To appear in JSAC issue on software Radios, 4th Qtr., '98

Virtual Radios

Vanu Bose, Mike Ismert, Matt Welborn, John Guttag *

Software Devices and Systems Group
Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

Conventional software radios take advantage of vastly improved A/D converters and DSP hardware. Our approach, which we refer to as virtual radios, also depends upon high performance A/D converters. However, rather than use DSPs, we have chosen to ride the curve of rapidly improving workstation hardware. We use wideband digitization and then perform all of the digital signal processing in user space on a general purpose workstation. This approach allows us to experiment with new approaches to signal processing that exploit the hardware and software resources of the workstation. Furthermore, it allows us to experiment with different ways of structuring systems in which the radio component of communication devices are integrated with higher-level applications.

This paper describes the design and performance of an environment we have constructed that facilitates building virtual radios and of two applications built using that environment. The environment consists of an I/O subsystem that provides high bandwidth low latency user-level access to digitized signals and a programming environment that provides an infrastructure for building applications. The applications, which exemplify some of the benefits of virtual radios, are a software cellular receiver and a novel wireless network interface.

1 Introduction

A *virtual radio* is a communications device that:

> *Does all its digital signal processing in user space on an off-the-shelf workstation (a PC in our case).*

These devices are quite different from most software radios, which typically are implemented using either application-specific digital hardware or digital signal processors under software control [BW94] [LU95].

The SpectrumWare project is devoted to building infrastructure to support the construction of virtual radios and to building virtual radios that take advantage of the resources available on the workstation to either provide distinctive functionality or to implement traditional functionality in a distinctive way.

Of course, using a general purpose workstation to do signal processing can impose a significant cost. Not only are most DSPs considerably cheaper than workstations but they are also considerably smaller and use less power. However, exploiting the resources available on the workstation offers many potential advantages:

- *Experimentation.* There is a large and growing gap between the capabilities and programming environments available on general purpose workstations and those available on specialized DSP hardware.

* {vanu,izzy,mwelborn,guttag}@lcs.mit.edu

Implementing the signal processing on the workstation makes it much easier to experiment with new algorithms and protocols.

- *Rapid deployment.* Users can easily deploy new devices or enhancements to existing devices in the same manner in which they currently upgrade device drivers and software installed in their workstations, for example, as a self-extracting archive downloaded from an ftp site.
- *Integration with other applications.* It is often the case that functionality provided by a radio device is only a small part of a larger application. The functionality provided by a wireless modem, for example, is only a part of what is needed for a network interface. Our approach to virtual radios allows one to blur the line between the radio and the rest of the application, thus allowing improved functionality and end-to-end efficiency.
- *Multi-purpose devices.* Increasingly, people are dealing with multiple wireless communication devices, e.g., they have both a cellular modem and a wireless ethernet for their notebook computer (not to mention more mundane devices such as garage door openers). While multi-purpose DSP devices, e.g., FAX modems, are becoming increasingly available, they perform a relatively small number of pre-defined functions. In our approach, one adds devices merely by downloading software.
- *Reduced cost (for special purpose devices).* While general purpose computers are never likely to be less expensive than specialized hardware for mass market items (e.g., cell phones) they are often less expensive than the hardware used in devices with small markets. Furthermore, our system architecture encourages sharing of hardware resources across devices. These devices need not even be what we conventionally think of as radios. We have, for example, investigated using the technology developed for our virtual radios to implement the signal processing done in medical ultrasound machines with the idea of allowing a number of ultrasound frontends to share the same backend workstation [Tha97].
- *Improved Functionality* Performing all of the signal processing in modular software permits not only the dynamic assignment of channel locations and widths, but also of the modulation and coding used on each channel. This allows for the construction of heterogeneous systems which employ different communications standards on different channels. This mechanism can be used to accelerate the migration to new standards.
- *Improved performance.* This is a bit surprising. Why should it be possible to get better performance running on a general purpose processor than on a specialized DSP? A partial answer is that market factors are driving rapid improvements in workstation performance. However, there are three other key factors involved:
  - The temporal decoupling afforded by the workstation's memory and the flexibility of the workstation's processor allows one to implement faster algorithms,
  - The ability to integrate the signal processing with higher-level applications affords the opportunity for system-wide optimization, and
  - The signal processing functions can be dynamically modified based on measurements of changing channel or system characteristics in order to improve performance. This information could come from background tasks that perform functions such as channel estimation when spare cycles are available.

The next section of this paper describes an environment for building virtual radios that run on off-the-shelf personal computers. This environment facilitates the construction of devices that enjoy many of the potential benefits listed above. It has two key parts: an I/O subsystem that provides high bandwidth low latency user-level access to digitized signals, and a programming environment that provides an infrastructure for building applications. We discuss the I/O subsystem in some detail, and sketch some of the key properties of the software environment.

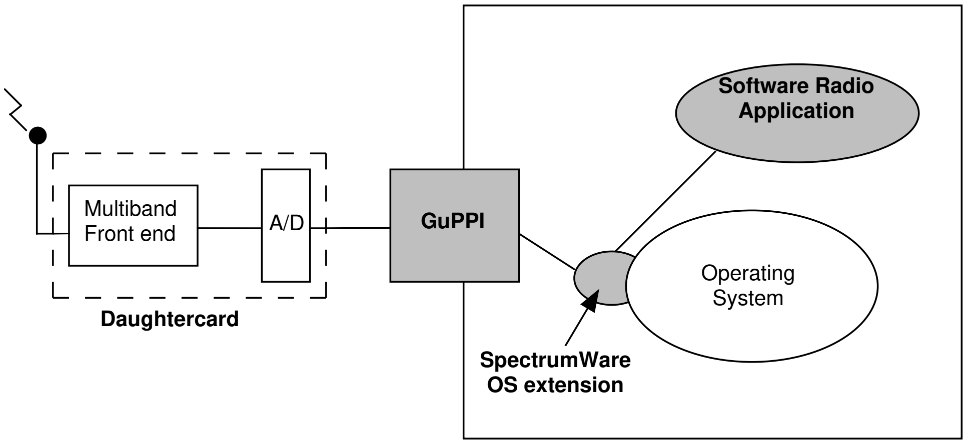

Figure 1: Block diagram of our virtual radio system. The hardware is used only to convert the desired RF band down to the IF frequency, digitized it and stream the samples into host memory. All subsequent processing of this digital wideband IF signal is performed in user-level software.

