Limits to Low-Latency Communication on High-Speed Networks

CHANDRAMOHAN A. THEKKATH and HENRY M. LEVY
University of Washington, Seattle

The throughput of local area networks is rapidly increasing. For example, the bandwidth of new ATM networks and FDDI token rings is an order of magnitude greater than that of Ethernets. Other network technologies promise a bandwidth increase of yet another order of magnitude in a few years. However, in distributed systems, lowered latency rather than increased throughput is often of primary concern. This paper examines the system-level effects of newer high-speed network technologies on low-latency, cross-machine communications.

To evaluate a number of influences, both hardware and software, we designed and implemented a new remote procedure call system targeted at providing low latency. We then ported this system to several hardware platforms (DECstation and SPARCstation) with several different networks and controllers (ATM, FDDI, and Ethernet). Comparing these systems allows us to explore the performance impact of alternative designs in the communication system with respect to achieving low latency, e.g., the network, the network controller, the host architecture and cache system, and the kernel and user-level runtime software.

Our RPC system, which achieves substantially reduced call times (170 μseconds on an ATM network using DECstation 5000/200 hosts), allows us to isolate those components of next-generation networks and controllers that still stand in the way of low-latency communication. We demonstrate that new-generation processor technology and software design can reduce small-packet RPC times to near network-imposed limits, making network and controller design more crucial than ever to achieving truly low-latency communication.

Categories and Subject Descriptors: B.4.2 [**Input / Output and Data Communications**] Input/Output Devices—*channels and controllers*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications, network operating systems*; D.4.4 [**Operating Systems**]: Communications Management—*message sending, network communication*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Design, Measurement, Performance

Additional Key Words and Phrases: ATM networks, host-network interfaces, interprocess communication, remote procedure calls, transport level protocols

This work was supported in part by the National Science Foundation under grants CCR-8619663, CCR-8703049, CCR-8907666, and CCR-9200832, by the Washington Technology Center, and by the Digital Equipment Corporation Systems Research Center and External Research Program, Hewlett-Packard Corp., and Apple Computer. C. Thekkath is supported in part by a fellowship from Intel Corp. H. Levy is supported in part by a Fullbright research award and by the INRIA.
Authors' address: University of Washington, Department of Computer Science and Engineering, Seattle, WA 98195.

1. INTRODUCTION

In modern computer systems, slow communication links and software-processing overheads have imposed a stiff penalty for cross-machine communication. These high costs limit the extent to which a network of computers can be used as a solution for either structuring-related or performance-related problems in applications.

Several recent developments, however, have the potential to change the way in which networks are used:

—New local area network technologies, such as FDDI [2], and B-ISDN [34] using Asynchronous Transfer Mode (ATM), offer a 10-fold bandwidth improvement over Ethernet [23]. Another order-of-magnitude improvement to 1 gigabit per second seems close behind.

—New processor technologies and RISC architectures have already given us an order-of-magnitude improvement in processing power, with 100 MIPS performance expected within the year. These processors will be capable of using the bandwidth that new networks provide.

—Performance-oriented operating system research has led to extremely low overhead operating system mechanisms [4, 22, 27, 33]. Such low-overhead mechanisms greatly reduce the software latency that has typically limited the performance of operating system primitives.

Taken together, these technologies should allow us to produce low-latency communication systems that can exploit novel distribution schemes that would not otherwise be possible.

As a motivating example, consider the design of a distributed-memory server [11]. In a cluster of networked workstations, paging to spare memory lying idle on other nodes could be much faster than paging to disk. A distributed server could manage the idle memory in the network; on a page fault, the page fault handler would communicate with the server and transfer pages to or from remote nodes. The effectiveness of this scheme is highly influenced by the communication latency between client and server.

As another example, reduced-latency communication would greatly facilitate the use of networked workstations as a loosely coupled multiprocessor or shared virtual-memory system [7, 21]. Such configurations would have significant cost performance, flexibility, and scalability benefits over tightly coupled systems or over dedicated message-based multiprocessors (e.g., cubes). These systems are characterized by frequent exchanges of small synchronization packets and data transfers—attributes that are well served by low communication latencies. The common thread in these examples is that latency and CPU overhead are at least as critical as network throughput. With the advent of very high-bandwidth networks, the throughput problem is relatively more tractable, leaving latency as the preeminent issue.

1.1 Paper Organization and Goals

The objectives of this paper are threefold:

(1) To evaluate the fundamental, underlying message costs for small-packet

communication on new-technology networks, when compared with Ethernet technology.

(2) To describe and evaluate the design of a new low-latency RPC system in the face of different networks, controllers, and architectures.

(3) To draw lessons for network and controller design based on our experience. We believe (as do others) that the new generation of network controllers are often poorly matched to the demands of modern distributed systems, particularly in the face of low-overhead RPC systems.

Overall, our objective is to take a *system-level* view of RPC on new-generation networks, examining in particular the software and hardware interface. To do this, we have designed, implemented, and measured a low-latency RPC on several different network and workstation combinations, specifically, on the MIPS R3000-based DECstation 5000/200 connected by Ethernet, FDDI, and ATM local area networks, and on the SparcStation I connected by an Ethernet network.

We have used Remote Procedure Call (RPC) as the remote communications model for several reasons: (1) it is a dominant paradigm for distributed applications; (2) general techniques for achieving good performance are well known [5, 27]; (3) it is user-to-user, in contrast to kernel-to-kernel memory communication that tends to underestimate communication latencies significantly; and (4) it is close in spirit to other communication models, such as V [8] or CSP [17], making our observations widely applicable. Our objective is *not* to attempt a completely new RPC design, but rather to assess the latency impact of modern networks, controllers, processors, and software. To do this required the design and implementation of a new RPC system, because existing high-performance RPC systems (such as SRC RPC [27]) do not run on the experimental hardware base we required.

The paper is organized around the objectives listed previously. Section 2 describes the network environments that we used and details the minimum cost for cross-machine communication in these environments. Section 3 examines the RPC design principles that make low-latency RPC feasible. Measurement of our RPC system on a variety of controllers indicates how the design of the controller has a significant influence on efficient cross-machine communication overheads. Complex controllers and host/controller interfaces on existing machines appear to add to the overall overhead in two ways—latency inherent in the controller and latency required by the host software to service the controller. Finally, Section 4 examines in greater detail the implications of low-latency communication for networks, controllers, and operating system abstractions. For example, we show the latency effects of cache management, interrupt handling, and data transfer techniques.

Our experience demonstrates that new-generation processor technology and performance-oriented software design can reduce RPC times for small packets to near network- and architecture-imposed limits. For example, using DECstation 5000/200 workstations on an ATM network, we achieve a simple user-to-user round-trip RPC in 170 microseconds. Our experiments suggest that RPC software overhead contributed by marshaling, stubs, and the packet exchange protocol need not be the bottleneck to low-latency remote communication on modern microprocessors. However, as software overheads decrease, network controller design becomes more crucial to achieving truly low-latency communication.

2. LOWER BOUNDS FOR CROSS-MACHINE COMMUNICATION

This section evaluates the overhead added to cross-machine communication from two important sources: (1) the network hardware (the controller and the communications link) and (2) the controller, memory, and CPU interfaces. This allows us to examine our RPC software (the stubs, RPC packet exchange protocol, and runtime support) relative to the lower-bound message-passing costs for cross-machine communication on our hardware. Given different network and processor technologies, the relative importance of these components may change; by isolating the components, we can see how the performance of each layer scales with technology change.

Similar measurements have been reported in the past [18, 27]. However, we wish to extend those measurements, first to examine newer technology networks, and second to compare different high-speed networks with each other and with Ethernet. In the following we briefly characterize our hardware and explain our experimental testbed and measurement methodology. Then we analyze our performance results and discuss controller and network issues that specifically impact latency in cross-machine communication.

