High-performance I/O and Networking Software in Sequoia 2000

Joseph Pasquale
Eric W. Anderson
Kevin Fall
Jonathan S. Kay

The Sequoia 2000 project requires a high-speed network and I/O software for the support of global change research. In addition, Sequoia distributed applications require the efficient movement of very large objects, from tens to hundreds of megabytes in size. The network architecture incorporates new designs and implementations of operating system I/O software. New methods provide significant performance improvements for transfers among devices and processes and between the two. These techniques reduce or eliminate costly memory accesses, avoid unnecessary processing, and bypass system overheads to improve throughput and reduce latency.

In the Sequoia 2000 project, we addressed the problem of designing a distributed computer system that can efficiently retrieve, store, and transfer the very large data objects contained in earth science applications. By very large, we mean data objects in excess of tens or even hundreds of megabytes (MB). Earth science research has massive computational requirements, in large part due to the large data objects often found in its applications. There are many examples: an advanced very high-resolution radiometer (AVHRR) image cube requires 300 MB, an advanced visible and infrared imaging spectrometer (AVIRIS) image requires 140 MB, and the common land satellite (LANDSAT) image requires 278 MB. Any throughput bottleneck in a distributed computer system becomes greatly magnified when dealing with such large objects. In addition, Sequoia 2000 was an experiment in distributed collaboration; thus, collaboration tools such as videoconferencing were also important applications to support.

Our efforts in the project focused on operating system I/O and the network. We designed the Sequoia 2000 wide area network (WAN) test bed, and we explored new designs in operating system I/O and network software. The contributions of this paper are twofold: (1) it surveys the main results of this work and puts them in perspective by relating them to the general data transfer problem, and (2) it presents a new design for container shipping. (For a complete discussion of container shipping, see Reference 1.) Since container shipping is a new design, this paper devotes more space to it in relation to the other surveyed work (whose detailed descriptions may be found in References 2 to 9). In addition to this work, we conducted other network studies as part of the Sequoia 2000 project. These include research on protocols to provide performance guarantees and multicasting.[10–17]

To support a high-performance distributed computing environment in which applications can effectively manipulate large data objects, we were concerned with achieving high throughput during the transfer of these objects. The processes or devices representing the data sources and sinks may all reside on the same workstation (single node case), or they may be distributed over many workstations connected by the network (multiple node case). In either case, we wanted applications, be they earth science distributed computations or collaboration tools involving multipoint video, to make full use of the raw bandwidth provided by the underlying communication system.

In the multiple node case, the raw bandwidth is from 45 to 100 megabits per second (Mb/s), because the Sequoia 2000 network used T3 links for long-distance communication and a fiber distributed data interface (FDDI) for local area communication. In the single node case, the raw bandwidth is approximately 100 megabytes per second, since the workstation of choice was one of the DECstation 5000 series or the Alpha-powered DEC 3000 series, both of which use the TURBOchannel as the system bus.

Our work focused only on software improvements, in particular how to achieve maximum system software performance given the hardware we selected. In fact, we found that the throughput bottlenecks in the Sequoia distributed computing environment were indeed in the workstation's operating system software, and not in the underlying communication system hardware (e.g., network links or the system bus). This problem is not limited to the Sequoia environment: given modern high-speed workstations (100+ millions of instructions per second [mips]) and fast networks (100+ Mb/s), performance bottlenecks are often caused by software, especially operating system software. System software throughput has not kept up with the throughputs of I/O devices, especially network adapters, which have improved tremendously in recent years. These technology improvements are being driven by a new generation of applications, such as interactive multimedia involving digital video and high-resolution graphics, that have high I/O throughput requirements. Supporting these applications and controlling these devices have taxed operating system technology, much of which was designed during times when intensive I/O was not an issue.

In the next section of this paper, we describe the Sequoia 2000 network, which served as an experimental test bed for our work. Following that, we analyze the data transfer problem, which serves as the context for the three subsequent sections. There we describe our solutions to the data transfer problem. Finally, we present our conclusions.

The Sequoia 2000 Network Test Bed

The Sequoia 2000 network is a private WAN that we designed to span five campuses at the University of California: Berkeley, Davis, Los Angeles, San Diego, and Santa Barbara. The topology is shown in Figure 1. The backbone link speeds are 45 Mb/s (T3) with the exception of the Berkeley-Davis link, which is 1.5 Mb/s (T1). At each campus, one or more FDDI local area networks (LANs) that operate at 100 Mb/s are used for local distribution. At some campuses, the configuration is a hierarchical set of rings. For example, at UC San Diego, one FDDI ring covered the campus and joined three separate rings: one at the Computer Systems Lab (our laboratory) in the Department of Computer Science and Engineering, one at the Scripps Institution of Oceanography, and one at the San Diego Supercomputer Center.

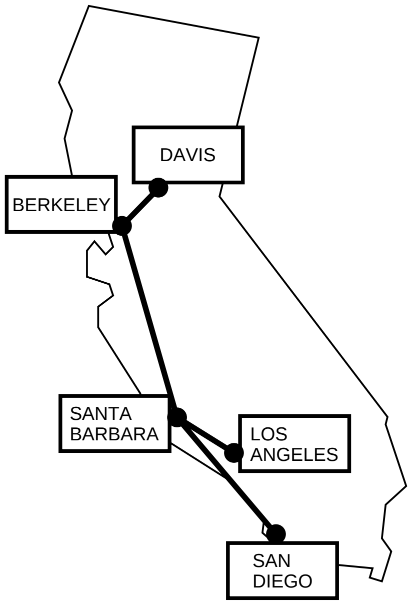

Figure 1
Sequoia 2000 Research Network

We used high-performance general-purpose computers as routers, originally DECstation 5000 series and later DEC 3000 series (Alpha-powered) workstations. Using workstations as routers running the ULTRIX or the DEC OSF/1 (now Digital UNIX) operating system provided us with a modifiable software platform for experimentation. The T3 (and T1) interface boards were specially built by David Boggs at Digital. We used off-the-shelf Digital products for FDDI boards, both models DEFTA, which supports both send and receive direct memory access (DMA), and DEFZA, which supports only receive DMA.

