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

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

This paper has been submitted to the 4^{th} IEEE Real-time Technology and Applications Symposium (RTAS), Denver, Colorado, June 3-5, 1998.

Abstract

There is increasing demand to extend CORBA to support applications with stringent real-time requirements. However, conventional CORBA Object Request Brokers (ORBs) exhibit substantial priority inversion and non-determinism, which makes them unsuitable for applications with deterministic real-time requirements. This paper focuses on software architectures that help to alleviate priority inversion and non-determinism in real-time CORBA ORBs. It also illustrates empirically why conventional ORBs do not yet support real-time quality of service.

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

1 Introduction

1.1 Emerging Trends in Distributed Real-time Systems

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

Increased focus on middleware and integration frameworks: There is a general industry trend away from *programming* real-time applications from scratch to *integrating* applications using reusable components based on object-oriented (OO) middleware [3].

Increased focus on QoS-enabled components and open systems: There is increasing demand for remote method invocation and messaging technology to simplify the collaboration of open distributed application components [4] that possess stringent QoS requirements.

Increased focus on standardizing real-time middleware: Several international efforts are currently addressing QoS for OO middleware. The most prominent is the OMG CORBA standardization effort [5]. CORBA is OO middleware that allows clients to invoke operations on objects without concern for where the objects reside, what language the objects are written in, what OS/hardware platform they run on, or what communication protocols and networks are used to interconnect distributed objects [6].

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

Notwithstanding the significant efforts of the OMG RT SIG, however, developing and standardizing distributed real-time CORBA ORBs remains hard. There are few, if any, successful exemplars of standard, commercially available distributed real-time ORB middleware. In particular, conventional CORBA ORBs are not well suited for performance-sensitive, distributed real-time applications due to (1) lack of QoS specification interfaces, (2) lack of QoS enforcement, (3) lack of real-time programming features, and (4) general lack of performance and predictability [9].

Although some operating systems, networks, and protocols now support real-time scheduling, they do not provide integrated end-to-end real-time ORB endsystem solutions [10]. Moreover, relatively little systems research has focused on

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

strategies and tactics for real-time CORBA. In particular, QoS research at the network and OS layers has not addressed key requirements and programming aspects of CORBA middleware. For instance, research on QoS for ATM networks has focused largely on policies for allocating bandwidth on a per-connection basis [11]. Likewise, research on real-time operating systems has focused largely on avoiding priority inversions and non-determinism in synchronization and scheduling mechanisms for multi-threaded applications [12].

1.2 Towards Real-time CORBA

We believe that developing real-time CORBA requires a systematic, measurement-driven methodology to identify and alleviate sources of ORB endsystem overhead, priority inversion, and non-determinism. The ORB software architectures presented in this paper are based on our experience developing, profiling, and optimizing next-generation avionics [13] and telecommunications [14] systems using OO real-time middleware such as ACE [15] and TAO [10].

ACE is an OO framework that implements core concurrency and distribution patterns [16] for communication software. It provides reusable C++ wrapper facades and framework components that support high-performance, real-time applications across a wide range of OS platforms, including Win32, most versions of UNIX, and real-time operating systems like VxWorks, Chorus Classix, pSoS, and LynxOS. TAO is a highly extensible ORB endsystem written with ACE. TAO is targeted for applications with deterministic and statistical quality of service requirements, as well as best effort requirements.

The TAO project focuses on the following topics related to real-time CORBA and ORB endsystems:

- Identifying the requisite enhancements to standard ORB specifications, such as OMG CORBA, that will enable applications to specify their Quality of Service (QoS) requirements to ORB endsystems [17].
- Empirically determining the features required to build real-time ORB endsystems that can enforce deterministic and statistical end-to-end applications QoS guarantees [10].
- Integrating the strategies for I/O subsystem architectures and optimizations [18] with ORB endsystems to provide end-to-end bandwidth, latency, and reliability guarantees to distributed applications.
- Capturing and documenting the key design patterns [19] necessary to develop, maintain, and extend real-time ORB endsystems.

Our earlier work on CORBA and TAO explores aspects of real-time ORB endsystem design including real-time scheduling [10], real-time request demultiplexing [20], real-time event processing [13], and real-time I/O subsystem integration [18]. This paper focuses on a previously unexamined point in the real-time ORB endsystem design space: *software architectures that minimize priority inversion and non-determinism in CORBA ORB Core implementations*.

An 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 *event demultiplexing* and *concurrency* architecture used by applications. Figure 1 illustrates how an ORB Core interacts with other CORBA components.

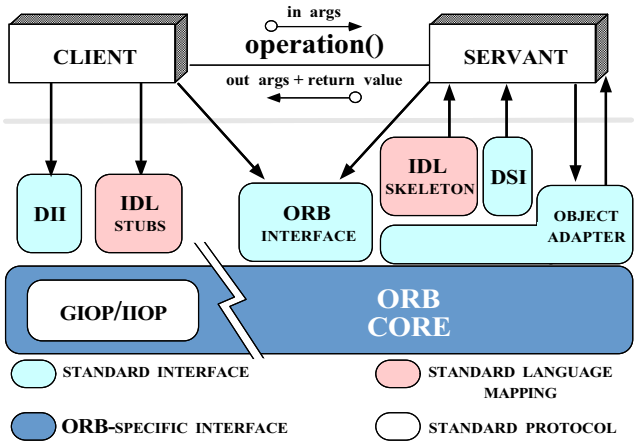

Figure 1: Components in the CORBA Reference Model

This paper is organized as follows: Section 2 outlines the general factors that impact real-time ORB endsystem performance and predictability; Section 3 focuses on software architectures for real-time ORB Cores, describing alternative ORB Core concurrency and connection architectures; Section 4 presents empirical results from systematically measuring the efficiency and predictability of alternative ORB Core architectures in Sun COOL[1], MT-Orbix, and TAO; Section 5 compares our research with related work; and Section 6 presents concluding remarks.

2 Factors Impacting Real-time ORB Endsystem Performance

Meeting the QoS needs of next-generation distributed applications requires much more than defining IDL interfaces or adding preemptive real-time scheduling into an OS. Instead, it requires a vertically and horizontally integrated *ORB endsystem architecture* that can deliver end-to-end QoS guarantees at

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

multiple levels throughout a distributed system. The key levels in an ORB endsystem include the network adapters, operating system I/O subsystems, communication protocols, and ORB middleware services.

The main thrust of this paper is on software architectures that are suitable for real-time ORB Cores. For completeness, Section 2.1 briefly outlines the general sources of overhead in ORB endsystems. Section 2.2 then describes the key sources of priority inversion and non-determinism that affect real-time ORB endsystems. After this overview, Section 3 focuses even more specifically on alternative ORB Core concurrency and connection architectures.

2.1 General Sources of ORB Endsystem Overhead

Our prior experience [21, 22, 20] measuring the throughput and latency of CORBA ORBs indicated that the performance overhead of real-time ORB endsystems stem from inefficiencies in the following:

1. Network connections and host-network interfaces: These endsystem components handle heterogeneous network connections and bandwidths, which can significantly affect latencies and cause variability in performance. Inefficient design of host-network interfaces can cause queueing delays and in some cases lost packets [23], which is unacceptable in hard real-time systems.

2. Communication protocol implementations and integration with the I/O subsystem and network interfaces: Inefficient protocol implementations and improper integration with I/O subsystems can adversely affect endsystem performance. Specific factors that cause problems include the protocol overhead caused by flow control, congestion control, retransmission strategies, and connection management. Likewise, lack of proper I/O subsystem integration yields excessive data copying, fragmentation, reassembly, context switching, synchronization, checksumming, demultiplexing, marshaling, and demarshaling overhead [24].

3. ORB transport protocols: Inefficient implementation of ORB transport protocols *e.g.*, the CORBA Internet Interoperability Protocol (such as IIOP) [5] and Simple Flow Protocol (SFP) [25] can cause priority inversions. Specific factors responsible for these inversions include improper connection management strategies, improper sharing of resources, and excessive synchronization overhead in ORB protocol implementations.

