In: *Proceedings of the Fifth IEEE International Symposium on High Assurance Systems Engineering*, Albuquerque, New Mexico, November 2000, pp. 177-186.

The Synthesis of Real-Time Systems from Processing Graphs

Steve Goddard
Computer Science & Engineering
University of Nebraska—Lincoln
Lincoln, NE 68588-0115
goddard@cse.unl.edu

Kevin Jeffay
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175
jeffay@cs.unc.edu

Abstract

Directed graphs, called processing graphs, are a standard design aid for complex real-time systems. The primary problem in developing real-time systems with processing graphs is transforming the processing graph into a predictable real-time system in which latency can be managed. Software engineering techniques are combined with real-time scheduling theory to solve this problem. In the parlance of software engineering methodologies, a synthesis method is presented. New results on managing latency in the synthesis of real-time systems from cyclic processing graphs are also presented. The synthesis method is demonstrated with an embedded signal processing application for an anti-submarine warfare (ASW) system.

1. Introduction

Directed graphs, called *processing graphs*, are a standard design aid for complex real-time systems. Processing graphs are large grain dataflow graphs in which nodes represent processing functions and graph edges depict the flow of data from one node to the next. Depending on the application domain, the flow of data represents a sampled signal to be processed (e.g., [7, 21, 5, 3, 25, 30]) or messages to be processed (e.g.,[6, 9, 17, 22, 24, 29, 27, 28]).

The primary problem in developing real-time systems with processing graphs is transforming the processing graph into a predictable real-time system in which latency can be managed. We combine software engineering techniques with real-time scheduling theory to solve this problem. In the parlance of software engineering methodologies, we present a synthesis method.

We also present new results on managing latency in the synthesis of real-time systems from cyclic processing graphs. The total latency encountered by a sample or message in a processing graph is an integral unit of time created by the sum of the latency inherent in the processing graph and the additional latency imposed by the implementation. *Inherent latency* in a graph is created by non-unity dataflow attributes (described in Section 3) and the graph topology. Inherent latency exists even if the graph is executed on an infinitely fast machine. *Imposed latency* comes from the scheduling and execution of nodes in the graph since we do not have an infinitely fast machine. Thus latency has two components: inherent latency and imposed latency. In [14], we presented equations to compute inherent latency in cyclic processing graphs. Here we use real-time scheduling theory to bound imposed latency and combine these results with our bounds on inherent latency to provide an upper bound on the total latency any sample (or message) will encounter in the synthesized system. If this bound is less than or equal to the latency requirement for each path in the graph, we can guarantee that the application will always meet its latency requirements.

We demonstrate our synthesis method with an embedded signal processing application for an anti-submarine warfare (ASW) system. For simplicity and concreteness, we present our synthesis method in terms of the U.S. Navy's Processing Graph Method (PGM) [23] and signal processing applications. However, our synthesis method applies to any general processing graph paradigm and many application domains.

The rest of this paper is organized as follows. Our results are related to other work in Section 2. Section 3 presents a brief overview of the processing graph model PGM. The synthesis of real-time uniprocessor systems from PGM graphs, including the the verification of latency requirements, is presented in Section 4. We demonstrate our synthesis method in Section 5 with the Directed Low Frequency Analysis and Recording (DIFAR) acoustic signal processing application from the Airborne Low Frequency Sonar (ALFS) system of the SH-60B LAMPS MK III anti-submarine helicopter. Our contributions are summarized in Section 6.

2. Related Work

This paper is part of a larger body of work that creates a framework for evaluating and managing processor demand,

In: *Proceedings of the Fifth IEEE International Symposium on High Assurance Systems Engineering*, Albuquerque, New Mexico, November 2000, pp. 177-186.

latency, and memory usage in the synthesis of real-time systems from processing graphs [10]. Here, we demonstrate the management of latency in the synthesis of a real-time uniprocessor system from cyclic processing graphs developed with PGM. Portions of our synthesis method have been presented previously [12, 13, 14]. However, this is the first complete presentation of our synthesis method. (For space considerations, we refer to our previous publications as needed in the full synthesis method presented here.)

Our early work on the synthesis of real-time uniprocessor systems from PGM was based on acyclic PGM graphs [12, 13]. We first introduced the concept of precise node execution rates in [12], and we showed that dynamic, on-line scheduling can achieve near minimal memory requirements in [13]. In [14], we extended these results to compute node execution rates and inherent latency for cyclic processing graphs. This is the first time *imposed latency* has been computed for cyclic processing graphs. Moreover, our latency results are novel in that they assume nodes are eligible for execution as soon as all required input data is available and then scheduled for execution with a dynamic on-line scheduler.

