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

An earlier version of this paper appeared at the 5^{th} USENIX Conference on OO Technologies and Systems (COOTS '99), San Diego, CA, May 1999.

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, demultiplexing, and protocol processing 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

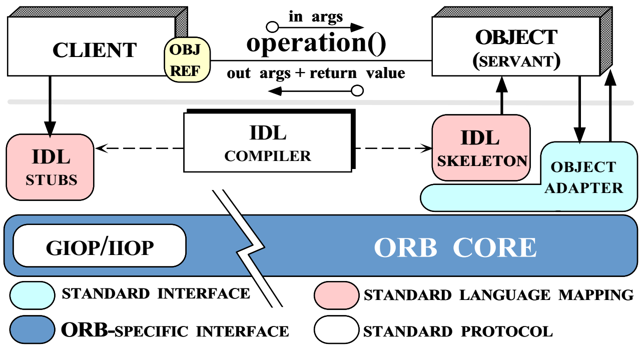

Figure 1: Key Components in the CORBA 2.x Reference Model

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→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 `structs`. 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.

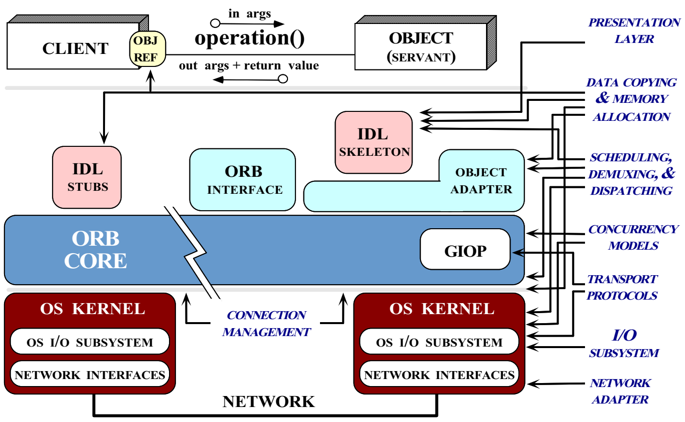

Figure 2: Real-time Features and Optimizations Necessary to Meet End-to-end QoS Requirements in ORB Endsystems

The first-generation of ORBs lacked many of the features and optimizations [6, 7, 8, 9] shown in Figure 2. This situation was not surprising, of course, since ORB developers focused initially on refining the OMG specifications [10] and developing core infrastructure components, such as the basic ORB communication mechanisms. In contrast, second-generation ORBs, such as The ACE ORB (TAO) [11], have leveraged the maturations of standards [12, 5, 13], patterns [14], and QoS-enabled framework components [15, 16], to provide end-to-end QoS guarantees to applications both *vertically* (*i.e.*, network interface ↔ 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 [17] and dynamic [18] scheduling, event processing [19], I/O subsystem integration [20], ORB Core architectures [21], systematic benchmarking of multiple ORBs [6], and design patterns for ORB extensibility [14]. This paper focuses on other previously unexplored dimensions in the high-performance and real-time ORB endsystem design space: *Ob-*

ject Adapter and ORB Core optimizations for (1) server-side concurrency, (2) collocation, (3) memory management, (4) ORB protocol processing, and (5) CORBA request demultiplexing .

The optimizations used in TAO are guided by a set of *principle patterns* [22] that we have applied in prior work to optimize middleware [11] and lower-level networking software [23], 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 [24]. 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 [21]. 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 collocation: The principle pattern of avoiding gratuitous waste 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). After looking up the servant in the POA, operations are directly invoked on servants in the context of the calling thread, thereby transforming operation invocations into local virtual method calls. TAO also supports direct collocated method invocations that bypass POA for more static configuration. Section 2.2 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 2.3 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.

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 2.4 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 and relaxing system requirements.

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 [8]. 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; Section 4 describes related work; and Section 5 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, 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 ORB Core connection management are described in [21].

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* [24]. As shown in Figure 3, the components in this model include a 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)

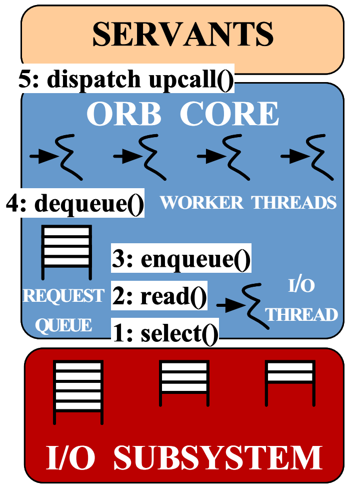

Figure 3: Server Queue-based Worker Thread Pool Concurrency Model

shares dynamically allocated data buffers between threads, which works against CPU cache affinity [25] and limits the applicability of other optimizations, such as thread-specific storage (TSS) memory management described in Section 2.3, (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

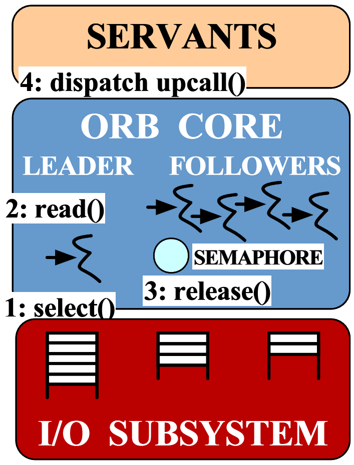

Figure 4: Leader/Followers Thread Pool Server Concurrency Model

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 [26], 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 concurrency 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.

