ENABLING REAL-TIME CERTIFICATION OF AUTONOMOUS DRIVING APPLICATIONS

Tanya Amert

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2021

ABSTRACT

Tanya Amert: Enabling Real-Time Certification of Autonomous Driving Applications
(Under the direction of James H. Anderson)

The push towards fielding advanced driver-assist systems (ADASs) is happening at breakneck speed. Semi-autonomous features are becoming increasingly common, including adaptive cruise control and automatic lane keeping. Today, graphics processing units (GPUs) are seen as a key technology in this push towards greater autonomy. However, realizing full autonomy in mass-production vehicles will necessitate the use of stringent certification processes.

Unfortunately, currently available GPUs tend to be closed-source "black boxes" that have features that are not publicly disclosed; these features must be documented for certification to be tenable. Furthermore, existing real-time task models have not evolved to handle historical-result requirements common in computer-vision (CV) applications, which introduce cycles in processing graphs; existing models must be extended to account for such dependencies. Additionally, due to size, weight, power, and cost constraints, multiple CV applications may need to share a single hardware platform; if the platform contains accelerators such as non-preemptive GPUs, such sharing must be managed in a way that ensures applications are isolated from one another. For ADAS certification to be possible, these challenges must be addressed.

This dissertation addresses each of these three challenges. First, scheduling details of NVIDIA GPU are presented, as derived through extensive micro-benchmarking experiments. These details provide the foundation for identifying and automatically detecting key issues when using NVIDIA GPUs in real-time safety-critical applications. Second, a generalization of a real-time task model is introduced, enabling the computation of response-time bounds for processing graphs that contain cycles. This model exposes a trade-off between the age of historical data, the resulting response-time bounds, and the accuracy of the CV application; this trade-off is explored in detail. Finally, a time-partitioning framework for multicore+accelerator platforms is introduced. When applied alongside existing methods for alleviating spatial interference, this framework can help enable component-wise ADAS certification on multicore+accelerator platforms.

ACKNOWLEDGEMENTS

The completion of this dissertation would not have been possible without the help, guidance, and support of many, many people. First and foremost, I am thankful for my advisor, Jim Anderson, who took me on when I realized my path in graduate school was going in the wrong direction, and has always been there to push me towards becoming a better version of myself. Jim has shown a great balance of severity and kindness, from helping me to be a better writer and researcher to holding my fussy months-old child during a weekly research meeting. I am also grateful for my committee: Marko Bertogna, Sridhar Duggirala, Glenn Elliott, Jan-Michael Frahm, and Don Smith. Their guidance and feedback has been crucial to my success.

I am exceedingly grateful for the efforts of my collaborators: Joshua Bakita, Michael Balszun, Samarjit Chakraborty, Charlotte Dorn, Martin Geier, Manish Goyal, Abel Karimi, Saujas Nandi, Catherine Nemitz, Nathan Otterness, Stephen Tang, Peter Tong, Sergey Voronov, Thanh Vu, Kecheng Yang, and Ming Yang, as well as Ming Lin and Shan Yang. You all taught me so much. I have also enjoyed getting to know other current and former members of the Real-Time Systems group, including Shareef Ahmed, Lee Barnett, Shai Caspin, Micaiah Chisholm, Calvin Deutschbein, Zhishan Guo, Clara Hobbs, Namhoon Kim, Vance Miller, Sims Osborne, Sarah Rust, Abhishek Singh, Brittany Subialdea, Bryan Ward, and Tyler Yandrofski, as well as our newest members, Angelos Angelopoulos and Randal Tuggle. Thank you all for your discussions, and varying combinations of mentorship/menteeship.

My time at UNC would not have been so positive if not for the wonderful staff in the Computer Science Department. Thank you to Jodie Gregoritsch for being such a delight, and helping me when I felt lost. Thank you to Missy Wood for being the best stand-in for a favorite aunt that anyone could hope for so far from home; your fierce passion for UNC CS and its students really made the department feel welcoming. Thank you to the fabulous External Relations staff, including Brandi Day, Melissa Gunnell, Erin Lane, Brett Piper, Gina Rozier, Rosario Vila, and Hope Woodhouse, for their efforts to keep the department engaged in the community, and their endless patience and genuine interests in us as graduate students. I am grateful for so many more UNC CS staff, including Murray Anderegg, Robin Brennan, Bil Hays, Alicia Holtz, Denise Kenney, Beth Mayo, David Musick, and Mike Stone. Without you all, the CS department would not have been such a welcoming place to spend so many years. I would also like to thank current and former UNC CS faculty, especially Sanjoy Baruah, Ming Lin, Dinesh Manocha, Don Porter, and Diane Pozefsky, as well as Gary Bishop and Stephen Pizer, who were pivotal in my teaching of COMP 550 at UNC.

My graduate school experience would not have been complete without the students I had the opportunities to teach at both UNC and Carleton. The terms I spent teaching were the perfect source of rejuvenation I needed, in particular for the final graduate-school push, and for that I am heavily indebted to the conversations and mentorship I received from Amy Csizmar Dalal, David Liben-Nowell, Dave Musicant, and Layla Oesper, as well as the support of the fantastic Paula Stowe and Mike Tie.

Getting through graduate school was made entirely possible by the friendships I formed and experiences those friends helped make happen. This includes students in the GAMMA group (especially Aniket Bera, Andrew Best, Nic Morales, Sahil Narang, Srihari Pratapa, Tanmay Randhavane, Atul Rungta, Auston Sterling, Justin Wilson, and Shan Yang), women from the Graduate Women in Computer Science (especially Catherine Nemitz, Marie Nesfield, and Sarah Rust, who helped keep it going in our early years), students in the board games group (Phil Ammirato and Marcy, Michael Deakin, Peter Lincoln, Nic Morales, True Price and Luci Pounders, Auston Sterling, Sergey Voronov, and Qiuyu Xiao), my wonderful Computer Science Student Associate co-officers (Marc Eder, Alan Kuntz, and Catherine Nemitz), members of our grading reading group (Shai Caspin, Calvin Deutschbein, Janine Hoelscher, Catherine Nemitz, Kaki Ryan, Aaron Smith, and Aaron Willcock), and all of the other wonderful friends I've made along the way, including Gio Acosta, Lisa Bauer, Kat Best, Alyssa Byrnes, Andrew Chi, Jeff Ichnowski and Chrissy Kistler, Bashima Islam, Kat Kirchoff, Kaci Kuntz (and Elias!), Xinran (Conny) Lu, Maja Najaran, and Natalie Stanley.

It is always a treasure when a friendship grows into something more like family, and I am so grateful to have met Catherine Nemitz so early in my time at UNC. We are so similar yet perfectly different, and that has made working on problem sets and paper deadlines, trips around the country and the world, and shared job searches all less lonely and more fun. You push me to be a kinder and stronger person, are the perfect extra aunt to Colin, and I cannot wait to continue to share in our personal and professional joys with each other.

I am endlessly grateful for the support of my family, especially my parents and my sisters, my in-laws, and my grandparents. You all mean so much to me, and have been so patient at the number of fishing trips that have turned into paper-deadline-induced couch-occupying work-cations.

All of the hard times over the past six years could only have been weathered with the neverending support of my wonderful husband, Chris. You have been my rock, and I could not have done it without you. I am so

grateful for the love you have shown me and Colin, and look forward to our next set of adventures together. To my son, Colin, thank you for being such a joy in my life. To our pets, Lulu, Hobbes, Therese, and Sadie, thank you for your attention, comfort, and unconditional love.

The research in this dissertation was supported by NSF grants CPS 1239135, CNS 1409175, CPS 1446631, CNS 1563845, CNS 1717589, CPS 1837337, CPS 2038855, and CPS 2038960; AFOSR grant FA9550-14-1-0161, ARO grants W911NF-14-1-0499, W911NF-17-1-0294, and W911NF-20-1-0237; ONR grant N00014-20-1-2698; and a grant from General Motors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADAS	Advanced Driver-Assist System
AI	Artificial Intelligence
A-MOTA	Average Multiple Object Tracking Accuracy
APF	Artificial Potential Field
API	Application Programming Interface
CCDF	Complementary Cumulative Distribution Function
CDF	Cumulative Distribution Function
CE	Copy Engine
CGLP	Concurrency Group Locking Protocol
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CUPiD^{RT}	CUDA Pitfall Detector for Real-Time Systems
CV	Computer Vision
DAG	Directed Acyclic Graph
DM	Deadline-Monotonic
DNN	Deep Neural Network
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processor
EDF	Earliest-Deadline-First
EE	Execution Engine
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
FPS	Frames per Second
fz-blocking	Forbidden-Zone-Induced Blocking
G-EDF	Global Earliest-Deadline-First
GEL	G-EDF-Like
GPU	Graphics Processing Unit

HOG	Histogram of Oriented Gradients
IoU	Intersection-over-Union
JLFP	Job-Level Fixed Priority
L1	Level 1 (Cache)
L2	Level 2 (Cache)
L3	Level 3 (Cache)
lidar	Light Detection and Ranging
LITMUS^{RT}	Linux Testbed for Multiprocessor Scheduling in Real-Time Systems
mAP	Mean Average Precision
MOT	Multi-Object Tracking
MOTA	Multiple Object Tracking Accuracy
MOTP	Multiple Object Tracking Precision
OMLP	$O(m)$ Locking Protocol
OS	Operating System
PA	Partition Allocator
PCI-e	Peripheral Component Interconnect Express
PCR	Periodic Component Reservation
PDF	Probability Density Function
pi-blocking	Priority-inversion Blocking
PID	Proportional-Integral-Derivative
PMF	Probability Mass Function
RM	Rate-Monotonic
RRT	Rapidly-exploring Random Tree
RTOS	Real-Time Operating System
SM	Streaming Multiprocessor
SoC	System-on-a-Chip
SVM	Support Vector Machine
SWaP-C	Size, Weight, Power, and Cost
TimeWall	Time-Isolated Multicore Execution With Accelerator Locking

VCP	Vertex Coloring Problem
WCET	Worst-Case Execution Time

CHAPTER 1: INTRODUCTION

Semi- and fully autonomous advanced driver-assist systems (ADASs) have become mainstream; systems such as Tesla Autopilot and Cadillac Super Cruise provide features like adaptive cruise control and automatic lane keeping. To produce fully autonomous vehicles at a mass scale, increasingly complex sensing and computer-vision (CV) algorithms will be necessary. However, there is a fundamental disconnect between the design of CV applications and the real-time guarantees necessary to certify their use in such safety-critical applications. CV applications are designed to be throughput-focused, *i.e.*, they seek to ensure *real-fast* execution. In contrast, *real-time* systems require predictable execution subject to timing constraints. If these timing constraints are violated, accidents and even fatalities can result.

Such timing constraints can be checked through extensive real-world testing. For example, Waymo, a leader in autonomous driving, has reported over 20 million miles driven by their autonomous vehicles (Waymo, 2021). However, it is prohibitive to drive million of miles to verify the predictable timing of any new features. Instead, timing guarantees can be provided by *real-time certification*. Such certification entails analyzing the computational workload and the hardware platform to provide bounds on the worst-case completion time, or *response time*, of any invoked task.

Response-time analysis of CV applications is complicated for several reasons; we focus on three specific challenges. First, many CV algorithms require results from prior execution instances, leading to potential delays in cases when those executions have not yet completed and made the results available. These CV algorithms are frequently structured as task graphs in which cycles represent dependencies on historical results. Unfortunately, existing analysis techniques prohibit cyclic dependencies or remove them in a way that results in overly conservative analysis. Second, the CV algorithms used in automotive applications typically make use of specialized hardware accelerators, such as graphics processing units (GPUs), to improve performance through parallelism. However, GPUs are designed for throughput rather than predictability, and therefore can have unexpected behavior (Otterness et al., 2016). Furthermore, autonomous vehicles are subject to size, weight, power, and cost (SWaP-C) constraints, so it is often necessary for multiple applications to execute on the same hardware. Thus, the scheduling policies of such hardware platforms must be understood in order to ensure that timing constraints will be met. Finally, some existing certification specifications, such as ARINC 653 (Prisaznuk, 2008) for avionics platforms, require that temporal isolation be guaranteed when executing multiple applications on a single shared hardware platform. This is made significantly more challenging for CV applications, which may be developed and tested independently, possibly by different vendors, and which may utilize GPUs, which present a challenge for isolation due to non-preemptive GPU execution.

In this dissertation, we seek to address these challenges and bridge the divide between the design of CV algorithms and the requirements of the real-time systems of which they are a part. In the remainder of this chapter, we discuss the requirements of real-time systems. Next, we explore the sources of the disconnects between real-time systems and GPU-equipped CV applications. Then, we present the thesis of this dissertation and provide an overview of the contributions supporting that thesis. Finally, we outline the remaining chapters in this dissertation.

1.1 Real-Time Systems

In real-time systems, the correctness of a system of applications depends not only upon the correctness of the output, but also upon the *timeliness* of the computations. Associated with each recurrent task in a real-time system are an invocation period, a relative deadline, and a worst-case execution time (WCET). To verify the correctness of a real-time task, one thus must prove that the response time of a task, *i.e.*, the time between its invocation and corresponding completion, will be bounded.

If the response time of a task must not exceed its relative deadline, then the deadline is considered to be *hard*; missing a hard deadline can have catastrophic consequences, such as a vehicle accident or occupant injury or fatality. Some deadlines, on the other hand, are classified as *soft*; it is sometimes acceptable for a soft deadline to be missed. Applications with soft deadlines are typically less safety critical. Multiple definitions exist for soft deadlines: for example, the *tardiness* (*i.e.*, the amount by which the deadline is missed) must be bounded, or at most a given number of deadline misses may occur. In this dissertation, we focus on soft deadlines for which the tardiness of each task must be bounded.

A hard-real-time task is *schedulable* if all of its deadlines are guaranteed to be met, whereas the soft-real-time tasks we consider are schedulable if every invocation of each task can be guaranteed to have bounded tardiness. Schedulability analysis is used to determine whether a set of tasks is schedulable, and takes into consideration the available processors on the platform, the parameters of each task, and the scheduling algorithm used to assign tasks to processors. Analysis has been developed for many different schedulers for uniprocessor platforms, and has been the subject of much research for multicore platforms in recent years. However, prior work has failed to adequately address systems that rely on multicore+accelerator platforms, or CV applications that may require cyclic data dependencies. In the next sections, we explore these challenges in more detail.

1.2 Autonomous Driving Applications

The input to an autonomous-driving system is a set of sensors, including cameras, lidar (light detection and ranging), and radar. Autonomous-driving applications use data from these sensors to determine the control values for the vehicle, typically specified as steering (direction), throttle (speed), and brake. There are two distinct approaches to autonomous-vehicle design. In the first approach, a system is composed of a large set of components, each developed to perform a specific task, *e.g.*, (Franke, 2017; Paden et al., 2016). Common components are those for sensor-based perception of the surrounding scene, path planning to navigate through the scene, and local control used to follow the desired path. The alternative approach takes a more end-to-end view: Deep Neural Networks (DNNs) are used to process all sensor input and directly select vehicle control values, *e.g.*, (Codevilla et al., 2018; Mnih et al., 2016). In this dissertation, we focus on the modular ADAS design approach. As many of these components involve scene perception via image-based sensors, much of our work has focused on CV algorithms used in ADAS applications.

ADAS features like adaptive cruise control necessitate the anticipation of dangerous scenarios with enough time for driver or vehicle intervention. Predicting dangerous situations typically entails tracking dynamic objects, such as pedestrians and other vehicles, and using a motion model to extrapolate future positions. Such a tracking application can be expressed as a dataflow graph in which nodes represent computations or other high-level CV primitives and edges represent data dependencies. A graph for a tracking pipeline is depicted in Figure 1.1. In this example, the "Predict track positions" node uses the track produced by the "Update tracks" node during the prior time step to predict the location of an object in the current time step, introducing a *cyclical dependency* on prior results.

Recent work has applied real-time scheduling principles to produce response-time bounds for graphs (Elliott et al., 2015; Yang et al., 2015, 2016, 2018a). Unfortunately, most graph-based response-time analysis

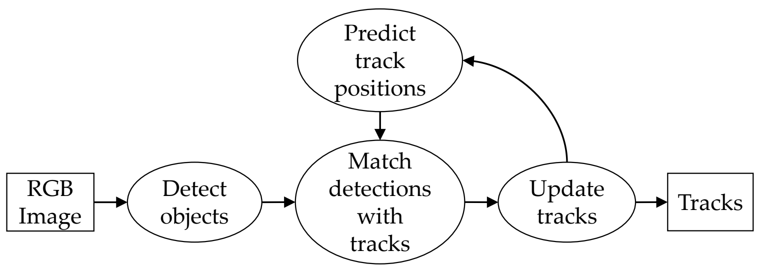


Figure 1.1: The tracking-by-detection pipeline.

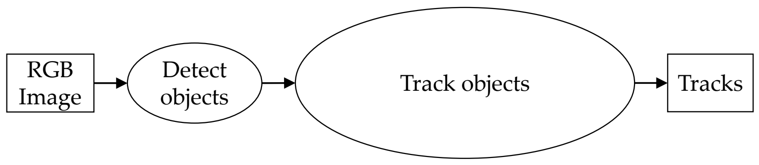


Figure 1.2: The tracking-by-detection pipeline with a supernode in place of the cycle.

assumes that all graphs are directed acyclic graphs (DAGs), that is, that they do not contain any cycles. Thus, for prior analysis to apply, any cycles must be replaced with *supernodes*, as shown in Figure 1.2. To ensure the cyclic dependencies are respected, the supernodes are forced to execute *sequentially*, *i.e.*, a given invocation of a supernode must complete before the next can begin. However, if the utilization of the supernode is greater than 1.0 (*i.e.*, the sum of the WCETs of the nodes comprising the cycle is greater than the invocation period of the graph), then sequential execution may lead to unbounded response times. In order to be able to certify CV applications as used in ADASs, response-time analysis for cyclic graphs is needed. This requires a new task model that ensures data dependencies are respected while enabling the parallelism necessary to guarantee bounded task response times.

1.3 Graphics Processing Units

GPUs were originally designed to work alongside Central Processing Units (CPUs) to accelerate computer-graphics applications. These applications require writing *shaders*, which process triangle-based matrix-multiply operations in parallel to render images. As a result, modern commercial GPUs have hundreds or thousands of parallel hardware cores to perform these computations.

The two primary GPU manufacturers are AMD and NVIDIA. AMD provides an open-source software stack for their GPUs, whereas the NVIDIA ecosystem is proprietary, including the drivers. However, in 2007, NVIDIA positioned itself as a market leader in GPU hardware with the introduction of the CUDA API (a C/C++ extension to enable general-purpose GPU computing). NVIDIA GPUs span the spectrum of space and computing power: the Jetson series of system-on-a-chip (SoC) platforms, with low power and size profiles, are tailored for ADAS applications; the GeForce series of discrete GPUs, on the other hand, are heavier, larger, and require more power, but focus on providing high performance for personal computers.

Unfortunately, with this focus on performance, GPUs have traditionally been designed for throughput rather than predictability. For example, many details related to GPU scheduling policies are omitted from NVIDIA's documentation. This omission prevents users from developing programs that depend on CUDA features or hardware micro-architectures that may change. The cost to real-time system developers is the loss of information necessary for analysis of predictability. Furthermore, as NVIDIA drivers and CUDA source code are closed source, determining such scheduling details is a challenging task.

Additionally, the SWaP-C constraints that arise for ADAS systems necessitate sharing of hardware accelerators, such as GPUs, between multiple system components implementing the many ADAS functions. The question then arises: how can we ensure temporal isolation between multiple GPU-using components? Put another way, how can different components be given the illusion of full ownership of a GPU, such that each component may execute independently of the presence of others in the system? Once again, the closed-source nature of the NVIDIA software stack and a lack of clarity in CUDA documentation prevent simple solutions to these problems.

1.4 Thesis Statement

The disconnect between real-time requirements and CV-application design is a key hurdle that must be overcome to enable the certification of ADASs. In this dissertation, we assume a multicore+accelerator platform, with a focus on NVIDIA GPUs as hardware accelerators. With such a platform in mind, we address the design disconnect from three directions, as summarized in the following thesis statement:

Enabling the real-time certification of autonomous-driving applications relies on bridging the divide between real-time and computer-vision system design. This includes evaluating the trade-offs between history requirements, accuracy, and response-time bounds of computer-vision applications, utilizing knowledge of GPU scheduling policies and best practices for using GPUs in real-time applications, and enabling temporal isolation on multicore+accelerator platforms.

1.5 Contributions

We now briefly overview the contributions that support this thesis.

1.5.1 Generalized Task Model for Applications with Historical Dependencies

In Chapter 3, we introduce the rp-sporadic task model (restricted parallelism), which adds a per-task settable parameter, p, specifying the number of invocations of the same task that may execute concurrently. From a schedulability point of view, such parallelism for graph nodes in a cycle equates to setting of history age requirements for data produced by prior node execution instances. Therefore, instead of insisting on *sequential* cycle execution, which is the root cause of any over-utilization, we instead allow *parallel* execution by permitting the use of slightly older history, *i.e.*, $p > 1$. By allowing $p > 1$, the rp-sporadic task model thus enables response-time bounds to be computed for any graph containing a cycle, even if the cycle utilization exceeds 1.0.

1.5.2 Evaluation of the History-vs.-Response-Time-vs.-Accuracy Trade-Off

Allowing $p > 1$ is clearly not a solution that comes entirely "for free": as p increases, although response-time bounds may decrease, so too may CV accuracy due to using older history. This history-versus-response-time-versus-accuracy trade-off is a key issue in the real-time certification of ADASs, of which both CV researchers and automotive designers should be aware, yet it has never been examined in depth.

In Chapter 3, we provide the first-ever detailed study of this trade-off. For the history-versus-response-time trade-off, we explore the impact on analytical response-time bounds as p increases for a large set of synthetically generated cycle-containing graph-based task systems. To observe the impact of increasing p on the accuracy of a cyclic workload, we consider a CV application in which pedestrians and other vehicles are tracked via images recorded by a camera attached to a moving vehicle. We perform our evaluation using CARLA (Dosovitskiy et al., 2017), an open-source simulator designed for ADAS research. The use of CARLA allows us to generate a broad range of scenarios, to consider each sensing component independently, and to consider and evaluate potential modifications to the vehicle's behavior based on tracking results.

1.5.3 Model of NVIDIA GPU Scheduling

NVIDIA has marketed its Jetson line of embedded CPU+GPU platforms specifically targeting autonomous artificial-intelligence (AI) applications. In Chapter 4, we present a set of experimentally determined rules that define how the NVIDIA Jetson TX2's GPU schedules work submitted to it. These rules take into account the ordering within and among first-in-first-out (FIFO) *streams* of requests for GPU operations, the means by which various internal queues of requests are handled, the ability of the GPU to co-schedule operations so they execute concurrently, the selection mechanism used to determine the order in which requests are handled, and the resource limits that constrain the GPU's ability to handle new requests. Our rules indicate that the TX2's GPU employs a variant of hierarchical FIFO scheduling that is amenable to real-time schedulability analysis.

We then explore multiple optional GPU features that cause certain request streams to be treated specially, further complicating GPU scheduling. These features include the usage of a special stream called the *NULL stream* and the usage of *stream priorities*. The available documentation regarding both of these features lacks several details necessary for predicting specific runtime behavior. Through further experiments, we show how each of these features affects our derived scheduling rules. In addition, we discuss the differences between the rules we discerned for the TX2 and the scheduling behavior of several discrete NVIDIA GPUs.

1.5.4 Guidance for Using NVIDIA GPUs in Real-Time Applications

In Chapter 4, we detail several pitfalls that can arise when using NVIDIA's CUDA API to utilize NVIDIA GPUs in real-time systems, leading to unexpected delays not only on the GPU, but also on the host CPU. Such CPU blocking must factor into response-time analysis if it may occur in safety-critical applications, and thus can lead to system capacity loss.

To address the problem of detecting improper GPU use, we present CUPiD$^{\text{RT}}$ ($\underline{\text{CU}}$DA $\underline{\text{Pi}}$tfall $\underline{\text{D}}$etector for $\underline{\text{R}}$eal-$\underline{\text{T}}$ime Systems), a library we developed to monitor operations submitted to NVIDIA GPUs and report issues based on the most severe pitfalls we identified. CUPiD$^{\text{RT}}$ can be used with any applications that utilize NVIDIA GPUs, and although CUPiD$^{\text{RT}}$ was designed with real-time systems in mind, throughput-oriented applications can also benefit from detection and remediation of these issues.

To evaluate its usefulness, we detail the results produced by using CUPiD$^{\text{RT}}$ to analyze ten GPU-using sample applications from the popular CV library OpenCV (Bradski, 2000). Based on two of the most problematic pitfalls we identified, we configured CUPiD$^{\text{RT}}$ to detect issues related to specific CUDA API commands. Of the ten sample applications we analyzed, we found that each was subject to at least one of the issues we sought to detect. We also performed a case study demonstrating the benefits of resolving the issues detected by CUPiD$^{\text{RT}}$. To do so, we modified the source code of one of the ten applications, resolving all detected issues, and showed that such changes reduced execution times and improved predictability of computation-submission times.

1.5.5 Enabling Isolation on Multicore+Accelerator Platforms

In Chapter 5, we address the problem of ensuring temporal isolation between different components on a multicore+accelerator platform. We specifically focus on accelerators that require non-preemptive execution, and assume that each component in the system has been assigned to a given set of hardware resources that the component "owns" for given slices of time.

To ensure such isolation, we designed a hierarchical scheduler, which consists of three main parts: a table-driven scheduler to allocate time slices and hardware resources to components, per-component schedulers that allocate processors to tasks, and a specialized locking protocol to orchestrate accelerator accesses while respecting time-slice boundaries. In addition, we provide blocking analysis for our locking protocol.

To evaluate our solution, we implemented the specialized locking protocol within the hierarchical scheduler as a scheduler plugin using LITMUS$^{\text{RT}}$ on a multicore+GPU platform. Enforcing temporal isolation between GPU accesses from different components revealed some unexpected edge cases, which we discuss in detail, as well as our resulting solution for budgeting GPU accesses.

1.5.6 Additional Evaluations

In Chapter 6, we consider a set of automotive applications, including the tracking application considered in Chapter 3, all running as real-time tasks within the LITMUS$^{\text{RT}}$ developmental real-time operating system (Brandenburg, 2011; Calandrino et al., 2006). We use this task set as a basis of further evaluation. First, we consider the full history-versus-response-time-versus-accuracy trade-off. Our experiments show that relaxing the history requirements (*i.e.*, allowing $p > 1$) can enable lower observed response times with only a minimal impact on application accuracy. Then, we provide a detailed evaluation of our time-partitioning solution from Chapter 5. We explore the appropriate choice of time-slice lengths and validate the isolation afforded by our approach.

1.6 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we provide an overview of related background on real-time scheduling, CV applications used in ADASs, and the software and hardware used to enable such applications. We then dive into the trade-offs between historical data dependencies, algorithm accuracy, and application response times in Chapter 3. In Chapter 4, we detail the scheduling policies of NVIDIA GPUs, and address the pitfalls that can arise in their use in real-time applications and how to detect such issues. In Chapter 5, we introduce our framework to ensure temporal isolation on multicore+accelerator platforms between unrelated system components. Then, in Chapter 6, we discuss the results of our extended evaluations using a set of ADAS-related applications. Finally, we conclude in Chapter 7.

CHAPTER 2: BACKGROUND

In this chapter, we provide an overview of background related to real-time autonomous-driving applications. We first discuss various applications necessary for ADASs (Section 2.1). Then, we list several software libraries and GPUs that can be used to realize such applications (Sections 2.2 and 2.3). Finally, we formally review existing real-time task models, including models for graph-based CV applications (Section 2.4). We conclude by discussing our contributions in the context of prior work (Section 2.5).

2.1 Autonomous-Driving Applications

A common approach to autonomous-vehicle design is to separately develop several software components that facilitate path planning, perception, and control for a vehicle, *e.g.* (Franke, 2017; Paden et al., 2016). We now briefly survey different applications used to implement these components. We will see that a feature of these algorithms is that they are often graph based, *e.g.*, organized as a dataflow graph of processing steps.

2.1.1 Path Planning

Path planning can be done at two granularity levels: a global planner determines the high-level path from the start position to the goal position, and a local planner determines more fine-grained motion between waypoints on that path. For example, a global planner may determine which intersections and lanes a vehicle should traverse, whereas the local planner chooses the steering angles necessary to perform lane changes.

Global planning. Global planning algorithms typically act on graph-based discretizations of mapping data. As an example, consider the map depicted in Figure 2.1. This map has eight intersections, labeled A-H, which correspond to nodes in the map's graph-based representation. Roads connecting intersections correspond to edges in the graph. Global planning can thus be viewed as a shortest-path problem in a graph. For example, breadth-first search, depth-first search, or Dijkstra's algorithm may be used. A breadth-first path from intersection C to intersection H (assuming ties are broken alphabetically) is shown in Figure 2.2. The corresponding search graph is shown in Figure 2.3.

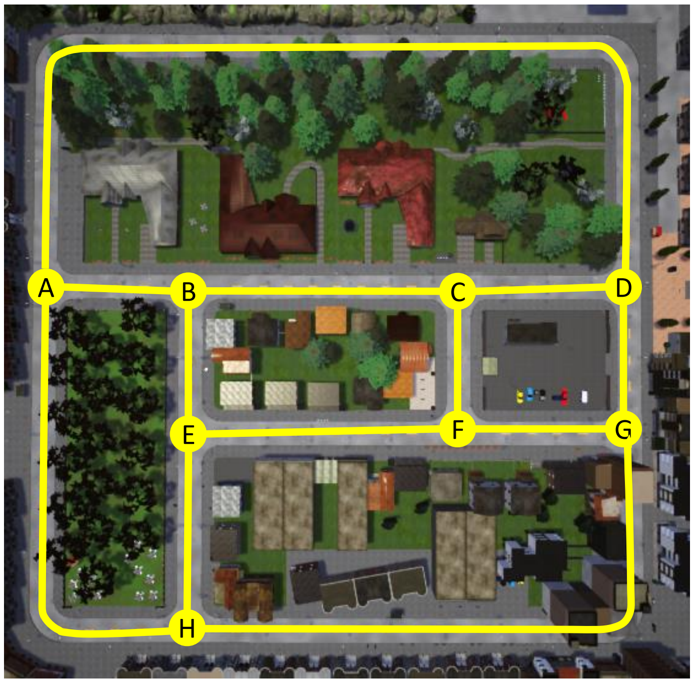


Figure 2.1: A town from the autonomous-driving simulator CARLA (Dosovitskiy et al., 2017), overlaid with the graph corresponding to intersections and their connecting roads.

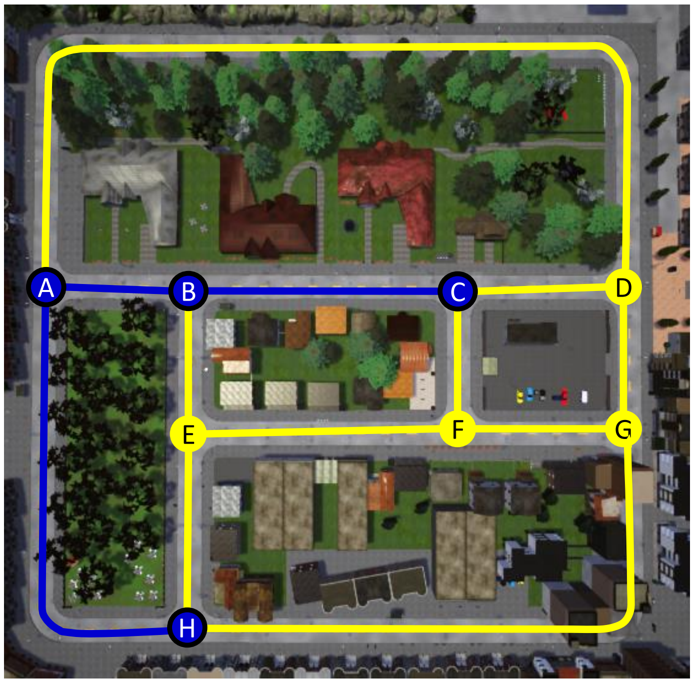


Figure 2.2: The path from node C to node H resulting from breadth-first search.

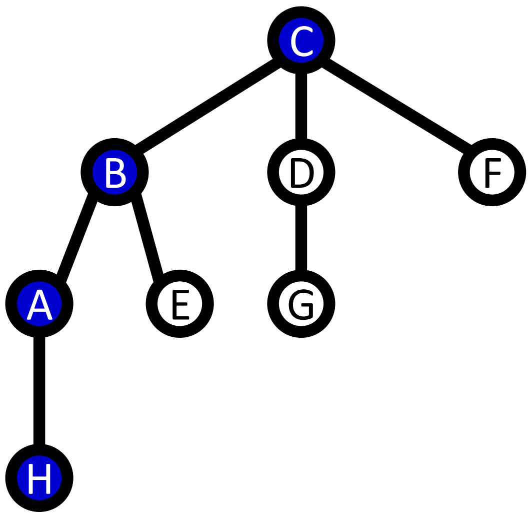


Figure 2.3: The path from node C to node H resulting from breadth-first search.

Other shortest-path algorithms, such as A^*, can make use of heuristics to improve the shortest path found in global planning. Another approach, called Rapidly-exploring Random Trees (RRT) (LaValle and Kuffner Jr, 2001), randomly chooses a node in the graph to extend at each iteration.

Local planning. Local planners act on segments of the path produced by a global planner to determine more fine-grained actions. For example, given the path segment between intersections A and H in Figure 2.2, a local planner will navigate between waypoints along that segment, and utilize perception algorithms to ensure that the vehicle stays in the appropriate lane. Additionally, a global planner may not consider obstacles in the scene, but the local planner must ensure an obstacle-free path if possible.

One approach to local planning is the Pure Pursuit path-tracking algorithm (Coulter, 1992). This algorithm computes the curved trajectory necessary for a vehicle to travel from its current position to a goal point (*e.g.*, the next waypoint along the path computed by the global planner). If perception algorithms detect obstacles along the computed arc, an alternate steering direction can be chosen to avoid those obstacles.

Local planning can also be implemented using the Artificial Potential Field (APF) method (Khatib, 1986), which associates with each point in space (*e.g.* positions on the 2D road map) a potential value corresponding to whether the vehicle should occupy that point. Obstacles (or points not on the road) are assigned a high repulsive potential, and the center of the appropriate lane is given a high attractive potential. Additionally, vehicle-dynamics constraints can be included when determining potential values to rule out regions of space that the vehicle cannot reach, *e.g.*, based on its current orientation and turning radius (Song et al., 2020). The local planner then chooses a path that tends towards the attractive-potential points on the way to the goal.

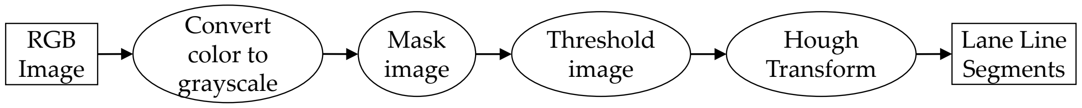


Figure 2.4: Lane-detection pipeline using the Hough Transform.

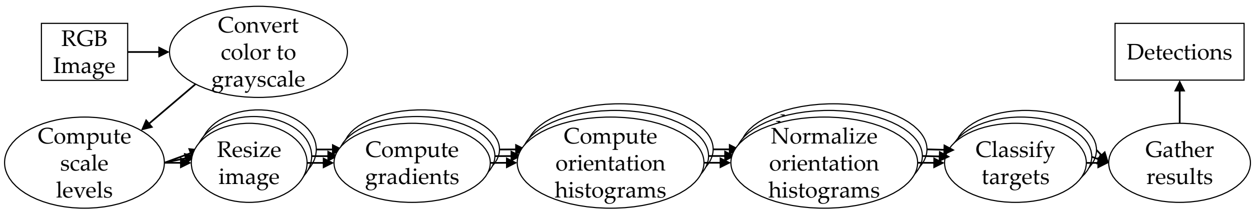


Figure 2.5: Pedestrian-detection pipeline using HOG features.

2.1.2 Perception

The path-planning algorithms discussed in Section 2.1.1 rely on perception algorithms, *e.g.*, to determine lane boundaries and detect and track potential obstacles. We now discuss a few of these algorithms for CV applications that process streams of images from cameras.

Lane detection via the Hough Transform. Lane detection can be done using the Hough Transform, which detects features such as lines in an image. As shown in Figure 2.4, the image is first converted to grayscale and unnecessary regions (*e.g.*, the top half, which may correspond to the sky and is unlikely to contain lane lines) are masked out. Then, a threshold is applied, given the assumption that lane lines are bright and the road is dark. Finally, the Hough Transform can be used to detect lane lines in the thresholded image.

Pedestrian detection via Histogram of Oriented Gradients (HOG). The HOG-based pedestrian-detection algorithm (Dalal and Triggs, 2005; Prisacariu and Reid, 2009) computes gradients (the HOG features) within an image at a range of different scales, and classifies potential detections at each scale. The computational cost increases with the number of image scales, but each scale enables detection of a pedestrian at a different distance from the camera. The HOG features are provided to a classifier such as a Support Vector Machine (SVM), which determines whether a potential detection is a pedestrian. The output is a series of rectangular regions of the image ("bounding boxes") of varying sizes and positions, along with probability values that reflect the likelihood of a correct detection. This process is depicted in Figure 2.5 for three image scales; in the experiments in Chapters 4–6, we used 13 image scales.

Multi-Object Tracking (MOT). MOT tracks an unknown number of objects, or *targets*, through a scene. A *track* is a sequence of estimated positions and sizes (as bounding boxes) of a target over time. A track is a model of a target's *trajectory*, *i.e.*, the sequence of its actual real-world positions through time. Time is measured by camera frames.

Tracking-by-detection is a common approach to MOT. This pipeline is illustrated in Figure 1.1. The output from frame t is the set of tracks after frame t. The input to frame t is an RGB image and the set of tracks from frame $t-1$. A key challenge in tracking-by-detection is to match detected bounding boxes with predictions of new track positions. To do this, the percentage overlap is compared for all detection-prediction rectangle pairs. The Hungarian method (also known as Munkres' algorithm) can be used to quickly match detections to predictions (Milan et al., 2016; Stiefelhagen et al., 2006). The overlap of two rectangles is computed using the *Intersection-over-Union* measure (IoU) (Rezatofighi et al., 2019), also known as the Jaccard index. The IoU (a scalar) is the ratio of the size of the intersection to the size of the union of two rectangles within an image. The Hungarian algorithm chooses an assignment of detections to predictions that maximizes the IoU of the selected pairs.

2.1.3 Control

Using the results of perception algorithms, the local planner must output control values. For a vehicle, these are typically in the form of steering, throttle, and brake. However, the desired values may change greatly from one time step to the next. To generate a stable and smooth path, a controller, *e.g.*, a proportional-integral-derivative (PID) controller (Åström and Murray, 2021), uses a feedback loop to determine the control output for the current time step. For a PID controller, this output depends on the error (computed as the difference between the desired value and the observed value), as well as the integral and derivative of the error. The resulting control output can then be translated to motor revolution speed, wheel angles, *etc.*

2.2 Software for Real-Time Autonomous-Driving Applications

In this section, we provide a brief overview of several software libraries that enable real-time graph-based applications to be developed, autonomous vehicles to be simulated, and CV and other AI applications to be designed and built upon.

2.2.1 Real-Time Software

A real-time operating system (RTOS) provides support for predictable job execution based on real-time scheduling algorithms. One such RTOS is LITMUS^{RT} (Brandenburg, 2011; Calandrino et al., 2006), which was designed for real-time researchers to prototype different schedulers and synchronization protocols. LITMUS^{RT} is a modified Linux kernel that introduces a real-time scheduling class for which scheduling plugins can be written. LITMUS^{RT} provides several existing plugins, and in Chapter 5 we discuss a new plugin that enables time partitioning between different system applications. As part of this plugin, we implemented a time-partitioning-aware locking protocol to arbitrate access to hardware accelerators.

To enable real-time scheduling of graph-based applications, Elliott *et al.* (2014) developed PGM^{RT}. This library enables inter-process communication by operating-system threads representing different graph nodes; each invocation of a node in the graph must wait until all of its predecessors (corresponding to incoming graph edges) have produced data. We utilize PGM^{RT} in our evaluation discussed in Chapter 6.

2.2.2 CARLA: an Autonomous-Driving Simulator

CARLA (Dosovitskiy et al., 2017) is an open-source urban-driving simulator. CARLA uses Unreal Engine 4 (Epic Games, 2020) to produce photo-realistic scenes, such as the one depicted in Figure 2.1, combined with accurate physical models of automobile dynamics. CARLA is a client-server system. The urban environment and the interactions of all vehicles and pedestrians with it are simulated on the server.

The client sends choices of steering, acceleration, and braking to the server. These values must be determined by manually operating the vehicle (*e.g.*, via the keyboard) or by a software *agent* that implements the perception, planning, and control components for driving. Sensor data for these components are provided by the server. Such data can include physically based graphics renderings of camera frames and the pose (location and orientation) of the camera and each vehicle and pedestrian in the scene.

2.2.3 CV and Other AI Frameworks

Many popular frameworks provide various AI functionality. We briefly discuss three of them here.

TensorFlow. TensorFlow (Abadi et al., 2016) was developed by Google to serve as an open-source framework for machine learning. TensorFlow is commonly used with Python to train and re-train deep-learning models; models trained using TensorFlow can then be packaged and used on other machines. In the work presented in

Chapter 3, we used TensorFlow to train a model (which had been pre-trained on a different dataset) to detect vehicles and pedestrians in CARLA.

PyTorch. Facebook developed the machine-learning framework PyTorch (Paszke et al., 2019). PyTorch also features a Python API for ease of use. Both TensorFlow and PyTorch provide native support for NVIDIA GPUs, which can enable substantial speed improvements for training and inference using neural networks.

OpenCV. Focusing specifically on computer-vision algorithms, OpenCV (Bradski, 2000) is a software library that provides C++ and Python APIs for a broad range of CV algorithms. In this dissertation, we leveraged many algorithms available as part of OpenCV 3.4.

OpenCV includes several sample applications to demonstrate different algorithmic functionality. Here, we list several of those that use GPUs as hardware accelerators:

- *Background/Foreground Segmentation*: A video sequence is processed to classify pixels in each video frame as being in the background or foreground of the image.
- *Optical Flow*: Using two images, the movement of each pixel from one image to the other is determined. Optical-flow algorithms include the Brox *et al*. algorithm (Brox et al., 2004), the Farnebäck algorithm (Farnebäck, 2003), the iterative Lucas-Kanade method with image pyramids (Lucas and Kanade, 1981), and the TV-L1 method (Pérez et al., 2013; Zach et al., 2007); these algorithms are demonstrated in three different OpenCV sample applications.
- *Feature Extraction*: Features of certain types (*e.g.*, lines or circles) are located in an individual image. The Generalized Hough Transform extends the original Hough transform to arbitrary shapes; these algorithms are demonstrated in two sample applications.
- *Object Detection*: Although similar to feature extraction, object detection entails detecting high-level objects, such as pedestrians. In the `hog` sample application, detections are performed for each frame of a video.
- *Morphology*: Morphological operations modify binary images, *e.g.*, by “eroding” the foreground (shrinking any white regions) or by “dilating” the foreground (expanding any white regions). More complex operations, such as “opening” and “closing,” can be performed by combining the basic erosion and dilation operations.

- *Stereo Matching*: Given a pair of stereo images taken from different positions, corresponding points in the two images can be used to compute a depth map of the distance to any point in the scene.

- *Super Resolution*: Given an input video, the resolution of each frame in the video can be increased by leveraging optical-flow techniques (Farsiu et al., 2004; Mitzel et al., 2009).

2.3 NVIDIA GPUs

We focus our attention on NVIDIA GPUs, as they are ubiquitous in CV and ADAS applications. First, we discuss the hardware comprising various NVIDIA GPUs, and then we provide an overview of CUDA, the API for GPU programming provided by NVIDIA.

2.3.1 NVIDIA GPU hardware

NVIDIA provides both discrete and integrated GPU families. An *integrated GPU* tightly shares DRAM memory with CPU cores, typically draws between 5 and 15 watts, and requires minimal cooling and additional space. Alternatively, *discrete GPUs* are packaged on adapter cards that plug into a host computer PCI-e bus, have their own local DRAM memory that is completely independent from that used by CPU cores, typically draw between 150 and 250 watts, need active cooling, and occupy substantial space.

As shown in Figure 2.6, the Jetson TX2 employs an SoC design that incorporates a quad-core 2.0-GHz 64-bit ARMv8 A57 processor, a dual-core 2.0-GHz superscalar ARMv8 Denver processor, and an integrated Pascal GPU. There are two 2-MB L2 caches, one shared by the four A57 cores and one shared by the two Denver cores. The GPU has two *streaming multiprocessors* (SMs), each providing 128 1.3-GHz cores that share a 512-KB L2 cache, and one *copy engine* (CE) used to copy data between the host CPU and the device GPU. The six CPU cores and integrated GPU share 8 GB of 1.866-GHz DRAM memory.

Discrete GPUs, such as the Titan V depicted in Figure 2.7, utilize their high space and energy requirements to provide significantly increased computational power compared to integrated GPUs. The Titan V boasts 80 SMs (compared to the TX2's two SMs), each with 64 1.46-GHz cores, and seven CEs (compared to the TX2's one).

For a simpler abstraction, programmers can think of a GPU as consisting of one or more CEs (*e.g.*, the TX2 has only one, whereas the Titan V has seven) that copy data between host memory and device memory, and an *execution engine* (*EE*) that has one or more SMs (*e.g.*, the TX2 has two SMs, the Titan V has 80)

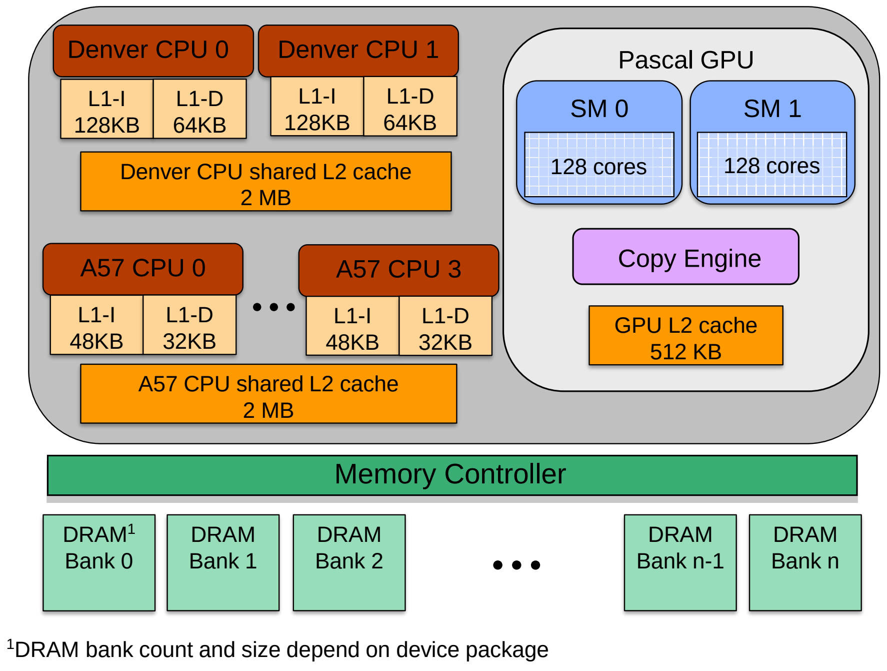


Figure 2.6: Architecture of the Jetson TX2.

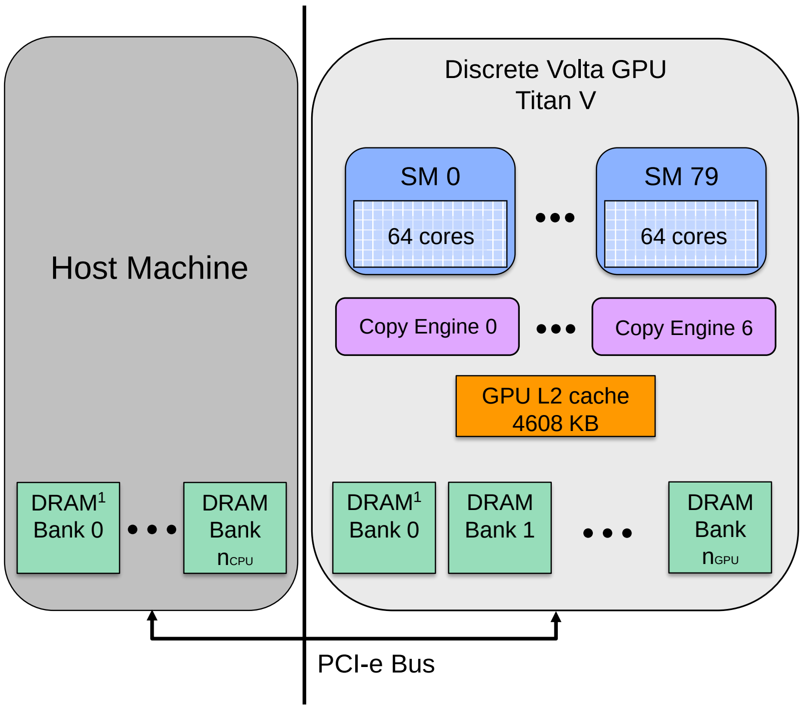


Figure 2.7: Architecture of a platform equipped with a Titan V.

Table 2.1: Details of NVIDIA GPUs we consider in this dissertation.

GPU	Type	Micro-architecture	Cores	CEs	SMs
Jetson TX2	Integrated	Pascal	256	1	2
GTX 980 Ti	Discrete	Maxwell	2816	2	22
GTX 1070	Discrete	Pascal	1920	2	15
Titan V	Discrete	Volta	5120	7	80

consisting of many cores that execute GPU kernels. An EE can execute multiple kernels simultaneously, but a CE can perform only one copy operation at a time. EEs and CEs can operate concurrently. Details regarding the different GPUs we consider in this dissertation are listed in Table 2.1.

