In: *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, Denver, CO, June 1998, pages 59-70.

Managing Memory Requirements in the Synthesis of Real-Time Systems from Processing Graphs*

Steve Goddard *Kevin Jeffay*
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175
{*goddard, jeffay*}*@cs.unc.edu*

Abstract

In the past, environmental restrictions on size, weight, and power consumption have severely limited both the processing and storage capacity of embedded signal processing systems. Today, however, as increases in processor speed and capabilities continually out-pace increases in memory densities and performance, processor capacity is no longer a major concern for many signal processing applications — memory usage is now the primary concern.

We present techniques for managing the memory requirements of signal processing applications in the synthesis of a real-time uniprocessor system from processing graphs. To demonstrate the effectiveness of our memory management techniques, we compare the memory requirements of a statically scheduled implementation of an INMARSAT (International Maritime Satellite) mobile receiver, with our dynamic scheduling techniques. The case study demonstrates that state-of-the-art, static schedulers use over 300% more memory than our simple, preemptive, EDF scheduler for a large class of signal processing applications.

1. Introduction

Directed graphs consisting of nodes that represent processing functions, and graph edges that depict the flow of data from one node to the next, are a standard design aid in the development of complex digital signal processing systems. When sufficient data arrives, a node executes its function from start to finish in isolation (i.e., without synchronization with other nodes). Such directed graphs are called *Processing Graphs*.

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

Processing graphs provide a natural means of describing signal processing applications, each node represents a mathematical function to be performed on a stream of data that flows on the edges of the graph from source nodes (sensors) to sink nodes (output devices). The processing graph methodology allows one to easily understand the signal processing performed by graphically depicting the structure of the algorithm. An important advantage of the graphical representation is that portions of the signal processing application (subgraphs) can be understood in the absence of the rest of the algorithm.

In the past, environmental restrictions on size, weight, and power consumption have severely limited both the processing and storage capacity of embedded signal processing systems. Today, however, as increases in processor speed and capabilities continually out-pace increases in memory densities and performance, processor capacity is no longer a major concern for many signal processing applications — memory usage is now the primary concern. Thus, although real-time analysis often involves timing analysis to make sure a CPU has the processing capacity to ensure real-time execution, we are using timing analysis to manage memory requirements. There are two aspects to memory analysis: code and data storage requirements. We only consider data space here. Optimizing compilers and efficiently written code can help to minimize code space, but processing graphs also require storage space for intermediate processing results temporarily stored on the graph edges. Once a node completes its execution, it appends data to the edge connecting it to a "downstream" (consumer) node. When sufficient data has accumulated on the input edge to the consumer node, it executes and removes the input data. The space required to hold the intermediate results on all graph edges simultaneously can be quite substantial.

Once the processing graph has been created, the only free variable in controlling the amount of data buffered on an edge is the execution relationship between the producer

In: *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, Denver, CO, June 1998, pages 59-70.

and consumer nodes. The canonical approach to minimizing memory requirements for the graph edges is to use static scheduling. Static node execution schedules are created off-line and then executed on a periodic basis. Numerous static scheduling algorithms have been created to minimize memory requirements [13, 17, 22, 18, 2]. The primary trade-off made by static schedulers is the storage requirement and execution complexity of the schedule vs. the storage requirement of data in the graph edges. Typically, minimal buffer space requires increased state space for the scheduler.

In contrast, our approach is to use a simple, dynamic, on-line scheduler for graph execution. With today's processors, it is possible to use dynamic, on-line scheduling to achieve near optimal memory usage and not be concerned with minimizing the on-line scheduling overhead. Moreover, and somewhat surprisingly, dynamic scheduling often requires less memory than static schedules created by off-line schedulers designed to minimize memory usage.

The primary problem in using dynamic scheduling techniques with processing graphs is building a predictable run-time system so that the buffer space required for graph edges can be bounded and controlled. For this, we appeal to real-time scheduling theory. Typically a processing graph is mapped to a set of tasks according to a model of real-time execution [3, 7, 10, 14, 16, 21, 19, 20]. The schedulability conditions for the model are used to see if the graph "fits" on the processor — i.e., to see if enough processing capacity is available to guarantee real-time execution to all tasks. By real-time execution, we refer to an execution in which the latency and memory requirements are met. For signal processing graphs, latency is the time between when a sensor produces a data token and when the graph outputs the processed signal.

This paper is part of a larger body of work [9] that creates a framework for evaluating and managing processor demand, latency, and memory usage in the synthesis of distributed real-time signal processing systems from the U.S. Navy's coarse-grain *Processing Graph Method* (PGM) [15]. Here, we demonstrate the management of memory requirements in the synthesis of a real-time uniprocessor system from acyclic processing graphs developed with PGM. We show that dynamic, on-line scheduling can achieve near minimal memory requirements.

We have selected PGM as our processing graph model since it is a general and widely used paradigm. Our previous work on the synthesis of real-time uniprocessor systems from PGM was based on simple PGM graphs called chains [1, 8]. In this paper, we extend the analyses developed for chains to handle directed acyclic graphs. In addition, we present new methods for deriving the memory needs of a common class of acyclic PGM graphs executed using the same simple, preemptive, earliest deadline first (EDF) scheduler of [8].

