Report Number: WUCSE-2004-54

2004-09-17

Techniques and Patterns for Safe and Efficient Real-Time Middleware

Angelo Corsaro

Over 90 percent of all microprocessors are now used for real-time and embedded applications. The behavior of these applications is often constrained by the physical world. It is therefore important to devise higher-level languages and middleware that meet conventional functional requirements, as well as dependably and productively enforce real-time constraints. Real-Time Java is emerging as a safe, real-time environment. In this thesis we use it as our experimentation platform; however, our findings are easily adapted to other similar platforms. This thesis provides the following contributions to the study of safe and efficient real-time middleware. First, it identifies potential bottlenecks... **Read complete abstract on page 2.**

Techniques and Patterns for Safe and Efficient Real-Time Middleware

Angelo Corsaro

Complete Abstract:

Over 90 percent of all microprocessors are now used for real-time and embedded applications. The behavior of these applications is often constrained by the physical world. It is therefore important to devise higher-level languages and middleware that meet conventional functional requirements, as well as dependably and productively enforce real-time constraints. Real-Time Java is emerging as a safe, real-time environment. In this thesis we use it as our experimentation platform; however, our findings are easily adapted to other similar platforms. This thesis provides the following contributions to the study of safe and efficient real-time middleware. First, it identifies potential bottlenecks and problem with respect to guaranteeing real-time performance in middleware. Second, it presents a series of techniques and patterns that allow the design and implementation of safe, predictable, and highly efficient real-time middleware. Third, it provides a set of architectural and design patterns that application developers can use when designing real-time systems. Finally, it provides a methodology for evaluating the merits and benefits of real-time middleware. Empirical results are presented using that methodology for the techniques presented in this thesis. The methodology helps compare the performance and predictability of general, real-time middleware platforms.

Department of Computer Science & Engineering

2004-54

Techniques and Patterns for Safe and Efficient Real-Time Middleware, Doctoral Dissertation, December 2004

Authors: Corsaro, Angelo

September 17, 2004

Abstract: Over 90 percent of all microprocessors are now used for real-time and embedded applications.
The behavior of these applications is often constrained by the physical world. It is therefore important to devise higher-level languages and middleware that meet conventional functional requirements, as well as dependably and productively enforce real-time constraints.

Real-Time Java is emerging as a safe, real-time environment. In this thesis we use it as our experimentation platform; however, our findings are easily adapted to other similar platforms.

This thesis provides the following contributions to the study of safe and efficient real-time middleware. First, it identifies potential bottlenecks and problem with respect to guaranteeing real-time performance in middleware

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

Short Title: Safe and Efficient RT-Middleware Corsaro, D.Sc. 2004

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

TECHNIQUES AND PATTERNS FOR SAFE AND EFFICIENT REAL-TIME MIDDLEWARE

by

Angelo Corsaro

Prepared under the direction of Ron K. Cytron and Douglas C. Schmidt

---

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

December, 2004

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

TECHNIQUES AND PATTERNS
FOR SAFE AND EFFICIENT REAL-TIME MIDDLEWARE

by Angelo Corsaro

ADVISORS: Ron K. Cytron and Douglas C. Schmidt

December, 2004
Saint Louis, Missouri

Over 90 percent of all microprocessors are now used for real-time and embedded applications. The behavior of these applications is often constrained by the physical world. It is therefore important to devise higher-level languages and middleware that meet conventional functional requirements, as well as dependably and productively enforce real-time constraints.

Real-Time Java is emerging as a safe, real-time environment. In this thesis we use it as our experimentation platform; however, our findings are easily adapted to other similar platforms.

This thesis provides the following contributions to the study of safe and efficient real-time middleware. First, it identifies potential bottlenecks and problem with respect to guaranteeing real-time performance in middleware.

Second, it presents a series of techniques and patterns that allow the design and implementation of safe, predictable, and highly efficient real-time middleware.

Third, it provides a set of architectural and design patterns that application developers can use when designing real-time systems.

Finally, it provides a methodology for evaluating the merits and benefits of real-time middleware. Empirical results are presented using that methodology for the techniques presented in this thesis. The methodology helps compare the performance and predictability of general, real-time middleware platforms.

Il vostro nome e’ in ogni mio gesto, in ogni mia azione.

Voi siete tutto! Tutto cio’ che ho. Tutto cio’ che conta.

Contents

List of Tables

List of Figures

Acknowledgments

A PhD is long and demanding journey. I like to think of it as Dante's journey thought the Inferno, Purgatorio and Paradiso. As Dante's journey, a PhD has moment of pain, of struggle, darkness, and moment of joy and light. Now that I am at the end of this trip, I'd like to acknowledge those that through my PhD have been my Virgilios. Thus I'd like to thank my advisors, Dr. Ron K. Cytron and Dr. Douglas C. Schmidt, and my mentors, Dr. Gordon Blair, Dr. Rajesh Gupta, Dr. Stephen Jenks, Dr. David L. Levine, and Dr. Sandeep Shukla. I'd also like to thank my committee members, Dr. Chris D. Gill, Dr. Chenyan Lu, Dr. Ronald K. Indeck, Dr. Aaron Stump.

I'd also like to thank all the wonderful people that have made this journey much more pleasant, (sorted by name): Akihiro Tanaka, Arvind Krishna, Ayman El'Ariss, Balachandran Natarajan, Carlos O'Ryan, Corrado Santoro, Francesco Cioci, Frank Hunleth, Gianfranco Doretto, Irfan Pyarali, Jeff Parsons, Joe Hoffert, Karin Zimmer, Kirthika Parameswaran, Krishnakumar Balasubramanian, Mayur Deshpande, Morgan Deters, Nanbor Wang, Ossama Othman, Peppinello Gatto, Pradeep Gore, Rabbih Zaboruh, Radu Handorean, Sharath Coletti, Vishal Kachroo, Windell Middlebrooks, Yamuna Krishnamurthy.

I'd also like to thank the real-time Java research community which has been continuous source of inspiration, and a very good source of feedback. Thus, great thanks go to Greg Bolella, Peter Dibble, Daniel Dvorak, David Holmes, Doug Lea, Doug Locke, Martin Rinard, Gautam Thaker, Ian Vitek.

Finally, I thank DARPA for supporting my research under contract F33615-00-C-1697

Angelo Corsaro

Washington University in Saint Louis
December 2004

Chapter 1

Introduction

1.1 Introduction

Current Challenges. The vast majority of all microprocessors are now used for embedded systems, in which computer processors control physical, chemical, or biological processes or devices in real-time. Examples of such systems include telecommunication networks (*e.g.*, wireless phone services), tele-medicine (*e.g.*, remote surgery), manufacturing process automation (*e.g.*, hot rolling mills), and defense applications (*e.g.*, avionics mission computing systems). These real-time embedded systems are increasingly being connected via wireless and wireline networks.

Designing real-time embedded systems that implement their required capabilities, are safe, dependable and predictable, and are parsimonious in their use of limited computing resources is hard; building them on time and within budget is even harder. Moreover, due to global competition for marketshare and engineering talent, many companies are now also faced with the problem of developing and delivering new products in short timeframes. It is therefore essential that the production of real-time embedded systems take advantage of languages, middleware, tools, and methods that enable higher software productivity, without unduly degrading the quality of service (QoS).

Another issue currently faced by most of the IT industry is the high cost of software maintenance, which, for long lived systems (for instance, Air Traffic Control Systems live for 20-25 years or more), is the largest contributor to software cost. To provide the reader with a feel of the order of magnitude, it is sufficient to consider that in most enterprises, the cost of software production is around $20 per line of code [6], and maintenance can be

accounted for more than the 65% of the total software cost (often 80% in very long lived systems). The relationship between the cost of initial software development and software maintenance should make apparent that a key way of saving money for the IT industry is to invest in making system maintenance as inexpensive as possible.

Finally, most real-time embedded systems such as Flight Control Software, Nuclear Plan Control Software, Air Traffic Control systems, etc., have very stringent safety requirements. Assuring an appropriate safety level can be quite expensive in term of effort and cost; thus, being able to offload some of the developer responsibility to the middleware platform makes it time and cost effective to develop safety critical systems. For instance, the use of a safe language such as Java^{TM} [32], has a positive impact on the software safety assurance, since some types of faults, *e.g.* those due to memory management, cannot happen.

From the above, it should be apparent that the current IT industry could take great advantage of a safe and efficient middleware platform that allows the reduction of costs associated with development, maintenance and safety assurance. While Java has partly solved the problem for the business IT industry, these same problems still need to be tackled in the real-time and embedded industry.

The State of the Art. Many real-time embedded systems are still developed in C, and increasingly in C++. While writing in C and C++ is more productive than assembly code, they are not the most productive or safe programming languages. A key source of errors in C/C++ stems from their *memory management* mechanisms, which require programmers to allocate and deallocate memory manually. Moreover, C++ is a feature rich, complex language with a steep learning curve, which makes it hard to find and retain experienced real-time embedded developers who are trained to use it well.

Real-time embedded software should ultimately be synthesized from high-level specifications expressed with domain-specific modeling tools [44]. Until those tools mature, however, a considerable amount of real-time embedded software still needs to be programmed by software developers. Ideally, these developers should use programming languages and middleware that shield them from many accidental complexities, such as type errors, memory management, real-time scheduling enforcement, and steep learning curves.

The Road Ahead. As described above, industries operating in the real-time embedded segment, are still mostly using development environments bases on C/C++. While the

transition to C++ is somewhat recent for some of these industries, many of them are already looking at safer, more productive and maintainable software infrastructure on which to base their next generation systems. In this domain, Java has raised a lot of interest, mostly because of its rapidly growing programmer base, its simplicity, its safety, and especially its cheaper maintenance cost when compared to C/C++. To this end, published studies [39] on productivity and reported defect rates, show the ratio of C++ bugs-per-KSLOC to Java bugs-per-KSLOC as being in the range 2.5 to 3.5. C++ generates between 15% and 50% more defects per KSLOC. C++ produces between 200% and 300% more bugs per hour. Java is also between 30% and 200% more productive, in terms of lines of code per minute.

However, conventional Java implementations are unsuitable for developing real-time embedded systems, mostly due to the following problems:

- The scheduling of Java threads is purposely underspecified to make it easy to develop Java Virtual Machine (JVM)s for new platforms.
- Most of Java garbage collectors work in a *stop the world and collect* fashion, in which the garbage collector has to stop all the running threads before attempting to reclaim dead storage. Other types of garbage collectors take a less restrictive approach, but most of the precise garbage collectors known in literature [31] are not sufficiently predictable to meet the needs of real-time systems.
- Java provides coarse-grained control over memory allocation and access, *i.e.*, it allows applications to allocate objects on the heap, and provides no control over the type of memory in which objects are allocated, e.g. DMA etc.
- Due to its interpreted origins, the performance of JVM middleware has historically lagged that of equivalent C/C++ programs by an order of magnitude or more.

To address these problems, the Real-time Java Experts Group has defined the Real-Time Specification for Java (RTSJ) [8], which provides the following capabilities:

- New memory management models that can be used in lieu of garbage collection.
- Access to raw physical memory.
- A higher resolution time granularity suitable for real-time systems.
- Stronger guarantees on thread semantics when compared to regular Java, *i.e.*, the most eligible runnable thread is always run.

Thesis Scope The goal of this thesis is that of investigating techniques and patterns for implementing safe and efficient middleware platforms, and empirically evaluating its predictability and performances. Since the most interesting emerging standard for safe, real-time middleware is The Real-Time Specification for Java (RTSJ), we have constrained our research space to this domain. Key findings will be reusable in any safe, real-time middleware, since the forces that have to be solved would be similar, and the overall architecture of the system might be very similar to the one used for building effective RTSJ middleware.

1.2 Related Work

Although the RTSJ was adopted fairly recently [8], there is already a number of research projects investigating how to make this platform effective. At the current time there is great interest around this platform, since it has great potential as a candidate for a safe middleware platform. The most interesting projects working on this topic are the following:

- **FLEX** [36] provides a Java compiler written in Java, along with an advanced code analysis framework. FLEX generates native code for StrongARM or MIPS processors, and can also generate C code. It uses advanced analysis techniques to automatically detect the portions of a Java application that can take advantage of certain real-time Java features, such as memory areas or real-time threads.

- The OVM [38] project is developing an open-source JVM framework for research on the RTSJ and programming languages. The OVM virtual machine is written entirely in Java and its architecture emphasizes customizability and pluggable components. Its implementation strives to maintain a balance between performance and flexibility, allowing users to customize the implementation of operations such as message dispatch, synchronization, and field accesses. OVM allows dynamic updates of the implementation of instructions on a running VM.

- Work on real-time storage allocation and collection [22] is being conducted at Washington University, St. Louis. The main goal of this effort is to develop new algorithms and architectures for memory allocation and garbage collection that provide worst-case execution bounds suitable for real-time embedded systems.

- The Real-Time Java for Embedded Systems (RTJES) program [30] 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 [43].

1.3 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 presents an overview of the RTSJ, by concentrating on those features that are more relevant to the scope of this work.
- Chapter 3 provides an explanation on the RTSJ programming model, and reports a series of key pattern for the development of RTSJ-based applications.
- Chapter 4 systematically highlights the limitations of the RTSJ that make it difficult for it to be a *truly* safe and efficient real-time platform, and provides solutions in terms of new algorithms or data structures. This chapter also provide a catalog of key idioms and patterns used to implement safe and efficient RTSJ middleware.
- Chapter 5 provides an overview of a series of proposed extensions to the RTSJ, and justifies the rationale behind each proposed extension.
- Chapter 6 introduces the benchmarking suite RTJPerf and provides an in depth performance evaluation of some of the most representative RTSJ platform.
- Chapter 7 provides some concluding remarks and outline a possible path for future research and development.

Chapter 2

Overview of Real-Time Java

2.1 Introduction

The RTSJ [8] extends the Java platform by providing additional APIs and refining the semantics of certain constructs. The additions and semantical refinements provided by the RTSJ are meant to enable the development of a vast class of real-time systems, ranging from soft to hard-real-time systems.

As motivated in the RTSJ specification, the guiding principles that were taken into consideration when designing it, were the following:

- **Applicability to particular Java environments.** The RTSJ should not be bound to a particular development environment such as J2SE or J2ME etc. Instead it should be equally applicable to all Java environments.

- **Backward compatibility.** Existing Java application should be able to run on RTSJ compliant JVMs, as if they were running on a regular JVM, thus not noticing any difference.

- **Write Once, Run Anywhere.** While the RTSJ recognizes the importance of *Write Once, Run Anywhere* (WORA), what it sets are its foremost goal is the execution predictability of RTSJ applications. Thus, it relies on the more relaxed *Write Once Carefylly, Run Anywhere Conditionally* (WOCRAC).

- **Current Practice vs. Advanced Features.** The RTSJ while treasuring from common practices, should not preclude future implementation of advanced features.

- **Predictable Execution.** The first and foremost goal of the RTSJ is predictable execution. Predictability should always come first then any other general purpose computing performance measures.

- **No Syntactic Extensions.** The RTSJ shall not include nor require any syntactic extension to the Java language.

- **Allow Variation in Implementation Decisions.** The RTSJ recognizes that different implementors might decide to implement different subsets of the specification as well as use different techniques to build their real-time JVM. The only mandatory compliance point, other than a minimum subset of features, is the semantics of the RTSJ is maintained.

The reminder of this chapter will provide an overview of the RTSJ. In order to understand the forces that drove certain of the RTSJ decision, it will be important to keep in mind the design constraint that were outlined above.

2.2 Threads, Scheduling and Synchronization

The Java language provides built-in support for multi-threading and synchronization; however the shortcomings described below limit the usage of these features in the context of real-time and embedded systems:

- Java threads can be preempted by the garbage collector for an unpredictable amount of time.

- The scheduling of Java threads is under specified [1].The Java scheduler, while priority -based, is not guaranteed to be priority preemptive, *e.g.*, it is not guaranteed that the highest priority runnable thread will be scheduled for execution.

- There is no support for specifying the execution characteristics of a thread, *e.g.*, periodicity, release characteristics, cost etc.

- Java's synchronization abstractions do not provide any provision to limit priority inversion, such as priority ceiling, priority inheritance.

[1]This underspecification was introduced intentionally to make it possible to implement a Java VM efficiently on as many platform as possible.

To address these limitations, and to provide a set of usable abstractions to real-time Java developers, the RTSJ extends the Java support for threading, synchronization, and scheduling so to allow complete control over thread execution, scheduling, and synchronization. In the remainder of this Chapter we provide an overview of the specific extension provided by the RTSJ on threading, scheduling and synchronization.

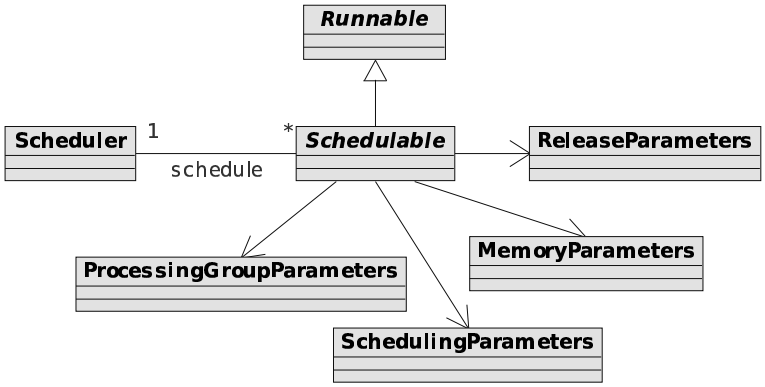


Figure 2.1: RTSJ Scheduling APIs.

2.2.1 Real-Time Java Scheduling

The RTSJ extends Java platform scheduling by introducing a real-time scheduling framework. The main actors of this framework, as depicted in Figure 2.1, are `Schedulable` entities and a `Scheduler`. The main idea at the foundation of this scheduling framework is that a platform scheduler controls the execution of schedulable entities. For example, a given scheduler might ensure through admission control that all threads can be feasibly scheduled, *i.e.* there are enough resources to complete all `Schedulable` entities within the specified timing constraints. Below we provide a more detailed description of the RTSJ scheduling framework and its participants.

The Scheduler. The `Scheduler` class, as depicted in Figure 2.1, defines the abstract protocol supported by concrete RTSJ's scheduler implementations. This behaviour consists of managing a `Schedulable`'s object's execution, and perhaps to perform feasibility analysis. The `Scheduler` class can be subclassed, as shown in Figure 2.2 [2], in order to provide specific scheduling disciplines, such as, priority preemptive, Rate Monotonic (RM), Earliest Deadline First (EDF), Least Laxity First (LLF), etc. To this end, the RTSJ

[2] In UML diagrams classes printed without any filling are user defined classes. Classes with filling are those defined in the RTSJ.

specifies only a concrete scheduler–a priority preemptive scheduler that supports a minimum of 28 different priorities[3]. The properties of a `Schedulable` entity, used by the platform scheduler in order to perform feasibility analysis and scheduling are described next.

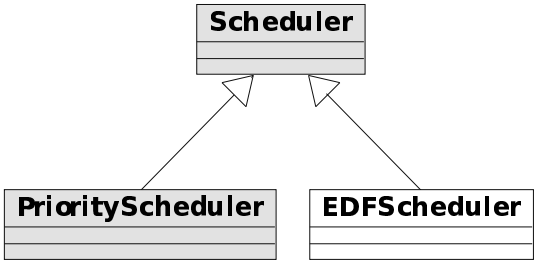


Figure 2.2: Customizing the RTSJ Scheduler.

Schedulable Entities. The `Schedulable` class provides an abstraction for those application entities that consume computational resources. In order to make it possible for the scheduler to (1) analyze the feasibility of the system, and (2) compute a schedule that will satisfy the application time-lines, `Schedulable` entities are associated with a set of properties that describe resource and timing requirements. Specifically, as depicted in Figure 2.2, with each `Schedulable` object it is possible to associate the following information:

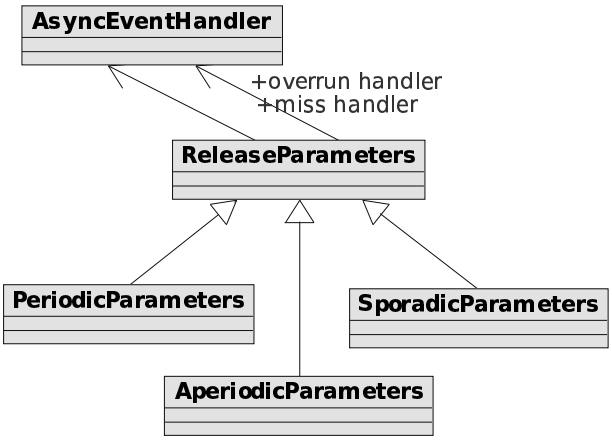


Figure 2.3: RTSJ Release Parameters.

- `ReleaseParameters` are the means provided by the RTSJ scheduling API to specify the release characteristics of a `Schedulable` entity. Specifically, as shown

[3]This 28 priorities are all above the 10 priority levels defi ned by the Java platform for java.lang.Thread.

in Figure 2.3, the class `ReleaseParameters` is the base class of a hierarchy of classes that provide a way of describing periodic, aperiodic, and sporadic computations. The properties shared by `ReleaseParameters`' subclasses are (1) computation cost, and (2) deadline. Each subclass adds other properties that are specific to the abstraction. For instance, the `PeriodicParameters` class provides a way of specifying a period; on the other hand the `AperiodicParameters` class provides a way of specifying a minimum inter-arrival time. To each `ReleaseParameters` instance two different kind of handlers can be associated for coping with the situation in which the associated `Schedulable` overruns or misses its deadline. These handlers (see Section 2.5 for a description of `AsyncEventHandler`), if registered, are notified by the RTSJ `Scheduler` when the `Schedulable` entity misses a deadline or executes for an amount of time greater than the one specified in its associated `ReleaseParameters`.

- The `SchedulingParameters` class provide the base class for concrete scheduling parameters. As shown in Figure 2.4, the RTSJ provides subclasses for describing scheduling information needed by priority based schedulers (`PriorityParameters`, `SchedulingParameters`). Other subclasses could be added to represent scheduling information needed by other scheduling disciplines.

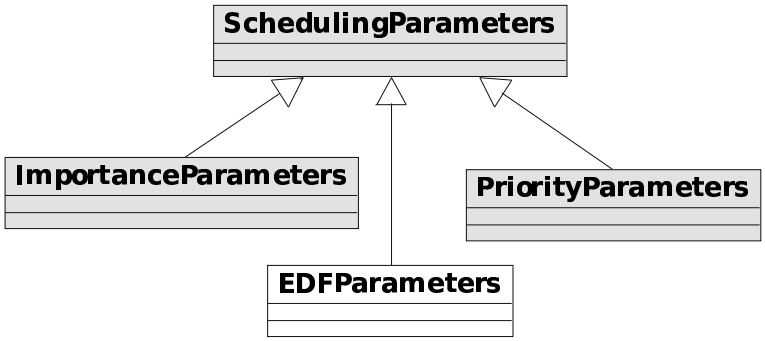


Figure 2.4: RTSJ Scheduling Parameters.

- `MemoryParameters` provides information on the memory allocation characteristics of a `Schedulable` entity. They provide a way of specifying the maximum amount of memory that a `Schedulable` entity might allocate and the allocation rate.

- `ProcessingGroupParameters` are associated to a set `Schedulable` entities so to guarantee, to the group as a whole, that the scheduler will not be giving more

time per period than indicated by a cost. The use of this class is useful when it is necessary to bound the aggregated utilization of a pool of `Schedulable` entities.

2.2.2 Real-Time Java Threads

The RTSJ extends the existing Java threading model with two new types of real-time threads: `RealtimeThread` and `NoHeapRealtimeThread`. As depicted in Figure 2.5

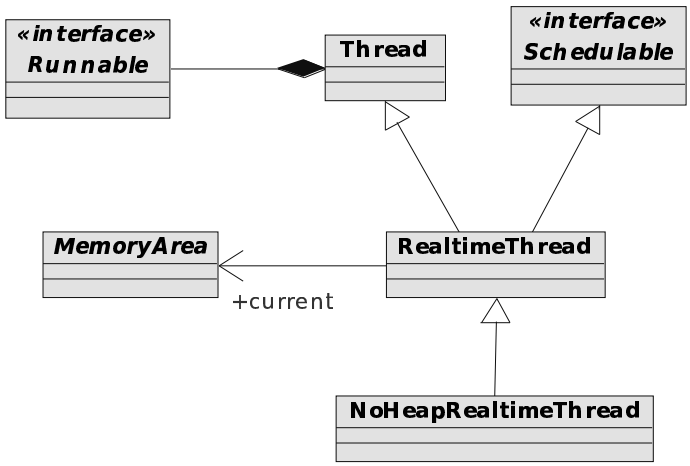


Figure 2.5: RTSJ Real-time Thread class hierarchy.

these threads are characterized by (1) being `Schedulable` entities, thus their execution is managed by the associated scheduler, and (2) having an associated `MemoryArea` instance. While the RTSJ memory subsystem will be described in great detail in Section 2.4, the RTSJ threading model is intimately related to the RTSJ memory model. It is impossible to explain one of the two in complete isolation of the other. Thus we provide a minimal discussion here. To understand the RTSJ threading model it is sufficient to know that the RTSJ introduces the concept of *memory area*; in RTSJ the Java heap is just one of the memory areas. These memory areas have different properties in terms of memory management and timing guarantees. The `MemoryArea` depicted in Figure 2.5 represents the current allocation context of the `RealtimeThread`, *i.e.* the region of memory from which objects are allocated. A Thread can change its allocation context dynamically, but the change is subject to some rules that are detailed in Section 2.4. In the remainder of this thesis, when referring to the `MemoryArea` which represents the allocation context of a thread, we will say that the given thread is *executing* within the `MemoryArea`.

The RTSJ threading model imposes restriction on the kind of memory area a thread can execute in, and as summarized in Table 2.1, the restrictions apply only to regular Java

`Thread` which can only execute in the heap, and the `NoHeapRealtimeThread` which, on the other hand, is not allowed to execute in the heap. It is worth noting that the restriction also apply to referentiality, *i.e.*, if a thread is not allowed to execute in a given kind of memory area, it is not allowed to hold references to objects allocated in that area. The

Table 2.1: Memory areas accessibility rules.

Thread Type	Accessible Memory Areas
`Thread`	Heap Only
`RealtimeThread`	All
`NoHeapRealtimeThread`	Heap Forbidden

reason why `NoHeapRealtimeThread` instances are not allowed to access the heap is to ensure that for this class of threads is safe to preempt the heap garbage collector. Thus, `NoHeapRealtimeThread` won't suffer from the unpredictable preemption latencies that a garbage collection might induce.

2.3 Synchronization

The RTSJ strengthens the semantics of Java synchronization for use in real-time systems by providing a way of performing priority inversion control. As shown in Figure 2.6 a `MonitorControl` class is defined as the superclass of all such execution eligibility control algorithms. `PriorityInheritance` is the default monitor control policy; the specification also defines a `PriorityCeilingEmulation` option. Another enhancement

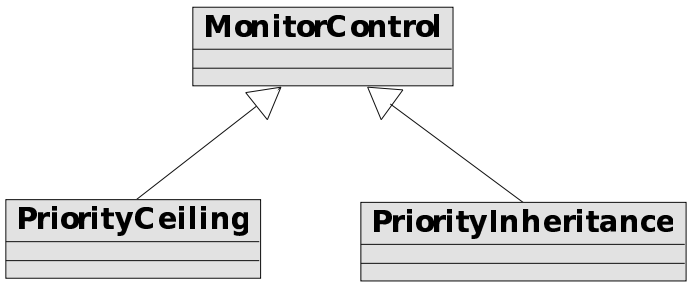


Figure 2.6: RTSJ Synchronization control classes.

provided by the RTSJ are read and write wait-free queues. These structures are useful for making it possible for `RealtimeThread` instances to cooperate with regular Java^{TM} `Thread` instances.

2.4 The RTSJ Memory Subsystem

The RTSJ extends the Java memory model by providing memory areas other than the heap. As shown in Figure 2.7, these memory areas are characterized by the anticipated lifetime of the contained objects (immortal, scoped) as well as the time taken for allocation (linear, variable). Objects allocated within the (singleton) *Immortal Memory* have the same lifetime

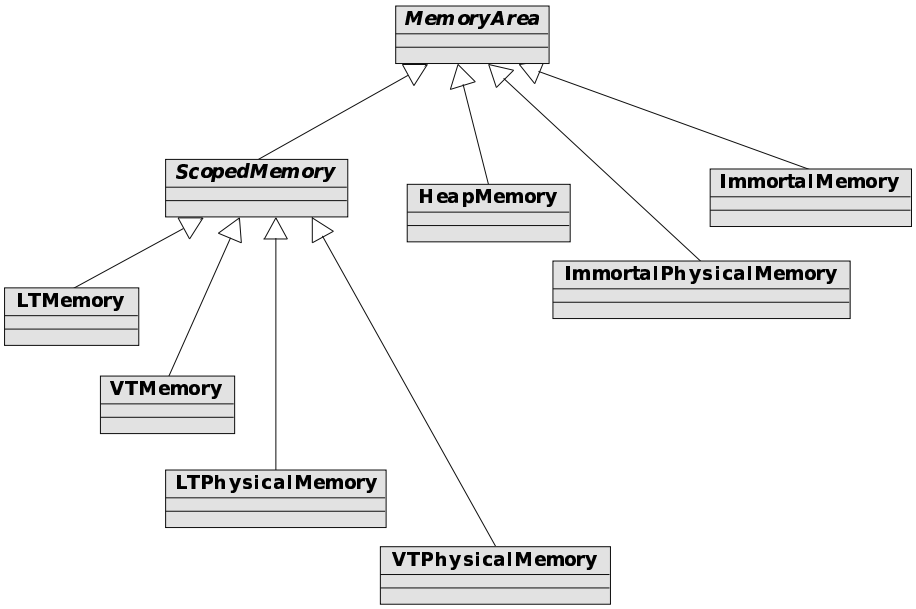


Figure 2.7: Hierarchy of Classes in the RTSJ Memory Model

as the application: they are never collected. Each *scoped memory* area is equipped with a reference count of the number of threads active in its area. The lifetime of objects allocated in such an area is keyed to the reference count. Figure 2.7 includes the `LTPhysicalMemory` and `VTPhysicalMemory` areas, which afford raw access to specific locations in an address space. We mention the physical memory areas for completeness; it is their scoped nature that is relevant to the work presented in this thesis.

