The Design and Performance of a Real-time I/O Subsystem

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

This paper appeared in the proceedings of the Real-Time Technology and Applications Symposium (RTAS), Vancouver, British Columbia, Canada, June 2–4, 1999.

Abstract

This paper describes the design and performance of a real-time I/O (RIO) subsystem that supports real-time applications running on off-the-shelf hardware and software. This paper provides two contributions to the study of real-time I/O subsystems. First, it describes how RIO supports end-to-end, prioritized traffic to bound the I/O utilization of each priority class and eliminates the key sources of priority inversion in I/O subsystems. Second, it illustrates how 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.

1 Introduction

This paper focuses on the design and performance of a real-time I/O (RIO) subsystem that enhances the Solaris 2.5.1 kernel to enforce the QoS features of the The ACE ORB (TAO) [1] endsystem. RIO provides QoS guarantees for vertically integrated ORB endsystems in order to (1) increase throughput, (2) decrease latency and (3) improve end-to-end predictability. 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 non-determinism.

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

The paper is organized as follows: Section 2 describes how the RIO subsystem enhances the Solaris 2.5.1 OS kernel to support end-to-end QoS for TAO applications; Section 3 presents empirical results from systematically benchmarking the efficiency and predictability of TAO and RIO over an ATM network; and Section 4 presents concluding remarks.

2 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. Instead, 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 CORBA applications. 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 [2].

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 [3].
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 [4].

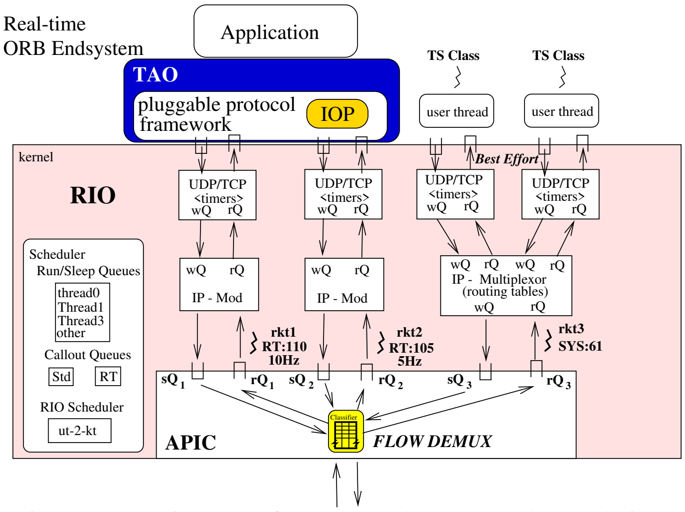

Figure 1: Architecture of the RIO Subsystem and Its Relationship to TAO

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 1.

Our RIO subsystem enhances Solaris by providing QoS specification and enforcement features that complement TAO's priority-based concurrency and connection architecture [5]. The resulting real-time ORB endsystem contains user threads and kernel threads that can be scheduled statically. As described in [1], TAO's static scheduling service runs off-line to map periodic thread requirements and task dependencies to a set of real-time global 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 [3]. 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 [6, 7, 8, 9]. 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 [10]. 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. Our discussion focuses on how we resolved the key design challenges faced when building the RIO subsystem.

2.0.1 Early Demultiplexing

Context: ATM is a connection-oriented network protocol that uses virtual circuits (VCs) to switch ATM cells at high speeds [2]. Each ATM connection is assigned a virtual circuit identifier (VCI)[1] which is included as part of the cell header.

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 containing the packets' priority or specific connection is lost.

Solution: The RIO subsystem uses a packet classifier [11] to exploit the early demultiplexing feature of ATM [3] by vertically integrating its ORB endsystem architecture, as shown in Figure 2. 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 destination user thread.

