Enhancing Real-time CORBA via Real-time Java features

Arvind S. Krishna, Douglas C. Schmidt
Electrical Engineering and Computer Science
Vanderbilt University
Nashville, TN 37203, USA
Email: {arvindk, schmidt}@dre.vanderbilt.edu

Raymond Klefstad
Electrical Engineering and Computer Science
University of California, Irvine
Irvine, CA 92697, USA
Email: klefstad@uci.edu

***Abstract*— End-to-end middleware predictability is essential to support quality of service (QoS) capabilities needed by distributed real-time and embedded (DRE) applications. Real-time CORBA is a middleware standard that allows DRE applications to allocate, schedule, and control the QoS of CPU, memory, and networking resources. Existing Real-time CORBA solutions are implemented in C++, which is generally more complicated and error-prone to program than Java. The Real-Time Specification for Java (RTSJ) provides extensions that enable Java to be used for developing DRE systems. Real-time CORBA does not currently leverage key RTSJ features, such as scoped memory and real-time threads. Thus, integration of Real-Time CORBA and RTSJ is essential to ensure the predictability required for Java-based DRE applications.**

This paper provides the following contributions to the study of middleware for DRE applications. First we analyze the architecture of ZEN, our implementation of Real-time CORBA, identifying sources for the application of RTSJ features. Second, we describe how RTSJ features, such as scoped memory and real-time threads, can be associated with key ORB components to enhance the predictability of DRE applications using Real-time CORBA and the RTSJ. Third, we perform preliminary qualitative and quantitative analysis of predictability enhancements arising from our application of RTSJ features. Our results show that use of RTSJ features can considerably improve the predictability of DRE applications written using Real-time CORBA and Real-time Java.

Keywords. Distributed Real-time and Embedded Systems, Real-time CORBA, Real-time Java.

I. Introduction

DRE systems are becoming increasingly widespread and important. Common DRE systems include telecommunication networks (*e.g.*, call processing services), process automation (*e.g.*, hot rolling mills), and defense applications (*e.g.*, total ship computing environments). Real-time CORBA [1] is a rapidly maturing standard middleware technology that allows DRE applications to configure and control processor, communication and memory resources. Experience during the past five years [2], [3] demonstrates that Real-time CORBA has been successfully used to develop middleware for DRE applications, just as CORBA [4], and Java RMI [5] have been applied in the business and desktop domains.

Although the Real-time CORBA specification was integrated into the OMG standard in 1998, it has not been adopted universally by DRE application developers. A key barrier to adoption arises from the steep learning curve caused by the complexity of the CORBA-C++ mapping [6], [7]. To address this problem, the Java programming language has emerged as an attractive alternative. Since Java has less inherent and accidental complexity than C++, it is easier for application programmers to master it. Java also has other desirable language features, such as strong typing, dynamic class loading, reflection/introspection, and native support for concurrency and synchronization.

Conventional Java runtime systems and middleware have historically been unsuitable for DRE applications, however, due to

1) The under-specified scheduling semantics of Java threads, which can lead to the most eligible thread not always being run.
2) The ability of the Java Garbage Collector (GC) to preempt any other Java thread, which can yield very long preemption latencies.
3) The lack of a standard distributed computing model, which leads to *ad hoc* mechanisms for communicating between different nodes in a DRE application.

To address problems 1 and 2, the Real-time Java Experts Group has defined the RTSJ [8], which extends Java in several ways, including (1) new memory management models that allow access to physical memory and can be used in lieu of garbage collection and (2) stronger guarantees on thread semantics than in conventional Java. To address problem 3, we have implemented ZEN, which is an an open-source[1] RTSJ-based Real-time CORBA Object Request Broker (ORB) that combines the benefits of these two standard technologies. ZEN is ported to both the TimeSys RTSJ reference implementation [9] (which uses a virtual machine architecture) and jRate [10] (which uses an ahead-of-time compiler architecture).

ZEN is inspired by many of the patterns, techniques, and lessons learned in The ACE ORB (TAO) [11], which is our other open-source implementation of Real-time CORBA written in C++. Our prior published work on ZEN focused on (1) the extensible component architecture [12] of its ORB Core [13] and Object Adapter [14] layers and (2) the pre-

[1]The source code for ZEN can be downloaded from `www.zen.uci.edu`.

dictable demultiplexing strategies [15] that it uses to ensure $O(1)$ lookup time irrespective of the depth of the Object Adapter hierarchy. This paper extends our earlier published work by focusing on a previously unexplored dimension in real-time middleware: *the integration of RTSJ features to support Real-time CORBA*. Our results show that effective use of RTSJ features to implement Real-time CORBA can considerably improve the predictability of DRE applications written using Real-time CORBA and Java.

