Measuring OS Support for Real-time CORBA ORBs

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*

Presented at the 4^{th} IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'99), Santa Barbara, California, January 27–29, 1999.

Abstract

This paper compares and evaluates 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 these operating systems and measure platform-specific variations in context switching overhead and priority inversions.

Our findings illustrate that general-purpose operating systems like Windows NT, Solaris, and Linux are not yet suited to meet the demands of applications with stringent QoS requirements. Although Linux provides good raw performance, its high jitter makes it unsuitable for real-time applications. Both LynxOS and VxWorks do enable predictable and efficient ORB performance, however, thereby making them suitable as OS platforms for real-time CORBA applications. In general, our results underscore the need for a measure-driven methodology to pinpoint sources of overhead and priority inversion in real-time ORB endsystems.

Keywords: Real-time Object-Oriented Systems, Operating System QoS Support, Real-time CORBA Object Request Broker

1 Introduction

There has been recent progress towards standardizing object-oriented (OO) middleware for real-time and embedded systems. In particular, the OMG is actively investigating standard extensions to CORBA to support distributed real-time applications [1]. The goal of standardizing real-time CORBA is to enable real-time applications to interwork throughout small footprint [2] embedded systems and heterogeneous distributed networks, such as the Internet.

Notwithstanding the significant efforts of the OMG real-time CORBA standardization effort, however, developing, standardizing, and leveraging distributed real-time ORB middleware remains hard. 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 [3].

Our prior research on CORBA middleware has explored several dimensions of real-time ORB endsystem design including static [4] and dynamic [5] real-time scheduling, real-time request demultiplexing [6], real-time event processing [7], real-time I/O subsystems [8], real-time ORB Core connection and concurrency architectures [9], real-time IDL compiler stub/skeleton optimizations [10], and performance comparisons of various commercial ORBs [11]. This paper presents our initial results 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*.

The remainder of this paper is organized as follows: Section 2 outlines the architecture and design goals of TAO [4], 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; and Section 4 presents concluding remarks.

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

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 standard OMG CORBA reference model [12], with the following enhancements designed to overcome the shortcomings of conventional ORBs [9] for high-performance and real-time applications:

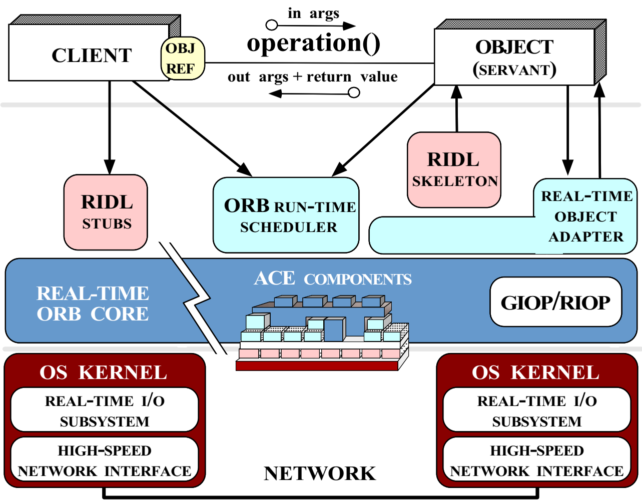

Figure 1: Components in the TAO Real-time ORB Endsystem

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

Real-time Object Adapter: An Object Adapter associates servants with the ORB and demultiplexes incoming requests to servants. TAO's real-time Object Adapter [10] uses perfect hashing [15] and active demultiplexing [6] 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 [1] 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 [4] and dynamic [5] 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 [9] uses a multi-threaded, preemptive, priority-based connection and concurrency architecture [13] to provide an efficient and predictable CORBA IIOP protocol engine.

Real-time I/O subsystem: TAO's real-time I/O subsystem [8] 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 [16]. 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 [17], which implements core concurrency and distribution patterns [18] 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

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

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 [6, 11, 19, 20, 9] 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

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 [21].

- **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 [22].

- **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 [23].

- **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 [24]. 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 [25]. 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 OS platforms except 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 scheduling is round-robin instead of FIFO since Windows NT does not support FIFO scheduling.[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 [4], 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 [26] 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.

