Operating System Performance in Support of Real-time Middleware

Douglas C. Schmidt, Mayur Deshpande, and Carlos O'Ryan
Department of Electrical and Computer Engineering
University of California, Irvine, 92612
{schmidt,deshpanm,coryan}@uci.edu [*]

A subset of this paper appeared in the 7th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'02), San Diego, California, January 7–9, 2002.

Abstract

Commercial-off-the-shelf (COTS) hardware and software is being evaluated and/or used in an increasing range of mission-critical distributed real-time and embedded (DRE) systems. Due to substantial R&D investment over the past decade, COTS middleware has recently matured to the point where it is no longer the dominant factor in the overhead, non-determinism, and priority inversion incurred by DRE systems. As a result, the focus has shifted to the COTS operating systems and networks, which are once again responsible for the majority of end-to-end latency and jitter.

This paper compares and evaluates the suitability of popular COTS operating systems for real-time COTS middleware, such as Real-time CORBA. We examine real-time operating systems (VxWorks and QNX), general-purpose operating systems with real-time thread scheduling classes (Windows NT, Windows 2K, and Linux), and a hybrid real-time/general-purpose operating system (Linux/RT). While holding the hardware and ORB constant, we vary these operating systems systematically to measure platform-specific variations in context switch overhead, throughput of the ORB in terms of two-way operations per-second and memory footprint of the ORB libraries. We also measure how the latency and jitter of high-priority operations are affected as the number of low-priority operations increase.

Our results indicate that real-time operating systems remain the platforms of choice to deliver predictable, efficient, and scalable performance for DRE middleware and applications. However, the emergence of hybrid general-purpose/real-time operating systems, such as Linux/RT, are a promising direction for future DRE systems. Although Linux/RT is not yet as deterministic as traditional real-time operating systems, such as QNX and VxWorks, it does provide more predictable and scalable behavior compared to mainstream operating systems, such as Windows NT/2K. Since traditional real-time operating systems tend to be expensive and tedious to configure/program, the maturation of Linux/RT will be a welcome advance for DRE system developers.

Keywords: Real-time CORBA Object Request Brokers, Real-time Operating System Middleware Support.

1 Introduction

Emerging trends. Two fundamental trends are having a profound influence on the way in which new *distributed real-time and embedded* (DRE) systems are being conceived and constructed:

- *Commoditization of Information Technology (IT)* — IT of all forms is becoming highly commoditized *i.e.*, COTS hardware and software artifacts are getting faster, cheaper, and better at a relatively predictable rate. During the past decade, many application domains have benefited from the commoditization of hardware, such as CPUs and storage devices, and networking elements, such as IP routers. More recently, the maturation of programming languages, such as Java and C++, operating environments, such as POSIX and Java Virtual Machines, and enabling middleware, such as CORBA and Enterprise Java Beans, are helping to commoditize many software components and architectural layers, as well.
- *Network-centric paradigm shift* — There is a growing acceptance of network-centric systems, where applications with a range of quality of service (QoS) needs are constructed by integrating separate components connected by various forms of communication services. The nature of this interconnection can range from small and tightly coupled DRE systems, such as avionics mission computing, to the large and loosely coupled DRE systems, such as global telecommunications.

The interplay of these two trends has led to new architectural concepts and services that are embodied in layers of COTS middleware [1]. These middleware layers are interposed between applications and commonly available

[*]This work was supported in part by Siemens and SAIC.

COTS hardware and software infrastructure to make it feasible, easier, and more cost effective to develop and evolve DRE systems. Middleware is the result of recognizing the need for more advanced and capable support–beyond simple connectivity–needed to construct effective DRE systems.

Although the quality of COTS software has generally lagged behind hardware, recent improvements in frameworks [2], patterns [3, 4], and development processes [5, 6] have encapsulated the knowledge that enables COTS software to be developed, combined, and used in an increasing number of mission-critical DRE applications. Common examples include e-commerce web sites, consumer electronics, avionics mission computing, hot rolling mills, backbone routers, and high-speed network switches. Over the past several years, there has been substantial improvement in the QoS of COTS DRE middleware, based largely on the maturation of implementations of industry standards, such as Real-time CORBA [7].

The growing importance of operating system infrastructure. First generation COTS middleware, such as initial implementations of CORBA, lacked appropriate optimizations and capabilities to support DRE systems with stringent QoS requirements [8]. Due to substantial R&D progress over the past decade [9, 10], COTS middleware, such as Real-time CORBA [7], has recently matured to the point where it is no longer the dominant factor in the overhead, non-determinism, and priority inversion incurred by DRE systems [11]. As a result, the focus has shifted to the COTS operating systems and networks, which are once again responsible for the majority of end-to-end latency and jitter.