2.1 Overview of the Networking Environment

This subsection summarizes the various networks we used and the particular capabilities of the specific network controller used to access each network.

Ethernet. The Ethernet is a 10 Mbit/sec CSMA/CD local area network, which is accessed on our DECstations and SparcStations by a LANCE controller [1]. However, the controller is packaged differently on the two machines. On the DECstations, the controller cannot do DMA to or from host memory; instead, it contains a 128-Kbyte on-board packet buffer memory into which the host places a correctly formatted packet for transmission [14]. Similarly, the controller places incoming packets in its buffer memory for the host to copy. In the case of the SparcStation, the controller uses DMA to transfer data to and from host memory. In this case, a cache flush operation is done on receives to remove old data from the cache. On both machines, packets are described by special descriptors initialized by either the host (on send) or the controller (on receive). Descriptors are kept in host memory on the SparcStations while they are in the special on-board packet memory on the DECstations. Two message sizes were used in our experiments, a minimum-sized (60 bytes) send and receive, and a maximum-sized (1514 bytes) send and receive.

FDDI. FDDI is a 100-Mbit/sec fiber token ring, accessed on the DECstation by the DEC FDDI controller. Like the DECstation Ethernet controller, the FDDI controller cannot perform DMA from host memory on message transmission; instead, it relies on an on-board packet buffer memory. However, the FDDI controller can perform DMA transfers directly to host memory on reception of a packet from the network. The controller and host software share descriptors as described above for the DECstation Ethernet. We used an unloaded private FDDI ring with two hosts. Thus, the overhead due to token passing is kept to an absolute minimum; in a more realistic environment, token-passing delay would have to be added to the overall latency. Packet sizes of 60 and 1514 bytes were chosen to facilitate direct comparison with the Ethernet.

ATM. ATM (Asynchronous Transfer Mode) is an international telecommunication standard used to implement B-ISDN. In an ATM network, data is exchanged between entities in fixed-length parcels called cells, usually on the order of a few tens of bytes. An ATM network typically consists of a set of hosts connected by a mesh of switches that form the network. In an ATM network, user-level data is segmented into cells, routed, and then reassembled at the destination using header information contained in the cells.

The particular ATM we used has 140-Mbit/sec fiber optic links and cell sizes of 53 bytes and is accessed using FORE Systems' ATM controller [16]. The controllers on the two DECstation hosts were directly connected without going through a switch; thus there is no switch delay. Unlike the Ethernet and FDDI controllers, the ATM controller uses two FIFOs, one for transmit and the other for receive. The controller has no DMA facilities. The host simply reads/writes complete ATM cells by accessing certain memory locations that correspond to the FIFOs. The host is notified via interrupt when cells arrive in the receive FIFO. The host has considerable flexibility in choosing how often it should be interrupted. Further, the controller does not provide any segmentation or reassembly of cells; that is the responsibility of the host software. Each ATM cell carries a payload of 44 bytes; in addition, there are 9 bytes of ATM and segmentation-related headers and trailers. In our experiment we chose packet sizes that were an integral number of cells as well as being close enough to the Ethernet and FDDI packet sizes for comparison.

2.2 The Testbed and Measurement Methodology

In order to isolate the performance of the controller and the network link, we built a minimal stand-alone testbed, which simply sends and receives packets on the network. There is no operating system intervention since only minimal software is executing on each machine. The testbed hardware consists of two workstations (either two DECstations or two SparcStations) connected through an isolated network. The DECstation uses a 25 MHz MIPS R3000 processor rated at 18.5 SPECmarks, and the SparcStation I uses a Sparc processor rated at 24.4 SPECmarks. The DECstations were connected in turn to an Ethernet, an FDDI ring, and an ATM network. The SparcStations were connected to an Ethernet. The DECstation network devices are connected on the 25 MHz TURBOChannel [13] while SparcStations use the 25 MHz SBus [29]. We measured the performance of each configuration in sending a source packet from one node to the other and in receiving a packet in response. Packets are sent and received from host memory, so the cost of moving the data over the host bus is included in our measurements. Network interrupts are enabled, so both the sender and the receiving hosts are interrupted on packet arrival. While it is generally possible to access the network in a dedicated fashion by disabling network interrupts, conventional time-sharing access will involve interrupt-processing overheads.

Both the DECstation and the SparcStation have clock chips that can provide periodic interrupts. These were set to provide interrupts at 4096 Hz on the DECstations (about once every 244 microseconds) and 1600 Hz on the SparcStations (once every 625 microseconds). No significant processing is involved in fielding a timer interrupt. Measurements were averaged over at least 10,000 successful repetitions.

The component costs for the round trip can be broken up as follows:

—**Time on the Wire**. This is the transmission time of the packet. We ignore the propagation time because it is negligible for the length of cable we are using.

—**Controller Latency**. This is the sum of two times: (1) the time taken by the sending controller to begin data transfer to the network once the host has made the data available to it, and (2) the delay between the arrival of the data at the receiving controller and the time it is available to the host.

—**Control / Data Transfer**. Data has to be moved at least once over the host bus between host memory and the network. Some of our controllers use DMA to move data over the host bus to the network; thus the CPU incurs no data transfer overhead. However, the CPU incurs a control transfer overhead because it has to use special memory descriptors to describe the location of the data to the controller. With such controllers, the actual time to do the data transfer is captured in the **Controller Latency** item.

—**Vectoring the Interrupt**. On the receive side, host software must vector the packet arrival interrupt to the interrupt handler in the device driver. The overhead involved is a function of the CPU architecture.

—**Interrupt Service**. On taking an interrupt, host software must perform some essential controller-specific bookkeeping before the interrupt can be dismissed.

2.3 Performance Analysis

To determine the latency of the controller itself, we ran separate experiments between a pair of hosts with interrupts disabled. Each host polled the controller's status registers in a loop. As soon as the register indicated arrival of data, a new transmission was begun. In the cases where the host was expected to copy data to the controller's memory before transmission, the host simply programmed the controller to start the data transfer, without actually giving it any data. Similarly, when the controller indicated the arrival of new data for the host to copy, the host ignored the data and began the next transfer. In addition, descriptors were prefilled before the data transfer started. In these circumstances very few machine instructions are executed by the host per round trip. The time required to execute these as well as the time spent on the wire was subtracted from the total measured round trip. This method gives satisfactory results in most cases but has the disadvantage that it does not account for any pipelining that the controller might perform. This artifact is particularly visible in the case of our ATM controller when multicell packets are exchanged. Typically, a controller chip can overlap the transmission of data between its internal buffer and the network, with the transmission between host (or on-board memory) and the chip itself. For instance, in sending a multicelled packet through the ATM controller, the host can fill the FIFO with a cell while the controller sends the previous cell from the FIFO onto the network.

Table I shows the cost of round-trip message exchanges on the host/network combinations described above. A few points of clarification are in order here.

First, in the case of the FDDI controller and the SparcStation controller, which perform DMA directly to host memory on packet receives, the cost of flushing the cache after the DMA is included in the **Interrupt Service** overhead. While it is true that cache flushes are not strictly necessary if data from the network is kept in uncached-memory locations, this means that the higher-level software will eventually pay a performance cost of accessing this data.

Second, in the case of the ATM network, Table I does *not* include the cost for segmenting and reassembling multicell packets. In addition, the controller was programmed so that it interrupted the host only when *a complete packet* had arrived in the FIFO. Thus, in our experiments, each round trip incurs only two interrupts. The performance reported is therefore a lower bound.

The row named **Sum of Components** is the sum of the rows above it. Most of the time, the sum of our component measurements is within 1–9 microseconds (about 2%) of the observed round-trip time. The only exception is in the case of the ATM network in sending multicell packets, where we have overestimated the round-trip time by about 12%. The most likely cause is an underestimate of the amount of overlap between the host memory–FIFO data transfers and the FIFO–network transfer.

