Efficient User-Space Protocol Implementations with QoS Guarantees Using Real-Time Upcalls

R. Gopalakrishnan and Gurudatta M. Parulkar, *Member, IEEE*

Abstract—**Two important requirements for protocol implementations to be able to provide quality of service (QoS) guarantees within the endsystem are: 1) efficient processor scheduling for application and protocol processing and 2) efficient mechanisms for data movement. Scheduling is needed to guarantee that the application and protocol tasks involved in processing each stream execute in a timely manner and obtain their required share of the CPU. We have designed and implemented an operating system (OS) mechanism called the real-time upcall (RTU) to provide such guarantees to applications. The RTU mechanism provides a simple real-time concurrency model and has minimal overheads for concurrency control and context switching compared to thread-based approaches. To demonstrate its efficacy, we have built RTU-based transmission control protocol (TCP) and user datagram protocol (UDP) protocol implementations that combine high efficiency with guaranteed performance. For efficient data movement, we have implemented a number of techniques such as: 1) direct movement of data between the application and the network adapter; 2) batching of input–output (I/O) operations to reduce context switches; and 3) header-data splitting at the receiver to keep bulk data page aligned. Our RTU-based user-space TCP/Internet protocol (TCP/IP) implementation provides bandwidth guarantees for bulk data connections even with real-time and "best-effort" load competing for CPU on the endsystem. Maximum achievable throughput is higher than the NetBSD kernel implementation due to efficient data movement. Sporadic and small messages with low delay requirements are also supported using *reactive* RTU's that are scheduled with very low delay. We believe that ours is the first solution that combines good data path performance with application-level bandwidth and delay guarantees for standard protocols and OS's.**

Index Terms—**Multimedia communication, networks, operating system kernels, processor scheduling, protocols, real-time systems, transport protocols.**

I. Introduction

THERE IS a growing need to provide support for multimedia processing within computer operating systems (OS's). This will enable a variety of exciting applications, such as interactive video, customized news services, virtual shopping malls, and many others. A large fraction of data handled by these applications will be of the continuous media (CM) type. The transfer of CM data over the network and its processing at the endsystem must be in such a way that its periodic (or "real-time") nature is preserved. Emerging networks such as asynchronous transfer mode (ATM) and the proposed integrated services Internet [9] with reservation protocols such as RSVP [42] can provide guarantees for data transfer by managing network resources appropriately. Similarly, the OS must manage endsystem resources so that processing needs implied by the bandwidth and delay requirements of each connection are satisfied. This will complement the guarantees provided by the network for data transfer and build upon the increasing processing power of the endsystem hardware. Thus, multimedia applications are commonly referred to as having *quality-of-service* (QoS) requirements. To support these applications networks and endsystems must provide *QoS guarantees* for the transfer and processing of multimedia streams.

This paper reports on our experiences in implementing high-speed protocol processing with QoS guarantees. Specifically, we describe our user-space implementation of the transmission control protocol/Internet protocol (TCP/IP) suite that provides throughput and delay guarantees to applications. Our implementation combines the following two aspects:

- a novel mechanism called the real-time upcall (RTU) that serves as a vehicle of concurrency for implementing protocols in user space and provides processing guarantees;
- state-of-the-art mechanisms for efficient data movement and network input–output (I/O) based on our implementation of a user–kernel shared-memory facility.

The important benefits of RTU mechanism are its ability to closely match stream requirements, improvement in runtime efficiency, and simplification of protocol code. The shared-memory facility allows data to be moved directly between application buffers and the network adapter, thereby reducing data copying overheads. The focus of this paper is on the QoS guarantee aspect. In the next section we summarize the key features of the RTU mechanism that make it a suitable mechanism to implement protocol processing with QoS guarantees. In addition, we briefly touch upon our data movement mechanism and highlight the important experimental results that we have obtained.

A. Summary of Main Ideas and Results

1) The RTU Mechanism: It is a well-recognized fact that the upcall mechanism allows efficient protocol implementations [7]. Our experience suggests that upcalls can also be a suitable vehicle of concurrency to implement protocol operations such as output, input, and timer processing. An RTU is similar to an upcall with the additional feature that an RTU handler function gets a guaranteed share of the

Manuscript received July 11, 1997; revised November 12, 1997; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor S. Keshav.

R. Gopalakrishnan is with AT&T Laboratories—Research, Florham Park, NJ 07932 USA (e-mail: gopal@dworkin.wustl.edu).

G. M. Parulkar is with the Applied Research Laboratory, Washington University, St. Louis, MO 63130 USA (e-mail: guru@arl.wustl.edu).

Publisher Item Identifier S 1063-6692(98)05904-4.

central processing unit (CPU) over periodic intervals in time. Therefore, structuring protocol (and application) code using RTU's provides isolation from other "best-effort" processing load on the endsystems.

Executing protocol code at a high priority (as is done in kernel implementations) does not solve the problem of endsystem QoS guarantees. Instead, both protocol and application code for a connection must be scheduled so that network data can be processed at the required rate. The RTU mechanism can partition aggregate CPU bandwidth among connections to meet the above requirement. Scheduling is important even when endsystems are faster (relative to the network) if there is a need to operate at high utilization. For example, a web server provider could support more clients if the server utilization could be increased while maintaining QoS guarantees.

To provide low delay, early demultiplexing (as in fast-remote-procedure-call (RPC) systems [34]) is important, but not sufficient. It is also necessary to schedule the protocol and application code that processes a message with low latency. The RTU mechanism provides low-latency user-level handler invocation for delay sensitive applications.

The main motivation for the RTU mechanism is to obtain efficiency improvements over current implementations of user-space protocols that provide QoS guarantees. This improvement is obtained by exploiting the fact that protocol processing is iterative and operates on (usually) fixed-size protocol data units in the following two ways.

1) RTU handlers track their CPU usage in terms of the number of protocol data units (PDU's) processed (which is simple) rather than measuring execution times (which can be cumbersome). This is feasible because protocol processing such as in TCP/IP, for example, is performed in terms of (usually fixed-size) PDU's, each of which requires a fixed amount of CPU time in the common case.
2) RTU's are scheduled using a cooperative scheduling mechanism rather than the preemptive style that is used for real-time threads. A RTU handler executes periodically based on its priority, where each execution consists of a sequence of atomic iterations with a scheduling opportunity at iteration boundaries. This is a blend of the event and thread model which lowers both runtime cost and implementation complexity. In particular, unlike threads, RTU's in an address space share a single stack; dispatching an RTU does not require execution state to be saved (and restored) per context switch, and delaying preemption until an iteration boundary simplifies real-time concurrency control. Our analysis in [18] derives scheduling bounds for real-time scheduling algorithms (such as rate-monotonic (RM) and earliest-deadline-first) with delayed preemption.

2) Efficient Data Movement: The importance of efficient data movement is well recognized and several techniques have been described in previous work [11], [14]. Our data movement mechanism is comparable to these state-of-the-art techniques and is designed to work well with our RTU-based user-space protocol implementations. Our implementation is based on a user–kernel shared-memory mechanism and has support for: 1) direct movement of data between application and adapter buffers to avoid data copying; 2) batching of network I/O operations to reduce context switches; and 3) lock-free receive and transmit queues to avoid synchronization overheads. In addition, to enable received data to be remapped rather than copied, the network adapter driver has the ability to extract application data from incoming packets and store it in separate pages. Using these techniques, we have obtained better efficiency than the best kernel resident protocol (KRP) implementations.

3) Important Experimental Results: The motivation behind presenting these results is to show that by combining efficient data movement and scheduling mechanisms, processing bottlenecks in the endsystem can be overcome to provide guaranteed throughput and delay performance for protocols such as TCP/IP. Our choice of TCP/IP to evaluate the QoS guarantees provided by the RTU mechanism was guided by several reasons. TCP/IP has significant computation requirement and complexity to evaluate the efficacy of the RTU concurrency model. We also needed something that is well known to compare against. While real-time transport protocol (RTP) is more likely to be used with QoS-sensitive applications, it was not mature enough when we began this work. Moreover, we are using TCP in an environment where each TCP connection has a reserved bandwidth from the ATM network. Thus, our results are valid in the regime where congestion control mechanisms of TCP are not brought into play.

We briefly state our important experimental results. The experiments were performed on 133-MHz Pentium machines running the NetBSD OS connected to a 155-Mb/s local area ATM network. To aid comparison with existing kernel protocol implementations, we took the existing kernel resident TCP/IP and user datagram protocol/IP (UDP/IP) code and restructured it in user space using the RTU mechanism. Minimal changes to the application code were required to use the RTU based protocol implementation. The important results are the following.

- Our RTU-based TCP/IP implementation delivers a maximum throughput at the application layer of over 120 Mb/s. In comparison the kernel resident TCP implementation delivers at most 80 Mb/s.[1] The lower maximum throughput for kernel TCP is due to the fact that it performs an additional copy from kernel buffers to the application buffer. On the other hand, RTU-based TCP is able to move data directly from the network adapter to application buffers and perform protocol processing "in place."[2] However, increasing throughput by reducing data copying is well known and is mentioned here only to quantify the improvement over the standard implementation and to aid comparison with the case described below.

  In this case we introduce "best-effort" (i.e. non-RTU) processing load on the endsystems. Due to the processing guarantees provided by RTU scheduling, there is no

[1] The maximum throughput using a kernel protocol is 105 Mb/s (when UDP is used).

[2] Existing socket applications must to changed to allocate network buffers in the shared-memory pool.