To illustrate these concepts and to quantify the memory savings achievable in a real application, we analyze an existing processing graph for a mobile receiver in a commercial satellite-based communication and navigational system known as INMARSAT (International Maritime Satellite). We compare the memory needs of the mobile satellite receiver application executed with our on-line scheduler to the memory needs of the same application executed using state-of-the-art, off-line, static schedulers presented in [18] and [2].

From the real-time literature, PGM graphs are most closely related to the Logical Application Stream Model (LASM) [3, 4]. Our work improves on the analysis of LASM graphs by not requiring periodic execution of the nodes in the graph. Instead, graph execution is modeled with the Rate-Based Execution (RBE) process model [11] (a generalization of the sporadic process model) to more accurately predict processor demand. The RBE process model allows node execution at an average (but deterministic) rate, which provides a more natural representation of node execution for PGM graphs. Forcing periodic execution of all graph nodes adds latency to the processed signal, but simplifies the analysis of memory requirements.

Restricted PGM graphs can also be represented by the dataflow graph models used in the Software Automation for Real-Time Operations (SARTOR) project [14] and the Real-Time Producer/Consumer (RTP/C) paradigm [10]. Unfortunately, neither of these paradigms correctly model the execution of general PGM graphs. Our goal, as with the SARTOR project, is to demonstrate that we can apply real-time scheduling results to real-life applications. Unfortunately, the techniques developed in [14] cannot be applied here without introducing additional latency since the execution of PGM nodes do not follow the periodic execution model assumed in [14]. Like the RTP/C paradigm, we use the structure of the graph to help specify execution rates of the processes that implement nodes in the graph. However, PGM graphs are capable of supporting much more sophisticated data flow applications than RTP/C.

The rest of the paper is organized as follows. §2 presents a brief overview of the processing graph model PGM. The synthesis of real-time uniprocessor systems from PGM graphs is presented in §3 with a discussion on managing memory requirements in §4. We evaluate our results in §5 with a case study of an INMARSAT mobile receiver application. Our contributions are summarized in §6.

2. Notation and the Processing Graph Method

The notation and terminology of this paper, for the most part, is an amalgamation of the notation and terminology used in [6] and [2]. 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 *E* of *edges*, and an incidence function ψ that associates with each edge of *E* an ordered pair of (not necessarily distinct) vertices of *V*. Consider an edge $e \in E$ and vertices $u, v \in V$ such that $\psi(e) = (u, v)$. We say e joins u to v, or u and v are adjacent. The vertex u is called the tail or source vertex of e and v is the head or sink vertex of edge e. The edge e is an *output edge* of u and an *input edge* of v. The number of input edges to a vertex v is the *indegree* $\delta^-(v)$ of v, and the number of output edges for a vertex v is the *outdegree* $\delta^+(v)$ of v. A vertex v with $\delta^-(v) = 0$ is an *input node*. The set $\mathcal{I} = \{v \mid v \in V \wedge \delta^-(v) = 0\}$ denotes the set of all input nodes. A vertex v with $\delta^+(v) = 0$ is an *output node*. The set $\mathcal{O} = \{v \mid v \in V \wedge \delta^+(v) = 0\}$ denotes the set of all output nodes. 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 $i = 1, 2, \ldots (k-1)$.

There are many processing graph models, but our synthesis method for building real-time systems from processing graphs is based on the U.S. Navy's Processing Graph Method (PGM). PGM was developed by the U.S. Navy to facilitate the design and implementation of signal processing applications.

In PGM, a system is expressed as a directed graph in which the nodes (or vertices) represent processing functions and the edges represent buffered communication channels called queues. The 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 First-In-First-Out (FIFO) queues with three attributes associated with each edge (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" (data structure elements) appended to the queue when the producing node (the source node for this queue) completes execution. The threshold amount represents the minimum number of tokens required to be present in the queue before the (queue's sink) 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 w, labeled q, has *prd*$(q) = 4$, *thr*$(q) = 7$, and *cns*$(q) = 3$. Node u must execute twice before node v is first eligible for execution. After node v executes, it consumes only 3 of the 8 tokens on its input queue. A threshold amount that is greater than the consume amount is often used in signal processing filters. The filter reads *thr*(q) tokens from the queue but only consumes *cns*(q) tokens, leaving at least (*thr*(q) − *cns*(q)) on the queue to be used in the next calculation.

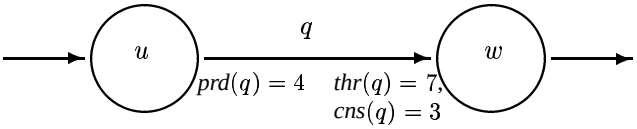

Figure 1. A two node chain.

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

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

3. Synthesis Method

Our synthesis method involves 3 steps: i) identifying node execution rates, ii) mapping each node to a task in the Rate-Based Execution (RBE) task model [11], and iii) verifying latency and memory requirements. Managing the memory needs of an application often requires an iteration of steps ii) and iii) as trade-offs are made between processor demand and memory requirements. §3.1 introduces the concept of node execution rates and how to derive the execution rate for each node in a PGM graph. The RBE task model is described in §3.2 for completeness. §3.3 shows how to map the processing graph nodes to tasks in the RBE task set and presents a sufficient schedulability condition for the resulting real-time system. An affirmative result after testing the scheduling condition means that the processor has enough

capacity to execute the graph such that latency and buffer requirements can be guaranteed. (Verifying and managing memory requirements are addressed in §4.)

