# Design and Implementation of Software Radios Using a General Purpose Processor

by

Vanu G. Bose

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

June 1999

© Massachusetts Institute of Technology 1999. All rights reserved.

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
April 9th, 1999

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
John V. Guttag
Professor of Computer Science and Engineering
Thesis Supervisor

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
David L. Tennenhouse
Senior Research Scientist, Computer Science
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

**Design and Implementation of Software Radios Using a General Purpose Processor**

by

Vanu G. Bose

Submitted to the Department of Electrical Engineering and Computer Science
on April 9th, 1999, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

## Abstract

This dissertation presents the design, implementation and evaluation of a novel software radio architecture based on wideband digitization, a general purpose processor and application level software. The system is designed to overcome the many challenges and exploit the advantages of performing real-time signal processing in a general purpose environment. The main challenge was overcoming the uncertainty in execution times and resource availability. The main advantages are faster clock speeds, large amounts of memory and better development environments. In addition it is possible to optimize the signal processing in conjunction with the application program, since they are running on the same platform.

The system has been used to implement a *virtual radio*, a wireless communication system in which all of the signal processing from the air interface through the application is performed in software. The only functions performed by dedicated hardware are the down conversion and digitization of a wide band of the RF spectrum. The flexibility enabled by this system provides the means for overcoming many limitations of existing communication systems. Taking a systems design approach, the virtual radio exploits the flexibility of software signal processing coupled with wideband digitization to realize a system in which any aspect of the signal processing can be dynamically modified. The work covers several areas, including: the design of an I/O system for digitizing wideband signals as well as transporting the sample stream in and out of application memory; the design of a programming environment supporting real-time signal processing applications in a general purpose environment; a performance evaluation of software radio applications on a general purpose processor; and the design of applications and algorithms suited for a software implementation. Several radio applications including an AMPS cellular receiver and a network link employing frequency hopping with FSK modulation have been implemented and measured.

This work demonstrates that it is both useful and feasible to implement real-time signal processing systems on a general purpose platform entirely in software. The virtual radio platform allows new approaches to both system and algorithm design that result in greater flexibility, better technology tracking and improved average performance.

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

Thesis Supervisor: David L. Tennenhouse
Title: Senior Research Scientist, Computer Science

**Design and Implementation of Software Radios Using a General Purpose Processor**

by

Vanu G. Bose

Submitted to the Department of Electrical Engineering and Computer Science on April 9th, 1999, in partial fulfillment of the requirements for the degree of Doctor of Philosophy


## Abstract

This dissertation presents the design, implementation and evaluation of a novel software radio architecture based on wideband digitization, a general purpose processor and application level software. The system is designed to overcome the many challenges and exploit the advantages of performing real-time signal processing in a general purpose environment. The main challenge was overcoming the uncertainty in execution times and resource availability. The main advantages are faster clock speeds, large amounts of memory and better development environments. In addition it is possible to optimize the signal processing in conjunction with the application program, since they are running on the same platform.

The system has been used to implement a *virtual radio*, a wireless communication system in which all of the signal processing from the air interface through the application is performed in software. The only functions performed by dedicated hardware are the down conversion and digitization of a wide band of the RF spectrum. The flexibility enabled by this system provides the means for overcoming many limitations of existing communication systems. Taking a systems design approach, the virtual radio exploits the flexibility of software signal processing coupled with wideband digitization to realize a system in which any aspect of the signal processing can be dynamically modified. The work covers several areas, including: the design of an I/O system for digitizing wideband signals as well as transporting the sample stream in and out of application memory; the design of a programming environment supporting real-time signal processing applications in a general purpose environment; a performance evaluation of software radio applications on a general purpose processor; and the design of applications and algorithms suited for a software implementation. Several radio applications including an AMPS cellular receiver and a network link employing frequency hopping with FSK modulation have been implemented and measured.

This work demonstrates that it is both useful and feasible to implement real-time signal processing systems on a general purpose platform entirely in software. The virtual radio platform allows new approaches to both system and algorithm design that result in greater flexibility, better technology tracking and improved average performance.


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

Thesis Supervisor: David L. Tennenhouse
Title: Senior Research Scientist, Computer Science

## Acknowledgments

I would first like to thank my supervisors John Guttag and David Tennenhouse, who convinced me that computer science was what I was really interested in, gave me the freedom to pursue a research topic that crossed several different domains, and forced me to finally bring the work to closure. Their friendship, guidance and support have been invaluable.

I would also like to thank my readers. Randy Katz for his time and guidance. Alan Oppenheim, for bringing a needed perspective and context to the work. I have greatly enjoyed our many conversations and the friendships that we have developed.

David Wetherall, my friend, and late night companion in the lab who made the whole experience much more enjoyable, whether it was thesis writing, a new research hack or playing Quake. Were it not for Dave's company, I'm quite sure I would have graduated years ago.

Bill Stasior, my friend and former roommate, for the discussions about the shortcomings of existing media processing systems led to some of the design innovations in the programming environment.

Members of the SpectrumWare team, Mike Ismert, Andrew Chiu, Brett Vasconcellos, Matt Welborn, Jon Santos, Alok Shah, Janet Wu and John Ankorn, here at MIT made great contributions to the project and made the work fun. I would especially like to thank Mike Ismert for his tremendous contributions to the project and his unique ability to keep us from taking ourselves too seriously. Andrew Chiu and Brett Vasconcellos have been on the team since they were freshman and have made incredible contributions to the work, provided many an interesting discussion, and answered more of my questions than I have of theirs.

Finally I would like to thank my mother and father, for providing me with this opportunity and supporting me for all these years.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This dissertation presents the design, implementation and evaluation of a novel software radio architecture based on wideband digitization, a general purpose processor and application level software. The goal of a software radio is to create a communications system in which any aspect of the signal processing can be dynamically modified to adapt to changing environmental conditions, traffic constraints, user requirements and infrastructure limitations. The coupling of wideband digitization with application level software running on a general purpose processor allows for the modification of a greater range of functionality than any existing solution.

Our approach to software radio design enables the construction of wireless communications systems that overcome many of the limitations of existing systems that result from their lack of flexibility. The next section describes how this lack of flexibility hampers existing systems. This is followed by a discussion of the motivation behind using a general purpose processor and some of the advantages that result from this choice. The introduction concludes with a brief description of the work and a listing of the primary contributions of this dissertation.

## 1.1 The Need for Flexible Wireless Systems

Many of the limitations of today's wireless communications systems stem from a lack of flexibility, leading to:

- Poor use of the available spectrum,
- Inability to incrementally add functionality,
- Slow adoption of new standards, and
- Proliferation of devices.

Many of these problems are manifest in the current cellular telephone network. The Advanced Mobile Phone System (AMPS) [DiPiazza *et al.*, 1979] is still the dominant system in the U.S. The system was standardized in 1978 and allocated 30 kHz for each voice channel. Recently, new digital standards (CDMA, GSM, DAMPS) have emerged, but their deployment has been slow because it requires new infrastructure to be deployed and new handsets to be distributed, which is both costly and time consuming. Many customers actually carry several phones to get adequate coverage. Some manufacturer's are producing dual- and tri-mode phones, but this is only a short term solution since they only interoperate with a few of the existing standards and won't interoperate with the next standard to be deployed. Furthermore, the lack of flexibility may significantly delay, or prevent, deployment of the next standard. Software programmability, coupled with wideband digitization in both the handset and basestation would allow for system upgrades without replacing any hardware and result in closer tracking of technology advances.

Wireless networking is another area in which static device functionality has led to significant problems. The recent rapid growth in wireless network technology has greatly expanded the capabilities of mobile computing devices. The wide variety of wireless networks, including satellite, cellular, and local area systems provide a wide range of overlapping coverage areas and services [Katz and Brewer, 1996]. However, the multitude of standards hinders seamless interoperability by requiring different physical devices to interoperate with different networks. Not only do different wireless LANs operate in different RF bands, but even those using the same band employ different coding, modulation, and network protocols. The implementation of wireless network interface cards in dedicated hardware limits the flexibility of these devices. By moving all of the functionality into software, the approach described in this thesis provides a solution to this problem by enabling one physical device to interoperate with any of the standards.

Another drawback of existing wireless systems is their limited, or complete lack of, ability to incrementally upgrade functionality. Encryption is an excellent example of the utility of being able to incrementally upgrade functionality. AMPS phones have no encryption, and encryption cannot be added to the system since the functionality is static. An even more compelling example is the encryption present in the GSM cellular system. The encryption scheme has been broken, but there is no way to change the encryption scheme, thus all GSM systems must be considered insecure. However, if the GSM system were re-programmable, a new encryption scheme could be downloaded to each phone.

Incremental changes can occur dynamically as well. A radio network could dynamically modify its coding, modulation or multiple access scheme based on traffic or environmental conditions. Existing physical layer technologies, with their static functionality, are designed to meet worst case channel and traffic conditions. In a wireless network, channel conditions can vary rapidly over several orders of magnitude and tend to be difficult to predict, especially when nodes are mobile [Nguyen *et al.*, 1996]. This is problematic because it leads to designs that are based on the worst case that can be tolerated, rather than the channel conditions that are currently being encountered. The result is inefficient use of the available spectrum.

In addition to varying channel conditions, variability in network traffic loads also leads to inefficient use of the spectrum. For example, most cellular systems are subdivided into statically allocated channels, designed to support a specific number of users during the worst case traffic load. A user is limited to the bandwidth of a single channel even if there are other unused channels available. Another limitation of static design is that the system cannot be tuned for different applications. For example, different applications such as real-time full duplex audio and file transfers can tolerate different tradeoffs between latency, jitter and error rate, but the coding, modulation and channel assignments cannot be optimized for the specific application.

The concept of a *software radio* has recently emerged as a potential solution to these problems [Mitola, 1995]. This thesis investigates an approach to building software radios that differs significantly from existing work in that it digitizes a wideband of the spectrum and performs all of the processing in software on a general purpose processor. In the context of this thesis, wideband refers to the entire band utilized by a particular communications system. For example, the U.S. cellular band allocates 12 MHz of spectrum for each of the uplink and downlink bands, thus 12 MHz is the bandwidth that would be digitized. The design decision to digitize a wideband is critical to the flexibility of the system. By acquiring the entire band of interest all of the processing, including the multiple access technique, can be re-programmed. This allows the software radio to be re-programmed to interoperate with any system in the band. For example, an AMPs cellular system could be software upgraded to a third generation system utilizing wideband CDMA. Digitizing the entire band of interest maximizes the flexibility of the software radio in that band. The motivation behind the choice of a general purpose platform is explained in the following section, and section 1.7 provides a review of current software radio research.

## 1.2 Choice of a General Purpose Platform

Only a few years ago, special purpose hardware was needed to perform real-time audio processing on a desktop PC. Now, however, applications such as audio mixers and video players are run as applications under full-featured operating systems (e.g. Windows95 and Linux) with no explicit real-time support. Although the application requirements have remained the same, the advances in technology, especially in processor speed, have enabled these applications to run in real-time even though they appear as "just another process" to the operating system. As processor technology continues to improve, the scope of *real-time* applications that can be implemented on general purpose platforms expands. One of the main motivations behind the choice of a general purpose platform was the desire to understand the scope of real-time applications that can be implemented on today's systems, and how this scope changes as the underlying technology improves.

The choice of a general purpose processor was further motivated by observation of the Speakeasy military software radio research project [Lackey and Upmal, 1995]. The goal of this project was to implement a system that could be re-programmed to interoperate with several different radio communication systems. However, the use of specialized configurations of DSP processors resulted in a system that could not track processor technology advances because the software could not be easily ported. If the Speakeasy software were easily ported to new processors, it could have been used to implement more sophisticated processing or simultaneously process multiple channels. In addition, the choice of hardware platform led to considerable software complexity which affected the utility of the system in several ways, including low code re-use and increased development time. General purpose processors with their better development environments, emphasis on portability and faster clock rates, provide a solution to these problems and de-couple the hardware from the software, allowing the system to track the rapid advances in processor technology.

One of the most significant advantages of a software implementation is flexibility, in particular the ability to dynamically adapt to changing conditions, resources and requirements. The flexibility encompasses not only adaptive algorithms where parameters can be varied according to changing constraints, but also applications where the functionality of the signal processing system can be dynamically altered. The set of events that the system can adapt can be grouped into three broad categories: User/Application requirements, resource availability and external constraints.

The potential performance improvements do not lie in the traditional metrics for communications devices, such as bit error rate or SNR, but in the performance from the applications point of view. The performance that ultimately matters is what is observed by the end user, and the metrics used are application dependent. The quality of a video stream may be measured by parameters such as bits per pixel and frame rate, while the performance of a network application might be quantified by the number of packets received per second. These constraints have different effects on the underlying parameters in the system. The use of a general purpose processor has the distinct advantage that applications and the underlying communications system can be tightly integrated, since they are running on the same processor.

### 1.2.1 Advantages

One of the most interesting attributes of a software implementation is that it maximizes the utility of the existing hardware. A software implementation has the distinct advantage of being able to apply work saved in one function to any other function or application. This is because all of the functions utilize the same engine for computation. In a hardware system, digital or analog, each component has different processing capabilities, and work saved in one function cannot be arbitrarily applied to another. For example, the computational power of a hardware MPEG decoder cannot be utilized to perform speech recognition when MPEG is not needed.

There are several other advantages which stem from the resources available on a general purpose system. The large amount of memory facilitates the implementation of algorithms that make multiple passes on the data, perform out of order computation, and store waveforms for future use. The memory also allows for the temporal de-coupling [Tennenhouse and Bose, 1996] of the data. With the waveform stored in memory, the timing can be determined by scanning the waveform and determining the appropriate timing information, such as the time at which the transmission commenced. This can greatly simplify the implementation of some systems, and reduce the amount of work required to maintain synchronization.

In the context of this thesis, general purpose processors are defined as those that are optimized to improve average performance over a wide range of applications. Exploiting the resources available in this environment offers many potential advantages, including:

- **Improved resource utilization.** Since all of the processing functions, as well as the application, are implemented on the same platform, any cycles saved in one function, can be applied to any other. This advantage is only fully realized when all of the processing, including the application (e.g. web browser, or video player) shares one common platform.
- **End-to-End optimization.** It is often the case that functionality provided by a radio device is only a small part of a larger application, but the radio is often treated

as a black box by the application. Running all of the functions on the same platform allows the entire system to be dynamically optimized.

- **Ease of Implementation.** There is a large gap between the capabilities and programming environments available on general purpose workstations and those available on specialized DSP hardware. This leads to reduced development time, better code re-use and portability, and a much smaller source code base.
- **Moore's Law.** The doubling in general purpose processor speed every eighteen months is largely driven by market factors, but not to be ignored. Designing for these systems allows the software to ride the underlying technology curve and reap the benefits of these performance improvements.

There are several obstacles that have limited the implementation of data intensive real-time signal processing systems on a general purpose platforms. One of the primary obstacles is the uncertainty in the availability of system resources and the existence of structures such as caches that can cause certain DSP algorithms to perform poorly [Bier *et al.*, 1997].

The uncertainty in execution times is addressed by designing applications and algorithms with this uncertainty in mind. This approach opens an opportunity to design algorithms that have variable computational requirements themselves.

If algorithms are designed to perform a non-constant amount of work, the cycles saved when the work required is low can be applied to other functions or applications. If an algorithm is designed to perform a constant amount of work, then the system component for that function can be designed with just enough computational power to meet the requirements. However, a software algorithm can be designed to perform a variable amount of work. For example, an algorithm might utilize a simple detection algorithm when the SNR is high, but switch to a more computationally complex algorithm as the SNR degrades.

Designing an algorithm that does a non-constant amount of work is only useful if the extra cycles that are available during periods of low computation can be either re-used by other functions and applications, utilized to generate a more precise result, or result in power savings. A general purpose processor has the ability to apply extra cycles where needed, even to other applications that are unrelated to the communications system. Recent advances in dynamic clock management, such as implemented in the Intel strongARM processor provide the ability to vary the clock speed dynamically, thereby saving power during periods of reduced computation. While other system architectures may provide some of this ability, for example a DSP could be programmed to utilize incremental refinement algorithms to use available cycles, the space in which the cycles can be applied is limited. The DSP could not utilize its extra cycles to improve the performance of other applications, such as a web browser or the user interface.

## 1.3 Scope of Thesis

The goal of this thesis work was to design and build a wireless communications system with enough flexibility to enable dynamic tradeoffs in the processing performed by the physical layer. In order to achieve this goal, most of the physical layer functions were implemented in software, and a general purpose processor was chosen since it provides the most flexible processing platform. The work provided an understanding of the capabilities and limitations of current technology and indicated future directions for research in both software radios and the enabling technologies.

The focus of this thesis is on software radio system design, software engineering and the development of algorithms for the efficient implementation of signal processing algorithms on a general purpose platform. This thesis does not focus on the issues involved with signal acquisition or the design of processing technology, and uses commercially available technology for these portions of the system.

Low power considerations were sacrificed in order to investigate the limits of the flexibility of the existing technology. However, many of the techniques and algorithms developed during the course of the work can be used to reduce the power requirements in hand-held applications. These developments, combined with recent work in low power processor technology and dynamic clock and power management may make the general purpose approach more feasible in a low power environment.

The software radio system presented in this dissertation provides and excellent development and evaluation platform for software radio applications. This system has used to design evaluate software radio applications, and provides an excellent experimental platform for future software radio applications.

## 1.4 System Architecture

The primary design goal was to create a wireless communications system with the flexibility to make dynamic resource tradeoffs and optimizations discussed above. The ideal software radio, shown in figure 1-1, would involve direct wideband digitization at the antenna, with all subsequent processing performed in software. Note that this architecture moves the analog/digital boundary as close as possible to the antenna and the hardware/software boundary to the same place as the analog/digital boundary.

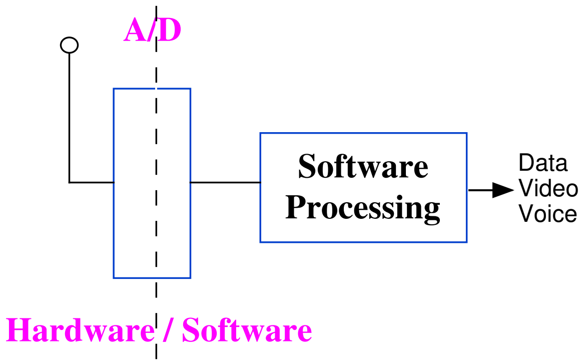


Figure 1-1: Ideal software radio.

The software radio system architecture presented in this dissertation and shown in figure 1-2, is divided into three sections:

- Analog front end plus A/D and D/A conversion.
- I/O system for transporting the digital data to and from application's memory, consisting of operating system extensions and the General Purpose PCI I/O system (GuPPI) DMA engine.
- A programming environment to support the implementation of computationally intensive, high data rate, real-time signal processing.

This system architecture makes only one concession to the ideal software radio architecture. The one concession was to first down convert a wideband of the spectrum to an IF frequency and then digitize it. The center frequency of the RF band is selectable in software and it is important to note that what is down converted is not just a single channel but a wideband (e.g. 10 - 20 MHz). For most systems this enables dynamic software modification of the multiple access protocol. Other than the restriction of looking at one particular band at a time, the system is functionally equivalent to the ideal software radio.

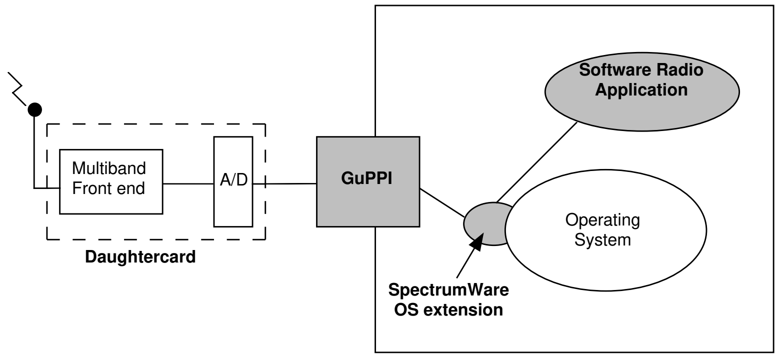


Figure 1-2: Virtual Radio Architecture

## 1.5 Software Architecture

To support the implementation of software radio applications, the SPECtRA programming environment was developed to support modular real-time signal processing applications. This environment is partitioned into two axes, as illustrated in figure 1-3. The in-band axis to support the data processing, and the out-of-band axis to support configuration and inter-module communication. The programming environment implements a novel data-pull architecture and a stream abstraction for data that result in computationally efficient, reusable software that requires relatively few lines of source code. A complete description of the programming environment is presented in Chapter 4.

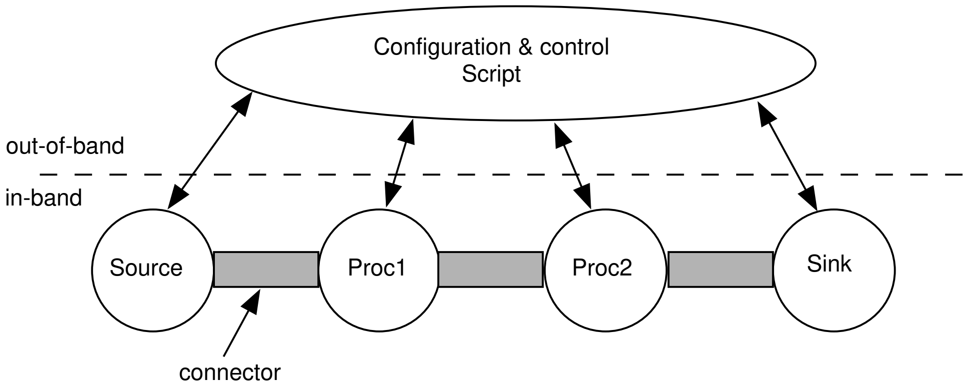


Figure 1-3: Graphical description of the programming environment.

## 1.6 Application Architecture

Software radio applications are implemented on top of the signal processing framework provided by the SPECtRA system. Radio systems have traditionally been divided into several separate stages, such as the RF, IF and baseband processing stage. These stages were defined primarily to facilitate hardware implementation. In a software implementation, layering is still important for modularity and specification, but the constraints that drive the definition of the layers are different. This thesis presents a layered software model, illustrated in figure 1-4, for specifying the signal processing requirements of a wireless communication system. The model provides a concise framework for specifying a radio and an implementation model that enables considerable software re-use. A description of the model along with a few example specifications are presented in chapter 5.

## 1.7 Previous Work on Software Radio

Software radio research has been primarily motivated by interoperability problems that result from the implementation of radios in dedicated hardware. Motivating applications have come from both the military and commercial cellular domains [Mitola, 1995]. The first significant software radio was the SpeakEasy system [Lackey and Upmal, 1995] which was designed to emulate more than ten different military radios. Commercial interest was spurred by an RFI from Bell South Cellular on “Software Defined Radio” [Blust, 1995]. The principle commercial application driver in the U.S. is the multiple incompatible cellular and PCS communications. In Europe the widespread deployment of GSM has mitigated interoperability problems, but there is significant interest is using software radio to enhance services for third generation cellular systems [Tuttlebee, 1999].

The ideal software radio, illustrated in Figure 1-1, would directly digitize the entire band of interest and transport the stream of digital samples to memory where it can be directly accessed by a microprocessor. Current wideband receivers, A/D converters, I/O systems and processors cannot meet the requirements imposed by a direct implementation of this architecture. Current research in the area of software radios covers the development of enabling technologies for signal acquisition and processing, the design of software systems

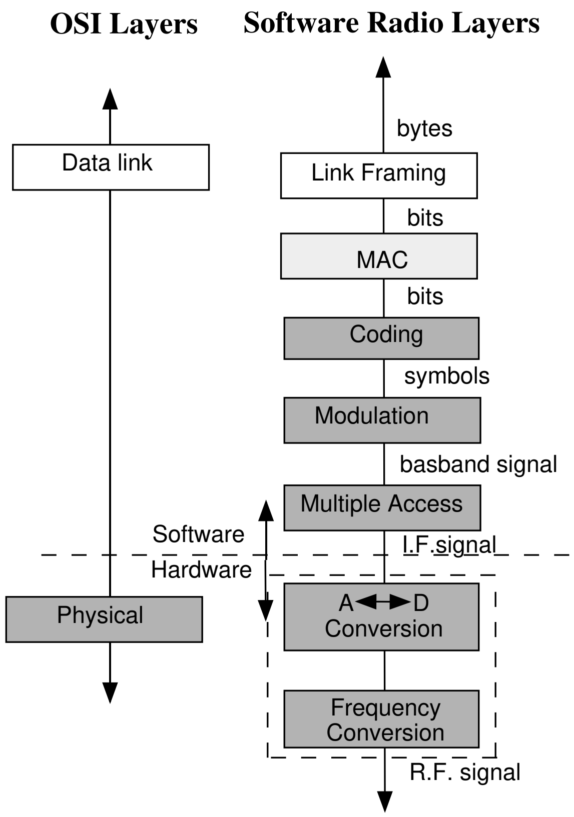


Figure 1-4: The Software Radio Layering Model shifts many physical layer functions into software.

for implementing data-intensive signal processing algorithms and software radio system design.

### 1.7.1 Signal Acquisition

Research into the enabling technologies for signal acquisition include the development of tunable wideband front-ends and of A/D converters capable of digitizing wideband signals with a signal-to-noise ratio sufficient to enable digital cellular applications.

Ideally, the front-end could be tuned to any band of interest, and would be capable of capturing the entire band of interest (e.g. 26 MHz for the 900 MHz ISM band). Most practical solutions involve several discrete front ends, each of which is optimized for a certain band. Active research aimed at producing a single chip tuner that is capable of operating in the 2 - 2000 MHz range is being performed as part of the DARPA Glomo program [1]. A discussion of the requirements and challenges associated with the design of a wideband tuners can be found in: [Brannon, 1998a] [Wepman, 1995] [Akos, 1997].

The development of A/D converters capable of digitizing the entire cellular band with enough resolution to implement the digital cellular standards in software is a significant

[1] http://www.darpa.mil/ito/research/glomo

research challenge. The key parameter is the spurious free dynamic range, which is a measure of the non-linear sources of error in the converter, and is often the limiting factor in the performance of high-speed converter. The GSM system imposes the most stringent constraints, requiring a 91 dB SFDR with a minimum sampling rate of 24 MHz. A review of the A/D parameters relevant to software radio applications is presented in Appendix A, and a discussion of the research challenges can be found in [Walden, 1999] [Brannon, 1998b] [Wepman *et al.*, 96].

### 1.7.2 Processing Engine

The choice of processing technology is a trade-off between flexibility and power consumption that is determined by the application requirements. Most software radio work has utilized DSP processors, such as the Texas Instruments C40 family as the primary computation engine. However, portable hand-held implementations have the most stringent power requirements. Recent work has examined methods such as dynamic clock and supply voltage control to reduce the power consumption of DSP processors [Gutnik and Chandrasakan, 1996], but the power dissipation requirements of some applications are so low that it has motivated research into alternate technologies that provide some degree of flexibility.

One low power solution to the interoperability problem is to build multiple hardware devices into one unit. Several companies are manufacturing multi-mode phones, that allow a single device to interoperate with multiple cellular standards. While these devices do enable a user to connect to multiple different networks it is a short term solution that cannot adapt to future standards. In addition, this solution does not provide flexibility to modify the signal processing algorithms or functions in order to adapt to the environmental or traffic conditions.

Reconfigurable logic is another technology that trades flexibility for a reduction in power consumption [Dick, 1998] [Rabaey, 1997]. The flexibility in the processing is significantly less than what can be achieved with general purpose or DSP processors, but the power consumption can be two orders of magnitude lower. The use of reconfigurable logic to perform the high data rate front-end processing, followed by a low power DSP for the baseband processing is an architecture that has been proposed to introduce more flexibility into a platform based on reconfigurable logic.

