A Framework for Real-Time High-Throughput Signal and Image Processing Systems on Workstations

Ph.D. Qualifying Proposal

Computer Engineering Area
Department of Electrical and Computer Engineering
The University of Texas at Austin

Submitted by Gregory E. Allen, B.S.E.E., M.S.E.
gallen@arlut.utexas.edu
http://www.ece.utexas.edu/~allen/

Committee Members:
Prof. James C. Browne (Dept. of CS)
Prof. Craig M. Chase (Dept. of ECE), chairman
Prof. Brian L. Evans (Dept. of ECE), advisor
Prof. Lizy K. John (Dept. of ECE)
Dr. Charles M. Loeffler (ARL:UT)

Abstract

Real-time data-intensive systems such as sonar beamformers and synthetic aperture radar processors have traditionally required implementation in expensive custom hardware. Current systems use off-the-shelf programmable processors in customized configurations to reduce development cost. To reduce development cost and time further, we consider the use of workstations as the target architecture and design environment. We present a general approach for realizing real-time data-intensive systems in software on a multiprocessor workstation.

First, we present several dataflow models which are commonly used to describe systems of this nature. Second, we present a framework for developing scalable software implementations of signal and image processing systems on workstations. The framework models the concurrency and parallelism in these systems using Process Networks. The Process Network model guarantees determinate execution of concurrent programs regardless of the scheduling algorithm used. We employ a scheduling algorithm that always finds a bounded execution if one exists. Third, we implement the framework in C++ using lightweight real-time POSIX threads.

We use two case studies to evaluate the performance of our framework: a high-resolution 3-D sonar beamformer, and a synthetic aperture radar processor. On a Sun Ultra Enterprise workstation, the 4-GFLOP beamformer exhibits near-linear speedup using 1 to 12 processors and executes in real-time with 12 336-MHz UltraSPARC-II processors.

Table of Contents

1. Introduction **3**
2. Models of Computation and Communication **6**
2.1. Synchronous Dataflow. 8
2.2. Computation Graphs 9
2.3. Boolean Dataflow and Dynamic Dataflow 10
2.4. The Process Network Model 10
3. Design Framework **13**
3.1. Implementation 14
3.1.1. Queues 14
3.1.2. Process Nodes 16
3.1.3. Node Granularity 16
3.1.4. Communication Channels 17
3.2. Examples 19
4. Case Study #1: Sonar Beamformer **20**
4.1. Vertical Beamformer 22
4.2. Horizontal Beamformer 23
4.3. Prototype of a 4-GFLOP 3-D Sonar Beamformer. 24
4.4. Performance Results 25
5. Case Study #2: SAR Processor **27**
6. Conclusion **30**
7. Future Work **30**
7.1. Framework Enhancements and Extensions 31
7.2. Formal Analysis. 31
7.3. Additional Case Studies. 32
7.4. Rapid Prototyping 32
7.5. Schedule. 32
References. 33
Graduate Coursework 35
Vita 35

1. Introduction

High-performance, high-throughput real-time systems, on the order of a billion (giga-) floating-point operations per second (GFLOPS) and 100 MB/s of data I/O, have traditionally required implementation in expensive custom parallel hardware. Examples of these systems that perform signal and image processing operations include high-resolution sonar beamformers, synthetic aperture radar (SAR) processors, and multichannel image restoration systems at video rates. These systems typically exhibit computational parallelism in both space and time. The non-recoverable engineering cost for custom hardware development may make this approach prohibitively expensive because these systems are not typically sold in high volumes. Because of technological advances following Moore's Law, problems requiring computation on the order of GFLOPS can now be solved using commercial programmable processors.

The more recent "second-generation" approach connects dozens of commercial programmable processors in customized configurations, e.g. using a VME backplane. Using this approach, one 4-GFLOP sonar beamformer requires about 100 80-MFLOPS Analog Devices SHARC digital signal processors. Although using commercial off-the-shelf (COTS) components reduces hardware development cost and time over custom parallel hardware, software development and system integration and testing are difficult and costly. Partitioning the algorithm among several dozen processors while guaranteeing synchronization is difficult, and debugging the software is difficult due to low observability of program state. Once complete, the software developed for a custom COTS configuration is closely tied to the hardware topology, thereby making it difficult to reuse source code or change configuration in the field.

For the next generation of these systems, we advocate the use of commodity multiprocessor workstations. Modern multiprocessor workstations are capable of native signal processing [1]

with GFLOPS performance. For a workstation implementation, development cost and time are significantly reduced compared to that for custom hardware or an embedded COTS system with the same level of performance. By implementing high-performance, high-throughput real-time systems in software on commodity high-performance workstations, we take advantage of advances in commercial workstations, thus sharing hardware development costs with the high-volume workstation server market. We also reduce software development efforts because mature high-volume operating systems and tools can be used. Workstations also offer better software portability, and better hardware upgradability and maintainability than embedded COTS solutions. Because the development environment and target architecture are the same, the design tools can be deployed with the design for in-the-field changes, thereby making the target system dynamically reconfigurable. Also, development can be performed on less powerful workstations, which makes multiple sets of target hardware available.

In an embedded COTS system, the system is often partitioned into a very large number of pieces, where each piece is statically scheduled. Dynamic scheduling has no real advantage in an embedded COTS system because tasks cannot migrate among processors for load balancing. Static scheduling eliminates run-time overhead, but the resulting program is dependent on the specific hardware topology for which the schedule was determined. Static scheduling may also be wasteful for algorithms which take a varying amount of processing time, because the scheduler must account for the worst case.

Dynamic scheduling requires an operating system and incurs a run-time overhead. Modern workstation operating systems dynamically schedule processes to balance the workload among available processors, thus giving efficient utilization. This load balancing is called *symmetric multiprocessing* (SMP) [2]. Unix extends this type of load balancing to *threads.* A thread is an inde-

pendent flow of control within a process. Each thread has its own registers and stack, but shares data and code space with the rest of the process. The Portable Operating System Interface (POSIX) [3] provides a standard thread library, called Pthreads. Pthreads have low overhead, and can be scheduled with a fixed real-time priority. By partitioning software systems using large granularity threads, we reduce the scheduling overhead by reducing context switches.

Two inherent problems with concurrent programming are *deadlock* and *non-determinate behavior.* Deadlock occurs when execution halts prematurely. Non-determinate behavior occurs when different executions of the same program yield different results, and is generally due to the use of a shared resource by multiple threads without proper synchronization. Proper synchronization between threads can guarantee determinate behavior, but can reduce parallelism and introduce deadlock. The Process Network model of computation [4,5] captures concurrency and parallelism, provides for correctness and liveness, and guarantees determinate execution regardless of the scheduling algorithm used. Dynamic scheduling based on the availability of data allows execution in bounded memory [6].

In this paper, we present a formal framework for developing scalable software implementations of high-performance high-throughput signal and image processing systems. The framework models the concurrency and parallelism in these systems using the bounded memory Process Network model. We implement the framework in C++ using POSIX threads to obtain low overhead, high performance, and scalability. We evaluate the framework with two case studies: a sonar beamformer and a SAR processor. Our 4-GFLOP beamformer case study on a Sun Ultra Enterprise workstation achieves near-linear speedup over 1 to 12 processors, and meets real-time constraints using 12 UltraSPARC-II processors running at 336-MHz. Table 1 compares the cost and volume for three generations of 4-GFLOPS sonar beamformers.

	Custom Hardware	Embedded COTS	Commodity Workstation
Development cost	$2M	$500K	$100K
Development time	2 years	1 year	6 months
Physical Size (m^3)	3.0	3.0	4.0
Reconfigurability	low	medium	high
Software portability	low	medium	high
Hardware upgradability	low	medium	high

Table 1: Three generations of low-volume 4-GFLOP sonar beamformers (under 50 units). Estimates are based on 1999 technology. The last column is the case study third-generation sonar beamformer developed in this paper.

