Design Considerations and Performance Optimizations for Real-time ORBs

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

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

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 request/response applications with best-effort quality of service (QoS) requirements. However, supporting applications with more stringent QoS requirements poses new challenges for next-generation real-time CORBA middleware.

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 techniques needed to optimize the Object Adapter, ORB Core, and IDL compiler components. Second, we illustrate how TAO, our real-time CORBA implementation, has addressed these challenges. Third, we describe the results of benchmarks that compare the impact of alternative design strategies on the efficiency, predictability, and scalability of real-time ORBs.

Our results 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.

Keywords: CORBA Middleware, Design Patterns, Tools, Performance.

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.

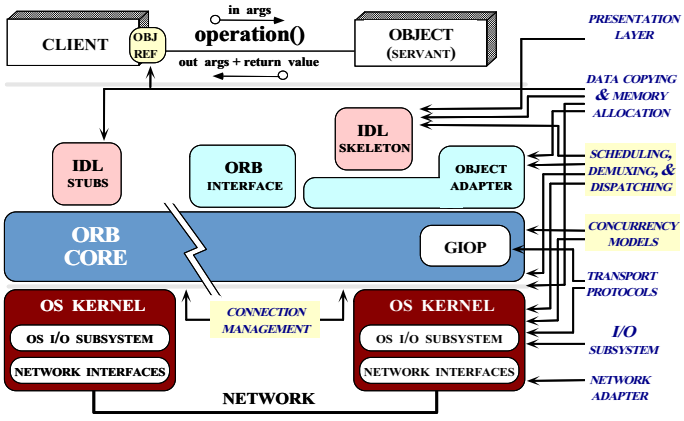

Figure 1: Real-time Features and Optimizations Necessary to Meet End-to-end QoS Requirements in ORB Endsystems

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

*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.

the ORB and its basic services, defined by the OMG specifications. In contrast, second-generation ORBs, such as The ACE ORB (TAO) [8], explicitly focus on providing end-to-end QoS guarantees to applications by *vertically integrating* highly optimized CORBA middleware with OS I/O subsystems, communication protocols, and network interfaces.

Our previous research has examined many dimensions of real-time ORB endsystem design, including static [8] and dynamic [9] real-time scheduling, real-time event processing [10], real-time I/O subsystems [11], real-time ORB Core connection and concurrency architectures [7], and performance comparisons of various commercial ORBs [4]. This paper focuses on three more dimensions in the real-time ORB endsystem design space: *Object Adapter demultiplexing optimizations, collocation optimizations, and IDL compiler stub/skeleton optimizations*, which are outlined below:

Request demultiplexing in the Object Adapter: The amount of time an ORB's Object Adapter spends demultiplexing requests to target objects, *i.e.*, servants, can constitute a key source of ORB overhead for real-time applications. Once a client request arrives at a CORBA server, the ORB Core must demultiplex it to the Object Adapter, which then demultiplexes it to the servant. The final step involves demultiplexing to a skeleton, which demarshals the request and dispatches the designated operation upcall in the servant.

Section 3 describes how an Object Adapter's demultiplexing strategies impact the scalability and predictability of a real-time ORB. Scalability is important for applications like real-time stock quote systems [12], which may service a large number of clients. Predictability is critical for hard real-time applications like avionics mission computing [10], which have stringent timing constraints.

Collocation optimizations: Another optimization supported by TAO minimizes 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. This mechanism elides all demultiplexing overhead by transforming operation invocations into local virtual method calls. Section 3.6 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.

Time and space optimizations for IDL stubs and skeletons: Next-generation distributed applications are being developed for a variety of heterogeneous platforms [13], ranging from embedded systems that have tight memory footprint constraints [14] to high-end multimedia servers with high bandwidth requirements [15]. In heterogeneous distributed systems, application data must be marshaled and demarshaled to ensure it is encoded and decoded correctly across different compilers and hardware platforms.

CORBA IDL compilers are responsible for generating *stubs* and *skeletons* that marshal and demarshal data types, respectively. There are two different forms of marshaling techniques: *compiled* and *interpreted*. In general, compiled marshaling generates larger, though more efficient, stubs and skeletons. Conversely, interpretive marshaling is generally less efficient than compiled marshaling, though it yields a smaller footprint, which is suited for memory-constrained embedded systems.

In general, hardcoding an IDL compiler to use either interpretive or compiled marshaling is too inflexible. [14] describes the design of TAO's IDL compiler, which can selectively generate compiled and/or interpreted stubs and skeletons, thereby maximizing optimization alternatives available to real-time CORBA application developers.

The remainder of this paper is organized as follows: Section 2 outlines the server-side architecture of CORBA ORBs, focusing on steps used to demultiplex incoming client requests. Section 3 describes the design and implementation of TAO's Portable Object Adapter (POA) and empirically compares the demultiplexing strategies; this section also describes TAO's collocation optimizations. Section 4 describes related work and Section 5 provides concluding remarks. For completeness, Appendix A outlines our real-time ORB testbed and the empirical methods used for our tests.

2 Server-side Architecture for CORBA ORBs

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

These new features allow application developers to write more flexible and portable CORBA servers. They also make it possible to conserve resources by activating objects on-demand [16] and to generate "persistent" object references [17] that remain valid after a server process terminates. Server applications can configure these features 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 process. This is particularly problematic for real-time applications since naive Object Adapter implementations can increase priority inversion and non-determinism [6].

The remainder of this section outlines the steps involved in demultiplexing a client request through the server-side of a CORBA ORB and qualitatively evaluates alternative demultiplexing strategies. Section 3 then quantitatively evaluates how these strategies perform in the TAO real-time ORB.