3.1. Node Execution Rates

We assume the strong synchrony hypothesis of [5] to introduce the concept of node execution rates. Under the strong synchrony hypothesis, we assume the graph executes on an infinitely fast machine and each node takes "no time" to execute — the system instantly reacts to external stimuli. If node execution takes no time, then the behavior of processes is unaffected by scheduling and the execution patterns are the same under all scheduling policies.

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 ρ time units. At least ρ time units separate every release 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 the chain of Figure 1. If node u executes at times $0, y, 2y, 3y, \ldots$, node v is eligible for one execution at times y and $2y$, and 2 executions at time $3y$. The execution pattern of node v is repetitive, but neither periodic nor sporadic. Even though one can force the graph execution to fit either a periodic or sporadic task model with one task representing each node, it is unnatural and introduces additional latency. Moreover, while multiple periodic or sporadic tasks may be used to model a node's execution, a paradigm that supports expected rates of the form x executions of a node in y time units is a more natural and simpler task model for the analysis of schedulability, latency, and buffer requirements for a processing graph application.

Node execution rates are thus defined as follows. The time of the j^{th} execution of node v is represented as $T_j(v)$. An *execution rate* is an integer pair (x, y). An execution rate specification for node v, $R_v = (x, y)$, is *valid* if v executes exactly x times in all time intervals of $[t,\ t + y)$ where $t > T_1(v)$.

We start with chains and work up to general acyclic graphs in the derivation of node execution rates. Proofs for the theorems presented in this paper a can be found in [9].

Theorem 3.1. *Let $i \rightsquigarrow w$ be a PGM chain such that $i \in \mathcal{I}$ (the set of input nodes), $u, v \in \{i \rightsquigarrow w\}$ with $\psi(q) = (u, v)$, and $R_i = (x_i, y_i)$. 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*

$$x_v = \frac{prd(q)}{\gcd(prd(q)x_u, cns(q))} \cdot x_u \quad \text{and} \quad y_v = \frac{cns(q)}{\gcd(prd(q)x_u, cns(q))} \cdot y_u. \tag{3.1}$$

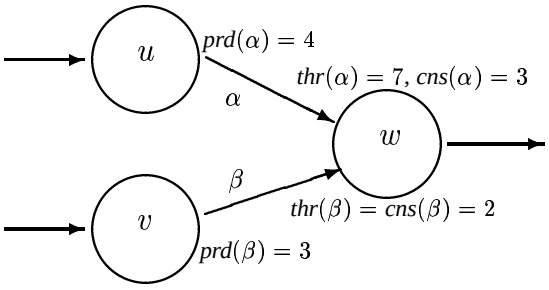

Figure 2. Node w has two input queues.

Equation (3.1) can be used to derive the execution rate of any consumer in terms of its producers in a chain of nodes. For example, given $\psi(q) = (u, w)$ for queue q and an execution rate of $R_u = (3, 16)$ for node u in Figure 1 on page 3 (i.e., node u executes 3 times in any interval of length 16), the execution rate of the consumer node w is derived using (3.1) as follows:

$$\begin{aligned} R_w &= (x_w, y_w) \\ &= \left(\frac{prd(q)x_u}{\gcd(prd(q)x_u, cns(q))}, \frac{cns(q)y_u}{\gcd(prd(q)x_u, cns(q))}\right) \\ &= \left(\frac{4 \cdot 3}{\gcd(4 \cdot 3, 3)}, \frac{3 \cdot 16}{\gcd(4 \cdot 3, 3)}\right) \\ &= \left(\frac{12}{\gcd(12, 3)}, \frac{48}{\gcd(12, 3)}\right) = (4, 16) \end{aligned} \tag{3.2}$$

Now consider the case when node w has another input queue, such as the graph in Figure 2. Node w is a consumer of data produced by both nodes u and v. The notation $R_{w \leftarrow u} = (x_{w \leftarrow u}, y_{w \leftarrow u})$ denotes the execution rate of node w with respect to data produced by node u as though they were a producer/consumer pair in a chain. Thus with $R_u = (3, 16)$, $R_{w \leftarrow u} = (4, 16)$, as derived in (3.2).

With $R_v = (2, 12)$, $R_{w \leftarrow v}$ is derived using (3.1) as follows:

$$\begin{aligned} R_{w \leftarrow v} &= (x_{w \leftarrow v}, y_{w \leftarrow v}) \\ &= \left(\frac{prd(\beta)x_v}{\gcd(prd(\beta)x_v, cns(\beta))}, \frac{cns(\beta)y_v}{\gcd(prd(\beta)x_v, cns(\beta))}\right) \\ &= \left(\frac{3 \cdot 2}{\gcd(3 \cdot 2, 2)}, \frac{2 \cdot 12}{\gcd(3 \cdot 2, 2)}\right) = \left(\frac{6}{2}, \frac{24}{2}\right) = (3, 12) \end{aligned}$$

Since w can only execute when *both* α and β are over threshold, neither $R_{w\leftarrow u}$ nor $R_{w\leftarrow v}$ satisfies the definition of a valid execution rate for node w because node w will not execute exactly 4 times in any interval of length 16 or exactly 3 times in any interval of length 12. It can be shown from Theorem 3.1, that although in general $R_{w\leftarrow u} \neq R_{w\leftarrow v}$, it must be the case that $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$ if a valid graph execution is possible. Intuitively, the expression $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$ means that the *steady state execution rate* of node w with respect to node u is equal to the *steady state execution rate* of node w with respect to node v. Lemma 3.2 states that without the equality of steady state execution rates, it would be impossible to schedule a valid execution of the graph using finite memory.

