

# Making Commodity PCs Fit for Signal Processing

Michael Ismert

*Software Devices and Systems Group*

*Laboratory for Computer Science*

*Massachusetts Institute of Technology*

*Cambridge, MA 02139*

izzy@lcs.mit.edu, <http://www.sds.lcs.mit.edu/>

## Abstract

Commodity PCs are on the verge of being capable of performing a variety of signal processing tasks that previously required special purpose hardware. Advances in the speed and width of their processors and internal buses allow these machines to manipulate data at rates that would allow their users to interact with a diverse range of sampled media, such as the raw RF spectrum and ultrasound. However, today's PCs lack an I/O system capable of delivering the appropriate bandwidth to these signal processing applications. These applications demand high *continuous* throughput I/O that smooths the inter-sample jitter introduced by interrupts, I/O bus latency, scheduling latency, etc. This paper presents a system that provides this functionality.

Our system is composed of two tightly integrated parts: a PCI device that provides high raw I/O bus throughput and operating system enhancements to manage the device and provide low overhead transfers across the boundary between kernel and user space. The performance is excellent, providing up to 512 Mbits/sec of continuous throughput for an application. A description of both parts of the system is given, along with performance measurements, and a brief description of an application.

## 1 Introduction

Processors in commodity PCs have reached the point where they are capable of performing the signal processing tasks typically left to special-purpose hardware[SPL92]. The possibility of moving these tasks from external hardware into software

running on the main CPU presents some exciting possibilities[TB96]. These include multi-purpose devices where new functions are possible by adding or upgrading software, allowing PCs to emulate devices ranging from cellular or cordless phones to ultrasound or EKG monitors[Tha97, BCT97]. This also allows rapid deployment of new or upgraded standards as well as the ability to rapidly prototype new signal processing algorithms or protocols. Finally, this approach can enable increased performance in several ways. The obvious one is to take advantage of the fact that market factors are driving rapid improvements in PC performance. However, this approach also allows the integration of the signal processing with higher-level applications, presenting the opportunity for system-wide optimization, as well as allowing the signal processing functions to be dynamically modified based on measurements of changing system characteristics in order to improve performance.

There are already industry movements in this direction. Vendors of software modems, for example, advocate digitizing the phone line signal and doing the necessary coding and modulation in high-priority interrupt handlers[Tra97]. Today's PCs can handle these low data rate tasks easily. However, for the more aggressive applications we envision, such as those encompassing large bands of the RF spectrum, the off-the-shelf PC is not yet a viable platform.

Increasing clock rates, superscalar architectures, and SIMD techniques such as Intel's MMX allow general-purpose processors to dedicate a reasonable number of cycles to each data sample. Along with the processors, the internal buses have increased in both speed and width, allowing high-bandwidth data to be moved about within the PC. However, today's commodity PCs are lacking two important

pieces that enable this sort of signal processing. The first is an I/O device capable of digitizing (and perhaps downconverting) a generic wide-band signal and delivering the samples to the PC's main memory (and performing the reverse operation as well). The second is the ability of current commercially available operating systems to deliver high-bandwidth, low-jitter, continuous data streams to the application.

This paper presents a system that fills in these gaps in current commodity PCs. There are several goals that our system must satisfy. First, it must provide high *continuous* throughput between the digitizing front-end and the applications. As an example, the A-side cellular telephony band is 12.5 MHz wide. When digitized at 25 MHz with a sample size of 16 bits, the data rate necessary to transfer this stream of samples to the application for processing is 400 Mbits/sec.

Second, the system must provide the applications with what appears to be a jitterless sample stream. 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 lock step 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 personal computer, however, there are no such simple guarantees. 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. The jitter introduced by all of these sources must be smoothed out.

Since many signal processing applications of interest, such as those involving two-way voice communications, are sensitive to latency, the system must not introduce an excessive amount of delay between the time when samples enter the system and when they are processed. In addition, signal processing applications are also processing-intensive. As a result, the system must accomplish the previous goals with as little processing overhead as possible.

Finally, we imposed the goal of simplicity on the system, both on its use and design. Application programmers should not have to use a new set of system calls to allocate memory for buffers, transfer data, and access the device, nor should the system design require such things to be implemented.

The system is composed of two co-designed parts: a hardware component described in section 2, and operating system support described in section 3. The performance of the system is presented in section 4. Section 5 is a discussion of the previous work related to this area.