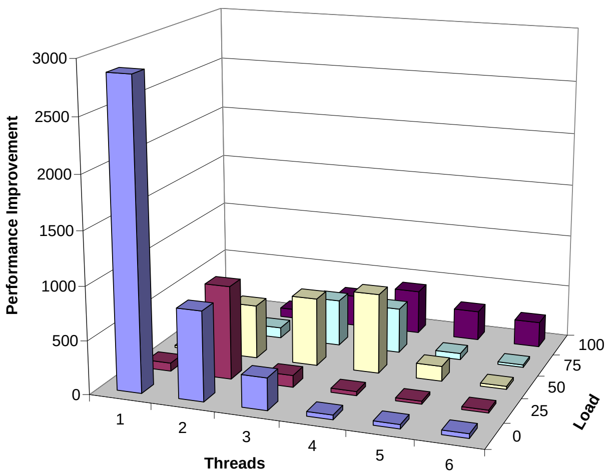

Figure 5: Performance of Leader/Follower vs. Queue-based Worker Thread Pool

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 Collocation Optimizations

Motivation: In addition to separating interface from implementation, CORBA decouples servant implementations from how servants are configured into server processes. In practice, CORBA is used primarily to communicate between distributed objects. However, there are configurations where a client and servant must be *collocated* in the same address space [27]. In this case, there is no need to incur the overhead of data marshaling or transmitting requests/replies through a "loopback" transport device. Such collocation optimizations are an application of the principle pattern of avoiding gratuitous waste.

TAO's collocation optimization technique: TAO optimizes for collocated client/servant configurations by generating a special stub for the client, which is an application of the principle pattern of replacing inefficient general-purpose operations with optimized special-purpose ones. This stub forwards all requests to the servant and eliminates data marshaling, thereby applying the principle pattern of avoiding gratuitous waste. TAO supports the following two collocation strategies in its stubs:

- **Thru_POA:** The `Thru_POA` strategy is the default collocation strategy in TAO. In this strategy, a *safe* collocated stub is used to handle operation invocations on a collocated object. Invoking an operation on this collocated stub ensures: (1) the server ORB (which may or may not be the same ORB as the clients') has not been shut down, (2) the thread-safety of all ORB and POA operations, (3) the POA managing the servant still exists, (4) the POA Manager of this POA is queried to make sure upcalls are allowed to be performed on the POA's servants, (5) the servant for the collocated object is still active, (6) the `POA::Current`'s context is initialized for this upcall, and (7) the POA's threading policy is respected. If it is safe to invoke the operation, the stub uses the servant exported from server's POA, downcasts it to the servant, and forwards the operation directly to the servant. The so-called safe stubs ensure that the `POA::Current` is restored to its previous context before the current invocation, various locks in the POA are released, and the servant upcall counter is restored, after either a successful or an unsuccessful operation invocation.

• **Direct:** In this TAO-specific extension, the collocation class forwards all requests directly to the servant class, *i.e.*, the POA is not involved at all. This design applies the principle pattern of optimizing for the common case, which ensures the performance is the same as for a direct virtual method call. However, this implementation does not support the following standard POA features: (1) the `POA::Current` is not initialized, (2) interceptors are bypassed, (3) POA Manager state is ignored, (4) Servant Managers are not consulted, (5) etherealized servants can cause problems, (6) location forwarding is not supported, and (7) the POA's `Thread_Policy` is circumvented. As shown in Figure 9, these features decrease collocation performance somewhat. Therefore, TAO provides the `Direct` strategy that is optimized for real-time applications with very stringent latency requirements.

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 remote stub to a distributed object.

Figure 6 shows the classes generated by TAO's IDL compiler. The stub and skeleton classes are required by the POA specification, though 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. As with remote method invocations, TAO's ORB Core assumes the responsibility of locating servants and makes sure the collocated stub class, rather than the remote stub class, is used by a client when the servant resides in the same address space.

