Managing Latency and Buffer Requirements in Processing Graph Chains

STEVE GODDARD[1] AND KEVIN JEFFAY[2]

[1]*Computer Science and Engineering, University of Nebraska—Lincoln, Lincoln, NE 68588-0115, USA*
[2]*Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3175, USA*
Email: goddard@cse.unl.edu

Real-time signal-processing applications for high assurance systems are commonly designed using a processing-graph software architecture. Here we demonstrate the management of latency and buffer requirements in such an architecture—the US Navy's *processing graph method* (PGM). By applying recent results in real-time scheduling theory to the subset of PGM employed by the US DARPA rapid prototyping of application-specific signal processors (RASSP) synthetic aperture radar (SAR) benchmark application, we identify inherent real-time properties of nodes in a PGM graph, and demonstrate how these properties can be exploited to perform useful and important system-level analyses such as schedulability analysis, end-to-end latency analysis, and memory requirements analysis. More importantly, we develop relationships between properties such as latency and buffer bounds and show how one may be traded off for the other.

Received 31 October 2000; revised 28 April 2001

1. INTRODUCTION

Signal-processing algorithms are often defined in the literature using processing graphs [1]: directed graphs in which a node is a sequential program that executes from start to finish in isolation (i.e. without synchronization), and the graph edges depict the flow of data from one node to the next. Thus, an edge represents a producer/consumer relationship between two nodes. Processing graphs provide a natural description of signal-processing applications with each node representing a mathematical function to be performed on an infinite stream of data that flows on the arcs of the graph. The streams of input data are typically generated by sensors sampling the environment at periodic rates, and sending the samples to the signal processor via an external channel. The processing graph methodology allows one to easily understand the signal processing performed by depicting the structure of the algorithm; any portion of the application can be understood in the absence of the rest of the algorithm.

Embedded signal-processing applications are naturally defined using processing graph techniques. As high-assurance real-time applications, they require deterministic performance. The signal processing graph must process data at the rates of a set of external devices (e.g. sonobuoys, dipping sonars or radars) without the loss of data. Hence signal-processing applications, like other real-time systems, have a dual notion of correctness: logical and temporal. It is not sufficient only to produce the correct output—e.g. the signature of a detected target; embedded signal-processing applications must produce the correct output within the correct time interval—e.g. detect the signature within 1 second.

Explicit methods for evaluating latency and buffering requirements are needed when applications developed using a processing-graph model are executed in an embedded environment with limited memory resources. Processing-graph models implicitly define a temporal semantics for a processing graph by specifying lower bounds to when nodes may execute as a function of the availability of data on input edges. However, most models do not support the specification of either an end-to-end latency constraint or an upper bound to the time that may elapse between a node becoming eligible to execute and the time the node either commences or completes execution. Without such a bound, the buffer requirements of the application cannot be determined.

Even the US Navy's own processing graph methodology, processing graph method (PGM) [2], lacks deterministic analysis methods to verify latency and buffer requirements. This is somewhat surprising since PGM is used to develop real-time, embedded, anti-submarine warfare (ASW) applications for the AN/UYS-2A (the US Navy's standard signal processor). PGM has also been used to create a real-time Ka-band synthetic aperture radar (SAR) benchmark application for DARPA's rapid prototyping of application-specific signal processors (RASSP) project.

Using the SAR application graph as a driving problem, the management of latency and buffer requirements is

demonstrated by applying real-time scheduling theory to the subset of PGM used in the RASSP SAR benchmark application. The AN/UYS-2A uses dynamic scheduling and resource allocation in the execution of PGM applications. The same dynamic execution environment is assumed in this work, with the exception that a simple (on-line) earliest deadline first (EDF) scheduler is used rather than the default (on-line) first come first served (FCFS) scheduler implemented in the AN/UYS-2A.

In this work, inherent relationships existing in real-time processing graphs that have not been recognized in the literature are identified. Latency and buffering requirements are dependent on the rate and order in which nodes execute. Thus, theorems that characterize the non-trivial execution rates of every node in the processing graph as a function of input rates are presented. Existing real-time scheduling theory is then used to determine the order in which nodes execute. From scheduling theory, conditions for various EDF scheduling algorithms are used to determine if the graph can be scheduled to meet specified latency requirements. We show that, by changing parameters used to schedule node execution, we can manage both latency and buffer requirements. More importantly, we develop relationships between latency and buffer bounds and show how one may be traded off for the other.

The rest of this paper is organized as follows. Our results are related to other work in Section 2. Section 3 presents a brief overview of the portion of PGM used by the SAR graph, which is introduced in Section 4. Section 5 presents our execution model including node execution rates and a schedulability condition for EDF scheduling. Section 6 addresses latency management issues and Section 7 shows how to bound and manage the buffer requirements of an implementation of a graph. We summarize our contributions in Section 8.

2. RELATED WORK

This paper is part of a larger body of work that creates a framework for evaluating and managing processor demand, latency and memory usage in the synthesis of real-time systems from general processing graphs (including cyclic graphs) [3]. Here, we demonstrate the management of latency and buffer requirements in the synthesis of a real-time uniprocessor system from processing graph chains developed with PGM. In [4], some of the results presented here have been extended to compute node execution rates and inherent latency for cyclic processing graphs that contain feedback loops.

From the real-time literature, PGM graphs are most closely related to the logical application stream model (LASM) [5] and the generalized task graph (GTG) model [6]. PGM, LASM and GTG were all developed independently and support very similar dataflow properties; PGM was the first of these to be developed. Our work improves on the analysis of LASM and GTG graphs by not requiring periodic execution of the nodes in the graph. Instead, we calculate a more general execution rate, which can be reduced to average execution rates assumed in the LASM and GTG models. Our general execution rate specification provides a more natural representation of node execution for PGM graphs. Forcing periodic execution of all graph nodes adds latency to the processed signal, but simplifies the analysis of latency and memory requirements.

Processing graphs are a standard design aid in digital signal processing. From the digital signal processing literature, PGM is most similar to Lee and Messerschmitt's synchronous dataflow (SDF) graphs [1] supported by the Ptolemy system [7]. The SDF graphs of Ptolemy utilize a subset of the features supported by PGM. Any SDF graph can be represented as a PGM graph where each queue's threshold is equal to its consume value. In addition to supporting a more general processing-graph model, our research differs from [1] in that we support dynamic real-time scheduling techniques rather than the creation of static schedules.

In 1996, Bhattacharyya *et al.* published a method for software synthesis from dataflow graphs [8]. Their software synthesis method is based on the static scheduling of Lee and Messerschmitt's SDF graphs. The main goal of Bhattacharyya *et al.*'s software synthesis method and related scheduling research based on SDF graphs has been to minimize memory usage by creating off-line scheduling algorithms [1, 8, 9]. Off-line schedulers create a static node execution schedule that is executed periodically by the processor. In contrast, the primary goal of our research has been to manage the latency and memory usage of processing graphs by executing them with an on-line scheduler. Recently we have shown that for a large class of applications, dynamic on-line scheduling creates less imposed latency than static scheduling. An even more surprising result is that, in many cases, dynamic on-line scheduling uses less memory for buffering data on graph edges than static scheduling [10].

Our latency analysis is related to the work of Gerber *et al.* in guaranteeing end-to-end latency requirements on a single processor [11]. However, Gerber *et al.* map a task graph to a periodic task model in the synthesis of real-time message-based systems rather than assuming a rate-based execution. Our analysis and management of latency differs from Gerber *et al.*'s in that PGM graphs allow non-unity dataflow attributes. Finally, Gerber *et al.* introduce new (additional) tasks to the task set in their synthesis method to synchronize processing paths. Our method does not need extra synchronization tasks since our analysis techniques are rate based rather than periodic and we assume tasks are released by the run-time system as soon as they are eligible for execution.

3. NOTATION AND THE PROCESSING GRAPH METHOD

The notation and terminology of this paper, for the most part, is an amalgamation of the notation and terminology used in [8] and [12]. A processing graph is formally described as a *directed graph* (or *digraph*) $G = (V, E, \psi)$. The ordered

triple (V, E, ψ) consists of a non-empty finite set V of *vertices*, a finite set E of *edges*, and an incidence function ψ that associates with each edge of E an ordered pair of (not necessarily distinct) vertices of V. Consider an edge $e \in E$ and vertices $u, v \in V$ such that $\psi(e) = (u, v)$. We say e joins u to v, or u and v are adjacent. The vertex u is called the tail or source vertex of e and v is the head or sink vertex of edge e. The edge e is an *output edge* of u and an *input edge* of v. The number of input edges to a vertex v is the *indegree* $\delta^-(v)$ of v, and the number of output edges for a vertex v is the *outdegree* $\delta^+(v)$ of v. A vertex v with $\delta^-(v) = 0$ is an *input node*. For $u, v \in V$, there is a *path* between u and v, written as $u \rightsquigarrow v$, if and only if there exists a sequence of vertices $(w_1, w_2, \ldots, w_k)$ such that $w_1 = u$, $w_k = v$ and w_i is adjacent to w_{i+1} for $i = 1, 2, \ldots, (k-1)$. A path $u \rightsquigarrow v$ is a *chain* if $u \neq v$, $\delta^-(u) \leq 1$, $\delta^+(u) = 1$, $\delta^-(v) = 1$ and $\delta^+(w) = \delta^-(w) = 1$ for all $w \in \{\{u \rightsquigarrow v\} - \{u, v\}\}$.

For concreteness, the US Navy's PGM is used to present our techniques for managing latency and buffer requirements in processing graph chains. PGM was developed by the US Navy to facilitate the design and implementation of signal-processing applications, but it is a very general processing-graph paradigm that is applicable to many other domains.

In PGM, a system is expressed as a directed graph in which the nodes (or vertices) represent processing functions and the edges represent buffered communication channels called queues. The topology of the graph defines a software architecture independent of the hardware hosting the application. The graph edges are first in first out (FIFO) queues. There are four attributes associated with each queue: a produce amount $prd(q)$, a threshold amount $thr(q)$, a consume amount $cns(q)$ and an initialization amount $init(q)$. Let queue q be directed from node u to node w. The produce amount $prd(q)$ specifies the number of tokens (data elements) appended to queue q when producing node u completes execution. A token represents an instance of a data structure, which may contain multiple data words. There must be at least $thr(q)$ tokens on queue q before node w is eligible for execution. A queue is *over threshold* if the number of enqueued tokens meets or exceeds the threshold amount $thr(q)$. After node w executes, the number of tokens consumed (deleted) from queue q by node w is $cns(q)$. The number of initial data tokens on the queue is $init(q)$. The length of queue q is denoted $length(q)$.

Unlike many processing graph paradigms, PGM allows non-unity produce, threshold and consume amounts as well as a consume amount less than the threshold. The only restrictions on queue attributes are that they must be non-negative values and the consume amount must be less than or equal to the threshold.

If a node has more then one input queue (input edge), then the node is eligible for execution when *all* of its input queues are over threshold (i.e. when each input queue q contains at least $thr(q)$ tokens). After the processing function finishes executing, $prd(q)$ tokens are appended to each output queue q. Before the node terminates, but after data is produced, $cns(q)$ tokens are dequeued from each input queue q. The execution of a node is *valid* if and only if the node executes only when it is eligible for execution, no two executions of the same node overlap, each input queue has its data atomically consumed after each output queue has its data atomically produced and data is produced at most once on an output queue during each node execution.

A graph execution consists of a (possibly infinite) sequence of node executions. A graph execution is *valid* if and only if all of the nodes in the execution sequence have valid executions and no data loss occurs.

4. SAR GRAPH

This section introduces the SAR graph, including a brief description of the processing performed by each node in the graph. This information is provided for concreteness for the reader with a signal-processing background. The actual logical operation of the SAR graph is immaterial to the results we derive and the analyses we perform. The only essential properties of the SAR graph are those that influence node execution: the produce, consume and threshold values for each node. For a more detailed description of the processing performed by the SAR benchmark, see [13].

