Distributed Real-Time Dataflow: An Execution Paradigm for Image Processing and Anti-Submarine Warfare Applications

S. Goddard K. Jeffay
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

Abstract

The central **thesis** of this project is that

> *real-time scheduling theory can be combined with dataflow methodologies to bound latency and memory utilization of distributed signal processing applications, such as those found in anti-submarine warfare and image processing.*

To this end, we propose a new real-time dataflow paradigm that is based on the Navy's *Processing Graph Method* (*PGM*) [5], which is similar to the dataflow methodology employed by Ptolemy [1, 4] of the University of California at Berkeley and the Rapid Prototyping of Application Specific Signal Processors (RASSP) project, funded by the Advanced Research Projects Agency (ARPA). While our dataflow methodology itself is not novel, our application of real-time scheduling theory to the model is.

1 Introduction

Signal processing algorithms are often defined in the literature using large grain dataflow graphs [4]: directed graphs in which a collection of sequential work that executes from start to finish without synchronization is called a node or vertex, and the graph edges (arcs) depict the asynchronous flow of data from one node to the next. Thus, an edge represents a producer/consumer relationship between two nodes. Large grain dataflow provides a natural description of signal processing applications with each node representing a mathematical function to be performed on an infinite stream of data that flows on the arcs of the graph. The dataflow methodology allows one to easily understand a signal processing application by depicting the structure of the algorithm; any portion of the application can be understood in the absence of the rest of the algorithm. Each node consumes the data produced by its predecessor in the graph and the data exists only between its production and consumption.

Dataflow is used by at least two government sponsored programs to develop embedded signal processing applications. The Rapid Prototyping of Application Specific Signal Processors (RASSP) project, funded by the Advanced Research Projects Agency (ARPA), uses dataflow techniques to specify and develop image processing benchmark applications. One of the primary benchmarks they have developed is that of synthetic aperture radar (SAR). SAR is used to identify man-made objects on the ground or in the air by producing high-resolution, all-weather images, which are usually processed in real-time. Dataflow methodology is also used by the Navy's standard signal processor, the AN/UYS-2A — used in detection of submarines. The Navy has spent millions of dollars developing the AN/UYS-2A and applications for it such as the Airborne Low Frequency Sonar (ALFS) subsystem of the Block II upgrade to the LAMPS MK III helicopter, and yet it has no way to guarantee that the hard real-time processing requirements of these anti-submarine warfare applications can be met.

Embedded signal processing applications are naturally defined using dataflow techniques, but require the deterministic performance of real-time applications. The signal processing graph must process data at the rates of a set of producers (e.g., sonobuoys, dipping sonars, or radars) without the loss of data. Hence, these applications have a dual notion of correctness: logical and temporal. It is not sufficient to produce the correct output, which may be the signature of a detected target (logical correctness); embedded signal processing applications must produce the correct output within the correct time interval (temporal correctness). Hard-real-time dataflow systems provide temporal correctness without loss of data while streams of input data arrive at rates defined by the external world. Soft-real-time systems may use buffering schemes to prevent data loss during transient over loads. The system then catches up during light load intervals. There are many products on the market that aid in the development of real-time dataflow applications (e.g., DSPView, DataFlow, RIPPEN, PGSE, GEDAE), but none of these provide support for analysis of the hard-real-time semantics of the system.

Dataflow models implicitly define a temporal seman-

In: Proceedings of the IEEE Real-Time Systems Symposium Work in Progress Abstracts, Washington, DC, December 1996, pages 55-58.

tics of a processing graph by specifying lower bounds on when nodes may execute as a function of the availability of data on input edges. However, most models do not specify an upper bound on the time that may elapse between when a node becomes eligible to execute and the time the node either commences or completes execution.

The lack of such a specification creates several problems:

- the lack of a feasibility or admission control test — Does a set of nodes or a graph "fit" on a processor?
- undetermined latency properties — Does a graph meet timing requirements?
- no upper bound on queue length — If latency isn't bounded, memory requirements for a graph cannot be bounded and hence data loss may occur.

System engineers use these metrics to size hardware and perform requirements verification. A cost tradeoff may be made on CPU utilization versus latency, or buffer space versus latency. High latency tolerances allow the use of a slower (and cheaper) CPU but may require more memory for increased buffer space. On the other hand, tighter latency requirements may demand a faster CPU (or lower utilization) but less memory. In keeping costs in line, a system architect uses these metrics to make fundamental design tradeoffs.

