Effective Use of Real-time Java to implement Real-time CORBA

Arvind S. Krishna
Douglas C. Schmidt
Elec & Comp. Eng. Dept
Vanderbilt University
{arvindk, schmidt}@dre.vanderbilt.edu

Krishna Raman
Raymond Klefstad
Elec & Comp. Eng. Dept
University of California, Irvine
{kraman, klefstad}@uci.edu

OMG Workshop on Distributed Object Computing
For Real-time and Embedded Systems
Washington D.C

Motivation for ZEN Real-time ORB

Integrate best aspects of several key technologies

- **Java**: Simple, less error-prone, large user-base
- **Real-time Java**: Real-time support
- **CORBA**: Standards-based distributed applications
- **Real-time CORBA**: CORBA with Real-time QoS capabilities

ZEN project goals

- Make development of distributed, real-time, & embedded (DRE) systems easier, faster, & more portable
- Provide open-source Real-time CORBA ORB written in Real-time Java to enhance international middleware R&D efforts

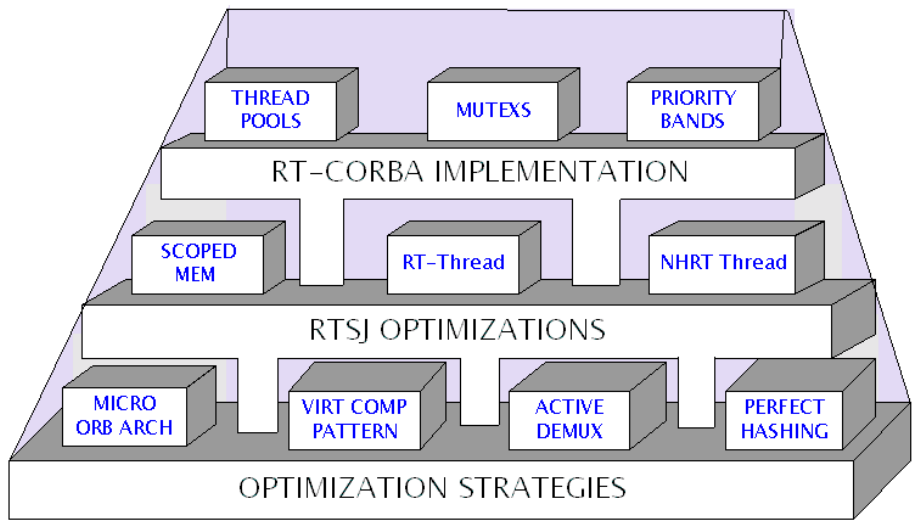
Overview - ZEN R&D Plan

Phase I → Apply Optimization patterns and principles

- ORB-Core Optimizations
  - Micro ORB Architecture → Virtual Component Pattern
  - Connection Management → Acceptor-Connector pattern, Reactor (java's nio package)
  - Collocation and Buffer Management Strategies
- POA Optimizations
  - Request Demultiplexing → Active Demultiplexing & Perfect Hashing
  - Object Key Processing Strategies → Asynchronous completion token pattern
  - Servant lookup → Reverse lookup map
  - Concurrency Strategies → Half-Sync/Half-Async

Phase II → Enhance Predictability by applying RTSJ features

- Associate Scoped Memory with Key ORB Components
  - I/O Layer: Acceptor Connector, Transports
  - ORB Layer: CDR Streams, Message Parsers
  - POA Layer: Thread-Pools and Upcall Objects
- Using *NoHeapRealtime Threads*
  - Ultimately use NHRT threads for request/response processing
  - Reduce priority inversions from Garbage Collector



Phase III → Build a Real-Time CORBA ORB that runs atop a mature RTSJ Layer

RTSJ Thread Model

The Real-Time Specification for Java (RTSJ) extends Java in the following areas:

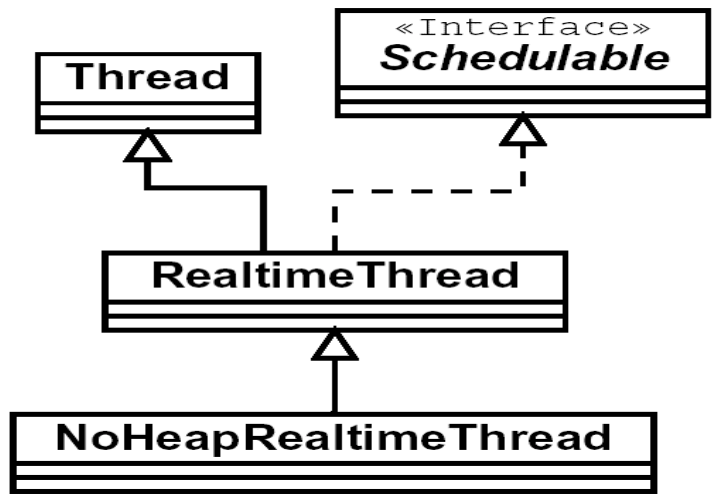
- New memory management models that can be used in lieu of garbage collection
- Stronger semantics on thread and their scheduling
- Access to physical memory
- Asynchronous Event handling mechanism
- Timers and Higher time resolution
- Priority pre-emptive scheduler

RTSJ Thread Model

Real-time Threads – priority and scheduling characteristics specified

NoHeapRealtimeThreads

- Do not "touch" the heap
- Use of NHRT threads have exec eligibility higher than that of GC



The RTSJ does not syntactically extend the Java language, strengthens the semantics of certain Java features

RTSJ Memory Model

Scoped Memory

Types

- LTMemory: Allocation time is linear.
- VTMemory: Variable Allocation time.

Properties

- Reference counted; no of active threads in region
- When reference count of a region drops to zero
  - All Objects within that region are considered unreachable
  - Finalizers of all objects run;

Scoped Memory Allocation

- Only a real-time thread may allocate from scoped region; making region the current allocation context.
  - enter() method on the scoped memory region

Single Parent Rule

- Every Scoped Memory region may have only one parent

Assignment Rules

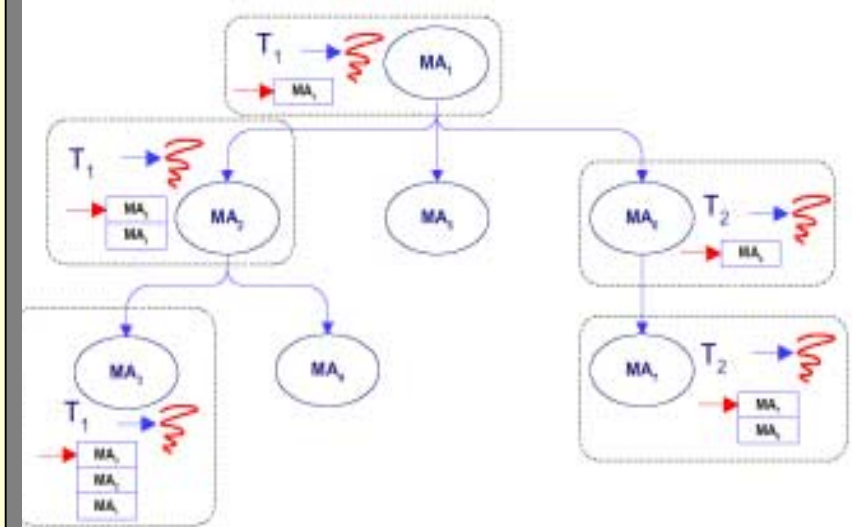
- obj in region m_a can hold ref to obj in region m_b if
  - lifetime (m_a) <= lifetime (m_b)
- Heap & Immortal can never refer to objs in Scoped regions