4. ORB implementations and integration with OS services: ORB software designs can yield a large number of memory accesses, cache misses, heap allocation and deallocation, and context switches [26]. These factors can increase latency and jitter, which is unacceptable for distributed systems with deterministic real-time requirements. Specific factors that can cause problems include: data copying, fragmentation/reassembly, context switching, synchronization, checksumming, socket demultiplexing, timer handling, request demultiplexing, marshaling/demarshaling, framing, error checking, connection and concurrency architectures. Many of these problems are similar to those listed in bullet 2. However, they occur at the user-level rather than at the kernel-level, which often make it easier for ORB implementers to solve them in a portable manner.

Figure 2 illustrates where these various factors impact ORB performance and where optimizations must be applied to alleviate key sources of ORB endsystem overhead, priority inversion, and non-determinism. Below, we focus on the sources of overhead in ORB endsystems that are chiefly responsible for priority inversions and non-determinism.

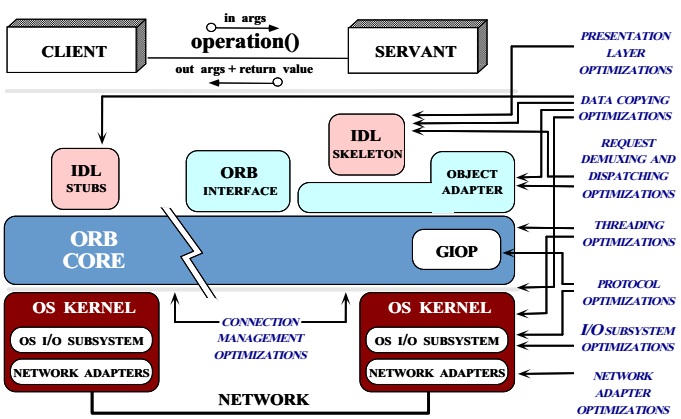

Figure 2: Optimizing Real-time ORB Endsystem Performance

2.2 Sources of Priority Inversion and Non-determinism in ORB Endsystems

Sources of priority inversion and non-determinism in ORB endsystems generally stem from resources that are shared by multiple threads or processes. Common examples of shared ORB resources include (1) TCP connections used by CORBA inter-ORB protocols such as GIOP and IIOP, (2) threads that manage request transfer through client and server end-points, (3) global dynamic memory allocators, and (4) internal ORB data structures like connection tables and socket/request demultiplexing maps. Below, we classify key sources of priority inversion and non-determinism in conventional ORB endsystems.

2.2.1 I/O Subsystem

The I/O subsystems of general-purpose operating systems, such as Solaris and Windows NT, do not perform preemptive, prioritized protocol processing [18]. Therefore, the protocol processing of lower priority packets is *not* deferred due to the arrival of higher priority packets. Instead, incoming packets are processed according to their order of arrival rather than their priority.

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

[18] examines key issues that cause priority inversion in I/O subsystems and describes how TAO's real-time I/O subsystem avoids priority inversion by co-scheduling pools of user-level and kernel-level real-time threads.

2.2.2 ORB Core

A CORBA ORB Core implements the GIOP protocol, which defines a standard format for interoperating between heterogeneous ORBs [5]. The mechanisms in an ORB Core establish connections and implement the concurrency architecture for processing GIOP requests. The following presents the common sources of priority inversion in conventional ORB Core implementations.

Connection architecture: The ORB Core's architecture for managing connections has a substantial impact on real-time ORB behavior. Therefore, a key challenge for real-time ORB developers is determining an ORB connection architecture that can efficiently and predictably utilize the networking mechanisms of an ORB endsystem. The following outlines the key sources of priority inversion and non-determinism exhibited by conventional ORB Core connection architectures:

- **Dynamic connection management:** ORBs that create connections dynamically can incur significant run-time overhead and priority inversion. For instance, a high-priority client might be required to wait for the connection establishment of a lower-priority client. In addition, the time required to establish connections can vary widely, ranging from hundreds of microseconds to milliseconds, depending on endsystem load and network congestion.

  Connection establishment overhead is difficult to bound. For instance, if an ORB needs to dynamically establish connections between the client and server, it is hard for the ORB to provide a reasonable guarantee of the worst-case execution time since this time also includes the (variable) connection establishment time.

  It is generally necessary to pre-allocate connections *a priori* in order for ORB endsystems to support applications with deterministic real-time QoS requirements. Therefore, the remainder of this paper focuses on ORB Core architectures that can pre-allocate connections.

- **Multiplexed connections:** Conventional ORBs typically multiplex requests through a single TCP connection for all object references held in a client process to a corresponding server process, as shown in Figure 3. The goal of connection multiplexing is to minimize the number of connections open to each server. This design is commonly used to build scalable TCP servers. However, connection multiplexing can cause a significant amount of packet-level priority inversion and synchronization overhead, as shown in Sections 4.2.1 and 4.2.2.

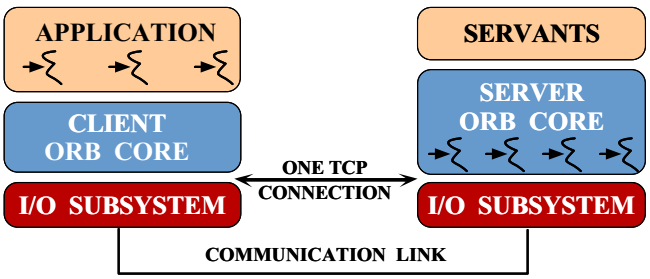

Figure 3: A Multiplexed Connection Architecture

Concurrency architecture: The ORB Core's architecture for concurrency also has a substantial impact on real-time ORB behavior. Therefore, another key challenge for real-time ORB developers is determining a concurrency architecture that can share the aggregate processing capacity of ORB endsystem components and application operations in one or more threads of control. The following outlines the key sources of priority inversion and non-determinism exhibited by conventional ORB Core concurrency architectures:

- **Twoway operation reply processing:** On the client-side, conventional ORB Core concurrency architectures for twoway operations often incur priority inversion. For instance, multi-threaded ORB Cores that use connection multiplexing incur priority inversions when low-priority threads awaiting replies from a server block higher priority threads awaiting replies from the same server.

- **Thread pools:** On the server-side, ORB Core concurrency architectures often use *thread pools* to select a thread to process an incoming request. However, conventional ORBs do not provide APIs to allow real-time applications to determine the priority of threads in this pool. Therefore, the priority of a thread in the pool is often inappropriate for the priority of the servant that ultimately executes the request. This connection architecture increases the potential for priority inversion.

2.2.3 Object Adapter

A standard GIOP-compliant client request contains the identity of its remote object and remote operation. The standard remote object is represented by an object ID `octet sequence` and the remote operation is represented as a string. Conventional ORBs demultiplex client requests to the appropriate operation of the servant implementation using the steps shown in Figure 4.

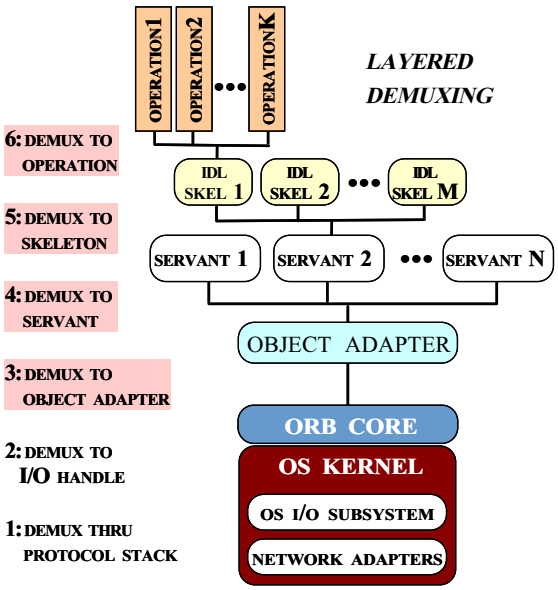

Figure 4: Layered CORBA Request Demultiplexing

• **Steps 1 and 2:** The OS protocol stack demultiplexes the incoming client request multiple times, *e.g.*, through the data link, network, and transport layers up to the user/kernel boundary and the ORB core;

• **Steps 3, 4, and 5:** The ORB core uses the addressing information in the client's object key to locate the appropriate Object Adapter, servant, and the skeleton of the target IDL operation;

• **Step 6:** The IDL skeleton locates the appropriate operation, demarshals the request buffer into operation parameters, and performs the operation upcall.