This paper compares and evaluates the suitability of popular COTS operating systems for real-time COTS middleware, in particular Real-time CORBA. We examine three types of operating systems:

- *Real-time operating systems*, *i.e.*, VxWorks and QNX
- *General-purpose operating systems* with real-time thread scheduling classes, *i.e.*, Windows NT, Windows 2K, and Linux, and
- A *hybrid real-time/general-purpose operating system*, *i.e.* Linux/RT.

Our findings extend earlier results in [12, 13] and illustrate that general-purpose operating systems like Windows NT, Windows 2K, and Linux are not yet suited to meet the demands of applications with stringent QoS requirements. We also find that real-time operating systems like QNX and VxWorks do enable predictable and efficient ORB performance, thereby making them suitable as OS platforms for real-time CORBA applications.

This paper extends our previous research by including measurements for Linux/RT. We find that Linux/RT provides better control over QoS capabilities than Linux, but its jitter makes it unsuitable for applications with hard real-time deadlines. In general, our results underscore the need for a measure-driven methodology to pinpoint sources of overhead and priority inversion in ORB middleware and operating systems for DRE systems.

Paper organization. The remainder of this paper is organized as follows: Section 2 describes the ORB/OS testbed that we used to systematically benchmark the performance of TAO on popular real-time and general-purpose operating systems; Section 3 analyzes empirical results from benchmarks that measure the efficiency and predictability of TAO on the different OS platforms; and Section 4 presents concluding remarks.

2 Overview of ORB/OS Testbed

Evaluating the performance of complex DRE systems is hard and becomes even harder if DRE system is built using a combination of real-time middleware and operating systems [11]. This section describes the ORB/OS testbed that we used to systematically benchmark the performance of TAO over different operating systems, while keeping the hardware platform constant. This testbed helps to isolate the overhead due to the OS.

2.1 Hardware Overview

All of the tests in this section were run on a single-cpu Dell 866MHz Intel Pentium III system configured with 512 Mbytes of RAM and a 256Kb cache. We focus our experiments on a single CPU hardware configuration to:

- Factor out differences in network interface driver support and
- Isolate the effects of OS design and implementation on ORB middleware and application performance.

2.2 Operating Systems Overview

We ran the ORB/OS benchmarks described in this paper on two well-established real-time operating systems (VxWorks 5.4 and QNX-RTP 6.0), three general-purpose operating systems with real-time scheduling classes (Windows NT 4.0 Workstation with SP5, Windows 2K Professional, and Debian GNU/Linux kernel version 2.2.14), and one hybrid general-purpose/real-time OS (Timesys Linux/RT version 2.2.14). On all platforms, we used the TCP/IP stack supplied with the OS. Each OS is described briefly below:

• **VxWorks and QNX** Both VxWorks and QNX are multi-threaded real-time OSes but QNX has a micro-kernel architecture that uses message-passing as a fundamental means for inter-process communication (IPC). Both VxWorks and QNX support preemptive priority-based first-in first-out (FIFO) scheduling of threads, in addition to semaphores that implement a priority inheritance protocol [14].

• **Linux and Linux/RT** 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 Linux thread implementation internally uses processes created by a variant of the `fork()` system function called `clone()`. This design simplifies the Linux kernel, though it limits scalability because kernel process resources are used for each application thread. Linux/RT adds a resource kernel (RK) [15] to the core Linux kernel. The RK enhances the real-time capabilities of Linux by providing fixed-priority scheduling with priority-inheritance and high-resolution timers. Linux/RT is also binary compatible with Linux, *i.e.*, it is possible to run Linux and Linux/RT applications on the same hardware without recompiling them. Booting with the Linux/RT kernel starts the Linux/RT OS. The rest of the OS, *e.g.*, file-systems, C-libraries, compiler, and command-line tools, behaves like regular Linux.

• **Windows NT and Windows 2K** are general-purpose, preemptive, multi-threading OS designed to provide fast interactive response. They use round-robin scheduling algorithm that attempts to share the CPU fairly among all ready threads of the same priority. Windows NT/2K and support high-priority threads via their REALTIME_PRIORITY_CLASS. Threads in this class are scheduled before most other threads, which are usually in the NORMAL_PRIORITY_CLASS. Windows NT/2K are not designed as a deterministic real-time OS, however. In particular, their 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.