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 7 shows the internal structure for collocation table management in TAO. Each collocation table maps

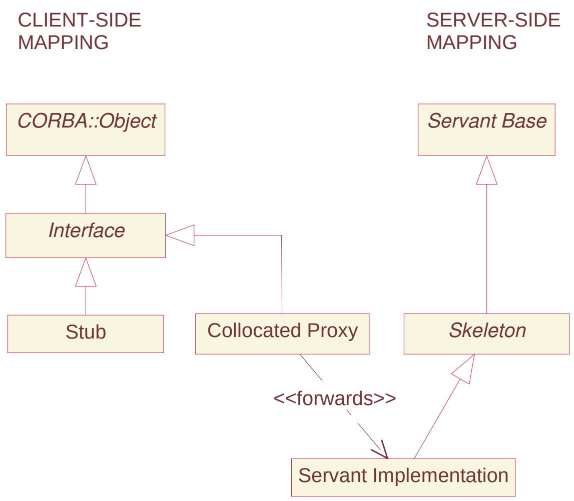

Figure 6: TAO's POA Mapping and Collocation Class

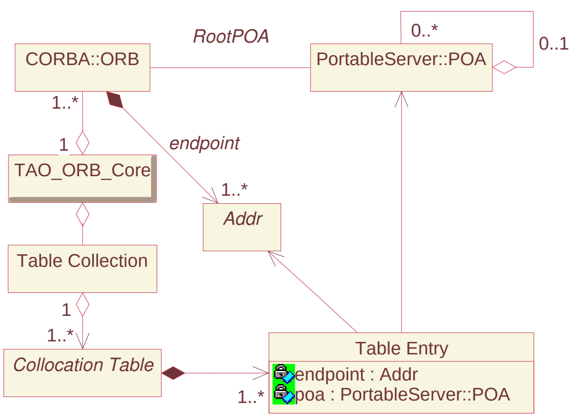

Figure 7: Class Relationship of TAO's Collocation Tables

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 formats, maintaining protocol specific collocation tables allows TAO 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 steps shown in Figure 8 to obtain a reference to the collocated object.

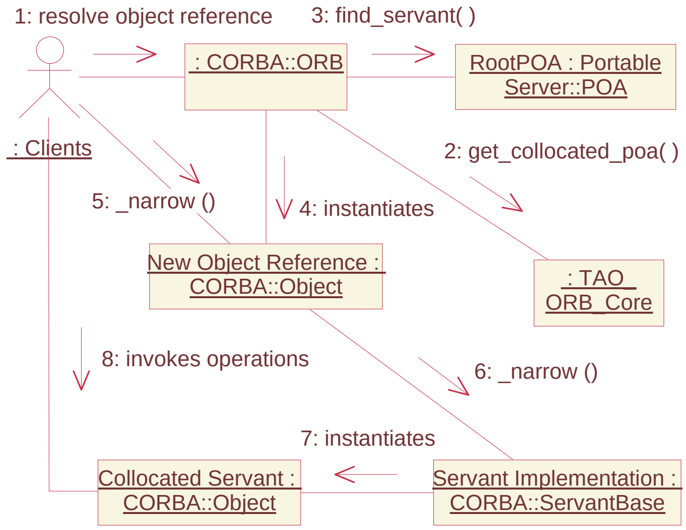

Figure 8: Finding a Collocated Object in TAO

When the `Thru_POA` collocation strategy is enabled, the ORB checks if the imported object reference is collocated or not only when it resolves the object reference. To determine this, TAO examines the endpoint information in the collocation table maintained by TAO's ORB Core. If the imported object reference is collocated, an object reference with the safe collocated stub is generated. This safe collocated stub contains information about the matching Object Adapter and server ORB.

If the ORB uses the `Direct` collocation strategy, the ORB resolves an imported object reference using the steps shown in Figure 8. To resolve an 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, 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` operation **(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 direct virtual method call.

Either operation **(2)** or **(3)** will fail if the imported object reference is not collocated. In this case, the ORB invokes the `_is_a` operation to verify that the remote object matches the target type. If the test succeeds, a remote stub is created and returned to the client and all subsequent operations are distributed. Thus, the process of selecting collocated stubs or non-collocated stubs is completely transparent to clients and are performed only 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 [28]. 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, therefore, applications can associate different access policies to endpoints so they appear collocated only to certain priority groups. Since endpoints and priority groups in many real-time applications are statically configured, this access control lookup imposes no 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 quad-CPU 300 Mhz UltraSparc-II running SunOS 5.7 and a dual-CPU 333 Mhz Pentium-II running Microsoft Windows NT 4.0 with SP4. To compare performance systematically, the test program was run with the `Thru_POA` collocation strategy, the `Direct` collocation strategy, direct invocation on servants, and as well as with neither collocation optimization, *i.e.*, using remote stubs via the loopback network interface.

