On the Management of Latency in the Synthesis of Real-Time Signal Processing Systems from Processing Graphs

by

Steve Goddard

A dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1998

Approved by:

Kevin Jeffay, Advisor

James H. Anderson

Lars S. Nyland

Don Smith

David Stotts

Abstract

Steve Goddard: On the Management of Latency in the Synthesis of Real-Time Signal Processing Systems from Processing Graphs.
(Under the direction of Kevin Jeffay.)

Complex digital signal processing systems are commonly developed using directed graphs called *processing graphs*. 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. When sufficient data arrives, a node executes its function from start to finish without synchronization with other nodes, and appends data to the edge connecting it to a consumer node.

We combine software engineering techniques with real-time scheduling theory to solve the problem of transforming a processing graph into a predictable real-time system in which latency can be managed. For signal processing graphs, real-time execution means processing signal samples as they arrive without losing data. Latency is defined as the time between when a sample of sensor data is produced and when the graph outputs the processed signal.

We study a processing graph method, called PGM, developed by the U.S. Navy for embedded signal processing applications. We present formulae for computing node execution rates, techniques for mapping nodes to tasks in the rate-based-execution (RBE) task model, and conditions to verify the schedulability of the resulting task set under a rate-based, earliest-deadline-first scheduling algorithm. Furthermore, we prove upper and lower bounds for the total latency any sample will encounter in the system. We show that there are two sources of latency in real-time systems created from processing graphs: inherent and imposed latency. *Inherent latency* is the latency defined by the dataflow attributes and topology of the processing graph. *Imposed latency* is the latency imposed by the scheduling and execution of nodes in the graph.

We demonstrate our synthesis method and the management of latency using three applications from the literature and industry: a synthetic aperture radar application, an INMARSAT mobile satellite receiver application, and an acoustic signal processing application from the ALFS anti-submarine warfare system.

This research is the first to model the execution of processing graphs with the real-time RBE model, and appears to be the first to identify and quantify inherent latency in processing graphs.

To my wife, Anne.

Acknowledgments

I would like to take this opportunity to thank my advisor, Kevin Jeffay. His unique combination of sharp insight, sense of humor and quick wit made this research endeavor both challenging and fun. Kevin has taught me well; especially that earliest-deadline-first scheduling may not be just an algorithm, but a way of life!

My thanks also goes to Jim Anderson and Sanjoy Baruah for sharing their excitement over new discoveries. I shall miss Jim's enthusiasm that led to so many impromptu research discussions. And without Sanjoy's ongoing participation, this dissertation would not be complete. Thanks as well to the rest of my committee: Lars Nyland, Don Smith, and David Stotts.

I have grown academically and personally from interactions with many other people from the Computer Science Department, but especially Tom Hudson, Mark Livingston, Mark Moir, Manuel Oliveira Neto, Mark Parris, Srikanth Ramamurthy, Tom White, and Jason Wilson. I would also like to thank the generous alumni of the Computer Science Department for their donations to the Alumni Fellowship, which funded my last year of research.

I have had the pleasure to work with some very talented people through my consulting contracts. Without them, I would not have been able to conduct this research. To my friends and colleagues at General Dynamics, Lucent Technologies, and Bell Labs: Thanks! Personally, I would like to thank Amaury Alvarez for his curiosity and keen intellect, Joe Ciriano for befriending and guiding two wayward northerners when my wife, Anne, and I first arrived in North Carolina, and lastly, Bob Judd for his enduring support of my career in both industry and academia.

Thanks to Chris and Beth Portier for sharing their faith, family, and love with me and my family. The creativity and intensity with which they approach life has permeated both our lives and this research.

Most of all, thanks to my wife, Anne. Not only would I have been unable to finish this dissertation without her support and understanding, I would have never had the courage to start.

Contents

List of Tables

List of Figures

List of Abbreviations

ALFS	Airborne Low Frequency Sonar
ASW	anti-submarine warfare
DFS	depth-first search
DIFAR	Directed Low Frequency Analysis and Recording
EDF	earliest-deadline-first
FCFS	first-come-first-served
FFT	fast Fourier transformation
FIFO	first-in-first-out
INMARSAT	International Maritime Satellite
LASM	Logical Application Stream Model
PGM	Processing Graph Method
RASSP	Rapid Prototyping of Application Specific Signal Processors
RBE	rate-based execution
RBE-EDF	rate-based-execution earliest-deadline-first
RTP/C	Real-Time Producer/Consumer
SARTOR	Software Automation for Real-Time Operations
SAR	synthetic aperture radar
SDF	Synchronous Dataflow

List of Symbols

$b \mid a$	Integer b divides integer a.
$\gcd(\mathcal{P})$	Greatest common divisor of the elements in set $\mathcal{P}$.
$\mathrm{lcm}(\mathcal{P})$	Least common multiple of the elements in set $\mathcal{P}$.
G	A directed graph.
V	A nonempty finite set of graph vertices (nodes).
E	A finite set of graph edges (queues).
$\psi(q)$	$\psi(e) = (u, v)$ represents a queue connecting node u to node v.
u	A node in a graph.
v	A node in a graph.
w	A node in a graph.
q	A queue in a graph.
$\mathit{prd}(q)$	Produce amount for queue q.
$\mathit{thr}(q)$	Threshold amount for queue q.
$\mathit{cns}(q)$	Consume amount for queue q.
$\mathit{init}(q)$	Number of initial tokens on queue q.
$\mathit{length}(q)$	Length of queue q.
$\delta^-(v)$	Number of input edges to node v.
$\delta^+(v)$	Number of output edges from node v.
$u \rightsquigarrow v$	Path from node u to node v.
$\mathcal{I}$	Set of all graph input nodes.
$\mathcal{I}_v$	Subset of input nodes $\mathcal{I}$ from which there exists a path from $u \in \mathcal{I}$ to node v.
$\mathcal{O}$	Set of all graph output nodes.
$\mathcal{O}_v$	Subset of output nodes $\mathcal{O}$ from which there exists a path from node v to node $w \in \mathcal{O}$.

$R_u = (x_u, y_u)$	A rate specification for node u.
x_u	The rate specification parameter that denotes the number of times node u executes in y_u time units.
y_u	The rate specification parameter that denotes the interval of time in which node u will execute x_u times.
d_u	Relative deadline for node u.
e_u	Execution time for node u.
s_u	Start time of node u
t_u	Denotes the beginning of the first time interval for which the execution rate specification of node u, $R_u = (x_u, y_u)$, is valid.
$R_{w \leftarrow u}$	Denotes the rate at which node w executes if nodes u and w were a producer/consumer pair in a chain.
$\mathcal{V}$	Denotes the set of nodes for which there exists a queue q in E and a node u in V such that $\psi(q) = (u, w)$.
$F_{u \rightsquigarrow w}$	The number of executions required of node u before node w is eligible for execution where $u \rightsquigarrow w$ is a chain.
F_p	The number of executions required of the source node in path p before the sink node in path p is eligible for execution where p is a chain.
$\mathcal{F}_{j \rightsquigarrow w}$	The maximum number of times source node j must execute before all of the input queues to node w in the set of paths $\{j \rightsquigarrow w\}$ are over threshold.
$L_{u \rightsquigarrow w}(t)$	Denotes the number of time units from time t before node u produces enough samples to put the input queues to node w in the set of paths $\{j \rightsquigarrow w\}$ over threshold.
$\mathcal{L}_w(t)$	The latency a sample produced at time t encounters before node w is eligible for execution when multiple periodic source nodes have paths that lead to node w.
$\mathcal{P}$	Denotes the set of paths from the set of source nodes $\mathcal{I}$ to node w.
$\hat{\mathcal{P}}$	Denotes the set of acyclic paths from the set of source nodes $\mathcal{I}$ to node w.
$\hat{\mathcal{L}}_w(t)$	The latency a sample produced at time t encounters before node w is eligible for execution when multiple periodic source nodes have paths that lead to node w in a cyclic graph.

Chapter 1

Introduction

Radar applications use a transmitter to direct radio (electro-magnetic) waves toward an object and a receiver to record the echo — the radio waves reflected off of the object. The same process occurs in sonar applications, except acoustic (sound) waves are used rather than radio waves. The distance to an object can be derived from the time it takes the echo to reach the receiver. Today this process is performed by a *signal processing* computer system in which an electro-magnetic or sound wave is represented as a sequence of numbers called a digital signal. The (digital) signal can represent information in either a time or frequency domain. In the time domain, each number represents the amplitude of the signal at a specific point in time. A time domain signal can be transformed to a frequency domain signal by applying a transform function such as a fast Fourier transformation (FFT). By measuring the shift in the frequency (the Doppler shift) of the signal, the speed of the targeted object can be identified. By using multiple receivers, the direction, distance, and speed of an object can be identified by correlating the received signals. The signal transformations and other operations performed on the data sequence are known collectively as *signal processing*.

A radar or sonar application is a signal processing application that is often executed on a dedicated computer system, called a signal processing system, which is specifically designed to meet rigid timing requirements. Embedded signal processing systems receive a continuous signal from external sensors and are required to process the signal in real time — as the signal arrives and without losing data — and present the signal processing results to an output device (often another computer or a display) within a specified time interval. For example, an embedded signal processing system may be 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 sent to the embedded signal processing system. The application 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 to the signal processing system.

Directed graphs, called *processing graphs*, 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, often referred to as a token, that is processed by a node is a sample of the signal — one number from the discrete sequence of numbers representing the signal. Processing graphs have become a standard design aid because they provide a natural means of describing signal processing applications. Each node represents a mathematical function, such as an FFT, to be performed on streams of data that flow on the edges of the graph from source nodes (sensors) to sink nodes (output devices). The streams of input data are typically generated by sensors, which sample the environment at periodic rates and send the samples to the signal processing application via an external channel. The processing graph methodology allows one to easily understand the signal processing performed by graphically depicting the structure of the algorithm. An important advantage of the graphical representation is that portions of the signal processing application (i.e., subgraphs) can be understood in the absence of the rest of the algorithm. Moreover, the algorithm can be represented and analyzed independent of the hardware architecture hosting the application.

This dissertation addresses the primary problem in developing embedded signal processing systems with a processing graph methodology: transforming a processing graph into a predictable real-time system in which latency and memory usage can be managed — all the while ensuring no data is lost. For signal processing graphs, latency is the time between when a sensor produces a data token and when the graph outputs the processed signal. Signal processing functions, such as FFTs, often operate on multiple tokens simultaneously, and the amount of data that accumulates on a graph edge before the node executes can be quite substantial. Thus, the memory usage of a graph is primarily dependent on the amount of buffering required by the graph edges during graph execution.

Our work is based on the U.S. Navy's Processing Graph Method (PGM) [50] — a processing graph model commonly used in the development of signal processing applications. We combine software engineering techniques with real-time scheduling theory to develop a method for building deterministic uniprocessor signal processing systems from existing real-time systems technology and PGM graphs. We do not, however, generate code for the system. In the parlance of software engineering methodologies, we develop a synthesis method, and show how to manage latency in the synthesis of real-time signal processing systems from PGM graphs. To better understand the problem and the issues addressed by this dissertation, Sections 1.1, 1.2, and 1.3 provide necessary background information on processing graph models (including PGM), real-time systems, and real-time scheduling theory. Our research approach and contributions are then presented in Section 1.4 and compared with related research performed by others in Section 1.5. Section 1.6 provides an overview of this dissertation.

1.1 Processing Graph Models

Processing graphs are usually represented as directed acyclic dataflow graphs in which nodes represent processing functions and graph edges depict the flow of data from one node to the next. An edge represents a producer/consumer relationship between two nodes. When sufficient data arrives, the function associated with a node executes from start to finish in isolation (i.e., without synchronization with other nodes). Processing graph paradigms differ in features, such as the number of input edges allowed per node and the rules for consumption and production of data. However, in all cases, the flow of data through the graph determines when nodes are eligible for execution.

In this section, we present two common paradigms used in processing graph models — task graphs and task chains — followed by four processing graph models in the order of their creation. The first model presented is Karp and Miller's computation graph model [39]. Next, we present the U.S. Navy's PGM [50]. PGM is used to create embedded signal processing applications and is the model assumed in this dissertation. Although our results are based on PGM graphs, they are also applicable to Lee and Messerschmitt's Synchronous Dataflow (SDF) graphs [41] and Chatterjee and Strosnider's Logical Application Stream Model (LASM) [15, 16] — the third and fourth models described in this section.

1.1.1 Task Chains and Task Graphs

The two most common paradigms used in processing graph models are *task chains* and *task graphs*. A *task chain* is a directed, acyclic, connected graph in which all nodes, except the source and sink nodes, have exactly one input edge and one output edge. The source node has one output edge and no input edges. The sink node has one input

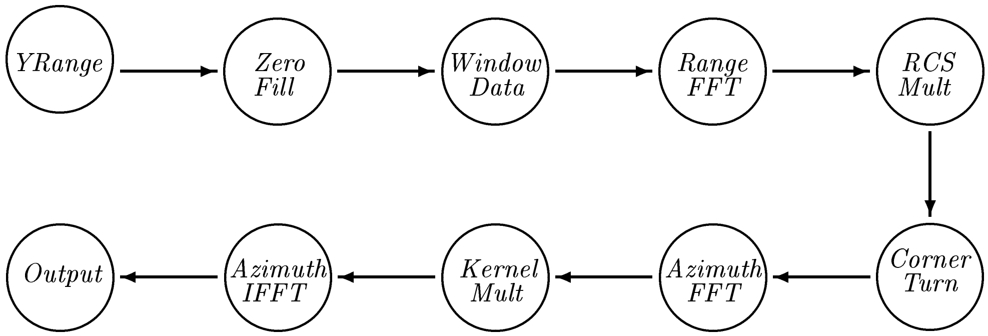


Figure 1.1: A synthetic aperture radar (SAR) task chain. Each node is represented as a circle and labeled with the type of processing performed by the node. The input device is represented as source node *YRange* and the output device is represented by the sink node *Output*.

edge and no output edges. Figure 1.1 is an example of a task chain where the nodes are represented with circles and labeled with the type of processing performed, the graph edges represent precedence constraints. This graph represents the processing performed by a synthetic aperture radar (SAR) application, which is described in Section 5.2. In most models, the source node is assumed to execute exactly once or once every p time units, and the other nodes in the graph execute after their predecessor executes. For example, in the graph of Figure 1.1, the node *Zero Fill* is eligible for execution only after the source node *YRange* executes. A *task graph* is a directed, acyclic, connected graph that can have multiple source and sink nodes. Unlike task chains, nodes in a task graph can have multiple input and output edges. For example, consider the task graph in in Figure 1.2 (adapted from [64]). This graph represents the processing performed by an International Maritime Satellite (INMARSAT) mobile satellite receiver application, which is described in Section 5.3. The graph in Figure 1.2 is a task graph rather than a task chain because several of the nodes have multiple input edges or multiple output edges. For example, the nodes labeled *Decimator* have two output edges, and the node labeled *Complex Division* has four input input and two output queues.

The effect of multiple multiple input edges on the execution of nodes depends on the execution semantics of the task graph model. In *AND models*, all predecessor nodes to a node u are required to execute before node u is eligible for execution. For example, in an AND model, the two *Demux* nodes and both *Decimator* nodes in Figure 1.2 must execute before the *Complex Division* node is able to execute. In *OR models*, a node can execute after any of its predecessor nodes execute. Thus, if the task graph of Figure 1.2

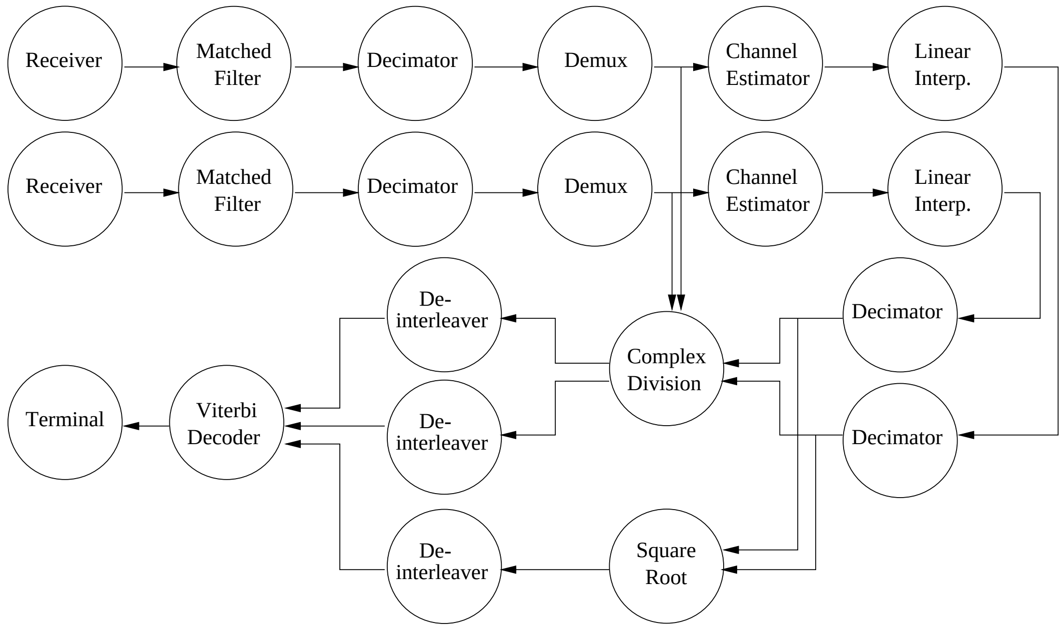


Figure 1.2: An International Maritime Satellite (INMARSAT) mobile satellite receiver task graph.

were executed according to an OR model, the *Complex Division* node could execute after either of the two *Demux* nodes or the two *Decimator* nodes execute.

1.1.2 Computation Graphs

In 1966, Karp and Miller [39] introduced an AND processing graph model called *computation graphs* that allows the specification of both precedence constraints and the amount of data required to be present on each input edge before a node is eligible for execution. The computation graph model was created to describe the parallel execution of operations in a numerical calculation. A computation graph is a directed, cyclic, connected graph that represents a sequence of parallel numerical computations. Each node in the graph corresponds to an operation in the computation, such as addition or multiplication, and each edge represents a first-in-first-out (FIFO) queue of data directed from one node to another.

It is easiest to understand the execution semantics of nodes if we assume each node in the graph executes on its own processor. Each queue has a producer and a consumer of data. The queue is an output queue for the results of the producer's computations and an input queue of data for the consumer. For example, let queue q_i be a directed edge from node n_i to node n_j. Queue q_i is an output queue to node n_i and one of possibly many input queues to node n_j. Results from the computations performed by node n_i are transported to node n_j in a FIFO order. The length of an input queue q_i (i.e., the amount of data on the queue) determines the eligibility of consumer node n_j for execution. When sufficient data has accumulated on all of the input queues to node n_j, node n_j commences execution by reading data from its input queue(s) and then deleting from the input queue(s) some, but not necessarily all, of the data read. When the computation completes, node n_j writes the results of its computation on its output queues. Since each node executes on its own processor, multiple nodes may be executing at the same time. However, no two executions of the same node are allowed to overlap.

More formally, the execution of nodes is controlled by a set of parameters that is associated with each queue. Let queue q_i be directed from node n_i to node n_j, as in Figure 1.3. The amount of data in queue q_i is quantitized into discrete units called tokens, and queues can be initialized with a number of tokens denoted by the attribute $\mathit{init}(q_i)$. When node n_i finishes its computation, it produces $\mathit{prd}(q_i)$ tokens and appends them to the tail of queue q_i. The number of tokens read by node n_j is not specified, but the number of tokens consumed (deleted) from queue q_i by node n_j is denoted $\mathit{cns}(q_i)$. The amount consumed may be less than the amount read. Queue q_i has a threshold denoted $\mathit{thr}(q_i)$. The threshold represents the minimum number of tokens that must be on queue q_i before node n_j executes. A queue is *over threshold* if the number of enqueued tokens meets or exceeds the threshold amount $\mathit{thr}(q_i)$. Since the computation graph model is an AND model, all of the input queues to a node must be over threshold before the node may execute. Thus, the four-tuple

$$(\mathit{init}(q), \mathit{prd}(q), \mathit{cns}(q), \mathit{thr}(q))$$

associated with each edge uniquely specifies the condition under which each node will execute.

To see how the the four-tuple affects node execution, consider the two nodes shown in Figure 1.3. Queue q_i (connecting nodes n_i and n_j) is associated with the four-tuple $(0, 4, 1, 2)$. This tuple specifies that queue q_i is not initialized with data, node n_i will produce 4 tokens every time it executes, node n_j will consume 1 token from queue q_i every time node n_j executes, and node n_j will not execute until queue q_i contains at least 2 tokens. Table 1.1 shows how the queue length changes as node n_i produces tokens and node n_j consumes tokens. Let $\mathit{length}(q_i)$ denote the number of tokens in queue q_i at a

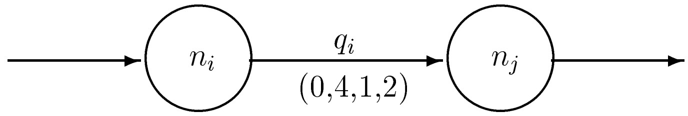


Figure 1.3: Two nodes in a computation graph. The edge connecting nodes n_i and n_j is labeled with the four-tuple (0,4,1,2), which represents the dataflow attributes $\mathit{init}(q) = 0$, $\mathit{prd}(q) = 4$, $\mathit{cns}(q) = 1$, and $\mathit{thr}(q) = 2$.

Time	**number of tokens produced by node n_i**	**number of tokens consumed by node n_j**	$\boldsymbol{length}(q_i)$
0	–	–	0
1	4	–	4
2	–	1	3
3	–	1	2
4	–	1	1

Table 1.1: A sequence of snapshots showing the length of queue q_i in Figure 1.3 at time t after node n_i produces or node n_j consumes data.

particular time. At time 0, the queue is empty since it was not initialized with data and $\mathit{length}(q_i) = 0$. Ignoring how node n_i becomes eligible for execution, assume it executes at time 1 and appends 4 tokens to queue q_i. Thus, $\mathit{length}(q_i) = 4$ at time 1. Since the threshold of queue q_i is 2 but node n_j only consumes 1 data word each time it executes, node n_j is able to execute 3 times without any more executions of node n_i. At time 2, node n_j executes and consumes 1 data word from queue q_i. It removes another data word at times 3 and 4. Thus, after the execution of node n_j at time 4, $\mathit{length}(q_i) = 1$ and node n_j is unable to execute again until node n_i produces more data.

A more complicated computation graph is shown in Figure 1.4. The computation represented in this graph is an abstraction of an approximation formula for an elliptic partial differential equation for the numerical solution to the Dirichlet problem [39]. For our purposes, the actual processing performed does not matter. There are, however, a few interesting features of the dataflow model used by this graph. Most of the queues are initially empty. The exception to this are the queues that form cycles in the graph. For example, the queue that creates a self-loop (a loop containing a single node) with node n_1 has the attributes $(n, 0, 1, 1)$ (i.e., $\mathit{init}(q) = n$, $\mathit{prd}(q) = 0$, $\mathit{cns}(q) = 1$, and $\mathit{thr}(q) = 1$). Each time node n_1 executes, it consumes one token from the queue that creates a self

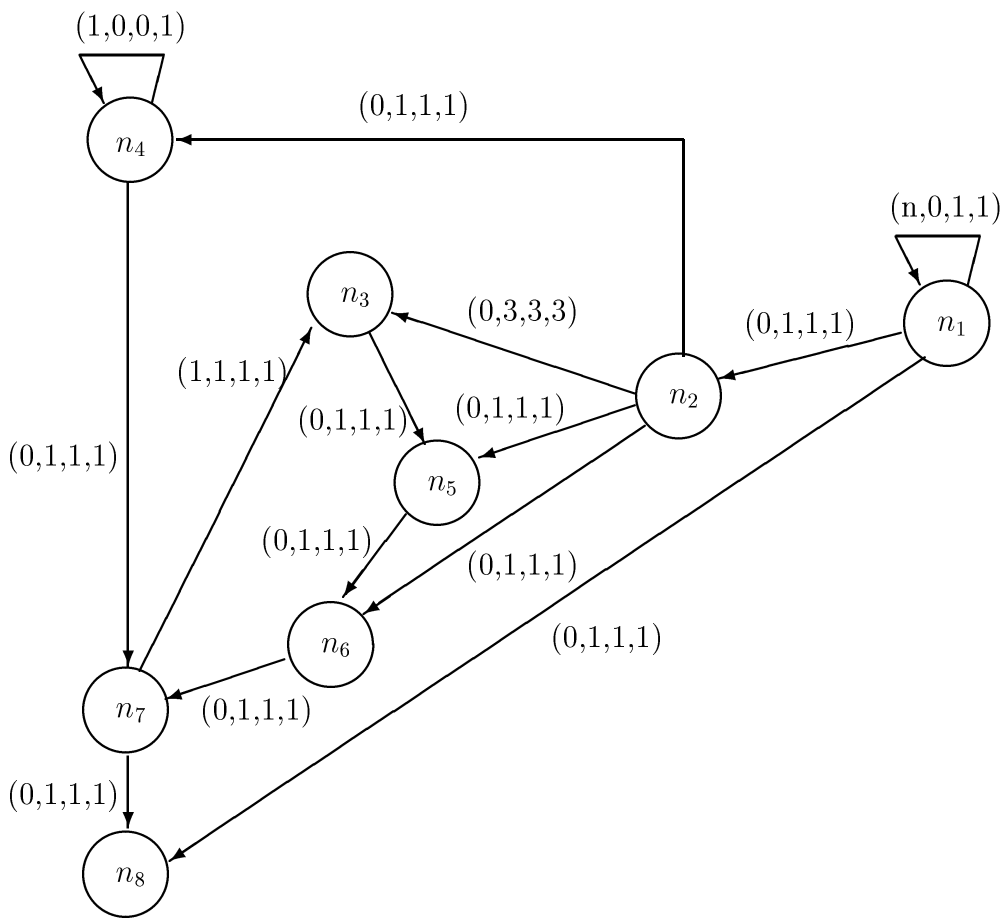


Figure 1.4: A computation graph representing an abstraction of an approximation formula for an elliptic partial differential equation. Each graph edge is labeled with a four-tuple representing the dataflow attributes associated with the queue: $(init(q), prd(q), cns(q), thr(q))$.

loop and produces one token on each of its output edges except the self loop queue. Thus, node n_1 executes exactly n times before all of the initial tokens are consumed from the input queue, and then it stops executing. Since the other nodes in the graph can only execute when node n_1 produces tokens on its output queues, node n_1 controls the number of times the rest of the nodes in the graph execute. Node n_1 is equivalent to a source node in a task graph, and an initialized self-loop is the only way to control the number of times the source node executes. Self-loops can also be used to store constants that are used in an operation, such as the self-loop at node n_4. The queue that creates a self-loop with node n_4 has the dataflow attributes $(1, 0, 0, 1)$ (i.e., $init(q) = 1$, $prd(q) = 0$, $cns(q) = 0$, and $thr(q) = 1$). Thus, n_4 never consumes the initial data word and never produces new data words on this queue; it uses the same value from this queue every time it executes.

The features of the computation graph model highlighted in this section are used extensively in signal processing graph models, including the processing graph model used in this dissertation.

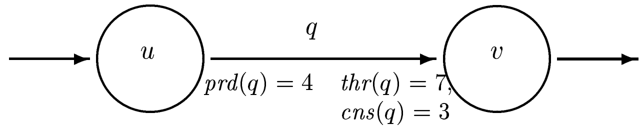
1.1.3 Processing Graph Method

The processing graph model used in this dissertation is the U.S. Navy's PGM [50,]. PGM is an extension of the computation graph model and is used to develop real-time, embedded, anti-submarine warfare (ASW) signal processing applications. The primary differences between PGM graphs and computation graphs are that *(i)* PGM source nodes do not need self-loops to control the number of times they execute, and *(ii)* the produce and consume dataflow attributes can be variable. Since PGM is used to create embedded signal processing applications, the source node usually represents an external device that executes periodically — once every p time units — and the other nodes represent signal processing functions such as an FFT. Changes in the produce and consume values of a queue are usually associated with mode changes in the application. A mode describes the type of processing an application performs such as full-band or half-band frequency analysis. Full-band frequency analysis is much more computationally intensive than half-band. The operator may select a half-band processing mode of operation until a possible submarine is located, and then change to full-band processing for verification and identification of the submarine. The change from half-band to full-band requires changes in produce and consume amounts while the graph topology stays the same.

The execution semantics of a node in PGM are slightly different than the execution semantics of a node in the computation graph model: the consumer node deletes data from its input queues *after* it completes execution, but before it produces data on its output queues. As with computation graphs, it is easiest to understand the execution semantics of nodes if we assume each node in the graph executes on its own processor. When sufficient data has accumulated on all of the input queues to a node, it commences execution by reading data from its input queues. The read is non-destructive in the sense that the data is not deleted. When the signal processing function completes, the node deletes some, but necessarily all, of the data read. The node then writes its results on its output queues. Since we assume each node executes on its own processor, multiple nodes may be executing at the same time, however, as before, no two executions of the same node are allowed to overlap. Thus, the execution of a node is *valid* if and only if *(i)* the node executes only when it is eligible for execution, *(ii)* no two executions of the same node overlap, and *(iii)* each input queue has its data atomically consumed after each output queue has its data atomically produced. A graph execution consists of a (possibly infinite) sequence of node executions. A graph execution is *valid* if and only if all of the node executions in the sequence are valid and no data loss occurs.

There are four attributes associated with each queue in a PGM graph: 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 v. The produce amount $prd(q)$ specifies the number of tokens 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 v is eligible for execution. A queue is *over threshold* if the number of enqueued tokens meets or exceeds the threshold amount $thr(q)$. Since PGM is an AND dataflow model, all of the input queues to a node must be over threshold before the node may execute. The number of tokens read by node v is not specified, but the number of tokens consumed (deleted) from queue q by node v is denoted $cns(q)$. The number of tokens consumed may be less than the number of tokens read. The number of initial data tokens on the queue is denoted $init(q)$.

Like the computation graph model, PGM allows non-unity produce, threshold, and consume amounts. However, PGM also allows variable produce and consume amounts as long as the dataflow attributes remain non-negative, and the consume amount is less than or equal to the threshold. This last requirement is important so that the node does not try to read more tokens than there are on the input queue. Because changes in produce and consume values are usually associated with application mode changes, we consider the variable dataflow attributes to be fixed values that only change when the



Time	number of tokens produced by node u	number of tokens consumed by node v	***length***(q)
0	–	–	0
1	4	–	4
3	4	–	8
4	–	3	5
5	4	–	9
6	–	3	6
7	4	–	10
8	–	3	7
9	4	3	8
10	–	3	5

Figure 1.5: A two-node PGM chain and a snapshot sequence. The snapshot sequence shows the length of queue q during the first 5 execution events of nodes u and v where $prd(q) = 4$, $thr(q) = 7$, and $cns(q) = 3$. For simplicity, nodes are assumed to execute in unit time.

application changes modes.

In signal processing applications, the concept of a "sliding window" is implemented by setting a queue's threshold to be larger than its consume amount. The last portion of data used in one execution of the consumer node is used as the first portion in the next execution. Imagine laying a window over an array of data so that only 1024 data points are visible, performing a calculation with these 1024 points, and then moving the window 768 positions to the right so that 256 old values and 768 new values are visible for the next calculation. This effect is achieved in PGM graphs by setting the threshold on a queue to 1024 and the consume amount to 768. When the consumer node executes, it reads all 1024 tokens (data points), but it only consumes 768 of the 1024 tokens, leaving 256 already read tokens on the queue. It practice, is common to initialize such queues with $(thr(q) - cns(q))$ tokens so that the amount of initialized data is equal to the amount of data read but not consumed by the node.

The impact of non-unity dataflow attributes on latency and memory usage can be demonstrated with snapshots of the execution of the two PGM nodes shown in Figure 1.5. The snapshot sequence summarizes the first 5 executions of nodes u and v (each on their own processor) and shows the length of queue q, denoted $length(q)$, after node u produces tokens and node v consumes tokens at each time step. In this example, we ignore how node u becomes eligible for execution and, instead, focus on the impact of the non-unity dataflow attributes of queue q on the execution of node v. We also ignore the time required to execute a node, and assume that nodes execute in unit time. The dataflow attributes of queue q are $prd(q) = 4$, $thr(q) = 7$, and $cns(q) = 3$. Initially, the queue has no tokens ($init(q) = 0$). Node u executes once every 2 time units, starting at time 1. It executes twice, at times 1 and 3, before node v is first eligible for execution. After time 2, there are 8 tokens on queue q ($length(q) = 8$). At time 3, node v reads 7 of the 8 tokens and executes, but it only consumes 3 of the 8 tokens. Thus the signal encounters a delay of at least $4 - 1 = 3$ time units before node v first produces data. This delay is due to the non-unity dataflow attributes and is called the *inherent latency* of a signal. Node u produces 4 tokens each time it executes at times 5, 7, and 9, as shown in Figure 1.5, and node v consumes 3 tokens when it executes at times 6, 8, 9, and 10. Thus queue q contains at most 10 tokens and at least 4 tokens during the interval $[1, 10]$.

In synthesizing an embedded real-time signal processing system from a PGM graph, we need to make sure the resulting system has enough memory to support the buffering of tokens in the graph queues. If it does not, data will be lost, and incorrect results will be produced by the application. We also need to be sure that the implementation meets the applications latency requirement. If it does not, the results produced will have little or no value and (depending on the application) may compromise the mission. Consider the execution trace recorded in Figure 1.5. Is 10 the upper bound on the number of tokens that will be buffered on queue q? Is 3 time units the maximum latency encountered by a signal? The answers to these questions depend on when nodes u and v execute. Since $prd(q) > cns(q)$, node v needs to execute more often than node u or data will accumulate indefinitely on queue q. For the two nodes in Figure 1.5, node v needs to execute $\frac{prd(q)}{cns(q)} = \frac{4}{3}$ times as often as node u to prevent data from accumulating on queue q. For example, if node u executes once every y time units and produces 4 tokens each time it executes, node v must execute 4 times every $3y$ time units and consume 3 tokens each time it executes. Node v needs to execute, on average, once every $\frac{3}{4}y$ time units. However, if we force node v to execute exactly once every $\frac{3}{4}y$ time units, then we create more delay than would naturally occur. In the execution trace of Figure 1.5, we have $y = 2$ and, hence, node u executes once every 2 time units. The 4 tokens produced by node u at time 7 enables two executions of node v. Forcing executions of node v to be periodic with period $\frac{3}{4} \cdot 2$ would create a delay of $\frac{6}{4}$ time units between the two executions that occurred at times 8 and 9. This is problematic as the extra delay is not needed for a valid graph execution. Since latency is a major concern in embedded signal processing applications, it is better to allow node v to execute whenever it is eligible.

When the system is implemented on a single processor, the signal may encounter more latency than it would if every node executed on its own processor. For example, in the execution trace in Figure 1.5, nodes u and v executed simultaneously. On a single processor, only one node can execute at a time. If node u is executed before node v, the samples already on queue q encounter a longer delay than if node v were executed before node u. The additional delay encountered when node u is executed before node v is called *imposed latency*. To control and manage imposed latency when the graph is implemented on a single processor, we execute the nodes according to a model of real-time execution. Imposed latency also affects the amount of buffering required on graph edges. For example, consider once again the simultaneous execution of nodes u and v at time 9 in Figure 1.5. If they are executed on a single processor, one of the nodes must execute before the other. If node v executes before node u, the length of queue q drops from 7 to 4 tokens before node u produces 4 tokens (after which $length(q) = 8$). If node u executes before node v the length of queue q grows from 7 to 11 tokens before node v consumes 3 tokens (after which $length(q) = 8$). We can manage the amount of buffering required on queue q by controlling when nodes u and v execute.

Thus, our goal in synthesizing a signal processing system for a single processor is to execute nodes such that we can manage both imposed latency and memory requirements. To achieve this goal, our synthesis method derives the precise execution rate of nodes. It will turn out that these rates are of the form of x executions in y time units. Given these rates, we map the nodes to real-time tasks. The tasks are executed according to the real-time rate-based execution (RBE) model [35], which is described in Section 3.3.1. Real-time task models provide determinism in the execution order of tasks, and the deterministic execution of nodes is required to manage latency and memory usage in the synthesis of signal processing systems from processing graphs.

A final note on notation. The dataflow attributes of queue q in Figure 1.5 were written out in full since the graph was simple enough that space on the page was not a problem. The parameter $init(q)$ was not shown since its value was 0. In general, the parameter $init(q)$ is assumed to be 0 unless otherwise specified. When the graph is more complex, the dataflow attributes are abbreviated as needed. For example, consider the PGM graph in Figure 1.6. This graph is essentially the same task chain in Figure 1.1 on page 4 but with annotations indicating the produce, threshold, and consume values for each queue.

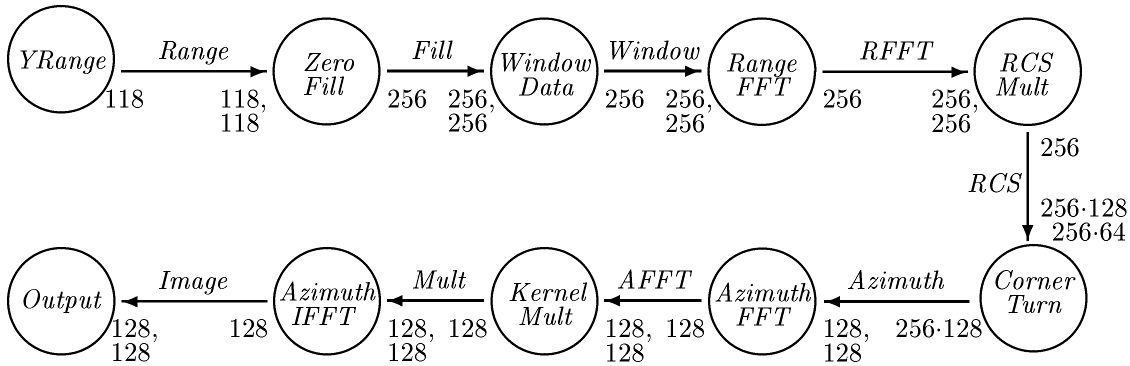


Figure 1.6: A PGM graph for the SAR application. The tail of each queue is annotated with its produce value. The head of each queue is annotated with its threshold and consume values. For example, the queue labeled *RCS* has $prd(q) = 256$, $thr(q) = 256{\cdot}128$, and $cns(q) = 256 \cdot 64$.

Only the actual values of each dataflow attribute are shown. The source node for the SAR graph in Figure 1.6 is labeled *YRange* and represents an external device that periodically produces data for the graph. Unlike, computation graphs, self-loops are not needed for source nodes to execute. The sink node, labeled *Output* represents an external device that executes whenever *Image* queue is over threshold. The SAR application is discussed in greater detail in Section 5.2.

A slightly more complicated PGM graph is shown in Figure 1.7 (adapted from [55]). This graph is the PGM graph corresponding to the INMARSAT mobile satellite receiver task graph shown in Figure 1.2 on page 5. (The processing nodes have been relabeled with letters.) The nodes labeled I_1, I_2, and O_1 represent external devices. Nodes I_1 and I_2 represent the input devices receiving the satellite signal. Node O_1 represents the output terminal accepting the processed signal. For this application, each queue's threshold is equal to its consume value. To reduce clutter in the figure, we have only labeled the non-unity dataflow attributes: produce values are located at the tail of the queue and consume values are at the head of the queue. The INMARSAT application is discussed in greater detail in Section 5.3.

1.1.4 Graph Models Similar to PGM

There are two AND processing graph models that are very similar to PGM: the SDF graph model [41] and the LASM [15, 16]. We work with PGM graphs, but our results

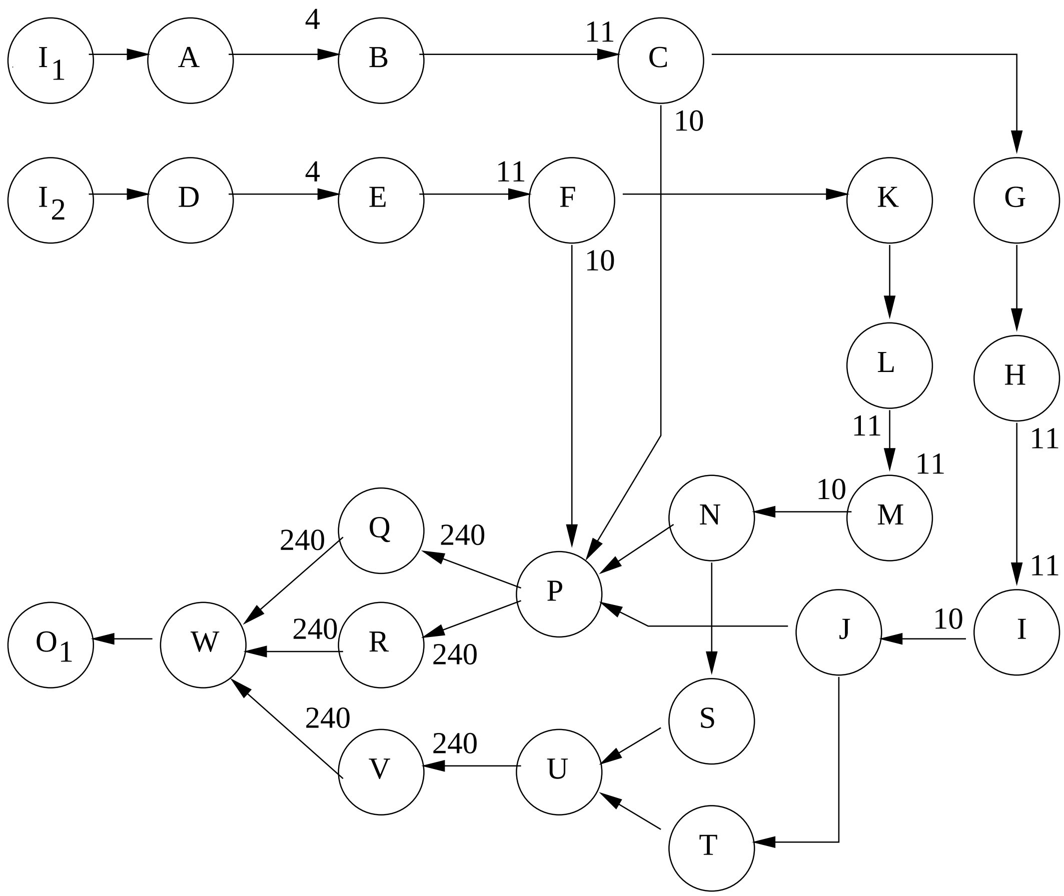


Figure 1.7: A PGM graph for an International Maritime Satellite (INMARSAT) mobile satellite receiver application whose task graph was shown in Figure 1.2 on page 5. The processing nodes have been relabeled with letters. The nodes labeled I_1, I_2, and O_1 represent external devices. Each queue's threshold is equal to its consume value. To reduce clutter in the figure, we have only labeled the non-unity dataflow attributes: produce values, $prd(q)$, are located at the tail of the queue and consume values, $cns(q)$, are at the head of the queue. For example, the queue directed from node A to node B has $prd(q) = 1$, $thr(q) = 4$, and $cns(q) = 4$.

can also be applied to these graph models.

The SDF graph model was created by Lee and Messerschmitt to develop signal processing applications and, like PGM, is an extension of Karp and Miller's computation graph model. The main difference between PGM graphs and SDF graphs is that a queue's threshold value must equal its consume value in an SDF graph. This means any SDF graph can also be represented as a PGM graph where $thr(q) = cns(q)$ for every queue in the graph. For example, the INMARSAT application graph in Figure 1.7 was originally presented by Ritz *et al.* as an SDF graph [55]. The execution semantics is the same for the two models.

The LASM was created by Chatterjee and Strosnider to represent a multimedia application's timing and logical processing requirements. It is remarkably similar to PGM, but was developed independently (and ten years later). The SAR graph in Figure 1.6 could represent either an LASM graph or a PGM graph. The main difference in the execution of nodes in the SAR graph under the two models is that the LASM adds latency to the signal by requiring a node u to execute exactly once every y_u time units (i.e., periodically). This is a reasonable requirement for multimedia applications, but it adds latency to a signal in a signal processing application. Another (minor) difference between the LASM and PGM models is that special *synchronization nodes* are used to synchronize chains of nodes in LASM graphs. Synchronization nodes take no time to execute, and simply represent points in the graph where multiple chains join or split. For example, Figure 1.8 is an LASM representation of the INMARSAT PGM graph of Figure 1.7. The rectangles represent synchronization nodes. Every node u in Figure 1.7 with multiple input queues has been mapped to a synchronization node followed by node u, and every node v in Figure 1.7 with multiple output queues has been mapped to node v followed by a synchronization node. For example, node C in Figure 1.7 has been mapped to node C followed by a synchronization node in Figure 1.8. This is a minor difference in the two models since synchronization nodes take no time to execute and simply represent points in the graph where multiple chains join or split. If LASM nodes were allowed to execute with deterministic rates of the form x executions in y time units, the analysis techniques presented in this dissertation for PGM graphs could be applied to LASM graphs.

1.2 Real-Time Systems

Our goal is to synthesize real-time signal processing systems from processing graphs. A real-time system is one that responds to external events within a probably bounded

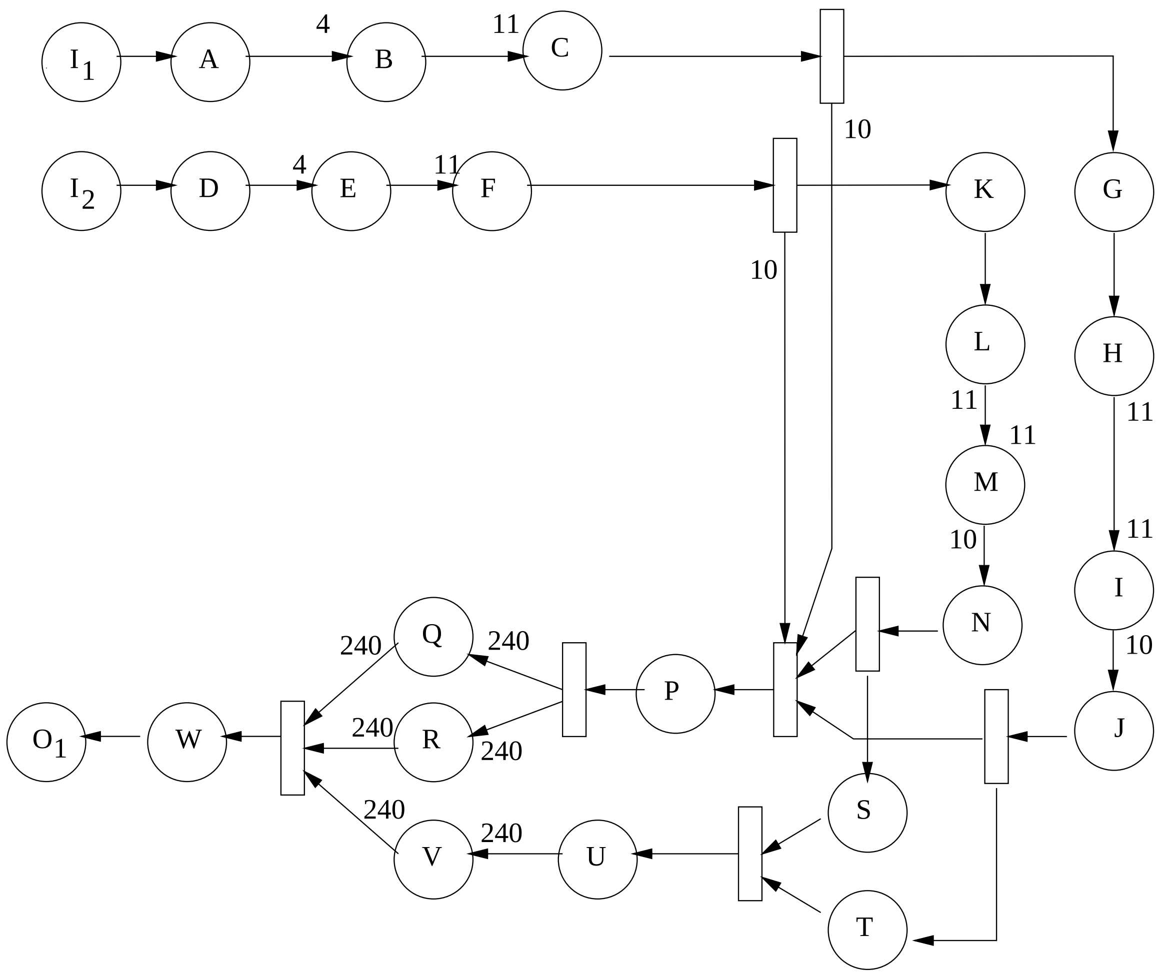


Figure 1.8: An LASM graph for the INMARSAT mobile satellite receiver application. This graph is logically equivalent to the PGM graph of Figure 1.7. The rectangles represent synchronization nodes. Every node u in Figure 1.7 with multiple input queues has been mapped to a synchronization node followed by node u, and every node v in Figure 1.7 with multiple output queues has been mapped to node v followed by a synchronization node. Each queue's threshold is equal to its consume value. The non-unity dataflow attributes have been identified using PGM notation. Non-unity produce values, $prd(q)$, are located at the tail of the queue and non-unity consume values, $cns(q)$, are at the head of the queue.

interval of time. Real-time systems are often classified as hard-real-time or soft-real-time. Hard-real-time systems usually require a guarantee that *all* processing completes within its time constraint *every time*. The temporal correctness of soft real-time systems is less stringent in that a stochastic measure of temporal correctness is sufficient (e.g., 90% of deadlines are met or the probability of meeting a task's deadline is 90%).

Real-time systems are frequently implemented as a collection of *tasks* in which each task is a sequential program that is invoked repeatedly. A single task invocation is called a *job*. Each invocation of a task creates a new job, and the time of the invocation is the job's *release time*. A job's *response time* is the time interval between when a job is released and when it completes execution. The maximum allowable response time for a job to be temporally correct is the task's *relative deadline*. The (absolute) *deadline* for a job is calculated by adding the task's relative deadline to the job's release time. If a job completes after its deadline, it is *late*. In hard real-time systems late jobs are not allowed. The task set is *independent* if the tasks do not share objects (such as variables), and no precedence constraints between the execution of jobs of different tasks are specified. When the first job of every task is released at the same time, the task set is *synchronous*. Since all task graphs define precedence constraints, any task set implementing the graph will be neither independent nor synchronous. Hence, real-time systems built from processing graphs have *dependent* and *asynchronous* tasks.

Hard real-time systems can be further classified by their models of task execution. Most real-time execution models define task execution to be *periodic*, *sporadic*, *aperiodic*, or a combination of these. A task whose jobs are released exactly once every p time units is periodic with period p. A sporadic task is a task where at at least p time units separate every job release. Hence, p is only a lower bound between job releases for sporadic tasks. The job release times of an aperiodic task are unspecified. The RBE model [35] uses an expected execution rate of the form x executions every y time units to describe the expected job release times of tasks. Thus, rate-based execution of a task is somewhere between sporadic and aperiodic: no more than x releases are expected in y time units, but job release times within the interval of y time units are not specified. The synthesis method developed in this dissertation uses the RBE task model to analyze latency in signal processing systems built from PGM graphs. Section 3.3.1 describes the RBE task model in greater detail.

When multiple job's are eligible for execution at the same time, a *scheduler* (or *scheduling algorithm*) decides the order in which the released jobs execute. How the jobs are selected and executed is the domain of real-time scheduling theory and the subject of the next section.

1.3 Real-Time Scheduling Theory

A *scheduler* decides when released jobs execute. Since hard-real-time systems require deterministic performance, it is important to determine if a task set can be scheduled such that all jobs meet their deadlines. A task set is *feasible* if and only if there exists a schedule in which there are no late jobs. A schedule is a sequence of jobs executed by the processor. For a given scheduling algorithm, a task set is *schedulable* if and only if the algorithm produces a schedule in which no jobs miss their deadlines. A task set may be schedulable under one scheduling algorithm and not under another. If a task set is schedulable, it is feasible. A feasible task set, however, may not be schedulable under all scheduling algorithms. An *optimal* scheduling algorithm can schedule any feasible task set.

There are many ways to classify scheduling algorithms. *Off-line* schedulers create a static schedule of jobs to be executed. The schedule is constructed "off-line" — i.e., before the task set commences execution. To reduce the storage space required for a static schedule, the schedule is usually implemented as a partial schedule that is executed in a loop. For example, consider the two-node PGM graph of Figure 1.9. Assume queue q is initialized with $(thr(q) - cns(q)) = 4$ tokens. One possible partial static schedule is $uvuvuvv$. That is, an execution of node u followed by and execution of node v repeated 3 times and then an extra execution of node v. This schedule could be executed in a loop such that the schedule observed at run time would be

$$uvuvuvv\ uvuvuvv\ uvuvuvv\ uvuvuvv\ uvuvuvv\ uvuvuvv\ldots$$

Execution Schedule A in Table 1.2 shows the length of queue q after each node in the partial schedule is executed. Another partial schedule is $uuuvvvv$, which can be represented in a more compact notation as $3u4v$. The effect of this schedule on the length of queue q of Figure 1.9 is shown in Execution Schedule B of Table 1.2.

On-line schedulers make scheduling decisions by running an algorithm to select the next job to execute when a scheduling decision is needed. *Priority* schedulers select the next job to execute based on a priority associated with each job. A *fixed-priority* scheduler is a priority scheduler in which all jobs of a task have the same priority. A *dynamic-priority* scheduler assigns a (possibly unique) priority to each job as it is released. A

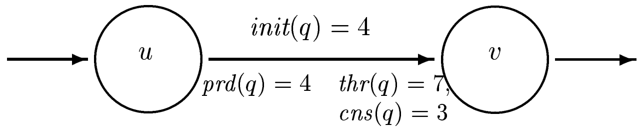


Figure 1.9: A two-node PGM chain in which queue q is initialized with 4 data tokens.

Execution Schedule A

Entry	Node	***length***(q)
–	–	4
1	u	8
2	v	5
3	u	9
4	v	6
5	u	10
6	v	7
7	v	4

Execution Schedule B

Entry	Node	***length***(q)
–	–	4
1	u	8
2	u	12
3	u	16
4	v	13
5	v	10
6	v	7
7	v	4

Table 1.2: A sequence of snapshots showing the length of queue q in Figure 1.9 after node u produces or node v consumes tokens.

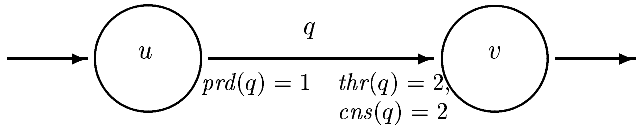
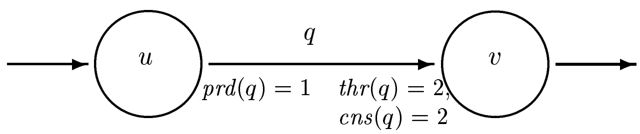


Figure 1.10: A two-node PGM chain in which queue q has no initial data.

scheduler that interrupts the execution of one job to execute another job is said to be *preemptive*. A *non-preemptive* scheduler executes each job to completion before another job begins execution on the same processor.

To understand the difference between non-preemptive and preemptive scheduling, consider the two-node graph in Figure 1.10. Assume nodes u and v are implemented as periodic tasks and executed according to a periodic execution model. Let the period of node u be 3 with its first release occurring at time 0, and the period of node v be 6 with its first release occurring at time 6. Assume both nodes require 2 time units to execute. If node u is given priority over node v with non-preemptive static-priority scheduling, Figure 1.11 shows the resulting execution and its effect on queue length. Even though node u is given priority over node v, node u is unable to execute when it is first released at time 9 because node v cannot be preempted from its execution. A preemptive execution of the graph under the same assumptions is shown in Figure 1.12. In the preemptive execution, node v is preempted at time 9 to let the higher priority node u execute. In this case, the non-preemptive scheduler requires less memory for buffering tokens than the preemptive scheduler: the maximum length of queue q is 3 for the non-preemptive execution, and it is 4 for the preemptive execution. Latency is also affected by the scheduling algorithm. In these examples, the non-preemptive scheduler creates a latency of $10 - 2 = 8$ time units for the first token produced by node u at time 2, and the preemptive scheduler creates a latency of $12 - 2 = 10$ time units. Part of the latency is due to the periodic execution model, which prevents node v from executing until time 6, and the rest of the imposed latency is due to the scheduling algorithm.

