Virtual Radios

Vanu Bose, Michael Ismert, Matt Welborn, and John Guttag

***Abstract*— Conventional software radios take advantage of vastly improved analog to digital converters (ADC's) and digital signal processing (DSP) hardware. Our approach, which we refer to as virtual radios, also depends upon high performance ADC's. However, rather than use DSP's, 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 is 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 input/output (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.

***Index Terms*— Digital down conversion, software radio, software signal processing.**

I. 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 very different from programmable digital radios using application specific digital hardware or digital signal processors (DSP's) under software control [4], [11]. They represent, to our knowledge, the first implementation of a software radio which digitizes a wideband (e.g., 10 MHz) of the RF spectrum and performs all of the signal processing in application level software using a general purpose processor [12].

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.

Manuscript received September 21, 1997; revised January 11, 1998 and July 2, 1998. This work was supported by the Advanced Research Projects Agency under Contract DABT-6395-C-0060 (monitored by U.S. Army, Fort Huachuca) and by equipment grants from Intel Corporation.

The authors are with the Software Devices and Systems Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 USA.

Publisher Item Identifier S 0733-8716(99)02972-8.

Of course, using a general purpose workstation to do signal processing can impose a significant cost. Not only are most DSP's 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. 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. 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.
- *Multipurpose Devices:* Increasingly, people are dealing with multiple wireless communication devices. 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 multipurpose DSP devices, e.g., FAX modems, are becoming increasingly available, they perform a relatively small number of predefined 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 devices for small markets. Furthermore, the virtual radio system architecture encourages sharing of hardware resources across devices. These devices need not even be what one conventionally thinks of as radios. Technology of the virtual radios has been used to implement the signal processing for medical ultrasound machines with the idea of allowing a number of ultrasound front ends to share the same backend workstation [18].
- *Improved Functionality:* Performing all of the signal processing in modular software permits not only the

0733–8716/99$10.00 © 1999 IEEE

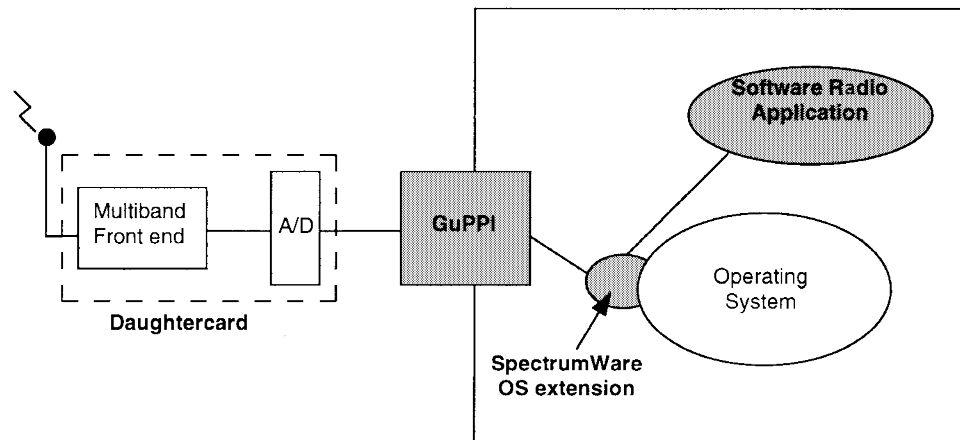

Fig. 1. Block diagram of the virtual radio. The hardware is used only to convert the desired RF band down to the IF frequency, digitize it, and to stream the samples into host memory. All subsequent processing of this digital wide-band IF signal is performed in user-level software.

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 Computational Efficiency:* 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 novel algorithms, which can be more efficient than the constrained DSP algorithms.
  - The ability to integrate the signal processing with higher-level applications affords the opportunity for system-wide optimization.
  - The signal processing functions can be dynamically modified based on measurements of changing channel or system characteristics to improve performance. This information could come from background tasks that perform functions such as channel estimation when spare cycles are available.

Section II 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 input/output (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 I/O subsystem is discussed in some detail, and the key properties of the software environment are outlined.

Section III describes the design and reports on the performance of two applications built using this environment. The applications are relatively simple, yet they illustrate many of the benefits of virtual radios. Section III-A describes a software cellular receiver and Section III-B a software wireless network interface.

The paper concludes by relating the specific results reported in this paper to the more general assumptions behind the on virtual radios.

II. Enabling Technology

The virtual radio research 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 ADC. Since current ADC technology and available processors will not support the direct sampling of wide RF bands, the approach, as illustrated in Fig. 1, is to use a multiband hardware front end to convert the desired RF band to the IF frequency, directly sample the wideband IF waveform and then transfer these samples into host memory. All subsequent processing is performed in user level software.

We have used several front ends for the system, but the most flexible is the AR5000,[1] which provides the ability to tune in any 10 MHz wide band located between 100 kHz and 2.6 GHz. The AR5000 offers no transmit capability, however, and for this we generally use a transmitter dedicated to a particular band. Several vendors are working on wideband receive/transmit units that provide better dynamic range and increased bandwidth.[2] These units should be available in the coming year.

