CHAPTER 5: GRAPH SCHEDULING AND GPUS[1]

In Chapter 4, we used GPUSync to arbitrate access to GPU resources among tasks that follow the sporadic task model, whereby each real-time task is represented by a single thread of execution. In this chapter, we apply GPUSync towards the graph-based PGM task model described earlier in Section 2.1.3. We begin by providing our motivation for supporting a graph-based task model, which we draw from recent trends in real-world software architectures and application constraints. We then describe PGM$^{\mathrm{RT}}$, a middleware library we developed to support real-time task sets derived from PGM graphs. PGM$^{\mathrm{RT}}$ integrates tightly with the LITMUS$^{\mathrm{RT}}$ kernel (which we also modify to support PGM-derived real-time task sets) to minimize system overheads. However, PGM$^{\mathrm{RT}}$ remains portable to POSIX-compliant operating systems. Next, we discuss the newly developed open standard, OpenVX™, which is designed to support computer vision applications (Khronos Group, 2014c).[2] OpenVX uses a graph-based software architecture designed to enable efficient computation on heterogeneous computing platforms, including those that use accelerators like GPUs. We examine assumptions made by the designers of the OpenVX that conflict with our real-time task model. We then discuss VisionWorks®, an OpenVX implementation by NVIDIA (Brill and Albuz, 2014).[3] With support from NVIDIA, we adapted an alpha-version of VisionWorks to run atop PGM$^{\mathrm{RT}}$, GPUSync, and LITMUS$^{\mathrm{RT}}$. We describe several challenges we faced in this effort, along with our solutions. We then present the results from a runtime evaluation of our modified version of VisionWorks under several configurations of GPUSync. We compare our GPUSync configurations against two purely Linux-based configurations, as well as a LITMUS$^{\mathrm{RT}}$ configuration *without* GPUSync. Our results demonstrate clear benefits from GPUSync. We conclude this chapter with a summary of our efforts and experimental results.

[1] Portions of this chapter previously appeared in conference proceedings. The original citation is as follows:
Elliott, G., Kim, N., Liu, C., and Anderson, J. (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.

[2] OpenVX is a trademark of the Khronos Group Inc.

[3] VisionWorks is a registered trademark of the NVIDIA Corp.

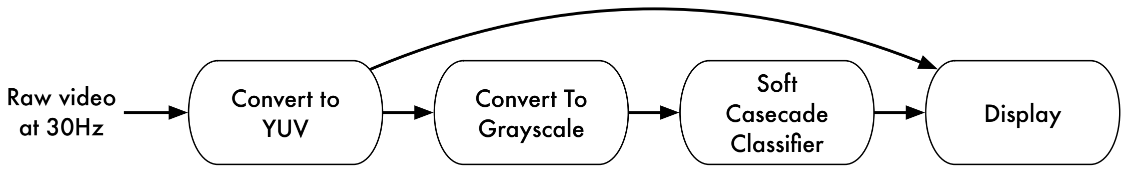


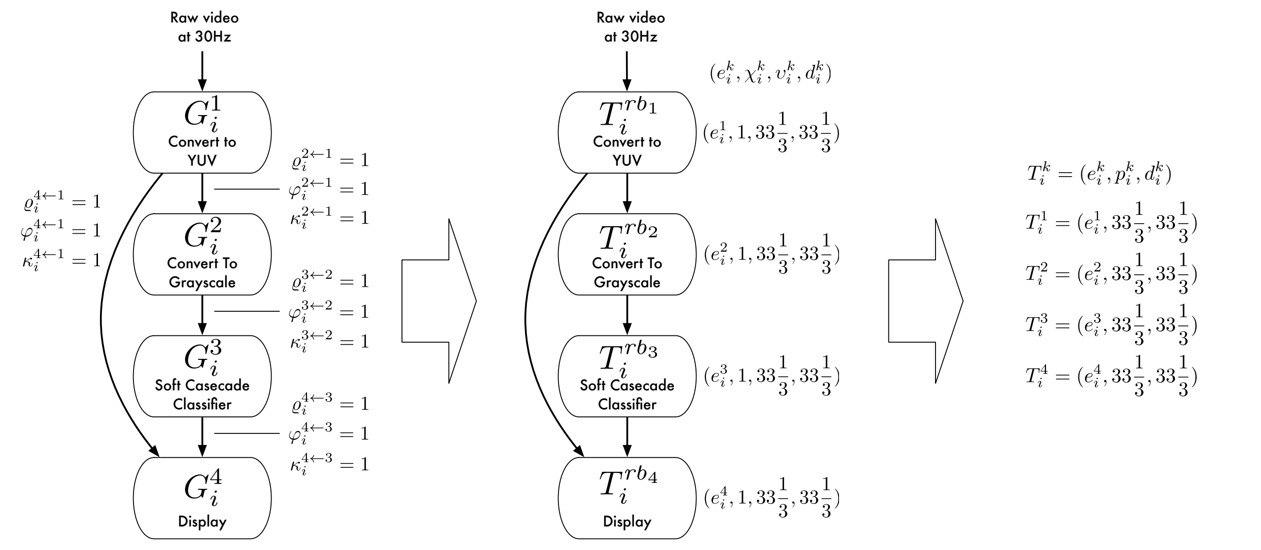
Figure 5.1: Dataflow graph of a simple pedestrian detector application.

5.1 Motivation for Graph-Based Task Models

Graph-based software architectures, often referred to as *dataflow* architectures, are common to software applications that process continual streams of data or events. In such architectures, vertices represent sequential code segments that operate upon data, and edges express the flow of data among vertices. The flexibility offered by such an architecture's inherent modularity promotes code reuse and parallel development. Also, these architectures naturally support concurrency, since parallelism can be explicitly described by the graph structure. These characteristics have made dataflow architectures popular in multimedia technologies (Khronos Group, 2005; Taymans *et al.*, 2013) and the emerging field of computational photography (Adams *et al.*, 2010; NVIDIA, 2013). Dataflow architectures are also prevalent in the sensor-processing components in prototypes of advanced automotive systems, for both driver-assisted and autonomous driving (*e.g.*, Miller *et al.* (2009); Urmson *et al.* (2009); Wei *et al.* (2013)). While many domains with dataflow architectures have timing requirements, the automotive case is set apart since timing violations may result in loss of life or property.

Figure 5.1 depicts a dataflow graph of a simple pedestrian detection application that could be used in an automotive application. We describe these nodes from left to right. A video camera feeds the source of the graph with video frames at $30Hz$ (or $30FPS$). The first node converts raw camera data into the common YUV color image format. The second node extracts the "Y" component of each pixel from the YUV image, producing a grayscale image. (Computer vision algorithms often operate only on grayscale images.) The third node performs pedestrian detection computations and produces a list of the locations of detected pedestrians. In this case, the node uses a common "soft cascade classifier" (Bourdev and Brandt, 2005) to detect pedestrians. Finally, the last node displays an overlay of detected pedestrians over the original color image.

We may shoehorn our pedestrian detection application into a single implicit deadline sporadic task. Here, we give such a task a period and relative deadline of $33\frac{1}{3}ms$ to match the period of the video camera. Each


Raw video at 30Hz
G_i^1 Convert to YUV
G_i^2 Convert To Grayscale
G_i^3 Soft Casecade Classifier
G_i^4 Display
$\varrho_i^{2\leftarrow 1}=1$ $\varphi_i^{2\leftarrow 1}=1$ $\kappa_i^{2\leftarrow 1}=1$
$\varrho_i^{3\leftarrow 2}=1$ $\varphi_i^{3\leftarrow 2}=1$ $\kappa_i^{3\leftarrow 2}=1$
$\varrho_i^{4\leftarrow 3}=1$ $\varphi_i^{4\leftarrow 3}=1$ $\kappa_i^{4\leftarrow 3}=1$
$\varrho_i^{4\leftarrow 1}=1$ $\varphi_i^{4\leftarrow 1}=1$ $\kappa_i^{4\leftarrow 1}=1$
Raw video at 30Hz
$(e_i^k,\chi_i^k,\upsilon_i^k,d_i^k)$
$T_i^{rb_1}$ Convert to YUV $(e_i^1,1,33\frac{1}{3},33\frac{1}{3})$
$T_i^{rb_2}$ Convert To Grayscale $(e_i^2,1,33\frac{1}{3},33\frac{1}{3})$
$T_i^{rb_3}$ Soft Casecade Classifier $(e_i^3,1,33\frac{1}{3},33\frac{1}{3})$
$T_i^{rb_4}$ Display $(e_i^4,1,33\frac{1}{3},33\frac{1}{3})$
$T_i^k=(e_i^k,p_i^k,d_i^k)$
$T_i^1=(e_i^1,33\frac{1}{3},33\frac{1}{3})$
$T_i^2=(e_i^2,33\frac{1}{3},33\frac{1}{3})$
$T_i^3=(e_i^3,33\frac{1}{3},33\frac{1}{3})$
$T_i^4=(e_i^4,33\frac{1}{3},33\frac{1}{3})$

(a) PGM representation. (b) Rate-based representation. (c) Sporadic representation.

Figure 5.2: Transformation of a PGM-specified graph for a pedestrian detection application to sporadic tasks.

job of this task executes the dataflow graph, end-to-end, once per job. This technique can be applied to any graph by executing graphs nodes in a topological order. However, this approach prevents us from exploiting parallelism in two ways. First, we cannot exploit parallelism expressed by parallel branches (or forks) in a graph since we serialize all graph nodes. Second, we cannot execute nodes in a pipeline, since the graph is executed end-to-end within each job. There is another significant drawback to this shoehorned approach: the combined execution time of graph nodes may exceed the period of the task. Such a task is intrinsically unschedulable.

In Section 2.1.3, we described a process for transforming a dataflow graph described by PGM into a set of sporadic tasks. Figure 5.2 depicts such a transformation for our pedestrian detection application. We describe this transformation in more detail. Figure 5.2(a) gives a PGM-specification for the pedestrian detection dataflow graph. Here, each non-sink node produces one token ($\varrho_i^{k\leftarrow j}=1$) for each of its consumers. Similarly, each non-source node consumes one token ($\kappa_i^{k\leftarrow j}=1$) from each of its producers, as tokens become available ($\varphi_i^{k\leftarrow j}=1$). Given an input video rate of 30Hz, the rate-based task for the source node, $T_i^{rb_1}$, is released once ($\chi_i^{rb_1}=1$) every $33\frac{1}{3}ms$ ($\upsilon_i^{rb_1}=33\frac{1}{3}ms$). The remaining nodes have the same rate-based specification since $\varrho_i^{k\leftarrow j}=\kappa_i^{k\leftarrow j}$ (see Equations (2.8)–(2.11)). This is depicted in Figure 5.2(b). Finally, Figure 5.2(c) gives the final transformation to implicit-deadline sporadic tasks, where $d_{i,j}^k=p_{i,j}^k=33\frac{1}{3}ms$.

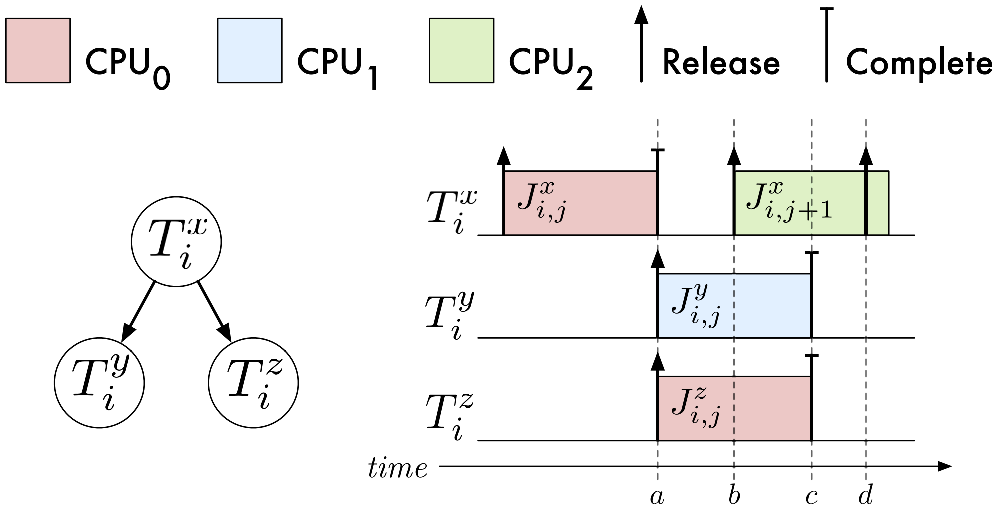


Figure 5.3: Parallel execution of graph nodes.

Ideally, any node with satisfied constraints should be eligible for scheduling to allow the parallel execution and pipelining of nodes. This would allow three types of parallelism: inter-node, intra-node, and pipeline parallelism. Consider the graph and schedule depicted in Figure 5.3. Here, the completion of job $J_{i,j}^x$ at time a releases jobs $J_{i,j}^y$ and $J_{i,j}^z$. These released jobs execute in parallel during the time interval $[a,c]$. This is an example of inter-node parallelism, since two nodes released by a shared parent may execute simultaneously. Job $J_{i,j+1}^x$ is released at time b. It is schedule in parallel with jobs $J_{i,j}^y$ and $J_{i,j}^z$ during the time interval $[b,c]$. This is an example of pipeline parallelism, since $J_{i,j+1}^x$ may execute before the jobs released by $J_{i,j}^x$ complete. Job $J_{i,j+2}^x$ is released at time d. If $J_{i,j+2}^x$ were scheduled before the completion of $J_{i,j+1}^x$, then this would be an example of intra-node parallelism. However, the schedule depicted in Figure 5.3 observes job precedence constraints; *i.e.*, two instances of the *same* node may not executing simultaneously.

The transformation of a PGM-specified graph into a set of sporadic tasks gives us *most* of the parallelism we seek. Although it would be desirable to support intra-node parallelism, it may be challenging to realize in implementation, since it may require dynamic thread creation or *preemptive* work assignment to threads in a pool of worker threads. Moreover, intra-node parallelism may also require us to re-order or buffer node outputs, since the $(j+1)^{th}$ invocation of a node may complete before its j^{th} invocation. From a practical perspective, we feel the loss of intra-node parallelism acceptable.

5.2 PGM^{RT}

In this section, we describe PGM^{RT}, a middleware library that we developed to support PGM-derived real-time task sets. PGM^{RT} is responsible for transmitting tokens (and, optionally, data) among the nodes and enforcing token constraints.[4] PGM^{RT} may be configured to integrate with LITMUS^{RT} in order to reduce token transmission overheads and ensure predictable real-time execution. However, PGM^{RT} may also be configured to support other POSIX-compliant platforms at the expense of greater overheads and priority inversions.

We begin by describing the underlying mechanisms of PGM^{RT}. Specifically, graph management and token transmission. We then address issues relating to proper real-time scheduling of tasks running under our PGM-derived sporadic task model on LITMUS^{RT}.

5.2.1 Graphs, Nodes, and Edges

Each graph is identified by a unique name and path, similar to a UNIX named pipe. Applications use PGM^{RT}'s APIs to create new graphs described by nodes and edges. Real-time tasks, as unique threads of execution within the same address space or separate processes, use PGM^{RT}'s APIs to access information about a named graph and claim/bind to a node and its edges. PGM^{RT} uses a plugin-based architecture to support different methods for transmitting tokens (and, optionally data) among tasks.

5.2.2 Precedence Constraints and Token Transmission

Non-source nodes have two types of precedence constraints: job and token constraints. Job constraints are satisfied in PGM^{RT}, since a single thread binds to each node—jobs are naturally serialized by this thread. Regarding token constraints, consumers block (suspend execution) whenever they lack the requisite tokens. Producers must have a mechanism to signal consumers of new tokens. The appropriate underlying IPC mechanism depends upon how tokens are used: tokens may be *event-signaling* or *data-passing*. A single node may use a mix of event-signaling and data-passing tokens, as appropriate. Regardless of the underlying IPC, nodes produce and consume tokens using a common API.

Event-Signaling Tokens. With event-signaling tokens, token production and consumption is realized through increment/decrement operations on per-edge token counters, similar to counting semaphores. To facilitate

[4]We distribute PGM^{RT} as open source under the Revised BSD license. Source code is currently available at `www.github.com/GElliott/pgm`.

IPC, token counters may be stored in POSIX shared memory that is mapped into each PGM$^{\text{RT}}$ application. Thus, a single graph may be made up of multiple coordinated processes.

Although tokens do not transmit data implicitly, tokens can coordinate data sharing in application-level logic. For example, a token may signal the availability of new data in an out-of-band queue (*i.e.*, a data structure outside the purview of PGM$^{\text{RT}}$) shared between two nodes. Signaling is achieved via a monitor synchronization primitive (which can also be stored in shared memory to facilitate inter-process communication). For POSIX-compliant operating systems, this monitor is realized by a POSIX (pthread) condition variable, one per consumer. A consumer blocks on its condition variable if it does not have requisite tokens on every edge. A producer signals the condition variable whenever it is the last producer to satisfy all of its consumer's token constraints.