Demultiplexing client requests through all these layers is expensive, particularly when a large number of operations appear in an IDL interface and/or a large number of servants are managed by an ORB. Hence, layered demultiplexing is inappropriate for high-performance and real-time applications for the following reasons [27]:

Increased performance overhead: Layered demultiplexing reduces performance by increasing the number of internal tables that must be searched while incoming client requests ascend through the processing layers in an ORB endsystem.

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

Conventional implementations of CORBA incur significant demultiplexing overhead. In particular, [21, 28] show that ∼17% of the total server processing time is spent demultiplexing requests. Unless this overhead is reduced and demultiplexing is performed predictably, ORBs cannot provide uniform quality of service guarantees to applications.

[20] presents alternative ORB demultiplexing techniques and describes how TAO's real-time Object Adapter provides optimal demultiplexing strategies that (1) execute deterministically in constant time and (2) avoid priority inversion via de-layered demultiplexing.

3 Alternative ORB Core Concurrency and Connection Architectures

This section describes alternative architectures for ORB Core concurrency and connection architectures. Each of these architectures is used by one or more commercial or research CORBA implementations. Below, we qualitatively evaluate how each architecture manages the aggregate processing capacity of ORB endsystem components and application operations. Section 4 then presents quantitative results that illustrate how efficiently and predictably these alternatives perform in practice.

3.1 Alternative ORB Core Connection Architectures

There are two general strategies for structuring connection architecture in an ORB Core: *multiplexed* and *non-multiplexed*. We describe and evaluate various design alternatives for each approach below.

3.1.1 Multiplexed Connection Architectures

Many ORBs multiplex client requests from a single process through one TCP connection to a corresponding server process. This architecture is commonly used to build scalable

ORBs by minimizing the number of connections open to each server. When multiplexing is used, the key challenge is to design an efficient ORB Core connection architecture that supports concurrent reads and writes.

Multiple threads cannot portably read or write from the same socket concurrently since TCP sockets provide an untyped bytestream data transfer abstraction. Therefore, concurrent write requests to a socket shared within an ORB process must be serialized. Typically, this is implemented by having all client threads in a process acquire a lock before writing to a shared socket.

For oneway calls, there is no need for additional locking or processing once a request is sent. However, twoway calls over a shared connection are more complicated. In this case, the ORB Core must support concurrent read access. For instance, if server replies are multiplexed through a single connection multiple threads cannot read simultaneously from that socket connection. Instead, the ORB Core must use the GIOP sequence number sent with the original client request to demultiplex incoming replies to the appropriate client thread waiting to read the reply.

Several common ways of implementing connection multiplexing to allow concurrent read and write access are described below.

Active connection architecture: One approach is the *active connection* architecture shown in Figure 5.

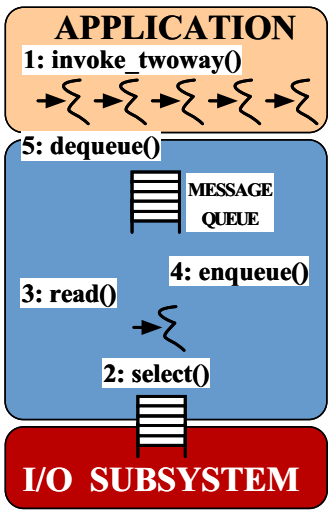

Figure 5: Active Connection Multiplexing Architecture

The active connection architecture performs all reads and writes in a separate thread that is associated with each multiplexed socket. The advantage of this architecture is that it simplifies the ORB connection architecture implementation. In addition, if every socket handles packets of the same priority level, *i.e.*, packets of different priorities are not received on the same socket, the active connection can handle these packets in FIFO order without causing priority inversion.

The disadvantage with this architecture is that the active connection forces an extra context switch on all outgoing/incoming operations. As a result, many ORBs use a variant of this model called the *leader/follower* connection architecture, which is described next.

Leader/followers connection architecture: An alternative to the active connection approach is the *leader/followers* architecture, shown in Figure 6. In this approach, the first thread

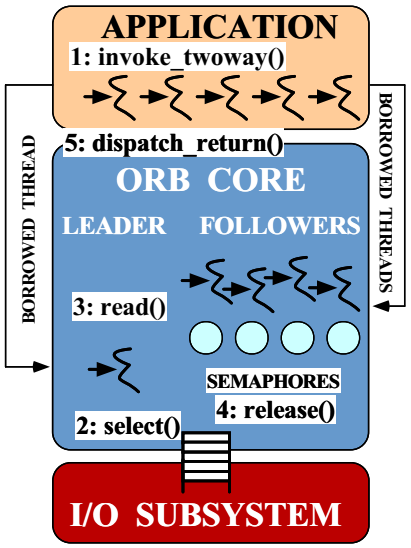

Figure 6: Leader/Follower Connection Multiplexing Architecture

that attempts to read on a shared connection blocks on the `select`[2] call, waiting for a reply. This thread is called the *leader*. To avoid corrupting the socket bytestream, only the leader thread can `select` on the socket(s).

All client threads that "follow the leader" to read replies from the shared socket must block on semaphores managed in FIFO order by the ORB Core. If replies return from the server in FIFO order this strategy is optimal since there is no unnecessary processing or context switching. However, replies may arrive in non-FIFO order. For instance, the next reply arriving from a server could be for any one of the threads blocked on semaphores.

When the next reply arrives from the server, the leader uses the sequence number returned in the GIOP reply header to identify the correct recipient of the reply. If the reply is for the leader's own request, the leader returns to its caller and signals the semaphore of the next follower, which becomes the new leader and blocks on `select`.

[2]The `select` call is typically used since a client may have multiple multiplexed connections to multiple servers.

If the reply is *not* for the leader, however, the leader must signal the semaphore of the appropriate thread. The signaled thread then wakes up, retrieves its reply, and then returns to its caller. Meanwhile, the leader thread continues to `select` for the next reply.

Compared with active connections, the advantage of the leader/follower connection architecture is that it minimizes the number of context switches incurred if replies arrive in FIFO order. The drawback, however, is that the complex implementation logic can lead to significant locking overhead and increased priority inversion. The locking overhead stems from the need to acquire mutexes when sending requests and to block on the semaphores while waiting for replies. The priority inversion occurs if the priority of the waiting threads is not taken into consideration by the leader thread when it demultiplexes replies to client threads.

3.1.2 Non-multiplexed Connection Architectures

One technique for minimizing ORB Core priority inversion is to use a non-multiplexed connection architecture, such as the one shown in Figure 7. In this connection architecture, each

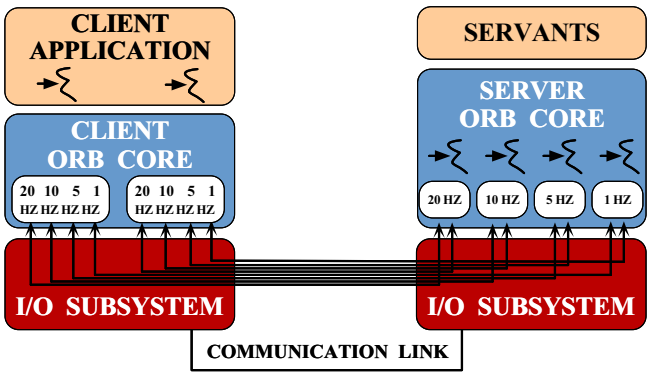

Figure 7: Non-multiplexed Connection Architecture

client thread maintains a table of pre-established connections to servers in thread-specific storage. A separate connection is maintained in each thread for every priority level, *e.g.*, 1 Hz, 5 Hz, 10 Hz, and 20 Hz rate groups in an avionics system. As a result, twoway operations invoked in separate threads do not share socket connections.

