Parallel Signal-Processing for Everyone

by

Brett W. Vasconcellos

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

February 2000

Author____
Department of Electrical Engineering and Computer Science
December 13, 1999

Certified by____
John V. Guttag
Professor, Computer Science and Engineering
Thesis Supervisor

Accepted by____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Parallel Signal-Processing for Everyone

by

Brett W. Vasconcellos

Submitted to the
Department of Electrical Engineering and Computer Science
on December 13, 1999, in partial fulfillment of the
requirements for the Degree of Master of Engineering in
Electrical Engineering and Computer Science

Abstract

We designed, implemented, and evaluated a signal-processing environment that runs on a general-purpose multiprocessor system, allowing easy prototyping of new algorithms and integration with applications. The environment allows the composition of modules implementing individual signal-processing algorithms into a functional application, automatically optimizing their performance. We decompose the problem into four independent components: signal processing, data management, scheduling, and control. This simplifies the programming interface and facilitates transparent parallel signal-processing. For tested applications, our system both runs efficiently on single-processors systems and achieves near-linear speedups on symmetric-multiprocessor (SMP) systems.

Thesis Supervisor: John V. Guttag
Title: Professor, Computer Science and Engineering

Acknowledgements

I would first like to thank to John Guttag for his guidance and support throughout my work on Pspectra.

I would also like to thank the members of the SpectrumWare group, without whom my contributions to this work would never have happened: Vanu Bose, Matt Welborn, and especially John Ankcorn for providing insight, both technologically and grammatically.

I am grateful to Katharine Riso for encouraging me to stay at MIT and finish my thesis, and for brightening my last year and a half.

I am also grateful for the years of support and encouragement my parents have lovingly bestowed upon me.

Contents

1 Introduction

As society becomes increasingly dependent on electronic devices for communication, entertainment, and medicine, the drive to develop smaller, cheaper, faster devices has made digital signal-processing increasingly important. Despite this, designing new signal-processing subsystems remains a difficult task. Devices traditionally use special-purpose signal-processing hardware, for which development tools remain relatively poor. Code is rarely reused; incremental changes are difficult to make and test; in general, development is a slow expensive process compared to mainstream software development.

We want to improve the way developers (1) develop signal-processing algorithms, (2) test and measure these algorithms using live data, and (3) integrate signal-processing with high-level applications. Specifically, we want these three tasks to be accessible to more people, and for development to be substantially faster and less expensive overall.

In order to achieve these goals, we must leverage what already exists. Building on others' work makes development much less time consuming and costly. What prior work can we take advantage of?

First, we consider hardware environments. Designing and debugging hardware is traditionally a very expensive part of building any device, especially when special-purpose hardware is required. We would like to reuse hardware designs from application to application.

Second, there are software tools: compilers, debuggers, profilers, libraries of existing code. We need to choose a platform for which these tools are available and well developed, a platform already in use by many developers.

Third, if we want to make signal-processing programming more accessible, we need to take common skills and experience into account. It is therefore important to choose a platform that supports commonly used high-level programming languages, making development easier for everyone and opening up the field to virtually anyone with programming experience.

A general-purpose commodity workstation is the clear solution that maximizes the possibilities for leveraging existing hardware, software, and experience. General-purpose hardware is inexpensive because of the large volumes in production. Much effort has already been put into software tools and libraries for general-purpose machines, resulting in significantly better tools than exist for other platforms such as DSPs. In addition, general-purpose machines are familiar to most programmers.

A few software environments (e.g., the Matlab Real-Time Workshop and Ptolemy [Pino *et. al.*, 1995]) exist for developing signal-processing algorithms and integrating them into usable subsystems that run on general-purpose machines. Unfortunately, these environments are generally for building simulations, not real-time systems. In addition, commodity workstations do not have a way of getting high-bandwidth streams of data into and out of the machine. A real-time signal-processing system needs some additional hardware and software to achieve our goals on a general-purpose machine.

The SpectrumWare signal-processing environment consists of both hardware and software additions to a general-purpose multiprocessor machine. A special PCI card for high data-rate, synchronous I/O, the GuPPI [Ismert, 1998], allows us to receive and transmit high-bandwidth signals, and to get this data into and out of the computer's main memory. This card is usable on any machine with a PCI bus, and requires only a small device driver for whatever OS the computer is running (we use Linux, but the driver could be ported to other environments). The software environment is user level C++ code that provides an API for programmers to write signal-processing algorithms and to join these algorithms into a functional system.

We would like our software development environment to aid programmers in building high-performance signal-processing subsystems for many applications. It should scale well from small tasks, such as receiving an AM or FM radio station, to more demanding tasks, such as our NTSC television reception application [Ankcorn, 1998], or even HDTV decoding or simultaneously processing many audio streams. Smaller tasks should require less powerful hardware, while complex tasks should be able to utilize multiprocessor machines. If we make our environment portable between different general-purpose hardware configurations, we can use less expensive, slower, hardware for some applications and expensive multiprocessor systems for others. Our environment needs to perform signal-processing efficiently on serial and parallel machines. In order to make parallel signal-processing accessible to as many programmers as possible (not just people who happen to be experts in writing parallel programs), we would like parallel processing to happen transparently and without changes to the serial code.

Our Parallel Signal-Processing Environment for Continuous Real-Time Applications (Pspectra) provides a portable environment that transparently scales signal-processing algorithms across multiple processors. Pspectra provides a usable platform for future digital signal-processing development and efficiently runs signal-processing code on any general-purpose symmetric multiprocessor (or single processor).

In developing Pspectra, we focused on DSP applications that we call "virtual radios." Virtual radios are radios implemented almost entirely in software, with hardware to digitize a wide-band portion of the RF spectrum and feed the samples (via the GuPPI card) into a general-purpose computer. In building a signal-processing environment designed for use in developing "virtual radios," we realized that the same techniques are applicable to a wide variety of "virtual devices" in which continuous, high data-rate, signal-processing can replace specialized hardware. Virtual radio applications include software radios used for applications such as cellular and CDMA base stations, HDTV reception and transmission, and wireless networking. Future virtual devices could include medical devices (e.g., EEGs, EKGs, ultrasound), radar and sonar equipment, and other applications for which DSPs are currently used.

In addition to those already mentioned, we list below some of the many advantages of signal-processing applications implemented on a general-purpose workstation.

- The software can dynamically alter operating parameters or even algorithms during operation. For example, Andrew Chiu developed a protocol for a software wireless network that allows the link to change operating parameters such as frequency, bandwidth, and modulation in response to changing environmental conditions and application requirements [Chiu, 1999].
- A portable environment allows us to easily upgrade the hardware to take advantage of performance improvements and decreasing prices of general-purpose machines.
- Because the same hardware can be used for many applications and types of algorithms, we can inexpensively prototype new real-time systems.
- Signal-processing applications can be easily integrated with existing applications (e.g., voice recognition software, MPEG decoders, networking software).
- New, more efficient, algorithms can often be developed on a general-purpose machine. For example, Matt Welborn designed a digital waveform transmission algorithm that takes advantage of the large main memory in these systems to efficiently synthesize waveforms [Welborn, 1999A].

1.1 Context

The two primary challenges in building any digital signal-processing system are input/output bandwidth and computational power. The most complex software radio applications, such as a cellular base station or a radar processing application, often require I/O speeds of 100-1000 Mbits/sec and processing performance in the range of 1-10 Gflops. Until recently, it was necessary to construct a special-purpose architecture with 100 or more DSP processors to achieve this level of performance. Many projects have proposed various combinations of processors, interconnects, and topologies as the ultimate platform for software radios (e.g., [Ledeczi and Abbott, 1994] and [Nalecz, *et. al.,* 1997]).

Modern processors have rapidly improved in performance and have added functionality to increase their signal processing performance (e.g., most processors now include SIMD instructions). In addition, bus and I/O speeds have increased dramatically. A single modern processor can now achieve over 1 Gflops for many signal-processing applications. For example, the newest Intel processor, the Pentium III, includes a technology called "Streaming SIMD Extensions" that can perform four 32-bit floating-point multiply-accumulates per two cycles and thereby achieve well over 1Gflop (e.g., 600MHz $\times$ 4 / 2 = approximately 1.2 Gflops peak processing power) [Intel, 1999B]. As processor clock speeds continue to increase, an SMP configuration with 4 or 8 processors will soon allow peak performance of over 10 Gflops. Lower end general-purpose processors (e.g., the Intel Celeron) can also provide performance at a cost comparable to high-end DSPs (e.g., the TI TMS320C62XX series)[1]. The GuPPI card, described in Chapter 2, provides 512 Mbits/s continuous I/O on a standard PCI bus. Future versions of the card could provide faster I/O on one of the newer buses.

Previous work on parallel digital signal-processing has primarily focused on special-purpose systems of DSP chips, rather than general-purpose workstations. Ptolemy [Buck *et. al.*, 1991] is a system for rapid-prototyping of DSP algorithms. Signal processing subsystems are built from reusable C++ modules, and Ptolemy generates DSP assembly code with a static scheduling algorithm for execution on a heterogeneous network of DSPs [Pino *et. al.*, 1995]. The software provides simulation tools for running the algorithms on a general-purpose machine, but does not support real-time operation on live data.

Others have built object-oriented C-based systems for distributed memory systems such as the Cray T3E [DeLuca *et. al.*, 1997] and the Mercury RACE series [Mercury, 1994]. Smaller proprietary systems containing multiple DSPs, such as the SpeakEasy system,

[1]For example, as of October 1999, http://www.ti.com/sc/docs/products/dsp/tms320c6211.html lists the TI TMS320C6211 with integer performance comparable to a 400MHz Intel Celeron at a similar price (around $40).

use a similar approach [Cook and Bonser, 1999]. These systems substantially ease the signal-processing development process, especially by encouraging modularity and code reuse. However, they are often tailored to a specific set of applications (e.g., radar signal processing). More importantly, distributed-memory platforms impede the ability of these systems to hide the details of parallel execution from the algorithm programmer. The management of the distribution of data arrays and matrices across the various processing elements seems to be particularly difficult to hide from the programmer.

The SpectrumWare group demonstrated the feasibility of building software radios on general-purpose processors with SPECtRA (Signal-Processing Environment for Continuous Real-Time Applications) [Bose, 1999]. As our research group pursued more challenging applications, such as receiving and transmitting digital signals such as HDTV or CDMA communications, we realized the original SPECtRA needed to be redesigned for higher-performance and lower-overhead operation. Although the processing requirements for such signals exceed the computational power currently available in a single general-purpose processor, SPECtRA only supports single-threaded operation.

1.2 Design Goals

The primary goal of Pspectra was to provide a tool to accelerate the development of signal-processing algorithms. We decided to build a system in which developers can write code on a general-purpose machine and run their signal-processing code in real-time on live data. Signal-processing "modules" can be swapped within a program without affecting other modules. A modular environment enables a programmer to focus on a particular step in the signal-processing chain and develop new algorithms to implement that functionality.

The various sub-goals considered in designing Pspectra fall into three categories: (1) Ease of use, (2) Efficiency/Performance, (3) Measurement:

- Ease of Use
  - The environment must provide a simple programming interface that supports easy construction of applications.
  - Parallelism should be hidden from the programmer as much as possible. The environment should provide simple mechanisms to help the programmer ensure correctness of algorithms when run on parallel machines.
- Efficiency/Performance
  - The overhead of Pspectra must be low. The proportion of signal-processing work to clerical work performed by the environment should be very high.
  - The system should be able to dynamically use available processors, achieving near-linear speedups.
  - The environment should have some automatic mechanisms for optimizing performance.
- Measurement
  - The system should facilitate measurements of individual signal-processing modules to aid in the design and testing of new signal-processing algorithms.

These goals are evaluated in the context of signal-processing algorithms and data-flow in "virtual radios." Our environment therefore works best, i.e., has the simplest programming interface and the best performance, when used with virtual radios. In building and evaluating Pspectra, we discovered that similar approaches could also be applied to many types of "virtual devices."

1.3 Contributions

We demonstrate a data-parallel based approach to signal-processing on a general-purpose symmetric multiprocessor (SMP) machine that is flexible and effective. We characterize which algorithms can directly use this approach and provide techniques for transforming algorithms to enable them to use this approach.

An "infinite stream" abstraction enables us to separate the signal-processing system into four independent components. We argue that this set of abstractions allows bounds checks and other overhead to be removed from processing inner loops, dramatically improving performance.