The remainder of this paper is organized as follows: Section II describes the main problems that arose while designing ZEN using conventional Java implementations, analyzes the critical request/response code path within ZEN to identify sources for the application of RTSJ features, illustrates how RTSJ features can be associated with key ORB components to enhance predictability, and empirically analyzes how the application of RTSJ features improved predictability; Section III summarizes how our work on ZEN relates to other research efforts; and Section III presents concluding remarks.

II. Enhancing the ZEN ORB using the RTSJ

The OMG Real-time CORBA specification was adopted several years before the RTSJ was standardized. Real-time CORBA's Java mapping therefore does not use any RTSJ features, such as `NoHeapRealtimeThread` and `ScopedMemory`. To have a predictable Java-based Real-time CORBA ORB like ZEN, however, it is necessary to take advantage of RTSJ features to reduce interference with the GC and ensure predictability.

This section first identifies problems in the original design of ZEN, which was initially based on regular (*i.e.*, non-RTSJ) Java. We then analyze a typical end-to-end critical code path of a CORBA request within the original ZEN ORB, which was based on regular Java. Based on this analysis, we describe how we are enhancing ZEN to use RTSJ features, such as real-time threads and scoped memory, to improve its predictability.

A. Problems in the Original Design of ZEN

In the original architecture of ZEN [13], key ORB components that are involved in request/response processing (such as acceptors, connectors, transports, and thread pools) were originally allocated in the heap. This architecture suffered from the following problem: *the ORB allocates several temporary objects during the processing of a remote request/response*. This allocation can lead to *demand garbage collection*, *i.e.*, execution of the GC when the Java `new` operator cannot find enough memory. Further, the Java Virtual Machine (JVM) also allocates heap memory as part of its execution which additionally entails garbage collection.

Execution of the GC can cause unbounded preemption latency to the thread processing the request. The situation is exacerbated if the request is critical (*i.e.*, highest priority), which can be catastrophic for certain types of safety- and mission-critical DRE applications. To eliminate priority inversions related to invocations of the garbage collector during a request upcall, it is essential that:

- ZEN avoids heap allocation by exploiting the RTSJ scoped and immortal memory
- Key ORB components be allocated either within scoped or immortal memory that would not cause demand garbage collection.

The aforementioned factors would minimize the interference with the GC enhancing the predictability of the ORB and DRE application.

Our redesign of ZEN for Real-time CORBA began by identifying the participants associated with processing a request at both the client and server sides. For each participant identified, we associated the component with non-heap regions and resolved challenges arising from this association. Allocating key ORB objects within scoped or immortal memory would not cause garbage collection, thus minimizing the interference with the GC and enhancing the predictability of the ORB and DRE application.

B. Applying RTSJ Features to ZEN

a) RTSJ application goals: The goals for applying RTSJ features to ZEN include:

- **Minimizing interference with GC.** Garbage collection is generally considered to be unsuitable for DRE applications with stringent real-time requirements. Although there have been recent advances in GC algorithms [16], the Java GC can preempt any thread in the system, leading to the thread incurring unacceptably long preemption latencies. A key goal of ZEN is therefore to avoid allocating critical ORB components in heap memory to reduce the number of GC sweeps.
- **Compliance with the CORBA specification.** To preserve compliance with the Real-time CORBA specification, the RTSJ features must be incorporated within ZEN's ORB Core and Object Adapter layers without requiring any modifications to the Real-time CORBA specification. Options that require the end-user to be RTSJ-aware, such as associating scoped memory at the POA level, are provided as non-standard ZEN-specific options.
- **Interoperability with non RTSJ Java.** ZEN is designed to use intelligent strategies for component creation and extensibility [12] that allow configurability of real-time features (such as the number of static/dynamic threads, thread priorities, and buffer size) using properties and policies. These strategies use techniques, such as reflection [17], [18] and aspects [19], to create real-time/vanilla Java components, thereby minimizing time/space overhead for applications that do not require real-time features.

b) Identification of steps: Our redesign of ZEN for Real-time CORBA began by identifying the participants associated with processing a request at both the client and server sides. For each participant identified, we associated the component with non-heap regions and resolved challenges arising from this association.

The first step needed to identify where to apply RTSJ features required the analysis of the end-to-end critical code path in ZEN. Figure 1 depicts the participants involved in servicing a CORBA two-way invocation. The discussion of

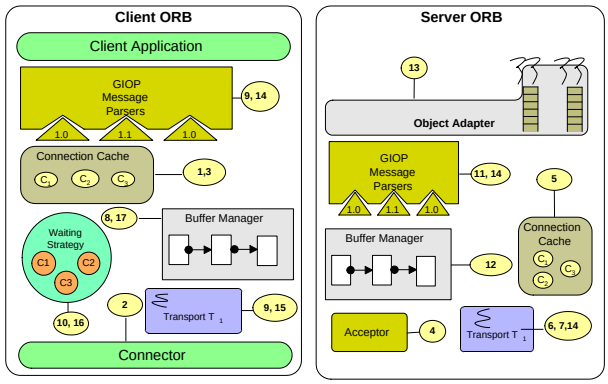

Fig. 1. Tracing an Invocation Through the ZEN CORBA ORB

the critical code path has been generalized using the Acceptor-Connector [20] pattern and thread-per-connection concurrency strategy.