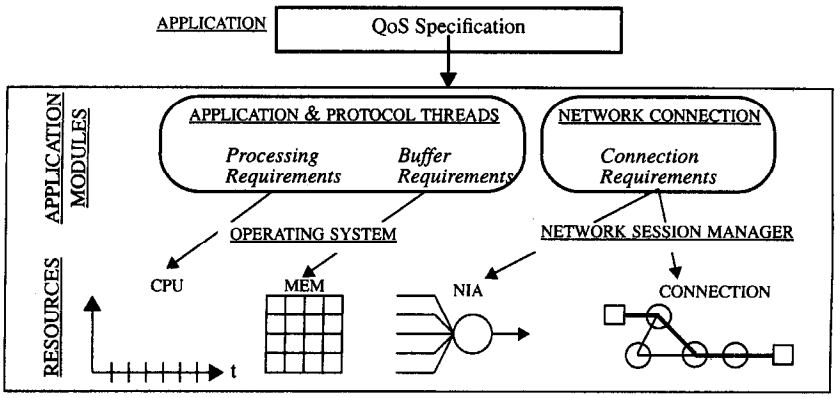

Fig. 1. QoS specification and mapping.

change in the performance of RTU-based TCP connections. Thus, real-time scheduling of RTU's ensures that the amount of CPU allocated for each RTU is independent of the amount of "best-effort" load. In contrast, throughput provided by kernel resident TCP drops from 80 to 30 Mb/s due to the UNIX time-sharing scheduling policy.

- The round-trip time (RTT) for a 4-KB message using TCP implemented with reactive RTU's is 2.3 ms. For kernel TCP, the RTT is 4.3 ms. On unloaded systems, the gains are due to the ability to move data directly between the adapter and user buffers. As before, the benefit of the RTU mechanism is mainly realized when "best-effort" streams are introduced. In this case kernel TCP connections see a fourfold to sixfold increase in RTT with four competing streams. In contrast, the delay performance of RTU-based TCP remains unchanged even with 16 competing streams.

From these experiments, we conclude that our implementation combines guaranteed processing with efficient data movement techniques to provide application to application guarantees. Our main contribution is the RTU mechanism and the protocols implemented using RTU's that support guaranteed-bandwidth as well as low-latency requirements. It must be pointed out that while theories for providing processing guarantees, and methods for efficient protocol processing, are known, we are not aware of any prior work that explores how best to combine these techniques and evaluates the performance for standard protocols such as TCP/IP. Our use of TCP/IP allows performance comparisons with the large installed base of IP implementations. Our performance numbers serve as a benchmark for those that provide QoS guarantees.

4) Paper Outline: The outline of the paper is as follows. Section II gives the QoS framework that we have developed for the endsystem and introduces the periodic processing model. Section III describes how the periodic processing model is implemented using the RTU mechanism, and the RM with delayed preemption (RMDP) scheduling policy that is used to schedule RTU's. Section IV provides a statement of the problems that need to be solved and the outline of our solution. Section V describes the shared-memory facility that is the basis for efficient data movement techniques. Section VI describes highlights of our user-space TCP/IP implementation. Section VII presents important experimental results with RTU-based TCP. Section VIII presents related work and our conclusions.

II. The QoS Framework

We have developed a QoS framework [16] from the point of view of application processes that run on the endsystem. There are four components in our framework.

1) QoS Specification: Specification of QoS requirements is essential to provide performance guarantees. To keep specifications simple, we have identified four application classes that encompass isochronous media, bulk data, low-bandwidth transaction messages, and high-bandwidth message streams. Within each class, we identify quantitative parameters that are easy to specify for the user. For example, a video stream would have QoS parameters such as frame rate and average frame size.

2) QoS Mapping: The QoS specifications mentioned above are at the application level. Since several resources (such as CPU, memory, and network connections) are involved in communication, the specifications must be mapped to resource requirements. The operation of deriving resource requirements from QoS specifications is referred to as QoS mapping and is illustrated in Fig. 1. From the QoS parameters specified for a stream, the mapping operation derives network connection attributes such as bandwidth and cell delay, processing attributes for the protocol code, and memory requirements. We have worked out details of the mapping operation for the resources mentioned above [16].

3) QoS Enforcement: The mapping operation as mentioned above derives resource requirements that are allocated by the OS to each application during the setup phase. During the data transfer phase, the operating system implements the QoS enforcement function, which involves scheduling various shared resources to satisfy these allocations. In particular, the CPU scheduling policy of the OS largely determines how the aggregate processing capacity of the endsystem is shared between different network sessions and is therefore crucial in determining the QoS provided. We therefore focus on CPU scheduling mechanisms for protocol processing that can be implemented efficiently in general purpose OS's.

4) Protocol Implementation Model: Given these solutions for QoS specification, mapping, and enforcement, protocol code has to be structured to take advantage of these facilities.

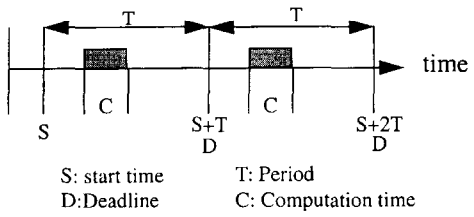

Fig. 2. Periodic processing model.

The protocol implementation model facilitates the mapping of protocol services to appropriate implementation components provided in the framework. An important objective of the model is to provide efficient OS mechanisms to improve the efficiency of protocol implementations by reducing the overhead associated with data movement and context-switching operations that have been shown to dominate protocol processing costs [8].

A. CPU Requirements for Protocol Processing

QoS specification and mapping steps are explained in greater detail in [19]. This paper mainly deals with QoS enforcement and protocol implementation issues. Before we explain how the RTU mechanism works, we briefly describe how CPU requirements are specified. We use the periodic processing model [25] as shown in Fig. 2 to specify processing requirements. In this model, protocol processing for a stream is regarded as a periodic task with period T. In each period, a batch of B PDU's are processed. Depending on the host platform and the amount of processing involved, the computation time C required to process the batch of PDU's is also specified in the model. The periodic processing requirements are typically derived in the QoS mapping step based on the QoS requirements of the stream and knowledge of the CPU requirements for protocol processing. Note that such a periodic model is widely used in the real-time scheduling domain to express processing requirements [38]. An important aspect of our model is that the number of PDU's processed per period, rather than time, is used as a measure of computation requirement. This is based on the assumption that (common case) protocol processing time does not vary much for each PDU. For bulk data transfer, this assumption is generally true since PDU size is constant during the life of the connection, and the time to process a PDU depends mainly on its size. The periodic model is also useful in processing CM data that is periodic in nature.

Adequacy of the Periodic Model: The periodic model has been chosen because it can be easily implemented in an endsystem, is concise, and is amenable to analysis. If the processing requirements of a stream change over time, the period and/or the batch size for the stream can be changed accordingly. Typically, this happens when the rate of the network connection changes or the processing requirement changes. With appropriate support from the QoS infrastructure within the endsystem and the network, RTU parameters can be chosen in such a way that most bandwidth values could be satisfied. This paper does not address these issues, but we believe that the periodic model can cope with these requirements.

III. Implementing the Periodic Processing Model

The existing UNIX scheduling mechanism is inadequate to support the periodic processing model. This is because the UNIX scheduler is designed for time sharing rather than real-time operation. We therefore need the functionality of a real-time thread mechanism provided by OS's such as RT-Mach [38] or Solaris [22]. Using a general purpose real-time thread mechanism to implement the periodic processing model has several drawbacks from the efficiency and implementation complexity standpoint. These are the overheads of multi-threading itself, the cost of real-time concurrency control, and increased context switching due to strict preemption required for real-time scheduling.

Only few OS's directly support the periodic model for real-time processing. Solaris only provides a fixed number of real-time priorities. Almost no OS closely integrates protocol processing with real-time scheduling. Thus, there was a real need to explore alternative abstractions and scheduling schemes that could meet the special needs of protocol processing and exploit its features to achieve efficiency.

A. The RTU Approach

An upcall is a well-known mechanism [7] to structure layered protocol code. Protocol code in a user process can register an upcall with the kernel and associate a handler function with each upcall. The kernel can then arrange to have the handler invoked when an event (such as packet arrival) occurs for the upcall. A real-time upcall also has a period and a computation requirement. These are expressed in terms of milliseconds and number of data units to be processed in each period, respectively (derived by the QoS mapping operation). An RTU has a priority that depends on its period and the scheduling policy used by the RTU scheduler.

The overall organization of the RTU facility is shown in Fig. 3. The RTU facility is layered on top of the normal UNIX process-scheduling mechanism. The layering signifies that runnable RTU's take precedence over runnable UNIX processes.[3] When no runnable RTU's are present, the system reverts to normal process scheduling. Thus, all kernel modifications have been confined to the process dispatcher, thereby requiring no changes to the existing UNIX process-scheduling policy.

A process creates an RTU using the system call application programming interface (API). Multiple RTU's can be created by a process. The create operation returns a *file descriptor* that can be used to perform subsequent operations on the RTU. A pointer to a data structure can also be associated with each RTU during creation. This structure is called the RTU control block (RCB) and is in the address space of the user process. For protocol code, this typically contains the protocol control block (PCB) which encapsulates connection state. When the upcall handler is called, the pointer to its RCB is passed as an argument.

The scheduler creates an RTU only if the admission control operation (schedulability test) succeeds. This is to ensure that

[3]To be fair to normal processes, the admission control policy would not allocate 100% of the CPU to RTU's.

