Supporting Real-Time Computer Vision Workloads using OpenVX on Multicore+GPU Platforms

Glenn A. Elliott, Kecheng Yang, and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

***Abstract*—In the automotive industry, there is currently great interest in supporting driver-assist and autonomous-control features that utilize vision-based sensing through cameras. The usage of graphics processing units (GPUs) can potentially enable such features to be supported in a cost-effective way, within an acceptable size, weight, and power envelope. OpenVX is an emerging standard for supporting computer vision workloads. OpenVX uses a graph-based software architecture designed to enable efficient computation on heterogeneous platforms, including those that use accelerators like GPUs. Unfortunately, in settings where real-time constraints exist, the usage of OpenVX poses certain challenges. For example, pipelining is difficult to support and processing graphs may have cycles. In this paper, graph transformation techniques are presented that enable these issues to be circumvented. Additionally, a case-study evaluation is presented involving an OpenVX implementation in which these techniques are applied. This OpenVX implementation runs atop a previously developed GPU-management framework called GPUSync. In this case study, the usage of GPUSync's GPU management techniques along with the proposed graph transformations enabled computer vision workloads specified using OpenVX to be supported in a predictable way.**

I. Introduction

In the automotive industry today, vision-based sensing through cameras is being used to support features such as automatic lane-keeping, adaptive cruise control, *etc.* In the coming years, such features are expected to evolve to include 360-degree sensing, which will ultimately be integrated with actuation logic that supports partial or full autonomy. To enable cost-effect deployments of such features, within an acceptable size, weight, and power envelope, multiple vision-based processing streams must be consolidated onto a single hardware platform in a way that enables real-time requirements to be validated.

With regard to such a consolidation, multicore platforms augmented with graphics processing units (GPUs) offer a promising way forward, as GPUs are well suited for accelerating the matrix-oriented computations inherent in many computer vision applications. To ease the development of such applications on heterogeneous architectures such as multicore+GPU platforms, and to enable system-level optimization [1], a standard computer vision API called OpenVX has been created and ratified [2]. Unfortunately, several aspects underlying the design of OpenVX make validating real-time requirements problematic, despite the fact that real-time applications are an intended use case [3]. This is disconcerting, given that OpenVX undoubtedly will be adopted as a standard in many settings where such requirements exist.

Work supported by NSF grants CNS 1115284, CNS 1218693, CPS 1239135, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and a grant from General Motors.

Problems with OpenVX. The OpenVX API provides the programmer with a set of basic operations, or *primitives*, commonly used in computer vision algorithms.[1] A computer vision algorithm is constructed by instantiating primitives as *nodes* and linking node outputs to node inputs to create a computer vision processing graph.

OpenVX has a simple execution model,[2] which simplifies its API and implementation and allows it to perform well on processors with a wide range of capabilities, ranging from simple ASICs to complex multicore+GPU platforms. However, this model imposes three significant implications on real-time scheduling. First and foremost, the specification has no notion of a repeating (*i.e.*, periodic or sporadic[3]) task, and lacks any framework for real-time analysis. With respect to analysis, a key issue is the allowance of "back edges" that can create cycles in a graph. Second, the specification does not define a threading model for graph execution. Finally, the specification requires a graph to execute end-to-end before it may be executed again. This significantly hinders the ability to exploit parallelism by "pipelining" portions of a graph's structure to improve performance.

Contributions. In this paper, we examine these real-time-related shortcomings of OpenVX in detail and present ways of mitigating them. Our specific contributions are threefold. First, we report on our efforts to implement a variant of OpenVX that is amenable to real-time analysis. With support from NVIDIA, we adapted an alpha-version of an NVIDIA OpenVX implementation called VisionWorks® [4] to run atop PGM^{RT} (a graph-based middleware developed by our group previously [5]), GPUSync (a real-time GPU management framework developed by our group previously [6, 7]), and LITMUS^{RT} (a real-time Linux extension jointly maintained by our group and MPI [8]).[4] Second, we explain how to transform OpenVX graphs to eliminate cycles due to back edges and support pipelining. These transformations enable real-time constraints to be

[1] In OpenVX, these basic operations are called "kernels." We avoid this term to eliminate confusion when referring to GPU and OS kernels.

[2] From Sec. 2.8.5 of the OpenVX standard [2]: "[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."

[3] We assume familiarity with the sporadic and periodic task models.

[4] VisionWorks is a registered trademark of the NVIDIA Corporation.

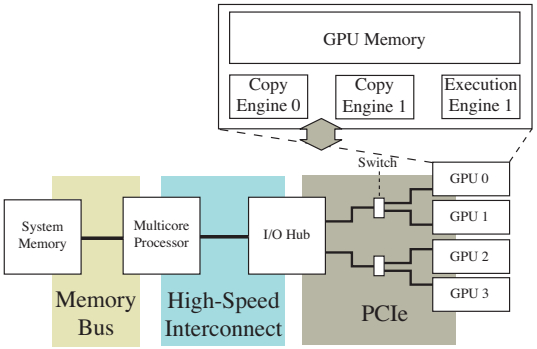


Figure 1: Example high-level architecture.

validated. The specific constraint we consider is that "end-to-end" graph response times are provably bounded. Third, we present results from a runtime evaluation of several configurations of our VisionWorks variant. In particular, we compare our GPUSync configuration against two purely Linux-based ones, as well as a LITMUS^{RT} configuration *without* GPUSync. Our results demonstrate the efficacy of our graph transformations and proper GPU management using GPUSync.

Organization. In the remainder of the paper, we provide needed background (Sec. II), describe our modified version of VisionWorks (Sec. III), present our experimental evaluation (Sec. IV), and conclude (Sec. V).

II. BACKGROUND

In this section, we present a brief overview of GPUSync [6, 7, 9] and discuss OpenVX in greater detail. GPUSync is a GPU management framework designed to be used on multicore platforms where multiple GPUs may be present. In describing it, we will consider the platform depicted in Fig. 1, which is based on that used in our experimental evaluation. In the depicted platform, each GPU has one *execution engine* (EE) (which is comprised of many parallel processors) and two DMA *copy engines* (CEs). The CEs connect to the host system via a full-duplex PCIe bus. PCIe is a hierarchically organized packet-switched bus with an I/O hub at its root. The structure depicted in Fig. 1 may be replicated in large-scale NUMA platforms, with CPUs and I/O hubs connected by high-speed interconnects.

