Applying Optimization Principle Patterns to Real-time ORBs

Irfan Pyarali, Carlos O'Ryan, Douglas Schmidt, Nanbor Wang, and Vishal Kachroo
{irfan,coryan,schmidt,vishal,nanbor}@cs.wustl.edu
Washington University
Campus Box 1045
St. Louis, MO 63130[†]

Aniruddha Gokhale[*]
gokhale@research.bell-labs.com
Bell Labs, Lucent Technologies
600 Mountain Ave Rm 2A-442
Murray Hill, NJ 07974

This paper will appear in the IEEE Concurrency magazine's Object-Oriented Systems track, edited by Murthy Devarakonda, to appear 2000.

Abstract

First-generation CORBA middleware was reasonably successful at meeting the demands of applications with best-effort quality of service (QoS) requirements. Supporting applications with more stringent QoS requirements poses new challenges for next-generation real-time CORBA middleware, however. This paper provides three contributions to the design and optimization of real-time CORBA middleware. First, we outline the challenges faced by real-time Object Request Broker (ORB) implementers, focusing on requirements for efficient, predictable, and scalable concurrency and demultiplexing in CORBA's ORB Core and Object Adapter components. Second, we describe how TAO, our real-time CORBA implementation, addresses these challenges by applying key ORB optimization principle patterns, which are rules for avoiding common design and implementation problems that can degrade the efficiency, scalability, and predictability of complex systems. Third, we present the results of benchmarks that evaluate the impact of TAO's patterns and design strategies empirically.

Our results indicate that it is possible to develop highly configurable, adaptable, and standard-compliant ORBs that can meet the QoS requirements of many real-time applications. A key contribution of our work is to demonstrate that the ability of CORBA ORBs to support real-time systems is largely an implementation detail. In particular, relatively few changes are required to the standard CORBA reference model and programming API to support real-time applications.

[*]Work done by the author while at Washington University.

[†]This work was supported in part by Boeing, NSF grant NCR-9628218, DARPA contract 9701516, Motorola, Siemens ZT, and Sprint.

1 Introduction

1.1 Overview of CORBA

CORBA Object Request Brokers (ORBs) allow clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware [1]. Figure 1 illustrates the key components in the CORBA reference model [2] that collaborate to provide this degree of portability, interoperability, and transparency.[1] Each component in the

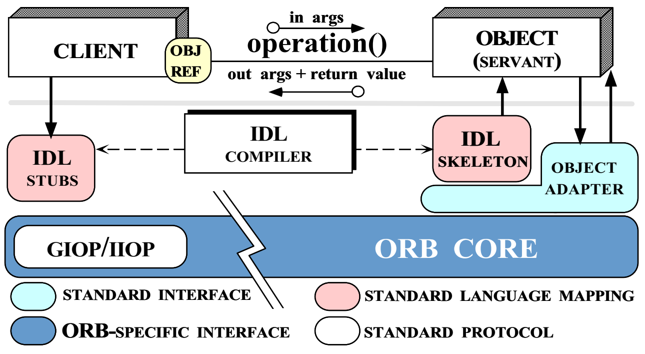

Figure 1: Key Components in the CORBA 2.x Reference Model

CORBA reference model is outlined below:

Client: A client is a *role* that obtains references to objects and invokes operations on them to perform application tasks. Objects can be remote or collocated relative to the client. Ideally, a client can access a remote object just like a local object, *i.e.*, `object`$\rightarrow$`operation(args)`. Figure 1 shows how the underlying ORB components described below transmit remote operation requests transparently from client to object.

[1]This overview only focuses on the CORBA components relevant to this paper. For a complete synopsis of CORBA's components see [2].

Object: In CORBA, an object is an instance of an OMG Interface Definition Language (IDL) interface. Each object is identified by an *object reference*, which associates one or more paths through which a client can access an object on a server. An *object ID* associates an object with its implementation, called a servant, and is unique within the scope of an Object Adapter. Over its lifetime, an object has one or more servants associated with it that implement its interface.

Servant: This component implements the operations defined by an OMG IDL interface. In object-oriented (OO) languages, such as C++ and Java, servants are implemented using one or more class instances. In non-OO languages, such as C, servants are typically implemented using functions and `struct`s. A client never interacts with servants directly, but always through objects identified by object references.

ORB Core: When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response, if any, to the client. An ORB Core is implemented as a run-time library linked into client and server applications. For objects executing remotely, a CORBA-compliant ORB Core communicates via a version of the General Inter-ORB Protocol (GIOP), such as the Internet Inter-ORB Protocol (IIOP) that runs atop the TCP transport protocol. In addition, custom Environment-Specific Inter-ORB protocols (ESIOPs) can also be defined.

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as a "glue" between the client and servants, respectively, and the ORB. Stubs implement the *Proxy* pattern [3] and provide a strongly-typed, *static invocation interface* (SII) that marshals application parameters into a common message-level representation. Conversely, skeletons implement the *Adapter* pattern [3] and demarshal the message-level representation back into typed parameters that are meaningful to an application.

IDL Compiler: An IDL compiler transforms OMG IDL definitions into stubs and skeletons that are generated automatically in an application programming language, such as C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [4].

Object Adapter: An Object Adapter is a composite component that associates servants with objects, creates object references, demultiplexes incoming requests to servants, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on a servant. Object Adapters enable ORBs to support various types of servants that possess similar requirements. This design results in a smaller and simpler ORB that can support a wide range of object granularities, lifetimes, policies, implementation styles, and other properties.

1.2 Challenges for Real-time CORBA

As described above, CORBA helps to improve the flexibility, extensibility, maintainability, and reusability of distributed applications [1]. A growing class of distributed real-time applications require ORB middleware that provides stringent quality of service (QoS) support, such as end-to-end priority preservation, hard upper bounds on latency and jitter, and bandwidth guarantees [5]. Figure 2 depicts the layers and components of an ORB endsystem that must be carefully designed and systematically optimized to support end-to-end application QoS requirements.