Figure 9 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. The performance of operation invocations improves dramatically when servants are collocated with clients. Depending on the size of arguments passed to the operations, our results show that `Thru_POA` improves performance from 3,000% to 6,000% compared to the loopback device. The application fo `Thru_POA` collocation optimization saves the time to transmit the invocation arguments and return values back and forth thru the local loopback device which also involve copying data between user and kernel memories. Also shown in the figure, we gain 130% $\sim$ 180% performance improvement by skip-

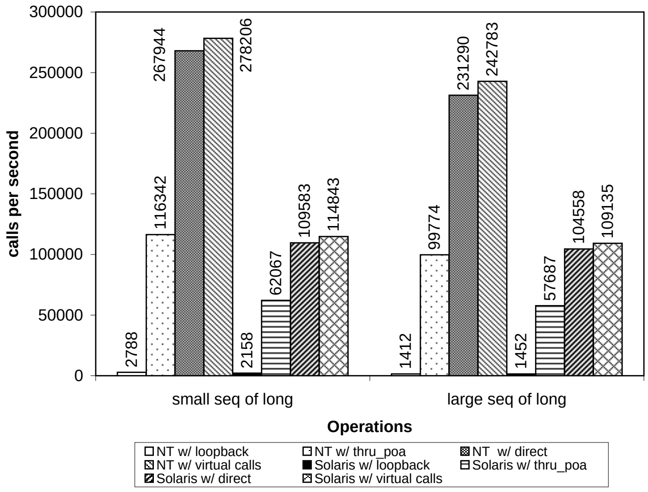

Figure 9: Results of TAO's Collocation Optimizations

ping related POA and ORB operations when switching from Thru_POA strategy to Direct strategy. The size of the arguments does not have a significant effect on the performance improvement in this case. Invoking the operation directly to servant through virtual function calls represents the optimal performance we can get in this scenario. As shown in the figure, there's only 5% performance gain compared to Direct collocateion strategy which is resulted from the extra virtual function call within the collocated stub.

TAO's Thru_POA collocation strategy is completely CORBA compliant. Although there are some overhads involved in the Thru_POA collocation strategy compared to Direct collocation, there is still a significant performance improvement over the non-optimized scheme. Moreover, the Thru_POA strategy preserves the semantics of CORBA's object architecture model and maintain the uniform behavior no matter an object is collocated or remote. For users who have systems that are more statically configured, they can take advantage of the TAO-specific Direct collocation strategy which provides near optimal performance but requires more consideration on objects lifetimes. The Direct collocation policy optimizations are not entirely compliant with the CORBA standard, though they provide more efficient collocated operation invocations. However, both collocation strategies are very efficient, compared with remote stubs that transmit data via the loopback network interface.

2.3 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 [21].

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 [14] 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 10, 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 mem-

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

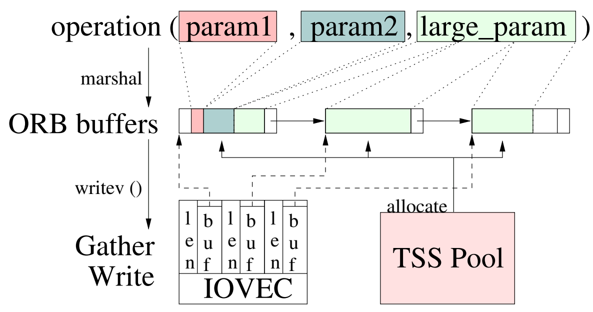

Figure 10: TAO's Internal Memory Management

ory management design also supports special allocators, such as zero-copy schemes [29] that share memory pools between user processes, the OS kernel, and network interfaces.

Empirical results: Figure 11 compares buffer allocation time for a CORBA request using thread-specific storage (TSS) allocators with that of using a global heap allocator. These experiments were executed on a Pentium II/450

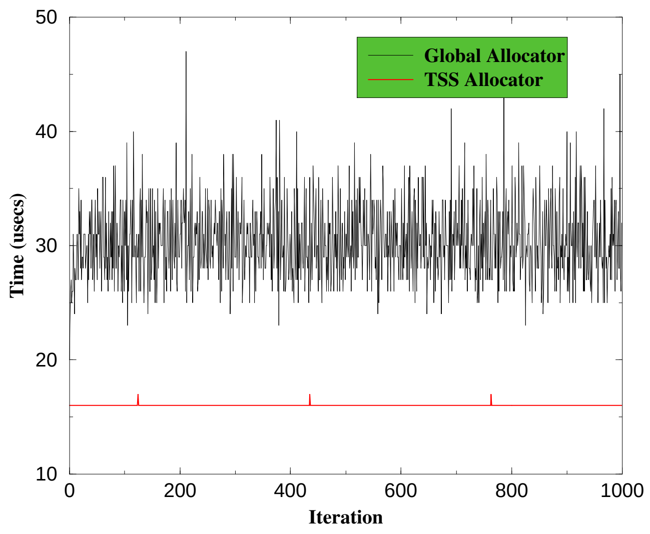

Figure 11: Buffer Allocation Time using TSS and Global Heap Allocators

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 11 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 [21].

2.4 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` operation. The fields removed by this optimization are shown in Table 2. These optimizations are guided by the principle patterns of relaxing system requirements and avoiding unnecessary generality.

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

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 2: 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 2. These experiments were executed on a Pentium II/450 with 256Mb of RAM, running LynxOS 3.0 in loopback mode. Table 3 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 3: 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 Axis 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 [30]. This framework generalizes TAO's current `-ORBgioplite` option to support both pluggable ORB protocols (ESIOPs) *and* pluggable transport protocols.

