In: *International Journal of Reliability, Quality and Safety Engineering*, Special Issue for top ranked HASE'99 papers, Vol. 7, No. 4, 2000.

Analyzing the Real-Time Properties of a U.S. Navy Signal Processing System

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

The state of the art in verifying the real-time requirements of applications developed using general processing graph models relies on simulation or off-line scheduling. We extend the state of the art by presenting analytical methods that support the analysis of cyclic processing graphs executed with on-line schedulers. We show that it is possible to compute the latency inherent in a processing graph independent of the hardware hosting the application. We also show how to compute the real-time execution rate of each node in the graph. Using the execution rate of each node and the time it takes per execution on a given processor, the resulting CPU utilization can be computed, as shown here for 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.

1. Introduction

We present the analysis and verification of the real-time properties of an embedded signal processing application for an anti-submarine warfare (ASW) system. More specifically, we study the CPU requirements and inherent processing latency of 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. The ALFS system processes low frequency signals received by sonobuoys in the water. Its primary function is to detect and track submarines and to calculate range and bearing estimates to each target [14].

The DIFAR application was developed using the U.S. Navy's Processing Graph Method (PGM) [17], and executes on the U.S. Navy's standard signal processing computer, the AN/UYS-2A. PGM is one of many application development paradigms based on directed graphs called *processing graphs*, which are a standard design aid in the development of complex digital signal processing 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. Each data element that is processed by a node is a sample of the signal — an element of the discrete sequence of numbers representing the signal.

General processing graph paradigms, such as PGM, have been used to create a wide variety of applications (e.g, command and control, distributed multimedia, and signal processing applications). While this paper focuses on a specific acoustic signal processing application, the analysis presented here is applicable to any application developed using a general processing graph model such as PGM.

The state of the art in verifying the real-time requirements of applications developed using general processing graph models relies on one of two techniques. The first simulates graph execution, and hopes that the simulation encounters the worst case scenario (i.e., that the simulated graph execution is performed long enough to encounter the peak processor demand). This technique is generally applied when dynamic scheduling is used for graph execution. The second technique is applied when static scheduling is used to determine the order of node executions. An "off-line" algorithm creates a node execution schedule that is repeated periodically. The length of the schedule (i.e., the period of the schedule) determines the latency and memory usage of the application. This technique requires the generation and on-line storage of a large number of schedules for the various modes of operation of the ALFS subsystem. Consequently, the ALFS system uses on-line scheduling rather than static scheduling. This means, however, that it has not been possible to verify the real-time processing requirements of the ALFS signal processing graphs.

We extend the state of the art in verifying the real-time requirements of applications developed using processing graph models by presenting analytical methods that support the analysis of cyclic processing graphs executed with

on-line schedulers, such as the DIFAR graph. In [11], we proposed a new synthesis technique for building hard real-time signal processing systems from general processing graphs, and demonstrated its use with a synthetic aperture radar (SAR) system, an International Maritime Satellite (INMARSAT) mobile satellite receiver application, and an acoustic digital signal processing application. Here we use some of those techniques to calculate the CPU utilization of the DIFAR graph executing on the AN/UYS-2A and to compute the inherent latency in the signal processing graph. Inherent latency is created by non-unity dataflow attributes and graph topology. We show that both the CPU utilization and the inherent latency of the signal processing graph can be analytically derived. Moreover, we show that the inherent latency in the signal processing graph is independent of the scheduling algorithm and even of the number of processors used to execute the graph.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 presents background knowledge on PGM and the real-time analysis techniques that are applied to the DIFAR graph in Section 4. We conclude our analysis of the real-time properties of the DIFAR graph with a summary in Section 5.

2. Related Work

Our previous work on the synthesis of real-time uniprocessor systems from PGM was based on acyclic PGM graphs [1, 9, 10]. In this paper, we present the analyses of a cyclic PGM graph. This is the first time general execution rates and inherent latency have been computed for cyclic graphs. Since we do not assume the existence of a real-time scheduler or even knowledge of the type of scheduling performed during graph execution, we do not, in this paper, address latency imposed by the scheduling and execution of the graph nodes. A complete discourse on latency in processing graphs is contained in [11].

From the real-time literature, PGM graphs are most closely related to the Logical Application Stream Model (LASM) [5] and the Generalized Task Graph (GTG) model [6]. PGM, LASM, and GTG were all developed independently and support very similar dataflow properties. PGM was the first of these to be developed. Our work improves on the analysis of LASM and GTG graphs by not requiring periodic execution of the nodes in the graph. Instead, we calculate a more general execution rate, which can be reduced to average execution rates assumed in the LASM and GTG models. Our general execution rate specification provides a more natural representation of node execution for PGM graphs. Forcing periodic execution of all graph nodes adds latency to the processed signal, but simplifies the analysis of latency and memory requirements. In [10], we model PGM node execution with the more natural execution rate *and* improve memory usage analysis by showing that dynamic scheduling with execution rates results in less memory usage than periodic or static scheduling.

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 [16] supported by the Ptolemy system [4]. 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 [16] 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 [2]. 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 [16, 18, 20, 19, 2]. Off-line schedulers create a static node execution schedule that is executed periodically by the processor. In contrast, the primary goal of our research has been to manage the latency and memory usage of processing graphs by executing them with an on-line scheduler. Recently we have shown that for a large class of applications, dynamic on-line scheduling creates less imposed latency than static scheduling. An even more surprising result is that, in many cases, dynamic on-line scheduling uses less memory for buffering data on graph edges than static scheduling [10].

Our latency analysis is related to the work of Gerber *et al.* in guaranteeing end-to-end latency requirements on a single processor [8]. 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. Background & Analysis Techniques

A basic understanding of PGM and the theory of real-time graph execution is necessary before one can understand our analysis of the real-time properties of the DIFAR graph. Thus, this section presents a brief overview of PGM followed by the theory supporting our analysis techniques.