The full SAR benchmark cannot execute in real time on a single processor. Therefore, the RASSP project allocates a portion of the full SAR graph to individual processors. The graph in Figure 1 is one such allocation. This graph, called the 'mini-SAR', was originally created to test tools developed by the RASSP project. It performs the range and azimuth compression processing in the formation of an image that is one eighth the size of that formed by the full SAR benchmark. Henceforth, we shall refer to the mini-SAR graph as the SAR graph since an analysis similar to what we develop shortly could be performed on each processor to analyze the full application.

The input node for the SAR graph (shown in Figure 1) is labeled *YRange* and represents a periodic external device that produces data for the graph. The output node represents an external device that executes whenever data is available on the *Image* queue. The nodes and queues of this graph have mnemonic labels. Produce, threshold, and consume values are displayed below the queue. For example, the produce, consume, and threshold values of the queue labeled *Range* are all 118. Queue *RCS* is the only queue that initially contains data. It is initialized with 256×64 zero-samples (i.e. $init(RCS) = 256 \times 64$).

The top row of nodes in the SAR graph each operate on one pulse of data at a time. The pulse delivered by the external source, labeled *YRange*, has already been converted to complex-valued data and consists of 118 range gate samples. The *Zero Fill* node pads the pulse with zeroes to create a pulse length of 256 samples in preparation for the *FFT* node. Before performing the FFT, the data is passed through a Kaiser window function, represented by the node *Window Data*, to reduce sidelobe levels and perform bandpass filtering. After being compressed in the range dimension by the *Range FFT* node, the pulse is passed through the radar cross-section calibration filter performed by the *RCS Mult* node.

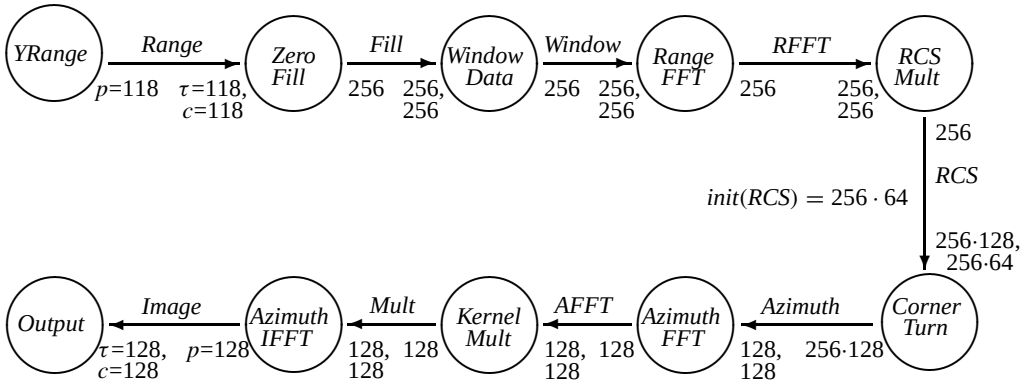


FIGURE 1. A PGM graph for the SAR application. The tail of each queue is annotated with its produce value. The head of each queue is annotated with its threshold and consume values. For example, the queue labeled *RCS* has $prd(q) = 256$, $thr(q) = 256 \times 128$, and $cns(q) = 256 \times 64$. Queue *RCS* is the only queue that initially contains data. It is initialized with 256×64 zero-samples (i.e. $init(RCS) = 256 \times 64$).

Unlike the previous nodes in the SAR graph, which require only one pulse of data before being eligible for execution, the *Corner Turn* node requires 128 pulses of data. A 2-D processing array is formed where each row of the array contains one sample from the 128 different pulses and each column contains the 256 range gates that form a pulse. The processing array consists of two 64×256 frames (or sequences of pulses). As a new frame is loaded in, the previous two frames are 'released' with the oldest frame being shifted out. This processing is achieved with threshold and produce values of 256×128 and a consume value of 256×64.

Convolution processing is performed on each row of the 2-D matrix by the *Azimuth FFT*, *Kernel Mult* and *Azimuth IFFT* nodes. The *Azimuth FFT* node performs a FFT on the signal, which has been aligned in the azimuth dimension. Next the *Kernel Mult* node multiplies each row of the matrix by a convolution kernel. Before the SAR image is output to the *Output* node, an inverse FFT is performed by the *Azimuth IFFT* node.

The SAR benchmark has a latency requirement (an upper bound) of 3 s, where latency refers to the elapsed time between when a frame of data (64 pulses) is input to the SAR processor and the time the corresponding image is output [13]. Assuming a pulse is received every 3.6 ms, it is natural to ask whether it is possible to meet this latency requirement when the graph is implemented on a certain processor and what is the maximum amount of buffer memory required by the queues? The last question is important when memory resources are scarce, as in the AN/UYS-2A. The answers to these questions are dependent on the execution model assumed. The next section presents the execution model and fundamental latent buffering bounds that affect the rate at which nodes execute and the latency inherent in the graph.

5. EXECUTION MODEL

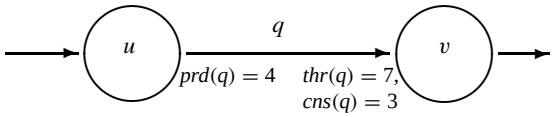
Latency and memory usage are dependent on the rate at which each node executes and the order in which producer/consumer pairs of nodes execute. Real-time scheduling theory provides a framework upon which we have developed an execution model that determines both the rate and order of node executions so that latency and memory usage can be managed.

This section introduces an execution paradigm and analysis techniques that support the evaluation of real-time properties for a graph. The first subsection explores fundamental execution relationships that exist between producer/consumer nodes, independent of the execution model. These relationships determine node execution rates and the latency inherent in any processing graph. The concepts and theorems presented in Section 5.1 are used throughout the rest of the paper. The remaining subsections address node execution rates and the rate-based execution (RBE) task model. These concepts are used to model an implementation of the SAR graph.

5.1. Node executions and latent buffer usage

In processing graph systems that require unity dataflow attributes (i.e. produce, threshold and consume values all one), deriving the execution rates of nodes is relatively straightforward. Deriving the execution rates of nodes in PGM graphs is not. In this section, we present an example that illustrates the impact of non-unity dataflow attributes on node execution and quantify the number of times a producer node must execute before its consumer node is eligible for execution. We also derive several bounds related to latent buffering that will be used throughout this paper.

To eliminate the influence of scheduling on node executions, assume each node executes on its own processor as soon as all of its input queues are over threshold. (This assumption is made to simplify the presentation of node execution rates and latent buffer usage. The execution rates and bounds on latent buffer usages also apply when the nodes execute on a single processor.) Consider the two-node chain of Figure 2. Queue q is annotated with its



Time	Data produced by node u	Data consumed by node v	$length(q)$
0			0
t_1	4		4
t_2	4		8
t_3		3	5
t_4	4		9
t_5		3	6
t_6	4		10
t_7		3	7
t_8		3	4

FIGURE 2. A two-node PGM graph and snapshot sequence. In this chain, the dataflow attributes $prd(q)$, $thr(q)$, and $cns(q)$ all have different values. The produce amount is 4, the threshold value is 7, and the consume amount is 3. Queue q has no initial data: $init(q) = 0$.

produce, threshold, and consume values below the queue; it has no initial data. Node u produces four tokens every time it executes. Node v has a threshold of seven tokens and consumes three tokens after it executes. Since queue q is not initialized, node u must fire twice before queue q is over threshold and node v executes for the first time. After node v executes, it consumes only three tokens—leaving five tokens on queue q. The third execution of node u produces four more tokens (for a total of nine tokens on queue q) and node v executes again, consuming three more tokens. The next execution of node u results in 10 tokens on queue q, and node v is able to execute twice—leaving four tokens on queue q, which is the same number of tokens that were on queue q after the first execution of node u. Thus, subsequent executions of node u and node v follow this same pattern: $uvuvuvv$. Therefore, if node u executes once every y_u time units, node v will execute with a rate of four times every $3y_u$ time units.

The number of tokens on queue q at time t is a function of the queue's dataflow attributes and the number of executions of nodes u and v prior to time t. Since node v executes whenever queue q contains at least $thr(q)$ tokens and it consumes $cns(q)$ tokens each time it executes, queue q will always contain at least $(thr(q) - cns(q))$ tokens after node v executes for the first time. Note, however, that this lower bound on the minimum number of tokens on q is not tight. Consider, for example, the case where the dataflow attributes of a queue q in a chain are $prd(q) = 8$, $thr(q) = 7$, $cns(q) = 6$. In this case, $thr(q) - cns(q) = 1$, but there will always be at least two tokens in the queue. The following theorem bounds the minimum number of tokens on queue q after the first execution of node v and the maximum number of tokens that can be on queue q without the queue being over threshold.

THEOREM 5.1. *Let $\psi(q) = (u, v)$ and queue q be initialized with $init(q) \geq 0$ tokens. After nodes v and u have executed at least once, the minimum number of tokens on queue q is at least $MinTokens(q)$ and the maximum number of tokens queue q can hold without being over threshold is $MaxUnderThr(q)$ where*

$$MinTokens(q) = f(q) + \left\lceil \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \tag{1}$$

$$MaxUnderThr(q) = \begin{cases} thr(q) - \gcd(prd(q), cns(q)) \\ \quad if\ \gcd(prd(q), cns(q)) \mid (thr(q) - f(q)) \\ f(q) + \left\lfloor \dfrac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \right\rfloor \cdot \gcd(prd(q), cns(q)) \\ \quad otherwise \end{cases} \tag{2}$$

and

$$f(q) = \begin{cases} init(q) - \left(\left\lfloor \dfrac{init(q) - thr(q)}{cns(q)} \right\rfloor + 1 \right) \cdot cns(q) \\ \quad if\ init(q) \geq thr(q) \\ init(q) \quad otherwise. \end{cases}$$

Proof. The proof of this theorem (as well as many others in this paper) has been omitted for space considerations. However, the full version of this paper, which includes all proofs, is available via the Web [14]. □

Consider the two-node chain of Figure 2 once again (where $prd(q) = 4$, $thr(q) = 7$ and $cns(q) = 3$). This time, assume queue q is initialized with seven tokens (thus $init(q) = 7$ and $thr(q) = 7$). Using Equation (1), the minimum number of tokens queue q contains after nodes u and v both execute at least once is

$$\begin{aligned} MinTokens(q) &= \\ & f(q) + \left\lceil \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \\ &= init(q) - \left(\left\lfloor \frac{init(q) - thr(q)}{cns(q)} \right\rfloor + 1 \right) \cdot cns(q) \\ &+ \left\lceil \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \\ &= 7 - \left(\left\lfloor \frac{7-7}{3} \right\rfloor + 1 \right) \cdot 3 + \left\lceil \frac{7 - f(q)}{\gcd(4,3)} \right\rceil \cdot \gcd(4,3) - 3 \\ &= 4 + \left(\left\lceil \frac{7-4}{1} \right\rceil \cdot 1 \right) - 3 = 4. \end{aligned}$$

Since $\gcd(prd(q), cns(q)) = \gcd(4, 3) = 1$, the gcd of the produce and consume values divides $(thr(q) - f(q))$. Therefore, by Equation (2), the maximum number of tokens queue q can hold without being over threshold is

$$\begin{aligned} MaxUnderThr(q) &= thr(q) - \gcd(prd(q), cns(q)) \\ &= 7 - 1 = 6. \end{aligned}$$

When $\gcd(prd(q), cns(q)) = 1$, the amount of initialized data does not affect these functions:

$$\begin{aligned}
&MinTokens(q) = \\
&f(q) + \left\lceil \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \\
&= f(q) + \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \cdot \gcd(prd(q), cns(q)) - cns(q) \\
&= f(q) + thr(q) - f(q) - cns(q) \\
&= thr(q) - cns(q)
\end{aligned}$$

and $MinTokens(q) = thr(q) - 1$ since 1 always divides $thr(q) - f(q)$.