The Data Transfer Problem

Since a data source or sink may be either a process or device, and the operating system generally performs the function of transferring data between processes and devices, understanding the bottlenecks in these operating system data paths is key to improving performance. These data paths generally involve traversing numerous layers of operating system software. In the case of network transfers, the data paths are extended by layers of network protocol software.

To understand the performance problem we were trying to solve, consider a common client-server interaction in which a client has requested data from a server. The data resides on some source device, e.g., a disk, and must be read by the server so that it may send the data to the client over a network. At the client, the data is written to some sink device, e.g., a frame buffer for display.

Figure 2 shows a typical end-to-end data path where the source and sink end-point workstations are running protected operating system kernels such as UNIX. The source device generates data into the memory of its connected workstation. This memory is generally only addressable by the kernel; to allow the server process to access the data, it is physically copied into memory addressable via the server process's address space, i.e., user space. Physically copying data from one memory location to another (or more generally, touching the data for any reason) is a major bottleneck in modern workstations.

In travelling through the kernel, the data generally travels over a device layer and an abstraction layer. The device layer is part of the kernel's I/O subsystem and manages the I/O devices by buffering data between the device and the kernel. The abstraction layer comprises other kernel subsystems that support abstractions of devices, providing more convenient services for user-level processes. Examples of kernel abstraction layer software include file systems and communication protocol stacks: a file system converts disk blocks into files, and a communication protocol stack converts network packets into datagrams or stream segments. Sometimes, a kernel implementation may cause physical copying of data between the device layer and the abstraction layer; in fact, copying may even occur within these layers.

From kernel space, the data may travel across several more layers in user space, such as the standard I/O layer and the application layer. The standard I/O layer buffers I/O data in large chunks to minimize the number of I/O system calls. The application layer generally has its own buffers where I/O data is copied.

From the server process in user space, the data is then given to the network adapter; this may cause transfers across user process layers and then across the kernel layers. The data is then transferred over the network, which generally consists of a set of links connected by routers. If the routers have kernels whose software structure is like that described above, a similar (but typically simpler) intramachine data transfer path will apply.

Finally, the data arrives at the client's workstation. There, the data travels in a similar way as was described for the server's workstation: from the network adapter, across the kernel, through the client process's address space, and across the kernel again, finally reaching the sink device.

From this analysis, one can surmise why throughput bottlenecks often occur at the end points of the end-to-end data transfer path, assuming sufficiently fast hardware devices and communication links. At the end points, there may be significant data copying as the data traverses the various software layers, and there is protection-domain crossing (kernel to user to kernel), among other functions. The overheads caused by these functions, directly and indirectly, can be significant.

Consequently, we focused on improving operating system I/O and network software, including optimizations for the four possible process/device data transfer scenarios: process to process, process to device, device to process, and device to device, with special care in addressing cases where either source or sink

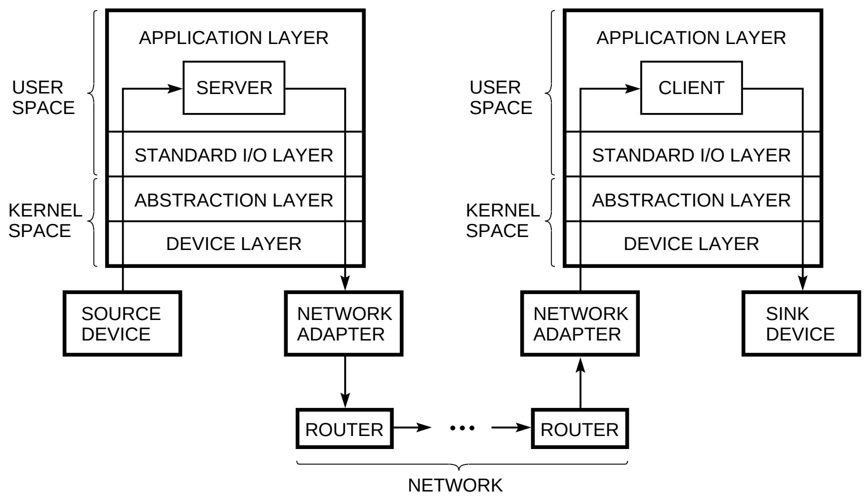

Figure 2
An End-to-End Data Path from a Source Device on One Workstation to a Sink Device on Another Workstation

device is a network adapter. In this paper, we use the term *data transfer problem* to refer to the problem of reducing these overheads to achieve high throughput between a source device and a sink device, either of which can be a network adapter within a single workstation.

Although the data transfer problem may also exist in intermediate routers, it does so to a much lesser degree than with end-user workstations (assuming modern router software and hardware technology). This is because of a router's simplified execution environment and its reduced needs for transfers across multiple protected domains. However, there is nothing that precludes the application of the techniques discussed in this paper to router software. In fact, since we used general-purpose workstations for routers to support a flexible, modifiable test bed for experimentation with new protocols, our work was also applied to router software.

In the next three sections, we describe various approaches to solving the data transfer problem. Since data copying/touching is a major software limitation in achieving high throughput, avoiding data copying/touching is a constant theme. Much of our work involves finding ways to avoid or limit touching the data without sacrificing the flexibility or protection commonly provided by most modern operating systems.

We describe two solutions to the data transfer problem that avoid all physical copying and are based on the principle of providing separate mechanisms for I/O control and data transfer.[18–21] The reader will see that while these two solutions are based on different approaches (indeed, they can even be viewed as competing), they fill different niches based on differing assumptions of how I/O is structured. In other words, each is appropriate and optimal for different situations. In addition to the data transfer problem, we address a special problem—the bottleneck created by the checksum computation for I/O on a network using the transmission control protocol/internet protocol (TCP/IP).

Container Shipping

Container shipping is a kernel service that provides I/O operations for user processes. High performance is obtained by eliminating the in-memory data copies traditionally associated with I/O. Additional gains are achieved by permitting the selective accessing (mapping) of data. Finally, the design we present makes possible specific optimizations that further improve performance.