The use of pthread condition variables has the drawback that threads that utilize the synchronization primitive must first acquire a pthread mutex. This suspension-based mutex can be problematic in a real-time setting for three reasons. First, it can introduce heavy overheads, due to context switching, with respect to very short critical sections in PGM$^{\text{RT}}$ (*e.g.*, the increment/decrement of a handful of token counters). Such overheads may be pessimistically accounted for in real-time analysis, but it is desirable to avoid them altogether. Second, suspensions are difficult to model under some methods of real-time analysis (*e.g.*, s-oblivious analysis). Finally, an operating system may offer little or no support for real-time priority inheritance for pthread mutexes. As an alternative to pthread-based monitor, PGM$^{\text{RT}}$ also offers a FIFO-ordered spinlock-based monitor built upon Linux's "fast user-space mutex" (or "futex") API. Its use can eliminate costly context switches and problematic suspensions. Furthermore, the duration of priority inversions are bounded since spinning tasks wait non-preemptively. On non-LITMUS$^{\text{RT}}$ platforms, non-preemptive waiting is achieved by disabling CPU interrupts from the user-space (*e.g.*, `sti/cli/pushf/popf` instructions on x86 processors). On LITMUS$^{\text{RT}}$, PGM$^{\text{RT}}$ uses LITMUS$^{\text{RT}}$'s special support for non-preemptive code sections. Here, an application enters and exits non-preemptive sections by merely writing to a variable shared by the application and LITMUS$^{\text{RT}}$.

Data-Passing Tokens. With data-passing tokens, each byte-sized token is interpreted as a byte of data. Byte-ordering is preserved through FIFO-ordered token consumption. Data-passing can be achieved through a variety of IPC channels. PGM$^{\text{RT}}$ supports PGM$^{\text{RT}}$-provided ring buffers, named pipes (FIFOs), message queues (MQs), and stream sockets (*e.g.*, TCP)—all IPCs are POSIX-standard/compatible. We classify these mechanisms collectively as IPC channels. One channel is created for each edge. With the exception of the ring-buffer IPC,[5] consumers block on the set of channels, represented as a list of file descriptors, from each inbound edge using `select()`.[6] The operating system wakes the consumer whenever data arrives on a previously empty channel. Consumers `read()/recv()` tokens from the channel once tokens are available on all edges. Under PGM$^{\text{RT}}$, all read/write operations are non-blocking in order to avoid excessive thread suspensions. Under non-blocking writes, producers continually write data from within a loop until all data has been written. Consumers similarly loop while reading. As a fail-safe mechanism, consumers suspend through `select()` if inbound data is exhausted before the requisite number of bytes have been read. Due to the general lack of introspective capabilities with the above IPC mechanisms (specifically, the inability to query the IPC channel regarding the amount of available data), PGM consumer thresholds greater than the number of tokens consumed per node invocation are not easily supported. However, PGM$^{\text{RT}}$ offers a solution to this problem that we discuss next.

The use of `select()` for data-passing tokens can introduce additional thread suspensions since `select()` wakes a blocked thread when data becomes available on any *one* channel. Thus, a consumer waiting for data on all inbound edges must loop on `select()` until data arrives on all inbound edges.[7] To avoid this, PGM$^{\text{RT}}$ offers "fast" variants of FIFO- and MQ-based channels (and PGM$^{\text{RT}}$ ring buffers are only available in this flavor), where the underlying channel IPC is *wrapped* with event-signaling tokens. Here, the availability of data is tracked by event-signaling tokens, with each token corresponding to one byte of data. As with plain event-signaling tokens, consumers block on a monitor, and are awoken by the last producer to satisfy all token constraints. Producers only transmit event-signaling tokens after they have written the produced data to the associated channel. Thus, consumers using the fast channels avoid repeated suspensions while looping on `select()`. Moreover, support for PGM consumer thresholds greater than the number of tokens consumed per node invocation is trivialized by event-signaling token counters. Thus, the fast channel variants also support PGM's consumer thresholds. There is one limitation to using event-signaling tokens in this context: a reliance upon shared memory. As a result, PGM$^{\text{RT}}$ does not offer a fast variant of stream socket

[5]Ring-buffers use the "fast" method described next.

[6]The use of `select()` with MQs is not strictly POSIX-compliant, but such use is commonly supported.

[7]An "all-or-nothing" variant of `select()` could be used to address this issue. However, we are unaware of any OS that supports such an API.

channels, since producers and consumers connected by these channels are expected to reside on different computers in a distributed system.

5.2.3 Real-Time Concerns

PGM$^{\text{RT}}$ described as above can be used with general-purpose schedulers. However, additional enhancements are required to ensure predictable real-time behavior. These relate to predictable token signaling and proper deadline assignment. We describe how we address these problems when PGM$^{\text{RT}}$ runs atop LITMUS$^{\text{RT}}$.

Early Releasing and Deadline Shifting. Under deadline-based schedulers, the technique of *early releasing* allows a job to be scheduled prior to its release time, provided that the job's absolute deadline is computed from the normal (*i.e.*, "non-early") release time. Under LITMUS$^{\text{RT}}$, the jobs of all tasks associated with non-source nodes are released early. However, early-released jobs must still observe token constraints.

A job's absolute deadline is computed as d_i time units after its (non-early) release. LITMUS$^{\text{RT}}$ employs high-resolution timers to track the minimum separation time between releases. However, recall from Section 2.1.3 that the release time of a non-source node can be no less than the time instant the job's token constraints are satisfied. Thus, the absolute deadline for each job must be computed on-the-fly. Immediately before a consumer blocks for tokens, it sets a "token-wait" flag stored in memory shared by user-space and the kernel. The kernel checks this flag whenever a real-time task is awoken from a sleeping state. If set, and the current time is later than the release time dictated by the sporadic task model, the kernel automatically computes an adjusted release and absolute deadline for the job and clears the flag. This computation requires the current time to approximate the arrival time of the last token—this is ensured by boosting the priority of a producer while it signals consumers. We discuss this next.

Priority-Boosting of Producers. To ensure properly computed deadlines, we boost the priority of a producer while it is signaling a sequence of consumers. Moreover, in cases where a graph spans multiple processor clusters, boosting is necessary to avoid leaving processors in remote clusters idle. Boosting is achieved through a lazy process: the priority of a producer is boosted only when the scheduler attempts to preempt it. A producer informs LITMUS$^{\text{RT}}$ of when it is signaling consumers through a "token-sending" flag stored in memory shared by user-space and the kernel. Once all consumers have been singled, a producer clears the "token-sending" flag and triggers LITMUS$^{\text{RT}}$ to "unboost" and reschedule the producer. Priority inversions due to boosting should be accounted for in real-time analysis.

We now discuss applications where we may use PGM$^{\text{RT}}$ to help realize real-time graph scheduling, especially those that use GPUs.

5.3 OpenVX

OpenVX is a newly ratified standard API for developing computer vision applications for heterogeneous computing platforms. The API provides the programmer with a set of basic operations, or *primitives*, commonly used in computer vision algorithms.[8] The programmer may supplement the standard set of OpenVX primitives with their own or with those provided by third-party libraries. Each primitive has a well-defined set of inputs and outputs. The implementation of a primitive is defined by the particular implementation of the OpenVX standard. Thus, a given primitive may use a GPU in one OpenVX implementation and a specialized DSP (*e.g.*, CongniVue's G2-APEX or Renesas' IMP-X4) or mere CPUs in another. OpenVX also defines a set of *data objects*. Types of data objects include simple data structures such as scalars, arrays, matrices, and images. There are also higher-level data objects common to computer vision algorithms—these include histograms, image pyramids, and lookup tables.[9] The programmer constructs a computer vision algorithm by instantiating primitives as *nodes* and data objects as *parameters*. The programmer binds parameters to node inputs and outputs. Since each node may use a mix of the processing elements of a heterogeneous platform, a single graph may execute across CPUs, GPUs, DSPs, *etc.*

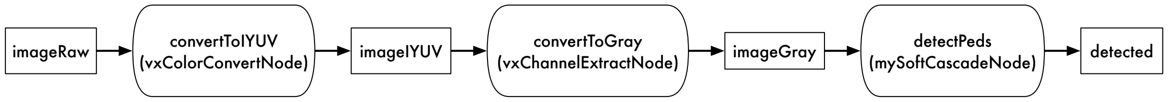
Node dependencies (*i.e.*, edges) are not explicitly provided by the programmer. Rather, the structure of a graph is derived from how parameters are bound to nodes. We demonstrate this with an example. Figure 5.4(a) gives the relevant code fragments for creating an OpenVX graph for pedestrian detection. The data objects `imageRaw` and `detected` represent the input and output of the graph, respectively. The data objects `imageIYUV` and `imageGray` store an image in color and grayscale formats, respectively. At line 12, the code creates a color-conversion node, `convertToIYUV`. The function that creates this node, `vxColorConvertNode()`, takes `imageRaw` and `imageIYUV` as input and output parameters, respectively. Whenever the node represented by `convertToIYUV` is executed, the contents of `imageRaw` is processed by the color-conversion primitive, and the resulting image is stored in `convertToIYUV`. Similarly, the node `convertToGray` converts the color image into a grayscale image. The grayscale image is processed

[8]The OpenVX specification calls these basic operations "kernels." However, we opt to avoid this term since we must already differentiate between GPU and OS kernels.

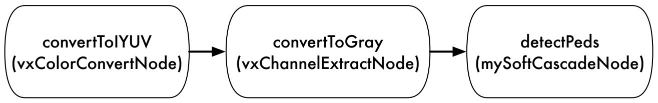
[9]An image pyramid stores multiple copies of the same image. Each copy has a different resolution or scale.

```
vx_image imageRaw; // graph input  : an image
vx_array detected; // graph output : a list of detected pedestrians
...
// instantiate a graph
vx_graph pedDetector = vxCreateGraph(...);
...
// instantiate additional parameters
vx_image imageIYUV = vxCreateVirtualImage(pedDetector, ...);
vx_image imageGray = vxCreateVirtualImage(pedDetector, ...);
...
// instantiate primitives as nodes
vx_node convertToIYUV = vxColorConvertNode(pedDetector, imageRaw, imageIYUV);
vx_node convertToGray = vxChannelExtractNode(pedDetector, imageIYUV,
                          VX_CHANNEL_Y, imageGray);
vx_node detectPeds    = mySoftCascadeNode(pedDetector, imageGray, detected, ...);
...
vxProcessGraph(pedDetector); // execute the graph end-to-end
```

(a) OpenVX code for constructing a graph.


imageRaw
convertToIYUV (vxColorConvertNode)
imageIYUV
convertToGray (vxChannelExtractNode)
imageGray
detectPeds (mySoftCascadeNode)
detected

(b) Bindings of data object parameters to nodes.


convertToIYUV (vxColorConvertNode)
convertToGray (vxChannelExtractNode)
detectPeds (mySoftCascadeNode)

(c) Derived graph structure.

Figure 5.4: Construction of a graph in OpenVX for pedestrian detection.

by a user-provided node created by the function `mySoftCascadeNode()`, which writes a list of detected pedestrians to `detected`.[10] Figure 5.4(b) depicts the bindings of parameters to nodes. Figure 5.4(c) depicts the derived structure of this graph.

OpenVX defines a simple execution model. From Section 2.8.5 of the OpenVX standard:

> *[A constructed graph] may be scheduled multiple times but only executes sequentially with respect to itself.*

Moreover:

> *[Simultaneously executed graphs] do not have a defined behavior and may execute in parallel or in series based on the behavior of the vendor's implementation.*

[10]The OpenVX standard does not currently specify a primitive for object detection, so the user must provide their own or use one from a third-party.

This execution model simplifies the OpenVX API and its implementation. Also, this simplicity is partly motivated by the variety of heterogeneous platforms upon which OpenVX applications are meant to run. The API must work well for simple processors such as ASICs as well as modern CPUs. Furthermore, the simple execution model enables interesting opportunities for optimization. For example, an entire graph could be transformed into gate-level logic or a single GPU kernel and executed entirely on an FPGA or GPU, respectively. However, OpenVX's execution model has four significant implications on real-time scheduling. First, the specification has no notion of a periodic or sporadic task. Second, the specification only allows the programmer to control when the root node of a graph is ready for execution, not when *internal* nodes are ready. Third, the specification does not define a threading model for graph execution. The intent of the standard is to allow the OpenVX implementation to be tailored to particular heterogeneous platforms. However, it provides no mechanism by which to control the number of threads used to execute a graph or the priority of these threads. Finally, the specification requires a graph to execute end-to-end before it may be executed again. This makes pipelining impossible.[11]

Given these limitations, how can we execute OpenVX graphs under a sporadic task model? As we discussed in Section 5.1, we may assign a single graph to a single sporadic real-time task. A job of this task would execute the nodes of the graph serially in topological order. Of course, we miss opportunities to take advantage of inherit graph parallelism with this approach. We present a better solution in the next section.

5.4 Adding Real-Time Support to VisionWorks

VisionWorks is an implementation of OpenVX developed by NVIDIA. Many of the OpenVX primitives are implemented using CUDA and are optimized for NVIDIA GPUs. The VisionWorks software provides an ideal tool with which we can evaluate the effectiveness of GPUSync for the real-time scheduling of real-world applications. With support from NVIDIA, we adapted an alpha-version of VisionWorks to run atop PGM^{RT}, GPUSync, and LITMUS^{RT}. In this section, we describe several challenges we faced in this effort, along with our solutions. However, first we wish to note that the alpha-version of VisionWorks provided to us by NVIDIA was under active development at the time. *The reader should not assume that statements we make regarding VisionWorks will necessarily hold when the software is made available to the public.* Also, our

[11] At line 17 of Figure 5.4(a), the pedestrian detection graph is executed once, from end-to-end, by calling the function `vxProcessGraph()`. This function does not return until the graph has completed. The OpenVX function `vxScheduleGraph()` may be used to *asynchronously* execute a graph without blocking. However, a graph instance must still complete before it may be reissued.

work with VisionWorks was funded by NVIDIA through their internship program—at this time we are unable to share the software we developed in this effort,[12] since it is the property of NVIDIA.

We describe the software we developed to bring real-time support to VisionWorks in three parts. We begin by describing the changes we made to the VisionWorks execution model to support a PGM-derived sporadic task model. Following this, we discuss a separate GPGPU interception library, called *libgpui*, we developed to transparently add GPUSync GPU scheduling to VisionWorks and the third-party libraries it uses. Finally, we describe how our modified VisionWorks execution model directly interacts with GPUSync; specifically, when GPU tokens are acquired and engine locks are obtained.

5.4.1 VisionWorks and the Sporadic Task Model

In this section, we describe the graph execution model used by our alpha-version of VisionWorks and how we modified it to support the sporadic task model.

VisionWorks adopts the simple execution model prescribed by the OpenVX specification. Moreover, in the alpha-version software, nodes of a graph are executed in topological order by a single thread. However, VisionWorks places no restrictions on the threading model used by individual primitives. We found that several primitives, by way third party libraries such as OpenCV, employ OpenMP to execute for-loops across several parallel threads. In order to remain within the constraints of the single-threaded sporadic task mode, we took the necessary steps to disable such intra-node multi-threading. For example, to control OpenMP, we set the environment variable `OMP_NUM_THREADS` $= 1$, effectively disabling it.

Sporadic Task Set Execution Model. Our first change to VisionWorks was to introduce a new API function, `nvxSpawnGraph()`, that is used to spawn a graph for *periodic* execution. This function is somewhat analogous to the OpenVX asynchronous graph-scheduling function `vxScheduleGraph()`. However, unlike `vxScheduleGraph()`, a graph spawned by `nvxSpawnGraph()` executes repetitively, instead of only once. Each node of a spawned graph is executed by a dedicated thread. Each thread is assigned a common period specified by the programmer. We assume implicit deadlines. An invocation of a node is equivalent to a periodic job (we sometimes use the terms "node" and "job" interchangeably). We call our new execution model for VisionWorks the "sporadic task set execution model."

[12]Specifically, we refer to our modified version of VisionWorks and a software library we call libgpui.

Graph Input and Output. The OpenVX API assumes that graph input is primed prior to the call of `vxProcessGraph()` (or `vxScheduleGraph()`). For example, in Figure 5.4(a), the contents of the input image `imageRaw` must be set prior to the call to `vxProcessGraph()` on line 17. We achieve a similar behavior by attaching an "input callback" to the spawned graph. The input callback is executed by an "input callback node" that is prepended to the graph—all source nodes of the original graph become children to the input callback node. The same approach is taken to handle graph output, where sink nodes become parents of an appended "output callback node."