GPU usage pattern. GPU-using programs execute on CPUs and invoke a sequence of *GPU operations*. There are two types of GPU operations. *Kernel* operations are programs executed by a GPU EE. *Memory copy* operations are data transfers to or from a GPU's local memory; these are processed by the CEs. A general execution sequence for a GPU-using program scheduled alone is depicted in Fig. 2. Observe that a program running on a CPU initiates GPU operations—the GPU does not initiate them independently. At time t_1, the program selects a GPU to use. At time t_2, the program transmits input data for the GPU kernel from system memory to GPU memory. The memory copy is processed by one of the GPU's CEs. The program waits (it may elect to either busy-wait or suspend) until the copy operation

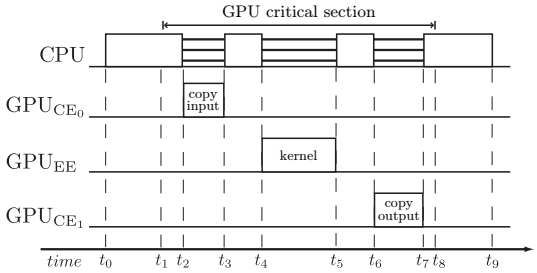


Figure 2: GPU-using program execution sequence.

completes at time t_3. A kernel that operates on the input data is executed at time t_4—computational results are stored in GPU memory. The program copies the kernel output from the GPU at time t_6. Finally, the program no longer requires the GPU at time t_8. We call the duration from time t_1 to time t_8 a *GPU critical section* because the program expects its sequence of operations to be carried out on the same GPU. GPU operations on the various engines are non-preemptive. For example, GPU_{CE_0} cannot be preempted within $[t_2, t_3]$. Note that this is only a simple execution sequence. Any number of GPU operations may actually be issued within a GPU critical section. Also, we have depicted the input and output memory copies as processed by different CEs—it is actually up to the GPU to select which CE to use.

GPUSync. In GPUSync, the management of GPU-related resources is viewed as a *synchronization problem* and thus real-time multiprocessor locking protocols are used to acquire and release such resources. A two-step process is followed. To access a GPU, a task must first acquire one of several *tokens* associated with that GPU using a locking protocol. Once a token has been acquired, a task may acquire an *engine lock* associated with one of the engines (EE or CE) of that GPU in order to access that engine.

GPUSync is highly configurable. For example, the number of tokens per GPU is configurable. Also, lock wait queues may be configured so that tasks wait in FIFO order or priority order. Other configuration parameters determine whether and how task state data may be copied from one GPU to another or to/from host memory. GPUSync organizes GPUs into clusters (a task may be assigned a token for any GPU in such a cluster) and the cluster size is configurable. GPUSync can be used in conjunction with various partitioned, clustered, or global real-time CPU schedulers supported in LITMUS^{RT}. Real-time constraints can be validated when GPUSync is used by applying bounds on priority-inversion blocking times within conventional scheduling analysis associated with the assumed CPU scheduling algorithm.

OpenVX. Computer vision algorithms are commonly expressed using dataflow graphs. An example is given in Fig. 3, which depicts a simple pedestrian detection application that could be used in an automotive application. In this example, 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

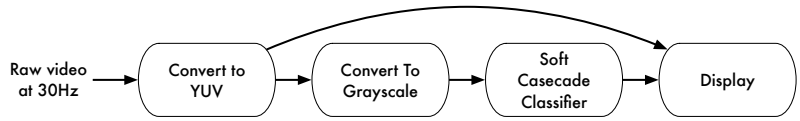


Figure 3: Dataflow graph of a simple pedestrian detector application.

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" [10] to detect pedestrians. Finally, the last node displays an overlay of detected pedestrians over the original color image. To support this pedestrian detection application in a real-time setting, we require a task model and implementation that will allow us to exploit the parallelism inherently expressed by the graph, while still supporting real-time analysis and predictable execution.

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.[1] 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.[5] 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 declared. Rather, the structure of a graph is derived from how parameters are bound to nodes. We demonstrate this with an example. Fig. 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 by a user-provided node created by the function `mySoftCascadeNode()`, which writes a list of detected pedestrians to `detected`.[6] Fig. 4(b) depicts the bindings of parameters to nodes. Fig. 4(c) depicts the derived structure of this graph.

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

III. Adding Real-Time Support to VisionWorks

In this section, we describe our efforts in adding real-time support to an OpenVX implementation by NVIDIA called VisionWorks. Our modifications were applied to an alpha-version of VisionWorks. This alpha-version was under active development at the time, so *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 work with VisionWorks was funded by NVIDIA through their internship program—at this time we are unable to share the VisionWorks-specific software we developed in this effort,[7] since it is the property of NVIDIA. Those wishing to investigate OpenVX implementations may find a sample (non-GPU-supporting) implementation of OpenVX at Khronos Group's website [3].

Basic implementation overview. We begin by providing a brief overview of our modifications to VisionWorks (further details can be found in [9]). Fig. 5 depicts the full software stack of VisionWorks in different configurations. Fig. 5(a) depicts the stock VisionWorks software stack. Here, user applications are written to the VisionWorks API. Internally, VisionWorks may use other libraries, such as OpenCV (a popular computer vision library). VisionWorks and these additional libraries make use of CUDA services by interfacing with NVIDIA's libcudart library, which implements the CUDA API. This library interfaces with another NVIDIA library, libcuda, which implements a lower-level "CUDA driver" API. This library is responsible for implementing most of the CUDA GPU runtime; it communicates with NVIDIA's GPU device driver through a proprietary API.[8] The device driver executes within the operating system and is directly responsible for managing the GPU.

We extend the simple OpenVX execution model in VisionWorks to support pipelined multithreaded execution,

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

[7]Specifically, we refer to our modified version of VisionWorks and a software library we call libgpui. Our non-VisionWorks-specific software *is* publicly available: GPUSync at www.litmus-rt.org, and PGM^{RT} at https://github.com/GElliott/pgm/.

[8]The libcuda library communicates with the device driver using the generic `ioctl()` Linux system call.

```
1  vx_image imageRaw;  // graph input  : an image
2  vx_array detected;  // graph output : a list of detected pedestrians
3  ...
4  // instantiate a graph
5  vx_graph pedDetector = vxCreateGraph(...);
6  ...
7  // instantiate additional parameters
8  vx_image imageIYUV = vxCreateVirtualImage(pedDetector, ...);
9  vx_image imageGray = vxCreateVirtualImage(pedDetector, ...);
10 ...
11 // instantiate primitives as nodes
12 vx_node convertToIYUV = vxColorConvertNode(pedDetector, imageRaw, imageIYUV);
13 vx_node convertToGray = vxChannelExtractNode(pedDetector, imageIYUV, VX_CHANNEL_Y, imageGray);
14 vx_node detectPeds    = mySoftCascadeNode(pedDetector, imageGray, detected, ...);
15 ...
16 vxProcessGraph(pedDetector);  // execute the graph end-to-end
```

(a) OpenVX code for constructing a graph.

