Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures

Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha Gokhale
{schmidt,sumedh,sergio,gokhale}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA*

This paper appeared in the proceedings of 4^{th} IEEE Real-time Technology and Applications Symposium (RTAS), Denver, Colorado, June 3-5, 1998.

Abstract

There is increasing demand to extend Object Request Broker (ORB) middleware to support distributed applications with stringent real-time requirements. However, conventional ORB implementations, such as CORBA ORBs, exhibit substantial priority inversion and non-determinism, which makes them unsuitable for applications with deterministic real-time requirements. This paper provides two contributions to the study and design of real-time ORB middleware. First, it illustrates empirically why conventional ORBs do not yet support real-time quality of service. Second, it evaluates connection and concurrency software architectures to identify strategies that reduce priority inversion and non-determinism in real-time CORBA ORBs.

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

1 Introduction

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. The key levels in an ORB endsystem include the network adapters, OS I/O subsystems, communication protocols, ORB middleware, and higher-level services [1].

The main focus of this paper is on software architectures that are suitable for real-time ORB Cores. The ORB Core is the component in the CORBA reference model that manages transport connections, delivers client requests to an Object Adapter, and returns responses (if any) to clients. The ORB Core also typically implements the transport endpoint demultiplexing and concurrency architecture used by applications. Figure 1 illustrates how an ORB Core interacts with other CORBA components. [2] describes each of these components in more detail.

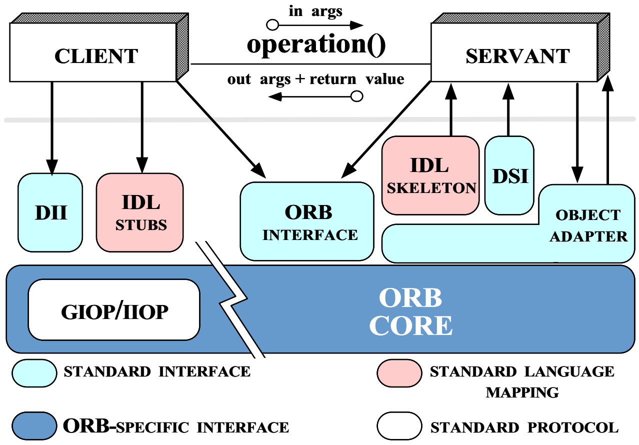

Figure 1: Components in the CORBA Reference Model

This paper is organized as follows: Section 2 presents empirical results from systematically measuring the efficiency and predictability of alternative ORB Core architectures in four contemporary CORBA implementations: CORBAplus, miniCOOL, MT-Orbix, and TAO; and Section 3 presents concluding remarks.

2 Real-time ORB Core Performance Experiments

This section describes the results of experiments that measure the real-time behavior of several commercial and research

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

ORBs, including IONA's MT-Orbix 2.2, Sun miniCOOL 4.3[1], Expersoft CORBAplus 2.1.1, and TAO 1.0. MT-Orbix and CORBAplus are not real-time ORBs, *i.e.*, they were not explicitly designed to support applications with real-time QoS requirements. Sun miniCOOL is a subset of the COOL ORB that is specifically designed for embedded systems with small memory footprints. TAO was designed at Washington University to support real-time applications with deterministic and statistical quality of service requirements, as well as best effort requirements.

2.1 Benchmarking Testbed

This section describes the experimental testbed we designed to systematically measure sources of latency and throughput overhead, priority inversion, and non-determinism in ORB endsystems. The architecture of our testbed is depicted in Figure 2. The hardware and software components in the experiments are outlined below.

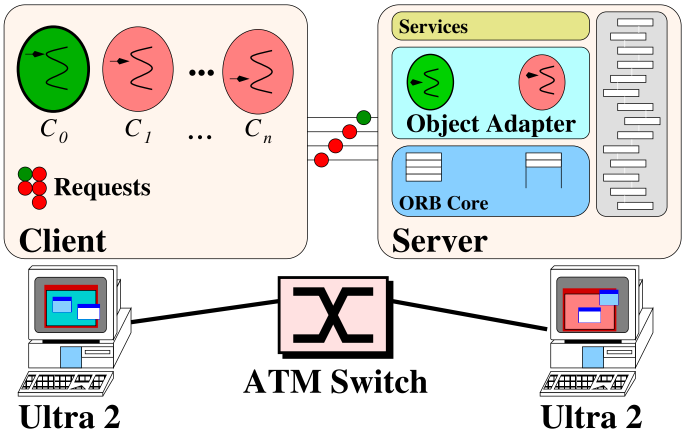

Figure 2: ORB Endsystem Benchmarking Testbed

2.1.1 Hardware Configuration

The experiments in this section were conducted using a FORE systems ASX-1000 ATM switch connected to two dual-processor UltraSPARC-2s running SunOS 5.5.1. The ASX-1000 is a 96 Port, OC12 622 Mbs/port switch. Each UltraSPARC-2 contains two 168 MHz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.5.1 TCP/IP protocol stack is implemented using the STREAMS communication framework.

Each UltraSPARC-2 has 256 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to eight switched virtual connections per card.

[1] COOL was previously developed by Chorus, which was recently acquired by Sun.