Graph Dependencies and Pipelining. We use PGM$^{\text{RT}}$ to coordinate the execution of the per-node threads. We achieve this by duplicating the VisionWorks graph structure with an auxiliary graph in PGM$^{\text{RT}}$. We connect the nodes of the PGM$^{\text{RT}}$ graph with edges that use event-signaling tokens. We set $\varrho_i^{k \leftarrow j} = \kappa_i^{k \leftarrow j} = \varphi_i^{k \leftarrow j} = 1$ for all edges.

Recall from Section 5.3 that OpenVX does not pass data through graph edges. Rather, node input and output is passed through *singular instances* of data objects. Although graph pipelining is naturally supported by the periodic task set execution model, a new hazard arises: *a producer node may overwrite the contents of a data object before the old contents has been read or written by consumer node!* Such consumers may not even be a direct successor of the producer. For instance, we can conceive of a graph where an image data object is passed through a chain of nodes, each node applying a filter to the image. The node at the head of this chain cannot execute again until after the image has been handled by the node at the tail. In short, the graph cannot be pipelined.

To resolve this pipelining issue, we begin by first *replicating* all data objects in the graph N times. We set N to the maximum depth of the graph as a rule of thumb. The number of replicas is actually configurable by the programmer. (Any value for $N \geq 1$ will work, given the failsafe mechanism we discuss shortly.) The j^{th} invocation of a node (*i.e.*, a job) accesses the $(j \bmod N)^{th}$ replica. However, replication alone does not ensure safe pipelining, since we do not enforce end-to-end graph precedence constraints. For example, a node on its $(j+N)^{th}$ invocation may execute before the data it generated on its j^{th} invocation has been fully consumed. To prevent this from happening, we introduce a failsafe that stalls the node whenever such hazards arise.

The failsafe is realized through an additional PGM "feedback graph." To generate the feedback graph, we first *copy* the auxiliary PGM graph. We then add *additional* edges to this copy. Each additional directed edge connects a node that accesses a data object to any *descendant* node that also accesses that data object

```
procedure DoJob()
    ConsumeTokens();                 ▷ Wait for input from producers.
    ConsumeFeedbackTokens();         ▷ Wait for go-ahead from descendants.
    DoPrimitive();                   ▷ Execute the primitive.
    ProduceFeedbackTokens();         ▷ Give go-ahead to any waiting ancestors.
    ProduceTokens();                 ▷ Signal output to consumers.
end procedure
```

Figure 5.5: Procedure for PGM$^{\mathrm{RT}}$-coordinated job execution.

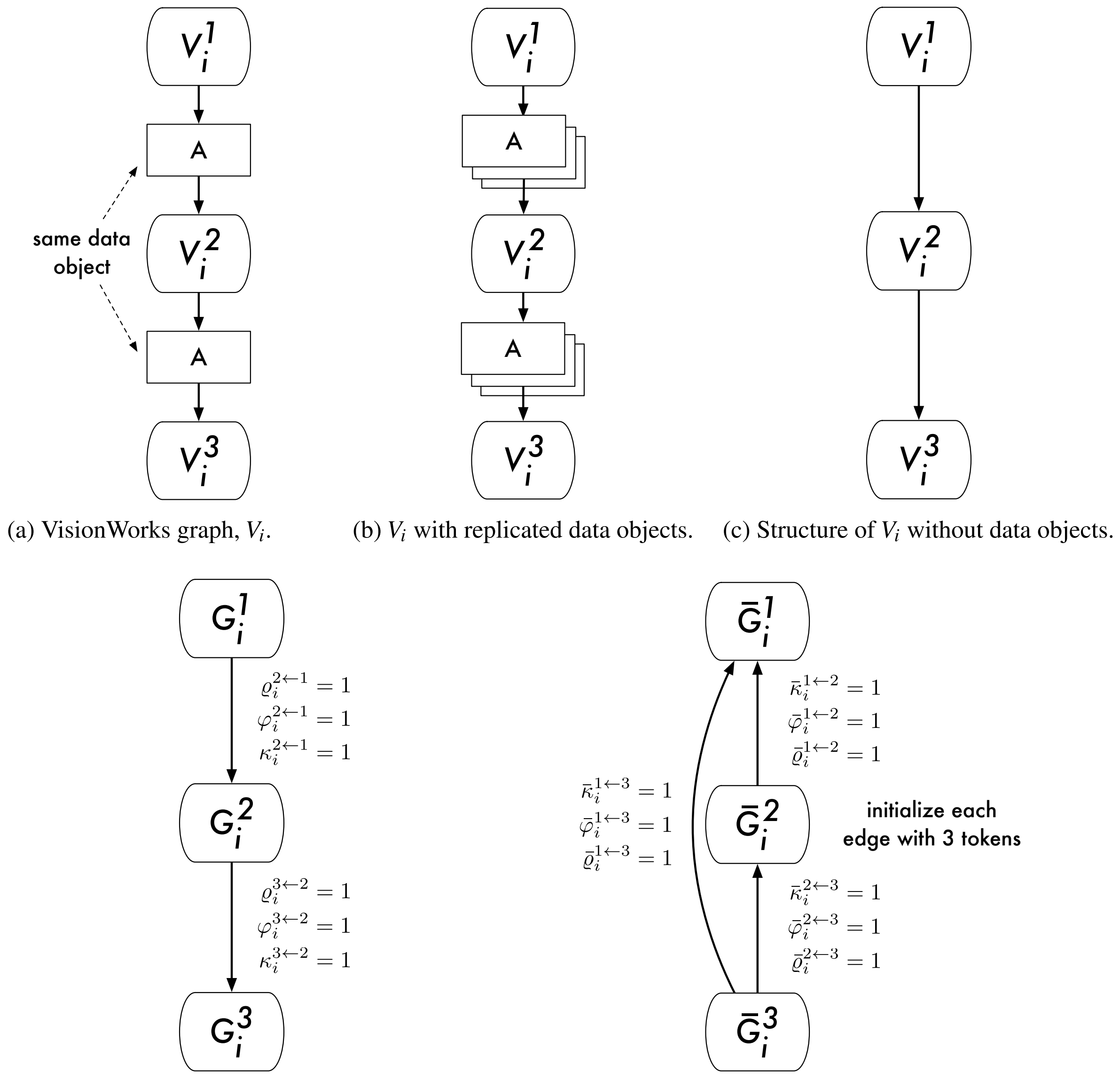
(no edge is added if such an edge already exists). We then *flip* the direction of all edges in this graph, and each edge is initialized with an available token count equal to N.

Figure 5.5 outlines the procedure executed by each thread to execute a job. The j^{th} invocation of the node first waits for and consumes a token from each of its producers. It then consumes a token from each inbound-edge in the feedback graph, blocking if the requisite tokens are unavailable. The node only blocks if the $(j \bmod N)^{th}$ data object replicas are still in use.[13] Thus, the node stalls, and the hazard is avoided. After executing the actual primitive, the node signals that it is done using the $(j \bmod N)^{th}$ data object replicas. The node then generates tokens for its consumers.

Example 5.1. We illustrate the measures we take to support pipelining with an example depicted in Figure 5.6. In Figure 5.6(a), we begin with a simple three-node VisionWorks graph, V_i. Suppose that node V_i^1 writes to the data object A; node V_i^2 modifies A; and node V_i^3 reads A. Graph V_i has a depth of three. In Figure 5.6(b), we replicate the data object A such that there are three replicas. Figure 5.6(c) gives the structure of V_i, without data objects. This structure is replicated in PGM$^{\mathrm{RT}}$, represented by graph G_i in Figure 5.6(d). Token production, consumption, and threshold parameters are set to one. We generate the feedback graph, $\bar{G}_i$, as depicted in Figure 5.6(e). Note the edge connecting node $\bar{G}_i^3$ to $\bar{G}_i^1$. ◊

Support for Back-Edges. Computer vision algorithms that operate on video streams often feed data derived from prior frames back into the computations performed on future frames. For example, an object tracking algorithm must recall information about objects of prior frames if the algorithm is to describe the motions of those objects in the current frame. OpenVX defines a special data object called a "delay," which is used to buffer node output for use by subsequent node invocations. A delay essentially a ring buffer used to contain other data objects (*e.g.*, prior image frames). The oldest data object is overwritten when a new data object

[13]We disable priority boosting and deadline shifting for operations on the feedback graph in order to prevent alterations to properly assigned real-time priorities. The feedback graph is a failsafe mechanism; we do not want it to interfere with the resumption of normal execution. Of course, these special measures only apply to spawned graphs executing under LITMUS$^{\mathrm{RT}}$, since these features are specific to LITMUS$^{\mathrm{RT}}$


V_i^1
A
same data object
V_i^2
A
V_i^3
G_i^1
$\varrho_i^{2\leftarrow1} = 1$
$\varphi_i^{2\leftarrow1} = 1$
$\kappa_i^{2\leftarrow1} = 1$
G_i^2
$\varrho_i^{3\leftarrow2} = 1$
$\varphi_i^{3\leftarrow2} = 1$
$\kappa_i^{3\leftarrow2} = 1$
G_i^3
$\bar{G}_i^1$
$\bar{\kappa}_i^{1\leftarrow2} = 1$
$\bar{\varphi}_i^{1\leftarrow2} = 1$
$\bar{\varrho}_i^{1\leftarrow2} = 1$
$\bar{\kappa}_i^{1\leftarrow3} = 1$
$\bar{\varphi}_i^{1\leftarrow3} = 1$
$\bar{\varrho}_i^{1\leftarrow3} = 1$
$\bar{G}_i^2$
initialize each edge with 3 tokens
$\bar{\kappa}_i^{2\leftarrow3} = 1$
$\bar{\varphi}_i^{2\leftarrow3} = 1$
$\bar{\varrho}_i^{2\leftarrow3} = 1$
$\bar{G}_i^3$

(a) VisionWorks graph, V_i.

(b) V_i with replicated data objects.

(c) Structure of V_i without data objects.

(d) PGM graph, G_i, of V_i.

(e) PGM feedback graph, $\bar{G}_i$, for V_i.

Figure 5.6: Derivation of PGM graphs used to support the pipelined thread-per-node execution model.

enters the buffer. The number of data objects stored in a ring buffer (or the "size" of the delay) is tied to how "far into the past" the vision algorithm must go. For example, suppose a node operates on frame *i* and it needs to access copies of the last two prior frames. In this case, the size of the delay would be two.

The consumer node of data buffered by a delay may appear anywhere within a graph. It may be a ancestor or descendant of the producer node—it may even be the producer itself. A back-edge is created when the consumer node of a delay is not a descendant of the producer node in the graph derived from non-delay data objects. Such back-edges can be used to implement the object tracking algorithm described above.

Due to complexities in the implementation and use of delays in VisionWorks, for our periodic task set execution model, we do not replicate delay data objects as we do for other types of data objects. Instead, we simply increase the size of the delay ring buffer to store an additional N data objects. For example, suppose we have a delay with size M; we increase the size of the delay to $M+N$. We also introduce the necessary back-edges to our auxiliary and feedback PGM graphs. The available token counts for these back-edges must be initialized with the appropriate number of tokens to allow a consuming node to execute while the delay is initially filling with data objects.

Scheduling Policies. The programmer may specify which scheduling policy to use for the threads in our sporadic task set execution model. Our modified VisionWorks supports the standard Linux SCHED_OTHER and SCHED_FIFO policies, as well as SCHED_LITMUS. We use POSIX real-time timers to implement periodic job releases under the standard Linux policies. We rely upon the periodic job release infrastructure of LITMUS^{RT} for the SCHED_LITMUS policy.

We assume deadline-based scheduling under the SCHED_LITMUS policy. When the SCHED_FIFO policy is employed, the programmer supplies a "base" graph priority. The priority of a thread of a given node is determined by taking the length of the longest path between the node and the input callback source node, plus the base graph priority. Thus, thread priorities increase monotonically down the graph. We use this prioritization scheme to expedite the movement of data down a graph. It is important to finish work deeper in the graph, since graph execution is pipelined.

5.4.2 Libgpui: An Interposed GPGPU Library for CUDA

VisionWorks is a large and complex software package. The alpha-version of VisionWorks we modified was made up of approximately 80k lines of code, not including third-party libraries. In the evaluation of

GPUSync in Chapter 4, the test programs interfaced directly with GPUSync through the user interface provided by liblitmus, as described in Section 3.3.5. We deemed it infeasible to manually alter VisionWorks and GPU-using third-party libraries to use GPUSync directly.

Instead, we developed an interposition library, *libgpui*, to intercept all calls to the lowest-level library of NVIDIA's CUDA software, libcuda. Libgpui intercepts all API calls that may launch a kernel or issue a DMA operation. With the exception of GPU token acquisition, libgpui is entirely transparent to the user—no modifications to user code are necessary. The library automatically interfaces with GPUSync on behalf of the API callee. For example, with features provided by libcuda, libgpui inspects the memory address parameters of the DMA API call to deduce the end-points of the DMA operation (*i.e.*, the memories of the source and destination of the DMA) and libgpui automatically requests the needed copy engine locks. Each API call is passed on to libcuda once GPUSync schedules the operation. Libgpui also automatically breaks large DMA operations into smaller chunks, as we discussed in Section 3.2.3.4.

Libgpui also overrides three default behaviors of the CUDA runtime to improve real-time predictability. First, libgpui can be configured to enforce suspension-based synchronization when tasks wait for a GPU operation to complete. Second, libgpui ensures that proper stream synchronization behaviors are followed to ensure engine independence (see Section 2.4.4), if it is required for the particular GPU being used. Finally, libgpui forces all GPU operations to be issued on unique per-thread CUDA streams. This is a change from the current CUDA runtime, where threads within a process share a common stream by default.[14] This prevents the threads for contending for access to a shared CUDA stream, which would be arbitrated by the CUDA runtime and result in unpredictable behavior.

5.4.3 VisionWorks, libgpui, and GPUSync

We now describe how VisionWorks interacts with GPUSync through libgpui. The alpha-version of VisionWorks does not support transparent migration of data objects among GPUs, so we focus our attention on partitioned GPU scheduling under GPUSync.[15] The user assigns a given VisionWorks application to a GPU partition through an environment variable when the application is launched. Libgpui reads this variable and initializes the CUDA runtime for the selected GPU on behalf of the VisionWorks application.

[14]Coincidentally, a stream-per-thread feature similar to libgpui's is slated to be a part of the forthcoming CUDA 7.0 (Harris, 2015).

[15]VisionWorks data objects abstractions do not preclude transparent migration. However, we deemed adding such support to be a non-trivial effort and outside the scope of our core goal of adding real-time GPU scheduling to VisionWorks.

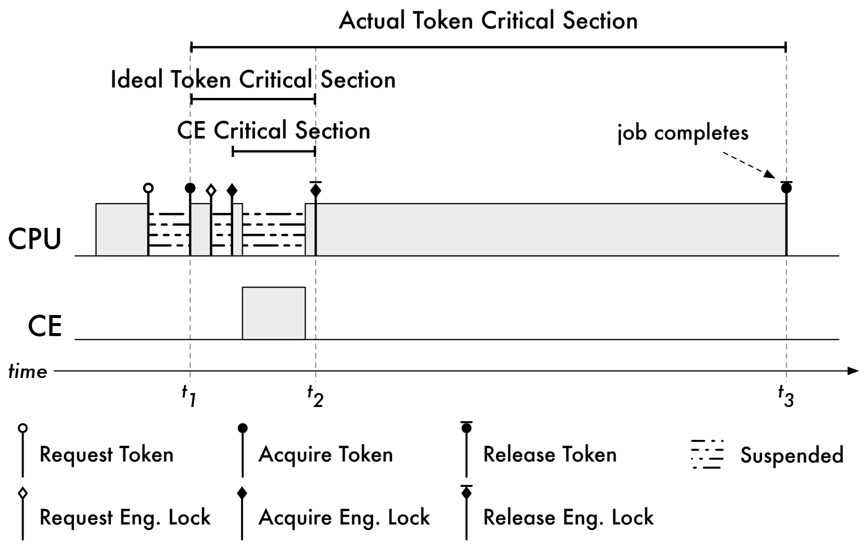


Figure 5.7: Overly long token critical sections may result by releasing tokens at job completion time.

Although there is only one GPU within each partition, GPUSync still requires each GPU-using job to acquire one of the ρ tokens before using the GPU. This is necessary to ensure that interrupt and CUDA runtime callback threads are assigned proper real-time priorities. When libgpui intercepts an API call, it first checks to see if the callee already holds a GPU token. If not, libgpui requests and obtains a token before proceeding with the necessary engine lock request. We modified VisionWorks to interface *directly* with libgpui to release any held token upon job completion—this is the only aspect to libgpui that is not transparent to VisionWorks. (Unfortunately, there is no CUDA API libgpui can intercept and interpret as a job completion, so libgpui must be notified explicitly.)