We next describe the sequence of steps a client ORB performs to actively create a connection when a CORBA request is invoked by the application, *i.e.*, `result = object.operation (arg)`.

c) Connection management: We first describe how ZEN establishes a connection between a CORBA client and a server.

1. The client ORB's connection cache (`ConnectorRegistry` class in ZEN) is queried for an existing connection to the server, obtained from the object reference on which the operation is invoked.
2. If no previous connection exists, a separate connection handler is created (`Transport` class in ZEN) T_1 and the Connector connects to the server
3. This connection is added to the `ConnectorRegistry` since C_1 is bidirectional.

The activities of the server ORB for accepting a connection are described next:

4. An acceptor accepts the new incoming connection.
5. This connection C_1 is then added to the server's connection cache (`AcceptorRegistry` class in ZEN) as the server may send requests to the client.
6. A new connection handler T_1 is created to service requests.
7. The Transport's event loop waits for data events from the client.

d) Synchronous request/reply processing: The following are the steps involved when a client invokes a synchronous two-way request to the server.

8. The `BufferManager` class is queried to obtain a buffer to marshal the parameters in the operation invocation.
9. The appropriate GIOP Message Writer marshals the request and the Transport sends the request to the server.
10. The `WaitingStrategy` class associated with the transport waits for a reply from the server.

The server ORB performs the following activities to process the request.

11. The request header on connection C_1 is read to determine the size of the request.
12. A buffer of the corresponding size is obtained from the buffer manager to hold the request and the request data is read into the buffer.
13. The request is the demultiplexed to obtain the target POA, servant, and skeleton servicing the request. The upcall is dispatched to the servant after demarshaling the request.
14. The reply is marshaled using the corresponding GIOP message writer; Transport sends reply to the client.

The client ORB performs the following activities to process the reply from the server:

15. The Reader reads the reply from the server on the connection.
16. Using the request ID, the Waiting Strategy identifies the target Transport.
17. The parameters are then demarshaled and control is returned to the client application, which processes the reply.

e) Analyzing request processing steps: The request processing steps described above reveal the following characteristics:

- **Repetitive.** The steps involved with request/reply processing are repetitive, *i.e.*, carried out for every request. Steps 11-14 at the server side for request processing and steps 15-17 at the client side remain the same for each request from the client/server. Similarly, steps 1-3 are performed for every remote request sent to the server.
- **Independent and memoryless.** Steps required for processing request/response from two different client/server(s) are independent, *i.e.*, they do not share any context. Moreover, two requests from the same client do not share any context.
- **Ephemeral.** The objects created during the execution of these steps remain valid only for the duration of *one* cycle of request/response processing. ORBs therefore usually cache these resources to minimize resource management.
- **Thread bound.** Each of the steps are executed by a request processing thread. For example, steps 11-14 at the server side are executed by the transport and thread-pool threads.

The aforementioned characteristics of the steps lend themselves to the application of RTSJ features in the following manner:

- **Associating Real-time threads.** The thread-bound property of the steps enables components *e.g.*, acceptor-connector and transports to be associated with real-time threads. In particular, each of these components is designed based on an *logic* part, implemented as a Java class that implements the `Runnable` interface. This

part is then associated with a scoped memory region and bound with the thread at creation time.

- **Associating Scoped memory.** The ephemeral property of the steps enable internal objects created during request/response processing to be associated with scoped memory regions.

The repetitive, independence, and memoryless properties of the steps further shape how an ORB implementor can associate scoped memory. The *repetitive and memoryless* properties enable the request/response processing steps to be carried out within a scoped memory region[2], process the request, and send the response to the client. The memory region is then exited[3] enabling all the objects created to be freed, thus minimizing the number of GC sweeps. This cycle is repeated for the next request. The *independence property* validates the above mechanism, allowing objects created during request processing to be freed before processing a subsequent request.

Unlike heap/immortal memory, creation of scoped memory regions requires the size of the memory region to be specified at creation time. However, the footprint required to process request/response is dynamic, *i.e.*, varies based on:

- **Request size.** The request size at the server depends on the size of the request sent by the client.
- **Options associated.** The footprint required during request processing depends on the options enabled.
- **Type of Request.** The request size directly depends on the type of GIOP request *e.g.*, a LOCATE_REQUEST message would be of a different size when compared to a normal request.

