Analysis for Supporting Real-Time Computer Vision Workloads using OpenVX on Multicore+GPU Platforms [*]

Kecheng Yang, Glenn A. Elliott, and James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill
{yangk,gelliott,anderson}@cs.unc.edu

ABSTRACT

In the automotive industry, there is currently great interest in utilizing computer vision algorithms to support driver-assist and autonomous-control features. OpenVX is an emerging standard for supporting workloads in which such algorithms are applied. OpenVX uses a graph-based software architecture designed to enable efficient computation on heterogeneous platforms that may include CPUs, graphics processing units (GPUs), digital signal processors (DSPs), and other accelerators. Unfortunately, in settings where real-time constraints exist, the usage of OpenVX poses certain challenges. In a recent paper, the authors presented a new implementation of OpenVX directed at platforms comprised of CPUs and GPUs that leverages various analytical techniques to enable these challenges to be addressed. In this paper, these analytical techniques are presented and discussed in detail. These techniques enable end-to-end frame processing times to be analytically bounded under OpenVX while encouraging parallelism through pipelining. Additionally, they enable bounds on frame buffering requirements to be determined.

1 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 and become 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 that may include components that accelerate certain computations. Such a consolidation must be done in a way that enables real-time requirements to be validated.

For computer vision algorithms, graphics processing units (GPUs) are a particularly compelling accelerator to consider, as GPUs are well suited for efficiently performing the matrix-oriented computations inherent in many computer vision applications. To ease the development of such applications on heterogeneous platforms such as those in which GPUs are employed, 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.

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. 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."

This model simplifies the API and implementation of OpenVX and allows it to perform well on platforms with a wide range of capabilities, ranging from simple ASICs to complex multicore+GPU platforms comprised of multiple CPUs and one or more GPUs. However, this model imposes three significant implications on real-time scheduling. First, the specification has no notion of a repeating (*i.e.*, periodic or sporadic[2]) 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, it requires a graph to execute end-to-end before it may be re-executed. This significantly hinders the ability to exploit parallelism by "pipelining" portions of a graph's structure to improve performance.

In a recent paper, we described a new OpenVX implementation devised by us that addresses all of these problems [4]. This new implementation extends an OpenVX implementation by NVIDIA called VisionWorks [5] and is directed at multicore+GPU platforms. Our extended version of VisionWorks is structured in a way that enables previously proposed analytical techniques to be adapted to

[*]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.

RTNS'15 , November 4-6, 2015, Lille, France

[1]In OpenVX, these basic operations are called "kernels."
[2]We assume familiarity with the sporadic and periodic task models.

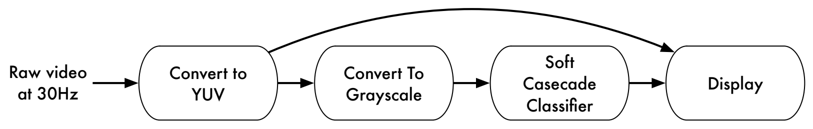


Figure 1: Dataflow graph of a simple pedestrian detector application.

bound end-to-end frame processing times and overall frame buffering requirements, both within an execution model that encourages parallelism through pipelining.

Contributions. From an implementation point of view, our extended version of VisionWorks is rather complex—in total, we added approximately 34K lines of code to VisionWorks. As a result, the presentation in our prior paper [4] is primarily directed at implementation details and a case study—needed analytical results are only briefly sketched. The main contribution of the current paper is to present these results in much greater detail. Specifically, we present an overview of the prior analytical results being leveraged, and explain in detail how these results can be applied to derive response-time bounds and buffer bounds. Because the two papers are linked—this one focusing on analytical issues and the prior one [4] focusing on implementation details—there is necessarily some overlap in presentation.[3]

Organization. The remainder of this paper is organized as follows. We begin by describing more carefully how OpenVX graphs are defined (Sec. 2), the real-time-related challenges pertaining to such graphs (Sec. 3), and the prior work we leverage to address these challenges (Sec. 4). We then explain how to apply this prior work in our setting to obtain bounds on end-to-end processing times (Sec. 5) and overall buffer-space requirements (Sec. 6). Following this, we conclude (Sec. 7).

2 OpenVX

Computer vision algorithms are commonly expressed using dataflow graphs. An example is given in Fig. 1, 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 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” [6] 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 being used. 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.[4] 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. 2(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`.[5] Fig. 2(b) depicts the bindings of parameters to nodes. Fig. 2(c) depicts the derived structure of this graph.

Our implementation of OpenVX, described in [4], is directed at multicore+GPU platforms and extends an OpenVX implementation by NVIDIA called VisionWorks. Specifically, a GPU-management framework developed previously by our group called GPUSync [7, 8, 9] is used along with an additional middleware layer. GPUSync treats GPUs as resources that may be acquired and released by tasks by invoking multiprocessor real-time locking protocols. A fairly comprehensive overview of this implementation is given in [4]; further details can be found in the second author's Ph.D. dissertation [9].

3 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

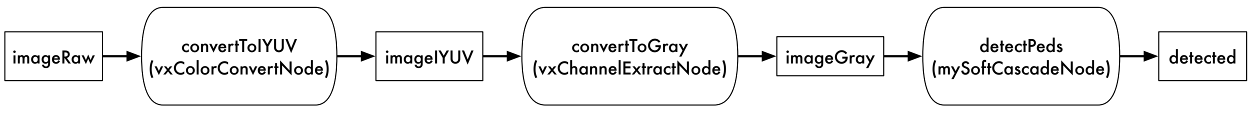
[3]The greatest overlap occurs within Secs. 1–3.