2.1 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

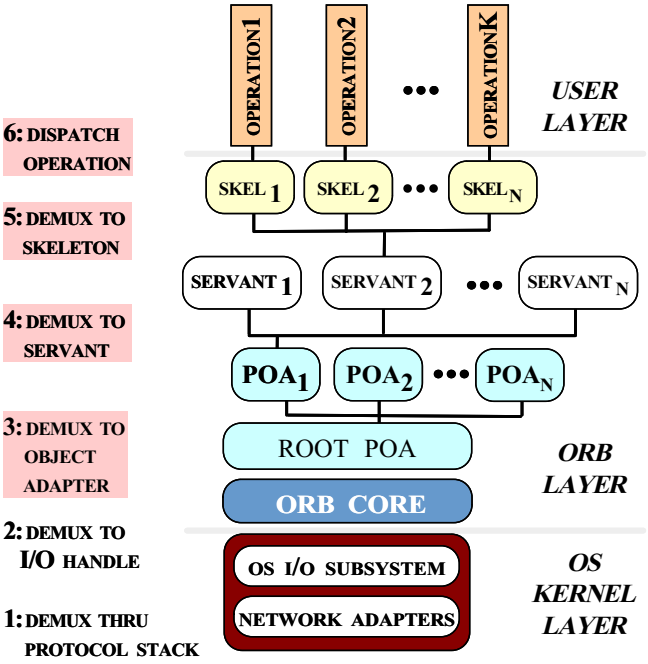

Figure 2: CORBA 2.2 Logical Server Architecture

endsystem must perform the following demultiplexing tasks:

Steps 1 and 2: The OS protocol stack demultiplexes the incoming client request multiple times, *e.g.*, from the network interface card, through the data link, network, and transport layers up to the user/kernel boundary (*e.g.*, the socket) and then dispatches the data to the ORB Core.

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 servant can involve multiple demultiplexing steps through the 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.

The conventional layered ORB endsystem demultiplexing implementation shown in Figure 2 is generally inappropriate for high-performance and real-time applications for the following reasons [18]:

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 an indeterminate period of time while lower priority packets are demultiplexed and dispatched [19].

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 middleware-level, *i.e.*, steps 3 to 6. Information on kernel-level demultiplexing optimizations for real-time ORB endsystems is available in [20, 21, 11].

2.2 General Demultiplexing Strategies

As illustrated in Figure 2, demultiplexing an operation on a servant involves several steps. Below, we outline the most common demultiplexing strategies used in CORBA ORBs. Section 3.1 empirically evaluates the strategies that are appropriate for each layer in the ORB.

Linear search: This strategy is examined largely to provide an upper-bound on worst-case performance, though some ORBs use linear search for operation demultiplexing [4]. If the number of operations on a given interface 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 large number of operations.

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 in practice. However, insertions and deletions can be complicated since data must be sorted in order for the binary search algorithm to work correctly. Therefore, binary search is particularly useful for operation demultiplexing since all insertions and sorting can be performed off-line by an IDL compiler. In contrast, using binary search to demultiplex 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 middleware behavior. Moreover, depending on the hash algorithm, dynamic hashing often has a fairly high constant overhead.

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* [22]. A demultiplexing strategy based on perfect hashing executes in constant time and space. This property makes perfect hashing well-suited for hard real-time systems that can be statically configured [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 as an index directly into a table managed by an Object Adapter. This so-called *active demultiplexing* [6] strategy offers a constant-time, low overhead approach that can be used in all layers of an Object Adapter.

3 The Design of TAO's Portable Object Adapter

Adapting the CORBA Portable Object Adapter (POA) specification to support real-time applications required us to resolve a number of design challenges. This section outlines these challenges and describes the optimizations we applied to maximize the predictability, performance, and scalability of TAO's POA. These optimizations include constant-time demultiplexing strategies, support for custom POA configurations, flexible synchronization policies in POAs, and minimizing dynamic memory allocations and data copies while demultiplexing a client request.

3.1 POA Demultiplexing

Section 2 describes the demultiplexing steps that 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 focuses on the strategies that TAO uses in each demultiplexing step.

3.1.1 POA Lookup

An ORB Core must locate the POA corresponding to an incoming client request. Figure 2 shows that POAs can be nested arbitrarily. These nesting semantics greatly complicate POA demultiplexing compared with the original 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.[1] We used a range of POA depths, 1 through 25. The results are shown in Figure 3.

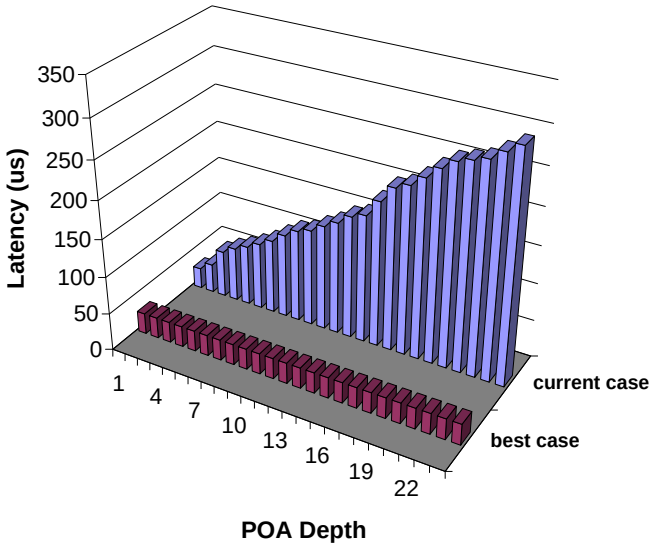

Figure 3: Effect of POA Depth on POA Demultiplexing Latency

[1] The hardware and software used in the benchmark testsuite are described in Appendix A.

