The Design and Performance of an I/O Subsystem for Real-time ORB Endsystem Middleware

Douglas C. Schmidt, Fred Kuhns, Rajeev Bector, and David L. Levine
{schmidt,fredk,rajeev,levine}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA *

Abstract

There is increasing demand to extend Object Request Broker (ORB) middleware to support applications with stringent quality of service (QoS) requirements. However, conventional ORBs do not define standard features for specifying or enforcing end-to-end QoS for applications with deterministic real-time requirements. This paper provides two contributions to the study of real-time ORB middleware. First, it describes the design of a real-time I/O (RIO) subsystem optimized for ORB endsystems that support real-time applications running on "off-the-shelf" hardware and software. Second, it illustrates how integrating a real-time ORB with a real-time I/O subsystem can reduce latency bounds on end-to-end communication between high-priority clients without unduly penalizing low-priority and best-effort clients.

Keywords: Real-time CORBA Object Request Broker, Quality of Service for OO Middleware, Real-time I/O Subsystems.

1 Introduction

Object Request Broker (ORB) middleware like CORBA [1], DCOM [2], and Java RMI [3] is well-suited for request/response applications with best-effort quality of service (QoS) requirements. However, ORB middleware has historically not been well-suited for performance-sensitive, distributed real-time applications. In general, conventional ORBs suffer from (1) lack of QoS specification interfaces, (2) lack of QoS enforcement, (3) lack of real-time programming features, and (4) lack of performance optimizations [4].

To address these shortcomings, we have developed *The ACE ORB* (TAO) [5]. TAO is a high-performance, real-time ORB endsystem targeted for applications with deterministic and statistical QoS requirements, as well as best effort requirements. The TAO ORB endsystem contains the network interface, OS, communication protocol, and CORBA-compliant middleware components and features shown in Figure 1.

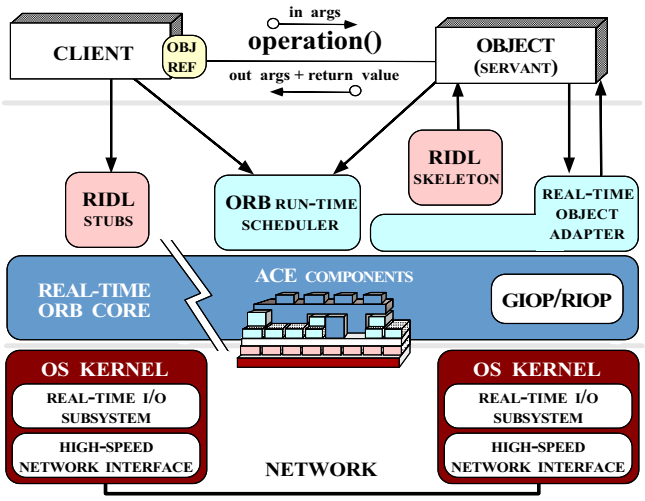

Figure 1: Components in the TAO Real-time ORB Endsystem

TAO's real-time I/O (RIO) subsystem runs in the OS kernel. It uses a pool of real-time threads to send/receive requests to/from clients across high-speed networks or I/O backplanes. TAO's ORB Core, Object Adapter, and servants run in user-space. TAO's ORB Core contains a pool of real-time threads that are co-scheduled with the RIO subsystem's thread pool. Together, these threads process client requests in accordance with their QoS requirements. TAO's Object Adapter uses perfect hashing [6] and active demultiplexing [7] to demultiplex these requests to application-level servant operations in constant $O(1)$ time.

We have used TAO to research key dimensions of real-time ORB endsystem design including static [5] and dynamic [8] real-time scheduling, real-time request demultiplexing [7], real-time event processing [9], and the performance of various commercial [10] and real-time research ORBs [11]. This paper focuses on an essential, and previously unexamined, di-

*This work was supported in part by Boeing, NSF grant NCR-9628218, DARPA contract 9701516, and US Sprint.

mension in the real-time ORB endsystem design space: *the development and empirical analysis of a real-time I/O (RIO) subsystem that enables TAO to support the QoS requirements of statically scheduled applications.*

The paper is organized as follows: Section 2 describes the architectural components in TAO's ORB endsystem that support statically scheduled applications; Section 3 explains how our real-time I/O (RIO) subsystem enhances the Solaris 2.5.1 OS kernel to support end-to-end QoS for TAO; Section 4 presents empirical results from systematically benchmarking the efficiency and predictability of TAO and RIO over an ATM network; and Section 6 presents concluding remarks.

2 The Design of the TAO Real-time ORB Endsystem

2.1 Real-time Support in TAO

To support the QoS demands of distributed object computing applications, an ORB endsystem must be flexible, efficient, predictable, and convenient to program. To meet these requirements, TAO provides end-to-end QoS specification and enforcement mechanisms and optimizations [5, 7, 11] for efficient and predictable request dispatching. The following overview of the real-time support in TAO's ORB endsystem explains how its components can be configured to support statically scheduled real-time applications.[1]

2.1.1 Architectural Overview

Figure 2 illustrates the key components in TAO's ORB endsystem. TAO supports the standard OMG CORBA reference model [1], with the following enhancements designed to overcome the shortcomings of conventional ORBs [11] for high-performance and real-time applications:

Real-time IDL Stubs and Skeletons: TAO's IDL stubs and skeletons efficiently marshal and demarshal operation parameters, respectively [13]. In addition, TAO's Real-time IDL (RIDL) stubs and skeletons extend the OMG IDL specifications to ensure that application timing requirements are specified and enforced end-to-end [14].

Real-time Object Adapter: An Object Adapter associates servants with the ORB and demultiplexes incoming requests to servants. TAO's real-time Object Adapter [15] uses perfect hashing [6] and active demultiplexing [7] optimizations to dispatch servant operations in constant $O(1)$ time, regardless of the number of active connections, servants, and operations defined in IDL interfaces.

[1] TAO's architecture for dynamically scheduled real-time applications is described in [12].

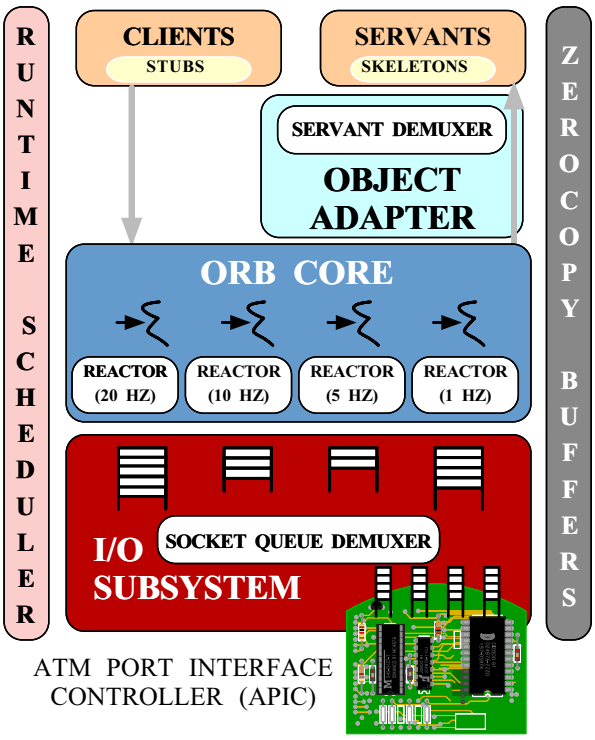

Figure 2: TAO's Real-time ORB Endsystem Architecture

ORB Run-time Scheduler: A real-time scheduler [16] maps application QoS requirements, such as include bounding end-to-end latency and meeting periodic scheduling deadlines, to ORB endsystem/network resources, such as ORB endsystem/network resources include CPU, memory, network connections, and storage devices. TAO's run-time scheduler supports both static [5] and dynamic [8] real-time scheduling strategies.

Real-time ORB Core: An ORB Core delivers client requests to the Object Adapter and returns responses (if any) to clients. TAO's real-time ORB Core [11] uses a multi-threaded, preemptive, priority-based connection and concurrency architecture [13] to provide an efficient and predictable CORBA IIOP protocol engine.

Real-time I/O subsystem: TAO's real-time I/O subsystem extends support for CORBA into the OS. TAO's I/O subsystem assigns priorities to real-time I/O threads so that the schedulability of application components and ORB endsystem resources can be enforced. TAO also runs efficiently and relatively predictably on conventional I/O subsystems that lack advanced QoS features.

High-speed network interface: At the core of TAO's I/O subsystem is a "daisy-chained" network interface consisting of one or more ATM Port Interconnect Controller (APIC)

chips [17]. APIC is designed to sustain an aggregate bi-directional data rate of 2.4 Gbps. In addition, TAO runs on conventional real-time interconnects, such as VME backplanes, multi-processor shared memory environments, as well as Internet protocols like TCP/IP.

TAO is developed atop lower-level middleware called ACE [18], which implements core concurrency and distribution patterns [19] for communication software. ACE provides reusable C++ wrapper facades and framework components that support the QoS requirements of high-performance, real-time applications. ACE runs on a wide range of OS platforms, including Win32, most versions of UNIX, and real-time operating systems like Sun/Chorus ClassiX, LynxOS, and VxWorks.

2.1.2 Configuring TAO for Statically Scheduled Real-time Applications

TAO can be configured to support different classes of application QoS requirements. For instance, the ORB endsystem configuration shown in Figure 2 is optimized for statically scheduled applications with periodic real-time requirements. Figure 3 illustrates an example of this type of application from the domain of avionics mission computing, where TAO has been deployed in mission computing applications at Boeing [9].

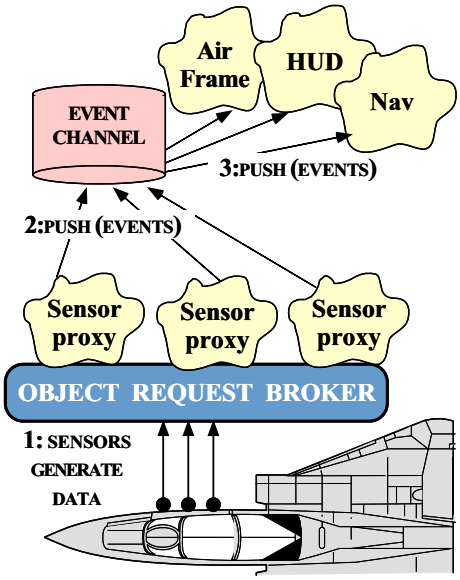

Figure 3: Real-time CORBA-based Avionics Mission Computing Application

Mission computing applications manage sensors and operator displays, navigate the aircraft's course, and control weapon release. These applications require an ORB endsystem that supports real-time scheduling and dispatching of periodic processing operations, as well as efficient event filtering and correlation mechanisms. To support these requirements, TAO's ORB Core supports the priority-based concurrency and connection architectures described below.

- **TAO's priority-based concurrency architecture:** TAO's ORB Core can be configured to allocate a real-time thread[2] for each application-designated priority level. Every thread in TAO's ORB Core can be associated with a `Reactor`, which implements the Reactor pattern [21] to provide flexible and efficient endpoint demultiplexing and event handler dispatching.

  When playing the role of a server, TAO's `Reactor`(s) demultiplex incoming client requests to connection handlers that perform GIOP processing. These handlers collaborate with TAO's Object Adapter to dispatch requests to application-level servant operations. Operations can either execute with one of the following two models [16]:

  - *Client propagation model* – The operation is run at the priority of the client that invoked the operation.
  - *Server sets model* – The operation is run at the priority of the thread in the server's ORB Core that received the operation.

  The server sets priority model is well-suited for deterministic real-time applications since it minimizes priority inversion and non-determinism in TAO's ORB Core [11]. In addition, it reduces context switching and synchronization overhead since servant state must be locked only if servants interact across different thread priorities.

  TAO's priority-based concurrency architecture is optimized for statically configured, fixed priority real-time applications. In addition, it is well suited for scheduling and analysis techniques that associate priority with *rate*, such as rate monotonic scheduling (RMS) and rate monotonic analysis (RMA) [22, 23]. For instance, avionics mission computing systems commonly execute their tasks in *rates groups*. A rate group assembles all periodic processing operations that occur at particular rates, *e.g.*, 20 Hz, 10 Hz, 5 Hz, and 1 Hz, and assigns them to a pool of threads using fixed-priority scheduling.