2.3 Compiler Overview

We use the GNU g++ compiler on all platforms except Windows NT, where we use Microsoft Visual C++ 6.0. On VxWorks we used the cygnus gcc cross-compiler supplied by WindRiver to compile from a Windows-NT host. To enhance performance, all libraries and executables were compiled statically and without debugging symbols.

The compiler settings for each platform are noted below.

• **VxWorks.** We used a cross-compiler to compile from a Windows NT host. The compiler was the gcc compiler from Cygnus, version-2.7.2-960126 supplied by WindRiver. The compiler options included: `fno-implicit-templates`, `DCPU=I80486`, `m486`, `DACE_VXWORKS=0x540`, `D_REENTRANT`, `O`, `fno-rtti`, `DACE_LACKS_RTTI`, `check-new`. The VxWorks gcc compiler has bugs with native C++ exception handling and the `O3` optimization level, so we could not use these options.

• **QNX.** We used gcc-2.95.2 for this platform. The options included: `fno-exceptions`, `fcheck-new`, `Wpointer-arith`, `O3`, `fno-implicit-templates`, `DACE_NDEBUG`, and `D__ACE_INLINE__`.

• **Linux and Linux/RT** We used gcc-2.95.2 for these platforms as well. The options included: `Wpointer-arith`, `O3`, `fno-implicit-templates`, `D_POSIX_THREADS`, `fno-exceptions`, `fcheck-new`, `D_POSIX_THREADS`, `D_POSIX_THREAD_SAFE_FUNCTIONS`, `D_REENTRANT` and `DACE_NDEBUG`.

• **Windows NT/2K.** We use the VC++ 6.0 `Static Release` build workspaces provided with the ACE+TAO distributions. C++ exception handling is enabled because Windows uses structured exceptions to report certain OS-level errors.

2.4 Real-time CORBA Overview

The vehicle for our experiments on middleware and OS support for DRE applications is version 1.2 of *The ACE ORB* (TAO) [16]. TAO is an open-source[1] Real-time CORBA (RT-CORBA)-compliant ORB developed at the University of California, Irvine and Washington University, St. Louis. It is designed to support applications with stringent end-to-end QoS requirements. As shown in Figure 1 TAO 1.2 implements all

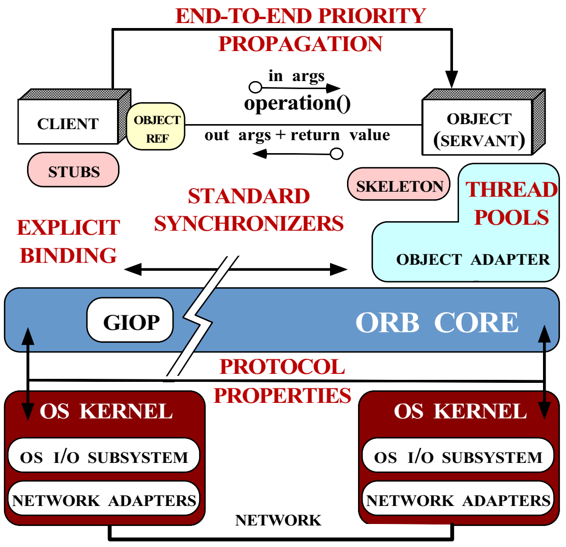

Figure 1: ORB Endsystem Features for Real-Time CORBA

[1]The source code and performance tests for TAO can be downloaded from `www.cs.wustl.edu/~schmidt/TAO.html`.

the RT-CORBA standard interfaces and QoS policies that allow applications to configure and control the following ORB endsystem resources:

- *Processor resources* via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service
- *Communication resources* via protocol properties and explicit bindings and
- *Memory resources* via buffering requests in queues and bounding the size of thread pools.

3 Empirical Benchmarking Results

We have developed the following ORB/OS benchmarking metrics to evaluate the performance and predictability of VxWorks, QNX, Windows NT, Windows 2K, Linux/RT, and Linux all running TAO 1.2:

• **Context switch overhead.** These tests measure general OS context switch overhead. High context switch overhead can significantly degrade application responsiveness and determinism. These tests and their results are presented in Section 3.1.

• **ORB/OS operation throughput.** This test indicates the maximum operation throughput that applications can achieve on various OS platforms. It measures end-to-end two-way response when a client sends a request immediately after receiving the response to the previous request. This test and its results are presented in Section 3.2.

