Analyzing the Real-Time Properties of a Dataflow Execution Paradigm using a Synthetic Aperture Radar Application*

Steve Goddard Kevin Jeffay
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

Abstract

Real-time signal processing applications are commonly designed using a dataflow software architecture. Here we attempt to understand fundamental real-time properties of such an architecture — the Navy's coarse-grain Processing Graph Method *(PGM).*

By applying recent results in real-time scheduling theory to the subset of PGM employed by the ARPA RASSP Synthetic Aperture Radar benchmark application, we identify inherent real-time properties of nodes in a PGM dataflow 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. Our results assume only the existence of a simple EDF scheduler and thus can be easily applied in practice.

1 Introduction

Signal processing algorithms are often defined in the literature using large grain dataflow graphs [12]: 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. Large grain dataflow provides 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 dataflow 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 dataflow techniques. As 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 to only 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.

Dataflow models implicitly define a temporal semantics for a processing graph by specifying lower bounds on 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 on the time that may elapse between when a node becomes eligible to execute and the time the node either commences or completes execution. Without one of these specifications, we are left with:

- no schedulability or admission control test — How does one determine if a set of nodes or a graph "fits" on a processor?
- undetermined latency properties — How does one determine if a graph meets its timing requirements?
- no upper bound on queue length — If latency is not bounded, memory requirements for a graph cannot be bounded and hence data loss may occur if enough storage is not provided.

System engineers use such metrics to size hardware and perform requirements verification. A cost tradeoff may be made on CPU utilization versus latency, or

*Supported, in part, by grants from the Intel and IBM corporations, and the National Science Foundation (grant CCR-9510156).

In: *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Montreal, Canada, June 1997, 60-71.

buffer space versus latency. High latency tolerances allow the use of a slower (and cheaper) CPU but may require more memory for increased buffer space. On the other hand, tighter latency requirements may demand a faster CPU (or lower utilization) but less memory. In keeping costs in line, a system architect uses these metrics to make fundamental design trade-offs.

Unfortunately, without the application of real-time scheduling theory to dataflow methodologies and a precise execution model, system architects have not been able to make these trade-offs in real-time dataflow systems. Even the Navy's own dataflow methodology, *Processing Graph Method* (PGM) [16], lacks real-time analysis techniques to support making cost trade-offs or to verify latency 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 Navy's standard signal processor). PGM has also been used to create a real-time Ka-band synthetic aperture radar (SAR) benchmark application for ARPA's Rapid Prototyping of Application Specific Signal Processors (RASSP) project.

In this paper, we present a novel application of real-time scheduling theory to the subset of PGM used in the RASSP SAR benchmark application. Using the SAR application graph as a driving problem, we identify inherent relationships existing in real-time dataflow that have not been recognized in the literature. We present theorems that characterize the non-trivial execution rates of every node in the dataflow graph as a function of input rates by applying existing real-time scheduling theory to the dataflow methodology. From scheduling theory, we get a scheduling condition for preemptive earliest deadline first (EDF) scheduling algorithms. If the scheduling condition returns an affirmative result, the graph can be scheduled (with a preemptive EDF algorithm) to meet specified execution deadlines. We also show how to set the deadline parameters to bound end-to-end latency and memory requirements.

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 node execution rates and a schedulability condition for EDF scheduling. Section 6 addresses latency issues and Section 7 shows how to bound the buffer requirements of an implementation of a graph. We summarize our contributions in Section 8.

2 Related Work

This research was inspired by the analysis techniques applied to three different dataflow models: the dataflow graphs found in the Software Automation for Real-Time Operations (SARTOR) project led by Mok [14, 15], Lee and Messerschmitt's Synchronous Dataflow (SDF) graphs [12] supported by the Ptolemy system [4], and the Real-Time Producer/Consumer (RTP/C) paradigm of Jeffay [8]. Unfortunately, none of these paradigms (or any other dataflow paradigms from the literature) correctly model the execution of PGM graphs.

The dataflow graphs of the SARTOR project have different (and incompatible) node execution rules from PGM. As with the SARTOR project, our goal is to demonstrate that we can apply real-time scheduling results to real-life applications.

Like the RTP/C paradigm, we use the structure of the graph to help specify execution rates of the processes. However, our execution model is capable of supporting much more sophisticated data flow models than RTP/C. Whereas RTP/C models processes as sporadic tasks, our paradigm uses the Rate-Based Execution (RBE) process model of [10] to more accurately predict processor demand. (The RBE process model is a generalization of sporadic tasks and the Linear-Bounded Arrival Process (LBAP) model employed by the DASH system [1].) Unlike the RTP/C paradigm, PGM supports *And* nodes (nodes that are eligible to execute only when all of the input queues are over threshold), which introduces different execution properties than those of the RTP/C paradigm.

The SDF graphs of Ptolemy utilize a subset of the features supported by PGM. In addition to supporting a more general dataflow model, our research differs from [12] in that we use dynamic, real-time, scheduling techniques rather than creating static schedules.