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

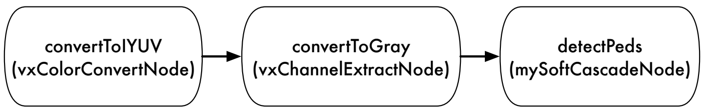
[5]The OpenVX standard does not currently specify a primitive for object detection, so the user must provide one or use one from a third party.

```
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)

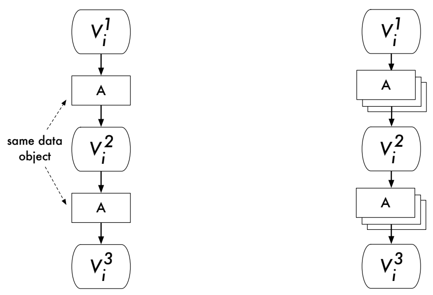
(c) Derived graph structure.

Figure 2: Construction of a graph in OpenVX for pedestrian detection.

is generated by a sink node. In particular, we require that such processing times are provably bounded. As we explain in detail later, such bounds can be obtained by adapting prior results of Elliott *et al.* [8], which are in turn based on even earlier results of Liu and Anderson [10], with synchronization-related blocking due to the usage of GPUSync accounted for using blocking bounds from [9]. However, to apply these results, 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, the VisionWorks framework that we modified fails to satisfy any of these requirements, hence the need for our modifications.

Graph dependencies and pipelining. Recall from Sec. 2 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 nodes 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 *replicating* data objects, as illustrated in Fig. 3. However, replication alone is not a sufficient solution unless *safe* replication bounds can be determined that are sufficient to ensure that no data object is prematurely overwritten



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

Figure 3: Replicating data objects to enable pipelining.

before being consumed. Later, in Sec. 6, we explain how to obtain such bounds.

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

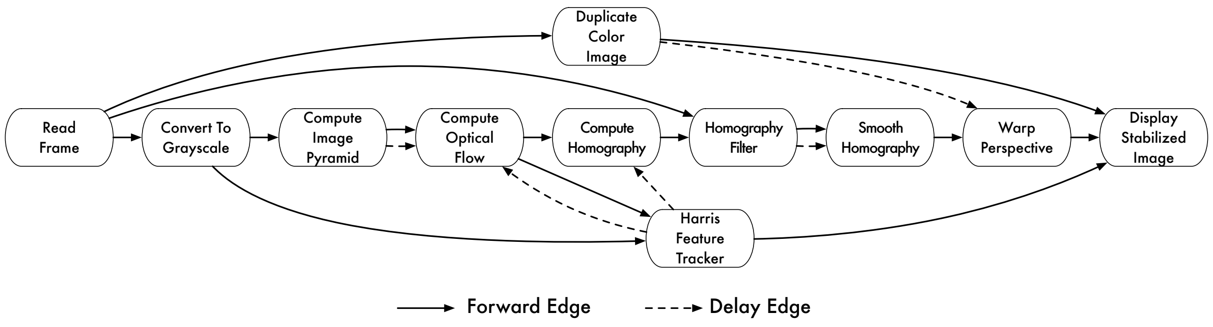


Figure 4: Dependency graph of video stabilization application.

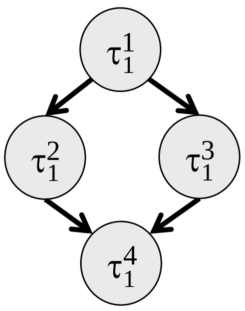


Figure 5: Precedence constraints within a DAG τ_1.

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, consider a node that operates on frame i and requires access to 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. 4, which is taken from the case study presented in [4], the delay edges sourced from the "Harris Feature Tracker" node are back-edges; the other delay edges are not. As seen in Fig. 4, back-edges ostensibly result in cycles. This is problematic because the prior end-to-end response-time analysis we leverage applies only to acyclic graphs. In Sec. 5, we explain how to break such cycles.

As the discussion above suggests, the analytical results we desire extrapolate heavily from prior work on graph-based task systems. Before delving into the details of how we addressed the problems noted above, we first review this prior work.

4 The Sporadic DAG Model

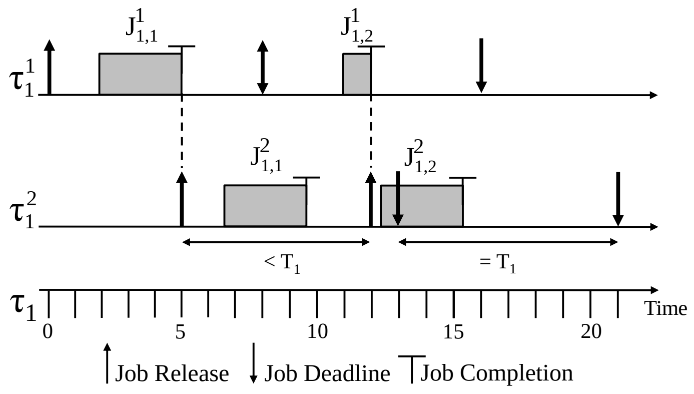
There is a growing body of work on real-time analysis methods for systems specified using graph-based formalisms and other formalisms that expose parallelism (*e.g.*, see [11, 12, 13, 14, 15, 16, 17, 18] and the references cited therein). Our formal analysis here is obtained by considering the implicit-deadline sporadic DAG task model, which has been the subject of prior research by our group [19]. The following description of this model is largely taken from [19] with minor modifications to suite our needs here.

Task model. We consider a system comprised of a set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n DAGs. Each DAG is a set $\tau_i = \{\tau_i^1, \tau_i^2, \ldots, \tau_i^{z_i}\}$ of z_i tasks, with producer/consumer relationships. Each task τ_i^v releases a (potentially infinite) series of *jobs* $J_{i,1}^v, J_{i,2}^v, \ldots$. An unfinished job $J_{i,j}^v$ is *ready* if it has been released and if $J_{i,j-1}^v$ (if $j \geq 2$) has completed execution. An example DAG τ_1 is depicted in Fig. 5. As seen in this example, tasks (nodes) may be connected by edges. Each edge is directed from a *producer* task that produces data to a *consumer* task that consumes that data. A particular task τ_i^v's producers are those on edges for which τ_i^v is the consumer, and its consumers are those on edges for which τ_i^v is the producer. Each job must wait to begin execution until one job from each of its producers has completed, so that its necessary input data is available. For example, in Fig. 5, for any j, $J_{1,j}^4$ needs input data from each of $J_{1,j}^2$ and $J_{1,j}^3$, so it must wait until those jobs complete.

To simplify analysis, we assume that each DAG τ_i has exactly one *source task* τ_i^1, which only has outgoing edges, and one *sink task* $\tau_i^{z_i}$, which has only incoming edges. Multi-source/multi-sink DAGs may be supported with the addition of singular "virtual" sources and sinks that connect multiple sources and sinks, respectively. Each DAG has a common period parameter T_i for all of its tasks—we explain how this parameter is interpreted when discussing scheduling below. Each task τ_i^v also has a parameter C_i^v, which denotes the worst-case execution time (WCET) for any of its jobs. We assume that τ is scheduled on an identical multiprocessor. For now, we also assume that all tasks are independent. Later, we explain how to deal with dependencies created when tasks share GPUs.