2.3.1 *Throughput and Latency.* It is interesting to compare the ratio **Controller Latency** to **Time on the Wire**. For small packets, this is 0.4 on the DECstation Ethernet, 0.8 on the SparcStation, 7 for the FDDI controller, and 3.2 for the ATM network. For larger packets of approximately 1514 bytes, these ratios are respectively 0.02, 0.04, 0.9, and 1.0. While these numbers are specific to the controllers we use, we believe they are indicative of a trend: *bandwidths are improving dramatically while latencies are not.*

The packet exchange times on Ethernet, FDDI, and ATM show the important difference between throughput and latency. If low latency for small packets is the goal, then we can achieve a round-trip 60-byte message exchange on our DECstation Ethernet in only 253 μseconds; FDDI on similar

Table I. Hardware-Level Round-Trip Packet Exchange Times in μseconds

Component	Round-Trip Time (μseconds)							
	Packet Size in bytes (send/recv)							
	Ethernet (DEC)		Ethernet (Sparc)		FDDI (DEC)		ATM (DEC)	
	60/60	1514/1514	60/60	1514/1514	60/60	1514/1514	53/53	1537/1537
Time on the Wire	115	2442	115	2442	13	245	6	176
Controller Latency	51	53	89	103	97	230	16	161
Control/Data Transfer	40	600	6	6	40	253	17	458
Vectoring the Interrupt	25	25	12	12	25	25	25	25
Interrupt Service	26	26	42	42	92	140	9	20
Sum of Components	257	3146	264	2605	267	893	73	840
Measured Round Trip Time	253	3137	263	2611	263	894	73	746

hardware, despite its 10-fold bandwidth advantage, takes 263 μseconds for the same operation, which is 4% longer. The ATM network is capable of doing single cell transfers in about 73 μseconds. However this is an optimistic lower bound because we have ignored switching delays, and the cost of checking whether the cell is part of a larger message that needs fragmentation/reassembly.

On the other hand, the high bandwidth of ATM and FDDI is significant for large packets. For example, latency for a 1514-byte packet on the DECstation Ethernet is 4.2 times worse than ATM and 3.5 times worse than FDDI. For packets even larger than 1514 bytes, the Ethernet situation is relatively worse, because FDDI will require fewer packet transmissions. The ATM comparison is slightly more complex; as pointed out earlier, we have ignored ATM segmentation and reassembly costs in arriving at the figures in Table I. The particular software implementation of the segmentation/reassembly we use requires about 11 μseconds per cell. If this is included, then a 1537-byte packet (29 cells) takes about 1065 μseconds, which is about 1.2 times the FDDI time. The situation further improves in favor of FDDI for packet sizes beyond this limit.

2.3.2 *Controller Structure and Latency*. As noted earlier, both the DECstation and the SparcStation use the same Ethernet controller. However, their performance is not identical. Recall that on the SparcStation the controller uses DMA transfers on the host bus. The overhead for this is included in **Controller Latency**; the **Control / Data Transfer** costs include only the cost of the instructions to program the controller. The controller is able to overlap the data transfer over the bus with the transfer on the network. Thus the sum of **Controller Latency** and **Control / Data Transfer** is relatively unchanged by the packet size. In contrast, the DECstation controller incurs a heavy latency overhead because the host first has to copy the data over the bus before the controller can begin the data transfer. Consequently, for larger packets, the SparcStation controller performance is likely to be better even though for small packets both have comparable round-trip times. Controllers that use on-board packet buffers instead of DMA or FIFO will generally incur this limitation. However, controllers with DMA or FIFO are not without problems; we will defer a more detailed discussion until Section 4.

It is also interesting to compare the interrupt-servicing latency on the various controllers. Both the Ethernet controllers have comparable interrupt-handling times. The SparcStation figure is slightly higher because a cache flush operation is included in the cost due to the DMA transfer. All except the ATM controller use descriptor-based interfaces between the host and the interface. Programming this requires many more accesses to memory than the simpler FIFO interface on the ATM. The FDDI controller is the most complex one to service and is 7–10 times as expensive as the ATM. While this figure is for a specific pair of controllers, we believe that FIFO-based controllers will in general reduce programming overhead.

Our experiments with low-level message passing seem to suggest that for lowered latency, simple FIFO-based controllers have some advantages over the more traditional types of controllers. However, ultimately it is the impact of controllers on user-to-user cross-machine communication that is crucial. In the next section, we describe a higher-level communication system based on RPC and explore the impact of controllers in this context. Unlike low-level message-passing performance, user-level cross-machine communication has to be concerned with additional issues like protection and input packet demultiplexing.

3. THE DESIGN, IMPLEMENTATION, AND PERFORMANCE OF A LOW-LATENCY RPC

The performance of the hardware and of the low-level message-passing system are two of the three components of user-level communication latency. If the performance of the third component, the high-level RPC system, is poor, this can easily dominate the overall cost. The purpose of this section is twofold. First, we wish to show that the technology exists for building RPC systems that are so efficient that the costs described in the previous section *are* significant. Second, we wish to outline some of the low-latency techniques used by higher-level software and how these might interact with the structure of the network controller.

3.1 Design and Implementation

The RPC system [27] built for the DEC SRC Firefly multiprocessor workstation [30] has demonstrated the latency-reducing effect of a large number of optimizations. Our RPC system closely follows the SRC design; however, where performance is a goal, many details in the structure of a communications system must be dictated by the hardware environment. Thus, our system differs from SRC RPC in several respects:

(1) We differ in the way stubs are organized and the way marshaling is performed: we use a scheme that minimizes copying costs without compromising protection between kernel and user spaces as is done in SRC RPC.
(2) Our system differs in the way in which control transfer is done.
(3) We do not use UDP/IP, but instead use network datagrams directly.

These differences are described in more detail in the following subsections.

Our low-latency RPC system is prototyped on the DECstation 5000 running the Ultrix operating system and on the SparcStation I running SunOS. The system is integrated into the Ultrix and SunOS operating systems executing on these machines without impacting other networked or distributed services.

Our implementation is entirely in C; we have not felt the need to program in assembly language, because our compilers generate high-quality code. The RPC system has a runtime component that is linked into the user's address space and another component that is integrated into the kernel.

Like other RPC systems, clients and servers can be placed on different machines; during the runtime binding process, the client imports the interface previously exported by the server. We have specially optimized for the common case [4] of single-packet transfers between machines with the same byte order. Multipacket RPCs are handled by a separate code path so as not to impact the common case. Byte-swapping overhead is reduced by ensuring that only the client, but not the server, swaps byte order when needed.

The underlying communication in the RPC relies on a simple request and response packet exchange similar to that described in Section 2. In addition, however, there are three fundamental aspects to RPC that add overhead beyond what is required for a simple message exchange: *marshaling and data copying*, *control transfer*, and *protocol processing*. Our system achieves its low latency by optimizing each of these areas. The optimizations are somewhat interrelated, and we consider them in turn in the following subsections.

3.1.1 *Marshaling and Data Copying.*

RPC stubs create the illusion of a simple procedural interface for the client and server. Stubs are application-specific and depend on the particular service function that is invoked. These procedures marshal the call arguments into and out of the message packet. Even in highly optimized RPC systems such as SRC RPC, marshaling time is significant. Marshaling overhead depends on the size and complexity of the arguments. Typically the arguments are simple—integer, booleans, or bytes; more involved data structures are used less frequently [4]; therefore simple byte copying is sufficient.