Additionally, scoped memory areas provide bounds on the allocation time; currently, variable (`VTMemory`) and linear-time (`LTMemory`) allocators are accommodated. For linear allocation time, the RTSJ requires that the time needed to allocate the $n > 0$ bytes to hold a class instance must be bounded by a polynomial function $f(n) \leq Cn$ for some constant $C > 0$.[4]

For JVM and application developers alike, scoped memory is one of the more interesting features added to Java^{TM} by the RTSJ. Object allocated within a scoped memory are

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

not garbage collected; instead, a reference-counting mechanism detects when all objects in a scope should be collected. Safety of scoped memory areas is ensured by reliance upon (1) a set of rules imposed on a thread's entrance of scoped memories, and (2) a set of rules that govern the legality of reference between objects allocated in different memory areas. The remainder of this section concerns memory areas and will provide an explanation of its mechanics.

2.4.1 Understanding the Scoped Memory Model

To understand the mechanics of scoped memory areas, it is important to understand the RTSJ rules that govern access to those areas. As described in Section 2.2 the RTSJ assumes that Java^{TM} has its traditional threads, but adds two new real-time thread types: `RealtimeThread` and `NoHeapRealtimeThread`, with access rules as follows:

1. A traditional thread can allocate memory only on the traditional heap.

2. Real-time threads may allocate memory from a memory area other than the heap by making that area the current allocation context.

3. A new allocation context, or scope, is entered by calling the `MemoryArea.enter()` method or by starting a real-time thread whose constructor was given a reference to an instance of `MemoryArea`. Once a scope is entered, all subsequent uses of the `new` keyword, within the program logic, will allocate the memory from the current scope. When the scope is exited by returning from the `enter()` method, all subsequent uses of the `new` operation will allocate memory from the memory area associated with the enclosing scope.

4. A real-time thread is associated with a scope stack containing all the memory areas that the thread has entered but not yet exited.

On the other hand, the rules that govern the scoped memory behavior are the following:

1. Each instance of the class `ScopedMemory` must maintain a reference count of the number of threads active in that instance.

2. When the reference count for an instance of the class `ScopedMemory` is decremented from one to zero, all objects within that area are considered unreachable and

are candidates for reclamation. The finalizers for each object in the memory associated with an instance of `ScopedMemory` are executed to completion before any statement in any thread attempts to access the memory area again.

3. Each `ScopedMemory` has at most one *parent*, defined as follows. For a `ScopedMemory` that has been pushed on a scope stack, *i.e.*, entered by at least one thread, its parent is the first instance of `ScopedMemory` below it on the scope stack, if there is one; otherwise, its parent is the *primordial scope*. For as scope not pushed on the scope stack, its parent is null.

Figure 2.8 depicts three scoped memory areas, A, B, and C, and two real-time thread T_1, and T_2. In Figure 2.8, T_1 enters A, B, and then tries to enter C, while T_2 enters A, C, and then tries to enter B. In Figure 2.8, circles represents scoped memories while arrows point from a child scope to its parent scope.

If T_1, as shown in Figure 2.8, tries to enter C after T_2 has entered it, than a compliant RTSJ JVM will detect a violation of the single parent rule and throw an exception. This violation, as visible in Figure 2.8 by the contents of the scope stack of T_1 and T_2, can be detected by a JVM inspecting the scope stack and checking that no single-parent rule violation happens.

Why is this single-parent rule necessary? The single parent rule guarantees that once a thread has entered a set of scoped memory in a given order, any other thread will have to enter them in the same order, up to the point at which the reference count for all these memory drops to zero. At that point, a new nesting will be possible. This requirement guarantees that a parent scope will have a lifetime that is at least that of any of its child scopes, making it safe for objects in a descendant scope to reference objects in an ancestor scope.

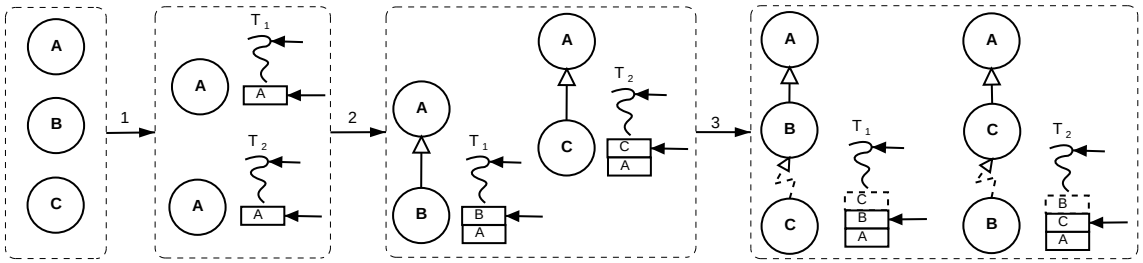


Figure 2.8: The scope stack and the single parent rule.

Figure 2.9 extends the example of Figure 2.8, showing a potential scope tree of an RTSJ application; all nodes of that tree represent scoped memory areas. An object in node

x of such a tree can reference an object in node y only if y is an ancestor of x. Thus, the curved arrows in Figure 2.9 show some (not all) legal references, while the zig-zag arrows represent some (not all) illegal references.

With the single-parent rule, the ancestor relationship described above guarantees that legal accesses occur only from a scope to another at least as long-lived as the former. In other words, no legal references are "dangling."

To enforce the rules, a compliant JVM has to check every attempt to enter a memory area by a thread, to ensure that the single parent rule is not violated, and it has also to check the creation of reference between objects belonging to different memory areas. Since object references occur frequently in Java^{TM} programs, it is important to implement the checks efficiently and predictably.

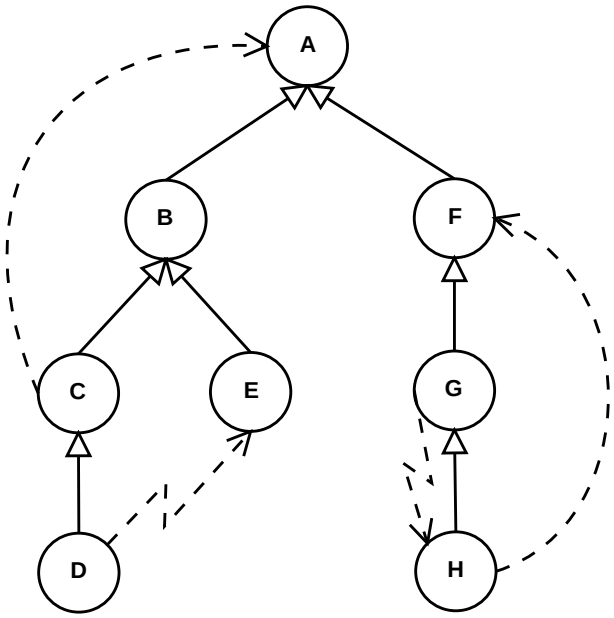


Figure 2.9: Scope Tree and Scoped Memory Reference Checking Sample.

2.4.2 RTSJ Suggested Runtime Check Implementation

We next present the current state of algorithms for scope and reference checks, as suggested by the RTSJ and its current implementations [21, 5]. we provide examples that explain why this approach is not satisfactory for real-time applications.

Algorithm 1: checkSingleParentRule

```
Input: MemoryArea ma, ScopeStack scopeStack
Output: boolean isSingleParentRuleOK
begin
    isSingleParentRuleOK ← true;
    if ma instanceOf ScopedMemory then
        parent = findFirstScope (scopeStack);
        if ma.parent = nil or ma.parent = parent then
            ma.parent ← parent ;
            scopeStack.push (ma );
            ma.refCount ← ma.refCount + 1;
        else
            isSingleParentRuleOK ← false ;
end
```

Data Structures

The RTSJ assumes that (1) there is a scope stack associated (at least logically) with each real-time thread, (2) each memory area keeps a reference to its parent, (3) scoped memories keep track of their reference count, and (4) for any object, it is possible to obtain a reference to the memory area that contains it. The algorithms used to enforce the single-parent rule and the assignment rules are based on these data structures.

Single Parent Rule

The single parent rule is enforced at the point a real-time thread tries to enter a scope s. At that time, if s has no parent, then entry is allowed. Otherwise, the thread entering s must have entered every proper ancestor of s in the scope tree. Algorithm 1 and Algorithm 2 contain the pseudocode that performs this test.

Examination of these algorithms reveals a time complexity of $O(n)$ where n represents the depth of the stack.

Memory Reference Checks

The rules that govern the validity of references across memory areas can be summarized as follows:

1. A reference to an object allocated in a `ScopedMemory` can never be stored in an object allocated in the Java heap or in the immortal memory.

Algorithm 2: findFirstScope

```
Input: ScopeStack scopeStack
Output: ScopedMemory firstScope
begin
    firstScope ← PrimordialScope ;
    for i ← scopeStack.size()−1 downto 0 do
        if scopeStack [i] instanceOf ScopedMemory then
            firstScope ← scopeStack [i];
            break ;
end
```

2. A reference to an object allocated in a `ScopedMemory` m can be stored in objects allocated in a `ScopedMemory` p only if p is a descendant of m; note that the case $p = m$ is thus allowed.

The RTSJ specification does not mandate any particular algorithm for checking the legality of a memory reference, but most implementation [35, 5], follow the advice given in the RTSJ specification. In the algorithm suggested by the RTSJ, a thread's scope stack has to be scanned to ensure that the memory area from which we are creating a reference was pushed later than the memory area of the reference's target. This approach is described by the Algorithm 3 and Algorithm 4. By inspection of the pseudocode, this check has time complexity $O(n)$ where n is the depth of the scope stack.

2.5 Asynchrony

The RTSJ defines mechanisms to bind the execution of program logic to the occurrence of internal and/or external events such as interrupts and POSIX signals. In particular, the RTSJ provides a way to associate an asynchronous event handler to some application-specific or external events. As shown in Figure 2.10 there are two types of asynchronous event handlers defined in RTSJ:

- The `AsyncEventHandler` class, which does not have a thread permanently bound to it—nor is it guaranteed that there will be a separate thread for each `AsyncEventHandler`. The RTSJ simply requires that after an event is fired the execution of all its associated `AsyncEventHandler`s will be dispatched.

Algorithm 3: checkReferenceValidity

```
Input: MemoryArea from, MemoryArea to, ScopeStack scopeStack
Output: boolean validReference
begin
    validReference ←true;
    if from ≠to then
        if to instanceOf ScopedMemory then
            if from instanceOf ScopedMemory then
                toDepth = depth (to, scopeStack );
                if toDepth = inf then
                    validReference ←false;
                else
                    fromDepth = depth (from, scopeStack );
                    deltaDepth = toDepth − fromDepth ;
                    if not (0 <deltaDepth < inf) then
                        validReference ←false;
            else
                validReference ←false;
end
```

- The `BoundAsyncEventHandler` class, which has a real-time thread associated with it permanently. The associated real-time thread is used throughout its lifetime to handle event firings.

Event handlers can also be specified a *no-heap*, which means that the thread used to handle the event must be a `NoHeapRealtimeThread`. Finally it is worth noting that the `AsyncEventHandler` class is a `Schedulable`, thus its execution is managed by the platform scheduler.

The RTSJ also introduces the concept of *Asynchronous Transfer of Control* (ATC), which allows a thread to asynchronously transfer the control from a locus of execution to another.

2.6 Time and Timers

The (standard) Java^{TM} platform provides only limited support for measuring time and performing time-driven operations. On the other hand, in real-time embedded systems, time and time-driven operation have a central role. In these systems, timers are often used to

Algorithm 4: depth

```
Input: MemoryArea ma, ScopeStack scopeStack
Output: int depth
begin
    depth ← inf;
    index ← scopeStack.size() - 1;
    while index > 0 and scopeStack [index] ≠ ma do
        index ← index −1;
    if scopeStack [index] = ma then
        depth ← scopeStack.size()−index −1;
end
```

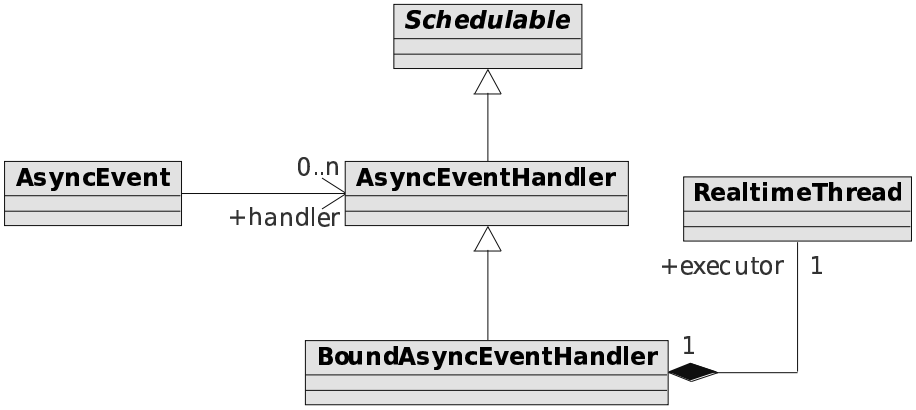


Figure 2.10: RTSJ Asynchronous Event Class Hierarchy

perform certain actions at a given time in the future, as well as at periodic future intervals. For example, timers can be used to sample data, play music, transmit video frames, etc. The RTSJ introduces the `Clock` abstraction, as well as a set of classes that represent absolute time, time intervals and frequency (see Figure 2.11. A clock advances from the past, through the present, into the future. It has a concept of "now" that can be queried, a `getTime()` operation, and it can have events queued on it which will be fired when their appointed time is reached. The class `Clock` shown in Figure 2.11 might be subclassed for representing different kind of clocks such as real-time clocks, user time clocks, simulation time clocks. In order to provide a means of performing time driven operation the RTSJ provides two types of timers (see Figure 2.12):

- `OneShotTimer`, which generates an event at the expiration of its associated time interval and
- `PeriodicTimer`, which generates events periodically.

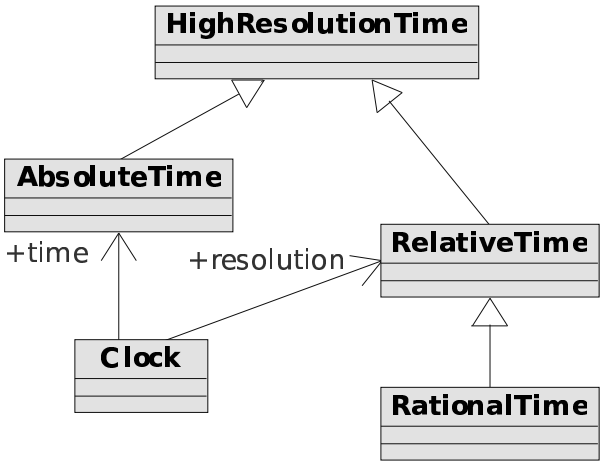


Figure 2.11: RTSJ Timer Class Hierarchy

Timers can be can be armed for any `Clock` instance. At expiration `OneShotTimer`s and `PeriodicTimer`s events are handled by registered `AsyncEventHandler`s.

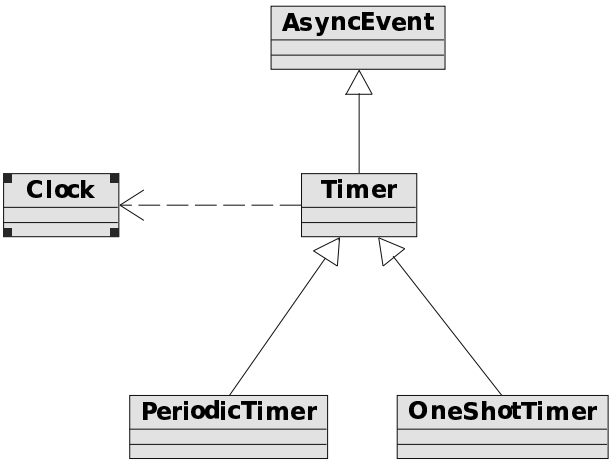


Figure 2.12: RTSJ Timer Class Hierarchy

Chapter 3

Patterns for Real-Time Java Programming

3.1 Introduction

In Chapter 2 we have provided an overview of the RTSJ's features. In this chapter we'll provide a catalog of Real-Time Java idioms, patterns, along with a series of clarification on the RTSJ programming model. Cataloging patterns is a very important activity since it (1) promotes good engineering practices, (2) improves developers understanding of a given technology, (3) establish a vocabulary that designer can use to synthetically communicate design decision when building new systems, (4) provide ready to use solutions to recurring problem in a given context.

Since real-time Java is a fairly new technology, there are currently no available pattern catalogs. Times have matured, for people like the authors, who have been involved in the development of Real-Time Java almost from its inception, to make available their experience by means of patterns.

The remainder of this Chapter is organized as follows, in Section 3.2 we provide an explanation of the RTSJ programming model, and provide rationales on how to organize RTSJ applications; in Section 3.3 we introduce some new Design Patterns or revisit some patterns from an RTSJ perspective.

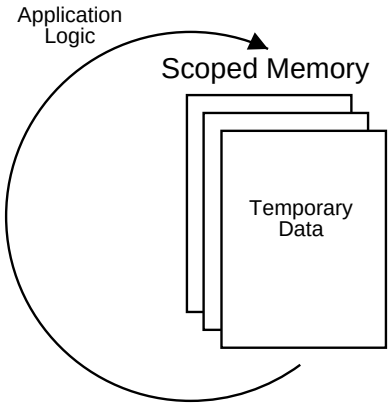
3.2 Understanding the RTSJ Programming Model

The RTSJ aims at specifying a viable platform for any real-time application environment, these promises are not necessarily delivered by the programming mechanism it provides. In fact, for a real-time application to take advantages of the Java^{TM} safety properties, it should be designed and coded so to rely on the automatic memory management facilities provided by scoped memories. This requires an effort both at design and coding level since, as shown in Section 3.3, RTSJ programming is not the same as Java^{TM} programming, and anyone who will try to program an RTSJ application *a la* Java^{TM} is deemed to fail in delivering the needed real-time characteristics. For this reason, in our view, it is important to identify the class of (hard) real-time applications [1] which properly fit in the RTSJ programming model, and those that require to twist it. Providing the user community with such a catalog will make it easier for both software analyst, designer and programmers to decide whether or not the RTSJ is the right tool to use, and how to use it. This said, the class of application can be classified with respect to the RTSJ as described below.

Class I: Stateless Applications. These applications don't have any state. The output produced by these applications depends entirely on their input. For instance a simple Web Server belongs to this category. This application class is characterized by the fact that all the memory allocated while computing a result can be discarded at the end of the computation. Figure 3.1 shows how this class of applications can be implemented by relying only on scoped memory. In fact since the application does not have any *persistent* state, all allocations can happen in scoped memory.

Class II: Applications with a Finite and Immutable Set of States. This class of applications are well described by finite state automaton. To each state of the automaton corresponds a configuration or operational modes of the application itself. These configurations don't change over time and are completely known at design time. Example of this class of application are control software, such as flight control software, which switch from an operational mode to another depending on the height, phase of flight etc. Figure 3.2 shows how this class of applications can be implemented by relying on immortal memory to store the persistent application state, and on scoped memory for performing computation.

[1] Here on with real-time application we intend any application which cannot experience unbounded or unpredictable preemption latencies.



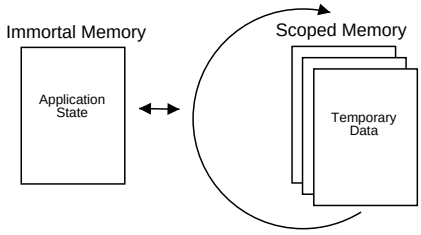

```
1  import javax.realtime.*;
2  public class MyStatelessApp {
3    ...
4    LTMemory memory_area =
5        new LTMemory (SIZE, SIZE);
6    for (;;) {
7      // Read the inputs
8      memory_area.enter (new Runnable () {
9        public void run () {
10          ...
11          // Compute the result using the inputs
12          ...
13       }
14     }); // Here scoped memory is exited and
15          // thus all object used during the
16          // computation are discarded
17     // Provide the results
18   } // End for
19  ...
20 }
```

Figure 3.1: Stateless Application Class.

Class III: Applications with a Time Dependent State. This class of application contains any application which does not follow in the previous categories. In this application class, state changes often require memory allocations. The memory allocated is required to hold the new state configuration. An example of this application class can be as simple as a real-time application that maintains a list of targets that have entered a monitored area, but not yet exited it. In this case the targets could represents flight that have to be tracked, or missile that constitute a threat for our system. In this scenario, the application state is represented by the set of target under our control, plus some other application specific data. Since the set of targets changes with time, it should be clear that there is not easy way to map allocate them on memory scopes while maintaining automatic memory reclamation.

3.2.1 Coping with RTSJ Programming Model Limitations

As shown in the previous Section, each of the application classes I and II are amenable to a particular memory organization. On the other hand, real-time applications that belong to the class III do not fit properly in the RTSJ programming model. The questions that we should pose ourselves at this point are, (1) can we develop class III applications with the RTSJ? (2) How can we evaluate if the additional complexity makes the RTSJ still preferable to languages such as C/C++? The answer to question (1) is certainly yes. It is easy to see that the RTSJ is Turing complete, thus any application can be written using it. At the same




```
1  import javax.realtime.*;
2  // Class containing the application state.
3  // Instances of this class are allocated in
4  // Immortal Memory
5  public class AppState {
6    //...
7  }
8  public class MyFiniteStateApp {
9
10   private AppState currentState;
11
12   LTMemory memory_area =
13       new LTMemory (SIZE, SIZE);
14   for (;;) {
15     // Read the inputs
16     memory_area.enter (new Runnable () {
17       public void run () {
18         ...
19         // Compute the result using the
20         // inputs using the proper state
21         ...
22         // Set new state if necessary
23       }
24     }); // Here scoped memory is exited and
25         // thus all object used during the
26         // computation are discarded
27     // Provide the results
28   } // End for
29  ...
30 }
```

Figure 3.2: Finite State Application Class.

time, it is true that assembly languages are Turing completes, yet, we seldom use them to write applications. The key point here is that in order to develop class III applications, we have to recur so some form of manual memory management, such as those shown in Section 3.3. This defeats one of the main advantages of Java^{TM}—its automatic memory management.

From a pragmatic perspective, we could state that it is worth developing a class III application using the RTSJ only if the portion for which manual memory management is needed is very small compared to the rest of the application. For instance, assuming that the application can be subdivided in a series of application components which belong to class I or II, and a few application components that belong to class III. Then designing the overall application in RTSJ might still make sense in the case in which the number of class I and II application components is greater than the number of class III application components. Clearly this statement assumes that applications components have roughly the same complexity, otherwise some sort of weigh should be used.

3.2.2 Case Study: The Mars Rover 7

The problem of developing class III applications was faced, at Jet Propulsion Laboratories of NASA in Pasadena, by the "Golden Gate Team", who is applying the RTSJ to develop control software for their 6-wheel experimental Mars rover (the "Rocky 7") [7]. Their proposal is based on using only one scoped memory area and thus performing object allocation and deallocation "manually" by relying on *memory pools*.

A memory pool is a (limited) set of pre-allocated objects, each one containing a *mark* to indicate whether the object is begin used or not. Creating a new object implies to find the first unmarked object in the pool and to mark it; on the contrary, releasing an object implies to remove the mark. This mechanism is depicted in the example of Figure 3.3.

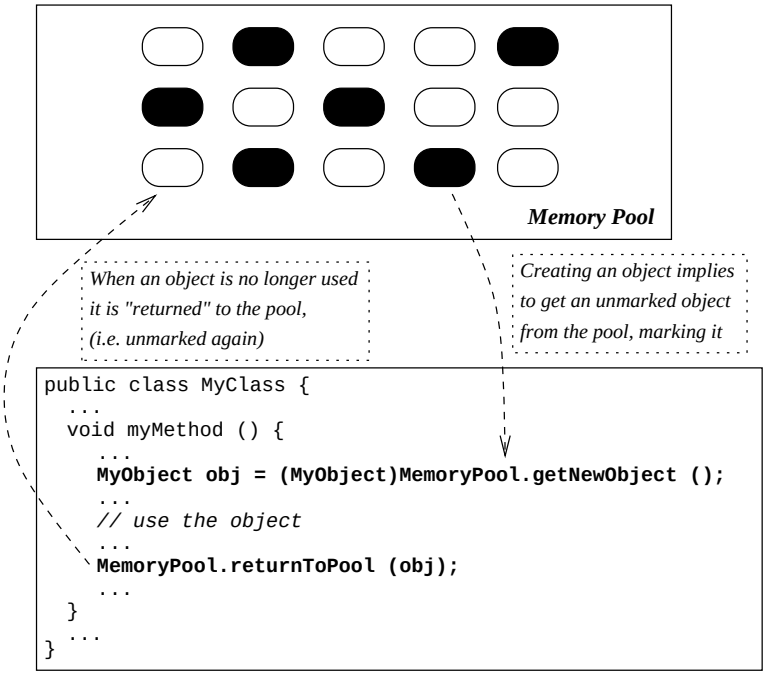


Figure 3.3: A memory pool and its usage

This solution is very interesting and also quite effective since allows application design with RTSJ without having to fight against the constraints of the memory model. However it presents some problems.

First of all, this solution requires to patch the source code to make it "pool-aware": each "`new`" operation, relevant to objects managed with this mechanism, must be replaced by a method call to the memory pool, aiming at obtaining an unmarked object (the `getNewObject()` method in Figure 3.3); moreover, another invocation to the memory pool must be added when the object is no more needed (`returnToPool()` in Figure 3.3).

Such patches are obviously possible when the source of the code that has to use memory pools is available; but if we need a library that is provided only in ".class" form, the problem is once again infeasible.

The second problem is tied to Java^{TM} semantics. Even if the memory pool approach makes possible the implementation of a large class of applications in RTSJ, the solution is, in our opinion, an *evident violation of Java^{TM} language semantics*: one of the most important characteristics of Java^{TM}—the *safeness*—completely disappears since we, as programmers, are committed not only to perform explicit memory management but also to take care that an object, once released to the pool, must be really no more used[2]. Without the safeness characteristic, the advantages of using Java^{TM}, with respect to C and C++, become minor and less evident.

3.3 Design Patterns

Now that we have understood what are the classes of application that nicely fit to the RTSJ programming model and which are those that require a stretch, it is worth moving our attention to the design patterns that can help us in writing robust RTSJ applications and which prevent us to fall in some common pitfall. In the remainder of this section we will provide an overview of some new RTSJ Design Patterns, as well as a re-interpretation of existing pattern into a RTSJ perspective.

3.3.1 Singleton

The Singleton Pattern, made popular by the famous GoF book [23], exposes some interesting issues in an RTSJ environment. A description of the pattern and the extension needed to make it applicable in an RTSJ context are reported below.

Intent. Ensure a class only has one instance, and provide a global point of access to it.

Example. In some application there are some abstraction which have only one associated instance. For example in a windowing system there would be a single window manager, or in a real-time system it is very likely that there is a single scheduler managing the various tasks execution. In all the cases in which there is an abstraction that has to have a single

[2]Please note that, while in C/C++ a dangling pointer could be discovered since its use often causes a "segmentation fault", identifying a block of Java ^{TM} code that improperly uses an unmarked object is very hard, because the latter is always alive in memory and no exception is raised.

instance it is a good design practice to (1) enforce, by design, that no more than one instance of the given class can be created, and (2) provide a global access point to it.

Problem. A design mechanism is needed in order to enforce that a given class has at most one instance, and this singleton instance is accessible via global access point.

Solution. Define a class method (*e.g.* static for Java and C++ programmers) that allows to get the unique instance of the class. Declare the constructor of the class protected and give the responsibility of creating the class to the class itself or a friend factory.

Structure. Figure 3.4 depicts a UML diagram which describes the implementation of the Singleton Pattern.

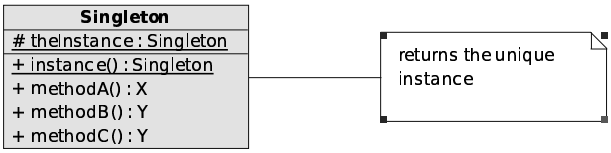


Figure 3.4: The Singleton Pattern.

Implementation. The typical Singleton Java implementation is shown in Figure 3.5. What is the problem with this way of implementing the Singleton Pattern? The Single-

```
public class Singleton {
  protected static Singleton theInstance;

  public static synchronized void instance() {
    if (theInstance == null) {
      theInstance = new Singleton();
    }

    return theInstance;
  }

  // Other methods
}
```

Figure 3.5: A typical Singleton implementation in Java

ton implementation shown in Figure 3.5, works perfectly fine in a Java environment, but

respect to the RTSJ programming model has a fatal flaw. The problem is in the **new** executed at the line 6 of the listing. The issue is that the Singleton instance allocated by that instruction will be placed in the current memory area of the calling thread. This is a big problem since static member of a class can only refer to objects allocated in the Heap or Immortal memory, and whenever a thread running within a scoped memory will happen to execute line 6 the runtime system will throw an invalid reference exception. Thus, it should be clear that the pure Java Singleton implementation is not robust in an RTSJ environment. A simple way of fixing the Singleton implementation and make it slightly more robust is

```
1 public class Singleton {
2   protected static Singleton theInstance = new Singleton();
3
4   public static void instance() {
5     return theInstance;
6   }
7
8   // Other methods
9 }
```

Figure 3.6: A possible RTSJ Singleton implementation.

to allocate the singleton object at class initialization time as shown in Figure 3.6. This way the Singleton object will be allocated in the JVM method memory (this is the memory region that contains all the constants and the class information). The problem, is that most of the JVM use as method memory the Heap. This means that for all those JVM that use as method area the heap, whenever a `NoHeapRealtimeThread` will try to use the Singleton instance, a runtime exception will thrown. Recall that `NoHeapRealtimeThread` are not allowed to allocate nor reference objects allocated in the heap.

Finally, Figure 3.7 reports the right way of implementing the Singleton pattern in RTSJ. Notice that in this case we explicitly allocate the Singleton instance in immortal memory. The code reported in Figure 3.6 could be also adapted to explicitly allocate the Singleton instance in immortal memory, however the author should be aware of the fact that this introduce a additional latency the first time the `instance` method is created, on the other hand, the solution provided in Figure 3.7 does not suffer of this problem. The lesson that should be learned by the RTSJ's Singleton implementation is that static field should be in general treated with great care in RTSJ. The programming idiom that should be used when coping with static field should be **Statics are Immortal**. This means that static field should be explicitly allocated in the RTSJ Immortal memory.

```
public class Singleton {
  protected static Singleton theInstance;

  static {
    ImmortalMemory im = ImmortalMemory.instance();
    theInstance = im.newInstance(Singleton.class);
  }

  public static void instance() {
    return theInstance;
  }

  // Other methods
}
```

Figure 3.7: A more appropriate RTSJ Singleton implementation.