2.1.2 Client/Server Configuration and Benchmarking Methodology

Server benchmarking configuration: As shown in Figure 2, our testbed server consists of two servants within an ORB's Object Adapter. One servant runs in a higher priority thread than the other. Each thread processes requests that are sent to its servant by client threads on the other UltraSPARC-2.

Solaris real-time threads [3] are used to implement servant priorities. The high-priority servant thread has the *highest* real-time priority available on Solaris and the low-priority servant has the *lowest* real-time priority.

The server benchmarking configuration is implemented in the various ORBs as follows:

- **CORBAplus:** which uses the worker thread pool architecture shown in Figure 3. In version 2.1.1 of CORBAplus, multi-threaded applications have an event dispatcher thread and a pool of worker threads. The dispatcher thread receives the requests and passes them to application worker threads, which process the requests. In the simplest configuration, an application can choose to create no additional threads and rely upon the main thread to process all requests.

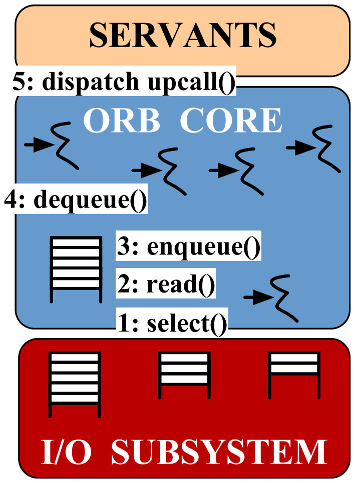

Figure 3: CORBAplus' Worker Thread Pool Concurrency Architecture

- **miniCOOL:** which uses the leader/follower thread pool architecture shown in Figure 4. Version 4.3 of miniCOOL al-

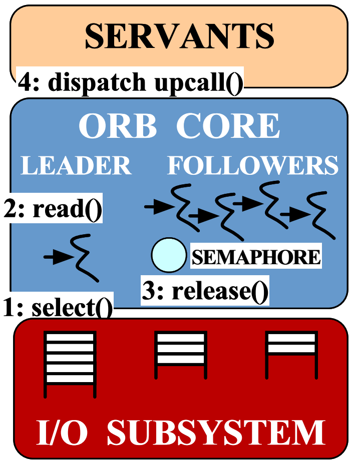

Figure 4: miniCOOL's Leader/Follower Concurrency Architecture

lows application-level concurrency control. The application developer can choose between thread-per-request or thread-pool. The thread-pool concurrency architecture was used for our benchmarks since it is better suited than thread-per-request for deterministic real-time applications. In the thread-pool concurrency architecture, the application initially spawns a fixed number of threads. In addition, when the initial thread pool size is insufficient, miniCOOL can be configured to dynamically spawn threads on behalf of server applications to handle requests, up to a maximum limit.

- **MT-Orbix:** which uses the thread pool framework architecture based on the Chain of Responsibility pattern shown in Figure 5. Version 2.2 of MT-Orbix is used to create two real-time servant threads at startup. The high-priority thread is associated with the high-priority servant and the low-priority thread is associated with the low-priority servant. Incoming requests are assigned to these threads using the Orbix thread filter mechanism. Each priority has its own queue of requests to avoid priority inversion within the queue. This inversion could otherwise occur if a high-priority servant and a low-priority servant dequeue requests from the same queue.

- **TAO:** which uses the thread-per-priority concurrency architecture described in [4]. Version 1.0 of TAO integrates the thread-per-priority concurrency architecture with a non-multiplexed connection architecture, as shown in Figure 6. In contrast, the other three ORBs multiplex all requests from client threads in each process over a single connection to the server process.

Client benchmarking configuration: Figure 2 shows how the benchmarking test used one high-priority client C_0 and n low-priority clients, $C_1 \ldots C_n$. The high-priority client runs in a high-priority real-time OS thread and invokes operations