[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.

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:

- **Context switch overhead:** These tests measure (1) general OS context switch overhead and (2) context switching overhead incurred when processing ORB requests. High context switch overhead can significantly degrade application responsiveness and determinism. These tests and their results are presented in Section 3.1.2.

- **Priority inversion:** This test measures priority inversion incurred when processing operations from client threads running at different priorities. Priority inversion is undesirable if an OS services applications that possess stringent QoS requirements. This test and its results are presented in Section 3.1.3.

3.1.2 Measuring ORB/OS Context Switching Overhead

Terminology synopsis: A *context switch* involves the suspension of one thread and immediate resumption of another thread. The time between suspension and resumption is the *context switching* overhead. Context switching overhead indicates the efficiency of the OS thread dispatcher. From the point of view of applications and ORB middleware, context switch time overhead should be minimized because it directly reduces the effective use of CPU resources.

There are two types of context switching, *voluntary* and *involuntary*, which are defined as follows:

- **Voluntary context switch:** This occurs when a thread voluntarily yields the processor before its time slice completes. Voluntary context switching commonly occurs when a thread blocks awaiting a resource to become available.

- **Involuntary context switch:** This occurs when a higher priority thread becomes runnable or because the current thread's time quantum has expired.

Overview of context switching overhead metrics: We measured OS context switching overhead using three metrics. The first context switching metric is the *Suspend-Resume* test. It is based on the Task Context Switching measurement described in [27]. In turn, this test is based on Superconducting Super Collider (SSC) Laboratory Ping Suspend/Resume Task and Suspend/Resume Task benchmarks. It measures two different times:

1. The time to resume a blocked high-priority thread, which does nothing other than block again immediately when it is resumed. A low-priority thread resumes the high-priority thread, so the elapsed time includes two context switches, one thread suspend, and one thread resume.
2. The time to suspend and resume a low-priority thread that does nothing. There is no context switching. This time is subtracted from the one described above, and the result is divided by two to yield the context switch time.

POSIX pthreads [28] do not support a suspend/resume thread interface. Therefore, the Suspend-Resume test is not applicable to OS platforms, such as LynxOS and Linux, that only support POSIX threads.

The second context switching metric is the *Yield* test. It runs two threads at the same priority. Each thread iteratively calls its system function to immediately yield the CPU.

The third context switching metric is the *Synchronized Suspend-Resume* test. This test contains two threads, one higher priority than the other. The test measures two different times:

1. The high-priority thread blocks on a mutex held by the low-priority thread. Just prior to releasing the mutex, the low-priority thread reads the high-resolution clock (tick counter).[3] Immediately after acquiring the mutex, the high-priority thread also reads the high-resolution clock. The time between the two clock reads includes a mutex release, context switch, and mutex acquire.

   The lower priority thread uses a semaphore to suspend each iteration of the high-priority thread. This prevents the high-priority thread from simply acquiring and releasing the mutex *ad infinitum*. The timed portions of the test do not include semaphore operation overhead.
2. The time to acquire and release a mutex in a single thread, without context switching, is measured. This time is subtracted from the one described above to yield the context switch time.

We used multiple context switch metrics because not all OS platforms support each approach. Moreover, some operating systems show anomalous results with certain metrics. For instance, Linux does not perform well on the Yield test relative to Windows NT, because its yield function always causes a schedule recalculation of all runnable processes and threads [29]. A simple change to the Linux yield function implementation can reduce the number of scheduling recalculations during the Yield tests, which dramatically reduces the measured context switch time. This example demonstrates the sensitivity of context switch time measurements to OS scheduling implementations.

[3] Solaris provides a high-resolution timer interface. On other OS platforms, the Pentium RDTSC instruction was used directly to read the tick counter.

Below, we describe the results from tests that measure (1) the OS context switching overhead and (2) the number of context switches incurred per CORBA request. To support real-time ORB middleware, an OS should minimize this overhead.

Results of OS context switch overhead metrics: Table 1 shows the context switch times measured on each of the platforms. These results emphasize the difficulty of measuring context switch time. Windows NT performs consistently well, while Solaris consistently performs the worst of the tested OS's, with the exception of the VxWorks Yield test.

