Feasibility concerns in PGM graphs with bounded buffers

Sanjoy Baruah
Department of Computer Science
The University of Vermont
Email: sanjoy@cs.uvm.edu

Steve Goddard **Kevin Jeffay**
Department of Computer Science
The University of North Carolina
Email: {goddard,jeffay}@cs.unc.edu

Abstract

The Processing Graph Method (PGM) — *a dataflow model widely used in the design and analysis of embedded signal-processing applications — is studied from a real-time scheduling perspective. It is shown that the problem of deciding if instances of the general model are feasible on a single processor is intractable (co-NP-complete in the strong sense); however, a useful special case is sometimes more tractable. An efficient feasibility test and an optimal preemptive scheduling algorithm are derived for this special case, and a procedure is presented which permits system architects to make efficient use of computational resources and memory requirements for buffers while constructing real-time dataflow applications that offer hard service guarantees.*

1. Introduction

Signal processing algorithms are often defined in the literature using large grain dataflow graphs [4]: directed graphs in which a node is a sequential program that executes from start to finish in isolation (i.e., without synchronization), and the graph edges depict the flow of data from one node to the next. Thus, an edge represents a producer/consumer relationship between two nodes. Large grain dataflow provides a natural description of signal processing applications with each node representing a mathematical function to be performed on an infinite stream of data that flows on the arcs of the graph. The streams of input data are typically generated by sensors sampling the environment at periodic rates and sending the samples to the signal processor via an external channel. The dataflow methodology allows one to easily understand the signal processing performed by depicting the structure of the algorithm; any portion of the application can be understood in the absence of the rest of the algorithm. Each node consumes the data produced by its predecessor in the graph and the data exists only between its production and consumption.

Embedded signal processing applications are naturally defined using dataflow techniques, but require the deterministic performance of real-time applications. The signal processing graph must process data at the rates of a set of producers (e.g., sonobuoys, dipping sonars, or radars) without the loss of data. Further, dataflow models implicitly define a temporal semantics of a processing graph by specifying lower bounds on when nodes may execute as a function of the availability of data on input edges. In order that system architects be able to successfully construct real-time dataflow applications that can offer run-time guaranteed service, it is critical that certain algorithmic tools and techniques be made available to them. These include:

- schedulability or admission-control tests — How does one determine if a set of nodes or a graph "fits" on a processor?
- upper bounds on queue length — How large must the storage buffers associated with each processing node be in order that no data loss occur due to inadequate storage?

Unfortunately, without the application of real-time scheduling theory to dataflow methodologies and a precise execution model, such tools, techniques, and methodologies have not been made available to the architects of real-time dataflow systems. Even the U.S. Navy's own dataflow methodology, *Processing Graph Method* (PGM) [1], lacks real-time analysis techniques to support making cost tradeoffs or to verify schedulability requirements. PGM is a U.S. Navy standard for developing real-time, embedded, anti-submarine warfare (ASW) applications for the AN/UYS-2A (the U.S. Navy's standard signal processor). The AN/UYS-2A is used in a number of U.S. Navy systems including airborne, surface, and sub-

surface platforms. Millions of dollars have been invested in developing the AN/UYS-2A and applications for it, yet the U.S. Navy has no way to guarantee that the hard real-time processing requirements of these ASW applications can be met.

Prior to [3], none of the dataflow models or real-time execution paradigms documented in the research literature were able to correctly model the execution of PGM applications. In [3], we identified inherent real-time properties of nodes in a PGM dataflow graph, and demonstrated how these properties can be exploited to perform useful and important system-level analyses such as schedulability anaysis, end-to-end latency analysis, and memory requirements analysis in single processor systems.

A primary focus in [3] was to bound latency and buffer requirements of a graph by assigning response times (relative deadlines) to nodes based on latency constraints and then bounding buffer requirements of the graph. We explore the complement of that problem in this paper. That is, we assume an implementation of a PGM graph has specified buffer constraints, and design a deadline assignment algorithm and a corresponding preemptive scheduling algorithm. We formally show that our deadline assignment and scheduling algorithms are *optimal* in the sense that when they are used to schedule the execution of any feasible PGM chain, no buffer overflow occurs (i.e., no data is lost).

Organization of this paper. The rest of this paper is organized as follows. In Section 2, we formally introduce the general PGM dataflow graph model, and define the **PGM dataflow graph feasibility problem.** In Section 3, we show that we are unlikely to be able to design efficient algorithms for solving this problem, by proving it to be co-NP-hard in the strong sense. In Section 4, we define PGM dataflow **chains** — a subclass of PGM dataflow graphs — and present a brief overview of the Synthetic Aperture Radar (SAR) graph from ARPA's Rapid Prototyping of Application Specific Signal Processors (RASSP) project. We discuss the issues of optimal scheduling and feasibility analysis for PGM dataflow chains in Section 5. We conclude in Section 6 with a summary of the major ideas presented herein.

2. PGM dataflow graphs

This section provides a formal description of the specification of a PGM dataflow graph we require to perform our analysis. See [1] for the actual PGM specification.