Section 3 describes the design and reports on the performance of two applications built using our environment. The applications are relatively simple, yet they illustrate many of the benefits of our approach to virtual radios. Section 3.1 describes a software cellular receiver and Section 3.2 a software wireless network interface.

We conclude the paper by relating the specific results reported in the paper to the more general assumptions behind our work on virtual radios.

2 Enabling Technology

Our goal is to move the analog/digital boundary as close to the antenna as possible, and to move the software/hardware boundary right up to the A/D converter. Since current A/D technology and available processors will not support the direct sampling of wide RF bands, our approach, as illustrated in figure 1, is to use a multi-band hardware frontend to convert the desired RF band to the IF frequency, then directly sample the wideband IF waveform and transfer these samples into host memory. All subsequent processing is performed in user level software.

At present, the multi-band frontend is the missing link in our system. Existing commercial frontends do not meet all of our needs, hence we are currently using different receivers for different sub-bands. However, several vendors are working on wideband receivers that are likely to be available in the next year or so.[1]

Once we have the digital samples, we are faced with the somewhat daunting task of transporting the samples into host memory. Traditionally, I/O devices have a device driver resident in the kernel which receives and processes the data, and then copies it into user space. However, the performance problems associated with using the default Unix I/O system to move data across the kernel/user boundary are well known, and the extra time required is unacceptable for our real-time signal processing applications. To avoid the expense of performing the data copy, previous research efforts have relied on different schemes using virtual memory manipulation and/or shared memory [DP93, CP94, And95, vEBBV95, BS96, Pai97]. Our solution to this problem was to use background direct memory access (DMA) to stream the samples directly into special buffers in the kernel; these buffers are mapped into the user's address space using virtual memory manipulations with very little overhead. To support this functionality, we developed the general purpose

[1]Such receivers are currently under development by Rockwell and Hughes Electronics.

PCI I/O system described below [Ism98]. This interface utilizes the gigabit capacity of the PCI bus, and supports continuous bidirectional I/O streams of up to 32 megasamples/second (MSPS).

We have also developed a programming environment for constructing applications. This environment consists of a library of portable (across platforms) signal processing routines designed to support a data-pull style of programming and various kinds of dynamic adaptation.

In the next subsection we describe the design of I/O system and report on its performance. In the following subsection we give a quick overview of the architecture supported by the programming environment.

2.1 The I/O System

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 our I/O system.

The other major requirement which 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 frontend 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 [CHT95] 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 frontend to the workstation's I/O bus, and operating system additions, which provide the means for the application to access the sample streams.

2.1.1 I/O System Architecture

The GuPPI provides the system's external interface to the analog frontend. This interface has two requirements. First, it must accept samples from and provide samples to the analog frontend at the sampling rate. Second, it must be easy to design to, otherwise the GuPPI provides no advantage in terms of connecting to the I/O bus.

The GuPPI provides the ability to burst data between the analog frontend 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 frontend 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; the required processing overhead per sample is less than half a cycle.

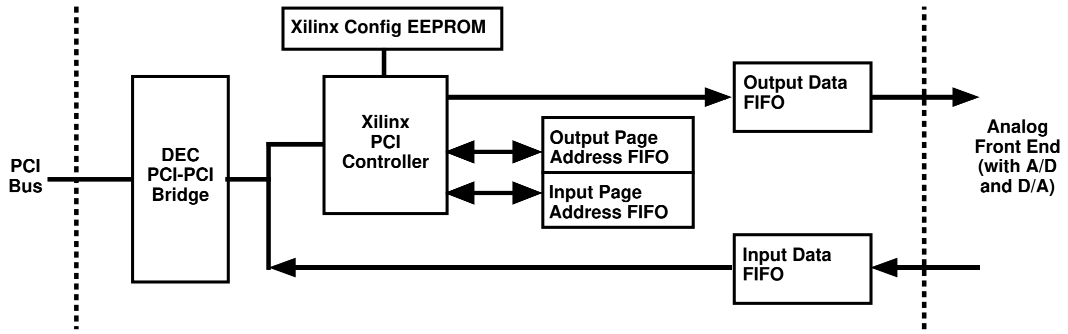

Figure 2: GuPPI Block Diagram.

A block diagram of the GuPPI implementation is shown in figure 2. The GuPPI has a simple, generic daughter card interface to which analog frontend-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 which 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 which 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, all for the Linux kernel[2]. 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. These components also provide 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. 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 stream.

[2]The Linux kernel version used is 2.0.30.

Input	Output
933	790

Table 1: Raw GuPPI Performance (Mbits/sec)

2.1.2 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 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 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.