### 1.7.3 Software Engineering

The design of software system capable of supporting the flexibility provided by a software radio has been identified as one of the most significant challenges in realizing the promise of software radios [Rissanen, 1997]. The key goals are the development of a modular environment that provides abstraction from hardware, enables software reuse, reduces the cost of conformance testing, and supports over the air configurability [Taylor, 1997]. In addition, the development of a *Radio Construction Language* is seen as a critical enabling technology. Despite the importance of the software design, little work has been done in this area. Existing software radio systems use software that is designed primarily to meet the desired processing requirements with little thought given to these other important goals. This dissertation presents the first programming environment designed to support software

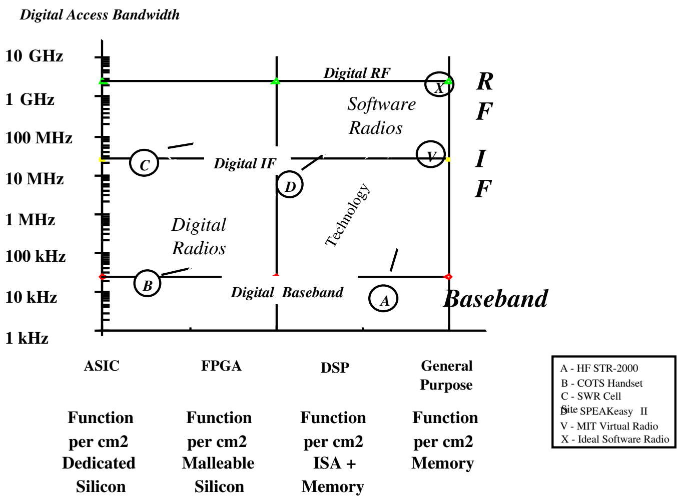


Figure 1-5: Software radio phase space

radios that meets these goals while supporting the real-time signal processing performance required by these applications. In addition, a software radio layering model is presented that forms the basis for the development of a radio construction language.

### 1.7.4 Software Radio Systems

Joe Mitola has proposed a software radio phase space [Mitola, 1999] that is a useful metric for comparing software radio systems. The software radio phase space, shown in Figure 1-5, represents a given radio implementation in terms of the digital access point and the degree of programmability. The vertical axis is not simply the frequency at which the A/Ds and D/A s operate, but the point at which the processing is fully programmable. For example, consider a cellular system that digitizes a 12 MHz band at a rate of 25 MHz and then uses dedicated hardware to select out a single 30 kHz channel, with a sampling rate of 60 kHz, to be processed in software. The digital access point would not be 25 MHz, but the 60 kHz sample rate that is subject to programmable processing. The horizontal access represents the degree of programmability of the underlying technology. Four different base technologies, ASICs, FPGAs, DSPs and general purpose processors are placed on the axis as a guide for mapping different implementations into this space. The virtual radio system described in this thesis and in [Bose *et al.*, 1999] occupies the point in the phase space that is closer to the ideal software radio than any other system reported in the literature. Following is a review of the significant work in software radio system design.

The Speakeasy Multiband Multimode Radio (MBMMR) is the only other complete software radio system in existence. The following reviews the Speakeasy system and then summarizes other research which has investigated various aspects of software radio systems.

### SpeakEasy

The primary goal of the SpeakEasy project is to utilize programmable processing to emulate more than 10 existing military radios, operating in frequency bands between 2 and 200 MHz . Further, another design goal was to be able to easily incorporated new coding and modulation standards can in the future, so that military communications can keep pace with advances in coding and modulation techniques.

In order to achieve the necessary processing capability with reasonable power constraints, a multichip module containing four TMS320C40 DSP processors was developed. The modules being developed under phase II of the program run at a clock speed of 50 MHz, which provides 200 million FLOPS, 1,100 MIPS and 300 Mbytes per second I/O, while consuming less than 10 watts of power.

The development of the software algorithms has directly emulated the analog processing functions, with additional features, such as digital encryption, incorporated into the system. The programming was done primarily in Ada, with time sensitive portions hand coded in assembly language.

The effort put into the DSP processor design, coupled with the low level programming, limit the portability of the system to other processing platforms. In fact, several DSP processors currently on the market outperform the 'C40, in particular the SHARC processor [Ana, 1994] from Analog Devices. A more generic approach to the processing and programming environments would have allowed the radio to track processor technology and advances in coding and modulation techniques.

The SpeakEasy system, and the Virtual Radio system presented in this thesis are the only two existing systems that sit in the upper right-hand quadrant of the software radio phase-space shown in figure 1-5. However, the flexibility of the Speakeasy system is considerably more limited than the Virtual Radio.

### Other Software Radio Systems Projects

An all digital L-band receiver with direct RF sampling is described in [Brown and Wolt, 94]. The receiver uses programmable digital hardware and is designed for satellite communications applications such as the Global Positioning System (GPS). The design demonstrates the feasibility of of direct RF sampling and digital separation of in-phase and quadrature signals. Software control over the hardware allows dynamic reconfiguration of the receiver for reception of GPS, radio-navigation and other spread spectrum applications. This system exhibits flexibility in the ability to interact with several existing communications systems, but the functionality was designed in to the hardware, so it cannot adapt to future developments.

A software radio implementation of the Global Positioning System (GPS) is presented in [Akos, 1997]. The intent of this work was not to explore the new opportunities presented by a software based system, but to explore the feasibility and reception performance of a software based system. The work is divided into two parts, design of the front end and the design of software to process the digitized signal. The system digitizes and stores a 30 second burst of the GPS signal, and then performs the processing in Matlab. This system doesn't map directly on to the software radio phase space, since it is not a real-time communications system, but does use wideband digitization as well as software processing on a general purpose processor. Front end architectures using direct conversion with both bandpass sampling and Nyquist sampling of a down converted signal were analyzed in terms of sensitivity and dynamic range. Several different processing algorithms were implemented. In order to facilitate the comparison with hardware systems, all of the algorithms were a direct software implementation of exiting algorithms. To demonstrate the flexibility of the software system, some of the algorithms for processing the signals from the GLONASS positioning system were also implemented.

Modems are an excellent example of how signal processing tasks can migrate from dedicated hardware to software as processing power increases. From their origins in dedicated hardware, modems have evolved through programmable hardware and DSP based designs to the software modem available from Motorola [Motorola, 1997]. This modem implements the V.34 standard and requires no special computational hardware, as it runs on the computer's Pentium processor (a version is also planned for the PowerPC). On a 150 MHz Pentium, the computational requirements of the modem are such that is runs transparently in conjunction with applications such as word processors and web browsers, but does cause a decrease in performance of more demanding applications such as video conferencing, and computationally intensive network games such as Quake. However, each increase in processor speed brings more free cycles per second, allowing the modem to coexist with more compute intensive applications in the future. Motorola plans to release software upgrades for future high-speed standards and software could be written to interoperate with non-standard modems. All of the signal processing functions are under software control and the modem can emulate other devices such as fax machines or telephones. The main differences between this modem and the proposed work on virtual radios is that the modem is a low bandwidth system, whereas virtual radios deal with passband signals and the data rates for the wideband radio applications will be 2-3 orders of magnitude higher, which will push the limits of the hardware and operating system software in ways that the modem does not.

Airnet currently manufactures software defined wireless base stations. The system uses a broadband radio (5 MHz Tx and 5 MHz Rx), and 100 DSP processors running the SPOX operating system and interconnected via a VME bus. SPOX is a real-time operating system and multitasking scheduler designed specifically to run on DSPs. The core of the system is the proprietary *Carney Engine*, which combines multiple baseband transmit channels into one wideband transmit waveform, and also extracts the individual waveforms from the received wideband signal. The airnet system currently supports both the AMPS and GSM protocols. Airnet's product provides a good contrast to the general purpose processor approach taken in this thesis. Airnet required 100 DSP processors to implement an entire basestation (e.g. 96 GSM channels). This thesis investigates the use of general purpose processors, with their considerably higher clock speeds, as an alternative. Not only can this approach result in the use of fewer processors, but the software development can be greatly simplified and the underlying hardware upgraded with little or no work needed to port the software.

## 1.8 Contributions

The major contributions of this thesis are:

- A demonstration that is is possible to implement high data-rate, computationally intensive real-time signal processing applications on a general purpose platform.
- The design of a system architecture for wideband software radios. The flexibility of this system, coupled with wideband digitization enables wireless communications systems that make better use of the available spectrum and provide better application level performance than existing systems.
- The development of the data-pull model for execution of computation in a modular signal processing system.
- The characterization of the computational variability of signal processing algorithms on a general purpose processor.
- The design and implementation of a programming environment to support real-time signal processing applications on a general purpose platform.
- Design guidelines for the implementation of real-time systems on a general purpose processor and the implementation of physical layer functions in software.
- An illustration of some areas in which the design of software signal processing algorithms differs from design for a hardware or DSP implementation.
- A layered model for the specification of physical layer of communications systems .

## 1.9 Road Map

The primary performance bottleneck for software radio applications is the data I/O rate. Chapter 2 describes the design and evaluation of the data acquisition sub-system which overcomes this limitation and provides I/O rates to and from the application which are required by many wideband software radio applications.

Once the I/O bottleneck was eliminated, the performance limiting bottleneck was moved to the memory access and processing of data on a general purpose platform. Chapter 3 examines one of the most significant challenges: the variable computation environment provided by a general purpose processor and operating system, presents a model for the computational variability and a method for characterizing the variability in execution times. This model is used to design application level mechanisms to support the real-time needs of virtual radio applications.

The next two chapters describe the software components of the architecture. Chapter 4 describes the SPECtRA programming environment, which implements a novel data-pull mechanism for scheduling of computation and handling of real-time constraints. This environment enables the software implementation of the wide range of physical layer functionality in a modular environment that supports dynamic modification to the processing. Chapter 5 describes the software radio layering model, a universal framework for specifying the physical layer. This model is proposed as a tool for organizing the vast and complicated functions traditionally lumped into the black box called the physical layer. This framework allows devices dynamically design a radio out of component building blocks, specify the capabilities to other devices and incorporate new functionality by downloading software modules.

Chapter 6 evaluates the performance of the traditional systems implemented in software. These applications demonstrate that the cost metrics used for traditional radio design are not appropriate for a software radio architecture. The primary bottlenecks for both reception and transmission applications are identified, and Chapter 7 presents several approaches to application and algorithm design that reduce, and in some cases eliminate, the performance bottlenecks.

The applications presented demonstrate that workstations are capable of performing a wide range of real-time signal processing tasks. Chapter 8 discusses the key system aspects that enable this performance, and discusses DSP design constraints in the context modern general purpose processors to better understand their capabilities and limitations. The thesis then concludes with some comments on ongoing and future work, and a summary of the contributions in chapter 9.

# Chapter 2

# Preparing the Signal for Processing

The software radio architecture described in this dissertation is based on a framework that defines abstraction boundaries between regions based on the manner in which the processing is performed. A typical digital receiver has two such regions, an analog processing region and a digital processing region, denoted by the dashed vertical line in figure 2-1. In a software radio, at least some of the processing is performed on a programmable processor, which adds a new region. This new barrier denotes the point at which the processing changes from hardware based to software based. The architecture presented here moves the analog/digital boundary as close to the antenna as possible, and moves the hardware software/boundary right up to the analog/digital boundary. This approach maximizes the flexibility by performing as much processing as possible in software.

The system architecture presented in this dissertation and shown in figure 2-2 makes only one concession to the ideal architecture depicted in figure 1-1, which is to first downconvert a wideband of the spectrum to an IF frequency and then digitize it. The center frequency of the RF band is selectable in software and it is important to note that what is downconverted is not just a single channel but a wideband (e.g. 10 - 20 MHz). For most systems this enables dynamic modification of the multiple access protocol, since it is implemented in software. Other than the restriction of looking at one particular band at a time, the system is functionally equivalent to the ideal software radio.

The primary bottleneck to system performance is the I/O throughput to the application. Existing workstation architectures cannot supply data at a rate sufficient to enable software radio applications. This chapter presents the design and performance of a signal acquisition and I/O sub-system that overcome this bottleneck.

## 2.1 Signal Acquisition

In order to perform all of the signal processing, including the multiple access protocol, in software it is necessary to digitize a signal containing multiple channels. For example, software access to any channel in the AMPS cellular system would require digitizing a 12 MHz wide portion of the 800 MHz band. The A/D and D/A conversion of a wideband RF or IF signal enables the software processing, but does have a cost: the noise introduced by

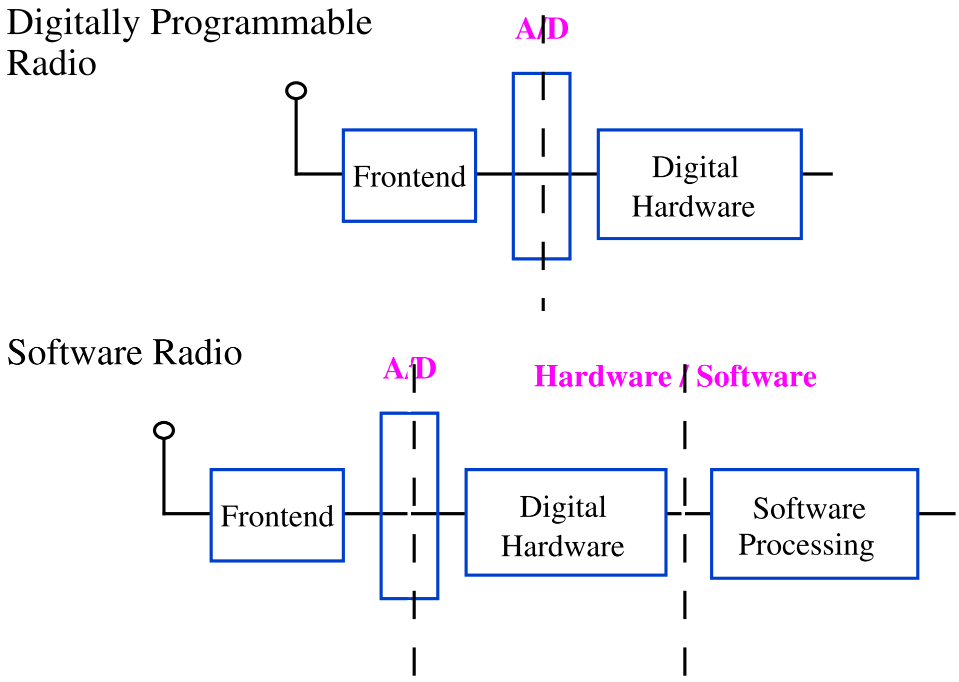


Figure 2-1: Abstraction boundaries for a digital and software receiver.

the converter can significantly limit the capabilities of the system. This issue is explored in Appendix A.

The signal acquisition portion of the system has three components: a wideband tuner that converts between the band and a 10.7 MHz IF frequency; a digital conversion stage; and an I/O system to transport streams of samples to and from memory. There is a clean interface between each stage, which allows each portion of the system to track technology advances. The signal acquisition and digital conversion portions of the system utilized off-the-shelf components. The custom designed GuPPI I/O system is described in section 2.2.

For signal acquisition, the system utilized off-the shelf components. Several different components were utilized throughout the course of development. The most flexible solution for

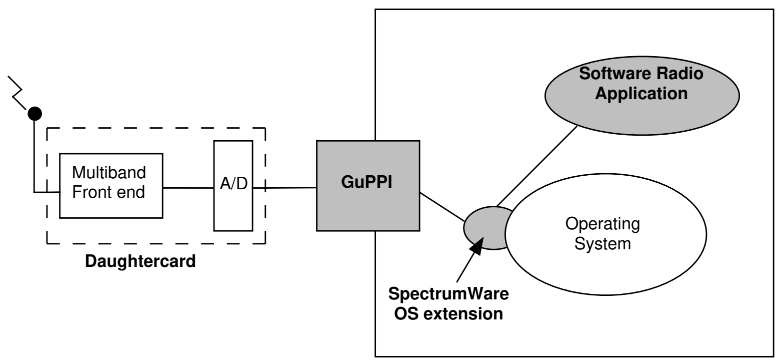


Figure 2-2: Virtual Radio Architecture

reception was the AR5000, which permitted the conversion of a 10 MHz wide band of the spectrum located between 100 kHz and 1.6 GHz. For transmit, tunable wideband solutions were not readily available. Instead distinct front ends, composed of components produced by RF Microdevices were implemented for the VHF, UHF and 2.5 GHz bands.

The Analog devices AD9042 was used for A/D conversion, which supports 12 bit quantization at 40 MSPS. For D/A conversion, the AD9713 provided 12 bit resolution at up to 80 MSPS. These converters were built onto a daughter card which attaches to the GuPPI. This allowed for the incorporation of new converters into the system without having to re-design the I/O system as well.

## 2.2 I/O system

The General Purpose PCI Interface (GuPPI), designed with Michael Ismert, was used to transport the samples from the D/A converter to memory, and from memory to the A/D converter. A brief overview of the system is given here, additional details can be found in [Ismert, 1998].

In standard signal processing systems based on dedicated digital hardware or DSPs, the incoming samples arrive at a constant rate and are processed with a fixed delay between when a sample enters the system and when the output based on that sample leaves the system. The processing happens in lockstep with the I/O, so the DSP is guaranteed that it will have a constant stream of regularly spaced samples on which to do processing.

In a general-purpose workstation, however, such simple guarantees do not exist. Virtual memory, multiple levels of caching, and competition for the I/O and memory buses add jitter to the expected amount of time required for a sample to travel from an I/O device to the processor. In addition, using a multi-tasking operating system ensures that the signal processing application will not always be the active process, which adds jitter to the rate at which samples are processed. Smoothing out these sources of jitter is one requirement that must be addressed by the I/O system.

The other major requirement that must be addressed is the need for high throughput between the application and the A/D converter. Consider, for example, a software cellular receiver. The A-side cellular telephony band (reverse link) is 12.5 MHz wide. If digitized at 25.6 MHz with a sample size of 16 bits, the data rate necessary to transfer this stream of samples to the application would be 409.6 Mbits/sec. These throughput needs expose two bottlenecks in the existing workstation architecture. First, workstations lack a high-throughput port into which our front end can be connected, creating the need to develop custom hardware. Second, the path between a device driver and the application is rather inefficient, requiring modifications to the operating system. For comparison, the VuSystem [Houh *et al.*, 1995] reported sustained throughput of 100 Mbits/sec to the application with an unmodified Digital Unix operating system.

There are two main components in the architecture of the I/O system: the GuPPI (for General Purpose PCI I/O), which physically connects the analog front end to the workstation's I/O bus, and operating system additions, which provide the means for the application to access the sample streams.

### 2.2.1 GuPPI design

The GuPPI provides the system's external interface to the analog front end. This interface must accept samples from and provide samples to the analog front end at the wideband sampling rate.

The GuPPI provides the ability to burst data between the analog front end and main memory at near the maximum I/O bus rate. In addition, the GuPPI is responsible for decoupling the timing between the fixed rate domain of the analog front end and the variable rate I/O bus without losing any samples. This effectively absorbs any jitter caused by the bursty access to the I/O bus. These functions are performed without significant intervention from the processor. For typical applications, the required processing overhead per sample is less than half a cycle.

The GuPPI has a simple, generic daughter card interface to which analog front end-specific daughter cards are designed. This interface is directly connected to a set of FIFO buffers in both the input and output directions. These FIFOs provide the buffering necessary to absorb jitter caused by the bursty access to the PCI bus.

The GuPPI implements a new variant of scatter/gather DMA that we have named *page-streaming*. The GuPPI has two page address FIFOs, one each for input and output, which hold the physical page addresses associated with buffers in virtual memory. At the end of a page transfer, the GuPPI reads the next page address from the head of the appropriate page address FIFO and begins transferring data to/from it. The GuPPI triggers an interrupt when the supply of page addresses runs low, and the page addresses are replenished by the interrupt handler in the device driver. Page-streaming is unique in two ways. First, the physical page addresses are stored on-board the GuPPI rather than in a table in memory. Since the processor replenishes the addresses, the GuPPI only uses its bus grants to transfer data; this simplified the design of the GuPPI and results in more efficient use of the PCI bus. Second, with page-streaming only complete pages are transferred; this is made possible by the constant flow of samples and automatically results in page-aligned, integral page length buffers that are easy for the operating system to manipulate.

Within our system, the operating system components are responsible for ensuring that the flow of data between the GuPPI and kernel buffers is continuous. This includes providing facilities for absorbing the jitter due to scheduling and interrupts. The operating system support consists of a device driver for the GuPPI and several small additions to the virtual memory system in the Linux kernel[1]. The total size of the code is just under 1000 lines, with the virtual memory system additions representing just 200 of those. Another important aspect of the additions is that they do not affect the performance or functionality of any part of the system not related to the GuPPI; all other applications run completely undisturbed.

The virtual memory additions provide the low-overhead, high-bandwidth transfer of data between the application and the device driver and the external interface to the application. To the application, the GuPPI appears to be a standard Unix device with copy semantics. However, virtual memory manipulations are used to make the `read` and `write` system calls to the GuPPI copy-free. If the calls were implemented in the usual manner, there would not be enough CPU cycles available to copy the data stream from kernel memory to application

[1]The Linux kernel version used is 2.0.30.

| Input | Output |
|---|---|
| 933 | 790 |

Table 2.1: Raw GuPPI Performance (Mbits/sec)

memory. Using the virtual memory manipulations, this overhead was reduced to less than one half of a cycles per input sample.

It is the responsibility of the *application* to provide real-time guarantees. If the application wishes to run in real-time, then it is responsible for processing data at or above the average rate at which our system will transfer data. If the application does not run in real-time, then our system will eventually begin to drop input samples or produce gaps in the output

The GuPPI I/O system is capable of supporting a sustained I/O of up to 512 Mbps, which is more than sufficient to support many interesting wideband software radio applications. A more detailed evaluation of the performance is presented in chapter 6, which demonstrates that, with the GuPPI I/O system, the I/O rate is no longer the limiting bottleneck to system performance. The primary bottleneck is now the processing of the data itself.

### 2.2.2 Performance

The measurements of the GuPPI and the I/O modifications were taken on a 200 MHz Pentium Pro system running Linux with a 33 MHz, 32 bit wide PCI bus. All cycle values were gathered using the Pentium Pro cycle counter.

The maximum rate at which an application using the GuPPI driver can maintain a continuous flow of input samples from the current version of the GuPPI is 512 Mbits/sec. This number was determined using an application that only accessed enough samples per input buffer to verify data continuity. This number provides an upper bound on the possible throughput that an application can achieve using the GuPPI. At rates above this point there is insufficient depth in the input data FIFO on the GuPPI to absorb jitter due to the PCI bus, but a new revision of the GuPPI currently being fabricated will have deeper data FIFOs, increasing the maximum continuous throughput.

In order to provide this high continuous throughput, the GuPPI must have higher raw throughput. Table 2.1 shows the maximum input and output burst performance of the GuPPI. These numbers reflect the amount of time required for the GuPPI to DMA approximately 1.2 MB of data. The measurements include only the amount of time required to DMA the data, not the time required to write page addresses into the appropriate page address FIFO. The maximum PCI throughput available is 1056 Mbits/sec, so the GuPPI is coming reasonably close to saturating the workstation's PCI bus. The lower maximum throughput for output is due to the latency incurred when reading values from main memory.

Figure 2-3 shows the average processing overhead imposed on the workstation by using the GuPPI to generate input. This measurement reflects the number of cycles required per input sample, and takes into account both the overhead required to perform the `read` system call (which includes the virtual memory swap) and the overhead required to handle

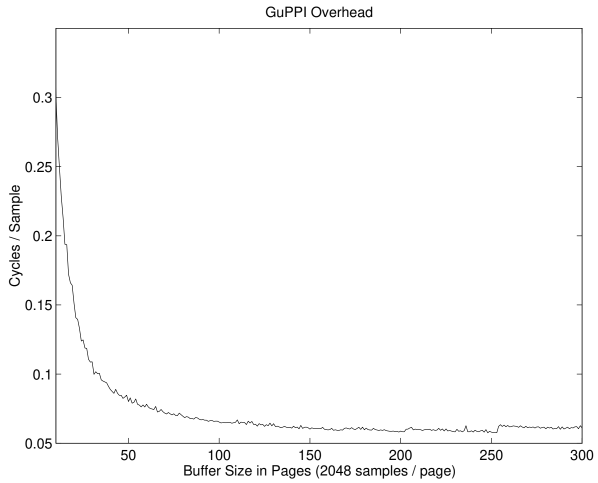


Figure 2-3: GuPPI Processing Overhead (Input)

interrupts generated by the need to replenish the supply of input page addresses[2].

## 2.3 System Design

The I/O sub-system presented in this chapter has eliminated the primary performance bottleneck for software radio applications in a workstation environment. The system can efficiently acquire samples of the spectrum and place them in memory where they can be accessed by the processor. The primary bottleneck has now been moved to the actual processing of the data. The key challenges are to design a programming environment that supports flexible real-time signal processing applications in an environment where the availability of computational resources varies.

The next three chapters address software design issues. Chapter 3 presents a method for characterizing the variability in execution times in a general purpose environment and discusses design methods that permit the implementation of real-time applications in this environment. These methods are then incorporated into the design of the SPECtRA programming environment presented in Chapter 4.

Chapter 5 describes a software radio layering model, which can be used to specify radio systems. The programming environment incorporates an object hierarchy for signal processing modules which supports the layering model and enables dynamic, incremental modification of a radio system.

[2]The measurements assume 16 bit samples.

# Chapter 3

# Dealing with Variability

One of the major differences between the work presented in this dissertation and other software radio work is our use of a conventional workstation, that is a computation platform that utilizes both a general purpose processor and operating system. While this choice brings tremendous flexibility and a much better set of design and development tools to bear on real-time signal processing problems, it also poses significant challenges to the implementation of real-time signal processing applications. The most significant challenge is the variability execution times [Bier *et al.*, 1997].

In this chapter we describe an application level approach to real-time system design in which deadline misses can be dealt with in a manner that has little or no effect on overall application performance. The approach uses application level mechanisms to implement a data-intensive real-time system in a workstation environment. We begin with a description and measurement of the computational variability, and the model of computation used to design the mechanisms. Following this, we present the notion of a statistical real-time system, describe the mechanisms required and discuss the class of real-time applications that can be supported by this model.

## 3.1 Computational Variability

The most significant challenge to implementing real-time signal processing applications in a general purpose environment is dealing with the variability in executions times. If the execution times were consistent, as is the case for many DSP processors and operating systems, then the real-time performance can be predicted. With a variable execution environment, the real-time performance can only be bounded by the worst case performance, which can be an order of magnitude worse than the average performance.

In the scope of this dissertation, general purpose processors are defined as those processors that are designed to optimize the average performance over a wide range of applications. By contrast, DSP processors are typically designed to optimize the worst case performance in order to place the best possible pessimistic bound on real-time performance. There are many features of general purpose processors that contribute to the variability in execution times, including caches, branch prediction, dynamic execution and data dependent execution times. These features improve the average performance at the expense of the worst case performance.

General purpose or *full-featured* operating systems have several characteristics that introduced variability into the execution time of a particular algorithm. System interrupts can interrupt a user-level process at any time and performs a variable amount of work depending on the circumstances. The scheduler, which facilitates multi-tasking, introduces a coarse grained variability as it swaps tasks in and out. A general purpose operating system offers many desirable features, including multi-tasking, abstraction and protection that could lead to improved implementations of existing communications systems, and greater functionality in future systems. However, the challenge of implementing a data intensive real-time signal processing system in this environment must first be overcome.

In order to design a real-time system using a general purpose environment it is necessary to have a model that describes the computational environment. The proposed model is a processor that runs at a variable rate. The model also proposes that the sources of variability can be divided into two groups, one that is dependent upon the particular application being run, and another that does not depend upon the particular application being run. There are many sources that contribute to the variability in execution time, and some of them depend on the data being processed or the activity of other processes. For these reasons, it is not feasible to build a deterministic model of the computational variability. Instead, the processing environment must be characterized by the statistics associated with the variability in execution time.