From the real-time literature, PGM graphs are most closely related to the Logical Application Stream Model (LASM) [6] and the Generalized Task Graph (GTG) model [7]. PGM, LASM, and GTG were all developed independently and support very similar dataflow properties. PGM was the first of these to be developed. Our work improves on the analysis of LASM and GTG graphs by not requiring periodic execution of the nodes in the graph. Instead, we calculate a more general execution rate, which can be reduced to average execution rates assumed in the LASM and GTG models. Our general execution rate specification provides a more natural representation of node execution for PGM graphs. Forcing periodic execution of all graph nodes adds latency to the processed signal, but simplifies the analysis of latency and memory requirements.

Processing graphs are a standard design aid in digital signal processing. From the digital signal processing literature, PGM is most similar to Lee and Messerschmitt's Synchronous Dataflow (SDF) graphs [21] supported by the Ptolemy system [5]. The SDF graphs of Ptolemy utilize a subset of the features supported by PGM. Any SDF graph can be represented as a PGM graph where each queue's threshold is equal to its consume value. In addition to supporting a more general dataflow model, our research differs from [21] in that we support dynamic, real-time, scheduling techniques rather than creating static schedules.

In 1996, Bhattacharyya, Murthy, and Lee published a method for software synthesis from dataflow graphs [3]. Their software synthesis method is based on the static scheduling of Lee and Messerschmitt's SDF graphs. The main goal of Bhattacharyya *et al.*'s software synthesis method and related scheduling research based on SDF graphs has been to minimize memory usage by creating off-line scheduling algorithms [21, 25, 30, 26, 3]. Off-line schedulers create a static node execution schedule that is executed periodically by the processor. In contrast, the primary goal of our research has been to manage the latency and memory usage of processing graphs by executing them with an on-line scheduler. Recently we have shown that for a large class of applications, dynamic on-line scheduling creates less imposed latency than static scheduling. An even more surprising result is that, in many cases, dynamic on-line scheduling uses less memory for buffering data on graph edges than static scheduling [13].

Our latency analysis is related to the work of Gerber *et al.* in guaranteeing end-to-end latency requirements on a single processor [9]. However, Gerber *et al.* map a task graph to a periodic task model in the synthesis of real-time message-based systems rather than assuming a rate-based execution. Our analysis and management of latency differs from Gerber *et al.*'s in that PGM graphs allow non-unity dataflow attributes. Finally, Gerber *et al.* introduce new (additional) tasks to the task set in their synthesis method to synchronize processing paths. Our synthesis method does not need extra synchronization tasks since our analysis techniques are rate-based rather than periodic.

3. Notation and the Processing Graph Method

The notation and terminology of this paper, for the most part, is an amalgamation of the notation and terminology used in [4] and [3]. 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 w_i is adjacent to w_{i+1} for $i = 1, 2, \ldots (k-1)$. The set $\mathcal{I}_v$ is the subset of input nodes $\mathcal{I}$ from which there exists a path from $u \in \mathcal{I}$ to the node v. Likewise, the set $\mathcal{O}_u$ is the subset of output nodes $\mathcal{O}$ from which there exists a path from node

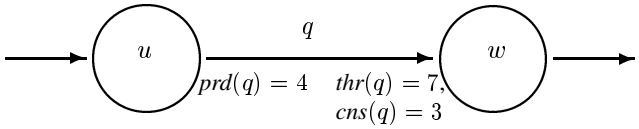

Figure 1. A two node chain.

u to $w \in \mathcal{O}$.

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

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 $prd(q) = 4$, $thr(q) = 7$, $cns(q) = 3$, and $init(q) = 0$. (A queue without an $init(q)$ label contains no initial data.) Node u must execute twice before node w is first eligible for execution. After node w 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.

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

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

4. Synthesis Method

In this section, we combine software engineering techniques with real-time scheduling theory to develop a synthesis method for transforming a processing graph into a predictable real-time system in which latency can be managed. The synthesis of real-time systems from PGM graphs involves four steps:

1. Identification of the rates at which nodes in a PGM graph must execute if they are to process data in real time.
2. Construction of a mapping of each node to a task in a real-time task model so that real-time processing can be achieved.
3. Verification that the resulting task set is schedulable so that we can guarantee real-time execution.
4. Analytical verification that latency requirements of the application are met.

The analysis of latency requirements only holds if the task set is schedulable. Thus, the schedulability of the task set is tested in Step 3. If the task set is not schedulable, Steps 2 and 3 must be repeated with a modified set of parameters used in the mapping of PGM nodes to real-time tasks. Step 4 uses analytical techniques (rather than simulation) to verify latency requirements are met. If the synthesized system will not meet specified latency requirements, then Steps 2, 3 and 4 must be repeated with a modified set of task parameters.