Implementing early demultiplexing in RIO: The RIO endsystem can be configured so that protocol processing for each STREAM is performed with appropriate 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 because 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* for receive queue and *sQ* for send queue). Each RIO queue pair is associated with exactly one STREAM, but each STREAM can be associated with

[1] A virtual path identifier is also assigned but for this work we only consider the VCI

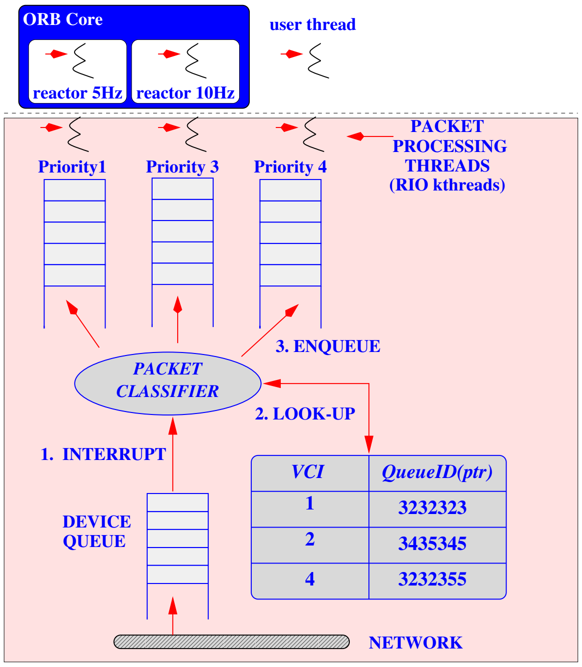

Figure 2: Early Demultiplexing in the RIO Subsystem

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 1.

Figure 1 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 1 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 received for the 10Hz connection are placed in RIO queue rQ_1. This queue is serviced periodically by RIO kthread rkt_1, which runs at real-time priority 110.

After it performs protocol processing, thread rkt_1 delivers the packet to TAO's ORB Core where it is processed by a periodic user thread with real-time priority 110. Likewise, the data packets received for the 5 Hz connection are processed periodically by RIO kthread rkt_2, which performs the protocol processing and passes the packets up to the user thread.

The remaining two connections handle best-effort network traffic. The best-effort RIO queue (rQ_3) is serviced by a relatively low-priority kthread rkt_3. Typically, this thread will be assigned a period and computation time[2] to bound the total throughput allowed on the best-effort connections, as describe in the following section.

The packet classifier in TAO's I/O subsystem can be configured to consult its real-time scheduling service to determine where the packet should be placed. This is required when multiple applications use a single VC, as well as when the link layer is not ATM. In these cases, it is necessary to identify packets and associate them with rates/priorities on the basis of higher-level protocol addresses like TCP port numbers. Moreover, the APIC device driver can be modified to search the TAO's run-time scheduler [1] in the ORB's memory. TAO's run-time scheduler maps TCP port numbers to rate groups in constant $O(1)$ time.

2.0.2 Schedule-driven Protocol Processing

Context: Many real-time applications require periodic I/O processing [12]. For example, avionics mission computers must process sensor data periodically to maintain accurate situational awareness [13]. If the mission computing system fails unexpectedly, corrective action must occur immediately.

Problem: Protocol processing of input packets in Solaris STREAMS is *demand-driven* [4], *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, such as when the incoming packet is destined for a thread with a priority lower than the currently executing thread. Thus, the ORB endsystem may fail to meet the QoS requirements of the higher priority thread.

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 because 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.

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

Implementing Schedule-driven protocol processing in RIO: The RIO subsystem uses a *thread pool* [14] concurrency model to implement its schedule-driven kthreads. Thread pools are appropriate for real-time ORB endsystems because they (1) amortize thread creation run-time overhead and (2) place an upper limit on the total percentage of CPU time used by RIO kthreads [15].