2.3.2 CUDA Programming Fundamentals

GPUs act as co-processors, performing operations requested by CPU code. CUDA programs use a set of C or C++ library routines to request GPU operations implemented by a combination of hardware and device-driver software. The typical structure of a CUDA program is as follows: **(i)** allocate GPU-local (device) memory for data; **(ii)** use the GPU to copy data from host memory to GPU device memory; **(iii)** launch a program, called a *kernel*,[1] to run on the GPU cores to compute some function on the data; **(iv)** use the GPU to copy output data from device memory back to host memory; **(v)** free the device memory. An example vector-addition CPU procedure and associated kernel are given in Listing 1.

In the usual programming model for writing CUDA code, input data is partitioned among hardware threads on the GPU. When *launching* a kernel (line 27), the programmer specifies the distribution of GPU threads into *thread blocks* (line 22) and thread blocks into a *grid* (line 23). Thread-related system variables, such as the number of threads per block and the total number of blocks, are used by the CUDA kernel code to identify the specific data element(s) handled by a given thread. In the vector-addition example, each thread calculates the index i of the element it operates on in Line 3 using these CUDA-provided built-in variables.

Additional optional kernel-launch arguments include the per-block shared-memory requirement (line 24), and the FIFO stream of operations into which to launch the kernel (line 25). By default, no shared memory is used, and all kernels are launched into a special stream, called the NULL stream. Note that kernel launches are typically asynchronous with respect to the CPU,[2] so a stream synchronization (line 28) can be used to ensure the kernel has completed execution before copying the results back to the CPU.

[1]Unfortunate terminology, not to be confused with an OS kernel.

[2]See Sec. 3.2.6.1 of the CUDA Programming Guide (NVIDIA, 2019b) for more details.

Listing 1 Vector Addition Routine using CUDA.

```
    /* Performs single addition:  C[i] = A[i] + B[i] */
 1: static __global__ void vecAdd(int* A, int* B, int* C) {
 2:    // Calculate index using built-in thread, block info
 3:    int i = blockDim.x * blockIdx.x + threadIdx.x;
 4:    C[i] = A[i] + B[i];
 5: }

    /* Element-wise addition:  C = A + B, returns pointer to C
       (assumes n is a multiple of 32) */
 6: int * vectorAdd(int* A, int* B, int n) {
 7:    // Allocate CPU (host) memory for result array C
 8:    size_t bytes = n * sizeof(int);
 9:    int* C = (int *) malloc(bytes);
10:
11:    // Allocate GPU (device) memory for arrays A, B, and C
12:    int *d_A, *d_B, *d_C;
13:    cudaMalloc(&d_A, bytes);
14:    cudaMalloc(&d_B, bytes);
15:    cudaMalloc(&d_C, bytes);
16:
17:    // Copy arrays A and B from CPU to GPU memory
18:    cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
19:    cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);
20:
21:    // Launch the kernel
22:    int nt = 32;               // threads per block
23:    int nb = n / 32;           // blocks per grid
24:    int sm = 0;                // no shared memory used
25:    cudaStream_t stream;       // user-defined stream
26:    cudaStreamCreate(&stream);
27:    vecAdd<<<nb, nt, sm, stream>>>(d_A, d_B, d_C);
28:    cudaStreamSynchronize(stream);
29:    cudaStreamDestroy(stream);
30:
31:    // Copy results from GPU to CPU
32:    cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
33:
34:    // Free GPU memory for arrays A, B, and C
35:    cudaFree(d_A);
36:    cudaFree(d_B);
37:    cudaFree(d_C);
38:
39:    return C;
40: }
```

The example in Listing 1 also demonstrates common CUDA API functions, such as `cudaMalloc` (lines 13–15), `cudaMemcpy` (lines 18 and 19), and `cudaFree` (lines 35–37) used to allocate, copy to and from, and free GPU device memory, respectively. Unlike `cudaMalloc` and `cudaFree`, `cudaMemcpy` has an asynchronous variant, `cudaMemcpyAsync`, that can be used with user-defined streams for additional concurrency; `cudaMemcpyAsync` can return control to the calling CPU task before the copy is completed, whereas `cudaMemcpy` blocks the CPU task until the memory copy completes.

As seen in Listing 1, dispersing code to execute across one or more blocks, each with many threads, enables the significant parallelism afforded by GPUs to be exploited. Recall from Table 2.1 that the TX2, for example, contains 256 GPU cores, 128 per SM. Each SM is capable of running four units of 32 threads each (called a *warp*) at any given instant. The SM's four *warp schedulers* (contained in hardware) take advantage of stalls, such as waiting to access memory, to immediately begin running a different warp on the set of 32 cores assigned to it. In this way, the SM's warp schedulers can hide memory latency.

2.4 Real-Time Systems

We now formally define the real-time task models that we utilize. We first consider a set of independent tasks, and discuss the scheduling algorithms and analysis that apply to such a model. Then, we describe a graph-based model extension that enables us to leverage existing analysis for applications that are represented as graphs.

2.4.1 Real-Time Task Model

In this dissertation, we consider a task set τ comprised of n tasks: $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i releases a (potentially infinite) sequence of jobs: $J_{i,1}$, $J_{i,2}$, $\ldots$; we refer to an arbitrary job of task τ_i as J_i. Under the *periodic task model* (Liu and Layland, 1973), jobs are released exactly T_i time units apart; this is generalized under the *sporadic task model* (Mok, 1983), in which the task period T_i is the minimum, rather than exact, separation between job releases. Also associated with each task is a relative deadline D_i; a task is said to be *implicit-deadline* if $T_i = D_i$, *constrained-deadline* if $T_i \leq D_i$, and *arbitrary-deadline* if no relation is assumed between T_i and D_i.

The *absolute deadline* for job $J_{i,j}$ is given by $d_{i,j} = r_{i,j} + D_i$, where $r_{i,j}$ is the *release time* of the job. The first job of task τ_i has a non-negative release *offset* given by Φ_i (*i.e.*, $r_{i,1} = \Phi_i$). Each job of τ_i performs work with *worst-case execution time (WCET)* given by C_i. The *utilization* of a task indicates the fraction of a single processor's capacity that it requires: $u_i = C_i/T_i$. The utilization of the entire task set τ is given by $U = \sum_i u_i$.

Given these definitions, each task can be represented as a four-tuple: $\tau_i = (\Phi_i, T_i, C_i, D_i)$. These parameters are depicted in Figure 2.8 for a task defined as $\tau_i = (2, 6, 2, 5)$. If not specified otherwise, we assume

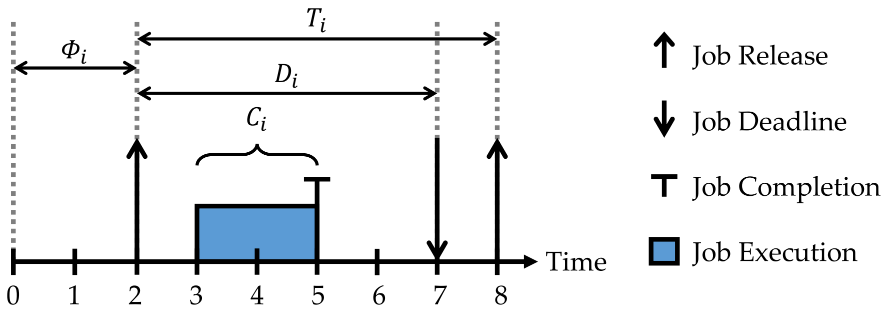


Figure 2.8: Illustration of the real-time task model for a task $\tau_i = (2,6,2,5)$.

synchronous releases (*i.e.*, $\forall i : \Phi_i = 0$), and implicit-deadline tasks, and thus represent a task as only a two-tuple: $\tau_i = (T_i, C_i)$. The traditional sporadic task model assumes that tasks must execute sequentially, *i.e.*, that no two jobs of the same task may execute concurrently; we briefly discuss an alternative to this model in Section 2.4.3, and introduce a generalized task execution model in Section 3.1.

We denote the *completion* (or *finish*) *time* of job $J_{i,j}$ as $f_{i,j}$. We can therefore refer to the *response time* of $J_{i,j}$ as $f_{i,j} - r_{i,j}$, and the *tardiness* of $J_{i,j}$ as $\delta_{i,j} = \max\{0, f_{i,j} - d_{i,j}\}$. For the job depicted in Figure 2.8, $r_{i,1} = 2$, $f_{i,1} = 5$, its response time is $5 - 2 = 3$, and $\delta_{i,1} = 0$.

2.4.2 Scheduling Algorithms

We assume that τ is scheduled on a platform with m CPUs; a *scheduling algorithm* is used to allocate processing time to tasks within τ. A scheduler is said to be *static* if it does not depend on jobs' actual execution times and releases (*i.e.*, it can be generated offline). For example, a table-driven scheduler uses a previously generated table to determine which task may execute jobs at any given time. *Dynamic* schedulers, on the other hand, make scheduling decisions at runtime, based on how the jobs in the system are prioritized. In this dissertation, we focus primarily on dynamic scheduling.

Dynamic schedulers can further be classified depending on the task or job parameters used to calculate a job's priority. *Static-priority* schedulers make decisions based on task parameters. For example, the rate-monotonic (RM) (Liu and Layland, 1973) and deadline-monotonic (DM) (Audsley et al., 1991) scheduling algorithms assign job priorities based on task periods and relative deadlines, respectively. *Dynamic-priority* schedulers, on the other hand, rely on job parameters to assign job priorities, and therefore may assign a different priority to each job of a given task. For example, the FIFO scheduling algorithm assigns higher priorities to jobs with earlier release times, whereas earliest-deadline-first (EDF) (Liu and Layland, 1973) assigns higher priorities to jobs with earlier absolute deadlines. The static-priority and dynamic-priority scheduling algorithms just discussed are *job-level fixed-priority (JLFP)* schedulers, *i.e.*, they assume that a job's priority is fixed once assigned; schedulers for which a job's priority may change after release are beyond the scope of this dissertation, and thus are not discussed here.

A scheduling approach is said to be *partitioned* if each task in τ is assigned to a specific CPU, and *global* if tasks can migrate between CPUs. From an implementation perspective, this corresponds to having one runqueue per CPU, or a global runqueue from which tasks are scheduled on any CPU. *Clustered* scheduling generalizes these two extremes by assigning each task to a cluster of CPUs; tasks are scheduled globally within a given cluster. Other alternatives, such as *semi-partitioned* (Anderson et al., 2008) and *federated* (Li et al., 2014) scheduling, allow some tasks to be assigned to specific processors while others may migrate freely between all or a given set of processors.

We can refer to a scheduler based on its job prioritization and the assignment of tasks to processors, *e.g.*, global EDF (G-EDF) corresponds to tasks scheduled globally using EDF. Scheduling algorithms can also be classified by whether they allow jobs to be preempted during execution. Because FIFO prioritizes jobs by release time, it is naturally non-preemptive. By default, RM, DM, and EDF are all assumed to be preemptive; non-preemptive EDF is specified as NP-EDF.

2.4.3 Schedulability

A *schedulability test* can be used to determine whether a task system is schedulable on a given platform using a given scheduling algorithm. For example, on a uniprocessor platform, the hard-real-time schedulability test for EDF and implicit-deadline tasks is a simple utilization-based test:

$$U \leq 1. \tag{2.1}$$

The schedulability test in Equation (2.1) is an *exact* test: if a task set violates this condition, then jobs may miss deadlines (in fact, this is guaranteed if jobs are released periodically and every jobs executes for its WCET).

On a multiprocessor platform, no exact utilization-based schedulability test exists for sporadic tasks with hard deadlines when scheduled under G-EDF. Instead, existing tests are merely *sufficient* (*i.e.*, there exist schedulable task sets for which a given test fails) (Baker, 2003; Baruah, 2007; Baruah and Baker, 2008a,b; Bertogna et al., 2005, 2008). For a set of periodic implicit-deadline tasks with hard deadlines, a sufficient utilization-based test was presented by Andersson *et al.* (2001):

$$U \leq (m+1)/2. \tag{2.2}$$

Unfortunately, the condition in Equation (2.2) is subject to *utilization loss*: a task set is deemed unschedulable if its utilization exceeds approximately half of the available processing capacity.

Recall from Section 1.1 that our focus in this dissertation is on soft real-time tasks for which response times must be bounded. Devi and Anderson proved a simple exact utilization-based test (Devi and Anderson, 2008) for determining whether tardiness and hence response times of all tasks in τ are bounded:

$$U \leq m. \tag{2.3}$$

This test does not result in the utilization loss present in Equation (2.2); as long as the m processors are not overutilized, then response times are bounded.

Devi and Anderson also proved the following upper bound on the tardiness of each task τ_i scheduled under G-EDF, assuming the condition in Equation (2.3) holds (Devi and Anderson, 2008):

$$\max_j \delta_{i,j} \leq \frac{C^\Lambda - C_{min}}{m - U^{\Lambda-1}} + C_i, \tag{2.4}$$

where C^k is the sum of the k highest task WCETs, U^k is the sum of the k highest task utilizations, and $\Lambda = \lceil U \rceil - 1$.

The response-time analysis from Devi and Anderson (2008) was extended by Leontyev and Anderson to ensure bounded response times for sequential sporadic tasks under any G-EDF-like (GEL) scheduler (Leontyev and Anderson, 2010) as long as Equation (2.3) holds. This includes any global JLFP scheduler with *window-constrained* job priorities, *i.e.*, schedulers for which a job's priority is chosen based on a time instant, called a *priority point*, within a bounded window of the job's release and deadline. For example, FIFO and G-EDF fall in this class, as they use a job's release and actual deadline, respectively, as priority points.

The schedulability tests discussed so far assume that tasks must execute sequentially (in fact, the sequential sporadic task model requires that $u_i \leq 1.0$). If this requirement is relaxed, then any number of jobs of the same task may execute in parallel. Erickson and Anderson explored this fully parallel task model (Erickson and Anderson, 2011), for which u_i can exceed 1.0, and showed that enabling such intra-task parallelism can result in lower per-task response-time bounds, as long as Equation (2.3) holds. In Section 3.1, we present a task model that generalizes both the sequential and fully parallel sporadic task models by introducing a per-task parameter that specifies the possible intra-task parallelism and utilization bound.

2.4.4 Graph-Based Tasks

Many CV applications can be represented as dataflow graphs. In order to leverage existing real-time schedulability analysis, such graph-based task systems can be converted to sets of sporadic tasks. We now discuss one such representation of dataflow graphs for CV applications, and give an overview of the transformation process from graphs to sets of sporadic tasks.

2.4.4.1 OpenVX

To facilitate the development of CV algorithms, the OpenVX standard was first ratified in 2014 (The Khronos Group, 2014). OpenVX allows programmers to specify CV algorithms as dataflow graphs by interconnecting high-level CV primitives.

In OpenVX, primitives and the data objects upon which they operate comprise a bipartite graph (The Khronos Group, 2018). An OpenVX graph $\mathcal{G}^k$ contains data objects $D_1^k, \ldots, D_{y_k}^k$ and nodes $N_1^k, \ldots, N_{z_k}^k$. An edge $\left(N_v^k, D_w^k\right)$ corresponds to a data object D_w^k that is written by node N_v^k, and $\left(D_w^k, N_v^k\right)$ corresponds to a data object read by node N_v^k. Data objects can optionally be *delay objects*, indicating that the data from prior time steps must be buffered for later use. Associated with each delay object is a value indicating the age, in time steps, of the data.

To simplify analysis, we assume that each graph has a single source node and a single sink node. (If this is not the case, then a single "virtual" source and/or sink can be added.) For all graphs we consider, we assume that the first indexed node (N_1^k for an OpenVX graph $\mathcal{G}^k$) is the source.

Example 2.1. An example OpenVX graph $\mathcal{G}^1$ is shown in Figure 2.9. In this figure, rectangles correspond to the eight data objects, $D_1^1, \ldots, D_8^1$, and round nodes indicate the four primitives, $N_1^1, \ldots, N_4^1$, that act on them. There are three delay objects, D_3^1, D_5^1, and D_6^1, with delay values 1, 3, and 2, respectively, shown as insets in the delay object boxes. $\Diamond$

The OpenVX standard specifies a series of rules for processing graphs (The Khronos Group, 2018). The rules relevant to the work in this dissertation are:

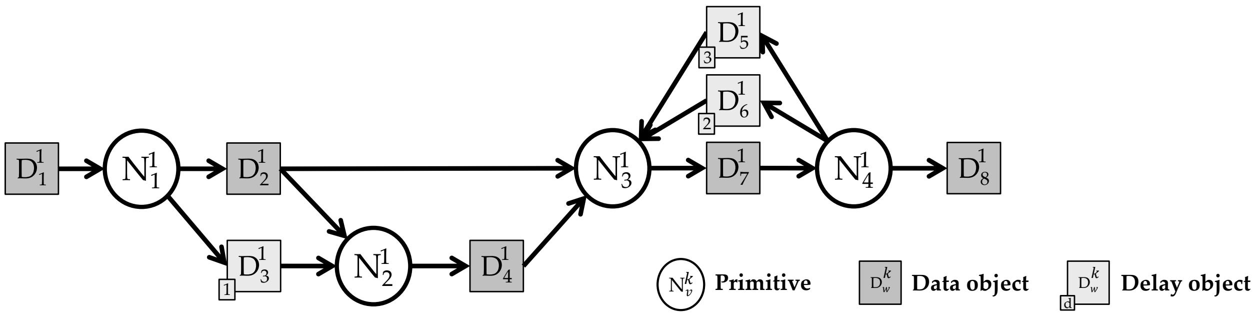


Figure 2.9: An example OpenVX graph $\mathcal{G}^1$.

1. *Single Writer*: Every data object has at most one incoming edge.

2. *Broken Cycles*: Every cycle in $\mathcal{G}^k$ must contain at least one input edge $\left(D_w^k, N_v^k\right)$ where D_w^k is a delay object.

Example 2.1 (continued). In $\mathcal{G}^1$, every data object has at most one incoming edge (although D_2^1 has two outgoing edges). Additionally, there are two cycles, containing edges from delay objects D_5^1 and D_6^1 to node N_3^1. ◊

2.4.4.2 Transforming OpenVX Graphs to Sporadic Task Sets

An OpenVX graph can be transformed into an "equivalent" set of independent sporadic tasks (Liu and Anderson, 2011; Yang et al., 2015, 2018a), for which response-time analysis exists (Devi and Anderson, 2008; Erickson and Anderson, 2011; Erickson et al., 2010; Leontyev and Anderson, 2007a,b, 2010). This process is depicted in Figure 2.10. Note that Step 3, as originally proposed (Yang et al., 2015), requires that the utilization of each cycle is at most 1.0.

The transformation process must be performed for each OpenVX graph $\mathcal{G}^k$ in a system. We now illustrate each step in detail.

Step 1: From a coarse- to a fine-grained OpenVX graph. The OpenVX standard specifies little about the concurrent execution of primitives within a graph. M. Yang *et al.* (2018a) showed that treating each primitive as a schedulable entity can be too coarse-grained to guarantee bounded response times, as pessimistic inflation of GPU execution times can result in utilization-based feasibility conditions not being met. Rather, primitives that utilize both a CPU and GPU should be split into multiple nodes, with each executing on either a CPU or a GPU. (Note that we assume the mapping of primitives to processor types is decided by the application

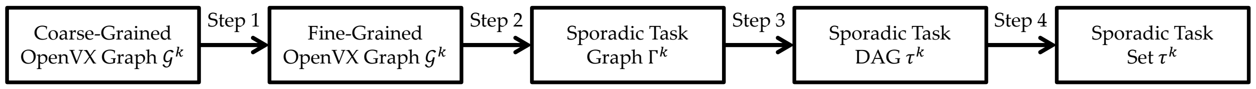

Coarse-Grained OpenVX Graph $\mathcal{G}^k$
Step 1
Fine-Grained OpenVX Graph $\mathcal{G}^k$
Step 2
Sporadic Task Graph Γ^k
Step 3
Sporadic Task DAG τ^k
Step 4
Sporadic Task Set τ^k

Figure 2.10: The transformation process from an OpenVX graph to a set of sporadic tasks.

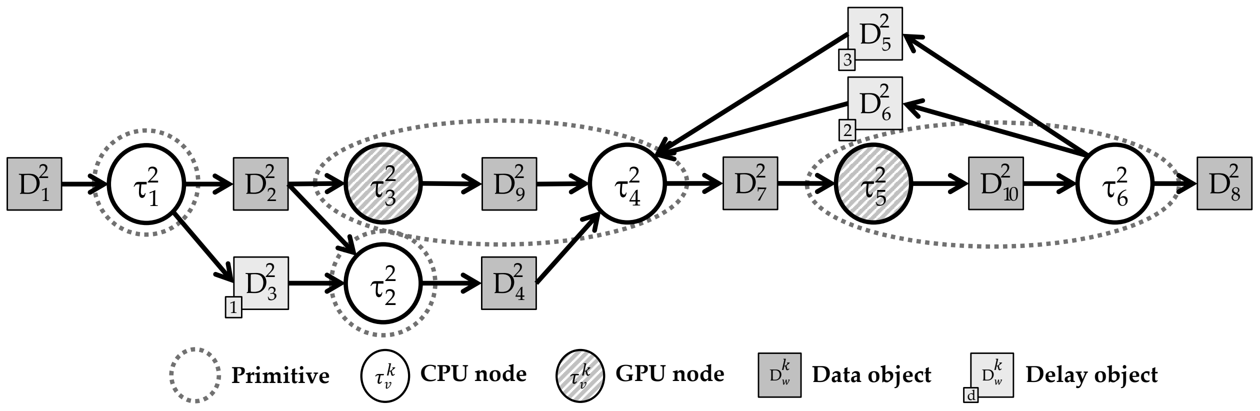


Figure 2.11: A fine-grained OpenVX graph $\mathcal{G}^2$ corresponding to the coarse-grained graph in Figure 2.9.

designer.) For the DAGs considered by M. Yang *et al.*, this transformation enabled the separate computation of response-time bounds for CPU and GPU nodes; in their DAG-based case study, feasibility conditions were satisfied for the fine-grained graph representation.

Example 2.2. We illustrate the transformation process with a continuing example. Figure 2.11 depicts a fine-grained OpenVX graph $\mathcal{G}^2$ corresponding to the coarse-grained OpenVX graph $\mathcal{G}^1$ from Figure 2.9. Graph $\mathcal{G}^2$ contains six nodes (four CPU nodes and two GPU nodes) and ten data objects.

Primitives N_3^1 and N_4^1 have been decomposed into nodes $\{\tau_3^2, \tau_4^2\}$ and $\{\tau_5^2, \tau_6^2\}$, respectively, with additional data objects D_9^2 and D_{10}^2 added between the new nodes. Additionally, each node in Figure 2.11 is shaded based on whether that node executes on a CPU or a GPU. $\Diamond$

Step 2: From a fine-grained OpenVX graph to a sporadic task graph. We now formalize the graph-based task model. As our final transformation target is a set of sporadic tasks, we refer to the graph produced by Step 2 as a *sporadic task graph* Γ^k. We add notation to the task model introduced in Section 2.4.1 to retain information aout the graph to which a given task corresponds: a sporadic task graph Γ^k is comprised of z_k nodes, $\tau_1^k, \ldots, \tau_{z_k}^k$, with each node corresponding to a task. Similarly, we refer to the jth job of task τ_v^k as $J_{v,j}^k$. Edges in Γ^k indicate producer/consumer relationships between tasks: a job must wait to begin execution until the corresponding job of each task from which it consumes data (*i.e.*, for each edge for which it is a consumer) has completed.

Given a fine-grained OpenVX graph $\mathcal{G}^k$, we can perform a simple transformation to obtain a sporadic task graph Γ^k:

- Each node τ_v^k in $\mathcal{G}^k$ becomes a node τ_v^k in Γ^k.
- Each input edge $\left(D_w^k, N_v^k\right)$ in $\mathcal{G}^k$, other than that into the source τ_1^k, becomes a directed edge $\left(\tau_u^k, \tau_v^k\right)$ in Γ^k, where τ_u^k is the single writer of data object D_w^k.
- An edge in Γ^k is a *delay edge* if its corresponding data object D_w^k in $\mathcal{G}^k$ is a delay object, and a *regular edge* otherwise.
- Multiple edges of the same type between the same pair of nodes are merged into a single edge of that type.

Note that delay edges can be either forward or backward edges, depending on whether they result in a cycle in the graph. For each delay edge $\left(\tau_v^k, \tau_u^k\right)$, we include a range $[p,q]$, $p \leq q$, corresponding to the range of delay values for that edge (note that, for simplicity of notation, we omit subscripts and superscripts for delay edges' ranges when the edge in question is clear). Thus, a delay edge $\left(\tau_v^k, \tau_u^k\right)$ with range $[p,q]$ indicates that a job $J_{u,j}^k$ relies on the outputs of $\{J_{v,j-q}^k, \ldots, J_{v,j-p}^k\}$.

Example 2.2 (continued). Figure 2.12 shows the sporadic task graph Γ^2 corresponding to the fine-grained OpenVX graph from Figure 2.11. The three delay objects are represented as two delay edges, one forward and one backward. The delay values are encapsulated in the p and q values for the delay edges. $\Diamond$

Step 3: From a sporadic task graph to a sporadic task DAG. K. Yang *et al.* (2015) provided a series of rules for removing delay edges from a graph Γ^k, resulting in a DAG τ^k, with nodes corresponding to sporadic tasks. K. Yang *et al.* showed that forward delay edges can simply be replaced by regular edges, and they proposed to break cycles by combining all nodes in a given cycle in a graph into a single *supernode*.

Example 2.2 (continued). Figure 2.13 shows the DAG τ^2 derived from the cyclic graph Γ^2 in Figure 2.12. The forward delay edge from τ_1^2 to τ_2^2 has been removed because a regular edge between these nodes exists, and nodes τ_4^2, τ_5^2, and τ_6^2 comprising the cycle have been combined into a single supernode τ_{456}^2. $\Diamond$

Step 4: From a sporadic task DAG to a sporadic task set. Given a sporadic task DAG τ^k, it is straightforward to consider each node as an independent sporadic task. Each task τ_v^k has a worst-case execution time

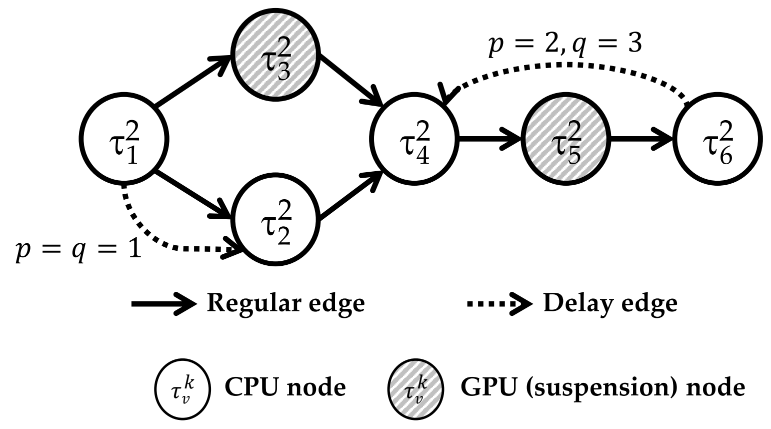


Figure 2.12: A sporadic task graph Γ^2 derived from the fine-grained OpenVX graph $\mathcal{G}^2$ in Figure 2.11.

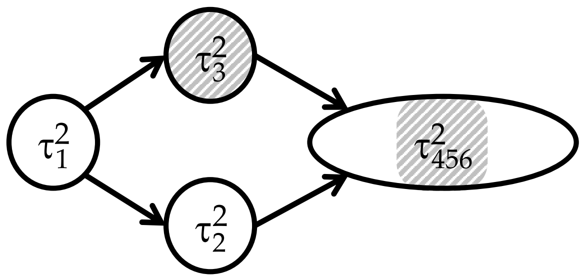


Figure 2.13: A sporadic task DAG τ^2 derived from the cyclic graph from Figure 2.12.

given by C_v^k and a relative deadline given by D_v^k. All tasks belonging to τ^k share a period T^k. Jobs of the source task τ_1^k are assumed to be released sporadically, at least T^k time units apart.

For non-source tasks, Liu and Anderson (2011) showed how response-time bounds R_u^k of tasks τ_u^k that produce data consumed by τ_v^k can be used to set an offset Φ_v^k. This offset specifies the release time of a job $J_{v,j}^k$ relative to that of its graph's corresponding source job $J_{1,j}^k$ (thus, the source task has $\Phi_1^k = 0$). For a non-source task τ_v^k, its offset is the sum of the offsets and response-time bounds for each incoming edge:

$$\Phi_v^k = \max_{\substack{u \\ \left(\tau_u^k, \tau_v^k\right) \text{is an edge in} \tau^k}} \{\Phi_u^k + R_u^k\}. \tag{2.5}$$

The end-to-end response-time bound for a DAG τ^k is given by the sum of the offset and response-time bound of its sink node: $R^k = \Phi_{z_k}^k + R_{z_k}^k$.

Example 2.2 (continued). Figure 2.14 depicts an example schedule for the task set derived from the sporadic task DAG τ^2 in Figure 2.13. In this example, we assume response-time bounds of the four DAG tasks have

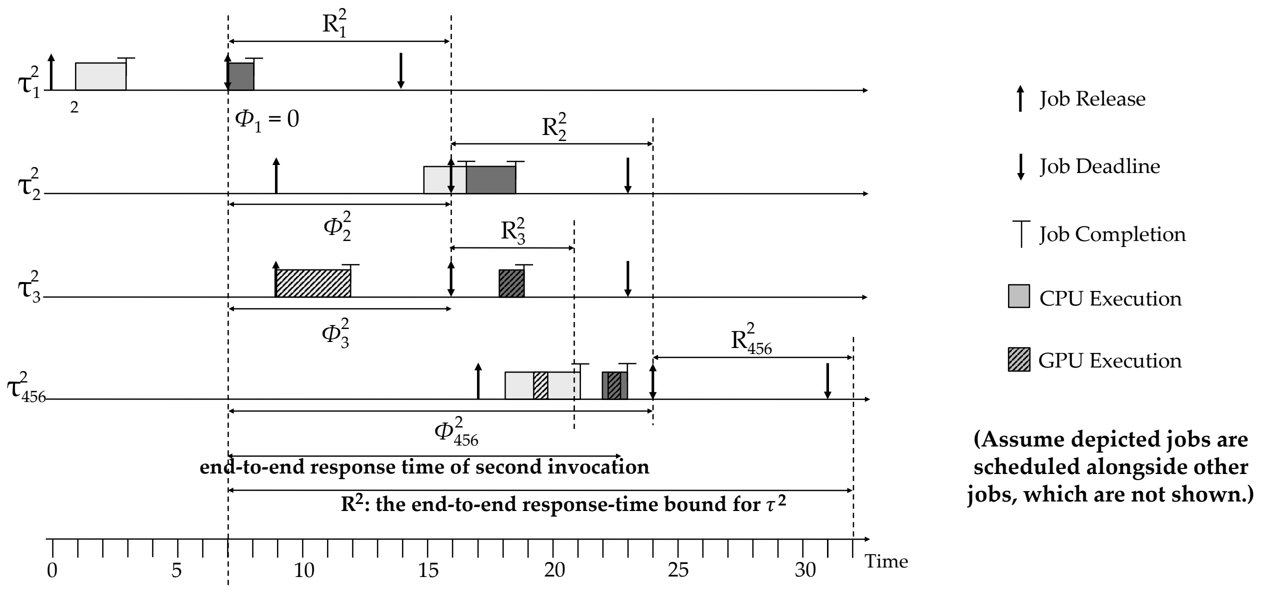


Figure 2.14: A possible schedule of two sets of jobs of the sporadic tasks in τ^2 in Figure 2.13. The second job of each task is shaded darker than the first.

been computed to be $R_1^2 = 9$, $R_2^2 = 8$, $R_3^2 = 5$, and $R_{456}^2 = 8$. Per-task response-time bounds and offsets are labeled in Figure 2.14 for the second invocation of the graph.

As τ_1^2 is the source node of τ^2 in Figure 2.13, it is defined to have $\Phi_1^2 = 0$. τ_2^2 and τ_3^2 each have only a single incoming edge, from τ_1^2. By Equation (2.5), their offsets are given by $\Phi_2^2 = \Phi_3^2 = \Phi_1^2 + R_1^2 = 0 + 9 = 9$ time units. The sink node, τ_{456}^2, has two incoming edges, from τ_2^2 and τ_3^2. Thus, by Equation (2.5), its offset is computed as $\Phi_{456}^2 = \max\{\Phi_2^2 + R_2^2, \Phi_3^2 + R_3^2\} = \{9+8, 9+5\} = 17$ time units.

The end-to-end response-time bound for the entire DAG τ^2 is based on the offset and response-time bound for the sink node: $R^2 = \Phi_{456}^2 + R_{456}^2 = 17 + 8 = 25$ time units. $\Diamond$

Note that as we focus on soft-real-time applications, task deadlines are used here to define priorities rather than strict (hard) timing constraints, and thus R_v^k may exceed D_v^k, *i.e.*, jobs may complete after their deadlines. Additionally, a job $J_{v,j}^k$ can be scheduled as soon as its predecessors (*i.e.*, previous jobs of the same task, depending on whether jobs of the same task execute sequentially, as well as all jobs $J_{u,j}^k$ from the same graph invocation, for which $\left(\tau_u^k, \tau_v^k\right)$ is an edge in the DAG τ^k) have completed. Such *early releasing* may occur before a job's actual release time, as long as its absolute deadline (and thus priority) remain unchanged (Devi, 2006). Liu and Anderson showed that early releasing can improve response times under G-EDF scheduling (Liu and Anderson, 2011).

Example 2.2 (continued). In Figure 2.14, job $J_{2,1}^2$ completes at time 16.5, after its deadline at time 16. Thus, job $J_{2,1}^2$ has a response time of 7.5 time units; however, this is less than the bound of $R_2^2 = 8$ time units.

Assuming that jobs of the same task must execute sequentially, job $J_{456,2}^2$ has three predecessors: $J_{2,2}^2$ and $J_{3,2}^2$ due to graph edges, and $J_{456,1}^2$ due to sequential execution. These jobs complete at times 18.5, 19, and 21, respectively. Thus, job $J_{456,2}^2$ is eligible for early releasing at time 21 and begins execution at time 22, even though its actual release does not occur until time 24. $\Diamond$

2.5 Chapter Summary

In this chapter, we have reviewed the algorithms necessary for ADAS development, software libraries used to implement these algorithms, details of GPUs used to accelerate them. We have additionally formally discussed existing real-time task models and scheduling algorithms. However, some of the applications discussed in Section 2.1 can be represented as graphs containing cycles, for which the existing response-time analysis discussed in Section 2.4 may not apply; we introduce a new task model in Chapter 3 that addresses this challenge, and we evaluate the trade-off that arises in its application. Many details of the GPUs discussed in Section 2.3 are missing from NVIDIA documentation, jeopardizing in their use in safety-critical applications. In Chapter 4, we explore the scheduling policies of NVIDIA GPUs, and discuss potential pitfalls of GPU use in detail, including a case study of some of the applications listed in Section 2.2. The planning, perception, and control components discussed in Section 2.1 may be developed independently but need to coexist on the same hardware platform; in Chapter 5, we introduce a framework to ensure temporal isolation between such components in the presence of non-preemptive GPU use. Finally, in Chapter 6, we present case-study evaluations of the results in prior chapters using several of the applications discussed in Section 2.1.

CHAPTER 3: THE HISTORY VS. RESPONSE TIME VS. ACCURACY TRADE-OFF[1]

In this chapter, we present the rp-sporadic (restriced parallelism) task model, which enables scheduling of tasks with utilization exceeding 1.0 by allowing a limited number of jobs of the same task to execute in parallel. Idle CPU time may be reduced by the execution of additional jobs of the same task, increasing throughput. Furthermore, enabling such intra-task parallel execution may result in lower response-time bounds.

However, for graph-based workloads containing cycles, allowing intra-task parallelism results in a trade-off between the degree of parallelism, the response-time bounds of the application, and the accuracy of the resulting computations. Added parallelism may lead to historical data used within the cycle being unavailable, and the use of older history may decrease application accuracy; the age of such historical data is directly related to the amount of available parallelism. In this chapter, we explore this trade-off in detail. For the history-versus-response-time side of the trade-off, we explore the impact on analytical response-time bounds as parallelism increases for a large set of synthetically generated graph-based task systems for which per-task utilizations may exceed 1.0. To investigate the history-versus-accuracy trade-off, we consider a multi-object tracking (MOT) system in which pedestrians and vehicles are tracked via images recorded by a camera attached to a moving vehicle, and measure the tracking accuracy as the age of historical data varies.

Organization. The remainder of this chapter is organized as follows. In Section 3.1, we discuss the drawbacks of existing task-execution models and formally define the rp-sporadic task model. Then, in Section 3.2, we detail our experiments to evaluate the history-versus-response-time trade-off. We describe our evaluation of the history-versus-accuracy trade-off in Section 3.3 before concluding in Section 3.4. Note that we consider the full history-versus-response-time-versus-accuracy trade-off in Chapter 6, in which we

[1]Contents of this chapter previously appeared in preliminary form in the following papers:

Amert, T., Voronov, S., and Anderson, J. H. (2019). OpenVX and real-time certification: The troublesome history. In *Proceedings of the 40th IEEE Real-Time Systems Symposium*, pages 312–325.

Amert, T., Yang, M., Nandi, S., Vu, T., Anderson, J. H., and Smith, F. D. (2020). The price of schedulability in multi-object tracking: The history-vs.-accuracy trade-off. In *Proceedings of the 23rd IEEE International Symposium on Real-Time Distributed Computing*, pages 124–133.

combine the results of this chapter to explore the impact on accuracy of our MOT applications when executing in the presence of other automotive-based applications under a real-time scheduler.

3.1 Rp-Sporadic Task Model

As detailed in Section 2.4.4.2, prior work has shown how to transform a graph into a set of independent sporadic tasks. The scheduling of these tasks depends on the choice of intra-task parallelism. We first consider both sequential and fully parallel task execution, and then introduce our generalized concept of restricted parallelism. Finally, we formally define our rp-sporadic task model.

3.1.1 Existing Task-Execution Models

The traditional sporadic task model assumes that jobs of the same task must execute sequentially, as discussed in Section 2.4. Erickson and Anderson considered a variant of the sporadic task model in which full parallelism was allowed for tasks (Erickson and Anderson, 2011), which we refer to as *fully parallel* execution.

Example 3.1. Figure 3.1 depicts possible schedules for jobs of the supernode τ^2_{456} from Figure 2.13 on a platform with four CPUs and one GPU, assuming $T^2 = 5$, $C^2_{456} = 6$, and $R^2_{456} = 21$. Each job $J^2_{456,j}$ is released at time $r^2_{456,j}$; the schedule is shown starting at time 100 when job $J^2_{456,21}$ is released. Successive jobs are shaded progressively darker, and the depicted jobs are assumed to be scheduled alongside other jobs in the system, which are not shown.

In schedule (a), the jobs execute sequentially. Due to jobs of other tasks (not shown), $J^2_{456,21}$ is released at time 100, but not scheduled until time 114. This postponement impacts the subsequent jobs; $J^2_{456,24}$ has a response time of 7.4. However, the $p = 2$ requirement is met, *i.e.*, $J^2_{456,21}$ completes before $J^2_{456,23}$ begins.

Schedule (b) shows the result of fully parallel execution. We assume GPU computations are FIFO scheduled, which causes three of the four jobs' execution times to increase due to blocking waiting to access the GPU. However, the response time of $J^2_{456,24}$ is reduced to 3.2 time units. $\Diamond$

Unfortunately, unrestricted intra-task parallelism creates two problems. First, the jobs of a task can complete out of order; however, this can be simply resolved by buffering job outputs, as discussed by Elliott *et al.* (2015). Second, and more importantly, such parallelism can violate the dependencies required by backward delay edges. Yang *et al.* (2015) proved that if $p = 1$ for some backward delay edge, then no two

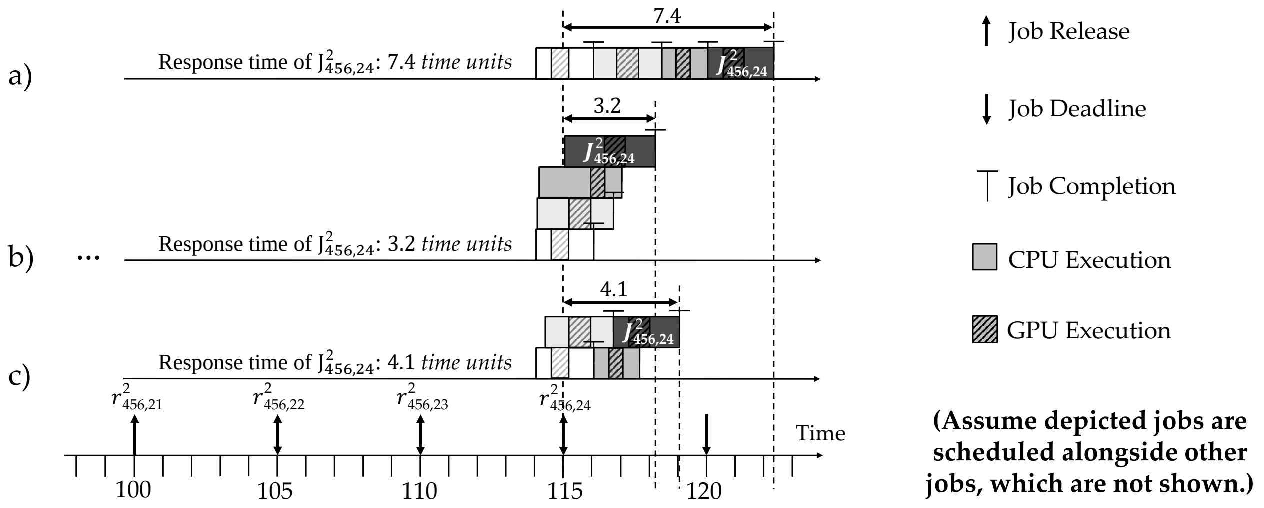


Figure 3.1: Scheduling repercussions of the degree of intra-task parallelism, including a) sequential execution, b) fully parallel execution, and c) restricted parallelism.

jobs of any task in that cycle may execute in parallel. This proof can be generalized to show that if more than p jobs of a task in a cycle execute concurrently, then a precedence constraint must be violated.

Example 3.1 (continued). Recall that the supernode τ^2_{456} was created from a cycle with $p = 2$ (see Figures 2.12 and 2.13). Thus, job $J^2_{456,23}$ requires output from job $J^2_{456,21}$. However, in schedule (b) of Figure 3.1, jobs $J^2_{456,21}$, $J^2_{456,22}$, and $J^2_{456,23}$ all execute concurrently, violating this precedence constraint. ♢

We define the *utilization* of τ^k_v to be $u^k_v = C^k_v/T^k$. The utilization of the entire graph-based task system is given by $U = \sum_{\tau^k \in \tau} \sum_{\tau^k_v \in \tau^k} u^k_v$. We can define the utilization of a cycle similarly: $\sum_{\tau^k_v \in \tau'} u^k_v$, where τ' is the set of tasks in the cycle.

Response-time analysis for sequential sporadic tasks requires $u^k_v \leq 1.0$ for all tasks. In their transformation, Yang *et al.* extended this requirement to supernodes (Yang et al., 2015): if sequential execution is assumed, the utilization of each cycle must be at most 1.0.

Example 3.1 (continued). If jobs execute sequentially as in Figure 3.1(a), response times can be unbounded for τ^2_{456}, as $u^2_{456} = 6/5$. ♢

Unfortunately, if smaller bounds are desired or if the cycle has utilization greater than 1.0, no existing analysis can be applied. Furthermore, cycles with utilization exceeding 1.0 can easily occur in actual CV graphs. When full intra-task parallelism is enabled, $u^k_v \leq 1.0$ is no longer required, but historical requirements may not be met.

3.1.2 Allowing Restricted Parallelism

Our work bridges this parallelism divide, resulting in response-time bounds for sporadic task graphs (and thus OpenVX graphs) that prior work deemed infeasible. We provide a new restricted-parallelism sporadic task model that specifies intra-task parallelism on a per-task basis. A key feature of our approach is that *per-task utilizations are allowed to exceed* 1.0*, yet parallelism (and hence accuracy) is controlled.*

Example 3.1 (continued). Restricted intra-task parallelism is shown in schedule (c) of Figure 3.1. The response time of $J^2_{456,24}$ is increased to 4.1, but the history requirements are respected, as only $p = 2$ jobs of τ^2_{456} execute concurrently. $\Diamond$

Abstracting GPU computations. Although M. Yang *et al.* (2018a) suggested considering CPU and GPU tasks separately in response-time analysis, their results hold only for DAGs. Instead, as did K. Yang *et al.* (2015), we arbitrate access to the GPU with a multiprocessor locking protocol (*e.g.*, the global OMLP (Brandenburg and Anderson, 2010)). Thus, we henceforth assume that all graph nodes are CPU nodes, with their worst-case execution times inflated to include GPU blocking and execution time, and that tasks can contain non-preemptive regions due to said locking protocol.

Transforming cycles, revisited. We leverage the supernode concept from K. Yang *et al.* (2015) to transform a sporadic task graph Γ^k into a sporadic task DAG τ^k. We supplement each node τ^k_v of the DAG with a value P^k_v indicating the allowed intra-task parallelism for jobs of that task.

Recall from Section 2.4.4.2 that p and q define the range of delay values for a delay edge in a sporadic task graph. All tasks within a cycle are combined into a single supernode τ^k_u, with P^k_u defined to be the smallest p of any forward or backward delay edge contained in the cycle (we do not use q, as it is does not limit the parallelism of the cycle). A task τ^k_v that is not part of any cycle has $P^k_v = m$, *i.e.*, unrestricted intra-task parallelism, as assumed by M. Yang *et al.* (2018a).

Example 3.2. Figure 3.2 depicts the DAG that results from our parallelism-aware supernode transformation.[2] The nodes correspond to those in Figure 2.13, and are labeled with their intra-task parallelism values P^k_v. For tasks that are not supernodes, the intra-task parallelism is m. Task τ^2_{456} is a supernode derived from a cycle with $p = 2$ in Figure 2.12, so it has $P^2_{456} = 2$. $\Diamond$

[2]Note that, while we ended up with the same number of compute nodes as in the original coarse-grained graph in Figure 2.9, this will generally not be the case.

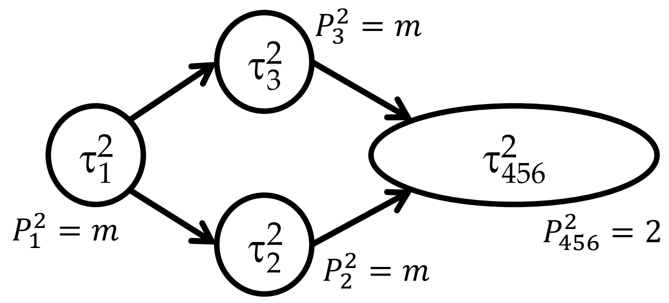


Figure 3.2: Intra-task parallelism for nodes of τ^2 from Figure 2.13.

Offset computation for forward delay edges. In prior work, forward delay edges were either deemed as out of scope (Yang et al., 2018a) or supported assuming only sequential task execution (Elliott et al., 2014; Yang et al., 2015). We propose a different method for handling such edges here.

Consider a forward delay edge $\left(\tau_u^k, \tau_v^k\right)$ with delay value p. Denote the offset of τ_v^k computed in a DAG without the delay edge as $\Phi_v'^k$. The forward delay edge adds the requirement that a job $J_{v,j}^k$ must not start earlier than the completion of $J_{u,j-p}^k$, p DAG periods prior. Thus, we require $\Phi_v^k \geq \Phi_u^k + R_u^k - p \cdot T^k$. At the same time, we require $\Phi_v^k \geq \Phi_v'^k$. Combining both expressions, we have $\Phi_v^k = \max(\Phi_v'^k, \Phi_u^k + R_u^k - p \cdot T^k)$.

Note that, because offsets are determined from source to sink (Liu and Anderson, 2011), by the definition of a forward delay, Φ_u^k is available when Φ_v^k is determined. Note also that the method above can be generalized for the case wherein forward delay edges are directed from several nodes to the node τ_v^k.

K. Yang *et al.* (2015) proposed instead to replace each forward delay edge with a regular forward edge. Effectively, such a replacement is equivalent to the computation of Φ_v^k with $p = 0$, so our approach generalizes theirs.

3.1.3 The Rp-Sporadic Task Model

We now formally define the *rp-sporadic task model*, which permits per-task parallelism to be specified. Under this model, the i^{th} task is specified as $\tau_i = (\Phi_i, T_i, C_i, P_i)$, where Φ_i, T_i, and C_i are as defined in Section 2.4.4 (but omitting the graph index, as it is not relevant to us here), and P_i is the parallelism level allowed for jobs of τ_i.

As in existing response-time analysis, we require

$$U \leq m, \tag{3.1}$$

or the entire system can become overutilized, with response times being unbounded. Additionally, at most P_i jobs of a task τ_i can execute at once, so we require

$$\forall i: \; u_i \leq P_i. \tag{3.2}$$

In particular, with jobs of τ_i restricted to execute on at most P_i processors at any time, if $u_i > P_i$, jobs execute for their full WCET, and τ_i releases jobs as early as possible, then the response time of τ_i will grow without bound.

3.2 Trading Off History and Response Time

Recall from Section 2.4.1 that the response time of a job is the length of the interval between its release and finish times. The response times of rp-sporadic tasks can be bounded using analysis provided by Amert *et al*. (2019).[3] In this section, we first give an overview of the closed-form response-time bound. Then, we explore the history-versus-response-time trade-off through an experimental evaluation.

3.2.1 Bounding Response Times for Rp-Sporadic Tasks

We facilitate our discussion of such bound computations with a running example, using the task set depicted in Figure 3.3 with task parameters given in Table 3.1.

Example 3.3. Consider the task set depicted in Figure 3.3 with task parameters given in Table 3.1. In order to apply the bounds provided by Amert *et al*. (2019), we must first verify that the feasibility conditions given by Equations (3.1) and (3.2) are satisfied.