The goals of the container shipping model of data transfer for I/O are to provide high performance without sacrificing protection and to fully support the principle of general-purpose computing. Full access to I/O data by user-level processes has long been a standard feature of operating systems. This ability has traditionally been provided by copying data to and from process memory at each instance when data is transferred. The divergence of CPU and dynamic random access memory (DRAM) speeds makes this in-memory copying more inefficient and costly every year. This problem is often attacked with application-specific silicon or kernel modifications. A less-costly and longer-lasting solution is to redesign the I/O subsystem to provide copy-free I/O. Container shipping provides this ability, as well as additional performance gains, in a uniform, general, and practical way.

Containers

A container is one or more pages of memory. In these pages, it may contain a single block of data, whose location is identified by an offset and a length. When a container is mapped into an address space, the pages form a contiguous region of memory, where the data can be manipulated. A container can be owned by one and only one domain, e.g., some user process or the kernel itself, at any single point in time. The owning domain may map the container for access. When access is not required, mapping can be avoided, which saves time.

User-level processes use container shipping system calls to perform the following functions:

- Allocation: cs_alloc and cs_free allocate and deallocate containers and their resources (e.g., physical pages).
- Transfer: cs_read and cs_write perform I/O using containers.
- Mapping: cs_map and cs_unmap allow a process to access the data in a container.

The cs_read and cs_write calls take as parameters an I/O path identifier (such as a UNIX file descriptor), a data size, and parameters describing a list of containers, or a return area for such a list. Several options are also available, such as one for cs_read that immediately maps all the resulting containers. Data is never copied within memory to satisfy cs_read and cs_write, so all I/O performed this way is copy-free.

Because the mapping of containers is always optional, a process can move data from one device to another without mapping it at all. When containers of data flow through a pipeline of several processes, substantial additional savings can be obtained if several of the processes do not map the containers, or if they map only some of the containers.

Although container shipping has six different system calls versus the two of conventional I/O, read and write, the actual number of calls a process issues with container I/O may be no greater than with conventional I/O. When data is not mapped, only cs_read and cs_write calls are required. Even if data is mapped, it may be possible to perform the mapping through flags to cs_read, without calling cs_map. Unmapping is automatic in cs_write, so if cs_unmap is not used, two system calls are still sufficient.

As shown in Figure 3, a process reads data in a container from one device and writes it to another device. Three pages of memory form one container that stores two and one-half pages of data. On input (cs_read), the source device deposits data into physical memory pages forming the container. The process that owns the container may then map (cs_map) it so that the data can be manipulated in its address space. The data is then output (cs_write) to the sink device. Output can occur without having mapped the container. Mapping can also occur automatically on cs_read.

Eliminating In-Memory Copying

Unconditionally avoiding the copying of data within memory during I/O leads to the first of several performance gains from container shipping. Other solutions exist that avoid copies only in limited cases. To be uniform and general, copy-free I/O must be possible without restrictions due to the devices used, the order of operations, or the availability of special device hardware.

In many I/O operations, the data requested by a user-level process is already in system memory when the request is made. This situation can arise when data is moving between two processes via the I/O system, such as is done with pipes. Many optimized file systems perform read-ahead and in-memory caching to improve performance, so file I/O requests may also be satisfied with data that is already in memory. Finally, conventional network adapters transfer entire packets into memory before they are examined by protocol layers in the kernel. Only after protocol processing can this data be delivered to the correct user-level process. When requested data is already in memory, the only possible copy-free transfer mechanism that allows full read/write access in the address space of a process is virtual memory remapping. Techniques that rely on device-specific characteristics such as programmable DMA or outboard protocol processors cannot provide uniform, device-independent copy-free I/O, because these mechanisms cannot transfer data that is already in memory.

Using virtual memory remapping, container shipping can perform copy-free I/O regardless of when or where data arrives in memory, and with or without any special device hardware that might be available. Virtual memory hardware is employed to control the ownership of, and access to, memory that contains I/O data. Ownership and access rights are transferred between domains when container I/O is performed, while data sits motionless in memory. This technique requires no special assistance from devices and applies to interprocess communication as well as all physical I/O. Because user-level processes retain complete access to I/O data with no in-memory copying, user-level programming remains a practical solution for high-performance systems.

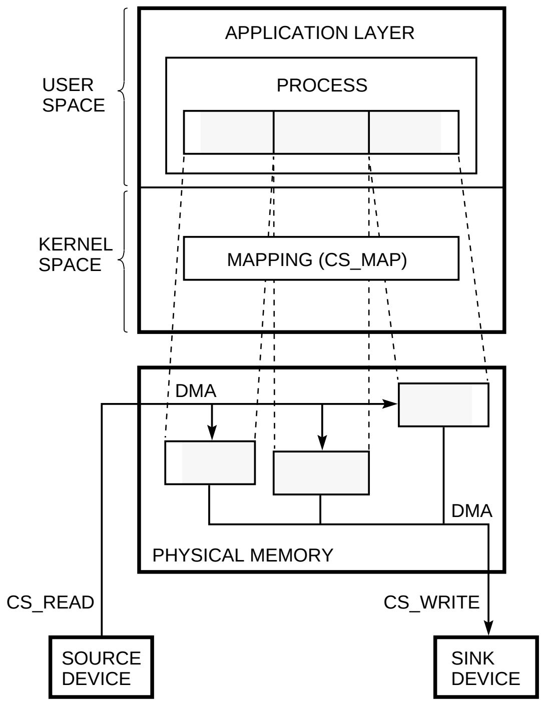

Figure 3
Container Shipping Transfer and Mapping

The Gain/Lose Model

In container I/O, reading and writing are coupled with the gain and loss of memory. We chose the gain/lose model because it is simple and provides higher performance without sacrificing protection. Shared memory is a more complicated alternative to the gain/lose model, which also avoids data copying. The use of shared memory to allow a set of processes to efficiently communicate, however, reduces the protection between domains. Shared-memory I/O schemes also tend to be complicated because of the close coordination required between a user process and the kernel when they both manipulate a shared data pool. Since data is never shared under the gain/lose model, protection domains need not be compromised, and less user/kernel cooperation and trust is required.

