Applying Optimization Principle Patterns to Design Real-Time ORBs

Irfan Pyarali, Carlos O'Ryan, Douglas Schmidt, Nanbor Wang, and Vishal Kachroo

Washington University, St. Louis

Aniruddha Gokhale

Lucent Technologies, Bell Labs

Applying Optimization Principle Patterns to Design 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

Abstract

First-generation CORBA middleware was reasonably successful at meeting the demands of request/response 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 ORBs implementers, focusing on optimization principle patterns that can be applied to CORBA's Object Adapter and ORB Core. Second, we describe how TAO, our real-time CORBA implementation, addresses these challenges and applies key ORB optimization principle patterns. Third, we present the results of empirical benchmarks that compare the impact of TAO's design strategies on ORB efficiency, predictability, and scalability.

Our findings indicate that ORBs must be highly configurable and adaptable to meet the QoS requirements for a wide range of real-time applications. In addition, we show how TAO can be configured to perform predictably and scalably, which is essential to support real-time applications. A key result of our work is to demonstrate that the ability of CORBA ORBs to support real-time systems is mostly an implementation detail. Thus, relatively few changes are required to the standard CORBA reference model and programming API to support real-time applications.

1 Introduction

Many companies and research groups are developing distributed applications using middleware components like CORBA Object Request Brokers (ORBs) [1]. CORBA helps to improve the flexibility, extensibility, maintainability, and reusability of distributed applications [2]. However, a growing class of distributed real-time applications also 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 [3]. Figure 1 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.

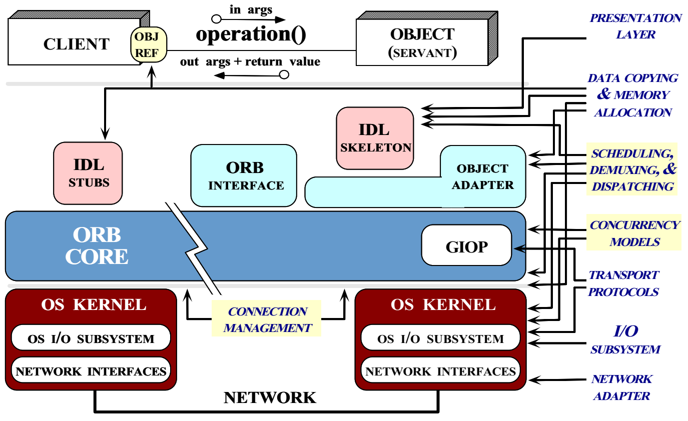


Figure 1: Real-time Features and Optimizations Necessary to Meet End-to-end QoS Requirements in ORB Endsystems

First-generation ORBs lacked many of the features and optimizations [4, 5, 6, 7] shown in Figure 1. This situation was not surprising, of course, since the focus at that time was largely on developing core infrastructure components, such as the ORB and its basic services, defined by the OMG specifications [8]. In contrast, second-generation ORBs, such as The ACE ORB (TAO) [9], explicitly focus on providing end-to-end QoS guarantees to applications *vertically* (*i.e.*, network

[*]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.

interface $\leftrightarrow$ application layer) and *horizontally* (*i.e.*, end-to-end) 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 [9] and dynamic [10] scheduling, event processing [11], I/O subsystem integration [12], ORB Core connection and concurrency architectures [7], systematic benchmarking of multiple ORBs [4], and design patterns for ORB extensibility [13]. This paper focuses on four more dimensions in the high-performance and real-time ORB endsystem design space: *Object Adapter and ORB Core optimizations for (1) request demultiplexing, (2) collocation, (3) memory management, and (4) ORB protocol overhead*.

The optimizations used in TAO are guided by a set of *principle patterns* [14] that have been applied to optimize middleware [15] and lower-level networking software [16], such as TCP/IP. Optimization principle patterns document rules for avoiding common design and implementation problems that degrade the performance, scalability, and predictability of complex systems. The optimization principle patterns we applied to TAO include: *optimizing for the common case; eliminating gratuitous waste; shifting computation in time such as precomputing; avoiding unnecessary generality; passing hints between layers; not being tied to reference implementations; using specialized routines; leveraging system components by exploiting locality; adding state; and using efficient data structures*. Below, we outline how these optimization principle patterns address the following TAO Object Adapter and ORB Core design and implementation challenges.

Optimizing 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. Section 2 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*, and *not being tied to reference models*.

Optimizing collocation: The principle pattern of relaxing system requirements enables TAO to minimize the run-time overhead for *collocated* objects, *i.e.*, objects that reside in the same address space as their client(s). Operations on collocated objects are invoked on servants directly in the context of the calling thread, thereby transforming operation invocations into local virtual method calls. Section 3.1 describes how TAO's collocation optimizations are completely transparent to clients, *i.e.*, collocated objects can be used as regular CORBA objects, with TAO handling all aspects of collocation.

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 3.2 describes the mechanisms used in TAO to allocate and manipulate the internal buffers it uses for parameter (de)marshaling. We illustrate how TAO minimizes fragmentation, data copying, and locking for most application use-cases. The principle patterns of *exploiting locality* and *optimizing for the common case* influence these optimizations.

Minimizing ORB protocol overhead: Real-time systems have traditionally been developed using proprietary protocols that are hard-coded for each application or application family. In theory, the standard CORBA GIOP/IIOP protocols obviate the need for proprietary protocols. In practice, however, many developers of real-time applications are justifiably concerned that standard CORBA protocols incur excessive overhead. Section 3.3 shows how TAO can be configured to reduce the overhead of GIOP/IIOP without affecting the standard CORBA programming APIs exposed to application developers. This optimization is based on the principle pattern of *avoiding unnecessary generality*.

The remainder of this paper is organized as follows: Section 2 outlines the Portable Object Adapter (POA) architecture of CORBA ORBs and evaluates the design and performance of POA optimizations used in TAO; Section 3 outlines the ORB Core architecture of CORBA ORBs and evaluates the design and performance of ORB Core optimizations used in TAO; Section 4 describes related work; and Section 5 provides concluding remarks.