Immortal Memory

- Same lifetime as the JVM
- Objects allocated never garbage collected

Physical Memory

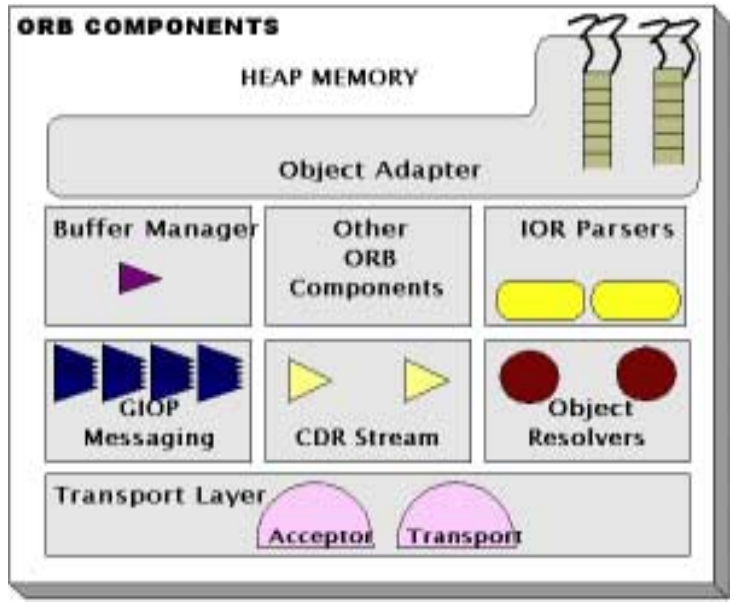
- Allows access to specific locations based on addresses.



Applying RTSJ features in ZEN

Original design of ZEN

- All components allocated in heap
- Request processing thread may be preempted by GC (demand garbage collection)

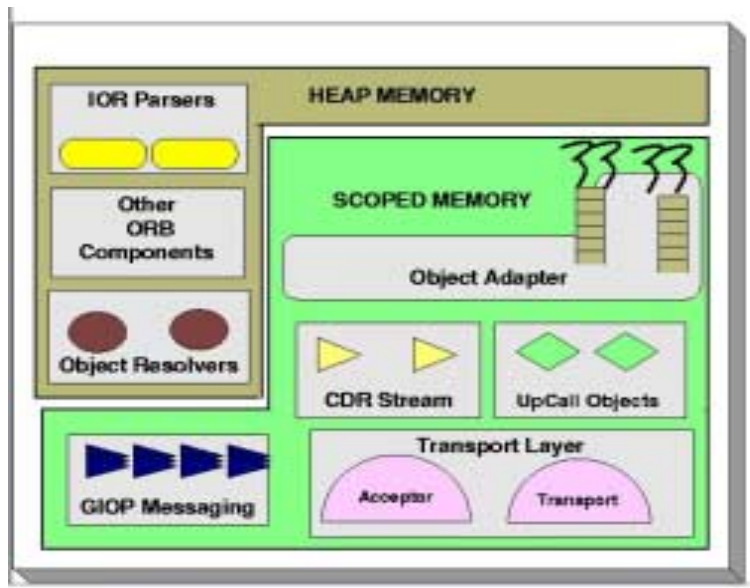


Goals

- Compliance with CORBA specification
- Interoperability with classic CORBA
- Reduce overhead for applications not using real-time features
- End-user transparent

Design of ZEN for Real-Time CORBA

- Apply scoped memory along critical request processing path
- Ultimately NHRT threads used for request processing
  - POA level policies
  - **Proper use of NHRT threads would minimize GC execution during request processing**