Currently, TAO uses a straightforward lookup strategy where each POA find its child using dynamic hashing and then delegates on the child for the rest of the search. This naive strategy results in $O(n)$ growth for the lookup time and does not scale up to deeply nested POAs. Therefore, we are currently implementing active demultiplexing for the POA lookup phase, which will operate 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 may also be needed, *e.g.*, if the POA is to be 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, as described next in Section 3.1.2.

3.1.2 Servant Demultiplexing

Once the ORB Core demultiplexes a client request to the right POA, this POA must then demultiplex 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 [23], Bridge, and Strategy patterns [24] to defer the configuration of the desired servant demultiplexing strategy until ORB initialization [25]. Figure 4 illustrates the class hierarchy of strategies that can be configured into TAO's POAs.

To evaluate the scalability of TAO, our experiments used a range of servants, 1 to 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 and 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 reasonably sized system – 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 the set of servants on each POA for certain applications *a priori*, creating perfect hash functions repeatedly during application development is tedious. We omitted binary search for similar reasons, *i.e.*, it requires sorting the 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.

[2] Certain demultiplexing strategies are not appropriate for real-time ORBs and are shown mainly for comparison purposes.

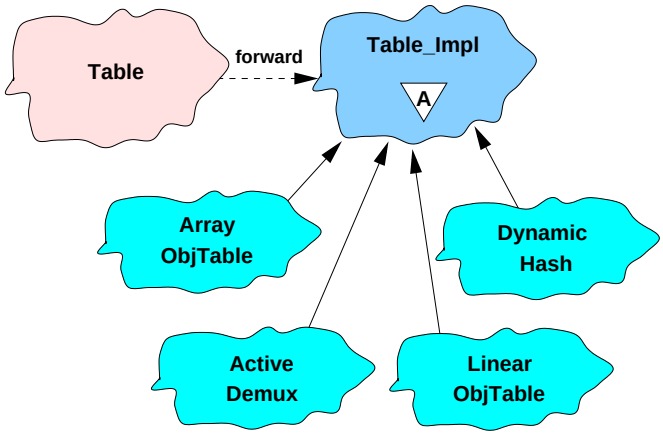

Figure 4: TAO's Class Hierarchy of POA Active Object Maps

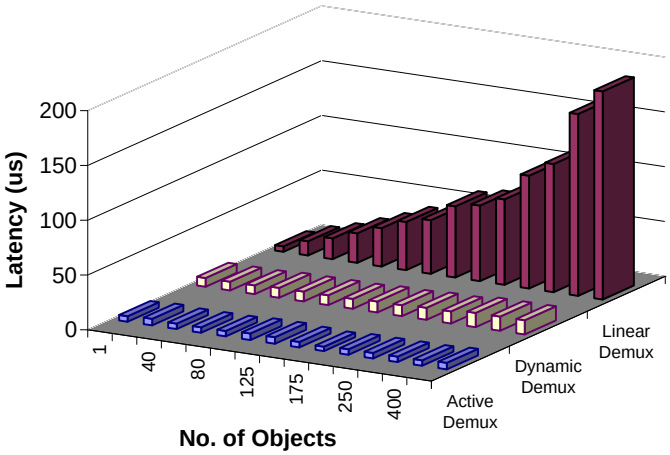

Figure 5: Servant Demultiplexing Latency with Alternative Search Techniques

3.1.3 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 over-

head, our experiments defined a range of operations, 1 through 100, in the IDL interface.

TAO's IDL compiler can be configured to generate skeletons that demultiplex operations using linear search, binary search, dynamic hashing, or perfect hashing. The lookup key for this phase is the operation name, which is a `string`. Therefore, it is infeasible to use active demultiplexing without modifying the GIOP protocol.[3] However, perfect hashing is well-suited to operation demultiplexing since all operations names are known *a priori*.

Figure 6 plots operation demultiplexing latency as a function of the number of operations. This figure indicates that perfect hashing behaves predictably and efficiently, outperforming dynamic hashing. As expected, linear and binary search depend on the number of operations, thereby complicating worst-case schedulability analysis for real-time applications.

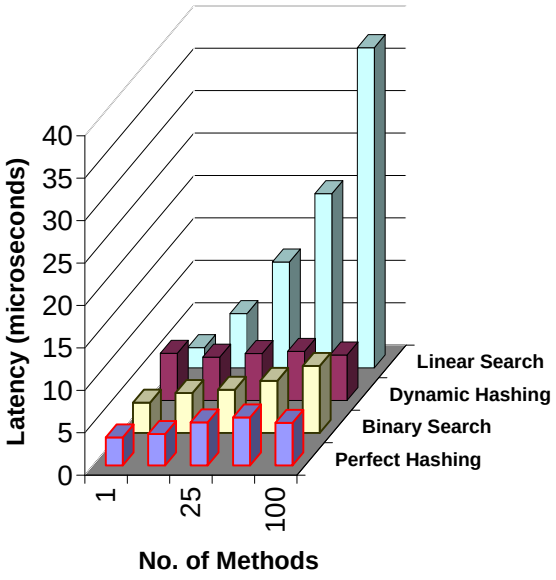

Figure 6: Operation Demultiplexing Latency with Alternative Search Techniques

3.2 Supporting Custom ORB Core and POA Configurations

To support real-time application QoS requirements, ORB middleware must be adaptable and configurable, as well as efficient, scalable, and predictable. To achieve these requirements, TAO supports several server configurations, including different ORB Core configurations that allow applications to customize request processing and transport connection management. For instance, TAO's ORB Core can be configured to process all requests in one thread, each request in a separate thread, or each connection in a separate thread [7].