The gain/lose model has three major implications for programmers. First, a process must dispose ofI/O data that it gains, or memory consumption may grow rapidly. One way to dispose of data is to perform a cs_write operation on it, so a process performing matched reads and writes on a stream of data will not accumulate any extra memory. Second, to avoid seriously complicating conventional memory models, not all memory is eligible for use in write operations. For example, writing data from the stack would leave an inconvenient hole in that part of the virtual memory, so this is not allowed. Finally, because data that is communication (IPC) within the ULTRIX version 4.2a operating system on a DECstation 5000/200 system.[1] With the new DEC OSF/1 implementation on Alpha workstations, we compared the I/O performance of conventional UNIX I/O to that of container shipping for a variety of I/O devices as well as IPC. These experiments are described in detail elsewhere.[23] Large improvements in throughput were observed, from 40 percent for FDDI network I/O (despite large non–data-touching protocol and device-driver overheads) to 700 percent for socket-based IPC.

We devised an experiment that exercises both the IPC and I/O capabilities of container shipping. Images (640 × 480 pixels, 1 byte per pixel) are sent by one process and received by a second process using socket IPC. The receiver process then does output to a frame buffer to display the images on the screen. This is a common application in the Sequoia project: viewing an animation composed of images displayed at a rate of up to 30 frames per second (fps). In fact, scientists often want to view as many simultaneously displayed animations as possible.

We carried out this experiment first using conventional UNIX I/O (i.e., read and write) and then using container shipping (i.e., cs_read and cs_write). Figure 4 shows the throughput obtained for a sender process transferring data to a receiver process, which then outputs the data to a frame buffer. The improvement of container shipping over UNIX I/O is almost 400 percent. Assuming the maximum 30 fps rate, conventional I/O supports the full display of one animation and container I/O supports six. In general, the greater the relative speed between an I/O device and memory, the greater the relative throughput of container shipping versus UNIX I/O will be.

Related Work

The use of virtual transfer techniques to avoid the performance penalty of physical copying goes back to TENEX.[24] Mach (like TENEX) uses virtual copying, i.e., transferring a data object by mapping it in the new address space, and then physically copying if the data is modified (copy-on-write).[22] This differs from container shipping, which uses virtual moving; i.e., the data object leaves the source domain and appears in the destination domain, where it can be read and written without causing fault handling, which is expensive. If the original domain wants to keep a copy, it may do so explicitly. Thus, container shipping places a greater burden on the programmer in return for improved performance.

The two systems that are most similar to container shipping are DASH and Fbufs.[25,26] Containers are similar to the IPC pages used in DASH and the fast buffers used by Fbufs. DASH provides high-performance interprocess communication: it achieves fast, local IPC by means of page remapping, which allows processes to own regions of a restricted area of a shared address space. The Fbufs system uses a similar technique, enhanced by caching the previous owners of a buffer, allowing reuse among trusted processes and eliminating memory management unit (MMU) updates associated with changing buffer ownership. The differences between these two systems and container shipping are examined in detail elsewhere.[23]

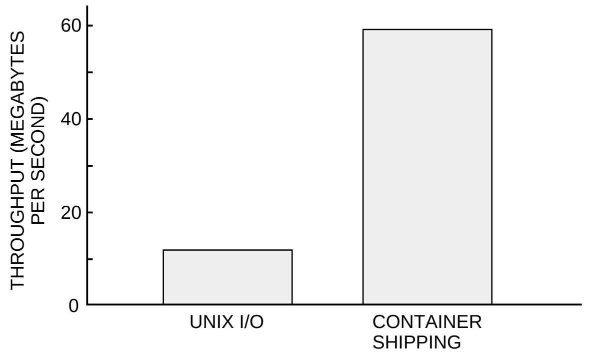

Figure 4
Throughput of IPC and Frame Buffer Output

Peer-to-Peer I/O

In addition to container shipping, we have investigated an alternative I/O system software model called peer-to-peer I/O (PPIO). As a direct result of the structure of this model, its implementation avoids the well-known overheads associated with data copying. Unlike other solutions, PPIO also reduces the number of context-switch operations required to perform I/O operations. In contrast to container shipping, PPIO is based on a streaming approach, where data is permitted to flow between a producer and consumer (these may be devices, files, etc.) without passing through a controlling process' address space. In PPIO, processes use the splice system call to create kernel-maintained associations between producer and consumer. Splice represents an addition to the conventional operating system I/O interfaces and is not a replacement for the read and write system functions.

The Splice Mechanism

The splice mechanism is a system function used to establish a kernel-managed data path directly between I/O device peers.[2,3] It is the primary mechanism that processes invoke to use PPIO. With splice, an application expresses an association between a data source and sink directly to the operating system through the use of file descriptors. These descriptors do not refer to memory addresses (i.e., they are not buffers):

```
sd = splice (fd1, fd2);
```

As shown in Figure 5, the call establishes an in-kernel data path, i.e., a splice, between a data source and sink

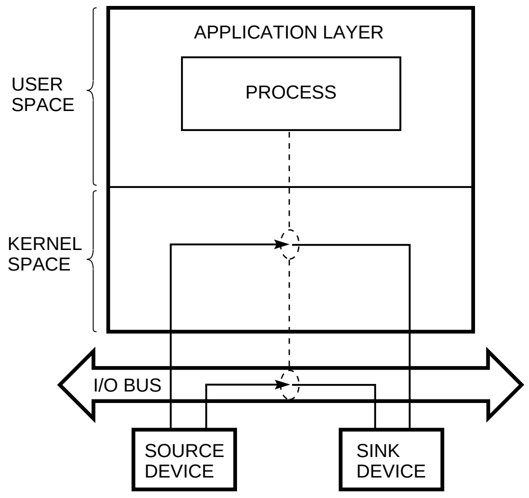

Figure 5
A Splice Connecting a Source to a Sink Device

device. If the I/O bus and the devices support hardware streaming, the data path is directly over the bus, avoiding system memory altogether. Although the process does not necessarily manipulate the data, it controls the size and timing of the dataflow. To manipulate the data, a processing module can be downloaded either into the kernel or directly on the devices if they support processing.