3.3.2 Handle Exceptions Locally

Due to the RTSJ restrictions on memory management, and validity of references, designing code that uses structured exception handling to cope with erroneous situation is rather tricky, especially if the exception handling is done ala Java. The **Eager Exceptions Handling** pattern provides a way of using structured exceptions in RTSJ applications so to avoid the traps and pitfalls that the platform might induce.

Intent. Ensure that the use exceptions do not cause invalid memory references, avoiding the coupling of scope structure to the handling of structured exceptions. Avoid the unnecessary consumption of memory by exception handling code.

Example. In programming languages that supports structured exceptions, erroneous conditions are handled by raising a proper exception. The thrown exception encapsulates the kind of unexpected situation that has occurred. For instance, exceptions are used in the Java library for handling indexes out of bounds, I/O errors, and so on. While using exceptions in Java is rather straightforward, the RTSJ introduces several complications. For instance, due to the referencing rules imposed by the RTSJ memory model, exceptions should be handled either in the same memory area in which they were raised (allocated), or in a memory area from which is legal referencing the given exception object. For instance, the scenario depicted in Figure 3.8 seems at a first sight rather innocent, but actually it could have several problems, depending on how the underlying RTSJ implementation copes with exceptions. In the case in which the `memArea_0` is the `ImmortalMemory`, one of the

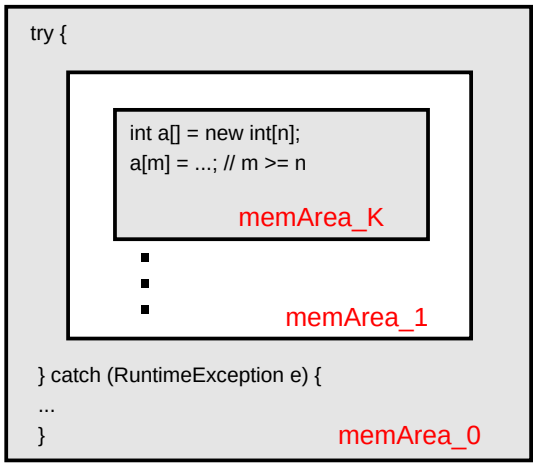


Figure 3.8: Runtime Exceptions in the RTSJ.

problem that could arise, in those RTSJ implementation that propagate exceptions from scope to scope, up to the scope within which the exception is handled, is that memory is allocated in `ImmortalMemory` for objects that are actually not immortal. This could lead to exhausting the `ImmortalMemory`. Another problem that might occur is that of creating an invalid reference exception (or a segmentation fault) in the case in which the RTSJ application does not take care of reallocating the same exception objects as it walks trough the scope stack.

The listing shown in Figure 3.9 is the pseudo code that implements the situation depicted in Figure 3.8. The pseudo language used in this thesis, in order to make it easier to understand the scoping of memory areas, uses a new keyword, **enter**, in order to represent that a given fragment of code is executed within the memory area passed an argument. For instance, in Figure 3.9 the code show at line 8 is executed within the memory area `memArea_K`.

Problem. The RTSJ does not mandate a specific way of coping with runtime exceptions that are thrown while executing within a memory area. Thus, exception handling in the RTSJ might have different behaviour based on the RTSJ implementation. This limits the portability of code that is not written carefully.

Solution. In order to write safe and portable RTSJ applications, exceptions should be handled in the same scope in which they are raised. Upon notification of an exceptional condition, the control has to be transfered to outer scopes, and the exception encountered has to be somehow propagated. Depending on the application, this can be achieved by

```
enter(memArea_0) {
  try {
    enter(memArea_1) {
      .
      .
      .
      enter(memArea_K) {
        a[N] = b;  // throws an ArrayOutOfBoundsException
      }
    }
  } catch(RuntimeException e) {
    // Handle Exception
  }
}
```

Figure 3.9: Problems with exception handling.

either catching the exception locally and re-throwing a singleton exception, or by using status objects or variables in the same fashion as the C/C++ `errno` variable to detect and propagate exceptional behaviour.

Implementation. The implementation of the pattern is rather straightforward. The rule is that all exceptions have to be caught and handled in the same scope in which they have been thrown. For instance, applying this pattern to the listing shown is Figure 3.9, we obtain the listing shown in Figure 3.10.

3.3.3 Memory-Area-Aware Factory

In the RTSJ memory model each occurrence of the new operator allocates memory from the *current* memory area, thus, unless some information is provided on the target memory area that should be used for allocating new instances of a given type, the current context will be used. One way to control the placing of certain types is to use a variation of the GoF Factory Pattern [23].

Intent. Provide an interface for creating families of objects without specifying their type nor their placement, i.e., memory area.

Example. Suppose you are writing a system in which both the real-time core and its GUI have the need of creating some concrete types through a factory. However, while the GUI will be run by non real-time thread and will be using the Heap, the application's real-time

```
enter(memArea_0) {
  try {
    enter(memArea_1) {
      try {
        .
        .
        .
        enter(memArea_K) {
          try {
            a[N] = b;  // throws an ArrayOutOfBoundsException
          } catch (RuntimeException e) {
            // Handle Exception
          }
        }
      } catch (RuntimeException e) {
        // Handle Exception
      }
    } catch(RuntimeException e) {
      // Handle Exception
    }
  }
}
```

Figure 3.10: Coping with runtime exceptions.

core will have to use no-heap real-time threads and thus rely only on immortal and scoped memory.

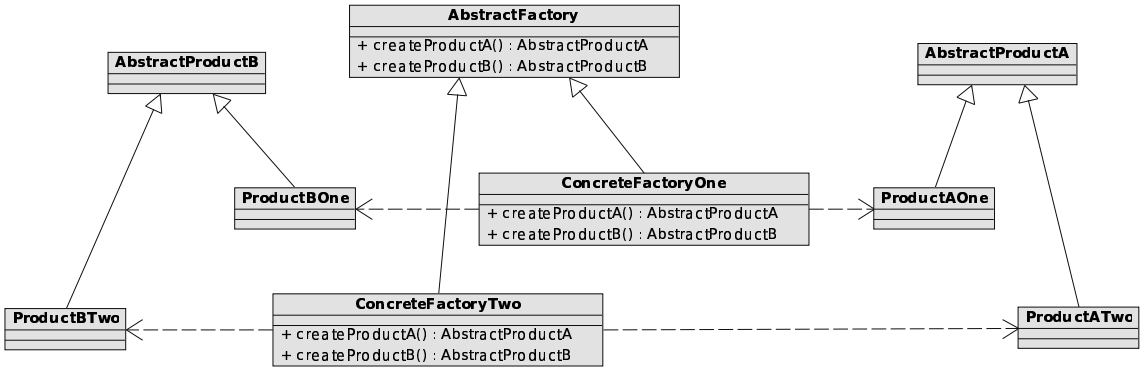


Figure 3.11: The Abstract Factory Pattern.

Problem. You want to create instance of concrete types by relying on the Factory pattern (see Figure 3.11), but at the same time you want the factory to allocate memory in different places based on the callee identity.

Solution. Use the Abstract Factory Pattern in synergy with the Strategy Pattern. The Strategy Pattern will be used to encapsulate the allocation strategy. Each time a request arrives to the Abstract factory it uses its strategy to decide where to allocate the newly created instance. The strategy can be programmed by the client.

Structure. The structure of the Memory-Area-Aware Factory Pattern is obtained by the plain Abstract Factory structure (see Figure 3.11) and adding the memory area selection strategy. The resulting structure is shown in Figure 3.12, while Figure 3.13 shows the typical sequence of message exchanged when creating a product (the example is shown for `ConcreteFactoryOne`.

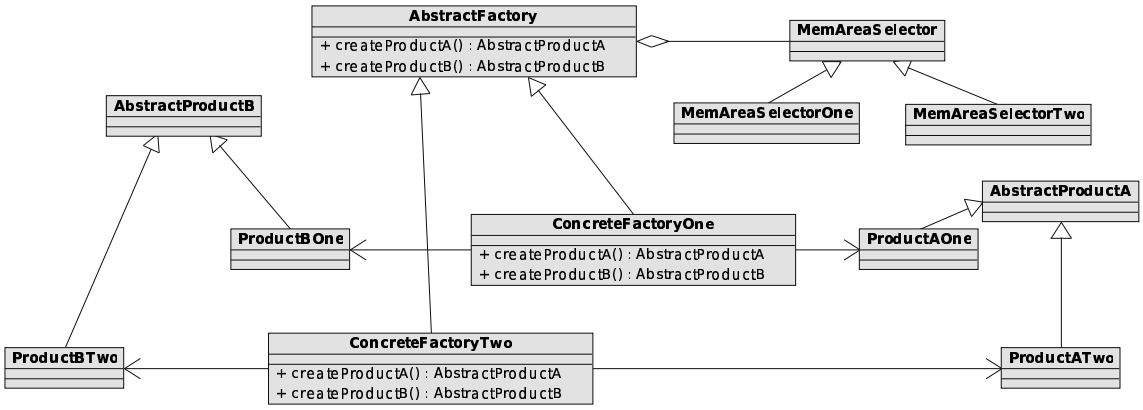


Figure 3.12: The Memory-Area Aware Abstract Factory Pattern.

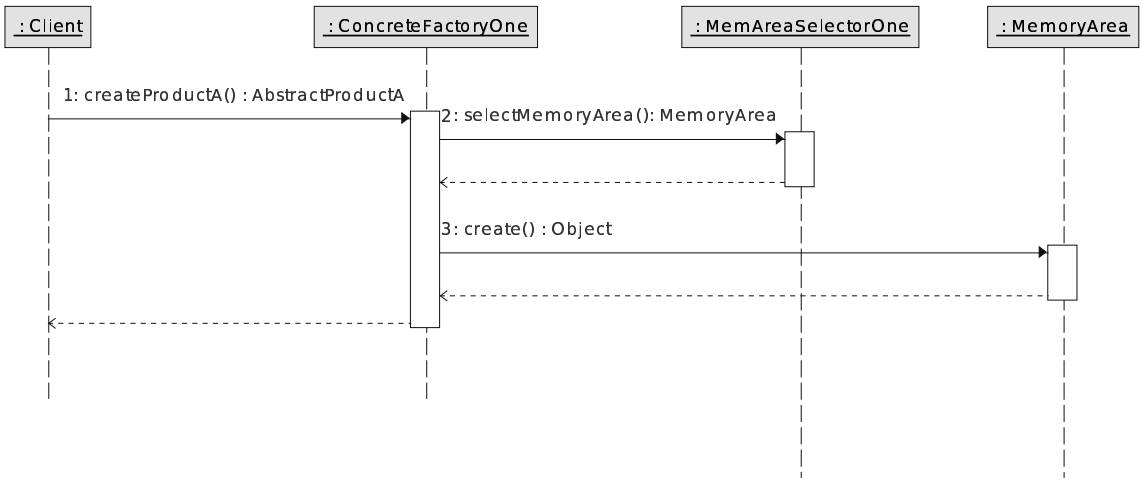


Figure 3.13: Memory-Area Aware Abstract Factory in action.

Implementation. The implementation of this pattern is the combination of the implementation of the Abstract Factory and the Strategy Pattern. Thus refer to [23] to see how to implement this.

3.3.4 Scoped Container

Java application take great advantage of the wide set of containers provided by the language. Since its inception Java provided abstraction such as `Vector`, `Stack`, `Hashtable` etc. However the use of this containers can be quite dangerous in an RTSJ environment. The key problem, once again, arise because of the differences between the RTSJ and the Java computational models.

Intent. Provide RTSJ safe containers.

Example. Assume you are developing a generic Java applications that has the need to store some information in a Java `java.util.Vector`. You want to be able to share this object between threads that are running in different memory scopes. This object has a thread safe interface, thus there no concurrency problem that can arise due to multiple threads trying to operate on it. However, each time a thread will call an operation on the container, the code will execute in the memory context of the caller thread. This might not be acceptable.

Problem. Most Java-based applications rely on containers. The main danger of using plain Java containers in RTSJ applications is that the containers code is always executed in the caller context. This means that if the container allocate some memory, for instance for resizing, this memory will be allocated in the current memory area of the calling thread. This might not be desirable.

Solution. Make the Java containers memomy area aware. Associate a memory area to each container instance and ensure that the memory allocated by the container is always taken by this memory area. This could be achieved, at least, by one of the two following way:

1. Modify the implementation of the container so that each memory allocation happens into the right memory area.

2. Use some sort of tagging for indicating to the compiler or the runtime system that each occurrence of the operator new within this class has to allocate memory from the same memory area in which the container was allocated.

The main difference between the first and the second solution is the trade off between the degree of human intervention on exiting code base and flexibility. Solution 1 requires the programmer to manually modify the code that implements the container and ensure that all allocation are performed in the right memory area. On the other hand, solution 2 simply tags a class, in a way similar to that used to declare a class serializable[3].

3.3.5 Scoped Leader Follower

The leader follower [42] is a is a well-known design pattern used to efficiently handle concurrent events coming from various sources. Because of the RTSJ memory model this pattern cannot be directly applied as described in [42], but it requires some non-trivial modifications. In the reminder of this section we will outline a possible specialization of this pattern for the RTSJ platform.

Intent. Efficiently multiplex and dispatch event from several sources.

Example. Suppose you have to implement a real-time network server that has to concurrently serve incoming requests from several different connections. Incoming requests will have to be demultiplexed efficiently and the resource has to be minimized, for instance because of the software running on a small embedded system. This problem is quite recurring in the design and implementation of Object Request Broker (ORB)s.

Problem. The problem to solve is to handle time-critical events, using RTSJ software, taking in particular into account the efficiency correctness and safety of the prosed solution.

Solution. The Scoped Leader/Follower pattern uses (1) a pool of threads allocated in a scoped memory area called the *pool scope*, and (2) a *leader selector* thread, running in that memory area, which iteratively picks the leader thread from the pool and activates it. Each thread of the pool is associated with another scoped memory area called the *handler scope*. This memory area is used to run the event handler.

[3] In Java in order to make a class serializable it has to implement an empty interface called `Serializable`

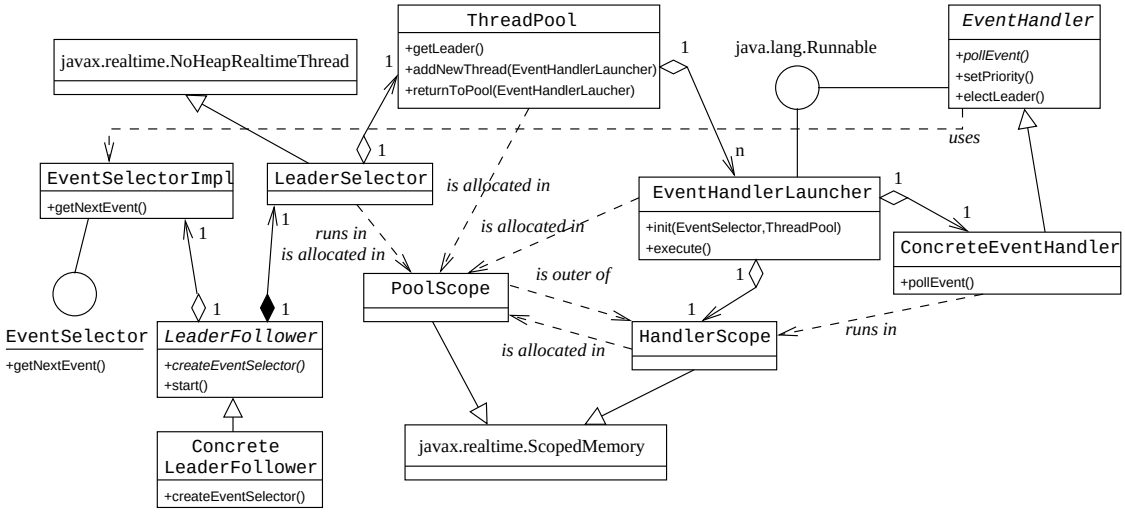


Figure 3.14: Class Diagram of the RTJ-Leader-Follower Pattern

To hide to programmers the details on memory management and scope entering/exiting, the threads of the pool act as *trampolines*, thus preparing the execution environment for user-defined objects that implement event capture and handling. For the same reasons, user-created threads cannot dynamically join to the pool, as happens in the standard Leader-Follower pattern, but all the threads of the pool, with the relevant scopes (and the pool itself), are created and allocated during startup.

All the threads (those of the pool and the leader selector) are `NoHeapRealtimeThread`s running with a preassigned priority. This is required since, before an event is caught, its criticality is unknown, therefore, the handler is first activated with the highest priority and then, once the event is caught and thus known, the priority is adjusted according to the urgency of the event itself. Using this "conservative" approach ensures that capturing and processing an event with a high priority is not delayed by any other activity.

The pattern includes an *event selector* object that provides the interface to the system that generates or receives the event.

Structure. The structure of the Scoped Leader/Follower pattern is depicted in Figure 3.14. The pattern is composed of the following classes:

- `LeaderFollower`. It is the responsible for starting the system, *i.e.* creating the thread pool and all the objects to be placed in the pool (threads, memory areas, etc.). It is also the main interface that a user has to control the system *e.g.* to change some parameters like priorities, etc. It is an abstract class because any realization must

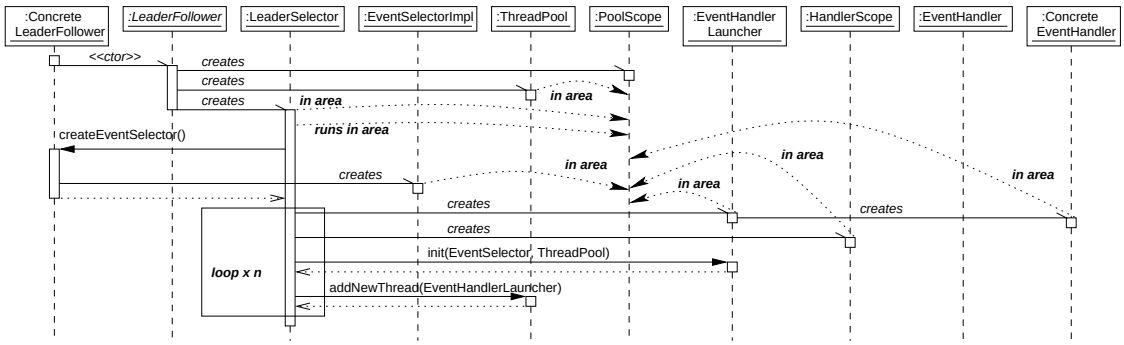


Figure 3.15: Sequence Diagram of the RTJ-Leader-Follower Pattern (Initialization)

define the `createEventSelector` method, needed to set up the *event selector* object.

- `LeaderSelector`. It is a `NoHeapRealtimeThread` responsible for selecting the leader from the thread pool, activating it.

- `EventSelector`. It is the interface that *event selector* objects must implement to be used in the RTJ-Leader-Follower.

- `EventHandlerLauncher`. Threads allocated in the pool are instances of this class. Each `EventHandlerLauncher` holds a reference to a scoped memory (the *handler scope*) and another to an instance of a concrete `EventHandler` class. The task of the `EventHandlerLauncher`, each time it is activated, is to enter the (concrete) `EventHandler` in the scope, to wait for `EventHandler`'s task completion, and finally to return itself to the thread pool.

- `EventHandler`. It represents the abstract class for the implementation of event handlers. Each concrete instance of this class (`ConcreteEventHandler`) is activated by the corresponding `EventHandlerLauncher` and has the task of capturing an incoming event, suitably processing it. It has also the responsibility for triggering new leader election.

- `PoolScope`. It is the memory scope in which the thread pool and all its objects are allocated. It is also the scope in which the `LeaderSelector` runs.

- `HandlerScope`. It is the memory scope that is associated to a `EventHandler-Launcher` and in which runs the (concrete) `EventHandler`.

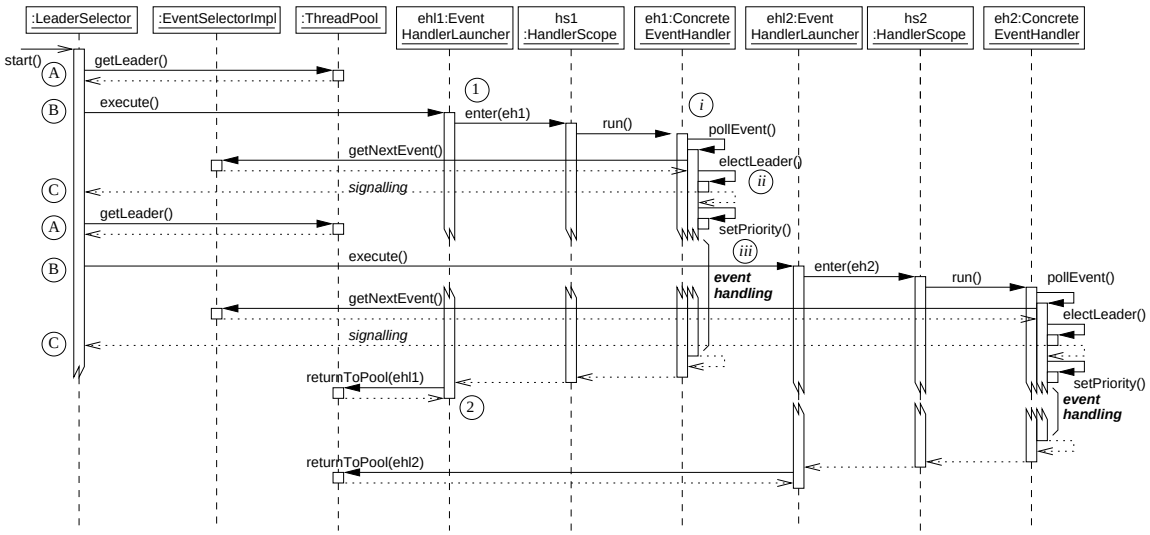


Figure 3.16: Sequence Diagram of the RTJ-Leader-Follower Pattern (Operation)

Implementation . Figures 3.15 and 3.16 report the sequence diagrams of the RTJ-Leader-Follower. As the Figures show, we consider two phases: *initialization* and *operation*.

In the former phase—*initialization*, see Figure 3.15—which is started when an object of class `LeaderFollower` is instantiated, all the objects that participate to the *operation phase* are created. Initialization phase, which is managed by classes `LeaderFollower` and `LeaderSelector`, is needed to set up memory areas and to properly allocate in them the objects of the pattern, in order to hide memory management details to the programmer. As Figure 3.15, first the `PoolScope` is created; then, all other objects are allocated in that memory area, i.e. the `ThreadPool`, the `EventSelectorImpl`), all the `EventHandlerLauncher`s, together with their `HandlerScope`s, and `ConcreteEventHandler`s. Finally the threads are added to the pool.

The latter phase—*operation*, see Figure 3.16—begins when the method `start()` of the `LeaderFollower` object is called. In this case, control is given to the `LeaderSelector` object that (A) selects the leader thread (`EventHandlerLauncher`) from the pool, (B) activates this thread by calling the `execute()` method, and (C) waits for a signal, which will arrive from the `ConcreteEventHandler` object, triggering reiteration and selection of the next leader.

Each leader thread (the `EventHandlerLauncher`), after being selected, (1) enters its `ConcreteEventHandler` object in the associated/relevant `HandlerScope`, and (2) when the handler terminates, return itself to the pool.

On the other hand, each `ConcreteEventHandler`, once activated by the `EventHandlerLauncher`, first (i) calls `pollEvent()` (that in turn invokes method `getNextEvent()` of the `EventSelectorImpl` object), then (ii), once it has acquired the events, triggers new leader election, and finally (iii) handles the event and adjusts the thread priority accordingly.

Chapter 4

Optimizing the RTSJ

4.1 Introduction

In the earlier Chapters we have illustrated the RTSJ, its programming model, and a set of design patterns that provide good programming practices, and good solutions to recurring problems. This Chapter, will highlight the limitation of the current RTSJ, and will devise solutions on how to overcome these limitations in order to get an efficient and predictable real-time platform. More specifically, this Chapter will (1) dig into all the points of shadow that are present in the RTSJ and limit its applicability to real-time systems, and (2) provide either optimal algorithm, or effective design on how to implement certain features. The content of this Chapter will be explained in the context of jRate an RTSJ-based ahead of time compiler and runtime system based on the GNU Compiler for Java (GCJ).

The remainder of this Chapter is organized as follows, in Section 4.2 we introduce jRate, its goals and its design principles; in Section 4.3 we provide and overview of the jRate Threading, Scheduling and Dispatching framework, in Section 4.4 we provide and overview of the jRate Generative Memory Area Framework, finally in Section 4.5 we will show why algorithms and data structures currently used by RTSJ implementation are not suitable, and will provide optimal algorithms for performing RTSJ's memory related safety checks.

4.2 jRate

4.2.1 Overview

jRate [1] [17, 18, 19, 3, 16, 33, 13, 15] is an open-source RTSJ-based real-time Java implementation that the author has developed in the course of his PhD while being at the University of California, Irvine and Washington University, St. Louis. jRate extends the open-source GCJ front-end and runtime system [24] to provide an ahead-of-time compiled platform for the development of RTSJ-compliant applications. The jRate architec-

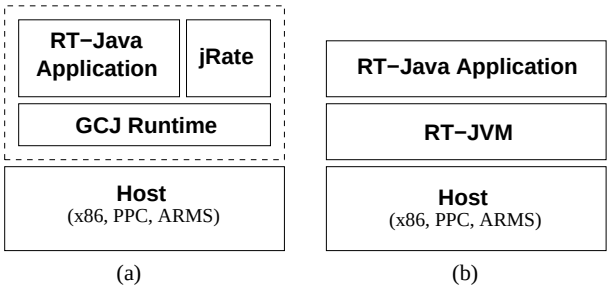


Figure 4.1: The jRate Architecture

ture shown in Figure 4.1(a) differs from the JVM model shown in Figure 4.1(b) since there is no JVM interpreting the Java bytecode. Instead, jRate ahead-of-time compiles RTSJ applications into native code. The Java and RTSJ services, such as garbage collection, real-time threads, and scheduling, are accessible via the GCJ and jRate runtime systems, respectively.

4.2.2 jRate's Design Principles

jRate uses Generative Programming (GP) [20] in order to make it possible to have a configurable, customizable and yet efficient RTSJ implementation. GP is also used as a way of exploring design alternatives, which differ from the RTSJ, in a well engineered manner. The generative behaviour is achieved by using a series of techniques and tools such as C++ template meta programming [20], and Python [34] scripting. For instance, Figure 4.2 depicts the stage involved in applying GP in an RTSJ setting. As shown in Figure 4.2, the *generation* of a specific instance of jRate is obtained by relying on:

- **Specification.** A specification is made by user input, and by platform specific information that is detected automatically by configuration scripts, such as CPU number

[1] jRate can be freely downloaded at `http://www.cs.wustl.edu/~corsaro/jRate`

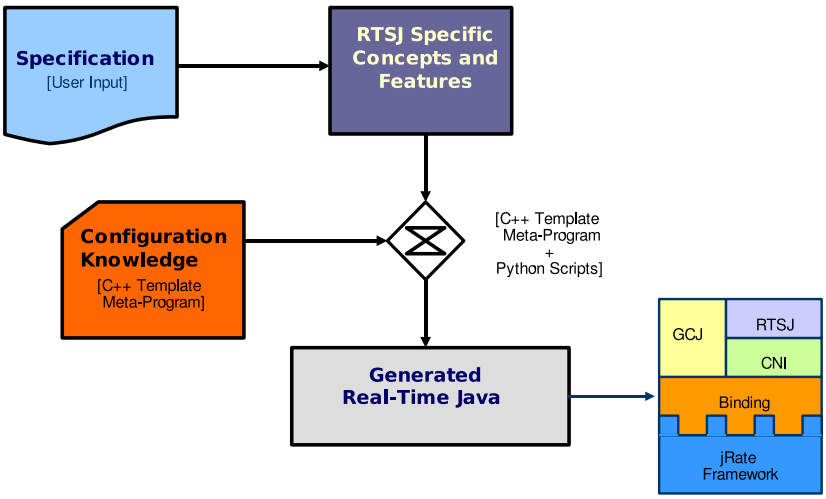


Figure 4.2: Applying Generative Programming to jRate.

and frequency, time and timer resolution, support for POXIX extension for real-time etc. The user input specifies properties of the generated platform such as the scheduling algorithm to be used, the peculiarities of the memory subsystem, and so on.

- **RTSJ Specific Concepts and Features.** These are a set of configurable components that provide key abstraction needed for building a RTSJ based system. Example of such concepts could be the scheduler, real-time threads, memory areas and so on.

- **Configuration Knowledge.** This the glue that makes the whole system stick together. Configuration knowledge encapsulates all the information needed in order to compose and configure the key components that provide implementation for RTSJ specific concepts and abstraction.

jRate implements RTSJ's *Specific Concepts and Features* by means of a set of C++ Template classes. This set of C++ Template classes, called the *jRate-Core*, provide a configurable kernel which can be reused in different settings and language binding by properly instantiating the template classes. As depicted in Figure 4.3, RTSJ applications developed using jRate rely on the GCJ runtime for basic Java services, and on the jRate runtime for RTSJ services. In order to make it easier to reuse the jRate code base, most of the extension that were needed by the GCJ runtime have been factored out and moved to the *jRate-Core*. As shown in Figure 4.3, the RTSJ binding represents just one instance of use of this C++ core, and by writing the proper binding, RTSJ-like abstraction could be provided to other languages such as C#, C++ etc. A big advantage of using C++ Template

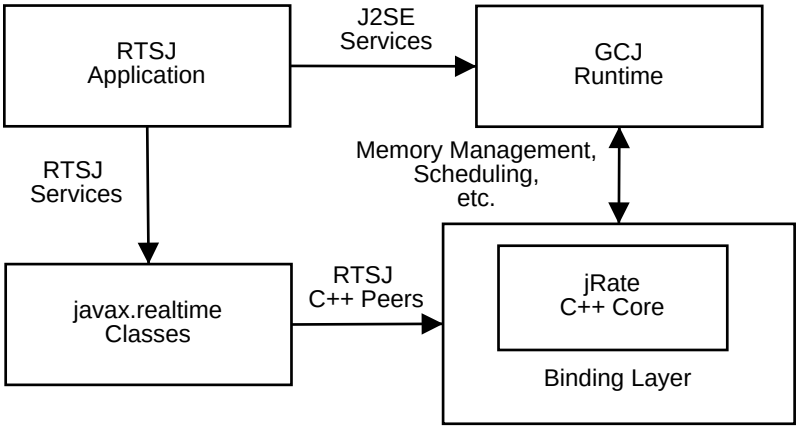


Figure 4.3: The jRate-Core and RTSJ binding

meta-programming, as a mean of implementing GP, is that we obtain a high degree of configurability, while at the same time being able to generate tight and efficient code.