## 2 The I/O Device

The General Purpose PCI Interface (GuPPI) provides the necessary high-speed I/O port for PC-based signal processing. It is a two board sandwich, consisting of a full-size PCI card which contains the engine for moving samples to and from memory, and a A/D card which is capable of sampling an analog input at 40 million samples/second with a 12 bit resolution and generating an analog output from a sample stream of the same rate and resolution.

A block diagram of the data transfer portion of the GuPPI is shown in figure 1. The core of the GuPPI is the Xilinx FPGA, which contains the PCI controller and DMA engine. The design provides high raw bandwidth between the A/D card and the PC's main memory.

The A/D card 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. In the input direction, the FIFO holds incoming samples until the GuPPI acquires the bus and can transfer them into memory. In the output direction, the FIFO holds excess samples that have been transferred from memory but are waiting to be accepted by the daughter card.

As well as absorbing jitter, the FIFOs also serve to temporally decouple the timing of the GuPPI, which operates using the PCI clock, from the daughter card, which typically operates using the A/D sampling clock. This eliminates the need for designers of daughter cards to worry about synchronization or metastability issues. It also separates the data

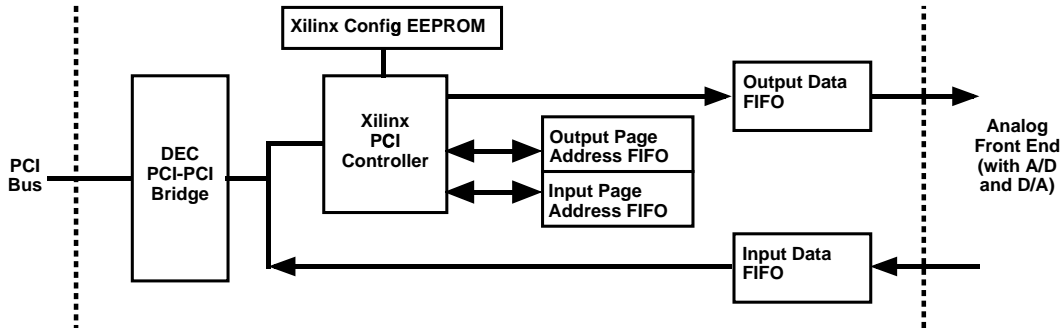


Figure 1: GuPPI Block Diagram.

transfer and processing from the fixed rate realm of the analog front-end, allowing it to be bursty.

The GuPPI is connected to the PCI bus for its high throughput. PCI is an example of how the internal buses in commodity PCs have improved enough to allow the movement of high speed data; its raw capacity is just over 1 Gbit/sec, which puts it well over the bandwidth needed for our initial target applications. Several other possible points of connection were possible, such as the ISA bus or the I/O bus used in some other architecture, such as the SBus. However, the low bandwidth of the ISA bus (approximately 140 Mbits/sec) eliminated it. The SBus, while providing reasonable bandwidth, limited the GuPPI to operation in Sun workstations. The PCI bus is a more popular I/O bus architecture that is common to all commercially available PCs, including Intel and AXP-based machines, and provided easy potential upgrades to double speed, double width, and mobility via CardBus.

The GuPPI is a PCI bus master; that is, it has the ability to initiate transfers on the PCI bus. As such, the GuPPI can DMA sample streams to/from main memory at high speed with minimal intervention from the processor, which supports our goal of low overhead. The GuPPI implements a new variant of scatter/gather DMA that we have named *page-streaming*. The GuPPI has two page address FIFOs, one each for input and output, that 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. Using the programmable FIFO flags, 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.

There are several benefits for signal processing applications from using page streaming DMA. First, since the processor replenishes the addresses, the GuPPI only uses its bus grants to transfer data, which results in more efficient use of the PCI bus by the GuPPI. Since DMA transfers are always page-length and page-aligned, no buffer length information needs to be transferred across the bus, reducing the bus overhead associated with the DMA. In addition, page streaming simplified the design of the GuPPI by reducing the complexity of Xilinx DMA controller. Page streaming also has several nice characteristics related to the operating system enhancements, which are described in the next section. The lone drawback to page streaming is that transfer sizes are fixed to page-sized, page-aligned buffers; however, the continuous nature of the sample streams ensures that transfers of this size can always be completed, and page alignment can be easily arranged by manipulating the beginning of buffers.

