A Software Synthesis Method for Building Real-Time Systems from Processing Graphs

Steve Goddard[*] *Kevin Jeffay*[†]

Technical Report TR98-002
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
{*goddard, jeffay*}*@cs.unc.edu*

January 1998

Keywords: Real-time systems, software engineering, embedded systems, processing graphs, scheduling theory.

Abstract

Embedded signal processing applications are commonly designed using a processing graph software architecture. Unfortunately, existing technology does not support mapping general processing graphs to predictable real-time systems that use on-line scheduling.

In this paper, we present software synthesis techniques that create a real-time uniprocessor system from processing graphs. Our synthesis method supports both latency and buffer management, though only buffer management is addressed in this paper (for space considerations). To achieve the synthesis, new rate theorems have been developed to derive the execution rate of each processing component in the graph. The processing graph is mapped to the existing Rate Based Execution (RBE) task model, and if the schedulability test for the resulting task set is affirmative, we show that efficient buffer management can be achieved using a simple Earliest Deadline First (EDF) scheduler.

We evaluate our results using an International Maritime Satellite (INMARSAT) mobile receiver application. The case study shows that a dynamically scheduled implementation of the INMARSAT mobile satellite receiver uses less memory and introduces less latency than a comparable static implementation.

[*]Supported, in part, by the University of North Carolina at Chapel Hill Department of Computer Science Alumni Fellowship, grants from the Intel and IBM corporations, and the National Science Foundation (grant CCR-9510156).

[†]Supported, in part, by grants from the Intel and IBM Corporations, and the National Science Foundation (grant CCR-9510156).

1 Introduction

Large grain processing graphs have become an important design aid in the development of complex digital signal processing systems. These directed graphs consist of nodes and graph edges depicting the flow of data from one node to the next. When sufficient data arrives, a node executes from start to finish in isolation (i.e., without synchronization).

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 graph methodology allows one to easily understand the signal processing performed by graphically depicting the structure of the algorithm. In this manner any portion of the application can be understood in the absence of the rest of the algorithm.

The primary problem in developing embedded signal processing applications with a processing graph methodology is creating a predictable real-time system in which one can determine if the graph "fits" on a uniprocessor. Typically a processing graph is mapped to a real-time task model (e.g., [7, 12, 16, 17, 19, 24, 22, 23]) and the accompanying schedulability condition is used to see if the graph "fits" on the processor. Unfortunately for general processing graph methodologies, such as the U.S. Navy's Processing Graph Method (PGM) [18], there is no obvious mapping to existing task models. Without a real-time task model and accompanying schedulability condition one cannot evaluate the latency or memory requirements of the application. In this paper, we present new theorems that characterize the non-trivial node execution rate of every node in a PGM graph, and demonstrate the synthesis of real-time systems from acyclic processing graphs by mapping the graph to the existing Rate Based Execution (RBE) task model developed by Jeffay [13]. The accompanying RBE schedulability condition is used to verify processor capacity. An affirmative result from the schedulability condition lets us bound latency and buffer requirements for the application.

Once the processing graph has been mapped to a task model, issues such as latency and memory requirements can be addressed. Due to environmental restrictions on size, weight, and power, many embedded signal processing systems do not offer programs an abundance of memory storage. In the past, processor speed was also a major factor limiting the features an embedded system could provide. Today, however, processor speed is a minor problem for acoustic signal processing applications compared with memory size and its power consumption.

Sizing the system to fit memory requirements is the second major concern in many embedded acoustic signal processing applications (with mapping the processing graph to a predictable real-time system being the first). An important aspect in controlling memory usage is the node execution schedule since this impacts the buffer requirements of the edges in the graph. The state of the art in minimizing the memory requirements of a signal processing graph is to create a node execution schedule off-line [15, 20, 25, 21, 4]. In contrast, our approach is to use a dynamic, on-line scheduler for graph execution. We show that preemptive earliest deadline first (EDF) scheduling can achieve near optimal memory usage and often requires less memory

than static schedules created by off-line schedulers designed to minimize memory usage. Moreover, dynamic scheduling introduces less latency and can provide more control over deterministic response times of selected processing paths than static scheduling.

To illustrate these concepts and to quantify the memory savings achievable in a real application, we analyze an existing processing graph implementation of an International Maritime Satellite (INMARSAT) mobile satellite receiver application. (INMARSAT is a global maritime communication and navigational system.) The case study demonstrates that the best static schedulers require between 1.94% and 291.74% more buffer space than our dynamic scheduling approach. (It is difficult to state precisely the amount of buffer space required by statically scheduled executions since the heuristic algorithms that create the schedule do not account for the amount of data that must accumulate on the graph input edges before the execution commences.)

It is a minor extension of the work presented here to include cyclic processing graphs. However, due to space considerations, results related to cyclic processing graphs and latency management are not presented.

The rest of the paper is organized as follows. Our results are related to other work in Section 2. Section 3 presents a brief overview of the processing graph model used. The synthesis of real-time systems from processing graphs is presented in Section 4 with buffer management in the real-time system addressed in Section 5. We evaluate our results in Section 6 with a case study of an International Maritime Satellite (INMARSAT) mobile receiver application. Our contributions are summarized in Section 7.

2 Related Work

Our software synthesis technique begins with the U.S. Navy's coarse-grain Processing Graph Method (PGM) [18], which is based on computation graphs introduced by Karp and Miller [14]. Another synthesis methodology based on computation graphs is the Synchronous Dataflow (SDF) software synthesis method [4, 15]. For our purposes, the differences between SDF graphs and PGM graphs are minor and are presented in §3. However, the differences between the two synthesis methods are dramatic.

The main goal of the SDF synthesis method and related scheduling research has been to minimize memory usage by creating off-line scheduling algorithms (e.g., [15, 20, 25, 21, 4]). Off-line schedulers create a static schedule that is executed periodically (or cyclically) by the processor. The primary goal of our synthesis method and related research has been to manage latency and buffer requirements of processing graphs executed with a work-conserving on-line scheduler [2, 9, 10, 11]. In contrast to the off-line schedulers developed for SDF graphs, on-line scheduling of PGM graphs uses input data rates and schedulability conditions from real-time scheduling theory.

The real-time literature is replete with examples in which some form of a processing graph is used to construct real-time systems [7, 8, 12, 16, 17, 19, 24, 22, 23] (to list a few). The processing graphs of [7, 19] and [24] are task graphs in which only precedence constraints are described. An even simpler processing graph, called task chains, is employed by [8, 22] and [23]. Both task graphs and task chains can be represented with PGM graphs, but this paper does not address the distributed or end-to-end latency

issues of [7, 8, 19, 24, 22, 23].

From the real-time literature, the results presented in this paper are most closely related to the dataflow graphs found in the Software Automation for Real-Time Operations (SARTOR) project led by Mok [16, 17] and the Real-Time Producer/Consumer (RTP/C) paradigm of Jeffay [12]. Unfortunately, neither of these paradigms correctly model the execution of PGM graphs.

The dataflow graphs of the SARTOR project have different (and incompatible) node execution rules from PGM. Without creating a non-work-conserving execution of the graph, we cannot use a periodic task model. 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 [13] 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].)

3 An Introduction to Processing Graphs

This section introduces some graph theory and notation, which will be used throughout the paper, followed by a summary of the U.S. Navy's Processing Graph Method (PGM). We have selected PGM as the processing graph model since it is a very general paradigm that includes features not found in other processing graph models.

3.1 Graph Theory and Notation

In this paper, 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 nonempty finite set V of *vertices*, a finite set of *edges* E (disjoint from V), and an incidence function ψ that associates with each edge of E an ordered pair of (not necessarily distinct) vertices of V. Vertices are often called nodes or points, and edges of E are often called arcs or lines.

Consider and 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. If u and v are distinct, then $\{u, v\}$ is called an *adjacent pair*. Whether or not u and v are distinct, u is a *predecessor* of v, and v is a *successor* of u. 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. We call a vertex v with $\delta^-(v) = 0$ an *input node*. The set of all input nodes is denoted by $\mathcal{I}$ (i.e., $\mathcal{I} = \{v \mid v \in V \wedge \delta^-(v) = 0\}$). We call a vertex v with $\delta^+(v) = 0$ an *output node*. The set of all output nodes is denoted by $\mathcal{O}$ (i.e., $\mathcal{O} = \{v \mid v \in V \wedge \delta^+(v) = 0\}$).