Figure 3 illustrates the thread pool model used in RIO. This

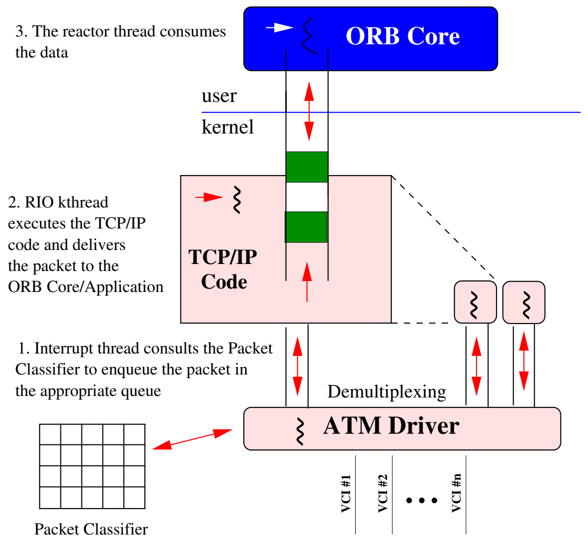

Figure 3: RIO Thread Pool Processing of TCP/IP with QoS Support

pool of protocol processing kthreads (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 thread is assigned a particular *rate*, based on computations from TAO's static scheduling service [1]. 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 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 2.0.2. 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 4 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.

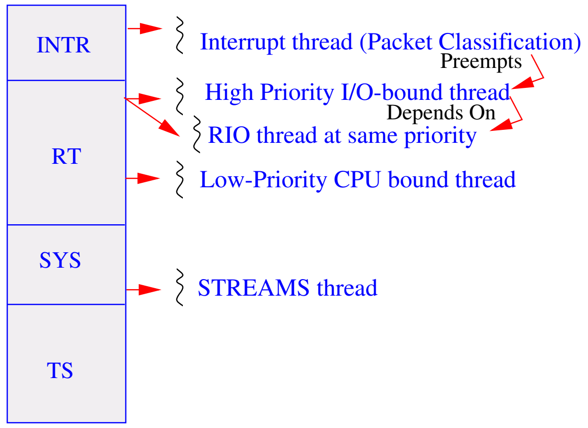

Figure 4: Alleviating Priority Inversion in TAO's ORB Endsystem

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.

2.0.3 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 are 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 1 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 priorities 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 high-priority threads to assure that its packets are processed before all other packets in the system.

3 Empirical Benchmarking Results

This section presents empirical results that show how the RIO subsystem decreases the upper bound on round-trip delay for latency-sensitive applications and provides periodic processing guarantees for bandwidth-sensitive applications. Other work [16] combines RIO and TAO to quantify the ability of the resulting ORB endsystem to support applications with real-time QoS requirements.

3.1 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 to an OC3 155 Mbps port on the ASX-1000. The testbed configuration is shown in Figure 5.

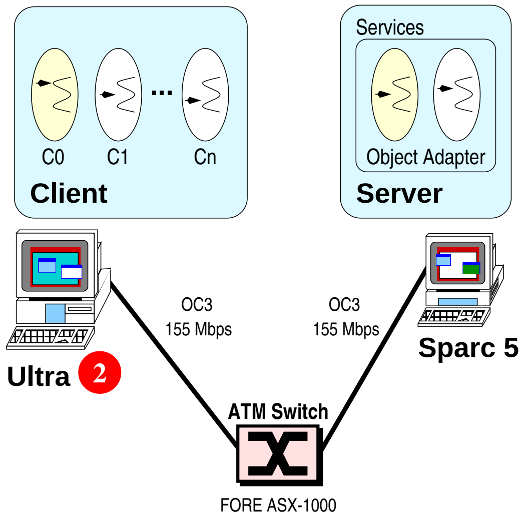

Figure 5: ORB Endsystem Benchmarking Testbed

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

Below, we present 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.

3.2.1 Benchmarking Configuration

Our experiments were performed using the testbed configuration shown in Figure 5. 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` [17] 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.