When a queue is initialized with $thr(q) - cns(q)$ tokens, the initialized data does not affect the functions $MinTokens(q)$ and $MaxUnderThr(q)$:

$$\begin{aligned}
&MinTokens(q) = \\
&f(q) + \left\lceil \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \\
&= thr(q) - cns(q) \\
&\quad + \left\lceil \frac{thr(q) - (thr(q) - cns(q))}{\gcd(prd(q), cns(q))} \right\rceil \\
&\quad \cdot \gcd(prd(q), cns(q)) - cns(q) \\
&= thr(q) - cns(q) \\
&\quad + \frac{cns(q)}{\gcd(prd(q), cns(q))} \cdot \gcd(prd(q), cns(q)) - cns(q) \\
&= thr(q) - cns(q) + cns(q) - cns(q) \\
&= thr(q) - cns(q)
\end{aligned}$$

and $MaxUnderThr(q) = thr(q) - \gcd(prd(q), cns(q))$ since

$$thr(q) - f(q) = thr(q) - (thr(q) - cns(q)) = cns(q)$$

and $\gcd(prd(q), cns(q))$ always divides $cns(q)$.

Whenever $\gcd(prd(q), cns(q)) \mid thr(q) - f(q)$, Equation (1) reduces to $MinTokens(q) = thr(q) - cns(q)$ and Equation (2) reduces to $MaxUnderThr(q) = thr(q) - \gcd(prd(q), cns(q))$.

Theorem 5.1 provides upper and lower bounds for the number of tokens that a queue joining two nodes can contain without being over threshold (after both nodes have executed at least once). Given $length(q)$ tokens on queue q, the following theorem computes the number of executions of node v as a function of the number of tokens produced by node u when queue q is the only queue joining the pair (i.e. in a chain).

THEOREM 5.2. *Let $length(q) \geq thr(q)$, $\psi(q) = (u, v)$, and $\delta^-(v) = 1$. At the current time, assuming node u does not execute, node v will execute $\lfloor (length(q) - thr(q))/cns(q) \rfloor + 1$ times, consume*

$$\left(\left\lfloor \frac{length(q) - thr(q)}{cns(q)} \right\rfloor + 1 \right) \cdot cns(q)$$

tokens, and leave l tokens on queue q where $MinTokens(q) \leq l \leq MaxUnderThr(q)$.

Proof. See [14]. □

Given $length(q)$ tokens on queue q, it is also useful to know how many more executions of node u are required before queue q is over threshold. In this case, the consume amount does not matter; we only care about $thr(q)$, $prd(q)$, and the existing number of tokens on queue q, $length(q)$.

THEOREM 5.3. *Let there be $length(q)$ tokens on queue q and $\psi(q) = (u, v)$. Node u must execute*

$$\max\left(0, \left\lceil \frac{thr(q) - length(q)}{prd(q)} \right\rceil\right) \tag{3}$$

times before queue q is over threshold.

Proof. If there are $length(q)$ tokens on queue q and $length(q) \geq thr(q)$, then queue q is already over threshold and no more executions of node u are required. If $length(q) < thr(q)$, then $thr(q) - length(q)$ more tokens are required before queue q is over threshold. Since node u produces $prd(q)$ tokens every time it executes, it follows that u must execute $\lceil thr(q) - length(q)/prd(q) \rceil$ times before queue q is over threshold. In either case, the number of executions required of node u before queue q is over threshold is $\max(0, \lceil thr(q) - length(q)/prd(q) \rceil)$ and Equation (3) holds. □

To illustrate Theorem 5.3 consider the chain of Figure 2 where $prd(q) = 4$, $thr(q) = 7$, and $cns(q) = 3$. Assuming four tokens on queue q in the chain of Figure 2, node u must execute

$$\begin{aligned}
\max\left(0, \left\lceil \frac{thr(q) - length(q)}{prd(q)} \right\rceil\right) &= \max\left(0, \left\lceil \frac{7 - 4}{4} \right\rceil\right) \\
&= \max\left(0, \left\lceil \frac{3}{4} \right\rceil\right) = 1
\end{aligned}$$

time before queue q is over threshold and node v is eligible for execution.

In this section we have informally derived node execution rates by simulating executions. Section 5.2 formally defines an execution rate and uses the theorems presented in this section to analytically compute node execution rates.

5.2. Node execution rates

PGM does not explicitly define temporal properties for the graph. However, the execution rate of every node in a graph is defined by the graph topology, the definition of nodes, the dataflow attributes and the rate at which the source node produces data. Thus, given only the rate at which a source node delivers data, the execution rates of all other nodes can be derived. This fundamental property of real-time processing graph chains is the basis of the result presented in this section.

Consider the two-node chain of Figure 2. For the producer/consumer pair of nodes u and v, the number of tokens present on queue q at time t is a function of the queue's dataflow attributes and the number of executions of nodes u and v prior to time t. Node v can only execute when

its input queue is over threshold, so the number of times it is able to execute in any interval of time is dependent on the number of times node u executes (and the dataflow attributes on queue q). In an implementation of the graph, the actual number of times that node v executes in any interval of time is dependent on the number of times node u executes and on the scheduling algorithm employed. If the scheduling algorithm delays executions of node v but continues to let node u execute, data will accumulate on queue q.

To bound latency and memory usage in an implementation of the graph, we need to schedule the execution of nodes in a deterministic manner. For this, we appeal to real-time scheduling theory and execute the nodes according to a model of real-time execution. Finally, to select the proper model of real-time execution, we need to determine the natural execution pattern of nodes. We informally called the execution pattern a 'rate' in Section 5.1. Here, we formally define an execution rate and show how to analytically derive the execution rates of nodes in a PGM chain.

To simplify the presentation of execution rates, and to eliminate the role scheduling and node execution times play in the derivation of node execution rates, we extend our assumption that each node executes on its own processor and assume that the processors are each infinitely fast so that node execution takes no time. More precisely, we assume that nodes execute in accordance with the strong synchrony hypothesis from the synchronous programming literature [15]. The strong synchrony hypothesis states that the system instantly reacts to external stimuli. For example, the snapshot sequence in Figure 3 shows both nodes u and v executing at time y. The system reacts instantaneously to the arrival of data on the input queue to node u and both nodes u and v execute at the same instant. At time $3y$, one execution of node u and two executions of node v occur at the same instant. Node execution rates are defined as follows.

DEFINITION 5.1. *An* execution rate *is a pair of non-negative integers* (x, y).

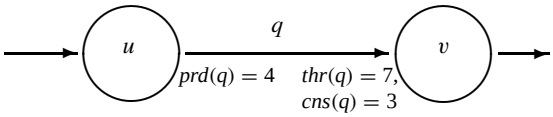
DEFINITION 5.2. *Execution rates* (x_1, y_1) *and* (x_2, y_2) *are* equal *if and only if* $x_1 = x_2$ *and* $y_1 = y_2$.

DEFINITION 5.3. *An execution rate specification for node* v, $R_v = (x, y)$, *is* valid *if there exists a time* t *such that node* v *executes exactly* x *times in time intervals* $[t + (k-1)y,\ t + ky)$ *for all* $k > 0$.

Notice that the interval is closed at the beginning and open at the end. Thus, if node u in Figure 3 continues to execute once every y time units, it has a valid execution rate of $R_u = (1, y)$ (starting at time 0). It executes exactly once in the interval $[0, y)$ since the execution at time y is counted in the interval $[y, 2y)$. While the periodic execution of node u satisfies the definition of a valid execution rate, the execution of node u does not need to be strictly periodic for it to have a valid execution rate of $R_u = (1, y)$. For example, if node u executed at times

$$0,\ 1.5y,\ 2y,\ 3.9y,\ 4y,\ 5y,\ 6y, \ldots,\ ky, \ldots$$

it still has a valid execution rate of $R_u = (1, y)$ starting at time 0 since it executes exactly once in each time interval $[0 + (k-1)y,\ 0 + ky)$ for all $k > 0$.



Time	Execution of	*length*(q)
0	u	4
y	u, v	5
$2y$	u, v	6
$3y$	u, v, v	4
$4y$	u, v	5
$5y$	u, v	6
$6y$	u, v, v	4
$7y$	u, v	5
$8y$	u, v	6
$9y$	u, v, v	4
$10y$	u, v	5

FIGURE 3. A two-node chain and a snapshot sequence that shows the execution of nodes u and v under the strong synchrony hypothesis. The execution rate of node u is $R_u = (1, y)$, and the execution rate of node v is $R_v = (4, 3y)$.

If the execution of node u is periodic, however, the execution of node u is 'well-defined' in that it executes at time ky for all $k \geq 0$. While the rate specification $R_u = (1, y)$ is a valid execution rate for node u, it does not describe the restricted execution pattern exhibited by node u.

DEFINITION 5.4. *An execution rate specification for node* v, $R_v = (x, y)$, *is* well-defined *if there exists a time* t_v *such that node* v *executes exactly* x *times in time intervals* $[t,\ t + y)$ *for all* $t \geq t_v$.

COROLLARY 5.4. *A well-defined rate specification* $R_v = (x, y)$ *for node* v *is also a valid rate specification for node* v.

Proof. If $R_v = (x, y)$ is a well-defined rate specification for node v, then by Definition 5.4 there exists a time t_v such that node v executes exactly x times in time intervals $[t,\ t + y)$ for all $t \geq t_v$. Thus, for any $t \geq t_v$ node v executes exactly x times in time intervals $[t + (k-1)y,\ t + ky)$ for all $k > 0$, and $R_v = (x, y)$ is a valid rate specification for node v. □

If $R_u = (1, y)$ is a valid execution rate for node u in Figure 3, then $R_u = (2, 2y)$ is also a valid execution rate since node u will execute twice in each time interval $[0 + (k-1)2y,\ 0 + k2y)$ for all $k > 0$. In fact, as shown by Corollary 5.5, there are an infinite number of valid execution rates for node u.

COROLLARY 5.5. *If* $R_v = (x, y)$ *is a valid rate specification for node* v, *then for all positive integers* m, $m \cdot R_v = (m \cdot x, m \cdot y)$ *is also a valid rate specification for node* v.

Proof. If $R_v = (x, y)$ is a valid rate specification for node

v, then by Definition 5.3, there exists a time t such that node v executes exactly x times in time intervals $[t + (k-1)y,\ t + ky)$ for all $k > 0$. Thus in each time interval $[t + m(k-1)y,\ t + mky)$ for all $k > 0$, node v will execute exactly mx times, and $m \cdot R_v = (m \cdot x, m \cdot y)$ is also a valid rate specification for node v. □

Although there exists an infinite number of valid execution rates for a node, not every execution rate is valid. For example, let the execution rate $R_u = (1, y)$ of node u in Figure 3 be valid. By looking at the executions of node v in the snapshot sequence, it would appear that node v executes with a rate of $R_v = (4, 4y)$. Even though node v does execute four times in the interval $[0, 4y)$, the rate specification $R_v = (4, 4y)$ is not valid because this is the only interval of length $4y$ in which node v executes exactly four times. Node v actually executes at a rate of $R_v = (4, 3y)$ starting at time y. To see this, we need to simulate more executions of nodes u and v. Consider the extended snapshot sequence in Figure 3. This snapshot sequence shows that node v executes four times in the interval $[y, 4y)$, four times in the interval $[4y, 7y)$, and four times in the interval $[7y, 10y)$.

The execution rate of node v in Figure 3 was derived by simulating executions of nodes u and v and 'guessing' a valid execution rate. Alternatively, Theorem 5.6 can be used to analytically compute the execution rate of node v using the execution rate of node u and the dataflow attributes of queue q.

THEOREM 5.6. *Let $u \rightsquigarrow v$ be a PGM chain with $\psi(q) = (u, v)$, and let $R_u = (x_u, y_u)$ be a valid execution rate for node u. Under the strong synchrony hypothesis, the execution rate $R_v = (x_v, y_v)$, where*

$$x_v = \frac{prd(q)}{\gcd(prd(q) \cdot x_u, cns(q))} \cdot x_u \tag{4}$$

and

$$y_v = \frac{cns(q)}{\gcd(prd(q) \cdot x_u, cns(q))} \cdot y_u, \tag{5}$$

is a valid execution rate for node v.

Proof. By Definition 5.3, because R_u is valid, there exists a time t_u such that node u executes exactly x_u times in each interval $[t_u + (k-1)y_u, t_u + ky_u)$ where $k > 0$. Let interval j be the first interval $[t_u + (j-1)y_u, t_u + jy_u)$ in which node v executes, and let $t_v = t_u + jy_u$.