For this example, assume that $m = 3$. Equation (3.1) is satisfied, as $U = 2.8$. Additionally, note that for all tasks, $P_i \geq u_i$, so Equation (3.2) is also satisfied. Furthermore, as $P_i \leq 2$, the precedence constraints are not violated. $\Diamond$

The closed-form response-time bound for rp-sporadic tasks relies on the summed utilizations and WCETs of the tasks with restricted parallelism. This required definition, as well as the bound, are copied below.

[3]The response-time analysis for rp-sporadic tasks was derived by a co-author, Sergey Voronov, and thus is not presented as a contribution here.

Table 3.1: Task parameters for the tasks in Figure 3.3, assuming a platform with $m = 3$ CPUs.

Task	Graph	C_i	T_i	P_i	u_i
τ_1	τ^1	4	10	m	0.4
τ_2	τ^1	12	10	2	1.2
τ_3	τ^1	2	10	m	0.2
τ_4	τ^2	1	5	m	0.2
τ_5	τ^2	4	5	1	0.8

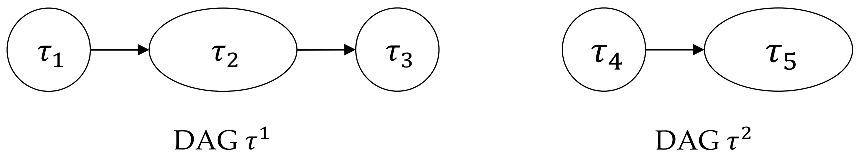


Figure 3.3: An rp-sporadic task set derived from two DAGs, containing two supernodes, τ_2 and τ_5, both with $p = 2$.

Definition 3.1. (Definition 5 in (Amert et al., 2019)) Call a task τ_i *p-restricted* (parallelism-restricted) if $P_i < m$, and *non-p-restricted* if $P_i \geq m$. Also, let

$$U^b_{res} = \sum_{\substack{\text{b largest values} \\ \tau_i \text{ is p-restricted}}} u_i \quad \text{and} \quad C^b_{res} = \sum_{\substack{\text{b largest values} \\ \tau_i \text{ is p-restricted}}} C_i,$$

and let $U_{res} = U^n_{res}$ and $C_{res} = C^n_{res}$.

Corollary 3.1. (Corollary 1 in (Amert et al., 2019)) *The response time of any task $\tau_i \in \tau$ is bounded by $x + T_i + C_i$, where*

$$x = \frac{(m-1)C_{max} + B_{max} + 2C_{res}}{m - U_{res}}. \tag{3.3}$$

Furthermore, if there exists $P_{min} \geq 1$ such that for every p-restricted task τ_i, $P_i \geq P_{min}$, then U_{res} and C_{res} in Equation (3.3) can be replaced with U^ℓ_{res} and C^ℓ_{res}, where $\ell = \lfloor (m-1)/P_{min} \rfloor$.

Corollary 3.1 gives a formula for calculating an upper bound on the response time of any task in the system. To bound the end-to-end response time of a graph, we take the maximum sum of the response times of each task along any path in the graph.

We assume global earliest-deadline first (G-EDF) scheduling on a system with m CPUs, and, as discussed in Section 3.1.2, we also assume that access to any accelerators (*e.g.*, a GPU) is non-preemptive and managed by locking protocols. Therefore, we let B_{max} be the duration of the longest non-preemptive accelerator access.

Example 3.3 (continued). Assume a maximum non-preemptive GPU access duration of $B_{max} = 2$ time units. For the system described in Table 3.1, $C_{max} = 12$, $C^{\ell}_{res} = 16$, $U^{\ell}_{res} = 2.0$. Corollary 3.1 gives a value for x of 58. Thus, the end-to-end response-time bounds of the graphs are $R^1 = 72 + 80 + 70 = 222$ and $R^2 = 64 + 67 = 131$ time units. $\Diamond$

To compare bounds between graphs with different periods, we instead will refer to the maximum *relative tardiness* of a graph τ^k. Recall that tardiness is defined as the amount by which a job misses its deadline: $\delta_{i,j} = \max\{0, f_{i,j} - d_{i,j}\}$, where the absolute deadline $d_{i,j}$ is assumed to be implicit (*i.e.*, $d_{i,j} = r_{i,j} + T_i$). Note that tardiness cannot be negative. Given a response-time bound R^k for a graph, we can thus compute the maximum relative tardiness using $\left(R^k - T^k\right)/T^k$ (recall that every node in a graph shares the same period).

Example 3.3 (continued). Graph τ^1 has a period of 10 time units and a response-time bound of $R^1 = 222$ time units. Thus, its maximum relative tardiness is 21.2. Similarly, graph τ^2 has a period of 5 time units and a response-time bound of $R^2 = 131$ time units, so its maximum relative tardiness is 25.2. $\Diamond$

Although the per-task parallelism P_i is constrained from below by the utilization of a supernode, it can be as high as p, albeit with a potential loss of algorithmic accuracy. Increasing P_i can increase P_{min} and thus decrease ℓ, leading to drastic reductions in analytical response-time bounds.

Example 3.3 (continued). Assume that $p = 2$ for both supernodes. If we instead set $P_5 = 2$, ℓ decreases to 1, reducing C^{ℓ}_{res} and U^{ℓ}_{res} to 12 and 1.2, respectively. As a result, $x = 27.8$, so the graphs' response-time bounds are reduced to $R^1 = 131.3$ and $R^2 = 70.6$ time units, respectively. Therefore, the maximum relative tardiness bounds are reduced to 12.1 and 13.1, respectively, a reduction of over 42%. $\Diamond$

This example demonstrates that allowing older history to be used can reduce the analytical bounds on tardiness (and thus response times) for a task system. Next, we provide an in-depth evaluation of this history-versus-response-time trade-off.

3.2.2 Evaluating the History-versus-Response-Time Trade-Off

Example 3.3 in Section 3.2.1 provides intuition for our study of the history-versus-response-time trade-off. We now explore the impact of changing P_{min} on the maximum end-to-end relative tardiness of a graph-based task system.[4]

[4]Source code is available at `https://cs.unc.edu/~anderson/diss/tanyadiss/`.

3.2.2.1 Experimental Setup

We consider a platform with $m = 16$ CPUs. Recall from the response-time bound calculation in Corollary 3.1 that B_{max} is independent of P_{min}. Thus, we did not consider GPU usage in our experiments (so $B_{max} = 0$). Instead, we focused on the effects of changing U_{res} and C_{res} by increasing P_{min} past the minimum possible value given by the per-task feasibility condition in Equation (3.2).

In this study, we generated $200,000$ graph-based task systems, with system utilizations in the range $[2.5, 16]$. To generate a task system, we first chose a target system utilization value, and selected per-task utilizations from a given distribution (described below) until an additional task would cause total utilization to exceed m. Then, tasks were randomly assigned to graphs such that four to eight tasks were assigned to each graph (once fewer than four tasks remained, the rest were assigned to a final graph). Finally, edges were selected with probability 0.3, according to the Erdős-Rényi graph generation model (Erdős and Rényi, 1958), which is commonly used to generate random graphs. Note that as each of the possible edges in the graph was independently added with probability 0.3, we do not require a graph to be connected.

We selected per-task utilizations from two distributions: *uniform* and *exponential*. For the uniform distribution, the utilization of each task was independently chosen uniformly from $[0, 1.5)$. Thus, approximately one-third of tasks generated using this distribution required $P_i \geq 2$. When using the exponential distribution, each task had utilization chosen independently from an exponential distribution with mean 0.6. With both distributions, any task with $u_i \leq 1$ was assumed to have $P_i = m$; otherwise, P_i was set to $\lceil u_i \rceil$. Periods were chosen per-graph uniformly from $[10, 100]$ time units.

3.2.2.2 Results

After computing the maximum relative tardiness for each task system, we grouped the results into system-utilization buckets, with bucket sizes of 1.0 for low system utilizations to 0.5 for high system utilizations, and report the average of the maximum relative tardiness values for each utilization bucket. The results of our tardiness study are shown in Figure 3.4 and Figure 3.5 for uniform and exponential per-task utilizations, respectively. Each figure contains three curves, one for each of three values of P_{min}. In this context, P_{min} corresponds to the minimum P_i of any task in the system. To ensure that $P_{min} = 2$ was valid for each task system, we re-generated the per-task utilizations (as described in Section 3.2.2.1) if no task had $P_i = 2$.

Observation 3.1. Increasing P_{min} decreases relative tardiness.

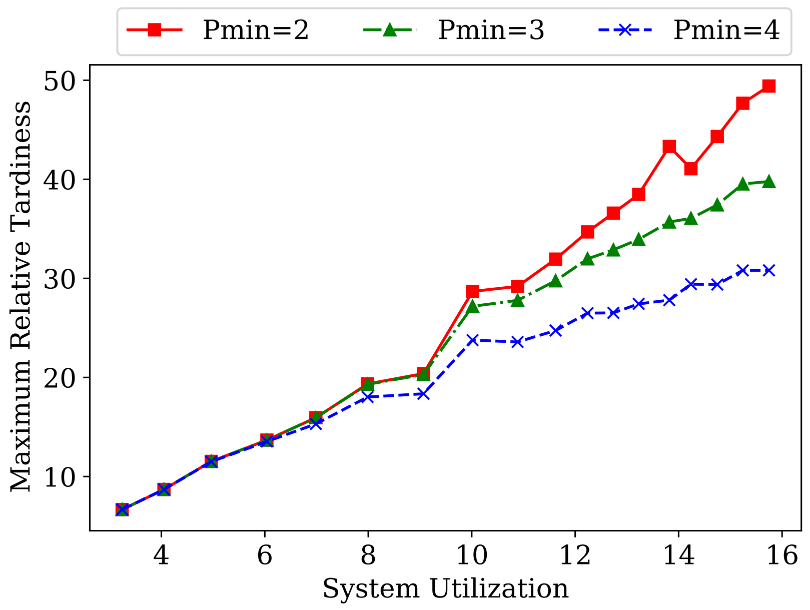


Figure 3.4: Maximum relative tardiness results for uniform per-task utilizations.

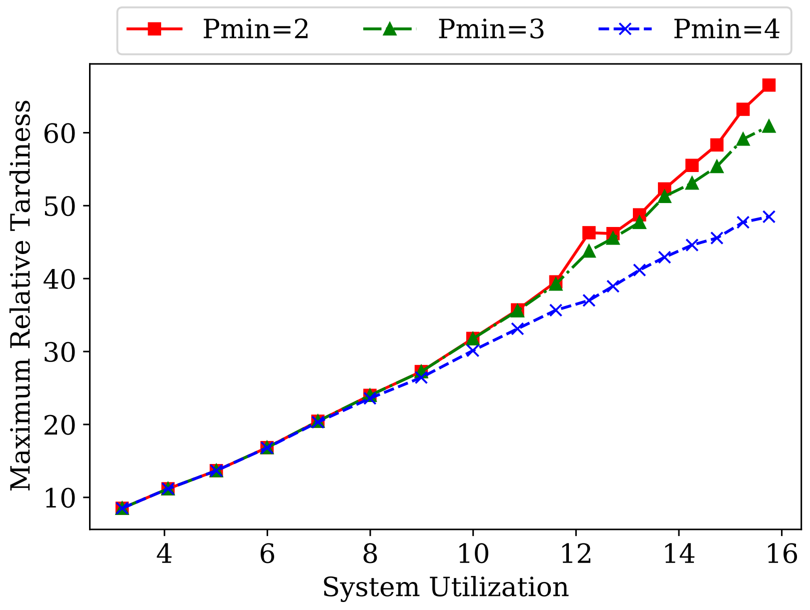


Figure 3.5: Maximum relative tardiness results for exponential per-task utilizations.

Increasing P_{min} corresponds to increasing the value of P_i for a subset of supernodes in the system, thus allowing more parallelism through the use of (possibly) older history. Given Corollary 3.1, this should decrease the relative tardiness for the task system. This intuition is borne out in both figures. Allowing $P_{min} = 3$ instead of $P_{min} = 2$ led to a 19.5% reduction in relative tardiness for uniform tasks, and a 8.5% reduction for exponential tasks. Allowing instead $P_{min} = 4$ greatly increased the reduction in relative tardiness, by 37.6% and 27.1% for uniform and exponential tasks, respectively.

These results demonstrate the significant analytical improvement that can be gained by allowing the use of older historical information. However, this gain comes at a price; the accuracy side of the history-versus-response-time-versus-accuracy trade-off will be explored next.

3.3 Trading Off History and Accuracy

We now describe the experiments we performed to evaluate the trade-off between history and accuracy. After describing our evaluation metrics and experimental setup, we discuss our experimental results.[5]

3.3.1 Evaluation Metrics for MOT

In this section, we provide an overview of the standard metrics used to evaluate MOT applications (Stiefelhagen et al., 2006). We consider first the metrics defined for each frame, then the metrics for each trajectory, and finally the high-level MOT metrics.

Per-frame metrics. A track represents not only where a target is believed to have been, but also a prediction of where it will be in future frames. Thus, a track for which the prediction is matched to a detection is considered a *true positive*, and we let TP_t be the number of such matches for frame t. A *false positive* represents a prediction that is not matched to any detected target, and a *false negative* corresponds to an unmatched detection. These are represented by FP_t and FN_t, respectively. The ground-truth number of targets present in frame t is represented by GT_t.

The final per-frame metric is the number of ID switches. We use the stricter definition given by Milan *et al.* (2016): an *ID switch* ($IDSW_t$) occurs for frame t when a target is assigned a track that is different from its prior track assignment. In the best case, $TP_t = GT_t$ and $FN_t = FP_t = IDSW_t = 0$.

[5]Source code is available at `https://cs.unc.edu/~anderson/diss/tanyadiss/`.

Per-trajectory metrics. In addition to per-frame metrics, we evaluate the results of tracking for each ground-truth trajectory. Following the CLEAR 2006 evaluation (Stiefelhagen et al., 2006), we classify each target as *mostly tracked (MT)* if it is successfully tracked (*i.e.*, its detections are matched to track predictions) for at least 80% of the frames for which it is present, *mostly lost (ML)* if it is tracked successfully for at most 20% of the frames for which it is present, and *partially tracked (PT)* otherwise. We can also measure the continuity of each track by considering the track's *fragmentation count* (*FM*). A fragmentation occurs when a target's trajectory becomes untracked and later becomes tracked again.

With an ideal tracker, all targets should be mostly tracked, and thus we would expect to find $PT = ML = 0$. Additionally, we would also expect $FM = 0$.

Overall metrics. A number of overall metrics are standardly used in the CV literature to evaluate a tracker's performance holistically. The *Multiple Object Tracking Accuracy (MOTA)* metric combines information about the false positives, false negatives, and ID switches:

$$MOTA = 1 - \frac{\sum_t (FN_t + FP_t + IDSW_t)}{\sum_t GT_t}.$$

For an ideal tracker, $MOTA = 1.0$. However, if maintaining the correct ID of each tracked object is not needed for a given application, the *MOTA* might not appropriately represent the success of the tracker. Instead, *A-MOTA* can be used in this case, as it ignores ID switches:

$$A\text{-}MOTA = 1 - \frac{\sum_t (FN_t + FP_t)}{\sum_t GT_t}.$$

The other common overall metric for MOT applications is the *Multiple Object Tracking Precision (MOTP)*, given as a ratio of the distances between ground-truth object positions and predicted track positions and the number of matches made, summed over all objects and all frames:

$$MOTP = \frac{\sum_{t,o} d_{t,o}}{\sum_t c_t}.$$

In this expression, $d_{t,o}$ is the IoU of the bounding boxes of object o and its predicted track position at frame t, and c_t is the number of detection-track matches found at frame t. For an ideal tracking system, $MOTP = 1.0$ (*i.e.*, each detection-track match has perfect overlap).

3.3.2 Experimental Setup

We now give an overview of our traffic scenario selection, and then discuss how we varied the age of historical data provided to the MOT application.

3.3.2.1 Driving Scenarios in CARLA

We evaluated the impact of increasing p in a broad range of scenarios generated from CARLA. These scenarios need to be challenging driving situations that require highly accurate tracking. The *CARLA Challenge* provides scenarios that are selected from the NHTSA (National Highway Traffic Safety Administration) pre-crash typology (Najm et al., 2007), which provides scenarios that are identified as common pre-crash scenarios of all police-reported crashes. From these scenarios, we selected the list below, which heavily rely on tracking and prediction, as our focus. We modified each scenario by adding additional vehicles and pedestrians to make the tracking task more challenging (as we aim to track all visible vehicles and pedestrians). In the descriptions that follow, "ego-vehicle" refers to the vehicle that navigates the scenario.

- **Scenario 1:** *Obstacle avoidance with prior action.* As the ego-vehicle turns right at a red light, an unexpected obstacle (a cyclist) crosses into the road. The ego-vehicle must perform an emergency brake or an avoidance maneuver.

- **Scenario 2:** *Right turn at an intersection with crossing traffic.* The ego-vehicle must turn right on red at an intersection and in the presence of crossing traffic. In this scenario, the ego-vehicle must track all crossing vehicles, yielding to avoid collisions.

- **Scenario 3:** *Crossing traffic running a red light at an intersection.* As the ego-vehicle enters an intersection going straight, a vehicle runs a red light from the right. In this scenario, the ego-vehicle must perform a collision avoidance maneuver.

- **Scenario 4:** *Unprotected left turn at an intersection with oncoming traffic.* The ego-vehicle must perform an unprotected left turn at an intersection, yielding to oncoming traffic.

- **Scenario 5:** *Lane changing to pass a slow-moving leading vehicle.* While driving on the highway, the vehicle in front of the ego-vehicle rapidly decelerates. The ego-vehicle must change lanes to avoid a collision, yielding to traffic in the next lane.

The five scenarios are depicted in Figure 3.6. Scenarios 1-4 feature city driving; Scenario 5 involves highway speeds. In our experiments, each scene is populated with additional vehicles and pedestrians that obey all traffic laws (*e.g.*, additional pedestrians do not enter the road, additional vehicles and cyclists obey stop lights and lane markings). Scenario 1 features three vehicles and twelve pedestrians. Scenarios 2-4 have six vehicles, and Scenarios 2 and 4 also have four pedestrians. Scenario 5 contains eleven vehicles. For each scenario, the start position and path of the ego-vehicle are indicated by a yellow star and solid line, respectively. The cyclist/vehicle the ego-vehicle must avoid are indicated with a pink dashed path and appropriate icon; additional vehicles and pedestrians are not shown.

3.3.2.2 Varying the Age of History

For our MOT experiments, we implemented the tracking-by-detection pipeline in Figure 3.7. This graph contains a cycle comprised of tasks τ_1, τ_3, and τ_4. The prior-history requirement p for the back edge from τ_4 to τ_1 is what we seek to vary; increasing p means that the track prediction step (τ_1) may use less-recently updated tracks to make predictions.

By definition, p is the *maximum* difference in time steps between the completion of an invocation of τ_4 and when those results are used by τ_1. However, more recent results can be used, if available. In our evaluation, we represent the distribution of available prior results using a *probability mass function* (PMF), which we represent as a tuple.

To measure the impact of varying p in our experiments, we executed the code sequentially, and for each invocation of τ_1, we chose the prior history to use based on a random number sampled from the PMF.[6] For example, for a PMF represented as $(0.8, 0.2)$, we selected one frame prior with probability 0.8, and two frames prior with probability 0.2.

We evaluated eight PMFs, listed in Table 3.2, chosen to answer four questions:

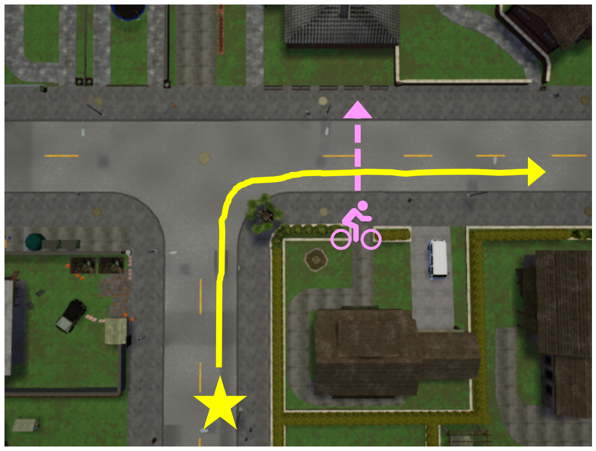
Q1 What if the most recent data are sometimes unavailable?

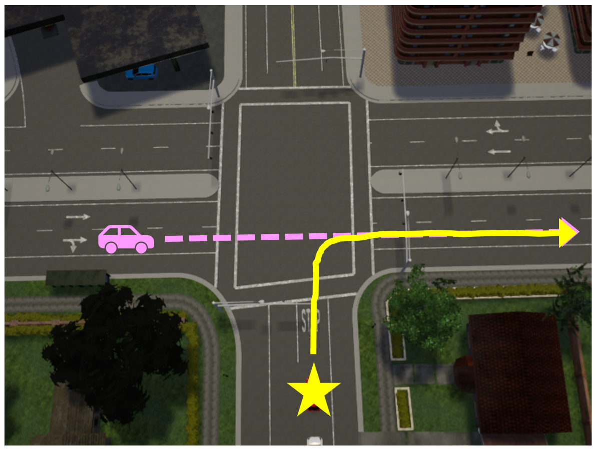
Q2 How much of an impact does the worst-case age have if the most recent data are usually available?

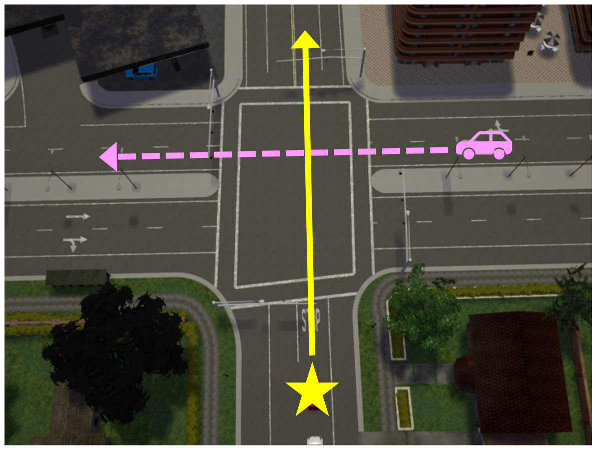
Q3 Is it better to have a higher chance of more recent data, or a lower worst-case age?

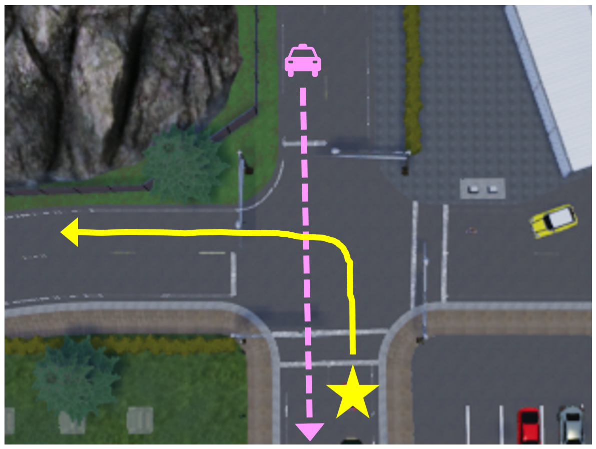
Q4 How does the average case differ from the worst case?

[6]The availability of prior results is unlikely to be a random variable, as we have assumed here. Rather, if the result of one frame is delayed, the next frame is likely to be subject to similar delaying effects; we leave such explorations to future work.


(a) Scenario 1: Obstacle avoidance with prior action.


(b) Scenario 2: Right turn at an intersection with crossing traffic.


(c) Scenario 3: Crossing traffic running a red light at an intersection.


(d) Scenario 4: Unprotected left turn at an intersection with oncoming traffic.

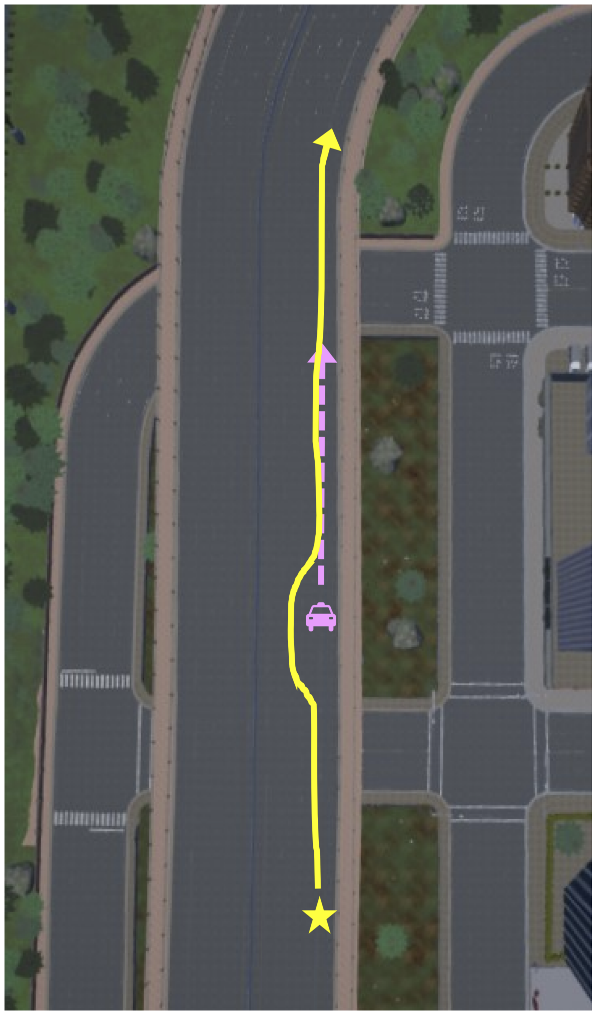

(e) Scenario 5: Lane changing to pass a slow-moving leading vehicle.

Figure 3.6: The five scenarios we considered.

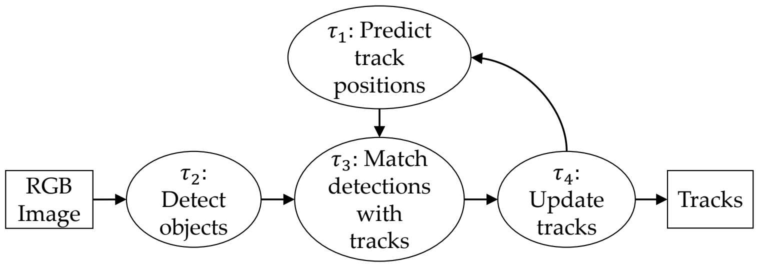

τ_1: Predict track positions
RGB Image
τ_2: Detect objects
τ_3: Match detections with tracks
τ_4: Update tracks
Tracks

Figure 3.7: The tracking-by-detection pipeline.

Table 3.2: Probability mass functions (PMFs) corresponding to available history results.

	PMF (represented as a tuple)
PMF 1	$(0.9, 0.09, 0.009, 0.001)$
PMF 2	$(0.8, 0.2)$
PMF 3	$(0.8, 0.02, 0.02, 0.16)$
PMF 4	$(0.5, 0.4, 0.1)$
PMF H_p	$\left(0, \overset{p-1}{\ldots}, 0, 1\right)$

The PMFs are described as tuples: (x, y) indicates that the result of one and two frames prior are available with probabilities x and y, respectively. In *PMF* H_p, $0, \overset{p-1}{\ldots}, 0$ denotes a sequence of $p-1$ 0's, where p ranges over $(1, \ldots, 4)$. Note that the values of each PMF sum to 1.0.

Comparing *PMFs* 1 and H_1 should answer question Q1. We can answer question Q2 by comparing *PMFs* 2 and 3. For question Q3 we compare *PMFs* 3 and 4. Finally, we compare *PMFs* 1-4 to the corresponding worst-case *PMF* H_p (*e.g.*, *PMFs* 4 and H_3) to answer question Q4.

The answers to these four questions will give insight into the history-versus-accuracy trade-off. We anticipate that *PMF* H_1 will perform the best, but that *PMFs* 1 and 2 may have similarly high accuracy to *PMF* H_1. If this is the case, then we can conclude that allowing a small reduction in accuracy is acceptable if it enables response-time bounds to be computed (as *PMFs* 1 and 2 enable intra-task parallelism of $P_i = 4$ and $P_i = 2$, respectively) for otherwise-unschedulable tasks.

3.3.3 Perfect Sensing

We now describe the results of tracking both vehicles and pedestrians in the scenarios listed in Section 3.3.2.1, sampling from the PMFs in Table 3.2. We first consider tracking in isolation, *i.e.*, in the presence of perfect sensors. For each frame of the video, we provide the ground-truth 3D motion of the camera (representing perfect ego-motion estimation), ground-truth 2D bounding boxes (as if from a perfect detector), and the 3D distance to each target within the scene (corresponding to perfect stereo estimation). In Section 3.3.4, we remove the assumption of a perfect detector.

Each tracking scenario lasted 300 camera frames. We evaluated tracking accuracy and precision, comparing the results for each PMF to those of *PMF* H_1. For each scenario, we controlled the vehicle manually and collected RGB images generated by a single front-facing camera, as well as ground-truth bounding boxes of all vehicles and pedestrians that were captured by the camera and the ground-truth motion of the camera itself.

The *A-MOTA*, *MOTP*, and average *FM* count for each scenario and PMF are reported in Table 3.3 for vehicle tracking and Table 3.4 for pedestrian tracking (note that Scenarios 3 and 5 had no pedestrians). For *PMFs* 1-4, we repeated the scenario 100 times, and report the average *A-MOTA*, *MOTP*, and *FM* count results. Given these data, we can now answer the four questions posed in Section 3.3.2.2.

Q1: What if the most recent data are sometimes unavailable? To explore this question, we compare the results of *PMFs* 1 and H_1. As *PMF* 1 has the most recent data available with probability 0.9, we expect that the accuracy and precision will be comparable to *PMF* H_1, for which the most recent data are always available.

Comparing the two columns in Tables 3.3 and 3.4, we see that *PMF* 1 has an *A-MOTA* score within 0.62% and 0.90% of that of *PMF* H_1 across all scenarios for vehicles and pedestrians, respectively. Similarly, the *MOTP* score for *PMF* 1 is within 2.07% and 1.61% of that of *PMF* H_1 for vehicles and pedestrians, respectively. The average *FM* count is only slightly larger for *PMF* 1 in all scenarios as well.

Q2: How much of an impact does the worst-case age have if the most recent data are usually available? This question can be answered by comparing *PMFs* 2 and 3. For both distributions, the most recent data are available with probability 0.8. However, *PMF* 3 represents a bimodal distribution, which may result if tasks become greatly delayed; its worst-case age is four frames, which occurs with probability 0.16. For *PMF* 2, the worst-case data age is only two frames.

For pedestrian tracking, *PMF* 3 was within at most 1.14% of *PMF* 2 in terms of *A-MOTA* across all scenarios. For vehicle tracking, this difference dropped to 0.73%. *PMF* 3 had *MOTP* within 4.95% that of *PMF* 2 across all scenarios and both target types. Additionally, *PMF* 3 resulted in up to 2.5 times as many track fragmentations than *PMF* 2. These results suggest that although it is better to have a lower worst-case

Table 3.3: Results for vehicle tracking using ground-truth detections.

	Metric	*PMF* 1	*PMF* 2	*PMF* 3	*PMF* 4	*PMF* H_1	*PMF* H_2	*PMF* H_3	*PMF* H_4
Scenario 1	A-MOTA	**0.951**	**0.952**	**0.947**	0.939	**0.951**	0.927	0.893	0.888
	MOTP	0.709	0.700	0.669	0.670	**0.724**	0.644	0.588	0.567
	Avg. FM	1.117	1.160	1.503	1.350	**1.000**	1.667	2.000	2.667
Scenario 2	A-MOTA	**0.966**	0.962	0.955	0.951	**0.972**	0.933	0.943	0.933
	MOTP	**0.730**	0.724	0.699	0.697	**0.736**	0.669	0.609	0.560
	Avg. FM	0.590	0.688	1.077	1.070	**0.500**	1.500	1.833	1.667
Scenario 3	A-MOTA	**0.982**	**0.980**	**0.977**	0.974	**0.985**	0.973	0.954	0.931
	MOTP	0.721	0.714	0.689	0.686	**0.730**	0.656	0.628	0.580
	Avg. FM	0.553	0.642	1.110	1.058	**0.500**	1.333	1.833	2.000
Scenario 4	A-MOTA	**0.965**	**0.962**	0.957	0.952	**0.968**	0.939	0.934	0.908
	MOTP	**0.777**	0.772	0.751	0.751	**0.784**	0.741	0.690	0.650
	Avg. FM	1.583	1.622	1.985	1.948	**1.500**	2.500	2.500	2.333
Scenario 5	A-MOTA	**0.978**	**0.976**	0.970	0.968	**0.982**	0.957	0.942	0.904
	MOTP	0.652	0.646	0.614	0.607	**0.659**	0.575	0.511	0.483
	Avg. FM	0.695	0.785	1.140	1.122	**0.636**	1.636	2.455	2.455

The best result in each row, as well as any within 1% of the best, are shown in bold.

age, the differences in both accuracy and precision are not extreme. However, higher worst-case age does contribute to more track fragmentations.

Q3: Is it better to have a higher chance of more recent data, or a lower worst-case age? *PMFs* 3 and 4 were chosen to answer this question: *PMF* 3 has more likely availability of the most recent results (probability 0.8 rather than 0.5), but has greater worst-case age (four frames versus three frames).

The results for *PMFs* 3 and 4 indicate that recency of available data is slightly more important than the worst-case data age. *PMF* 3 outperformed *PMF* 4 for *A-MOTA* and *MOTP* in seven out of eight comparisons each. However, the difference in these scores was only up to 0.84% for *A-MOTA* and 1.89% for *MOTP*. On the other hand, *PMF* 3 resulted in higher average *FM* count in six out of eight comparisons, although in the worst case *PMF* 3 only had 10.18% more track fragmentations. Therefore, although *PMF* 3 seems to perform slightly better, our experiments have not demonstrated a clear answer to this question.

Q4: How does the average case differ from the worst case? For the average and worst cases, we compare *PMFs* 1-4 to the corresponding worst-case *PMF* H_ps. We expect the average case to result in higher *A-MOTA* and *MOTP* scores and lower average *FM* counts, and for this difference to become more pronounced as the worst-case history age increases.

Table 3.4: Results for pedestrian tracking using ground-truth detections.

	Metric	*PMF* 1	*PMF* 2	*PMF* 3	*PMF* 4	*PMF* H_1	*PMF* H_2	*PMF* H_3	*PMF* H_4
Scenario 1	A-MOTA	**0.884**	0.878	0.868	0.863	**0.892**	0.847	0.842	0.825
	MOTP	0.672	0.663	0.634	0.622	**0.683**	0.601	0.569	0.541
	Avg. FM	0.853	0.888	1.324	1.237	**0.667**	1.917	1.583	1.667
Scenario 2	A-MOTA	**0.968**	**0.969**	0.959	**0.962**	**0.971**	**0.962**	0.956	0.936
	MOTP	**0.808**	0.803	0.785	0.783	**0.814**	0.767	0.731	0.702
	Avg. FM	0.225	0.220	0.568	0.590	**0.000**	1.000	1.000	1.000
Scenario 4	A-MOTA	**0.936**	**0.934**	0.928	0.927	**0.938**	0.920	0.856	0.840
	MOTP	**0.794**	0.788	0.767	0.767	**0.800**	0.748	0.736	0.720
	Avg. FM	0.135	0.178	0.445	0.525	**0.000**	1.000	1.000	1.000

The best result in each row, as well as any within 1% of the best, are shown in bold.

For a worst-case history age of two, we compare *PMF* 2 to *PMF* H_2: *PMF* 2 performed better than *PMF* H_2 in every comparison. Similarly, for worst-case history age of three frames, *PMF* 4 outperformed *PMF* H_3 in every comparison. As expected, for a worst-case history age of four frames, both *PMF* 1 and *PMF* 3 beat the worst-case *PMF* H_4 in every comparison, and for most by a large margin.

The history-versus-accuracy trade-off. The experimental results relating to question Q4 hint at our overall conclusion: allowing the infrequent use of older results in a MOT application has only minimal impact on the application's accuracy and precision, while allowing the computation of bounded response times for use in real-time certification. To explore this a little further, we make a final comparison against *PMF* H_2, which always uses the results of two frames prior, and thus corresponds to tracking using only half of the frames.

PMFs 1-4 are indexed in order of the expected results. That is, prior to performing experiments, we expected *PMF* 1 to perform the best and *PMF* 4 to perform the worst of this group. In fact, comparing these four PMFs to *PMF* H_2, we see that *PMF* H_2 did not perform as well as *PMF* 1 or *PMF* 2 in any comparison. Additionally, *PMF* H_2 scored better than *PMF* 3 or *PMF* 4 in only one of the 24 comparisons.

Although results hold for both city- and highway-driving scenarios, the motion of the ego-vehicle does factor into the trade-off. Scenarios 1-4 feature city driving (of these, Scenario 3 involves the highest speed, as it does not involve the ego-vehicle turning sharply), and Scenario 5 occurs at highway speeds with only minor direction changes. As shown in Table 3.3, direction changes were inversely correlated with *A-MOTA* scores: Scenarios 3 and 5 had much higher *A-MOTA* values for *PMFs* 1-4 and *PMF* H_1 than the other scenarios. For *MOTP*, the speed seemed to be the largest factor: Scenario 5 had much lower *MOTP* than all other scenarios.

3.3.4 Camera-Based Sensing

We have thus far examined the history-versus-accuracy trade-off in the presence of perfect detections, *i.e.* using the ground-truth bounding boxes of pedestrians and vehicles. However, in real-world scenarios, such ground-truth data are not available, which necessitates the use of CV-based object-detection algorithms.

We chose for a detector a state-of-the-art deep-learning model, Faster R-CNN (Ren et al., 2015), which has been shown to achieve a high level of accuracy. We used TensorFlow (Abadi et al., 2016) to train a Faster R-CNN model with the Inception v2 feature extractor (Ioffe and Szegedy, 2015) (that was pre-trained on the COCO dataset (Huang et al., 2017; Lin et al., 2014)) on a small dataset of $1,300$ images of bicycles, motorbikes, cars, and pedestrians generated from CARLA (DanielHfnr, 2020; tkortz, 2020).

We measured the detection accuracy of our model using a popular object-detection accuracy metric, the *mean average precision (mAP)* (Everingham et al., 2010; Hui, 2020). After over $42,000$ training iterations, our Faster R-CNN model achieved a mAP score of 92.89%, indicating a high level of detection accuracy; a perfect object-detection algorithm would have a mAP score of 100%.

We now examine the impact of imperfect detections on tracking accuracy and precision. For this second set of accuracy experiments, we replaced the ground-truth vehicle and pedestrian detections with those generated by the Faster R-CNN model; the remainder of the experimental setup was unchanged. The resulting *A-MOTA* and *MOTP* values are given in Tables 3.5 and 3.6. Note that computing *FM* counts requires mapping the ground-truth trajectories to detections in order to determine whether a target's trajectory becomes untracked and later re-tracked; when using detections produced by the Faster R-CNN model, we did not have such a ground-truth-based mapping, so we do not include *FM* counts in Tables 3.5 and 3.6.

The impact of imperfect detections. We first compare the results using ground-truth data (Tables 3.3 and 3.4) with those from imperfect detections (Tables 3.5 and 3.6). As we would expect, using a detector decreased the accuracy of vehicle tracking and the precision of pedestrian tracking. Furthermore, *PMF* H_1 had the highest *MOTP* in each scenario for both vehicles and pedestrians. However, accuracy seems to be more affected by potentially incorrect detections. For example, *PMF* 4 resulted in the highest *A-MOTA* score for vehicles in Scenario 1, despite using older data with probability 0.5, and *PMF* H_4, which corresponds to always using data from four frames prior) had the highest *A-MOTA* score for pedestrians in Scenario 2. Altogether, our results suggest that tracking of pedestrians is less impacted by imperfect data, whereas for vehicles the impact can be quite large.

Table 3.5: Results for vehicle tracking using Faster R-CNN to detect vehicles.

	Metric	*PMF* 1	*PMF* 2	*PMF* 3	*PMF* 4	*PMF* H_1	*PMF* H_2	*PMF* H_3	*PMF* H_4
Scenario 1	A-MOTA	0.741	0.785	0.810	**0.844**	0.761	0.727	0.698	0.659
	MOTP	**0.727**	**0.721**	0.664	0.696	**0.728**	0.642	0.584	0.561
Scenario 2	A-MOTA	0.833	**0.844**	0.810	0.815	**0.836**	**0.836**	**0.837**	0.833
	MOTP	**0.704**	0.689	0.673	0.649	**0.705**	0.631	0.593	0.547
Scenario 3	A-MOTA	**0.896**	0.874	0.883	0.877	**0.904**	0.858	0.850	0.836
	MOTP	0.601	0.586	0.556	0.573	**0.609**	0.532	0.506	0.496
Scenario 4	A-MOTA	**0.828**	0.818	0.793	0.802	**0.831**	0.817	0.810	0.789
	MOTP	**0.766**	0.757	0.711	0.712	**0.766**	0.712	0.688	0.656
Scenario 5	A-MOTA	**0.968**	**0.968**	**0.964**	**0.964**	**0.970**	0.960	**0.961**	0.949
	MOTP	0.747	0.728	0.703	0.703	**0.758**	0.659	0.579	0.536

The best result in each row, as well as any within 1% of the best, are shown in bold.

Table 3.6: Results for pedestrian tracking using Faster R-CNN to detect pedestrians.

	Metric	*PMF* 1	*PMF* 2	*PMF* 3	*PMF* 4	*PMF* H_1	*PMF* H_2	*PMF* H_3	*PMF* H_4
Scenario 1	A-MOTA	0.882	**0.888**	**0.894**	0.873	**0.890**	0.847	0.834	0.835
	MOTP	0.639	0.640	0.600	0.612	**0.654**	0.622	0.535	0.449
Scenario 2	A-MOTA	0.697	0.688	**0.717**	0.697	0.703	0.641	0.706	**0.723**
	MOTP	0.729	**0.745**	0.690	0.734	**0.751**	0.707	0.681	0.663
Scenario 4	A-MOTA	**0.938**	**0.938**	**0.934**	**0.938**	**0.938**	**0.938**	**0.936**	0.920
	MOTP	**0.763**	0.745	0.723	0.727	**0.769**	0.721	0.694	0.663

The best result in each row, as well as any within 1% of the best, are shown in bold.

Q1 through Q4, revisited. For question Q1, we again compare *PMF* 1 and *PMF* H_1. The *MOTP* score for *PMF* 1 was at most 2.93% lower than that of *PMF* H_1, indicating that there is still a precision loss due to having older data. Similarly for *A-MOTA*, the score for *PMF* 1 was at most 2.63% lower than that of *PMF* H_1.

In comparing *PMF* 2 and *PMF* 3, we see that having a higher worst-case history age (*PMF* 3) corresponded to much lower *MOTP* in all scenarios: *MOTP* was decreased by up to 7.91% compared to using *PMF* 2. However, *A-MOTA* was much more varied in the presence of a CV-based vehicle detector. In four of the eight comparisons, *PMF* 3 resulted in *higher* accuracy than *PMF* 2.

For the comparison for questions Q3, we again see different trends for precision relative to accuracy. In four of the eight comparisons, *PMF* 3 corresponded to lower accuracy than *PMF* 4; for *MOTP*, this number was six out of eight. Combined with the comparison for question Q2, this suggests that MOT precision relies on having lower worst-case history age, but MOT accuracy does not have such a clear dependence.

Finally, we revisit the comparisons related to question Q4. As with ground-truth detections, *PMFs* 1 and 3 usually outperformed *PMF* H_4, both by a large margin; for Scenario 2, however, this trend was surprisingly reversed. The differences for the other comparisons were less pronounced with CV-based detections than ground-truth detections: *PMF* 4 usually outperformed *PMF* H_3 in *A-MOTA* but had much higher *MOTP*, and the same trend held for *PMF* 2 and *PMF* H_2.

The trade-off, revisited. Compared to always having the most recent data (*i.e.*, *PMF* H_1), the lowest *A-MOTA* score among *PMFs* 1-4 was reduced by 4.57%, and the lowest *MOTP* score was reduced by 8.79%. This suggests only a minor drop in accuracy, with a moderate drop in precision, as a result of using older data. It is also worth restating that the introduction of a real detector into the MOT pipeline makes these results less definitive, however: in some scenarios, using older data actually resulted in higher precision or accuracy.

However, if requiring $p = 1$ results in a system for which no response-time bounds can be computed, then measuring accuracy relative to *PMF* H_1 no longer has any relevance. Instead, we compare against *PMF* H_2, to compare to a system in which the history age of two is always used. In this case, both precision and accuracy are somewhat robust to a potentially imperfect detector. This is evident when comparing *PMF* H_2 to *PMFs* 1-2, which correspond to situations in which the most recent data are available with probability at least 0.8 and the the worst-case age is two with probability at least 0.99. Aside from a single comparison in which *PMF* H_2 has slightly higher accuracy, when using our Faster R-CNN model to detect vehicles and pedestrians, *PMFs* 1 and 2 had 0.12-8.74% higher accuracy than that of *PMF* H_2 and 2.73-13.35% higher tracking precision. These results suggest that for a system that would otherwise be unschedulable, the increased worst-case history requirement due to increased parallelism only slightly reduces MOT accuracy and precision, and that reduction can be greatly mitigated if the most recent results are usually available.

3.4 Chapter Summary

In this chapter, we have shown that a new real-time task model can enable scheduling of cyclic real-time processing graphs if $p > 1$ is allowed for any cycle with utilization exceeding 1.0. Such graphs are crucial to consider in real-time certification processes applicable to autonomous vehicles due to the prevalence of use cases where historical information must be tracked.

Our new task model reveals interesting trade-offs pertaining to graph cycles that hinge on response times, allowed parallelism, and CV accuracy. We explored the history-versus-response-time trade-off through synthetic experiments, and demonstrated that relative tardiness can be reduced by as much as 37.6% by increasing the minimum intra-task parallelism as high as $P_{min} = 4$. However, $P_{min} > 1$ permits non-immediate back history to be used, which in the context of CV tracking applications could potentially compromise tracking accuracy. We have additionally studied this issue in detail using a tracking algorithm and data generated in the CARLA autonomous-driving simulator. In our experiments, allowing P_i to be as high as 4 resulted in at most a 2.63% drop in accuracy, as long as immediate back history is available approximately 90% of the time.

The reduction of response-time bounds enabled by the trade-off explored in this chapter can improve the safety of the resulting system by ensuring that ADAS-related tasks may complete execution more quickly. Although reducing the accuracy of the resulting computations may serve to decrease safety, it is worth noting that accuracy loss is mainly being contemplated here to ensure the schedulability of graphs that would otherwise be unschedulable. In our evaluations of this trade-off, we have not sought to explore how increasing parallelism directly impacts safety; we leave such an exploration to future work.

Acknowledgements. The initial work on the rp-sporadic task model presented in Section 3.1 and the bounds listed in Corollary 3.1 were the result of a collaboration between Tanya Amert and Sergey Voronov (Amert et al., 2019). Voronov derived the bounds and provided an initial implementation of the graph-generation scripts for the experiments to evaluate the history-versus-response-time trade-off, and Amert tailored the scripts to perform the experiments depicted in Figures 3.4 and 3.5.

The history-versus-accuracy evaluation discussed in Section 3.3 was a collaboration among Tanya Amert, Ming Yang, and Saujas Nandi (Amert et al., 2020). Amert guided the overall evaluation and implemented the tracking-by-detection pipeline. Yang performed the initial CARLA scenario selection and generated the preliminary data for Scenarios 1–4. Nandi performed the initial training of the Faster R-CNN model for pedestrian detection. Amert added Scenario 5, collected a new vehicle-detection dataset from CARLA, and re-trained the model to detect both vehicles and pedestrians.

CHAPTER 4: USING NVIDIA GPUS IN REAL-TIME APPLICATIONS[1]

In this chapter, we explore the challenges that arise when using NVIDIA GPUs in real-time applications. The first such challenge is centered around the scheduling policies that determine the ordering of work on a GPU. Despite the extensive documentation provided by NVIDIA, important details are omitted, such as what conditions permit multiple computations to execute concurrently on a GPU. Through extensive experimentation, we were able to discern many of these details.

Unfortunately, complexities arise in even the simplest GPU-using applications. Our experiments revealed a series of pitfalls, many relating to unexpected sources of synchronization on the GPU, that may entrap even the most seasoned GPU programmer. We designed CUPiD$^{\text{RT}}$ (CUDA Pitfall Detector for Real-Time Systems), a software tool to detect the most problematic of these pitfalls, and illustrated the benefits of fixing the detected issues through a case-study evaluation.

Organization. The remainder of this chapter is organized as follows. In Section 4.1, we overview what details are already known about scheduling on NVIDIA GPUs, and what open questions remain. We answer these questions in Sections 4.2 and 4.3 by providing a series of scheduling rules that we validated experimentally. In Section 4.4, we detail the possible pitfalls when using NVIDIA GPUs in real-time applications. Then we describe CUPiD$^{\text{RT}}$ and present an experimental evaluation of its use in Sections 4.5 and 4.6, respectively. Finally, we conclude our discussion of GPU use in real-time applications in Section 4.7.

[1]Contents of this chapter previously appeared in preliminary form in the following papers:

Amert, T., Otterness, N., Yang, M., Anderson, J. H., and Smith, F. D. (2017). GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 104–115.

Yang, M., Otterness, N., Amert, T., Bakita, J., Anderson, J. H., and Smith, F. D. (2018b). Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, pages 20:1–20:21.

Amert, T. and Anderson, J. H. (2021). CUPiD$^{\text{RT}}$: Detecting improper GPU usage in real-time applications. In *Proceedings of the 24th International Symposium on Real-Time Distributed Computing*, pages 86–95.