A common dynamic-priority scheduler is the earliest-deadline-first (EDF) scheduler [45]. Under EDF scheduling, the priority of a job is its absolute deadline, which is calculated by adding the task's relative deadline to the job's release time. At any instant in time, the preemptive EDF scheduling algorithm executes the job with the nearest (the earliest) deadline. When the relative deadline is equal to the task's period (in a periodic task model), the execution schedule created by EDF scheduling may be the same as the schedule created by static-priority scheduling. For example, consider the graph of



Time	**Node**	***length***(q)
0	–	0
2	u	1
5	u	2
8	u	3
10	v	1
12	u	2

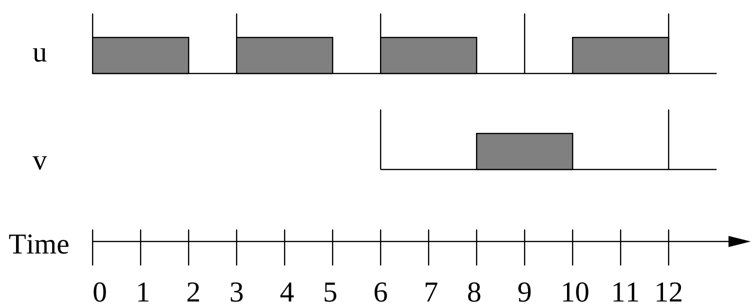
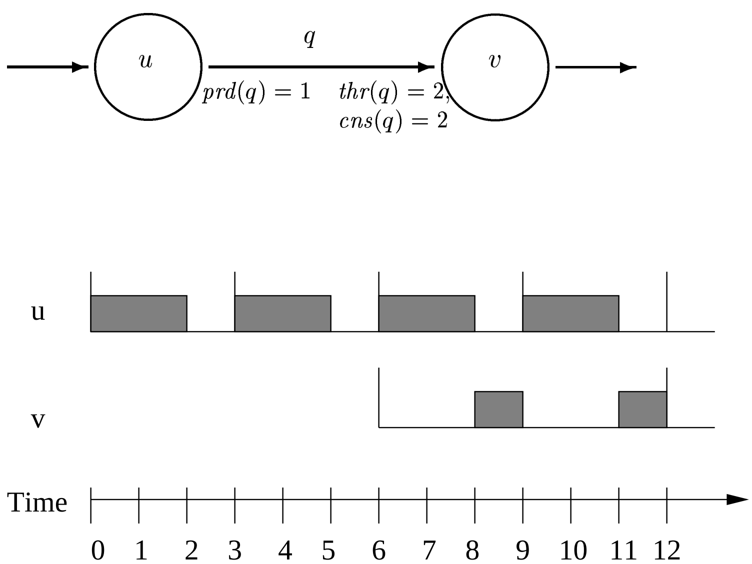
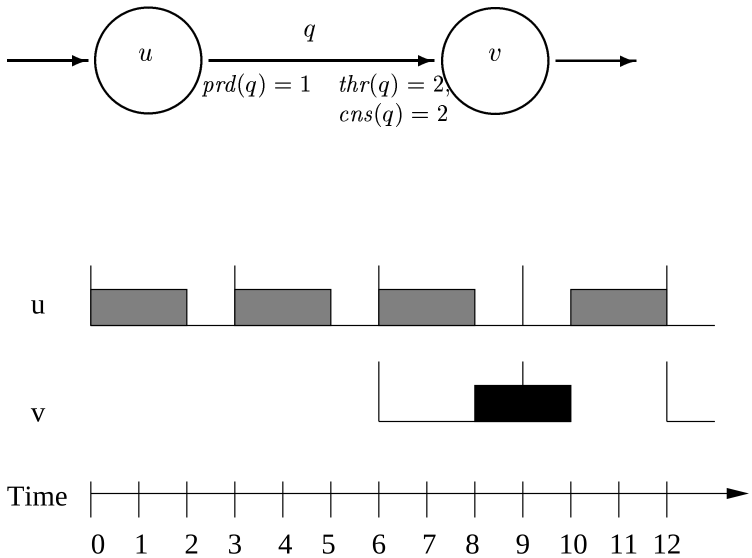


Figure 1.11: A two-node PGM graph in which nodes u and v are implemented as non-preemptive periodic tasks on a uniprocessor. The first release of node u is at time 0 and its period is 3. The first release of node v is at time 6 and its period is 6. This execution pattern and resulting lengths of queue q can be derived with either non-preemptive static-priority or dynamic-priority EDF scheduling. For non-preemptive static-priority scheduling, node u is given priority over node v. For dynamic-priority EDF scheduling, the relative deadline of each node is equal to its period.



Time	**Node**	***length***(q)
0	–	0
2	u	1
5	u	2
8	u	3
11	u	4
12	v	2

Figure 1.12: A two-node PGM graph in which nodes u and v are implemented as preemptive periodic tasks on a uniprocessor. The first release of node u is at time 0 and its period is 3. The first release of node v is at time 6 and its period is 6. This execution pattern and resulting lengths of queue q can be derived with either preemptive static-priority or dynamic-priority EDF scheduling. For preemptive static-priority scheduling, node u is given priority over node v. For dynamic-priority EDF scheduling, the relative deadline of each node is equal to its period and deadline ties are broken in favor of node u. Thus, node u preempts node v at time 9. Node v resumes its execution at time 11.



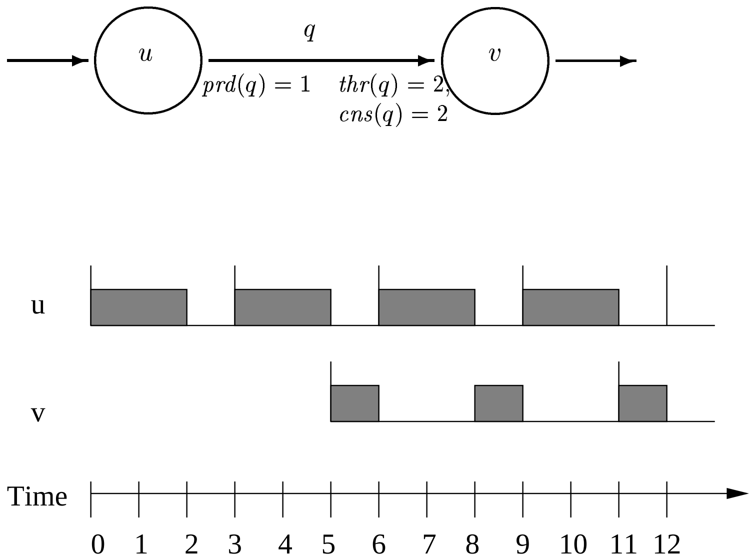
Time	**Node**	***length***(q)
0	–	0
2	u	1
5	u	2
8	u	3
10	v	1
12	u	2

Figure 1.13: A two-node PGM graph in which nodes u and v are implemented as periodic tasks on a uniprocessor and executed with EDF scheduling. The relative deadline for node v has been reduced from 6 to 4, and the EDF scheduler gives node u priority over node v when deadline ties occur. Thus, node v misses its deadline at time 9.

Figure 1.10 once again. As before, assume node u and v are executed according to a periodic execution model. Let the period of node u be 3 with its first release at time 0, and the period of node v be 6 with its first release at time 6. Assume both nodes require 2 time units to execute. Under these assumptions, the non-preemptive execution schedule of Figure 1.11 occurs under either EDF or static-priority scheduling, as does the preemptive execution schedule of Figure 1.12.

When EDF scheduling is used in an implementation of a graph, the relative deadline parameter of a node can be used to control latency and the buffering requirements for a queue. For example, if the relative-deadline parameter of node v in Figure 1.11 is changed from 6 to 4, the resulting execution schedule for preemptive EDF scheduling is the same as the schedule created by non-preemptive EDF and non-preemptive static-priority scheduling (shown in Figure 1.11). However, the relative-deadline parameter cannot be set arbitrarily small. If the relative deadline of node v is reduced to 3, the release of node v at time 6 will miss its deadline, as shown in Figure 1.13. In this case, the graph is not schedulable with EDF scheduling and the assumed scheduling parameters. Either a faster processor is required, which reduces the execution times of nodes u and v, or the deadline of node v must be increased before the graph is schedulable once again.

The results in this dissertation are based on the use of preemptive EDF scheduling



Time	**Node**	$\boldsymbol{length}(q)$
0	–	0
2	u	1
5	u	2
8	u	3
10	v	1
12	u	2

Figure 1.14: A two-node PGM graph in which nodes u and v are implemented as RBE tasks on a uniprocessor and executed with preemptive EDF scheduling. The nodes are released as soon as they are eligible for execution.

and the RBE task model. The RBE task model is used because it provides more flexibility in scheduling the nodes and introduces less imposed latency than the periodic execution model. We can say that node u executes once every 3 time units and node v executes once every six time units without requiring periodic execution of the two nodes. For example, consider the preemptive-EDF-rate-based execution of nodes u and v shown in Figure 1.14. Node u completes its second execution at time 5, at which time node v is released and executed immediately. Thus, the latency for the first token produced by node u is now $9 - 2 = 7$ time units — 1 time unit less than the minimum latency created under the periodic execution model. This illustrates the impact of scheduling on latency.

Chapters 3 and 4 discuss in detail the effect of relative-deadline parameters on the schedulability of the task set and latency when the RBE task model is combined with EDF scheduling.

1.4 Research Approach and Contributions

The goal of this research is to develop a method for building predictable real-time systems from PGM graphs. The central **thesis** of this project is that

> *Real-time systems can be synthesized from PGM processing graphs by applying existing real-time scheduling theory. Moreover, properties of PGM graphs and*

scheduling algorithms can be exploited to quantify and manage latency and memory usage requirements in signal processing applications.

Our research approach is to first identify and quantify inherent properties in PGM processing graphs such as node execution rates and latency. Second, the latency imposed by an implementation of the graph is identified and quantified. Third, the management of latency is demonstrated using existing processing graphs from the literature and actual applications. We leave open the problem of managing memory requirements in the synthesis of real-time systems from processing graphs, though, recently, we have made significant progress toward solving this problem as well [28, 27, 29]. The advantage of our approach is that we can identify the properties of the graph that affect latency and memory requirements independent of the implementation. We can then evaluate an implementation by the amount of additional latency or memory usage it imposes on top of the latency and memory usage inherent in a graph.

As with any real-time application, a signal processing graph must continuously process data at the rates of a set of input devices, such as sonobuoys, dipping sonars, or radars, and send the results to a set of external devices, such as displays or other computers, without the loss of data. In this dissertation, we identify the rates at which nodes need to execute to keep up with the graph input data rates and then map the nodes to real-time tasks that are executed according to the RBE model to ensure that no data is lost during graph execution.

Once the processing graph has been mapped to a set of real-time tasks and a scheduling algorithm selected that executes the tasks according to the RBE model of execution, issues such as latency and memory usage can be addressed. We identify and bound two sources of latency in real-time systems created from processing graphs. The first, *inherent latency*, is latency defined by the dataflow attributes and topology of the processing graph. The second source of latency, *imposed latency*, comes from the scheduling and execution of nodes in the graph. We develop a framework for evaluating and managing latency by deriving upper and lower bounds for both types of latency as functions of the data flow attributes and graph topology. The bounds on inherent latency assist signal processing engineers in selecting dataflow attributes for queues. No matter how the graph is implemented, the latency a sample encounters will never be less than the inherent latency. If the inherent latency is too large, the signal processing engineer must choose new dataflow attributes for the graph edges or redesign the graph. The bounds for imposed latency are used in the selection of relative deadline parameters that are associated with each node for EDF scheduling, and real-time scheduling theory is used to prove that the implementation meets the application's latency requirements by showing that the resulting task set is schedulable.

Although the results of this dissertation are based on PGM graphs, they are fundamental to AND processing graph models. Thus, our results are applicable to systems developed with other processing graph notations such as Lee and Messerschmitt's Synchronous Dataflow (SDF) graphs [41] or Chatterjee and Strosnider's Logical Application Stream Model (LASM) [15, 16].

1.5 Related Work

The U.S. Navy's PGM [50] is based on the computation graph model introduced by Karp and Miller in 1966 [39]. As described in Section 1.1.2, the computation graph model is very similar to PGM. A primary difference between computation graphs and PGM graphs is that computation graphs represent the parallel execution of finite computations, and PGM graphs represent the continuous execution of signal processing applications. Thus, Karp and Miller were primarily concerned with the termination of computation graphs, and their techniques for deriving node execution schedules and bounding the length of queues do not apply to the execution sequence of nodes in a PGM graph representing a signal processing application.

In 1996, Bhattacharyya, Murthy, and Lee published a method for software synthesis from dataflow graphs [9]. Their software synthesis method is based on the static scheduling of Lee and Messerschmitt's SDF graphs [41]. As described in Section 1.1.4, any SDF graph can be represented as a PGM graph where each queue's threshold is equal to its consume value. The main goal of Bhattacharyya *et al.*'s software synthesis method and the related scheduling research based on SDF graphs has been to minimize memory usage by creating off-line scheduling algorithms [41, 54, 64, 55, 9]. 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 signal processing applications, dynamic on-line scheduling creates less imposed latency than static scheduling [28, 27, 29]. 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.

From the real-time literature, PGM is most closely related to LASM, which was created by Chatterjee and Strosnider to represent a multimedia application's timing and logical processing requirements [15, 16]. It is very similar to PGM, but assumes a different real-time execution model than we use. Our work improves on the analysis of LASM graphs by not requiring periodic execution of the nodes in the graph. Instead, graph execution is modeled with the RBE process model to more accurately predict processor demand without imposing the additional latency created by periodic node execution.

Mok's Software Automation for Real-Time Operations (SARTOR) project also uses processing graphs to develop real-time systems. The node execution semantics of the SARTOR graph model, however, require a node to execute when data exists on *any* of its input queues [47, 48]. When the node executes, it consumes all of the data on *all* of its input queue. This execution semantics would result in erroneous signal processing results if it were applied to PGM graphs. The SARTOR project achieves deterministic execution of a graph by mapping nodes to periodic tasks — rather than RBE tasks, as is done here. As shown in Section 1.3, forcing periodic execution of PGM nodes creates unwanted latency. Thus, the synthesis techniques of the SARTOR project are very different than the techniques presented here. Moreover, the synthesis techniques of the SARTOR project are not well suited to synthesizing embedded signal processing systems from processing graphs.

Jeffay's Real-Time Producer/Consumer (RTP/C) paradigm is another synthesis method that uses the structure of the graph to help derive the execution rates of tasks that implement nodes in the graph [33]. Queues in an RTP/C graph have unity produce values, but are allowed to have non-unity threshold values. The consume amount is equal to the threshold of the queue. The biggest difference between RTP/C graphs and PGM graphs, however, is that RTP/C graphs are OR graphs rather than AND graphs. Since our results are based on the execution of nodes when all of their input queues are over threshold, only our results for PGM chains are applicable to RTP/C graphs.

Our latency analysis is related to the work of Gerber *et al.* in guaranteeing end-to-end latency requirements on a single processor [22, 23]. However, Gerber *et al.* map a task graph to a periodic task model in the synthesis of real-time message-based systems rather than the RBE model. 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. Since our analysis techniques are rate-based rather than periodic, our synthesis method does not need extra synchronization tasks.

1.6 Dissertation Overview

Chapter 2 explores the real-time properties of PGM graphs by deriving when nodes in a PGM graph must execute if they are to process a continuous input signal without losing data. First, the notation and terminology used throughout this dissertation are presented. We then begin the exploration of the real-time properties of PGM graphs by trying to understand the impact of non-unity dataflow attributes on node execution patterns and the minimal buffering required for each queue. We informally describe the execution patterns as execution rates, and then formalize the definition of a node execution rate and derive equations that compute the execution rate of any node in a PGM graph from its immediate predecessors and the dataflow attributes of its input queues.

The software synthesis method presented in Chapter 3 is used to build predictable real-time system from processing graphs. Algorithms are presented that compute the execution rate of every node in a PGM graph using the equations derived in Chapter 2. Once node execution rates have been computed, the graph can be mapped to real-time tasks according to a task model. The RBE task model has been selected for this step since it provides the most natural mapping and does not introduce any undue latency. For completeness, we describe the RBE task model and then present a sufficient condition for determining the schedulability of the resulting real-time system under EDF scheduling.

Chapter 4 addresses the issue of managing latency in an implementation of a PGM graph. There are two types of latency: inherent and imposed. Inherent latency in a graph is created by non-unity dataflow attributes and graph topology. Although inherent latency cannot be managed during the synthesis step, we quantify the inherent latency in a graph and use it as a lower bound for the latency achievable in an implementation of the graph. Imposed latency comes from the scheduling and execution of nodes in the graph, and it must be managed. Thus, we quantify imposed latency and discusses its management in the synthesis of real-time systems from processing graphs. The analysis of both types of latency begins with chains and is then extended to general graphs to simplify the presentation. The analysis techniques, theorems, and equations presented in Chapter 4 are used by system engineers to evaluate the feasibility of latency requirements. The results also provide a framework for tools to assist signal processing engineers in the creation of the signal processing graph by quantifying the impact of processing graph parameters on latency early in the design process when it is most cost effective to make changes.

To illustrate our techniques for managing latency in the synthesis of real-time systems from processing graphs, three PGM graphs from the literature and actual implementations are analyzed in Chapter 5. The first is a SAR application [65]. The SAR system can be used to identify man-made objects on the ground or in the air by producing high-resolution, all-weather images in real-time [49]. The second application evaluated is an International Maritime Satellite (INMARSAT) mobile satellite receiver application [64, 55]. INMARSAT is a global maritime communication and navigational system used in the commercial shipping industry. We conclude our case studies by evaluating latency in an anti-submarine warfare (ASW) system — the Directed Low Frequency Analysis and Recording (DIFAR) acoustic signal processing program from the Airborne Low Frequency Sonar (ALFS) subsystem of the LAMPS MK III anti-submarine helicopters. The topologies of the processing graphs for these three applications ranges from a simple chain for the SAR application to a cyclic graph containing over 80 nodes and 400 queues for the DIFAR application.

Our conclusions and open problems are summarized in Chapter 6.

Chapter 2

Real-Time Properties of PGM Graphs

2.1 Introduction

This chapter explores the real-time properties of PGM graphs by deriving when nodes in a PGM graph must execute if they are to process a continuous input signal without losing data. First, the notation and terminology used throughout this dissertation are presented in Section 2.2. We then begin the exploration of the real-time properties of PGM graphs in Section 2.3 by trying to understand the impact of non-unity dataflow attributes on node execution patterns and the minimal buffering required for each queue. We informally describe the execution patterns as execution rates, and then formalize the definition of a node execution rate in Section 2.4 and use the results of Section 2.3 to derive equations that compute the execution rate of any node in a PGM graph from its immediate predecessors and the dataflow attributes of its input queues.

2.2 Notation and Terminology

The notation and terminology used throughout this dissertation is an amalgamation of the notation and terminology used in *Graph Theory with Applications* by Bondy and Murty [12]; *Introduction to Algorithms* by Cormen, Leiserson, and Rivest [14]; and *Software Synthesis from Dataflow Graphs* by Bhattacharyya, Murthy, and Lee [9].

Let $\mathcal{Z}$ be the set of integers, $\mathcal{P}$ be a finite set of positive integers, and let $\mathcal{R}$ be a finite set of real numbers. For $a, b, c \in \mathcal{Z}$ and $b \neq 0$, if $a/b = c$ we say b divides a and represent it as $b|a$. We note that $b|a$ implies $a \bmod b = 0$. The greatest common divisor of the elements in $\mathcal{P}$ is denoted by $\gcd(\mathcal{P})$, and $\operatorname{lcm}(\mathcal{P})$ denotes the least common multiple of the elements in $\mathcal{P}$. The elements of $\mathcal{P}$ are *relatively prime* if $\gcd(\mathcal{P}) = 1$.

A processing graph is formally described as a *directed graph* (or *digraph*) $G = (V, E, \psi)$. The ordered triple (V, E, ψ) consists of a nonempty finite set V of *vertices*, a finite set of *edges* E, and an incidence function $\psi : E \to V \times V$ that associates with each edge of E an ordered pair of (not necessarily distinct) vertices of V. $\psi(e) = (u, v)$ represents a directed edge from u to v. In this dissertation, vertices are often called *nodes*, and *edges* of E are often called *queues*.

Consider an edge $e \in E$ and vertices $u, v \in V$ such that $\psi(e) = (u, v)$ is an edge from u to v. We say e joins u to v, or u and v are adjacent. If u and v are distinct, then $\{u, v\}$ is an *adjacent pair*. Whether or not u and v are distinct, u is a *predecessor* of v, and v is a *successor* of u. The vertex u is the *tail* or *source* vertex of e and v is the *head* or *sink* vertex of edge e. The edge e represents a buffered communication channel transporting data from node u to node v, and the vertex u is a *producer* of data for the *consumer* node v. For the producer/consumer relationship of u and v, 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 of all input nodes is denoted by $\mathcal{I}$ (i.e., $\mathcal{I} = \{v \mid v \in V \wedge \delta^-(v) = 0\}$). A vertex v with $\delta^+(v) = 0$ is an *output node*. The set of all output nodes is denoted by $\mathcal{O}$ (i.e., $\mathcal{O} = \{v \mid v \in V \wedge \delta^+(v) = 0\}$).

For $u, v \in V$, there is *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 $1 \leq i < k$.

A directed path $(w_0, w_1, w_2, \ldots, w_k)$ forms a cycle if $w_0 = w_k$. The cycle is *simple* if the vertices $w_1, w_2, \ldots, w_k$ are distinct. (For every cycle, there must exist a simple cycle.) Two paths $(w_0, w_1, w_2, \ldots, w_{k-1}, w_0)$ and $(w'_0, w'_1, w'{}_2, \ldots, w'{}_{k-1}, w'{}_0)$ are the same if there exists an integer j such that $w'{}_i = w_{(i+j) \bmod k}$ for $0 \leq i < k$. For example, in Figure 2.1a, the paths (u, v, w, u) and (v, w, u, v) are the same cycle. Two cycles are *disjoint* if they are not the same and there does not exist a vertex u that is in both cycles. If two cycles are not the same but they share a common vertex u, the cycles are *non-disjoint* — or *complex*. For example, the cycles (s, t, s) and (u, v, w, u) in Figure 2.1a are disjoint, but the cycles (u, v, w, u) and (s, t, u, s) in Figure 2.1b are non-disjoint. A path $u \rightsquigarrow v$ is a *chain* if $u \neq v$, $\delta^-(u) \leq 1$, $\delta^+(u) = 1$, $\delta^-(v) = 1$, and $\delta^+(w) = \delta^-(w) = 1$ for all $w \in \{\{u \rightsquigarrow v\} - \{u, v\}\}$. If there exists a path from $u \in V$ to $v \in V$, then u is an *ancestor*

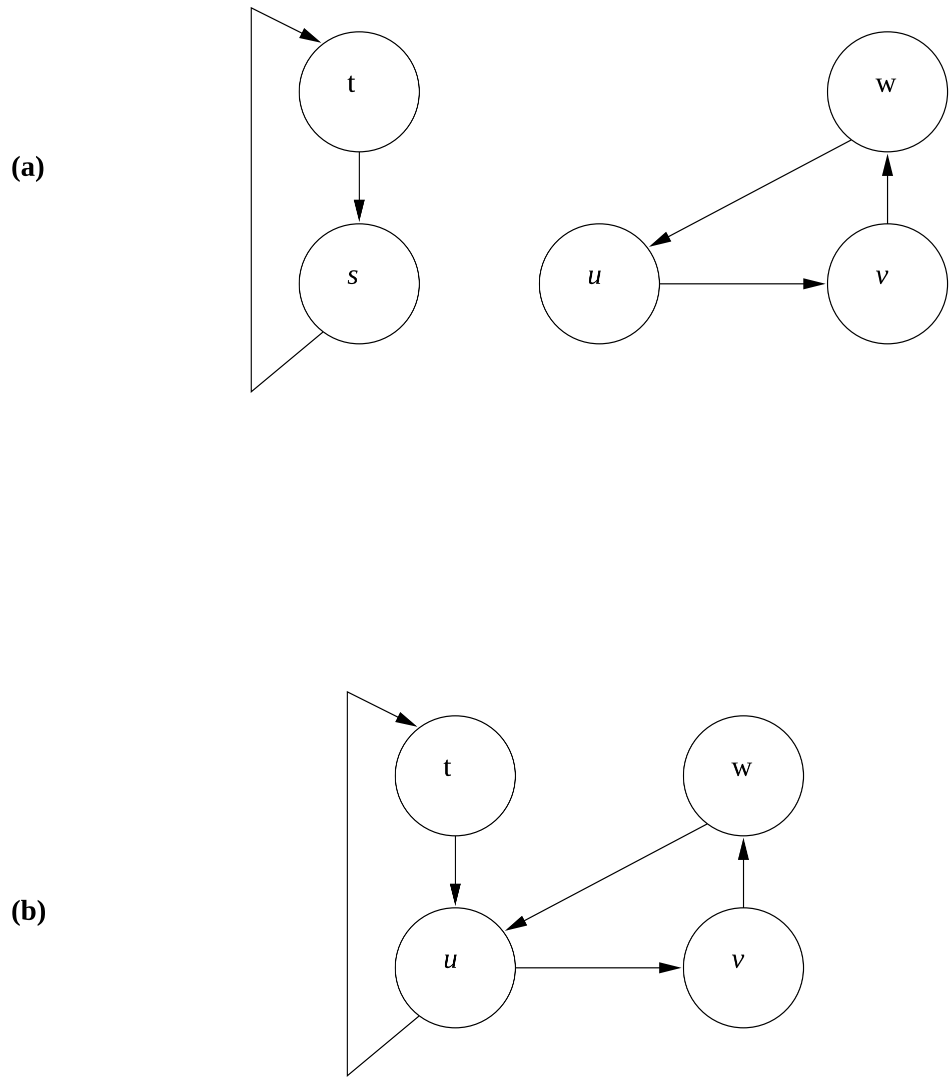


Figure 2.1: An example of disjoint and non-disjoint cycles. **(a)** The cycles (s, t, s) and (u, v, w, u) are disjoint. **(b)** The cycles (t, u, t) and (u, v, w, u) are non-disjoint.

of v and v is a *descendant* of u.

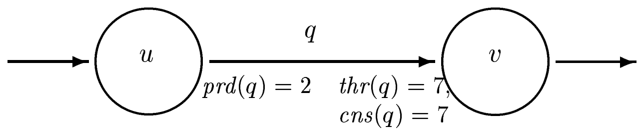
A graph source node for a PGM graph represents an external sensor device, and a graph sink node represents and external output device such as a display. A graph source node is also called an input node, and a graph sink node is also called an output node. Let $\mathcal{I}$ denote the set of input nodes for a PGM graph G, and let $\mathcal{O}$ denote the set of output nodes for a PGM graph G. 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 $w \in \mathcal{O}$.

When discussing PGM directed graphs, we use the term graph rather than digraph or directed graph since all PGM graphs are digraphs. We also use the terms nodes and queues to refer to elements of V and E respectively. The value of dataflow attributes produce, threshold, and consume for queue q are denoted $\mathit{prd}(q)$, $\mathit{thr}(q)$, and $\mathit{cns}(q)$ respectively. The number of initial tokens on queue q is denoted $\mathit{init}(q)$, and the length of queue q is denoted $\mathit{length}(q)$. Before any nodes execute, $\mathit{init}(q) = \mathit{length}(q)$.

2.3 Node Executions and Minimal Buffering Requirements

In task graph systems that require unity dataflow attributes (i.e., produce, threshold, and consume values all one), deriving the execution rates of nodes is relatively straightforward. Deriving the execution rates of nodes in PGM graphs is not. In this section, we present a series of examples that illustrate the impact of non-unity dataflow attributes on node execution and quantify the number of times a producer node must execute before its consumer node is eligible for execution. We also derive several bounds related to minimal buffering requirements that will be used throughout this dissertation.

To eliminate the influence of scheduling on node executions, assume each node executes on its own processor as soon as all of its input queues are over threshold. For our first example, consider the two node chain and the snapshots of queue q in Figure 2.2 where $\mathit{length}(q)$ is the number of tokens on queue at time t_i. Queue q is annotated with its produce, threshold, and consume values below the queue; it has no initial data. Node u produces 2 tokens every time it executes. The input queue to node v has a threshold of 7 and node v consumes 7 tokens after it executes. Since each execution of node u produces 2 tokens, 4 executions of node u are required before the first execution of node v occurs at time t_5. When node v executes (at time t_5 in the snapshot table), it consumes 7 of the 8 tokens on queue q. Hence, only 3 additional executions of node u (producing 6 more

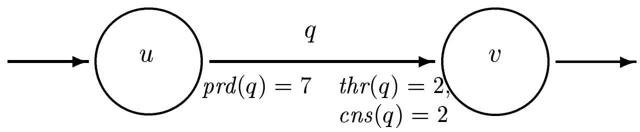


Time	data produced by node u	data consumed by node v	***length***(q)
0			0
t_1	2		2
t_2	2		4
t_3	2		6
t_4	2		8
t_5		7	1
t_6	2		3
t_7	2		5
t_8	2		7
t_9		7	0

Figure 2.2: A two-node PGM graph and snapshot sequence. The produce amount is 2, and the threshold and consume values are both 7. The queue has no initialized data. The table shows snapshots of the length of queue q at time t after node u produces or node v consumes tokens.

tokens for a total of 7 tokens on queue q) are needed for node v to execute a second time. After node v executes the second time (at time t_9), it consumes all 7 tokens on queue q leaving it in the same state as it began, with 0 tokens. Simulating subsequent executions would show that the number of executions required of node u to produce enough data for node v to execute continues to alternate between 4 and 3 executions. Every 7 executions of node u will produce 14 tokens and result in node v executing twice. If node u executes once every y_u time units, it will execute 7 times in $7y_u$ time units and node v will execute at a rate of only 2 times every $7y_u$ time units. Thus, node v executes at a slower rate than node u. In general, whenever $prd(q) < cns(q)$, the consumer node will execute at a slower rate than its producer.

What happens when the produce amount is greater than the threshold? Consider the two node chain of Figure 2.3 where $prd(q) = 7$, $cns(q) = 2$, and $thr(q) = 2$. These dataflow attributes result in the first execution of node u enabling 3 executions of node v and the second execution of node u enabling 4 executions of node v. This is because the first 3 executions of node v left 1 token on queue q. After 2 executions of node u and the resulting 7 executions of node v, queue q is left in its original state: containing 0 tokens. Thus, every 2 executions of node u will produce 14 tokens and result in 7 executions



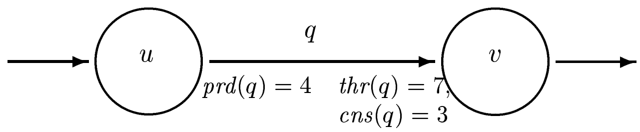
Time	data produced by node u	data consumed by node v	$\boldsymbol{length}(q)$
0			0
t_1	7		7
t_2		2	5
t_3		2	3
t_4		2	1
t_5	7		8
t_6		2	6
t_7		2	4
t_8		2	2
t_9		2	0

Figure 2.3: A second two-node PGM graph and snapshot sequence. Although the threshold and consume values are still equal, this time they are set to 2, and the produce amount is 7.

of node v — node v now executes at a faster rate than node u. In general, whenever $prd(q) > cns(q)$, the consumer node will execute at a faster rate than its producer.

Finally, consider the two node chain of Figure 2.4. Node u produces 4 tokens every time it executes. Node v has a threshold of 7 tokens and consumes 3 tokens after it executes. Since queue q is not initialized, node u must fire twice before queue q is over threshold and node v executes for the first time. After node v executes, it consumes only 3 tokens — leaving 5 tokens on queue q. The third execution of node u produces 4 more tokens (for a total of 9 tokens on queue q) and node v executes again, consuming 3 more tokens. The next execution of node u results in 10 tokens on queue q, and node v is able to execute twice — leaving 4 tokens on queue q, which is the same number of tokens that were on queue q after the first execution of node u. Thus, subsequent executions of node u and node v follow this same pattern: $uvuvuvv$. Therefore, if node u executes once every y_u time units, node v will execute with a rate of 4 times every $3y_u$ time units.

These examples demonstrate that the number of tokens on queue q at time t is a function of the the queue's dataflow attributes and the number of executions of nodes u and v prior to time t. Since node v executes whenever queue q contains at least



Time	data produced by node u	data consumed by node v	***length***(q)
0			0
t_1	4		4
t_2	4		8
t_3		3	5
t_4	4		9
t_5		3	6
t_6	4		10
t_7		3	7
t_8		3	4

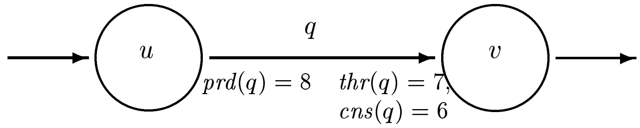
Figure 2.4: A third two-node PGM graph and snapshot sequence. In this chain, the dataflow attributes $prd(q)$, $thr(q)$, and $cns(q)$ all have different values. The produce amount is 4, the threshold value is 7, and the consume amount is 3.

$thr(q)$ tokens and it consumes $cns(q)$ tokens each time it executes, queue q will always contain at least $(thr(q) - cns(q))$ tokens after node v executes for the first time. Note, however, that this lower bound on the minimum number of tokens on q is not tight. Consider, for example, the chain shown in Figure 2.5 where the dataflow attributes are $prd(q) = 8$, $thr(q) = 7$, $cns(q) = 6$. In this case, $thr(q) - cns(q) = 1$, but there will always be at least two tokens in the queue. We now present two theorems that bound the minimum number of tokens on queue q after the first execution of node v and the maximum number of tokens that can be on queue q without the queue being over threshold. The first, Theorem 2.3.1, requires $init(q) = 0$. We then relax this restriction to allow initialized data on queue q. These buffering bounds will be used extensively throughout this dissertation.

Theorem 2.3.1. *Let $\psi(q) = (u, v)$ with $init(q) = 0$. The number of tokens on queue q after node v has executed once is at least $MinTokens(q)$ where*

$$MinTokens(q) = \left\lceil \frac{thr(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \qquad (2.1)$$

and the maximum number of tokens queue q can hold without being over threshold is



Time	**data produced by node** u	**data consumed by node** v	***length***(q)
0			0
t_1	8		8
t_2		6	2
t_3	8		10
t_4		6	4
t_5	8		12
t_6		6	6
t_7	8		14
t_8		6	8
t_9		6	2

Figure 2.5: A fourth two-node PGM graph and snapshot sequence. The dataflow attributes $prd(q)$, $thr(q)$, and $cns(q)$ are 8, 7, and 6 respectively.

MaxUnderThr(q) *where*

$$MaxUnderThr(q) = \begin{cases} thr(q) - \gcd(prd(q), cns(q)) & \textit{if } \gcd(prd(q), cns(q)) \mid thr(q) \\ \left\lfloor \frac{thr(q)}{\gcd(prd(q), cns(q))} \right\rfloor \cdot \gcd(prd(q), cns(q)) & \textit{otherwise} \end{cases} \tag{2.2}$$

Before proving Theorem 2.3.1, we demonstrate its application using the chains of Figures 2.4 and 2.5. The dataflow attributes for queue q in Figure 2.4 are $prd(q) = 4$, $thr(q) = 7$, and $cns(q) = 3$. Using Equation (2.1), the number of tokens queue q contains after the first execution of node v is at least

$$\begin{aligned} MinTokens(q) &= \left\lceil \frac{thr(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \\ &= \left\lceil \frac{7}{\gcd(4,3)} \right\rceil \cdot \gcd(4,3) - 3 \\ &= \left(\left\lceil \frac{7}{1} \right\rceil \cdot 1 \right) - 3 = 4. \end{aligned}$$

Since $\gcd(prd(q), cns(q)) = \gcd(4,3) = 1$, the gcd of the produce and consume values divides the threshold amount (i.e., $\gcd(prd(q), cns(q)) \mid thr(q)$). Therefore, by Theorem 2.3.1, the maximum number of tokens queue q can hold without being over threshold is

$$MaxUnderThr(q) = thr(q) - \gcd(prd(q), cns(q)) = 7 - 1 = 6.$$

If node v is always able to execute as soon as queue q is over threshold and it completes its execution before node u produces any more tokens, queue will never contain more than

$$MaxUnderThr(q) + prd(q) = 6 + 4 = 10$$

tokens. Thus, any implementation of the graph will require buffer space for at least 10 tokens. The actual upper bound for the length of queue q, however, is determined by the scheduling algorithm and the topology of the graph.

Now consider the chain of Figure 2.5 where $prd(q) = 8$, $thr(q) = 7$, and $cns(q) = 6$. In this case, $\gcd(prd(q), cns(q)) = \gcd(8, 6) = 2$ and

$$\begin{aligned} MinTokens(q) &= \left\lceil \frac{thr(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \\ &= \left\lceil \frac{7}{\gcd(8, 6)} \right\rceil \cdot \gcd(8, 6) - 6 \\ &= \left(\left\lceil \frac{7}{2} \right\rceil \cdot 2 \right) - 6 = (4 \cdot 2) - 6 = 2. \end{aligned}$$

Since $\gcd(prd(q), cns(q)) = 2$, which does not divide $thr(q) = 7$, the maximum number of tokens the queue can hold without being over threshold is

$$\begin{aligned} MaxUnderThr(q) &= \left\lfloor \frac{thr(q)}{\gcd(prd(q), cns(q))} \right\rfloor \cdot \gcd(prd(q), cns(q)) \\ &= \left\lfloor \frac{7}{\gcd(8, 6)} \right\rfloor \cdot \gcd(8, 6) \\ &= \left(\left\lfloor \frac{7}{2} \right\rfloor \cdot 2 \right) = 3 \cdot 2 = 6. \end{aligned}$$

Thus, as shown in the snapshots recording the execution of nodes u and v in Figure 2.5, queue q will contain between 2 and

$$MaxUnderThr(q) + prd(q) = 6 + 8 = 14$$

tokens if node v executes as described above.

Proof of Theorem 2.3.1: Let $\mathit{length}(q)$ be the number of token on queue q. Since $\mathit{init}(q) = 0$, $\mathit{length}(q) = 0$ initially. Let k_u denote a number of executions of node u, and k_v denote a number of executions of node v. Observe that $\mathit{length}(q)$ is always of the form $\mathit{length}(q) = k_u \cdot \mathit{prd}(q) - k_v \cdot \mathit{cns}(q)$ when $\mathit{init}(q) = 0$, and that after node v first executes, queue q will always contain at least $\mathit{thr}(q) - \mathit{cns}(q)$ tokens. Thus, to prove Equation (2.1), we must find the smallest value the expression $k_u \cdot \mathit{prd}(q) - k_v \cdot \mathit{cns}(q)$ can be without being less than $\mathit{thr}(q) - \mathit{cns}(q)$.

Observe that for all integer values of k_u and k_v that satisfy the expression

$$k_u \cdot \mathit{prd}(q) - k_v \cdot \mathit{cns}(q) \geq \mathit{thr}(q) - \mathit{cns}(q),$$

there exists integers a, b, and k such that

$$\begin{aligned} k_u \cdot \mathit{prd}(q) - k_v \cdot \mathit{cns}(q) &= k_u \cdot (a \cdot \gcd(\mathit{prd}(q), \mathit{cns}(q))) - k_v \cdot (b \cdot \gcd(\mathit{prd}(q), \mathit{cns}(q))) \\ &= (k_u \cdot a - k_v \cdot b) \cdot \gcd(\mathit{prd}(q), \mathit{cns}(q)) \\ &= k \cdot \gcd(\mathit{prd}(q), \mathit{cns}(q)). \end{aligned}$$

Thus, the lower bound on the length of queue q, is the smallest integer k that satisfies the expression

$$k \cdot \gcd(\mathit{prd}(q), \mathit{cns}(q)) \geq \mathit{thr}(q) - \mathit{cns}(q). \tag{2.3}$$

The smallest integer k that satisfies Equation (2.3) is

$$k = \left\lceil \frac{\mathit{thr}(q) - \mathit{cns}(q)}{\gcd(\mathit{prd}(q), \mathit{cns}(q))} \right\rceil$$

Therefore, the minimum number of tokens that will ever be on queue q after node v has executed at least once is:

$$\begin{aligned} \mathit{MinTokens}(q) &= \left\lceil \frac{\mathit{thr}(q) - \mathit{cns}(q)}{\gcd(\mathit{prd}(q), \mathit{cns}(q))} \right\rceil \cdot \gcd(\mathit{prd}(q), \mathit{cns}(q)) \\ &= \left(\left\lceil \frac{\mathit{thr}(q)}{\gcd(\mathit{prd}(q), \mathit{cns}(q))} \right\rceil - \frac{\mathit{cns}(q)}{\gcd(\mathit{prd}(q), \mathit{cns}(q))} \right) \cdot \gcd(\mathit{prd}(q), \mathit{cns}(q)) \\ &= \left\lceil \frac{\mathit{thr}(q)}{\gcd(\mathit{prd}(q), \mathit{cns}(q))} \right\rceil \cdot \gcd(\mathit{prd}(q), \mathit{cns}(q)) - \mathit{cns}(q) \end{aligned}$$

which proves that Equation (2.1) is the lower bound on the length of queue q after node v has executed at least once.

We now prove Equation (2.2) derives the largest number of tokens queue q can hold without being over threshold. Since $length(q)$ is always of the form

$$k_u \cdot prd(q) - k_v \cdot cns(q),$$

we must find the largest value this expression can have while still being less than $thr(q)$. For all integer values of k_u and k_v that satisfy the expression

$$k_u \cdot prd(q) - k_v \cdot cns(q) < thr(q),$$

there exists integers a, b, and k such that

$$\begin{aligned} k_u \cdot prd(q) - k_v \cdot cns(q) &= k_u \cdot (a \cdot \gcd(prd(q), cns(q))) - k_v \cdot (b \cdot \gcd(prd(q), cns(q))) \\ &= (k_u \cdot a - k_v \cdot b) \cdot \gcd(prd(q), cns(q)) \\ &= k \cdot \gcd(prd(q), cns(q)). \end{aligned}$$

Thus, the largest number of tokens queue q can hold without being over threshold, is the largest integer k that satisfies the expression

$$k \cdot \gcd(prd(q), cns(q)) < thr(q).$$

Observe that $k = \left\lfloor \frac{thr(q)}{\gcd(prd(q), cns(q))} \right\rfloor$ is the largest integer such that

$$k \cdot \gcd(prd(q), cns(q)) \leq thr(q).$$

If $k = \left\lfloor \frac{thr(q)}{\gcd(prd(q), cns(q))} \right\rfloor$ and $\gcd(prd(q), cns(q))$ does not divide $thr(q)$, then

$$k \cdot \gcd(prd(q), cns(q)) = \left\lfloor \frac{thr(q)}{\gcd(prd(q), cns(q))} \right\rfloor \cdot \gcd(prd(q), cns(q)) < thr(q).$$

Hence, Equation (2.2) holds.

If $\gcd(prd(q), cns(q)) \mid thr(q)$, then $k = \frac{thr(q)}{\gcd(prd(q), cns(q))}$ and

$$\begin{aligned} k \cdot \gcd(prd(q), cns(q)) &= \frac{thr(q)}{\gcd(prd(q), cns(q))} \cdot \gcd(prd(q), cns(q)) \\ &= thr(q). \end{aligned}$$

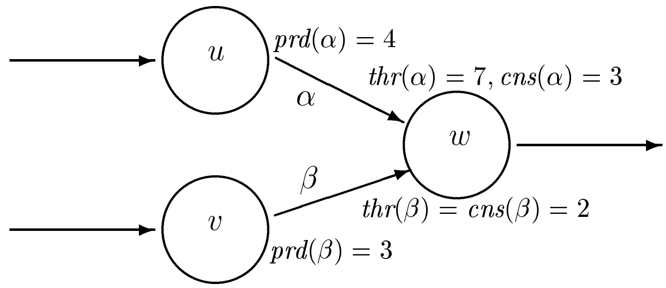


Figure 2.6: A PGM node with two input queues.

In this case, the maximum number of tokens queue q can hold without being over threshold is

$$\begin{aligned}(k-1)\cdot \gcd(prd(q), cns(q)) &= (\frac{thr(q)}{\gcd(prd(q), cns(q))} - 1)\cdot \gcd(prd(q), cns(q)) \\ &= (\frac{thr(q)}{\gcd(prd(q), cns(q))}\cdot \gcd(prd(q), cns(q))) - \gcd(prd(q), cns(q)) \\ &= thr(q) - \gcd(prd(q), cns(q)).\end{aligned}$$

Hence, Equation (2.2) also holds when $\gcd(prd(q), cns(q)) \mid thr(q)$. □

Theorem 2.3.1 is independent of the number of input queues to a node and holds for each input queue individually. Consider the graph of Figure 2.6 in which node w has two input queues. $\psi(\alpha) = (u, w)$ and $\psi(\beta) = (v, w)$ define two producer/consumer pairs. Queue α in Figure 2.6 has the same dataflow attributes as queue q connecting nodes u and v in Figure 2.4. Hence, as shown previously, $MinTokens(\alpha) = 4$ and $MaxUnderThr(\alpha) = 6$. Applying Theorem 2.3.1 to queue β yields $MinTokens(\beta) = 0$ and $MaxUnderThr(\beta) = 1$.

We now relax the restriction requiring $init(q) = 0$, and extend the analysis to include queues with initialized data.

Theorem 2.3.2. *Let $\psi(q) = (u, v)$ and queue q be initialized with $init(q) \geq 0$ tokens. After nodes v and u have executed at least once, the minimum number of tokens on queue q is at least $MinTokens(q)$ and the maximum number of tokens queue q can hold without being over threshold is $MaxUnderThr(q)$ where*

$$MinTokens(q) = f(q) + \left\lceil \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \qquad (2.4)$$

$$MaxUnderThr(q) =$$

$$\begin{cases} thr(q) - \gcd(prd(q), cns(q)) & \textit{if } \gcd(prd(q), cns(q)) \mid (thr(q) - f(q)) \\ f(q) + \left\lfloor \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \right\rfloor \cdot \gcd(prd(q), cns(q)) & \textit{otherwise} \end{cases} \tag{2.5}$$

and

$$f(q) = \begin{cases} init(q) - \left(\left\lfloor \frac{init(q) - thr(q)}{cns(q)} \right\rfloor + 1 \right) \cdot cns(q) & \textit{if } init(q) \geq thr(q) \\ init(q) & \textit{otherwise} \end{cases}$$

Proof: The proof proceeds with three steps. In step one, we prove that $f(q) < thr(q)$. Step two proves Equation (2.4) is a valid lower bound on the number of tokens queue q will contain after nodes u and v have executed at least once. Step three shows Equation (2.5) is an upper bound on the number of tokens queue q can contain without being over threshold.

Step 1: Prove $f(q) < thr(q)$. If $init(q) < thr(q)$, then $f(q) = init(q)$ and $f(q) < thr(q)$. If $init(q) \geq thr(q)$, then node v (which consumes $cns(q)$ tokens each time it executes) can execute at most k times before the number of tokens on queue q is less than $thr(q)$ where k is the least natural number such that $init(q) - (k \cdot cns(q)) < thr(q)$. This implies $k > \frac{init(q) - thr(q)}{cns(q)}$. The smallest natural number satisfying this inequality is $\left\lfloor \frac{init(q) - thr(q)}{cns(q)} \right\rfloor + 1$. Since each execution of node v consumes $cns(q)$ tokens from queue q, it immediately follows that the number of tokens consumed after k executions of node v is $\left(\left\lfloor \frac{init(q) - thr(q)}{cns(q)} \right\rfloor + 1 \right) \cdot cns(q)$. Thus, assuming node u does not produce any tokens (i.e., it does not execute), the number of tokens on queue q after node v executes k times is

$$init(q) - k \cdot cns(q) = init(q) - \left(\left\lfloor \frac{init(q) - thr(q)}{cns(q)} \right\rfloor + 1 \right) \cdot cns(q) < thr(q).$$

Hence $f(q) < thr(q)$.

Step 2: Prove Equation (2.4) is a valid lower bound on the number of tokens queue q will contain after nodes u and v have executed at least once. Let $length(q)$ be the number of tokens on queue q after node v has consumed

$$\left(\left\lfloor \frac{init(q) - thr(q)}{cns(q)} \right\rfloor + 1 \right) \cdot cns(q)$$

tokens from queue q.

Let k_u denote a number of executions of node u, and k_v denote a number of executions of node v. Observe that $length(q)$ is always of the form

$$length(q) = f(q) + k_u \cdot prd(q) - k_v \cdot cns(q),$$

and that after node v first executes, queue q will always contain at least $thr(q) - cns(q)$ tokens. Thus, to prove Equation (2.4), we must find the smallest value the expression $f(q) + k_u \cdot prd(q) - k_v \cdot cns(q)$ can be without being less than $thr(q) - cns(q)$.

Observe that for all integer values of k_u and k_v that satisfy the expression

$$f(q) + k_u \cdot prd(q) - k_v \cdot cns(q) \geq thr(q) - cns(q),$$

there exists integers a, b, and k such that

$$\begin{aligned}
&f(q) + k_u \cdot prd(q) - k_v \cdot cns(q) \\
&\qquad = f(q) + k_u \cdot (a \cdot \gcd(prd(q), cns(q))) - k_v \cdot (b \cdot \gcd(prd(q), cns(q))) \\
&\qquad = f(q) + (k_u \cdot a - k_v \cdot b) \cdot \gcd(prd(q), cns(q)) \\
&\qquad = f(q) + k \cdot \gcd(prd(q), cns(q)).
\end{aligned}$$

Thus, the lower bound on the length of queue q, is the smallest integer k that satisfies the expression

$$f(q) + k \cdot \gcd(prd(q), cns(q)) \geq thr(q) - cns(q). \tag{2.6}$$

The smallest integer k that satisfies Equation (2.6) is

$$k = \left\lceil \frac{thr(q) - cns(q)}{\gcd(prd(q), cns(q))} \right\rceil$$

Therefore, the minimum number of tokens that will ever be on queue q after node v has executed at least once is:

$$\begin{aligned}
MinTokens(q) &= f(q) + \left\lceil \frac{thr(q) - cns(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) \\
&= f(q) + \left(\left\lceil \frac{thr(q)}{\gcd(prd(q), cns(q))} \right\rceil - \frac{cns(q)}{\gcd(prd(q), cns(q))} \right) \cdot \gcd(prd(q), cns(q)) \\
&= f(q) + \left\lceil \frac{thr(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q)
\end{aligned}$$

which proves that Equation (2.4) is the lower bound on the length of queue q after nodes u and v have executed at least once.

Step 3: Prove Equation (2.5) is a valid upper bound on the number of tokens queue q can contain without being over threshold after nodes u and v have executed at least once. The maximum number of tokens queue q can hold without being over threshold is

$$f(q) + k \cdot \gcd(prd(q), cns(q))$$

where k is the largest integer such that $f(q) + k \cdot \gcd(prd(q), cns(q)) < thr(q)$. As before, $k = \left\lfloor \frac{thr(q)-f(q)}{\gcd(prd(q),cns(q))} \right\rfloor$ is the largest integer such that

$$f(q) + k \cdot \gcd(prd(q), cns(q)) \leq thr(q).$$

If $\gcd(prd(q), cns(q))$ does not divide $(thr(q) - f(q))$, then

$$\begin{aligned} f(q) + k' \cdot \gcd(prd(q), cns(q)) &= f(q) + \left\lfloor \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \right\rfloor \cdot \gcd(prd(q), cns(q)) \\ &< thr(q) \end{aligned}$$

and Equation (2.5) holds. If $\gcd(prd(q), cns(q)) \mid (thr(q) - f(q))$, then

$$\begin{aligned} f(q) + k' \cdot \gcd(prd(q), cns(q)) &= f(q) + \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \cdot \gcd(prd(q), cns(q)) \\ &= f(q) + thr(q) - f(q) = thr(q). \end{aligned}$$

In this case, the maximum number of tokens queue q can hold without being over threshold is

$$\begin{aligned} &f(q) + (k' - 1) \cdot \gcd(prd(q), cns(q)) \\ &\quad = f(q) + \left(\frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} - 1 \right) \cdot \gcd(prd(q), cns(q)) \\ &\quad = f(q) + \left(\frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \cdot \gcd(prd(q), cns(q)) \right) - \gcd(prd(q), cns(q)) \\ &\quad = f(q) + thr(q) - f(q) - \gcd(prd(q), cns(q)) \\ &\quad = thr(q) - \gcd(prd(q), cns(q)), \end{aligned}$$

which proves Equation (2.5) also holds when $\gcd(prd(q), cns(q)) \mid (thr(q) - f(q))$. $\square$

Note that, when $init(q) = 0$, Equation (2.4) reduces to Equation (2.1) and Equation (2.5) reduces to Equation (2.2). Unlike Theorem 2.3.1, however, Theorem 2.3.2 can be applied to any queue in a processing graph without restriction on the number of tokens with which the queue was initialized.

Consider the two node chain of Figure 2.4 once again (where $prd(q) = 4$, $thr(q) = 7$, and $cns(q) = 3$). This time, assume queue q is initialized with 7 tokens (thus $init(q) = 7$, and $thr(q) = 7$). Using Equation (2.4), the minimum number of tokens queue q contains after nodes u and v both execute at least once is

$$\begin{aligned}
MinTokens(q) &= f(q) + \left\lceil \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \\
&= init(q) - \left(\left\lfloor \frac{init(q) - thr(q)}{cns(q)} \right\rfloor + 1 \right) \cdot cns(q) \\
&\quad + \left\lceil \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \\
&= 7 - \left(\left\lfloor \frac{7-7}{3} \right\rfloor + 1 \right) \cdot 3 + \left\lceil \frac{7 - f(q)}{\gcd(4,3)} \right\rceil \cdot \gcd(4,3) - 3 \\
&= 4 + \left(\left\lceil \frac{7-4}{1} \right\rceil \cdot 1 \right) - 3 = 4.
\end{aligned}$$

Since $\gcd(prd(q), cns(q)) = \gcd(4,3) = 1$, the gcd of the produce and consume values divides $(thr(q) - f(q))$. Therefore, by Equation (2.5), the maximum number of tokens queue q can hold without being over threshold is

$$MaxUnderThr(q) = thr(q) - \gcd(prd(q), cns(q)) = 7 - 1 = 6.$$

Thus, when queue q is initialized with $init(q) = 7$ tokens, the functions $MinTokens(q)$ and $MaxUnderThr(q)$ evaluate to the same values as when the queue was not initialized with data. When $\gcd(prd(q), cns(q) = 1$, the amount of initialized data does not affect these functions:

$$\begin{aligned}
MinTokens(q) &= f(q) + \left\lceil \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \\
&= f(q) + \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \cdot \gcd(prd(q), cns(q)) - cns(q) \\
&= f(q) + thr(q) - f(q) - cns(q) \\
&= thr(q) - cns(q)
\end{aligned}$$

and $MinTokens(q) = thr(q) - 1$ since 1 always divides $thr(q) - f(q)$.

Now consider the chain of Figure 2.5 where $prd(q) = 8$, $thr(q) = 7$, and $cns(q) = 6$, and assume queue q is initialized with one token ($init(q) = 1$). In this case, $\gcd(prd(q), cns(q)) = \gcd(8, 6) = 2$ and $init(q) < thr(q)$. Thus

$$\begin{aligned}
MinTokens(q) &= f(q) + \left\lceil \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \\
&= init(q) + \left\lceil \frac{thr(q) - init(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \\
&= 1 + \left\lceil \frac{7-1}{\gcd(8,6)} \right\rceil \cdot \gcd(8,6) - 6 \\
&= 1 + \left(\left\lceil \frac{7-1}{2} \right\rceil \cdot 2 \right) - 6 = 1 + (3 \cdot 2) - 6 = 1.
\end{aligned}$$

Since $\gcd(prd(q), cns(q)) = 2$ divides $(thr(q) - f(q)) = 6$, the maximum number of tokens the queue can hold without being over threshold is

$$MaxUnderThr(q) = thr(q) - \gcd(prd(q), cns(q)) = 7 - 1 = 6.$$

Although $\gcd(prd(q), cns(q)) \neq 1$, it is still the case that the functions $MinTokens(q)$ and $MaxUnderThr(q)$ evaluate to the same values as when the queue was not initialized with data. This is because the queue was initialized with $thr(q) - cns(q)$ tokens. When the queue is initialized in this manner, the initialized data does not affect the functions $MinTokens(q)$ and $MaxUnderThr(q)$:

$$\begin{aligned}
MinTokens(q) &= f(q) + \left\lceil \frac{thr(q) - f(q)}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \\
&= thr(q) - cns(q) + \left\lceil \frac{thr(q) - (thr(q) - cns(q))}{\gcd(prd(q), cns(q))} \right\rceil \cdot \gcd(prd(q), cns(q)) - cns(q) \\
&= thr(q) - cns(q) + \frac{cns(q)}{\gcd(prd(q), cns(q))} \cdot \gcd(prd(q), cns(q)) - cns(q) \\
&= thr(q) - cns(q) + cns(q) - cns(q) \\
&= thr(q) - cns(q)
\end{aligned}$$

and $MaxUnderThr(q) = thr(q) - \gcd(prd(q), cns(q))$ since

$$thr(q) - f(q) = thr(q) - (thr(q) - cns(q)) = cns(q)$$

and $\gcd(prd(q), cns(q))$ always divides $cns(q)$.