In the remainder of the proof, we show that $R_v = (x_v, y_v)$ is a valid rate specification by showing that node v executes exactly x_v times in time intervals $[t_v + (k-1)y_v, t_v + ky_v)$ for all $k > 0$ where x_v and y_v are as defined by Equations (4) and (5). Under the strong synchrony hypothesis, node v executes instantaneously whenever its input queue is over threshold. Let $length(q) = n$ at time t_v. Thus, by Theorem 5.1, n is bounded such that

$$thr(q) - cns(q) \leq MinTokens(q) \leq n \leq MaxUnderThr(q) < thr(q).$$

By Definition 5.3, node u executes exactly x_u times in intervals $[t_u + (k-1)y_u, t_u + ky_u)$ for all $k > 0$. Thus, by Corollary 5.5 and because $t_v = t_u + jy_u$ and y_v is a multiple of y_u, node u executes $(y_v/y_u) \cdot x_u$ times in every time interval $[t_v + (k-1)y_v, t_v + ky_v)$ for all $k > 0$. Since node u produces $prd(q)$ tokens each time it executes, it enqueues a total of

$$\begin{aligned} &prd(q) \cdot \frac{y_v}{y_u} \cdot x_u \\ &\quad = prd(q) \cdot \frac{cns(q) \cdot y_u/(\gcd(prd(q) \cdot x_u, cns(q)))}{y_u} \cdot x_u \\ &\quad = prd(q) \cdot \frac{cns(q)}{\gcd(prd(q) \cdot x_u, cns(q))} \cdot x_u \end{aligned}$$

tokens on queue q in an interval of length y_v. Since each execution of node v consumes $cns(q)$ tokens, x_v executions of node v in an interval of length y_v will consume $(x_v \cdot cns(q))$ tokens. Thus, if queue q contains n tokens at the beginning of the interval, it will contain

$$\begin{aligned} &n + \left(prd(q) \cdot \frac{cns(q)}{\gcd(prd(q) \cdot x_u, cns(q))} \cdot x_u\right) - (x_v \cdot cns(q)) \\ &\quad = n + \left(prd(q) \cdot \frac{x_u \cdot cns(q)}{\gcd(prd(q) \cdot x_u, cns(q))}\right) \\ &\qquad - \left(\frac{x_u \cdot prd(q)}{\gcd(prd(q) \cdot x_u, cns(q))} \cdot cns(q)\right) \\ &\quad = n + \left(\frac{x_u \cdot cns(q) \cdot prd(q)}{\gcd(prd(q) \cdot x_u, cns(q))}\right) \\ &\qquad - \left(\frac{x_u \cdot cns(q) \cdot prd(q)}{\gcd(prd(q) \cdot x_u, cns(q))}\right) = n \end{aligned}$$

tokens at the end of the interval. Furthermore, no more than x_v executions could have occurred since the x_vth execution leaves exactly $n < thr(q)$ tokens on q. Any fewer executions would have left $n \geq thr(q)$ tokens on q, and another execution of node v would have occurred. Therefore, node v executes exactly x_v times in time intervals $[t_v + (k-1)y_v, t_v + ky_v)$ for all $k > 0$ where x_v and y_v are as defined by Equations (4) and (5). □

As required, the proof of Theorem 5.6 only proves that Equations (4) and (5) can be used to compute a valid rate specification for the consumer node v, and there are infinitely many other valid execution rate specifications for node v, as shown by Corollary 5.5.

We now consider the case where the specification of node u is well-defined. In this case, the execution rate of node v is also well-defined when it is computed using Equations (4) and (5).

THEOREM 5.7. *Let $u \rightsquigarrow v$ be a PGM chain with $\psi(q) = (u, v)$, and let $R_u = (x_u, y_u)$ be a well-defined execution rate for node u. Let $R_v = (x_v, y_v)$ be computed using Equations* (4) *and* (5)*. Under the strong synchrony hypothesis, the execution rate $R_v = (x_v, y_v)$ is a well-defined execution rate for node v.*

Proof. This proof follows, for the most part, the proof of Theorem 5.6. See [14]. □

Many non-trivial high assurance systems, such as the SAR system, are described using a chain, and Theorem 5.6 can be used to compute the execution rate of every node in the system. If the execution rate of the source node is well-defined, Theorem 5.7 can be used to compute well-defined execution rates for all other nodes in the chain. For example, let $R_{YRange} = (1,\ 3.6\text{ ms})$ be a well-defined rate specification for source node *YRange* beginning at time 0 with the first execution of source node *YRange* occurring at time 0. That is, source node *YRange* executes at times $k \cdot 3.6$ ms for all $k \geq 0$. Theorem 5.7 is used to compute well-defined rate specifications for the other nodes in the SAR graph as follows.

$$R_{ZeroFill} = (x_{ZeroFill}, y_{ZeroFill})$$

where

$$x_{ZeroFill} = \frac{prd(Range) \cdot x_{YRange}}{\gcd(prd(Range) \cdot x_{YRange}, cns(Range))},$$
$$y_{ZeroFill} = \frac{cns(Range) \cdot y_{YRange}}{\gcd(prd(Range) \cdot x_{YRange}, cns(Range))}.$$

Thus,

$$\begin{aligned} R_{ZeroFill} &= \left(\frac{118 \cdot 1}{\gcd(118 \cdot 1,\ 118)}, \frac{118 \cdot 3.6\text{ ms}}{\gcd(118 \cdot 1,\ 118)}\right) \\ &= \left(\frac{118}{118}, \frac{118 \cdot 3.6\text{ ms}}{118}\right) \\ &= (1,\ 3.6\text{ ms}), \end{aligned}$$

$$\begin{aligned} R_{WindowData} &= R_{RangeFFT} = R_{RCSMult} \\ &= \left(\frac{256 \cdot 1}{\gcd(256 \cdot 1,\ 256)}, \frac{256 \cdot 3.6\text{ ms}}{\gcd(256 \cdot 1,\ 256)}\right) \\ &= \left(\frac{256}{256}, \frac{256 \cdot 3.6\text{ ms}}{256}\right) = (1,\ 3.6\text{ ms}), \end{aligned}$$

$$\begin{aligned} &R_{CornerTurn} \\ &= \left(\frac{256 \cdot 1}{\gcd(256 \cdot 1,\ 256 \cdot 64)}, \frac{(256 \cdot 64) \cdot 3.6\text{ ms}}{\gcd(256 \cdot 1,\ 256 \cdot 64)}\right) \\ &= \left(\frac{256}{256}, \frac{(256 \cdot 64) \cdot 3.6\text{ ms}}{256}\right) = (1,\ 64 \cdot 3.6\text{ ms}) \\ &= (1,\ 230.4\text{ ms}), \end{aligned}$$

$$\begin{aligned} &R_{AzimuthFFT} \\ &= \left(\frac{(256 \cdot 128) \cdot 1}{\gcd((256 \cdot 128) \cdot 1,\ 128)}, \frac{128 \cdot 64 \cdot 3.6\text{ ms}}{\gcd((256 \cdot 128) \cdot 1,\ 128)}\right) \\ &= \left(\frac{256 \cdot 128}{128}, \frac{128 \cdot 64 \cdot 3.6\text{ ms}}{128}\right) = (256,\ 64 \cdot 3.6\text{ ms}) \\ &= (256,\ 230.4\text{ ms}), \end{aligned}$$

and

$$\begin{aligned} R_{KernelMult} &= R_{AzimuthIFFT} = R_{Output} \\ &= \left(\frac{128 \cdot 256}{\gcd(128 \cdot 256, 128)}, \frac{128 \cdot 64 \cdot 3.6\text{ ms}}{\gcd(128 \cdot 256, 128)}\right) \\ &= (256, 230.4ms). \end{aligned}$$

5.3. RBE task model

Moving from the strong synchrony hypothesis to an actual implementation, we need to implement the graph as one or more tasks on a single processor. A scheduling algorithm and a schedulability test that will analytically determine whether or not a graph will meet its temporal requirements are also necessary. Since node execution is neither periodic nor sporadic, even when the source is periodic, the RBE [16] is used to model the execution of the processing graph chain. The advantage of modeling graph execution with the RBE model is that it supports the simple implementation of representing each node as a task that is released when the input queue goes over threshold. Indeed, this is how the SAR benchmark that we evaluated was implemented.

RBE is a general task model consisting of a collection of independent processes specified by four parameters: (x, y, d, e). The pair (x, y) represents the execution rate of a RBE task where x is the number of executions expected to be requested in an interval of length y. Parameter d is a response time parameter that specifies the maximum desired time between the release of a task instance and the completion of its execution (i.e. d is the relative deadline). The parameter e is the maximum amount of processor time required for one execution of the task.

A RBE task set is schedulable if there exists a schedule such that the jth release of task T_i at time $t_{i,j}$ is guaranteed to complete execution by time $D_i(j)$, where

$$D_i(j) = \begin{cases} t_{i,j} + d_i & \text{if } 1 \leq j \leq x_i \\ \max(t_{i,j} + d_i,\ D_i(j - x_i) + y_i) & \text{if } j > x_i. \end{cases} \tag{6}$$

The RBE task model makes no assumptions regarding when a task will be released, however Equation (6) ensures that no more than x_i deadlines come due in an interval of length y_i, even when more than x_i releases of T_i occur in an interval of length y_i. Hence, the deadline assignment function prevents jitter from creating more process demand in an interval by a task than that which is specified by the rate parameters.

The schedulability of a RBE task set under preemptive EDF scheduling can be checked with Theorem 5.8 [16]. Schedulability conditions for non-preemptive EDF scheduling are also presented in [16]. Note that if the cumulative processor utilization for a graph is strictly less than one (i.e. $\sum_{i=1}^{n}(x_i \cdot e_i)/y_i < 1$) then condition (7) can be evaluated efficiently (in pseudo-polynomial time) using techniques developed in [17].

THEOREM 5.8. *Let* $\mathcal{T} = \{(x_1, y_1, d_1, e_1), \ldots, (x_n, y_n, d_n, e_n)\}$ *be a set of RBE tasks.* $\mathcal{T}$ *will be feasible if and*

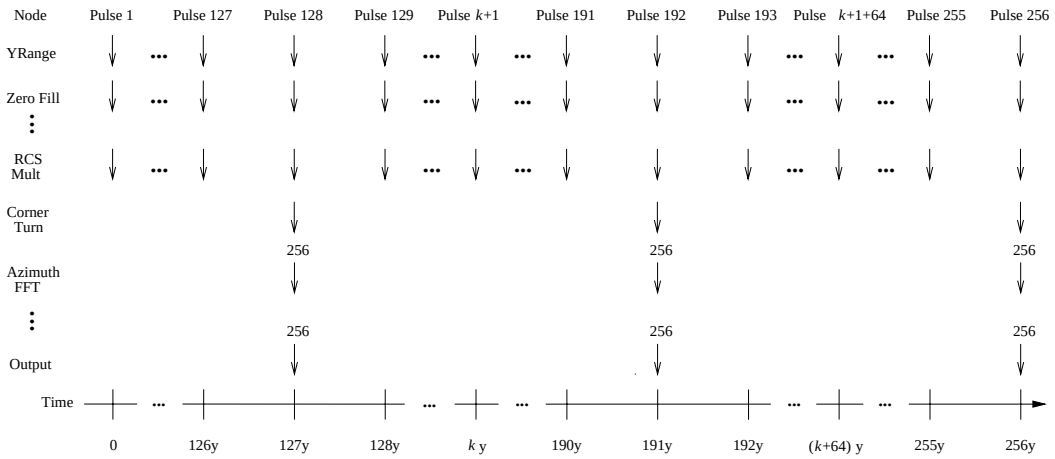


FIGURE 4. A simulation showing latency for the SAR graph under the strong synchrony hypothesis. Each down arrow represents the release and instantaneous execution of a node, and the number 256 above a down arrow means 256 instantaneous executions of a node.

only if

$$\forall\, L > 0,\ L \geq \sum_{i=1}^{n} f\left(\frac{L - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i \qquad (7)$$

where

$$f(a) = \begin{cases} \lfloor a \rfloor & \text{if } a \geq 0 \\ 0 & \text{if } a < 0. \end{cases}$$

For a PGM graph, Equation (7) becomes a sufficient condition (but not necessary) for preemptive EDF scheduling as long as nodes execute only when their input queues are over threshold (i.e. the tasks are released when the node's input queue is over threshold—thereby ensuring precedence constraints are met). Equation (7) is not a necessary condition for PGM graphs since it assumes that all x_i releases of a node may occur at the beginning of an interval of length y_i. For some nodes, such as v in Figure 2, this is not possible.