3.2.2 Measuring the Relative Cost of Using RIO kthreads

Benchmark design: This set of experiments measures the relative cost of using RIO kthreads versus interrupt threads (the default Solaris behavior) to process network protocols. The results show that it is relatively 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 described in Section 3.2.1:

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 6 and the results are summarized in the table below:

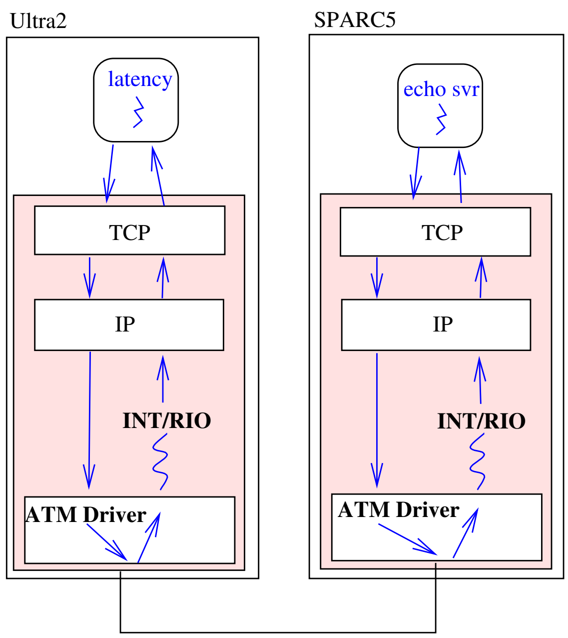

Figure 6: RIO kthread Test Configuration

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

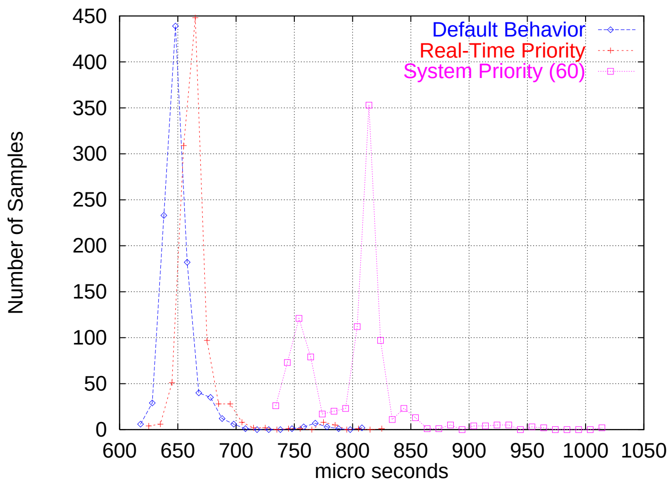

Figure 7: Latency Measurements versus Priority of kthreads

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.

Our 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 7 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.

3.2.3 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 because 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 3.2.1. 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 8 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 9. This figure highlights how RIO lowers the upper bound of the round-trip latency values.