Whenever $\gcd(\mathit{prd}(q), \mathit{cns}(q)) \mid \mathit{thr}(q) - f(q)$, Equation (2.4) reduces to

$$\mathit{MinTokens}(q) = \mathit{thr}(q) - \mathit{cns}(q)$$

and Equation (2.5) reduces to

$$\mathit{MaxUnderThr}(q) = \mathit{thr}(q) - \gcd(\mathit{prd}(q), \mathit{cns}(q)).$$

Theorem 2.3.2 provides upper and lower bounds for the number of tokens a queue joining two nodes can contain without being over threshold (after both nodes have executed at least once). The following theorem, computes the number of executions of node v as a function of the number of tokens produced by node u when queue q is the only queue joining the pair.

Theorem 2.3.3. *Let* $\mathit{length}(q) \geq \mathit{thr}(q)$, $\psi(q) = (u, v)$, *and* $\delta^-(v) = 1$. *At the current time, assuming node* u *does not execute, node* v *will execute* $\left\lfloor \frac{\mathit{length}(q) - \mathit{thr}(q)}{\mathit{cns}(q)} \right\rfloor + 1$ *times, consume*

$$\left(\left\lfloor \frac{\mathit{length}(q) - \mathit{thr}(q)}{\mathit{cns}(q)} \right\rfloor + 1 \right) \cdot \mathit{cns}(q)$$

tokens, and leave l *tokens on queue* q *where* $\mathit{MinTokens}(q) \leq l \leq \mathit{MaxUnderThr}(q)$.

Before proving the theorem, we demonstrate its application using the chains of Figures 2.4 and 2.5. First, consider queue q in Figure 2.4 on page 37 where $\mathit{prd}(q) = 4$, $\mathit{thr}(q) = 7$, and $\mathit{cns}(q) = 3$. Recall from the previous discussion that after 4 executions of node u, node v executed a total of 4 times. Therefore, assuming 4 executions of node u (with no intervening executions of node v), queue q would contain 16 tokens. Applying the equations of Theorem 2.3.3 with $\mathit{length}(q) = 16$ shows that node v would be able to execute four times (just as the example demonstrated):

$$\left\lfloor \frac{\mathit{length}(q) - \mathit{thr}(q)}{\mathit{cns}(q)} \right\rfloor + 1 = \left\lfloor \frac{16 - 7}{3} \right\rfloor + 1 = 4.$$

Following 4 executions of node u and the resulting 4 executions of node v, there would be

$$\begin{aligned} \mathit{length}(q) - \left(\left\lfloor \frac{\mathit{length}(q) - \mathit{thr}(q)}{\mathit{cns}(q)} \right\rfloor + 1 \right) \cdot \mathit{cns}(q) &= 16 - \left(\left\lfloor \frac{16 - 7}{3} \right\rfloor + 1 \right) \cdot 3 \\ &= 16 - (4 \cdot 3) = 4 \end{aligned}$$

tokens on queue q. (Recall from the application of Equation (2.1) to this graph that 4 is the minimum number of tokens queue q will contain after the first execution of node v.)

Assuming 24 tokens on queue q of Figure 2.5 where $prd(q) = 8$, $thr(q) = 7$, and $cns(q) = 6$, node v will execute

$$\left\lfloor \frac{length(q) - thr(q)}{cns(q)} \right\rfloor + 1 = \left\lfloor \frac{24 - 7}{6} \right\rfloor + 1 = 3$$

times and consume

$$\left(\left\lfloor \frac{length(q) - thr(q)}{cns(q)} \right\rfloor + 1 \right) \cdot cns(q) = \left(\left\lfloor \frac{24 - 7}{6} \right\rfloor + 1 \right) \cdot 6 = 3 \cdot 6 = 18$$

of the 24 tokens, leaving 6 tokens on the queue.

Proof of Theorem 2.3.3: [1] If there are $length(q) \geq thr(q)$ tokens on queue q where $\psi(q) = (u, v)$ and $\delta^-(v) = 1$ and node u does not execute, then the number of times node v will execute is the least natural number k such that

$$length(q) - (k \cdot cns(q)) < thr(q),$$

which implies $k > (length(q) - thr(q))/cns(q)$. The smallest natural number satisfying this inequality is $k = \left\lfloor \frac{length(q) - thr(q)}{cns(q)} \right\rfloor + 1$.

Since each execution of node v consumes $cns(q)$ tokens from queue q, it immediately follows that the number of tokens consumed is $\left(\left\lfloor \frac{length(q) - thr(q)}{cns(q)} \right\rfloor + 1 \right) \cdot cns(q)$. Finally, from Theorem 2.3.2, the minimum number of tokens on queue q after node v executes is $MinTokens(q)$, and the maximum number of tokens that can be on the queue without the queue being over threshold is $MaxUnderThr(q)$. Therefore, it follows that

$$MinTokens(q) \ \leq \ l \ \leq \ MaxUnderThr(q)$$

and the theorem holds. □

Given $length(q)$ tokens on queue q, it is also useful to know how many more executions of node u are required before queue q is over threshold. In this case, the consume amount does not matter; we only care about $thr(q)$, $prd(q)$, and the existing number of tokens on queue q, $length(q)$.

[1] We thank the anonymous reviewer of a previous paper [28] who suggested this proof.

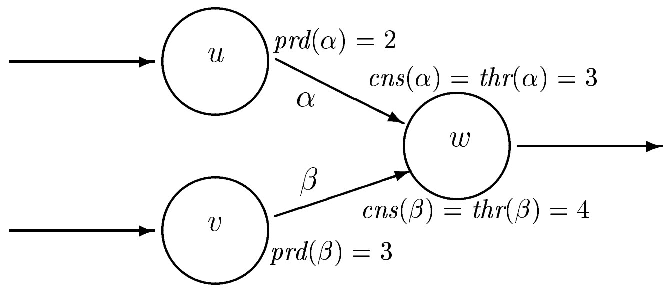


Figure 2.7: Another PGM node with two input queues.

Theorem 2.3.4. *Let there be* $length(q)$ *tokens on queue* q *and* $\psi(q) = (u, v)$*. Node* u *must execute*

$$\max\left(0, \left\lceil \frac{thr(q) - length(q)}{prd(q)} \right\rceil\right) \tag{2.7}$$

times before queue q *is over threshold.*

Proof: If there are $length(q)$ tokens on queue q and $length(q) \geq thr(q)$, then queue q is already over threshold and no more executions of node u are required. If $length(q) < thr(q)$, then $thr(q) - length(q)$ more tokens are required before q is over threshold. Since node u produces $prd(q)$ tokens every time it executes, it follows that u must execute $\left\lceil \frac{thr(q)-length(q)}{prd(q)} \right\rceil$ times before q is over threshold. In either case, the number of executions required of node u before queue q is over threshold is $\max\left(0, \left\lceil \frac{thr(q)-length(q)}{prd(q)} \right\rceil\right)$ and Equation (2.7) holds. □

To illustrate Theorem 2.3.4 consider the chain of Figure 2.4 on page 37 where $prd(q) = 4$, $thr(q) = 7$, and $cns(q) = 3$. Assuming 4 tokens on queue q in the chain of Figure 2.4, node u must execute

$$\max\left(0, \left\lceil \frac{thr(q) - length(q)}{prd(q)} \right\rceil\right) = \max\left(0, \left\lceil \frac{7-4}{4} \right\rceil\right) = \max\left(0, \left\lceil \frac{3}{4} \right\rceil\right) = 1$$

time before queue q is over threshold and node v is eligible for execution.

Next consider the graph of Figure 2.7 in which node w has two input queues. $\psi(\alpha) = (u, w)$ and $\psi(\beta) = (v, w)$ define two producer/consumer pairs. Theorem 2.3.4 can be applied to both input queues. Assuming no data on queue α, node u must execute

$$\max\left(0, \left\lceil \frac{thr(\alpha) - length(\alpha)}{prd(\alpha)} \right\rceil\right) = \max\left(0, \left\lceil \frac{3-0}{2} \right\rceil\right) = \max\left(0, \left\lceil \frac{3}{2} \right\rceil\right) = 2$$

times before queue α is over threshold. Since there are two input queues to node w, node w is not eligible for execution until both input queues are over threshold. Assuming $length(\beta) = 1$, node v must execute

$$\max\left(0, \left\lceil \frac{thr(\beta) - length(\beta)}{prd(\beta)} \right\rceil\right) = \max\left(0, \left\lceil \frac{4-1}{3} \right\rceil\right) = \max\left(0, \left\lceil \frac{3}{3} \right\rceil\right) = 1$$

time before queue β is over threshold. Thus, starting with $length(\alpha) = 0$ and $length(\beta) = 1$, node u must execute twice and node v must execute once before node w is eligible for execution.

Throughout this section we have informally derived node execution rates by simulating executions. Section 2.4 formally defines an execution rate and uses the theorems presented in this section to analytically compute node execution rates.

2.4 Node Execution Rates

Our goal in synthesizing a signal processing system from a processing graph is to execute nodes such that we can manage both imposed latency and memory usage. Consider the two-node chain of Figure 2.8. For the producer/consumer pair of nodes u and v, the number of tokens present on queue q at time t is a function of the queue's dataflow attributes and the number of executions of nodes u and v prior to time t. Node v can only execute when its input queue is over threshold, so the number of times it is able to execute in any interval of time is dependent on the number of times node u executes (and the dataflow attributes on queue q). In an implementation of the graph, the actual number of times that node v executes in any interval of time is dependent on the number of times node u executes and on the scheduling algorithm employed. If the scheduling algorithm delays executions of node v but continues to let node u execute, data will accumulate on queue q. To bound latency and memory usage in an implementation of the graph, we need to schedule the execution of nodes in a deterministic manner. For this, we appeal to real-time scheduling theory and execute the nodes according to a model of real-time execution. Finally, to select the proper model of real-time execution, we need to determine the natural execution pattern of nodes. We informally called the execution

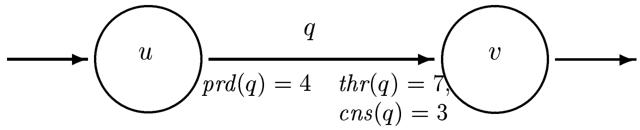
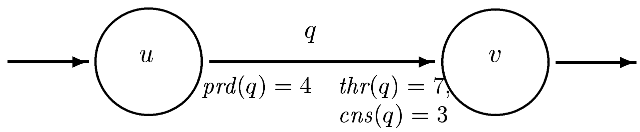


Figure 2.8: A two-node chain. The queue connecting nodes u and v has the dataflow attributes $prd(q) = 4$, $thr(q) = 7$, and $cns(q) = 3$.



Time	**Execution of**	$\boldsymbol{length}(q)$
0	u	4
y	u, v	5
$2y$	u, v	6
$3y$	u, v, v	4

Figure 2.9: A two-node chain and a snapshot sequence that shows the execution of the nodes under the strong synchrony hypothesis.

pattern a rate in Section 2.3. Here, we formally define an execution rate and show how to analytically derive the execution rates of nodes in a PGM graph. In Chapter 3, we map nodes to tasks in the RBE model and show how to schedule node executions so that we can manage latency and memory usage in an implementation of the graph.

To simplify the presentation of execution rates and to eliminate the role scheduling and node execution times play in the derivation of node execution rates, we extend our assumption that each node executes on its own processor and assume that the processors are each infinitely fast 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 [21]. The strong synchrony hypothesis states that the system instantly reacts to external stimuli. For example, the snapshot sequence in Figure 2.9 shows both nodes u and v executing at time y. The system reacts instantaneously to the arrival of data on the input queue to node u and both nodes u and v execute at the same instant. At time $3y$, one execution of node u and two executions of node v occur at the same instant. Node execution rates are defined as follows.

Definition 2.4.1. An *execution rate* is a pair of non-negative integers (x, y).

Definition 2.4.2. Execution rates (x_1, y_1) and (x_2, y_2) are *equal* if and only if $x_1 = x_2$ and $y_1 = y_2$.

Definition 2.4.3. 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 $k > 0$.

Notice that the interval is closed at the beginning and open at the end. Thus, if node u in Figure 2.9 continues to execute once every y time units, it has a valid execution rate of $R_u = (1, y)$ (starting at time 0). It executes exactly once in the interval $[0, y)$ since the execution at time y is counted in the interval $[y, 2y)$. While the periodic execution of node u satisfies the definition of a valid execution rate, the execution of node u does not need to be strictly periodic for it to have a valid execution rate of $R_u = (1, y)$. For example, if node u executed at times

$$0,\ 1.5y,\ 2y,\ 3.9y,\ 4y,\ 5y,\ 6y,\ \ldots,\ ky,\ \ldots$$

it still has a valid execution rate of $R_u = (1, y)$ starting at time 0 since it executes exactly once in each time interval $[0 + (k - 1)y,\ 0 + ky)$ for all $k > 0$.

If the execution of node u is periodic, however, the execution of node u is "well-defined" in that it executes at time ky for all $k \geq 0$. While the rate specification $R_u = (1, y)$ is a valid execution rate for node u, it does not describe the restricted execution pattern exhibited by node u.

Definition 2.4.4. An execution rate specification for node v, $R_v = (x, y)$, is *well-defined* if there exists a time t_v such that node v executes exactly x times in time intervals $[t,\ t+y)$ for all $t \geq t_v$.

Corollary 2.4.1. *A well-defined rate specification $R_v = (x, y)$ for node v is also a valid rate specification for node v.*

Proof: If $R_v = (x, y)$ is a well-defined rate specification for node v, then by Definition 2.4.4, there exists a time t_v such that node v executes exactly x times in time intervals $[t,\ t + y)$ for all $t \geq t_v$. Thus, for any $t \geq t_v$ node v executes exactly x times in time intervals $[t + (k - 1)y,\ t + ky)$ for all $k > 0$, and $R_v = (x, y)$ is a valid rate specification for node v. □

If $R_u = (1, y)$ is a valid execution rate for node u in Figure 2.9, then $R_u = (2, 2y)$ is also a valid execution rate since node u will execute twice in each time interval $[0 + (k -$

$1)2y,\ 0 + k2y)$ for all $k > 0$. In fact, as shown by Corollary 2.4.2, there are an infinite number of valid execution rates for node u.

Corollary 2.4.2. *If $R_v = (x, y)$ is a valid rate specification for node v, then for all positive integers m, $m \cdot R_v = (m \cdot x, m \cdot y)$ is also a valid rate specification for node v.*

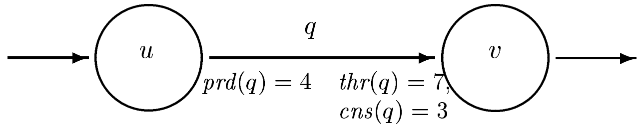
Proof: If $R_v = (x, y)$ is a valid rate specification for node v, then by Definition 2.4.3, there exists a time t such that node v executes exactly x times in time intervals $[t + (k-1)y,\ t + ky)$ for all $k > 0$. Thus in each time interval $[t + m(k-1)y,\ t + mky)$ for all $k > 0$, node v will execute exactly mx times, and $m \cdot R_v = (m \cdot x, m \cdot y)$ is also a valid rate specification for node v. □

Although there exists an infinite number of valid execution rates for a node, not every execution rate is valid. For example, let the execution rate $R_u = (1, y)$ of node u in Figure 2.9 be valid. By looking at the executions of node v in the snapshot sequence, it would appear that node v executes with a rate of $R_v = (4, 4y)$. Even though node v does execute 4 times in the interval $[0, 4y)$, the rate specification $R_v = (4, 4y)$ is not valid because this is the only interval of length $4y$ in which node v executes exactly 4 times. Node v actually executes at a rate of $R_v = (4, 3y)$ starting at time y. To see this, we need to simulate more executions of nodes u and v. Consider the extended snapshot sequence in Figure 2.10. This snapshot sequence shows that node v executes 4 times in the interval $[y, 4y)$, 4 times in the interval $[4y, 7y)$, and 4 times in the interval $[7y, 10y)$.

Sections 2.4.1 and 2.4.2 present the analytical derivation of valid execution rates for nodes in acyclic and cyclic graphs executed under the strong synchrony hypothesis. Under the strong synchrony hypothesis, we assume nodes execute on an infinitely fast machine, but we do not assume an infinite amount of memory. Thus, for a producer/consumer pair of nodes u and v with valid rate specifications $R_u = (x_u, y_u)$ and $R_v = (x_v, y_v)$, node v must consume the same number of tokens produced by u during the interval y_v. Otherwise, data will continue to accumulate on their joining queue until no more memory is available for buffering. At that point, either data will be lost or one of the nodes will violate their rate specification. In Chapter 3, we show how to implement a PGM graph so that nodes execute with these rates.

2.4.1 Rates in Acyclic Graphs

The derivation of node execution rates is done first for nodes with single input queues (Section 2.4.1.1) and then for nodes with multiple input queues (Section 2.4.1.2).



Time	Execution of	*length*(q)
0	u	4
y	u, v	5
$2y$	u, v	6
$3y$	u, v, v	4
$4y$	u, v	5
$5y$	u, v	6
$6y$	u, v, v	4
$7y$	u, v	5
$8y$	u, v	6
$9y$	u, v, v	4
$10y$	u, v	5

Figure 2.10: An extended snapshot sequence showing the execution of nodes u and v under the strong synchrony hypothesis. The execution rate of node u is $R_u = (1, y)$, and the execution rate of node v is $R_v = (4, 3y)$.

2.4.1.1 Deriving Rates for Nodes with Single Input Queues

We derived the execution rate of node v in Figure 2.10 by simulating executions of nodes u and v and "guessing" a valid execution rate. Alternatively, Theorem 2.4.3 can be used to analytically compute the execution rate of node v using the execution rate of node u and the dataflow attributes of queue q.

Theorem 2.4.3. *Let $u \rightsquigarrow v$ be a PGM chain with $\psi(q) = (u, v)$, and let $R_u = (x_u, y_u)$ be a valid execution rate for node u. Under the strong synchrony hypothesis, the execution rate $R_v = (x_v, y_v)$, where*

$$x_v = \frac{prd(q)}{\gcd(prd(q) \cdot x_u, cns(q))} \cdot x_u \tag{2.8}$$

$$\text{and} \quad y_v = \frac{cns(q)}{\gcd(prd(q) \cdot x_u, cns(q))} \cdot y_u, \tag{2.9}$$

is a valid execution rate for node v.

Proof: By Definition 2.4.3, because R_u is valid, there exists a time t_u such that node u executes exactly x_u times in each interval $[t_u + (k-1)y_u, t_u + ky_u)$ where $k > 0$. Let interval j be the first interval $[t_u + (j-1)y_u, t_u + jy_u)$ in which node v executes, and let $t_v = t_u + jy_u$.

In the remainder of the proof, we show that $R_v = (x_v, y_v)$ is a valid rate specification by showing that node v executes exactly x_v times in time intervals

$$[t_v + (k-1)y_v, t_v + ky_v)$$

for all $k > 0$ where x_v and y_v are as defined by Equations (2.8) and (2.9). Under the strong synchrony hypothesis, node v executes instantaneously whenever its input queue is over threshold. Let $\mathit{length}(q) = n$ at time t_v. Thus, by Theorem 2.3.2, n is bounded such that

$$\mathit{thr}(q) - \mathit{cns}(q) \leq \mathit{MinTokens}(q) \leq \;n\; \leq \mathit{MaxUnderThr}(q) < \mathit{thr}(q).$$

By Definition 2.4.3, node u executes exactly x_u times in intervals

$$[t_u + (k-1)y_u, t_u + ky_u)$$

for all $k > 0$. Thus, by Corollary 2.4.2 and because $t_v = t_u + jy_u$ and y_v is a multiple of y_u, node u executes $\frac{y_v}{y_u} \cdot x_u$ times in every time interval $[t_v + (k-1)y_v, t_v + ky_v)$ for all $k > 0$. Since node u produces $\mathit{prd}(q)$ tokens each time it executes, it enqueues a total of

$$\begin{aligned} \mathit{prd}(q) \cdot \frac{y_v}{y_u} \cdot x_u &= \mathit{prd}(q) \cdot \frac{\frac{\mathit{cns}(q) \cdot y_u}{\gcd(\mathit{prd}(q) \cdot x_u, \mathit{cns}(q))}}{y_u} \cdot x_u \\ &= \mathit{prd}(q) \cdot \frac{\mathit{cns}(q)}{\gcd(\mathit{prd}(q) \cdot x_u, \mathit{cns}(q))} \cdot x_u \end{aligned}$$

tokens on queue q in an interval of length y_v. Since each execution of node v consumes $\mathit{cns}(q)$ tokens, x_v executions of node v in an interval of length y_v will consume $(x_v \cdot \mathit{cns}(q))$ tokens. Thus, if queue q contains n tokens at the beginning of the interval, it will contain

$$\begin{aligned} &n + \left(\mathit{prd}(q) \cdot \frac{\mathit{cns}(q)}{\gcd(\mathit{prd}(q) \cdot x_u, \mathit{cns}(q))} \cdot x_u\right) - (x_v \cdot \mathit{cns}(q)) \\ &= n + \left(\mathit{prd}(q) \cdot \frac{x_u \cdot \mathit{cns}(q)}{\gcd(\mathit{prd}(q) \cdot x_u, \mathit{cns}(q))}\right) - \left(\frac{x_u \cdot \mathit{prd}(q)}{\gcd(\mathit{prd}(q) \cdot x_u, \mathit{cns}(q))} \cdot \mathit{cns}(q)\right) \\ &= n + \left(\frac{x_u \cdot \mathit{cns}(q) \cdot \mathit{prd}(q)}{\gcd(\mathit{prd}(q) \cdot x_u, \mathit{cns}(q))}\right) - \left(\frac{x_u \cdot \mathit{cns}(q) \cdot \mathit{prd}(q)}{\gcd(\mathit{prd}(q) \cdot x_u, \mathit{cns}(q))}\right) \\ &= n \end{aligned}$$

tokens at the end of the interval. Furthermore, no more than x_v executions could have occurred since the x_v^{th} execution leaves exactly $n < thr(q)$ tokens on q. Any fewer executions would have left $n \geq thr(q)$ tokens on q, and another execution of node v would have occurred. Therefore, node v executes exactly x_v times in time intervals $[t_v + (k-1)y_v, t_v + ky_v)$ for all $k > 0$ where x_v and y_v are as defined by Equations (2.8) and (2.9). □

As required, the proof of Theorem 2.4.3 only proves that Equations (2.8) and (2.9) can by used to compute a valid rate specification for the consumer node v, and there are infinitely many other valid execution rate specifications for node v, as shown by Corollary 2.4.2.

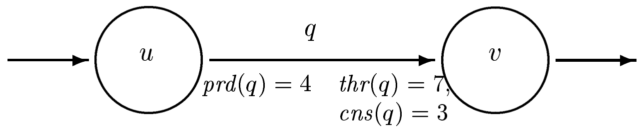
We now illustrate the derivation of rates using Theorem 2.4.3 by considering the graph of Figure 2.8 where $prd(q) = 4$, $thr(q) = 7$, and $cns(q) = 3$. It was assumed node u had a periodic execution rate of $R_u = (1, y)$. We "guessed" an execution rate of $R_v = (4, 3y)$ for node v using the extended snapshot sequence presented in Figure 2.10. Using Theorem 2.4.3, the execution rate of node v is computed analytically as follows:

$$\begin{aligned} R_v = (x_v, y_v) &= \left(\frac{prd(q) \cdot x_u}{\gcd(prd(q) \cdot x_u, cns(q))}, \frac{cns(q) \cdot y_u}{\gcd(prd(q) \cdot x_u, cns(q))} \right) \\ &= \left(\frac{4 \cdot 1}{\gcd(4 \cdot 1, 3)}, \frac{3 \cdot y}{\gcd(4 \cdot 1, 3)} \right) \\ &= \left(\frac{4}{\gcd(4, 3)}, \frac{3 \cdot y}{\gcd(4, 3)} \right) = \left(\frac{4}{1}, \frac{3y}{1} \right) = (4, 3y) \end{aligned}$$

Now assume an execution rate of $R_u = (3, 16)$ for node u. The corresponding snapshot sequence (and a reproduction of the graph) is shown in Figure 2.11. The execution rate of node v is derived using Theorem 2.4.3 as follows:

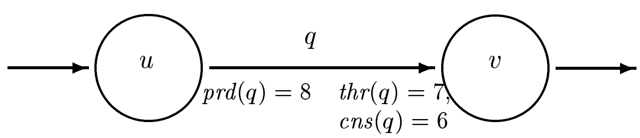
$$\begin{aligned} R_v = (x_v, y_v) &= \left(\frac{prd(q) \cdot x_u}{\gcd(prd(q) \cdot x_u, cns(q))}, \frac{cns(q) \cdot y_u}{\gcd(prd(q) \cdot x_u, cns(q))} \right) \\ &= \left(\frac{4 \cdot 3}{\gcd(4 \cdot 3, 3)}, \frac{3 \cdot 16}{\gcd(4 \cdot 3, 3)} \right) \\ &= \left(\frac{12}{\gcd(12, 3)}, \frac{48}{\gcd(12, 3)} \right) \\ &= \left(\frac{12}{3}, \frac{48}{3} \right) = (4, 16) \end{aligned}$$

Finally, consider an example in which the greatest common divisor of the produce and consume values is not one, such as in the chain in Figure 2.12. Let the execution



Time	**Execution of**	***length***(q)
0	u	4
1	u, v	5
15	u, v	6
16	u, v, v	4
17	u, v	5
20	u, v	6
28	u, v, v	4
33	u, v	5
40	u, v	6
48	u, v, v	4
55	u, v	5

Figure 2.11: Another extended snapshot sequence showing the execution of nodes u and v under the strong synchrony hypothesis. This time the execution rate of node u is $R_u = (3, 16)$, and the execution rate of node v is $R_v = (4, 16)$.



Time	**Execution of**	***length***(q)
0	u, v	2
4	u, v	4
15	u, v	6
20	u, v, v	2
30	u, v	4
34	u, v	6
42	u, v, v	2

Figure 2.12: A snapshot sequence showing the execution of nodes u and v under the strong synchrony hypothesis where $\gcd(prd(q), cns(q)) > 1$. The dataflow attributes $prd(q)$, $thr(q)$, and $cns(q)$ are 8, 7, and 6 respectively. If $R_u = (2, 15)$, then $R_v = (8, 45)$. The execution rates for nodes u and v are valid after time 0.

rate of node u in Figure 2.12 be $R_u = (2, 15)$. By Theorem 2.4.3, the execution rate of node v is

$$\begin{aligned} R_v = (x_v, y_v) &= \left(\frac{prd(q) \cdot x_u}{\gcd(prd(q) \cdot x_u,\, cns(q))}, \frac{cns(q) \cdot y_u}{\gcd(prd(q) \cdot x_u,\, cns(q))} \right) \\ &= \left(\frac{8 \cdot 2}{\gcd(8 \cdot 2, 6)}, \frac{6 \cdot 15}{\gcd(8 \cdot 2, 6)} \right) \\ &= \left(\frac{16}{\gcd(16, 6)}, \frac{90}{\gcd(16, 6)} \right) = \left(\frac{16}{2}, \frac{90}{2} \right) = (8, 45) \end{aligned}$$

We now consider the case where the specification of node u is well-defined. In this case, the execution rate of node v is also well-defined when it is computed using Equations (2.8) and (2.9).

Theorem 2.4.4. *Let $u \rightsquigarrow v$ be a PGM chain with $\psi(q) = (u, v)$, and let $R_u = (x_u, y_u)$ be a well-defined execution rate for node u. Let $R_v = (x_v, y_v)$ be computed using Equations* (2.8) *and* (2.9). *Under the strong synchrony hypothesis, the execution rate $R_v = (x_v, y_v)$ is a well-defined execution rate for node v.*

Proof: By Definition 2.4.4, because R_u is well-defined, there exists a time t_u such that node u executes exactly x_u times in each interval $[t,\ t + y_u)$ where $t \geq t_u$. If $MinTokens(q) \leq init(q) \leq MaxUnderThr(q)$ then let $t_v = t_u$. If $init(q) < MinTokens(q)$ or $init(q) > MaxUnderThr(q)$, then let time $t_v{}'$ be the first time node v executes after time t_u, and let $t_v = t_v{}' + 1$.

In the remainder of the proof, we show that $R_v = (x_v, y_v)$ is a well-defined rate specification by showing that node v executes exactly x_v times in time intervals

$$[t,\ t + y_v)$$

for all $t \geq t_v$ where x_v and y_v are as defined by Equations (2.8) and (2.9). Let $length(q) = n$ at time t_v. Observe that n is bounded such that

$$thr(q) - cns(q) \leq MinTokens(q) \leq\ n\ \leq MaxUnderThr(q) < thr(q)$$

either by Theorem 2.3.2, which bounds the size of n after nodes u and v have each executed at least once, or because $n = init(q)$, which was so bounded.

By Corollary 2.4.2 and the definition of a well-defined rate, node u executes $\frac{y_v}{y_u} \cdot x_u$ times in every interval $[t,\ t + y_v)\,, \forall t \geq t_v$. Since node u produces $prd(q)$ tokens each time it executes, it enqueues a total of

$$
\begin{aligned}
prd(q) \cdot \frac{y_v}{y_u} \cdot x_u &= prd(q) \cdot \frac{\frac{cns(q) \cdot y_u}{\gcd(prd(q) \cdot x_u, cns(q))}}{y_u} \cdot x_u \\
&= prd(q) \cdot \frac{cns(q)}{\gcd(prd(q) \cdot x_u, cns(q))} \cdot x_u
\end{aligned}
$$

tokens on queue q in an interval of length y_v. Moreover, since each execution of node v consumes $cns(q)$ tokens, x_v executions of node v in an interval of length y_v will consume $x_v \cdot cns(q)$ tokens. Thus, if queue q contains n tokens at the beginning of the interval, it will contain

$$
\begin{aligned}
n &+ \left(prd(q) \cdot \frac{cns(q)}{\gcd(prd(q) \cdot x_u, cns(q))} \cdot x_u\right) - (x_v \cdot cns(q)) \\
&= n + \left(prd(q) \cdot \frac{x_u \cdot cns(q)}{\gcd(prd(q) \cdot x_u, cns(q))}\right) - \left(\frac{x_u \cdot prd(q)}{\gcd(prd(q) \cdot x_u, cns(q))} \cdot cns(q)\right) \\
&\qquad = n + \left(\frac{x_u \cdot cns(q) \cdot prd(q)}{\gcd(prd(q) \cdot x_u, cns(q))}\right) - \left(\frac{x_u \cdot cns(q) \cdot prd(q)}{\gcd(prd(q) \cdot x_u, cns(q))}\right) \\
&\qquad = n
\end{aligned}
$$

tokens at the end of the interval. Furthermore, no more than x_v executions could have occurred since the x_v^{th} execution of node v leaves $n < thr(q)$ tokens on q. Any fewer executions would have left $n \geq thr(q)$ tokens on q, and another execution of node v would have occurred. Therefore, exactly x_v executions take place in an interval of length y_v, and $R_v = (x_v, y_v)$ is a well-defined specification for v. □

Theorem 2.4.3 can be used to compute the execution rate of every node in a PGM chain, and many non-trivial signal processing applications (such as the processing graph for the SAR application studied in Section 5.2) are described using a chain. When the node has multiple input queues, however, a valid execution rate cannot always be computed using Equations (2.8) and (2.9). The next section illustrates this point and then extends our results to compute the execution rate of any node in an acyclic PGM graph.

2.4.1.2 Deriving Rates for Nodes with Multiple Input Queues

While Equations (2.8) and (2.9) do not directly compute the execution rate of a node with multiple input queues, it is instructive to apply these equations to each input queue as though they did. Consider the graph in Figure 2.13. Node w is a consumer of data produced by both nodes u and v; $\psi(\alpha) = (u, w)$ and $\psi(\beta) = (v, w)$ define two producer/consumer pairs. The notation $R_{w\leftarrow u} = (x_{w\leftarrow u}, y_{w\leftarrow u})$ denotes the rate at which node w executes if nodes u and w were a producer/consumer pair in a chain. Thus with $R_u = (3, 16)$, $R_{w\leftarrow u}$ is computed using Equations (2.8) and (2.9) as follows:

$$\begin{aligned}
R_{w\leftarrow u} = (x_{w\leftarrow u}, y_{w\leftarrow u}) &= \left(\frac{prd(\alpha) \cdot x_u}{\gcd(prd(\alpha) \cdot x_u, cns(\alpha))}, \frac{cns(\alpha) \cdot y_u}{\gcd(prd(\alpha) \cdot x_u, cns(\alpha))} \right) \\
&= \left(\frac{4 \cdot 3}{\gcd(4 \cdot 3, 3)}, \frac{3 \cdot 16}{\gcd(4 \cdot 3, 3)} \right) = (4, 16)
\end{aligned}$$

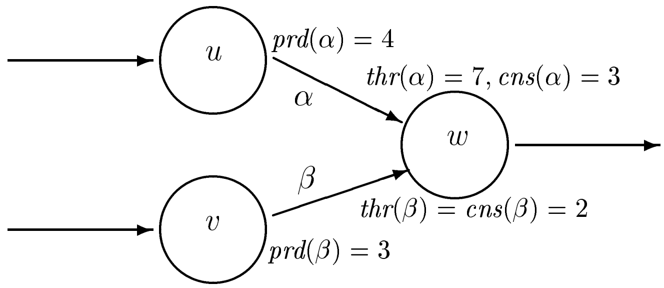
With $R_v = (2, 12)$, $R_{w\leftarrow v}$ is computed using Equations (2.8) and (2.9) as follows:

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

Since node w can only execute when *both* queues α and β are over threshold, neither $R_{w\leftarrow u}$ nor $R_{w\leftarrow v}$ satisfies the definition of a valid execution rate for node w because node w will not execute exactly 4 times in any interval of length 16 or exactly 3 times in any interval of length 12 — see the snapshot sequence in Figure 2.13. Although in general $R_{w\leftarrow u} \neq R_{w\leftarrow v}$, it must be the case that the equality $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$ holds if a valid rate specification for node w exists. Otherwise, either node u or node v will produce tokens faster than node w can consume them and data will be lost.

Lemma 2.4.5. *Let $G = (V, E, \psi)$ be a PGM graph and let w be a node in V that has at least two input queues. Let $\mathcal{V}$ denote the set of nodes for which there exists a queue q in E and a node u in V such that $\psi(q) = (u, w)$. Let $R_u = (x_u, y_u)$ be a valid execution rate specification for each node u in $\mathcal{V}$. If a valid execution rate specification for node w exists, then $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$ for all nodes u and v in $\mathcal{V}$.*

Proof: (By contradiction.) For nodes u and v in $\mathcal{V}$, let q_u connect node u to node w ($\psi(q_u) = (u, w)$), and q_v connect node v to node w ($\psi(q_v) = (v, w)$). Suppose there exists a valid rate specification $R_w = (x_w, y_w)$ for node w, but $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} \neq \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$. If $R_w = (x_w, y_w)$ is a valid rate specification, then, by Corollary 2.4.2, $y_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot R_w$ is also a valid rate specification for node w. If $y_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot R_w$ is a valid rate specification, then there exists



Time	**Execution of**	***length***(α)	***length***(β)
0	u	4	0
1	u	8	0
2	v, w	5	1
10	v	5	4
14	v	5	7
15	u, w	6	5
16	u, w, w	4	1
17	u	8	1
20	u	12	1
22	v, w, w	6	0
26	v	6	3
28	u, w	7	1
33	u	11	1
34	v, w, w	5	0
38	v	5	3
40	u, w	6	1
46	v	6	4
48	u, w, w	4	0

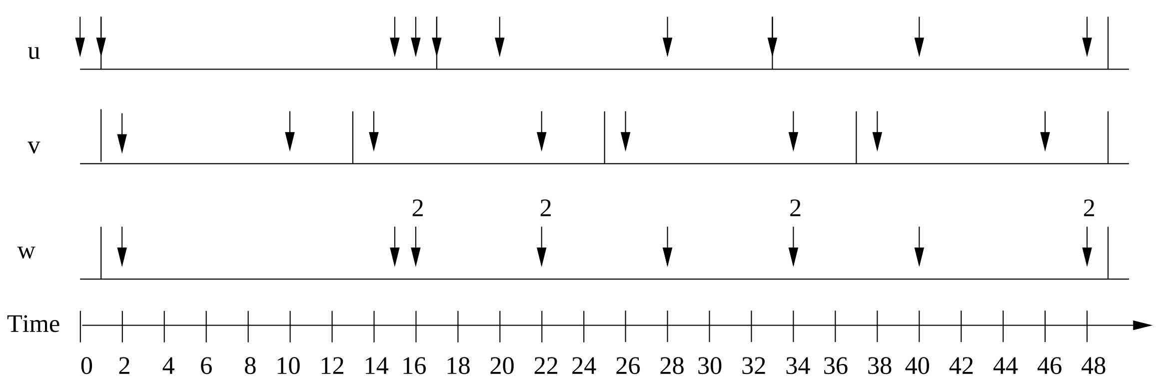


Figure 2.13: A three-node graph, snapshot sequence, and time-line execution showing the execution of nodes under the strong synchrony hypothesis. If $R_u = (3, 16)$ and $R_v = (2, 12)$ are valid after time 0, then $R_w = (12, 48)$. Each down arrow represents an execution of the node. Multiple executions of a node at the same instant are represented by a number above the down arrow.

a time t_w such that node w executes $y_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot x_w$ times in intervals

$$[t_w + (k-1) \cdot y_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot y_w, t_w + k \cdot y_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot y_w), \quad \forall k > 0.$$

In each of these intervals, however, node u will only produce enough tokens for node w to execute

$$\frac{y_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot y_w}{y_{w\leftarrow u}} \cdot x_{w\leftarrow u} = x_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot x_w$$

times since, by Theorem 2.4.4, node u produces enough tokens for node w to execute $x_{w\leftarrow u}$ times in an interval of length $y_{w\leftarrow u}$. Node v will only produce enough tokens for node w to execute

$$\frac{y_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot y_w}{y_{w\leftarrow v}} \cdot x_{w\leftarrow v} = x_{w\leftarrow v} \cdot y_{w\leftarrow u} \cdot x_w$$

times since, by Theorem 2.4.4, node v produces enough tokens for node w to execute $x_{w\leftarrow v}$ times in an interval of length $y_{w\leftarrow v}$. Since

$$\begin{aligned} \frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} \neq \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}} &\implies x_{w\leftarrow u} \cdot y_{w\leftarrow v} \neq x_{w\leftarrow v} \cdot y_{w\leftarrow u} \\ &\implies x_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot x_w \neq x_{w\leftarrow v} \cdot y_{w\leftarrow u} \cdot x_w \end{aligned}$$

and since node w can only execute when all of its input queues are over threshold, node w can execute at most

$$\min(x_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot x_w, x_{w\leftarrow v} \cdot x_{w\leftarrow u} \cdot x_w)$$

times. Thus, if rate $R_w = (x_w, y_w)$ is valid, it must be the case that

$$y_{w\leftarrow u} \cdot y_{w\leftarrow v} = \min(x_{w\leftarrow u} \cdot y_{w\leftarrow v}, x_{w\leftarrow v} \cdot x_{w\leftarrow u}).$$

Without loss of generality, assume

$$x_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot x_w < x_{w\leftarrow v} \cdot y_{w\leftarrow u} \cdot x_w.$$

In an interval of length $y_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot y_w$,

$$(x_{w\leftarrow v} \cdot y_{w\leftarrow u} \cdot x_w - x_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot x_w) \cdot cns(q_v)$$

more tokens will be produced on queue q_v (the queue connecting node v to node w) than node w consumes in the same interval. If node w continues to execute with a rate of $y_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot R_w$, data will eventually be lost since we assume a finite amount of memory exists. To prevent the loss of data, a requirement for valid graph execution, either node v must execute less often in an interval of length $y_{w\leftarrow u} \cdot y_{w\leftarrow v} \cdot y_w$, or node w must execute more often. Both cases violate a rate specification, a contradiction. Thus, if a valid rate specification exists for node w, given valid rate specifications for nodes u and v, it must be the case that $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$. □

Even if $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$ for producer nodes u, and v and consumer node w, it may not be possible to define a valid rate specification for node w. For example, consider the simple three-node graph in Figure 2.14. Node u executes exactly once in intervals $[0 + (k-1)y_u, 0 + ky_u)$ for all $k > 0$, and node v executes exactly once in intervals $[1 + (k-1)y_u, 1 + ky_u)$ for all $k > 0$. It would seem that there exists a time t_w such that node w will execute exactly once in intervals $[t_w + (k-1)y_u, t_w + ky_u)$ for all $k > 0$. The execution time line in Figure 2.14, shows an execution using this rate specification with $t_w = 1$. Notice that in the interval $[5, 8)$ node w executes twice, and in the interval $[8, 11)$ it does not execute at all. No matter how long we make the interval nor what value we choose for t_w, there can always be an execution pattern for node w that will violate its rate specification if it executes as soon as both of its input queues are over threshold. The intuitive reason for this is that nodes u and v are executing "out of phase." Their execution intervals will never begin at the same time.

If nodes u and v have valid execution rates beginning at times t_u and t_v respectively, then each execution interval of node u starts at time

$$t_u + k_u y_u$$

for all $k_u \geq 0$, and each execution interval of node v starts at time

$$t_v + k_v y_v$$

for all $k_v \geq 0$. Thus, a valid execution rate for node w exists if and only if there exists non-negative integers k_u and k_v such that

$$t_u + k_u y_u = t_v + k_v y_v,$$

which represents a time in which execution intervals of nodes u and v begin at the same instant. The following theorem provides a necessary and sufficient condition for detecting whether there exists a time in which the two intervals will begin at the same time.

Lemma 2.4.6. *Let $G = (V, E, \psi)$ be a PGM graph. Let u and v in V have valid rate specifications $R_u = (x_u, y_u)$ and $R_v = (x_v, y_v)$ beginning at times t_u and t_v respectively. There exists non-negative integers k_u and k_v such that*

$$t_u + k_u y_u = t_v + k_v y_v. \tag{2.10}$$

if and only if $\gcd(y_u, y_v) \mid (t_u - t_v)$.

Proof: Observe that

$$t_u + k_u y_u = t_v + k_v y_v \implies k_v y_v - k_u y_u = (t_u - t_v). \tag{2.11}$$

The latter form of the equation is called a *linear diophantine equation in two variables* where k_u and k_v are the two integer variables [56].

($\Rightarrow$) We prove that if $\gcd(y_u, y_v) \mid (t_u - t_v)$, then there exists non-negative integers k_u and k_v such that $t_u + k_u y_u = t_v + k_v y_v$. From Theorem 2.14 of Rosen's *Elementary Number Theory and its Applications* [56], if $\gcd(y_u, y_v)$ divides $(t_u - t_v)$, then there are infinitely many integral solutions to Equation (2.11) and if integers ${k_u}'$, ${k_v}'$ are one particular solution to Equation (2.11), then all solutions are given by

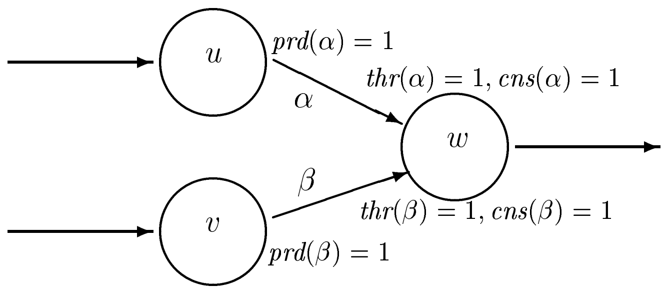
$$k_u = {k_u}' - \frac{y_u}{\gcd(y_u, y_v)} \cdot n, \qquad k_v = {k_v}' - \frac{y_v}{\gcd(y_u, y_v)} \cdot n,$$

where n is an integer. Let integers ${k_u}'$, ${k_v}'$ be one particular solution to Equation (2.11) derived using the Euclidean algorithm [56]. If either ${k_u}'$ or ${k_v}'$ is less than zero, a non-negative integral solution is obtained with

$$k_u = {k_u}' - y_u \cdot \min\left(\left\lfloor \frac{{k_u}'}{y_u} \right\rfloor, \left\lfloor \frac{{k_v}'}{y_v} \right\rfloor\right)$$

and

$$k_v = {k_v}' - y_v \cdot \min\left(\left\lfloor \frac{{k_u}'}{y_u} \right\rfloor, \left\lfloor \frac{{k_v}'}{y_v} \right\rfloor\right)$$



Time	**Execution of**	***length***(α)	***length***(β)
1	v	0	1
2	u, w	0	0
3	u	1	0
6	u, v, w	1	0
7	v, w	0	0
10	u	1	0
12	v, w	0	0

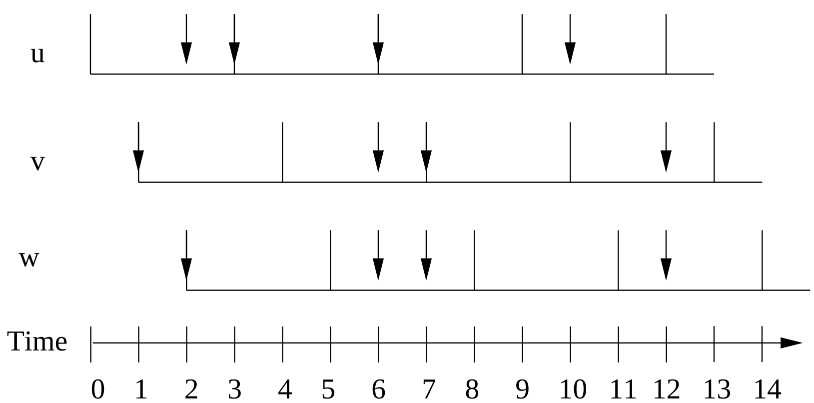


Figure 2.14: A three-node graph, snapshot sequence, and time-line execution showing that no valid rate specification exists for consumer node w under the strong synchrony hypothesis. No valid rate specification exists for node w even though the dataflow attributes on both input queues are identical and $R_u = R_v = (1, 3)$ are valid rate specifications. This is because the rate for node u is valid starting at time 0 and the rate for node v is valid starting at time 1.

since

$$\begin{aligned} k_u &= {k_u}' - y_u \cdot \min\left(\left\lfloor \frac{{k_u}'}{y_u} \right\rfloor, \left\lfloor \frac{{k_v}'}{y_v} \right\rfloor\right) \\ &= {k_u}' - y_u \cdot \min\left(\left\lfloor \frac{{k_u}'}{y_u} \right\rfloor, \left\lfloor \frac{{k_v}'}{y_v} \right\rfloor\right) \cdot \frac{\gcd(y_u, y_v)}{\gcd(y_u, y_v)} \\ &= {k_u}' - \frac{y_u}{\gcd(y_u, y_v)} \cdot n \end{aligned}$$

and

$$\begin{aligned} k_v &= {k_v}' - y_v \cdot \min\left(\left\lfloor \frac{{k_u}'}{y_u} \right\rfloor, \left\lfloor \frac{{k_v}'}{y_v} \right\rfloor\right) \\ &= {k_v}' - y_v \cdot \min\left(\left\lfloor \frac{{k_u}'}{y_u} \right\rfloor, \left\lfloor \frac{{k_v}'}{y_v} \right\rfloor\right) \cdot \frac{\gcd(y_u, y_v)}{\gcd(y_u, y_v)} \\ &= {k_v}' - \frac{y_v}{\gcd(y_u, y_v)} \cdot n \end{aligned}$$

where $n = \min\left(\left\lfloor \frac{{k_u}'}{y_u} \right\rfloor, \left\lfloor \frac{{k_v}'}{y_v} \right\rfloor\right) \cdot \gcd(y_u, y_v)$. Thus, if $\gcd(y_u, y_v) \mid (t_u - t_v)$, then there exists non-negative integers k_u and k_v such that $t_u + k_u y_u = t_v + k_v y_v$.

($\Leftarrow$) We now prove that if there exists non-negative integers k_u and k_v such that

$$t_u + k_u y_u = t_v + k_v y_v$$

then $\gcd(y_u, y_v) \mid (t_u - t_v)$. Let k_u and k_v be non-negative integers such that $t_u + k_u y_u = t_v + k_v y_v$. Then

$$t_u + k_u y_u = t_v + k_v y_v \implies k_v y_v - k_u y_u = (t_u - t_v)$$

and there exists integers a and b such that

$$\gcd(y_u, y_v) \cdot (k_v a - k_u b) = (t_u - t_v)$$

Thus $\gcd(y_u, y_v) \mid (t_u - t_v)$ if there exists non-negative integers k_u and k_v such that

$$t_u + k_u y_u = t_v + k_v y_v.$$

$\square$

The following theorem provides a necessary and sufficient condition for detecting whether a valid execution rate specification exists for a node w with multiple input queues.

Theorem 2.4.7. *Let $G = (V, E, \psi)$ be an acyclic PGM graph and let w be a node in V that has at least two input queues. Let $\mathcal{V}$ denote the set of nodes for which there exists a queue q in E and a node v in V such that $\psi(q) = (v, w)$. A valid execution rate specification for node w exists if and only if, for all nodes u and v in $\mathcal{V}$, $R_u = (x_u, y_u)$ and $R_v = (x_v, y_v)$ are valid execution rates beginning at times t_u and t_v respectively, $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$, and $\gcd(y_u, y_v) \mid (t_u - t_v)$.*

Proof: ($\Rightarrow$) We prove that if, for all nodes u and v in $\mathcal{V}$, $R_u = (x_u, y_u)$ and $R_v = (x_v, y_v)$ are valid execution rates beginning at times t_u and t_v respectively, $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$, and $\gcd(y_u, y_v) \mid (t_u - t_v)$, then there exists a valid execution rate specification for node w. The existence of a valid rate execution specification is proven by deriving a rate specification for node w and proving it to be valid.

Let integers k_u, k_v be one particular solution to

$$k_v y_v - k_u y_u = t_u - t_v$$

derived using the Euclidean algorithm [56]. If either k_u or k_v is less than zero, let

$$k_u = k_u - y_u \cdot \min\left(\left\lfloor \frac{k_u{}'}{y_u} \right\rfloor, \left\lfloor \frac{k_v{}'}{y_v} \right\rfloor\right)$$

and

$$k_v = k_v - y_v \cdot \min\left(\left\lfloor \frac{k_u{}'}{y_u} \right\rfloor, \left\lfloor \frac{k_v{}'}{y_v} \right\rfloor\right).$$

Let $t_w = t_u + k_u y_u$. Since nodes u and v have valid rate specifications, both nodes begin an execution interval at time t_w. Let the rate specification of node w be $R_w = (x_w, y_w)$ where

$$y_w = \operatorname{lcm}\left\{ \frac{cns(q) \cdot y_v}{\gcd(prd(q) \cdot x_v, cns(q))} \mid q \in E \wedge v \in \mathcal{V} \ : \psi(q) = (v, w) \right\}, \tag{2.12}$$

and

$$x_w = y_w \cdot \frac{prd(q) \cdot x_u}{cns(q) \cdot y_u}. \tag{2.13}$$

We now prove that $R_w = (x_w, y_w)$ is a valid execution rate starting at time t_w. The proof is constructed in two steps. In step one we prove that, for nodes u and w such that there exists queue $\alpha \in E$ with $\psi(\alpha) = (u, w)$, $R_w = (x_w, y_w)$ computed using Equations (2.12) and (2.13) is a valid execution rate specification for node w with respect to node u. In step two we show that the rate derived using node u and queue α is the same rate that would be derived for any $v \in \mathcal{V}$.

Step 1: Prove that, for nodes u and w such that there exists queue $\alpha \in E$ with $\psi(\alpha) = (u, w)$, $R_w = (x_w, y_w)$ computed using Equations (2.12) and (2.13) is a valid execution rate specification for node w with respect to node u. If node w was in a chain such that nodes u and w were a producer/consumer pair, then, using Theorem 2.4.3, the execution rate of node w with respect to node u starting at t_w is $R_{w\leftarrow u} = (x_{w\leftarrow u}, y_{w\leftarrow u})$ where

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

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

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

Note that m_u is an integer since $y_{w\leftarrow u} = \frac{cns(q)}{\gcd(prd(q)\cdot x_u, cns(q))} \cdot y_u$ is a factor in the expression

$$\operatorname{lcm}\left\{\frac{cns(q)}{\gcd(prd(q)\cdot x_v,\, cns(q))} \cdot y_v \mid q \in E \wedge v \in \mathcal{V}\ : \psi(q) = (v, w)\right\}.$$

By Theorem 2.4.3 and Corollary 2.4.2 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 $y_w = m_u \cdot y_{w\leftarrow u}$ to the form of Equation (2.12):

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

Expression $x_w = m_u \cdot x_{w\leftarrow u}$ is reduced to the form of Equation (2.13) as follows:

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

Thus, if node u was the only producer for node w, node w would execute exactly x_w times in time intervals $[t_w + (k-1)y_w, t_w + ky_w)$ for all $k > 0$.

Step 2: Show that the rate derived using node u and queue α is the same rate that would be derived for any $u \in \mathcal{V}$. Let nodes u and v be in $\mathcal{V}$ such that $\psi(\alpha) = (u, w)$ and $\psi(\beta) = (v, w)$. The execution rate of node w with respect to node u, $R_{w\leftarrow u} = (x_{w\leftarrow u}, y_{w\leftarrow u})$ is derived using Theorem 2.4.3 as

$$
\begin{aligned}
x_{w\leftarrow u} &= \frac{prd(\alpha)}{\gcd(prd(\alpha)\cdot x_u,\, cns(\alpha))} \cdot \; x_u \\
y_{w\leftarrow u} &= \frac{cns(\alpha)}{\gcd(prd(\alpha)\cdot x_u,\, cns(\alpha))} \cdot y_u
\end{aligned}
$$

and the execution rate of node w with respect to node v, $R_{w\leftarrow v} = (x_{w\leftarrow v}, y_{w\leftarrow v})$ is derived using Theorem 2.4.3 as

$$
\begin{aligned}
x_{w\leftarrow v} &= \frac{prd(\beta)}{\gcd(prd(\beta)\cdot x_v,\, cns(\beta))} \cdot \; x_v \\
y_{w\leftarrow v} &= \frac{cns(\beta)}{\gcd(prd(\beta)\cdot x_v,\, cns(\beta))} \cdot y_v.
\end{aligned}
$$

Therefore,

$$\begin{aligned}\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} &= \frac{\frac{prd(\alpha)}{\gcd(prd(\alpha)\cdot x_u, cns(\alpha))}\cdot\ x_u}{\frac{cns(\alpha)}{\gcd(prd(\alpha)\cdot x_u, cns(\alpha))}\cdot y_u} \\ &= \frac{prd(\alpha)\cdot\ x_u}{cns(\alpha)\cdot y_u}\end{aligned}$$

and

$$\begin{aligned}\frac{x_{w\leftarrow v}}{y_{w\leftarrow v}} &= \frac{\frac{prd(\beta)}{\gcd(prd(\beta)\cdot x_v, cns(\beta))}\cdot\ x_v}{\frac{cns(\beta)}{\gcd(prd(\beta)\cdot x_v, cns(\beta))}\cdot y_v} \\ &= \frac{prd(\beta)\cdot\ x_v}{cns(\beta)\cdot y_v}.\end{aligned}$$

Since $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$ for all u and v in $\mathcal{V}$,

$$x_w = \frac{prd(\alpha)\cdot\ x_u}{cns(\alpha)\cdot y_u} = \frac{prd(\beta)\cdot\ x_v}{cns(\beta)\cdot y_v}$$

for all u and v in $\mathcal{V}$.

Thus, since node w executes exactly x_w times in intervals $[t_w + (k-1)y_w, t_w + ky_w)$ for all $k > 0$ where x_w and y_w are computed using Equations (2.12) and (2.13), a valid execution rate specification exists if, for all nodes u and v in $\mathcal{V}$, $R_u = (x_u, y_u)$ and $R_v = (x_v, y_v)$ are valid execution rates beginning at times t_u and t_v respectively, $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$, and $\gcd(y_u, y_v) \mid (t_u - t_v)$.

($\Leftarrow$) We now prove that if, there exists a valid rate specification for node w, then, for all nodes u and v in $\mathcal{V}$, *(i)* $R_u = (x_u, y_u)$ and $R_v = (x_v, y_v)$ are valid execution rates beginning at times t_u and t_v respectively, *(ii)* $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$, and *(iii)* $\gcd(y_u, y_v) \mid (t_u - t_v)$.

(i) Assume there exists a valid rate specification for node w. Since node w can only execute when all of its input queues are over threshold and there exists a valid execution rate specification $R_w = (x_w, y_w)$ for node w, there must exist a positive integer x_v, for all nodes v in $\mathcal{V}$, such that node v executes exactly x_v times in intervals $[t_w + (k-1)y_v, t_w + ky_v)$ for all $k > 0$. Thus there exists a valid rate specification for all nodes v in $\mathcal{V}$ where $t_v \leq t_w$.

(ii) By Lemma 2.4.5, if a valid execution rate specification for node w exists, then $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$ for all nodes u and v in $\mathcal{V}$.