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 $\forall i \; 1 \leq i < k : \exists e \in E :: \psi(e) = (w_i, w_{i+1})$. In

other words, there is path between $u = w_1$ and $v = w_k$ if there exists a sequence of vertices $(w_1, w_2, \ldots, w_k)$ such that w_i is adjacent to w_{i+1} for $i = 1, 2, \ldots (k-1)$. A path from some vertex to itself (i.e., $u \rightsquigarrow u$) is called a *cycle*. A path $u \rightsquigarrow v$ is a *chain* if $u \neq$v, $\delta^+(u) = 1$, $\delta^-(v) = 1$, and $\delta^+(w) = \delta^-(w) = 1$ for all $w \in \{\{u \rightsquigarrow v\} - \{u, v\}\}$. If there exists a path from $u \in V$ to $v \in V$, then u is an *ancestor* of v and v is a *descendant* of u.

When discussing PGM directed graphs, we use the term graph rather than digraph or directed graph since all PGM graphs are digraphs. We also use the terms nodes and queues to refer to elements of V and E respectively.

3.2 Processing Graph Method

As stated in §1, there are many processing graph models, but our synthesis method begins with the U.S. Navy's Processing Graph Method (PGM). PGM was developed by the U.S. Navy to facilitate the design and implementation of (acoustic) signal processing applications. This section provides a brief overview of the portions of PGM used in this paper.

In PGM, a system is expressed as a directed graph in which the nodes (or vertices) represent processing functions and and the edges represent logical communication channels. The function ψ is implicitly defined by the topology of the graph, which defines a software architecture independent of the hardware hosting the application. The graph edges are typed First-In-First-Out (FIFO) queues, and 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). As with any digraph, 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 amount $prd(q)$, a threshold amount $thr(q)$, and a consume amount $cns(q)$. The produce amount specifies the number of tokens 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 $thr(q)$. 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 is that they must be non-negative values and the consume amount must be less than or equal to the threshold. For example consider the portion of a chain shown in Figure 1. The queue connecting nodes u and v, labeled q, has $prd(q) = 4$, $thr(q) = 7$, and $cns(q) = 3$.

If a node has more then one input queue, 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 is 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

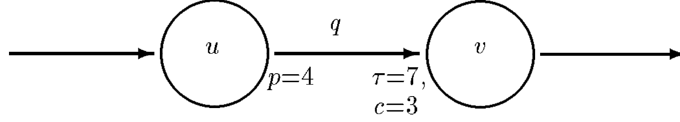

Figure 1: $Chain_1$. The dataflow attributes $prd(q)$, $thr(q)$, and $cns(q)$ are represented by the variables p, τ, and c respectively.

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.

Graph execution consists of executing a (possibly infinite) sequence of nodes from the set of nodes in the graph. 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. Standard practice in implementing processing graph systems ([12, 15, 17]), though not part of the PGM specification, is to disallow two overlapping executions of the same node; we have adopted this restriction.

For the purposes of this paper, the only significant difference between PGM graphs and SDF graphs is that SDF graphs require a queue's threshold value to be equal to its consume value (i.e., $\forall q \in E, thr(q) = cns(q)$). The graph of Figure 4 on page 17 is an abstraction of the processing graph for an INMARSAT mobile satellite receiver applications. (It is an abstraction since the nodes have been re-labeled with capital letters and the queue labels have been removed to simplify the presentation.) Since all of the queues have thresholds equal to their consume values, the acyclic graph of Figure 4 is an example of either an SDF or PGM processing graph.

4 Synthesis Method

Our synthesis method begins with the creation of a PGM processing graph. Once a processing graph has been created for the application, we identify the execution rates of graph nodes and map each node to a task in Jeffay's Rate Based Execution (RBE) task model [13]. The presentation of our software synthesis method for building real-time uniprocessor systems from PGM graphs (or any processing graph that can be represented as a PGM graph) is organized as follows. §4.1 introduces the concept of node execution rates and how the execution rate for each node in a PGM graph is derived. The RBE task model is described in §4.2 for completeness. §4.3 shows how to map the processing graph nodes to tasks in the RBE task set and presents a sufficient (but not necessary) schedulability condition for the resulting real-time system.

4.1 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 processing graph 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.

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 easily described as either periodic or sporadic. For example, consider $Chain_1$ of Figure 1 on page 5 once again. If node u executes at times 0, y, $2y$, ... , node v is eligible for one execution at times y and $2y$, but twice at time $3y$. Clearly neither a single periodic nor sporadic task can model node v. It is less obvious that three sporadic tasks are insufficient to model the execution of node v efficiently. As identified in [23], jitter caused by the execution of other nodes can create an execution pattern that has a much less than expected inter execution time for a particular node. For example, it may be possible for node u to complete one execution at time $y + 8$ and the next at time $2y + 1$ and so on, although we expect them to be separated by y time units. Even though one can force the graph execution to fit either a periodic or sporadic task model, it is unnatural and introduces additional complexity. An execution paradigm that supports expected rates of the form x executions in y time units is a much more natural and simpler task model for the analysis of schedulability, latency, and buffer requirements for a processing graph application.

We assume the strong synchrony hypothesis of [6] to introduce the concept of node execution rates. The strong synchrony hypothesis is that the system instantly reacts to external stimuli by updating internal states such that the response to input appears instantaneously. Under the strong synchrony hypothesis, we assume the graph executes on an infinitely fast machine and each node takes "no time" to execute; data passes from source to sink node instantaneously. The synchrony hypothesis lets us define rate executions in the absence of scheduling algorithms and deadlines. Throughout this section, we assume constant produce, threshold, and consume values with $cns(q) \leq thr(q)$. 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. Node execution rates are defined as follows.

Definition 4.1. The time of the jth execution of node v is represented as $T_j(v)$.

Definition 4.2. An execution rate is an integer pair (x, y). A node v, executes at rate $R(v) = (x, y)$ if v executes exactly x times in all time intervals of $[t,\ t + y)$ where $t > T_1(v)$.

Corollary 4.1. *If $R(v) = (x, y)$ is a valid rate specification for node v, then $m \cdot R(v) = (m \cdot x, m \cdot y)$ is also a valid rate specification for node v.*

Proof: The proof follows immediately from Definition 4.2. □

In [11], we derived execution rates for nodes in a PGM chain. The execution rate theorem of that paper is reproduced here (without proof[1]) since we will use this result to extend the analysis to include nodes with

[1] See [9] for the proof of Theorem 4.2.

multiple input queues.

Theorem 4.2. *Given a PGM chain $i \rightsquigarrow w$ such that $i \in \mathcal{I}$ (the set of input nodes), $u, v \in \{i \rightsquigarrow w\}$ with $\psi(q) = (u, v)$, and $R(i) = (x(i), y(i))$, then assuming the strong synchrony hypothesis and no tokens on queue q prior to the beginning of graph execution the execution rate of node v is $R(v) = (x(v), y(v))$ where*

$$x(v) = \frac{prd(q)}{\gcd(prd(q)x(u), cns(q))} \cdot x(u) \quad \text{and} \quad y(v) = \frac{cns(q)}{\gcd(prd(q)x(u), cns(q))} \cdot y(u). \tag{4.1}$$

Equation (4.1) can be used to derive the execution rate of any consumer in terms of its producers in a chain of nodes. For example, given $\psi(q) = (u, v)$ for queue q and an execution rate of $R(u) = (x(u) = 1, y(u) = y)$ for node u in Figure 1 on page 5, the execution rate of the consumer node v is derived as follows:

$$\begin{aligned} R(v) = (x(v), y(v)) &= \left(\frac{prd(q) \cdot x(u)}{\gcd(prd(q) \cdot x(u), cns(q))}, \frac{cns(q) \cdot y(u)}{\gcd(prd(q) \cdot x(u), cns(q))}\right) \\ &= \left(\frac{4 \cdot 1}{\gcd(4 \cdot 1, 3)}, \frac{3 \cdot y}{\gcd(4 \cdot 1, 3)}\right) \\ &= \left(\frac{4}{\gcd(4, 3)}, \frac{3 \cdot y}{\gcd(4, 3)}\right) = \left(\frac{4}{1}, \frac{3y}{1}\right) = (4, 3y) \quad \Rightarrow \quad \begin{cases} x(v) = 4 \\ y(v) = 3y \end{cases} \end{aligned}$$