Our latency analysis is related to the work of Gerber et al. in guaranteeing end-to-end latency requirements on a single processor [6]. Our work differs from [6] in that we cannot assume a periodic task model and that our node execution rates are derived from the input data rate and the graph. Moreover, unlike [6], we do not introduce new (additional) tasks for the purpose of synchronization.

3 Dataflow Model

This section describes the features of PGM used in the SAR graph. For a complete description of PGM, see [16].

In PGM, a system is expressed as a directed graph of large grain nodes (processing functions) and edges (logical communication channels). The topology of the graph defines the flow of data from an input source to an output sink, defining a software architecture independent of the hardware hosting the application. The edges of a graph are typed First-In-First-Out (FIFO)

queues. The data type of the queue indicates the size of each token (a data structure) transported from a producer to a consumer. Tokens are appended to the tail of the queue (by the producer) and read from the head (by the consumer). The tail of a queue can be attached to at most one node at any time. Likewise, the head of a queue can be attached to at most one node at any time.

There are three attributes associated with a queue: a produce, threshold, and consume amount.[1] The produce amount specifies the number of tokens atomically appended to the queue when the producing node completes execution. The threshold amount represents the minimum number of tokens required to be present in the queue before the node may process data from the input queue. The consume amount is the number of tokens dequeued (from the head of the queue) after the processing function finishes execution. A queue is *over threshold* if the number of enqueued tokens meets or exceeds the threshold amount. Unlike many dataflow 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 is that they must be non-negative values and the consume amount must be less than or equal to the threshold. For example, a queue may have a produce of 2, a threshold of 5, and a consume of 3.

Although PGM supports general graphs consisting of nodes with multiple input queues and variable produce and consume values, the SAR graph does not use these features. Since our driving application has the topology of a chain of nodes, for space consideration we restrict our analysis to chains and simply note that all of the results presented in this paper can be extended to general PGM graphs.

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 [17].

The full SAR benchmark cannot execute in real-time on a single processor. Therefore, the RASSP project allocates a portions 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 source 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 sink 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. (For a generic chain, we would label the source node N_0 and the sink node N_{n+1}. The output queue for node N_i would be labeled Q_i.) Produce, threshold, and consume values are annotated below the queue. For example, the produce, consume, and threshold values of the queue labeled *Range* are all 118.

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.

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 $256 \cdot 128$ and a consume value of $256 \cdot 64$.

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

[1] In PGM, a produce, threshold, or consume attributes is associated with a node port rather than the queue. For the subset of PGM used by the SAR application, it is easier to associate these attributes with the queue rather than the node.

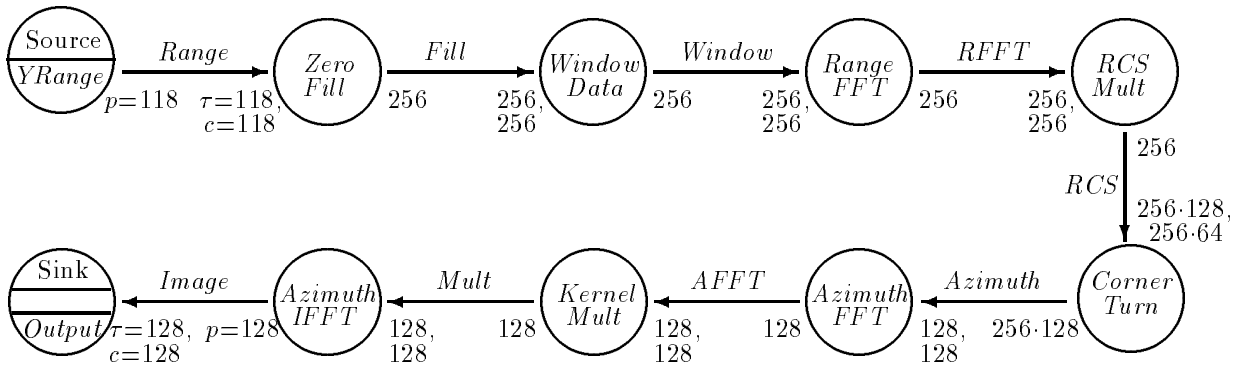

Figure 1: SAR Graph

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 *Sink* node, an inverse FFT is performed by the *Azimuth IFFT* node.

5 Execution Model

Real-time scheduling theory provides a framework upon which we have developed an execution model that supports bounding latency and memory usage for PGM graphs. These bounds in turn can be used to guarantee no data loss occurs during graph execution. We also appeal to scheduling theory to provide guarantees that these bounds will be met without the need to check for violations during graph execution (assuming the basic assumptions made during the analysis phase are true at run-time).

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. The remaining subsections address node execution rates and the RBE task model. These concepts are used to model an implementation of the graph.

5.1 Node Executions

Before exploring the fundamental execution relationships that exist between producer/consumer nodes, we must first define the restrictions on node execution. In accordance with PGM, our execution model requires all of the input queues to a node to be over threshold before the node is eligible for execution. Standard practice in implementing dataflow systems ([8, 12, 15]), though not part of the PGM specification, is to disallow two overlapping executions of the same node; we have adopted this restriction. PGM also requires that data be read from an input queue at the beginning of node execution, but data is consumed after the node has produced data on its output queues, which simply makes it clear that a node requires simultaneous input and output buffer space. We add the common real-time dataflow restriction that no data loss can occur during graph execution. The following definitions provide a formal basis for discussing the execution of nodes.