4.2.3 jRate's Current Status

Currently jRate supports most of the RTSJ features such as memory areas, real-time threads, periodic threads, asynchronous event handlers, timers, POSIX signals, etc., and also provides some extensions. It allows a fine control over the properties of the different types of memory areas, such as size, allocators, locking etc. jRate provides several optimizations such a constant time memory reference checking, and single parent rule test implementation [13] and lock free dispatching of events [19]. The remainder of this Chapter will describe all the techniques used in jRate to implement efficient and predictable real-time middleware.

4.3 Techniques for Efficient RTSJ Threading, Scheduling and Dispatching Implementation

Real-Time systems usually heavily rely on threads their scheduling, and on the efficient dispatch of events. Thus, a real-time middleware cannot neglect the importance of providing an efficient, predictable and yet customizable design and implementation of the threading, scheduling and dispatching (TSD)[2] subsystem. jRate takes advantage of C++ templates

[2]In this context with the term *dispatching* we intend the demultiplexing and handling of system or application generated events.

to provide an efficient, predictable and configurable threading and dispatching service. In the remainder of this section we will provide an overview of the design of jRate TSD subsystem. We will also show some example of how this generic framework is instantiated to be applied in different context such as asynchronous event dispatching, and timers implementation.

4.3.1 Threads and Scheduling

The RTSJ provides a rich set of abstractions for threads and their scheduling (an overview of the services available was provided in Section 2.2). jRate's Core provides the basic building blocks used to implement the RTSJ's defined abstractions for threading and scheduling. Figure 4.4 shows the class diagram for jRate's threading and scheduling framework. The classes depicted in Figure 4.4 provide the key abstraction for managing threads execution, their scheduling, and their relationship with memory areas, *i.e.*, the scope stack. It is worth noticing that threads are parametrized by the platform scheduler, and that the scheduler is parametrized by the scheduling parameters. While this fixes the kind of scheduler to be used at compile time, it allows an efficient and strongly typed implementation of the scheduling framework. It is worth comparing this approach with the one followed by the RTSJ architects, since (1) the absence of templates in Java, along with (2) some design choices, lead the expert group to design some interfaces that don't take advantage of static typing for enforcing correctness. For instance, in the case of RTSJ's scheduling parameters, a run-time check is responsible to ensure the right parameters are being used for the current platform scheduler. On the other hand, in the jRate scheduling framework, the scheduler, scheduling parameters and thread are all tied at compile time. The right scheduler, and thread implementation is generated by means of static template meta-programming using C++ templates. This allows for efficient implementation as well as run-time correctness implied by compile time correctness. Below we provide a detailed description of the classes depicted in Figure 4.4.

- **Runnable**. The `Runnable` interface is used to define a contract between `Thread`s and the logic they will run. As is in the case for Java threads, the logic to be executed by a thread has to be embedded in the `run` method of a class that implements the `Runnable` interface.

- **Thread**. This class implements the core thread functionalities. It is parametrized by a `Scheduler` type, and has an associated `ScopeStack` instance and a `Runnable`. The `ScopeStack` is used to keep track of all the memory areas that the thread has

Figure 4.4: The jRate Threading and Scheduling Framework.

entered but yet not exited. The execution of the `Thread` instance is managed by the associated scheduler.

- **ScopeStack**. This class contains the memory areas that a `Thread` has entered and yet not exited.
- **PriorityScheduler**. This class represents one kind of concrete scheduler. Specifically it is a priority preemptive scheduler that relies on the OS scheduler.
- **PriorityParameters**. This class defines the scheduling parameters that have to be used to specify execution eligibility for a `PriorityScheduler`.

Threads are a core building block for many different kind of applications. Because of this jRate extends the thread abstraction by providing both in the core, and also in the Java binding, higher level abstractions such as executors and channels. In the remainder of this Section we will provide an overview of the extended abstraction provided by jRate and how this are used to implement the dispatching framework.

4.3.2 Dispatching

Event dispatching is one of the very fundamental activity that take place for performing several tasks in real-time and non real-time systems. As an example think about the event being dispatched to an application by the windowing system, or the event due to timer expiration or any other user defined activity. Event demultiplexing and dispatching is particularly important in real-time systems since this has to be performed while (1) limiting, or avoiding if possible, the introduction of priority inversion, and (2) being efficient and parsimonious in the use of resources. To solve this problem once, and in a generative manner, jRate provides the core abstraction in a dispatching framework. The class diagram of the classes that implement this framework is shown in Figure 4.5. While at first sight the framework seems to have a limited set of classes, the reader should concentrate on the composability of the core abstractions. As shown in Figure 4.5, the core abstraction are *executors* and *channels*. The basic idea is very simple, channels contain items, this items are sorted based on a policy that depends on the channel implementation. On the other hand executors simply executes tasks, where tasks are represented by `Runnable` instances. These simple and orthogonal concepts can be composed to created rather complex logic. Before providing some example of the instantiation of this generic dispatching framework to solve specific problems, we provide a description of the classes depicted in Figure 4.5.

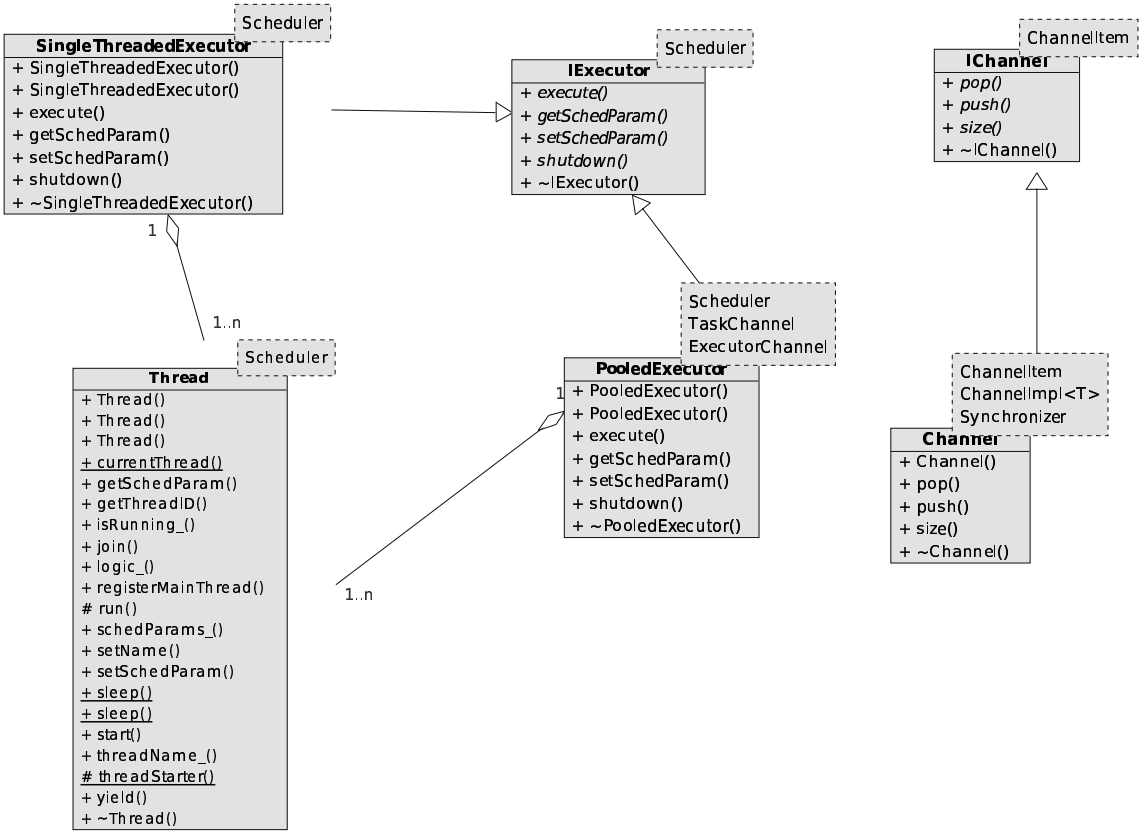


Figure 4.5: The jRate Executors and Dispatching Framework.

- **IExecutor**. The `IExecutor` interface provide defines the protocol between the executor abstraction and its client. This interface is implemented by concrete executors, so to make it possible to program by type rather than by class, *i.e.*, avoid tying client code to a specific executor implementation rather than an abstract protocol.

- **SingleThreadedExecutor**. This class provides a concrete implementation of the `IExecutor` that relies on a single thread of execution. This means that the executor is able to execute at most one task at the time.

- **PooledExecutor**. This class provide an implementation of the `IExecutor` interface, based on a pool of `SingleThreadedExecutor`. This class has associated a queue of task to perform, as well as a queue of executor that are meant to execute the requested tasks. Since both the task and the executor queues are template parameters, the user of this class can choose the most appropriate queuing policy based on the application logic. By default, executors are managed by using a LIFO policy (this improves cache performances), while tasks are managed by a FIFO policy.

- **IChannel**. This interface is implemented by concrete channels, so to make it possible to program by type rather than by class, *i.e.*, avoid tying client code to a specific channel implementation rather than an abstract protocol.

- **Channel**. This class implements a channel abstraction. It is parametrized by the type of item it stores, and by the container used to implement the channel itself.

Now that we have seen what are the core principles and abstraction behind the TSD framework, we can see some concrete example of how it is used within jRate in order to implement RTSJ services. Figure 4.6 and Figure 4.7 show, respectively, the class and sequence diagram that illustrates how the STD framework is used to implement the timer engine. As depicted in Figure 4.6, the timer engine is constituted by:

1. a `TimerManager` which receives requests regarding timers scheduling, and takes care of interfacing with OS provided timers to program appropriate intervals, and

2. a `PooledExecutor` which takes care of dispatching timeout notification to registered handlers, and

3. two different `Channel` instantiation which are used to keep timeout handlers and executors.

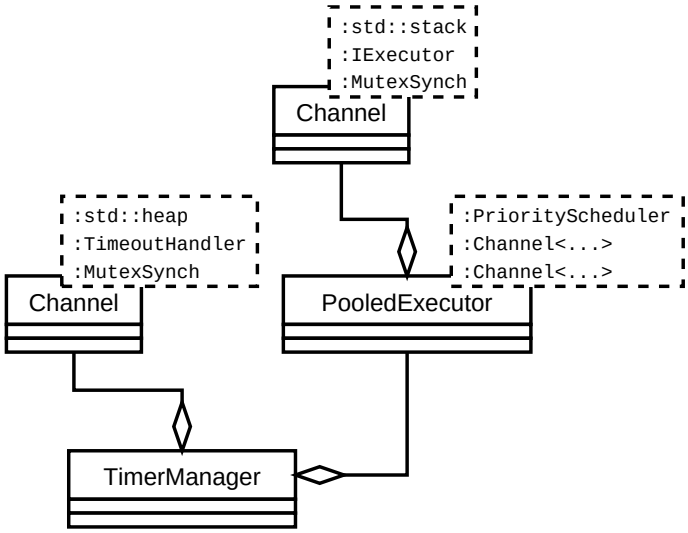


Figure 4.6: jRate Timer Implementation.

Figure 4.7 shows the sequence of message that are exchanged when scheduling and dispatching a timer. Another example of use of the TSD framework within jRate is to support the implementation of RTSJ asynchronous event. Figure 4.8 shows how jRate's core classes (shown in the diagram in darker shade) are used as native peers for implementing the `BoundAsyncEventHandler` and the `PooledAsyncEventHandler`. The latter is a jRate non standard extension, but as it will be apparent from the performance results shown in Section 6.4.3, the expert group should consider adding this as a standard RTSJ feature.

4.4 Techniques for Efficient RTSJ Memory Implementation

The RTSJ memory subsystem is one of the most challenging feature to architect and implement so to guarantee the right degree of performances, predictability and extensibility. This Section will provide an overview of the jRate generative memory area framework.

4.4.1 jRate Generative Memory Areas Framework

As described in Section 4.2.3, jRate takes a generative approach to the development of software. Specifically, it provides a framework that can be used to generate several different memory models. The building block at the base of this framework are depicted in the UML

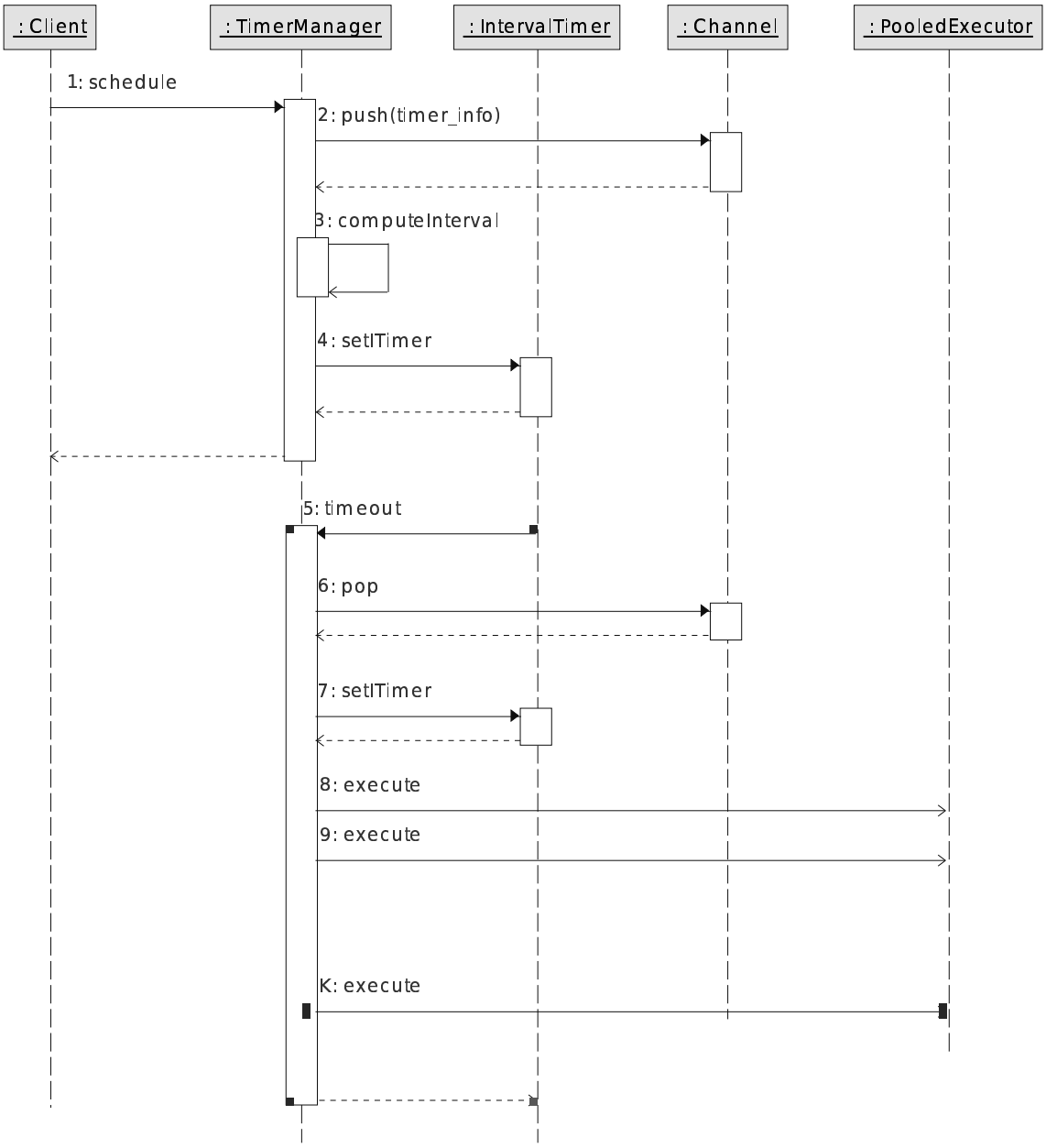


Figure 4.7: jRate Timeout Dispatching.

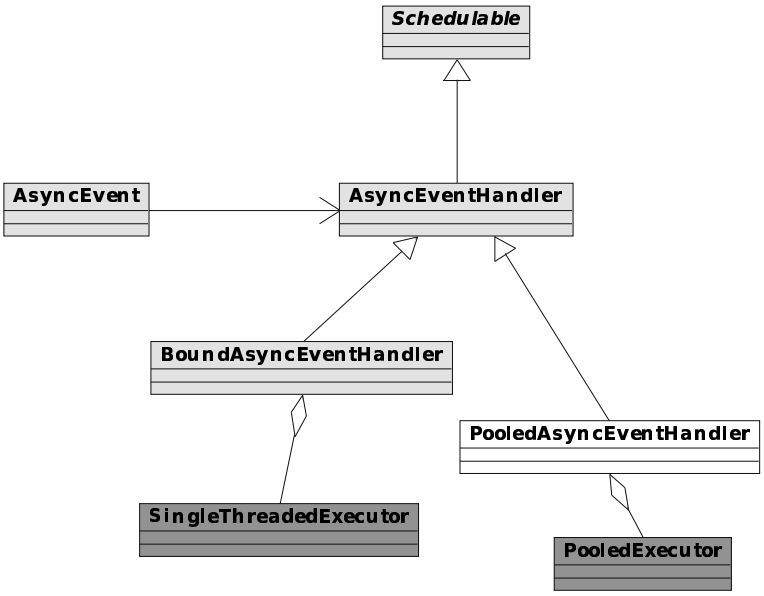


Figure 4.8: jRate Asynchronous Event Handling.

diagram shown in Figure 4.9. In this diagram (see Figure 4.9) the two template classes, `IMemoryArea` and `MemoryArea` represent the *host classes* [2] whose behaviour can be fully customized using a set of *policy classes* [2] passed as template arguments.

Conceptually the classes depicted in Figure 4.9 provide key abstractions, such as allocator, reference checker etc. The memory area framework is based on templates and implicit class protocols, thus the relationship between different abstractions is not made explicit by inheritance or association relationship. Below we provide a detailed description of the classes that compose this framework.

- **IMemoryArea.** The class `IMemoryArea` provides the basic services for needed by a RTSJ-like memory area. Thus it supports the concept of parent, portal etc. Its template parameters make it possible to parametrize the algorithm used to perform reference checking, and the type traits of memory being allocated e.g. `java.lang.Object`, `char` etc.

- **MemoryArea.** This class provides basic abstractions needed by a generic memory area, and not necessarily tied to the RTSJ memory model. In the context of jRate its `BaseMemory` is represented by the `IMemoryArea` class. This way, this class is enriched with a set of specific RTSJ services. The template parameters that can be used to configure this class are (1) the `Allocator`, which provides a means for allocating raw memory, (2) the `BufferProvider` which provides the memory managed

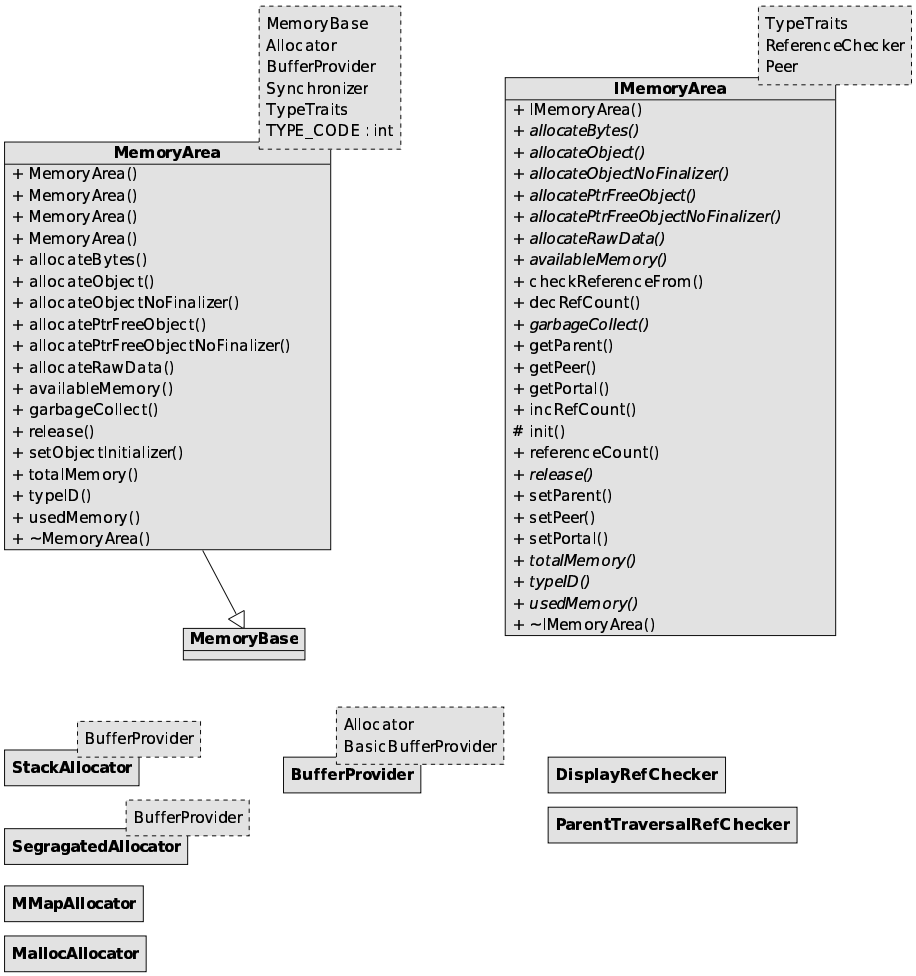


Figure 4.9: The jRate Generative Memory Area Framework.

by the `Allocator`, (3) the `Synchronizer` which provides the synchronization mechanism to be used by the `MemoryArea`, and (4) the `TypeTraits`, which describe the characteristics of the type being allocated.

- **Allocators.** As it can be seen from Figure 4.9, there is no common superclass for the various allocators, instead an implicit protocol is used to provide this service. The allocators provided with jRate are the `StackAllocator`, `SegregatedAllocator`, `MallocAllocator`, and the `MMapAllocator`. The behaviour of these classes is parametrized by means of a set of template parameters which allow to specify things like the memory alignment, the header and so on. However, to simplify the exposition, in Figure 4.9 it is shown as template parameter only the `BufferProvider`.

- **BufferProvider.** The `BufferProvider` is a *functor*, *e.g.* a function object, that takes care of creating, or retrieving, the bootstrap allocators. It's template arguments `Allocator` and `BasicBufferProvider`, represent respectively the allocator that will be created, and the means that will be used in order to get the memory for the allocator to manage. It is worth noticing that often, applications have only one bootstrap allocator.

Based on the description given so far of the jRate generative memory area framework, the reader might have found that there is a circular dependency in the classes depicted in Figure 4.9. The circular dependency arise from the fact that allocators depend on buffer provider and vice-versa. This recursive dependency is the real power of the generative memory area framework. In fact it allows the composition of the different classes depicted in Figure 4.9 so to create arbitrary complex memory systems. As in every recursive structure the circularity is broken by using base cases. As an example, let's consider the concrete memory system depicted in Figure 4.10. In this scenario we have a bootstrap allocator that gets its working chunk of memory using a `MMapAllocator`, and then manages this memory by using a `SegregatedAllocator`. The `SegregatedAllocator` uses the classic segregated list approach to memory management in order to allow efficient allocation and release of memory chunks. Then we have a `MemoryArea` that relies on a `StackAllocator` whose buffer provider is the `SegregatedAllocator`. The reader might wonder at this point why all this machinery is needed. The great advantage of being able to create *layers* of allocator is that of using different allocation strategies at different level so to maximize the performance and predictability of the overall memory system. For

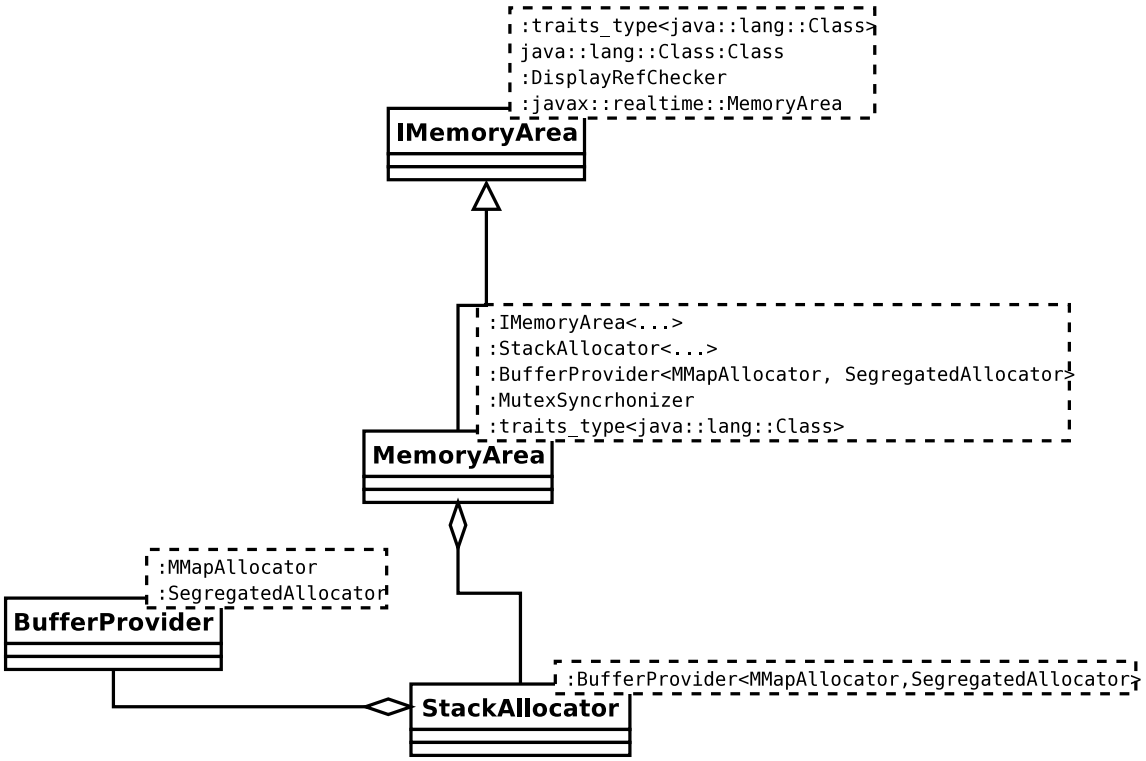


Figure 4.10: An sample instantiation of the jRate Generative Memory Area Framework.

instance, one of the commonly used configuration in jRate uses a singleton bootstrap allocator based on a `SegregatedAllocator`, which provides the big chunk of memory needed by the various memory areas. This allows for a very efficient and predictable creation of `MemoryAreas`. Figure 4.11 represent a typical application scenario, where the `BufferProvider` provides access to a singleton bootstrap allocator, the `MemoryArea` relies on the bootstrap allocator in order to obtain memory for its own allocator, and defines its specific allocator for the given memory chunk.

4.5 Optimizing the RTSJ Run-Time Checks

In Chapter 2 we described the checks required by a compliant RTSJ implementation in order to (1) detect the creation of illegal references, and thus the potential for dangling pointers, and (2) enforcing the single parent rule. We showed that the complexity of those runtime checks was $O(n)$, with n being the depth of the scope stack. A possible optimization of the Algorithm 3 is reported in Algorithm 5. This algorithm, based on the observation that reference from scoped memory to heap or immortal memory are always

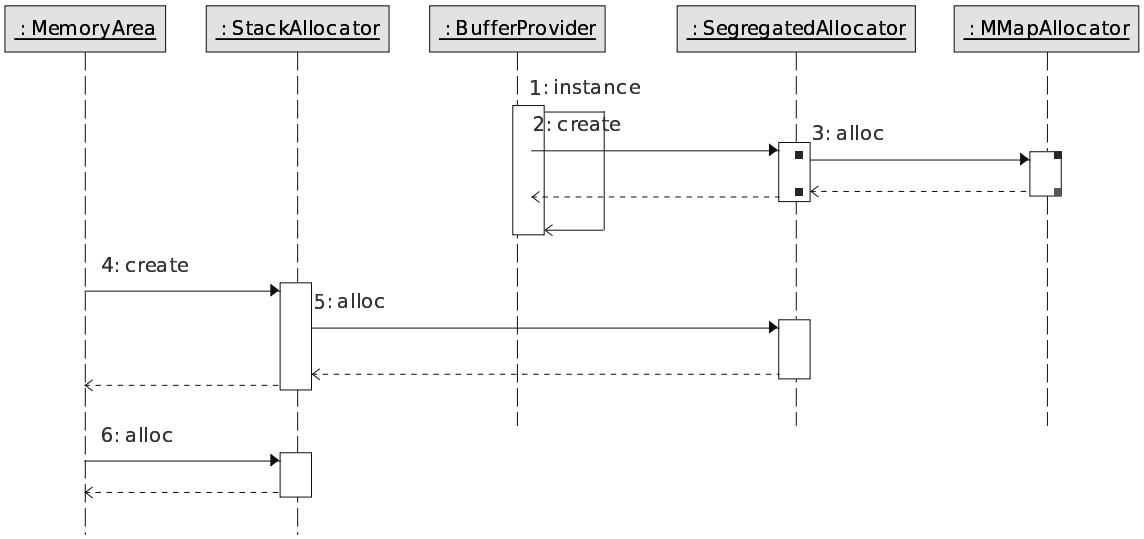


Figure 4.11: jRate Generative Memory Area Framework at Work.

legal, and the opposite is always illegal, avoids scanning the scope stack, and simply follows a scope's parent link to discover if the target reference is in an ancestor scope of the source. However, the complexity in the worst case is always $O(n)$, and thus linear. While at first sight this might not seem a serious problem, it has a rather subtle implication. The problem is that by having a linear time check for single parent rule enforcement and reference safety checking makes simple operations such as the enter of a memory area or the assignment of a reference to an object field dependent on the structure of the scope stack. Taking this argument to an extreme, the sample code reported in Listing 4.12 has some surprising properties. For instance, the time taken by an operation as simple as a field setting (for reference-type fields) shown at line 7 in Listing 4.12 takes an amount of time that depends linearly on the length of the scope stack. This means that the time taken to execute an instruction as simple as a field set is not constant, and it depends on the structure of the scope stack which might depends on some application specific runtime properties. Entering a memory area has a similar problem, in fact, the time taken to execute the instruction reported at line 5 in Listing 4.12 depends linearly on the length of the scope stack. It is well known that for real-time applications, it is important to be able to put bounds on the execution of operations within some piece of code, and for code within a whole application. Real-time applications inevitably contain code fragments whose execution time must be statically known. The scope stacks for an application—in particular, their depth—are not necessarily decidable at compile-time. Thus, linear-time algorithms for checking the single-parent rule and the memory references may incur unpredictable

Algorithm 5: checkReferenceValidity2

```
Input: MemoryArea from, MemoryArea to
Output: boolean validReference
begin
    validReference ←true;
    if from ≠to then
        if to instanceOf ScopedMemory then
            if from instanceOf ScopedMemory then
                ancestor ← from.getParent();
                while to ≠ ancestor and ancestor ≠ nil do
                    ancestor ← ancestor.getParent();
                if to ≠ ancestor then
                    validReference ←false;
            else
                validReference ←false;
end
```

```
...
Runnable logic = new Runnable {...};
LTMemory mem = new LTMemory(minSize, maxSize);
mem.enter(logic);
...
XType x = ...;
obj.setFieldX(x); // obj.x = x;
...
```

Figure 4.12: Code fragments that illustrate the RTSJ reference issue.

overhead. As a result, the application must either overprovision, thus wasting resources, or else underprovision, and perhaps miss a critical deadline.

