the overhead increases.

The amount 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 4.10 shows the percentage of schedulable task sets as the ratio of miscellaneous operations in GPU access segments increases. As MPCP makes tasks busy-wait during their entire GPU access, the performance of the synchronization-based approach remains unaffected. On the other hand, as expected, the performance of the server-based approach degrades as the ratio of miscellaneous operations increases. When the ratio reaches 90%, the server-based approach begins to underperform compared to the synchronization-based approach. However, 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.

In summary, the server-based approach outperforms the synchronization-based approach in most of the 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 or the number of tasks is large. However, we 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 rather high.

Chapter 5

Conclusion

5.1 Summary of Contributions

In this work, we presented a new, server-based approach for predictable CPU access control in real-time embedded and cyber-physical systems. Our server-based approach is motivated by the limitations of the synchronization-based approach, namely busy-waiting and long priority inversion, that we identified in our work. By introducing a dedicated server task for GPU request handling, the server-based approach addresses these limitations, while ensuring predictability 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 also indicate that the server-based approach yields significant improvements in task schedulability compared to the synchronization-based approach in most practical cases.

5.2 Future Work

Our proposed server-based approach offers several interesting directions for future work, and we propose some of them below.

- **Multi-GPU and Dynamic Priorities:** While we focus on a single GPU in this work, we believe that the server-based approach can be extended to multi-GPU systems including dynamic-priority scheduling schemes, similar to GPUSync [14, 15, 13]. One possible way to support multi-GPU platforms would be to create a GPU server for each GPU and allocating a subset of GPU-using tasks to each server.

- **GPU Co-Scheduling:** 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 [32], and the use of the GPU server inherently satisfies this requirement.
- **Exploiting Centrally-Aware Design:** As the GPU server has central knowledge of all GPU-using tasks and their requests, we expect that it can help develop various additional features such as GPU fault tolerance and power management.

Bibliography

[1] Björn Andersson, Sanjoy Baruah, and Jan Jonsson. “Static-priority scheduling on multiprocessors”. In: *IEEE Real-Time Systems Symposium (RTSS)*. 2001.

[2] Neil Audsley and Konstantinos Bletsas. “Realistic analysis of limited parallel software/hardware implementations”. In: *IEEE Real-Time Technology and Applications Symposium (RTAS)*. 2004.

[3] Andrea Bastoni, Björn Brandenburg, and James Anderson. “An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor EDF Schedulers”. In: *IEEE Real-Time Systems Symposium (RTSS)*. 2010.

[4] Konstantinos Bletsas et al. *Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions*. Tech. rep. CISTER-TR-150713. CISTER, 2015.

[5] Aaron Block et al. “A flexible real-time locking protocol for multiprocessors”. In: *IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2007.

[6] Bjorn Brandenburg. “Scheduling and Locking in Multiprocessor Real-time Operating Systems”. PhD thesis. Chapel Hill, NC, USA, 2011. ISBN: 978-1-267-25618-8.

[7] Björn Brandenburg and James Anderson. “The OMLP family of optimal multiprocessor real-time locking protocols”. In: *Design Automation for Embedded Systems* 17.2 (2013), pp. 277–342.

[8] Björn Brandenburg, John Calandrino, and James Anderson. “On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study”. In: *IEEE Real-Time Systems Symposium (RTSS)*. 2008.

[9] Björn Brandenburg and Mahircan Gül. “Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations”. In: *IEEE Real-Time Systems Symposium (RTSS)*. 2016.

[10] Jon Calhoun and Hai Jiang. “Preemption of a CUDA Kernel Function”. In: *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (ACIS)*. 2012.

[11] Jian-Jia Chen et al. *Many Suspensions, Many Problems: A Review of Self-Suspending Tasks in Real-Time Systems*. Tech. rep. 854. Department of Computer Science, TU Dortmund, 2016.

[12] Glenn Elliott and James Anderson. “An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems”. In: *Real-Time Systems* 49.2 (2013), pp. 140–170.

[13] Glenn Elliott and James Anderson. “Globally scheduled real-time multiprocessor systems with GPUs”. In: *Real-Time Systems* 48.1 (2012), pp. 34–74.

[14] Glenn Elliott and James Anderson. “Robust real-time multiprocessor interrupt handling motivated by GPUs”. In: *Euromicro Conference on Real-Time Systems (ECRTS)*. 2012.

[15] Glenn Elliott, Bryan Ward, and James Anderson. “GPUSync: A framework for real-time GPU management”. In: *IEEE Real-Time Systems Symposium (RTSS)*. 2013.

[16] Laurent George, Domaine De Voluceau, and Bp Le Chesnay Cedex. *Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling*. 1996.