In addition, the Pspectra environment supports parallel execution of algorithms in a transparent manner. The applications we tested achieve near-linear speedups. We also demonstrate the effectiveness of optimizing for cache occupancy rather than merely the number of operations performed.

Pspectra is a full-featured, stable environment that can be used in a production setting to develop, debug, and test signal-processing algorithms and systems.

1.4 Roadmap

In Chapter 2, we briefly sketch the hardware architecture SpectrumWare uses to transfer high-bandwidth real-time data into and out of a workstation and the high-level Pspectra environment. Chapter 3 describes the details of the API chosen for Pspectra and the features available to the programmer, as well as the responsibilities placed on the module developer.

The next three chapters discuss the unique issues encountered in building a parallel signal-processing environment and the analysis of Pspectra's near-linear speedups. Chapter 4 gives an overview of previous work on parallel signal-processing systems and provides the motivation for Pspectra's interface to the programmer. Chapter 5 describes some interesting details of Pspectra's implementation on general-purpose SMP machines. Chapter 6 discusses the performance of the Pspectra system and the tradeoffs involved in building a "high-performance" system. We conclude with some thoughts on future applications for Pspectra and possible improvements.

2 Design / System Architecture

Development of Pspectra and the applications that use it has primarily been on Intel-based platforms, but the software is written in C and C++ using the standard POSIX threads (pthreads) interface and could be easily ported to any Unix –like system.

2.1 Hardware

Pspectra builds upon the same hardware architecture SPECtRA used [Bose, 1999]. This consists of a general-purpose computer with a GuPPI PCI card and some analog hardware to interface with the outside world. In the case of "virtual radios" (see Section 1.2), SpectrumWare's current focus, this hardware handles reception and down conversion of a wide-band portion of the RF spectrum, as well as transmission. The GuPPI card contains A/D and D/A converters and logic to directly transfer samples to and from the computer's main memory [Ismert, 1998]. As part of the work described here, I implemented a new GuPPI driver that provides very low overhead read/write calls that allow the user-level program to receive or transmit an arbitrarily large block of data with a single system call.

2.2 Why SMP Machines?

There are basically three architectures for bringing parallel processing power (other than instruction-level parallelism) to bear on a problem: 1) a general-purpose symmetric multiprocessor (SMP) machine, 2) distributed computing on a network of machines, or 3) a distributed-memory multi-processor machine.

We chose SMP machines over the other two choices because the memory hierarchy transparently handles the low-level details of interprocessor communication and concurrency. The interface used for parallel programming on a particular architecture cannot, in general, be completely hidden from the programmer. Programmers are much more familiar and comfortable working with the shared-memory paradigm implemented by SMP than the message-passing interfaces used with distributed-memory machines or networks. Completely hiding the message-passing interface would be difficult; either the system would become less flexible or we would have to expose a messy and difficult parallel programming interface to the programmer. Using shared-memory SMP machines keeps the software simpler and makes the environment accessible to more developers.

SMP machines also have a larger code base, enabling us to integrating existing software libraries (such as MPEG encoders/decoders, voice recognition software, etc.) into our applications. Another big advantage over distributed-memory systems is our ability to run a commodity operating system (Linux) and to use its development tools: compilers, debuggers, profiling tools, etc.

2.3 Modular Programming Environment

The basic structure of the programming environment is interconnected signal-processing modules. Modules are intended to be reusable and contain signal-processing code for a particular algorithm (e.g., FM demodulation or NTSC sync-pulse detection). This modular structure allows new algorithms to be swapped in for older algorithms, facilitating the development of new signal-processing techniques. The system can also measure the performance of each module separately, helping programmers build more efficient algorithms.

Pspectra's provides an "infinite stream" abstraction for all communication between modules. This interface allows us to separate the signal-processing system into four independent components:

- the setup and control of modules (e.g., the modification of parameters via some user interface),
- the actual signal-processing algorithms (encapsulated in the various modules),
- the dynamic scheduling of the various signal-processing modules using a generic scheduling algorithm, and
- data management and buffering, including the sharing of data between processors.

Programmers using the Pspectra environment write the first two of these components, which change from application to application. We built generic scheduling and data management algorithms into the Pspectra system.

Programmers using Pspectra write two different types of code: signal-processing algorithms and related control code. All code related to a particular signal-processing task resides in a single C++ class (or "module"). To produce the final signal-processing application, the top-level control program creates and initializes the modules, connects them together, and initiates the Pspectra signal-processing loop. This control program also handles any other components of the program, such as the GUI.

2.4 Infinite Streams

Modules exchange data with other modules through "infinite" data streams. Each module may have multiple outputs and multiple inputs. All in-band interaction between a pair of modules is confined to a single, one-way, "infinite" stream of data connecting the modules. Data is generated by "sources," processed by some chain of modules, and consumed by "sinks." Sinks "pull" data downstream – this enables the system to avoid generating intermediate data that is not necessary for the production of the final output data.

Locations in a stream are identified by a 64-bit index, numbered sequentially beginning with zero. Modules use indices to reference ranges of data in their input and output streams. Pspectra then provides modules with pointers to the actual data in memory. This allows the environment to separate the implementation of "infinite" streams from both the signal-processing and the scheduling of the various signal-processing tasks.

All data storage in Pspectra is encapsulated in infinite streams. We implement infinite streams with fixed-size circular buffers that use a virtual memory trick to ensure data never has to be copied and always appears contiguous (see Section 5.1.1). Each buffer keeps track of which portions of data have been written by the corresponding upstream module and are therefore safe for use by downstream modules. On multiprocessor systems, buffers also track portions of data the different processors are producing to avoid replicating work. Every buffer has associated "connectors," which connect the buffer with downstream modules reading data from the buffer. Connectors keep track of what portions of the buffer are in use by different processors and allow the buffer to decide when it is safe to reuse space in the buffer. Modules with multiple inputs have multiple connectors associated with different buffers (see Figure 6 in Chapter 5).

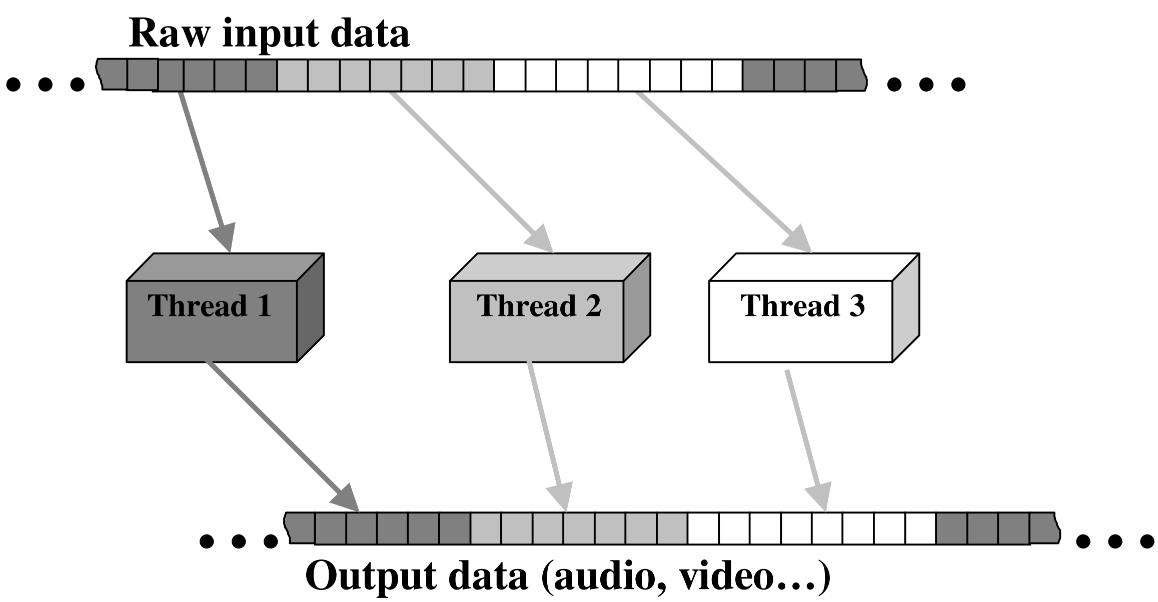

Figure 1 Multiple threads process different data in parallel (a "data parallel" paradigm)

Modules' use of indices enables the environment to transparently run a module in parallel on different portions of data (see Figure 1). If the processing produces the same results independently of the order of requests for output blocks, the module can take advantage of multiple processors without explicit coding. Some signal-processing algorithms do not meet this condition; Section 3.3 discusses strategies for modifying a module to allow it to produce blocks out-of-order correctly or for serializing the module when this modification is not possible.

Multidimensional streams of data (e.g., video, audio from an array of microphones, etc.) can be composed into a one-dimensional stream of data by choosing an appropriate access pattern for the data (e.g., video scan-lines). Different access patterns may improve performance for different signal-processing algorithms [Watlington and Bove, 1995].

2.5 The Signal-Processing Loop

After the initial setup of the modules and of the environment, Pspectra runs a two-phase loop (discussed in Section 5.2). During the first phase (the "data-marking" phase), the scheduler finds one or more modules whose inputs are "ready," that is, they have already been computed by the appropriate upstream module. This module or modules can therefore produce data without blocking. The scheduler "marks" this output data and moves onto the second phase.

During the second phase (the "work" phase), the signal-processing modules process the marked data. When a module finishes producing a portion of data, Pspectra marks the data as completed, or "ready," so that in future iterations the scheduler knows that other modules dependent on that data can be scheduled.

We run this loop on a single-processor system without the use of extra threads by performing other tasks (e.g., the GUI) between iterations of the loop (see Section 3.4).

On multiprocessor systems, this two-phase loop runs in parallel on each processor, with synchronization mechanisms only necessary in the scheduling phase. Because Pspectra spends the bulk of the processor time (often more than 99%, see Section 6.2) in the second phase (doing the actual signal-processing work), lock contention is low and the overhead from the scheduler is minimal. We can reduce overhead by marking and working on larger blocks of data, since the cost of running the scheduler is independent of the size of the blocks that it marks. Section 6.4 discusses Pspectra's mechanisms for automatically optimizing system performance.

3 Pspectra Programmer's View

This chapter describes the programming interface to Pspectra and the ways the module programmer can help Pspectra achieve maximum performance. We kept the Pspectra interface as simple as possible, while providing maximum flexibility in module and topology design.

3.1 Basic Module Structure and Requirements

This section describes the structure of a Pspectra signal-processing module and the responsibilities placed on the implementor of the module. Well-written modules are essential for efficient performance of the system.

Modules refer to data using indices (see Section 2.4). To refer to a range of data in an input or output stream, a *VrSampleRange*[2] object is used, which contains a starting index and a size (see Figure 2). Streams do not always produce constant rate output data; modules choose what data to store at any particular index.

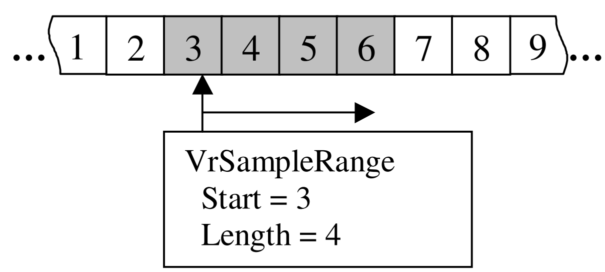

Figure 2 A range of data within an "infinite" data stream

Each block is numbered with its index and represents one unit of data.

Programmers implement modules as C++ classes that extend the Pspectra base class *VrSigProc*. This section describes the functions that a module must implement to interact with the system, divided into functions that are run once during initialization and functions that are run repeatedly in the signal-processing loop.

[2] Vr stands for Virtual Radio, see [Bose *et. al.*, 1998B] for a discussion of Virtual Radios.

To understand how these functions interact to influence the behavior of a module, examine Figure 3. The environment calls *initialize()* and *mapSizeUp()* once during the initial setup of the signal-processing topologies (see Section 3.1.1). Then *forecast()* and *work()* are called alternatively. Pspectra's scheduler uses the data-dependencies returned by *forecast()* to determine what upstream data must be computed first. Pspectra then computes this data (by calling *forecast()* and *work()* on other modules), and finally instructs the original module to compute its output by calling *work()*.