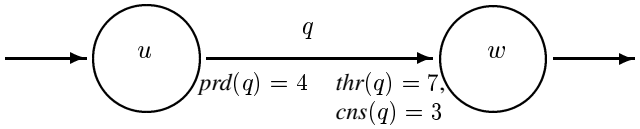

Figure 1. A two node chain.

3.1. Processing Graph Method

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.

3.2 Real-Time Graph Execution Theory

Embedded signal processing systems receive a continuous signal from external sensors. They are required to process the signal in real time and present the signal processing results to an output device within a specified time interval. Processing the signal in real time requires executing the PGM graph nodes so that they execute their processing functions as the signal arrives and without losing data. For example, some of the ALFS signal processing graphs are used to track submarines by calculating the distance, speed, and direction of a submarine. External sensors, called sonobuoys, convert the sound wave created by a submarine to a digital signal that is input to a PGM graph. The graph must process the signal and send the results, such as updated distance, speed, and direction, to a display before the next portion of the signal is sent by the sonobuoys.

The notation and terminology of this paper, for the most part, is an amalgamation of the notation and terminology used in [3] 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 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 u to $w \in \mathcal{O}$.

To simplify the presentation of execution rates and inherent latency, we assume the graph executes on an infinitely fast machine so that node execution takes no time. More precisely, we assume nodes execute in accordance with the strong synchrony hypothesis from the synchronous programming literature [7]. The strong synchrony hypothesis states that the system instantly reacts to external stimuli, which lets us define rate executions in the absence of scheduling algorithms.

Definition 3.1. An *execution rate* is an integer pair (x, y).

Definition 3.2. An execution rate specification for node v, $R_v = (x, y)$, is *valid* if there exists a time t such that node v executes exactly x times in time intervals $[t + (k-1)y,\ t + ky)$ for all $k > 0$.

Corollary 3.1. *If $R_v = (x, y)$ is a valid rate specification for node v, then $m \cdot R_v = (m \cdot x, m \cdot y)$ is also a valid rate specification for node v.*

Proof: The proof follows immediately from Definition 3.2. □

In [9], we derived execution rates for nodes in a PGM chain. The execution rate theorem of that paper is reproduced here since we will use this result to extend the analysis to include nodes with multiple input queues.

Theorem 3.2. *Let $u \rightsquigarrow w$ be a PGM chain with $\psi(q) = (u, w)$, and let $R_u = (x_u, y_u)$ be a valid execution rate for node u. Under the strong synchrony hypothesis, the execution rate $R_w = (x_w, y_w)$, where*

$$x_w = \frac{prd(q)}{\gcd(prd(q) \cdot x_u, cns(q))} \cdot x_u \quad \text{and} \quad y_w = \frac{cns(q)}{\gcd(prd(q) \cdot x_u, cns(q))} \cdot y_u, \tag{1}$$

is a valid execution rate for node w.

Equation (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 $R_u = (x_u = 3, y_u = 16)$ for node u in Figure 1, the execution rate of the consumer node w is derived as follows:

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

Now consider a general PGM graph in which a node has multiple input queues, such as the graph in Figure 2. Node w is a consumer of data produced by both u and v; $\psi(\alpha) = (u, w)$ and $\psi(\beta) = (v, w)$ define two producer/consumer pairs. We use the notation $R_{w \leftarrow u} = (x_{w \leftarrow u}, x_{w \leftarrow u})$ to represent the execution rate of node w with respect to node u. With $R_u = (3, 16)$ (the execution rate of node u), $R_{w \leftarrow u}$ is derived using Equation (1) as shown above with $R_{w \leftarrow u} = (4, 16)$. Likewise with $R_v = (2, 12)$, $R_{w \leftarrow v}$ (the execution rate of node w with respect to node v) is:

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

Since node 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. Observe that although $R_{w \leftarrow u} \neq R_{w \leftarrow v}$, we do have $\frac{x_{w \leftarrow u}}{y_{w \leftarrow u}} = \frac{x_{w \leftarrow v}}{y_{w \leftarrow v}}$. Lemma 3.3 states that without this equality, it would be impossible to schedule a valid execution of the graph.

Lemma 3.3. *Using Equation* (1) *to evaluate the execution rate of node w with respect to its adjacent predecessors in the digraph, if a valid graph execution is possible using finite memory for buffering then $\frac{x_{w \leftarrow u}}{y_{w \leftarrow u}} = \frac{x_{w \leftarrow v}}{y_{w \leftarrow v}}$ for all u, v for which there exists queues α, β such that $\psi(\alpha) = (u, w)$ and $\psi(\beta) = (v, w)$.*