(iii) If there exists a valid execution rate specification for node w with producers u

and v in $\mathcal{V}$, then there exists a non-negative t_w such that node w executes exactly x_w times in intervals $[t_w + (k-1)y_w, t_w + ky_w)$ for all $k > 0$. Therefore, there must exist non-negative integers k_u and k_v such that

$$t_w = t_u + k_u y_u = t_v + k_v y_v$$

since execution intervals for node u begin at time $t_u + k_u y_u$, for all $k_u \geq 0$, and execution intervals for node v begin at time $t_v + k_v y_v$, for all $k_v \geq 0$. Thus, by Lemma 2.4.6, if there exists non-negative integers k_u and k_v such that

$$t_u + k_u y_u = t_v + k_v y_v,$$

then $\gcd(y_u, y_v) \mid (t_u - t_v)$.

Thus, a valid execution rate specification for node w exists if and only if, for all nodes u and v in $\mathcal{V}$, $R_u = (x_u, y_u)$ and $R_v = (x_v, y_v)$ are valid execution rates beginning at times t_u and t_v respectively, $\frac{x_{w \leftarrow u}}{y_{w \leftarrow u}} = \frac{x_{w \leftarrow v}}{y_{w \leftarrow v}}$, and $\gcd(y_u, y_v) \mid (t_u - t_v)$. $\square$

Theorem 2.4.8. *Let $G = (V, E, \psi)$ be an acyclic PGM graph with node w in V that has at least one input queue. Under the strong synchrony hypothesis, if there exists a valid rate specification for node w, then the execution rate specification $R_w = (x_w, y_w)$, where where x_w and y_w are computed using Equations* (2.12) *and* (2.13)*, is a valid rate specification for node w.*

Proof: Let $\mathcal{V}$ denote the set of nodes for which there exists a queue q in E and a node v in V such that $\psi(q) = (v, w)$. By Theorem 2.4.7, if a valid execution rate specification for node w exists, then, for all nodes u and v in $\mathcal{V}$, $R_u = (x_u, y_u)$ and $R_v = (x_v, y_v)$ are valid execution rates beginning at times t_u and t_v respectively, $\frac{x_{w \leftarrow u}}{y_{w \leftarrow u}} = \frac{x_{w \leftarrow v}}{y_{w \leftarrow v}}$, and $\gcd(y_u, y_v) \mid (t_u - t_v)$. The proof of Theorem 2.4.7 proved the existence of a valid rate specification, given these preconditions, by showing there exists a time t_w such that node w will execute exactly x_w times in intervals

$$[t_w + (k-1)y_u, t_w + ky_u)$$

for all $k > 0$ where x_w and y_w are computed using Equations (2.12) and (2.13).

Therefore, if a valid execution rate specification exists for node w, the rate specification $R_w = (x_w, y_w)$ computed using Equations (2.12) and (2.13) is a valid execution rate specification. $\square$

If a node has only one input queue, either Theorem 2.4.8 or Theorem 2.4.3 can be used to derive its execution rate since Equation (2.12) reduces to Equation (2.9):

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

and Equation (2.13) reduces to Equation (2.8):

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

We now return to the problem of finding the execution rate of node w in Figure 2.13 on page 62. Let the execution rates of nodes u and v in Figure 2.13 be $R_u = (3, 16)$ and $R_v = (2, 12)$, and let $t_u = t_v = 1$. By Theorem 2.4.8,

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

and

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

since $\frac{x_{w \leftarrow u}}{y_{w \leftarrow u}} = \frac{x_{w \leftarrow v}}{y_{w \leftarrow v}}$ and $\gcd(16, 12) \mid 0$. Thus $R_w = (x_w, y_w) = (12, 48)$ and node w in Figure 2.13 will execute 12 times in each interval $[1 + (k-1)48, 1 + 48k)$ for all $k > 0$.

As was the case with single input queues, if all of the immediate predecessors to node w have well-defined execution rate specifications, then there exists a well-defined rate specification for node w.

Theorem 2.4.9. *Let $G = (V, E, \psi)$ be an acyclic PGM graph and let w be a node in V that has at least one input queue. Let $\mathcal{V}$ denote the set of nodes for which there exists a queue q in E and a node v in V such that $\psi(q) = (v, w)$. For all nodes u and v in $\mathcal{V}$, let $R_u = (x_u, y_u)$ and $R_v = (x_v, y_v)$ be well-defined execution rates beginning at times t_u and t_v respectively and $\frac{x_{w \leftarrow u}}{y_{w \leftarrow u}} = \frac{x_{w \leftarrow v}}{y_{w \leftarrow v}}$. Under the strong synchrony hypothesis, the execution rate $R_w = (x_w, y_w)$ for node w is well-defined if*

$$y_w = \operatorname{lcm}\left\{\frac{cns(q) \cdot y_v}{\gcd(prd(q) \cdot x_v, cns(q))} \mid q \in E \wedge v \in \mathcal{V} \ : \psi(q) = (v, w)\right\}, \tag{2.14}$$

$$x_w = y_w \cdot \frac{prd(q) \cdot x_u}{cns(q) \cdot y_u}. \tag{2.15}$$

Proof: The proof is constructed in three steps. First we select a time t_w that denotes the beginning of a time interval for which the execution rate of node w, $R_w = (x_w, y_w)$, will be well-defined. In step two we prove that, for nodes u and w such that there exists a queue $\alpha \in E$ with $\psi(\alpha) = (u, w)$, $R_w = (x_w, y_w)$, computed using Equations (2.15) and (2.14), is a well-defined execution rate specification for node w with respect to node u. In step three we show that the rate derived using node u and queue α is the same rate that would be derived for any $v \in \mathcal{V}$.

Step 1: Select a time t_w that denotes the beginning of a time interval for which the execution rate of node w, $R_w = (x_w, y_w)$, will be well-defined. By Definition 2.4.3, node $u \in \mathcal{V}$ executes exactly x_u times in intervals $[t,\ t + y_u)$ for all $t \geq t_u$. If, for all $q \in E$ such that $\psi(q) = (u, w)$ and $u \in \mathcal{V}$, $MinTokens(q) \leq init(q) \leq MaxUnderThr(q)$ then let $t_w = \max_{u \in \mathcal{V}}(t_u)$. If there exists a $q \in E$ such that $\psi(q) = (u, w)$ and $u \in \mathcal{V}$ with $init(q) < MinTokens(q)$ or $init(q) > MaxUnderThr(q)$, then let time s_w denote the time when node w first executes after $\max_{u \in \mathcal{V}}(t_u)$, and let $t_w = s_w + 1$. (Note that in either case $s_w + 1 \geq t_w$. This fact will be used in the proof of Theorem 2.4.11 in Section 2.4.2.)

Step 2: Prove $R_w = (x_w, y_w)$ is a well-defined rate specification by showing that node w executes exactly x_w times in time intervals $[t,\ t + y_w)$ for all $t \geq t_w$ where x_w and y_w are computed using Equations (2.15) and (2.14). Let $length(q) = n$ at time t_w, for all $q \in E$ such that $\psi(q) = (u, w)$ and $u \in \mathcal{V}$, and observe that n is bounded such that

$$thr(q) - cns(q) \leq MinTokens(q) \leq \ n \ \leq MaxUnderThr(q) < thr(q)$$

either by Theorem 2.3.2, which bounds the size of n after nodes u and v have each executed at least once, or because $n = init(q)$, which was so bounded. Thus, if node w was in a chain such that nodes u and w were a producer/consumer pair joined by queue

α, then, using Theorem 2.4.3, the execution rate of node w with respect to node u is $R_{w\leftarrow u} = (x_{w\leftarrow u}, y_{w\leftarrow u})$ where

$$\begin{aligned} x_{w\leftarrow u} &= \frac{prd(\alpha)}{\gcd(prd(\alpha)\cdot x_u, cns(\alpha))} \cdot\ x_u \quad \text{and} \\ y_{w\leftarrow u} &= \frac{cns(\alpha)}{\gcd(prd(\alpha)\cdot x_u, cns(\alpha))} \cdot y_u. \end{aligned} \tag{2.16}$$

Moreover, if queue α was the only input queue to node w, the rate specification is well-defined for all $t \geq t_w$ by Theorem 2.4.4. Let $R_w = (x_w, y_w) = (m_u \cdot x_{w\leftarrow u}, m_u \cdot y_{w\leftarrow u})$ where

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

Note that m_u is an integer since $y_{w\leftarrow u} = \frac{cns(q)}{\gcd(prd(q)\cdot x_u, cns(q))} \cdot y_u$ is a factor in the expression

$$\operatorname{lcm}\left\{\frac{cns(q)}{\gcd(prd(q)\cdot x_v, cns(q))} \cdot y_v \mid q \in E \wedge v \in \mathcal{V}\ : \psi(q) = (v, w)\right\}.$$

By Theorem 2.4.4 and Corollary 2.4.2 if $R_{w\leftarrow u}$ is a well-defined rate specification for node w with respect to node u, then R_w is as well. We now reduce $y_w = m_u \cdot y_{w\leftarrow u}$ to the form of Equation (2.14):

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

and $x_w = m_u \cdot x_{w\leftarrow u}$ to the form of Equation (2.15):

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

Step 3: Show that the rate derived using node u and queue α is the same rate that would be derived for any $u \in \mathcal{V}$. Let nodes u and v be in $\mathcal{V}$ such that $\psi(\alpha) = (u, w)$ and $\psi(\beta) = (v, w)$. The execution rate of node w with respect to node u, $R_{w\leftarrow u} = (x_{w\leftarrow u}, y_{w\leftarrow u})$ is derived using Theorem 2.4.3 as

$$\begin{aligned}
x_{w\leftarrow u} &= \frac{prd(\alpha)}{\gcd(prd(\alpha)\cdot x_u,\, cns(\alpha))} \cdot\ x_u \\
y_{w\leftarrow u} &= \frac{cns(\alpha)}{\gcd(prd(\alpha)\cdot x_u,\, cns(\alpha))} \cdot y_u
\end{aligned}$$

and the execution rate of node w with respect to node v, $R_{w\leftarrow v} = (x_{w\leftarrow v}, y_{w\leftarrow v})$ is derived using Theorem 2.4.3 as

$$\begin{aligned}
x_{w\leftarrow v} &= \frac{prd(\beta)}{\gcd(prd(\beta)\cdot x_v,\, cns(\beta))} \cdot\ x_v \\
y_{w\leftarrow v} &= \frac{cns(\beta)}{\gcd(prd(\beta)\cdot x_v,\, cns(\beta))} \cdot y_v.
\end{aligned}$$

Therefore,

$$\begin{aligned}
\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} &= \frac{\frac{prd(\alpha)}{\gcd(prd(\alpha)\cdot x_u, cns(\alpha))} \cdot\ x_u}{\frac{cns(\alpha)}{\gcd(prd(\alpha)\cdot x_u, cns(\alpha))} \cdot y_u} \\
&= \frac{prd(\alpha)\cdot\ x_u}{cns(\alpha)\cdot y_u}
\end{aligned}$$

and

$$\frac{x_{w\leftarrow v}}{y_{w\leftarrow v}} = \frac{\frac{prd(\beta)}{\gcd(prd(\beta)\cdot x_v, cns(\beta))} \cdot \ x_v}{\frac{cns(\beta)}{\gcd(prd(\beta)\cdot x_v, cns(\beta))} \cdot y_v}$$
$$= \frac{prd(\beta) \cdot \ x_v}{cns(\beta) \cdot y_v}.$$

Since $\frac{x_{w\leftarrow u}}{y_{w\leftarrow u}} = \frac{x_{w\leftarrow v}}{y_{w\leftarrow v}}$ for all u and v in $\mathcal{V}$,

$$x_w = \frac{prd(\alpha) \cdot \ x_u}{cns(\alpha) \cdot y_u} = \frac{prd(\beta) \cdot \ x_v}{cns(\beta) \cdot y_v}$$

for all u and v in $\mathcal{V}$.

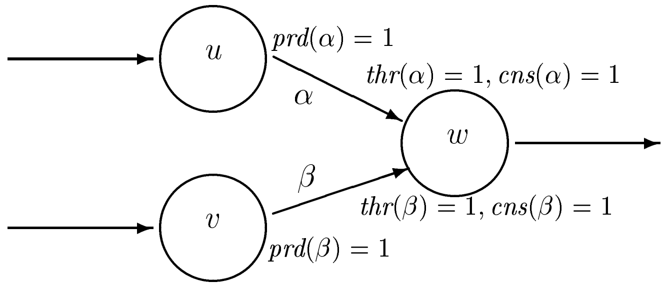
Therefore the rate specification $R_w = (x_w, y_w)$ derived with Equations (2.14) and (2.15) using node u and queue α is the same for all nodes $v \in V$ and queues $q \in E$ such that $\psi(q) = (v, w)$.

Thus, node w will execute exactly x_w times in intervals $[t, t + y_w)$ for all $t \geq t_w$ where x_w and y_w are computed using Equations (2.15) and (2.14), and the execution rate specification $R_w = (x_w, y_w)$ is well-defined. □

When graph source nodes have well-defined rates, the derivation of rates for the other nodes in the graph proceeds as one would expect. We simply compute the execution rate of a consumer node using the input queues and producer nodes. Moreover, the computed execution rate will be well-defined. For example, consider a case where the rate specifications for nodes u and v are well-defined, as in Figure 2.15. Node u executes exactly once in intervals $[t_u, t_u + y_u)$ for all $t_u \geq 0$, and node v executes exactly once in intervals $[t_v, t_v + y_v)$ for all $t_v \geq 1$. Thus, by Theorem 2.4.9, the execution rate of node w, $R_w = (x_w, y_w)$, is well-defined and can be computed using Equations (2.14) and (2.15). Doing so yields a well-defined rate specification for node w of $R_w = (1, 3)$, just as we would expect in this case. Node w executes exactly once every 3 time units after time 0.

Thus, when graph source nodes have well-defined execution rates, every node in an acyclic graph will also have well-defined execution rates, and Equations (2.14) and (2.15) can be used to compute the well-defined execution rates. If the graph source nodes have valid rate specifications, but not well-defined rate specifications, we may not be able to compute a valid rate specification for nodes in an acyclic graph. We can, however, detect when a valid rate specification cannot exist.

Most signal processing system synthesized from PGM graphs have periodic source



Time	Execution of	$\boldsymbol{length}(\alpha)$	$\boldsymbol{length}(\beta)$
1	v	0	1
2	u, w	0	0
4	v	0	1
5	u, w	0	0
7	v	0	1
8	u, w	0	0
10	v	0	1
11	u, w	0	0
13	v	0	1
14	u, w	0	0

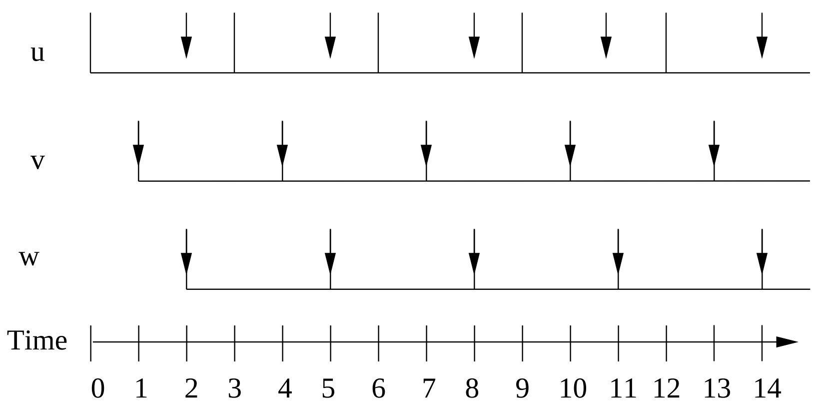


Figure 2.15: A three-node graph, snapshot sequence, and time-line execution showing well-defined execution rates under the strong synchrony hypothesis. A well-defined rate specification exists for node w when the rate specifications $R_u = R_v = (1, 3)$ are well-defined, even when the rate for node u is valid starting at time 0 and the rate for node v is valid starting at time 1.

devices. In the few cases that we have encountered where the source was not periodic, its execution rate was well-defined. Thus, for the rest of this dissertation, we only consider PGM graphs in which the source nodes have well-defined execution rates.

2.4.2 Rates in Cyclic Graphs

We now address the issue of deriving execution rate specifications in cyclic graphs. If a node is not in a cycle, its rate specification is computed using the same equations that were used when the graph was acyclic. When the node is in a cycle, its rate specification is computed using a slight variation of Equations (2.14) and (2.15).

Consider the simple cycle of nodes in Figure 2.16 — the queues in the cycle are labeled β, δ, and ζ. Let the well-defined execution rate of graph source node u by $R_u = (x_u, y_u)$. If the graph was acyclic, the execution rate of node w would be computed using the execution rates of nodes u and v. The execution rate of node v, however, is dependent on the execution rate of node r, which is dependent on the execution rate of node w. The only way we can compute a well-defined rate specification for node w (or any of the other nodes in the cycle) is to break the circular dependencies. We do this by requiring each *back edge* to be initialized with enough data to ensure that it is always over threshold. A *back edge* is an edge e that joins node v to an ancestor u when the graph is topologically sorted. Thus, for the graph in Figure 2.16, queue ζ is a back edge. If queue ζ is always over threshold, the execution rate of node v cannot affect the execution rate of node w.

Lemma 2.4.10. *Let $G = (V, E, \psi)$ be a PGM graph and let w be a node in V that has two input queues. Let queues q_u and q_v be queues in E and nodes u and v be nodes in V such that $\psi(q_u) = (u, w)$ and $\psi(q_v) = (v, w)$. If the execution rate specification $R_u = (x_u, y_u)$ be well-defined and queue q_v is always be over threshold, then the execution rate $R_w = (x_w, y_w)$ is well-defined where x_w and y_w are computed using Equations* (2.8) *and* (2.9).

Proof: If $\mathit{MinTokens}(q) \leq \mathit{init}(q) \leq \mathit{MaxUnderThr}(q)$ then let $t_w = t_u$. If $\mathit{init}(q) < \mathit{MinTokens}(q)$ or $\mathit{init}(q) > \mathit{MaxUnderThr}(q)$, then let time $t_w{}'$ be the first time node v executes after time t_u, and let $t_w = t_w{}' + 1$. If queue q_u were the only input queue to node w, then, by Theorem 2.4.4, Equations (2.8) and (2.8) can be used to compute a well-defined execution rate specification for node w. Thus, if queue q_v is always over threshold, node w will execute exactly x_w times in intervals $[t, t + y_w)$ for all $t \geq t_w$, and $R_w = (x_w, y_w)$ is a well-defined execution rate specification. $\square$

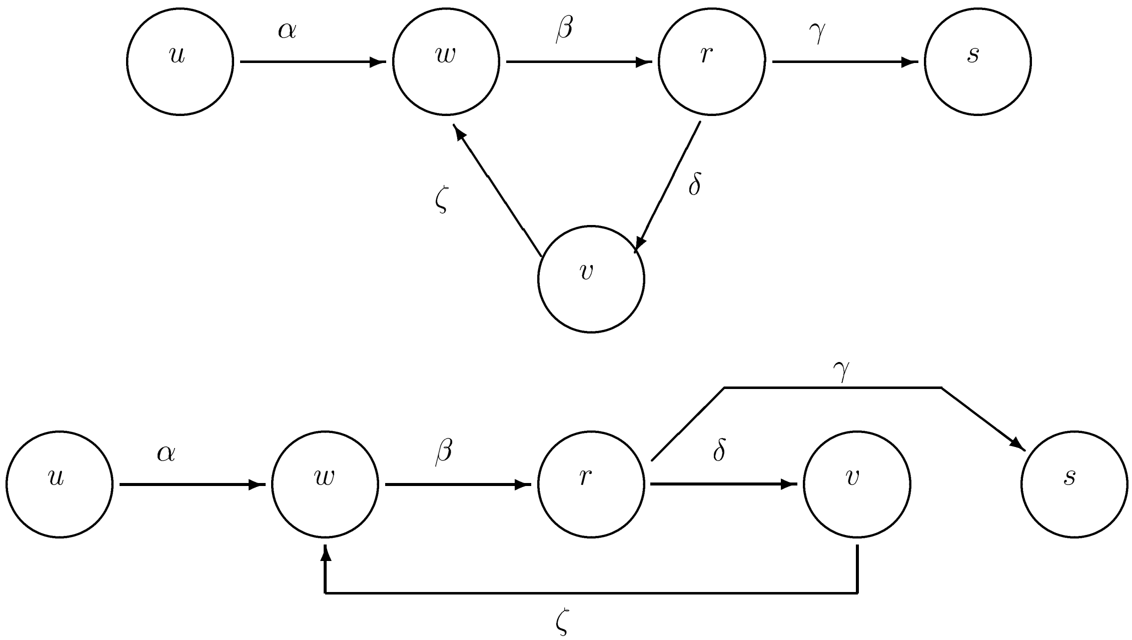


Figure 2.16: A graph with a simple three-node cycle and its topological sort. The queues in the cycle are labeled β, δ, and ζ.

Thus, if queue ζ is always over threshold, the execution rate of node v cannot affect the execution rate of node w. However, the execution rate of node v does determine how much initial data is needed for queue ζ to ensure that it is always over threshold.

Theorem 2.4.11. *Let $G = (V, E, \psi)$ be a PGM graph. Let queue q be a queue in E and node v and w be nodes in V such that $\psi(q) = (v, w)$ and queue q is a back edge in a cycle. Let the execution rate specifications $R_v = (x_v, y_v)$ and $R_w = (x_w, y_w)$ be well-defined beginning at times t_v and t_w respectively where t_v and t_w are derived using the technique outlined in the proof of Theorem 2.4.9. If nodes v and w first execute at times s_v and s_w respectively and*

$$init(q) = \left\lceil \frac{s_v - s_w + y_v}{y_w} \right\rceil \cdot x_w \cdot cns(q) + thr(q) \tag{2.17}$$

then queue q will always be over threshold.

Proof: When t_v and t_w are derived using the technique outlined in the proof of Theorem 2.4.9, $s_v + 1 \geq t_v$ and $s_w + 1 \geq t_w$. Since the execution rate of node v is derived using the execution rate of node w, there exists a positive integer k such that $y_v = k \cdot y_w$ and after time s_w node w will execute x_w times in every interval of length y_w. In an interval of length y_v, node w will execute $k \cdot x_w$ times. If the number of initialized tokens on queue q is computed using Equation (4.14), queue q will remain over threshold until time $(s_v + y_v)$ even if node v produces no tokens. However, after time s_v, node v will always produce enough data for node w to execute $k \cdot x_v$ times in any interval of length y_v. Thus, if the number of initialized tokens on queue q is computed using Equation (4.14), queue q will always remain over threshold. $\square$

Thus, we can compute a rate specification for a node w using Equations (2.14) and (2.15) if the computation is performed without including back edges. Moreover, the rate specification computed using Equations (2.14) and (2.15) will be well-defined if the number of initial tokens on each back edge is greater than or equal to the number of tokens computed using Equation (2.17) and the other preconditions of Theorem 2.4.9 are met.

2.5 Summary

In this chapter we derived the point in time when nodes in a PGM graph must execute if they are to process a continuous input signal without losing data. In Section 2.3, we derived bounds related to minimal buffering requirements that will be used throughout this dissertation. We informally called a node's execution pattern a rate in Section 2.3, and then formally defined an execution rate in Section 2.4. We also defined two types of node execution rate specifications: valid and well-defined. A well-defined rate specification is more restrictive than a valid rate specification, but unless the graph source nodes have well-defined rate specifications, it may not be possible to derive valid rate specifications for the other nodes in an otherwise valid PGM graph. A necessary and sufficient condition was presented for detecting this situation.

Given a rate specification $R_u = (x_u, y_u)$ for node u and a queue q such that $\psi(q) = (u, v)$, a rate specification for node v is $R_v = (x_v, y_v)$ where

$$\begin{aligned} x_v &= \frac{prd(q)}{\gcd(prd(q) \cdot x_u,\, cns(q))} \cdot x_u \\ \text{and} \quad y_v &= \frac{cns(q)}{\gcd(prd(q) \cdot x_u,\, cns(q))} \cdot y_u \end{aligned}$$

if queue q is the only input queue to node v. Moreover, it was shown that if $R_u = (x_u, y_u)$ is valid, then $R_v = (x_v, y_v)$ is valid, and if $R_u = (x_u, y_u)$ is well-defined, then $R_v = (x_v, y_v)$

is well-defined.

When a consumer node has more than one input queue, a slightly more complicated formula is needed to compute its rate specification. If node v has more than one input queue, and node v is not in a cycle, the execution rate $R_v = (x_v, y_v)$ for node v is well-defined if

$$\begin{aligned} y_v &= \mathrm{lcm}\left\{ \frac{cns(q) \cdot y_u}{\gcd(prd(q) \cdot x_u, cns(q))} \mid \forall q \in E \wedge \forall u \in V \; : \psi(q) = (u, v) \right\}, \\ x_v &= y_v \cdot \frac{prd(q) \cdot x_u}{cns(q) \cdot y_u}, \end{aligned}$$

and the execution rate specifications of its producers are all well-defined. If one of the input queues to node v is a back edge in a cycle, then a well-defined execution rate for node v can be computed using the same formula if the back edge is initialized with enough tokens to ensure that it is always over threshold.

Chapter 3

Software Synthesis

3.1 Introduction

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, an embedded signal processing system developed with a PGM graph may be 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 sent to the 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 primary problem in developing embedded signal processing systems with PGM is transforming the processing graph into a predictable real-time system in which latency and memory usage can be managed — all the while ensuring no data is lost. In this chapter, we show how to combine software engineering techniques with real-time scheduling theory to solve this problem. In the parlance of software engineering methodologies, we develop a synthesis method. We show how to manage latency using this method in Chapter 4. The synthesis of real-time systems from PGM graphs involves 3 steps:

1. Identification of the rates at which nodes in a PGM graph must execute if they are to process the signal in real time.

2. Construction of a mapping of each node to a task in the RBE task model so that real-time processing can be achieved in the embedded system.

3. Verification that the resulting task set is schedulable so that we can guarantee real-time execution.

Step 1 is described in Section 3.2. Steps 2 and 3 are described in Section 3.3. Step 2 provides a model of execution that we can use to analyze the latency a signal will encounter and the amount of memory required to buffer tokens in an implementation. However, the analysis of latency and memory requirements only holds if the task set is schedulable. Thus, the schedulability of the task set is tested in Step 3. If the task set is not schedulable, Steps 2 and 3 must be repeated with a modified set of parameters used in the mapping of PGM nodes to RBE tasks.

We continue to assume the strong synchrony hypothesis when we compute well- defined execution rate specifications in Section 3.2 and then relax this assumption in Section 3.3 where it is shown how to schedule nodes on a uniprocessor so that each node executes according to its rate specification.

3.2 Computing Node Execution Rates

In Chapter 2, we presented formulas for computing valid and well-defined execution rate specifications for a consumer node using the rate specifications of its producer nodes and the dataflow attributes on its input queues. Here we present algorithms for computing execution rate specifications for every node in a PGM graph using the equations of Chapter 2 and the execution rate specifications of the graph source nodes. We begin with acyclic graphs and then extend the algorithms to support cyclic graphs.

3.2.1 Computing Node Execution Rates in Acyclic Graphs

The execution rate of every node in an acyclic PGM graph can be computed in a straightforward manner: simply perform a topological sort of the graph and walk through the sorted list of nodes computing each node's execution rate using the theorems of Chapter 2. A topological sort of an acyclic graph $G = (V, E, \psi)$ is a linear ordering of all nodes in V such that, if queue q is in E and $\psi(q) = (u, v)$, then node u appears before node v in the ordering. For example, consider the graph of Figure 3.1(a). The topologically sorted list of nodes is $rpstuvwz$ and is shown in Figure 3.1(b). Given well defined execution rates for source nodes p, r, and t, the execution rate of every other node in Figure 3.1(a) can be derived by applying Theorem 2.4.9 to each node as it is encountered in the topologically sorted list.

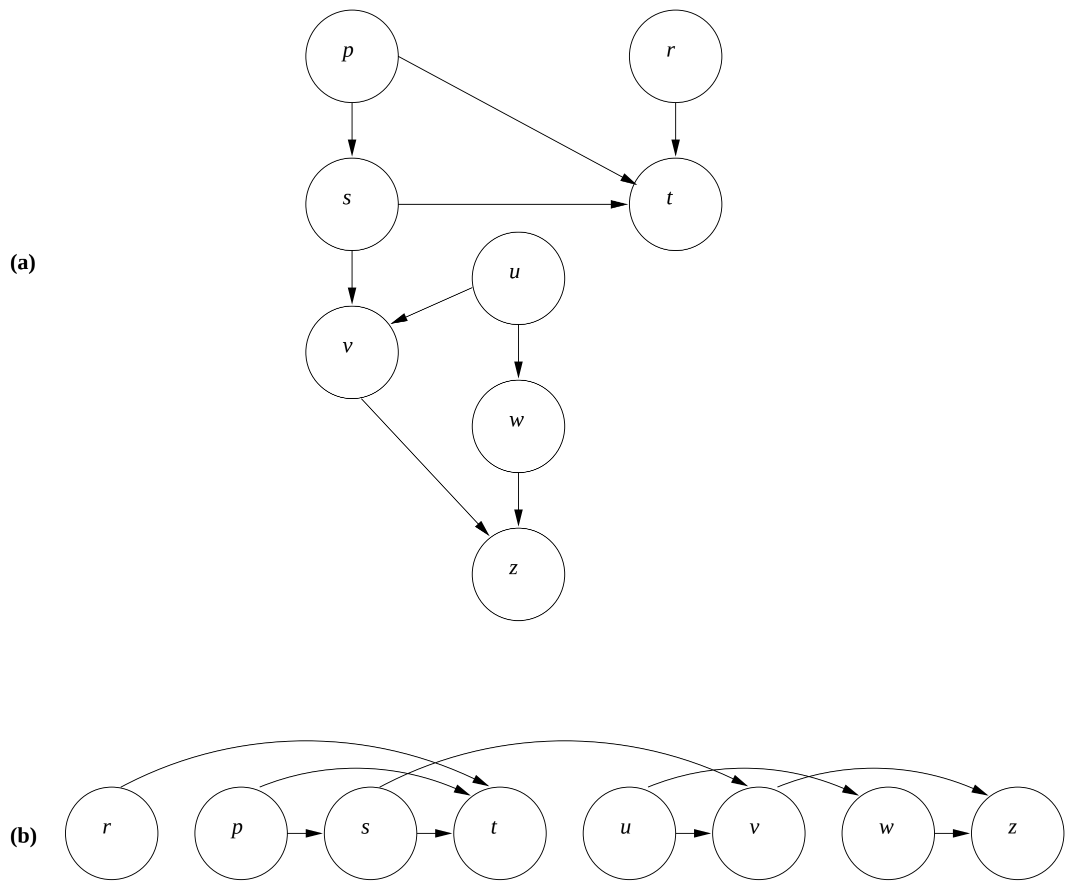


Figure 3.1: A simple acyclic graph and its topological sort.

Algorithm 1. Topological-Sort for graph G [14].
Input: *A directed acyclic Graph G.*
Output: *A linked list of sorted nodes.*

```
Topological-Sort(G)
    call DFS(G) to compute finishing times f[v] for each node v
    as each vertex is finished, insert it onto the front of a linked list
    return the linked list of nodes

DFS(G)
    for each vertex u ∈ V do
        color[u] := WHITE
    endfor
    time := 0
    for each vertex u ∈ V do
        if color[u] = WHITE then
            DFS-Visit(u)      /* Visit the previously undiscovered vertex */
        endif
    endfor

DFS-Visit(u)
    color[u] := GRAY          /* White vertex u has just been discovered */
    time := time + 1                                     /* Update time */
    d[u] := time                               /* Store the discovery time */
    for each v ∈ Adj[u] do                          /* Explore edge (u,v) */
        if color[u] = WHITE then
            DFS-Visit(v)      /* Visit the previously undiscovered vertex */
        endif
        color[u] := BLACK                     /* Blacken u; it is finished */
        time := time + 1                                 /* Update time */
        f[u] := time                              /* Store the finish time */
    endfor
```

Algorithm 2. Derive Node Execution Rates for each node in acyclic G.
Input: *A directed acyclic Graph G with execution rates defined for all source nodes.*
Output: *A linked list of sorted nodes with execution rates computed.*
Derive-Acyclic-Rates(G)

1 *SortedList* := *Topological-Sort*(G)
2 ***for*** *each node* $u \in$ *SortedList* ***do***
3   ***if*** *node* u *is not a source node* ***then***
4   *Use Theorem 2.4.9 to derive* R_u
5   ***endif***
6 ***endfor***
7 ***return*** *the linked list of nodes with execution rates computed*

The pseudo code for computing node execution rates in acyclic PGM graphs is presented in Algorithms 1 and 2. Algorithm 1 uses a depth-first search (DFS) to create a topological sort of the graph, and is the pseudo code from *Introduction to Algorithms* by Cormen *et al.* [14]. (The algorithm has been reproduced here for completeness since it is used by several algorithms presented here and in the next section.) Algorithm 2 uses the Topological-Sort(G) function of Algorithm 1 to derive the execution rate of each node in an acyclic graph. Once an acyclic graph is topologically sorted, all of the producers to node v will be placed in front of node v in the sorted list. Thus by the time the execution rate of node v is derived at line 4 of Algorithm 2, the execution rates for all of node v's producers will have been computed. For example, consider the graph and its topological sort in Figure 3.1 once again. By the time the execution rate of node t is derived, the execution rates of nodes r, p, and s will be known.

A slightly more complicated acyclic graph is shown in Figure 3.2. The topologically sorted list of nodes for this graph is

$$I_2DEFKLMNSI_1ABCGHIJTUVPRQWO_1.$$

Notice that all of the predecessors to node P come before node P in the sorted list of nodes. It is this property of a topologically sort list of nodes that Algorithm 2 exploits to derive the execution rate of nodes in an acyclic graph. Applying Algorithm 2 to the graph of Figure 3.2, with the execution rate of the two source nodes defined to be $(1, y)$, yields the node execution rates shown in Table 3.1.

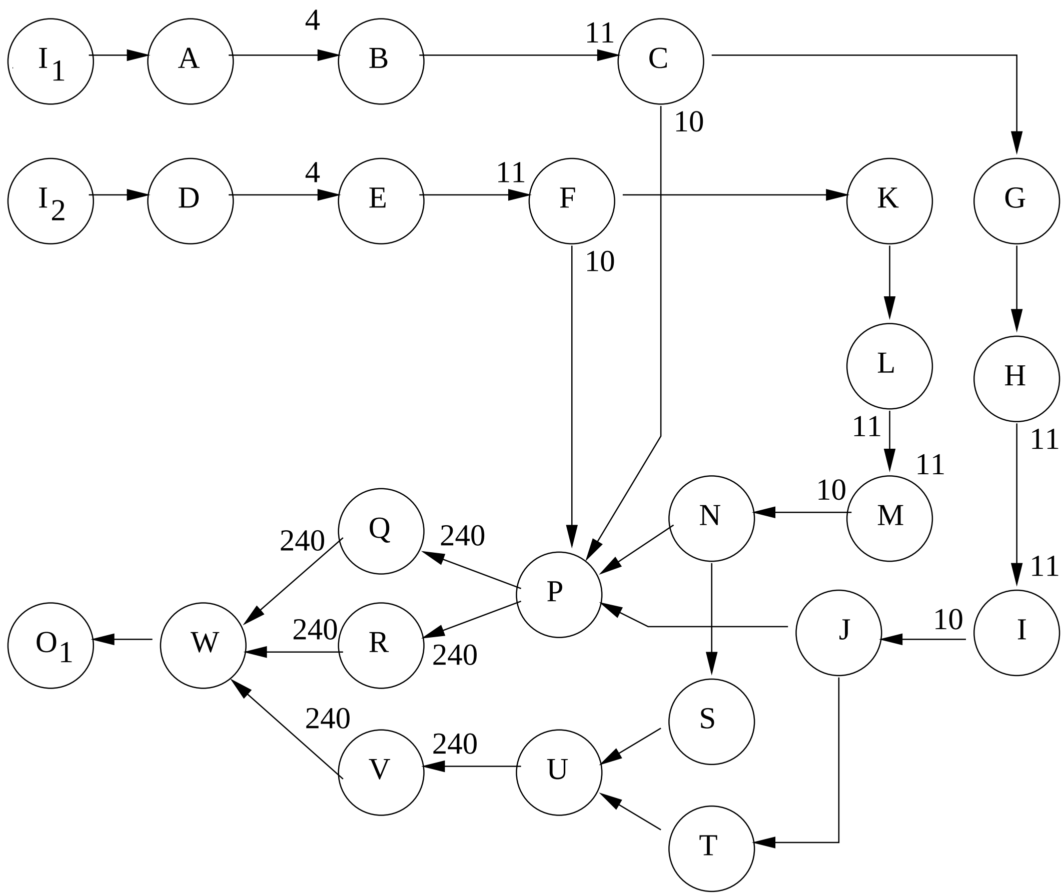


Figure 3.2: An acyclic PGM graph. All queues have produce amounts equal to 1 unless otherwise noted. Each queue's threshold value is equal to its consume value, and all consume (threshold) values are equal to 1 unless otherwise specified. Produce amounts greater than 1 are displayed near the tail of the queue and consume (threshold) amounts greater than one are displayed near the head of the queue. For example, the queue joining nodes A and B has a produce amount of 1, a threshold of 4, and a consume amount of 4.

Node	(x_u,y_u)
I_2	$(1, y)$
D	$(1, y)$
E	$(1, 4y)$
F	$(1, 44y)$
K	$(1, 44y)$
L	$(1, 44y)$
M	$(1, 44y)$
N	$(10, 44y)$
S	$(10, 44y)$
I_1	$(1, y)$
A	$(1, y)$
B	$(1, 4y)$
C	$(1, 44y)$
G	$(1, 44y)$
H	$(1, 44y)$
I	$(1, 44y)$
J	$(10, 44y)$
T	$(10, 44y)$
U	$(10, 44y)$
V	$(1, 24 \cdot 44y)$
P	$(10, 44y)$
R	$(1, 24 \cdot 44y)$
Q	$(1, 24 \cdot 44y)$
W	$(240, 24 \cdot 44y)$
O_1	$(240, 24 \cdot 44y)$

Table 3.1: Execution rate specifications for nodes in the graph of Figure 3.2. The graph nodes are listed in the order returned from the topological sort of the graph of Figure 3.2 that was performed by Derive-Acyclic-Rates(G) of Algorithm 2 — that is, the nodes are sorted topologically. The second column shows the execution rate of each node in the graph derived by Derive-Acyclic-Rates(G) of Algorithm 2.

3.2.2 Computing Node Execution Rates in Cyclic Graphs

We now extend Algorithm 2 to derive valid execution rate specifications for nodes in cyclic graphs. Let nodes v and w be nodes in a cycle and let queue q be a back edge in the cycle such that $\psi(q) = (v, w)$. Recall from Section 2.4.2 that if queue q is always over threshold, then the execution rate of node w can be derived without using the execution rate of node v. Moreover, by Theorem 2.4.11, if the number of initial tokens on a back edge q is computed using Equation (2.17), then queue q will always be over threshold. If each back edge in a PGM graph G is initialized so that it is always over threshold and G contains only simple cycles, then Algorithm 3 can be used to derive the execution rate of every node in the cyclic graph, even though a strict topological sort of a cyclic graph is not possible. When the graph contains a cycle, Topological-Sort(G) of Algorithm 1 produces a partial ordering of the graph. The resulting list of sorted nodes is a topological sort of an equivalent graph without back edges. Consider the cyclic graph in Figure 3.3. Other than the back edge labeled q, the graphs of Figure 3.2 and Figure 3.3 are identical. Applying Algorithm 1 to the graph in Figure 3.3 yields the same sorted list of nodes we obtained with the acyclic version of the graph. Moreover, the execution rates computed for each node in Figure 3.3 with Algorithm 3 are the same as the rates computed with Algorithm 2 and the graph in Figure 3.1.

Algorithm 3 cannot be applied to graphs containing non-disjoint cycles. In this case, the sorted list of nodes must be processed multiple times. Consider the graph of Figure 3.4. Using Algorithm 3 to derive the execution rates of the nodes in Figure 3.4, would yield invalid execution rates for nodes u, v, w, and z because their computed rates would not reflect the influence of source node r. Processing the list a second time would correct this deficiency because the second time through the list the execution rate for node t is defined. This results in the execution rate of source node r being reflected in the new execution rates computed for nodes u, v, w, and z. Even if there are more than two source nodes producing data for the two non-disjoint cycles, the sorted list of nodes only needs to be processed twice because the other source nodes will be placed in the list in front of their descendents. That is, the non-disjoint cycles are discovered in the DFS beginning with the first source node producing data for either of the cycles, and any other source nodes producing data for the cycle will be placed in the sorted list in front of the nodes contained within the cycle. Thus, the total number of times the list must be processed is $\max(1, n)$ where n is the maximum number of different cycles in which the same node appears.

The routine Derive-Rates(G) of Algorithm 4 uses Derive-Simple-Cycle-Rates(G) of

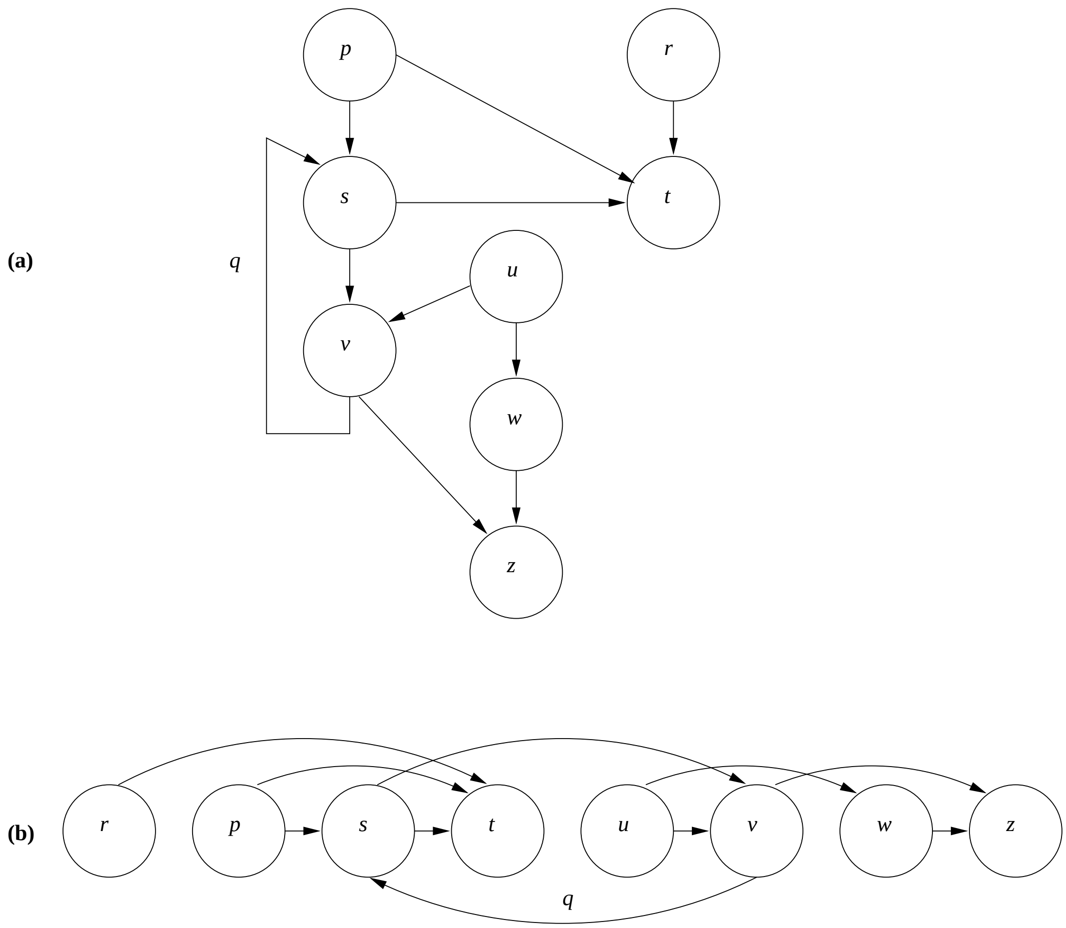


Figure 3.3: A cyclic graph and its topological sort. This is the same graph as Figure 3.1a with back edge q added to create a simple cycle. Note that the order of the nodes in the sorted graph is identical to Figure 3.1b.

Algorithm 3. Derive Node Execution Rates for each node in G.

Input: *A directed Graph G (which may contains simple cycles) with execution rates defined for all source nodes.*

Output: *A linked list of sorted nodes with execution rates defined.*

Derive-Simple-Cycle-Rates(G)

SortedList := *Topological-Sort(G)*
for each node $u \in$ *SortedList* **do**
  if node u is not a source node ***then***
    Use Theorem 2.4.9 to derive R_u *using the execution rate of*
      only those producers for which an execution rate has been
      defined and exclude back edges in the computation
  endif
endfor
return *the linked list of nodes with execution rates defined*

(a)

(b)

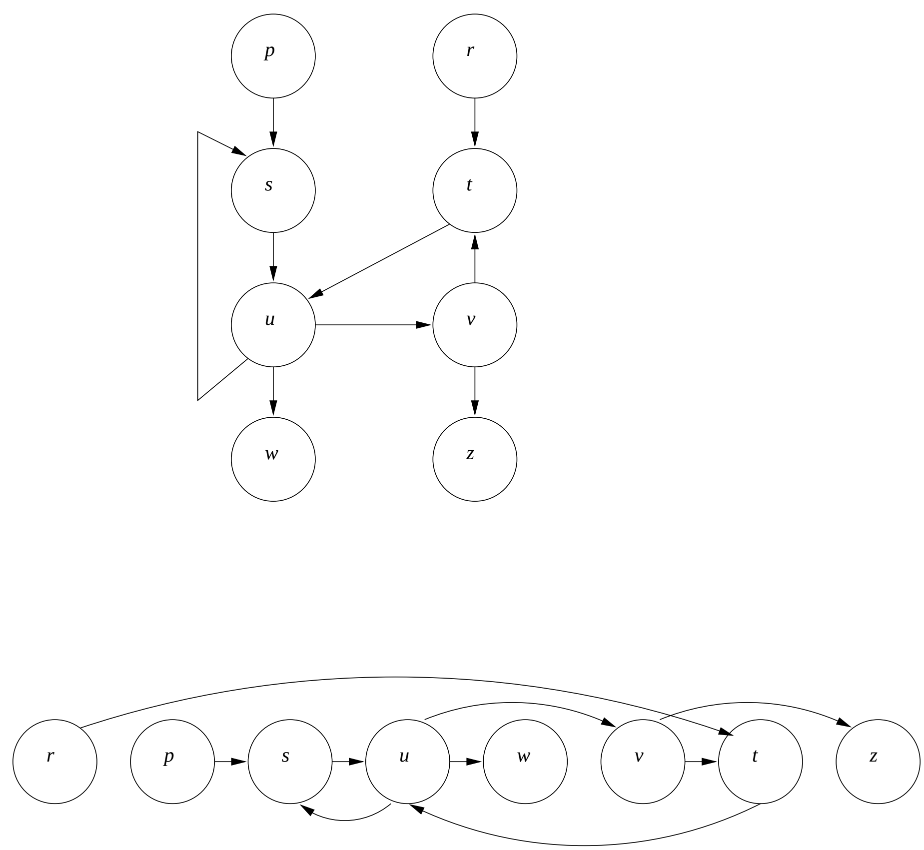


Figure 3.4: **(a)** A cyclic graph containing two non-disjoint cycles. Node u appears in cycles sus and $uvtu$. **(b)** The same cyclic graph topologically sorted. The back edges detected with a depth-first search are drawn below the sorted graph.

Algorithm 4. Derive Node Execution Rates for each node in G.
Input: *A directed Graph G (which may contain non-disjoint cycles) with execution rates defined for all source nodes.*
Output: *A linked list of sorted nodes with execution rates defined.*
Derive-Rates(G)

SortedList := *Derive-Simple-Cycle-Rates(G)*
n := *maximum number of cycles in which the same node appears*
for $i = 1$ ***to*** $n - 1$ ***do***
    for each node $u \in SortedList$ ***do***
        if node u *is not a source node* ***then***
            Use Theorem 2.4.9 to derive R_u using the execution rate of only those producers for which an execution rate has been defined and exclude back edges in the computation
        endif
    endfor
endfor
return *the linked list of nodes with execution rates defined*

Algorithm 3 to sort the nodes of G and to derive the node execution rates as though the graph contained only simple cycles or was acyclic. If the graph contains non-disjoint cycles, the sorted list of nodes is processed another $n-1$ times where n is the maximum number of cycles in which the same node appears. If the graph is acyclic, or only contains simple cycles, the list is only processed once (by Derive-Simple-Cycle-Rates(G)). Thus, Algorithm 4 can be used to derive the execution rate of every node in a PGM graph. If the rate specifications for the graph source nodes are well defined and the number of initial tokens on each identified back edge is equal to or greater than the number of tokens computed using Equation (2.17), then the rate specifications computing using Algorithm 4 will also be well defined.

From the preceding discussion, Lemma 2.4.5, and Theorem 2.4.9, we obtain:

Theorem 3.2.1. *Let $G = (V, E, \psi)$ be a PGM graph for which a well defined execution rate specifications exist for every node v in V and let $R_i = (x_i, y_i)$ be a well defined rate specification for source node $i \in \mathcal{I}$ (the set of source nodes to graph G). Algorithm 4 returns well defined execution rates for every node in V if each identified back edge q with*

$\psi(q) = (v, w)$ is initialized with

$$init(q) = \left\lceil \frac{s_v - s_w + y_v}{y_w} \right\rceil \cdot x_w \cdot cns(q) + thr(q)$$

tokens where s_v and s_w represent the first execution time of nodes v and w respectively.

Equations to derive the time of the first execution of a nodes v and w (s_v and s_w) are presented in Chapter 4 as part of the analysis and management of latency.

We now have the techniques necessary to derive the execution rate of every node in a processing graph. Once the execution rates have been derived, the next step in the synthesis method is to map each node to a task in the RBE task model so that real-time processing can be achieved in the embedded system. The third step in the synthesis method is to verify the schedulability of the resulting RBE task set. The next section presents these two steps.

3.3 Scheduling Node Executions

The execution rate specifications computed using the algorithms in Section 3.2 represent the rate at which nodes need to execute if they are to process a signal in real time. We now address issues related to scheduling nodes in accordance with their rate specifications. To make sure nodes execute according to their rate specifications we execute the nodes according to the RBE model. As stated previously, the advantage of executing nodes according to the RBE model is that nodes are eligible for execution as soon as they are released, even if multiple release of a node occur at the same time. In comparison, the periodic model of execution requires that each release of a node be separated by a constant amount of time, which, as shown in Section 1.3, imposes additional latency on the signal. We manage the amount of imposed latency and memory required for buffering tokens on input queues when the nodes are executed according to the RBE model by scheduling nodes with a simple EDF scheduling algorithm. A relative-deadline parameter is associated with each node and released nodes are scheduled for execution according to deadlines derived using the relative-deadline parameter. As we saw in Section 1.3, the deadlines cannot be made arbitrarily small if we wish to maintain a schedulable task set. A task set is schedulable with the EDF algorithm if and only if the EDF algorithm produces a schedule in which no task instances miss their deadlines.

This section is organized as follows. First, we present the RBE model for completeness. We then show how to map each node in a PGM graph to a RBE task so that nodes execute in accordance with their rate specifications. Finally, we show how to determine if the resulting task set is schedulable.

3.3.1 RBE Task Model

RBE is a general task model consisting of a collection of independent processes specified by four parameters: (x, y, d, e). The pair (x, y) represents the execution rate of an RBE task where x is the number of executions expected to be requested in an interval of length y. Parameter d is a response time parameter that specifies the maximum desired time between the release of a task instance and the completion of its execution (i.e., d is the relative deadline). Parameter e is the maximum amount of processor time required for one execution of the task.

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

$$D_i(j) = \begin{cases} t_{i,j} + d_i & \text{if } 1 \leq j \leq x_i \\ \max(t_{i,j} + d_i, D_i(j - x_i) + y_i) & \text{if } j > x_i \end{cases} \tag{3.1}$$

The RBE task model makes no assumptions regarding when a task will be released. To understand how deadlines are assigned in an RBE task set using Equation (3.1), consider an RBE task set with $x = 1$ for all tasks. If tasks are released periodically, then deadlines are assigned as though each task were a periodic task with parameters (y, d, e) in a (p, d, c) periodic model. In a (p, d, c) periodic model, task T_i is released every p_i time units, must complete its execution within d_i time units of its release, and has a worst-case execution time of c_i time units. If $x = 1$ but the release times of tasks in an RBE task set are not periodic, then the second line of Equation (3.1) ensures that the j^{th} release of an RBE task will be assigned the same deadline as the j^{th} release of a periodic task with period y. Now consider an RBE task set in which x is greater than 1 for some of the tasks. If the tasks are periodic such that all x_i instances of task T_i are released at the same time (with period y_i), then deadlines are assigned as though each of the x_i releases was a periodic task with parameters (y_i, d_i, e_i). Thus, all x_i task releases would be assigned the same deadline: d_i time units from its release. If the x_i releases are periodic but not simultaneous, then each of the x_i releases in an interval of y_i time units is modeled as a separate periodic task with parameters (y_i, d_i, e_i) and assigned a deadline d_i time units from its (periodic) release. When the x_i releases of RBE task T_i are not periodic or when more than x_i releases of the task occur in an interval of length y_i, the second line of deadline-assignment Function (3.1) ensures that no more than x_i deadlines come due in an interval of length y_i.

Deadline-assignment Function (3.1) prevents release jitter from creating more processor demand in an interval by a task than that which is specified by the rate parameters. Release jitter is the phenomenon that occurs when release times vary such that in one interval fewer releases occur than expected and in another interval more releases occur than expected. The processor demand in an interval $[a, b]$ is the amount of processor time required to be available in $[a, b]$ to ensure that all tasks released prior to time b with deadlines in $[a, b]$ complete in $[a, b]$. The maximum processor demand in an interval $[a, b]$ occurs when

1. a marks the end of an interval in which the processor was idle (or 0 if the processor is never idle),
2. the processor is never idle in the interval $[a, b]$, and
3. as many deadlines as possible occur in the interval $[a, b]$.

If deadlines for instances of RBE task T_i were assigned by simply adding d_i to the task's release time, then more than x_i releases in an interval of length y_i may create more processor demand than the processor can support and deadlines may be missed. To ensure that no task instance misses a deadline, we must bound the maximum processor demand for all tasks in all intervals and verify that the processor has enough capacity to support the processor demand created by the RBE task set. We begin by bounding the maximum processor demand for a task in the interval $[0, L]$.

Lemma 3.3.1. *For preemptive scheduling of the execution of an RBE task T with parameters (x, y, d, e),*

$$\forall L > 0, \quad f\left(\frac{L - d + y}{y}\right) \cdot x \cdot e \tag{3.2}$$

is a least upper bound on the number of units of processor time required to be available in the interval $[0, L]$ to ensure that no task instance of T misses a deadline in $[0, L]$, where

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

Proof: To derive a least upper bound on the amount of processor time required to be available in the interval $[0, L]$, it suffices to consider a set of release times of T that results in the maximum processor demand in $[0, L]$. If t_j is the time of the j^{th} release of task T, then clearly the set of release times $t_j = 0$, $\forall j > 0$, is one such set. Under these release times, x instances of T have deadlines in $[0, d]$. After d time units have elapsed, x instances of T have deadlines every y time units. Thus the number of instances with deadlines in the interval $[d, L]$ is $\left\lfloor \frac{L-d}{y} \right\rfloor \cdot x$. Therefore, $\forall L \geq d$, the number of instances of T with deadlines in the interval $[0, L]$ is

$$\begin{aligned} x + \left\lfloor \frac{L-d}{y} \right\rfloor \cdot x &= \left(1 + \left\lfloor \frac{L-d}{y} \right\rfloor\right) \cdot x \\ &= \left\lfloor \frac{L-d}{y} + 1 \right\rfloor \cdot x \\ &= \left\lfloor \frac{L-d+y}{y} \right\rfloor \cdot x. \end{aligned} \tag{3.3}$$

For all $L < d$, no instances of T have deadlines in $[0, L]$, hence the right-hand side of Equation (3.3) gives the maximum number of instances of T with deadlines in the interval $[0, L]$, for all $L > 0$.

Finally, as each instance of T requires e units of processor time to execute to completion, Expression (3.2) is a least upper bound on the number of units of processor time required to be available in the interval $[0, L]$ to ensure that no instance of T misses a deadline in $[0, L]$. □

Note that there are many sets of task release times that maximize the processor demand of a task in the interval $[0, L]$. For example, given the recurrence relation for deadlines defined by Equation (3.1), it is straightforward to show that the less pathological set of task release times $t_j = \left\lfloor \frac{j-1}{x} \right\rfloor \cdot y$, $\forall j > 0$, also maximizes the processor demand of task T in the interval $[0, L]$.