Analyzing Request Processing Steps

Client Side Connection Initiation

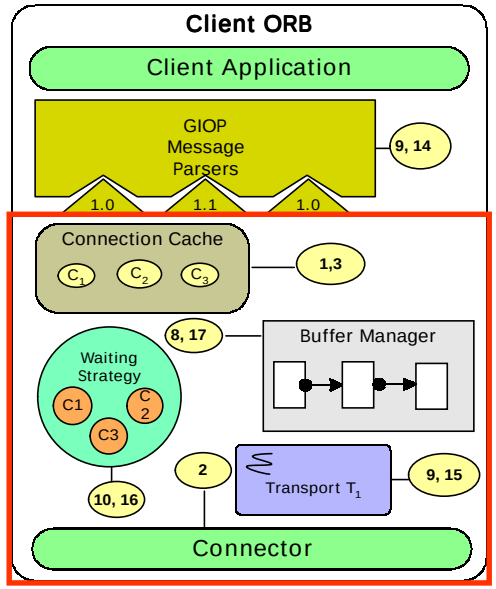
1. The client ORB's connection cache *ConnectorRegistry* is queried for an existing connection to the server
2. If no previous connection exists, a separate connection handler (*Transport* T_1) is created; *Connector* connects to the server
3. This connection is added to the *ConnectorRegistry* since C_1 is bidirectional

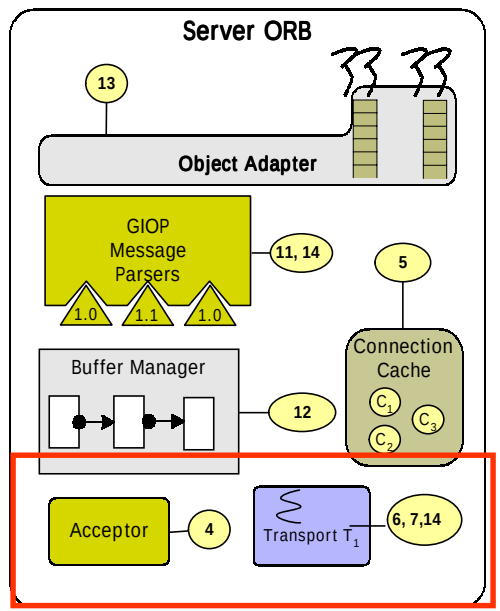
Steps are done for every request

These activities are done for every client connection

Server Side Connection Acceptance

4. An acceptor accepts the new incoming connection.
5. This connection C_1 is then added to the server's connection cache, AcceptorRegistry 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.





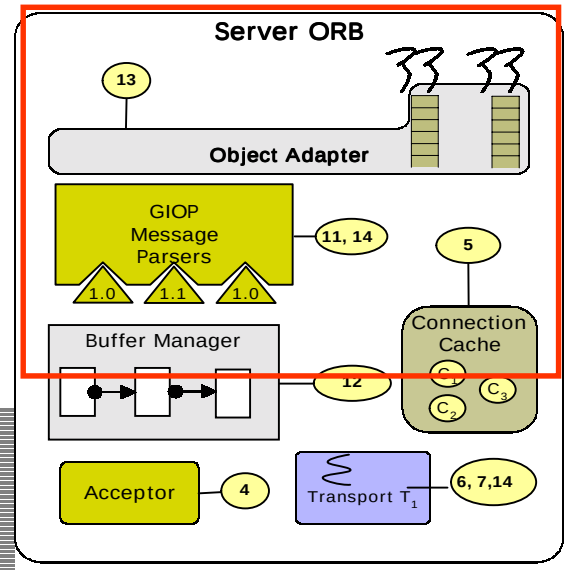
Request Processing – Server Side

Independent & Memory less

Steps for two different clients do not share context
Same for one client as well

Repetitive & Ephemeral

Carried out for each client request
Typically objects live for one cycle



Server Side Request Processing Steps

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 request and read data.
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.