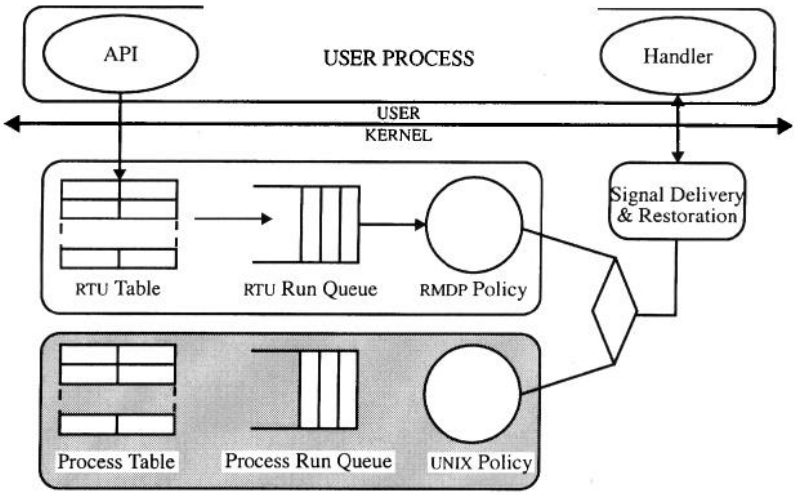

Fig. 3. RTU organization.

each RTU gets its requested time to run in each period [18]. Once created, an RTU can be run using an API call. The scheduler inserts a running RTU into the RTU run queue in each period. The run queue is ordered by RTU priority. When an RTU reaches the head of the run queue, its handler is upcalled. The RTU delivery and restoration routines ensure that the handler function in the process is invoked with the correct arguments, and the system state is restored after the handler returns. Runnable RTU's are scheduled using the RMDP policy which is presented in the following section.

B. The RMDP Scheduling Policy

RTU's in the run queue are scheduled according to a modified RM scheme called RMDP. In both RM and RMDP the priority of a task is inversely proportional to its period. The main difference between the two policies relates to the way preemption is implemented. In the basic RM policy the running task is preempted whenever a higher priority task becomes runnable. In RMDP the scheduler informs the running task (RTU handler) of the arrival of a higher priority RTU by writing into a shared-memory variable. Since RTU handlers are written in an iterative fashion, the running RTU checks the shared variable after each iteration, and yields the CPU if required. Thus, preemption can be delayed for the duration of one iteration. We have derived schedulability tests to determine if a set of RTU's are schedulable [18] with delayed preemption. For most cases, the reduction in the utilization is within a few percent. The preemptive yielding mechanism has very low overhead since it only involves a memory write-and-read operation and does not require interrupts to be turned off. It takes advantage of the iterative nature of protocol processing to achieve almost the same real-time performance of preemptive scheduling algorithms.

In a protocol implementation an iteration corresponds to processing one (or more) PDU's. For example, an RTU that implements the input path of a TCP/IP connection processes received packets in every period. Each iteration in the packet processing "loop" dequeues a packet from the connection input queue, verifies the checksum, checks header fields, and queues data (if any) for the application. The application may also set up an RTU to process the received data at the desired rate.

This scheme has the following advantages: 1) since a handler completes at least one iteration every time it runs, the number of context switches is reduced; 2) Since there is no asynchronous preemption, concurrency control operations such as locking can be avoided; and 3) there is no need to save (and subsequently restore) the runtime stack and register context of an RTU on preemption since the handler function returns and therefore does not need its stack and local variables. In fact, all RTU's in an address space share the same stack. The RTU concurrency model is similar to that of events where each iteration corresponds to an event that is processed to completion. This is a key difference between real-time threads and RTU's.

The periodic processing model is not work conserving since a fixed amount of work is done in each RTU period. However, any leftover CPU bandwidth is allocated to normal processes.[4] It is possible to use the RTU mechanism with a work-conserving scheduling policy as well.

An implementation of the RMDP policy must consider two security issues. It must ensure that an invocation does not run past its stated computation time. This can be achieved using timers similar to the way a typical OS enforces the time quantum. It must also ensure that an RTU yields the CPU when it is requested to do so. Several solutions to this problem have been evaluated in the context of implementing atomic sequences on a uniprocessor using rollforward [30]. These solutions can be directly applied to RMDP since iterations correspond to atomic steps, and a preemption check can be made after rolling forward to an iteration boundary. At present, our implementation assumes cooperative RTU's that yield at iteration boundaries.

C. Reactive RTU's

To support low-delay streams, we have implemented a version of the RTU mechanism called *reactive* RTU's. A reactive RTU does not specify a period or a computation time requirement. Instead, all reactive RTU's are assigned a period of zero and are therefore treated by the RM priority

[4]The existing process scheduling policy determines how the runnable processes share this leftover CPU bandwidth.

scheme as having the highest priority. A reactive RTU is associated with a connection and is made runnable by the network adapter driver in response to packet arrivals on its connection. A reactive RTU can also be made runnable by an application process using a system call. When a reactive RTU is made runnable, its handler is called either immediately or after the running RTU handler (if any) yields. If more than one reactive RTU is runnable, the order of invocation is arbitrary. Reactive RTU's eliminate scheduling delays within the endsystem since they run at the highest priority. Since reactive RTU's are not accounted for in the schedulability test, they should be used only for low-bandwidth sporadic traffic. Techniques such as "limited hijacking" [3] can be used to prevent reactive RTU's from monopolizing the CPU. Our experiments in Section VII show the benefit of using reactive RTU's for latency-critical streams.

Reactive RTU's are different from low-overhead mechanisms such as Firefly [34] or [39]. Firefly uses early demultiplexing to identify the target thread for an incoming RPC. However, it makes no effort to influence the scheduling policy that decides when the thread is to be run. The reactive RTU mechanism ensures early demultiplexing as well as low-latency scheduling. The application specific safe handler (ASH) approach to low latency is to not involve the scheduler at all; thus, the handler is run in the context of the interrupt in a safe manner. The drawback is that the ASH handler can only do a restricted set of operations since it runs in interrupt context.

D. Other Applications of RTU's

The RTU mechanism has been applied to other tasks besides protocol processing. One example is a multimedia-on-demand (MOD) server implementation. The MOD server reads blocks of video data from a disk array, does some processing on each, and sends it over an ATM virtual circuit (VC) to a client. For each outgoing stream, an RTU is used to ensure that the processing requirement for the stream are met. Likewise at the MOD client, an RTU is associated with each incoming stream so that processing guarantees can be provided for MPEG-1 decoding [5]. However, MPEG decoding does not conform to the periodic processing model since the time to process various types of frames varies. In this case applications must yield based on time spent rather than number of frames processed. The use of the RTU mechanism for middleware such as real-time CORBA is currently being explored. There is also an implementation of kernel RTU's in which the RTU handlers are kernel functions as opposed to being in user processes. Kernel RTU's have been used to restructure existing TCP/IP protocol implementation in the kernel.

IV. Problem Statement and Solution Outline

Traditionally, protocols have been implemented in the kernel [24]. We refer to this as the KRP model. The motivating factors for the KRP model are layering considerations, integration with the I/O subsystem, ease of demultiplexing data to user processes, and security. However, service differentiation in the KRP model is problematic since all protocol processing occurs at a single priority level. In addition, protocol processing in the kernel takes precedence over all application processing. This causes priority inversion where protocol processing of low-priority traffic (in kernel space) preempts processing of higher priority traffic (in user space). Therefore, there is a need to provide a guaranteed share of the CPU to both protocol and application code.

An attractive alternative is to implement protocols in user space and use the QoS requirements of a stream to determine the priority at which protocol processing and application processing are performed. We refer to this as the application-level protocol (ALP) model. While previous efforts in this area have looked at issues such as integration with the I/O subsystem [26] or security [37], our focus is on implementing protocols with QoS guarantees while maintaining high efficiency. In our case we use the RTU mechanism to provide processing guarantees for protocols implemented in the ALP model and integrate it with efficient data movement. Appropriate choice of RTU parameters provides service differentiation and avoids priority inversion, both of which are necessary for providing guarantees. The ALP model enables better resource accounting since it allows resources to be allocated and scheduled among application-level entities.

The protocol processing model strongly influences the overheads of data movement and context switching that dominate the cost of protocol processing [8] and, hence, determine its efficiency. For example, in the KRP model, data has to be moved between the application buffers and kernel buffers, and between kernel buffers and the network adapter. Data is typically moved by copying or, in some cases, by page remapping. These operations have different costs in the ALP and KRP models. Another significant source of overhead in protocol processing is context switching and system calls. Context switches occur due to network interrupts, and due to priority-based scheduling of protocol processing tasks. System calls occur when the data path enters the kernel from user space and when control operations during data transfer require kernel intervention. Therefore, overheads due to data movement, context switches, and system calls must be minimized as much as possible.

A. Protocol Processing Overheads

In this section we discuss the specific problems that need to be solved to realize efficient implementations of protocols in user space with QoS guarantees.

1) System Calls for Network I/O: Applications use system calls such as *send* and *recv* to move data between user and kernel domains. In the ALP model the data units that move across the user–kernel boundary are network PDU's. In the KRP model these data units are application buffers. Because each application buffer corresponds to multiple PDU's, the number of system calls required to move the same amount of data is much higher in the ALP model. Thus, it is important to reduce the number of system calls for network I/O in the ALP model.

2) Data Movement: To realize the efficiency benefit of moving data directly between protocol buffers in user space and the network adapter, the following two problems must be addressed: 1) protocol buffers must be located in "wired-

down" memory so that they are not paged out while a direct memory access (DMA) operation is in progress; a buffer management system that takes this into account must be implemented, and 2) virtual-to-physical address conversion is usually required to setup DMA. Since protocol buffers are in user space, this conversion must be done in an efficient manner.

3) Asynchronous Event Processing: Processing of asynchronous network and protocol events such as packet arrivals and timer expirations are usually triggered by interrupts in the KRP model. Using this mechanism in the ALP model will lead to excessive context switching because successive events may be destined for different processes. In addition, interrupt-driven scheduling disrupts priority-based scheduling schemes, leading to priority inversions. Therefore, the problem of handling protocol events without excessive context switching or delay needs to be addressed.