2. Models of Computation and Communication

The computation and communication in signal and image processing systems are commonly modeled as *dataflow* systems. For dataflow models, a program is represented as a directed graph. The arcs of the graph represent one-way FIFO queues for communication of *tokens*. The nodes of the graph represent *actors*, which have *firing rules* that specify when the actor can fire. When an actor fires, it consumes input tokens and produces output tokens. Each actor may have any number of incoming or outgoing queues, and may communicate with other actors only through these queues. A *stream* is a sequence of tokens, and a process which is formed from repeated actor firings is called a *dataflow process* [6]. Fig. 1 shows an actor firing.

We call a program *determinate* if the results (the tokens produced on the communication channels) are the same regardless of which order the actors fire. Obviously this is a desirable property for a dataflow model; if this is not the case, then the results depend on the *execution order* of the

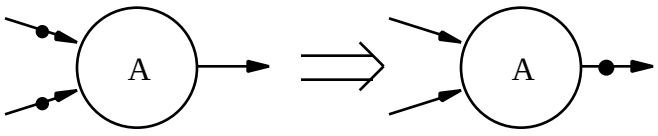


Fig. 1: Dataflow actor “A” firing by consuming two input tokens and producing one output token.

actors in the program. The *state* of the graph is the number of tokens on each arc.

Termination is an additional property of a dataflow program. The unique least fixed point of the state of the dataflow graph determines the total length and value of every stream in the program, but many possible execution orders that lead to this solution may exist. In a *terminating* program, all streams in the least fixed point solution are finite in length. A *non-terminating* program contains at least one stream in the least fixed point solution that is infinitely long. Just as it is impossible to decide (in finite time) whether or not a Turing machine will halt, it is impossible to determine if a dataflow graph that is Turing complete will terminate. Termination is a property of the program and does not depend on execution order. A terminating program has reached a *complete execution* if no actor is capable of firing.

Another important property of a dataflow program is *boundedness*. A program is *bounded* if token accumulation on any channel will not exceed some finite constant. In a *strictly bounded* program, token accumulation on any channel will not exceed some finite constant for *all* executions. A program is *unbounded* if at least one channel is not bounded for all complete executions. Although the total stream lengths are a property of the program, the number of unconsumed tokens that can accumulate on communication channels depends on the execution order [6].

Several determinate dataflow models have been developed, including (from more restrictive to more general) Synchronous Dataflow (SDF), Computation Graphs (CG), Boolean Dataflow (BDF), and Dynamic Dataflow (DDF). For SDF and CG, which are not Turing complete, program properties (such as termination and boundedness) are decidable, and scheduling can be performed statically. BDF and DDF are Turing complete, and in general, must rely on dynamic scheduling. Fig. 2 shows a Venn diagram for these dataflow models of computation. The Process Network model is a superset of dataflow models.

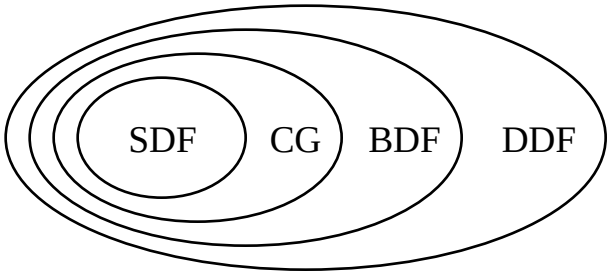


Fig. 2: Dataflow models of computation. The Process Network model is a superset of dataflow models.

2.1. Synchronous Dataflow

When an actor in a Synchronous Dataflow [7] program executes, it always produces and consumes the same fixed number of tokens. Additionally, the flow of data through the graph may not depend on values of the data. Because of these restrictive properties, the questions of termination and boundedness are decidable in finite time. Therefore a static, finite schedule can be constructed at compile time, and both the flow of control and token accumulation are known. This static schedule can be periodically repeated to implement a dataflow process that operates on infinite streams of data tokens. The consequence of this static scheduling is that an SDF graph may not contain data-dependent switch statements such as an *if-then-else* construct or data-dependent iteration such as a *for* loop.

A *complete cycle* is a sequence of actor firings that returns a graph to its original state. To determine a static schedule, we must first determine the number of firings of each actor to *balance* the graph, such that the total number of tokens produced on each arc is equal to the total number consumed. For the program in Fig. 3, node A must fire 3 times, node B must fire 4 times, and node

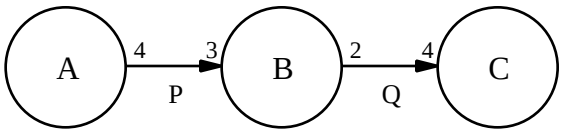


Fig. 3: A balanced SDF program.

C must fire 2 times to balance the number of tokens produced and consumed. The next step is to schedule the firings required by balancing so as to resolve the data dependencies. Multiple valid schedules may exist. Valid schedules for Fig. 3 include {AAABBBBCC} and {ABABCABBC}. In terms of memory usage, the latter schedule is optimal because it requires the least amount of data buffering (6 tokens on arc P and 4 tokens on arc Q).

This type of operation is well-suited to multirate digital signal processing and communications systems. Indeed, this model has met some commercial success, and is utilized in such design automation packages as HP EEsof's Advanced Design System [19] and Cadence Design's Cierto Signal Processing Work System [20].

2.2. Computation Graphs

Karp and Miller [8] Computation Graphs are a slightly more general model. Here, each arc has four non-negative integer constants associated with it:

- *A* - The number of tokens initially present.
- *U* - The number of tokens inserted each time the producer node fires.
- *W* - The number of tokens removed each time the consumer node fires.
- *T* - The number of tokens required to be in the arc before the consumer can fire.

Clearly $T \geq W$. The questions of termination and boundedness are decidable in finite time, and Computation Graphs can be statically scheduled. Karp and Miller provide iterative algorithms to determine these properties based on the above constants.

What separates Computation Graphs from SDF is that the threshold number of tokens required to fire a node (*T*) need not match the number of tokens consumed upon firing that node (*W*). SDF is the special case where $T=W$ for every edge. We will leverage this firing threshold in

our framework to allow efficient implementation of nodes which require more tokens than they consume (e.g. digital filters).

2.3. Boolean Dataflow and Dynamic Dataflow

Boolean Dataflow and Dynamic Dataflow add data-dependent control flow (*if-then-else* statements and *for* loops) and are Turing complete. Dynamic Dataflow additionally supports run-time recursion. With the addition of these constructs, termination and boundedness are no longer decidable in finite time. This means that static scheduling is no longer possible – the schedule must be determined dynamically, as the program executes.

2.4. The Process Network Model

Kahn Process Networks are a concurrent model of computation that is a superset of dataflow graphs. Like dataflow graphs, Process Network programs are described by a directed graph, where each arc represents a FIFO queue for communication. Here, each node of the graph represents an independent, concurrent process. A Process Network can be thought of as a set of Turing Machines connected by one-way tapes, where each machine has its own working tape [4]. Figure 4 shows a sample Process Network program.

For Process Networks, a node suspends execution when attempting to consume data from an empty queue, and cannot detect the presence or absence of data in that queue. Hence, reads are blocking. However, nodes are never suspended for producing data. Hence, writes are non-block-

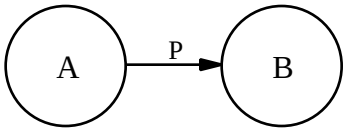


Fig. 4: A simple Process Network program. Processes A and B execute concurrently, and A sends data to B through a one-directional channel (FIFO queue) P.

ing. Non-blocking writes can cause unbounded accumulation of data on a given queue, so queues are of infinite length.