Summarizing, the time spent checking a given memory reference using the above algorithms will vary depending on the particular scope stack in place when the check is performed. Even a simple store instruction will thus take varying amounts of time depending on the scope stack. This prevents analysis of the code's timing in any particular context, but makes it necessary to analyze its timing in *all* the different contexts in which it could be executed. In our experience in developing an Object Request Broker (ORB) for RTSJ, such unpredictability makes it impossible to bound reasonably the time for servicing clients' requests.

Before we tackled this problem in [14], solution presents in literature [8, 5, 35] where based on linear time algorithm, and thus not ideal for real-time applications. We next show how to extend the data structures used by the RTSJ to perform all required checks in constant time. Our approach is inspired and based upon type-inclusion tests for single-inheritance, object-oriented languages [37].

We have implemented the algorithms described here in the jRate [19] memory subsystem; for convenience, we describe our approach in the context of jRate.

4.5.1 Optimizing the Singe Parent Rule Check

jRate's scope-stack implementation uses data structures that allow all scope stack operations to be performed in constant time. As shown in Figure 4.13, jRate augments the scope-stack data structure suggested by RTSJ, linking those slots that represent scoped memory areas. An index is also maintained to the topmost scoped memory, so that the next scoped memory area pushed onto the stack can be linked with the others.

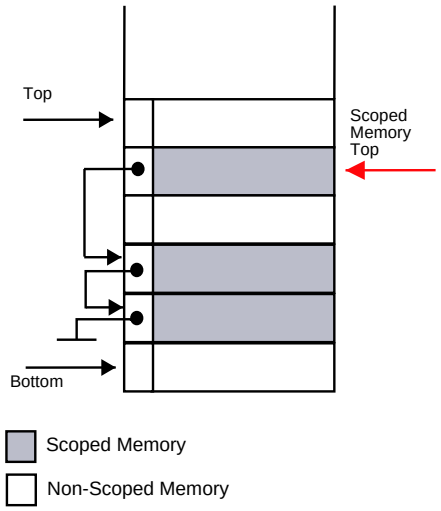


Figure 4.13: The jRate Scope Stack structure.

This design allows a constant-time implementation of $findFirstScope()$, while at the same time maintaining constant-time bounds for $push()$ and $pop()$.

Algorithm 6 provides the pseudocode for the constant-time implementation of $findFirstScope()$, while Algorithm 7 and Algorithm 8 provide the pseudocode for the $push$ and pop operations. On inspection it can be seen that all of these operation are performed in constant time. As compared with the RTSJ-inspired implementations, no scanning of the scope stack is required.

Algorithm 6: findFirstScope

```
Input: ScopeStack ss
Output: ScopedMemory firstScope
begin
    firstScope ← primordialScope;
    if ss.lastSMIndex ≠ STACK_END then
        firstScope ← ss [ss.lastSMIndex ].ma ;
end
```

Algorithm 7: Push

```
Input: MemoryArea ma, ScopeStack ss
begin
    if ma instanceOf ScopedMemory then
        if ma.getParent() = nil then
            ma.setParent(findFirstScope(ss ));
        else
            if ma.getParent() ≠ findFirstScope(ss ) then
                throw ScopedCycleException;
        ss.top ← ss.top +1;
        ss [ss.top ].ma ← ma ;
        ss [ss.top ].prevSM ← ss.lastSMIndex ;
        ss.lastSMIndex ← ss.top ;
    else
        ss.top ← ss.top +1;
        ss [ss.top ].ma ← ma ;
end
```

The actions required for processing memory areas require knowing what kind of memory area is at-hand (immortal, scoped, etc). While such information could be determined by Java^{TM}'s `instanceof` test, that test may require time linear in the depth of the program's class hierarchy. Instead, jRate] optimizes such tests by tagging memory area objects to allow a constant-time test.

This enhancement of the scope stack structure makes it possible to know exactly which entries of the scope stack are scoped memories and which are not. This knowledge enables it to elide a test on the type of memory area, and avoids blindly searching for scoped memories on the stack to decrement the reference count when destroying the scope stack. As a final note, the size of a jRate real-time thread's scope stack can be fixed at

Algorithm 8: Pop

```
Input: ScopeStack ss
begin
    if ss [ss.top ].ma instanceOf ScopedMemory then
        ss.lastSMIndex ← ScopedMemory[ss.top ].prevSM ;
    ss.top ← ss.top −1;
end
```

thread-creation time. This design makes jRate more space efficient by avoiding the use of pointers to implement the linked list.

4.5.2 Optimizing the Memory Area Reference Check

To understand the main idea behind our technique for implementing the reference checks in constant time, consider a generic scope stack tree, such as the one in Figure 4.14. In this example, some of the memory areas are scoped and some are not. The scope stack of any running real-time thread can be represented as a path from the root down to some interior or leaf note. As stated in Section 2.4.2, references can always be made from objects in a scoped memory to object in the heap or immortal memory; the opposite is never allowed. Also, the ancestor relation among scope memory areas is defined by the nesting of the areas themselves; intervening entry into the heap or immortal memory areas do not affect the scopes. We call the tree implied by the scoped memory areas' ancestor relation the *parenthood tree*. For instance, collapsing all the nodes that represent either the heap or immortal memory in the scope tree of Figure 4.14, we obtain the tree depicted in Figure 4.15. In this tree, a direct edge from node B to node A, means that A is the parent of B.

The advantage of this formulation is that *subtype-testing* algorithms [37] can be applied to the parenthood tree to determine legal references. In particular, if we think of each node of the tree as representing a type, as well as a memory area, then we can rephrase the memory reference checking into a sub-type testing, for single type inheritance. The proof of this fact is straightforward, and it follows easily from the definition of subtype, memory reference validity, and single parent rule. More specifically, given a *parenthood tree* T, we have that:

$$\forall x, y \in T : (y \preceq x) \rightarrow (y \rightsquigarrow x)$$

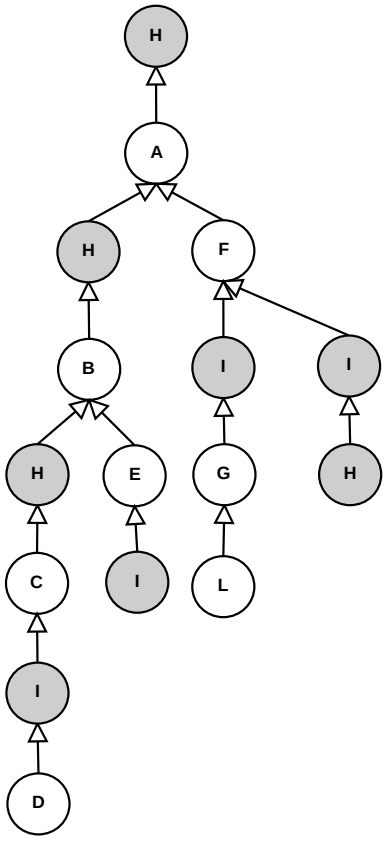


Figure 4.14: A sample Scope Tree structure (Grey nodes represent the Heap(H), and Immortal (I) Memory).

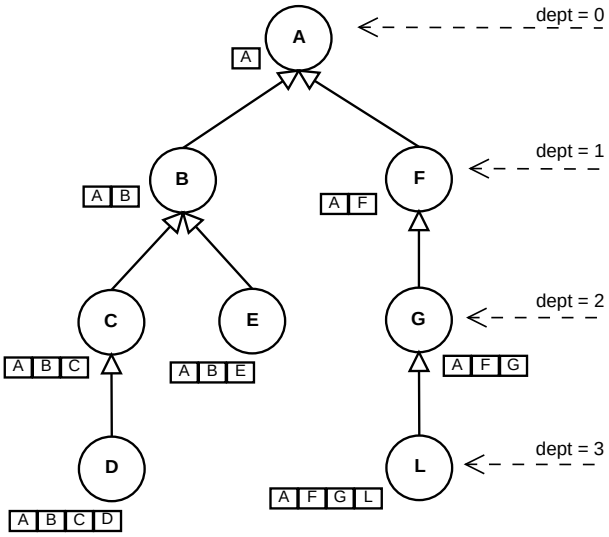


Figure 4.15: Transformed and Decorated Scope Tree structure.

Where the symbol $\preceq$ has the usual meaning of subtype, while we use the symbol $\rightsquigarrow$ to indicate that a memory area on the left side of the $\rightsquigarrow$ can contain objects that point to the memory area on the right side of the $\rightsquigarrow$. Relying on this observation, we can use a technique based on displays, similar to that proposed by Cohen [37], to determine the validity of a memory reference in constant time. The *parenthood tree* can change with time, since the scopes' parent-child relation changes as the application runs and scoped memory areas are entered and exited. Thus, the associated type hierarchy is not fixed, but changes at well-defined points—when the reference count of a scoped memory goes from zero to one or from one to zero. At such points, the typing information associated with a scoped memory has to be created and destroyed, respectively.

To facilitate a constant-time memory-reference check, we augment the information associated with a scoped memory to include its depth in the *parenthood tree* and a display that contains the type identification codes of its ancestors in the parenthood tree. The address of a scoped area serves nicely as a unique type identifier, thanks to the single-parent rule: once a scoped memory area is reclaimed, no memory area can store its address in its display. This avoids the need to map a storage area to a unique number and improves the efficiency of our algorithms.

For example, if we consider the scoped memory C in Figure 4.15, its depth will be 2 and its display will be (A, B, C). Notice that the depth is the same as the display length minus one, so an implementation won't necessarily need to store this information twice. Here we treat them separately only to simplify exposition of our algorithms. Algorithm 9 contains the pseudocode that shows how, with these extensions, it is possible to perform the memory reference check in constant time. In this algorithm it is assumed that both the heap and the immortal memory have a depth of -1.

Finally, we note that the management of displays does not add considerable complexity when entering or exiting a memory area. When a scoped memory is entered for the first time, or after its reference count has dropped to zero, setting up the display simply requires copying the parent's display and adding itself at the end. The only operation required when the last thread leaves the scoped memory is to invalidate the display.

In summary we have described a constant-time algorithm that offers far greater predictability and asymptotic efficiency over the approach currently implemented for RTSJ.

Algorithm 9: checkReferenceValidity

```
Input: MemoryArea from, MemoryArea to
Output: boolean validReference
begin
    validReference ←false;
    if from.depth ≥ to.depth then
        if to.depth = −1 then
            validReference ←true;
        else
            if from.display [to.depth ] = to then
                validReference ←true;
end
```

4.5.3 Selecting the Most Appropriate Algorithm

As we have seen in the previous sections, there are several algorithms that could be chosen for implementing the RTSJ's safety checks. Some new constant time algorithms have been proposed at the time of the writing of this thesis [48]. The algorithm proposed in [48] is based on the a known techniques that encodes types with integer intervals. The main difference between the technique proposed in [48] and the one proposed in this thesis are (1) the need for recomputing of intervals at runtime, and (2) the use of constant space for type encoding. As it should be evident to the reader this solution trades space for time.

Table 4.1: Criteria for Reference Checking Algorithm Selection.

	Real-Time	**Embedded**	**Embedded and Real-Time**
Deep Scope Stacks	O(1) Display)	O(N) Parent Traversal	O(1) Display or Interval
Shallow Scope Stacks	O(1) Display)	O(N) Parent Traversal	O(1) Display

Table 4.1 contains some suggestions on the algorithms to use for different combinations of scope stack dept and system. For instance, for real-time systems on which memory footprint is not an issue the algorithm to choose is the display based algorithm presented in this thesis. On the other hand, for real-time systems that have memory constraints, and a deep scope stack, one might chose either the display or the interval based algorithm.

4.5.4 Empirical Validation

We next present the results of a performance comparison between the memory-reference checking scheme proposed in this paper and the one proposed by the RTSJ. The scope of

this performance measurement is to empirically validate the effectiveness of the display solution approach. For a description of the testbed on which the measurement were made, please, refer to Section 6.2.

4.5.5 Test Description and Results

To compare the performance of our approach against that of other systems based on the RTSJ, we implemented both algorithms in jRate. Other RTSJ systems to-date follow Algorithm 3—scanning the scope stack to determine the validity of a reference check. But, if the $instanceof$ tests can be carried out in constant time[3], then Algorithm 5 is more efficient since there is no need to scan the scope stack. Instead, the scope hierarchy is consulted. We therefore implemented and compared the better Algorithm 5 with our display-based approach as described by Algorithm 9.

Both algorithms are implemented in C++, on the native side of jRate. To better estimate the performance of the two algorithms, and to avoid the overhead and interference of Cygnus Native Interface (CNI), we instrumented the native code directly to measure times for both approaches.

In order to compare the efficiency of the different memory reference algorithms, we extended the RTJPerf[4] [17] benchmarking suite, with a test that creates a reference from a scoped memory B to a scoped memory A; where A is the nth ancestor of B, *i.e.*, an ancestor that has a distance of n from B. The values of n considered were 0, 1, 2, 4, 8, 16, 32, and 64. The time for performing the reference check was computed as the average of 2000 samples.

Figure 4.16 shows the average, 99%, maximum, and standard deviation for the two algorithms.

As can be seen from Figure 4.16, the display-based memory-reference check outperforms the parent-traversal-based check in each of the test cases. Moreover, as claimed in Section 4.5.2, the experiments confirm that the checks execute in constant time, regardless of the depth of the scope stack. The display based algorithm, not only provides average constant time checks, but its worst case performance indexes, *i.e.*, 99% and max, are very closely bound—practically identical—to the average.

Note that the results provided by this test also predict the timing behavior of these algorithms for *invalid* memory references. For the display-based approach, the time taken

[3] jRate uses a custom encoding for those classes that require frequent `instanceof` tests. This allows `instanceof` to be replaced by an integer comparison or a bitwise-and operation.

[4] RTJPerf is currently the only available RTSJ benchmarking suite

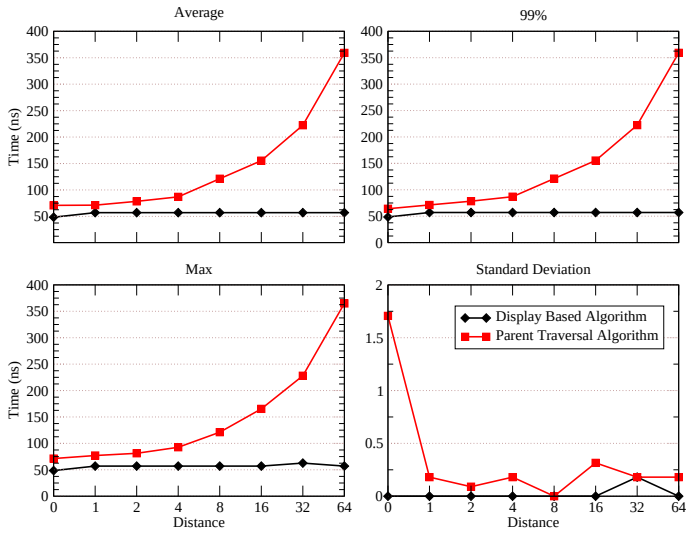


Figure 4.16: Performances comparison between Display based vs. Parent traversal algorithm.

to detect a valid or an invalid reference is the same, and does not depend on the structure of the parenthood tree. On the other hand, if we consider the parent-traversal algorithm, the time necessary for detecting an invalid reference (in the case of two scoped memories) requires traversing the parents path to the root. Thus, results shown in Figure 4.16 determine the time taken to check an invalid reference when the $from$ scoped memory has a distance of n from the root of the *parenthood tree*.

Chapter 5

Extending the RTSJ

5.1 Introduction

In Chapter 4 we've shown the current limitation of the RTSJ and have provided solutions to them that did not have an impact on the API, nor on the programming model. This Chapter motivates a set of extensions to the RTSJ which are meant to make it either more performant or easier to program. The performances of most of these extensions will be evaluated empirically in Chapter 6 .

5.2 Constant Time Scoped Memory

The scoped memory is one of the most fundamental addition introduced by the RTSJ. This addition has both an impact on the programming model and on the way to architect applications. The key goal RTSJ's architects had in mind when introducing this abstraction was that of allowing the automatic reclamation of objects, without requiring the use of a tracing garbage collector. The RTSJ defines only two allocation strategy for scoped—linear time and variable time. The case for constant time allocation was not considered[1].

Intent. Provide and efficient and predictable constant time allocation scoped memory, so to make it easier to perform timing analysis.

Example. For real-time applications it is very important to be able to perform worst case execution timer (WCET) analysis. Thus, each time there is a code fragment that takes a non

[1]To be more precise, it had been considered in earlier version of the specification and then dropped. However, based on the findings of this thesis, we believe that a constant time scoped memory is really needed.

constant amount of time, in order to determine its execution time, worst case bounds should be considered for appropriate inputs. This clearly makes the analysis harder to perform and less accurate. Because of this it is very important for a middleware that has to support real-time applications to easy WCET analysis.

Problem. Linear or variable time allocation make it harder to perform WCET analysis, and can also introduce unpredictability.

Solution. Provide `CTMemory`, a highly efficient and predictable constant time scoped memory.

Structure. Figure 5.1 indicates the structure of the proposed solution. The `CTMemory` extends the RTSJ `ScopedMemory` class and delegates its implementation to a native peer. The native peer is just an instance of a jRate memory area framework template (properly instantiated). Notice that the only thing that is special about the template instantiation is that it is configured with a constant time allocator—the `StackAllocator`.

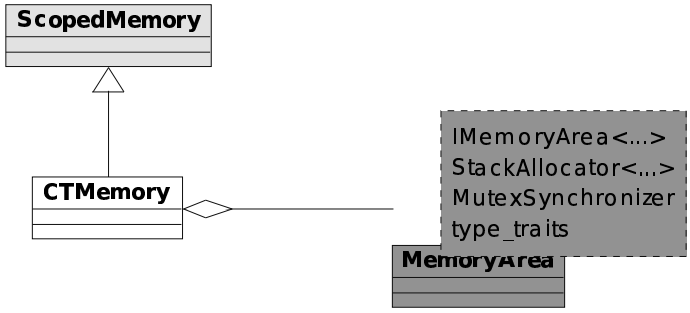


Figure 5.1: jRate's CTMemory.

5.3 Private Scoped Memory (Scratch-pad)

Scoped Memory give a way of controlling the allocation and deallocation of a group of objects—those allocated within the memory area. The issue that has to be considered when working with memory areas is that the RTSJ specification is underspecified respect to the following point (1) when the object allocated within a scoped memory are finalized, and (2) who finalizes those objects. A typical choice taken by implementors is that of having the last thread leaving the scoped memory finalizing the objects contained within it. Other implementation use a separate thread to cope with reclamation.

The private scopes idiom, also called the scratch-pad, is useful in all those situation in which an RTSJ real-time thread has to perform some operation which requires allocation of many temporary objects, but does not have the need to share any objects with another thread i.e. the temporary objects can be considered private.

Intent. Provide a way to ensure that a scoped memory is used at any point in time by a single real-time thread.

Example. Consider a simple Web Server application. When a server thread serves a HTTP request it might generates quite a bit of temporary data, and this data is usually not relevant to other thread concurrently serving other requests (in this example we are ignoring caching).

Problem. There is an application thread that cannot tolerate the unpredictability introduced by the finalization of traditional scoped memory. At the same time this thread does not need to share any data with any other thread.

Solution. Extend the scoped memory (for instance the `CTMemory`) so to enforce that only one thread at the time can enter it. Take advantage of this by removing all the locking needed for thread safety.

Structure. The resulting structure of the solution is the one reported in Figure 5.2. Here a `CTPrivateMemory` extends the RTSJ `ScopedMemory` class and delegates its implementation to a native peer. The native peer is just an instance of a jRate memory area framework template (properly instantiated). Notice that the only thing that is special about the template instantiation is that it is configured with a constant time allocator—the `StackAllocator`, and it does not use any locking to make its operation thread safe.

5.4 Memory Tunnels

As we have seen so far, the RTSJ does not provide any specific mechanism to transfer object graphs from one memory area to another. The only mechanism that could be used is the Java serialization, but this has some limitation since (1) it requires that all the objects to be serialized implement the `Serializable` interface, and (2) is not the most efficient mean of getting an object graph from one memory region to another.

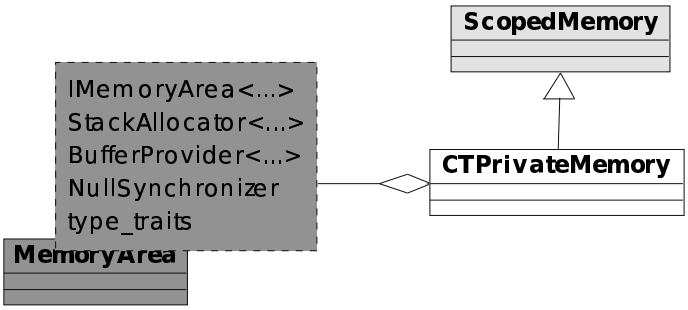


Figure 5.2: jRate's CTPrivateMemory

Intent. Provide an efficient mechanism for threads to transfer object graphs across memory areas.

Example. The RTSJ makes it quite complicated to implement producer consumer paradigms since they conflict with the programming model imposed by scoped memories. This issue has been faced by people who have tried to implement real-time Java ORB that rely on the RTSJ. The issue that often arise is that an object, or an object graph, from one memory area needs to get into another, but there is no neat way of accomplishing the goal.

Problem. Solve the problem introduced by the scoped memories which limits the possibility of data sharing between threads, as well as the implementation of producer consumer applications.

Solution. The idea is that of "breaking the barriers" of scoped memories. To this aim, a structure that can be used to perform object passing among different threads, forcing a violation of the memory constraint of RTSJ but without provoking consequences, is proposed. Like in some diodes the "tunnel effect" allows electrons to break a barrier of potential, *memory tunnels* allow a message passing by-value among threads that live in different and not compatible memory areas. From an abstract point of view, a memory tunnel is a traditional queue exposing two operations—*put* and *get*—that can be used to write an object, at one side, and to read it from another side (Figure 5.3). The underlying mechanism, like any other technique used to access data from different domains[2], is based on performing a *copy* of the object: when `put` is executed, the object is copied from the source memory onto an *internal temporary area* that holds the queue in the tunnel; similarly, the effect of `get`

[2]think, for example, to accessing user space by kernel code, operation usually done by copying data from user to kernel space [45].

is doing a copy the object from the temporary area into the memory of the destination domain. Since the real operation is *cloning* the object to be passed and the latter could contain references to other objects of the source (or a compatible) domain, the implementation of memory tunnels provide two versions of `put`: a `deep_put`—which performs a *deep-copy* of the object—and a `shallow_put`—which instead does not copy the referred objects. In the latter case, when a `get` is performed, a suitable check verifies that references present in the copied object are still valid in the new domain.

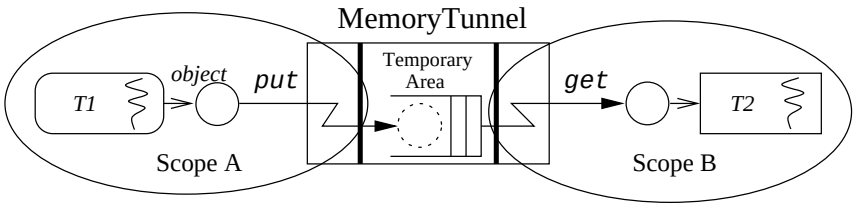


Figure 5.3: A Memory Tunnel

Using this structure, data can be transferred without needing to share anything among the threads involved; for this reason, memory tunnels can be employed without any problem when traditional forms of data passing-by-reference fail due to RTSJ memory model constraints. However there is an implementation requirement that, if not met, does not allow the use of memory tunnels: the internal temporary area must be allocated in a memory space and using management mechanisms that are outside the RTSJ memory model, otherwise we easily fall again in the non-feasibility conditions caused by RTSJ constraints. A memory tunnel is thus identified univocally by means of a *literal name* (a Java^{TM} string), which is given to the `TunnelProxy` object to perform access to that tunnel.

Structure. Figure 5.4 and Figure 5.5 show the structure of the solution in the context of a dispatcher/worker problem. In this solution, there is a different memory tunnel for each worker that is referred by two different `TunnelProxy` objects: one for dispatcher side and the other one for worker side. The behaviour of the dispatcher is quite simple: once received the message from the network, it gets the `TunnelProxy` (dispatcher side) object, relevant to the worker, and puts the message in it. On the other hand, from the worker's point of view, the task of handling messages is split between two different objects: the `Worker` and the `Executor`. The former implements a loop that continuously passes the control to the latter, while the latter has the real responsibility of processing messages. The presence of two different objects is due to the necessity of using scoped memory to avoid

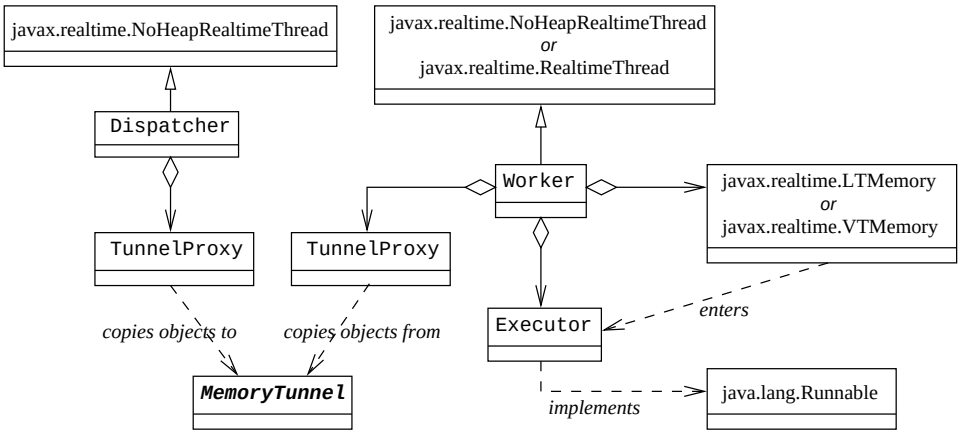


Figure 5.4: Dispatcher/Worker pattern with Memory Tunnels

memory leaks and/or explicit memory management. In fact, the memory used by each object message got from the tunnel, after the latter has been used, should be reclaimed: the only way to do this, with RTSJ, is (1) entering a scoped memory, (2) getting the message, (3) processing it, and (4) exiting the scope. To this aim, the processing task has to be encapsulated in a `Runnable` object that is iteratively entered/exited in/from the scoped memory. As Figure 5.5 shows, this iterative operation is performed by the `Worker` object, while the `Executor` object encapsulates the real processing task. Since the message is got by the `Executor` (which runs in a scope), the memory allocated is automatically released when the task is finished (as the latter exits the scope).

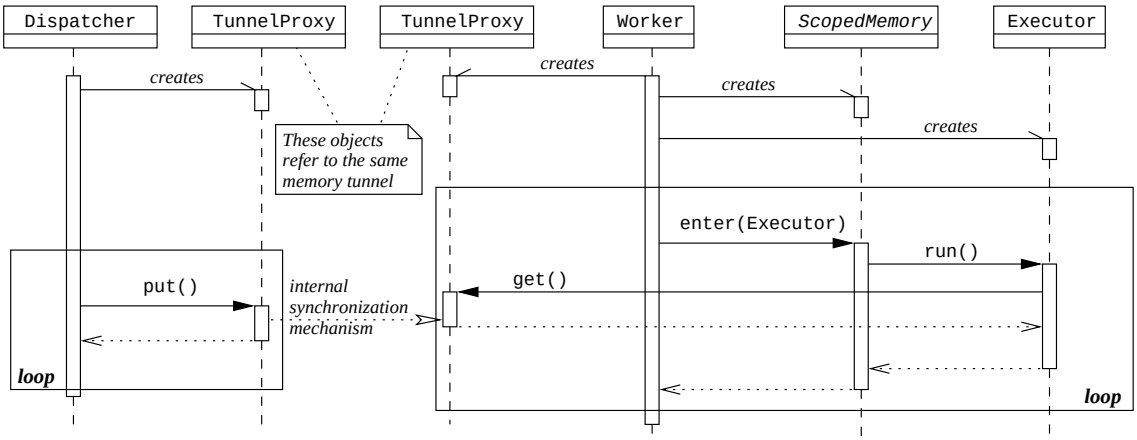


Figure 5.5: Sequence Diagram of Dispatcher/Worker with Memory Tunnels

Implementation. The prototype of the class `TunnelProxy` is given in Figure 5.6. As it can be noted, the class has two constructors: one to open an existing memory tunnel and

another to create a new memory tunnel, given the maximum number of items and the size of the temporary memory area to be allocated for the queue. Methods provided allow to put an object by using a deep or shallow copy, and to retrieve a stored object, checking also the validity of embedded references. The proxy provides also a `close()` method to signal that the opening/creating thread is no longer interested in using the tunnel. As Figure 5.6 shows, class `TunnelProxy` does not have a method to (explicitly) destroy a memory tunnel; indeed, the lifetime of a memory tunnel is based on a reference counting mechanism: a reference count that is incremented each time the tunnel is created or opened (i.e. a `TunnelProxy` is created for that tunnel), and is decremented each time a `close()` method is called; when the reference count is zero the tunnel is destroyed and memory allocated is released.

```
1  public class TunnelProxy {
2
3    /* Creates a memory tunnel.
4       "size" gives the dimension of the temporary area  and "items" gives
5       the maximum number of objects that can be put in the queue. */
6    public TunnelProxy(java.lang.String name,int itemSize,int items);
7
8    /* Opens a memory tunnel */
9    public TunnelProxy(java.lang.String name);
10
11   /* Writes the object in the tunnel using a shallow copy */
12   public synchronized void shallowPut(java.lang.Object object)
13                        throws java.lang.InterruptedException;
14
15   /* Writes the object in the tunnel using a deep copy */
16   public synchronized void deepPut(java.lang.Object object)
17                        throws java.lang.InterruptedException;
18
19   /* Retrieves the object from the tunnel.
20      It also checks whether the references contained are valid in the new domain */
21   public synchronized java.lang.Object get()
22                        throws javax.realtime.IllegalAccessException,
23                               java.lang.InterruptedException;
24
25   public synchronized void close ()
26                        throws java.lang.InterruptedException;
27 }
```

Figure 5.6: The TunnelProxy Class

Chapter 6

Performance Evaluation

6.1 Introduction

In the previous Chapters were shown several techniques that can be used to improve performances and predictability of Real-Time Java platforms. In this Chapter, we provide empirical evidence that the techniques discussed previously are effective in improving performances and predictability. This will be done by comparing jRate's [10] performance with those of two commercially available RTSJ implementations, JTime [47] and Jamaica [1]. In the remainder of this Chapter we will introduce RTJPerf [17, 12], the benchmarking suite used to evaluate these RTSJ implementations, and we will provide a detailed analysis of the empirical evaluation results.