Scheduling. The results of this paper can be applied to any system of DAGs where tasks are scheduled via any *window-constrained* global scheduler [20]; however, for ease of exposition, we specifically focus on the most widely studied such scheduler, the *global earliest-deadline-first* (*G-EDF*) scheduler. Under G-EDF, ready jobs are prioritized for scheduling on an earliest-deadline-first basis, any job may execute on any processor, and jobs may be preempted or may migrate among processors, except when executing within a non-preemptive section (*e.g.*, when accessing a GPU). On large platforms, global algorithms such as this can be applied within clusters of processors, so our results can be adapted for applicability on such platforms as well.

As in [19], we assume that tasks corresponding to source nodes release jobs sporadically; that is, job releases of the task τ_i^1 must occur at least T_i time units apart. As noted above, a task corresponding to a non-source node releases its jobs as the data they require becomes available. As seen in the example schedule in Fig. 6, this can cause consecutive jobs of the same non-source task to be released fewer than T_i time units apart. However, the *deadlines* corresponding to those jobs must still be defined to be at least T_i time units apart, as the figure shows for the the task τ_1^2. In particular, note that jobs $J_{1,1}^2$ and $J_{1,2}^2$ are released only 7 time units apart, which is less than $T_1 = 8$,



(Assume depicted jobs are scheduled alongside other jobs, which are not shown.)

Figure 6: Partial schedule of the DAG in Fig. 5 assuming a period T_1 of 8 time units.

yet their deadlines are defined to be 8 time units apart. The technique used here for defining deadlines is called *deadline postponement* and dates back to early work on rate-based scheduling [21].

The sporadic DAG systems considered here are special cases of DAG-based systems that can be specified using the more general *processing graph method* (*PGM*) [22], the real-time scheduling of which has been studied in the context of both uniprocessors [23] and multiprocessors [10]. In PGM, the movement of data through a DAG is abstracted by considering the transmission of *tokens* from producer to consumer tasks. The rules that govern how tokens are produced and consumed are quite general, and as a result, the manner in which non-source tasks release jobs becomes more complicated. This level of generality is not needed in the application domains that are the subject of this paper.

End-to-end latency bounds. Define the *utilization* of the task τ_i^v to be $U_i^v = C_i^v/T_i$, and the total system utilization to be $U_{sum} = \sum_{i,v} U_i^v$. Assume that the considered hardware platform has m processors. Then, as long as $U_i^v \leq 1$ holds for each i and v, and $U_{sum} \leq m$ holds, it can be shown that any task in any DAG has bounded deadline tardiness. In the context of the more general PGM model, this result was first established by Liu and Anderson [10] by leveraging prior work on tardiness bounds under G-EDF by Devi and Anderson [24]. In the context of the simpler sporadic DAG model, Elliott *et al.* [19] used these earlier results to establish per-task end-to-end latency bounds. Specifically, let τ' denote the set of *independent* implicit-deadline sporadic tasks corresponding to the sporadic DAG task system τ, *i.e.*, each task $\tau_i'^v$ in τ' has the same period and WCET as the corresponding task τ_i^v in τ. Then, the deadline tardiness of any task $\tau_i'^v$ in τ' is guaranteed to be at most Δ_i^v time units, where Δ_i^v is defined according to an expression given in Theorem 1 in [24]. Based on this, Elliott *et al.* [19] established an end-to-end latency bound L_i^v for each task τ_i^v in the original sporadic DAG task system τ. L_i^v upper bounds the difference $f_{i,j}^v - a_{i,j}^1$, where $a_{i,j}^1$ denotes the release time (or activation time) of the j^{th} job of the DAG τ_i's source task τ_i^1, and $f_{i,j}^v$ denotes the finish time (or completion time) of the j^{th} job of the task τ_i^v in τ_i. Such bounds are given by the following theorem.

THEOREM 1 (THEOREM 1 IN [19]). *If Θ is the set of all tasks along the worst-case path*[6] *from τ_i^1 to τ_i^v, including both τ_i^1 and τ_i^v, then any job $J_{i,j}^v$ completes within*

$$L_i^v = \sum_{\tau_i^w \in \Theta} (T_i + \Delta_i^w)$$

time units after time $a_{i,j}^1$.

[6]That is, the path that maximizes the given sum

It is important to note that the existence of this bound relies crucially on the fact that all task graphs are acyclic. As mentioned earlier, this is not necessarily true of task graphs defined via the OpenVX specification.

