An Empirical Evaluation of OS Support for Real-time CORBA Object Request Brokers

David L. Levine, Sergio Flores-Gaitan, and Douglas C. Schmidt
{levine,sergio,schmidt}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA*

Submitted to the Real-Time Technology and Applications Symposium (RTAS), Vancouver, British Columbia, Canada, June 2–4, 1999.

Abstract

There is increasing demand to extend Object Request Broker (ORB) middleware to support distributed applications with stringent real-time requirements. However, lack of proper OS support can yield substantial inefficiency and unpredictability for ORB middleware. This paper provides two contributions to the study of OS support for real-time ORBs.

First, we empirically compare and evaluate the suitability of real-time operating systems, VxWorks and LynxOS, and general-purpose operating systems with real-time extensions, Windows NT, Solaris, and Linux, for real-time ORB middleware. While holding the hardware and ORB constant, we vary the operating system and measure platform-specific variations, such as latency, jitter, operation throughput, and CPU processing overhead. Second, we describe key areas where these operating systems must improve to support predictable, efficient, and scalable ORBs.

Our findings illustrate that general-purpose operating systems like Windows NT and Solaris are not yet suited to meet the demands of applications with stringent QoS requirements. However, LynxOS does enable predictable and efficient ORB performance, thereby making it a compelling OS platform for real-time CORBA applications. Linux provides good raw performance, though it is not a real-time operating system. Surprisingly, VxWorks does not scale robustly. In general, our results underscore the need for a measure-driven methodology to pinpoint sources of priority inversion and non-determinism in real-time ORB endsystems.

Keywords: Real-time CORBA Object Request Broker, QoS-enabled OO Middleware, Performance Measurements

*This work was supported in part by Boeing, CDI/GDIS, DARPA contract 9701516, Lucent, Motorola, NSF grant NCR-9628218, Siemens, and US Sprint.

1 Introduction

Next-generation distributed real-time applications, such as teleconferencing, avionics mission computing, and process control, require endsystems that can provide statistical and deterministic quality of service (QoS) guarantees for latency [1], bandwidth, and reliability [2]. The following trends are shaping the evolution of software development techniques for these distributed real-time applications and endsystems:

Increased focus on middleware and integration frameworks: There is a trend in real-time R&D projects away from developing real-time applications from scratch to *integrating* applications using reusable components based on object-oriented (OO) middleware [3]. The objective of middleware is to increase quality and decrease the cycle-time and effort required to develop software by supporting the integration of reusable components implemented by different suppliers.

Increased focus on QoS-enabled components and open systems: There is increasing demand for remote method invocation and messaging technology to simplify the collaboration of open distributed application components [4] that possess deterministic and statistical QoS requirements. These components must be customizable to meet the functionality and QoS requirements of applications developed in diverse contexts.

Increased focus on standardizing and leveraging real-time COTS hardware and software: To leverage development effort and reduce training, porting, and maintenance costs, there is increasing demand to exploit the rapidly advancing capabilities of standard common-off-the-shelf (COTS) hardware and COTS operating systems. Several international standardization efforts are currently addressing QoS-related issues associated with COTS hardware and software.

One particularly noteworthy standardization effort has yielded the OMG CORBA specification [5]. CORBA is OO middleware software that allows clients to invoke operations on objects without concern for where the objects reside, what

language the objects are written in, what OS/hardware platform they run on, or what communication protocols and networks are used to interconnect distributed objects [6].

There has been recent progress towards standardizing CORBA for real-time [7] and embedded [8] systems. Several OMG groups, most notably the Real-Time Special Interest Group (RT SIG), are actively investigating standard extensions to CORBA to support distributed real-time applications. The goal of standardizing real-time CORBA is to enable real-time applications to interwork throughout embedded systems and heterogeneous distributed environments, such as the Internet.

However, developing, standardizing, and leveraging distributed real-time ORB middleware remains hard, notwithstanding the significant efforts of the OMG RT SIG. There are few successful examples of standard, widely deployed distributed real-time ORB middleware running on COTS operating systems and COTS hardware. Conventional CORBA ORBs are generally unsuited for performance-sensitive, distributed real-time applications due to their (1) lack of QoS specification interfaces, (2) lack of QoS enforcement, (3) lack of real-time programming features, and (4) overall lack of performance and predictability [9].

Although some operating systems, networks, and protocols now support real-time scheduling, they do not provide integrated end-to-end solutions [10]. Moreover, relatively little systems research has focused on strategies and tactics for real-time ORB endsystems. For instance, QoS research at the network and OS layers is only beginning to address key requirements and programming models of ORB middleware [11].

Historically, research on QoS for high-speed networks, such as ATM, has focused largely on policies for allocating virtual circuit bandwidth [12]. Likewise, research on real-time operating systems has focused largely on avoiding priority inversions in synchronization and dispatching mechanisms for multi-threaded applications [13]. An important open research topic, therefore, is to determine how best to map the results from QoS work at the network and OS layers onto the OO programming model familiar to many developers and users of ORB middleware.

Our prior research on CORBA middleware has explored several dimensions of real-time ORB endsystem design including static [10] and dynamic [14] real-time scheduling, real-time request demultiplexing [15], real-time event processing [16], real-time I/O subsystems [17], real-time ORB Core connection and concurrency architectures [18], real-time IDL compiler stub/skeleton optimizations [19], and performance comparisons of various commercial ORBs [20]. This paper focuses on a previously unexamined point in the real-time ORB endsystem design space: *the impact of OS performance and predictability on ORB performance and predictability*. A companion paper [21] covers additional aspects of ORB/OS performance and predictability.

The remainder of this paper is organized as follows: Section 2 outlines the architecture and design goals of TAO [10], which is a real-time implementation of CORBA developed at Washington University; Section 3 presents empirical results from systematically benchmarking the efficiency and predictability of TAO in several real-time operating systems, *i.e.*, VxWorks and LynxOS, and operating systems with real-time extensions, *i.e.*, Solaris, Windows NT, and Linux; Section 4 compares our research with related work; and Section 5 presents concluding remarks. For completeness, Appendix A explores the general factors that impact the performance of real-time ORB endsystems.

2 Overview of TAO

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. TAO supports the

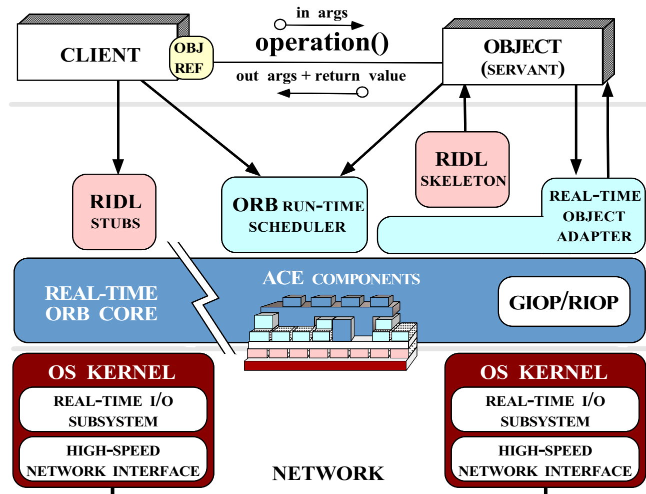

Figure 1: Components in the TAO Real-time ORB Endsystem

standard OMG CORBA reference model [5], with the following enhancements designed to overcome the shortcomings of conventional ORBs [18] 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 [22]. 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 [23].

Real-time Object Adapter: An Object Adapter associates servants with the ORB and demultiplexes incoming requests to servants. TAO's real-time Object Adapter [19] uses perfect hashing [24] and active demultiplexing [15] optimizations to dispatch servant operations in constant $O(1)$ time, regardless of the number of active connections, servants, and operations defined in IDL interfaces.

ORB Run-time Scheduler: A real-time scheduler [7] 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 [10] and dynamic [14] 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 [18] uses a multi-threaded, preemptive, priority-based connection and concurrency architecture [22] to provide an efficient and predictable CORBA IIOP protocol engine.

Real-time I/O subsystem: TAO's real-time I/O subsystem [25] 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 [12]. 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 [26], which implements core concurrency and distribution patterns [27] 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.

3 Real-time ORB Endsystem Performance Experiments

A real-time OS provides applications with mechanisms for priority-controlled access to hardware and software resources. Mechanisms commonly supported by real-time operating systems include real-time scheduling classes and real-time I/O subsystems. These mechanisms enable applications to specify their processing requirements and allow the OS to enforce the requested quality of service (QoS) usage policies.

This section presents the results of experiments conducted with a real-time ORB/OS benchmarking framework developed at Washington University and distributed with the TAO release.[1] This benchmarking framework contains a suite of test metrics that evaluate the effectiveness and behavior of real-time operating systems using various ORBs, including MT-Orbix, COOL, VisiBroker, CORBAplus, and TAO.

Our previous experience [15, 20, 28, 29, 18] measuring the performance of CORBA implementations showed that TAO supports efficient and predictable QoS better than other ORBs. Therefore, the experiments reported below focus solely on TAO.