Lemma 3.2. *Let $G = (V, E, \psi)$ be a PGM digraph and $u, v, w \in V$ for which there exists queues α and β such that $\psi(\alpha) = (u, w)$ and $\psi(\beta) = (v, w)$. If a valid graph execution is possible using finite memory for buffering tokens, then $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$.*

When a consumer node has a constant *steady state execution rate* with respect to its producers, (3.1) can be generalized to derive the execution rate of a node with multiple input queues.

Theorem 3.3. *Let $G = (V, E, \psi)$ be a PGM digraph for which a valid execution is possible using finite memory and node $v \in V$ with $\delta^-(v) \geq 1$. Assuming the strong synchrony hypothesis 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*

$$y_v = \operatorname{lcm}\{\frac{cns(q)y_u}{\gcd(prd(q)x_u, cns(q))} \mid \psi(q) = (u, v)\},$$
$$x_v = y_v \cdot \left(\frac{prd(q)x_u}{cns(q)y_u}\right), \ \forall q, u : \psi(q) = (u, v). \tag{3.3}$$

For example, given nodes u and v in Figure 2 with $R_u = (3, 16)$ and $R_v = (2, 12)$, since $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$, the execution rate of w is:

$$\begin{aligned} y_w &= \operatorname{lcm}\{\frac{cns(\alpha)y_u}{\gcd(prd(\alpha)x_u, cns(\alpha))}, \frac{cns(\beta)y_v}{\gcd(prd(\beta)x_v, cns(\beta))}\} \\ &= \operatorname{lcm}\{\frac{3 \cdot 16}{\gcd(4 \cdot 3, 3)}, \frac{2 \cdot 12}{\gcd(3 \cdot 2, 2)}\} \\ &= \operatorname{lcm}\{\frac{3 \cdot 16}{3}, \frac{2 \cdot 12}{2}\} = \operatorname{lcm}\{16, 12\} = 48 \\ \implies x_w &= y_w \cdot \left(\frac{prd(\alpha) \cdot x_u}{cns(\alpha) \cdot y_u}\right) = 48 \cdot \left(\frac{4 \cdot 3}{3 \cdot 16}\right) = 12 \end{aligned}$$

Thus $R_w = (x_w, y_w) = (12, 48)$ and, after its first execution, node w in Figure 2 will execute 12 times in every interval of length 48.

3.2. RBE Task Model

Once node execution rates have been established, the nodes of a graph can be mapped to real-time tasks. Unfortunately since nodes are neither periodic nor sporadic, even when the source is periodic, most task models from the literature are inapplicable. The Rate-Based Execution (RBE) paradigm [11], however, does provide a natural description of node executions in an implementation of processing graphs. This section provides a brief overview of the RBE 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 to be requested in an interval of length y. Parameter d is a response time parameter that specifies the maximum desired time between the release of a task instance and the completion of its execution (i.e., d is the relative deadline). The parameter e is the maximum amount of processor time required for one execution of the task.

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

$$D_i(j) = \begin{cases} t_{i,j} + d_i & \text{if } 1 \leq j \leq x_i \\ \max(t_{i,j} + d_i, D_i(j - x_i) + y_i) & \text{if } j > x_i \end{cases} \tag{3.4}$$

The RBE task model makes no assumptions regarding when a task will be released, however (3.4) ensures that no more than x_i deadlines come due in an interval of length y_i, even when more than x_i releases of T_i occur in an interval of length y_i. Hence, the deadline assignment function prevents jitter from creating more process demand in an interval by a task than that which is specified by the rate parameters.

3.3. Mapping Nodes to Real-Time Tasks

To map a PGM graph to a set of RBE tasks, we associate a task with each node. Thus for each node u in the graph, node u is associated with the four tuple (x_u, y_u, d_u, e_u). The parameters x_u and y_u are derived using (3.3). The parameter e_u is the worst case execution time for node u, which we assume is supplied. The only free parameter is the relative deadline parameter d_u, which influences processor capacity requirements, latency, and buffer requirements. In general, a smaller value chosen for d_u will result in less latency and memory requirements than a larger d_u value, but at a cost of increased processor capacity requirements. Execution time, produce, threshold, consume, and deadline values all affect schedulability, latency and buffer requirements, and one can trade-off one metric for any other.

The synthesis method outlined here provides a framework for evaluating schedulability and memory requirements, but leaves open the problem of partitioning a processing graph in a distributed system when the graph is not schedulable on a uniprocessor.

In mapping the graph to a set of RBE tasks, relative deadline parameters need to be selected that result in modest buffering on the graph edges without overloading the processor with too much processing demand. Since d_u affects processor capacity requirements, latency and buffer requirements, a good starting point for the selection of d_u is one such that d_u is greater than or equal to the deadline of node u's predecessor node and less than or equal to y_u. As shown in [8], when the deadline for each node is greater than or equal to its predecessor's deadline, a scheduling technique called release time inheritance can be used to minimize latency. Under release time inheritance, node u is assigned a logical release time (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, (3.4), then uses the logical release times rather than the actual release times.

After we have associated each node u in the graph with a four tuple (x_u, y_u, d_u, e_u), we have an RBE task system $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$. A task is released when all of the node's input queues are over threshold, ensuring precedence constraints are met for correct graph execution. Released tasks are scheduled with the RBE-EDF scheduling algorithm — a simple, preemptive, EDF scheduler using deadline assignment function (3.4) with release time inheritance.