One drawback to using libgpui in this way is that token critical section lengths may be longer than are strictly necessary. We depict such a case in Figure 5.7, where a primitive performs a DMA operation early in its execution and does not use the GPU during the remainder of its computations. We may see such an execution pattern in a primitive that executes primarily on a CPU, but consumes the output of another primitive that executes on a GPU. In Figure 5.7, at time t_1, the primitive operation executing on the CPU obtains a GPU token. The primitive then copies an OpenVX data object from GPU memory to host memory. This DMA operation completes at time t_2. The remaining computations of the primitive take place entirely on the CPU, completing at time t_3. At this point, the GPU token is released. Under our methodology, the GPU token is held during these CPU computations over the time interval $[t_1, t_3]$, even though the GPU is no longer required after time t_2. Ideally, the GPU token would be freed at time t_2. This is a compromise we make by not integrating GPUSync with VisionWorks directly. We may work around this issue by splitting the primitive into two parts: a DMA primitive and a CPU-computation primitive. However, such a change is invasive—perhaps more invasive than simply modifying the primitive to communicate directly with libgpui to release the token early. Perhaps future versions of VisionWorks or OpenVX could include an API that allows a primitive to express to the execution framework of when a particular heterogeneous computing element (*e.g.*, a GPU) is no longer needed. This API could be leveraged to shorten the token critical section length.

5.5 Evaluation of VisionWorks Under GPUSync

In this section, we evaluate the runtime performance of our sporadic task set execution model for VisionWorks under eight configurations of GPUSync. We compare our results against VisionWorks (using the same sporadic task set execution model) running under two purely Linux-based configurations, as well as a LITMUS^{RT} configuration *without* GPUSync.

The rest of this section is organized as follows. We begin with a description of the computer vision application we used to evaluate VisionWorks with GPUSync. We then discuss our experimental setup. Finally, we present our results in two parts. In the first part, we report the observed frame-to-frame output delays of many concurrently executed graphs in our experiments. In the second part, we report on observed end-to-end response time latencies of these same graphs.

5.5.1 Video Stabilization

VisionWorks includes a variety of demo applications, including a pedestrian detection application, not unlike the one illustrated in Figure 5.4. However, the pedestrian detection application is relatively uninteresting from a scheduling perspective—it is made up of only a handful of nodes arranged in a pipeline. In our evaluation, we use VisionWorks' "video stabilization" demo, since the application is far more complex and uses primitives common to other computer vision algorithms. Video stabilization is used to digitally dampen the effect of shaky camera movement on a video stream. Vehicle-mounted cameras are prone to camera shake. A video stream may require stabilization as a pre-processing step before higher-level processing is possible. For example, an object tracker may require stabilization—too much shake may decrease the accuracy of predicted object trajectories.

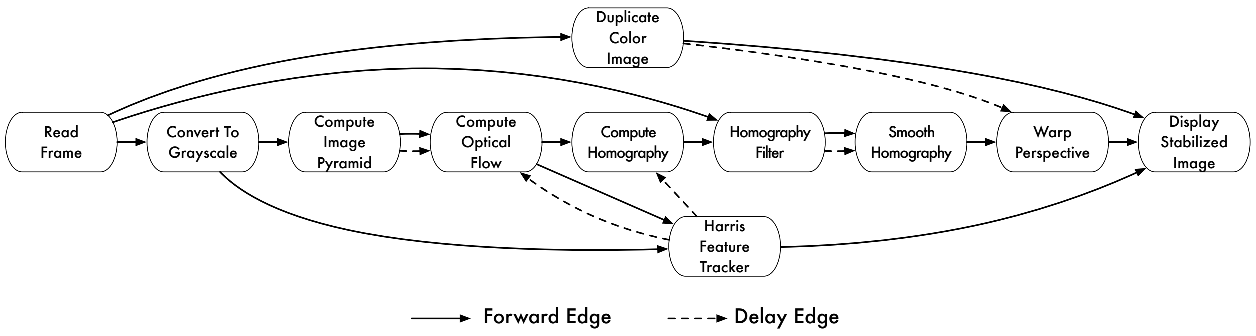


Figure 5.8: Dependency graph of video stabilization application.

Node Name	Function
Read Frame	Reads a frame from the video source.
Duplicate Color Image	Copies the input color image for later use.
Convert To Grayscale	Converts a frame from a color to grayscale (*i.e.*, “black and white”) image.
Harris Feature Tracker	Detects Harris corners (features) in an image.
Compute Image Pyramid	Resizes the image into several images at multiple resolutions.
Compute Optical Flow	Determines the movement of image features from the last frame into the current frame.
Compute Homography	Computes a “homography matrix” that characterizes the transformation from the last frame into the current frame.
Homography Filter	Filters noisy values from homography matrix.
Smooth Homography	Merges the homography matrices of the last several frames into one.
Warp Perspective	Transforms an image using a provided homography matrix. (Stabilization occurs here.)
Display Stabilized Image	Displays the stabilized image.

Table 5.1: Description of nodes used in the video stabilization graph of Figure 5.8.

Figure 5.8 depicts the dependency graph of the video stabilization application. Table 5.1 gives a brief description of each node in this graph. We make note of two characteristics of this graph. First, video stabilization operates over a temporal window of several frames. This is needed in order to differentiate between movements due to camera shake and desired camera translation (*i.e.*, stable long-term movement of the camera). These inter-frame dependencies are implemented using OpenVX delay data objects, which are reflected by delay edges in Figure 5.8. Second, although the primitive of a node may execute entirely on CPUs, it may still use a GPU to pull data out of GPU memory through DMA operations. The “Display Stabilized Image” node is such an example. Here, the “Warp Perspective” node performs its computation and stores a stabilized frame in GPU memory. The Display Stabilized Image node pulls this data off of the GPU through DMA. This means that the Display Stabilized Image node will compete with other nodes for GPU tokens, even though it does not use the GPU to perform computation.

Period	No. Graphs	Base Priority (for SCHED_FIFO only)
20*ms*	2	75
30*ms*	2	60
40*ms*	2	45
60*ms*	5	30
80*ms*	4	15
100*ms*	2	1

Table 5.2: Evaluation task set using VisionWorks' video stabilization demo.

5.5.2 Experimental Setup

In this evaluation, we focus on clustered CPU scheduling with a single GPU. This focus is motivated by the results of Chapter 4, as well as the inability of our version of VisionWorks to support the migration of data objects among GPUs. Moreover, such a multicore single-GPU system reflects many common embedded computing platforms available today.

Our experimental workload was comprised of seventeen instances of the video stabilization application in order to load the CPUs and GPU engines. Each graph was executed within its own Linux process. Thus, each instance had eleven regular real-time threads (one for each node in the graph depicted in Figure 5.8) and one CUDA callback thread, which GPUSync automatically schedules as a real-time task (see Section 3.2.5.2). With the inclusion of the GPU interrupt handling thread (see Section 3.2.5.1), there were a total of 205 real-time threads in the workload.

To each graph we assigned a period that was shared by every real-time task of the nodes therein. Two graphs had 20*ms* periods; another two graphs had 30*ms* periods; and yet another two graphs had 40*ms* periods. Five graphs had a period of 60*ms*; four graphs had a period of 80*ms*; and another two graphs had a period of 100*ms*. These periods represent a range of those we find in ADAS, such as those we described in Chapter 1 (see Table 1.1). Table 5.2 summarizes these period assignments.

We isolated the real-time tasks to a single NUMA node of the hardware platform we used in Chapter 4. The remaining NUMA node was used for performance monitoring and did not execute any real-time tasks. The workload was executed across six CPUs and a single NUMA-local GPU. This heavily loaded the CPUs and GPU to near capacity.[16]

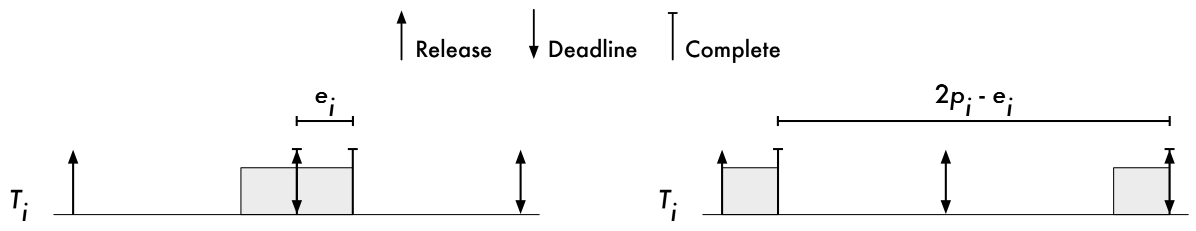
[16]During experimentation, the system tool `top` reported the NUMA node to be 96% utilized (a CPU utilization of 5.76). Similarly, the NVIDIA tool `nvidia-smi` reported the EE to be 66% utilized.

The workload was scheduled under eleven different system configurations. Eight of these configurations used GPUSync under LITMUS$^{\text{RT}}$'s C-EDF scheduling, with CPU clustering around the L3 cache (*i.e.*, G-EDF scheduling within the NUMA node). We experimented with four GPUSync configurations of the GPU Allocator. These configurations differed in the number of tokens, ρ, and maximum FIFO length, f. The number of tokens were set to one, two, three, or six. The maximum FIFO length was set correspondingly to six, three, two, or one. We are interested in these GPU Allocator configurations because the associated GPU token blocking terms under each configuration are optimal with respect to suspension-oblivious schedulability analysis for a platform with six CPUs and a single GPU. Under each corresponding pairing of ρ and f , we experimented with both FIFO-ordered (FIFO) and priority-ordered (PRIO) engine locks, resulting in the eight GPUSync configurations. We refer to each configuration with a tuple of the form $(\rho, f, \text{Engine Lock Protocol})$. For example, a configuration with $\rho = 2$, $f = 3$, and priority-ordered engine locks is denoted by $(2, 3, \mathsf{PRIO})$. The remaining three system configurations were as follows: LITMUS$^{\text{RT}}$ C-EDF scheduling without GPUSync; Linux under SCHED_FIFO fixed-priority scheduling; and Linux under SCHED_OTHER scheduling. These last three configurations relied upon the GPGPU runtime, GPU driver, and GPU hardware to handle all GPU resource arbitration and scheduling. Under the SCHED_FIFO policy, graphs were assigned the base priorities given in Table 5.2. A fixed-priority for the thread of each node was derived using the method we described in Section 5.4.1.

We executed our task set under each of the eleven system configurations for 400 seconds. Each graph processed a pre-recorded video file cached in system memory. For all configurations, we used libgpui to force all tasks to suspend while waiting for GPU operations to complete, and to also ensure that all nodes used distinct GPU streams to issue GPU operations. Libgpui invoked GPUSync for only the GPUSync configurations—libgpui passed CUDA API calls immediately on to the underlying CUDA library for non-GPUSync configurations.

5.5.3 Results

We now report our findings in two parts. In the first part, we examine the observed delay between consecutive outputs of the graphs. In the second part, we examine observed end-to-end response time latencies (*i.e.*, the time from the release of a source node to the completion of the corresponding sink node) under the LITMUS$^{\text{RT}}$-based configurations.



(a) Back-to-back job completion.

(b) First job completes as early as possible, while last job completes as late as possible.

Figure 5.9: Scenarios that lead to extreme values for measured completion delays.

5.5.3.1 Completion Delays

We require a common observational framework in order to fairly compare the eleven system configurations. Although LITMUS^{RT} offers kernel-level low-overhead tracing capabilities, we are unable to make use of them for the non-LITMUS^{RT}-based configurations. Instead, we take our measurements from user-space within the graph applications themselves, and we examine the timing properties of the sink node in each graph. Specifically, we look at *the delay between consecutive completions* of the sink node in each graph. To gather these metrics, the sink node of each graph (*i.e.*, the "Display Stabilized Image" node) records a time stamp at the end of its computation. We then measure the difference in consecutive timestamps to obtain a "completion delay" metric.

There are two benefits to this completion delay metric. First, we may make observations from user space; we do not require special support from the operating system, besides access to accurate hardware time counters (which are commonly available). Second, we may apply the completion delay metric to any periodic task set, regardless of any processor schedulers, be they real-time or general purpose. There are two limitations, however. First, completion delay metrics are only useful within the context of periodic task sets, since task periods not only ensure a minimum separation time between job releases of a given task, but also a *maximum* separation time. We cannot meaningfully compare measured completion delays if the gap between consecutive job releases varies. Second, the metric is inherently noisy. Figure 5.9 depicts two extreme completion delay values that are possible for an implicit-deadline periodic task that is *hard* real-time schedulable. In Figure 5.9(a), the two jobs complete back-to-back. In Figure 5.9(b), the first job completes as early as possible and the second as late as possible. As we see, any completion delay measurement for a job J_i of a schedulable periodic task set may vary on the interval $[e_i, 2p_i - e_i]$. Of course, this interval may grow if the first of two consecutive jobs executes for less than e_i time units. The upper-bound (*i.e.*, $2p_i - e_i$) may also increase if we consider task sets that are schedulable with bounded tardiness. Despite these limitations, we feel that observed completion delays are useful in reasoning about the real-time runtime performance of our various system configurations.

What makes a "good" completion delay measurement? In these experiments, we desire smooth playback of stabilized video—we want completion delays that are close to the graph period. We also want to see little variation in completion delays. We measure variation by computing the standard deviation (σ) of measured completion delays of each graph.

Tables 5.3 through 5.13 report the characteristics of the completion delays for each graph under each labeled configuration. The tables include columns for the *maximum*, 99.9*th percentile*, 99*th percentile*, *median*, and *mean* observed completion delay for each graph. The tables also include the standard deviation for each mean. The tables also include a column reporting the percentage of video frames that a graph could not complete in the allotted time. For example, in a 400 second experiment, a graph with a period of 20*ms* should complete 20,000 frames. If it only completes 15,000 frames, then we say that the graph has dropped 5,000 frames, or 25%. A graph with dropped frames is indicative of processor starvation. This gives us another method by which to detect an unschedulable task set observationally. We make the following observations.

Observation 28. *The task set is clearly unschedulable under the* SCHED_FIFO*, and GPUSync with* $\rho \in \{1, 2\}$ *configurations.*

We look at the percentage of dropped frames to detect unschedulability. Each system configuration exhibits different characteristics as to which graphs dropped frames.

In Table 5.4, we see that the graphs with the longest periods (100*ms*), *i.e.*, those with the lowest fixed priorities, dropped many frames under SCHED_FIFO. For instance, G_{16} dropped 88.25% of the 4,000 frames that should have been processed. This sort of behavior is characteristic of fixed-priority schedulers—the lowest priority tasks may be starved of CPU time.

In Tables 5.6 and 5.7, we see that *many* frames are dropped by all tasks under GPUSync with $\rho = 1$.[17] The percentage of dropped frames increases with period. For example, G_1 in Table 5.6 drops 25.41% of its

[17]Since $\rho = 1$, these configurations are theoretically equivalent since a token-holding job immediately receives every engine lock it requests. That is, the token grants exclusive access to the GPU. However, we do see some minor variation. These variations may be simply due to experimental noise or differences in runtime overheads due to different locking logic behind FIFO and PRIO engine locks.

frames, while G_{17} drops about 47.33%. Although performance is generally bad under these configurations, we do observe that the number of dropped frames is strongly correlated with graph period. That is, performance is very regular. For instance, the five graphs in Table 5.7 with a 60*ms* period (graphs G_7 through G_{11}) each drop either 44.76% or 44.78% of frames—they all exhibit nearly the same behavior. Similar regularity is reflected by the median and mean completion delays for these tasks.

In Tables 5.8 and 5.9, we see that performance improves with a greater number of GPU tokens, where $\rho = 2$. However, frames are still dropped. We observe a similar regularity in the percentage of dropped frames as we saw in Tables 5.6 and 5.7 (GPUSync with $\rho = 1$).

Observation 29. *GPUSync, with $\rho \in \{3,6\}$, improved observed predictability.*

We can observe this by looking at several different metrics. We first compare GPUSync with $\rho \in \{3,6\}$ (Tables 5.10 through 5.13) against LITMUS^{RT} without GPUSync (Table 5.5). These system configurations all share the same CPU scheduler (C-EDF), yet there are clear differences in runtime behavior.