Definition 5.1. Node N_i is *eligible for execution* when all of its input queues are over threshold.

Definition 5.2. The execution of a node is *valid* if and only if: (1) the node executes only when it is eligible for execution, no two executions of the same node overlap, (2) each input queue has its data atomically consumed after each output queue has its data atomically produced, and (3) data is produced at most once on an output queue during each node execution.

Definition 5.3. The execution of a graph is *valid* if and only if all of the nodes in the execution sequence have valid executions and no data loss occurs.

We introduce the execution relationship that exists between producer/consumer nodes using the two node portion of a generic chain shown in Figure 2. Unlike the SAR graph, $Chain_1$ of Figure 2 contains a queue whose produce, threshold, and consume values are relatively prime — this is done to illustrate the general relationships between dataflow attributes and node execution. N_i produces 4 tokens every time it executes. N_{i+1} has a threshold of 7 and consumes 3 after it executes. Consequently, N_i must fire twice before Q_i is over threshold and N_{i+1} executes for the first time. After N_{i+1} executes, it consumes only 3 tokens — leaving 5 tokens on Q_i. The third execution of N_i produces 4 more tokens (for a total of 9 tokens on Q_i) and N_{i+1} executes again, consuming 3 more tokens. The next execution of N_i results in 10 tokens on Q_i, and N_{i+1} is

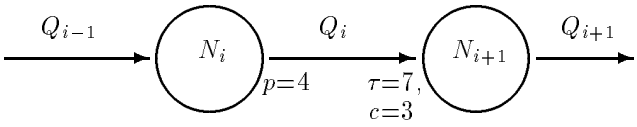

Figure 2: $Chain_1$

able to execute twice — leaving 4 tokens on Q_i, which is the same number that were on Q_i after the first execution of N_i. Hence, subsequent executions of N_i and N_{i+1} follow this same pattern.

The following lemma establishes the relationship for the number of executions of N_{i+1} as a function of the number of tokens produced by N_i.

Lemma 5.1. *Given $r \geq \tau_i$ tokens on Q_i, N_{i+1} will execute $\left\lfloor \frac{r-\tau_i}{c_i} \right\rfloor + 1$ times, consume $\left(\left\lfloor \frac{r-\tau_i}{c_i} \right\rfloor + 1 \right) \cdot c_i$ tokens, and leave r' tokens on Q_i where $(\tau_i - c_i) \leq r' < \tau_i$.*

Proof:[2] The number of times N_{i+1} will execute is the least natural number n such that $r-(n \cdot c_i) < \tau_i$, which implies $n > (r - \tau_i)/c_i$. The smallest natural number satisfying this inequality is $\left\lfloor \frac{r-\tau_i}{c_i} \right\rfloor + 1$. Since each execution of N_{i+1} consumes c_i tokens from Q_i, it immediately follows that the number of tokens consumed is $\left(\left\lfloor \frac{r-\tau_i}{c_i} \right\rfloor + 1 \right) \cdot c_i$. The number of tokens left on Q_i, r', is at least $(\tau_i - c_i)$ since if $r' < \tau_i - c_i$, the last execution was not enabled. Furthermore, if $r' \geq \tau_i$, another execution is enabled. Therefore, $(\tau_i - c_i) \leq r' < \tau_i$, and the lemma holds. □

Lemma 5.1 establishes the execution relationship that exists between producer/consumer nodes, but it does not tell us the frequency with which the node will execute — only how many times the consumer will execute given some number of tokens generated by the producer. The rate at which nodes execute is the subject of the next section.

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 dataflow is the basis of the results presented in this section.

[2] We thank the anonymous reviewer who suggested this proof and the proof of Lemma 5.2.

Most real-time execution models define task execution to be *periodic* or *sporadic*. Each time a task is ready to execute, it is said to be *released*. A periodic task is released exactly once every p time units (and p is called the period of the task). At least p time units separate every release of a sporadic task — no upper bound is given on subsequent releases of a sporadic task. Even when the source node of a PGM chain is periodic, the execution of the other nodes in the graph cannot be described as either periodic or sporadic. For example, consider $Chain_1$ of Figure 2. If N_i executes at times $0, y, 2y, \ldots$, N_{i+1} is eligible for one execution at y and $2y$, but twice at time $3y$. For this instance of the problem, we may be able to model N_{i+1} as two periodic or sporadic tasks (that interleave their execution), but this technique does not generalize. If the consume value is 5 rather than 3, we get a very different execution pattern. When the source is not periodic and data arrives in bursts, which is common in many implementations, even modeling a node as x invocations of a $(1, y)$ periodic or sporadic task is insufficient. An execution paradigm that supports generic rates of the form x executions in y time units is required to analyze the execution of generic dataflow graphs.

We assume the strong synchrony hypothesis of [5] to introduce the concept of node execution rates. Under the synchrony hypothesis, we assume the graph executes on an infinitely fast machine. Hence, each node takes "no time" to execute and data passes from source to sink node instantaneously. The synchrony hypothesis lets us define rate executions in the absence of scheduling algorithms and deadlines. Node execution rates are defined as follows.