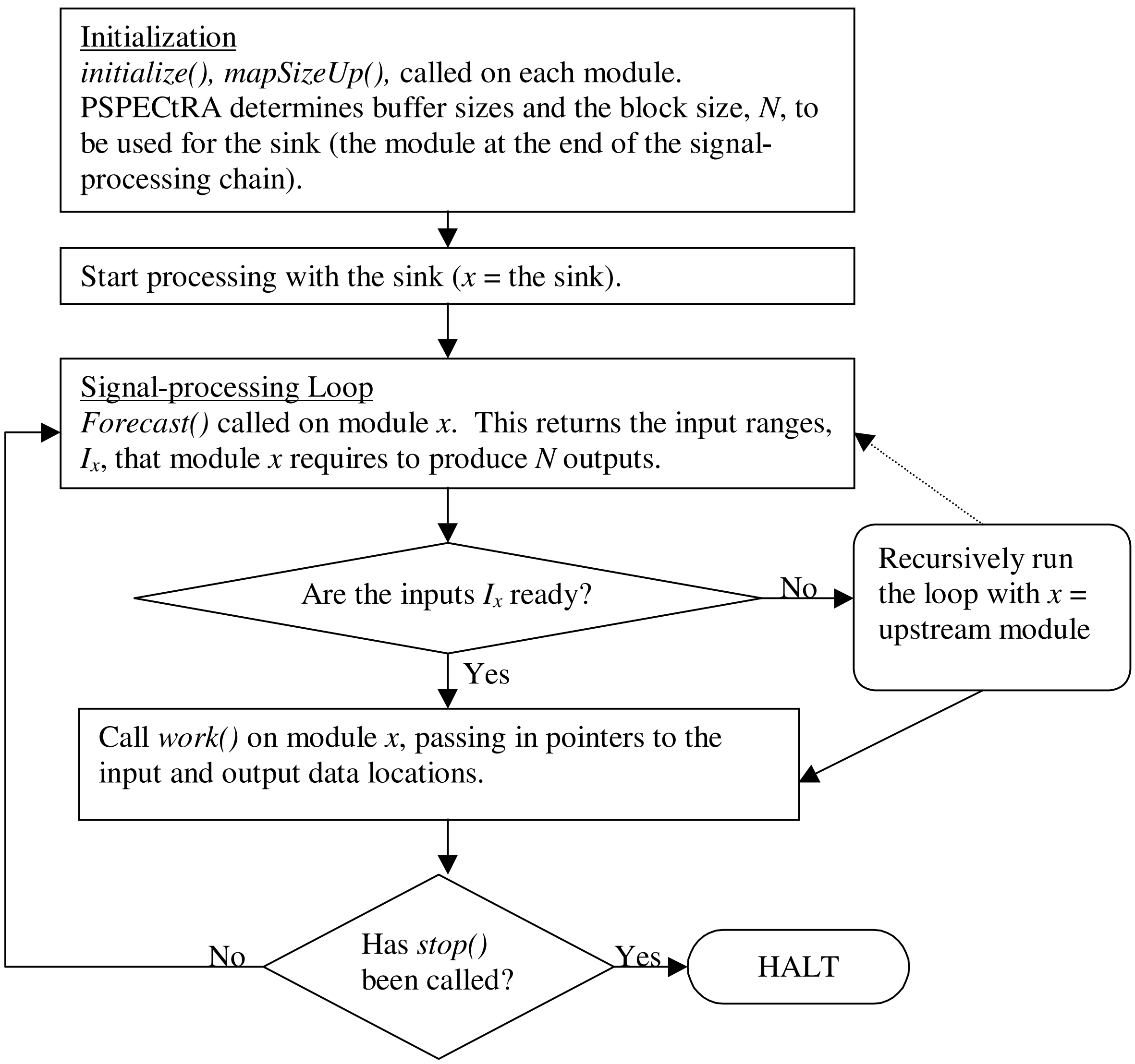

Figure 3 Flowchart illustrating the operation of PSPECtRA

The separation of *forecast()* and *work()* allows the system to perform all data management and scheduling functions and frees the signal-processing algorithm in *work()* from overhead such as checking that its input data has been computed or that space is available in its output buffer.

Initialization begin at the "sinks" that terminate the chain of processing modules and propagates upstream through all the modules. Scheduling proceeds in two-phases, a "data-marking" phase, which proceeds upstream, following data-dependencies in a "data-pull" fashion, and a "work" phase, which proceeds downstream to produce the output data.

3.1.1 Initialization

Before signal-processing can begin, Pspectra must initialize the modules and the buffers that implement the "infinite" streams. This process accomplishes three things:

- Pspectra calls *initialize()* on each module to allow the module to initialize module-specific parameters,
- Pspectra determines the size required for each buffer based on the number of processors, the block size for each sink, and the behavior of upstream and downstream modules (as determined by calling *mapSizeUp(),* see Section 5.1.1 for a discussion of how the buffer sizes are chosen), and
- Pspectra attempts to determine the optimal block size for each sink (see Chapter 6).

3.1.1.1 initialize()

Pspectra calls *initialize()* to allow the module to perform any necessary setup after the module has been connected to other modules. Because input and output sampling rates are properties of the individual infinite streams in Pspectra, module parameters that depend on sampling rate should be configured in *initialize()* (since the streams do not exist when the constructor is called). A typical example is a filter that computes its tap values in *initialize()* (see Section 3.1.3).

A module may also wish to call *setOutputSize()* in its *initialize()* procedure to fix the smallest number of units on which the module will run (for example, a particular interpolating filter may create three output points per input and thus have a natural *outputSize* of three). Pspectra guarantees w*ork()* and *forecast()* are called with a multiple of *outputSize*.

3.1.1.2 mapSizeUp()

MapSizeUp() returns the maximum number of input samples required for the module to generate *N* output samples. *MapSizeUp()* allows Pspectra to determine the space required for data buffers to hold all the necessary data for the operation of the system (see Section 5.1.1).

3.1.2 Signal-Processing Loop

After the initial setup of the modules and of the environment, Pspectra runs a two-phase loop on each available processor. The first phase is the "data-marking" phase, during which Pspectra uses *forecast()*'s return values to find modules whose inputs are ready. Pspectra "marks" these modules' output data to tell other processors not to work on the same data. In the second phase, Pspectra calls the *work()* procedures of these modules to instruct them to produce the marked data.

3.1.2.1 forecast()

The module developer must implement this method to inform Pspectra what input data is necessary to produce a particular range of output data. For example, a filter with a history of *h* would request $n+h$ units of data to produce *n* outputs. Pspectra uses *forecast()* to determine what work needs to be done upstream and to schedule work appropriately on the available processors. If the required inputs are not precisely known, the module should return a best guess.

If the range returned by *forecast()* does not include all the necessary data, *work()* will later determine that the guess was incorrect (see Section 3.2.1). In this case, Pspectra will call *forecast()* again. This mechanism results in a performance loss, so *forecast()* should err on the side of returning a larger range than necessary, in order to avoid rescheduling. Too large of a range, however, will also results in performance loss and increased latency in the system.

One caveat: Pspectra assumes that input data use will be processed in a monotonically increasing fashion. Everything preceding the input data range returned by *forecast()* is effectively thrown out and unavailable to produce later outputs. If the module might need this data at a later time, it must include this data in the range returned by *forecast()*.

3.1.2.2 work()

This is the heart of every module, where the actual signal-processing takes place. Pspectra provides as arguments pointers to input and output data locations in memory, along with indices corresponding to the location of the data in the "infinite streams." *Work()* then enters a loop to produce the requested outputs and returns the actual number of outputs it was able to produce from the available inputs, which should usually be equal to the number requested. If the module needs more input data than originally requested (with the return value of *forecast()*), *work()* returns without computing all the output data and *forecast()* is called again by Pspectra to determine what data is now needed (see Section 3.2.1).

3.1.3 Example Module: FIR Filter

This is somewhat simplified C++ code for an FIR filter. Note that modules can have multiple inputs and multiple outputs, which explains the need for arrays in the arguments for *forecast()* and *work()*. *Forecast()* ensures that there will be enough input data that the expression *inputArray[j]* always corresponds to valid data.

```
void VrFilter::initialize()
{
  taps=buildFilter(numberOfTaps);
}

unsigned int VrFilter::mapSizeUp(int inputNumber,
                                 unsigned int size) {
  return size + history - 1;
}
```

```
void VrFilter::forecast(VrSampleRange output,
                       VrSampleRange inputs[]) {
  for(unsigned int i=0;i<numberInputs;i++) {
    inputs[i].index=output.index;
    inputs[i].size=output.size + numberOfTaps-1;
    //numberOfTaps is the size of the taps[] array.
  }
}

int VrFilter::work(VrSampleRange output, oType *o[],
                   VrSampleRange inputs[], iType *i[]) {
  float result;
  unsigned int size = output.size;
  for (;size>0;size--,i[0]++) { //increment pointer to input # 0
    iType* inputArray = i[0];

    /* Compute the dot product of inputArray[] and taps[] */
    result = 0;
    for (int j=0; j < numberOfTaps; j++)
      result += taps[j] * inputArray[j];

    *o[0]++ = (oType) result; //output the result to output # 0
  }

  return output.size; //all outputs completed
}
```

3.2 Special Cases

This section describes some of the more advanced features of Pspectra's interface, which only benefit a select set of modules.

3.2.1 Unpredictable Input:Ouput Ratios

Pspectra was designed to work best with predictable signal-processing modules that facilitate a data-parallel strategy. However, a few signal-processing modules cannot take advantage of this strategy and must be run serially. Many other signal-processing modules run very predictably with occasional corrections necessitated by data loss or loss of synchronization. This section provides some pointers for making these unpredictable and semi-predictable modules run well under Pspectra.

To handle unpredictable, or data-dependent, input:output ratios, *forecast()* guess what input data the module will need and *work()* returns the actual number of outputs (less than or equal to the number requested) produced using this input data. If Pspectra asked the module to compute n outputs and *work()* discovers that the input ranges *forecast()* requested are in fact inadequate, then *work()* returns the number of outputs it was actually able to generate, m. Pspectra calls *forecast()* and *work()* again to complete the n-m unfinished outputs. Note that this causes the scheduling algorithm to be run extra times and can degrade performance. However, the programmer can take steps to make rescheduling rare. We will discuss techniques for the two types of modules that generally use this functionality: variable-rate modules and lossy data-reception modules.

3.2.1.1 Variable-Rate Modules

A good example of a variable-rate module is a compression (or decompression) module. Since a compression module eventually uses all its input data, there is no penalty for requesting more data than is necessary for the production of the current block of output data. Therefore, the best way to handle the fact that the compression rate is unknown is to define *mapSizeUp()* and *forecast()* so they assume the maximum compression ratio expected under normal operation. (Choosing too large of a number increases both the size of the buffers and the latency of the module, however.) This means *work()* will almost always have enough data to produce n units of output. In the rare case that the compression rate exceeds this maximum, *work()* returns the number of outputs it was able to produce from the inputs *forecast()* requested. Pspectra then reschedules the module to finish the rest of the output data. A decompression module would assume a compression ratio of 1:1 (no compression) and would thereby never need to request rescheduling.

3.2.1.2 Lossy Reception

As an example of lossy reception, consider a wireless network receiver module that throws out packets that have bad checksums. Pspectra may call *forecast()* and *work()* to produce, for example, 10 packets. The module, through *forecast()*'s return value, tells Pspectra that it needs 10 packets of input to produce 10 packets of output. *Work()* may then look at these 10 packets and pass on only 7, discarding 3 bad packets. Pspectra would then call *forecast()* and *work()* again to finish 3 more output packets. If packet loss is a common event, this rescheduling can be avoided by having *forecast()* (and *mapSizeUp()*) assume a loss rate of, for example, 50%. In this case, *forecast()* would produce overlapping requests for 20 input packets to meet downstream requests for 10 output packets. (For example, first requesting packets 1-20, then 11-30, 21-40, etc.)

Note in the original example here that *mapSizeUp()* would return 10 when given an argument of 10, even though the module may need more than 10 input packets during actual operation – the key here is the module may change which 10 it examines by forcing Pspectra to call *forecast()* again (effectively sliding the input window as necessary).

A reception module that must stay synchronized with the incoming signal (e.g., television reception) may use a similar approach by periodically skewing which inputs the module requests to produce outputs.