A task set is feasible if and only if there exists a schedule such that no task instance misses its deadline. Thus, if $Demand(L)$ represents the total processor demand in an interval of length L, a task set is feasible if and only if $L \geq Demand(L)$ for all $L > 0$.

Theorem 3.3.2. *Let $\mathcal{T} = \{(x_1, y_1, d_1, e_1), \ldots\ (x_n, y_n, d_n, e_n)\}$ be a set of RBE tasks. $\mathcal{T}$ will be feasible if and only if*

$$\forall L > 0, \quad L \geq \sum_{i=1}^{n} f\left(\frac{L - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i \tag{3.4}$$

where $f(a)$ is as defined in Lemma 3.3.1.

Proof: ($\Rightarrow$) The necessity of Equation (3.4) is shown by establishing the contrapositive, i.e., a negative result from Equation (3.4) implies that $\mathcal{T}$ is not feasible. To show that $\mathcal{T}$ is not feasible it suffices to demonstrate the existence of a set of task release times for which at least one release of a task in $\mathcal{T}$ misses a deadline.

Assume a negative result from Equation (3.4), that is,

$$\exists l > 0 : l < \sum_{i=1}^{n} f\left(\frac{l - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i.$$

Let t_{ij} be the release time of the j^{th} instance of task i in $\mathcal{T}$. Consider the set of release times $t_{ij} = 0$, where $1 \leq i \leq n$ and $j > 0$. By Lemma 3.3.1, the least upper bound for the processor demand created by task T_i is $f\left(\frac{L-d_i+y_i}{y_i}\right) \cdot x_i \cdot e_i$ units of processor time in the interval of $[0, l]$. Moreover, from the proof of Lemma 3.3.1, the set of release times $t_{ij} = 0$, $1 \leq i \leq n$ and $j > 0$, creates the maximum processor demand possible in the interval $[0, l]$. Therefore, for $\mathcal{T}$ to be feasible, it is required that $\sum_{i=1}^{n} f\left(\frac{l-d_i+y_i}{y_i}\right) \cdot x_i \cdot e_i$ units of work be available in $[0, l]$. However, since

$$l < \sum_{i=1}^{n} f\left(\frac{l - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i,$$

an instance of a task in $\mathcal{T}$ must miss a deadline in $[0, l]$. Thus there exists a set of release times such that a deadline is missed when Equation (3.4) does not hold. This proves the contrapositive.

($\Leftarrow$) To show the sufficiency of Equation (3.4) it is shown that the preemptive EDF scheduling algorithm can schedule all releases of tasks in $\mathcal{T}$ without any job missing a deadline if the tasks satisfy Equation (3.4) (see Section 1.3 for a definition of the EDF algorithm). This is shown by contradiction.

Assume that $\mathcal{T}$ satisfies Equation (3.4) and yet there exists a release of a task in $\mathcal{T}$ that misses a deadline at some point in time when $\mathcal{T}$ is scheduled by the EDF algorithm. Let t_d be the earliest point in time at which a deadline is missed and let t_0 be the later of:

- the end of the last interval prior to t_d in which the processor has been idle (or 0 if the processor has never been idle), or
- the latest time prior to t_d at which a task instance with deadline after t_d stops executing prior to t_d (or time 0 if such an instance does not execute prior to t_d).

By the choice of t_0, *(i)* only releases with deadlines less than time t_d execute in the interval $[t_0, t_d]$, and *(ii)* the processor is fully used in $[t_0, t_d]$. Only releases with deadlines less than time t_d execute in the interval $[t_0, t_d]$ and, by the choice of t_0, any task instances released before t_0 will have completed executing by t_0. Thus, by a result due to Baruah *et al.* (Lemma 3.5 in reference [6]), at most

$$\sum_{i=1}^{n} \left\lfloor \frac{t_d - t_0 - d_i + y_i}{y_i} \right\rfloor \cdot x_i$$

instances of tasks in $\mathcal{T}$ can have deadlines in the interval $[t_0, t_d]$, and

$$\sum_{i=1}^{n} \left\lfloor \frac{t_d - t_0 - d_i + y_i}{y_i} \right\rfloor \cdot x_i \cdot e_i$$

is the least upper bound on the units of processor time required to be available in the interval $[t_0, t_d]$ to ensure that no task release misses a deadline in $[t_0, t_d]$. The problem of scheduling the RBE task set in the interval $[t_o, t_d]$ is equivalent to scheduling a periodic task set where each of the x_i instances of task T_i are represented by a separate periodic task since we have assumed worst-case task releases in which all x_i instances of task T_i are released at the same time. It is a well known fact that EDF is an optimal scheduling algorithm for independent periodic task sets [45]. By optimal, we mean that if a valid schedule exists, the EDF scheduling algorithm will create one. Let $\mathcal{E}$ be the amount of processor time consumed by tasks in $\mathcal{T}$ in the interval $[t_0, t_d]$ when scheduled by the EDF algorithm. Since $\sum_{i=1}^{n} f\left(\frac{t_d - t_0 - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i$ is a least upper bound on the processor time required in the interval $[t_0, t_d]$ and $\mathcal{E}$ is processor time consumed using the EDF algorithm, it must be the case that

$$\sum_{i=1}^{n} f\left(\frac{t_d - t_0 - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i \geq \mathcal{E}.$$

Thus, since the processor is fully used in the interval $[t_0, t_d]$ and since a deadline is missed at time t_d, it follows that $\mathcal{E}$ is greater than the processor time available in the interval $[t_0, t_d]$, namely $t_d - t_0$. Hence,

$$\sum_{i=1}^{n} f\left(\frac{t_d - t_0 - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i \geq \mathcal{E} > t_d - t_0.$$

However this contradicts our assumption that $\mathcal{T}$ satisfies Equation (3.4). Hence if $\mathcal{T}$

satisfies Equation (3.4), then no release of a task in $\mathcal{T}$ misses a deadline when $\mathcal{T}$ is scheduled by the EDF algorithm. It follows that satisfying Equation (3.4) is a sufficient condition for feasibility. Thus Equation (3.4) is a necessary and sufficient condition for determining if a task set $\mathcal{T}$ is feasible. $\square$

If the cumulative processor utilization for an RBE task set is strictly less than one (i.e., $\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} < 1$) then Condition (3.4) can be evaluated efficiently (in pseudo-polynomial time) using techniques developed by Baruah *et al.* [6]. Moreover, when $d_i = y_i$ for all T_i in $\mathcal{T}$, the evaluation of Condition (3.4) reduces to the polynomial-time feasibility condition

$$\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} \leq 1 \tag{3.5}$$

since

$$\begin{aligned}
\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} \leq 1 \implies \forall L > 0, \quad L &\geq \sum_{i=1}^{n} L \cdot \frac{x_i \cdot e_i}{y_i} \\
&= \sum_{i=1}^{n} \frac{L}{y_i} \cdot x_i \cdot e_i \\
&= \sum_{i=1}^{n} \frac{L - y_i + y_i}{y_i} \cdot x_i \cdot e_i \\
&= \sum_{i=1}^{n} \frac{L - d_i + y_i}{y_i} \cdot x_i \cdot e_i \quad \text{since } d_i = y_i \\
&\geq \sum_{i=1}^{n} f\left(\frac{L - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i.
\end{aligned} \tag{3.6}$$

Equation (3.5) computes processor utilization for the task set $\mathcal{T}$ and is a generalization of the EDF feasibility condition $\sum_{i=1}^{n} \frac{e_i}{y_i} \leq 1$ for independent tasks with deadlines equal to their period given by Liu & Layland [45].

3.3.2 Mapping Nodes to Real-Time Tasks

Step 2 of the synthesis method associates each PGM node with an RBE task by associating each node u with a four tuple (x_u, y_u, d_u, e_u) that characterizes an RBE task. Parameters x_u and y_u are derived using Algorithm 4 of Section 2.4.2. Parameter e_u is the worst-case execution time for node u, which we assume is supplied.

The only free parameter is the relative-deadline parameter d_u. The choice for the value of d_u influences processor capacity requirements, latency, and buffer requirements. In general, a smaller value chosen for d_u will result in less latency and memory requirements than a larger d_u value, but at a cost of increased processor capacity requirements. The increased processor capacity is due to the increased processor demand for node u that is created in an interval of length L as d_u is reduced. For example, if $d_u = y_u$, then the processor demand in the interval $[0, L]$ must be less than or equal to

$$\left\lfloor \frac{L - d_u + y_u}{y_u} \right\rfloor \cdot x_u \cdot e_u = \left\lfloor \frac{L - y_u + y_u}{y_u} \right\rfloor \cdot x_u \cdot e_u = \left\lfloor \frac{L}{y_u} \right\rfloor \cdot x_u \cdot e_u$$

by Lemma 3.3.1. If $d_u < y_u$, then the processor demand in the interval can increase since

$$\left\lfloor \frac{L - d_u + y_u}{y_u} \right\rfloor \cdot x_u \cdot e_u \geq \left\lfloor \frac{L}{y_u} \right\rfloor \cdot x_u \cdot e_u.$$

Execution time, produce, threshold, consume, and deadline values all affect schedulability, latency, and buffer requirements, and one can trade-off one metric for any other. In mapping the graph to a set of RBE tasks, relative-deadline parameters need to be selected that result in modest buffering on the graph edges without overloading the processor with too much processing demand. Selecting deadline values for nodes is usually an iterative process. We begin the process by setting $d_u = y_u$ for each node u in the graph. We then evaluate the schedulability of the resulting task set in Step 3. If the task set is schedulable, then latency and memory requirements are computed using techniques outlined in the next two chapters. If the computed latency or memory requirements exceed tolerances specified for the application, then the process starts over at Step 2 with smaller deadline values.

In addition to requiring deadline parameters that result in a schedulable task set, we require $d_v \geq d_u$ when there exists a queue joining node u to node v (i.e. $\exists q : \psi(q) = (u, v)$). Thus, for a path from a graph source node to a graph output node w, the deadline parameter of node w will be greater than or equal to the deadline parameter of any other node in the path. Observe that, since $y_v \geq y_u$ for a producer/consumer pair, setting $d_u = y_u$ for every node in the graph satisfies the requirement $d_v \geq d_u$. We show in the next chapter that setting deadlines in this manner greatly simplifies the management of latency. Moreover, it allows us to achieve near minimal latency in an implementation.

Given an RBE task system derived from a PGM graph, we must ensure valid execution. Thus a FIFO data queue is created for each PGM queue and a task is released when all of its input queues are over threshold. Released tasks are scheduled with a preemptive

EDF scheduling algorithm and deadlines are assigned using Equation (3.1).

3.3.3 Scheduling Theory Results

Step 3 of the synthesis method is to verify that the resulting task set is schedulable so that we can guarantee real-time execution. For a given scheduling algorithm, a task set is *schedulable* if and only if the algorithm produces a schedule in which no task misses its deadline. A scheduling algorithm is *optimal* if it can produce a valid schedule for any task set that is feasible. The sufficiency of Condition (3.4) for determining the feasibility of an RBE task set was established by showing that, if the task set was feasible, the EDF scheduling algorithm produces a valid schedule. Thus, for an RBE task set, EDF is an optimal scheduling algorithm. In this section, we show that the feasibility problem for PGM graphs is co-NP-hard in the strong sense, and EDF is not an optimal scheduling algorithm for PGM graphs. Although in general the feasibility problem is co-NP-hard in the strong sense and EDF is not optimal, we show that Condition (3.4) is a sufficient condition for determining the schedulability of PGM graphs with the EDF algorithm. It is shown that Condition (3.4) is not a necessary condition because it over states the maximum processor demand for some graphs. However, in practice, the sufficiency of Condition (3.4) is adequate for approximating processor demand since most system engineers prefer to over estimate the processor demand and almost never allow the processor to be fully utilized. We leave open the problem of finding an optimal scheduling algorithm for PGM graphs, and focus instead on scheduling PGM graphs with a variation of the EDF scheduling algorithm called rate-based-execution earliest-deadline-first (RBE-EDF).

We have previously shown that the feasibility problem for PGM graphs is co-NP-hard when a maximum buffer capacity is associated with each queue [5]. The same argument (first made by Sanjoy Baruah [4]) is used here to show that the feasibility problem for PGM graphs implemented as an RBE task set is also co-NP-hard in the strong sense.

Theorem 3.3.3. *The problem of determining if a PGM graph implemented as an RBE task set is feasible is co-NP-hard in the strong sense.*

Proof: The proof is by reduction from the *Simultaneous Congruences Problem* (SCP). This problem was introduced by Leung and Merrill and shown to be NP-hard [43]. Subsequently, it was proven by Baruah *et al.* that the problem is NP-hard in the strong sense [7].

The Simultaneous Congruences Problem is defined as follows:

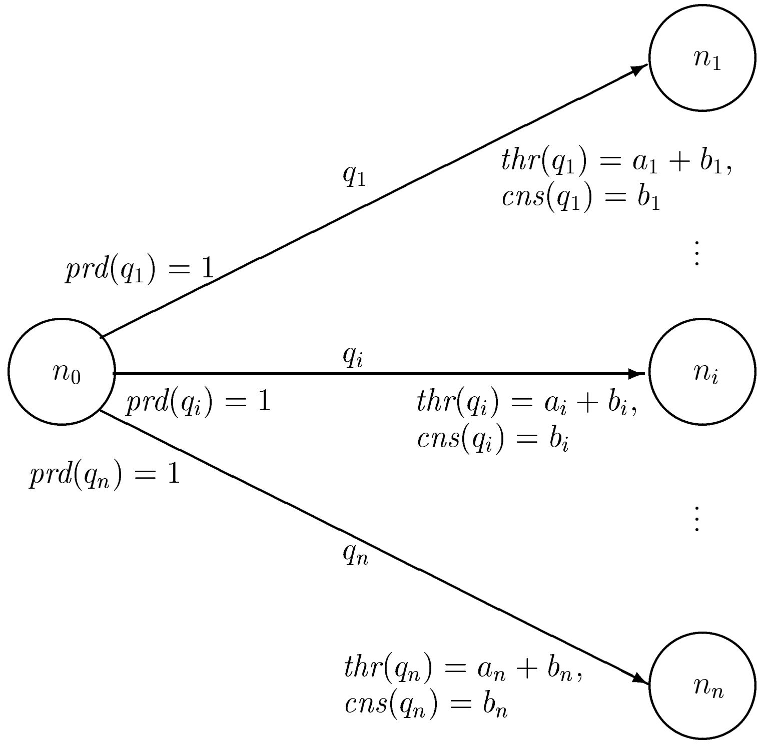


Figure 3.5: A PGM dataflow graph constructed during the proof of Theorem 3.3.3. The RBE parameters for source node n_0 are $(x_0 = 1, y_0 = 1, d_0 = 1, e_0 = 0)$. The RBE parameters for nodes n_i, for $1 \leq i \leq n$, are $(x_i, y_i, d_i = 1, e_i = \frac{1}{r-1})$ where x_i and y_i are derived using equation Theorem 2.4.9 and r is the integer given in the SCP problem.

Given n ordered pairs of positive integers (a_1, b_1), (a_2, b_2), ..., (a_n, b_n), and 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 with integer r, we construct an instance of the PGM graph feasibility problem as shown in Figure 3.5. Consider a graph with $n+1$ nodes, labeled n_0 through n_n. Let the RBE parameters for source node n_0 be $(x_0 = 1, y_0 = 1, d_0 = 1, e_0 = 0)$. Let the RBE parameters for sink node n_i, $1 \leq i \leq n$, be $(x_i, y_i, d_i = 1, e_i = \frac{1}{r-1})$ where x_i and y_i are derived using Theorem 2.4.9 and r is the integer given in the SCP problem. For each queue from node n_0 to node n_i, $1 \leq i \leq n$, let $prd(q_i) = 1$, $thr(q_i) = a_i + b_i$, and $cns(n_i) = b_i$. Observe that the i'th node is enabled for the first time at time $a_i + b_i$, and, if no deadlines are missed, completes execution within one time unit later. 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, a deadline *is* missed 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, $1 \le j \le r$. Thus, it is co-NP-hard in the strong sense to determine whether a PGM graph implemented as an RBE task set is feasible. □

As a consequence of Theorem 3.3.3, we are unable to determine in pseudo-polynomial time whether a given PGM dataflow graph is feasible or not (unless P = NP). Moreover EDF is not even an optimal scheduling algorithm for scheduling a PGM graph [4].

Theorem 3.3.4. *The EDF scheduling algorithm is not optimal for scheduling a PGM graph implemented as an RBE task set.*

Proof: We prove the non-optimality of EDF scheduling for PGM graphs implemented as an RBE task set by constructing a graph that is feasible but not schedulable with EDF scheduling. Consider the graph of Figure 3.6. The RBE parameters associated with a node are displayed above the node. For example, the RBE parameters for node v are $(x_v = 1, y_v = 7, d_v = 5, e_v = 5)$. Source nodes i_1 and i_2 are periodic with period 7 starting at time 0. The time-line execution shown in Figure 3.6 shows the execution of the graph under EDF scheduling for the first 14 time units. Node v misses its deadline at times 6 and 13 because node u executes before node w under EDF scheduling. If node w is given priority over node u, as in a static priority scheduler, then the graph is schedulable. The second time-line execution shows no deadlines are missed under static priority scheduling. The graph is feasible, but not schedulable under EDF scheduling. Thus, the EDF scheduling algorithm is not optimal for scheduling a PGM graph implemented as an RBE task set. □

It can also be shown that rate-monotonic scheduling, which is optimal for static-priority periodic task systems, is not optimal for scheduling PGM graphs. We leave open the problem of finding an optimal scheduling algorithm, and focus instead on scheduling PGM graphs with a variation of the EDF scheduling algorithm called RBE-EDF. Before describing the RBE-EDF scheduling algorithm, however, we first motivate its creation by showing why Condition (3.4) is a sufficient but not necessary condition for scheduling PGM graphs with an EDF algorithm.

For scheduling RBE tasks derived from a PGM graph, Condition (3.4) is not, in general, a necessary condition. In order for Condition (3.4) to be necessary, it is required,

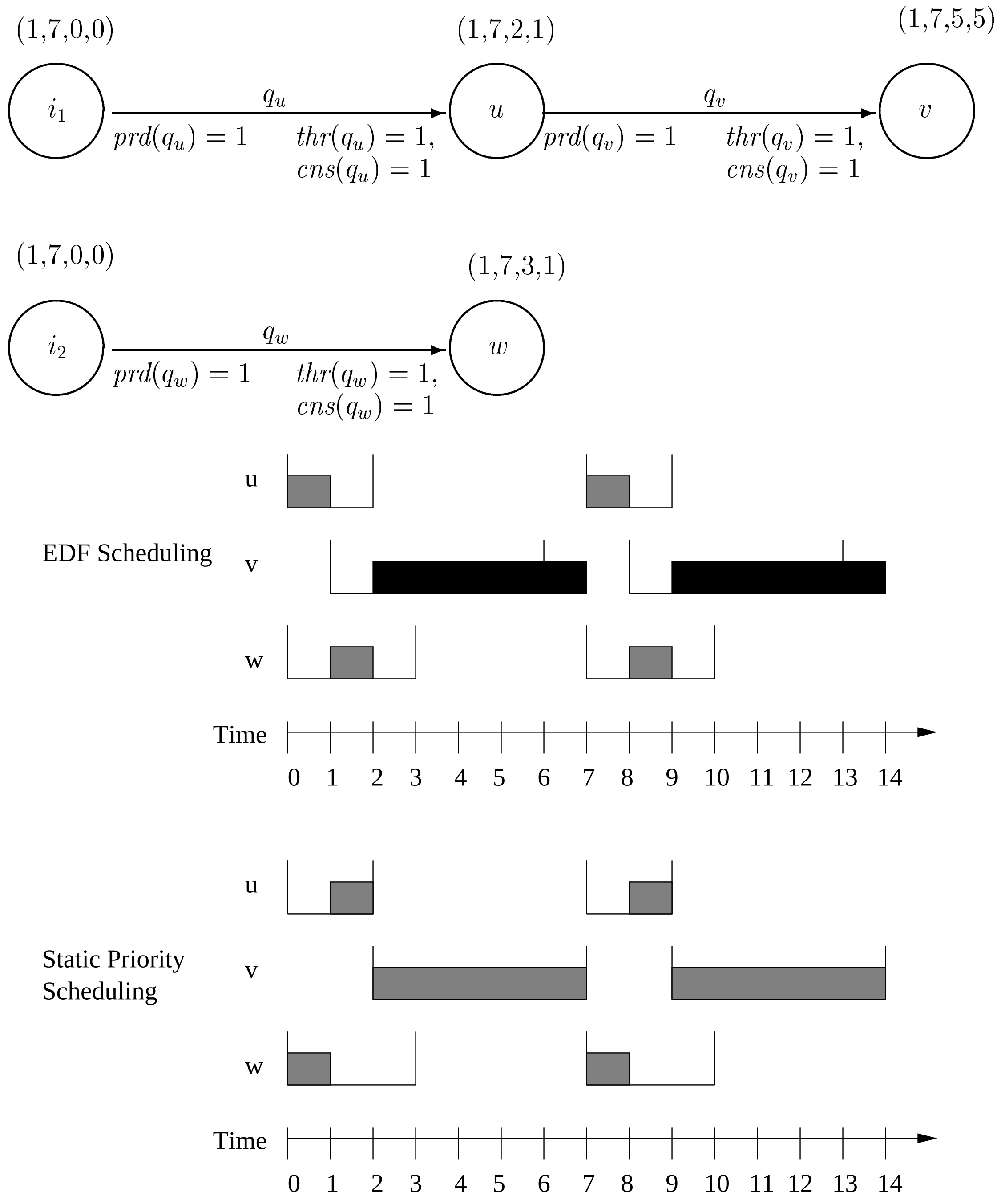


Figure 3.6: A PGM dataflow graph constructed during the proof of Theorem 3.3.4 and two time-line executions. The RBE parameters associated with a node are shown above the node. For example, the RBE parameters for node v are $(x_v = 1, y_v = 7, d_v = 5, e_v = 5)$. Source nodes i_1 and i_2 are periodic with period 7 starting at time 0. Thus the graph is feasible, but not schedulable under EDF scheduling. Node v misses its first deadline at time 6 under EDF scheduling.

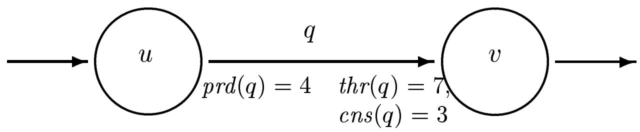


Figure 3.7: A two-node PGM graph. If $R_u = (1, y)$, scheduling condition (3.4) is a sufficient but not necessary condition for evaluating the schedulability of this graph under preemptive EDF scheduling.

in essence, that all x_u releases of node u occur immediately at the beginning of an interval of length y_u. For some nodes, such as node v in Figure 3.7, this is not possible. If the execution rate of node u is $R_u = (1, y)$, then $R_v = (4, 3y)$ but any execution of node u releases at most 2 executions of node v. Thus if $d_v = y$, there cannot be 4 simultaneous unexpired jobs for the task associated with node v, as assumed by schedulability condition (3.4).

While Condition (3.4) over states the processor demand that can exist for some graphs, it is a sufficient condition for evaluating the schedulability of a PGM graph implemented as an RBE task set. An affirmative result from condition (3.4) means that if all x_i instances of each task T_i in the RBE task set are released at the same time, there exists enough processor capacity that no task will miss its deadline. Thus, the sufficiency proof of Theorem 3.3.2 can also be used prove the sufficiency of Condition (3.4) for scheduling PGM graphs with the EDF algorithm according to the RBE model of execution. Moreover, we can take advantage of the release times allowed when Condition (3.4) evaluates in the affirmative to simplify the management of latency in the system.

Our RBE-EDF scheduling algorithm is the simple EDF scheduling algorithm in which deadlines are assigned using Equation (3.1) and a technique called *release-time inheritance*. Release-time inheritance creates a logical release time for a node that is equal to the release time that the node would have under the strong synchrony hypothesis. This is accomplished by setting the logical release time of node w to the logical release time of the node v that released node w. For example, consider the graph of Figure 3.8. Node u represents an external device that produces data once every three time units. Thus, the execution rates of nodes v and w are $R_v = (1, 3)$ and $R_w = (1, 6)$. Each node requires two time units of processor time to execute. Setting the relative deadline parameter for each node equal to its y parameter in its rate specification results in RBE parameters $(1, 3, 3, 2)$ and $(1, 6, 6, 2)$ for nodes v and w respectively. Three different executions of the graph are shown in Figure 3.9. In the first execution, under EDF scheduling, node w is released at time 5 and its absolute deadline is

$$\begin{aligned} \textit{actual release time} + d_w &= 5 + 6 \\ &= 11. \end{aligned}$$

Under RBE-EDF scheduling, node w is still released at time 5, but its logical release time is 3 since this is the logical release time of node v. Thus, the absolute deadline for node w under RBE-EDF scheduling is

$$\begin{aligned} \textit{logical release time} + d_w &= 3 + 6 \\ &= 9. \end{aligned}$$

There are three things to note here. First, the actual execution pattern under EDF and RBE-EDF scheduling is the same, as shown in the time-line of executions in Figure 3.9. Second, the logical release times for nodes v and w are the same as their execution times under the strong synchrony hypothesis. Third, node w finishes before its deadline under either EDF or RBE-EDF scheduling.

As long as the relative deadline parameter for node w is greater than the relative deadline parameter for any of its producers, release-time inheritance can be used to assign deadlines if Condition (3.4) results in the affirmative. This is because Equation (3.4) evaluates the processor demand over an interval of time as though all nodes were released at the same time. The greatest processor demand is created when every node w in the graph executes x_w times each time the source node executes. Under this condition, RBE-EDF scheduling is equivalent to releasing all nodes at the same time and then scheduling their execution with an EDF scheduler that breaks deadline ties according to topological order. Thus, if the task set created using our synthesis method is schedulable according to Condition (3.4), RBE-EDF will create an execution in which no task misses its deadline.

Thus, from Theorem 3.3.2 and the preceding discussion, we obtain Theorem 3.3.5.

Theorem 3.3.5. *Let $\mathcal{T} = \{(x_1, y_1, d_1, e_1), \ldots\ (x_n, y_n, d_n, e_n)\}$ be a set of tasks constructed from a PGM graph using our synthesis method. The processing graph $G = (V, E, \psi)$ is schedulable with the RBE-EDF scheduling algorithm if Equation* (3.4) *holds for $\mathcal{T}$.*

Proof: Under RBE-EDF scheduling, the maximum processor demand for a task set $\mathcal{T}$ synthesized from a PGM graph occurs when each production of a graph source node releases all x_i instances of task T_i, for $1 \leq i \leq n$, at the same time. Under the strong synchrony hypothesis, this is equivalent to having every node u in the graph execute x_u

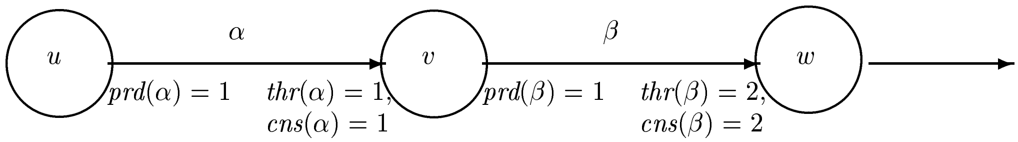


Figure 3.8: A three-node chain. Source node u is periodic with period 3. The RBE parameters of nodes v and w are $(1, 3, 3, 2)$ $(1, 6, 6, 2)$.

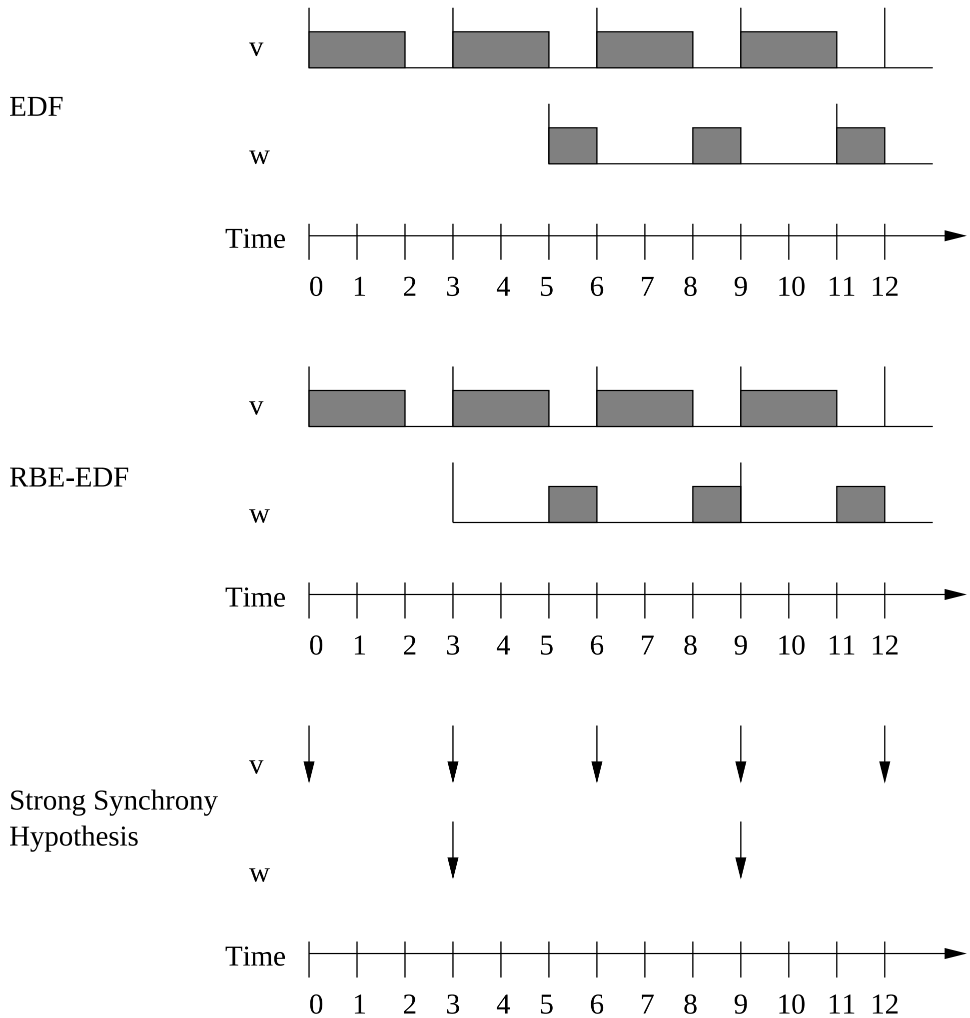


Figure 3.9: Execution of the graph in Figure 3.8 under EDF, RBE-EDF, and the strong synchrony hypothesis.

times at the instant a source node executes. Since RBE-EDF scheduling uses release-time inheritance, every task released when the graph source node produces data will have the same logical release time. Thus, by Lemma 3.3.1, the maximum processor demand over an interval L, is

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

By, Theorem 3.3.2, when deadlines are assigned using Equation (3.1), the EDF scheduling algorithm will schedule an RBE task set such that no deadlines are missed if

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

Since RBE-EDF scheduling is equivalent to releasing all task instances at the same time and then scheduling their execution with an EDF scheduler that breaks deadline ties according to a topological sort of the graph, the RBE-EDF scheduling algorithm will create a schedule in which no task misses its deadline if Equation (3.4) results in the affirmative. Thus, the processing graph $G = (V, E, \psi)$ is schedulable with the RBE-EDF scheduling algorithm if Equation (3.4) holds for $\mathcal{T}$. □

3.4 Summary

The synthesis of real-time systems from PGM graphs presented in this chapter involves 3 steps:

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

We presented techniques for deriving the execution rate of every node in an acyclic or cyclic processing graph and demonstrated the mapping of nodes to RBE real-time tasks. We also presented the RBE-EDF scheduling algorithm and a sufficient, but not necessary, condition for the schedulability of PGM graphs mapped to tasks and executed according the RBE task model. The RBE-EDF assigns deadlines using Equation (3.1) and a technique called release-time inheritance. Release-time inheritance creates a logical release time for a node that is equal to the release time that the node would have under the strong synchrony hypothesis.

In the next chapter, we show that managing latency is a straightforward process when graphs are scheduled with the RBE-EDF scheduling algorithm. However, the management of latency often requires an iteration of Steps 2 and 3 of the synthesis method as new deadline parameter values are selected to make trade-offs between processor demand and latency.

Chapter 4

Managing Latency

4.1 Introduction

We now address the issue of managing latency in signal processing systems. We begin with a definition of latency in the context of signal processing applications, and demonstrate that multiple latency values exist for a PGM graph executed with the strong synchrony hypothesis. We then relate these latency values to latency created by scheduling node executions with an RBE-EDF scheduler in an implementation, and show how to use the deadline response parameters associated with each task (node) to control and manage latency.

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 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. Thus, latency is the delay between when a source node produces a sample ($prd(q)$ tokens) and when the graph outputs the processed signal. Moreover, 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. *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

$$Total\ Latency = Inherent\ Latency + Imposed\ Latency.$$

The first step in managing latency in an implementation is to quantify both inherent and imposed latency. This is a difficult task for two reasons. First, PGM graphs frequently change the number of tokens in the data stream. For example, a filter may zero-pad data or interpolate data to smooth out the signal, which adds tokens to the data stream. A filter may also remove tokens from the data stream when a signal is down sampled. Second, the scheduling and execution of nodes in a PGM graph make it hard to correlate the execution of a source node with the corresponding execution of the sink node. For example, consider the SAR graph shown in Figure 4.1. We do not need to understand the processing performed to recognize that the node labeled *Zero Fill* produces 138 more tokens that it consumes each time it executes. In this case the node is adding zeros to the data stream. Nodes may also reduce the number of tokens in the data stream by down sampling the data — using every k^{th} data element. Or an input queue's threshold may be set greater than its consume value, which causes some number of tokens to be used in multiple executions of the node such as the input queue, labeled *RCS*, to the *Corner Turn* node in the SAR graph of Figure 4.1. Many executions of source node *YRange* occur before the first execution of sink node *Output*. If the scheduling algorithm creates additional delays in the execution of nodes, how do we identify which execution of the source node enabled an execution of the sink node when there are multiple executions of the source node before the sink node finally executes?

The strong synchrony hypothesis simplifies the problem of correlating an execution of the source node and the subsequent execution of the sink node. Under the strong synchrony hypothesis, we can simply measure the time (if any) that elapses between the execution of the source and sink nodes. Scheduling does not affect latency under the strong synchrony hypothesis because nodes execute instantaneously. Thus, any latency that occurs is due to the dataflow attributes of the queues or the graph topology. This type of latency is *inherent* in the graph and provides a lower bound on latency in an implementation. If an application's latency requirement exceeds the graph's inherent latency, no implementation of the graph will meet the latency requirement. Either the dataflow attributes of the queues must be changed, or the graph topology must be changed (or both). Section 4.2 shows how to compute the inherent latency in a PGM graph.

Any additional latency created by scheduling and node execution is *imposed* upon the graph by the implementation and can be managed using techniques from real-time scheduling theory. Our goal in building real-time systems from processing graphs is to analytically bound and then manage imposed latency. Section 4.3 presents techniques to achieve this goal for acyclic and cyclic graphs executed with our RBE-EDF scheduler.

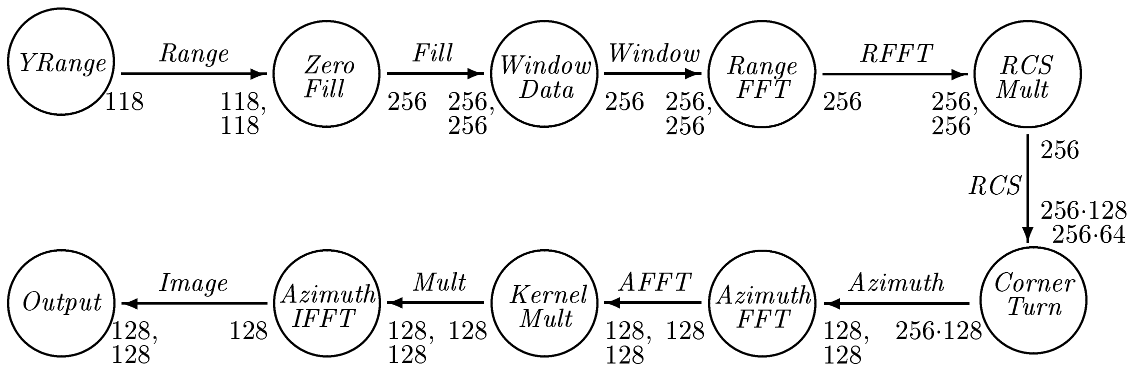


Figure 4.1: The SAR PGM graph. The tail of each queue is annotated with the queue's produce value, and the head of each queue is annotated with the threshold and consume values. For example, the queue labeled *RCS* has $prd(RCS) = 256$, $thr(RCS) = 256 \cdot 128$, and $cns(RCS) = 256 \cdot 64$.

Finally, in Section 4.4, we combine the bounds on inherent latency with the bounds for imposed latency to bound the total latency any sample will encounter in an implementation of a PGM built with our synthesis method.

4.2 Inherent Latency

Inherent latency is the latency created by the topology of the graph and the dataflow attributes for each queue q: $prd(q)$, $cns(q)$, and $thr(q)$. Inherent latency is most recognizable under the strong synchrony hypothesis where processor speed and scheduling do not affect latency since nodes execute instantaneously. 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. When the strong synchrony hypothesis is assumed, 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. In simple dataflow models that require unity dataflow attributes and only allow one source node, the inherent latency of the graph is 0. However, as the following sections demonstrate, non-unity dataflow values or multiple source nodes can create significant latency in processing the signal, even under the strong synchrony hypothesis. Our analysis of inherent latency begins with chains in Section 4.2.1. Inherent latency in acyclic processing graphs is analyzed in Section 4.2.2, and the effect of cycles on inherent latency is described in Section 4.2.3.

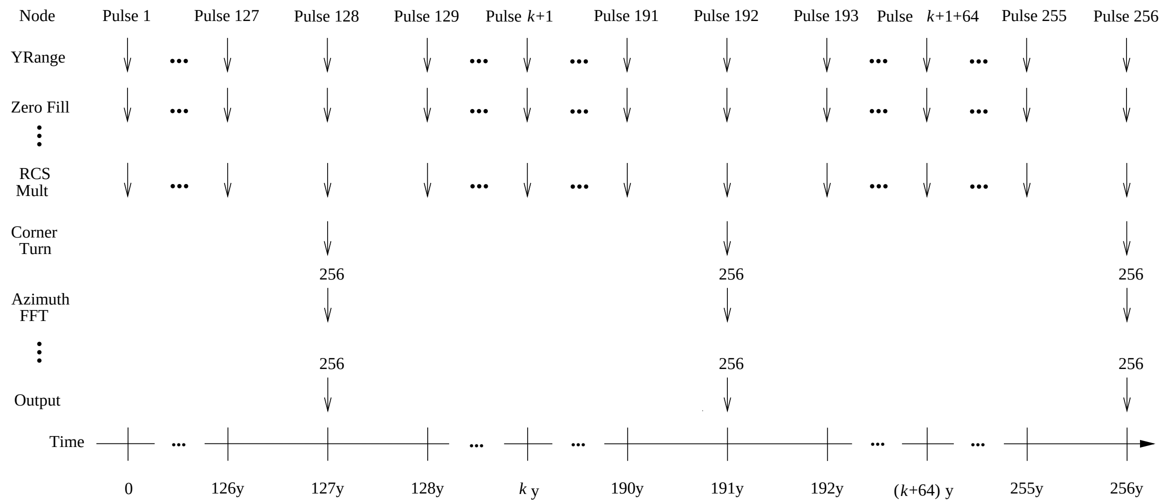


Figure 4.2: A simulation showing latency for the SAR graph under the strong synchrony hypothesis. Each down arrow in Figure 4.2 represents the release and instantaneous execution of a node, and the number 256 above a down arrow means 256 instantaneous executions of a node.

4.2.1 Inherent Latency in Chains of Nodes

The analysis of inherent latency in chains reveals the impact dataflow attributes have on inherent latency. An input signal encounters different amounts of latency depending on the state of the graph (i.e., the number of tokens on each queue of the graph) when a sample arrives. In general, there is no single latency value that all samples encounter. There is, however, a pattern of executions that result in a set of latency values for the input signal.

Consider the SAR graph shown in Figure 4.1. There are 128 unique latency values for this simple graph. A representation of its execution under the strong synchrony hypothesis is shown in Figure 4.2 where it is assumed that the external source, labeled *YRange*, produces one radar pulse of data (or sample) every y time units. Each down arrow in Figure 4.2 represents the release and instantaneous execution of a node, and the number 256 above a down arrow means 256 instantaneous executions of a node. Notice that one pulse from the *YRange* node enables one execution of each of the next four nodes in Figure 4.1 since each queue's threshold, produce, and consume values are equal. However, the processing stops at node *RCS Mult* until node *YRange* produces 128 pulses of data because the input queue to the *Corner Turn* node, queue *RCS*, has a threshold of $256 \cdot 128$ tokens. Thus, since a pulse is delivered every y time units and the last four nodes in the graph only execute after the *Corner Turn* node, the graph has an inherent maximum latency of $127y$. The maximum latency is $127y$ and not $128y$ because the latency a sample encounters begins after the sample's arrival. The inherent maximum latency of $127y$ is reflected in Figure 4.2 by the first execution of the *Corner Turn* node and the 256 simultaneous executions of the rest of the nodes in the graph at time $127y$. The first sample received by the graph always encounters the maximum latency (assuming the queues have no initial data). The next sample produced by the source node, at time y, encounters a latency of $126y$, and the third sample produced encounters a latency of $125y$. The latency each subsequent sample encounters decreases until node *Corner Turn* first executes at time $127y$. Thus, the inherent latency for pulse j, $1 \leq j \leq 128$, is $(128 - j)y$, and the maximum inherent latency any sample encounters is $127y$.

There is, however, another "maximum" latency that is of more interest, and that is the maximum latency that occurs after the first execution of every node in the graph. Since queue *RCS* has a threshold of $256 \cdot 128$ tokens and a consume of $256 \cdot 64$ tokens, whenever the *Corner Turn* node executes, it consumes half of the tokens on queue *RCS* and leaves $256 \cdot 64$ tokens on queue *RCS*. Therefore, source node *YRange* only needs to execute another 64 times before the *Corner Turn* node (and the rest of the nodes in the chain) are eligible for execution again. The 64 executions of the source node before the node *Corner Turn*executes again creates the other (and more interesting) "maximum" inherent latency value of $63y$ (since latency begins after the source node produces). This latency value of $63y$ is more interesting than the latency encountered by samples produced before the first execution of node *Corner Turn* since it is the maximum recurring latency value. In the execution example shown in Figure 4.2 for the SAR graph, a latency of $63y$ is encountered by pulses $128 + 1 + 64k$, $\forall k \geq 0$ (i.e., pulses 129, 193, 257, 321, ...). From the execution pattern shown in Figure 4.2 for the SAR graph, we see that the latency for pulse $j > 128$ is

$$(63 - ((j - 1) \bmod 64))y.$$

Moreover, when each queue q is initialized with $thr(q) - cns(q)$ tokens, the recurring maximum latency value begins with the first sample since, by Theorem 2.3.2, $thr(q) - cns(q)$ tokens is the minimum number of tokens that will ever be on queue q.

The latency a sample encounters under the strong synchrony hypothesis is dependent on the data flow attributes of the graph, the state of the queues (i.e., the number of tokens on each queue of the graph) when the sample arrives, and the execution rate of the graph source node. Lemma 4.2.1 states analytically what these relationships are, and, at any point in time, it also tells us the number of samples that must be produced by node u before node w is eligible for execution. This number is used by Lemma 4.2.2 to compute the inherent latency a sample will encounter when the source is periodic.

Lemma 4.2.1. *Let path $u \rightsquigarrow w$ be a PGM chain, and let*

$$F_{u \rightsquigarrow w} = \begin{cases} \max\left(0, \left\lceil \frac{thr(q) - length(q)}{prd(q)} \right\rceil\right) & \text{if } \exists q : \psi(q) = (u, w) \\ \max\left(0, \left\lceil \frac{(F_{v \rightsquigarrow w} - 1) \cdot cns(q) + thr(q) - length(q)}{prd(q)} \right\rceil\right) & \text{if } \exists q : \psi(q) = (u, v) \wedge v \neq w \\ & \quad \wedge\ F_{v \rightsquigarrow w} > 0 \\ 0 & \text{if } \exists q : \psi(q) = (u, v) \wedge v \neq w \\ & \quad \wedge\ F_{v \rightsquigarrow w} = 0 \end{cases} \tag{4.1}$$

Node u must execute $F_{u \rightsquigarrow w}$ times to produce enough tokens in order to put the input queue to node w over threshold.

Equation (4.1) defines a recursive function that determines the number of times node u must execute before the input queue to node w is over threshold. The first branch of the function handles a path of length one where node u is attached to node w. For example, consider the chain *Azimuth IFFT* $\rightsquigarrow$ *Output* in the SAR graph of Figure 4.1 whose length is one. Assuming $length(Image) = 0$, node *Azimuth IFFT* must execute

$$\begin{aligned} F_{AzimuthIFFT \rightsquigarrow Output} &= \max\left(0, \left\lceil \frac{thr(Image) - length(Image)}{prd(Image)} \right\rceil\right) \\ &= \left\lceil \frac{128 - 0}{128} \right\rceil = 1 \end{aligned}$$

time before sink node *Output* executes.

The second branch of the function $F_{u \rightsquigarrow w}$ recursively references itself when applied to a path whose length is reduced by one (until the path is of length one). Thus, by recursively invoking $F_{u \rightsquigarrow w}$, the second branch returns the number of times the current source node u must execute in order for the node attached to it, node v, to execute $F_{v \rightsquigarrow w}$ times (which is the number of times node v must execute in order to put the input queue to node w over threshold). For example, let $length(RCS) = 256 \cdot 100$ and $length(q) = 0$

for the rest of the queues in the graph. Node *RCS Mult* must execute

$$F_{RCSMult \rightsquigarrow Output} = \max\left(0, \left\lceil \frac{(F_{CornerTurn \rightsquigarrow Output} - 1) \cdot cns(RCS) + thr(RCS) - length(RCS)}{prd(RCS)} \right\rceil\right)$$

times before sink node *Output* executes. Since $F_{CornerTurn \rightsquigarrow Output} = 1$, the total number of times node *RCS Mult* must execute is

$$\begin{aligned} \left\lceil \frac{(1-1) \cdot cns(RCS) + thr(RCS) - length(RCS)}{prd(RCS)} \right\rceil \\ &= \left\lceil \frac{(1-1) \cdot (256 \cdot 64) + (256 \cdot 128) - (256 \cdot 100)}{256} \right\rceil \\ &= \left\lceil \frac{(256 \cdot 28)}{256} \right\rceil \\ &= 28. \end{aligned}$$

The third branch of the function $F_{u \rightsquigarrow w}$ returns zero when the input queue to node w is already over threshold, or when other queues in the chain have enough data that the input queue to node w will go over threshold without node u executing again.

Proof of Lemma 4.2.1: We prove, by induction on the length of the path from node u to node w, that $F_{u \rightsquigarrow w}$, defined in Equation (4.1), yields the number of times node u must execute in order to put the input queue to node w over threshold. Consider the chain in Figure 4.3 where the nodes and queues in path $u \rightsquigarrow w$ have been relabeled with sequential indices starting at 0 and ending at j so that node u is labeled n_0 and node w is labeled n_j. Hence, the nodes in path $u \rightsquigarrow w$ (of length j) are labeled $(n_0, n_1, n_2, \ldots, n_j)$ and the output queue for node n_i labeled q_i.

For the base case of the induction, consider the path from node n_{j-1} to node n_j whose length is one since $j - (j-1) = 1$. By Theorem 2.3.4, node n_{j-1} must execute

$$\max\left(0, \left\lceil \frac{thr(q_{j-1}) - length(q_{j-1})}{prd(q_{j-1})} \right\rceil\right)$$

times before the input queue to node n_j is over threshold. This is the same value returned by the first branch of $F_{u \rightsquigarrow w}$ in Equation (4.1). Thus, $F_{u \rightsquigarrow w}$ returns the correct number of executions required of node u before for the input queue to node w is over threshold when the length of the path is one.

By the induction hypothesis, assume Lemma 4.2.1 holds for all paths of length i :

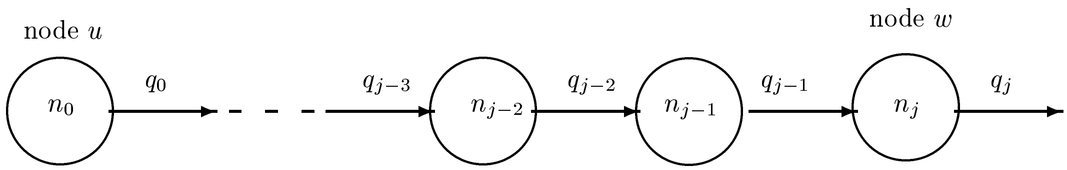


Figure 4.3: The path $u \rightsquigarrow w$ is relabeled $(n_0, n_1, n_2, \ldots, n_j)$ so that node u is labeled n_0 and node w is labeled n_j with the respective output queues labeled q_0 and q_j.

$1 \leq i \leq k$, where $k < j$. Thus, for a path of length i, $F_{n_{j-i} \rightsquigarrow n_j}$ yields the number of times node n_{j-i} must execute before the input queue to node n_j is over threshold. Let $i = k+1$. By the induction hypothesis, node n_{j-k} must execute $F_{n_{j-k} \rightsquigarrow n_j}$ times before the input queue to node n_j is over threshold. Therefore, to prove Lemma 4.2.1, we need only determine how many times node n_{j-i} must execute in order for node n_{j-k} to execute $F_{n_{j-k} \rightsquigarrow n_j}$ times.

If $F_{n_{j-k} \rightsquigarrow n_j} = 0$, then no executions of node n_{j-i} are required. Lemma 4.2.1 holds in this case because the third branch of Equation (4.1) will return 0 when $F_{n_{j-k} \rightsquigarrow n_j} = 0$.

If $F_{n_{j-k} \rightsquigarrow n_j} > 0$ and

$$\mathit{length}(q_{j-i}) \geq ((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot \mathit{cns}(q_{j-i})) + \mathit{thr}(q_{j-i}),$$

then, by Theorem 2.3.3, node n_{j-k} will execute at least

$$\begin{aligned}
&\left\lfloor \frac{\mathit{length}(q_{j-i}) - \mathit{thr}(q_{j-i})}{\mathit{cns}(q_{j-i})} \right\rfloor + 1 \\
&\qquad = \left\lfloor \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot \mathit{cns}(q_{j-i})) + \mathit{thr}(q_{j-i}) - \mathit{thr}(q_{j-i})}{\mathit{cns}(q_{j-i})} \right\rfloor + 1 \\
&\qquad = \left\lfloor \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot \mathit{cns}(q_{j-i}))}{\mathit{cns}(q_{j-i})} \right\rfloor + 1 \\
&\qquad = (F_{n_{j-k} \rightsquigarrow n_j} - 1) + 1 \\
&\qquad = F_{n_{j-k} \rightsquigarrow n_j}
\end{aligned}$$

times, and no more executions of node n_{j-i} are required. In this case, the second branch of Equation (4.1) evaluates to

$$\max\left(0, \left\lceil \frac{(F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot \mathit{cns}(q_{j-i}) + \mathit{thr}(q_{j-i}) - \mathit{length}(q_{j-i})}{\mathit{prd}(q_{j-i})} \right\rceil \right) = 0$$

since

$$\left\lceil \frac{(F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i}) + thr(q_{j-i}) - length(q_{j-i})}{prd(q_{j-i})} \right\rceil \leq 0$$

when

$$length(q_{j-i}) \geq ((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i})) + thr(q_{j-i}),$$

and Lemma 4.2.1 holds.

If $F_{n_{j-k} \rightsquigarrow n_j} > 0$ and

$$length(q_{j-i}) < ((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i})) + thr(q_{j-i}),$$

then

$$F_{n_{j-i} \rightsquigarrow n_j} = \left\lceil \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i})) + thr(q_{j-i}) - length(q_{j-i})}{prd(q_{j-i})} \right\rceil$$

executions of node n_{j-i} will produce

$$\left\lceil \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i})) + thr(q_{j-i}) - length(q_{j-i})}{prd(q_{j-i})} \right\rceil \cdot prd(q_{j-i})$$

tokens on queue q_{j-i}, and (before any executions of node n_{j-k} occur) the length of queue q_{j-i} will be

$$\left\lceil \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i})) + thr(q_{j-i}) - length(q_{j-i})}{prd(q_{j-i})} \right\rceil \cdot prd(q_{j-i}) + length(q_{j-i}).$$

By Theorem 2.3.3, given

$$\begin{aligned}
&\left\lceil \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i})) + thr(q_{j-i}) - length(q_{j-i})}{prd(q_{j-i})} \right\rceil \cdot prd(q_{j-i}) + length(q_{j-i}) \\
&\qquad \geq \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i})) + thr(q_{j-i}) - length(q_{j-i})}{prd(q_{j-i})} \cdot prd(q_{j-i}) + length(q_{j-i}) \\
&\qquad = ((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i})) + thr(q_{j-i}) - length(q_{j-i}) + length(q_{j-i}) \\
&\qquad = ((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i})) + thr(q_{j-i})
\end{aligned}$$

tokens on queue q_{j-i}, node n_{j-k} will execute

$$
\begin{aligned}
\left\lfloor \frac{length(q_{j-i}) - thr(q_{j-i})}{cns(q_{j-i})} \right\rfloor + 1 & \\
&\geq \left\lfloor \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i})) + thr(q_{j-i}) - thr(q_{j-i})}{cns(q_{j-i})} \right\rfloor + 1 \\
&= \left\lfloor \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i}))}{cns(q_{j-i})} \right\rfloor + 1 \\
&= (F_{n_{j-k} \rightsquigarrow n_j} - 1) + 1 \\
&= F_{n_{j-k} \rightsquigarrow n_j}
\end{aligned}
$$

times. Moreover, if node n_{j-i} executes any fewer times, say

$$F_{n_{j-i} \rightsquigarrow n_j} - 1 = \left\lceil \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i})) + thr(q_{j-i}) - length(q_{j-i})}{prd(q_{j-i})} \right\rceil - 1$$

times, it will produce

$$\left(\left\lceil \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i})) + thr(q_{j-i}) - length(q_{j-i})}{prd(q_{j-i})} \right\rceil - 1 \right) \cdot prd(q_{j-i})$$

tokens on queue q_{j-i}, and (before any executions of node n_{j-k} occur) the length of queue q_{j-i} will be

$$\left(\left\lceil \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot cns(q_{j-i})) + thr(q_{j-i}) - length(q_{j-i})}{prd(q_{j-i})} \right\rceil - 1 \right) \cdot prd(q_{j-i}) + length(q_{j-i}).$$

Since, for positive integers a and b,

$$
\begin{aligned}
\left(\left\lceil \frac{a}{b} \right\rceil - 1 \right) \cdot b &= \left\lceil \frac{a}{b} \right\rceil \cdot b - b \\
&\leq a + (b - 1) - b \\
&= a - 1,
\end{aligned}
$$

the number of tokens on queue q_{j-i} is

$$\left(\left\lceil \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot \mathit{cns}(q_{j-i})) + \mathit{thr}(q_{j-i}) - \mathit{length}(q_{j-i})}{\mathit{prd}(q_{j-i})} \right\rceil - 1\right) \cdot \mathit{prd}(q_{j-i}) + \mathit{length}(q_{j-i})$$
$$\leq ((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot \mathit{cns}(q_{j-i})) + \mathit{thr}(q_{j-i}) - \mathit{length}(q_{j-i}) - 1 + \mathit{length}(q_{j-i})$$
$$= ((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot \mathit{cns}(q_{j-i})) + \mathit{thr}(q_{j-i}) - 1$$

By Theorem 2.3.3, given $((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot \mathit{cns}(q_{j-i})) + \mathit{thr}(q_{j-i}) - 1$ tokens on q_{j-i}, node n_{j-k} will execute

$$\left\lfloor \frac{\mathit{length}(q_{j-i}) - \mathit{thr}(q_{j-i})}{\mathit{cns}(q_{j-i})} \right\rfloor + 1$$
$$= \left\lfloor \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot \mathit{cns}(q_{j-i})) + \mathit{thr}(q_{j-i}) - 1 - \mathit{thr}(q_{j-i})}{\mathit{cns}(q_{j-i})} \right\rfloor + 1$$
$$= \left\lfloor \frac{((F_{n_{j-k} \rightsquigarrow n_j} - 1) \cdot \mathit{cns}(q_{j-i})) - 1}{\mathit{cns}(q_{j-i})} \right\rfloor + 1$$
$$= \left\lfloor \frac{-1}{\mathit{cns}(q_{j-i})} \right\rfloor + (F_{n_{j-k} \rightsquigarrow n_j} - 1) + 1$$
$$= -1 + (F_{n_{j-k} \rightsquigarrow n_j} - 1) + 1$$
$$= F_{n_{j-k} \rightsquigarrow n_j} - 1$$

times (which is one less than the number of executions required). Thus, node n_{j-i} must execute $F_{n_{j-i} \rightsquigarrow n_j}$ times to produce enough tokens to put the input queue to node w over threshold, and Lemma 4.2.1 holds for all paths of length i, $1 \leq i \leq j$. $\square$

We now know the number of executions of source node u that are required before node w in the path $u \rightsquigarrow w$ can execute. To get inherent latency bounds we need to consider how long it takes before these executions occur. This will depend on the rate at which source node u executes. Most external sensors produce data on a periodic basis, but some execute with a rate of x executions in any interval of y time units. If the chain executes on a single processor and the external source is periodic, we can derive precise latency values for each sample. If the source node is rate-based (i.e., it executes with a rate of x executions in any interval of y time units), exact latency values cannot be derived unless it is known exactly when each of the x executions occur in each interval of length y. Without this information, however, upper and lower bounds for the inherent latency a sample encounters can still be derived. We first derive inherent latency values for chains with periodic source nodes (i.e., nodes with execution rates where it is known exactly when each execution occurs). We then relax this restriction to include general rate-based sources.