(b) Bindings of data object parameters to nodes.

(c) Derived graph structure.

Figure 4: Construction of a graph in OpenVX for pedestrian detection.

where each node is assigned a *dedicated thread* for execution. Within the real-time context, we may view each thread as a sporadic real-time task that executes its node's primitive operation once per job. We realize this thread-per-node execution model through the use of a previously developed library, PGM^{RT}, which is a portable middleware framework for managing real-time dataflow applications on multicore platforms [5]. Fig. 5(b) depicts the resulting software stack. Although this software stack enables pipelined multithreaded execution, applications remain at the mercy of the stock GPU software stack with respect to GPU resource arbitration and scheduling. As we have demonstrated in prior work [6, 7], this is insufficient for supporting predictable real-time execution of GPU-using applications.

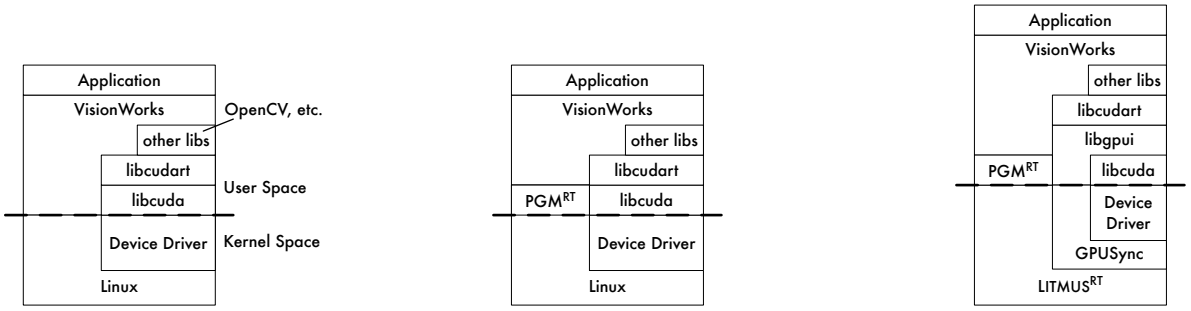
We improve upon the prior software stack by integrating with GPUSync. This is depicted in Fig. 5(c). GPUSync is an API-driven GPU scheduler, meaning that applications must explicitly request GPU resources through an API provided by GPUSync—by default, GPU scheduling is *not* transparent to user applications. Unfortunately, VisionWorks has a large and complex code base. Moreover, the libraries that VisionWorks uses (*e.g.*, OpenCV) also execute work on GPUs. We deemed it infeasible to directly modify VisionWorks and the associated libraries to use the GPUSync API directly. (Even if this were feasible, it would remain error-prone, as we would have to comb these large code bases to identify and evaluate each and every code path on which the CUDA API is used.) To overcome these challenges, we developed an additional library, which we call libgpui ("GPU interposition library"). This library provides an API that matches the low-level CUDA driver API. At dynamic-link time (*i.e.*, when a program is initially launched), libgpui forcibly overrides all CUDA driver API calls.[9] Libgpui is essentially inserted between all user code, including VisionWorks and the libraries it uses, and the GPU. With this infrastructure in place, libgpui evaluates every CUDA driver API call, and invokes the GPUSync scheduler as needed, before passing these calls onto libcuda. Thus, libgpui transparently extends GPUSync GPU scheduling to VisionWorks and all other GPU-using libraries.

In total, our additions to the VisionWorks software stack took approximately 34K lines of C/C++ code: 15K for GPUSync, 8K for libgpui, 6K for PGM^{RT}, and 5K to extend VisionWorks itself.

Ensuring conformance to an analyzable task model. The timing constraints of interest to us pertain to *end-to-end graph processing times*, *i.e.*, the duration of time from when an input frame is consumed by a source node to when any corresponding output is generated by a sink node. In particular, we require that such processing times are provably bounded. When tasks are scheduled on CPUs using the clustered earliest-deadline-first (C-EDF) scheduler[10] or other EDF-based scheduling policies, such bounds can be computed using results of Liu and Anderson [11] with synchronization-related blocking due to the usage of GPUSync accounted for using blocking bounds from [9]. However, to apply these results, source

[9]This is accomplished through the `LD_PRELOAD` environment variable, which can be used to preload shared libraries when a program is launched.

[10]Under a clustered scheduler, CPUs are partitioned into clusters, each task is assigned to one cluster, and a task may execute on any CPU in its assigned cluster.



(a) Original software stack. (b) Pipelined execution support. (c) Full LITMUS^{RT} and GPUSync support.

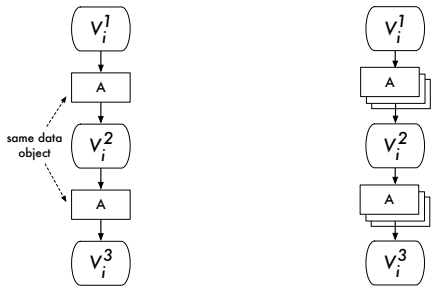
Figure 5: Software layers of an application using our modified version of VisionWorks.

nodes must be invoked sporadically, no task can exhibit intra-task parallelism, and no cycles may exist in any processing graph. Also, each node of a graph should be viewed as an individual schedulable entity, rather than the entire graph, to enable parallelism due to pipelining effects. Unfortunately, VisionWorks fails to satisfy any of these requirements, hence the need for our modifications to it.

Conceptually, modifying VisionWorks so that source nodes are invoked sporadically and intra-task parallelism is prevented is straightforward, although a few nontrivial technical details arise. Due to space constraints, we refer the reader to [9] for a discussion of these details and concentrate here on explaining our modifications to enable pipelining and to eliminate graph cycles.

Graph dependencies and pipelining. Recall from Sec. II 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 if tasks rather than entire graphs are schedulable entities, a new hazard arises: *a producer node may overwrite the contents of a data object before the old contents have been read or written by a 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.

This pipelining issue can be resolved by sufficiently *replicating* data objects, as illustrated in Fig. 6. If we replicate a given data object N times, then the j^{th} invocation of a node (*i.e.*, a job) that accesses it accesses the $(j \bmod N)^{th}$ replica. Clearly, N must be set sufficiently large, for otherwise, data objects could still be overwritten even with replication. Fortunately, prior work of Goddard and Jeffay [12, 13, 14] can be leveraged to derive safe replication bounds that eliminate the possibility of overwriting. These bounds are determined by examining the invocation rates of the nodes that access a given data object; bounds on any node's invocation rate can be determined as a function of the invocation rate of the source node of its graph. Further details concerning replication are given in an appendix.