A **PGM dataflow graph** G is a directed graph with

- a subset S of the set of nodes designated as **source** nodes; each source node s is labeled with a *period* y_s, indicating that s fires at all time instants $y_s \cdot k$, $k \in \mathbf{N}$.
- for each node N_i, an non-negative parameter e_i called the *execution requirement* of N_i. The execution requirement e_i denotes the amount of time for which N_i must be assigned the processor each time it fires. (We assume that the scheduling model is *preempt-resume*; i.e., the execution of a node may be preempted at some point in time, and resumed later, and that there is no penalty associated with such preemption.)
- for each edge $E_j = (\tilde{N}_j, \hat{N}_j)$
  - an integer parameter B_j called the *buffer capacity* of the edge; B_j denotes the maximum number of tokens that may be stored in the buffer associated with edge E_j.
  - a *threshold amount* τ_j, denoting the minimum number of tokens that must be present in the buffer associated with edge E_j in order that node $\hat{N}_j$ may fire.
  - a *produce amount* p_j, denoting the number of tokens generated and deposited in the buffer associated with edge E_j each time node $\tilde{N}_j$ fires.
  - a *consume amount* c_j, denoting the number of tokens consumed and hence removed from the buffer associated with edge E_j each time node $\hat{N}_j$ fires.

Edge E_k is *over threshold* when the buffer contains at least τ_k tokens. A node $\hat{N}_j$ is eligible for execution when all of its input edges are over threshold. A node begins execution by reading the specified amount of data from each of its input edges. Next the processing function associated with the node is executed. After the processing function completes, the specified produce amount of tokens is appended to each output edge. Before the node terminates, but after data is produced, c_k tokens are dequeued (*consumed*) from each input queue. The production and consumption of data are both atomic operations. That is, all p_k tokens are available to the consumer at the same time; all c_k tokens are removed from an input queue in one indivisible operation.

The **PGM dataflow graph feasibility problem** is defined as follows: Given the specifications of a PGM

dataflow graph G, determine if it is possible to schedule G in such a manner that no buffer overflow ever occurs. A PGM dataflow graph for which such scheduling is possible is said to be **feasible**.

We conclude this section with two observations regarding feasible PGM dataflow graphs:

1. We can associate a *steady state execution rate* r_i with each node N_i of a PGM dataflow graph G, as follows:

$$r_i = \begin{cases} \frac{1}{y} & \text{if } N_i \text{ is a source node and } y \text{ its associated period} \\ r_k \cdot \frac{p_j}{c_j} & \text{if there is an edge } E_j = (N_k, N_i) \text{ in graph } G \end{cases}$$

   Since there may be several edges leading into a node, and each incoming edge may, in general, define a different execution rate for the node, we say that a PGM dataflow graph is *consistent* if and only if the rates defined on each node by all its incoming edges are the same. We observe that, *for a PGM dataflow graph to be feasible on a single processor, it is necessary that it be consistent*, since there would otherwise be unbounded buildup of tokens on some edge.

2. We define the *utilization* of a consistent PGM dataflow graph to be the quantity $\sum_{\text{all nodes } N_i}(r_i \cdot e_i)$. *For a consistent PGM dataflow graph to be feasible, it is necessary that its utilization not exceed unity.*

3. Intractability

In Section 2 we saw that, for a PGM dataflow graph to be feasible, it is necessary that it be consistent, and that its utilization not exceed one. We now explore the issue of determining necessary and sufficient conditions for a a PGM dataflow graph to be feasible. Unfortunately, we are unlikely to be able to determine any such conditions that can be efficiently checked. More specifically, we prove below that the PGM dataflow graph feasibility problem is intractable:

Theorem 1 *It is co-NP-hard in the strong sense to determine whether a given PGM dataflow graph is feasible.*

Proof: The proof is by reduction from the *Simultaneous Congruences Problem* (SCP). This problem was introduced by Leung and Merrill [5], and shown to be NP-hard. Subsequently, it was proven [2] that the problem is NP-hard in the strong sense.

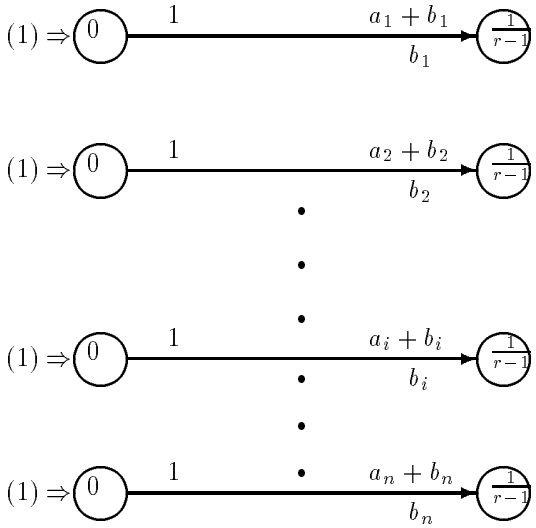

Figure 1. PGF dataflow graph constructed during the proof of Theorem 1. (Node execution requirements are written within the nodes, and the produce amount, threshold, and consume amount of an edge are written above the edge at its beginning, and above and below the edge at its end, respectively. Each source node is indicated by a double incoming arrow which is labelled by the source node's period, in parentheses.)

The Simultaneous Congruences Problem is defined as follows:

Given n ordered pairs of positive integers (a_1, b_1), (a_2, b_2), ..., (a_n, b_n). A positive integer r, $2 \leq r \leq n$.

Determine whether there exists a positive integer x, and r ordered pairs (a_{i1}, b_{i1}), (a_{i2}, b_{i2}), ..., (a_{ir}, b_{ir}) from among the given ordered pairs, such that $x \equiv a_{ij} \bmod b_{ij}$ for each j, $1 \leq j \leq r$.

Given an instance of the SCP, we construct an instance of the PGM dataflow graph feasibility problem as shown in Figure 1. This PGM dataflow graph consists of n disjoint chains, each of length 2. Each of the n source nodes (the leftmost nodes in the figure) has an execution requirement of zero, fires once every time unit, and generates one token onto the queue corresponding to its outgoing edge per firing. Each sink node has an execution requirement of $\frac{1}{r-1}$ (where r is as given in the SCP instance). The queue corresponding to the i'th edge from the top of the figure has a threshold of $a_i + b_i$, a buffer bound of the same size, and consumes b_i tokens per firing.