Related to the cost of marshaling is the cost of making the network packet available to the controller in a form suitable for transmission. A complete transmission packet contains network and protocol headers, which must be constructed by the operating system, and user-level message text, which is assembled by the application and its runtime system. There are several strategies for assembling the packet for transmission, with the cost depending on the capability of the controller and the level of protection required in the kernel. With a controller that does scatter-gather DMA over the bus, like the SparcStation controller, the data can be first marshaled in host memory by the host (in one or more locations) and then moved over the bus by the controller. This is the scheme we use on the SparcStations—the data packet comes from user space, and the header is taken from kernel space, a technique commonly used in such an environment [18, 25]. However, due to the way SparcStation DMA is architected, the controller performs DMA only to and from a fixed range of kernel virtual addresses. This necessitates mapping the user's data buffer into kernel addresses that are accessible to the controller. Since the cost of using SunOS virtual-memory primitives for this mapping is more expensive than copying for packet sizes below a threshold, our RPC design uses mapping only selectively.

One general drawback with a DMA scheme is that it involves at least two accesses of the data over the bus: once when the host builds the pieces of the packet and again when the controller transfers the pieces to the network. Likewise, given a controller with special on-board memory or a FIFO, the straightforward technique of the user marshaling the data into a buffer and the kernel copying it over to the controller requires two copies. Another approach is to relax kernel/user protection and to map the packet buffer into user space and allow the user direct access. This reduces the number of copies to one. However, there is an alternative technique that retains all the benefits of the protection without incurring the cost of two copies.

In an effort to minimize copying of the call arguments, we perform argument-marshaling *in the kernel* rather than in the user's address space, as is conventional. To do this we synthesize code on the fly, which is then linked into the kernel and executed. Code synthesis has been used in the past to generate optimized routines for specific situations to achieve high performance [20, 22]. Our focus is slightly different: we are more concerned with avoiding the copy cost than with generating extremely efficient code for a special situation.

At bind time, when the client imports the server's interface, the client calls into the kernel with a template of the marshaling procedure. The kernel directly supports simple-valued types such as words, halfwords, bytes, and pointers to bytes. Using this template the kernel synthesizes a marshaling procedure.[1] As mentioned earlier, the marshaling is typically simple and involves only assignments and byte copying. Thus, the task of synthesizing a procedure is nothing more than assembling the right sequence of primitive instructions. The marshaling procedure contains code to check the validity of each input argument passed at runtime. This approach has the benefit that since the sizes of the request and reply are known in advance, the more general multipacket code path can be avoided if arguments and results fit in a single network packet.

The kernel then installs the synthesized procedure as a system call for subsequent use by this specific client. Thus, the stubs linked into the user's address space do not do marshaling; they merely serve as wrappers to trap into the kernel where the marshaling is done. A client RPC sees a regular system call interface with all the usual protection rules that go with it. This approach has the benefit of performing the minimum amount of copying required without compromising the safety of a "firewall" between the user

[1] We do not yet have a template compiler, and so our kernel stubs are currently hand generated.

and the kernel, or the user and the network controller. Our scheme does impose the overhead of probing the validity of pointers before data can be copied, but this is not a significant cost for most RPCs.

In addition to this scheme, which handles most of the common cases, our RPC system provides another interface that is designed for a more general case. Instead of calling into a specialized kernel-marshaling procedure, the client calls into a fixed-kernel entry point, passing an array of data descriptors. Each descriptor describes a primitive data type (including its parameter type—IN or OUT) that is directly supported. The kernel can use these descriptors to marshal and unmarshal arguments directly between the user space and the controller's memory or FIFO, thereby eliminating extra copying cost without compromising safety. Unmarshaling on the server side is done using this general-purpose interface as described above. A server typically exports several procedures with different types of arguments. The server provides the kernel with a generic template that applies to all types of received packets.

We have used kernel-level marshaling and unmarshaling with the DECstation Ethernets, but a hybrid scheme is used with the FDDI controller. On transmissions the usual scheme is used, but on receives, the controller's DMA engine copies the data over the host bus and hands it to the kernel. The kernel unmarshals the data either by copying or by virtual-memory mapping if the alignments are suitable. A similar approach is used with the ATM controller. In this case, before the RPC layer receives the packet, the device driver performs reassembly in a page-aligned buffer. Furthermore, the driver could read only the header from the FIFO and determine if reassembly is required, and if not let the RPC layer unmarshal the data from the FIFO. In fact, in a non-ATM network accessed via FIFOs, this would be the preferred method, but reassembly is such a common and frequent operation on the ATM that we did not feel justified in making this optimization.

3.1.2 *Control Transfer*. Researchers have reported that context switching causes a significant portion of the overhead in RPC [27]; in addition, there is a substantial impact on processor performance due to cache misses after a context switch [24]. An RPC call typically requires four context switches: switching the client out, switching the server in, switching the server out, and finally switching the client back in. Two of these—switching the client or the server out—can be overlapped with the transmission of the packet. Systems with high-performance RPC usually have lightweight processes that can be context switched at low cost, but unless there is more work to do in the client and the server, or no work elsewhere, a process context switch usually occurs.

Both the DECstation and the SparcStation have context-switching times that can be significant to the latency of a small packet. To reduce the cost of context switches, our RPC system defers blocking the client thread on the call. Instead, the client spin-waits for a short period before it is blocked on the sleep queue. If the service response time is very small, the reply from the server is received before the client's spin-waiting period has expired. When there is no other work to be done, i.e., when the run queue is empty, there is no penalty to spinning the caller indefinitely. When there is useful work to be done, the caller spins for a short time relative to its round-robin quantum before blocking.

In most cases the low response time of the server ensures that the process is never put to sleep. This approach can be improved if estimates of the round-trip time are available. A simple extension to the current scheme would be to block the caller without spinning if the expected round trip is greater than some threshold related to the context switch penalty and to spin otherwise. An estimate of the round trip could be obtained either statically by using a user-supplied hint, or dynamically, by using past response times as an estimate. In general, this technique trades throughput for latency if there are processes on the run queue waiting their turn.

Additional control transfer overhead arises from protocol layering. The traditional approach is for each layer to queue the incoming packet on the input queue of the next higher layer; software interrupts are then used to wake a thread of control in that layer. This leads to a modular approach but often with unacceptable performance. In contrast, we try to dispatch the packet directly from the lowest layer to the destination process. This dispatch is done directly within the interrupt handler, yielding a path of very low latency.

3.1.3 *Protocol Processing*. The overhead of protocol processing can dominate communication costs if general-purpose protocols are used for RPC. In the usual case of a homogeneous environment, with frequent remote requests and low service response times, special-purpose protocols can be effectively employed to optimize the latency.

There are typically two protocol layers in RPC systems: a transport-level protocol like UDP/IP that provides a basic unreliable transport, and a specialized RPC protocol to provide a close approximation of conventional procedure call semantics for RPC.

Several aspects of protocols contribute to RPC latency. As mentioned above, multiple layers of protocol tend to add to the overhead of RPC in two ways: increasing the number of context switches and increasing the number of data copies between layers. Further, the primary cost of using protocols such as UDP/IP is the cost of checksums. Calculating checksums in the absence of hardware support involves manipulating a packet as a byte stream; this can nullify any advantage gained by the controller or host processor in assembling the packet using scatter-gather or wordlength operations. For efficient RPC implementations, then, the checksum must be either calculated in hardware or made optional. Most conventional protocol formats do not lend themselves to the former approach. The latter approach presupposes that the transmission medium is reliable and that the transport protocol is used only for routing, not for reliability. These factors argue for the use of a simpler protocol and hardware checksumming.