A Process Network program is determinate: the results are the same for every possible execution order. This model guarantees correctness regardless of the choice of scheduling algorithm (e.g. data-driven or demand-driven) or implementation style (e.g. sequential, distributed, or parallel). The problems of determining whether a Process Network program will terminate, and whether it can be scheduled in bounded memory are undecidable in finite time. In this context, the scheduler must work dynamically, as the program executes.

For Process Networks, termination and total stream length are properties of the program and do not depend on the execution order. However, the number of tokens that can accumulate on communication channels depends on the choice of execution order. Using Fig. 4 as an example, if the nodes are executed as {ABAB…}, then channel P must buffer only one data element. However, if process A executes an infinite number of times before B executes, then the queue P must buffer an infinite number of data elements.

Infinitely large queues cause obvious problems; execution in bounded memory is necessary for any practical implementation. Any arbitrary Process Network can be transformed into a strictly bounded one by adding a feedback channel for every data channel and modifying each process. Fig. 5 shows how the simple bounded network in Fig. 4 can be made strictly bounded. However, this method could introduce *artificial deadlock*, thus transforming a non-terminating program into a terminating one.

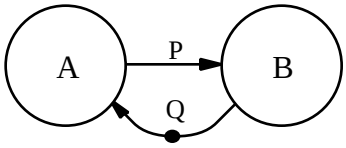


Fig. 5: A strictly bounded Process Network.

Rather than transform the Process Network, Parks developed rules for dynamic scheduling in bounded memory [6]. Parks lists two requirements for the scheduler:

1. *Complete Execution* – The scheduler should implement a complete execution of the Process Network program (or execute it forever if the program is non-terminating).
2. *Bounded Execution* – The scheduler should (if possible) execute so that only a bounded number of tokens ever accumulate on any of the communication channels.

When these two requirements conflict (such as for unbounded programs), requirement 1 takes precedence over requirement 2. That is, a complete, unbounded execution is preferable to a partial, bounded one.

Parks also shows that the following rules will yield a bounded schedule, if one exists:

1. Block when attempting to read from an empty queue.
2. Block when attempting to write to a full queue.
3. If we reach *artificial deadlock*, which occurs when execution has stopped because processes are blocked writing to full channels, then increase the capacity of the smallest full queue until the producer associated with it can fire.

This bounded scheduling policy has the desired behavior for all types of programs – terminating or non-terminating, strictly bounded, bounded, or unbounded. Now any scheduler will work, because any execution leads to bounded buffering on the queues. Bounded scheduling of Process

Networks is well-suited for implementation using the threaded model of concurrent programming.

3. Design Framework

This section presents a framework for modeling the concurrency and parallelism in scalable software programs using Process Networks [4,5]. The Process Network model guarantees determinate execution and correctness regardless of the scheduling algorithm being used. The bounded scheduling algorithm [6] permits a workstation implementation that efficiently utilizes multiple processors.

Our framework extends Process Networks by borrowing the concept of a firing threshold (*T*) from Computation Graphs [8]. Having a consumer (input) threshold is useful for "overlapping" algorithms which require access to more tokens than they consume, such as digital filters. Having a producer (output) threshold is useful for algorithms that have an unknown number of outputs, such as a prime sieve or highlight detection processing. These thresholds also aid efficient implementation, because node computations can operate directly in queue memory, which avoids the copying of data into the local memory of a node.

An additional goal of this framework is to decouple the computation (nodes) from communication (queues). This allows compositional programming of correct, scalable parallel programs from a library of computational nodes, while also allowing node clustering for reduction of scheduling overhead. Although the case studies presented in this paper can be modeled as SDF, our framework dynamically schedules the programs so that execution on a symmetric multiprocessing system can effectively utilize parallel hardware. Static scheduling of algorithms across multiple processors is beyond the scope of this implementation.

3.1. Implementation

Our implementation of the Process Network framework is intended for high-throughput computationally intensive algorithms on server-class symmetric multiprocessing workstations. Our implementation uses C++ template data types, and uses a layered approach based on the C++ inheritance mechanism to build interfaces and functionality. This scalable software framework is built upon a highly portable POSIX Pthread class library which can execute on many different Unix operating systems.

3.1.1. Queues

The ThresholdQueue C++ class is near the base of the inheritance hierarchy, and is optimized for data-intensive applications. The ThresholdQueue works much like a typical queue, but is intended to make up for the lack of circular address buffers in general purpose processors. Because ThresholdQueue is a C++ template class, it can queue any type of data. In order to prevent unnecessary copying of data, pointers are used to read and write data directly from queue memory, and data is guaranteed to be contiguous in memory. This reduces overhead, and simplifies the implementation of algorithms that interface to these queues.

The Karp and Miller concept of separating the firing threshold (T) from the dequeue count (W) is fundamental to the ThresholdQueue. In addition to the type of data being queued, instantiation requires the total queue length and a maximum threshold of the number of elements that will ever be requested (input or output). This threshold is the maximum of the Computation Graph parameters U, W, and T for a queue as described in Section 2.2.

The basic interface to the ThresholdQueue is reused throughout the Process Networks implementation, and uses pointers to avoid data copying by the user. A transaction with a ThresholdQueue is a three-step process:

1. Get a pointer to some number of contiguous data elements with the GetEnqueuePtr or GetDequeuePtr method, each of which takes a threshold and returns the pointer.
2. Operate on the data by de-referencing the pointer, up to the threshold length.
3. Actually insert or remove the data by calling the Enqueue or Dequeue method, each of which takes a data count.

The threshold and count values are not static, and may change on each queue transaction. The count must always be equal to or less than the threshold, although this is not strictly enforced. For enqueueing, Computation Graphs require that the count and the threshold are equal. However, that is not the case in this framework.

The ThresholdQueue implements its apparent circular addressing by mirroring the beginning of the queue's data region (up to the maximum threshold) just past the end of the queue's data region. Using this methodology, the queue can provide a pointer to a contiguous block of data elements even when operating near the end of the data region. The queue manages this mirroring, and guarantees that the same data resides in both locations. Fig. 6 illustrates the ThresholdQueue implementation.

The ThresholdQueue has a trade-off between memory usage and overhead. When the data region is much larger than the mirror region, the queue rarely needs to copy data. When the mirror region is as large as the data region, copying must occur frequently, thereby increasing overhead and sacrificing performance. Fortunately, memory is usually abundant on a workstation.

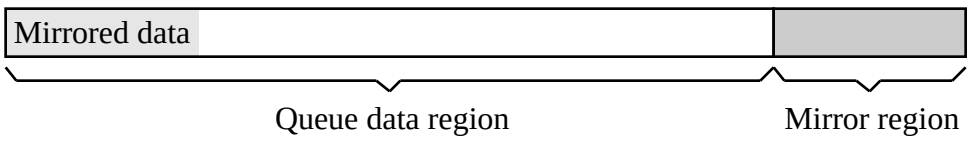


Fig. 6: Mirroring for circular addressing in the ThresholdQueue implementation.

On some Unix operating systems, such as Sun Solaris, the virtual memory manager can be used to prevent the ThresholdQueue from ever having to copy data when managing the mirror region. The system call `mmap` is used to map virtual memory objects into the address space of a process. By mapping a shared memory object to multiple virtual addresses, the same physical memory pages appear at multiple locations, and apparent circular addressing is achieved. As a side effect, the queue data and mirror regions must both be multiples of the system memory page size, which is 8 kb in Solaris. In this case, the queue size and threshold size are rounded up to the next multiple of the page size. Using `mmap` allows the queue circularity to be maintained by the hardware of the virtual memory manager, and can result in a significant performance gain. For the beamformer case study in Section 4.4, the virtual memory manager prevents a slowdown of 7.5% due to data copying.

