Network Subsystem Design: A Case for an Integrated Data Path

Peter Druschel
Mark B. Abbott
Michael A. Pagels
Larry L. Peterson *

Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

This paper argues that the CPU/memory data path is a potential throughput bottleneck in workstations connected to high-speed networks, and considers the implications for the design of the I/O subsystem.

1 Introduction

Emerging network technologies hold the promise of delivering bandwidths approaching 1Gbps to end-user workstations. Such bandwidths should enable an exciting new class of applications. One important factor that could dramatically influence the ability of these applications to take advantage of high-speed networks is the operating system that runs on the workstation. It is the responsibility of the operating system (OS) to turn good network throughput into good application-to-application throughput.

The Network Systems Research Group at the University of Arizona is studying issues related to OS support for high-speed networking. Our work is experimental, and is being done in the context of the *x*-kernel [12, 15] integrated into the Mach operating system [2, 10]. The resulting system runs on DecStation 5000/200 and HP 9000/720 workstations connected by ATM and FDDI networks.

In way of overview, Mach provides a microkernel-based OS framework, and the *x*-kernel serves as the network subsystem. Our approach is to *logically centralize* all the network protocols in a single *x*-kernel protocol graph, and then *physically distribute* this graph throughout the system, including the OS and application protection domains. For example, Figure 1 depicts a protocol graph that spans an application task, a dedicated network server, and the kernel. The decision as to what protection domain a particular

*This work supported in part by National Science Foundation Grants CCR-9102040 and NCR-9204390, and DARPA Contract DABT63-91-C-0030

protocol belongs in is delayed until configuration time (rather than OS design time) and, therefore, can be made based on how the system configurer is willing to trade performance against trust.

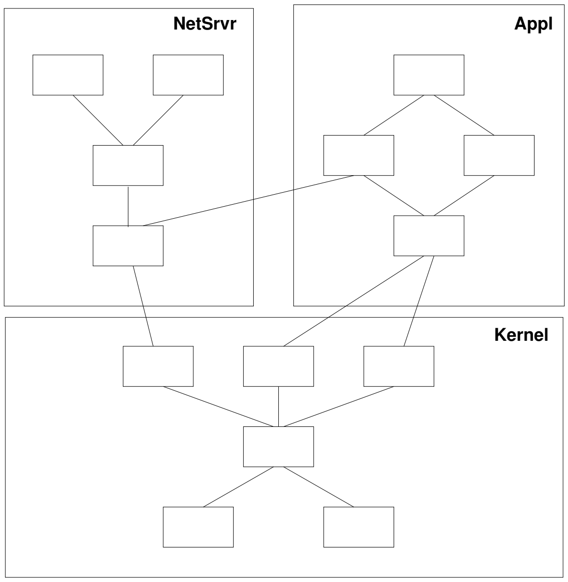

Figure 1: Distributed Protocol Graph

In our effort to optimize the performance of this architecture, one challenge has been to overcome the limitations of the workstation's memory architecture. The problem is that improvement in memory performance of workstations has not kept pace with improvements in processor performance and network bandwidth. Hennessy and Patterson report, for example, that processor performance has improved at a rate of 50–100% per year since 1985, while memory performance has improved at a rate of only 7% per year [11]. Moreover, we expect this trend to continue because cost considerations will prevent the use of dramatically faster main memory and interconnect technology in this class of machine.

Keep in mind that we are considering the movement of data through a microkernel-based system, where device drivers, network protocols, and application software all potentially reside in different protection domains. We believe that such a focus is well justified in light of the advantages of microkernel-based systems (e.g., configurability, distributability, portability, and so on), as well as the recent trend towards such systems in commercial offerings. We are attempting to demonstrate that it is possible to achieve high application bandwidth regardless of the structure of the OS.

This paper argues that the CPU/memory bandwidth of workstations will remain within the same order of magnitude as their network bandwidth, and as a consequence, the network subsystem must take care to minimize the number of trips that network data takes across the CPU/memory data path. It then surveys several techniques that can be applied to this problem. One important lesson we learn from this survey is that naively applying these techniques is not sufficient for achieving good application-to-application throughput. It is also necessary to integrate the entire data path, from a source device, through the operating system, application programs, and possibly to a sink device. The paper concludes by outlining a fully integrated data path that optimizes end-to-end throughput.

2 Performance of Workstation Hardware

This section analyzes several performance characteristics of desktop workstation hardware that affect I/O data throughput. It includes both measurements of commercially available workstations, and projections of the relevant characteristics of next-generation workstations.

Table 1 shows, for each of four commercial workstations, the peak memory bandwidth of the memory subsystem (taken from the hardware specification), and the results of several CPU/memory bandwidth measurements described below. The peak memory bandwidth is the bandwidth that the memory subsystem can deliver during a burst-mode memory transfer. The ratio of measured bandwidth to the peak memory bandwidth is shown in parentheses.

	Memory	CPU/Memory (Mbps, sustained)		
	(Mbps, peak)	Copy	Read	Write
IBM RS/6000 340	2133	405 (.19)	650 (.30)	590 (.28)
Sun SS10/30	2300	220 (.10)	350 (.15)	330 (.14)
HP 9000/720	1600	160 (.10)	450 (.28)	315 (.20)
Dec 5000/200	800	100 (.13)	300 (.38)	570 (.71)

Table 1: Memory Bandwidth of several Workstations

The CPU/Memory bandwidth numbers reported in Table 1 were obtained using a simple benchmark that measures the sustained bandwidth during a series of read, write, and copy operations. The benchmark was coded in C and compiled with each workstation vendor's native C compiler at the highest level of optimization. No attempt was made to tune the C source or the generated machine code for a particular machine. The read and write columns were measured by reading (writing) each element of an array. The copy column was measured using two methods: Element-wise assignment of two arrays, and invoking the bcopy() library function. The benchmark used arrays of type int (32bits) and double (64bits). The numbers reported are the best that could be achieved in each case; i.e., int versus double for the read and write columns, and int versus double versus bcopy() for the copy column. Finally, the size and alignment of the arrays was chosen to eliminate effects of the data cache.