The most appropriate memory size would therefore have to be chosen during initialization time. One solution to this problem is to create the one huge chunk of memory. However, this solution is non scalable. Further, some JVMs may not be able to allocate a huge chunk of scoped memory region. To address this problem, in ZEN we use *Nested Scopes* i.e. inner scopes, for every request/response demultiplexing phase, which is explained in Section II-C.

C. Applying Scoped Memory within ZEN

To enhance predictability, we apply RTSJ features *e.g.*, scoped memory to ORB components along the critical request/response processing path. Moreover, to minimize the effect of pre-allocating memory regions, we use nested scope memory regions for each demultiplexing phase. Below, we explain the three broad phases of request processing, at the server ORB and describe how we associate scoped memory with each of the three phases. Similar condition exist at the client ORB.

1. I/O layer

- **Steps.** This phase of demultiplexing corresponds to the steps 4-7 described in Section II-B.
- **Participants.** The participants for this phase include acceptors, connectors, and transports.
- **RTSJ application.** Each of these components are thread-bound components and are designed based on the *inner class* paradigm. This class derives from the `Runnable` interface and corresponds to the logic run by the thread. Instead of creating the entire component in scoped memory, we create the inner logic class in a scoped memory region, $m_{I/O}$. This logic class is associated with the thread at creation time. During ORB execution, multiple clients may connect to it, creating transports for every active client. Each of the transports will have a dedicated $m_{I/O}$ region. We collectively refer to these regions as a *space*.

2. ORB Core layer

- **Steps.** This phase of demultiplexing corresponds to the Steps 11-12 in Section II-B.
- **Participants.** GIOP Message parsers, Buffer Allocators and CDR Streams.
- **RTSJ application.** On receipt of new data events from the socket, the Transport reads the message header from the stream. Based on the size of the header, a `RequestMessage`[4] is created. After reading the request from the stream, the appropriate message parser is associated based on the type of the request. The message parser and the RequestMessage buffer are created in a nested memory region, m_{ORB}. The *ORB space* is a nested memory region. Based on RTSJ memory rules, references from the ORB to the I/O space are valid, *i.e.*, every m_{ORB} scope may hold references to the corresponding $m_{I/O}$ region.

3. POA layer

- **Steps.** This phase of demultiplexing corresponds to the steps 13-14 in Section II-B.
- **Participants.** Upcall objects[5], and thread-pools
- **RTSJ application.** The message parser parses the request to find the target POA and servant. An `Upcall` object is created to hold all information necessary to perform the upcall on the skeleton. A worker thread in the thread-pool then performs the upcall. A `CDROutputStream` is created, to hold the response, which is then sent to the client. The `Upcall` objects and the output buffers are created in a nested scoped memory region m_{POA}. The *POA space* is the innermost memory region. Again, references from POA to ORB or I/O space are valid.

4. Application layer

The application layer typically corresponds to the region where servant IDL skeletons are created. In the current design of ZEN, this application layer is *heap allocated*. Thus in our architecture, a `NoHeapRealtimeThread` cannot be used for request processing. The use of `NoHeapRealtimeThread` requires the

[2]Using the `enter()` method the memory region can be made the current allocation context.

[3]Exiting a memory region is implicit, done after the completion of the `run` method.

[4]This class encapsulates a buffer to hold the request.

[5]Upcall objects are *per request* objects that hold context necessary to perform upcall and send response.

application developer to be RTSJ aware and also entails modifications to the Java mapping of Real-time CORBA specification which in *conflict* with our goals. The use of a `NoHeapRealtimeThread`, however, is critical to enhancing the predictability of a Real-time CORBA ORB. In ZEN, we plan to provide policies at the POA level that would determine the type of real-time thread to used for request processing. Thus an RTSJ aware server application can allocate servants in a memory region other than the heap and set the type of upcall processing thread to NHRT to enhance predictability.

ZEN's architecture does not violate any of the RTSJ reference rules as (1) any of the ORB layers may hold references to the application layer and (2) a real-time thread can always allocate from Heap memory (enter it) without violating the single-parent rule. Moreover, our application is compliant with the CORBA specification.

Figure 2 (A) illustrates the nesting of scopes within the ZEN ORB Core. The I/O space is the outermost memory region while the POA layer is the innermost. Memory regions are entered from outer →inner, while references are maintained from inner → outer. On completion of a request, the memory regions are exited from innermost to outermost. All the objects thus created for request processing are finalized, thereby minimizing interference with the GC. Figure 2(B) depicts the parenthood tree of the memory regions in ZEN.