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

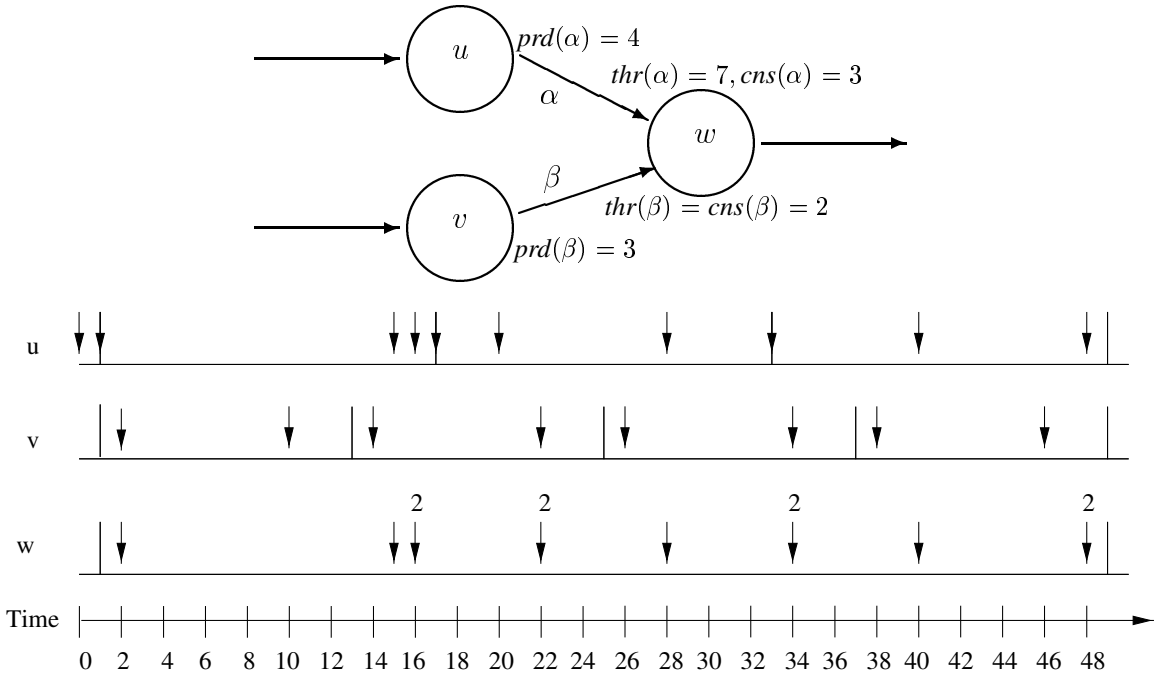

Figure 2. A three-node graph and time-line execution showing the execution of nodes under the strong synchrony hypothesis. If $R_u = (3, 16)$ and $R_v = (2, 12)$ are valid after time 0, then $R_w = (12, 48)$. Each down arrow represents an execution of the node. Multiple executions of a node at the same instant are represented by a number above the down arrow.

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

Theorem 3.4. *Let $G = (V, E, \psi)$ be an acyclic PGM digraph for which a valid execution is possible using finite memory and node $w \in V$ with $\delta^-(w) \geq 1$. The execution rate of node w is $R_w = (x_w, y_w)$ where*

$$y_w = \operatorname{lcm}\{\frac{cns(q)y_u}{\gcd(prd(q)x_u, cns(q))} \mid \psi(q) = (u, w)\},$$
$$x_w = y_w \cdot \left(\frac{prd(q)x_u}{cns(q)y_u}\right), \ \forall q, u : \psi(q) = (u, w). \quad (2)$$

Proof: The proof is constructed in two parts. In the first part we prove that for a (u, w) pair, with an adjoining queue α, Equation (2) derives a valid rate for node w with respect to node u. In the second part we show that the rate derived using the $\psi(\alpha) = (u, w)$ pair is the same for all $u \in V$ and $q \in E$ such that $\psi(q) = (u, w)$.

If node w were in a chain such that u and w were a producer/consumer pair $\psi(\alpha) = (u, w)$, then Equation (1) derives the execution rate of w with respect to u as $R_{w\leftarrow u} = (x_{w\leftarrow u}, y_{w\leftarrow u})$ where

$$x_{w\leftarrow u} = \frac{prd(\alpha)}{\gcd(prd(\alpha) \cdot x_u, cns(\alpha))} \cdot \ x_u$$
$$\text{and} \quad y_{w\leftarrow u} = \frac{cns(\alpha)}{\gcd(prd(\alpha) \cdot x_u, cns(\alpha))} \cdot y_u.$$

Let $R_w = (x_w, y_w) = (m_u \cdot x_{w\leftarrow u}, m_u \cdot y_{w\leftarrow u})$ where

$$m_u = \frac{\operatorname{lcm}\{\frac{cns(q)}{\gcd(prd(q)\cdot x_v, cns(q))} \cdot y_v \mid \psi(q) = (v, w)\}}{y_{w\leftarrow u}}.$$

By Theorem 3.2 and Corollary 3.1 if $R_{w\leftarrow u}$ is a valid rate specification for node w with respect to node u, then R_w is as well. We now reduce this expression to the form of

Equation (2):

$$\begin{aligned} y_w &= m_u \cdot y_{w \leftarrow u} \\ &= \frac{\operatorname{lcm}\{\frac{cns(q)}{\gcd(prd(q)\cdot x_v, cns(q))} \cdot y_v \mid \psi(q) = (v,w)\}}{y_{w\leftarrow u}} \cdot y_{w\leftarrow u} \\ &= \operatorname{lcm}\{\frac{cns(q)}{\gcd(prd(q)\cdot x_v, cns(q))} \cdot y_v \mid \psi(q) = (v,w)\}. \end{aligned}$$

Thus,

$$\begin{aligned} x_w &= m_u \cdot x_{w\leftarrow u} \\ &= \frac{\operatorname{lcm}\{\frac{cns(q)}{\gcd(prd(q)\cdot x_v, cns(q))} \cdot y_v \mid \psi(q) = (v,w)\}}{y_{w\leftarrow u}} \cdot x_{w\leftarrow u} \\ &= \frac{y_w}{y_{w\leftarrow u}} \cdot x_{w\leftarrow u} \\ &= \left(\frac{y_w}{\frac{cns(\alpha)}{\gcd(prd(\alpha)\cdot x_u, cns(\alpha))} \cdot y_u}\right)\left(\frac{prd(\alpha)\cdot x_u}{\gcd(prd(\alpha)\cdot x_u, cns(\alpha))}\right) \\ &= \frac{y_w}{cns(\alpha)\cdot y_u} \cdot prd(\alpha)\cdot x_u \\ &= y_w \cdot \left(\frac{prd(\alpha)\cdot x_u}{cns(\alpha)\cdot y_u}\right). \end{aligned}$$