4.2.1.1 A Periodic Source Node

Let $u \rightsquigarrow w$ be a PGM chain such that $u \in \mathcal{I}$ is a periodic source node with period y_u and $w \in \mathcal{O}$ is a sink node. Evaluating $F_{u \rightsquigarrow w}$ just before a sample arrives will tell us how many more samples are required before the input queue to node w is over threshold. Thus, the inherent latency a sample encounters is given by

$$\max\left(0, (F_{u \rightsquigarrow w} - 1) \cdot y_u\right).$$

We again subtract one from $F_{u \rightsquigarrow w}$ before converting it to time units since the latency interval begins after the sample arrives. For example, consider again the SAR graph of Figure 4.1 on Page 113. Let source node *YRange* execute with period y. As before, let $length(RCS) = 256 \cdot 100$, and let $length(q) = 0$ for the rest of the queues in the graph just before the 101^{st} sample is produced. Source node *YRange* must execute $F_{YRange \rightsquigarrow Output} = 28$ times before queue $Image$ is over threshold. Thus, the 101^{st} sample will encounter an inherent latency of

$$(F_{YRange \rightsquigarrow Output} - 1) \cdot y = (28 - 1) \cdot y = 27y$$

time units. This inherent latency value is also reflected in Figure 4.2 on page 114, where the 101^{st} sample arrives at time $100y$ and node *Output* first executes at time $127y$. Thus, as computed here, the sample has an inherent latency of $27y = 127y - 100y$ time units.

Inherent latency analysis for a PGM chain with a periodic source is formalized as follows.

Lemma 4.2.2. *Let $u \rightsquigarrow w$ be a PGM chain such that $u \in \mathcal{I}$ is a periodic source node with period y_u and $w \in \mathcal{O}$ is a sink node. Under the strong synchrony hypothesis, the latency a sample encounters is*

$$\max\left(0, (F_{u \rightsquigarrow w} - 1) \cdot y_u\right) \tag{4.2}$$

Proof: By Lemma 4.2.1, $F_{u \rightsquigarrow w}$ executions of source node u are required before sink node w is eligible for execution. If $F_{u \rightsquigarrow w} = 0$, the sample's inherent latency is 0, and Equation (4.2) returns 0 as desired. If $F_{u \rightsquigarrow w} = 1$, the next sample will encounter an inherent latency of 0 since sink node w will execute as soon as the sample arrives. In this case

$(F_{u \rightsquigarrow w} - 1) \cdot y_u = 0$ as desired. If $F_{u \rightsquigarrow w} > 1$, the next sample will encounter an inherent latency of $(F_{u \rightsquigarrow w} - 1) \cdot y_u$ time units since $(F_{u \rightsquigarrow w} - 1)$ additional executions of source node u are required after the sample arrives before sink node w executes. Therefore, under the strong synchrony hypothesis, if source node u has a period of y_u, a sample's latency will be $(F_{u \rightsquigarrow w} - 1) \cdot y_u$ time units. □

Using Lemma 4.2.2, we compute the inherent latency of the first pulse received by the SAR graph to be $127y$. This matches the simulation in Figure 4.2. Recall from Lemma 2.3.2 that the most tokens queue q can hold without being over threshold is $MaxUnderThr(q)$ and the minimum possible number of tokens on queue q after both producer and consumer nodes have fired at least once is $MinTokens(q)$. When all of the queues in the graph contain $MaxUnderThr(q)$ tokens, as is the case after pulses $127 + 64k$, $\forall k \geq 0$, in Figure 4.2, the next sample's inherent latency will be 0 as shown in Figure 4.2 for pulses $128 + 64k$, $\forall k \geq 0$. Evaluating Equation (4.2) when each queue in the SAR graph contains $MinTokens(q)$ tokens, we get an inherent latency of $63y$ as shown in Figure 4.2 for pulses 129 and 193.

4.2.1.2 A Rate-Based Source Node

If a source node is rate-based rather than periodic, evaluating $F_{u \rightsquigarrow w}$ just before a sample arrives will still tell us how many more samples are required before the input queue to node w is over threshold. However, if $x_u > 1$ and $F_{u \rightsquigarrow w} > 1$, we can no longer convert this number directly to an inherent latency value. We can compute the interval in which the ${F_{u \rightsquigarrow w}}^{th}$ execution of source node u occurs, but the exact time of the ${F_{u \rightsquigarrow w}}^{th}$ execution of node u cannot be computed. This is because we do not know when, in an interval of length y_u, the x_u executions of node u occur. Thus, for rate-based source nodes, the inherent latency a sample encounters is bounded by an interval.

Consider the SAR graph of Figure 4.1 on page 113 once again. Let source node *YRange* execute with a well-defined execution rate of $R_{YRange} = (3, y)$ starting at time 0 (i.e., *YRange* executes three times every y time units). As before, let $length(RCS) = 256 \cdot 100$, and let $length(q) = 0$ for the rest of the queues in the graph at a time t just before the 101^{st} sample is produced. Source node *YRange* must execute another $F_{YRange \rightsquigarrow Output} = 28$ times before queue $Image$ is over threshold. Since exactly three executions of source node *YRange* occur in any interval of length y, there will be exactly three executions of source node *YRange* in each interval $[0 + (k-1)y, 0 + ky), \forall k > 0$.

Thus, the next 28 executions of node *YRange* will occur *before* time

$$\begin{aligned} t + \left\lceil \frac{F_{YRange \rightsquigarrow Output}}{x_{YRange}} \right\rceil \cdot y_{YRange} &= t + \left\lceil \frac{28}{3} \right\rceil \cdot y \\ &= t + 10y. \end{aligned}$$

The next 28 executions occur before time $t+10y$ because time $t+10y$ is the beginning of the next rate interval. Therefore, the inherent latency encountered by the 101^{st} sample is less than

$$\begin{aligned} \left\lceil \frac{F_{YRange \rightsquigarrow Output}}{x_{YRange}} \right\rceil \cdot y_{YRange} &= \left\lceil \frac{28}{3} \right\rceil \cdot y \\ &= 10y. \end{aligned}$$

The last of the next 28 executions of node *YRange* will occur no earlier than time

$$\begin{aligned} t + \left\lfloor \frac{F_{YRange \rightsquigarrow Output} - 1}{x_{YRange}} \right\rfloor \cdot y_{YRange} &= t + \left\lfloor \frac{28-1}{3} \right\rfloor \cdot y = 9 \cdot y \\ &= t + 9y. \end{aligned}$$

Therefore, the inherent latency of the next sample will be greater than or equal to

$$\begin{aligned} \left\lfloor \frac{F_{YRange \rightsquigarrow Output} - 1}{x_{YRange}} \right\rfloor \cdot y_{YRange} &= \left\lfloor \frac{27}{3} \right\rfloor \cdot y \\ &= 9y. \end{aligned}$$

Thus, the inherent latency the 101^{st} sample encounters is bounded such that

$$9y \leq \text{Sample Latency} < 10y.$$

We have informally argued that the inherent latency a sample encounters is bounded by an interval when the source node in a chain executes with a rate. We now formalize the computation of inherent latency for samples produced by rate-based source nodes.

Lemma 4.2.3. *Let $u \rightsquigarrow w$ be a PGM chain such that $u \in \mathcal{I}$ is a source node with a well-defined execution rate $R_u = (x_u, y_u)$ and $w \in \mathcal{O}$ is a sink node. Under the strong synchrony hypothesis, the latency a sample will encounter is bounded such that*

$$\max\left(0, \left\lfloor \frac{F_{u \rightsquigarrow w} - 1}{x_u} \right\rfloor \cdot y_u\right) \leq \textit{ Sample Latency } < \max\left(1, \left\lceil \frac{F_{u \rightsquigarrow w}}{x_u} \right\rceil \cdot y_u\right) \tag{4.3}$$

Proof: By Lemma 4.2.1, $F_{u\rightsquigarrow w}$ executions are required of source node u before sink node w is eligible for execution. If $F_{u\rightsquigarrow w} = 0$, the sample's inherent latency is 0, and the lower bound of Expression (4.3) returns 0 as desired. In this case, the upper bound of Expression (4.3) returns one, and the inherent latency is bounded such that

$$0 \leq \text{ Sample Latency } < 1$$

as desired. If $F_{u\rightsquigarrow w} = 1$, the next sample will encounter a latency of 0 since sink node w will execute as soon as the sample arrives. In this case, Expression (4.3) evaluates to

$$\begin{aligned} \max\left(0, \left\lfloor \frac{F_{u\rightsquigarrow w} - 1}{x_u} \right\rfloor \cdot y_u\right) \leq \textit{Sample Latency } &< \max\left(1, \left\lceil \frac{F_{u\rightsquigarrow w}}{x_u} \right\rceil \cdot y_u\right) \\ \max\left(0, \left\lfloor \frac{1-1}{x_u} \right\rfloor \cdot y_u\right) \leq \textit{Sample Latency } &< \max\left(1, \left\lceil \frac{1}{x_u} \right\rceil \cdot y_u\right) \\ 0 \leq \textit{Sample Latency } &< y_u, \end{aligned}$$

which is the tightest bound possible without knowing when in the interval the sample will be produced.

Let $F_{u\rightsquigarrow w} > 1$. By Theorem 2.4.8, since node u executes x_u times in intervals $[t, t+y_u)$, $\forall t \geq 0$, node u will execute $\left\lceil \frac{F_{u\rightsquigarrow w}}{x_u} \right\rceil \cdot x_u$ times in intervals $[t, t + \left\lceil \frac{F_{u\rightsquigarrow w}}{x_u} \right\rceil \cdot y_u)$, $\forall t \geq 0$. Thus, since

$$F_{u\rightsquigarrow w} \leq \left\lceil \frac{F_{u\rightsquigarrow w}}{x_u} \right\rceil \cdot x_u,$$

the inherent latency the next sample will encounter is less than $\left\lceil \frac{F_{u\rightsquigarrow w}}{x_u} \right\rceil \cdot y_u$, and the upper bound of Expression (4.3) holds when $F_{u\rightsquigarrow w} > 1$.

Since

$$F_{u\rightsquigarrow w} - 1 \geq \left\lfloor \frac{F_{u\rightsquigarrow w} - 1}{x_u} \right\rfloor \cdot x_u$$

and the measurement of latency begins *after* a sample arrives, a sample's inherent latency will be greater than or equal to $\left\lfloor \frac{F_{u\rightsquigarrow w}-1}{x_u} \right\rfloor \cdot y_u$. Moreover, these bounds are tight since, if x_u does not divide $F_{u\rightsquigarrow w}$,

$$\left\lfloor \frac{F_{u\rightsquigarrow w} - 1}{x_u} \right\rfloor = \left\lfloor \frac{F_{u\rightsquigarrow w}}{x_u} \right\rfloor$$

and, when $x_u | F_{u \rightsquigarrow w}$,

$$\left\lfloor \frac{F_{u \rightsquigarrow w} - 1}{x_u} \right\rfloor = \left\lfloor \frac{F_{u \rightsquigarrow w}}{x_u} \right\rfloor - 1 = \left\lceil \frac{F_{u \rightsquigarrow w}}{x_u} \right\rceil - 1.$$

Thus, under the strong synchrony hypothesis, the latency a sample will encounter is bounded such that

$$\max\left(0, \left\lfloor \frac{F_{u \rightsquigarrow w} - 1}{x_u} \right\rfloor \cdot y_u\right) \leq \textit{ Sample Latency } < \max\left(1, \left\lceil \frac{F_{u \rightsquigarrow w}}{x_u} \right\rceil \cdot y_u\right)$$

□

Note that when source node u is periodic with period y_u, it executes with rate $R_u = (1, y_u)$, and either Equation (4.2) or Equation (4.3) can be used to derive the inherent latency a sample will encounter since

$$\begin{aligned}
\max\left(0, \left\lfloor \frac{F_{u \rightsquigarrow w} - 1}{x_u} \right\rfloor \cdot y_u\right) &\leq \textit{ Sample Latency } < \max\left(1, \left\lceil \frac{F_{u \rightsquigarrow w}}{x_u} \right\rceil \cdot y_u\right) \\
\max\left(0, \left\lfloor \frac{F_{u \rightsquigarrow w} - 1}{1} \right\rfloor \cdot y_u\right) &\leq \textit{ Sample Latency } < \max\left(1, \left\lceil \frac{F_{u \rightsquigarrow w}}{1} \right\rceil \cdot y_u\right) \text{ for } x_u = 1 \\
\max\left(0, (F_{u \rightsquigarrow w} - 1) \cdot y_u\right) &\leq \textit{ Sample Latency } < \max\left(1, (F_{u \rightsquigarrow w}) \cdot y_u\right).
\end{aligned}$$

Thus, $\textit{ Sample Latency } = \max\left(0, (F_{u \rightsquigarrow w} - 1) \cdot y_u\right)$ when samples are produced once each period of length y_u.

4.2.2 Inherent Latency in Acyclic Graphs

Equation (4.3) quantifies the impact dataflow attributes have on inherent latency for a path in the graph from a source node to a sink node. We now extend the latency analysis to include acyclic graphs, and study the effects graph topology has on inherent latency. We begin the analysis by considering an acyclic graph with a single periodic source. We then gradually relax this restriction until the analysis supports multiple rate-based source nodes in an acyclic graph.

4.2.2.1 One Periodic Source With Multiple Paths

For a PGM chain $u \rightsquigarrow w$, where $u \in \mathcal{I}$ is a periodic source node producing data for node w, a sample's inherent latency is $(F_{u \rightsquigarrow w} - 1) \cdot y_u$ (as shown in Section 4.2.1). When the graph is not a chain, determining inherent latency is more difficult. Consider a graph

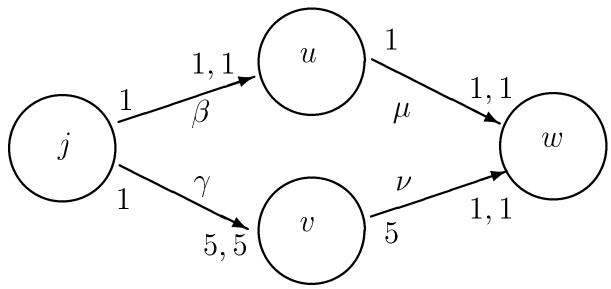
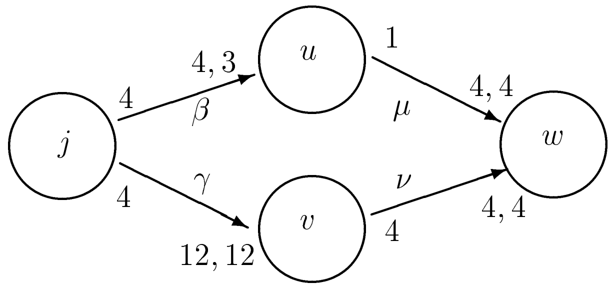


Figure 4.4: A graph with two paths from source node j to node w.

that contains two paths from a source node to node w, such as the graph in Figure 4.4. We now make a change in notation and identify source nodes by the letters i and j rather than the letter u since we will soon be dealing with multiple source nodes. For each path from a source to node w, we can use Lemma 4.2.1 to determine the number of times the source node needs to execute before the input queue to node w along the path under consideration is over threshold. For example, let $R_j = (1, y)$. Evaluating $(F_{j \rightsquigarrow w} - 1) \cdot y$ over the path (j, u, w) yields zero inherent latency since $F_{j \rightsquigarrow w} = 1$, but for path (j, v, w), when $\mathit{length}(\gamma) = 0$ and $\mathit{length}(\nu) = 0$ at the time of evaluation, we get a latency of $4y$ since $F_{j \rightsquigarrow w} = 5$.

Let $\mathcal{F}_{j \rightsquigarrow w} = \max(\{F_p \mid p \in j \rightsquigarrow w\})$ denote the maximum number of times source node j must execute before all of the input queues to node w in the set of paths $\{j \rightsquigarrow w\}$ are over threshold. The function F_p represents a change in notation from $F_{j \rightsquigarrow w}$ to distinguish between multiple paths from a source to node w. F_p is the same function as $F_{i \rightsquigarrow w}$ when $p = i \rightsquigarrow w$ is a chain. Hence, as before, F_p is the number of executions required of the source node in path p before the sink node in path p is eligible for execution when p is a chain. Since $F_p = 0$ for the path $p = (j, u, w)$, and $F_p = 5$ for the path $p = (j, v, w)$, $\mathcal{F}_{j \rightsquigarrow w} = 5$ for the graph in Figure 4.4. Since node w can only execute when both input queues are over threshold, $\mathcal{F}_{j \rightsquigarrow w}$ gives the number of times source node j must execute before all of the input queues to node w are over threshold. Thus, the latency of a sample is determined by $\mathcal{F}_{j \rightsquigarrow w}$ and the execution rate of source node j. Therefore if R_j = (1,y), the first sample produced by source node j will encounter a inherent latency of $(\mathcal{F}_{j \rightsquigarrow w} - 1) \cdot y = 4y$ time units.

Now consider the graph of Figure 4.5. This graph has the same topology of the graph in Figure 4.4, but the dataflow attributes are different. In particular, notice that $\mathit{thr}(\beta) > \mathit{cns}(\beta)$. Let node j be periodic with rate $R_j = (1, 2)$. At time 0, $F_{(j,v,w)} = 3$,



Time	**Execution of**	***length***(β)	***length***(γ)	***length***(μ)	***length***(ν)
0	j, u	1	4	1	0
1	—	1	4	1	0
2	j, u	2	8	2	0
3	—	2	8	2	0
4	j, u, v	3	0	3	4
5	—	3	0	3	4
6	j, u, u, w	1	4	1	0

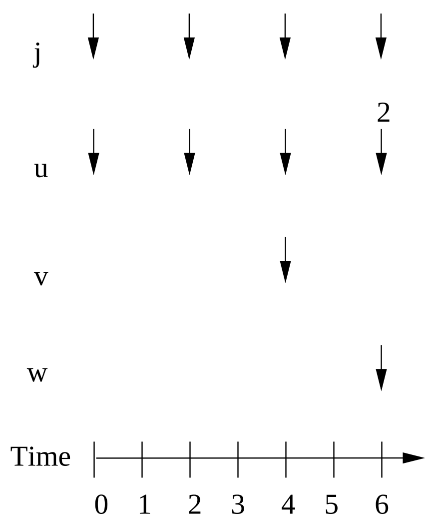


Figure 4.5: A graph with two paths from source node j to node w, a snapshot sequence, and a time-line execution. This graph has the same topology as the graph of Figure 4.4, but the dataflow attributes are different. The most distinguishing features of this graph are the attributes of queue β: $prd(\beta) = 4$, $thr(\beta) = 4$, and $cns(\beta) = 3$. The number two above the down arrow at time six for node u represents two executions of node u at that time.

but $F_{(j,u,w)} = 4$. Hence at time 0, $\mathcal{F}_{j \rightsquigarrow w} = 4$, and the latency of the first sample is

$$\begin{aligned}(\mathcal{F}_{j \rightsquigarrow w} - 1) \cdot y_j &= (\mathcal{F}_{j \rightsquigarrow w} - 1) \cdot 2 \\ &= (4 - 1) \cdot 2 \\ &= 6\end{aligned}$$

time units. It is instructive to step through the first 6 time units of graph execution under the strong synchrony hypothesis. As shown in the snapshot sequence and timeline execution in Figure 4.5, node j appends 4 tokens to queues β and γ at time 0. This enables one execution of node u, which consumes 3 of the 4 tokens on queue β and produces one token on queue μ. At time 2, 4 more tokens are appended to queues β and γ and node u executes once again. We now have $\mathit{length}(\beta) = 2$, $\mathit{length}(\mu) = 2$, and $\mathit{length}(\gamma) = 8$. The next production of data by node j, at time 4, enables both u and v. Even though the execution of node v puts queue ν over threshold, node w is not able to execute since $\mathit{length}(\mu) = 3 < \mathit{thr}(\mu)$. At time 6 (with $\mathit{length}(\beta) = 3$), node j appends 4 tokens to queues β and γ. This enables 2 executions of node u and (finally) an execution of node w. The latency of the sample produced at time 0 was indeed 6, as we computed above.

Inherent latency analysis for an acyclic graph with one periodic source is formalized as follows.

Lemma 4.2.4. *Let $\{j \rightsquigarrow w\}$ be the set of acyclic paths from node j to node w. Node j must execute*

$$\mathcal{F}_{j \rightsquigarrow w} = \max(\{F_p \mid p \in j \rightsquigarrow w\}) \tag{4.4}$$

times to produce enough tokens to put each of the input queues to node w in the set of paths $\{j \rightsquigarrow w\}$ over threshold.

Proof: By Lemma 4.2.1, evaluating F_p for a path p in the set $\{j \rightsquigarrow w\}$ gives the number of executions of node j required to produce enough tokens to put the input queue to node w over threshold if the path p were a chain. Thus, after

$$\mathcal{F}_{j \rightsquigarrow w} = \max(\{F_p \mid p \in j \rightsquigarrow w\})$$

executions of node j, all paths from node j to node w will have produced enough tokens to put each of the input queues to node w in the set of paths $\{j \rightsquigarrow w\}$ over threshold. $\square$

Lemma 4.2.5. *Let $G = (V, E, \psi)$ be an acyclic PGM graph with one periodic source node $j \in \mathcal{I}$ with period y_j. Let $\{j \rightsquigarrow w\}$ be the set of acyclic paths from periodic source node j to sink node $w \in \mathcal{O}$. Under the strong synchrony hypothesis, the latency a sample will encounter from source node j to sink node w is*

$$\max\left(0, (\mathcal{F}_{j \rightsquigarrow w} - 1) \cdot y_j\right) \tag{4.5}$$

Proof: By Lemma 4.2.4, evaluating $\mathcal{F}_{j \rightsquigarrow w}$ just before a sample arrives will tell us how many additional samples are required before each of the input queues to node w in the set of paths $\{j \rightsquigarrow w\}$ over threshold. Since node j is the only source node, all of the input queues to node w will be over threshold after $\mathcal{F}_{j \rightsquigarrow w}$ executions of node j, and node w will execute. Thus, since source node j is periodic, a sample will encounter a latency of

$$\max(0, (\mathcal{F}_{j \rightsquigarrow w} - 1) \cdot y_j)$$

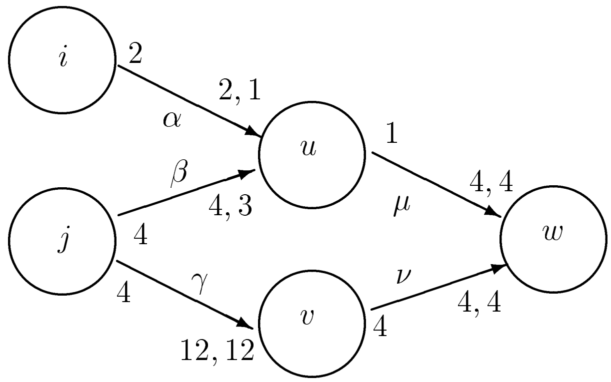
time units. We subtract one from $\mathcal{F}_{j \rightsquigarrow w}$ before converting it to time units since the latency interval begins after the sample arrives. □

Lemma 4.2.5 can be used to determine the latency a sample will encounter from source node j to sink node w if the graph has only one periodic source node. We now relax the restriction of a single periodic source node and consider multiple periodic source nodes.

4.2.2.2 Multiple Periodic Sources Nodes

Let $\mathcal{I}_w$ be the set of source nodes for which there exists a path from $j \in \mathcal{I}_w$ to node w. When more than one source node produces data for a node w, such as in the graph of Figure 4.6, we need to evaluate $(\mathcal{F}_{j \rightsquigarrow w} - 1) \cdot y_j$ for each node $j \in \mathcal{I}_w$ and use the maximum of these values to compute inherent latency. For example, let $R_i = (1, 3)$ and $R_j = (1, 2)$ for the source nodes in Figure 4.6. From the time-line execution, we see that the first sample has a latency of six time units. Evaluating (4.4) at time 0 (with $\mathit{length}(q) = 0$ for all q), we have $(\mathcal{F}_{i \rightsquigarrow w} - 1) \cdot y_i = 6$ and $(\mathcal{F}_{j \rightsquigarrow w} - 1) \cdot y_j = 6$. Therefore, assuming nodes i and j both produce data at time 0, the first sample produced by node i (which consists of two tokens) and the first sample produced by node j (which consists of four tokens) will each have an inherent latency of six time units, which is the same latency we observed in the time-line execution.

What will the latency be for the second sample produced by node j at time 2? From the time-line execution, we see that the second sample has an inherent latency of



Time	Execution of	***length***(α)	***length***(β)	***length***(γ)	***length***(μ)	***length***(ν)
0	i, j, u	1	1	4	1	0
1	—	1	1	4	1	0
2	j	1	5	8	1	0
3	i, u	2	2	8	2	0
4	j, u, v	1	3	0	3	4
5	—	1	3	0	3	4
6	i, j, u, u, w	1	1	4	1	0

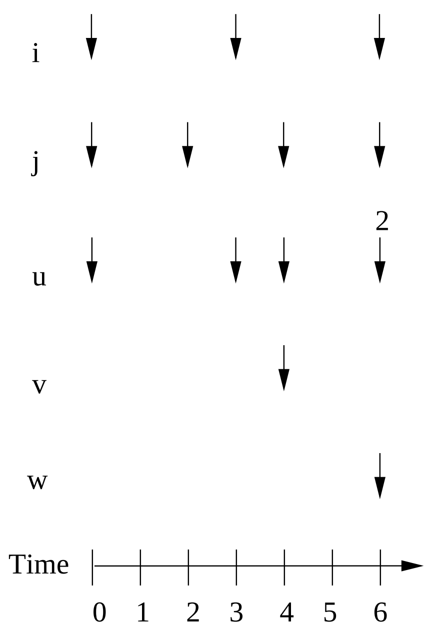


Figure 4.6: A graph with two source nodes, a snapshot sequence, and a time-line execution. Source nodes i and j have well-defined execution rates of $R_i = (1, 3)$ and $R_j = (1, 2)$ with $s_i = 0$ and $s_j = 0$ (i.e., both nodes first execute at time 0).

$4 = 6 - 2$ time units (since node w first executes at time six and the second sample is produced at time two). From the snapshot sequence, the state of the queues just before the second sample is produced at time 2 will be $length(\alpha) = 1$, $length(\beta) = 1$, $length(\gamma) = 4$, $length(\mu) = 1$, and $length(\nu) = 0$. Using these values with Equation (4.5) we get $(\mathcal{F}_{i \rightsquigarrow w} - 1) \cdot y_i = (2-1) \cdot 3 = 3$ and $(\mathcal{F}_{j \rightsquigarrow w} - 1) \cdot y_j = (2-1) \cdot 2 = 2$, which would imply an inherent latency of 3 time units for this sample. This means, however, that node w has to fire at time $2 + 3 = 5$, which contradicts the time-line execution shown in Figure 4.6. The problem is that the expression $(\mathcal{F}_{i \rightsquigarrow w} - 1) \cdot y_i$ assumes source i is producing the sample for which we are deriving the latency value. In this case, node i does not produce until time 3, and the sample was produced by node j at time 2. Certainly we can show that the actual latency will lie in the interval $[(\mathcal{F}_{i \rightsquigarrow w} - 1) \cdot y_i, \mathcal{F}_{i \rightsquigarrow w} \cdot y_i)$, which evaluates to a lower bound of five and an upper bound of eight.

We can, however, collapse this interval to a single point if we know when each source node first executes. Let each source node begin their periodic execution at time 0. Observe that for any point in time t, source j will next produce at time

$$t + y_j - (t \bmod y_j). \tag{4.6}$$

For example, let $t = 5$, and consider, once again, the graph and time-line execution in Figure 4.6. Using Equation (4.6), node j will next execute at time

$$\begin{aligned} t + y_j - (t \bmod y_j) &= 1 + 2 - (1 \bmod 2) \\ &= 2, \end{aligned}$$

and node i will next execute at time

$$\begin{aligned} t + y_i - (t \bmod y_i) &= 1 + 3 - (1 \bmod 3) \\ &= 3. \end{aligned}$$

Similarly, the k^{th} execution of node j from time t will occur at time

$$t + k \cdot y_j - (t \bmod y_j). \tag{4.7}$$

For example, let $k = 3$ and $t = 1$. Starting at time 1, the third execution of node j will occur at time

$$\begin{aligned} t + k \cdot y_j - (t \bmod y_j) &= 1 + 3 \cdot 2 - (1 \bmod 2) \\ &= 1 + 6 - 1 \\ &= 6, \end{aligned}$$

as shown in the time-line execution of Figure 4.6.

We can combine Equation (4.7) with Lemma 4.2.5 to compute the inherent latency a sample produced at time t will encounter. For example, let $t = 2$. The inherent latency the second sample produced by node j at time 2 encounters is computed as follows. First, calculate the number of executions required of each source node $u \in \mathcal{I}_w$ before node w next executes using Function $\mathcal{F}_{u \rightsquigarrow w}$. Recall from above that $\mathcal{F}_{j \rightsquigarrow w} = 2$, and $\mathcal{F}_{i \rightsquigarrow w} = 2$. Second, compute the maximum time, starting at time t, before each source node $u \in \mathcal{I}_w$ executes $\mathcal{F}_{u \rightsquigarrow w}$ times. This determines the latency of the sample produced at time 2, and can be done using Equation (4.7). Since node j executes at time 2 and $\mathcal{F}_{j \rightsquigarrow w} = 2$ includes this execution, the ${\mathcal{F}_{j \rightsquigarrow w}}^{th}$ execution of source node j from time 2 will occur at time

$$\begin{aligned} t + (\mathcal{F}_{j \rightsquigarrow w} - 1) \cdot y_j - (t \bmod y_j) &= 2 + (2 - 1) \cdot 2 - (2 \bmod 2) \\ &= 2 + 2 - 0 \\ &= 4. \end{aligned}$$

Source node i will next execute at time

$$\begin{aligned} t + \mathcal{F}_{i \rightsquigarrow w} \cdot y_i - (t \bmod y_i) &= 2 + 2 \cdot 3 - (2 \bmod 3) \\ &= 2 + 6 - 2 \\ &= 6. \end{aligned}$$

Here, we do not subtract one from $\mathcal{F}_{i \rightsquigarrow w}$ since node i does not execute at time $t = 2$. Since node w cannot execute until both of its input queues are over threshold, it does not execute until time 6 when the second execution of node i from time 2 occurs. Hence, the inherent latency of the second sample produced by node j at time 2, is

$$\begin{aligned} t - &\max(t + \mathcal{F}_{j \rightsquigarrow w} \cdot y_j - (t \bmod y_j), t + \mathcal{F}_{i \rightsquigarrow w} \cdot y_i - (t \bmod y_i)) \\ &= \max(\mathcal{F}_{i \rightsquigarrow w} \cdot y_i - (t \bmod y_i), \mathcal{F}_{i \rightsquigarrow w} \cdot y_i - (t \bmod y_i)) \\ &= \max((2-1) \cdot 2 - (2 \bmod 2), 2 \cdot 3 - (2 \bmod 3)) \\ &= \max(2, 4) \\ &= 4, \end{aligned} \tag{4.8}$$

as shown in the time-line execution of Figure 4.6.

To generalize the process of computing inherent latency in an acyclic graph with multiple periodic source nodes, let

$$L_{j \rightsquigarrow w}(t) = \begin{cases} \max(0, (\mathcal{F}_{j \rightsquigarrow w} - 1) \cdot y_j) & \text{if } ((t - s_j) \bmod y_j) = 0 \\ \max(0, \mathcal{F}_{j \rightsquigarrow w} \cdot y_j - ((t - s_j) \bmod y_j)) & \text{if } ((t - s_j) \bmod y_j) > 0 \end{cases} \tag{4.9}$$

denote the number of time units from time t before node j produces enough samples to put the input queues to node w along the paths $j \rightsquigarrow w$ over threshold. Recall that s_j denotes the time of the first execution of node j. The term $((t - s_j) \bmod y_j)$ is used to determine if node j will execute at time t given that it first executed at time s_j and has a period of y_j time units. The first branch of Equation (4.9) handles the case where node j executes at time t, and the second branch handles the case when it does not. Using Equation (4.9), computing the inherent latency encountered by a sample produced at time t is reduced to finding the maximum value of $L_{u \rightsquigarrow w}(t)$ computed for each node u in the set of graph source nodes that have paths leading to node w.

From Lemma 4.2.4, Lemma 4.2.5, and the preceding discussion, we obtain:

Lemma 4.2.6. *Let $G = (V, E, \psi)$ be an acyclic PGM graph. Let $\mathcal{I}_w$ be the set of periodic source nodes producing data for sink node $w \in \mathcal{O}$. Let s_j denote the time of the first execution of source node j, and let y_j denote the period of source node j. Under the strong synchrony hypothesis, the latency for a sample produced at time t is*

$$\mathcal{L}_w(t) = \max_{j \in \mathcal{I}_w}(L_{j \rightsquigarrow w}(t)) \tag{4.10}$$

where $L_{j \rightsquigarrow w}(t)$ is defined by Equation (4.9).

We now apply Lemma 4.2.6 to the first and second samples produced by node j in the graph of Figure 4.6 to compute their latency values. As before, let $R_i = (1, 3)$,

$R_j = (1,2)$, $s_i = 0$, and $s_j = 0$. By Lemma 4.2.6, the first sample is produced at time 0 and has a latency of 6 time units since

$$\begin{aligned}
\mathcal{L}_w(0) &= \max_{j \in \mathcal{I}_w}(L_{j \rightsquigarrow w}(0)) \\
&= \max(L_{j \rightsquigarrow w}(0), L_{i \rightsquigarrow w}(0)) \\
&= \max((\mathcal{F}_{j \rightsquigarrow w} - 1) \cdot y_j, (\mathcal{F}_{i \rightsquigarrow w} - 1) \cdot y_i) \\
&\text{since } ((t - s_j) \bmod y_j) = 0 \text{ and } ((t - s_i) \bmod y_i) = 0 \\
&= \max((\mathcal{F}_{j \rightsquigarrow w} - 1) \cdot 2, (\mathcal{F}_{i \rightsquigarrow w} - 1) \cdot 3) \\
&= \max((4 - 1) \cdot 2, (3 - 1) \cdot 3) \\
&= \max(6, 6) \\
&= 6.
\end{aligned}$$

This is the same inherent latency value we computed before, and it is the same inherent latency represented by the time-line execution in Figure 4.6.

We now compute the the inherent latency of the second sample produced by node j at time 2. From the snapshot sequence in Figure 4.6 we see that the state of the queues at time 2 are $\mathit{length}(\alpha) = 1$, $\mathit{length}(\beta) = 1$, $\mathit{length}(\gamma) = 4$, $\mathit{length}(\mu) = 1$, and $\mathit{length}(\nu) = 0$. By Lemma 4.2.6, the inherent latency for this sample is

$$\begin{aligned}
\mathcal{L}_w(2) &= \max_{j \in \mathcal{I}_w}(L_{j \rightsquigarrow w}(2)) \\
&= \max(L_{j \rightsquigarrow w}(2), L_{i \rightsquigarrow w}(2)) \\
&= \max((\mathcal{F}_{j \rightsquigarrow w} - 1) \cdot y_j, \mathcal{F}_{i \rightsquigarrow w} \cdot y_i - ((2 - s_i) \bmod y_i)) \\
&\text{since } ((t - s_j) \bmod y_j) = 0 \text{ and } ((t - s_i) \bmod y_i) > 0 \\
&= \max((\mathcal{F}_{j \rightsquigarrow w} - 1) \cdot 2, \mathcal{F}_{i \rightsquigarrow w} \cdot 2 - ((2 - 0) \bmod 3)) \\
&= \max((2 - 1) \cdot 2, 2 \cdot 3 - 2) \\
&= \max(2, 4) \\
&= 4,
\end{aligned}$$

which is the same latency value computed by Equation (4.8) and shown in the time-line execution in Figure 4.6.

4.2.2.3 Rate-based Source Nodes

Under the strong synchrony hypothesis, Lemma 4.2.6 provides a means for deriving the latency of any sample when the source nodes are periodic. We now remove our assumption of periodic input source nodes and derive lower and upper bounds for the latency produced by rate-based source nodes under the strong synchrony hypothesis.

Computing the inherent latency for acyclic graphs with rate-based source nodes is a straightforward extension of Lemma 4.2.3. Function $\mathcal{F}_{j\rightsquigarrow w}$ is evaluated for all $j \in \mathcal{I}_w$ and the maximum of these values is used in Equation (4.3) (as opposed to $F_{j\rightsquigarrow w}$). Substituting $\mathcal{F}_{j\rightsquigarrow w}$ for $\mathcal{F}_{u\rightsquigarrow w}$ in Equation (4.3) on page 124, a sample's inherent latency is bounded such that

$$\begin{aligned}\max\left(0, \left\{\left\lfloor \frac{\mathcal{F}_{j\rightsquigarrow w}-1}{x_j}\right\rfloor \cdot y_j \mid \forall j \in \mathcal{I}_w\right\}\right) \leq \; & \textit{Sample Latency} \\ & < \max\left(1, \left\{\left\lceil \frac{\mathcal{F}_{j\rightsquigarrow w}}{x_j}\right\rceil \cdot y_j \mid \forall j \in \mathcal{I}_w\right\}\right) \end{aligned} \tag{4.11}$$

Recall that $\mathcal{F}_{j\rightsquigarrow w}$ is defined as $\mathcal{F}_{j\rightsquigarrow w} = \max(\{F_p \mid p \in j\rightsquigarrow w\})$. Thus, since

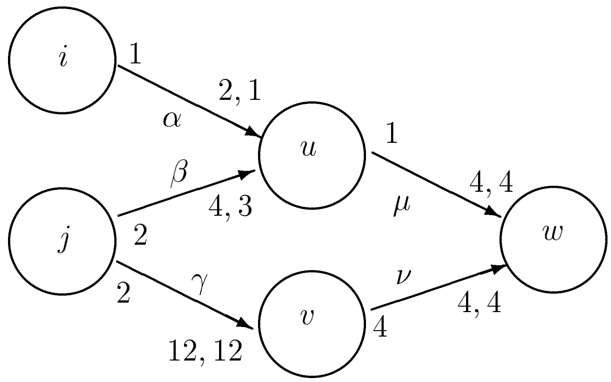
$$\begin{aligned}&\max\left(0, \left\{\left\lfloor \frac{\mathcal{F}_{j\rightsquigarrow w}-1}{x_j}\right\rfloor \cdot y_j \mid \forall j \in \mathcal{I}_w\right\}\right) \\ &\qquad = \max\left(0, \left\{\left\lfloor \frac{\max(\{F_p \mid p \in j\rightsquigarrow w\})-1}{x_j}\right\rfloor \cdot y_j \mid \forall j \in \mathcal{I}_w\right\}\right) \\ &\qquad = \max_{p\in\{j\rightsquigarrow w \mid j\in\mathcal{I}_w\}}\left(0, \left\lfloor \frac{F_p-1}{x_j}\right\rfloor \cdot y_j\right)\end{aligned}$$

and

$$\begin{aligned}&\max\left(1, \left\{\left\lceil \frac{\mathcal{F}_{j\rightsquigarrow w}}{x_j}\right\rceil \cdot y_j \mid \forall j \in \mathcal{I}_w\right\}\right) \\ &\qquad = \max\left(1, \left\{\left\lceil \frac{\max(\{F_p \mid p \in j\rightsquigarrow w\})}{x_j}\right\rceil \cdot y_j \mid \forall j \in \mathcal{I}_w\right\}\right) \\ &\qquad = \max_{p\in\{j\rightsquigarrow w \mid j\in\mathcal{I}_w\}}\left(1, \left\lceil \frac{F_p}{x_j}\right\rceil \cdot y_j\right),\end{aligned}$$

we can rewrite Equation (4.11) as

$$\max_{p\in\{j\rightsquigarrow w \mid j\in\mathcal{I}_w\}}\left(0, \left\lfloor \frac{F_p-1}{x_j}\right\rfloor \cdot y_j\right) \leq \textit{Sample Latency} < \max_{p\in\{j\rightsquigarrow w \mid j\in\mathcal{I}_w\}}\left(1, \left\lceil \frac{F_p}{x_j}\right\rceil \cdot y_j\right) \tag{4.12}$$



Time	**Execution of**	***length***(α)	***length***(β)	***length***(γ)	***length***(μ)	***length***(ν)
0	i, j, j	1	4	4	0	0
1	i, u	1	1	4	1	0
2	j, j	1	5	8	1	0
3	i, u	1	2	8	2	0
4	i, j, j, u, v	1	3	0	3	4
5	—	1	3	0	3	4
6	i, j, j, u, w	1	4	4	0	0
7	i, u	1	1	4	1	0

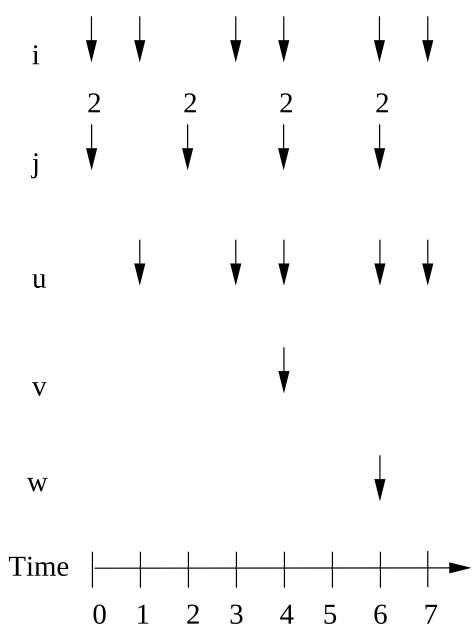


Figure 4.7: A graph with two rate-based source nodes, a snapshot sequence, and a time-line execution. Source nodes i and j have well-defined execution rates of $R_i = (2, 3)$ and $R_j = (2, 2)$ with $s_i = 0$ and $s_j = 0$.

so that only one maximum function is used.

Consider the graph and its time-line execution in Figure 4.7. This graph is the same as the graph of Figure 4.6 but with source nodes i and j producing half as many tokens during each execution. That is, $prd(\alpha) = 1$, $prd(\beta) = 2$, and $prd(\gamma) = 2$. Let the rates of the source nodes i and j be $R_i = (2, 3)$ and $R_j = (2, 2)$ so that they execute twice as often in the same interval of time as they did before. The lower bound for the inherent latency encountered by the first sample produced by node j is bounded using Equation (4.12) as follows:

$$\begin{aligned}
&\max_{p \in \{j \rightsquigarrow w | j \in \mathcal{I}_w\}} \left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j \right) \\
&\qquad = \max(0, \left\lfloor \frac{F_{(i,u,w)} - 1}{x_i} \right\rfloor \cdot y_i, \left\lfloor \frac{F_{(j,u,w)} - 1}{x_j} \right\rfloor \cdot y_j, \left\lfloor \frac{F_{(j,v,w)} - 1}{x_j} \right\rfloor \cdot y_j) \\
&\qquad = \max(0, \left\lfloor \frac{5 - 1}{2} \right\rfloor \cdot 3, \left\lfloor \frac{7 - 1}{2} \right\rfloor \cdot 2, \left\lfloor \frac{6 - 1}{2} \right\rfloor \cdot 2) \\
&\qquad = \max(0, 2 \cdot 3, 3 \cdot 2, 2 \cdot 2) \\
&\qquad = \max(0, 6, 6, 4) \\
&\qquad = 6
\end{aligned}$$

time units. The upper bound for the inherent latency encountered by the first sample produced by node j is bounded using Equation (4.12) as follows:

$$\begin{aligned}
\max_{p \in \{j \rightsquigarrow w | j \in \mathcal{I}_w\}} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j \right) &= \max(0, \left\lceil \frac{F_{(i,u,w)}}{x_i} \right\rceil \cdot y_i, \left\lceil \frac{F_{(j,u,w)}}{x_j} \right\rceil \cdot y_j, \left\lceil \frac{F_{(j,v,w)}}{x_j} \right\rceil \cdot y_j) \\
&= \max(0, \left\lceil \frac{5}{2} \right\rceil \cdot 3, \left\lceil \frac{7}{2} \right\rceil \cdot 2, \left\lceil \frac{6}{2} \right\rceil \cdot 2) \\
&= \max(0, 3 \cdot 3, 4 \cdot 2, 3 \cdot 2) \\
&= \max(0, 9, 8, 6) \\
&= 9
\end{aligned}$$

time units.

Inherent latency analysis for acyclic graphs with rate-based sources is formalized as follows.

Lemma 4.2.7. *Let $G = (V, E, \psi)$ be an acyclic PGM graph. Let $\mathcal{I}_w$ be the set of nodes producing data for sink node $w \in \mathcal{O}$. Let $R_j = (x_j, y_j)$ be a well-defined execution rate for node $j \in \mathcal{I}_w$ starting at time 0. Under the strong synchrony hypothesis, the latency a* *sample will encounter is bounded such that*

$$\max_{p \in \{j \rightsquigarrow w | j \in \mathcal{I}_w\}} \left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j \right) \leq \textit{ Sample Latency } < \max_{p \in \{j \rightsquigarrow w | j \in \mathcal{I}_w\}} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j \right) \tag{4.13}$$

Proof: By Lemma 4.2.1, for a source node $j \in \mathcal{I}_w$ and a path $p \in \{j \rightsquigarrow w\}$, F_p computes the number of times node j must execute before the input queue to node w along the path p is over threshold. By Lemma 4.2.3, it takes at least

$$\max \left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j \right)$$

time units before source node j in path $p \in \{j \rightsquigarrow w\}$ produces enough tokens to put the input queue to node w along path p over threshold. Thus, at least

$$\max_{p \in \{j \rightsquigarrow w | j \in \mathcal{I}_w\}} \left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j \right)$$

time units are required before all of the input queues to node w are over threshold (which is the lower bound for the inherent latency a sample encounters).

By Lemma 4.2.3, it takes less than

$$\max \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j \right)$$

time units for source node j in path $p \in \{j \rightsquigarrow w\}$ to produce enough tokens to put the input queue to node w along path p over threshold. Thus, less than

$$\max_{p \in \{j \rightsquigarrow w | j \in \mathcal{I}_w\}} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j \right)$$

time units are required before all of the input queues to node w are over threshold (which is the upper bound for the inherent latency a sample encounters).

Thus, under the strong synchrony hypothesis, the latency a sample will encounter is bounded such that

$$\max_{p \in \{j \rightsquigarrow w | j \in \mathcal{I}_w\}} \left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j \right) \leq \textit{ Sample Latency } < \max_{p \in \{j \rightsquigarrow w | j \in \mathcal{I}_w\}} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j \right),$$

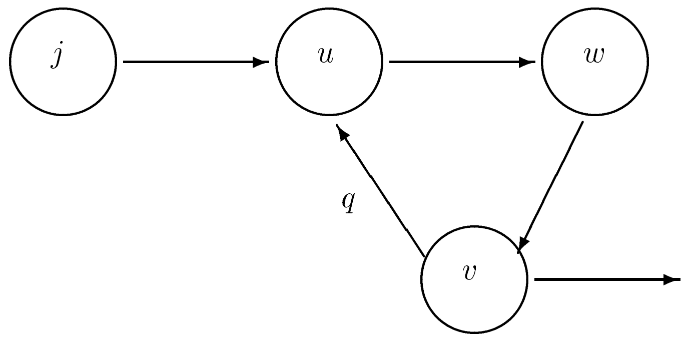


Figure 4.8: A simple three-node cycle. The queue labeled q is a back edge.

and Lemma 4.2.7 holds. □

4.2.3 Inherent Latency in Cyclic Graphs

Cycles do not change the way one computes inherent latency if each cycle's back edge is always over threshold. Recall from Section 2.4.2 that a back edge is an edge e that joins node v in a cycle to an ancestor u when the graph is topologically sorted. For the graph in Figure 4.8, the queue labeled q is a back edge. Not only is inherent latency calculated for cyclic graphs in the same way as it is for acyclic graphs, if each back edge is always over threshold, the latency computed for a cyclic graph is identical to the latency computed for an equivalent (acyclic) graph that does not contain the back edge. To see this, consider the cyclic graph in Figure 4.8. Assume all queues have unity dataflow attributes. Thus, one execution of node j puts the queue from node j to node u over threshold, and $F_{(j,u)} = 1$. If queue q is always over threshold, then it will always be the case that $F_{(j,u,w,v,u)} = 0$ since $F_{(v,u)} = 0$ when queue q is over threshold. Thus, since latency is determined by the expression

$$\max(\{F_p \mid p \in j \rightsquigarrow u\}),$$

the path (j, u, w, v, u) will never affect the inherent latency computation for the graph if queue q is always over threshold, and the inherent latency computed for the graph in Figure 4.8 will always be the same as the inherent latency computed for the graph without back edge q.

Here, as in Section 2.4.2, we negate the effect of cycles by making sure the back edge in the cycle is always over threshold. Effectively the cycle is broken by requiring back edges, such as queue q in Figure 4.8 to be initialized with

$$\left\lceil \frac{s_v - s_u + y_v}{y_u} \right\rceil \cdot x_u \cdot cns(q) + thr(q) \tag{4.14}$$

tokens, where s_u and s_v represent the first execution time of nodes u and v respectively and $\psi(q) = (v, u)$. When cycles are initialized with data in this manner, the back edge cannot influence latency since it is always over threshold. Recall that in Section 2.4.2 we did not show how to determine the first execution time of node u (i.e., the value of variable s_u). We now have the necessary equations to present the derivation of the first execution of nodes u and v, s_u and s_v used in Equation (4.14), but first we show formally that if the back edge is always over threshold it cannot affect latency. We then show how to compute s_u and s_v so that, when the back edge is initialized with a number of tokens given by Equation (4.14), the back edge is always over threshold, and that the latency computed for a cyclic graph is the same as the latency that would be computed if the back edges were removed.

Lemma 4.2.8. *Let $G = (V, E, \psi)$ be an cyclic PGM graph. Let $\mathcal{I}_u$ be the set of nodes producing data for node u. Let $\mathcal{P} = \{j \rightsquigarrow u\}$, for $j \in \mathcal{I}_u$, denote the set of paths from source node j to node u, and let $\hat{\mathcal{P}} = \{\mathcal{P} - \{(j, \ldots, u, \ldots, v, u)\}\}$ denote the set of acyclic paths from source node j to node u, where edge $\psi(q) = (v, u)$ denotes a back edge q that connects node v to node u. Under the strong synchrony hypothesis, if each back edge q is always over threshold, then*

$$\max(\left\{F_p \mid p \in \hat{\mathcal{P}}\right\}) = \max(\{F_p \mid p \in \mathcal{P}\}),$$

and the latency computed for a cyclic graph is the same as the latency that would be computed if the back edges were removed (making the graph acyclic).

Proof: By Lemma 4.2.4, if back edge q did not exist, node j would need to execute $\max(\left\{F_p \mid p \in \hat{\mathcal{P}}\right\})$ times before enough tokens were produced to put every input queue to node w in the set $\hat{\mathcal{P}}$ over threshold. If back edge q is always over threshold, then it can never prevent node u from being eligible for execution and $F_{(v,u)}$ will always be zero. Therefore, if queue q is always over threshold, it will be the case that

$$\max(\left\{F_p \mid p \in \hat{\mathcal{P}}\right\}) = \max(\{F_p \mid p \in \mathcal{P}\}),$$

and the latency computed for a cyclic graph is the same as the latency that would be computed if the back edges were removed (making the graph acyclic). $\square$

We now show how to compute s_u and s_v, so that, when the back edge is initialized with a number of tokens given by Equation (4.14), the back edge is always over threshold. We begin by restricting the set of source nodes to be periodic and then relax this restriction to include rate-based sources.

4.2.3.1 Periodic Sources Nodes

Let $G = (V, E, \psi)$ be a PGM graph with periodic source nodes. Let $\mathcal{I}_w$ be the set of nodes producing data for node $w \in V$. Let $\mathcal{P} = \{j \rightsquigarrow w\}$, for all $j \in \mathcal{I}_w$, denote the set of paths from source node j to node w. Let $\hat{\mathcal{P}} = \{\mathcal{P} - \{(j, \ldots, u, \ldots, v, u, \ldots, w)\}\}$ denote the set of acyclic paths from source node j to node $w \in \mathcal{O}$, where edge $\psi(q) = (v, u)$ denotes a back edge q that connects node v to node u. Note that if G is acyclic, then $\mathcal{P} = \hat{\mathcal{P}}$.

By Lemmas 4.2.6 and 4.2.8, if a graph is acyclic, the first sample produced at time $t = 0$ will encounter a latency of $\mathcal{L}_w(0)$ (defined by Equation (4.10) on page 134). For a cyclic graph in which every back edge is always over threshold, let

$$\hat{\mathcal{L}}_w(t) = \max_{p \in \hat{\mathcal{P}}} \begin{cases} \max(0, (F_p - 1) \cdot y_j) & \text{if } ((t - s_j) \bmod y_j) = 0 \\ \max(0, F_p \cdot y_j - ((t - s_j) \bmod y_j)) & \text{if } ((t - s_j) \bmod y_j) > 0 \end{cases}$$

denote the number of time units that elapse after time t before node u produces enough samples to put the input queues to node w in the set of paths $\{j \rightsquigarrow w\}$ over threshold. If every back edge is always over threshold, the first sample produced at time $t = 0$ will encounter a latency of $\hat{\mathcal{L}}_w(0)$. If all of the source nodes begin executing at time $t = 0$, then

$$\begin{aligned} \hat{\mathcal{L}}_w(t) &= \max_{p \in \hat{\mathcal{P}}} \begin{cases} \max(0, (F_p - 1) \cdot y_j) & \text{if } ((t - s_j) \bmod y_j) = 0 \\ \max(0, F_p \cdot y_j - ((t - s_j) \bmod y_j)) & \text{if } ((t - s_j) \bmod y_j) > 0 \end{cases} \\ &= \max_{p \in \hat{\mathcal{P}}}(0, (F_p - 1) \cdot y_j) \end{aligned}$$

since $t = 0$, $s_j = 0$, and $(t \bmod y_j) = 0$, $\forall j \in \mathcal{I}_w$. Thus, the time of the first execution of node $v \in V$ is computed as

$$s_v = \hat{\mathcal{L}}_v(0). \tag{4.15}$$

When all of the source nodes begin execution at time 0, Equation (4.15) reduces to

$$\begin{aligned} s_v &= \hat{\mathcal{L}}_v(0) \\ &= \max_{p \in \hat{\mathcal{P}}}(0, (F_p - 1) \cdot y_j). \end{aligned}$$

Inherent latency analysis for cyclic graphs with periodic sources is formalized as follows.