Dealing with blocking times due to GPU accesses. The latency bounds mentioned above entail no CPU capacity loss because the only preconditions for their existence are that $U_{sum} \leq m$ holds and $U_i^v \leq 1$ holds for each i and v. However, when accounting for delays that jobs may experience as they wait to access GPUs, CPU capacity loss will generally occur. Under GPUSync [7, 8, 9], such delays are accounted for through suspension-oblivious analysis [25] wherein priority-inversion-related blocking times due to the usage of locking protocols are analytically modeled as CPU computation time. This causes an artificial inflation of per-task WCETs, and correspondingly inflated task utilizations. Such inflations can cause a loss of some fraction of the underlying hardware platform's available CPU capacity. However, any such loss is usually more than offset by the significant acceleration afforded by the usage of GPUs [9]. Because the effects of GPUs are dealt with by inflating WCETs, we can henceforth ignore them and assume we are working with WCETs that have already been properly inflated.

Buffer bounds. As mentioned in Sec. 3, pipelined execution can be enabled under OpenVX by replicating data objects, but this requires safe replication bounds. Such bounds can be obtained by extrapolating from prior work by Goddard and Jeffay on bounding the size of token buffers in PGM graphs [26]. However, because we are working with simpler sporadic DAGs here, it is possible to obtain tighter 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).

Leveraging these results. To summarize, to leverage prior work on end-to-end latency bounds, we must find a way of eliminating the apparent cycles caused by delay edges in OpenVX graphs. To be able to enable pipelined execution in OpenVX graphs, we must determine safe bounds for replicating data objects. These issues are considered in the following two sections.

5 Dealing with Delay Edges

In order to leverage the prior results just discussed, we introduce the concept of a *dependency graph*. Given a set of OpenVX graphs, the i^{th} dependency graph, G_i, is associated with the i^{th} OpenVX graph. The v^{th} node in G_i is viewed as a sequential task τ_i^v, as in the sporadic DAG task model. Dependencies among tasks in G_i are as implied by the corresponding OpenVX graph. Specifically, G_i has the same forward and delay edges as the i^{th} OpenVX graph. A forward edge from the v^{th} node to the w^{th} node, $v \rightarrow w$, indicates that job $J_{i,j}^w$ cannot commence execution until job $J_{i,j}^v$ completes; a delay edge from the v^{th} node to the w^{th} node, $v \dashrightarrow w$, indicates that job $J_{i,j}^w$ cannot commence execution until jobs *prior to* $J_{i,j}^v$ have completed. To be more precise about the back-trace history associated with the delay edge $v \dashrightarrow w$, we introduce two per-edge parameters h and k, where $h \geq k$, to specify the precise back-trace history implied by the delay edge $v \dashrightarrow w$: $J_{i,j}^w$ may need the results of the jobs $J_{i,j-h}^v, \ldots, J_{i,j-k}^v$, but does not need the results of jobs outside of this range. Note that, in most existing computer vision algorithms, $k = 1$ for every delay edge. (Although h and k are per-edge parameters, we have avoided using superscripts or subscripts to indicate the intended edge, for simplicity.)

We define a dependency graph to be *well-formed* if and only if it contains no cycles or delay edges. A set of well-formed dependency graphs corresponds naturally to a sporadic DAG task system, assuming (as we do here) that each graph's source node is invoked periodically (and hence sporadically) according to some given video frame rate. However, the set of dependency graphs arising from a given OpenVX-specified application may not be well-formed. Our goal in this section is to show how to transform such a set of graphs to a corresponding set where each graph *is* well-formed. We show this by considering the concept of a *refinement*. The dependency graph G_i' is a *refinement* of the dependency graph G_i if both have the same nodes and G_i' is at least as restrictive as G_i, *i.e.*, all dependency restrictions in G_i are implied by G_i' or can be guaranteed under G-EDF scheduling. For now, we ignore the issue of replicating data objects to prevent overwriting (equivalently, each data object can be assumed for now to be infinitely replicated to prevent overwriting); that issue is addressed in Sec. 6.

Rules for constructing well-formed refinements. In the rest of this section, we consider three rules that can be repeatedly applied as needed to a dependency graph G_i to obtain a well-formed refinement of it. Each such rule application eliminates one or more delay edges in G_i. Once all delay edges have been eliminated, no cycles can exist. After all three rules have been stated and explained, we illustrate them with an example at the end of this section. (The reader may wish to consult the example as each rule is introduced.) The first rule handles delay edges that do not actually cause cycles.

> **Delay-Edge Strengthening Rule:** If the delay edge $v \dashrightarrow w$ is not part of any cycle, then replace it by a forward edge $v \rightarrow w$.