To ensure consistent behavior throughout the layers in an ORB endsystem, TAO's POA is designed to support TAO's various ORB Core configurations. The important variations are (1) each ORB Core in a process has its own POA and (2) all ORB Cores in a process share one POA, as described below.

POA per ORB Core: Figure 7 shows the POA per ORB Core configuration, where each ORB Core in a server process maintains a distinct POA instance. This configuration is generally chosen for deterministic real-time applications, such as avionics mission computing [10], where each ORB Core has its own thread of control that runs at a distinct priority.

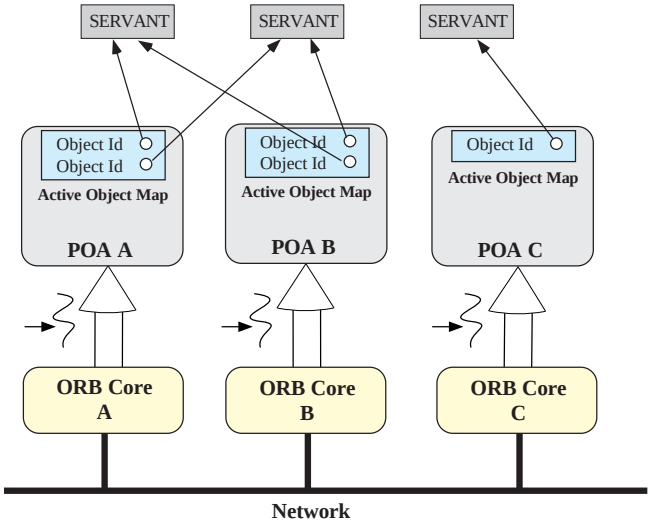

Figure 7: TAO's POA-per-ORB Core Configuration

When the POA per ORB Core configuration is used, each POA is accessed by only one thread in the process. Thus, no locking is required within a POA, thereby reducing the overhead and non-determinism incurred to demultiplex servant requests. However, the drawback of the POA per ORB Core configuration is that registering servants becomes more complicated if servants must be registered in multiple POAs.

Global POA: Figure 8 shows the Global POA configuration, where all ORB Core threads in a server process share the same POA. The main benefit of this configuration is that servant registration is straightforward since there is only one POA. However, the drawback is that this POA requires additional locks since it is shared by all the ORB Core threads in the process. These threads may simultaneously change the state of active object maps in the POA by adding and removing servants.

[3] We are investigating modifying the GIOP protocol for hard real-time systems with extremely stringent latency requirements.

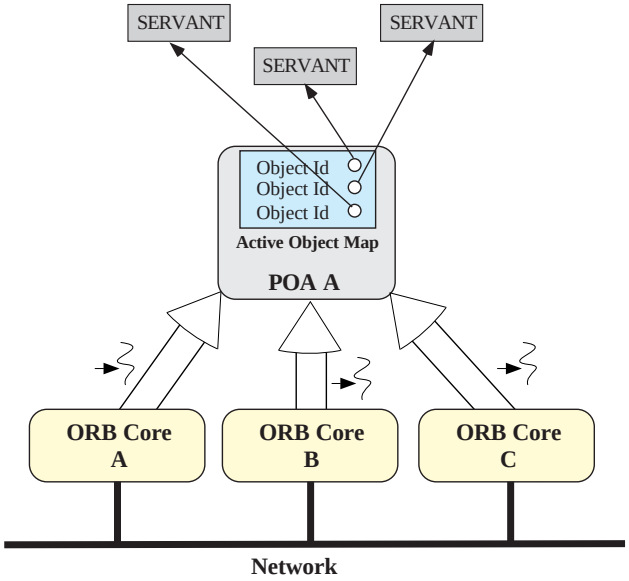

Figure 8: TAO's Global POA Configuration

3.3 POA Synchronization

To improve predictability and maximize performance, TAO minimizes synchronization in the critical request processing path of the ORB. Under certain ORB configurations, such as the POA per ORB Core configuration described in Section 3.2, no synchronization is required in a POA. Likewise, synchronization is unnecessary if POA state does not change during server execution. This situation can occur when all the servants and servant managers are registered at server startup and no dynamic registrations occur at run-time.

To enable applications to select the most efficient POA synchronization, TAO provides the following POA creation policy extensions:

```
// IDL
enum SynchronizationPolicyValue
{
NULL_LOCK, THREAD_LOCK, DEFAULT_LOCK
};

interface SynchronizationPolicy
: CORBA::Policy
{
readonly attribute
 SynchronizationPolicyValue value;
};

SynchronizationPolicy create_synchronization_policy
(in SynchronizationPolicyValue value);
```

Objects that support the `SynchronizationPolicy` interface can be obtained using the TAO's POA extension `create_synchronization_policy` operation. They are passed to the `POA::create_POA` operation to specify the synchronization policy used in the created POA. The `value` attribute of `SynchronizationPolicy` contains the value supplied to the `create_synchronization_policy` operation from which it was obtained. The following values can be supplied by server developers:

NULL_LOCK**:** No synchronization will be used to protect the internal state of the POA. This option should be used when the state of the created POA will not change during the execution of the server or when only one thread will use the POA.

THREAD_LOCK**:** The internal state of the POA will be protected against simultaneous changes from multiple threads. This option should be used when multiple threads will use the POA simultaneously.