This section focuses characterizing the variability in the execution times required by signal processing algorithms on a general purpose platform, and assessing the validity of the proposed model of computation. The measurement technique used to characterize the variability is described, followed by experimental results that verify the model. The verification of the model is broken into two stages. First the variability is characterized for a single algorithm, an FIR filter. This characterization includes the effects of parameter changes and system activity on the variability. The second stage of the analysis, presented in section 3.1.3, addresses the performance of algorithms for several other signal processing functions. The section concludes with some observations about the processing environment, the validity of the model of computation in light of the measurements, and the partitioning of the design challenges that must be addressed in order to implement real-time signal processing applications on a general purpose processor.

### 3.1.1 Measurement Technique

Most computer system performance metrics involve taking the average over a large number of trials. This averaging smoothes out the details of the variability present in the system. To characterize the variability associated with signal processing applications on a general purpose processor a novel method was developed to determine the cumulative distribution of cycles required, which captures both the magnitude and character of the variation in the computation without significantly affecting the performance of the application.

The measurement technique used was to incrementally compute a cumulative distribution of the cycles required to compute each output point using the model specific registers on the Pentium family of processors [Intel, 1998]. First a threshold number of cycles is chosen, and

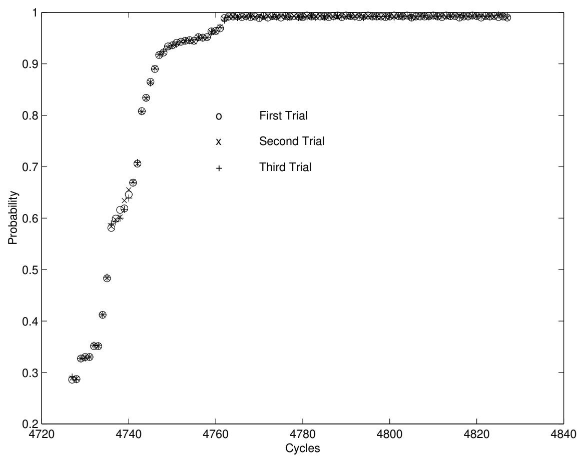


Figure 3-1: Cumulative distributions of the number of cycles required to produce each of 10,000 output points of a 500 tap FIR filter for three separate trials, which demonstrates the reproducibility of the distribution.

then the number of trials for which the cycles (or other quantity of interest) required were below the threshold was computed. To achieve this, the cycles were accumulated during the computation of a single output point. The resulting value was then compared to the threshold, if it was less than the threshold a count variable was incremented by one. This was done for each of the output points and the resulting count was one point of a cumulative distribution. This was then repeated for other threshold values in order to obtain more data points of the cumulative distribution. This measurement method required storage of only one additional value, the cumulative count, which minimized the effect on the algorithm's performance.

Figure 3-1 shows the cumulative distribution of the time required to produce each output of a 500 tap FIR filter for three separate trials of 10,000 output points each. Each output point was written into a unique location in an array, simulating the output of real data. Note that the cumulative distribution is repeatable over different trials, which This is significant because it indicates that although the computation is variable, the statistics are not. Thus the measurement technique of using many different trials to generate the distribution is valid.

### 3.1.2 FIR Computational Performance

This section evaluates the performance of several different FIR filters on a lightly loaded system[1]. This function was chosen because it very common in DSP systems and displays data intensive characteristics that are typical of many DSP algorithms. Each of the filters were implemented using floating point representations for both the taps and the data.

[1] The performance numbers given are for an FIR filter implemented in direct form in C-code and compiled with gcc version 2.7.2.1, using the -O2 flag. All measurements were made on a PPro-200. Lightly loaded refers to a systems running the normal system daemons but no other applications that use a significant amount of resources

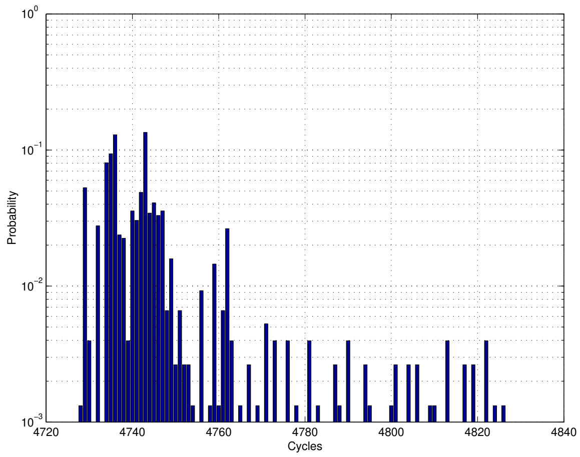


Figure 3-2: Histogram for the number of cycles required for the computation of a single output point of a 500 tap FIR filter. Note that the y-axis is plotted on a log scale, and that over 99% of the trials fall into the region below 4820 cycles, which corresponds to 24 $\mu$sec of running time.

Figure 3-2 shows the histogram for the number of cycles required for the computation of a single output point of a 500 tap FIR filter. The histogram was derived by taking the derivative of the cumulative distribution shown in figure 3-1. The minimum number of cycles required was repeatable over different trials. What is interesting is that most of the trials require a value close to the minimum value, and there are only a few outliers that take significantly more time. Table 3.1.2 shows the percentage of trials that would be completed within three different time bounds. Note that 99.5% of the trials required a computation time that was roughly 3% greater the minimum computation time. This suggests that the sources of variation can be divided into two groups, one which causes small perturbations, on the order of a few percent, in the number of samples required, but occurs frequently and one which causes larger variation in the number of samples required, but occurs rarely.

| Percent Completed | 34% | 64% | 99.5% |
|---|---|---|---|
| Time bound | 23.2 $\mu$sec | 23.7 $\mu$sec | 23.9 $\mu$sec |

Table 3.1: Time bound required to complete the given percentage of the computations.

There are two factors that can affect the computational requirements of the FIR filter, one is the running time and the other is the number of filter taps. The cache structure plays a major role in determining the impact of these factors. Since DSP algorithms are often long running, data-intensive applications, the data caches will periodically go stale as new data is brought in, and outputs are written into new locations. Increasing the number of taps means that the computation of each output point will require accessing more memory locations, thus the memory hierarchy will play a more significant role.

The PentiumPro used for these measurements has closely coupled L1 and L2 caches, both of which can be accessed via a full clock speed bus. The L1 cache contains a 4-way set associative 8 Kbytes segment for code and an 8 Kbytes, 2-way set associative segment for data.

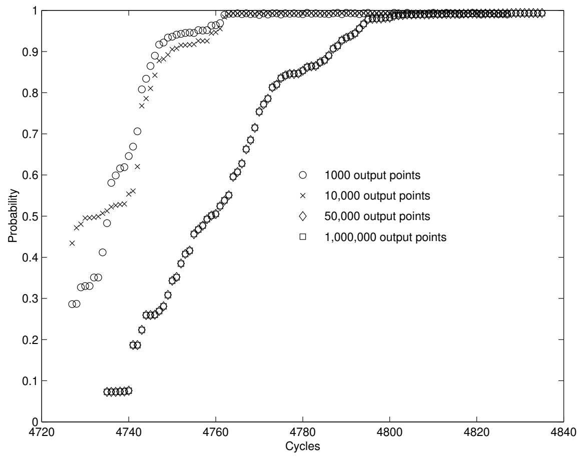


Figure 3-3: Cumulative distributions of the number of cycles required to produce each output point of a 500 tap FIR filter. Plots for several different running times of the algorithm are shown.

Figure 3-3 shows several cumulative distributions of a 500 tap FIR filter for five different running times. Plots are shown for the computation of 100, 10,000, 100,000, 1 million and 3 million output points. In each case there is also an input buffer that has a size equal to the output buffer plus the number of taps. The spread of the distribution of cycles near the minimum is affected, and the number of cycles required to surpass a particular threshold (e.g. 99% of computations) is also affected. However, there is no significant change in the longer term variation.

Figure 3-4 shows the cumulative distributions for several FIR filters of different lengths. The number of cycles required has been normalized to the minimum number of cycles required for that particular experiment, which allows a comparison of the variation between filters that require a different number of cycles. The point at which the distribution reaches the 99% threshold increases as the number of taps increase. However, increasing the number of taps does not necessarily increase the degree of variability, as is illustrated by the plots for the 2000 and 5000 tap filters. The reason for this behavior is the interaction with the underlying cache structure.

Figure 3-5 shows the cycles consumed while a data cache miss is outstanding as a function of the number of taps, and lends insight into how the cache structure affects the variability. For tap numbers below 1000, there are very few cache misses. This is because 1000 floating point taps occupy 4 Kbytes worth of cache space, which corresponds exactly to one of the sets in the 8 Kbytes, 2-way set associative data cache. For tap values above 1000, some of the values are stored in the other set, and collide with the input and output data streams. The entire cache is 8 Kbytes in size, which corresponds to 2000 floating point taps. As the number of taps is increased beyond 2000 taps, there are proportionally more accesses to the L2 cache, and the cycles during a DCU miss increase linearly with the number of taps.

The 2000 tap filter displays more variability than the 5000 tap filter because it still takes considerable advantage of the L1 cache, which improves the average performance. The 5000 tap filter does not benefit as much from the L1 cache, so it's average performance is worse, but it's variability is less. This comparison is similar to the tradeoff made in many DSP

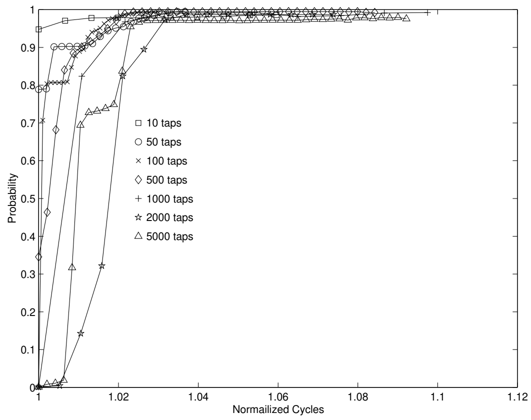


Figure 3-4: Cumulative distributions of the number of cycles required to produce each of 10,000 output points for several FIR filters of different length. The cycles required have been normalized to the minimum for each filter in order to compare the variability associated with different filter lengths.

designs which sacrifice data caching in return for less variability.

Although these experiments significantly taxed the cache structure of the system, the variation introduced into the execution time of the algorithm was less than 5% of the minimum execution time required. The rare but larger variations are not affected by the utilization of the caching system.

To assess the suitability of this platform for real-time signal processing applications three additional questions must be answered: what is the source of the larger variations, what is the magnitude of these variations, and do these results apply to other signal processing algorithms besides the FIR filter ?

### System Activity

This section demonstrates that the infrequent, but larger variation in execution times are primarily a result of system activity. The performance numbers presented in the previous section were made under the condition that the signal processing application was the only application that placed significant demands on the CPU. Figure 3-6 shows the cumulative distribution of cycles required for a 100 tap FIR filter under five different system loads. The five system conditions are: 1) with another CPU intensive process 2) with another memory intensive, but not CPU bound process 3) while the machine is being flood pinged 4) heavy interrupt load due to mouse activity 5)lightly loaded. The flood ping case is the only one that differs significantly, with a considerable number of trials that take orders of magnitude longer than the average. This is most likely due to receiver livelock [Mogul and Ramakrishnan, 1997], and not a condition that would be expected to occur under normal operation. However, it is encouraging to note that the excessive interrupt processing seems to only have a significant on the long term variation, moving the boundary down to down to about 95%.

It seems reasonable to conclude that system activity does not have a significant impact on

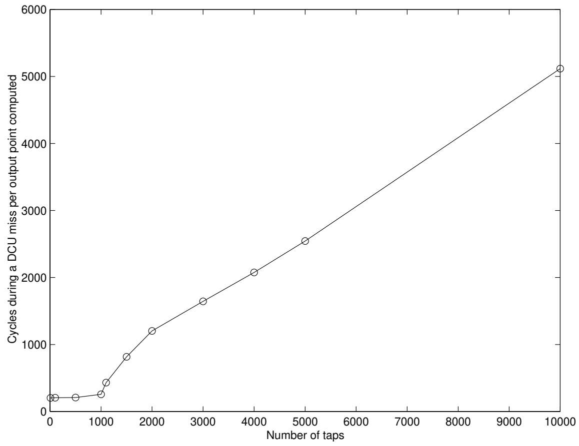


Figure 3-5: Number of cycles while a Data Cache Unit miss is outstanding, for several different FIR filter lengths.

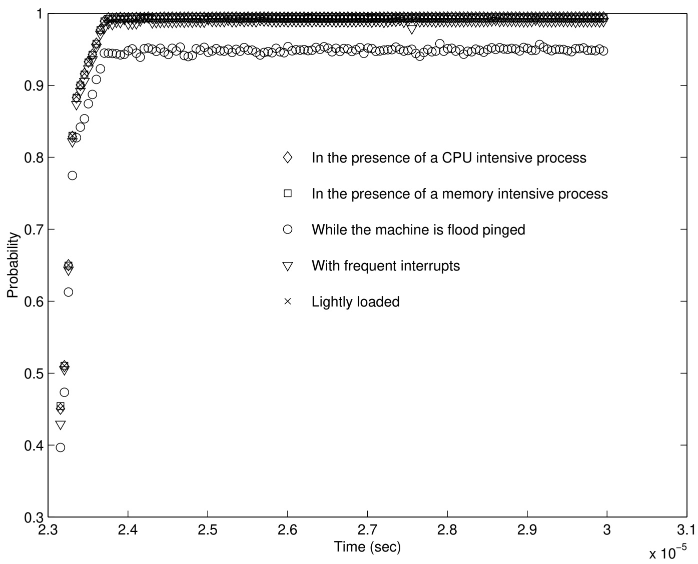


Figure 3-6: Cumulative distribution of cycles for a 500 tap FIR filter under five different system conditions: 1) with another CPU intensive process 2) with another memory intensive, but not CPU bound process 3) while the machine is flood pinged 4) heavy interrupt load due to mouse activity 5)lightly loaded

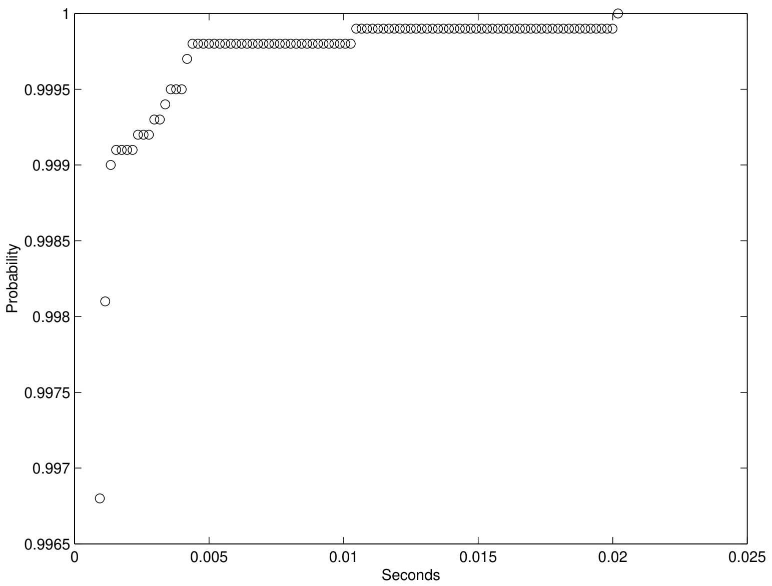


Figure 3-7: Expanded graph of the distribution for a 500 tap FIR filter while the machine is being flood pinged. The plot shows the effects of frequent interrupt activity as well as system scheduler activity.

the overall distribution of cycles required. However, it does have an impact on what happens during the tail of the distribution. The flood ping experiment provides insight into the larger variation components, since it involves considerable interrupt processing and thus runs for a longer time which brings the scheduler into play. Examining the tail of the distribution, shown in figure 3-7, reveals these two primary source of the larger variation. The abrupt shifts at 10 and 20 msec are the effect of the Linux system scheduler, which has a 10 msec granularity. The other component, that adds variation on the scale of 10-50 $\mu$sec is due to the interrupt processing. Note that this is not the interrupt latency number often reported for operating systems. Interrupt latency refers to the amount of time elapsed between when an interrupt is signalled and an interrupt is serviced, whereas interrupt processing refers to how much time the kernel spends actually processing the interrupt. The interrupt latency of Linux is typically 10 $\mu$sec with a maximum of 20 $\mu$sec.

Assuming that the starting time of the process is uniformly distributed over the timer interval, and that required computation takes less than one timer interval, the probability of a scheduler interrupt can be calculated by taking the minimum number of cycles required and dividing by the number of cycles in a 10 msec interval. The 100 tap filter required a minimum of 974 cycles, on a 200 MHz PentiumPro processor, this corresponds to scheduler interrupt probability of:

$$\frac{cycles\ required}{clock\ speed \cdot timer\ interval} = \frac{974}{200000000 * .01} = 4.9 \cdot 10^{-4}$$

If the starting time of the process is not uniformly distributed over the interval, but weighted towards the beginning of the interval, then the probability of a deadline miss would decrease.

The measured probability for the 100 tap example was $2.5 \cdot 10^{-4}$, indicating that the uniform distribution of the starting time is a conservative assumption. The 10 msec granularity is an artifact of the Linux scheduler. The granularity of the scheduler is chosen to balance the overhead of a context switch with the responsiveness of the system. The overhead associated with a context switch is primarily due to a small constant number of memory load store operations to save the current context and initialize the new one. The actual time consumed by the overhead decreases with increasing CPU and memory performance, however the granularity of the scheduler has remained unchanged. This is primarily due to the fact that the applications that drive the system responsiveness are primarily based on human interaction, and the 10 msec time slice has been sufficient. However, with the more demanding responsiveness of real-time signal processing applications, the scheduler interval could be decreased without incurring more overhead, in terms of time, than has been present historically. In fact the Version of Linux for the Axp processor has a default system timer of 1 msec, which does not adversely affect system performance.

If we were interested in using a general purpose processor for a single dedicated signal processing function, the operating system would not be necessary and we would only have to concern ourselves with variability on the scale of a 5% in the required number of cycles. However, the scope of this thesis is concerned with integrating real-time signal processing into the desktop environment, which includes running concurrently with other applications, and integrating signal processing into existing user applications.

### 3.1.3 Characterization of Other Algorithms.

The analysis presented above was for the FIR filter, and it is quite reasonable to suspect that other algorithms, with different computational and memory usage patterns, would have very different characteristics.

Figure 3-8 show the cumulative distributions for several different algorithms. The FFT routine used was the *four1* routine from Numerical Recipes in C [Press *et al.*, 1993]. These algorithms have different computational and memory access requirements, but they all display small scale variability on the order of a few percent of the minimum number of cycles required. While the localized behavior around the median may be very different for different algorithms, as shown in figure 3-9, the distribution of the longer times is primarily a function of the running time of the algorithm.

This is an important result for system and algorithm design, as it indicates that a estimate of the running time of the algorithm will provide enough information to estimate the larger variations in execution times, and the more frequent variations only cause variability on the order of a few percent of the minimum number of cycles required.

### 3.1.4 Model of Computation

The development of the statistical real-time concepts presented later in this chapter are based on a model of computation for a general purpose workstation. The model is simply a processor that runs at a variable speed that is not predictable. While the actual speed of the processor at any instant cannot be predicted, it is assumed that the statistics of the variation are known. The measurements presented in this section demonstrate that the

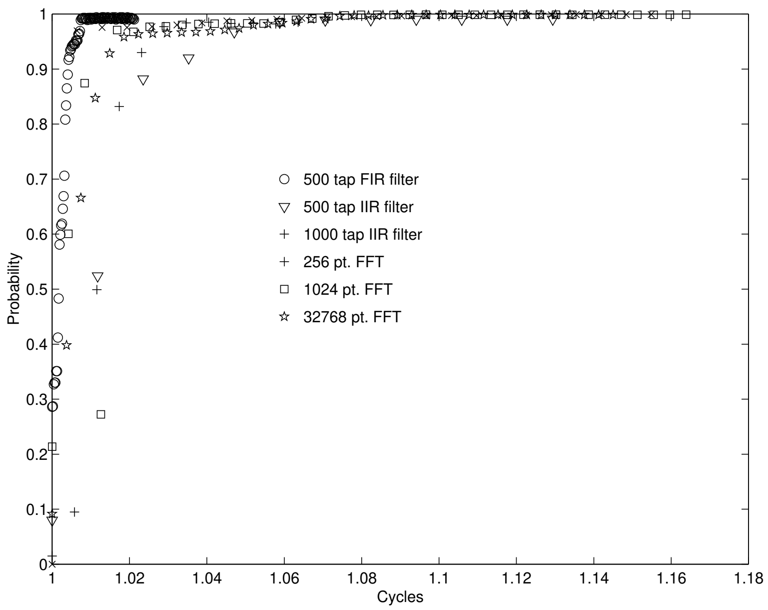


Figure 3-8: Cumulative distribution of cycles required for several different algorithms.

sources of variability can be grouped into two classes: small scale variations (on the order 100 cycles) that occur frequently and large scale variations (on the order of 10,000 cycles) that occur rarely. The large scale variations are primarily due to interrupt handling in the operating system. The scheduler also contributes occasional variations on the time scale of 10 msec, which is fixed regardless of the CPU speed. This time granularity is a result of scheduler design that has not tracked technology improvements. The increases in clock and memory speeds of modern processors allows the scheduler interval to be significantly reduced without incurring additional overhead in terms of the time required for a context switch.

The model suggests that two mechanisms are required. Since the small scale variations are frequent, they must be dealt with in a way that does not cause a deadline miss. The larger variations will cause deadline misses, but are infrequent, so the goal is to find a way to minimize their impact on the overall application performance.

### 3.1.5 Observations

The experiments presented in this section characterized the variability in execution times associated with running a several different DSP algorithms on a a general purpose platform. They demonstrate that there are two different time scales associated with the variability of the computation time.

The short term variability, which is on the order of 5% of the minimum number of cycles required, is due to the various structures in the processors that introduced variation into the execution time and memory access times. This variability is algorithm dependent, as it depends upon the algorithms usage of both memory and processor features that introduce variability. In the absence of another mechanism, this variation could be dealt with in most

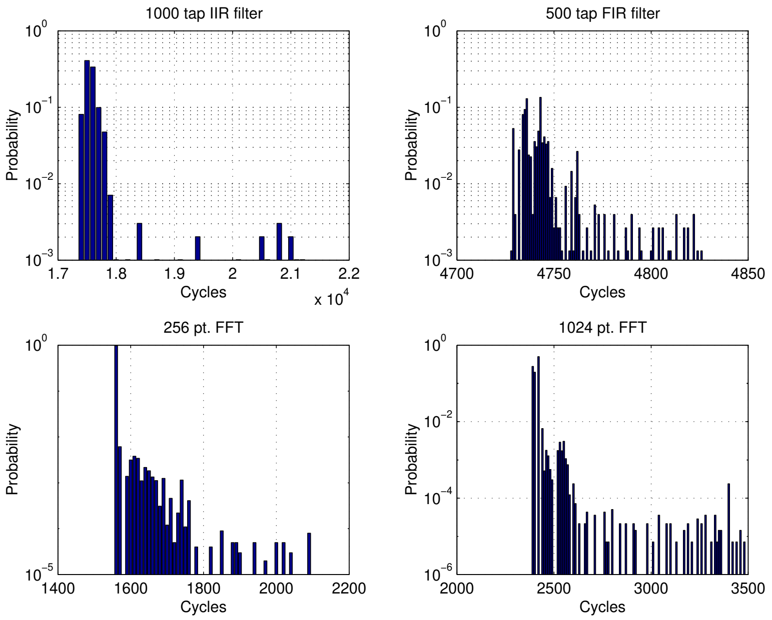


Figure 3-9: Histograms of the number of cycles required to produce a single output point for several different algorithms.

cases by simply setting the real-time bound to be 5% greater than the minimum.

The longer term variability, which occurs on the time scale of milliseconds or more , is primarily due to the operating system and requirements of other applications running concurrently. This variability is insensitive to the particular algorithm that is being run. This long term variation has little affect on systems that are able to occasionally buffer large amounts of data, and then process this data faster than real-time when the cycles are available. A packet data network is a good example of such a system. However, the long term variation has a significant affect on the performance of low-latency applications, such as real-time full duplex voice communication, or packets networks where the packet must be acknowledged in a short period of time.

As CPU and memory speed continue to improve, the system dependent sources of variability will decrease. System activity, such as interrupt processing takes a fixed number of instructions and memory accesses. These operations will get faster as the underlying technology improves, and the actual time required to service an interrupt will continue to decrease. Similarly, the system timer interval can be decreased without incurring a larger penalty, as measured in elapsed time, for a context switch. This will reduce the maximum amount of variation that an algorithm will be subject to.

However, as Moore's Law has provided double the transistor density every 18 months, CPU designers have used them to implement considerably more complex mechanisms in the processor, such as super-scalar architectures and branch prediction. These mechanism typically improve the average performance, but introduce variability into the system. It is likely that this trend will increase as processors get more complex. Fortunately, the variation introduced by these components only increases the worst case running time by a few percent. Furthermore, as processor speed increases, the worst case running time, and the time penalty incurred by this variation decrease. Looking to the future, it seems quite likely that the scope of real-time applications that can be implemented in a general purpose environment will continue to expand.

The proposed model for computation seems quite reasonable in light of the experimental results. The source of variability can be divided into two categories, frequent variations that cause variations in computation on the order of a few percent, and less frequent variations that cause larger variations. Furthermore, it the larger variations are due to the operating system, whereas the smaller variations are due to processor architecture. The fact that only the short term variation is dependent upon the algorithm being used is a key observation that greatly simplified the design of the system.

## 3.2 Review of Real-Time Definitions

Before presenting the the statistical real-time model, it is useful to review existing definitions of real-time systems. Since real-time constraints can apply to a wide range of applications, there are many different definitions that exist in the literature.

At a high level of abstraction, a real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred[2].

Another definition for a real-time system comes from the field of real-time language development, which states that a real-time system is "any information processing activity or system which has to respond to externally generated input stimuli within a finite and specifiable time delay" [Young, 1982]. The concept of external stimuli is too narrow a definition for our purposes. For example a transmitter may be sending data such as a repetitive beacon that does not result from an external stimulus, but nonetheless has a tight timing constraint. However, the fundamental notion that every action must have a finite and specifiable time bound is an important aspect.

More specific definitions of real-time exist for DSP systems. In a real-time DSP process, "the analyzed (input) and/or generated (output) samples (whether they are grouped together in large segments or processed individually) can be processed (or generated) continuously in the time it takes to input and/or output the same set of samples independent of the processing delay"[3].

This definition often is used for real-time audio and video, and it specifies that the throughput must be greater than the arrival rate of input samples. With a finite memory constraint added, this definition then also specifies the maximum latency of the system.

Real-time systems are traditionally divided into two categories: hard real-time and soft real-time [Jensen, 1997]. In a hard real-time system, all time critical functions have deadlines which must always be met in order for the system to function properly. Soft real-time

[2]Donald Gillies, University of British Columbia, in the comp.realtime FAQ: http://www.bookcase.com/library/faq/archive/realtime-computing/faq.html

[3]Robert Bristow-Johnson, Wave Mechanics, Inc., in the comp.realtime FAQ: http://www.bookcase.com/library/faq/archive/realtime-computing/faq.html

system are not well defined, they are generally thought of as real-time systems that are *not hard*. What this typically means is that the system can still function reasonably well even if deadlines are occasionally missed. Some tasks may have deadlines that aren't binary, but rather are multi-valued and effectively specify the actions' utility to the system as a function of the completion time. Clearly in this sense soft real-time is the most general and realistic case, of which hard real-time is an extreme special case.