Definition 5.4. The time of the j^{th} execution of node N_i is represented as $T_{i,j}$.

Definition 5.5. An execution rate is a pair (x, y). A node N_i, $\forall i > 0$, executes at rate $R_i = (x_i, y_i)$ if, $\forall j > 0$, N_i executes exactly x_i times in all time intervals of $[t + y_i \cdot (j-1), t + y_i \cdot j)$ where $t > T_{i,1}$.

Throughout this paper, we assume constant produce, threshold, and consume values with $c_i \leq \tau_i$. If the produce and consume values for a node are not constant, then the node's maximum produce and minimum consume values can be used to determine the maximum execution rate. We also assume a periodic source. As implied by Theorem 5.3, a periodic source is not required for our analysis techniques. All lemmas and theorems in this paper can be generalized to support the analysis of graphs that receive data from source nodes specified by rates rather than periods.

Given a periodic source node, N_0, we present and prove the execution rate for N_1, the second node in the

chain. Theorem 5.3 is a generalization of Lemma 5.2. Its proof, and the proofs for all subsequent lemmas and theorems, can be found in [7].

Lemma 5.2. *Assuming the strong synchrony hypothesis and no tokens on Q_0 prior to the beginning of graph execution, if $R_0 = (1, \rho)$ is the execution rate of N_0 with $T_{0,1} = 0$, then $R_1 = (x_1, y_1)$ is the execution rate of N_1 where $x_1 = \frac{p_0}{\gcd(p_0, c_0)}$ and $y_1 = \frac{c_0}{\gcd(p_0, c_0)} \cdot \rho$.*

Proof: Let $t > T_{1,1}$, and r be the number of tokens on Q_0 before any executions of N_1 at time t. By Lemma 5.1, $\tau_0 - c_0 \leq r < \tau_0$. Since $R_0 = (1, \rho)$, a total of $p_0 \cdot (c_0/gcd(p_0, c_0))$ tokens are enqueued on Q_0 over the interval $[t, t + y_1)$. Since each execution of N_1 removes c_0 tokens, x_1 executions during the interval $[t, t + y_1)$ will leave r tokens on Q_0. Furthermore, no more executions could have occurred since the ${x_1}^{th}$ execution leaves $r < \tau_0$ tokens on Q_0. Any fewer executions would have left $r > \tau_0$ tokens on Q_0, and another execution of N_1 would have occurred. Therefore, exactly x_{i+1} executions take place in this interval.

Simple induction shows that N_1 will execute exactly $\frac{p_0}{\gcd(p_0, c_0)}$ times in all intervals of $\left[t + (j-1) \cdot \frac{c_0}{\gcd(p_0, c_0)} \cdot \rho,\ \ t + j \cdot \frac{c_0}{\gcd(p_0, c_0)} \cdot \rho\right)$, $\forall j > 0$ where $t > T_{1,1}$,leaving r tokens on Q_0 at the end of each interval. Therefore R_1 is a valid rate specification for N_1. □

Theorem 5.3. *$\forall i \geq 0$: Assuming the strong synchrony hypothesis and no tokens on Q_i prior to the beginning of graph execution, if $R_0 = (x_0, y_0)$ is the execution rate of N_0, then the execution rate of N_{i+1} is $R_{i+1} = (x_{i+1}, y_{i+1})$ where $x_{i+1} = \frac{p_i}{\gcd(p_i, c_i)} \cdot x_i$ and $y_{i+1} = \frac{c_i}{\gcd(p_i, c_i)} \cdot y_i$.*

Theorem 5.3 can be used to derive the execution rate of every node in the SAR graph. For example, assuming $R_{YRange} = (1, y)$ is the execution rate of the source node, the execution rates of the other nodes in the SAR graph are: $R_{ZeroFill} = R_{WindowData} = R_{RangeFFT} = R_{RCSMult} = (1, y)$, $R_{CornerTurn} = (1, 64y)$, and $R_{AzimuthFFT} = R_{KernelMult} = R_{AzimuthIFFT} = (256, 64y)$. We will use these numbers later to develop latency and buffer bounds.

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. A scheduling algorithm and a schedulability test that will analytically determine whether or not a graph will meet its temporal requirements are also necessary. We have already seen that nodes are neither periodic nor sporadic, even when the source is periodic, which eliminates most execution models from the literature. Nevertheless, it is appealing to implement each node as a task that is released when the input queue goes over threshold. If we schedule the tasks using the preemptive *earliest deadline first* (EDF) scheduling algorithm, we can verify the real-time requirements of the application using the techniques Jeffay has developed for the *Rate Based Execution* (RBE) model [10].

RBE is a general task model that consists 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 in an interval of length y. The response time parameter d specifies the maximum time between release of the task 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.

In the RBE model, the j^{th} release of task T_i at time $t_{i,j}$ is guaranteed to complete execution by time $D_i(j)$, where

$$D_i(j) = \begin{cases} t_{i,j} + d_i & \text{if } 1 \leq j \leq x_i \\ \max(t_{i,j} + d_i, D_i(j - x_i) + y_i) & \text{if } j > x_i \end{cases} \tag{5.1}$$

The second line of the deadline assignment function (5.1) 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.