We now show by contradiction that $\forall u \in V, q \in E$ such that $\psi(q) = (u,w)$ used to derive R_w, if the graph is valid, $R_w = (x_w, y_w)$ is a valid rate specification for node w. Suppose $\exists i,j \in V \wedge \exists \alpha,\beta \in E$ such that $\psi(\alpha) = (i,w)$, $\psi(\beta) = (j,w)$, and $R_w = (m_i \cdot x_{w\leftarrow i}, m_i \cdot y_{w\leftarrow i}) \neq (m_j \cdot x_{w\leftarrow j}, m_j \cdot y_{w\leftarrow j})$ where m_i and m_j are defined as above. Then $\frac{x_{w\leftarrow i}}{y_{w\leftarrow i}} \neq \frac{x_{w\leftarrow j}}{y_{w\leftarrow j}}$ since $\frac{m_k\cdot x_{w\leftarrow k}}{m_k\cdot y_{w\leftarrow k}} = \frac{x_{w\leftarrow k}}{y_{w\leftarrow k}}$. But if $\frac{x_{w\leftarrow i}}{y_{w\leftarrow i}} \neq \frac{x_{w\leftarrow j}}{y_{w\leftarrow j}}$ then, by Lemma 3.3, a valid graph execution is not possible with finite memory. Therefore if the graph is valid, $R_w = (x_w, y_w)$ derived with Equation (2) using the $\psi(\alpha) = (u,w)$ pair is the same for all $v \in V$ and $q \in E$ such that $\psi(q) = (v,w)$. □

It is reassuring (and necessary for correctness since Equation (1) and Equation (2) define minimal intervals in which a valid rate specification exists) that Equation (2) reduces to Equation (1) when a node has only one input queue:

$$\begin{aligned} y_v &= \operatorname{lcm}\{\frac{cns(q)}{\gcd(prd(q)\cdot x_u, cns(q))} \cdot y_u\} \\ &= \frac{cns(q)}{\gcd(prd(q)\cdot x_u, cns(q))} \cdot y_u \\ \implies x_v &= y_v \cdot \left(\frac{prd(q)\cdot x_u}{cns(q)\cdot y_u}\right) \\ &= \frac{cns(q)}{\gcd(prd(q)\cdot x_u, cns(q))} \cdot y_u \cdot \left(\frac{prd(q)\cdot x_u}{cns(q)\cdot y_u}\right) \\ &= \frac{prd(q)\cdot x_u}{\gcd(prd(q)\cdot x_u, cns(q))} \\ &= \frac{prd(q)}{\gcd(prd(q)\cdot x_u, cns(q))} \cdot x_u \end{aligned}$$

Therefore, Equation (2) is equal to Equation (1) when a node has only one input queue and either expression can be used to calculate the consumer node's execution rate.

We now return to the problem of finding the execution rate of node w in Figure 2. Since $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$, we can use Equation (2) to derive the execution rate of node w:

$$\begin{aligned} y_w &= \operatorname{lcm}\{\frac{cns(\alpha)\cdot y_u}{\gcd(prd(\alpha)\cdot x_u, cns(\alpha))}, \frac{cns(\beta)\cdot y_v}{\gcd(prd(\beta)\cdot x_v, cns(\beta))}\} \\ &= \operatorname{lcm}\{\frac{3}{\gcd(4\cdot 3,3)}\cdot 16, \frac{2}{\gcd(3\cdot 2,2)}\cdot 12\} \\ &= \operatorname{lcm}\{\frac{3}{\gcd(12,3)}\cdot 16, \frac{2}{\gcd(6,2)}\cdot 12\} \\ &= \operatorname{lcm}\{\frac{3}{3}\cdot 16, \frac{2}{2}\cdot 12\} \\ &= \operatorname{lcm}\{16,12\} = 48 \\ \implies x_w &= y_w \cdot \left(\frac{prd(\alpha)\cdot x_u}{cns(\alpha)\cdot y_u}\right) \\ &= 48 \cdot \left(\frac{4\cdot 3}{3\cdot 16}\right) = 48 \cdot \left(\frac{12}{48}\right) = 12 \\ &= y_w \cdot \left(\frac{prd(\beta)\cdot x_v}{cns(\beta)\cdot y_v}\right) \\ &= 48 \cdot \left(\frac{3\cdot 2}{2\cdot 12}\right) = 48 \cdot \left(\frac{6}{24}\right) = 12 \end{aligned}$$

Therefore $R_w = (x_w, y_w) = (12, 48)$.

For example, given nodes u and v in Figure 2 with $R_u = (3,16)$ and $R_v = (2,12)$, the execution rate of w is:

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

and $x_w = y_w \cdot \left(\frac{prd(\alpha)\cdot x_u}{cns(\alpha)\cdot y_u}\right) = 48 \cdot \left(\frac{4\cdot 3}{3\cdot 16}\right) = 12.$

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.

The processor utilization created by the execution of the nodes in a graph can be calculated using Equation (3), where e_u is the execution time of node u.

$$\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} \tag{3}$$

If $\sum_{i=1}^{n} \frac{x_i\cdot e_i}{y_i} \leq 1$, the nodes can be scheduled using a simple on-line scheduler such that the nodes execute at their required execution rate and no incoming data is lost [11].