The feasibility of an RBE task set can be determined with (3.5) [8]. Notice that (3.5) reduces to the simple Liu & Layland [12] EDF feasibility condition of $U \leq 1$ when $x_i = 1$ and $d_i = y_i$.

Lemma 3.4. *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*

$$\forall L > 0, \quad L \geq \sum_{i=1}^{n} f\left(\frac{L - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i \qquad (3.5)$$

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

Sufficiency of (3.5) was established in [8] 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 (3.5). For scheduling RBE tasks derived from a PGM graph as described above, (3.5) becomes a sufficient but not necessary condition for preemptive EDF scheduling. (3.5) 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 3, this is not possible — if $R_u = (1, y)$, then any execution of node u releases at most 2 executions of node v even though $R_v = (4, 3y)$, and node v will not have 4 releases at one time.

From Lemma 3.4 and the preceding discussion, we obtain:

Theorem 3.5. *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 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* (3.5) *holds for $\mathcal{T}$.*

An affirmative result after testing (3.5) means that the RBE-EDF scheduler can be used to execute the graph without missing a deadline, and, as explained next, we can bound the buffer requirements of each queue and the entire graph.

4. Managing Memory Requirements

The three primary uses of memory in an embedded signal processing system are (1) scheduler state space, (2) code space for each node, and (3) buffer space for intermediate results stored on graph edges. In [9], we argue that the memory requirements for (1) and (2) are similar for either statically or dynamically scheduled implementations. Here, we only address buffer space requirements.

The canonical approach to managing the memory requirements of the graph edges is to use static scheduling. Static node execution schedules are created off-line and then executed on a periodic basis. The primary trade-off made by static schedulers is the storage requirement and execution complexity of the schedule vs. the storage requirement of data in the graph edges. Typically, efficient buffer usage requires increased state space for the scheduler. Some scheduling algorithms produce a simple flat schedule with each entry in the schedule representing the execution of a single node. Other algorithms save scheduler state space by associating a number of executions with each scheduler entry to reduce state space for multiple executions of the same node. Still other scheduling algorithms produce slightly more complicated schedules by creating scheduling loops encompassing many scheduling entries [2]. For example, three different possible schedules for the chain in Figure 1 are: $uwuwuww$ — a multiple appearance flat schedule, $(3u)(4w)$ — a single appearance flat schedule, and $(3uw)w$ — a multiple appearance looped schedule.[1] A single appearance looped schedule is not possible for this graph. The schedules $uwuwuww$ and $(3uw)w$ produce identical execution results.

The buffer space required by static schedulers is dependent on the particular scheduling algorithm. For example,

[1] The notation in the looped schedule is such that the 3 applies to all subsequent nodes until the right parenthesis is reached [2].

assume the input queue to node u in Figure 1 is labeled q' and that it is attached to a periodic external device i. Let $R_i = (1, 8)$, $prd(q') = 6$, and $thr(q') = cns(q') = 4$. The multiple appearance flat schedule $uwuwuwuw$ and the multiple appearance looped schedule $(3uw)w$ require storage for 10 tokens on queue q, while the single appearance flat schedule $3u4w$ requires storage for 16 tokens on queue q. To derive the storage requirement of the input queue q', we need to consider the input data rate and the duration of the schedule. Static schedules are usually executed on a periodic basis with a timer indicating when to start executing the schedule. Since the schedule executes without inserting idle time (once it starts), execution cannot begin until enough data has accumulated on the input queues to the graph to ensure a valid graph execution.

The period of a static schedule is equal to the maximum y value of the execution rates of the set of nodes connected to output devices — i.e., $\max\{y_w\}$ where $w \in \mathcal{O}$. Let y_w be the period of a static schedule, and let s_w be the the logical release time of the first execution of node w. The first execution of the static schedule cannot begin before time S where S is bounded such that $s_w \leq S \leq s_w + y_w$.[2] For example with $R_i = (1, 8)$, the execution rates for nodes u and w in Figure 1 are $R_u = (3, 16)$ and $R_w = (4, 16)$. Let node w be the only node connected to an output device, and $S = s_w = 8$. Since $y_w = 16$, the scheduling period is 16 time units. To ensure data availability for each execution of node u, the schedule cannot begin until time 8 (the first logical release time of node w, which will also be the logical release time of the execution of node u that releases node w at s_w). Hence, with $prd(q') = 6$, 12 tokens accumulate on the input queue q' before the schedule even begins — source node i first produces at time 0. Depending on the schedule, the node execution times, and the execution characteristics of the source node, more data may accumulate during the execution of the schedule. Therefore, buffering for at least $12 + 16 = 28$ tokens are required for queues q and q' when a single appearance flat schedule is used to execute the two nodes. When deriving buffer requirements for queues connected to external sink devices, we assume that the external device consumes data as soon as it is available. Since node w produces 1 token every time it executes, a statically scheduled implementation of the graph requires storage for at least 29 tokens (and possibly more). It is important to recognize that depending on the length of the scheduling period and when the static schedule first starts, the amount of data that accumulates on the input queues of an actual application can be quite substantial.