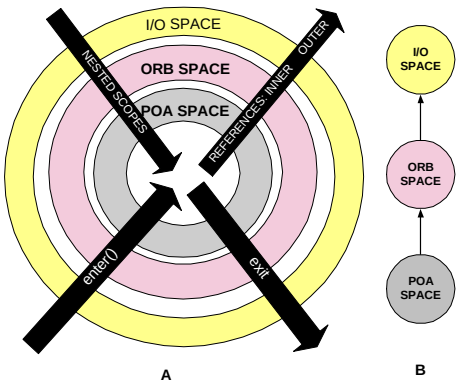

Fig. 2. Scope Nesting in ZEN ORB

D. Empirically Evaluating the Application of RTSJ in ZEN

To measure the predictability improvements accrued by using RTSJ features, we first associated scoped memory and real-time threads within the innermost scope *i.e.*, POA layer, allocating both I/O and ORB scopes in heap memory. The motivation for this application was to obtain a baseline for improvement in predictability when applying RTSJ features in just one layer. In this regard, applying RTSJ features in either the ORB or the I/O scopes alone would violate RTSJ memory reference rules making our application the only possible option.

Although ZEN supports a variety of options, we make the following assumptions for this analysis:

1) Portable interceptors are not considered in request processing.
2) The sequence of steps analyzed is for a remote client request, not a collocated request.
3) Servants are normal CORBA servants that inherit from `org.omg.PortableServer.Servant`, *i.e.*, we do not consider DII and DSI.
4) ZEN's buffer manager was disabled to prevent caching of I/O buffers
5) No proprietary policies are used in the ORB and
6) The scoped memory used was of type `LTMemory`, which is a RTSJ memory region with linear allocation time with respect to object size.[6]

These assumptions are representative of ways in which DRE applications commonly apply ORB middleware. All the experiments in this section were performed on an Intel Pentium III 864 Mhz processor with 256 MB of main memory. For these experiments, ZEN version 0.8 was compiled using the GNU `gcj` compiler version 3.2.1 and executed using jRate 0.3a on Linux 2.4.7-timesys-3.1.214 kernel. For each experiment, a sample size of 50,000 data points was used for result analysis.

The term predictability has different connotations in different disciplines. For example, in real-time scheduling theory, a predictable system means that each task always meets its deadline. For these experiments, we define *predictability* as the measure of standard deviation of the data points.

1) Demultiplexing Predictability:

a) **Test overview:** The following are the demultiplexing stages present in the POA layer:

1) *POA demultiplexing* – using the addressing information in the object key, locate the target POA in the POA hierarchy;
2) *Servant demultiplexing* – from the *object id* part of the object key, locate the servant within the POA; and
3) *Operation demultiplexing* – finally using the operation name, dispatch the upcall on the skeleton.

The predictability of the ORB depends directly on the predictability of the aforementioned demultiplexing stages. As discussed in [14], ZEN uses active demultiplexing along with perfect hashing to ensure $O(1)$ lookup time for all cases. In this experiment, we focus on the POA demultiplexing stage and analyze the predictability improvements accrued by associating scoped memory with ZEN's Object Adapter.

b) **Test settings:** To measure the variation in POA demultiplexing time with the depth of the POA hierarchy, the experiment increased the nesting of transient POAs from 1 to 150 in increments of 25. For each case the time to reach the leaf POA was measured.

c) **Analysis of results:** We now analyze the results of benchmarks that measure the average latency, the dispersion

[6]This bound does not include the time taken by an object's constructor or a class's static initializers.

and worst case behavior for the various POA demultiplexing test cases.

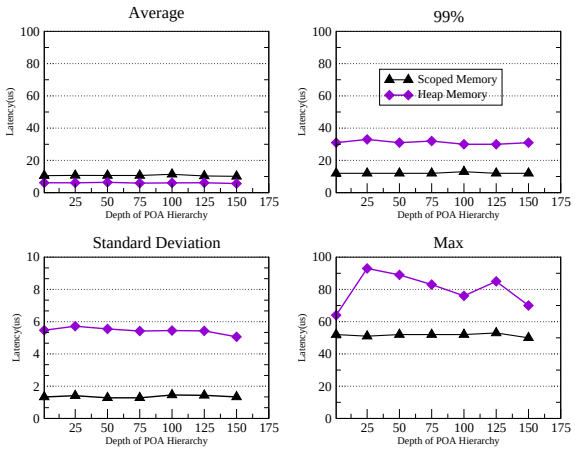

Fig. 3. POA Demultiplexing Predictability Analysis

- **Average measures.** Figure 3 shows that average latency for both heap and scoped memory does not increase with an increase in the POA hierarchy. However, the average measures for the scoped ($\sim$ 10 μsecs) are more than that of heap memory ($\sim$ 7 μsecs) by $\sim$ 3 μsecs. Although, this represents a increase in the POA location overhead by a factor of $\sim$1.4, predictability improves considerably.
- **Dispersion measures.** The dispersion of scoped memory is smaller than that of heap memory for all cases. Moreover, the measures are better by a factor of $\sim$ 4, indicating that our association scoped memory has considerably improved predictability for this stage of demultiplexing.
- **Worst case measures.** The 99% bound shows that using scoped memory 99% of the values are less than $\sim$12 μsecs for all cases while those for heap are $\sim$30 μsecs,a factor of 2 improvement. Maximum measures show that scoped memory considerably improves the predictability for all cases by bounding the worst case values. The worst case measures for scoped memory are tighter across the POA hierarchy. Moreover, the maximum latency for scoped memory is nearly constant across the POA hierarchy, while the latency of heap incurred measurable variability.