Operating System	**Context Switch Time, μsec** mean (standard deviation)		
	Suspend-Resume Test	Yield Test	Synch Test
VxWorks	0.946 (0.041)	N/A	1.62 (0.023)
LynxOS	N/A	5.42 (0.008)	5.96 (0.042)
Windows NT	1.41 (0.036)	1.78 (0.021)	2.79 (0.110)
Solaris	21.3 (0.569)	11.2 (0.900)	131.2 (0.613)
Linux	N/A	2.60 (0.023)	9.72 (0.187)

Table 1: Context Switch Time Measurements

The VxWorks context switch times as measured by the Suspend-Resume and Synchronized Suspend-Resume tests are very low, around 1 μsec. However, they are not as consistent as on some of the other platforms, with a standard deviation of up to about 4% of the mean. The Yield test was not run on VxWorks because it does not support an immediate thread yield without delaying the calling task for a non-zero time interval. Measuring the yield time would include that interval (of 1/60 second on Pentium target), therefore adding to the inaccuracy of the context switch time calculation.

The LynxOS context switch times are relatively high, between 5 and 6 μsec. Surprisingly, the times are no better than we measured on a 200 MHz Pentium Pro. This may be an anomaly in either then OS or our tests, possibly with respect to caching behavior. The jitter is very low on LynxOS, less than 1% of the mean.

The context switch times measured on Windows NT are consistently low, but with jitter of up to 3.9%. Conversely, Solaris has very high context switch times, the best being 11.2 μsec for the Yield test, and very high jitter of 8%. The Linux Yield test context switch time of 2.60 μsec is also low, though its Synchronized Suspend-Resume time of 9.72 μsec is high. The jitter on Linux is less than 2%.

The Suspend-Resume test, Yield test, and Synchronized Suspend-Resume test results are not directly comparable. All measure voluntary context switches. However, the scheduling ramifications of thread suspension and yield may be different. This is apparent from the results on Solaris and Linux, especially, which show different times for the approaches.

The results above demonstrate that it is difficult to measure context switching overhead reliably. Therefore, the multiple measures of context switch time are useful.

Impact of context switching overhead on two-way CORBA operations: OS context switching overhead significantly impacts the performance and predictability of real-time ORB endsystems. In addition, context switching complicates real-time scheduling analysis [30]. Thus, high levels of OS context switching overhead are undesirable for applications with stringent performance requirements.

To study the effect of context switching overhead on CORBA operations, we consider two-way operations, *i.e.*, round-trip request-response from client to server and back. The client and server execute in different threads in the same process (on systems that have process boundaries). For this canonical case, we expect two context switches. The first occurs when the ORB passes the operation to the servant, executed in the context of the server thread. The second context switch occurs when the client thread executes to handle the response.

We measured the number of context switches for this case on several of the OS platforms.[4] On Solaris, we calculated the number of context switches using the `getrusage` library function. It reports voluntary and involuntary context switches incurred by the current process; we summed the two values.

On Windows NT, we used the *Microsoft Spy++* utility that comes with the Microsoft Visual C++ compiler. This utility displays the number of context switches incurred by each thread. To read the number of context switches, we forced the threads to block on exit waiting for input from the console.

To determine the number of context switches performed by the OS, we made 4,000 two-way CORBA requests in n client threads and computed the number of context switches incurred by the OS. There were one high-priority client thread and n low-priority client threads, where n ranges from 1 to 50. The low-priority threads all run at different priorities ranging from $P_1 \ldots P_n$. On both Solaris and Windows NT, we measured an average of two context switches per two-way request, as expected.

Result synopsis: In general, context switching overhead is an important measure of the efficiency of an OS thread dispatcher. Our measurements confirm that there are two context switches per two-way CORBA operation. In addition, we measured the actual cost of a context switch. Typically, it is between 1 and 10 μsec on all of the OS's that we surveyed. Therefore, its contribution to the overall two-way operation latency is very small.

The standard deviations of the context switch measurements for LynxOS and Windows NT, and to a lesser extent Vx-

[4] LynxOS does not provide the internal instrumentation to measure context switches, to minimize context switching overhead.