It is straightforward to observe that the i'th sink node is enabled for the first time at time $a_i + b_i$, and, if there is never to be any buffer overflow, is henceforth periodically enabled every b_i time units. Furthermore, if there is to be no buffer overflow, each node must fire within one time unit of being enabled, since tokens arrive along its incoming edge every time unit, and its buffer is at capacity when it is enabled. Since each sink node has an execution requirement of $\frac{1}{r-1}$, at most $r-1$ of them can execute during any time unit. Hence, there *is* buffer overflow if and only if at least r nodes are enabled simultaneously at some time t; i.e., if and only if r ordered pairs (a_{i1}, b_{i1}), (a_{i2}, b_{i2}), ..., (a_{ir}, b_{ir}) from among the ordered pairs (a_1, b_1), (a_2, b_2), ..., (a_n, b_n) satisfy $t \equiv a_{ij} \bmod b_{ij}$ for each j, $1 \leq j \leq r$. ■

As a consequence of Theorem 1, we are unlikely to be able to determine in polynomial time whether a given PGM dataflow graph is feasible or not. As a consequence of the *co*-NP hardness of the problem, it is also less likely that efficient heuristics will be found. Our approach is therefore to focus on a special class of PGM dataflow graphs, for which the feasibility problem is not necessarily as hard. This class — PGM dataflow chains — turns out to be a practically significant one, in that a large number of signal-processing applications can be modelled as PGM dataflow chains. We study these systems in greater detail in the rest of this paper.

4. PGM dataflow chains

A **PGM dataflow chain** G is a PGM dataflow graph with the additional properties that (i) there is exactly one source node, (ii) there is exactly one *sink* node, where a sink node is defined to be one which has outdegree zero (i.e., it has no edges leaving it), and (iii) Every node other than the sink has outdegree one. An example PGM dataflow chain is shown in Figure 2. As in Figure 1, the (unique) source node is marked by a double incoming arrow which is labelled by its period, in parentheses. We let N_i denote the node at distance i from the source node (which is therefore denoted as N_o); e_i the execution requirement of node N_i; p_i, τ_i, and c_i denote the produce amount, threshold, and consume amount respectively of the edge (N_i, N_{i+1}); and B_i the buffer associated with edge (N_{i-1}, N_i). (For notational convenience, we assume that $e_o = 0$; i.e., the source node has no execution requirement and hence completes firing at exactly $k \cdot y$ for all $y \in \mathbf{N}$.)

PGM chains can represent non-trivial signal processing applications. For example, the graph of Figure 3 represents a real-time image processing application from ARPA's Rapid Prototyping of Application Specific Signal Processors (RASSP) project.[1] This graph processes data collected from a Ka-band synthetic aperture radar (SAR) sensor in real-time. The SAR application is used to identify man-made objects on the ground or in the air by producing high-resolution, all-weather images.

The rest of this section provides a brief description of the processing performed by each node in the graph. This information is provided for concreteness for the reader with a signal processing background. The actual logical operation of the SAR graph is immaterial to the results we derive and the analyses we perform. The only essential properties of the SAR graph are those that influence node execution (i.e., the produce, consume, and threshold values for each node). For a more detailed description of the processing performed by the SAR benchmark, see [6].

The top row of nodes in the SAR graph each operate on one pulse of data at a time. The pulse delivered by the external source, labeled *YRange*, has already been converted to complex-valued data and consists of 118 range gate samples. The *Zero Fill* node, however, pads the pulse with zeroes to create a pulse length of 256 in preparation for the *FFT* node. Before performing the FFT, the data is passed through a Kaiser window function, represented by the node *Window Data*, to reduces sidelobe levels and perform bandpass filtering. After being compressed in the range dimension by the *Range FFT* node, the pulse is passed through the radar cross section calibration filter performed by the *RCS Mult* node.

Unlike the previous nodes in the SAR graph, which require only one pulse of data before being eligible for execution, the *Corner Turn* node requires 128 pulses of data. The *corner turn* node forms a 2-D processing array where each row of the array contains one sample from 128 different pulses and each column contains the 256 range gates that form a pulse. The processing array consists of two 64×256 frames (or sequences of pulses). As a new frame is loaded in, the previous two frames are "released" with the oldest frame being shifted out. Hence, the $256 \cdot 128$ threshold and the $256 \cdot 64$ consume values on the RCS input queue to the *Corner Turn* node.

Convolution processing is performed on each row of the 2-D matrix by the *Azimuth FFT*, *Kernel Mult*, and *Azimuth IFFT* nodes. The *Azimuth FFT* node performs a FFT on the signal, which has been aligned in

[1] The graph shown in Figure 3 is actually the mini-SAR graph, which was created to test tools developed for the RASSP project. It performs the range and azimuth compression processing in forming an image that is one eighth the size of that formed by the full SAR benchmark.