A hard real-time time system imposes a hard deadline for each task, and provides a mechanism to insure that this deadline is met. A hard real-time solution often results in decreased resource utilization, since the bounds are pessimistic. The system designed in this thesis has no such mechanism, and is therefore a soft real-time system. A soft real-time application has timing constraints, but there is no guarantee that they will be met. It is not only signal processing applications that have timing constraints. An application that did not have a timing constraint at all would be exhibiting acceptable behavior if it took an unbounded amount of time to respond or complete a given task. In this sense all practical applications are soft real-time, and general purpose systems are soft real-time systems.

The critical issue is the strictness of the timing constraints. It is acceptable for a word processor to occasionally take a considerable amount of time, even seconds, to perform some tasks. Furthermore, it is also acceptable for the time to perform a given task to vary significantly. However, many signal processing applications require much tighter timing constraints, and can tolerate less variance.

What is needed is a framework for classifying soft real-time systems. A simple time bound on the response time of the operating system, or the computation time of an application is not sufficient. The classification must capture the notion that it is acceptable to occasionally take a very long time, if the average performance is good. The application level real-time mechanisms described in the next section provide such a framework.

## 3.3 Statistical Real-Time

This section describes an approach to implementing real-time signal processing applications on a general purpose processor without explicit real-time support in the operating system. We define the class of applications that need to be supported, and the mechanisms to handle the two classes of variability.

### 3.3.1 Class of Supported Applications

The system described in this dissertation was designed to support a class of real-time applications with the following characteristics:

- For each input value there is a finite bound on the time required to produce the output the corresponding output value.
- The only external events that the system must respond to are the arrival of input data and user initiated changes to the parameters or topology of the system.
- The application can tolerate jitter in both the input and output streams.

This class encompasses a wide range of applications which require the capture and processing of signal data in a timely manner. Examples of such applications include many radio receivers, medical equipment such as ultrasound and EKG, processing of radar, and monitoring equipment such as oscilloscopes and sensors.

Applications that fall outside of this class are those that must respond to additional external events in a timely manner. For example, any system in which there are any time sensitive inputs to the system that are not part of either the input stream(s) or specified by the user. Note that events that occur as part of any input stream can be dealt with, however if an external mechanism path, such as an interrupt, is required then the application falls outside of the defined class.

The ability to handle jitter is an application specific tradeoff with latency. If the input or output can be buffered, jitter can be absorbed at the expense of additional latency. A good example of this is the RealAudio System[4], which transmits real-time audio over the Internet by buffering several seconds worth of data at the client in order to absorb the jitter introduced by the network. Synchronous systems, in which the the input and output streams must be tightly synchronized, are limited in the use buffering to absorb jitter. Tight synchronization is common in many DSP and control applications. This thesis demonstrates that this tight synchronization is not necessary for many interesting applications.

### 3.3.2 Choosing Deadlines

By taking advantage of the ability to process data faster than real-time, jitter in the computation time for any one function can be absorbed. This provides a mechanism for dealing with the frequent, small scale variability.

Many algorithms can process data faster than required to meet the real-time bound, for example an audio amplifier module simply multiplies each incoming sample by a constant, which can be performed at a rate much faster than the 8 kHz audio rate. Resource unpredictability may result in the processing time occasionally exceeding the real-time rate, but the average processing rate can still be well below the real-time threshold. Thus there is a trade-off between higher average throughput and jitter in the computation time.

Consider the illustration of the processing chain for an AMPS cellular receiver, shown in figure 3-10. One method for enforcing real-time processing would be to set a timer for each processing module, and if the timer expires before the processing is finished a real-time deadline miss will occur. However, this is an unnecessarily tight constraint. Each module computes results, on average, faster than real-time, but may on occasion exceed this bound. However, if a single module exceeds its allotted computation time, this does not mean that the entire chain will miss the time limit for computing a result, since other modules are computing faster than required. This is typically the case, since the significant processing delays are caused by external events, such as interrupts and scheduler activity, which are not correlated with the processing itself.

The distribution of cycles required by a particular chain of modules cannot be determined by profiling each module independently and combining the results, since the distribution is

[4] http://www.realaudio.com

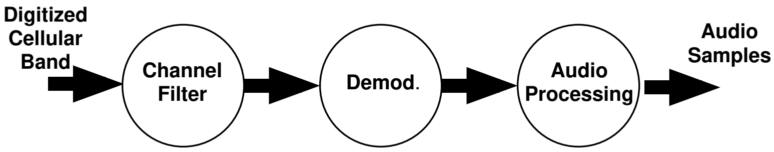


Figure 3-10: Processing modules for the AMPS cellular receiver.

affected by the other modules in the chain. For example, the state of the cache may differ dramatically depending upon what other modules are doing. The scheduler activity is another factor that affects the independence of the distributions. Assume the two modules together require less than 10 msec (the scheduler interval) to complete. Taken independently, there is a probability that each module could be swapped out by the scheduler. Taken together, if the first module gets swapped out, then the next scheduler interrupt will not occur for 10 msec. Thus given that a scheduler interrupt occurs during the running of the first module, the probability is very low that there will be a scheduler interrupt during the running of the second module[5]. Conversely, if the scheduler does not interrupt the first module, then the probability that the second gets interrupted increases, since the process has been running longer without being swapped out. However, treating the chain of modules as a single module, and characterizing it's variability appears to retain the separation between the short and long term variability.

### Dealing with Large Scale Variability

In order to deal with the larger variations, the concept of *statistical real-time performance* is introduced, in which an application is characterized by:

- The cumulative distribution of the number of cycles required to complete the task.
- A desired real-time bound.
- A specification of the action that is to be taken when the deadline is not met.

This is a form of soft real time performance, since deadlines can be missed without disastrous consequences. Rather than insuring real-time performance with the tight synchronous control over the processing that is typical of many DSP and digital hardware designs, this approach is statistical in nature. The approach requires that there are more than enough cycles available on average to perform the processing, but realizes that the actual number of cycles available over any given period of time varies as due to other demands on the system.

The probability that the task will be completed within the desired time bound can be computed from the cumulative distribution of cycles required by a given application, as illustrated in figure 3-11. This is possible since the statistics associated with the execution time are consistent. As the time bound for a particular application is increased, there is a higher probability of meeting that bound. On systems that have real-time schedulers

[5]A second interrupt could occur if there is excessive interrupt activity that causes most of the time slice to be used for interrupt processing.

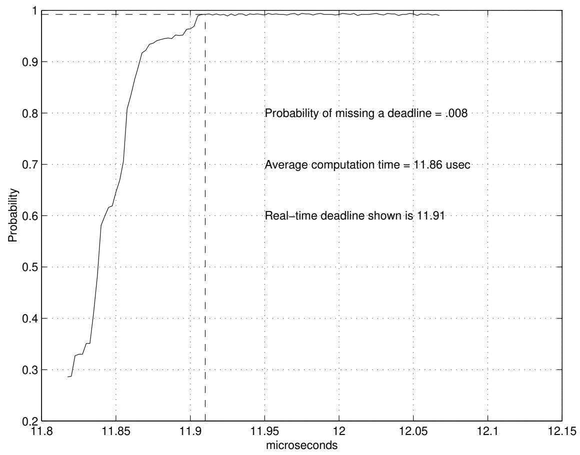


Figure 3-11: Calculation of the probability of meeting the real-time constraint from the cumulative distribution of the number of cycles required.

and bounds on the operating system services the probability can reach one under normal operation[6]. Note that if the task completes with a probability of one, then the system can provide hard real-time performance. If the probability of completion is less than one it is a soft real-time application. This framework captures a continuum that encompasses hard real-time applications on one end, and fairly time insensitive applications such as a word processor on the other. The probability of meeting the deadline provides a measure of how *hard* the real-time system is. Note that this definition does not incorporate any notion of how much a deadline is missed by. It is sometimes useful to know the maximum amount by which a deadline can be missed by, but in general a miss by the maximum amount is so rare that it does not significantly impact that application's performance. This simple framework captures the scope of applications that a given system is capable of supporting, which lends insight into the real-time capabilities of general purpose systems.

This classification is only useful if it is beneficial to have a real-time system that is *harder*, but not necessarily *hard*. The migration to the desktop of many traditionally real-time applications, such as audio and video processing, is an excellent example of a *harder* real-time system being useful. It is possible to run audio applications on older machines, but too much data would be dropped since the system cannot keep up with the processing demands. On newer faster systems, deadlines can still be missed, but the probability of such an occurance is so much lower that the application level performance is deemed acceptable.

### Deadline Misses

How hard the real-time system needs to be is application dependent. In order to assess how hard the system needs to be, a mechanism is needed to translate missed deadlines into a parameter that is directly related to the performance metric(s) of the application. This relationship is determined by the action taken when the deadline is not met. Possible

[6]If external events such as power failures are taken into account, then no system can truly meet the deadline with a probability of one.

actions include dropping data, using a partial estimate of the result, substituting a value for the result, continuing processing for an additional period of time, or saving the data to be processed at a time when there are spare cycles available. Dropping the data would be appropriate in applications such as a full-duplex real-time voice system, where missing a small amount of data may be more tolerable than increased latency. On the other hand, a packet data application may be able to deal with jitter quite well, so a more graceful failure mode could be chosen.

For example, consider the function for demodulating one bit from an FSK modulated signal. We implemented this function as part of a network interface. The probability that more than 10 $\mu$s was required for processing was measured to be less than 0.003. The measure of performance for the demodulator is the probability of a bit error, thus if we can translate the missed deadline into an effective bit error, we can directly calculate the increase in bit error rate caused by a real-time system with a given hardness factor. If we chose to stop the processing after 10 $\mu$s and make an arbitrary decision as to the value of the bit, this would correspond to an increase in the probability of a bit error of only 0.0015.

It is reasonable to assume that the errors due to deadline misses, which are generally caused by system or user activity are uncorrelated with the errors that normally occur in the channel. In the FSK example, it was assumed that a deadline miss only causes the failure to demodulate a single bit. Thus the additional resulting bit error rate can be simply added to the existing bit error rate of the channel. A larger deadline miss can cause a block error if the application cannot tolerate enough latency. This introduces a correlation into the errors seen by the end user. Fortunately, these types of deadline misses are rare, and block errors are common in wireless communications systems, often resulting from multipath fading or a temporarily hidden terminal. Many systems have mechanisms to combat block errors, for example the GSM system uses bit interleaving to reduce the correlation in the error of adjacent bits.

The missing of a deadline and the associated cleanup action have serious implications for the real-time performance of the system, but it is important to realize that all systems have an inherent error rate, and ideally these missed deadlines could be mapped into errors, and then insure that the resulting error rate does not significantly increase the error rate already present for that application. If this is achieved, the application level performance of the system should remain unchanged, thus the user should not experience a noticeable change in performance. The "cleanup" procedures are application dependent, the cleanup procedure for a missed deadline on a packet may throw away the rest of the packet, whereas an audio system cannot afford large block errors.

The worst case time requirement in the FSK demodulation example is nearly 210 $\mu$sec, which is slightly more than 20 times longer than the real-time bound. However, the median processing time is approximately 5 $\mu$sec. If the worst case did occur, the system could *catch up* with real-time in roughly 40 $\mu$sec on average, provided the system has enough buffering to hold the unprocessed data during that period. Thus, if the end application in this example can tolerate this amount of jitter, the system won't have to drop any data. This assumes that the trials that require a large amount of time don't occur very close to each other, which has experimentally been seen to be the case.

## 3.4 Summary

This chapter described a method for characterizing the variability in execution times on a general purpose processor. This method was used to derive a model of computation for a general purpose processors that divided the variability into two classes, small, frequent variations and large but infrequent variations. The remainder of the chapter discussed application level mechanisms designed to deal with these two classes of variability. These mechanisms were implemented in the SPECtRA programming environment, which is described in the next chapter.

# Chapter 4

# The SPECtRA Programming Environment

SPECtRA is a Signal Processing Environment for Continuous Real-time Applications designed to support software radio applications. It supports the implementation of real-time software radio applications in a variable computation environment. In addition, the environment provides for the efficient implementation of these systems, and allow for the design of algorithms and applications aimed at improving the processing performance.

Increased flexibility and improved performance are the two main goals that drive the software approach. The system must be flexible enough to adapt to a wide range of conditions, from changes in the channel environment to upgrades in the processing technology. The performance of the system is best defined in terms of application level performance. The performance parameters of the underlying communications system, such as bit error rate and throughput are constrained differently for each application. On a general purpose processor, the application and the radio are running on the same platform, which provides for tighter integration and optimization between the communications system and the applications

In order to realize the potential flexibility offered by an all software signal processing implementation, it is necessary to have a programming environment that is suited to modular real-time data intensive signal processing applications. Thus the design of the programming environment is a critical piece of the overall system. In order to manage the complexity of the system, the design goals included the following software engineering goals:

- **Simple Implementation.** The translation from an equation or algorithm to the actual code should involve as little work as possible. This reduces development time as well as simplifies the work required for debugging and upgrades.
- **Code Re-use.** The ability to re-use computationally intensive signal processing code not only reduces the work required to implement a given application, but also allows significant effort to be put into optimizing the routines, since they can be used by many different applications. Furthermore, the number of bugs will decrease, since the same functions do not have to be re-implemented many times,
- **Dynamic Modularity.** The system must allow for any portion of the processing to

be modified or replaced while the system is running.

- **Single Development and Run Time Environment.** The design of an environment that produces a single source code base that can be used for testing and verification as well as run time optimization.

The key design aspects are the data-pull model, the stream abstraction, and the partitioning of in-band and out-of-band functions. The design aspects are reflected in the SPECtRA architecture, which consists of three components, processing modules, connectors and an out-of-band script. Together, these components comprise an extremely flexible system that is capable of handling the data intensive signal processing associated with many wireless communications systems.

## 4.1 Data Pull Model

The SPECtRA programming environment employs a novel "data-pull" model, which was designed to overcome some of the limitations of a traditional data flow implementation. On the surface, the approach looks very much like data flow, as a graph of interconnected modules is defined to create a signal processing system. In a typical data flow implementation, the data is pushed from the source to the sink, in the data-pull model, however, the execution is driven by the data sink which requests data, as it is needed, by the upstream modules.

The data-pull approach has three significant advantages:

- Improved computational efficiency resulting from the ability to transparently implement lazy evaluation.
- Rapid and efficient response to changes in the processing requirements.
- Improved performance on platforms employing data caching.

Most signal processing and media toolkits utilize some form of a data flow model [Lindblad *et al.*, 1994] [McCanne *et al.*, 1997]. One of the defining characteristics characteristics of a data-flow system is that execution is triggered by the availability of data [Guernic *et al.*, 1986]. In a modular media processing systems, this means each individual module can begin execution as soon as data is available at it's input port. However, this can lead to extra work being performed and limit the dynamic flexibility of the system, limitations that are overcome by the data-pull approach.

At first glance, it might seem that the data-pull system, with it's lazy evaluation approach, might not be able to take advantage of a parallel processing architecture, since we have introduced a dependency on the result of the output of the final stage of the processing chain. The work presented in this dissertation does utilize multi-tasking for signal processing applications, but it does not use parallel processing architectures. However, in section 9.1.2 ongoing work is presented on a design for a parallel implementation of a data-pull system that retains many of the advantages of data-pull and incorporates some of the parallelization advantages of data flow.

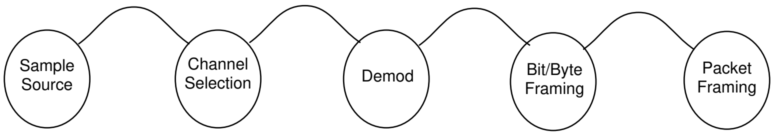


Figure 4-1: Block diagram of a software radio network application.

### 4.1.1 Computational Efficiency

Consider the high level diagram of a software radio system designed to receive network data packets shown in figure 4-1. The first three modules extract the digital data from the received analog waveform. In a data flow system, it is possible for each module to execute, whenever it has data available to work on. However, some of the data produced by the modules may never be needed. For example, the first task performed by the packet framing modules is to extract and examine the header of the packet. If the packet were not destined for the host machine the remainder of the packet is not needed, and the processing of a considerable amount of input data could be skipped. The remaining bytes of the packet translate to a much larger number of bytes of the wideband input data stream, which not only don't need to be processed, but don't have to be moved from main memory into the data cache. In a data flow system, the signal processing to extract the rest of the packet will be performed as soon as the data is available. The unnecessary work done on this data could have been applied to other functions, or used to reduce the power consumption. For example, the dynamic clock management features now appearing on some processors could be used to keep the clock speed low while only the packet headers are processed, and then increase the clock speed when the header indicates that the packet is destined for the host machine.

Since each module only requests the data it needs from the upstream module, only the data required is computed. For example, a down-sampling module might output every nth input sample, and in some cases it may not have been necessary to compute some or all of the intervening samples. In the pull model, only every nth sample is requested, if the intermediate samples are required for the computation they are evaluated, otherwise they are skipped. In this way the pull model can lead to transparent computational savings by eliminating computation that is not required for the desired result.

The data flow model cannot easily accommodate the elimination of computation that the lazy model naturally achieves. However, in many cases it is not known *a priori* what data needs to be skipped. For example, in the reception of data that has been segmented into smaller cells, the loss of a certain number of cells may render the entire sequence useless, so the remaining cells in the sequence can be ignored. This information is not known until after a certain number of cells have been processed, and the earlier stages may have already performed some processing that was unnecessary.

### 4.1.2 Efficient Adaptation

Another advantage of the data-pull approach is that the dynamic adaptability can be improved. In many cases the data itself contains information that can affect the processing of the signal. For example, the AMPS system utilizes in band FSK signalling to dynamically manage hand-offs and channel re-assignments. In this case, the processing of the wideband data is dependent on the result of processing the resulting data stream. In a data-flow system, it is possible to have already processed some of the wideband data that occurs after the change has been detected. In the data-pull approach, the data sink would request just enough data to be able to look for the presence of a control signal in the audio. If it is detected, it can immediately implement the required changes to the signal processing, and then request more data. In this way, the system can adapt very rapidly and efficiently to changes in the signal or environmental conditions.

The data-pull model can also rapidly adapt to changes in the processing topology. Whenever the structure of the system changes, the system requirements have to be recomputed. In a data-flow model, inserting a new module into the processing stream would require examining it's impact on the upstream modules, often resulting in new block sizes for computation. The beauty of the lazy model is that modules can be dynamically modified or replaced, and the system automatically produces just what is needed by the downstream modules.

### 4.1.3 Caching Benefits

There are many signal processing algorithms, such as the FFT, that do not display good locality of data reference, and therefore have poor performance on many cache based systems. Although there are many algorithms that display such behavior, poor locality is not a characteristic of signal processing systems as a whole. In a system, comprised of many modules, the output of any particular module is consumed by another, so there is data reuse of each output value computed. In a lazy evaluation system, values are not computed until they are needed. Thus the data that a module is operating on was the most recently computed quantity, and is very likely to be in the data cache. If possible, it is desirable to work on data sets that fit in the cache, however this is a system performance issue that has to be traded against other factors such as the function call overhead.

The caching benefits could be obtained in a data flow system by carefully examining, in advance, the requirements of each stage and designing the data flow accordingly, and then re-designing it any time the system topology changes. Again, the data-pull model allows transparent adaptation to these changes.

## 4.2 SPECtRA Architecture

A visual description of the programming environment is shown in Figure 4-2. The system is partitioned into in-band and out-of-band axes, in a manner similar to that used in the design of the VuSystem [Lindblad, 1994]. The in-band axis is where the temporally sensitive, computationally intensive work takes place. The out-of-band axis is used for control, configuration and inter-module communication. All of the code that is specific to a particular application is contained in the out-of-band section. This partitioning allows for maximal re-use of the computationally intensive in-band signal processing modules, since none of their functionality is specific to a particular application. The design of this programming environment incorporates the data-pull model, and the data is pulled down the pipe by the sink.

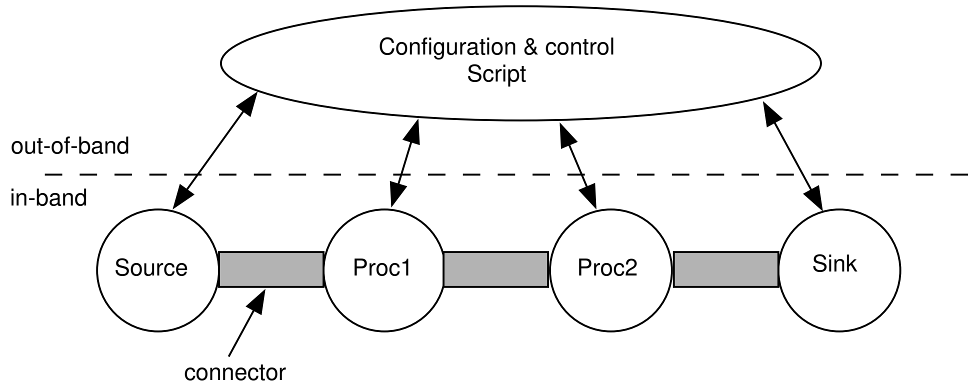


Figure 4-2: Graphical description of the programming environment.

### 4.2.1 In-Band Components

The in-band path is responsible for the actual signal processing and data manipulation tasks. The in-band path consists of two primary components: processing modules and connectors. The modules perform the signal processing and exchange data with other modules through the connectors. Any signal processing path starts with a source module, and ends with a sink module. Processing paths may overlap, for example the path from a source may split and end up at two different sinks.

The processing modules are objects that contain the code required to perform a particular signal processing task, as well as class variables and accessor functions to allow for configuration and monitoring by the control script. The partitioning of the system allows for the objects to be viewed as abstract entities, as in the *mash* toolkit [McCanne *et al.*, 1997] with some methods implemented in the out-of-band section when appropriate.

The connectors can be thought of as wires that carry signals from the output of one processing module to the input of one or more processing modules. Each output port has an output buffer associated with it, which allows the output stream to be multiplexed to several different output ports. The output buffers are automatically created with each output port.

The following rules govern the interconnection of the in-band components to form a processing system:

- A processing module must have one or more input ports and one or more output ports.
- Each connector must be connected to exactly one input port and exactly one output buffer.
- A module's output buffer must have one or more connectors connected to it.
- A module's input port must have exactly one connector connected to it.
- *Sinks* are specialized processing modules that must have one or more input ports and no output ports.

- *Sources* are specialized processing modules that must have one or more output ports and no input ports.
- Each processing system must have at least one source and at least one sink, and there must be a data path between the source and the sink.

### 4.2.2 Out-of-Band Components

The out-of-band path is responsible for the following tasks:

- Creating and modifying the topology of the system.
- Defining and executing the communication between processing modules that does not directly involve the data.
- Handling user interaction.
- Monitoring overall system performance.

The out-of-band system does not actually touch the data being processed. The only functions that do not fall in the domain of the out-of-band processing are those that are deal with the actual signal processing and buffering of the resulting data. The out-of-band path is realized as a script, as described in section 4.4.3. The script makes calls on the processing module objects to modify processing parameters, and the modules can also initiate calls back to the script. Modification to the modules is performed through the manipulation of state variables in both the modules and the out-of-band script. If a class variable is to be modified by the out-of-band script, it must have reader and writer functions. This simplifies the abstraction to other parts of the system, and allows the inner workings of the module to be modified without changing any other pieces of code that use that particular module.

### 4.2.3 Execution Stages

The running of the system is sub-divided into three separate stages:

- **Setup Time** Processing modules are created, initialized, and the topology of the system is defined.
- **Configuration Time** Connectors are created and the topology dependent parameters are initialized.
- **Run Time** The signal processing is performed.

Note that *Setup Time* and *Configuration Time* do not only occur at the beginning of execution. Any time the connector parameters, such as the sampling rate, are modified it is necessary to recompute the topology dependent parameters. This is accomplished by initiating a *Configuration* phase. If processing modules are inserted and/or removed from the system, then it is necessary to enter a *Setup* phase to create and initialize the new modules, and then initiate a *Configuration* phase since the topology has been modified. *Run Time* is when the actual signal processing takes place, and commences at the termination of a *Configuration* phase.

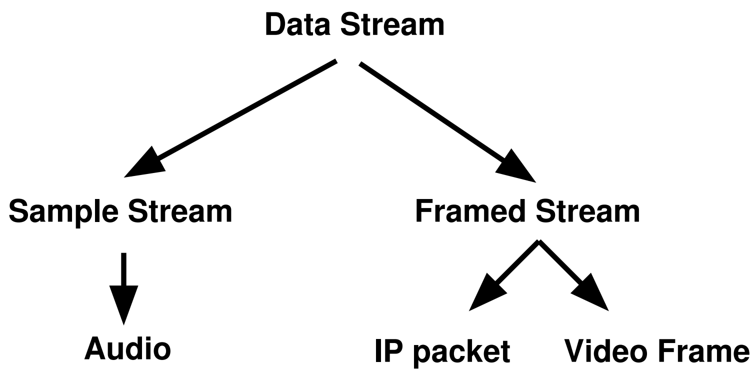


Figure 4-3: SPECtRA stream type hierarchy, with a few representative derived types.

## 4.3 The Stream Abstraction

In order to realize software implementation of all of the processing required to transform between the samples streams produced by A/D converters and the application data the system must handle a wide variety of data types. The output of an A/D converter (or input to a D/A converter) is most naturally represented as a continuous stream of samples, which is parameterized by a sampling rate and resolution. In digital communications systems these streams are processed to produce a sequence of symbols. These symbols are often aggregated into frames with error detection and correction information. Once the data reaches the application, it may be represented as network packets, video frames or even converted back to a sample stream as in the case of audio transmission over the Internet.

Although this may seem like a disparate collection of data types, the SPECtRA stream abstraction unifies them into one object hierarchy. The fundamental observation is that the input or output of any processing function can be represented as a stream of data objects. The A/D samples naturally form a stream, video and network data can be viewed as streams of frames and packets respectively. For purposes of discussion, these objects contain the data as well as a time stamp for each data unit[1]. An outline of the stream hierarchy, with a few representative derived types, is depicted in figure 4-3.

For many signal processing applications, an infinite stream is a natural I/O model. Functions such as filters operate on a continuous stream of input samples and produce a continuous stream of output samples. Many systems use finite buffers to pass data between one processing module and the next, but this leads to considerable extra code to take care of end conditions and make the sequence of buffers appear to be a seamless infinite stream of data. Furthermore, since the code required to achieve this effect is dependent on the processing function, this extra code must be incorporated into each processing module. The stream abstraction is implemented in the connectors, relieving the signal processing programmer of the burden of managing the data stream. From a processing module's perspective, these streams can be thought of as infinite buffers. This abstraction facilitates the translation from a mathematical representation of a signal processing algorithm to actual signal pro-

[1]For performance reasons the implementation of sample streams does not actually contain a timestamp, since the timing information is implicit in these regularly spaced streams.

cessing code. This has resulted in a very small source code base for the system, and very short development times.

### 4.3.1 Stream Interfaces

There are two interfaces to a stream, one for input and one for output. The input interface is used when a module needs to access data from one of its input streams, and the output interface is used when a module writes data to one of its output streams.

A stream is parametrized by the data object handled by that particular instance, which is denoted by **T**. The input interface supports the following methods:

- **T inputRead(int n)** Read the value associated with the index n units from the current time value. Note that to read past values n is a negative number, which is consistent with the manner in which DSP functions are often expressed. It is legal to read *future* input values, which does not actually mean that it is a non-causal system, but is a reflection of the fact that there is buffering built into the system.
- **void incInput(int n)** Increment the current time value of all input streams by n.

The output interface supports the following method:

- **void outputWrite(const T x)** Writes the value of x to the output buffer and increments the output stream current time by the amount corresponding to the output unit.

It is sometime desirable to read past output values, e.g. to implement recursive functions such as IIR filters. A design decision was made that prohibits modules from storing past output values internally. This permits dynamic replacement of processing modules without having to worry about losing state that was stored in the replaced module. The past output values could have been accessed by implementing an output read function on the output stream, but a second design decision was made that requires a stream to be connected from the output back to an input port of the module to read these values. This preserves a clean abstraction for the output interface, and greatly simplified the system implementation.

The design decision to use streams as the interface to all processing modules also provided a more natural interface to signal processing applications, which greatly reduced the amount of code required to implement processing modules.