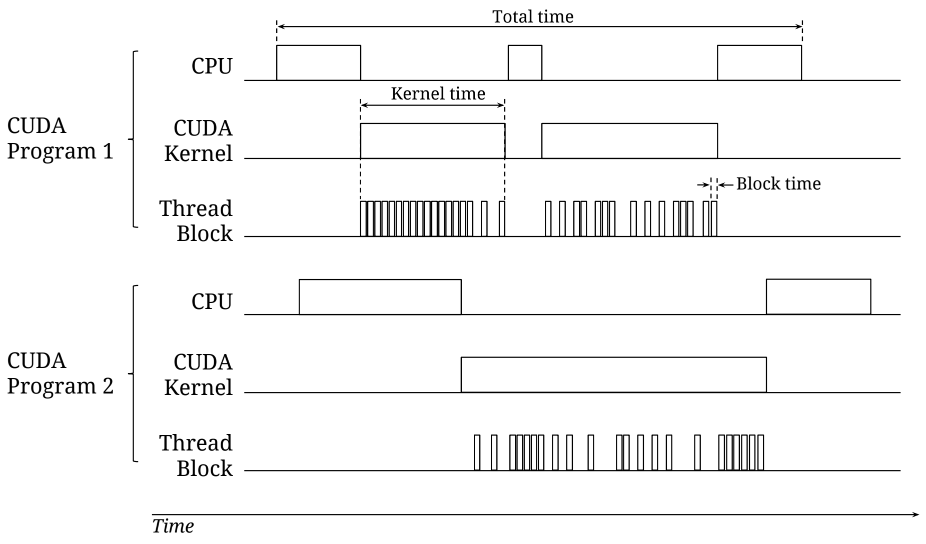


Figure 4.1: The relation between CUDA programs, kernels, and thread blocks.

4.1 Documented Ordering and Execution of NVIDIA GPU Operations

As discussed in Section 2.3.2, a CUDA program consists of CPU code that invokes GPU code contained in CUDA kernels, each of which is executed by running a programmer-specified number of thread blocks on the GPU. This hierarchy is depicted in Figure 4.1 for two CUDA programs executed by processes running on different CPUs in a multicore platform.

Multiple thread blocks from the same CUDA kernel can execute concurrently if sufficient GPU resources exist, although this is not depicted in Figure 4.1. The figure distinguishes between *block time*, which is the time taken to execute a single thread block, *kernel time*, which is the time taken to execute a single kernel (from when its first block commences execution until its last block completes), and *total time*, which is the time taken to execute an entire CUDA program (including CPU portions). We refer to copy operations and kernels collectively as *GPU operations*. For simplicity, copy operations are not depicted in Figure 4.1.

In CUDA, GPU operations can be ordered by associating them with a *stream*. Operations in a given stream are executed in FIFO order, but the order of execution across different streams is determined by the GPU scheduling policy. Kernels from different streams may execute concurrently or even out of launch-time order. Kernel executions and copy operations from different streams can also operate concurrently depending on the GPU hardware. By default, GPU operations from all CPU programs are inserted into the single *default stream*. Unless specified otherwise, this is the *NULL stream* (NVIDIA, 2019b). Programmers can create and use additional streams to allow for concurrent operations.

Kernel launches are supposed to be asynchronous with respect to the CPU program:[2] when the CUDA kernel-launch call returns, the CPU can perform additional computation including issuing more GPU operations. By default, copy operations are synchronous with respect to a CPU program: they do not return until the copy is complete. Asynchronous copy operations are also available so the CPU program can continue to perform computations, kernel launches, or copies. Copy operations, whether synchronous or asynchronous, will not start until all prior kernel launches from the same stream have finished.

To our knowledge, complete details of kernel attributes and policies used by NVIDIA to schedule GPU operations are not available. The official CUDA documentation only states that operations within a stream are processed in FIFO order, and that kernels from different streams *may* run concurrently.[3] An NVIDIA developer presentation (Luitjens, 2014) gives slightly more information: the older Fermi architecture multiplexed CUDA streams into a single internal queue based on issue order, and in the Kepler architecture, this was expanded to include multiple internal queues.

Open questions. The NVIDIA documentation says that kernels submitted to different user-defined streams *may* execute concurrently, but it does not make clear what the ordering is between blocks of those kernels. Specifically, it is unclear whether each kernel must wait for the last block of the previous kernel to (a) finish execution or (b) begin execution, or whether (c) blocks of one kernel can "cut ahead" of blocks of other kernels, if space permits.

Example 4.1. Figure 4.2 depicts these potential scenarios. In this example, kernel K1 utilizes an entire SM, and one block each of kernel K2 and K3 together use all threads; assume that kernels are named in order of submission. The question lies in the timing of the block of K3 relative to the two blocks of K2.

In inset (a), blocks of one kernel must wait until all blocks of the previous kernel have completed. In inset (b), K3 must wait only until all blocks of K2 have begun execution, rather than finished. Both insets (a) and (b) depict non-work-conserving scheduling, as K3 is available but does not execute alongside the first block of K2. One could hope that the scenario in inset (c) matches actual behavior—that a block of K3 could "cut ahead" of the final block of K2, or else capacity is wasted as those threads are unused for the duration of

[2]The CUDA documentation uses a narrow definition of "asynchronous" that can be misleading; see Section 3.2.5.1 of the *Programming Guide* for CUDA version 10.2.89 (NVIDIA, 2019b). We explore the unexpected occurrences of CPU-blocking kernel-launch commands in Section 4.4.

[3]For example, this is how streams are described in Sections 9.1.2 and 10.2 of the *Best Practices Guide* for CUDA version 10.2.89 (NVIDIA, 2019a).

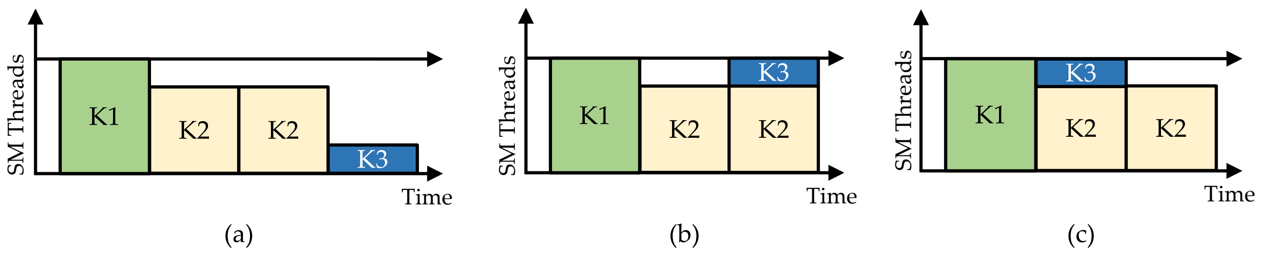


Figure 4.2: Potential GPU scheduling options.

a block of K2. However, as we shall see in Section 4.2, inset (b) depicts the actual behavior of the NVIDIA scheduler. ◊

Results such as these indicate that there are many possibilities for scheduling behavior, and the actual scheduling model may have a large impact on possible platform utilization. Questions to answer include how work is ordered between user-defined streams, the NULL stream, and prioritized streams. Additionally, these interactions must be explained for both kernel execution and copy operations.

4.2 Basic Scheduling Rules for NVIDIA GPUs

In this section, we present GPU scheduling rules for NVIDIA GPUs assuming the GPU is accessed only by CPU tasks that share an address space, using only user-specified streams. (We consider the NULL stream in Section 4.3. Unless stated otherwise, "stream" should henceforth be taken to mean a user-specified stream.) We inferred the scheduling rules given here by conducting extensive experiments using CUDA 10.2.89.[4] The experiments presented in this section were performed on an NVIDIA TX2; any discrepancies on other GPUs are discussed in Section 4.3.3.

We begin by discussing one of these experiments in Section 4.2.1. We use this experiment as a continuing example to explain various nuances of the rules, which are covered in full in Section 4.2.2. Before continuing, we note that all code, scripts, and visualization tools developed for these experiments are available online.[5]

4.2.1 Motivating Experiment

The experiment considered in this section involved running instances of a synthetic benchmark that launches several kernels. Our synthetic workload allows flexibility in configuring block resource requirements,

[4]These experiments were originally performed using CUDA 8.0.62, and have also been validated using CUDA 9.

[5]`https://cs.unc.edu/~anderson/diss/tanyadiss/`.

Table 4.1: Details of kernels used in the experiment in Figure 4.3.

Kernel	Launch Info	Start Time (s)	# Blocks	# Threads per Block	Shared Mem. per Block	Copy In	Copy Out
K1	CPU 0, Stream S1	0.0	6	768	0	-	-
K2	CPU 0, Stream S1	0.0	2	512	0	-	256MB
K3	CPU 0, Stream S1	0.0	2	1,024	0	256MB	256MB
K4	CPU 1, Stream S2	0.2	4	256	32KB	-	-
K5	CPU 1, Stream S3	0.4	2	256	32KB	-	256MB
K6	CPU 1, Stream S2	2.8	2	512	0	-	256MB

Note that "start times" are defined relative to the end of benchmark initialization.

kernel durations, and copy operations. We have also experimented with a variety of real-world GPU workloads involving image-processing functions common in autonomous-driving use cases, and to our knowledge, the scheduling rules presented in this chapter are valid for such workloads as well.

In the experiment considered here, we configured each block of each benchmark kernel to spin for one second. As detailed in Table 4.1, three kernels, K1, K2, and K3, were launched by a single task to a single stream, and three additional kernels, K4, K5, and K6, were launched by a second task to two separate streams. The kernels are numbered by launch time. Copy operations occurred in Stream S1 after K2 and before and after K3, in Stream S2 after K6, and in Stream S3 after K5.

Figure 4.3 depicts the GPU timeline produced by this experiment. Each rectangle represents a block: the j^{th} block of kernel Kk is labeled "Kk:j." The left and right boundaries of each rectangle correspond to that block's start and end times, as measured on the GPU using the `globaltimer` register. The height of each rectangle is the number of threads used by the block. The y-position of each rectangle indicates the SM upon which it executed. Arrows below the x-axis indicate kernel launch times. Dashed lines correspond to time points used in the continuing example covered in Section 4.2.2.

4.2.2 Scheduling Rules

With respect to kernels, blocks are the schedulable entities: the basic job of the GPU scheduler is to determine which thread blocks can be scheduled at any given time. These scheduling decisions are impacted by the availability of limited GPU *resources* that blocks utilize such as GPU shared memory, registers, and threads. Required resources are determined for the entire kernel when it is launched. All blocks of a given kernel have the same resource requirements.

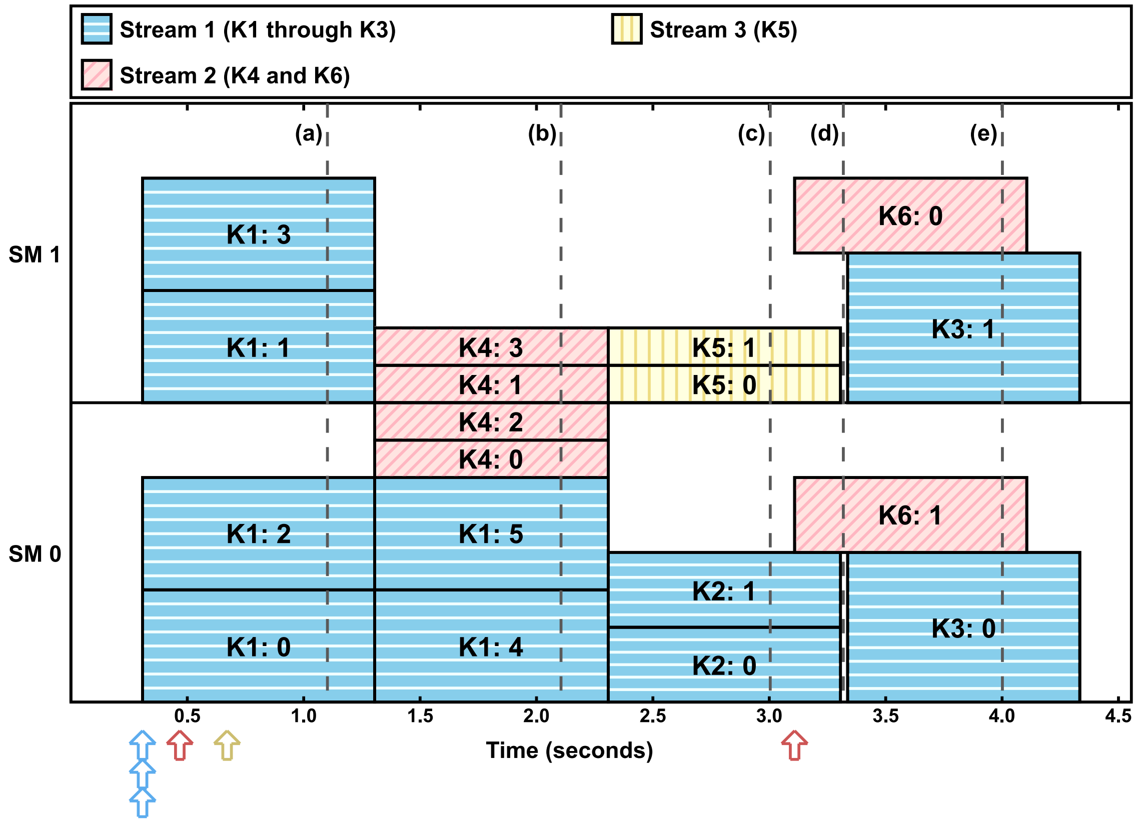


Figure 4.3: Basic GPU scheduling experiment.

We say that a block is *assigned* to an SM when that block has been scheduled for execution on the SM. A kernel is *dispatched* when at least one of its blocks is assigned, and is *fully dispatched* once all of its blocks have been assigned. Similarly, we say that a copy operation is *assigned* to a CE once it has been selected to be performed by the CE.

Example 4.2. Figure 4.4 provides additional details regarding the kernel launches in Figure 4.3. In Figure 4.4, we use additional notation to depict copies: C*k*i denotes an input copy operation of kernel K*k*, and C*k*o denotes an output copy operation of kernel K*k*. Each inset in Figure 4.4 corresponds to the time point in Figure 4.3 with the same designation (*e.g.*, inset (a) corresponds to the time point labeled “(a)”). We will repeatedly revisit Figure 4.4 to illustrate individual scheduling rules as they are stated, and then consider the entire example in full once all rules have been stated. ◊

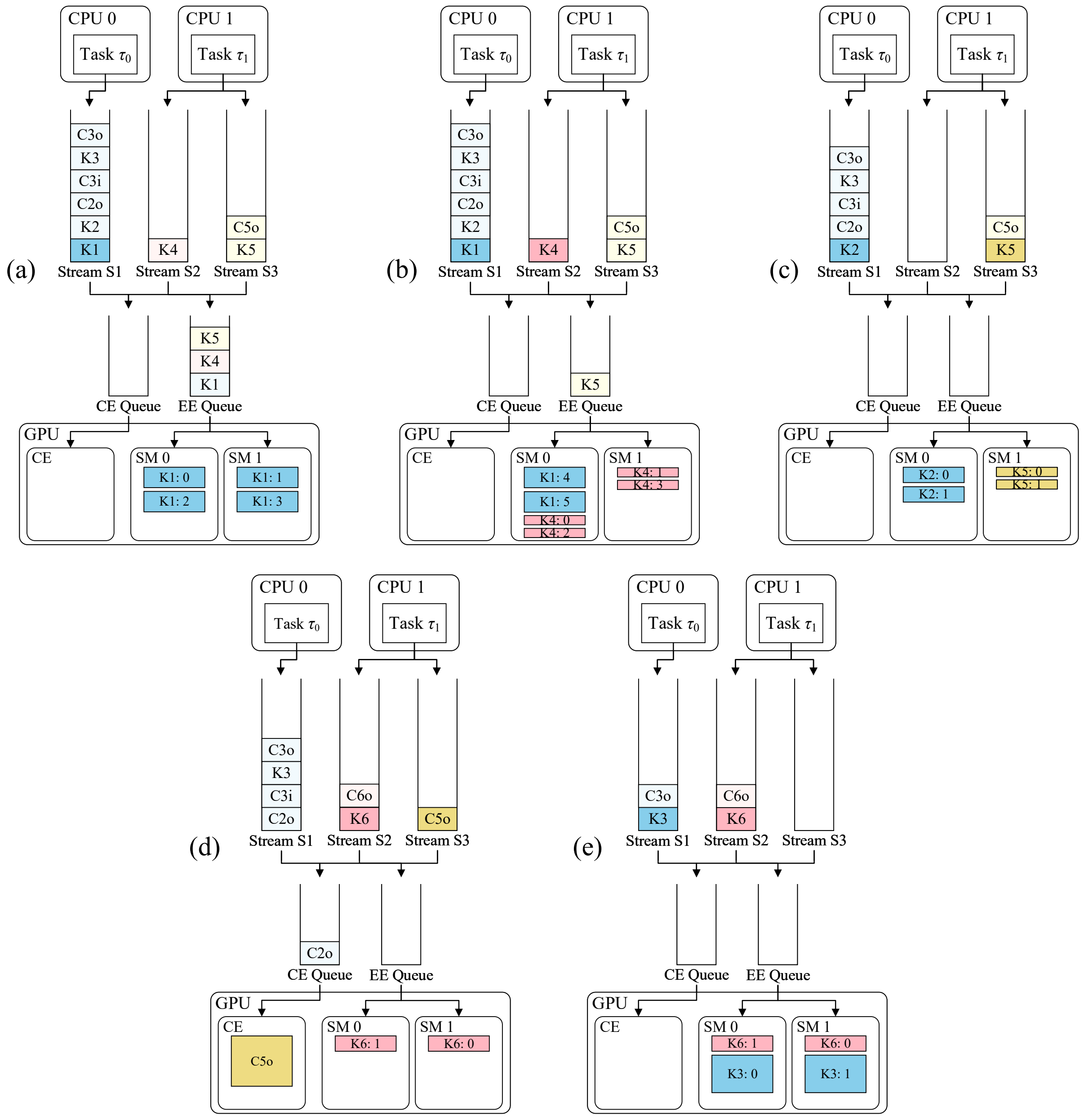


Figure 4.4: Detailed state information at various time points in Figure 4.3.

4.2.2.1 General Scheduling Rules

To our knowledge, the actual data structures used by an NVIDIA GPU to schedule copy operations and kernels are undocumented. From our experiments, we hypothesize that several queues are used: one FIFO *EE queue* per address space, one FIFO *CE queue* that is used to order copy operations for assignment to the GPU's CEs, and one FIFO queue per CUDA stream (including the NULL stream, which we consider in

Section 4.3). We refer to the latter as *stream queues*. We begin by listing general rules that specify how copy operations and kernels are moved between queues:

G1 A copy operation or kernel is enqueued on the stream queue for its stream when the associated CUDA API function (memory transfer or kernel launch) is invoked.

G2 A kernel is enqueued on the EE queue when it reaches the head of its stream queue.

G3 A kernel at the head of the EE queue is dequeued from that queue once it becomes fully dispatched.

G4 A kernel is dequeued from its stream queue once all of its blocks complete execution.

Example 4.2 (continued). In Figure 4.4, there are two CUDA programs executing as CPU tasks τ_0 and τ_1 on CPUs 0 and 1, respectively, that share an address space. τ_0 uses a single stream, S1, and τ_1 uses two streams, S2 and S3. The various queues, two SMs, and single CE are depicted in each inset of Figure 4.4. The start times in Table 4.1 give the time the kernel, or its input copy operation if one exists, was issued. Output copy operations, when present, immediately followed kernel completions.

In inset (a), τ_0 has issued kernels K1, K2, and K3 and copy operations C2o, C3i, and C3o to stream S1. τ_1 has issued kernels K4 and K5 to streams S2 and S3, respectively, and copy operation C5o to stream S3. The operations in streams S1 and S3 were enqueued in the order the CUDA commands were executed (Rule G1). Kernels K1, K4, and K5 are at the heads of their respective stream queues, and have been placed in the EE queue (Rule G2). K1 is dispatched to the GPU, so it is shaded in its stream queue. Each SM has two blocks of K1 assigned to it.

In inset (b), the remaining blocks of K1 and K4 have been assigned to the GPU, so both K1 and K4 have been removed from the EE queue (Rule G3), but remain in their respective stream queues (Rule G4). ◊

4.2.2.2 Non-Preemptive Execution

Ignoring complications considered later in Section 4.3, the kernel at the head of the EE queue will non-preemptively block later-queued kernels:

X1 Only blocks of the kernel at the head of the EE queue are eligible to be assigned.

Example 4.2 (continued). In Figure 4.4 (a), two blocks of K1 have been assigned to each SM. As explained in detail below, on each SM, there are enough resources available that two blocks from K4 could execute concurrently with the two blocks from K1. However, K4 is not at the head of the EE queue, so its blocks are not eligible to be assigned (Rule X1). ◊

4.2.2.3 Rules Governing Thread Resources

On all recent NVIDIA GPU architectures (including Kepler, Maxwell, Pascal, and Volta), the total thread count of all blocks assigned to an SM is limited to 2,048 per SM,[6] and the total number of threads each block can use is limited to 1,024. These resource limits can delay the kernel at the head of the EE queue:

R1 A block of the kernel at the head of the EE queue is eligible to be assigned only if its resource constraints are met.

R2 A block of the kernel at the head of the EE queue is eligible to be assigned only if there are sufficient thread resources available on some SM.

Example 4.2 (continued). In Figure 4.4 (a), K1 is at the head of the EE queue, so its blocks are eligible to be assigned (Rule R1). However, only four blocks of K1 have been assigned. This is because, as seen in Table 4.1, each block of K1 requires 768 threads, and each SM is limited to use at most 2,048 threads at once. Thus, only four blocks of K1's six can be scheduled together, so the remaining two must wait (Rule R2).

In inset (b), these remaining two blocks have been scheduled. K4, next in the EE queue, requires four blocks of only 256 threads each, so all of its blocks fit on the GPU simultaneously with the remaining two blocks of K1, and all four blocks of K4 are assigned. ◊

4.2.2.4 Rules Governing Shared-Memory Resources

Another constrained resource is the amount of GPU memory shared by threads in a block. On the TX2, shared memory usage is limited to 64KB per SM and 48KB per block. Similar to threads, a block being considered for assignment will not be assigned until all of the shared memory it requires is available.

R3 A block of the kernel at the head of the EE queue is eligible to be assigned only if there are sufficient shared-memory resources available on some SM.

Example 4.2 (continued). Each block of K4 or K5 requires 32KB of shared memory, so at most two of these blocks can be assigned concurrently on an SM. In Figure 4.4 (b), two blocks of K4 are assigned to each

[6]Recall from Section 2.3.1 that on the TX2, each SM has 128 hardware cores available. The (up to) 2,048 threads assigned to blocks currently executing on an SM are multi-programmed on those cores.

SM. Even though there are available threads for K5 to be assigned and it is the head of the EE queue at that time, no block of K5 is assigned until the blocks of K4 complete (Rule R3), as shown in inset (c). ♢

4.2.2.5 Register Resources

Another resource constraint is the size of the register file. On the TX2, each thread can use up to 255 registers, and a block can use up to 32,768 registers (regardless of its thread count). Additionally, each SM has a limit of 65,536 registers in total. Unfortunately, using synthetic kernels makes it difficult to demonstrate limits on registers because the NVIDIA compiler optimizes register usage. However, based upon available documentation (NVIDIA, 2019b), we expect limits on register usage to have exactly the same impact as the limits on thread and shared-memory resources demonstrated above. Note also that the number of registers can be limited at compile-time using the `maxregcount` compiler option. Decreasing the number of registers used by a kernel makes its blocks easier to schedule at the expense of potentially greater execution time.

4.2.2.6 Copy Operations

Copies are governed by rules similar to those above:

C1 A copy operation is enqueued on the CE queue when it reaches the head of its stream queue.

C2 A copy operation at the head of the CE queue is eligible to be assigned to the CE.

C3 A copy operation at the head of the CE queue is dequeued from the CE queue once the copy is assigned to the CE on the GPU.

C4 A copy operation is dequeued from its stream queue once the CE has completed the copy.

Example 4.2 (continued). As shown in Figure 4.4 (d), once K2 and K5 complete execution and are dequeued from S1 and S3, the copies C2o and C5o become the heads of S1 and S3, respectively. C5o and C2o are thus enqueued on the CE queue (Rule C1). C5o is immediately assigned to the CE (Rule C2), so it is dequeued from the CE queue (Rule C3). The CE can perform only one copy operation at a time, so C2o remains at the head of the CE queue until C5o completes. ♢

4.2.2.7 Full Example

Having presented all of the basic scheduling rules for NVIDIA GPUs, we now consider our example in its entirety.

Example 4.2 (continued). Inset (a) of Figure 4.4 corresponds to time $t = 1.1s$ in Figure 4.3. The first five kernels have been launched, and the kernels at the heads of each stream, K1, K4, and K5, have also been added to the EE queue, in issue order. K1 remains in the EE queue as it is not yet fully dispatched, so no blocks of K4 are eligible to be assigned.

Inset (b) corresponds to time $t = 2.1s$. The first four blocks of K1 have finished executing. Both K1 and K4 are now fully dispatched, so they have been removed from the EE queue, but remain at the heads of their stream queues. No blocks of K5 are able to be dispatched because their required shared-memory resources are not available.

Inset (c) corresponds to time $t = 3.0s$. Upon completion of K1 and K4, both of K5's blocks are assigned, and K2 is added to the EE queue and immediately becomes fully dispatched. C2o remains blocked by K2 in its stream queue due to FIFO stream ordering. C5o is similarly blocked.

Inset (d) corresponds to time $t = 3.32s$. K2 and K5 both complete execution, enabling C2o and C5o to be enqueued on the CE queue. C5o is enqueued first, so it is assigned to the CE and removed from the CE queue. Due to FIFO stream ordering, both C3i and K3 are delayed behind C2o. Upon being issued, K6 immediately moved unhindered through the queues, as K2 and K5 are fully dispatched; C2o, C3i, and C5o execute concurrently with the blocks of K6.

Inset (e) corresponds to time $t = 4.0s$. K3 and K6 are both fully dispatched. C3o and C6o are blocked in their stream queues until K3 and K6 complete, respectively. Thus, the CE and EE queues are empty. $\Diamond$

4.2.3 Open Questions, Answered

Recall the three potential GPU-scheduling scenarios discussed in Example 4.1 and depicted in Figure 4.2. Given the rules presented in this section, we can see that insets (a) and (c) do not depict the correct behavior. The three kernels in Example 4.1 were submitted to different streams, so K3 need not wait for K2 to complete execution, unlike in inset (a). Additionally, K3 is blocked in the EE queue until K2 is fully dispatched, unlike in inset (c). This behavior is properly depicted by inset (b).

4.3 Extended GPU Scheduling Rules

The basic rules in Section 4.2 are not the end of the story. In this section, we consider additional features available to CUDA programmers that can impact scheduling. These include usage of the default (NULL) stream and stream priorities. We also explore the differences in different NVIDIA GPU architectures and CE counts, comment on other less commonly used features that can influence scheduling that we do not examine in detail, and discuss extensions that this work has enabled.

4.3.1 The NULL Stream

Available documentation makes clear that two kernels cannot run concurrently if, between their issuances, any operations are submitted to the NULL stream (NVIDIA, 2019b). However, this documentation does not explain how kernel execution order is affected. In an attempt to elucidate this behavior, we conducted experiments in which interactions between (user-specified) streams and the NULL stream were observed. We found that these interactions are governed by the following rules, which reduce to rule G2 if the NULL stream is not used:

N1 A kernel K*k* at the head of the NULL stream queue is enqueued on the EE queue when, for each other stream queue, either that queue is empty or the kernel at its head was launched after K*k*.

N2 A kernel K*k* at the head of a non-NULL stream queue cannot be enqueued on the EE queue unless the NULL stream queue is either empty or the kernel at its head was launched after K*k*.

Example 4.3. The result of an experiment demonstrating these rules is given in Figure 4.5. The kernels launched in this experiment are fully specified in Table 4.2 (shared memory and copy operations were not used, so these are not reported). K2 and K5 were submitted to the NULL stream. K2 did not move to the EE queue until K1 completed (Rule N1); likewise, K5 did not move to the EE queue until both K3 and K4 had completed. No other kernel could move to the EE queue while K2 was at the head of the NULL stream queue, as all but K1 were launched after K2 (Rule N2). Because of the NULL-stream kernels, K6 was unnecessarily blocked and could not execute concurrently with K1, K3, or K4, so much capacity was lost. ◊

This result demonstrates that usage of the NULL stream is problematic if real-time predictability and efficient platform utilization are desired.

Table 4.2: Details of kernels used in the NULL-stream scheduling experiment in Figure 4.5.

Kernel	Launch Info	Start Time (s)	Duration (s)	# Blocks	# Threads per Block
K1	Stream S1	0.0	1.0	6	768
K2	NULL Stream	0.2	1.0	1	1,024
K3	Stream S2	0.2	1.0	4	256
K4	Stream S2	0.4	1.0	4	256
K5	NULL Stream	0.6	1.0	1	1,024
K6	Stream S3	0.8	1.0	2	256

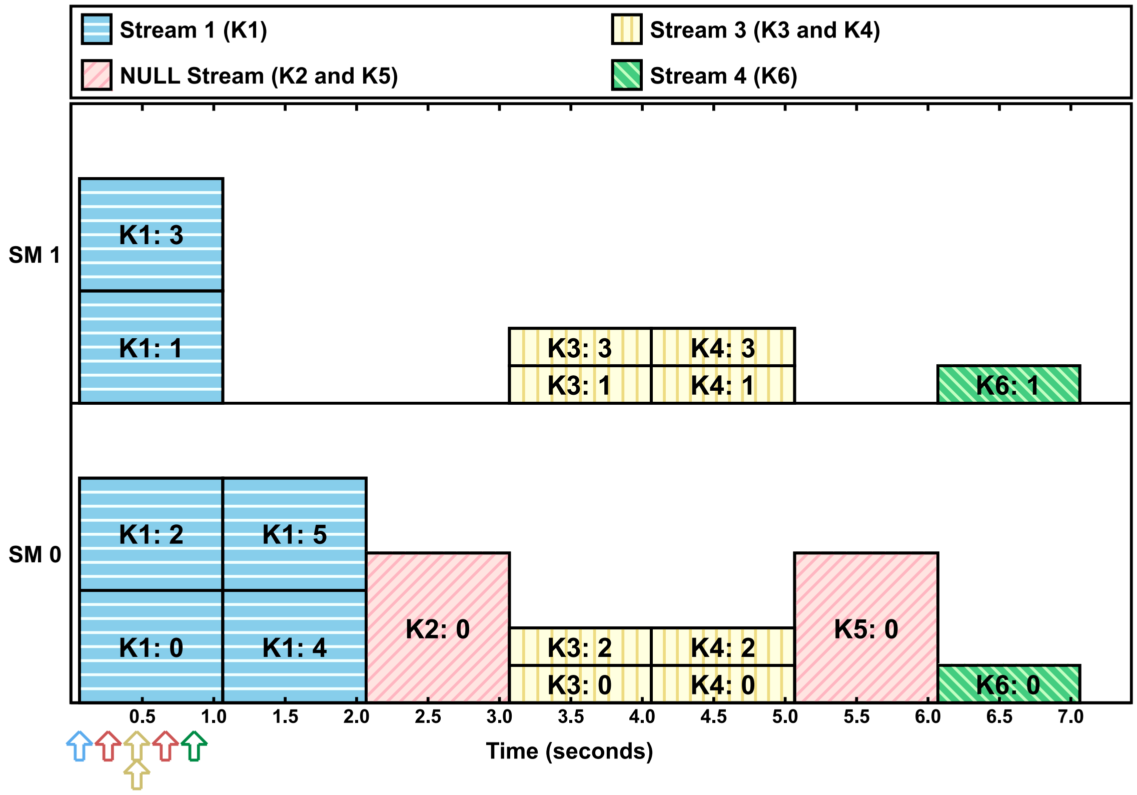


Figure 4.5: NULL stream scheduling experiment.

4.3.2 Stream Priorities

We now consider how the usage of prioritized streams impacts the rules defined in Section 4.2.2. CUDA programmers can prioritize some streams over others and can determine the allowable priority settings by the API call `cudaDeviceGetStreamPriorityRange`. On all GPUs we considered, this call returns only two priority values: -1 (*priority-high*) and 0 (*priority-low*). As we show below, a stream with no priority specified (*priority-none*) is treated as priority-low.

The following experiments use the same synthetic benchmarking techniques applied in Section 4.2.2. After discussing these experiments, we postulate new scheduling rules that specify how stream priorities affect scheduling.

4.3.2.1 Scheduling of Priority-Low Streams vs. Priority-High Streams

Given that blocks are schedulable entities, the handling of prioritized streams as described in the available CUDA documentation (NVIDIA, 2019b) is exactly as one might expect. In particular, stream priorities are considered each time a block finishes execution and a new block can be assigned, with blocks from priority-high streams always being favored for assignment if their resource requirements are met. Note that this assignment behavior can potentially lead to the starvation of priority-low streams.

Example 4.4. We conducted an experiment to illustrate this behavior using the kernels defined in Table 4.3. Figure 4.6 shows the GPU timeline that resulted from this experiment. As seen, K1 was launched first in a priority-low stream and four of its eight blocks had already been assigned when K2 and K3 were later launched in two priority-high streams. When the four initially assigned blocks of K1 completed execution, freeing all SM threads, K2 (launched second) effectively preempted K1, preventing K1's remaining four blocks from being assigned. K3 was later dispatched when K2 completed, continuing the "starvation" of K1 until K3 completed. ◊

4.3.2.2 Scheduling of Priority-None Streams vs. Prioritized Streams

The available CUDA documentation lacks clarity with respect to how scheduling is done when both priority-none and prioritized streams are used. We experimentally investigated this issue and found that priority-none streams have the same priority as priority-low streams on the TX2.

Example 4.5. Evidence of this can be seen in an experiment we conducted using the kernels defined in Table 4.4. Figure 4.7 shows the resulting GPU timeline. In this experiment, K2 was launched in a priority-none stream, K1 and K4 were launched in priority-low streams, and K3 was launched in a priority-high stream. K1 was launched first and four of its eight blocks were immediately assigned to the GPU. Once these four blocks completed execution, K3 effectively preempted K1 because K3 was at the head of a priority-high stream. After K3 executed to completion, K1 resumed execution because K2 and K4 had equal priority and could not preempt it. When K1 completed, K2 and K4 were dispatched in launch-time order. If

Table 4.3: Details of kernels used in the prioritized-stream scheduling experiment in Figure 4.6.

Kernel	Launch Info	Start Time (s)	Duration (s)	# Blocks	# Threads per Block
K1	Stream S1 (low priority)	0.0	0.5	8	1,024
K2	Stream S2 (high priority)	0.2	0.5	16	1,024
K3	Stream S3 (high priority)	0.5	0.5	16	1,024

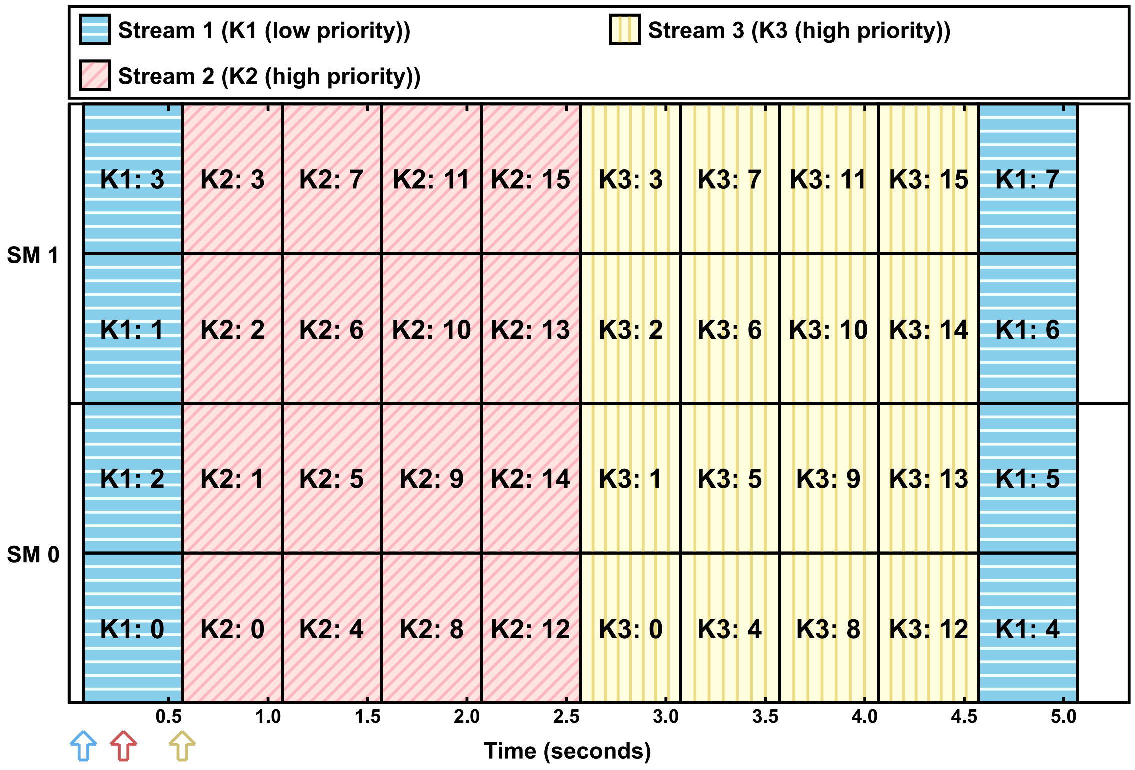


Figure 4.6: Experiment showing starvation of a priority-low stream.

priority-none were higher than priority-low, then K2 would have preempted K1; if priority-none were lower than priority-low, then K4 would have run before K2, despite being released later. ◊

4.3.2.3 Scheduling of Prioritized Streams When Resource Blocking Can Occur

We wondered whether it was possible for kernels in priority-high streams experiencing resource blocking to be indefinitely delayed by kernels in priority-low streams that "cut ahead" and consume available resources. Our experimental results suggest this is not possible.

Example 4.6. Evidence of this claim can be seen in an experiment we conducted using the kernels defined in Table 4.5, which yielded the resulting GPU timeline in Figure 4.8. K1 and K2 were launched to two distinct priority-low streams. After both were dispatched, 512 threads were available on each SM. Then K3, with

Table 4.4: Details of kernels used in the prioritized-stream scheduling experiment in Figure 4.7.

Kernel	Launch Info	Start Time (s)	Duration (s)	# Blocks	# Threads per Block
K1	Stream S1 (low priority)	0.0	0.5	8	1,024
K2	Stream S2 (unspecified priority)	0.2	0.5	8	1,024
K3	Stream S3 (high priority)	0.3	0.5	8	1,024
K4	Stream S4 (low priority)	1.2	0.5	8	1,024

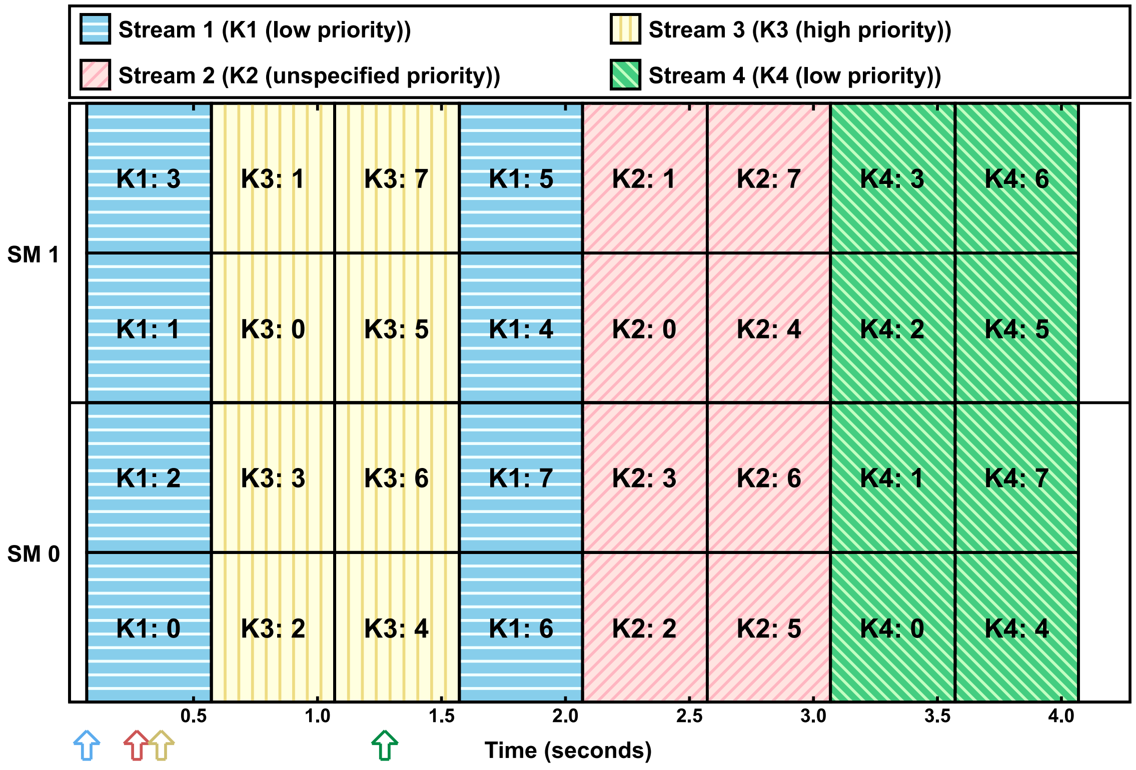


Figure 4.7: Experiment demonstrating two actual stream priority levels.

one block of 1,024 threads, was launched to a priority-high stream, followed by K4, with one block of 512 threads, launched to another priority-low stream. At this time, neither K1 nor K2 had completed execution, so the priority-high K3 could not be dispatched due to a lack of available threads. Note that even though K4 required only 512 threads, it could not be dispatched because K3 was of higher priority. Finally, when K1 completed, K3 was dispatched because at least 1,024 threads became available on an SM; K4 was then dispatched because 512 other threads were available and the priority-high K3 no longer blocked it. ◊

Table 4.5: Details of kernels used in the prioritized-stream scheduling experiment in Figure 4.8.

Kernel	Launch Info	Start Time (s)	Duration (s)	# Blocks	# Threads per Block
K1	Stream S1 (low priority)	0.0	1.0	2	768
K2	Stream S2 (low priority)	0.1	1.0	2	768
K3	Stream S3 (high priority)	0.65	0.5	1	1,024
K4	Stream S4 (low priority)	0.7	1.0	1	512

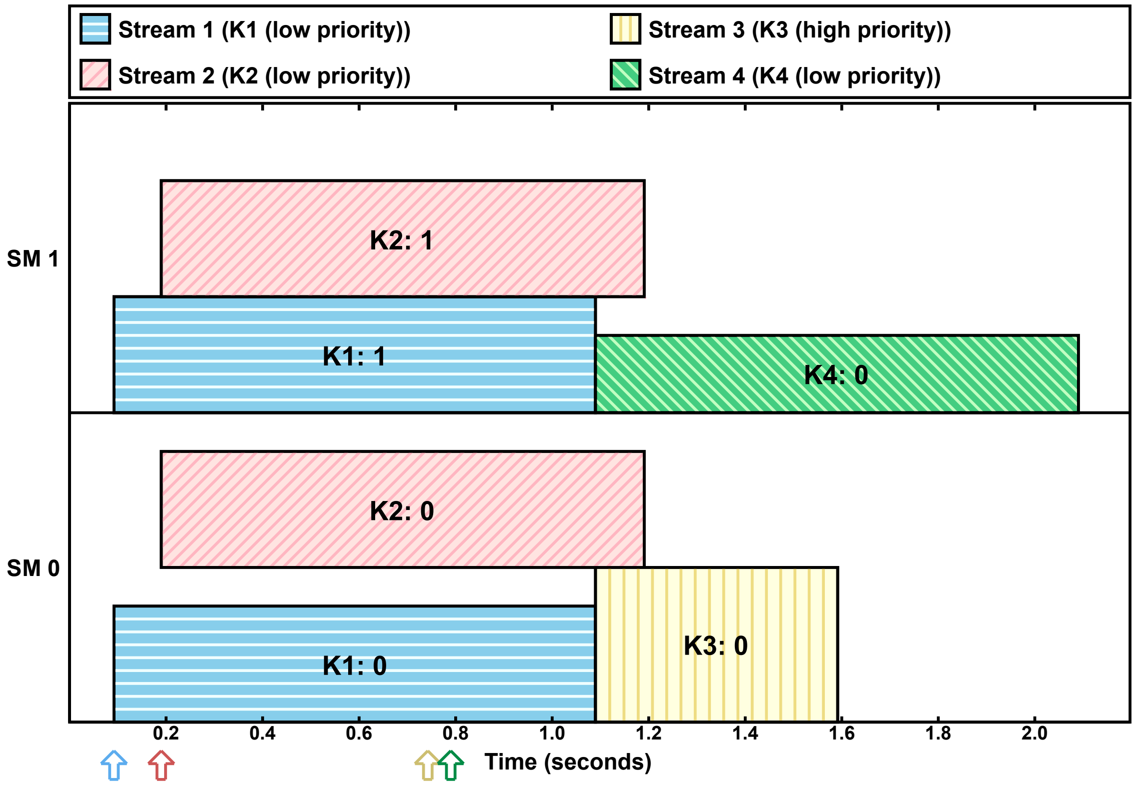


Figure 4.8: Experiment with both stream priorities and resource blocking.

4.3.2.4 Additional Scheduling Rules

Based on these experiments, we hypothesize that the TX2's GPU scheduler includes one additional EE queue, for priority-high kernels. The original EE queue described in Section 4.2.2 is for priority-low (and thus priority-none). These queues are subject to these rules:

A1 A kernel can only be enqueued on the EE queue matching the priority of its stream.

A2 A block of a kernel at the head of any EE queue is eligible to be assigned only if all higher-priority EE queues (priority-high over priority-low) are empty.

Table 4.6: Details of kernels used in the concurrent-kernel scheduling experiment in Figures 4.9 and 4.10.

Kernel	Launch Info	Start Time (s)	Duration (s)	# Blocks on TX2	# Blocks on Titan V	# Threads per Block
K1	Stream S1	0.00	1.0	1	40	512
K2	Stream S2	0.05	1.0	1	40	512
K3	Stream S3	0.10	1.0	1	40	512
K4	Stream S4	0.15	1.0	1	40	512
K5	Stream S5	0.20	1.0	1	40	512
K6	Stream S6	0.25	1.0	1	40	512
K7	Stream S7	0.30	1.0	1	40	512
K8	Stream S8	0.35	1.0	1	40	512

4.3.3 Deviations From These Rules

The experiments discussed so far in this chapter were additionally performed on three discrete GPUs (recall the GPUs we considered, as listed in Table 2.1 in Section 2.3). We now discuss observed differences between these four GPUs.

4.3.3.1 Concurrent Kernel Execution Limitations on the TX2

We observed a discrepancy between the TX2 and the discrete GPUs we considered when submitting more than four kernels to the GPU concurrently. The results of executing the same experiment are depicted in Figures 4.9 and 4.10 for the Titan V and TX2, respectively. Experiment parameters are given in Table 4.6.

This experiment suggests that when using CUDA version 10, the number of kernels that can execute concurrently depends on the platform. For the TX2, this number appears to be four; later-submitted kernels then execute sequentially, which suggests a potential source of synchronization. However, the three discrete GPUs we considered (the GeForce 980 Ti, GeForce 1070, and Titan V) were each able to support up to 32 concurrent kernels, and did not result in synchronous kernel execution. Note that both the TX2 and GeForce 1070 are both based on the Pascal micro-architecture.

Also, it is worth noting that in the original versions of our experiments (using CUDA 8), we did not observe such synchronization behavior on the TX2, and were successfully able to execute eight kernels concurrently. However, Otterness *et al*. noted significant changes between previous CUDA versions (Otterness et al., 2016), specifically CUDA 7 and CUDA 8, so such behavior differences between CUDA 8 and CUDA 10 should not come as a surprise.

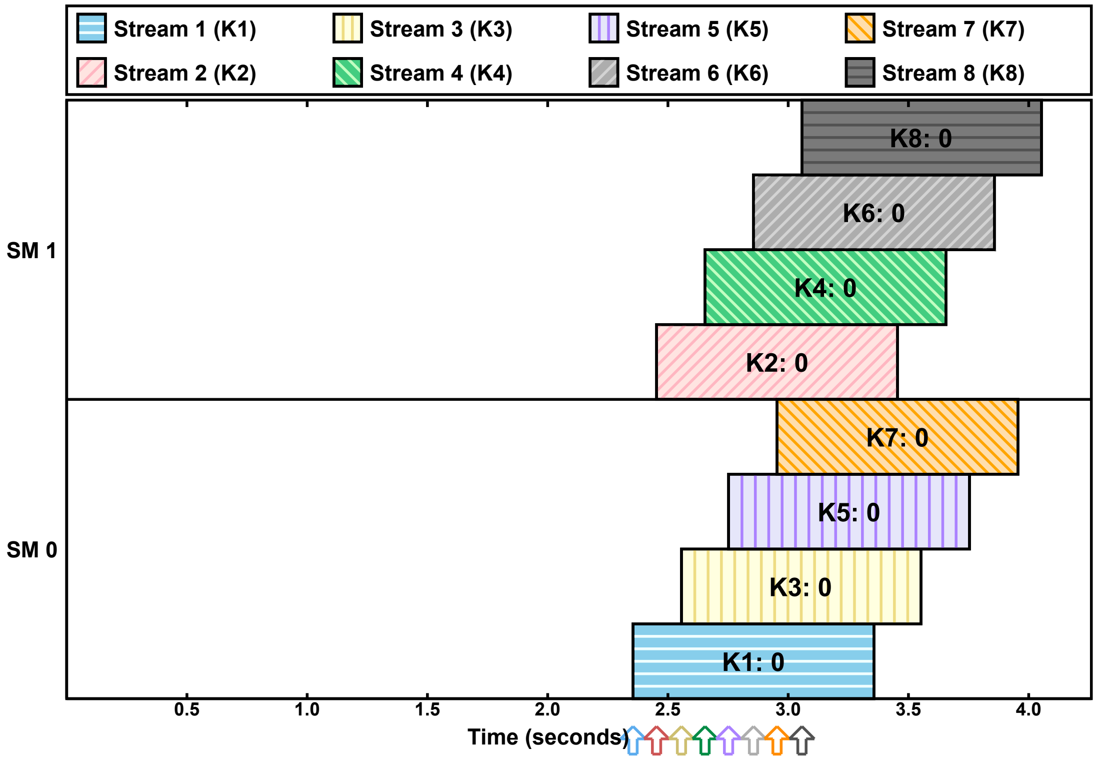


Figure 4.9: Experiment with eight concurrent kernels on a Titan V (only two of 80 SMs are shown).