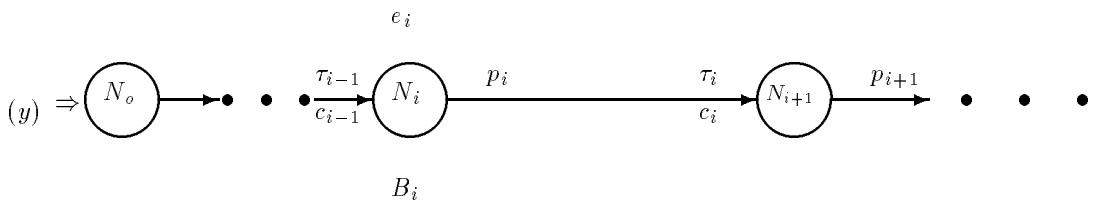

Figure 2. A PGM dataflow chain

the azimuth dimension. Next the *Kernel Mult* node multiplies each row of the matrix by a convolution kernel. Before the SAR image is output to the *Sink* node, an inverse FFT is performed by the *Azimuth IFFT* node.

5. Analyzing PGM dataflow chains

We now address the issue of determining whether a given PGM dataflow chain is feasible and, if so, the issue of actually constructing a schedule for it. This section is organized as follows. In Section 5.1, we determine lower bounds on the size of the buffer associated with each edge (Lemma 1), and reiterate the utilization bound of Section 2 (Lemma 2). In Section 5.2, we introduce the concept of *deadlines* for particular firings of nodes, and present a scheduling algorithm that is *optimal* in the sense that it will successfully schedule any feasible PGM dataflow chain with no buffer overflow. In Section 5.3, we design a reasonably efficient feasibility test which, while not necessary, is certainly sufficient in that any PGM dataflow chain that passes this test is guaranteed to be feasible. Hence, given a PGM dataflow chain

1. We can check and see whether it possesses the necessary properties (Lemma 1 and Lemma 2); if not, the chain is infeasible;
2. If it does possess these necessary properties, we can check to determine whether it passes the sufficient feasibility test (Section 5.3);
3. If so, the system can be scheduled to never suffer buffer overflow, by simply using the optimal scheduling algorithm of Section 5.2. On the other hand, if the system does not pass the feasibility test, we do not know whether it is feasible (although we are guaranteed that, if it is feasible, then the optimal scheduling algorithm of Section 5.2 can schedule it with no buffer overflow).

In Section 5.4, we put all these pieces together, and describe a procedure for implementing applications that are modelled as PGM dataflow chains. While our implementations are not optimal in terms of computing resources (CPU or memory), it is possible to make a CPU/ design-time tradeoff: by expending more time in the design process, higher CPU utilization can be obtained (equivalently, a less powerful CPU can be used). And, our approach can be formally proved to require not much more than twice the minimum amount of memory needed in an optimal solution.

5.1. Necessary conditions for feasibility

Lemma 1 *For a given PGM dataflow chain to be feasible, it is necessary that $B_i \geq b_i$ for all nodes N_i, where b_i is defined as follows:*

$$b_i \stackrel{\text{def}}{=} \left(\left\lceil \frac{\tau_{i-1}}{\gcd(p_{i-1}, c_{i-1})} \right\rceil - 1 \right) \cdot \gcd(p_{i-1}, c_{i-1}) + p_{i-1} \ .$$

Example 1 Before presenting the proof sketch for Lemma 1, we derive b_i for the *Corner Turn* and *Azimuth FFT* nodes from the SAR graph of Figure 3.

$$\begin{aligned} b_{Corner\ Turn} &= \left(\left\lceil \frac{256 \cdot 128}{\gcd(256, 256 \cdot 64)} \right\rceil - 1 \right) \cdot \gcd(256, 256 \cdot 64) \\ &\quad +256 \\ &= \left(\left\lceil \frac{256 \cdot 128}{256} \right\rceil - 1 \right) \cdot 256 + 256 = 128 \cdot 256 \end{aligned}$$

$$\begin{aligned} b_{Azimuth\ FFT} &= \left(\left\lceil \frac{128}{\gcd(256 \cdot 128, 128)} \right\rceil - 1 \right) \cdot \gcd(256 \cdot 128, 128) \\ &\quad +256 \cdot 128 \\ &= \left(\left\lceil \frac{128}{128} \right\rceil - 1 \right) \cdot 128 + 256 \cdot 128 = 256 \cdot 128 \end{aligned}$$

■

Proof Sketch: Since each firing of node N_{i-1} deposits p_{i-1} tokens into the i'th buffer and each firing of N_i consumes c_{i-1} tokens, it follows that