(a) VisionWorks graph, V_i. (b) V_i with replicated data objects.

Figure 6: Replicating data objects to enable pipelining.

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 is 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 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 an 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. For example, in Fig. 7, which is considered in more detail later, the delay edges sourced from the "Harris Feature Tracker" node are back-edges; the other delay edges are not. As seen in Fig. 7, back-edges ostensibly result in cycles. This is problematic because the end-to-end response-time analysis of Liu and Anderson [11] applies only to acyclic graphs. In the appendix, we explain how to break such

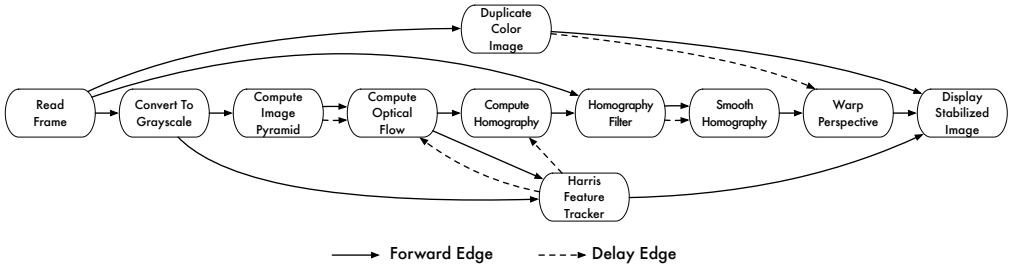


Figure 7: Dependency graph of video stabilization application.

cycles. Two basic approaches may be applied. First, if the delay represented by a back-edge involves accessing data sufficiently "far into the past," then no delay actually occurs as a result and so there is really no cycle. Second, if a delay actually can occur, then it may be possible to break the corresponding cycle by combining certain graph nodes into "super nodes" to enforce a desired sequential execution pattern. Of course, such an approach sacrifices parallelism.

Scheduling policies. In our modified version of VisionWorks, the programmer may specify which scheduling policy to use in allocating CPU time to tasks (*i.e.*, graph nodes). Our modified VisionWorks supports the standard Linux SCHED_OTHER and SCHED_FIFO policies, as well as SCHED_LITMUS. Though our *analysis* supports sporadic source-node releases, our *implementation* supports only the periodic release of such nodes. We use POSIX real-time timers to implement periodic source-node releases under the standard Linux policies. We rely upon the periodic job release infrastructure of LITMUS^{RT} for the SCHED_LITMUS policy.

We assume EDF-based scheduling under the SCHED_LITMUS policy. When the SCHED_FIFO policy is employed, the programmer supplies a "base" graph priority. The priority of the thread that implements a given node is determined by taking the length of the longest path between that node and the source node of its graph, 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.

IV. EVALUATION

In this section, we evaluate the observed real-time performance of our enhanced version of VisionWorks. We begin with a description of the computer vision application we used in this evaluation. We then discuss our experimental setup and present our results.

Video stabilization. VisionWorks includes a variety of demo applications, including a pedestrian detection application, not unlike the one illustrated in Fig. 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 can be 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.

Fig. 7 depicts the dependency graph of the video stabilization application. Table I gives a brief description of each node in this graph. Video stabilization exercises many types of primitives that are common to other computer vision algorithms; this makes it a good candidate for system evaluation purposes. For example, image pyramid computation is a core step in many object detection algorithms. 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 Fig. 7. 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 memory copy 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. Under GPUSync, 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.

Experimental setup. In this evaluation, we focus on multicore+GPU scheduling for a computing platform with a single GPU. This focus is motivated by two reasons. First, such a multicore single-GPU system reflects many common embedded system-on-chip computing platforms available today. Second, our alpha-version of VisionWorks does not allow nodes of a processing graph to span several GPUs.[11]

Our experimental workload was comprised of 17 instances of the video stabilization application in order to load the CPUs and GPU engines. This is reflective of a use-case

[11]This is not a fundamental shortcoming of OpenVX or VisionWorks. Rather, this capability had not yet been implemented.

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 I: Description of nodes used in the video stabilization graph of Figure 7.

Period	Number of 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 II: Task set using VisionWorks' video stabilization demo.

scenario where multiple computationally intensive vision processing computations are multiplexed onto a common hardware platform (*e.g.* to due size, weight, and power constraints). Each processing graph was executed within its own Linux process. Each graph node was executed by a dedicated thread, resulting in eleven threads per process, and 187 threads across all 17 processes. To each graph we assigned a period that was shared by every real-time task of the nodes therein. As shown in Table II, each of the 17 considered processing graphs was assigned one of six different periods. These periods represent a range of those we find in automotive applications.

We used a dual-socket Xeon X5060 hardware platform, which is similar to that depicted in Fig. 1, as our evaluation platform. Each socket contains six CPU cores running at 2.67GHz. Real-time tasks were isolated to six cores on one socket; the remaining socket was used for performance monitoring. All GPU calculations of the 17 graphs were executed on a single NVIDIA Quadro K5000 "Kepler" GPU. We used CUDA 6.5 and NVIDIA GPU driver 331.62. By design, our 17 graphs heavily loaded the six CPUs and GPU to near capacity.[12]

We examine the real-time performance of our workload on four different system configurations: Linux's fixed-priority scheduler (SCHED_FIFO); Linux's general-purpose "completely fair" scheduler (SCHED_OTHER); LITMUS^{RT}'s C-EDF scheduler; and LITMUS^{RT}'s C-EDF scheduler with GPUSync. All but the last of these configurations lack real-time management of the GPU, and instead default to the resource arbitration mechanisms used collectively by CUDA, the GPU driver, and the GPU hardware. For brevity, we henceforth refer to each of these configurations according to their scheduling policy (and we refer to the LITMUS^{RT} configuration with GPUSync as "GPUSync").

Under the SCHED_FIFO policy, processing graphs were assigned the base priorities listed in Table II. (Under SCHED_FIFO, priorities with a greater numerical value have higher priority.) A fixed priority for the thread of each node was derived using the method we described at the end of Sec. III. The gaps between base priorities ensure that the range of thread priorities of graphs with different periods never overlap. As a result, the priority of any thread within a given graph will always exceed those of threads of graphs with lower base priorities.