The data source and sink device are specified by the references fd1 and fd2, respectively. The splice descriptor sd is used in subsequent calls to read or write to control the flow of data across the splice. For example, the following call causes size bytes of data to flow from the source to the sink:

```
splice_ctrl_msg sc;
sc.op = SPLICE_OP_STARTFLOW;
sc.increment = size;
write (sd, &sc, sizeof(sc));
```

Data produced by the devices referenced by fd1 is automatically routed to fd2 without user process intervention, until size bytes have been produced at the source. The increment field specifies the number of bytes to transfer across a splice before returning control to the calling user application. When control is returned, dataflow is stopped. A SPLICE_OP_STARTFLOW must be executed to restart dataflow. The increment represents an important concept in PPIO and refers to the amount of data the user process is willing to have transferred by the operating system on its behalf. In effect, it specifies the level of delegation the user process is willing to give to the system. Specifying SPLICE_INCREMENT_DEFAULT indicates the system should choose an appropriate increment. This is generally a buffer size deemed convenient by the operating system.

The splice mechanism eliminates copy operations to user space by not relying on buffer interfaces such as those present in the conventional I/O functions read and write. By eliminating the user-level buffering, kernel buffer sharing is possible. More specifically, when block alignment is not required by an I/O device, a kernel-level buffer used for data input may be used subsequently for data output.

In addition to removing the buffering interfaces, splice also combines the read/write functionality together in one call. The splice call indicates to the operating system the source and sink of a dataflow, providing sufficient information for the kernel to manage the data transfer by itself without requiring user-process execution. Thus, context-switch operations for data transfer are eliminated. This is important: context switches consume CPU resources, degrade cache performance by reducing locality of reference, and affect the performance of virtual memory by requiring TLB invalidations.[27,28]

For applications making no direct manipulation of I/O data (or for those allowing the kernel to make such manipulations), splice relegates the issues of managing the dataflow (e.g., buffering and flow control) to the kernel. Data movement may be accomplished by a kernel-level thread, possibly activated by completion events (e.g., device interrupt) or operating in a more synchronous fashion. Flow control may be achieved by selective scheduling of kernel threads or simply by posting reads only to data-producing devices when data-consuming peers complete I/O operations. A kernel-level implementation provides much flexibility in choosing which control abstraction is most appropriate.

One criticism of streaming-based data transfer mechanisms is that they inhibit innovation in application development by disallowing applications direct access to I/O data.[29] However, many applications that do not require direct manipulation of I/O data can benefit from streaming (e.g., data-retrieving servers that do not need to inspect the data they have been requested to deliver to a client). Furthermore, for applications requiring well-known data manipulations, kernel-resident processing modules (e.g., Ritchie's Streams) or outboard dedicated processors are more easily exploited within the kernel operating environment than in user processes.[30,31] In fact, PPIO supports processing modules.[4]

PPIO Implementation and Performance

The PPIO design was conceived to support large data transfers. The decoupling of I/O data from process address space reduces cache interference and eliminates most data copies and process manipulation. PPIO and the accompanying splice system call have been implemented within the ULTRIX version 4.2a operating system for the DEC 5000 series workstations, and within DEC OSF/1 version 2.0 for DEC 3000 series (Alpha-powered) workstations, each for a limited number of devices.

Three performance evaluation studies of PPIO have been carried out and are described in our early papers.[2,3,4] They indicate CPU availability improves by 30 percent or more; and throughput and latency improve by a factor of two to three, depending on the speed of I/O devices. Generally, the latency and throughput performance improvements offered by PPIO improve with faster I/O devices, indicating that PPIO scales well with new I/O device technology.

Improving Network Software Throughput

Network I/O presents a special problem in that the complexity of the abstraction layer (see Figure 2), a stack of network protocols, is generally much greater than that for other types of I/O. In this section, we summarize the results of an analysis of overheads for an implementation of TCP/IP we used in the Sequoia 2000 project. The primary bottleneck in achieving high throughput communication for TCP/IP is due to data-touching operations: one expected culprit is data copying (from kernel to user space, and vice versa); another is the checksum computation. Since we have already focused on how to avoid data copying in the previous two sections, we discuss how one can safely avoid computing checksums for a common case in network communication.

Overhead Analysis

We undertook a study to determine what bottlenecks might exist in TCP/IP implementations to direct us in our goal of optimizing throughput. The full study is described elsewhere.[9]

First, we categorized various generic functions commonly executed by TCP/IP (and UDP/IP) protocol stacks:

- Checksum: the checksum computation for UDP (user datagram protocol) and TCP
- DataMove: any operations that involve moving data from one memory location to another
- Mbuf: the message-buffering scheme used by Berkeley UNIX-based network subsystems
- ProtSpec: all protocol-specific operations, such as setting header fields and maintaining protocol state
- DataStruct: the manipulation of various data structures other than mbufs or those accounted for in the ProtSpec category
- OpSys: operating system overhead
- ErrorChk: The category of checks for user and system errors, such as parameter checking on socket system calls
- Other: This final category of overhead includes all the operations that are too small to measure. Its time was computed by taking the difference between the total processing time and the sum of the times of all the other categories listed above.

Other studies have shown some of these overheads to be expensive.[32–34]

We measured the total amount of execution time spent in the TCP/IP and UDP/IP protocol stacks as implemented in the DEC ULTRIX version 4.2a kernel, to send and receive IP packets of a wide range of sizes, broken down according to the categories listed above. All measurements were taken using a logic analyzer attached to a DECstation 5000/200 workstation connected to another similar workstation by an FDDI LAN attached through a Digital DEFZA FDDI adapter.