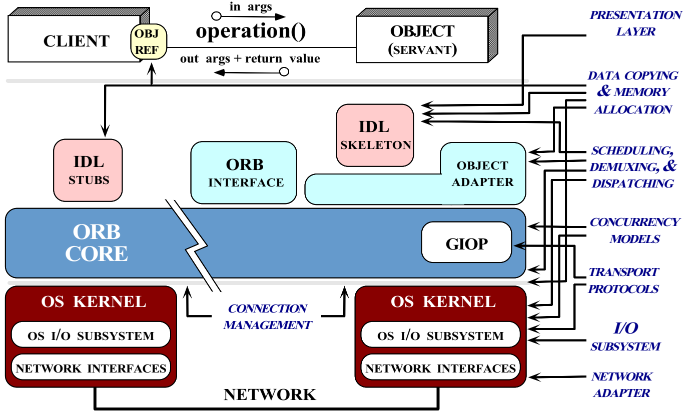

Figure 2: Real-time Features and Optimizations Necessary to Meet End-to-end QoS Requirements in ORB Endsystems

The first-generation of ORBs lacked many of the features and optimizations [6] shown in Figure 2. This situation was not surprising, of course, since ORB developers focused initially on refining the OMG specifications and developing core infrastructure components, such as the basic ORB communication mechanisms. In contrast, second-generation ORBs, such as The ACE ORB (TAO) [7], have leveraged the maturations of standards [5, 3], patterns [8], and QoS-enabled framework components [6], to provide end-to-end QoS guarantees to applications both *vertically* (*i.e.*, network interface $\leftrightarrow$ application layer) and *horizontally* (*i.e.*, end-to-end) by integrating highly optimized CORBA middleware with OS I/O subsystems, communication protocols, and network interfaces.

Our previous research has examined many dimensions of high-performance and real-time ORB endsystem design, including static and dynamic scheduling, event processing, I/O subsystem integration, ORB Core architectures [6], systematic benchmarking of multiple ORBs, and design patterns for ORB extensibility [8]. This paper focuses on other previously unexplored dimensions in the high-performance and real-time ORB endsystem design space: *Object Adapter and ORB Core optimizations for (1) server-side concurrency, (2) memory man-*

agement, and (3) CORBA request demultiplexing.

The optimizations used in TAO are guided by a set of *principle patterns* [9] that we have applied in prior work to optimize middleware [7] and lower-level networking software, such as TCP/IP. Optimization principle patterns document rules for avoiding common design and implementation mistakes that degrade the performance, scalability, and predictability of complex systems. The optimization principle patterns we applied to TAO are shown in Table 1. We applied these optimization principle patterns in TAO to address the following ORB design and implementation challenges:

	Optimization Principle Pattern
1	Optimizing for the common case
2	Eliminating gratuitous waste
3	Shifting computation in time via precomputing
4	Passing hints between layers and components
5	Not being tied to reference models and implementations
6	Replacing inefficient general-purpose operations with special-purpose ones
7	Leveraging system components by exploiting locality
8	Adding redundant state to minimize computations
9	Using efficient/predictable data structures

Table 1: Optimization Principle Patterns Applied in TAO

Optimizing the server-side ORB concurrency model: The concurrency model used to multi-thread an ORB has a substantial impact on its performance, predictability, and scalability. However, concurrency models supported in conventional ORBs, such as thread-per-request or queue-based worker thread pools, incur excessive context switching, synchronization, and data movement overhead [6]. Therefore, TAO employs a leader/followers thread pool model described in Section 2.1. This concurrency model requires no heap memory allocations or locks in the critical path, which is optimal for many types of real-time applications. This optimization is based on the principle patterns of optimizing for the common case, eliminating gratuitous waste, and not being tied to reference implementations.

Optimizing memory management: ORBs allocate buffers to send and receive (de)marshaled data. It is important to optimize these allocations since they are a significant source of dynamic memory management and locking overhead. Section 2.2 describes the mechanisms TAO uses to allocate and manipulate internal parameter (de)marshaling buffers. We illustrate how TAO minimizes fragmentation, data copying, and locking for many common application use-cases. The principle patterns of exploiting locality and optimizing for the common case influence these optimizations.

Optimizing CORBA request demultiplexing: The time an ORB's Object Adapter spends demultiplexing requests to target object implementations, *i.e.*, servants, can constitute a significant source of ORB overhead for real-time applications [10]. Section 3 describes how Object Adapter demultiplexing strategies impact the scalability and predictability of real-time ORBs. This section also illustrates how TAO's Object Adapter optimizations enable constant time request demultiplexing in the average- and worst-case, *regardless* of the number of objects or operations configured into an ORB. The principle patterns that guide our request demultiplexing optimizations include precomputing, using specialized routines, passing hints in protocol headers, adding extra state, and not being tied to reference models.

The remainder of this paper is organized as follows: Section 2 outlines the ORB Core architecture of CORBA ORBs and evaluates the design and performance of ORB Core optimization principle patterns used in TAO; Section 3 outlines the Portable Object Adapter (POA) architecture of CORBA ORBs and evaluates the design and performance of POA optimization principle patterns used in TAO; and Section 4 provides concluding remarks.

2 Optimizing the ORB Core for Real-time Applications

The ORB Core is a standard component in CORBA that is responsible for connection and memory management, data transfer, endpoint demultiplexing, and concurrency control [2]. An ORB Core is typically implemented as a run-time library linked into both client and server applications. When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response, if any, to the client. For objects that reside remotely, a CORBA-compliant ORB Core transfers requests via the General Inter-ORB Protocol (GIOP), which is commonly implemented with the Internet Inter-ORB Protocol (IIOP) that runs atop TCP.

Optimizing an ORB Core to support real-time applications requires the resolution of many design challenges. This section outlines several of the most important challenges and describes the optimization principle patterns we applied to maximize the efficiency, predictability, and scalability of TAO's ORB Core. These optimizations include minimizing context switching, synchronization, and data movement in TAO's concurrency model, as well as minimizing dynamic memory allocations and data copies. Additional optimizations for ORB Core connection management are described in [6].