From the analysis above, it is evident that the use of scoped memory enhances demultiplexing predictability by bounding dispersion and maximum measures. Moreover, the use of scoped memory does not significantly degrade the average demultiplexing latency.

2) Scoped Memory Overhead:

a) **Test overview:** The application of scoped memory at ZEN's Object Adapter layer involves the following additional steps:

1) The request processing thread needs to make the scoped memory region its current allocation context by explicitly entering it (using `enter()` method defined on the region), incrementing the reference count of the region.
2) After processing the request, the thread exits the region, which changes the allocation context to the enclosing memory region or the primordial scope (heap), in turn decrementing reference count of the region.
3) If the reference count is zero, then all the objects allocated within the region are to be finalized.

The steps described above *must* be executed for every client request. Thus, the overhead and predictability involved in the steps directly affect the predictability of the ORB. In ZEN, the size of the scoped memory region is pre-allocated at initialization time. However, as explained earlier, the foot-print at the server varies depending on the message size, options associated and the type of request. In this experiment, we analyzed the behavior of the enter and exit methods as the request sizes increase.

b) **Test settings:** The scoped memory region size was fixed at 512KB. A client then issued requests starting at 1K in increments of 25KB up to a maximum request size of 100K.

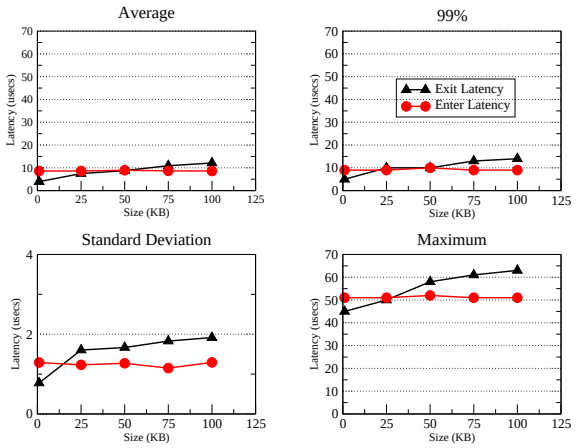

Fig. 4. Enter and Exit Time Analysis

c) **Analysis of results:** We now analyze the results of benchmarks that measure the average latency, the dispersion, and the worst case behavior of the memory region's `enter()` method and its exit time.

- **Average measures.** Figure 4 shows that the average additional overhead incurred by the `enter()` method is $\sim$ 8 μsecs and is constant across all request message sizes. However, the latency of the memory region's exit time increase gradually with the request size. For example, the average exit time for a 1K message is $\sim$ 4μsecs and increases to $\sim$12 μsecs for a 100K message. This behavior occurs because finalizes for every allocated object in the scoped memory region are called before exiting the region, in which case the size of messages increase the finalization time.
- **Dispersion measures.** The dispersion latencies for the `enter()` method show a trend similar to the average – constant ($\sim$ 1-1.5) across all message sizes. However, the dispersion for the exit time increases with the request size. For example, dispersion for 1K message is 0.7 and

increases to 1.9 for 100K. This behavior is induced by the finalization required.

- **Worst case measures.** The 99% bound for both the enter and exit latencies follows a trend similar to the respective average cases. Moreover, the values are closer to the average measures. However, the worst case measures for exit values are higher than those of the enter method and increase with message sizes.

The analysis above reveals that the average overhead incurred due to the `enter()` method is $\sim 8\mu$secs and is constant across all request sizes. However, the behavior of the exit method varies with the request size, *i.e.*, the larger the request size, the greater the exit latency and worse the dispersion.

3) Round-trip Delay:

a) **Test overview:** This experiment analyzes the effect of scoped memory on round-trip delay. Since we associate scoped memory at ZEN's Object Adapter layer, the only factor affecting round-trip delay is the request execution time. These experiments measure the effect of using scoped memory in the critical end-to-end request path.

b) **Test settings:** The `IDL_Cubit` test was run to measure the request execution time at the server and round-trip delay at the client. The thread pool size was set to 2 and the buffer size to 500. Size of the scoped memory region was set to 512 KB. The number of simultaneous client connections was increased from 1 to 200 in increments of 50.

c) **Analysis of results:** We now analyze the results of benchmarks that measure the average latency, the dispersion and worst case behavior of round-trip latency.