4) Scheduling Concurrent Protocol Activities: Protocol processing for each connection is usually structured as "upper" and "lower" halves. The upper half gets control when the application process initiates read or write operations to a connection. The lower half is driven by network events such as packet arrivals and timer expirations. In current KRP implementations the upper half does concurrency control by preventing the lower half from running by disabling interrupts. This leads to priority inversion and highly variable delays in message processing and delivery. The lower half can also block out the upper half in an uncontrolled manner since it runs at interrupt level leading to "livelock." In the ALP model we can use a concurrency mechanism such as the RTU to implement each half that allows the CPU to be partitioned between the two halves based on their processing requirements. In addition, the delayed preemption feature removes the need for lock-based concurrency control between RTU's that implement the two halves of a connection. While the ALP model does not completely eliminate livelock, it reduces it significantly since the amount of work done at system priority is less.

To summarize, the following problems need to be solved to realize efficient protocol implementations in the ALP model: 1) perform network I/O with the least amount of system calls; 2) move data directly between application buffers and the network adapter (zero-copy operation) for both send and receive; 3) efficiently process asynchronous events such as packet arrivals with minimal priority inversion; and 4) take advantage of the RTU mechanism to realize efficient protocol implementations that provide bandwidth and delay guarantees.

B. Solution Outline

The solutions that we have implemented to address the problems mentioned above are as follows.

- We use control structures in user–kernel shared memory instead of system calls to perform network I/O. The shared memory contains queues of outgoing and incoming PDU's for each connection belonging to the process. These queues are accessed by protocol code that runs in the context of RTU handlers that are bound to a given connection. For example, an RTU that does output processing for a connection creates a data structure for each PDU in shared memory. It then appends it to the output queue (also in shared memory) for the underlying VC. The data structure contains control information such as the buffer address and the length of the PDU data. When the RTU handler gives control to the kernel during scheduling context switches, the adapter driver examines the output queue and initiates a DMA transfer operation to the network adapter. Thus, scheduling and network I/O take place in the same system call. An important feature of our implementation is that these queues are "lock-free" to eliminate the need for synchronization between the user and kernel. Fig. 4 shows two queues for a connection, one for send and the other for receive. These two queues are associated with a VC on the ATM adapter. It also shows a generic control data structure between the process and the adapter driver to queue commands and return status information.
- We use a pool of pages in the user–kernel shared-memory area to store headers and data for both outgoing and incoming PDU's. Shared memory eliminates the need to remap or copy pages between user and kernel domains during data movement. This mechanism provides zero-copy operation for applications that can work with noncontiguous memory buffers. The shared-memory pages are also "wired down" so that the adapter can DMA the data from the shared buffers. In addition, we have implemented the shared-memory pool so that expensive virtual memory (VM) table lookups are not required to map from virtual to physical addresses during DMA operations.
- We use the user–kernel shared memory to also enqueue incoming packets and process them in a batch using a periodically scheduled RTU. Thus, asynchronous events such as packet arrivals are not processed in first-in first-

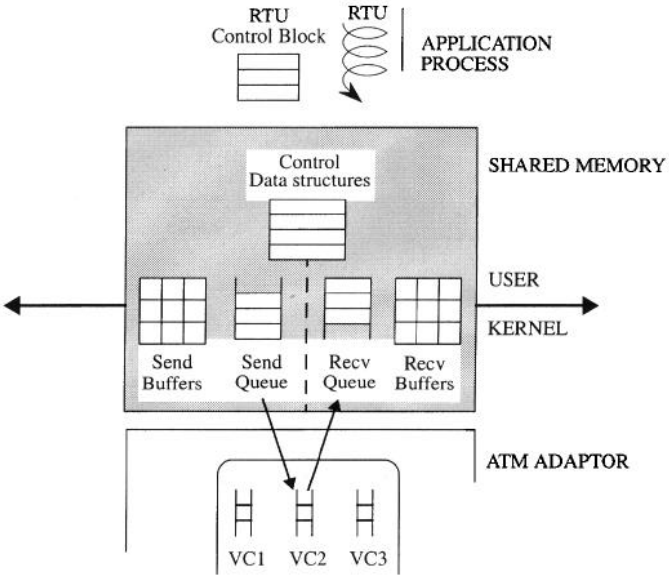

Fig. 4. Using shared-memory buffers.

out (FIFO) order, but in an order determined by the priority of the RTU associated with the connection. This solves the problem of priority inversion. The batching of PDU's also helps reduce the number of context switches.

- By closely integrating scheduling and protocol processing using the RTU mechanism, we have been able to provide guaranteed bandwidth for bulk data transfer, and low delay for request–response traffic. For example, bulk data transfer protocols are implemented using periodic RTU's that get a guaranteed share of the CPU over time. On the other hand, low-latency messages are handled using the reactive RTU mechanism so that user-level handlers are invoked as soon as a message arrives.

V. User–Kernel Shared-Memory Facility

In this section we describe the implementation of the "wired-down" shared-memory facility between a user process and the kernel that is used by user-space protocol implementations to perform network I/O. The shared-memory region is called the communication area (CAR) of a process. The CAR is very similar to the *pool* model used in [13] and to the *communication segments* in U-Net [14] although recent U-Net implementations wire memory pages "on the fly." Unlike these, the CAR does not need any support from the network adapter hardware. In this sense it resembles *comapped* buffers used in [40] and is used in a similar manner. The CAR is of fixed size and is allocated in a contiguous portion of the process address space. The CAR is also mapped to a region of the kernel address space. The pages in the CAR are wired down so that the kernel can setup DMA between the network adapter and memory pages of the CAR in the adapter interrupt service routine. All connections within a process share the pages in the CAR. The pages in the CAR are partitioned into transmit and receive portions. The adapter driver manages free buffers of the CAR for receive, and the user process manages the free buffers for transmit. Thus, the process and the driver can manipulate the free lists without having to make system calls to obtain locks. Fig. 5 shows two processes with their CAR's that are also mapped to kernel address space. The *management page* of the CAR contains information about transmit and receive queues for each VC owned by the process. The VM system secures the CAR of a process from other processes. In addition, a process can only modify those pages of the kernel address space that are part of its CAR.

Most of the TCP/IP protocol implementations use the *Mbuf* facility [24]. To use existing TCP/IP code with minimal modifications, we ported the *Mbuf* facility to run in user space. The *Mbuf* routines use the memory in the CAR to allocate the buffers used by *Mbufs*. Accordingly, pages in the CAR are allocated among normal *Mbufs* and *cluster Mbufs* [24] where a cluster is equal to a page size (4 KB). The use of *Mbufs* does not necessarily impose an overhead of copying data from the *Mbufs* to contiguous application buffers. For example, since the driver can place data in pages, these pages can be remapped to appear contiguous in the application's address space. In addition, some high-performance applications may directly use data contained in *Mbufs*.

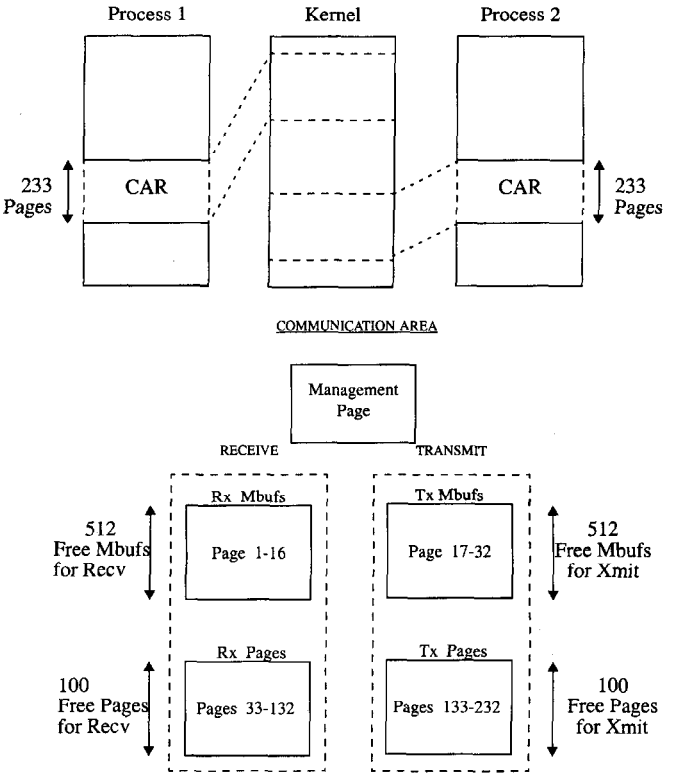

Fig. 5. Communication area.

1) Tradeoffs with Wired Pages: Wiring down pages is necessary to move data directly between adapter and user space, but it comes at the cost of potentially lowering performance of other application processes. We do not consider this to be a problem in our 32-MB machines. It must be noted that the size of transmit area must be at least as big as the bandwidth delay product since protocols such as TCP free transmitted data only after receiving an acknowledgment (ACK). The receive area size for a connection is a function of the connection bandwidth, and the maximum duration before the application consumes the data. If the application processing is also performed using RTU's, received buffers can be returned back to the free pool within a bounded time. Thus, the memory requirements can be bounded and can be calculated during connection setup.

In general, a shared-memory facility has implications for security, amount of pinned memory, amount of copying, and the semantics of send and receive [4]. In our implementation memory is shared pairwise between a process and the kernel only. It is therefore fully secure. According to the taxonomy in [4], our implementation is categorized as *system-allocated* buffers with *move* semantics. In [33] a contiguous range of shared memory is used, but it is partitioned between different processes and these partitions are disjoint. Since there is no sharing between partitions, it is equivalent to our approach of pairwise sharing in terms of security and amount of memory required. If a single pool of pinned memory is to be statistically shared among processes to reduce overall memory requirement, then either a remapping or a copy operation is required. It also has potentially lower security since pages can move between processes. However, the shared-memory

approach can still save memory compared to kernel protocol implementations, since in the latter case buffers must be allocated both in the application and the kernel (such as *Mbufs*). The kernel protocol approach also incurs an extra copy.