In contrast to using off-line scheduling algorithms to manage memory requirements, our approach is to use a simple, dynamic, on-line scheduler for graph execution. One of the advantages of the RBE model over other dynamic scheduling algorithms 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. In [8], we presented functions to bound the buffer requirements of queues in chains scheduled with variations of the RBE-EDF algorithm. Applying those functions to the 2 node chain of Figure 1 with the assumption that $d_u = d_w = 16$, queue q would require storage space for either 10 or 16 tokens — depending on how deadline ties between executions of nodes u and w were broken. Unfortunately, the execution rules for PGM nodes with multiple input queues are such that one input queue may be over threshold long before another input queue, and the equations derived in [8] to bound queue buffers do not apply to general PGM graphs. In the general case, the buffer bounds that we can derive are much too loose to be useful (though they are valid upper bounds). However, many signal processing applications possess dataflow characteristics that we can exploit to get relatively tight buffer bounds.

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.

To realize these tighter bounds, however, we need to identify the release time of the first execution of a producer node u. As in the static scheduling example, let s_u denote the time associated with the first release of node u.

For the common cases in signal processing applications where all of the queues in the graph are initialized with $(thr(q) - cns(q))$ tokens and $\gcd(cns(q), prd(q)x_u) = \min(cns(q), prd(q)x_u)$ for each queue, Theorem 4.1 gives a tight bound on the buffer space required by queue q when the graph is executed using the RBE-EDF scheduler.

Theorem 4.1. *Let $G = (V, E, \psi)$ be valid PGM digraph such that each queue, q, in the path(s) from a periodic source to node v is initialized with $(thr(q) -$*

[2] Due to space limitations, we are unable to present the functions that derive S and s_w —see [9].

cns(q)) *tokens prior to the beginning of graph execution and* $\gcd(cns(q), prd(q)x_u) = \min(cns(q), prd(q)x_u)$. *Under the RBE-EDF scheduling algorithm, the buffer requirements for queue q with* $\psi(q) = (u, v)$ *is Buf*(q) *where*

$$Buf(q) \leq \left\lceil \frac{\max(y_v, s_v + d_v - s_u)}{y_u} \right\rceil x_u prd(q) + (thr(q) - cns(q)). \quad (4.1)$$

In practice, we have found (4.1) to be applicable to the majority of signal processing applications we have analyzed. More importantly Theorem 4.1 provides guidance to signal processing engineers developing processing graphs by quantifying the impact of their choice for thresholds, produce, and consume values on the memory requirements of the graph.

Theorem 4.1 also demonstrates the impact of the deadline parameters chosen for each task in the synthesis of a real-time system from the processing graph. A general heuristic to follow in selecting the deadline parameter for node v is to set it less than or equal to node v's execution interval y_v and greater than or equal to the predecessor's deadline. This often results in modest buffering requirements while allowing deadline inheritance to be used to minimize latency. A second heuristic to follow, when processor capacity allows, is to set a consumer node's deadline parameter equal to the producer's deadline parameter if the consumer executes once for every k executions of the producer, which minimizes the data that accumulates on the consumer's input queue between the time when it is released and when it completes execution.

5. Case Study

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

The INMARSAT system has been offering mobile satellite communication service to ocean-going vessels 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 kbps 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 is a block diagram of the digital signal processing performed by the satellite receiver portion of an INMARSAT mobile terminal [22]. The corresponding processing graph for this application is shown in Figure 4 [18]. 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. 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.

5.1. Node Execution Rates

Most signal processing applications do not have relatively prime produce and consume dataflow attributes as the earlier examples do (unless one of the values is 1). They do, however, typically have node execution rate changes throughout the graph — both execution rate decreases and execution rate increases. For example, let the period of each of the two input nodes to the INMARSAT graph of Figure 4 be y. Then the execution rate of node A is $(1, y)$ and the execution rate of node B is

$$R_B = \begin{cases} y_B = \mathrm{lcm}(\frac{4}{\gcd(1\cdot 1, 4)} \cdot y) = 4y \\ x_B = 4y \cdot \left(\frac{1\cdot 1}{4\cdot y}\right) = 1 \end{cases} \implies (1, 4y)$$

Another rate change occurs at node P in Figure 4, 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

$$R_P = \begin{cases} y_P = \mathrm{lcm}\{\frac{1\cdot 44y}{\gcd(10\cdot 1, 1)}, \cdots \frac{1\cdot 44y}{\gcd(1\cdot 10, 1)}\} = 44y \\ x_P = 44y \cdot \left(\frac{10\cdot 1}{1\cdot 44y}\right) = 10 \end{cases}$$
$$= (10, 44y)$$

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) tasks used to implement the graph.

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.