Theorem 3.4 can also be applied to cyclic graphs if each back edge q in every cycle is initialized such that it is always over threshold. (A *back edge* is a queue q that joins node v to an ancestor w when the graph is topologically

sorted.) Let q be a back edge with $\psi(q) = (v, w)$. If it can be guaranteed that node v will always finish executing within d_v time units of when it is eligible for execution, we can guarantee back edge q will always be over threshold if it is initialized such that

$$\mathit{init}(q) = \left\lceil \frac{s_v + d_v - \bar{s}_w + y_v}{y_w} \right\rceil \cdot x_w \cdot \mathit{cns}(q) + \mathit{thr}(q) \tag{4}$$

where s_v is the latest possible time node v will first be eligible to execute, $s_v + d_v$ is the latest possible time node v will complete its first execution and $\bar{s}_w$ is the earliest possible time node w can begin its first execution [11]. Before we can present equations to compute s_v and $\bar{s}_w$, we need to introduce the concept of latency and how it is computed in PGM graphs.

A signal processing engineer describes latency as the time delay between the sampling of a signal and the presentation of the processed signal to the output device (which may be a screen, speaker, or another computer). While intuitive, this definition is not precise enough for our purposes since individual input samples cannot be identified in the *prd*(q) tokens produced at one time by an external source. We define a sample to be the set of tokens delivered by a source node at one time. Under the strong synchrony hypothesis, latency is the delay between when a source node produces a sample (*prd*(q) tokens) and when the graph outputs the processed signal. The total latency encountered by a sample is an integral unit of time created by the sum of the latency inherent in the signal 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

Total Latency = Inherent Latency + Imposed Latency.

Let node u be a source node in the set of graph source nodes $\mathcal{I}$, and let queue q be an output queue to source node u. Inherent latency is the delay between the enqueuing of *prd*(q) tokens onto queue q by source node u and the next execution of the sink node when the graph is executed on an infinitely fast computer, as assumed by the strong synchrony hypothesis. In simple dataflow models that require unity dataflow attributes and only allow one source node, the inherent latency of the graph is 0 under the strong synchrony hypothesis. However, non-unity dataflow values (as supported by PGM) or multiple source nodes can create significant latency in processing the signal, even if we have an infinitely fast computer. The inherent latency any sample encounters in a cyclic graph is bounded by Theorem 3.5.

Theorem 3.5. *Let $G = (V, E, \psi)$ be a cyclic PGM graph with rate-based source nodes. Let $w \in \mathcal{O}$, and let the execution rate of source 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 source 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*

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

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

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

Equation (6) computes the number of times source node j must execute before enough data is produced to execute sink 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 $j \rightsquigarrow w$.

We now have the necessary theory to show how s_v and $\bar{s}_w$ are computed such that we can guarantee that a back edge that joins node v to node w in a cycle is always over threshold. (Recall that s_v is the latest possible time node v will first be eligible to execute and $\bar{s}_w$ is the earliest possible time node w can begin its first execution.)

Theorem 3.6. *Let queue q be a back edge in a cycle with $\psi(q) = (v, w)$. Let $\hat{\mathcal{P}}_w$ denote the set of acyclic paths from source node j to node w. Let $\hat{\mathcal{P}}_v$ denote the set of acyclic paths from source node j to node v. If queue q is initialized with a number of tokens given by Equation* (4)*, where $\bar{s}_w$ is*

$$\bar{s}_w = \max_{p \in \hat{\mathcal{P}}_w} \left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j \right) \tag{7}$$

and s_v is

$$s_v = \max_{p \in \hat{\mathcal{P}}_v} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j \right), \tag{8}$$

then queue q will always be over threshold.

The proofs of Theorems 3.5 and 3.6 are in [11].

4. DIFAR Application

The analysis presented here is based on a portion of the actual DIFAR graph. However, the same analysis methods have been applied to the complete DIFAR graph, as well as all of the other graphs in the worst-case concurrency modes that the ALFS system must support [12]. The actual processing performed by the DIFAR graph is classified by the U.S. Government, so the following is an unclassified and abbreviated description of the graph [13]. An understanding of the actual processing is not necessary to compute CPU utilization or to analyze latency in the graph.

The DIFAR graph receives directed low frequency acoustic data from a sonobuoy and analyzes the data for possible targets, such as enemy submarines or surface ships. The DIFAR graph has over 80 nodes and 400 queues and operates in three different modes: constant percent resolution (CPR), constant resolution (CR), and vernier. The ALFS subsystem can execute many different graphs simultaneously on a distributed system of processors. One worst-case concurrency mode that it supports is the execution of 16 instances of the DIFAR graph, each processing data from one sonobuoy. The frequency spectrum of data received by the DIFAR graph is usually partitioned into bands, and the graph can be configured to process from one to eight bands. Thus, while the full DIFAR graph has over 85 nodes and 400 queues, there are many duplicate paths in the graph with each path operating on a different portion of the signal. The graph of Figure 3 is an abstract representation of a one-band DIFAR graph. It is a cyclic graph 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 processing specific to the modes CPR, CR, and vernier are located in the upper left portion of the graph in Figure 3. The CPR processing is performed by the node *DDAD* (DIFAR direction and detection filter). The CR processing is performed by the nodes *DDAD*, *CRfilter* (CR filter), *CRspec* (CR spectral analysis), and *CRdetect* (CR detection filter). The vernier processing is performed by the nodes *DDAD*, *VernFilter* (vernier filter), *VernSpec* (vernier spectral analysis), and *VernDet* (vernier detection filter). The node *BndMrg* (band merge) merges data from all of the active bands into one data stream. The DIFAR graph in Figure 3 only shows one processing band for each of the three modes. In the full DIFAR graph, there would be 8 sets of CPR, CR, and vernier nodes, each ready to process a separate band of data partitioned from the input signal by the node *BDF* (band definition filter). The heaviest processing load is created when the graph operates in CR mode. In this mode, no data is sent to the *Vernfilter* node. Thus, vernier processing is inactive in the CR mode, and nodes *Vernfilter*, *VernSpec*, and *VernDet* do not execute.