DEFAULT_LOCK**:** The ORB run-time configuration file, `svc.conf`, will be consulted to determine whether to use a thread lock or null lock. This option should be used when the server programmer wants to delay the POA synchronization choice until ORB initialization at run-time.

If no `SynchronizationPolicy` object is passed to `create_POA`, the synchronization policy defaults to `DEFAULT_LOCK`. The `DEFAULT_LOCK` option allows applications to make the synchronization decision once for all the POAs created in the server. For example, if the server is single threaded, the application can specify in the `svc.conf` configuration file that the default lock should be the null lock. Hence, the application does not have to specify the `NULL_LOCK` policy in every call to `create_POA`.

Figure 9 shows the class hierarchy of the POA locks. The locking strategies used in TAO's POA are an example of the External Polymorphism pattern [26], where C++ classes unrelated by inheritance and/or having no virtual methods can be treated polymorphically.

3.4 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 the Active Object Map of the POA. Section 3.1.2 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.

Certain POA operations and policies require lookups on Active Object Map to be based on the *servant pointer* rather than

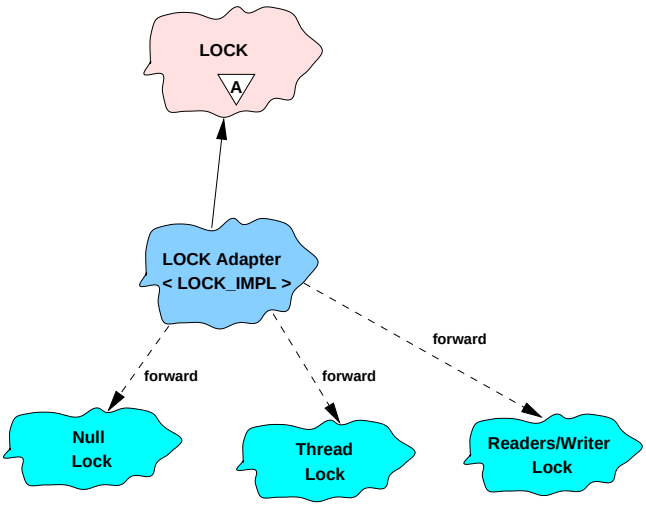

Figure 9: TAO's Class Hierarchy of POA Locks

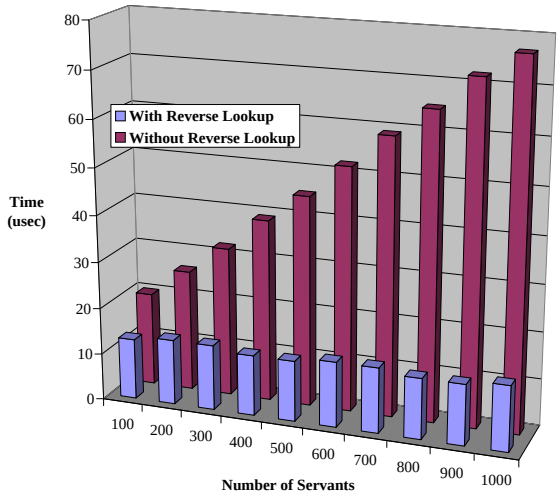

Figure 10: Benefits of Adding a Reverse-Lookup Map to the POA

the Object Id, however. For instance, the `_this` method[4] 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, a naive Active Demultiplexing implementation of a POA's Active Object Map, where the primary key is the Object Id, provides worst-case performance for servant-based lookups, which degenerate to linear search. Linear search becomes prohibitively expensive as the number of servants in the Active Object Map increases. This overhead is particularly problematic for server applications that create a large number of objects using `_this` during their initialization phase.

To alleviate this bottleneck, a POA can support a reverse-lookup map that associates each servant with its Object Id in $O(1)$ average-case time. This reverse map is used in conjunction with the existing lookup map that associates each Object Id to its servant. Figure 10 shows the time it takes to find a servant, with and without the reverse map, as the number of servants in the POA increases.

Servants are allocated from arbitrary memory locations. Sxince we don't have any control over the format of the pointer values, we use a hash map for the reverse-lookup map. The value of the servant pointer is used as the hash value. Although hash maps do not provide $O(1)$ worst-case behavior, they do provide a significant average-cast performance improvement over linear searching.

[4]The `_this` method is the most common way of creating object references in the POA model.

TAO's reverse-lookup map is an implementation detail of the Active Object Map used by the POA. It does not require any changes to the standard POA interfaces specified in CORBA specification [1]. Also, note that the reverse-lookup map can only be used with the `UNIQUE_ID` POA policy since with the `MULTIPLE_ID` POA policy, a servant may support many Object Ids. This, however, is not a shortcoming since servant based lookups are only required with the `UNIQUE_ID` policy.

The disadvantage of adding a reverse-lookup map to the POA is the increased overhead of maintaining an additional table in the POA. For every object activation and deactivation, two changes are now required in the Active Object Map: one to the reverse-lookup map and the other to the regular map that is used for Object Id based lookups. However, the additional reverse-lookup map does not effect the performance of Object Id based lookups.