3.2.2 Unused Inputs

Modules may have input streams that go unused during the production of a particular block of output data. For example, a particular multiplexing module may read 1000 bytes from one input module followed by 1000 bytes from another input module. In this case, *forecast()* must still return a *VrSampleRange* for the unused input(s). This range should be a zero length range to indicate no data is needed, with the starting index equal to the smallest index that the module may request in the future. Pspectra uses this information to determine what data can safely be thrown out.

3.3 Designing Modules for Parallelism

Many common signal-processing algorithms can take advantage of Pspectra's implicit parallelism, while others require explicit control of parallel execution or cannot be run in parallel at all.

For Pspectra's implicit parallelism to work, a module's *work()* procedure must produce the same outputs regardless of what order blocks are produced in. For example, producing block N before block N-1 should result in the same output data as producing these blocks in order. The combination of Pspectra's scheduling algorithm and the uncertainty in the relative execution speed of different threads makes this order unpredictable.

Modules that satisfy the necessary out-of-order condition will run correctly when Pspectra runs them in parallel. The programmer need do nothing special other than verify out-of-order correctness for his module. Pspectra will run *work()* on multiple processors to produce different blocks of output data.

This out-of-order correctness condition may be hard for the average programmer to verify. A tool for verifying this condition would make this easier, but does not currently exist. One simple sufficient condition is that a module's *work()* procedure never modify the module's state variables or other observable external state. A module that meets this condition also meets the necessary condition for correctness.

If a module does modify state in its *work()* procedure, the programmer must either manually eliminate race conditions or else perform one of the fixes described below. Otherwise, the module may output corrupt data or cause more serious system errors. The next two sections discuss techniques for modifying an algorithm that does not meet the out-of-order correctness condition either to avoid the use of state, or to force Pspectra to serialize the module.

3.3.1 Transforming a Module to Remove State

Many signal-processing algorithms that seem to require serial execution can often be modified slightly to allow parallel execution. For example, for many IIR filters, truncating the input to a reasonably small history results in negligible errors in the filter's output. For these IIR filters, we can implement a close approximation that does not change its state as it processes data: for every output block, the filter uses enough extra input samples to ensure the first output sample is reasonably accurate. This allows blocks to be computed independently of each other. If the filter's history is reasonably small compared to the block size, the additional overhead is negligible. This enables parallel execution and also preserves the ability of Pspectra to avoid computing data that isn't needed downstream.

3.3.2 Serializing a Module

When a module's *work()* procedure needs to force data to be computed serially instead of in parallel, it calls *sync()*, which allows *work()* to safely use state that the module modified during the production of past outputs. *Sync()* ensures that no other processors are computing (or will later compute) data that precedes the data this thread is computing. In other words, all previous data has been completed. Other threads may be running *work()* on later data, however. For example, in Figure 1 above, if Thread 2 calls *sync()*, Thread 2 will wait for Thread 1 to finish but the call will not affect the operation of Thread 3 (since from Thread 2's perspective Thread 3 is operating in the future).

The variable-rate compression example introduced above (Section 3.2.1.1) would have to use the *sync()* call since the module only knows exactly what input data is needed to produce a block of output data after it has finished the previous block of output data. An IIR filter also requires all the outputs to be generated in order, and therefore requires the use of *sync()*, unless we apply the transformation described in the previous section.

Using *sync()* prevents Pspectra from running the module in parallel. Other modules can still run in parallel, but the system will demonstrate superior performance when it can run all modules in parallel.

3.4 Meta-Modules

One element contributing to Pspectra's overall ease-of-use is its ability to bundle multiple signal-processing modules into a single larger module. The larger module can be exported to other programmers for use in their systems while maintaining an abstraction layer that allows the sub-modules implementing it to be changed or rearranged.

Meta-modules can be built from any signal-processing modules. For example, consider the two modules *VrSyncDetect* (which detects sync pulses in a signal and outputs indices corresponding to their location in the input data stream) and *VrDecodeFrame* (which decodes the data framed by sync pulses). These modules can be combined into a single module *VrDecodeSyncedData* for easier use in applications. The meta-module does no processing itself, but simply creates the appropriate sub-modules, connects their inputs and outputs, and provides methods to access visible methods on the component modules.

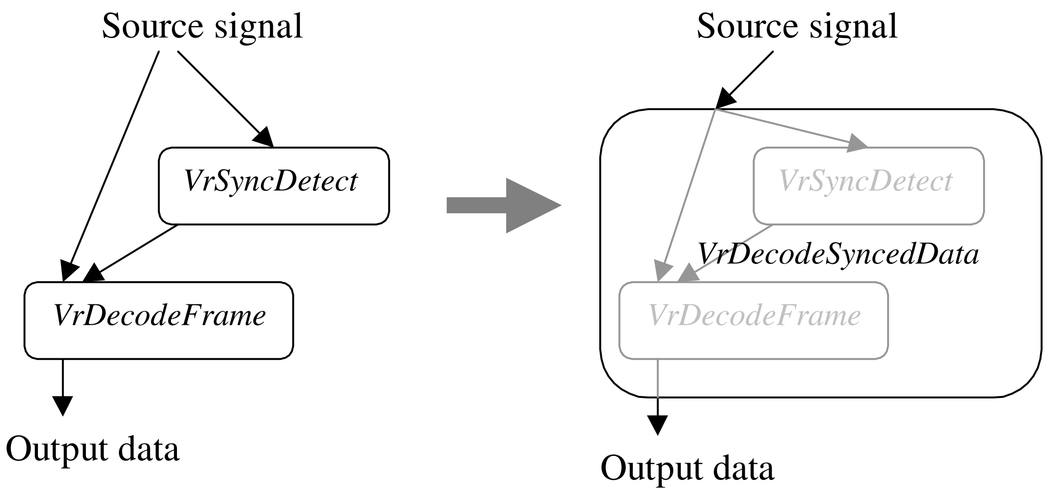

Figure 4 An example meta-module

A meta-module creates its component modules in its constructor and overrides *connect()*, which connects the module's inputs to an upstream output buffer, and *getOutputBuffer()*, which returns the module's output buffer (see Section 5.1). The meta-module's *connect()* and *getOutputBuffer()* implementations redirect inputs and outputs respectively by calling the corresponding methods on the components modules.

For example, *VrDecodeSyncedData* would override *connect()* to call *connect()* on each of its component modules to connect each of them to the data source, and then *connect()* on *VrDecodeFrame* to connect its second input to *VrSyncDetect*'s output. The meta-module would override *getOuputBuffer()* to call *getOuputBuffer()* on the component *VrDecodeFrame* module, overriding the default behavior of creating a separate buffer for *VrDecodeSynchedData*.

3.5 Control Program

The control program creates all the modules, configures their parameters, connects them in an appropriate topology, and enters a processing loop. It initiates by calling *start()* on a sink (or on *VrMultiTask*, which handles topologies with multiple sinks). Calling *start()* runs the initialization process described above (Section 3.1.1) and begins the actual signal-processing.

After calling *start()*, the control program loops, calling *process()* repeatedly on the sink, which runs an iteration of Pspectra's scheduling algorithm. The control program may perform other tasks, such as running a GUI, or dynamically changing parameters in the system, between calls to *process()*. On multiprocessor systems, *start()* creates new threads to perform the signal-processing and *process()* does nothing. The program may end the processing and cleanup any remaining threads by calling *stop()* on the sink.

Here is an example Pspectra program that demodulates an FM channel, filters the audio, and plays it on the computer's speaker.

```
void main() {

  /* Create the signal-processing modules */
  VrGuppiSource<char>* source = new VrGuppiSource<char>();

  VrComplexFIRfilter<char>* channel_filter =
    new VrComplexFIRfilter<char>(CFIRdecimate,cTaps,
                                 chanFreq,chanGain);

  VrQuadratureDemod<float>* demod = new
     VrQuadratureDemod<float>(quadGain);

  VrRealFIRfilter<float,short>* if_filter =
    new VrRealFIRfilter<float,short>(RFIRdecimate,realCutoff,
                           realTaps,realGain);

  VrAudioSink<short>* sink = new VrAudioSink<short>();

  /* Connect Modules */
  CONNECT(sink, if_filter, audioRate, 16);
  CONNECT(if_filter, demod, quadRate, 32);
  CONNECT(demod, channel_filter, quadRate, 64);
  CONNECT(channel_filter, source, gupRate, 8);
```

```
  /* Start System */
  sink->start();
  while(sink->elapsedTime() < SECONDS)
    sink->process();
  sink->stop();

}
```

Note that modules in Pspectra almost always use templates that allow different data types to be used. The two filters shown take parameters to configure the number of taps to use, the center or cutoff frequency for the filter, and a gain. Both filters also decimate the signal, producing a lower-rate output. *CONNECT* connects modules together, and optionally takes sampling rate and bits-per-sample parameters to perform a sanity check on the modules passed to *CONNECT*. The signal-processing loop in this example simply runs the system for *SECONDS* seconds.

4 Parallel Signal-Processing

This chapter outlines past work in parallel signal-processing and the weaknesses in past systems that inspired Pspectra. Past work is basically divisible into two categories: task parallelism and data parallelism.

4.1 Task Parallelism

Task parallelism (Multiple-Program, Multiple-Data, or MPMD) can take two forms: designers achieve parallelism either through pipelining or by splitting multiple channels across different pieces of hardware. A pipelined system splits the various signal-processing tasks across different physical processors (e.g., one processor running a filtering algorithm, another performing demodulation, etc.; see Figure 5). Large multi-channel systems split the processing so that each channel (e.g., in a mobile base station) is processed on a separate DSP chip (e.g., AirNet Communications Corporation's GSM base station). Some systems may even use a combination of these two forms of task parallelism, using a separate pipeline of processors to process each channel.

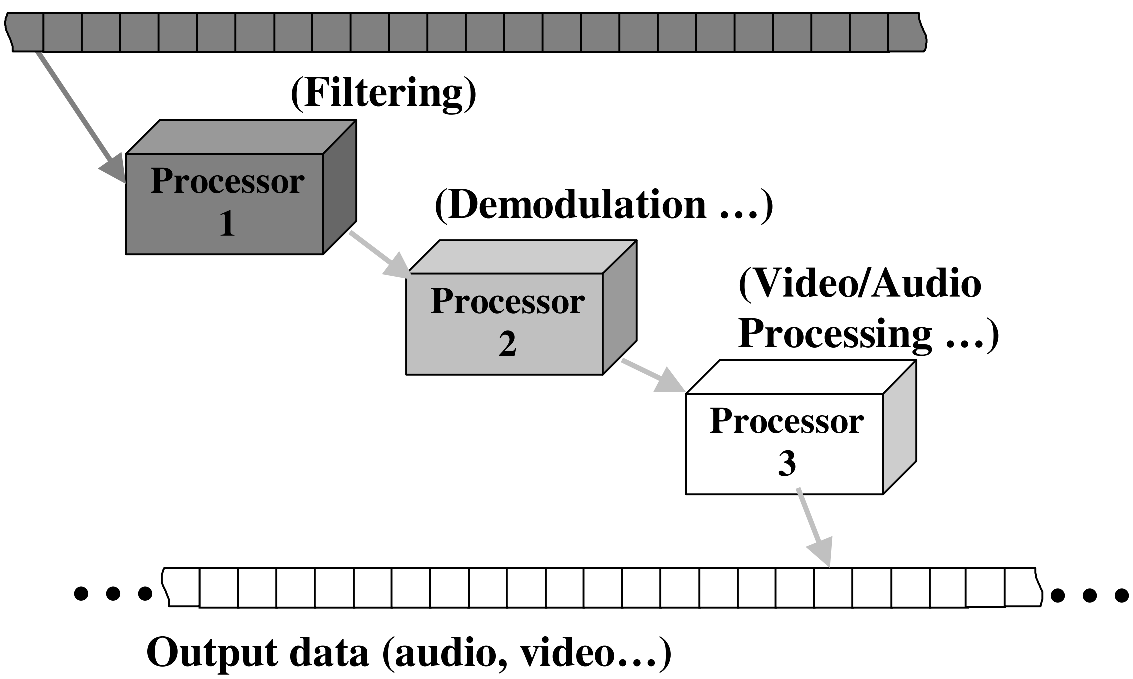

Figure 5 Task Parallelism

Task parallelism greatly simplifies the hardware design. The design is similar to the corresponding analog systems, with digital processing substituted for the analog components and subsystems. The pipelined approach allows a modular design of both hardware and software; different development teams can work on each task separately. Different DSPs are chosen for different portions of the signal-processing system based on their suitability for particular algorithms, and to balance the pipeline, allowing the hardware to be tailored to the application at hand. In addition, a straightforward implementation of the physical datapaths for passing signals from one processor to the next, mimics the datapaths in the analogous analog circuit. Unfortunately, task parallelism faces three real-world problems: poor scalability, poor accommodation of incremental improvements, and inefficient use of resources. Designers may make a trade-off between these last two. For example, a system that very efficiently utilizes the resources available leaves no room for the later introduction of more processor-intensive algorithms.