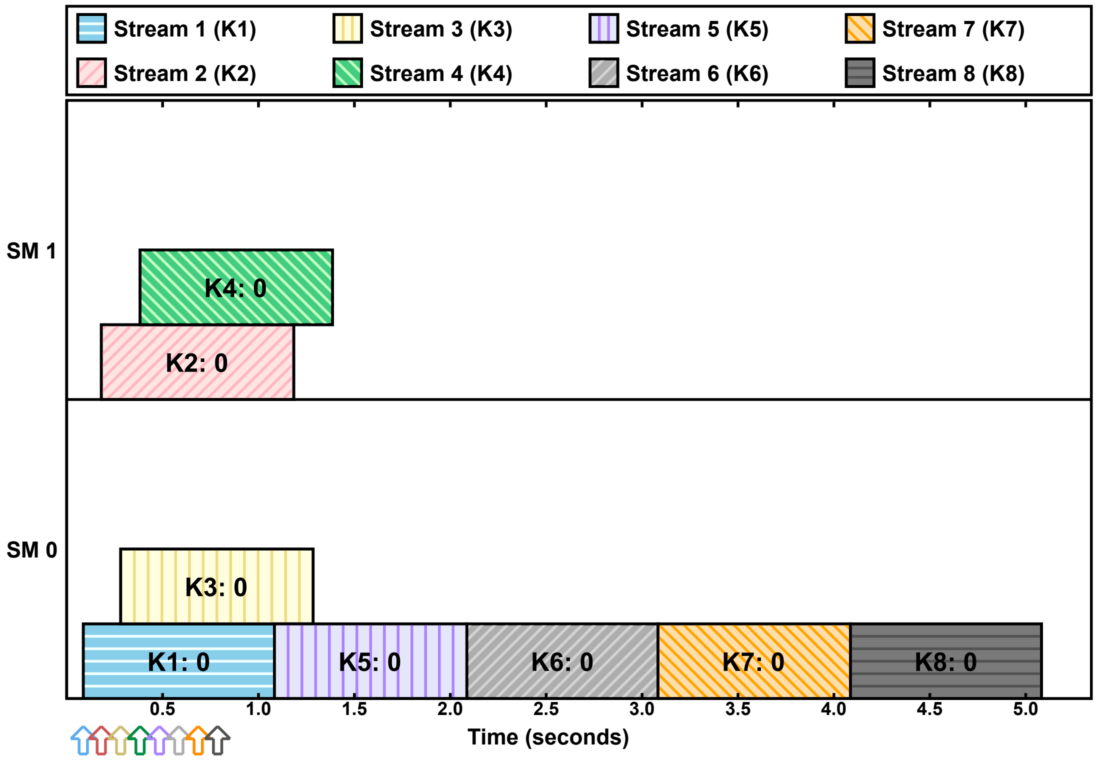


Figure 4.10: Experiment with eight concurrent kernels on a TX2.

Table 4.7: Details of kernels used in the copy-engine experiment in Figures 4.11 and 4.12.

Kernel	Launch Info	Start Time (s)	Duration (s)	# Blocks on TX2	# Blocks on 1070	Copy In	Copy Out
K1	Stream S1	0.0	1.0	2	15	-	256MB
K2	Stream S1	0.0	1.0	2	15	256MB	256MB
K3	Stream S2	0.2	1.0	2	15	-	256MB
K4	Stream S2	0.2	1.0	2	15	-	-
K5	Stream S3	0.8	1.0	2	15	256MB	256MB
K6	Stream S4	1.6	1.0	2	15	256MB	256MB

4.3.3.2 Copy-Engine Count and Concurrent Copies

Another source of discrepancies from the rules is related to the number of CEs available on a given GPU. Recall from Table 2.1 that the TX2 has a single CE, the 980 Ti and 1070 each have two CEs, and the Titan V has seven CEs. For GPUs with two CEs, the NVIDIA documentation states that each performs one direction of copies (host-to-device or device-to-host); the documentation does not make clear how the seven CEs of the Titan V behave.

Copy ordering on the TX2. We first sought to explore the difference between the single CE on the TX2 and the multiple CEs on the discrete GPUs. Although the copy-operation rules in Section 4.2.2.6 describe the behavior of copy operations relative to kernel executions, they do not tell the entire story on the TX2.

The experiments shown in Figures 4.11 and 4.12 using the 1070 and the TX2, respectively, differ only in the number of blocks per kernel, as listed in Table 4.7, and heavily feature copy operations. As shown in Figure 4.11 for the 1070, the discrete GPUs completely follow the copy-operation rules presented in Section 4.2.2.6. Therefore, copy-in and copy-out operations occur immediately before and after, respectively, their associated kernels execute, and kernel execution is not impacted by the presence of copies.

On the TX2, however, copy operations are synchronized into a single CE queue based on submission order, regardless of when they arrive at the head of their stream queues. Thus, copies in Figure 4.12 occur in submission order: c1o, c2i, c2o, c3o, c5i, c5o, c6i, and c6o. Thus, K4 cannot begin until after K2 completes (with c2o and c3o occurring in between), K5 cannot begin until after K4 begins (K4 begins immediately after c3o, at which point c5i occurs), and K6 cannot begin until after K5 completes (with c5o and c6i occurring in between).

Concurrent copies with 2 CEs. NVIDIA documentation states that for discrete GPUs with two CEs, one serves host-to-device copies and the other serves device-to-host copies. Therefore, two copies should be able

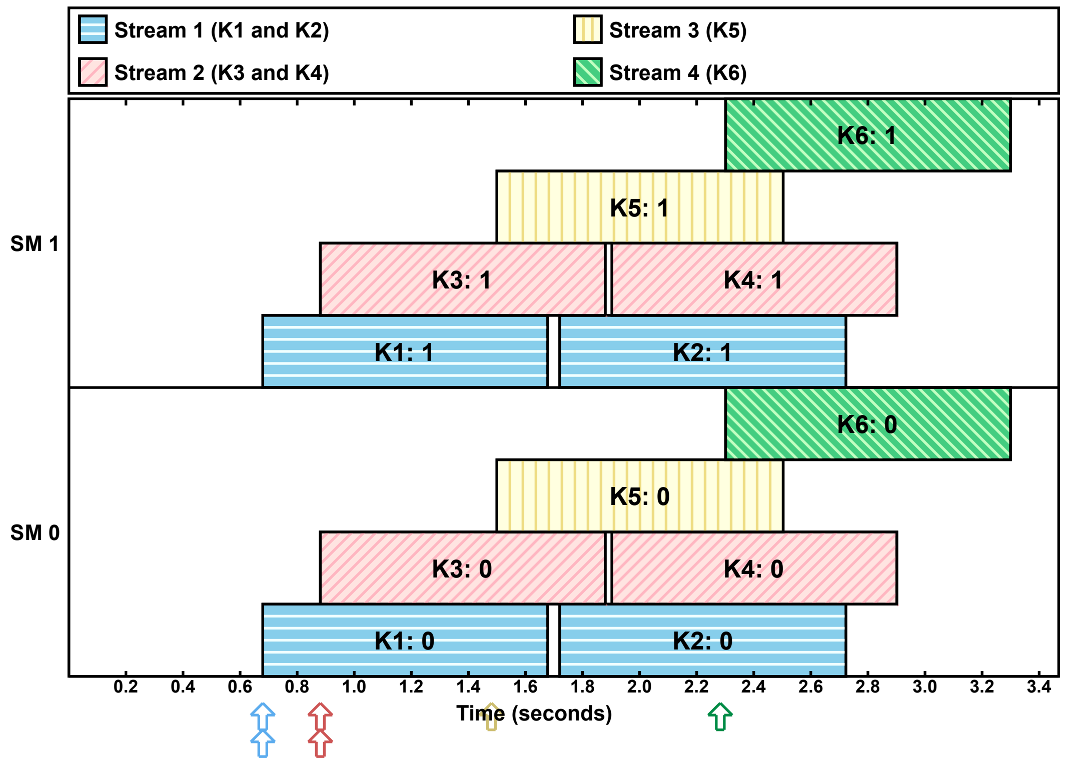


Figure 4.11: Experiment with many copy operations on a 1070 (only two of 15 SMs are shown).

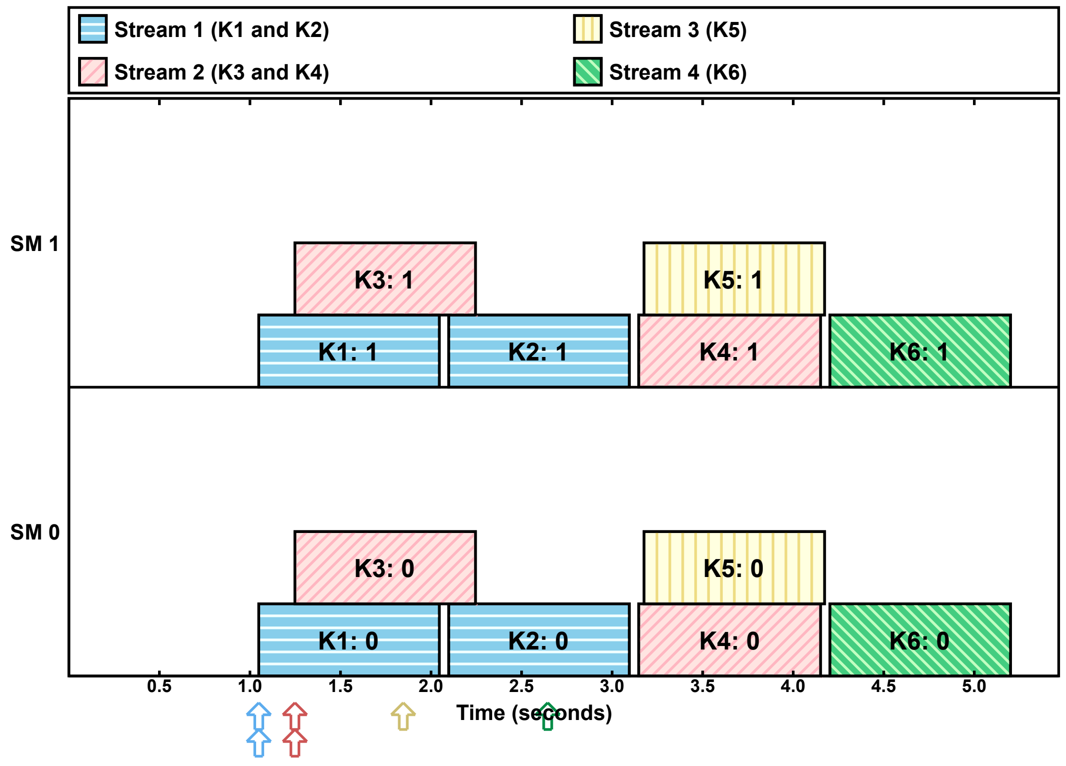


Figure 4.12: Experiment with many copy operations on a TX2.

Table 4.8: Details of kernels used in the multiple-copy-engine experiment in Figure 4.13.

Kernel	Launch Info	Start Time (s)	Duration (s)	# Blocks	Copy In	Copy Out
K1	Stream S1	0.0	1.0	15	-	-
K2	Stream S1	0.0	1.0	15	256MB	-
K3	Stream S2	0.0	1.0	15	-	256MB
K4	Stream S2	0.0	1.0	15	-	-
K5	Stream S3	0.0	1.0	15	-	-
K6	Stream S3	0.0	1.0	15	256MB	-
K7	Stream S4	0.0	1.0	15	-	256MB
K8	Stream S4	0.0	1.0	15	-	-

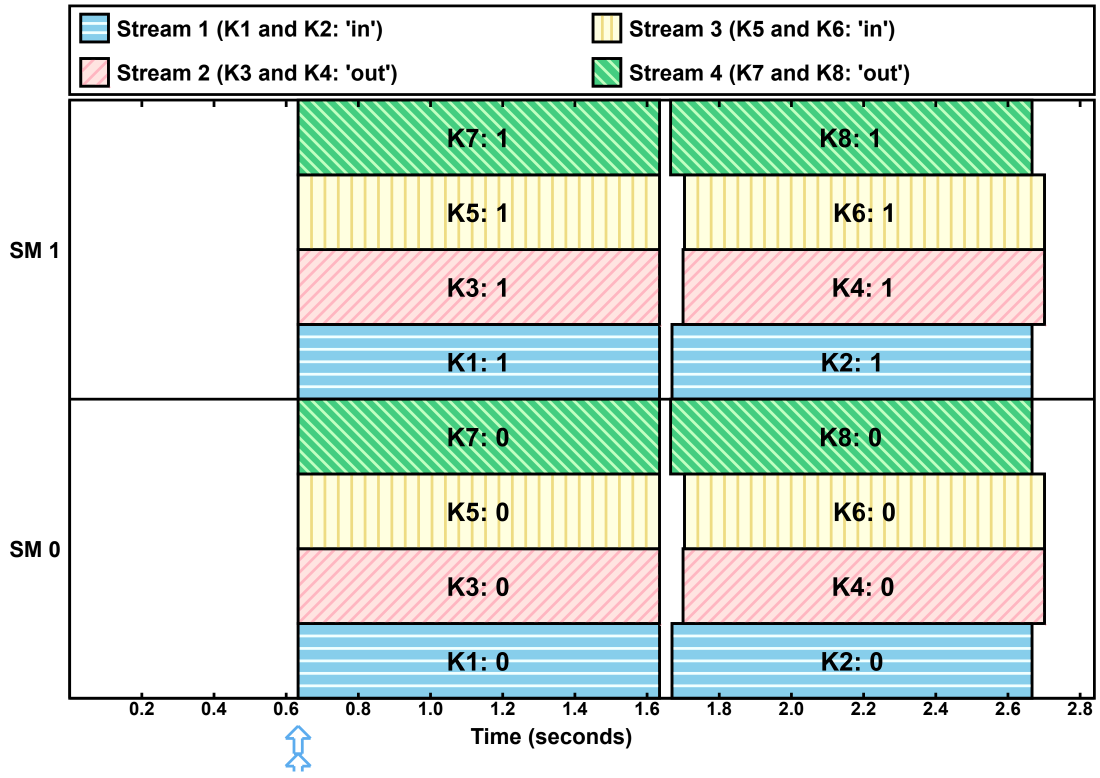


Figure 4.13: Experiment with different copy directions on a 1070 (only two of 15 SMs are shown).

to execute concurrently. We explored this in the experiment depicted in Figure 4.13 on the 1070, with kernel parameters listed in Table 4.8.

In the experiment shown in Figure 4.13, odd streams have two copies separated by a single copy-in (host-to-device) operation, whereas even streams contain a single copy-out (device-to-host) operation. Thus, if two copies may execute concurrently, pairs of streams should have the second kernels begin execution concurrently (indicating that the intermediate copies completed at the same time). This behavior is demonstrated in this

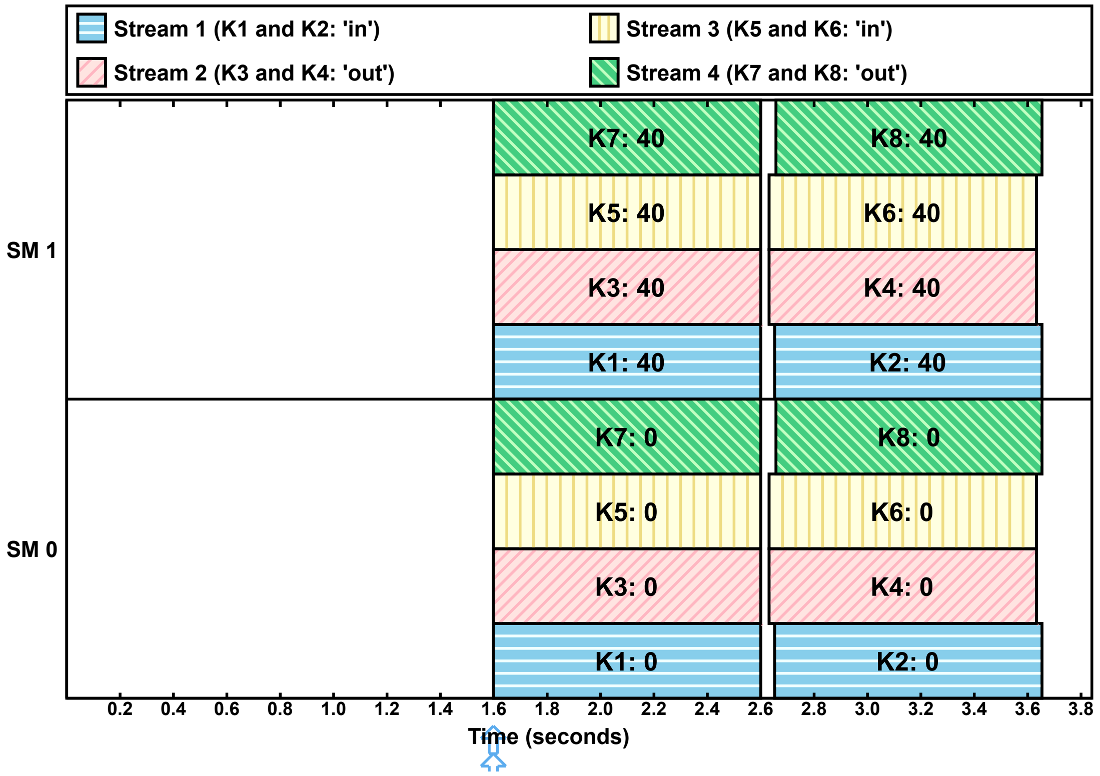


Figure 4.14: Experiment with different copy directions on a Titan V (only two of 80 SMs are shown).

experiment: K2 and K8 begin execution together, as do K4 and K6. This is because c2i and c7o executed concurrently, as did c3o and c6i.

Concurrent copies with 7 CEs. Our final copy-related experiment involves the seven CEs of the Titan V. The documentation does not make clear whether more than one concurrent copy-in or copy-out operation can occur simultaneously. To answer this question, we can perform the same experiment from Figure 4.13 on a Titan V, with 80 blocks per kernel instead of 15 (to match the Titan V's increased SM count). However, despite the presence of additional CEs, the results are the same as on the 1070, as shown in Figure 4.14.

4.3.4 Other Complications

We now note several less commonly used features that may potentially impact scheduling. These include: **(i)** the `nvcc` compilation option `--default-stream per-thread`, which enables per-task NULL streams; **(ii)** the stream-creation flag `cudaStreamNonBlocking`, which enables kernels from user-specified streams to execute concurrently with the NULL stream; **(iii)** a mechanism introduced with the Pascal GPU architecture that allows an executing kernel to be preempted at the instruction level; and **(iv)** a feature called *dynamic* *parallelism* that allows kernels to dynamically submit extra work by calling other kernel functions inside the GPU. We conjecture that these features are detrimental to use if real-time predictability is a requirement.

4.3.5 Extensions to this Work

The GPU scheduling rules presented in this chapter served as a foundation for response-time analysis of GPU-using workloads by Yang *et al*. (2018a). In their work, Yang *et al*. leveraged knowledge of GPU scheduling policies to derive improved response-time bounds for GPU-using workloads represented as DAGs.

Additionally, since the publication of the scheduling rules presented in this chapter, various works have sought to explain the remaining levels of NVIDIA's scheduling hierarchy.

At the highest level, the NVIDIA driver selects a channel from a queue called the runlist; each channel belongs to a single GPU-using application, and each application is mapped to one or more channels. Associated with each channel is a timeslice length and a priority, which determine the frequency that the channel appears in the runlist. The runlist mechanism was described by Capodieci *et al*. (2018) as part of their work to modify the NVIDIA driver and enable preemptive EDF scheduling on an NVIDIA GPU.

At the next level, the stream scheduler determines which CUDA stream may execute operations on the GPU. The stream-scheduling rules were presented in this dissertation chapter.

For each stream that is chosen to execute, the thread-block scheduler determines a mapping of blocks to SMs. Finally, within an SM, each of multiple warp schedulers determines which instructions execute from which warps at any given moment. Details of the thread-block scheduler and the warp scheduler were revealed by Sañudo Olmedo *et al*. (2020). In particular, they detailed the block-to-SM mapping via a series of expressions derived experimentally and via NVIDIA's "CUDA Occupancy Calculator" (NVIDIA, 2021), and showed that the same warp scheduler can execute both integer and floating-point operations simultaneously, enabling potential throughput improvements for GPU-using workloads.

4.4 Pitfalls When Using NVIDIA GPUs in Real-Time Systems

The rules provided in the prior sections provide only part of the necessary knowledge to use NVIDIA GPUs in safety-critical systems. As the CUDA API was designed for throughput rather than predictability, special care must be taken when using NVIDIA GPUs, even if the scheduling rules are clearly understood. Specifically, there are unexpected sources of delays that can arise due to both explicit and implicit synchronization, as well as inconsistencies in the NVIDIA documentation, which we detail in this section.

4.4.1 Synchronization-Related Pitfalls

Most developers are familiar with the concept of synchronization in a CPU-only context where two or more tasks must communicate or coordinate their actions. Synchronization becomes more complicated when a CPU task must coordinate with programs executed on the GPU. The common case is that the CPU task must determine when data in GPU memory is safe to access (*e.g.*, copy back to CPU memory). This is accomplished using *GPU synchronization*, where the GPU must complete outstanding work and reach a *synchronization point*: a point in time when data access can safely occur. There are also other, less common, cases when GPU synchronization is necessary.

In CUDA there are multiple ways to achieve GPU synchronization. They fall into two broad categories: *explicit synchronization*, which is always programmer-requested, and *implicit synchronization*, which can occur as a side effect of CUDA API functions intended for purposes other than synchronization. We have uncovered in our research some unfortunate pitfalls relating to actual GPU synchronization behavior, especially with respect to *blocking*. So, while these may not be pitfalls for non-safety-critical applications, ignoring the effects of certain specific mechanisms for achieving synchronization would be perilous in a safety-critical system where blocking must be anticipated and accounted for in analysis.

4.4.1.1 Explicit Synchronization

Explicit synchronization refers to synchronization points that the CUDA programmer explicitly requests using the CUDA API. Explicit synchronization is typically used after a program has launched one or more asynchronous CUDA kernels or memory-transfer operations and must wait for computations to complete. In contrast to implicit synchronization, the sole purpose of explicit-synchronization functions is to block the calling CPU task until the GPU reaches a synchronization point.

The CUDA documentation[7] states that explicit synchronization will block the calling task until “all preceding commands” have completed. For example, if the API function `cudaDeviceSynchronize` is invoked, “preceding commands” may encompass all commands issued to the device from all CPU tasks.

[7]Section 3.2.5.5.3 of the Programming Guide for CUDA version 10.2.89 (NVIDIA, 2019b).

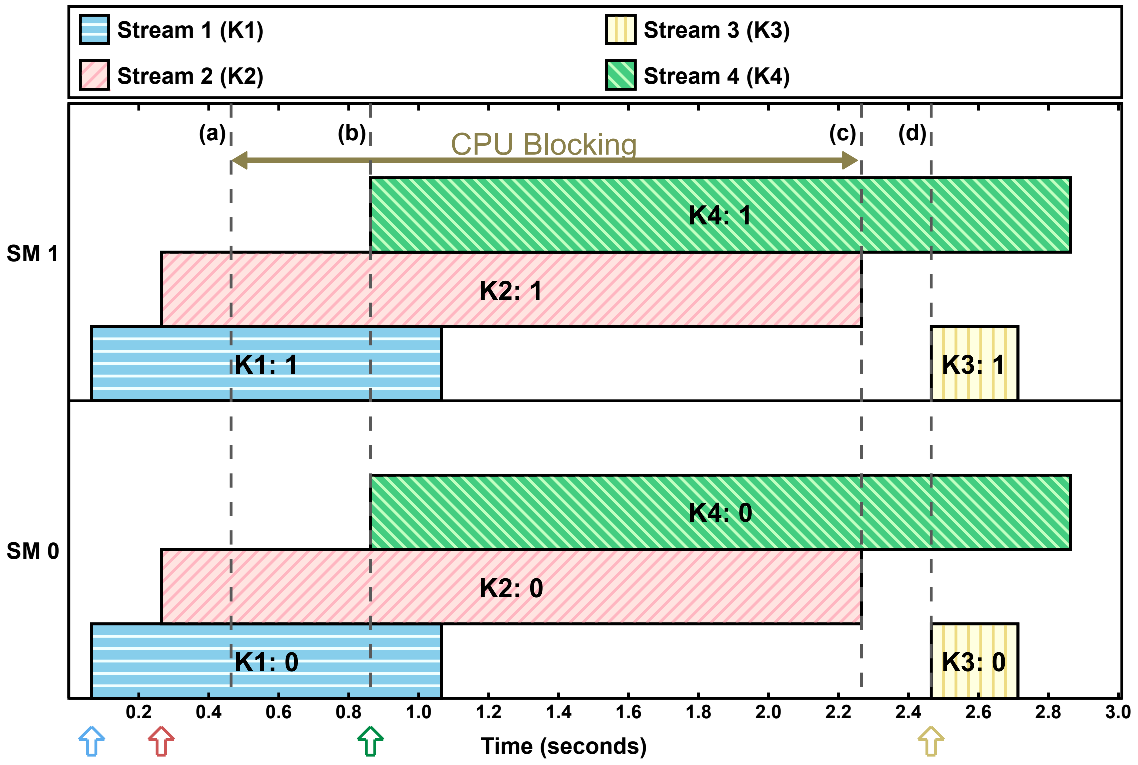


Figure 4.15: Explicit synchronization due to `cudaDeviceSynchronize`.

Other explicit-synchronization options, including `cudaStreamSynchronize`, will only block until preceding commands from a specified stream have completed.

Example 4.7. Figure 4.15 shows the results of an experiment featuring explicit synchronization. In this experiment, four CPU tasks shared a single address space, and each task launched one kernel in a separate user-defined stream. Each kernel consisted of two blocks of 512 threads, and the figure shows that one block from each kernel was scheduled on each SM. As with experiments in the previous sections, each thread of each block performed a busy-loop for a set amount of time.

An explicit-synchronization command, `cudaDeviceSynchronize`, was issued at time (a) by the CPU task responsible for launching kernel K3. This caused K3's CPU task to be blocked until the prior-submitted GPU operations, namely the executions of kernels K1 and K2, had both completed at time (c). Approximately 0.2 seconds after the synchronize completed, the same CPU task launched kernel K3. $\diamondsuit$

This behavior is exactly what one would expect, given the description of explicit synchronization from official documentation. However, our experiments also uncovered Pitfall P1 for the unwary:

P1 Explicit synchronization does not block future commands issued by other tasks.

The fact that the launch of K4 by its CPU task was not blocked at time (b) is an example of this pitfall. Implicit synchronization, which we cover next, presents even more serious pitfalls.

4.4.1.2 Implicit Synchronization

Implicit synchronization occurs as a side effect of CUDA API calls that are otherwise unrelated to synchronization. For example, implicit GPU synchronization may occur due to freeing GPU memory or launching a kernel to the default stream. Presumably, this is because some modifications to GPU device state can only occur while no kernels are executing. The CUDA documentation about implicit synchronization[8] states that "two commands from different streams cannot run concurrently if any one of the following operations is issued in-between them by the host thread:

- a page-locked host memory allocation,
- a device memory allocation,
- a device memory set,
- a memory copy between two addresses to the same device memory,
- any CUDA command to the NULL stream,
- a switch between the L1/shared memory configurations described in Compute Capability 3.x and Compute Capability 7.x."

Unlike the relatively straightforward documentation about explicit synchronization, our experiments revealed that this list includes several operations that do not necessarily cause implicit synchronization, and fails to include some functions that do. We consider this particularly problematic for real-time systems, where the ability to accurately model blocking is critical.

P2 Documented sources of implicit synchronization may not occur.

Pitfall P2 became apparent to us when, in all of our experiments, we never observed implicit synchronization as a result of a device-memory operation (allocation, set, or copy) or a page-locked host memory allocation. Our experiments covered three of the most recent CUDA versions (8.0, 9.0, and 10.2) and three

[8]Section 3.2.5.5.4 of the Programming Guide for CUDA version 10.2.89 (NVIDIA, 2019b).

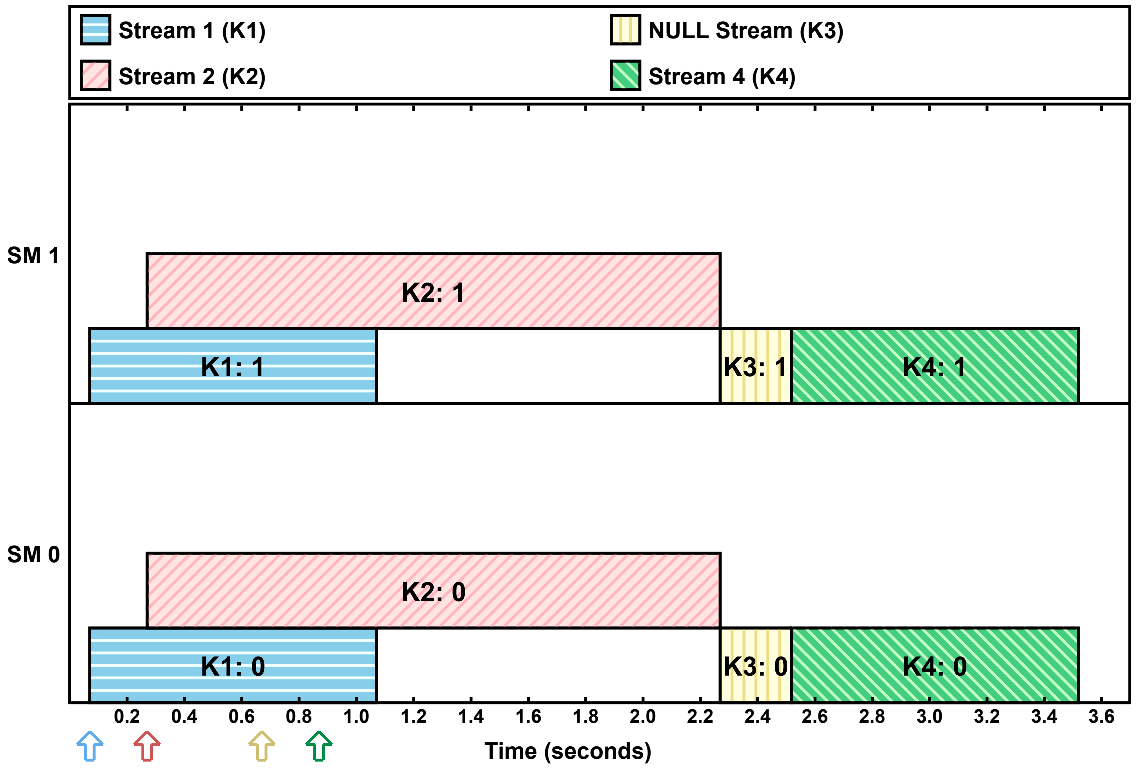


Figure 4.16: Implicit synchronization caused by a NULL-stream kernel.

of the most recent NVIDIA GPU architectures (Maxwell, Pascal, and Volta). This, of course, does not prove that implicit synchronization can *never* happen under such circumstances, but it does indicate that the documentation's statement that "two commands cannot run concurrently" is not a reliable rule.

The only case (from this list) in which we did observe implicit synchronization was launching GPU operations in the NULL stream.

Example 4.8. Figure 4.16 shows a similar scenario to the one in Figure 4.15, with one key difference: the CPU task for K3 did not call `cudaDeviceSynchronize` before K3 was launched, but instead launched K3 in the NULL stream. The implicit synchronization, and resulting loss of concurrency, is clearly visible in the figure. Execution of K3 must wait for the first two kernels to complete, and, in contrast to explicit synchronization, K4 is also prevented from running concurrently. ◊

Even though this loss of concurrency may be striking, it is notably explicitly documented, and can be used (or avoided) in a careful design for a real-time task. We found, however, a different source of implicit synchronization that is a far more problematic pitfall, and is not even listed in the documentation on synchronization: *freeing device memory*.

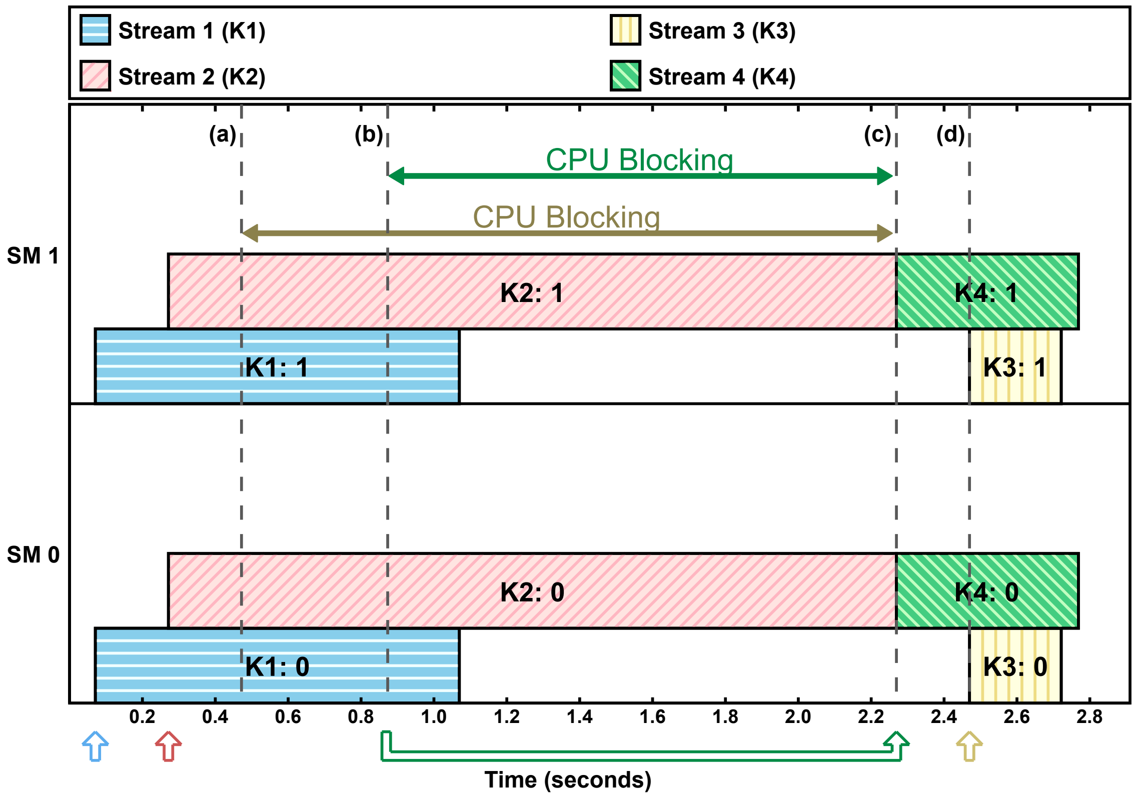


Figure 4.17: Implicit synchronization causing additional CPU blocking due to `cudaFree`.

P3 The CUDA documentation neglects to list some functions that cause implicit synchronization.

P4 Some CUDA API functions will block future, unrelated, CUDA tasks ***on the CPU***.

Example 4.9. Figure 4.17 shows the results of an experiment identical to the one in Figure 4.15, but this time the call to `cudaDeviceSynchronize` at time (a) was replaced with a call to `cudaFree`, which is used to de-allocate memory on the GPU. Pitfalls P3 and P4 can be observed in this plot. The fact that this blocked the calling CPU thread until all prior GPU work had completed at time (c) indicates that `cudaFree` created implicit synchronization. Similar to the NULL-stream behavior, implicit synchronization also prevented subsequent kernels from starting to execute until `cudaFree` completed at time (c). We speculate that this behavior by `cudaFree` is necessary because alterations to memory-mapping state requires a quiescent execution environment.

However, the most surprising effect was not that K4 was blocked, but that K4's task was blocked *on the CPU* until time (c), even though it issued an "asynchronous" kernel launch. This reveals a pitfall that can harm real-time analysis that does not consider the fact that CPU tasks can experience blocking from GPU operations that are launched from unrelated tasks. ◊

4.4.2 Best-Practices-Related Pitfalls

One of the primary challenges when designing a real-time task system that uses a GPU is that *all possible* blocking must be accounted for in analysis. Therefore, reducing the amount of blocking on both the CPU and GPU is essential. On the GPU, this requires issuing all CUDA operations to user-defined (non-NULL) streams, and carefully controlling the use of other API functions, like `cudaFree`, that cause blocking via implicit synchronization.

However, when using user-defined streams, concurrency can easily be thwarted by the presence of unexpected implicit synchronization. This gives rise to the first best-practices-related pitfall.

P5 The suggestion from NVIDIA's documentation to exploit concurrency through user-defined streams may be of limited use for improving performance in thread-based tasks.

Even though it may seem like an easy task for a programmer to just specify a user-defined stream as opposed to the NULL stream, we note that simple mistakes in doing so may be easy to miss. This is particularly true when using the `Async` versions of CUDA API functions, such as `cudaMemsetAsync`; if no stream parameter is provided, such API functions default to using the NULL stream, and thus result in implicit synchronization.

P6 Async CUDA functions use the GPU-synchronous NULL stream by default.

The fact that the CUDA documentation indicates that these functions cause implicit synchronization makes potential programmer errors even harder to notice in cases where synchronization is due to NULL-stream usage rather than memory operations.

4.4.3 Additional Documentation-Related Pitfalls

Another substantial danger stems from the inaccurate official documentation provided by NVIDIA. While function signatures and data structures seem to receive accurate (but often sparse) official documentation, scheduling and synchronization remain under-discussed. In our work to demystify implicit synchronization, we came across not only missing documentation, but incorrect documentation.

P7 Observed CUDA behavior often diverges from what the documentation states or implies.

P8 CUDA documentation can be contradictory.

Table 4.9: Observed vs. documented synchronization sources in CUDA.

	Observed Behavior			Documented Behavior	
Source	Blocks Other CPU Tasks	Implicit Sync.	Caller Waits for GPU	Implicit Sync.	Caller Waits for GPU
`cudaDeviceSynchronize`	No	No	Yes	No	Yes
`cudaFree`	Yes	Yes	Yes	**No** (undoc.)	**No** (impl.)
`cudaFreeHost`	Yes	Yes	Yes	**No** (undoc.)	**No** (impl.)
`cudaMalloc`	?	No	No	**Yes**	No (impl.)
`cudaMallocHost`	?	No	No	**Yes**	No (impl.)
`cudaMemcpyAsync D-D`	No	No	No	**Yes**	No
`cudaMemcpyAsync D-H`	No	No	No	**Yes***	No
`cudaMemcpyAsync H-D`	No	No	No	**Yes***	No
`cudaMemset (sync.)`	No	Yes	No	Yes	No
`cudaMemsetAsync`	No	No	No	**Yes**	No
`cudaStreamSynchronize`	No	No	Yes	No	Yes

For `cudaMemcpyAsync` we distinguish the direction of copy between device and host: (D-D) internal to GPU memory; (D-H) GPU memory to CPU memory; (H-D) CPU memory to GPU memory. *The documentation is contradictory for these instances, but the more detailed option indicates that these functions only cause synchronization if host memory is not page-locked. We were unable to observe this regardless of whether host memory was page-locked or not.

Example 4.10. As shown in Table 4.9, for example, `cudaFree` variants were not documented to cause implicit synchronization. Conversely, the NVIDIA documentation states that `cudaMalloc` does cause implicit synchronization, but we were unable to observe such behavior.

In the case of `cudaMemcpyAsync`, we discovered that the CUDA documentation actively contradicts itself. Section 3.2.5.1 of the CUDA Programming Guide (NVIDIA, 2019b) states "The following device operations are asynchronous with respect to the host: ... Memory copies performed by functions that are suffixed with `Async`," but Section 2 of the CUDA Runtime API documentation (NVIDIA, 2019c) states "For transfers from device memory to pageable host memory, the function will return only once the copy has completed." ◊

All of the pitfalls discussed in this section can be attributed to a single overarching problem: the black-box nature of current GPU-enabled platforms means that *developers do not have a reliable model of GPU behavior*. This highlights what is perhaps the most important pitfall for developing software meant to last:

P9 What we learn about current black-box GPUs may not apply in the future.

Despite the fact that we validated our experimental results on several of the most recent CUDA versions and GPU architectures, there is no guarantee that our results will hold after future GPU-architecture or CUDA-version updates. Even though other safety-critical hardware inevitably undergoes changes and updates, future-proof programs can still be developed against a stable specification. Likewise, the only way to truly mitigate Pitfall P9 is for GPU manufacturers to release stable, accurate documentation about their GPU platforms, along, preferably, with giving developers greater control over GPU scheduling and synchronization. Only then we can have a reliable GPU model upon which to base real-time analysis and certification. We hope that work such as ours signals to manufacturers like NVIDIA that greater openness is a desirable feature when marketing in safety-critical domains.

4.5 CUPiD$^{\text{RT}}$ Design

In this section, we introduce CUPiD$^{\text{RT}}$. To do so, we first list the issues we seek to detect in GPU-using programs, and relate them to the most egregious of the pitfalls listed in Section 4.4. Then, we describe how we designed CUPiD$^{\text{RT}}$ to detect these issues at runtime.

4.5.1 Issues We Detect

The pitfalls we identified in Section 4.4 focus primarily on ways in which GPU-using programs may behave unexpectedly. However, it is not clear how to rectify or even detect the issues that arise due to these pitfalls. In fact, the majority of the pitfalls center around errors and ambiguities in NVIDIA's CUDA documentation.

In the design of CUPiD$^{\text{RT}}$, we have focused on the most egregious issues that arise when attempting to perform timing analysis of GPU-using programs. In particular, we are concerned with unexpected cross-CPU-thread blocking (Pitfall P4) and NULL-stream GPU operations (Pitfall P6).

Issue: Unexpected CPU blocking. Of the pitfalls that arise when using NVIDIA GPUs, Pitfall P4 is perhaps the most problematic, as it relates to an unexpected source of blocking on the CPU. Such potential blocking must be taken into account during real-time schedulability analysis, and corresponds to lost system capacity. Thus, we seek to detect any sources of unexpected CPU blocking due to GPU use.

The two identified sources of this pitfall are the variants of the CUDA API functions `cudaMalloc` and `cudaFree`, which serve to allocate and free GPU memory, respectively. These function calls are necessary if GPU memory is to be used, and are only problematic while other operations (*e.g.*, kernel launches) are occurring. Thus, we exclude the setup and cleanup regions of programs from our analysis, as described in Section 4.5.2. We call the remaining region the *analyzed region*.

Issue: NULL-stream GPU operations. As identified in Pitfall P6, asynchronous CUDA functions (and kernel launches) default to using the NULL stream if no stream parameter is provided. For asynchronous functions in particular, this can cause further unexpected delays as kernels and other operations must then execute sequentially (recall the overview of NVIDIA's GPU scheduling rules in Section 4.2).

We thus also configured CUPiD^{RT} to detect the use of the NULL stream, both in kernel launches and in other CUDA API functions (*e.g.*, `cudaMemcpyAsync`). As these operations are typically part of the actual work of a GPU-using program, we do not distinguish between code regions, and have configured CUPiD^{RT} to report any detections of NULL stream use, including those during setup and cleanup.

4.5.2 Analyzing GPU-Using Programs

Program analysis can be performed in one of two ways: (1) at compile time, using a static code analyzer, or (2) at runtime, based on tracing of CUDA API calls. Although static analysis guarantees all issue occurrences are detected regardless of program inputs and settings, dynamic analysis can be used if the code path of interest is known. Additionally, as the improper usage we seek to detect is based on individual CUDA operations rather than a more complex sequence of commands, we chose dynamic analysis for CUPiD^{RT}. We leave a static-analysis implementation to future work.

CUPiD^{RT} design. CUPiD^{RT} is comprised of a series of scripts to trace the behavior of a CUDA-using program at runtime, logging all CUDA calls (*e.g.*, `cudaMalloc` or kernel launches), and then parse the log file to detect the issues described in Section 4.5.1. For the tracing stage, we utilize the `nvprof` tracing tool provided by NVIDIA. When analyzing the trace output, we detect issues corresponding to the following operations:

- `cudaMalloc`: Any invocation of `cudaMalloc` (or a variant) inside the analyzed region.
- `cudaFree`: Any invocation of `cudaFree` (or a variant) inside the analyzed region.
- NULL stream kernels: Any kernel submitted to the NULL stream, regardless of location in the program.
- Asynchronous CUDA functions: Any asynchronous function (*e.g.*, `cudaMemcpyAsync`) called with a stream parameter of 0 (the NULL stream), regardless of location in the program.
- Other CUDA functions: Any other NULL-stream-using function (*e.g.*, `cudaMemset`), regardless of location within the program.

Using CUPiD^{RT} to analyze a program. There are two steps to use CUPiD^{RT} to analyze a GPU-using program. First, the analyzed region must be annotated in the program source code. Then, CUPiD^{RT} can be used to run the GPU-using program; CUPiD^{RT} scripts analyze the tracing output and report any issues detected. Our implementation is based on CUDA 10.2, and is provided open source online.[9]

The only source-code modifications necessary to use CUPiD^{RT} are analyzed-region demarcations, for which we utilize the NVIDIA Tools Extension (NVTX) API. The statements `nvtxMarkA("TRACE_START")` and `nvtxMarkA("TRACE_END")` must be added at the start and end, respectively, of each analyzed region; these functions add messages to the trace output without otherwise affecting program functionality.

4.6 CUPiD^{RT} Evaluation

In this section, we present the results of using CUPiD^{RT} to detect issues in ten GPU-using sample applications provided with OpenCV. Then, we demonstrate the value of remedying these issues via a case study involving one of the applications.

4.6.1 Experimental Setup

We performed our experiments on a machine with a single Titan V NVIDIA GPU, two eight-core 2.10-GHz Intel CPUs, and 32-GB of DRAM. Each core includes a 32-KB L1 data cache, a 32-KB L1 instruction cache, and a 1-MB L2 cache; all eight cores on a CPU socket share an 11-MB L3 cache. We used Ubuntu 16.04, CUDA 10.2, the NVIDIA 440.33 driver, and all experiments were run using native Linux scheduling.

We evaluated the benefits of CUPiD^{RT} by analyzing ten GPU-using sample applications provided as part of OpenCV (Bradski, 2000) version 3.4, as listed in Table 4.10. These sample applications cover a broad range of different computer-vision algorithms (recall the overview in Section 2.2.3), including segmentation, feature extraction, object detection, and optical flow.

4.6.2 Evaluation of Issue Detection via CUPiD^{RT}

We list the issues detected by CUPiD^{RT} for each of the ten applications in Table 4.10. When analyzing the applications, we used default parameters when possible. For applications that act on videos, we inserted

[9]`https://cs.unc.edu/~anderson/diss/tanyadiss/`.

Table 4.10: Issue occurrences for ten GPU-using OpenCV sample applications.

OpenCV Sample	Description	`cudaMalloc`/ `cudaFree` calls	NULL stream kernels	NULL stream async functions	Other NULL stream ops
bgfg_segm	Segmentation	0	3	0	5
farneback_optical_flow	Optical Flow	10	2	0	14
generalized_hough	Feature Extraction	8	22	8	26
`hog`	Object Detection	64	64	143	157
houghlines	Feature Extraction	4	3	0	8
morphology (Erode/Dilate)	Morphology	2	2	0	2
morphology (Open/Close)		5	4	0	3
optical_flow (Brox)	Optical Flow	7	3993	40	43
optical_flow (TV-L1)		123	2937	0	262
pyrlk_optical_flow	Optical Flow	23	21	13	26
stereo_match	Stereo Matching	0	2	0	7
super_resolution	Super Resolution	26	168	0	74

the `nvtxMarkA` statements at the beginning and end of the processing loop, as discussed in Section 4.5.2. For applications that process just one or two images, we added a processing loop and the `nvtxMarkA` statements. Note that results for two `optical_flow` sample algorithms are omitted as they duplicate other samples.

4.6.2.1 Allocating and Freeing GPU Memory

We found that eight of the ten applications included calls to `cudaMalloc` and `cudaFree` within the analyzed regions. This is due, in part, to the structure of OpenCV. Sample applications showcase functionality provided by various "modules," *e.g.*, the `hog` sample application utilizes the `cudaobjdetect` module to perform GPU-based object detection. Unfortunately, inspection of the code reveals that the vast majority of the GPU-memory allocations and frees that we observed are within these modules themselves.

Observation 4.1. OpenCV modules, as implemented, are prone to unexpected CPU blocking.

Several of the applications we considered allocated and later freed over twenty regions of GPU memory when processing an individual video frame or pair of images. In fact, `hog` allocated 64 memory regions per video frame, and the TV-L1 algorithm of `optical_flow` allocated over 100 GPU-memory regions.

We expect that the execution times of these applications would be greatly improved by mediating these issues (recall the discussion of CPU blocking in Section 4.5.1); we explore the benefits of such fixes for the `hog` application in Section 4.6.3.

Unfortunately, for other applications with high GPU-memory allocation and free counts, these operations occur deep within the module implementations (*e.g.*, involving dynamic pools of GPU memory), and thus would be extremely challenging to extricate into setup and cleanup code regions.

4.6.2.2 NULL-Stream GPU Operations

The number of NULL-stream operations (both kernels and other operations) are given in the rightmost three columns in Table 4.10. Unlike the frequent GPU-memory allocations and frees throughout the OpenCV modules source code, the modules do take a CUDA stream as a parameter. Thus, the presence of issues with NULL-stream operations is primarily due to how the modules are used, rather than their implementations. However, taking a stream as a parameter does not guarantee that module-internal functionality always uses it.

Observation 4.2. Even modules that are implemented to take a stream parameter may still internally use the NULL stream.