### 3 Operating System Enhancements

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<sup>1</sup>. The total size of the code is just under 1200 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

<sup>1</sup>The Linux kernel version used is 2.0.31.

run completely unperturbed. The device driver provides for the continuous transfer of data between the GuPPI and main memory while absorbing jitter due to the scheduling of the signal processing applications. The virtual memory additions provide low overhead, high-bandwidth transfer of data between the application and the device driver.

### 3.1 GuPPI Device Driver

The device driver is responsible for using the raw burst performance of the GuPPI to provide a continuous stream of I/O to the application, and to absorb jitter caused by interrupts and the scheduling of other processes.

Initially, a purely user-level approach, similar to [vEBBV95], was implemented and tested. The GuPPI's control and status registers were memory-mapped into the virtual memory space, allowing the application to completely control the GuPPI. Application-allocated buffers were locked down in physical memory and given to the GuPPI for DMA. While the burst performance was excellent, the continuous performance suffered, particularly during periods where either another process received a substantial portion of the processor time or an interrupt for another device occurred and a slow interrupt handler was called. These performance breaks were due to the inability of the application to keep the DMA engine supplied with buffers to be transferred. Ironically, the process was being swapped out while it was attempting to provide the GuPPI with the buffers necessary to keep the I/O continuous in the event that it was swapped out.

The failure of the user-level approach to be able to maintain a continuous I/O stream of samples motivated the use of an interrupt-driven device driver within the kernel. This driver made the virtual memory system additions (described later) necessary. The implementation and performance of the kernel-level driver are presented in this paper.

#### 3.1.1 Input

The input portion of the GuPPI device driver uses a ring of buffers into which samples are transferred. These buffers are part of the kernel virtual memory space, but are locked down in physical memory. The driver initially fills the input page address FIFO

with addresses from buffers at the head of the ring. When this FIFO drains to a certain (programmable) level, an interrupt is triggered, and the interrupt handler queues up more buffers from the head of the ring. The level that triggers the interrupt is set such that there are sufficient pages remaining to absorb samples arriving before the interrupt handler can provide new buffers.

When an application reads data from the driver, it is given the buffer at the tail of the ring. Rather than copy the data from the kernel to the user buffer, the driver uses the virtual memory additions to swap the buffer provided by the application for the buffer in the kernel<sup>2</sup>. The swapping not only avoids the cost of copying the data, but eliminates the need for the kernel to allocate a buffer to replace the one given to the application.

The ring buffers allow the driver to absorb scheduling jitter for those applications attempting to run in real-time. When the signal processing application is not scheduled, the head of the ring, which is the last buffer given to the GuPPI for DMA, will move away from the tail of the ring, which is the next buffer to be given to the application. When the application is once again scheduled, it will need to be able to read and process buffers faster than the GuPPI can fill them, moving the head back towards the tail. If the application cannot do this, it has no chance of maintaining any real-time operation. The amount of scheduling jitter that the system can absorb is dependent on the size of the ring.

#### 3.1.2 Output

The output portion of the GuPPI driver maintains a queue of buffers that the application has written to the driver. The total size of this queue is bounded to keep applications from using all of physical memory. Initially, the driver fills the GuPPI's output page address FIFO with addresses from buffers at the head of the queue. Similar to input, when this FIFO drains to a certain level, an interrupt is triggered and the interrupt handler replenishes the FIFO from buffers at the head of the queue, if they exist.

Our early experience with writing applications has

---

<sup>2</sup>The application can configure the size and number of buffers used in the ring; this is the means by which the application knows the size of the buffer to provide to the `read` system call.

shown that it is usually efficient to generate output waveforms into buffers on the fly. This allows the output portion of the driver to use the same virtual memory additions to swap application buffers with kernel buffers of the same size. In order to avoid constantly allocating new buffers to swap back to the application, the driver maintains a queue of recently-used buffers and only allocates new ones when a buffer of the proper size is not in the cache. Since applications tend to use buffers of the same size, this works quite well.

The output queue plays the same role for output as the ring buffers do for input. While scheduled, the application should be able to put several buffers on the waiting queue. If the application is running faster than the GuPPI, these buffers will not be depleted by the next time the application is again scheduled to run. There is a maximum total memory size that the driver allows to be on the waiting and sending queues; writes will return unsuccessfully when this point is reached.