Task parallelism scales very poorly in general. The number of tasks present in the signal-processing chain limits the maximum number of processors. Very processor intensive algorithms (e.g., radar or real-time image processing) require another strategy from the start, since no one processor can provide enough computational power for a single task in some of these algorithms.

Second, task parallelism encourages customization of the hardware to the current application resulting in special-purpose hardware with poor sensitivity to algorithmic upgrades. For example, if the designer wishes to replace a particular channel filter algorithm running on existing hardware with a more computationally intensive algorithm in order to eliminate unforeseen interference, the new algorithm may exceed the computational power of the DSP allotted to it. The hardware must be redesigned, even if some of the other DSPs are not fully utilized by the tasks assigned to them.

This inflexibility is also the root of the third failing of task parallelism: inefficiency. Tasks either utilize one processor exactly to its maximum capacity, leaving no room for upgrades, or they leave computational capacity unused. In addition, unused capacity often cannot be pooled together, so, for example, extra cycles available on the processor doing the demodulation cannot be used to help beef up the channel filtering algorithm running on another processor. In addition, if one of the tasks in a system is too processor-intensive for any single processor, task parallelism fails completely.

4.2 Data Parallelism

The second methodology for parallel digital signal-processing systems is data parallelism (Single-Program, Multiple Data, or SPMD). Systems employing this strategy partition the data (e.g., in time). This allows much better scalability, in theory, since every processor runs the same program, and adding processors simply requires splitting the input data into more blocks. Data parallel systems also have more potential to accommodate software-only algorithm changes and to dynamically change their operation (e.g., using less processing power, and thus less electrical power, when environmental conditions are better than the worst case). Data parallel systems can also encourage more software reuse since algorithms always run on similar hardware.

Traditional DSP designers have trouble implementing data parallel systems for two reasons. First, DSPs usually have a very limited code space, preventing them from efficiently running all the tasks in a signal-processing chain. Second, communication among DSPs is much more complex than in task parallelism and requires specifically designed interconnects, often using a message-passing protocol. Still, many data parallel systems have been built, including Mercury Computers' RACE® Architecture and radar processing systems such as [Nalecz, *et. al.*, 1997]. These systems have made great strides into more flexible and powerful software signal-processing systems, but still require the programmer to include explicit parallelism in the algorithm and rely on a complex application-dependent control program for efficient operation. Most systems also provide no ability to dynamically change their operation (e.g., to react to changing environmental conditions).

4.3 Where Pspectra Fits in

As illustrated in the previous section, data parallelism appears to offer great promise for building a scalable, flexible, and efficient signal-processing system. Ease-of-use is the biggest hurdle to overcome. Choosing general-purpose SMP machines avoids many of the difficulties encountered in message-passing systems. Memory subsystems on SMP machines guarantee cache coherency, which allows processors to transparently share intermediate data. Pspectra's "infinite streams" build on this memory model by tracking which data each processor is using and recycling buffer space as needed.

Pspectra is not a strictly data parallel system, however. Rather, it incorporates a combination of task and data parallelism in its scheduler; each processor does not always run every task in the signal-processing chain, but may run one or more tasks as necessary. This method of scheduling is more robust when the system must respond dynamically to unforeseen environmental changes (e.g., if synchronization is lost and extra work needs to be done in the synchronization module in order to resynchronize). It also results in lower overhead for signal-processing chains in which there are large decimation or interpolation factors. If a processor tried to run such a chain from start to finish, the modules at the lower sampling rates would produce many fewer samples per iteration of the scheduler, and thus overhead would increase relative to useful work done (see Section 6).

Although we chose SMP machines for the first implementation of Pspectra, a message-passing version of Pspectra could provide a similar interface for parallel signal-processing on massively parallel machines or networks of workstations (see future work, Section 7.1.1). Message-passing would complicate the implementation of the infinite streams, of the scheduler, and of the modules themselves. In addition, ensuring good performance would require different mechanisms than those discussed in Chapter 6.

The suitability of general-purpose processors for signal-processing has been established, especially because they are relatively strong in most of the areas useful for signal processing. For example, many processors now include SIMD instructions, which accelerate many common signal-processing algorithms; see Section 1.1 for more details. Pspectra achieves a flexible and simple interface for programming signal-processing applications on general-purpose SMP machines.

5 Buffering and Scheduling

Because of Pspectra's partitioning strategy, algorithms avoid performing any scheduling or data buffer management in their inner loops. This minimizes system overhead compared to real work done by the algorithms. The data buffers allow processors to share intermediate data in the data parallel approach to signal-processing, as described in the previous section.

5.1 Dataflow Implementation

Signal-processing modules in the Pspectra system communicate data using "infinite streams" as an abstraction for input and output. Each module can directly read and write a range of data in a steam using a pointer provided by Pspectra. The streams are implemented using two classes, *VrBuffer* and *VrConnect*, which implement two abstractions: "buffers" and "connectors" (see Figure 6).

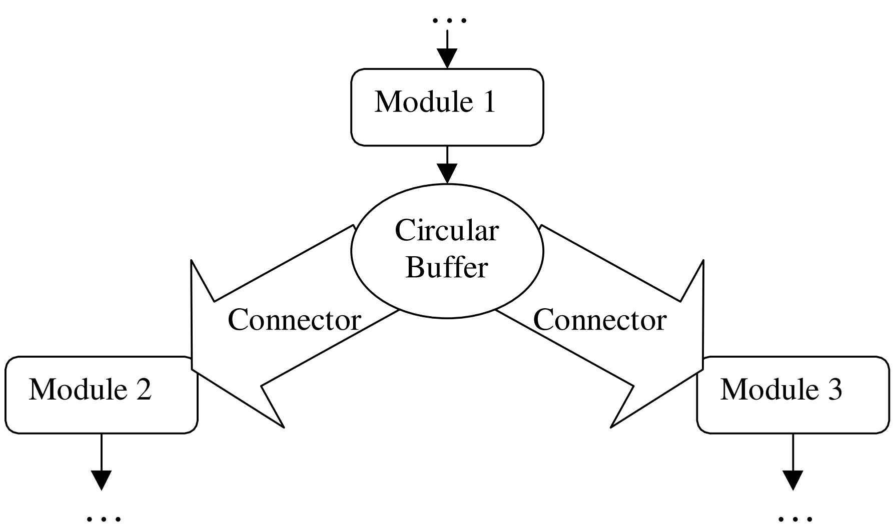

Figure 6 Buffer and Connectors

Modules may have multiple outputs (e.g., a module that separates the left and right audio stream from a stereo input), each represented by a *VrBuffer* object. The "buffer" is simply a circular data buffer with a single writer (the module) and multiple readers (the "connectors"). A single writing module can have multiple, parallel instantiations, which can simultaneously write disjoint portions of the buffer.

Each input of a module is a *VrConnect* object associated with a particular *VrBuffer* object. The separation of the infinite stream functionality into these two objects simplifies allowing multiple downstream modules to connect to a single *VrBuffer* object. Downstream modules share data produced by the upstream module, avoiding the recomputation of data used by multiple modules. In addition, data produced and stored in the buffer by one processor may be read by the other processors.

We will discuss three implementations of the buffers: *VrBuffer*, which buffers the output of most modules; *VrGuppiBuffer*, which shares the data in the buffer with the GuPPI driver; and *VrMUXBuffer*, which efficiently redirects reads to one of a number of other buffers. *VrGuppiBuffer* and *VrMUXBuffer* extend the *VrBuffer* class and thus share the same "infinite stream" semantics for both input and output.

5.1.1 *VrBuffer*: Low-Overhead Circular Buffers

The *VrBuffer* class implements a circular buffer with a single writer and multiple readers. To cut down overhead seen in the original SPECtRA system [Bose, 1999], Pspectra's buffers take advantage of the virtual memory system to allow modules to read and write arbitrarily large chunks of data in the buffer without checking for "buffer wrap." In a standard circular buffer (such as in SPECtRA), the data a procedure is reading or writing may be split into two pieces, as seen at the top of Figure 7. An algorithm reading or writing data in the buffer must "wrap" around to the beginning of the data once it passes the end of the buffer's memory region. This requires the addition of checks (or extra instructions for modular arithmetic) during data access and pointer manipulation.

VrBuffer maps the data buffer twice consecutively in virtual memory, as shown in the bottom half of Figure 7. This does not increase the amount of physical memory used, but requires the length of the buffer to be a multiple of the page size. Modules are given a pointer to data in the first copy of the buffer. The modules can then access an arbitrarily large (up to the size of the buffer) chunk of data without checking for "buffer wrap." The details of the actual buffer (such as its size and where it ends) are completely hidden from the module programmer. The module avoids all the overhead involved in checking for buffer wrap (or in using modular arithmetic to increment the pointer). *VrBuffer* translates the index (see Section 2.4) for a read or write request into a pointer in the first virtual memory space. This calculation requires just one AND and one ADD instruction for page-aligned buffers that have a size that is a power of two.

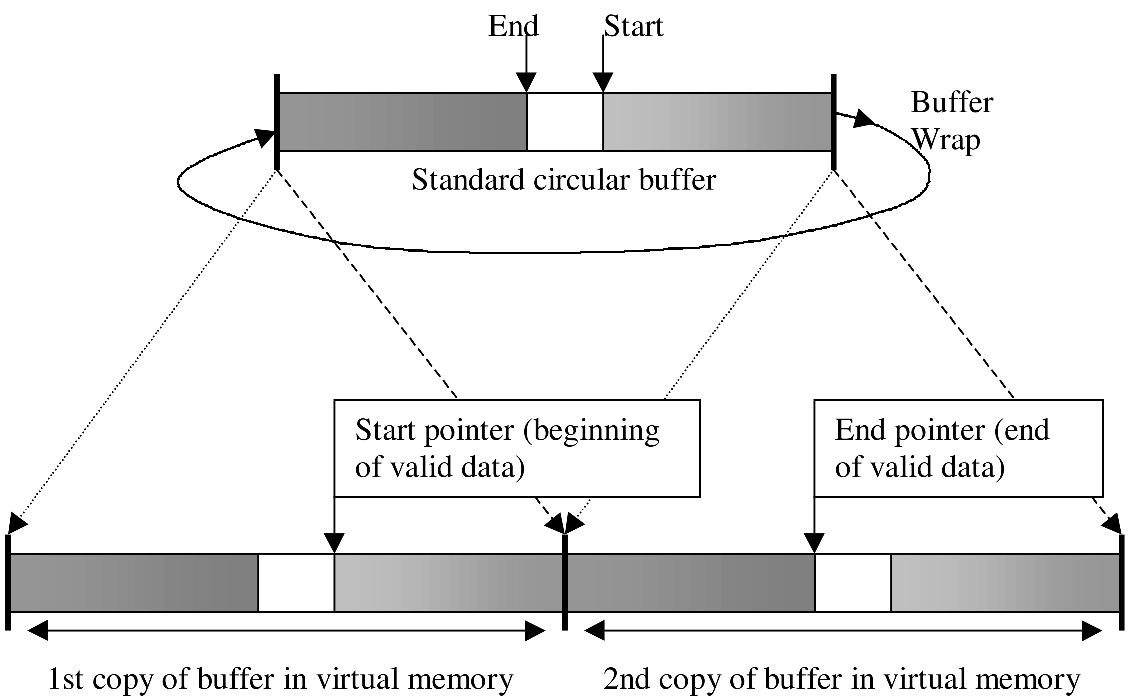

Figure 7 Use of virtual memory in *VrBuffer*

The size of the buffer is calculated during initialization of the module topology. The buffer for a particular module must be large enough for one block of data to be produced at all downstream sinks. The buffer sizes are calculated by the same algorithm that calculates optimal block sizes, as discussed in Chapter 6. When the system uses multiple threads, this size is conservatively adjusted by multiplying the required size for the single-threaded case by the number of threads that will be running. (In practice, the data required by the various threads usually overlaps.)