6.2 Overview of RTJPerf

Two quality dimensions should be considered when assessing the effectiveness of the RTSJ as a technology for developing real-time embedded systems:

- **Quality of the RTSJ API**, *i.e.*, how consistent, intuitive, and easy is it to write RTSJ programs. If significant *accidental complexity* is introduced by the RTSJ, it may provides little benefit compared to using C/C++. This quality dimension is clearly independent from any particular RTSJ implementation.

- **Quality of the RTSJ implementations**, *i.e.*, how well do RTSJ implementations perform on critical real-time embedded system metrics, such as event dispatch latency, context switch latency, and memory allocator performance. If the overhead incurred

by RTSJ implementations are beyond a certain threshold, it may not matter how easy or intuitive it is to program real-time embedded software since it will not be usable in practice.

This Chapter focuses on the latter quality dimension and systematically measures various performance criteria that are critical to real-time embedded applications. To codify these measurements, we use an open-source[1] benchmarking suite called RTJPerf that the author has developed during his PhD.

6.2.1 Capabilities of the RTJPerf Benchmarks

RTJPerf provide benchmarks for most of the RTSJ features that are critical to real-time embedded systems. Below, we describe these benchmark tests and reference where we present the results of the tests in subsequent sections.

Threads

In Chapter 2 we saw how the RTSJ extends the Java threading model with two new types of real-time threads: `RealtimeThread` and `NoHeapRealtimeThread`. RTJPerf provides the following benchmarks that measure important performance parameters associated with threading for real-time embedded systems.

Context Switch Test. High levels of thread context switching overhead can significantly degrade application responsiveness and determinism. Minimizing this overhead is therefore an important goal of any runtime environment for real-time embedded systems. To measure context switching overhead, RTJPerf provides two tests that contains two real-time threads—configurable to be either either `RealtimeThread` or `NoHeapRealtimeThread`—which cause a context switch in one of the following ways:

1. **Yielding**—In this case, there are two real-time threads characterized by the same execution eligibility that yield to each other. Since there are just two real-time threads, whenever one thread yields, the other thread will have the highest execution eligibility, so it will be chosen to run.

2. **Synchronizing**—In this case, there are two real-time threads—T_H and T_L—where T_H has higher execution eligibility than T_L. T_L, enters a monitor M and then waits

[1] RTJPerf is freely available at `http://www.cs.wustl.edu/rtj`.

on a condition C that is set by T_H just before it is about to try to enter M. After the condition C is notified, T_L exits the monitor, which allows T_H to enter M. The test measures the time from when T_L exits M to when T_H enters. This time minus the time needed to enter/leave the monitor represents the context switch time.

The results for these tests are presented in Section 6.4.1.

Periodic Thread Test. Real-time embedded systems often have activities, such as data sampling and control law evaluation, that must be performed periodically. The RTSJ provides programmatic support for these activities via the ability to schedule the execution of real-time threads periodically. To program this RTSJ feature, an application specifies the proper release parameters and uses the `waitForNextPeriod()` method to schedule thread execution at the beginning of the next period (the period of the thread is specified at thread creation time via `PeriodicParameters`). The accuracy with which successive periodic computation are executed is important since excessive jitter is detrimental to most real-time systems.

RTJPerf provides a test that measures the precision at which the periodic execution of real-time thread logic is managed. This test measures the actual time that elapses from one execution period to the next. These test results are reported in Section 6.4.1.

Thread Creation Latency Test. The time required to create and start a thread is a metric important to some real-time embedded applications. particularly useful for dynamic real-time embedded systems, such as some telecom call processing applications, that cannot spawn all their threads statically in advance. To assess whether a real-time embedded application can afford to spawn threads dynamically, it is important to know how much time it takes. RTJPerf therefore provides two tests that measure this performance metric. The difference between the tests is that in one case the instances of real-time threads are created and started from a regular Java thread, whereas in the other case the instances are created and started from another real-time thread. The results of this test are reported in Section 6.4.1.

Memory

In Chapter 2 we saw how the RTSJ extends the Java memory model by providing memory areas other than the heap. These memory areas are characterized by the lifetime the objects created in the given memory area and/or by their allocation time. *Scoped memory areas*

provide guarantees on allocation time. Each real-time thread is associated with a *scope stack* that defines its allocation context and the *history* of the memory areas it has entered. RTJPerf provides the following test that measures key performance properties of RTSJ memory area implementations.

Allocation Time Test. Dynamic memory allocation is forbidden or strongly discouraged in many real-time embedded systems to minimize memory leaks, latency, and non-predictability. The scoped memory specified by the RTSJ is designed to provide a relatively fast and safe way to allocate memory that has nearly the flexibility of dynamic memory allocation, but the efficiency and predictability of stack allocation. The measure of the allocation time and its dependency on the size of the allocated memory is a good measure of the efficiency of various types of scoped memory implementations.

To measure the allocation time and its dependency on the size of the memory allocation request, RTJPerf provides a test that allocates fixed-sized objects repeatedly from a scoped memory region whose type is specified by a command-line argument. To control the size of the object allocated, the test allocates an array of bytes. It is possible to determine the allocation time associated with each type of scoped memory by running this test with different allocation sizes.

Scoped Memory Lifetime Test Scoped memory is one of the key features introduced by the RTSJ. It enables applications to circumvent the garbage collector, yet still use automatic memory management, by (1) associating with each memory scope a reference count that depends on the number of real-time threads within the memory area (*i.e.*, that have entered the scope but yet not exited it), and (2) ensuring that all the objects allocated in the scope are finalized, and the space reclaimed, as soon as the reference count associated with the memory area drops to zero. Since most RTSJ applications use scoped memory heavily it is essential to characterization its performance precisely.

RTJPerf provide a test that measures (1) the time needed to create a memory scope, (2) the time needed to enter it, and (3) the time needed to exit it. The time needed to create a scoped memory area depends on the following factors:

- The allocation context of the thread that creates the memory scope. The Allocation Time Test measures this aspect of memory scope creation time.

- The native C/C++ implementation of scoped memory. The Scoped Memory Lifetime Test measures the efficiency and predictability of the native C/C++ implementation of scoped memory.[2]

The time needed to exit a memory scope is measured by the case in which its reference count drops to zero as a result of the thread exiting the scope. In this case, the memory scope must finalize all the objects allocated within it and reclaim the used storage.

To determine the time needed to enter, exit, and create a memory scope – and to determine how efficient the implementation is – this test creates a memory scope, enters it, fills it with objects, and then exits the scope. The test can be run by configuring the type of scoped memory to be used and by having the object allocated selectively override the default `finalize` method. Measuring this latter point is important since some Java implementation are smarter than others in handling the case where an object does not override the finalizer.

Asynchrony

The RTSJ defines mechanisms to bind the execution of program logic to the occurrence of internal and/or external events. In particular, the RTSJ provides a way to associate an `AsyncEventHandler` to some application-specific or external events. Since event handling mechanisms are commonly used to develop real-time embedded systems [28], a robust and scalable implementation is essential. RTJPerf provide the following tests that measure the performance and scalability of RTSJ event dispatching mechanisms:

Asynchronous Event Handler Dispatch Delay Test. Several performance parameters are associated with asynchronous event handlers. One of the most important is the *dispatch latency*, which is the time from when an event is fired to when its handler is invoked. Events are often associated with alarms or other critical actions that must be handled within a short time and with high predictability. This RTJPerf test measures the dispatch latency for the different types of asynchronous event handlers prescribed by the RTSJ. The results of this test are reported in Section 6.4.3.

[2]The memory used by the scoped memory to allocate an object is not retrieved by the current allocation context, but is allocated in a platform-specifi c way, *e.g.*, using `malloc()` or `mmap()`.

Asynchronous Event Handler Priority Inversion Test. If the right data structure is not used to maintain the list of event handlers associated with an event, an unbounded priority inversion can occur during the dispatching of the event. This test therefore measures the degree of priority inversion that occurs when multiple handlers with different `SchedulingParameters` are registered for the same event. This test registers N handlers with an event in order of increasing importance. The time between the firing and the handling of the event is then measured for the highest priority event handler.

By comparing the results for this test with the result of the test described above, we can determine the degree of priority inversion present in the underlying RTSJ event dispatching implementation. Section 6.4.3, provides an analysis of the implementation of the current Reference Implementation (RI) and presents an implementation that overcomes some shortcomings of the RI.

Timers

Real-time embedded systems often use timers to perform certain actions at a given time in the future, as well as at periodic future intervals. For example, timers can be used to sample data, play music, transmit video frames, etc. Since real-time embedded systems often require predictable and precise timers, RTJPerf provides the following tests that measure the precision of the timers supported by an RTSJ implementation:

One Shot Timer Test. Different RTSJ timer implementations can trade off complexity and accuracy. RTJPerf therefore provides a test that fires a timer after a given time T has elapsed and measures the actual time elapsed. By running this test for different value of T, it is possible to determine the resolution at which timers can be used predictably.

Periodic Timer Test. Since periodic timers are often used for audio/video (A/V) playback, it is essential that little jitter is introduced by the RTSJ timer mechanism since humans are sensitive to jitter in A/V streams and tend to be annoyed by it. A quality RTSJ implementation should therefore provide precise, low-jitter periodic timers. RTJPerf provides a test that fires a timer with a period T and measures the actual elapsed time. By running this test for different values of T, it is possible to determine the resolution at which timers can be used predictably.

6.2.2 Statistics

Since RTJPerf tests produce traces in ASCII format, samples can be examined with any of the statistical tools available. However, since definitions used for some statistical properties, *e.g.* percentile, are sometimes different for different statistical tools, with RTJPerf is shipped PyStat [11], a set of Python scripts that perform basic statistical analysis on univariate data sets. The statistics computed by PyStat are based on those described in [29]. Throughout this Section box plots will be used extensively to show in a compact way performances results. Figure 6.1 shows the elements that make up a box plot. As it can be easily seen it provides information on the average, dispersion and outliers. It is worth pointing out that for each outlier value, a box plot, shows a small circle; it thus it provide visual feedback on the outlier frequency. The results shown in this thesis were produced

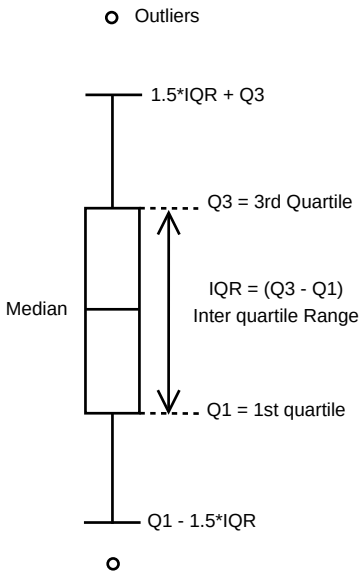


Figure 6.1: Box Plot.

using PyStat, while the graphics were produce using **R** [46] and XMGrace [27].

6.2.3 Timing Measurements in RTJPerf

An issue that arises when conducting benchmarks is which timing mechanism to use. To ensure fair measurements—irrespective of the time measurement mechanism provided by an RTSJ implementation—we implement our own native timers in RTJPerf. In particular, on all Pentium based systems, we use the *read-time stamp counter* RDTSC[3] instruction [9]

[3]The RDTSC is a 64 bit counter that can be read with the single x86 assembly instruction RDTSC.

to obtain timing resolutions that are a function of the CPU clock period and thus independent of system load.

This technique can also be used in multiprocessor systems if the OS initializes the RDTSC of different processors to the same value. The Linux SMP kernel performs this operation at boot time, so the initial value of the RDTSC is the same for all the processors. Once the counters are initialized to the same value, they stay in sync since their count increases at the same pace.

RTJPerf timer's implementation relies on the Java Native Interface (JNI) to invoke the platform-dependent mechanism that implements high resolution time. Although different Java platforms have different JNI performance, carefully implementing the JNI method can ensure sufficient accuracy of time measurements. The technique we use is shown in Figure 6.2, where two time measurements written in Java are performed at T_1 and T_2, *i.e.*, the RTJPerf timer is started at T_1 and stopped at T_2. The actual time measurement will

Figure 6.2: Time Measurement in RTSJ.

happen respectively at $T_a = T_1 + D_1$, and $T_b = T_2 + D_2$, where D_1 and D_2 represent the overhead of invoking the native implementation and executing the native call. If the high resolution time implementation is done in such a way that $D_1 = D_2$, and the time taken to return the time measurement to the Java code is negligible, we can then assume that $T_b - T_a = T_2 - T_1$. Moreover, we can assume that the timing measurement are largely independent of the underlying JNI implementation.

6.3 Overview of the Hardware and Software Testbed

The hardware platform used for our experimentation was an Intel Pentium IV 3 GHz with 512 MB RAM. The operating system used for our experimentation was Linux, but two different distributions with different kind of kernel were used. One of the two operating environment was based on a RedHat 9.0 distribution which was running the TimeSys Linux/RT 4.1.147 Kernel, the other operating environment was based on a Mandrake 10.0 distribution which was running a Kernel 2.6.3-4.

Table 6.1: RTSJ Platform v.s. Operating System Coverage

	TimeSys Linux/RT 4.1	**Linux 2.6**
Jamaica	Yes	Yes
jRate	Yes	Yes
JTime	Yes	No

AICAS Jamaica. The Jamaica Virtual Machine (JamaicaVM) is an implementation of the Java Virtual Machine Specification. It is a runtime system for the execution of applications written for the Java 2 Environment. It has been designed for realtime and embedded systems and offers a good support for this target domain. While originally Jamaica was not RTSJ compliant, it has been recently extended to support the RTSJ API. In order to improve the runtime efficiency, Jamaica is shipped with an ahead of time compiler which performs several optimizations. Jamaica other than running on any Linux platform, it is also supported on a series of real-time operating systems such as VxWorks, QNX, EUROS, ThreadX, ect.

TimeSys JTime. TimeSys JTime is a fully compliant implementation of Java [4, 26] that implements all the mandatory features specified in the RTSJ. JTime is based on a Java 2 Micro Edition (J2ME) JVM. It provides both supports for interpreted execution mode *i.e.*, and it also ships with a bytecode optimizer and an ahead of time compiler. JTime runs on TimeSys Linux platforms.

jRate. jRate is an open-source RTSJ-based extension of GCJ front-end and runtime systems that the author has entirely developed as part of his PhD. By relying on GCJ, jRate provides an ahead-of-time compiled platform for the development of RTSJ-compliant applications. Currently jRate is only supported on x86 based Linux platforms.

It is worth noticing that all the RTSJ platform under exam provide ahead of time compilation capabilities. This should not surprise the reader. As we said in at the beginning of this thesis, the main reason for using Java in real-time systems is to gain safety, and not necessarily portability. It should also be noticed that just in time compilation is not suitable for real-time systems since it adds unpredictable latencies.

6.3.1 Compiler and Runtime Options

The following options were used when compiling and running the tests for different real-time Java platforms:

Jamaica. When compiling the RTJPerf benchmark for Jamaica, the almost all the parameters were left to the distribution default values. The only parameter that were adjusted were the number of threads and the size of immortal and heap memory. The generated code was optimized for speed.

TimeSys JTime. When compiling the RTJPerf benchmark for JTime, the following options were used in the code optimizer -Xjnmlev6 -Xinllev5.

jRate. The Java code for the test was compiled with GCJ with the -O5 flag linked with the GCJ and jRate runtime libraries. The jRate configuration used was the default one.

In order to minimize the *noise* in the measurement, all the tests were ran on the target machine as super user, so to allow the use of real-time scheduling class, at run level 3, *e.g.* no windowing system, and with the network disabled.

6.3.2 Testbed Interference Estimation

The testbed described above does not include any hard real-time operating systems; this operating systems give complete control over the maximum interference that the operating system might introduce to a running application. In order to understand the level of noises introduced by a non hard-real-time OS, such as Linux, it is important to measure the Operating System interference. In this thesis we define the interference as the CPU time taken by the operating system for executing work which is not on behalf of the currently running application. In the remainder of this section we will propose a non-intrusive way of measuring the OS interference, and show the result that we have found for the target OS.

Estimating the Operating System Interference

The technique used in order to estimate the operating system interference is similar to that used by hourglass described in [40]. The basic idea is that of having a simple application that performs a very simple algorithm such as the one described in Figure 6.3.

```
n = 0;
while ((now = rdtsc()) < endTime) {
  delta = now - lastTime;

  if (delta > THRESHOLD) {
    data_sample[n] = delta;
    ++n;
  } else {
    // Code to balances the previous branch.
  }
  last = now;
}
```

Figure 6.3: The interference estimation kernel.

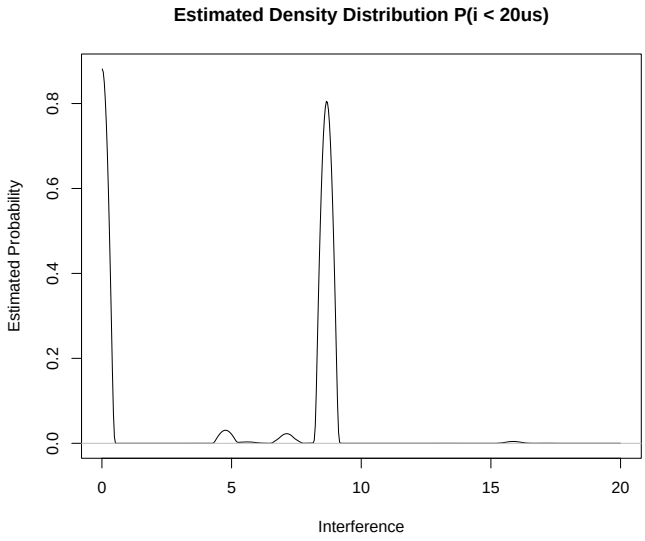


Figure 6.4: Lower Interference Density.

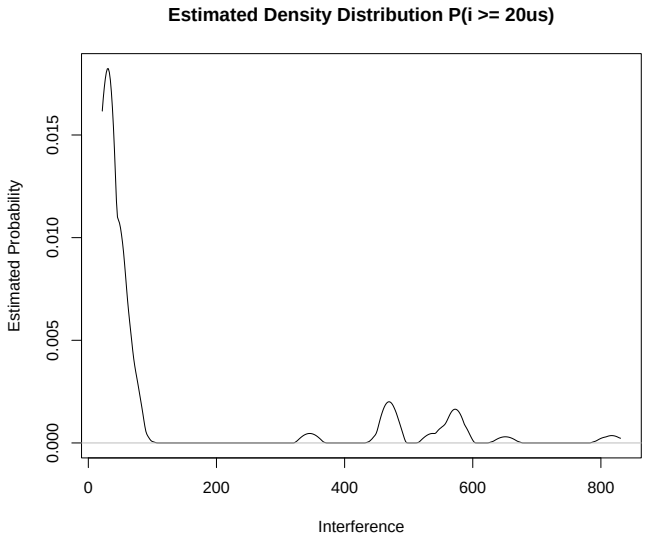


Figure 6.5: Upper Interference Density.

In Figure 6.3 the `rdtsc()` call reads the *read time stamp counter* that is present on all Intel Pentium processors, and which keeps the count of the clock ticks since the machine was booted. This register can be read with a single assembly instruction, and it allows to perform timing measurements that have the same resolution as the machine hardware clock (e.g. several GHz).

The algorithm described in Figure 6.3 allows the measurement of the operating system interference since it logs all the execution time that exceed a given threshold (see line 5). This threshold can be tuned to be roughly the execution time of the while loop, thus allowing the fine grain measurement of operating system interferences. Notice that in the algorithm reported in Figure 6.3 the branches are balanced; this is rather important to make sure that the code has two code path that are as similar as possible.

Operating System Interference

Below we describe the results found by executing the interference meter application, developed by using the algorithm described in Figure 6.3, on Linux. The interference meter application is single threaded application (e.g. has only the main thread), and it executes at the maximum priority allowed by the SCHED_FIFO scheduling class. The executing environment, is the same used for the execution of benchmarking tests, *i.e.*, no windowing system, no network, nor many other unneeded daemons such as sshd etc.

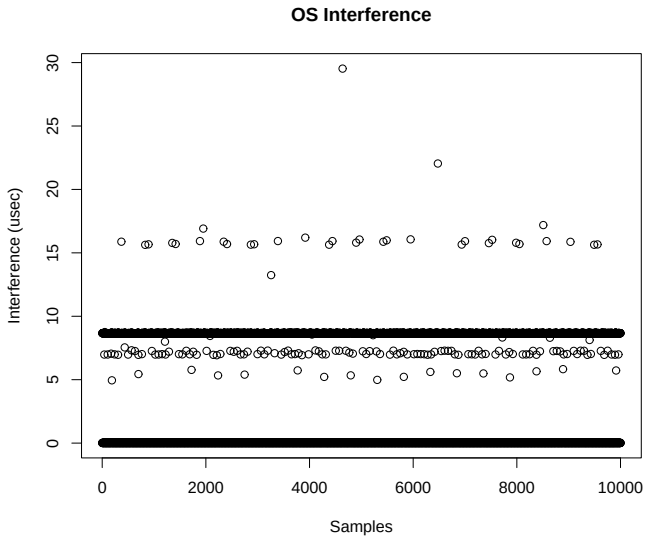


Figure 6.6: Subset of Interference Sample Data.

To measure the OS interference, after having performed some threshold tuning, we let the interference meter application run for 90 seconds. The dataset collected contained more than three millions data point. The result we found were similar for the different Linux version in our testbed, thus below we only report the results for Linux 2.6. Figure 6.4 and Figure 6.5 report the estimated probability density distribution, respectively, for interference values smaller than 20μsec and greater or equal to 20μsec. As it can be deduced from this estimated density function, the interference is spread over a very wide range, the maximum measured value was around 800μsec! However the most commonly introduced interference is centered around two data values (see Figure 6.4), one is around tens of nanoseconds and the other that is around tens of microseconds. In order to get a better understanding of the interference introduced by the operating system, Figure 6.6 and Figure 6.7, report a subset of the observed data. In particular, Figure 6.7 shows only the data points that are smaller then 1μsec. As it can be seen from these figures, the OS exposes several modes, each of which introduce a different interference value.

Using System Interference Estimates

Having an estimate of the operating system interferences can help in better understanding the measures obtained when benchmarking a software platform such as an RTSJ JVM. The knowledge of the interference density distribution can help in filtering some outliers, and in classifying some unfiltered outliers as due to operating system interference. This clearly

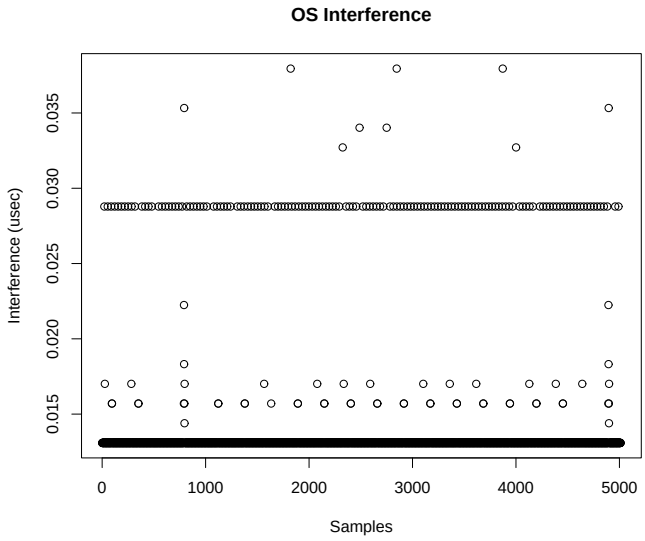


Figure 6.7: Subset of Interference Sample Data < 20 μsec.

depends on the time scale of the feature that is being measured, and on the knowledge of the implementation of the infrastructure under test, and of the test itself.

In the reminder for this Chapter, while looking at the RTJPerf results, please keep in mind the results shown in this section for better understanding the value of some outliers.

6.4 RTJPerf Benchmarking Results

This section presents the results obtained when running the tests discussed in Section 6.2.1 in the testbed described above. We analyze the results and explain why the various Java implementations performed differently.[4]

For each test a complete statistical characterization will be provided. Specifically, for each test we will provide (1) basic statistics such as mean, Standard Deviation (STD), mode, median, Coefficient of Variation (COV), and skewness, (2) representative quantiles, and (3) relevant intervals, such as sample interval, confidence intervals, outliers and non outliers intervals, and extreme and non extremes intervals. If some of this concept are not familiar, it would be helpful to briefly review them from any book of statistics [29] or consulting $MathWorld$ [49]. However, the statistical indexes used in this thesis are computed based on the definition given in [29].

[4]Explaining certain behaviors requires inspection of the source code of a particular JVM feature, which is not always feasible for Java implementations that are not open-source.

Table 6.2: jRate Yield Latency

jRate Yield Latency – Linux/RT

Basic Statistics	
Mean	0.795 μs
STD	0.001 μs
Mode	0.795 μs
Median	0.795 μs
COV	0.001
Skew	37.225

Quantile	
0.10	0.795 μs
0.25	0.795 μs
0.75	0.795 μs
0.90	0.795 μs
0.99	0.795 μs
0.999	0.799 μs

Intervals	
IQR	0.0 μs
NTR	0.0 μs
Interval	[0.787, 0.829] μs
90% Conf. Int.	[0.795, 0.795] μs
99% Conf. Int.	[0.795, 0.795] μs
99.9% Conf. Int.	[0.795, 0.795] μs
Top Outlier	[0.795, 0.829] μs
Non Outlier	[0.795, 0.795] μs
Top Extremes	[0.795, 0.829] μs
Non Extremes	[0.795, 0.795] μs

jRate Yield Latency – Linux 2.6

Basic Statistics	
Mean	0.634 μs
STD	0.135 μs
Mode	0.631 μs
Median	0.631 μs
COV	0.213
Skew	49.909

Quantile	
0.10	0.631 μs
0.25	0.631 μs
0.75	0.631 μs
0.90	0.631 μs
0.99	0.631 μs
0.999	0.634 μs

Intervals	
IQR	0.0 μs
NTR	0.0 μs
Interval	[0.617, 7.416] μs
90% Conf. Int.	[0.631, 0.636] μs
99% Conf. Int.	[0.629, 0.638] μs
99.9% Conf. Int.	[0.628, 0.64] μs
Top Outlier	[0.631, 7.416] μs
Non Outlier	[0.631, 0.631] μs
Top Extremes	[0.631, 7.416] μs
Non Extremes	[0.631, 0.631] μs

As a final note, the statistical processing of the data samples was performed by using R [46], Octave [25], and PyStat [11]. Graphics were generated using R [46] and XMGrace [27].

6.4.1 Thread Benchmark Results

Below, we present and analyze the results from the yield and synchronized context switch test, periodic thread test, and thread creation latency test, which were described in Section 6.2.1. Before reporting the results found by executing this experiments, it is worth noticing that for real-time systems having small and predicable the thread context switch is a very important. In fact the jitter introduced by the context switch has an impact in many other activities such as event handling. Moreover, the reader should realize that a context switch happens in Linux every 10 milliseconds, this is due to the fact that every 10 milliseconds the system timer ticks, and it needs to be handled.

Yield Context Switch Test. This test measures the time incurred for a thread context switch. The results we obtained are presented and analyzed below.

Test Settings. For each Java platform in our test suite, we collected 5,000 samples of the the context switch time, which we forced by explicitly yielding the CPU. Real-time threads were used for all the tested RTSJ implementations.

Table 6.3: JTime Yield Latency

JTime Yield Latency – Linux/RT

Basic Statistics	
Mean	1.263 μs
STD	0.081 μs
Mode	1.261 μs
Median	1.262 μs
COV	0.065
Skew	67.508

Quantile	
0.10	1.245 μs
0.25	1.253 μs
0.75	1.27 μs
0.90	1.278 μs
0.99	1.296 μs
0.999	1.323 μs

Intervals	
IQR	0.017 μs
NTR	0.033 μs
Interval	[1.173, 6.934] μs
90% Conf. Int.	[1.262, 1.265] μs
99% Conf. Int.	[1.26, 1.266] μs
99.9% Conf. Int.	[1.26, 1.267] μs
Top Outlier	[1.296, 6.934] μs
Non Outlier	[1.227, 1.296] μs
Top Extremes	[1.323, 6.934] μs
Non Extremes	[1.202, 1.321] μs

Table 6.4: Jamaica Yield Latency

Jamaica Yield Latency – Linux/RT

Basic Statistics	
Mean	2.55 μs
STD	1.759 μs
Mode	2.504 μs
Median	2.506 μs
COV	0.69
Skew	49.585

Quantile	
0.10	2.495 μs
0.25	2.501 μs
0.75	2.516 μs
0.90	2.534 μs
0.99	2.586 μs
0.999	2.611 μs

Intervals	
IQR	0.015 μs
NTR	0.039 μs
Interval	[2.436, 90.198] μs
90% Conf. Int.	[2.518, 2.581] μs
99% Conf. Int.	[2.492, 2.607] μs
99.9% Conf. Int.	[2.473, 2.626] μs
Top Outlier	[2.539, 90.198] μs
Non Outlier	[2.479, 2.538] μs
Top Extremes	[2.561, 90.198] μs
Non Extremes	[2.456, 2.561] μs

Jamaica Yield Latency – Linux 2.6

Basic Statistics	
Mean	2.294 μs
STD	1.256 μs
Mode	2.243 μs
Median	2.25 μs
COV	0.548
Skew	34.141

Quantile	
0.10	2.223 μs
0.25	2.236 μs
0.75	2.27 μs
0.90	2.289 μs
0.99	2.331 μs
0.999	9.043 μs

Intervals	
IQR	0.034 μs
NTR	0.066 μs
Interval	[2.195, 46.122] μs
90% Conf. Int.	[2.271, 2.317] μs
99% Conf. Int.	[2.253, 2.335] μs
99.9% Conf. Int.	[2.239, 2.349] μs
Top Outlier	[2.321, 46.122] μs
Non Outlier	[2.185, 2.321] μs
Top Extremes	[2.376, 46.122] μs
Non Extremes	[2.134, 2.372] μs

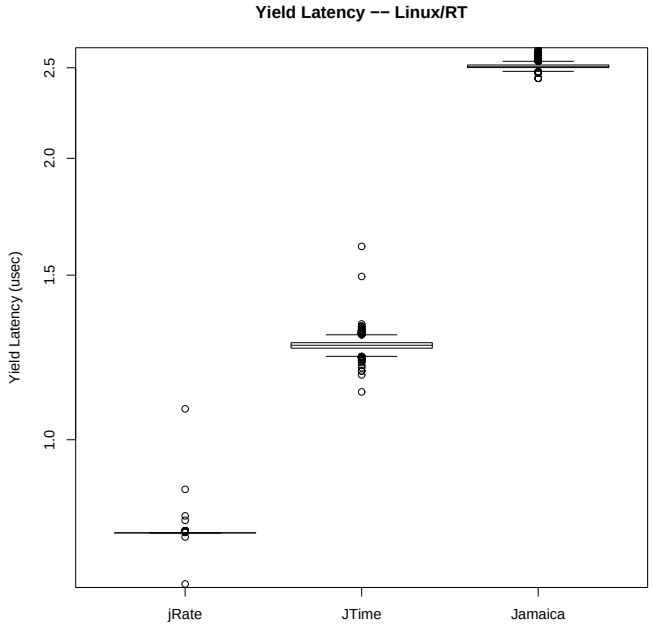


Figure 6.8: Yield Time.