4.1. Step 1: Computing Node Execution Rates

We have shown, in [10, 13, 14], that nodes in a PGM graph execute with a precise rate of x executions every y time units. We call the integer pair (x, y) an execution rate, and represent the execution rate of node w as $R_w = (x_w, y_w)$. Moreover, given the execution rate of input nodes producing input data for the application, we have

shown that the execution rate of every node w in a cyclic PGM graph can be calculated using Equation (1):

$$y_w = \mathrm{lcm}\{\frac{cns(q)y_u}{\gcd(prd(q)x_u, cns(q))} \mid \psi(q) = (u, w)\},$$
$$x_w = y_w \cdot \left(\frac{prd(q)x_u}{cns(q)y_u}\right), \ \forall q, u : \psi(q) = (u, w). \quad (1)$$

We also showed in [14] how to initialize back edges in a cycle so that they are always over threshold during graph execution, which is necessary for Equation (1) to compute valid execution rates in cyclic graphs. (A *back edge* is a queue q that joins node v to an ancestor w when the graph is topologically sorted.) We assume in this paper that all back edges in cyclic graphs are so initialized.

4.2. Step 2: Mapping Nodes to a Real-Time Task Model

The execution rate specifications computed using Equation (1) represent the rate at which nodes need to execute to achieve real-time execution without losing data. We now address issues related to scheduling nodes in accordance with their rate specifications. To make sure nodes execute according to their rate specifications we execute the nodes according to the rate-based execution (RBE) model [18]. The RBE paradigm provides a natural description of node executions in an implementation of processing graphs. The advantage of executing nodes according to the RBE model is that nodes are eligible for execution as soon as they are released, even if multiple releases of a node occur at the same time. In comparison, a periodic model of execution requires that each release of a node be separated by a constant amount of time, which imposes additional latency on the signal.

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

$$D_i(j) = \begin{cases} t_{i,j} + d_i & \text{if } 1 \le j \le x_i \\ \max(t_{i,j} + d_i, D_i(j - x_i) + y_i) & \text{if } j > x_i \end{cases} \quad (2)$$

The RBE task model makes no assumptions regarding when a task will be released, however Equation (2) 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.

Mapping Nodes to RBE Tasks. To map a PGM graph to a set of RBE tasks, a task is associated 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 Equation (1). 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 latency 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 relative deadline of node u's predecessor node and less than or equal to y_u. As shown in [12], when the deadline for each node is greater than or equal to its predecessor's relative 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. Deadline assignment function (2) then uses the logical release times rather than the actual release times to assign deadlines for the completion of node execution.

4.3. Step 3: Verifying Schedulability

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, earliest-deadline-first (EDF) scheduler using deadline assignment function (2) with release time inheritance.

The schedulability of the resulting task set can be checked with Theorem 4.1 [13].

Theorem 4.1. *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 Equation* (3) *holds for* $\mathcal{T}$.

$$\forall L > 0, \;\; L \geq \sum_{i=1}^{n} f\left(\frac{L - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i \qquad (3)$$

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

An affirmative result after evaluating Condition (3) means that the RBE-EDF scheduler can be used to execute the graph without missing a deadline. If the cumulative processor utilization for an RBE task set is strictly less than one (i.e., $\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} < 1$) then Condition (3) can be evaluated efficiently (in pseudo-polynomial time) using techniques developed by Baruah *et al*. [1]. Moreover, when $d_i = y_i$ for all T_i in $\mathcal{T}$, the evaluation of Condition (3) reduces to the polynomial-time schedulability condition

$$\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} \leq 1 \qquad (4)$$

since $\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} \leq 1 \implies \forall L > 0, \;\; L \geq \sum_{i=1}^{n} L \cdot \frac{x_i \cdot e_i}{y_i}$

$$\begin{aligned} &= \sum_{i=1}^{n} \frac{L}{y_i} \cdot x_i \cdot e_i \\ &= \sum_{i=1}^{n} \frac{L - y_i + y_i}{y_i} \cdot x_i \cdot e_i \\ &= \sum_{i=1}^{n} \frac{L - d_i + y_i}{y_i} \cdot x_i \cdot e_i \quad \text{since } d_i = y_i \\ &\geq \sum_{i=1}^{n} f\left(\frac{L - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i. \end{aligned}$$

Equation (4) computes processor utilization for the task set $\mathcal{T}$ and is a generalization of the EDF feasibility condition $\sum_{i=1}^{n} \frac{e_i}{y_i} \leq 1$ for independent tasks with deadlines equal to their period given by Liu & Layland [20].