3.5 POA Upcall Optimizations

Motivation: The POA is in the critical path of request processing in a server ORB. Therefore, TAO provides the following upcall optimizations that reduce run-time processing overhead and jitter in its Object Adapter layer. Figure 11 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 their respective objects. The problem with this approach is that it requires memory allocation for the individual objects and data copying to move the object key fields to the individual

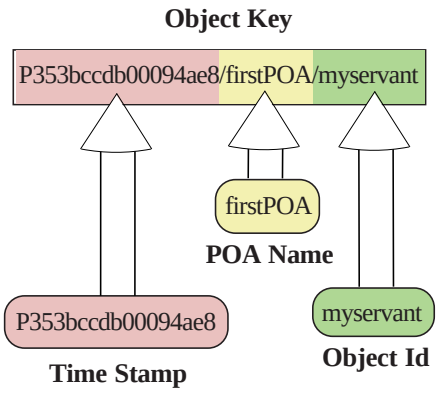

Figure 11: Naive Parsing of Object Keys

objects. These operations increase POA overhead.

TAO's optimization technique: Certain optimizations are possible during request demultiplexing in the POA. TAO takes advantage of the fact that the object key is available through the entire upcall and is not modified. Thus, the objects for the individual portions of the object key can be optimized to point to the correct locations in the object key. This approach is shown in Figure 12.

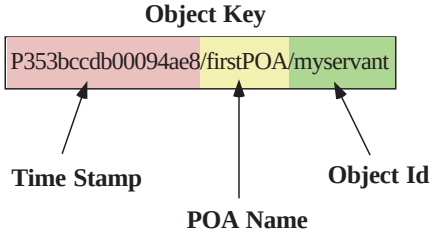

Figure 12: TAO's Optimized Parsing of Object Keys

Empirical results: The following table presents the time in microseconds (μs) spent during each activity TAO performs when demultiplexing a request:

Demultiplexing Stage	Absolute Time (μs)
1. *Request parsing*	6.89
2. *POA lookup*	21.11
3. *Servant dispatch*	5.00
4. *Operation demultiplexing*	5.55
5. *Parameter demarshal*	operation dependent
6. *User upcall*	servant dependent
7. *Return value marshal*	operation dependent

Each activity is outlined below:

1. The object key is parsed according to Figure 12 in the *request parsing* stage.
2. The *POA lookup* stage locates the POA where the servant resides. The time in this table is for a POA that is one level deep, although POAs can be many levels deep, in general.
3. The *servant dispatch* stage lookups a servant in the target POA. The time shown in the table for this stage is based on the active demultiplexing strategy.
4. The *operation demultiplexing* stage is where the skeleton associated with the operation resides. TAO uses the perfect hashing search strategy here.
5. The *parameter demarshal* stage is where the incoming request `in` and `inout` parameters are demarshaled. The overhead of this processing depends on the operation signature, *i.e.*, the complexity of the data parameters.
6. The time for the *user upcall* stage depends upon the actual implementation of the operation in the servant.
7. The *return value marshal* stage is where the `return`, `inout` and `out` parameters are marshaled. This time also depends on the signature of the operation.

The final three stages typically consume the most amount of time. Thus, the *request parsing* stage in TAO is a small percentage of its overall request dispatch time.

3.6 Collocation Optimizations

Motivation: 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 [27]. In this case, there is no need to incur the overhead of data marshaling or transmitting operations through a "loopback" transport device.

TAO's optimization technique: TAO's POA optimizes for collocated client/servant configurations by generating a special stub for the client. This stub forwards all requests to the servant. Figure 13 shows the classes produced by TAO's IDL compiler.

The stub and skeleton classes shown in Figure 13 are required by the POA specification; the collocation class is specific to TAO. This feature is entirely transparent since the client only uses 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 is in the same address space of the client.