- **TAO's priority-based connection architecture:** Figure 4 illustrates how TAO can be configured with a priority-based connection architecture. In this model, each client thread maintains a `Connector` [24] in thread-specific storage. Each `Connector` manages a map of pre-established connections to servers. A separate connection is maintained for each thread priority in the server ORB. This design enables clients to preserve end-to-end priorities as requests traverse through ORB endsystems and communication links [11].

[2] In addition, TAO's ORB Core can be configured to support other concurrency architectures, including thread pool, thread-per-connection, and single-threaded reactive dispatching [20].

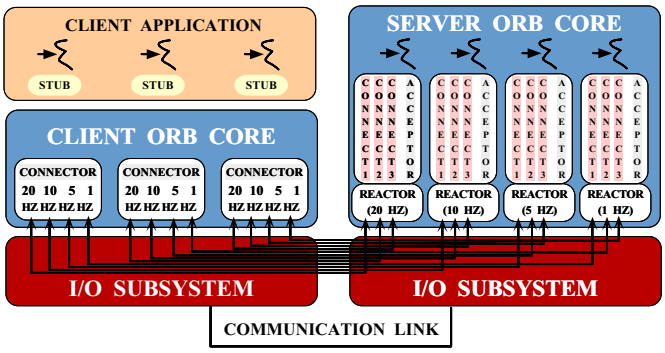

Figure 4: TAO's Priority-based Connection and Concurrency Architectures

Figure 4 also shows how the `Reactor` that is associated with each thread priority in a server ORB can be configured to use an `Acceptor` [24]. The `Acceptor` is a socket endpoint factory that listens on a specific port number for clients to connect to the ORB instance running at a particular thread priority. TAO can be configured so that each priority level has its own `Acceptor` port. For instance, in statically scheduled, rate-based avionics mission computing systems [12], ports 10020, 10010, 10005, 10001 could be mapped to the 20 Hz, 10 Hz, 5 Hz, and 1 Hz rate groups, respectively. Requests arriving at these socket ports can then be processed by the appropriate fixed-priority real-time threads.

Once a client connects, the `Acceptor` in the server ORB creates a new socket queue and a GIOP connection handler to service that queue. TAO's I/O subsystem uses the port number contained in arriving requests as a demultiplexing key to associate requests with the appropriate socket queue. This design minimizes priority inversion through the ORB endsystem via *early demultiplexing* [25, 26, 17], which associates requests arriving on network interfaces with the appropriate real-time thread that services the target servant. Early demultiplexing is used in TAO to vertically integrate the ORB endsystem's QoS support from the network interface up to the application servants.

2.2 Handling Real-time Client Requests in TAO

Real-time applications that use the TAO ORB endsystem can specify their resource requirements to TAO's *static scheduling service* [5]. TAO's scheduling service is implemented as a CORBA object, *i.e.*, it implements an IDL interface. Applications can provide QoS information to TAO's scheduling service on a per-operation basis, off-line, *i.e.*, before application execution. For CPU requirements, the QoS requirements are expressed by `RT_Operation`s using the attributes of the

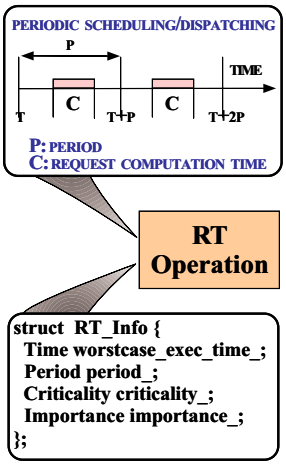

Figure 5: TAO's QoS Specification Model

`RT_Info` IDL `struct` shown in Figure 5.

An `RT_Operation` is a *scheduled operation*, *i.e.*, one that has expressed its resource requirements to TAO using an `RT_Info struct`. `RT_Info` attributes include worst-case execution time, period, importance, and data dependencies. Using scheduling techniques like RMS and analysis approaches like RMA, TAO's real-time static scheduling service determines if there is a feasible schedule based on knowledge of all `RT_Info` data for all the `RT_Operations` in an application. TAO's QoS specification model is described further in [5].

2.3 TAO's Run-time Scheduling Model

The scheduling information used by TAO's run-time scheduler shown in Figure 2 consists of *priorities* and *subpriorities*. Applications can associate these with each servant operation, as follows:

Priority: The priority is the (OS-dependent) thread priority of the highest priority thread that can execute the operation. It is determined based on the maximum *rate* at which servant operations can execute. For example, systems that are scheduled using RMS can compute this information based on knowledge of the computation time, C_o, and period, P_o, for each operation o.

Subpriority: The subpriority orders servant execution within each rate group. This ordering is based on two factors: (1) data dependencies between servants and (2) the relative importance of servants. Data dependencies are expressed directly to TAO's scheduler via the `RT_Info` QoS specification mechanism described in [5]. In most cases, these dependencies can be determined automatically.

The subpriority of an operation also depends on its *importance*. Importance is an application-specific indication of rela-

tive operation importance. It is used by the run-time scheduler as a "tie-breaker" when other scheduling parameters, *e.g.*, rate and data dependencies, are equal.

As described earlier, TAO's static scheduling service supports RMA and RMS. Therefore, its run-time scheduler component is simply an interface to a table of scheduling information that is pre-computed off-line, described in [5]. TAO's dynamic scheduling service [12] uses the same interface to drive more sophisticated on-line algorithms and retrieve the scheduling parameters interactively.

3 The Design of TAO's Real-time I/O Subsystem on Solaris over ATM

Meeting the requirements of distributed real-time applications requires more than defining QoS interfaces with CORBA IDL or developing an ORB with real-time thread priorities. In particular, it requires the integration of the ORB and the I/O subsystem to provide end-to-end real-time scheduling and real-time communication to the ORB endsystem.

This section describes how we have developed a real-time I/O (RIO) subsystem for TAO by customizing the Solaris 2.5.1 OS kernel to support real-time network I/O over ATM/IP networks [27]. Below, we examine the components that affect the performance and determinism of the RIO subsystem. Our main focus is on techniques that alleviate key sources of end-system priority inversion to reduce non-deterministic application and middleware behavior.

3.1 Overview of Solaris

3.1.1 From VxWorks to Solaris

TAO's original real-time I/O subsystem ran over a proprietary VME backplane protocol integrated into VxWorks running on a 200 MHz PowerPC CPU [9]. All protocol processing was performed at interrupt-level in a VxWorks device driver. This design was optimized for low latency, *e.g.*, two-way ORB operations were $\sim$350 μsecs.

Unfortunately, the VME backplane driver is not portable to a broad range of real-time systems. Moreover, it is not suitable for more complex transport protocols, such as TCP/IP, which cannot be processed entirely at interrupt-level without incurring excessive priority inversion [28]. Therefore, we developed a real-time I/O (RIO) subsystem that is integrated into a standard protocol processing framework: the STREAMS [29] communication I/O subsystem on Solaris.

We used Solaris as the basis for our research on TAO and RIO for the following reasons:

Real-time support: Solaris attempts to minimize dispatch latency [30] for real-time threads. Moreover, its fine-grained locking of kernel data structures allows bounded thread preemption overhead.

Multi-threading support: Solaris supports a scalable multi-processor architecture, with both kernel-level (kthreads) and user-level threads.

Dynamic configurability: Most Solaris kernel components are dynamically loadable modules, which simplifies debugging and testing of new kernel modules and protocol implementations.

Compliant to open system standards: Solaris supports the POSIX 1003.1c [31] real-time programming application programming interfaces (APIs), as well as other standard POSIX APIs for multi-threading [32] and communications.

Tool support: There are many software tools and libraries [18] available to develop multi-threaded distributed, real-time applications on Solaris.

Availability: Solaris is widely used in research and industry.

Kernel source: Sun licenses the source code to Solaris, which allowed us to modify the kernel to support the multi-threaded, real-time I/O scheduling class described in Section 3.3.

In addition to Solaris, TAO runs on a wide variety of real-time operating systems, such as LynxOS, VxWorks, and Sun/Chorus ClassiX, as well as general-purpose operating systems with real-time extensions, such as Digital UNIX, Windows NT, and Linux. We plan to integrate the RIO subsystem architecture described in this section into other operating systems that implement the STREAMS I/O subsystem architecture.

3.1.2 Synopsis of the Solaris Scheduling Model

Scheduling classes: Solaris extends the traditional UNIX time-sharing scheduler [33] to provide a flexible framework that allows dynamic linking of custom *scheduling classes*. For instance, it is possible to implement a new scheduling policy as a scheduling class and load it into a running Solaris kernel. By default, Solaris supports the four scheduling classes shown ordered by decreasing global scheduling priority below:

Scheduling Class	Priorities	Typical purpose
Interrupt (INTR)	160-169	Interrupt Servicing
Real-Time (RT)	100 - 159	Fixed priority scheduling
System (SYS)	60-99	OS-specific threads
Time-Shared (TS)	0-59	Time-Shared scheduling

The Time-Sharing (TS)[3] class is similar to the traditional UNIX scheduler [33], with enhancements to support interactive windowing systems. The System class (SYS) is used to schedule system kthreads, including I/O processing, and is not available to user threads. The Real-Time (RT) scheduling class uses fixed priorities above the SYS class. Finally, the highest system priorities are assigned to the Interrupt (INTR) scheduling class [34].

By combining a threaded, preemptive kernel with a fixed priority real-time scheduling class, Solaris attempts to provide a worst-case bound on the time required to dispatch user threads or kernel threads [30]. The RT scheduling class supports both Round-Robin and FIFO scheduling of threads. For Round-Robin scheduling, a time quantum specifies the maximum time a thread can run before it is preempted by another RT thread with the same priority. For FIFO scheduling, the highest priority thread can run for as long as it chooses, until it voluntarily yields control or is preempted by an RT thread with a higher priority.

Timer mechanisms: Many kernel components use the Solaris timeout facilities. To minimize priority inversion, Solaris separates its real-time and non-real-time timeout mechanisms [30]. This decoupling is implemented via two callout queue timer mechanisms: (1) `realtime_timeout`, which supports real-time callouts and (2) `timeout`, which supports non-real-time callouts.

The real-time callout queue is serviced at the lowest interrupt level, after the current clock tick is processed. In contrast, the non-real-time callout queue is serviced by a thread running with a SYS thread priority of 60. Therefore, non-real-time timeout functions cannot preempt threads running in the RT scheduling class.

3.1.3 Synopsis of the Solaris Communication I/O Subsystem

The Solaris communication I/O subsystem is an enhanced version of the SVR4 STREAMS framework [29] with protocols like TCP/IP implemented using STREAMS modules and drivers. STREAMS provides a bi-directional path between user threads and kernel-resident drivers. In Solaris, the STREAMS framework has been extended to support multiple threads of control within a Stream [35] and TCP/IP is based on the Mentat TCP/IP implementation for SVR4 STREAMS.

Below, we outline the key components of the STREAMS framework and describe how they affect communication I/O performance and real-time determinism.

[3] In this discussion we include the Interactive (IA) class, which is used primarily by Solaris windowing systems, with the TS class since they share the same range of global scheduling priorities.

General structure of a STREAM: A Stream is composed of a Stream head, a driver and zero or more modules linked together by read queues (RQ) and write queues (WQ), as shown in Figure 6. The Stream head provides an interface between a

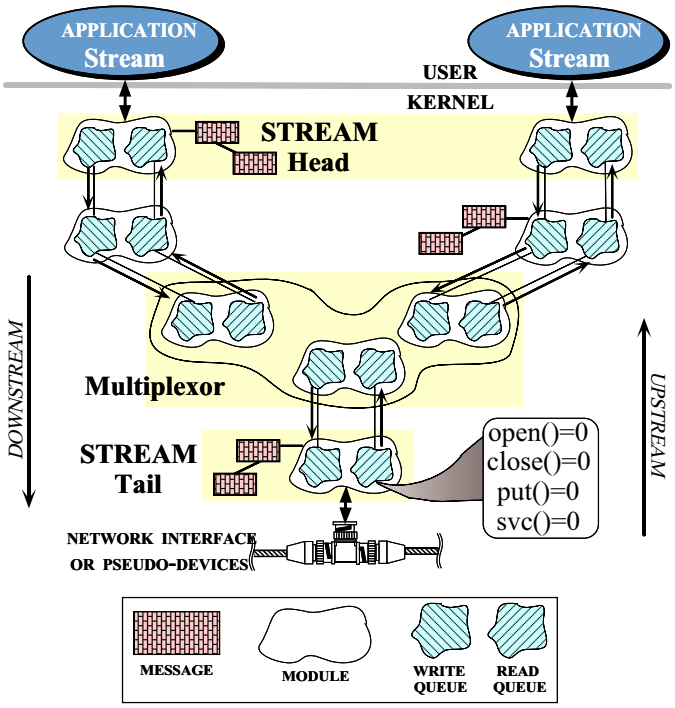

Figure 6: General Structure of a Stream

user process and a specific instance of a Stream in the kernel. It copies data across the user/kernel boundary, notifies user threads when data is available, and manages the configuration of modules into a Stream.