Test Results. Table 6.2, 6.3, and 6.4 reports the relevant statistical information for the platform under test, while Figure 6.8 shows a box plot of the yield time.

Results Analysis. Below, we analyze the results of the tests that measure the average context switch time, its dispersion, and its worst-case behavior for the different test settings:

- **Average Measures**—Table 6.2, 6.3 and 6.4 shows that jRate has the lowest expected context switch time, and its value is half of that measured for JTime. On the other hand, JTime has an average context switch time that is half of that measured for Jamaica. As it can be inspected by reading the confidence intervals for the mean in the Tables 6.2, 6.3 and 6.4, our estimate of the mean context switch time is very accurate. The higher context switch time experimented for Jamaica (4 times higher than jRate) is due to the fact that this RTSJ implementation uses user-level threads, while both jRate and JTime rely directly on kernel level threads. The context switch time changes slightly between Linux 2.6 and Linux/RT. Both jRate and Jamaica have a slightly better mean context switch on Linux 2.6.

Table 6.5: jRate Synch Yield Latency

jRate Synch Yield Latency – Linux/RT

Basic Statistics	
Mean	3.178 μs
STD	0.004 μs
Mode	3.175 μs
Median	3.178 μs
COV	0.001
Skew	0.84

Quantile	
0.10	3.175 μs
0.25	3.175 μs
0.75	3.182 μs
0.90	3.182 μs
0.99	3.188 μs
0.999	3.203 μs

Intervals	
IQR	0.007 μs
NTR	0.007 μs
Interval	[3.161, 3.206] μs
90% Conf. Int.	[3.178, 3.179] μs
99% Conf. Int.	[3.178, 3.179] μs
99.9% Conf. Int.	[3.178, 3.179] μs
Top Outlier	[3.193, 3.206] μs
Non Outlier	[3.164, 3.192] μs
Top Extremes	[3.203, 3.206] μs
Non Extremes	[3.154, 3.203] μs

jRate Synch Yield Latency – Linux 2.6

Basic Statistics	
Mean	2.007 μs
STD	0.307 μs
Mode	2.001 μs
Median	1.999 μs
COV	0.153
Skew	22.243

Quantile	
0.10	1.979 μs
0.25	1.992 μs
0.75	2.001 μs
0.90	2.001 μs
0.99	2.001 μs
0.999	8.874 μs

Intervals	
IQR	0.009 μs
NTR	0.022 μs
Interval	[1.935, 9.083] μs
90% Conf. Int.	[2.002, 2.013] μs
99% Conf. Int.	[1.997, 2.018] μs
99.9% Conf. Int.	[1.994, 2.021] μs
Top Outlier	[2.025, 9.083] μs
Non Outlier	[1.978, 2.015] μs
Top Extremes	[2.029, 9.083] μs
Non Extremes	[1.965, 2.028] μs

- **Dispersion Measures**—Based on the data reported in Table 6.2, 6.3, and 6.4, and on the box plot depicted in Figure 6.8 it can be easily seen that jRate is the most predictable between the tested platforms. It is worth noticing that jRate, on both Linux/RT and Linux 2.6, has a practically zero Inter-Quartile Range (IQR) and Nineth-Tenth Quantile Range (NTR). Something else worth noticing is that the quantiles from the 0.10 to the 0.99 are the same for jRate. This is index of great predictability. While the context switch time measured for JTime and Jamaica is more dispersed than the jRate's one, the results show quite predictable context switches time.

- **Worst-case Measures**—The interval, outliers and extremes ranges reported in Table 6.2, 6.3, and 6.4, show that all the implementation have a very good 0.999 quantile, but on the other hand, the sample interval, *e.g.* min and max, measured for JTime and Jamaica are quite wide, especially if compared with jRate. This means that overall jRate has better worst case behaviour.

Synchronized Context Switch Test. This test measures the context switch time incurred when a higher priority thread T_H enters a monitor owned by a lower priority thread T_L. The results we obtained are presented and analyzed below.

Test Settings. The settings used for this test are the same as the previous one.

Table 6.6: JTime Synch Yield Latency

JTime Synch Yield Latency – Linux/RT

Basic Statistics	
Mean	3.595 μs
STD	0.026 μs
Mode	3.601 μs
Median	3.596 μs
COV	0.007
Skew	22.437

Quantile	
0.10	3.571 μs
0.25	3.58 μs
0.75	3.608 μs
0.90	3.618 μs
0.99	3.639 μs
0.999	3.655 μs

Intervals	
IQR	0.028 μs
NTR	0.047 μs
Interval	[3.543, 4.839] μs
90% Conf. Int.	[3.594, 3.595] μs
99% Conf. Int.	[3.594, 3.596] μs
99.9% Conf. Int.	[3.594, 3.596] μs
Top Outlier	[3.651, 4.839] μs
Non Outlier	[3.538, 3.65] μs
Top Extremes	[4.839, 4.839] μs
Non Extremes	[3.496, 3.692] μs

Table 6.7: Jamaica Synch Yield Latency

Jamaica Synch Yield Latency – Linux/RT

Basic Statistics	
Mean	18.24 μs
STD	0.251 μs
Mode	18.226 μs
Median	18.235 μs
COV	0.014
Skew	13.059

Quantile	
0.10	18.166 μs
0.25	18.199 μs
0.75	18.269 μs
0.90	18.297 μs
0.99	18.353 μs
0.999	23.803 μs

Intervals	
IQR	0.07 μs
NTR	0.131 μs
Interval	[11.197, 24.19] μs
90% Conf. Int.	[18.235, 18.245] μs
99% Conf. Int.	[18.232, 18.248] μs
99.9% Conf. Int.	[18.229, 18.251] μs
Top Outlier	[18.374, 24.19] μs
Non Outlier	[18.094, 18.374] μs
Top Extremes	[23.053, 24.19] μs
Non Extremes	[17.989, 18.479] μs

Jamaica Synch Yield Latency – Linux 2.6

Basic Statistics	
Mean	10.387 μs
STD	0.813 μs
Mode	10.26 μs
Median	10.268 μs
COV	0.078
Skew	9.58

Quantile	
0.10	10.182 μs
0.25	10.221 μs
0.75	10.351 μs
0.90	10.462 μs
0.99	17.081 μs
0.999	19.453 μs

Intervals	
IQR	0.13 μs
NTR	0.28 μs
Interval	[9.996, 27.157] μs
90% Conf. Int.	[10.372, 10.401] μs
99% Conf. Int.	[10.36, 10.413] μs
99.9% Conf. Int.	[10.351, 10.422] μs
Top Outlier	[10.547, 27.157] μs
Non Outlier	[10.026, 10.546] μs
Top Extremes	[11.162, 27.157] μs
Non Extremes	[9.831, 10.741] μs

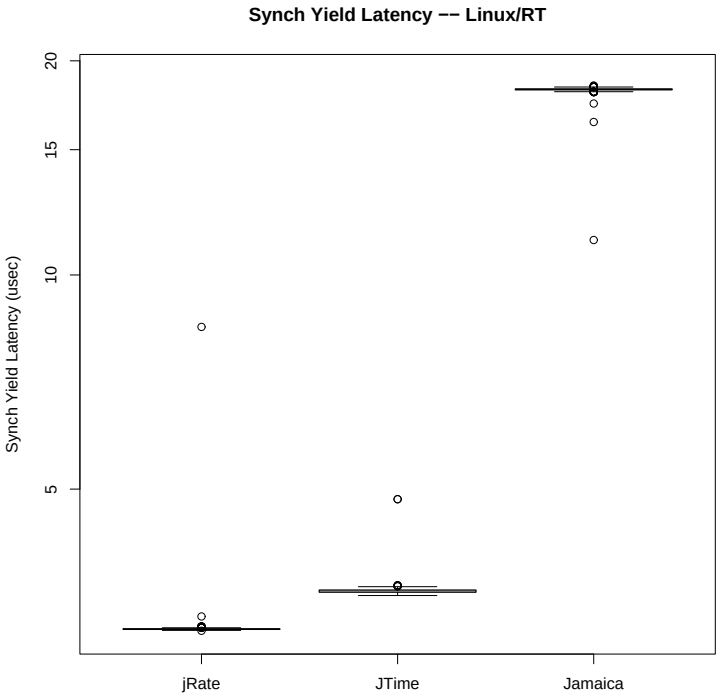


Figure 6.9: Synch Yield Time.

Test Results. Table 6.5, 6.6, and 6.7 shows the statistics for the aggregate context switch time[5] for the different RTSJ implementations we tested.

Results Analysis. Below we analyze the results of the test that measure the synchronized context switch time:

- **Average Measures**—Table 6.5, 6.6, and 6.7 show how jRate and JTime have a similar average synchronized context switch time–this is far smaller than Jamaica. As in the case of the yield context switch this might be due to the fact that Jamaica uses user level threads. It is also interesting to observe that, in this test, jRate and Jamaica perform better on Linux 2.6. It is indeed quite surprising to see that Jamaica, on Linux 2.6, has an average synchronized context switch that is half of the one measured on Linux/RT.

- **Dispersion Measures**—From the data reported in Table 6.5, 6.6, and 6.7, is can be seen that all the implementation are rather predictable, with jRate and JTime showing tighter values than Jamaica. It is worth noticing that values are more dispersed on Linux 2.6 than in Linux/RT.

- **Worst-case Measures**—In this test jRate and JTime have much better worst case behaviour than Jamaica, with JTime being slightly better than jRate, at least for what concert the sample interval. It is interesting to notice that the sample interval is much wider on Linux 2.6 than on Linux/RT. Finally, observing Figure 6.9, it can be seen how the wide interval experienced by jRate is due to an outlier which is very likely coming from some interference from the operating system.

Thread Creation Latency Test. This test measures the time needed to create a thread. Thread creation in RTSJ platform involves many operation and checks concerning memory areas and scope stack. Thus the thread creation time is affected by the memory area implementation. The results we obtained for this test are presented and analyzed below.

Test Settings. For each RTSJ platform in our test suite, we collected 5000 samples of the thread creation time and thread start time.

[5] Aggregate context switch time is defined as the time taken to perform the context switch from T_H to T_L, plus the time taken for the T_L to exit the monitor, plus the time taken by T_H to enter the monitor.

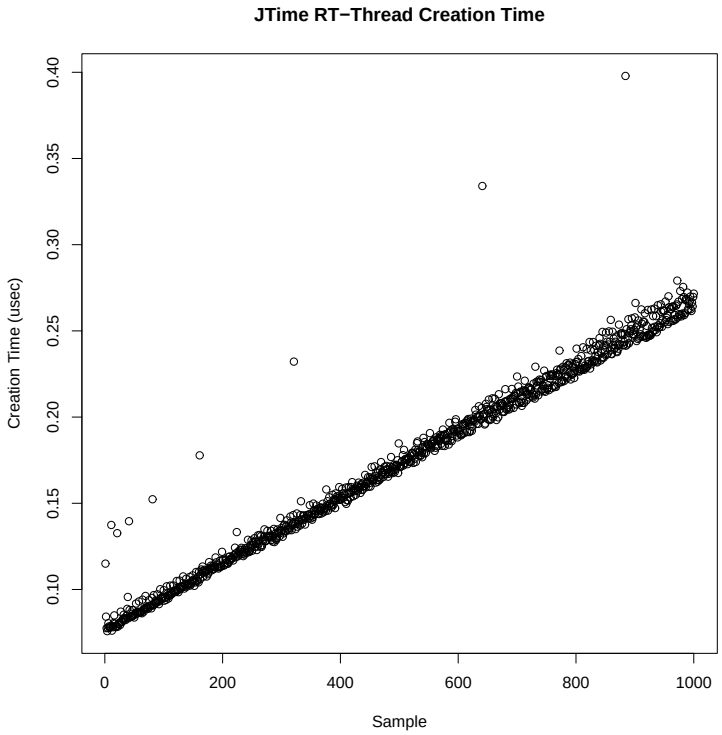


Figure 6.10: JTime Real-Time Thread Creation Time.

Test Results. For this particular test, we seem to have stressed a bug in JTime. The creation time grows linearly with the sample as depicted in Figure 6.10. We had discovered a similar problem in the TimeSys reference implementation in [17]. Table 6.8, and 6.9 report the statistics for the thread creation time.

Results Analysis. Below, we analyze the results of the tests that measure the average-case, the dispersion, and the worst-case for thread creation time and thread start time.

- **Average Measures**—As noted above, this test exposes a bug in JTime, thus, it is not possible to provide any meaningful statistics on the average creation time, since, as appears from Figure 6.10, the creation time grows with the sample number. On the other hand, the results reported in Table 6.8, and 6.9 show that jRate has a much better average creation time than Jamaica. On the tested platforms jRate is more than 20 times faster than Jamaica. This big difference is likely to reside in the way in which jRate implements both the scope stack management and the runtime checks.

Table 6.8: jRate Real-Time Thread Creation Latency

jRate RT-Thread Creation Latency – Linux/RT

Basic Statistics	
Mean	8.022 μs
STD	0.348 μs
Mode	7.79 μs
Median	7.955 μs
COV	0.043
Skew	8.569

Quantile	
0.10	7.74 μs
0.25	7.834 μs
0.75	8.118 μs
0.90	8.456 μs
0.99	8.834 μs
0.999	14.354 μs

Intervals	
IQR	0.284 μs
NTR	0.716 μs
Interval	[7.476, 15.453] μs
90% Conf. Int.	[8.016, 8.028] μs
99% Conf. Int.	[8.01, 8.033] μs
99.9% Conf. Int.	[8.007, 8.037] μs
Top Outlier	[8.545, 15.453] μs
Non Outlier	[7.408, 8.544] μs
Top Extremes	[8.99, 15.453] μs
Non Extremes	[6.982, 8.97] μs

jRate RT-Thread Creation Latency – Linux 2.6

Basic Statistics	
Mean	5.89 μs
STD	0.609 μs
Mode	5.806 μs
Median	5.831 μs
COV	0.103
Skew	13.15

Quantile	
0.10	5.783 μs
0.25	5.805 μs
0.75	5.869 μs
0.90	5.92 μs
0.99	6.129 μs
0.999	13.495 μs

Intervals	
IQR	0.064 μs
NTR	0.137 μs
Interval	[5.689, 19.479] μs
90% Conf. Int.	[5.879, 5.901] μs
99% Conf. Int.	[5.87, 5.91] μs
99.9% Conf. Int.	[5.863, 5.917] μs
Top Outlier	[5.965, 19.479] μs
Non Outlier	[5.709, 5.965] μs
Top Extremes	[6.061, 19.479] μs
Non Extremes	[5.613, 6.061] μs

The confidence intervals shown in Table 6.8, and 6.9 are very narrow, thus our estimate of the average creation time is very accurate. Finally it is worth noticing, that in this tests, jRate performs better on Linux 2.6, while Jamaica performs better on Linux/RT.

- **Dispersion Measures**—As it can be seen from Table 6.8, and 6.9, both implementation are quite predictable, with jRate showing somewhat a denser distribution of values. Even if Jamaica has an higher standard deviation, the coefficient of variation is very similar for the two platform.

- **Worst-case Measures**—Quantiles and intervals, show that for this test the worst case behaviour is rather distant from the average, for both jRate and Jamaica. This is true for both Linux/RT and Linux 2.6. These results suggest to preallocate thread whenever possible, since the creation time, can have rather bad extremes values. These values are infrequent, but yet, when designing hard real-time systems wort-case measures are those that matter.

Thread Startup Latency Test. This test measures the time needed to start a thread which has already been created. The results we obtained are presented and analyzed below.

Table 6.9: Jamaica Real-Time Thread Creation Latency

Jamaica RT-Thread Creation Latency – Linux/RT

Basic Statistics	
Mean	207.181 μs
STD	6.36 μs
Mode	206.393 μs
Median	206.592 μs
COV	0.031
Skew	23.692

Quantile	
0.10	206.19 μs
0.25	206.358 μs
0.75	206.904 μs
0.90	207.347 μs
0.99	216.802 μs
0.999	322.518 μs

Intervals	
IQR	0.546 μs
NTR	1.157 μs
Interval	[205.625, 468.982] μs
90% Conf. Int.	[207.066, 207.297] μs
99% Conf. Int.	[206.972, 207.391] μs
99.9% Conf. Int.	[206.903, 207.459] μs
Top Outlier	[207.729, 468.982] μs
Non Outlier	[205.539, 207.723] μs
Top Extremes	[208.553, 468.982] μs
Non Extremes	[204.72, 208.542] μs

Jamaica RT-Thread Creation Latency – Linux 2.6

Basic Statistics	
Mean	219.652 μs
STD	12.942 μs
Mode	217.328 μs
Median	217.784 μs
COV	0.059
Skew	40.61

Quantile	
0.10	217.058 μs
0.25	217.337 μs
0.75	218.616 μs
0.90	225.063 μs
0.99	228.932 μs
0.999	273.63 μs

Intervals	
IQR	1.279 μs
NTR	8.005 μs
Interval	[216.015, 875.258] μs
90% Conf. Int.	[219.418, 219.887] μs
99% Conf. Int.	[219.226, 220.079] μs
99.9% Conf. Int.	[219.086, 220.219] μs
Top Outlier	[220.544, 875.258] μs
Non Outlier	[215.418, 220.535] μs
Top Extremes	[223.566, 875.258] μs
Non Extremes	[213.5, 222.453] μs

Table 6.10: jRate Real-Time Thread Startup Latency

jRate RT-Thread Startup Latency – Linux/RT

Basic Statistics	
Mean	66.348 μs
STD	3.921 μs
Mode	65.872 μs
Median	65.991 μs
COV	0.059
Skew	23.247

Quantile	
0.10	64.836 μs
0.25	65.338 μs
0.75	66.704 μs
0.90	67.447 μs
0.99	74.08 μs
0.999	154.7 μs

Intervals	
IQR	1.366 μs
NTR	2.611 μs
Interval	[63.793, 186.169] μs
90% Conf. Int.	[66.277, 66.419] μs
99% Conf. Int.	[66.219, 66.477] μs
99.9% Conf. Int.	[66.176, 66.519] μs
Top Outlier	[68.757, 186.169] μs
Non Outlier	[63.289, 68.753] μs
Top Extremes	[70.848, 186.169] μs
Non Extremes	[61.24, 70.802] μs

jRate RT-Thread Startup Latency – Linux 2.6

Basic Statistics	
Mean	5.638 μs
STD	1.016 μs
Mode	5.449 μs
Median	5.456 μs
COV	0.18
Skew	6.175

Quantile	
0.10	5.388 μs
0.25	5.414 μs
0.75	5.511 μs
0.90	5.58 μs
0.99	11.158 μs
0.999	14.355 μs

Intervals	
IQR	0.097 μs
NTR	0.192 μs
Interval	[5.282, 18.19] μs
90% Conf. Int.	[5.619, 5.656] μs
99% Conf. Int.	[5.604, 5.671] μs
99.9% Conf. Int.	[5.593, 5.682] μs
Top Outlier	[5.657, 18.19] μs
Non Outlier	[5.269, 5.656] μs
Top Extremes	[5.811, 18.19] μs
Non Extremes	[5.123, 5.802] μs

Table 6.11: JTime Real-Time Thread Startup Latency

JTime RT-Thread Startup Latency – Linux/RT

Basic Statistics	
Mean	1227.816 μs
STD	22.199 μs
Mode	1229.34 μs
Median	1230.48 μs
COV	0.018
Skew	1.344

Quantile	
0.10	1197.9 μs
0.25	1212.8 μs
0.75	1241.5 μs
0.90	1247.64 μs
0.99	1320.74 μs
0.999	1381.89 μs

Intervals	
IQR	28.7 μs
NTR	49.74 μs
Interval	[1183.04, 1381.89] μs
90% Conf. Int.	[1226.911, 1228.72] μs
99% Conf. Int.	[1226.175, 1229.457] μs
99.9% Conf. Int.	[1225.636, 1229.996] μs
Top Outlier	[1309.51, 1381.89] μs
Non Outlier	[1169.75, 1284.55] μs
Top Extremes	[1329.61, 1381.89] μs
Non Extremes	[1126.7, 1327.6] μs

Test Settings. For each RTSJ platform in our test suite, we collected 5000 samples of the thread creation time and thread start time.

Test Results. Table 6.10, 6.11, and 6.12, report the statistics for the thread start time.

Results Analysis. Below, we analyze the results of the tests that measure the average-case, the dispersion, and the worst-case for thread start time.

- **Average Measures**—The results reported in Table 6.10, 6.11, and 6.12, show how jRate's and Jamaica's average thread start time are roughly 20 and 10 times, respectively, smaller than the one measured for JTime. Moreover, it is very interesting to observe how jRate performs much better on Linux 2.6 than on Linux/RT. The difference in the average thread start time is a factor of 10! The other thing that should be observed is that jRate has to do a system call in order to create a thread, since it relies on kernel thread, and has also to correctly setup the scope stack, and perform the appropriate safety checks. On the other hand, since Jamaica relies on user level threads, it should not need to make a system call and, on theory, this should save it some time, compared to jRate.

- **Dispersion Measures**—From the data reported in Table 6.10, 6.11, and 6.12, we can see that while JTime has a much higher average start time, it does not necessarily

Table 6.12: Jamaica Real-Time Thread Startup Latency

Jamaica RT-Thread Startup Latency – Linux/RT

Basic Statistics	
Mean	116.889 μs
STD	3.501 μs
Mode	116.367 μs
Median	116.47 μs
COV	0.03
Skew	27.997

Quantile	
0.10	116.087 μs
0.25	116.232 μs
0.75	117.231 μs
0.90	117.631 μs
0.99	122.23 μs
0.999	172.651 μs

Intervals	
IQR	0.999 μs
NTR	1.544 μs
Interval	[114.758, 237.915] μs
90% Conf. Int.	[116.825, 116.952] μs
99% Conf. Int.	[116.773, 117.004] μs
99.9% Conf. Int.	[116.735, 117.042] μs
Top Outlier	[118.793, 237.915] μs
Non Outlier	[114.733, 118.73] μs
Top Extremes	[121.183, 237.915] μs
Non Extremes	[113.235, 120.228] μs

Jamaica RT-Thread Startup Latency – Linux 2.6

Basic Statistics	
Mean	98.404 μs
STD	3.662 μs
Mode	98.749 μs
Median	98.758 μs
COV	0.037
Skew	0.227

Quantile	
0.10	91.982 μs
0.25	98.236 μs
0.75	99.071 μs
0.90	99.635 μs
0.99	106.476 μs
0.999	116.733 μs

Intervals	
IQR	0.835 μs
NTR	7.653 μs
Interval	[83.053, 143.165] μs
90% Conf. Int.	[98.338, 98.471] μs
99% Conf. Int.	[98.284, 98.525] μs
99.9% Conf. Int.	[98.244, 98.564] μs
Top Outlier	[100.353, 143.165] μs
Non Outlier	[96.983, 100.324] μs
Top Extremes	[101.673, 143.165] μs
Non Extremes	[95.731, 101.576] μs

behave less predictably. It has in fact a rather small coefficient of variance, and its standard deviation is relatively small. Both jRate and Jamaica show a much more predictable behaviour on Linux 2.6. This difference in behaviour is interesting, and might stems from the fact that jRate and Jamaica have been designed and implemented to run on plain Linux and does not takes advantages of proprietary TimeSys Linux/RT features. However, once again, it should be noticed that Linux 2.6, while not being tagged as a real-time operating systems, it shows a relatively good behaviour.

- **Worst-case Measures**—It is worth noticing how jRate and Jamaica have better worst case parameters on Linux 2.6. However, the data reported in Table 6.10, 6.11, and 6.12, shows that the worst case startup time can be quite higher than the average behaviour. This can be noticed, observing how the quantiles grow.

Periodic Thread Test. This test measures the accuracy with which the `waitForNextPeriod()` method in the `RealtimeThread` class schedules the thread's execution periodically. The results we obtained are presented and analyzed below.

Test Settings. This test runs a `RealtimeThread` that does nothing but reschedule its execution for the next period. The actual time between each activation was measured and 1000 of these measurements were made for the periods $1ms$, $5ms$, $10ms$, $50ms$,

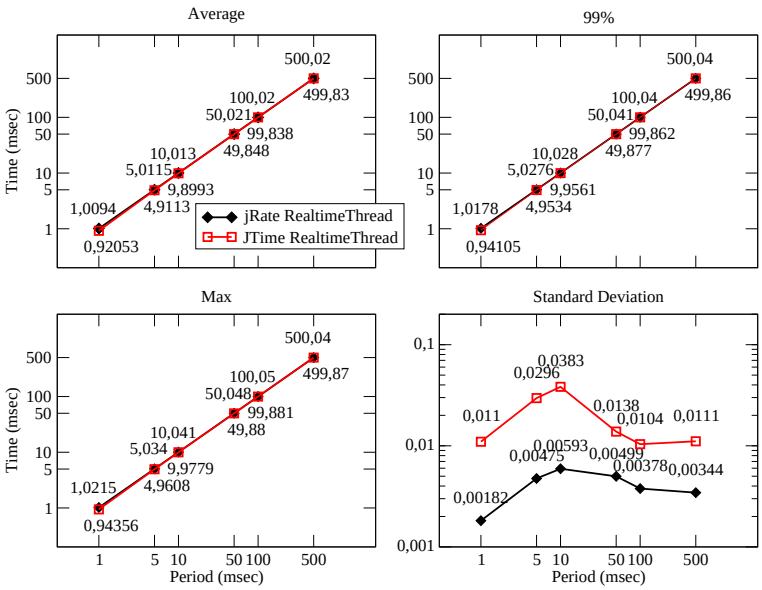


Figure 6.11: Measured Period Statistics.

100ms, and 500ms. The test was performed only on Linux/RT for jRate and JTime.

Test Results Figure 6.11 shows average and dispersion values that we measured for this test.[6]

Results Analysis Below we analyze the results of the test that measures the accuracy with which periodic a thread's logic is activated:

- **Average Measures**—Figure 6.11 shows that both jRate and the JTime have an average period that is quite close to the target period. Whereas JTime is always at least several hundreds of microseconds early, however, jRate is at most several tens of microseconds late. To understand the reason for this behavior, we inspected the JTime implementation of periodic threads, (*i.e.*, at the implementation of `waitForNextPeriod()`) and found that a JNI method call is used to wait for the next period. Without the source for the JTime's JVM, it is hard to tell exactly how the native method is implemented. On the TimeSyS Linux/RT kernel, jRate relies on the `nanosleep()` system call to implement periodic thread behavior. To produce more accurate periods, a calibration test can be run at configuration time to obtain a

[6]Whenever the plot for jRate and the JTime overlap, the values for jRate are shown above the graph and the value for the JTime are shown below the graph.

slack time that should be considered as an approximation of the overhead of calling the `waitForNextPeriod()` and then getting the control back.

- **Dispersion Measures**—Figure 6.11 shows the dispersion of the measured period for both jRate and the JTime has the same trend. While jRate generally has less dispersed values than the JTime, both implementation are quite predictable.

- **Worst-case Measures**—As shown in Figure 6.11 both jRate and the JTime have worst-case behavior that is close to the average-case values and the 99% bound. In general, jRate's worst-case values are closer to the average, but the JTime values are not much further away.

6.4.2 Memory Benchmark Results

Below we present and analyze the results of the RTJPerf memory benchmarks that we ran.

Allocation Time Test. This test measures the allocation time for different types of scoped memory. The results we obtained are presented and analyzed below.

Test Settings This tests measure the average allocation time incurred by the RTSJ `LTMemory` and `VTMemory`, and by jRate non standard scoped memories `CTMemory` and `CTPrivateMemory` which were described in Chapter 5. The RTJPerf allocation time test was performed for allocation sizes ranging from 1 Byte to 8 KBytes. Each test samples 1,000 values of the allocation time for the given allocation size.

Test Results The data obtained by running the allocation time tests were processed to obtain relevant statistics of the allocation time. We represent the measurements using box plots since they provide a nice and compact way of expressing both average and dispersion indexes since they indicate the following information:

- How predictable is the behavior of a scope memory implementation

- How much variation in allocation time can occur and

- How the worst-case behavior compares to the average-case and to the first and third quartile.