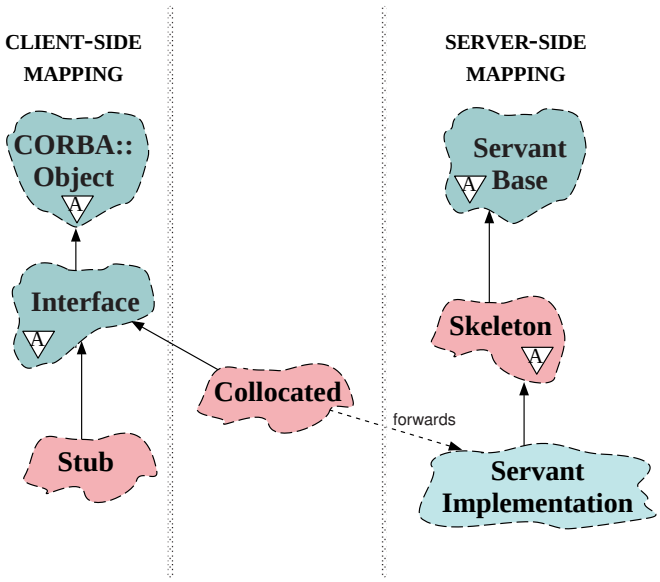

Figure 13: TAO's POA Mapping and Collocation Class

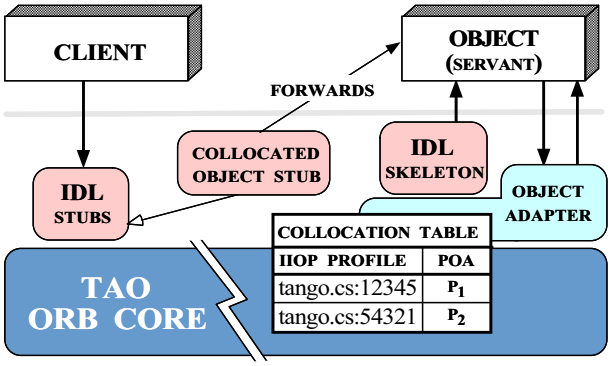

Figure 14: TAO's Collocation Table

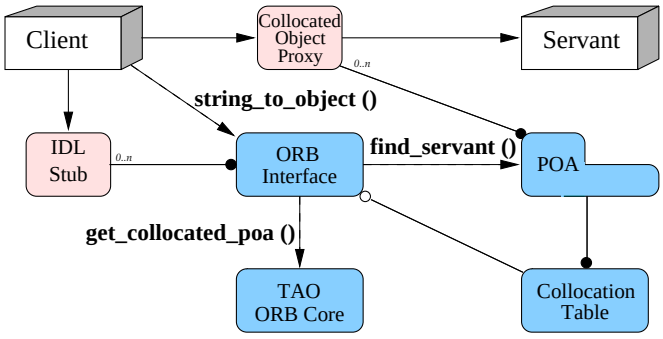

Figure 15: Finding a Collocated Object in TAO

Supporting transparent collocation in TAO: Clients can obtain an object reference in several ways. For instance, they can obtain object references from a CORBA Naming Service or from a factory operation. In addition, clients can use `string_to_object` to convert a stringified interoperable object reference (IOR) into an object reference. When a client tries to use this imported object reference, the ORB must automatically determine if the object reference refers to a collocated object.

As describe in Section 3.2, different POAs could be associated with different ORBs in a process. Therefore, to determine if an object reference is collocated, TAO maintains a *collocation table*. This table is a process-wide Singleton that maps ORB endpoints, *e.g.*, host name and port number tuples, onto to `Root POAs` used by ORBs, as shown in Figure 14.

Figure 15 demonstrates how collocation lookups are performed in TAO. When resolving an imported IOR using `string_to_object` or demarshaling an object reference, TAO examines the global collocation table to determine if the object is collocated or not. If a matching entry is found, TAO obtains a collocated stub from the associated POA and returns it to the client. If either operation fails, however, TAO simply returns a regular stub to the client. Thus, the process of selecting collocated stubs or regular stubs is completely transparent to clients.

In TAO, collocated operation invocations borrow the client's thread to run the servant's operation. Therefore, they are executed with the client thread's priority. This design is undesirable, however, for certain types of real-time applications [28] since priority inversion may occur when a client in a lower priority thread invokes operations on a collocated object in a higher priority ORB. To provide greater access control over which objects are collocated, TAO can be configured to use a collocation table in each ORB, which allows applications to control the scope of collocation optimizations.

Empirical results: To measure the performance gain from TAO's collocation optimizations, server and client threads were run 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 and without TAO's collocation optimizations to compare the performance gain.

Figure 16 shows the performance improvement measured by calls-per-second using TAO's collocation optimizations for operations that cube a variable-length sequence of `long`s containing 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 performance improves from 2,000% to 200,000% de-

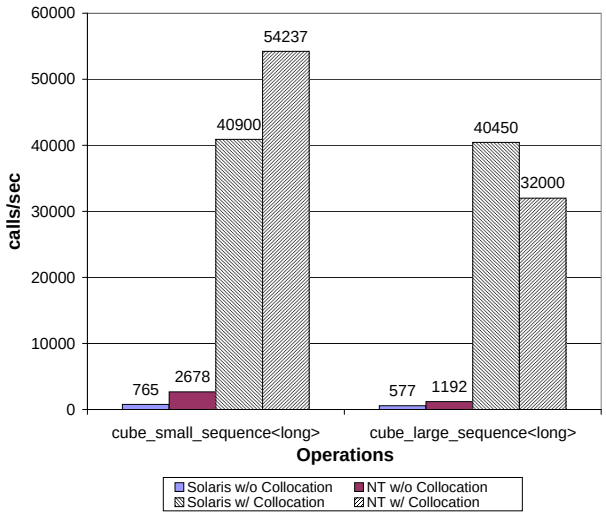

Figure 16: Results of Collocation Optimizations

Selecting strategies: In addition to avoid calling unpredictable operations, TAO must be configured to use predictable and scalable demultiplexing strategies. Figure 17 shows the demultiplexing strategies that we have determined to be most appropriate for real-time applications [10]. Figure 17 shows

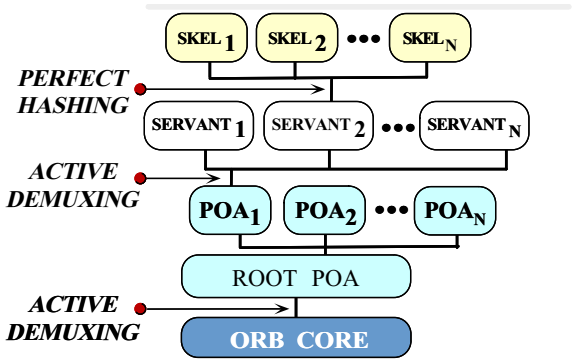

Figure 17: TAO's Default POA Strategies

pending on the size of arguments passed in the operation invocations.

3.7 Enhancing TAO's Predictability for Real-time Applications

To provide end-to-end predictability to real-time applications, ORBs must (1) avoid invoking non-deterministic operations and (2) select the appropriate demultiplexing strategies, as described below.

Omitting non-deterministic features: The following features of TAO's POA can be disabled 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.

In addition to increasing the predictability of POA request processing, omitting these features also decreases TAO's memory footprint.

the use of active demultiplexing for the POA names, active demultiplexing for the servants, and perfect hashing for the operation names. Our previous experience [29, 4, 30, 6, 7] measuring the performance of CORBA implementations showed TAO is more efficient and predictable than widely used conventional CORBA ORBs.

4 Related Work

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 (as shown in Figure 2). Layered multiplexing and demultiplexing is generally disparaged for high-performance communication systems [18] due to the additional overhead incurred at each layer. [31] describes a fast and flexible message demultiplexing strategy based on dynamic code generation. [29] evaluates the performance of alternative demultiplexing strategies for real-time CORBA.