The main result is that the measured bandwidths are only a fraction of the peak memory bandwidth. In particular, the read bandwidth ranges from 15 to 38%, and the copy bandwidth from 10 to 19% of the peak memory bandwidth. Note that the maximal possible copy bandwidth is one half the peak memory bandwidth. This is because copy involves moving the data across the CPU/memory bus twice.

This bandwidth degradation is the combined effect of two limitations. First, the compiler-generated machine code, and the vendor-provided implementation of the bcopy() function may be less than optimal for the intended task of the benchmark program. We have deliberately not tried to eliminate this limitation, since it equally affects real programs. Second, there is a hardware imposed limit to the bandwidth that the CPU can sustain. All memory subsystems use some form of pipelining (interleaving/page mode) to achieve high bandwidth despite the relatively high access latency of dynamic RAM. The average bandwidth drops off for small transfer sizes, where the startup latency contributes significantly to the transfer time. Data is transferred between cache and main memory at the size of a cache line, which is too small to achieve a large fraction of the peak memory bandwidth. We conclude that the CPU/memory bandwidth of current workstations is within the order of magnitude of their network bandwidth—hundreds of Mbps.

Next generation workstations will support Gbps network adapters that can transfer data into main memory at network speeds. The continuing increases in CPU speed will soon permit software data manipulations at this speed. For example, the first implementation of DEC's Alpha processor [7] permits the execution of 12 to 24 CPU instructions for every machine word in a 1Gbps data stream. However, dramatic increases in CPU/memory bandwidth are not expected. Consider what designers of the next generation of workstations might do to improve CPU/memory bandwidth. Peak memory bandwidth can be improved by increasing the memory width, but cache line size must be increased proportionally to achieve a substantial increase in CPU bandwidth. Cache line sizes that result in an optimal hit rate, however, are typically too small to achieve a substantial fraction of the peak memory bandwidth. Another approach is to reduce transfer latencies, but DRAM access times are considered to be near their technological limit. Several recently announced components integrate some form of a cache with a dynamic RAM to reduce the average access latency [6]. These integrated second level caches use large cache lines and are connected to the DRAM using wide data paths. As for any cache, the hit rate of these components depends on locality of reference. As we shall discuss in Section 3, accesses to I/O data exhibit poor locality. Consequently, we expect CPU/memory bandwidth to be of the same order of magnitude as network bandwidth for at least the next generation of desktop workstations.

3 Effects of Data Caches

Workstations employ caches to bridge the gap between CPU and main memory speeds. The idea is to keep a portion of the main memory's data in a high speed memory close to the CPU. A cache increases system performance by reducing the average access latency for data and instructions. It also reduces contention for main memory access in shared memory multiprocessors. However, the effectiveness of caches is influenced by a number of factors, such as size and organization of the cache, locality of data access, and processor scheduling.

Assuming a system that supports full application-level manipulation of high-bandwidth data, processing the data requires the CPU to inspect and possibly modify every word of a data unit, potentially multiple times. (The next section identifies several reasons why the data might be transferred across the CPU/memory data path.) This section argues that data caches are not effective in eliminating these CPU/memory transfers.

Consider the following factors, ordered according to importance.

Processor Scheduling: CPU scheduling may cause the execution of other programs to be interleaved with the processing of a data unit. By the time processing resumes, cached portions of the data unit have most likely been replaced. On a multiprocessor, the processing may even be resumed by a different CPU with its own data cache. There are a number of situations where scheduling occurs during the processing of a data unit. When the data unit is passed on to another thread (i.e., queued), a processor must be scheduled to run that thread. Queuing typically occurs at the user/kernel boundary, in certain protocols, and between the device driver's interrupt handler and the top half of the driver. In the worst case, additional queuing may occur between protocol layers. Moreover, hardware interrupts, and the events they signal, trigger processor rescheduling.

Cache Write Policy: Uniprocessor data caches often employ a write-through policy, meaning that every store operation requires a write to main memory. Write buffers are typically used with write-through caches to decouple the CPU from the timing of memory writes. However, many consecutive writes—such as would occur when reading and writing every word of a data unit—could still cause the CPU to stall on store instructions.

Cache Lookup: Caches that are virtually indexed and tagged do not require a virtual-to-physical address translation to access cached data. With this approach, cached data from virtually shared pages cannot remain valid across protection domain boundaries. Physically tagged caches do not have this problem. They require, however, that a *Translation Lookaside Buffer (TLB)* entry be active for the page that contains the referenced data.

Cache Size: Data caches, particularly fast on-chip caches, are limited in size. For data to remain cached during a data inspection/modify step that involves loading and storing every word, the cache must be at least twice the size of the data unit. In practice, cache size requirements are further increased by cache line collisions due to limited associativity of the cache, and by accesses to program variables during and between data manipulations.

To quantify the effectiveness of the data cache in handling network data, we conducted some preliminary experiments that measure the amount of network data resident in the cache after OS processing. This cache residency value is a measure of the potential benefit obtained by a user process due to the caching of network data, thereby avoiding CPU/memory transfers. It provides an upper bound on the benefits provided by the cache when network data is processed without copying.

These experiments were run on HP 9000/720 workstations running Mach 3.0 (MK67). These workstations employ a 256 KByte virtually indexed, physically tagged data cache. Mach 3.0 implements Unix system calls via a user process called the Unix Server. The Unix Server receives messages from user processes requesting that Unix system call functionality be performed, and from the Mach microkernel to enable the processing of device requests. Incoming network packets are acquired by the microkernel in an interrupt service routine, copied and encapsulated in a Mach message and then sent to the Unix Server for processing through the Unix Server's protocol stack. If a network packet contains data destined for another

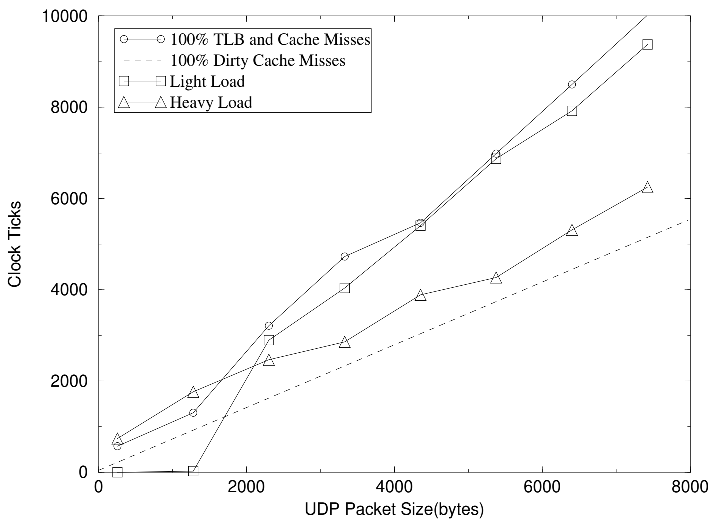

Figure 2: Network buffer data access times.

user process, it must be copied from the Unix Server's address space into the destination's address space. As Mach's Unix Server contains all of the network code found in a typical monolithic Unix implementation, its behavior, from the receipt of the network data message to final user copyout closely mimics a monolithic Unix's processing of an incoming network packet.