• **ORB/OS latency and jitter.** This test measures how the latency and jitter of high-priority operations are affected as the number of low-priority operations increases. Ideally, high-priority operations should not be affected at all, while the latency of low-priority operations should increase gradually as their numbers increase. For plus, for real-time systems, it is also imperative that the jitter of high-priority tasks remain as constant as possible as the number of lower priority tasks is varied. This test and its results are presented in Section 3.3.

• **ORB memory footprint.** This test measures the size of the static TAO library on various operating systems. As mentioned in Section 2.3, TAO was compiled and linked statically. The results are presented in Section 3.4.

The remainder of this section describes these benchmarks and their results in more depth.

3.1 Measuring ORB/OS Context Switch 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 switch overhead. Context switch overhead is a measure of the efficiency of the OS thread dispatcher. From the point of view of applications and ORB middleware, the lower the context switch time, the better the performance, since this overhead reduces the effective use of CPU resources.

There are two types of context switches—*voluntary* and *involuntary*—which are defined as follows:

• **Voluntary context switches** occur when a thread voluntarily yields the processor before its time slice completes. Voluntary context switches commonly occurs when a thread blocks awaiting a resource to become available.

• **Involuntary context switches** occur when a higher priority thread becomes runnable or because the current thread's time quantum has expired.

Overview of context switch overhead metrics. We measured OS context switch overhead using the following metrics:

1. The Suspend-Resume test. This test 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, *i.e.*, there is no context switch. This time is subtracted from the one described above, and the result is divided by two to yield the context switch time.

POSIX threads do not support a suspend/resume thread interface. The Suspend-Resume test is therefore inapplicable for the QNX, Linux, and Linux/RT platforms that only support POSIX threads.

2. The Yield test. This test runs two threads at the same priority. Each thread iteratively calls a system function that yields the CPU immediately.

3. 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). 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.

This test is applicable to all OS platforms.

We use multiple context switch metrics because no single approach is supported by all OS platforms. Moreover, some operating systems show anomalous results with certain metrics, *e.g.*, Windows NT/2K performs poorly on the Suspend-Resume and Yield tests. [2]

Results of OS context switch overhead metrics. Table 1 shows the results of the context switch overhead tests. We describe the results for the various OS platforms below.

Operating System	Suspend-Resume Test	Yield Test	Synch Test
VxWorks	0.586 (0.025)	0.649 (0.013)	0.821 (0.019)
QNX	N/A	0.470 (0.007)	0.861 (0.003)
Linux/RT	N/A	0.645 (0.007)	2.88 (0.015)
Linux	N/A	0.548 (0.006)	2.559 (0.011)
Windows NT	1.147 (0.005)	1.056 (0.006)	1.914 (0.010)
Windows 2K	1.148 (0.004)	1.075 (0.007)	2.810 (0.016)

Table 1: **Context Switch Latency in μseconds (Jitter Shown in Parentheses)**

• **VxWorks and QNX results.** These results show that classic real-time operating systems like QNX and VxWorks are top performers. QNX was better than VxWorks on the Yield test and only slightly worse off on the Suspend-Resume test. The jitter for QNX, though, was lower than that of VxWorks, making it more predictable. QNX performed the best of all the operating systems on the Yield test.

• **Linux and Linux/RT results.** Linux/RT was ∼3 times slower than QNX and VxWorks on the Synchronized Suspend-Resume test. It was also slower than QNX and Linux on the Yield test. Linux/RT showed a consistent trend of having a higher context switch time than Linux, which may be due to the addition of more preemption points in the kernel. Surprisingly, its context switch jitter was also higher slightly than Linux.

• **Windows NT and Windows 2K results.** Windows-NT performed better than Linux or Linux/RT on the Synchronized-Suspend-Resume test but fared worse than the Linux-es on the Yield test taking almost twice the time. But in keeping with the general trend, both NT and 2K were worse than the real-time OSes (QNX and VxWorks) in terms of raw-performance on comparable tests.

[2]The tests described in this section are available in the ACE release at `$ACE_ROOT/performance-tests/Misc`.

Our subsequent results demonstrate that although context switch overhead is a useful metric to compare OS performance, it is not a good predictor for actual end-to-end application performance. It should therefore only be considered along with other metrics in a performance evaluation. Moreover, the results above demonstrate that the context switch overhead measurements depend largely on the particular benchmark used. Practitioners and researchers should therefore be careful to use (multiple) standardized benchmarks in their OS comparisons.