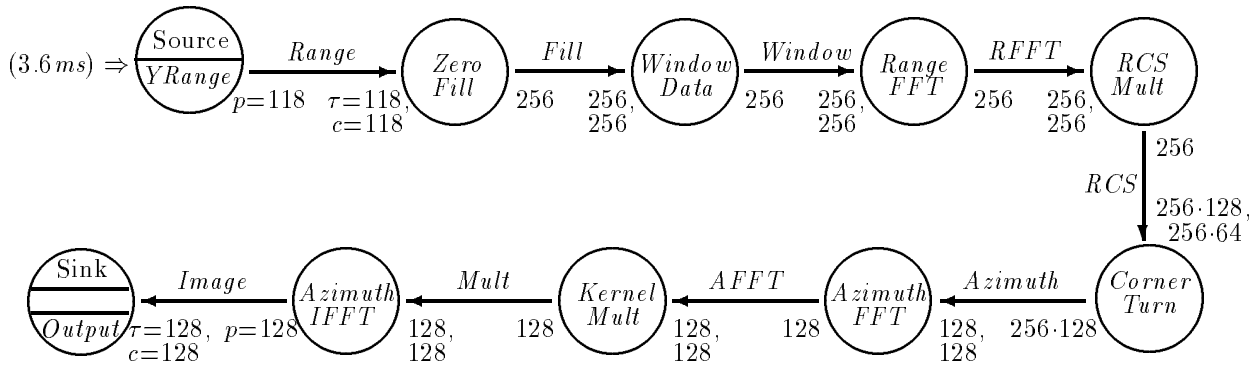

Figure 3. SAR Graph

(i) the number of tokens in the i'th buffer at any instant in time is a multiple of $\gcd(p_{i-1}, c_{i-1})$, and (ii) at some instant during a valid execution, there will be exactly $k \cdot \gcd(p_{i-1}, c_{i-1})$ tokens in the buffer, for *each* multiple of $\gcd(p_{i-1}, c_{i-1})$ less than τ_{i-1}. By algebraic manipulation, it can be seen that the largest multiple of $\gcd(p_{i-1}, c_{i-1})$ less than τ is $(\lceil \tau_{i-1}/\gcd(p_{i-1}, c_{i-1})\rceil - 1) \cdot \gcd(p_{i-1}, c_{i-1})$; when the buffer contains this many tokens, node N_i is not eligible for execution. Hence, when N_{i-1} fires once again, the buffer will contain exactly $(\lceil \tau_{i-1}/\gcd(p_{i-1}, c_{i-1})\rceil - 1) \cdot \gcd(p_{i-1}, c_{i-1}) + p_{i-1}$ tokens. ■

For each node N_i, we define a *steady state execution rate* r_i as follows:[2]

$$r_i = \begin{cases} r_{i-1} \cdot \frac{p_{i-1}}{c_{i-1}} & \text{if } i > 0 \\ \\ \frac{1}{y} & \text{if } i = 0 \end{cases} \qquad (1)$$

Lemma 2 *For a given PGM dataflow chain to be feasible on a single processor, it is necessary that*

$$\sum_{\text{all nodes } N_i} (r_i \cdot e_i) \quad \leq \quad 1$$

Proof Sketch: Observe that, for large t each node N_i executes approximately $t \cdot r_i$ times over the interval of time $[0, t]$. Hence, $r_i \cdot e_i \cdot t$ denotes the amount of time for which node N_i must be executed over interval $[0, t]$, as $t \to \infty$. Summing over all nodes and observing that the total amount of execution time available over $[0, t]$ is t units, we obtain the desired condition. ■

5.2. An optimal scheduling algorithm

Let β_i denote the number of tokens initially in the buffer at node N_i, at time $t = 0$.

[2]For a more precise treatment of execution rates, see [3].

Definition 1 (Deadlines) *For each $k \in \mathbf{N}$ and for each node N_i, we define a $\mathsf{dline}(N_i, k)$ that denotes the latest time at which node N_i* must *complete execution for the k'th time, if buffer overflow is not to occur. This is defined as follows:*

$$\mathsf{dline}(N_i, k) = \begin{cases} \mathsf{dline}(N_{i-1}, \left\lfloor \frac{(k-1)c_{i-1} + B_i - \beta_i}{p_{i-1}} \right\rfloor + 1) \\ \qquad \text{if } i > 0 \\ \\ (k-1) \cdot y \qquad \text{if } i = 0 \end{cases} \qquad (2)$$

Given the specification for a PGM dataflow chain, $\mathsf{dline}(N_i, k)$ can be determined in time $\Theta(i)$, by making a recursive call for computing some deadline for node N_{i-1}. In Figure 4, we illustrate the process, by deriving deadlines for the first 2 executions of the *Zero Fill* node, *Corner Turn*, and *Azimuth FFT* nodes.

The definition of $\mathsf{dline}(N_i, k)$ is motivated by the following considerations: With respect to a particular execution of the PGM dataflow chain, let $\mathsf{finish}(N_i, k)$ denote the time at which node N_i completes execution for the k'th time. Suppose that node N_{i-1} has completed ℓ firings by then. For buffer overflow to not have occurred at node N_i, it is necessary that

$$\beta_i + \ell p_{i-1} - (k-1)c_{i-1} \leq B_i \ .$$

By algebraic manipulation, we obtain

$$\begin{aligned} & \beta_i + \ell p_{i-1} - (k-1)c_{i-1} \leq B_i \\ \equiv \quad & \ell \leq \frac{(k-1)c_{i-1} + B_i - \beta_i}{p_{i-1}} \\ \equiv \quad & \ell \leq \left\lfloor \frac{(k-1)c_{i-1} + B_i - \beta_i}{p_{i-1}} \right\rfloor \end{aligned}$$

Let, $\forall i$, $B_i = b_i$ and $\beta_i = 0$.

$$\begin{aligned}
\mathsf{dline}(Zero\ Fill, 1) &= \mathsf{dline}(Source, \left\lfloor \frac{(1-1)118 + 118 - 0}{118} \right\rfloor + 1) \\
&= \mathsf{dline}(Source, 2) = y \\
\mathsf{dline}(Zero\ Fill, 2) &= \mathsf{dline}(Source, \left\lfloor \frac{(2-1)118 + 118 - 0}{118} \right\rfloor + 1) \\
&= \mathsf{dline}(Source, 3) = 2y \\
\mathsf{dline}(Corner\ Turn, 1) &= \mathsf{dline}(RCS\ Mult, \left\lfloor \frac{(1-1)(256 \cdot 64) + 128 \cdot 256 - 0}{256} \right\rfloor + 1) \\
&= \mathsf{dline}(RCS\ Mult, 129) = 132y \\
\mathsf{dline}(Corner\ Turn, 2) &= \mathsf{dline}(RCS\ Mult, \left\lfloor \frac{(2-1)(256 \cdot 64) + 128 \cdot 256 - 0}{256} \right\rfloor + 1) \\
&= \mathsf{dline}(RCS\ Mult, 193) = 196y \\
\mathsf{dline}(Azimuth\ FFT, 1) &= \mathsf{dline}(Corner\ Turn, \left\lfloor \frac{(1-1)128 + 256 \cdot 128 - 0}{256 \cdot 128} \right\rfloor + 1) \\
&= \mathsf{dline}(Corner\ Turn, 2) = 196y \\
\mathsf{dline}(Azimuth\ FFT, 2) &= \mathsf{dline}(Corner\ Turn, \left\lfloor \frac{(2-1)128 + 256 \cdot 128 - 0}{256 \cdot 128} \right\rfloor + 1) \\
&= \mathsf{dline}(Corner\ Turn, 2) = 196y
\end{aligned}$$