Given the high rate digital samples, one is 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 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

[1] AOR Ltd., Tokyo, Japan.

[2] Such receivers are currently under development by Rockwell/Collins and Hughes Electronics.

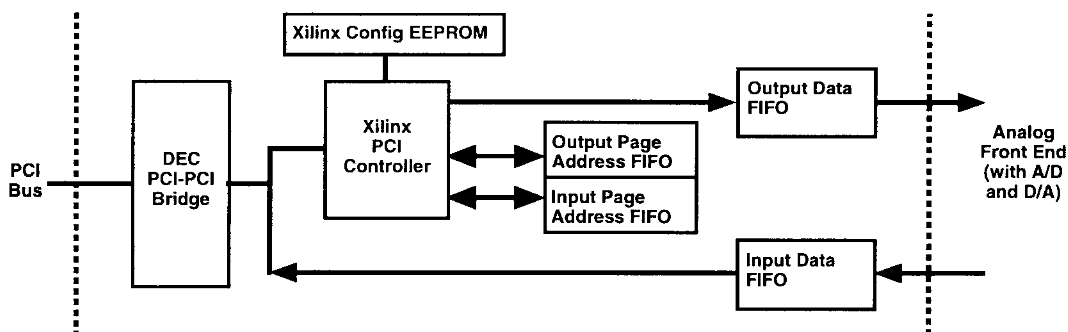

Fig. 2. GuPPI block diagram.

memory [1], [3], [6], [7], [15], [19]. The 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. The general purpose peripheral component interconnect (PCI) I/O system, described below [9], supports this functionality. This interface utilizes the gigabit capacity of the PCI bus, and supports continuous bidirectional I/O streams of up to 32 megasamples/second (ms/s) with 16 bits/sample.

The virtual radio includes 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.

The next section describes the design of the I/O system and reports on its performance. The following section provides a quick overview of the architecture of the programming environment.

A. The I/O System

In standard signal processing systems based on dedicated digital hardware or DSP's, 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 sample.

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 multitasking 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 addressed by the I/O system.

The other major requirement which must be addressed is the need for high throughput between the application and the ADC. 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 Mb/s. This throughput need exposes two bottlenecks in the existing workstation architecture. First, workstations lack a high-throughput port capable of supporting the required samples stream, creating the need to develop custom hardware. Second, the path between a device driver and the application is inefficient, requiring modifications to the operating system. For comparison, the VuSystem [5] reported sustained throughput of only 100 Mb/s 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 the operating system additions, which provide the means for the application to access the sample streams.

1) I/O System Architecture: The GuPPI provides the system's external interface to the analog front end. It bursts data between the front end and main memory at near the maximum I/O bus rate. In addition, the GuPPI decouples the timing between the fixed rate domain of the analog front end and the variable rate I/O bus without losing any samples. This 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.

A block diagram of the GuPPI is shown in Fig. 2. 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 first-in–first-out (FIFO) buffers in both the input and output directions. These FIFO's provide buffering 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 FIFO's, 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 in the GuPPI rather than in a table in memory. Since the processor replenishes the addresses, the GuPPI uses its

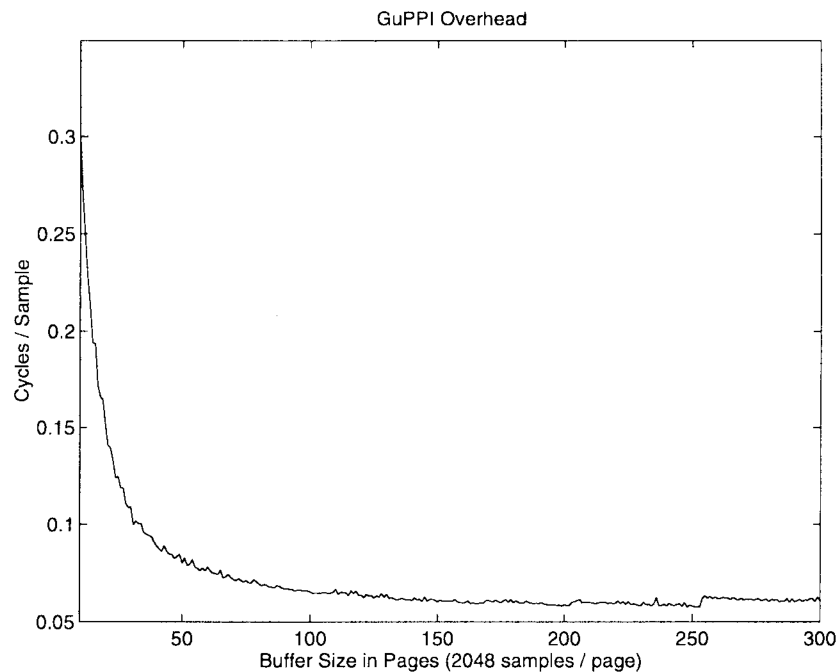

Fig. 3. GuPPI processing overhead (input).

bus grants only to transfer data; this simplified the design of the GuPPI and resulted 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.