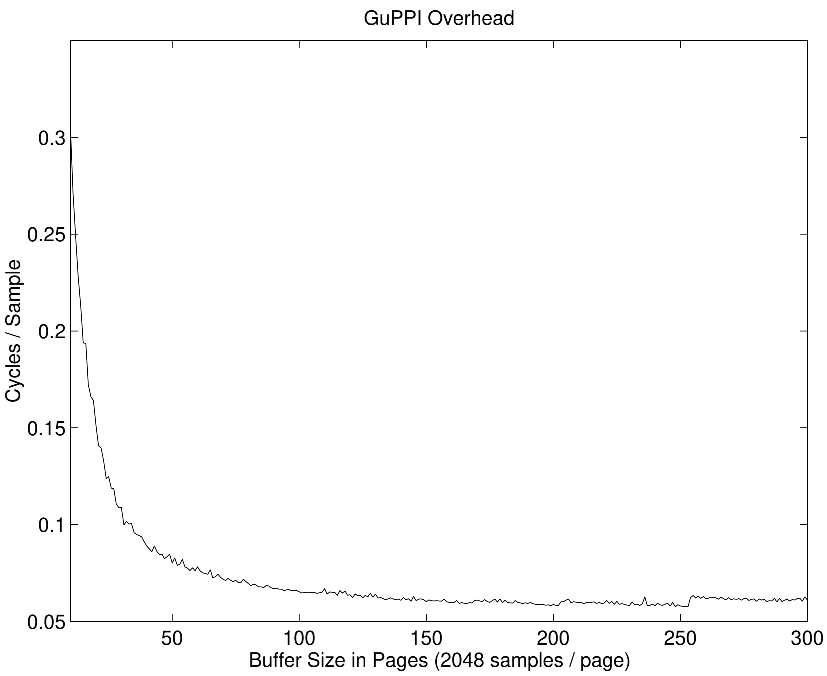

Figure 3: GuPPI Processing Overhead (Input)

Figure 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 interrupts generated by the need to replenish the supply of input buffer pages[3].

2.1.3 Discussion

The GuPPI hardware provides a high-bandwidth, low-latency connection between an analog frontend and the I/O bus of a PC. The software drives the GuPPI, smoothes jitter, and makes the data transferred by the hardware accessible to applications in user space. Together, the hardware and software provide an application-level interface that simplifies the construction of virtual radios and related applications by making the analog frontend appear to be a conventional Unix device. This characteristic simplifies experimentation and rapid deployment by hiding the details of the operation of the GuPPI behind a standard interface while still providing high performance. In addition, the use of the PCI bus, a widely-used industry standard, permits our system to be portable to a wide range of workstations and allows us to easily get increased performance from newer, faster processors.

The average processor overhead attributable to the GuPPI is less than half a cycle per sample. For large buffers it is less than a tenth of a cycle per sample. The maximum sustainable receive data rate visible to applications is 512 Mbits/sec. This is more than enough for most of the applications for which the system was designed. The two applications discussed in the next section, an software cellular telephony receiver and a software wireless network interface, are not I/O-limited but processor-limited instead.

2.2 Programming Environment

This section briefly describes the architecture of SPECTRA, a programming environment that supports the construction of portable adaptive signal processing systems with real-time constraints.

The design of SPECTRA was heavily influence by our experience building and using the VuSystem [LT96]. The VuSystem, which was built to support multimedia applications running over a desk area network, provides considerable support for composing signal processing modules. This made it an excellent environment in which to conduct preliminary experiments with virtual radios. However, as we came to understand better the opportunities that virtual radios afford for adaptive signal processing, we realized that we needed a more flexible programming environment.

The SPECTRA environment is designed to support several different notions of adaptive signal processing:

- **Adaptation to the environment**: For example, a modem equalizing for the channel. This is what is most commonly thought of as adaptive signal processing. It requires algorithms to detect environment changes (such as a channel estimation algorithm) and a mechanism for specifying system modifications based on the detected changes.
- **Adaption to the user**: The users requirements may change, necessitating certain changes in the system. A simple example of a conventional system adapting to the user is changing the station on radio. This causes changes in certain parameters of the system. A more sophisticated example is pushing the AM/FM button, which requires different algorithms to be inserted into the system.
- **Functional Adaptation**: Many signal processing techniques, such as a phase-locked loop, are inherently adaptive. These algorithms typically adapt to the signal, and may modify system parameters and/or functionality to meet specified requirements. For example, a receiver attempting to lock on to

[3]The measurements assume 16 bit samples.

the start frequency of a hopping sequence adapts the system to examine different frequencies until a start code is found. Unlike the two types of adaptation defined previously, this type of adaptation is driven by a specified constraint, not external changes.

- **Adaptation to resources**: Having the ability to adapt to resource availability can improve system performance. For example, if there are a lot of spare CPU cycles available, then running a more computationally intensive channel estimation algorithm could lead to better overall system performance. Conversely, if the resources suddenly become scarce due to a burst of activity by other processes, the system should be able to adapt, perhaps by running less demanding algorithms and sacrificing some robustness or accuracy. This form of adaptability also enables applications to transparently improve performance as faster processors become available.

The desire to support these kinds of adaptation led us to a "data pull" architecture. While the flow of data is from input to output, the flow of control is from output to input. By placing control downstream, the architecture facilitates the construction of systems that aggressively employ lazy evaluation and adapt to the requirements of the downstream modules. The total amount of processing that needs to be done is reduced because downstream components can request just the data they need from upstream modules. Consider, for example, an audio system to which a variety of output devices can be attached. The driver for the output device knows whether it needs to request telephone quality [4] or CD quality [5] audio.

The flow of control requires an upstream communication path, which is not present in typical signal processing applications. This is implemented in the obvious way. An application is started by invoking a sink module. This module calls the immediately upstream module (or modules) requesting the needed data. That module, in turn, makes a request on the module (or modules) that lie immediately upstream from it. The one complication lies in exactly what is returned downstream.

Typically, the connections between modules in digital signal processing architectures represent a stream of samples. Usually, each module operates on a sample by sample basis, and maintains some state between samples. This approach, however, would not work in our system. Firstly, the overhead of a function call for processing each sample would, in many cases, be greater than the work required to process the sample. Secondly, calling each module in turn, and operating on only a single sample would lead to poor locality of both data and code. This in turn would lead to poor performance of both data and instruction caches, a critical factor in achieving high performance on modern processors.