We observe significant differences in terms of outlier behavior. For example, in Table 5.5 for LITMUS^{RT} without GPUSync, G_7 has a maximum, 99.9*th* percentile, and 99*th* percentile completion delays of 1108.68*ms*, 776.63*ms*, and 465.71*ms*, respectively. In Table 5.13 for GPUSync with $(6, 1, \mathsf{PRIO})$, G_7 has a maximum, 99.9*th* percentile, and 99*th* percentile completion delays of 224.33*ms*, 192.2*ms*, 88.0*ms*, respectively. The observed completion delays for G_7 under GPUSync with $(6, 1, \mathsf{PRIO})$ were reduced by factors of approximately 4.9, 4.0, and 5.3 for maximum, 99.9*th* percentile, and 99*th* percentile measurements, respectively. We see similar improvements when we consider GPUSync with $(3, 2, \mathsf{FIFO})$, $(3, 2, \mathsf{PRIO})$, and $(6, 1, \mathsf{FIFO})$.

We also compare the behavior of GPUSync with $\rho \in \{3,6\}$ against LITMUS^{RT} without GPUSync in terms of the standard deviation of completion delays. In Table 5.5 for LITMUS^{RT} without GPUSync, the standard deviations are on the range $[15.47ms, 130.21ms]$. In Table 5.10 for GPUSync with $(3, 2, \mathsf{FIFO})$, the range is $[6.21ms, 10.07ms]$. In Table 5.11 for GPUSync with $(3, 2, \mathsf{PRIO})$, the range is $[6.32ms, 10.55ms]$. In Table 5.12 for GPUSync with $(6, 1, \mathsf{FIFO})$, the range is $[8.11ms, 14.65ms]$. In Table 5.13 for GPUSync with $(6, 1, \mathsf{PRIO})$, the range is $[10.64ms, 16.94ms]$. With the exception of GPUSync with $(6, 1, \mathsf{PRIO})$, the *greatest* standard deviations of these GPUSync configurations are less than the *smallest* standard deviation under LITMUS^{RT} without GPUSync.

From these observations, we conclude that *real-time CPU scheduling alone is not enough to ensure real-time perceptibility in a system with GPUs.*

We make similar comparisons of GPUSync with $\rho \in \{3,6\}$ against $\mathsf{SCHED\_OTHER}$ (Table 5.3), and we also observe significant differences in terms of outlier behavior. For example, in Table 5.3 for $\mathsf{SCHED\_OTHER}$, G_1 has a maximum, 99.9^{th} percentile, and 99^{th} percentile completion delays of $405.88ms$, $233.32ms$, and $54.61ms$, respectively. In Table 5.10 for GPUSync with $(3,2,\mathsf{FIFO})$, G_1 has a maximum, 99.9^{th} percentile, and 99^{th} percentile completion delays of $115.11ms$, $83.21ms$, $33.81ms$, respectively. GPUSync with $(3,2,\mathsf{FIFO})$ reduces maximum, 99.9^{th} percentile, and 99^{th} percentile completion delays by factors of approximately 3.5, 2.8, and 1.6 times, respectively. We see similar improvements when we consider GPUSync with $(3,2,\mathsf{PRIO})$, $(6,1,\mathsf{FIFO})$, and $(6,1,\mathsf{PRIO})$.

We may also compare the behavior of GPUSync with $\rho \in \{3,6\}$ against $\mathsf{SCHED\_OTHER}$ in terms of the standard deviation of completion delays. In Table 5.3 for $\mathsf{SCHED\_OTHER}$, the standard deviations are on the range $[13.90ms, 18.36ms]$. In Table 5.10 for GPUSync with $(3,2,\mathsf{FIFO})$, the range is $[6.21ms, 10.07ms]$. In Table 5.11 for GPUSync with $(3,2,\mathsf{PRIO})$, the range is $[6.32ms, 10.55ms]$. In Table 5.12 for GPUSync with $(6,1,\mathsf{FIFO})$, the range is $[8.11ms, 14.65ms]$. In Table 5.13 for GPUSync with $(6,1,\mathsf{PRIO})$, the range is $[10.64ms, 16.94ms]$. Although the standard deviation ranges among these GPUSync configurations differ, they are less than that of $\mathsf{SCHED\_OTHER}$. Indeed, the greatest standard deviations when $\rho = 3$ are less than the smallest standard deviation under $\mathsf{SCHED\_OTHER}$.

Observation 30. *In this experiment, the $\mathsf{SCHED\_OTHER}$ configuration outperformed both real-time configurations that lack real-time GPU management.*

A surprising result from these experiments is that the $\mathsf{SCHED\_OTHER}$ configuration (Table 5.3) outperforms both $\mathsf{SCHED\_FIFO}$ (Table 5.4) and LITMUS^{RT} without GPUSync (Table 5.5) configurations. The $\mathsf{SCHED\_FIFO}$ configuration dropped frames while the $\mathsf{SCHED\_OTHER}$ configuration did not. The LITMUS^{RT} without GPUSync configuration has worse outlier behavior than the $\mathsf{SCHED\_OTHER}$ configuration.

These differences in behavior may be due to busy-waiting employed by the CUDA runtime or VisionWorks software, despite the fact that libgpui forces tasks to suspend while waiting for GPU operations to complete. Under $\mathsf{SCHED\_FIFO}$ and LITMUS^{RT} without GPUSync configurations, the CPU scheduler always schedules the m-highest priority tasks (in this experiment, $m = 6$) that are ready to run. CPU time may be wasted if any of these tasks busy-wait for a long duration of time. The amount of CPU time wasted by a task at the expense of other tasks is limited under $\mathsf{SCHED\_OTHER}$, since the scheduler attempts to distribute CPU time equally among all tasks. That is, the scheduler will preempt a task that busy-waits for too long.

This assumes that the tasks use *preemptive* busy-waiting, but this would be expected of software developed for general purpose computing.[18] Interestingly, the fact that the SCHED_FIFO and LITMUS^{RT} without GPUSync configurations do not deadlock suggests that any such busy-waiting is probably not used to implement a locking protocol. (Preemptive busy-waiting in a locking protocol under real-time scheduling can easily lead to deadlock.)

It is difficult to draw general conclusions from the completion delay data presented in Tables 5.3 through 5.13 because there is so much data. To help us gain additional insights into the performance of the eleven system configurations, we collapsed the information in the above eleven tables into a single table, Table 5.14. We collapse the data with the following process. We compute the total percentage of dropped frames by summing the number of dropped frames of all graphs under a given system configuration and compute its share of the total number of frames that should have been completed within the allotted time. To combine the completion delay data, we first *normalized* each measurement by dividing the measurements by the graph period. We then compute the average of the normalized values. (We use the term "average" to differentiate from the completion delay means that we discuss.) For example, the "Max" for SCHED_OTHER in Table 5.14 reflects the average normalized-maximum of the "Max" values in Table 5.3. Likewise, the standard deviation of GPUSync with $(1, 6, \text{PRIO})$ in Table 5.14 reflects an average of normalized standard

[18]Non-preemptive execution generally requires privileged permissions (*e.g.*, "superuser" permissions).

Graph	Period	% Dropped	Max	$99.9^{th}\%$	$99^{th}\%$	Median	Mean	σ
G_1	20	0	405.88	233.32	54.61	15.94	20.00	15.85
G_2	20	0	412.24	248.62	53.06	16.12	20.00	15.57
G_3	30	0	377.97	246.86	62.68	28.34	29.99	17.33
G_4	30	0	399.43	240.92	64.64	28.08	30.00	17.71
G_5	40	0	397.06	291.17	71.02	40.03	40.00	17.75
G_6	40	0	452.15	295.14	69.62	39.94	39.99	18.13
G_7	60	0	473.23	376.42	82.87	60.07	59.99	17.68
G_8	60	0	453.20	367.44	85.32	60.47	59.99	17.91
G_9	60	0	414.78	352.29	86.16	60.36	59.99	17.14
G_{10}	60	0	462.29	372.52	84.26	60.50	59.99	17.91
G_{11}	60	0	454.22	331.98	87.79	59.99	59.99	17.66
G_{12}	80	0	485.56	338.03	107.36	79.86	79.98	17.68
G_{13}	80	0	442.45	340.89	105.01	79.87	79.98	16.90
G_{14}	80	0	391.75	349.92	105.96	79.78	79.98	16.48
G_{15}	80	0	486.90	351.47	106.70	79.62	80.00	18.36
G_{16}	100	0	440.83	256.71	127.19	100.00	99.99	15.48
G_{17}	100	0	376.85	273.21	127.43	99.92	99.98	13.90

Table 5.3: Completion delay data for SCHED_OTHER. Time in *ms*.

Graph	Period	% Dropped	Max	99.9th%	99th%	Median	Mean	σ
G_1	20	0	99.25	96.09	25.68	20.26	20.00	4.78
G_2	20	0	99.29	95.93	25.86	20.34	20.00	4.88
G_3	30	0	115.89	105.92	38.39	28.93	30.00	6.60
G_4	30	0	116.27	105.41	38.72	28.90	30.00	6.81
G_5	40	0	184.15	140.43	45.91	38.92	40.00	6.73
G_6	40	0	184.30	142.32	45.89	38.94	40.00	6.69
G_7	60	0	533.52	387.16	96.62	59.22	59.98	21.72
G_8	60	0	456.71	351.90	96.87	59.58	60.00	20.61
G_9	60	0	519.93	347.87	94.97	59.24	59.99	20.36
G_{10}	60	0	463.48	346.12	91.02	59.76	59.99	20.61
G_{11}	60	0	396.98	353.92	96.65	59.63	59.99	19.86
G_{12}	80	0	904.76	770.43	204.38	66.19	79.98	49.61
G_{13}	80	0	964.77	729.72	213.20	66.37	79.95	49.63
G_{14}	80	0	903.07	771.49	202.04	66.64	79.93	48.67
G_{15}	80	0	936.06	717.49	233.15	66.59	79.94	49.44
G_{16}	100	88.25	3786.02	3786.02	2279.03	920.63	850.65	559.04
G_{17}	100	88.25	3844.70	3844.70	2333.78	879.83	848.36	580.58

Table 5.4: Completion delay data for SCHED_FIFO. Time in *ms*.

Graph	Period	% Dropped	Max	99.9th%	99th%	Median	Mean	σ
G_1	20	0	612.16	300.41	159.49	4.62	20.00	34.69
G_2	20	0	254.20	143.12	62.49	19.30	20.00	15.47
G_3	30	0	309.02	242.21	91.72	27.21	30.00	22.61
G_4	30	0	272.94	170.40	73.96	27.71	30.00	19.36
G_5	40	0	450.22	295.02	135.39	37.88	40.00	31.04
G_6	40	0	497.19	335.93	154.33	36.91	40.00	38.84
G_7	60	0	1108.68	776.63	465.71	43.80	59.96	85.33
G_8	60	0	785.02	596.92	256.90	56.42	59.96	67.26
G_9	60	0	459.19	373.74	152.66	58.97	59.95	33.11
G_{10}	60	0	383.78	307.72	143.46	58.83	59.95	25.82
G_{11}	60	0	383.59	307.70	143.25	58.81	59.95	25.81
G_{12}	80	0	703.36	543.54	257.50	70.84	79.86	65.42
G_{13}	80	0	861.65	652.05	315.17	69.61	79.86	76.18
G_{14}	80	0	829.90	601.26	264.91	70.27	79.86	70.62
G_{15}	80	0	865.79	666.63	363.45	69.41	79.86	79.94
G_{16}	100	0	1278.93	1077.55	641.47	79.46	99.78	124.68
G_{17}	100	0	1416.51	1014.76	711.54	79.27	99.78	130.21

Table 5.5: Completion delay data for LITMUS^{RT} without GPUSync. Time in *ms*.

Graph	Period	% Dropped	Max	99.9th%	99th%	Median	Mean	σ
G_1	20	25.41	413.27	200.79	43.21	25.48	26.90	14.83
G_2	20	25.41	406.52	200.29	43.57	25.47	26.90	15.09
G_3	30	36.42	422.43	345.47	113.72	45.70	47.35	18.18
G_4	30	36.42	422.15	270.20	117.96	45.75	47.34	17.43
G_5	40	41.23	435.51	364.95	200.15	66.22	68.29	20.48
G_6	40	41.24	436.16	356.91	204.60	66.11	68.29	20.16
G_7	60	44.82	462.04	454.50	238.66	105.89	109.09	24.54
G_8	60	44.48	460.74	453.70	238.53	105.84	108.58	25.26
G_9	60	44.48	458.31	450.14	238.97	106.00	108.58	25.21
G_{10}	60	44.48	451.62	442.64	238.14	105.95	108.58	25.28
G_{11}	60	44.49	455.47	448.70	241.87	105.83	108.60	25.67
G_{12}	80	46.16	492.02	479.68	285.07	145.90	149.23	29.64
G_{13}	80	46.16	491.48	479.88	281.86	145.69	149.23	29.51
G_{14}	80	46.16	492.47	473.16	281.49	145.84	149.22	28.73
G_{15}	80	46.16	471.79	462.25	285.98	145.79	149.22	29.47
G_{16}	100	47.33	519.71	498.72	329.67	186.90	190.60	33.30
G_{17}	100	47.33	543.10	508.17	324.20	186.96	190.59	33.38

Table 5.6: Completion delay data for GPUSync for $(1, 6, \mathsf{FIFO})$. Time in *ms*.

Graph	Period	% Dropped	Max	$99.9^{th}\%$	$99^{th}\%$	Median	Mean	σ
G_1	20	25.3	407.33	200.10	43.65	25.22	26.87	14.89
G_2	20	25.3	357.44	199.38	43.77	25.79	26.87	14.81
G_3	30	36.35	418.08	276.83	119.78	45.81	47.32	17.38
G_4	30	36.35	420.18	282.08	113.06	45.77	47.32	18.07
G_5	40	41.18	429.04	356.54	203.34	66.12	68.27	20.12
G_6	40	41.17	436.99	352.97	201.51	66.29	68.26	20.07
G_7	60	44.76	462.70	452.84	237.07	105.94	109.00	24.34
G_8	60	44.76	458.72	454.74	239.81	105.86	109.00	24.54
G_9	60	44.76	451.96	450.42	239.74	105.90	109.00	24.48
G_{10}	60	44.78	452.22	444.57	237.70	105.96	109.03	24.37
G_{11}	60	44.78	455.94	449.44	239.17	105.81	109.03	24.68
G_{12}	80	46.44	485.87	473.34	285.59	145.74	149.81	27.96
G_{13}	80	46.44	495.53	479.42	279.90	145.98	149.81	27.58
G_{14}	80	46.44	499.83	478.61	282.22	145.82	149.81	28.22
G_{15}	80	46.44	544.97	474.02	282.61	145.84	149.81	28.30
G_{16}	100	47.63	522.80	515.44	328.67	187.30	191.44	31.31
G_{17}	100	47.63	566.30	515.94	327.71	187.12	191.44	31.92

Table 5.7: Completion delay data for GPUSync for $(1, 6, \mathsf{PRIO})$. Time in *ms*.

Graph	Period	% Dropped	Max	99.9th%	99th%	Median	Mean	σ
G_1	20	0.99	174.27	93.66	30.40	19.84	20.29	6.00
G_2	20	1.06	124.28	96.09	30.85	19.86	20.31	6.09
G_3	30	1.14	180.10	111.63	43.56	30.00	30.49	7.09
G_4	30	1.15	178.34	108.24	44.06	29.96	30.49	7.00
G_5	40	1.37	183.82	120.26	66.28	40.05	40.76	7.64
G_6	40	1.33	206.13	122.24	61.22	40.05	40.74	7.62
G_7	60	7.32	218.13	170.21	109.48	63.10	65.03	10.55
G_8	60	7.28	215.98	162.14	105.90	63.04	65.00	10.26
G_9	60	7.35	216.59	162.00	112.48	63.11	65.04	10.54
G_{10}	60	7.29	218.09	152.88	109.10	63.02	65.00	10.32
G_{11}	60	7.31	217.27	167.35	109.57	63.11	65.01	10.35
G_{12}	80	22.66	248.40	217.49	162.55	102.92	103.85	13.80
G_{13}	80	22.64	250.84	208.36	156.96	103.03	103.82	13.55
G_{14}	80	22.58	250.55	208.46	155.46	102.80	103.72	13.53
G_{15}	80	22.64	246.19	223.89	158.22	102.86	103.80	14.05
G_{16}	100	30.08	275.23	240.95	208.00	142.62	143.53	15.73
G_{17}	100	30.05	244.98	229.37	200.21	142.66	143.46	15.31

Table 5.8: Completion delay data for GPUSync for $(2,3,\mathsf{FIFO})$. Time in *ms*.