Works and Linux, are much lower than for Solaris relative to their means, indicating that their dispatchers are more efficient and predictable. If the efficiency of the Solaris thread dispatcher can be improved, ORBs will perform more predictably, thereby helping to meet application QoS requirements more effectively.

3.1.3 Measuring ORB/OS Priority Inversion

Terminology synopsis: Priority inversion occurs when a high-priority thread must block waiting for a low-priority thread to release a resource required by the higher priority thread. Two types of priority inversions exist, *thread-based* and *packet-based* [8]:

• **Thread-based priority inversion:** This inversion occurs when higher priority threads must block waiting for lower priority threads to release a resource required by the higher priority threads. Unbounded thread-based priority inversion is highly undesirable for most real-time systems since it yields non-deterministic behavior. In turn, this can result in missed deadlines for real-time application and ORB endsystem tasks.

• **Packet-based priority inversion:** Even if thread-based priority inversion is bounded or eliminated, another potential priority inversion problem exists. This problem stems from the fact that many protocol implementations queue and process packets in FIFO order. FIFO queueing is prone to *packet-based priority inversions*. These inversions occur when higher priority threads must block until the packet they need to process is at the front of the queue.

Overview of the priority inversion metric: Priority inversion can be detected by observing the latencies of client and server threads that run at different priorities. Higher latency in a higher priority client indicates priority inversion. The degree of the priority inversion is the difference in latency from the average lower priority latency.

In this benchmark we measured packet-based and thread-based priority inversion. The configuration used for this benchmark is shown in Figure 2. This benchmark is based on a *priority-based* concurrency architecture [31], which is often used by real-time applications with deterministic QoS requirements. For instance, avionics mission computing systems [7] commonly execute fixed priority threads corresponding to the *rates*, *e.g.*, 20 Hz, 10 Hz, 5 Hz, and 1 Hz, at which operations are called by clients.

Each client thread generates CORBA requests at a constant rate. This test exposes the two types of priority inversion by using a range of priorities in the client and server threads. For instance, the OS I/O subsystem may not consider the priority of each thread when queueing the network packets, *e.g.*, it may just queue them in FIFO order. As lower priority threads send

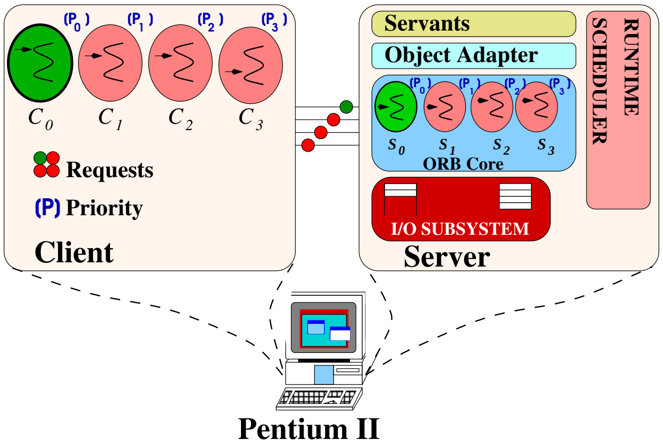

Figure 2: ORB Endsystem Priority Inversion Test Configuration

CORBA requests and the lower layers in the network queue those requests, some high-priority requests can be delayed by lower priority requests. This behavior can cause higher latency for higher priority requests, *i.e.*, packet-based priority inversion.[5]

The client and server processes for the priority inversion benchmark are configured as follows:

• **Server configuration:** As shown in Figure 2, our testbed server consists of four servants $S_0 \ldots S_3$, each running in a thread with a corresponding real-time priority $P_0 \ldots P_3$. Each thread processes requests that are sent to its servant by the corresponding client threads $C_0 \ldots C_3$ in another process on the same machine. Each pair of client/server threads have matching priorities, *i.e.*, a client thread C_i communicates with a servant thread S_i with the same thread priority p_i.

• **Client configuration:** Figure 2 shows the client-side of the priority inversion benchmarking test. The highest priority client (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 rest of the clients $C_1 \ldots C_3$ have lower priority OS threads $P_1 \ldots P_3$ and invoke operations at 10, 5, and 1 Hz, *i.e.*, they invoke 10, 5, and 1 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.