Jeffay established and proved the following feasibility condition for an RBE task set.

Theorem 5.4. *Let* $\mathbf{T} = \{(x_1, y_1, d_1, e_1), \ldots (x_n, y_n, d_n, e_n)\}$ *be a set of tasks.* $\mathbf{T}$ *will be feasible if and only if*

$$\forall L > 0, \quad L \geq \sum_{i=1}^{n} f\left(\frac{L - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i \tag{5.2}$$

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

In [10], Jeffay established sufficiency of (5.2) by showing that the preemptive EDF scheduling algorithm can schedule releases of the tasks in $\mathbf{T}$ without a task missing a deadline if the task set satisfies (5.2). For a PGM graph, (5.2) 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). (5.2) is not a necessary condition 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 N_{i+1} in Figure 2, this is not possible.

Note that if the cumulative processor utilization for a graph is strictly less than one (i.e., $\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} < 1$) then condition (5.2) can be evaluated efficiently (in pseudo-polynomial time) using techniques developed in [2] and applied in [3] and [11].

6 Latency

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 p_0 tokens delivered each period by N_0 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 p_0 tokens onto Q_0 by the source node N_0 and the next enqueuing of p_n tokens on Q_n by node N_n.

Latency is a function of the scheduling algorithm. It is the case for graph models, however, that latency also has a structural component. The next section illustrates this property.

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 3. 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 128^{th} pulse received by the SAR graph. As shown in Figure 3, the 128^{th} pulse arrives at time $127y_0$ 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 $127y_0$. 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 3 for the SAR graph, this maximum latency is encountered by pulses 129, 193, 257, 321, ..., which have a latency of $63y_0$. Notice that there are 126 other unique latency values for this simple graph (e.g., the latency for pulse $j+1$ is $(127-j)y_0$).

The latency encountered by a sample of the signal (under the strong synchrony hypothesis) is dependent on the data flow attributes of the graph and the state of the queues (i.e., the number of tokens on each queue of the graph) when the sample arrives. We can determine the magnitude of a sample's latency by determining how many more samples are required before node N_n executes. Lemma 6.1 fulfills this role.

Lemma 6.1. *Given $r_k < \tau_k$ tokens on Q_k $\forall k : i \leq k < j$, N_i must execute $F(N_i, N_j)$ times to produce enough data to put Q_{j-1} over threshold (and thus making N_j eligible for execution) where $\forall i, j : 0 \leq i < j \leq n$ ::*

$$F(N_i, N_j) = \begin{cases} \left\lceil \frac{\tau_i - r_i}{p_i} \right\rceil & \text{if } i+1 = j \\ \left\lceil \frac{(F(N_{i+1}, N_j)-1) \cdot c_i + \tau_i - r_i}{p_i} \right\rceil & \text{if } i+1 < j \end{cases}$$

Evaluating $F(N_0, N_n)$ just before the i^{th} sample's arrival will tell us how many samples are required before N_n will be eligible for execution. Hence, as implied by Lemma 6.2, the latency the i^{th} sample will encounter is given by $(F(N_0, N_n)-1) \cdot y_0$ when $F(N_0, N_n)$ is evaluated just before the sample arrives. We subtract one from $F(N_0, N_n)$ before converting it to time units since the latency interval begins after the sample arrives.

Lemma 6.2. *Given $R_0 = (1, \rho)$. When $F(N_0, N_n)$ is evaluated just before the sample's arrival,*

$$\textit{Sample Latency} = (F(N_0, N_n) - 1) \cdot \rho \qquad (6.1)$$

Applying Lemma 6.2 to the SAR graph, we find the same latency values identified at the beginning of this subsection.

6.2 Latency in an Implementation

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 §6.1, the graph attributes. The lower bound for the latency interval is the latency value derived using (6.1) 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 $(F(N_0, N_n) - 1) \cdot y_0 + \sum_{i=1}^{n} e_i$.

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

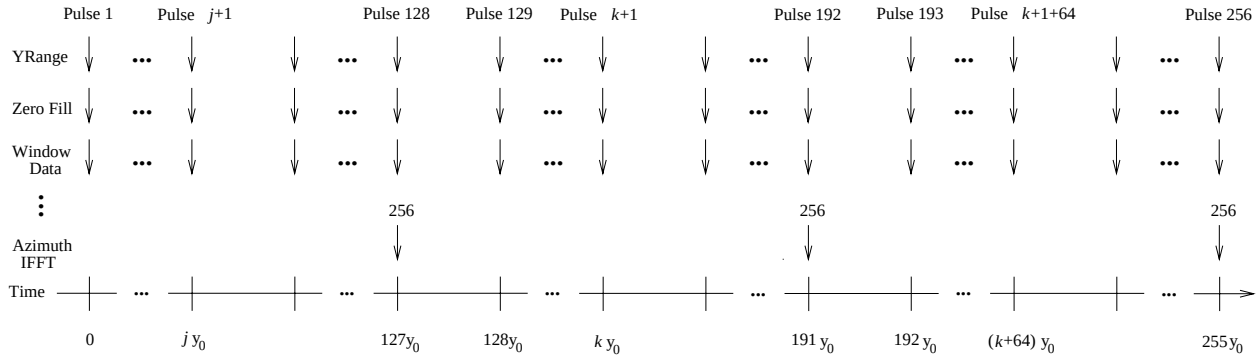

Figure 3: Latency for the SAR graph under strong synchrony hypothesis. Each down arrow represents the release and instantaneous execution of a node.