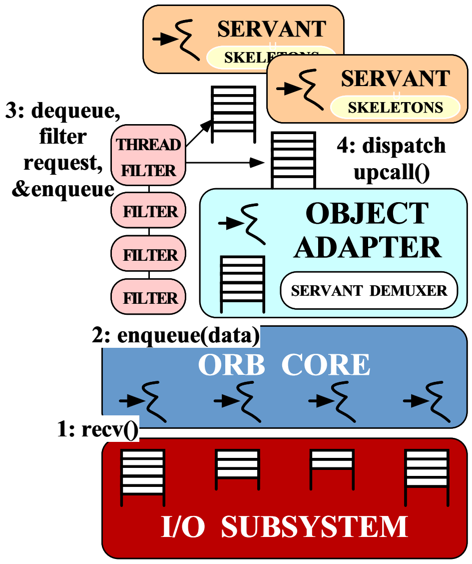

Figure 5: MT-Orbix's Thread Framework Concurrency Architecture

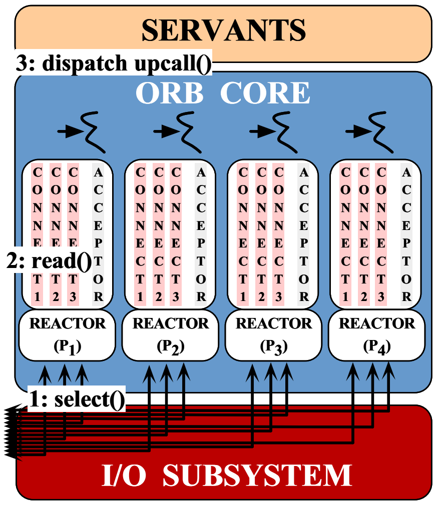

Figure 6: TAO's Thread-per-Priority Thread Pool Architecture

at 20 Hz, *i.e.*, it invokes 20 CORBA twoway calls per second. All low-priority clients have the same lower priority OS thread priority and invoke operations at 10 Hz, *i.e.*, they invoke 10 CORBA twoway calls per second. 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, they block on a barrier lock so that no client begins work 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 arbitrary order determined by the Solaris real-time thread dispatcher. Each client invokes 4,000 CORBA twoway requests at its prescribed rate.

2.2 Performance Results on Solaris

Two categories of tests were used in our benchmarking experiments: *blackbox* and *whitebox*.

Blackbox benchmarks: We computed the average twoway response time incurred by various clients. In addition, we computed twoway operation jitter, which is the standard deviation from the average twoway response time. High levels of latency and jitter are undesirable for deterministic real-time applications since they complicate the computation of worst-case execution time and reduce CPU utilization. Section 2.2.1 explains the blackbox results.

Whitebox benchmarks: To precisely pinpoint the *source* of priority inversion and performance non-determinism, we employed whitebox benchmarks. These benchmarks used profiling tools such as UNIX `truss` and `Quantify` [5]. These tools trace and log the activities of the ORBs and measure the time spent on various tasks, as explained in Section 2.2.2.

Together, the blackbox and whitebox benchmarks indicate the end-to-end latency/jitter incurred by CORBA clients and help explain the reason for these results. In general, the results reveal why ORBs like MT-Orbix, CORBAplus, and miniCOOL are not yet suited for applications with deterministic real-time performance requirements. Likewise, the results illustrate empirically how and why the non-multiplexed, priority-based ORB Core architecture used by TAO is more suited for these types of real-time applications.

2.2.1 Blackbox Results

As the number of low-priority clients increases, the number of low-priority requests sent to the server also increases. Ideally, a real-time ORB endsystem should exhibit no variance in the latency observed by the high-priority client, irrespective of the number of low-priority clients. Our measurements of end-to-end twoway ORB latency yielded the results in Figure 7.