Using CUPiD$^{\text{RT}}$, we found that all ten of the applications submitted at least some kernels to the NULL stream and performed other synchronous NULL-stream operations, and four of the ten submitted asynchronous operations (*e.g.*, `cudaMemcpyAsync`) to the NULL stream. Only two of the ten applications utilized user-defined streams for kernel execution: `farneback_optical_flow` submitted only two of its 158 kernels to the NULL stream, and for `super_resolution`, only 168 of its 884 kernels were submitted to the NULL stream. However, these remaining NULL-stream kernels are the result of inner functionality that does not use the provided stream parameter.

The `hog` application performed 143 asynchronous and 157 synchronous copy operations using the NULL stream; these NULL-stream operations were hard-coded in the module implementation. In the next section, we explore the benefits of fixing these issues for the `hog` application.

4.6.3 Case Study: Impact of Issue Remediation

To determine the benefits of remedying the issues detected by CUPiD$^{\text{RT}}$, we performed a case-study experiment using the `hog` OpenCV sample application. We modified the application to perform all `cudaMallocs` before processing the video and all `cudaFrees` after processing was complete. Additionally, we modified `hog` to perform all per-frame kernel executions and other GPU operations in user-defined streams rather than the NULL stream. After making these changes, we used CUPiD$^{\text{RT}}$ to ensure that no further issues remained.

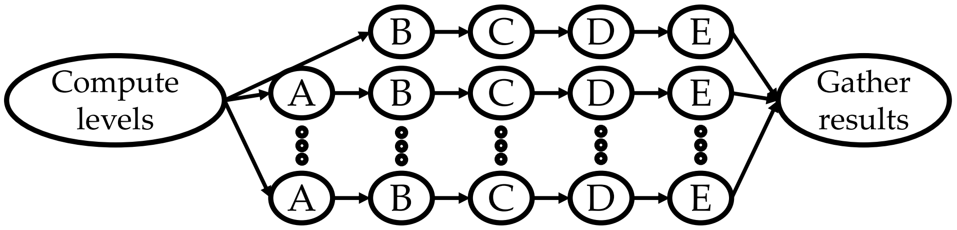


Figure 4.18: Simplified structure of the hog application.

Recall the description of the HOG-based pedestrian-detection algorithm from Section 2.1.2. We abstract the functionality of the different algorithm steps, and represent the hog OpenCV sample application as the graph shown in Figure 4.18, with the five GPU kernel operations labeled A–E. The first level processes the original image, so the resize operation (A) occurs 12 times; the other four kernels occur 13 times each. Thus, a total of 64 kernels are submitted to the GPU per video frame.

4.6.3.1 Impact on Execution Times

For the first part of our case study, we sought to measure the impact of issue remediation on the overall execution time of the hog application. We modified the application to run within a configurable number of threads, such that one, two, three, or four threads were executing hog simultaneously; this was meant to simulate a vehicle in which multiple cameras might be processing different video streams simultaneously.

All experiments represent the results of processing 5,000 video frames. For each frame, we used two calls to clock_gettime to measure the elapsed time from just before uploading the image to the GPU (thus, after reading in the frame from the video file) until just after downloading the resulting detections from the GPU and aggregating them into detected locations. The program was configured to process video frames as quickly as possible, rather than at a set frequency, thus maximizing potential cross-thread interference.

The per-frame execution times are depicted as cumulative distribution functions (CDFs) in Figure 4.19. The corresponding worst- and average-case execution times are also reported in Table 4.11, along with the 90^{th}- and 99^{th}-percentile values.

Observation 4.3. Per-frame execution times increase when multiple threads execute the hog application in parallel.

This is expected (similar behavior has been observed for other applications in prior work (Otterness et al., 2017b)) and occurs for both the original hog implementation and our modified version with issues fixed. For

Table 4.11: Per-frame execution times (in milliseconds) of `hog` with and without fixing issues.

Configuration	avg	90th	99th	max
Original x1	10.18	10.29	11.17	12.64
Original x2	17.12	18.71	21.16	24.30
Original x3	24.39	29.32	33.48	45.43
Original x4	32.83	39.77	47.61	63.36
Modified x1	2.84	2.90	2.99	5.93
Modified x2	4.93	5.67	6.16	8.77
Modified x3	5.29	7.68	9.04	14.06
Modified x4	7.36	10.29	13.06	24.27

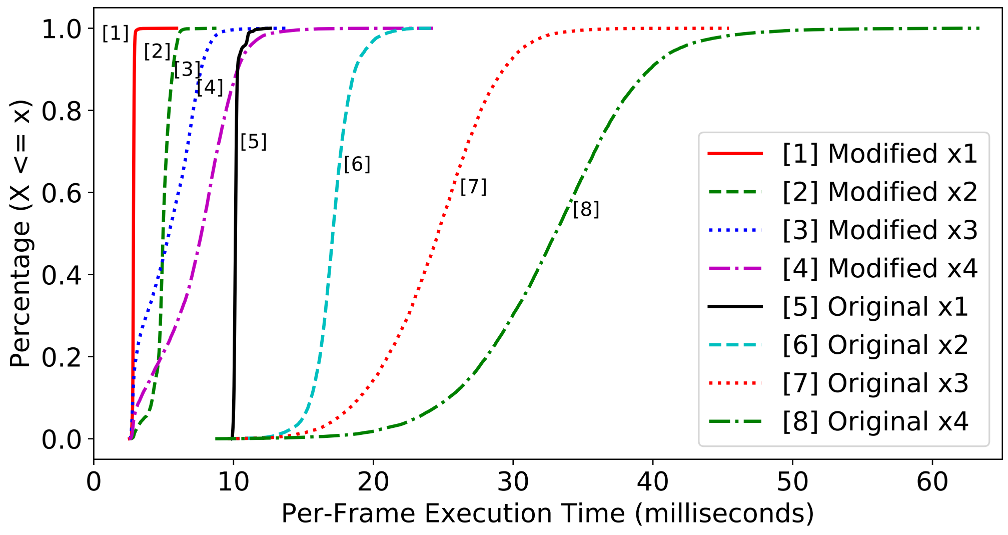


Figure 4.19: CDF of `hog` per-frame execution times with and without fixing issues.

example, the 90th-percentile of the original `hog` execution times increased from 10.29 milliseconds for one thread to 39.77 milliseconds with four threads.

Observation 4.4. Per-frame execution times significantly improve after all detected issues are fixed.

This can be observed when comparing curves between the original implementation and our modified implementation. For instance, the 90th-percentile measurement for the modified implementation with four threads was 10.29 milliseconds, a 74.1% decrease compared to the original implementation.

It is worth noting that our modified implementation with four threads had a 99th-percentile execution-time measurement of 13.06 milliseconds, which is not much higher than the worst-case measurement of 12.64 milliseconds for *one thread* in the original implementation. This, combined with the improvement of

Table 4.12: Kernel launch times (in microseconds) of `hog` with and without fixing issues, with percentage improvements in bold.

Kernel	avg	90th	95th	99th	max
A (orig.)	44.02	81.62	169.40	346.75	4134.34
A (fixed)	25.22	42.12	54.39	97.91	4352.95
% Decrease	**42.71**	**48.40**	**67.89**	**71.76**	**-5.29**
B (orig.)	56.96	178.18	242.89	407.58	8530.10
B (fixed)	17.18	37.71	56.11	109.28	5157.46
% Decrease	**69.84**	**78.84**	**76.90**	**73.19**	**39.54**
C (orig.)	91.12	224.56	297.26	462.55	5127.76
C (fixed)	26.91	59.37	84.81	153.05	5128.24
% Decrease	**70.47**	**73.56**	**71.47**	**66.91**	**-0.01**
D (orig.)	90.35	229.09	297.46	474.76	5232.71
D (fixed)	26.54	58.92	84.84	150.43	4911.89
% Decrease	**70.63**	**74.28**	**71.48**	**68.31**	**6.13**
E (orig.)	88.98	228.64	309.67	490.57	6280.39
E (fixed)	21.94	49.88	69.27	123.16	4592.75
% Decrease	**75.34**	**78.18**	**77.63**	**74.89**	**26.87**

the single-threaded configurations, suggests that CUPiD^{RT} can be useful for performance tuning in addition to ensuring predictable execution for real-time applications; we explore this in more detail later.

4.6.3.2 Impact on CPU Blocking

We can use the time required to launch[10] a kernel as a proxy for the CPU blocking time. If CPU blocking occurs, such as that depicted in Figure 4.17 (Section 4.4.1), we expect to observe multiple peaks in a histogram of kernel launch times. For example, most launches should take a small amount of time, but some launches would take longer due to overlapping with CPU-blocking operations, such as `cudaFree`. We measured kernel launch times via calls to `clock_gettime` immediately before and after each kernel-launch CUDA command.

Histograms of kernel launch times for the original `hog` implementation and our fixed version are depicted as probability density functions (PDFs) in Figure 4.20 for each of the five kernels in `hog`. These curves are normalized, such that the area under the curve sums to one. Note that the x-axis is truncated; the worst-case values range up to 8.5 milliseconds. Kernel launch times are also reported in Table 4.12.

[10]Note that this does not include the time to complete execution.

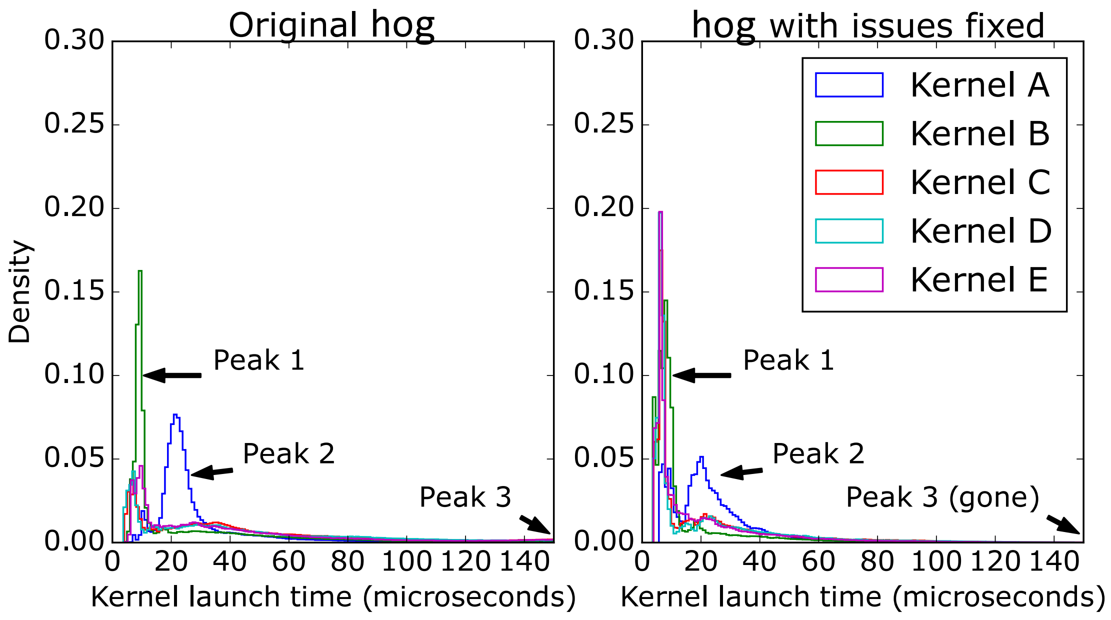


Figure 4.20: Distribution of `hog` kernel launch times before and after fixing issues, with one, two, or four `hog` threads executing simultaneously.

Observation 4.5. Fixing detected issues results in many more kernel launches taking the minimum observed time.

This behavior can be observed by comparing the height of the first peak (occurring around 8 microseconds) for each kernel between the two plots in Figure 4.20. The increase in the peak height for our modified implementation indicates that the average-case kernel launch duration is significantly reduced; this is also evident in Table 4.12.

The increase in the first peak height also suggests that our modifications led to reductions in later peaks. To observe this, we look at a different region of the PDFs, depicted in Figures 4.21 and 4.22. These plots depict the first three peaks at around 8, 21, and 165 microseconds, respectively.

Observation 4.6. Fixing issues results in much more predictable kernel launch times.

This can be observed when comparing Figures 4.21 and 4.22. In particular, fixing issues completely removed the third peak at around 165 microseconds, as shown in Figure 4.22. Furthermore, our modifications reduced the 99^{th}-percentile launch times for all kernels by up to 74.89%, as shown in Table 4.12.

Unfortunately, our modifications did not always improve the worst-case observed kernel launch times.

Observation 4.7. Worst-case kernel launch times of some kernels are higher after fixing issues.

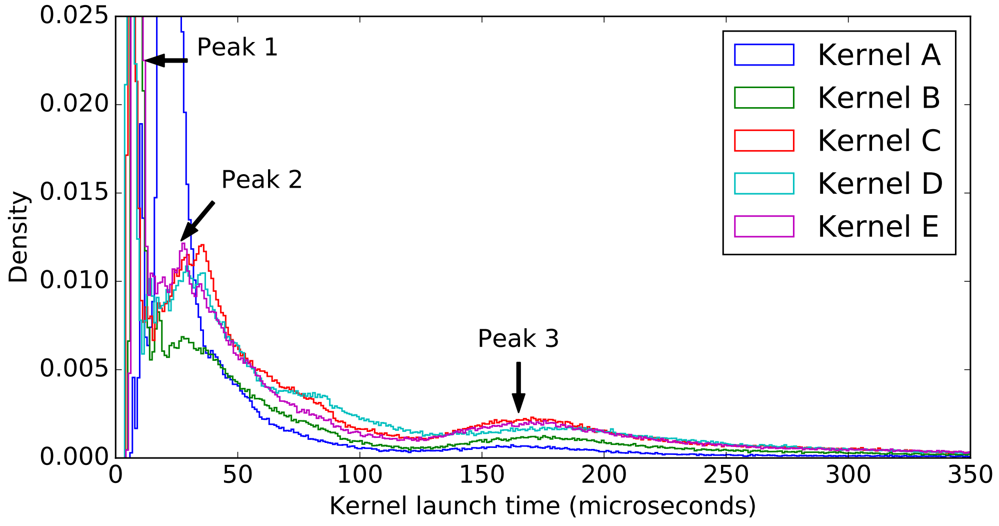


Figure 4.21: Zoomed-in distribution of the original `hog` kernel launch times.

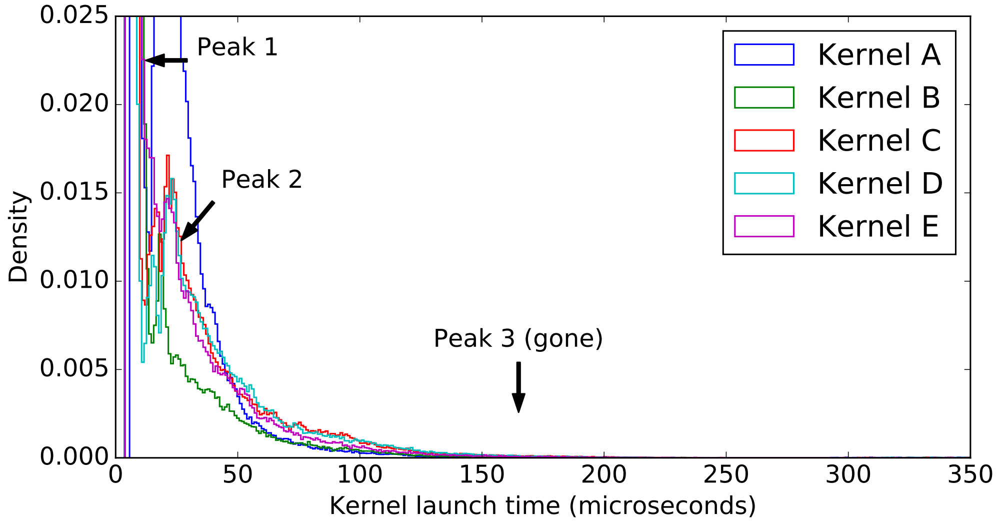


Figure 4.22: Zoomed-in distribution of `hog` kernel launch times after fixing issues.

This suggests that some other unexpected behaviors may be occurring. However, the 99^{th}-percentile results shown in Table 4.12 indicate that these are likely outliers. These outliers do, however, impact the worst-case per-frame execution times, as worst-case values for the four-thread modified implementation listed in Table 4.11 are significantly higher than the 99^{th}-percentile execution times.

4.6.3.3 Case Study Summary

We have observed both per-frame execution times and the distribution of kernel launch times between the original `hog` implementation and our modified implementation with all detected issues fixed. We now summarize the case study as a whole.

Observation 4.8. Using CUPiD$^{\mathrm{RT}}$ to ensure all detected issues are resolved improves throughput in addition to predictability.

Our fixes greatly reduced each of the average, 90th-, 95th-, and 99th-percentile kernel launch times, as listed in Table 4.12. Furthermore, our modifications resulted in more predictable memory usage; our modified implementation used 129 MB of GPU memory throughout its runtime, compared to the original implementation, for which memory usage was 139-147 MB. Combined with the improved per-frame execution times listed in Table 4.11, we have shown that the issues we have designed CUPiD$^{\mathrm{RT}}$ to detect can greatly hinder both average and worst-case performance and thus predictability.

CUPiD$^{\mathrm{RT}}$ can be a valuable tool for the development of systems which require optimizing for either worst-case or average-case performance. However, despite the benefits we have shown of resolving the issues detected by CUPiD$^{\mathrm{RT}}$, such efforts can require comprehensive application refactoring.

Observation 4.9. Not all applications are well suited to remedying the issues detected by CUPiD$^{\mathrm{RT}}$.

Fixing issues was fairly straightforward for the `hog` application. However, we had limited success using the same techniques with other OpenCV sample applications that used dynamically allocated GPU memory buffers. For these applications, remedying `cudaMalloc` and `cudaFree` uses was prohibitively challenging, as it would have required massively refactoring the underlying OpenCV modules to avoid using GPU memory-buffer pools. Further work is necessary to determine how best to fix issues in these cases.

Additionally, our fixes to the `hog` application reduced the runtime flexibility of the program; *e.g.*, the choice of grayscale versus full-color image processing cannot be changed after program initialization (as color scheme impacts GPU memory allocated). However, we consider this to be a reasonable trade-off for real-time systems, in which inputs and program behavior can reasonably be assumed stable after design time.

4.7 Chapter Summary

In this chapter, we have presented an in-depth study of the scheduling behavior of NVIDIA GPUs, as well as the potential pitfalls that may arise when developing CUDA applications for real-time systems. To combat such difficulties, we presented CUPiD$^{\text{RT}}$, a software tool designed to detect issues in CUDA-enabled applications. We used CUPiD$^{\text{RT}}$ to analyze ten GPU-using OpenCV applications, and showed that all were subject to at least one of the issues we sought to detect. Although we designed CUPiD$^{\text{RT}}$ with a focus on real-time workloads, our case study experiment demonstrated that fixing all issues detected by CUPiD$^{\text{RT}}$ can result in significant improvements in both throughput and predictability, making it a useful tool for anyone developing applications that utilize NVIDIA GPUs.

Acknowledgements. The scheduling rules (Amert et al., 2017) and pitfalls (Yang et al., 2018b) presented in this chapter (Sections 4.2–4.4) were the result of a collaboration among Tanya Amert, Nathan Otterness, Ming Yang, and Joshua Bakita.

The CUDA Scheduling Examiner experimental framework was designed and written primarily by Otterness (Otterness et al., 2017a); Amert implemented the visualization script for GPU experiments. Additionally, Amert contributed benchmarks to utilize shared memory and copy data between the CPU and GPU and designed the large experiment featured in Section 4.2 (Amert et al., 2017).

For the additional complexities detailed in Section 4.3 (Amert et al., 2017), Amert performed the NULL stream experiments and Yang performed the prioritized-stream experiments. This chapter focused on GPU operations all submitted from the same address space. However, Otterness also explored the effects due to multi-programming of GPU operations submitted from different address spaces; those results are not included in this dissertation. The further explorations of deviations from the discussed rules were performed by Amert.

Of the pitfalls discussed in Section 4.4 (Yang et al., 2018b), Pitfalls P1–P4 and P6 were determined via experiments collaboratively performed by Yang, Otterness, and Amert. Pitfall P5 was the result of extensive implementation efforts by Yang to modify multiple GPU programs to run under different CPU process/thread configurations. Table 4.9 and the associated pitfalls (P7–P9) were contributed by Otterness and Bakita.

CHAPTER 5: ENABLING TIME PARTITIONING FOR REAL-TIME MULTICORE+ACCELERATOR PLATFORMS

In this chapter, we focus our attention on ensuring temporal isolation between multiple components that require access to the same hardware accelerator that does not allow operations to be preempted; even if preemption is supported, such overheads are often prohibitively high.

We base our approach on ARINC 653 (Prisaznuk, 2008), which specifies the RTOS design for certifiable avionics platforms. The fundamental RTOS concept in ARINC 653 is a *partition*, which encapsulates a set of software modules and affords them isolation in time and space. Note that we use *partition* to indicate an allocation by the OS and *component* when considering a portion of an application system that requires such an allocation. Each component executes within a unique partition.

Broadly speaking, *time partitioning* means that at most one component may access each processing resource (*e.g.*, a CPU core) at any time, and *space partitioning* means that components cannot adversely interfere with each other in accessing non-processing resources, such as memory. Space partitioning on multicore platforms can be provided by memory-protection functions, *e.g.*, page coloring (Liedtke et al., 1997; Mueller, 1995). As depicted in Figure 5.1, time partitioning in ARINC 653 is achieved via two-level scheduling: time slices are allocated to partitions, and for each partition, an in-partition scheduler allocates time to tasks of the contained component. However, time partitioning is violated if an accelerator access by one component extends beyond a time-slice boundary. Thus, a non-preemptive access must be postponed if it may cross a time-slice boundary.

In this chapter, we introduce TimeWall (Time-Isolated Multicore Execution With AcceLerator Locking), a framework for providing time partitioning on multicore+accelerator platforms. TimeWall consists of a hierarchical scheduler and a specialized locking protocol, which orchestrates accelerator accesses while respecting time-slice boundaries. We additionally present an experimental evaluation of TimeWall.

Organization. The remainder of this chapter is organized as follows. In Section 5.1, we describe our system model and provide additional background on the real-time multiprocessor locking protocol that we extend. In Section 5.2, we describe the different schedulers and the modified locking protocol that comprise TimeWall.

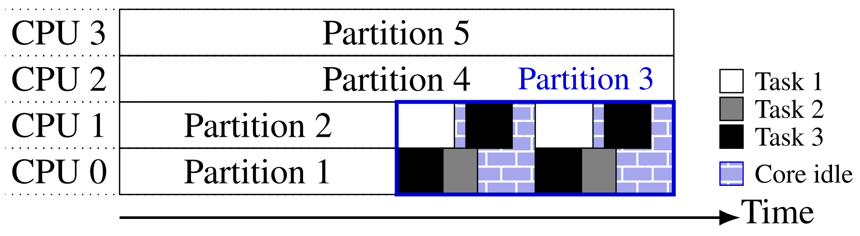


Figure 5.1: Five partitions on four cores. Partition 3 executes on two cores; other partitions each execute on one core. In-partition scheduling is shown for Partition 3.

We discuss the various challenges we encountered in moving from theory to practice in Section 5.3, and present our experimental evaluation of TimeWall in Section 5.4 before concluding in Section 5.5.

5.1 System Model

We now describe our system, task, and accelerator-request models, as well as the locking protocol we build upon.

5.1.1 Component Model

We consider a multicore+accelerator platform comprised of m identical CPUs alongside a set of accelerators. We allow different types of accelerators, *e.g.*, GPUs, digital signal processors (DSPs), and field-programmable gate arrays (FPGAs).

In this chapter, we assume that all system components, and their assigned partition time slices, have been predetermined. Associated with each component Ω is a set τ of tasks to be scheduled. During a time slice, Ω has exclusive access to specified sets of CPU cores and accelerators. Note that some accelerators (*e.g.*, GPUs) can be broken into multiple virtual accelerators (NVIDIA, 2020; Otterness and Anderson, 2021); we leave such sharing to future work.

5.1.2 Task Model

We assume that component Ω's task set τ is comprised of n implicit-deadline tasks. In some AI applications, a task's utilization may exceed 1.0 (*e.g.*, due to merging the nodes of a cycle into a supernode, as in Section 2.4.4.2). Such tasks cannot be partitioned onto a single processor without over-utilizing the processor. Thus, for such tasks to be scheduled, CPU scheduling within a component must be done via global scheduling (*e.g.*, G-EDF), considering all jobs of all tasks in Ω together on the CPUs available to Ω, with consistent deadline tie breaking. Task utilizations exceeding 1.0 also necessitate that some degree of intra-task parallelism be allowed. Therefore, in this chapter we assume the rp-sporadic task model introduced in Section 3.1.3.

5.1.3 Request Model

We consider accelerators to be shared resources that can be accessed by at most one job at a given time.[1] Such accesses can be managed using a real-time mutual-exclusion locking protocol. A job J_i may issue one or more requests, $\mathcal{R}_i^1, \mathcal{R}_i^2, \ldots$, to the locking protocol; we use $\mathcal{R}_i$ to represent an arbitrary request of a job of task τ_i. Once J_i is granted access, $\mathcal{R}_i$ is said to be *satisfied* until the job releases the lock. Request $\mathcal{R}_i$ is *active* from its issuance until J_i releases the lock; an active request is either waiting to acquire the lock or is satisfied.

5.1.4 Global OMLP

As discussed in Section 5.1.2, allowing tasks with utilization exceeding 1.0 precludes partitioned scheduling, and we therefore require global scheduling (*e.g.*, G-EDF) and a multiprocessor mutual-exclusion locking protocol. One such protocol is the suspension-based global OMLP (Brandenburg and Anderson, 2010), which has been shown to have optimal priority-inversion blocking (pi-blocking) under suspension-oblivious analysis, which is the suspension-accounting method usually used under G-EDF.

When used on m processors, the global OMLP ensures $O(m)$ pi-blocking by utilizing a dual-queue structure, with an m-element FIFO queue fed into by a priority queue, as depicted in Figure 5.2. Using the global OMLP, when a new request is issued, it is enqueued in the FIFO queue if fewer than m requests are already active, and in the priority queue otherwise. When the request at the head of the FIFO queue (*i.e.*, the lock holder) completes, it is dequeued, and the next request (if any) in the FIFO queue becomes satisfied; if the priority queue is not empty, the highest-priority request is moved from the priority queue to the tail of the FIFO queue.

Example 5.1. Figure 5.3 depicts six jobs that issue requests to the global OMLP with $m = 4$. The global OMLP state shown in Figure 5.2 corresponds to the set of active requests at time 4.5. The first four requests

[1]Similarly to the division of accelerators into virtual accelerators that could be accessed concurrently by different components, we leave the exploration of intra-component accelerator sharing for future work. For this work, we assume mutual exclusion for simplicity.

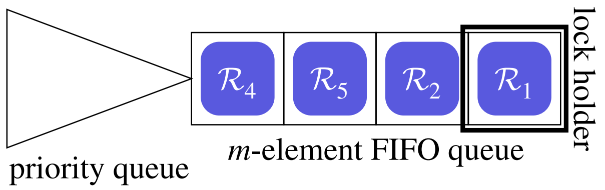


Figure 5.2: The global OMLP structure for $m = 4$ CPUs.

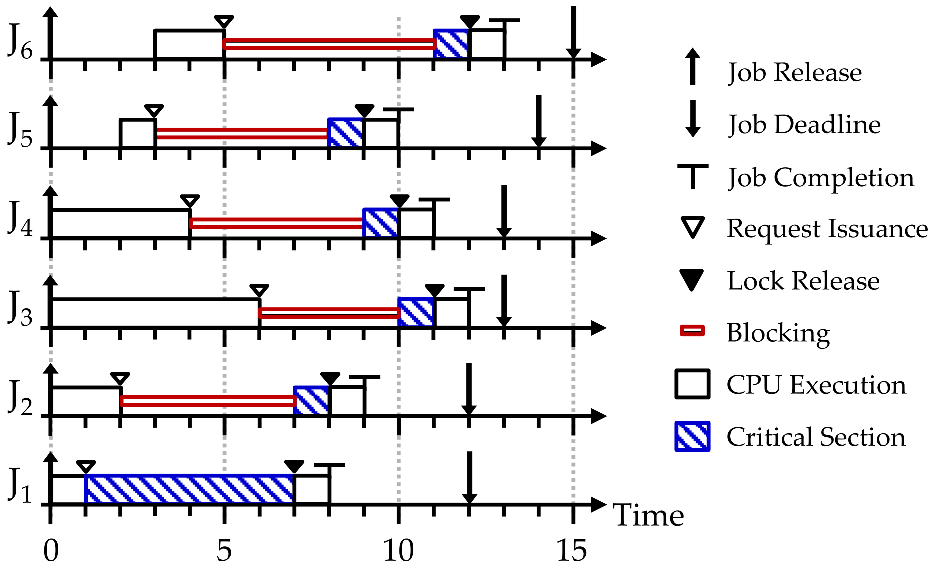


Figure 5.3: Jobs issuing requests to the global OMLP with $m = 4$ CPUs.

issued are enqueued directly in the FIFO queue in issuance order; thus, $\mathcal{R}_5$ is satisfied before $\mathcal{R}_4$, even though J_4 has higher priority than J_5.

Requests $\mathcal{R}_6$ and $\mathcal{R}_3$ are enqueued in the priority queue upon issuance, as the FIFO queue is full. When J_1 releases the lock at time 7, $\mathcal{R}_2$ becomes satisfied (*i.e.*, the head of the FIFO queue), and $\mathcal{R}_3$ is moved from the priority queue to the FIFO queue, as J_3 has higher priority than J_6. Thus, $\mathcal{R}_3$ is satisfied before $\mathcal{R}_6$, despite being issued later. $\Diamond$

The global OMLP relies on the concept of *priority inheritance* to ensure progress: the lock holder "inherits" the priority of the highest-priority active job waiting to acquire the lock, if that priority exceeds that of the lock holder itself.

Example 5.1 (continued). While job J_5 holds the lock from time 8 to time 9, it inherits the priority of job J_3, which has the highest priority of any job enqueued in either the FIFO or the priority queue. $\Diamond$

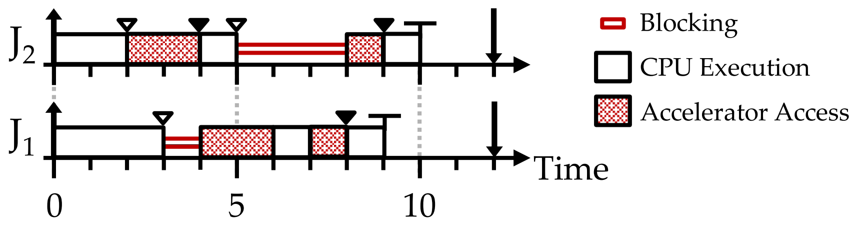


Figure 5.4: Two jobs issuing requests to access the same accelerator.

5.1.5 Accelerator Access Model

We refer to the computations performed while a request is satisfied as its *critical section*. We assume that each job of a task τ_i may make any number of accelerator accesses and that successive accesses to the *same* accelerator may be grouped into the critical section of a single request. We denote by $Y_i^{k,1}, Y_i^{k,2}, \ldots$ the accelerator accesses occurring during the critical section of request $\mathcal{R}_i^k$.

Example 5.2. Two request-issuing jobs are depicted in Figure 5.4. Job J_2 makes two separate lock requests, whereas the two accesses by job J_1 are grouped into a single request.

Request $\mathcal{R}_2^2$ is active in the interval $[5,9)$: it is blocked by both accesses of $\mathcal{R}_1^1$ from time 5 to time 8 and then satisfied from time 8 to time 9 while $\mathcal{R}_2^2$'s critical section executes. $\Diamond$

5.2 TimeWall Design

Certification procedures tend to evolve slowly over time. Furthermore, avionics certification is typically stricter than that for automotive applications. Given this reality, our proposal for ensuring time partitioning on a multicore+accelerator platform is based on the current ARINC 653 time-slicing approach for avionics.

We abstract the idea of time slicing by ensuring that each component Ω is granted exclusive periodic access to a set Υ of computing resources (accelerators as well as M unit-speed CPUs, where $M \leq m$) by defining a *periodic component reservation* (PCR) for Ω (similarly to the Single Time Slot Periodic Partition model of Mok and Chen (2001)).

The PCR for component Ω is defined as a three-tuple (Θ, Π, Υ), denoting that Ω receives exclusive access to the computing resources in Υ within continuous intervals of Θ time units that begin every Π time units $(\Theta \leq \Pi)$.

Example 5.3. Figure 5.5 shows the first few time slices for four components on a platform with four CPUs and two GPUs. In this example, Component A is specified by $(2, 3, \{\text{CPU 0}, \text{CPU 1}, \text{CPU 2}, \text{GPU 0}\})$. $\Diamond$

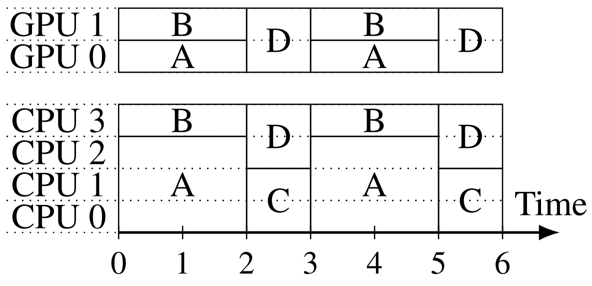


Figure 5.5: A time-sliced schedule. Rectangles represent reservations.

We now introduce TimeWall, a framework to enable time partitioning on multicore+accelerator platforms.

5.2.1 Scheduling Hierarchy

At the core of TimeWall is a two-level scheduling hierarchy. The top-level scheduler is the *partition allocator* (PA), which ensures that partitions are scheduled according to their PCRs. The PA is realized using a table-driven scheduling approach. We assume that the table is determined offline; optimizing the table creation is outside the scope of this dissertation.

The second-level scheduler is the *in-partition scheduler*. Conceptually, any multiprocessor scheduler could be used; our implementation uses G-EDF, but other G-EDF-like schedulers (Erickson and Anderson, 2012) could be applied similarly. In the rest of the chapter, we focus on the allocations within a single arbitrary component, so we will simply refer to that component's in-partition scheduler as "the scheduler."

To realize this scheduling hierarchy, we extended the existing reservation-based scheduling mechanisms available in LITMUS^{RT}. In our implementation, each reservation is contained within a scheduling environment: the PA corresponds to an environment that schedules partition reservations, and each in-partition scheduler is associated with an environment that schedules task reservations. Using this hierarchical approach, a job can query the remaining budget for its partition, which is necessary to enforce time partitioning, as discussed next.

5.2.2 Time Partitioning via Forbidden Zones

We chose the global OMLP to ensure mutually exclusive access to each accelerator. We apply the global OMLP within a component, treating each accelerator in Υ as a separate resource protected by a unique global OMLP lock with an M-element FIFO queue. We leave the exploration of other protocols to future work.

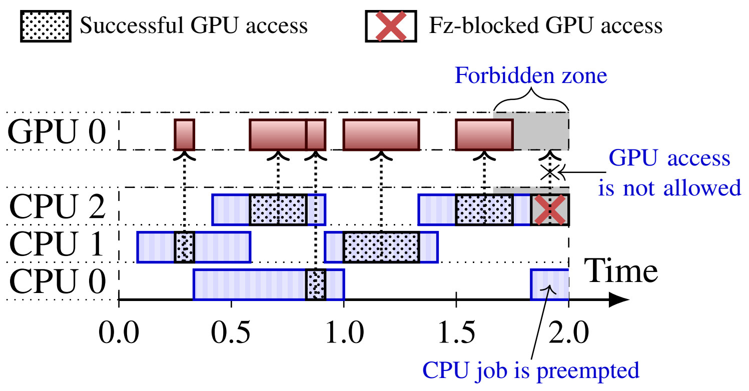


Figure 5.6: Expanded view of the first time slice of Component A from Figure 5.5. Rectangles represent jobs of tasks in Component A.

The key challenge of time partitioning accelerators is preventing the continuation of a non-preemptive accelerator access past the end of a time slice. To prevent such time-slice overruns, we use a variant of a concept known as a *forbidden zone* (Holman and Anderson, 2006).[2] Defined for each access to a non-preemptive accelerator, a forbidden zone is the time interval in which the access may not be *initiated*, otherwise it may not complete before the end of the component's time slice. Thus, the length of the forbidden zone for a given access is the worst-case duration of that access—*accelerator usage by other components has no impact on a given component's forbidden-zone lengths*. Note that the use of a forbidden zone requires that no accelerator access takes more than Θ time units.

Example 5.3 (continued). Forbidden zones are illustrated in Figure 5.6, which depicts a detailed view of the execution of Component A from Figure 5.5 within the time interval $[0,2)$. The forbidden zone corresponding to the final GPU access by a job executing on CPU 2 is shown in grey, prior to the time-slice boundary at time 2. $\Diamond$

The enforcement of forbidden zones in TimeWall is applied at the level of an *individual accelerator access*, rather than an entire critical section. To enable this fine-grained enforcement, we augmented the global OMLP to include an additional "forbidden-zone-check" mechanism in addition to the traditional "lock" and "unlock" functionality. Prior to initiating an accelerator access $Y_i^{k,\ell}$, the job J_i holding the lock must invoke the forbidden-zone check, which verifies that $Y_i^{k,\ell}$ is not within its forbidden zone, *i.e.*, that the time remaining in the time slice is at least the worst-case duration of $Y_i^{k,\ell}$. Otherwise, the forbidden-zone

[2] A similar idea was later applied in a component-based setting (Behnam et al., 2007), but that work focused on uniprocessor CPU platforms and did not allow for skipping ahead in a forbidden zone, which we discuss later.

check suspends J_i until the next time slice of its containing component. We call such forbidden-zone-induced blocking *fz-blocking*.

Example 5.3 (continued). As shown in Figure 5.6, the last job to execute on CPU 2 accesses GPU 0 twice. The first GPU access is initiated at time 1.5, before the start of its forbidden zone; this access is allowed to execute, as it will complete before the end of the time slice, by the definition of the forbidden-zone length. The second GPU access is fz-blocked, and cannot begin until Component A's next time slice. CPU execution, like that on CPU 0, is allowed and is preempted at time 2.0. $\Diamond$

5.2.3 Bounding Fz-Blocking

Bounds on additional blocking due to using forbidden zones have been provided in previous work (Holman and Anderson, 2006), assuming that forbidden zones were enforced for each entire critical section. We now show how to compute zone-based blocking due to per-access forbidden zones. Consider a job J_i that initiates a lock request $\mathcal{R}_i$ with critical-section length B, where $\mathcal{R}_i$ is comprised of one or more accesses for an accelerator $A \in \Upsilon$. Let $B_{max}^{A,\Omega}$ (resp., $K_{max}^{A,\Omega}$) denote the worst-case critical-section length of a lock request (resp., worst-case duration of an access) for A by any task in component Ω. Ignoring zone-based blocking, under the global OMLP, $\mathcal{R}_i$ will be blocked for up to $X = 2\,(M-1)\,B_{max}^{A,\Omega}$ time units (Brandenburg and Anderson, 2013), *i.e.*, it will finish at most $X + B$ time units after being initiated.

The following lemma bounds the blocking experienced by a lock request when forbidden zones are enforced for each accelerator access.

Lemma 5.1. *The total blocking introduced by the management of non-preemptive accelerator accesses is at most* $X + \lceil (X+B) \,/\, \left(\Theta - K_{\max}^{A,\Omega}\right) \rceil \cdot K_{\max}^{A,\Omega}$ *time units for each lock request by a task in Component* Ω *for an accelerator* $A \in \Upsilon$.

Proof. In each time slice of length Θ, at least $\Theta - K_{\max}^{A,\Omega}$ time units are available for job J_i to initiate accesses to A, as $K_{\max}^{A,\Omega}$ is the maximum forbidden-zone duration for any access to A. (Recall that job J_i may not access other accelerators until $\mathcal{R}_i$ completes and the lock is released.)

In executing this work, and while the last access in $\mathcal{R}_i$ is unfinished, additional blocking of up to $K_{\max}^{A,\Omega}$ time units may be incurred for each time-slice boundary crossed. Thus, we upper bound the number of such boundaries that may be crossed between the initiation and completion of $\mathcal{R}_i$. The worst case occurs when $\mathcal{R}_i$ is initiated right before a time-slice boundary, in which case $\lceil (X+B) / \left(\Theta - K_{\max}^{A,\Omega}\right) \rceil$ boundaries are crossed. □

5.2.4 Performance Optimizations

The delays caused by fz-blocking can be mitigated slightly via two performance improvements, which we discuss now.

Skipping ahead. If the lock holder is fz-blocked due to an accelerator access in its forbidden zone, we can allow other requests to "skip ahead" of that access until the beginning of the next time slice. This corresponds to the *Skip Protocol* proposed previously (Holman and Anderson, 2006), but requires some additional machinery due to the separate enforcement of forbidden zones and critical sections. Because allowing a request to skip ahead reorders the global OMLP's queues of requests, *critical-section lengths* rather than individual access durations are compared to the time remaining in the slice to determine whether skipping ahead is allowed. A consequence is that individual accesses are not permitted to skip ahead. Additionally, the priority-inheritance mechanism used by the global OMLP must be modified: while a job is fz-blocked and another job is allowed to skip ahead, only the skipping-ahead job inherits the highest priority of any enqueued job (*i.e.*, the fz-blocked job resumes its original scheduling priority until it is no longer fz-blocked).

Example 5.4. Consider the set of active lock requests depicted in Figure 5.7 (note that request and access subscripts are dropped here, as they are not relevant). Suppose that after $Y^{1,1}$ completes, the job becomes fz-blocked, *i.e.*, that the worst-case duration of $Y^{1,2}$ is longer than the time remaining in the time slice.

In this case, a later-enqueued request may be allowed to skip ahead. The next request that can be satisfied is $\mathcal{R}^5$, as its critical-section length is less than the worst-case duration of $Y^{1,2}$. Note that $\mathcal{R}^3$ is not eligible, even though its individual accesses are each of shorter duration than $Y^{1,2}$. ◊

Requests that skip ahead do so while another job is fz-blocked, and thus do not introduce any additional blocking.

Merging accelerator accesses. Each request for accelerator accesses incurs delays due to locking-protocol overhead as well as pi-blocking. As given by Lemma 5.1, such blocking depends on B_{max}, the worst-case duration of any critical section.

As we allow multiple accelerator accesses within a critical section, a trade-off arises as to whether to merge successive accesses into a single critical section. If a job J_i acquires the lock for each access

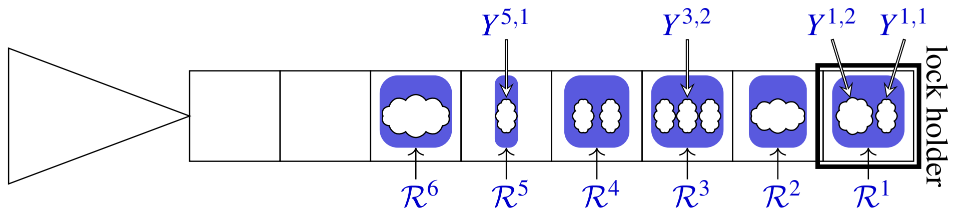


Figure 5.7: The global OMLP with support for forbidden zones. Request and access widths indicate worst-case critical-section and access durations, respectively.

individually, the locking-protocol overhead and pi-blocking suffered by J_i are duplicated for each access. However, if the accesses are merged, then the single critical section has longer duration than if the lock were separately acquired for each access, increasing the blocking experienced by other jobs. We discuss the overhead associated with this trade-off for our implementation in Section 5.4.1 and explore the effects of merging accelerator accesses in more detail via case-study experiments in Section 6.3.

5.3 Theory Meets Practice

In theory, forbidden zones ensure total isolation between accelerator-using components, *but does such isolation occur in practice?* To answer this question, we implemented the two-level scheduler and forbidden-zone-aware global OMLP comprising TimeWall within the 5.4.0-rc7 LITMUS$^{\text{RT}}$ kernel (Brandenburg, 2011; Calandrino et al., 2006) and conducted experiments involving a GPU.[3] These experiments led to several surprises, which we detail throughout this section.

Experimental setup. Our experimental platform contains two eight-core 2.10-GHz Intel Xeon Silver 4110 processors and one NVIDIA Titan V GPU. Each CPU core utilizes a 32-KB L1 instruction cache, a 32-KB L1 data cache, and a 1-MB L2 cache; all eight CPU cores on a socket share an 11-MB L3 cache. To mitigate spatial interference between components, we partitioned each L3 cache evenly between components executing on the corresponding socket. We also disabled hyperthreading and graphics output for all experiments.

Our case study, described in more detail in Section 5.4, featured a GPU-enabled version of the pedestrian-detection `hog` application. Recall from the descriptions in Sections 2.1.2 and 4.6.3 that `hog` utilizes seven GPU operations: a CPU-to-GPU image copy-in, five GPU kernels, and one final GPU-to-CPU result copy-out. To avoid notation confusion, we refer here to the five GPU kernels as K1–K5.

[3]Source code is available at `https://cs.unc.edu/~anderson/diss/tanyadiss/`.

Table 5.1: Statistics for durations of the two copies and five kernels comprising the `hog` case study in microseconds, as measured on the GPU using `nvprof` and on the CPU using `clock_gettime()`.

Device	Statistic	Copy-In	K1	K2	K3	K4	K5	Copy-Out
GPU	max	77	27	42	56	28	49	29
CPU	99^{th}	154	144	138	73	30	64	46
	99.5^{th}	157	146	139	74	31	65	47
	99.9^{th}	180	154	146	79	45	70	52
	99.95^{th}	200	161	148	86	49	76	57
	99.99^{th}	1391	1342	163	1265	55	1236	69
	max	5247	1393	1388	1332	1286	1317	1300

To provision forbidden zones for the seven GPU operations in `hog`, we needed the worst-case duration of each access. Due to the complexity of multicore platforms, the industry standard for WCET analysis on such platforms is measurement based (Wilhelm, 2020). Thus, for 25,000 video frames, we collected GPU- and CPU-based GPU-access-duration measurements, using NVIDIA's `nvprof` profiling tool and `clock_gettime()`, respectively. The results are listed in Table 5.1, including the 99^{th}, 99.5^{th}, 99.9^{th}, 99.95^{th}, and 99.99^{th} CPU percentiles.

Perplexing edge cases. We expected the CPU-measured GPU-access durations to approximately match what we measured on the GPU (with minor CPU-GPU communication overhead), yet this behavior only holds up to the 99.95^{th} percentile of CPU measurements. In fact, the worst-case CPU-measured times are *two orders of magnitude* higher than those measured on the GPU. *What was causing such extreme edge cases, and why had prior work not noted these cases?*

Prior work seems to have obviated edge cases either by only timing GPU accesses on the GPU (Heo et al., 2020; Mei et al., 2017; Yang et al., 2018b) or by using only a percentile of measured CPU times (Yang et al., 2018a, 2019; Yao et al., 2020). We can make no such simplification. Provisioning forbidden zones using lower percentiles would risk GPU accesses crossing time-slice boundaries, violating temporal isolation. However, as the average- and worst-case diverge, fz-blocking using worst-case measurements becomes exceedingly conservative, greatly reducing GPU utilization. Only one choice remained: we needed to identify the cause of the edge cases.

5.3.1 Investigating Potential Culprits

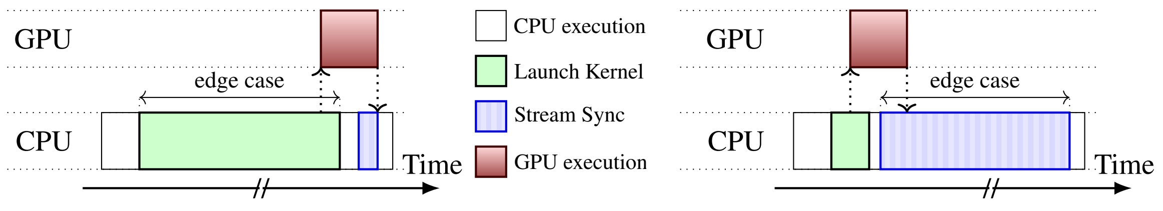
Prior work identified several potential issues in using GPUs in real-time systems, providing us with a starting point. After ruling out the pitfalls listed in Chapter 4, we suspected GPU interrupt handling as the edge-case culprit, as Elliott and Anderson (2012) detailed priority inversions that could occur due to interrupt processing for CPU-GPU communication in prior kernel and GPU driver versions.

Interrupt processing. In Linux (upon which LITMUS^{RT} is based), interrupts are processed in two steps: "top halves," which typically execute immediately and non-preemptively to acknowledge the interrupt, and "bottom halves," which handle the interrupt processing itself. Despite recent changes in Linux interrupt handling (Rybczyńska, 2020), bottom halves in LITMUS^{RT} still execute at a lower priority than any LITMUS^{RT} task, leading to potential priority inversions.

We mitigated edge cases due to top halves by reserving one CPU per socket for top-half handling. To identify if bottom-half-related priority inversions were causing the edge cases, we used KUTrace (Sites, 2018, 2021), a tracing tool that provides a timeline of all work on the system, including interrupts, syscalls, and page faults. The trace data surprisingly revealed that interrupts were not at fault; each edge case occurred entirely in userspace. However, this raised the question: what source could account for a userspace slowdown of this magnitude?

Power management. The trace data revealed an additional oddity, providing our next clue: just before a worst-case GPU access completed, all CPUs other than the one awaiting notification of a GPU-operation completion would exit a low-power state. Thus, we sought to disable power management, including both low-power states and CPU frequency scaling.

For low-power states, we disabled Linux's `cpuidle` mechanism (Brown, 2016). Frequency scaling proved more challenging, and in fact, we discovered that it is impossible to completely disable frequency scaling in modern Intel processors. Thus, we implemented a monitor to periodically log CPU frequency. We correlated these measurements with our trace data, and observed no CPU-frequency changes during edge-case occurrences. To identify the userspace operations in which the edge cases occurred, we added KUTrace markers around each CPU-side operation occurring within our GPU-access-duration timing interval; we found that the edge cases occurred within library functions used to communicate with the GPU.



(a) A long `cudaLaunchKernel` call. (b) A long `cudaStreamSynchronize` call.