3.1.2. Process Nodes

Each node in the Process Network program corresponds to a thread. These multiple threads can run concurrently when the program has parallelism, and thus can take advantage of multiple processors. Thread implementations are generally intended to provide high performance with low overhead. POSIX provides a standard thread interface, called Pthreads. Pthreads can be given real-time scheduling priority. By realizing Process Networks with POSIX Pthreads, our implementation can be run on many different Unix platforms. Since all nodes are derived from a PNNode base class, porting this system to a different thread implementation is relatively simple, requiring only a change to the PNNode class.

3.1.3. Node Granularity

Many signal processing algorithms are modeled using directed graphs, in which each node represents fine-grain computations such as addition and multiplication [11]. For example, the fast Fou-

rier transform (FFT) butterfly may be modeled in this manner. However, a fine level of granularity is inappropriate for this implementation, because the overhead of dynamic scheduling will dominate the overall execution time.

We use nodes of larger granularity – such as an FFT node, a filter node, or a beamformer node. The graphs drawn using this methodology are essentially block diagrams. The cost of firing a node should be much larger than the cost of a lightweight thread context switch (about 10 μs for the Sun Solaris Operating System). However, if the computation of a node is too costly, then the node may need to be divided into smaller pieces in order to run in real time. Generally, a trade-off exists between overhead, latency, and parallelism.

3.1.4. Communication Channels

Since the only way that nodes can communicate with each other is via communication channels, an abstracted interface to these channels is provided by the classes PNNodeInput and PNNodeOutput. These are virtual base classes and therefore cannot be instantiated – they only provide an interface for writing process nodes. Now each node is not concerned with the implementation of the communication channels, but instead only the interface.

These classes use the same transaction mechanism as the ThresholdQueue. Here, the methods `GetDequeuePtr` and `GetEnqueuePtr` are intended to be blocking. That is, they will not return until the threshold amount of data (or free space) is available. This interface obeys Parks' rules for bounded scheduling of Process Networks, as described in Section 2.4. Fig. 7 shows a partial declaration of PNNodeInput and PNNodeOutput.

Because of the abstracted interface to the communication channels, many different implementations could exist. A channel could send data to another process via shared memory, or to another computer through a network. A channel could also save its entire history to disk for program veri-

```
template<class T> class PNNodeInput {
    virtual const T* GetDequeuePtr(ulong threshold) = 0;
    virtual void     Dequeue(ulong count) = 0;
    // others omitted for brevity
};

template<class T> class PNNodeOutput {
    virtual T*        GetEnqueuePtr(ulong threshold) = 0;
    virtual void     Enqueue(ulong count) = 0;
    // others omitted for brevity
};
```

Fig. 7: A partial declaration of PNNodeInput and PNNodeOutput.

fication or debugging. However, most of the time data will simply be sent from one node (thread) to another within the same process. The primary mechanism for this is the C++ template class PNThresholdQueue, derived from the ThresholdQueue class described in Section 3.1.1.

This class is multiply inherited from the classes PNNodeInput, PNNodeOutput, and ThresholdQueue. Since it is derived from PNNodeInput and PNNodeOutput, it can be used as the input or output of any process node. It is responsible for blocking any nodes according to bounded scheduling rules. To minimize latency, the POSIX Pthread condition variable mechanism is used to awaken a blocked node as soon as it can continue executing.

At this time, deadlock detection as described in Section 2.4 is not implemented. For the class of real-time problems that this implementation is intended to solve, deadlock detection is not necessary – it is required for avoiding artificial deadlock, and for execution of unbounded programs. In this implementation, queues are generally allocated to be much larger than their minimum possible (deadlock avoiding) size, because of the performance reasons described in Section 3.1.1. Unbounded programs have no place in real-time implementations. However, the addition of deadlock detection would make this a more complete implementation of the Process Network model, and may be implemented in the future.

```
typedef float T;
while (true) {
    // blocking calls to get in/out data pointers
    const T* inPtr  = inputQ.GetDequeuePtr(inThreshold);
    T*       outPtr = outputQ.GetEnqueuePtr(outThreshold);

    DoComputation( inPtr, inThreshold, outPtr, outSize );

    // complete node transactions
    inputQ.Dequeue(inSize);
    outputQ.Enqueue(outSize);
}
```

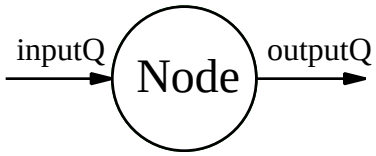


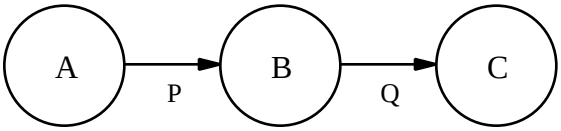
Fig. 8: A sample Process Network node.

3.2. Examples

To illustrate the use of our framework, we show the code for a sample node, and then we construct a sample Process Network program. Fig. 8 shows the code to implement a sample node. The thread implementing this node (and running this code) will block on GetDequeuePtr until inputQ has inThreshold data tokens available. It will then block on GetEnqueuePtr until outputQ has free space for outThreshold data tokens. The thread is now free to perform the computation with the two arrays, inPtr and outPtr. After completion, the node notifies the input queue to release inSize data tokens, and then notifies the output queue to insert outSize data tokens.

This interface obeys the rules for bounded scheduling of Process Networks, and the thresholds and sizes can be different on every firing. Also, the node computation is isolated from the communication. This node could be operating on a continuous overlapping stream of data, but the implementation is not concerned with that detail.

A sample Process Network graph and the code to implement it are shown in Fig. 9. The arcs *P* and *Q* are used to connect nodes *A*, *B*, and *C*. Recall that the parameters to the queue constructor are the queue length and the maximum threshold. This program spawns three threads and could utilize three processors.




```
int main() {
    PNThresholdQueue<T>  P (queueLen,maxThreshold);
    PNThresholdQueue<T>  Q (queueLen,maxThreshold);
    MyProducerNode       A (P);
    MyTransmuterNode     B (P, Q);
    MyConsumerNode       C (Q);
}
```

Fig. 9: A sample Process Network program and the C++ code to implement it.

Currently, we only provide support for building a Process Network by programming in the C++ language. A future goal is to support building parallel programs from a text file or a graphical user interface.

4. Case Study #1: Sonar Beamformer

Sonar is a method for detecting and locating objects using acoustic waves. Sonar can be used for navigation, to identify hazardous or hostile objects, and to map terrain features in the surrounding environment. Fig. 10 shows a sample underwater environment with a navigational hazard.

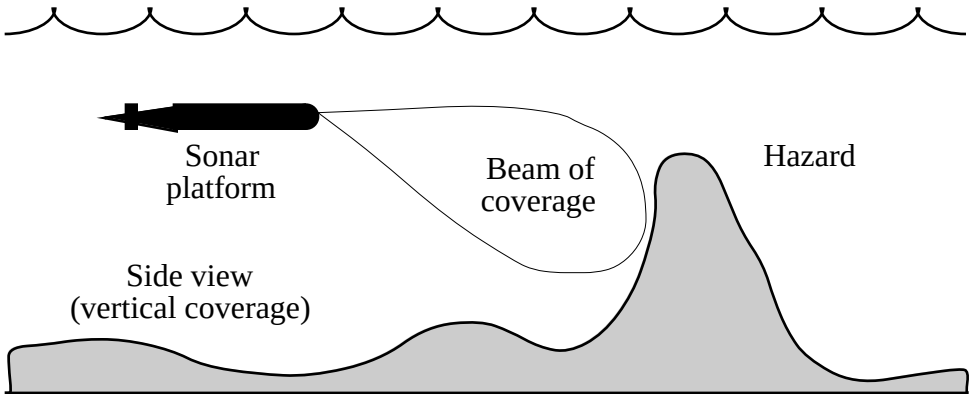


Fig. 10: A sample underwater environment.