For the LITMUS^{RT}-based configurations, we divide our two-socket platform into two processor clusters, one cluster per socket. This is accomplished through LITMUS^{RT}'s C-EDF scheduler. Since our real-time tasks only execute within one cluster, we may also think of these configurations as using a global EDF (G-EDF) scheduler for six CPUs.

As we described in Sec. II, GPUSync is highly configurable. For GPUSync, we set the number of GPU tokens, denoted by the symbol ρ, to $\rho = 3$. An additional parameter, f, which controls the maximum length of FIFO queues within the locking protocol that arbitrates access to tokens, was set to $f = 2$. Together, these values for ρ and f lead to a locking protocol structure that is optimal with respect to suspension-oblivious analysis for a system of six CPUs [9]. We configured GPUSync's engine locks to prioritize requests in FIFO order.[13]

To ensure that each configuration performed to the best of its ability, we used libgpui to enforce two behaviors across *all* system configurations. First, libgpui set the appropriate CUDA runtime environment parameters to force all tasks to suspend while waiting for GPU operations to complete.

[12] During experimentation, the system tool `top` reported the CPUs to be approximately 96% utilized (a total CPU utilization of about 5.76). Similarly, the NVIDIA tool `nvidia-smi` reported the GPU execution engine to be approximately 66% utilized.

[13] We also evaluated seven additional GPUSync configurations, which tested different combinations of values for ρ, f, and engine lock prioritization methods. Due to page constraints, we present these results in [9].

Configuration	Total % of Frames 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	0	**3.67**	**2.37**	**1.39**	**0.99**	**1.00**	**0.15**

Table III: Average normalized completion delay data.

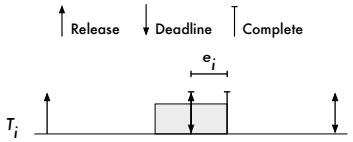
Second, libgpui automatically routed GPU operations to distinct per-node (per-thread) CUDA "streams." CUDA uses "streams" to order serially dependent GPU operations. In CUDA 6.5 and earlier, all threads within a process share the same stream by default, leading to false dependencies among GPU operations issued by different threads. By using streams, we avoid needless serialization of GPU operations.[14] For GPUSync, libgpui invoked GPUSync as needed prior to passing intercepted CUDA driver API calls on to the underlying CUDA library; libgpui passed intercepted API calls on to the underlying CUDA library immediately under the other configurations.

We executed our task set under each of the above system configurations for 400 seconds. Each graph processed a pre-recorded video file, which we pinned in system memory in order to avoid delays to due to disk access and page faults.

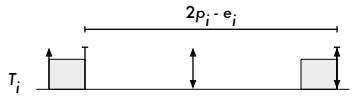
Completion delays. We require a common observational framework in order to fairly compare the four system configurations. Although LITMUS^{RT} offers kernel-level low-overhead tracing capabilities, we are unable to make use of them for the non-LITMUS^{RT}-based configurations. Instead, we opt for a simpler and more straightforward method. At the end of its execution, the sink node (*i.e.*, the "Display Stabilized Image" node) in each processing graph records a timestamp. We compute the duration between consecutive completion timestamps of each graph to obtain a "completion delay" metric. The benefit to using the completion delay metric is that 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). 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. Second, the metric is inherently noisy. For example, as depicted in Fig. 8, if a periodic task with period p_i meets all of its deadlines and always executes for its worst-case execution time e_i, then its completion delay may range within $[e_i, 2p_i - e_i]$. Despite these limitations, we feel that observed completion delays are useful in reasoning about the real-time runtime performance of our various system configurations.

Results. Table III summarizes our collected data. We compute the total number of video frames we expect our

[14]The recently released CUDA 7.0 implements this same behavior.



(a) Back-to-back job completion.



(b) Distant job completions.

Figure 8: Scenarios that lead to extreme values for measured completion delays.

video stabilization graphs to process within an allotted time, given graph periods. We say that a given configuration has "dropped" frames if the actual number of processed frames falls short of the number expected. The second column of Table III gives the percentage of dropped frames with respect to the expected number of processed frames. The remaining columns give information on completion delay metrics. We *normalize* each measured completion delay by dividing the measurement by the period of the task's graph. We then analyze the normalized completion delays collectively. In Table III, we give the maximum, $99.9^{th}\%$, $99^{th}\%$, median, and mean normalized completion delay for each configuration. The table also includes the standard deviation, denoted by σ, from the mean. We 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 observations.

Observation 1. *Mean completion delays are generally good under SCHED_OTHER,* LITMUS^{RT}*, and GPUSync.*

In Table III, we see that SCHED_OTHER, LITMUS^{RT}, and GPUSync all had a mean of approximately 1.0. This reflects good average-case behavior. However, we also observe that these configurations differ significantly in terms of maximum, $99.9^{th}\%$, and $99^{th}\%$, median completion delay.

Observation 2. *GPUSync was superior among the four configurations.*

In Table III, we see that GPUSync had the best outlier behavior among the four configurations. For example, the maximum completion delay under GPUSync was 3.67. Compare this to the maximums of 9.20, 11.12, and 12.12 for the SCHED_OTHER, SCHED_FIFO, and LITMUS^{RT}, respec-

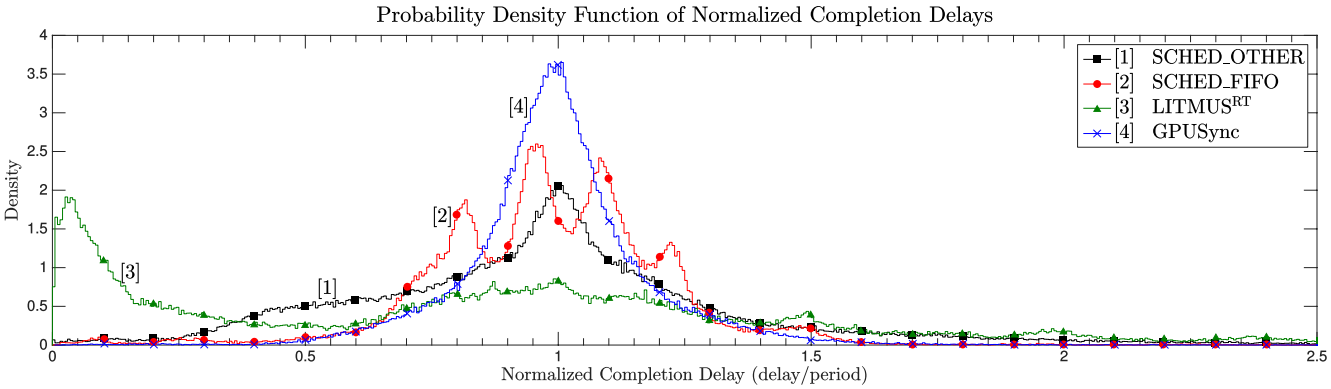


Figure 9: PDFs of normalized completion delay data. (This graph is probably best viewed in color.)

tively. In fact, the maximum completion under GPUSync was less than the 99^{th}% completion delay under SCHED_FIFO and LITMUS^{RT}. The superiority of GPUSync is also reflected by its small standard deviation of only 0.15. This is more than half that of the next-best, SCHED_OTHER.