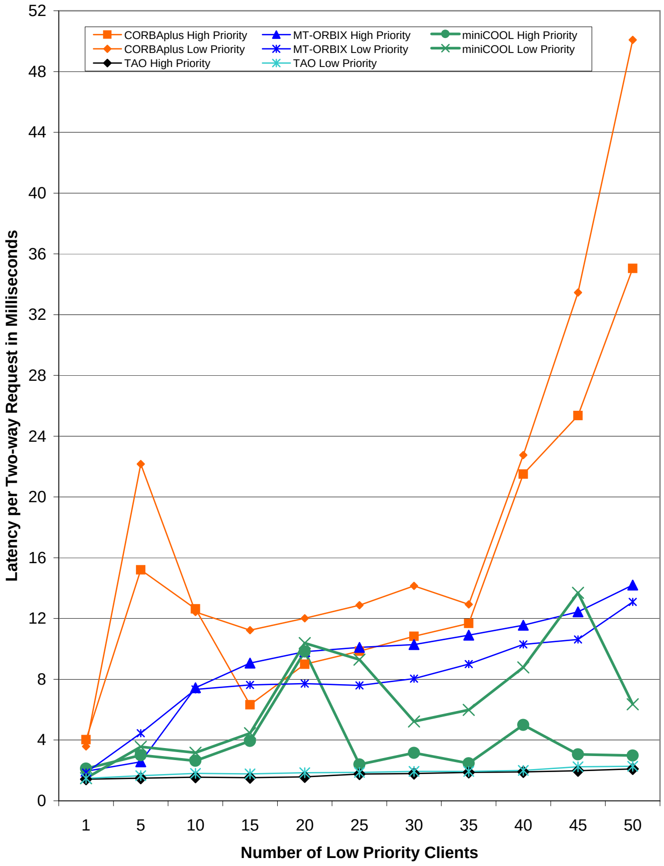

Figure 7: Comparative Latency for CORBAplus, MT-Orbix, miniCOOL, and TAO

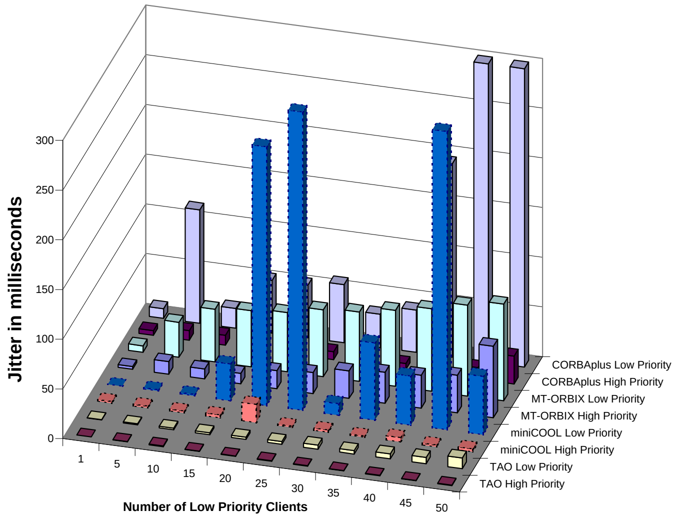

Figure 8: Comparative Jitter for CORBAplus, MT-Orbix, miniCOOL and TAO

Figure 7 shows that as the number of low-priority clients increases, MT-Orbix and CORBAplus incur significantly higher latencies for their high-priority client thread. Compared with TAO, MT-Orbix's latency is 7 times higher and CORBAplus' latency is 25 times higher. Note the irregular behavior of the average latency that miniCOOL displays, *i.e.*, from 10 msec latency running 20 low-priority clients down to 2 msec with 25 low-priority clients. Such source of non-determinism is clearly undesirable for real-time bounds.

The low-priority clients for MT-Orbix, CORBAplus and miniCOOL also exhibit very high levels of jitter. Compared with TAO, CORBAplus incurs 300 times as much jitter and MT-Orbix 25 times as much jitter in the worst case, as shown in Figure 8. Likewise, miniCOOL's low-priority clients display an erratic behavior with several high bursts of jitter, which makes it undesirable for deterministic real-time applications.

The blackbox results for each ORB are explained below.

CORBAplus results: CORBAplus incurs priority inversion at various points in the graph shown in Figure 7. After displaying a high amount of latency for a small number of low-priority clients, the latency drops suddenly at 10 clients, then eventually rises again. Clearly, this behavior is not suitable for deterministic real-time applications. Section 2.2.2 reveals how the poor performance and priority inversions stem largely from CORBAplus' concurrency architecture. Figure 8 shows that CORBAplus generates high levels of jitter, particularly when tested with 40, 45, and 50 low-priority clients. These results show an erratic and undesirable behavior for applications that require real-time guarantees.

MT-Orbix results: MT-Orbix incurs substantial priority inversion as the number of low-priority clients increase. After the number of clients exceeds 10, the high-priority client performs increasingly worse than the low-priority clients. This behavior is not conducive to deterministic real-time applications. Section 2.2.2 reveals how these inversions stem largely from the MT-Orbix's concurrency architecture on the server. In addition, MT-Orbix produces high levels of jitter, as shown in Figure 8. This behavior is caused by priority inversions in its ORB Core, as explained in Section 2.2.2.

miniCOOL results: As the number of low-priority clients increase, the latency observed by the high-priority client also increases, reaching $\sim$10 msec, at 20 clients, at which point it decreases suddenly to 2.5 msec with 25 clients. This erratic behavior becomes more evident as more low-priority clients are run. Although the latency of the high-priority client is smaller than the low-priority clients, the non-linear behavior of the clients makes miniCOOL problematic for deterministic real-time applications.

The difference in latency between the high- and the low-priority client is also non-deterministic. For instance, it grows from 0.55 msec to 10 msec. Section 2.2.2 reveals how this behavior stems largely from the connection architecture used by the miniCOOL client and server.

The jitter incurred by miniCOOL is also fairly high, as shown in Figure 8. This jitter is similar to that observed by the CORBAplus ORB since both spend approximately the same percentage of time executing locking operation. section 2.2.2 evaluates ORB locking behavior.

TAO results: Figure 7 reveals that as the number of low-priority clients increase from 1 to 50, the latency observed by TAO's high-priority client grows by $\sim$0.7 msecs. However, the difference between the low-priority and high-priority clients starts at 0.05 msec and ends at 0.27 msec. In contrast, in miniCOOL, it evolves from 0.55 msec to 10 msec, and in CORBAplus it evolves from 0.42 msec to 15 msec. Moreover, the rate of increase of latency with TAO is significantly lower than MT-Orbix, Sun miniCOOL, and CORBAplus. In particular, when there are 50 low-priority clients competing for the CPU and network bandwidth, the low-priority client latency observed with MT-Orbix is more than 7 times that of TAO, the miniCOOL latency is $\sim$3 times that of TAO, and CORBAplus is $\sim$25 times that of TAO.

In contrast to the other ORBs, TAO's high-priority client always performs better than its low-priority clients. This demonstrates that the connection and concurrency architectures in TAO's ORB Core can maintain real-time request priorities end-to-end. The key difference between TAO and other ORBs is that its GIOP protocol processing is performed on a dedicated connection by a dedicated real-time thread with a suitable end-to-end real-time priority. Thus, TAO shares the minimal amount of ORB endsystem resources, which substantially reduces opportunities for priority inversion and locking overhead.

The TAO ORB produces very low jitter (less than 11 msecs) for the low-priority requests and negligible jitter (less than 1 msec) for the high-priority requests. The stability of TAO's latency is clearly desirable for applications that require predictable end-to-end performance.

In general, the blackbox results described above demonstrate that improper choice of ORB Core concurrency and connection software architectures can play a significant role in exacerbating priority inversion and non-determinism.

2.2.2 Whitebox Results

For the whitebox tests, we used a configuration of ten concurrent clients similar to the one described in Section 2.1. Nine clients were low-priority and one was high-priority. Each client sent 4,000 twoway requests to the server, which had a low-priority servant and high-priority servant thread.

Our previous experience using CORBA for real-time avionics mission computing [6] indicated that locks constitute a significant source of overhead, non-determinism and potential priority inversion for real-time ORBs. Using `Quantify` and `truss`, we measured the time the ORBs consumed performing tasks like synchronization, I/O, and protocol processing.

In addition, we computed a metric that records the number of calls made to user-level locks (*i.e.*, `mutex_lock` and `mutex_unlock`) and kernel-level locks (*i.e.*, `_lwp_mutex_lock`, `_lwp_mutex_unlock`, `_lwp_sema_post` and `_lwp_sema_wait`). This metric computes the average number of lock operations per-request. In general, kernel-level locks are considerably more expensive since they incur kernel/user mode switching overhead.

The whitebox results from our experiments are presented below.

CORBAplus whitebox results: Our whitebox analysis of CORBAplus reveals high levels of synchronization overhead from mutex and semaphore operations at the user-level for each twoway request, as shown in Figure 13. Synchronization overhead arises from locking operations that implement the connection and concurrency architecture used by CORBAplus.

As shown in Figure 9, CORBAplus exhibits high synchronization overhead (52%) using kernel-level locks in the client and the server incurs high levels of processing overhead (45%) due to kernel-level lock operations.

For each CORBA request/response, CORBAplus's client ORB performs 199 lock operations, whereas the server performs 216 user-level lock operations, as shown in Figure 13. This locking overhead stems largely from excessive dynamic memory allocation, as described in Section 2.3. Each dynamic allocation causes two user-level lock operations, *i.e.*, one acquire and one release.

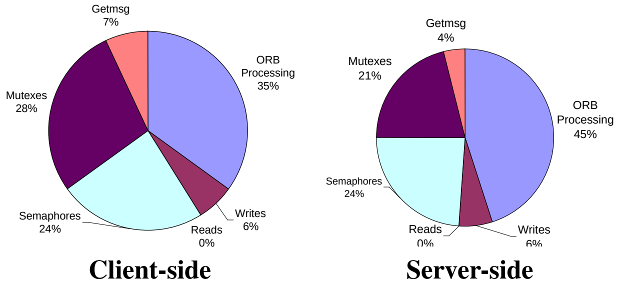

Figure 9: Whitebox Results for CORBAplus

The CORBAplus connection and concurrency architectures are outlined briefly below.

- **CORBAplus connection architecture:** The CORBAplus ORB connection architecture multiplexes all requests to the same server through one active connection thread, which simplifies ORB implementations by using a uniform queueing mechanism.

- **CORBAplus concurrency architecture:** The CORBAplus ORB concurrency architecture uses the thread pool architecture with a single I/O thread to `accept` and `read` requests from socket endpoints. This thread inserts the request on a queue that is serviced by a pool of worker threads.

The CORBAplus connection architecture and the server concurrency architecture help reduce the number of simultaneous open connections and simplify the ORB implementation. However, concurrent requests to the shared connection incur high overhead since each send operation incurs a context switch. In addition, on the client-side, threads of different priorities can share the same transport connection, which can cause priority inversion. For instance, a high-priority thread may be blocked until a low-priority thread finishes sending its request. Likewise, the priority of the thread that blocks on the semaphore to receive a reply from a twoway connection may not reflect the priority of the *request* that arrives from the server, thereby causing additional priority inversion.

miniCOOL whitebox results: Our whitebox analysis of miniCOOL reveals that synchronization overhead from mutex and semaphore operations consume a large percentage of the total miniCOOL ORB processing time. As with CORBAplus, synchronization overhead in miniCOOL arises from locking operations that implement its connection and concurrency architecture. Locking overhead accounted for $\sim$50% on the client-side and more than 40% on the server-side, as shown in Figure 10).

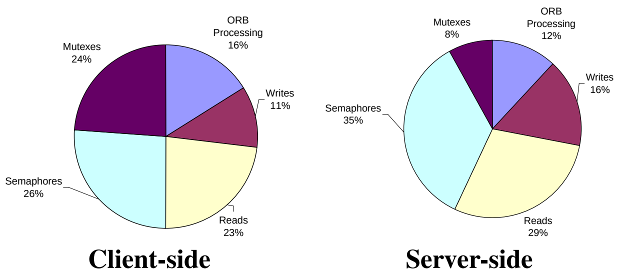

Figure 10: Whitebox Results for miniCOOL

For each CORBA request/response, miniCOOL's client ORB performs 94 lock operations at the user-level, whereas the server performs 231 lock operations, as shown in Figure 13. As with CORBAplus, this locking overhead stems largely from excessive dynamic memory allocation. Each dynamic allocation causes two user-level lock operations, *i.e.*, one acquire and one release.

The number of calls per-request to kernel-level locking mechanisms at the server (shown in Figure 14) are unusually high. This overhead stems from the fact that miniCOOL uses "system scoped" threads on Solaris, which require kernel intervention for all synchronization operations [7].

The miniCOOL connection and concurrency architectures are outlined briefly below.