4.1. Node Execution Rates

Let $R_{Source} = (16,\ 625ms)$ be a well-defined rate specification for source 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 (2) of Theorem 3.4. Excluding self-loops, two back edges are detected with a topological sort of the graph: the queue connecting node *MstrMCS* to node *BDF*, which is initialized with one token, and the queue connecting node *GramData* to node *SlvMCS*, which is initialized with two tokens. However, before we can be guaranteed that the rate specifications derived using Equation (2) are well-defined, the number of initial tokens on both back edges must be increased so that they are guaranteed to always be over threshold. In the calculation of the number of tokens with which back edges must be initialized, assume $d_v = y_v$ for each node v attached to the tail of a back edge in a cycle. Let $\hat{\mathcal{P}}_w$ denote the set of acyclic paths from a node *Source* to node w, and $\hat{\mathcal{P}}_v$ denote the set of acyclic paths from source node *Source* to node v in the DIFAR graph. By Theorem 3.6, back edge q, connecting node v to node w, will always be over threshold if it is initialized with at least

$$\left\lceil \frac{s_v + d_v - \bar{s}_w + y_v}{y_w} \right\rceil \cdot x_w \cdot cns(q) + thr(q)$$

tokens where

$$s_v = \max_{p \in \hat{\mathcal{P}}_v} \left(1, \left\lceil \frac{F_p}{x_{Source}} \right\rceil \cdot y_{Source}\right), \text{ and}$$

$$\bar{s}_w = \max_{p \in \hat{\mathcal{P}}_w} \left(0, \left\lfloor \frac{F_p - 1}{x_{Source}} \right\rfloor \cdot y_{Source}\right).$$

Using these expressions and the rate specifications listed in Table 1 to compute the number of initial tokens on the queue connecting node v = *MstrMCS* to node w = *BDF*, the

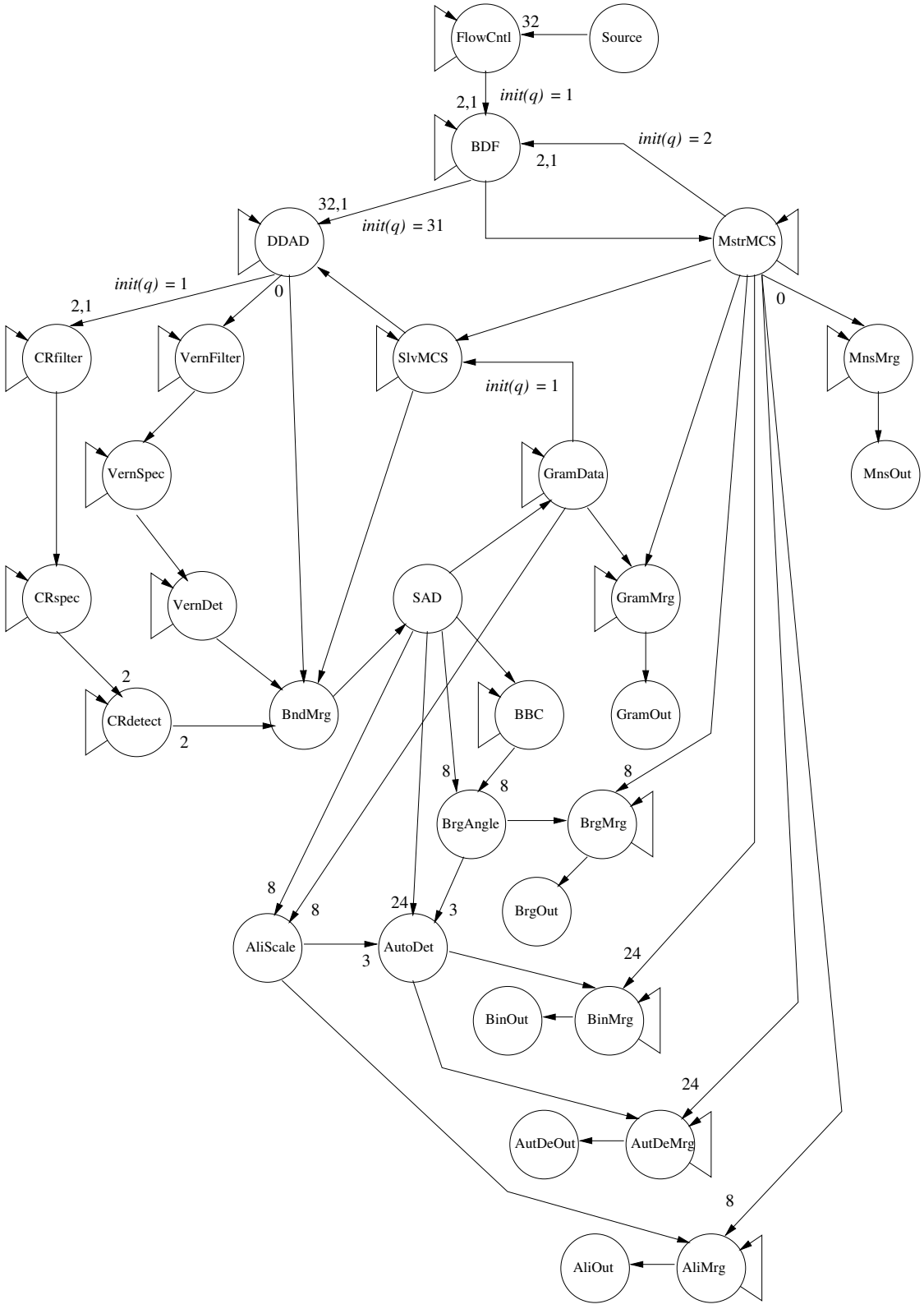

Figure 3. The PGM DIFAR Graph. All back edges, including self-loop edges, are initialized so that they are always over threshold.