Each module and driver must define a set of entry points that handle `open`/`close` operations and process Stream messages. The message processing entry points are `put` and `svc`, which are referenced through the read and write queues. The `put` function provides the mechanism to send messages *synchronously* between modules, drivers, and the Stream head.

In contrast, the `svc` function processes messages *asynchronously* within a module or driver. A background thread in the kernel's SYS scheduling class runs `svc` functions at priority 60. In addition, `svc` functions will run after certain STREAMS-related system calls, such as `read`, `write`, and `ioctl`. When this occurs, the `svc` function runs in the context of the thread invoking the system call.

Flow control: Each module can specify a high and low watermark for its queues. If the number of enqueued messages exceeds the HIGH_WATERMARK the Stream enters the flow-controlled state. At this point, messages will be queued upstream or downstream until flow control abates.

For example, assume a STREAM driver has queued HIGH_WATERMARK+1 messages on its write queue. The first module atop the driver that detects this will buffer messages on its write queue, rather than pass them downstream. Because the Stream is flow-controlled, the `svc` function for the module will not run. When the number of messages on the driver's write queue drops below the LOW_WATERMARK the Stream will be re-enable automatically. At this point, the `svc` function for this queue will be scheduled to run.

STREAM Multiplexors: Multiple STREAMS can be linked together using a special type of driver called a *multiplexor*. A multiplexor acts like a driver to modules above it and as a Stream head to modules below it. Multiplexors enable the STREAMS framework to support layered network protocol stacks [36].

Figure 7 shows how TCP/IP is implemented using the Solaris STREAMS framework. IP behaves as a multiplexor by

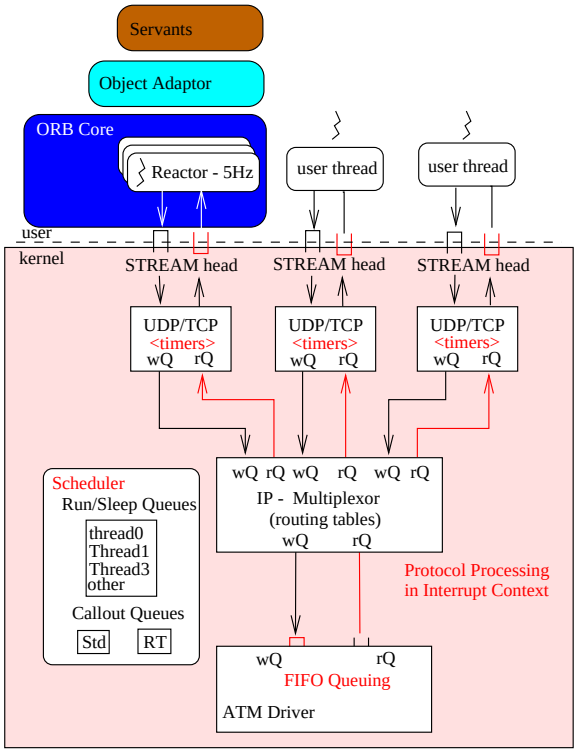

Figure 7: Conventional Protocol Stacks in Solaris STREAMS

joining different transport protocols with one or more link layer interfaces. Thus, IP demultiplexes both incoming and outgoing datagrams.

Each outgoing IP datagram is demultiplexed by locating its destination address in the IP routing table, which determines the network interface it must be forwarded to. Likewise, each incoming IP datagram is demultiplexed by examining the transport layer header in a STREAMS message to locate the transport protocol and port number that designates the correct upstream queue.

Multi-threaded STREAMS: Solaris STREAMS allows multiple kernel threads to be active in STREAMS I/O modules, drivers, and multiplexors concurrently [37]. This multi-threaded STREAMS framework supports several levels of concurrency, which are implemented using the *perimeters* [35] shown below:

Per-module with single threading
Per-queue-pair single threading
Per-queue single threading
Any of the above with unrestricted `put` and `svc`
Unrestricted concurrency

In Solaris, the concurrency level of IP is "per-module" with concurrent `put`, TCP and `sockmod` are "per-queue-pair," and UDP is "per-queue-pair" with concurrent `put`. These perimeters provide sufficient concurrency for common use-cases. However, there are cases where IP must raise its locking level when manipulating global tables, such as the IP routing table. When this occurs, messages entering the IP multiplexor are placed on a special queue and processed asynchronously when the locking level is lowered [35, 34].

Callout queue callbacks: The Solaris STREAMS framework provides functions to set timeouts and register callbacks. The `qtimeout` function adds entries to the standard non-real-time callout queue. This queue is serviced by a system thread with a SYS priority of 60, as described in Section 3.1.2. Solaris TCP and IP use this callout facility for their protocol-specific timeouts, such as TCP keepalive and IP fragmentation/reassembly.

Another mechanism for registering a callback function is `bufcall`. The `bufcall` function registers a callback function that is invoked when a specified size of buffer space becomes available. For instance, when buffers are unavailable, `bufcall` is used by a STREAM queue to register a function, such as `allocb`, which is called back when space is available again. These callbacks are handled by a system thread with priority SYS 60.

Network I/O: The Solaris network I/O subsystem provides service interfaces that reflect the OSI reference model [36]. These service interfaces consist of a collection of primitives and a set of rules that describe the state transitions.

Figure 7 shows how TCP/IP is structured in the Solaris STREAMS framework. In this figure, UDP and TCP implement the Transport Protocol Interface (TPI) [38], IP the Network Provider Interface (NPI) [39] and ATM driver the Data Link Provider Interface (DLPI) [40]. Service primitives are used (1) to communicate control (state) information and (2)

to pass data messages between modules, the driver, and the Stream head.

Data messages (as opposed to control messages) in the Solaris network I/O subsystem typically follow the traditional BSD model. When a user thread sends data it is copied into kernel buffers, which are passed through the Stream head to the first module. In most cases, these messages are then passed through each layer and into the driver through a nested chain of `puts` [35]. Thus, the data are sent to the network interface driver within the context of the sending process and typically are not processed asynchronously by module `svc` functions. At the driver, the data are either sent out immediately or are queued for later transmission if the interface is busy.

When data arrive at the network interface, an interrupt is generated and the data (usually referred to as a frame or packet) is copied into kernel buffer. This buffer is then passed up through IP and the transport layer in interrupt context, where it is either queued or passed to the Stream head via the socket module. In general, the use of `svc` functions is reserved for control messages or connection establishment.

3.2 Limitations of the Solaris I/O Subsystem for Real-time Scheduling and Protocol Processing

Section 3.1.3 outlined the structure and functionality of the existing Solaris 2.5.1 scheduling model and communication I/O subsystem. Below, we review the limitations of Solaris when it is used as the I/O subsystem for real-time ORB endsystems. These limitations stem largely from the fact that the Solaris RT scheduling class is not well integrated with the Solaris STREAMS-based network I/O subsystem. In particular, Solaris only supports the RT scheduling class for CPU-bound user threads, which yields the priority inversion hazards for real-time ORB endsystems described in Sections 3.2.1 and 3.2.2.

3.2.1 Thread-based Priority Inversions

Thread-based priority inversion can occur when a higher priority thread blocks awaiting a resource held by a lower priority thread [41]. In Solaris, this type of priority inversion generally occurs when real-time user threads depend on kernel processing that is performed at the SYS or INTR priority levels [30, 28, 41]. Priority inversion may not be a general problem for user applications with "best-effort" QoS requirements. It is problematic, however, for real-time applications that require bounded processing times and strict deadline guarantees.

The Solaris STREAMS framework is fraught with opportunities for thread-based priority inversion, as described below:

STREAMS-related `svc` threads: When used inappropriately, STREAMS `svc` functions can yield substantial unbounded priority inversion. The reason is that `svc` functions are called from a kernel `svc` thread, known as the STREAMS background thread. This thread runs in the SYS scheduling class with a global priority of 60.

In contrast, real-time threads have priorities ranging from 100 to 159. Thus, it is possible that a CPU-bound RT thread can starve the `svc` thread by monopolizing the CPU. In this case, the `svc` functions for the TCP/IP modules and multiplexors will not run, which can cause unbounded priority inversion.

For example, consider a real-time process control application that reads data from a sensor at a rate of 20 Hz and sends status messages to a remote monitoring system. Because this thread transmits time-critical data, it is assigned a real-time priority of 130 by TAO's run-time scheduler. When this thread attempts to send a message over a flow-controlled TCP connection, it will be queued in the `TCP` module for subsequent processing by the `svc` function.

Now, assume there is another real-time thread that runs asynchronously for an indeterminate amount of time responding to external network management trap events. This asynchronous thread has an RT priority of 110 and is currently executing. In this case, the asynchronous RT thread will prevent the `svc` function from running. Therefore, the high-priority message from the periodic thread will not be processed until the asynchronous thread completes, which can cause the unbounded priority inversion depicted in Figure 8.

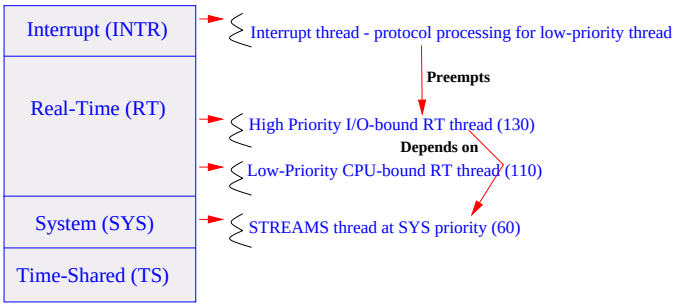

Figure 8: Common Sources of Priority Inversion in Solaris

In addition, two other STREAMS-related system kthreads can yield priority inversions when used with real-time applications. These threads run with a SYS priority of 60 and handle the callback functions associated with the `bufcall` and `qtimeout` system functions described in Section 3.1.3. This problem is further exacerbated by the fact that the priority of the thread that initially made the buffer request is not considered when these `svc` threads process the requests on their respective queues. Therefore, it is possible that a lower priority connection can receive buffer space before a higher priority connection.

Protocol processing with interrupt threads: Another source of thread-based priority inversion in Solaris STREAMS occurs when protocol processing of incoming packets is performed in interrupt context. Traditional UNIX implementations treat all incoming packets with equal priority, regardless of the priority of the user thread that ultimately receives the data.

In BSD UNIX-based systems [33], for instance, the interrupt handler for the network driver deposits the incoming packet in the IP queue and schedules a software interrupt that invokes the `ip_input` function. Before control returns to the interrupted user process, the software interrupt handler is run and `ip_input` is executed. The `ip_input` function executes at the lowest interrupt level and processes all packets in its input queue. Only when this processing is complete does control return to the interrupted process. Thus, not only is the process preempted, but it will be charged for the CPU time consumed by input protocol processing.

In STREAMS-based systems, protocol processing can either be performed at interrupt context (as in Solaris) or with `svc` functions scheduled asynchronously. Using `svc` functions can yield the unbounded priority inversion described above. Similarly, processing all input packets in interrupt context can cause unbounded priority inversion.

Modern high-speed network interfaces can saturate the system bus, memory, and CPU, leaving little time available for application processing. It has been shown that if protocol processing on incoming data is performed in interrupt context this can lead to a condition known as *receive* [28]. Livelock is a condition where the overall endsystem performance degrades due to input processing of packets at interrupt context. In extreme cases, an endsystem can spend the majority of its time processing input packets, resulting in little or no useful work being done. Thus, input livelock can prevent an ORB endsystem from meeting its QoS commitments to applications.