• **miniCOOL connection architecture:** The miniCOOL ORB connection architecture uses a "leader/follower" model that allows the leader thread to block in `select` on the shared socket. All following threads block on semaphores waiting for one of two conditions: (1) the leader thread will `read` their reply message and signal their semaphore or (2) the leader thread will `read` its own reply and signal another thread to enter and block in `select`, thereby becoming the new leader.

• **miniCOOL concurrency architecture:** The Sun miniCOOL ORB concurrency architecture also uses a leader/follower model that waits for connections in a single thread. Whenever a request arrives and validation of the request is complete, the leader thread (1) signals a follower thread in the pool to wait for incoming requests and (2) services the request.

The miniCOOL connection architecture and the server concurrency architecture help reduce the number of simultaneous open connections and the amount of context switching when replies arrive in FIFO order. As with CORBAplus, however, this design yields high levels of priority inversion. For instance, threads of different priorities can share the same transport connection on the client-side. Therefore, a high-priority thread may block until a low-priority thread finishes sending its request. In addition, the priority of the thread that blocks on the semaphore to access a connection may not reflect the priority of the *response* that arrives from the server, which yields additional priority inversion.

MT-Orbix whitebox results: Figure 11 shows the whitebox results for the client-side and server-side of MT-Orbix.

• **MT-Orbix connection architecture:** Like miniCOOL, MT-Orbix uses the leader/follower multiplexed connection architecture. Although this model minimizes context switching overhead, it causes intensive priority inversions.

• **MT-Orbix concurrency architecture:** In the MT-Orbix implementation of our benchmarking testbed, multiple servant threads were created, each with the appropriate priority, *i.e.*, the high-priority servant had the highest priority thread. A thread filter was then installed to look at each request, determine the priority of the request (by examining the

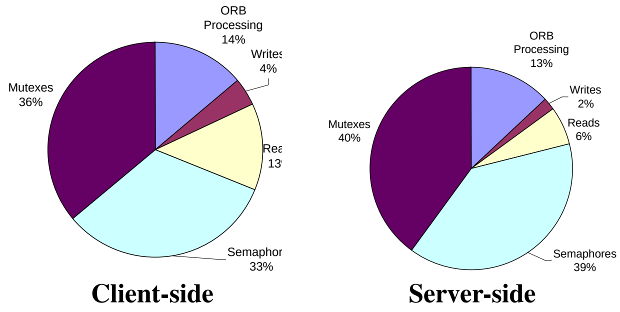

Figure 11: Whitebox Results for MT-Orbix

target object), and pass the request to the thread with the correct priority. The thread filter mechanism is implemented by a high-priority real-time thread to minimize dispatch latency.

The thread pool instantiation of the MT-Orbix mechanism is flexible and easy to use. However, it suffers from high levels of priority inversion and synchronization overhead. MT-Orbix provides only *one* filter chain. Thus, all incoming requests must be processed sequentially by the filters before they are passed to the servant thread with an appropriate real-time priority. As a result, if a high-priority request arrives after a low-priority request, it must wait until the low-priority request has been dispatched before the ORB processes it.

In addition, a filter can only be called after (1) GIOP processing has completed and (2) the Object Adapter has determined the target object for this request. This processing is serialized since the MT-Orbix ORB Core is unaware of the request priority. Thus, a higher priority request that arrived after a low-priority request must wait until the lower priority request has been processed by MT-Orbix.

MT-Orbix's concurrency architecture is chiefly responsible for its substantial priority inversion shown in Figure 7. This figure shows how the latency observed by the high-priority client increases rapidly, growing from $\sim$2 msecs to $\sim$14 msecs as the number of low-priority clients increase from 1 to 50.

The MT-Orbix filter mechanism also causes an increase in synchronization overhead. Because there is just one filter chain, concurrent requests must acquire and release locks to be processed by the filter. The MT-Orbix client-side performs 175 user-level lock operations per-request, while the server-side performs 599 user-level lock operations per-request, as shown in Figure 13. Moreover, MT-Orbix displays a high number of kernel-level locks per-request, as shown in Figure 14.

TAO whitebox results: As shown in Figure 12, TAO exhibits negligible synchronization overhead. TAO performs 41 user-level lock operations per-request on the client-side, and 100 user-level lock operations per-request on the server-side.

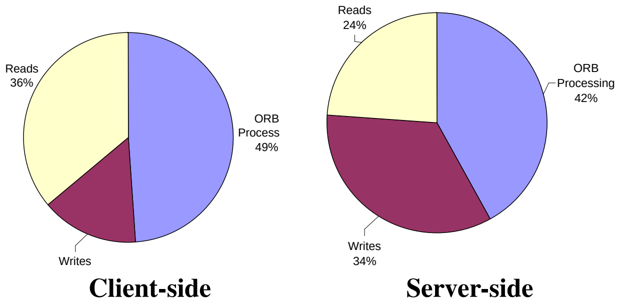

Figure 12: Whitebox Results for TAO

This low amount of synchronization results from the design of TAO's ORB Core, which allocates a separate connection for each priority, as shown in Figure 6. Therefore, TAO's ORB Core minimizes additional user-level locking operations per-request and uses no kernel-level locks in its ORB Core.

- **TAO connection architecture:** TAO uses a non-multiplexed connection architecture, which pre-establishes connections to servants, as described in [4]. One connection is pre-established for each priority level, thereby avoiding the non-deterministic delay involved in dynamic connection setup. In addition, different priority levels have their own connection. This design avoids request-level priority inversion, which would otherwise occur from FIFO queueing *across* client threads with different priorities.

- **TAO concurrency architecture:** TAO supports several concurrency architectures, as described in [4]. The *thread-per-priority* architecture was used for the benchmarks in this paper. In this concurrency architecture, a separate thread is created for each priority level *i.e.*, each rate group. Thus, the low-priority client issues CORBA requests at a lower rate than the high-priority client (10 Hz vs. 20 Hz, respectively).

On the server-side, client requests sent to the high-priority servant are processed by a high-priority real-time thread. Likewise, client requests sent to the low-priority servant are handled by the low-priority real-time thread. Locking overhead is minimized since these two servant threads share minimal ORB resources, *i.e.*, they have separate `Reactors`, `Acceptors`, Object Adapters, etc. In addition, the two threads service separate client connections, thereby eliminating the priority inversion that would otherwise arises from connection multiplexing, as exhibited by the other ORBs we tested.

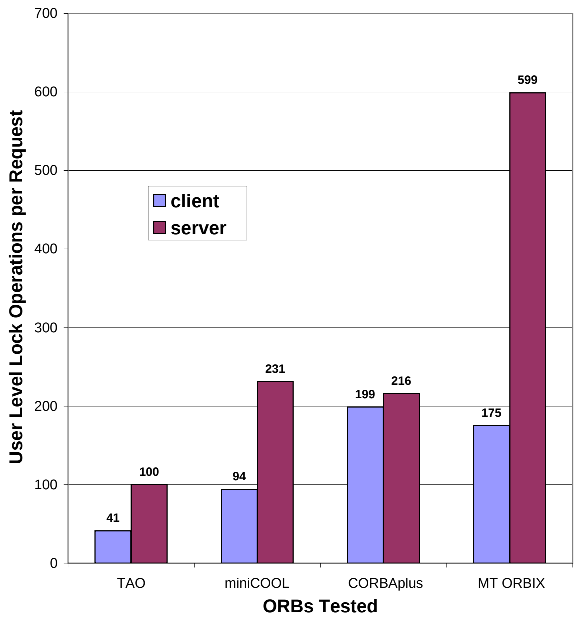

Figure 13: User-level Locking Overhead in ORBs

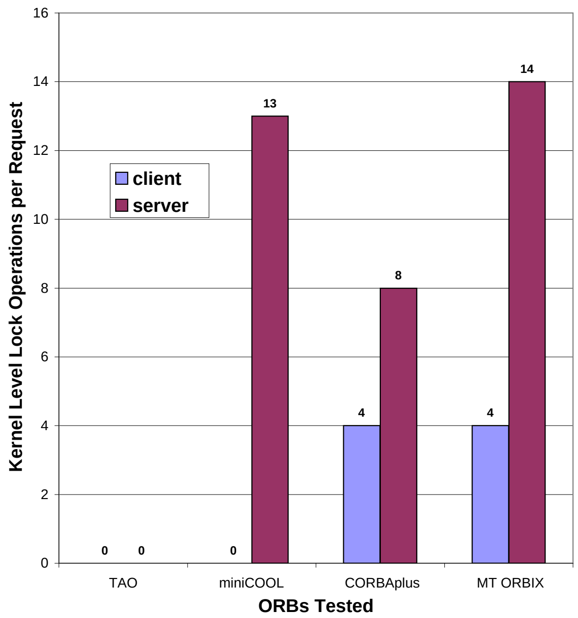

Figure 14: Kernel-level Locking Overhead in ORBs

Locking overhead: Our whitebox tests measured user-level locking overhead (shown in Figure 13) and kernel-level locking overhead (shown in Figure 14) in the CORBAplus, MT-Orbix, miniCOOL, and TAO ORBs. User-level locks are typically used to protect shared resources within a process. A common example is dynamic memory allocation using global C++ operators `new` and `delete`. These operators allocate memory from a globally managed heap in each process.

Kernel-level locks are more expensive since they typically require mode switches between user-level and the kernel. The semaphore and mutex operations depicted in the whitebox results for the ORBs evaluated above arise from kernel-level lock operations.

TAO limits user-level locking by using buffers that are pre-allocated off the run-time stack. This buffer is subdivided to accommodate the various fields of the request. Kernel-level locking is limited due to the fact that TAO can be configured so that ORB resources are not shared between its threads.

2.3 Evaluation and Recommendations

The results of our benchmarks illustrate the non-deterministic performance incurred by applications running atop conventional ORBs. In addition, the results show that priority inversion and non-determinism are significant problems in conventional ORBs. As a result, these ORBs are not currently suitable for applications with deterministic real-time requirements. Based on our results, and our prior experience [8, 9, 10, 11] measuring the performance of CORBA ORB endsystems, we suggest the following recommendations to decrease non-determinism and limit priority inversion in real-time ORB endsystems.

1. Real-time ORBs should avoid dynamic connection establishment: ORBs that establish connections dynamically suffer from high jitter. Thus, performance seen by individual clients can vary significantly from the average. Neither CORBAplus, miniCOOL, nor MT-Orbix provide APIs for pre-establishing connections; TAO provides these APIs as extensions to CORBA.

We recommend that APIs to control the pre-establishment of connections should be defined as an OMG standard for real-time CORBA [12, 13].

2. Real-time ORBs should avoid multiplexing requests of different priorities over a shared connection: Sharing connections requires synchronization. Thus, high-priority requests can be blocked until low-priority threads release the shared connection lock. Moreover, priority inversion is exacerbated if multiple thread with multiple levels of thread priorities share common locks. For instance, medium priority threads can preempt a low priority thread that is holding a lock required by a high priority thread, which can lead to unbounded priority inversion [3].

We recommend that real-time ORBs should allow application developers to determine whether requests with different priorities are multiplexed over shared connections. Currently, neither miniCOOL, CORBAplus, nor MT-Orbix support this level of control, though TAO provides this model by default.

3. Real-time ORBs should minimize dynamic memory allocation: Thread-safe implementations of dynamic memory allocators require user-level locking. For instance, the C++ `new` operator allocates memory from a global pool shared by all threads in a process. Likewise, the C++ `delete` operation, that releases allocated memory, also requires user-level locking to update the global shared pool. This lock sharing contributes to the overhead shown in Figure 13.

We recommend that real-time ORBs avoid excessive sharing of dynamic memory locks via the use of OS features such as thread-specific storage [14], which allocates memory from separate heaps that are unique to each thread.

4. Real-time ORB concurrency architectures should be flexible, yet efficient and predictable: Many ORBs, such as miniCOOL and CORBAPlus, create threads on behalf of server applications. This design prevents application developers from customizing ORB performance by selecting a custom concurrency architecture. Conversely, other ORB concurrency architectures are flexible, but inefficient and non-deterministic, as shown by Section 2.2.2's explanation of the MT-Orbix performance results. Thus, a balance is needed between flexibility and efficiency.

We recommend that real-time ORBs provide APIs that allow application developers to select concurrency architectures that are flexible, efficient, *and* predictable. For instance, TAO offers a range of concurrency architectures (such as thread-per-priority, thread pool, and thread-per-connection) that can selectively use thread-specific storage to minimize unnecessary sharing of ORB resources.

5. Real-time ORB endsystem architectures should be guided by empirical performance benchmarks: Our prior research on pinpointing performance bottlenecks and optimizing middleware like Web servers [15, 16] and CORBA ORBs [9, 8, 11, 10] demonstrates the efficacy of a measurement-driven research methodology. We recommend that the OMG adopt standard real-time CORBA benchmarking techniques and metrics. These benchmarks will simplify the communication and comparison of performance results and real-time ORB behavior patterns. The real-time ORB benchmarking test suite described in this section is available at `www.cs.wustl.edu/~schmidt/TAO.html`.

3 Concluding Remarks

Conventional CORBA ORBs exhibit substantial priority inversion and non-determinism. Consequently, they are not yet suitable for distributed, real-time applications with deterministic QoS requirements. Meeting these demands requires that ORB Core software architectures be designed to reduce priority inversion and increase end-to-end determinism.

The TAO ORB Core described in this paper reduces priority inversion and enhances determinism by using a priority-based concurrency architecture and non-multiplexed connection architecture that share a minimal amount of resources among threads. The architectural principles used in TAO can be applied to other ORBs and other real-time software systems.

TAO has been used to develop a number of real-time applications, including a real-time audio/video streaming service [17] and a real-time ORB endsystem for avionics mission computing [6]. The avionics application manages sensors and operator displays, navigate the aircraft's course, and control weapon release. To meet the scheduling demands of real-time applications, TAO supports predictable scheduling and dispatching of periodic processing operations [1], as well as efficient event filtering and correlation mechanisms [6]. The C++ source code for TAO and ACE is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`.

Acknowledgments

We gratefully acknowledge Expersoft, IONA, and Sun for providing us with their ORB software for the benchmarking testbed. In addition, we would like to thank Frank Buschmann and Bil Lewis for their comments on this paper.

References

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

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

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

[4] 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.

[5] P. S. Inc., *Quantify User's Guide*. PureAtria Software Inc., 1996.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] A. Gokhale and D. C. Schmidt, "Evaluating Latency and Scalability of CORBA Over High-Speed ATM Networks," in *Proceedings of the International Conference on Distributed Computing Systems*, (Baltimore, Maryland), IEEE, May 1997.

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

[13] Object Management Group, *Realtime CORBA 1.0 Request for Proposals*, OMG Document orbos/97-09-31 ed., September 1997.

[14] D. C. Schmidt, T. Harrison, and N. Pryce, "Thread-Specific Storage – An Object Behavioral Pattern for Accessing per-Thread State Efficiently," in *The* 4^{th} *Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-34)*, September 1997.

[15] J. Hu, I. Pyarali, and D. C. Schmidt, "Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks," in *Proceedings of the* 2^{nd} *Global Internet Conference*, IEEE, November 1997.

[16] J. Hu, S. Mungee, and D. C. Schmidt, "Principles for Developing and Measuring High-performance Web Servers over ATM," in *Proceeedings of INFOCOM '98*, March/April 1998.

[17] Object Management Group, *Control and Management of Audio/Video Streams: OMG RFP Submission*, 1.2 ed., Mar. 1997.