High-resolution sonars generally consist of an array of underwater sensors along with a beamformer [12] to determine from which direction a sound is coming. The higher the resolution of a sonar system, the more accurately the location of an object can be determined. To achieve high resolution over a wide coverage area, a large number of beams may be formed. Fig. 11 illustrates a high resolution, multi-beam sonar with several narrow horizontal beams covering a wide horizontal sector.

The sensor element outputs must be combined to form these multiple narrow beams, each of which "looks" in a single direction and is insensitive to sound in neighboring directions. This combination must be performed with precise time delays and amplitude weighting applied to the sensor outputs. As a result, beamforming for high-resolution sonar systems is extremely computationally intensive.

We prototype a 3-D beamformer which utilizes both horizontal and vertical elements to image an underwater environment. Fig. 12 shows a block diagram of the beamformer, where each stage corresponds to a node in the Process Network implementation. The sensor array consists of 80 elements horizontally by 10 elements vertically, for a total of 800 elements. Digital data comes

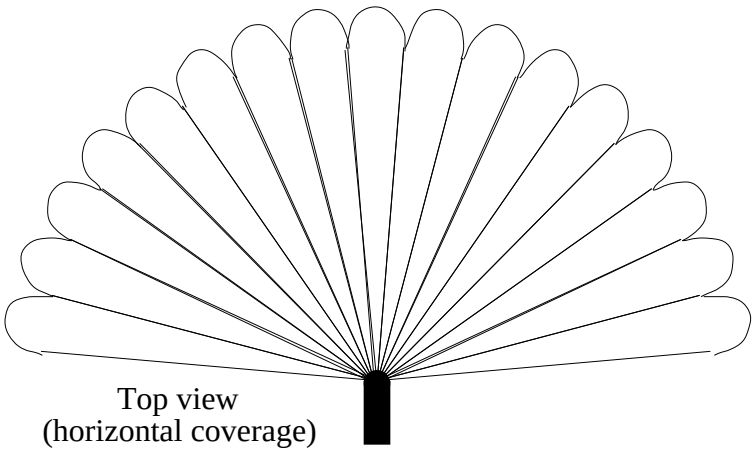


Fig. 11: Ahead-looking horizontal beam coverage for a sonar system. Multiple sensor outputs are combined to form each beam.

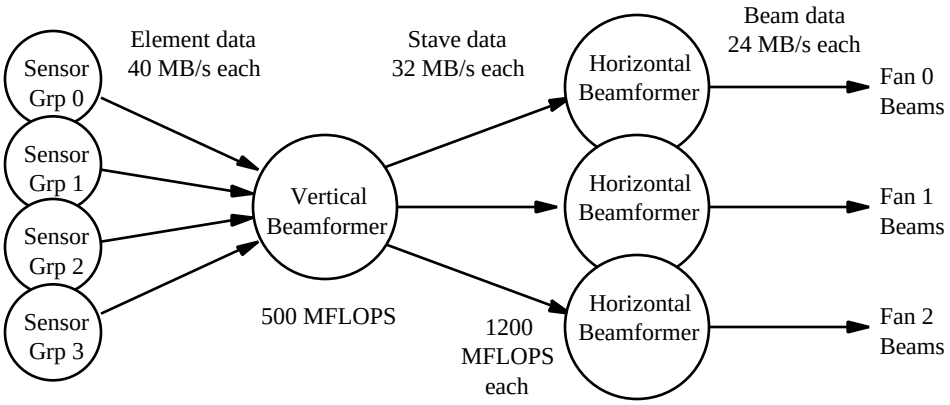


Fig. 12: A block diagram of the beamformer.

from the sensor array via four telemetry links, each at 40 Mb/s. First, the vertical beamformer weights and sums the vertical elements, thereby calculating the vertical response of 80 logical horizontal elements (staves). Three different sets of vertical weights are used to calculate 3 sets of stave outputs. Second, three horizontal digital interpolation beamformers compute 61 beams each, consuming the bulk of the computational requirement in the system.

4.1. Vertical Beamformer

For every horizontal position in the array, there are multiple vertical transducers. Each vertical sensor column is combined into a *stave*. For each sample of the array, one dot product per stave per vertical shading set (or fan) must be calculated. Although this has been described as a 500 MFLOPS algorithm, note that it is actually performed with integer arithmetic, because the sampled sensor element data is in an integer format. After calculation, this stage must perform integer-to-float conversion, and interleave the telemetry links for the following horizontal beamformers.

In implementing the integer arithmetic, the best performance on an UltraSPARC-II processor was obtained by using the Visual Instruction Set (VIS) [13]. The highest precision (and slowest) mode of VIS was required – 16-bit by 16-bit multiplication and 32-bit accumulation. In this mode, the peak multiply-accumulate (MAC) performance is only one operation per cycle (on average),

compared to two operations per cycle for floating-point. Despite the peak performance numbers, the VIS version is more than twice as fast as the floating-point version, because the overhead of the integer-to-floating point conversion is effectively eliminated [14].

Because of the high data rates at the vertical beamformer, memory latency hiding techniques are utilized to increase performance. Hand-coded software data prefetching [23] increases kernel performance by 34% by reducing load and store stalls. Although the vertical beamformer is performed with integer math using VIS, we measure the performance in MFLOPS for the sake of comparison. For the vertical beamformer kernel, we achieve 313.3 MFLOPS on a 336 MHz processor, which is 93% of peak [14].

4.2. Horizontal Beamformer

The horizontal beamformer utilizes time-domain beamforming, which is realized by weighting, delaying, and summing the outputs of the sensor array. In a digital system, the sensors are sampled at just above the Nyquist rate, and digital interpolation is used to achieve the desired time-delay quantization. This technique is called interpolation beamforming [9,10]. Fig. 13 shows a digital system with an interpolation beamformer. Multiple different beams are formed in parallel from the sensor data. For this system, each of the three horizontal beamformers compute 61 beams from the 80 staves. Floating-point numbers are used to preserve the desired dynamic range.

The horizontal beamformer kernel implementation for the UltraSPARC-II processor utilizes highly optimized C++ with loop unrolling, using the SPARCompiler5.0. The compiler did not generate software prefetching instructions, and attempts to improve performance with hand-insertion were unsuccessful. The horizontal beamformer kernel operates at 440 MFLOPS on a 336 MHz processor, which is 60% of peak [14].

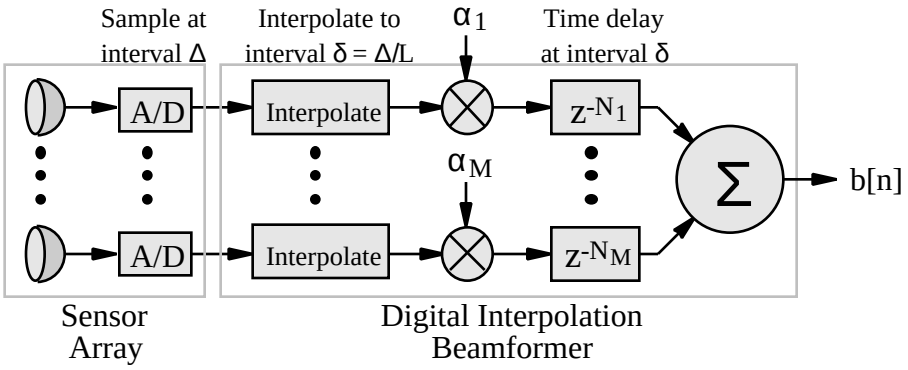


Fig. 13: Digital interpolation beamformer with a digitizing sensor array.