Our experiment measures the amount of network data resident in the cache just before the Unix Server copies it into the user's address space. If the Unix Server implemented a copy-free network data system the cache residency seen by the Unix Server at the copy-out point would be seen by the receiving user process. Details of the modifications made to the Mach Unix Server to collect these data can be found in [16].

Experimental data was collected in two settings. In the first, the receiving processor was lightly loaded by running only normal system programs and the experimental receiving program. In the second, the receiving processor was heavily loaded by being the recipient of 8000 byte UDP packets transmitted as fast as possible by a third processor.

Figure 2 plots the number of clock cycles beyond that expected if all of the accessed network buffer data were found in the cache, versus the size of the UDP datagram. The "100% Dirty Cache Misses" line indicates the number of clock cycles required to load into the cache from main memory all of the cache lines used in the network data buffers. We note that in both settings our measurements exceed the number of cycles explained by a 100% cache miss rate.

In our timing loop there are three other possible clock cycle contributors. As the Unix Server is a user process it may be context switched while the timing loop is executing. If this occurs, more than 10 000 cycles are added to the measured time. These occurrences are obvious in the data and have been excluded from the graph. *Direct Memory Access* (DMA) by I/O devices can lengthen the number of cycles required

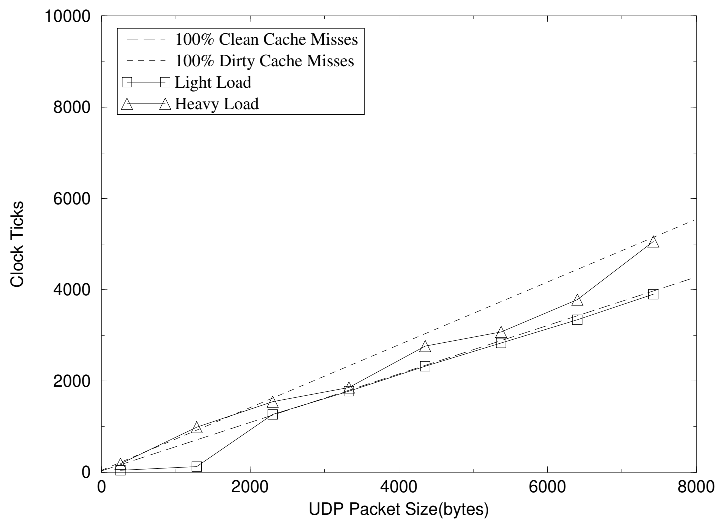

Figure 3: Network buffer data access times when forced into TLB.

to perform a cache line fill by preventing the cache from accessing the memory until the DMA transfer has completed. This effect should be most pronounced in the heavily loaded case as the ethernet network device uses DMA to transfer data between the network adaptor and memory. The HP 720 processor uses a virtually addressed, physically tagged cache which requires that for data to be accessed from the cache an entry must exist in the processor's TLB. Experimental measurements indicate that TLB miss processing under Mach 3.0 require 130 clock cycles. Due to the algorithms used in Mach to handle TLB reloads and the structure of network data buffer, it is possible for a single buffer access to require four TLB loads. The "100% TLB and Cache Misses" line in the graph displays the accumulation of worst case TLB and cache miss clock cycle times. This line provides an upper bound assuming that DMA accesses have minimal effect.

Interestingly, the heavily loaded case had a lower access time than the lightly loaded case. As single memory pages contain more than one network buffer, we posit that in the heavily loaded case network data processed for the flooding sender preloads into the TLB memory pages that are then used by the distinguished sender. In order to minimize the effects of TLB loading another experiment was conducted where before the timing loop was run the memory page on which the network buffer resided was loaded into the TLB. Special care was taken to insure that loading of the TLB did not cause any of the network buffer data to be loaded into the cache.

In Figure 3 we note that all of the measured times are below the "100% Dirty Cache Misses" line and that the heavily loaded time is now larger than the lightly loaded. The linear nature of the lightly loaded times may indicate that rather than experiencing a miss rate of approximately 72% on dirty cache lines, instead we are observing a 100% miss rate on clean cache lines[1].

[1]Cache lines not requiring writeback to memory require fewer clock cycles to load thereby increasing the time derived miss rate.

Our experiments demonstrated the existence of three significant contributors to limitations on network buffer data access times. First, DMA used by other system devices may increase contention for access to memory. This contention effectively increases the amount of time it takes to load data into the processor's cache thereby increasing average data access times. Second, on processors with physically tagged caches contention for TLB slots caused by the fragmentation of network data over multiple virtual memory pages produced significant access overhead. Finally, with TLB slot contention eliminated we measured an average cache hit rate of 28%, implying that the cache has a minimal positive effect on the transfer of network data between the Unix Server's and user's address spaces. This low cache residency implies that even with copy-free techniques the cache will not provide the user with significantly improved average network buffer data access times, nor act to minimize CPU/memory transfers.

4 Avoiding Data Transfers

The two previous sections document our argument that the CPU/memory bandwidth of workstations is within an order of magnitude of the network bandwidth, and that data caches have a minimal effect in reducing CPU/memory traffic for network data. It follows, therefore, that in order to preserve the bandwidth on the data path from the source (e.g., network device), through the OS and application, and possibly to a sink (e.g., a display device), multiple transfers of the data between the CPU and memory must be avoided.