Graph	Period	% Dropped	Max	99.9th%	99th%	Median	Mean	σ
G_1	20	1.12	188.25	102.50	30.83	19.88	20.32	6.36
G_2	20	1.07	178.11	105.39	30.85	19.85	20.31	6.40
G_3	30	1.13	197.98	111.49	43.57	29.97	30.49	7.23
G_4	30	1.19	162.21	115.02	44.53	29.98	30.51	7.30
G_5	40	1.42	191.41	133.79	63.56	39.98	40.78	8.30
G_6	40	1.47	214.40	126.83	65.99	40.02	40.80	8.09
G_7	60	7.89	216.31	193.13	108.98	63.35	65.43	10.87
G_8	60	7.89	218.85	190.42	105.69	63.41	65.43	10.75
G_9	60	7.86	222.21	186.86	111.05	63.51	65.41	10.70
G_{10}	60	7.95	214.20	170.20	114.40	63.54	65.47	10.78
G_{11}	60	7.89	226.12	197.13	108.45	63.47	65.43	10.79
G_{12}	80	23.08	255.15	222.58	157.23	103.61	104.41	13.77
G_{13}	80	23.08	238.12	228.92	154.70	103.48	104.41	13.98
G_{14}	80	23.06	248.43	229.53	154.51	103.64	104.39	13.72
G_{15}	80	23.04	261.14	238.38	154.46	103.40	104.35	14.16
G_{16}	100	30.25	308.29	261.81	204.78	142.94	143.88	15.95
G_{17}	100	30.30	275.49	256.45	210.65	142.91	143.98	16.34

Table 5.9: Completion delay data for GPUSync for $(2,3,\mathsf{PRIO})$. Time in *ms*.

Graph	Period	% Dropped	Max	$99.9^{th}\%$	$99^{th}\%$	Median	Mean	σ
G_1	20	0	115.11	83.21	33.81	19.56	20.01	6.21
G_2	20	0	154.24	89.77	32.45	19.83	20.01	6.41
G_3	30	0	119.40	91.59	44.81	29.55	30.01	6.26
G_4	30	0	119.96	90.32	45.51	29.60	30.01	6.48
G_5	40	0	154.99	93.90	56.19	39.60	40.02	6.30
G_6	40	0	176.54	95.40	55.83	39.70	40.01	6.54
G_7	60	0	299.21	118.69	78.54	59.51	60.04	7.09
G_8	60	0	209.89	125.10	79.49	59.68	60.04	7.10
G_9	60	0	246.73	112.11	79.03	59.57	60.03	6.98
G_{10}	60	0	175.79	130.78	80.90	59.72	60.04	7.34
G_{11}	60	0	205.56	122.91	79.30	59.60	60.03	6.66
G_{12}	80	0	152.52	141.36	105.04	79.44	80.01	8.19
G_{13}	80	0	198.76	141.15	104.87	79.63	80.01	7.93
G_{14}	80	0	176.11	142.24	106.64	79.55	80.01	8.42
G_{15}	80	0	226.29	152.43	103.39	79.55	80.02	8.52
G_{16}	100	0	209.92	172.15	129.92	99.57	100.07	10.07
G_{17}	100	0	209.91	170.36	131.09	99.59	100.07	9.85

Table 5.10: Completion delay data for GPUSync for $(3, 2, \mathsf{FIFO})$. Time in *ms*.

Graph	Period	% Dropped	Max	$99.9^{th}\%$	$99^{th}\%$	Median	Mean	σ
G_1	20	0	100.51	86.71	33.42	19.09	20.01	6.32
G_2	20	0	171.95	82.87	34.49	19.37	20.01	6.50
G_3	30	0	163.08	91.38	44.38	29.25	30.01	6.84
G_4	30	0	119.83	95.17	44.77	29.62	30.01	6.62
G_5	40	0	216.41	101.41	55.15	40.46	40.02	7.35
G_6	40	0	238.03	95.13	56.15	39.73	40.02	7.60
G_7	60	0	215.71	120.58	81.55	59.57	60.00	6.70
G_8	60	0	295.60	122.66	80.61	59.61	60.01	7.99
G_9	60	0	210.39	121.58	81.13	59.50	60.01	7.37
G_{10}	60	0	182.70	122.54	79.40	59.59	59.99	6.21
G_{11}	60	0	142.32	119.76	78.38	59.58	59.98	6.08
G_{12}	80	0	175.73	146.69	106.27	79.47	80.00	8.65
G_{13}	80	0	163.97	148.16	109.05	79.78	80.00	8.94
G_{14}	80	0	170.10	147.12	109.53	79.56	80.00	9.93
G_{15}	80	0	218.05	152.67	109.07	79.62	80.01	9.59
G_{16}	100	0	208.42	168.30	139.57	99.89	100.07	10.55
G_{17}	100	0	201.03	163.87	135.47	99.80	100.06	10.13

Table 5.11: Completion delay data for GPUSync for $(3, 2, \mathsf{PRIO})$. Time in *ms*.

Graph	Period	% Dropped	Max	$99.9^{th}\%$	$99^{th}\%$	Median	Mean	σ
G_1	20	0	84.66	69.29	39.55	18.56	20.01	8.11
G_2	20	0	100.91	71.41	40.40	18.31	20.01	8.28
G_3	30	0	106.65	78.04	49.33	29.45	30.02	9.32
G_4	30	0	117.70	67.74	49.16	29.97	30.02	9.33
G_5	40	0	188.84	87.52	58.98	40.21	40.02	9.21
G_6	40	0	242.19	80.76	58.93	40.09	40.03	9.14
G_7	60	0	378.94	120.78	83.18	60.86	60.03	9.42
G_8	60	0	378.56	122.67	82.59	60.77	60.03	8.89
G_9	60	0	378.50	120.32	84.30	60.93	60.03	10.73
G_{10}	60	0	304.69	123.79	85.33	60.45	60.02	9.47
G_{11}	60	0	163.24	126.17	89.90	60.42	59.99	10.48
G_{12}	80	0	166.87	135.46	115.25	78.99	79.95	11.93
G_{13}	80	0	189.56	151.96	115.58	78.77	79.95	12.50
G_{14}	80	0	173.29	146.28	112.74	78.33	79.95	12.25
G_{15}	80	0	170.68	148.38	105.88	78.12	79.95	10.10
G_{16}	100	0	242.56	197.47	139.18	100.00	99.98	14.51
G_{17}	100	0	240.90	209.31	140.24	100.09	100.00	14.65

Table 5.12: Completion delay data for GPUSync for $(6,1,\mathsf{FIFO})$. Time in *ms*.

Graph	Period	% Dropped	Max	$99.9^{th}\%$	$99^{th}\%$	Median	Mean	σ
G_1	20	0	126.79	113.19	44.66	16.83	20.01	10.64
G_2	20	0	136.47	118.97	45.99	17.26	20.01	11.56
G_3	30	0	156.19	120.81	52.68	29.89	30.01	11.72
G_4	30	0	141.65	113.71	53.14	30.12	30.01	11.68
G_5	40	0	174.46	102.46	60.99	41.35	40.01	11.76
G_6	40	0	177.19	135.44	61.72	41.84	40.02	12.68
G_7	60	0	224.33	192.20	88.00	60.66	59.99	11.08
G_8	60	0	224.36	184.27	88.33	60.66	59.99	11.09
G_9	60	0	197.53	164.71	88.50	60.72	59.99	10.96
G_{10}	60	0	215.30	169.92	89.38	60.64	59.99	11.16
G_{11}	60	0	214.23	184.25	89.54	60.71	59.99	11.56
G_{12}	80	0	237.41	199.40	116.73	77.81	79.96	15.12
G_{13}	80	0	216.04	178.55	116.64	77.79	79.93	13.55
G_{14}	80	0	224.25	181.11	116.91	77.82	79.93	13.98
G_{15}	80	0	237.55	214.03	116.46	77.25	79.93	14.11
G_{16}	100	0	286.73	267.92	147.47	100.18	99.98	16.94
G_{17}	100	0	257.77	234.21	139.80	100.26	99.97	16.24

Table 5.13: Completion delay data for GPUSync for $(6,1,\mathsf{PRIO})$. Time in *ms*.

Configuration	Total % Dropped	Normalized and Averaged					
		Max	$99.9^{th}\%$	$99^{th}\%$	Median	Mean	σ
SCHED_OTHER	0	9.20	6.33	1.65	0.97	**1.00**	0.37
SCHED_FIFO	4.77	11.12	9.83	4.25	1.89	1.88	0.99
LITMUS^{RT} C-EDF	0	12.12	8.40	4.22	0.86	**1.00**	0.93
GPUSync							
$(1,6,\mathsf{FIFO})$	37.83	9.65	7.64	3.70	1.69	1.74	0.47
$(1,6,\mathsf{PRIO})$	37.88	9.53	7.53	3.70	1.69	1.74	0.46
$(2,3,\mathsf{FIFO})$	6.99	4.26	3.05	1.78	1.13	1.15	0.20
$(2,3,\mathsf{PRIO})$	7.23	4.54	3.34	1.78	1.14	1.16	0.20
$(3,2,\mathsf{FIFO})$	0	**3.67**	2.37	**1.39**	**0.99**	**1.00**	**0.15**
$(3,2,\mathsf{PRIO})$	0	3.83	2.38	1.41	**0.99**	**1.00**	0.16
$(6,1,\mathsf{FIFO})$	0	3.99	**2.22**	1.51	**0.99**	**1.00**	0.21
$(6,1,\mathsf{PRIO})$	0	3.92	3.23	1.60	0.98	**1.00**	0.26

Table 5.14: Average normalized completion delay data.

deviations from Table 5.13. In Table 5.14, we also highlight the "best" values in each column. We consider values closest to 1.0 as best for average normalized completion delays, and values closest to zero as best for standard deviations. We make the following additional observations.

Observation 31. *For GPUSync configurations, completion delays are more regular when* $\rho = 3$.

In Table 5.14, we observe that the average normalized maximum completion delay is smallest under GPUSync with $(3,2,\mathsf{FIFO})$. The next smallest is the GPUSync $(3,2,\mathsf{PRIO})$ configuration. The same holds for the average normalized 99^{th} percentile completion delay and standard deviation. We note that the GPUSync with $(6,1,\mathsf{FIFO})$ configuration exhibited the best average normalized 99.9^{th} percentile completion delay. However, given that this configuration is beaten or matched by GPUSync configurations with $\rho = 3$ in other measures, we assert that $\rho = 3$ still gives the best real-time performance overall.

It appears that $\rho = 3$ is the "sweet spot" for ρ in this experiment. The experimental workload is unschedulable when $\rho = 2$, yet real-time performance worsens when $\rho = 6$. The likelihood that GPU resources are left idle is greater with smaller values of ρ. When $\rho = 1$, two of the GPU's three engines (one EE and two CEs) are guaranteed to be left idle. Similarly, at least one engine will always be idle when $\rho = 2$. Engines may still be left idle when $\rho = 3$, but the *possibility* remains for all engines to be used simultaneously. However, this line of reasoning fails when we consider the case when $\rho = 6$. In Section 3.2.3.2, we argued that we should constrain the number of GPU tokens in order to limit the aggressiveness of migrations. This is no longer a concern in a single-GPU system, so why does performance not continue to improve when $\rho = 6$? We provide the following possible explanation.

In this experiment, we executed the *same amount of work* under GPUSync configurations with $\rho = 3$ and $\rho = 6$. More tokens allows a finite amount work (on both CPUs and GPU engines) to be *shifted* to an earlier point in time of the schedule, since the CPU scheduler and locking protocols are work-conserving. Here, the shifted work "piles up" at an earlier point in the schedule. From Table 5.14, we see that $\rho = 3$ provides sufficient GPU parallelism to achieve good real-time performance; the constrained number of tokens meters out GPU access. However, when $\rho = 6$, bursts in GPU activity, where all GPU engines are used simultaneously, become more likely. These bursts may result in moments of heavy interference (particularly on the system memory bus) with other tasks, thus increasing *variance* in completion delays. This interference would only increase the likelihood of outlier behavior on both ends of the spectrum of observed completion delays, since bursts in GPU activity would be followed by corresponding lulls. We theorize that median-case performance would remain unaffected. This explanation is borne out by the data in Table 5.14, where we see that the average normalized median completion delays under $\rho = 6$ are generally indistinguishable from configurations where $\rho = 3$.

Observation 32. *For GPUSync configurations, FIFO engine locks offered better observed predictability than PRIO engine locks.*

To see this, we compare the GPUSync configurations that only differ by engine lock protocol. With exception of configurations where $\rho = 1$ (which we discuss shortly), we see that the average normalized standard deviations of the mean completion delays are less (or equal in the case of $\rho = 2$) under FIFO. FIFO-ordered engine locks impart very regular blocking behaviors—any single engine request may only be delayed between zero and $\rho - 1$ other engine requests. Under priority-ordered locks, higher-priority requests may continually "cut ahead" of a low-priority engine request. Thus, the low-priority engine request may be delayed by more than $\rho - 1$ requests. A job with such requests may still meet its deadline, but there may be increased variance in completion delays due to increased variance in blocking delays.

Observation 33. *GPUSync with $(3, 2, \mathsf{FIFO})$ outperformed the other ten system configurations.*

In Table 5.14, we see that the GPUSync configuration with $(3, 2, \mathsf{FIFO})$ produced the smallest Max, 99^{th}, and standard deviation values among all configurations. On average, the normalized standard deviation of the mean completion delays was only 15% (0.15 in Table 5.14) of a graph's period. Compare this to 37%, 99%, and 93% under SCHED_OTHER, SCHED_FIFO, and LITMUS^{RT} without GPUSync, respectively. Other GPUSync configurations where $\rho \in \{3, 6\}$ were competitive, but none performed as well as $(3, 2, \mathsf{FIFO})$.

The above tables give us insight into worst-case and average-case behaviors of the tested system configurations. However, the distribution of observed completion delays is somewhat obscured. To gain deeper insights into these distributions, we plot the PDF of normalized completion delays in Figures 5.10 through 5.20. Each plot is derived from a histogram with a bucket width of 0.005. In each of these figures, the x-axis denotes a normalized completion delay. The y-axis denotes a probability density. To determine the probability that a normalized completion delay falls within the domain $[a, b]$, we sum the area under the curve between $x = a$ and $x = b$; the total area under each curve is 1.0. Generally, distributions with the greatest area near 1.0 are best.

Our goal is to understand the *shape* of completion delay distributions, so each distribution is plotted on the same domain and range. We also *clip* the domain at $x = 2.5$, so the long tails of these distributions are not depicted. However, we have examined worst-case behaviors in the prior tables, so we do not revisit the topic here. We make several observations.

Observation 34. *The PDFs for GPUSync with* $\rho = 3$ *show that normalized completion delays are most likely near* 1.0*.*

We see this in Figures 5.17 and 5.18 for GPUSync with $\rho = 3$ for FIFO and PRIO engine locks, respectively. This result is not surprising, given prior Observation 31. However, in these PDFs we also see that they closely resemble the curve of a normal distribution, more so than any PDF of the other configurations. The PDFs for GPUSync with $\rho = 6$ (Figures 5.19 and 5.20), and even the PDF for the SCHED_OTHER configuration (Figure 5.10), have similar shapes, but they are not as strongly centered around 1.0. Also, they do not exhibit the same degree of symmetry.

Another characteristic of the PDFs for $\rho = 3$ is that they are clearly unimodal. This is unlike the PDF in Figure 5.11 for the SCHED_FIFO configuration, which has at least four distinct modes (indicated by the four peaks in the PDF), or the PDF in Figure 5.12 for LITMUS^{RT} without GPUSync, which appears to be bimodal.

Observation 35. *The PDF for* LITMUS^{RT} *without GPUSync suggests bursty completion behaviors.*

Figure 5.12 depicts the PDF for the LITMUS^{RT} without GPUSync configuration. The PDF appears to have two modes. One mode is near $x = 0.05$; the other is centered around 1.0. Not depicted in this figure is the long tail of the PDF.

When a job of a sink node of the video stabilization graph completes late, work can “back up” within the graph. Under the PGM early releasing policy of non-source nodes, a sink node with backed-up work may

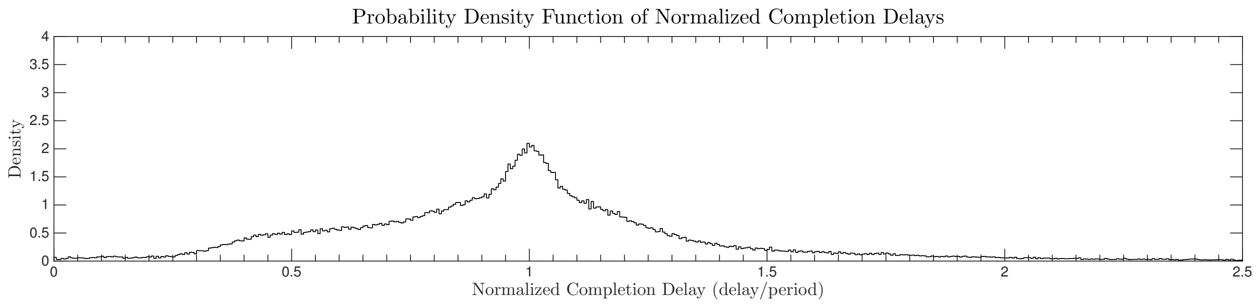


Figure 5.10: PDF of normalized completion delay data for SCHED_OTHER.