6. MANAGING LATENCY

Latency can be defined many different ways. An appealing definition is the delay between a start event and a corresponding stop event. In graph models that require unity dataflow attributes, the start event may be the arrival of a token from the source, and the stop event can be identified as the enqueuing of a token on the graph's output queue. But it is difficult to apply this definition to PGM graphs. As the SAR graph demonstrates, nodes may add tokens to the data stream. Nodes may also reduce the number of tokens in the data stream (known as data decimation), or the node may delay some number of tokens and use the delayed tokens in both the current and the subsequent execution as the *Corner Turn* node does in the SAR graph.

A signal-processing engineer describes latency as the time delay between the sampling of a signal and the presentation of the processed signal to the output device (which may be a screen, speaker or another computer). We use this definition with a clarification. Since we can only measure time in units of the period of the source, we consider the $prd(q)$ tokens delivered each period by an external device to be 'one sample'; each pulse in the SAR graph constitutes one sample, which consists of 118 tokens. Hence, under the strong synchrony hypothesis, latency is the delay between the enqueuing of $prd(q)$ tokens onto queue q by an external source and the next enqueuing of $prd(q')$ tokens on queue q' attached to an external output device.

Using the strong synchrony hypothesis, the next section demonstrates that there exist multiple inherent latency values for a graph. In Section 6.2 these inherent latency values are added to the latency imposed by the scheduling algorithm and node execution to bound the total latency any signal encounters. Finally, we analyze the effect of deadlines on latency in Section 6.3.

6.1. Latency with the strong synchrony hypothesis

There is a pattern of executions that result in various latency values for the input signal. Consider the execution of the SAR graph shown in Figure 4. In this example, we assume the strong synchrony hypothesis and each down arrow represents the release and instantaneous execution of a node. The minimum latency for a sample is zero, which is the case for the 128th pulse received by the SAR graph. As shown in Figure 4, the 128th pulse arrives at time $127y$ and results in the execution of every node in the graph. Pulses 192, 256, 320, 384, ... all have a latency of 0. The maximum latency value, encountered by the first pulse, is $127y$. The first signal received by the graph always encounters the maximum latency (assuming the queues have no initial data). There is, however, another 'maximum' latency that is of more interest, and that is the maximum latency that occurs after the first execution of every node in the graph. In the execution example shown in Figure 4 for the SAR graph, this maximum latency is encountered by

pulses 129, 193, 257, 321, ..., which have a latency of $63y$. Notice that there are 126 other unique latency values for this simple graph (e.g. the latency for pulse $j+1$ is $(127-j)y$).

The latency which a sample encounters under the strong synchrony hypothesis is dependent on the dataflow attributes of the graph, the state of the queues (i.e. the number of tokens on each queue of the graph) when the sample arrives, and the execution rate of the graph source node. Lemma 6.1 states analytically what these relationships are, and, at any point in time, it also tells us the number of samples that must be produced by node u before node w is eligible for execution. This number is used by Theorem 6.2 to compute a lower bound on the latency a sample will encounter when the source is periodic.

LEMMA 6.1. *Let path $u \rightsquigarrow w$ be a PGM chain, and let*

$$F_{u\rightsquigarrow w} = \begin{cases} \max\left(0, \left\lceil \dfrac{thr(q) - length(q)}{prd(q)} \right\rceil\right) \\ \quad if\ \exists q : \psi(q) = (u, w) \\ \max\left(0, \left\lceil \dfrac{(F_{v\rightsquigarrow w}-1)\cdot cns(q)+thr(q)-length(q)}{prd(q)} \right\rceil\right) \\ \quad if\ \exists q : \psi(q) = (u, v) \wedge v \neq w \ \wedge \ F_{v\rightsquigarrow w} > 0 \\ 0 \quad if\ \exists q : \psi(q) = (u, v) \wedge v \neq w \ \wedge \ F_{v\rightsquigarrow w} = 0. \end{cases} \tag{8}$$

Node u must execute $F_{u\rightsquigarrow w}$ times to produce enough tokens in order to put the input queue to node w over threshold.

Proof. See [14]. □

Equation (8) defines a recursive function that determines the number of times node u must execute before the input queue to node w is over threshold. The first branch of the function handles a path of length one where node u is attached to node w. For example, consider the chain *Azimuth IFFT* $\rightsquigarrow$ *Output* in the SAR graph of Figure 1 whose length is one. Assuming $length(Image) = 0$, node *Azimuth IFFT* must execute

$$F_{AzimuthIFFT\rightsquigarrow Output} = \max\left(0, \left\lceil \frac{thr(Image) - length(Image)}{prd(Image)} \right\rceil\right) = \left\lceil \frac{128-0}{128} \right\rceil = 1$$

times before node *Output* executes. The second branch of the function $F_{u\rightsquigarrow w}$ recursively references itself when applied to a path whose length is reduced by one (until the path is of length one). Thus, by recursively invoking $F_{u\rightsquigarrow w}$, the second branch returns the number of times the current source node u must execute in order for the node attached to it, node v, to execute $F_{v\rightsquigarrow w}$ times (which is the number of times node v must execute in order to put the input queue to node w over threshold). For example, let $length(RCS) = 256 \times 100$ and $length(q) = 0$ for the rest of the queues in the graph. Node *RCS Mult* must execute $F_{RCSMult\rightsquigarrow Output}$ times before node *Output* executes. Since $F_{CornerTurn\rightsquigarrow Output} = 1$, the total number of times node *RCS Mult* must execute is

$$\left\lceil \frac{(1-1)\cdot cns(RCS) + thr(RCS) - length(RCS)}{prd(RCS)} \right\rceil = \left\lceil \frac{(1-1)\cdot(256\cdot 64) + (256\cdot 128) - (256\cdot 100)}{256} \right\rceil = \left\lceil \frac{(256\cdot 28)}{256} \right\rceil = 28.$$

The third branch of the function $F_{u\rightsquigarrow w}$ returns zero when the input queue to node w is already over threshold, or when other queues in the chain have enough data that the input queue to node w will go over threshold without node u executing again.

Let $u \rightsquigarrow w$ be a PGM chain such that node u is a periodic source node with period y_u and node w is an output node. Evaluating $F_{u\rightsquigarrow w}$ just before a sample arrives will tell us how many more samples are required before the input queue to node w is over threshold. Thus, the latency a sample encounters under the strong synchrony hypothesis is $\max(0, (F_{u\rightsquigarrow w} - 1)\cdot y_u)$. We subtract one from $F_{u\rightsquigarrow w}$ before converting it to time units since the latency interval begins after the sample arrives.

THEOREM 6.2. *Let $u \rightsquigarrow w$ be a PGM chain such that u is a periodic source node with period y_u and w is an output node. Under the strong synchrony hypothesis, the latency a sample encounters is*

$$\max(0, (F_{u\rightsquigarrow w} - 1)\cdot y_u). \tag{9}$$

Proof. By Lemma 6.1, $F_{u\rightsquigarrow w}$ executions of source node u are required before output node w is eligible for execution. If $F_{u\rightsquigarrow w} = 0$, the sample's latency is 0 and Equation (9) returns 0 as desired. If $F_{u\rightsquigarrow w} = 1$, the next sample will encounter an latency of 0 since output node w will execute as soon as the sample arrives. In this case $(F_{u\rightsquigarrow w} - 1)\cdot y_u = 0$ as desired. If $F_{u\rightsquigarrow w} > 1$, the next sample will encounter a latency of $(F_{u\rightsquigarrow w} - 1)\cdot y_u$ time units since $(F_{u\rightsquigarrow w} - 1)$ additional executions of source node u are required after the sample arrives before output node w executes. Therefore, under the strong synchrony hypothesis, if source node u has a period of y_u, a sample's latency will be $(F_{u\rightsquigarrow w} - 1)\cdot y_u$ time units. □

The latency defined by Theorem 6.2 provides a lower bound on the latency any sample will encounter—even on an infinitely fast machine. We call this latency *inherent latency* because it is inherently defined by the dataflow attributes of the graph. Using Theorem 6.2 we find, as expected, that the inherent latency of the first pulse received by the SAR graph is $127y$. Recall from Theorem 5.1 that the most tokens queue q can hold without being over threshold is $MaxUnderThr(q)$ and the minimum possible number of tokens on queue q after its sink node has fired once is $MinTokens(q)$. When all of the queues in the graph contain $MaxUnderThr(q)$ tokens, as is the case just before pulses 192, 256, 320, ...,

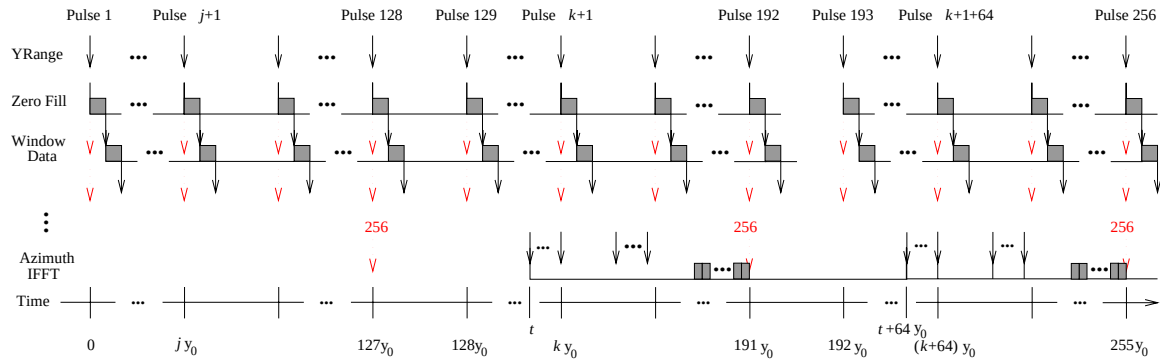


FIGURE 5. Latency for the SAR graph. A light arrow represents a node's release under the strong synchrony hypothesis. A dark arrow represents the actual release time, and the node's execution is represented by a box.

the next sample's latency will be 0—just as Figure 4 shows. Evaluating Equation (9) when each queue in the SAR graph contains $MinTokens(q)$ tokens, we get a sample latency of $63y$—just as Figure 4 shows for pulses 129 and 193.

6.2. Latency in an implementation

Moving from the strong synchrony hypothesis to an actual implementation, we assume each node in the graph is implemented as a RBE task that is released whenever its input queue is over threshold. To simplify the presentation of managing latency and buffer requirements in an implementation, we make a change in notation. For the rest of the paper, we assume nodes in a chain are numbered sequentially such that the first node is labeled N_0 and the last node is labeled N_{n+1}. Queues are numbered sequentially such that the output queue of node N_0 is labeled Q_0 and the input queue to node N_{n+1} is labeled Q_n. Node N_0 represents an external input device and node N_{n+1} represents an external output device. Neither nodes N_0 nor N_{n+1} require CPU time. Without loss of generality, the chain from node N_0 to node N_n is represented as $0 \rightsquigarrow n$. In Section 7 we assume the output device (node N_{n+1}) consumes data as soon as node N_n produces data, which is the case for the SAR graph. Thus, since the output device requires no CPU time and is assumed to consume data as soon as it is produced, latency will be computed using the chain $0 \rightsquigarrow n$.

Scheduling an implementation of the graph results in an upper and lower bound for each of the latency values identified with the strong synchrony hypothesis. In other words, we get latency intervals rather than precise latency values for a given sample. The lower bound for a sample's latency is a function of the scheduling algorithm and, as shown in Section 6.1, the graph attributes. The lower bound for the latency interval is the latency value derived using Equation (9) plus the sum of the execution times for the nodes in the chain. That is, a sample's latency must be greater than or equal to $(F_{0 \rightsquigarrow n} - 1) \cdot y_0 + \sum_{i=1}^{n} e_i$.