The primary benefit of a non-multiplexed connection architecture is that it enables clients to preserve end-to-end priorities while sending requests through ORB endsystems and across communication links. In addition, this design incurs low synchronization overhead since no additional locks are required in the ORB Core when sending/receiving twoway requests.

The drawback with a non-multiplexed connection architecture is that it can uses a larger number of sockets than the multiplexed connection model. Therefore, it is most effective when used for statically configured, fixed priority real-time applications, such as avionics mission computing systems [18].

3.2 Alternative ORB Core Concurrency Architecture

There are several strategies for structuring concurrency in an ORB Core. A common ORB concurrency architecture for real-time ORBs is the *thread pool* concurrency architecture. In this subsection, we describe and evaluate several alternative thread pool architectures.

3.2.1 Leader/Follower Thread Pool Architecture

This ORB concurrency architecture uses a design similar to the connection architecture discussed in Section 3.1.1. The structure is shown in Figure 8. Initially, a pool of threads is

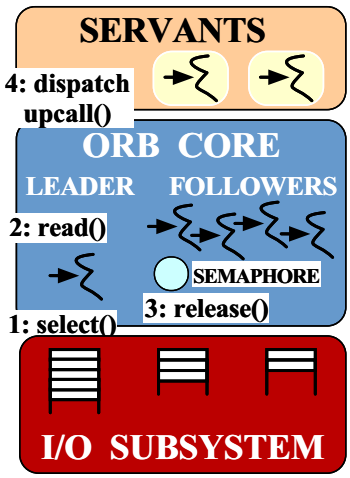

Figure 8: Server-side Leader/Follower Concurrency Architecture

allocated and a leader thread is chosen to `select` on connections for all objects in the server process. When a request arrives, this thread reads it into an internal buffer. If this is a valid request for a servant, the leader thread makes the upcall and a follower thread in the pool is chosen to become the new leader. After the upcall is done, the original leader thread becomes a follower and returns to thread pool. New requests must wait until a thread in the pool is available to execute the request.

The chief advantage of the leader/follower concurrency architecture is that it minimizes context switching overhead incurred by incoming requests. The disadvantages of this model

stem from priority inversion caused due to connection multiplexing, as demonstrated in Section 4.2.2. Since different priority requests share the same communication channel, a high priority request might have to wait until the lower priority request that arrived earlier is processed. Moreover, additional priority inversions can occur if the priority of the thread that dispatches the request is not consistent with the priority of the servant that processes the request.

3.2.2 Threading Framework Architecture

A more flexible way to implement an ORB concurrency architecture is to allow application developers to customize hooks in a general *threading framework*. One way to structure such an approach is shown in Figure 9. The design in this figure is

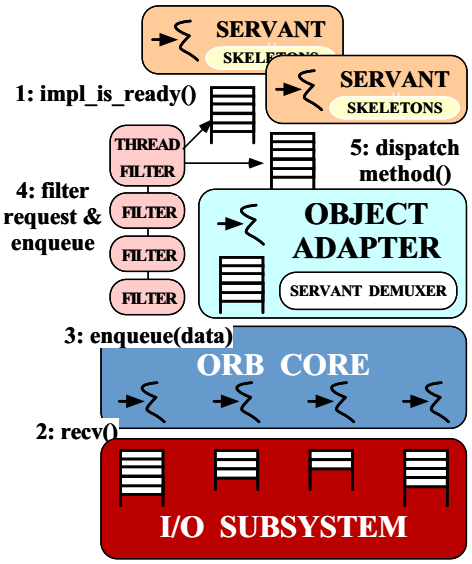

Figure 9: Server-side Thread Pool Framework Concurrency Architecture

based on the MT-Orbix thread filter concurrency framework. In MT-Orbix, an application can install a thread filter at the top of a *chain* of filters that are called successively by the ORB core. This architecture is a variant of the Chain of Responsibility pattern [16].

As shown in the figure, filters are called by the ORB after it has read a CORBA request. The thread filter determines which thread should handle this request, *i.e.*, it implements the desired concurrency policy. For instance, to implement a *thread-per-request* concurrency policy, the filter can spawn a new thread and pass the request to this new thread.[3] To implement a *thread-pool* concurrency policy, the thread filter can enqueue the request into a queue serviced by a thread with the appropriate priority. The new thread is then passes control back to the ORB, which uses this thread to perform the operation demultiplexing and subsequent upcall.

The advantages of a threading framework lie in its flexibility. The thread filter mechanism can be used to implement a variety of concurrency mechanisms. However, since there is only a single chain of filters, extensive priority inversion occurs since each request must traverse the filter chain in FIFO order. Therefore, a high priority request might be processed after several lower priority requests that happened to arrive earlier. In addition, the filter mechanism increases locking overhead since the thread filter needs to acquire locks to enqueue requests into the queue of the appropriate thread.

3.2.3 Thread-per-Rate Thread Pool Architecture

In this approach, the server associates each servant with a thread using the thread-per-rate concurrency architecture shown in Figure 10. The ORB Core can be configured to pre-

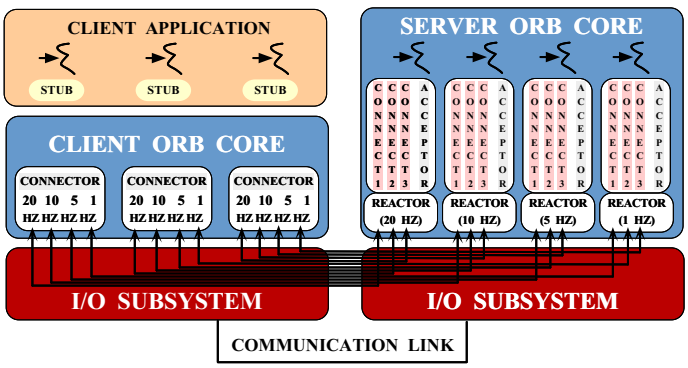

Figure 10: End-to-End Thread-per-Rate Concurrency Architecture

allocate a real-time thread for each *rate group*. A rate group assembles all periodic processing operations that occur at a particular rate. For instance, avionic mission computing systems commonly execute their tasks in rates groups such as 20 Hz, 10 Hz, 5 Hz, and 1 Hz.

Each thread in ORB Core can be configured with a `Reactor` [29] that demultiplexes all incoming client requests for its rate group. A `Reactor` demultiplexes client requests to GIOP connection handlers. Also shown in Figure 10 is how each rate group's `Reactor` in a server is associated with an `Acceptor` [30]. The `Acceptor` is a factory that listens on a particular port number for clients to connect to that rate group. There is typically a listener port per rate, *e.g.*, ports 10020, 10010, 10005, 10001 correspond to the 20 Hz, 10 Hz, 5 Hz, and 1 Hz rates groups, respectively.

[3] The thread-per-request architecture is generally unsuited for real-time applications since the overhead of creating a thread for each request is excessive and non-deterministic.

The thread-per-rate concurrency model can be integrated seamlessly with the non-multiplexed connection model described in Section 3.1.2 to provide end-to-end priority preservation in real-time ORB endsystems. For instance, once a client connects, the `Acceptor` creates a new socket queue and GIOP connection handler to service that queue. The I/O subsystem uses the port number contained in arriving requests as a demultiplexing key to associate requests with the appropriate socket queue. This design minimizes priority inversion through the ORB endsystem by eagerly demultiplexing incoming requests onto the appropriate real-time thread that services the rate group of the target servant.

The design in Figure 10 uses a *Reactor, per thread, per rate group* design to minimize the number of threads in the ORB. The primary benefit of this design is its extremely low overhead for context switching and synchronization. In particular, ORB and application component state need only be locked when objects in different rate groups interact. In addition, this ORB endsystem concurrency model supports scheduling and analysis techniques that associate priority with rate, such as Rate Monotonic Scheduling (RMS) and Rate Monotonic Analysis (RMA) [31, 32].