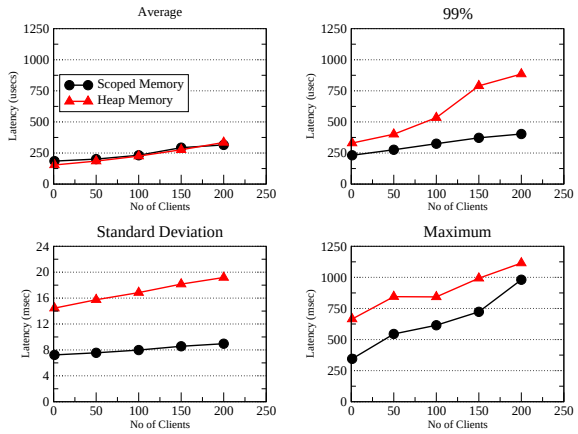

Fig. 5. Round-trip Latency Analysis

- **Average Measures.** Figure 5 shows that the use of scoped memory increases the average round-trip delay. This increase is due to the increase request execution time at the server side. As the number of clients increase, however, latency for heap memory also increases. At 200 clients the round-trip time for heap memory exceeds scoped memory by $\sim$ 20 μsecs. This behavior occurs due to the increased GC activity for heap memory with an increase in the number of clients.
- **Dispersion measures.** The dispersion measures for both heap and scope memory increases with the number of clients. However, measures for scope memory vary little with an increase in the number of clients, indicating better predictability. On the average, predictability improves as much as 50%.
- **Worst case measures.** The experiment reveal that use of scoped memory significantly bounds worst case measures. Though the average request execution latency is greater for scoped memory, its 99% and worst case latencies are smaller.

The empirical results described above indicates that the use of RTSJ scoped memory enhances predictability without unduly degrading performance. Moreover, the worst case measures are tighter and significantly better compared to that of heap memory.

III. Related Work

In recent years, a considerable amount of research has focused on enhancing the predictability of real-time middleware for DRE applications. RTSJ middleware is an emerging field of study. Researchers are focusing at RTSJ implementations, benchmarking efforts, and program compositional techniques. In this section, we summarize key efforts related to our work on ZEN.

The TimeSys corporation has developed the official RTSJ Reference Implementation (RI) [9], which is a fully compliant implementation of Java that implements all the mandatory features in the RTSJ. TimeSys has also released the commercial version, JTime, which is an integrated real-time JVM for embedded systems. In addition to supporting a real-time JVM, JTime also provides an ahead-of-time compilation model that can enhance RTSJ performance considerably.

The jRate *[10], [21]* project is an open-source RTSJ-based real-time Java implementation developed at Washington University, St. Louis. jRate extends the open-source GNU Compiler for Java (GCJ) run-time system [22] to provide an ahead-of-time compiled platform for RTSJ.

The *Real-Time Java for Embedded Systems* (RTJES) program [23] is working to mature and demonstrate real-time Java technology. A key objective of the RTJES program is to assess important real-time capabilities of real-time Java technology via a comprehensive benchmarking effort. This effort is examining the applicability of real-time Java within the context of real-time embedded system requirements derived from Boeing's Bold Stroke avionics mission computing architecture [24].

The researchers at the Washington University, St Louis are investigating automatic mechanisms [25] that enable existing Java programs to become storage-aware RTSJ programs. Their work centers on validating RTSJ storage rules using program traces and introducing storage mechanisms automatically and reversibly into Java code. Their other endeavors include building small foot-print feature rich Real-time Event Channel [26] using AspectJ [27], where Event Channel features are added via aspects.

IV. Concluding Remarks

Distributed Real-time and Embedded (DRE) systems are growing in number and importance as software is increasingly used to automate and integrate information systems with physical systems. Over 99% of all microprocessors are now used for DRE systems [28]. Ensuring end-to-end middleware predictability is essential to support the QoS capabilities needed by DRE applications. Therefore integration of RTSJ and Real-time CORBA is essential to ensure predictability required for DRE applications.

This paper described our R&D activities associated with an previously unexplored dimension in real-time middleware: *the integration of RTSJ features to support Real-time CORBA*. We showed how scoped memory and real-time threads can be associated within a real-time ORB Core without violating RTSJ rules, yet still remaining compatible with the CORBA specification. The empirical results presented in Section II-D demonstrate that significant predictability improvements can be achieved by applying RTSJ features in ZEN. All of our optimizations and enhancements are compliant with the CORBA specification and are transparent to DRE application developers.