3.1 Performance Results on Intel

3.1.1 Benchmark Configuration

Hardware overview: All of the tests in this section were run on a 450 MHz Intel Pentium II with 256 Mbytes of RAM. We focused primarily on a single CPU hardware configuration to factor out differences in network interface driver support and to isolate the effects of OS design and implementation on the end-to-end performance of ORB middleware and applications.

Operating system and compiler overview: We ran the ORB/OS benchmarks described in this paper on two real-time operating systems, VxWorks 5.3.1 and LynxOS 3.0.0, and three general-purpose operating systems with real-time extensions, Windows NT 4.0 Workstation with SP3, Solaris 2.6 for Intel, and RedHat Linux 5.1 (kernel version 2.0.34). A brief overview of each OS follows:

- **VxWorks:** VxWorks is a real-time OS that supports multi-threading and interrupt handling. By default, the VxWorks thread scheduler uses a priority-based first-in first-out (FIFO) preemptive scheduling algorithm, though it can be configured to support round-robin scheduling. In addition, VxWorks provides semaphores that implement a priority inheritance protocol [30].

[1] TAO and the ORB/OS benchmarks described in this paper are available at `www.cs.wustl.edu/~schmidt/TAO.html`.

• **LynxOS:** LynxOS is designed for complex hard real-time applications that require fast, deterministic response. LynxOS handles interrupts predictably by performing asynchronous processing at the priority of the thread that made the request. In addition, LynxOS supports priority inheritance, as well as FIFO and round-robin scheduling policies [31].

• **Windows NT:** Microsoft Windows NT is a general-purpose, preemptive, multi-threading OS designed to provide fast interactive response. Windows NT uses a round-robin scheduling algorithm that attempts to share the CPU fairly among all ready threads of the same priority. Windows NT defines a high-priority thread class called REAL-TIME_PRIORITY_CLASS. Threads in this class are scheduled before most other threads, which are usually in the NORMAL_PRIORITY_CLASS.

Windows NT is not designed as a deterministic real-time OS, however. In particular, its internal queueing is performed in FIFO order and priority inheritance is not supported for mutexes or semaphores. Moreover, there is no way to prevent hardware interrupts and OS interrupt handlers from preempting application threads [32].

• **Solaris:** Solaris is a general-purpose, preemptive, multi-threaded implementation of SVR4 UNIX and POSIX. It is designed to work on uniprocessors and shared memory symmetric multiprocessors [33]. Solaris provides a real-time scheduling class that attempts to provide worst-case guarantees on the time required to dispatch application or kernel threads executing in this scheduling class [34]. In addition, Solaris implements a priority inheritance protocol for mutexes and queues/dispatches threads in priority order.

• **Linux:** Linux is a general-purpose, preemptive, multi-threaded implementation of SVR4 UNIX, BSD UNIX, and POSIX. It supports POSIX real-time process and thread scheduling. The thread implementation utilizes processes created by a special `clone` version of `fork`. This design simplifies the Linux kernel, though it limits scalability because kernel process resources are used for each application thread.

We use the GNU g++ compiler with −O2 optimization on all but Windows NT, where we use Microsoft Visual C++ 6.0 with full optimization enabled, and VxWorks, where we use the GreenHills C++ version 1.8.8 compiler with −OL −OM optimization. For optimal performance our executables use static libraries.

Our tests on Solaris, LynxOS, Linux, and VxWorks were run with real-time, preemptive, FIFO thread scheduling. This provides strict priority-based scheduling to application threads. On Windows NT, tests were run in the Real-time priority class, which provides preemption capability over non-real-time threads. However, the scheduing is round-robin instead of FIFO[2]

ORB overview: Our benchmarking testbed is designed to isolate and quantify the impact of OS-specific variations on ORB endsystem performance and predictability. The ORB used for all the tests in this paper is version 1.0 of TAO [10], which is a high-performance, real-time ORB endsystem targeted for applications with deterministic and statistical QoS requirements, as well as "best-effort" requirements. TAO uses components in the ACE framework [35] to provide a common implementation framework on each OS platform in our benchmarking suite. Thus, the differences in performance reported in the following tests are due entirely to variations in OS internals, rather than ORB internals.

Benchmarking metric overview: The remainder of this section describes the results of the following benchmarking metrics we developed to evaluate the performance and predictability of VxWorks, LynxOS, Windows NT, Solaris, and Linux running TAO:

• **Latency and jitter:** This test measures ORB latency overhead and jitter for two-way operations. High latency directly affects application performance by degrading client/server communication. Large amounts of jitter complicate the computation of accurate worst-case execution time, which is necessary to schedule many real-time applications. This test and its results are presented in Section 3.1.2.

• **ORB/OS operation throughput:** This test provides an indication of the maximum operation throughput that applications can expect. It measures end-to-end two-way response when the client sends a request immediately after receiving the response to the previous request. This test and its results are presented in Section 3.1.3.

• **ORB/OS CPU processing overhead:** This test measures client/server ORB CPU processing overhead, which includes system call processing, protocol processing, and ORB request dispatch overhead. CPU processing overhead can significantly increase end-to-end latency. Overall system utilization can be reduced by excessive CPU processing per ORB operation. This test and its results is presented in Section 3.1.4.

3.1.2 Measuring ORB/OS Latency and Jitter

Terminology synopsis: ORB end-to-end *latency* is defined as the average amount of delay seen by a client thread from the time it sends the request to the time it completely receives the response from a server thread. *Jitter* is the variance of the

[2] Our high-priority client test results discussed below are not affected by using round-robin, because we have only one high priority thread. The low-priority results, however, do reflect round-robin scheduling on Windows NT.

latency for a series of requests. Increased latency directly impairs the ability to meet deadlines, whereas jitter makes real-time scheduling more difficult.

Overview of the latency and jitter metric: We computed the latency and jitter incurred by various clients and servers using the following configurations shown in Figure 2. The clients and servers in these tests were run on the same machine.

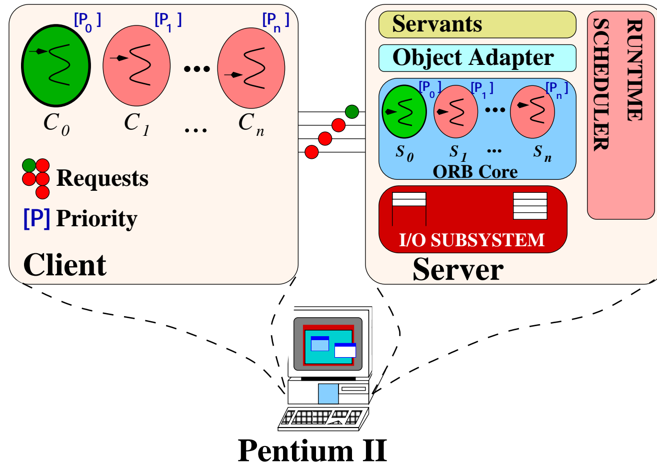

Figure 2: ORB Endsystem Latency and Jitter Test Configuration

- **Server configuration:** As shown in Figure 2, our testbed server consists of one servant S_0, with the highest real-time priority P_0, and servants $S_1 \ldots S_n$ that have lower thread priorities, each with a different real-time priority $P_1 \ldots P_n$. Each thread processes requests that are sent to its servant by client threads in the other process. Each client thread communicates with a servant thread that has an identical priority, *i.e.*, a client A with thread priority P_A communicates with a servant A that has thread priority P_A.

- **Client configuration:** Figure 2 shows how the benchmarking test uses clients from $C_0 \ldots C_n$. The highest priority client, *i.e.*, C_0, runs at the default OS real-time priority P_0 and invokes operations at 20 Hz, *i.e.*, it invokes 20 CORBA two-way calls per second. The remaining clients, $C_1 \ldots C_n$, have different lower OS thread priorities $P_1 \ldots P_n$ and invoke operations at 10 Hz, *i.e.*, they invoke 10 CORBA two-way calls per second.

All client threads have matching priorities with their corresponding servant thread. In each call, the client sends a value of type `CORBA::Octet` to the servant. The servant cubes the number and returns it to the client, which checks that the returned value is correct.

When the test program creates the client threads, these threads block on a barrier lock so that no client begins until the others are created and ready to run. When all client threads are ready to begin sending requests, the main thread unblocks them. These threads execute in an order determined by the real-time thread dispatcher.

Each low-priority client thread invokes 4,000 CORBA two-way requests at its prescribed rate. The high-priority client thread makes CORBA requests as long as there are low-priority clients issuing requests. Thus, high-priority client operations run for the duration of the test.

In an ideal ORB endsystem, the latency for the low-priority clients should rise gradually as the number of low-priority clients increased. This behavior is expected since the low-priority clients compete for OS and network resources as the load increases. However, the high-priority client should remain constant or show a minor increase in latency. In general, a significant amount of jitter complicates the computation of realistic worst-case execution times, which makes it hard to create a feasible real-time schedule.

Results of latency metrics: The average two-way response time incurred by the high-priority clients is shown in Figure 3. The jitter results are shown in Figure 4.