Each of the following subsections identifies a potential data transfer along this path, briefly describes one or more technique for efficiently handling or avoiding it, and discusses the assumptions and limitations of these techniques.

4.1 Device/Memory Transfers

Data must be moved between main memory and network/device adapters. The techniques most commonly used are DMA and *Programmed Input/Output* (PIO). DMA implies that the I/O adapter transfers data directly from/to main memory, without involving the CPU. PIO requires the processor to transfer every word between main memory and I/O adapter in a programmed loop.

With DMA, it is generally possible to transfer large blocks of data in a single bus transaction, thereby achieving transfer rates close to the limits of main memory and I/O bus. The data transfer can proceed concurrently with activity by the processor(s), although contention for main memory access may induce processor stalls during periods of heavy DMA traffic. On the downside, DMA requires some complexity in the device adapters. Also, data caches are often not coherent with respect to DMA transfers. That is, after a DMA transfer from an I/O adapter to main memory, the cache may contain stale data. Consequently, the appropriate portions of the data cache must be invalidated by system software.

With PIO, the CPU is occupied during transfers from/to a device. Only a small fraction of the peak I/O bandwidth is often achieved, for the following reasons. The adapter's control and data ports can either be mapped as cacheable or non-cacheable memory locations. In the non-cacheable case, each load/store instruction results in a short (i.e., one machine word) I/O bus transaction, resulting in poor sustained bandwidth. In the cacheable case, I/O bus transfers occur at cache line length. The resulting bandwidth is improved, but may still be far below the peak I/O bandwidth. Moreover, it is again necessary to flush the data cache to maintain consistency with the adapter's ports.

Nevertheless, there are situations where PIO can be preferable over DMA. First, computations on the data that occur in the kernel, such as checksum calculations, can sometimes be integrated with the PIO data movement, saving one trip to main memory. Second, after a programmed data movement from an I/O adapter to main memory, the data is in the cache. This can result in reduced memory traffic if the data is accessed while it remains in the cache.

A scatter-gather capability in DMA-based I/O adapters is important for reducing memory traffic. Scattering allows an incoming data unit to be placed into multiple, non-contiguous portions of main memory. Gathering allows an outgoing data unit to be collected from multiple, non-contiguous buffers. Scatter-gather allows DMA transfer from/to non-contiguous physical page frames, which greatly simplifies physical memory management in the operating system, and helps avoid copying data into contiguous storage. With PIO, scatter-gather can be trivially implemented in software.

Network adapters may support packet demultiplexing prior to data transfer to main memory, allowing filtering and selective placement of data units in memory. In the simplest case, the adapter allows the host to peek at a network packet's header. The host makes the demultiplexing decision and initiates the data transfer to the appropriate location in main memory, using DMA or PIO. More elaborate adapters can be programmed by the host CPU to automatically recognize network packets by matching their headers, and place them into appropriate memory locations using DMA [13, 3].

4.2 Cross-Domain Transfers

Protection necessitates the transfer of data between protection domains. In the simplest case, an I/O data stream is handled by a single application process running on top of a conventional monolithic kernel. In this case, each data unit must cross the user/kernel boundary. In general, additional user processes, such as window managers and multimedia servers, and the trend in operating system design towards a kernelized system structure, introduces additional domain boundaries into the I/O data path.

Software data copying as a means of transferring data across domain boundaries exacerbates the memory bottleneck problem. A number of techniques exist that rely on the virtual memory system to provide copy-free cross-domain data transfer. *Virtual page remapping* [4, 18] unmaps the pages containing data units from the sending domain and maps them into the receiving domain. *Virtual copying* [9] shares the transferred pages among the sending and receiving domain, and delays copying until one of the sharing domains attempts to write the shared data unit. *Shared virtual memory* [17] employs buffers that are shared among two or more domains to avoid data transfers.

Virtual page remapping has move rather than copy semantics, which limits its utility to situations where the sender needs no further access to the transferred data. Virtual copying has copy semantics, but it can only avoid physical copying when the data is not written by either the sender or the receiver after the transfer. Both techniques require careful implementation to achieve low latency. The time it takes to switch to supervisor mode, acquire necessary locks to VM data structures, change VM mappings—perhaps at several levels—for each page, perform TLB/cache consistency actions, and return to user mode poses a limit to the achievable performance. Measurements we performed on a DecStation 5000/200 suggest that page remapping is not fast enough to sustain the bandwidth of a high-speed network adapter [8].

Another complication arises from the fact that both techniques work at the granularity of the VM page size. A mismatch between data unit size and VM page size implies that a portions of the last page overlapped by the data unit will remain unused. For reasons of data privacy, the kernel must clear (e.g., fill with zeroes) this portion of a newly allocated buffer page, which can incur a substantial cost, relative to the overall overhead of buffer management and cross-domain data transfer [8]. A similar problem occurs if headers at the front of a received network packet contain sensitive data that must be hidden from user processes. The partial use of memory pages also requires more pages per amount of network data, resulting in increased physical memory consumption, page allocation and remap overhead, and demand for TLB entries.

Shared virtual memory avoids data transfer and its associated costs altogether. However, its use may compromise protection and security between the sharing protection domains. Since sharing is static—a particular page is always accessible in the same set of domains—a priori knowledge of all the recipients of a data unit is required. A novel technique, called *dynamic page sharing (fbufs)* [8], combines page remapping with shared virtual memory, and exploits locality in network traffic to overcome some of the shortcomings of either technique.

4.3 Data Manipulations

Data manipulations are computations that inspect and possibly modify every word of data in a network packet. Examples include encryption, compression, error detection/correction, and presentation conversion. Data manipulations can be performed in hardware or software. Hardware support for data manipulations can reduce CPU load, and when properly integrated, reduce memory traffic. For certain manipulations like video (de)compression, hardware support may be necessary in the short term, due to the computational complexity of the task. To rely on hardware for all data manipulations, however, seems too constraining for innovative high-bandwidth applications.

Software data manipulations typically access the data independently from each other since they are generally part of distinct program modules such as protocol layers. Consequently, each data manipulation requires potentially uncached data to be loaded from memory and—if the manipulation is read-write—stored back to memory. These repeated transfers across the CPU/memory data path frequently dominate the time required to process a message, and thus the bandwidth.