The operating system components are responsible for ensuring that the flow of data between the GuPPI and kernel buffers is continuous. This includes 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, version used is 2.0.30. The total size of the code is just under 1000 lines, with the virtual memory system additions representing just 200 of those. The additions 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 must process the data at or above the average rate at which the system will transfer data. If the application does not run in real time, then the system will eventually begin to drop input samples or produce gaps in the output stream.

2) Performance: The measurements of the GuPPI and 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.

TABLE I
RAW GUPPI THROUGHPUT (Mbits/s)

Input	Output
933	790

The maximum rate at which an application using the GuPPI driver can maintain a continuous flow of input samples from the GuPPI is 512 Mb/s. This number was determined using an application that only accessed enough samples per input buffer to verify data continuity. This 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, however, the next revision of the GuPPI will have deeper data FIFO's, increasing the maximum continuous throughput.

To provide this high continuous throughput, the GuPPI must have higher raw throughput. Table I shows the maximum input and output burst performance of the GuPPI. These numbers reflect the amount of time required for the GuPPI to complete the DMA of approximately 1.2 Mb of data. The measurements include only the 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 Mb/s, 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.

Fig. 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 to replenish the supply of input buffer pages.

3) Discussion: The GuPPI provides a high-bandwidth, low-latency connection between an analog front end 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 front end 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 PCI bus, a widely used industry standard, permits the system to be portable to a wide range of workstations and take advantage of 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 Mb/s. This is more than enough for most of the applications for which the system was designed. The two applications, discussed subsequently, are not I/O-limited but processor-limited instead.

B. Programming Environment

SPECtRA is a signal processing environment for continuous real-time applications. The system embodies a set of abstractions and design rules that allow for simple implementation of signal processing functions and enable significant software reuse. The system has three basic components:

- a library of signal processing modules;
- a set of objects to connect the processing modules;
- a scripting language used to define the topology of the system and the interactions between modules.

The design of SPECTRA was heavily influenced by the experience obtained through building and using the VuSystem [10]. The VuSystem, which was designed 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 the adaptive signal processing applications could benefit from a different implementation model. The SPECtRA environment thus was designed to support several different notions of adaptive signal processing.

- *Adaptation to the Environment*: Consider a system equalizing for a wireless 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 sometimes a mechanism for specifying system modifications based on the detected changes.
- *Adaption to the User*: The user's requirements may change, necessitating changes in the system. A simple example of a conventional system adapting to the user is changing the station on the 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 processing path.
- *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 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 many spare CPU cycles available, then running a more computationally intensive channel estimation algorithm could lead to better overall application 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.

A visual description of an application built in SPECtRA is shown in Fig. 4. The system is partitioned into *in-band* and *out-of-band axes*. The in-band axis is where the temporally sensitive, computationally intensive work takes place. The out-of-band axis is used for control and intermodule communication. All of the application specific functionality is contained in the out-of-band section. This partitioning allows for maximal reuse of the computationally intensive in-band signal processing modules, since none of their functionality is specific to a particular application.

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 connectors can be thought of as a wire that carries signals from the output of one processing module to the input of one or more processing modules.

One key innovation in the design of this programming environment is that data is not pushed down the pipe as in most systems, but it is pulled down the pipe by the sink. This allows for the implementation of lazy evaluation along the pipeline, which leads to significant computational savings in applications that use decimation or skip certain portions of the input data. In the data pull case, the downstream module only asks for samples that it needs and can transparently eliminate considerable amounts of computation. The pull architecture also provides performance improvements in systems employ-

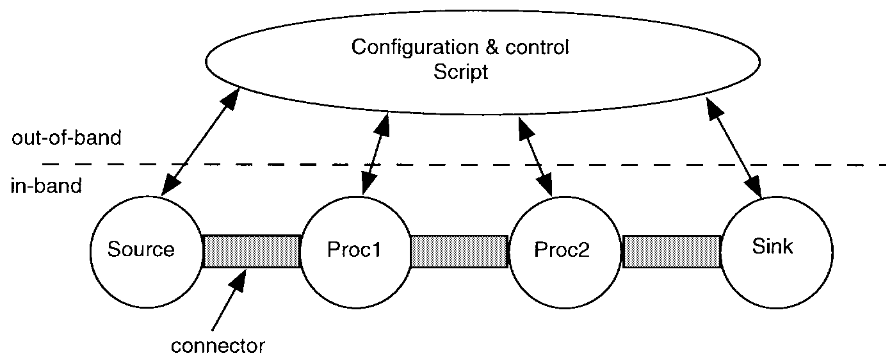

Fig. 4. Graphical description of the programming environment.