2 Optimizing the POA for Real-time Applications

2.1 POA Overview

The OMG CORBA 2.2 specification [1] standardizes several components on the server-side of 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++ [2].

These standard POA features allow application developers to write more flexible and portable CORBA servers [17]. They also make it possible to conserve resources by activating objects on-demand [18] and to generate "persistent" object references [19] that 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 increase priority inversion and non-determinism [6].

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. These POA optimizations include constant-time demultiplexing strategies, reducing run-time object key processing overhead during upcalls, and generally optimizing POA predictability and reducing memory footprint by selectively omitting non-deterministic POA features.

2.2 Optimizing POA Demultiplexing

Scalable and predictable POA demultiplexing is important for many applications such as real-time stock quote systems [20] that service a large number of clients, and avionics mission systems [11] that have stringent hard real-time timing constraints. Below, we outline the steps involved in demultiplexing a client request through the server-side of a CORBA ORB and then qualitatively and quantitatively evaluate alternative demultiplexing strategies.

2.2.1 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 2, the ORB 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.

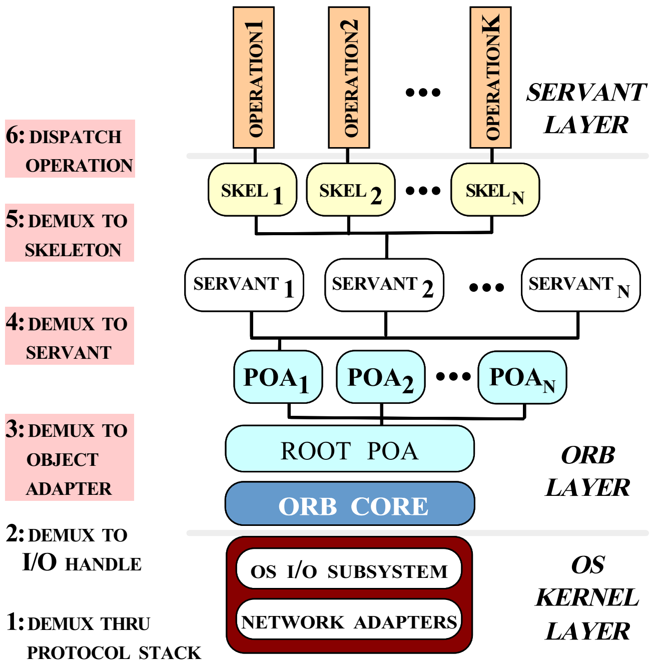


Figure 2: CORBA 2.2 Logical Server Architecture

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 2 is generally inappropriate for high-performance and real-time applications for the following reasons [21]:

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 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 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 [12].

Conventional implementations of CORBA incur significant demultiplexing overhead. For instance, [4, 6] show that conventional ORBs spend ∼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. Information on OS kernel layer demultiplexing optimizations for real-time ORB endsystems is available in [22, 12].

2.2.2 Overview of Alternative Demultiplexing Strategies

As illustrated in Figure 2, 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 2.2.3 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 [4] 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 particularly useful 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 often has a fairly high constant overhead [6].

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* [23]. Perfect Hashing is based on the principle pattern 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 [6], *i.e.*, 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. Active demultiplexing uses the principle pattern of *relaxing system requirements*, *not being tied to reference models*, and *passing hints in headers*. This so-called *active demultiplexing* [6] strategy provides a low-overhead, $O(1)$ lookup technique that can be used throughout an Object Adapter.

Table 1 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 1: Summary of Alternate POA Demultiplexing Strategies

2.2.3 The Performance of Alternative POA Demultiplexing Strategies

Section 2.2.1 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. All POA demultiplexing measurements were conducted on an UltraSPARC-II with two 300 MHz CPUs, a 512 Mbyte RAM, running SunOS 5.5.1, and C++ Workshop Compilers version 4.2.

POA lookup: An ORB Core must locate the POA corresponding to an incoming client request. Figure 2 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 [6] specification.

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 3.

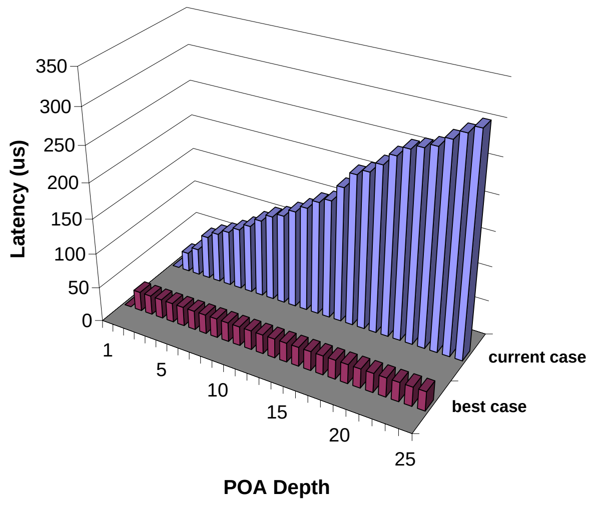


Figure 3: Effect of POA Depth on POA Demultiplexing Latency

Since most ORB server applications do not have deeply nested POA hierarchies, TAO currently uses a POA demultiplexing strategy where each POA finds its child using dynamic hashing and delegates to the child POA where this process is repeated until the search is complete. This POA demultiplexing strategy results in $O(n)$ growth for the lookup time and does not scale up to deeply nested POAs. Therefore, we are adding active demultiplexing to the POA lookup phase, which operates as follows:

1. All lookups start at the `RootPOA`.
2. The `RootPOA` will maintain a `POA table` that points to all the POAs in the hierarchy.
3. Object keys will include an index into the `POA table` to identify the POA where the object was activated. TAO's ORB Core will use 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 will contain both the name and the index.

Using active demultiplexing for POA lookup should provide optimal predictability and scalability, just as it does when used for servant demultiplexing, which is 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 2.2.2. TAO uses the Service Configurator [24], Bridge, and Strategy design patterns [25] to defer the configuration of the desired servant demultiplexing strategy until ORB initialization, which can be performed either *statically* (*i.e.*, at compile-time) or *dynamically* (*i.e.*, at run-time) [13]. Figure 4 illustrates the class hierarchy of strategies that can be configured into TAO's POAs.