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, [18, 32, 21] 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 [33]. A number of schemes to implement fast and efficient packet filters are available. These include the BSD Packet Filter (BPF) [34], the Mach Packet Filter (MPF) [35], PathFinder [36], demultiplexing based on automatic parsing [37], and the Dynamic Packet Filter (DPF) [31].

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).

Our framework uses a delayered demultiplexing architecture to select optimal demultiplexing strategies based on compile-time and run-time analysis of CORBA IDL interfaces.

5 Concluding Remarks

Developers of real-time systems are increasingly migrating towards the use of off-the-shelve middleware components to lower software lifecycle costs and decrease time-to-market. In this economic climate, the flexibility and adaptability offered by CORBA makes it an attractive middleware architecture. The optimizations and performance results presented in this paper underscore our belief that the next-generation of real-time CORBA ORBs will be well-suited for distributed real-time systems that require efficient, scalable, and predictable performance.

Since CORBA is not tightly coupled to a particular operating system or programming language, it can be adapted readily to "niche" markets, *e.g.*, the real-time embedded system market, which are not well covered by traditional major players such as Sun, Microsoft, or IBM. In this sense, CORBA has an advantage over other middleware technologies, such as DCOM [38] or Java RMI [39], since it can be integrated into a wider range of platforms.

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 [10]. In hard real-time systems, the ORB must meet deterministic QoS requirements to ensure proper overall system functioning. These requirements motivated 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.

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] 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.

[9] C. D. Gill, D. L. Levine, D. C. Schmidt, and F. Kuhns, "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.

[10] 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.

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

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

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

[14] A. Gokhale and D. C. Schmidt, "Techniques for Optimizing CORBA Middleware for Distributed Embedded Systems," in *Proceedings of INFOCOM '99*, Mar. 1999.

[15] S. Mungee, N. Surendran, and D. C. Schmidt, "The Design and Performance of a CORBA Audio/Video Streaming Service," in *Proceedings of the Hawaiian International Conference on System Sciences*, Jan. 1999.

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

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

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

[19] D. C. Schmidt, R. Bector, D. L. Levine, S. Mungee, and G. Parulkar, "An ORB Endsystem Architecture for Statically Scheduled Real-time Applications," in *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] C. Cleeland, D. C. Schmidt, and T. Harrison, "External Polymorphism – An Object Structural Pattern for Transparently Extending Concrete Data Types," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[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] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

A Experimental Setup

This section describes our benchmarking environment and empirical methods for measuring the impact of different demultiplexing strategies on CORBA request demultiplexing.

Hardware and software platforms: All experiments reported in this paper were conducted on two different hardware/software platforms, including:

- An UltraSPARC-II with two 300 MHz CPUs, a 512 Mbyte RAM, running SunOS 5.5.1, and C++ Workshop Compilers version 4.2;
- A Pentium Pro 200 with 128 Mbyte RAM running Windows NT 4.0 and the Microsoft Visual C++ 5.0 compiler;

Traffic generators: The experiments conducted for this paper extend our earlier studies [29] measuring the CORBA request demultiplexing overhead. Our prior work analyzed the impact of various object key and IDL skeleton demultiplexing techniques (such as linear search and direct demultiplexing). However, these measurements were conducted using the Basic Object Adapter (BOA) which did not involve the object adapter level demultiplexing that is necessary in the Portable Object Adapter (POA) model. In addition, the testbed was hand-crafted inside the ORB. This paper describes our experiments by in the context of the TAO ORB that allows us to configure different demultiplexing strategies without requiring any manual intervention. As in [29], we measure CORBA scalability by determining the performance impact of increasing the number of POAs, the objects in endsystem server processes, and operations in an interface.

Client-side request invocation strategies: Our experiments in Section 3 used two different invocation strategies for invoking different operations on the server objects. The two invocation strategies are:

- **Random request invocation strategy:** In this case, the client makes a request on a randomly chosen object reference for a randomly chosen operation. This strategy is useful to test the efficiency of the hashing-based strategy. In addition, it measures the average performance of the linear search based strategy. The algorithm for randomly sending client requests is shown below.

```
for (int i = 0; i < NUM_OBJECTS; i++) {
  for (int j = 0; j < NUM_OPERATIONS; j++) {
     choose an object at random from
        the set [0, NUM_OBJECTS - 1];
     choose an operation at random from
        the set [0, NUM_OPERATIONS - 1];
     invoke the operation on that object;
  }
}
```

- **Worst-case request invocation:** In this case, we choose the last operation of the last object. This strategy elicits the worst-case performance of the linear search strategy. The algorithm for sending the worse-case client requests is shown below:

```
for (int i = 0; i < NUM_OBJECTS; i++) {
  for (int j = 0; j < NUM_OPERATIONS; j++) {
     invoke the last operation on the
       last object
  }
}
```

Profiling tools: Detailed timing measurements used to compute latency were made with the `gethrtime` system call available on SunOS 5.5.1. This system call uses the SunOS 5.5.1 high-resolution timer, which expresses time in nanoseconds from an arbitrary time in the past. The time returned by `gethrtime` is very accurate since it does not drift.

The profile information for the empirical analysis was obtained using the `Quantify` performance measurement tool. `Quantify` analyzes performance bottlenecks and identifies sections of code that dominate execution time. Unlike traditional sampling-based profilers (such as the UNIX `gprof` tool), `Quantify` reports results without including its own overhead. In addition, `Quantify` measures the overhead of system calls and third-party libraries without requiring access to source code.