### 3.2 Virtual Memory Additions

The standard Unix approach, when faced with a new device, is to implement a Unix device driver to control the device and transfer data to and from it. However, the performance problems associated with using the default Unix I/O system to move data across the kernel/user boundary are well known. 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 approach to this problem uses virtual memory manipulation but, unlike most of the related work, avoids adding new buffering and I/O semantics. Systems such as those presented in [DP93, CP94, Pai97] require the use of a new I/O API in order to access the high-performance I/O system, including a new set of system calls, data structures, etc. In order to maintain a simple and familiar interface for the user, the system was designed to use the standard Unix API and copy semantics. However, virtual memory manipulations are used to make the `read` and `write` system calls to the GuPPI copy-free.

To this end, facilities have been added to the virtual

memory system to swap a buffer in an application's virtual memory space with a buffer in the kernel's virtual memory space. The choice to swap buffers rather than share them between the kernel and the application had several motivating factors. For input (reading), the application may want to perform in-place computation on the samples in that buffer, so the kernel cannot turn around and reuse the buffer as part of the ring until it sure that the application has no further need for it. This involves either an addition to the API for the application to inform the kernel that a buffer can be reclaimed, some amount of data copying to keep the buffer from being overwritten, or allocating a new buffer to become part of the ring. These options all involved a significant increase in design complexity without a clear improvement in performance. For output (writing), since the applications generate data on the fly, there is no real advantage to sharing between the application and the kernel.

The ability to easily perform this buffer swap is due to the unframed, continuous nature of the data streams; enough data can be acquired such that all buffers that the GuPPI uses are page aligned and an integral number of pages long. This makes the virtual memory swap almost trivial as long as the user provides a buffer of the proper size. Each page in the user buffer is faulted into a distinct physical page. The physical addresses in the page table entries for the kernel and user buffers are then swapped, and the appropriate TLB entries in the memory management system flushed. The kernel and user buffers maintain their same protections, and there is no violation of protection because only kernel buffers that are guaranteed to contain only new incoming data (in the case of input) or stale output data (in the case of output) are given to the user. The performance gain of this swap over a kernel-to-user data copy is two orders of magnitude; more detail will be presented in section 4.

The use of page streaming DMA coupled with the virtual memory swapping has some nice characteristics. First, the constraint of page-aligned, integral page-length buffers for both operations means that buffers used for DMA are ideally suited for transfer to the application, and vice versa. In addition, since the physical addresses for both the user and kernel buffers are determined during the swap, the list of pages used to be used for the page streaming DMA is generated for free.

## 4 Performance

The performance numbers presented in this section demonstrate the effectiveness of our system. The experiments were all performed with warm caches on an unloaded 200 MHz Pentium Pro system running Debian Linux. The CPU on-chip cycle counter was used to measure throughput and the processing requirements generated by the GuPPI. Due to our lack of a front-end for transmit, most of the applications which have been developed are receive only, and so the receive performance has been studied significantly more.

### 4.1 Throughput and Overhead

The maximum rate at which an application using the GuPPI driver can maintain a continuous flow of input samples from 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 increase the maximum continuous throughput. For output samples, the maximum throughput is only 260 Mbits/sec. The output data FIFO is half the size of the input data FIFO, and so is not able to absorb as much PCI bus jitter; increasing this FIFO size will similarly increase the output throughput.

Input	Output
930	850

Table 1: Raw GuPPI Performance (Mbits/sec)

In order to provide 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 larger latency required to initiate a read from main memory to the GuPPI.

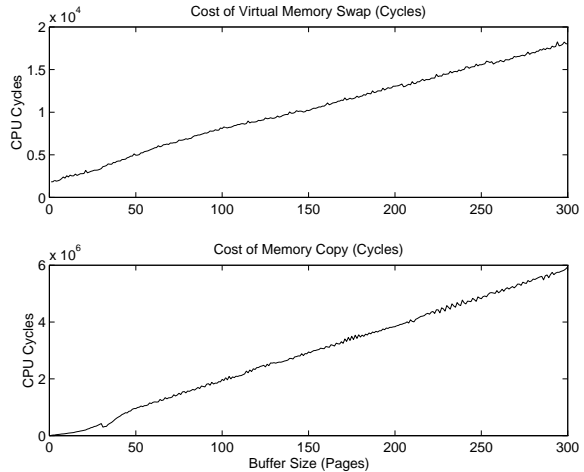


Figure 2: Virtual Memory Swap vs. Memory Copy

The performance benefits of using a virtual memory swap rather than a data copy to move samples from kernel to user buffers are shown in figure 2. The values for both cases are the average number of cycles over 100 buffer reads from the GuPPI. The same user buffer is recycled for each read, so after the first iteration all the pages in that buffer have been faulted into physical memory. Since it is expected that applications will frequently reuse their input buffers, this should not be a performance issue.