4.4. Step 4: Analytical Verification of Latency Requirements

Latency is the delay between when an input node produces a sample (or message), which consists of $prd(q)$ tokens, and when the graph generates a corresponding output. The total latency encountered by a sample (or message) is an integral unit of time created by the sum of the latency inherent in the processing graph and the additional latency imposed by the implementation. *Inherent latency* in a graph is created by non-unity dataflow attributes and the graph topology. Inherent latency exists even if the graph is executed on an infinitely fast machine. *Imposed latency* comes from the scheduling and execution of nodes in the graph since we do not have an infinitely fast machine. Thus latency has two components, and the total latency any sample encounters can be expressed with the simple equation

$$\textit{Total Latency} = \textit{Inherent Latency} + \textit{Imposed Latency}.$$

We now show how to bound the maximum latency any sample (or message) will encounter in the synthesized system. If this bound is less than or equal to the latency requirement for each path in the graph, we can guarantee that the application will always meet its latency requirements.

In [14], we showed inherent latency can be calculated in cyclic processing graphs using Equations (5) and (6) of Theorem 4.2.

Theorem 4.2. *Let* $G = (V, E, \psi)$ *be a cyclic PGM graph with rate-based input nodes. Let* $w \in \mathcal{O}$, *and let the execution rate of input node* $j \in \mathcal{I}_w$ *be* $R_j = (x_j, y_j)$. *Let length*(q) *denote the current number of tokens in queue* $q \in E$. *Let* $\hat{\mathcal{P}}$ *denote the set of acyclic paths from input node* j *to node* w. *Let every back edge be initialized such that it is always over threshold. The inherent latency a sample will encounter is bounded such that*

$$\max_{p \in \hat{\mathcal{P}}} \left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j\right) \leq \textit{ Sample Latency} < \max_{p \in \hat{\mathcal{P}}} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j\right) \qquad (5)$$

where p represents a path $u \rightsquigarrow w$ *and* $F_{u \rightsquigarrow w}$ *is defined as*

$$F_{u \rightsquigarrow w} = \begin{cases} \max\left(0, \left\lceil \frac{thr(q) - length(q)}{prd(q)} \right\rceil\right) & \text{if } \exists q : \psi(q) = (u, w) \\ \max\left(0, \left\lceil \frac{(F_{v \rightsquigarrow w} - 1) \cdot cns(q) + thr(q) - length(q)}{prd(q)} \right\rceil\right) & \\ \quad \text{if } \exists q : \psi(q) = (u, v) \wedge v \neq w \wedge \; F_{v \rightsquigarrow w} > 0 & \\ 0 \quad \text{if } \exists q : \psi(q) = (u, v) \wedge v \neq w \wedge \; F_{v \rightsquigarrow w} = 0 & \end{cases} \qquad (6)$$

Equation (6) computes the number of times input node j must execute before enough data is produced to execute output node w. Equation (5) then uses this value to bound the interval of time in which node w will next be eligible to execute, which is the inherent latency a signal encounters in path $j \rightsquigarrow w$.

Using our synthesis method to transform a processing graph into a real-time system, managing imposed latency is a straightforward process. Moreover, unlike the computation of inherent latency, computing imposed latency is easy.

Inherent latency is the delay between when a sample is produced by graph input node j and when graph output node w executes under the strong synchrony hypothesis. (Under the strong synchrony hypothesis from the synchronous programming literature [8], the system instantly reacts to external stimuli, as though the application were executing on an infinitely fast machine.) Thus, imposed latency is the delay between when node w executes under the strong synchrony hypothesis and when it actually finishes executing in an actual implementation. Under RBE-EDF scheduling, which uses release time inheritance, a node's logical release time is equal to the time it would be released (and execute) under the strong synchrony hypothesis. By Theorem 4.1, if the graph is schedulable with the RBE-EDF scheduling algorithm (i.e., Equation (3) results in the affirmative), every released node v finishes its execution within d_v time units of its logical release. Thus, the upper bound on imposed latency incurred by a sample produced by input node j and consumed by output node w is equal to d_w. Moreover, as shown next, computing total latency has now been reduced to adding d_w to our bound for inherent latency.

Lemma 4.3. *Let $G = (V, E, \psi)$ be a PGM graph. Let $\mathcal{T}$ be an RBE task set synthesized from graph G. Let j be a graph input node for which there exists a path to graph output node w. If $\mathcal{T}$ is schedulable by Equation* (3)*, then the maximum imposed latency a sample incurs along the path $j \rightsquigarrow w$ is less than or equal to d_w.*

Proof: Since RBE-EDF uses release-time inheritance, each task's logical release time is equal to its release time (and execution) under the strong synchrony hypothesis. Thus, the maximum imposed latency a sample incurs along the path $j \rightsquigarrow w$ is determined by when node w finishes executing. An affirmative result from Equation (3) means that every released task v will finish executing within d_v time units of its logical release time. Thus, output node w will finish executing within d_w time units of its logical release time, and the maximum imposed latency a sample incurs along the path $j \rightsquigarrow w$ is less than or equal to d_w. □