## 4.4 System Components

### 4.4.1 Processing Modules

The signal processing modules are the heart of the SPECtRA system. The system has been designed around making it easy to implement these modules and facilitating reuse of existing modules in future applications. Signal processing modules support the following class methods and variables:

Class Methods:

- **virtual void work(int n)** This is the procedure that actually performs the signal processing. It is the only procedure that must be implemented by each signal processing module.
- **virtual void initialize()** This procedure is used to initialize class variables that depend upon the topology of the system. Class variables that do not depend upon topology should be initialized in the module's constructor. The initialize procedure is run at configuration time. Implementation of this method is optional.

Class Variables:

- **outputSize** Indicates the minimum number of output samples that the module can produce at one time. Each request for samples from the system is guaranteed to be an integral multiple of this value. The default value is 1.
- **history** Indicates the minimum number of past samples required for the computation of each block of samples as determined by the outputSize variable. For example, an $N$ tap FIR filter, with output Size of 1, would have a history of $N$. The default value is 0. The system guarantees that at least *history* past values of the input are available at any time.
- **inputConn** A pointer to the array of connectors that are attached to the input ports of the module.
- **outBuffer** A pointer the array of buffers that are attached to the output ports of the module.

The history and outputSize parameters are used by the connectors to determine the appropriate buffer sizes, as explained in section 4.4.2. An example of function requiring no past values (i.e. a history of 0) would be an amplifier with *outputSize* set to 1. This module simply multiplies each input value by a predetermined value and writes it to the output stream.

A generic processing module has a connection to an arbitrary number of input and output connectors. The modules are parameterized by their input and output data types. This eliminates the duplication of code for identical functions that operate on multiple data types. For example the same filter module could be used to process data expressed in signed or unsigned fixed-point or floating point representations.

#### Example Processing Module

In order to provide a more detailed description of the implementation details for a signal processing module, consider the example of implementing lowpass FIR filtering, as expressed by equation 4.1.

$$y[n] = \sum_{i=0}^{N-1} h[i]\ x[n-i] \tag{4.1}$$

Figure 4-4 shows the declaration for a lowpass FIR filter signal processing module. Class

```
template<class iType,class oType>
class VrRealFIRfilter : public VrSigProc<iType,oType> {
protected:
  int numTaps;
  float* taps;
  float cutoff;
  float gain;
  void computeFilter();

public:
  virtual void work(int n);
  virtual void initialize();

  int getNumTaps() {return numTaps;}
  void setNumTaps(float n) {numTaps = n; computeFilter();}

  float* getTaps() {return taps;}
  void setTaps(float* t) {taps = t;computeFilter()};

  float getCutoff() {return cutoff;}
  void setNumTaps(float c) {cutoff = c;computeFilter();}

  float getGain() {return gain;}
  void setGain(float g) {gain = g;}

  VrRealFIRfilter(float c,int t, float g);
};
```

Figure 4-4: Declaration for FIR filter.

variables are declared that hold all of the information necessary to create the filter (number of taps, cutoff frequency and gain). Another class variable is declared to hold the filter taps computed from these parameters. The filter tap calculation depends upon the sampling frequency, which is not known when the object is instantiated, but only when the module is connected to another. Therefore, the calculation of the filter taps must take place in the initialize procedure, not the constructor.

This FIR filter implementation automatically re-computes the filter taps whenever any of the key variables are modified. Thus when writing an out-of-band script it is not necessary to know what parameters would require the filter to be re-computed. Furthermore, the filter tap calculation routine can be changed without changing any of the calls to the filter module.

The work procedure shown in figure 4-5 is where the actual processing takes place. The implementation follows in a straightforward manner from the equation describing the function. The call to `inputRead` retrieves input values relative to the current time, and the call to `outputWrite` writes a single output value and increments the current time value of the output stream. These methods provide a clean abstraction, but their implementation

```
template<class iType,class oType> void
VrFIRfilter<iType,oType>::work(int n)
{
  float result;
  for (int i=0; i < n; i++, incInput(1), result=0) {
    for (int j=0; j < numTaps; j++)
      result += taps[j] *  inputRead(j-numTaps);
    outputWrite((oType)result);
  }
}
```

Figure 4-5: FIR filter work procedure implementation.

can lead to considerable system overhead in some circumstances. For efficiency reasons a second set of interfaces that allows access to blocks of data was also implemented. These interfaces require a modification to the pure lazy evaluation approach, which is described in section 4.4.2.

Note that the module is responsible for updating the current time value of the input stream explicitly with a call to `incInput(int n)`, since it may involve updates that are not synchronized to the output stream. A call to `incInput(int n)` simultaneously updates all of the inputs to the module. This keeps the system synchronized, and facilitates the implementation of systems that employ feedback.

### 4.4.2 Connectors

The connectors implement the stream abstraction and buffer data between signal processing modules. The appearance of an infinite stream is realized through the use of a circular buffer, as illustrated in figure 4-6. The read and write pointers represent the values that the respective downstream and upstream modules are currently accessing. Since the buffers act like wires in transporting the signal from one module to another, there can be multiple read pointers, one for each input that the module's output is connected to. Only a single read pointer is shown here, for clarity.

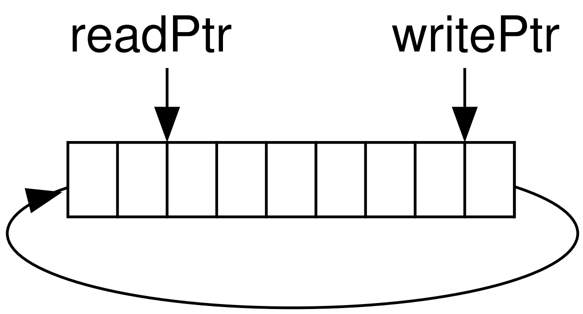


Figure 4-6: Circular buffer and pointer used to create the appearance of an infinite stream. The size of the buffer is determined using the **history** and **outputSize** parameters of the modules that the connector is attached to.

When the `inputRead()` method is called on a connector by an upstream module, the connector returns the requested data if it is present, i.e. if the value had already been computed. If it is not present, it then generates a call to the work procedure of its upstream module to generate more data.

A call to the `outputWrite()` method results in the data being written into the location pointed to by the write pointer, and then the write pointer being incremented by one unit.

When the `incReadPtr(int n)` method is called, the connector increments the read pointer by `n` units. This may cause the read pointer to be incremented past the write pointer, indicating that the data now pointed to by the read pointer is not valid. This condition is noted, but new data is not automatically generated, since it is not know whether it will actually be needed. New data is generated lazily when it is actually read.

It may be the case that some data is never read, and ideally it should then never be generated. This could be accomplished with a connector that kept a status bit for each value, and generated only the values needed. This approach supports fully lazy evaluation with random data access, but adds significant overhead to the system. The status bits would increase memory usage and reduce the effective cache size for the signal processing data. The checking of this bit on each read would add significant overhead to each data access.

In order to balance the advantages of lazy evaluation against the overhead incurred, a partially lazy system was implemented. As noted above, when the read pointer passes the write pointer, a condition indicating invalid data is set. When the next read occurs, a call to the work procedure is made to generate a number of output values equal to the `history` of the downstream module. The system convention is that any given module only needs `history` past values in order to compute its next set of output values.

This approach still allows for lazy evaluation on a block basis and can achieve the associated computational savings. If the read pointer is incremented past the write pointer by a number $n$ that is greater than `history`, then a number of values equal to $n - \texttt{history}$ are not computed. By incrementing the read pointer by an amount greater than `history`, the upstream module is effectively saying that there are certain values that it does not care about, and can be ignored. A sub-sampler is a good example of such a module. A straight forward implementation of a sub sampler is $outputWrite(inputRead(M * n))$ where $M$ is a positive integer. In this case, `history` would be set to 0 (no past values are needed), and the input read pointer would be incremented by $M$ after each computation, allowing the system to save the cost of computing the intervening $M - 1$ values.

Potentially even more important are the savings realized further upstream as a result of the skipping of the intervening values. Skipping of any values requires the upstream modules to increment their current time values to reflect the new state of the system, which is likely to result in the skipping of some computation in those modules as well. In a wideband receiver, it is often the case that the input data rate is much higher than the output data rate. For example, an AMPS cellular receiver might digitize the 12 MHz band at 33 MSPS, but the resulting audio signal requires only an 8 kHz sampling rate. Thus skipping even a single audio sample will often result in eliminating computation for a very large number of input samples.

In order to retain the ability to make dynamic changes to the system, a method was implemented to re-synchronize the time stamps of the connectors. This causes samples that were computed, but not yet asked for, to be discarded, so that parameters changes can be made at the same time across all modules.

Note that functions that rely on past output values for computation of future output values do not allow for the skipping of data, since all past output values must be computed in order to know the current output value. For low data rate applications, such as audio processing, this does not place much of a burden on the system. However, for high data rate applications such as channel selection, this can be a significant performance penalty.

### Buffering

The data buffer, although modeled as an infinite buffer, is implemented as a circular buffer. The circular buffer must be large enough to hold all of the values needed by the downstream module to compute its next output, which is specified by the `history` variable of the downstream module. It must also be large enough to hold the minimum number of data points that are generated by each call to the upstream module, specified by the `outputSize` variable of the upstream module. Given these two constraints, the buffer size should be kept as small as possible, to maximize the performance improvements associated with caching.

Since access to memory is a very time consuming operation, it is tempting to collapse successive processing modules into one processing step without buffering via analysis at configuration time. This kind of transformation is performed by many dynamic code generation systems [Engler, 1996]. If an output value were used only once by the system, storing it in memory would clearly be a waste of time. However, in systems where the same output value is used by a downstream module multiple times, the cost of re-computing this value may be more expensive than retrieving it from the cache or a register. Consider the quadrature demodulator shown in Eq. 4.2. In this case, computing a value of $y[n]$ without buffering of the input data requires the re-computation of the value represented by $x[n-1]$. Alternatively the cost of computation could be saved at the expense of writing and reading the value into memory.

$$y[n] = arctan\left(x[n] * x^*[n-1]\right) \tag{4.2}$$

The following example illustrates this tradeoff. The system analyzed contains a 200 complex tap FIR filter followed by the quadrature demodulator described by eq. 4.2. The performance was evaluated for two separate cases, in the first case the functions were in separate modules with a memory buffer in between, and in the second case the functions were collapsed into a single module with no intermediate storage.

In each case, the number of cycles required for the computation of each output of the demodulator was recorded. The cumulative distribution of the number of cycles required is shown in figure 4-7. The vertical line on the graph indicates the real-time constraint. If there were no uncertainty in the system, the cumulative distribution would be a step function. The uncertainty in this system is captured by the shapes of the curves. 75% of the time, both implementations run much faster than real-time. Above this level, the behavior diverges. The difference in the two systems is that the separate module implementation relies more

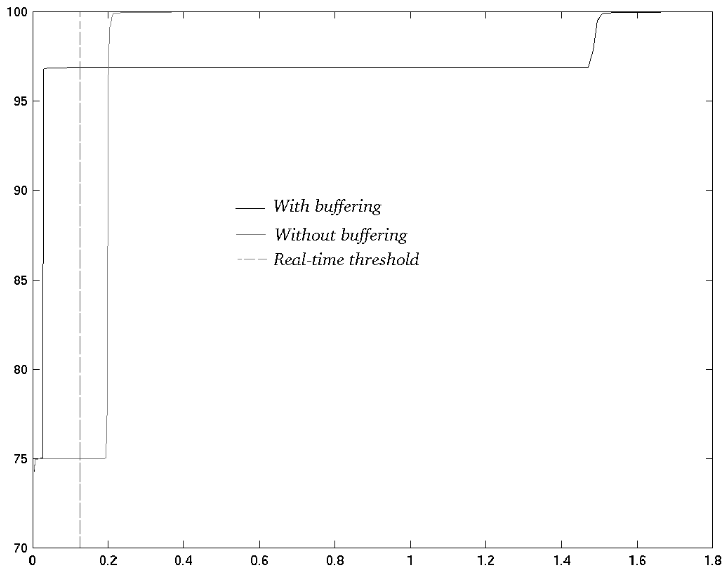


Figure 4-7: Cumulative distribution of the number of cycles required to calculate each block of samples for the combination of a 200 tap FIR filter and a quadrature demodulator. Results are shown for two implementations, one involving buffering, and the other without. The dotted vertical line is the real-time threshold for this function.

heavily on memory, while the collapsed implementation relies more on computation. Both must access memory to obtain the input values to the filter. The uncertainty introduced by the cache causes the larger variations in the distribution for the separate modules. The penalty for relying more heavily on memory is that there will be occasions where the access time is very long, due to cache misses. However, caching provides a tremendous benefit when data is present in the cache, and this is evident from the intermediate region where the separate modules outperform the monolithic module.

The example is a good illustration of the trade-off between optimizing for the worst case and the average case. Collapsing the modules is the correct strategy for optimizing for the worst case. However, if occasional deadline misses can be tolerated, the performance can be improved for over 95% of the trials by buffering between the modules. The spare cycles resulting from the faster than real-time computation can be applied to modules that may miss their individual deadlines.

### 4.4.3 Out-Of-Band Script

To demonstrate the flexibility of the in-band/out-of-band partitioning, two separate implementations of the out-of-band system were written, one in C++ and the other in java. The C++ version provided excellent computational performance as well as a compatible interface to the C++ processing code. For the java version a set of wrapper classes using the java native interface were written to provide a java program with control of the C++ processing modules. Thus the actual class was implemented partially in C++ and partially in java.

The java system also enabled multi-threaded control of the processing modules. Although the java version was considerably slower, it did not significantly affect the performance of the system since the java code was not performing the temporally sensitive computations. However, the overhead associated with locks to facilitate multi-threading was prohibitively expensive, and were not used in practice. Current work on modifications to the system to support multi-threading are discussed in chapter 9.

An example C++ out-of-band program is shown in figure 4-8. This program creates a system for receiving data packets that were transmitted using FSK modulation and frequency hopping for multiple access. The first section instantiates each of the modules that will be needed for the signal processing. The second section defines the topology by connecting the modules.

Prior to starting the signal processing system the *configure* method must be run on the data sink module (i.e. the module at which the processing terminates) to initiate the configuration phase. This procedure causes the initialization procedure to be run in all of the modules that have been connected, allowing them to initialize any parameters that are topology dependent, e.g., those that depend upon the input sampling rate. Finally, the system is started by running the start method on the sink. The argument to the start function indicates the number of seconds (represented as a floating point number) that the system is to run for. This time should be set according to the block size that is acceptable for the particular application. The program terminates if a condition is set in the run time loop or a termination event, such as running out of data in a file source, occurs.

## 4.5 Performance Overhead

The programming environment trades overhead costs for gains in flexibility and ease of implementation. The overhead imposed by the SPECtRA system is concentrated in the connectors and buffers. These components facilitate modularity and simplify programming by abstracting buffer maintenance functions. An important programming convention that simplifies connector management is that input streams are read only and output streams are write only, as illustrated in figure 4-9.

For read operations on input streams, it is necessary to check the bounds on the circular buffer to see if the the required values have been computed and if they have not, to initiate a call to the upstream module to compute the values. These operations must be performed once for each read, regardless of how many values are being read. These operations require between 10 and 20 cycles, depending upon the state of the buffer. Note that this reflects only the overhead associated with the programming environment, and does not include the cost of actually reading the value from memory. The connector overhead for reading is a function of the number of read calls made, not the number of values read. As will be shown in the next section, many of the functions in a typical operation operate on high sample rate streams, and it is not unusual to read buffers with sizes between 100 and 1000 items at a time. In these situations, the 20 cycles overhead is amortized over all of the items.

For write operations, the situation is simpler since only one module can write to a given connector. There are two types of write operations implemented in the connectors, one for writing single values and another for writing complete blocks. The single value write incurs

```
int main(void) {

  int cTaps = 40;
  int cpuRate = 530000000;
  int gupRate = 33000000;
  int CFIRdecimate = 825;
  int quadRate = gupRate / CFIRdecimate;
  int RFIRdecimate = 5;
  int audioRate = quadRate / RFIRdecimate;

  //
  // SETUP TIME
  //
  // Create Modules
  VrGuppiSource<char>* source = new VrGuppiSource<char>(200,gupRate);

  VrComplexFIRfilter<char>* channel_filter =
    new VrComplexFIRfilter<char>(CFIRdecimate,cTaps,1.0,2.0);

  VrQuadratureDemod<float>* demod = new  VrQuadratureDemod<float>(0.0);

  VrRealFIRfilter<float,short>* if_filter =
                new VrRealFIRfilter<float,short>(RFIRdecimate,4000.0,20,1.0);

  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);

  //
  // CONFIGURATION TIME
  //
  sink->configure();

  //
  // RUN TIME
  //
  while (1) {
    start(0.3};
    // Can add a check for termination conditions here
  }
}
```

Figure 4-8: Example out-of-band program

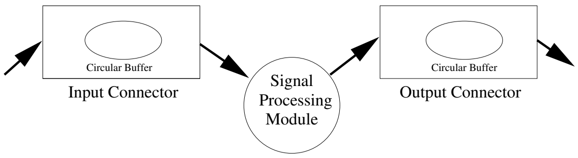


Figure 4-9: Illustration of the connector storage.

the overhead each time, and the block write trades the overhead for the cost of writing values into a temporary buffer.

The connector is constructed so that the current write pointer is always pointing at the next location to be written. The single value write operation involves retrieving the write pointer values prior to writing the value and then incrementing the write pointer after writing the value. The check on the bounds of the circular buffer is done during the increment operation. If the pointer has reached the end of the buffer, then the circular buffer management function is invoked. This copies a pre-determined portion of the end of the buffer onto the beginning of the buffer, and both the read and write pointers are moved to the appropriate positions at the beginning of the buffer to maintain continuity. The initialization procedure appropriately sizes these buffers so that this operation only happens once for each output block that the sink produces.

The cost of the block write involves first writing each value to a temporary buffer, then copying the block into the connector. The block copy was facilitated by the use of the *memcpy* function, which groups the copy operations into sizes that maximize the memory bus utilization.

The overhead associated with writing to connectors varies considerably. For large blocks writes (e.g. 400 items), typical values are a few cycles (1-5) per item written. On the opposite end of the scale, when the buffer management function is invoked, typical values are 100-200 cycles, even if only a single item is written to the connector.

# Chapter 5

# Software Radio Layering Model

The previous chapters have demonstrated that it is possible to acquire the data required for wireless communications applications and place it in memory where it can be processed by application level software. This capability presents an opportunity to design communications systems that can dynamically modify any aspect of their signal processing to interact with different systems and/or adapt to changing conditions. For example, a cellular base station could tailor its channel allocation and modulation scheme based on traffic and environmental conditions, and then indicate to each mobile unit what kind of radio to compile. However, this requires a well defined way of describing a communication system, so that the necessary changes can be indicated to the radio at the other end of the channel.

Radio systems have traditionally been divided into several separate stages, such as the RF, IF and baseband processing stage. This stages were defined primarily to facilitate hardware implementation. This chapter describes a software framework for specifying the signal processing requirements for a wireless communication system.

The software radio architecture presented in this thesis consists of several well-defined processing layers, which can be used to completely specify a wireless communications system. The layering presented in section 5.1 is a refinement of the OSI layering model [Tanenbaum, 1988]. Our layer model is a subdivision of the traditional Physical layer. The signal processing performed by this layer can be naturally subdivided into a finer grain model, but has traditionally been lumped into one layer because of its implementation in dedicated hardware. For our purposes, however, this is too coarse. To interoperate with different networks, it may only be necessary to change small parts of the existing layers. For example, two different systems may employ the same modulation and coding but use different multiple access protocols, or a given system may only need to change the type of coding to dynamically adapt to changing channel conditions. To facilitate this flexibility, we would like to create new communication systems by simply combining existing functional modules, rather than by writing a new piece of software that encompasses all of the functions in the link and/or physical layers for each network interface.

While layering provides an excellent framework for modularity and specification, it can lead to an inefficient implementation [Clark and Tennenhouse, 1990]. Since many software radio applications involve intensive data manipulation functions, the overhead of a layered implementation can be quite significant. The specification and implementation of radio

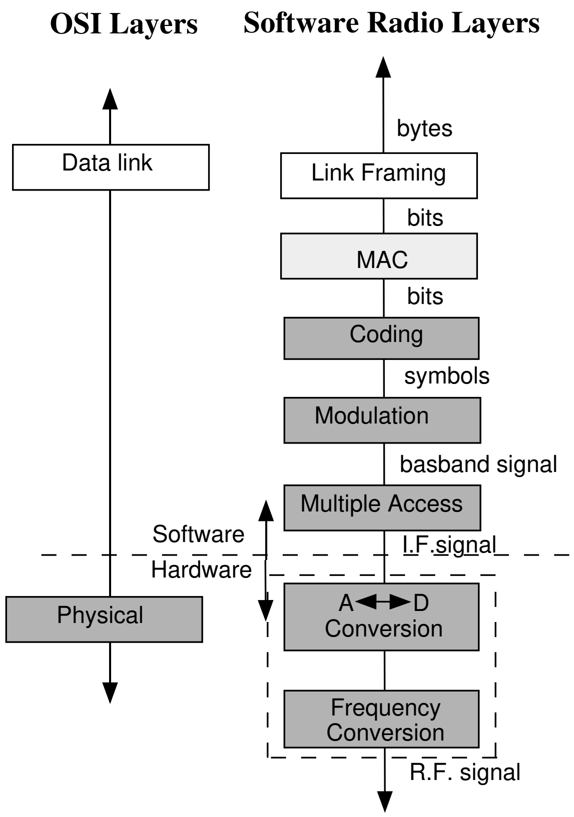


Figure 5-1: The Software Radio Layering Model shifts many physical layer functions into software.

systems are treated separately. The engineering approach has been to group layers together where necessary, but to insure that the grouping still uses the layered interfaces at its edges. The penalty here is that the ability to re-use software modules is at a coarser granularity when these grouped layers are implemented. This approach to dealing with the tradeoff between modularity and efficiency is discussed in section 5.2.

## 5.1 Processing Layers

The definition of new layers is not something to be done lightly. Our layers were defined according to the design principles of the OSI model [Tanenbaum, 1988], in particular the principle that the number of layers should be large enough that distinct functions need not be thrown together in the same layer out of necessity, and small enough that the architecture does not become unwieldy. The network layering model was revisited in the light of software signal processing that allows modification of functions that were traditionally placed in the same layer since it was necessary to implement them in dedicated hardware.

Figure 5-1 illustrates the layering model, and its relation to the OSI model. The function of the link layer is to take the raw transmission facility and transform it into a line that appears to be free of transmission errors to the network layer. Examples of link layer protocols include HDLS and X.25. The traditional link layer is preserved in this model.

The media access control (MAC) functions are often postulated as a layer that sits between the physical and link layers [Tanenbaum, 1988]. The primary MAC functions are mediating shared medium access and collision avoidance. There are many systems which use the same link layer protocols with different MAC protocols. For a software re-use point of view, it makes sense to break the MAC functions out as a separate layer for the software radio model.

The remaining layers in figure 5-1 comprise the functions that are traditionally lumped into the physical layer, which is concerned with transmitting bits over a communication channel. In a communications system, a signal goes through several transformations, and the intermediate representations form natural boundaries for subdividing the processing. These intermediate forms are:

- Bits
- Symbols
- Base band
- I.F.
- R.F.

The coding sublayer contains all of the coding necessary to transform between bits and symbols. This layer can contain both channel coding and line coding functions. A channel code is designed specifically for one of three purposes: error detection, error correction or error prevention [Lee and Messerschmitt, 1994]. They are used to reduce errors at the bit level, but the physical layer does not have to guarantee error free transmission. Spectrum control over the physical layer can be achieved through the use of line coding. Unlike channel codes, lines codes are not concerned with errors, but rather controlling the statistics of the data symbols, such as the removal of baseline drift or undesirable correlations in the symbol stream. The desired parameters are determined by physical characteristics of the transmission medium.

The modulation sublayer is concerned with the transformation between symbols and signals. This not only includes traditional modulation functions such as QAM, but also some types of coding and channel equalization functions. Any coding that is required for a receiver to reach a hard symbol decision is contained in this layer. It is tempting to define equalization as its own layer, but this would not be appropriate for two reasons. First, it would violate the layering principle that layer $n$ on one machine *carries on a conversation* with layer $n$ on another machine. In general, equalization is concerned with correcting for effects imposed by the channel, not by a corresponding function on the transmitter, so there is no comparable layer on the transmission side with which to communicate. The second reason is that equalizers are an integral part of the transformation between signals and symbols, and therefore should be part of the same functional layer as modulation techniques.

Finally, we have the multiple-access sublayer, which includes techniques such as TDMA and FDMA. Note that this multiple access layer performs a very different function from the MAC layer, which may also involve a multiple access technique. To illustrate this difference consider the IEEE 802.11 wireless networking standard [IEEE, 1997]. The MAC layer in 802.11 is CSMA, which provides shared access among all of the users of the network. The physical layer for 802.11 specifies two different multiple access techniques: frequency hopping and direct sequence spread spectrum. These multiple access techniques do not provide multiple access between users of a particular network, but for the sharing of the spectrum between different networks. Two different 802.11 networks can coexist in the same spectrum by using different spreading codes. The users of each network are not aware of the existence of the other network. In many cases, the other users of the band are not even data networks, but systems such as cordless phones, and wireless microphones. The physical layer multiple access provides isolation from all other systems using the band. When the physical medium is not shared by more than one network, as is the case with ethernet, there is no need for a physical layer multiple access technique. The choice of multiple access technique is usually independent of the modulation technique, and therefore should occupy its own layer.

In general a given system may contain only a subset of the layers. However, one could think of such a system as containing all of the layers, with null functions in some of the layers that do not manipulate the data in any way. This model provides a clean way of thinking about a system design.

### 5.1.1 Interfaces

One of the goals of layering is to minimize the information flow across the layer boundaries. This is enforced by the implementation of clean interfaces. The layering model uses a stream as the interface between layers. There are two basic classes of streams, synchronous and asynchronous. A synchronous stream consists of a sequence of regularly spaced objects of a certain type, for example, the streams generated by A/D converters. An asynchronous stream consists of a sequence of objects, each with its own time stamp. The only relation between the time stamps of consecutive samples is that they must be strictly increasing. A sequence of network packets are a good example of an asynchronous stream, since their arrival is typically irregular. There are five different object types that exist at the interfaces of the software portion of the layered structure, as described above.

The data type required for each interface is indicated on the right-hand side of figure 5-1. The interface data structure also contains parameters relevant to the data type, such as sampling frequency and bits per sample in the case of digital signals. The data types handled by higher layers, such as packets and frames can also be handled by this framework, as they are simply different data types.

## 5.2 Integrated Layer Processing

A layered architecture provides functional modularity, but often at the cost of efficiency in the implementation. The approach used in this thesis is to use the layered architecture as a design tool, but to separate this model from the engineering of applications, where integrating layers can provide significant performance gains [Clark and Tennenhouse, 1990],[Abbott and Peterson, 1993]. However, we do wish to maintain some of the modularity of the layered model in the implementation, so that a given wireless application can be dynamically modified by changing only a small amount of code.

In many applications, the processing involved is quite intensive, and it is often necessary to combine layers to achieve the desired latency or throughput characteristics. In particular, the layers involved with the processing of discrete signals involve many load and store operations because their data sets are typically quite large. For example, the waveform associated with a single bit in the FSK example described in section 7.3 requires a buffer of size $BitPeriod * SamplingFrequency \approx 16$ samples, and each sample requires two bytes of storage. A typical IP packet containing an ICMP packet generated by the "ping" application is 64 bytes. By the time each of these bits are framed and then modulated up to the IF frequency, the waveform requires over 20K bytes of storage. In order to balance the tradeoff between flexibility and performance a few guidelines for implementing integrated layer processing have been developed:

- Combine layers only if necessary to meet performance requirements, and then start by combining the layers closest to the IF signal, as these will provide the largest performance gains,
- Any combination of layers must still use the defined interfaces at the edges,
- Do not combine across OSI layer boundaries.