queue must be initialized with at least

$$\left\lceil \frac{1250ms + 1250ms - 625ms + 1250ms}{1250ms} \right\rceil \cdot 1 \cdot 1 + 2 = 3 + 2 = 5$$

tokens since

$$s_{MstrMCS} = \max_{p \in \hat{\mathcal{P}}_{MstrMCS}} \left(1, \left\lceil \frac{F_p}{x_{Source}} \right\rceil \cdot y_{Source}\right) = \left\lceil \frac{32}{16} \right\rceil \cdot 625ms = 1250ms, \text{ and}$$

$$s_{BDF} = \max_{p \in \hat{\mathcal{P}}_{BDF}} \left(0, \left\lfloor \frac{F_p - 1}{x_{Source}} \right\rfloor \cdot y_{Source}\right) = \left\lfloor \frac{32 - 1}{16} \right\rfloor \cdot 625ms = 625ms.$$

Similarly, the number of initial tokens on the queue connecting node $v =$ *GramData* to node $u =$ *SlvMCS* must be at least

$$\left\lceil \frac{2500ms + 2500ms - 625ms + 2500ms}{1250ms} \right\rceil \cdot 1 \cdot 1 + 1 = 6 + 1 = 7$$

tokens since

$$s_{GramData} = \max_{p \in \hat{\mathcal{P}}_{GramData}} \left(1, \left\lceil \frac{F_p}{x_{Source}} \right\rceil \cdot y_{Source}\right) = \left\lceil \frac{64}{16} \right\rceil \cdot 625ms = 2500ms, \text{ and}$$

$$s_{SlvMCS} = \max_{p \in \hat{\mathcal{P}}_{SlvMCS}} \left(0, \left\lfloor \frac{F_p - 1}{x_{Source}} \right\rfloor \cdot y_{Source}\right) = \left\lfloor \frac{32 - 1}{16} \right\rfloor \cdot 625ms = 625ms.$$

The original implementation of the DIFAR graph on the U.S. Navy's standard signal processing computer, the AN/UYS-2A (a multi-processor computer), was scheduled with a non-preemptive first-come-first-served (FCFS) scheduler. The application had trouble meeting its latency requirement when multiple DIFAR graphs were executing at the same time [12]. It turns out that part of the problem was the initialization of the two back edges found during the topological sort of the graph. When the amount of initialized data was increased as described above, the application was determined by simulation to meet its latency requirement with a non-preemptive FCFS scheduler. However, this is not the same as a guarantee that it will always meet its latency requirement under non-preemptive FCFS scheduling. In contrast, after successfully completing synthesis method presented in [11], the DIFAR application can be guaranteed to always meet is latency requirement.

4.2. CPU Utilization

The last column of Table 1 lists the worst-case execution time of each node when it is executed on the AN/UYS-2A [12]. Using the node execution rates, the worst case execution time for each node, and Equation (3) (on page 6), the processor utilization for a single instance of the DIFAR graph executing on a single arithmetic processor of the AN/UYS-2A is 4.84%:

$$\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} = .0484.$$

However, 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 0.77. (The worst case concurrency requirement for the full 80 node DIFAR graph requires 5 bands to be processed for each sonobuoy, which results in a cumulative processor utilization of 2.11 [12]. Hence, the AN/UYS-2A used in the ALFS system has a total of 3 arithmetic processors.) In general, the analytical calculation of the CPU utilization is only as accurate as the execution times used in the computation. Since we are concerned with guaranteeing latency, we used worst case node execution times rather than average execution times.

We also measured the CPU utilization of one AN/UYS-2A arithmetic processor executing one instance of the one-band DIFAR graph using real-time data collection features of the AN/UYS-2A. The peak processor utilization measured was 4.7%, as compared to our predicted 4.84%. (The full DIFAR graph that processes 5 bands of sonobuoy data resulted in a peak processor utilization of 13.09%, as compared to a predicted 13.2% [12].) It should be noted that the worst case execution times are regularly encountered in this graph, and that the worst case execution times for the AN/UYS-2A are extremely accurate. Thus, it is not surprising that our predicted processor utilization values were so close to measured values.

4.3. Computing Inherent Latency

The worst case latency cannot be less than the inherent latency defined by the graph topology and dataflow parameters, no matter what type of hardware is used to host the application. In the DIFAR graph, the first sample produced encounters the maximum latency [11]. Thus, to verify a latency requirement, only the latency for the first sample needs to be checked. However, there are six graph sink nodes so the latency of the first sample must be checked at each graph sink node.

By Theorem 3.5, the latency between the time the first sample arrives and when sink node *AliOut* can first be eligi-

Node	t_u	(x_u,	y_u)	e_u	Node	t_u	(x_u,	y_u)	e_u
Source	0	(16,	$625ms$)	—	AliScale	0	(1,	$10000ms$)	$8.68ms$
FlowCntl	0	(1,	$1250ms$)	$2.02ms$	AliMrg	0	(1,	$10000ms$)	$0.18ms$
BDF	0	(1,	$1250ms$)	$25.62ms$	AliOut	0	(1,	$10000ms$)	—
MstrMCS	0	(1,	$1250ms$)	$0.13ms$	BBC	0	(2,	$2500ms$)	$4.08ms$
MnsMrg	0	(0,	$1250ms$)	$0.23ms$	BrgAngle	0	(1,	$10000ms$)	$16.67ms$
MnsOut	0	(0,	$1250ms$)	—	BrgMrg	0	(1,	$10000ms$)	$0.18ms$
SlvMCS	0	(1,	$1250ms$)	$0.07ms$	BrgOut	0	(1,	$10000ms$)	—
DDAD	0	(1,	$1250ms$)	$2.58ms$	AutDet	0	(1,	$30000ms$)	$3.22ms$
CRfilter	0	(1,	$1250ms$)	$4.12ms$	AutDetMrg	0	(1,	$30000ms$)	$0.13ms$
CRspec	0	(1,	$1250ms$)	$10.28ms$	AutDetOut	0	(1,	$30000ms$)	—
CRdetect	0	(1,	$2500ms$)	$1.37ms$	BinMrg	0	(1,	$30000ms$)	$0.07ms$
BndMrg	0	(2,	$2500ms$)	$0.01ms$	BinOut	0	(1,	$30000ms$)	—
SAD	0	(2,	$2500ms$)	$1.26ms$	VernFilter	0	(0,	$1250ms$)	N/A
GramData	0	(2,	$2500ms$)	$7.85ms$	VernSpec	0	(0,	$1250ms$)	N/A
GramMrg	0	(2,	$2500ms$)	$0.07ms$	VernDet	0	(0,	$1250ms$)	N/A
GramOut	0	(2,	$2500ms$)	—					