4 Real-time ORB Core Performance Experiments

This section describes the results of experiments that measure the real-time behavior of several commercial and research ORBs: MT-Orbix 2.2, Sun COOL 4.1, and TAO 1.0. MT-Orbix is a widely used non-real-time ORB, *i.e.*, it was not designed to support applications with real-time requirements. Sun COOL was specifically designed for embedded systems with small memory footprints. TAO was designed to support real-time applications with deterministic and statistical quality of service requirements, as well as best effort requirements.

4.1 Benchmarking Testbed

This section describes the design of an experimental testbed that systematically measures sources of latency and throughput overhead, priority inversion, and non-determinism in ORB endsystems. The architecture of our ORB endsystem testbed is depicted in Figure 11. The hardware and software components in the experiments are described briefly below.

4.1.1 Hardware Configuration

The experimental testbed is depicted in Figure 11. The experiments was conducted using a Bay Networks LattisCell 10114 ATM switch connected to two dual-processor UltraSPARC-2s running SunOS 5.5.1. The LattisCell 10114 is a 16 Port, OC3

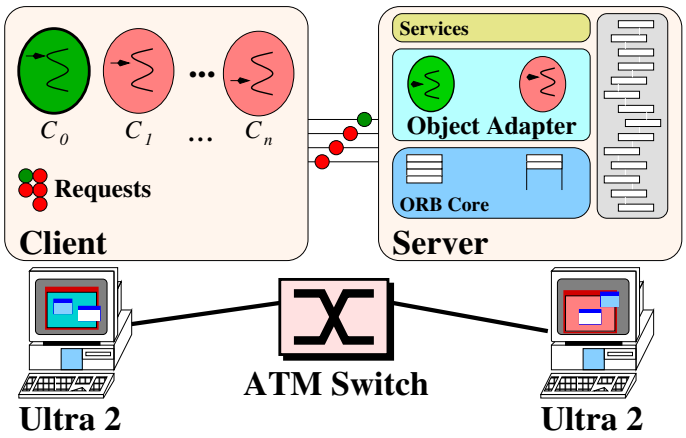

Figure 11: Testbed for ORB Endsystem Evaluation

155 Mbs/port switch. Each UltraSPARC-2 contains 2 168 MHz CPUs[4] with a 1 Megabyte cache per-CPU, 256 Mbytes of RAM, and an ENI-155s-MF ATM adaptor card that supports 155 Megabits per-sec (Mbps) SONET multi-mode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of onboard 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.

4.1.2 Client/Server Configuration and Benchmarking Methodology

Server benchmarking configuration: As shown in Figure 11, our testbed server consists of two servants running in two separate threads. One servant runs in a higher priority thread than the other. Each thread processes requests that are sent to its servant from the remote client threads.

Solaris real-time threads [33] 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:

- **MT-Orbix:** which uses the thread pool architecture based on the Chain of Responsibility pattern described in Section 3.2.2. The server creates two threads at startup time. 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, as shown in Figure 9. Each

[4]To ensure that all the real-time threads were competing for the same CPU, the second CPU was disabled using the Solaris `psradm(1M)` utility.

priority has its own queue of requests, to avoid priority inversion within the queue, which can otherwise occur if a high priority servant and a low-priority servant dequeue requests from the same queue.

• **Sun COOL:** which uses the leader/follower thread pool architecture described in Section 3.2.1. Version 4.1 of Sun COOL does not allow direct application-level concurrency control. Instead, COOL dynamically spawns a pool of threads on behalf of server applications to handle requests, as shown in Figure 8.

• **TAO:** which uses the thread-per-rate concurrency architecture described in Section 3.2.3. Version 1.0 of TAO integrates the thread-per-rate concurrency architecture with the non-multiplexed connection architecture, as shown in Figure 10.

Client benchmarking configuration: Figure 11 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 invoke operations at 20 Hz, *i.e.*, it invokes 20 CORBA twoway calls per second. The low-priority clients run in lower-priority OS threads[5] 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.

The latency metrics were computed by having each client invoke 100 CORBA twoway requests at the prescribed rate. We measured the average latency, *i.e.*, the time taken for the request to complete. The latencies measured by the *n* low-priority clients were then averaged and graphed in Figure 12. In addition, the latency measured by the high-priority client is also shown in the figure.

4.2 Performance Results

Two categories of tests were used in our benchmarking experiments: *blackbox* and *whitebox*. The blackbox tests measured the average twoway response time, as well as the jitter[6] incurred by various clients, as described in Section 4.1.2. Section 4.2.1 explains the blackbox results.

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(1)`, `TNF` [34], and `Quantify` [35]. These tools trace and log the activities of the ORBs and measure the time spent on various tasks, as explained in Section 4.2.2.

[5] All low-priority clients have the same OS thread priority.

[6] Jitter is computed as the standard deviation from the average twoway response time.

The blackbox results indicate the end-to-end performance seen by CORBA applications, as well as the degree of variance in performance as the load changes. The whitebox analysis helps to explain the blackbox results. Together, this information reveals why conventional ORBs, such as MT-Orbix and Sun COOL, are not yet suited for applications with deterministic real-time performance requirements. Likewise, the results illustrate empirically how the ORB Core architecture used by TAO is more suited for these types of applications.

4.2.1 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 show no variance in the latency observed by the high-priority client, irrespective of the number of low-priority clients. However, our measurements of end-to-end twoway ORB latency yielded the results in Figure 12. As shown in this figure, MT-Orbix and Sun

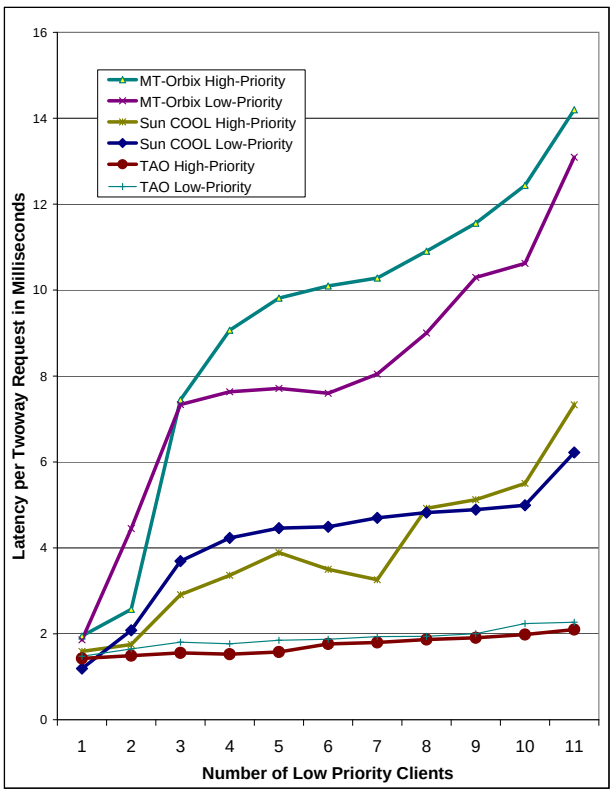

Figure 12: Comparative Latency for MT-Orbix, Sun COOL, and TAO

COOL incurred significantly higher latencies as the number of low-priority clients increased. In addition, MT-Orbix and Sun COOL exhibited very high levels of jitter, as shown in Figure 13. Large amounts of jitter are undesirable for deter-

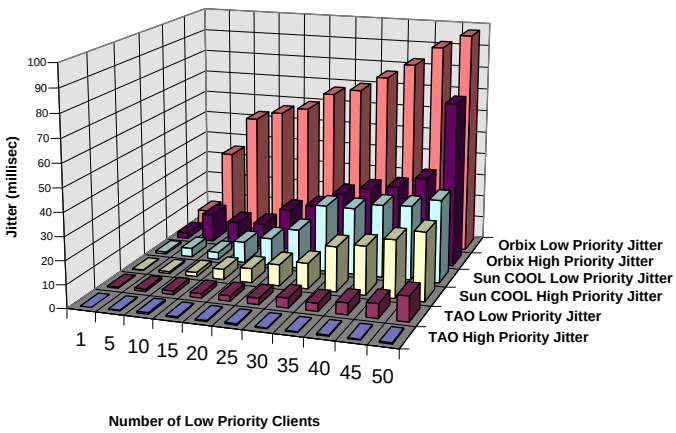

Figure 13: Comparative Jitter for MT-Orbix, Sun COOL, and TAO

ministic real-time applications since the twoway latency observed by individual client requests will vary widely from the average latency.