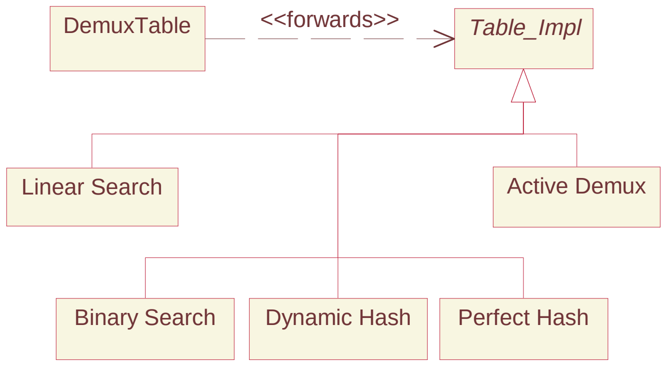


Figure 4: TAO's Class Hierarchy for POA Active Object Map Strategies

To evaluate the scalability of TAO, our experiments used a range of servants, 1 to 500 by increments of 100, in the server. Figure 5 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, *i.e.*, its performance degrades rapidly as the number of servants increase.

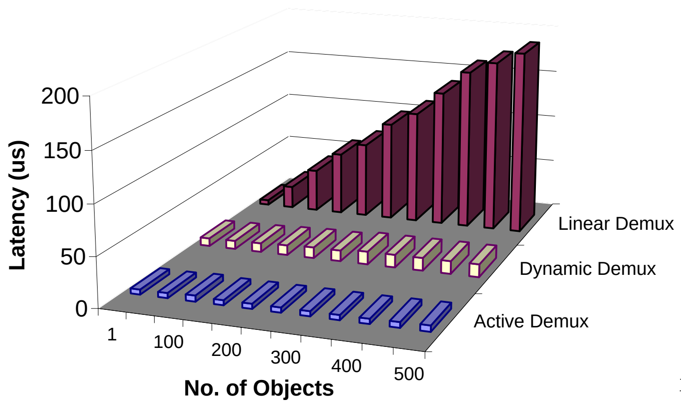


Figure 5: Servant Demultiplexing Latency with Alternative Search Techniques

Note that we did not implement the perfect hashing strategy for servant demultiplexing. Although it is possible to know the set of servants on each POA for certain statically configured applications *a priori*, 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 [23], which is a freely available perfect hash function generator we developed.

GPERF [26] automatically constructs perfect hash functions from a user-supplied list of keywords. In addition to the perfect hash functions, GPERF can also generate linear and binary search strategies.

Figure 6 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.

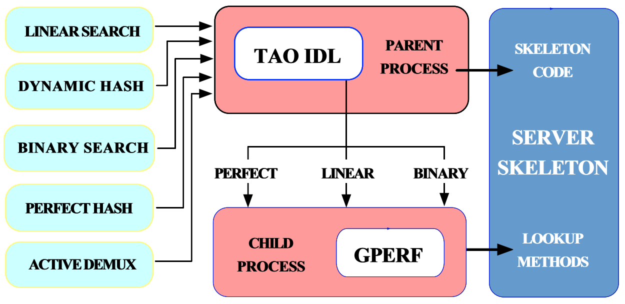


Figure 6: Integrating TAO's IDL Compiler and GPERF

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. Since active demultiplexing cannot be used without modifying the GIOP protocol.[1] 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 7 plots operation demultiplexing latency as a function of the number of operations. This figure illustrates that

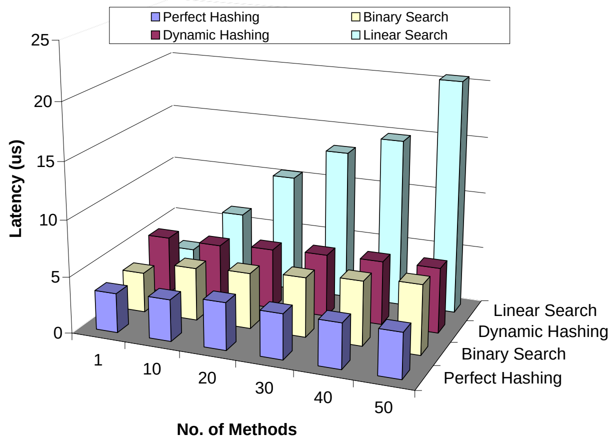


Figure 7: Operation Demultiplexing Latency with Alternative Search Techniques

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 complicates worst-case schedulability analysis for real-time applications.

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

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 2.2.3 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.

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 5, linear search is prohibitively expensive as the number of servants in the Active Object Map increases. This overhead is particularly problematic for real-time applications, such as avionics mission computing systems [11], 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 8 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 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-based lookups. However, this additional processing does not affect the critical path of Object Id-based lookups during run-time.

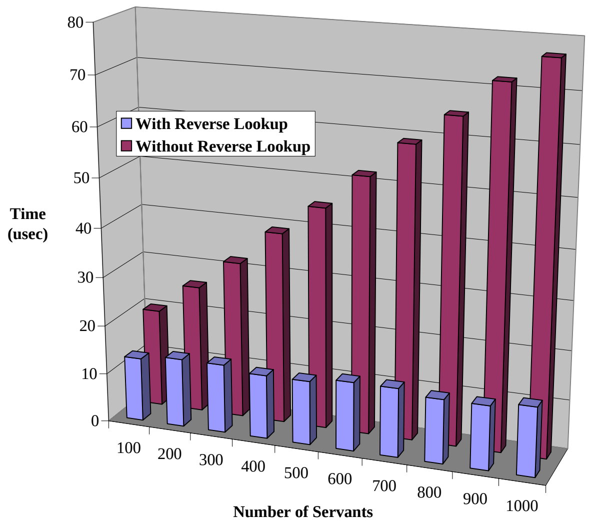


Figure 8: Benefits of Adding a Reverse-Lookup Map to the POA

Summary of TAO's POA demultiplexing strategies: Based on the results of our benchmarks described above, Figure 9 summarizes the demultiplexing strategies that we have determined to be most appropriate for real-time applications [11]. Figure 9 shows the use of active demultiplexing for the POA names, active demultiplexing for the servants, and perfect hashing for the operation names. Our previous experience [27, 4, 28, 6, 7] measuring the performance of