None of the dataflow models or real-time execution paradigms in the research literature correctly model the execution of both SAR and ALFS applications. A new real-time dataflow execution paradigm is needed that supports these distributed applications and the fundamental tradeoffs made by system engineers. The central **thesis** of this project is that

> *real-time scheduling theory can be combined with dataflow methodologies to bound latency and memory utilization of distributed signal processing applications, such as those found in anti-submarine warfare and image processing.*

To this end, we propose a new real-time dataflow paradigm that is based on the Navy's *Processing Graph Method* (*PGM*) [5], which is similar to the dataflow methodology employed by Ptolemy [1, 4] (of the University of California at Berkeley) and RASSP. While our dataflow methodology itself is not novel, our application of real-time scheduling theory to the model is. Using real-time scheduling theory, we will be able to:

- determine the execution characteristics of every node in the graph,
- determine whether or not a graph can be scheduled to meet its hard real-time processing requirements,
- bound the latency between receipt of sensor data and the presentation of processed data,
- use dynamic scheduling techniques to ensure latency bounds are met,
- determine the maximum amount of memory required by the application, and
- add or remove graphs to the system "on the fly" while still guaranteeing the hard real-time requirements of the executing applications.

2 The Dataflow Model

In our dataflow model, a system is expressed as a directed graph of large grain nodes (processing functions) and edges (logical communication channels). The topology of the graph defines the flow of data tokens from an input source to an output sink, defining a software architecture independent of the hardware hosting the application.

The edges of a graph are typed First-In-First-Out (FIFO) queues. The data type of the queue indicates the size of each token transported from a producer to a consumer. Tokens are appended to the tail of the queue (by the producer) and read from the head (by the consumer). The tail of a queue can be attached to at most one node at any time. Likewise, the head of a queue can be attached to at most one node at any time.

The nodes of the graph embody processing functions. Each node must have at least one input port and zero or more output ports. At least one of the input ports must be attached to a queue. The rest of the node ports may be attached to queues, graph variables, or constants. The port type and the type of the attached data object (queue, variable, or constant) must match.

There are three attributes associated with a queue: a produce, threshold, and consume amount. The produce amount specifies the number of tokens atomically appended to the queue when the producing node completes execution. The threshold amount represents the minimum number of tokens required before the node may process data from the input queue. The consume amount is the number of tokens dequeued (starting at the head) after the processing function finishes execution. A queue is *over threshold* if the number of enqueued tokens meets or exceeds the threshold amount. Unlike most dataflow paradigms, our model (as does *PGM*) allows non-unity produce, threshold, and consume amounts as well as a consume amount less than the threshold. All three attributes must be non-negative values, and the consume amount must be less than or equal to the threshold amount.

A node within a graph is eligible for execution whenever all of its input queues are over threshold. Although

PGM refers to these nodes as *pooled nodes* we use the more common term *And* nodes. An *And* node begins execution by reading the specified amount of data from all of its input ports. Next the processing function is executed. After the processing function completes, the output graph variables are written and the specified produce amount of tokens is appended to each output queue. Before the node terminates, but after data is produced, *consume* tokens are dequeued from each input queue. The production and consumption of data are both atomic operations. That is, all *produce* tokens are available to the consumer at the same time; all *consume* tokens are removed from an input queue in one indivisible operation. We assume that no data can be lost during graph execution and that no two executions of the same node overlap.

The graph of Figure 1, labeled $Chain_1$, contains two types of objects: nodes and queues, and has the topology of a chain. For simplicity, we employ chains to introduce our real-time dataflow model. A chain has exactly one queue connecting any two nodes. The queue connecting N_i to N_{i+1} is labeled Q_i. For example, the queue connecting the source node N_0 to its consumer N_1 is Q_0. The produce amount is the number located at the tail of the queue (connected to the producing node). The threshold and consume values are placed at the head of the queue and separated by a comma. For example, the produce amount for Q_1 is 3, the threshold is 3, and the consume value is 2.

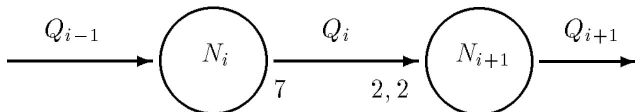
3 Execution Model

We have made substantial progress in defining the execution model for a single processor. We are currently working on completing the single processor execution model and extending it to a distributed paradigm. This section briefly covers some of our results. A much more thorough and rigorous treatise of our real-time dataflow execution model is provided in [3]. In addition to the two theorems presented without proof in this section, we have established (or a working to establish):

- schedulability tests,
- bounds on end-to-end latency, and
- bounds on the memory requirements of a graph.

Example 3.1. Node Execution:

In the two node chain below, N_i produces 7 tokens every time it executes. N_{i+1} has a threshold of 2 and consumes 2 after it executes.



Assume Q_i contains no data. The first execution of N_i enables 3 executions of N_{i+1}, which consumes 6 of the 7 tokens produced — leaving 1 token on Q_i. Hence, the second execution of N_i enables 4 executions of N_{i+1}. After 2 executions of N_i and the resulting 7 executions of N_{i+1}, Q_i is left in its original state: containing 0 tokens. □

We assume the synchrony hypothesis of [2] to introduce the concept of node execution rates. Under the synchrony hypothesis, we assume the graph executes on an infinitely fast machine. Hence, each node takes no time to execute and data passes from source to sink node instantaneously. The synchrony hypothesis lets us define execution rates in the absence of scheduling algorithms and deadlines.