2.1 ORB Core Concurrency Model Optimizations

Motivation: A common concurrency model used in conventional ORBs is to use a *queue-based worker thread pool*. As shown in Figure 3, the components in this model include a

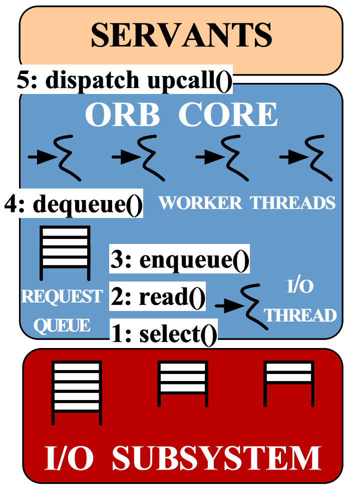

Figure 3: Server Queue-based Worker Thread Pool Concurrency Model

designated I/O thread, a request queue, and a pool of worker threads. The I/O thread `selects` (**1**) on the socket endpoints, (**2**) `reads` new client requests, and (**3**) inserts them into the tail of the request queue. A worker thread in the pool dequeues (**4**) the next request from the head of the queue and (**5**) dispatches it to a user-defined servant operation via an upcall.

The queue-based worker thread pool model is popular for several reasons: (1) it bounds the resources dedicated to threads, (2) it isolates the I/O thread from the concurrency strategy ultimately used to process the request, (3) it is relatively easy to implement, (4) CORBA server applications can control thread creation and control via factory patterns [3], and (5) other concurrency mechanisms, such as thread-per-request or thread pools with lanes [5], can be implemented using this basic model.

However, the queue-based worker thread pool model is inadequate for many types of real-time systems because it (1) *shares dynamically allocated data buffers between threads*, which works against CPU cache affinity and limits the applicability of other optimizations, such as thread-specific storage (TSS) memory management described in Section 2.2, (2) *increases locking overhead* due to the synchronization required to pass data between threads, and (3) can result in *unbounded priority inversions* since a FIFO request queue will queue up all requests at the tail of the queue, irrespective of their priority.

TAO's leader/followers thread pool server concurrency model: To alleviate the drawbacks outlined above, TAO uses the *leader/followers* thread pool model shown in Figure 4. In

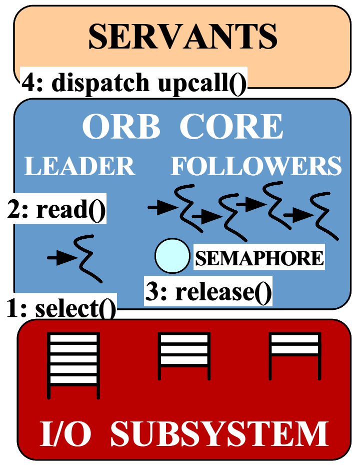

Figure 4: Leader/Followers Thread Pool Server Concurrency Model

this model, there is no designated I/O thread. Instead, a pool of threads is allocated and all threads in the pool take turns playing the role of the I/O thread. The current leader thread in the server process `selects` (**1**) on all open client connections. When a request arrives, the leader thread reads (**2**) it into an internal buffer. Once the request is validated, a follower thread in the pool is released to become the new leader (**3**) and the original leader thread dispatches the upcall (**4**). After the upcall returns, the original leader thread becomes a follower and returns to the thread pool. New requests are queued in socket endpoints until a thread in the pool is available to execute the requests.

Compared with the queue-based worker thread pool, the leader/followers thread pool model (1) *improves CPU cache affinity and eliminates dynamic allocation and data buffer sharing between threads* by reading the request into buffer space allocated on the stack of the leader or by using TSS memory allocations, (2) *minimizes locking overhead* by not exchanging data between threads, thereby reducing thread synchronization, and (3) *minimizes priority inversion* since no extra queueing is introduced by the ORB Core. When combined with real-time I/O subsystems, the leader/follower thread pool model can significantly reduce sources of non-determinism in server ORB request processing.

Empirical results: Figure 5 compares the performance of the leader/follower and queue-based worker thread pool concurrency models. These benchmarks were conducted using TAO version 1.0 on a quad-CPU 400 MHz Pentium II Xeon, with 1 GByte RAM, 512 Kb cache on each CPU, running Debian Linux release 2.2.5, and g++ version egcs-2.91.66.

Our benchmarks measure the total time required by each con-

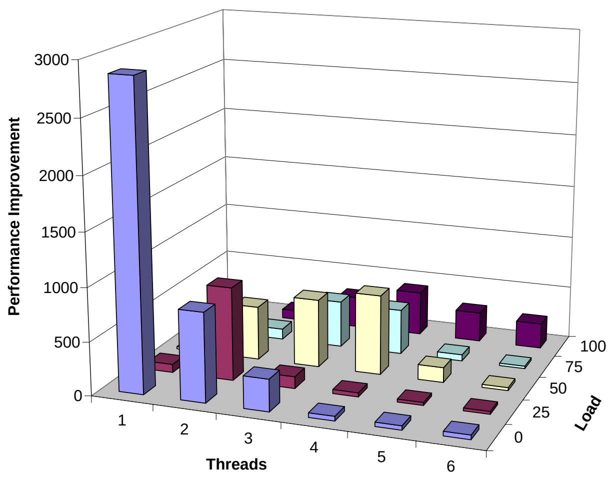

Figure 5: Performance of Leader/Follower vs. Queue-based Worker Thread Pool

currency model to process 100,000 CORBA request messages. We varied the number of threads and the amount of application-level processing performed for each request. The results in Figure 5 illustrate the percentage improvement in performance for the leader/followers thread pool model compared with the queue-based worker thread pool model.

As shown in the figure, the leader/followers concurrency model outperformed the queue-based approach for all combinations of threads and application workload. The largest improvement, ∼2,800%, occurred for a small number of threads and a small amount of work-per-request. As the number of threads and the amount of work-per-request increased the percentage improvement decreased to ∼8%. These results illustrate that the queue-based worker thread pool model incurs a higher amount of overhead for memory allocation, locking, and data movement than the leader/followers model.

Note that on a lightly loaded real-time system, using a small number of threads will generally yield better throughput than a higher number of threads. This difference stems from the higher context switching and locking overhead incurred by threading. As workloads increase, however, addition threads may help improve server throughput, particularly when the server runs on a multi-processor.