3.2.2 Packet-based Priority Inversions

Packet-based priority inversion can occur when packets for high-priority applications are queued behind packets for low-priority user threads. In the Solaris I/O subsystem, for instance, this can occur as a result of serializing the processing of incoming or outgoing network packets. To meet deadlines of time-critical applications, it is important to eliminate, or at least minimize, packet-based priority inversion.

Although TCP/IP in Solaris is multi-threaded, it incurs packet-based priority inversion since it enqueues network data in FIFO order. For example, TAO's priority-based ORB Core, described in Section 2.1.2, associates all packets destined for a particular TCP connection with a real-time thread of the appropriate priority. However, different TCP connections can be associated with different thread priorities. Therefore, packet-based priority inversion will result when the OS kernel places packets from different connections in the same queue and processes sequentially. Figure 7 depicts this case, where the queues shared by all connections reside in the IP multiplexor and interface driver.

To illustrate this problem, consider an embedded system where Solaris is used for data collection and fault management. This system must transmit both (1) high-priority real-time network management status messages and (2) low-priority bulk data radar telemetry. For the system to operate correctly, status messages must be delivered periodically with strict bounds on latency and jitter. Conversely, the bulk data transfers occur periodically and inject a large number of radar telemetry packets into the I/O subsystem, which are queued at the network interface.

In Solaris, the packets containing high-priority status messages can be queued in the network interface *behind* the lower priority bulk data radar telemetry packets. This situation yields packet-based priority inversion. Thus, status messages may arrive too late to meet end-to-end application QoS requirements.

3.3 RIO – An Integrated I/O Subsystem for Real-time ORB Endsystems

Enhancing a general-purpose OS like Solaris to support the QoS requirements of a real-time ORB endsystem like TAO requires the resolution of the following design challenges:

1. Creating an extensible and predictable I/O subsystem framework that can integrate seamlessly with a real-time ORB.
2. Alleviating key sources of packet-based and thread-based priority inversion.
3. Implementing an efficient and scalable packet classifier that performs early demultiplexing in the ATM driver.
4. Supporting high-bandwidth network interfaces, such as the APIC [17].
5. Supporting the specification and enforcement of QoS requirements, such as latency bounds and network bandwidth.
6. Providing all these enhancements to applications via the standard STREAMS network programming APIs [36].

This section describes the RIO subsystem enhancements we applied to the Solaris 2.5.1 kernel to resolve these design challenges. Our RIO subsystem enhancements provide a highly predictable OS run-time environment for TAO's integrated real-time ORB endsystem architecture, which is shown in Figure 9.

Our RIO subsystem enhances Solaris by providing QoS specification and enforcement features that complement

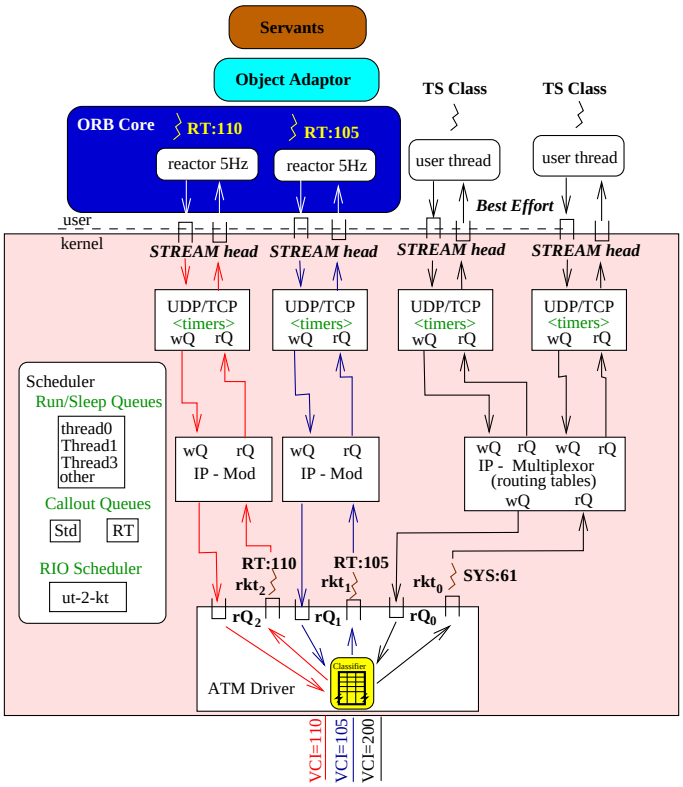

Figure 9: Architecture of the RIO Subsystem and Its Relationship to TAO

TAO's priority-based concurrency and connection architecture discussed in Section 2.1.2. The resulting real-time ORB endsystem contains user threads and kernel threads that can be scheduled statically. As described in Section 2.2, TAO's static scheduling service [5] runs off-line to map periodic thread requirements and task dependencies to a set of real-time global Solaris thread priorities. These priorities are then used on-line by the Solaris kernel's run-time scheduler to dispatch user and kernel threads on the CPU(s).

To develop the RIO subsystem and integrate it with TAO, we extended our prior work on ATM-based I/O subsystems to provide the following features:

Early demultiplexing: This feature associates packets with the correct priorities and a specific Stream early in the packet processing sequence, *i.e.*, in the ATM network interface driver [17]. RIO's design minimizes thread-based priority inversion by vertically integrating packets received at the network interface with the corresponding thread priorities in TAO's ORB Core.

Schedule-driven protocol processing: This feature performs all protocol processing in the context of kernel threads that are scheduled with the appropriate real-time priorities [25, 26, 42, 28]. RIO's design schedules network interface bandwidth and CPU time to minimize priority inversion and decrease interrupt overhead during protocol processing.

Dedicated Streams: This feature isolates request packets belonging to different priority groups to minimize FIFO queueing and shared resource locking overhead [43]. RIO's design resolves resource conflicts that can otherwise cause thread-based and packet-based priority inversions.

Below, we explore each of RIO's features and explain how they alleviate the limitations with Solaris' I/O subsystem described in Section 3.2. Our discussion focuses on how we resolved the key design challenges faced when building the RIO subsystem.

3.3.1 Early Demultiplexing

Context: ATM is a connection-oriented network protocol that uses virtual circuits (VCs) to switch ATM cells at high speeds [27]. Each ATM connection is assigned a virtual circuit identifier (VCI).

Problem: In Solaris STREAMS, packets received by the ATM network interface driver are processed sequentially and passed in FIFO order up to the IP multiplexor. Therefore, any information regarding the packets priority or specific connection is lost.

Solution: The RIO subsystem uses a packet classifier [44] to exploit the early demultiplexing feature of ATM [17] by vertically integrating its ORB endsystem architecture, as shown in Figure 10. Early demultiplexing uses the VCI field in a request packet to determine its final destination thread efficiently.

Early demultiplexing helps alleviate packet-based priority inversion because packets need not be queued in FIFO order. Instead, RIO supports *priority-based queueing*, where packets destined for high-priority applications are delivered ahead of low-priority packets. In contrast, the Solaris default network I/O subsystem processes all packets at the same priority, regardless of the user thread they are destined for.

Implementing early demultiplexing in RIO: The RIO endsystem can be configured so that protocol processing for each Stream is performed at different thread priorities. This design alleviates priority inversion when user threads running at different priorities perform network I/O. In addition, the RIO subsystem minimizes the amount of processing performed at interrupt level. This is necessary since Solaris does not consider packet priority or real-time thread priority when invoking interrupt functions.

At the lowest level of the RIO endsystem, the ATM driver distinguishes between packets based on their VCIs and stores them in the appropriate RIO queue (*rQ*). Each RIO queue pair is associated with exactly one Stream, but each Stream can be

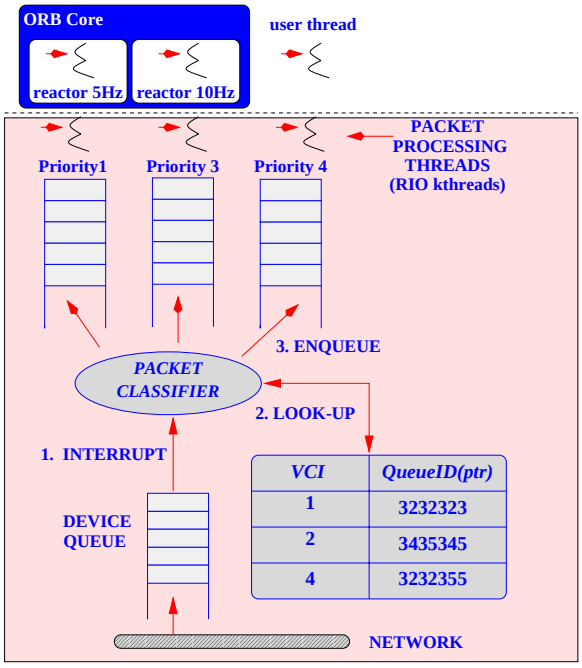

Figure 10: Early Demultiplexing in the RIO Subsystem

associated with zero or more RIO queues, *i.e.*, there is a many to one relationship for the RIO queues. The RIO protocol processing kthread associated with the RIO queue then delivers the packets to TAO's ORB Core, as shown in Figure 9.

Figure 9 also illustrates how all periodic connections are assigned a dedicated Stream, RIO queue pair, and RIO kthread for input protocol processing. RIO kthreads typically service their associated RIO queues at the periodic rate specified by an application. In addition, RIO can allocate kthreads to process the output RIO queue.

For example, Figure 9 shows four active connections: one periodic with a 10 Hz period, one periodic with a 5 Hz period, and two best-effort connections. Following the standard rate monotonic scheduling (RMS) model, the highest priority is assigned to the connection with the highest rate (10 Hz). In this figure, all packets entering on VCI 110 are placed in RIO queue rQ_2. This queue is serviced periodically by RIO kthread rkt_2, which runs at real-time priority 110.

After it performs protocol processing, thread rkt_2 delivers the packet to TAO's ORB Core where it is processed by a periodic user thread at real-time priority 110. Likewise, the 5 Hz connection transmits all data packets arriving on VCI 105 and protocol processing is performed periodically by RIO kthread rkt_1, which passes the packets up to the user thread.

The remaining two connections handle best-effort network traffic. The best-effort RIO queue (rQ_0) is serviced by a relatively low-priority kthread rkt_0. Typically this thread will be assigned a period and computation time[4] to bound the total throughput allowed on the best-effort connections.

3.3.2 Schedule-driven Protocol Processing

Context: Many real-time applications require periodic I/O processing. For example, avionics mission computers must process sensor data periodically to maintain accurate situational awareness [9]. If the mission computing system fails unexpectedly, corrective action must occur immediately.

Problem: Protocol processing of input packets in Solaris STREAMS is *demand-driven* [36], *i.e.*, when a packet arrives the STREAMS I/O subsystem suspends all user-level processing and performs protocol processing on the incoming packet. Demand-driven I/O can incur priority inversion, *e.g.*, when the incoming packet is destined for a thread with a priority lower than the currently executing thread. Thus, the ORB endsystem may become overloaded and fail to meet application QoS requirements.

When sending packets to another host, protocol processing is typically performed within the context of the user thread that performed the `write` operation. The resulting packet is passed to the driver for immediate transmission on the network interface link. With ATM, a pacing value can be specified for each active VC, which allows simultaneous pacing of multiple packets out the network interface. However, pacing may not be adequate in overload conditions since output buffers can overflow, thereby losing or delaying high-priority packets.

Solution: RIO's solution is to perform *schedule-driven*, rather than demand–driven, protocol processing of network I/O requests. We implemented this solution in RIO by adding kernel threads that are *co-scheduled* with real-time user threads in the TAO's ORB Core. This design vertically integrates TAO's priority-based concurrency architecture throughout the ORB endsystem.

Implementing Schedule-driven protocol processing in RIO: The RIO subsystem uses a *thread pool* [20] concurrency model to implement its schedule-driven kthreads. Thread pools are appropriate for real-time ORB endsystems since they (1) amortize thread creation run-time overhead and (2) place an upper limit on the percentage of CPU time used by RIO kthreads [11].

Figure 11 illustrates the thread pool model used in RIO. This pool of protocol processing kthreads, known as RIO kthreads, is created at I/O subsystem initialization. Initially these threads are not bound to any connection and are inactive until needed.

Each kthread in RIO's pool is associated with a queue. The queue links the various protocol modules in a Stream. Each

[4] Periodic threads must specify both a period P and a per period computation time T.