For modularity, it is desirable to leave as many layers separate as possible, so it makes sense to start by integrating the layers that require the most processing time. Using valid interfaces still allows for modularity, albeit at a coarser level, and by preserving the OSI interfaces, the ability to interoperate with other software or hardware systems that implement these layers is left open. The different priorities for specification and implementation are reflected by the fact that the implementation environment design, presented in chapter 4 utilizes an architectural design that differs from the layered model, but supports the layered model and allows for integrating layers.

## 5.3 Example Applications

Ultimately the software layering model will serve as a mechanism for implementing digital communications systems that can dynamically alter any aspect of their signal processing to provide better application level performance. In order to realize such a system, considerable research on characterizing the tradeoffs between performance and computation will have to be done along with the development of algorithms for dynamically modifying the structure of the processing system. However, the viability of the model can be assessed by examining how existing systems map into the model, and how changes to some aspects of these systems could be expressed.

The following two sections demonstrate the utility of the software radio layering model by using it to describe the 802.11 and GSM wireless communications systems. The model captures a specification of the signal processing required to receive (or transmit) the signal. It is not a specification of the entire system, which would include features such as the beacon and control channel protocols.

### 5.3.1 802.11 Specification

As an example of the utility of the layering model, consider the 802.11 specification for Wireless Local Area Networks [IEEE, 1997]. This is a standard for 1 and 2 Mbps wireless LANs that have a single MAC layer and three physical-layer technologies. This situation leads to some confusion, as two 802.11 compliant devices may not be able to interoperate if they are using different, but acceptable, physical layers. The interoperability picture is further clouded by the different spectrum regulations in different countries.

A virtual radio, coupled with a way to communicate the details of the particular implementation, would enable radios to compile the appropriate flavor of 802.11 and then communicate. The layered model provides a mechanism for specifying the details of the particular instance.

The 802.11 MAC protocol is Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). The standard provides 2 Physical layer specifications (PHY) for radios operating in the 2400-2483.5 MHz band and one for infrared.

The standard also provides for a Physical Layer Convergence Protocol (PLCP), which is a physical layer dependent protocol that provides framing with error detection and correction. The physical layer segments the data into small frames. It is included because wireless links are often lossy, which leads to a very high likelihood of packet loss and high recovery overhead for long packets. The PLCP looks like a link layer protocol, but it depends upon physical layer parameters. Th PLCP operates in a manner similar to block coding, and is designed to detect and prevent errors, thus it is mapped into the channel coding layer. The PLCP for frequency hopping also performs DC bias control which is technically a line coding layer function, so the PLCP is an integration of both coding layers. This does not preclude the use of additional functionality in the line coding layer.

The Frequency Hopping Spread Spectrum Radio PHY provides for 1 Mbit/s (with 2 Mbit/s optional) operation. The 1 Mbit/s version uses 2 level Gaussian Frequency Shift Keying (GFSK) modulation and the 2 Mbit/s version uses 4 level GFSK.

The Direct Sequence Spread Spectrum Radio PHY provides both 1 and 2 Mbit/s operation. The 1 Mbit/s version uses Differential Binary Phase Shift Keying (DBPSK) and the 2 Mbit/s version uses Differential Quadrature Phase Shift Keying (DQPSK).

The infrared PHY provides 1 Mbit/s with optional 2 Mbit/s. The 1 Mbit/s version uses Pulse Position Modulation with 16 positions (16-PPM) and the 2 Mbit/s version uses 4-PPM.

| Layer | FHSS | Spec. DSSS | IR |
|---|---|---|---|
| Link | unspecified | unspecified | unspecified |
| MAC | CSMA/CA | CSMA/CA | CSMA/CA |
| Coding | PLCP-FH | PLCP-DS | PCLP-IR |
| Modulation | 2GFSK, 4GFSK | DBPSK, DQPSK | 16-PPM, 4-PPM |
| Multiple Access | FH | DS | none |
| Upconversion | 2400-2483.5 @ 1 W EIRP | 2400-2483.5 @ 1 W EIRP | baseband |

Table 5.1: Specification of layers for the three different versions of the 802.11 standard.

Table 5.1 illustrates the mapping of these specifications (for North America) into our layered model. Each layer has several parameters associates with it, but what the parameters are depends upon the specification of the layer. For example, the complete specification of one instance of an 802.11 compliant radio is given below:

- **MAC Layer**

  CSMA/CA, 802.11 implementation

- **Coding**

  PLCP-FH

- **Modulation**
  - GFSK 2 level
  - 1 +/- 50 ppm Msymbol/sec
  - h = 0.34, BT = 0.5
  - Low-pass Gaussian filter

- **Multiple Access** Frequency Hopping:
  - 79 nonoverlapping hopping frequencies (specified as a list)
  - 60 kHz center Frequency, accuracy 100 Hz
  - hop rate 224 usec
  - hop settling time 8 usec
  - transmit ramp up time 8 usec
  - transmit ramp down time

The layering model facilitates dynamic modifications to the processing system. If two radios were using this specification and one decided to increase its transmit rate to 2 Msymbols per second using the 4-level GFSK modulation, it would only have to inform the other radio of the changes to the modulation sublayer. Not only does the incremental nature of the change reduce communication overheads, but it also speeds reconfiguration time, as the entire system does not have to be re-built. Furthermore, it provides a concise framework for comparing different systems which often specify the wide range of physical layer functions in an *ad-hoc* manner.

### 5.3.2 GSM Specification

The Global System for Mobile communications (GSM) is a digital cellular communications system. It was designed to be compatible with ISDN systems and the services provided by GSM are a subset of the ISDN services, a block diagram of the GSM system is shown in figure 5-2 [Turletti, 1996]. Unlike 802.11, the GSM system is designed primarily for voice applications using dedicated channels for each connection. Mapping the GSM system into the layering model demonstrates how the specification, in particular the separation of multiple access functions, helps to clarify the functionality of the standard.

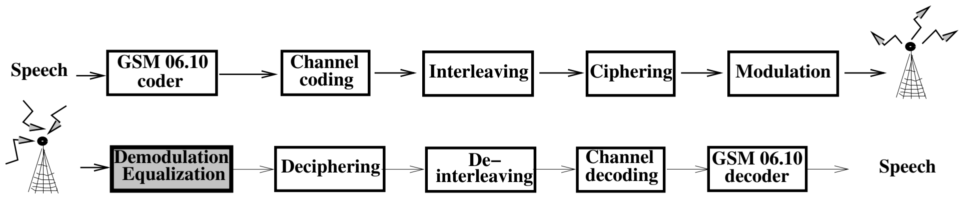


Figure 5-2: A Block Diagram of the GSM radio interface.

Mapping the GSM system into the layered model presented above results in a different expression of the system than is traditionally used [Mouly and Pautet, 1992]. The GSM multiple access scheme uses both FDMA and TDMA. The spectrum is divided into 200 kHz channels, each of which is subdivided into 8 timeslots. Each mobile unit is assigned one time slot on one particular channel. In the context of the layering model, this can be expressed in terms of the two multiple access protocols. The FDMA is the physical layer multiple access technique, which effectively defines a network, which is then shared by 8 different mobile units through the MAC layer time division. Moving the time division to the MAC layer does not change the processing performed by the system. In order to meet the tight timing constraints, the MAC layer may need timing information from the lower layers, but how this information is obtained is an issue for implementation, not specification of the functionality.

Following are the physical layers functions that are mapped into the each sub-layer for a single channel in the GSM Traffic Channel/Full-Rate Speech specification, which carries speech at the rate of 13 kbps:

- **MAC Layer**

  The speech data is handled in bursts of coded data that represent 20ms of speech, assigned to the appropriate time slot.

- **Coding**

  There are four GSM functions that are mapped into the channel coding layer: the GSM channel coding algorithm, bit interleaving, ciphering and burst formatting. The coding and interleaving are functions that are designed to detect, prevent and correct errors, so they are appropriately placed in this layer. The burst formatting adds training bits to the middle of the burst, to help in equalizing on the receive side, and guard bits to prevent interframe errors. The handling of encryption is discussed in section 5.5.

- **Modulation**

  GSM uses Gaussian Mean Shift Keying (GMSK) modulation with a modulation index of 05., BT=0.3 and a modulation rate of 270 5/6 kbauds.

- **Multiple Access**

  The FDMA scheme, which divides up the spectrum into 200 kHz channels is placed in this layer.

- **Frequency Conversion**

The are several versions of GSM that operate in different RF bands, GSM-900, DCS-1800 and PCS-1900. The uplink and downlink channels are specified for each system.

## 5.4 Security

There is no specific layer for security in the layering model. This is because there are many different security concerns, such as privacy, authentication and denial of service, and solutions can be implemented in different layers.

Existing systems implement security mechanisms in many different layers. Encryption of the source data at the application level is quite common, but systems such as GSM implement encryption at the bit level in the physical layer. Older systems even implement forms of encryption in the analog signal layers, such as cable TV scrambling and spectrum inverters for analog cellular telephones. Spread spectrum techniques are used by many voice and data networks to prevent denial of service attacks, both hostile and benign.

There are good reasons to implement security mechanisms at a variety of layers. In order to accommodate the specification of the various systems, the provision that security can be added to any layer is included.

However, this can lead to inefficiencies. For example, GSM supports data services as well as voice and it may well be the case that application data is already encrypted, making the physical layer encryption a waste of processing resources. Future enhancements to the layering model should include a mechanism for specifying the security aspects of each layer. These specifications could be used to determine the overall security level of the particular system, and also to eliminate redundancies that may exist in the security functions at different layers.

## 5.5 Summary

Traditionally, the physical layer is treated as a black box, which allowed for modularity but precluded system wide optimization. Implementing most of the physical layer in software opens an opportunity for system wide optimization while maintaining the ability to mix and match physical and link layers.

The multiple framing formats in the 802.11 specification are an example of the opportunity for optimization. The framing in the physical layer is included primarily to reduce the frame lengths for lossy channels. Rather than incurring the overhead of both link and physical layer protocols, the link layer could be configured to use smaller frame sizes. Two units wishing to communicate can dynamically specify the layers needed to communicate over the channel given existing channel and traffic conditions. They are not locked into a particular protocols, opening the opportunity to choose the best one for each communication.

The physical layer performs a wide range of functions. The layering model provides a framework for comparing different physical layer specifications, and identifying potential redundancies in functions such as error detection and encryption.

# Chapter 6

# Performance of Conventional Radio Architectures

The GuPPI I/O system presented in section 2.2.2 eliminated the primary system bottleneck facing software radio applications. To understand where the next most significant bottlenecks lie, we will analyze two typical applications, an FM receiver and and FM transmitter. The topologies investigated in this section were software implementations of traditional hardware system designs. However, the constraints that led to these designs, such as the functionality of certain devices and the cost of making wideband variable components do not apply to a software system. These applications illustrate the performance bottlenecks in a software system, and chapter 7 focuses on the design of applications and algorithms aimed at reducing these bottlenecks.

## 6.1 Application Performance

### 6.1.1 Reception Applications

To understand the bottlenecks facing reception applications a wideband FM receiver that mimicked a typical hardware implementation was measured. The channel selection filter was a 400 tap FIR filter that was sufficient to meet the demands of applications such as FM radio and to demonstrate the reception of AMPS cellular phone calls.

The performance graph shown in Figure 6-2 shows that the traditional system implementation requires more than 100% of the available CPU. In order to run this application in

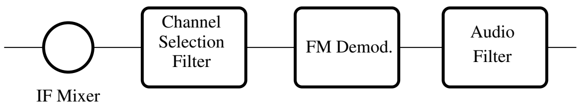


Figure 6-1: Topology of a typical hardware implementation of an FM receiver, which is used to evaluate the performance bottlenecks for reception applications.

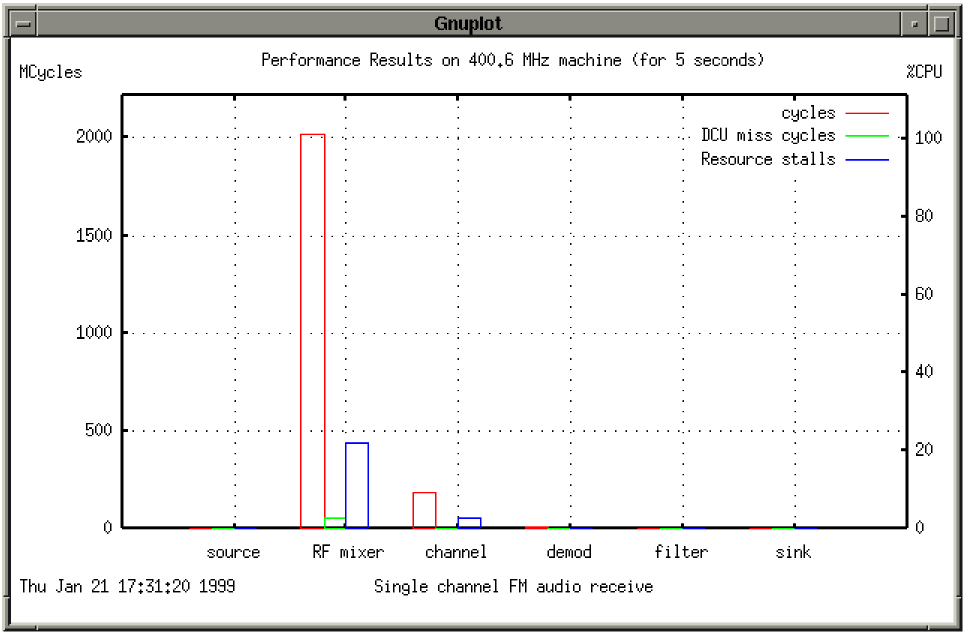


Figure 6-2: Performance of the FM receiver application on a PII/400.

real-time we must identify the key performance bottlenecks and engineer techniques for reducing or eliminating them.

The format used in figure 6-2 will be used throughout the dissertation to describe application performance. The horizontal axis is divided into groups that represent each module in the chain. Each group has three bars associated with it, the first is always the cycles utilized by that module. The other two bars can be chosen from a variety of events that can be monitored by the Pentium model specific registers. In this dissertation they will represent the number of cycles while a data cache unit (DCU) miss is outstanding, and the number of cycles during a resource stall. The cycles during a DCU miss indicate the amount of time that is spent servicing the cache. The processor is not necessarily stalled during this time, since the L1 cache is dual ported and may be performing a speculative execution. In addition, the cycles during a DCU miss are weighted by the number of cache misses that are outstanding. The cycles during a resource stall are cycles that accumulate during any of the many conditions that cause the processor pipeline to stall, which is a reasonable indicator of the degree of variability in execution times. For a more detailed description of these counters, the reader is referred to the Pentium Optimizations Manual [Intel, 1998].

Figure 6-2 indicates that the first stage of the receiver, the IF mixer, consumes the most CPU cycles. However, the reading of input data or the computation involved are not the reasons that this module is so CPU intensive. The system was not stalled waiting for input data due to the high data rate I/O supplied by the GuPPI, and the cycles while a DCU miss is outstanding are almost insignificant when compared to the cycles consumed. Conventional wisdom would predict that the computation of the *sin()* function for each multiplication might be the reason for the heavy CPU requirement, but figure 6-3 shows the performance for the same application with the mixer function replaced by a simple loop that writes a constant to the output for each value. This experiment eliminates both the

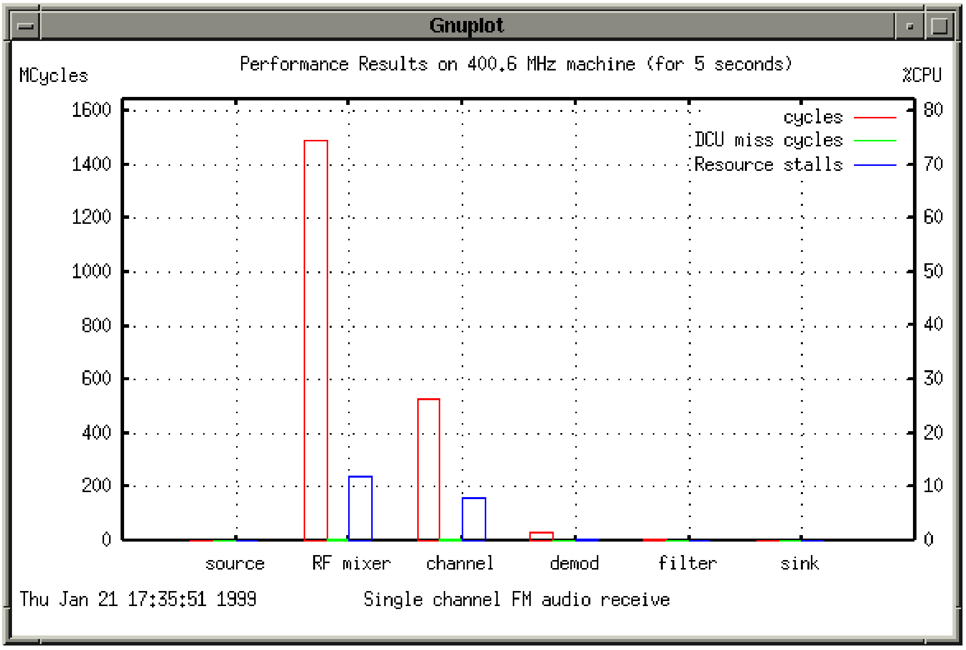


Figure 6-3: Performance of the FM receiver application on a PII/400 with the computation in the mixer eliminated.

computation and the dependence upon input data from the performance, and we see that the mixer still consumes substantially more cycles than any other module. The reason for this is that the mixer must write output data at a 33 MHz rate. Values written to the cache must be written through to main memory, and this is the primary performance bottleneck.

### 6.1.2 Transmission Applications

The transmission problem is considerably less complex than reception, because it is a straight mapping from symbols to signals whereas reception must deal with extracting a signal that has been corrupted by noise. However, in wideband software radio applications, the computational resource demands can be much higher for transmit applications because transmission is essentially writing data at a high rate. Figure 6-4 shows the topology of a typical transmitter, and Figure 6-5 shows the performance of this application in the SPECtRA environment. As with the reception application, the primary bottleneck is the writing of data to memory. In a wideband transmission application, the last module in the chain requires the most memory write operations since it must place the wideband waveform in memory where can be transferred directly to the D/A converter.

### 6.1.3 Summary

The GuPPI I/O system eliminated I/O bandwidth as the primary bottleneck. Through the measurement of representative transmission and reception applications in this chapter, we have shown that the most significant remaining bottleneck is now memory write operations.

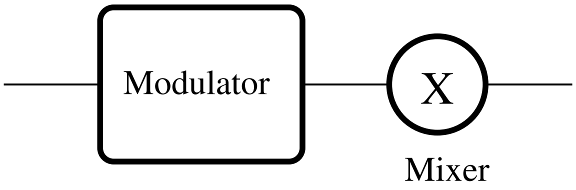


Figure 6-4: Topology of a typical hardware implementation of an FM transmitter, which is used to evaluate the performance bottlenecks for transmission applications.

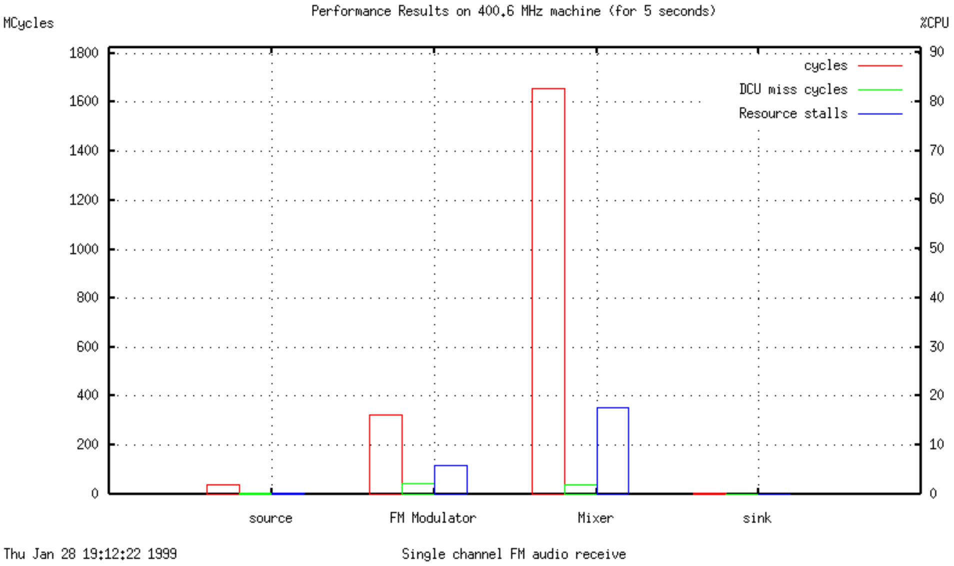


Figure 6-5: Performance of the FM transmitter application on a PII/400.

For wideband transmission applications, it is necessary to write a large number of samples into memory. The next chapter explores some design approaches that utilize a combination of pre-computed buffers and computation to reduce the number of write operations.

For reception algorithms, writing to memory is not an inherent requirement. Since write operations are so much more expensive than anything else, it is quite reasonable to trade additional computation, even floating point computation, for a reduction in memory writes. If we can remove the memory writes as the main bottleneck, then the system will be CPU limited. This is the ideal situation, since the CPU is the most rapidly improving component of the system. The next chapter examines application and algorithm design techniques aimed at reducing the number of memory write operations required.

The next chapter investigates different application architectures and algorithms designed to overcome the bottlenecks and take advantage of the benefits provided by the software implementation.

# Chapter 7

# Application Design

This chapter addresses the key bottlenecks identified in the previous chapter through the use of both application and algorithm design. Although the bottlenecks for reception and transmission both involve high data rate streams, different design approaches are needed to improve performance in each case. For reception it is possible to eliminate the writing of data as a bottleneck by adding functionality to the first processing stage. Since it is possible to bring the required data rate into the cache, the challenge is then in performing the often complex computation on that stream. For transmission, writing the data to memory is essential, thus, the performance limiting factor is the rate at which data can be written into memory, which is considerably slower than the data read operations required by reception applications.

Wideband reception applications can be divided into two classes, those that must separate out the channel or channels of interest, and those that receive a single dedicated channel and must perform demodulation and detection on the wideband stream. Section 7.1 presents the design and analysis of channel selection filters in a frequency division system, as an example of a multi-user system. This is followed by the description of a detection algorithm that might be used on a single wideband channel. The chapter concludes with a look at transmission systems in the context of a wireless network link utilizing frequency hopping with FSK modulation.

## 7.1 FDM Channel Selection

This section examines several performance optimizations for the selection of frequency division channel as well as the design of an application for the selection of multiple channels simultaneously.

### 7.1.1 Frequency Translating Filter

By combining the functionality of the mixer, channel selection filter and decimation into one module, we can significantly reduce the number of memory write operations. The functions are combined using a novel frequency translating filter [Welborn, 1999] that takes

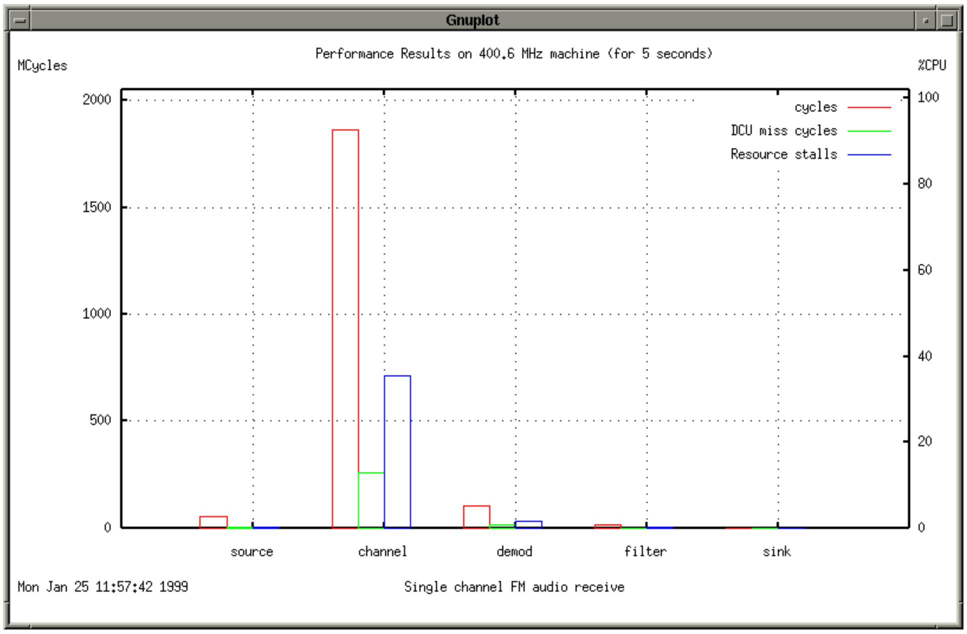


Figure 7-1: Performance of the FM receiver application on a PII/400 with the stages of frequency translation and channel selection combined.

advantage of the flexibility of software by designing the filter on-the-fly for each application and by combining the functions of frequency translation and filtering. Figure 7-1 show the performance of this application. By combining these functions, this application adds considerable computation to the first stage in the form of a 400 tap FIR filter and the multiplication by a complex exponential for the frequency translation. In spite of this, the overall cycle requirement is reduced by approximately 15%, and the application is just barely able to run in real-time. Note that the cache performance is worse that the example in the previous chapter due to the more memory intensive filtering operation, but this is a reasonable tradeoff in terms of overall performance. More importantly, the bottleneck is now no longer the memory write bandwidth of the system, but the computation performed in the first stage. The next section shows how the overall load can be further reduced by addressing the computational structure of the combined frequency translation and filtering function.

### 7.1.2 MMX Channel Selection Filter

The Intel MMX instruction set adds single instruction multiple data (SIMD) capability to the Pentium family of processors. This allows for the implementation of vector style functions that operate on blocks containing multiple input samples with one instruction. We improved the performance of frequency translating filter by implementing the complex multiply-add instructions with MMX. The structure shown in Figure 7-2 replaces 4 multiplies and one add with a single MMX instruction. This reduced the CPU requirements of the frequency translating filter by more than 50%, as shown in figure 7-3.

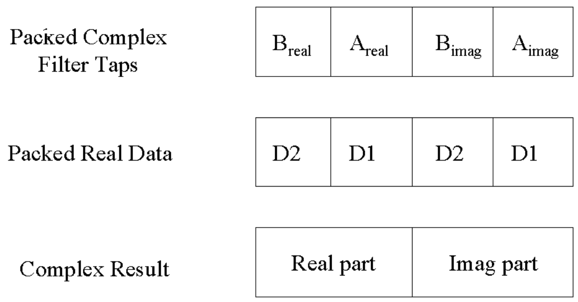


Figure 7-2: MMX implementation of Complex multiply-adds.

The results of the MMX enabled filter confirm that the primary bottleneck for reception applications is now CPU performance. This allows the system to directly reap the benefit of Moore's Law. Moving to the Pentium II reduced the overall CPU load significantly below 50%, and enabled the implementation of functions requiring significantly more computation, or of systems that could receive multiple channels.

### 7.1.3 Multi-Channel Receiver

The improvement in performance gained by reducing memory writes and utilizing the MMX instructions open the possibility of performing more computation on the incoming stream. This section describes the design of an application that simultaneously receives four FM channels located anywhere within a given 10 MHz band. The receiver was designed to minimize the performance penalty due to caching in the first processing stage. The channel filter is simply four separate channel filters built into the same module, and they run serially, each writing data to the appropriate buffer. Figure 7-4 shows the block diagram for an application that demodulates four AMPS channels simultaneously. The percentages taken by each of the blocks is shown in the figure. Note that this filter requires considerably less than four times the number of cycles of the single channel AMPS filter. Figure 7-5 shows that the CPU requirements increase by less than a factor of 3. This is due to a number of factors including: the overhead of only one function call for the four filters, improvement in icache performance for subsequent filters and warming of the data cache for both input data and filter tap arrays. The cycles while a DCU miss is outstanding increase by only a factor of 2. It is likely that the cache performance could be further improved by designing the filters to run in parallel, rather than serially, so that together they exhibit better locality of data reference.