Figure 4. An example of determining deadlines

Thus, for overflow to not occur, it is necessary that

$$\mathsf{finish}(N_i, k) < \mathsf{finish}(N_{i-1}, \left\lfloor \frac{(k-1)c_{i-1} + B_i - \beta_i}{p_{i-1}} \right\rfloor + 1) \quad (3)$$

be satisfied. Thus $\mathsf{dline}(N_i, k)$ — the latest time at which node N_i may complete execution for the k'th time — is related to the latest times at which executions of node N_{i-1} may complete.

Algorithm edf. We propose the following algorithm for scheduling PGM dataflow chains. Recall that a node N_i is defined to be *eligible* at a particular time instant if there are at least τ_{i-1} tokens buffered at its incoming edge at that time instant. At each time instant, Algorithm edf executes the currently eligible node with the highest priority according to the following priority rule: Eligible node N_i, which has fired $(k_i - 1)$ times thus far, has priority over eligible node N_j, which has fired $(k_j - 1)$ times thus far, iff

- $\mathsf{dline}(N_i, k_i) < \mathsf{dline}(N_j, k_j)$, or
- $\mathsf{dline}(N_i, k_i) = \mathsf{dline}(N_j, k_j)$, and $i > j$.

That is, nodes are scheduled for execution according to deadlines, with ties broken in favor of the node further down the chain.

A scheduling algorithm is defined to be *optimal* for PGM dataflow graphs on uniprocessors if and only if it can schedule every feasible PGM dataflow chain with no buffer overflow on a single processor.

Theorem 2 *Algorithm* edf *is optimal for PGM dataflow chains on uniprocessors.*

Proof Sketch: In the appendix. ■

5.3. A sufficient condition for feasibility

As a consequence of Lemma 2, it is necessary that the *utilization* ρ of a PGM dataflow chain be at most one, where ρ is defined as follows:

$$\rho = \sum_{\text{all nodes } N_i} (r_i \cdot e_i) \ .$$

When ρ is strictly less than one, it is possible to design an efficient sufficient (albeit not necessary) feasibility test for PGM dataflow chains. As we shall soon see, when $\rho \rightarrow 1$ the complexity of the feasibility test $\rightarrow \infty$. This feasibility test is based upon identifying a "worst-case" scenario for the PGM dataflow chain, and

then simulating the behavior of the optimal scheduling algorithm – Algorithm edf — on the chain in this worst-case scenario.

Definition 2 (Saturation) *A PGM dataflow chain is said to be* saturated *if, $\forall i$, β_i — the number of tokens in the buffer at node N_i at time $t = 0$ — is greater than or equal to b_i, where b_i is as defined in Lemma 1.*

Recall that by Lemma 1, it is necessary that the buffer at node N_i be of size at least b_i, for all i. Hence, each potentially feasible PGM dataflow chain can saturated.

The following lemma asserts that the saturated state represents a "worst-case" scenario for a PGM dataflow chain:

Lemma 3 *If a saturated PGM dataflow chain can be scheduled with no buffer overflow by Algorithm* edf*, then the chain is feasible when $\beta_i = 0$ for all nodes N_i.*

(Notice that it is not claimed that this saturated chain is actually ever achieved during execution — the lemma merely claims that no realized state is "worse" than a saturated state.)

Lemma 4 characterizes the behavior of feasible PGM dataflow chains when executed starting from their saturated state:

Lemma 4 *If Algorithm* edf *idles the processor while scheduling a saturated PGM dataflow chain, then Algorithm* edf *will successfully schedule the saturated chain for all time, with no buffer overflow.*

From Lemmas 3 and 4, we may conclude that if Algorithm edf idles the processor while scheduling a PGM dataflow chain starting from a saturated state, then the PGM dataflow chain is feasible when starting with all its buffers initially empty. A sufficient feasibility test for PGM dataflow chains now suggests itself:

Algorithm feas**:** Simulate the behavior of Algorithm edf on the saturated PGM dataflow chain. Continue the simulation until one of the two following events occur:

buffer overflow:	return "not known to be feasible."
processor idles:	return "guaranteed feasible."

Run-time analysis: For chains that have $\rho < 1$, the simulation in Algorithm feas is guaranteed to terminate. A loose upper bound on the time interval for which the simulation must be carried out is obtained as follows: The b_i tokens at node N_i at time 0 may force it to fire no more than (b_i/c_{i-1}) times, for a total execution requirement (over all nodes) of $\sum_{\text{all nodes } N_i}((b_i/c_{i-1}) \cdot e_i)$. The externally triggered firing of the source node causes an additional execution requirement of no more than $\rho \cdot t$ over any interval of size t; therefore, the processor will idle at or before the smallest t satisfying

$$\sum_{\text{all nodes } N_i} \frac{b_i e_i}{c_{i-1}} + \rho \cdot t \quad < \quad t \ ,$$

which is satisfied by

$$t > \frac{1}{1-\rho} \left(\sum_{\text{all nodes } N_i} \frac{b_i e_i}{c_{i-1}} \right)$$

For constant ρ, this expression is pseudopolynomial in the parameters b_i and e_i, and hence in the input instance.