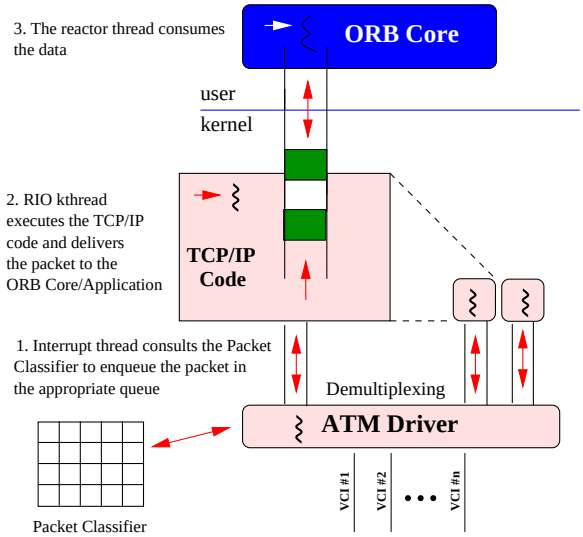

Figure 11: RIO Thread Pool Processing of TCP/IP with QoS Support

thread is assigned a particular *rate*, based on computations from TAO's static scheduling service [5]. This rate corresponds to the frequency at which requests are specified to arrive from clients. Packets are placed in the queue by the application (for clients) or by the interrupt handler (for servers). Protocol code is then executed by the thread to shepherd the packet through the queue to the network interface card or up to the application.

An additional benefit of RIO's thread pool design is its ability to bound the network I/O resources consumed by best-effort user threads. Consider the case of an endsystem that supports both real-time and best-effort applications. Assume the best-effort application is a file transfer utility like `ftp`. If an administrator downloads a large file to an endsystem, and no bounds are placed on the rate of input packet protocol processing, the system may become overloaded. However, with the RIO kthreads the total throughput allowed for best-effort connections can be bounded by specifying an appropriate period and computation time.

In statically scheduled real-time systems, kthreads in the pool are associated with different *rate groups*. This design complements the `Reactor`-based thread-per-priority concurrency model described in Section 2.1.2. Each kthread corresponds to a different rate of execution and hence runs at a different priority.

To minimize priority inversion throughout the ORB endsystem, RIO kthreads are co-scheduled with ORB Reactor threads. Thus, a RIO kthread processes I/O requests in the STREAMS framework and its user thread equivalent processes client requests in the ORB. Figure 12 illustrates how thread-based priority inversion is minimized in TAO's ORB endsystem by (1) associating a one-to-one binding between TAO user threads and STREAMS protocol kthreads and (2) minimizing the work done at interrupt context.

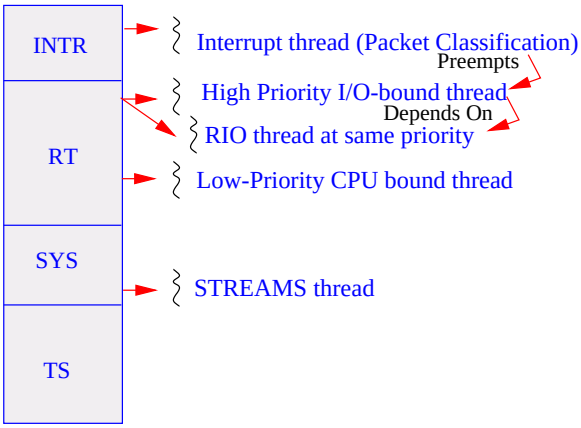

Figure 12: Alleviating Priority Inversion in TAO's ORB Endsystem

Both the ORB Core `Reactor` user thread and its associated RIO protocol kthread use Round-Robin scheduling. In this scheme, after either thread has a chance to run, its associated thread is scheduled. For instance, if the protocol kthread has packets for the application, the `Reactor`'s user thread in the ORB Core will consume the packets. Similarly if the application has consumed or generated packets, the protocol kthread will send or receive additional packets.

3.3.3 Dedicated Streams

Context: The RIO subsystem is responsible for enforcing QoS requirements for statically scheduled real-time applications with deterministic requirements.

Problem: Unbounded priority inversions can result when packets are processed asynchronously in the I/O subsystem without respect to their priority.

Solution: The effects of priority inversion in the I/O subsystem is minimized by isolating data paths through STREAMS such that resource contention is minimized. This is done in RIO by providing a *dedicated* Stream connection path that (1) allocates separate buffers in the ATM driver and (2) associates kernel threads with the appropriate RIO scheduling priority for protocol processing. This design resolves resource conflicts that can otherwise cause thread-based and packet-based priority inversions.

Implementing Dedicated Streams in RIO: Figure 9 depicts our implementation of Dedicated STREAMS in RIO. Incoming packets are demultiplexed in the driver and passed to

the appropriate Stream. A map in the driver's interrupt handler determines (1) the type of connection and (2) whether the packet should be placed on a queue or processed at interrupt context.

Typically, low-latency connections are processed in interrupt context. All other connections have their packets placed on the appropriate Stream queue. Each queue has an associated protocol kthread that processes data through the Stream. These threads may have different scheduling parameters assigned by TAO's scheduling service.

A key feature of RIO's Dedicated STREAMS design is its use of multiple output queues in the client's ATM driver. With this implementation, each connection is assigned its own transmission queue in the driver. The driver services each transmission queue according to its associated priority. This design allows RIO to associate low-latency connections with a relatively high-priority thread to assure that its packets are processed before all other packets in the system.

4 Empirical Benchmarking Results

This section presents empirical results from two groups of experiments. First, we show that the RIO subsystem decreases the upper bound on round-trip delay for latency-sensitive applications and provides periodic processing guarantees for bandwidth-sensitive applications. Second, we combine RIO and TAO to quantify the ability of the resulting ORB endsystem to support applications with real-time QoS requirements.

4.1 Hardware Configuration

Our experiments were conducted using a FORE Systems ASX-1000 ATM switch connected to two SPARCs: a uniprocessor 300 MHz UltraSPARC2 with 256 MB RAM and a 170 MHz SPARC5 with 64 MB RAM. Both SPARCs ran Solaris 2.5.1 and were connected via a FORE Systems SBA-200e ATM interface over an OC3 155 Mbps port on the ASX-1000. This benchmarking testbed is shown in Figure 13.

4.2 Measuring the End-to-end Real-time Performance of the RIO Subsystem

This section presents results that quantify (1) the cost of using kernel threads for protocol processing and (2) the benefits gained in terms of bounded latency response times and periodic processing guarantees. RIO uses a periodic processing model to provide bandwidth guarantees and to bound maximum throughput on each connection.

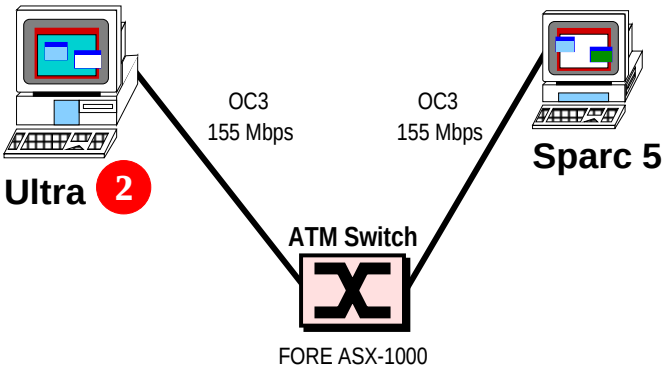

Figure 13: ORB Endsystem Benchmarking Testbed

4.2.1 Benchmarking Configuration

Our experiments were performed using the endsystem configuration shown in Figure 13. To measure round-trip latency we use a client application that opens a TCP connection to an "echo server" located on the SPARC5. The client sends a 64 byte data block to the echo server, waits on the socket for data to return from the echo server, and records the round-trip latency.

The client application performs 10,000 latency measurements, then calculates the mean latency, standard deviation, and standard error. Both the client and server run at the same thread priority in the Solaris real-time (RT) scheduling class.

Bandwidth tests were conducted using a modified version of `ttcp` [45] that sent 8 KB data blocks over a TCP connection from the UltraSPARC2 to the SPARC5. Threads that receive bandwidth reservations are run in the RT scheduling class, whereas best-effort threads run in the TS scheduling class.

4.2.2 Measuring the Relative Cost of Using RIO kthreads

Benchmark design: This set of experiments is designed to measure the relative cost of using RIO kthreads versus interrupt threads (*i.e.*, the default Solaris behavior) to process network protocols. The results show that it is relative efficient to perform protocol processing using RIO kthreads in the RT scheduling class.

The following three test scenarios used to measure the relative cost of RIO kthreads are based on the latency test in Section 4.2.1:

1. The default Solaris network I/O subsystem.
2. RIO enabled with the RIO kthreads in the real-time scheduling class with a global priority of 100.
3. RIO enabled with the RIO kthreads in the system

scheduling class with a global priority of 60 (system priority 0).

In all three cases, 10,000 samples were collected with the client and server user threads running in the real-time scheduling class with a global priority of 100.

Benchmark results and analysis: In each test, we determined the mean, maximum, minimum, and jitter (standard deviation) for each set of samples. The benchmark configuration is shown in Figure 14 and the results are summarized in the

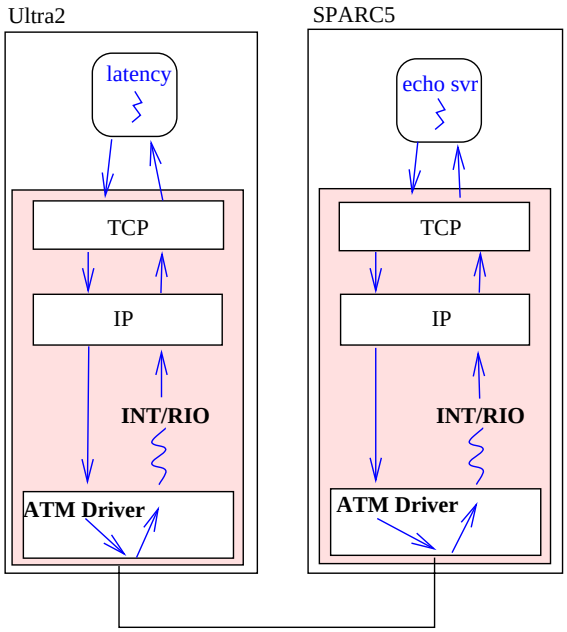

Figure 14: RIO kthread Test Configuration

table below:

	Mean	Max	Min	Jitter
Default behavior	653 μs	807 μs	613 μs	19.6
RIO RT kthreads	665 μs	824 μs	620 μs	18.8
RIO SYS kthreads	799 μs	1014 μs	729 μs	38.0

As shown in this table, when the RIO kthreads were run in the RT scheduling class the average latency increased by 1.8% or 12μs. The maximum latency value, which is a measure of the upper latency bound, increased by 2.1% or 17μs. The jitter, which represents the degree of variability, actually decreased by 4.1%. The key result is that jitter was not negatively affected by using RIO kthreads.

As expected, the mean latency and jitter increased more significantly when the RIO kthreads ran in the system scheduling class. This increase is due to priority inversion between the user and kernel threads, as well as competition for CPU time with other kernel threads running in the system scheduling class. For example, the STREAMS background threads, callout queue thread, and deferred bufcall processing all run with a global priority of 60 in the system scheduling class.

Figure 15 plots the distribution of the latency values for the latency experiments. This figure shows the number of samples

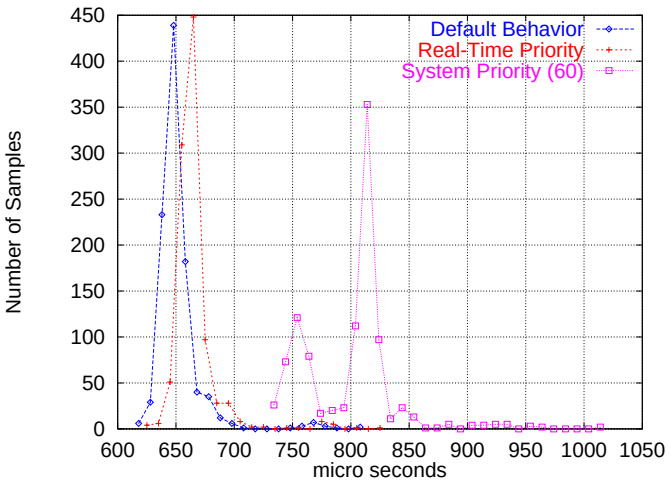

Figure 15: Latency Measurements versus Priority of kthreads

obtained at a given latency value $+/-5$ μs. The distribution of the default behavior and RIO with RT kthreads are virtually identical, except for a shift of $\sim$12 μs.