Such data transfers can be minimized using a technique called *Integrated Layer Processing* (ILP) [5, 1]. ILP is a technique for implementing communication protocols that avoids accessing memory for data manipulations. The data manipulation steps from different protocols are combined into a pipeline. A word

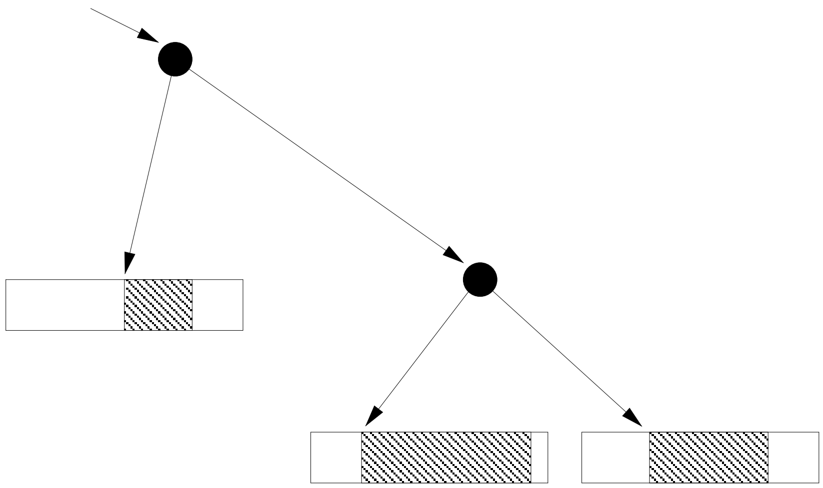

Figure 4: Implementation of Messages in the *x*-kernel

of data is loaded into a register, then manipulated by multiple data manipulation layers while it remains in a register, then finally stored—all before the next word of data is processed. In this way, a combined series of data manipulations only transfers data from memory to the CPU and back once, instead of potentially transferring the data once per distinct layer.

A detailed performance study demonstrates that integration can have a significant impact on performance [1]. On the other hand, a major limitation of ILP is that data manipulations performed by program modules in different protection domains cannot be easily integrated.

4.4 Buffer Management

Buffer editing—which we distinguish from data manipulations that require the inspection and/or modification of each word of the data—can be expressed as a combination of operations to *create*, *share*, *clip*, *split*, *concatenate*, and *destroy* buffers.

A buffer manager that employs lazy evaluation of buffers to implement the aforementioned primitives can facilitate copy-free buffer editing. The manager provides an abstract data type that represents the abstraction of a single, contiguous buffer. An instance of this abstract buffer type might be implemented as a sequence of not necessarily contiguous fragments. For example, the *x*-kernel represents messages with a tree, where the leaves of the tree correspond to buffers, and the leftmost leaf can have information (e.g., headers) prepended to the front. Figure 4 depicts *x*-kernel messages.

Since buffer editing occurs frequently in network protocol implementations, buffer managers are used in the network subsystem of many operating systems [14]. The scope of these managers is restricted to a single protection domain, typically the kernel. In most systems, a software copy into a contiguous buffer is necessary when a data unit crosses a protection domain boundary.

4.5 Interface Design

The *Application Programming Interface* (API) defines the semantics for data transfer to and from the application program. The semantics defined by this interface can be a significant factor in the efficiency of the implementation. Consider, for example, the UNIX read() and write() system calls. Applications programs may choose a contiguous data buffer in their address space with arbitrary address, size, alignment, and have unconstrained access to that buffer. This low-level representation of data buffers makes it difficult for the system to avoid copying of data.

First, all VM based techniques for cross-domain data transfer operate at the granularity of a VM page. If the user buffer's first and last address are not page aligned, the system must copy portions of the first and last page overlapped by the buffer. Depending on the length of the user buffer, this can eliminate most of the benefits of using a copy-avoiding technique. Second, semantics of the write system call permit the user process to modify (reuse) the user buffer immediately after the call returns. If this happens, the system must either copy the affected page, or block the user process until the system is done with the previous data. The first approach re-introduces a copy, the second approach may degrade the user processes' effective I/O bandwidth.

An API that lends itself to efficient data transfer methods should use an abstract data type to represent data buffers. Application programs access the buffer type indirectly through its operations. They ask the system to create an instance when they need a buffer, pass/receive instances as arguments/results in system calls, and ask the system to deallocate the instance when the associated buffer is no longer needed. The system then has complete control over buffer management, including address, alignment, method of transfer, and so on. A proposal for an API of this kind is presented in Section 6.

5 End-to-End Design

This section analyzes the design space for an I/O subsystem that optimizes data-dependent costs along the entire data path from source to sink device. The goal is to minimize the number of CPU/memory transfers. We analyze several representative sample points in the design space. For each sample point, we discuss tradeoffs, determine the optimal data path, and select appropriate implementation techniques to achieve the optimal data path.

Throughout this section, we refer to the block diagrams in Figure 5 to depict the data flow from source to sink device. In the architectural model underlying these diagrams, a single-level cache is connected to main memory using a dedicated memory bus. A bus converter ties the memory bus to a separate I/O bus, to which all devices are attached.

5.1 Hardware Streaming

One approach to avoiding the CPU/memory bottleneck is to remove both the CPU and main memory from the data path. The data path may be set up and controlled by software, but the data itself is transferred from