V. Acknowledgments

We would like to acknowledge the efforts of the other members of the Distributed Object Computing (DOC) research group at UC Irvine who are contributing to the design and implementation of ZEN. Special thanks to Angelo Corsaro, Carlos O'Ryan, and Ossama Othman for contributing to ZEN's initial design.

References

[1] Realtime Platform SIG, "Realtime CORBA," White Paper, Object Management Group, Dec. 1996, Editor: Judy McGoogan, Lucent Technologies.

[2] David C. Sharp, "Avionics Product Line Software Architecture Flow Policies," in *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Oct. 1999.

[3] Chris Gill, Douglas C. Schmidt, and Ron Cytron, "Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing," *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, vol. 91, no. 1, Jan. 2003.

[4] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 2.6*, Dec. 2001.

[5] Ann Wollrath, Roger Riggs, and Jim Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, no. 4, November/December 1996.

[6] Douglas C. Schmidt and Steve Vinoski, "The History of the OMG C++ Mapping," *C/C++ Users Journal*, Nov. 2000.

[7] Inc ZeroC, "The Internet Communications Engine^{TM}," `www.zeroc.com/ice.html`, 2003.

[8] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull, *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[9] TimeSys, "Real-Time Specifi cation for Java Reference Implementation," `www.timesys.com/rtj`, 2001.

[10] Angelo Corsaro and Douglas C. Schmidt, "Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems," in *Proceedings of the* 8^{th} *IEEE Real-Time Technology and Applications Symposium*, San Jose, Sept. 2002, IEEE.

[11] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

[12] Angelo Corsaro, Douglas C. Schmidt, Raymond Klefstad, and Carlos O'Ryan, "Virtual Component: a Design Pattern for Memory-Constrained Embedded Applications," in *Proceedings of the* 9^{th} *Annual Conference on the Pattern Languages of Programs*, Monticello, Illinois, Sept. 2002.

[13] Raymond Klefstad, Douglas C. Schmidt, and Carlos O'Ryan, "The Design of a Real-time CORBA ORB using Real-time Java," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing*. IEEE, Apr. 2002.

[14] Raymond Klefstad, Arvind S. Krishna, and Douglas C. Schmidt, "Design and Performance of a Modular Portable Object Adapter for Distributed, Real-Time, and Embedded CORBA Applications," in *Proceedings of the 4th International Symposium on Distributed Objects and Applications*, Irvine, CA, October/November 2002, OMG.

[15] Arvind S. Krishna, Douglas C. Schmidt, Raymond Klefstad, and Angelo Corsaro, "Towards predictable real-time Java object request brokers," in *Proceedings of the 9th Real-time/Embedded Technology and Applications Symposium (RTAS)*, Washington, DC, May 2003, IEEE.

[16] David F. Bacon, Perry Cheng, and V. T. Rajan, "A real-time garbage collector with low overhead and consistent utilization," in *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2003, pp. 285–298, ACM Press.

[17] Gordon S. Blair and G. Coulson and P. Robin and M. Papathomas, "An Architecture for Next Generation Middleware," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, 1998, pp. 191–206, Springer-Verlag.

[18] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell, "The Case for Reflective Middleware," *Communications of the ACM*, vol. 45, no. 6, pp. 33–38, June 2002.

[19] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[20] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*, Wiley & Sons, New York, 2000.

[21] Angelo Corsaro and Douglas C. Schmidt, "The Design and Performance of the jRate Real-Time Java Implementation," in *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, Robert Meersman and Zahir Tari, Eds., Berlin, 2002, Lecture Notes in Computer Science 2519, Springer Verlag, pp. 900–921.

[22] GNU is Not Unix, "GCJ: The GNU Compiler for Java," `http://gcc.gnu.org/java`, 2002.

[23] Jason Lawson, "Real-Time Java for Embedded Systems (RTJES)," `http://www.opengroup.org/rtforum/jan2002/slides/java/lawson.pdf`, 2001.

[24] David C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," in *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.

[25] Morgan Deters, Nicholas Leidenfrost, and Ron K. Cytron, "Translation of Java to Real-Time Java using aspects," in *Proceedings of the International Workshop on Aspect-Oriented Programming and Separation of Concerns*, Lancaster, United Kingdom, Aug. 2001, pp. 25–30, Proceedings published as Tech. Rep. CSEG/03/01 by the Computing Department, Lancaster University.

[26] Frank Hunleth, "Building customizable middleware using aspect-oriented programming," M.S. thesis, Washington University in Saint Louis, 2002.

[27] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, "An overview of AspectJ," *Lecture Notes in Computer Science*, vol. 2072, pp. 327–355, 2001.

[28] Alan Burns and Andy Wellings, *Real-Time Systems and Programming Languages, 3rd Edition*, Addison Wesley Longmain, Mar. 2001.