[17] Joël Goossens, Sanjoy Baruah, and Shelby Funk. “Real-time scheduling on multiprocessors”. In: *International Conference on Real-Time Systems*. 2002.

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

[19] Shinpei Kato et al. “Gdev: First-Class GPU Resource Management in the Operating System.” In: *USENIX Annual Technical Conference (ATC)*. 2012.

[20] Shinpei Kato et al. “RGEM: A responsive GPGPU execution model for runtime engines”. In: *IEEE Real-Time Systems Symposium (RTSS)*. 2011.

[21] Shinpei Kato et al. “TimeGraph: GPU scheduling for real-time multi-tasking environments”. In: *USENIX Annual Technical Conference (ATC)*. 2011.

[22] Junsung Kim et al. “Parallel Scheduling for Cyber-Physical Systems: Analysis and Case Study on a Self-Driving Car”. In: *International Conference on Cyber-Physical Systems (ICCPS)*. 2013.

[23] Junsung Kim et al. “Segment-fixed priority scheduling for self-suspending real-time tasks”. In: *IEEE Real-Time Systems Symposium (RTSS)*. 2013.

[24] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. "Coordinated Task Scheduling, Allocation and Synchronization on Multiprocessors". In: *IEEE Real-Time Systems Symposium (RTSS)*. 2009.

[25] Karthik Lakshmanan, Ragunathan Rajkumar, and John Lehoczky. "Partitioned fixed-priority preemptive scheduling for multi-core processors". In: *Euromicro Conference on Real-Time Systems (ECRTS)*. 2009.

[26] JongHo Lee et al. "Kernel-based traffic sign tracking to improve highway workzone recognition for reliable autonomous driving". In: *IEEE International Conference on Intelligent Transportation Systems (ITSC)*. 2013.

[27] *Linux PREEMPT_RT v4.9-rt*. URL: `https://wiki.linuxfoundation.org/realtime/start`.

[28] Chung Laung Liu and James Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61.

[29] Jane Liu. *Real-Time Systems*. Prentice Hall, 2000.

[30] Aloysius Mok. "Fundamental design problems of distributed systems for the hard-real-time environment". PhD thesis. 1983.

[31] *NVIDIA CUDA: Compute Unified Device Architecture*. URL: `http://http://www.nvidia.com`.

[32] *NVIDIA GP100 Pascal Whitepaper*. URL: `http://www.nvidia.com`.

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

[34] *NXP i.MX6 Processors*. URL: `http://www.nxp.com`.

[35] Shuichi Oikawa and Ragunathan Rajkumar. "Linux/RK: A Portable Resource Kernel in Linux". In: *IEEE Real-Time Systems Symposium (RTSS) Work-In-Progress*. 1998.

[36] *OpenCL*. URL: `https://www.khronos.org/opencl/`.

[37] Nathan Otterness et al. "An Evaluation of the NVIDIA TX1 for Supporting Real-time Computer-Vision Workloads". In: *IEEE Real-Time Technology and Applications Symposium (RTAS)*. 2017.

[38] Ragunathan Rajkumar. "Real-time synchronization protocols for shared memory multiprocessors". In: *International Conference on Distributed Computing Systems (ICDCS)*. 1990.

[39] Ragunathan Rajkumar, Lui Sha, and John P Lehoczky. “Real-Time Synchronization Protocols for Multiprocessors”. In: *IEEE Real-Time Systems Symposium (RTSS).* 1988.

[40] Krithi Ramamritham, John Stankovic, and Perng-Fei Shiah. “Efficient scheduling algorithms for real-time multiprocessor systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 1.2 (1990), pp. 184–194.

[41] Jack Stankovic. “Misconceptions about real-time computing: a serious problem for next-generation systems”. In: *Computer* 21.10 (1988), pp. 10–19.

[42] John Stankovic et al. “Implications of classical scheduling results for real-time systems”. In: *Computer* 28.6 (1995), pp. 16–25.

[43] Ivan Tanasic et al. “Enabling preemptive multiprogramming on GPUs”. In: *International Symposium on Computer Architecture (ISCA).* 2014.

[44] Bryan Ward, Glenn Elliott, and James Anderson. “Replica-Request Priority Donation: A Real-Time Progress Mechanism for Global Locking Protocols”. In: *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA).* 2012.

[45] Junqing Wei et al. “Towards a viable autonomous driving research platform”. In: *IEEE Intelligent Vehicles Symposium (IV).* 2013.

[46] Husheng Zhou, Guangmo Tong, and Cong Liu. “GPES: a preemptive execution system for GPGPU computing”. In: *IEEE Real-Time Technology and Applications Symposium (RTAS).* 2015.