The upper bound for a sample's latency is dependent on the scheduling algorithm, dataflow attributes and deadline values. Generally, the deadline parameters are the only free variables in the function. To determine a sample's latency in an implementation of the graph, we need to provide a value for each d_i in the RBE task set. Realizing that d_i affects latency, what should it be? How does d_i affect latency?

We start by observing that if $\forall\, i : 1 \leq i \leq n, d_i = y_i$ and the graph is not schedulable (i.e. Condition (7) returns a negative result) then the processor is overloaded since Condition (7) reduces to the Liu and Layland feasibility test [18] and we get $1 < \sum_{i=1}^{n} (x_i \cdot e_i)/y_i$. We also observe that increasing $d_i > y_i$ will not improve latency and, as we will show later, increases buffer requirements. Hence, we will set $d_i = y_i$ and see how this affects the upper bound for latency values.

Figure 5 shows an execution of the SAR graph with $d_i = y_i$. In this figure, the light arrows represent the release time for N_i under the strong synchrony hypothesis and the dark arrows represent the actual release time. We see from Figure 5 that task *Zero Fill* is released at times 0, y_0, $2y_0$, $3y_0, \ldots$ and the deadlines corresponding to each release time are y_0, $2y_0$, $3y_0, \ldots$, since $d_1 = y_1 = y_0$. Due to scheduling and execution times, however, the task *Window Data* is not released until times $0 + e_1$, $y_0 + e_1$, $2y_0 + e_1$, $3y_0 + e_1, \ldots$ and the corresponding deadlines are $0+e_1+d_2 = y_0+e_1, 2y_0+e_1, 3y_0+e_1, \ldots$. In this example, the first execution of task *Azimuth IFFT* is released at time t, which is after $128y_0$. Its deadline is $t + 64y_0$, which is after $192y_0$. Also note that the 256th execution of task *Azimuth IFFT* completes execution by time $191y_0$—well before its deadline.

The release times shown in Figure 5 for the tasks *Zero Fill* and *Window Data* are the earliest possible release times. As we have noted, the task *Azimuth IFFT* completes its 256th execution by time $191y_0$ even though the deadline for the first release of *Azimuth IFFT* is not until $t + 64y_0$. This was no accident. All of the first 256 executions of *Azimuth IFFT* will be released and complete execution between $127y_0$ and $191y_0$. To see this, we must look at the earliest possible release time for the first execution of *Azimuth IFFT* and the schedulability Condition (7). From Theorem 6.2, we know that the first release of task *Azimuth IFFT* cannot occur before $127y_0$. An affirmative result from Condition (7) means that there exists enough processor capacity for nodes

N_1 through N_k, $1 \leq i \leq k \leq n$, to execute $(y_k/y_i) \cdot x_i$ times during an interval of length y_k. This means that 64 executions of *Zero Fill*, *Window Data*, *Range FFT*, and *RCS Mult*, one execution of *Corner Turn* and 256 executions of *Azimuth FFT*, *Kernel Mult* and *Azimuth IFFT* will all complete execution within $64y_0$ time units even when they are all released at the same instant (i.e. when *Zero Fill* is first released). We will exploit this fact, similarly to the way Jeffay did in [19], to bound a sample's latency.

We can use the release point derived with the strong synchrony hypothesis and add d_i to get the time at which N_i will have completed execution—even if this time is less than the actual release time plus d_i. We call this *release-time inheritance*. When the deadline for each node is greater than or equal to its predecessor's relative deadline, release-time inheritance can be used to minimize latency. Under release time-inheritance, node N_i is assigned a logical release time (at the time of its actual release) that is equal to the logical release time of the node N_{i-1}. Deadline assignment function (6) then uses the logical release times rather than the actual release times to assign deadlines for the completion of node execution. Theorem 6.3 uses release-time inheritance to provide a lower and upper bound for any sample's latency.

THEOREM 6.3. *Given $R_0 = (1, y_0)$ and a schedulable graph in which $\forall\, i : 1 \leq 1 < n :: d_i \leq d_{i+1}$, a sample's latency under EDF scheduling with release-time inheritance and deadline-assignment function* (6) *is bounded such that*

$$(F_{0\rightsquigarrow n} - 1) \cdot y_0 + \sum_{i=1}^{n} e_i \leq \textit{Sample Latency} \leq (F_{0\rightsquigarrow n} - 1) \cdot y_0 + d_n$$

where $F_{0\rightsquigarrow n}$ is evaluated just before the sample's arrival.

Proof. By Theorem 6.2, a sample's latency in an implementation of the graph cannot be less than $(F_{0\rightsquigarrow n} - 1) \cdot y_0$ since this is the latency on an infinitely fast machine. The minimum latency for a signal occurs when each node in the chain must only execute once after node N_0 delivers the additional $(F_{0\rightsquigarrow n} - 1) \cdot prd(Q_0)$ tokens. Therefore the sample's latency must be greater than or equal to $(F_{0\rightsquigarrow n} - 1) \cdot y_0 + \sum_{i=1}^{n} e_i$. Let $sample_k$ be the sample for which we are bounding latency. When the $(F_{0\rightsquigarrow n} - 1)$th sample after $sample_k$ arrives, every node in the graph will fire at least once before node N_n produces data. Using release time inheritance, every task released (either directly or indirectly) from this last sample will have a deadline less than or equal to d_n time units from the arrival of the last signal. Therefore if the graph is schedulable, node N_n will execute within $(F_{0\rightsquigarrow n} - 1) \cdot y_0 + d_n$ time units of the arrival of $sample_k$. Hence, a sample's latency is bounded such that

$$(F_{0\rightsquigarrow n} - 1) \cdot y_0 + \sum_{i=1}^{n} e_i \leq \textit{Sample Latency} \leq (F_{0\rightsquigarrow n} - 1) \cdot y_0 + d_n$$

where $F_{0\rightsquigarrow n}$ is evaluated just before the sample's arrival, and Theorem 6.3 holds. □

As long as the scheduler ensures that a task only executes when its input queue is over threshold, it does not matter if node N_{i+1} executes before node N_i. When the RBE task set is specified such that $d_i \leq d_{i+1}$, a release of node N_{i+1} will never be assigned a deadline earlier than a release of node N_i, even when logical release times are used. Moreover, the latency bound of Theorem 6.3 holds even when a release of node N_{i+1} executes before a release of node N_i, which may occur when both are assigned the same deadline. The EDF scheduling algorithm does not specify how to break ties. Hence, a variant of EDF may break ties based on topological sorting rather than actual release times, which may result in node N_{i+1} executing before node N_i when $d_i = d_{i+1}$. Although latency is not affected by the tie breaking algorithm, buffer bounds are. We address this issue in Section 7.

6.3. Reducing latency further

If the latency bounds derived using $d_i = y_i$ do not meet the application's latency requirements, we can evaluate the latency with smaller deadlines. As long as we keep $d_i \leq d_{i+1}$, Theorem 6.3 can be used to evaluate new latency bounds. A simple technique to reduce the maximum latency any signal will encounter (for a graph executing on a uniprocessor) is to iteratively decrease the maximum deadline(s) to the maximum y_i such that $y_i < \max\{d_j\}$ in the graph. For example, after a positive result from Condition (7) with $d_n = y_n$, we would set $d_n = y_{n-1}$, assuming $y_{n-1} < y_n$, otherwise we would set $d_{n-1} = d_n = y_{n-2}$. When Condition (7) finally returns a negative result we have found a 'breaking point'. We can either use the deadlines from the previous iteration or find *the* 'breaking point' (for this technique), which lies between the deadline values used in the last two iterations.

7. MANAGING BUFFER REQUIREMENTS

This section shows how to bound and manage the buffer requirements of chains executed under the RBE model with release time inheritance. Using logical release times rather than actual release times, creates deadline ties during execution. These ties can then be broken based on topology to reduce the buffer requirements from what they would be if the ties were broken arbitrarily. To simplify the presentation of the buffer bounds (by shortening equations), another change in notation is made. Let $p_i = prd(Q_i)$, $c_i = cns(Q_i)$, $\tau_i = thr(Q_i)$ and $r_i = MaxUnderThr(Q_i)$.

Since N_{n+1} represents an external device and is not scheduled, we cannot give an upper bound on Q_n. One may assume the device takes data as it is produced and bound the buffer space for Q_n with p_n (i.e. $prd(Q_n)$). Or assuming double buffering techniques (common in I/O interfaces), one might bound the buffer space as $2p_n$. In either case, the bound is platform specific and not considered in the total buffer bounds presented here.

Recall from Theorem 5.1 that $r_i = MaxUnderThr(Q_i)$ is the greatest number of tokens Q_i can hold without being over threshold. After Q_i goes over threshold, the

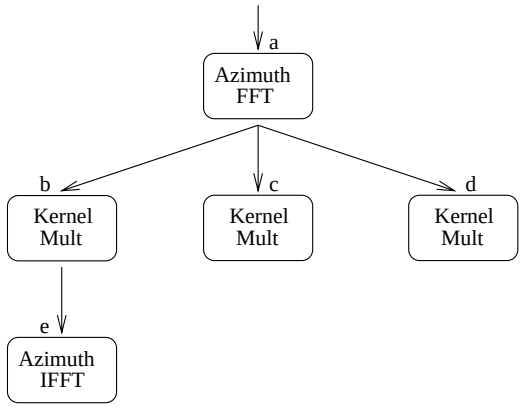


FIGURE 6. A scheduling graph.

number of tokens that can accumulate on the queue is a function of dataflow attributes, deadlines and the scheduling algorithm. Here we assume dataflow attributes are fixed by the signal-processing engineer and use deadline parameters and scheduling algorithms to manage buffer requirements of the application.

We have derived buffer bounds for preemptive EDF scheduling and two variations of EDF: breadth-first EDF (BF-EDF) and depth-first EDF (DF-EDF). The names for these EDF variants become apparent when one looks at a possible scheduling graph, which is used to break deadline ties. A scheduling graph is a topologically-sorted graph of vertices representing releases of RBE tasks with the same deadline. The graph is sorted with respect to the dataflow graph and all jobs in the graph have the same deadline. Consider the scheduling graph in Figure 6—a possible snapshot of the ready queue for the SAR graph after Pulse 128 has been processed by the *Corner Turn* node. The BF-EDF scheduling algorithm performs a breadth-first search of eligible jobs, beginning at the left-most side of each level. Hence, the BF-EDF algorithm would select the *Azimuth FFT* task followed by the left-most release of *Kernel Mult*. Using the labels *a*, *b*, *c*, *d* and *e* to refer to the tasks releases in Figure 6, BF-EDF would schedule them in order: *a*, *b*, *c*, *d*, d', *e* and e^* where d' represents the new release of *Kernel Mult* caused by the execution of *Azimuth FFT*, and e^* represents the new releases of *Azimuth IFFT* which result from executions of *Kernel Mult*. The DF-EDF scheduling algorithm performs a depth-first search of eligible jobs by traversing down the left-most side of the tree until it reaches a leaf. In this case, DF-EDF would select the *Azimuth IFFT* task to execute followed by the left-most release of *Kernel Mult*. A DF-EDF schedule, starting with the schedule graph of Figure 6, would be *e*, *b*, e', *c*, e'', *d*, e''', *a* where e', e'' and e''' are new releases of *Azimuth IFFT* caused by the executions of *Kernel Mult*.

7.1. Buffer bounds for EDF scheduling

We begin the process of bounding the buffer requirements of a graph by observing that when $d_{i+1} > d_i$, the buffer bound for Q_i is the same for all EDF variant scheduling algorithms that use release-time inheritance to schedule PGM graphs. The case of $d_{i+1} > d_i$ prevents deadline ties from occurring; hence, the tie breaking algorithm has no impact on the buffer bound for Q_i. When $d_{i+1} > d_i$ and $d_{i+1} \geq y_0$, we can bound the buffer requirements of Q_i independently of the number of tokens that may accumulate on Q_{i-1}.