Now consider a general PGM graph in which a node has multiple input queues, such as the graph in Figure 2. Node w is a consumer of data produced by both u and v; $\psi(\alpha) = (u, w)$ and $\psi(\beta) = (v, w)$ define two producer/consumer pairs. We use the notation $R_u(w) = (x_u(w), x_u(w))$ to represent the execution rate of w with respect to u. With $R(u) = (3, 4)$ (the execution rate of u), $R_u(w)$ is derived using (4.1) as follows:

$$\begin{aligned} R_u(w) = (x_u(w), y_u(w)) &= \left(\frac{prd(\alpha) \cdot x(u)}{\gcd(prd(\alpha) \cdot x(u), cns(\alpha))}, \frac{cns(\alpha) \cdot y(u)}{\gcd(prd(\alpha) \cdot x(u), cns(\alpha))}\right) \\ &= \left(\frac{2 \cdot 3}{\gcd(2 \cdot 3, 3)}, \frac{3 \cdot 4}{\gcd(2 \cdot 3, 3)}\right) \\ &= \left(\frac{6}{\gcd(6, 3)}, \frac{12}{\gcd(6, 3)}\right) = \left(\frac{6}{3}, \frac{12}{3}\right) = (2, 4) \quad \Rightarrow \quad \begin{cases} x_u(w) = 2 \\ y_u(w) = 4 \end{cases} \end{aligned}$$

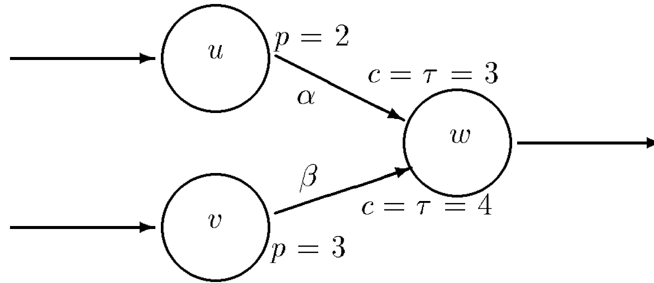

Figure 2: Node w has two input queues: α and β (which implies $\delta^-(w) = 2$). $\psi(\alpha) = (u, w)$ and $\psi(\beta) = (v, w)$. The dataflow attributes for α are $prd(\alpha) = 2$, $cns(\alpha) = 3$, and $thr(\alpha) = 3$. The dataflow attributes for β are $prd(\beta) = 3$, $cns(\beta) = 4$, and $thr(\beta) = 4$.

Likewise with $R(v) = (2,3)$, $R_v(w)$ (the execution rate w with respect to v) is:

$$R_v(w) = (x_v(w), y_v(w)) = \left(\frac{prd(\beta)\cdot x(v)}{\gcd(prd(\beta)\cdot x(v), cns(\beta))}, \frac{cns(\beta)\cdot y(v)}{\gcd(prd(\beta)\cdot x(v), cns(\beta))}\right)$$
$$= \left(\frac{3\cdot 2}{\gcd(3\cdot 2, 4)}, \frac{4\cdot 3}{\gcd(3\cdot 2, 4)}\right)$$
$$= \left(\frac{6}{\gcd(6,4)}, \frac{12}{\gcd(6,4)}\right) = \left(\frac{6}{2}, \frac{12}{2}\right) = (3,6) \quad \Rightarrow \quad \begin{cases} x_v(w) = 3 \\ y_v(w) = 6 \end{cases}$$

Since w can only execute when *both* α and β are over threshold, neither $R_u(w)$ nor $R_v(w)$ satisfies the definition of a valid execution rate for w. Observe that although $R_u(w) \neq R_v(w)$, we do have $\frac{x_u(w)}{y_u(w)} = \frac{x_v(w)}{y_v(w)}$. Lemma 4.3 states that without this equality, it would be impossible to schedule a valid execution of the graph.

Lemma 4.3. *Using* (4.1) *to evaluate the execution rate of node w with respect to its adjacent predecessors in the digraph, if a valid graph execution is possible using finite memory for buffering then $\frac{x_u(w)}{y_u(w)} = \frac{x_v(w)}{y_v(w)}$ for all u, v for which there exists queues α, β such that $\psi(\alpha) = (u, w)$ and $\psi(\beta) = (v, w)$.*

Proof: (We prove the contrapositive.) Suppose $\frac{x_u(w)}{y_u(w)} \neq \frac{x_v(w)}{y_v(w)}$. Observe that in an interval of length $y_u(w)\cdot y_v(w)$ time units u would produce enough data for w to execute $x_u(w)\cdot y_v(w)$ times if u and w were producer/consumer pairs in a chain, and v would produce enough data for w to execute $x_v(w)\cdot y_u(w)$ times if v and w were producer/consumer pairs in a chain.

Since $\frac{x_u(w)}{y_u(w)} \neq \frac{x_v(w)}{y_v(w)} \implies x_u(w)\cdot y_v(w) \neq x_v(w)\cdot y_u(w)$ and w can only execute when all of the input queues are over threshold, w can execute at most $\min(x_u(w)\cdot y_v(w), x_v(w)\cdot x_u(w))$ times in any interval of length $y_u(w)\cdot y_v(w)$. Therefore data will back up on the queue that requires $\max(x_u(w)\cdot y_v(w), x_v(w)\cdot x_u(w))$ executions of w in the same interval. Since we have only finite memory available, data will eventually be lost and a valid graph execution is impossible. Therefore if valid graph execution is possible, $\frac{x_u(w)}{y_u(w)} = \frac{x_v(w)}{y_v(w)}$ for all u, v for which there exists queues α, β such that $\psi(\alpha) = (u, w)$ and $\psi(\beta) = (v, w)$. □

Theorem 4.4. *Assuming a valid PGM digraph $G = (V, E, \psi)$, the strong synchrony hypothesis, $\delta^-(v) \geq 1$, and no tokens on the input queues to node v prior to the beginning of graph execution: the execution rate of node v is $R(v) = (x(v), y(v))$ where*

$$y(v) = \operatorname{lcm}\{\frac{cns(q)}{\gcd(prd(q)\cdot x(u), cns(q))}\cdot y(u) \mid \psi(q) = (u,v)\} \text{ and}$$
$$x(v) = y(v)\cdot\left(\frac{prd(q)\cdot x(u)}{cns(q)\cdot y(u)}\right), \ \forall q, u \text{ such that } \psi(q) = (u,v). \tag{4.2}$$

Proof: The proof is constructed in two parts. In the first part we prove that for a (w, v) pair with an adjoining queue α (4.2) derives a valid rate for v with respect to w. In the second part we show that the rate derived using the $\psi(\alpha) = (w, v)$ pair is the same for all $u \in V$ and $q \in E$ such that $\psi(q) = (u, v)$.