Our RPC uses a simple and efficient unreliable transport protocol and relies on a specialized RPC protocol to provide robustness, which is similar to that used by the SRC or Xerox [5] RPC systems. In general, the choice of protocols reflects a set of assumptions about the location of clients and servers and the error characteristics of the network. Typically clients and servers are expected to be on the same local area network. Furthermore, local area networks have good packet loss characteristics, dropping packets at only heavy loads due to overruns. In the case when the RPC destination is within the same LAN, raw network datagrams are used with the checksum provided by the controller. An erroneous packet is simply dropped by the receiving controller. If the target is not on the local network, it is a straightforward extension to use UDP/IP without checksums. The choice can be made at bind time when the client/server connect is established, and the marshaling code can be generated to include the appropriate header.

In addition to the overhead imposed by transport-level protocols, the RPC protocol per se adds to the latency. The primary purpose of RPC protocols is to provide a natural set of semantics reminiscent of simple procedure call. To achieve high performance, our protocol was implemented so as not to intrude on the critical fast-path of the code.

Under normal error-free operation, the server's response to a call from the client acknowledges it at the client end. Similarly the next call from the client acknowledges the previous response. The state of an RPC is maintained by the client and server using a "call/response" record. Call records are preallocated at the client at bind time. These contain a header, most of whose fields do not change with each call. These are therefore prefilled so as to minimize the latency at call time. Similarly, response records are retained at the server side and contain header information from a previous call that can be reused.

In order to recover from dropped packets, our RPC transport buffers packets on the client and the server side. On the client side, since the client is blocked for the duration of the call, retransmission proceeds from the data in the client address space, as with the original call. Thus no latency is added due to buffering when the call is first transmitted. On the server side, the reply is nonblocking; hence a copy of the data has to be made before returning from the kernel. The copy is overlapped with the transmission on the network. Once again no latency is added to the reply path. One alternative to this would be to use Copy-on-Write, which would not affect latency but can potentially save buffer space for large multipacket RPCs.

3.2 RPC Performance Measurements

This section examines the performance of our RPC implementation. Our goal is to show that structuring techniques, such as those we have used, yield an RPC software system so effective that the hardware costs shown in Table I are significant. We gather the data to support our analysis of controller design in the next section.

Table II shows the time in microseconds for RPC calls on the various platforms. Two procedures called Minus and MaxArg were timed. Minus takes two integer arguments and returns an integer result. MaxArg takes two arguments—an integer parameter and a variable-length array, returning an integer result. The exact sizes of the packets exchanged varies depending

Table II. Allocation of RPC Time in Microseconds

Activity	RPC Time (μseconds)							
	Ethernet (DEC)		Ethernet (Sparc)		FDDI (DEC)		ATM (DEC)	
	Minus	MaxArg	Minus	Maxarg	Minus	Maxarg	Minus	MaxArg
Client Call	28	145	59	137	46	173	25	159
Controller Latency	26	27	45	54	48	82	8	44
Time on the Wire for Call	58	1221	58	1221	4	122	3	88
Interrupt Handling on Server	25	25	27	27	56	70	17	20
Server Packet Receipt	39	470	59	169	42	42	29	347
Server Reply	27	27	46	46	36	36	25	25
Controller Latency	25	25	44	44	49	82	8	44
Time on the Wire for Reply	57	57	57	57	5	5	3	3
Interrupt Handling on Client	26	26	27	27	56	56	17	17
Client Reply Receipt	29	29	49	49	29	29	29	29
Total Attributed Time	340	2052	471	1831	371	697	164	776
Measured Time	340	2070	496	1997	380	693	170	675

on the network type. Minus causes 60 bytes to be exchanged on the Ethernet, 48 bytes on FDDI, and 53 bytes on the ATM. MaxArg transmits to the server 1514 bytes on Ethernet and FDDI, and 1537 bytes on the ATM. The reply from the server is 60, 48, and 53 bytes respectively on the three networks. Measurements were made both in single-user and in multiuser modes. The times were not significantly different; the times reported in the tables are the single-user measurements which showed less variance.

The various rows are explained below:

—**Client Call**. This is the total time required on the client side for sending out a call packet. It consists of five major components: the time for setting up the argument to the system call, the time to perform a kernel entry, the time to validate the arguments, the time spent in the marshaling procedure, and the time for setting up the controller to do a transmit.

—**Controller Latency**. This figure represents the controller's latency in getting the packet to and from the wire. This is computed as in Section 2.

—**Time on the Wire for Call**. An estimate of the packet transmission time based on the bandwidth of the network.

—**Interrupt Handling on Server**. This is a sum of the times to vector the network interrupt to the interrupt handler in the device driver, the time in the handler, and to return after the interrupt is dismissed.

—**Server Call Receipt**. This is the time spent on the server machine when the call packet arrives before it is handed to user code. It consists of two major parts: the cost of examining the packet and kernel data structures to locate the correct server and dispatch the packet, and the time spent in unmarshaling and copying/mapping data into the server's address space. The cost of this component varies depending on the capabilities of the controller.

—**Server Reply**. This is quite similar to **Client Call**. This includes the time needed to call the correct server procedure and to set up the results for the system call, the time spent in the kernel in checking the arguments, and the time to set up the controller for transmit.

—**Client Reply Receipt**. This is the counterpart of the **Server Call Receipt** and takes place on the client side.

—**Total Attributed Time**. This is the total of the components. It is the time we have been able to attribute to the various activities, either by direct measurements or by estimating from functional specifications.

A few points about Table II are in order here. First, though the SparcStation and the DECstation 5000 have similar CPU performance and use identical Ethernet controllers, the software cost for doing comparable tasks varies. In general, our experience has been that SunOS extracts a higher penalty than Ultrix. We believe this is primarily due to the richer SunOS virtual-memory architecture rather than to the SparcStation's architectural peculiarities such as register windows. Second, in the case of MaxArg on the ATM network, 29 cells are sent to the server. The cost of segmenting the user packet into 29 cells at the sender is included in the row entitled **Client Call**. Similarly, the cost of reassembling the cells is included in the **Server Packet Receipt** row. In the case of single-cell transmission, the segmentation and reassembly code is completely bypassed. Finally, we exploited the flexible interrupt structure of the ATM controller to ensure that in the normal case only the last cell in a multicell packet caused a host interrupt. In comparing the **Total Attributed Time** with the **Measured Time** we note that as in Section 2, in all cases except one, the difference is within experimental error, in the range of 1–8%. The only exception is in the case of a multicell packet on the ATM, where for the reasons mentioned earlier in Section 2, we have overestimated the total time required.

Overall, as we can see by comparing the FDDI and the DECstation Ethernet tables, the latency for the small Minus RPC request on FDDI is 12% higher than for Ethernet. On the other hand, as expected, the much higher bandwidth on FDDI becomes evident for the larger packet sizes; the MaxArg RPC request takes nearly three times longer on Ethernet than on FDDI. Comparing the performance on the ATM network with the others, we see how effective a simple controller design is for reducing latency for small packets. However, for larger packets, the software fragmentation/reassembly is a factor in reducing performance. ATM controllers that perform fragmentation and reassembly either in hardware or by dedicated on-board processors are being built [12, 32]. It would be interesting to compare the latency characteristics of these controllers with the simpler approach taken in the FORE controller we use.

As noted earlier in Section 2, for small packets, the cost of "doing business" (i.e., controller latency) on the network has increased with FDDI. However, compared to the low-level software latency imposed by the packet exchange, the higher-level RPC functions (stubs, context switching, and protocol overhead) add only 87 μseconds for Ethernet, 117 μseconds for FDDI, and 97 μseconds for ATM.

Higher-level RPC functions cost more on FDDI for two reasons, both relating to the controller/host interface. First, because of the more complex DMA interface the high-level cost of dispatching the server process and data management is greater for the DEC FDDI. Second, for both FDDI and Ethernet, the code path required for RPC for these controllers is slightly different from that used for a simple packet exchange. Thus, there is an inherent overhead added by RPC. However, the overhead is more in the case of FDDI than for the Ethernet because of some of the characteristics of the controller/host interface.