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 [31]. They also make it possible to (1) conserve resources by activating objects on-demand [32] and to (2) generate so-called persistent object references [33], 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 [8].

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.

3.2 Optimizing POA Demultiplexing

Scalable and predictable POA demultiplexing is important for many applications that have stringent hard real-time timing 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 12, 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

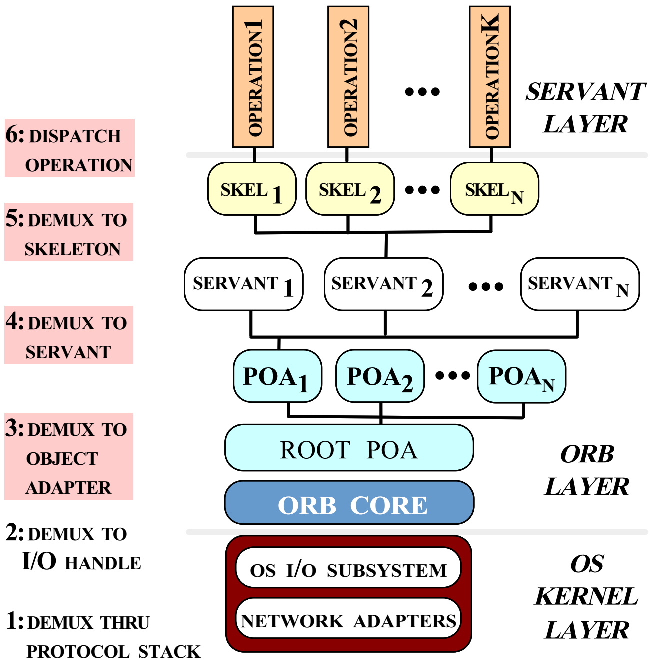

Figure 12: CORBA 2.2 Logical Server Architecture

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 12 is generally inappropriate for high-performance and real-time applications for the following reasons [34]:

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 [20].

Conventional implementations of CORBA incur significant demultiplexing overhead. For instance, [6, 8] 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 [35, 20].

3.2.2 Overview of Alternative Demultiplexing Strategies

As illustrated in Figure 12, 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 [6] 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 [8].

Perfect hashing: If the set of operations or servants is known *a priori*, dynamic hashing can be improved by pre-computing a collision-free *perfect hash function* [36]. Perfect Hashing is based on the optimization principle patterns of pre-computing 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 [8], *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* [8] 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. Passing hints is also an example of the Asynchronous Completion Token (ACT) design pattern [37].

Table 4 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 4: 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 12 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 [8] 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 13. The experiment was conducted on POAs 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 [14], Bridge, and Strategy patterns [3] to defer the configuration of the desired servant demultiplexing strategy until ORB initialization, which can be performed either *statically*

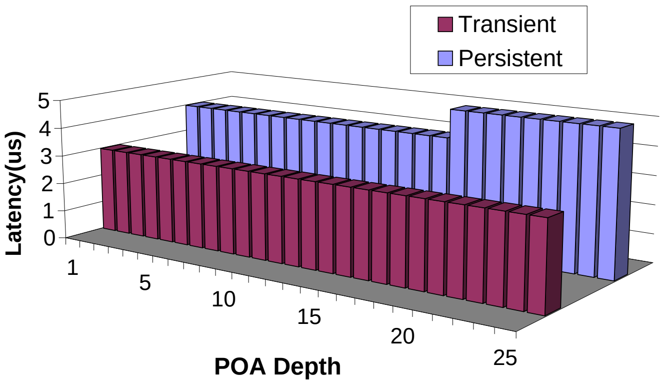

Figure 13: Effect of POA Depth on POA Demultiplexing Latency

at compile-time or *dynamically* at run-time. Figure 14 illustrates the class hierarchy of strategies that can be configured into TAO's POAs.

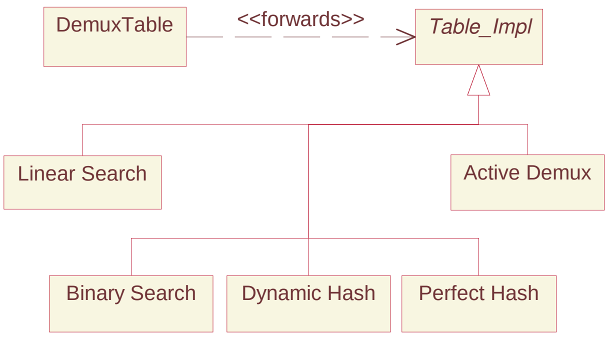

Figure 14: TAO's Class Hierarchy for POA Active Object Map Strategies

To evaluate the scalability of TAO, our experiments used a range of servants, 1 to 1,000 by increments of 100, in the server. Figure 15 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 priori* the set of servants in each POA for highly static systems,