3.2 Measuring ORB/OS Operation Throughput

Terminology synopsis. *Operation throughput* is the maximum rate at which CORBA 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. The one-way operation measurement eliminates the server reply overhead. This test indicates the overhead imposed by the ORB and OS for 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.[3] The server performs the operation, which cubes each parameter in the request. For two-way operations, the client thread waits for the response and checks its correctness. Inter-process communication is performed via the network loop back interface because 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 representative of DRE application workload. Many real-time and embedded applications do rely, however, on a large volume of small messages that each requires a small amount of processing. The `IDL_Cubit` benchmark is therefore 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`, `short`, `long`, and `sequence` types. The one-way operation measurement eliminates the server reply overhead. The `void` keyword instructs the server to not perform any processing other than that necessary to prepare and send a no-op response, *i.e.*, no input parameters are passed to cube. The `sequence` data types exercise TAO's marshaling/demarshaling performance [17].

[3]The `IDL_Cubit` test is available in the TAO release at `$TAO_ROOT/performance-tests/Cubit/TAO/IDL_Cubit`.

Results of the operation throughput measurements. The throughput measurements are shown in Figure 2. The following discussion describes the results for the various OS platforms.

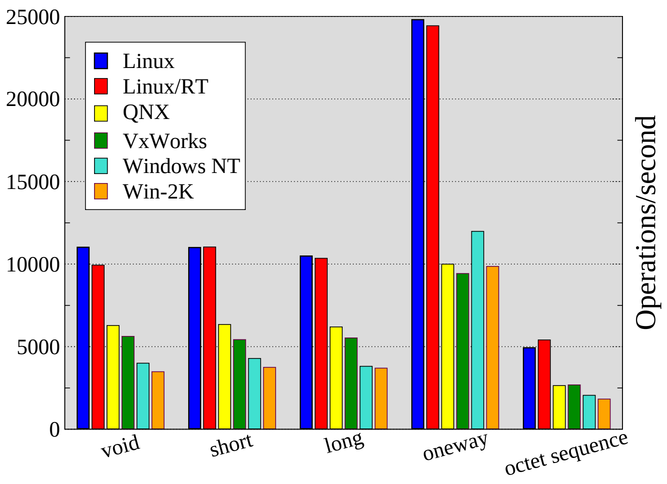

Figure 2: **Operation Throughtput Results**

- **Linux and Linux/RT results.** Linux and Linux/RT exhibit the highest operation throughput for all the data types tested, with around 10,000 operations/sec for the simple data types. Moreover, the one-way throughput on Linux was significantly higher—nearly double that on the other OS platforms.

- **QNX and VxWorks results.** QNX and VxWorks offer consistently good performance for simple types, such as `void`, `short`, and `long`, with QNX's throughput being around 6,000 calls/sec and VxWorks a little lower at 5,500 calls/sec.

- **Windows NT and Windows 2K results.** Windows NT and 2K performed the worst of all the operating systems on the two-way tests. On the one-way test, Windows NT was better than QNX and VxWorks but worse than Linux and Linux/RT.

Result synopsis. Operation throughput provides a measure of the overhead imposed by the ORB/OS. Our `IDL_Cubit` test measures throughput for a variety of operation types and data types. Our measurements show that end-to end performance depends dramatically on type of data exchanged and the type of OS.

The raw performance of an OS or middleware platform is not the best metric, however, when evaluating the suitability of an infrastructure for DRE systems. A more important metric is the predictability of the system, *i.e.*, how the system behaves under different load conditions. In particular, the effect of low-priority operations on the performance of high-priority operations is often a more essential property of real-time ORB/OS combinations.

3.3 Measuring ORB/OS Latency and Jitter

Terminology synopsis. ORB end-to-end *latency* is defined as the average amount of delay observed by a client thread from the time it sends the request to the time it completely receives the response from a server thread. Likewise, *jitter* is the variance of the latency for a series of requests. High latency impairs the ability to meet deadlines, whereas high jitter makes it harder to devise feasible real-time schedules [14].