5.4. Synthesizing a PGM dataflow chain

Suppose that we have a DSP application that can be represented as a PGM dataflow chain (e.g., the SAR application), and we wish to implement this chain on appropriate hardware. The values of the *execution requirements* of the nodes are influenced by the choice of CPU: in general, the faster the CPU, the smaller the e_i's. We also have some control over the *buffer capacities*: by making more memory space available, we can make the B_i's larger. The remaining parameters — the period of the source node, the p_i's, the τ_i's, and the c_i's — are determined by the application characteristics, and are not really under our control.

Given such a DSP application, we outline below how we would choose a CPU and the buffer sizes in order to be able to guarantee that our implementation is feasible.

1. Choose a fast enough CPU such that the *utilization* ρ of the chain is strictly less than one. (There is a cost-efficiency tradeoff here in that the faster the CPU — consequently, the smaller the value of ρ — the more efficient the remainder of this design procedure.)
2. Choose buffer sizes B_1, B_2, ..., B_n such that $B_i \geq b_i$ for each i, where the b_i's are as defined in Lemma 1.
3. Execute Algorithm feas on the system.
   - if Algorithm feas returns "guaranteed feasible" then we have a feasible design; return.

- if buffer overflow occurs at the i'th buffer, increase B_i by (at least) the amount of the overflow, and re-execute Step 3.

And how close to optimal is the design so obtained? It can be shown that if a PGM dataflow chain is feasible with buffer-sizes $\hat{B}_1, \hat{B}_2, \ldots, \hat{B}_n$, then Algorithm feas will return "guaranteed feasible" with the values of $B_1, B_2, \ldots, B_n$ set to $b_1+\hat{B}_1, b_2+\hat{B}_2, \ldots, b_n+\hat{B}_n$, where the b_i's are as defined in Lemma 1. Since by Lemma 1 it is necessary that $\hat{B}_i \geq b_i$ for all i, this means that the design obtained above is likely to use no more than twice the minimum amount of buffer space necessary to achieve feasibility.

6. Conclusions

The dataflow paradigm offers a convenient and powerful means to design and analyze signal processing applications. In *embedded* signal-processing applications, particularly those for safety-critical systems, hard performance guarantees must be met. We have studied the issue of determining whether a given application, represented as a dataflow graph, can be feasibly scheduled on available resources and if so, how to go about constructing the actual schedule. For general dataflow graphs, we have shown that this problem is intractable (co-NP hard in the strong sense); for the more restricted (but still immensely useful) model of dataflow chains, we have suggested a synthesis approach that permits the system architect to make efficient use of computational and memory resources. In conjunction with the results in [3], the research presented here represents a comprehensive scheduling-theoretic analysis of a popular software engineering methodology; it is hoped that this analysis will (i) be useful to the system architect by extending this methodology for use in constructing systems with hard-real-time guarantees, and (ii) serve as a concrete "proof-of-concept" example to validate our thesis that formal analysis of real-time properties and behavior has the potential to provide major benefits in the ongoing endeavor to obtain more powerful tools, techniques, and methodologies for the design and implementation of complex computer application systems.

References

[1] *PGM – Processing Graph Method Specification*, December 1987. Prepared by the Naval Research Laboratory for use by the Navy Standard Signal Processing Program Office (PMS-412). Version 1.0.

[2] S. Baruah, R. Howell, and L. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 118:3–20, 1993.

[3] S. Goddard. Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application. In *Proceedings of the Real-Time Technology and Applications Symposium*, Montreal, Canada, June 1997.

[4] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[5] J. Leung and M. Merrill. A note on the preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11:115–118, 1980.

[6] B. Zuerndorfer and G. A. Shaw. SAR processing for RASSP application. In *Proceedings of First Annual RASSP Conference*, Arlington, VA, August 1994.

Appendix

A. Proof of optimality of Algorithm edf

Lemma 5 *If* $(i > j)$ *and* $\mathsf{dline}(N_i, k_i) \leq \mathsf{dline}(N_j, k_j)$, *then* N_i *completes its* k_i*'th execution strictly* before N_j *completes its* k_j*'th execution in any valid schedule.*

Proof Sketch: By the recursive definition of deadlines, and since $i > j$, $\mathsf{dline}(N_i, k_i) = \mathsf{dline}(N_j, \hat{k})$ for some $\hat{k}$. Since $\mathsf{dline}(N_i, k_i) \leq \mathsf{dline}(N_j, k_j)$, it follows that $\hat{k} \leq k_j$. Let $\mathsf{finish}(N_i, k_i)$ denote the time at which node N_i completes execution for the k_i'th time. Suppose that node N_{i-1} has completed ℓ firings by then. For buffer overflow to not have occurred at node N_i, it is necessary that

$$\beta_i + \ell p_{i-1} - (k_i - 1)c_{i-1} \leq B_i \ .$$

By algebraic manipulation, we obtain

$$\begin{aligned} & \beta_i + \ell p_{i-1} - (k-1)c_{i-1} \leq B_i \\ \equiv \quad & \ell \leq \frac{(k-1)c_{i-1} + B_i - \beta_i}{p_{i-1}} \\ \equiv \quad & \ell \leq \left\lfloor \frac{(k-1)c_{i-1} + B_i - \beta_i}{p_{i-1}} \right\rfloor \end{aligned}$$

Thus, for overflow to not occur, it is necessary that

$$\begin{aligned} \mathsf{finish}(N_i, k) \quad < \quad & \\ \mathsf{finish}(N_{i-1}, & \left\lfloor \frac{(k-1)c_{i-1} + B_i - \beta_i}{p_{i-1}} \right\rfloor + 1) \end{aligned} \tag{4}$$

be satisfied. Observe that the second argument of the $\mathsf{finish}(,)$ on the right hand side above is identical to the second argument of the $\mathsf{dline}(,)$ on the right hand side in the recursive definition of deadlines.