Note that applying this rule always yields a valid refinement. To see why, observe that the original delay edge $v \dashrightarrow w$ indicates that the job $J_{l,j}^w$ cannot commence execution until after the jobs $J_{l,j-h}^v, \cdots, J_{l,j-k}^v$ have completed, while the forward edge $v \rightarrow w$ indicates that $J_{l,j}^w$ cannot commence execution until after $J_{l,j}^v$ has completed. Because tasks are sequential, the latter clearly implies the former.

The remaining two rules can be applied to eliminate cycles. The first of these eliminates delay edges that are not actually necessary.

> **Delay-Edge Dropping Rule:** If, under G-EDF scheduling, job $J_{i,j-k}^v$ is guaranteed (via response-time analysis) to be complete by the release time of job $J_{i,j}^v$ for all j, then the delay edge $v \dashrightarrow w$ can be removed.

Intuitively, this rule can be applied if k is "large enough" to ensure that the back-trace history required by $J_{i,j}^v$ is sufficiently "far in the past" that the precedence constraint implied by the delay edge is satisfied by G-EDF scheduling anyway. The following theorem can be applied to determine if k is "large enough."

THEOREM 2. *If, for each delay edge $v \dashrightarrow w$ in a dependency graph, k satisfies*

$$k \geq \left\lceil \frac{L_i^v}{T_i} \right\rceil, \tag{1}$$

then the Delay-Edge Dropping Rule can be applied to eliminate all such edges. Specifically, for each such edge, the job $J_{i,j-k}^v$ is guaranteed to be complete by time $a_{i,j}^w$, where (generalizing our earlier notation) $a_{i,j}^w$ denotes the release time of the job $J_{i,j}^w$.

Proof. We prove this theorem by contradiction. Assume that (1) holds and consider the corresponding graph where all delay edges have been eliminated. This graph is acyclic, and hence Theorem 1 can be applied. Assume that $J_{i,j-k}^v$ has not completed by $a_{i,j}^w$. Because the j^{th} job release of the task τ_i^w cannot precede the j^{th} job release of the source task τ_i^1, $a_{i,j}^1 \leq a_{i,j}^w$. From our assumption, this implies that $J_{i,j-k}^v$ has not completed by time $a_{i,j}^1$, *i.e.*, $f_{i,j-k}^v > a_{i,j}^1$. By Theorem 1, $J_{i,j-k}^v$ is guaranteed to complete within L_i^v time units after time $a_{i,j-k}^1$, *i.e.*, $f_{i,j-k}^v \leq a_{i,j-k}^1 + L_i^v$. Therefore,

$$a_{i,j}^1 - a_{i,j-k}^1 < L_i^v. \tag{2}$$

Because the source task τ_i^1 is invoked sporadically with a minimum release separation of T_i, we have

$$a_{i,j}^1 - a_{i,j-k}^1 \geq k \cdot T_i. \tag{3}$$

By (2) and (3),

$$k < \frac{L_i^v}{T_i} \leq \left\lceil \frac{L_i^v}{T_i} \right\rceil, \tag{4}$$

which contradicts (1). □

Theorem 2 gives the system designer the option of adjusting the k parameter of any delay edge to be "large enough" so that that edge can be effectively eliminated. However, in practical terms, this means that the computer vision algorithm is being altered to rely on back-trace history that is "older." This could result in a loss of accuracy in some vision algorithms. Therefore, we need a rule that provides an option for breaking cycles that does not involve such algorithmic alterations. Our final rule provides such an option.

> **Super-Node Creation Rule:** Combine several nodes from the same graph that have dependencies with respect to each other due to delay edges into a single "super-node" that is executed as an ordinary task.[7] Each edge (forward or delay) from a node outside of the

[7]Referring back to our definition of a "refinement," we note that the orginally nodes actually still do exist; the notion of a super-node is an abstraction.

super-node to a node within the super-node becomes an incoming edge of the super-node. Similarly, each edge (forward or delay) from a node within the super-node to a node outside of the super-node becomes an outgoing edge of the super-node. The j^{th} job of the super-node is executed sequentially by executing the j^{th} jobs of all tasks within the super-node in an order allowed by forward edges. The WCET of the super-node is the sum of the WCETs of the contained tasks. (Recall that all tasks within the same graph have the same period.) The super-node's utilization must be at most one.

The application of this rule will result in a valid refinement, because any precedence constraints among tasks within a super-node implied by delay edges among them will be implicitly satisfied due to the enforced serial execution order. Such an enforced serialization order reduces parallelism, which may seem like a heavy-handed technique for eliminating delay edges. However, for the common case in computer vision algorithms where $k = 1$ for such an edge, the following theorem shows that an implicit serialization order often exists anyway.

THEOREM 3. *Suppose there is a forward-edge path from the w^{th} node to the v^{th} node and $v \dashrightarrow w$ is a delay edge that therefore causes a cycle. Assuming $k = 1$ for this edge, no jobs of any two tasks in this cycle can execute in parallel.*

Proof. Let $J^p_{i,j}$ and $J^q_{i,l}$ be two arbitrary jobs of two tasks τ^p_i and τ^q_i in the mentioned cycle. Jobs of the same task clearly execute in sequence, so assume that $p \neq q$ holds. We consider two cases.