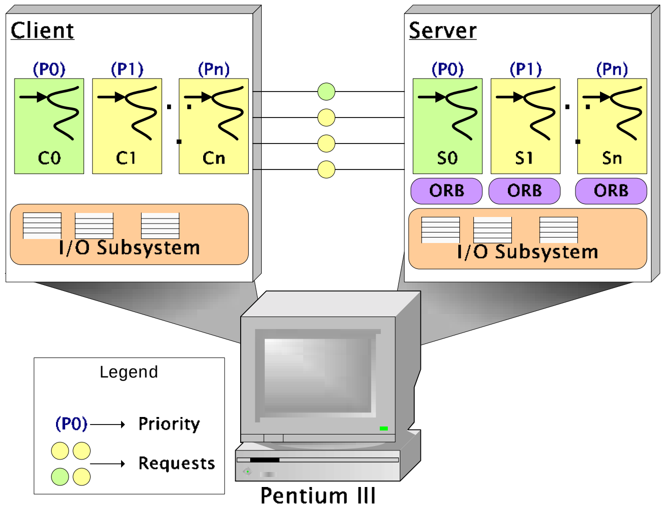

Figure 3: **ORB Endsystem Latency and Jitter Test Configuration**

Overview of latency and jitter metrics. Our latency/jitter test, called `MT_Cubit`, uses a multi-threaded client that issues IDL operations at several rates.[4] We computed the latency and jitter incurred by various clients and servers using the configurations shown in Figure 3 and described below.

- **Server configuration.** As shown in Figure 3, our `MT_Cubit` 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 on the same machine. 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 3 shows how the `MT_Cubit` 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, $C_1 \ldots C_n$,

[4]The `MT_Cubit` test is available in the TAO release at `$TAO_ROOT/performance-tests/Cubit/TAO/MT_Cubit`.

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 OS thread dispatcher.

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

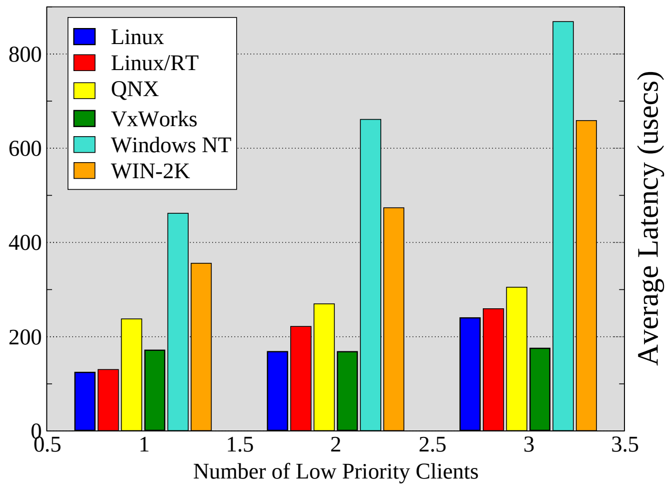

Figure 4: **TAO/OS Latency Results for High-priority Clients**

In an ideal ORB endsystem, the latency for the low-priority clients should rise gradually as the number of low-priority client threads increases. This behavior is expected because the low-priority clients compete for OS and network resources as the load increases. However, the latency of the high-priority client should remain constant or show only a minor increase with increasing number of low priority clients. 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 and jitter metrics. The average two-way response time incurred by the high-priority clients is shown in Figure 4. The jitter results are shown in Figure 5. Below, we describe the results for each OS.

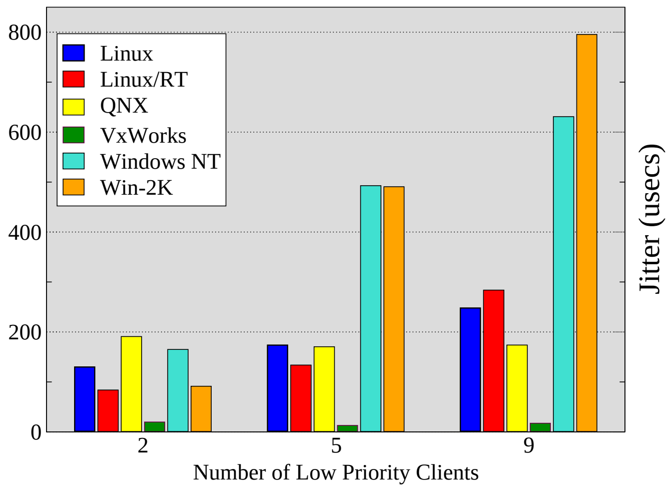

Figure 5: **TAO/OS Jitter Results for High-priority Clients**

• **QNX and VxWorks results.** In VxWorks, the latency of the high-priority client remained nearly constant as the number of low-priority clients increased. In QNX, there was a slight increase as the number of low-priority clients increased. Jitter experienced by the high-priority client remained essentially unchanged on both QNX and VxWorks, but the jitter in VxWorks was an order of magnitude smaller than on any other OS.

• **Linux/RT and Linux results.** With client threads less than 5, Linux/RT's high-priority client jitter was smaller than that of QNX but noticeably higher at 9 client threads.

Linux did surprisingly well on high-priority latency when the number of low-priority client threads remained small, being lower than that of Linux/RT, QNX or VxWorks. As the number of low-priority clients increased, however, it's latency as compared to Linux/RT, was higher for high-priority clients, though it still performed better than the Windows platforms.