5.1.2 *VrGuppiBuffer*: Low-Overhead Input/Output

VrGuppiSource and *VrGuppiSink* replace the default output and input buffer, respectively, with a *VrGuppiBuffer* object. *VrGuppiBuffer* uses the GuPPI driver to map the GuPPI DMA buffers into the source's output buffer and the sink's input buffer. This allows data to be directly read from, or written to, the DMA buffers. The driver's DMA buffer is mapped twice consecutively in virtual memory, to allow the same operation as in *VrBuffer*.

5.1.3 *VrMUXBuffer*: Redirecting Read Pointers

The *VrMUX* module switches between multiple input modules. It achieves this without copying data from its selected input to *VrMUX*'s output buffer by using another subclass of *VrBuffer*, *VrMUXBuffer*. *VrMUXBuffer* does not allocate memory for a buffer, but instead replaces the index-to-pointer routine with one that retrieves a pointer into the selected upstream buffer. This pointer allows the downstream module to read data directly (and transparently) from the selected upstream buffer.

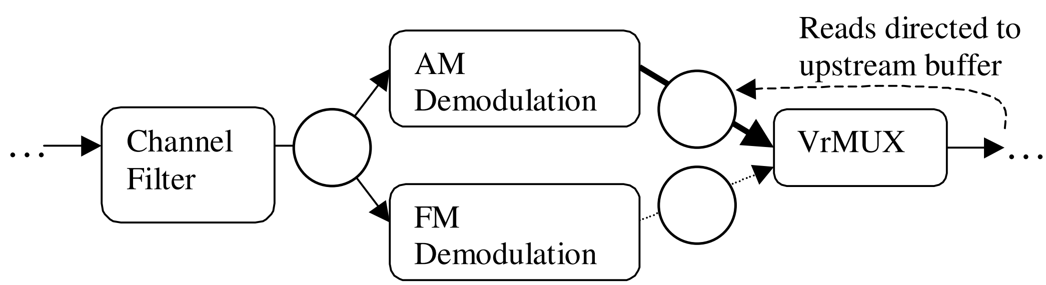

Figure 8 VrMUX passes data from one of its inputs to its output without actually copying any data. Circles represent buffers.

For example, Figure 8 depicts an audio receiver stream that allows *VrMUX* to switch between AM and FM demodulators without reconfiguring the topology. *VrMUX* allows the control program to quickly change the execution topology between a set of pre-determined topologies. In addition, *VrMUX* skips data on the unused inputs by returning zero size ranges from *forecast()* (see Section 3.2.2 above), allowing Pspectra to refrain from producing data in the unused portion of the topology.

The *VrSwitch* module also uses *VrMUXBuffer* to allow a portion of the topology to be turned off. The downstream module reads zeros when the switch is off. It is functionally equivalent to a *VrMUX* module with a *VrZeroSource* module (which outputs zeros) as its second input.

5.1.4 Multithreaded Functionality

Pspectra allows multiple threads to execute a particular module's *work()* procedure simultaneously. This means multiple threads may be writing different, non-overlapping, ranges of data in a buffer. Although there is only one module writing to a buffer, there may be many threads writing to the buffer at any one time. *VrBuffer* and its subclasses keep track of the ranges each thread is writing and what data has been completed. This allows Pspectra's scheduling algorithm (see Section 5.2.1) to determine what data can be computed downstream without blocking. *VrConnect* keeps track of threads that are reading data from a particular downstream module's input. *VrBuffer* uses its associated *VrConnect* objects to determine when data is no longer being used downstream. *VrBuffer* then reuses this space within the circular buffer.

VrConnect and *VrBuffer* use a simple, linked-list, implementation for tracking reading and writing threads. This could probably be improved, but is currently not a performance problem, because we use a small number of threads and the use and update of these lists is relatively infrequent in the operation of Pspectra.

Keeping track of data use in the infinite streams frees the module programmer from concerning himself with the tedium of determining when to reuse buffer space. Pspectra ensures that no module will ever "overflow" the buffer by overwriting data that a downstream module is still using. Eliminating these checks from the module greatly improves the efficiency of Pspectra's infinite streams. In addition, ensuring that concurrent writes under multithreaded operation are disjoint and do not overflow the buffers is a somewhat difficult job. Because Pspectra handles this, the programmer can rest assured that the dataflow implementation is correct in a parallel environment. The efficient implementation and simple semantics of Pspectra's "infinite streams" are one of the primary reasons for its success.

5.2 Scheduling

The scheduler implements a data-pull style of signal-processing, which provides good cache efficiency and can avoid the production of unnecessary intermediate data [Bose, 1999].

The Pspectra scheduling algorithm works in two phases. In the first phase, the scheduler searches the topology to find a series of modules whose input data is ready. The processor then marks this data to prevent any other processors from working on the same data. Then the second phase runs *work()* on the chosen modules to produce the marked data.

5.2.1 Data Marking

Pspectra implements a data-pull paradigm; scheduling starts with the sinks and proceeds upstream only by following data-dependencies. This lazy-evaluation style of signal-processing enables Pspectra to avoid computing intermediate data that is not necessary for the production of the final output data.

The scheduler attempts to schedule a consecutive string of modules to help keep data in the cache between the execution of different modules. The scheduler starts with a block of data (whose size is determined by the algorithm discussed in Section 6.3.3) at one of the sink modules. To determine if this block can be scheduled for computation, the scheduler recursively checks the block's data dependencies, i.e., the data ranges returned by calling *forecast()* on the module that produces the block. To schedule a block, the conditions listed below must be met.

- No more than one data dependency may be incomplete.
    - If one data dependency is not complete, then this data must also be scheduled for computation in this iteration (and thus must also meet these requirements).

- If more than one data dependency is incomplete, this block cannot be scheduled, but the scheduler may choose one of the data dependencies to mark (provided it meets these requirements).
- All other input data was completed in a previous iteration of the scheduling algorithm.

The input data ranges are determined by the return value of *forecast()*. The scheduler marks data in a consecutive string of modules (e.g., A → B → C in Figure 9). The first module in this string (A) is either a source, which has no inputs, or has inputs that are entirely complete. Subsequent modules (B and C) use data computed both in previous iterations and in this iteration. For example, module C may use data that B and D computed in previous iterations in addition to the data B will compute on this iteration. Scheduling a string of consecutive modules helps ensure that as much data as possible remains in the cache. In our example, most of the data module B reads will still be in the cache since module A has just written it.

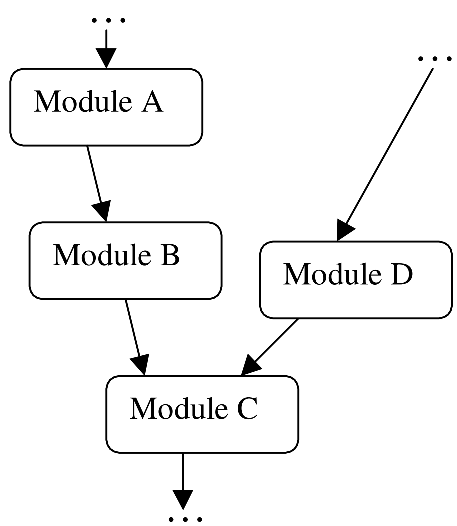

Figure 9 An example module topology

In the current implementation, only one thread may run the data-marking portion of the scheduler at a time. This helps ensure correctness and prevents the loss of performance that could be encountered by multiple threads duplicating work in this phase. This global lock is not a serious performance problem since the data-marking phase is short compared to the work phase and thus is a small fraction of the total running time. In practice, the threads tend to become staggered such that very little lock contention is seen in the scheduler.

5.2.2 Work

In the second phase of the scheduling algorithm, Pspectra calls *work()* on the appropriate modules (A, B, and C in our example above) to produce the marked data, re-marking the data as "completed" after each return from *work()*. This phase proceeds in the opposite direction as the marking phase, computing data first in the upstream modules (e.g., A) before proceeding downstream.

Pspectra efficiently executes signal-processing code. *Work()* is passed pointers to input and output data and need not include any code to check if input data is ready, if buffer space is available or will wrap, etc. It can simply run a tight signal-processing loop to produce output data from input data.

In modules that cannot always accurately predict their input requirements (see Section 3.2.1), *work()* may signal that not all of the output data was completed. In this case, Pspectra terminates the work phase prematurely and unmarks any downstream data that had not been computed (e.g., if B did not finish the requested data, C would not be run and its output data would be unmarked). The scheduling process then begins again in the data-marking phase to reassess what needs to be computed (e.g., B may need A to generate more data before it can compute anything).

6 Performance

First we will focus on three aspects of the overall performance of Pspectra: latency, overhead, and throughput. We examine how the size of the scheduled blocks affects these three measures of performance, with supporting measurements of Pspectra[3]. We discuss the tradeoff between these factors and which factors we chose to optimize for and why. Section 6.4 discusses how Pspectra automatically optimizes the performance for any signal-processing topology. Section 6.5 will focus on how Pspectra's performance scales on multiprocessor machines (its "speedup").

6.1 Latency Versus Overhead

Pspectra is a soft real-time system – although there are no hard bounds on the latency of any signal-processing topology in Pspectra, we would like the system to operate with the lowest latency possible (e.g., to improve interactive performance). To achieve low latencies, we would like as little data in the buffers as possible, thus the system to run on very small pieces of data and move information from sources to sinks as fast as possible.

Unfortunately, running on extremely small blocks of data dramatically increases the overhead both of scheduling the modules and within the modules themselves (from computation in the *work()* routine that is not in the inner loop and from the procedure call). Pspectra optimizes the block size for a given signal-processing topology to maximize useful processing power. If extra processing power is available after this optimization, the application could dynamically reduce block sizes to decrease latency.

[3] Version 0.7 of Erik Hendriks's kernel patch (hendriks@cesdis.gsfc.nasa.gov), which virtualizes the Intel Pentium Performance Monitoring Counters [Intel, 1999A], allows Pspectra to maintains per-process counts of the total cycles, cache misses, etc. spent in each module's work() procedure, as well as overall counts for the entire system.

6.2 Sources of Overhead

Figure 10 shows the percentage of processor time spent scheduling blocks (versus doing useful work) with increasing block size[4]. Overhead includes: the entire data-marking phase of the scheduler, all computation done in modules' *forecast()* procedures, all internal Pspectra code executed during the work phase of the scheduler, all synchronization operations, and cycles lost due to lock contention – everything not within the body of a *work()* procedure. The block sizes in Figure 10 represent the number of 8kHz audio output samples produced from an AMPS reception topology. Block sizes above 512 produce undesirably large latencies (over 1/16 of a second). This graph demonstrates that Pspectra can run with small latencies and still achieve scheduling overhead of 1-2 percent.

Further overhead not measured in Figure 10 results from the procedure call to *work()* and the setup a module does before and after its inner loop. This overhead is seen in the left-hand portion of Figure 11. This graph measures the average number of processor cycles required to perform a quadrature demodulation. The unoptimized C++ code for the quadrature demodulation consumes about 125 cycles per output. Pspectra's overhead at the module level becomes insignificant above a block size of about 40. This is roughly true for most signal-processing modules we examined.

6.3 Throughput

Looking again at Figure 11 above, it is apparent that the processing requirements for a module rise by about 20-25% for large block sizes (2560 and above). This results because upstream data is flushed from the cache before the downstream module has a chance to use it. Although in this example, these block sizes represent undesirably large

[4] The aberrations for the four-processor curve (at block sizes of 512 and 4096) are due to a 4MB limitation on shared memory segments (which are used for buffer memory). Due to large decimation factors in the application we measured, the upstream buffers must be more than 2000 times the block sizes at the sink. This forced block sizes above 256 to actually not be computed all at once, and exposed some nonlinear inefficiencies in the scheduling algorithm.