Our solution is to have each module operate on blocks of data (which we refer to as *payloads*) that are large enough to take advantage of caching effects, but small enough to avoid introducing unacceptable latency. It should be noted that this approach represents a significant change from the status quo. It requires those writing signal processing applications to shift to a style of programming in which algorithms are designed to work on large blocks of data rather than on single samples. This is not without precedent. There are already some important signal processing algorithms, e.g., the fast Fourier transform, that do this. These algorithms maintain continuity of the output by maintaining state between blocks, rather than between samples.

3 Demonstration Applications

We have built several applications using the SpectrumWare system. These applications demonstrate both the feasibility and some of the advantages of our approach to building virtual radios. In this section we describe two such applications: a software cellular receiver and a software wireless network interface.

[4] 8 kHz sampling rate with 8 bit quantization
[5] 44 kHz sampling rate with 16 bit quantization

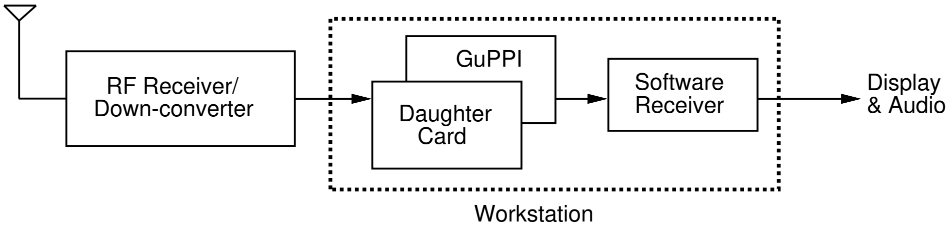

Figure 4: Cellular Receiver Block Diagram

3.1 A Software Cellular Receiver

This section presents an implementation of a wideband digital receiver designed to operate in the "A-side" of the U.S. cellular band. This receiver continuously monitors 10 MHz of the cellular band and provides the capability to demodulate FM signals anywhere in that band.

A block diagram of the cellular receiver is shown in figure 4. The frontend is a Tellabs RF receiver that translates the 825-835 MHz band to a baseband signal and then samples the signal at 25.6 MSPS. The 12-bit sample stream is fed to a GuPPI card which DMAs the samples into the main memory of the host for processing. All subsequent processing of the signals is performed in software.

3.1.1 Software Architecture

The sequence of processing steps for the wideband AMPS receiver is shown in figure 5. As one would expect for a wideband receiver demodulating a narrowband signal, the sample rate get successively lower as we move through the processing stages.

At this point we present a short description of each of the processing steps of the software receiver. It is worth noting that all of the system parameters, such as channel filter size and the various sample rates, are under software control. This allows many of these parameters to be easily modified, even while the receiver is operating.

The first step of processing is the channel selection filter. This module has the task of extracting a narrowband FM signal (AMPS channel bandwidth is 30 kHz) from a 10 MHz wide frequency band. This step is accomplished using a novel filter design that combines the three conventional steps of translating the signal to baseband, lowpass filtering and decimating to an intermediate sample rate. This step of processing comprises the largest portion of the computational load and the details of its design are given in a later section. The channel section filter uses the raw samples from the RF frontend at $R_S = 25.6$ MSPS as input and produces a complex baseband signal at a variable intermediate sample rate, R_D.

In a complex FM signal, the desired information (the voice, in this case) is carried by the instantaneous frequency, which is the time derivative of the phase of the complex signal. This signal is therefore demodulated using a simple quadrature demodulation algorithm that approximates the derivative of the signal phase by the phase difference between successive samples, appropriately scaled.

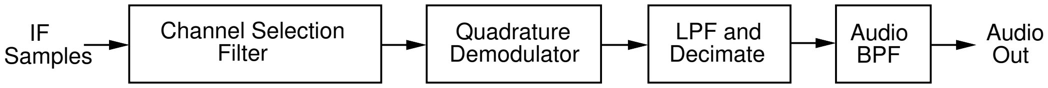

Figure 5: Software Architecture Block Diagram

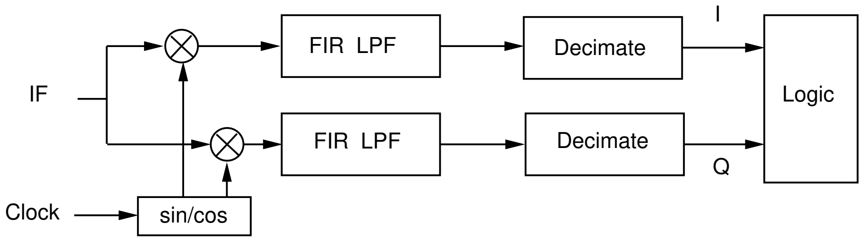

Figure 6: Dedicated Hardware Digital Down-converter

The two final steps of processing are implemented as finite impulse response (FIR) filters. The first is simply a lowpass decimating filter that removes high frequency components that would cause aliasing when the sample rate is reduced to the audio rate, R_A, typically 8 Ksamples/sec. The final step is a bandpass filter which removes out-of-band noise from the voice signal.

Separating narrowband channels in a wideband receiver is a computationally intensive task that is usually done using special purpose hardware. Figure 6 depicts a typical digital down-converter (DDC) implemented in dedicated hardware [Bai95]. The wideband signal is translated to a complex baseband signal by the quadrature multiplier and then lowpass filtered to prevent aliasing due to decimation. Special purpose FIR filters for decimation exist that do this operation very efficiently–which is important since the sample rates for a wideband receiver could be in excess of 30 MSPS. One author estimates [Bai95] that a good channel selection filter will require about 100 operations per input sample for a total of 3000 MOPS.