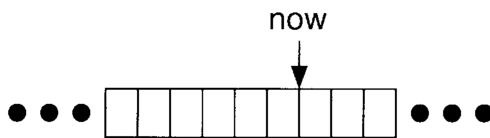

Fig. 5. Representation of streams as infinite buffers.

ing data caching, since the data being operated on is the data most recently produced, and therefore is very likely to be present in the cache.

One of the main motivations behind the development of SPECtRA was to create a programming environment that provides for simple and straightforward implementation of signal processing functions to reduce code development and maintenance overheads. Ideally, there should be little difference between the equations that represent a function, and the code required to implement it.

For many signal processing applications, a stream is a natural I/O model. Functions such as filters operate on a continuous stream of input samples and produce continuous streams of output samples. Many systems use 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 to 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.

A stream abstraction was developed to provide a more natural interface to the data samples. A stream serves as the interface between a processing module and a connector, as shown in Fig. 4. Each streams can be thought of as infinite an buffer, with a pointer that indicates the value corresponding to the current instance in time, indicated by the *now* pointer in Fig. 5. The *now* pointer eliminates the need for a absolute time index into the buffer and allows values to be addressed relative to the current time. Eliminating the need for an absolute index is beneficial not only for systems that are designed to run for arbitrarily long times, but also enables straightforward implementation of functions that do not have a consistent relationship between the input and output streams. Consider a detector in a communications system that recognizes the transmitted waveforms corresponding to binary digits. When there is no waveform present, the processing module would continually process input data without writing output data. When a waveforms is present and recognized, only then is the output stream updated.

The system is an efficient mechanism for implementing real-time wireless communications systems. The base system is reasonably compact, consisting of roughly 8000 lines of code. Writing applications is greatly simplified. For example, the AMPS cellular receiver required writing only 600 lines of code. Furthermore, other applications that reused some of the signal processing functions (e.g., the FM demodulation) realized up to 90% code reuse. Section III examines two example applications in detail.

III. Demonstration Applications

Several applications have been built with the virtual radio system, including a radio capable of receiving four half-duplex communications channels simultaneously and a *virtual patch panel* which enables interoperation between two systems using different RF bands, modulation schemes, and channel widths.

In this section we describe two prototype applications: a software cellular receiver and a software wireless network interface. These applications demonstrate both the feasibility as well as some of the advantages of virtual radios.

A. A Software Cellular Receiver

This section describes a wideband digital receiver which operates in the "A-side" of the U.S. cellular band. It continuously monitors 10 MHz of the cellular band and can demodulate FM signals anywhere in that band. Although the control channel [which uses frequency shift keying (FSK)] and supervisory tone were not performed in this implementation, other work using this architecture has demonstrated the implementation of an (FSK) system [16].

A block diagram of the cellular receiver is shown in Fig. 6. The front end is a Tellabs receiver that translates the 825–835 MHz band to a baseband signal and then samples the signal at 25.6 ms/s. The 12-bit sample stream is connected to a GuPPI daughtercard which digitizes and then DMA's the samples into the main memory of the host for processing. All subsequent processing is performed in software.

1) Software Architecture: The sequence of processing steps for the wideband advanced mobile phone system (AMPS) receiver is shown in Fig. 7. Since this wideband receiver

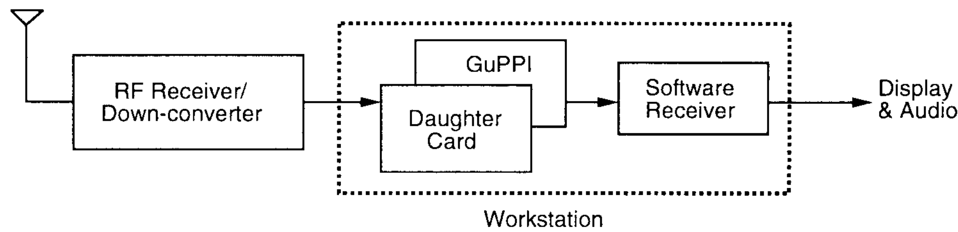

Fig. 6. Cellular receiver block diagram.

Fig. 7. Software architecture block diagram.

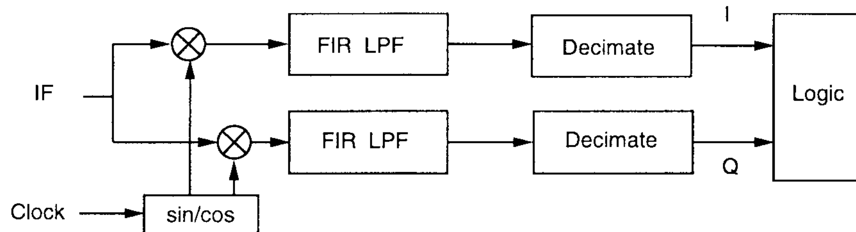

Fig. 8. Dedicated hardware digital downconverter.

demodulates a narrowband signal, the sample rate becomes lower at successive processing stages.