Section 3.1.3 describes the default Solaris I/O subsystem behavior. These measurements reveal the effect of performing network protocol processing at interrupt context versus performing it in a RIO kthread. With the interrupt processing model, the input packet is processed immediately up through the network protocol stack. Conversely, with the RIO kthreads model, the packet is placed in a RIO queue and the interrupt thread exits. This causes a RIO kthread to wake up, dequeue the packet, and perform protocol processing within its thread context.

A key feature of using RIO kthreads for protocol processing is their ability to assign appropriate kthread priorities and to defer protocol processing for lower priority connections. Thus, if a packet is received on a high-priority connection, the associated kthread will preempt lower priority kthreads to process the newly received data.

The results shown in Figure 15 illustrate that using RIO kthreads in the RT scheduling class results in a slight increase of 13-15 μs in the round-trip processing times. This latency increase stems from RIO kthread dispatch latencies and queuing delays. However, the significant result is that latency jitter decreases for real-time RIO kthreads.

4.2.3 Measuring Low-latency Connections with Competing Traffic

Benchmark design: This experiment measures the determinism of the RIO subsystem while performing prioritized protocol processing on a heavily loaded server. The results illustrate how RIO behaves when network I/O demands exceed the ability of the ORB endsystem to process all requests. The SPARC5 is used as the server in this test since it can process only ∼75% of the full link speed on an OC3 ATM interface using `ttcp` with 8 KB packets.

Two different classes of data traffic are created for this test: (1) a low-delay, high-priority message stream and (2) a best-effort (low-priority) bulk data transfer stream. The message stream is simulated using the latency application described in Section 4.2.1. The best-effort, bandwidth intensive traffic is simulated using a modified version of the `ttcp` program, which sends 8 KB packets from the client to the server.

The latency experiment was first run with competing traffic using the default Solaris I/O subsystem. Next, the RIO subsystem was enabled, RIO kthreads and priorities were assigned to each connection, and the experiment was repeated. The RIO kthreads used for processing the low-delay, high-priority messages were assigned a real-time global priority of 100. The latency client and echo server were also assigned a real-time global priority of 100.

The best-effort bulk data transfer application was run in the time-sharing class. The corresponding RIO kthreads ran in the system scheduling class with a global priority of 60. In general, all best effort connections use a RIO kthread in the SYS scheduling class with a global priority of 60.

Figure 16 shows the configuration for the RIO latency benchmark.

Benchmark results and analysis: The results from collecting 1,000 samples in each configuration are summarized in the table below:

	Mean	Max	Min	Jitter
Default	1072 μs	3158 μs	594 μs	497 μs
RIO	946 μs	2038 μs	616 μs	282 μs

This table compares the behavior of the default Solaris I/O subsystem with RIO. It illustrates how RIO lowers the upper bound on latency for low-delay, high-priority messages in the presence of competing network traffic. In particular, RIO lowered the maximum round-trip latency by 35% (1,120 μs), the average latency by 12% (126 μs), and jitter by 43% (215 μs). The distribution of samples are shown in Figure 17. This figure highlights how RIO lowers the upper bound of the round-trip latency values.

These performance results are particularly relevant for real-time systems where ORB endsystem predictability is crucial. The ability to specify and enforce end-to-end priorities

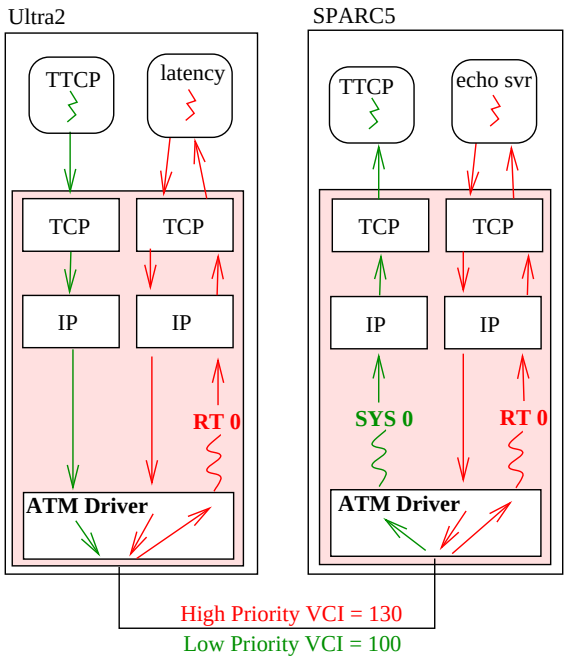

Figure 16: RIO Low-latency Benchmark Configuration

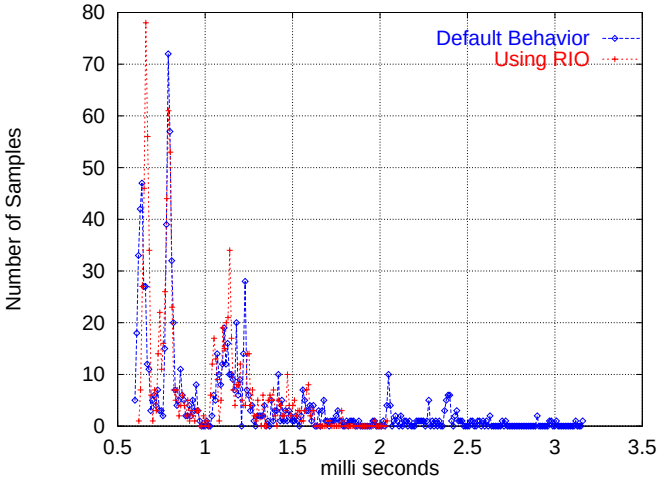

Figure 17: Latency with Competing Traffic

over transport connections helps ensure that ORB endsystems achieve end-to-end determinism.

Another advantage of RIO's ability to preserve end-to-end priorities is that the overall system utilization can be increased. For instance, the experiment above illustrates how the upper bound on latency was reduced by using RIO to preserve end-to-end priorities. For example, system utilization may be unable to exceed 50% while still achieving a 2 ms upper bound for high-priority message traffic. However, higher system utilization can be achieved when an ORB endsystem supports real-time I/O. This situation is demonstrated by the results in this section, where RIO achieved latencies no greater than 2.038 ms, even when the ORB endsystem was heavily loaded with best-effort data transfers.

Figure 18 shows the average bandwidth used by the modified `ttcp` applications during the experiment. The dip in

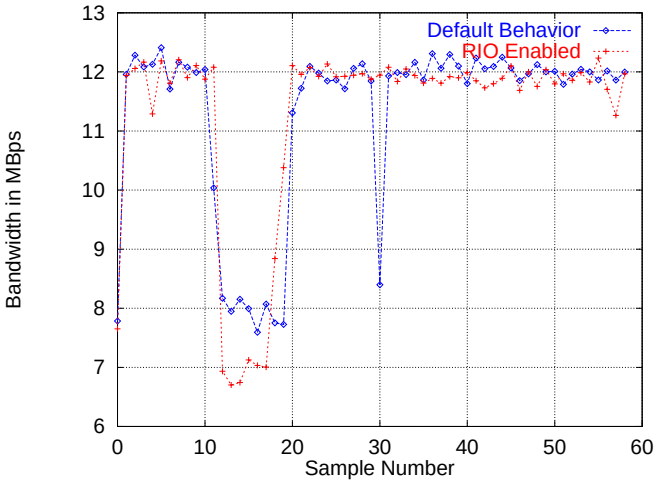

Figure 18: Bandwidth of Competing Traffic

throughput between sample numbers 10 and 20 occured when the high-priority latency test was run, which illustrates how RIO effectively reallocates resources when high-priority message traffic is present. Thus, the best-effort traffic obtains slightly lower bandwidth when RIO is used.

4.2.4 Measuring Bandwidth Guarantees for Periodic Processing

Benchmark design: RIO can enforce bandwidth guarantees since it implements the schedule-driven protocol processing model described in Section 3.3.2. In contrast, the default Solaris I/O subsystem processes all input packets on-demand at interrupt context, *i.e.*, with a priority higher than all other user threads and non-interrupt kernel threads.

The following experiment demonstrates the advantages and accuracy of RIO's periodic protocol processing model. The experiment was conducted using three threads that receive specific periodic protocol processing, *i.e.*, bandwidth, guarantees from RIO. A fourth thread sends data using only best-effort guarantees.

All four threads run the `ttcp` program, which sends 8 KB data blocks from the UltraSPARC2 to the SPARC5. For each bandwidth-guaranteed connection, a RIO kthread was allocated in the real-time scheduling class and assigned appropriate periods and packet counts, *i.e.*, computation time. The best-effort connection was assigned the default RIO kthread, which runs with a global priority of 60 in the system scheduling class. Thus, there were four RIO kthreads, three in the real-time scheduling class and one in the system class. The following table summarizes the RIO kthread parameters for the bandwidth experiment.

RIO Config	**Period**	**Priority**	**Packets**	**Bandwidth**
kthread 1	10 ms	110	8	6.4 MBps
kthread 2	10 ms	105	4	3.2 MBps
kthread 3	10 ms	101	2	1.6 MBps
kthread 4 (best-effort)	Async	60	Available	Available

The three user threads that received specific bandwidth guarantees were run with the same real-time global priorities as their associated RIO kthreads. These threads were assigned priorities related to their guaranteed bandwidth requirements – the higher the bandwidth the higher the priority. The `ttcp` application thread and associated RIO kthread with a guaranteed 6.4 MBps were assigned a real-time priority of 110. The application and RIO kernel threads with a bandwidth of 3.2 MBps and 1.6 MBps were assigned real-time priorities of 105 and 101, respectively.

As described in Section 3.3.1, the RIO kthreads are awakened at the beginning of each period. They first check their assigned RIO queue for packets. After processing their assigned number of packets they sleep waiting for the start of the next period.

The best-effort user thread runs in the time sharing class. Its associated RIO kthread, called the "best-effort" RIO kthread, is run in the system scheduling class with a global priority of 60. The best-effort RIO kthread is not scheduled periodically. Instead, it waits for the arrival of an eligible network I/O packet and processes it "on-demand." End-to-end priority is maintained, however, since the best-effort RIO kthread has a global priority lower than either the user threads or RIO kthreads that handle connections with bandwidth guarantees.

Benchmark results and analysis: In the experiment, the best-effort connection starts first, followed by the 6.4 MBps, 3.2 MBps and 1.6 MBps guaranteed connections, respectively. The results are presented in Figure 19 where the effect of the

guaranteed connection on the best-effort connection can be observed.

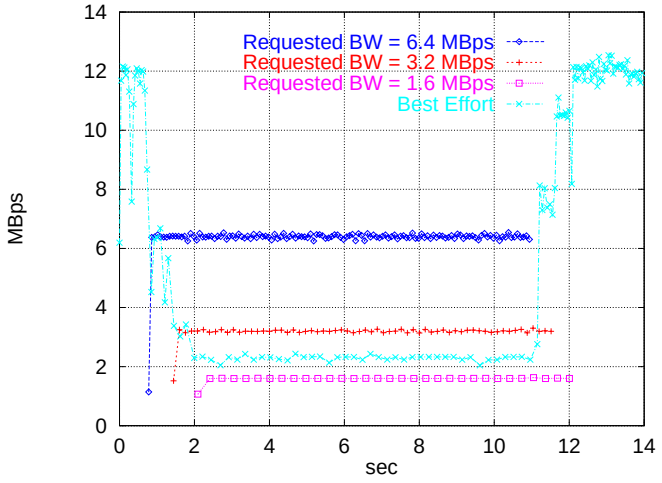

Figure 19: Bandwidth Guarantees in RIO

This figure clearly shows that the guaranteed connections received their requested bandwidths. In contrast, the best-effort connection loses bandwidth proportional to the bandwidth granted to guaranteed connections. The measuring interval was small enough for TCPs "slow start" algorithm [46] to be observed.

Periodic protocol processing is useful to guarantee bandwidth and bound the work performed for any particular connection. For example, we can specify that the best-effort connection in the experiment above receive no more than 40% of the available bandwidth on a given network interface.