It is also interesting to compare Linux/RT to other general purpose operating systems. For an increasingly large number of low priority clients, Linux/RT performed better than any of the other general-purpose operating systems. As shown in Figure 6, its jitter remains relatively low and constant (at around 200 μsecs).

• **Windows NT and Windows 2K results.** The high-priority client latency for Windows NT/2K increased linearly with the number of client threads and they were the worst of all the OS platforms tested. Windows 2K had lower latency than Windows NT, which would suggest a higher throughput but ss shown in Figure 2, however, Windows NT had higher throughput than Windows 2K.

The high-priority client jitter for Windows NT/2K rose linearly as well and even though for a smaller number of low-priority clients (2), Windows 2K has less jitter than Linux and

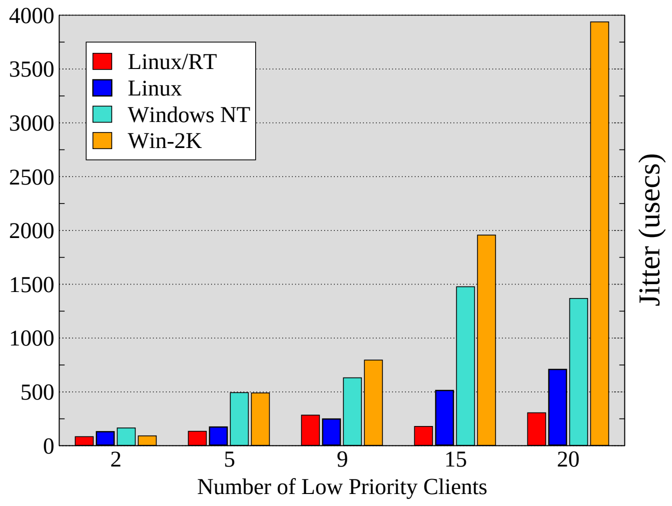

Figure 6: **TAO/OS Jitter Results for Large number of Clients**

QNX, for higher number of clients (5 and 9) it had considerably more jitter than all the other non-windows platforms.

Overall, Windows NT/2K produced poor results for all test cases. In addition, the high variability in the results indicates that Windows NT/2K is unsuitable for applications requiring predictable QoS guarantees.

Result synopsis. In general, low latency and consistent jitter are necessary for real-time operating systems to bound application execution times. The general-purpose operating systems we tested showed erratic behavior, particularly under high load. For example, Windows NT and Linux exhibit higher latency for high-priority clients. Windows NT/2K had almost three times the jitter as compared to Linux/RT (9 clients). In contrast, real-time operating systems are more predictable, showing very stable jitter even with high number of clients. For example, high-priority jitter for QNX and VxWorks was almost constant while the jitter of Linux/RT rose with an increase in load.

3.4 Measuring ORB Footprint

Overview of static memory footprint metric. The TAO real-time ORB is implemented with components from the ACE toolkit [18]. Since the size of the static memory footprint is important for many DRE systems, we measured the sizes of the ACE (libACE) and TAO (libTAO) libraries using various commands provided by the operating systems. Since the libraries were compiled as static libraries, they give a measure of the memory overhead that applications would incur if they include all the features of ACE and TAO.

Results of footprint metrics. Figure 7 shows the memory footprint measured on each of the platforms in kilobytes. These results show that Windows NT/2K has the smallest

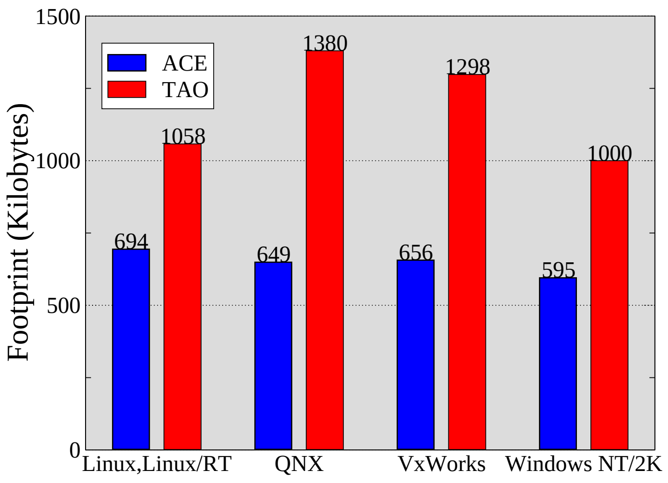

Figure 7: **Memory Footprint in Kilobytes**

static memory footprint, which indicates that the NT compiler produces highly compact code. The larger footprint on the gcc-compiled platforms may have been caused due to aggressive inlining that occurs with the higher compiler optimization levels. The QNX compiler had the largest memory footprint for TAO, whereas Linux had the largest footprint for ACE.