Observation 3. *The real-time configurations SCHED_FIFO and* LITMUS^{RT} *performed poorly.*

We look at the percentage of dropped frames to detect unschedulability, since this is a sign of task starvation. We see that among the tested configurations, SCHED_FIFO dropped approximately 4.77% of its video frames. Moreover, it is the only configuration to drop frames. This is somewhat disappointing, since SCHED_FIFO is meant for use with real-time tasks on POSIX-compliant platforms. However, we see that this standard real-time scheduler actually exhibited some of the worst behaviors of the four configurations.

LITMUS^{RT} exhibits bad outlier behavior. For example, its max, 99.9^{th}%, and 99^{th}% completion delays are greater than those of the SCHED_OTHER and GPUSync. This high degree of variability is also reflected by its relatively large standard deviation of 0.93. This is over 2.5 and 6.2 times greater than the standard deviation of SCHED_OTHER and GPUSync, respectively.

The poor behavior of the real-time configurations that lack real-time GPU management may be due to busy-waiting, despite the fact that libgpui forces tasks to suspend while waiting for GPU operations to *complete*. Since GPUSync does not exhibit this behavior, we suspect this busy-waiting may occur within the CUDA runtime when tasks wait for GPU resources to become *available*. To explain how the real-time schedulers are sensitive to busy-waiting, consider that the SCHED_FIFO and LITMUS^{RT} schedulers always schedule the m-highest priority ready tasks on m CPUs (in this experiment, $m = 6$). CPU time is wasted if any of these tasks busy-waits for too long. The amount of CPU time wasted in this way is limited under SCHED_OTHER, since this general-purpose scheduler attempts to distribute CPU time equitably—busy-waiting tasks are soon preempted. GPUSync avoids this problem altogether by performing GPU resource arbitration "up-front" with real-time locking protocols.

Table III give us insight into the worst- 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 probability density functions (PDFs) of normalized completion delays in Fig. 9. Each PDF is derived from a histogram with a bucket width of 0.005. 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. To facilitate easy comparisons, we *clip* the domain of each PDF at $x = 2.5$, so the long tails of some of these distributions are not depicted. However, we have examined worst-case behaviors in Table III, so we do not revisit the topic here. We make several observations.

Observation 4. *The PDF for GPUSync show that normalized completion delays are most likely near* 1.0.

We see this in line 4 of Fig. 9. This result is not surprising, given prior Obs. 2. However, we also see that this PDF closely resembles the curve of a normal distribution. This trend is harder to see in the other PDFs. Another characteristic of the PDF for GPUSync is that it is clearly unimodal. This is unlike the PDF for SCHED_FIFO (line 2), which has at least four distinct modes (indicated by the four peaks in the PDF), or the PDF for LITMUS^{RT} (line 3), which appears to be bimodal. The PDF for SCHED_OTHER (line 1) has a similar shape to the PDF for GPUSync (line 4). However, the peak of line 1 has a wider base than line 4. Line 1 also lacks the symmetry shown by line 4.

Observation 5. *The PDF for* LITMUS^{RT} *without GPUSync suggests bursty completion behaviors.*

In Fig. 9, line 3 depicts the PDF for the LITMUS^{RT} 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. A sink node that has fallen behind may complete several jobs in quick succession as it catches up, especially under C-EDF, which gives priority to late work. Consequently, one very long completion delay may be followed by a sequence of short completion delays. The long tail of line 3 indicates that long completion delays occur (this tail has been clipped at $x = 2.5$, but we can still observe that line 3 lies above the others after $x = 2$). The first mode (the one near $x = 0.05$) indicates the corresponding "catch-up" behavior. Although LITMUS^{RT} did not drop any frames, the playback of the stabilized video is far from smooth. Indeed, LITMUS^{RT} exhibits some behaviors less desirable than SCHED_OTHER.

Due to space constraints, the above discussion has focused on only a subset of our results. A much more extensive overview that covers significantly more collected data and further experiments can be found in [9].

V. Conclusion

The need to support real-time graph-based computer vision applications in embedded domains such as in the automotive industry is of growing importance. Moreover, to reap size, weight, and power advantages, there is growing interest in using GPUs in supporting such applications. Given that OpenVX is a ratified standard, it is likely to see widespread use for this purpose in the future. The case for adopting OpenVX is further strengthened by NVIDIA's dominance in the GPU sector and their implicit backing of OpenVX through the development of VisionWorks.

Unfortunately, when real-time correctness is a concern, the use of OpenVX creates several challenges. In this paper, we have discussed these challenges and have presented techniques for dealing with them. We have also reported on our efforts in implementing a version of VisionWorks that incorporates these techniques and leverages the GPU-management mechanisms of GPUSync to support predictable real-time execution. Additionally, we have demonstrated the increased predictability made possible by our new VisionWorks variant through a case study evaluation involving a complex computer vision workload.

The study presented in this paper focused on observed runtime behavior rather than analytically predicted schedulability. We are currently planning to augment this runtime study by conducting a major schedulability study in which analysis-related tradeoffs pertaining to the workloads and platforms considered in this paper will be assessed. This will be a significant undertaking because both the considered workloads (graph-based computer vision applications) and platforms (heterogeneous multicore+GPU platforms) are complex. The basic schedulability analysis needed to drive such a study can be obtained by applying the techniques sketched in the appendix within the overall analytical framework associated with GPUSync described in [9]; in a forthcoming paper, we plan to describe the overall analysis framework that results from this combination of results in greater detail. To conduct the planned schedulability study, this analysis will need to be carefully extended to incorporate system overheads. Additionally, GPUSync is the source of continual development and new variants of it may raise new analytical questions.

References

[1] E. Rainey, J. Villarreal, G. Dedeoglu, K. Pulli, T. Lepley, and F. Brill, "Addressing system-level optimization with OpenVX graphs," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 658–663.

[2] Khronos Group, *The OpenVX™ Specification*, October 2014, version 1.0, Revision r28647.

[3] Kkronos Group, "OpenVX homepage," https://www.khronos.org/openvx/, 2015.