The blackbox results are explained below for each ORB.

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. Clearly, this behavior is not conducive to deterministic real-time applications. Section 4.2.2 reveals how these inversions stem largely from the MT-Orbix concurrency architecture on the server. In addition, the MT-Orbix ORB also produces high jitter, as shown in Figure 13. This behavior is caused by priority inversions in the ORB, as explained in Section 4.2.2.

Sun COOL results: When a small number of low-priority clients exists, Sun's round-trip latency is lower than the other ORBs (1.2 μsec vs. 1.4 μsec for TAO and 2.0 μsec for MT-Orbix). These results are due to the efficient marshaling performed by COOL's IDL compiler. As the number of low-priority clients increase, however, the latency observed by the high-priority client and the low-priority client increase gradually until $\sim$35 clients, at which point the priority inverts. Section 4.2.2 reveals how these inversions stem largely from the connection architecture used by the COOL client and server.

The jitter observed by applications using the Sun COOL ORB is also fairly high, as shown in Figure 13. This jitter is not as high as that observed with the MT-Orbix ORB, however, since COOL's concurrency architecture does not incur as much locking overhead or use as many FIFO queues.

TAO results: Figure 12 reveals that as the number of low-priority clients increases from 1 to 50, the latency observed by TAO's high-priority client grows by $\sim$0.5 milliseconds. However, TAO's rate of increase is significantly lower than both MT-Orbix and Sun COOL. In particular, when there are 50 low-priority clients competing for the CPU and network bandwidth, the latency observed with MT-Orbix is more than 4 times that of TAO and the Sun COOL latency is more than 3 times that of TAO.

TAO's high-priority client always performs better than its lower priority clients. This indicates that TAO's ORB Core is better suited for maintaining real-time priorities end-to-end. The key difference is that TAO's GIOP protocol processing is performed on a dedicated connection by a dedicated real-time thread with a suitable real-time priority. Thus, TAO shares the minimal amount of resources end-to-end, which substantially reduces opportunities for priority inversion and overhead.

The TAO ORB produces very low jitter (less than 15 milliseconds) for the low-priority requests and negligible jitter (less than 1 millisecond) for the high-priority requests. The stability of TAO's latency is clearly desirable for applications that require predictable end-to-end performance.

4.2.2 Whitebox Results

For the whitebox tests, we used a configuration of ten concurrent clients similar to the one described in Section 4.1. 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. Using Quantify and `truss`, we measured the time consumed by the ORBs performing tasks like synchronization, I/O, and protocol processing. These whitebox results are presented below.

Sun COOL whitebox results: Our whitebox analysis reveals that synchronization overhead from mutex and semaphore operations consume a large fraction of the total COOL ORB processing time. Synchronization overhead arises from mutex and semaphore locking operations that implement the connection and concurrency architecture used by Sun COOL.

Locking overhead accounted for $\sim$50% on the client-side (shown in Figure 14) and more than 40% on the server-side (shown in Figure 15).

For each CORBA request/response, COOL's client ORB performs $\sim$110 lock operations, whereas the server performs $\sim$175 lock operations. This locking overhead stems from excessive dynamic memory allocation, as described in Section 4.3. Each dynamic allocation causes two user-level lock operations, *i.e.*, one acquire and one release.

The COOL connection and concurrency architectures are briefly outlined below.

- **COOL Connection architecture:** The Sun COOL ORB connection architecture uses a variant of the leader/followers architecture described in Section 3.1.1. To summarize, this architecture allows the first thread to

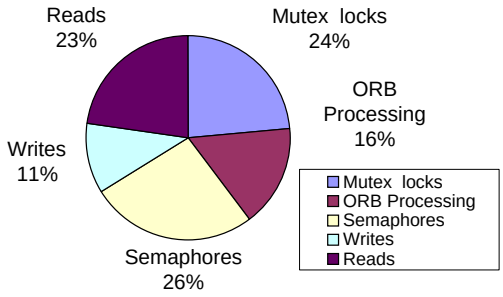

Figure 14: Client-side Whitebox Results for Sun COOL

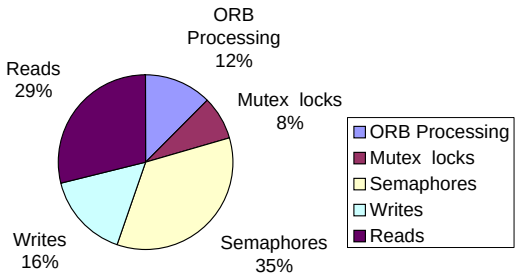

Figure 15: Server-side Whitebox Results for Sun COOL

perform the read on the shared socket, *i.e.*, the leader blocks in `read`. 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 `read`, thereby becoming the new leader.

Thus, COOL multiplexes multiple object references in one client process to a server process through a single connection. This leader/follower connection architecture minimizes the number of simultaneous connections. However, COOL's connection architecture also increases overhead and potential for priority inversion. These problems arise since connection multiplexing requires multiple threads to read/write to a single socket connection shared by the threads.

• **COOL Concurrency architecture:** The Sun COOL ORB concurrency architecture uses the leader/followers thread pool architecture described in Section 3.2.1. To summarize, this architecture initially uses a single thread to wait for connections. 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) its services the request.

The COOL connection architecture and the server concurrency architecture work well to reduce the number of simultaneous open connections and context switching. However, these architectures also lead to heavy priority inversion. For instance, on the client-side, threads of different priorities can share the same channel. Therefore, a high-priority thread might need to wait until a lower priority thread finishes sending it's request. In addition, the priority of the thread that blocks on the semaphore to access a connection may not reflect the priority of the *request* that arrives from the server, which may cause additional priority inversion.

MT-Orbix whitebox results: Figure 17 shows the whitebox results for the server-side of MT-Orbix and Figure 16 shows the whitebox results for the client-side.

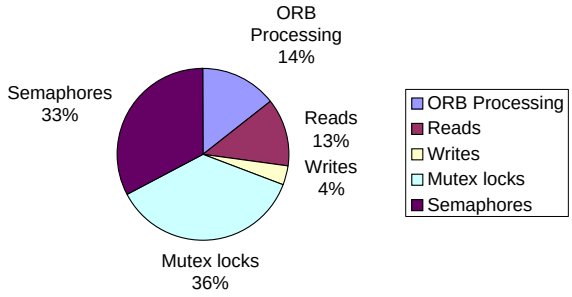

Figure 16: Client-side Whitebox Results for MT-Orbix

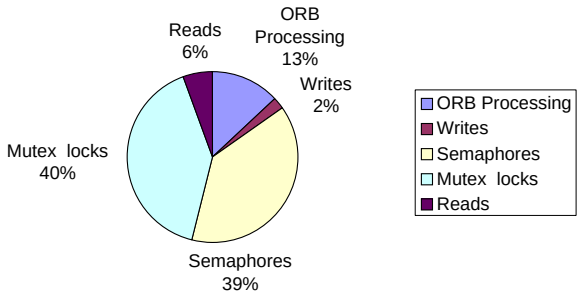

Figure 17: Server-side Whitebox Results for MT-Orbix

• **MT-Orbix Connection architecture:** Like Sun COOL, MT-Orbix uses the leader/follower connection architecture, described in Section 3.2.1. Although this model minimizes context switching overhead, it causes intensive priority inversions, as explained in Section 3.2.1.

• **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.*, high-priority servants had a high-priority thread. A thread filter was then installed to look at each request, determine the priority of the request (by examining the 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 the dispatch time.