Lemma 4.2.9. *Let $G = (V, E, \psi)$ be a cyclic PGM graph with periodic source nodes. Let queue q be a back edge in a cycle with $\psi(q) = (v, u)$, and let execution rates R_u and R_v be well-defined. Let $\hat{\mathcal{P}}_u$ denote the set of acyclic paths from source node j to node u, and $\hat{\mathcal{P}}_v$ 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 (4.14), where s_u and s_v in Equation (4.14) are computed using Equation (4.15), queue q will always be over threshold.*

Proof: Without loss of generality, assume all source nodes begin executing at time 0. From the preceding discussion, the first execution of node u occurs at time

$$s_u = \max_{p \in \hat{\mathcal{P}}_u}(0, (F_p - 1) \cdot y_j)$$

and the first execution of node v occurs at time

$$s_v = \max_{p \in \hat{\mathcal{P}}_v}(0, (F_p - 1) \cdot y_j).$$

Since queue q is a back edge, there exists an acyclic path from source node j to node v that includes node u. Thus, $s_v \geq s_u$. Since the execution rate of node v is not derived using the execution rate of node u, there exists a positive integer k such that $y_v = k \cdot y_u$, and after time s_u node u will execute x_u times in every interval of length y_u. In an interval of length y_v, node u will execute $k \cdot x_u$ times. If queue q is initialized with a number of tokens given by Equation (4.14), it will remain over threshold until time $(s_v + y_v)$ even if node v produces no data. However, after time s_v, node v will always produce enough data for node u to execute $k \cdot x_v$ times in any interval of length y_v. Thus, since execution rates R_u and R_v are well-defined, if queue q is initialized with a number of tokens given by Equation (4.14), queue q will always remain over threshold. □

From Lemma 4.2.6, Lemma 4.2.8, and Lemma 4.2.9, we obtain:

Lemma 4.2.10. *Let $G = (V, E, \psi)$ be a cyclic PGM graph with periodic source nodes. Let queue q be a back edge in a cycle with $\psi(q) = (v, u)$. Let $\hat{\mathcal{P}}$ denote the set of acyclic paths from source node j to node $w \in \mathcal{O}$. Let every back edge q be initialized with a number of tokens given by Equation* (4.14)*, where s_u and s_v in Equation* (4.14) *are computed using Equation* (4.15)*. Under the strong synchrony hypothesis, the latency for a sample produced at time t is*

$$\hat{\mathcal{L}}_w(t) = \max_{p \in \hat{\mathcal{P}}} \begin{cases} \max(0, (F_p - 1) \cdot y_j) & \textit{if } ((t - s_j) \bmod y_j) = 0 \\ \max(0, F_p \cdot y_j - ((t - s_j) \bmod y_j)) & \textit{if } ((t - s_j) \bmod y_j) > 0 \end{cases} \tag{4.16}$$

4.2.3.2 Rate-based Source Nodes

Let $G = (V, E, \psi)$ be a PGM graph with rate-based source nodes. Let $\mathcal{I}_w$ be the set of nodes producing data for node $w \in V$. Let $\mathcal{P} = \{j \rightsquigarrow w\}$, for all $j \in \mathcal{I}_w$, denote the set of paths from source node j to node w. Let $\hat{\mathcal{P}} = \{\mathcal{P} - \{(j, \ldots, u, \ldots, v, u, \ldots, w)\}\}$ denote the set of acyclic paths from source node j to node $w \in \mathcal{O}$, where edge $\psi(q) = (v, u)$ denotes a back edge q that connects node v to node u. Note that if G is acyclic, then $\mathcal{P} = \hat{\mathcal{P}}$.

Under the strong synchrony hypothesis, Lemma 4.2.10 provides a means for computing the latency of any sample when the source nodes in the set $\mathcal{I}$ are periodic. We now remove our assumption of periodic input source nodes and derive lower and upper bounds for the latency produced by rate-based source nodes in a cyclic graph under the strong synchrony hypothesis.

If back edge q is always over threshold, it cannot affect latency (by Lemma 4.2.8). When the source nodes are rate-based, we need new equations for the start times of nodes u and v to ensure that back edge q is always over threshold (where $\psi(q) = (v, u)$). By Lemmas 4.2.8 and 4.2.7, the latency the first sample encounters in an acyclic graph is bounded such that

$$\max_{p \in \{j \rightsquigarrow w | j \in \mathcal{I}_w\}} \left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j \right) \leq \textit{ Sample Latency } < \max_{p \in \{j \rightsquigarrow w | j \in \mathcal{I}_w\}} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j \right)$$

where the execution rate of source node j is $R_j = (x_j, y_j)$. Thus, if all of the back edges in the graph are always over threshold, the earliest node u can execute is

$$s_u = \max_{p \in \hat{\mathcal{P}}} \left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j \right) \tag{4.17}$$

and node v must execute before

$$s_v = \max_{p \in \hat{\mathcal{P}}} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j \right). \tag{4.18}$$

From the proof of Lemma 4.2.9 and the preceding discussion, it follows that back edge q will always be over threshold if it is initialized with a number of tokens given by Equation (4.14), where s_u and s_v in Equation (4.14) are computed using Equations (4.17) and (4.18) respectively.

Lemma 4.2.11. *Let $G = (V, E, \psi)$ be a cyclic PGM graph with rate-based source nodes. Let queue q be a back edge in a cycle with $\psi(q) = (v, u)$, and let execution rates R_u and R_v be well-defined. Let $\hat{\mathcal{P}}_u$ denote the set of acyclic paths from source node j to node u, and $\hat{\mathcal{P}}_v$ 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* (4.14)*, where s_u in Equation* (4.14) *is computed using Equation* (4.17) *and s_v in Equation* (4.14) *is computed using Equation* (4.18)*, queue q will always be over threshold.*

From Lemma 4.2.7, Lemma 4.2.8, and Lemma 4.2.11, we obtain:

Lemma 4.2.12. *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 queue q be a back edge in a cycle with $\psi(q) = (v, u)$. Let $\hat{\mathcal{P}}$ denote the set of acyclic paths from source node j to node w. Let every back edge q be initialized with a number of tokens given by Equation* (4.14)*, where s_u in Equation* (4.14) *is computed using Equation* (4.17) *and s_v in Equation* (4.14) *is computed using Equation* (4.18)*. Under the strong synchrony hypothesis, the latency a sample will encounter is bounded such that*

$$\max_{p \in \hat{\mathcal{P}}} \left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j \right) \leq \textit{ Sample Latency } < \max_{p \in \hat{\mathcal{P}}} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j \right). \tag{4.19}$$

Lemma 4.2.12 can be used to compute the latency any sample will encounter in cyclic or acyclic graphs under the strong synchrony hypothesis. In the next section, we show how to manage imposed latency using our synthesis method and scheduling algorithms. We then combine the latency bounds derived under the strong synchrony hypothesis with the latency bounds derived for imposed latency to show how to bound and manage the total latency a sample encounters in an implementation of the graph.

4.3 Imposed Latency

Let $G = (V, E, \psi)$ be a PGM graph mapped to the RBE task set $\mathcal{T}$ according to the synthesis method presented in Chapter 3. Thus, for each node v in the graph, node v is associated with the four tuple (x_v, y_v, d_v, e_v) that characterizes an RBE task. The parameters x_v and y_v are derived using dataflow attributes of the input queues to node u and Algorithm 4 of Section 2.4.2. Parameter e_v is the worst case execution time for node v, which we assume is supplied. For ease of modeling, we will also assume e_v is constant. In an implementation of the graph, the source and sink nodes represent external devices. These are represented as RBE tasks with parameters x and y given by the external environment, and $e = 0$. (The sink and source nodes are not implemented as tasks, but are represented as tasks for ease of modeling.) The only free parameter is the relative deadline parameter d_v, the choice of which influences processor capacity requirements, latency, and buffer requirements. In general, a smaller value chosen for d_v will result in less latency and memory requirements than a larger d_v value, but at a cost of increased processor capacity requirements. We use the relative deadline parameter d_w of graph output node w to manage imposed latency. Recall that our synthesis method requires the deadline parameter associated with output node w to be greater than or equal to the deadline parameter of every node in the path from graph source node j to node w. Thus, under RBE-EDF scheduling, the maximum imposed latency a sample incurs along the path from node j to node w is bounded by d_w.

Using our synthesis method to transform a processing graph into a real-time system, managing imposed latency is a straightforward process. Moreover, unlike the computation of inherent latency, computing imposed latency is easy. Inherent latency is the delay between when a sample is produced by graph source node j and when graph sink node w executes under the strong synchrony hypothesis. Thus, imposed latency is the delay between when node w executes under the strong synchrony hypothesis and when it actually finishes executing in an actual implementation. Under RBE-EDF scheduling, which uses release time inheritance, a node's logical release time is equal to the time it would be released (and execute) under the strong synchrony hypothesis. By Theorem 3.3.5, if the graph is schedulable with the RBE-EDF scheduling algorithm (i.e., Equation (3.4) results in the affirmative), every released node v finishes its execution within d_v time units of its logical release. Thus, the upper bound on imposed latency incurred by a sample produced by source node j and consumed by sink node w is equal to d_w.

Theorem 4.3.1. *Let $G = (V, E, \psi)$ be a PGM graph. Let $\mathcal{T}$ be an RBE task set synthe-*

sized from graph G. Let j be a graph source node for which there exists a path to graph output node w. If $\mathcal{T}$ is schedulable by Equation (3.4), then the maximum imposed latency a sample incurs along the path $j \rightsquigarrow w$ is less than or equal to d_w.

Proof: Since RBE-EDF uses release-time inheritance, each task's logical release time is equal to its release time (and execution) under the strong synchrony hypothesis. Thus, the maximum imposed latency a sample incurs along the path $j \rightsquigarrow w$ is determined by when node w finishes executing. An affirmative result from Equation (3.4) means that every released task v will finish executing within d_v time units of its logical release time. Thus, sink node w will finish executing within d_w time units of its logical release time, and the maximum imposed latency a sample incurs along the path $j \rightsquigarrow w$ is less than or equal to d_w. □

A lower bound on imposed latency is the sum of the execution times of each node in the path from source node j to sink node w. Thus, the lower bound on imposed latency is proportional to the speed of the processor. A processor twice as fast as another will have an imposed latency lower bound that is half the bound computed for the slower processor. An infinitely fast processor has an imposed latency lower bound of zero since the nodes take no time to execute.

Theorem 4.3.2. *Let $G = (V, E, \psi)$ be a PGM graph. Let $\mathcal{T}$ be an RBE task set synthesized from graph G. Let j be a graph source node for which there exists a path to graph output node w. The minimum imposed latency a sample incurs along the path $j \rightsquigarrow w$ is greater than or equal to $\sum_{v \in j \rightsquigarrow w} e_v$.*

Proof: The minimum imposed latency possible occurs when source node j produces a sample and each node in the path $j \rightsquigarrow w$ executes once. Since the execution time for each node v in the path is e_v, it immediately follows that

$$\textit{Imposed Latency} \geq \sum_{v \in j \rightsquigarrow w} e_v.$$

□

4.4 Total Latency

The total latency a sample incurs can be expressed with the equation

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

Thus, no matter how a graph is implemented, a sample will never incur a latency lower than the inherent latency we derived under the strong synchrony hypothesis. The upper bound on the total latency a sample incurs is determined by the upper bound on the sample's inherent latency plus the latency imposed by the synthesis method.

In the next section we combine the bounds on inherent and imposed latency developed in Sections 4.2 and 4.3 to develop bounds on the total latency a sample incurs. The latency bounds presented in Section 4.4.1 assume that back edge, if any exist, are initialized with enough data that their queue is always over threshold. We then show, in Section 4.4.2, how to initialize back edges using the information known about the source nodes.

4.4.1 Latency Bounds

Before presenting the collection of latency bounds we have developed for various graph topologies and types of source nodes, we present a general latency theorem.

Theorem 4.4.1. *Let $G = (V, E, \psi)$ be a PGM graph. Let $\mathcal{T}$ be an RBE task set synthesized from graph G. Let j be a graph source node for which there exists a path to graph output node w. If $\mathcal{T}$ is schedulable by Equation* (3.4)*, then the latency a sample incurs along the path $j \rightsquigarrow w$ under RBE-EDF scheduling is bounded such that*

$$\mathit{Inherent\ Latency\ Lower\ Bound}\ + \sum_{v \in j \rightsquigarrow w} e_v \leq \mathit{SampleLatency}$$

and

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

Proof: The minimum latency a sample incurs is bounded from below by a lower bound on inherent latency plus the lower bound on imposed latency. Thus, by Theorem 4.3.2,

$$\mathit{Inherent\ Latency\ Lower\ Bound}\ + \sum_{v \in j \rightsquigarrow w} e_v \leq \quad \mathit{Sample\ Latency}$$

The maximum latency a sample incurs is bounded from above by the upper bound on inherent latency plus the upper bound on imposed latency. Thus, by Theorem 4.3.1 and the fact that inherent latency is always less than the *Inherent Latency Upper Bound*,

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

□

Therefore, the total latency a sample incurs is bounded using a combination of inherent latency bounds and imposed latency bounds. Table 4.1 summarizes these bounds based on graph topology and the type of graph source node (periodic or rate-based). They are also formally presented below with a theorem for each bound. Each theorem follows immediately from Theorem 4.4.1 and the corresponding inherent latency lemma from Section 4.3.

Theorem 4.4.2. *Let $j \rightsquigarrow w$ be a PGM chain $G = (V, E, \psi)$. Let $\mathcal{T}$ be an RBE task set synthesized from graph G. Let j be a periodic graph source node with period y_j. If $\mathcal{T}$ is schedulable by Equation* (3.4)*, then the latency a sample incurs along the path $j \rightsquigarrow w$ under RBE-EDF scheduling is bounded such that*

$$\begin{aligned} \max\left(0, (F_{j\rightsquigarrow w} - 1)\cdot y_u\right) + \sum_{v\in\{j\rightsquigarrow w\}} e_v \quad &\le \quad \textit{Sample Latency} \\ &< \quad \max\left(0, (F_{j\rightsquigarrow w} - 1)\cdot y_u\right) + d_w. \end{aligned}$$

Theorem 4.4.3. *Let $j \rightsquigarrow w$ be a PGM chain $G = (V, E, \psi)$. Let $\mathcal{T}$ be an RBE task set synthesized from graph G. Let j be a source node with $R_j = (x_j, y_j)$. If $\mathcal{T}$ is schedulable by Equation* (3.4)*, then the latency any sample incurs along the path $j \rightsquigarrow w$ under RBE-EDF scheduling is bounded such that*

$$\begin{aligned} \max\left(0, \left\lfloor \frac{F_{j\rightsquigarrow w} - 1}{x_i} \right\rfloor \cdot y_j\right) + \sum_{v\in\{j\rightsquigarrow w\}} e_v \quad &\le \quad \textit{Sample Latency} \\ &< \quad \max\left(1, \left\lceil \frac{F_{j\rightsquigarrow w}}{x_j} \right\rceil \cdot y_j\right) + d_w. \end{aligned}$$

Theorem 4.4.4. *Let $G = (V, E, \psi)$ be an acyclic PGM graph. Let $\mathcal{T}$ be an RBE task set synthesized from graph G. Let $\mathcal{I}_w$ be the set of periodic source nodes producing data for sink node $w \in \mathcal{O}$. Let s_j denote the time of the first execution of source node j, and let y_j denote the period of source node j. Let $\mathcal{P}_w$ denote the path from source node j to node w that creates the maximum inherent latency for the sample. If $\mathcal{T}$ is schedulable by Equation* (3.4)*, then the latency a sample produced at time t incurs along the path $j \rightsquigarrow w$ under RBE-EDF scheduling is bounded such that*

$$\mathcal{L}_w(t) + \sum_{v\in\mathcal{P}_w} e_v \quad \le \quad \textit{Sample Latency} \quad < \quad \mathcal{L}_w(t) + d_w.$$

Latency in a Chain

Source	Lower Bounds	Upper Bounds
Periodic	$\max\left(0, (F_{j\rightsquigarrow w} - 1) \cdot y_u\right) + \sum_{v \in \{j \rightsquigarrow w\}} e_v$	$\max\left(0, (F_{j\rightsquigarrow w} - 1) \cdot y_u\right) + d_w$
RBE	$\max\left(0, \left\lfloor \frac{F_{j\rightsquigarrow w} - 1}{x_i} \right\rfloor \cdot y_j\right) + \sum_{v \in \{j \rightsquigarrow w\}} e_v$	$\max\left(1, \left\lceil \frac{F_{j\rightsquigarrow w}}{x_j} \right\rceil \cdot y_j\right) + d_w$

Latency in Acyclic Graphs

Sources	Lower Bounds	Upper Bounds
Periodic	$\mathcal{L}_w(t) + \sum\limits_{v \in \mathcal{P}_w} e_v$	$\mathcal{L}_w(t) + d_w$
RBE	$\max\limits_{p \in \mathcal{P}}\left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j\right) + \sum\limits_{v \in \mathcal{P}_w} e_v$	$\max\limits_{p \in \mathcal{P}}\left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j\right) + d_w$

Latency in Cyclic Graphs

Sources	Lower Bounds	Upper Bounds
Periodic	$\mathcal{L}_w(t) + \sum\limits_{v \in \hat{\mathcal{P}}_w} e_v$	$\mathcal{L}_w(t) + d_w$
RBE	$\max\limits_{p \in \hat{\mathcal{P}}}\left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j\right) + \sum\limits_{v \in \hat{\mathcal{P}}_w} e_v$	$\max\limits_{p \in \hat{\mathcal{P}}}\left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j\right) + d_w$

Table 4.1: A summary of latency bounds. Function F_p returns the number of executions required of the source node in the path p before node w is eligible for execution. Function $\mathcal{L}_w(t)$ returns the latency a sample produced at time t incurs before node w is eligible for execution when multiple periodic source nodes have paths that lead to node w. $\mathcal{P}$ denotes the set of all paths from source nodes $j \in \mathcal{I}_w$ to node w. $\mathcal{P}_w$ denotes the path from source node j to node w that creates the maximum inherent latency for the sample. Function $\hat{\mathcal{L}}_w(t)$ returns the latency a sample produced at time t incurs before node w is eligible for execution when the graph is cyclic. $\hat{\mathcal{P}}$ denotes the set of all acyclic paths from source nodes $j \in \mathcal{I}_w$ to node w. $\hat{\mathcal{P}}_w$ denotes the acyclic path from source node j to node w that creates the maximum inherent latency for the sample.

Theorem 4.4.5. *Let $G = (V, E, \psi)$ be an acyclic PGM graph. Let $\mathcal{T}$ be an RBE task set synthesized from graph G. Let $\mathcal{I}_w$ be the set of nodes producing data for sink node $w \in \mathcal{O}$. Let $R_j = (x_j, y_j)$ be a well-defined execution rate for node $j \in \mathcal{I}_w$ starting at time 0. Let $\mathcal{P}$ denote the set of paths from source node j to node w for all $j \in \mathcal{I}_w$. Let $\mathcal{P}_w$ denote the path from source node j to node w that creates the maximum inherent latency for the sample. If $\mathcal{T}$ is schedulable by Equation* (3.4)*, then the latency a sample will incur is bounded such that*

$$\max_{p \in \mathcal{P}} \left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j \right) + \sum_{v \in \mathcal{P}_w} e_v \leq \textit{ Sample Latency } < \max_{p \in \mathcal{P}} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j \right) + d_w.$$

Theorem 4.4.6. *Let $G = (V, E, \psi)$ be a cyclic PGM graph. Let $\mathcal{T}$ be an RBE task set synthesized from graph G. Let $\mathcal{I}_w$ be the set of periodic source nodes producing data for sink node $w \in \mathcal{O}$. Let s_j denote the time of the first execution of source node j, and let y_j denote the period of source node j. Let $\hat{\mathcal{P}}_w$ denote the acyclic path from source node j to node w that creates the maximum inherent latency for the sample. If $\mathcal{T}$ is schedulable by Equation* (3.4)*, then the latency a sample produced at time t incurs along the path $j \rightsquigarrow w$ under RBE-EDF scheduling is bounded such that*

$$\hat{\mathcal{L}}_w(t) + \sum_{v \in \hat{\mathcal{P}}_w} e_v \quad \leq \quad \textit{Sample Latency} \quad < \quad \hat{\mathcal{L}}_w(t) + d_w.$$

Theorem 4.4.7. *Let $G = (V, E, \psi)$ be a cyclic PGM graph. Let $\mathcal{T}$ be an RBE task set synthesized from graph G. Let $\mathcal{I}_w$ be the set of nodes producing data for sink node $w \in \mathcal{O}$. Let $R_j = (x_j, y_j)$ be a well-defined execution rate for node $j \in \mathcal{I}_w$ starting at time 0. Let $\hat{\mathcal{P}}$ denote the set of acyclic paths from source node j to node w for all $j \in \mathcal{I}_w$. Let $\hat{\mathcal{P}}_w$ denote the acyclic path from source node j to node w that creates the maximum inherent latency for the sample. If $\mathcal{T}$ is schedulable by Equation* (3.4)*, then the latency a sample will incur is bounded such that*

$$\max_{p \in \hat{\mathcal{P}}} \left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j \right) + \sum_{v \in \hat{\mathcal{P}}_w} e_v \leq \textit{ Sample Latency } < \max_{p \in \hat{\mathcal{P}}} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j \right) + d_w.$$

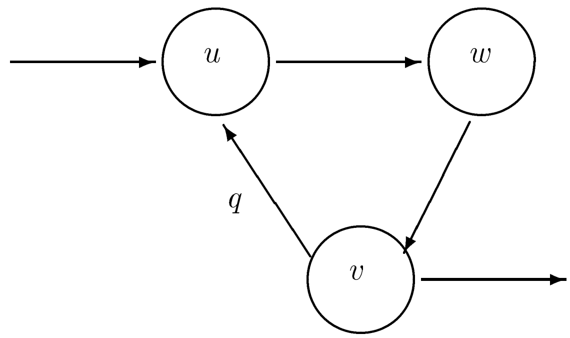


Figure 4.9: A three-node simple cycle. The queue labeled q is a back edge.

4.4.2 Initializing Back Edges

The impact of cycles on latency is negated by initializing a back edge q with enough tokens to ensure that it is always over threshold. Figure 4.9 shows a cycle with the back edge detected by a depth-first-search labeled q. The only changes we need to make to the results of Section 4.2.3, which derives the number of initial tokens required on back edge q, is to add the deadline parameter for node v to Equation (4.14). Thus, the cycle is effectively broken (as in Section 4.2.3) by requiring back edge q to be initialized with

$$\left\lceil \frac{s_v + d_v - s_u + y_v}{y_u} \right\rceil \cdot x_u \cdot cns(q) + thr(q) \tag{4.20}$$

tokens, where $\psi(q) = (v, u)$.

4.4.2.1 Periodic sources

Consider the graph of Figure 4.9. Using release-time inheritance, the release times for the first execution of nodes u and v, denoted s_u and s_v respectively, are derived using Equation (4.15) presented in Section 4.2.3. That is, if the source nodes are periodic, then $s_u = \hat{\mathcal{L}}_u(0)$ and $s_v = \hat{\mathcal{L}}_v(0)$.

Theorem 4.4.8. *Let queue q be a back edge in a cycle with $\psi(q) = (v, u)$, and let the set of source nodes $\mathcal{I}$ be periodic. If queue q is initialized with a number of tokens given by Equation* (4.20)*, where s_u and s_v in Equation* (4.20) *are derived using* (4.15)*, then queue q will always be over threshold.*

Proof: From Lemma 4.2.6, the logical release time of the first execution of node u is $s_u = \hat{\mathcal{L}}_u(0)$ and the logical release time of first execution of node v is $s_v = \hat{\mathcal{L}}_v(0)$. Since queue q is a back edge, there exists an acyclic path from source node i to node v that includes node u. Thus, $s_v \geq s_u$. Since the execution rate of node u is not derived using the execution rate of node v, there exists a non-negative integer k such that $y_v = k \cdot y_u$. Moreover, after time s_u at most x_u releases of node u will occur in every interval of length y_u. In an interval of length y_v, at most $k \cdot x_u$ releases of node u will occur. If queue q is initialized with a number of tokens given by Equation (4.20), it will remain over threshold until time $(s_v + d_v + y_v)$ even if node v produces no data. However, after time $s_v + d_v$, node v will always have produced enough data for node u to execute $k \cdot x_v$ times in any interval of length y_v since nodes u and v have well-defined rate specifications. Thus, if queue q is initialized with a number of tokens given by Equation (4.20), queue q will always remain over threshold. □

4.4.2.2 Rate-based sources

If the source nodes producing data for the cycle shown in Figure 4.9 are rate-based, and the graph is scheduled using release-time inheritance, then s_u and s_v are computed using Equations (4.17) and (4.18) presented in Section 4.2.3. That is,

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

and

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

Theorem 4.4.9. *Let queue q be a back edge in a cycle with $\psi(q) = (v, u)$. If queue q is initialized with a number of tokens given by Equation* (4.20)*, where s_u in Equation* (4.20) *is derived using Equation* (4.17) *and s_v in Equation* (4.20) *is derived using Equation* (4.18)*, then queue q will always be over threshold.*

Proof: From Lemma 4.2.7 and the proof of Theorem 4.4.8, it follows that back edge q will always be over threshold if it is initialized with a number of tokens given by Equation (4.20), where s_u and s_v are derived using Equations (4.17) and (4.18) respectively. □

4.5 Summary

In this chapter we discussed the management of latency in a real-time system synthesized from PGM graphs. We demonstrated that latency has two components, and the total latency any sample encounters can be expressed with the simple equation

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

Inherent latency is the delay between the enqueing of $prd(q)$ tokens onto queue q by source node $u \in \mathcal{I}$ and the next execution of the sink node when the strong synchrony hypothesis is assumed. Inherent latency is a function of the topology of the graph and the dataflow attributes for each queue q. If the graph's inherent latency exceeds the application's latency requirement, no implementation of the graph will meet the latency requirement and the graph must be changed. Techniques for measuring inherent latency in both acyclic and cyclic graphs were presented in Section 4.3.

Any additional latency created by scheduling and node execution is *imposed* upon the graph by the implementation and can be managed using the techniques from real-time scheduling theory that were outlined in this chapter. Our goal in building real-time systems from processing graphs is to analytically bound and then manage imposed latency. Section 4.3 presented techniques to achieve this goal for acyclic and cyclic graphs executed with our RBE-EDF scheduler.

In Section 4.4, we combined the bounds on inherent latency with the bounds for imposed latency to bound the total latency any sample will encounter in an implementation of a PGM system built with our synthesis method. These bounds are summarized in Table 4.2.

The deadline parameter associated with each node in the synthesis method of Chapter 3 was used extensively in the management of imposed latency. In general, reducing latency requires decreasing deadline parameters, which increases the processor capacity requirement or decreases the sustainable processor utilization. When the deadline parameter for every node u is set to the y parameter of its execution rate (i.e., $d_u = y_u$), a sustainable processor utilization of 100% can be achieved. As deadline parameters are reduced for low latency paths in the graph, the processor demand is increased during the intervals in which the low latency paths execute. Thus the sustainable processor utilization must decrease proportionally for the task set to remain schedulable. Using the techniques presented in this chapter, one can quantify the tradeoff between latency and sustainable processor utilization in embedded signal processing systems built from

Latency in a Chain

Source	Lower Bounds	Upper Bounds
Periodic	$\max\left(0, (F_{j\rightsquigarrow w} - 1)\cdot y_u\right) + \sum_{v\in\{j\rightsquigarrow w\}} e_v$	$\max\left(0, (F_{j\rightsquigarrow w} - 1)\cdot y_u\right) + d_w$
RBE	$\max\left(0, \left\lfloor\frac{F_{j\rightsquigarrow w}-1}{x_i}\right\rfloor\cdot y_j\right) + \sum_{v\in\{j\rightsquigarrow w\}} e_v$	$\max\left(1, \left\lceil\frac{F_{j\rightsquigarrow w}}{x_j}\right\rceil\cdot y_j\right) + d_w$

Latency in Acyclic Graphs

Sources	Lower Bounds	Upper Bounds
Periodic	$\mathcal{L}_w(t) + \sum\limits_{v\in\mathcal{P}_w} e_v$	$\mathcal{L}_w(t) + d_w$
RBE	$\max\limits_{p\in\mathcal{P}}\left(0, \left\lfloor\frac{F_p-1}{x_j}\right\rfloor\cdot y_j\right) + \sum\limits_{v\in\mathcal{P}_w} e_v$	$\max\limits_{p\in\mathcal{P}}\left(1, \left\lceil\frac{F_p}{x_j}\right\rceil\cdot y_j\right) + d_w$

Latency in Cyclic Graphs

Sources	Lower Bounds	Upper Bounds
Periodic	$\mathcal{L}_w(t) + \sum\limits_{v\in\hat{\mathcal{P}}_w} e_v$	$\mathcal{L}_w(t) + d_w$
RBE	$\max\limits_{p\in\hat{\mathcal{P}}}\left(0, \left\lfloor\frac{F_p-1}{x_j}\right\rfloor\cdot y_j\right) + \sum\limits_{v\in\hat{\mathcal{P}}_w} e_v$	$\max\limits_{p\in\hat{\mathcal{P}}}\left(1, \left\lceil\frac{F_p}{x_j}\right\rceil\cdot y_j\right) + d_w$

Table 4.2: A summary of latency bounds. Function F_p returns the number of executions required of the source node in the path p before node w is eligible for execution. Function $\mathcal{L}_w(t)$ returns the latency a sample produced at time t encounters before node w is eligible for execution when multiple periodic source nodes have paths that lead to node w. $\mathcal{P}$ denotes the set of all paths from source nodes $j \in \mathcal{I}_w$ to node w. $\mathcal{P}_w$ denotes the path from source node j to node w that creates the maximum inherent latency for the sample. Function $\hat{\mathcal{L}}_w(t)$ returns the latency a sample produced at time t encounters before node w is eligible for execution when the graph is cyclic. $\hat{\mathcal{P}}$ denotes the set of all acyclic paths from source nodes $j \in \mathcal{I}_w$ to node w. $\hat{\mathcal{P}}_w$ denotes the acyclic path from source node j to node w that creates the maximum inherent latency for the sample.

PGM graphs.

Chapter 5

Case Studies

5.1 Introduction

We illustrate our synthesis method and techniques for managing latency by analyzing three processing graphs from the literature and industry. Each corresponds to a non-trivial embedded real-time signal processing system. The first case study is an airborne synthetic aperture radar (SAR) application [65]. The SAR system can be used to identify man-made objects on the ground or in the air by producing high-resolution, all-weather images in real-time [49]. The second application is an International Maritime Satellite (INMARSAT) mobile receiver application [64, 55]. INMARSAT is a global maritime communication and navigational system used in the commercial shipping industry. The SAR and INMARSAT graphs have been used as examples throughout this dissertation, but a more complete analysis of these applications is performed here. We conclude our case studies by evaluating the latency of an anti-submarine warfare (ASW) system — the Directed Low Frequency Analysis and Recording (DIFAR) acoustic signal processing program from the Airborne Low Frequency Sonar (ALFS) subsystem of the LAMPS MK III anti-submarine helicopter [18]. The ALFS system processes low frequency signals received by sonobuoys in the water. Its primary function is to detect submarines and to calculate range and bearing estimates to each target.

The topologies of the processing graphs for these three applications ranges from a simple ten-node chain for the SAR application to a cyclic graph containing over 80 nodes and 400 queues for the DIFAR application. Each application is representative of a class of processing graphs for its respective topology. We demonstrate and evaluate our synthesis method with a chain, an acyclic graph with multiple periodic source nodes, and a cyclic graph with a rate-based source.

5.2 Synthetic Aperture Radar Application

We begin the cases studies by applying our synthesis method to a real-time Ka-band SAR application. This application is the benchmark application for the ARPA Rapid Prototyping of Application Specific Signal Processors (RASSP) project [65]. The SAR system can be used to identify man-made objects on the ground or in the air by producing high-resolution, all-weather images in real time [49]. The full SAR benchmark cannot execute in real-time on a single processor. Therefore, the RASSP project allocates portions of the full SAR graph to individual processors. The graph in Figure 5.1 is one such allocation. This graph, called the *mini-SAR*, was originally created to test tools developed by the RASSP project. It performs the range and azimuth compression processing in the formation of an image that has resolution one eighth of that formed by the full SAR benchmark. We refer to the mini-SAR graph as the SAR graph since an analysis similar to that we develop shortly could be performed on each processor individually to analyze the full application. The SAR application is interesting because it is a simple chain with a periodic source and multiple rate changes in the path from the source to the sink node.

In what follows, we provide 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: the produce, consume, and threshold values for each queue, and the execution rate of the source node.

The node labeled *YRange* represents an external computer that sends one pulse of radar data every 3.6 milliseconds (ms). (In this processing description, we deviate from our use of the terms sample and token and use terms commonly used by SAR application developers. Here a sample represents a digital value of the signal at a point in time and a pulse consists of multiple samples.) Source node *YRange* delivers 118 complex-valued range-gate samples every 3.6ms. Together, these samples constitute one pulse of radar data. The *Zero Fill* node pads the pulse with zeroes to create a pulse length of 256 samples 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 reduce side-lobe 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. Next is the *Corner Turn* processing, which is

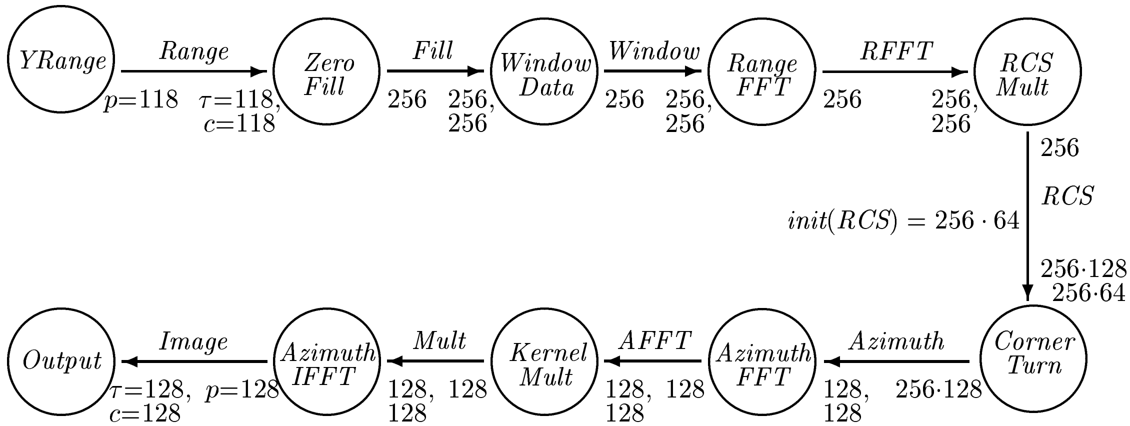


Figure 5.1: A PGM graph for the SAR application. The tail of each queue is annotated with its produce value. The head of each queue is annotated with its threshold and consume values. For example, the queue labeled *RCS* has $prd(q) = 256$, $thr(q) = 256 \cdot 128$, and $cns(q) = 256 \cdot 64$. Queue *RCS* is the only queue that initially contains data. It is initialized with $256 \cdot 64$ zero-samples (i.e., $init(RCS) = 256 \cdot 64$).

performed after 128 pulses have accumulated in its input queue. A 2-D processing array is formed where each row of the array contains one sample from the 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. This processing is achieved with a threshold of $256 \cdot 128$ samples, a consume value of $256 \cdot 64$ samples on queue *RCS*, and a produce value of $256 \cdot 128$ samples on queue *Azimuth*. Queue *RCS* is initialized with $256 \cdot 64$ zero-samples, which reduces the initial latency of the system. 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 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. For a more detailed description of the processing performed by the SAR benchmark, see "SAR Processing for RASSP Application" by Zuerndorfer *et al.* [65].

The SAR benchmark has a maximum latency requirement of three seconds, where latency refers to the elapsed time between the time a pulse is received and the time a corresponding image is output [65]. The portion of the SAR application being synthesized here has been allocated one third of the full latency requirement. Thus, the total latency inherent in the graph and imposed by the implementation must be less than one second [25].

Our synthesis method is applied to the SAR graph in the next three sections. We then compute latency bounds for the synthesized system and conclude with a discussion of latency management in the SAR application.

5.2.1 Step 1: Computation of Node Execution Rates

The first step of the synthesis method is to use Algorithm 4 from Section 3.2.2 to compute the execution rate of each node in the graph. Let $R_{YRange} = (1,\ 3.6ms)$ be a well-defined rate specification for source node *YRange* beginning at time 0 with the first execution of source node *YRange* occurring at time 0. That is, source node *YRange* executes at times $k \cdot 3.6ms$ for all $k \geq 0$. Algorithm 4 computes well-defined rate specifications for the other nodes in the graph as follows.

$$R_{ZeroFill} = (x_{ZeroFill}, y_{ZeroFill})$$

where

$$\begin{aligned} x_{ZeroFill} &= \frac{prd(Range) \cdot x_{YRange}}{\gcd(prd(Range) \cdot x_{YRange},\, cns(Range))}, \\ y_{ZeroFill} &= \frac{cns(Range) \cdot y_{YRange}}{\gcd(prd(Range) \cdot x_{YRange},\, cns(Range))}. \end{aligned}$$

Thus,

$$\begin{aligned} R_{ZeroFill} &= \left(\frac{118 \cdot 1}{\gcd(118 \cdot 1,\ 118)}, \frac{118 \cdot 3.6ms}{\gcd(118 \cdot 1,\ 118)} \right) \\ &= \left(\frac{118}{118}, \frac{118 \cdot 3.6ms}{118} \right) \\ &= (1,\ 3.6ms), \end{aligned}$$

$$
\begin{aligned}
R_{WindowData} &= \left(\frac{256 \cdot 1}{\gcd(256 \cdot 1,\ 256)}, \frac{256 \cdot 3.6ms}{\gcd(256 \cdot 1,\ 256)}\right) \\
&= \left(\frac{256}{256}, \frac{256 \cdot 3.6ms}{256}\right) \\
&= (1,\ 3.6ms),
\end{aligned}
$$

$$
\begin{aligned}
R_{RangeFFT} &= \left(\frac{256 \cdot 1}{\gcd(256 \cdot 1,\ 256)}, \frac{256 \cdot 3.6ms}{\gcd(256 \cdot 1,\ 256)}\right) \\
&= \left(\frac{256}{256}, \frac{256 \cdot 3.6ms}{256}\right) \\
&= (1,\ 3.6ms),
\end{aligned}
$$

$$
\begin{aligned}
R_{RCSMult} &= \left(\frac{256 \cdot 1}{\gcd(256 \cdot 1,\ 256)}, \frac{256 \cdot 3.6ms}{\gcd(256 \cdot 1,\ 256)}\right) \\
&= \left(\frac{256}{256}, \frac{256 \cdot 3.6ms}{256}\right) \\
&= (1,\ 3.6ms),
\end{aligned}
$$

$$
\begin{aligned}
R_{CornerTurn} &= \left(\frac{256 \cdot 1}{\gcd(256 \cdot 1,\ 256 \cdot 64)}, \frac{(256 \cdot 64) \cdot 3.6ms}{\gcd(256 \cdot 1,\ 256 \cdot 64)}\right) \\
&= \left(\frac{256}{256}, \frac{(256 \cdot 64) \cdot 3.6ms}{256}\right) \\
&= (1,\ 64 \cdot 3.6ms) \\
&= (1,\ 230.4ms),
\end{aligned}
$$

$$
\begin{aligned}
R_{AzimuthFFT} &= \left(\frac{(256 \cdot 128) \cdot 1}{\gcd((256 \cdot 128) \cdot 1,\ 128)}, \frac{128 \cdot 64 \cdot 3.6ms}{\gcd((256 \cdot 128) \cdot 1,\ 128)}\right) \\
&= \left(\frac{256 \cdot 128}{128}, \frac{128 \cdot 64 \cdot 3.6ms}{128}\right) \\
&= (256,\ 64 \cdot 3.6ms) \\
&= (256,\ 230.4ms),
\end{aligned}
$$

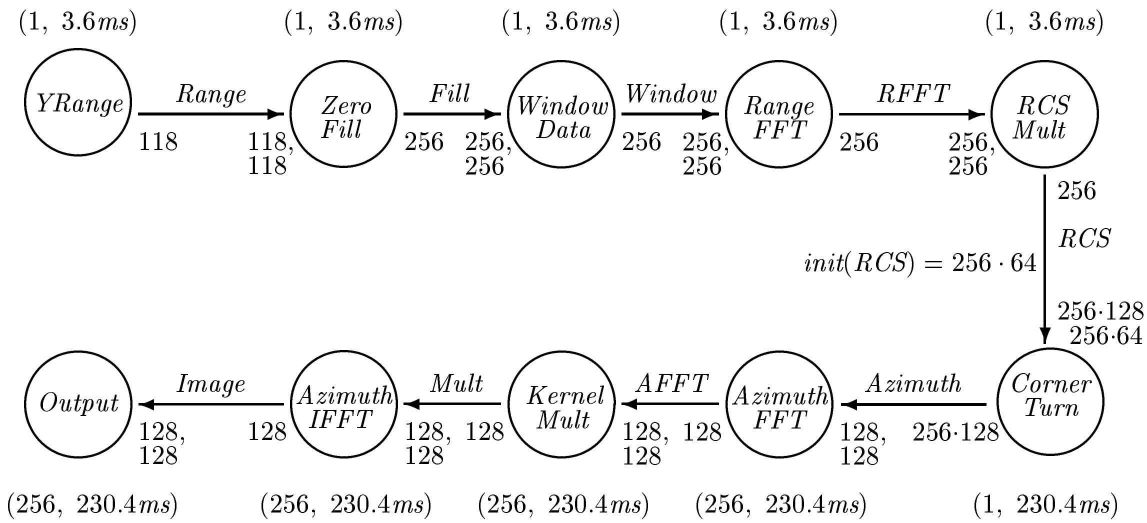


Figure 5.2: The SAR graph annotated with execution rates.

$$R_{KernelMult} = \left(\frac{128 \cdot 256}{\gcd(128 \cdot 256,\ 128)}, \frac{128 \cdot 64 \cdot 3.6ms}{\gcd(128 \cdot 256,\ 128)}\right)$$
$$= (256,\ 230.4ms),$$

$$R_{AzimuthIFFT} = \left(\frac{128 \cdot 256}{\gcd(128 \cdot 256,\ 128)}, \frac{128 \cdot 64 \cdot 3.6ms}{\gcd(128 \cdot 256,\ 128)}\right)$$
$$= (256,\ 230.4ms),\ \text{and}$$

$$R_{Output} = \left(\frac{128 \cdot 256}{\gcd(128 \cdot 256,\ 128)}, \frac{128 \cdot 64 \cdot 3.6ms}{\gcd(128 \cdot 256,\ 128)}\right)$$
$$= (256,\ 230.4ms).$$

By Theorem 2.4.4, the execution rate specifications are all well-defined beginning at time 0 (i.e., $t_u = 0$ for each node u). This is because the source node has a well-defined execution rate beginning at time 0 and because

$$MinTokens(q) \leq init(q) = thr(q) - cns(q) \leq MaxUnderThr(q)$$

for each queue q in the graph. Figure 5.2 shows the SAR graph annotated with execution rates.

Node	e_u
YRange	—
Zero Fill	$0.012ms$
Window Data	$0.25ms$
Range FFT	$0.25ms$
RCS Mult	$0.25ms$
Corner Turn	$32.0ms$
Azimuth FFT	$0.13ms$
Kernel Mult	$0.13ms$
Azimuth IFFT	$0.13ms$
Output	—

Table 5.1: Estimated worst-case execution times for nodes in the SAR graph. Nodes *YRange* and *Output* represent external devices.

5.2.2 Step 2: Map Nodes to Tasks in the RBE Model

The second step of the synthesis method associates each PGM node of the SAR graph with an RBE task by associating each node u with a four tuple (x_u, y_u, d_u, e_u) that characterizes an RBE task. The parameters x_u and y_u were derived in the rate computation step as the rate specification for node u. Table 5.1 lists the estimated worst-case execution time e_u for each node u executing on a 100MHz PowerPC processor with all cache misses. The actual worst-case execution times were not available for this application. These numbers are approximations based on cycle counts of similar processing functions.

The only free parameter is the relative deadline parameter d_u. The choice for the value of d_u influences latency and processor demand. We begin by setting $d_u = y_u$ for each node u in the SAR graph. It will turn out that this mapping is sufficient to guarantee that the application will meet its latency requirement. Table 5.2 shows the resulting RBE parameters associated with each node.

5.2.3 Step 3: Verify Schedulability

The third step of the synthesis method is to verify that the resulting task set is schedulable. This ensures that a real-time execution can be guaranteed. By Theorem 3.3.5, the RBE task set constructed from the SAR graph is schedulable using RBE-EDF scheduling

Node	t_u	s_u	(x_u,	y_u,	d_u,	e_u)
YRange	0	0	(1,	$3.6ms$)	—	—
Zero Fill	0	0	(1,	$3.6ms$,	$3.6ms$,	$0.012ms$)
Window Data	0	0	(1,	$3.6ms$,	$3.6ms$,	$0.25ms$)
Range FFT	0	0	(1,	$3.6ms$,	$3.6ms$,	$0.25ms$)
RCS Mult	0	0	(1,	$3.6ms$,	$3.6ms$,	$0.25ms$)
Corner Turn	0	$226.8ms$	(1,	$230.4ms$,	$230.4ms$,	$32.0ms$)
Azimuth FFT	0	$226.8ms$	(256,	$230.4ms$,	$230.4ms$,	$0.13ms$)
Kernel Mult	0	$226.8ms$	(256,	$230.4ms$,	$230.4ms$,	$0.13ms$)
Azimuth IFFT	0	$226.8ms$	(256,	$230.4ms$,	$230.4ms$,	$0.13ms$)
Output	0	$226.8ms$	(256,	$230.4ms$)	—	—

Table 5.2: RBE parameters associated with each node in the SAR graph. For each node u in the graph, $d_u = y_u$. The variable t_u represents the beginning of an interval in which the rate specification for node u is well-defined. Nodes *YRange* and *Output* represent external devices and are not implemented as a task.

if

$$\forall L > 0, \quad L \geq \sum_{i=1}^{n} f\left(\frac{L - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i \tag{5.1}$$

where

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

Since $d_u = y_u$ for every node u in the graph, we can use a simpler test, namely Equation (5.2), to evaluate the schedulability of the graph under RBE-EDF scheduling.

$$\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} \leq 1 \tag{5.2}$$

The left-hand side of Equation (5.2) measures processor utilization. By Equation (3.6), when $d_u = y_u$ for every node u, if a task set satisfies Equation (5.2) then the task set will also satisfy Equation (5.1). Using the RBE parameters from Table 5.2, we see that the graph is schedulable since

$$\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} \;=\; 0.003 + 0.07 + 0.07 + 0.07 + 0.144 + 0.144 + 0.144 + 0.144 \;=\; 0.79$$

is less than one. Thus, the graph is schedulable with $d_u = y_u$ for each node u in the graph. We must now verify that the latency requirement is met with these RBE parameters.

5.2.4 Computing Latency

The latency a sample encounters is dependent on the number of samples in each queue when the sample arrives. By Theorem 4.4.2 this latency is bounded such that

$$\begin{aligned} max\,(0, (F_{j \rightsquigarrow w} - 1) \cdot y_u) + \sum_{v \in \{j \rightsquigarrow w\}} e_v \quad &\leq \quad \textit{sample latency} \\ &< \quad \max\,(0, (F_{j \rightsquigarrow w} - 1) \cdot y_u) + d_w \end{aligned}$$

$$\begin{aligned} \max\,(0, (F_{\mathit{YRange} \rightsquigarrow \mathit{Output}} - 1) \cdot 3.6ms) + \sum_{u \in \{\mathit{YRange} \rightsquigarrow \mathit{Output}\}} e_u \quad &\leq \quad \textit{sample latency} \\ &< \quad \max\,(0, (F_{\mathit{YRange} \rightsquigarrow \mathit{Output}} - 1) \cdot 3.6ms) + 230.4ms \end{aligned} \tag{5.3}$$

where $F_{j \rightsquigarrow w}$, defined by Equation (4.1), gives the number of executions required of node j before node w is eligible for execution.

Thus, since only queue RCS is initialized with data, the latency the first sample encounters is bounded such that

$$\begin{aligned} \max\,(0, (64 - 1) \cdot 3.6ms) + 33.152ms \leq \textit{sample latency}& \\ < \max\,(0, (64 - 1) \cdot 3.6ms) + 230.4ms& \\ 259.952ms \leq \;\; \textit{sample latency} \;\; < 457.2ms.& \end{aligned}$$

For the second sample, we again evaluate Equation (5.3) at time $3.6ms$ (just before the source node produces). The latency the second sample encounters is bounded such that

$$\begin{aligned} \max\,(0, (63 - 1) \cdot 3.6ms) + 33.152ms \leq \textit{sample latency}& \\ < \max\,(0, (63 - 1) \cdot 3.6ms) + 230.4ms& \\ 256.352ms \leq \;\; \textit{sample latency} \;\; < 453.6ms.& \end{aligned}$$

The latency bounds continue to decrease by $3.6ms$ for each sample so that the latency incurred by the 64^{th} sample is bounded such that

$$\begin{aligned}\max(0, (1-1)\cdot 3.6ms) + 33.152ms \leq\ & \textit{sample latency} \\ & < \max(0, (1-1)\cdot 3.6ms) + 230.4ms \\ 33.152ms \leq\ \textit{sample latency}\ & < 230.4ms.\end{aligned}$$

The latency bounds computed for the 65^{th} sample are the same as the latency bounds for the first sample. This is because $y_{Output} = 64 \cdot y_{\text{Source}}$ and the execution rate specification for node *Output* is well-defined beginning at time 0. Thus, there are 64 unique latency bounds, and after the 64^{th} sample, latency patterns repeat. The latency for sample j, $1 \leq j \leq 64$, is

$$\begin{aligned}(63 - ((j-1) \bmod 64)) \cdot 3.6ms + 33.152ms \leq\ & \textit{sample latency} \\ & < (63 - ((j-1) \bmod 64)) \cdot 3.6ms + 230.4ms.\end{aligned}$$

From this expression we see that any sample will encounter a latency of at least $33.152ms$ and at most $457.2ms$. Thus, the application is guaranteed to meet its latency requirement since the upper bound latency requirement for the graph is one second.

5.2.5 Discussion

The synthesis of a real-time system from the SAR PGM graph that meets the application's latency requirements can be accomplished with only one iteration of the synthesis method. However, if, for example, the latency requirement was $400ms$, we would need to repeat the second and third steps of the synthesis method with new deadline parameter values. Alternatively, we could have selected deadline parameter values that met this latency requirement during the first iteration. For example, assume the latency requirement was $400ms$. By Lemma 4.2.2, the inherent latency of the first sample is

$$\begin{aligned}\max(0, (F_{YRange \rightsquigarrow Output} - 1) \cdot y_{YRange}) &= \max(0, 64-1) \cdot 3.6ms) \\ &= 63 \cdot 3.6ms \\ &= 226.8ms,\end{aligned}$$

which is also the maximum latency any sample will encounter. Thus, to meet a latency requirement of $400ms$, we simply need to make sure that every deadline parameter in the

Node	t_u	s_u	(x_u,	y_u,	d_u,	e_u)
YRange	0	0	(1,	$3.6ms$)	—	—
Zero Fill	0	0	(1,	$3.6ms$,	$3.6ms$,	$0.012ms$)
Window Data	0	0	(1,	$3.6ms$,	$3.6ms$,	$0.25ms$)
Range FFT	0	0	(1,	$3.6ms$,	$3.6ms$,	$0.25ms$)
RCS Mult	0	0	(1,	$3.6ms$,	$3.6ms$,	$0.25ms$)
Corner Turn	0	$226.8ms$	(1,	$230.4ms$,	$173.2ms$,	$32.0ms$)
Azimuth FFT	0	$226.8ms$	(256,	$230.4ms$,	$173.2ms$,	$0.13ms$)
Kernel Mult	0	$226.8ms$	(256,	$230.4ms$,	$173.2ms$,	$0.13ms$)
Azimuth IFFT	0	$226.8ms$	(256,	$230.4ms$,	$173.2ms$,	$0.13ms$)
Output	0	$226.8ms$	(256,	$230.4ms$)	—	—

Table 5.3: RBE parameters associated with each node in the SAR graph that result in a maximum latency that is less than $400ms$.

RBE task set is less than or equal to

$$\begin{aligned} \textit{latency requirement} - \textit{maximum inherent latency} &= 400ms - 226.8ms \\ &= 173.2ms. \end{aligned}$$

Table 5.3 lists the new RBE parameters associated with each node using this approach.

Note that the last five nodes have deadline parameters that are less than their y parameters. This means we cannot use Equation (5.2) to evaluate the schedulability of the task set. Thus, by Theorem 3.3.5 we have to evaluate Equation (5.1) for all $L > 0$. This is clearly not a tractable option. However, by combining Theorem 3.3.5 with the scheduling theory results of Baruah *et al.* [6], we can evaluate the schedulability of the new RBE task set using the expression

$$0 < L \leq (s_{Output} + 2 \cdot y_{Output}), \quad L \geq \sum_{i=1}^{n} f\left(\frac{L - d_i + y_i}{y_i}\right) \cdot x_i \cdot e_i \tag{5.4}$$

where $s_{Output} = \mathcal{L}_{Output}(0)$ is the first logical release time of sink node *Output* and $f(a)$ is the floor function used in Equation (5.4). In this case, the upper bound on L is equal to $687.6ms$, and Equation (5.4) is a theorem for the RBE parameters in Table 5.3.

Not all PGM applications have as clearly stated latency requirements as the SAR. In many cases, the graph must simply execute in real time. That is, the nodes must process the signal as it arrives and without losing data. Thus, the first iteration of second step in the synthesis method (where nodes are mapped to RBE tasks) assumes a deadline parameter equal to the y parameter. This mapping guarantees real-time execution since each node will execute according to its rate specification when the processor is not overloaded. Moreover, this mapping guarantees real-time execution without creating more processor demand in any interval of time than that which is required to process the signal in real time.

Many signal processing applications have a latency requirement that is less than or equal to twice the maximum y parameter of the graph output nodes. In this case, deadline parameters that ensure the requirement is met can be selected during the first iteration of synthesis method when nodes are mapped to tasks. This is accomplished using the technique demonstrated in the previous example where we assumed a $400ms$ latency requirement of the SAR. That is, if the latency requirement were l, then for each node u in the SAR graph, we would set

$$\begin{aligned} d_u &= \min(y_u, \textit{latency requirement} - \textit{maximum inherent latency}) \\ &= min(y_u, l - 226.8) \end{aligned}$$

since $226.8ms$ is an upper bound on inherent latency for the graph.

5.3 Mobile Satellite Receiver Application

We now apply our synthesis method to an INMARSAT mobile receiver application from the literature [54, 64, 55, 9]. This example illustrates our synthesis method and latency analysis techniques with an acyclic graph that has two periodic source nodes. Since no execution times are available for the processing nodes, we will assume the graph is schedulable and only consider the first two steps of the synthesis method. That is, we compute node execution rates and then map the nodes to RBE tasks, but we do not perform a schedulability analysis. We will assume the resulting task set is schedulable. We begin with a brief introduction to INMARSAT and the mobile satellite receiver application.

The INMARSAT system has been offering mobile satellite communications service since 1982 to the commercial shipping industry. It is a global satellite constellation of seven geostationary satellites providing communications in the L-band frequencies. The INMARSAT-B mobile terminal provides digital telecommunications supporting facsimile and data transmissions at the standard rate of 9.6 kbps and an optional high-speed data rate of 64 kbps. The 64 kbps channel can be multiplexed to offer several simultaneous voice and data lines. The high-speed data option of the INMARSAT-B mobile terminal is

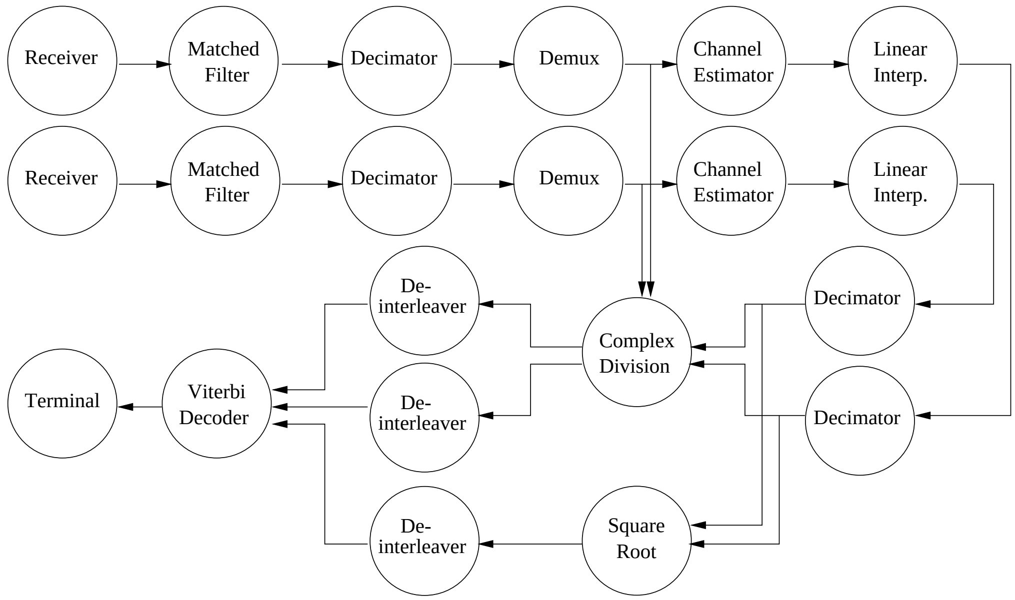


Figure 5.3: A block diagram of the INMARSAT mobile satellite receiver.

also used to provide video teleconferencing and compressed or delayed video transmission services to remote locations on land or at sea.