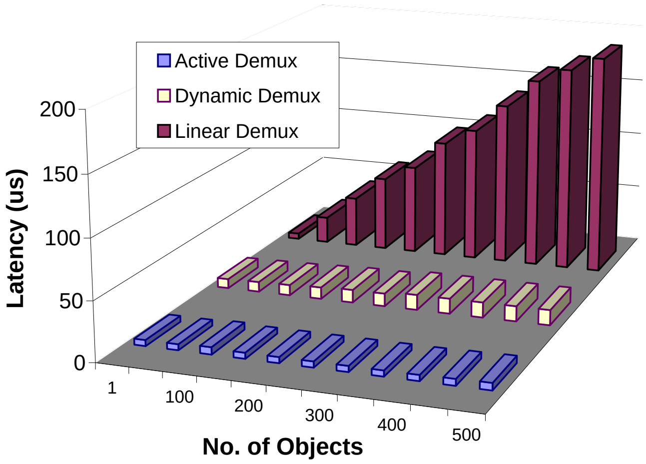

Figure 15: Servant Demultiplexing Latency with Alternative Search Techniques

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 [36], which is a freely available perfect hash function generator we developed to automatically construct perfect hash functions from user-supplied keyword lists.

Figure 16 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.

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]

[4] We are investigating modifications to the GIOP protocol for hard real-

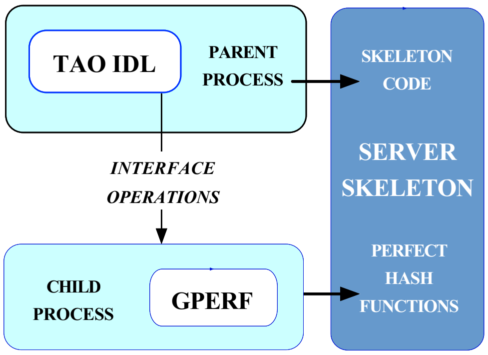

Figure 16: Integrating TAO's IDL Compiler and GPERF

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 17 plots operation demultiplexing latency as a function of the number of operations. This figure illustrates that

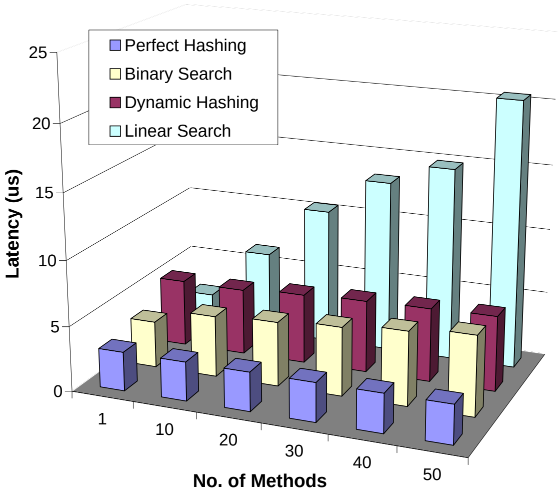

Figure 17: Operation Demultiplexing Latency with Alternative Search Techniques

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 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.

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 15, 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 [19], 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 18 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 lookups. However, this additional processing does not affect the critical path

time systems that possess stringent latency and message-footprint requirements.

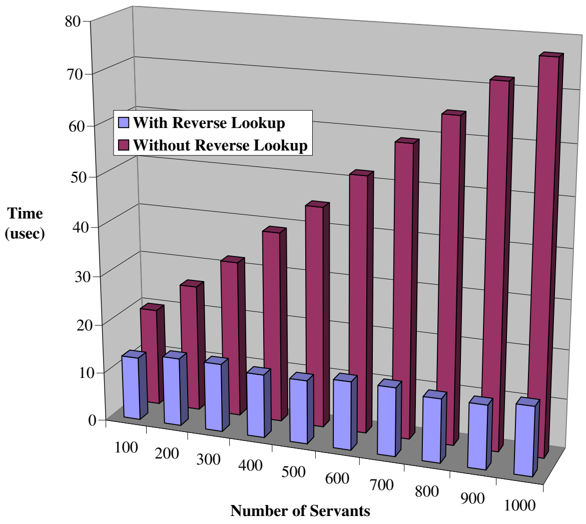

Figure 18: Benefits of Adding a Reverse-Lookup Map to the POA

of object ID lookups during run-time.

Summary of TAO's POA demultiplexing strategies: Based on the results of our benchmarks described above, Figure 19 summarizes the demultiplexing strategies that we have determined to be most appropriate for real-time applications [19]. Figure 19 shows the use of active demultiplexing for the POA names, active demultiplexing for the servants, and perfect hashing for the operation names. Table 5 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.

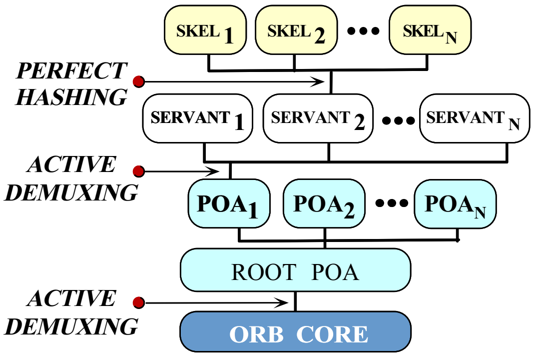

Figure 19: TAO's Default Demultiplexing Strategies

All of TAO's optimized demultiplexing strategies described above are entirely compliant with the CORBA specification.