Definition 3.1. An *execution interval* is a period of time during which a node is guaranteed to have a constant number of executions.

Definition 3.2. Let t be the beginning of the first execution interval for node N. The *execution rate* for N is specified as (x, y); (Assuming the strong synchrony hypothesis of [2]) N will execute exactly x times in all time intervals of $[t + y \cdot (j-1),\ t + y \cdot j),\ \forall j > 0$.

Throughout this section, we assume that each node produces a constant number of tokens and consumes a constant number of tokens with $consume_i \leq threshold_i$, where the threshold is also constant. If the produce and consume values are not constant, then maximum produce and minimum consume values can be used to determine the maximum node execution rates.

Definition 3.3. The rate specification for N_i is represented as (x_i, y_i).

Definition 3.4. The beginning of the first execution interval for N_i is represented as t_i.

Theorem 3.1. *Given the rate specification of* (x_i, y_i) *for* N_i*,* N_i *will execute exactly* x_i *times in every interval of* $[t_i + (k \cdot y_0),\ t_i + (k \cdot y_0) + y_i),\ \forall k \geq 0$.

Theorem 3.2. $\forall i \geq 0, j > 0$*: Let* t_{i+1} *represent the beginning of first execution interval of* N_{i+1}*. Given the attributes for* Q_i *in a Chain and the execution rate for* N_0 *to be* (x_0, y_0)*, the execution rate of* N_{i+1} *for all intervals defined by* $[t_{i+1} + (j-1) \cdot y_{i+1},\ t_{i+1} + j \cdot y_{i+1}]$ *is* (x_{i+1}, y_{i+1}) *where*

$$\begin{cases} x_{i+1} = \frac{produce_i \cdot x_i}{\gcd(produce_i \cdot x_i, consume_i)} \\ y_{i+1} = \frac{consume_i \cdot y_i}{\gcd(produce_i \cdot x_i, consume_i)} \end{cases} \qquad (3.1)$$

Example 3.2. Let's apply Theorem 3.2 to $Chain_1$ of Figure 1: Let N_0 be periodic such that it executes once

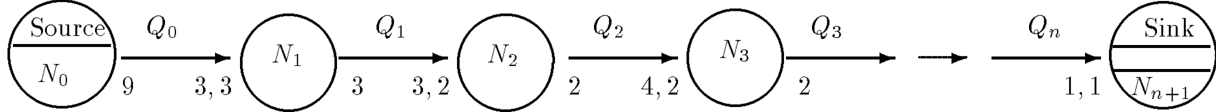


Figure 1: $Chain_1$

every y time units, which means it has an execution rate of $(1, y)$. We use Theorem 3.2 to derive the execution rate of the N_1 through N_3 as follows:

$$ExecuteRate_1 = (x_1, y_1)$$
$$= \left(\frac{9 \cdot 1}{\gcd(9 \cdot 1, 3)}, \frac{3 \cdot y}{\gcd(9 \cdot 1, 3)}\right) = \left(\frac{9}{3}, \frac{3y}{3}\right) = (3, y)$$
$$\Rightarrow \begin{cases} x_1 = 3 \\ y_1 = y \end{cases}$$

$$ExecuteRate_2 = (x_2, y_2)$$
$$= \left(\frac{3 \cdot 3}{\gcd(3 \cdot 3, 2)}, \frac{2 \cdot y}{\gcd(3 \cdot 3, 2)}\right) = \left(\frac{9}{1}, \frac{2y}{1}\right) = (9, 2y)$$
$$\Rightarrow \begin{cases} x_2 = 9 \\ y_2 = 2y \end{cases}$$

□

4 Conclusion and Future Work

We are in the process of completing the proofs for our latency and buffer bounds. We are also attempting to verify a feasibility test that can be used as schedulability test for preemptive EDF scheduling and variants of the canonical EDF algorithm. The single processor model will be useful for evaluating simple signal processing applications, but the real benefit of the real-time dataflow methodology is the ability to support a variable number of processors in a distributed system.

After completing the single processor model, which has already yielded a surprising number of new results, we will extend the model to distributed systems. The distributed model can then be integrated into existing programs such as the RASSP project, or the An/UYS-2A COTS Variant (ACV) being developed by Lucent Technologies for the Navy.

References

[1] 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.

[2] 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.

[3] Goddard, S., Jeffay, K., "A Temporal Semantics for Real-Time Dataflow" , University of North Carolina, Department of Computer Science, Technical Report, *in progress.*

[4] 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.

[5] *Processing Graph Method Specification: Version 1.0*, prepared by the NRL for use by the Navy Standard Signal Processing Program Office (PMS-428), December 1987.