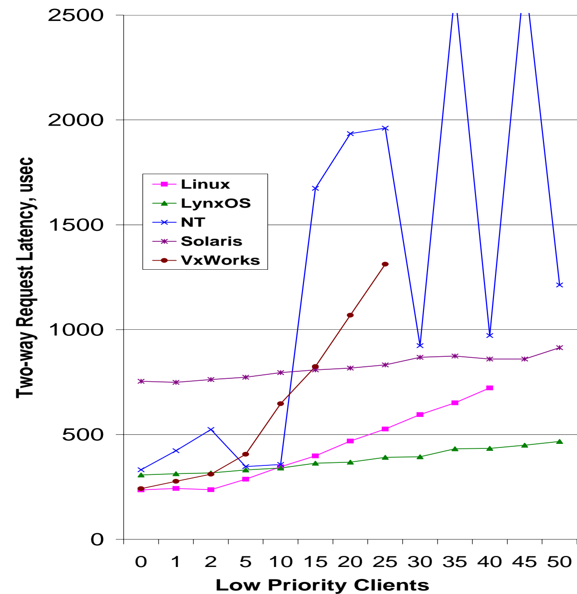

Figure 3: TAO's Latency for High-Priority Clients

The average two-way response time incurred by the low-priority clients is shown in Figure 5. The jitter results for the low-priority clients are shown in Figure 6. Our analysis of the results obtained for each OS platform are described below.

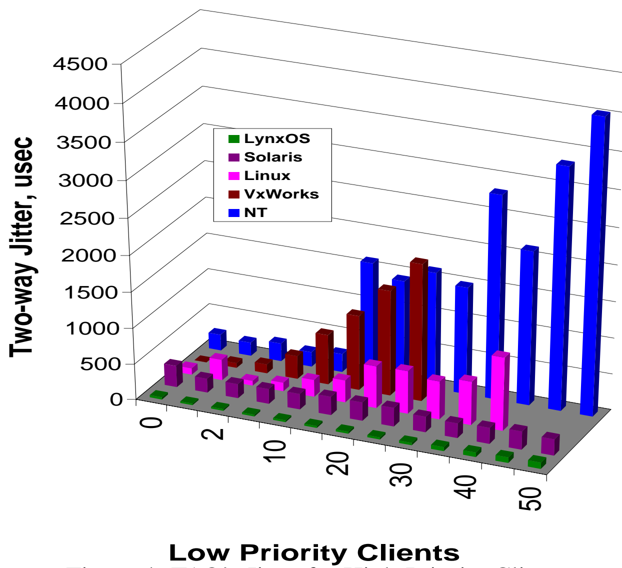

Figure 4: TAO's Jitter for High-Priority Clients

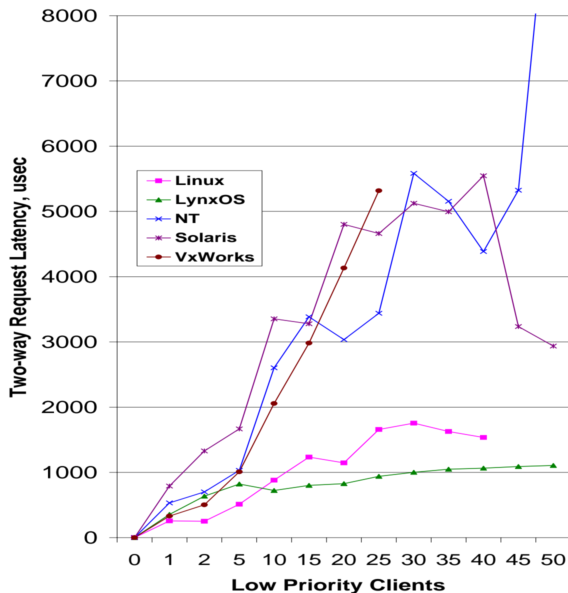

Figure 5: TAO's Latency for Low-Priority Clients

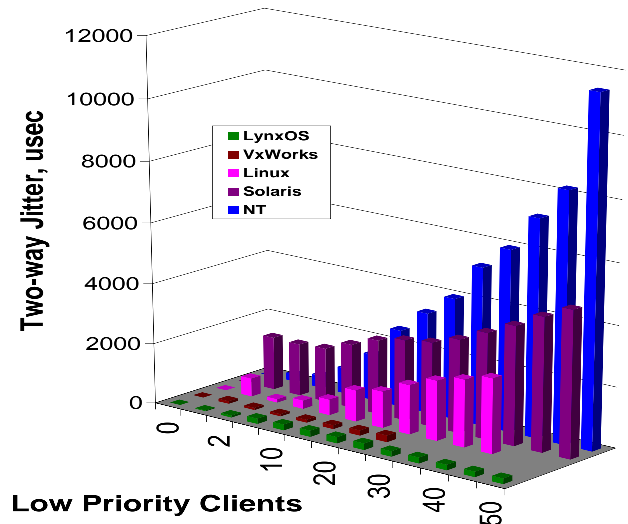

Figure 6: TAO's Jitter for Low-Priority Clients

• **Linux results:** The results on Linux are comparable to those on LynxOS, with a small number of low-priority clients. As shown in Figure 3, the high-priority latency ranged from 236 μsec with no low-priority clients to 722 μsec with 40 low-priority clients. Linux's performance is better than LynxOS with a very small number of low-priority clients. However, its rate of growth is higher, showing that its performance does not scale well as the number of low-priority clients increase. In addition, we could not run the test with more than 40 low-priority clients, because the default limit on open files was reached. Though it should be possible to increase that limit, the fact that Linux currently implements threads by using OS processes further indicates that it is not designed to scale up gracefully under heavy workloads.

• **LynxOS results:** LynxOS exhibited very low latency. Moreover, its I/O subsystem is closely integrated with its OS threads, which enables applications running over the ORB to behave predictably [36]. In addition, the interrupt handling mechanism used in LynxOS [36] is very responsive. Both high- and low-priority clients exhibited stable response times, yielding the low jitter shown in Figure 4. In addition, we observed low latency for the high-priority client, ranging from 307 μsec with no low-priority clients to 467 μsec with 50 low-priority clients, as shown in Figure 3.

• **Windows NT results:** The performance on Windows NT is best characterized as unpredictable. When the number of clients exceeds 10 the high-priority client latency varies dramatically and is higher than on other OS platforms. However, Windows NT is better than Solaris and VxWorks with 5 and 10 clients. This indicates good code optimization, though the

scheduling behavior of Windows NT does not currently appear well suited to demanding real-time systems. The result is confirmed by our ORB/OS overhead measurements in Section 3.1.4.

The low-priority request latency on Windows NT is comparable to that on Solaris and VxWorks, though it incurs more variation. In general, The jitter on Windows NT is the highest of the OS platforms tested when the number of low-priority clients exceeds 10. As with Solaris, the variation in behavior of Windows NT is problematic for systems that require predictable QoS.

• **Solaris results:** As shown in Figure 3, TAO's high-priority request latency on Solaris shows a trend of gradual growth from ~750 μsec with no low-priority clients to over 900 μsec with 50 low-priority clients. In general, the low-priority latency requests in Figure 5 grow with number of low-priority clients, though with a very large number of requests the latency drops dramatically.

Solaris' high-priority request jitter is relatively constant, as shown in Figure 4, at ~250 μsec. In contrast, the low-priority request jitter grows with the number of low-priority clients. Both the low- and high-priority request jitter are higher than those of the real-time operating systems, *i.e.*, LynxOS and VxWorks.

It appears that Solaris' relatively high jitter is due to the lack of integration between subsystems in the Solaris kernel. In particular, Solaris does not integrate its I/O processing with its CPU scheduling [17]. Therefore, it cannot ensure the availability of OS resources like I/O buffers and network bandwidth.

• **VxWorks results:** As shown in Figure 3, the high- and low-priority latencies for TAO on VxWorks are comparable to those of LynxOS and Linux for less than 5 clients. However, both latencies grow rapidly with the number of clients. With 15 clients, latencies are comparable or worse than those of Solaris. High-priority request jitter is very low on VxWorks, comparable to that on LynxOS. Low-priority jitter grows very rapidly with number of clients. These results indicate that VxWorks scales poorly on Intel platforms. Nevertheless, VxWorks does have stable behavior for a low range of clients, *i.e.*, 15 low-priority clients or less. We were not able to run with more than 30 low-priority threads due to exhaustion of an OS resource.

Result synopsis: In general, low latency and jitter are necessary for real-time operating systems to bound application execution times. However, general-purpose operating systems like Windows NT show erratic latency behavior, particularly under higher load. In contrast, LynxOS exhibited lower latency and better predictability, even under load. This stability makes it more suitable to provide QoS required by ORB middleware and applications. VxWorks offered low latency and jitter with low load, but its performance did not scale with increasing low-priority load.

3.1.3 Measuring ORB/OS Operation Throughput

Terminology synopsis: *Operation throughput* is the maximum rate at which operations can be performed. We measure the throughput of both two-way (request/response) and one-way (request without response) operations from client to server. This test indicates the overhead imposed by the ORB and OS on each operation.

Overview of the operation throughput metric: Our throughput test, called `IDL_Cubit`, uses a single-threaded client that issues an IDL operation at the fastest possible rate. The server performs the operation, which is to cube each parameter in the request. For two-way operations, the client thread waits for the response and checks that it is correct. Interprocess communication is performed via the network loopback interface since the client and server process run on the same machine.