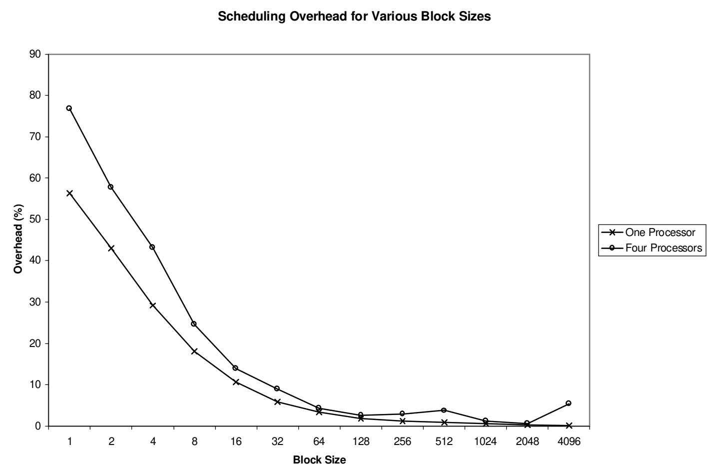

Figure 10 Graph of scheduling overhead for various block sizes

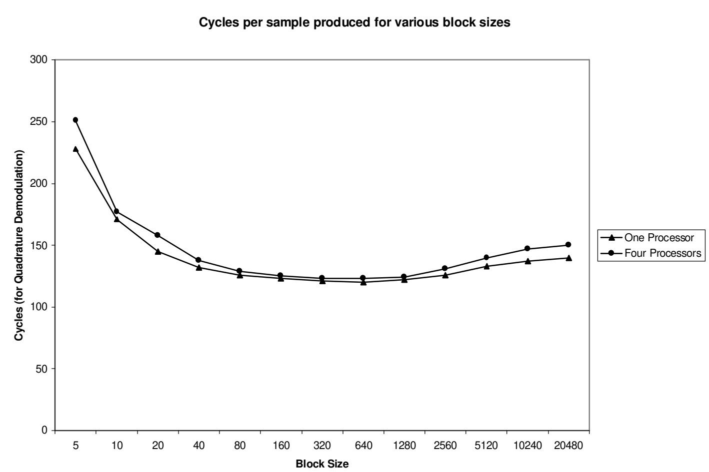

Figure 11 Graph of the number of cycles required to produce a sample for various block sizes.

latencies, this same phenomenon would be present demodulating higher data-rate channels (e.g., television) at much lower latencies and therefore merits attention. Block sizes must be kept small enough to allow the cache to hold all the data produced by the upstream module.

To achieve the maximum throughput possible for a given module topology, Pspectra should run with the largest block size that does not incur performance losses from cache misses. This section describes how Pspectra estimates this optimal block size. First, we describe the assumptions we made about the cache and the implications of the memory hierarchy on module design. We then describe the formula for computing the maximum efficient block size. Lastly, we compare the theoretical results with the experimental results depicted in Figure 11.

6.3.1 Assumptions

In calculating the theoretical effects of Pspectra signal-processing on the processor's cache, we make some assumptions about the behavior of the cache. First, the cache size available to the Pspectra process is assumed to be that of the entire physical data cache. This is plausible as signal-processing is quite data-intensive and will completely flush the cache of other data during a Pspectra process's timeslice. Since each Pspectra thread is intended to take up most of the processor time on the processor to which it is assigned, Pspectra's data will be remain in the cache even between context switches, i.e., we assume little interaction from other processes and the kernel.

We also assume that each module can create an output block from available input data without using a significant fraction of the cache for storage of intermediate data. If this assumption is not valid, Pspectra will incorrectly optimize cache performance.

The cache is assumed to be a perfect LRU cache, which is a reasonably accurate approximation of the Intel Pentium caches.

6.3.2 Implications of the Memory Hierarchy

To take better advantage of the general-purpose processor's memory hierarchy (which usually consists of multiple levels of caching and a slow but large main memory), the original SPECtRA system [Bose, 1999] implemented a lazy-evaluation, data-pull architecture. This allowed data to be generated only when it was needed and thus used immediately after being written. This greatly increased the probability of input data's presence in the cache for any particular module. SPECtRA, however, experienced high overhead because the scheduling algorithm was called in modules' inner loops and modules ran on very small blocks of data (only producing what was immediately necessary to compute the next downstream output sample). Lazy-evaluation and data-pull are also employed in Pspectra, although Pspectra attempts to determine the largest block of data that the entire signal-processing chain can compute without overflowing the cache and allows the modules to do the work without any buffer management or scheduling routines in their inner loops.

The programmer should be aware that increasing the number of modules in the system increases cache contention, since data must be stored in memory buffers between modules. Modules should be bundled together in a manner that takes advantage of compiler optimizations and faster register-only computations. Although Pspectra has minimal overhead for buffer management[5], the more data modules put into buffers, the less they can process before filling the cache. A good example of module optimization is the addition of a gain parameter into a filter module, since the gain can be incorporated into the filter taps with no additional processing requirements. This is a much better solution than a separate amplification module that multiplies every sample by a fixed gain, not only because incorporating it into the filter results in fewer multiply operations, but also because not using a separate module results in much less cache pollution. This allows the system to run on larger block sizes, and thus with less overhead. The goal is to

[5] The overhead of copying data from cached input to output buffers is only 4-10 cycles/sample compared to the useful work of 100-1500 cycles/sample for typical modules (on Intel PIII 500MHz).

minimize the amount of data that runs through the cache to produce a single output from the system.

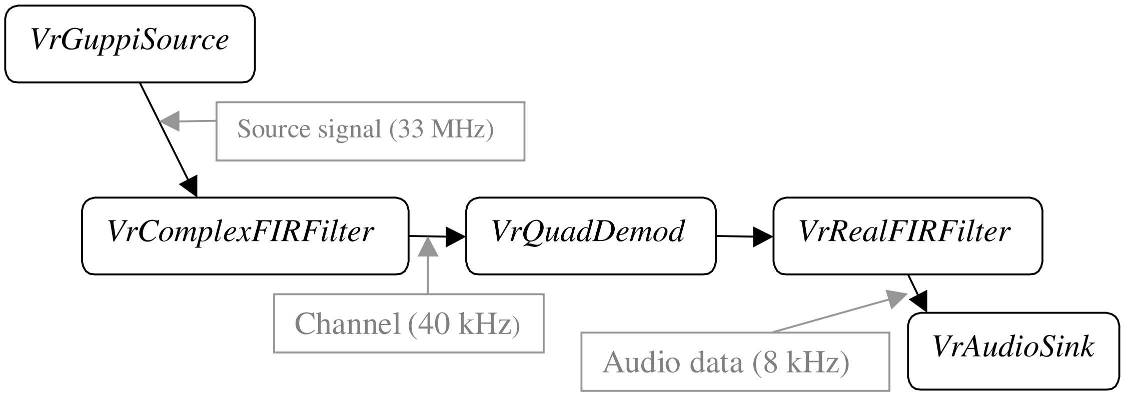

Figure 12 Topology for an AMPs cellular receiver

6.3.3 Maximum Optimal Block Size

To determine how much data can be computed before the cache is filled, we simply need to sum the amount of data each module (*m*) reads into the cache to produce N units of output at the sink (*s*), set this sum equal to the cache size (*C*), and solve for N:

$$C = \sum_{m} b_m \times r_m$$

where:

$r_m = m\text{->}mapSizeUp(\ r_{m-1})$ i.e., the amount of data read by module *m* during the production of *N* output units at the sink. (*mapSizeUp()* is described in Section 3.1.1.2)

$b_m =$ the size of a unit of module *m*'s input data, in bytes.

As an example application, we shall consider an FM receiver for a 30kHz AMPS channel using a 400 tap FIR filter (see Figure 11). The amount of data read to produce N output units (of 8kHz 16-bit audio):

Module	Units read	Size of input unit in bytes	Bytes read
VrAudioSink	N	2 (short)	2N
VrRealFIRFilter (20 taps, decimation of 5)	*mapSizeUp(N)=* 19+5N	4 (float)	76+20N
VrQuadDemod (history of 2)	*mapSizeUp(19+5N)=* 20+5N	8 (complex)	160+40N
VrComplexFIRfilter (400 taps, decimation of 825)	*mapSizeUp(20+5N)=* 400*(20+5N)	1 (char)	8000+2000N
TOTAL			8236+2062N

Running on a system with a 512 Kbyte cache, we should be able to produce (512*1024-8236)/2062 ≈ 250 units of output from this topology before we encounter loss of performance from cache misses.

6.3.4 Comparison with Experimental Results

In the previous section we argued that for a particular FM receiver, Pspectra can produce 250 units without experiencing degradation in performance due to cache misses. The 250 units of data at *VrAudioSink* corresponds to 5*250=1250 units produced by the quadrature demodulator. Examining Figure 10 again, we see that the performance of the quadrature demodulator is indeed negatively impacted as block size increases from 1280 to 2560. However, block size is not as important to the performance of individual modules as it is to the performance of the scheduling algorithm. Figure 10 demonstrates that the performance only degrades by 20% after the block size is 8 times optimal. The "sweet spot" for optimal performance in the modules is quite large – over an order of magnitude of change in block size (from 80 to 1280). Drastic performance degradation only occurs when blocks are very small or too large to fit everything in the cache.

6.4 Optimizing the Performance

Based on the analysis above, Pspectra automatically attempts to optimize the performance of the signal-processing topology. A static computation based on the cache size (which can be easily queried on a PIII Linux system, see [Intel, 1999A] for details on this feature) determines the block size during the setup procedure described in Section 3.1.1.

Pspectra arrives at the block size through the method illustrated in Section 6.3.3. This block size insures cache-friendly computation and a low scheduling overhead at the expense of latency. This block size may be tweaked statically by the programmer if Pspectra's optimization algorithm fails to achieve optimal performance (e.g., if the assumptions in Section 6.3.1 do not hold). The program itself may dynamically change the block size to reduce latency (if more processing power is available) or to reduce scheduling overhead (for very data-intensive programs in which the block size calculated above is too small to actually achieve optimal performance).

Pspectra modules that use a small proportion of their input data may override the function *averageInputUse()*. *AverageInputUse()* returns a conservative estimate of what fraction of cache lines in its inputs are actually used (a number between 0 and 1). *AverageInputUse()* defaults to returning 1 and can be overridden to help Pspectra choose a larger block size and thus slightly improve performance for modules that skip significant portions of their input. For example, a particular decimating filter may only read a quarter of its input samples, skipping large portions of data, in order to produce all of its output samples and will thus return .25.

6.5 Multiprocessor Performance

As shown in Figure 13, Pspectra achieves a near linear speedup for some typical signal-processing applications running on up to four processors. Our data parallel approach allows us to achieve this speedup even for applications such as the AMPS receiver, where over two-thirds of the processor cycles are spent in a single module (the channel filter).

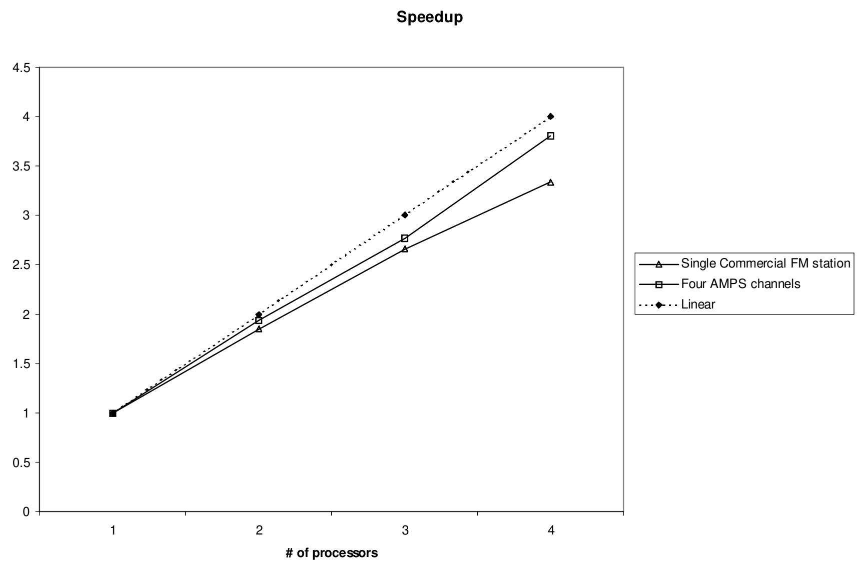

Figure 12 Mutliprocessor Performance (Speedup)

We believe Pspectra will scale just as well on machines with more than four processors; our data parallel approach scales to an arbitrary number of processors. Our experience indicates that most applications can take advantage of the data-parallel approach, even if some modules within the application must be run serially (e.g., for synchronization).