2) Lock-Free Buffer Operations: All *Mbuf* network I/O and memory management operations that require synchronization between the user protocol and the kernel are implemented using lock-free data structures [11]. The basic data structure is a one-reader-one-writer FIFO queue to pass buffers between the user protocol and the kernel. The *management page* has two such queues—one used by the ATM driver to return transmit *Mbufs* to the CAR after they have been transmitted (*txdone_q*), and the other for the RTU's to return receive *Mbufs* to the driver after they have been consumed by the application (*rxdone_q*). For each virtual circuit identifier (VCI), the *management page* also has a *tx_q* that contains PDU's for transmission, and an *rx_q* that contains the received PDU's. A queue consists of an array with a *read* pointer and a *write* pointer that are indices into this array. The operation of the queue is explained in [11]. Up to six VC's can be allocated in a process. More VC's can be supported by increasing the size of the CAR *management page*. One important feature of our implementation is that we store indices of *Mbufs* rather than their addresses in the array. Since the CAR is contiguous in memory, these indices can be converted into either user addresses or kernel addresses without the need for any VM operations.

An important advantage of RTU scheduling is that there is no need for synchronization among RTU's in a process for updating queues that are shared among all connections. These include the *txdone_q*, *rxdone_q*, and the free list of transmit *Mbufs* and cluster *Mbuf* pages. Thus, the use of lock-free queues eliminates locking between user and kernel, and the cooperative scheduling implemented by RMDP eliminates locking between RTU's that share CAR resources.

3) Data Movement Without Copying and System Calls: Using the user-level *Mbuf* facility, data movement can occur as part of the scheduling system call, and without cross domain copying or VM operations. We consider a send operation on application data contained in an *Mbuf* chain.[5] The protocol code appends other control information needed to create a packet. It then places the index of the head of the chain in the send queue corresponding to its VCI. Once a batch of packets have been placed in the queue, the sending RTU yields the CPU by making a system call. For each *Mbuf* index in the queue, the kernel converts it into the corresponding kernel virtual address using simple pointer arithmetic. The resulting *Mbuf* chain is then enqueued at the adapter for transmission. The transmission completion interrupt returns the *Mbuf* chain in the *txdone_q*. It should be clear from the description that the send operation does not require any copy operations.

In the receive direction, essentially a similar operation is performed except that the ATM driver has the ability to separate headers from data. The header size (typically 52 for TCP/IP with the TCP timestamp option) is looked up based on the incoming VCI. If the data portion is a multiple of the page size, it is placed in pages using DMA and is associated with a *cluster Mbuf*. Again, no data copying is required. Received data can be made contiguous in the application's address space using remapping.

From the above description, we can see that the use of shared memory along with batching eliminates system calls to initiate data movement in both the transmit and receive directions. In addition, data movement and scheduling can occur in a single context switch. Our use of indices rather than addresses also obviates the need for VM operations in the critical path. Finally, we make an observation regarding *zero-copy* operation. Since the ATM adapter supports cache-coherent DMA into *Mbufs*, *zero-copy* operation comes for free if the application can accept data in the *Mbuf* format. If there is a need to place data in contiguous memory, then we must either incur one extra copy or remap data pages. If page-size segment sizes and header-data separation are used, remapping can be done without any further copying.

VI. Protocol Implementation Using RTU's—TCP/IP Example

In this section we illustrate how protocols are structured using RTU's, taking the example of our RTU-based TCP implementation. It incorporates all of the techniques for efficient data movement introduced in the previous section. We took the existing NetBSD kernel code for TCP/IP and ported it as a user-space library. This section focuses on the TCP data path implemented by RTU's. The connection setup mechanism maps each TCP connection to an ATM VC with the required QoS parameters and sets up the adapter to pace data out at the negotiated rate. Other details of connection setup may be found in [15]. In the data transfer phase each concurrent task in a protocol is typically implemented as an RTU. For TCP, three RTU's are created as part of connection setup—one each for output, input, and timer processing. Periodic RTU's are used for guaranteed-bandwidth connections, and reactive RTU's are used for low-latency connections. We describe the periodic RTU case below.

The TCP/IP input and output paths are depicted in Fig. 6. The application layer reads and writes data using a *socket*-layer [24] interface. The socket layer mainly implements buffering functions. The TCP and IP layers implement the transport- and network-layer functions. The link layer communicates with the network adapter driver to send and receive data over VC's.

1) TCP Input Processing: In the input direction the input RTU handler *ipintr* processes IP datagrams enqueued by the network adapter driver in the VC queue for the connection. Although the figure shows the input RTU in the IP layer, TCP and socket-layer input processing also occurs in the context of this RTU. The computation requirements of the input RTU depend on the throughput requested—typical values for the period are 10 ms, and the batch size can be up to 20 PDU's (with size up to 8 KB). In each invocation *ipintr* dequeues an IP packet from its VC queue and does IP processing. It then calls the *tcp_input* function that does the TCP header

[5] Typically, the sender allocates send buffers from its CAR to hold its data. These buffers can be converted to an *Mbuf* chain at the protocol interface without requiring copying.

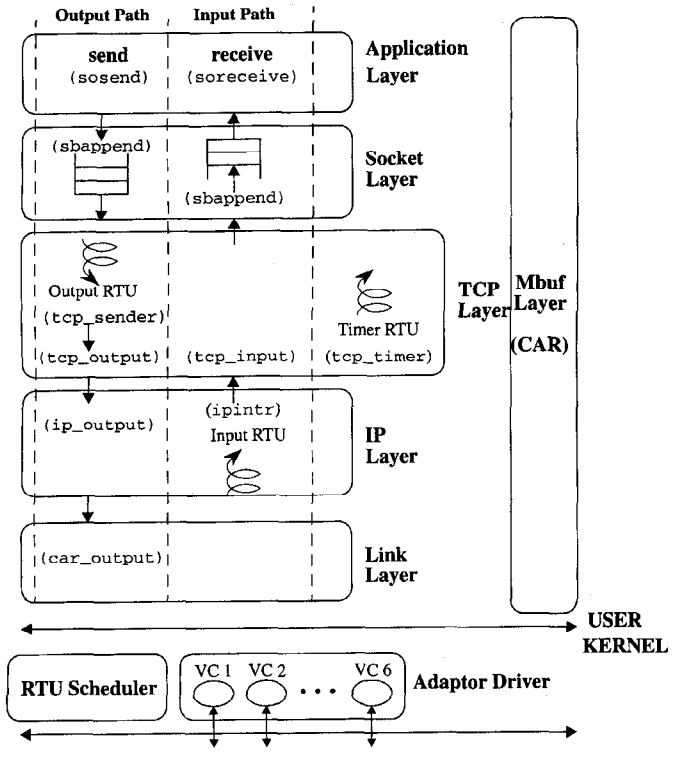

Fig. 6. RTU-based TCP/IP organization.

processing and checksum computation. The data (in the form of an *Mbuf* chain) are then enqueued at the *socket* queue to be consumed by the application. The *ipintr* function checks for yield requests after each packet is processed. If the check is positive, the RTU *yields* (*ipintr* returns). Control is transferred to the RTU scheduler, which reassigns the CPU to a higher priority RTU. The RTU that yields will be resumed later (*ipintr* will be upcalled again) and it can process the remaining PDU's for the current period. An invocation completes when a batch of PDU's is processed or when the input queue is empty. This behavior repeats in each period until the input RTU exits (typically when the TCP connection is closed).

2) TCP Output Processing: The description is similar to the input case. In the output direction the RTU handler looks at the send queue in the socket structure and calls the *tcp_output* function with the number of segments to send in the batch.[6] This calls *ip_output*, which then enqueues each PDU for transmission on the VC output queue. Unlike kernel TCP, output handlers are triggered by the RTU scheduler and not by packet arrivals. Yield requests are handled as before.

3) TCP Timer Processing: The timer processing RTU is upcalled every 100 ms. It calls the functions required to process the six TCP timers as in the BSD TCP implementation [36]. Whereas in the kernel the timer functions process all TCP connections, in our case they process only a single TCP connection. While this is more accurate in terms of timing, it has the overhead of a context switch to process timers for each TCP connection. However, it is possible to eliminate this extra context switch by doing timer processing as part of the output or input RTU invocation.

For guaranteed-bandwidth streams, the input and output RTU's are periodic. Thus, they are invoked periodically by the kernel and process data in the VC and socket queues. For low-delay TCP streams, both of these RTU's are reactive. The input RTU is called on packet arrival and the output RTU is invoked using a system call.

4) Comparison to Kernel TCP Implementation: When packet processing is triggered by periodic RTU invocations as opposed to packet arrivals (as in typical kernel protocols), there potentially can be half a period of delay on average before the packet is processed by TCP. Thus, ACK generation is also delayed by this amount, causing an increase in RTT. For streams that require throughput guarantees, this is not a problem since the periodic RTU's ensure that packets will be processed at the negotiated rate. As far as the RTT estimate at the sender is concerned, the effect is again minimal since standard kernel TCP implementations use 500-ms timers to estimate RTT, whereas typical RTU periods are only a few tens of milliseconds. Another effect of the increase in RTT is a moderate increase in buffer requirements at the sender since a packet buffer can be freed only after its ACK is received. One can limit the impact of this by choosing smaller RTU periods for high-bandwidth connections. Overall, memory requirements for our implementation are comparable to kernel implementations since separate buffering is not required for application and protocol data. Scheduling packet processing at the priority of the receiver is an approach that is taken in other works such as "lazy" receiver processing [12] and has benefits such as overload control and service guarantees.

VII. Experiments and Results

The reported set of experiments bring out the overall performance of our implementation. Thus, we do not present results that measure the contribution of individual optimizations. Wherever appropriate, we identify the main reason for the observed performance improvement. Our experimental setup is as follows. The hosts are a pair of 133-MHz Intel Pentium-based machines. Each host is equipped with a 155-Mb/s ATM interface card from Efficient Networks, Inc. (ENI) to connect to a Baynetworks ATM switch port. Several permanent virtual circuits (PVC's) are set up between the hosts through the switch. The OS used is NetBSD, with our additions to the kernel that include the RTU mechanism, the "wired-down" shared-memory facility, and modifications to the network adapter driver. Table I summarizes the experiments and states their main conclusions. Only the starred experiments are described in this paper. A more comprehensive account may be obtained from [15].