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 to perform the read system call (which includes the virtual memory swap) and the number of cycles required to handle interrupts generated by the input page address FIFO running near empty. The average number of cycles per sample is very low; for standard buffer sizes that applications might handle (30 to 300 pages), the processing overhead of the GuPPI is less than a half cycle per sample.

### 4.2 Latency

Another performance issue to consider is that of latency. By adding buffering, both FIFOs and I/O kernel buffers, to smooth the jitter in the system, we have added to the delay between when samples

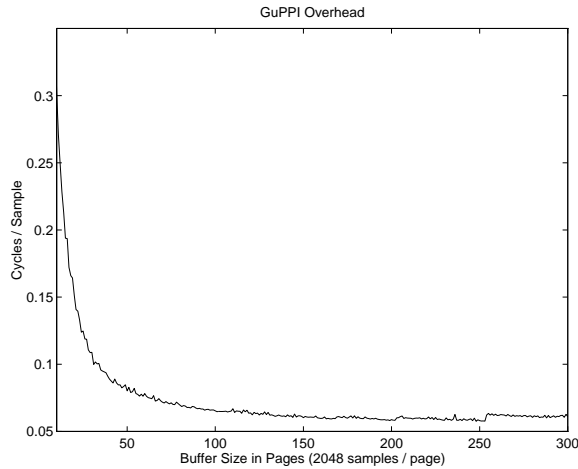


Figure 3: GuPPI Processing Overhead (Input)

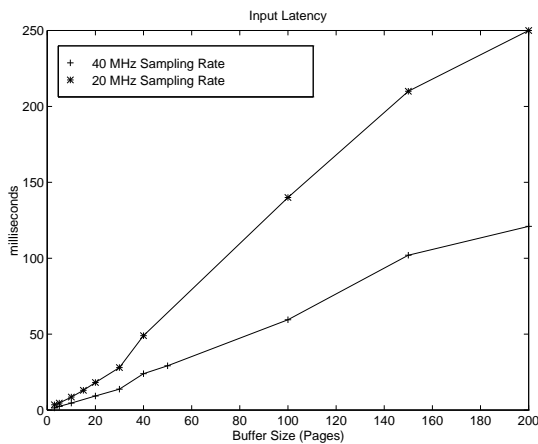


Figure 4: GuPPI Latency (Input)

are generated by the A/D converter and when they are processed. Figure 4 shows a graph of the input latency as a function of buffer size. To generate these values, a sampled step function was sampled by the GuPPI and streamed into memory. An application monitored the incoming buffers for the step and strobed a pin on the parallel port when it was detected.

The input latency of the system is heavily dependent on the sampling rate; larger buffers require more time to fill with samples before they can be delivered to the user, while higher sampling rates cause buffers to fill more quickly. The graph shows the linear relationship between the buffer size (with ring size fixed at 4 buffers) and the latency, as well as the effect of reducing the sampling rate in half; the

numbers are consistent with our expectations.

The minimum input latency measured was just over 1 millisecond for a two page buffer. While the latency measured for smaller buffer sizes is acceptable for a two-way voice application or a point-to-point data connection, it is not sufficiently small for multiple-access schemes such as CSMA or TDM. In these cases, some additional hardware may be necessary to synchronize the transmitter with the receiver.

### 4.3 A Real Application

Along with these abstract measurements, we were also able to measure the performance of an actual real-time signal processing application that used our I/O system. This application is an AMPS cellular telephony receiver using a data rate of 204.8 Mbits/sec (8 bit samples at 25.6 MHz). The application selects a single AMPS channel from a 10 MHz wide band, demodulates it, and plays the audio to the workstation speaker in real-time. The application is processor-limited; our I/O system is able to supply it with the sample stream for the entire cellular band but the application cannot currently demodulate more than one channel. On the original 200 MHz Pentium Pro platform, the application was not even able to meet the full AMPS-specified channel selection filter because the amount of processing required caused it to quickly fall behind the data provided by our I/O system. However, the entire system has been ported to a 533 MHz AXP-based system which can meet the required specifications for one channel. So, there is still a bottleneck in our system, but it has shifted from I/O to processing, allowing us to ride the curve of improved processor performance. It is important to note that, although specialized DSPs can ride this same performance curve, they do not track it nearly as well. DSP clock rates are just reaching 200 MHz; Digital has had AXP processors which operate at 600 MHz for quite some time now. In addition, porting the I/O system and the AMPS application to the AXP system required less than a week, something which would have taken significantly longer moving from one DSP architecture or generation to another.