If v were in a chain such that w and v were a producer/consumer pair $\psi(\alpha) = (w, v)$, then (4.1) derives the execution rate of v with respect to w as $R_w(v) = (x_w(v), y_w(v))$ where

$$x_w(v) = \frac{prd(\alpha)}{\gcd(prd(\alpha)\cdot x(w), cns(\alpha))}\cdot x(w) \quad\text{and}\quad y_w(v) = \frac{cns(\alpha)}{\gcd(prd(\alpha)\cdot x(w), cns(\alpha))}\cdot y(w).$$

Let $R(v) = (x(v), y(v)) = (m_w \cdot x_w(v), m_w \cdot y_w(v))$ where

$$m_w = \frac{\mathrm{lcm}\{\frac{cns(q)}{\gcd(prd(q)\cdot x(u), cns(q))} \cdot y(u) \mid \psi(q) = (u,v)\}}{y_w(v)}.$$

By Theorem 4.2 and Corrollary 4.1 if $R_w(v)$ is a valid rate specification for v with respect to w, then $R(v)$ is as well. We now reduce this expression to the form of (4.2):

$$\begin{aligned}
y(v) = m_w \cdot y_w(v) &= \frac{\mathrm{lcm}\{\frac{cns(q)}{\gcd(prd(q)\cdot x(u), cns(q))} \cdot y(u) \mid \psi(q) = (u,v)\}}{y_w(v)} \cdot y_w(v) \\
&= \mathrm{lcm}\{\frac{cns(q)}{\gcd(prd(q)\cdot x(u), cns(q))} \cdot y(u) \mid \psi(q) = (u,v)\} \\
\implies x(v) = m_w \cdot x_w(v) &= \frac{\mathrm{lcm}\{\frac{cns(q)}{\gcd(prd(q)\cdot x(u), cns(q))} \cdot y(u) \mid \psi(q) = (u,v)\}}{y_w(v)} \cdot x_w(v) \\
&= \frac{y(v)}{y_w(v)} \cdot x_w(v) \\
&= \left(\frac{y(v)}{\frac{cns(\alpha)}{\gcd(prd(\alpha)\cdot x(w), cns(\alpha))} \cdot y(w)}\right) \cdot \left(\frac{prd(\alpha)}{\gcd(prd(\alpha)\cdot x(w), cns(\alpha))} \cdot x(w)\right) \\
&= \frac{y(v)}{cns(\alpha)\cdot y(w)} \cdot prd(\alpha) \cdot x(w) \\
&= y(v) \cdot \left(\frac{prd(\alpha)\cdot x(w)}{cns(\alpha)\cdot y(w)}\right)
\end{aligned}$$

We now show by contradiction that $\forall u \in V, q \in E$ such that $\psi(q) = (u,v)$ used to derive $R(v)$, if the graph is valid, $R(v) = (x(v), y(v))$ is a valid rate specification for v. Suppose $\exists i, j \in V \wedge \exists \alpha, \beta \in E$ such that $\psi(\alpha) = (i,v)$, $\psi(\beta) = (j,v)$, and $R(v) = (m_i \cdot x_i(v), m_i \cdot y_i(v)) \neq (m_j \cdot x_j(v), m_j \cdot y_j(v))$ where m_i and m_j are defined as above. Then $\frac{x_i(v)}{y_i(v)} \neq \frac{x_j(v)}{y_j(v)}$ since $\frac{m_k \cdot x_k(v)}{m_k \cdot y_k(v)} = \frac{x_k(v)}{y_k(v)}$. But if $\frac{x_i(v)}{y_i(v)} \neq \frac{x_j(v)}{y_j(v)}$ then, by Lemma 4.3, a valid graph execution is not possible with finite memory. Therefore if the graph is valid, $R(v) = (x(v), y(v))$ derived with (4.2) using the $\psi(\alpha) = (w,v)$ pair is the same for all $u \in V$ and $q \in E$ such that $\psi(q) = (u,v)$. □

It is reassuring (and necessary for correctness since (4.1) and (4.2) define minimal intervals in which a valid rate specification exists) that (4.2) reduces to (4.1) when a node has only one input queue:

$$\begin{aligned}
y(v) &= \mathrm{lcm}\{\frac{cns(q)}{\gcd(prd(q)\cdot x(u), cns(q))} \cdot y(u)\} = \frac{cns(q)}{\gcd(prd(q)\cdot x(u), cns(q))} \cdot y(u) \\
\implies x(v) &= y(v) \cdot \left(\frac{prd(q)\cdot x(u)}{cns(q)\cdot y(u)}\right) = \frac{cns(q)}{\gcd(prd(q)\cdot x(u), cns(q))} \cdot y(u) \cdot \left(\frac{prd(q)\cdot x(u)}{cns(q)\cdot y(u)}\right) \\
&= \frac{prd(q)\cdot x(u)}{\gcd(prd(q)\cdot x(u), cns(q))} = \frac{prd(q)}{\gcd(prd(q)\cdot x(u), cns(q))} \cdot x(u)
\end{aligned}$$

Therefore, (4.2) = (4.1) when a node has only one input queue and either expression can be used to calculate the consumer node's execution rate.

We now return to the problem of finding the execution rate of node w in Figure 2 on page 7. Recall node w is a consumer of data produced by both u and v, $\psi(\alpha) = (u,w)$ and $\psi(\beta) = (v,w)$ with dataflow attributes $prd(\alpha) = 2$, $cns(\alpha) = 3$, $thr(\alpha) = 3$, $prd(\beta) = 3$, $cns(\beta) = 4$, and $thr(\beta) = 4$. The execution rates

for nodes u and v were defined as $R(u) = (3, 4)$ and $R(v) = (2, 3)$. Since $\frac{x_u(w)}{y_u(w)} = \frac{x_v(w)}{y_v(w)}$, we can use (4.2) to derive the execution rate of w:

$$\begin{aligned}
y(w) &= \text{lcm}\{\frac{cns(\alpha)}{\gcd(prd(\alpha) \cdot x(u), cns(\alpha))} \cdot y(u), \frac{cns(\beta)}{\gcd(prd(\beta) \cdot x(v), cns(\beta))} \cdot y(v)\} \\
&= \text{lcm}\{\frac{3}{\gcd(2 \cdot 3, 3)} \cdot 4, \frac{4}{\gcd(3 \cdot 2, 4)} \cdot 3\} \\
&= \text{lcm}\{\frac{3}{\gcd(6, 3)} \cdot 4, \frac{4}{\gcd(6, 4)} \cdot 3\} = \text{lcm}\{\frac{3}{3} \cdot 4, \frac{4}{2} \cdot 3\} = \text{lcm}\{4, 6\} = 12 \\
\implies x(w) &= y(w) \cdot \left(\frac{prd(\alpha) \cdot x(u)}{cns(\alpha) \cdot y(u)}\right) = 12 \cdot \left(\frac{2 \cdot 3}{3 \cdot 4}\right) = 12 \cdot \left(\frac{6}{12}\right) = 6 \\
&= y(w) \cdot \left(\frac{prd(\beta) \cdot x(v)}{cns(\beta) \cdot y(v)}\right) = 12 \cdot \left(\frac{3 \cdot 2}{4 \cdot 3}\right) = 12 \cdot \left(\frac{6}{12}\right) = 6
\end{aligned}$$

Therefore $R(w) = (x(w), y(w)) = (6, 12)$.

4.2 RBE Task Model

Once the node execution rates have been established, the nodes can be mapped to a real-time task model. Unfortunately 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. If we are willing to accept additional latency, the graph can be mapped to a periodic execution model by delaying the first release of every node until enough data has accumulated on the input queue to ensure data availability. Our goal, however, is to be able to manage both buffer requirements and latency. Since adding latency to the execution usually increases the buffer requirements of the graph, our synthesis method should let a successor node be released as soon as it is eligible for execution in accordance with the processing graph methodology. The jitter caused by such a release mechanism during execution makes it difficult to efficiently use the canonical sporadic tasking model. For this reason, we use the Rate Based Execution (RBE) paradigm [13] developed by Jeffay to model node execution in a graph implementation. This section provides a brief overview of the task model.

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 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.

A RBE task set is feasible if there exists a preemptive 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

$$D_i(j) = \begin{cases} t_{i,j} + d_i & \text{if } 1 \leq j \leq x_i \\ \max(t_{i,j} + d_i, D_i(j - x_i) + y_i) & \text{if } j > x_i \end{cases} \tag{4.3}$$

The RBE task model makes no assumptions regarding when a task will be released, but the second line of the deadline assignment function (4.3) 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 execution jitter from creating more process demand in an interval by a task than that which is specified by the rate parameters.

4.3 From Processing Graph to Real-Time System

The next step in the synthesis method is to map the nodes of a processing graph to tasks in the RBE model and verify schedulability of the resulting task set. An affirmative result from the scheduling condition means that the processor has enough capacity to execute the graph with the given parameters such that latency and buffer requirements can be guaranteed. This section explains how each node in the graph is mapped to a task in the RBE task set. To complete the synthesis method, we present a sufficient but not necessary condition for the resulting task set.