It is certainly possible to build software that has the same structure as a hardware DDC. However, on current workstation hardware performance would be far too slow. The limiting factor here is that the high sample rate of a wideband receiver allows time for only a few operations per sample. In hardware DDCs, the separate steps of the down-conversion process (that is, the generation of the sine/cosine multiplication factors, frequency translation, and filtering) are all done in separate physical locations in the DDC, and a high degree of pipeline parallelism is achieved. Furthermore, there is additional fine grained parallelism within the FIR filter itself. Fortunately, there is no compelling reason for our software to mimic the design of the hardware it replaces. In fact, one of the great advantages of our approach to virtual radios is that it permits us to develop and evaluate unconventional solutions to problems.

Figure 7 depicts the design of our software DDC. Here the main consideration is to perform minimal processing at the high sample rate and reduce the rate as early as possible in the processing chain. The choice of an FIR filter (versus a lower-order IIR filter) allows us to produce an output that depends only upon the filter input samples, $x[n]$, and therefore to compute only those output samples, $y[n]$, that will be required after decimation [OS89]. This lets us greatly reduce the computation load by taking advantage of the large decimation factors that will exist when processing a narrowband signal in a wideband receiver. Furthermore, the pre-computed frequency shift factors can be incorporated into a composite FIR filter and will then require

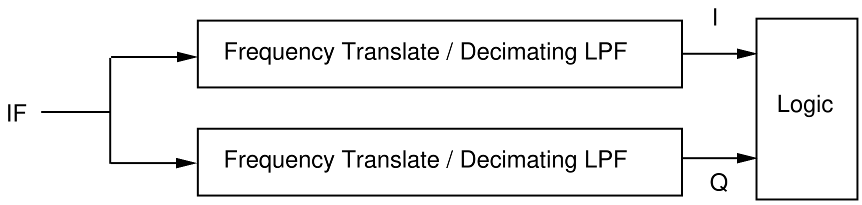

Figure 7: Proposed Software Radio Down-converter

only a phase correction after each output sample is calculated. This feature makes this a time-varying FIR filter, but all of the variation is combined into a single, time-varying multiplicative factor which is applied after the filtering.

Now we will look at the specific signal processing steps to see how the multiplication and convolution required to perform the frequency shift and filtering have been combined to produce this composite filter. The first step in the selection of a specific channel is to translate (in frequency) the real-valued received signal samples, $r[n]$, to baseband by multiplication with the appropriate complex exponential:

$$x[n] = r[n]e^{-j2\pi f_c n T_s} = r[n]\{cos(2\pi f_c n T_s) - j sin(2\pi f_c n T_s)\} \tag{1}$$

where f_c is the carrier frequency before translation to baseband and T_s is the sample interval. Next, we filter the result with the order-M FIR filter $h[m]$:

$$y[n] = \sum_{m=0}^{M} h[m]x[n-m] = \sum_{m=0}^{M} h[m]r[n-m]e^{-j2\pi f_c (n-m) T_s} \tag{2}$$

At this point we see that the two steps of frequency translation and filtering can be combined:

$$y[n] = e^{-j2\pi f_c n T_s} \sum_{m=0}^{M} h[m]r[n-m]e^{j2\pi f_c m T_s} = e^{-j2\pi f_c n T_s} \sum_{m=0}^{M} c[m]r[n-m] \tag{3}$$

where $c[m] = h[m]e^{j2\pi f_c m T_s}$ are the composite filter coefficients. Here we see that not only are we able to reduce the number of computations, but we have actually eliminated the need to compute the unfiltered baseband signal, $x[n]$, which includes the burden of writing the intermediate results to memory only to recall them in the next step. Also, while many techniques are available to design real-valued FIR filters to meet desired specifications with minimum order, we see from (3) that the use of complex-valued filter coefficients for $h[m]$ would impose no additional computational cost. For this reason we can take advantage of recent advances in the design of complex-coefficient FIR filters to reduce the required filter order M of the original LPF, relative to a real-valued $h[m]$, without increasing the required computation load for the final composite filter [OK88].

This technique also has costs, however. Although the filter requires less computation to perform the required processing steps, it is more complicated to set-up and will potentially use more memory to store filter coefficients. Because the frequency translation and filtering are combined, knowledge of the desired carrier frequency is required to compute the filter coefficients, $c[m]$. This also requires us to recompute the coefficients when we want to change the frequency to which the filter is tuned (or else pre-compute and store separate filters for any desired frequencies). Since we anticipate that each set of filter coefficients will be used many thousands or millions of times, however, the cost of pre-computing or recomputing is well worth the improved efficiency. This technique is similar to the idea of using an analytic filter to select only the positive frequency components of the real-valued signal for the passband of interest and decimating, but the combination of translation and lowpass filtering in figure 7 allows the filter to be quickly redesigned to select a different f_c.

The key improvement here over the technique of figure 6 is that we perform no operations at the high sample rate, but rather reverse the order, in a sense, and perform all of the work at the lower, post-decimation rate. In the case of AMPS, the final rate might be on the order of 100 KHz and a complex filter with several hundred taps might require less than 100 MOPS, something much more achievable.

Another interesting feature of the software cellular receiver is the ability to allow the system to dynamically optimize its own performance in a variety of ways. A particularly striking example of this is the way in which our wideband receiver does frequency-domain analysis. The receiver allows the user to perform frequency-domain analysis of either the entire band or individual signals using FFTs. Our implementation does this using an FFT library called FFTW[6] that evaluates the performance of many different FFT algorithms and selects the best one for any particular processor or data set size. The evaluation of different FFT algorithms is based either on an estimate of relative performance or the system can generate test code and actually measure the performance of different algorithms in the current environment [FJ97]. This measurement process could also be repeated periodically to see if changes in the system load indicate the use of a different algorithm to compute the FFT.

3.1.2 Discussion

This receiver was developed rapidly and requires only 600 lines of code, including the user interface. During development, standard software debugging tools were used and modifications required only seconds to re-compile and execute–a refreshing change from our experience developing hardware. We took advantage of the ease of modification to experiment with a variety of different filtering algorithms during development.