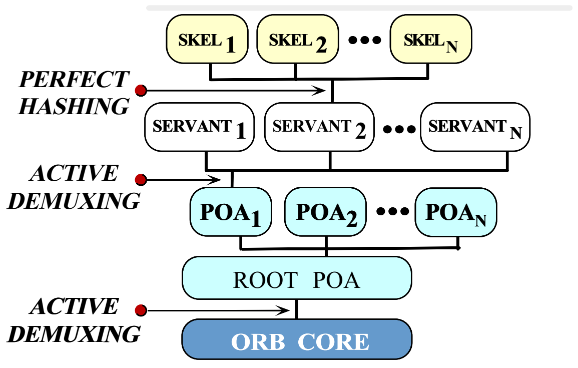


Figure 9: TAO's Default Demultiplexing Strategies

CORBA implementations showed TAO is more efficient and predictable than widely used conventional CORBA ORBs.

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 [1].

2.3 Optimizing Object Key Processing in POA Upcalls

Motivation: Since the POA is in the critical path of request processing in a server ORB, it is important to optimize its processing. Figure 10 shows a naive way to parse an object key. In this approach, the object key is parsed and the individual fields of the key are stored in separate components. Unfortunately, this approach (1) allocates memory dynamically for each individual object key field and (2) copies data to move the object key fields into individual objects.

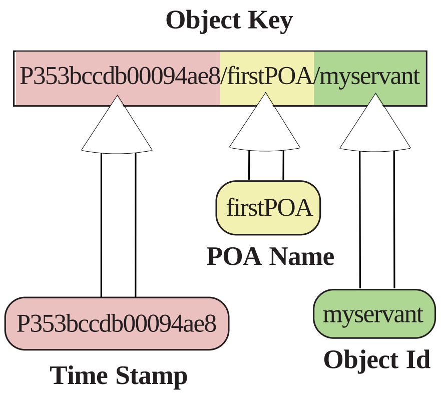


Figure 10: Naive Parsing of Object Keys

TAO's object key upcall optimizations: TAO provides the following object key optimizations based on the principle patterns of *avoiding obvious waste* and *avoiding unnecessary generality*. TAO leverages the fact that the object key is available through the entire upcall and is not modified. Thus, the individual components in the object key can be optimized to point directly to their correct locations, as shown in Figure 11. This eliminates wasteful memory allocations and data copies. This optimization is entirely compliant with the standard CORBA specification.

2.4 Optimizing POA Predictability and Minimizing Footprint

Motivation: To adequately support real-time applications, an ORB's Object Adapter must be *predictable* and *minimal*. For instance, it must omit non-deterministic operations to improve end-to-end predictability. Likewise, it must provide a minimal memory footprint to support embedded systems [15].

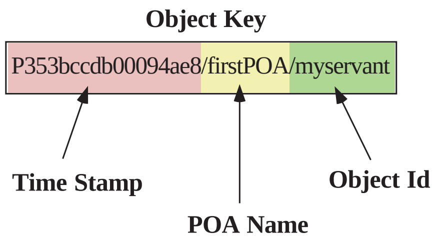


Figure 11: TAO's Optimized Parsing of Object Keys

TAO's predictability optimizations: Based on the principle patterns of *avoiding unnecessary generality* and *relaxing system requirements*, we enhanced TAO's POA to selectively disable the following features in order to improve end-to-end predictability of request processing:

- **Servant Managers are not required:** There is no need to locate servants in a real-time environment since all servants must be registered with POAs *a priori*.

- **Adapter Activators are not required:** Real-time applications create all their POAs at the beginning of execution. Therefore, they need not use or provide an adapter activator. The alternative is to create POAs during request processing, in which case end-to-end predictability is hard to achieve.

- **POA Managers are not required:** The POA must not introduce extra levels of queueing in the ORB. Queueing can cause priority inversion and excessive locking. Therefore, the POA Manager in TAO can be disabled.

TAO's footprint optimizations: In addition to increasing the predictability of POA request processing, omitting these features also decreases TAO's memory footprint. These omissions were done in accordance with the Minimum CORBA specification [29], which removes the following features from the CORBA 2.2 specification [1]:

- Dynamic Skeleton Interface
- Dynamic Invocation Interface
- Dynamic Any
- Interceptors
- Interface Repository
- Advanced POA features
- CORBA/COM interworking

Component	**CORBA**	**Minimum CORBA**	**Percentage Reduction**
POA	281,896	207216	26.5
ORB Core	347,080	330,304	4.8
Dynamic Any	131,305	0	100
CDR Interpreter	68,687	68,775	0
IDL Compiler	10,488	10,512	0
Pluggable Protocols	14,610	14,674	0
Default Resources	7,919	7,975	0
Total	861,985	639,456	25.8

Table 2: Comparison of CORBA with Minimum CORBA Memory Footprint

Table 2 shows the footprint reduction achieved when the features listed above are excluded from TAO. The 25.8% reduction in memory footprint for Minimum CORBA is fairly significant. However, we plan to reduce the footprint of TAO even further by streamlining its CDR Interpreter [15]. In Minimum CORBA, TAO's CDR Interpreter only needs to support the static skeleton interface (SSI) and static invocation interface (SII). Thus, support for the dynamic skeleton interface (DSI) and dynamic invocation interface (DII) can be omitted.

3 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 [1]. 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 executing 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 a CORBA ORB Core to support real-time applications requires the resolution of many design challenges. This section outlines several of these challenges and describes the optimization principle patterns we applied to maximize the predictability, performance, and scalability of TAO's ORB Core. These optimizations include transparently collocating clients and servants that are in the same address space, minimizing dynamic memory allocations and data copies, and minimizing GIOP/IIOP protocol overhead. Additional optimizations for real-time ORB Core connection management and concurrency strategies are described in [30].