Theorem 4.4. *Let $G = (V, E, \psi)$ be a PGM graph. Let $\mathcal{T}$ be an RBE task set synthesized from graph G. Let j be a graph input node for which there exists a path to graph output node w. If $\mathcal{T}$ is schedulable by Equation* (3)*, then the latency a sample incurs along the path $j \rightsquigarrow w$ under RBE-EDF scheduling is bounded such that*

$$SampleLatency < \textit{Inherent Latency Upper Bound} + d_w.$$

Proof: The maximum latency a sample incurs is bounded from above by the upper bound on inherent latency plus the upper bound on imposed latency. Thus, by Lemma 4.3 and the fact that inherent latency is always less than the *Inherent Latency Upper Bound*,

$$\textit{Sample Latency} < \textit{Inherent Latency Upper Bound} + d_w.$$ □

Corollary 4.5. *Let $G = (V, E, \psi)$ be a cyclic PGM graph. Let $\mathcal{T}$ be an RBE task set synthesized from graph G. Let $\mathcal{I}_w$ be the set of nodes producing data for output node $w \in \mathcal{O}$. Let $R_j = (x_j, y_j)$ be a well-defined execution rate for node $j \in \mathcal{I}_w$ starting at time 0. Let $\hat{\mathcal{P}}$ denote the set of acyclic paths from input node j to node w for all $j \in \mathcal{I}_w$. If $\mathcal{T}$ is schedulable by Equation* (3)*, then the latency a sample will incur is bounded such that*

$$\textit{Sample Latency} < \max_{p \in \hat{\mathcal{P}}} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j \right) + d_w. \quad (7)$$

Thus, verifying that the application meets its latency requirements has been reduced to ensuring that the right-hand side of Equation (7) is less than or equal to the latency requirement for each path in the processing graph. Of course Corollary 4.5 can also be used to determine the deadline parameter to be used for each node in the graph during Step 2. For example, suppose the maximum latency a signal (or message) encounters from an input node j to output node w must be less than k. Let $\hat{d_w} = k - \max_{p \in \hat{\mathcal{P}}} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j \right)$. If the graph is schedulable with the deadline parameter of each node u set such that $d_u = \min(y_u, \hat{d_w})$, then we can be sure that the latency requirement will be met.

5. Case Study

We demonstrate our synthesis method with a signal processing graph in an anti-submarine warfare (ASW) system—the Directed Low Frequency Analysis and Recording (DIFAR) acoustic signal processing graph from the Airborne Low Frequency Sonar (ALFS) subsystem of the LAMPS MK III anti-submarine helicopter. The ALFS system processes low frequency signals received by sonobuoys in the water. Its primary function is to detect submarines and to calculate range and bearing estimates to each target. Our example uses the portion of the DIFAR graph shown in Figure 2, which is an abstract representation of a one-band DIFAR graph [15]. The actual processing performed by the DIFAR graph is classified by the U.S. Government. However, an unclassified and abbreviated description of the graph was presented in [14]. An understanding of the actual processing is not necessary to follow our synthesis example.

The DIFAR graph shown in Figure 2 is a cyclic graph with with 31 nodes and 59 queues. All queues have unity produce, consume, and threshold attributes unless otherwise labeled. Non-unity produce values are labeled near the tail of the queue, and non-unity threshold and consume values are labeled near the head of the queue. The dataflow attributes used here are not the actual values from the graph

(the actual values are classified). However, the ratio between the attributes of a queue is the same. For example, if queue q had a produce of 1024 tokens; a threshold of 2048 tokens; and a consume of 1024 tokens, these values would be represented as: $prd(q) = 1$, $thr(q) = 2$, and $cns(q) = 1$. All back edges, including self-loop edges, are initialized so that they are over threshold. The number of initial tokens is shown on all queues that are initialized except self-loop edges. Self-loop edges are initialized so that they are always over threshold, but the number of initial tokens is not shown to reduce clutter in the figure.

The results presented here are from a study conducted under contract to General Dynamics to determine the number of 200 MHz PowerPC processors that are needed to meet seven different ALFS worst-case concurrent processing requirements [11]. One of the concurrency modes supports processing data from 16 different sonobuoys simultaneously. The actual input data rates and the specific latency requirements are classified.

Step 1: Computing Node Execution Rates. Let $R_{Source} = (16,\ 625ms)$ be a well-defined rate specification for input node *Source* beginning at time 0. That is, node *Source* delivers 16 samples of the signal (tokens) in every interval of $625ms$. Table 1 lists, in topological order, the rate specifications for the other nodes in the graph derived using Equation (1). (We showed several examples of computing node execution rates for this graph using Equation (1) in [14]. We omit those details here for space considerations.)

Step 2: Map Nodes to Tasks in the RBE Model. Table 1 lists the RBE parameters associated with each node when it is mapped to an RBE task. Parameters x_u and y_u are as derived in the rate computation step. Parameter d_u is set to y_u for each node u in the graph. Parameter e_u is the worst-case execution time for node u on a 200MHz PowerPC processor.