It is worth noting that all of the system parameters, including the number and values of the channel filter coefficients and the sample rates, are under software control. This allows many of these parameters to be easily modified, even while the receiver is operating.

The first processing step is the channel selection filter. This module extracts a 30 kHz FM AMPS channel from a 10-MHz input band. This step uses a novel filter that combines the three conventional steps of translating the signal to baseband, lowpass filtering, and decimating. This step comprises the largest portion of the overall computational load. The details of its design are given in a later section. The channel selection filter consumes the raw samples from the RF front end at $R_S = 25.6$ ms/s and produces a complex baseband signal with a variable intermediate sample rate R_D. The channel selection filter, in this implementation, is initially tuned to the nominal carrier frequency of the desired channel. No tracking of the carrier is required, and any small frequency offset will create only a small dc offset at the output of the discriminator which is then removed by the audio filter.

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 FM demodulation algorithm that approximates the derivative of the signal phase by the phase difference between successive samples, appropriately scaled.

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

The cellular FM demodulator presented here is implemented on a personal computer with a Pentium II/300 MHz microprocessor. The implementation runs in real-time without any glitches or dropped samples and requires approximately 60% of the CPU. Quantitative measurements of the audio output were not made, but the subjective quality is as good or better than a commercial AMPS cellular handset.

Separating narrowband channels in a wideband receiver is a computationally intensive task that is often allocated to special purpose hardware. Fig. 8 depicts a typical digital down-converter (DDC) implemented in dedicated hardware [2]. 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 perform this operation very efficiently—which is important with the sample rates for a wideband receiver could be in excess of 30 ms/s. One author [2] estimates that a good channel selection filter requires about 100 operations per input sample for a total of 3000 million operations per second (MOPS).

It is 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 is the high sample rate of a wideband receiver allows time for only a few operations per sample. In hardware DDC's, the separate steps of down-conversion are each done in separate physical devices in the DDC, resulting in a high degree of pipeline parallelism. Furthermore, there is additional fine grained parallelism within the FIR filter itself. Fortunately,

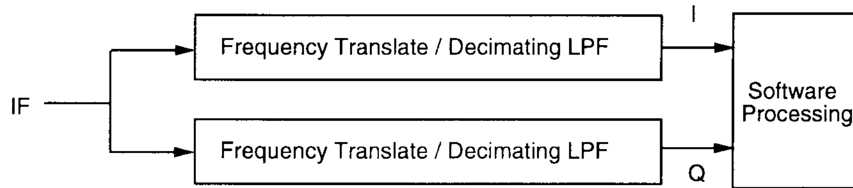

Fig. 9. Proposed software radio downconverter.

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 virtual radios is the freedom to develop and evaluate unconventional solutions to problems.

Fig. 8 depicts the design of the software DDC. The main consideration is to minimize processing at the high sample rate and reduce the rate as early as possible. The choice of an FIR filter (versus a lower-order IIR filter) means that an output that depends only upon the filter input samples, $x[n]$, and therefore it computes only those output samples, $y[n]$, that are required after decimation [14]. This greatly reduces the computation load by taking advantage of large decimation factors. Furthermore, the precomputed frequency shift factors can be incorporated into a composite FIR filter which then require 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 multiplication factor which is applied at the lower sample rate after filtering.

The multiplication for frequency shifting and filtering have been combined into a composite filter. The first step in the selection of a specific channel is to translate the real-valued received signal samples $r[n]$ to baseband by multiplication by a complex exponential

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

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

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

The two steps of frequency translation and filtering can be combined

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

where $c[m] = h[m]e^{j2\pi f_c m T_s}$ are the composite filter coefficients. Not only is the number of computations reduced, but there is no need to compute the unfiltered baseband signal $x[n]$, nor to write 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, (3) shows that the use of complex-valued filter coefficients for $h[m]$ imposes no additional computational cost. For this reason the algorithm can take advantage of recent advances in the design of complex-coefficient FIR filters to reduce the required filter order M of the original low pass filter (LPF), relative to a real-valued $h[m]$, without increasing the required computation load for the final composite filter [13].

This technique also has costs, however. Although the filter requires less real-time computations, it is more complicated to set up and can require 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 recomputation of the coefficients when there is a change in the frequency to which the filter is tuned (or else precompute and store separate filters for any several specific frequencies). Since each set of filter coefficients is used many thousands or millions of times, however, the cost of precomputing or recomputing is well worth the improved efficiency. This technique is similar to using an analytic filter to select only the positive frequency components of the real-valued signal for the passband of interest and decimating. The combination of translation and lowpass filtering in Fig. 9 allows the filter to be quickly redesigned to select a different f_c.

The key improvement over the technique of Fig. 8 is that no operations are performed at the high sample rate. In the case of AMPS, the final rate is on the order of 100 KHz and a complex filter with several hundred taps might require less than 100 MOPS, which is achievable.