3.1 Collocation Optimizations

Motivation: In addition to separating interface and implementation, a key strength of CORBA is its decoupling of (1) servant implementations from (2) how servants are configured into server processes throughout a distributed system. In practice, CORBA is used primarily to communicate between remote objects. However, there are configurations where a client and servant must be collocated in the same address space [31]. In this case, there is no need to incur the overhead of data marshaling or transmitting requests and replies through a "loopback" transport device, which is an application of the principle pattern of *avoiding obvious waste*.

TAO's collocation optimization technique: TAO's POA optimizes for collocated client/servant configurations by generating a special stub for the client, which is an application of the principle pattern of *relaxing system requirements*. This stub forwards all requests to the servant and eliminates data marshaling, which is an application of the principle pattern of *avoiding waste*. Figure 12 shows the classes produced by TAO's IDL compiler.

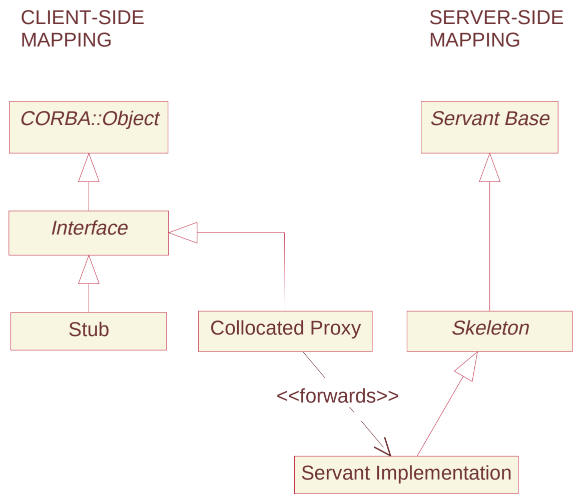


Figure 12: TAO's POA Mapping and Collocation Class

The stub and skeleton classes shown in Figure 12 are required by the POA specification; the collocation class is specific to TAO. Collocation is transparent to the client since it only accesses the abstract interface and never uses the collocation class directly. Therefore, the POA provides the collocation class, rather than the regular stub class, when the servant resides in the same address space as the client.

Supporting transparent collocation in TAO: Clients can obtain an object reference in several ways, *e.g.*, from a CORBA Naming Service or from a Lifecycle Service generic factory operation. Likewise, clients can use `string_to_object` to convert a stringified interoperable object reference (IOR) into an object reference. To ensure locality transparency, an ORB's collocation optimization must determine if an object is collocated. If it is, the ORB returns a collocated stub – if it is not, the ORB returns a regular stub to a distributed object.

The specific steps used by TAO's collocation optimizations are described below:

Step 1 – Determining collocation: To determine if an object reference is collocated, TAO's ORB Core maintains a *collocation table*, which applies the principle of *maintaining extra state*. Figure 13 shows the internal structure for collocation table management in TAO. Each collocation table maps

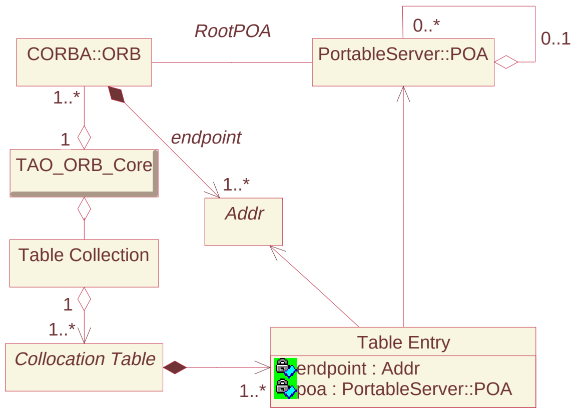
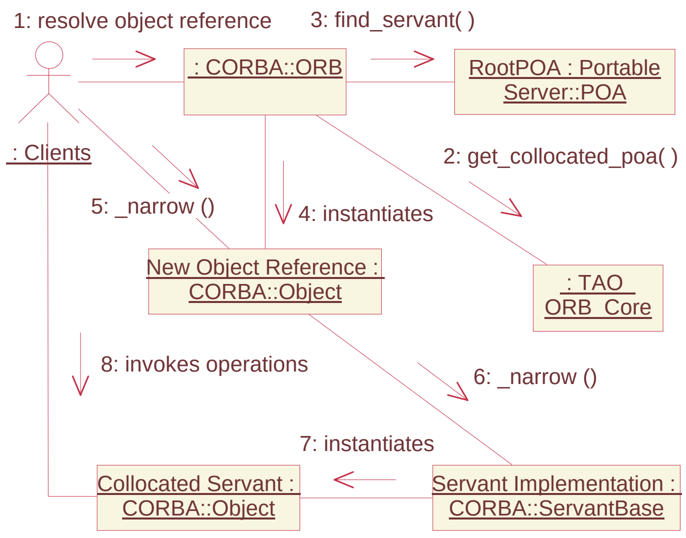


Figure 13: Class Relationship of TAO's Collocation Tables

an ORB's transport endpoints to its RootPOA. In the case of IIOP, endpoints are specified using {hostname, port number} tuples.

Multiple ORBs can reside in a single server process. Each ORB can support multiple transport protocols and accept requests from multiple transport endpoints. Therefore, TAO maintains multiple collocation tables for all transport protocols used by ORBs within a single process. Since different protocols have different addressing methods, maintaining protocol specific collocation tables allows us to strategize and optimize the lookup mechanism for each protocol.

Step 2 – Obtaining a reference to a collocated object: A client acquires an object reference either by resolving an imported IOR using `string_to_object` or by demarshaling an incoming object reference. In either case, TAO examines the corresponding collocation tables according to the profiles carried by the object to determine if the object is collocated or not. If the object is collocated, TAO performs the series of steps shown in Figure 14 to obtain a reference to the collocated object.