This application is an excellent example of the benefits of technology tracking enabled by the virtual radio approach. When the first applications were implemented, only a single channel could be received in real-time with the fastest available processor. The subsequent processor improvements enabled this four channel receiver since the entire system was easily portable to the new processor generation. Future performance improvements will come from new algorithm work as well as improvements in the underlying technology.

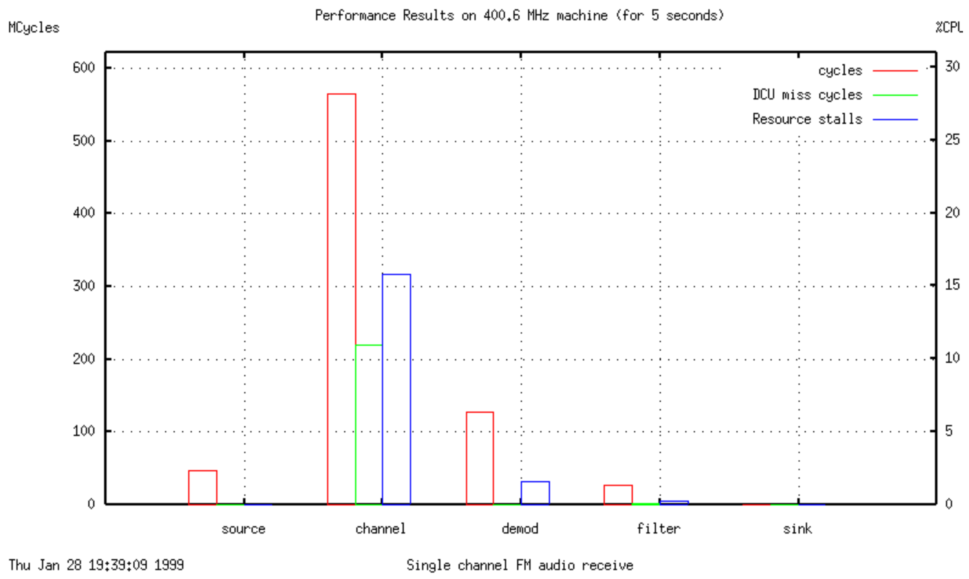


Figure 7-3: Performance of the FM receiver application on a PII/400 with the stages of frequency translation and channel selection combined and implemented with MMX instructions.

## 7.2 Symbol Detection

Receivers for dedicated wireless links have the same performance bottlenecks as the multiple channel receivers. The difference is that the first stage is not channel selection, but the reception of transmitted signals. Reducing the number memory write operations is still the top priority. Once this has been achieved, the performance bottleneck is moved to the processing of the signal.

In a communications system, symbols are modulated into waveforms which are then transmitted. The receiver must then perform a detection operation to determine which symbol was actually transmitted. For the software implementation of wideband transmission sys-

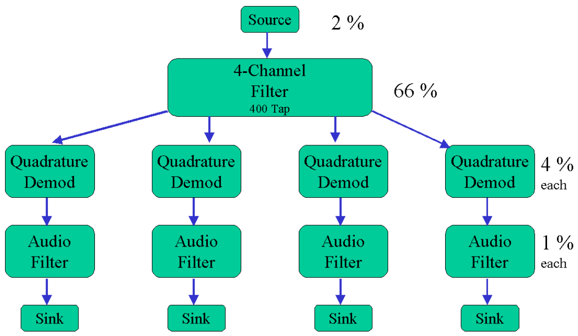


Figure 7-4: Block diagram of the 4 channel amps receiver. The percentages indicate the percent of the CPU utilized by blocks in that layer.

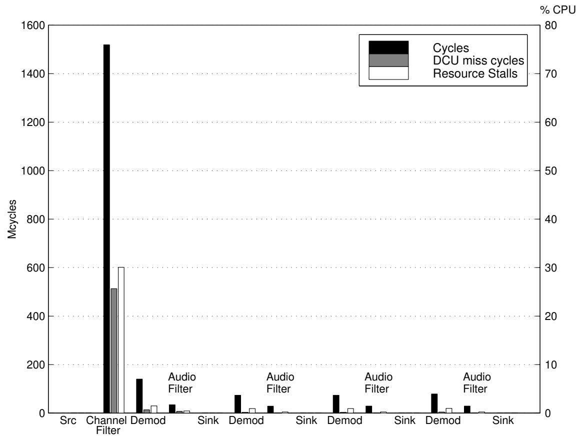


Figure 7-5: Resource usage for the four channel receiver.

tems it is important to optimize the performance of detection algorithms, since they operate on the high data rate streams. Optimal detection performance is obtained by the matched filter, which maximizes the ratio of signal to noise components in its output. This section compares two algorithms for implementing the matched filter, the first is a straightforward implementation of the matched filter as a correlation, and the second is an algorithm that performs incremental comparisons of a function of the received signal to an energy threshold. The latter algorithm has variable computational requirements, but a much lower average cycle requirement.

The problem addressed is that of detecting a signal ($s[n]$) in the presence of additive white Gaussian noise ($w[n]$). The input to the detector is either the signal plus noise, or simply white noise. This can be viewed as a binary hypothesis testing problem, where:

$$r[n] = \begin{cases} H_1 : \; s[n] + w[n] \; if \; signal \; is \; present \\ H_0 : \mathrm{w[n]} \qquad otherwise. \end{cases} \tag{7.1}$$

This representation of the matched filter is briefly reviewed in order to provide a context for analyzing the proposed detector. This is followed by the presentation and computational performance analysis of the detector.

### 7.2.1 Matched Filter

The matched filter for a signal subject to additive white Gaussian noise can be expressed as a correlation filter. The output of the correlator is then thresholded to determine the presence or absence of the signal. The matched filter can be expressed as a hypothesis testing problem, which tests the signal received at time index $i$:

$$\sum_{n=1}^{N} r[n+i]s[n] \begin{matrix} > \\ < \end{matrix} \alpha \tag{7.2}$$

where $\alpha$ is a predetermined threshold. For example, if the two hypothesis are equally likely, the threshold to minimize the probability of error is:

$$\alpha_{minP_e} = \frac{1}{2}N_o + \frac{1}{2}\sum_{n=1}^{N} s^2[n]$$

### 7.2.2 Alternative Matched Filter Algorithm

In the matched filter, each term of the correlation sum can be either positive or negative, hence an decision cannot be made to terminate the computation of the correlation early. The detector presented next has the property that each term of the sum is non-negative, hence the sum is monotonically non-decreasing with each term. Furthermore, the sum is accumulating a measure of error, so that once the sum reaches a predetermined error threshold the summation can be terminated because the error is deemed to be too high. To demonstrate that the proposed approach has the same detection performance as the matched filter, the likelihood ratio will be derived from the matched filter likelihood ratio.

To start, both sides of the hypothesis test given in Eq. 7.2 are multiplied by $-2$, then signal energy $E_{signal} = \sum_{n=1}^{N} s^2[n]$ is added to each side. Note that $E_{signal}$ is a known constant.

$$\sum_{n=1}^{N} \left(-2\ r[n+i]s[n] + s^2[n]\right) \begin{matrix} < \\ > \end{matrix} E_{signal} - 2\alpha$$

Next the received signal energy for the interval beginning with the $ith$ sample $E_i = \sum_{n=1}^{N} r^2[n+i]$ is added to each side. This quantity is not a constant and must be computed for each interval. The brute force method of computing this quantity requires $O(N^2)$ computations. However, since we are interested in continually computing this quantity for each interval, the increments can be computed at a small cost. $E_i$ can be computed as a running sum, that only requires one addition and one subtraction and two squaring operations to update for each iteration, as illustrated by the following equation:

$$E_{i+1} = r^2[N+i+1] - r^2[i] + E_i$$

The received energy in each signal interval can be pre-computed for each threshold comparison, and treated as a constant for the comparison. Adding this quantity to each side of the test yields:

$$\sum_{n=1}^{N}\left(r^2[n+i]-2r[n+i]s[n]+s[n]^2\right)\ \underset{<}{>}\ E_i+E_{signal}-2\alpha$$

Letting $\beta = E_{signal} - 2\alpha$ and simplifying, the likelihood ratio can be expressed as follows:

$$\sum_{n=1}^{N}(r[n+i]-s[n])^2\ \underset{>}{<}\ E_i+\beta \tag{7.3}$$

This hypothesis test has the same performance as the matched filter, but the term on the left hand side, the sum of the squared error between the received and desired signals, is monotonically non-decreasing. Taking advantage of this property, the next section describes an algorithm that can terminate computation early, as soon as the threshold is exceeded. Note that $\beta$ can be a negative quantity. Thus, it is possible that $E_i+\beta$ is negative, indicating that there is no need to compute any terms of the sum since zero is the lowest possible value for the sum. This is similar to using a received signal strength indicator to asses whether computation is warranted.

### 7.2.3 Algorithm

The detection algorithm using the proposed hypothesis test is given below. The variable $i$ is the index into the input data array, and $n$ is the summation variable for the correlation computation; it is assumed that both are initialized to 1. $S_n$ denotes the first $n$ terms of the sum on the left hand side of the test.

1. Compute $E_i$
2. If $E_i < -\beta$ then i += 1, goto step 1.
3. Add the $nth$ term to the sum, forming $S_n$.
4. If the $S_n > E_i + \beta$, $n = 1$, $i+ = 1$, goto step 1.
5. If $i = N$ indicate detection, $n = 1$, $i+ = N$, goto step 1.
6. n += 1, goto step 3.

Note that step one requires summing $N$ terms when the algorithm is started, but then can be updated with one addition and one subtraction and two squaring operations as discussed above. The algorithm searches for a match in the interval beginning at time index $i$. This search can terminate in step 2, step 4 or step 5, and the amount of computation varies depending upon where each search terminates.

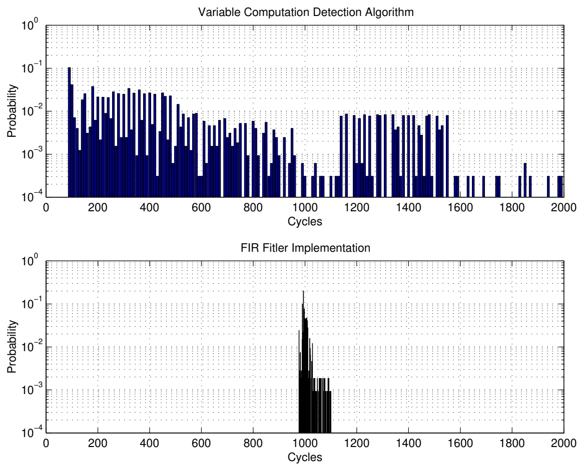


Figure 7-6: Histograms for the number of cycles required to compute each output point for the variable computation detection algorithm and an FIR filter implementation of a matched filter, for a signal length of 100 samples.

### 7.2.4 Computational Performance

A traditional evaluation of the computational performance of the detection algorithm results in a worst case bound that is $O(N^2)$. However, with a variable computation algorithm, the bound is not a good estimate of the required computation if the worst case occurs rarely. The worst case occurs when the algorithm terminates in step 4 after $N-1$ iterations. At this point the pointer to the input array in incremented by one, and the algorithm is started again. Thus each input point could be involved in maximum of $N*(N-1)$ iterations. $N-1$ is the maximum number of computations that can be performed before it is determined that no signal is present in the given interval. In many cases, the algorithm requires significantly fewer iterations before terminating.

Figure 7-6 presents histograms of the cycles required to produce each output point for the detection scheme outlined above and an FIR filter implementation of a matched filter, both for a signal length $N = 100$. Both algorithms were tested using simulated input data that was equally likely to contain white noise or a signal in a given interval. The variable algorithm has a lower average cycle requirement, but displays more variation as illustrated in Table 7.1.

| Cycles | Variable Algorithm | FIR filter |
|---|---|---|
| Minimum | 80 | 975 |
| Maximum | 3400 | 1100 |
| Average | 506 | 1001 |

Table 7.1: Cycle requirements for the variable computation algorithm and the FIR filter implementation.

The variable computation detector provides for computational savings on a significant percentage of the trials, and using the system approach described in this thesis, those cycles can be applied to other applications. Alternatively, this algorithm could dynamically control the CPU speed to save power. The CPU speed could be kept low during steps 1 and 2, and for the first few iterations over step 3. If more iterations are required, then the CPU speed could be increased to provided the necessary cycles. An upper bound on the potential power savings can be determined by assuming that the algorithm was smart enough to exactly adjust for each iteration. Table 7.2 shows the square of the cycles required for each implementation, which is proportional to the power consumed if the CPU can be dynamically regulated. Note that this cannot be computed directly from the average number of cycles required, since the power dissipation is proportional to the square of the cycles.

| Variable Algorithm | FIR filter |
|---|---|
| $4.4 \cdot 10^5$ | $10 \cdot 10^5$ |

Table 7.2: Total power consumption, approximated by the square of the required cycles, for each algorithm with dynamic clock management.

Another interesting aspect of this detection algorithm is that the computation requirements vary with the SNR of the received signal. It makes intuitive sense that a signal can be distinguished more easily if the SNR is high. This algorithm transparently adapts the amount of computation to the SNR of the received signal.

The detection algorithm presented in this chapter demonstrates some of the potential benefits of designing algorithms that display variable computation. By reducing the constraint on the worst case, algorithms can be designed to have much better average performance.

## 7.3 Transmission

Early in the previous section we demonstrated that it is possible to remove the high data rate memory write operations from reception applications by combining functionality in the first processing stage. For transmission applications, this is not possible, since we must generate the high data rate sequence for transmission. We have investigated transmission applications that trade memory usage for reduced writes in some applications. The following section describes the transmission portion of a frequency hopping network link that we designed and implemented in the SPECtRA processing environment.

### 7.3.1 Example Transmission system

The sequence of processing modules for the transmission application is shown in figure 7-7. The system interfaces with the host at the IP layer, through our *SoftLink* device driver. The first level of processing is the network framing. For this example, a technique known as Consistent Overhead Byte Stuffing (COBS) [Chechire and Baker, 1997] is used. Start and stop codes frame the packet and the data is byte stuffed with a maximum overhead of 1 byte in 254. COBS also handles byte framing, which is a channel coding mechanism. Note that this system does not contain a line coding layer, which means that the symbols input to the modulation layer are actually bits.

The conversion of each symbol into a discrete signal is performed by the FSK module, representing the modulation layer. The multiple access technique is frequency hopping,

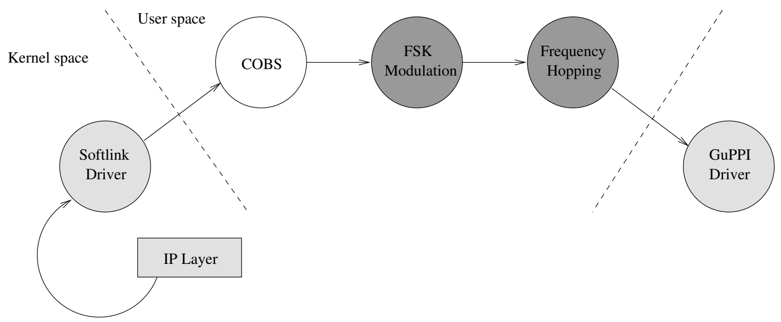


Figure 7-7: Software components of the transmission application

which assigns the waveform to the appropriate IF frequency. In our implementation, we combined the modulation and multiple access layers, which resulted in significant computational savings. This allowed us to directly generate the IF sinusoid corresponding to the particular bit and hopping frequency, rather than generating a sinusoid for each bit, and then re-modulating that sinusoid to the appropriate hopping frequency. All of the possible transmission waveforms are known *a priori*. There are two possible waveforms, corresponding to 1 or 0 for each hop frequency. All of these waveforms can be precomputed and stored at startup, significantly reducing the computation required to produce the transmitted waveform. The amount of time required for each transmission function is given in table 7.3.

| Transmit Function | Time |
|---|---|
| COBS | 0.25 $\mu s$ / bit |
| Frequency Hopping and FSK Modulation | .3 $\mu s$ / bit |

Table 7.3: Average time required for each transmission function on a 400 MHz Pentium II.

On a 400 MHz Pentium II, .725 $\mu s$ were required for producing the IF waveform for a single bit, which corresponds to a maximum possible transmit data rate of $\approx 1.38Mbps$. The average time needed to produce the waveform for a bit can be decreased by exploiting the behavior of the memory and the cache. Since a memory copy of $2n$ bytes takes less than twice the time to copy $n$ bytes, grouping two bits at a time and producing IF waveforms corresponding to 00, 01, 10, and 11 results in a smaller average processing time per bit. We are trading off increased memory usage for decreased processing time. In this implementation, 3 bits were output at a time, resulting in an average processing time of .3 $\mu s$ per bit. This corresponds to a maximum possible transmit data rate of $\approx 3.3Mbps$.

This application was able to take advantage of the pre-stored waveforms because it utilized carefully chosen frequencies and bit periods so that the phase at the beginning and end of each waveform matched. If we were to use arbitrary frequencies, then there would be no guarantee of phase matching. This would be a problem for many systems which require continuous phase transmissions. For analog systems the problem is even worse, since there are no pre-defined sequences that are transmitted repeatedly.

The inherent memory write bottleneck for transmission applications limited our usable transmission sampling rate to 5 MHz, as opposed to the 33 MHz input sampling rate. Algorithms and processor support for transmission waveforms is a critical area for further study if a bi-directional wideband software radio is to be realized.

## 7.4 Summary

This chapter has demonstrated several application design approaches that help to overcome the primary system bottlenecks. Wideband reception algorithms show great promise, as careful applications design can move the bottleneck to the processor. This allows reception applications to ride the Moore's Law technology curve, as was demonstrated by the implementation of the four channel receiver on the recent PII/400 processor. Transmission applications on the hand need significant attention, as the memory write speed is the limiting factor.

The potential utility of signal processing algorithms that display variable computation requirements was also demonstrated in this chapter. Our design approach not only can reduce the average cycle requirements, but also may have significant implications for the design and use of low power processors as research on dynamic clock and voltage control matures.

# Chapter 8

# Discussion

In this chapter, we discuss the implications that this thesis work has for processor and workstation architecture with respect to signal processing capabilities. The work in this thesis has demonstrated that workstations and general purpose processors are capable of supporting a wide range of computationally intensive real-time signal processing applications. With the advent of multimedia many real-time signal processing applications are migrating to the desktop environment and these applications should be considered, along with traditional applications such as data bases and graphics, in the design of general purpose processors. This chapter discusses these issues beginning with an analysis of some of the key DSP architecture features as compared to a general purpose architecture in section 8.1. Section 8.2 discusses the implications that this work has for low power systems and section 8.3 concludes with a discussion of potential modifications to workstation architecture that would enhance real-time signal processing.

## 8.1 Processor Architecture

With the advent of multi-media, some features that were previously unique to DSP's are beginning to appear in general purpose designs. Both the alpha and strongARM processors have single cycle multiply accumulate instructions which greatly enhance the performance of many signal processing functions such as FIR filters and mixers. Parallelism is now found at many levels in the processor. SIMD instruction sets such as Intel's MMX and Sun's VIS offer parallelism at the instructions level. Super scalar architectures provide parallelism at the instruction unit level, and many operating systems provide multi-tasking support. Beyond that the parallelism from SMP processor configurations and distributed systems has the potential to bring tremendous computational resources to bear on real-time signal processing problems.

Data-intensive real-time signal processing applications place several challenging demands on any processor architecture. Primary among these are support for continuous high data rate streams into and out of the processors and for performing considerable computation on the input streams. Other constraints include support for low latency services and for algorithms that do not display the locality of data reference needed to take advantage of memory cache structures.

Until recently, general purpose processors could not support the data I/O or the processing required by many DSP applications. As a result DSP architectures have evolved many special features design to meet these challenges. This section examines the utility some of these features in light of the signal processing capability of general purpose processors demonstrated in this dissertation.

### 8.1.1 Memory Architecture

The combination of high data rate streams and the need to support algorithms that do not display locality of data reference has resulted in specialized memory architectures for DSP processors. This section examines some of the most common DSP processor features, the challenges they were designed to overcome, and the ability of general purpose processors to handle these challenges.

#### Memory Bandwidth

The Harvard architecture, a very common feature in DSP designs, specifies the separation of program and data memories, each with a separate path to the processors. This was developed in order to avoid interrupting the data stream to access program instructions and thereby reducing the sustainable data flow. However, modern general purpose processors include an instruction cache, which can greatly reduce the number of cycles during which the memory bus is busy with instruction transfers. In addition, many DSP applications involve the execution of fairly small loops many times, during which the icache can typically hold all of the necessary instructions. The applications presented in this dissertation were not starved of input data, which indicates that accessing instructions from memory does not significantly impact the sustainable data I/O rates to and from the processor.

#### Locality of Reference

Many DSP algorithms, such as the FFT, do not display locality of data reference. Caching structures, such as those used on most general purpose processors can be a considerable detriment to performance for such algorithms. As a result most DSP architectures do not include data caches. Many DSP designs have on-chip memory, which provides the fast access to data that is provide by on chip caches without the performance hit for algorithms that do not display locality of data reference. However, this model places the burden of fine-grained management of the on-chip memory on the programmer and/or compiler. By contrast, cache management in general purpose processors is done by the processor. While theoretically, the on chip memory can be optimally managed, this would require information that is not available to the compiler such as the memory usage of other processes.

Instead of re-designing the processor for these algorithms, a more practical approach may be to re-design the algorithms. The canonical example of a DSP algorithm with poor locality of reference is the FTT. The traditional FFT algorithm was designed to minimize multiply-add instructions, which today are much less costly than cache misses. The FFTW algorithm [Frigo and Johnson, 1998] modifies the algorithm to take advantage of the cache, and results in much better performance. The algorithm runs a few tests at startup to determine the effective available cache size, and then sub-divides the FFT into a set of smaller problems, each of which fit into the cache. The drawback of this approach is that the availability of the cache is determined at startup, and the algorithm will not perform as predicted if the cache availability changes due to the activity of other functions. However, the examples presented in chapter 7 have fairly constant cache utilization within a given processing chain. Even if other processes run, the high input data rate to the processing chain will cause the entire cache to be flushed. Thus the cache conditions when the FFTW module is run should be consistent.

Another example of taking advantage of locality of data access is the 4-channel AMPS receiver presented in chapter 7. Instead of processing each of the chains sequentially, the four channel selection filters are run first, since they all operate on the same input data which is brought into the cache when the first filter is run.

The lack of locality of data reference is not a fundamental problem with DSP functions, but a property of many of the algorithms that are traditionally used. Signal processing algorithms and applications can take advantage of the benefits of data caching if the algorithms are properly designed. In general, software signal processing algorithms must be designed with the appropriate cost metrics for a software environment, and one of these metrics is locality of data reference.

### 8.1.2 Computational Capabilities

Many DSP algorithms require extensive computation which could not be performed in real-time on early generations of general purpose processors. DSP processors include specialized instructions, program control and addressing modes designed to increase the computational performance for DSP applications. This section reviews these features, the performance enhancements that they provide and discusses the ability of general purpose features to provide similar performance improvements.

#### Specialized Instructions

There are many specialized combinations of functions that are commonly found in DSP algorithms, and DSP processors often incorporate these combinations into single instructions. The most common such instruction is the multiply-add, which can significantly improve the performance of a wide range of functions such as filters, mixers and a variety of transforms. Another common feature is a vector instruction set which can take advantage of the parallelism inherent in many DSP functions.

The examples presented in this thesis have shown that many receiver applications are computationally limited. Therefore adding specialized instructions makes sense, since they will help overcome the primary bottleneck. In fact, many modern general purpose architectures incorporate instructions that were previously only found in DSP architectures. For example, the alpha processor has a single cycle multiply-add instruction, and both the Pentium (MMX) and Sparc (VIS) processors contain vector instruction sets. The examples presented in this dissertation have demonstrated that the Pentium MMX instruction set can significantly boost the performance of some signal processing applications. The MMX instruction set does have some drawbacks, in particular the requirement that the data be 64 bit aligned and the inability to mix floating point and MMX instructions. In future architectures, a more general vector capability, and one that incorporates floating point data types, would be a significant improvement.

The incorporation of these instruction sets has been motivated by the increase in multimedia applications, which require extensive signal processing of both audio and video signals. In addition, mechanisms such as branch prediction and speculative execution improve the performance of signal processing applications by efficiently implementing small loops and program branches. These instructions, and architectural features coupled with the raw speed today's general purpose processors have resulted in computational engines that can outperform DSP processors at many signal processing tasks [Stewart *et al.*, 1992].

### 8.1.3 Computational Variability

The examples presented in this dissertation demonstrate that general purpose processors are capable of supplying the raw computational power and memory bandwidth required to implement a wide range of real-time signal processing applications. However, the most significant difference in current DSP and general purpose platforms is the variability in execution times [Bier *et al.*, 1997].

The variability is primarily a result of a design choice to optimize for the average case, rather than to reduce the worst case. Many of the mechanisms discussed in this section that assist signal processing applications do so at the expense of adding variation to the execution times. Caching, branch prediction and speculative execution all improve average performance, but when conditions are not right the performance suffers. Fortunately, with appropriate application design, these worst case scenarios can be limited to an occasional performance hit.

In order to take advantage of the superior computational performance of general purpose processors, it is necessary to understand and design for computational variation. The methods and mechanisms outlined in chapter 3 permit a certain class of real-time signal processing applications to take advantage of general purpose processors and operating systems. The key factors are the ability to process data faster than real-time, and transforming deadline misses into a form that can handled in the context of the application's inherent data error rate. As processors continue to get faster, the class of real-time signal processing applications that can be implemented on general purpose processors will expand.

## 8.2 Low Power Processing

Power consumption is a significant factor for many DSP processor applications. It is also a growing factor in general purpose designs. The most significant factor in processor power dissipation the clock speed, as the power dissipation is proportional to the square of the CPU clock speed. Specialized instructions that reduce the number of cycles required allow the clock speed to be reduced and power to be saved.

Traditionally the clock speed is determined by the maximum number of cycles required by the application, so reducing the maximum number of cycles required by a particular application is a critical factor. However, with recent advances in dynamic clock management it is possible to adjust the clock speed to supply the required cycles as the algorithm is running. To take advantage of this feature, requires a different way of thinking about the design of DSP algorithms. The goal is no longer to minimize the maximum number of cycles required, but to minimize the average of the square of the number of cycles required.

The variable computation detection algorithm presented in chapter 7 is a good example of an algorithm that can take advantage of dynamic clock management. In addition to having a lower average cycle requirement, this algorithm automatically adapted the amount of computation required as the received SNR changes. An algorithm designed to meet the worst case SNR requirements would utilize more power than necessary when the conditions are not worst case.

While the use of a desktop platform for the SpectrumWare testbed does not directly lend itself to a low power application, it is a good platform for developing and evaluating software design approaches aimed at lowering power consumption.

## 8.3 System Issues

The system design presented in this dissertation involved the modification of a conventional PC-based workstation architecture to support real-time, high data rate signal processing applications. This section suggests some recommendations for modifications to workstation architecture that would better support these types of applications.