Another interesting feature of the software 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 the system performs frequency-domain analysis. The user may perform frequency-domain analysis of either the entire band or individual signals using fast Fourier transform (FFT's) using a library called FFTW.[3] FFTW evaluates the performance of many different FFT algorithms and selects the best one for the particular processor and data set size. The evaluation of different FFT algorithms is based either on an estimate of relative performance or measured performance of different algorithms in the

[3]The FFTW package was developed at M.I.T. by M. Frigo and S. G. Johnson. Additional information, including source code, is available at http://theory.lcs.mit.edu/~fftw.

current environment using system generated test code [8]. This measurement process can also be repeated periodically to see if changes in the system load indicate the use of a different algorithm to compute the FFT.

2) Discussion: This receiver requires only 600 lines of code, including the user interface. The SPECtRA environment enables rapid implementation of wireless systems. Many applications have been completely implemented in only a few hours. During development, standard software debugging tools were used and modifications required only seconds to recompile and execute—a refreshing change from the hardware design and debugging cycle. The development team took advantage of the ease of modification to experiment with a variety of different filtering algorithms during development.

The use off-the-shelf components facilitated the construction of this receiver (for example, the FFTW package). This not only shortened the development cycle but encouraged us to be more ambitious in adding functionality.

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

In designing this receiver, a conscious effort was made 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 of the workstation. The novel channel filter achieves excellent performance and can be redesigned on-the-fly. The time required to recompute the filter tap weights for a different filter size or carrier frequency is not noticeable during operation.

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

B. 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 LAN's 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 (NIC's) in dedicated hardware limits the flexibility of these devices. One approach to this problem is to implement as much of the processing as possible in software, allowing the wireless network interface to be dynamically modified. The SPECtRA software environment provides all of the processing needed to convert between wideband IF signals and network packets.

This section presents a software wireless network interface designed to be compatible with a commercial frequency hopping radio operating in the 2.4 GHz ISM band employing

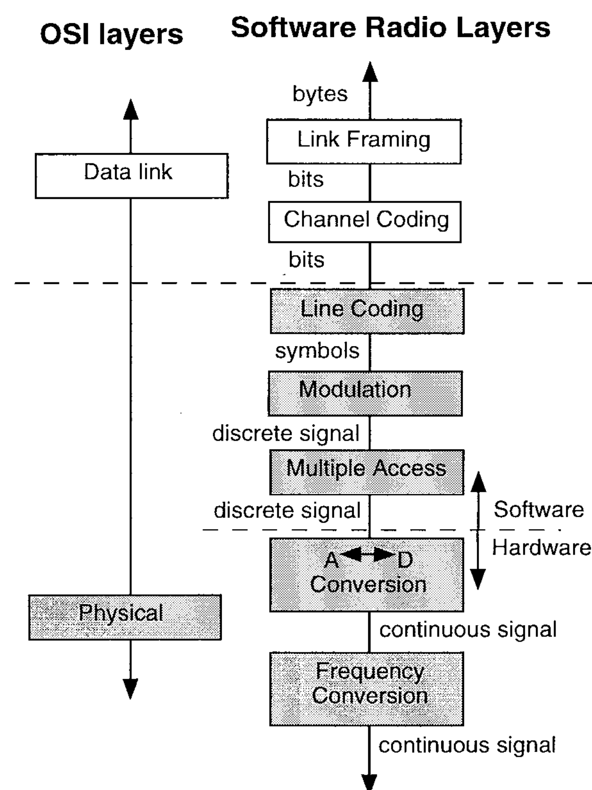

Fig. 10. Software network layering model.

FSK modulation [16].[4] Parameters such as the FSK frequency deviation and the spacing of the hopping channels can be dynamically modified in software. The only constraints imposed by the hardware are the width of the IF band and the center of the RF band. 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.

1) Architecture: The software network interface architecture is a refinement of the OSI layering model [17], which subdivides the *Link* and *Physical* layers as shown in Fig. 10. The signal processing performed 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 the purposes of a virtual radio, 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 a new piece of software for the link and physical layer functions of each interface.

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

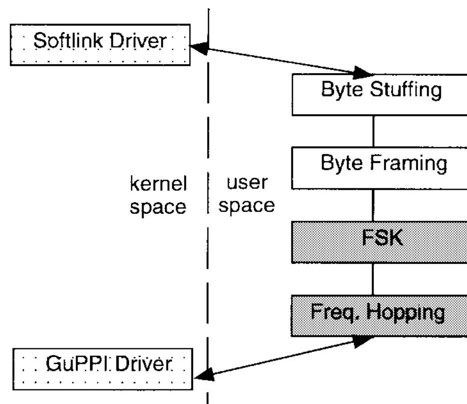

Fig. 11. Software components of the transmission application.

2) Transmission: The sequence of processing modules for transmission is shown in Fig. 11. The system interfaces with the host at the IP layer, through the *SoftLink* device driver. This appears to the kernel as a normal 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 are 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.