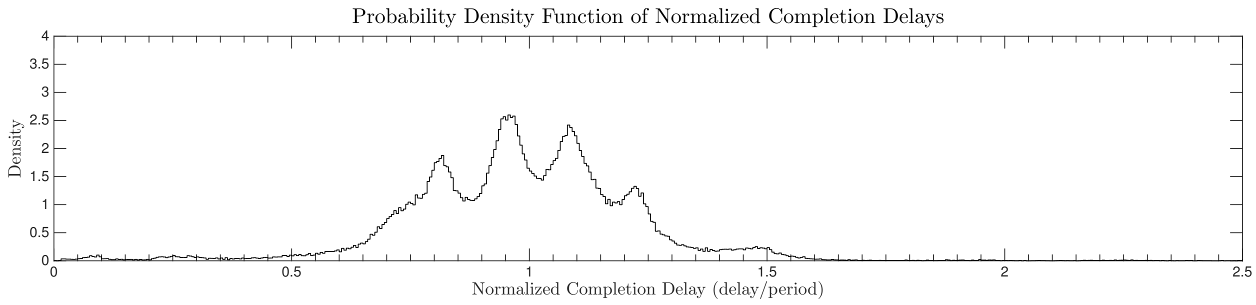


Figure 5.11: PDF of normalized completion delay data for SCHED_FIFO.

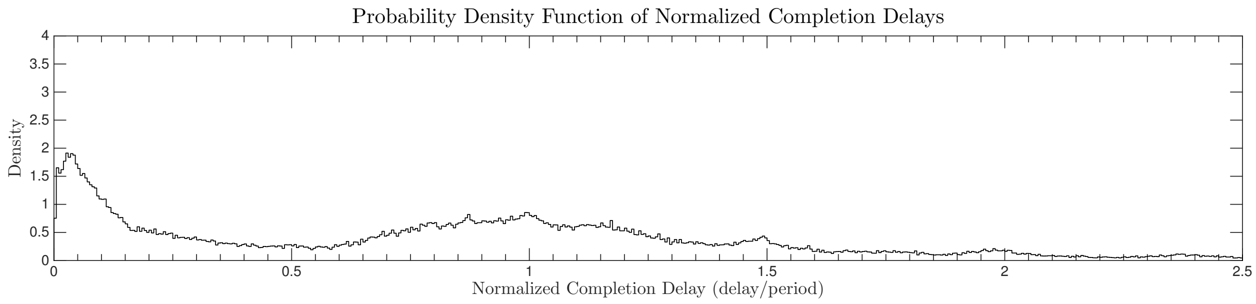


Figure 5.12: PDF of normalized completion delay data for LITMUS^{RT} without GPUSync.

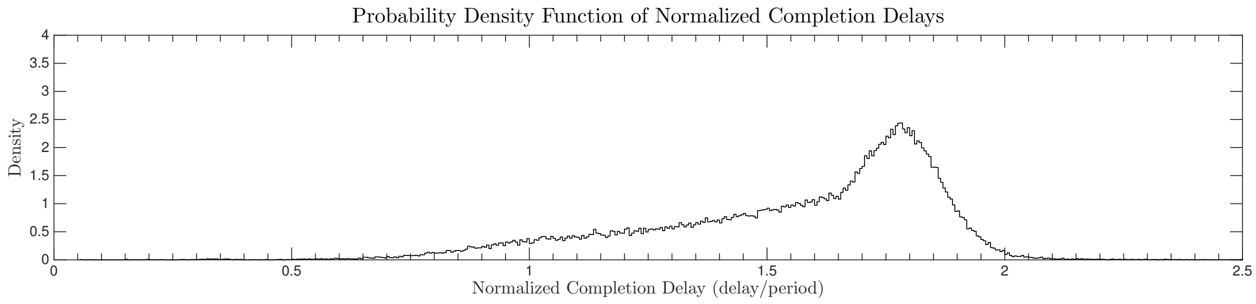


Figure 5.13: PDF of normalized completion delay data for GPUSync with $(1, 6, \mathsf{FIFO})$.

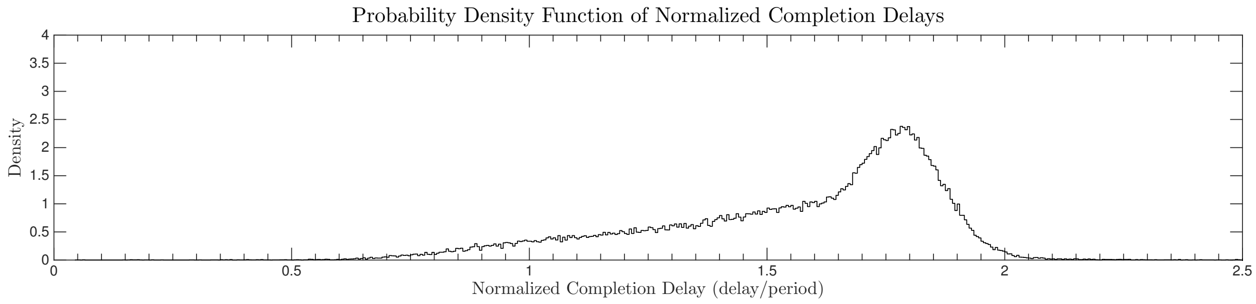


Figure 5.14: PDF of normalized completion delay data for GPUSync with $(1, 6, \mathsf{PRIO})$.

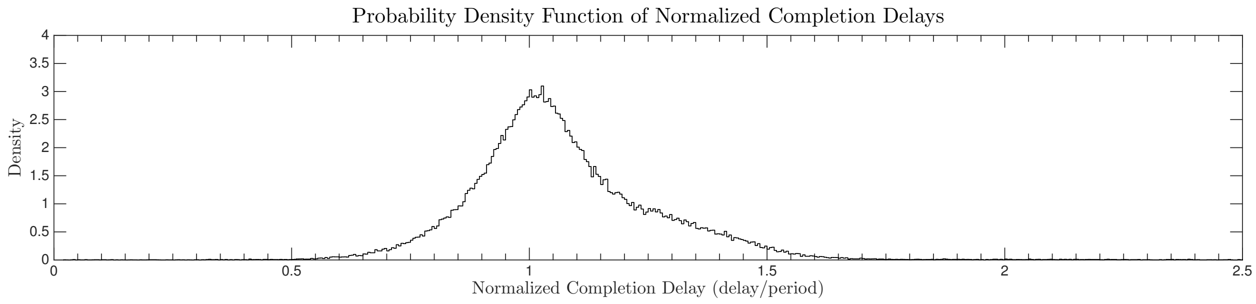


Figure 5.15: PDF of normalized completion delay data for GPUSync with $(2, 3, \mathsf{FIFO})$.

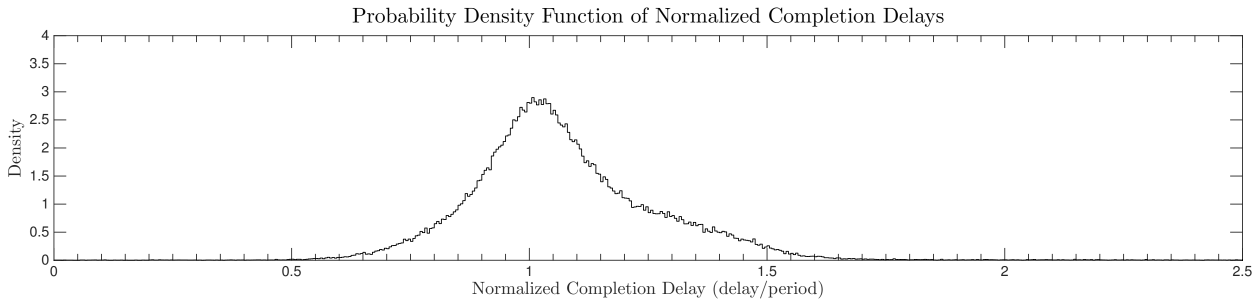


Figure 5.16: PDF of normalized completion delay data for GPUSync with $(2, 3, \mathsf{PRIO})$.

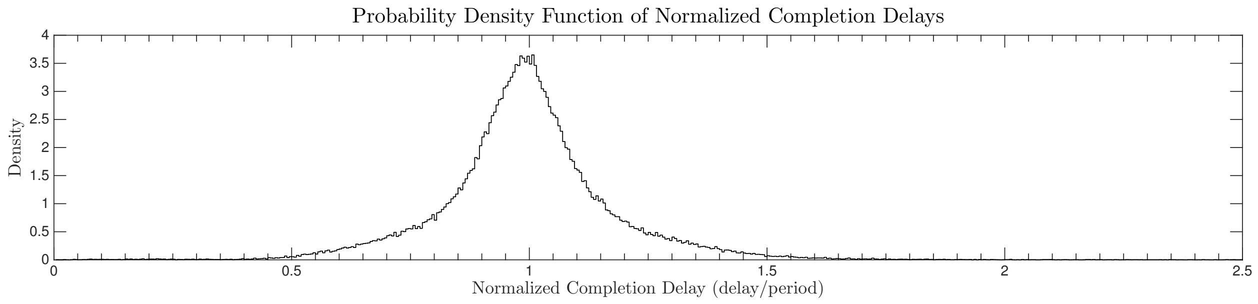


Figure 5.17: PDF of normalized completion delay data for GPUSync with $(3, 2, \mathsf{FIFO})$.

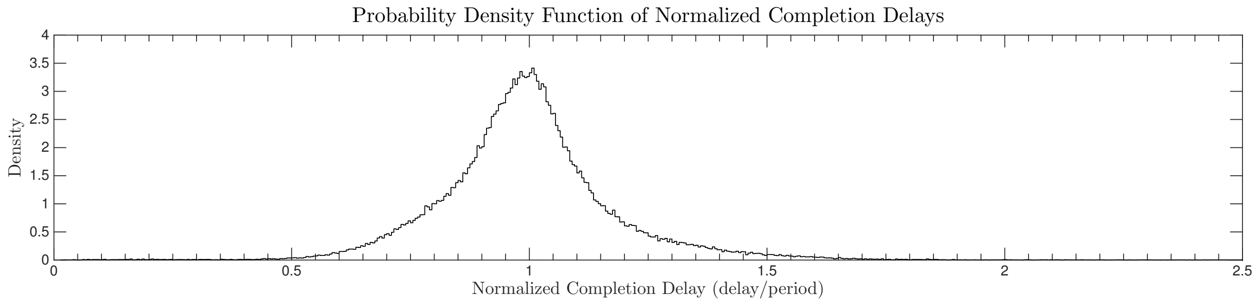


Figure 5.18: PDF of normalized completion delay data for GPUSync with $(3, 2, \mathsf{PRIO})$..

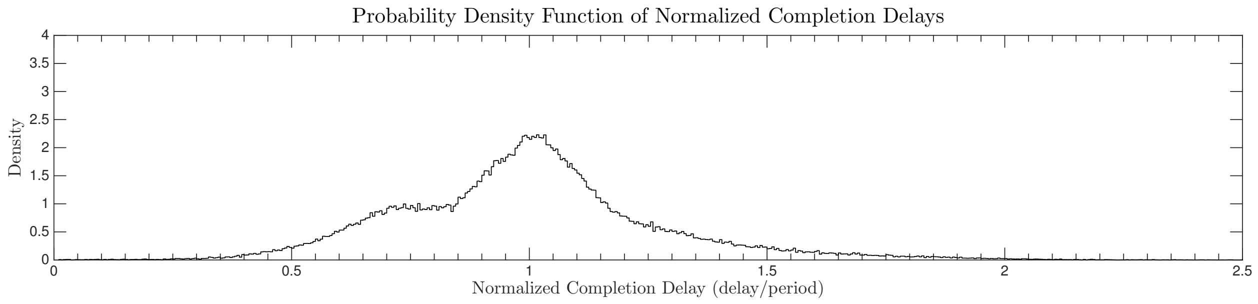


Figure 5.19: PDF of normalized completion delay data for GPUSync with $(6, 1, \mathsf{FIFO})$.

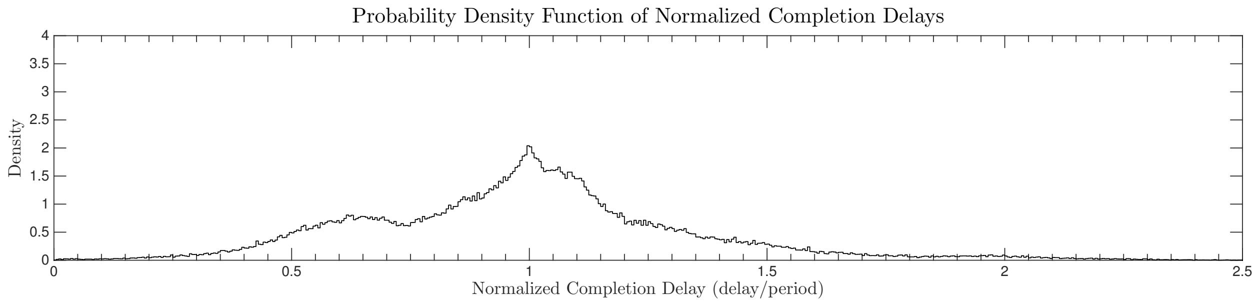


Figure 5.20: PDF of normalized completion delay data for GPUSync with $(6, 1, \mathsf{PRIO})$.

complete several jobs in quick succession as it catches up. In other words, one or more short completion delays may follow a very long completion delay. The first mode (the one near $x = 0.05$) may indicate such a behavior. The corresponding long completion delays make up the long tail of the PDF, which we have clipped at $x = 2.5$.

Although the LITMUS^{RT} without GPUSync configuration did not drop any frames, the playback of the stabilized video is far from smooth.

Observation 36. *The PDFs of the GPUSync configurations with $\rho \in \{1, 2\}$ indicate unschedulability.*

Figures 5.13 through 5.16 depict the PDFs for GPUSync configurations where $\rho \in \{1, 2\}$. We see in these distributions that the majority of normalized completion delays are greater than 1.0. We may expect to see this characteristic in a PDF of an unschedulable configuration, as it indicates that completion delays are continually greater than task period (*i.e.*, greater than 1.0 after normalization), so deadlines are missed by continually greater margins. In this experiment, this ultimately results in dropped frames when the experiment terminates after 400 seconds. This is most clearly demonstrated in the PDFs for GPUSync configurations with $\rho = 1$ in Figures 5.13 and 5.14—the bulk of normalized completion delays occur on the domain $[1.5, 2]$.

We also see this characteristic in Figures 5.15 and 5.16 for GPUSync configurations with $\rho = 2$, although it is less pronounced. In Figure 5.15, observe that the descent from the peak on the right of $x = 1.0$ is more gradual than the ascent to the left of 1.0—more normalized completion delays are greater than 1.0. We see the same trend in Figure 5.16.

This concludes our examination observed completion delays.

5.5.3.2 End-to-End Latency

We now examine the observed end-to-end response time latency of graphs under a subset of our various system configurations. Unlike in our study of completion delays, we used LITMUS^{RT}'s low-level tracing capabilities to accurately record the release and completion time of source and sink node jobs, respectively. We compute the end-to-end latency of a single end-to-end execution of a graph by computing the difference between the release time of the source job and the completion time of the corresponding sink job. Due to our reliance on LITMUS^{RT} for gathering these measurements, we limit our investigation here to GPUSync configurations and LITMUS^{RT} without GPUSync. That is, we do not study end-to-end latencies under the SCHED_OTHER or SCHED_FIFO configurations.