2.2 Memory Management Optimizations

Motivation: A key source of overhead and non-determinism in conventional ORB Core implementations stems from improper management of memory buffers. Memory buffers are used by CORBA clients to send requests containing marshaled parameters. Likewise, CORBA servers use memory buffers to receive requests containing marshaled parameters.

One source of memory management overhead is incurred by dynamic memory allocation, which is problematic for real-time ORBs. For instance, dynamic memory can fragment the global heap, which decreases ORB predictability. Likewise, locks used to protect a global heap from simultaneous access by multiple threads can increase synchronization overhead and incur priority inversion [6].

Another significant source of memory management overhead involves excessive data copying. For instance, conventional ORBs often resize their internal marshaling buffers multiple times when encoding large operation parameters. Naive memory management implementations use a single buffer that is resized automatically as necessary, which can cause excessive data copying.

TAO's memory management optimization techniques: TAO's memory management strategies leverage its concurrency strategies, which minimize thread context switching overhead and priority inversions by eliminating queueing within the ORB's critical path. For example, on the client, the thread that invokes a remote operation is the same thread that completes the I/O required to send the request, *i.e.*, no queueing exists within the ORB. Likewise, on the server, the thread that reads a request completes the upcall to user code, also eliminating queueing within the ORB.[2] These optimizations are based on the principle pattern of exploiting locality and optimizing for the common case.

By avoiding thread context switches and unnecessary queueing, TAO can benefit from memory management optimizations based on *thread-specific storage* (TSS). TSS is a common design pattern [8] for optimizing buffer management in multi-threaded middleware. This pattern allows multiple threads to use one logically global access point to retrieve thread-specific data without incurring locking overhead for each access, which is an application of the optimization principle pattern of avoiding waste. TAO uses this pattern to place its memory allocators into TSS. Using a thread-specific memory pool eliminates the need for intra-thread allocator locks, reduces fragmentation in the allocator, and helps minimize priority inversion in real-time applications.

In addition, TAO minimizes unnecessary data copying by keeping a linked list of marshaling buffers. As shown in Figure 6, operation arguments are marshaled into TSS allocated buffers. The buffers are linked together to minimize data copying. Gather-write I/O system calls, such as `writev`, can then write these buffers atomically without requiring multiple OS calls, unnecessary data allocation, or copying. TAO's memory management design also supports special allocators, such

[2]Any queueing required by the ORB endsystem is performed in the OS I/O subsystem.

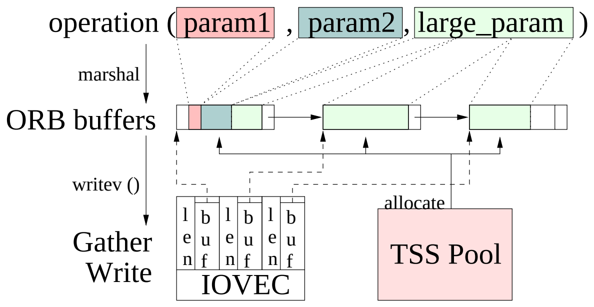

Figure 6: TAO's Internal Memory Management

as zero-copy schemes that share memory pools between user processes, the OS kernel, and network interfaces.

Empirical results: Figure 7 compares buffer allocation time for a CORBA request using thread-specific storage (TSS) allocators with that of using a global heap allocator. These

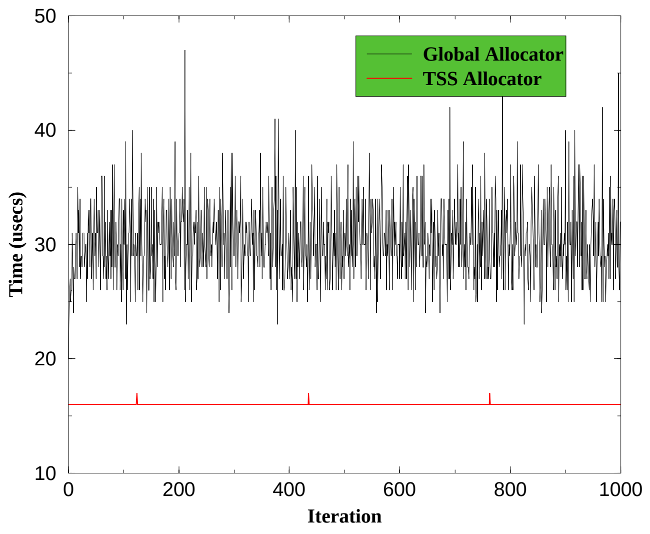

Figure 7: Buffer Allocation Time using TSS and Global Heap Allocators

experiments were executed on a Pentium II/450 with 256Mb of RAM, running LynxOS 3.0, which is a real-time OS. The test program contained a group of ORB buffer (de)allocations intermingled with a pseudo-random sequence of regular (de)allocations. This use-case is typical of middleware frameworks like CORBA, where application code is called from the framework and vice-versa. Both experiments perform the same sequence of memory allocation requests, with one experiment using a TSS allocator for the ORB buffers and the other using a global allocator.

In this experiment, we perform ~16 ORB buffer allocations and ~1,000 regular data allocations. The exact series of allocations is not important, as long as both experiments perform the same number. If there is one series of allocations where the global heap allocator behaves non-deterministically, it is not suitable for hard real-time systems.

Our results in Figure 7 illustrate that TAO's TSS allocators isolate the ORB from variations in global memory allocation strategies.[3] In addition, this experiment shows how TSS allocators are more efficient than global memory allocators since they eliminate locking overhead. In general, reducing locking overhead throughout an ORB is important to support real-time applications with deterministic QoS requirements [6].

3 Optimizing the POA for Real-time Applications

3.1 POA Overview

The OMG CORBA specification [2] standardizes several server-side components in CORBA-compliant ORBs. These components include the Portable Object Adapter (POA), standard interfaces for object implementations (*i.e.*, servants), and refined definitions of skeleton classes for various programming languages, such as Java and C++.