While the absolute increase contributed by RPC functions is comparable for both ATM and Ethernet, as a percentage, RPC adds about 130% to the cost of low-level messages. Part of this overhead is due to architectural features like traps and context switching that may not scale with processor speeds. However, the bulk of the overhead can be reduced with increasing CPU speeds. A significant factor in achieving this reduction is to keep the overheads of memory copying to a minimum.

The main factors that contribute to the system's performance are (1) the use of preallocation, (2) overlapping computation and network transmission, (3) exploiting the peculiarities of the network controller, optimizing for the simple and common case, and (4) the speed of the host processors.

4. IMPLICATIONS FOR CONTROLLER AND NETWORK DESIGN

Our experience with low-latency RPC on a variety of controllers indicates that the design of the controller has a significant influence on efficient cross-machine communication overheads. Similarly, different network access protocols also determine the latency experienced by an RPC. Our measurements in Tables I and II reinforce the fact that a faster network does not necessarily imply a lower latency. In this section, we discuss the effects of both the controller and the network on latency.

4.1 DMA versus Programmed I/O

Our experiments, which included controllers that are capable of scatter-gather DMA, ordinary DMA, and no DMA, allow us to examine some of the essential latency-impacting tradeoffs between these different controller types. There are basically two issues to be considered in choosing between controllers that provide DMA and those that support programmed I/O (PIO) (i.e., the processor moves the data to and from the bus, usually without using a block transfer primitive). These issues are (1) the cost of servicing the device and (2) the overhead of transferring the data to the user. We discuss each of these in more detail below.

4.1.1 *Data Transfer.* The details of moving data on the system bus vary depending on the capabilities of the controller. Certain combinations of features make it difficult to restrict the number of data movements to one, the minimum achievable. Excessive and unnecessary data movement imposed by the lower levels of the system can lead to bad overall performance of the communication system. We discuss below the interaction between controller types and copying costs.

Table III summarizes the number of copies that the controller, the kernel, and the user need to perform, respectively, for each type of device: PIO, DMA,

Table III. Number of Copies for Various Controller Types

Number of Copies (Device/Kernel/User)				
Fragment	**PIO (KM)**	**PIO**	**DMA**	**DMA (S-G)**
	Send			
Header	0/1/0	0/1/0	1/1/0	1/0/0
Data	0/1/0	0/1/1	1/0/1	1/0/0
	Receive			
Header	0/1/0	0/1/0	1/1/0	1/0/0
Data	0/1/0	0/1/1	1/0/1	1/0/0

and DMA with scatter-gather. The column **PIO (KM)** represents PIO with some form of kernel-level marshaling as described in Section 3, for instance, FRPC running on the ATM controller with its FIFOs. The column **DMA S-G** represents scatter-gather DMA.

Certain implicit assumptions made in the table are clarified below. First, we are ignoring the cost of copies that might occur between the network and the controller's internal buffer. Typically this cost can be made negligible either by using video RAMs or some similar technique. Second, we assume that with PIO, the on-board memory, or FIFO, cannot be reliably mapped into multiple user spaces, while with DMA, user data is mapped (instead of copied) to be adjacent to the header in kernel memory, so that the DMA can use a contiguous set of addresses. Finally, we assume that with scatter-gather DMA, the controller is capable of transferring arbitrarily small amounts of data over the bus and that on scatter DMA, the controller can perform address demultiplexing so that incoming data goes to the correct destination.

Ideally, one would like to minimize the number of times data is moved on the host bus between the network and the host memory. With PIO and kernel-level marshaling it is possible to keep the number of copies to one without compromising protection. With DMA this would not be possible, in general, because application-level data could be located in multiple noncontiguous locations in memory. Most scatter-gather controllers (e.g., LANCE) have minimum size requirements for each segment and a maximum number of allowable segments. Thus, the user will have to marshal the data into one (or a few) location(s), and then have the controller move it. While it is possible to build controllers to overcome this restriction[2]. Using several small segments to gather data comes at a price, because the controller has to set up multiple DMA transfers. A similar situation is true on the receiving side as well.

As shown in Table III, using PIO with kernel-level marshaling allows the data-copying cost to be kept to the minimum possible. However, one aspect of copying that is not captured in the table is the different rate at which data is

[2] the Autonet controller being built at DEC SRC; personal communication, Charles P. Thacker, Sept. 1991.

moved over the bus for PIO and DMA. Typically, word-at-a-time PIO accesses over the bus are slower than block DMA accesses; this is the case on both the DECstation TURBOChannel bus and the SparcStation SBus. While PIO versus DMA is of limited concern for short packets, there is a break-even point beyond which PIO will be slower than DMA. Thus, unless the processor is required to touch each byte of the data for reasons other than moving it across the bus (for instance, to generate a checksum in software), it is usually more efficient to use DMA beyond a certain size.

4.1.2 *Cache and TLB Effects.* While Table III seems to indicate that PIO and scatter-gather DMA can perform the same number of copies, very often with current architectures the interaction of the memory subsystem with DMA might extract a heavier overall penalty than PIO. This subsection describes this effect in more detail.

In addition to the copying costs outlined above, without adequate support from the memory and processor subsystem, controller-initiated data transfers could be a source of overhead due to cache effects. If the cache does not snoop on I/O operations, cache blocks could be left incoherent as a result of the DMA operation to memory, requiring cache flushes. If the cache is write-back, dirty entries may need to be purged before a DMA operation from memory.

Table IV shows the contribution of the cache flush cost as a percentage of the total interrupt-handling cost of our DMA controllers. The total interrupt-handling cost is the sum of cache flush costs and the essential controller-related bookkeeping overhead. The cache flush cost is simply the time taken to execute the instructions required by the host architecture to flush the cache lines corresponding to the data that was transferred. As is evident from Table IV, cache flushes are a serious penalty on current memory architectures. However, the situation is even worse than the table suggests, because in addition to the costs of executing additional cache flush instructions, cache flushes have a negative impact on performance by destroying locality.

Newer processor/cache designs recognize this problem and provide memory coherence for DMA [15]. In the absence of snooping caches, the usual "solution" to this problem is to allocate buffers temporarily in uncached memory before they are copied to user space. This approach, used in the stock SunOS Ethernet driver, either incurs an extra copy overhead or loses the benefit of cached accesses to frequently used data.

Another cost of DMA is the manipulation of page tables that is often necessary. On packet arrival, the controller stores the data on a page; however, there is generally no way to guarantee that the page is mapped into the correct destination address space. Thus the kernel is faced with the option of either remapping or performing an extra copy. On a multiprocessor, this remapping can require an explicit and expensive TLB coherency operation [6].

To summarize our experience with the various controllers, we believe that DMA capabilities without adequate support from the cache and memory subsystem can be bad for performance in modern RISC processors. We also believe that controllers that have simpler interfaces to the host have the

Table IV. Cache Flush Cost

	Received Packet Size in bytes			
	Ethernet (Sparc)		FDDI (DEC)	
	60	1514	60	1514
Total Interrupt Time	21	21	46	70
Cache Flush Time	3	10	7	30
Percentage Overhead	14	48	15	43

potential for reducing overheads. The next subsection argues for the use of simple controller/host interfaces.

4.2 Host / Controller Memory Interface

There are two common ways of transferring data between the host and the controller. One way is to designate a range (possibly all) of host memory to be used as a packet buffer and have the controller and the host share descriptors in memory. The other alternative is to use a simple FIFO for transmits and receives and have the host access it directly. A significant overhead in interfacing the controller to the processor system is the cost of servicing an interrupt and getting the data to the user. We have already discussed the role of data movement overhead and shall therefore restrict ourselves to interrupt-handling costs of different controller types.