4.3 Measuring the End-to-end Real-time Performance of the TAO/RIO ORB Endsystem

Section 4.2 measures the performance of the RIO subsystem in isolation. This section combines RIO and TAO to create a vertically integrated real-time ORB endsystem and then measures the impact on end-to-end performance when run with prototypical real-time ORB application workloads [11].

Benchmark design: The benchmark outlined below was performed twice: (1) without RIO, *i.e.*, using the unmodified default Solaris I/O subsystem and (2) using our RIO subsystem enhancements. Both benchmarks recorded average latency and the standard deviation of the latency values, *i.e.*, jitter. The server and client benchmarking configurations are described below.

• **Server benchmarking configuration:** As shown in Figure 20, the server host is the 170 MHz SPARC5. This host runs the real-time ORB with two servants in the Object Adapter. The *high-priority* servant runs in a thread with an RT

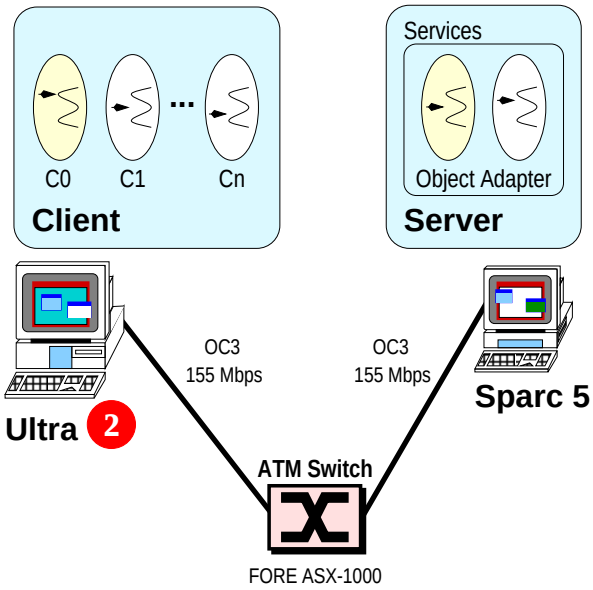

Figure 20: End-to-End ORB Endsystem Benchmark

priority of 130. The *low-priority* servant runs in a lower priority thread with an RT thread priority of 100. Each thread processes requests sent to it by the appropriate client threads on the UltraSPARC2. The SPARC5 is connected to a 155 Mpbs OC3 ATM interface so the UltraSPARC2 can saturate it with network traffic.

• **Client benchmarking configuration:** As shown in Figure 20, the client is the 300 MHz, uni-processor UltraSPARC2, which runs the TAO real-time ORB with one high-priority client C_0 and n low-priority clients, $C_1 \ldots C_n$. The high-priority client is assigned an RT priority of 130, which is the same as the high-priority servant. It invokes two-way CORBA operations at a rate of 20 Hz.

All low-priority clients have the same RT thread priority of 100, which is the same as the low-priority servant. They invoke two-way CORBA operations at 10 Hz. In each call the client thread sends a value of type `CORBA::Octet` to the servant. The servant cubes the number and returns the result.

The benchmark program creates all the client threads at startup time. The threads block on a barrier lock until all client threads complete their initialization. When all threads inform the main thread that they are ready, the main thread unblocks the clients. The client threads then invoke 4,000 CORBA two-way operations at the prescribed rates.

• **RIO subsystem configuration:** When the RIO subsystem is used, the benchmark has the configuration shown in Figure 21. With the RIO subsystem, high- and low-priority requests are treated separately throughout the ORB and I/O subsystem.

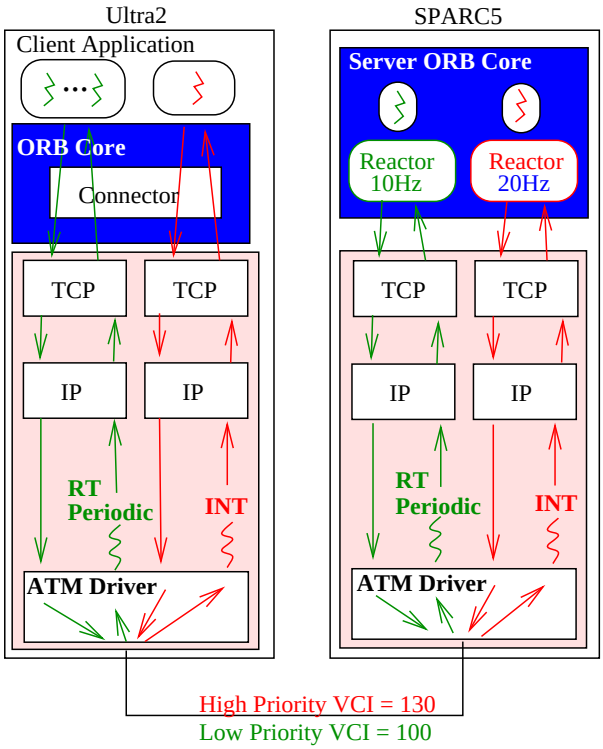

Figure 21: ORB Endsystem Benchmarking Configuration

Low-priority client threads transmit requests at 10 Hz. There are several ways to configure the RIO kthreads. For instance, we could assign one RIO kthread to each low-priority client. However, the number of low-priority clients varies from 0 to 50. Plus all clients have the same period and send the same number of requests per period, so they have the same priorities. Thus, only one RIO kthread is used. Moreover, since it is desirable to treat low-priority messages as best-effort traffic, the RIO kthread is placed in the system scheduling class and assigned a global priority of 60.

To minimize latency, high-priority requests are processed by threads in the Interrupt (INTR) scheduling class. Therefore, we create two classes of packet traffic: (1) low-latency, high priority and (2) best-effort latency, low-priority. The high-priority packet traffic preempts the processing of any low-priority messages in the I/O subsystem, ORB Core, Object Adapter, and/or servants.

Benchmark results and analysis: This experiment shows how RIO increases overall determinism for high-priority, real-time applications without sacrificing the performance of best-effort, low-priority, and latency-sensitive applications. RIO's impact on overall determinism of the TAO ORB endsystem is shown by the latency and jitter results for the high-priority client C_0 and the average latency and jitter for 0 to 49 low-priority clients, $C_1 \ldots C_n$.

Figure 22 illustrates the average latency results for the high- and low-priority clients both with and without RIO. This figure

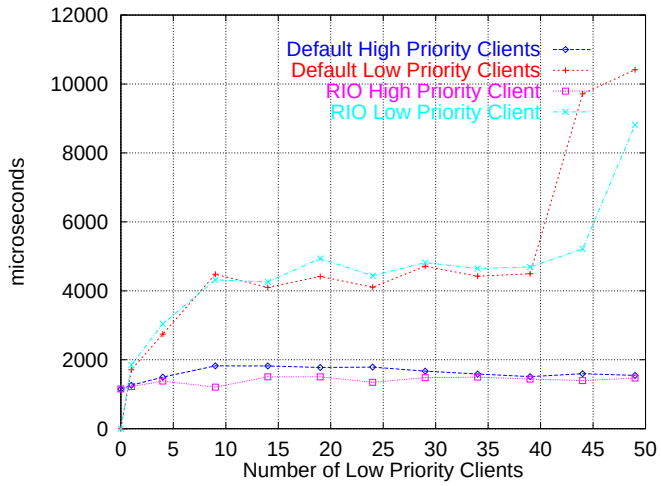

Figure 22: Measured Latency for All Clients with and without RIO

shows how TAO eliminates many sources of priority inversion within the ORB. Thus, high-priority client latency values are relatively constant, compared with low-priority latency values. Moreover, the high-priority latency values decrease when the the RIO subsystem is enabled. In addition, the low-priority clients' average latency values track the default I/O subsystems behavior, illustrating that RIO does not unduly penalize best-effort traffic. At 44 and 49 low-priority clients the RIO-enabled endsystem outperforms the default Solaris I/O subsystem.

Figure 23 presents a finer-grained illustration of the round-trip latency and jitter values for high-priority client vs. the number of competing low-priority clients. This figure illus-

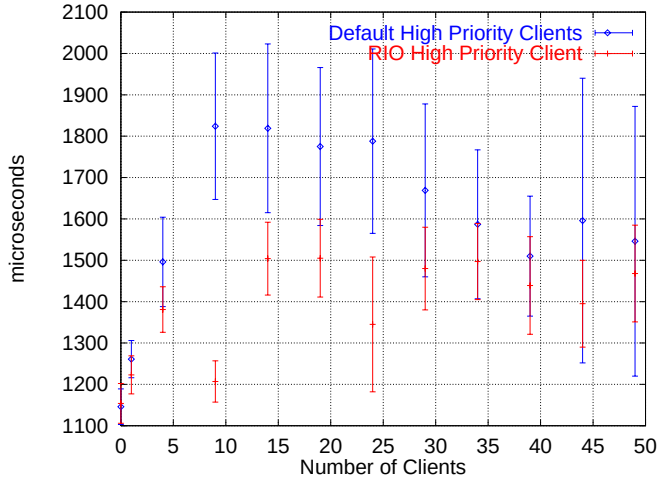

Figure 23: High-priority Client Latency and Jitter

trates how not only did RIO decrease average latency, but its jitter results were substantially better, as shown by the error bars in the figure. The high-priority clients averaged a 13% reduction in latency with RIO. Likewise, jitter was reduced by an average of 51%, ranging from a 12% increase with no competing low-priority clients to a 69% reduction with 44 competing low-priority clients.

In general, RIO reduced average latency and jitter because it used RIO kthreads to process low-priority packets. Conversely, in the default Solaris STREAMS I/O subsystem, servant threads are more likely to be preempted because threads from the INTR scheduling class are used for all protocol processing. Our results illustrate how this preemption can significantly increase latency and jitter values.

Figure 24 shows the average latency of low-priority client threads. This figure illustrates that the low-priority clients incurred no appreciable change in average latency. There was a slight increase in jitter for some combinations of clients due to the RIO kthreads dispatch delays and preemption by the higher priority message traffic. This result demonstrates how the RIO design can enhance overall end-to-end predictability for real-time applications while maintaining acceptable performance for traditional, best-effort applications.

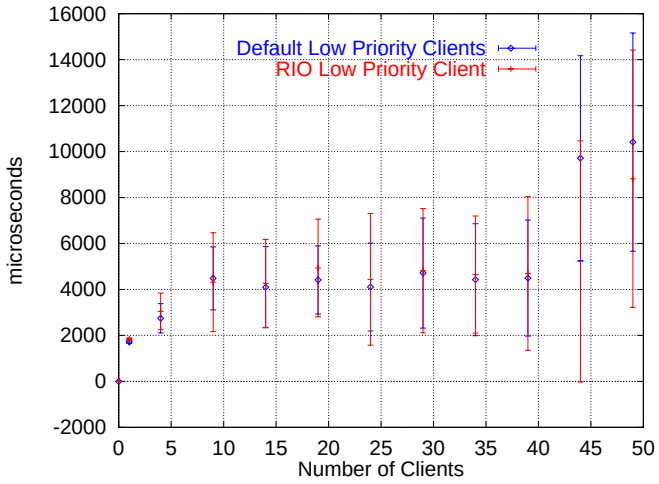

Figure 24: Low-priority Client Latency

4.4 Summary of Empirical Results

Our empirical results presented in Section 4 illustrate how RIO provides the following benefits to real-time ORB endsystems:

1. Reduced latency and jitter: RIO reduces round-trip latency and jitter for real-time network I/O, even during high network utilization. RIO prioritizes network protocol processing to ensure resources are available when needed by real-time applications.

2. Enforced bandwidth guarantees: The RIO periodic processing model provides network bandwidth guarantees. RIO's schedule-driven protocol processing enables an application to specify periodic I/O processing requirements which are used to guarantee network bandwidth.

3. Fine-grained resource control: RIO enables fine-grained control of resource usage, *e.g.*, applications can set the maximum throughput allowed on a per-connection basis. Likewise, applications can specify their priority and processing requirements on a per-connection basis. TAO also uses these specifications to create off-line schedules for statically configured real-time applications.

4. End-to-end priority preservation: RIO preserves end-to-end operation priorities by co-scheduling TAO's ORB Reactor threads with RIO kthreads that perform I/O processing.