The thread pool instantiation of the MT-Orbix mechanism described in Section 3.2.2 is very flexible and easy to use. However, it suffers from heavy priority inversion and high synchronization overhead. The MT-Orbix ORB only provides *one* thread filter chain. Therefore, all incoming requests must be sequentially processed by the filter 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 it can be processed.

In addition, a filter can only be called after (1) IIOP processing has completed and (2) the Object Adapter has determined the target object for this request. This ORB processing is serialized since the MT-Orbix protocol engine 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 the ORB Core.

The concurrency architecture is chiefly responsible for the substantial priority inversion exhibited by MT-Orbix, as shown in Figure 12. This figure shows how the latency observed by the high-priority client increases rapidly, from ∼2 milliseconds to ∼14 milliseconds as the number of low-priority clients increase from 1 to 50.

In addition, the MT-Orbix filter mechanism causes an increase in the synchronization overhead. Since 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 ∼150 lock operations per request, while the server-side performs ∼500 lock operations per request, as shown in Figures 20 and 21.

TAO whitebox results: As shown in Figures 18 and 19, TAO exhibits negligible synchronization overhead.

TAO performs ∼21 lock operations per request on the client-side, and ∼100 lock operations per request on the server-side. This low level of synchronization results from the careful design of TAO's ORB Core, which allocates a separate connection for each priority *rate*, as shown in Figure 10. Therefore, TAO's ORB Core minimizes additional user-level locking operations per request.

• **TAO Connection architecture:** TAO uses a non-multiplexed connection architecture, which pre-establishes connections to servants, as described in Section 3.1.2. One

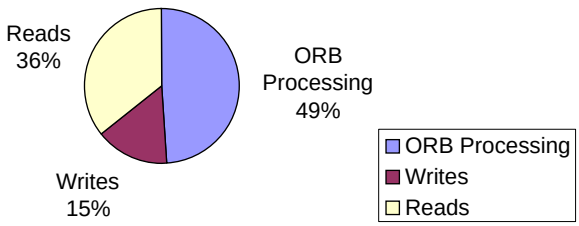

Figure 18: Client-side Whitebox Results for TAO

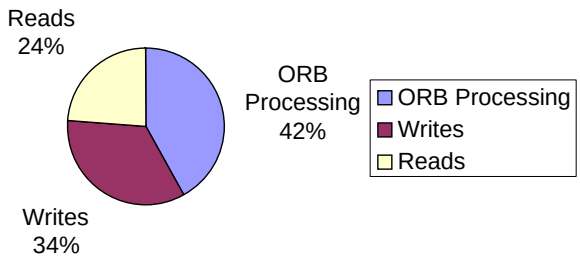

Figure 19: Server-side Whitebox Results for TAO

connection is pre-established per priority level, *i.e.*, rate group, thereby avoiding the non-deterministic delay involved in dynamic connection setup. In addition, different priority levels have their own connection, thus avoiding priority inversion due to the FIFO ordering of packet transmission by the network and I/O subsystem.

• **TAO Concurrency architecture:** TAO supports a variety of concurrency architectures, as described in [18]. The *thread-per-rate* architecture was used for the benchmarks described in this paper. In this concurrency architecture, a separate thread is created for each priority level *i.e.*, rate-group. Thus, the low-priority client issues CORBA requests at a lower rate (10 Hz) than the high-priority client (20 Hz).

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. In addition, the two threads service separate client connections, thereby eliminating the priority inversion that otherwise arises from connection multiplexing, as exhibited by Sun COOL and MT-Orbix.

Locking overhead: Our whitebox tests measured user-level locking overhead (shown in Figure 20) and kernel-level locking overhead (shown in Figure 21) in the MT-Orbix, Sun COOL and TAO ORBs. User-level locks typically protect dy-

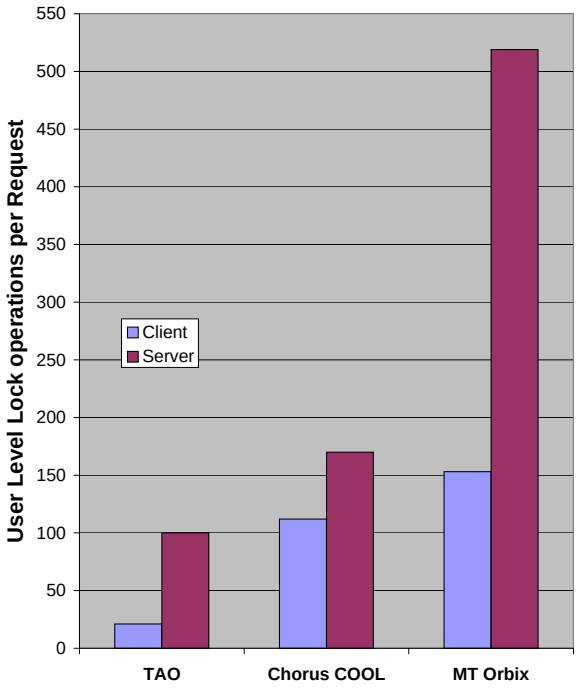

Figure 20: User-level Locking Overhead in ORBs

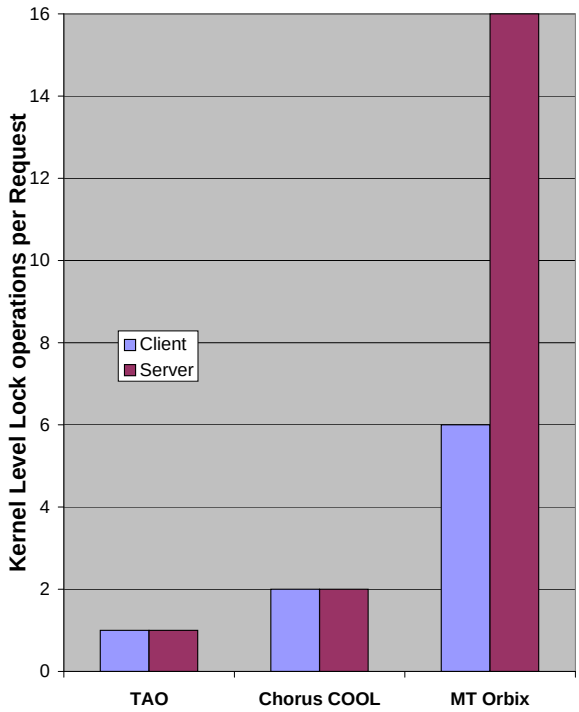

Figure 21: Kernel-level Locking Overhead in ORBs

namic memory allocation since memory is allocated from a global per-process heap.

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 arise from kernel-level lock operations.

TAO limits user-level locking by using pre-allocated buffers. A single buffer is allocated per request. This buffer is subdivided to accommodate the various fields of the request. Kernel-level locking is limited due to the fact that ORB resources are not shared between the threads.

4.3 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 is a significant problem in conventional ORBs. As a result, these ORBs are currently unsuitable for applications with deterministic real-time requirements. Based on our results, and our past experience [21, 20, 28, 22] measuring the performance of CORBA ORB endsystems, we suggest the following recommendations to decrease non-determinism and limit priority inversions in real-time ORB endsystems.

1. An ORB 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 Sun COOL nor MT-Orbix provide APIs for pre-establishing connections, though TAO does provide these APIs as extensions to CORBA.

We recommend that APIs to control the pre-establishment of connections should be defined as an OMG standard.

2. An ORB 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.

We recommend that real-time ORBs should allow application developers to determine whether requests with different priorities are multiplexed over shared connections. Currently, neither COOL or MT-Orbix supports this level of control, though TAO provides this flexibility.

3. Dynamic memory allocation should be minimized in an ORB: Thread-safe implementations of dynamic memory allocators require user-level locking. For instance, the C++

new operator allocates memory from global pool shared by all threads in a process. Likewise, the C++ delete operation that releases allocate memory also requires user-level locking to update the global shared pool. This lock sharing contributes to the overhead shown in Figure 20.