The initial bottleneck for software radio applications in existing workstations was the sustainable I/O rate to and from the application. Design of the SpectrumWare system involved the development of a specialized I/O system and operating system modifications to support the high data rates required by wideband software radio applications. With the custom designed I/O system, the performance bottleneck was moved from the I/O system to the processing of the data. The I/O system can sustain a 512 Mbps rate which is more than sufficient for many wideband software radio applications, thus the performance of the software radio system can ride the computational improvements of Moore's Law for some time to come. However, the performance can be hurt by other bus activity, in particular transfers involving the network or graphics cards. Similarly, graphics applications have been limited by the the available bus bandwidth. To combat this problem, the advanced graphics port (AGP) has been developed, which off-loads graphics activity to a separate, dedicated bus. The software radio applications implemented in this thesis have sustained data rates that are greater than many graphics applications and in addition they require bi-directional data transport support. Specialized *analog* ports with a dedicated path to memory would reduce the load on other parts of the system and be justified since the data transfer rates are so high. The ports should contain A/D and D/A converters and provide operating system control of sampling rates and filter cutoff frequencies. These ports would enable workstations to access a vast array of information sources and support a wider range of services.

After the I/O system, the next most significant bottleneck is memory write operations. This limitation can be overcome for receive applications, but is more fundamental for transmit applications. For receive applications the bottleneck is the processing power that can be brought to bear on the input stream, whereas memory writes are the bottleneck for transmit applications. Receive applications benefit from Moore's law by the incorporation of instruction set enhancements as discussed in section 8.1.2.

Unfortunately, transmit applications do not benefit from these same enhancements. Workstations can be better designed to support transmit applications by observing that, from the processor's point of view, the transmit waveform buffer is *write only* memory. A write-back, rather than write through cache mode could be used to reduce the dependence upon memory writes, but there must be a mechanism to insure synchronization before this region of memory is handed to the DMA engine.

Another approach to improving transmit application performance would be the incorporation of external support, such as an IRAM processor [Patterson *et al.*, 1997]. An IRAM is essentially a small processor embedded into the memory. This processor could handle the final transmission stage of mixing the signal up to the IF frequency, which requires creating a high data rate stream. Mixing is a fairly simple task that would not require much computation, and the IRAM would not have to deal with the write-back or write-though issues that the processor does. This would move the bottleneck for transmit applications to the processor as well, allowing all applications to track Moore's Law.

While this dissertation has shown that it is possible to implement real-time applications using a general purpose operating system, these systems can be further improved. In particular, the reduction of the scheduler granularity would significantly improve the worst case performance of applications without adversely affecting the responsiveness of the system to the user. Streamlining interrupt and context switching overheads would also help to reduce the variability, and bring a broader class of real-time applications into the domain of software signal processing. However, it should be emphasized that these are not the primary bottlenecks to performance, and many applications run very well without these improvements.

# Chapter 9

# Conclusions and Future Work

The primary goal of the work presented in this dissertation was to introduce flexibility, in the form of software processing, to every aspect of the physical layer of a wireless communications system. The added flexibility enables solutions to many problems and limitations seen in today's wireless communications.

The virtual radio system presented in this dissertation implements all of the physical layer processing in software on a general purpose platform. The system was designed to overcome the many challenges to real-time signal processing present in a general purpose environment, in particular the uncertainty in execution times and resource availability, while taking advantage of many of the tools and resources that aren't typically found in digital signal processing systems. Overcoming these challenges opened opportunities for new approaches to both system and algorithm design that result in greater flexibility, better technology tracking and improved average performance. The work demonstrates that general purpose processors and operating systems are capable of implementing a broad class of real-time signal processing applications. Furthermore, the scope of the applications that can be implemented will increase as processors and memories get faster.

The SPECtRA programming environment was written to support the real-time, data intensive signal processing. It employs a data-pull model of execution and a stream abstraction which greatly simplified the implementation of signal processing algorithms. The system enabled code re-use of up to 90

Several demonstration applications were implemented to demonstrate that the platform can be re-programmed to interoperate with several different wireless communications systems. These demonstrations included and AMPS receiver, FM radio, walkie talkie, CB radio, television audio, and the reception of black and white television video signals. In addition, a *virtual patch* application was implemented that enabled an analog cordless phone (49 MHz, FM modulation, 15 kHz channels) to communicate with a CB radio (26 MHz, AM modulation, 10 kHz channels) with a latency of less than 100 msec.

The portability of the system was demonstrated by moving the entire system, including the DMA hardware engine, operating system extensions and the programming environment, from the Pentium to an alpha based platform.

Traditionally, the physical layer has encompassed a wide range of functions, including coding, modulation, and multiple access. In order to capture the wide range of functionality, promote software reuse, and enable dynamic incremental modification of the physical layer it is necessary to have a framework for organizing the various functions. The software radio layering model was proposed as such a framework, and it provides a guide for the structuring of both signal processing modules and applications in the future.

## 9.1 Future Work

The flexibility demonstrated by the virtual radio opens a wide range of future research areas. Other research groups working on compilers, parallel hardware and memory architecture have used some of the virtual radio software as a test application to better understand the requirements of these types of signal processing applications. Three interesting opportunities are described below.

### 9.1.1 Wireless Networking

Today's network protocols and applications are designed for a relatively static environment in which link characteristics, topologies, and trust relationships are configured once and then fixed for long periods. This is appropriate for a wired infrastructure. In a wireless network, however, channel conditions can vary rapidly and tend to be difficult to predict, especially when nodes are mobile [Nguyen *et al.*, 1996]. This is problematic because it leads to designs that are based on the worst case that can be tolerated, rather than the channel conditions that are currently being encountered. The result is inefficient use of the available spectrum.

A network based on software radio technology enables a much more dynamic organization of resources [Bose and Wetherall, 1999]. Software radios allow the characteristics of all communication layers, including the physical layer that is normally implemented in hardware, to be changed at essentially any time. While the motivation behind their development has been to solve the interoperability problems caused by different cellular standards, we believe that this technology is well-suited to wireless networking.

Recent research has shown that adaptive link layer technologies [Lettieri *et al.*, 1997] can significantly enhance wireless network performance. A fully-programmable software radio can incorporate adaptive link layer techniques and extend the adaptability to the physical layer. This enables more effective use of the spectrum by dynamically adapting the physical layer of the network to best meet the current environmental conditions, network traffic constraints and application requirements, rather than a lowest common denominator service that must accommodate the worst case. This adaptation has the potential to significantly improve the performance of wireless networking systems, as well as enhance their functionality by taking into account different application requirements for bandwidth, latency, error rate, and security. For example, in a basestation to mobile system, the basestation can dynamically create channels depending upon the number of mobile units in its coverage area and their particular service requirements. When additional mobile units enter the area, the bandwidth can be appropriately apportioned to each unit. Units requiring real-time or high data rate services may be assigned a dedicated channel customized to their application, while others with bursty data requirements might be assigned to a shared channel. Furthermore, bandwidth can be apportioned to upstream and downstream channels depending on application needs. These kind of adaptations are typically not possible with conventional wireless networks.

In order to realize these many opportunities there are several research challenges that must be addressed, including:

- The collection of information about the current state of the channel.
- Policies for determining the appropriate modifications to the radio.
- Mechanisms for executing the changes to a network node, and synchronizing them across multiple nodes.
- Development of routing algorithms that incorporate information from the physical layer.
- An evaluation of the effect that a dynamic infrastructure might have on existing network protocols.

### 9.1.2 Parallel Processing

Many virtual radio applications with extreme computational requirements are naturally parallelizable. For example, a software cellular basestation would exceed the processing capacity of any existing or soon to be available processor. Fortunately, the processing of multiple channels can be parallelized in a straight forward manner. However, there are significant challenges to using the SPECtRA approach in a parallel system. The most challenging is preserving the benefits of lazy evaluation without introducing a serial dependency on the computation. One possible mechanism which is currently being explored is to allow modules to process data that has not yet been requested, and to treat this as speculative execution because parameters on the computation may change before the data is actually requested.

Related to parallel processing is the notion of distributed signal processing. Some work on distributed capture and processing of wideband RF signals was investigated early on in the project [Bose *et al.*, 1997] in the context of an ATM network. The system had the ability to capture and digitize data at remote sites on the network and transfer this data across the network from processing at a remote node or nodes. This work investigated the most basic form of distributed processing, where the capture and processing of data could be in geographically distinct locations. To take advantage of parallel processing in a distributed environment would require an examination of the scheduling of computation that would include the cost of transferring data around the network. Data intensive processing applications cannot be migrated without thought to the the bandwidth available to transfer the data as well as the availability of computational resources.

### 9.1.3 Filter Design

The performance evaluation of the multi-rate channel selection filter demonstrated that costs associated with a software environment can be very different form the costs associated with digital hardware implementations. This suggests that, to improve the computational efficiency of software implementations, new filter design techniques must be developed.

In order to develop a new design methodology, it is first necessary to understand and model the costs associated with the system. Issues such as locality of data reference, memory write costs and parallelism must be taken into account. The wide variation in processor architectures suggests that to truly optimize performance, the costs for a particular processor would need to be used. However, the performance results presented in this dissertation suggest that there are a few primary costs, which may be able to be modeled more generally. This could lead to a design methodology that provides reasonably efficient implementations, which can be hand tailored if they are the primary bottleneck for a given application.

## 9.2 Summary of Contributions

The major contributions of the work presented in this dissertation are:

- A demonstration that is is possible to implement high data-rate, computationally intensive real-time signal processing applications on a general purpose platform.
- The design of a system architecture for wideband software radios. The flexibility of this system, coupled with wideband digitization enables wireless communications systems that make better use of the available spectrum and provide better application level performance than existing systems.
- The development of the data-pull model for execution of computation in a modular signal processing system.
- The characterization of the computational variability of signal processing algorithms on a general purpose processor.
- The design and implementation of a programming environment to support real-time signal processing applications on a general purpose platform.
- Design guidelines for the implementation of real-time systems on a general purpose processor and the implementation of physical layer functions in software.
- An illustration of some areas in which the design of software signal processing algorithms differs from design for a hardware or DSP implementation.
- A layered model for the specification of physical layer of communications systems .

## 9.3 Conclusion

While the SpectrumWare software radio system has proven to be an excellent design and development platform, considerable work must be done to support software radio capabilities in a mobile, hand-held device. Solutions such as reconfigurable computing, advanced A/D technology and methods for controlling the power dissipation of processors hold the promise of making hand-held software radios a reality.

Fortunately, research and development of software radio applications does not have to wait for the development of these technologies. The SpectrumWare software radio system provides an excellent prototyping and measurement platform which will permit the development of software radios applications which can then be ported to hand-held devices as they emerge.

# Appendix A

# A/D Conversion for Wideband Software Radios

In order to perform all of the signal processing, including the multiple access protocol, in software it is necessary to digitize a signal containing multiple channels. For example, software access to any channel in the AMPS cellular system would require digitizing a 12 MHz wide portion of the 800 MHz band. The A/D and D/A conversion of a wideband RF or IF signal enables the software processing, but does have a cost: the noise introduced by the converter can significantly limit the capabilities of the system.

For the case of generic wireless communications, we can make a few assumptions about the input signal that simplify the analysis. Since the input signal could represent any modulation, coding and access scheme, we assume that uniform quantization is the best approach. For a specific application, it may make sense to use a non-uniform scheme, such as A-law or $\mu$-law. however, since the goal is to develop the most flexible platform possible, no assumptions are made that might optimize for one case at the expense of another. Furthermore, we can assume that the input signal is uncorrelated to the sampling clock, which is quite reasonable given that there is no synchronization mechanism between the transmitter and receiver at the sample timing level.

The process of analog-to-digital conversion introduces several sources of error to the signal. The performance of the converters can be summarized by a few parameters: resolution (number of output bits, $B$), signal-to-noise ration (SNR), spurious-free dynamic range (SFDR) and intermodulation distortion (IMD). The impact of these various sources on system performance is application dependent. This section first reviews these converter parameters, and then analyzes the affect of these parameters on several important wireless communications applications.

The SNR is the ratio of the RMS signal amplitude to the square root of the integral of the noise power spectrum over the frequency band of interest. For signals with broad frequency content (e.g. a section of the spectrum containing many channels) the quantization noise is well approximated by white noise. The quantization noise has been well characterized, and the resulting signal-to-nosie ratio for a $B$ bit Nyquist converter is:

$$SNR_q = 6.02B + 1.76 \ dB$$

There are several other error mechanisms, such as aperture uncertainty, circuit noise and comparator ambiguity, which can also be characterized as white. The parameter $SNR_{bits}$ can be used to characterize all of these source with one measurement [Walden, 1999]. The difference between the stated number of bits and the $SNR_{bits}$ is a measure of the degradation in SNR due to all of the converter noise sources.

$$SNR_{bits} = (SNR - 1.76)/6.02$$

where $SNR$ is expressed in dB.

The SNR due to quantization can be improved by oversampling the incoming signal (i.e. sampling at a rate $f_s$ that is greater than twice the highest frequency component present in the signal $f_{max}$). Intuitively, this spreads out the white quantization noise over a larger band, reducing the power of the noise that interferes with the signal of interest.

The improvement in SNR due oversampling is a form of processing gain. Using Nyquist sampling, there is no processing gain, since the bandwidth of the signal is the same as the bandwidth of the sampling signal. Incorporating this term into the expression for the quantization SNR yields [Wepman, 1995]:

$$SNR = 6.02B + 1.76 + 10\log_{10}\left(\frac{f_s}{2f_{max}}\right) \ dB$$

Oversampling provides a mechanism to trade sampling rate against the number of bits of precision. This is useful since available ADC technology permits the construction of very fast low resolution (i.e. 8 bits or less) sampling, but the sampling rate drops of dramatically as the number of bits increases, as illustrated in the current snapshot of ADC technology presented in table A.1. Taking oversampling to the extreme, we could achieve any desired SNR with one bit sampling at a sufficiently high rate.

An important parameter to keep in mind for software radio applications is the *bit rate* of the sample stream. The I/O, memory and computation sub-systems of a processing system are often specified in terms of the bit rate that they can handle. This is the appropriate characterization, because the various subsystems handle data in units that may have little to do with the sample size. For example, bus transfers are often in units of 32. By specifying the sample stream in terms of its bit rate, the ability for each subsystem to handle the data can be assessed independent of the sample size. One caveat to this is that for many systems, units in multiples of bytes are handled more efficiently. In these cases it is more useful to talk about the sample rate rounded up to the nearest byte rate. In many systems significant efficiency can be gained through packing bits into bytes, and this becomes the over-riding factor.

Oversampling, places a bigger burden on the subsystems, because it increases the bit rate.

Furthermore, if the SNR were kept constant, and bits of precision were traded for a higher sampling rate, the overall bit rate would increase. For example, a 20 MSPS bit stream with 12 bits of precision requires the subsystems to handle 240 Mbps. Reducing the precision to 11 bits, and quadrupling the sample rate to keep the SNR constant, requires a bit rate of 880 Mbps for the same information. Furthermore, in terms of the byte rate it is even worse, since both 11 and 12 bit samples would get packed into 16 bit words, so the required byte rate would quadruple. When bytes are the relevant metric, the increase in I/O requirements are less, but still significant. For example a 20 MSPS stream of 9-bit samples carries the same SNR as an 80 MSPS stream of 8-bit samples, but the byte rate only doubles because the 9-bit samples require two bytes of storage whereas the 8-bit samples require only a single byte.

In general, the burden on the processing system is minimized if the highest precision is used with the lowest possible sampling rate necessary to achieve the desired SNR. For many wideband applications, such as those using frequency division, wideband sampling naturally over samples the channel of interest.

Converters also have non-linear sources of error, which are characterized by the spurious free dynamic range (SFDR). The SFDR is the ratio of the signal power to the peak power of the largest spurious (i.e. not in the signal) component of the frequency spectrum within the band of interest. It is often measured by using a sinusoidal input to the ADC and measuring the largest spurious component by performing an FFT on the converter output.

The SFDR is an important parameter for digital receivers. It is particularly useful when the desired bandwidth is less than than half of the sampling frequency. In this case, a considerable portion of the converter's white nosie contributions can be filtered out, but the non-white spurious components remain. These may be larger than the remaining component of the white noise. Thus the SFDR may be worse than the SNR and limit the theoretical performance of the system. The SFDR is also relevant when oversampling is used, as it upper bounds the improvement in SNR that can be realized.

The noise power ratio (NPR) is a very useful measure for applications that require digitizing many narrowband channels that are close in amplitude, such as found in conventional frequency division systems like broadcast FM or the AMPS cellular system. The NPR provides information on how well the ADC can limit cross-talk between channels. It is measured by using a noise input to the converter that is flat, except for a narrow frequency band that is notched out. This notch is typically slightly wider than the width of the channel of interest. The ratio of the total noise power to the power folded into the notched band of the converter signal is the NPR.

Another important specification is the intermodulation distortion. This occurs when there are two large signals in the presence of multiple smaller signals, and the two large signals are mixed together due to non-linearities in the conversion process. The result of this mixing is spurs that may reside in bands occupied by the smaller signals [Brannon, 1998b] [Frerking, 1994].

Unfortunately, the specification of A/D converters has not been standardized, and typically only a few of the specifications relevant to software radio applications can be found for a particular converter. The specifications for several commercially available state-of-the-art high-speed converters are given in table A.1.

| Converter | Max. $f_s$ | Spec. $f_s$ | bits | eff. bits | SFDR | NPR | IMD |
|---|---|---|---|---|---|---|---|
| AD6640 | 65 MSPS | 65 MSPS | 12 | 10.9 | 85 dB | N/A | 80 dBc |
| AD9042 | 41 MSPS | 41 MSPS | 12 | 11.0 | 80 dB | 70 dB | 91 dB |
| AD9070 | 100 MSPS | 100 MSPS | 10 | 9.2 | N/A | N/A | 70 dB |
| MAX101A | 500 MSPS | 500 MSPS | 8 | 7.1 | N/A | N/A | N/A |
| SPT7760 | 1 GSPS | 250 MSPS | 8 | 5.5 | 39 dB | N/A | N/A |

Table A.1: Specifications for several state-of-the-art commercially available converters. $f_s$ is the maximum sampling frequency, *Spec.* $f_s$ is the sampling frequency used to measure the parameters, *bits* is the actual resolution of the converter, *eff. bits* is the effective number of bits after the noise sources have been taken into account, *SFDR* is the spurious free dynamic range, *NPR* is the noise-power ratio and *IMD* is the inter-modulation distortion.

This appendix reviewed the A/D parameters relevant to software radio applications and demonstrated that it is feasible to digitize widebands of down converted RF spectrum and stream these samples into memory that can be accessed by application software. The scope of applications that can be implemented is limited by the front end and D/A conversion technology, and with current technology it is possible to implement a wide range of useful and interesting wireless applications.

# Bibliography


[Abbott and Peterson, 1993] Mark B. Abbott and Larry L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *ACM Transactions on Networking*, 1(5):600–610, October 1993.

[Akos, 1997] Dennis M. Akos. *A Software Radio Approach to Global Navigation Satellite System Receiver Design*. PhD thesis, Ohio University, May 1997.

[Ana, 1994] Analog Devices, One Technology Way, P.O. Box 9106, Norwood, MA. *ADSP-21060/62 SHARC Preliminary Data Sheet*, November 1994.

[Bier *et al.*, 1997] Jeffrey C. Bier, AMit Shoham, Harri Hakkarainen, Ole Wolf, Garrick Blalock, and Philip D. Lapsley. DSP on General Purpose Processors. Technical report, Berkeley Design Technology Inc., 39355 California St., Suite 206, Fremont CA, 1997.

[Blust, 1995] Stephen M. Blust. Software Defined Radio - Industry Request for Information. Technical report, Bell South Cellular, December 1995.

[Bose and Wetherall, 1999] Vanu G. Bose and David J. Wetherall. Radioactive networks: Freedom from worst case design. *Mobicom'99*, August 1999.

[Bose *et al.*, 1997] Vanu G. Bose, Andrew G. Chiu, and David L. Tennenhouse. Virtual Sample Processing: Extending the Reach of Multimedia. *Multimedia Tools and Applications*, 5(1), 1997. to appear.

[Bose *et al.*, 1999] Vanu G. Bose, Michael Ismert, Matthew Welborn, and John Guttag. Virtual Radios. *JSAC issue on Software Radios*, February 1999.

[Brannon, 1998a] Brad Brannon. Digital-radio-receiver design requires re-evaluation of parameters. *EDN*, pages 163–170, November 1998.

[Brannon, 1998b] Brad Brannon. Wideband Radios Need Wide Dynamic Range Converters. *Analogue Dialogue*, 29(2), 1998.

[Brown and Wolt, 94] Alison Brown and Barry Wolt. Digital L-Band Receiver Architecture with Direct RF Sampling. In *IEEE Postition Location and Navigation Symposium*, pages 209–216, April 94.

[Chechire and Baker, 1997] S. Chechire and M. G. Baker. Consistent Overhead ByteStuffing. In *Proc. SIGCOMM'97*, September 1997.

[Clark and Tennenhouse, 1990] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *ACM SIGCOMM '90*, pages 200–208, Philadelphia, PA, September 1990.

[Dick, 1998] Chris Dick. FPGA's for High Performance Communications. In *International Symposium on Advanced Radio Technologies*, 1998.

[DiPiazza *et al.*, 1979] G. C. DiPiazza, A. Plitkins, and G. I. Zysman. AMPS: The Cellular Test Bed. *Bell System Technical Journal*, 58(1):215–248, January 1979.

[Engler, 1996] Dawson Engler. vcode: a portable, very fast dynamic code generation system. In *SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.

[Frerking, 1994] Marvin E. Frerking. *Digital Signal Processing in Communication Systems.* Van Nostrand Reinhold, 1994.

[Frigo and Johnson, 1998] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive Software Architecture for the FFT. *ICASSP*, 3:1381, 1998.

[Guernic *et al.*, 1986] Paul Le Guernic, Albert Benveniste, Patricia Bournai, and Thierry Gautier. SIGNAL - A Data Flow-Oriented Language for Singal Processing. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-34(2):362–374, April 1986.

[Gutnik and Chandrasakan, 1996] Vadim Gutnik and Anantha Chandrasakan. An Efficient Controller for Variable Supply-Voltage Low Power Processing. In *VLSI'96*, 1996.

[Houh *et al.*, 1995] Henry H. Houh, Joel F. Adam, Michael Ismert, Christopher J. Lindblad, and David L. Tennenhouse. The VuNet Desk Area Network: Architecture, Implementation, and Experience. *IEEE J-SAC*, 13(4):710–721, May 1995.

[IEEE, 1997] IEEE. IEEE Std 802.11-1997. Technical report, IEEE, 1997. Working Group for Wireless Local Area Networks.

[Intel, 1998] Intel. Intel Architecture Optimiztions Manual. http://developer.intel.com/design/pro/manuals/, 1998.

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

[Jensen, 1997] Douglas Jensen. Eliminating the 'hard'/'soft' real-time dichotomy. *Computing & Control Engineering Journal*, 8(1):15–19, February 1997.

[Katz and Brewer, 1996] Randy H. Katz and Eric. A. Brewer. The Case for Wireless Overlay Networks. In *Proceedings 1996 SPIE Conference on Multimedia and Networking*, San Jose, CA, January 1996. MMCN '96.

[Lackey and Upmal, 1995] Raymond J. Lackey and Donal W. Upmal. Speakeasy: The Military Software Radio. *IEEE Communications Magazine*, 33(5):56–61, May 1995.

[Lee and Messerschmitt, 1994] Edward A. Lee and David G. Messerschmitt. *Digital Communication.* Kluwer Academic Publishers, 2nd edition, 1994.

[Lettieri *et al.*, 1997] P. Lettieri, C. Fragouli, and M. Srinastava. Low Power Error Control for Wireless Links. In *Mobicom97*, Budapest, Hungary, September 1997.

[Lindblad *et al.*, 1994] Christopher J. Lindblad, David J. Wetherall, and David L. Tennenhouse. The VuSystem: A Programming System for Visual Processing of Digital Video. In *Proceedings of ACM Multimedia*, 1994.

[Lindblad, 1994] Christopher J. Lindblad. A Programming System for the Dynamic Manipulation of Temporally Sensitive Data. Technical Report MIT/LCS/TR-637, M.I.T., August 1994.

[McCanne *et al.*, 1997] Steven McCanne, Eric Brewer, Randy Katz, Lawrennce Rowe, Elan Amir, Yatin Chawathe, Alan Coopersmith, Ketan Mayer-Patel, Suchitra Raman, Angela Schuett, David Simpson, Andrew Swan, Teck-Lee Tung, and David Wu. Toward a Common Infrastructure for Multimedia-Networking Middleware. In *Proceedings of the 7th InternaProceeding of NOSSDAV'97*, St. Louis, Missouri, May 1997. Invited Paper.

[Mitola, 1995] Joe Mitola. The Software Radio Architecture. *IEEE Communications Magazine*, 33(5):26–38, May 1995.

[Mitola, 1999] Joe Mitola. Software Radio Architecture A Mathematical Perspective. *JSAC issue on Software Radios*, 1999.

[Mogul and Ramakrishnan, 1997] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Drive Kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[Motorola, 1997] Motorola. Low-Cost, Powerful Software Modem Solution. Press Release, December 1997.

[Mouly and Pautet, 1992] Michel Mouly and Marie-Bernadette Pautet. *The GSM System for Mobile Communications.* Cell & Sys, 4,rue Elisee Reclus, F-91120 Palaiseau, France, 1992.

[Nguyen *et al.*, 1996] G. T. Nguyen, R. H. Katz, B. D. Noble, and M. Satyanarayanan. A Trace-based Approach for Mofdelling Wireless Cahnnel Behavior. In *Winter Simulation Conference*, December 1996.

[Patterson *et al.*, 1997] David Patterson, Krste Asanovic, Aaron Brown, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberley Keeton, Christoforos Kozyrakis, David Martin, Stylianos Perissakis, Randi Thomas, Noah Treuhaft, and Katerine Yelick. Intelligent RAM (IRAM): the Industrial Setting, Applications, and ARchtectures. In *ICCD'97 International Conference on Computer Design*, Austin, Texas, October 1997.

[Press *et al.*, 1993] William H. Press, Saul A. Teulolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C : The Art of Scientific Computing.* Cambridge Univ Press, 2nd edition, 1993.

[Rabaey, 1997] Jan M. Rabaey. Reconfigurable Computing: The Solution to Low Power Programmable DSP. In *1997b ICASSP Conference*, Munich, April 1997.

[Rissanen, 1997] K. Rissanen. Challenges in Software Radio. In *Software Radio Workshop*, Brussels, Belgium, May 1997. European Commission, DG XIII-B.

[Stewart *et al.*, 1992] Lawrence C. Stewart, Andrew C. Payne, and Thomas M. Levergood. Are DSP Chips Obsolete ? Technical Report CRL 92/10, Cambridge Research Labratory, Cambridge, MA, 1992.

[Tanenbaum, 1988] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, Englewood Cliffs, NJ 07632, second edition, 1988.

[Taylor, 1997] Carl Taylor. Flexible Integrated Radio Systems. In *Software Radio Workshop*, BVrussles, Belgium, May 1997. European Commission, DG XIII-B.

[Tennenhouse and Bose, 1996] David L. Tennenhouse and Vanu G. Bose. The SpectrumWare Approach to Wireless Signal Processing. *Wireless Network Journal*, 2(1), 1996.

[Turletti, 1996] Thierry Turletti. A Brief Overview of the GSM Radio Interface. Technical Report TM-547, MIT, March 1996.

[Tuttlebee, 1999] Walter H Tuttlebee. Software Radio Technology: A European Perspective. *IEEE Communications Magazine*, February 1999.

[Walden, 1999] R. H. Walden. Analog-to-Digital Converter Survey and Analysis. *JSAC issue on Software Radios*, 1999.

[Welborn, 1999] Matthew L. Welborn. Narrowband Channel Extraction for Wideband Receivers. In *ICASSP'99*, 1999. to appear.

[Wepman *et al.*, 96] J.A. Wepman, J. R. Hoffman, and J. E. Schroeder. An initial study of RF and IF digitization in radio recivers. Technical report, NTIA, 96. in preparation.

[Wepman, 1995] Jeffery A. Wepman. Analog-to-Digital Convertors and Their Applications in Radio Receivers. *IEEE Communications Magazine*, 33(5):39–45, May 1995.

[Young, 1982] C. J. Young. *Real Time Languages: Design and Development.* Ellis Horwood, 1982.