LEMMA 7.1. *For EDF scheduling algorithms with release time inheritance, if $R_0 = (1, y_0)$ and $d_{j+1} > d_j \wedge d_{j+1} \geq y_0$, the maximum buffer space required by Q_i of a schedulable graph is less than or equal to*

$$B_{\mathrm{EDF}}(Q_i) = \begin{cases} \left(\left\lceil \frac{d_1}{y_0} \right\rceil \cdot p_0\right) + r_0 & \text{if } i = 0 \\ \left(\left\lceil \frac{d_{i+1}}{y_i} \right\rceil \cdot x_i \cdot p_i\right) + r_i & \\ \quad \text{if } (i > 0 \wedge ((d_{i+1} > d_i \wedge y_0 \leq d_{i+1} < y_i) & \\ \quad\quad \vee (d_i < y_i \leq d_{i+1}))) & \\ \left(\left\lfloor \frac{d_{i+1}}{y_i} \right\rfloor \cdot x_i \cdot p_i\right) + r_i & \\ \quad \text{if } (i > 0 \wedge y_i \leq d_i < d_{i+1}). & \end{cases}$$

Proof. See [14]. □

The EDF scheduling algorithm does not specify how deadline ties are broken, and the buffer requirements of a queue are greatest when breadth-first scheduling is used to break deadline ties between two eligible nodes. Thus, to bound a queue's buffer requirements, we must assume that whenever a deadline tie is possible it may be broken by performing a breadth-first search of the scheduling graph.

LEMMA 7.2. *For EDF scheduling algorithms with release time inheritance that may break ties using a breadth-first search of the scheduling graph, if $R_0 = (1, y_0)$, the maximum buffer space required by Q_i, $\forall\ i \geq 0$, of a schedulable graph is less than or equal to*

$$B_{\mathrm{BF}}(Q_i) = \begin{cases} \left(\left\lceil \frac{d_1}{y_0} \right\rceil \cdot p_0\right) + r_0 & \text{if } i = 0 \\ \left(\left\lceil \frac{d_{i+1}}{y_i} \right\rceil \cdot x_i \cdot p_i\right) + r_i & \\ \quad \text{if } (i > 0 \wedge d_{i+1} > d_i \wedge y_0 \leq d_{i+1} < y_i) & \\ \quad\quad \vee\ (i > 0 \wedge d_i < y_i \leq d_{i+1}) & \\ \left(\left\lfloor \frac{d_{i+1}}{y_i} \right\rfloor \cdot x_i \cdot p_i\right) + r_i & \\ \quad \text{if } (i > 0 \wedge y_i \leq d_i < d_{i+1}) & \\ \left(\left\lfloor \frac{B_{\mathrm{BF}}(Q_{i-1}) - \tau_{i-1}}{c_{i-1}} \right\rfloor + 1\right) \cdot p_i + r_i & \text{otherwise.} \end{cases} \tag{10}$$

Proof. See [14]. □

Since EDF does not specify how ties are broken, we need to sum $B_{\mathrm{BF}}(Q_i)$ over all of the queues in the chain to bound a graph's simultaneous buffer requirements.

THEOREM 7.3. *For EDF scheduling with release time inheritance, if $R_0 = (1, y_0)$ and $d_{i+1} \geq d_i$, $\forall\, i : 0 \leq i < n$, the maximum buffer space required is less than or equal to $\sum_{i=0}^{n-1} B_{\mathrm{BF}}(Q_i)$.*

Proof. From Lemma 7.2, the maximum space Q_i will require is $B_{\mathrm{BF}}(Q_i)$. Since EDF does not specify how ties are broken, we need to sum $B_{\mathrm{BF}}(Q_i)$ over all of the queues in the chain to bound a graph's simultaneous buffer requirements. Therefore, the maximum buffer space required is less than or equal to $\sum_{i=0}^{n-1} B_{\mathrm{BF}}(Q_i)$. □

If deadline ties are broken in a deterministic manner specified by the deadline driven scheduling algorithm, we can get a much tighter bound on buffer requirements.

7.2. Buffer bounds for BF-EDF scheduling

The BF-EDF scheduling algorithm is an EDF algorithm in which deadline ties are broken by performing a breadth-first search of the scheduling graph. The function $B_{\mathrm{BF}}(Q_i)$, Equation (10), returns the maximum number of tokens the ith queue will ever hold when deadline ties *may* be broken with a breadth-first search of the scheduling graph. The BF-EDF algorithm *always* breaks ties with a breadth-first search of the scheduling graph, so we can use $B_{\mathrm{BF}}(Q_i)$ to bound the memory needs of Q_i when a schedulable graph is executed under BF-EDF scheduling. When $d_i < d_{i+1}$, $\forall\, i > 0$, no deadline tie is possible and Theorem 7.3 bounds the graph's buffer requirements for BF-EDF scheduling as well as EDF scheduling. When there exist consecutive nodes in the chain with the same deadline, however, we can reduce the buffer bounds for the graph.

Consider the case when $d_i = d_{i+1}$, $\forall\, i > 0$. Since EDF does not specify how ties are broken, we had to sum $B_{\mathrm{BF}}(Q_i)$ over all of the queues in the chain to bound a graph's simultaneous buffer requirements. With BF-EDF, however, we know that, $\forall\, j > i > 1$, any release of N_i will execute before a release of N_j when N_i and N_j both have the same deadline. When N_i executes, it reads data from Q_{i-1} and writes data to Q_i—using both queues simultaneously. By the time N_{i+1} executes, however, Q_{i-i} will be under threshold and will hold at most r_{i-1} tokens. Much of the space that was used by Q_{i-1} when N_i was executing can be reclaimed and used by Q_{i+1} to hold the data produced by N_{i+1}. Therefore the total buffer space required for Q_{i-1} and Q_{i+1} is

$$\max\{B_{\mathrm{BF}}(Q_{i-1}) - r_{i-1}, B_{\mathrm{BF}}(Q_{i+1}) - r_{i+1}\} + r_{i-1} + r_{i+1}.$$

Theorem 7.4 divides the queues into two disjoint sets and uses this technique to bound the total buffer space required by the chain when all of the nodes have the same deadline values.

THEOREM 7.4. *For the BF-EDF scheduling algorithm with release time inheritance, if $R_0 = (1, y_0)$ and $d_{i+1} = d_i$, $\forall\, i : 0 < i < n$, the maximum buffer space required is $\leq \beta$, where*

$$\begin{aligned}\beta = {} & B_{\mathrm{BF}}(Q_0) \\ & + \max\{B_{\mathrm{BF}}(Q_k) - r_k \mid \forall\, k = 2i : i > 0 \ \wedge\ k < n\} \\ & + \max\{B_{\mathrm{BF}}(Q_k) - r_k \mid \forall\, k = 2i - 1 : i > 0 \ \wedge\ k < n\} \\ & + \sum_{i=1}^{n-1} r_i. \end{aligned} \tag{11}$$

Proof. See [14]. □

Depending on the d_i values, we may be able to use variations of the techniques shown in this section to get tighter buffer bounds for a specific graph than either Theorem 7.3 or Theorem 7.4. We leave open the problem of finding a tight buffer bound for generic chains executed with the BF-EDF scheduling algorithm.

7.3. Buffer bounds for DF-EDF scheduling

The DF-EDF scheduling algorithm is an EDF algorithm in which deadline ties are broken by performing a depth-first search of the scheduling graph. For some applications, breaking deadline ties with a depth-first search of the scheduling graph rather than a breadth-first search results in a lower upper bound on buffer requirements for the graph. The function $B_{\mathrm{DF}}(Q_i)$ returns the maximum number of tokens Q_i will ever hold when the graph is scheduled with release inheritance and DF-EDF. This function is used in Theorem 7.6 to bound the total buffer space required for the graph to execute with DF-EDF scheduling.

LEMMA 7.5. *For the depth-first EDF scheduling algorithm with release time inheritance, if the graph is schedulable, $R_0 = (1, y_0)$, and $d_{i+1} \geq d_i$, $\forall\, i : 0 \leq i < n$, the maximum buffer space required by Q_i is less than or equal to*

$$B_{\mathrm{DF}}(Q_i) = \begin{cases} \left(\left\lceil \dfrac{d_1}{y_0} \right\rceil \cdot p_0\right) + r_0 & \text{if } i = 0 \\ \left(\left\lceil \dfrac{d_{i+1}}{y_i} \right\rceil \cdot x_i \cdot p_i\right) + r_i & \\ \quad \text{if } (i > 0 \ \wedge\ d_{i+1} > d_i \ \wedge\ y_0 \leq d_{i+1} < y_i) & \\ \quad\ \vee\ (i > 0 \ \wedge\ d_i < y_i \leq d_{i+1}) & \\ \left(\left\lfloor \dfrac{d_{i+1}}{y_i} \right\rfloor \cdot x_i \cdot p_i\right) + r_i & \\ \quad \text{if } (i > 0 \ \wedge\ y_i \leq d_i < d_{i+1}) & \\ \left(\left\lfloor \dfrac{B_{\mathrm{DF}}(Q_{i-1}) - \tau_{i-1}}{c_{i-1}} \right\rfloor + 1\right) \cdot p_i + r_i & \\ \quad \text{if } i > 0 \ \wedge\ d_i < d_{i+1} < y_0 & \\ p_i + r_i \quad \text{otherwise.} & \end{cases} \tag{12}$$

Proof. See [14]. □

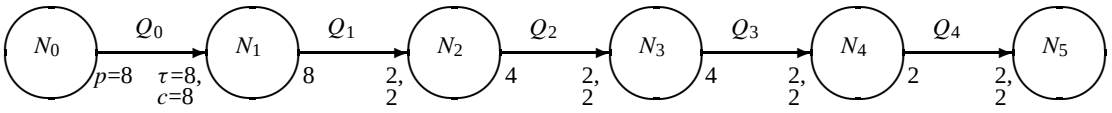


FIGURE 7. Six-node chain scheduled with the DF-EDF algorithm.

TABLE 1. Maximum buffer space required per queue evaluated with $B_{\mathrm{BF}}(Q_i)$ and $B_{\mathrm{DF}}(Q_i)$.

Queue	$B_{\mathrm{BF}}(Q_i)$ $d_i = y_0$	$B_{\mathrm{DF}}(Q_i)$ $d_i = y_0$	$B_{\mathrm{BF}}(Q_i)$ $d_i = y_i$	$B_{\mathrm{DF}}(Q_i)$ $d_i = y_i$
Range	118	118	118	118
Fill	256	256	256	256
Window	256	256	256	256
RFFT	256	256	256	256
RCS	32,768	32,768	48,896	48,896
Azimuth	32,768	32,768	32,768	32,768
AFFT	32,768	128	32,768	128
Mult	32,768	128	32,768	128

THEOREM 7.6. *For DF-EDF scheduling with release time inheritance, if* $R_0 = (1, y_0)$ *and* $d_{i+1} \geq d_i$, $\forall\ i : 0 < i < n$, *the maximum buffer space required is less than or equal to* $\sum_{k=0}^{n-1} B_{\mathrm{DF}}(Q_k)$.

Proof. From Lemma 7.5, the maximum space Q_k will require is $B_{\mathrm{DF}}(Q_k)$. Therefore, the maximum buffer space required is less than or equal to $\sum_{k=0}^{n-1} B_{\mathrm{DF}}(Q_k)$. □

When the graph is scheduled with the BF-EDF algorithm and $d_{i+1} = d_i$, $\forall\ i : 0 < i < n$, Theorem 7.4 provides a tighter bound on the buffer space required by the graph than Theorem 7.3 by reclaiming unused buffer space. Unfortunately we cannot use the same technique when the graph is scheduled with the DF-EDF algorithm to get a tighter bound than Theorem 7.6 provides. To see this, consider the graph of Figure 7 and let $R_0 = (1, y_0)$. Under DF-EDF scheduling, every time N_4 executes, Q_3 will require space for four tokens. At the same time Q_1 will require space for six tokens (not $r_1 = 0$ as it does with BF-EDF scheduling). We leave open the problem of finding a tight buffer bound for generic chains executed with the DF-EDF scheduling algorithm.