An important factor in facilitating the construction of this receiver was our ability to use off-the-shelf components, for example the FFTW package. This not only shortened the development cycle but encouraged us to be more ambitious in adding functionality.

We also took advantage of the ease with which our receiver could be integrated with application software to provide a user-friendly frontend. For example, the output of the FFT calculations is fed to standard plotting software which can graphically display any signals present in the receiver's frequency band.

In designing this receiver, we made a conscious effort to take advantage of the available computational resources to experiment with new filtering algorithms. The channel selection filter uses optimized data structures that take advantage of the large memory and computational flexibility. This novel channel filter achieves excellent performance during operation and can even be redesigned on-the-fly. The time required to recompute the filter tap weights for a different filter size or center frequency is not noticeable during the operation of the receiver.

We are now experimenting with even more radical filter designs. In particular, we are exploring the possibility of using randomized algorithms as an even more efficient way to prevent aliasing during the decimation process.

3.2 A Software Wireless Network Interface "Card"

The recent rapid growth in wireless network technology has greatly expanded the capabilities of mobile computing devices. However, the multitude of wireless network standards hinders seamless interoperability by requiring different physical devices to interoperate with different networks. Not only do wireless LANs operate in different RF bands, but even those using the same band employ different coding, modulation and network protocols. The implementation of network interface cards (NICs) in dedicated hardware limits the flexibility of these devices. Our approach to solving this problem is to implement as much of the processing as possible in software, allowing the wireless network interface functionality to be dynamically modified. Our software system provides all of the processing needed to transform between wideband IF signals and network packets.

This section presents an implementation of a software wireless network interface designed to be compatible with a commercial frequency hopping radio operating in the 2.4 GHz ISM band employing FSK modulation [Sha97].[7] Parameters such as the FSK frequency deviation and the spacing of the hopping channels can be

[6] The FFTW package was developed at MIT by Matteo Frigo and Steven G. Johnson. Additional information, including source code, is available at http://theory.lcs.mit.edu/~fftw

[7] The wireless network interface was designed to be compatible with the 2.4 GHz frequency hopping radio from GEC Plessey, model DE6003.

dynamically modified in software, the only constraints imposed by the hardware are the width of the IF band and the RF band to which the signal is converted. The ability to dynamically modify the channel width, channel spacing and the hopping sequence allows the system to adapt to its environment and provide better noise rejection and immunity from hostile jamming attacks.

3.2.1 Architecture and Implementation

The software network interface architecture is a refinement of the OSI layering model [Tan88], which subdivides the existing *Link* and *Physical* layers as shown in figure 8. The signal processing involved in these layers can be naturally subdivided into a finer-grained 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. In the future, we envision flexible networks, where the type of coding or access protocol may be altered dynamically to adapt to changing conditions and/or user requirements. To facilitate this flexibility, we would like to create new network interfaces by simply combining existing functional modules, rather than by writing an entirely new piece of software for each interface to perform all of the functions in the link and/or physical layers.

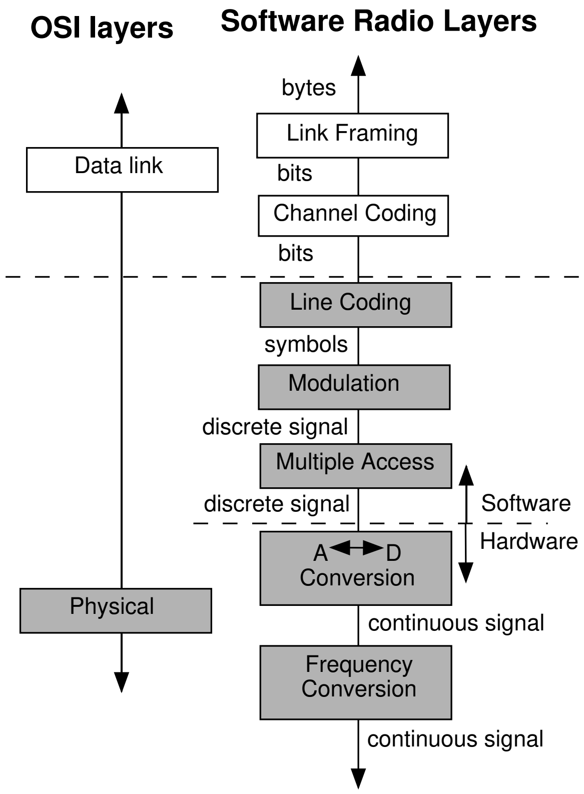

Figure 8: Software Network Layering Model

3.2.2 Transmission

The sequence of processing modules for the transmission application is shown in figure 9. The system interfaces with the host at the IP layer, through our *SoftLink* device driver. This appears to the kernel as just another network device driver. However, instead of handing the packets off to a hardware device, this driver hands them up to user space, where the virtual radio processing is performed. The first level of processing is the network framing. For this example, the packets were framed by inserting a start code and byte stuffing the data. A length code, indicating the total length of the packet including the stuffed values, was also inserted after the start code. The next module takes the sequence of bits output by the network framing layer and performs byte framing, inserting start, stop and parity bits.

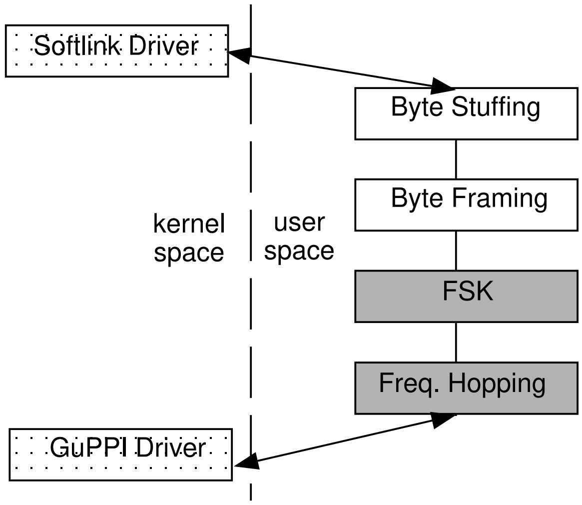

Figure 9: Software components of the transmission application

The conversion of each bit into a discrete signal is performed by the FSK module and the frequency hopping module then assigns this waveform to the appropriate 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 pre-computed and stored at startup, significantly reducing the computation required to produce the transmitted waveform. On a 180 MHz PentiumPro 2.2 μs were required for producing the IF waveform corresponding to a single bit. This corresponds to a maximum possible transmit data rate of $\approx$ 450 kbps.