Thread Bound

Steps executed by I/O threads
Thread-Pool threads

Application of Scoped Memory – ZEN

Analyzing Properties

Threadbound → associate with real-time threads

- ZEN's Acceptor-Connector and Transport classes use RT-Threads

Independent & Memoryless → associate with scoped memory

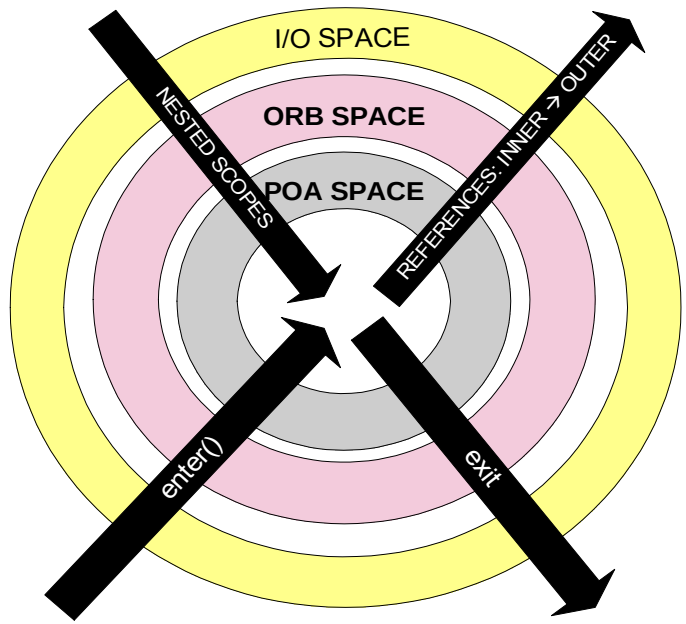
Ephemeral → objects created in scoped region not required after one cycle of request/response processing

Associating Scoped Memory

- Encapsulate steps as "logic" class → instance of Java's Runnable class
- Associate this logic class with real-time threads
- Threads make the scoped region the current allocation context by invoking enter() on the scoped region

Applying Scoped Memory

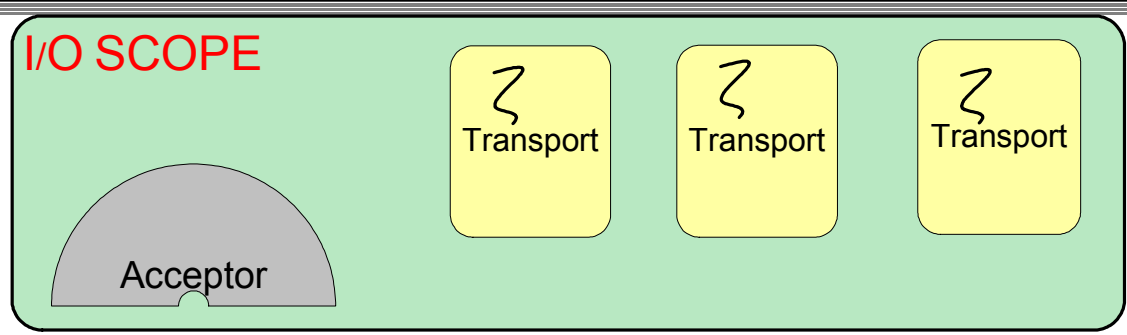
1. Break Steps into three broad regions based on request processing steps
   - I/O scope → read request
   - ORB scope → process request
   - POA scope → perform upcall send reply
2. Recursively enter each space from I/O → POA scopes
3. Implicitly exit regions from POA → I/O scope



Applying Scoped Memory – I/O Scope

I/O Scope

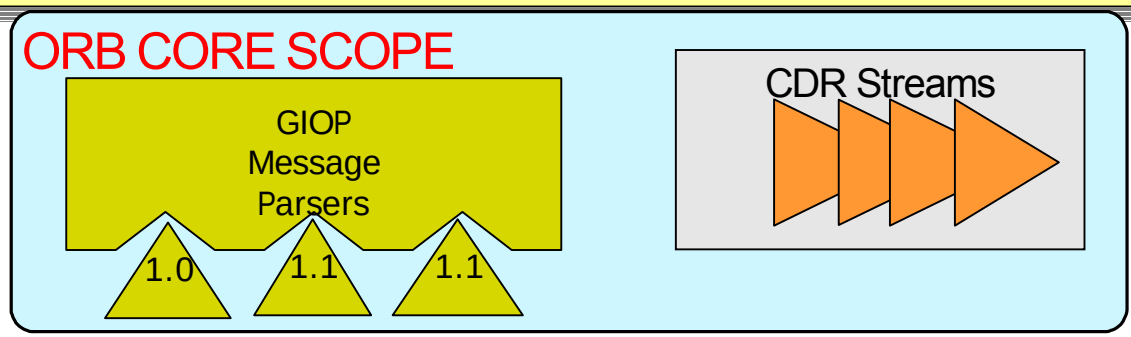
- **Steps** **This phase of demultiplexing corresponds to the steps 4-7**
- **Participants** **The participants for this phase include, acceptors, connectors, and transports.**
- **RTSJ application**
  - **Each of these components are thread-bound components and are designed based on inner logic class**
  - **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_{io}**
  - **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_{io} region**