## 5 Related Work

### 5.1 Data Acquisition Devices

For many years, data acquisition cards for PCs have been available; these have been capable of storing samples on-board in a circular buffer until some trigger condition is met, and then dumping the contents of the buffer into memory on the workstation. These cards have typically been ISA bus devices, with no support for the direct streaming of samples into the PC's memory. Recently, however, several companies have released PCI variants of these data acquisition cards which are capable of streaming samples directly into memory[Ins98, Sci98]. These devices are capable of high burst data transfers; the Gage CompuScope 8500 is capable of bursts of 933 Mbits/sec. However, these devices are still based on the trigger/capture model; after a trigger, the device transfers samples into PC memory rather than onboard memory for some fixed period of time. In addition, these devices are input-only devices, and do not have the operating system support required for continuous high throughput to/from the application.

### 5.2 Host Adapter Research

Most of the research into high performance host adapters is related to network adapters. [DDP94] presents the experience of integrating Osiris, one of the first experimental ATM host adapters, with the high performance I/O work presented in [DP93]. The authors describe the ability to tune the programmable logic on the host adapter in order to simplify the task of writing efficient operating system software. Our approach to designing the GuPPI and its operating system support is very similar; however, we had the luxury of designing both the hardware and the software concurrently, and so very little tuning was actually necessary. There are also several similarities between the operation of the Osiris board and the GuPPI. Both boards implement fixed-size DMA transfers, both boards use a FIFO queue, fed by the host processor, to provide buffers for input or output, and both boards use an interrupt-driven mechanism for replenishing input buffers. However, the need to support network data rather than sample streams causes different functionality in the two boards. The performance of

this pairing of host adapter and I/O system is quite high, with the ability to transfer up to 516 Mbits/sec from the adapter to the host and 325 Mbits/sec in the opposite direction.

[DPC97] presents another high-performance ATM adapter. The paper describes a technique for performing zero-copy data transfers between the host adapter and user buffers. The authors are developing a special ATM protocol that provides page aligned, integral page length data in each packet. This is essentially the same approach as that used by the GuPPI, except the continuous input stream removes the need for any special protocol.

### 5.3 Operating System I/O Research

A significant amount of research has been directed at optimizing the data transfer across the kernel/user boundary. [BS96] provides a taxonomy for the various ways in which the kernel/user boundary can be crossed, as well as discussing possible optimizations, implementing them in the Genie system, and providing some quantitative analysis of the various methods. The operating system modifications for the GuPPI provide facilities that are similar to the emulated copy semantics described in this paper, with some optimization based on the application characteristics.

[DP93] investigates the use of virtual memory remapping applied to system-allocated buffers, called *fbufs*, as a means of efficiently moving data across protection domains. Several optimizations are applied to this strategy, including caching and shared memory. IO-Lite[Pai97] is a system that attempts to unify all I/O related buffering and caching into a single efficient, copy-free system. IO-Lite is based on aggregates of the *fbufs* and uses identical methods (virtual memory remapping, shared memory) to move buffers made of these aggregates between domains. Either of these systems could replace the emulated copy mechanism as the mechanism for crossing the kernel/user boundary. However, the need for a new I/O API to support them makes them unattractive for our use.

Universal Continuous Media I/O (UCM I/O)[CP94] and Container Shipping[And95] are two more systems that seek to provide more generic I/O support. These systems also use virtual memory manipulations to avoid copying, but with move, rather

than share or copy semantics; output data disappears from and input data appears in the application's virtual memory space. Several optimizations are presented in each, including the recycling of virtual memory information (page tables, addresses, pages), the elimination of unnecessary zero-filling of allocated buffers, and the selective mapping of only the used portions of buffers. The move semantics make this sort of system unattractive in our context for two reasons. First, the loss of ring buffers in the driver reintroduces the overhead of allocating new buffers to replace those moved to the application. Second, the loss of the application's buffers makes the output buffer reuse that is characteristic of many of the signal processing applications impossible without a data copy.