The time required for cubing the argument on the server is small but non-zero. The client performs the same operation and compares it with the two-way operation result. The cubing operation itself is not intended to be a representative workload. However, many applications do rely on a large volume of small messages that each require a small amount of processing. Therefore, the `IDL_Cubit` benchmark is useful for evaluating ORB/OS overhead by measuring operation throughput.

We measure throughput for one-way and two-way operations using a variety of IDL data types, including `void`, `sequence`, and `struct` types. The one-way operation measurement eliminates the server reply overhead. The `void` data type instructs the server to not perform any processing other than that necessary to prepare and send the response, *i.e.*, it does not cube its input parameters. The `sequence` and `struct` data types exercise TAO's marshaling/demarshaling engine. The `Many struct` contains an `octet`, a `long`, and a `short`, along with padding necessary to align those fields.

Results of the operation throughput measurements: The throughput measurements are shown in Figure 7 and described below.[3]

• **Linux results:** Linux (along with VxWorks) exhibits the best operation throughput for simple data types, at 236 μsec for both `void` and `long`. This demonstrates that the

[3] To compare these results with other results in this paper, operation throughput is expressed in terms of *request latency*, in units of μsec per operation. Throughput is often expressed in terms of operations per second, however. Our results can be converted to those terms by simply dividing into 1,000,000.

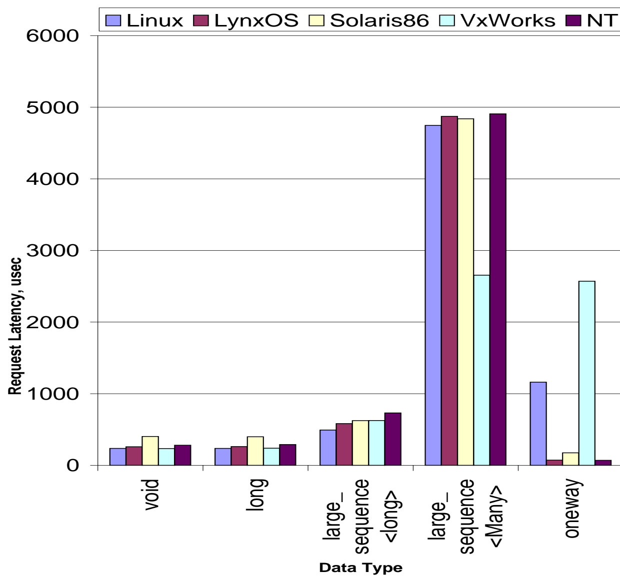

Figure 7: TAO's Operation Throughput for OS Platforms

cost of the cubing operation is negligible compared with the remainder of the operation processing. The one-way performance on Linux, however, was significantly higher than on most of the other platforms.

• **LynxOS results:** LynxOS offers consistently good performance: 259 μsec and 262 μsec for `void` and `long`, respectively. Similarly, it was close to Linux for large `sequence` performance. Its one-way performance was the best: $\sim$72 μsec.

• **Solaris results:** The throughput on Solaris is roughly in the middle of the platforms tested. For the simple data types, it requires about 400 μsec per request/response and 175 μsec for a one-way request.

• **VxWorks results:** VxWorks (along with Linux) offers the best operation throughput for `void` and `long` data types, of 234 and 239 μsec, respectively. It performs moderately well on large `sequence` of `longs`, relative to the other platforms. It performs the best, by far, on large `sequence` of `Many`. This may be due to the use of the different compiler than on most of the other platforms. The GreenHills compiler may optimize the data marshaling code differently than GNU g++ and Visual C++. VxWorks performs the worst on `oneway` operations, though it is not clear why.

• **Windows NT results:** Windows NT performed well for simple data types, at $\sim$310 μsec for `void` and 320 μsec for `long`. Its one-way performance was also good. However, it was at the slow end on large `sequence` processing.

Result synopsis: Operation throughput provides a measure of the overhead imposed by the ORB/OS. The `IDL_Cubit` test directly measures throughput for a variety of operation types and data types. Our measurements show that end-to-end performance depends dramatically on data type. In addition, the performance variation across platforms emphasizes the need for running benchmarks with different compilers, as well as other OS platform components such as network interface drivers.

3.1.4 Measuring ORB/OS CPU Processing Overhead

Terminology synopsis: ORB/OS processing overhead represents the amount of time the CPU spends (1) processing ORB requests, *e.g.*, marshaling/demarshaling in the IIOP communication protocol, request demultiplexing and dispatching, and data copying in the Object Adapter and (2) I/O request handling in the I/O subsystem of the OS kernel, *e.g.*, performing socket calls and processing network protocols.

Overview of CPU processing overhead metrics: CPU processing overhead is computed using a variant of the benchmark described in Section 3.1.2. The test in Section 3.1.2 measures the response time of clients' two-way CORBA requests. That response time includes servant processing time and overhead in the ORB and OS communication path. To determine the overhead, we developed a version of the latency test that contains the client and server in the same address space in separate threads.

There are two parts to this test: (1) invoking calls through ORB requests, *i.e.*, client/server and (2) invoking collocated calls directly on the object. Figure 8 and Figure 9 illustrate these two parts, respectively. The overhed of a collocated call is simply one virtual function call [19]. The difference in the latency of the two part reveals the amount of overhead. We express the overhead difference as a percentage of the collocated-call latency.

The test in Figure 9 has three threads: 1) the client thread, which issues operation requests, 2) the server thread, which processes CORBA requests, and 3) a "scavenger" thread, which picks up CPU cycles *not* used by the higher priority client and server threads running CORBA requests. This *scavenger thread* has system scheduling scope and runs at a lower OS thread priority than the CORBA thread. The scavenger thread should never run in these tests, because the client issues requests as rapidly as possible. The only activity in the system should be the client issuing requests, and the server handling those requests. If the "scavenger" thread does run, then the OS does not strictly obey real-time priorities.

We used a two-step process to compute the amount of ORB/OS overhead when making two-way requests. This process was applied to the following configurations of the utilization test, as shown in Figure 8 and Figure 9:

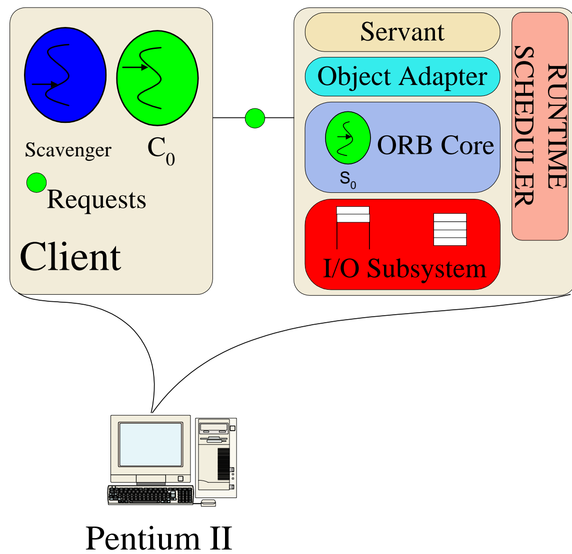

Figure 8: ORB Client/Server (C/S) Request Utilization Benchmark Configuration

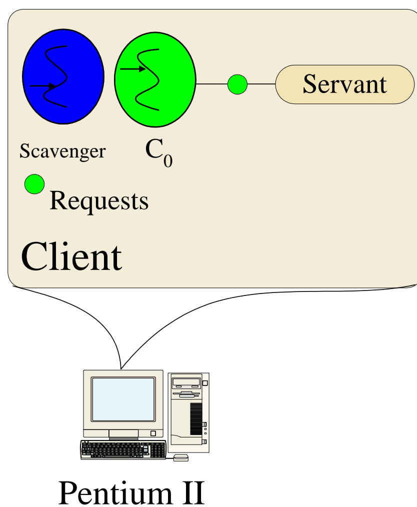

Figure 9: Collocated (CO) Utilization Benchmark Configuration

• **Client/Server (C/S) configuration:** Figure 8 illustrates invocations made to a servant located in the same address space at the client, but running in a separate thread. Running the servant in a different process, on platforms that support it, would needlessly incur additional process context switch overhead.

• **Collocated (CO) configuration:** Figure 9 illustrates collocated (CO) calls from client to servant object. The CO configuration incurs the overhead of only a single virtual function call[4]. This test provides a lower bound to compare with the C/S results.

We ran each test for a fixed number of calls, *i.e.*, 10,000,000. The total time for the C/S test is $T_{C/S}$. The total time spent in the collocated test is T_{CO}. The difference between the duration of these two tests, $T_{C/S} - T_{CO}$, yields the time spent performing ORB/OS processing. In an ideal ORB endsystem, the ORB and OS would incur minimal overhead, providing more stable response time and enabling the endsystem to meet QoS requirements.

Results of CPU processing overhead metrics: The utilization results for each OS platform are shown in Figure 10 and described below. The figure shows the mean, over 10 runs of 10,000,000 calls each, and plus/minus one standard deviation.