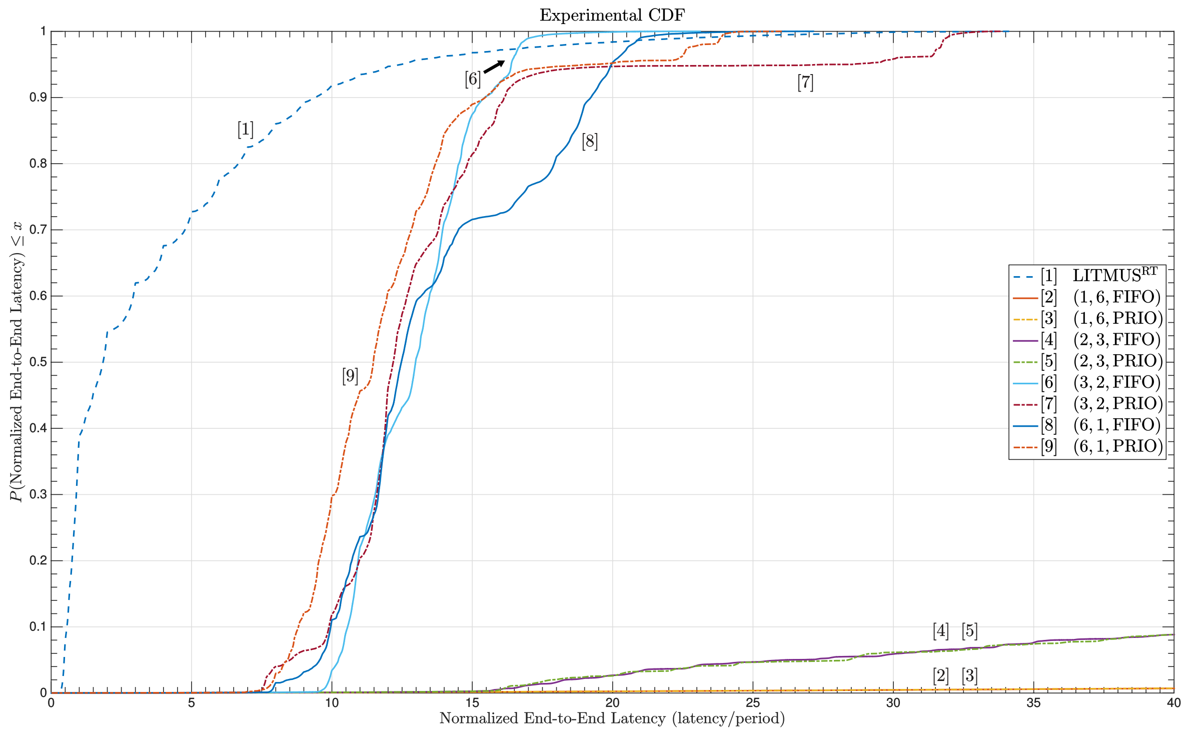


Figure 5.21: CDF of normalized observed end-to-end latency (domain clipped at $x = 40$).

We used the same experimental setup as before: The seventeen graphs described in Table 5.2 were executed for a duration of 400 seconds on six CPU cores and one GPU. The data we present in this section was gathered in a separate batch of experiments from the ones we performed to gather completion delay measurements, so there may be some discrepancy in worst-case measurements between the two data sets. However, as we shall see, we observe similar trends in both sets of measurements.

As with completion delays, end-to-end latencies are influenced by graph period. In order to study the end-to-end latencies of graphs with different periods, we *normalize* each measurement by dividing each measured end-to-end latency by the graph period. For every tested system configuration, we obtained a collection of normalized end-to-end latencies. Figure 5.21 plots the cumulative distribution function (CDF) for the observations of each tested configuration. These curves plot the likelihood that a given normalized end-to-end latency is less than a given value. For example, for the GPUSync configuration $(3, 2, \mathsf{PRIO})$ (curve 7), we see that approximately 95% of normalized end-to-end latency were less than 25. To make this observation, we find the y-value of curve 7 at $x = 25$.

In general, a curve that tends most towards the top-left corner of the figure is considered "best," as this indicates that most end-to-end latencies are short. However, since we are most interested in *worst-case* *behavior* and *predictability*, we look for other characteristics in the curves. Good worst-case behavior is indicated by a curve with a short tail, *e.g.*, one where $y = 1$ for a small x-value. Good predictability is indicated by a large increase in a curve over a short x-interval, as this means that there is little variance among many observations. Correspondingly, gradually increasing curves indicate a high degree of variance in the observations.

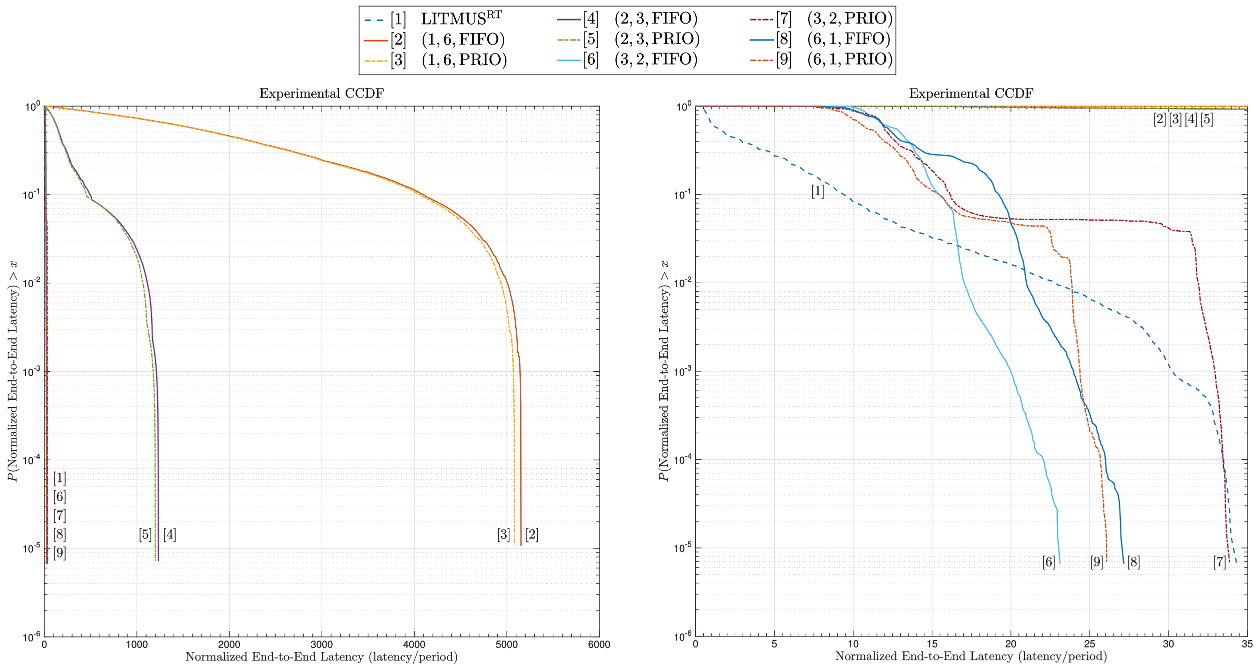
In order to study the characteristics of the more interesting curves, we have clipped the domain of Figure 5.21 to $x = 40$. This significantly truncates the curves for the GPUSync configurations where $\rho \in \{1,2\}$ (curves 2 through 5). However, we will study the end-to-end latencies of these configurations with a method better suited to studying curves with long tails, shortly. We make the following observations.

Observation 37. *GPUSync with* $(3,2,\mathsf{FIFO})$ *exhibits the least variance in end-to-end latency.*

To make this observation, for each curve plotted in Figure 5.21, find the approximate normalized end-to-end latency (on the x-axis) where $P(x)$ (on the y-axis) is first greater than zero. Let us denote this point with the variable a. Next, find the approximate normalized end-to-end latency where $P(x)$ is nearly one. Let us denote this point with the variable b. Most, if not all, normalized end-to-end latencies for the curve lie within the domain of $[a,b]$, which is $(b-a)$ units in length. For GPUSync with $(3,2,\mathsf{FIFO})$ in curve 6, this domain is roughly $[9,20]$, with a length of about 11 units. Compare this to LITMUS$^{\text{RT}}$ without GPUSync in curve 1, where the domain is $[0.25,30]$ (29.75 units long). GPUSync with $(3,2,\mathsf{PRIO})$ (curve 7), the domain is about $[7.5,32.5]$ (25 units long). GPUSync with $(6,1,\mathsf{FIFO})$ (curve 8), the domain is about $[7.5,22.5]$ (15 units long). GPUSync with $(6,1,\mathsf{PRIO})$ (curve 9), the domain is about $[7.5,24.5]$ (17 units long). The domains for the remaining GPUSync configurations (curves 2 through 5) cannot be observed in Figure 5.21 due to the clipping the x-axis at $x = 50$.

Observation 38. *GPUSync configurations where* $\rho \in \{3,6\}$ *exhibit similar behavior for approximately* 70% *of normalized end-to-end latencies.*

In Figure 5.21, there is a clear clustering of curves 5 through 9 on the domain $[7.5,14]$. For instance, for curves 5 through 8 normalized end-to-end latencies are below 13.5 about 35% of the time. Is not until $y = 0.7$ (70%) that these curves really begin to differentiate. This is not to say that the curves for GPUSync configurations where $\rho \in \{3,6\}$ are indistinguishable before $y = 0.7$. For example, GPUSync configuration $(6,1,\mathsf{PRIO})$ initially shows the best end-to-end latency behavior, as its curve (curve 9) is above the other GPUSync configurations until about $x = 15$.



(a) full domain (b) domain clipped at $x = 35$

Figure 5.22: CCDF of normalized observed end-to-end latency (y-axis on log scale).

To study the worst-case end-to-end latency behavior of the system configurations in our experiment, we plot the *complementary* cumulative distribution function (CCDF) of our normalized observations of each tested configuration in Figure 5.22. In Figure 5.22(a), we plot our data on an x-axis long enough to capture all observations. In Figure 5.22(b), we plot the same data, but we clip the domain at $x = 35$ in order to better depict the shape of the CCDFs for the better-performing configurations.

The CCDF is useful for determining the probability that a given observation is *greater* than a given x value. In other words, the CCDF is useful for studying worst-case behavior. This is especially true when the CCDF is plotted on a logarithmic y-axis, as we have done in Figure 5.22. For example, in Figure 5.22(b), find where curve 9 crosses the horizontal line for $y = 10^{-1}$. This occurs at about $x = 15$. This indicates that 10% of normalized end-to-end latencies for curve 9 are greater than 15. We may make similar types of observations along the horizontal lines for $y = 10^{-2}$ (1%), $y = 10^{-3}$ (0.1%), $y = 10^{-4}$ (0.01%), and $y = 10^{-5}$ (0.001%). We make the following observations.

Observation 39. *The normalized end-to-end latencies for GPUSync configurations are $\rho \in \{1, 2\}$ are* very *poor.*

In Figure 5.22(a), curves 2 through 5 (those for GPUSync configurations where $\rho \in \{1,2\}$) extend beyond a normalized end-to-end latency of 1,000. This result should not be surprising, given the poor performance we observed for these configurations in our study of completion delays (Observations 28 and 36)—our video stabilization task set is clearly unschedulable under these configurations of GPUSync.

Observation 40. *GPUSync configurations with FIFO-ordered engine locks exhibit a smooth degradation in performance.*

In Figure 5.22(b), we observe that the curves for GPUSync with $(3,2,\mathsf{FIFO})$ (curve 6) and $(6,1,\mathsf{FIFO})$ (curve 8) descend gradually, in comparison to the priority-ordered engine lock configurations (curves 7 and 9). This indicates that performance degrades more smoothly under FIFO-ordered engine locks.

Observation 41. *GPUSync configurations with priority-ordered engine locks perform poorly for approximately* 5% *of the time, and they are prone to creating extreme outliers in end-to-end latencies.*

In Figure 5.22(b), observe the flat table-like trend for GPUSync configurations where priority-ordered engine locks are used (curves 7 and 9). These trends begin around $x = 17$ at $y = 5\%$. The trend lasts until about $x = 22$ and $x = 31.5$ for GPUSync configurations with $(6,1,\mathsf{PRIO})$ (curve 9) and $(3,2,\mathsf{FIFO})$ (curve 7), respectively. After these points, the curves begin to drop precipitously.

These flat trends indicate a performance gap, where the majority of end-to-end graph invocations have good end-to-end latencies, while a few end-to-end graph invocations have very poor end-to-end latencies, relatively speaking. That is, these GPUSync configurations with priority-ordered locks are prone to extreme outlier behavior. Let us examine curve 7 for GPUSync with $(3,2,\mathsf{PRIO})$ more closely. Here, approximately 93% of end-to-end graph invocations have a normalized end-to-end latency *less* than 17. *Only* 3% of end-to-end graph invocations have a normalized end-to-end latency between 17 and 31.5—a very long interval of 14.5 units! The remaining 4% of end-to-end graph invocations have a normalized end-to-end latency greater than 31.5—this occurs over short interval from 31.5 to 34 (an interval of 2.5 units). We may make similar observations for GPUSync with $(6,1,\mathsf{PRIO})$ (curve 9), although outlier behavior is less extreme.

Observation 42. *GPUSync with* $(3,2,\mathit{FIFO})$ *exhibits the best worst-case behavior.*

In Figure 5.22(b), find where each depicted curve intersects with the vertical line where normalized observed end-to-end latencies are 20 units ($x = 20$). We find that only 0.09% of normalized observed end-to-end latencies for GPUSync with $(3,2,\mathsf{FIFO})$ (curve 6) are greater than 20. Contrast this to about 5%

for the other GPUSync configurations where $\rho \in \{3,6\}$ (curves 7 through 9), or about 1.5% for LITMUS$^{\text{RT}}$ without GPUSync (curve 1). The differences are even more dramatic where normalized observed end-to-end latencies are 23. We find that only 0.002% of normalized observed end-to-end latencies for GPUSync with $(3,2,\mathsf{FIFO})$ (curve 6) are greater than 23. Contrast this to 0.25% for GPUSync with $(6,1,\mathsf{FIFO})$ (curve 8), 1% for LITMUS$^{\text{RT}}$ without GPUSync (curve 1), 2.5% for GPUSync with $(6,1,\mathsf{PRIO})$ (curve 9), and 5% for GPUSync with $(3,2,\mathsf{PRIO})$ (curve 7).

Observation 43. *GPUSync with* $(3,2,\mathit{FIFO})$ *exhibits the best real-time behavior.*

The observation follows from Observations 37 and 42. The GPUSync configuration with $(3,2,\mathsf{FIFO})$ exhibits both the least variance in normalized end-to-end latencies, as well as the best worst-case behavior.

We make one last general observation before concluding our examination of the real-time runtime performance of our video stabilization task set.

Observation 44. *The behaviors we observed through the study of normalized end-to-end latencies correlate to those we made through the study of completion delays.*

In Section 5.5.3.1, we studied the performance of the video stabilization task set through completion delay metrics. As we discussed earlier, completion delay metrics are "fuzzy" in that we may observe a range of values for task set that meets even hard real-time constraints (recall the discussion of Figure 5.9). Nevertheless, through completion delay metrics, we observed the following: **(i)** the video stabilization task set was clearly unschedulable under GPUSync configurations with $\rho \in \{1,2\}$; **(ii)** GPUSync configurations with FIFO-ordered engine locks exhibited less variance in comparison to priority-ordered counterparts; **(iii)** GPUSync configurations with $\rho = 3$ yielded better real-time behaviors than $\rho = 6$; and **(iv)** the GPUSync configuration with $(3,2,\mathsf{FIFO})$ performed best among all tested configurations. Each of these observations are supported by observations we make in this section based upon end-to-end latency data. This speaks towards the validity and usefulness of completion delay-based metrics.

5.6 Conclusion

Real-time applications with graph-based software architectures represent an important segment of computing that is not directly supported by the periodic or sporadic task models. As code complexity increases and heterogeneous computing platforms become more common and varied, we may expect to see such graph-based applications to become more common. This claim is supported by the recently ratified OpenVX standard and NVIDIA's development of VisionWorks.

In this chapter, we extended GPUSync to support these graph-based real-time applications. We developed PGM$^{\text{RT}}$, a middleware library, to track data dependencies among graph nodes and coordinate their execution. We also enhanced operating system support for graph-based real-time applications with modifications to LITMUS$^{\text{RT}}$ to dynamically adjust job release times and deadlines in accordance to real-time theory. Although PGM$^{\text{RT}}$ may be configured to support any POSIX-compliant platform, it may also be configured to tightly integrate with LITMUS$^{\text{RT}}$ to reduce system overheads and mitigate priority inversions.

We applied GPUSync and PGM$^{\text{RT}}$ to real-world software: an alpha-version of NVIDIA's VisionWorks. Through the use of PGM$^{\text{RT}}$, we enhanced the back-end of VisionWorks to support multi-threaded pipelined execution (which is expressly *not* supported according to the OpenVX standard upon which VisionWorks is based). Due to the shear amount of code and complexity, we could not feasibly modify VisionWorks to use GPUSync directly. Instead, we developed an interposition library to the CUDA runtime, libgpui. Libgpui enabled us to intercept all relevant CUDA-runtime calls made by VisionWorks and schedule them under GPUSync. The benefits offered by libgpui came at the expense of larger GPUSync token critical sections than strictly necessary. We then evaluated the real-time performance of our real-time version of VisionWorks under several Linux and LITMUS$^{\text{RT}}$-based configurations, including eight unique configurations of GPUSync. Our results show that real-time GPU scheduling is necessary in order to achieve predictable real-time performance. We also identified which configurations of GPUSync gave the best real-time performance within the context of our experiments. Importantly, we identified that there is a balance to be struck between GPU engine parallelism and the interference that *too much* parallelism may introduce.