The conversion of each bit into a discrete signal is performed by the FSK module. 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 precomputed and stored at startup, significantly reducing the computation required to produce these waveforms. On a 180-MHz PentiumPro the IF waveform corresponding to a single bit required 2.2 μs to produce. This corresponds to a maximum transmit data rate of $\approx$450 kbps.

The generation of continuous phase waveforms is fairly straightforward in software. The precomputed waveforms are oversampled, and only a subsampled set, corresponding to the output sampling rate, are copied into the output buffer. The oversampling allows for indexing into the buffer to match the phase. 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, continuous phase is maintained between hops, even when the hop occurs in the middle of the bit.

3) Reception: Reception is considerably more complex than transmission even though the sequence of processing modules is essentially the reverse of the transmission system shown in Fig. 11. The receiver detects the presence of a valid transmission and synchronizes to it. It also reverses the function of each of the transmission layers. Combining the parameters of the frequency hopping and the FSK demodulation, the receiver to looks for one of the two valid waveforms at a given hop frequency. Separate functions track the hopping sequences and lock onto and demodulate the bits. These bits are deframed, and then the IP packets is extracted. The driver then hands the packet off to the host IP layer for processing.

TABLE II
AVERAGE TIME REQUIRED FOR EACH RECEPTION FUNCTION ON A 180 MHz PENTIUMPRO

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

4) Performance: The current implementation uses a 4.8 MHz wide IF 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 receiver sustains a rate of 64 kbps and supports the same hopping rate. The amount of time to perform each reception function is given in Table II.

Rather than insuring real-time performance with the tight synchronous control over the processing that is typical of many DSP and digital hardware designs, the virtual radio employs an approach that is statistical in nature. It is required that, on average, there are enough cycles available to perform the processing, given that the actual number of cycles available in a given period of time varies 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. To quantify the performance of the system, we introduce the notion of *statistical real-time performance*. The system is characterized 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 Fig. 12, under the expected conditions on the work station. In this case the expected load is 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 is 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 resources are 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, so a more graceful failure mode could be chosen.

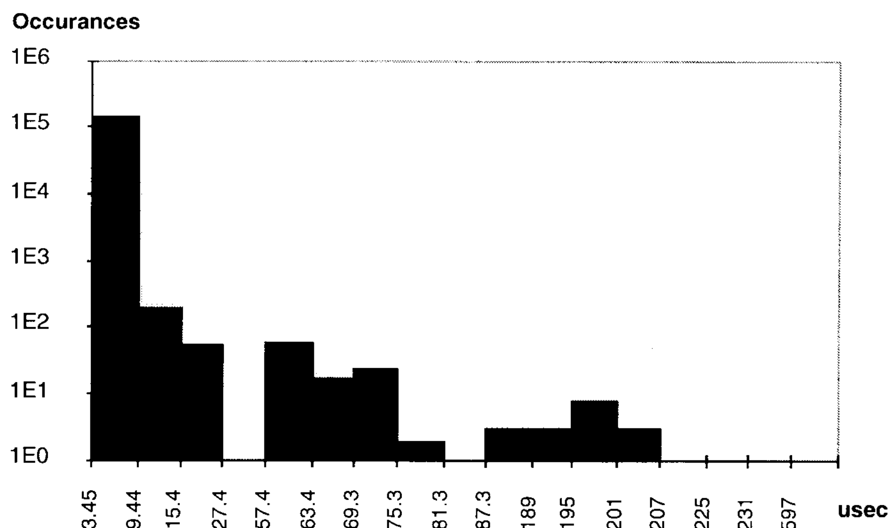

Fig. 12. Histogram of the time required (per bit) to perform the FSK demodulation function on a 180 MHz PentiumPro. The histogram is comprised of data from 170 000 trials, the vertical axis is the number of trials occurring for a given time bin, plotted on a log scale. The probability is less than 0.003 that more than 10 μs were required for processing for a given bit.

Fig. 12 shows a sample distribution of the time required to extract one bit using quadrature demodulation. The probability that more than 10 μs is required is less than 0.003. If the processing is stopped after 10 μs and an arbitrary decision as to the value of the bit is made, this would correspond to an increase in the probability of a bit error by only 0.0015.

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. More importantly, it is illustrative of several of the advantages associated with the virtual radio approach to building virtual wireless devices.

- The network interface was built 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 with a conventional hardware/software boundary.

IV. 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 ADC's and DSP hardware. The virtual radio approach also depends upon high performance ADC's. However, rather than use DSP's, virtual radios ride the curve of rapidly improving workstation hardware.

This choice was dictated by three key assumptions.

1) Today's off-the-shelf workstations have enough processing power to support real-time signal processing applications.
2) Easy access to the computational resources and development environments available on conventional workstations will lead to new and potentially better approaches to signal processing.
3) 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 this project began, the I/O bandwidth was the bottleneck. With the completion of the I/O system discussed here, the bottleneck moved to processing. Measurements show that a 200 MHz 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 these kinds of applications and a variety of new applications will become tractable.