Figure 6 shows the per-packet processing times versus packet size for the various overheads for UDP packets. These are for a large range of packet sizes, from 1 to 8,192 bytes. One can distinguish two different types of overheads: those due to data-touching operations (i.e., data move and checksum) and those due to non–data-touching operations (all other categories). Data-touching overheads dominate the processing time for large packets that typically contain application data, whereas non–data-touching operations dominate the processing time for small packets that typically contain control information. Generally, data-touching overhead times scale linearly with packet size, whereas non–data-touching overhead times are comparatively constant. Thus, data-touching overheads present the major limitations to achieving maximum throughput.

Data-touching operations, which do identical work in the TCP and UDP software, also dominate processing times for large TCP packets.[9]

Minimizing the Checksum Overhead

As can be seen in Figure 6, the largest bottleneck to achieving maximum throughput (i.e., which one achieves by sending large packets) is the checksum computation. We applied two optimizations to minimize this overhead: improving the implementation of the checksum computation, and avoiding the checksum altogether in a special but common case where we felt we were not compromising data integrity.

We improved the checksum computation implementation by applying some fairly standard techniques: operating on 32-bit rather than 16-bit words, loop unrolling, and reordering of instructions to maximize pipelining. With these modifications, we

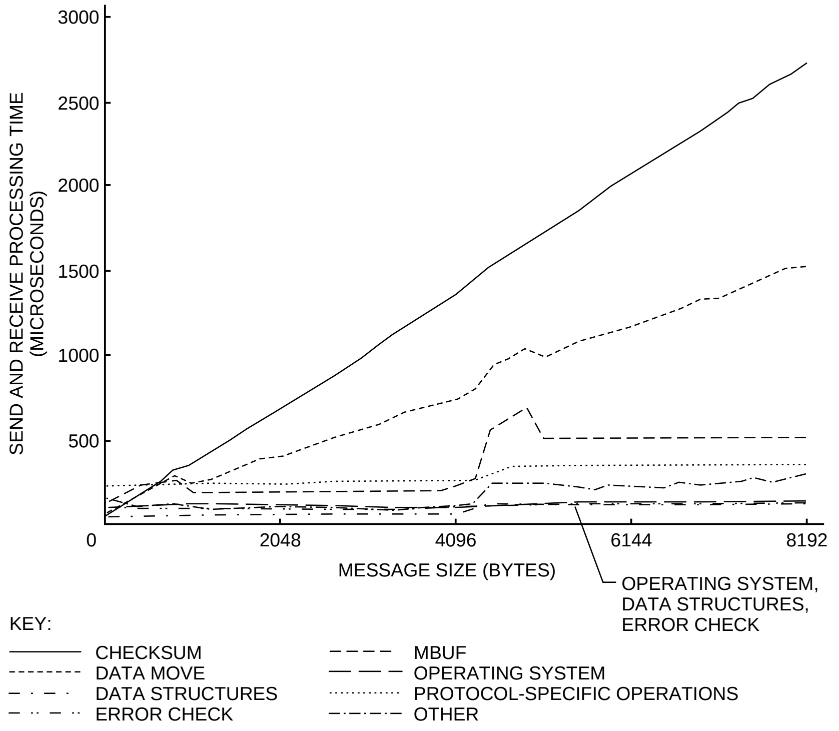

Figure 6
UDP Processing Overhead Times

reduced the checksum computation time by more than a factor of two. Figure 7 shows that the overall throughput improvement is 37 percent. The throughput measurements were made between two DECstation 5000/200 systems communicating over an FDDI network. Overall throughput is still a fraction of the maximum FDDI network bandwidth (100 Mb/s) because of data-copying overheads and machine-speed limitations. See Reference 6 for detailed results.

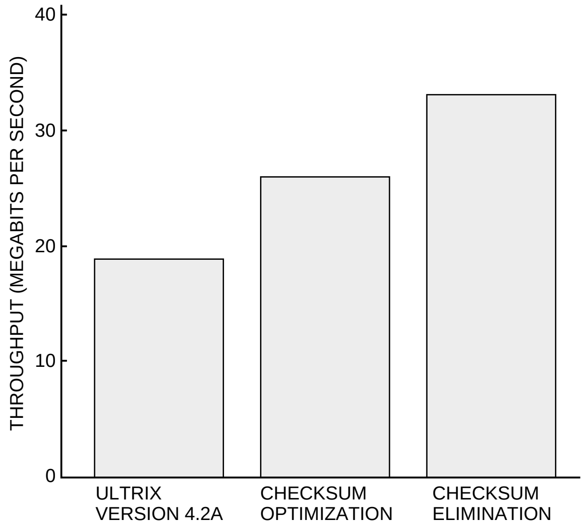

Figure 7
UDP/IP End-to-End Throughput

A very easy way of significantly raising TCP and UDP throughput is to simply avoid computing checksums; in fact, many systems provide options to do just this. The Internet checksum, however, exists for a good reason: packets are occasionally corrupted during transmission, and the checksum is needed to detect corrupted data. In fact, the Internet Engineering Task Force (IETF) recommends that systems not be shipped with checksumming disabled by default.[35]

Ethernet and FDDI networks, however, implement their own cyclic redundancy checksum (CRC). Thus, packets sent directly over an Ethernet or FDDI network are already protected from data corruption, at least at the level provided by the CRC. One can argue that for LAN communication, the Internet checksum computation does not significantly add to the machinery for error detection already provided in hardware.

Thus, our second optimization was simply to eliminate the software checksum computation altogether when computing the checksum would make little difference. Consequently, as part of the implementation of the protocol, when the source and destination are determined to be on the same LAN, the software checksum computation is avoided. Figure 7 shows the resulting 74 percent improvement in throughput over the unmodified ULTRIX version

4.2a operating system, and a 27 percent improvement over the implementation with the optimized checksum computation algorithm.

Of course, one must be very careful about deciding when the Internet checksum is of minimal value. We believe it is reasonable to turn off checksums when crossing a single network that implements its own CRC, especially when one considers the performance benefits of doing so. In addition, since the destinations of most TCP and UDP packets are within the same LAN on which they are sent, this policy eliminates the software checksum computation for most packets.