Our experiments with the two types of controllers mentioned above indicate that the interrupt-handling overhead can be significantly reduced by using a FIFO. Interrupt-handling cost is composed of two components: (1) the CPU-dependent cost of vectoring the interrupt and (2) the controller-dependent cost of servicing the interrupt. Previous research has studied interrupt-vectoring costs on RISC processors [3, 26], and so we shall examine only the second component. Our objective is to compare a simple FIFO interface such as that found in the ATM with a more elaborate descriptor interface such as that found in the Ethernet or the FDDI controllers.

We reproduce some of the measurements from Section 2 in Table V. The table shows the cost of servicing the interrupt and transferring the data through a copy or a remapping operation, as appropriate. Since our intent here is to compare the interfaces and not the network, we have ignored the reassembly overhead on the ATM controller.

As the table indicates, for transferring small amounts of data to the user, the overhead of the FIFO-based controller (10 μseconds) is less than half that of the best-case descriptor-based controller (24 μseconds for the DEC Ethernet). For larger packets, the balance is in favor of controllers that allow the kernel to perform page-mapping operations, because the copying cost dominates the interrupt-handling cost. The ability to map data into user spaces at low cost is one alternative to kernel-level marshaling with PIO, which retains the benefits of protection and reduced data movement without the need to synthesize code. A typical limitation with FIFO-based controllers, such as the FORE ATM, is that there is no easy way to map the memory in a protected manner simultaneously into multiple user spaces.

Table V. Interrupt-Handling Cost

	Ethernet (DEC)		Ethernet (Sparc)		FDDI (DEC)		ATM (DEC)	
	60	1514	60	1514	60	1514	53	1537
Interrupt Time	13	13	21	21	46	70	5	10
Copy/Mapin Time	11	201	9	39	10	10	5	138
Total	24	214	30	60	56	80	10	148

In contrast, with descriptor-based packet memory, it is possible in principle to support address mapping irrespective of DMA support. However, typically, on descriptor-based PIO controllers, the existence of a small amount of on-board buffer memory makes it difficult to provide address-mapping support, because that memory is a scarce resource that must be managed sparingly. For instance, the DECstation's Ethernet controller has only 128 Kbytes of buffer memory to be used for both send and receive buffers. Mapping pages of the buffer memory into user space would be costly, because the smallest units that can be individually mapped are 4-Kbyte pages. Reducing the number of available buffers this way could lead to delays due to dropped packets during periods of high load. On the other hand, conservatively managing the scarce buffer resource results in the kernel making an extra copy of the data from user space into the packet buffer.

With a trivial amount of controller hardware support, it is possible to solve the protection granularity problem, providing a larger number of individually protected buffers in controller memory. The basic idea is to populate only a fraction of each virtual page that refers to controller memory. As a concrete example, we consider an alternative design to a DEC Ethernet controller.

Figure 1 shows the sketch of the design. Controller memory is organized as 2-Kbyte buffers, each of which will hold an Ethernet packet; this would allow us to have 64 buffers in our 128-Kbyte controller. To allow user processes to write directly to controller memory without sacrificing protection, the controller ignores the high-order bit of the page-offset and concatenates it with the physical page number (PFN) field of each physical address presented by the TLB. This has the effect of causing each 2-Kbyte physical page of controller memory to be doubly mapped into both the top half and the bottom half of a 4-Kbyte process virtual page.

Our experience with host/controller interfaces leads us to believe that while simple FIFO-based controllers are ideally suited for small packets, larger packets would be better served with a more conventional descriptor-based packet buffer. Thus, it might be beneficial to support both forms on the same controllers. To our knowledge, the only controller that has multiple host interfaces on board is the VMP-NAB [19]. Our experiments also suggest that in the absence of memory system support, DMA may incur the cost of additional copies and/or cache-flushing overheads. In such cases, it would be advantageous to use PIO with a descriptor-based packet memory that the host CPU can either copy or map into user space. This could be done either by providing enough buffers or with hardware support (e.g., as mentioned above). The decision to copy or map in will depend on whether the processor is *required* to touch every byte of the packet or not. For instance, if it is

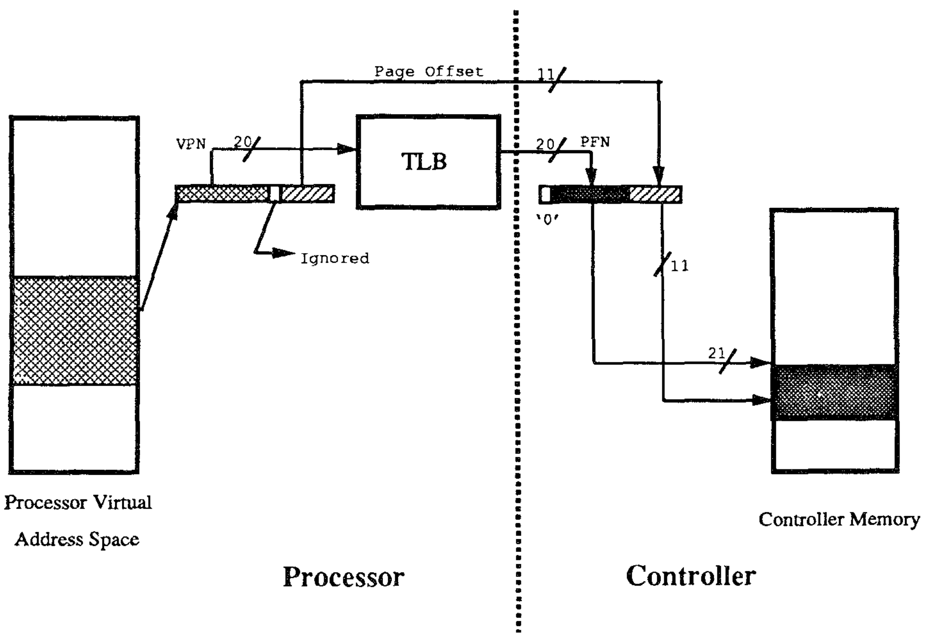

Fig. 1. Address mappıng for Ethernet buffers.

necessary to calculate a software checksum, then an integrated copy and checksum loop (as proposed in [9] and [10]) would suggest that mapping is of limited benefit. This consideration will be even more important with future memory subsystem designs that will provide support for I/O-initiated data transfers. If the processor does need to mediate the transfer, then I/O-initiated data moves would be of benefit if the memory system provides adequate support for cache consistency.

4.3 Network Types

Compared to an Ethernet, a high-speed token ring like FDDI offers greater bandwidth. However, token rings have a latency that increases with the number of stations on the network. Consider a network where the offered load is quite small. That is, on the average, only a few nodes have data to transmit at a given time. As stations are added to the network, the Ethernet latency remains practically constant while the latency of a token ring will increase linearly due to the delay introduced by each node in reinserting tokens. This implies that even on a lightly loaded, moderately-sized token network, achieving low-latency cross-machine communication is difficult. As an example, if each station introduces a one-microsecond token rotation delay, a network of 100 stations would make it infeasible to provide low-latency communication. As load is increased, both the Ethernet and the FDDI token ring will experience greater latencies, with the FDDI reaching an asymptotic value [32]. Thus, on balance, it appears that low-latency communications are not well served by a token ring, despite high bandwidth. We should point out that in our experiments with RPC, token rotation latency *was not* a problem because we used a private ring with two nodes on it. However, if we added more nodes to the ring, we would expect to see a degradation in the latency of RPC.

ATM-style networks that fragment and interleave packets have to incur the delay of fragmenting and reassembly of medium-sized packets. For instance, our experience indicates that with software reassembly, latency begins to be impacted with packet sizes in the range of 1500 bytes. It is not immediately clear how adding fragmentation and reassembly support in hardware or in an on-board processor, as provided in [12] and [31], will affect overall latency, even though fragmentation/reassembly is fast.