A. Multiple TCP Connections

In this experiment we have six TCP connections active simultaneously between the two hosts. In one case we had six processes, each with a RTU-based TCP connection, and in the other case we ran six copies of the `ttcp` program that uses the regular kernel TCP. Fig. 7(a) is a scatter plot that

[6]The number of packets actually sent depends on the state of the TCP connection and is decided by the *tcp_output function*.

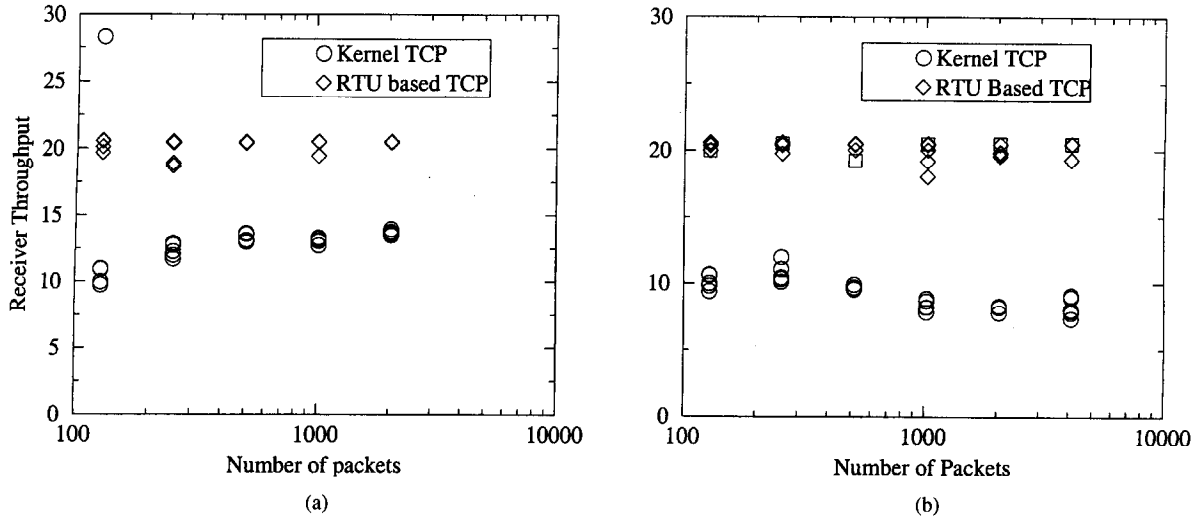

Fig. 7. RTU-based TCP performance. (a) Throughput sharing (no competition for CPU). (b) Throughput sharing (with competing processes).

TABLE I
EXPERIMENTAL RESULTS AT A GLANCE

Expt.	Brief Description	Comments
	Guaranteed Bandwidth Class	
1	Compares throughput of kernel and user TCP (single connection case)	RTU based TCP delivers higher throughput than kernel TCP. It also provides bandwidth guarantees whereas kernel TCP does not.
2*	Compares throughput of kernel and user TCP (multiple connections case)	No reduction in aggregate bandwidth of RTU based TCP connections; Bandwidth guarantees are provided per connection. Kernel TCP does not provide bandwidth guarantees.
3	Coexistence of guaranteed bandwidth and best effort connections	User space TCP connections get guaranteed bandwidth; remaining bandwidth obtained by best effort connection.
	Low Delay Class	
4*	Compares round trip times for user and kernel TCP (single connection)	User space TCP has smaller delays compared to kernel TCP. In addition user space TCP delays do not increase when best effort load is introduced. In contrast, kernel TCP delay increases by an order of magnitude.
5	Effect of guaranteed bandwidth streams on low delay streams	Delay increases with increasing preemption delay introduced by the guaranteed bandwidth streams. However the increase is bounded and predictable.
	Coexistence of Different Classes	
6*	Coexistence of multiple stream types	Both guaranteed bandwidth streams and low delay streams coexist in a manner that their individual QoS requirements are satisfied.
	Loss Performance of TCP/IP	
7	Throughput Performance of TCP with packet loss in the network	Using a combination of periodic and reactive RTUs graceful degradation in performance is obtained with increasing packet loss.

shows the throughput measured at each of the six receiving processes in the two cases when there is no competition for the CPU from other processes. The x-axis is the number of packets sent. Socket buffer sizes for each connection are 200 KB. In fact, `ttcp` throughput tops out at socket buffer sizes in the 100-KB range. In this case we see that the throughput seen by each RTU-based TCP connection is 20 Mb/s giving an aggregate of 120 Mb/s. The aggregate throughput of the `ttcp` processes is only 80 Mb/s. This difference in performance is solely due to eliminating the extra copy operation that kernel TCP does to move data from user to kernel buffers.

To demonstrate the benefit of protocol processing using RTU's, we repeat the same experiment but with four competing processes on both machines that do prime number calculation. This case is shown in Fig. 7(b), where the aggregate throughput for `ttcp` falls to 50 Mb/s. In contrast, there is no change in the throughput of RTU-based TCP connections. This experiment shows that RTU-based TCP provides bandwidth guarantees to connections in different processes even when there is competition for the CPU. More importantly, the extra work involved in the RTU scheduling mechanism does not degrade data path performance.

The experiment also reveals that to obtain guaranteed throughput for a TCP connection, the application-level processing of received data must also be guaranteed. Otherwise, due to the window-based flow control mechanism of TCP, the sender will run out of window space and data transfer will stall. In our experiment the receiving application function simply frees the data buffers without any additional processing. Thus, the TCP input RTU handler is able to call the application function directly, thereby ensuring guaranteed processing for the entire data path. In general, the application function can be called either from the input RTU or from a separate application-level RTU that only does application processing. This reiterates the importance of using the RTU to provide application-level guarantees. It must be noted that we have not changed the TCP/IP control mechanisms in any way. For example, packet loss triggers conventional TCP congestion control mechanisms such as slow-start. This can cause a short-term reduction in throughput until the congestion window opens up. Thus, the throughput guarantee will not be maintained during loss periods.

B. TCP Delay Performance

This experiment measures the delay performance of RTU-based TCP. It consists of a sender program that sends a message to a receiver on the remote host, which reflects the message back to the sender. The RTT for the message is measured at the sender. This experiment is performed on unloaded machines and then repeated with a different number of background streams at the sending and receiving hosts. In one case the sender and receiver programs use RTU-based TCP, and in the other case they use kernel TCP. Fig. 8 shows the measured RTT's for different message sizes. The left half is for the RTU-based TCP case. We see that

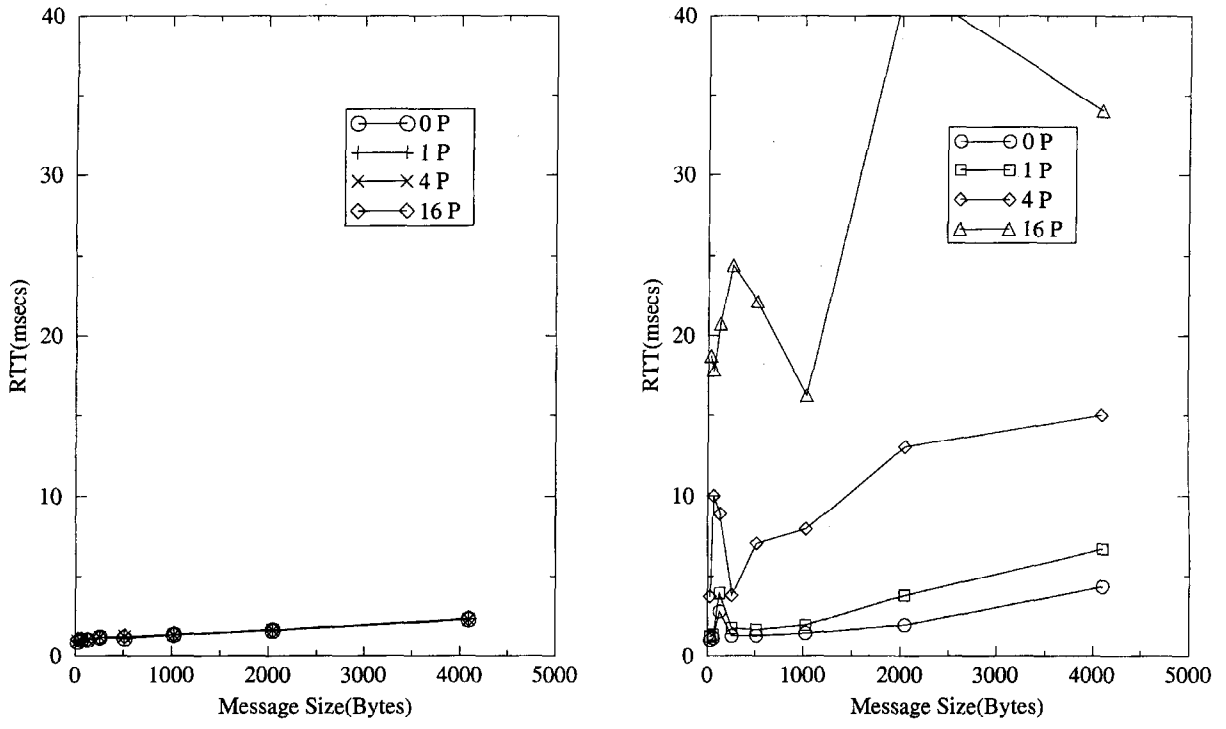

Fig. 8. Delay performance of (a) RTU-based TCP and (b) kernel TCP.

the effect of best-effort streams on the delay experienced by RTU-based TCP is negligible and is not noticeable in the plot. The maximum delay for 4-KB message size is 2.3 ms for the unloaded case and 2.4 ms with 16 "best-effort" streams. In contrast, the corresponding figures for kernel TCP are 4.3 and 34 ms, respectively. These numbers remain consistent over several runs with very little variance. In the unloaded case the extra delay in the case of kernel TCP is due to the extra copy. However, for the loaded case, the increase in delay is entirely due to scheduling delays introduced by the OS scheduling policy. Thus, the use of reactive RTU's eliminates scheduling delays caused due to contention by "best-effort" streams. The main conclusion of this experiment is that the use of reactive RTU's provides delay guarantees on message transfer to the application. We observe that the delay due to process scheduling dominates the time taken for protocol processing. Thus, for good delay performance, it is more beneficial to provide real-time scheduling support rather than to reduce protocol processing overheads such as additional copying. We also observe that the improvement in delay is not due to the fact that we do early demultiplexing into separate IP queues. Having separate queues in itself does not imply low delays unless packets are processed in priority order using mechanisms such as RTU's. It must be noted that recent work on low-latency message handling does not deal with methods to reduce scheduling delays. For example, [39] describes an ASH mechanism that allows message handling code to be loaded in the kernel to avoid changing the process scheduler. This trades off flexibility for reduced latency. Another effort studies the effect of code structure on latency [31] but does not consider scheduling delays.