1: resolve object reference
3: find_servant( )
: CORBA::ORB
RootPOA : Portable Server::POA
: Clients
2: get_collocated_poa( )
5: _narrow ()
4: instantiates
New Object Reference : CORBA::Object
: TAO ORB_Core
8: invokes operations
6: _narrow ()
7: instantiates
Collocated Servant : CORBA::Object
Servant Implementation : CORBA::ServantBase

Figure 14: Finding a Collocated Object in TAO

As shown in Figure 14, when a client process tries to resolve an imported object reference **(1)**, the ORB checks **(2)** the collocation table maintained by TAO's ORB Core to determine if any object endpoints are collocated. If a collocated endpoint is found this check succeeds and the RootPOA corresponding to the endpoint is returned. Next, the matching Object Adapter is queried for the servant, starting at its RootPOA **(3)**. The ORB then instantiates a generic `CORBA::Object` **(4)** and invokes the `_narrow` operation on it. If a servant is found, the ORB's `_narrow` operation **(5)** invokes the servant's `_narrow` method **(6)** and a collocated stub is instantiated and returned to the client **(7)**. Finally, clients invoke operations **(8)** on the collocated stub, which forwards the operation to the local servant via a virtual method call.

If the imported object reference is not collocated, then either operation **(2)** or **(3)** will fail. In this case, the ORB invokes the `_is_a` method to verify that the remote object matches the target type. If the test succeeds, a distributed stub is created and returned to the client. All subsequent operations are invoked remotely. Thus, the process of selecting collocated stubs or non-collocated stubs is completely transparent to clients and it's only performed at the time of object reference creation.

Step 3 – Performing collocated object invocations: Collocated operation invocations in TAO borrow the client's thread-of-control to execute the servant's operation. Therefore, they are executed within the client thread at its thread priority.

Although executing an operation in the client's thread is very efficient, it is undesirable for certain types of real-time applications [32]. For instance, priority inversion can occur when a client in a lower priority thread invokes operations on a collocated object in a higher priority thread. To provide greater access control over the scope of TAO's collocation optimizations, applications can associate different access policies to endpoints so they only appear collocated to certain priority groups. Since endpoints and priority groups in many real-time applications are statically configured, this access control lookup does not impose additional overhead.

Empirical results: To measure the performance gain from TAO's collocation optimizations, we ran server and client threads in the same process. Two platforms were used to benchmark the test program: a dual 300 Mhz UltraSparc-II running SunOS 5.5.1 and a dual 400 Mhz Pentium-II running Microsoft Windows NT 4.0 (SP3.) The test program was run both with TAO's collocation optimizations enabled and disabled to compare the performance systematically.

Figure 15 shows the performance improvement, measured in calls-per-second, using TAO's collocation optimizations. Each operation cubed a variable-length sequence of `long`s that contained 4 and 1,024 elements, respectively. As expected, collocation greatly improves the performance of operation invocations when servants are collocated with clients. Our results show, depending on the size of arguments passed to the operations, performance improves from 2,000% to 200,000%. Although the test results are foreseeable, they show that by using TAO's collocation optimization, invocations on collocated CORBA objects can be as fast as calling functions on local C++ objects.

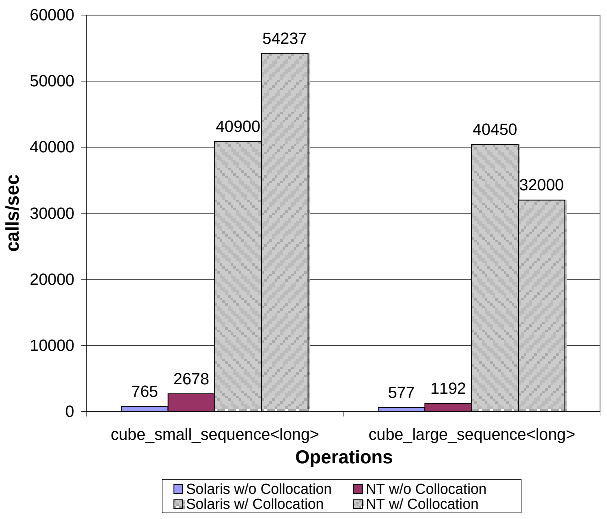


Figure 15: Results of TAO's Collocation Optimizations

TAO's collocation optimizations are not totally compliant with the CORBA standard since its collocation class forwards all requests directly to the servant class. Although this makes the common case very efficient, this implementation does not support the following advanced POA features:

- `POA::Current` is not setup
- Interceptors are bypassed
- POA Manager state is ignored
- Servant Managers are not consulted
- Etherealized servants can cause problems
- Location forwarding is not supported
- The POA's `Thread_Policy` is circumvented

Adding support for these features to TAO's collocation class slow downs the collocation optimization, which is why TAO currently omits these features. We plan to support these advanced features in future releases of TAO so that if applications know these advanced features are not required they can be ignored selectively.

3.2 Memory Management Optimizations

Motivation: A key source of overhead and non-determinism in conventional ORB Core implementations is 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 stems from the use of dynamic memory allocation, which is problematic for real-time ORBs. For instance, dynamic memory can fragment the global process heap, which decreases ORB predictability. Likewise, locks used to access a global heap from multiple threads can increase synchronization overhead and incur priority inversion [30].

Another significant source of memory management overhead involves excessive data copying. For instance, conventional ORB's 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 optimizations leverage off the design of 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-side, 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-side, the thread that reads a request completes the upcall to user code, also eliminating queueing within the ORB. These optimizations are based on the principle pattern of *exploiting locality* and *optimizing for the common case*.

By avoiding thread context switches and queueing, TAO can benefit from memory management optimizations based on *thread-specific storage*. Thread-specific storage is a common design pattern [13] 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 pattern of *avoiding waste*. TAO uses this pattern to place its memory allocators into thread-specific storage. Using a thread-specific memory pool eliminates the need for intra-thread allocator locks, reduces fragmentation in the allocator, and helps to minimize priority inversion in real-time applications.