4.3. Prototype of a 4-GFLOP 3-D Sonar Beamformer

We construct a prototype beamformer by connecting the beamformer kernels in the Process Network framework according to Fig. 12. Implementation of a node that simply calls a beamformer kernel routine is straightforward, but cannot achieve real-time performance. A method for dividing the beamforming task in time is needed for a real-time workstation implementation. In order to divide this calculation without copying data, a beamformer node manages a *thread pool*, which is a common workstation multiprocessor programming strategy [15].

In the thread pool model, a manager task creates a fixed number of worker threads at initialization time which survive for the duration of the program. When the manager has work to do, it places a request on a queue. Workers remove requests from the queue and process them. When a beamformer node fires, it queues a request to each of several worker threads, and then blocks on each worker thread, to wait for completion. The number of worker threads can easily be increased or decreased as the processing performance requires. Four threads are allocated to each Process Network beamformer node, which is more than enough to keep up with the real-time requirement. This thread pool can also be modeled as a Process Network [16].

4.4. Performance Results

We benchmark the Process Network beamformer, comparing its performance to a thread-pool beamformer and to the sequential kernel performance. These benchmarks were performed on a Sun Ultra Enterprise 4000 with twelve 336-MHz UltraSPARC-II processors and 3 GB of RAM. We used the Sun Solaris 2.6 operating system, with threads executing in the "real-time" class. All results are determined as the average time over 100 trials to calculate 2.6 s of data. This benchmark uses over 400 Mb of element data and requires over 10 billion floating-point operations. Performance is measured as execution time, and MFLOPS are computed from execution time. We ensure that incoming data was not cached before the benchmarks were performed.

Fig. 14 shows performance results. The Process Network framework has a low overhead; on a single processor, the framework causes a slowdown of less that 0.5%, compared to sequential kernel performance. The Process Network beamformer program scales almost linearly from one to twelve processors; on twelve processors, the speedup is 11.28, which is an efficiency of 94%. Our real-time goal of about 4.1 GFLOPS is met with 10.5 336-MHz UltraSPARC-II processors. On twelve processors, it operates with 14% to spare.

The thread pool implementation first calculates three sets of stave data from the element data, using a vertical beamformer with eight threads. Then, three eight-thread horizontal beamformers sequentially execute, calculating beam results from the stave data. On eight processors, the Process Network beamformer performs about 7% faster than the thread pool beamformer. It also has lower latency and uses less memory – the communication channels in the network need only be large enough to prevent artificial deadlock. In this comparison, the resulting memory size is reduced by over 20 percent. Because it is more "stream" oriented, it is better suited for real-time execution. All nodes are independently operating all of the time, as the flow of data permits.

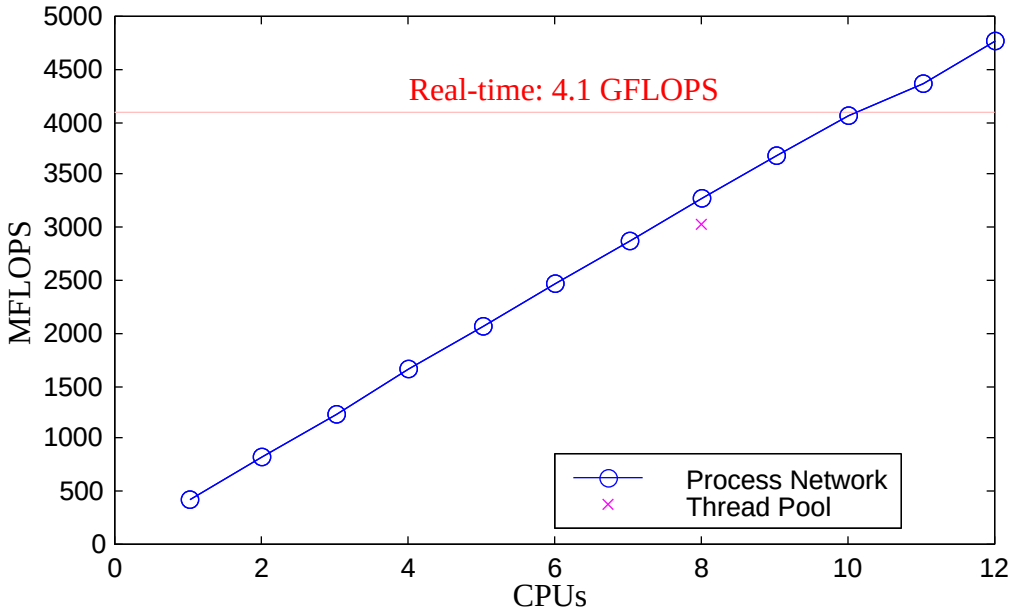


Fig. 14: Beamformer benchmark results using 336-MHz UltraSPARC-II processors. The results show near-linear speedup.

Although the memory, latency, and scheduling issues could be better addressed in the thread pool implementation, these advantages come automatically when using the bounded Process Network framework. An additional advantage of the Process Network implementation is that the program is scaled by the operating system according to the number of available processors. The thread pool beamformer was written specifically for eight processors. Utilizing more processors requires more worker threads, and arbitrarily creating large numbers of worker threads causes unnecessary overhead (and reduced performance). The Process Network beamformer scales without any change to the executable. If there were more processors than the sum of all worker threads, those processors would not be utilized. However, those extra processors are not needed for the system to meet its real-time goal.

5. Case Study #2: SAR Processor

Synthetic aperture radar (SAR) is used to collect high-resolution image data, typically of the Earth's surface. SAR systems can be deployed from aircraft or spacecraft, and are used in a wide variety of applications, including estimating terrain topography, classifying land surfaces, and cartography. SAR is also used to identify man-made objects. Such object identification typically requires SAR processing to be performed in real-time, and is usually implemented by means of a COTS embedded signal processor. Fig. 15 shows a sample airborne SAR system.

The length of a radar antenna determines the resolution in the azimuth (along-track) direction of the image: the longer the antenna, the finer the resolution in this dimension. SAR is a technique used to make an antenna longer synthetically by combining signals received by the radar as it moves along its ground track. As the radar moves, a *pulse* is transmitted at each position and the return echoes pass through the receiver and are recorded in an *echo store*. These multiple echoes can be combined and *focused* on a single point, effectively increasing the length of the imaging

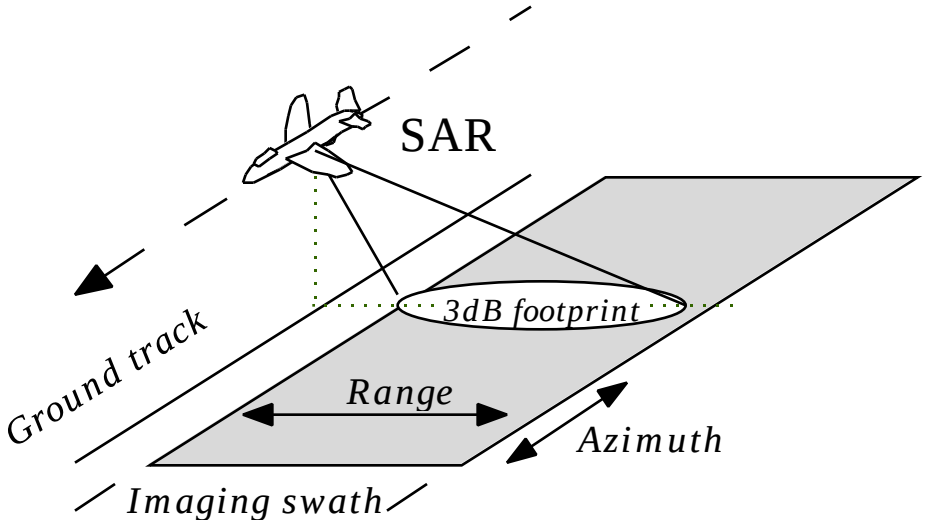


Fig. 15: An aircraft-based SAR system.