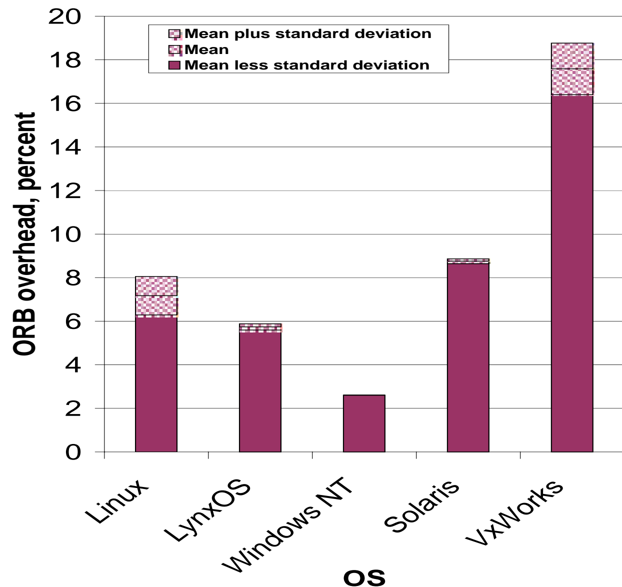

Figure 10: TAO CPU Utilization on Various OS Platforms

[4]We measured the time for a virtual function call at ∼20 nanoseconds on our test platform, with each of the tested OS's.

• **LynxOS results:** TAO on LynxOS exhibits CPU overhead of 5.73%. As shown in Figure 10, this value is relatively low for the platforms tested. The interaction of I/O events and other processing is optimized [36], minimizing overhead.

• **Linux results:** TAO on Linux exhibits moderate CPU utilization, 7.17%, but with relatively high standard deviation, as shown in Figure 10. The scavenger thread was able to run a small number of iterations, between 2.1 and 2.4 per 100 operations/function calls. This has a small effect on the measured overhead. But more importantly, it shows that thread priorities are not strictly obeyed on Linux. Linux was the only platform that displayed this anomaly.

• **Windows NT results:** The ORB/OS overhead on Windows NT is the lowest on the operating systems tested, at 2.61%, as shown in Figure 10. This indicates protocol processing overhead in Windows NT is low and that the compiler produces efficient code.

• **Solaris results:** The CPU overhead on Solaris of 8.75% is moderately high for the platforms tested, as shown in Figure 10. This overhead is sufficiently large to account for some of the latency discussed in Section 3.1.2.

• **VxWorks results:** TAO on VxWorks has 17.6% CPU utilization, the highest on the tested platforms as shown in Figure 10. We use a different compiler for VxWorks, though our experience has been that the GreenHills compiler usually produces very efficient code. Furthermore, the standard deviation of 6.75% of the mean is high, suggesting an OS rather than compiler inefficiency. The high overhead may contribute to the poor scalability on VxWorks shown in Section 3.1.2.

Result synopsis: We measured the overhead of the ORB and OS loopback communication path by comparing direct function calls to operations through the ORB and loopback interface. We observe that TAO on Windows NT displays very low overhead of 2.61%, TAO on LynxOS, Linux and Solaris show moderate overhead of 5.73 to 8.75%, and TAO on VxWorks has overhead of over 17 percent. Factors besides overhead affect real-time performance, in particular, it is important to eliminate priority inversion and non-determinism. The overhead test revealed that Linux does not strictly obey thread priority, even with preemptive scheduling.

3.2 Evaluation and Recommendations

The ORB/OS benchmarks presented in this paper illustrate the performance, priority inversion, and non-determinism incurred by five widely used operating systems running the same real-time ORB middleware. Since we use the same ORB, TAO, for our tests, the variation in results stems from differences in OS designs and implementations.

Based on our results, and our prior experience [15, 20, 28, 29, 18] measuring the performance of CORBA ORB endsystems, we propose the following recommendations to decrease non-determinism and limit priority inversion in operating systems that support real-time ORB middleware:

1. Real-time operating systems should provide low, deterministic context switching and mode switching latency: High context switch latency and jitter can significantly degrade the ORB efficiency and predictability of ORB endsystems [21]. High context switching overhead indicates that the OS spends too much time in the mechanics of switching from one thread to another. Thus, operating systems should tune their context switch mechanisms to provide a deterministic and minimal response context switch time.

In addition, system calls can incur a significant amount of overhead, particularly when switching modes between the kernel/application threads and vice versa. Since mode switching also yields significant overhead, operating systems should optimize system call overhead and minimize mode switches into the kernel. For instance, to reduce latency, operating systems should execute system calls in the calling thread's context, rather than in a dedicated I/O worker thread in the kernel [18].

2. Real-time operating systems should integrate the I/O subsystem with ORB middleware: Meeting the demands of real-time ORB endsystems requires a vertically integrated architecture that can deliver end-to-end QoS guarantees at multiple levels of a distributed system. For instance, to avoid *packet-based priority inversion*, the I/O subsystem level of the OS must process network packets in priority order, *e.g.*, as opposed to strict FIFO order [25].

To minimize packet-based priority inversion, an ORB endsystem must distinguish packets on the basis of their priorities and classify them into appropriate queues and threads. For instance, TAO's I/O subsystem exploits the *early demultiplexing* feature of ATM [12, 17]. Early demultiplexing detects the final destination of the packets based on the VCI field in the ATM cell header. The use of early demultiplexing alleviates packet-based priority inversion because packets need not be queued in FIFO order.

In addition, TAO's I/O subsystem supports *priority-based queuing*, where packets destined for higher-priority applications are delivered ahead of lower-priority packets that remain unprocessed in their queues. Support for *priority-based queuing* is also needed in the I/O subsystem. For instance, conventional implementations of network protocols in the I/O subsystems of Solaris and Windows NT process all packets at the same priority, regardless of the application thread destined to receive them.

For instance, Solaris 2.6 provides real-time scheduling but

not real-time I/O [34]. Therefore, Solaris is unable to guarantee the availability of resources like I/O buffers and network bandwidth. Moreover, the scheduling performed by the I/O subsystem is not integrated with other OS resource management strategies.

3. Real-time operating systems should support QoS specification and enforcement: Real-time applications often specify QoS requirements, such as CPU processing requirements, in terms of computation time and period. TAO supports QoS specification and enforcement via a real-time I/O scheduling class [17] and real-time scheduling service [10] that supports periodic real-time applications [16].

The scheduling abstractions defined by real-time operating systems like VxWorks and LynxOS are relatively low-level. In particular, they do not support application-level QoS specifications. In addition, operating systems must enforce those QoS specifications. To accomplish this, an OS should allow applications to specify their QoS requirements using higher-level APIs. TAO's real-time I/O scheduling class and real-time scheduler service permits applications to do this. In general, the OS should allow priorities to be assigned to real-time I/O threads so that application QoS requests can be enforced.

In addition, operating systems should provide admission control, which allows the OS to either guarantee the specified computation time or refuse to accept the resource demands of the thread. Because admission control is exercised at run-time, this is usually necessary with dynamically scheduled systems.

In addition to supporting the specification of processing requirements, an OS must enforce end-to-end QoS requirements of ORB endsystems. Real-time ORBs cannot deliver end-to-end guarantees to applications without integrated I/O subsystem and networking support for QoS enforcement. In particular, transport mechanisms in the OS, *e.g.*, TCP and IP, provide features like adaptive retransmissions and delayed acknowledgements, that can cause excessive overhead and latency. Such overhead can lead to missed deadlines in real-time ORB endsystems.

Therefore, operating systems should provide optimized real-time I/O subsystems that can provide end-to-end bandwidth, latency, and reliability guarantees to middleware and distributed applications [12].

4. Real-time operating systems should provide better tools to determine sources of overhead: Solaris provides a number of general-purpose performance analysis tools like `Quantify`, UNIX `truss`, and `time`. `Quantify` precisely indicates the function and system call overhead of an application. `truss` shows number of calls and overhead of an application's system calls. The `time` program shows the CPU utilization of the application and the time spent in user-mode and kernel-mode. These tools are valuable to pinpoint the sources of overhead incurred by an ORB endsystem and its applications.

However, there is also a need for tools that can pinpoint sources of overhead, priority inversion, and non-determinism. Such tools should provide (1) mechanisms to determine context switch overhead, *i.e.*, precisely determine the number and duration of context switches incurred by a task, (2) code profilers, *e.g.*, to determine number of system calls, and duration, and (3) high-resolution timers, to allow fine-grained latency measurements.

Certain real-time operating systems, such as LynxOS, provide insufficient tools to pinpoint sources of OS overhead. It is understandable, to a certain degree, that OS implementors optimize the OS by excluding support of performance measurement mechanisms. For instance, OS implementors may do this to eliminate overhead caused by adding performance counters. Nevertheless, a profiling/debugging mode should be available for the OS to enable performance measurements. This mode should be disabled at compilation time or run-time for normal operation and enabled when profiling applications.

5. Real-time operating systems should support priority inheritance protocols: Minimizing thread-based priority inversion is commonly handled with a *priority inheritance* protocol [34]. In this protocol, when a high-priority thread wants a resource, such as a mutex or semaphore, held by a low-priority thread, the low-priority thread's priority is boosted to that of the high-priority thread until the resource is released.