The generation of continuous phase waveforms is fairly straightforward in software. The pre-computed waveforms are actually oversampled, and only a sub-sampled set, corresponding to the output sampling rate, are copied into the output buffer. The oversampling allows us to index into the buffer to match the phase, and the pattern is treated as a circular buffer, allowing the generation of waveforms for any bit period. After copying the samples to the output buffer, the phase value is updated and used as the index for the waveform corresponding to the next bit. In a similar manner, we are able to maintain continuous phase between hops, even when the hop occurs in the middle of the bit.

3.2.3 Reception

As is usually the case, reception is considerably more complex than transmission even though the sequence of processing modules is essentially the reverse of the transmission system shown in figure 9. The receiver must detect the presence of a valid transmission and synchronize to it, as well as perform the inverse function of each of the transmission layers. Again, combining the parameters of the frequency hopping and the FSK demodulation, we constrained the receiver to look for one of the two valid waveforms at a given hop frequency. Separate functions were implemented to track the hopping sequences, and to lock onto and demodulate the bits. These bits are de-framed, and then the IP packets is extracted. The driver then hands the packet off to the host IP layer for processing.

3.2.4 Performance

Our current implementation uses a 4.8 MHz wide IF band sampled at 10 MSPS with 12 bit resolution, and an RF band centered at 2.45 GHz. The transmission system generates continuous phase waveforms at a sustainable data rate of 320 kbps while hopping 1000 times per second. The reception currently runs at a rate of 64 kbps and supports the same hopping rate. The amount of time to perform each reception function is given in table 2.

Rather than insuring real-time performance with the tight synchronous control over the processing that is typical of many DSP and digital hardware designs, we take an approach that is statistical in nature. We require that, on average, there are enough cycles available to perform the processing, but realize that the actual number of cycles available over any given period of time varies as due to other demands on the system.

A hard real-time system imposes a hard deadline for each task, and provides a mechanism to insure that this deadline is met. In order to quantify the performance of our system, we introduce the notion of *statistical real-time performance*. We characterize the system by defining a probability that the work will be completed within the specified time limit, and specifying the action that is to be taken when the deadline is not met.

The probability is determined by profiling the algorithm, as shown in figure 10, under the expected conditions on the work station. In this case the expected load was simply a Linux workstation running an X server, an NFS file system, and the usual network daemons (e.g. sendmail, inetd, etc.). In other cases, the expected load might involve other signal processing tasks, or significant user activity.

The action to be taken when the deadline is not met will be application dependent. Possible action could include dropping the data, 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 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.

Figure 10 shows a sample distribution of the time required to extract one bit using the quadrature demodulation. The probability that more than 9.44 μs is required is less than 0.003. If we chose to stop the processing after 9.44 μs 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 by only 0.0015.

Function	Time
Frequency Hopping	0.5 μs / hop
FSK lock	66.3 μs / bit
FSK Demodulation	4.5 μs / bit
De-Byte Framing	5.7 μs / bit
De-Packet Framing	4.6 μs / bit

Table 2: Average time required for each reception function on a 180 MHz PentiumPro.

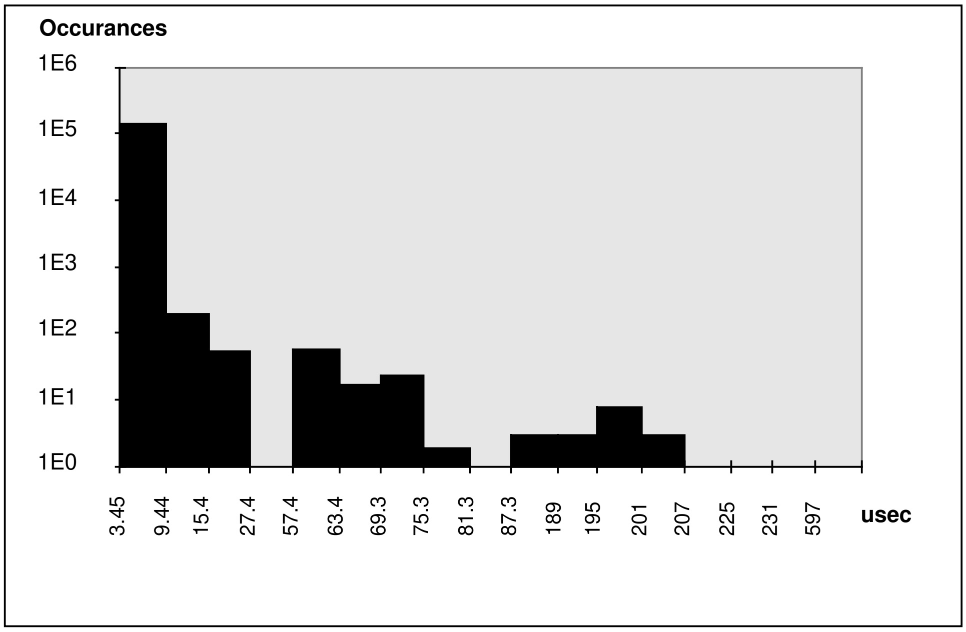

Figure 10: Histogram of the time required (per bit) to perform the FSK demodulation function on a 180 MHz PentiumPro. The histogram is comprised of data from 170,000 trials, the vertical axis is the number of trials occurring for a given time bin, plotted on a log scale. The probability is less than 0.003 that more than 9.44 μs were required for processing for a given bit.