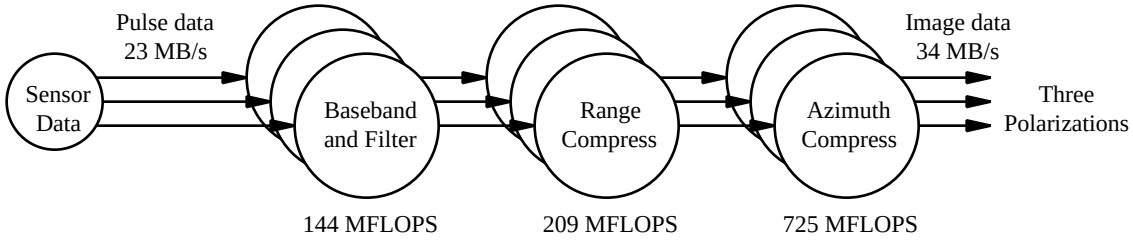


Fig. 16: A block diagram of the SAR processor. The SAR processor requires about 1 GB of memory and 1.1 GFLOPS of computation for a real-time implementation.

antenna. This focusing operation is commonly known as SAR processing. This requires precise knowledge of the relative motion between the platform and the imaged objects, and is computationally intensive.

We prototype a SAR processor which utilizes data collected from the M.I.T. Lincoln Laboratory Advanced Detection Technology Sensor, an aircraft-based 35 GHz Ka-band SAR sensor with an on-board data recording system [17]. The SAR processor we implement was used as the first benchmark case for the Rapid Prototyping of Application Specific Signal Processors (RASSP) project at Lincoln Laboratory [18], sponsored by the Advanced Research Projects Agency. Fig. 16 shows a block diagram of the SAR processor. For three different polarizations, complex baseband demodulation and filtering, range compression, and azimuth compression are performed. This system requires about 1.1 GFLOPS of processing power.

The complex baseband demodulation and filtering node will most likely be implemented with VIS. This node converts the real sampled data to in-phase and quadrature complex data, and then performs filtering with an eight-tap finite impulse response filter, which amounts to a dot-product of two eight-element vectors per input sample. This node also converts the incoming integer-format data to single-precision floating-point data for the subsequent stages.

In range compression, the 2024 uncompressed samples of each pulse are transformed into a compressed range pulse with 2048 samples. First, amplitude weighting is performed to reduce

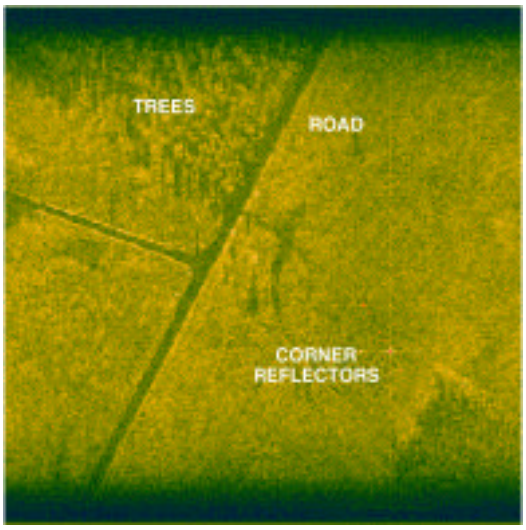


Fig. 17: A typical SAR image.

sidelobes. Next, zero padding expands the pulse to 2048 samples. Finally, a 2048-point fast Fourier transform (FFT) is executed. Each of the 2048 resulting samples can be thought of as constituting a range gate [18].

Azimuth compression is performed using cross-range convolution filtering. Each processing frame from the echo store contains 512 pulses, which is an array of 2048 x 512 complex samples. Convolution is performed in the frequency domain using FFTs with the overlap-and-save method [11]. For each of the 2048 range gates, a 1024-point FFT, 1024 complex multiplications, and a 1024-point inverse FFT are performed. This yields 2048 x 512 samples of image data. A typical SAR image output is shown in Fig. 17.

Sample source code and data sets for this SAR processor are available for download on the Web [21]. We propose to implement this processing within the bounded Process Network framework, which would give a high-performance, scalable implementation. This prototype SAR processor is being developed in collaboration with the UT Center for Space Research.

6. Conclusion

Traditionally, expensive custom hardware has been required to implement real-time signal and image processing systems. While the current generation of embedded COTS technology reduces development costs over custom hardware, software development and system integration are still costly and time consuming. To reduce development and integration costs, we consider the use of commodity multiprocessor workstations as target architectures. A workstation solution offers better software portability, reconfigurability, maintainability, and upgradability than COTS solutions.

We describe a formal framework for developing scalable software implementations of high-performance high-throughput signal processing systems. The framework is based on the Process Network model of computation which captures concurrency and parallelism in data-intensive systems. The Process Network model provides for correctness and determinacy, and can guarantee execution in bounded memory.

We implement the framework in C++ using lightweight real-time POSIX threads. This low-overhead, high-performance framework removes the dependency of the software on the hardware topology. Systems using this framework scale automatically. That is, the operating system dynamically schedules the threads to balance the workload among the available processors. When the workstation is both the development platform and the target architecture, development can be performed on less powerful workstations, multiple sets of development hardware are available, and the electronic design automation tools can be deployed with the design.

7. Future Work

The sonar beamformer case study has already proven to be a valuable data analysis tool at ARL, and the possibility exists that it will become a deployable real-time system. For my dissertation, I propose to analyze the formal properties of this framework, while adding enhancements and

extensions. I also propose to include additional case studies, and investigate the integration of this framework with rapid prototyping tools.

7.1. Framework Enhancements and Extensions

A planned enhancement is the ability to support busses of streams on an arc (e.g. multiple beams in a sonar system). Each arc would contain multiple identical channels, each with the same number of tokens, but different data values. I do not believe this affects the formal properties of the framework, but may serve to ease implementation and increase performance by modeling data parallelism and reducing overhead.

A desirable feature of this framework would be the ability to support multiple executions of the same program graph (e.g. multiple pings in a sonar system) through flushing and re-arming. Insertion and initialization of data tokens on flushing and re-arming would be useful for many applications (e.g. filters). The ability to pipeline this flushing would increase parallelism, but may not be possible with cycles in the graph.

I also propose to lay the groundwork for the development of an extensible library of signal processing nodes by implementing several common tools (e.g. fork, join, and filters). This will make compositional programming and rapid prototyping possible.

7.2. Formal Analysis

I propose to verify that the use of thresholds within bounded Process Networks preserves the formal properties of correctness and determinate execution, while still guaranteeing execution in bounded memory. I also propose to analyze the formal consequences of support for multiple executions.

7.3. Additional Case Studies

After completing the SAR processor case study, I propose to develop further high-performance high-throughput case studies for this framework. Possibilities include multimedia applications [24], multichannel image restoration [25], sonar post-beamformer processing, and sonar bottom profiling and mapping [26].