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 5: Time Spent in Each Demultiplexing Step

Thus, no changes are required to the standard POA interfaces specified in CORBA specification [2].

3.3 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 20 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.

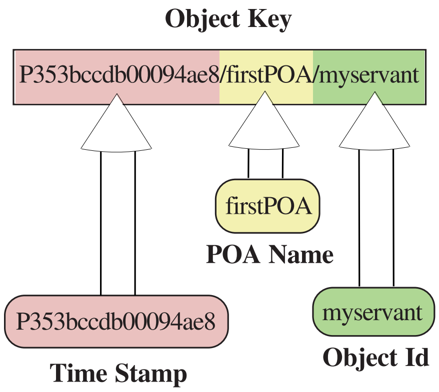

Figure 20: Naive Parsing of Object Keys

TAO's object key upcall optimizations: TAO provides the following object key optimizations based on the principle patterns of avoiding gratuitous 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 21. This eliminates wasteful memory allocations and data copies. This optimization is entirely compliant with the stan-

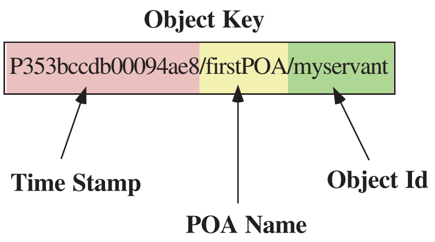

Figure 21: TAO's Optimized Parsing of Object Keys

dard CORBA specification.

3.4 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 [11].

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 [12], which removes the following features from the CORBA specification [2]:

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

Table 6 shows the footprint reduction achieved when the features listed above are excluded from TAO [5]. The measurements were taken for code compiled by the egcs compiler (version 2.91.60) on Solaris operating system (version 5.7). The options used for the compiler were (1) no debugging, (2) optimization was set to -O2, and (3) TAO was compiled into a static library. The 25.8% reduction in memory footprint for

Component	**CORBA**	**Minimum CORBA**	**Percentage Reduction**
POA	281.9	207.2	26.5
ORB Core	347.1	330.304	4.8
Dynamic Any	131.3	0.0	100
CDR Interpreter	68.7	68.7	0
IDL Compiler	10.5	10.5	0
Pluggable Protocols	14.6	14.6	0
Default Resources	7.9	7.9	0
Total	862.0	639.5	25.8

Table 6: Comparison of CORBA with Minimum CORBA Memory Footprint (in Kbytes)

Minimum CORBA is fairly significant. However, we plan to reduce the footprint of TAO even further by streamlining its CDR Interpreter [11]. 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.

4 Related Work

Real-time middleware is an emerging field of study. An increasing number of research efforts are focusing on designing and optimizing CORBA middleware to meet the requirements of real-time applications. This section outlines related work on concurrency and demultiplexing and compares it with the techniques applied in TAO.

There is a striking similarity between the TAO concurrency model and that recommended by Ousterhout [38]. To avoid the difficulties of threading at the application level, Ousterhout recommends an event-driven model for most applications. But for performance-critical kernel code, Ousterhout recommends

[5] The IDL Compiler row refers to the code required to collaborate between the IDL compiler and the ORB, and not to the code for the IDL compiler itself.

that threads be used in the kernel. If the TAO ORB Core and Object Adapter are viewed as the "kernel" then, because servants are application code, the TAO model corresponds with Ousterhout's recommendation.

Demultiplexing is an operation that routes messages through the layers of an ORB endsystem. Most protocol stack 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 12).

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, [34, 39, 35, 40] 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 [41]. A number of schemes to implement fast and efficient packet filters are available. These include the BSD Packet Filter (BPF) [42], the Mach Packet Filter (MPF) [43], PathFinder [44], demultiplexing based on automatic parsing [45], and the Dynamic Packet Filter (DPF) [40].

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 12).

4.1 Related Work on Optimization Principle Patterns

This section describes results from existing work on protocol optimization based on one or more of the principle patterns in Table 1.

4.1.1 Optimizing for the expected case

[46] describes a technique called *header prediction* that predicts the message header of incoming TCP packets. This technique is based on the observation that many members in the header remain constant between consecutive packets. This observation led to the creation of a template for the expected packet header. The optimizations reported in [46] are based on Principle Pattern 1, which *optimizes for the common case* and Principle Pattern 3, which is *precompute, if possible*.

4.1.2 Eliminating gratuitous waste

[47, 48, 49] describe the application of an optimization mechanism called *Integrated Layer Processing* (ILP). ILP is based on the observation that data manipulation loops that operate on the same protocol data are wasteful and expensive. The ILP mechanism integrates these loops into a smaller number of loops that perform all the protocol processing. The ILP optimization scheme is based on Principle Pattern 2, which *gets rid of gratuitous waste*. [49] cautions against improper use of ILP since this may increase processor cache misses.