To evaluate the schedulability, latency and buffer requirements of a processing graph on a uniprocessor, each node u in the graph is associated with the four tuple $(x(u), y(u), d(u), e(u))$. The parameters $x(u)$ and $y(u)$ are derived using (4.2). The parameter $e(u)$ is the worst case execution time for node u. The only free parameter (after the processing graph has been set) is the relative deadline parameter $d(u)$. The value chosen for $d(u)$ influences processor demand, latency, and buffer requirements. Execution time, produce, threshold, consume, and deadline values all affect the real-time properties of schedulability, latency and buffer requirements, and there are an exponential number of trade-offs to be made. Optimizing any one of processor utilization, latency, or buffer requirements is beyond the scope of this paper. The synthesis method outlined here provides a framework for evaluating these real-time properties, and leaves open the problem of optimization and partitioning of a processing graph in a distributed system.

Since $d(u)$ affects processor demand, latency and buffer requirements, a good starting point for the selection of $d(u)$ is one such that it is greater than or equal to the deadline of its predecessor node and less than or equal to $y(u)$. As shown in [11], when the deadline for each node is greater than or equal to its predecessor's deadline, release time inheritance can be used to minimize latency. Release time inheritance is achieved by assigning a logical release time to a node u (at the time of its actual release) that is equal to the logical release time of the node that enabled u during graph execution. The deadline assignment function, (4.3), then uses the logical release times rather than the actual release times.

After we have associated each node u in the graph with the four tuple $(x(u), y(u), d(u), e(u))$, the graph is mapped to a task system $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$. The mapping is performed by first assigning each node labeled u an integer i. Then node u in the graph G is mapped to the periodic task $T_i = (x_i, y_i, d_i, e_i)$ in $\mathcal{T}$ by setting $x_i = x(u)$, $y_i = y(u)$, $d_i = d(u)$, and $e_i = e(u)$.

We proved the following feasibility condition for an RBE task set in [9].

Lemma 4.5. *Let $\mathcal{T} = \{(x_1, y_1, d_1, e_1), \ldots (x_n, y_n, d_n, e_n)\}$ be a set of tasks. $\mathcal{T}$ will be feasible if and only if*

$$\forall L > 0, \quad L \geq \sum_{i=1}^{n} f\left(\frac{L - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i \tag{4.4}$$

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

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 (4.4) can be evaluated efficiently (in pseudo-polynomial time) using techniques developed in [3].

In [9] we established sufficiency of (4.4) by showing that the preemptive EDF scheduling algorithm can schedule releases of the tasks in $\mathcal{T}$ without a task missing a deadline if the task set satisfies (4.4). For a PGM graph, (4.4) becomes a sufficient but not necessary condition for preemptive EDF scheduling as long as nodes execute only when their input queues are over threshold (i.e., the tasks are released when all of the node's input queues are over threshold — thereby ensuring precedence constraints are met). (4.4) is not a necessary condition since it assumes that all $x(u)$ releases of node u may occur at the beginning of an interval of length $y(u)$. For some nodes, such as node v in Figure 1 on page 5, this is not possible. Henceforth we refer to the preemptive EDF scheduler using deadline assignment function (4.3) as the RBE-EDF scheduler.

From Lemma 4.5 and the preceding discussion, we obtain:

Theorem 4.6. *Let* $\mathcal{T} = \{(x_1, y_1, d_1, e_1), \ldots (x_n, y_n, d_n, e_n)\}$ *be a set of tasks such that for the mapping* $u \in V \rightarrow i$: $(x_i, y_i, d_i, e_i) = (x(u), y(u), d(u), e(u))$. *The processing graph* $G = (V, E, \psi)$ *is schedulable with the RBE-EDF scheduler if* (4.4) *holds for* $\mathcal{T}$.

An affirmative result from (4.4) means that the RBE-EDF scheduler can be used to execute the graph without missing a deadline, and we can bound the buffer requirements of each queue and the entire graph.

5 Buffer Requirements

We now address the second major concern for many acoustic signal processing applications: sizing the system to fit memory requirements. Once again, this paper does not offer an algorithm for minimizing the buffer requirements of an application, rather it provides a framework for doing so by presenting a theorem that bounds the buffer requirements for common applications. In general, it is difficult to get tight buffer bounds for graphs implemented with the RBE task model. There are, however, common special cases that lead to near optimal buffer bounds for individual queues. In this section, we present function $\mathit{Buf}(q)$, which derives an upper bound on the number of tokens queue q will ever contain when the queues are initialized with $(\mathit{thr}(q) - \mathit{cns}(q))$ tokens[2].

The advantage of the RBE model is that it provides great flexibility in describing when nodes will execute. We can say 5 executions will occur in 10 time units without being required to say that 2 executions occur at time 5 and 3 more executions occur at time 10. This flexibility in scheduling comes at a price; it makes it difficult to derive tight bounds for the buffer requirements of a queue. To make the task even more difficult, the definition of *And* nodes are such that one input queue may be over threshold long before another input queue. In the general case, the buffer bounds that we can derive are much too loose to be useful (though they are valid upper bounds). Fortunately, many signal processing applications possess dataflow characteristics that we can exploit to get relatively tight buffer bounds.

Due to the nature of signal processing applications, it is quite common for the produce or consume value to be a multiple of the other. Hence the producer and consumer nodes usually execute at the same rate, or one fires an integral number of times in the rate interval of the other. This property lets us find relatively

[2]See [10] for a complete discussion on buffer bounds for graphs implemented with the RBE model.

tight buffer bounds even when we don't know exactly when the node will fire. For example, in the mobile satellite receiver application described in §6, if nodes H and I each have the same deadline and release time inheritance is used, the queue joining them will never contain more than 11 tokens. If node J also has the same deadline, its input queue will never contain more than 10 tokens.

Many signal processing functions use the concept of a "sliding window". The last portion of data used in one execution of the node is used as the first portion in the next execution. Imagine laying a window over an array of data so that only 1024 data points are visible, performing a calculation with these 1024 points, and then moving the window 768 positions to the right so that 256 old values and 768 new values are visible for the next calculation. This effect is achieved by setting the threshold on a queue to 1024 and the consume amount to 768. It is common practice to initialize such queues with $(thr(q) - cns(q))$ tokens so that the amount of initialized data is equal to the overlap. When the queues are initialized this way, or equivalently when the threshold equals the consume amount, we are able to provide a fairly tight bound on the buffer requirements of the queue.

In addition to the execution rate of each node in a producer/consumer pair, the logical release time of the first execution of each node is needed to bound the joining queue's buffer requirements. Given $\psi(q) = (u, v)$, the first release of node v, $T_1(v)$, is derived as follows. Let $\mathcal{I}(v)$ be a subset of the periodic input nodes $\mathcal{I}$ from which there exists a path from $i \in \mathcal{I}$ to the vertex v, and $\mathcal{F}(i, v) = \max(\{F(i, v)\})$ when (5.1) is evaluated over all paths $\{i \rightsquigarrow v\}$. It was shown in [10] that $T_1(v) = s(v)$ where $s(v)$ is defined by (5.2). (For a full explanation of these equations and proof that they derive the first logical release time for node v, see [10].)

$$F(i,v) = \begin{cases} \max\left(0, \left\lceil \frac{thr(q) - size(q)}{prd(q)} \right\rceil\right) & \text{if } \psi(q) = (i, v) \\ \max\left(0, \left\lceil \frac{(F(u,v)-1)\cdot cns(q) + thr(q) - size(q)}{prd(q)} \right\rceil\right) & \text{if } \psi(q) = (i, u) \wedge u \neq v \wedge F(u,v) > 0 \\ 0 & \text{if } \psi(q) = (i, u) \wedge u \neq v \wedge F(u,v) = 0 \end{cases} \tag{5.1}$$

$$s(v) = \max(0, \{(\mathcal{F}(i, v) - 1) \cdot y(i) \mid i \in \mathcal{I}(v)\}) \tag{5.2}$$

The first logical release time of node u is also derived using (5.2). In the rest of this paper we use $s(u)$ and $s(v)$ rather than $T_1(u)$ and $T_1(v)$ to denote the times associated with the first release of these nodes since we are only concerned with these particular release times.