Step 3: Verify Schedulability. The third step of the synthesis method is to verify that the resulting task set is schedulable so that we can guarantee real-time execution. By Theorem 4.1, the RBE task set constructed from the DIFAR graph is schedulable using RBE-EDF scheduling if an affirmative result is obtained when the following scheduling condition is evaluated:

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

where $f(a)$ is the floor function defined in Theorem 4.1. Since $d_u = y_u$ for every node u in the graph, we again use the simpler utilization expression of Equation (4) to evaluate the schedulability of the graph under RBE-EDF scheduling. Using the RBE parameters from Table 1, we see that the graph is schedulable since $\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} = .0639 \leq 1$. Thus, since the processor utilization is less than one, the graph is schedulable with $d_u = y_u$ for each node u in the graph.

Note, however, that this graph only processes one band of one sonobuoy. If data from all 16 sonobuoys is processed simultaneously, then 16 instances of the graph are required, which results in a cumulative processor utilization of 1.0224. Thus, not all 16 instances of the graph can be executed simultaneously on the same processor. Moreover, while theoretically we can execute with the processor 100% loaded, the U.S. Navy has a requirement that limits resource utilization to 80% in new applications. The processor utilization limit of 80% provides room for application enhancements as well as a margin of error for safety. Thus, at most twelve instances of the graph may be executed on a single processor given the deadline parameters we have selected.

If $d_u = y_u$ for each node u and the graph is not schedulable, then relaxing any of the deadline parameters will not change the schedulability of the graph since increasing deadline parameters in this case does not reduce utilization. A negative result from Equation (4) when $d_u \geq y_u$ means that the processor is over loaded (i.e., the processor utilization is greater than 100%).

Step 4: Verifying Latency. As with all graphs in which each queue q is initialized with at least $thr(q) - cns(q)$ tokens, the first sample produced encounters the maximum latency [12]. Thus, to verify the latency requirement, only the latency for the first sample needs to be checked. However, as there are six graph output nodes, the latency of the first sample reaching each output node must be checked.

By Corollary 4.5, the latency between the time the first sample arrives and when output node *AliOut* executes is less than

$$\begin{aligned}
&\max\left(1, \left\lceil \frac{F_{Source \rightsquigarrow AliOut}}{x_{Source}} \right\rceil \cdot y_{Source}\right) + d_{AliOut} \\
&\quad = \max\left(1, \left\lceil \frac{256}{16} \right\rceil \cdot 625ms\right) + 10000ms \\
&\quad = 16 \cdot 625ms + 10000ms \\
&\quad = 20000ms \\
&\quad = 20 \text{ seconds}
\end{aligned}$$

when all of the deadline parameters in the path from node *Source* to node *AliOut* are less than or equal to d_{AliOut}. Thus, we can manage the amount of latency any sample encounters by choosing appropriate deadline values for node *AliOut* and its predecessors. For example, if $d_{AliScale}$, d_{AliMrg}, and d_{AliOut} were reduced to $2,500ms$, the maximum latency a sample encounters from node *Source* to node *AliOut* is less than 12.5 seconds.

The maximum latency the first sample encounters in the path from node *Source* to each of the other output nodes is