When the test program creates the client threads, these threads block on a barrier lock so that no client begins work

[5]TAO's ORB Core is designed to alleviate *thread-based* priority inversion by using a priority-based concurrency architecture [31] and non-multiplexed connection architecture [9] that share a minimal amount of resources among threads. Consequently, TAO incurs minimal thread-based priority inversion.

until the others are created and ready to run. When all threads inform the main thread they are ready to begin, the main thread unblocks all client threads. These threads execute in an order determined by the real-time thread dispatcher. Each client invokes 1,000 CORBA two-way requests at its prescribed rate. All clients, except for the lowest priority client, *i.e.*, C_3, make CORBA requests as long as the lowest priority client is issuing requests. Thus, there will always be higher priority traffic for the duration of the test.

Priority inversion occurs when a higher priority client incurs higher latency than lower priority threads. In an ideal ORB endsystem, we should see no priority inversion, *i.e.*, the higher the priority, the lower the latency. In the figure, this would look like a "staircase," climbing slightly higher from left to right.

Results of priority inversion metrics: The average priority inversion incurred by various clients is shown in Figure 3. The

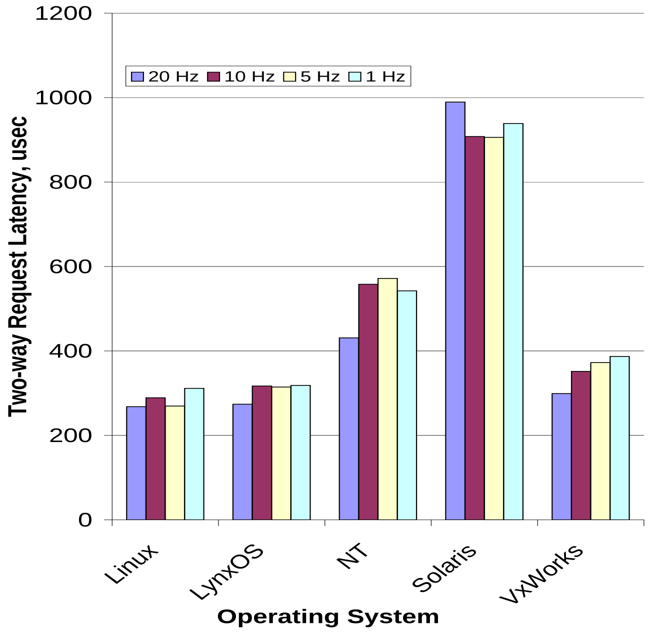

Figure 3: TAO's Priority Inversion for OS Platforms

jitter results for this test are shown in Figure 4. An important characteristic of real-time operating systems and ORBs is *predictability*. In particular, for real-time applications with deterministic QoS requirements, low jitter is essential to bound computation time and to ensure that deadlines are met. Therefore, operating systems that exhibit high jitter in Figure 4 may not be suitable for certain classes of real-time applications, even though their average priority inversion is low.

- **Linux results:** The TAO latency on Linux is comparable to that of the real-time operating systems. However, it