By repeating the above argument for $i-1, i-2, \ldots, j+1$, we conclude that it is necessary that $\mathsf{finish}(N_i, k_i) < \mathsf{finish}(N_j, \hat{k})$; since $\hat{k} \leq k_j$, it follows that N_i must complete its k_i'th execution before N_j completes its k_j'th execution. ■

Lemma 6 *In any valid schedule, node N_i completes execution for the k_i'th time no later than time $\mathsf{dline}(N_i, k_i)$, for all $i \geq 1$.*

Proof Sketch: This is easily proved by induction on i. The base case is $i = 0$. From the definition, $\mathsf{dline}(N_o, k) = y \cdot (k-1)$; by assumption (Section 4), the source node N_o has no execution requirement, and hence completes execution the instant it is enabled, at exactly these time instants.

For the induction step, assume that $\mathsf{finish}(N_{i-1}, k') \leq \mathsf{dline}(N_{i-1}, k')$ for all k'. Suppose that node N_{i-1} has completed ℓ firings before N_i completes its k'th firing. For buffer overflow to not have occurred at node N_i, it is necessary that

$$\begin{aligned} & \beta_i + \ell p_{i-1} - (k-1)c_{i-1} \leq B_i \\ \equiv \quad & \ell \leq \frac{(k-1)c_{i-1} + B_i - \beta_i}{p_{i-1}} \\ \equiv \quad & \ell \leq \left\lfloor \frac{(k-1)c_{i-1} + B_i - \beta_i}{p_{i-1}} \right\rfloor \end{aligned}$$

It therefore follows that

$$\mathsf{finish}(N_i, k) < \mathsf{finish}(N_{i-1}, \left\lfloor \frac{(k-1)c_{i-1} + B_i - \beta_i}{p_{i-1}} \right\rfloor + 1)$$

Given the induction hypothesis, the lemma follows. ■

A schedule is defined to be *work conserving* if and only if it never idles the processor while there is some node eligible for execution.

Lemma 7 *Every feasible PGM dataflow chain can be scheduled in a work-conserving manner with no buffer overflow.*

Proof Sketch: Since the time at which a node most complete a particular firing depends upon the time at which other nodes complete firing (Equation 4), one may at first assume that there is some benefit to be obtained by adopting a non work-conserving strategy. However, observe that while being non-conserving may delay the actual time by which some task must complete, such a strategy will not cause any reduction in the amount of actual *work* that must be done prior to this completion time (indeed, by perhaps having the source node fire during the idle interval, the amount of work may actually increase). ■

We are now ready to sketch out a proof for Theorem 2.

Proof Sketch of Theorem 2: Let G be a feasible PGM dataflow chain in which all parameters are integers. Our proof of correctness proceeds by induction on time t. The induction hypothesis is that there is a valid work-conserving schedule — OPT(t) — which agrees exactly with the schedule generated by Algorithm edf over the interval $[0, t)$. (By Lemma 7, this hypothesis is true for $t = 0$, since G has a workconserving schedule in which no buffer overflow occurs.) The inductive step consists of obtaining a new valid work-conserving schedule OPT$(t+1)$ from OPT(t) such that (i) OPT$(t+1)$ and OPT(t) are identical over the interval $[0, t)$, and (ii) OPT$(t+1)$ and Algorithm edf make the same scheduling decision during $[t, t+1]$. The inductive hypothesis would then be true for $t+1$ as well, and the proof would be complete.

Suppose that Algorithm edf schedules the k_i'th firing of Node N_i during interval $[t, t+1]$, while OPT(t) scheduled the k_j'th firing of node N_j. (Observe that this implies that $\mathsf{dline}(N_i, k_i) \leq \mathsf{dline}(N_j, k_j)$.)

Case: $(i = j)$**.** The schedule OPT$(t+1)$ in this case is simply the schedule OPT(t).

Case: $(i > j)$**.** By Lemma 5, OPT(t) completes N_i's k_i'th firing strictly before N_j's k_j'th firing. Thus, we can swap the allocation to N_i made in OPT(t) with the allocation made to N_j during $[t, t+1]$, and still not cause either firing to complete after its deadline[3]. (If this completes the execution of N_i, then it is possible that N_{i+1} transits from an ineligible to an eligible state; if this firing of N_{i+1} has a deadline before $\mathsf{dline}(N_j, k_j)$, it may be necessary to further swap the allocation to N_j with the an allocation to N_{i+1}, and so on a maximum of $n-i$ times, where n is the length of the chain.) The schedule obtained after these swaps is OPT$(t+1)$.

Case: $(i < j)$**.** Once again, we can swap the allocation to N_i made in OPT(t) with the allocation made to N_j during $[t, t+1]$. However, this may cause nodes in $N_{i+1}, \ldots, N_{j-1}$ to become eligible, and these firings may have deadlines prior to N_j's. The point to note, however, is that none of these firings can have a deadline prior to N_i's k_i'th firing, since they are activated by N_i's k_i'th firing. Hence once again a series of swaps – a maximum of $j-1$ – will yield a schedule in which no buffer overflow occurs; this schedule is OPT$(t+1)$ ■

[3] Where the deadline is as implied by Lemma 6