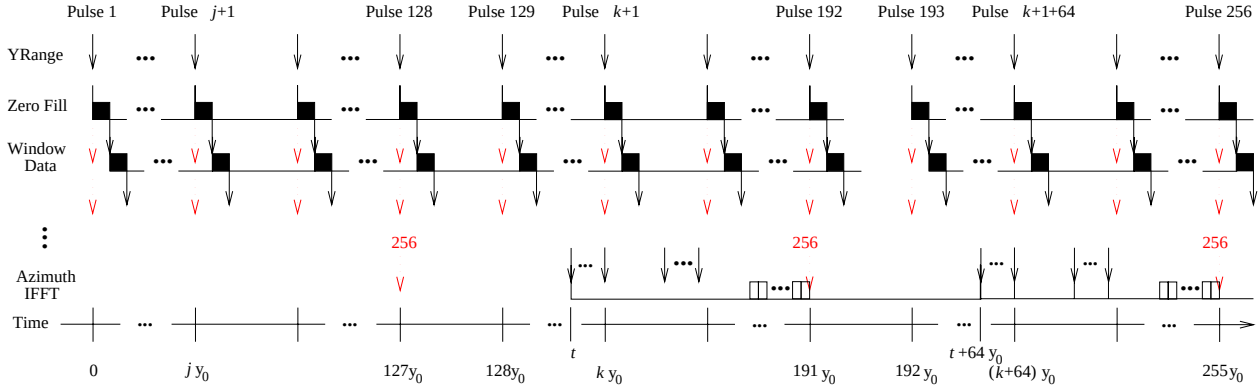

Figure 4: Latency for the SAR graph. A light arrow represents a node's release under the strong synchrony hypothesis. A dark arrow represents the actual release time, and the node's execution is represented by a box.

the 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., (5.2) returns a negative result) then the processor is overloaded since (5.2) reduces to the Lui & Layland feasibility test [13] and we get $1 < \sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i}$. 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 4 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 4 that task *Zero Fill* is released at times 0, y_0, $2y_0$, $3y_0$, ..., and the deadlines corresponding to each release time is y_0, $2y_0$, $3y_0$, ... since $d_1 = y_1 = y_0$. Due to scheduling and execution times, however, the task *Window Data* is not released until times $0 + e_1$, $y_0 + e_1$, $2y_0 + e_1$, $3y_0 + e_1$, ..., and the corresponding deadlines are $0 + e_1 + d_2 = y_0 + e_1$, $2y_0 + e_1$, $3y_0 + e_1$, ... 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 256^{th} execution of task *Azimuth IFFT* completes execution by time $191y_0$ — well before its deadline.

The release times shown in Figure 4 for the tasks *Zero Fill* and *Window Data* are the earliest possible release times. As we have noted, the task *Azimuth IFFT* completes its 256^{th} 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 (5.2). From Lemma 6.2, we know that the first release of task *Azimuth IFFT* cannot occur before $127y_0$. An affirmative

result from (5.2) means that there exists enough processor capacity for nodes N_1 thru N_k, $1 \leq i \leq k \leq n$, to execute $\frac{y_k}{y_i} \cdot x_i$ times during an interval of length y_k. This means that 64 executions of *Zero Fill*, *Window Data*, *Range FFT*, and *RCS Mult*; 1 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 [9], 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. Theorem 6.3 uses this fact to provide a lower and upper bound for any sample's latency.

Theorem 6.3. *Given $R_0 = (1, \rho)$ 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 deadline assignment function* (5.1) *is bounded such that*

$$(F(N_0, N_n) - 1) \cdot \rho + \sum_{i=1}^{n} e_i \leq \textit{ Sample Latency} \leq (F(N_0, N_n) - 1) \cdot \rho + d_n$$

where $F(N_0, N_n)$ is evaluated just before the sample's arrival.

Theorem 6.3 tells us that if a graph is schedulable by (5.2), task N_i will complete execution within d_i time units of the release time calculated under the strong synchrony hypothesis. Therefore, if we let the released task N_i *inherit* the release time of its predecessor in the dataflow graph (i.e., N_{i-1}) and calculate its deadline by adding d_i to this *logical* release time, we get a deadline equal to the time that Theorem 6.3 states the task will have finished executing. Since the first node of a chain receives data from an external device, it has no release time to inherit and its logical release time is the same as its actual release time. Consider the execution diagram of Figure 4. *Zero Fill* is first released at time 0. The first actual release of *Window Data* occurs after *Zero Fill* completes execution, at time e_1, and its deadline is set to $y_0 + e_1$. The logical release of *Window Data*, however, occurs at time 0 since this is the release time inherited from *Zero Data*. (It is also the release time derived under the strong synchrony hypothesis.) Using the logical release time, we get a deadline of y_0 for *Zero Data*, which is the time Theorem 6.3 gave as the upper bound for when the task will finish execution if the task set is schedulable. Similarly, we get a logical release time of $127y_0$ and a deadline of $191y_0$ for the first execution of *Azimuth IFFT*.