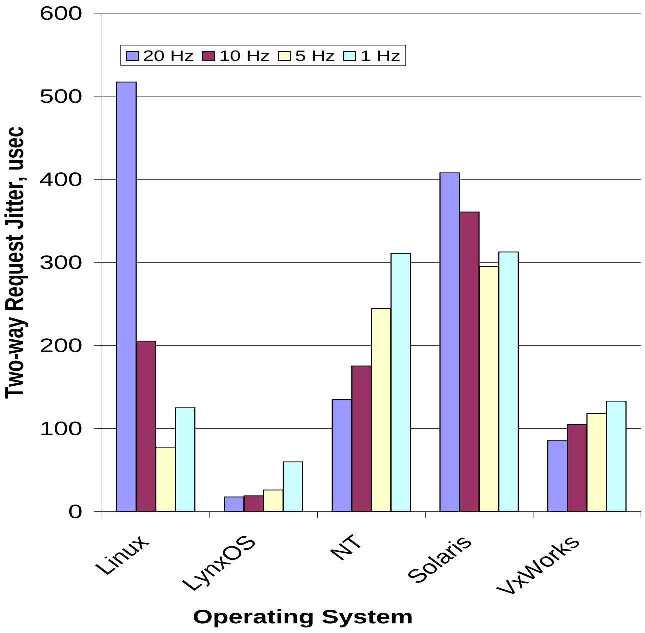

Figure 4: TAO's Jitter for OS Platforms

does incur priority inversion, *e.g.*, the 10 Hz client latency of 289 μsec is higher than the 269 μsec 5 Hz client latency. Furthermore, jitter is very high on Linux, from 28.8% to 193% of the mean latency.

- **LynxOS results:** LynxOS does not display measurable priority inversion in our tests, as shown in Figure 3. In addition, LynxOS exhibited the lowest jitter of any of the tested systems, *i.e.*, 6.42% of the mean latency for the 20 Hz client, up to 18.8% for the 1 Hz client.

- **Windows NT results:** TAO displays priority version on Windows NT. As shown in Figure 3, the latency of the 5 and 10 Hz clients is higher than that of the 1 Hz client. In addition, jitter is high on Windows NT, as shown in Figure 4, ranging from 31.3 to 57.3% of the mean latency.

- **Solaris results:** Solaris exhibits priority inversion, as shown in Figure 3. Figure 4 shows the jitter on Solaris is high, 32.6% to 41.2%. The high-priority 20 Hz client has higher latency than the three lower-priority clients. This relative inversion does not occur for the 10 Hz, 5 Hz, and 1 Hz client threads.

- **VxWorks results:** No priority inversion is observed in the VxWorks benchmark, as shown in Figure 3. Furthermore, the jitter of the measurements is low, 28.7% to 34.4%, from Figure 4.

Result synopsis: To bound application execution time, it is important for real-time ORB endsystems to minimize priority inversion. However, thread-based priority inversion and packet-based priority inversion are hard to eliminate completely since lower layers of the protocol stack are often unaware of the priorities of the packet's sender/receiver thread. For instance, Solaris and Windows NT incur a fair amount of priority inversion. In contrast, LynxOS and VxWorks behave more deterministically, which makes them better suited to provide QoS required by applications.

4 Concluding Remarks

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

This paper shows the initial results of our investigation into the characteristics that determine the suitability of the OS component in an ORB endsystem to support real-time applications. OS context switch overhead contributes little to overall two-way CORBA operation latency. Priority inversion is successfully avoided by real-time operating systems, but not by general-purpose operating systems. Our preliminary results indicate that a real-time ORB like TAO, run on a real-time OS like LynxOS or VxWorks, can provide a very predictable and efficient ORB endsystem platform for real-time applications.

We are also exploring other OS characteristics that affect ORB endsystem performance [32]. We are expanding our latency and jitter measurement techniques to provide a better indication of the end-to-end performance that applications can expect. In addition, we are developing techniques to measure and reduce ORB endsystem overhead, which is important given the constrained CPU resources of most real-time systems.

References

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

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

[3] 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.

[4] 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.

[5] C. D. Gill, D. L. Levine, and D. C. Schmidt, "Evaluating Strategies for Real-Time CORBA Dynamic Scheduling," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 1999, to appear.

[6] 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.

[7] 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.

[8] D. C. Schmidt, R. Bector, D. L. 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.

[9] 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.

[10] 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 *Proceedings of the 5^{th} Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[11] 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.

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

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

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

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

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

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

[27] D. Cathey, "RTOS Benchmarking – All Things Considered...Important Factors in Choosing a Real-Time Development System," *Real-Time Magazine*, Second Quarter 1993. http://www.wrs.com/corporate/html/artreqfm.html.

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

[29] R. Appleton, "Understanding a Context Switching Benchmark," *Linux Journal*, pp. 70–71, Jan. 1999.

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

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

[32] D. L. Levine, S. Flores-Gaitan, and D. C. Schmidt, "An Empirical Evaluation of OS Support for Real-time CORBA Object Request Brokers," in *Submitted to the* 5^{th} *IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), IEEE, June 1999.