4.1.3 Passing information between layers

Packet filters [42, 44, 40] are a classic example of Principle Pattern 6, which recommends *passing information between layers*. A packet filter demultiplexes incoming packets to the appropriate target application(s). Rather than having demultiplexing occur at every layer, each protocol layer passes certain information to the packet filter, which allows it to identify which packets are destined for which protocol layer.

4.1.4 Moving from generic to specialized functionality

[50] describes a facility called fast buffers (FBUFS). FBUFS combines virtual page remapping with shared virtual memory to reduce unnecessary data copying and achieve high throughput. This optimization is based on Principle Pattern 2, which focuses on *eliminating gratuitous waste* and Principle Pattern 3, which *replaces generic schemes with efficient, special purpose ones*.

4.1.5 Improving cache-affinity

[51] describes a scheme called "outlining" that when used improves processor cache effectiveness, thereby improving performance.

4.1.6 Efficient demultiplexing

Demultiplexing routes messages between different levels of functionality in layered communication protocol stacks. Most conventional communication 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 protocol stacks. In addition, conventional CORBA ORBs utilize several extra levels of demultiplexing at the application layer to associate incoming client requests with the appropriate servant and operation. Layered multiplexing and demultiplexing is generally disparaged for high-performance communication systems [34] due to the additional overhead incurred at each layer. [40] describes a fast and flexible message demultiplexing strategy based on dynamic code generation.

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 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 [52] or Java RMI [53], 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 7 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.

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

Optimization	**Principle Patterns**	**Compliant**
Concurrency	Optimize for common case Avoid gratuitous waste Not tied to reference models	yes
Collocation	Replace general-purpose operations with optimized special-purpose ones Optimize for common case Avoid gratuitous waste Add extra state	yes
Memory management	Exploit Locality Optimize for common case	yes
Protocol msg footprint	Replace general-purpose operations with optimized special-purpose ones Avoid unnecessary generality Relax system requirements	no
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
Object keys in upcalls	Avoid gratuitous waste Exploit locality	yes
Predictability and footprint	Relax system requirements	yes

Table 7: Degree of CORBA-compliance for Real-time Optimization Principle Patterns

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] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

[13] Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05 ed., May 1998.

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

[15] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6^{th} USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

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

[17] 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.

[18] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 2000.

[19] 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.

[20] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in *Proceedings of the 5^{th} IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), pp. 154–163, IEEE, June 1999.

[21] 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.

[22] 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.

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

[24] D. C. Schmidt, "Evaluating Architectures for Multi-threaded CORBA Object Request Brokers," *Communications of the ACM special issue on CORBA*, vol. 41, Oct. 1998.

[25] J. D. Salehi, J. F. Kurose, and D. Towsley, "The Effectiveness of Affinity-Based Scheduling in Multiprocessor Networking," in *IEEE INFOCOM*, (San Francisco, USA), IEEE Computer Society Press, Mar. 1996.

[26] 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 International Symposium on Distributed Objects and Applications (DOA'99)*, (Edinburgh, Scotland), OMG, Sept. 1999.

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

[28] 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.

[29] Z. D. Dittia, G. M. Parulkar, and J. R. Cox, Jr., "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, (Kobe, Japan), pp. 179–187, IEEE, April 1997.

[30] 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 *Proceedings of the IFIP 6^{th} International Workshop on Protocols For High-Speed Networks (PfHSN '99)*, (Salem, MA), IFIP, August 1999.

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

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

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

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

[35] Z. D. Dittia, J. R. Cox, Jr., 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.

[36] 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.

[37] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Asynchronous Completion Token: an Object Behavioral Pattern for Efficient Asynchronous Event Handling," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[38] J. Ousterhout, "Why Threads Are A Bad Idea (for most purposes)," in *USENIX Winter Technical Conference*, (San Diego, CA), USENIX, Jan. 1996.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine*, vol. 27, pp. 23–29, June 1989.

[47] D. D. Clark and D. L. Tennenhouse, “Architectural Considerations for a New Generation of Protocols,” in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 200–208, ACM, Sept. 1990.

[48] M. Abbott and L. Peterson, “Increasing Network Throughput by Integrating Protocol Layers,” *ACM Transactions on Networking*, vol. 1, October 1993.

[49] T. Braun and C. Diot, “Protocol Implementation Using Integrated Layer Processnig,” in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, ACM, September 1995.

[50] P. Druschel and L. L. Peterson, “Fbufs: A High-Bandwidth Cross-Domain Transfer Facility,” in *Proceedings of the* 14^{th} *Symposium on Operating System Principles (SOSP)*, Dec. 1993.

[51] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O’Malley, “Analysis of Techniques to Improve Protocol Processing Latency,” in *Proceedings of SIGCOMM ’96*, (Stanford, CA), pp. 73–84, ACM, August 1996.

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

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