We recommend that real-time ORBs avoid excessive sharing of dynamic memory locks via the use of OS features such as thread-specific storage [36], which allocates memory from heaps that are unique in each thread.

4. The ORB Concurrency architecture should be flexible, yet efficient and predictable: Many ORBs, such as Sun COOL and Expersoft CORBAPlus, creates thread on behalf of server applications. This design prevents application developers from customizing ORB performance by selecting an appropriate concurrency architecture. Conversely, other ORB concurrency architectures are flexible, but inefficient and non-deterministic, as shown in the Section 4.2.2 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 (*e.g.*, thread-per-rate, thread pool, thread-per-connection, thread-per-request, etc.) that are carefully designed using thread-specific storage to minimize unnecessary sharing of ORB resources.

5. The selection of ORB endsystem architectures should be driven by empirical performance benchmarks: Our prior research on pinpointing performance bottlenecks and optimizing middleware like Web servers [37, 38] and CORBA ORBs [21, 20, 28, 22] demonstrates the efficacy of this 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.

5 Related Work

An increasing number of research efforts are focusing on integrating QoS into CORBA. The work presented in this paper is based on the TAO project [10]. This section compares TAO with related work.

Krupp, *et al*, at MITRE Corporation were among the first to elucidate the needs of real-time CORBA systems [39]. They identified key requirements and outlined mechanisms for supporting end-to-end timing constraints [40]. A system consisting of a commercial off-the-shelf RTOS, a CORBA-compliant ORB, and a real-time object-oriented database management system is under development [41]. Similar to the TAO approach, the initial static scheduling approach is rate monotonic, but a strategy for dynamic deadline monotonic scheduling support has been designed [40]. Other dynamic scheduling approaches may be considered in the future.

Wolfe, *et al*, are developing a real-time CORBA system at the US Navy Research and Development Laboratories (NRaD) and the University of Rhode Island (URI) [42]. The system supports expression and enforcement of dynamic end-to-end timing constraints through timed distributed operation invocations (TDMIs) [43]. A TDMI corresponds to TAO's RT_Operation [18] and an RT_Environment structure contains QoS parameters similar to those in TAO's RT_Info [10].

One difference between TAO and the URI approaches is that TDMIs [40] express required timing constraints, *e.g.*, deadlines relative to the current time, whereas TAO's RT_Operations publish their resource, *e.g.*, CPU time, requirements. The difference in approaches may reflect the different time scales, seconds versus milliseconds, respectively, and scheduling requirements, dynamic versus static, of the initial application targets. However, the approaches should be equivalent with respect to system schedulability and analysis.

The QuO project at BBN [44] has defined a model for communicating changes in QoS characteristics between applications, middleware, and the underlying endsystems and network. The QuO model uses the concept of a connection between a client and an object to define QoS characteristics, and treats these characteristics as first-class objects. These objects can then be aggregated to enable the characteristics to be defined at various levels of granularity, *e.g.*, for a single method invocation, for all method invocations on a group of objects, and similar combinations. The model also uses several QoS definition languages (QDLs) that describe the QoS characteristics of various objects, such as expected usage patterns, structural details of objects, and resource availability.

The QuO architecture differs from our work on real-time QoS provision since QuO does not provide hard real-time guarantees of ORB endsystem CPU scheduling. Furthermore, the QuO programming model involves the use of several QDL specifications, in addition to OMG IDL, based on the separation of concerns advocated by Aspect-Oriented Programming (AoP) [45]. We believe that while the AOP paradigm is quite powerful, the proliferation of definition languages may be overly complex for common application use-cases. Therefore, the TAO programming model focuses on the RT_Operation and RT_Info QoS specifiers, which can be expressed in standard OMG IDL.

The Epiq project [46] defines an open real-time CORBA scheme that provides QoS guarantees and runtime scheduling flexibility. Epiq extends TAO's off-line scheduling model to provide on-line scheduling. In addition, Epiq allows clients to

be added and removed dynamically via an admission test at runtime. The Epiq project is work-in-progress and does not yet have empirical results.

The ARMADA project [47] defines a set of communication and middleware services that supports fault-tolerant and end-to-end guarantees for real-time distributed applications. ARMADA provides real-time communication services based on the X-kernel and the Mach microkernel. This infrastructure serves as a foundation for constructing higher-level real-time middleware services. TAO differs from ARMADA in that most of the real-time features in TAO are built using TAO's ORB Core. In addition, TAO implements the OMG's CORBA standard, while also providing the hooks that are necessary to integrate with an underlying real-time I/O subsystem. Thus, the real-time services provided by ARMADA's communication system can be utilized by TAO's ORB Core to support a vertically integrated real-time system.

6 Concluding Remarks

Conventional CORBA ORBs exhibit substantial priority inversion and non-determinism. Therefore, they are not yet suited for distributed, real-time applications with deterministic QoS requirements. Meeting these demands requires that the ORB Core minimize priority inversion. The TAO ORB Core described in this paper minimizes priority inversion by using a rate-based concurrency architecture and rate-based connection architecture that share a minimal amount of resources among ORB core threads.

TAO has been used to develop a real-time ORB endsystem for avionics mission computing applications. These applications manage sensors and operator displays, navigate the aircraft's course, and control weapon release. To meet the scheduling demands of mission computing applications, TAO supports real-time scheduling and dispatching of periodic processing operations, as well as efficient event filtering and correlation mechanisms [13]. The C++ source code for TAO and ACE is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`. This release also contains the real-time ORB benchmarking test suite described in Section 4.1.

Acknowledgments

We gratefully acknowledge Sun and IONA for providing us with their ORB software for the benchmarking testbed.

References

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

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

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

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

[5] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.

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

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

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

[9] D. C. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar, "A High-Performance Endsystem Architecture for Real-time CORBA," *IEEE Communications Magazine*, vol. 14, February 1997.

[10] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications, to appear*, 1998.

[11] 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.

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

[13] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

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

[15] 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.

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

[17] A. Gokhale and D. C. Schmidt, "Design Principles and Optimizations for High-performance ORBs," in 12^{th} OOPSLA Conference, poster session, (Atlanta, Georgia), ACM, October 1997.

[18] 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.

[19] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible and Maintainable ORB Middleware," *Communications of the ACM, to appear*, 1998.

[20] 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.

[21] 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.

[22] 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.

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

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

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

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

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

[28] 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.

[29] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[30] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[31] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, pp. 46–61, January 1973.

[32] 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.

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

[34] SunSoft Inc., *TNFtools:Programming Utilities Guide*. 2550 Garcia Avenue, Mountain View CA 94043, May 1995.

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

[36] 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.

[37] 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.

[38] 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.

[39] B. Thuraisingham, P. Krupp, A. Schafer, and V. Wolfe, "On Real-Time Extensions to the Common Object Request Broker Architecture," in *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications (OOPSLA) Workshop on Experiences with CORBA*, ACM, Oct. 1994.

[40] G.Cooper, L. C. DiPippo, L. Esibov, R. Ginis, R. Johnston, P. Kortman, P. Krupp, J. Mauer, M. Squadrito, B. Thurasignham, S. Wohlever, and V. F. Wolfe, "Real-Time CORBA Development at MITRE, NRaD, Tri-Pacific and URI," in *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.

[41] "Statement of Work for the Extend Sentry Program, CPFF Project, ECSP Replacement Phase II," Feb. 1997. Submitted to OMG in response to RFI ORBOS/96-09-02.

[42] 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.

[43] V. Fay-Wolfe, J. K. Black, B. Thuraisingham, and P. Krupp, "Real-time Method Invocations in Distributed Environments," Tech. Rep. 95-244, University of Rhode Island, Department of Computer Science and Statistics, 1995.

[44] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.

[45] G. Kiczales, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[46] W. Feng, U. Syyid, and J.-S. Liu, "Providing for an Open, Real-Time CORBA," in *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.

[47] T. Abdelzaher, S. Dawson, W.-C.Feng, F.Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaikh, K. Shin, Z. Wang, and H. Zou, "ARMADA Middleware Suite," in *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.