4 Concluding Remarks

Over the past several years, many companies have used Real-time CORBA successfully in distributed real-time and embedded (DRE) systems to (1) lower software development costs and (2) decrease their time-to-market. The flexibility, reusability, and platform-independence offered by COTS Real-time CORBA makes it attractive for use in object-oriented DRE systems. Some developers continue to doubt the applicability of CORBA for DRE systems, however, due to concerns about its overhead, non-determinism, and priority inversion.

As recently as two years ago, it was reasonable to be skeptical about the determinism of COTS CORBA implementations for DRE systems [19]. However, the results in this paper show the following:

- The main sources of overhead, non-determinism, and priority inversion can no longer be attributed to the ORB middleware, which is consistent with the findings in [11]. In our experiments, real-time operating systems showed consistency of jitter and predictability while running a representative mixture of CORBA-based DRE applications. In contrast, general-purpose operating systems exhibit various shortcomings that make them impractical for DRE applications with stringent QoS requirements.

- The emergence of hybrid real-time/general-purpose operating systems, such as Linux/RT, is a promising development since our earlier work [12, 13] measuring ORB/OS performance. Although Linux/RT is not yet as deterministic as traditional real-time operating systems, such as QNX and VxWorks, it does provide more predictable and scalable behavior compared to mainstream operating systems, such as Windows NT/2K. Since traditional real-time operating systems tend to be expensive and tedious to configure/program, the maturation of Linux/RT will be a welcome advance.
- Our experiments confirm that OS context switch overhead contributes little to end-to-end two-way CORBA operation latency or throughput, and should not be used as the only predictor of OS performance.

Our future work is exploring software architectures, optimizations, and patterns that can most effectively implement the Real-time CORBA capabilities incorporated recently into the CORBA 2.4 specification [7]. We plan to use the benchmarking techniques and ORB/OS testbed described in this paper to evaluate Real-time CORBA capabilities empirically to determine their suitability for different classes of DRE systems.

References

[1] R. E. Schantz and D. C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," in *Encyclopedia of Software Engineering* (J. Marciniak and G. Telecki, eds.), New York: Wiley & Sons, 2001.

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

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

[4] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000.

[5] K. Beck, *Extreme Programming Explained: Embrace Change*. Boston: Addison-Wesley, 2000.

[6] I. Jacobson, G. Booch, and J. Rumbaugh, *Unified Software Development Process*. Addison-Wesley Object Technology Series, Reading, Massachusetts: Addison-Wesley, 1999.

[7] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.4 ed., Oct. 2000.

[8] 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.

[9] G. Coulson and S. Baichoo, "Implementing the CORBA GIOP in a High-Performance Object Request Broker Environment," *ACM Distributed Computing Journal*, vol. 14, Apr. 2001.

[10] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Using Principle Patterns to Optimize Real-time ORBs," *Concurrency Magazine*, vol. 8, no. 1, 2000.

[11] Gautam Thaker and Patrick Lardieri, "In Search of Commercial off the Shelf (COTS), Hard Real-time, Object Oriented Middleware," in *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA 2001)*, Sept. 2001.

[12] D. L. Levine, D. C. Schmidt, and S. Flores-Gaitan, "An Empirical Evaluation of OS Support for Real-time CORBA Object Request Brokers," in *Proceedings of Multimedia Computing and Networking 2000 (MMCN00)*, (San Jose, CA), ACM, Jan. 2000.

[13] D. L. Levine, S. Flores-Gaitan, C. D. Gill, and D. C. Schmidt, "Measuring OS Support for Real-time CORBA ORBs," in *Proceedings of the 4^{th} Workshop on Object-oriented Real-time Dependable Systems*, (Santa Barbara, CA), IEEE, Jan. 1999.

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

[15] S. Oikawa and R. Rajkumar, "Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior," in *Proceedings of the 5^{th} IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia), IEEE, June 1999.

[16] 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.

[17] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, Sept. 1999.

[18] D. C. Schmidt and S. D. Huston, *C++ Network Programming, Volume 1: Mastering Complexity With ACE and Patterns*. Boston: Addison-Wesley, 2002.

[19] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, vol. 21, no. 2, 2001.