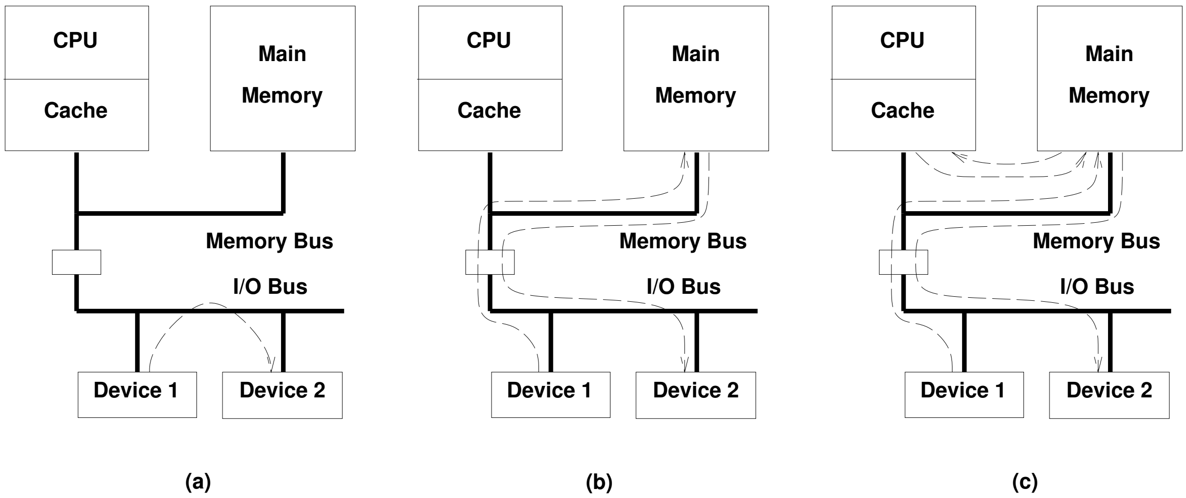

Figure 5: Various Forms of Data Streaming

source to sink device without CPU involvement.

In the simplest case, both the source and sink device are integrated on the same I/O board. Data never reaches the I/O bus, memory, or CPU. Data can only be exchanged among the devices integrated on the same board, and functionality is restricted to that provided by the board. The second approach uses peer-to-peer I/O bus transactions to transfer data directly from source to sink device, bypassing both main memory and the CPU (Figure 5, (a)). This method offers some limited hardware configurability. Device adapters must support peer-to-peer bus transfers and have the capability to perform demultiplexing, and any necessary data format conversions.

Both methods can be characterized by a lack of integration with the host computer system. Since application programs have no access to the data, they are constrained by the functionality provided by the adapter boards. The result is a tradeoff between complexity in the device adapters on one hand, and flexibility and functionality on the other.

Adapters that support a fixed set of capabilities offer little room for innovation in high-bandwidth applications. An alternative is for the adapter to provide components that can be programmed by applications. This can take the form of special-purpose processors, or programmable circuitry. Unfortunately, this approach has problems of its own. First, it is difficult for applications to use programmable device adapters in a portable way. Second, at any given point in time, the technology used in I/O adapter hardware tends to lag behind that of the main computer system, due to economics. Consequently, applications that rely on out-board processing may not be able to exploit performance gains resulting from an upgrade of the host computer system. An exception are computations whose performance is bound by the characteristics of the device, e.g., network link speed. Performing such computations in the adapter seems appropriate, since they cannot take advantage of increased processing speed [3].

With hardware streaming, no specific techniques are required for data transfer in the I/O subsystem.

However, many of the issues discussed in this paper may apply to the *internal* design of the adapter boards.

5.2 DMA-DMA Streaming

A second approach is for data to be transferred using DMA from a source device to buffers in main memory, where some buffer editing may occur under control of the operating system, and then, the data is transferred to a sink device using DMA (Figure 5, (b)). The CPU controls data transfers, may change the size of the data units and control information (headers), but remains removed from the data path. Unlike the previous approach, DMA-DMA streaming requires no special hardware other than DMA support on both the source and sink device adapters. Consequently, generic devices (e.g., disks) can be used.

Two DMA I/O bus trips are required by this approach. It follows that the throughput is bounded by one half of the I/O bus bandwidth. In practice, the sustainable throughput is lower, since main memory accesses caused by concurrent CPU activity compete with the DMA transfers for main memory access, even if the I/O bus and memory bus are separated.

Two key techniques are required to keep the CPU removed from the data path in this approach: scatter-gather DMA support on the device adapters, and a buffer manager that supports copy-free buffer editing. Support for gather DMA in the sink device adapter is critical. Recall that a lazily evaluating buffer manager may cause a buffer to consist of multiple discontiguous fragments. In the absence of a gather capability, it is necessary to copy data units into contiguous space prior to a transfer to the sink device.

5.3 OS Kernel Streaming

Now consider systems where software data manipulations are performed, but all the manipulations are executed within the privileged kernel protection domain. In other words, the data is not accessed in any user-level protection domain. Clearly, the data must pass through the CPU/data cache (Figure 5, (c)), implying that the achievable throughput is bounded by the system's copy bandwidth. The goal is to keep the resulting CPU/memory data traffic to the minimum, namely two transfers. The solution is to integrate data manipulations—if there is more than one—using, for example, ILP. Note that data manipulations may include data movement from and to devices via PIO.

If a device supports both DMA and PIO, it may be beneficial to use PIO in this type of system, since programmed data movement can be integrated with other data manipulations. That is, instead of first using DMA and then loading/storing data to/from main memory, the CPU could directly load data from the source I/O adapter, and store data to the sink I/O adapter, bypassing main memory. This approach saves two DMA bus trips, which would otherwise compete with the CPU for memory access. However, it also trades memory accesses for potentially much slower load and store operations across the I/O bus. Which approach results in a more efficient data path depends on the relative performance of memory accesses and DMA transfers, versus PIO on the target hardware.

Unlike the previous methods, OS kernel streaming offers full programmability of the data path. However, all data manipulations must be performed in the kernel of the operating system. Applications and user-level servers are restricted to the set of data manipulations implemented by the kernel.

5.4 User-Level Streaming

Next consider systems where data passes through the kernel, plus one or more user-level protection domains. These user-level domains could be part of a kernelized operating system, implement third-party servers for windowing, or be part of a high-bandwidth application. In addition to the issues discussed in the previous subsections, the designer is faced with protection boundaries. Protection domain boundaries are an obstacle to the integration of data manipulations. We conclude, therefore, that designers should make every effort to locate all data manipulation functions in the same domain. Protection also requires an efficient method for transferring data between domains.

Section 4.2 discusses several VM techniques for cross-domain data transfer. The use of such a facility is critical for data transfer between two user-level protection domains. Otherwise, two software copies are required for user-to-user transfers on most systems—from the source user domain to a kernel buffer and from the kernel buffer to the target user domain.

As a special case, data transfer between a user domain and the kernel domain can be implemented without extra cost, if the transfer is combined with the data movement between I/O adapter and main memory. That is, data is transferred directly between the I/O adapter and the user buffer. However, there are several problems with this approach when applied to network I/O. On the receiving end, the data transfer must be deferred until the appropriate user buffer can be determined, which may depend on the behavior of the user program. This can lead to contention for buffers in the network adapter. Support for packet demultiplexing is required in the network adapter, since the receiver must be determined, prior to the data transfer to a user buffer. On the transmitting side, network protocols may need to retain a copy of the data for retransmission, but the only copy of the data is in a user buffer. Finally, a system that relies on this approach as the sole means of cross-domain data transfer is constrained to a monolithic OS structure, and therefore limits the data to pass through a single user domain.

5.5 Need for Integration

The API, cross-domain data transfer facility, and buffer manager must be integrated in a manner that takes into account their subtle interactions. Consider the effects of naively applying efficient cross-domain data transfer and lazily evaluated buffer management to a system with a UNIX-style API. The buffer manager is restricted to the kernel domain. A virtual copy facility is used for cross-domain data transfer. The system API specifies a pointer to a contiguous buffer in the user domain's address space.

In this case, data units from the source device are placed in main memory buffers, and some buffer editing occurs as part of the in-kernel I/O processing; e.g., reassembly of network packets. When a data unit represented by a lazily evaluated buffer reaches the user/kernel boundary, it must be evaluated (copied into contiguous storage), despite the use of a virtual copy facility. The reason is that the interface defines data buffers to be contiguous. Since the API allows applications to specify an arbitrarily aligned buffer address and length, the buffer's first and last address may not be aligned with page boundaries. Consequently, the data transfer facility may be forced to copy the portion of the first and last page that is overlapped by the buffer.

Once in the user domain, more buffer editing may need to be performed. Since the buffer management tool is not available at user-level, the application must either perform data copying, or employ its own lazily evaluating buffer management. In the latter case, another copy is required when data crosses the next domain boundary. After data is transferred back to the kernel, the semantics of the API allow the user program to reuse the data buffer instantly, which is likely to force the virtual copy facility to copy parts of the buffer. We conclude that, despite the use of copy-avoiding techniques, multiple copies occur along the data path, and these copies are an artifact of the poor integration of these techniques.

One problem is that the implementation of the buffer manager is local to the kernel domain; a global implementation is necessary to maintain the lazily evaluated representation of data buffers along the entire data path. A global abstract buffer type has the additional benefit that all domains (including applications) can perform copy-free buffer editing. A second problem is the API, which commonly does not permit a non-contiguous representation of buffers, and as a consequence, stands in the way of efficient data transfer. A potential third problem is the cross-domain data transfer facility's inability to efficiently support the transfer of non-contiguous buffer fragments.

Figure 6 depicts some of the dependencies among the copy-avoiding techniques. The boxes represent situations where copies may occur along the data path, and the bubbles correspond to the appropriate copy-avoiding techniques. Edges indicate dependencies among the techniques that must be satisfied to achieve an efficient data path. For example, VM techniques for data transfer, the API's data passing semantics, and the buffer manager must agree on a common buffer representation. The dashed lines indicate that we currently don't know how to effectively integrate ILP with API and data transfer facility, to permit integration of data manipulations across protection domain boundaries.

In conclusion, it is necessary to maintain a lazily evaluated representation of data buffers along the entire data path. This implies that all programs must deal with this representation of buffers. Consequently, a global buffer manager is needed that is integrated with the API and cross-domain transfer facility. The choice of a cross-domain data transfer method may further influence the design of a network adapter. For example, shared virtual memory requires demultiplexing prior to the data transfer from adapter to main memory.

6 An Integrated Data Path

This section proposes an API based on an abstract buffer type that permits the integration of the API, buffer manager, and cross-domain data transfer facility to achieve an efficient end-to-end data path. The approach hinges on an abstract data type called *IOData*. This section gives an overview of the IOData type design, discusses its use, and briefly sketches how it can be implemented using different buffer management schemes and cross-domain data transfer facilities.

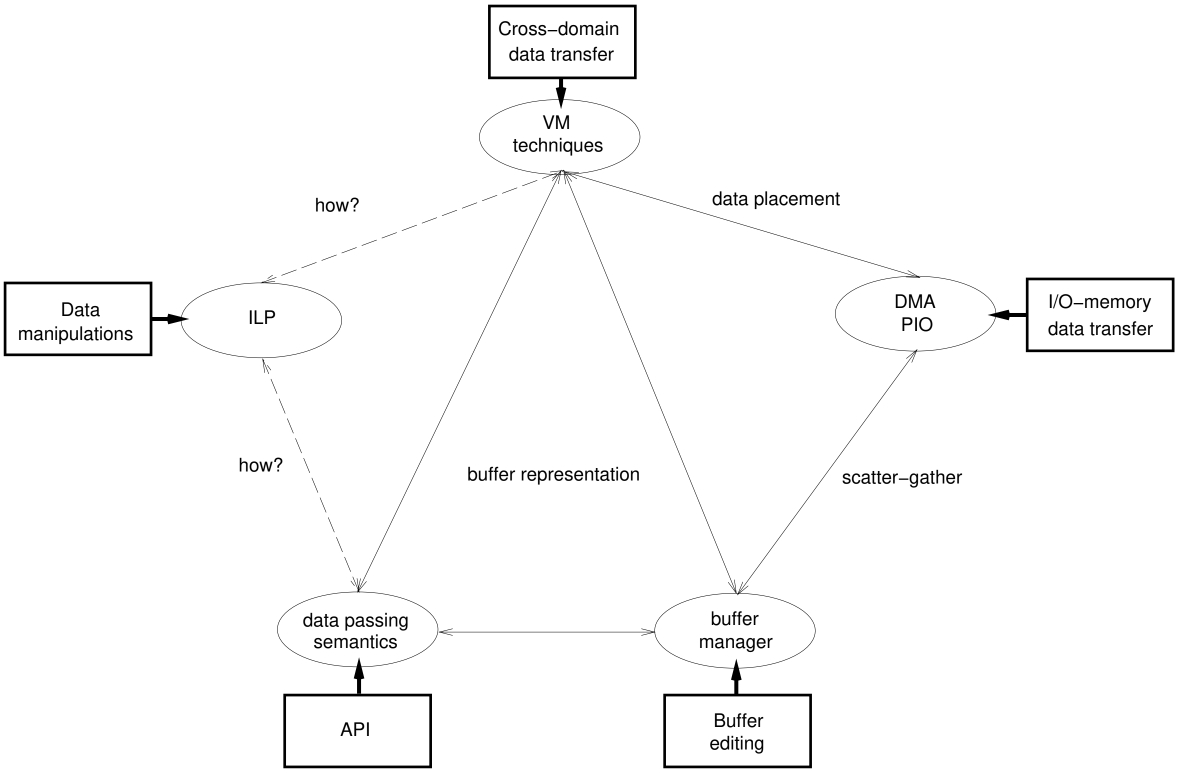

Figure 6: Dependencies among copy-avoiding techniques

An instance of the IOData type represents an abstract data buffer of arbitrary length. It encapsulates one or more physical buffers that contain the data. At any given time, the physical buffers may not be contiguous, mapped in the current domain, or even in main memory. The IOData type is *immutable*, i.e., once an instance is created with an initial data content, the content cannot be subsequently changed. IOData instances can be manipulated using a well-defined set of operations. An implementation of this abstract type—i.e., code that implements its operations—is included as part of a library in each protection domain. The exact form and syntax of IOData's operations depends on the programming language used, which may vary from domain to domain.

The use of the IOData type for data buffers has important advantages. First, it ensures that a single buffer representation can be maintained along the entire data path, permitting lazy buffer evaluation. Second, it isolates applications, user-level servers, and large parts of the kernel from details of buffer management and cross-domain data transfer. This increases portability of both applications and operating system, and permits the use of the most efficient buffer management and data transfer techniques on different platforms. Third, the IOData type gives applications access to efficient buffer manipulation operations, and eliminates the need for separate application-level buffer management.

The IOData type supports the following operations. An *allocate* operation creates a new instance of the requested size and allocates an appropriate number of physical buffers. During an initialization phase, the client is provided with a list of pointers to these physical buffers, for the purpose of initialization. A *share* operation creates a logical copy of an IOData instance; it does not actually copy the physical buffers. *Clip*, *split*, and *concatenate* operations implement the necessary buffer editing operations. A *retrieve* operation generates a list of pointers to the physical data buffer fragments, thereby allowing the client to access the data. A *mutate* operation is a combination of retrieve and allocate. It allows a client to read the data from an IOData instance, and store the (perhaps modified) data into a new IOData instance. The operation generates a list of pointer pairs, one referring to a fragment of the source, the other pointing to a physical buffer of the target. Finally, a *deallocate* operation destroys an IOData instance, and deallocates the physical buffers if no logical copies of the data remain.

Consider an implementation of the IOData type. One key feature is that the implementation has complete control over the size, location, and alignment of physical buffers. Consequently, a variety of buffer management schemes are feasible. All buffers may be part of a system-wide pool, allocated autonomously by each domain, located in a shared VM region, or they may reside outside of main memory on an I/O adapter. Physical buffers can be of a fixed size to simplify and speed allocation. The other key feature of the IOData type is its immutability. It allows the transparent use of page remapping, shared virtual memory, and other VM techniques for the cross-domain transfer of IOData instances. Virtual copying can be used with increased efficiency since physical buffers are guaranteed not to be written after a transfer.

It is possible to extend an existing API (such as that of UNIX) to include input/output operations based on the IOData type. New applications that depend on high bandwidth (such as multimedia) can use the new interface. The conventional interface can be maintained for backward compatibility.

7 Concluding Remarks

This paper has argued that the bandwidth of the CPU/memory data path on workstations will remain within the same order of magnitude as the network bandwidth delivered to the workstation. This makes it essential that the number of times network data traverses the CPU/memory data path be minimized. Moreover, evidence suggests that the cache cannot be expected to significantly reduce the number of data movements over this path.

This paper goes on to discuss hardware and software techniques for avoiding the CPU/memory bottleneck. One important conclusion is that naively applying these techniques is not sufficient for achieving good application-to-application throughput; one must also carefully integrate the techniques. Toward this end, we outline how various techniques can be integrated to provide a high bandwidth data path between I/O devices and application programs.

Acknowledgements

We would like to thank Steve Scott for providing technical information on IBM RS/6000 workstations and for running our benchmark on these machines.

References

[1] M. B. Abbott and L. L. Peterson. Automated integration of communication protocol layers. Technical Report TR 92-24, Department of Computer Science, University of Arizona, Tucson, Ariz., Dec. 1992.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the USENIX Summer '86 Conference*, July 1986.

[3] M. L. Bailey, M. A. Pagels, and L. L. Peterson. The *x*-chip: An experiment in hardware multiplexing. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, February 1992.

[4] D. R. Cheriton. The V distributed system. *Commun. ACM*, 31(3):314–333, Mar. 1988.

[5] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 200–208, Philadelphia, PA, Sept. 1990. ACM.

[6] R. Comerford and G. F. Watson. Memory catches up. *IEEE Spectrum*, 29(10):34–57, Oct. 1992.

[7] Digital Equipment Corporation, Palo Alto, California. *Alpha Architecture Technical Summary*, 1992.

[8] P. Druschel and L. L. Peterson. High-performance cross-domain data transfer. Technical Report TR 93-5, Department of Computer Science, University of Arizona, Tucson, Ariz., Mar. 1993.

[9] R. Fitzgerald and R. F. Rashid. The integration of virtual memory management and interprocess communication in Accent. *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[10] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proceedings of the USENIX Summer '90 Conference*, pages 87–95, June 1990.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., Palo Alto, California, 1990.

[12] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[13] H. Kanakia and D. R. Cheriton. The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 175–187, Stanford, CA, Aug. 1988. ACM.

[14] S. J. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[15] S. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2), May 1992.

[16] M. A. Pagels and L. L. Peterson. Cache and tlb effectiveness in the processing of network data. Technical Report TR93-04, University of Arizona, Department of Computer Science, 1993.

[17] M. D. Schroeder and M. Burrows. Performance of firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, Feb. 1990.

[18] S.-Y. Tzou and D. P. Anderson. The performance of message-passing using restricted virtual memory remapping. *Software—Practice and Experience*, 21:251–267, Mar. 1991.