The two applications presented above 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 close coupling of the 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 interoperation with a variety of standards.

While there has been considerable progress thus 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 very 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.

ACKNOWLEDGMENT

The authors would like to thank the many individuals who have influenced and contributed to the development of this project. In particular they thank D. Tennenhouse for his guidance and vision, A. Shah, A. Chiu, and J. Lin for contributions to the testbed, and D. Wetherall and A. Joseph for their comments on the manuscript.

REFERENCES

[1] E. Anderson, "Container shipping: A uniform interface for fast, efficient, high-bandwidth I/O," Ph.D. dissertation, Univ. California, San Diego, 1995.
[2] R. Baines, "The DSP bottleneck," *IEEE Commun. Mag.,* vol. 33, pp. 46–54, May 1995.
[3] J. C. Brustoloni and P. Steenkiste, "Effects of buffering semantics on I/O performance," in *Proc. 2nd Symp. Operating Syst. Design and Implementation (OSDI'96),* Seattle, WA, Oct. 1996, USENIX, pp. 277–291.
[4] A. Brown and B. Wolt, "Digital L-band receiver architecture with direct RF sampling," in *IEEE Position Location and Navigation Symp.,* Apr. 1994, pp. 209–216.
[5] D. D. Clark, H. H. Houh, and D. L. Tennenhouse, "Aurora at MIT," Project Aurora Final Rep., Oct. 1995.
[6] C. D. Cranor and G. M. Parulkar, "Universal continuous media I/O: Design and implementation," Dep. Comput. Sci., Washington Univ., Tech. Rep. TR 94-34, Dec. 1994.
[7] P. Druschel and L. Peterson, "Fbufs: A high bandwidth cross-domain transfer facility," in *Proc. 14th Symp. Operating Syst. Principles,* Asheville, NC, Dec. 1993, pp. 189–202.
[8] M. Frigo and S. G. Johnson, "The fastest Fourier transform in the west," Lab. Comput. Sci., Massachusettes Inst. Technol., Tech. Rep. MIT-LCS-TR-728, 1997.
[9] M. Ismert, "The GuPPI: Hardware and software I/O support for PC-based real-time signal processing," in *Proc. IEEE INFOCOM'98,* 1998.
[10] C. J. Lindblad and D. L. Tennenhouse, "The VuSystem: A programming system for compute-intensive multimedia," *IEEE J. Select. Areas Commun.,* 1996.
[11] R. J. Lackey and D. W. Upmal, "Speakeasy: The military software radio," *IEEE Commun. Mag.,* vol. 33, pp. 56–61, May 1995.
[12] J. Mitola, "The software defined radio: Technology and marketplace," in *IEEE Western Electron. Conf. (WESCON),* Oct. 1996.
[13] H. Ochi and N. Kambayashi, "Design of complex coefficient FIR digital filters using weighted approximation," in *Proc. IEEE Int. Symp. Circuits Syst.,* 1988, pp. 43–46.
[14] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1989.
[15] V. S. Pai, "IO-lite: A copy-free UNIX I/O system," Master's thesis, Rice Univ., Jan. 1997.
[16] A. B. Shah, "Software-based implementation of a frequency hopping two-way radio," Master's thesis, M.I.T., May 1997.
[17] A. S. Tanenbaum, *Computer Networks,* 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1988.
[18] S. R. Thadani, "Software-based ultrasound system for medical diagnosis," Master's thesis, M.I.T., May 1997.
[19] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-net: A user-level network interface for parallel and distributed computing," in *Proc. 15th ACM Symp. Operating Syst. Principles,* Copper Mountain, CO, Dec. 1995.

Vanu Bose, for a photograph and biography, see this issue, p. 512.

Michael Ismert received the M.E. and B.S. degrees in electrical engineering from M.I.T. in 1995 and 1993, respectively.

He has experience in ATM networking and high bandwidth host interfaces. His current work at M.I.T.'s Laboratory for Computer Science is focused on I/O and operating system support for software radios.

Matt Welborn received the B.S. degree in systems engineering from the U.S. Naval Academy in 1989, and the M.S. degree in electrical engineering from the Virginia Polytechnic Institute and State University in 1996.

He is presently a Ph.D. candidate in the M.I.T. Department of Electrical Engineering and Computer Science. His current work is in the development of DSP algorithms for wireless communication systems, particularly software radios.

John Guttag received the Bachelor's and Master's degrees from Brown University, and the Doctorate degree from the University of Toronto.

He is a Professor and Associate Department Head for Computer Science of M.I.T.'s Department of Electrical Engineering and Computer Science. He also heads the Software Devices and Systems Group within M.I.T.'s Laboratory for Computer Science. He has done research, published, and taught in the areas of software engineering, programming language design, computer security, computer systems and networks, mechanical theorem proving, hardware and software verification, and compilation.