These standard POA features allow application developers to write more flexible and portable CORBA servers. They also make it possible to (1) conserve resources by activating objects on-demand and to (2) generate so-called persistent object references, which remain valid after the originating server process terminates. Server applications can configure these new features portably using *policies* associated with each POA.

CORBA 2.2 allows server developers to create *multiple* Object Adapters, each with its own set of policies. Although this is a powerful and flexible programming model, it can incur significant run-time overhead because it complicates the request demultiplexing path within a server ORB. This is particularly problematic for real-time applications since naive Object Adapter implementations can substantially increase priority inversion and non-determinism [10].

Optimizing a POA to support real-time applications requires the resolution of several design challenges. This section outlines these challenges and describes the optimization principle patterns we applied to maximize the predictability, performance, and scalability of TAO's POA.

3.2 Optimizing POA Demultiplexing

Scalable and predictable POA demultiplexing is important for many applications that have stringent hard real-time timing

[3]There is a very small variation in the TSS allocator performance; but the variation is bounded and thus the strategy is completely predictable.

constraints. Below, we outline the steps involved in demultiplexing a client request through a CORBA server and then qualitatively and quantitatively evaluate alternative demultiplexing strategies.

3.2.1 Overview of CORBA Request Demultiplexing

A standard GIOP-compliant client request contains the identity of its object and operation. An object is identified by an object key, which is an `octet sequence`. An operation is represented as a `string`. As shown in Figure 8, the ORB

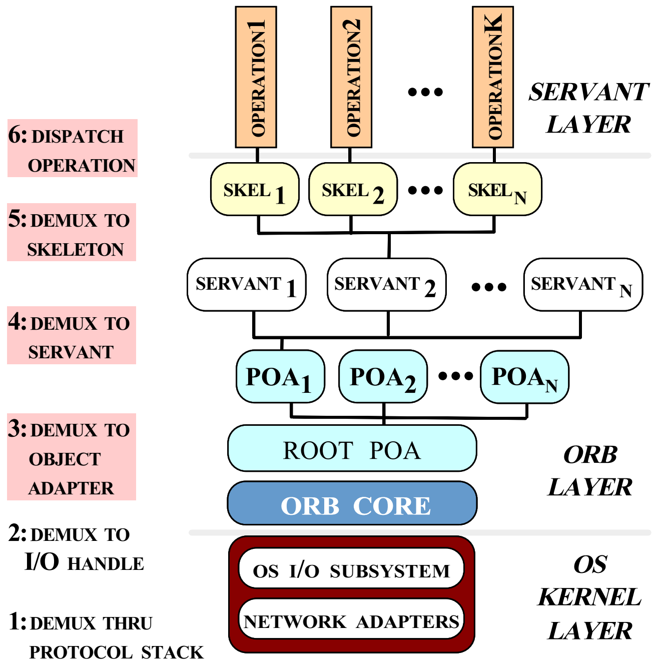

Figure 8: CORBA 2.2 Logical Server Architecture

endsystem must perform the following demultiplexing tasks:

Steps 1 and 2: The OS protocol stack demultiplexes the incoming client request multiple times, starting from the network interface, through the data link, network, and transport layers up to the user/kernel boundary (*e.g.*, the socket layer), where the data is passed to the ORB Core in a server process.

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

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

The conventional deeply-layered ORB endsystem demultiplexing implementation shown in Figure 8 is generally inappropriate for high-performance and real-time applications for the following reasons:

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

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

Conventional implementations of CORBA incur significant demultiplexing overhead. For instance, [10] shows that conventional ORBs spend $\sim$17% of the total server time processing demultiplexing requests. Unless this overhead is reduced and demultiplexing is performed predictably, ORBs cannot provide uniform, scalable QoS guarantees to real-time applications.

The remainder of this section focuses on demultiplexing optimizations performed at the ORB layer, *i.e.*, steps 3 through 6.

3.2.2 Overview of Alternative Demultiplexing Strategies

As illustrated in Figure 8, demultiplexing a request to a servant and dispatching the designated servant operation involves several steps. Below, we qualitatively outline the most common demultiplexing strategies used in CORBA ORBs. Section 3.2.3 then quantitatively evaluates the strategies that are appropriate for each layer in the ORB.

Linear search: This strategy searches through a table sequentially. If the number of elements in the table is small, or the application has no stringent QoS requirements, linear search may be an acceptable demultiplexing strategy. For real-time applications, however, linear search is undesirable since it does not scale up efficiently or predictably to a large number of servants or operations. In this paper, we evaluate linear search only to provide an upper-bound on worst-case performance, though some ORBs still use linear search for operation demultiplexing.

Binary search: Binary search is a more scalable demultiplexing strategy than linear search since its $O(\lg n)$ lookup time is effectively constant for most applications. However, insertions and deletions can be complicated since data must be sorted for the binary search algorithm to work correctly. Therefore, binary search is primarily applicable for ORB operation demultiplexing since all insertions and sorting can be performed off-line by an IDL compiler. In contrast, using binary search to demultiplex requests to servants is more problematic since servants can be inserted or removed dynamically at run-time.

Dynamic hashing: Many ORBs use dynamic hashing as their Object Adapter demultiplexing strategy. Dynamic hashing provides $O(1)$ performance for the average case and supports dynamic insertions more readily than binary search. However, due to the potential for collisions, its worst-case execution time is $O(n)$, which makes it inappropriate for hard real-time applications that require efficient and predictable worst-case ORB behavior. Moreover, depending on the hash algorithm, dynamic hashing may incur a fairly high constant overhead [10].

Perfect hashing: If the set of operations or servants is known *a priori*, dynamic hashing can be improved by precomputing a collision-free *perfect hash function*. Perfect Hashing is based on the optimization principle patterns of precomputing and using specialized routines. A demultiplexing strategy based on perfect hashing executes in constant time and space. This property makes perfect hashing well-suited for deterministic real-time systems that can be configured statically [10], *i.e.*, if the number of objects and operations can be determined off-line.