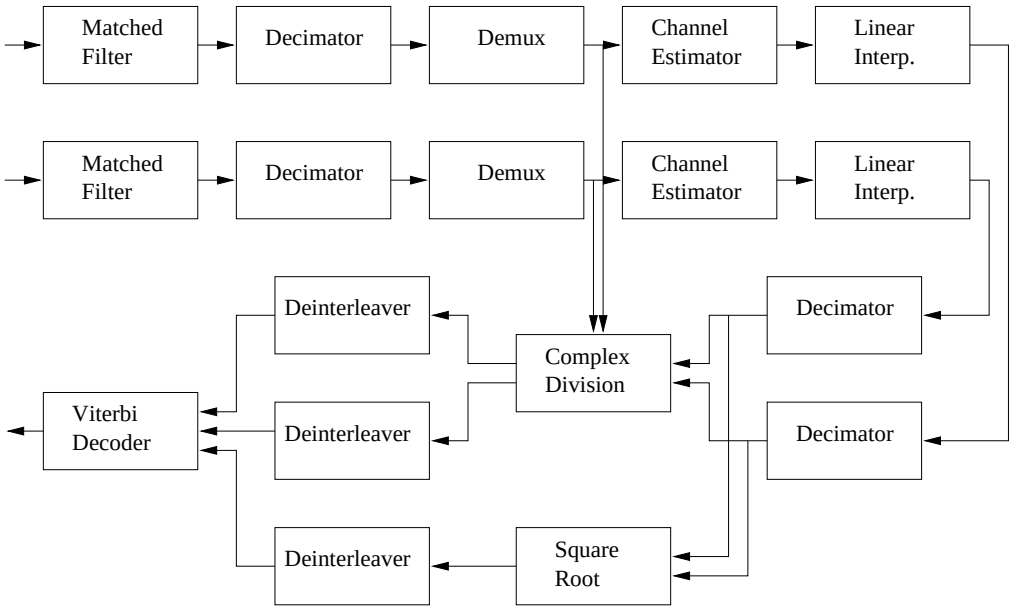

Figure 3. Block diagram of the INMARSAT mobile receiver.

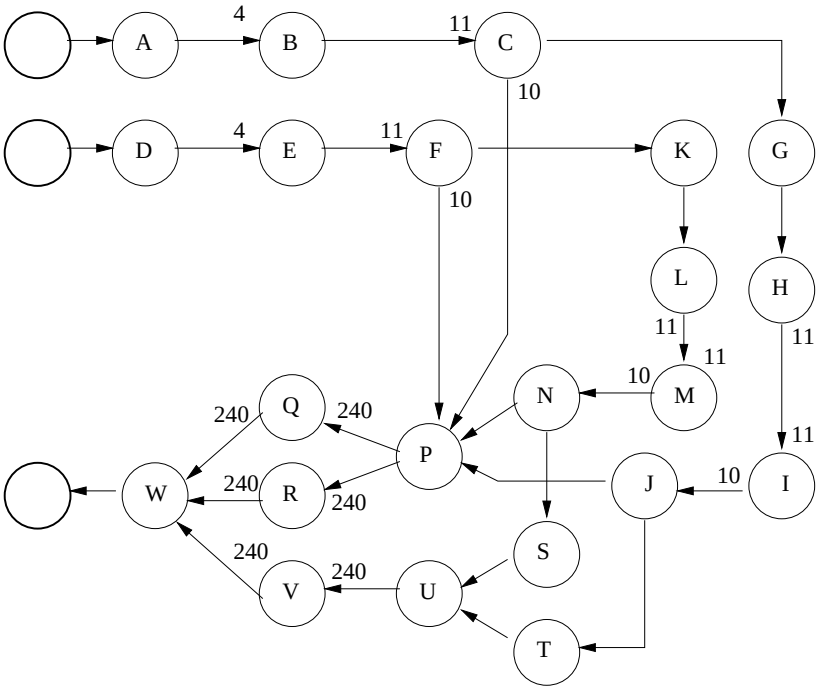

Figure 4. PGM application graph for the INMARSAT mobile receiver.

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.

5.2. 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 [18, 2]. We derive the bounds for one queue as an example and refer the reader to Table 2 (and [9]) for the buffer bounds on the remaining queues. The analysis is abstract in that all tokens are assumed to be the same size.

Let a be the label for the queue joining nodes A and B in the application graph of Figure 4 on page 9. Applying (4.1) of §4 to queue a (using the RBE and start time parameters of Table 1), we get an upper bound on the buffer requirement for a of

$$\begin{aligned} \mathit{Buf}(a) &\leq \left\lceil \frac{\max(y_B, s_B + d_B - s_A)}{y_A} \right\rceil x_A \mathit{prd}(a) \\ &\quad + (\mathit{thr}(a) - \mathit{cns}(a)) \\ &= \left\lceil \frac{\max(4y, 3y + y - 0)}{y} \right\rceil \cdot 1 \cdot 1 + (4 - 4) = 4. \end{aligned}$$

If we had chosen a deadline value of $4y$ for node B rather than y, the buffer requirement would have been 7 (instead of 4). In this case, we applied the second heuristic from the end of §4 and used the producer's deadline value since B only executes once every $4y$ time units.

The values of $\mathit{Buf}(q)$ 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. The queues attached to input or output devices are omitted from the table since these queues were ignored in the buffer calculations of [18, 2]. The buffer space required for each of these queues in our model is 1 token.

$\psi(q)$	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 4.1.

The minimum buffer requirement possible for the INMARSAT graph is 1,545 tokens (including the graph input and output queues) — derived by summing $\frac{\mathit{prd}(q) \cdot \mathit{cns}(q)}{\gcd(\mathit{prd}(q), \mathit{cns}(q))} = \max(\mathit{prd}(q), \mathit{cns}(q))$ over all queues in the graph [2]. 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 buffer requirement of 1,545 tokens.

If a shared buffer implementation is used for the RBE task set, we can reduce the upper bound of 1,599 to 1,101. Observe that, since the RBE-EDF scheduler 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 tokens we derived by we summing individual 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 $3 \cdot 240 = 720$ 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 $240 + 720 = 960$ 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 tokens. 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 node J complete before the next execution of node I and the input queue to node J will never contain more than 10 tokens. Each time node 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.