7.4. Rapid Prototyping

A key feature of this methodology is the ability to rapidly develop high-performance, correct, scalable software systems via the composition of individual nodes. Integration with design automation and rapid prototyping tools would allow non-experts to develop these systems from a graphical user interface. Ptolemy II [22] from the University of California at Berkeley is the likely choice for such a tool.

7.5. Schedule

- Summer 1999: Complete SAR processor case study.
- Fall 1999: Extend framework with busses of streams, investigate multiple executions. Implement a multimedia case study.
- Spring 2000: Analyze the formal consequences of the use of thresholds, and of multiple executions.
- Summer, Fall 2000: Investigate design automation tools, and begin integration.
- Spring 2001: Case study which utilizes design automation tools.
- Fall 2001: Write and defend dissertation.

References

[1] P. Lapsley, "NSP Shows Promise on Pentium, PowerPC," *MicroProcessor Report*, 1995.

[2] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah, D. Williams, M. Smith, S. Barton, G. Skinner, "Symmetric Multiprocessing in Solaris 2.0", *Proc. 37th IEEE International Computer Conf.*, San Francisco, CA, Feb 24-28, 1992, pp. 181-186.

[3] The IEEE Portable Applications Standards Committee Web Page: http://www.pasc.org/

[4] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Information Processing*, pp. 471-475, Stockholm, Aug. 1974.

[5] G. Kahn and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," *Information Processing*, pp. 993-998, Toronto, Aug. 1977.

[6] T. M. Parks, "Bounded Scheduling of Process Networks," *Technical Report UCB/ERL-95-105*, Ph.D. Dissertation, EECS Department, University of California, Berkeley, CA 94720-1770, Dec. 1995.

[7] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, vol. 36, no. 1, pp. 24-35, Jan. 1987.

[8] R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal*, vol. 14, pp. 1390-1411, Nov. 1966.

[9] R. G. Pridham and R. A. Mucci, "A Novel Approach to Digital Beamforming," *Journal Acoustical Society of America*, vol. 63, no. 2, pp. 425-434, Feb. 1978.

[10] R. G. Pridham and R. A. Mucci, "Digital Interpolation Beamforming for Low-Pass and Bandpass Signals," *Proc. of the IEEE*, vol. 67, no. 6, pp. 904-919, June 1979.

[11] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1989.

[12] R. J. Urick, *Principles of Underwater Sound*, McGraw-Hill Book Company, New York, NY, 1975.

[13] *Visual Instruction Set User's Guide*, Sun Microsystems, 1995.

[14] G. Allen, "An Evaluation of Native Signal Processing on the UltraSPARC-II with Sonar Beamforming Algorithms," Final Report for EE382M Computer Performance Evaluation and Benchmarking, Dept. of Electrical and Computer Engineering, The

University of Texas, Austin, TX 78712-1084, Dec. 1998. http://www.ece.utexas.edu/~allen/EE382M-F98/

[15] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*, O'Reilly and Associates, Sebastopol, CA, 1996.

[16] G. Allen, *Real-Time Sonar Beamforming on a Symmetric Multiprocessing UNIX Workstation Using Process Networks and POSIX Pthreads*, Master's Report, Dept. of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712-1084, Aug. 1998. http://www.ece.utexas.edu/~allen/MSReport/

[17] J.C. Henry, "The Lincoln Laboratory 35 GHz Airborne Polarimetric SAR Imaging System," *IEEE National Telesystems Conf.*, Atlanta, GA, 26-27 March, 1991, p.353.

[18] B. Zuerndorfer and G.A. Shaw, "SAR Processing for RASSP Application," *Proc. 1st Annual RASSP Conf.*, Arlington, VA, Aug. 15-18, 1994, pp. 253-268.

[19] J. L. Pino and K. Kalbasi, "Cosimulating Dataflow with Analog RF Circuits," *Proc. IEEE Asilomar Conf. on Signals, Systems, and Computers*, Pacific Grove, CA, Nov. 1-4, 1998, pp. 1710-1714. http://www.ece.utexas.edu/~bevans/professional/asilomar98/pino.pdf

[20] Cadence Design Cierto Signal Processing Work System Web Page: http://www.cadence.com/technology/hwsw/products/SPW.html

[21] The RASSP Web Page: http://www.ll.mit.edu/llr assp/

[22] The Ptolemy II Web Page: http://ptolemy.eecs.berkeley.edu/

[23] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching" *Proc. ACM International Conf. on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991, pp. 40-52.

[24] W. Chen, H. Reekie, S. Bhave, and E. Lee, "Native Signal Processing on the UltraSPARC in the Ptolemy Environment," *Proc. IEEE Asilomar Conf. on Signals, Systems and Computers*, Pacific Grove, CA, Nov. 1996, pp. 1368-1372.

[25] H.-T. Pai, *Multichannel Blind Image Restoration*, Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712-1084, May 1999.

[26] T. L. Henderson, "Wide-Band Monopulse Sonar: Processor Performance in the Remote Profiling Application," *IEEE Journal of Oceanic Engineering*, vol. OE-12, no. 1, Jan 1987, pp.182-197.

Graduate Coursework

Semester	Course	Title	Instructor	Unique	Grade	Area
Fall 1993	EE 380L.6	Interfacing to Operating Systems	Lipovski	14605	A	1
Spring 1994	EE 382N.10	Parallel Computer Architecture	Chase	14540	A	2
Fall 1994	EE 381K.3	Digital Filtering and Data Smoothing	Smith	14170	A	(TISE)
Spring 1995	EE 381K.8	Digital Signal Processing	Bovik	14320	A	(TISE)
Fall 1995	EE 380L.5	Engineering Programming Languages	Chase	14515	A	1
Spring 1996	EE 380L	Neural Networks for Pattern Recognition	Ghosh	14400	A	3
Fall 1996	EE 382M.1	Fault Tolerant Computing I	Abraham	14405	A	4
Spring 1997	EE 382C	Embedded Software Systems	Evans	14280	A	3
Fall 1997	EE 397K.1	Conference Course	Evans	15400	A	-
Spring 1998	CS 395T	Real-time Systems	Mok	48385	A	(CS)
Fall 1998	EE 382M	Comp Perf Eval and Benchmarking	John	15330	A	2
Fall 1998	EE 397K.1	Conference Course	Evans	15550	A	-
Spring 1999	CS 392C	Meth and Tech for Parallel Programming	Browne	49375	A	(CS)
Fall 1999	CS 384M	Multimedia Systems	Vin	50210	-	(CS)
Spring 2000	M 365C	Real Analysis	-	-	CR/NC	(Math)
Fall 2000	EE 381J	Probability and Stochastic Processes	-	-	-	(TISE)

Vita

Gregory Eugene Allen was born to Gordon Eugene and D'Maris Anne Allen on October 30, 1968, in Austin, Texas. He attended Austin's John H. Reagan High School, and graduated salutatorian in 1987. In 1991, Greg received his Bachelor of Science in Electrical Engineering from The University of Texas at Austin, graduating with Highest Honors. In 1993, he returned to The University as a part-time graduate student. In 1998, he earned his Master of Science in Electrical Engineering. Currently, Greg is a part-time Electrical Engineering Ph.D. student.

Greg has been employed with Applied Research Laboratories (ARL) at UT's J. J. Pickle Research Campus since the summer of 1986, when he was hired through the High School Apprentice Program. Since 1988 he has worked on high-frequency, high-resolution sonar systems in the Sonar Development Division of the Advanced Technology Laboratory at ARL. In addition to design engineering, Greg has been involved in system-level testing, installation, and deployment aboard U.S. Navy submarines, including a surfacing at the North Pole in 1993.

Greg and his wife, Dara Marie, have a daughter named Sabrina Fair, born in 1997. They live in northwest Austin.