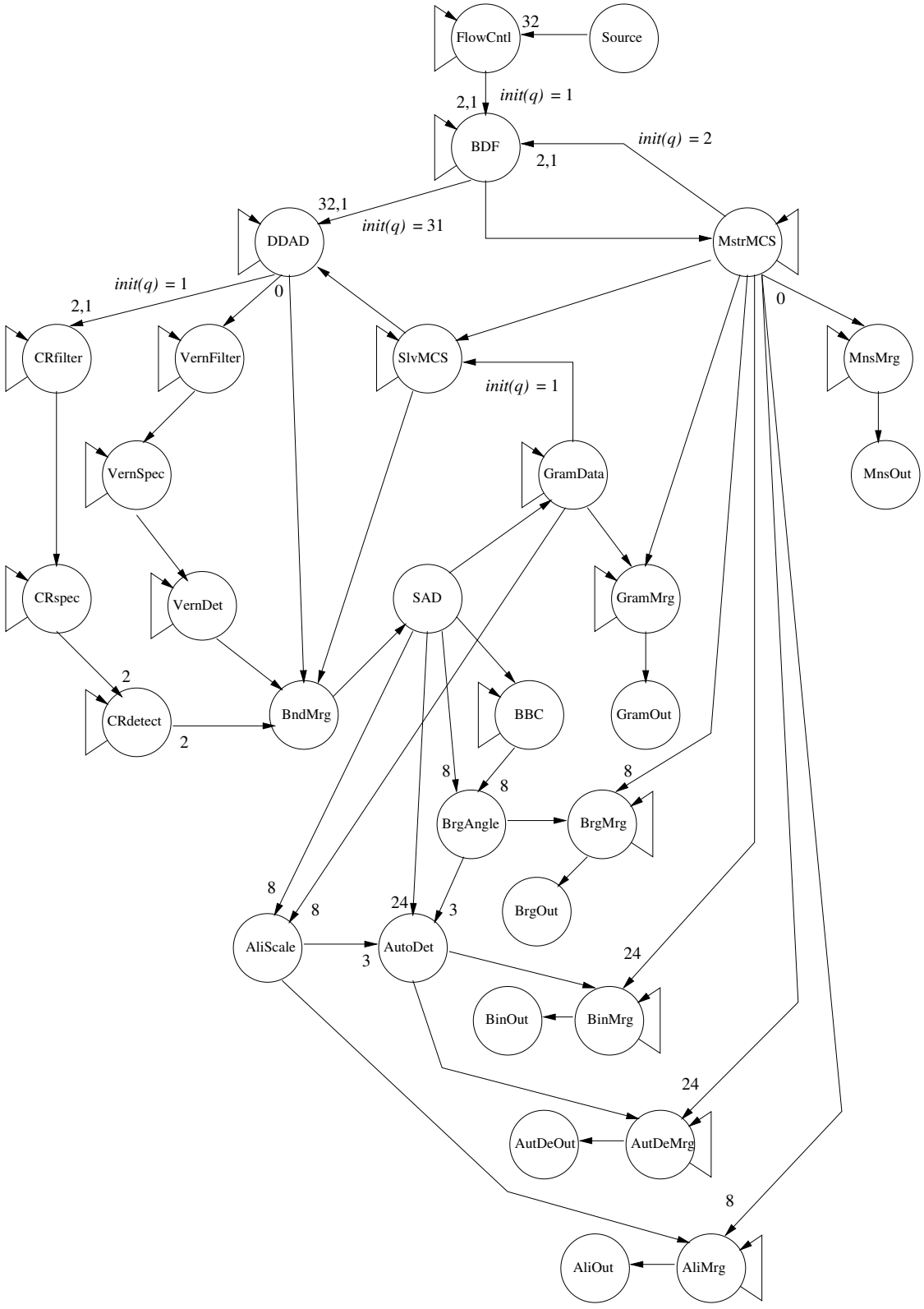

Figure 2. The PGM DIFAR Graph. All back edges, including self-loop edges, are initialized so that they are always over threshold.

Node	(x_u,	y_u,	d_u,	e_u)	Node	(x_u,	y_u,	d_u,	e_u)
Source	(16,	625*ms*)	—	—	AliScale	(1,	10000*ms*,	10000*ms*,	3.19*ms*)
FlowCntl	(1,	1250*ms*,	1250*ms*,	6.46*ms*)	AliMrg	(1,	10000*ms*,	10000*ms*,	0.51*ms*)
BDF	(1,	1250*ms*,	1250*ms*,	30.13*ms*)	AliOut	(1,	10000*ms*)	—	—
MstrMCS	(1,	1250*ms*,	1250*ms*,	0.34*ms*)	BBC	(2,	2500*ms*,	2500*ms*,	5.19*ms*)
MnsMrg	(0,	1250*ms*,	1250*ms*,	0.75*ms*)	BrgAngle	(1,	10000*ms*,	10000*ms*,	5.11*ms*)
MnsOut	(0,	1250*ms*)	—	—	BrgMrg	(1,	10000*ms*,	10000*ms*,	0.59*ms*)
SlvMCS	(1,	1250*ms*,	1250*ms*,	0.1*ms*)	BrgOut	(1,	10000*ms*)	—	—
DDAD	(1,	1250*ms*,	1250*ms*,	6.05*ms*)	AutDet	(1,	30000*ms*,	30000*ms*,	2.5*ms*)
CRfilter	(1,	1250*ms*,	1250*ms*,	8.7*ms*)	AutDetMrg	(1,	30000*ms*,	30000*ms*,	0.69*ms*)
CRspec	(1,	1250*ms*,	1250*ms*,	9.2*ms*)	AutDetOut	(1,	30000*ms*)	—	—
CRdetect	(1,	2500*ms*,	2500*ms*,	3.37*ms*)	BinMrg	(1,	30000*ms*,	30000*ms*,	0.2*ms*)
BndMrg	(2,	2500*ms*,	2500*ms*,	3.22*ms*)	BinOut	(1,	30000*ms*)	—	—
SAD	(2,	2500*ms*,	2500*ms*,	3.52*ms*)	VernFilter	(0,	1250*ms*,	1250*ms*,	2.92*ms*)
GramData	(2,	2500*ms*,	2500*ms*,	3.64*ms*)	VernSpec	(0,	1250*ms*,	1250*ms*,	3.08*ms*)
GramMrg	(2,	2500*ms*,	2500*ms*,	0.15*ms*)	VernDet	(0,	1250*ms*,	1250*ms*,	1.18*ms*)
GramOut	(2,	2500*ms*)	—	—					