UCM I/O also provides buffers shared between the kernel and the application as part of the support for continuous media. To use this support, the application allocates a ring of buffers and attaches them to a control buffer. The kernel uses a clock interrupt to schedule the I/O of the next buffer in the ring. The approach used to support input from the GuPPI is similar to the UCM I/O's continuous media support, but with some important differences. First, the ability to use the GuPPI's interrupts rather than a fixed period clock interrupt allows the system to adapt to variance in the time required to fill a buffer. Swapping the kernel ring buffers with the buffer provided by the application allows the application to hold the contents of a buffer without the contents being bashed by the kernel when the ring wraps around. In the output direction, the need to reuse the output buffers in arbitrary orders by the application makes using a ring buffer system virtually impossible without adding data copies.

## 6 Conclusion

The GuPPI and its operating system support close the gaps preventing today's PCs from being effective platforms for intensive signal processing. By designing the hardware and software in tandem, we were able to create a system which is small and low in complexity, yet achieves excellent performance for the applications for which it was intended. Together, the hardware and software provide an application-level interface that simplifies the construction of software radios and related applications by making the analog front-end appear to be

a conventional Unix device.

The GuPPI hardware provides a high bandwidth, low latency connection between an analog front-end and the I/O bus of a PC; it makes use of a new variant of scatter/gather DMA, page-streaming that simplifies the hardware and provides for efficient use of the PCI bus. The software drives the GuPPI, smooths jitter, and makes the data transferred by the hardware accessible to applications in user space. While essential, the operating system support needed to make this functionality available to applications is minimal, and has been implemented in such a way as to have no impact on the functionality provided to other applications.

For future commodity systems, we imagine that hardware similar to the GuPPI, coupled with an appropriate, flexible front-end, would be an invaluable I/O device to integrate into a system, with the possibility of becoming as ubiquitous as serial and parallel ports are today. In addition, we have demonstrated that the necessary operating system support to make this functionality available to applications is minimal, and can be implemented in such a way as to have no impact on the functionality provided to other applications. Software modems, while a step in the right direction, are only focussed on the limited bandwidth of a telephone line and will have little application beyond it; a system like the GuPPI would continue to enable interesting and useful applications for many years to come.

## Acknowledgements

The author would like to thank the many individuals who have influenced and contributed to the development of this project. First, David Tennenhouse and John Guttag for their guidance; Vanu Bose for providing a testbed for this work; and Matt Welborn, Alok Shah, and Andrew Chiu for finding bugs by using the system for their work. 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.
- [BCT97] Vanu G. Bose, Andrew G. Chiu, and David L. Tennenhouse. Virtual Sample Processing: Extending the Reach of Multimedia. *Multimedia Tools and Applications*, 5(3):259–276, November 1997.
- [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.
- [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.
- [DDP94] Bruce S. Davie, Peter Druschel, and Larry L. Peterson. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *Proceedings of SIGCOMM '94*, September 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.
- [DPC97] Zubin D. Dittia, Guru M. Parulkar, and Jerome R. Cox. The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques. In *Proceedings of IEEE Infocom 1997*, April 1997.
- [Ins98] National Instruments. NI5102 PCI Digital Oscilloscope Data Sheet, 3 98.
- [Pai97] Vivek S. Pai. IO-Lite: A Copy-free UNIX I/O System. Master's thesis, Rice University, January 1997.
- [Sci98] Gage Applied Sciences. CompuScope 8500/PCI Hardware Installation and Reference Manual, March 1998.
- [SPL92] Lawrence C. Stewart, Andrew C. Payne, and Thomas M. Levergood. Are DSP Chips Obsolete? Technical Report CRL 92/10, Cambridge Research Laboratory, Cambridge, MA, 1992.
- [TB96] David L. Tennenhouse and Vanu G. Bose. The SpectrumWare approach to Wireless Signal Processing. *Wireless Networks*, 2:1–12, 1996.
- [Tha97] Samir R. Thadani. A Software-Based Ultrasound System for Medical Diagnosis. Master's thesis, MIT, May 1997.
- [Tra97] Mike Tramontano. Host Signal Processing: A New Way to Communicate. Technical report, Motorola ISG, 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.