Table 1. DIFAR node execution rates and worst case execution times.

ble for execution is bounded such that

$$\max\left(0, \left\lfloor \frac{\mathcal{F}_{Source \rightsquigarrow AliOut} - 1}{x_{Source}} \right\rfloor \cdot y_{Source}\right) \leq \textit{Sample Latency} < \max\left(1, \left\lceil \frac{\mathcal{F}_{Source \rightsquigarrow AliOut}}{x_{Source}} \right\rceil \cdot y_{Source}\right)$$

$$\max\left(0, \left\lfloor \frac{256 - 1}{16} \right\rfloor \cdot 625ms\right) \leq \textit{Sample Latency} < \max\left(1, \left\lceil \frac{256}{16} \right\rceil \cdot 625ms\right)$$

$$9.375 \text{ seconds} \leq \textit{Sample Latency} < 10 \text{ seconds}.$$

Thus, no matter how fast the processor or how many are used, the minimum latency a sample encounters from the source node to node *AliOut* is 9.375 seconds and it may be almost 10 seconds. If we can guarantee that node *AliMrg* completes its execution within 10 seconds of when it is first eligible for execution (i.e., if 10 seconds is the bound on imposed latency), then we can guarantee that the maximum latency any sample encounters in the path is less than 20 seconds since total latency is equal to inherent latency plus imposed latency.

The maximum inherent latency the first sample encounters in the path from node *Source* to each of the other output nodes is computed in the same manner. The upper bound on inherent latency from node *Source* to: node *GramOut* is 2.5 seconds, node *BrgOut* is 10 seconds, node *AutDetOut* is 30 seconds, and node *BinOut* is 30 seconds. Thus, if the processing simply keeps up with the input data rates, total latency may be as high as 60 seconds on some paths.

At first it is rather surprising that latency as high as 60 seconds is tolerable in an embedded application. Acoustic signal processing applications can tolerate 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.

5. Summary and Conclusions

We presented the analysis and verification of the real-time properties of the DIFAR signal processing graph of the ALFS system using analytical techniques. Prior to this work, the two most common ways to verify the real-time requirements of applications developed using general processing graph models was to simulate graph execution or to create a static schedule off-line to determine the period of the schedule.

While the U.S. Navy has spent millions of dollars developing applications with PGM, it has never before been able to analytically verify the real-time requirements of PGM graphs. Thus, we claim to extend the state of the art in real-time analysis and verification by showing that it is possible to analytically compute the inherent latency of cyclic graphs independent of the hardware hosting the application. We also showed how to compute the real-time execution rate of each node in the graph. Using the execution rate of

each node and the time it takes per execution on a given processor, the resulting CPU utilization can also be computed, as shown here. Using a deterministic scheduling algorithm to bound imposed latency, it is also possible to bound total latency and memory requirements for any PGM graph [11].

The analysis presented here is based on a portion of the actual DIFAR graph. However, the same analysis methods have been applied to the complete DIFAR graph, as well as all of the other graphs in the worst-case concurrency modes that the ALFS system must support [12]. Moreover, our analysis methods are applicable to any application developed using a general processing graph model such as PGM.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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. *IEEE Real-Time Technology and Applications Symposium*, 1998

[11] 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

[12] 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.

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

[14] *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.

[15] 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.

[16] 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.

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

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

[19] 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.

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

Biographies

Steve Goddard (goddard@cse.unl.edu) is a J.D. Edwards Assistant Professor in the Department of Computer Science & Engineering at the University of Nebraska-Lincoln (UNL). He received the B.A. degree in computer science and mathematics from the University of Minnesota (1985). He received the M.S. and Ph.D. degrees in computer science from the University of North Carolina at Chapel Hill (1995, 1998). Dr. Goddard worked as a Systems Engineer with Unisys Corporation for four years and as a real-time, distributed, systems consultant with S.M. Goddard & Co., Inc. for nine years before joining the Computer Science & Engineering faculty at UNL in 1998. He is the founding co-director of the Advanced Networking and Distributed Experimental Systems (ANDES) Laboratory at UNL. His research and teaching interests are in real-time systems, distributed systems, operating systems, software engineering and computer networks.

Kevin Jeffay (jeffay@cs.unc.edu) is the S. Shepard Jones Distinguished Term Professor of Computer Science in the

Department of Computer Science at the University of North Carolina at Chapel Hill. He also serves as the Director of Undergraduate Studies for the Department. He received his Ph.D. from the University of Washington in 1989. His research and teaching interests are in real-time systems, operating systems, networking, and multimedia systems. He and his students focus on the general problem of providing real-time computation and communication services across the Internet. Dr. Jeffay is currently the editor-in-chief of the ACM/Springer-Verlag journal *Multimedia Systems*. He is the program chair of the 2000 IEEE Real-Time Systems Symposium and the 2000 International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV).