This article was processed using the LaTeX macro package with LLNCS style

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.

Acknowledgement

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 92-25, Department of Computer Science, University of Arizona, Tucson, Ariz., Dec. 1992.
2. D. R. Cheriton. The V distributed system. *Commun. ACM*, 31(3):314–333, Mar. 1988.
3. 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.
4. R. Comerford and G. F. Watson. Memory catches up. *IEEE Spectrum*, 29(10):34–57, Oct. 1992.
5. Digital Equipment Corporation, Palo Alto, California. *Alpha Architecture Technical Summary*, 1992.
6. P. Druschel and L. L. Peterson. High-performance cross-domain data transfer. Technical Report TR 92-11, Dept. of Comp. of Sc., U. of Arizona, Tucson, AZ (USA), Mar. 1992.
7. 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.
8. R. Govindan and D. P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 68–80. Association for Computing Machinery SIGOPS, October 1991.
9. 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.
10. S. J. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.
11. 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.

6 An Integrated I/O 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.

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 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.

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 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

5.5 Need for Integration

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; due to per-operation latencies, it works most efficiently for large, contiguous buffers. 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.

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, virtual shared memory requires demultiplexing prior to the data transfer from adapter to main memory.

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 multimedia 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 multimedia support or windowing, or be part of a multimedia 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 requires that data passes only through a single user domain.

multimedia hardware on one hand, and flexibility and programmability on the other. In either case, software programs have no access to the data, and are constrained by the functionality provided in hardware. Thus, hardware streaming offers little room for innovation in multimedia applications. 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 multimedia 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. 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 to support multimedia.

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. A DMA scatter ability of the source device is not strictly necessary, but it allows the use of a fixed buffer size, which may lead to a more efficient implementation of the buffer manager.

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. 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 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

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.

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 of sample point, we discuss tradeoffs, determine the optimal data path, and select appropriate implementation techniques to achieve the optimal data path.

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 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 intergrated 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. 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 workstation computer system. The result is a tradeoff between complexity in the

4.3 Data Manipulations

Data manipulations are computations that inspect and possibly modify every word in a data unit. 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 multimedia applications.

Since software data manipulations are often associated with distinct program modules, they are typically implemented in series, resulting in poor locality of data access. This can lead to serious bandwidth degradation due to repeated transfers across the CPU/memory data path. Techniques for avoiding this bottleneck in the implementation of network protocol suites are known as *Integrated Layer Processing* (ILP) [3, 1]. ILP combines several serial data manipulation steps, so that they can be performed while a particular data item is in registers or cache. The basic approach is to restructure the software so that all data manipulations are performed during one traversal of a data unit. This results in a much improved locality of data access, so that each data item can remain in the data cache or in registers until all data manipulations have been performed on that item. A major limitation of ILP is that data manipulations performed by program modules in different protection domains cannot be effectively 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.

Since buffer editing occurs frequently in network protocol implementations, buffer managers are used in the network subsystem of many operating systems [10]. However, the scope of these managers is restricted to a single protection domain, typically the kernel. Consequently, an evaluation of buffers—that is, a software copy into a contiguous buffer—is necessary when a data unit crosses a protection domain boundary.

I/O bus transaction. Consequently, only a small fraction of the peak I/O bandwidth is typically achieved using PIO.

Network adapters may support some 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 [9].

4.2 Cross-Domain Transfers

Protection necessitates the transfer of data between protection domains. In the simplest case, CM data 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* [2, 11] unmaps the pages containing data units from the sending domain and maps them into the receiving domain. *Virtual copying* [7] 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, in the hope that the data won't be written at all. *Shared virtual memory* 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 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, perform TLB/cache consistency actions, and return to user mode poses a limit to the achievable performance of both techniques.

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. *Dynamic shared memory* [6] is a variation of this technique that attempts to overcome these shortcomings.

to another address space. However, some machines use virtually indexed caches tagged with an address space identifier. With these caches, accesses to cached data from shared pages cause cache misses after a processor reallocation.

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.

4 Avoiding Data Transfers

The previous two 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 resulting from CM data access. 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, to the 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 *Direct Memory Access* (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, the data transfer can proceed concurrently with other activity by the processor(s), allowing a degree of parallelism. Moreover, 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. 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 data cache may be stale. Consequently, there are costs due to the necessary flushing of the data cache, and subsequent cache misses when the data is accessed by the CPU.

PIO maintains data cache coherency. On the downside, CPU cycles are consumed, and a transfer between main memory and an I/O adapter requires two bus trips, compared to one for DMA. In general, data in memory mapped I/O locations cannot be cached, meaning that each load/store instruction causes an

of a cache with a dynamic RAM to reduce the average access latency [4]. 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 CM 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 an integrated system that supports full application-level programmability of CM data—i.e., ignoring workstations with a built-in TV set—processing CM data requires the CPU to inspect and possibly modify every word of a CM data unit, potentially multiple times. (The next section identifies several reasons why CM 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 indicate, 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 Strategy: Many operating systems employ virtual memory techniques—e.g., virtual copying—that cause physical page frames to be mapped into different virtual address spaces. With a physically addressed cache, cached portions of such pages remain cached even after a processor reallocation

ries 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 28 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 [5] 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 cannot be 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

data manipulation steps are performed serially, resulting in poor locality of reference and thus increased memory traffic; data is often copied from one main memory location to another when crossing the user/kernel protection boundary; and additional copies may occur when the data passes through a user-level server, such as a window manager. We show that the CPU/memory bandwidth will remain within an order of magnitude of network bandwidth. Therefore, the CPU/memory data path will form a bottleneck in this type of system.

This paper analyzes the I/O data path from an end-to-end perspective, focusing on data dependent costs. (To support CM applications, appropriate scheduling and IPC mechanism such as the ones described in [8] are required in addition to the techniques presented in this paper.) Unlike previous work on this topic, which concentrates on alternative network interfaces and OS buffering techniques, we consider the entire data path, highlighting the subtle interactions that occur when various techniques and mechanisms are used in combination. Moreover, our analysis considers the trend towards kernelized operating systems and third-party user-level servers. We take the position that, in order to enable software innovations, applications should have access to CM data, which precludes the use of low-level streaming. The paper concludes with the introduction of a new abstract type for data buffers, called *IOData*, that serves as a flexible basis for an integrated I/O data path.

2 Performance of Workstation Hardware

This section analyzes several performance characteristics of desktop workstation hardware that affect CM 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 SPEC performance rating (integer and floating point), 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 ratio of measured bandwidth to the peak memory bandwidth is shown in parentheses.

	SPECInt89	SPECfp89	Memory	CPU/Memory (Mbps)		
			(Mbps)	Copy	Read	Write
IBM RS/6000 340	28.8	88.8	2133	405 (.19)	650 (.30)	590 (.28)
HP 9000/720	39.5	78.5	1600	160 (.10)	450 (.28)	315 (.20)
Dec 5000/200	19.5	26.7	800	100 (.13)	300 (.38)	570 (.71)
Sun SPARC 2	21.7	27.4	640	90 (.14)	180 (.28)	140 (.22)

Table 1. Performance of some current Workstations

The CPU/Memory bandwidth numbers reported in Table 1 were obtained using a simple benchmark that measures the sustained bandwidth during a se-

Analysis of I/O Subsystem Design for Multimedia Workstations

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

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Abstract. This paper argues that the CPU/memory data path is a potential throughput bottleneck in multimedia workstations, and analyzes the implications for the design of the I/O data path. It identifies known techniques for optimizing the data path, and highlights subtle interactions that occur when these techniques are used in combination. The paper concludes by proposing an abstract type for data buffers, called *IOData*, and argues that this abstraction provides a flexible basis for the end-to-end optimization of the I/O data path.

1 Introduction

With the emergence of fiber optics technology, local area networks have recently seen an order-of-magnitude increase in bandwidth. Another order-of-magnitude leap—to 1Gbps—is likely to occur in the near future. Suitable network interfaces can deliver this increased bandwidth to desktop workstations. At the same time, dramatic increases in microprocessor performance continue. These technological advances make it feasible to support powerful multimedia applications, including continuous media (CM) such as audio and video.

The substantial increases in processor performance and network bandwidth contrast with only moderate improvements in main memory and bus performance of workstations. 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 machines. In particular, 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, care must be taken to minimize the number of trips that CM data takes across the CPU/memory data path.

To achieve a well-balanced system that minimizes bus traffic, an end-to-end design that integrates the entire data path—from source device, through the operating system and one or more application programs, to the sink device—is required. Existing I/O systems typically fail to do this. For example, software

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