[4] F. Brill and E. Albuz, "NVIDIA VisionWorks toolkit," 2014, presented at the 2014 GPU Technology Conference.

[5] G. Elliott, N. Kim, C. Liu, and J. Anderson, "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*, 2014, pp. 1–10.

[6] G. Elliott, B. Ward, and J. Anderson, "GPUSync: A framework for real-time GPU management," in *Proceedings of the 34th IEEE International Real-Time Systems Symposium*, 2013, pp. 33–44.

[7] G. Elliott and J. Anderson, "Exploring the multitude of real-time multi-GPU configurations," in *Proceedings of the 35th IEEE International Real-Time Systems Symposium*, 2014, pp. 260–271.

[8] B. Brandenburg, "LITMUS^{RT}: Linux testbed for multiprocessor scheduling in real-time systems," http://www.litmus-rt.org, 2014.

[9] G. Elliott, "Scheduling of GPUs, with applications in advanced automotive systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2015.

[10] L. Bourdev and J. Brandt, "Robust object detection via soft cascade," in *Proceedings of the 2005 IEEE Conference on Computer Vision and Pattern Recognition*, vol. 2, 2005, pp. 236–243.

[11] C. Liu and J. Anderson, "Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss," in *Proceedings of the 31st IEEE International Real-Time Systems Symposium*, 2010, pp. 3–13.

[12] S. Goddard, "On the management of latency in the synthesis of real-time signal processing systems from processing graphs," Ph.D. dissertation, University of North Carolina at Chapel Hill, 1998.

[13] S. Goddard and K. Jeffay, "Managing memory requirements in the synthesis of real-time systems from processing graphs," in *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, 1998, pp. 59–70.

[14] K. Jeffay and S. Goddard, "A theory of rate-based execution," in *Proceedings of the 20th IEEE International Real-Time Systems Symposium*, 1999, pp. 304–314.

[15] K. Yang, G. Elliott, and J. Anderson, "Analysis for supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms," in *Proceedings of the 23rd International Conference on Real-Time Networks and Systems*, 2015, to appear.

[16] U. Devi and J. Anderson, "Tardiness bounds under global EDF scheduling on a multiprocessor," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, December 2006, pp. 330–341.

APPENDIX A.
SKETCH OF GRAPH TRANSFORMATION TECHNIQUES

We sketch here in greater detail the graph transformation techniques described in Sec. III. A full explanation of these techniques can be found in [15].

Prior related work. Liu and Anderson examined dataflow applications on globally scheduled multiprocessors [11]. They showed that task response times are bounded under G-EDF scheduling, *without* utilization loss. This result builds upon several prior results, which we review next.

Goddard investigated DAG-based systems specified via the *processing graph method* (*PGM*) in his dissertation [12]. In PGM graphs, data movement is abstracted by specifying the movement of "tokens" through the graph. The rules by which tokens are produced and consumed along a graph edge are very general. The DAG-based systems considered in our work are special-case PGM graphs in which "tokens" are produced and consumed one at a time. Goddard [12] showed that any PGM-specified system can be naturally represented by a corresponding *rate-based (RB)* task system [14]. In a RB system, the long-term invocation rate of any task stays within a specified bound, but over short intervals such bounds may be exceeded.

Based on prior uniprocessor results of Jeffay and Goddard [12, 14], Liu and Anderson developed techniques to transform a RB task system to a sporadic one for which response times are bounded under G-EDF [11]. In a RB task system, consecutive jobs of the same task may arrive either "too close together" (their separation is less than the task's period) or "too far apart" (their separation is more than the task's period). The latter possibility is already handled by the sporadic task model. Jeffay and Goddard showed that the former possibility can be dealt with by *deadline postponement*, *i.e.*, by ensuring that consecutive deadlines of the same task have a separation at least its period.

Liu and Anderson obtain a similar effect by computing *redefined release times* that have the proper separation for any task, by computing *redefined deadlines* based on the redefined release times, and by potentially allowing a job to execute *before* its redefined release time—this is called *early releasing*. When applied to the tasks that comprise the nodes of a DAG, these redefinitions cause job release times to be shifted into the future by a *bounded* amount. Intuitively, this is because each task's long-term invocation rate is bounded. This property ensures that job response times, when computed with respect to the original, unaltered release times, remain bounded.

Fig.10 depicts a partial schedule that illustrates these ideas. In this figure, $T^A_{l,j}$ denotes the j^{th} job of the task associated with the node A of the l^{th} DAG. The top line of the schedule shows the original RB releases and deadlines. Observe that $T^A_{l,1}$ and $T^A_{l,2}$ are released "too close together." The bottom line shows the redefined releases and deadlines. Note that a proper spacing is maintained here (as given by the sporadic task's period, which is 3 time units). Note also

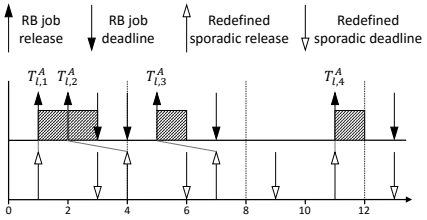


Figure 10: Illustration of redefined release times and deadlines. This figure is adapted Fig. 8 in [11].

that $T^A_{l,2}$ and $T^A_{l,3}$ are both "early released," *i.e.*, they are scheduled before their actual (sporadic) release time.

Devi and Anderson showed that for any sporadic task system that does not cause over-utilization, deadline tardiness is bounded for any task [16]; this implies that response times are bounded as well. This result holds regardless of whether early releasing is allowed or whether scheduling is preemptive or non-preemptive (thus, the non-preemptive execution on GPUs is supported).

Taken together, the results reviewed here can be used to show (as Liu and Anderson did [11]) that bounded response times can be ensured for PGM-based task systems under G-EDF with no utilization loss. When applied in our setting, there is some utilization loss due to GPU-related priority-inversion-related blocking, which we deal with using suspension-oblivious analysis [9]. Similar results can be applied within a single cluster under C-EDF scheduling.

Handling delay edges. Delay edges that are back edges (ostensibly) create cycles, which are not supported in the analysis reviewed above. In that which follows, we consider a delay edge d from node A to node B. Also, we introduce two parameters k and h, where $1 \leq k \leq h$, to represent the range of the back-trace history associated with the delay edge d. In particular, with respect to the delay edge d, $T^B_{l,j}$ may require data produced by the jobs $T^A_{l,j-h}, \ldots, T^A_{l,j-k}$, but not by jobs outside of this range. In most existing computer vision algorithms, $k = 1$.