5. SUMMARY AND CONCLUSIONS

Modern distributed systems require *both* high throughput and low latency. While faster processors help to improve both throughput and latency, it is high throughput, and not low latency, that has been the target of most newer networks and controllers.

In this paper we have explored avenues for achieving low-latency communications on new-generation networks (specifically, FDDI and ATM). We have implemented a low-latency RPC system using techniques from previous designs in addition to our own. Using newer RISC processors and performance-oriented software structures, our system achieves small-packet, round-trip, user-to-user RPC times of 170 μseconds on ATM, 340–496 μseconds on Ethernet, and 380 μseconds on FDDI. Our RPC system demonstrates that it is possible to build an RPC system whose overhead is only 1.5 times the bare hardware cost of communication for small packets.

Our experiments indicate that controllers play an increasingly crucial role in determining the overall latency in cross-machine communications and can often be the bottleneck. However, we believe that there are alternatives to controller design that can provide lowered latency, facilitating software techniques that achieve excellent performance. Specifically, our experience leads us to believe that FIFO-based network interfaces are well suited for small packets and that DMA- and descriptor-based controllers may have many hidden costs depending on the memory system architecture. Hybrid controllers that provide multiple host interfaces appear to be an attractive alternative to current designs. Of course, the network itself is an important factor for performance. For instance, both of our high-throughput networks have some peculiarities that could affect latency of packets: e.g., the token rotation latency in FDDI network and the fragmentation and reassembly in the ATM network.

Finally, we believe that with careful design at all levels of the communication system, communications latencies can be substantially reduced, enabling entirely new approaches and applications for distributed systems.

ACKNOWLEDGMENTS

The authors thank Ed Lazowska for his numerous patient readings of earlier drafts of this paper and his comments, which added greatly to its clarity of presentation. Thanks are also due to Tom Anderson for his comments and suggestions for improving the paper. Finally, we wish to thank the designers of the FDDI controller from DEC, who gave so willingly of their time in helping us understand many of its details.

REFERENCES

1. Advanced Micro Devices. *Am7990 Local Area Network Controller for Ethernet (LANCE)*. Advanced Micro Devices, Sunnyvale, Calif., 1986.
2. Ross, F. E. FDDI—A tutorial. *IEEE Commun. Mag. 24*, 5 (May 1986), 10–17.
3. Anderson, T. E., Levy, H. M., Bershad, B. N., and Lazowska, E. D. The interaction of architecture and operating system design. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (Apr. 1991).
4. Bershad, B., Anderson, T., Lazowska, E., and Levy, H. Lightweight remote procedure call. *ACM Trans. Comput. Syst. 8*, 1 (Feb. 1990).
5. Birrell, A. D., and Nelson, B. J. Implementing remote procedure calls. *ACM Trans Comput. Syst. 2*, 1 (Feb. 1984), 39–59.
6. Black, D., Rashid, R., Golub, D., Hill, C., and Baron, R. Translation lookaside buffer consistency: A software approach. In *Proceedings of the 3rd ACM Conference on Architectural Support for Programming Languages and Operating Systems* (Apr. 1989). ACM, New York, 113–122.
7. Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M., and Littlefield, R. J. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Dec. 1989). ACM, New York, 147–158.
8. Cheriton, D. R The V kernel: A software base for distributed systems. *IEEE Softw. 1*, 2 (Apr. 1984), 19–42.
9. Clark, D. D., and Tennenhouse, D. L. Architectural considerations for a new generation of protocols. In *Proceedings of the 1990 SIGCOMM Symposium on Communications Architectures and Protocols* (Sept. 1990). ACM, New York, 200–208.
10. Clark, D. D., Jacobson, V., Romkey, J., and Salwen, H. An analysis of TCP processing overhead. *IEEE Commun. Mag. 27*, 6 (June 1989), 23–36.
11. Comer, D., and Griffioen, J. A new design for distributed systems: The remote memory model. In *Proceedings of the Summer 1990 USENIX Conference* (June 1990), 127–135.
12. Davie, B. S. A host-network interface architecture for ATM. In *Proceedings of the 1991 SIGCOMM Symposium on Communications Architectures and Protocols* (Sept. 1991). ACM, New York, 307–315.
13. Digital Equipment Corporation. *TURBOChannel Hardware Specification*. 1991.
14. Digital Equipment Corporation. *PMADD-AA TurboChannel Ethernet Module Functional Specification, Rev. 1.2*. Workstation Systems Engineering, 1990.
15. Digital Equipment Corporation. *Alpha Architecture Reference Manual*, 1992.
16. FORE Systems. *TCA-100 TURBOchannel ATM Computer Interface, User's Manual*. FORE Systems, Pittsburgh, Pa. 1992
17. Hoare, C. A. R. Communicating sequential processes. *Commun ACM 21*, 8 (Aug. 1978), 666–677.
18. Johnson, D. B., and Zwaenepoel, W. The Peregrine high performance RPC system. Tech. Rep. COMP TR91-152, Dept. of Computer Science, Rice Univ., 1991.
19. Kanakia, H., and Cheriton, D. R. The VMP network adapter board (NAB): High-performance network communications for multiprocessors. In *Proceedings of the 1988 SIGCOMM Symposium on Communications Architectures and Protocols* (Aug. 1988). ACM, New York, 175–187.
20. Keppel, D. A portable interface for on-the-fly instruction space modification. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (Apr. 1991). ACM, New York, 86–95.
21. Li, K., and Hudak, P. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst. 7*, 4 (Nov. 1989), 321–359.

22. MASSALIN, H., AND PU, C. Threads and input/output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Dec. 1989). ACM, New York, 191–201.
23. METCALFE, R. M., AND BOGGS, D. R. Ethernet: Distributed packet switching for local computer networks. *Commun. ACM 19*, 7 (July 1976), 395–404.
24. MOGUL, J. C., AND BORG, A. The effect of context switches on cache performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (Apr. 1991). ACM, New York, 77–84.
25. MULLENDER, S. J., AND TANENBAUM, A. S. The design of a capability-based operating system. *Comput. J. 29*, 4 (1986), 289–299.
26. OUSTERHOUT, J. K. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference* (June 1990), 247–256.
27. SCHROEDER, M. D., AND BURROWS, M. Performance of Firefly RPC. *ACM Trans. Comput. Syst. 8*, 1 (Feb. 1990), 1–17.
28. SPEC. Newsletter benchmark results. Systems Performance Evaluation Cooperative, 1990.
29. SUN MICROSYSTEMS, INC. *SBus Specification B.0.* Sun Microsystems, Inc., Mountain View, Calif., 1990.
30. THACKER, C. P., STEWART, L. C., AND SATTERTHWAITE, E. H., JR. Firefly: A multiprocessor workstation. *IEEE Trans. Comput. 37*, 8 (Aug. 1988), 909–920.
31. BRENDAN, C., TRAW, S., AND SMITH, J. M. A high-performance host interface for ATM networks. In *Proceedings of the 1991 SIGCOMM Symposium on Communications Architectures and Protocols* (Sept. 1991). ACM, New York, 317–325.
32. ULM, J. N. A timed-token ring local area network and its performance characteristics. In *Proceedings of the 7th IEEE Conference on Local Computer Networks* (Feb. 1982). IEEE, New York, 50–56.
33. VAN RENESSE, R., VAN STAVEREN, H., AND TANENBAUM, A. S. The performance of the Amoeba distributed operating system. *Softw.—Pract. Exp. 19*, 3 (Mar. 1989), 223–234.
34. MINZER, S. E. Broadband ISDN and asynchronous transfer mode (ATM). *IEEE Commun. Mag. 27*, 9 (Sept. 1989), 17–24, 57.

Received July 1991; revised November 1992; accepted January 1993