Active demultiplexing: Although the number and names of operations can be known *a priori* by an IDL compiler, the number and names of servants are generally more dynamic. In such cases, it is possible to use the object ID and POA ID stored in an object key to index directly into a table managed by an Object Adapter. This so-called *active demultiplexing* [10] strategy provides a low-overhead, $O(1)$ lookup technique that can be used throughout an Object Adapter. Active demultiplexing uses the optimization principle pattern of not being tied to reference models and passing hints in headers.

Table 2 summaries the demultiplexing strategies considered in the implementation of TAO's POA.

Strategy	**Search Time**	**Comments**
Linear Search	$O(n)$	Simple to implement Does not scale
Binary Search	$O(\lg n)$	Additions/deletions are expensive
Dynamic Hashing	$O(1)$ average case $O(n)$ worst case	Hashing overhead
Perfect Hashing	$O(1)$ worst case	For static configurations, generate collision-free hashing functions
Active Demuxing	$O(1)$ worst case	For system generated keys, add direct indexing information to keys

Table 2: Summary of POA Demultiplexing Strategies

3.2.3 The Performance of Alternative POA Demultiplexing Strategies

Section 3.2.1 describes the demultiplexing steps a CORBA request goes through before it is dispatched to a user-supplied servant method. These demultiplexing steps include finding the Object Adapter, the servant, and the skeleton code. This section empirically evaluates the strategies that TAO uses for each demultiplexing step. The hardware and software configuration for this experiment is described in Section 2.1.

POA demultiplexing: An ORB Core must locate the POA corresponding to an incoming client request. Figure 8 shows that POAs can be nested arbitrarily. Although nesting provides a useful way to organize policies and namespaces hierarchically, the POA's nesting semantics complicate demultiplexing compared with the original CORBA Basic Object Adapter (BOA) demultiplexing [10] specification.

To support ORB server applications that have deeply nested POA hierarchies, we use active demultiplexing for the POA demultiplexing phase, as follows:

1. All lookups start at the `RootPOA`.
2. The `RootPOA` maintains a `POA table` that points to all the POAs in the hierarchy.
3. Object keys include an index into the `POA table` to identify the POA where the object was activated. TAO's ORB Core uses this index as the active demultiplexing key.
4. In some cases, the POA name also may be needed, *e.g.*, if the POA is activated on-demand. Therefore, the object reference contains both the name and the index.

We conducted an experiment to measure the effect of increasing the POA nesting level on the time required to lookup the appropriate POA in which the servant is registered. We used a range of POA depths, 1 through 25. The results are shown in Figure 9. The experiment was conducted on POAs

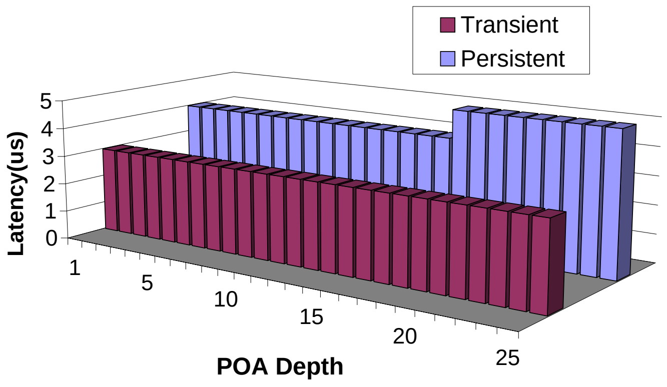

Figure 9: Effect of POA Depth on POA Demultiplexing Latency

whose object references remain valid across different executions of a server (persistent) and those that do not (transient). The results show that using active demultiplexing for POA demultiplexing provides optimal predictability and scalability for both the cases, just as it does when used for servant demultiplexing, as described next.

Servant demultiplexing: Once the ORB Core demultiplexes a client request to the right POA, this POA demultiplexes the request to the correct servant. The following discussion compares the various servant demultiplexing techniques described in Section 3.2.2. TAO uses the Service Configurator [8], Bridge, and Strategy patterns [3] to defer the configuration of the desired servant demultiplexing strategy until ORB initialization, which can be performed either *statically* at compile-time or *dynamically* at run-time. Figure 10 illustrates the class hierarchy of strategies that can be configured into TAO's POAs.

To evaluate the scalability of TAO, our experiments used a range of servants, 1 to 1,000 by increments of 100, in the server. Figure 11 shows the latency for servant demultiplexing as the number of servants increases. This figure illustrates that active demultiplexing is a highly predictable, low-latency servant lookup strategy. In contrast, dynamic hashing incurs higher constant overhead to compute the hash function. Moreover, its performance degrades gradually as the number of servants increases and the number of collisions in the hash table increase. Likewise, linear search does not scale for any realistic system, since its performance degrades rapidly as the number of servants increase.

Note that we did not implement the perfect hashing strategy for servant demultiplexing. Although it is possible to know *a*

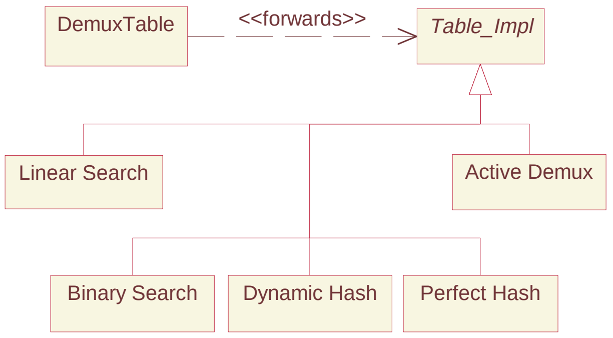

Figure 10: TAO's Class Hierarchy for POA Active Object Map Strategies

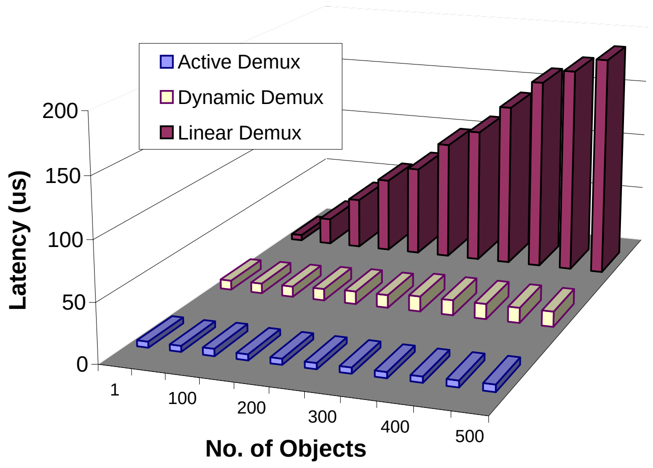

Figure 11: Servant Demultiplexing Latency with Alternative Search Techniques

priori the set of servants in each POA for highly static systems, creating perfect hash functions repeatedly during application development is tedious. We omitted binary search for similar reasons, *i.e.*, it requires maintaining a sorted active object map every time an object is activated or deactivated. Moreover, since the object key is created by a POA, active demultiplexing provides equivalent, or better, performance than perfect hashing or binary search.