As long as the scheduler ensures that a task only executes when its input queue is over threshold, it does not matter if N_{i+1} executes before N_i. When the RBE task set is specified such that $d_i \leq d_{i+1}$, a release of N_{i+1} will never be assigned a deadline earlier than a release of N_i, even when logical release times are used. Moreover, the latency bound of Theorem 6.3 holds even when a release of N_{i+1} executes before a release of 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 N_{i+1} executing before 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 §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 (5.2) 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 (5.2) 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 Bounding Buffers

This sections gives bounds for the buffer requirements of chains executed under the RBE model with release inheritance, as described in the previous section. We use logical release times rather than actual release times so that deadline ties are created 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.

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. 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.

The most tokens Q_i can hold without being over

threshold is represented by r_i where[3]

$$r_i = \begin{cases} \tau_i - \gcd(p_i, c_i) & \text{if } \exists k : \tau_i = k \cdot \gcd(p_i, c_i) \\ \left\lfloor \frac{\tau_i}{\gcd(p_i, c_i)} \right\rfloor \cdot \gcd(p_i, c_i) & \text{otherwise} \end{cases}$$

After Q_i goes over threshold, the number of tokens that can accumulate on the queue is a function of dataflow attributes, deadlines, and the scheduling algorithm.

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 becomes 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 5 — a possible snapshot

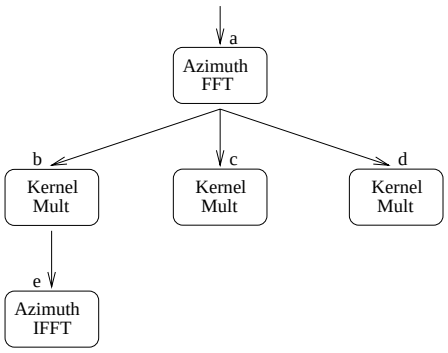

Figure 5: A scheduling graph.

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 5, 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 5, 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*.

[3]We thank Sanjoy Baruah for the inspiration that led to the tight bound represented by r_i.

7.1 Buffer Bounds with BF-EDF

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. Under BF-EDF scheduling, the input queue to *Azimuth IFFT* would accumulate data from 256 executions of *Kernel Mult* before *Azimuth IFFT* would execute.

The function $B_{BF}(Q_i)$, which assumes release time inheritance, returns the maximum number of tokens the i^{th} queue will ever hold when the graph is scheduled with either BF-EDF or the canonical EDF scheduling algorithm.

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

The last expression in $B_{BF}(Q_i)$ handles the cases when $d_i = d_{i+1}$ (which creates deadline ties) or $d_{i+1} \leq y_0$ (which means N_1 through N_{i+1} all complete their executions before the next produce by N_0) by working "back up the chain".

Since EDF does not specify how ties are broken, we would need to sum $B_{BF}(Q_i)$ 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 $\max(B_{BF}(Q_{i-1}) - r_{i-1}, B_{BF}(Q_{i+1}) - r_{i+1}) + r_{i-1} + r_{i+1}$. Theorem 7.1 divides the queues into two disjoint sets and uses this technique to bound the total buffer space required by the chain.

Theorem 7.1. *For the BF-EDF scheduling algorithm with release time inheritance and $d_{i+1} \geq d_i$, $\forall i : 0 \leq i < n$, the maximum buffer space required is $\leq \beta$, where*

$$\beta = B_{BF}(Q_0) + \sum_{i=1}^{n-1} r_i + \max\{B_{BF}(Q_k) - r_k \mid \forall k = 2i : i > 0 \ \wedge \ k < n\} + \max\{B_{BF}(Q_k) - r_k \mid \forall k = 2i-1 : i > 0 \ \wedge \ k < n\}$$

7.2 Buffer Bounds with DF-EDF

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. The DF-EDF schedule, starting with the schedule graph of Figure 5, is e, b, e', c, e'', d, e''', a and the input queue to *Azimuth IFFT* only accumulates data from one execution of *Kernel Mult* before it executes. The function $B_{DF}(Q_i)$ returns the maximum number of tokens Q_i will ever hold when the graph is scheduled with release inheritance and DF-EDF. 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.

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

Theorem 7.2. *For DF-EDF scheduling with release time inheritance and $d_{i+1} \geq d_i$, $\forall i : 0 \leq i < n$, the maximum buffer space required is less than or equal to $\sum_{i=0}^{n-1} B_{DF}(Q_i)$.*

7.3 Buffer Bounds for the SAR graph

Table 1 shows the values returned from $B_{BF}(Q_i)$ and $B_{DF}(Q_i)$ for each queue in the SAR graph with $d_i = y_i$. These values were used to derive the maximum buffer space required to execute the graph when release time inheritance is used in conjunction with EDF, BF-EDF, and DF-EDF scheduling: 148,086; 81,782; and 82,806 respectively. Using $d_i = y_i$ in the SAR graph, BF-EDF scheduling yields the lowest memory bound.