TABLE II
STREAM PARAMETERS AND MEASUREMENTS FOR EXPERIMENT 6

Stream ID	Stream Type	Period	Batchsize	Size	Specified Bandwidth	Measured Throughput
1	TCP-1	20 msecs	12	8KB	39.3	38.7
2	TCP-2	20 msecs	12	8KB	39.3	39.2
3	UDP-1	40 msecs	16	4KB	13.1	13.1
4	UDP-2	40 msecs	16	4KB	13.1	13.1
5	TCP-3	50 msecs	12	8KB	15.7	15.6
6	UDP-3	reactive	N/A	varies	N/A	N/A

C. Coexistence of Multiple Stream Types

In this experiment we combine bandwidth-guaranteed streams and low-delay streams and measure how they affect each other. We set up six streams with the attributes shown in Table II. The first five are bandwidth-intensive streams with an aggregate bandwidth of 120 Mb/s. The experiment consists of running the six streams simultaneously. The throughput seen by the first five streams is measured. For stream six, the message size is varied from 32 to 4096 B and the RTT is measured for each size. The measured throughput for the first five streams is shown in Table II. It can be seen that the measured and requested values are very close.

Fig. 9 plots the RTT values for different message sizes. The lower curve is the RTT value when the first five streams are not present. The upper curve is the measured RTT with the bandwidth intensive streams present. We notice that the RTT increases at most by 1.2 ms. This increase is due to preemption delays, interrupt overheads, and the fact that the network adapter uses a FIFO DMA channel to transfer reassembled packets to the host. Nevertheless, the increase is not substantial and is bounded. In contrast, the delays experienced by kernel TCP (not shown in the plot) were an order of magnitude larger.

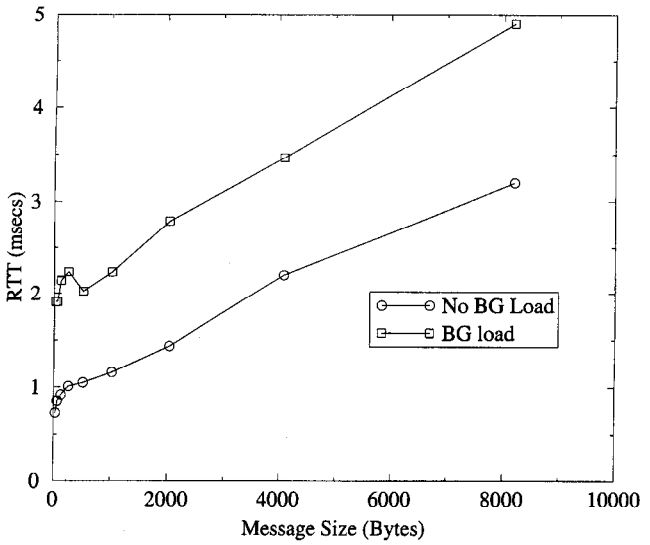

Fig. 9. RTT of low-delay stream.

We conclude that the RTU-based protocol implementations can concurrently support guaranteed-bandwidth and low-delay streams. This is mainly because of the close integration of scheduling and protocol processing. Thus, guaranteed-bandwidth streams use periodic RTU's that ensure a certain share of the CPU, and low-delay streams use the reactive RTU's that guarantee response time. In addition, the scheduling mechanisms required to enforce the QoS for each stream does not diminish absolute performance.

VIII. Related Work

While there are several studies that measure performance of user-space TCP/IP implementations [26], [11], 13], [14], they do not consider the dimension of QoS guarantees. This paper provides the first set of TCP/IP throughput and delay benchmarks with a focus on the QoS guarantee aspect.

We begin with some recent work in scheduling support to provide QoS guarantees for application-layer protocols for audio and video streaming. While these streams have strict timing requirements, the bandwidth requirement and transport-layer processing requirement for these streams is modest. A user-space protocol implementation model is advocated in [40] and its implementation in Solaris is described. They implement a user–kernel shared-memory facility for efficient data movement. Solaris real-time threads are used to do application and protocol processing. The results show that the scheduling support meets the delay and jitter requirements of multimedia streams. However, the performance gains of efficient data movement and scheduling have not been evaluated in comparison to the existing Solaris implementation. A user-space protocol approach is also used in the RT-Mach in [23] OS to provide QoS guarantees. A processor reserve mechanism has been implemented to schedule the CPU among different protocol sessions. A video phone application is implemented based on this architecture and it is shown that the delay jitter for audio and video streams is lower than in the case where protocol processing is done by a central protocol server. However, the bandwidth requirement for this application was only 1.4 Mb/s, and more work is needed to evaluate the performance with multiple streams, higher rates, and TCP-like protocols.

We now review user-space protocol implementations [26], [37], [13], [14] that mainly focus on efficient data movement. Most of them share common features such as using a connection server for connection setup and teardown, avoiding copying of data by using memory "pools" to locate network data, and demultiplexing packets at the adapter level. Most previous efforts have focused on the TCP/IP protocol stack. The first implementation was for the Mach OS using Mach threads for concurrency and Mach interprocess communication (IPC) for data movement [37]. This implementation was shown to perform better than the standard TCP implementation in Mach. However, it did not perform as well as a monolithic kernel implementation. We expect the performance to be even lower if the overhead of real-time scheduling and real-time concurrency control were factored into the above implementation. Implementations of TCP in HP-UX [13] and of U-Net in SunOS [14] are representative user-space implementations in UNIX. The HP implementation used real-time priorities to speedup protocol processing, although it was not intended to provide bandwidth guarantees. U-Net does not deal with concurrency issues in protocol processing. For instance, it focuses on tuning the data path for lower latency but ignores scheduling latency due to competing processes. It provides very good throughput performance, partly because it uses the CPU on the adapter to implement functions that would otherwise be done by the network adapter driver.

The Scout OS uses the notion of a *path* to expose the state and resource requirements of all processing components of a flow. In many ways our implementation methodology also reflects the path principle and incorporates it into the NetBSD OS. For instance, in our system, resources (such as CPU, memory, network interface, and network connection bandwidth) are allocated to a connection/application at the time of connection setup and, thus, it is similar to binding resources to a path in Scout. We also demonstrate that by locating all processing components of a data path in the same domain, we can schedule protocol and application functions in a single RTU invocation, thus obtaining end-to-end guarantees. In addition, since part of a data path (i.e., the VC queues) is in the kernel, we locate its state in shared memory so that the protocol code can initiate optimizations such as dropping packets at the VC queue when the socket buffer corresponding to it is full. At present, we are only aware of the use of paths in Scout for MPEG video decoding and display [32] and not for protocol processing.

Another approach that has been taken to provide scheduling support in the endsystems has been to take existing fair queueing schemes and adapt them for CPU scheduling. One such approach is adaptive rate-controlled (ARC) scheduling [41]. ARC scheduling [41] ensures that each thread makes progress by incrementing the tag value of a running thread and favoring threads with lower tags when the CPU is reassigned. While it represents a uniform approach to process scheduling, results relating to the design and performance of typical protocols using ARC scheduling have not been reported. A

possible drawback of ARC scheduling is that the amount of data processed in a period is not fixed but depends on the load on the system. This does not match well with a fixed-rate network connection and requires additional mechanisms in the endsystem to compensate. Another approach along similar lines is the one based on the start-time fair-queueing algorithm [21]. The focus is mainly on hierarchical partitioning of CPU between different scheduling classes. Our remarks about ARC scheduling apply to this case as well. In general, previous efforts have concentrated either on improving data path performance or on scheduling, but the two have not been treated in an integrated manner.

IX. Conclusions

We have presented a new mechanism to implement protocols in user space with QoS guarantees. We observe that current process scheduling is inappropriate for providing QoS guarantees. We also discuss why general-purpose mechanisms such as real-time threads will lead to inefficiencies. Our approach is to tradeoff the generality of real-time threads for efficiency and simplicity of our real-time upcall model. We have argued that the RTU mechanism is both necessary (from an efficiency standpoint) and sufficient (as an OS abstraction) for providing QoS guarantees for applications as well as protocol processing. We have implemented the RTU mechanism and the RMDP scheduling scheme in the NetBSD OS and demonstrated its efficient performance. We also highlight important performance issues in user-space protocol implementations. We show that system support is required to reduce the high cost of data movement and context switching in user-space protocol implementations. Using our implementation framework, we have eliminated system calls, VM operations, and data-copying operations in the packet processing path, while providing QoS guarantees. Our experimental results show that: 1) RTU-based protocol implementations deliver higher throughput than their kernel counterparts by keeping data movement overheads low; 2) the RTU mechanism provides throughput guarantees, bandwidth sharing, and partitioning among connections, even in the presence of background system load; and 3) the close integration of protocol processing and scheduling allows us to support both guaranteed-bandwidth as well as low-delay streams. To the best of our knowledge, our user-space protocol implementation is the first of its kind to combine both high efficiency as well as end-to-end QoS guarantees.