Our checksum elimination policy differs somewhat from traditional TCP/IP design in one aspect of protection against corruption. In addition to the protection between network interfaces given by the Ethernet and FDDI checksums, we require a software checksum in host memory as a protection from errors in data transfer over the I/O bus. For common devices such as disks, however, data transfers over the I/O bus are routinely assumed to be correct and are not checked in software. Therefore, a reduction in protection against I/O bus transfer errors for network devices does not seem unreasonable.

Turning off the Internet checksum protection in any wider area context seems unwise without considerable justification. Not all networks are protected by CRCs, and it is difficult to see how one might check that an entire routed path is protected by CRCs without undue complications involving IP extensions. A more fundamental problem is that network CRCs protect a packet only between network interfaces; errors may arise while a packet is in a gateway machine. Although such corruption is unlikely for a single machine, the chance of data corruption occurring increases exponentially with the number of gateways a packet crosses.

Summary and Conclusions

We described various solutions to achieving high performance in operating system I/O and network software, with a particular emphasis on throughput. Two of the solutions, container shipping and peer-to-peer I/O, focused on changes in the I/O system software structure to avoid data copying and other overheads. The third solution focused on the avoidance of additional data-touching overheads in TCP/IP network software.

Container shipping is a kernel service that provides I/O operations for user processes. High performance is obtained by eliminating the in-memory data copies traditionally associated with I/O, without sacrificing safety or relying on devices with special-purpose functionality. Further gains are achieved by permitting the selective accessing (mapping) of data. We measured performance improvements over UNIX of 40 percent (network I/O) to 700 percent (socket IPC).

PPIO is based on the hypothesis that the memory-oriented model of I/O present in most operating systems presents a bottleneck that adversely affects overall performance. PPIO decouples user-process execution from interdevice dataflow and can achieve improvements in both latency and throughput over conventional systems by a factor of 2 to 3.

Finally, we considered the special case of network I/O where data moving/copying is not the only major overhead. We showed that the checksum computation is a major source of TCP/IP network processing overhead. We improved performance by optimizing the checksum computation algorithm and eliminating the checksum computation when communicating over a single LAN that supports its own CRC, improving throughput by 37 percent to 74 percent for UDP/IP.

Acknowledgments

We are indebted to David Boggs, who built the T1 and T3 boards which worked like a charm. We appreciate the efforts of Richard Bartholomaus and Ira Machefsky who helped us get our first DECstation 5000/200 workstations. As our interface to Digital, Ira Machefsky helped in many other ways; we sincerely thank him. We thank Fred Templin for the technical expertise he provided us on Digital networking equipment. We thank Mike Stonebraker and Jeff Dozier for their leadership of the Sequoia 2000 project. We thank our network research colleagues, Professors Domenico Ferrari and George Polyzos, and their students, with whom we enjoyed collaborating; we have benefited from their advice. Finally, thanks to Jean Bonney for supporting our project from start to end.

References

1. J. Pasquale, E. Anderson, and K. Muller, "Container Shipping: Operating System Support for Intensive I/O Applications," *IEEE Computer,* vol. 27, no. 3 (1994): 84–93.

2. K. Fall and J. Pasquale, "Exploiting In-kernel Data Paths to Improve I/O Throughput and CPU Availability," *Proceedings of the USENIX Winter Technology Conference,* San Diego (January 1993), pp. 327–333.

3. K. Fall and J. Pasquale, "Improving Continuous-media Playback Performance with In-kernel Data Paths," *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS),* Boston, Mass. (June 1994), pp. 100–109.

4. K. Fall, "A Peer-to-Peer I/O System in Support of I/O Intensive Workloads," Ph.D. dissertation, University of California, San Diego, 1994.

5. J. Kay and J. Pasquale, "The Importance of Non–Data-Touching Processing Overheads in TCP/IP," *Proceedings of the ACM Communications Architectures and Protocols Conference (SIGCOMM),* San Francisco (September 1993), pp. 259–269.

6. J. Kay and J. Pasquale, "Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000," *Proceedings of the USENIX Winter Technology Conference,* San Diego (January 1993), pp. 249–258.

7. J. Kay and J. Pasquale, "A Summary of TCP/IP Networking Software Performance for the DECstation 5000," *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS),* Santa Clara, Calif. (May 1993), pp. 266–267.

8. J. Kay, "PathIDs: Reducing Latency in Network Software," Ph.D. dissertation, University of California, San Diego, 1995.

9. J. Kay and J. Pasquale, "Profiling and Reducing Processing Overheads in TCP/IP," *IEEE/ACM Transactions on Networking* (accepted for publication).

10. D. Ferrari, A. Banerjea, and H. Zhang, *Network Support for Multimedia: A Discussion of the Tenet Approach* (Berkeley, Calif.: International Computer Science Institute, Technical Report TR-92-072, 1992).

11. H. Zhang, D. Verma, and D. Ferrari, "Design and Implementation of the Real-Time Internet Protocol," *Proceedings of the IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems,* Tucson, Ariz. (February 1992).

12. A. Banerjea, E. Knightly, F. Templin, and H. Zhang, "Experiments with the Tenet Real-time Protocol Suite on the Sequoia 2000 Wide Area Network," *Proceedings of the ACM Multimedia,* San Francisco (October 1994).

13. V. Kompella, J. Pasquale, and G. Polyzos, "Multicast Routing for Multimedia Applications," *IEEE/ACM Transactions on Networking,* vol. 1, no. 3 (1993): 286–292.

14. V. Kompella, J. Pasquale, and G. Polyzos, "Two Distributed Algorithms for Multicasting Multimedia Information," *Proceedings of the Second International Conference on Computer Communications and Networks (ICCCN),* San Diego (June 1993), pp. 343–349.

15. J. Pasquale, G. Polyzos, E. Anderson, and V. Kompella, "Filter Propagation in Dissemination Trees: Trading Off Bandwidth and Processing in Continuous Media Networks," *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV),* D. Shepherd, G. Blair, G. Coulson, N. Davies, and F. Garcia (eds.), *Lecture Notes in Computer Science,* vol. 846 (Springer-Verlag, forthcoming).