Applying Scoped Memory – ORB Scope

ORB Scope

- **Steps** **This phase of demultiplexing corresponds to the Steps 11-12**
- **Participants** **GIOP Message parsers, Buffer Allocators and CDR Streams.**
- **RTSJ application**
  - **Based on the size of the header a RequestMessage buffer to hold the request is created.**
  - **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**
  - **Using RTSJ memory rules, references from the ORB to the I/O space are valid**



Scoped Memory – POA Scope

Steps

- Demux request to get target POA, servant and skeleton
- Perform upcall on the servant
- Marshall reply back to client

Participants

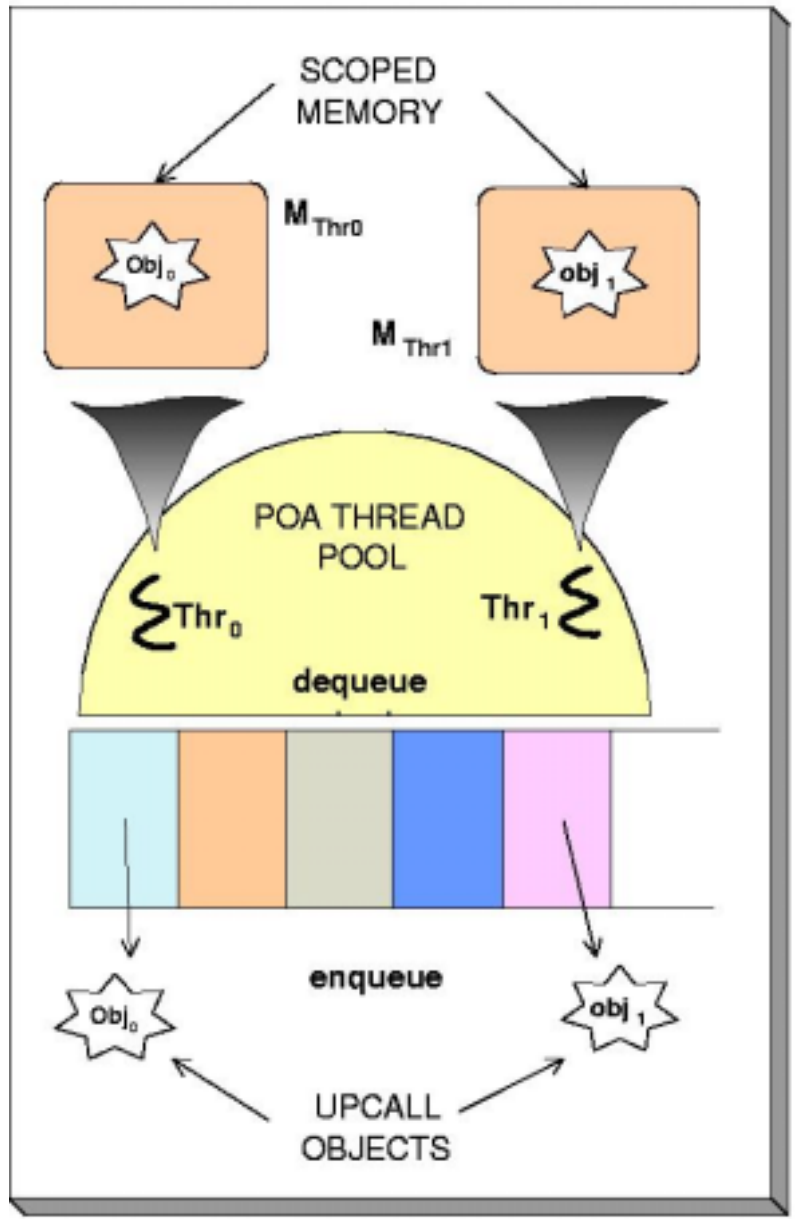
- Worker thread – deques reques from buffer
- Message parser – parses the request to find target POA & servant
- Upcall Object – holds info necessary to perform upcall
- Output buffer – holds response