7.4. Buffer bounds for the SAR graph

This section develops buffer bounds for the SAR graph (see Figure 1) using different d_i values and scheduling algorithms by applying Theorems 7.3, 7.4 and 7.6. We begin by finding the minimum buffer space required to execute the graph. This occurs when we set $d_i = y_0$, $\forall\ i > 0$, assuming we have a fast enough CPU that the graph is schedulable with these deadline values. Observe that $d_i = y_0$, $\forall\ i > 0$, also minimizes the latency any sample will encounter. Let $R_{YRange} = (1, 3.6\text{ ms})$, as before, and each $d_i = 3.6$ ms.

7.4.1. *EDF and BF-EDF scheduling with* $d_i = y_0$

To bound the graph's buffer needs when it is executed with either canonical EDF or BF-EDF scheduling, we first need to bound the buffer requirements of each queue using $B_{\mathrm{BF}}(Q_i)$, Equation (10). These results are summarized in Table 1.

By Theorem 7.3, the total buffer space required to execute the SAR graph with EDF scheduling when $d_i = y_0 =$ 3.6 ms is less than or equal to $\sum_{i=0}^{n-1} B_{\mathrm{BF}}(Q_i) = 131{,}958$. By Theorem 7.4, if BF-EDF scheduling is used when $d_i = y_0 = 3.6$ ms, the required buffer spaced is less than or equal to β where

$$\begin{aligned}\beta &= B_{\mathrm{BF}}(Q_0) \\ &\quad + \max\{B_{\mathrm{BF}}(Q_k) - r_k \mid \forall\ k = 2i : i > 0 \ \wedge\ k < n\} \\ &\quad + \max\{B_{\mathrm{BF}}(Q_k) - r_k \mid \forall\ k = 2i - 1 : i > 0 \ \wedge\ k < n\} \\ &\quad + \sum_{i=1}^{n-1} r_i = 118 + 32{,}768 + 32{,}768 + 32{,}512 \\ &= 98{,}116.\end{aligned}$$

7.4.2. *DF-EDF scheduling with* $d_i = y_0$

To bound the graph's buffer needs when it is executed with DF-EDF scheduling, we first need to bound the buffer requirements of each queue using $B_{\mathrm{DF}}(Q_i)$, Equation (12). These results are summarized in Table 1. By Theorem 7.6, the total buffer space required to execute the SAR graph with DF-EDF scheduling when $d_i = y_0 = 3.6$ ms is less than or equal to $\sum_{i=0}^{n-1} B_{\mathrm{DF}}(Q_i) = 66{,}678$.

7.4.3. *Scheduling with* $d_i = y_i$

Now consider what happens to the buffer bounds when $\forall\ i > 0 : d_i = y_i$. Table 1 shows the values returned from $B_{\mathrm{BF}}(Q_i)$ and $B_{\mathrm{DF}}(Q_i)$ for each queue in the SAR graph with $d_i = y_i$. Notice that only the queue labeled *RCS* is affected by the new deadline values. This is because the *Corner Turn* node acts as a gating node in which its deadline is 64 times greater than the *RCS Mult* node, but the deadline values for the remaining nodes are the same as the *Corner Turn* node. Since $i > 0$ and $y_i \leq d_i < d_{i+1}$, $B_{\mathrm{BF}}(Q_i)$ and $B_{\mathrm{DF}}(Q_i)$ for the queue labeled *RCS* are evaluated with the same expression:

$$\begin{aligned}B_{\mathrm{BF}}(Q_4 = RCS) &= B_{\mathrm{DF}}(Q_4 = RCS) \\ &= B_{\mathrm{EDF}}(Q_4 = RCS) \\ &= \left(\left\lfloor \frac{d_{Corner\ Turn}}{y_{RCS\ Mult}} \right\rfloor \cdot x_{RCS\ Mult} \cdot p_{\mathrm{RCS}}\right) + r_{\mathrm{RCS}}\end{aligned}$$

$$= \left(\left\lfloor \frac{64 \cdot 3.6\text{ ms}}{3.6\text{ ms}} \right\rfloor \cdot 1 \cdot 256 \right) + (256 \cdot 127)$$
$$= (64 \cdot 256) + (256 \cdot 127)$$
$$= 256 \cdot 191 = 48{,}896.$$

By Theorem 7.3, the total buffer space required for the graph to execute with EDF or BF-EDF scheduling is less than or equal to 148,086 tokens. Since $d_{i+1} \neq d_i, \forall\, i$, Theorem 7.4 is not applicable. By Theorem 7.6, the total buffer space required to execute the SAR graph with DF-EDF scheduling when $d_i = y_i$ is less than or equal to $\sum_{i=0}^{n-1} B_{\mathrm{DF}}(Q_i) = 82{,}806$.

Theorems 7.3, 7.4 and 7.6 are upper bounds on buffer needs for the graph, depending on deadline values and scheduling algorithms. For some graphs, these may even be least upper bounds. However, these are not tight bounds for the SAR graph. The buffer bounds derived here can be reduced further by taking advantage of specific attributes of the SAR graph and features of the scheduling algorithms, as shown in [14].

8. SUMMARY

In most processing graph methodologies used to build high assurance systems, system engineers are unable to analyze the real-time properties of processing graphs like those created using PGM. We have shown that this is not an intrinsic property of the methodologies, and that by applying scheduling theory to a PGM graph, we can determine exact node execution rates, which are dictated by the input data rate and the dataflow attributes of the graph. We have also shown how to bound and manage latency and buffer requirements for an implementation of the graph scheduled with simple EDF algorithms under the RBE task model.

Given a graph, the only free parameters we have to affect the latency or buffer bounds of the application are deadlines. If the latency requirement of the application is less than the latency value from the strong synchrony hypothesis (i.e. $(F_{0 \rightsquigarrow n} - 1) \cdot y_0$), then the given graph will never meet its latency requirement since this latency is inherent in the graph. If the latency requirement is greater than the strong synchrony hypothesis bound but less than the lower bound $(F_{0 \rightsquigarrow n} - 1) \cdot y_0 + \sum_{i=1}^{n} e_i$, changing deadlines will not help the graph meet its latency requirement; a faster CPU is required.

If the latency requirement is greater than this lower bound but less than the upper bound $(F_{0 \rightsquigarrow n} - 1) \cdot y_0 + d_n$ (where $d_i < d_{i+1}, 1 \leq i < n$), then one can attempt to follow the procedures outlined in Section 6.3 to reduce latency to the desired bound. Should this technique fail, the system engineer may need to make cost trade offs. For example, if the deadline assignment technique outlined in Section 6.3 failed to yield satisfactory latency bounds before the schedulability test returned a negative result, the system engineer can decide whether to use a faster processor, or add memory to increase buffering. It is clear that the first choice resolves the latency problem, assuming a fast enough CPU exists. It may not be clear, however, that adding memory can reduce latency. Suppose the deadlines have been reduced such that the first k nodes in the chain all have deadlines equal to their rate interval (i.e. $d_i = y_i, \forall\, i : 1 \leq i \leq k$), and the last $(n - k)$ nodes have deadline values of d_k, but the latency bound is still too high; and lowering the deadline parameters for the last $(n - k)$ nodes yields a negative result from Equation (7). We may be able to reduce the latency bound further by setting all of the deadline parameters to $\mathit{LatencyRequirement} - (F_{0 \rightsquigarrow n} - 1) \cdot y_0$. This increases the buffer requirements of the first k nodes, but may produce enough slack in the schedule such that the graph is now schedulable even though the deadline parameters of the last $(n - k)$ nodes have been reduced to achieve the desired latency bound. Should the graph become schedulable with these new deadline parameters, but require too much memory, the system engineer can make cost trade offs: more memory, faster CPU or relaxed requirements.

Since our driving application has the topology of a chain, for space consideration we have restricted our analysis to chains. Many of the results presented in this paper have been extended to general PGM graphs in [4, 10, 20].

REFERENCES

[1] Lee, E. A. and Messerschmitt, D. G. (1987) Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comp.*, **C-36**, 24–35.

[2] *Processing Graph Method Specification* (1987) Prepared by the Naval Research Laboratory for use by the Navy Standard Signal Processing Program Office (PMS-412), Version 1.0.

[3] Goddard, S. (1998) *On the Management of Latency in the Synthesis of Real-Time Signal Processing Systems from Processing Graphs*. PhD Dissertation, University of North Carolina at Chapel Hill. http://www.cse.unl.edu/~goddard/Papers/Dissertation.ps

[4] Goddard, S. and Jeffay, K. (2000) The synthesis of real-time systems from processing graphs. *Proc. 5th IEEE Int. Symp. on High Assurance Systems Engineering*, Albuquerque, NM, pp. 177–186. IEEE Computer Society Press, Los Alamitos, CA.

[5] Chatterjee, S. and Strosnider, J. (1995) Distributed pipeline scheduling: A framework for distributed, heterogeneous real-time system design. *Comp. J.*, **38**, 271–285.

[6] Dasdan, A., Ramanathan, D. and Gupta, R. K. (1998) A timing-driven design and validation methodology for embeded real-time systems. *ACM Trans. Design Automaton Electron. Syst. (HLDVT'97 Special Issue)*, **3**, 533–553.

[7] Buck, J., Ha, S., Lee, E. A. and Messerschmitt, D. G. (1994) Ptolemy: a framework for simulating and prototyping heterogeneous systems. *Int. J. Comp. Simul. (Special Issue on Simulation Software Development)*, **4**, 155–182.

[8] Bhattacharyya, S. S., Murthy, P. K. and Lee, E. A. (1996) *Software Synthesis from Dataflow Graphs*. Kluwer, Norwell, MA.

[9] Ritz, R., Willems, M. and Meyer, H. (1995) Scheduling for optimum data memory compaction in block diagram oriented software synthesis. *Proc. ICASSP 95*, Detroit, MI, pp. 133–143.

[10] Goddard, S. and Jeffay, K. (1998) Managing memory requirements in the synthesis of real-time systems from processing graphs. *Proc. 4th IEEE Real-Time Technology and*

Applications Symp., Denver, CO, 3–5 June, pp. 59–70. IEEE Computer Society Press, Los Alamitos, CA.

[11] Gerber, R., Seongsoo, H. and Saksena, M. (1995) Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Trans. Software Eng.*, **21**, 579–592.

[12] Bondy, J. A. and Murty, U. S. R. (1976) *Graph Theory with Applications*. North Holland, New York.

[13] Zuerndorfer, B. and Shaw, G. A. (1994) SAR Processing for RASSP Application. *Proc. 1st Ann. RASSP Conf.*, Arlington, VA, pp. 253–268. IEEE Computer Society Press, Los Alamitos, CA.

[14] Goddard, S. and Jeffay, K. (2000) *Managing Latency and Buffer Requirements in Processing Graph Chains*. Technical Report UNL-CSE-00-530, Computer Science and Engineering, University of Nebraska–Lincoln. http://www.cse.unl.edu/~goddard/Papers/TR-UNL-CSE-00530.ps

[15] Berry, G. and Cosserat, L. (1985) The ESTEREL synchronous programming language and its mathematical semantics. *Seminar on Concurrency*. *Lecture Notes in Computer Science*, **197**, 389–448. Springer, Berlin.

[16] Jeffay, K. and Goddard, S. (1999) A theory of rate-based execution. *Proc. 20th IEEE Real-Time Syst. Symp.*, Phoenix, AZ, pp. 304–314. IEEE Computer Society Press, Los Alamitos, CA.

[17] Baruah, S., Howell, R. and Rosier, L. (1990) Algorithms and complexity concerning the preemptively scheduling of periodic, real-time tasks on one processor. *Real-Time Syst. J.*, **2**, 301–324.

[18] Liu, C. and Layland, J. (1973) Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, **30**, 46–61.

[19] Jeffay, K. (1994) On latency management in time-shared operating systems. *Proc. 11th IEEE Workshop on Real-Time Operating Systems and Software*, Seattle, WA, pp. 86–90. IEEE Computer Society Press, Los Alamitos, CA.

[20] Goddard, S. and Jeffay, K. (1999) Analyzing the real-time properties of a U.S. navy signal processing system. *Proc. 4th IEEE Int. Symp. on High Assurance Systems Engineering*, Washington, DC, pp. 141–150. IEEE Computer Society Press, Los Alamitos, CA.