Case 1: $j = l$**.** In this case, $J^p_{i,j}$ and $J^q_{i,l}$ are in the same invocation of the OpenVX graph, *e.g.*, handling the same video frame. By definition, p and q are two nodes in a forward-edge path from the w^{th} node to the v^{th} node. Thus, $J^p_{i,j}$ and $J^q_{i,l}$ $(j = l)$ cannot execute in parallel.

Case 2: $j \neq l$**.** In this case, with out loss of generality, let us assume $j < l$. Let $J \prec J'$ denote that job J must complete execution before job J' commences execution, and let $\preceq$ denote the reflexive closure of $\prec$. Because p and q are two nodes along a forward-edge path from the w^{th} node to the v^{th} node, by the precedence constraints implied by forward edges,

$$J^p_{i,j} \preceq J^v_{i,j}, \tag{5}$$

and

$$J^w_{i,l} \preceq J^q_{i,l}. \tag{6}$$

Because there is a forward-edge path from w to v,

$$J^w_{i,s} \prec J^v_{i,s} \quad \text{for all } s. \tag{7}$$

Because $v \dashrightarrow w$ is a delay edge with $k = 1$,

$$J^v_{i,s} \prec J^w_{i,s+1} \quad \text{for all } s. \tag{8}$$

By (7) and (8),

$$J^w_{i,s} \prec J^w_{i,s+1} \quad \text{for all } s. \tag{9}$$

Because $j < l$ and both j and l are integers, $j + 1 \leq l$. Hence, by (8) and (9),

$$J^v_{i,j} \prec J^w_{i,j+1} \preceq J^w_{i,l}. \tag{10}$$

By (5), (6), and (10),

$$J^p_{i,j} \prec J^q_{i,l}.$$

Thus, $J^p_{i,j}$ and $J^q_{i,l}$ do not execute in parallel. □

For any dependency graph, it is possible to repeatedly apply the above rules and eliminate all delay edges and cycles, resulting in a final graph that is well-formed, provided applications of the Super-Node Creation Rule do not create a super-node with utilization exceeding one (according to Theorem 3, if this occurs, over-utilization may likely have been inherent in the original graph anyway). However, whenever the Super-Node Creation Rule is applied, parallelism is sacrificed. Thus, its use should be avoided if possible. We conclude this section by illustrating these rules with an example.

Example. Consider again the graph in Fig. 4. As a first step, we apply the Delay-Edge Strengthening Rule to each delay edge that does not cause a cycle, *i.e.*, all delay edges except the one from the node "Harris Feature Tracker" to the node "Compute Optical Flow." We can then eliminate any potential cycles by applying the Delay-Edge Dropping Rule to this last remaining delay edge, yielding the well-formed graph shown in Fig. 7. Note, however, that applying this rule could involve potentially altering the computer vision algorithm to use a value of k that satisfies (1) for the dropped delay edge. If this is not feasible, then we could alternatively apply the Super-Node Creation Rule to combine the two nodes connected via this delay edge into a single super-node, provided the utilization of this super-node is at most one, and obtain the well-formed refinement shown in Fig. 8. For either well-formed graph, Theorem 1 could be applied to determine latency bounds.

6 Replica and Buffer Bounds

The analysis in the prior section focused on maintaining required precedence constraints when eliminating cycles when individual graph nodes, rather than entire graphs, are viewed as schedulable entities, *i.e.*, as tasks. However, as noted in Sec. 2, graph edges are not explicitly declared in OpenVX but are inferred from how data objects are bound to nodes as parameters. Furthermore, as noted in Sec. 3, when individual nodes are viewed as schedulable entities, there is a danger that the data objects associated with a given edge may be overwritten. As noted there, this problem can be addressed by replicating such objects. However, for such an approach to be feasible, safe replication bounds are needed. In this section, we present such bounds. We assume that the transformations discussed in the prior section have already been applied, but we still require information exposed by the original untransformed graph. We consider ordinary data objects and those associated with delays in separate subsections.

6.1 Data Object Replicas

In discussing data object replication, we assume that no data object is accessed by multiple OpenVX graphs (or equivalently, the set of such graphs would have to be treated as one graph here). Therefore, we assume that we are working with a fixed graph and avoid introducing identifiers to indicate which graph where possible.

To avoid overwriting with respect to forward edges in the transformed graph, we can replicate each data object N times, indexing the replicas from 0 to $N - 1$, and storing them in per-data-object buffers with N entries each. We require the j^{th} invocation (*i.e.*, job) of any task in the graph under consideration to access the $(j \bmod N)^{th}$ replica. (Note that we are replicating every data object accessed within the graph to the same degree; different per-object replica bounds can be obtained with finer-grained analysis.)

With data objects replicated like this, we merely need to guarantee that when the $(j + N)^{th}$ job of *any* task is executing, no job prior to the $(j + 1)^{st}$ of *any* task can access any data object, *i.e.*, the j^{th}

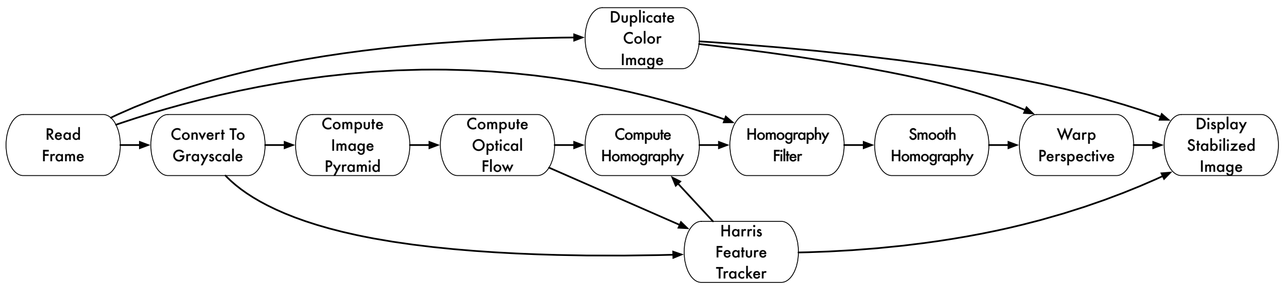


Figure 7: Well-formed dependency graph corresponding to Fig. 4, where the Delay-Edge Dropping Rule has been applied.