Since node W is the only node attached to an output device and $S = s_w = (24 \cdot\ 44y) - y$ for the INMARSAT graph, a statically scheduled execution cannot commence until time $(24 \cdot\ 44 - 1)y = 1055y$ — we assume the source devices begin producing one token every y time units starting at time 0. Thus, the buffer space required for each of the queues attached to input devices is at least 1,056 tokens when an off-line scheduler is employed for this graph. The total memory space required to buffer tokens on each input queue may be even higher. For example, once the single appearance looped schedule $(24(11(4A)B)CGHI(11(4D)E)FKLM(10NSJTUP))$ $QRV(240W)$ produced by the off-line *Acyclic Pairwise Grouping of Adjacent Nodes* (APGAN) scheduling algorithm of [2] begins, additional tokens will accumulate on the input queues, which may increase the total buffer space required. Thus, the total buffer space required for the two queues attached to input devices is at least 2,112 tokens for the APGAN schedule. The single appearance flat schedule $(1056A)(264B)(24C)(24G)(24H)(24I)(240J)(1056D)$ $(264E)(24F)(24K)(24L)(24M)(240N)(240P)(240S)$ $(240U)VQR(240W)$ of [18] also requires a delay of 1054 periods of the input devices. For this schedule, however, no more than 1056 tokens will accumulate on an input queue since the execution time of A must be less than the period of the external source for the schedule to be feasible. Thus, the total buffer space required for the queues attached to input devices is 2,112 tokens for the single appearance flat schedule.

The off-line APGAN scheduling algorithm uses unique buffers for each queue and 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 external devices, however, the buffer requirement of a statically scheduled implementation is at least 3,655 tokens — 128.6% higher than the dynamically scheduled, unique buffer implementation and 331.97% higher than the dynamically scheduled, shared buffer implementation.

An advantage of the single appearance flat schedule is that a single shared buffer can be used, which often results in less memory requirements for the graph edges. For this applications, however, even with a shared buffer the flat schedule requires space for 2040 tokens for the queues listed in Table 2. Adding storage space for the 2,113 tokens buffered on the queues attached to external devices raises the total memory requirement to 4,153 tokens, which is 159.7% higher than the 1599 tokens buffered in the dynamically scheduled, unique buffer implementation and 377.2% higher than the dynamically scheduled, shared buffer implementation.

When a unique buffer is used for each graph edge, static schedules require 128.6% more total buffer space than our dynamic scheduling approach. When a shared buffer approach is combined with dynamic scheduling, static schedulers require between 331.97% and 377.2% more buffer memory than the RBE-EDF scheduler. Of course the use of a shared buffer requires additional code and overhead to manage the shared buffer, but we can quantify the trade-off and make reasonable choices.

6. 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 schedulability, latency and buffer requirements, and one can trade-off one metric for any other. The synthesis method outlined here provides a framework for evaluating schedulability and memory requirements, but leaves open the problem partitioning of a processing graph in a distributed system when the graph is not schedulable on a uniprocessor.

We have also shown that by judiciously selecting deadline parameters for the nodes of an INMARSAT mobile 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 many signal processing applications. By using some of the processor capacity for on-line scheduling decisions, we can achieve better latency and may execute with less memory than off-line scheduling.

References

[1] Baruah, S., Goddard, S., Jeffay, K., "Feasibility Concerns in PGM Graphs with Bounded Buffers," Proc. of the Third Intl. Conference on Engineering of Complex Computer Systems, Sept., 1997, pp 130-139.

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

[3] Chatterjee, S., Strosnider, J., "Distributed Pipeline Scheduling: A Framework for Distributed, Heterogeneous Real-Time System Design," *The Computer Journal* (British Computer Society), Vol. 38, No. 4, 1995.

[4] Chatterjee, S., Strosnider, J., "A Generalized Admissions Control Strategy for Heterogeneous, Distributed Multimedia Systems," *Proc. of ACM Mutlimedia 95*, Nov. 1995.

[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] Bondy, J.A., Murty, U.S.R., *Graph Theory with Applications*, North Holland, 1976.

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

[9] Goddard, S., "On the Management of Latency and Memory Requirements in the Synthesis of Distributed Real-Time Signal Processing Systems from Processing Graphs," Ph.D. Dissertation, University of North Carolina at Chapel Hill, 1998.

[10] Jeffay, K., "The Real-Time Producer/Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems," *Proc. of ACM/SIGAPP Symp. on Appl. Computing*, Feb. 1993, pp. 796-804.

[11] Jeffay, K., Bennett, D. "A Rate-Based Execution Abstraction For Multimedia Computing," *Lecture Notes in Computer Science*, T.D.C. Little and R. Gusella eds., Vol. 1018, Springer-Verlag, Heidelberg, 1995, pp 65-75.

[12] 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.

[13] Lee, E.A., Messerschmitt, D.G., "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, C-36(1), Jan. 1987, pp. 24-35.

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

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

[16] Ramamritham, K., "Allocation and Scheduling of Precedence-Related Periodic Tasks," *IEEE Trans. on Parallel and Dist. Syst.*, 6(4), April 1995, pp 412-420.

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

[18] 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.

[19] Sun, J., Liu, J., "Synchronization Protocols in Distributed Real-Time Systems," *Proc Intl. Conference on Dist. Computing Syst.*, May, 1996.

[20] 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*, Dec. 1996, pp. 2-12.

[21] 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, Dec. 1994, pp. 1407-1412.

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