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

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

It remains an open questions as to whether BF-EDF scheduling actually uses the least memory. We believe DF-EDF scheduling will actually use the least memory for the SAR graph, but we do not yet have a tight bound for DF-EDF scheduling of general chains.

8 Summary

In most "real-time" dataflow methodologies, system engineers are unable to analyze the real-time properties of dataflow 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 latency and buffer requirements for an implementation of the graph scheduled with the preemptive EDF algorithm (and variations thereof) 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., $(F(N_0, N_n) - 1) \cdot y_0$), then the given graph will never meet its latency requirement. If the latency requirement is greater than the strong synchrony hypothesis bound but less than the lower bound $(F(N_0, N_n) - 1) \cdot y_0 + \sum_{i=1}^{n} e_i$, 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 $(F(N_0, N_n) - 1) \cdot y_0 + d_n$ (where $d_i < d_{i+1}, 1 \leq i < n$) then one can attempt to follow the procedures outlined in §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 §6.3 failed to yield satisfactory latency bounds before the schedulability test returned a negative result, the system engineer can de-

cide 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 (5.2). We may be able to reduce the latency bound further by setting all of the deadline parameters to $LatencyRequirement - (F(N_0, N_n) - 1) \cdot y_0$. 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 and note that the results presented in this paper can be extended to general PGM graphs.

References

[1] Anderson, D.P., Tzou, S.Y., Wahbe, R., Govindan, R., Andrews, M., "Support for Live Digital Audio and Video", *Proc. of the Tenth International Conference on Distributed Computing Systems*, Paris, France, May 1990, pp. 54-61.

[2] Baruah, S., Mok, A., Rosier, L., "Algorithms and Complexity Concerning the Preemptively Scheduling of Periodic, Real-Time Tasks on One Processor" *Real-Time Systems Journal*, Vol. 2, 1990, pp. 301-324.

[3] Baruah, S., Mok, A., Rosier, L., "Preemptively Scheduling Hard-Real-Time Sporadic Tasks With One Processor" *Proc. 11th IEEE Real-Time Systems Symp.*, Lake Buena Vista, FL, Dec. 1990, pp. 182-190.

[4] Buck, J., Ha, S., Lee, E.A., Messerschmitt, D.G., "Ptolemy: A Framework For Simulating and Prototyping Heterogeneous Systems", *International Journal of computer Simulation, special issue on Simulation Software Development*, Vol. 4, 1994.

[5] Berry, G., Cosserat, L., "The ESTEREL Synchronous Programming Language and its Mathematical Semantics", Lecture Notes in Computer Science, Vol. 197 Seminar on Concurrency, Springer Verlag, Berlin, 1985.

[6] Gerber, R., Seongsoo, H., Saksena, M., "Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes", *Proc. of IEEE Real-Time Systems Symposium*, Dec. 1994.

[7] Goddard, S., Jeffay, K. "Analyzing the Real-Time Properties of a Dataflow Execution Paradigm using a Synthetic Aperture Radar Application", Technical Report TR97-007, Dept. of Computer Science, University of North Carolina at Chapel Hill, April 1997.

[8] Jeffay, K., "The Real-Time Producer/Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems", *Proc. of the ACM/SIGAPP Symposium on Applied Computing*, Indianapolis, IN, February 1993, pp. 796-804.

[9] Jeffay, K., "On Latency Management in Time-Shared Operating Systems", *Proc. of the* 11^{th} *IEEE Workshop on Real-Time Operating Systems and Software*, Seatle, WA, May 1994, pp. 86-90.

[10] Jeffay, K., Bennett, D. "A Rate-Based Execution Abstraction For Multimedia Computing", *ACM Multimedia Systems*, to appear.

[11] Jeffay, K., Stone, D., "Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems", *Proc. of the* 14^{th} *IEEE Symposium on Real-Time Systems*, Durham, NC, 1993, pp. 212-221.

[12] Lee, E.A., Messerschmitt, D.G., "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Transactions on Computers*, Vol. C-36, No. 1, January 1987, pp. 24-35.

[13] Liu, C., Layland, J., "Scheduling Algorithms for multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM*, Vol 30., Jan. 1973, pp. 46-61.

[14] Mok, A.K., Sutanthavibul, S., "Modeling and Scheduling of Dataflow Real-Time Systems", *Proc. of the IEEE Real-Time Systems Symposium*, San Diego, CA, Dec. 1985, pp. 178-187.

[15] Mok, A. K., et al., "Synthesis of a Real-Time System with Data-driven Timing Constraints", *Proc. of the IEEE Real-Time Systems Symposium*, San Jose, CA, Dec. 1987, pp. 133-143.

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

[17] Zuerndorfer, B., Shaw, G.A., "SAR Processing for RASSP Application", *Proc. of* 1^{st} *Annual RASSP Conference*, Arlington, VA, August 15-18, 1994.