In addition to the applications shown in Figure 8, which have a small number of modules, we tested Pspectra's performance for applications with large numbers of modules. Figure 14 illustrates how Pspectra scales with the number of channels in a multi-channel AMPS receiver. Such a receiver would be a key component in a software cellular base station.

Note that processor usage scales close to linearly with the number of channels, both on one and four processors. Note also that receiving two channels on a single-processor machine is only 1.5 times as processor-intensive as receiving one channel! This is because in generating the second channel and subsequent channels, the channel filter reuses the same inputs, which are already in the cache. Finally, note that the processor utilization in the four-processor case is roughly one-fourth of the single-processor utilization (e.g., six channels is about 80% of one processor, and 20% of a four-processor machine), indicating Pspectra achieves linear speedups for topologies consisting of a large number of modules.

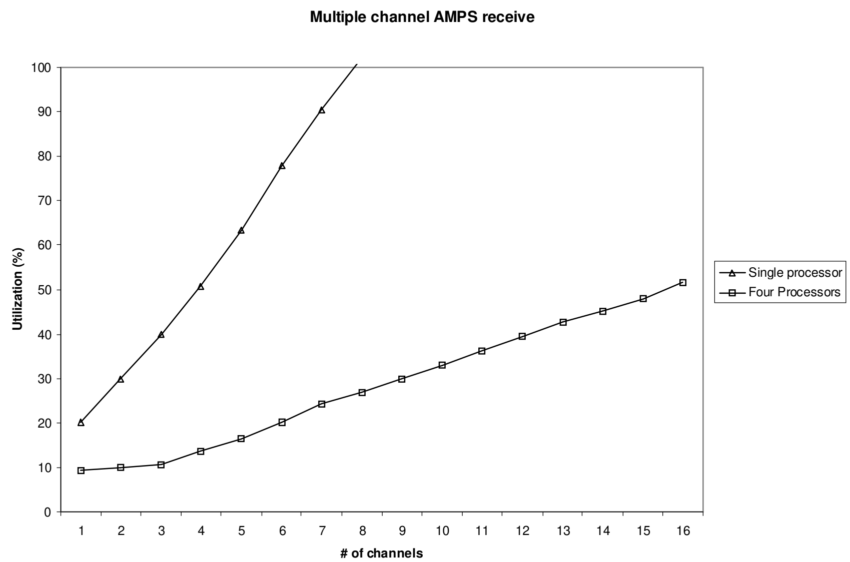

Figure 12 PSPECtRA's performance on systems with large numbers of modules

We tested four processors up to full utilization, at 32 channels, which uses 97 distinct modules. This result indicates that current SMP technology easily has enough horsepower to build small wireless base stations. As processors become ever more powerful, the possible applications of Pspectra will multiply.

7 Conclusions and Future Work

Pspectra is an innovative new environment for signal-processing work. Pspectra opens doors both for further research into parallel signal-processing and for the implementation and testing of new algorithms and applications, using Pspectra as a development platform.

7.1 Future Work

Future work should explore the possible useful applications of the Pspectra environment, and the particular advantages a general-purpose processing platform has over DSPs and other special-purpose hardware. We describe some improvements to Pspectra that could further boost the performance and functionality for signal-processing applications.

7.1.1 Distributed Signal-Processing

Pspectra is currently implemented only for SMP machines. The system could be extended to distributed-memory machines or clusters of machines simply by adding message-passing functionality in the buffers. The scheduler and the modules would require very minor modifications. A particularly interesting system would involve the use of the GuPPI card itself to transfer the data between machines in a cluster.

Current high-end SMP machines can adequately address most of the applications the SpectrumWare group has discussed working on. It is not clear that a distributed version of Pspectra is necessary except for very demanding special-purpose applications, such as radar processing.

7.1.2 Signal-Processing Compiler

Pspectra's modular design makes development easier, and allows a generic scheduler to run any combination of modules. Unfortunately, the system also suffers a performance hit compared to a monolithic signal-processing program. The C++ compiler cannot make optimizations across module boundaries. A special compiler (or pre-processor of some sort) that could combine modules' *work()* procedures in some manner would help the compiler keep intermediate data in registers[6] instead of writing and re-reading it from the cache between modules.

It is not immediately clear how to both combine the modules and still use a dynamic scheduling algorithm. One possible approach would be to compile a monolithic version for normal, predictable operation and also a standard modular version to fall back on when necessary (for example, to re-establish synchronization when synchronization is lost in a particular application). It would be difficult to build a general solution, and this problem warrants further investigation.

7.1.3 A Possible Commercial Application: Flexible Wireless Base Station

One of the more intriguing and realizable applications of the Pspectra technology is an extremely flexible wireless base station. A base station built on a general-purpose SMP machine, or cluster of such machines, could take advantage of the flexibility of software to both adapt to its users and to provide an easy upgrade path for wireless providers as they move to third-generation (3G) wireless technology. For example, such a base station could dynamically allocate different portions of its frequency band to various standards (e.g., GSM, CDMA, analog cellular, 3G services, wireless data services).

A software base station would intelligently partition its allocated spectrum based on the client/subscriber/handheld usage. Pspectra's more flexible architecture would simplify the implementation of dynamic allocation of arbitrarily higher bandwidth channels for data transfers. A software radio based architecture would also allow base stations to be upgraded as new algorithms and standards become available. This would reduce the time and money spent replacing existing hardware with updated technology, more efficiently utilize spectrum allocations, and ease transition for users of outdated technology by continuing to support older technology.

---

[6] This would be more helpful on an architecture that has many general-purpose registers, such as the Alpha, than on the Intel x86 architecture.

Virtual radio technology could also be incorporated into handsets to allow them to upgrade to meet new specifications or to change their specifications as mobile users move between countries with different wireless infrastructures. Given that low-power is a top priority in handhelds, however, it may take some time before general-purpose processor are competitive with DSPs and ASICs in handheld devices.

7.2 Conclusion

Commodity workstations make an excellent platform for signal-processing development, testing, and prototyping. As processors become more powerful, and software radios more popular, general-purpose platforms will increasingly replace special-purpose signal-processing hardware and DSPs in many applications. The signal-processing methodology used in Pspectra provides an efficient and easy-to-use platform for the development of parallel signal-processing algorithms and applications.

Pspectra demonstrates the effectiveness of partitioning the system into four components: the signal-processing modules, the data management mechanisms, the scheduler, and the control program. This partitioning simplifies the programming interface and supports a high-performance model for parallel signal-processing.

Our system allows the algorithm designer to ignore data management issues and focus only on the construction of efficient signal-processing algorithms. Applications in Pspectra are easy to build by combining signal-processing modules.

Pspectra has a simple, yet effective scheduler. This scheduler, based on data-dependency information received from the modules, efficiently schedules computation on multiple processors. Complex static analysis is not necessary; the programmer can rely on generic scheduling and optimization algorithms for efficient execution. The separation of data management and scheduling from the modules enables Pspectra to efficiently run modules in parallel, achieving near-linear speedups.

The Pspectra environment demonstrates that general-purpose workstations are useful for developing and prototyping real-time signal-processing systems. As general-purpose processors and architectures move into more devices, new and interesting applications of this work will appear. If mainstream signal-processing development can make the jump to general-purpose platforms, the newfound ease of development and testing will accelerate further advances in signal-processing algorithms and technologies.

8 Bibliography

[Ankcorn, 1998] John Ankcorn. *SpectrumWare NTSC*. Advanced Undergraduate Project, MIT, 1998.

[Bose, 1999] Vanu G. Bose. *Design and Implementation of Software Radios Using a General Purpose Processor*. PhD Thesis, MIT, 1999.

[Bose *et. al.*, 1998A] Vanu Bose, Alok Shah, Michael Ismert. *Software Radios for Wireless Networking*. *Infocomm '98*.

[Bose *et. al.*, 1998B] Vanu Bose, Michael Ismert, Matthew Welborn, J. Guttag, "Virtual Radios," *IEEE JSAC* issue on Software Radios, April, 1999

[Buck *et. al.*, 1991] Joseph Buck, Soonhoi Ha, Edward A. Lee, David G. Messerchmitt. "Ptolomey: A Platform for Heterogeneous Simulation and Prototyping," *European Simulation Conference*, Copenhagen, Denmark, 1991.

[Chen, 1997] William Chen. *Real-Time Signal Processing on the Ultrasparc*. Master's Thesis, UC Berkeley, 1997.

[Chiu, 1999] Andrew G. Chiu. *Adaptive Channels for Wireless Networks*. Master's Thesis, MIT, 1999.

[Cook and Bonser, 1999] Peter G. Cook and Wayne Bonser. *Architectural Overview of the SPEAKeasy System*. IEEE Journal on Selected Areas in Communications. Vol. 17, No. 4, April 1999.

[DeLuca *et. al.*, 1997] Cecelia DeLuca, Curtis W. Heisey, Robert A. Bond, Jim M. Daly. "A Portable Object-Based Parallel Library and Layered Framework for Real-Time Radar Signal Processing," presented at *1st International Conference on Scientific Computing in Object-Oriented Environments (ISCOPE)*, 1997.

[Fisher and Dietz, 1998] Randall J. Fisher, Henry G. Dietz. "Compiling for SIMD Within A Register," presented at *11th annual Workshop on Languages and Compilers for Parallel Computing (LCPC98)*, 1998.

[Geary, 1990] MJ. Eccles, R. Geary, "Parallel signal processing – a modular approach," presented at *Digital Signal Processing: Application Opportunities*, London, UK, October 1990.

[Hwang and Xu, 1996] Kai Hwang and Zhiwei Xu. Scalable Parallel Computers for Real-Time Signal-Processing. *IEEE Signal-Processing Magazine Volume: 134*, July 1996, pp. 50-66.

[Intel, 1999A] Intel. *Intel Architecture Software Developer's Manual*, Volumes 1-3, 1999.

[Intel, 1999B] Intel. *Intel Architecture Optimization Reference Manual*, 1999.

[Ismert, 1998] Michael Ismert. "Making Commodity PCs Fit for Signal Processing." In *USENIX*, June 1998.

[Lackley and Upmal, 1995] *Speakeasy: The Military Software Radio.* IEEE Communications Magazine, May 1995.

[Lang, *et. al.,* 1988] Gorgon R. Lang, Moez Dharssi, Fred M. Longstaff, Philip S. Longstaff, Peter A.S. Metford, and Malcolm T. Rimmer. *An Optimum Parallel Architecture for High-Speed Real-Time Digital Signal Processing*. IEEE Computer, Vol. 21, Issue 2, 1988.

[Ledeczi and Abbott, 1994] Akos Ledeczi and Ben Abbott. "Parallel Architectures with Flexible Topology," *Proceedings of the Scalable High-Performance Computing Conference*, 1994, pp. 271-276.

[Mercury, 1994] *Parallel Application System*. http://www.mc.com/Data_sheets/dspas.pdf

[Nalecz, *et. al.,* 1997] M. Nalecz, K Kulpa, A Piatek, and G. Wojdolowicz. "Scalable Hardware and Software Architecture for Radar Signal Processing System," *IEEE Radar Conference*, 1997.

[Pino *et. al.*, 1995] José Luis Pino, Soonhoi Ha, Edward A. Lee. Joseph T. Buck. "Software Synthesis for DSP Using Ptolemy," *Journal of VLSI Signal Processing*, 1995.

[Pino and Lee, 1995] José Luis Pino, Edward A. Lee. "Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors," presented at *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Detroit, MI, May 1995.

[Thoeni, 1994] Urban A. Thoeni. *Programming Real-Time Multicomputers for Signal Processing*. Prentice Hall, 1994.

[Tuttlebee, 1997] Dr. Walter Tuttlebee. *The Impact of Software Radio*. CEC Software Radio Workshop, Brussels, May 1997.

[Watlington and Bove, 1995] John A. Watlington and V. Michael Bove, Jr. *Stream-Based Computing and Future Television*. Proceedings of the 137th SMPTE Technical Conference, September 1995.

[Welborn, 1999A] Matt Welborn *Direct Waveform Synthesis for Software Radios*, to appear in Proceedings of WCNC'99.

[Welborn, 1999B] Matt Welborn. *Narrowband Channel Extraction for Wideband Receivers*, Proceedings of ICASSP'99.

[Welborn and Ankcorn, 1999] Matt Welborn and John Ankcorn. *Waveform Synthesis for Transmission of Complex Waveforms*, to appear in Proceedings of RAWCOM'99.