Figure 5.8: Illustrations of two edge-case scenarios we observed using KUTrace and `nvprof`.

CUDA runtime library overhead. NVIDIA provides the CUDA runtime library for communicating with an NVIDIA GPU. For example, there are CUDA functions to submit a GPU kernel to the GPU (`cudaLaunchKernel`) and await its completion (`cudaStreamSynchronize`).

By aligning a GPU trace using NVIDIA's `nvprof` tracing tool with our KUTrace results, we observed two unexpected scenarios, shown in Figure 5.8. Figure 5.8a illustrates a scenario in which the asynchronous `cudaLaunchKernel` took multiple milliseconds; this call typically takes tens of microseconds. In the other scenario, depicted in Figure 5.8b, the `cudaStreamSynchronize` call did not return until multiple milliseconds after the GPU execution had completed.[4]

Unfortunately, due to the closed-source nature of the NVIDIA ecosystem, discovering the root cause of these edge cases was exceedingly difficult. Furthermore, we may have observed only one of many possible edge cases in using such black-box GPUs. Thus, *having to deal with edge cases is an unavoidable consequence of using an NVIDIA GPU*. Consequently, to support a robust real-time system, edge-case mitigation is a necessity.

5.3.2 Mitigating Edge Cases through Budget Enforcement

The edge cases just discussed highlight our dual need to ensure that our provisioned GPU-related execution times are both respected and reasonable. The classic way of providing such assurance is by enforcing *budgets*, and we do that here. However, in the case of GPU operations, budget enforcement is trickier than for CPU-only tasks.

Our budget-enforcement solution uses a two-pronged approach. To enforce forbidden zones we utilize a watchdog timer, and provision forbidden zones using worst-case GPU-measured access durations. To handle

[4]We use the CUDA option `cudaDeviceScheduleYield` to suspend on the CPU while waiting for the GPU operation to complete, but we observed the same results by instead spinning via `cudaDeviceScheduleSpin`.

budget overruns, we monitor GPU access durations and cancel any additional job processing if a budget overrun occurs. We provision per-access budgets using 99.95^{th}-percentile CPU-measured access durations.

5.3.2.1 Enforcing Forbidden Zones

The watchdog timer is controlled via a pair of syscalls (as part of the forbidden-zone-aware global OMLP) for each accelerator access. Note that, like lock/unlock calls, these sycalls are performed by task code, and therefore forbidden-zone enforcement does rely on an unenforced programming convention.

The first syscall takes two parameters: the access budget and the forbidden-zone length. If the time remaining in the component's time slice is less than the budget, the job is suspended until its component's next time slice. Otherwise, the timer is set to fire at the start of the forbidden zone (*i.e.*, the time-slice end minus the forbidden-zone length), and the budget expiration time is set. The second syscall takes no parameters—it simply cancels the timer (if it has not already fired) after the operation launch has completed; a GPU operation initiated before its forbidden zone begins will complete before the time-slice boundary.

Example 5.5. Our budget-enforcement mechanisms are illustrated for a single GPU access in Figure 5.9. In each inset, the budget expires at time t_b, the forbidden zone begins at t_{fz}, and the time slice ends at t_{ts}.

The "well-behaved" case is depicted in Figure 5.9a. In this example, the first syscall occurs at time (1); the budget is less than the time remaining in the time slice, so the timer is set to fire at time t_{fz} (calculated as t_{ts} minus the forbidden-zone length), t_b is set (calculated as the current time plus the budget), and the access is allowed to proceed. After the kernel launch completes (time (2)), the watchdog timer is cancelled via the second syscall. As the access begins before t_{fz}, it is allowed to execute within its forbidden zone. Once the access completes, the budget is checked (time (3)); this check occurs before time t_b, so the budget has not expired. ♢

If the timer fires, the timer callback function immediately suspends the job, ensuring that the GPU operation will not be initiated in the current time slice.

Example 5.5 (continued). The scenario depicted in Figure 5.9b corresponds to the edge case in Figure 5.8a. In this scenario, the first syscall sets the watchdog timer, but an edge case occurs during the kernel launch, so the timer fires at time t_{fz} before the second syscall occurs. The job is immediately suspended, so the GPU operation is not submitted in this time slice. ♢

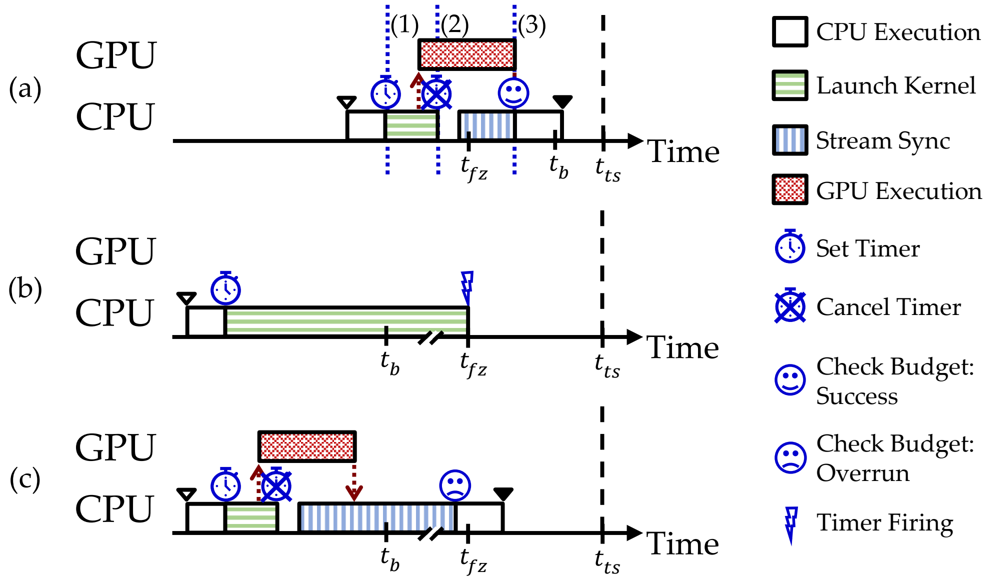


Figure 5.9: The budget-enforcement mechanisms used in TimeWall, for (a) a "well-behaved" GPU access, (b) a GPU access for which the watchdog timer fires, and (c) a GPU access that exceeds its budget.

5.3.2.2 Handling Budget Overruns

Unfortunately, it is not simple to immediately kill a misbehaving GPU-using task, as this invalidates the CUDA context shared by any other tasks in the same process. Thus, we must wait until the `cudaStreamSynchronize` call completes to enforce GPU budgeting. If the access completes after the budget expires, a `SIGSYS` signal sent to the application, which must handle the budget overrun.[5]

Example 5.5 (continued). In Figure 5.9c, the edge case occurs after the GPU operation completes, as in Figure 5.8b, so temporal isolation of the GPU is not violated. However, the `cudaStreamSynchronize` call completes after the access's budget expires at time t_b, so a `SIGSYS` signal is sent to the application. ♢

In our `hog` case study, the `SIGSYS` signal results in stopping all processing for the current video frame, *i.e.*, the frame is dropped—in the AI use cases that motivate this work, occasionally cancelling work (*e.g.*, dropping a video frame) is often deemed as acceptable. The choice of provisioning for GPU-access budgeting provides an interesting trade-off between job completion and system utilization; provisioning GPU-access budgeting on a lower percentile enables better utilization of the processors available to the partition at the cost of a higher number of jobs (*i.e.*, frames) being dropped. We explore this trade-off in Section 5.4.

[5]Note that, like forbidden-zone enforcement, handling of the `SIGSYS` signal relies on unenforced programming conventions.

5.4 Experimental Evaluation

In this section, we present an experimental evaluation of our TimeWall implementation using both synthetic experiments and a case-study featuring the `hog` application. All experiments described in this section were performed on the platform described in Section 5.3.[6]

5.4.1 Temporal Isolation and the Cost of Enforcement

We first discuss synthetic experiments we performed to verify temporal isolation and to quantify the overheads associated with forbidden-zone enforcement.

5.4.1.1 Verifying Temporal Isolation

We designed two GPU-using tasks to test temporal isolation. `GPU-LIGHT`, which accesses the GPU at the start of each time slice, is affected by any GPU interference: the `GPU-LIGHT` kernel executes for a given number of cycles, and if any GPU operations from another component overrun the time-slice boundary, the `GPU-LIGHT` task's response time will increase. `GPU-HEAVY`, on the other hand, submits GPU kernels near the end of each time slice, attempting to cause GPU interference.

We validated that our TimeWall implementation achieves temporal isolation and measured any context-switch costs due to alternating components sharing the GPU. We used two components, each with 16 ms time slices; we executed one `GPU-LIGHT` task in the first component, and measured its response time as we varied the workload in the second component. Our results are shown in Figure 5.10, with the second component containing (a) nothing, (b) one `GPU-LIGHT` task, or one `GPU-HEAVY` task (c) with or (d) without forbidden-zone enforcement. Note that we disabled forbidden-zone enforcement for `GPU-LIGHT` tasks, as each component contains at most one `GPU-LIGHT` task (so `GPU-LIGHT`-containing components have no intra-component GPU contention due) and its kernel completes well before the end of the time slice.

We use the approach of Capodieci *et al.* (2018) to calculate the theoretical context-switch cost based on parameters of our Titan V GPU: with a GPU-internal bus bandwidth of 652.8-GB/s and 64-K 4-byte registers per SM, 128-KB of L1-cache per SM, 4.5-MB of L2 cache, and 80 SMs, it takes around 51.65 μs to store or load the GPU state.

[6]Source code and resulting graphs are available at `https://cs.unc.edu/~anderson/diss/tanyadiss/`.

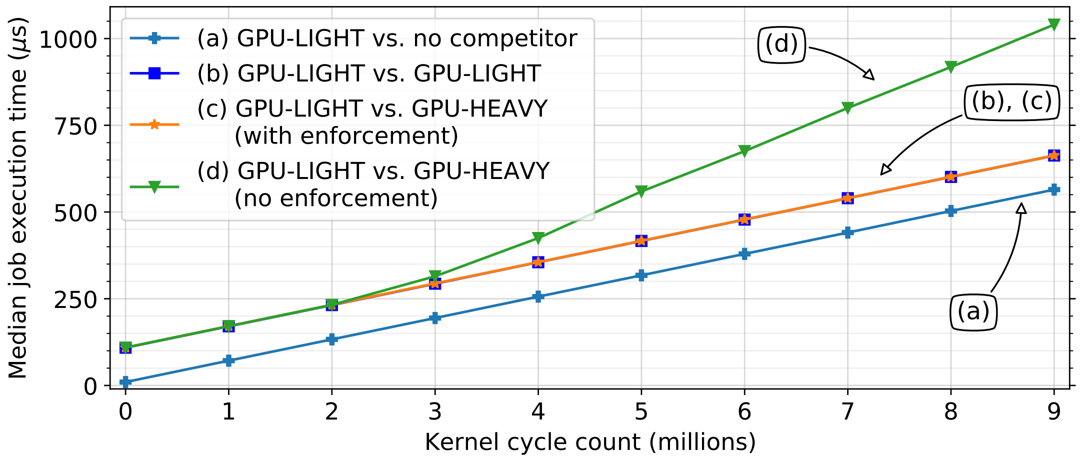


Figure 5.10: Comparison of median `GPU-LIGHT` task execution times in the presence of alternating component workloads (the middle 50th-percentile percentile varied by less than 3%). `GPU-LIGHT` tasks were not subject to forbidden-zone enforcement.

Observation 5.1. Observed GPU-context-switch costs match our theoretical calculation.

Curves (a) and (b) in Figure 5.10 correspond to our `GPU-LIGHT` task running in isolation and against another `GPU-LIGHT` task, respectively; in neither case should GPU operations cross time-slice boundaries. Demonstrating this, the slopes of these curves are nearly identical, with a near-constant offset ranging from 98.2 to 99.2 μs. A context switch requires both a store and a load, resulting in a theoretical cost of about 103.3 μs, which is in line with our measurements.

Observation 5.2. TimeWall enforces temporal isolation between GPU-using components.

The difference between curves (c) and (d) in Figure 5.10 indicates the benefit of our watchdog timer. Additionally, as curves (b) and (c) are nearly identical, we can also observe that our forbidden-zone enforcement reduces any temporal interference to just the cost of GPU context switches.

5.4.1.2 Overhead of Forbidden-Zone Enforcement

We used FeatherTrace (Brandenburg and Anderson, 2007) to measure the overheads[7] associated with enforcing forbidden zones for our TimeWall implementation. We performed a synthetic experiment with 30

[7]We used 99th-percentile measurements. LITMUS$^{\mathrm{RT}}$ was developed for academic research purposes to investigate functionality that *could be* fielded in an RTOS for safety-critical contexts. We take the 99th-percentile as representative of achievable worst-case overheads in a well-honed RTOS.

GPU-using tasks in a component with $M = 5$ CPU cores; the overhead for each GPU access was 0.9 μs to set the watchdog timer, and an additional 1.2 μs if the timer fired.

To consider the trade-off associated with merging multiple GPU accesses into one critical section, we compare the timer overheads to the lock and unlock calls, for which we observed overhead costs of 0.7 μs and 4.7 μs, respectively. (The high unlock overhead is expected due to ensuring priority inheritance for the next lock holder.) These measurements suggest potential benefits of merging a few accesses into one critical section; *e.g.*, two individual requests incur a total of 12.6 μs overhead ($0.7+0.9+4.7 = 6.3$ μs each), but one request that comprises two accesses incurs only $0.7+0.9+0.9+4.7 = 7.2$ μs overhead.

5.4.2 Choosing a Time-Slice Length

The overhead and GPU context-switch costs discussed in Section 5.4.1 become increasingly relevant for shorter time slices. However, long time slices can result in high response times due to long intervals when jobs are released but not scheduled, *e.g.*, if $\Theta = 100$ ms, $\Pi = 200$ ms, and $T_i = 25$ ms for some task τ_i, then four jobs of τ_i may be released but not scheduled during each time interval that τ_i's component does not execute. To explore the "sweet spot" between these two time-slice-length extremes, we performed experiments using randomly generated synthetic tasks.

Components and partitions. Our time-slice experiments used two components, scheduled as shown in Figure 5.11. Components A and B alternately utilized CPUs 1–7 and the GPU. We reserved CPU 0 for interrupt handling, and the other socket (CPUs 8–15) for non-real-time tasks. We used Intel's Resource Directory Technology (via the `resctl` command) to partition the L3 cache and DRAM between components.

Component A was comprised of GPU-using tasks with $C_i = 4$ ms; each job spun on the CPU for its WCET, and accessed the GPU once at a random point in its execution, with $Y_i^{1,1}$ uniformly chosen from $[0.02, 0.04]$ ms ("short"), $[0.2, 0.4]$ ms ("medium"), or $[2, 4]$ ms ("long"). For each of these three access-duration ranges, we generated a task set for Component A by adding randomly generated tasks until we reached utilization restrictions.[8] In Component B, we executed a cache-thrashing workload designed to evict the contents of the per-core L1 and L2 caches.

We varied Θ and Π while maintaining the ratio $\Theta/\Pi = 0.5$, and measured the response times of the tasks in Component A for two minutes. We set Θ to be powers of two from 0.5 ms to 256 ms. To observe

[8]The feasibility conditions for Ω can be shown to be $U \leq \Theta/\Pi \cdot M$ and $u_i \leq \Theta/\Pi \cdot P_i$, assuming per-task WCETs have been inflated to account for the blocking given by Lemma 5.1.

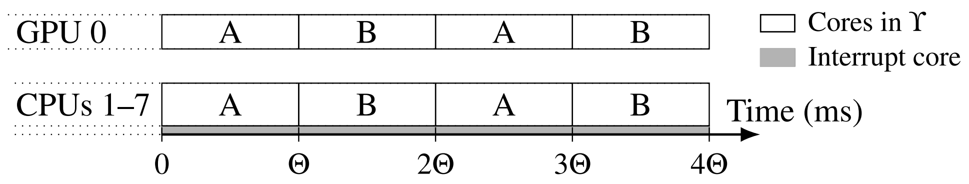


Figure 5.11: Reservations for the two components in our time-slice experiments. We measured response times of tasks in Component A.

the impact of time-slice-aligned job releases, we separately performed experiments in which periods of tasks in Component A were uniformly chosen to be either 32 ms or 64 ms (“aligned”) or 25 ms or 50 ms (“unaligned”). We assume periodic tasks for these experiments, so releases being aligned with time slices means that for $\Pi \geq T_i$, a job is released at the start of each time slice; it is possible that multiple jobs may be released during a time slice, *e.g.*, if $\Theta = 64$ ms and $T_i = 32$ ms.

Job releases aligned with time slices. The results of our aligned-releases experiments are shown in Figure 5.12 for each range of GPU-access durations. Note that for a task system to be schedulable, we require $\Theta \geq Y_i^{k,\ell}$, so we do not include response-time curves for long GPU-access durations (up to 4 ms) with $\Theta < 4$ ms.

Observation 5.3. Extremely short time slices result in unbounded response times.

This can be observed for the 0.5 ms curve in Figure 5.12a. For such short time slices, forbidden zones, scheduling and locking overhead, and context-switch costs take a larger proportion of the capacity available to the component. This is even more pronounced for the medium GPU-access durations in Figure 5.12b; forbidden zones take up to 80% of the 0.5 ms time slices.

Note that although $\Theta = 1$ ms resulted in long response times for short GPU accesses, the overhead was not high enough to cause unbounded response times. Additionally, the higher response times for $\Theta = 1$ ms with short GPU accesses compared to medium GPU accesses was likely due to having more tasks in Component A.

Observation 5.4. Extremely long time slices result in prohibitively large response times.

For $\Theta \geq 64$ ms and $T_i = 32$ or 64 ms, multiple jobs may be released while the component is not scheduled. This results in high response times in all scenarios depicted in Figure 5.12.

Observation 5.5. The lowest response times occur for $\Theta > C_i$ and $\Pi \leq T_i$.

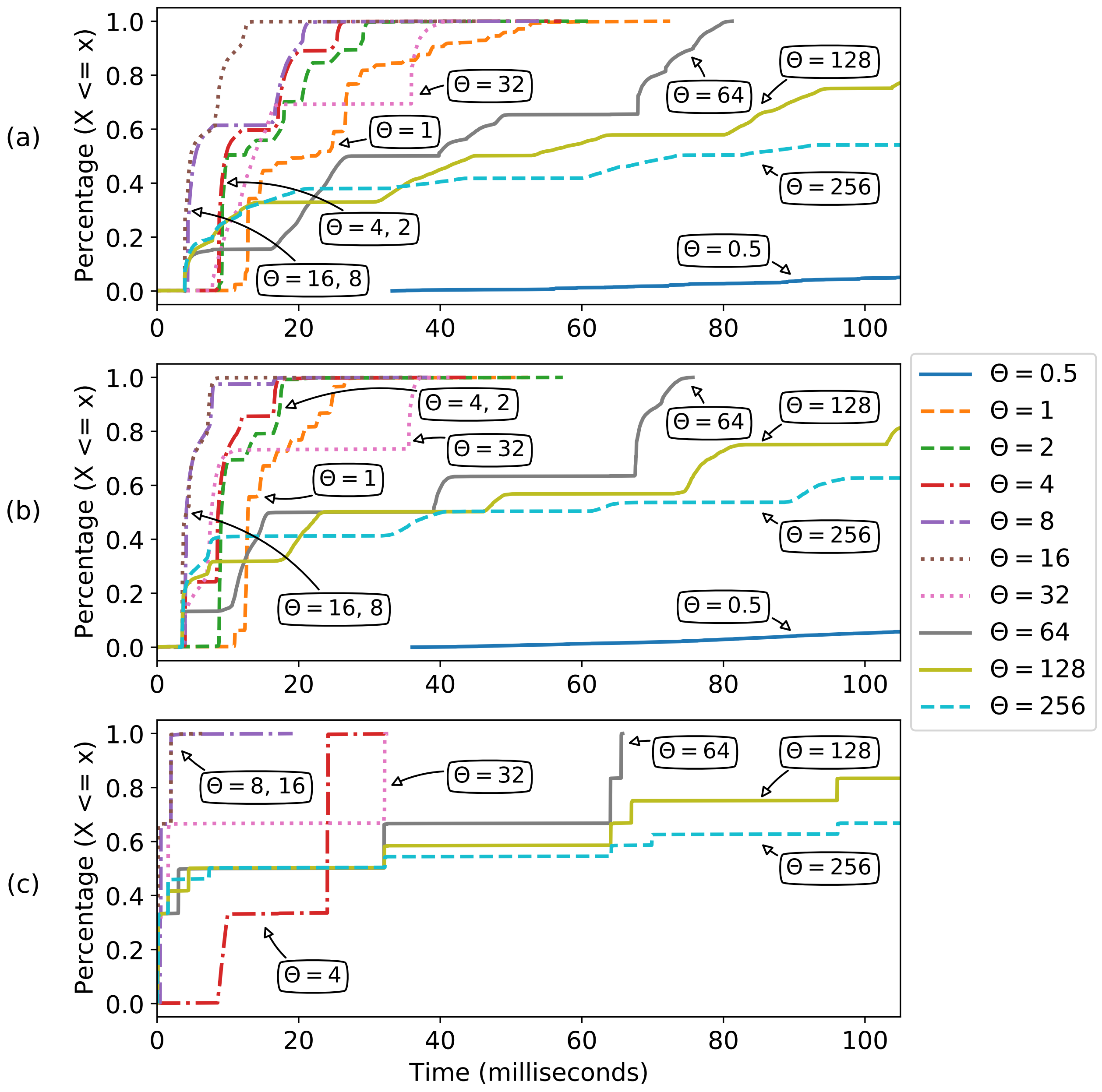


Figure 5.12: Response times of jobs with releases aligned to time slices, for (a) short, (b) medium, and (c) long GPU accesses, with varying Θ (in milliseconds).

In all three scenarios, the lowest response times occurred for $\Theta = 16$ ms ($\Pi = 32$ ms) and $\Theta = 8$ ms ($\Pi = 16$ ms). Thus, the best reservation choices had budgets greater than the WCET of any task and periods at most that of any task.

Job releases unaligned with time slices. The results of our unaligned-releases experiments are shown in Figure 5.13.

Observation 5.6. The lowest response times occur for the same Θ choices regardless of whether job releases align with time slices.

In these experiments, we observed that $\Theta = 8$ ms also resulted in the lowest response times for tasks with unaligned releases. This indicates that these time-slice budgets are not artifacts of aligned job releases, but rather, for our task system parameters, provide the best balance of the negative effects of overhead and fz-blocking with the amount of time a component is not scheduled.

5.4.3 Case-Study Evaluation

We now present our case-study evaluation of TimeWall using a real workload.

Components and partitions. Our case study used the same partitions depicted in Figure 5.11, with different workloads running in Components A and B. We again partitioned the DRAM and the L3 cache between components. Component A was comprised of three `hog` tasks, each with $T_i = 40$ ms (corresponding to 25 frames per second) and $P_i = 2$. Component B contained seven cache-thrashing and GPU-using tasks designed to evict the contents of the per-core L1 and L2 caches, cause GPU context switches, and stress the watchdog timer. These tasks had a period of 160 ms and each job accessed the GPU for 5 ms.

The HOG-based pedestrian-detection algorithm. As discussed in Section 5.3, the HOG-based pedestrian-detection algorithm processes a video frame by copying the image to the GPU, detecting pedestrians at multiple image-scale levels (we used 13 levels in our experiments), and then copying the results for each level back from the GPU. Thus, processing each frame takes 78 GPU operations: one copy-in, 64 kernels (kernels K2–K5 operate on all scale levels, and K1 operates on all but the first), and 13 copy-out operations. We configured each `hog` task to process one frame per job.

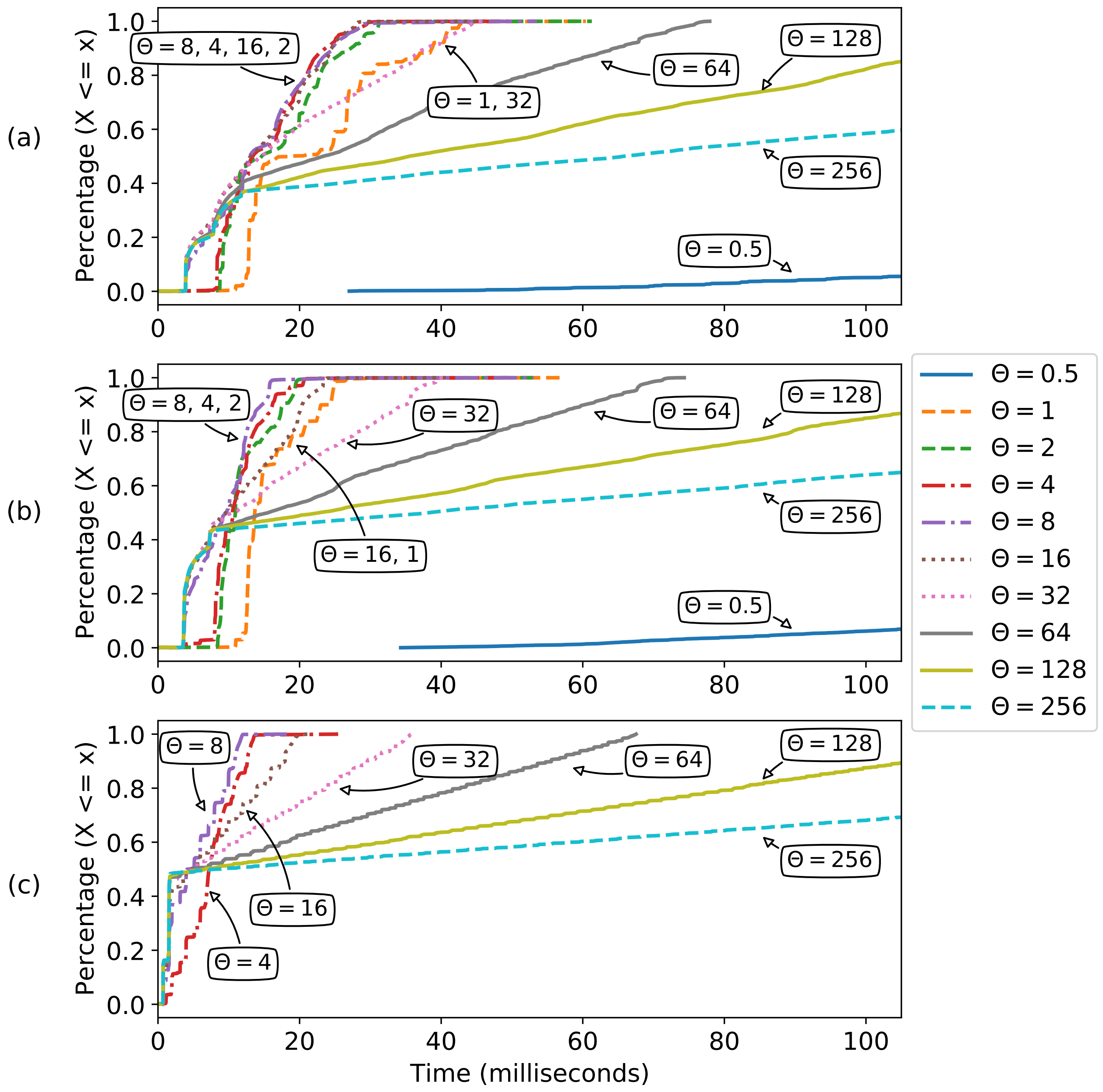


Figure 5.13: Response times of jobs with releases unaligned to time slices, for (a) short, (b) medium, and (c) long GPU accesses, with varying Θ (in milliseconds).

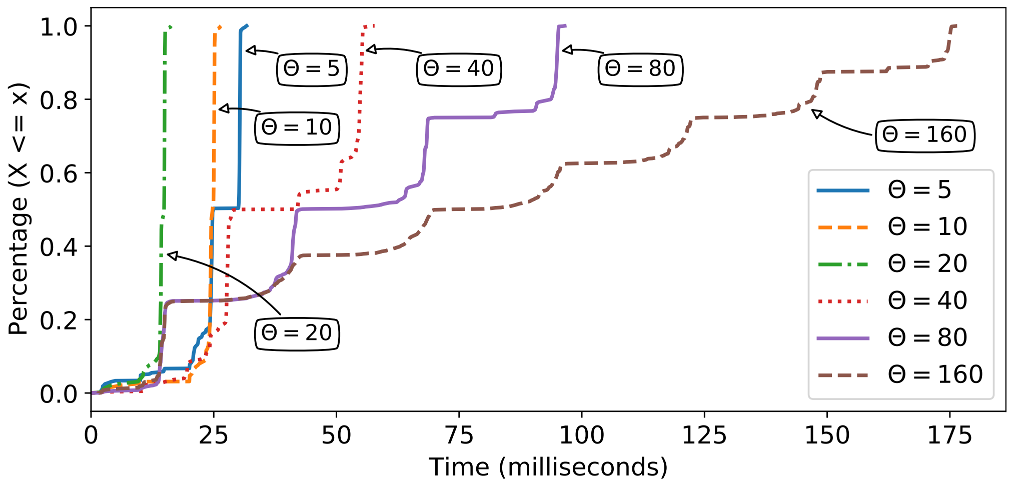


Figure 5.14: CDFs of hog response times with varying Θ (in milliseconds).

5.4.3.1 Choosing Time-Slice Lengths for hog

We performed experiments similar to those in Figure 5.12 for the hog tasks in Component A, with each instance processing 5,000 video frames. We aligned job releases with time slices (in practice, real-time systems typically have harmonic periods), and chose Θ values ranging from 5 ms to 160 ms. The results of our hog time-slice experiments are depicted in Figure 5.14.

Observation 5.7. The lowest response times for hog occur for $\Theta = 20$ ms.

This matches what we expect from Observation 5.5. The hog tasks have a period of 40 ms, so for $\Theta = 20$ ms, each task releases a job each time slice of Component A. The worst-case response time we observed was 16.31 ms, which corresponds to completing a job within the time slice in which it was released. All remaining experiments discussed in this section thus use $\Theta = 20$ ms.

5.4.3.2 Frame Dropping Due to GPU-Budget Enforcement

The provisioning choice for budgeting GPU accesses greatly impacts the number of frames dropped. If any of hog's 78 GPU accesses exceeds its budget, the frame is dropped. We treat each access as an independent random variable with probability ρ of not exceeding its budget. Thus, ρ^{78} is the probability that all 78 accesses for a given frame will not exceed their budgets, and the probability of dropping a frame in hog is given by $1-\rho^{78}$. We choose a value of ρ to provision GPU accesses in hog as described below.

This expected frame-drop rate of $1-\rho^{78}$ is plotted as the "Theoretical" curve in Figure 5.15. The other curves correspond to the observed frame-drop rates for different values of a safety-margin multiplier we apply

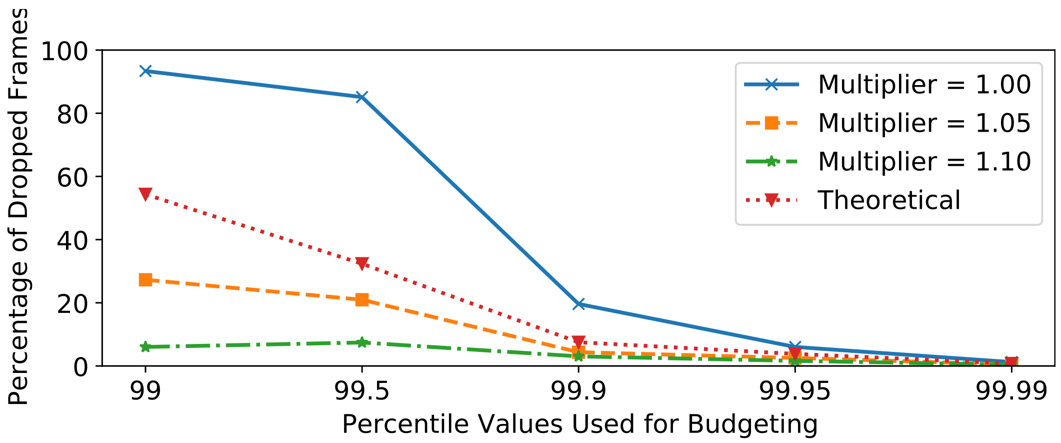


Figure 5.15: Measured frame-drop rates for `hog` depending on multiplier used for provisioned GPU-access budgeting values, compared to the theoretical frame-drop rate.

to CPU-based measurements of GPU access times for provisioning budgets. To measure these frame-drop rates, we provisioned GPU-access budgeting using these different multipliers, and ran three `hog` tasks for 25,000 jobs each, counting the number of frames dropped.

Observation 5.8. Theoretical frame drop rates increase rapidly when provisioning on less than the 99.9th percentile.

For 99.9th-percentile measurements, the theoretical frame-drop rate is 7.5%. However, this rises to 32.4% and 54.3% for the 99.5th- and 99th-percentile measurements, respectively.

Observation 5.9. Using a multiplier of 1.0 results in highly conservative budgeting.

This can be seen by comparing the 1.0-multiplier and "Theoretical" curves in Figure 5.15. For example, provisioning using exactly the 99.9th percentile results in almost a three-times increase in the frame-drop rate over the expected value.

Observation 5.10. Increasing budget provisioning by a small multiplier results in a sizeable reduction in frame-drop rates.

Provisioning on the 99.95th percentile with multiplier 1.1 resulted in a 1.6% frame-drop rate, compared to the expected rate of 3.8%. By using a slightly higher multiplier, we can avoid the outliers occurring for worst-case GPU-access measurements by using a lower percentile, enabling better platform utilization. Thus, for the remaining `hog` experiments in this subsection, we chose a multiplier of 1.1 and to provision budgeting at the 99.95th percentile.

5.4.3.3 The Cost of Enforcing Time Partitioning

Next, we sought to understand the trade-offs associated with using TimeWall with a real workload. For this, we measured the response times of the `hog` tasks in Component A with no workload in Component B (*i.e.*, during Component B's time slices, the CPUs and GPU were idle), and varied the method of enforcing temporal isolation between GPU-using components. We compared no GPU management (and thus no guarantee of isolation), employing the global OMLP (but without support for forbidden zones), and using TimeWall (including forbidden-zone enforcement and CPU budgeting) to ensure temporal isolation.

To also consider the effects of interference between `hog` tasks, we performed experiments with three or one `hog` task executing in Component A. Our results are shown in Figure 5.16. To more clearly observe the worst-case behavior, we additionally plot the complementary CDF (CCDF), or tail distribution, in Figure 5.17.

Observation 5.11. Using TimeWall results in only a slight increase in average-case response times.

This difference is visible for three `hog` tasks in Figure 5.16a. This increase is most likely the result of enforcing forbidden zones. When only the global OMLP is employed, accesses are allowed to begin within the forbidden zone, which can result in temporal-isolation violations, but increases the amount of work that may complete within a given time slice.

Observation 5.12. Using TimeWall results in a lower worst-case response time than using the global OMLP.

The worst-case response times for `hog` using either the global OMLP or TimeWall were 42.9 ms and 19.4 ms, respectively, as shown in Figure 5.17a. This difference is likely due to GPU-access budgeting employed by TimeWall, which terminates a job when it experiences a GPU-access timing anomaly. Using just the global OMLP allows for work to continue after a timing anomaly, which could extend a job's execution long enough to cross a time-slice boundary (recall that we chose $\Theta = 20$ ms).

Observation 5.13. When GPU accesses are not managed, GPU contention between multiple `hog` tasks can greatly increase response times.

This can be observed by comparing the "No GPU management" curves in Figures 5.16 and 5.17. In our experiments, each `hog` task was executed as a separate process in Linux with its own CUDA context. When only a single `hog` task was present in Component A and GPU accesses were not managed (as in insets (b)), the average and worst-case response times were 3.9 ms and 11.0 ms, respectively. However, when executing three `hog` tasks without using any locking protocol (as in insets (a)), the average and worst-case response

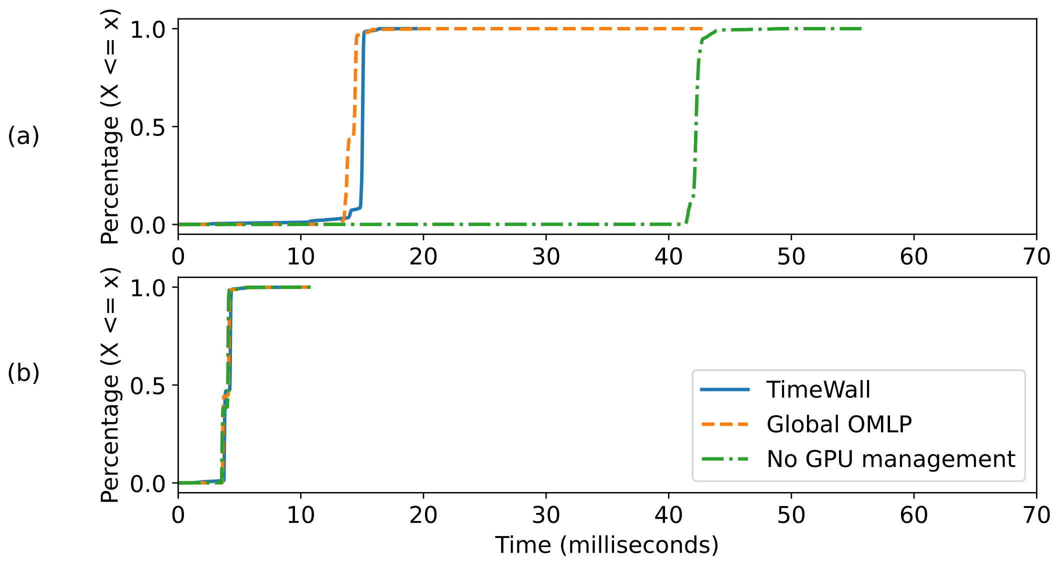


Figure 5.16: CDFs of `hog` response times with different GPU-management approaches, with either (a) three or (b) one `hog` task(s) in Component A.

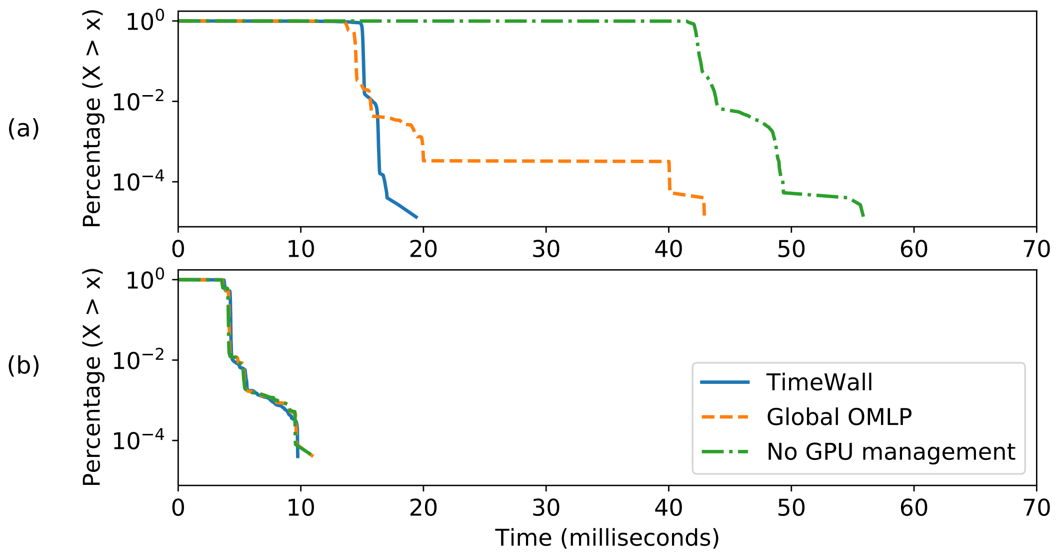


Figure 5.17: CCDFs (tail distributions) of `hog` response times with different GPU-management approaches, with either (a) three or (b) one `hog` task(s) in Component A. Note that the y-axis uses a logarithmic scale.

times were 42.3 ms and 55.9 ms, respectively; almost every job took more than a single time slice, and thus was delayed an additional 20 ms while Component A was not active. This increase in response times is most likely due to multiprogramming effects on the GPU (Amert et al., 2017),[9] which were not present when only one `hog` task executed in Component A.

[9]Multiprogramming experiments were performed by a coauthor, Nathan Otterness, and thus were not presented as a contribution in Chapter 4.

Table 5.2: Observed response times across 75,000 frames for `hog` tasks in Component A without or with other tasks executing in Component B.

	A	A+B
Observed Maximum Response Time (ms)	19.40	16.54
Observed Average Response Time (ms)	14.81	14.61
Number of Dropped Frames	1212	1238

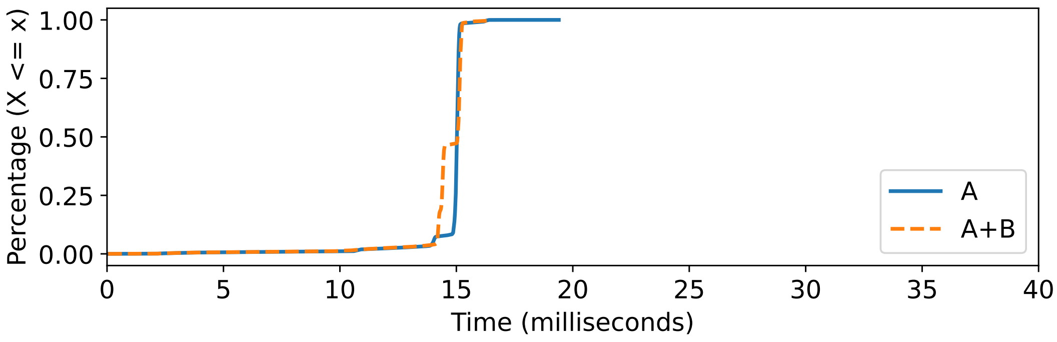


Figure 5.18: CDFs of observed response times for the `hog` tasks in Component A without and with a workload in Component B.

5.4.3.4 Putting It All Together

We executed the three `hog` tasks in Component A for a total of 75,000 frames (25,000 each), either alone or alongside our cache-thrashing and GPU-using workload in Component B, and measured the observed response time for each frame and the overall number of frames dropped. These values are listed in Table 5.2. We also plot the cumulative distribution function of the observed response times in Figure 5.18.

Observation 5.14. The `hog` tasks in Component A were not significantly affected by the presence of tasks in Component B.

This is supported by comparing the observed-response-time curves in Table 5.2 and rows in Figure 5.18, as well as the number of frames dropped in Table 5.2.

Observation 5.15. The frame-drop rate of `hog` matched the expected rate for our budget provisioning choice.

The frame-drop rates given in Table 5.2 were 1.6% and 1.7% when only Component A executed work and when both components executed work, respectively. This matches our expectation of 1.6% given our provisioned GPU-access budgets.

5.5 Chapter Summary

In this chapter, we have presented TimeWall, a time-partitioning approach for real-time systems deployed on multicore+accelerator platforms. TimeWall utilizes a modified multiprocessor locking protocol to maintain the invariant that each time-partitioned component always has exclusive access to all processing resources allocated to it. We discussed the specific challenges associated with realizing a full scheduler implementation of TimeWall for a multicore+GPU platform, and demonstrated the isolation properties afforded by TimeWall via experiments on actual hardware involving a synthetic tasks as well as a computer-vision application. In Chapter 6, we will detail the results of additional experiments that explore the impacts of merging accelerator accesses in the presence of additional automotive-based workloads.

Acknowledgements. The work presented in this chapter is the result of a collaboration among Tanya Amert, Sergey Voronov, Zelin (Peter) Tong, and Joshua Bakita. Amert and Voronov contributed the initial design of TimeWall presented in Section 5.2 and Tong and Bakita performed the bulk of the implementation. In particular, Voronov provided response-time analysis (not presented in this dissertation) for time-sliced scheduling, Voronov and Amert provided the fz-blocking analysis in Section 5.2.3, Tong and Bakita implemented the hierarchical scheduler within LITMUS^{RT} and performed the edge-case explorations discussed in Section 5.3, and Amert and Tong implemented the forbidden-zone-aware variant of the global OMLP. The experiments presented in Section 5.4 were designed collaboratively by Amert and Tong, and performed by Tong.

CHAPTER 6: ADDITIONAL EVALUATIONS

In this chapter, we evaluate the contributions from prior chapters in the presence of real automotive-based workloads. After introducing the workloads we consider in Section 6.1, we return to the history-versus-response-time-versus-accuracy trade-off in Section 6.2, in which we measure the observed history distributions when running our vehicle- and pedestrian-detector-based MOT applications as real-time tasks alongside other work and varying the MOT intra-task parallelism. We then discuss accuracy, precision, and observed response times as parallelism changes. In Section 6.3, we present an extended case-study evaluation of TimeWall. In addition to executing the `hog` application alongside the competing automotive workloads in other components, we also explore the impact of merging GPU accesses into single lock requests. Finally, we conclude in Section 6.4.

6.1 Competing Automotive-Based Workloads

Before discussing our additional experimental evaluations, we first describe the competing automotive workloads that we used.[1]

Path planning. Recall from Section 2.1.1 that planning is typically handled at two levels: the global planner determines a high-level path through a graph-based representation of the scene, and a local planner determines finer-grained goals along that path. For global planning, we used an RRT-based planner. We considered two options for local planning, either based on the Pure Pursuit path-tracking algorithm or the Artificial Potential Field (APF) method. For all planners, we used Python implementations provided online (RuslanAgishev, 2021; Sakai et al., 2018), with modifications to configure them as real-time tasks, as discussed below. None of the path-planning applications used the GPU.

Lane detection. Recall the description of a lane-detection pipeline from Section 2.1.2 and Figure 2.4. Following this pipeline, we implemented a lane-detection application in C++ based on the `houghlines` OpenCV sample. Unlike the path-planning applications, the OpenCV `houghlines` sample used the GPU

[1]Source code is available at `https://cs.unc.edu/~anderson/diss/tanyadiss/`.

(and thus was the only source of GPU contention provided by the competing workloads), so we also modified the necessary OpenCV module's implementation to remedy the issues detected by CUPiD$^{\text{RT}}$, as listed in Table 4.10.

Task configurations. We implemented a Python wrapper for a subset of the `liblitmus` library used with LITMUS$^{\text{RT}}$, and configured each competing workload to be a real-time task. We chose task periods inspired by Scenario 2 from Section 3.3.2.1, in which both vehicles and pedestrians are tracked by a camera attached to a vehicle moving at city-driving speeds. Thus, we assume the vehicle travels at around 30 miles per hour, or 44 feet per second. For the global-planner task, we selected a period of 5 seconds (corresponding to re-planning the global path every 220 feet). The local planner determines the more immediate path, so we selected for it a period of 200 milliseconds, corresponding to approximately every 9 feet. We chose a period of 200 milliseconds for lane detection as well. For all tasks, we assume implicit deadlines, *i.e.*, that a task's relative deadline equals its period.

6.2 Case Study: The History-vs.-Response-Time-vs.-Accuracy Trade-Off

The history-versus-response-time experiments in Section 3.2 indicate that increasing P_i past the minimum required to ensure that $P_i \geq u_i$ can significantly reduce response-time bounds. However, the experiments in Section 3.3 showed that the impact on accuracy of increasing P_i depends not only on the absolute value of P_i, but also the distribution of available historical results.

What remains to be seen is what history distributions can occur in practice, and how this impacts the accuracy and response times of the MOT application. We now present the results of a case-study experiment including the three automotive-based applications discussed in Section 6.1 (RRT, Pure Pursuit, and lane detection) running alongside our deep-learning-based pedestrian and vehicle MOT applications, all as real-time tasks under G-EDF scheduling using LITMUS$^{\text{RT}}$.[2]

Experimental setup. We performed our case study on a platform with a four-core 3.50 GHz Intel i5-6600K processor and one NVIDIA GeForce GTX 980 Ti GPU. Each CPU core has a 32-KB L1 instruction cache, a 32-KB L1 data cache, and a 256-KB L2 cache; all four CPU cores share a 6-MB L3 cache.

[2]Source code and experiment scripts are available at `https://cs.unc.edu/~anderson/diss/tanyadiss/`.

The deep-learning-based detector used in our MOT application takes around 2 seconds to process a single image. Therefore, we chose a period of 2 seconds for both the pedestrian- and vehicle-MOT tasks; as a result, the most recent historical results are not always expected to be available.

To process 300 frames, our experiments ran for approximately 600 seconds, during which we executed 300 jobs of each MOT task, around 120 jobs of the global-planner task, and about 3,000 jobs of each of the local-planner and lane-detection tasks. We anticipate that the presence of additional tasks in the system will result in intermittent delays to the MOT tasks, particularly when a job of the global-planner task executes every 5 seconds, during which there will be jobs of five tasks attempting to utilize four CPU cores.

Distribution of available history. To determine the distribution of available history for the MOT tasks, we set P_i to one of $\{2,3,4\}$ and performed tracking of both vehicles and pedestrians for the 300 frames of Scenario 2. The resulting history distributions are given as tuples in Table 6.1, both when no competing work was present as well as when both planners and lane detection were executing alongside the MOT tasks.

Observation 6.1. Competing work increases the response times of MOT tasks.

This is expected, and demonstrated by comparing the "No competing work" to the "With competing work" rows in Table 6.1. In the presence of competing work, the availability of the most recent data (given by the first element in each tuple) decreases, indicating that interference causes more jobs to have response times greater than the MOT tasks' period of $T_i = 2$ seconds.

Observation 6.2. Vehicle tracking is more prone to interference due to other tasks.

Recall from Section 3.3.2.1 that Scenario 2 features six vehicles and four pedestrians to be tracked. Therefore, it is expected that vehicle detection and tracking would take more time, and thus be more affected by interfering workloads. This is indicated by comparing the "Vehicle Tracking" and "Pedestrian Tracking" columns in Table 6.1 in the presence of competing work, especially for $P_i = 2$.

Observation 6.3. Increasing intra-task parallelism can increase the availability of more recent history results.