16. J. Pasquale, G. Polyzos, E. Anderson, and V. Kompella, "The Multimedia Multicast Channel," *Journal of Internetworking: Research and Experience* (in press).

17. J. Pasquale, G. Polyzos, and V. Kompella, "Real-time Dissemination of Continuous Media in Packet-switched Networks," *Proceedings of the 38th IEEE Computer Society International Conference (COMPCON),* San Francisco (February 93), pp. 47–48.

18. K. Muller and J. Pasquale, "A High-Performance Multi-Structured File System Design," *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP),* Asilomar, Calif. (October 1991), pp. 56–67.

19. J. Pasquale, "I/O System Design for Intensive Multimedia I/O," *Proceedings of the Third IEEE Workshop Workstation Operation Systems (WWOS),* Key Biscayne, Fla. (April 1992), pp. 29–33.

20. J. Pasquale, "System Software and Hardware Support Considerations for Digital Video and Audio Computing," *Proceedings of the 26th Hawaii International Conference on System Sciences (HICSS),* Maui, IEEE Computer Society Press (January 1993), pp. 15–20.

21. C. Thekkath, H. Levy, and E. Lazowska, "Separating Data and Control Transfer in Distributed Operating Systems," *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* San Jose, Calif. (October 1994), pp. 2–11.

22. R. Rashid et al., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *IEEE Transactions on Computers,* vol. 37, no. 8 (1988): 896–908.

23. E. Anderson, "Container Shipping: A Uniform Interface for Fast, Efficient, High-bandwidth I/O," Ph.D. dissertation, University of California, San Diego, 1995.

24. D. Bobrow, J. Burchfiel, D. Murphy, and R. Tomlinson, "TENEX, a Paged Time-Sharing System for the PDP-10," *Communications of the ACM,* vol. 15, no. 3 (1972): 135–143.

25. S.-Y. Tzou and D. Anderson, "The Performance of Message-Passing Using Restricted Virtual Memory Remapping," *Software—Practice and Experience,* vol. 21, no. 3 (1991): 251–267.

26. P. Druschel and L. Peterson, "Fbufs: a High-bandwidth Cross-Domain Transfer Facility," *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP),* Asheville, N.C. (December 1993), pp. 189–202.

27. J. Mogul and A. Borg, "The Effect of Context Switches on Cache Performance," *Proceedings of the ASPLOS-IV* (April 1991), pp. 75–84.

28. B. Bershad, T. Anderson, E. Lazowska, and H. Levy, "Lightweight Remote Procedure Call," *ACM Transactions on Computer Systems,* vol. 8, no. 1 (1990): 37–55.

29. P. Druschel, M. Abbott, M. Pagels, and L. Peterson, "Analysis of I/O Subsystem Design for Multimedia Workstations," *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV),* November 1992.

30. D. Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal,* vol. 63, no. 8 (1984): 1897–1910.

31. D. Presotto and D. Ritchie, "Interprocess Communication in the Eighth Edition UNIX System," *Proceedings of the USENIX Winter Conference* (January 1985), pp. 309–316.

32. L.-F. Cabrera, E. Hunter, M. Karels, and D. Mosher, "User-Process Communication Performance in Networks of Computers," *IEEE Transactions on Software Engineering,* vol. 14, no. 1 (1988): 38–53.

33. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead," *IEEE Communications* (1989): 23–29.

34. R. Watson and S. Mamrak, "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices," *ACM Transactions on Computer Systems,* vol. 5, no. 2 (1987): 97–120.

35. R. Braden, "Requirements for Internet Hosts—Communication Layers," *Internet Request for Comments 1122,* (Network Information Center, 1989).

Biographies

Joseph Pasquale
Joseph Pasquale is an associate professor in the Department of Computer Science and Engineering at the University of California at San Diego. He has a B.S. and an M.S. from the Massachusetts Institute of Technology and a Ph.D. from the University of California at Berkeley, all in computer science. In 1989, he established the UCSD Computer Systems Laboratory, where he and his students do research in network and operating system software design, especially to support I/O-intensive applications such as distributed multimedia (digital video and audio) and scientific computing. He also investigates issues of coordination and decentralized control in large distributed systems. He has published more than 40 refereed conference and journal articles in these areas and received the NSF Presidential Young Investigator Award in 1989.

Eric W. Anderson
Eric Anderson received B.A. (1989), M.S. (1991), and Ph.D. (1995) degrees from the University of California at San Diego. His dissertation is on the development of a uniform interface for fast, efficient, high-bandwidth I/O. As a research assistant at UCSD, he contributed to the planning and installation of the Sequoia 2000 network and conducted research in operating system I/O and high-speed networking. He is currently a postgraduate researcher with the Computer Systems Laboratory at UCSD, where he is involved in further studies of high-performance I/O techniques. He is a member of ACM and has coauthored papers on the multimedia multicast channel, operating system support for I/O-intensive applications, and filter propagation in dissemination trees in continuous media networks.

Kevin R. Fall
Kevin Fall received a Ph.D. in computer science from the University of California at San Diego in 1994 and a B.A. in computer science from the University of California at Berkeley in 1988. He held concurrent postdoctoral positions with UCSD and MIT before joining the Lawrence Berkeley National Laboratory in September 1995, where he is a staff computer scientist in the Network Research Group. While at UC Berkeley, he was responsible for the integration of security software into Berkeley UNIX and protocol development for the campus' supercomputer. While at UC San Diego, Kevin developed a high-performance I/O architecture designed to the support the large I/O demands of the Sequoia 2000 database and multimedia applications. He was also responsible for the routing architecture and system configuration of the Sequoia 2000 network.

Jonathan S. Kay
Jon Kay received a Ph.D. in computer science from the University of California at San Diego. While working toward his doctorate, he was involved in the Sequoia 2000 networking project. He joined Isis Distributed Systems of Ithaca, N.Y. in 1994 to work on distributed computing toolkits. He also holds a B.S. and an M.S. in computer science from Johns Hopkins University.