For the common cases in applications when all queues in the graph are initialized with $(thr(q) - cns(q))$ tokens and $\gcd(cns(q), x(u) \cdot prd(q)) = \min(cns(q), x(u) \cdot prd(q))$ for each queue, (5.3) provides a tight bound on the buffer space required by queue q if the graph is executed using the RBE-EDF scheduler.

Theorem 5.1. *Assuming a valid PGM digraph $G = (V, E, \psi)$ and each queue in the path(s) from a periodic source to node v is initialized with $(thr(q) - cns(q))$ tokens prior to the beginning of graph execution using*

the RBE-EDF scheduling algorithm with release time inheritance, $\forall q, u$ *such that* $\psi(q) = (u, v)$*, the buffer requirements for queue* q *is* $Buf(q)$ *where*

$$Buf(q) \leq \left\lceil \frac{\max(y(v), s(v) + d(v) - s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q) + (thr(q) - cns(q)). \tag{5.3}$$

Proof: Since each queue in the path to node v is initialized with $(thr(q) - cns(q))$ tokens (which is 0 when $thr(q) = cns(q)$), the execution of node v at time $s(v)$ marks the first execution interval of length $y(v)$ for node v. Since the theorem assumes a valid graph execution, the same amount of data produced by the predecessors of node v in $[t,\ t + y(v)), \forall t \geq s(v)$, will be consumed by node v before time $t + y(v) + d(v)$. Therefore, the maximum data queue q can contain occurs (at least once) at time $t' = \max(y(v), s(v) + d(v) - s(u))$ when node u produces the maximum possible before time t'. This amount is represented by (5.3). Clearly, since queue q is initialized with $(thr(q) - cns(q))$ tokens, q requires space for at least $(thr(q) - cns(q))$ tokens.

The proof proceeds by cases.

Case 1: $s(v) + d(v) - s(u) > y(v)$. If $s(v) + d(v) - s(u) > y(v)$ then

$$\begin{aligned}\left\lceil \frac{\max(y(v), s(v) + d(v) - s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q) + (thr(q) - cns(q)) &\geq Buf(q) \\ &> \frac{y(v)}{y(u)} \cdot x(u) \cdot prd(q) + (thr(q) - cns(q)) \\ &= x(v) \cdot cns(q) + (thr(q) - cns(q))\end{aligned}$$

since, by Theorem 4.2 $x(v) = \frac{y(v)}{y(u)} \cdot \frac{x(u) \cdot prd(q)}{cns(q)}$. Assume the worst case and all

$$\left\lceil \frac{\max(y(v), s(v) + d(v) - s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q)$$

tokens arrive at once. Then $x(v)$ deadlines expire at time $s(v) + d(v)$, which will consume $x(v) \cdot cns(q)$ tokens from queue q, leaving

$$\left\lceil \frac{\max(y(v), s(v) + d(v) - s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q) + (thr(q) - cns(q)) - (x(v) \cdot cns(q))$$

tokens on queue q. After the first deadline of node v, $x(v)$ deadlines of node v will expire within any $y(v)$ time units when the source is periodic and release time inheritance is used. Again in the worst case, at most another $(x(v) \cdot cns(q))$ tokens will be appended to queue q before time $(s(v) + d(v) + y(v))$ since node u can produce at most

$$\frac{y(v)}{y(u)} \cdot x(u) \cdot prd(q) = x(v) \cdot cns(q)$$

tokens in any interval of $y(v)$. Therefore, queue q can contain no more than

$$\begin{aligned}&\left\lceil \frac{\max(y(v), s(v) + d(v) - s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q) + (thr(q) - cns(q)) - (x(v) \cdot cns(q)) + (x(v) \cdot cns(q)) \\ &\qquad = \left\lceil \frac{\max(y(v), s(v) + d(v) - s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q) + (thr(q) - cns(q))\end{aligned}$$

tokens before node v must consume another $(x(v) \cdot cns(q))$ tokens. Therefore queue q will never contain more than

$$\left\lceil \frac{\max(y(v), s(v)+d(v)-s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q) + (thr(q) - cns(q)) - (k \cdot x(v) \cdot cns(q)) + (k \cdot x(v) \cdot cns(q))$$

$$= \left\lceil \frac{\max(y(v), s(v)+d(v)-s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q) + (thr(q) - cns(q))$$

tokens at time $(s(v) + d(v) + k \cdot y(v))$ for all $k \geq 0$ and (5.3) is an upper bound on the size of queue q if $s(v) + d(v) - s(u) > y(v)$.

Case 2: $s(v) + d(v) - s(u) \leq y(v)$. If $s(v) + d(v) - s(u) \leq y(v)$ then

$$\left\lceil \frac{\max(y(v), s(v)+d(v)-s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q) + (thr(q) - cns(q)) = \frac{y(v)}{y(u)} \cdot x(u) \cdot prd(q) + (thr(q) - cns(q))$$

$$= x(v) \cdot cns(q) + (thr(q) - cns(q)).$$

Assume the worst case and all $\left\lceil \frac{\max(y(v), s(v)+d(v)-s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q)$ tokens arrive at once. Then $x(v)$ deadlines expire at time $s(v) + d(v)$, which will consume $x(v) \cdot cns(q)$ tokens from queue q, leaving

$$\left\lceil \frac{\max(y(v), s(v)+d(v)-s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q) + (thr(q) - cns(q)) - (x(v) \cdot cns(q)) = 0$$

tokens on queue q. After the first deadline of node v, $x(v)$ deadlines of node v will expire within any $y(v)$ time units when the source is periodic and release time inheritance is used. In the worst case, at most another $(x(v) \cdot cns(q))$ tokens will be appended to queue q before time $(s(v) + d(v) + y(v))$ since node u can produce at most $\frac{y(v)}{y(u)} \cdot x(u) \cdot prd(q) = x(v) \cdot cns(q)$ tokens in any interval of $y(v)$. Therefore, queue q can contain no more than

$$\left\lceil \frac{\max(y(v), s(v)+d(v)-s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q) + (thr(q) - cns(q)) - (x(v) \cdot cns(q)) + (x(v) \cdot cns(q))$$

$$= \left\lceil \frac{\max(y(v), s(v)+d(v)-s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q) + (thr(q) - cns(q)) = x(v) \cdot cns(q) + (thr(q) - cns(q))$$

tokens before node v must consume another $(x(v) \cdot cns(q))$ tokens. Therefore queue q will never contain more than

$$\left\lceil \frac{\max(y(v), s(v)+d(v)-s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q) + (thr(q) - cns(q)) - (k \cdot x(v) \cdot cns(q)) + (k \cdot x(v) \cdot cns(q))$$

$$= \left\lceil \frac{\max(y(v), s(v)+d(v)-s(u))}{y(u)} \right\rceil \cdot x(u) \cdot prd(q) + (thr(q) - cns(q)) = x(v) \cdot cns(q) + (thr(q) - cns(q))$$

tokens at time $(s(v) + d(v) + k \cdot y(v))$ for all $k \geq 0$ and (5.3) is an upper bound on the size of queue q. □

In practice, we have found (5.3) to be applicable to the majority of signal processing applications we have analyzed. More importantly Theorem 5.1 provides guidance to signal processing engineers developing processing graphs by quantifying the impact of their choices for thresholds, produce, and consume values as well as choosing the amount of initialized data. The case study in the next section provides examples of applying Theorems 4.4 and 5.1 to a real application.

6 Case Study: INMARSAT Mobile Satellite Receiver Application

This section provides an evaluation of our synthesis method by applying our techniques to an International Maritime Satellite (INMARSAT) mobile satellite receiver application. To fully appreciate the efficiency of on-line scheduling, we compare the buffer requirements of a statically scheduled implementation of the application with our dynamic scheduling approach. We begin with a brief introduction to INMARSAT and the mobile satellite receiver application.

The INMARSAT system has been offering mobile satellite service since 1982. It is a global satellite constellation of 7 geostationary satellites providing communications in the L-band frequencies. The INMARSAT-B mobile terminal provides digital telecommunications supporting facsimile and data transmissions at the standard rate of 9.6 kbps and an optional high speed data rate of 64 kbps. The 64 kpbs channel can be multiplexed to offer several simultaneous voice and data lines. The high speed data option of the INMARSAT-B mobile terminal is also used to provide video teleconferencing and compressed or delayed video transmission services to remote locations on land or at sea.