Table 1. RBE parameters associated with each node in the DIFAR graph for the CR mode of operation. For each node u in the graph, $d_u = y_u$.

computed in the same manner. Using the RBE parameters in Table 1, the maximum latency from node *Source* to node:

- *GramOut* is five seconds,
- *BrgOut* is 20 seconds,
- *AutDetOut* is 60 seconds, and
- *BinOut* is 60 seconds.

At first it is rather surprising that latency as high as 60 seconds is acceptable in an embedded application. Acoustic signal processing applications require much higher latency bounds than other real-time applications such as radar applications. The main reason for this is that sound waves travel much slower than radar waves, and, thus, it takes longer to accumulate acoustic samples than radar samples — at least 30 seconds must elapse before enough data is available to execute some of the DIFAR signal processing functions. Consequently, the high latency is due to the time it takes for data to accumulate in a node's input queues (where it is buffered) until enough data exists for the node to execute.

6. Summary

We combined software engineering techniques with real-time scheduling theory to develop a synthesis method for transforming a processing graph into a predictable real-time system in which latency can be managed. The synthesis of real-time systems from PGM graphs involves four steps:

1. Identification of the rates at which nodes in a PGM graph must execute if they are to process data in real time.
2. Construction of a mapping of each node to a task in the RBE task model so that real-time processing can be achieved.
3. Verification that the resulting task set is schedulable so that we can guarantee real-time execution.
4. Analytical verification that latency requirements of the application are met.

Latency has two components: inherent latency and imposed latency. We used real-time scheduling theory to bound imposed latency and combined these results with our prior results on inherent latency to provide an upper bound on the total latency any sample or message will encounter in the synthesized system. If this bound is less than or equal to the latency requirement for each path in the graph, we can guarantee that the application will always meet its latency requirements. This is the first time imposed latency has been computed for cyclic processing graphs. Moreover, our latency results are novel in that they assume nodes are eligible for execution as soon as all required input data is available and then scheduled with the dynamic on-line RBE-EDF scheduler.

We demonstrated our synthesis method with DIFAR acoustic signal processing graph from the ALFS subsystem of the LAMPS MK III anti-submarine helicopter. For simplicity and concreteness, we present our synthesis method in terms of the U.S. Navy's PGM [23] and signal processing applications. However, our synthesis method applies to any general processing graph paradigm and many application domains.

References

[1] 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.

[2] 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.

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

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

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

[6] 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.

[7] Dasdan, A., Ramanathan, D., Gupta, R.K., "A Timing-Driven Design and Validation Methodology for Embeded Real-Time Systems," *ACM Trans. Design Automaton of Electronic Systems* (HLDVT'97 Special Issue), 3(4), Oct. 1998.

[8] 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.

[9] 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.

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

[11] Goddard, S., "Graph Performance Analysis Report on the ALFS Worst-Case Concurrency Modes," Technical Report 300832-980514-01, S.M. Goddard & Co., Inc., under contract to General Dynamics, May 14 1998.

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

[13] Goddard, S., Jeffay, K. "Managing Memory Requirements in the Synthesis of Real-Time Systems from Processing Graphs," *Proc. of IEEE Real-Time Technology and Applications Symposium*, June 1998, pp. 59-70.

[14] Goddard, S., Jeffay, K. "Analyzing the Real-Time Properties of a U.S. Navy Signal Processing System," *Proceedings of the Fourth IEEE International Symposium on High Assurance Systems Engineering*, Nov. 1999, pp. 141-150.

[15] *Airborne Low Frequency Sonar Subsystem System Requirements Specifications*, prepared by Hughes Aircraft Corporation, Version 1.0, Apr. 1991.

[16] *System/Segment Specificaton for the Airborne Low Frequency Sonar (ALFS) (Dipper & Integrated Sonobuoy)*, prepared by Hughes Aircraft Corporation, Aeorspace & Defense Sector, Document Number SS12070, Revision D, April 1994.

[17] 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.

[18] Jeffay, K., Goddard, S., "A Theory of Rate-Based Execution," *Proceedings of the 20th IEEE Real-Time Systems Symposium*, Dec. 1999, pp. 304-314.

[19] Karp, R.M., Miller, R.E., "Properties of a model for parallel computations: Determinacy, termination, queuing," *SIAM J. Appl. Math*, 14(6), 1966, pp 1390-1411.

[20] 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.

[21] 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.

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

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

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

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

[26] 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.

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

[28] 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.

[29] 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.

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