With $P_i = 2$, the task system may not be able to recover from delays, such as those caused by the execution of the global-planning task every 5 seconds. With increased parallelism, more jobs of a given MOT task may execute concurrently, allowing recovery from such delays. This is demonstrated by the higher availability of the most recent history for $P_i = 3$ and $P_i = 4$ compared to $P_i = 2$ in Table 6.1 in the presence of competing work.

Table 6.1: Distributions of available history results, represented as tuples, for MOT tasks without and with competing workloads present.

	MOT Parallelism	Vehicle Tracking	Pedestrian Tracking
No competing work	$P_i = 2$	$(0.993, 0.007)$	$(0.993, 0.007)$
	$P_i = 3$	$(0.990, 0.010, 0.000)$	$(0.990, 0.010, 0.000)$
	$P_i = 4$	$(0.987, 0.010, 0.003, 0.000)$	$(0.987, 0.013, 0.000, 0.000)$
With competing work	$P_i = 2$	$(0.702, 0.298)$	$(0.903, 0.097)$
	$P_i = 3$	$(0.930, 0.067, 0.003)$	$(0.980, 0.017, 0.003)$
	$P_i = 4$	$(0.963, 0.027, 0.007, 0.003)$	$(0.920, 0.067, 0.010, 0.003)$

Similar to the PMFs in Table 3.2, (x, y) indicates that the result of one and two frames prior are available for x and y proportions of jobs, respectively.

Table 6.2: Results for vehicle and pedestrian tracking in Scenario 2 with MOT tasks in the presence of competing workloads, alongside selected results from Tables 3.5 and 3.6.

Target	Metric	$P_i = 2$	$P_i = 3$	$P_i = 4$	*PMF* 1	*PMF* 2	*PMF* H_1	*PMF* H_4
Vehicle	*A-MOTA*	**0.838**	0.833	0.835	0.833	**0.844**	**0.836**	0.833
	MOTP	0.682	**0.705**	**0.708**	**0.704**	0.689	**0.705**	0.547
Pedestrian	*A-MOTA*	0.697	0.706	0.650	0.697	0.688	0.703	**0.723**
	MOTP	**0.756**	**0.752**	0.736	0.729	**0.745**	**0.751**	0.663

The best result from Tables 3.5 and 3.6 in each row, as well as any within 1% of the best, are shown in bold.

The history-versus-accuracy trade-off, revisited. Given the results discussed in Section 3.3 and the observed history distributions listed in Table 6.1, we expect that the accuracy and precision for $P_i = 4$ should be similar to that of *PMF* 1 and *PMF* H_1, and that the results for $P_i = 2$ should be closer to those of *PMF* 2. To perform this comparison, we measured both *A-MOTA* and *MOTP* when randomly sampling from the distributions given in Table 6.1. The resulting accuracy and precision scores are reported in Table 6.2; several results from Tables 3.5 and 3.6 are also copied here.

Observation 6.4. The observed history distributions result in similar accuracy and precision results to comparable PMFs from Section 3.3.

For $P_i = 2$, we compare to *PMF* 2, which is represented as the tuple $(0.8, 0.2)$. For vehicle tracking, $P_i = 2$ results in *A-MOTA* and *MOTP* within 0.7% and 1.0%, respectively, of those when using *PMF* 2; for pedestrians, we saw even higher accuracy and precision.

For $P_i = 4$, we compare to both *PMF* 1, corresponding to the tuple $(0.9, 0.09, 0.009, 0.001)$, and *PMF* H_1, indicating sequential execution. As expected, vehicle-tracking accuracy and precision are within 0.1% of that

Table 6.3: Observed response times (in seconds) of MOT tasks with varying intra-task parallelism.

MOT Parallelism	Percentile	Vehicle Tracking	Pedestrian Tracking
$P_i = 2$	50^{th}	1.84	1.80
	90^{th}	118.20	2.06
$P_i = 3$	50^{th}	1.80	1.80
	90^{th}	1.90	1.86
$P_i = 4$	50^{th}	1.81	1.82
	90^{th}	1.87	1.93

of sequential execution (*PMF* H_1), if not higher. Additionally, although pedestrian-tracking accuracy is 7.3% lower than that for sequential execution, precision is between that of *PMF* 1 and *PMF* H_1, as expected.

The complete history-versus-response-time-versus-accuracy trade-off. To facilitate a discussion of the entire trade-off in full, we additionally measured the response times of MOT tasks when running alongside competing workloads. Recall from Corollary 3.1 in Section 3.2.1 that response-time bounds do not depend on P_i, but rather on P_{min}; as the RRT task relies on having immediate history (*i.e.*, sequential execution), we have $P_{min} = 1$. Thus, we instead focus on average-case rather than worst-case behavior. The 50^{th}- and 90^{th}-percentile observed response time measurements are given in Table 6.3.

Observation 6.5. Lower P_i reduces the ability of a task to recover from delays.

For $P_i = 2$, the 90^{th}-percentile observed response time measurement in Table 6.3 was significantly higher than the median. The presence of such outliers indicates that the MOT tasks were more susceptible to delays due to the presence of competing workloads. Recall also that there were more vehicles than pedestrians present in the scenario, which increases the tracking effort, and thus may have contributed to the higher measurements for vehicle tracking over pedestrian tracking.

Observation 6.6. Increasing parallelism can improve response times and accuracy.

This observation is supported by the difference between $P_i = 2$ and $P_i = 3$ in Tables 6.2 and 6.3. By increasing parallelism, we observed an increase in precision for tracking both vehicles and pedestrians, and an accuracy drop of only 0.6% for vehicles (and an increase for pedestrians). Furthermore, we observed a minor decrease in response times of both vehicle and pedestrian tracking, with more resilience towards delays due to competing work. Furthermore, our observed accuracy is within 2.3% of that when using one-fourth the frame rate (*i.e.*, *PMF* H_4), and the precision is better.

In this section, we have shown that we can extend our evaluation from Section 3.3 to a case study of real automotive-based workloads, and that we observe similar results. We have also shown that increasing parallelism may have a positive impact on observed response times, in addition to the benefit to worst-case response-time bounds demonstrated in Section 3.2. It is our hope that such work provides a foundation for further exploration of the response-time-versus-accuracy trade-off more broadly within real-time CV-using systems.

6.3 Case Study: Evaluating Time Partitioning for Automotive Applications

In this section, we present an extended case-study of the temporal isolation provided by TimeWall, using the `hog` application alongside other automotive-based workloads.[3] Specifically, we divide the set of applications into two GPU-using perception-based components (one executing `hog`, the other executing lane detection) and one motion-planning component (with both the RRT global planner and the APF local planner). Additionally, we explore the impact of merging accelerator accesses by `hog` into single lock requests, and find that such merging is beneficial for GPUs.

Experimental setup. We used the same experimental platform as in Sections 5.3 and 5.4: a machine with two eight-core 2.10-GHz Intel Xeon Silver 4110 processors and one NVIDIA Titan V GPU. Each CPU core utilizes a 32-KB L1 instruction cache, a 32-KB L1 data cache, and a 1-MB L2 cache; all eight CPU cores on a socket share an 11-MB L3 cache. As before, we disabled hyperthreading and graphics output.

We distributed `hog` and the competing workloads discussed in Section 6.1 into three components, as depicted in Figure 6.1. CPUs 1–4 were used alternately by Components A and B, which contained two instances of `hog` and one instance of the lane detection application, respectively; as both applications make use of the single GPU, they cannot execute concurrently in different components. We executed the two path-planning tasks in Component C on CPUs 5–7. Each `hog` task had a period of 20 ms; the remaining tasks had periods specified in Section 6.1. We distributed the 11-MB L3 cache between components with 4.4-MB available to Component A and 3.3-MB available to each of Components B and C.

The impact of merging accesses on GPU-access durations. For the experiments in this section, we explored the impact on GPU-access times and observed per-frame response times of the `hog` tasks due to

[3] Source code and experiment scripts are available at `https://cs.unc.edu/~anderson/diss/tanyadiss/`.

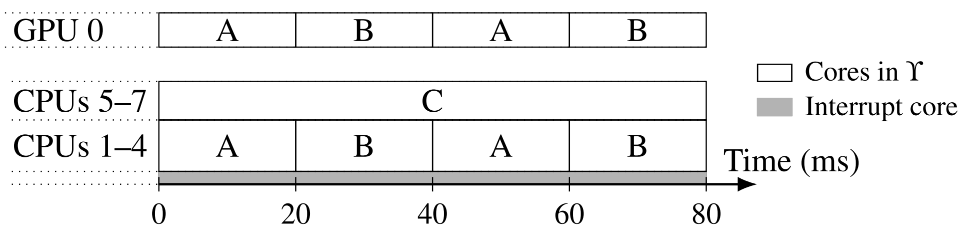


Figure 6.1: Reservations for the three components in our extended evaluation experiments.

merging individual accelerator accesses into a single lock request. We considered the three configurations listed below.

- `MERGE-NONE`: No accesses were merged together. Processing one frame required 78 lock requests.
- `MERGE-MINOR`: For each of the 13 image-scale levels, kernels K3–K5 occurred within a single lock request. Also, every three of the 13 copy-out operations were grouped into lock requests. Thus, processing one frame involved $1 + 13 \times 3 + \lceil 13/3 \rceil = 45$ lock requests.
- `MERGE-MAJOR`: For each scale level, all kernels were grouped into one lock request (K2–K5 for the first level, K1–K5 for the others). Additionally, all 13 copy-out operations were performed within a single lock request. Processing one frame using this configuration required $1 + 13 + 1 = 15$ lock requests.

For each configuration, we executed the two instances of `hog` for 5,000 frames each in Component A with no workloads executing in either of Components B or C. In order to measure the CPU-based access durations, we disabled the watchdog timer and budget checks. The resulting GPU-access-duration timings are given in Table 6.4 for each of the three configurations.

Observation 6.7. Grouping GPU accesses reduces the durations of later accesses within the same lock request.

Comparing `MERGE-NONE` and `MERGE-MINOR` (for which kernels K3–K5 are merged into one lock request), we see that kernels K4 and K5 take significantly less time when they occur immediately following K3 without lock and unlock calls in between. This is even more apparent when considering the median access times. In all configurations, K2 and one of the copy-outs occurs as the first access of a lock request. However, the median access duration for these accesses are greatly reduced by merging, further indicating a potentially significant performance impact due to merging GPU accesses. Possible causes for this effect are frequency

Table 6.4: GPU-access durations (in microseconds) as measured on the CPU for each of the three access-merging configurations we considered.

Configuration	Statistic	Copy-In	K1	K2	K3	K4	K5	Copy-Out
MERGE-NONE	50^{th}	139	120	125	150	123	140	20
	99^{th}	164	145	146	177	145	171	45
	99.5^{th}	170	146	148	182	147	177	46
	99.9^{th}	441	150	152	191	150	182	49
	99.95^{th}	1355	156	157	193	154	186	53
	99.99^{th}	1910	1355	1328	1372	1339	1356	66
	max	5270	5590	3828	5613	5579	5608	1285
MERGE-MINOR	50^{th}	139	120	125	151	24	41	16
	99^{th}	162	146	146	184	34	65	45
	99.5^{th}	168	147	147	186	37	65	47
	99.9^{th}	448	153	152	191	48	73	50
	99.95^{th}	1344	161	157	202	50	80	52
	99.99^{th}	2308	1314	1338	1385	68	134	71
	max	7851	5567	5575	5523	2777	5543	1261
MERGE-MAJOR	50^{th}	143	121	25	46	23	40	15
	99^{th}	164	144	140	73	31	63	46
	99.5^{th}	171	146	145	76	34	64	47
	99.9^{th}	240	153	150	90	48	76	51
	99.95^{th}	476	165	151	92	50	81	53
	99.99^{th}	1865	1350	280	1265	54	1268	61
	max	4863	5539	5512	5535	1266	5504	1263

scaling of NVIDIA GPUs or reduced overhead in the CUDA runtime library or NVIDIA driver due to the clustering of GPU operations; future work is needed to explore these effects more deeply.

The impact of merging accesses on observed response times. The decreased GPU-access durations in Table 6.4 suggest that merging several GPU accesses together, such as in the `MERGE-MAJOR` configuration, should result in lower response times. To verify this, we performed similar experiments to those in Section 5.4.3.4. We executed the two `hog` tasks in Component A for $5,000$ frames each, considering each of the three access-merging configurations. To observe whether we maintain isolation, we additionally varied whether only Component A executed work, or whether lane detection executed in Component B and global and local planning were present in Component C; we refer to a scenario as a choice of access-merging configuration and set of active components. The observed maximum and average response times, as well as number of frames dropped, are reported in Table 6.5, and a CDF of observed response times for each scenario is plotted in Figure 6.2.

Table 6.5: Observed response times across 10,000 frames for `hog` tasks in Component A with varying access-merging configurations and active components.

Configuration	Metric	A	A+B	A+B+C
`MERGE-NONE`	Observed Maximum Response Time (ms)	55.76	55.06	49.23
	Observed Average Response Time (ms)	31.99	25.03	26.29
	Number of Dropped Frames	323	255	262
`MERGE-MINOR`	Observed Maximum Response Time (ms)	44.94	48.04	42.06
	Observed Average Response Time (ms)	14.81	14.84	14.69
	Number of Dropped Frames	240	199	182
`MERGE-MAJOR`	Observed Maximum Response Time (ms)	45.49	42.43	44.57
	Observed Average Response Time (ms)	9.51	9.46	9.51
	Number of Dropped Frames	130	110	110

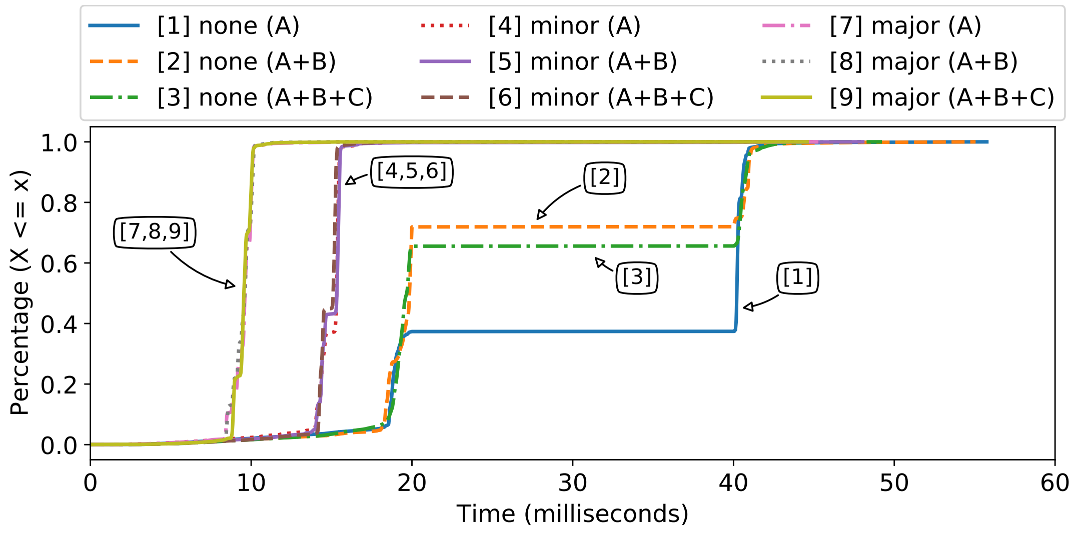


Figure 6.2: CDFs of observed response times for the `hog` tasks in Component A with varying access-merging configurations and active components.

Observation 6.8. Merging GPU accesses improves both worst-case and average-case response times.

This result matches our expectations, and is supported by both Table 6.5 and Figure 6.2, when comparing between the three access-merging configurations.

Observation 6.9. Merging GPU accesses decreases the frame-drop rate.

This can be observed in Table 6.5, and is likely due to the decreased variance in GPU-access durations as the values decrease.

Observation 6.10. Isolation is preserved when using real automotive-based workloads.

The experiments in Section 5.4.3.4 featured `hog` tasks running alongside GPU-stressing work in the alternating component. However, for `MERGE-MINOR` and `MERGE-MAJOR` curves in Figure 6.2, the presence of lane detection and path planning tasks in other components did not have any significant impact on response times of the `hog` tasks running in Component A.

Observation 6.11. Not merging GPU accesses greatly increases observed response times.

The `MERGE-NONE` configuration suffered longer response times due to the longer GPU-access durations listed in Table 6.4. In addition, the per-frame response times were very close to 20 ms, the same as our choice of Θ. Thus, minor variations in job execution times caused some jobs' executions to exceed the time-slice length, resulting in a jump from around 20 ms to around 40 ms (due to a 20 ms time slice of Component B) in response times for curves [1]–[3] in Figure 6.2.

These variations further resulted in only approximately 40% of the jobs completing in one time slice when only Component A executed tasks; this number was more than 60% when either Component B or Components B and C executed tasks. However, it is worth noting that for the fraction of frames in which `hog` response times were at least 40 ms if only Component A executed tasks but less than 20 ms otherwise, the average observed response times were 19.85 ms if Component B also executed tasks, or 19.66 ms if Components B and C also executed tasks. Thus, we expect that these differences would diminish if we considered more data points or used slightly longer time slices.

6.4 Chapter Summary

In this chapter, we have presented additional evaluations considering a set of automotive-based competing workloads, extending the results from prior chapters. For the history-versus-response-time-versus-accuracy trade-off, originally introduced in Chapter 3, we presented a case-study evaluation in which we ran our MOT application as a real-time task alongside other competing work. We measured the observed history-age distribution, and showed that the results we presented in Section 3.3 for synthetically chosen distributions of available history ages apply for real observed distributions,and that increasing parallelism for a given task may improve both accuracy and response times. We used similar competing workloads alongside `hog` tasks to perform an extended case-study evaluation of TimeWall, originally proposed in Chapter 5. We showed that the temporal isolation provided by TimeWall holds in the presence of real workloads executing in several system components, and demonstrated the response-time benefits of merging multiple GPU accesses into a single lock request while still enforcing per-access forbidden zones.

CHAPTER 7: CONCLUSION

Despite the increasing prevalence of ADASs, there is still a vast divide between the design considerations important to CV-application and GPU-hardware developers and the certification standards necessary to guarantee safety for such systems. In this dissertation, we have addressed three challenges that must be met for this design disconnect to be mitigated and ADAS certification to become a reality.

First, we have introduced a task model that can enable response-time analysis to be performed for cycle-containing graph-based workloads, and explored the trade-off that arises between the age of data within the cycle, the response-time bounds of the workload itself, and the accuracy of its resulting computations. Second, we considered the disconnect between GPU manufacturers and their focus on throughput versus the need for predictability when GPUs are used in safety-critical systems. To this end, we detailed the experimentally derived scheduling rules used by NVIDIA GPUs and discussed our approach to detecting the resulting pitfalls that occur in their use. Finally, we introduced a framework to ensure that the various applications needed for an ADAS to operate may be temporally isolated when executing on a single multicore platform, even in the presence of GPUs and other non-preemptive accelerators.

In the following sections, we summarize the results presented in this dissertation (Section 7.1), discuss other contributions not included in this dissertation (Section 7.2), again acknowledge the contributions of the several coauthors who made our results possible (Section 7.3), and conclude with a discussion of several possible future research directions (Section 7.4).

7.1 Summary of Results

In Chapter 1, we presented the following thesis statement:

Enabling the real-time certification of autonomous-driving applications relies on bridging the divide between real-time and computer-vision system design. This includes evaluating the trade-offs between history requirements, accuracy, and response-time bounds of computer-vision *applications, utilizing knowledge of GPU scheduling policies and best practices for using GPUs in real-time applications, and enabling temporal isolation on multicore+accelerator platforms.*

We now summarize the contributions presented in this dissertation in support of this thesis.

The rp-sporadic task model. Existing real-time task models break down in the presence of high-utilization cycles in graph-based task systems. The traditional sequential sporadic task model requires cycle utilization to be at most 1.0. If full intra-task parallelism is allowed, then cycle utilization is not a concern, but precedence constraints may be violated.

In Chapter 3, we introduced the rp-sporadic task model, which adds a per-task parameter P_i that specifies the allowed degree of intra-task parallelism; this parameter is applied for each node in a DAG in which cycles have been replaced by supernodes. If P_i is at most the history age of a given cycle, then precedence constraints are not violated. Furthermore, as long as the utilization of the supernode (and therefore the cycle) is at most P_i, then response-time bounds may be computed (assuming the system is not overutilized).

Evaluation of the history-versus-response-time-versus-accuracy trade-off. The rp-sporadic task model reveals an application-design trade-off, which we also explored in Chapter 3. The utilization of a cycle serves as a lower bound for the necessary intra-task parallelism of the corresponding supernode. However, the application design itself may be able to support higher levels of intra-task parallelism without sacrificing too much accuracy.

We explored the response-time-bound benefits of increasing P_i through a study of synthetically generated graph-based task systems, and found that increasing the minimum P_i in the task system from 2 to 3 or 4 led to a reduction of up to 19.5% and 37.6%, respectively, in relative tardiness.

We evaluated the trade-off between history and accuracy using a multi-object tracking application and the autonomous-driving simulator CARLA. We considered five scenarios, four of which featured city driving and one that involved highway driving. Given ground-truth object detections, we found that tracking accuracy was within 1% of the optimal tracking (assuming sequential execution and thus always having the most recent prior results) as long as the prior results were available 90% of the time, even if the worst-case data age was 4 (corresponding to $P_i = 4$). These results held in almost every scenario, even when using a deep-learning based vehicle and pedestrian detector.

Scheduling rules for NVIDIA GPUs. In Chapter 4, we presented the scheduling rules for NVIDIA GPUs, which we discovered through the use of micro-benchmarking experiments. We validated these rules across multiple NVIDIA GPU architectures (Maxwell, Pascal, and Volta) and CUDA versions (8, 9, 10.2). In addition to basic rules covering user-defined streams, we also explored the impact of using the NULL stream or prioritized streams, as well as the differences for different GPUs, for example due to the presence of additional copy engines.

The rules we detailed in Chapter 4 indicate that scheduler for NVIDIA GPUs has predictable FIFO-oriented properties that are amenable to real-time schedulability analysis if all work is submitted by CPU tasks that share an address space. Such analysis for GPU-using workloads (Yang et al., 2018a) is discussed in the next section. Additionally, the rules we presented have been expanded upon by other researchers, *e.g.*, to explain the mapping between thread-blocks and SMs (Sañudo Olmedo et al., 2020).

Best practices for using NVIDIA GPUs in real-time applications. The GPU scheduling rules we derived also provided the foundation for a study of the possible pitfalls when using NVIDIA GPUs in real-time applications, which we discussed in Chapter 4 as well. These pitfalls include those related to explicit and implicit synchronization as well as missing or contradictory details in the NVIDIA documentation.

We also introduced CUPiD^{RT}, which enables the analysis of a GPU-using program for the presence of issues related to the most egregious of the pitfalls, specifically those that limit concurrency or can cause CPU blocking of other GPU-using tasks. We used CUPiD^{RT} to analyze ten GPU-using sample applications available as part of the CV library OpenCV, and found that all ten applications were subject to the issues we detected. In addition, we explored the benefits of fixing all issues detected by CUPiD^{RT} for one application, and found that our modifications resulted in up to a 74.1% reduction in execution times and a 74.9% reduction in computation-submission times.

A framework for temporal isolation on multicore+accelerator platforms. In Chapter 5, we presented TimeWall, a time-partitioning approach for real-time systems deployed on multicore+accelerator platforms. TimeWall's design was motivated by the need to enable component-wise certification of AI-based embedded applications in safety-critical settings. It utilizes a modified multiprocessor locking protocol to maintain the invariant that each time-partitioned component always has exclusive access to all processing resources allocated to it. We discussed the specific challenges associated with realizing an implementation of TimeWall for a multicore+GPU platform, and demonstrated the isolation properties afforded by TimeWall via experiments on actual hardware involving a synthetic tasks as well as a computer-vision application.

Extended evaluations using autonomous-driving workloads. We extended the contributions presented in prior chapters by presenting the results of extended evaluations in Chapter 6. These evaluations featured several autonomous-driving-related applications, including a global and local path planner and a lane-detection algorithm. We showed that the results presented in Chapter 3 for the history-versus-accuracy trade-off hold when the MOT application is executed as a real-time task. Expanding on the TimeWall framework presented in Chapter 5, we showed that the temporal isolation holds even with real automotive-based workloads, and that merging several accelerator accesses into a single lock request can improve both average- and worst-case observed response times, as well as decreasing the frame-drop rate.

7.2 Other Related Work

In this section, we summarize other research contributions, not presented in this dissertation, of which the author has been a part.

7.2.1 Response-Time Bounds for GPU-Using Workloads[1]

The OpenVX API discussed in Section 2.4.4 leaves several key details to be decided for a given implementation of the specification. One such detail is the available parallelism within a graph. By default, it is assumed that one entire invocation of a graph must finish before the next may begin. This is an extreme example of sequential execution; although multiple nodes in a graph may execute concurrently with one another, such monolithic execution can easily lead to unbounded response times. Prior work by Elliott and Yang *et al*. (Elliott et al., 2015; Yang et al., 2015) considered an alternative in which each OpenVX primitive is treated as a schedulable entity, enabling a coarse-grained scheduling paradigm that reduces response times. However, this coarse-grained scheduling approach does not allow for intra-task parallelism of individual graph nodes.

We presented an approach to reclaim this lost parallelism by allowing full intra-task parllelism for nodes in an OpenVX DAG (Yang et al., 2018a). As a part of allowing such fine-grained scheduling, we subdivide

[1]This work appeared in the following paper:

Yang, M., Amert, T., Yang, K., Otterness, N., Anderson, J. H., Smith, F. D., and Wang, S. (2018a). Making OpenVX really ‘Real Time’. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, pages 80–93.

each OpenVX primitive into a series of CPU and GPU nodes. Making use of the GPU scheduling rules presented in Chapter 4, we additionally proposed response-time analysis for GPU nodes in the graph, enabling response-time bounds to be computed for CPU and GPU nodes separately. We demonstrated the benefits of our approach through a case study involving the `hog` application, in which monolithic and coarse-grained scheduling precluded the computation of response-time bounds, but fine-grained scheduling enabled bounds to be computed and resulted in lower observed response times than either previous approach.

7.2.2 Real-Time Locking Protocols

Several contributions have been made towards real-time locking protocols that enable requests to be made for multiple resources at once. Such *nested* requests can greatly complicate protocols, and easily lead to either high overhead (due to protocol logic), high blocking (due to chains of transitive blocking between requests that share resources), or both. Thus, we sought to develop an approach to reduce overhead, and also to improve blocking for both nested and non-nested requests.

Decreasing locking-protocol overhead.[2] Allowing nested lock requests can greatly increase the amount of program state necessary to execute a locking protocol. As the number of CPUs available on a platform increases, this state becomes an increasingly large factor in the high overhead of such locking protocols. We proposed an approach using lock servers (Nemitz et al., 2018) to execute the lock/unlock logic on specific processors, ensuring that locking protocols execute cache hot. In our experiments, we showed that our approach can reduce lock/unlock overhead by up to 86%.

Decreasing blocking for nested requests.[3] Locking protocols typically approach the concept of mutual exclusion as enforcing the invariant that at most one request may access a resource at a time. However, an alternative approach is to consider which requests may safely be satisfied concurrently. For example, if one

[2]This work appeared in the following paper:

Nemitz, C. E., Amert, T., and Anderson, J. H. (2018). Using lock servers to scale real-time locking protocols: Chasing ever-increasing core counts. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, pages 25:1–25:24.

request must access resources 1 and 2 and another requires resources 3 and 4, such requests do not conflict, and can safely be satisfied together.

This example provides the key intuition behind the Concurrency Group Locking Protocol (CGLP) (Nemitz et al., 2019b, 2021). Given a set of requests, some of which may be for multiple resources, we showed how to transform the problem of determining which requests may be satisfied concurrently into an instance of the Vertex Coloring Problem (VCP). A solution to the VCP maps directly to subsets of requests that may be satisfied concurrently. We provided several linear programs that can be used to optimize for various objectives, *e.g.*, the smallest number of groups or the lowest worst-case blocking. Additionally, we performed a schedulability study in which we showed that our approach outperforms prior approaches that seek to minimize contention-sensitive blocking (*i.e.*, blocking only due to other requests for the same resources).

Decreasing blocking for non-nested requests.[4] Prior approaches that support nested lock requests have done so at the expense of non-nested requests; despite requiring only a single resource, such non-nested requests could still suffer $O(m)$ blocking. Furthermore, prior work has shown that such non-nested requests are by far the common case (Bacon et al., 1998; Brandenburg and Anderson, 2007). To enable contention-sensitive blocking for non-nested requests, we developed the fast RW-RNLP (Nemitz et al., 2017, 2019a), a protocol that provides a fast-path mechanism for such non-nested requests. We performed experiments using synthetically generated task sets in which we showed that our approach resulted in overhead values for non-nested requests that were similar to existing protocols that only support non-nested requests, yet for non-nested requests we achieved significantly lower blocking than existing protocols that support nested requests. We confirmed these improvements via a large-scale schedulability study comparing two alternative approaches that we proposed for arbitrating between nested and non-nested requests.

[3]This work appeared in the following papers:

Nemitz, C. E., Amert, T., Goyal, M., and Anderson, J. H. (2019b). Concurrency groups: A new way to look at real-time multiprocessor lock nesting. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, pages 187–197.

Nemitz, C. E., Amert, T., Goyal, M., and Anderson, J. H. (2021). Concurrency groups: A new way to look at real-time multiprocessor lock nesting. *Real-Time Systems*, 57(1):190–226.

[4]This work appeared in the following papers:

Nemitz, C. E., Amert, T., and Anderson, J. H. (2017). Real-time multiprocessor locks with nesting: Optimizing the common case. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 37–46.

Nemitz, C. E., Amert, T., and Anderson, J. H. (2019a). Real-time multiprocessor locks with nesting: Optimizing the common case. *Real-Time Systems*, 55(2):296–348.

7.3 Acknowledgements

The work presented in this dissertation would not have been possible without the collaborative efforts of many people. The individual contributions are detailed at the end of each of chapters Chapters 3–5, and are summarized again here.

Response-time analysis for the rp-sporadic task model presented in Chapter 3 was developed by Sergey Voronov, as was a preliminary version of the code used to perform the history-versus-response-time trade-off evaluation in Section 3.2. Initial configurations for four of the five scenarios evaluated for the history-versus-accuracy trade-off in Section 3.3 were prepared by Ming Yang. The deep-learning-based detector used in that same evaluation was initially trained by Saujas Nandi.

The GPU scheduling rules and pitfall investigations presented in Chapter 4 relied heavily on the micro-benchmarking framework designed by Nathan Otterness. Although many experiments were a group effort, experiments involving running GPU-using workloads as processes, rather than threads, were performed by Nathan Otterness and Ming Yang (these experiments were not presented in this dissertation, however).

The design of TimeWall detailed in Chapter 5 was done with Sergey Voronov, who also provided the response-time analysis for time-sliced workloads, although this analysis was not presented in this dissertation. The full scheduler implementation, debugging, and final modifications to the forbidden-zone-aware global OMLP implementation were primarily the work of Zelin (Peter) Tong, with guidance and wisdom from Joshua Bakita.

Finally, much of the work presented in this dissertation was published collaboratively with Jim Anderson and Don Smith, without whom the contributions summarized in Section 7.1 truly could not have been accomplished. They both provided significant guidance on system design, experimental choices, and presenting the results in writing.

7.4 Future Work

We now conclude this dissertation by discussing several possible directions to extend the presented work in the future.

Higher-level notions of accuracy in autonomous driving. In Section 3.3, accuracy was assessed using well-established metrics pertaining to CV algorithms, and only done for an individual application. In the future, we intend to extend our experimental efforts to consider higher-level notions of accuracy in autonomous driving, such as missed obstacles and traffic-rule violations. Furthermore, we plan to integrate the usage of relaxed back-history requirements within the control and decision-making components of CARLA, and to re-assess the impact of relaxing such requirements in actual driving scenarios. Given successful operation within CARLA, we then plan to execute our autonomous-driving applications on a real vehicle, such as a 1/10-th scale F1TENTH Autonomous Vehicle System (F1TENTH Foundation, 2021).

Middleware for NVIDIA GPUs. The work presented in Chapter 4 is part of a long-term effort to devise a formal model for GPU computation. In the future, we intend to investigate whether delays due to the pitfalls enumerated in Section 4.4 can be eliminated or lessened by introducing middleware capable of intercepting and reordering or delaying GPU operations. Our hope is that the control afforded by such middleware will enable us to produce reasonable analytical bounds on blocking and response times, while maintaining high GPU utilization wherever possible.

Automatically fixing issues in GPU-using programs. Even with better management, certifiable safety in the face of GPU sharing requires a *guarantee* that pitfalls including blocking due to GPU synchronization are controlled, which is only possible if developers of GPU-using software are aware of the consequences and how to avoid them. In the future, we plan to modify CUPiD$^{\text{RT}}$ to use static analysis rather than relying on NVIDIA's tracing tools. Static analysis will enable us to report which lines of code represent issues, as well as ensure a more comprehensive set of detections that does not rely on runtime behavior (*e.g.*, a program may conditionally allocate or free GPU memory, and runtime behavior may not demonstrate this). We will then target automatically fixing issues, utilizing CUPiD$^{\text{RT}}$ to ensure none remain. This will be a significant effort, and will require static analysis and compiler techniques.

Extending TimeWall to support virtual accelerators. The initial design of TimeWall proposed in Chapter 5 assumes that only one request may access an entire accelerator at a time. However, some accelerators, such as GPUs, can be partitioned into several virtual accelerators (Otterness and Anderson, 2021). We plan to further extend our modified global OMLP, as well as consider other locking-protocol options, in an effort to enable multiple tasks within a component or multiple components to concurrently access the different virtual accelerators comprising a single physical one.

Response-time-guided system decomposition into components. TimeWall serves as a necessary first step towards considering the higher-level question of system and platform decomposition into components and partitions, respectively. In the future, we plan to further examine the higher-level issues that arise when breaking a system into certifiable components, such as how to divide a set of applications into individual components, and what time slices and which accelerators or virtual accelerators to assign to partitions.

BIBLIOGRAPHY

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2016). TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283.

Amert, T. and Anderson, J. H. (2021). CUPiD^{RT}: Detecting improper GPU usage in real-time applications. In *Proceedings of the 24th International Symposium on Real-Time Distributed Computing*, pages 86–95.

Amert, T., Otterness, N., Yang, M., Anderson, J. H., and Smith, F. D. (2017). GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 104–115.

Amert, T., Voronov, S., and Anderson, J. H. (2019). OpenVX and real-time certification: The troublesome history. In *Proceedings of the 40th IEEE Real-Time Systems Symposium*, pages 312–325.

Amert, T., Yang, M., Nandi, S., Vu, T., Anderson, J. H., and Smith, F. D. (2020). The price of schedulability in multi-object tracking: The history-vs.-accuracy trade-off. In *Proceedings of the 23rd IEEE International Symposium on Real-Time Distributed Computing*, pages 124–133.

Anderson, J. H., Bud, V., and Devi, U. C. (2008). An EDF-based restricted-migration scheduling algorithm for multiprocessor soft real-time systems. *Real-Time Systems*, 38(2):85–131.

Andersson, B., Baruah, S., and Jonsson, J. (2001). Static-priority scheduling on multiprocessors. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 193–202.

Åström, K. J. and Murray, R. M. (2021). *Feedback Systems: An Introduction for Scientists and Engineers, 2nd Edition*. Princeton University Press.

Audsley, N. C., Burns, A., Richardson, M. F., and Wellings, A. J. (1991). Hard real-time scheduling: The deadline-monotonic approach. *IFAC Proceedings Volumes*, 24(2):127–132.

Bacon, D. F., Konuru, R., Murthy, C., and Serrano, M. (1998). Thin locks: Featherweight synchronization for java. In *Proceedings of the 19th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268.

Baker, T. P. (2003). Multiprocessor EDF and deadline monotonic schedulability analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 120–129.

Baruah, S. (2007). Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 119–128.

Baruah, S. and Baker, T. (2008a). Global EDF schedulability analysis of arbitrary sporadic task systems. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 3–12.

Baruah, S. and Baker, T. (2008b). Schedulability analysis of global EDF. *Real-Time Systems*, 38(3):223–235.

Behnam, M., Shin, I., Nolte, T., and Nolin, M. (2007). SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *Proceedings of the 7th ACM and IEEE International Conference on Embedded Software*, pages 279–288.

Bertogna, M., Cirinei, M., and Lipari, G. (2005). Improved schedulability analysis of EDF on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 209–218.

Bertogna, M., Cirinei, M., and Lipari, G. (2008). Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on parallel and distributed systems*, 20(4):553–566.

Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

Brandenburg, B. B. (2011). *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA.

Brandenburg, B. B. and Anderson, J. H. (2007). Feather-Trace: A lightweight event tracing toolkit. In *Proceedings of the 3rd International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 20–27.

Brandenburg, B. B. and Anderson, J. H. (2010). Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 49–60.

Brandenburg, B. B. and Anderson, J. H. (2013). The OMLP family of optimal multiprocessor real-time locking protocols. *Design automation for embedded systems*, 17(2):277–342.

Brown, N. (2016). Improvements in CPU frequency management. *LWN.net*.

Brox, T., Bruhn, A., Papenberg, N., and Weickert, J. (2004). High accuracy optical flow estimation based on a theory for warping. In *Proceedings of the 8th European Conference on Computer Vision*, pages 25–36.

Calandrino, J., Leontyev, H., Block, A., Devi, U. C., and Anderson, J. H. (2006). LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 111–126.

Capodieci, N., Cavicchioli, R., Bertogna, M., and Paramakuru, A. (2018). Deadline-based scheduling for GPU with preemption support. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, pages 119–130.

Codevilla, F., Müller, M., López, A., Koltun, V., and Dosovitskiy, A. (2018). End-to-end driving via conditional imitation learning. In *Proceedings of the 35th IEEE International Conference on Robotics and Automation*, pages 4693–4700.

Coulter, R. C. (1992). Implementation of the pure pursuit path tracking algorithm. Technical report, Carnegie Mellon University Robotics Institute.

Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1, pages 886–893.

DanielHfnr (2019 (accessed 10 September 2020)). Carla object detection dataset. `https://github.com/DanielHfnr/Carla-Object-Detection-Dataset`.

Devi, U. (2006). *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA.

Devi, U. and Anderson, J. (2008). Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38:133–189.

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. (2017). CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16.

Elliott, G. A. and Anderson, J. H. (2012). Robust real-time multiprocessor interrupt handling motivated by GPUs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 267–276.

Elliott, G. A., Kim, N., Erickson, J. P., Liu, C., and Anderson, J. H. (2014). Minimizing response times of automotive dataflows on multicore. In *Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

Elliott, G. A., Yang, K., and Anderson, J. H. (2015). Supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the 36th IEEE Real-Time Systems Symposium*, pages 273–284.

Epic Games (2020 (accessed 10 September 2020)). Unreal Engine. `https://www.unrealengine.com`.

Erdős, P. and Rényi, A. (1958). On random graphs I. *Publicationes Mathematicae*, 6:290–297.

Erickson, J. P. and Anderson, J. H. (2011). Response time bounds for G-EDF without intra-task precedence constraints. In *Proceedings of the 15th International Conference On Principles Of Distributed Systems*, pages 128–142.

Erickson, J. P. and Anderson, J. H. (2012). Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 3–11.

Erickson, J. P., Guan, N., and Baruah, S. (2010). Tardiness bounds for global EDF with deadlines different from periods. In *Proceedings of the 14th International Conference On Principles Of Distributed Systems*, pages 286–301.

Everingham, M., Van Gool, L., Williams, C. K., Winn, J., and Zisserman, A. (2010). The Pascal visual object classes (VOC) challenge. *International Journal of Computer Vision*, 88(2):303–338.

F1TENTH Foundation (2021 (accessed 25 June 2021)). F1tenth. `https://f1tenth.org/`.

Farnebäck, G. (2003). Two-frame motion estimation based on polynomial expansion. In *Proceedings of the 13th Scandinavian Conference on Image Analysis*, pages 363–370.

Farsiu, S., Robinson, M. D., Elad, M., and Milanfar, P. (2004). Fast and robust multiframe super resolution. *IEEE transactions on image processing*, 13(10):1327–1344.

Franke, U. (2017). Autonomous driving. *Computer Vision in Vehicle Technology: Land, Sea & Air*, pages 24–54.

Heo, S., Cho, S., Kim, Y., and Kim, H. (2020). Real-time object detection system with multi-path neural networks. In *Proceedings of the 26th Real-Time and Embedded Technology and Applications Symposium*, pages 174–187.

Holman, P. and Anderson, J. H. (2006). Locking under Pfair scheduling. *ACM Transactions on Computer Systems*, 24(2):140–174. (an earlier version appeared at RTSS 2002).

Huang, J., Rathod, V., Sun, C., Zhu, M., Korattikara, A., Fathi, A., Fischer, I., Wojna, Z., Song, Y., Guadarrama, S., et al. (2017). Speed/accuracy trade-offs for modern convolutional object detectors. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7310–7311.

Hui, J. (2018 (accessed 10 September 2020)). mAP (mean Average Precision) for object detection. https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Khatib, O. (1986). Real-time obstacle avoidance for manipulators and mobile robots. In *Autonomous robot vehicles*, pages 396–404. Springer.

LaValle, S. M. and Kuffner Jr, J. J. (2001). Randomized kinodynamic planning. *The international journal of robotics research*, 20(5):378–400.

Leontyev, H. and Anderson, J. H. (2007a). Generalized tardiness bounds for global multiprocessor scheduling. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 413–422.

Leontyev, H. and Anderson, J. H. (2007b). Tardiness bounds for FIFO scheduling on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 71–80.

Leontyev, H. and Anderson, J. H. (2010). Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44(1-3):26–71.

Li, J., Chen, J. J., Agrawal, K., Lu, C., Gill, C., and Saifullah, A. (2014). Analysis of federated and global scheduling for parallel real-time tasks. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, pages 85–96.

Liedtke, J., Hartig, H., and Hohmuth, M. (1997). OS-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, pages 213–224.

Lin, T., Maire, M., Belongie, S. J., Bourdev, L. D., Girshick, R. B., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft COCO: Common objects in context. *CoRR*, abs/1405.0312.

Liu, C. and Anderson, J. H. (2011). Supporting graph-based real-time applications in distributed systems. In *Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 143–152.

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

Lucas, B. D. and Kanade, T. (1981). An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pages 674–679.

Luitjens, J. (2014). CUDA Streams: Best practices and common pitfalls. Online at https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf.

Mei, X., Chu, X., Liu, H., Leung, Y.-W., and Li, Z. (2017). Energy efficient real-time task scheduling on CPU-GPU hybrid clusters. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications*, pages 1–9.

Milan, A., Leal-Taixé, L., Reid, I., Roth, S., and Schindler, K. (2016). MOT16: A benchmark for multi-object tracking. *arXiv preprint arXiv:1603.00831*.

Mitzel, D., Pock, T., Schoenemann, T., and Cremers, D. (2009). Video super resolution using duality based TV-L1 optical flow. In *Proceedings of the 31st Symposium of the German Association for Pattern Recognition*, pages 432–441.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937.

Mok, A. K. (1983). *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA.

Mok, A. K., Feng, X., and Chen, D. (2001). Resource partition for real-time systems. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium*, pages 75–84.

Mueller, F. (1995). Compiler support for software-based cache partitioning. In *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 125–133.

Najm, W. G., Smith, J. D., and Yanagisawa, M. (2007). Pre-crash scenario typology for crash avoidance research. Technical report, United States. National Highway Traffic Safety Administration.

Nemitz, C. E., Amert, T., and Anderson, J. H. (2017). Real-time multiprocessor locks with nesting: Optimizing the common case. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 37–46.

Nemitz, C. E., Amert, T., and Anderson, J. H. (2018). Using lock servers to scale real-time locking protocols: Chasing ever-increasing core counts. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, pages 25:1–25:24.

Nemitz, C. E., Amert, T., and Anderson, J. H. (2019a). Real-time multiprocessor locks with nesting: Optimizing the common case. *Real-Time Systems*, 55(2):296–348.

Nemitz, C. E., Amert, T., Goyal, M., and Anderson, J. H. (2019b). Concurrency groups: A new way to look at real-time multiprocessor lock nesting. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, pages 187–197.

Nemitz, C. E., Amert, T., Goyal, M., and Anderson, J. H. (2021). Concurrency groups: A new way to look at real-time multiprocessor lock nesting. *Real-Time Systems*, 57(1):190–226.

NVIDIA (2019a). Best practices guide. Online at `https://docs.nvidia.com/cuda/archive/10.2/cuda-c-best-practices-guide/index.html`.

NVIDIA (2019b). CUDA C++ programming guide v10.2.89. Online at `https://docs.nvidia.com/cuda/archive/10.2/cuda-c-programming-guide/index.html`.

NVIDIA (2019c). NVIDIA CUDA runtime API. Online at `https://docs.nvidia.com/cuda/archive/10.2/cuda-runtime-api/index.html`.

NVIDIA (2020). Multi-process service. Online at `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`. Version R450.

NVIDIA (2021). CUDA occupancy calculator. `https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html`.

Otterness, N. and Anderson, J. H. (2021). Exploring AMD GPU scheduling details by experimenting with “worst practices”. In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*.

Otterness, N., Miller, V., Yang, M., Anderson, J. H., and Smith, F. D. (2016). GPU sharing for image processing in embedded real-time systems. In *Proceedings of the 12th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 23–39.

Otterness, N., Yang, M., Amert, T., Anderson, J. H., and Smith, F. D. (2017a). Inferring the scheduling policies of an embedded CUDA GPU. In *Proceedings of the 13th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 47–52.

Otterness, N., Yang, M., Rust, S., Park, E., Anderson, J. H., Smith, F. D., Berg, A., and Wang, S. (2017b). An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 353–364.

Paden, B., Čáp, M., Yong, S. Z., Yershov, D., and Frazzoli, E. (2016). A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on intelligent vehicles*, 1(1):33–55.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Pérez, J. S., Meinhardt-Llopis, E., and Facciolo, G. (2013). TV-L1 optical flow estimation. *Image Processing On Line*, 2013:137–150.

Prisacariu, V. and Reid, I. (2009). fastHOG–a real-time GPU implementation of HOG. Technical report, Department of Engineering Science, Oxford University.

Prisaznuk, P. J. (2008). ARINC 653 role in integrated modular avionics (IMA). In *Proceedings of the 27th IEEE/AIAA Digital Avionics Systems Conference*, pages 1–E.

Ren, S., He, K., Girshick, R., and Sun, J. (2015). Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99.

Rezatofighi, H., Tsoi, N., Gwak, J., Sadeghian, A., Reid, I., and Savarese, S. (2019). Generalized intersection over union: A metric and a loss for bounding box regression. In *Proceedings of the 32nd IEEE Conference on Computer Vision and Pattern Recognition*, pages 658–666.

RuslanAgishev (2021 (accessed 14 May 2021)). Motion planning. `https://github.com/RuslanAgishev/motion_planning`.

Rybczyńska, M. (2020). Modernizing the tasklet API. *LWN.net*.

Sañudo Olmedo, I., Capodieci, N., Martinez, J. L., Marongiu, A., and Bertogna, M. (2020). Dissecting the CUDA scheduling hierarchy: A performance and predictability perspective. In *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 213–225.

Sakai, A., Ingram, D., Dinius, J., Chawla, K., Raffin, A., and Paques, A. (2018). PythonRobotics: a Python code collection of robotics algorithms.

Sites, R. (2018). Benchmarking ”hello, world!”: Six different views of the execution of ”hello, world!” show what is often missing in today’s tools. *ACM Queue*, 16(5):54–80.

Sites, R. (2021). *Understanding Software Dynamics*. Addison-Wesley Professional.

Song, J., Hao, C., and Su, J. (2020). Path planning for unmanned surface vehicle based on predictive artificial potential field. *International Journal of Advanced Robotic Systems*, 17(2).

Stiefelhagen, R., Bernardin, K., Bowers, R., Garofolo, J., Mostefa, D., and Soundararajan, P. (2006). The CLEAR 2006 evaluation. In *Proceedings of the 1st International Evaluation Workshop on Classification of Events, Activities and Relationships*, pages 1–44.

The Khronos Group (2014). OpenVX: Portable, Power Efficient Vision Processing. Online at `https://www.khronos.org/openvx/`.

The Khronos Group (2018). The OpenVX Specification. Online at `https://www.khronos.org/registry/OpenVX/specs/1.2.1/OpenVX_Specifi-cation_1_2_1.html#sub_graphs_rules`.

tkortz (2020 (accessed 10 September 2020)). Carla object detection dataset. `https://github.com/tkortz/Carla-Object-Detection-Dataset`.

Waymo (2021). Autonomous driving FAQ. `https://ltad.com/about/self-driving-faq.html`.

Wilhelm, R. (2020). Real time spent on real time. In *Proceedings of the 41st IEEE Real-Time Systems Symposium*, pages 1–2.

Yang, K., Elliott, G. A., and Anderson, J. H. (2015). Analysis for supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the 23th International Conference on Real-Time Networks and Systems*, pages 77–86.

Yang, K., Yang, M., and Anderson, J. H. (2016). Reducing response-time bounds for DAG-based task systems on heterogeneous multicore platforms. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 349–358.

Yang, M., Amert, T., Yang, K., Otterness, N., Anderson, J. H., Smith, F. D., and Wang, S. (2018a). Making OpenVX really ‘Real Time’. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, pages 80–93.

Yang, M., Otterness, N., Amert, T., Bakita, J., Anderson, J. H., and Smith, F. D. (2018b). Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, pages 20:1–20:21.

Yang, M., Wang, S., Bakita, J., Vu, T., Smith, F. D., Anderson, J. H., and Frahm, J.-M. (2019). Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *Proceedings of the 25th Real-Time and Embedded Technology and Applications Symposium*, pages 305–317.

Yao, S., Hao, Y., Zhao, Y., Shao, H., Liu, D., Liu, S., Wang, T., Li, J., and Abdelzaher, T. (2020). Scheduling real-time deep learning services as imprecise computations. In *Proceedings of the 26th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

Zach, C., Pock, T., and Bischof, H. (2007). A duality based approach for realtime TV-L1 optical flow. In *Proceedings of the 29th Symposium of the German Association for Pattern Recognition*, pages 214–223.