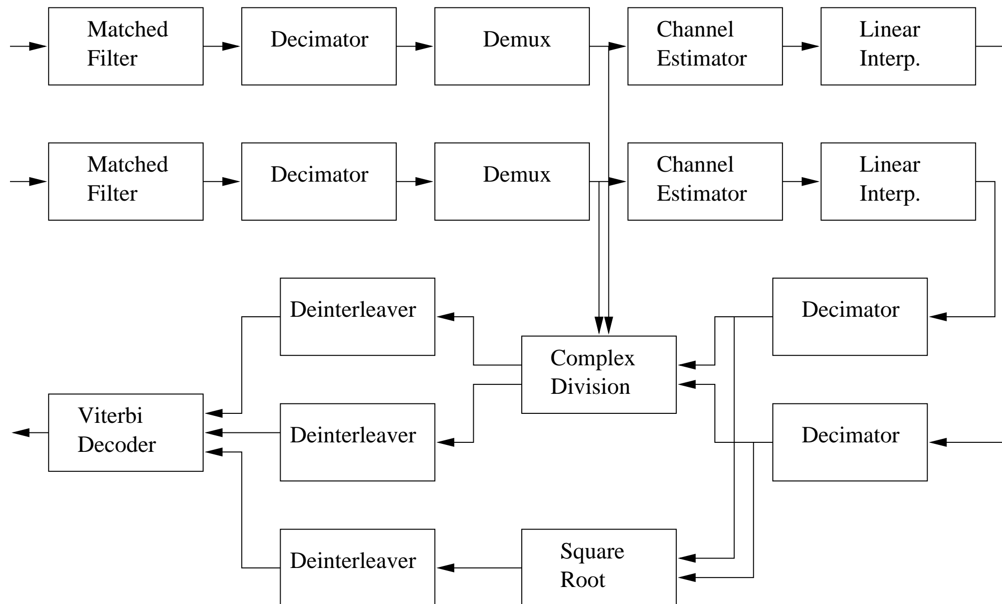

Figure 3: Block diagram of the INMARSAT mobile satellite receiver.

Figure 3 is a block diagram of the digital signal processing performed by the satellite receiver portion of an INMARSAT mobile terminal [25]. The corresponding processing graph for this application is shown in Figure 4 [21] on page 17. The two unlabeled circles with single output queues represent the input devices receiving the satellite signal. The other unlabeled circle represents the terminal accepting the processed signal. For this application, each queue's threshold is equal to its consume value and the graph can be represented as either a SDF graph or a PGM graph — the two models are identical for this graph. As shown in the previous section, we can get a near optimal bound on the buffer requirements of graph edges when all of the queues in the graph have their threshold values equal to their consume values. To reduce clutter

in the figure, we have only labeled the non-unity dataflow attributes: produce values are located at the tail of the queue and consume values are at the head of the queue.

Figure 4: PGM application graph for the INMARSAT mobile satellite receiver.

6.1 Node Execution Rates

Most signal processing applications do not have relatively prime produce and consume dataflow attributes as the examples in §4 do (unless one of the values is 1). They do, however, typically have node execution rate changes throughout the graph. Execution rates usually decrease near the source nodes where data decimation occurs, and rate increases occur when data is de-multiplexed or interpolated to increase frequency resolution. For example, let the period of each of the two input nodes to the INMARSAT mobile satellite receiver application graph of Figure 4 be y. Then the execution rate of node A (D) is $(1, y)$ and the execution rate of node B (E) is

$$R(B) = \begin{cases} y(B) = \mathrm{lcm}(\frac{4}{\gcd(1 \cdot 1, 4)} \cdot y) = 4y \\ x(B) = 4y \cdot \left(\frac{1 \cdot 1}{4 \cdot y}\right) = 1 \end{cases} \quad \Longrightarrow \quad (1, 4y)$$

An example where de-multiplexing and linear interpolation/data decimation paths merge is the execution rate of node P, which has four input queues. De-multiplexing nodes F and C execute at the rate $(1, 44y)$, and decimator nodes N and J each have an execution rate of $(10, 44y)$. Therefore the execution rate of P is

$$R(P) = \begin{cases} y(P) = \mathrm{lcm}\{\frac{1}{\gcd(10 \cdot 1, 1)} \cdot 44y, \frac{1}{\gcd(10 \cdot 1, 1)} \cdot 44y, \frac{1}{\gcd(1 \cdot 10, 1)} \cdot 44y, \frac{1}{\gcd(1 \cdot 10, 1)} \cdot 44y\} = 44y \\ x(P) = 44y \cdot \left(\frac{10 \cdot 1}{1 \cdot 44y}\right) = 10 \end{cases}$$
$$= (10, 44y)$$

The execution rates for rest of the nodes in the application are listed in Table 1 with the relative deadline parameters selected for the Rate Based Execution (RBE) task model used to implement the graph.

Node	($x(u)$,$y(u)$,$d(u)$)	$s(u)$
A	$(1, y, y)$	0
B	$(1, 4y, y)$	$3y$
C	$(1, 44y, 4y)$	$43y$
D	$(1, y, y)$	0
E	$(1, 4y, y)$	$3y$
F	$(1, 44y, 4y)$	$43y$
G	$(1, 44y, 44y)$	$43y$
H	$(1, 44y, 44y)$	$43y$
I	$(1, 44y, 44y)$	$43y$
J	$(10, 44y, 44y)$	$43y$
K	$(1, 44y, 44y)$	$43y$
L	$(1, 44y, 44y)$	$43y$
M	$(1, 44y, 44y)$	$43y$
N	$(10, 44y, 44y)$	$43y$
P	$(10, 44y, 44y)$	$43y$
Q	$(1, 24 \cdot 44y, 44y)$	$(24 \cdot 44y) - y$
R	$(1, 24 \cdot 44y, 44y)$	$(24 \cdot 44y) - y$
S	$(10, 44y, 44y)$	$43y$
T	$(10, 44y, 44y)$	$43y$
U	$(10, 44y, 44y)$	$43y$
V	$(1, 24 \cdot 44y, 44y)$	$(24 \cdot 44y) - y$
W	$(240, 24 \cdot 44y, 24 \cdot 44y)$	$(24 \cdot 44y) - y$

Table 1: The second column shows the RBE parameters for each node, excluding the execution time. The third column shows the start time (i.e., the first release time) for each node assuming release time inheritance.

For this case study, we assume the application is schedulable with our selected parameters. The best way to evaluate the synthesis method is to compare the memory requirements of an RBE implementation of the mobile satellite receiver with an implementation scheduled by a state-of-the-art, static scheduler, which is designed to minimize memory requirements.

6.2 Buffer Requirements

In this section we present the buffer requirements of each queue in the mobile satellite receiver application and compare the total memory requirements of our synthesis of the application with the bounds reported in [21] and [4]. Due to space limitations, we derive the bounds for one queue as an example and refer the reader to Table 2 for the buffer bounds on the remaining queues.

Let h be the label for the queue joining nodes H and I in the application graph of Figure 4 on page 17. Applying (5.3) of §5 to the queue h (using the RBE and start time parameters of Table 1), we get an upper

bound on the buffer requirement for h of

$$\begin{aligned} Buf(h) &\leq \left\lceil \frac{\max(y(I), s(I) + d(I) - s(H))}{y(H)} \right\rceil \cdot x(H) \cdot prd(h) + (thr(h) - cns(h)) \\ &= \left\lceil \frac{\max(44y, 43y + 44y - 43y)}{44y} \right\rceil \cdot 1 \cdot 11 + (11 - 11) \\ &= 11. \end{aligned}$$

The values of $Buf(q)$ for the rest of the queues in the mobile satellite receiver graph (calculated with the RBE parameters of Table 1) are shown in Table 2 with the exception of the queues attached to input or output devices since the buffer space for these queues was ignored in the buffer calculations done by [20, 25, 21] and [4]. The buffer space required for each of those queues in our model is 1 token. When an off-line scheduler is employed, the buffer space required for each of the queues attached to input devices varies from 44 to 4,224 tokens depending on the scheduler.