RTSJ Association

- Each worker thread has a dedicated scoped region
- Worker thread dequeues the request from queue
- Makes the region its current allocation context
  - enter() method on scoped region
- Performs upcall and sends reply to the client
- Exits the region, enabling allocated objects to be freed

Our Application

- Compliant with the CORBA specification
- Does not violate RTSJ rules



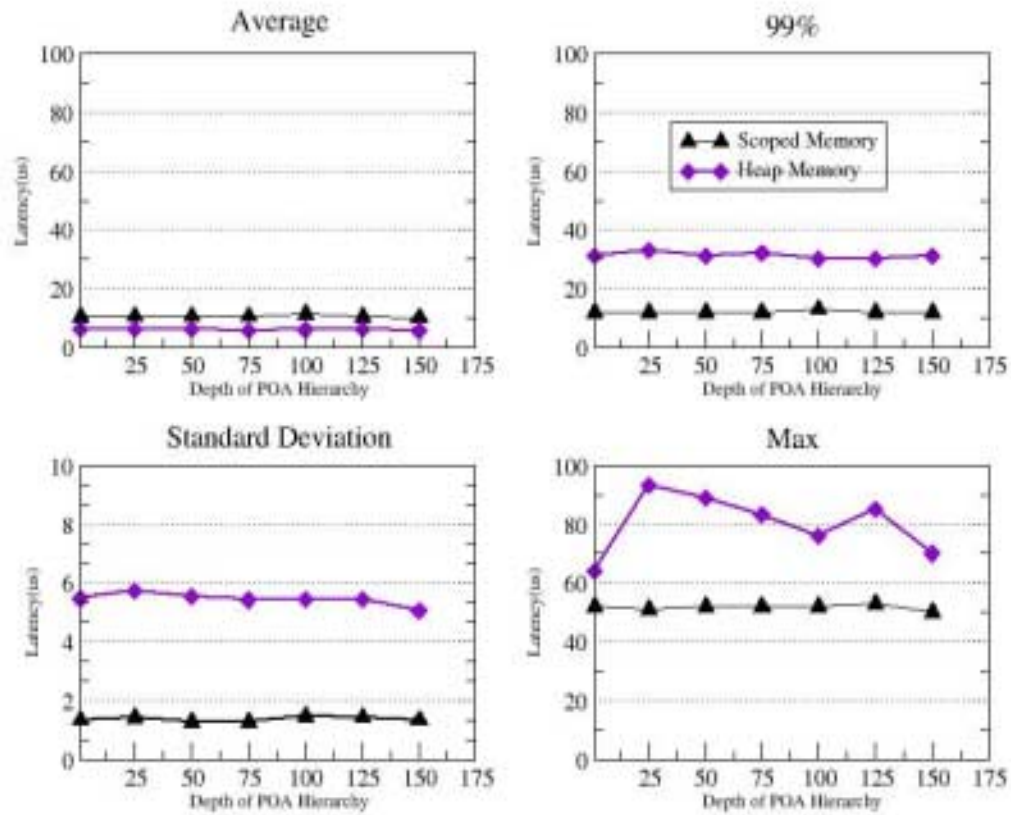
Predictability Enhancement

Overview

- POA Demultiplexing experiment conducted to measure improvement in predictability