3.2.5 Discussion

The frequency-hopping wireless network interface presented in this section demonstrates the feasibility of using application level software to perform the real-time signal processing associated with such an application. More importantly, it is illustrative of several of the advantages we believe are associated with our approach to building virtual radio devices:

- We were able to build the network interface in a straightforward way using conventional software development tools. The entire implementation is only 520 lines of C++.
- During development the network interface was debugged in simulation using the same code that was later used to implement the actual radio. No porting from a simulation environment was necessary, but the full benefits of off-line simulation are retained.
- The design of the software network interface refines the OSI layering model in a way that would not be practical if we had not moved the hardware/software boundary.

4 Conclusion

The last several years have seen dramatic changes in the hardware used to build software radios. Conventional software radios take advantage of vastly improved A/D converters and DSP hardware. Our approach also depends upon high performance A/D converters. However, rather than use DSPs, we have chosen to ride the curve of rapidly improving workstation hardware.

This choice was dictated by three key assumptions:

- Today's off-the-shelf workstations have enough processing power to support real-time signal processing applications,
- Easy access to the computational resources and development environments available on conventional workstations will lead to new and potentially better approaches to signal processing, and
- Increasingly, there are advantages to be gained by coupling the "radio" component of communication devices with applications.

The work described in this paper provides strong evidence of the validity of these assumptions.

When we started working on this project, I/O bandwidth was the bottleneck. With the completion of the I/O system discussed here, the bottleneck moved to processing. Our measurements show that a 200Mhz Pentium class machine is (just) fast enough to keep up with the demands of the kinds of applications described above. As processor speeds improve and caches grow it will become even easier to implement the kinds of applications we have already built and a variety of new applications will become tractable.

The two applications presented here provide evidence about the validity of the second two assumptions.

Each required a relatively small amount of code and was developed quickly using standard programming tools. The implementation of the cellular receiver took particular advantage of the ease of incorporating existing software that had been built without signal processing applications in mind. Furthermore, its novel channel selection filter is suggestive of the opportunities for algorithmic improvement afforded by virtual radios.

Each application also demonstrates the advantages of being able to closely couple communications devices with applications. This was particularly important in the design of the software network interface, where refining the OSI layering model led to a flexible interface that facilitates inter-operation with a variety of standards.

While we are happy with the progress we have made this far, this work still has a long way to go. We believe that we have demonstrated that virtual radios provide new and useful ways to build familiar communications devices using algorithms that are not too different from existing algorithms. In the next stage of our work, we plan to use the flexibility and power of virtual radios to experiment with radically different algorithms and with new kinds of functionality.

Acknowledgements

The authors would like to thank the many individuals who have influenced and contributed to the development this project. In particular David Tennenhouse for his guidance and vision. Also, Alok Shah, Andrew Chiu and Jeremy Lin for their contributions to the testbed, as well as David Wetherall and Anthony Joeseph for their comments on the manuscript. This research was supported by the Advanced Research Projects Agency under contract No. DABT-6395-C-0060 (monitored by US Army, Fort Huachuca) and by equipment grants from Intel Corporation.

References

[And95] Eric Anderson. *Container Shipping: A Uniform Interface for Fast, Efficient, High-bandwidth I/O.* PhD thesis, University of California, San Diego, 1995.

[Bai95] Rupert Baines. The DSP bottleneck. *IEEE Communications Magazine*, 33(5):46–54, May 1995.

[BS96] Jose C. Brustoloni and Peter Steenkiste. Effects of Buffering Semantics on I/O Performance. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 277–291, Seattle, WA, October 1996. USENIX.

[BW94] 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.

[CHT95] David D. Clark, Henry H. Houh, and David L. Tennenhouse. Aurora at MIT. Project Aurora Final Report, October 1995.

[CP94] Charles D. Cranor and Gurudatta M. Parulkar. Universal Continuous Media I/O: Design and Implementation. Technical Report TR 94-34, Washington University Department of Computer Science, December 1994.

[DP93] Peter Druschel and Larry Peterson. Fbufs: A High Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 189–202, Asheville, NC, December 1993.

[FJ97] Matteo Frigo and Steven G. Johnson. The fastest Fourier transform in the west. Technical Report MIT-LCS-TR-728, Laboratory for Computer Science, Massachusettes Institue of Technology, 1997.

[Ism98] Michael Ismert. The GuPPI: Hardware and Software I/O Support for PC-based Real-time Signal Processing. In *Proceedings of IEEE INFOCOM '98*, 1998. submitted.

[LT96] Christopher J. Lindblad and David L. Tennenhouse. The VuSystem: A Programming System for Compute-Intensive Multimedia. *Journal on Selected Areas of Communication*, 1996. to appear.

[LU95] Raymond J Lackey and Donald W Upmal. Speakeasy: the military software radio. *IEEE Communications Magazine*, 33(5):56–61, May 1995.

[OK88] Hiroshi Ochi and Noriyoshi Kambayashi. Design of complex coefficient FIR digital filters using weighted approximation. In *IEEE International Symposium on Circuits and Systems Proceedings*, pages 43–46, 1988.

[OS89] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, 1989.

[Pai97] Vivek S. Pai. IO-Lite: A Copy-free UNIX I/O System. Master's thesis, Rice University, January 1997.

[Sha97] Alok B. Shah. Software-Based Implementation of a Frequency Hopping Two-Way Radio. Master's thesis, MIT, May 1997.

[Tan88] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, Englewood Cliffs, NJ 07632, second edition, 1988.

[Tha97] Samir R. Thadani. Software-Based Ultrasound System for Medical Diagnosis. Master's thesis, MIT, May 1997.

[vEBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 3-6 1995.