In addition, TAO minimizes unnecessary data copying by keeping a linked list of CDR buffers. As shown in Figure 16, 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 as zero-copy schemes [33] that share memory pools between user processes, the OS kernel, and network interfaces.

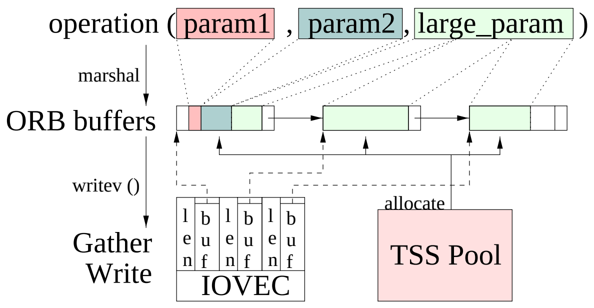


Figure 16: TAO's Internal Memory Managment

Empirical results: Figure 17 compares buffer allocation time for a CORBA request using thread-specific storage (TSS) allocators with that of using a global allocator. These experiments were executed on a Pentium II/450 with 256Mb of RAM, running LynxOS 3.0. The test program contained a group of ORB buffer (de)allocations intermingled with a pseudo-random sequence of regular (de)allocations. This 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.

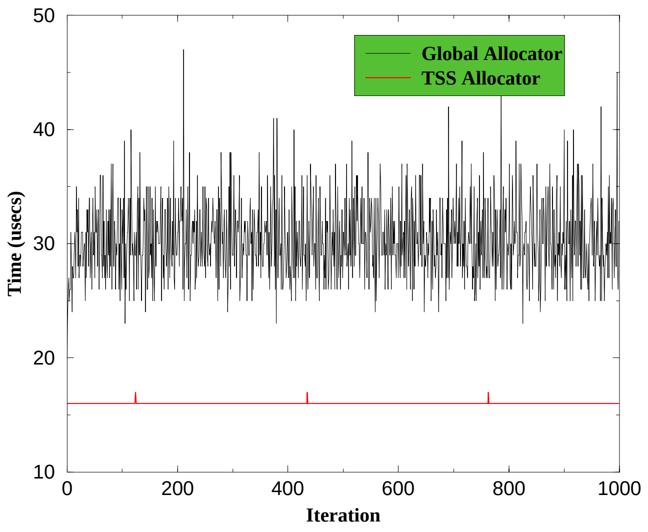


Figure 17: Buffer Allocation Time using TSS and Global Allocators

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 allocator behaves non-deterministically, it is not suitable for hard real-time systems.

Our results in Figure 17 illustrate that TAO's TSS allocators isolate the ORB from variations in global memory allocation strategies. 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 [30].

3.3 Minimizing ORB Protocol Message Footprint

Motivation: Real-time systems have traditionally been developed using proprietary protocols that are hard-coded for each application. In theory, CORBA's GIOP/IIOP protocols obviate the need for proprietary protocols. In practice, however, many developers of real-time applications are justifiably concerned that standard CORBA protocols will cause excessive overhead. For example, some applications have very strict constraints on latency, which is affected by the total time required to transmit the message. Other applications, such as mobile PDAs running over wireless access networks, have limited bandwidth, which makes them more sensitive to protocol message footprint overhead.

TAO's ORB protocol optimization techniques: A GIOP request includes a number of fields, such as the version number, that are required for interoperability among ORBs. However, certain fields are not required in all application domains. For instance, the magic number and version fields can be omitted if a single supplier and single version is used for ORBs in a real-time embedded system. Likewise, if the communicating ORBs are running on systems with the same endianess, *i.e.*, big-endian or little-endian, the byte order flag can be omitted from the request.

Since embedded and real-time systems typically run the same ORB implementation on similar hardware, we have modified TAO to optionally remove some fields from the GIOP header and the GIOP Request header when the `-ORBgioplite` option is given to the client and server `CORBA::ORB_init` method. The fields removed by this optimization are shown in Table 3. These optimizations are guided by the principle patterns of *relaxing system requirements* and *avoiding unnecessary generality*.

Header Field	**Size**
GIOP magic number	4 bytes
GIOP version	2 bytes
GIOP flags (byte order)	1 byte
Request Service Context	$\geq$ 4 bytes
Request Principal	$\geq$ 4 bytes
Total	$\geq$ 15 bytes

Table 3: Messaging Footprint Savings for TAO's GIOPlite Optimization

Empirical results: We conducted an experiment to measure the performance impact of omitting the GIOP fields in Table 3. These experiments were executed on a Pentium II/450 with 256Mb of RAM, running LynxOS 3.0 in loopback mode. Table 4 summarizes the results, expressed in calls-per-second:

	Marshaling Enabled			**Marshaling Disabled**		
	min	max	avg	min	max	avg
GIOP	2,878	2,937	2,906	2,912	2,976	2,949
GIOPlite	2,883	2,978	2,943	2,911	3,003	2,967

Table 4: Performance of TAO's GIOP and GIOPlite Protocol Implementations

Our empirical results reveal a slight, but measurable, 2% improvement when removing the GIOP message footprint "overhead." More importantly though, these changes do not affect the standard CORBA APIs used to develop applications. Therefore, programmers can focus on the development of applications, and if necessary, TAO can be optimized to use this lightweight version of GIOP.

To obtain more significant protocol optimizations, we are adding a *pluggable protocols* framework to TAO [34]. This framework generalizes TAO's current `-ORBgioplite` option to support both pluggable ORB protocols (ESIOPs) *and* pluggable transport protocols.

4 Related Work

Demultiplexing is an operation that routes messages through the layers of an ORB endsystem. Most protocol stacks models, such as the Internet model or the ISO/OSI reference model, require some form of multiplexing to support interoperability with existing operating systems and peer protocol stacks. Likewise, conventional CORBA ORBs utilize several extra levels of demultiplexing at the application layer to associate incoming client requests with the appropriate servant and operation (as shown in Figure 2).