5. Supports best-effort traffic: RIO supports the four QoS features described above without unduly penalizing best-effort, *i.e.*, traditional network traffic. RIO does not monopolize the system resources used by real-time applications. Moreover, since RIO does not use a fixed allocation scheme, resources are available for use by best-effort applications when they are not in use by real-time applications.

5 Related Work on I/O Subsystems

Our real-time I/O (RIO) subsystem incorporates advanced techniques [47, 17, 42, 43, 48] for high-performance and real-time protocol implementations. This section compares RIO with related work on I/O subsystems.

I/O subsystem support for QoS: The Scout OS [49, 50] employs the notion of a *path* to expose the state and resource requirements of all processing components in a *flow*. Similarly, our RIO subsystem reflects the path principle and incorporates it with TAO and Solaris to create a vertically integrated real-time ORB endsystem. For instance, RIO subsystem resources like CPU, memory, and network interface and network bandwidth are allocated to an application-level connection/thread during connection establishment, which is similar to Scout's binding of resources to a path.

Scout represents a fruitful research direction, which is complementary with our emphasis on demonstrating similar capabilities in existing operating systems, such as Solaris and NetBSD [25]. At present, paths have been used in Scout largely for MPEG video decoding and display and not for protocol processing or other I/O operations. In contrast, we have successfully used RIO for a number of real-time avionics applications [9] with deterministic QoS requirements.

SPIN [51, 52] provides an extensible infrastructure and a core set of extensible services that allow applications to safely

change the OS interface and implementation. Application-specific protocols are written in a typesafe language, *Plexus*, and configured dynamically into the SPIN OS kernel. Because these protocols execute within the kernel, they can access network interfaces and other OS system services efficiently. To the best of our knowledge, however, SPIN does not support end-to-end QoS guarantees.

Enhanced I/O subsystems: Other related research has focused on enhancing performance and fairness of I/O subsystems, though not specifically for the purpose of providing real-time QoS guarantees. These techniques are directly applicable to designing and implementing real-time I/O and providing QoS guarantees, however, so we compare them with our RIO subsystem below.

[43] applies several high-performance techniques to a STREAMS-based TCP/IP implementation and compares the results to a BSD-based TCP/IP implementation. This work is similar to RIO since they parallelize their STREAMS implementation and implement early demultiplexing and dedicated STREAMS, known as Communication Channels (CC). The use of CC exploits the built-in flow control mechanisms of STREAMS to control how applications access the I/O subsystem. This work differs from RIO, however, since it focuses entirely on performance issues and not sources of priority inversions. For example, minimizing protocol processing in interrupt context is not addressed.

[28, 42] examines the effect of protocol processing with interrupt priorities and the resulting priority inversions and livelock [28]. Both approaches focus on providing fairness and scalability under network load. In [42], a network I/O subsystem architecture called *lazy receiver processing* (LRP) is used to provide stable overload behavior. LRP uses early demultiplexing to classify packets, which are then placed into per-connection queues or on network interface channels. These channels are shared between the network interface and OS. Application threads read/write from/to network interface channels so input and output protocol processing is performed in the context of application threads. In addition, a scheme is proposed to associate kernel threads with network interface channels and application threads in a manner similar to RIO. However, LRP does not provide QoS guarantees to applications.

[28] proposed a somewhat different architecture to minimize interrupt processing for network I/O. They propose a polling strategy to prevent interrupt processing from consuming excessive resources. This approach focuses on scalability under heavy load. It did not address QoS issues, however, such as providing per-connection guarantees for fairness or bandwidth, nor does it charge applications for the resources they use. It is similar to our approach, however, in that (1) interrupts are recognized as a key source of nondeterminism and (2) schedule-driven protocol processing is proposed as a solution.

While RIO shares many elements of the approaches described above, we have combined these concepts to create the first vertically integrated real-time ORB endsystem. The resulting ORB endsystem provides scalable performance, periodic processing guarantees and bounded latency, as well as an end-to-end solution for real-time distributed object computing middleware and applications.

6 Concluding Remarks

Conventional operating systems and ORBs do not provide adequate support for the QoS requirements of distributed, real-time applications. Meeting these needs requires an integrated ORB endsystem architecture that delivers end-to-end QoS guarantees at multiple levels. The ORB endsystem described in this paper addresses this need by combining a real-time I/O (RIO) subsystem with the TAO ORB Core [11] and Object Adapter [53], which are explicitly designed to preserve QoS properties end-to-end in distributed real-time systems.

This paper focuses on the design and performance of RIO. RIO is a real-time I/O subsystem that enhances the Solaris 2.5.1 kernel to enforce the QoS features of the TAO ORB endsystem. RIO provides QoS guarantees for vertically integrated ORB endsystems that increase (1) throughput and latency performance and (2) end-to-end determinism. RIO supports periodic protocol processing, guarantees I/O resources to applications, and minimizes the effect of flow control in a Stream.

A novel feature of the RIO subsystem is its integration of real-time scheduling and protocol processing, which allows RIO to support guaranteed bandwidth and low-delay applications. To accomplish this, we extended the concurrency architecture and thread priority mechanisms of TAO into the RIO subsystem. This design minimizes sources of priority inversion that cause nondeterministic behavior.

After integrating RIO with TAO, we measured a significant reduction in average latency and jitter. Moreover, the latency and jitter of low-priority traffic was not affected adversely. As a result of our RIO enhancements to the Solaris kernel, TAO is the first ORB to support end-to-end QoS guarantees over ATM/IP networks [27].

In addition, implementing RIO allowed us to experiment with alternative concurrency strategies and techniques for processing network I/O requests. Our results illustrate how configuring periodic protocol processing [54] strategies in the Solaris kernel can provide significant improvements in system behavior, compared with the conventional Solaris I/O subsystem.

The TAO and RIO integration focused initially on statically scheduled applications with deterministic QoS re-

quirements. We have extended the TAO ORB endsystem to support dynamically scheduling [8] and applications with statistical QoS requirements. The C++ source code for ACE, TAO, and our benchmarks is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`. The RIO subsystem is available to Solaris source licensees.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.

[2] D. Box, *Essential COM*. Addison-Wesley, Reading, MA, 1997.

[3] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, November/December 1996.

[4] D. C. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar, "A High-Performance Endsystem Architecture for Real-time CORBA," *IEEE Communications Magazine*, vol. 14, February 1997.

[5] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[6] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the 2^{nd} C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, April 1990.

[7] A. Gokhale and D. C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," in *Proceedings of GLOBECOM '97*, (Phoenix, AZ), IEEE, November 1997.

[8] C. D. Gill, D. L. Levine, and D. C. Schmidt, "Evaluating Strategies for Real-Time CORBA Dynamic Scheduling," *submitted to the International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 1998.

[9] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[10] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 306–317, ACM, August 1996.

[11] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," in *Proceedings of the 4^{th} IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[12] D. L. Levine, C. D. Gill, and D. C. Schmidt, "Dynamic Scheduling Strategies for Avionics Mission Computing," in *Proceedings of the 17th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Nov. 1998.

[13] A. Gokhale and D. C. Schmidt, "Techniques for Optimizing CORBA Middleware for Distributed Embedded Systems," in *Proceedings of INFOCOM '99*, Mar. 1999.

[14] V. F. Wolfe, L. C. DiPippo, R. Ginis, M. Squadrito, S. Wohlever, I. Zykh, and R. Johnston, "Real-Time CORBA," in *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, (Montréal, Canada), June 1997.

[15] A. Gokhale, I. Pyarali, C. O'Ryan, D. C. Schmidt, V. Kachroo, A. Arulanthu, and N. Wang, "Design Considerations and Performance Optimizations for Real-time ORBs," in *Submitted to the 5^{th} Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[16] Object Management Group, *Realtime CORBA 1.0 Joint Submission*, OMG Document orbos/98-11-03 ed., November 1998.

[17] Z. D. Dittia, G. M. Parulkar, and J. Jerome R. Cox, "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, (Kobe, Japan), IEEE, April 1997.

[18] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[20] D. C. Schmidt, "Evaluating Architectures for Multi-threaded CORBA Object Request Brokers," *Communications of the ACM special issue on CORBA*, vol. 41, Oct. 1998.

[21] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[22] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, pp. 46–61, January 1973.

[23] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Norwell, Massachusetts: Kluwer Academic Publishers, 1993.

[24] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[25] C. Cranor and G. Parulkar, "Design of Universal Continuous Media I/O," in *Proceedings of the 5th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '95)*, (Durham, New Hampshire), pp. 83–86, Apr. 1995.

[26] R. Gopalakrishnan and G. Parulkar, "Bringing Real-time Scheduling Theory and Practice Closer for Multimedia Computing," in *SIGMETRICS Conference*, (Philadelphia, PA), ACM, May 1996.

[27] G. Parulkar, D. C. Schmidt, and J. S. Turner, "$\mathrm{a^I t^P m}$: a Strategy for Integrating IP with ATM," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, ACM, September 1995.

[28] J. C. Mogul and K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-driver Kernel," in *Proceedings of the USENIX 1996 Annual Technical Conference*, (San Diego, CA), USENIX, Jan. 1996.

[29] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

[30] S. Khanna and et. al., "Realtime Scheduling in SunOS 5.0," in *Proceedings of the USENIX Winter Conference*, pp. 375–390, USENIX Association, 1992.

[31] "Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language]," 1995.

[32] IEEE, *Threads Extension for Portable Operating Systems (Draft 10)*, February 1996.

[33] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[34] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[35] S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan, "Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes," in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 85–106, Jan. 1993.

[36] S. Rago, *UNIX System V Network Programming*. Reading, MA: Addison-Wesley, 1993.

[37] Sun Microsystems, *STREAMS Programming Guide*. Sun Microsystems, Inc., Mountain View, CA, August 1997. Revision A.

[38] OSI Special Interest Group, *Transport Provider Interface Specification*, December 1992.

[39] OSI Special Interest Group, *Network Provider Interface Specification*, December 1992.

[40] OSI Special Interest Group, *Data Link Provider Interface Specification*, December 1992.

[41] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-time Synchronization," *IEEE Transactions on Computers*, vol. 39, September 1990.

[42] P. Druschel and G. Banga, "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems," in *Proceedings of the* 1^{st} *Symposium on Operating Systems Design and Implementation*, USENIX Association, October 1996.

[43] T. B. Vincent Roca and C. Diot, "Demultiplexed Architectures: A Solution for Efficient STREAMS-Based Communication Stacks," *IEEE Network Magazine*, vol. 7, July 1997.

[44] M. L. Bailey, B. Gopal, P. Sarkar, M. A. Pagels, and L. L. Peterson, "Pathfinder: A pattern-based packet classifier," in *Proceedings of the* 1^{st} *Symposium on Operating System Design and Implementation*, USENIX Association, November 1994.

[45] USNA, *TTCP: a test of TCP and UDP Performance*, Dec 1984.

[46] W. R. Stevens, *TCP/IP Illustrated, Volume 1*. Reading, Massachusetts: Addison Wesley, 1993.

[47] T. v. Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," in *15th ACM Symposium on Operating System Principles*, ACM, December 1995.

[48] J. Mogul and S. Deering, "Path MTU Discovery," *Network Information Center RFC 1191*, pp. 1–19, Apr. 1990.

[49] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. P. sting, and J. H. Hartman, "Scout: A communications-oriented operating system," Tech. Rep. 94-20, Department of Computer Science, University of Arizona, June 1994.

[50] D. Mosberger and L. Peterson, "Making Paths Explicit in the Scout Operating System," in *Proceedings of OSDI '96*, Oct. 1996.

[51] B. Bershad, "Extensibility, Safety, and Performance in the Spin Operating System," in *Proceedings of the* 15^{th} *ACM SOSP*, pp. 267–284, 1995.

[52] M. Fiuczynski and B. Bershad, "An Extensible Protocol Architecture for Application-Specific Networking," in *Proceedings of the 1996 Winter USENIX Conference*, Jan. 1996.

[53] I. Pyarali and D. C. Schmidt, "An Overview of the CORBA Portable Object Adapter," *ACM StandardView*, vol. 6, Mar. 1998.

[54] R. Gopalakrishnan and G. Parulkar, "A Real-time Upcall Facility for Protocol Processing with QoS Guarantees," in 15^{th} *Symposium on Operating System Principles (poster session)*, (Copper Mountain Resort, Boulder, CO), ACM, Dec. 1995.