Figure 5.3 is a block diagram of the digital signal processing performed by the satellite receiver portion of an INMARSAT mobile terminal [64]. An abstract representation of the PGM graph for this application is shown in Figure 5.4 [55]. Source nodes I_1 and I_2 represent the input devices receiving the satellite signal. Sink node O_1 represents the terminal accepting the processed signal. For this application, each queue's threshold is equal to its consume value. To reduce clutter in the figure, we have only labeled the non-unity dataflow attributes. Produce values are located at the tail of the queue and consume values are at the head of the queue.

5.3.1 Software Synthesis

Let $R_{\mathrm{I}_1} = (1, y)$ and $R_{\mathrm{I}_2} = (1, y)$ be well-defined execution rates starting at time 0. Algorithm 4 from from Section 3.2.2 performs a topological sort of the graph and then steps through the sorted list once to compute well-defined rate specifications for each node in the graph. The topologically sorted list of nodes is

$$I_2 DEFKLMNSI_1 ABCGHIJTUVPRQWO_1$$

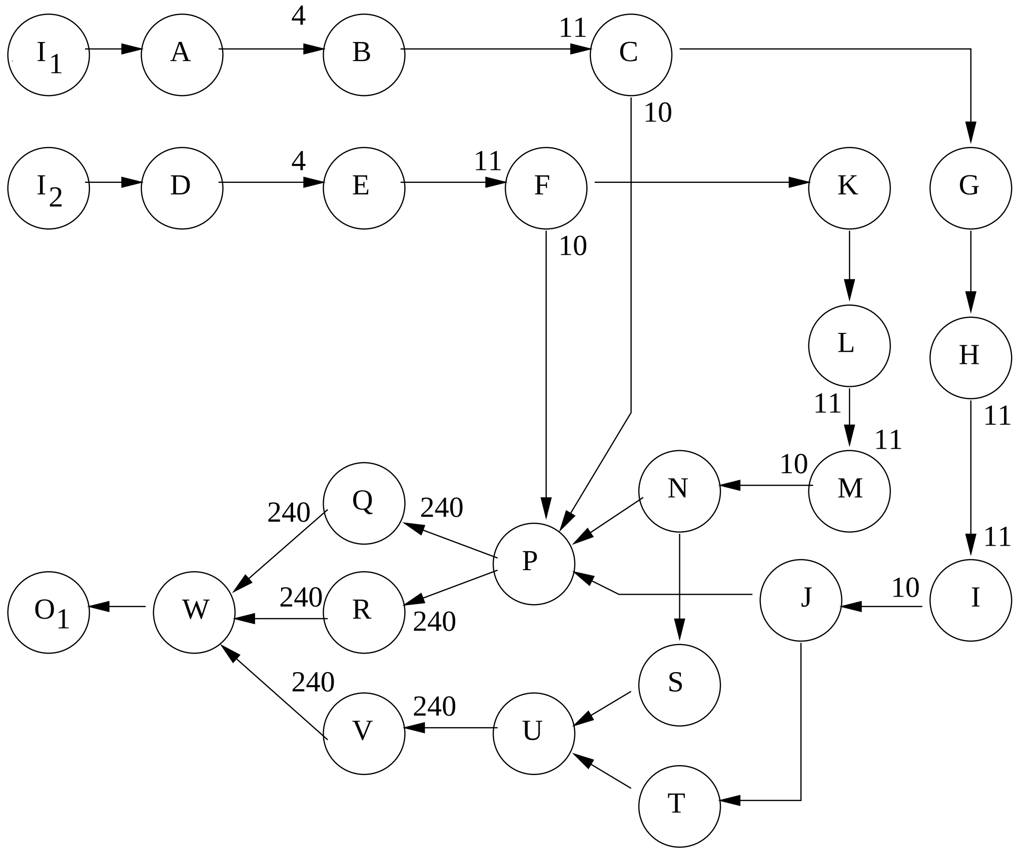


Figure 5.4: A PGM graph for the INMARSAT mobile satellite receiver. All queues have produce amounts equal to one unless otherwise noted. Each queue's threshold value is equal to its consume value, and all consume (threshold) values are equal to one unless otherwise specified. Produce amounts greater than one are displayed near the tail of a queue and consume (threshold) amounts greater than one are displayed near the head of a queue. For example, the queue joining nodes A and B has a produce amount of one, a threshold of four, and a consume amount of four.

when the depth-first search of the sort starts with node I_1. The computation of well-defined rate specifications for nodes D, E, F, and P are shown below using Theorem 2.4.9.

$R_D = (x_D, y_D)$ where

$$\begin{aligned} y_D &= \mathrm{lcm}\{\frac{cns(q)\cdot y_v}{\gcd(prd(q)\cdot x_v,\, cns(q))} \mid \forall q \in E \wedge \forall v \in V\ : \psi(q) = (v, D)\} \\ &= \mathrm{lcm}(\frac{1}{\gcd(1\cdot 1, 1)}\cdot y) \\ &= y, \end{aligned}$$

and thus,

$$\begin{aligned} x_D &= y_D \cdot \frac{prd(q)\cdot x_v}{cns(q)\cdot y_v}, \\ &= y \cdot \left(\frac{1\cdot 1}{1\cdot y}\right) \\ &= 1. \end{aligned}$$

Therefore $R_D = (1,\ y)$.

$R_E = (x_E, y_E)$ where

$$\begin{aligned} y_E &= \mathrm{lcm}(\frac{4}{\gcd(1\cdot 1, 4)}\cdot y) \\ &= 4y, \\ x_E &= 4y \cdot \left(\frac{1\cdot 1}{4\cdot y}\right) \\ &= 1. \end{aligned}$$

Therefore $R_E = (1,\ 4y)$.

$R_F = (x_F, y_F)$ where

$$\begin{aligned} y_F &= \mathrm{lcm}(\frac{11}{\gcd(1\cdot 1, 11)}\cdot 4y) \\ &= 44y, \\ x_F &= 44y \cdot \left(\frac{1\cdot 1}{11\cdot 4y}\right) \\ &= 1. \end{aligned}$$

Therefore $R_F = (1,\ 44y)$.

$R_P = (x_P, y_P)$ where

$$\begin{aligned}
y_P &= \text{lcm}\{\frac{1}{\gcd(10\cdot 1,1)}\cdot 44y, \frac{1}{\gcd(10\cdot 1,1)}\cdot 44y,\\
&\qquad \frac{1}{\gcd(1\cdot 10,1)}\cdot 44y, \frac{1}{\gcd(1\cdot 10,1)}\cdot 44y\}\\
&= 44y,\\
x_P &= 44y\cdot\left(\frac{10\cdot 1}{1\cdot 44y}\right)\\
&= 10.
\end{aligned}$$

Therefore $R_P = (10,\ 44y)$.

Figure 5.5 shows each node of the INMARSAT graph annotated with its execution rate, and Table 5.4 shows the rate specification and t_u values for each node u in the graph. The variable t_u represents the beginning of the first interval in which well-defined rate specification for node u exists. By Theorem 2.4.4, the execution rate specifications are all well-defined beginning at time 0 (i.e., $t_u = 0$ for each node u). This is because the source node has a well-defined execution rate beginning at time 0 and, since $thr(q) - cns(q) = 0$,

$$MinTokens(q) \le init(q) = thr(q) - cns(q) \le MaxUnderThr(q)$$

for each queue q in the graph.

For this case study, we assume the application is schedulable with our selected parameters. Thus, we skip the third step of our synthesis method where the schedulability of the graph is verified, and analyze the latency of the application.

5.3.2 Managing Latency

We consider two cases in the analyses performed here so that we can illustrate the difference between latency analysis for graphs with rate-based source nodes and latency analysis for graphs with periodic source nodes. In the first example we consider the case where the source nodes are rate-based.

Assume $R_{\mathrm{I}_1} = (1, y)$ and $R_{\mathrm{I}_2} = (1, y)$ are well-defined execution rates starting at time 0, but suppose the time of their first execution has not been specified. In this case we must use Theorem 4.4.5 for rate based sources to derive an upper bound on the latency any sample will encounter. The first sample produced encounters the maximum latency possible since none of the queues are initialized with data. Thus, by Theorem 4.4.5, when

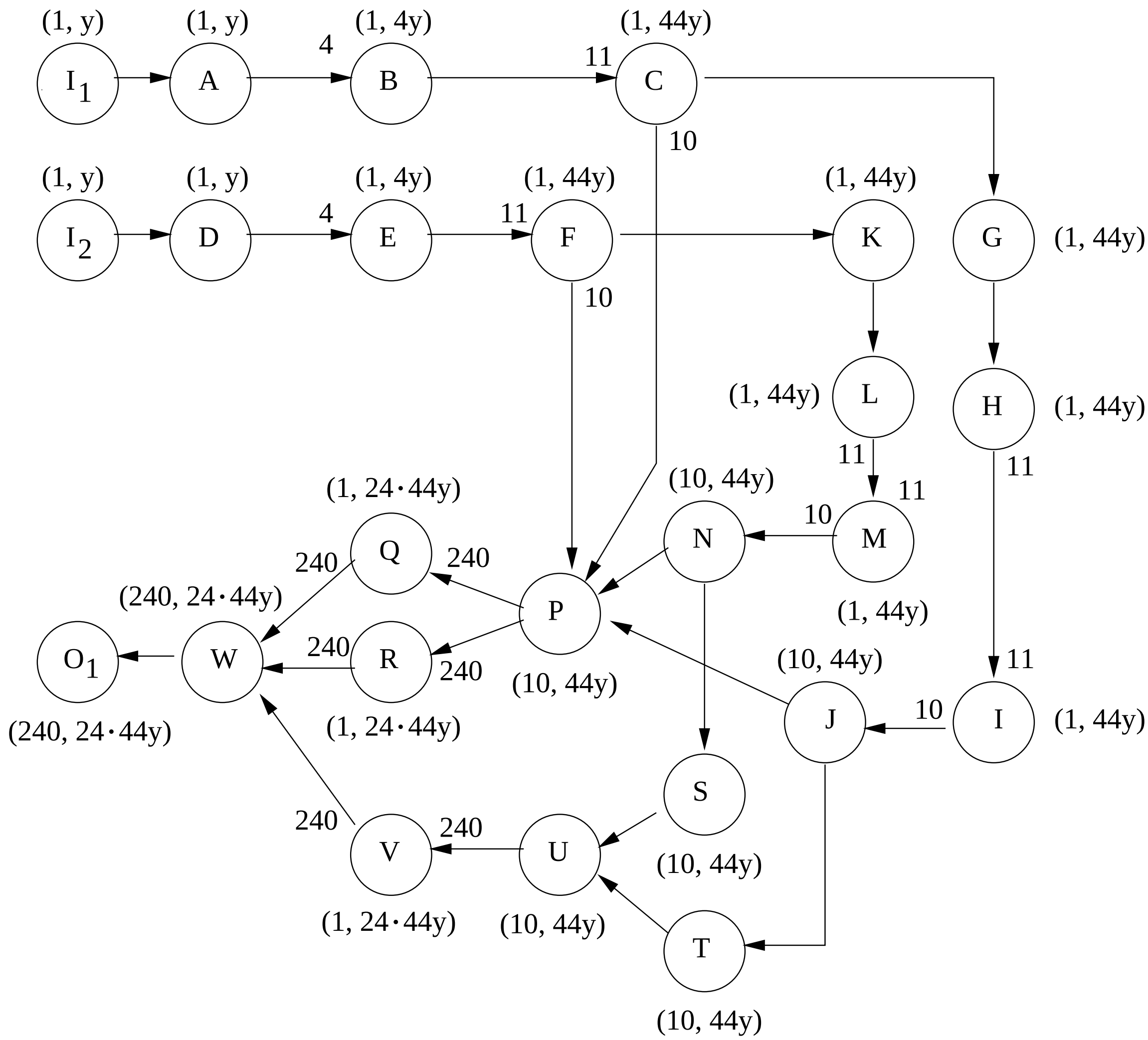


Figure 5.5: The INMARSAT graph with node execution rates. This is the same application graph shown in Figure 5.4, but now the nodes are annotated with their execution rates.

Node	t_u	$(\ x_u,$	$y_u)$
I_2	0	(1,	y)
D	0	(1,	y)
E	0	(1,	$4y$)
F	0	(1,	$44y$)
K	0	(1,	$44y$)
L	0	(1,	$44y$)
M	0	(1,	$44y$)
N	0	(10,	$44y$)
S	0	(10,	$44y$)
I_1	0	(1,	y)
A	0	(1,	y)
B	0	(1,	$4y$)
C	0	(1,	$44y$)
G	0	(1,	$44y$)
H	0	(1,	$44y$)
I	0	(1,	$44y$)
J	0	(10,	$44y$)
T	0	(10,	$44y$)
U	0	(10,	$44y$)
V	0	(1,	$24 \cdot 44y$)
P	0	(10,	$44y$)
R	0	(1,	$24 \cdot 44y$)
Q	0	(1,	$24 \cdot 44y$)
W	0	(240,	$24 \cdot 44y$)
O_1	0	(240,	$24 \cdot 44y$)

Table 5.4: Execution rate specifications for nodes in the INMARSAT PGM graph. The variable t_u represents the beginning of the first interval for which there exists a well-defined rate specification for node u. The graph nodes are listed topological order.

all of the deadline parameters are less than or equal to d_{O_1}, the latency must be less than

$$\begin{aligned}\max(1, (\mathcal{F}_{\mathrm{I}_1 \rightsquigarrow \mathrm{O}_1} \cdot y), (\mathcal{F}_{\mathrm{I}_1 \rightsquigarrow \mathrm{O}_1} \cdot y)) + d_{\mathrm{O}_1} &= \max(1, (24 \cdot 44)y, (24 \cdot 44)y) + d_{\mathrm{O}_1} \\ &= \max(1, 1056y, 1056y) + d_{\mathrm{O}_1} \\ &= 1056y + d_{\mathrm{O}_1},\end{aligned}$$

Hence, we can manage the amount of latency any sample encounters by choosing appropriate values for d_{O_1}. For example, if

$$\begin{aligned}d_{\mathrm{O}_1} &= y_{\mathrm{O}_1} \\ &= 1,056y,\end{aligned}$$

the maximum latency is less than $2,112y$. If

$$\begin{aligned}d_{\mathrm{O}_1} &= y_P \\ &= 44y,\end{aligned}$$

the maximum latency is less than $1,100y$. The lower bound on the choice for d_{O_1} is, of course, limited since can only choose values that result in a schedulable task set.

Now consider the case when the time of the first execution for each source node is specified. Assume source node I_1 first executes at time 0, and source node I_2 first executes at time 2. Since the rate specifications are well-defined, source node I_1 executes with period y starting at time 0, and source node I_2 executes with period y starting at time 2. Thus, we can apply Theorem 4.4.4 to derive a tighter upper bound on latency. Let $y > 2$. By, Theorem 4.4.4, the first sample produced at time 0 by source node I_1 encounters a latency of $\mathcal{L}_{\mathrm{O}_1}(0) + d_{\mathrm{O}_1}$, where $\mathcal{L}_{\mathrm{O}_1}(t)$ is the inherent latency of a sample at time t, and d_{O_1} is the maximum imposed latency. The inherent latency encountered by the sample produced at time $t = 0$, is

$$\begin{aligned}\mathcal{L}_{\mathrm{O}_1}(0) &= \max(0, (\mathcal{F}_{\mathrm{I}_1 \rightsquigarrow \mathrm{O}_1} - 1)y, (\mathcal{F}_{\mathrm{I}_2 \rightsquigarrow \mathrm{O}_1} \cdot y) - (2 \bmod y)) \\ &= \max(0, 1055y, 1056y - 2) \\ &= 1056y - 2.\end{aligned}$$

Thus the first sample produced by source node I_1 encounters a latency of $(1056y-2)+d_{\mathrm{O}_1}$.

The first sample produced by source node I_2 encounters a latency of $\mathcal{L}_{\mathrm{O}_1}(2) + d_{\mathrm{O}_1}$ where

$$\begin{aligned}\mathcal{L}_{\mathrm{O}_1}(2) &= \max(0, (\mathcal{F}_{\mathrm{I}_1 \rightsquigarrow \mathrm{O}_1} \cdot y) - (2 \bmod y), (\mathcal{F}_{\mathrm{I}_2 \rightsquigarrow \mathrm{O}_1} - 1)y) \\ &= \max(0, 1055y - 2, 1055y) = 1055y.\end{aligned}$$

Thus, an upper bound on the latency any sample will encounter is $1,056y - 2 + d_{\mathrm{O}_1}$.

5.4 DIFAR Application

We conclude our case studies by applying our synthesis method to a graph in an anti-submarine warfare (ASW) system — the Directed Low Frequency Analysis and Recording (DIFAR) acoustic signal processing graph from the Airborne Low Frequency Sonar (ALFS) subsystem of the LAMPS MK III anti-submarine helicopter. The ALFS system processes low frequency signals received by sonobuoys in the water. Its primary function is to detect submarines and to calculate range and bearing estimates to each target. The actual processing performed by the graph is classified by the U.S. Government, so the following is an unclassified and abbreviated description of the DIFAR graph [26]. An understanding of the actual processing is not necessary to perform the software synthesis 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 5.6 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

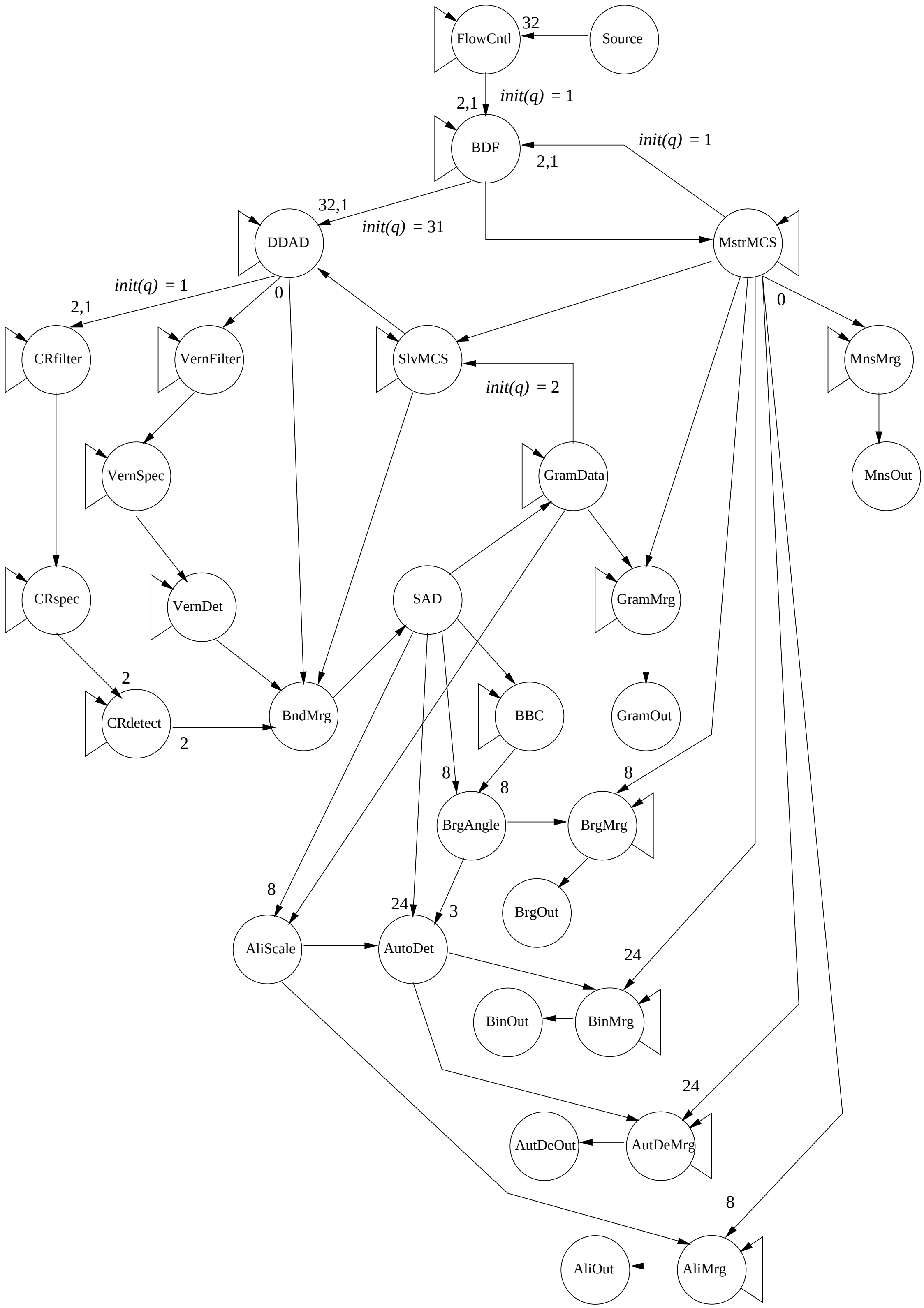


Figure 5.6: The PGM DIFAR Graph. All back edges, including self-loop edges, are initialized so that they are always over threshold.

(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 5.6. 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 5.6 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.

The software synthesis and analysis performed in this section is done using the portion of the DIFAR graph shown in Figure 5.6. The results presented here are from a study conducted under contract to General Dynamics to determine the number of 200 MHz PowerPC processors that are needed to meet seven different ALFS worst-case concurrent processing requirements [26]. One of the concurrency modes supports processing data from 16 different sonobuoys simultaneously. The actual input data rates and the specific latency requirements are classified. However, it turns out that the graph is able to meet its latency requirement when nodes are executed according to their rate specifications with a deadline parameter equal to the y parameter of their rate specification. Here, we will assume the source delivers 16 samples every $625ms$.

5.4.1 Step 1: Computation of Node Execution Rates

Let $R_{Source} = (16,\ 625ms)$ be a well-defined rate specification for source node *Source* beginning at time 0. Table 5.5 lists, in topological order, the rate specifications for the other nodes in the graph as computed by Algorithm 4. Excluding self-loops, two back edges were detected: 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, according to Theorem 4.4.9 and Theorem 3.2.1, before we can be guaranteed that the rate specifications derived using Algorithm 4 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. Let $\hat{\mathcal{P}}_u$ denote the set of acyclic paths from a node *Source* to node u, and let $\hat{\mathcal{P}}_v$ denote the set of acyclic paths from source node *Source* to node v in the DIFAR graph. By Theorem 4.4.9, back edge q, connecting node v to node u, will always be over threshold if it is initialized with at least

$$\left\lceil \frac{s_v + d_v - s_u + y_v}{y_u} \right\rceil \cdot x_u \cdot cns(q) + thr(q)$$

tokens where

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

and

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

Using these expressions and the RBE parameters listed in Table 5.6 to compute the number of initial tokens on the queue connecting nodes $v =$*MstrMCS* and $u =$*BDF*, the 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

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

and

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

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

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

and

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

The original implementation of the DIFAR graph (on custom militarized hardware) 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

Node	t_u	(x_u,	y_u)
Source	0	(16,	625ms)
FlowCntl	0	(1,	1250ms)
BDF	0	(1,	1250ms)
MstrMCS	0	(1,	1250ms)
MnsMrg	0	(0,	1250ms)
MnsOut	0	(0,	1250ms)
SlvMCS	0	(1,	1250ms)
DDAD	0	(1,	1250ms)
CRfilter	0	(1,	1250ms)
CRspec	0	(1,	1250ms)
CRdetect	0	(1,	2500ms)
BndMrg	0	(2,	2500ms)
SAD	0	(2,	2500ms)
GramData	0	(2,	2500ms)
GramMrg	0	(2,	2500ms)
GramOut	0	(2,	2500ms)
AliScale	0	(1,	10000ms)
AliMrg	0	(1,	10000ms)
AliOut	0	(1,	10000ms)
BBC	0	(2,	2500ms)
BrgAngle	0	(1,	10000ms)
BrgMrg	0	(1,	10000ms)
BrgOut	0	(1,	10000ms)
AutDet	0	(1,	30000ms)
AutDetMrg	0	(1,	30000ms)
AutDetOut	0	(1,	30000ms)
BinMrg	0	(1,	30000ms)
BinOut	0	(1,	30000ms)
VernFilter	0	(0,	1250ms)
VernSpec	0	(0,	1250ms)
VernDet	0	(0,	1250ms)

Table 5.5: Well-defined execution rates for each node in the DIFAR graph for the CR mode of operation.

executing at the same time. 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 our synthesis method, the DIFAR application can be guaranteed to always meet is latency requirement.

5.4.2 Step 2: Map Nodes to Tasks in the RBE Model

Table 5.6 lists the RBE parameters associated with each node when it is mapped to an RBE task. Parameters x_u and y_u are as derived in the rate computation step. Parameter d_u is set to y_u for each node u in the graph. Parameter e_u is the worst-case execution time for node u on a 200MHz PowerPC processor based on cycle counts for each processing function, assuming all memory references result in cache misses and worst-case processing paths within each processing function. As shown in the next section, this mapping will be sufficient to guarantee that the application will meet its latency requirement.

5.4.3 Step 3: Verify Schedulability

The third step of the synthesis method is to verify that the resulting task set is schedulable so that we can guarantee real-time execution. By Theorem 3.3.5, the RBE task set constructed from the DIFAR graph is schedulable using RBE-EDF scheduling if an affirmative result is obtained when the following scheduling condition is evaluated:

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

where $f(a)$ is the floor function defined in Lemma 3.3.1. Since $d_u = y_u$ for every node u in the graph, we again use the simpler utilization expression

$$\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} \leq 1 \tag{5.5}$$

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

Table 5.6: RBE parameters associated with each node in the DIFAR graph for the CR mode of operation. For each node u in the graph, $d_u = y_u$.

to evaluate the schedulability of the graph under RBE-EDF scheduling. Using the RBE parameters from Table 5.6, we see that the graph is schedulable since

$$\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} = .0639$$

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

If $d_u = y_u$ for each node u and the graph is not schedulable, then relaxing any of the deadline parameters will not change the schedulability of the graph since increasing deadline parameters in this case does not reduce utilization. A negative result from Equation (5.5) when $d_u \geq y_u$ means that the processor is over loaded (i.e., the processor utilization is greater than 100%).

5.4.4 Computing Latency

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

By Theorem 4.4.5, the latency between the time the first sample arrives and when sink node *AliOut* executes is less than

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

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

The maximum latency the first sample encounters in the path from node *Source* to each of the other output nodes is computed in the same manner. Using the RBE parameters in Table 5.6, the maximum latency from node *Source* to node:

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

At first it is rather surprising that latency as high as 60 seconds is tolerable in an embedded application. However, it is the case that acoustic signal processing applications can tolerate much higher latency bounds than 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.

5.5 Discussion and Summary

We have demonstrated the successful synthesis of three signal processing systems from PGM graphs with topologies ranging from a simple chain to a cyclic graph. While the theory supporting our synthesis method and latency analyses is general enough to handle any PGM graph, in our experience most PGM graphs created for signal processing applications have dataflow attributes and topologies similar to one of the three analyzed here. Moreover, in most signal processing graphs, the produce or consume value of a queue are typically multiples of each other. When this is not the case, it is usually because a mistake was made by the signal processing engineer that created the graph. Thus, our synthesis method can also be used identify inadvertent coding errors.

Prior to this work, the two most common ways to schedule PGM graphs was with a dynamic non-preemptive FCFS scheduler or with a static non-preemptive schedule created off-line. In the case of FCFS scheduling, the execution order of nodes is indeterminate and latency requirements can not be guaranteed. Instead, long simulations are executed to estimate worst case latency values. In the case of static scheduling, latency requirements can be guaranteed, but latency requirements must be evaluated with every schedule created. In complicated systems, such as the ALFS subsystem, many different graphs are executed simultaneously and the the type of processing performed changes dynamically. This can result in the need for hundreds of different static schedules to be created and stored. For example, one of the worst case concurrency modes for the ALFS subsystem requires the simultaneous execution of 24 different graphs, and each graph can execute in several different modes of operation. In these types of systems, our dynamic scheduling technique has a clear advantage over static scheduling since only the schedulability analysis needs to be performed off-line. For any schedulable combination of graphs, the RBE-EDF scheduler will dynamically create a valid schedule in which no deadline is missed. The maximum latency any signal encounters only needs to be evaluated once for each path from a source node to a sink node and not for every possible schedule.

Finally, if every node has its deadline parameter equal to the y parameter of its rate specification, the utilization expression

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

can even be used as an efficient on-line admission control test for RBE-EDF scheduling. When a new graph is added, processor utilization can be calculated in time polynomial in the number of nodes in the new graph to see if the system is still schedulable with the additional graph.

Chapter 6

Contributions and Conclusions

6.1 Summary

The primary problem in developing embedded signal processing systems with a processing graph methodology is transforming a processing graph into a predictable real-time system in which latency and memory usage can be managed. In this dissertation, we combined software engineering techniques with real-time scheduling theory to create a synthesis method that solves this problem. We also demonstrated how the processing latency of signals can be managed using our synthesis method.

Our synthesis method for building deterministic signal processing systems from PGM graphs involves three steps:

1. Identification of the rate $R_v = (x_v, y_v)$ at which each node v in a PGM graph $G = (V, E, \psi)$ must execute if it is to process the signal in real-time. A rate specification of $R_v = (x_v, y_v)$ means that node v must execute x_v times every y_v time units. Our algorithms for computing node execution rates perform a topological sort of the graph and set the rates in topological order such that

$$y_v = \mathrm{lcm}\{\frac{cns(q) \cdot y_u}{\gcd(prd(q) \cdot x_u, cns(q))} \mid \forall q \in E \wedge \forall u \in V : \psi(q) = (u, v)\},$$
$$x_v = y_v \cdot \frac{prd(q) \cdot x_u}{cns(q) \cdot y_u}.$$

   For cycles, the topological sort creates a partial ordering of the graph, and the partial order must be processed $\max(1, n)$ times where n is the maximum number of different cycles in which a given node appears.

2. Construction of a mapping of each node v to a task in the RBE task model. Each node is associated with a four-tuple (x_v, y_v, d_v, e_v), where the parameters x_v and y_v are the execution rates computed in the first step, and the parameter e_v is the execution time of node v. The only free parameter in the mapping is the relative deadline parameter d_v. This parameter is used to guarantee real-time execution and to manage latency in the implementation.

   If the resulting RBE task set is schedulable, then our RBE-EDF scheduling algorithm will schedule an execution such that the j^{th} release of task T_i at time $t_{i,j}$ is guaranteed to complete execution by time $D_i(j)$, where

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

   The RBE task model makes no assumptions regarding when a task will be released. For an RBE task set synthesized from a PGM graph, a task is released when all of its input queues are over threshold, and there will be exactly x_i releases of task T_i in any interval of y_i time units. However, we do not know, when in an interval of y_i time units, any of the x_i releases will occur.

3. Verification that the resulting task set is schedulable by evaluating the expression

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

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

   so that real-time execution can be guaranteed.

If the task set is not schedulable, the mapping of nodes to RBE tasks and the verification of schedulability must be repeated with a new set of deadline parameters. However, as long as the processor utilization is less than or equal to one, there exists a set of deadline parameters that are guaranteed to be schedulable. One such set of deadline parameters is the set created by setting $d_i = y_i$ for each task T_i in the task set.

We identified and bounded two sources of latency in real-time systems created from processing graphs. The first, *inherent latency*, is latency defined by the dataflow attributes and topology of the processing graph. The second source of latency, *imposed* *latency*, comes from the scheduling and execution of nodes in the graph. Thus, the total latency any sample encounters can be expressed with the equation

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

We developed a framework for evaluating and managing latency by deriving upper and lower bounds for both types of latency as functions of the data flow attributes and graph topology. These bounds are summarized in Table 6.1. The inherent latency portion of each latency bound can be used by signal processing engineers in selecting dataflow attributes for queues. No matter how the graph is implemented, the latency a sample encounters will never be less than its inherent latency. For example, consider a PGM chain $u \rightsquigarrow w$ where node u is a periodic source with $R_u = (1, y_u)$. No matter how the chain is implemented, the latency a sample encounters will never be less than

$$\max\left(0, (F_{u \rightsquigarrow w} - 1) \cdot y_u\right).$$

If the inherent latency is too large, the signal processing engineer must choose new dataflow attributes for the graph edges.

To illustrate our techniques for managing latency, three PGM graphs from the literature and industry were analyzed. Each graph corresponds to a non-trivial embedded real-time signal processing system. The first was a SAR radar application, which has the topology of chain. The second application evaluated was an INMARSAT mobile satellite receiver application. The acyclic PGM graph for this application has a more complicated topology than the simple chain of the SAR graph, as well as multiple source nodes. We concluded our case studies with an evaluation of latency in the cyclic DIFAR graph from the ALFS subsystem of the LAMPS MK III anti-submarine helicopters. Most signal processing graphs have topologies that closely resemble one of the three applications evaluated in this dissertation.

Although the results of this dissertation are based on PGM graphs, they are fundamental to AND processing graph models. Thus, our results are applicable to systems developed with other processing graph notations such as Lee and Messerschmitt's Synchronous Dataflow (SDF) graphs [41] or Chatterjee and Strosnider's Logical Application Stream Model (LASM) [15, 16].

Latency in a Chain

Source	Lower Bounds	Upper Bounds
Periodic	$\max\left(0, (F_{j \rightsquigarrow w} - 1) \cdot y_u\right) + \sum_{v \in \{j \rightsquigarrow w\}} e_v$	$\max\left(0, (F_{j \rightsquigarrow w} - 1) \cdot y_u\right) + d_w$
RBE	$\max\left(0, \left\lfloor \frac{F_{j \rightsquigarrow w} - 1}{x_i} \right\rfloor \cdot y_j\right) + \sum_{v \in \{j \rightsquigarrow w\}} e_v$	$\max\left(1, \left\lceil \frac{F_{j \rightsquigarrow w}}{x_j} \right\rceil \cdot y_j\right) + d_w$

Latency in Acyclic Graphs

Sources	Lower Bounds	Upper Bounds
Periodic	$\mathcal{L}_w(t) + \sum\limits_{v \in \mathcal{P}_w} e_v$	$\mathcal{L}_w(t) + d_w$
RBE	$\max\limits_{p \in \mathcal{P}} \left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j\right) + \sum\limits_{v \in \mathcal{P}_w} e_v$	$\max\limits_{p \in \mathcal{P}} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j\right) + d_w$

Latency in Cyclic Graphs

Sources	Lower Bounds	Upper Bounds
Periodic	$\mathcal{L}_w(t) + \sum\limits_{v \in \hat{\mathcal{P}}_w} e_v$	$\mathcal{L}_w(t) + d_w$
RBE	$\max\limits_{p \in \hat{\mathcal{P}}} \left(0, \left\lfloor \frac{F_p - 1}{x_j} \right\rfloor \cdot y_j\right) + \sum\limits_{v \in \hat{\mathcal{P}}_w} e_v$	$\max\limits_{p \in \hat{\mathcal{P}}} \left(1, \left\lceil \frac{F_p}{x_j} \right\rceil \cdot y_j\right) + d_w$

Table 6.1: A summary of latency bounds. Function F_p returns the number of executions of the source node in the path p required before node w is eligible for execution. Function $\mathcal{L}_w(t)$ returns the latency a sample produced at time t encounters before node w is eligible for execution when multiple periodic source nodes have paths that lead to node w. $\mathcal{P}$ denotes the set of all paths from source nodes $j \in \mathcal{I}_w$ to node w. $\mathcal{P}_w$ denotes the path from source node j to node w that creates the maximum inherent latency for the sample. Function $\hat{\mathcal{L}}_w(t)$ returns the latency a sample produced at time t encounters before node w is eligible for execution when the graph is cyclic. $\hat{\mathcal{P}}$ denotes the set of all acyclic paths from source nodes $j \in \mathcal{I}_w$ to node w. $\hat{\mathcal{P}}_w$ denotes the acyclic path from source node j to node w that creates the maximum inherent latency for the sample.

6.2 Contributions

Prior to this work, the two most common ways to execute PGM graphs were with a dynamic, non-preemptive FCFS scheduler or with a static, non-preemptive schedule created off-line. In the case of FCFS scheduling, the execution order of nodes is indeterminate and latency requirements can not be guaranteed. Because of this, long simulations and test scenarios are required to estimate worst case latency values. In the case of static scheduling, latency requirements can be guaranteed, but they must be evaluated for every schedule created. Moreover, in complicated systems with multiple graphs and many modes of operations, static scheduling may require creating and storing hundreds of static schedules. In contrast, for any schedulable combination of graphs, our RBE-EDF scheduler will dynamically create a valid schedule in which no deadline is missed.

From the real-time literature, the two most common models of real-time execution are the periodic and sporadic task models. The advantage of modeling node execution as a task in the RBE model rather than as a task in a periodic or sporadic model is that less latency is imposed when the node is allowed to execute according to its rate specification rather than when execution is forced to follow a periodic or sporadic execution model. Our research is the first to model the execution of processing graphs with the RBE model.

This work appears to be the first to identify and quantify inherent latency in processing graphs. Using our synthesis method, the maximum imposed latency a sample encounters before it is processed by node w is bounded by RBE parameter d_w. This is the only free parameter available to control the total latency a sample encounters.

6.3 Future Work

There are several interesting issues in the synthesis of embedded real-time systems from PGM graphs that are still unresolved. The research results presented here also touch on areas of research in other application domains that should be explored in greater detail.

6.3.1 Further PGM Analysis

There are two major problems left open in this dissertation. The first is managing memory requirements in the synthesis of real-time systems from PGM graphs, and the second is synthesizing a distributed real-time system from PGM graphs.

6.3.1.1 Managing Memory Requirements

In the past, environmental restrictions on size, weight, and power consumption have severely limited both the processing and storage capacity of embedded signal processing systems. Today, however, as increases in processor speed and capabilities continually out-pace increases in memory densities and performance, processor capacity is less of a problem for many signal processing applications, and memory usage is becoming a primary concern.

There are two aspects to memory analysis: code and data storage requirements. Optimizing compilers and efficiently written code can help to minimize code space, but processing graphs also require storage space for intermediate processing results temporarily stored on the graph edges (i.e., space required to buffer tokens). The space required to hold the intermediate results on all graph edges simultaneously can be quite substantial. Recently, we have begun work on extending the results presented here to bound and manage memory usage in signal processing systems synthesized from PGM graphs [28, 27, 29].

Once the processing graph has been created, the only free variable in controlling the amount of data buffered on an edge is the execution relationship between producer and consumer nodes. The canonical approach to minimizing memory requirements for the graph edges is to use static scheduling. Static node execution schedules are created off-line and then executed on a periodic basis. Numerous static scheduling algorithms have been created to minimize memory requirements [41, 54, 64, 55, 9]. The primary trade-off made by static schedulers is the storage requirement and execution complexity of the schedule vs. the storage requirement of data in the graph edges. Typically, achieving minimal buffer space requires increased state space for the scheduler. Some scheduling algorithms produce a simple flat schedule with each entry in the schedule representing the execution of a single node. Other algorithms save scheduler state space by associating a number of executions with each scheduler entry to reduce state space for multiple executions of the same node. Still other scheduling algorithms produce slightly more complicated schedules by creating scheduling loops encompassing many scheduling entries [9]. For example, consider the chain in Figure 6.1. (Nodes u and o represent external devices that are not scheduled, and only nodes v and w appear in execution schedules for this graph.) Three possible schedules for the chain in Figure 6.1 are:

- $vwvwvww$ — *a multiple appearance flat schedule* where the same node appears multiple times in a list;

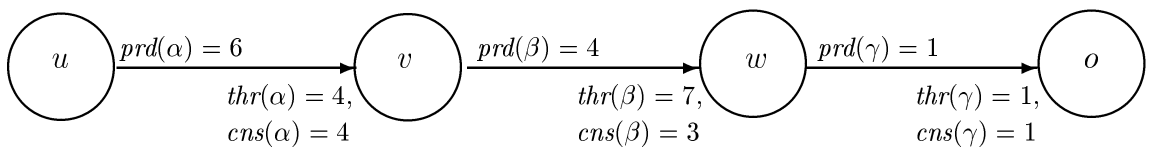


Figure 6.1: A four-node chain. Nodes u and o represent external devices. Thus, only nodes v and w are scheduled.

- $(3v)(4w)$ — *a single appearance flat schedule* where a node appears once in the list, but the node may be executed multiple times before the next node in the list is executed;

- $(3vw)w$ — *a multiple appearance looped schedule* [9], which supports scheduling loops that consist of multiple nodes. The notation in the looped schedule is such that the 3 applies to all subsequent nodes until the right parenthesis is reached. In this case, the schedule $(3vw)w$ produces identical execution results as the multiple appearance flat schedule $vwvwvww$.

The buffer space required by static schedulers is dependent on the particular scheduling algorithm. For example, the multiple appearance flat schedule $vwvwvww$ or the multiple appearance looped schedule $(3vw)w$ requires storage for 10 tokens on queue β, while the single appearance flat schedule $3v4w$ requires storage for 16 tokens on queue β. To derive the storage requirement of the input queue α, we need to know when the static schedule is executed.

Static schedules are usually executed on a periodic basis with a timer indicating when to start executing the schedule. Since the schedule executes without inserting idle time (once it starts), execution cannot begin until enough data has accumulated on the input queues to the graph to ensure a valid graph execution. The minimal period of a static schedule is equal to the maximum y value of the execution rates of the set of output nodes $\mathcal{O}$. For example, the period of a static schedule created for the 2 schedulable nodes in Figure 6.1 is y_o since node o is the only graph output node. The first execution of the static schedule cannot begin before time S where S is bounded such that $s_o \leq S \leq s_o + y_o$, and s_o is the logical release time of the first execution of node o. If the graph source node is periodic, s_o is computed using Equation (4.15) on page 142. For example, let source node u execute periodically with a well-defined rate specification of $R_u = (1, 8)$. The execution rates for nodes v, w, and o in Figure 6.1 are $R_v = (3, 16)$, $R_w = (4, 16)$, and $R_o = (4, 16)$. Evaluating Equation (4.15) yields, $s_o = 8$. Since $y_o = 16$, the scheduling period is 16 time units. To ensure data availability for each execution of node v, the schedule cannot begin until time 8 (the first logical release time of node o, which is also the first logical release time of node w). Hence, with $prd(\alpha) = 6$, 12 tokens accumulate on the input queue α before the schedule even begins (assuming source node u first produces at time 0). Depending on the schedule, node execution times, and the execution characteristics of the source node, more data may accumulate during the execution of the schedule. Therefore, buffering for at least $12 + 16 = 28$ tokens are required for queues α and β when a single appearance flat schedule is used to execute the two nodes. When deriving buffer requirements for queues connected to external sink devices, we assume that the external device consumes data as soon as it is available. Since node w produces one token every time it executes, a statically scheduled implementation of the graph requires storage for at least 29 tokens (and possibly more). It is important to recognize that depending on the length of the scheduling period and when the static schedule first starts, the amount of data that accumulates on the queues attached to source nodes in an actual application can be quite substantial.

In contrast to static scheduling techniques, our approach is to use a simple, dynamic, on-line scheduler for graph execution. With today's processors, it is possible to use dynamic, on-line scheduling to achieve near optimal memory usage and not be concerned with minimizing the on-line scheduling overhead. Moreover, and somewhat surprisingly, dynamic scheduling often requires less memory than static schedules created by off-line schedulers designed to minimize memory usage. For example, we have recently shown that state-of-the-art static schedulers require almost 300% more memory for buffering tokens on graph queues than our dynamic scheduling approach [29]. The primary reason dynamic scheduling can use so much less memory for buffering tokens is that nodes are able to execute as soon as they are eligible. Thus, we don't have a large accumulation of data on the queues attached to source nodes, which is what happens when static scheduling is used.

The open issue in managing memory requirements using our synthesis method is developing tight bounds on queue lengths for general graphs. We have identified common special cases, for example chains of nodes or when produce and consume values are multiples of the other, for which we are able to derive tight buffer bounds [28, 27, 29], but we have not yet derived a satisfactory upper bound for all cases.

6.3.1.2 Distributed Systems

Some of the interesting and important open problems in synthesizing a distributed real-time system from PGM graphs are: allocating graphs to different processors in the distributed system, evaluating latency over multiple processors, scheduling rate-based traffic on the network, and non-uniform communication costs.

The problem of evaluating latency over multiple processors is perhaps the easiest to solve, assuming the latency a sample encounters in the network can be bounded. However, even within this problem, interesting issues arise. What happens if a graph cycle spans multiple processors? Do processor clocks need to be synchronized to bound latency? What are the network requirements needed to bound latency in the network?

Since nodes generate data at well-defined rates, the data traffic on the network in a distributed signal processing system built from PGM graphs will also be rate-based. If the system has a dedicated network, as is usually the case in embedded systems, do results from traditional network scheduling theory apply to rate-based traffic? Can we get better utilize the network if all traffic is rate-based, or does this compound the problem? Does rate-based traffic, make it easier to bound latency in the network? What type of communications protocol will best support rate-based traffic? Is a reservation-based protocol necessary to guarantee latency?

In practice, communication costs are non-uniform. By communications costs, we mean the time to either send a message or to access (shared) memory. How does this affect latency analysis or schedulability analysis? Must all queue data be stored in local processor memory? What happens if data is stored in a shared repository on a remote processor, as is sometimes the case in embedded systems?

Probably the most important open problem with respect to synthesizing a distributed system from a PGM graph is determining how to allocate nodes to processors. Algorithms ranging from traditional graph partitioning algorithms to simulated annealing have been used to allocate nodes to processors with other graph models. How well do these algorithms work with PGM graphs? Is there one algorithm that works best for partitioning PGM graphs over a distributed system, or do different algorithms work better for particular types of graphs?

6.3.2 Other Application Domains

The results of this dissertation were driven by open problems we encountered in embedded signal processing applications. However, our results are also applicable to other application domains. For example, our synthesis method and latency analysis can also be applied to multimedia applications developed with Chatterjee and Strosnider's LASM system design method [15, 16]. An open problem in doing so, however, is jitter management. In multimedia applications, jitter is defined as variation in the time required to acquire, process, transmit and display media samples. Most jitter management algorithms rely on periodic (or sporadic) execution models. It is not clear how rate-based execution affects existing jitter management algorithms, or whether they are even applicable. This is somewhat surprising since the RBE model was originally created to more accurately model the work load created by multimedia applications.

The RBE model was used extensively in this dissertation, but the model itself seems widely applicable to general real-time systems. In practice, allowing a task to execute with an expected execution rate is much easier to work with than trying to force it into a periodic or sporadic execution, especially when precedence constraints exists. However, there remain unresolved issues in the RBE model. Schedulability conditions must be developed that account for blocking time that occurs when resources are shared. A schedulability condition for non-preemptive scheduling is also needed. Perhaps the most important unresolved problem in the RBE model is limiting the number of releases for a task that can occur in an interval of time, and then accounting for this overhead.

6.4 Conclusions

From this research, we conclude that predictable, real-time systems can be synthesized from PGM graphs. Moreover, dynamic scheduling can be used to manage latency (and memory requirements) in an implementation of the graph. While we recognize that our synthesis method is not a panacea for building real-time systems from processing graphs, we have demonstrated that it does provide a viable option to the static or FCFS scheduling techniques commonly used today. For some applications, our dynamic scheduling approach is far better than existing options for managing latency (and memory requirements). We believe that, for very simple applications, static scheduling provides adequate latency bounds and should probably be chosen over our synthesis method for implementing a graph. However, for complicated signal processing applications with multiple modes of operations and hard-real-time processing requirements, we believe our synthesis method should be used over static or FCFS scheduling. Our synthesis method guarantees real-time execution without storing a schedule for every possible combination of graphs and modes of operation.

Our experience developing embedded systems with PGM indicates that integrating the results of this dissertation into the development environment will improve the software engineering process. Our algorithms for computing node execution rates and inherent latency can be used to identify dataflow problems and violations of latency requirements early in the design process when such problems are easiest to correct. For example, we have found that large increases in the y parameter of the rate specification for a node (with respect to the rate specifications of its producers) often indicates processing in the graph that should be investigated. There are two common cases in which the large increase occurs:

1. a filter that processes samples from an extended time interval, and
2. the produce and consume values on a queue are (nearly) relatively prime.

The first case is usually not an error, but its impact on inherent latency should be noted. As for the second case, it is reasonable for the produce and consume values on a queue may be relatively prime when the period of the external source device does not match the rate at which an external output device can receive data and a rate conversion must occur in the graph. In almost all other cases, however, relatively prime produce and consume values indicate an unintentional rate change — usually caused by cut and paste editing of graph source code — that should be corrected.

It seems that most embedded signal processing applications have latency requirements that are twice the application's maximum inherent latency value. The reason for this is not clear, but it appears to be common practice to set the latency requirement to twice the longest time interval over which data is integrated — and the longest time interval over which data is integrated creates the maximum inherent latency in the application. The result of this practice is that setting the deadline parameter d_v equal to y_v for node v when nodes are mapped to RBE tasks almost always results in a mapping that meets the latency requirement if the graph is schedulable. Moreover, this means that the simple schedulability condition

$$\sum_{i=1}^{n} \frac{x_i \cdot e_i}{y_i} \leq 1,$$

which measures processor utilization, can be used as an efficient on-line admission control test for RBE-EDF scheduling. An on-line admission control test is useful when multiple applications supporting different modes of operation are executed simultaneously on the processor. For example, the U.S. Navy tries to anticipate all modes of operations in which a signal processing system must perform, but it recognizes that on-board operators occasionally need to improvise. Thus, the availability of an efficient on-line admission control test is important when unanticipated combinations of graphs or modes of operation are needed due to ever changing tactical situations.

Bibliography

[1] Ackerman, W.B., "Data flow languages," *COMPUTER*, Vol. 15, February 1982.

[2] Anderson, D.P., Tzou, S.Y., Wahbe, R., Govindan, R., Andrews, M., "Support for Live Digital Audio and Video," *Proc. of the Tenth International Conference on Distributed Computing Systems*, Paris, France, May 1990, pp. 54-61.

[3] Barnes, K.B., et al., "A Data Flow Graph Programming Environment for Embedded Multiprocessing", Technical Report, GE Advanced Technology Laboratories, Moorestown, New Jersey, 1992.

[4] Baruah, S., Personal communication, March 1996.

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

[6] Baruah, S., Howell, R., Rosier, L., "Algorithms and Complexity Concerning the Preemptively Scheduling of Periodic, Real-Time Tasks on One Processor," *Real-Time Systems Journal*, Vol. 2, 1990, pp. 301-324.

[7] Baruah, S., Howell, R., Rosier, L., "Feasibility Problems For Recurring Tasks On One Processor," *Theoretical Computer Science*, 118:3-20, 1993.

[8] Baruah, S., Mok, A., Rosier, L., "Preemptively Scheduling Hard-Real-Time Sporadic Tasks With One Processor" *Proc. 11th IEEE Real-Time Systems Symp.*, Lake Buena Vista, FL, Dec. 1990, pp. 182-190.

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

[10] Bettati, R., Liu, J., "End-to-End Scheduling to Meet Deadlines in Distributed Systems," *Proceedings of the 12th International Conference on Distributed Computing Systems*, Yokohama, Japan, June 1992, pp. 452-459.

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

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

[13] 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.

[14] Cormen, T.H., Leiserson, C.E., Rivest, R.L., *Introduction to Algorithms*, McGraw-Hill, 1990.

[15] 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.

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

[17] *DataFlo User's Guide*, Axiom Technology Inc., version 3.5, February 1, 1996.

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

[19] *DSPView User's Manual*, Lucent Technologies Inc., version 0.9, May 1996.

[20] Evans, J.D., Kessler, R.R., "A Communication-Ordered Task Graph Allocation Algorithm", Technical Report UUCS-92-026, University of Utah, Department of Computer Science, April 1992.

[21] 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.

[22] Gerber, R., Seongsoo, H., Saksena, M., "Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes", *Proc. of IEEE Real-Time Systems Symposium*, Dec. 1994.

[23] 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.

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

[25] Goddard, S., Unpublished performance study, S.M. Goddard & Co., Inc., under contract to Bell Labs, July 1996.

[26] 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.

[27] Goddard, S., Jeffay, K. "Analyzing the Real-Time Properties of a Dataflow Execution Paradigm using a Synthetic Aperture Radar Application," Technical Report TR97-007, Dept. of Computer Science, University of North Carolina at Chapel Hill, April 1997.

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

[29] 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

[30] Hillson, R., "Support Tools for the Processing Graph Method," *Proceedings of the Fifth International Conference in Signal Processing Applications and Technology [ICPAT 94]*, Dallas, TX, October 1994, pp. 756-761.

[31] Jeffay, K., *The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations*, Ph.D. Dissertation, University of Washington, Department of Computer Science and Engineering, TR-89-09-15, September 1989.

[32] Jeffay, K., "Scheduling Sporadic Tasks with Shared Resources in Hard Real-Time Systems", *Proceedings of the* 13^{th} *IEEE Symposium on Real-Time Systems*, Phoenix, AZ, 1992, pp. 89-99.

[33] Jeffay, K., "The Real-Time Producer/Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems," *Proc. of the ACM/ SIGAPP Symposium on Applied Computing*, Indianapolis, IN, February 1993, pp. 796-804.

[34] Jeffay, K., "On Latency Management in Time-Shared Operating Systems," *Proc. of the* 11^{th} *IEEE Workshop on Real-Time Operating Systems and Software*, Seattle, WA, May 1994, pp. 86-90.

[35] Jeffay, K., Bennett, D. "A Rate-Based Execution Abstraction For Multimedia Computing," Proc. of the Fifth Intl. Workshop on Network and Operating System Support for Digital Audio and Video, Durham, NH, April 1995, published in *Lecture Notes in Computer Science*, T.D.C. Little and R. Gusella eds., Vol. 1018, Springer-Verlag, Heidelberg, Germany, 1995, pp. 65-75.

[36] Jeffay, K., Stone, D., "Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems", *Proc. of the* 14^{th} *IEEE Symposium on Real-Time Systems*, Durham, NC, 1993, pp. 212-221.

[37] Jeffay, K., Stone, D.L., Smith, F.D., "Kernel Support for Live Digital Audio and Video", *Computer Communications*, Vol. 15, No. 6, July/August 1992, pp. 388-395.

[38] Johnson, T., "A Concurrent Dynamic Task Graph," *Proc. Intl Conf. on Parallel Processing*, 1993.

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

[40] Lee, B., Hurson, A.R., "Dataflow Architectures and Multithreading," *COMPUTER*, August 1994, pp. 27-39.

[41] Lee, E.A., Messerschmitt, D.G., "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, Vol. C-36, No. 1, January 1987, pp. 24-35.

[42] Lehoczky, J., Sha, L., Ding, Y., "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior", *Proceedings of the Tenth IEEE Real-Time Systems Symposium*, Santa Monica, CA, 1989, pp. 166-171.

[43] Leung, J., Merrill, M., "A Note On The Preemptive Scheduling of Periodic, Real-Time Tasks", *Information Processing Letters*, Vol. 11, 1980, pp. 115-118.

[44] Leung, J., Whitehead, J., "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks", *Performance Evaluation*, Vol. 2, No. 4, 1982, pp. 237-250.

[45] Liu, C., Layland, J., "Scheduling Algorithms for multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM*, Vol 30., Jan. 1973, pp. 46-61.

[46] Mok, A.K., *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, Ph.D. Dissertation, MIT, Department of EE and CS, MIT/LCS/TR-297, May 1983.

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

[48] Mok, A. K., et al., "Synthesis of a Real-Time System with Data-driven Timing Constraints," *Proc. of the IEEE Real-Time Systems Symposium*, San Jose, CA, Dec. 1987, pp. 133-143.

[49] Novak, L.M., Owirka, G.J., Netishen, C.M., "Performance of a high-resolution polarimetric SAR automatic target recognition system," M.I.T. Lincoln Laboratory Journal, Vo. 6, No. 1, 1991, pp. 11-25.

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

[51] *Reference Manual for the Processing Graph Support Environment: Parts 1-4*, prepared by the Hughes Aircraft Ground Systems Group for the Naval Research Laboratory, April 20, 1992.

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

[53] *RIPPEN User's Guide*, ORINCON Technologies, Inc., Rev 2.0, June, 1996.

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

[55] 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.

[56] Rosen, K., *Elementary Number Theory and its Applications*, Addison Wesley Publishing Company, 1988.

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

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

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

[60] Stevens, R.S., Kaplan, D.J., "Determinacy of Generalized Schema," *IEEE Transactions on Computers*, Vol. 41, No. 6, 1992, pp. 776-779.

[61] Stone, D., *Managing the Effect of Delay Jitter on the Display of Live Continuous Media*, Doctoral Dissertation, University of North Carolina, Chapel Hill, 1995.

[62] Ramos-Thuel, S., Lehoczky, J., "On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed Priority Systems", *Proceedings of the* 14^{th} *IEEE Symposium on Real-Time Systems*, Durham, NC, 1993, pp. 160-171.

[63] Woo, N.S., Smith, C.H., Agrawala, A., "A proof of the determinacy property of the data flow schema," *Inform. Processing Lett.*, Vol. 19, pp. 13-16, 1984.

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

[65] Zuerndorfer, B., Shaw, G.A., "SAR Processing for RASSP Application," *Proc. of 1^{st} Annual RASSP Conference*, Arlington, VA, August 15-18, 1994.