Several operating systems, *i.e.*, VxWorks, LynxOS, and Solaris, tested by our ORB/OS benchmarks implement priority inheritance protocols. In contrast, Linux and Windows NT do not support priority inheritance. Windows NT has a feature that temporarily boosts the priority of threads in processes in the NORMAL_PRIORITY_CLASS class. Thread priority boosting makes a foreground process react faster to user input. Threads in REALTIME_PRIORITY_CLASS processes that have a priority level in the class are never boosted by the OS.

Thread boosting can be beneficial for applications that require high responsiveness, such as applications that obtain user mouse and keyboard input. However, for applications that need high predictability and have stringent QoS requirements, thread boosting can be non-deterministic, which is detrimental to such applications. Therefore, operating systems that support real-time applications with stringent QoS requirements should alleviate priority inversion by providing mechanisms like priority inheritance protocols.

6. Real-time OS implementors should develop or adopt standard measurement-driven methodologies to identify sources of overhead, priority inversion, and

non-determinism: We believe that developing real-time ORB middleware requires a systematic, measurement-driven methodology to identify and alleviate sources of ORB endsystem and OS overhead, priority inversion, and non-determinism. The results presented in this paper are based on our experience developing, profiling, and optimizing avionics [16] and telecommunications [37] systems using OO middleware such as ACE [35] and TAO [10] developed at Washington University.

4 Related Work

An increasing number of research efforts are focusing on developing and optimizing real-time CORBA. The work presented in this paper is based on the TAO project [10]. For a comparison of TAO with related QoS projects see [38]. In this section, we briefly review related work on OS and middleware performance measurement.

lmbench benchmarks: The `lmbench` micro-benchmark suite evaluates important aspects of system performance [39]. It includes a context switching benchmark that measures the latency of switching between processes, rather than threads. The processes pass a token through a UNIX pipe. The time measured to pass the token includes the context switch time and the *pipe overhead*, *i.e.*, time measured to read from and write to the pipe. The pipe overhead is measured separately and subtracted from the total time to yield the context switch time. It adds the time the OS takes to fault the working set of the new process into the CPU cache. This models more closely what a user might see with large data-intensive processes. This approach provides a more realistic test environment than the suspend/resume and thread yield tests described in [21]. The reported `lmbench` context switch times for Linux is 6 μsec (lowest reported) and 101 μsec (highest reported), and for Solaris is 36 μsec (lowest reported) and 118 μsec (highest reported) on an Intel Pentium Pro 167 MHz. In contrast, we measure context switch time using different methods, on different OSs, and on different hardware. Our measurements in [21] yield context switch times for Linux of 2.60 μsec (lowest reported) and 9.72 μsec (highest reported), and for Solaris of 11.2 μsec (lowest reported) and 131.2 (highest reported), on an Intel Pentium II 450 MHz.

Rhealstone benchmarks: Different context switching metrics are used by the Rhealstone real-time benchmarking proposal [40]. It calls synchronous, non-preemptive context switching *task switching*. An example of task switching is the expiration of the current thread's time quantum. In contrast, *preemption* occurs when a context switch suspends a lower priority task in order to resume a higher priority task.

hbench benchmarks: The `hbench` benchmark suite [41], based on the `lmbench` package [39], measures the interactions between OS and hardware architecture, *i.e.*, Intel. It measures the scalability of operating system primitives, *e.g.*, process creation and cached file read, and hardware capabilities, *e.g.*, memory bandwidth and latency. In addition, `hbench` measures context switch latency, based on the original `lmbench` context switch test. The key difference is that `hbench` does not measure the overhead for cache conflicts or faulting in the processes data region. Their results yield 3 percent standard deviation for context switch time. Our measurements show standard deviations ranging from less than 0.15 percent to 8.0 percent.

Lai and Baker context switch time benchmarks: Lai and Baker utilize a similar approach to measure context switch time [42]. Again, they only measure the context switch time between processes, rather than threads. With one active process in the system, they measure context switch times of a small number to to under 100 μsec on a 100 MHz Pentium. These measurements are consistent with ours, between threads, of 3 to 128 μsec on a 200 MHz Pentium Pro.

In contrast to our evaluation, Lai and Baker focus on general purpose operating systems and a wide range of users, rather than real-time operating systems running applications that require responsiveness. They measured parameters including system call overhead, memory bandwidth, file system performance and raw network performance. In addition, Lai and Baker evaluate qualitative factors such as license agreements, ease of installation, and available support.

Distributed Hartstone benchmarks: The Distributed Hartstone (DHS) benchmark suite [43] is an extension of the Hartstone benchmark [44] to distributed systems. DHS quantitatively analyzes operating systems issues like the preemptability of the protocol stack and the effects of various queueing mechanisms. Moreover, DHS gauges the performance of real-time operating systems that must schedule harmonic and non-harmonic periodic activities. In addition, DHS measures the ability of the system to handle priority inversion.

The DHS benchmarks have two implementations of a protocol processing engine, a software interrupt based mechanism, that runs at a higher priority than any other user task. The second uses several prioritized worker threads to handle messages with different priorities. Their results show that the worker thread mechanism is 10 percent slower than the software interrupt version, but it increased preemptability.

The results of our tests show how the protocol processing engine of a determined OS does not account for messages with different priorities. They measure context switch times for threads that are in the same address space and in different address spaces on a SUN3/140. Their results show no difference in context switch time between these two measurements.

In contrast, we measure only for threads that are in the same address space (VxWorks only has one address space).

Our results show differences in context switch times between several OSs running on the same hardware. The DHS benchmark also measures response time of the communication subsystem, using two different sets of messages. One set uses different message priorities, and the other uses the same priority for all messages. Their results show better results for messages that use priority information from network packets, than a system which does FIFO processing of network packets.

Windows NT Real-time Applications Experiments: Gonzalez [32], *et al.*, ran a set of experiments on Windows NT to evaluate the ability of the OS to run applications with components that have real-time constraints. They simulate periodicity functionality in their prototype application by setting events on which different threads wait. They measure latency of various process/thread related Win32 API calls.

They report the 90th percentile of measurements, instead of average, since they are more interested in soft real-time, rather than hard real-time performance. In addition, they measure CPU overhead for system related tasks when the OS is idle. They reported 153 msec out of a second are used by system activities, *i.e.*, 15.3 percent. In contrast, we measure system activities under heavy workload. Our results are similar, yielding 12.8 percent. Furthermore, their experiments included measuring roundtrip latency for remote command from a Realvideo player at different frequencies (10, 20, and 33 Hz).

We also measured roundtrip latency and observed a similar unpredictable pattern in the behavior. We used 10 and 20 Hz frequencies. Their observations were that the variance was higher in lower priority processes. In the Windows NT REAL_TIME priority class, they observed significantly less variance, even though the latency was similar.

5 Concluding Remarks

There is a significant interest in developing high performance, real-time systems using ORB middleware like CORBA to lower development costs and decrease time-to-market. The flexibility, reusability, and platform-independence offered by CORBA makes it attractive for use in real-time systems. However, meeting the stringent QoS requirements of real-time systems requires more than just specifying QoS via IDL interfaces [10]. Therefore, it is essential to develop integrated ORB endsystems that can enforce application QoS guarantees end-to-end.

This paper illustrates the characteristics that the OS component in an ORB endsystem should provide to support real-time applications. By holding the hardware and ORB implementation constant and systematically varying the underlying OS, we demonstrated empirically the extent to which operating systems are (and are not) capable of meeting real-time application QoS requirements.

Our benchmarks revealed that certain operating systems do not behave optimally under high load conditions. For instance, general-purpose operating systems, such as Windows NT and Solaris, exhibit priority inversion and non-determinism. Thus, these operating systems may be unsuitable for some distributed, real-time applications and ORB middleware with deterministic QoS requirements. Meeting these demands requires operating systems to increase predictability and responsiveness, as well as reduce priority inversion.

LynxOS consistently performed better than other operating systems in our ORB/OS benchmarking testsuite. LynxOS provides low latency and stable predictability by reducing interrupt overhead and performing most asynchronous processing at the priority of the process that made the request.

In general, real-time operating systems need not necessarily exhibit lower latency than general-purpose operating systems. It is important, however, that real-time operating systems provide deterministic behavior. For instance, VxWorks showed low variability in the latency measurements when there was little contention for system resources. However, it did not scale up gracefully. In particular, high-priority client jitter rose progressively with an increasing number of low-priority clients.

To support the development of real-time ORB applications with stringent QoS requirements, operating systems must eliminate sources of non-determinism and priority inversion and fully integrate the various subsystems to provide end-to-end QoS guarantees. In particular, the I/O subsystem is an important factor for determining a bound on responsiveness, which is crucial for certain types of real-time applications [17].

Many real-time applications can benefit from flexible and open distributed architectures, such as those defined by CORBA [5]. Our previous work [18] has shown that conventional CORBA ORBs have limitations that make them inadequate for real-time middleware with stringent QoS requirements. TAO has overcome these limitations, however, through careful design and implementation. Therefore, our research demonstrates that real-time support is possible in CORBA ORBs when they are run over well-designed real-time operating systems.