Operation demultiplexing: The final step at the Object Adapter layer involves demultiplexing a request to the appropriate skeleton, which demarshals the request and dispatches the designated operation upcall in the servant. To measure operation demultiplexing overhead, our experiments defined a range of operations, 1 through 50, in the IDL interface.

For ORBs like TAO that target real-time embedded systems, operation demultiplexing must be efficient, scalable, and predictable. Therefore, we generate efficient operation lookup using GPERF, which is a freely available perfect hash function generator we developed to automatically construct perfect hash functions from user-supplied keyword lists.

Figure 12 illustrates the interaction between the TAO IDL compiler and GPERF. When perfect hashing, linear search and binary search operation demultiplexing strategies are selected, TAO's IDL compiler invokes GPERF as a co-process to generate an optimized lookup strategy for operation names in IDL interfaces.

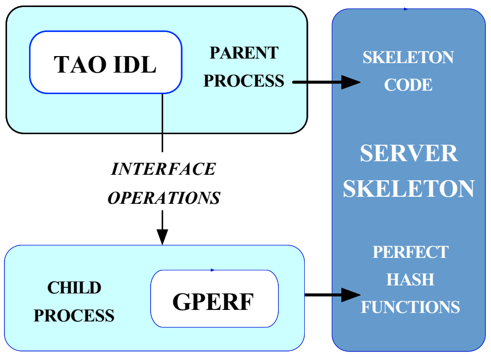

Figure 12: Integrating TAO's IDL Compiler and GPERF

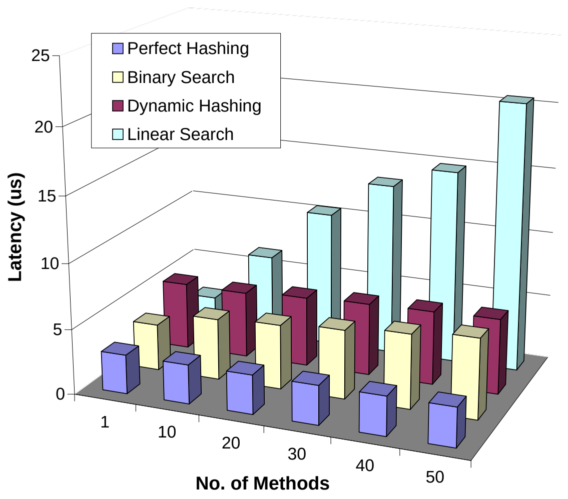

Figure 13: Operation Demultiplexing Latency with Alternative Search Techniques

The lookup key for this phase is the operation name, which is a `string` defined by developers in an IDL file. However, it is not permissible to modify the operation `string` name to include active demultiplexing information. Active demultiplexing cannot be used without modifying the GIOP protocol.[4] Therefore, TAO uses perfect hashing for operation demultiplexing. Perfect hashing is well-suited for this purpose since all operations names are known at compile time.

Figure 13 plots operation demultiplexing latency as a function of the number of operations. This figure illustrates that perfect hashing is extremely predictable and efficient, outperforming dynamic hashing and binary search. As expected, linear search depends on the number and ordering of operations, which is not only inefficient, but also complicates worst-case schedulability analysis for real-time applications.

Optimizing servant-based lookups: When a CORBA request is dispatched by the POA to the servant, the POA uses the object ID in the request header to find the servant in its *active object map*. Section 3.2.3 describes how TAO's lookup strategies provide efficient, predictable, and scalable mechanisms to dispatch requests to servants based on object IDs. In particular, TAO's active demultiplexing strategy enables constant $O(1)$ lookup in the average- and worst-case, regardless of the number of servants in a POA's active object map.

[4]We are investigating modifications to the GIOP protocol for hard real-time systems that possess stringent latency and message-footprint requirements.

However, certain POA operations and policies require lookups on active object map to be based on the *servant pointer* rather than the object ID. For instance, the `_this` method on the servant can be used with the IMPLICIT_ACTIVATION POA policy outside the context of request invocation. This operation allows a servant to be activated implicitly if the servant is not already active. If the servant is already active, it will return the object reference corresponding to the servant.

Unfortunately, naive POA's active object map implementations incur worst-case performance for servant-based lookups. Since the primary key is the object ID, servant-based lookups degenerate into a linear search, even when active demultiplexing is used for the object ID-based lookups. As shown in Figure 11, linear search becomes prohibitively expensive as the number of servants in the active object map increase. This overhead is particularly problematic for real-time applications, such as avionics mission computing systems, that (1) create a large number of objects using `_this` during their initialization phase and (2) must reinitialize rapidly to recover from transient power failures.

To alleviate servant-based lookup bottlenecks, we apply the principle pattern of adding extra state to the POA in the form of a *reverse-lookup* map that associates each servant with its object ID in $O(1)$ average-case time. In TAO, this reverse-lookup map is used in conjunction with the Active Demultiplexing map that associates each object ID to its servant. Figure 14 shows the time required to find a servant, with and without the reverse-lookup map, as the number of servants in a POA increases.