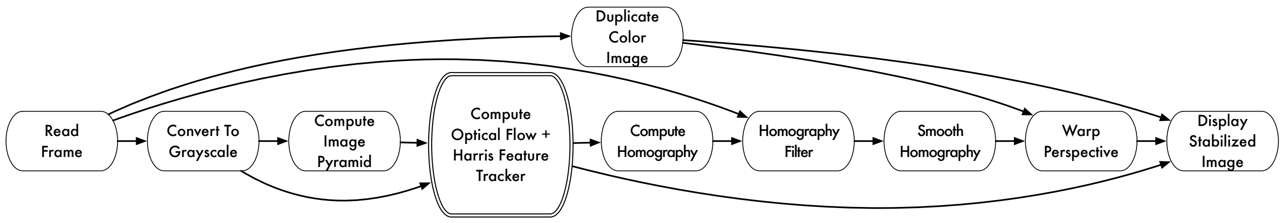


Figure 8: Well-formed dependency graph corresponding to Fig. 4, where the Super-Node Creation Rule has been applied.

and prior jobs of *any* task have already completed execution. The following theorem provides a minimum value of N for which this property can be guaranteed. That is, assuming that we are working with the i^{th} graph, if we set $N = \lfloor L_i^{z_i}/T_i \rfloor + 1$, then no overwriting will occur with respect to that graph. Recall from Sec. 4 that $f_{i,l}^p$ denotes the finish time (or completion time) of $J_{i,l}^p$ (*i.e.*, the l^{th} job of the task τ_i^p in τ_i).

THEOREM 4. *With respect to the i^{th} graph, if N satisfies the following condition,*

$$N \geq \left\lfloor \frac{L_i^{z_i}}{T_i} \right\rfloor + 1, \tag{11}$$

then $J_{i,j+N}^v$ will not execute at or before $f_{i,l}^p$ for all p and for all $l \leq j$.

Proof. We prove this theorem by contradiction. Suppose that (11) holds and at time t, where $t \leq f_{i,l}^p$, $J_{i,j+N}^v$ is executing. Then, because the z_i^{th} node is the sink node, $f_{i,l}^p \leq f_{i,l}^{z_i}$. Therefore,

$$t \leq f_{i,l}^{z_i}. \tag{12}$$

By Theorem 1,

$$f_{i,l}^{z_i} - a_{i,l}^1 \leq L_i^{z_i}. \tag{13}$$

Furthermore, $J_{i,j+N}^v$ cannot execute until at or after the $(j+N)^{th}$ invocation of the source node (task τ_i^1), *i.e.*, time $a_{i,j+N}^1$. Therefore,

$$t \geq a_{i,j+N}^1. \tag{14}$$

By (12), (13), and (14),

$$L_i^{z_i} \geq a_{i,j+N}^1 - a_{i,l}^1. \tag{15}$$

Because the source node (task τ_i^1) releases jobs sporadically and $l \leq j$,

$$a_{i,j+N}^1 - a_{i,l}^1 \geq (N+j-l) \cdot T_i \geq N \cdot T_i. \tag{16}$$

By (15) and (16),

$$N \leq \frac{L_i^{z_i}}{T_i} < \left\lfloor \frac{L_i^{z_i}}{T_i} \right\rfloor + 1, \tag{17}$$

which contradicts (11). □

6.2 Ring Buffers for Delay Edges

As mentioned in Sec. 2, each delay edge in OpenVX is actually defined by 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, where the oldest data object is overwritten when a new data object is produced. Therefore, if the ring buffer size is not large enough, data objects that are being used may be prematurely overwritten. Thus, we also require safe bounds on ring buffer sizes, so that no such overwriting will occur.

Although in Sec. 5 we analytically transformed each original dependency graph G_i to a well-formed refinement, in the context of considering ring buffer sizes, we still need to consider G_i, which directly represents the original OpenVX graph, wherein the needed delay data objects are fully exposed. We consider a delay edge $v \dashrightarrow w$ in G_i. The following theorem provides a sufficiently safe buffer size for each delay edge.

THEOREM 5. *For any delay edge $v \dashrightarrow w$ in G_i, a ring buffer size of $N+h$ is sufficient, where $N = \lfloor L_i^{z_i}/T_i \rfloor + 1$.*

Proof. We consider an arbitrary job of τ_i^v, $J_{i,j}^v$. By Theorem 4, $J_{i,j}^v$ will not execute at or before $f_{i,l}^w$ for all $l \leq j-N$. That is, when $J_{i,j}^v$ is executing, $J_{i,j-N}^w$ and all prior jobs of τ_i^w have already completed. Therefore, only $J_{i,j-N+1}^w$ or later jobs of τ_i^w may execute afterwards. Those jobs may require the result of some prior jobs of node v but no earlier than job $J_{i,j-N+1-h}^v$ (recall the definition of h given earlier in Sec. 5). So, when $J_{i,j}^v$ is writing data into the ring buffer, we only need to keep the result of $J_{i,j-N+1-h}^v$ and later jobs in this ring buffer. Thus, a ring buffer size of $N+h$ is sufficient. □

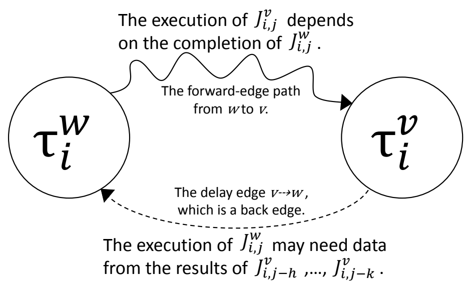

The execution of $J^v_{i,j}$ depends on the completion of $J^w_{i,j}$.
The forward-edge path from w to v.
τ^w_i
τ^v_i
The delay edge $v \dashrightarrow w$, which is a back edge.
The execution of $J^w_{i,j}$ may need data from the results of $J^v_{i,j-h}$, ..., $J^v_{i,j-k}$.

Figure 9: Illustration for the ring buffer bound for a delay edge q that is a back-edge (*i.e.*, causes a cycle).

The following theorem provides a significantly tighter buffer size bound in a common special case.

THEOREM 6. *If $v \dashrightarrow w$ causes a cycle in G_i and is the only delay edge in that cycle, then a ring buffer size of h is sufficient.*

Proof. If $v \dashrightarrow w$ is the only delay edge in a cycle, then there is a forward-edge path from the $w^{th} node$ to the v^{th} node, as shown in Fig. 9. Suppose that the most recently ready job of τ_i^v is $J^v_{i,j}$. Due to the forward-edge path, $J^v_{i,j}$ being ready implies that $J^w_{i,j}$ has already completed, which means only $J^w_{i,j+1}$ or later jobs of τ_i^w could execute next and need delay buffer data. Therefore, the earliest delay buffer data that will be needed in the future is that from $J^v_{i,j+1-h}$. (By the definition of h given earlier in Sec. 5, $J^w_{i,j+1}$ may require the result of some prior jobs of node v but no earlier than job $J^v_{i,j+1-h}$.) Moreover, since $J^v_{i,j}$, by definition, is the most recently ready job of τ_i^v, no job of τ_i^v later than $J^v_{i,j}$ is ready, let alone is executing. Thus, a buffer size of h is sufficient. □