Acknowledgments

We gratefully acknowledge the support and direction of the Boeing Principal Investigator, Bryan Doerr. In addition, we would like to thank Steve Kay for comments on this paper.

References

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

[2] S. Landis and S. Maffeis, “Building Reliable Distributed Systems with CORBA,” *Theory and Practice of Object Systems*, Apr. 1997.

[3] R. Johnson, “Frameworks = Patterns + Components,” *Communications of the ACM*, vol. 40, Oct. 1997.

[4] Z. Deng and J. W.-S. Liu, “Scheduling Real-Time Applications in an Open Environment,” in *Proceedings of the 18th IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, Dec. 1997.

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

[6] S. Vinoski, “CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments,” *IEEE Communications Magazine*, vol. 14, February 1997.

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

[8] Object Management Group, *Minimum CORBA - Request for Proposal*, OMG Document orbos/97-06-14 ed., June 1997.

[9] 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.

[10] 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.

[11] A. Campbell and K. Nahrstedt, *Building QoS into Distributed Systems*. London: Chapman & Hall, 1997. Proceedings of the IFIP TC6 WG6.1 Fifth International Workshop on Quality of Service (IWQOS ’97), 21-23 May 1997, New York.

[12] 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.

[13] R. Rajkumar, L. Sha, and J. P. Lehoczky, “Real-Time Synchronization Protocols for Multiprocessors,” in *Proceedings of the Real-Time Systems Symposium*, (Huntsville, Alabama), December 1988.

[14] 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.

[15] 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.

[16] 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.

[17] D. C. Schmidt, F. Kuhns, R. Bector, and D. L. Levine, “The Design and Performance of RIO – A Real-time I/O Subsystem for ORB Endsystems,” in *Submitted to the* 5th *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[18] 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* 4th *IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[19] 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* 5th *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[20] 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.

[21] D. Levine, S. Flores-Gaitan, and D. C. Schmidt, “Measuring OS Support for Real-time CORBA ORBs,” in *Proceedings of the* 4th *Workshop on Object-oriented Real-time Dependable Systems*, (Santa Barbara, CA), IEEE, January 1999.

[22] A. Gokhale and D. C. Schmidt, “Techniques for Optimizing CORBA Middleware for Distributed Embedded Systems,” in *Proceedings of INFOCOM ’99*, Mar. 1999.

[23] 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.

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

[25] D. C. Schmidt, R. Bector, D. Levine, S. Mungee, and G. Parulkar, “An ORB Endsystem Architecture for Statically Scheduled Real-time Applications,” in *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.

[26] 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.

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

[28] A. Gokhale and D. C. Schmidt, “The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks,” in *Proceedings of GLOBECOM ’96*, (London, England), pp. 50–56, IEEE, November 1996.

[29] A. Gokhale and D. C. Schmidt, “Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks,” *Transactions on Computing*, vol. 47, no. 4, 1998.

[30] Wind River Systems, “VxWorks 5.2 Web Page.” `http://www.wrs.com/products/html/vxwks52.html`, May 1998.

[31] Lynx Real-Time Systems, "LynxOS - Hard Real-Time OS Features and Capabilities." http://www.lynx.com/-products/ds_lynxos.html, Dec. 1997.

[32] K. Ramamritham, C. Shen, O. Gonzáles, S. Sen, and S. Shirgurkar, "Using Windows NT for Real-time Applications: Experimental Observations and Recommendations," in *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[33] 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.

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

[35] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the* 6th *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[36] W. Weinberg, "Lynx Patented Technology Speeds Handling of Hardware Events." http://www.lynx.com/news_and_-events/Patent_Exp.html, Sept. 1997.

[37] D. C. Schmidt, "A Family of Design Patterns for Application-level Gateways," *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, vol. 2, no. 1, 1996.

[38] 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," *Submitted to the Journal of Real-time Systems*, 1998.

[39] L. McVoy, "lmbench: Portable tools for performance analysis," in *Proceedings of the 1996 USENIX Technical Conference*, USENIX, January 1996.

[40] R. P. Kar and K. Porter, "Rhealstone: A Real-Time Benchmarking Proposal," *Dr. Dobbs Journal*, vol. 14, pp. 14–24, Feb. 1989.

[41] A. B. Brown and M. I. Seltzer, "Operating System Benchmarking in the Wake of Lmbench: Case Study of the Performance of NetBSD on the Intel Architecture," in *Proceedings of the 1997 Sigmetrics Conference*, June 1997.

[42] K. Lai and M. Baker, "A Performance Comparison of UNIX Operating Systems on the Pentium," in *Proceedings of the 1996 USENIX Technical Conference*, USENIX, January 1996.

[43] Clifford W. Mercer and Yutaka Ishikawa and Hideyuki Tokuda, "Distributed Hartstone Real-Time Benchmark Suite," in *Proceedings of the 10th International Conference on Distributed Computing Systems*, (Paris, France), May 1990.

[44] N. Weiderman, "Hartstone: Synthetic Benchmark Requirements for Hard Real-Time Applications," tech. rep., Software Engineering Institute, Carnegie Mellon University, June 1989.

[45] Z. D. Dittia, J. Jerome R. Cox, and G. M. Parulkar, "Design of the APIC: A High Performance ATM Host-Network Interface Chip," in *IEEE INFOCOM '95*, (Boston, USA), pp. 179–187, IEEE Computer Society Press, April 1995.

[46] N. C. Hutchinson and L. L. Peterson, "The *x*-kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, January 1991.

[47] Object Management Group, *Control and Management of A/V Streams Request For Proposals*, OMG Document telecom/96-08-01 ed., August 1996.

[48] A. Gokhale and D. C. Schmidt, "Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance," in *Hawaiian International Conference on System Sciences*, January 1998.

[49] J. C. Mogul and A. Borg, "The Effects of Context Switches on Cache Performance," in *Proceedings of the* 4th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Santa Clara, CA), ACM, Apr. 1991.

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

[51] D. L. Tennenhouse, "Layered Multiplexing Considered Harmful," in *Proceedings of the* 1st *International Workshop on High-Speed Networks*, May 1989.

A Factors Impacting Real-time ORB Endsystem Performance

Meeting the QoS needs of next-generation distributed applications requires much more than defining IDL interfaces or adding preemptive real-time scheduling into an OS. It requires a vertically and horizontally integrated *ORB endsystem* architecture that can deliver end-to-end QoS guarantees at multiple levels throughout a distributed system [10]. The key levels in an ORB endsystem include the network adapters, OS I/O subsystems, communication protocols, ORB middleware, and higher-level services shown in Figure 1.

For completeness, Section A.1 briefly outlines the general sources of performance overhead in ORB endsystems. Section A.2 describes the key sources of priority inversion and non-determinism that affect the predictability and utilization of real-time ORB endsystems. Section 3 illustrates quantitatively how OS characteristics like context switching, synchronization, and system call overhead impact ORB performance and predictability.

A.1 General Sources of ORB Endsystem Performance Overhead

Our experience [15, 20, 28, 29] measuring the throughput and latency of CORBA implementations indicates that performance overheads in real-time ORB endsystems arise from inefficiencies in the following components:

1. Network connections and network adapters: These endsystem components handle heterogeneous network connections and bandwidths, which can significantly increase latency and cause variability in performance. Inefficient design of network adapters can cause queueing delays and lost packets [45], which are unacceptable for certain types of real-time systems.

2. Communication protocol implementations and integration with the I/O subsystem and network adapters: Inefficient protocol implementations and improper integration with I/O subsystems can adversely affect endsystem performance. Specific factors that cause inefficiencies include the protocol overhead caused by flow control, congestion control, retransmission strategies, and connection management. Likewise, lack of proper I/O subsystem integration yields excessive data copying, fragmentation, reassembly, context switching, synchronization, checksumming, demultiplexing, marshaling, and demarshaling overhead [46].

3. ORB transport protocol implementations: Inefficient implementations of ORB transport protocols such as the CORBA Internet inter-ORB protocol (IIOP) [5] and Simple Flow Protocol (SFP) [47] can cause significant performance overhead and priority inversion. Specific factors responsible for these inversions include improper connection management strategies, inefficient sharing of endsystem resources, and excessive synchronization overhead in ORB protocol implementations.

4. ORB core implementations and integration with OS services: An improperly designed ORB Core can yield excessive memory accesses, cache misses, heap allocations/deallocations, and context switches [48]. In turn, these factors can increase latency and jitter, which is unacceptable for distributed applications with deterministic real-time requirements. Specific ORB Core factors that cause inefficiencies include data copying, fragmentation/reassembly, context switching, synchronization, checksumming, socket demultiplexing, timer handling, request demultiplexing, marshaling/demarshaling, framing, error checking, connection and concurrency architectures. Many of these inefficiencies are similar to those listed in bullet 2 above. Since they occur at the user-level rather than at the kernel-level, however, ORB implementers can often address them more readily.

Figure 11 pinpoints where the various factors outlined above impact ORB performance and where optimizations can be applied to reduce key sources of ORB endsystem overhead, priority inversion, and non-determinism. Below, we describe the components in an ORB endsystem that are chiefly responsible for priority inversion and non-determinism.