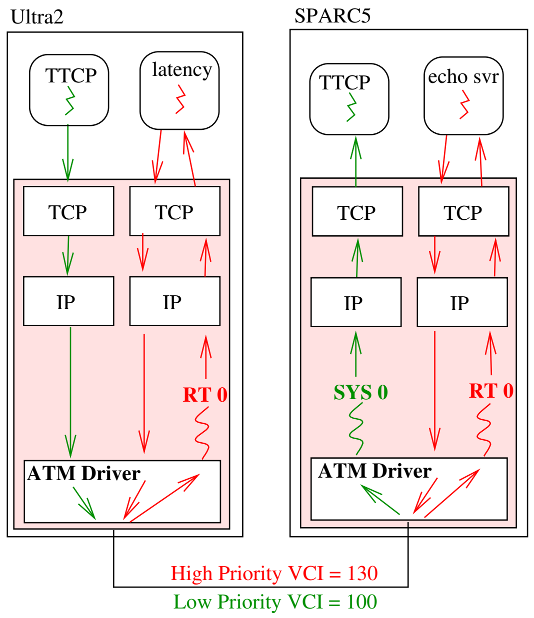

Figure 8: RIO Low-latency Benchmark Configuration

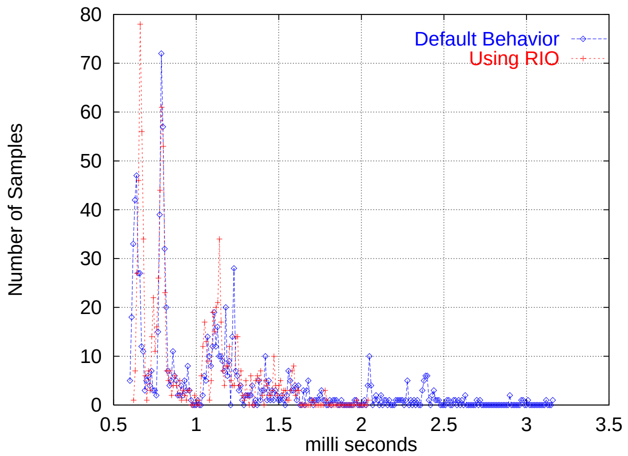

Figure 9: Latency with Competing Traffic

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 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. The results in this section demonstrate this: RIO achieved latencies no greater than 2.038 ms, even when the ORB endsystem was heavily loaded with best-effort data transfers.

Figure 10 shows the average bandwidth used by the modified `ttcp` applications during the experiment. The dip in throughput between sample numbers 10 and 20 occurred 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.

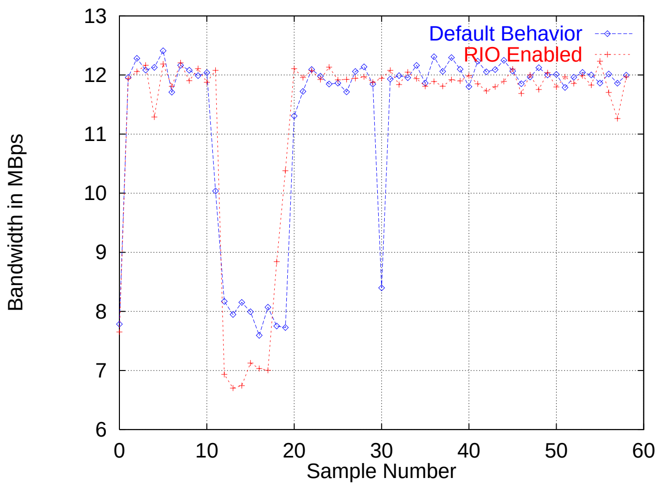

Figure 10: Bandwidth of Competing Traffic

3.2.4 Measuring Bandwidth Guarantees for Periodic Processing

Benchmark design: RIO can enforce bandwidth guarantees because it implements the schedule-driven protocol processing model described in Section 2.0.2. 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 2.0.1, 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, because 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. Figure 11 presents the results, showing the effect of the guaranteed connection on the best-effort connection.

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 [18] to be observed.

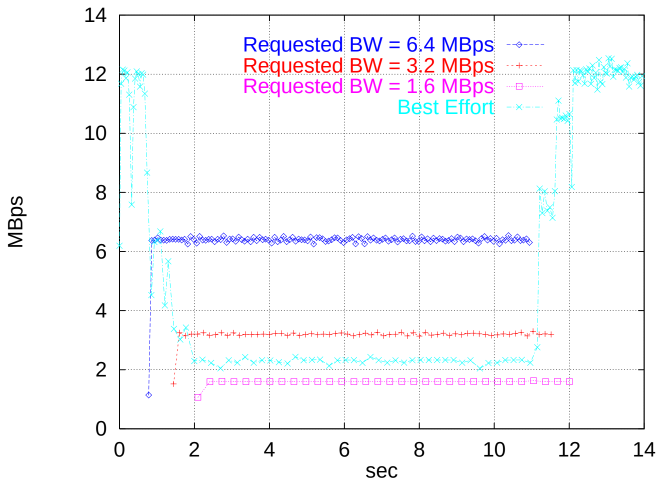

Figure 11: Bandwidth Guarantees in RIO

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.

3.3 Summary of Empirical Results

Our empirical results presented in Section 3 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, because 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.

4 Concluding Remarks

This paper focuses on the design and performance of a real-time I/O (RIO) subsystem that enhances the Solaris 2.5.1 kernel to enforce the QoS requirements of applications. RIO supports a vertically integrated, high-performance endstation from the network interface through software protocol processing to the user application threads. Three classes of I/O, best-effort, periodic and low latency, are supported in RIO.

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 the TAO real-time ORB into the RIO subsystem. This design minimizes sources of priority inversion that cause non-determinism.