Related work on demultiplexing focuses largely on the lower layers of the protocol stack, *i.e.*, the transport layer and below, as opposed to the CORBA middleware. For instance, [21, 35, 22, 36] study demultiplexing issues in communication systems and show how layered demultiplexing is not suitable for applications that require real-time quality of service guarantees.

Packet filters are a mechanism for efficiently demultiplexing incoming packets to application endpoints [37]. A number of schemes to implement fast and efficient packet filters are available. These include the BSD Packet Filter (BPF) [38], the Mach Packet Filter (MPF) [39], PathFinder [40], demultiplexing based on automatic parsing [41], and the Dynamic Packet Filter (DPF) [36].

As mentioned before, most existing demultiplexing strategies are implemented within the OS kernel. However, to optimally reduce ORB endsystem demultiplexing overhead requires a vertically integrated architecture that extends from the OS kernel to the application servants. Since our ORB is currently implemented in user-space, however, our work focuses on minimizing the demultiplexing overhead in steps 3, 4, 5, and 6 (which are shaded in Figure 2).

5 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 this economic climate, 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 [42] or Java RMI [43], 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 5 summarizes which TAO optimizations are associated with which principle patterns, as well as which optimizations conform to the CORBA standard and which are non-standard.

Optimization	Principle Patterns	Compliant
Request demuxing	Precompute, Avoid waste Passing hints in header Relaxing system requirements Using specialized routines Not tied to reference models Adding extra state	yes
Object keys in upcalls	Avoid waste Exploit locality	yes
Predictability and footprint	Relaxing system requirements	yes
Collocation	Relax system requirements Avoid waste Add extra state	no
Memory management	Exploit Locality Avoid waste Optimize for common case	yes
Protocol msg footprint	Avoid generality Relax system requirements	no

Table 5: Degree of CORBA-compliance for Real-time Optimization Principle Patterns

Our primary focus on the TAO project has been to research, develop, and optimize policies and mechanisms that allow CORBA to support hard real-time systems, such as avionics mission computing [11]. 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 statistical 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.

Acknowledgements

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 anonymous reviewers their constructive suggestions for improving the paper.

References

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

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

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

[4] 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.

[5] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.

[6] 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.

[7] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," in *Proceedings of the 4^{th} IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

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

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

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

[11] 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.

[12] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of RIO – A Real-time I/O Subsystem for ORB Endsystems," in *Proceedings of the 5^{th} IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), IEEE, June 1999.

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

[14] 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.

[15] 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*, 1999.

[16] G. Varghese, "Algorithmic Techniques for Efficient Protocol Implementations ," in *SIGCOMM '96 Tutorial*, (Stanford, CA), ACM, August 1996.

[17] I. Pyarali and D. C. Schmidt, "An Overview of the CORBA Portable Object Adapter," *ACM StandardView*, vol. 6, Mar. 1998.

[18] D. C. Schmidt and S. Vinoski, "C++ Servant Managers for the Portable Object Adapter," *C++ Report*, vol. 10, Sept. 1998.

[19] D. C. Schmidt and S. Vinoski, "Using the Portable Object Adapter for Transient and Persistent CORBA Objects," *C++ Report*, vol. 10, April 1998.

[20] D. Schmidt and S. Vinoski, "Distributed Callbacks and Decoupled Communication in CORBA," *C++ Report*, vol. 8, October 1996.

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

[22] 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.

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

[24] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the* 3^{rd} *Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

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

[26] A. Gokhale, D. C. Schmidt, C. O'Ryan, and A. Arulanthu, "The Design and Performance of a CORBA IDL Compiler Optimized for Embedded Systems," in *Submitted to the LCTES workshop at PLDI '99*, (Atlanta, GA), IEEE, May 1999.

[27] 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.

[28] 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.

[29] Object Management Group, *Minimum CORBA - Joint Revised Submission*, OMG Document orbos/98-08-04 ed., August 1998.

[30] 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.

[31] D. C. Schmidt and S. Vinoski, "Developing C++ Servant Classes Using the Portable Object Adapter," *C++ Report*, vol. 10, June 1998.

[32] D. L. Levine, C. D. Gill, and D. C. Schmidt, "Dynamic Scheduling Strategies for Avionics Mission Computing," in *Proceedings of the 17th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Nov. 1998.

[33] 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.

[34] F. Kuhns, C. O'Ryan, D. C. Schmidt, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware," in *Submitted to the IFIP* 6^{th} *International Workshop on Protocols For High-Speed Networks (PfHSN '99)*, (Salem, MA), IFIP, August 1999.

[35] D. C. Feldmeier, "Multiplexing Issues in Communications System Design," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 209–219, ACM, Sept. 1990.

[36] D. R. Engler and M. F. Kaashoek, "DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation," in *Proceedings of ACM SIGCOMM '96 Conference in Computer Communication Review*, (Stanford University, California, USA), pp. 53–59, ACM Press, August 1996.

[37] J. C. Mogul, R. F. Rashid, and M. J. Accetta, "The Packet Filter: an Efficient Mechanism for User-level Network Code," in *Proceedings of the* 11^{th} *Symposium on Operating System Principles (SOSP)*, November 1987.

[38] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 259–270, Jan. 1993.

[39] M. Yuhara, B. Bershad, C. Maeda, and E. Moss, "Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages," in *Proceedings of the Winter Usenix Conference*, January 1994.

[40] M. L. Bailey, B. Gopal, P. Sarkar, M. A. Pagels, and L. L. Peterson, "Pathfinder: A pattern-based packet classifier," in *Proceedings of the* 1^{st} *Symposium on Operating System Design and Implementation*, USENIX Association, November 1994.

[41] M. Jayaram and R. Cytron, "Efficient Demultiplexing of Network Packets by Automatic Parsing," in *Proceedings of the Workshop on Compiler Support for System Software (WCSSS 96)*, (University of Arizona, Tucson, AZ), February 1996.

[42] Microsoft Corporation, *Distributed Component Object Model Protocol (DCOM)*, 1.0 ed., Jan. 1998.

[43] Sun Microsystems, Inc, *Java Remote Method Invocation Specification (RMI)*, Oct. 1998.