$\psi(q)$	**Maximum Buffer Space**
(A, B)	4
(B, C)	11
(C, G)	1
(C, P)	10
(D, E)	4
(E, F)	11
(F, K)	1
(F, P)	10
(G, H)	1
(H, I)	11
(I, J)	10
(K, L)	1
(L, M)	11
(M, N)	10
(J, P)	10
(N, P)	10
(J, T)	10
(N, S)	10
(P, Q)	240
(P, R)	240
(Q, W)	240
(R, W)	240
(S, U)	10
(T, U)	10
(U, V)	240
(V, W)	240

Table 2: Maximum buffer space required per queue evaluated using theorem 5.1.

Assuming a unique buffer for each queue, the minimum buffer requirement for the INMARSAT mobile satellite receiver graph is 1,545 tokens — derived by summing $\frac{prd(q) \cdot cns(q)}{\gcd(prd(q), cns(q))} = \max(prd(q), cns(q))$ over

all queues in the graph [4]. If each queue is implemented with a unique buffer in an RBE task set, the total buffer requirement for the application is less than or equal to 1,599 tokens, which is the sum of the values listed in Table 2 plus 3 tokens for the queues attached to external devices. This value is 3.5% greater than the minimum possible buffer requirement of 1,545 tokens.

In comparison, the off-line *Acyclic Pairwise Grouping of Adjacent Nodes* (APGAN) scheduling algorithm of [4][3] achieves the optimal buffer requirement of 1,542 for the queues listed in Table 2. When we add the buffer space required for the queues attached to input devices, however, the buffer requirement of a statically scheduled implementation is at least 1,630 tokens – at least 1.94% greater than the dynamically scheduled implementation.

If a shared buffer implementation is used for the RBE task set, as assumed in the statically scheduled implementation of [21], we can reduce the upper bound of 1,599 to 1,101. Observe that, since the execution uses release time inheritance, nodes Q, R, and V have a combined buffer requirement of at most 960 tokens for their input and output queues rather than the 1,440 assumed when we sum the queue requirements. This is because each of the nodes Q, R, and V will have the same logical release time (relative to each other) whenever they are released and they each have the same deadline parameter. Hence, in the worst case, all three nodes will be eligible for execution at the same time, requiring buffer space for $720 = 3 \cdot 240$ tokens. When the first of these executes, say node Q, it will produce 240 tokens on its output queue and then consume 240 from its input queue. Before the data is consumed, the input and output queues of these three nodes requires space for $960 = 240 + 720$ tokens but after node Q executes the combined buffer space for these queues is back to 720 tokens. This pattern repeats while the remaining two nodes execute except that, when the third node completes, the total buffer space is left at 720 tokens on the output queues and 0 on the input queues for these three nodes. This reduces the upper bound to 1,119. We can reduce this upper bound even further by observing that the input and output queues to node J need space for at most 21 tokens and not 30. All 10 executions of J complete before the next execution of I and the input queue to J will never contain more than 10 tokens. Each time J executes it produces one token on each of its output queues and consumes one token from the input queue. This results in a maximum of 21 tokens existing simultaneously on the input and output queues to node J. The same is true for the input and output queues to node N. Combining these savings with the previous observation, the upper bound for a shared buffer space is reduced to 1,101 tokens.

Whether one considers a unique or shared buffer implementation, our dynamically scheduled execution requires less buffer space than schedules created by state-of-the-art, off-line schedulers. Just as important, run-time complexity is comparable and scheduling state is reduced.

[3]APGAN is the same algorithm as the *Pairwise Grouping of Adjacent Nodes* (PGAN) scheduling algorithm of [5] except the graph is assumed to be acyclic.

7 Summary

In most "real-time" processing graph methodologies, system engineers are unable to analyze the properties of schedulability, latency, and memory requirements. 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 synthesis a predictable real-time application from a general processing graph. When the graph satisfies our schedulability condition for a simple preemptive EDF scheduler, the buffer requirements of each queue can be bound. For many signal processing applications, our bound on the memory requirements for each queue leads to nearly optimal buffer bounds for the entire graph.

Of course, execution time, produce, threshold, consume, and deadline values all affect the real-time properties of schedulability, latency and buffer requirements, and there are an exponential number of trade-offs to be made. The synthesis method outlined in this paper provides a framework for evaluating these real-time properties, and we leave open the problem of optimizing any one of processor utilization, latency, or buffer requirements.

We have also shown that by judiciously selecting deadline parameters for the nodes of an INMARSAT mobile satellite receiver application, a dynamically scheduled execution of the graph actually uses less buffer space than implementations scheduled by state-of-the-art, off-line schedulers. In the past, off-line scheduling has been favored since it requires little overhead (on-line), and as much processor capacity as possible needed to be applied to the signal processing application. Today, as processor speed continues to increase faster than memory densities, memory management has become more critical in acoustic signal processing applications. By using some of the processor capacity for on-line scheduling decisions, we can achieve better latency and execute with less memory than off-line scheduling.

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., Goddard, S., Jeffay, K., "Feasibility Concerns in PGM Graphs with Bounded Buffers," Proceedings of the Third International Conference on Engineering of Complex Computer Systems, pp 130-139, Como, Italy. September, 1997. IEEE Computer Society Press.

[3] Baruah, S., Howell, R., 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.

[4] Bhattacharyya, S.S., Murthy, P.K., Lee, E.A., *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.

[5] Bhattacharyya, S.S., Lee, E.A., "Scheduling Synchronous Dataflow Graphs for Efficient Looping", *Journal of VLSI Signal Processsing*, Vol. 6, No. 3, December 1993.

[6] 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.

[7] Gerber, R., Seongsoo, H., Saksena, M., "Guaranteeing Real-Time Requirements with Resource-Based Calibration of Periodic Processes", *IEEE Transactions on Software Engineering*, 21(7), July 1995.

[8] Kang, D.-I., Gerber, R., Saksena, M., "Performance-Based Design of Distributed Real-Time Systems", *Proc. of IEEE Real-Time Technology and Applications Symposium*, June 1997, pp. 2-13.

[9] 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.

[10] Goddard, S., Jeffay, K. "Analyzing the Real-Time Properties of Processing Graphs Implemented with the Rate Based Execution Model", Technical Report TR98-001, Dept. of Computer Science, University of North Carolina at Chapel Hill, In Submission.

[11] Goddard, S., Jeffay, K. "Analyzing the Real-Time Properties of a Dataflow Execution Paradigm using a Synthetic Aperture Radar Application", *Proc. of IEEE Real-Time Technology and Applications Symposium*, June 1997, pp. 60-71.

[12] 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.

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

[14] Karp, R.M., Miller, R.E., "Properties of a model for parallel computations: Determinacy, termination, queuing", *SIAM J. Appl. Math*, Vol. 14, No. 6, pp. 1390-1411, 1966.

[15] 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.

[16] 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.

[17] 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.

[18] *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.

[19] Ramamritham, K., "Allocation and Scheduling of Precedence-Related Periodic Tasks", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 4, April 1995, pp. 412-420.

[20] Ritz, S., Meyer, H., "Exploring the design space of a DSP-based mobile satellite receiver", *Proc. of ICSPAT 94*, Dallas, TX, October 1994.

[21] Ritz, R., Willems, M., Meyer, H., "Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis", *Proc. of ICASSP 95*, Detroit, MI, May 1995, pp. 133-143.

[22] Sun, J., Liu, J., "Synchronization Protocols in Distributed Real-Time Systems", *Proc. of 16th International Conference on Distributed Computing Systems*, May, 1996.

[23] Sun, J., Liu, J., "Bounding Completion Times of Jobs with Arbitrary Release Times and Variable Execution Times", *Proc. of the IEEE Real-Time Systems Symposium*, Washington, DC, Dec. 1996, pp. 2-12. December, 1996.

[24] Spuri, M., Stankovic, J.A., "How to Integrate Precedence Constraints and Shared Resources in Real-Time Scheduling", *IEEE Transactions on Computers*, Vol. 43, No. 12, December 1994, pp. 1407-1412.

[25] Živojnović, V., Ritz, S., Meyer, H., "High Performance DSP Software Using Data-Flow Graph Transformations", *Proc. of ASILOMAR 94*, Pacific Grove, November 1994.