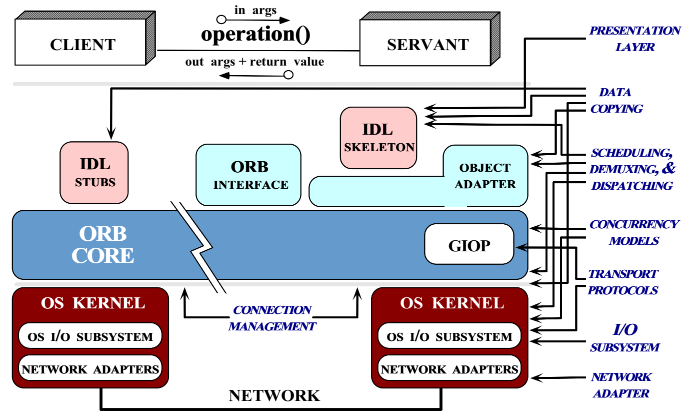

Figure 11: Optimizing Real-time ORB Endsystem Performance

A.2 Sources of Priority Inversion and Non-determinism in ORB Endsystems

Minimizing priority inversion and non-determinism is important for real-time operating systems and ORB middleware in order to bound application execution times. In ORB endsystems, priority inversion and non-determinism generally stem from resources that are shared between multiple threads or processes. Common examples of shared ORB endsystem resources include (1) TCP connections used by a CORBA IIOP protocol engine, (2) threads used to transfer requests through client and server transport endpoints, (3) process-wide dynamic memory managers, and (4) internal ORB data structures like connection tables for transport endpoints and demultiplexing maps for client requests. Below, we describe key sources of priority inversion and non-determinism in conventional ORB endsystems.

A.2.1 The OS I/O Subsystem

An I/O subsystem is the component in an OS responsible for mediating ORB and application access to low-level network and OS resources, such as device drivers, protocol stacks, and the CPU(s). Key challenges in building a high-performance, real-time I/O subsystem are (1) to minimize context switching and synchronization overhead and (2) to enforce QoS guarantees while minimizing priority inversion and non-determinism [17].

A context switch is triggered when an executing thread relinquishes the CPU it is running on voluntarily or involuntarily. Depending on the underlying OS and hardware platform, a context switch may require hundreds of instructions to flush register windows, memory caches, instruction pipelines, and translation look-aside buffers [49]. Synchronization overhead arises from locking mechanisms that serialize access to shared resources like I/O buffers, message queues, protocol connec-

tion records, and demultiplexing maps used during protocol processing in the OS and ORB.

The I/O subsystems of general-purpose operating systems, such as Solaris and Windows NT, do not perform preemptive, prioritized protocol processing [25]. Therefore, the protocol processing of lower priority packets is *not* deferred due to the arrival of higher priority packets. Instead, incoming packets are processed by their arrival order, rather than by their priority.

For instance, in Solaris if a low-priority request arrives immediately before a high priority request, the I/O subsystem will process the lower priority packet and pass it to an application servant before the higher priority packet. The time spent in the low-priority servant represents the degree of priority inversion incurred by the ORB endsystem and application.

[25] examines key issues that cause priority inversion in I/O subsystems and describes how TAO's real-time I/O subsystem avoids many forms of priority inversion by co-scheduling pools of user-level and kernel-level real-time threads. The results in Section 3 illustrate the extent to which the priority inversion and non-determinism in an OS affect ORB performance and predictability.

A.2.2 The ORB Core

An ORB Core is the component in CORBA that implements the General Inter-ORB Protocol (GIOP) [5], which defines a standard format for interoperating between (potentially heterogeneous) ORBs. The ORB Core establishes connections and implements concurrency architectures that process GIOP requests. The following discussion outlines common sources of priority inversion and non-determinism in conventional ORB Core implementations.

Connection architecture: The ORB Core's *connection architecture*, *i.e.*, how requests are mapped onto network connections, has a major impact on real-time ORB behavior. Therefore, a key challenge for developers of real-time ORBs is to select a connection architecture that can utilize the transport mechanisms of an ORB endsystem efficiently and predictably.

Conventional ORB Cores typically share a single multiplexed TCP connection for all object references to servants in a server process that are accessed by threads in a client process. The goal of connection multiplexing is to minimize the number of connections open to each server, *e.g.*, to improve server scalability over TCP. However, connection multiplexing can yield substantial packet-level priority inversions and synchronization overhead [18]. Therefore, it should be avoided for most real-time systems.

Concurrency architecture: The ORB Core's *concurrency architecture*, *i.e.*, how requests are mapped onto threads, also has a substantial impact on its real-time behavior. Therefore, another key challenge for developers of real-time ORBs is to select a concurrency architecture that can effectively share the aggregate processing capacity of an ORB endsystem and its application operations in one or more threads.

ORB Core concurrency architectures often use *thread pools* [50] to select a thread to process an incoming request. However, conventional ORBs do not provide programming interfaces that allow real-time applications to assign the priority of threads in this pool. Therefore, the priority of a thread in the pool is often inappropriate for the priority of the servant that ultimately executes the request. An improperly designed ORB Core increases the potential for, and duration of, priority inversion and non-determinism [18].

A.2.3 The Object Adapter

An Object Adapter is the component in CORBA that is responsible for demultiplexing incoming requests to servant operations that handle the request. A standard GIOP-compliant client request contains the identity of its object and operation. An object is identified by an object key which is an `octet sequence`. An operation is represented as a `string`. As shown in Figure 12, the ORB endsystem must perform the following demultiplexing tasks:

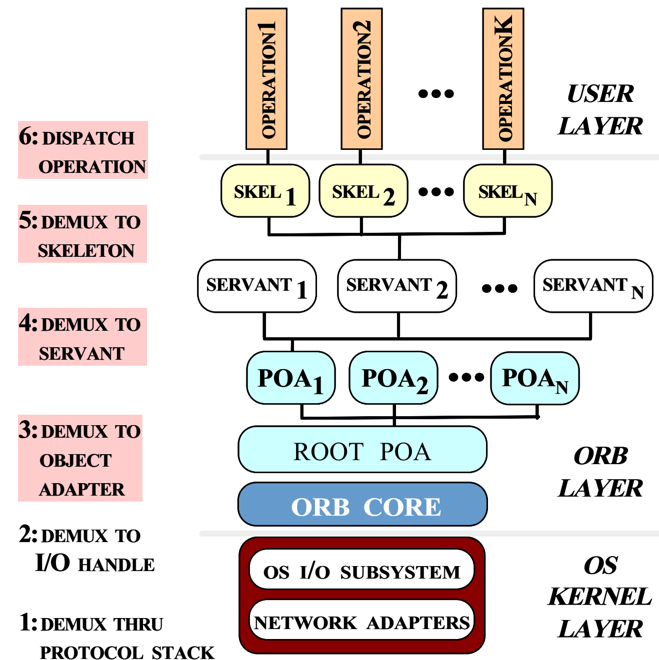

Figure 12: CORBA 2.2 Logical Server Architecture

Steps 1 and 2: The OS protocol stack demultiplexes the incoming client request multiple times, *e.g.*, from the network

interface card, through the data link, network, and transport layers up to the user/kernel boundary (*e.g.*, the socket) and then dispatches the data to the ORB Core.

Steps 3, and 4: The ORB Core uses the addressing information in the client's object key to locate the appropriate POA and servant. POAs can be organized hierarchically. Therefore, locating the POA that contains the servant can involve multiple demultiplexing steps through the hierarchy.

Step 5 and 6: The POA uses the operation name to find the appropriate IDL skeleton, which demarshals the request buffer into operation parameters and performs the upcall to code supplied by servant developers.

The conventional layered ORB endsystem demultiplexing implementation shown in Figure 12 is generally inappropriate for high-performance and real-time applications for the following reasons [51]:

Decreased efficiency: Layered demultiplexing reduces performance by increasing the number of internal tables that must be searched as incoming client requests ascend through the processing layers in an ORB endsystem. Demultiplexing client requests through all these layers is expensive, particularly when a large number of operations appear in an IDL interface and/or a large number of servants are managed by an Object Adapter.

Increased priority inversion and non-determinism: Layered demultiplexing can cause priority inversions because servant-level quality of service (QoS) information is inaccessible to the lowest-level device drivers and protocol stacks in the I/O subsystem of an ORB endsystem. Therefore, an Object Adapter may demultiplex packets according to their FIFO order of arrival. FIFO demultiplexing can cause higher priority packets to wait for an indeterminate period of time while lower priority packets are demultiplexed and dispatched [25].

Conventional implementations of CORBA incur significant demultiplexing overhead. For instance, [20, 29] show that conventional ORBs spend $\sim$17% of the total server time processing demultiplexing requests. Unless this overhead is reduced and demultiplexing is performed predictably, ORBs cannot provide uniform, scalable QoS guarantees to real-time applications.

Our prior work has focused on the impact the ORB Core [18] and Object Adapter [15] has on ORB priority inversion and non-determinism. Section 3 focuses on the impact of the OS and its I/O subsystem on the predictability and performance of ORB endsystems.