Servants are allocated from arbitrary memory locations. Since we have no control over the pointer value format, TAO

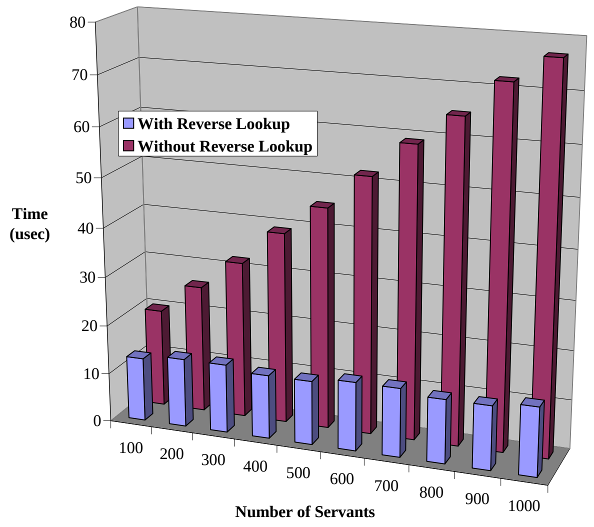

Figure 14: Benefits of Adding a Reverse-Lookup Map to the POA

uses a hash map for the reverse-lookup map. The value of the servant pointer is used as the hash key. Although hash maps do not guarantee $O(1)$ worst-case behavior, they do provide a significant average-case performance improvement over linear search.

A reverse-lookup map can be used only with the UNIQUE_ID POA policy since with the MULTIPLE_ID POA policy, a servant may support many object IDs. This constraint is not a shortcoming since servant-based lookups are only required with the UNIQUE_ID policy. One downside of adding a reverse-lookup map to the POA, however, is the increased overhead of maintaining an additional table in the POA. For every object activation and deactivation, two updates are required in the active object map: (1) to the reverse-lookup map and the (2) to the active demultiplexing map used for object ID lookups. However, this additional processing does not affect the critical path of object ID lookups during run-time.

Summary of TAO's POA demultiplexing strategies: Based on the results of our benchmarks described above, Figure 15 summarizes the demultiplexing strategies that we have determined to be most appropriate for real-time applications. Figure 15 shows the use of active demultiplexing for the POA names, active demultiplexing for the servants, and perfect hashing for the operation names. Table 3 depicts the time in microseconds (μs) spent in each activity as a TAO server processes a request on the quad-CPU 400 MHz Pentium II Xeon used for the benchmarks described in Section 2.1.

All of TAO's optimized demultiplexing strategies described above are entirely compliant with the CORBA specification. Thus, no changes are required to the standard POA interfaces specified in CORBA specification [2].

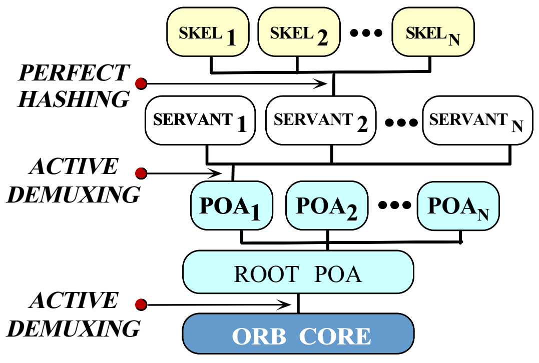

Figure 15: TAO's Default Demultiplexing Strategies

Demultiplexing Stage	Absolute Time (μs)
1. *Parsing object key*	2
2. *POA demux*	2
3. *Servant demux*	3
4. *Operation demux*	3
5. *Parameter demarshal*	operation dependent
6. *User upcall*	servant dependent
7. *Return value marshal*	operation dependent

Table 3: Time Spent in Each Demultiplexing Step

4 Concluding Remarks

Developers of real-time systems are increasingly using off-the-shelf middleware components to lower software lifecycle costs and decrease time-to-market. In contemporary business environments, the flexibility offered by CORBA makes it an attractive middleware architecture. Since CORBA is not tightly coupled to a particular OS or programming language, it can be adapted readily to "niche" markets, such as real-time embedded systems, which are not well covered by other middleware. In this sense, CORBA has an advantage over other middleware, such as DCOM or Java RMI, since it can be integrated into a wider range of platforms and languages.

The POA and ORB Core optimizations and performance results presented in this paper support our contention that the next-generation of standard CORBA ORBs will be well-suited for distributed real-time systems that require efficient, scalable, and predictable performance. Table 4 summarizes which TAO optimizations are associated with which principle patterns, as well as emphasizing that all the optimizations are fully compliant with the standard CORBA specification.

Our primary focus on the TAO project has been to research, develop, and optimize policies and mechanisms that allow

Optimization	Principle Patterns	Compliant
Concurrency	Optimize for common case Avoid gratuitous waste Not tied to reference models	yes
Memory management	Exploit Locality Optimize for common case	yes
Request demuxing	Precompute, Avoid gratuitous waste Passing hints in header Replace general-purpose operations with optimized special-purpose ones Not tied to reference models Adding extra state	yes

Table 4: Degree of CORBA-compliance for Real-time Optimization Principle Patterns

CORBA to support applications with hard real-time requirements. In hard real-time systems, the ORB must meet deterministic QoS requirements to ensure proper overall system functioning. These requirements motivate many of the optimizations and design strategies presented in this paper. However, the architectural design and performance optimizations in TAO's ORB endsystem are equally applicable to many other types of real-time applications, such as telecommunications, network management, and distributed multimedia systems, which have less stringent QoS requirements.

The C++ source code for TAO and ACE is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`. This release also contains the ORB benchmarking test suites described in this paper.

Acknowledgments

We would like to thanks our COOTS shepherd, Steve Vinoski, whose comments helped improve this paper. In addition, we would like to thank the COOTS Program Committee and the IEEE Concurrency anonymous reviewers for their constructive suggestions for improving the paper. In addition, Jeff Parsons provided valuable proof-reading improvements.

References

[1] M. Henning and S. Vinoski, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1999.

[2] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.

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

[4] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, (Las Vegas, NV), ACM, June 1997.

[5] Object Management Group, *Realtime CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999.

[6] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems*, To appear 1999.

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

[8] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, April 1999.

[9] Alistair Cockburn, "Prioritizing Forces in Software Design," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), pp. 319–333, Reading, MA: Addison-Wesley, 1996.

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