Result Synopsis

- Average Measures:
  - Scoped Memory does have some overhead ~ 3 µs
- Dispersion Measures:
  - Considerable improvement in predictability
  - Dispersion improves by a ~ factor of 4
- Worst-Case Measures:
  - Scoped memory bounds worst case
  - Heap shows marked variability



Associating scoped memory

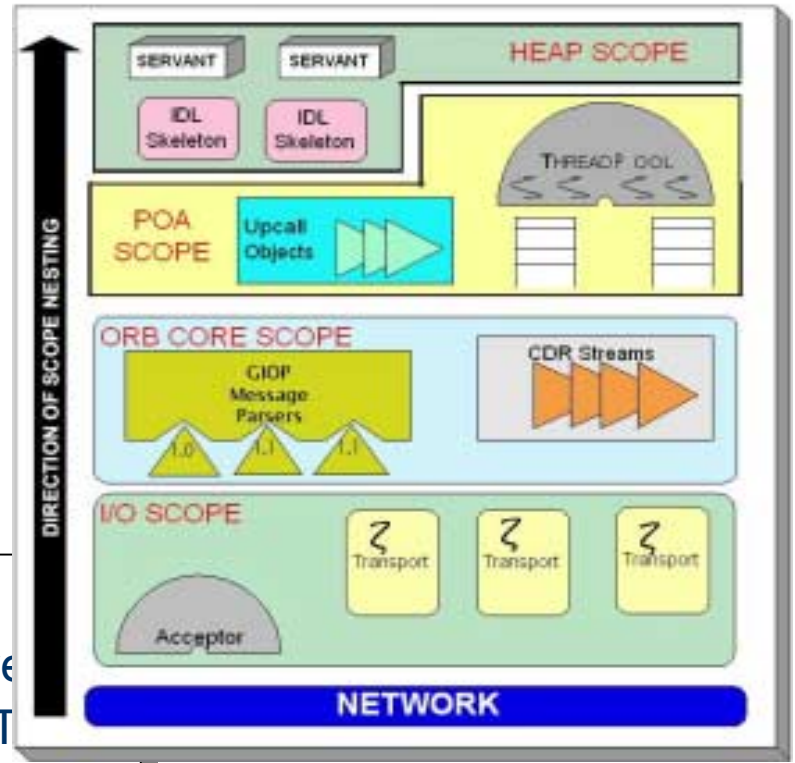
– Does not compromise performance
– significantly enhances predictability
– bounds worst case latency

http://www.cs.wustl.edu/~schmidt/PDF/RT-POA.pdf

Concluding Remarks & Future Work

Concluding Remarks

- We present optimizations applied to address RT-CORBA demultiplexing predictability & scalability challenges
- Our strategies in conjunction with RTSJ platform help bound jitter and worst case performance



Future Real-Time CORBA Research

- Applying Scoped Memory in ORB Core & I/O laye
- Resolving challenges arising from associating RT features
  - Thread borrowing
  - Use of NHRT threads may require end-user to be RTSJ aware
  - Modeling RTSJ exceptions e.g. *ScopedCycleException*
- Complete implementation of Real-time CORBA specification

Downloading ZEN

- www.zen.uci.edu

References

- ZEN open-source download & web page:
  - http://www.zen.uci.edu
- Real-time Java (JSR-1):
  - http://java.sun.com/aboutJava/communityprocess/jsr/jsr_001_real_time.html
- Dynamic scheduling RFP:
  - http://www.omg.org/techprocess/meetings/schedule/Dynamic_Scheduling_RFP.html
- Distributed Real-time Java (JSR-50):
  - http://java.sun.com/aboutJava/communityprocess/jsr/jsr_050_drt.html
- AspectJ web page:
  - http://www.aspectJ.org
- JRate
  - http://tao.doc.wustl.edu/~corsaro/jRate/