References

[1] M. L. Bailey *et al.*, "PathFinder: A pattern-based packet classifier," in *1st Symp. Operating System Design and Implementation*, Monterey, CA, Nov. 1994.

[2] B. N. Bershad, D. D, Redell, and J. R. Ellis, "Fast mutual exclusion for uniprocessors," in *Proc. ACM ASPLOS V*, Oct. 1992.

[3] J. C. Brustolini and P. Steenkiste, "Evaluation of data processing and scheduling avoidance," in *Network and Operating System Support for Digital Audio and Video (NOSSDAV'97)*, St. Louis, MO, May 1997, pp. 101–112.

[4] ——,"Effects of buffering semantics on I/O performance," in *2nd USENIX Symp. Operating Systems Design and Implementation*, Seattle, WA. Oct. 1996, pp. 271–291.

[5] M. M. Buddhikot, G. M. Parulkar, and R. Gopalakrishnan, "Scalable multimedia on demand via WWW," in *NOSSDAV 96*, Zushi, Japan, Apr. 1996.

[6] J. B. Chen, *et al.*, "The measured performance of personal computer operating systems," *15th ACM Symp. Operating Systems Principles*, Copper Mountain, CO, Dec. 1995.

[7] D. D. Clark, "The structuring of systems using upcalls," in *15th ACM Symp. Operating Systems Principles*, Copper Mountain, CO, Dec. 1985, pp. 171–180.

[8] D. D. Clark, V. Jacobsen, J. Romkey, and H. Salwen, "An analysis of TCP processing overhead," *IEEE Commun. Mag.*, vol. 27, pp. 23–23, July 1989.

[9] D. D. Clark, S. Shenker, and L. Zhang, "Supporting real-time applications in an integrated services packet network: Architecture and mechanism," in *Proc. ACM SIGCOMM*, Aug. 1992, pp. 14–26.

[10] R. B. Dannenberg, *el al.*, "Performance measurements of the multimedia testbed in RT-mach," Sch. Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-94-141, 1994.

[11] P. Druschel, L. L. Peterson, and B. S. Davie, "Experiences with a high-speed network adaptor: A software perspective," in *Proc. ACM SIGCOMM*, London, U.K., Aug. 1994, pp. 2–13.

[12] P. Druschel and B. Gaurav, "Lazy receiver processing (LRP): A network subsystem architecture for server systems," in *2nd USENIX Symp. Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[13] A. Edwards and S. Muir, "Experiences implementing a high performance TCP in user space," in *Proc. ACM SIGCOMM*, Cambridge, MA, 1995, pp. 196–205.

[14] T. V. Eicken, *et al.*, "U-net: A user level network interface for parallel and distributed computing," *15th ACM Symp. Operating Systems Principles*, Copper Mountain, CO, Dec. 1995.

[15] R. Gopalakrishnan, "Efficient quality of service support within endsystems for networked multimedia," Ph.D. dissertation, Dep. Comput. Sci., Washington Univ., St. Louis, MO, Dec. 1996.

[16] R. Gopalakrishnan and G. M. Parulkar, "Efficient quality of service support in multimedia computer operating systems," Dep. Comput. Sci., Washington Univ., St. Loius, MO, Tech. Rep. WUCS-94-26, Sept. 1994.

[17] ——, "Real-time upcalls: A mechanism to provide real-time processing guarantees," Dep. Comput. Sci., Washington Univ., St. Louis, MO, Tech. Rep. WUCS-95-06, Sept. 1995.

[18] ——, "Bringing real-time scheduling theory and practice closer for multimedia computing," in *Proc. ACM SIGMETRICS*, Philadelphia, PA, May 1996.

[19] ——, "Quality of service support for protocol processing within endsystems," in *High Speed Networking for Multimedia Applications*. Norwell, MA: Kluwer, 1995.

[20] R. Govindan and D. P. Anderson, "Scheduling and IPC mechanisms for continuous media," in *13th ACM Symp. Operating Systems Principles*, 1991.

[21] P. Goyal, X. Guo, and H. M. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *Symp. Operating Systems Design and Implementation*, Seattle, WA, Jan. 1997.

[22] S. Khanna, *et al.*, "Realtime scheduling in SunOS5.0," in *Proc. USENIX*, Winter 1992, pp. 375–390.

[23] C. Lee, K. Yoshida, C. W. Mercer, and R. Rajkumar, "Predictable communication protocol processing in real-time mach," in *Real-Time Technology and Applications Symp. RTAS'96*, June 1996.

[24] S. J. Leffler *et al.*, *The Design and Implementation of the 4.3BSC UNIX Operating System*. Reading, MA: Addison-Wesley, 1989.

[25] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. Ass. Comput. Mach.*, vol. 20, no. 1, pp. 46-61, Jan. 1973.

[26] C. Maeda and B. N. Bershad, "Protocol service decomposition for high performance networking," *14th ACM Symp. Operating Systems Principles*, Dec. 1993, pp. 244-245.

[27] B. D. Marsh *et al.*, "First class user level threads," *ACM Symp. Operating Systems Principles*, Oct. 1991, pp. 110–121.

[28] S. McCanne and V. Jacobson, "The BSC packet filter: A new architecture for user level packet capture," *USENIX '93 Winter Conf.*, Jan. 1993, pp. 259–269.

[29] J. C. Mogul *et al.*, "The packet filter: An efficient mechanism for user level network code," in *11th ACM Symp. Operating System Principles*, Nov. 1987, pp. 39–51.

[30] D. Mosberger, P. Druschel, and L. L. Peterson, "Implementing atomic sequences on uniprocessors using rollforward," *Software—Practice & Experience*, vol. 26, no. 1, pp. 1–24, Jan. 1996.

[31] D. Mosberger and L. L. Peterson, *et al.*, "Analysis of techniques to improve protocol latency," in *Proc. ACM SIGCOMM*, Stanford, CA, Aug. 1996.

[32] D. Mosberger and L. L. Peterson, "Making paths explicit in the scout operating system," in *2nd Symp. Operating Systems Design and Implementation (OOSDI96)*, Oct. 1996, pp. 153–167.
[33] D.J. Scales, M. Burrows, and C. A. Thekkath, "Experience with parallel computing on the AN2 network," in *10th Int. Parallel Processing Symp.*, Apr. 1996, pp. 94–103.
[34] M. D. Schroeder and M. Burrows, "Performance of Firefly RPC," *ACM Trans. Comput. Syst.*, vol. 8, no. 1, pp. 1–17, 1990.
[35] L. Sha, *et al.*, "Priority inheritanc protocols: An approach to real-time synchronization," *IEEE Trans. Computers*, vol. 39, pp. 1175–1185, Sept. 1990.
[36] G. R. Wright and W. R. Stevens, *TCP/IP Illustrated*, vol. 2. Reading, MA: Addison-Wesley, 1995.
[37] C. A. Thekkath, *et al.*, "Implementing network protocols at the user levels," in *Proc. ACM SIGCOMM*, Sept. 1993, pp. 64–72.
[38] H. Tokuda, T. Nakajima, and P. Rao, "Real-time mach: Toward predictable real-time systems," in *USENIX Mach Workshop*, Oct. 1990.
[39] D. Wallach, D. Engler, and F. Kaashoek, "ASHs: Application-specific handlers for high-performance messaging," in *Proc. ACM SIGCOMM*, Stanford, CA, Aug. 1996.
[40] D. Yau and S. S. Lam, "An architecture toward efficient OS support for distributed multimedia," *Proc. IS&T/SPIE Multimedia Computing and Networking Conf. '96*, San Jose, CA, Jan. 1996.
[41] ——, "Adaptive rate controlled scheduling for multimedia applications," *Proc. ACM Multimedia Conf. '96*, Boston, MA, Nov. 1996.
[42] L. Zhang, *et al.*, "RSVP: A new resource reservation protocol," *IEEE Network*, pp. 8–18, Sept. 1993.

R. Gopalakrishnan received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, India, in 1987, the M.Tech. degree in computer science from the Indian Institute of Technology, Delhi, India, in 1990, and the Ph.D degree in computer science from Washington University, St. Louis, MO, in 1996.

From 1987 to 1989 he was with the Indian Institute of Technology, Kanpur, India, as a Research Engineer in the ERNET project, where he wrote drivers to interface a variety of network adapters to UNIX TCP/IP. From 1990 to 1991 he was with Wipro Infotech, Bangalore, India, as a Senior Design Engineer, where he was involved in the development of an i860-based workstation focusing on the network subsystem under UNIX SVR4. He is currently with AT&T Laboratories—Research, Florham Park, NJ, as a Senior Member of the Technical Staff. His research interests are packet telephony, service differentiation in protocol stacks, and I/O subsystem optimizations for multimedia applications.

Gurudatta M. Parulkar (S'85–M'87) received the Ph.D. degree in computer science from the University of Delaware, Newark, in 1987 (with Professor David Farber).

He is currently with Washington University, St. Louis, MO, where he is Professor of Computer Science and Director of the Applied Research Laboratory (a network and multimedia systems laboratory). At Washington University, he has been leading several projects that emphasize prototyping of high-performance systems including a gigabit ATM desk-area network, an I/O subsystem in NetBSD that can support QoS guarantees, a scalable multimedia on-demand storage server and service, IP switching with integrated services, reliable multicast protocols, and a network monitoring, visualization, and control system.

Dr. Parulkar received the Frank A. Pehrson Graduate Student Achievement Award in Computer and Information Sciences from the University of Delaware, Newark.