7 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.

When real-time correctness is a concern, the use of OpenVX creates several challenges. In a prior paper [4], we presented a new OpenVX implementation, based on a variant of VisionWorks, that addresses these challenges. That paper specifically focused on implementation details and a case study, with needed analytical results that justify the implementation only briefly sketched. In fact, a complete explanation of these analytical results was deferred to a separate paper—namely, this one.

These analytical results can be factored into two main contributions. First, we presented transformations that can be applied to OpenVX-derived graphs to eliminate delay edges and cycles so that prior work on end-to-end latency bounds can be applied. These transformations involve treating individual graph nodes as schedulable entities. This can create data hazards that can be avoided by replicating data objects, but safe replications bounds are needed for such an approach to be feasible. As a second contribution, we showed how to derive such bounds. Together with [4], the results of this paper provide a solid foundation for supporting OpenVX graphs on multicore+GPU platforms in a way that encourages parallelism through piplelining while allowing real-time guarantees to be validated.

References

[1] E. Rainey, J. Villarreal, G. Dedeoglu, K. Pulli, T. Lepley, and F. Brill, "Addressing system-level optimization with OpenVX graphs," in *Proc. of the IEEE Conf. 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] G. Elliott, K. Yang, and J. Anderson, "Supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms," in *Proc. of the IEEE Real-Time Sys. Symp.*, 2015, to appear.

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

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

[7] G. Elliott, B. Ward, and J. Anderson, "GPUSync: A framework for real-time GPU management," in *Proc. of the 34th IEEE Int'l Real-Time Sys. Symp.*, 2013, pp. 33–44.

[8] G. Elliott and J. Anderson, "Exploring the multitude of real-time multi-GPU configurations," in *Proc. of the IEEE Real-Time Sys. Symp.*, 2014, pp. 260–271.

[9] G. Elliott, "Scheduling of GPUs, with applications in advanced automotive systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2015 (see especially Chapter 5, available at http://www.cs.unc.edu/~anderson/papers.html).

[10] C. Liu and J. Anderson, "Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss," in *Proc. of the IEEE Real-Time Sys. Symp.*, 2010, pp. 3–13.

[11] S. Baruah, "Improved multiprocessor global schedulability analysis of sporadic DAG task systems," in *Proc. of the Euromicro Conf. on Real-Time Sys.*, 2014, pp. 97–105.

[12] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility analysis in the sporadic DAG task model," in *Proc. of the Euromicro Conf. on Real-Time Sys.*, 2013, pp. 225–233.

[13] S. Collette, L. Cucu, and J. Goossens, "Integrating job parallelism in real-time scheduling theory," *Information Processing Letters*, vol. 106, no. 5, pp. 180–187, 2008.

[14] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *RTSS*, 2010, pp. 259–268.

[15] J. Li, K. Agrawal, C. Lu, and C. Gill, "Analysis of global EDF for parallel tasks," in *Proc. of the Euromicro Conf. on Real-Time Sys.*, 2013, pp. 3–13.

[16] J. Li, A. Saifullah, K. Agrawal, C. Gill, and C. Lu, "Analysis of federated and global scheduling for parallel real-time tasks," in *Proc. of the Euromicro Conf. on Real-Time Sys.*, 2014, pp. 85–96.

[17] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *Proc. of the IEEE Real-Time Sys. Symp.*, 2011, pp. 217–226.

[18] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The digraph real-time task model," in *Proc. of the IEEE Real-Time Technology and Applications Symp.*, 2011, pp. 71–80.

[19] G. Elliott, N. Kim, C. Liu, and J. Anderson, “Minimizing response times of automotive dataflows on multicore,” in *Proc. of the IEEE Int’l Conf. on Embedded and Real-Time Computing Sys. and Applications*, 2014, pp. 1–10.

[20] H. Leontyev and J. Anderson, “Generalized tardiness bounds for global multiprocessor scheduling,” in *Proc. of the IEEE Real-Time Sys. Symp.*, 2007, pp. 413–422.

[21] K. Jeffay and S. Goddard, “A theory of rate-based execution,” in *Proc. of the IEEE Real-Time Sys. Symp.*, 1999, pp. 304–314.

[22] “Processing graph method specification,” Prepared by the Naval Research Laboratory for use by the Navy Standard Signal Processing Program Office (PMS-412), 1987.

[23] 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.

[24] U. Devi and J. Anderson, “Tardiness bounds under global EDF scheduling on a multiprocessor,” in *Proc. of the IEEE Real-Time Sys. Symp.*, 2006, pp. 330–341.

[25] B. Brandenburg and J. Anderson, “Optimality results for multiprocessor real-time locking,” in *Proc. of the IEEE Real-Time Sys. Symp.*, 2010, pp. 49–60.

[26] S. Goddard and K. Jeffay, “Managing memory requirements in the synthesis of real-time systems from processing graphs,” in *Proc. of the IEEE Real-Time Technology and Applications Symp.*, 1998, pp. 59–70.