If d does not cause cycles, *i.e.*, it is a delay edge such as that from "Duplicate Color Image" to "Warp Perspective" in Fig. 7 that is not actually a back edge, then it can simply be replaced by a forward edge f. To see why this is sufficient, observe that the original delay edge d indicates that $T^B_{l,j}$ cannot be invoked until $T^A_{l,j-h}, \ldots, T^A_{l,j-k}$ have completed. However, because tasks are sequential, if $T^A_{l,j}$ has completed (as indicated by the forward edge), then these prior jobs must have completed as well.

If d does cause cycles, *i.e.*, it is a back edge such as that from "Harris Feature Tracker" to "Compute Optical Flow" in Fig. 7, then the situation is more complex. Referring back to our review of related work above, if the redefined release time of $T^B_{l,j}$ is after the latest finish time of $T^A_{l,j-k}$ (as given by response-time analysis), then no cycle impacting the analysis is in fact introduced. In this case, d can simply be removed (from an analysis point of view) because all required precedence relationships are implicitly satisfied.

If the redefined release time of $T^B_{l,j}$ does not meet the

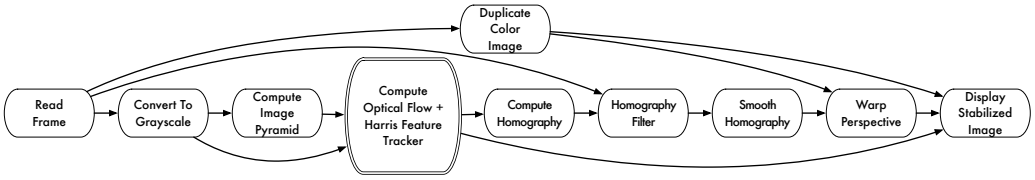


Figure 11: Graph with no delay edges.

requirement state in the prior paragraph, then a cycle that impacts analysis truly does exist. One way to handle this situation is by replacing nodes A and B by a single "super node." Each incoming edge to either A or B becomes an incoming edge to the super node, and each outgoing edge from either A or B becomes an outgoing edge from the super node. Within the super node, we serialize the execution of jobs of A and B. That is, to start the execution of *either* $T^A_{l,j}$ or $T^B_{l,j}$, *both* $T^A_{l,j-1}$ and $T^B_{l,j-1}$ must have completed. This ensures that the delay-edge requirement is implicitly satisfied. This transformation is analytically safe and sufficient to eliminate the cyclic delay edge d at the cost of reducing parallelism. While this approach may seem heavy-handed, recall that VisionWorks, as originally defined, effectively executes entire graphs as super nodes. As an example, consider again the graph in Fig. 7. If we combine the two nodes "Harris Feature Tracker" and "Compute Optical Flow" into a single super node, as shown in Fig. 11, and replace all delay edges that are not back edges by ordinary forward edges, then all delay edges and cycles are eliminated.

Forward edge replica bounds. As mentioned in Sec. III, safe pipelined execution can be ensured by replicating data objects. For the moment, we ignore issues created by the presence of delay objects and focus only on non-delay data objects. For such data objects, needed replica bounds can be obtained by extrapolating from prior work by Goddard and Jeffay on bounding the size of token buffers in PGM graphs [13]. Their work can be applied because, as stated above, the graphs we consider are special cases of those considered by them. Moreover, even though their work pertained to uniprocessors and ours pertains to multiprocessors, their computed buffer bounds were obtained by applying per-node bounds on invocation rates and response times to producing and consuming nodes. It is only necessary that these per-node bounds exist for correct buffer bounds to be obtained, and such per-node bounds exist in both the uniprocessor and multiprocessor cases.

Furthermore, because we are working with simpler sporadic DAGs here, it is possible to obtain simpler results by proving new bounds from first principles. Additionally, we must concern ourselves with the possibility that the same data object may be accessed by different tasks at different times (*e.g.*, the i^{th} video frame might be accessed by the i^{th} invocations of several tasks without being copied between accesses). It can be shown that the desired replica bound for any forward edge f in the l^{th} graph is

$$N(f) = \left\lfloor \frac{R_l}{p_l} \right\rfloor + 1,$$

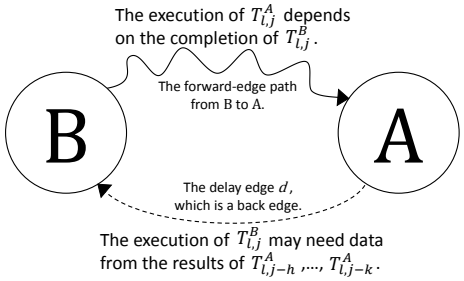


Figure 12: Illustration for the ring buffer bound for a delay edge q that is a back edge.

where p_l is the period of the l^{th} graph and R_l is the end-to-end response time of the l^{th} graph. Although we do not defend this bound here due to space constraints, a formal proof of it is provided in [15].

Delay edge ring buffer bounds. As mentioned in Sec. III, the back-trace history data indicated by a delay edge is stored in a ring buffer. The size of such buffers must be bounded as well. To explain how this is done, we consider such a delay edge d from a node A to a node B.

If d is not a back edge, then as explained above, we replace it by an forward edge f. Let $N(f)$ denote the replica bound computed for this forward edge as discussed above. Because f replaces a delay edge, this bound may need to be augmented. In particular, the bound $N(f)$ implies that when the currently active job of node B is $T^B_{l,j}$, no job of node A later than the $T^A_{l,j+N(f)-1}$ has already completed. Now, accounting for the semantics of the replaced delay edge, $T^B_{l,j}$ may require the result of some prior jobs of node A but no earlier than the $T^A_{l,j-h}$ (recall the definition of h given earlier). Thus, a buffer size of $N(f)+h$ is sufficient.

If d is a back edge, then we argued above that it either can be analytically removed or can be encapsulated within a super node. In either case, the constraints implied by the original delay edge must be met. It is relatively easy to show that in both cases, such constraints will hold assuming a buffer size of h. To see this, observe that because d is a back edge, there is a forward-edge path from node B to node A, as shown in Fig. 12. Suppose that the most recently ready job of node A is $T^A_{l,j}$. Due to the forward-edge path, $T^A_{l,j}$ being ready implies that $T^B_{l,j}$ has already completed, which means only $T^B_{l,j+1}$ or later jobs of node B could execute next and need delay buffer data. Therefore, the earliest delay buffer data that will be needed in the future is the result of the $T^A_{l,j+1-h}$. Moreover, since $T^A_{l,j}$ is the most recently ready job of node A, no job of node A later than $T^A_{l,j}$ is ready, let alone has completed. Thus, a buffer size of h is sufficient.