RIO is designed to operate with high-performance interfaces such as the 1.2 Gbps ATM port interconnect controller (APIC) [3]. The APIC supports (1) shared memory pools between user and kernel space, (2) per-VC pacing, (3) two levels of priority queues, and (4) interrupt disabling on a per-VC bases. The current RIO prototype has been developed using a commercial Fore interface, as described in Section 3.

We learned the following lessons from the RIO project:

Vertical integration of endsystems is essential for end-to-end priority preservation: Conventional operating systems do not provide adequate support for the QoS requirements of distributed, real-time applications. By vertically integrating the I/O subsystem, the endsystem can reduce the duration of priority inversions and maximize overall system utilization. Consequently, effective throughput increases and upper bounds on latencies are reduced. Moreover, by combining RIO with the TAO rela-time ORB, QoS properties can be preserved end-to-end in a distributed object, real-time environment [19].

Schedule-driven protocol processing reduces jitter significantly: After integrating RIO with TAO, we measured a significant reduction in average latency and jitter. Moreover, the latency and jitter of low-priority traffic were not affected adversely. Our results illustrate how configuring asynchronous protocol processing [20] strategies in the Solaris kernel can provide significant improvements in ORB endsystem behavior, compared with the conventional Solaris I/O subsystem. As a result of our RIO enhancements to Solaris, TAO is the first ORB to support end-to-end QoS guarantees over ATM/IP networks [2].

Input livelock is a dominant source of ORB endsystem non-determinism: During the development and experimentation of RIO it became obvious that the dominant source of non-determinism was *receive livelock*. Priority inversion resulting from processing all input packets at interrupt context is unacceptable for many real-time applications. Using RIO kthreads for input packet processing yielded the largest gain in overall system predictability.

The TAO and RIO integration focused initially on statically scheduled applications with deterministic QoS requirements. we have subsequently extended the TAO ORB endsystem to support dynamic scheduling [21] 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.

The RIO research effort is currently directed toward integration with other platforms and in providing a standardized API. We are developing a pluggable protocols [22] framework for TAO that hides platform dependencies and extends RIO's functionality. TAO's pluggable protocols framework supports the addition of new messaging and transport protocols. Within this framework are (1) connection concurrency strategies, (2) endsystem/network resource reservation protocols, (3) high-performance techniques, such as zero-copy I/O, shared memory pools, periodic I/O, and interface pooling, (4) enhancement of underlying communications protocols, *e.g.*, provision of a reliable byte-stream protocol over ATM, and (5) tight coupling between the ORB and efficient user-space protocol implementations, such as Fast Messages [23]

The TAO research effort has influenced the OMG Realtime CORBA specification [24], which was recently adopted as a CORBA standard. We continue to track the process of this standard and to contribute to its evolution.

References

[1] 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.

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

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

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

[5] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems*, To appear 1999.

[6] 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.

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

[8] 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.

[9] 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.

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

[11] 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.

[12] R. Gopalakrishnan and G. M. Parulkar, "Efficient Quality of Service Support in Multimedia Computer Operating Systems," Tech. Rep. 94-26, Dept. of Computer Science, Washington University in St. Louis, 1994.

[13] 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.

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

[15] 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.

[16] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of RIO – A Real-time I/O Subsystem for ORB Endsystems," in *Submitted to the International Symposium on Distributed Objects and Applications (DOA'99)*, (Edinburgh, Scotland), OMG, Sept. 1999.

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

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

[19] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the 5^{th} Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[20] 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.

[21] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 1999, to appear.

[22] F. Kuhns, C. O'Ryan, D. C. Schmidt, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware," Department of Computer Science, Technical Report WUCS-99-12, Washington University, St. Louis, 1999.

[23] M. Lauria, S. Pakin, and A. Chien, "Efficient Layering for High Speed Communication: Fast Messages 2.x.," in *Proceedings of the 7th High Performance Distributed Computing (HPDC7) conference*, (Chicago, Illinois), July 1998.

[24] Object Management Group, *Realtime CORBA 1.0 Joint Submission*, OMG Document orbos/98-12-05 ed., December 1998.