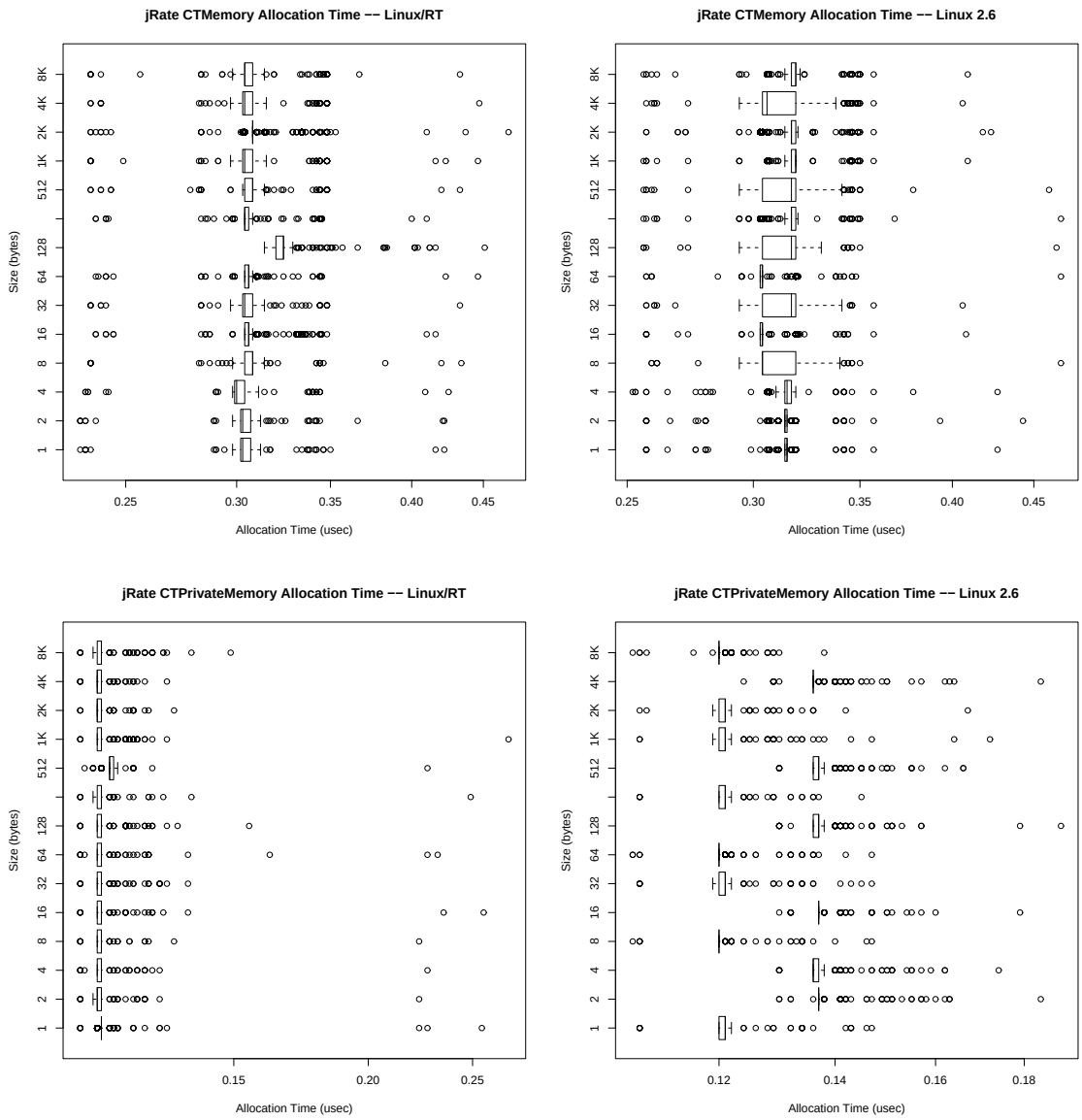


Figure 6.12: jRate Allocation Time.

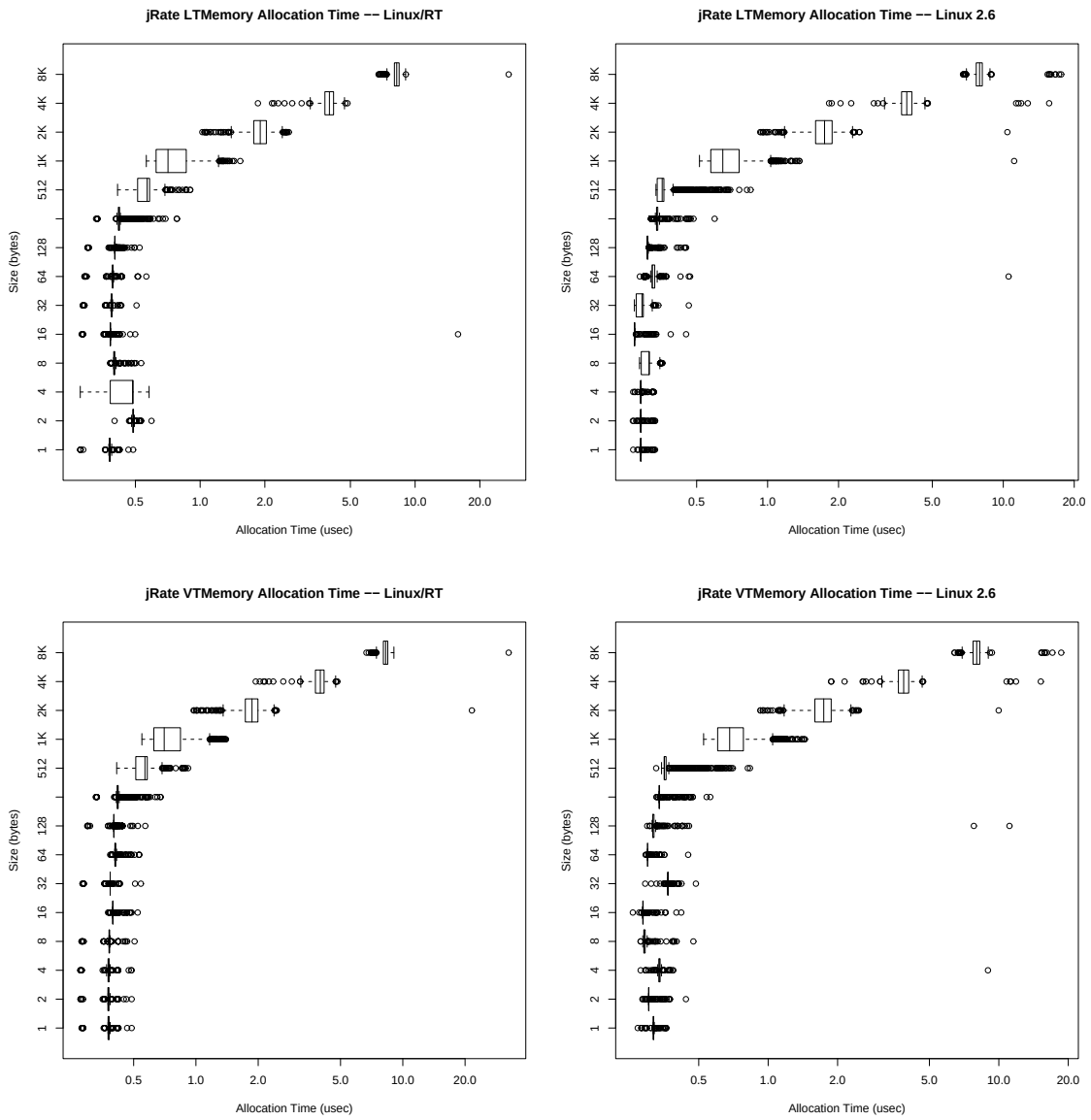


Figure 6.13: jRate Allocation Time.

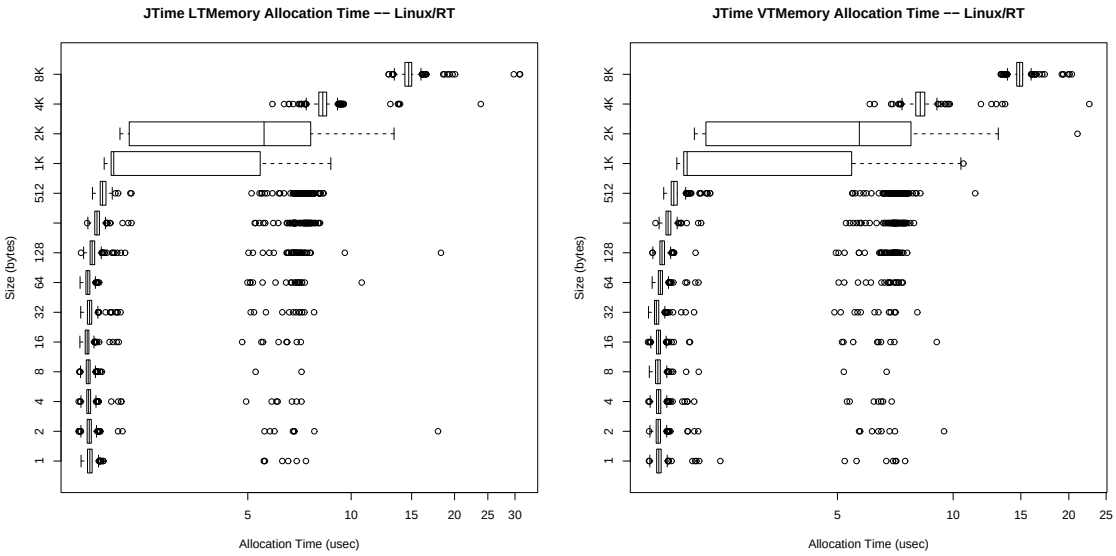


Figure 6.14: JTime Allocation Time.

Figure 6.12, 6.13, 6.14, 6.15 show the resulting allocation time for the different test runs. Results are illustrated by means of box plot.

Results Analysis We now analyze the results of the tests that measured the average- and worst-case allocation times, along with the dispersion for the different test settings:

- **Average Measures**—As shown in From Figures 6.12, 6.13, 6.14, 6.15, it can be easily seen that, regardless of the RTSJ implementation, `LTMemory` and `VTMemory` provide linear time allocation with respect to the allocated memory size. From the result found, it is apparent `LTMemory` and `VTMemory` provide very similar (practically identical) performances.

  As it can be easily seen from Figure 6.12, jRate's `CTMemory` and `CTPrivateMemory` have an allocation time that is independent of the allocated chunk, which helps analyze the timing of RTSJ code, even without knowing the amount of memory that will be needed. By comparing the results show in Figures 6.12, 6.14, 6.15, it can be seen that jRate's `CTMemory` allocator can be between 5 and 100 times faster than JTime and Jamaica's `LTMemory`.

- **Dispersion Measures**—The with of the box plot depicted in Figures 6.12, 6.13, 6.14, 6.15, provide a measure of the dispersion for the different allocation time cases. The results clearly show how jRate's `CTMemory` and `CTPrivateMemory` are the

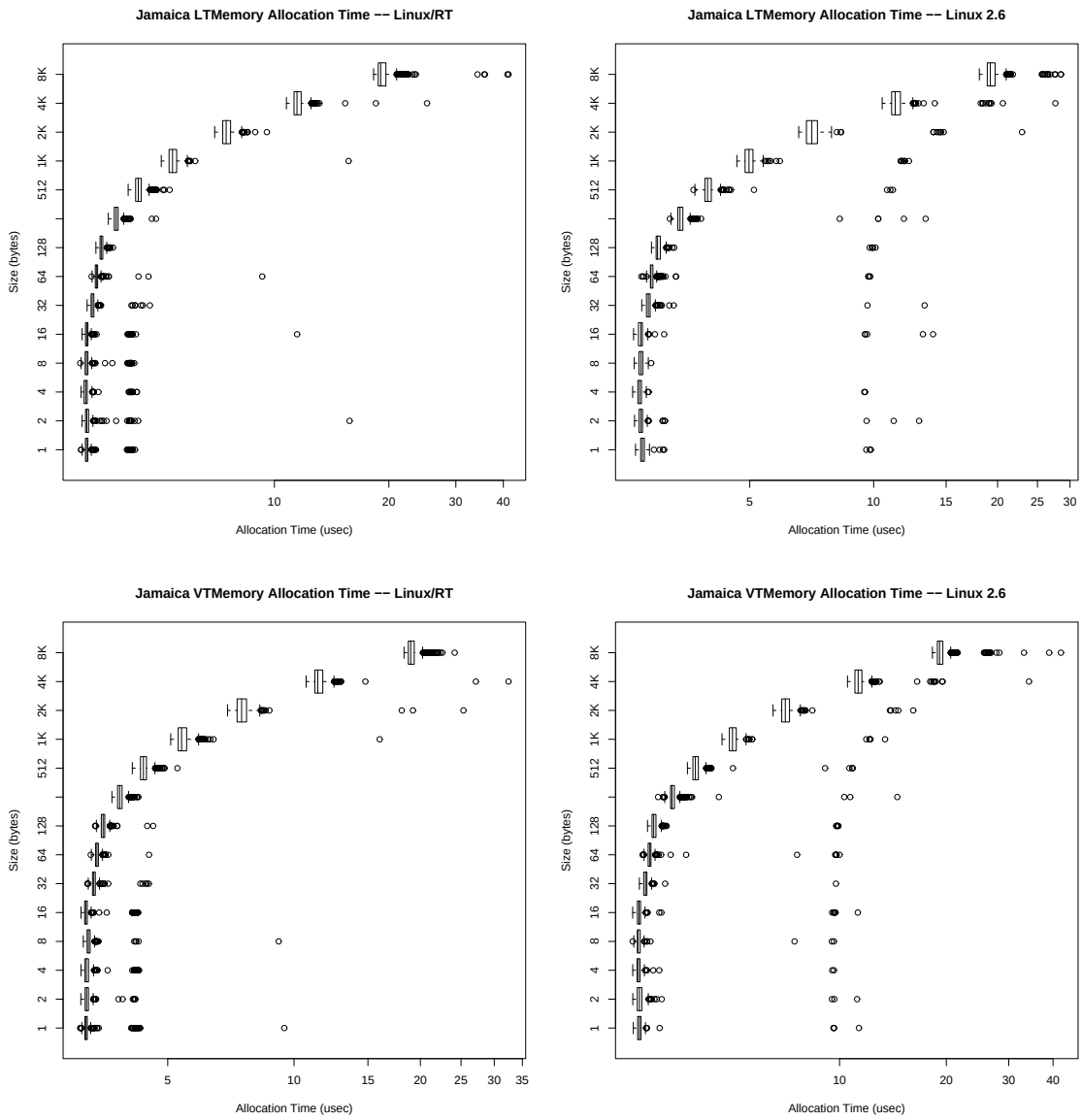


Figure 6.15: Jamaica Allocation Time.

less dispersed data samples. The predictability shown by these memory areas are due to their allocator with simply have to perform pointer arithmetic. Moreover, `CTPrivateMemory`, does not have to lock any mutex thus providing both better response time and tighter data values. It is also interesting to observe how JTime allocation time looses predictability in allocation time for memory chunks of size of 1 and 2 Kbytes. This behavior is rather strange, and might stem from the allocator implementation. It is worth pointing out that nor jRate nor Jamaica expose similar strangeness.

Finally it is worth noticing that the result found for the different RTSJ implementation are similar on both Linux/RT and Linux 2.6.

- **Worst-case Measures**—Figures 6.12, 6.13, 6.14, 6.15, show the bounds on the allocation time. From these results we can see that for `LTMemory` `VTMemory`, JTime and Jamaica have a similar worst case allocation time; for the same kind of memory jRate is slightly better. On the other hand, jRate's `CTMemory` and `CTPrivateMemory` have a much smaller sample interval, thus resulting in the overall most predictable memory area implementation.

Scoped Memory Lifetime Test

Test Settings To measure scoped memory creation, enter, and exit time, we ran the RTJPerf scoped memory timing test for memory sizes ranging from 256 to 256K bytes. The test was designed to ensure that the allocated objects overrode the finalizer, which enabled a worst-case measurement of the exit time. For each test, 1000 values were sampled for each of the measured variables.

Test Results Figure 6.16 to Figure 6.29 are reported results represented using box plots.

Results Analysis Below we analyze the results of the test that measure the creation, enter, exit, and execution time for a scoped memory area.

- **Average Measures**—From Figure 6.16 to Figure 6.29 we can see how the jRate scoped memories implementation have a constant creation time. On the other hand, JTime and Jamaica have a creation time that grows linearly with the size of the memory being created. jRate has better creation time performances thanks to its memory

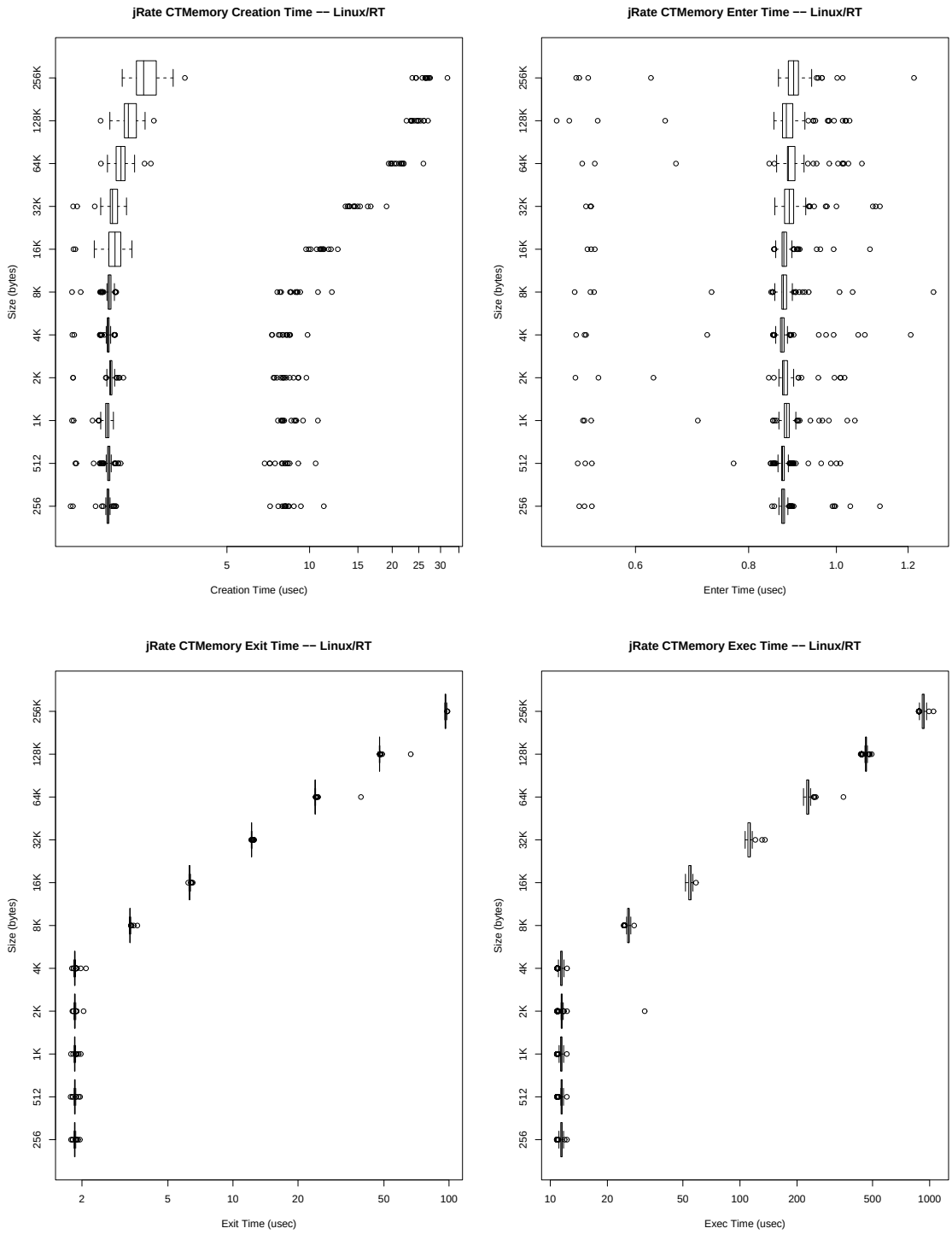


Figure 6.16: jRate CTMemory Timings.

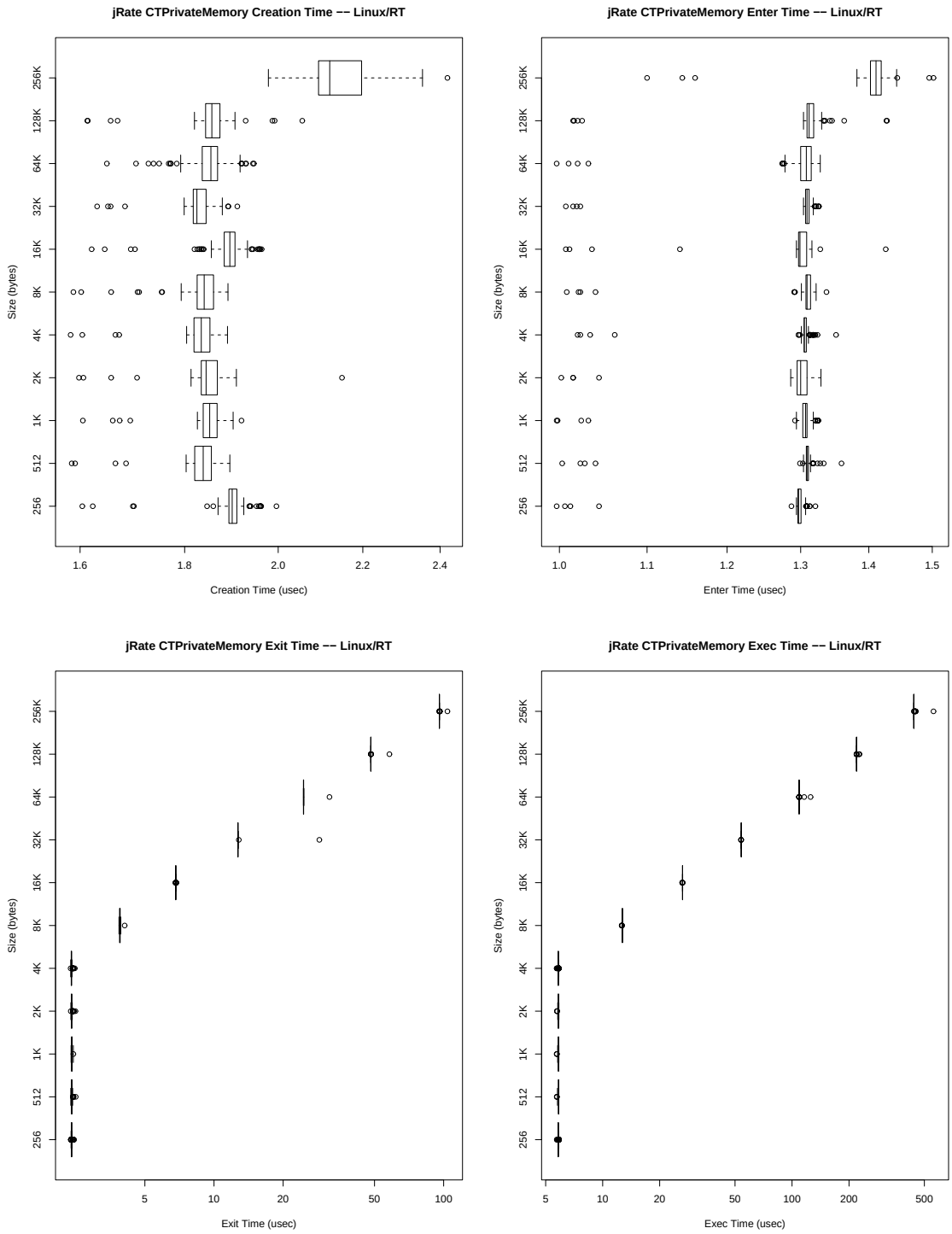


Figure 6.17: jRate CTPrivateMemory Timings.

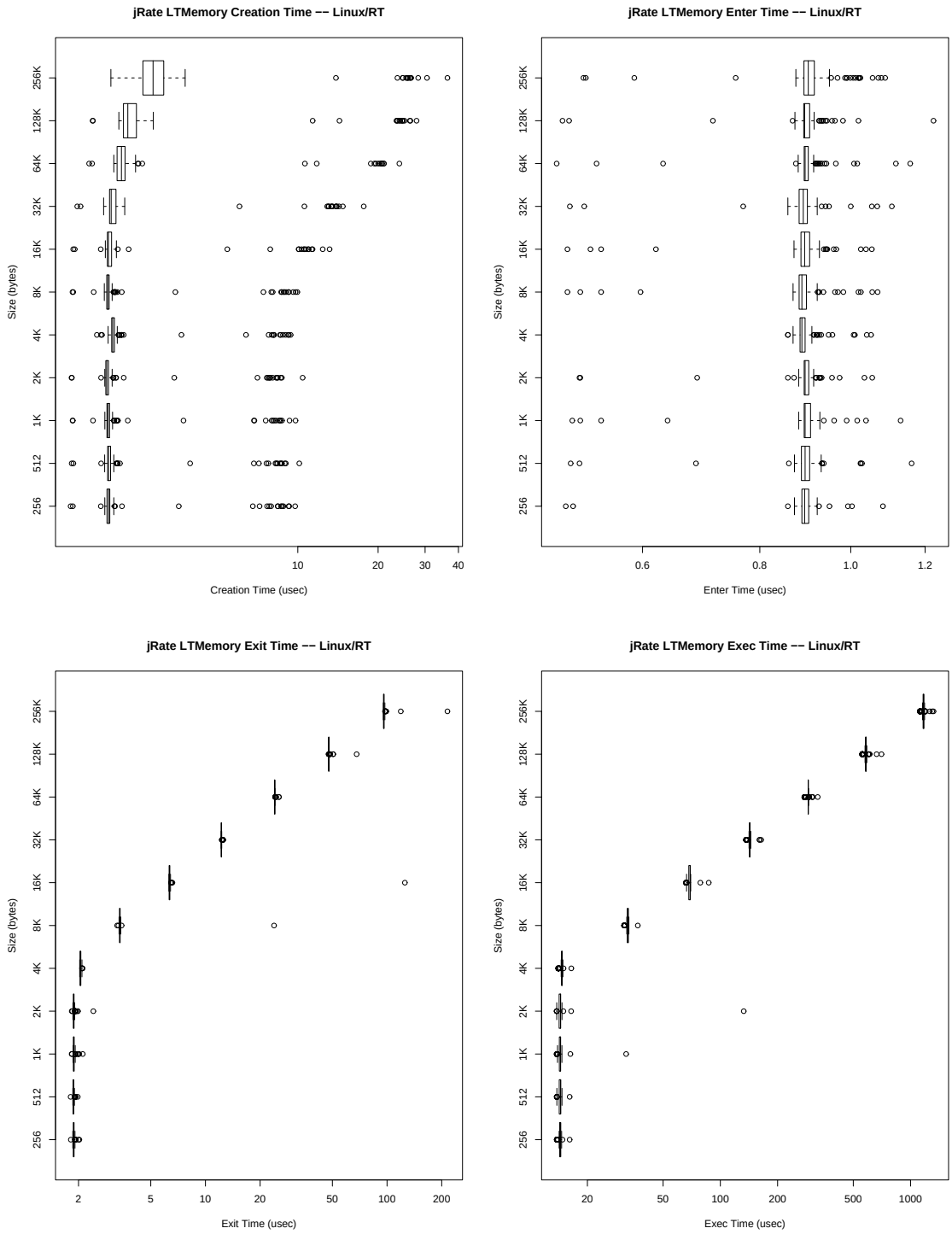


Figure 6.18: jRate LTMemory Timings.

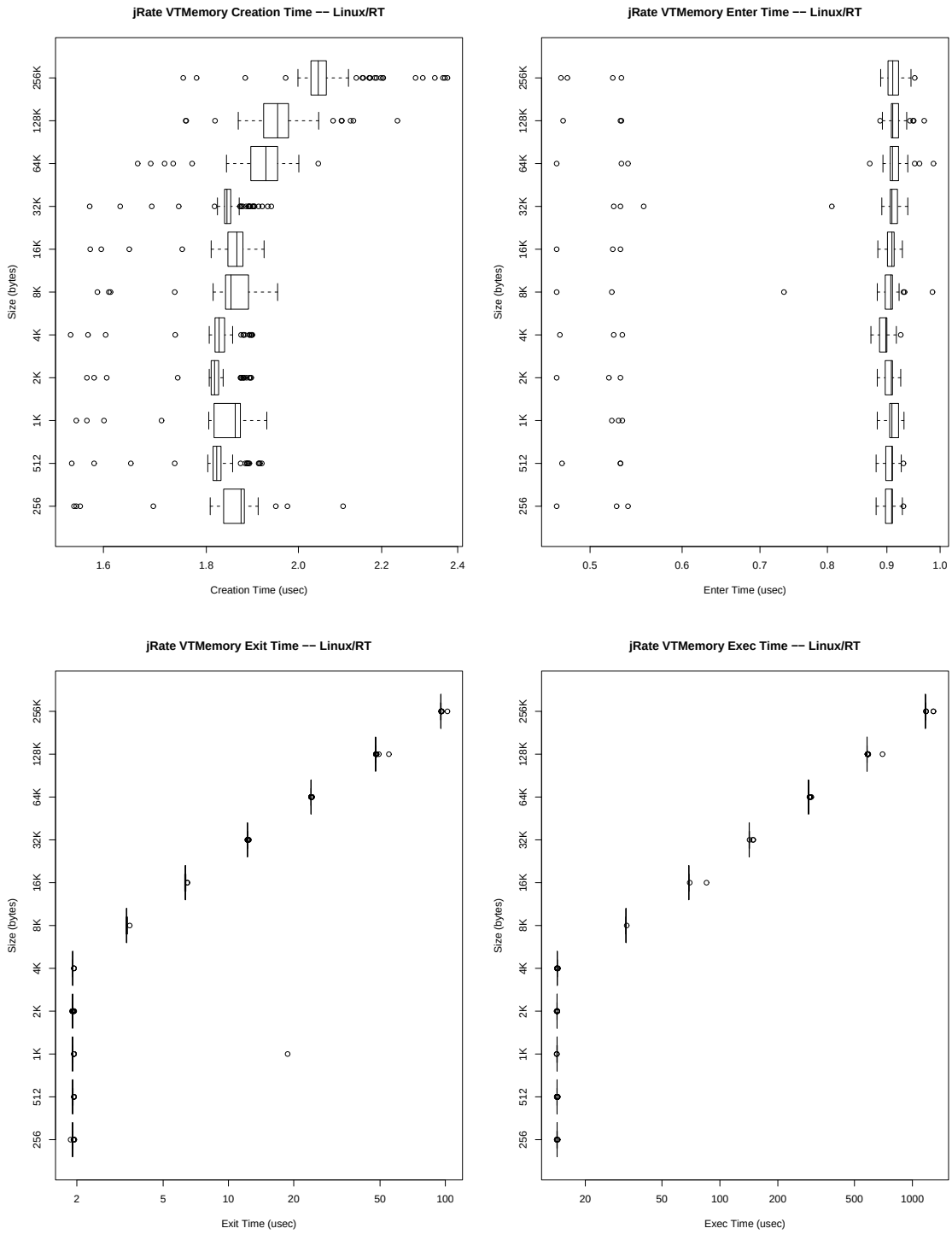


Figure 6.19: jRate VTMemory Timings.

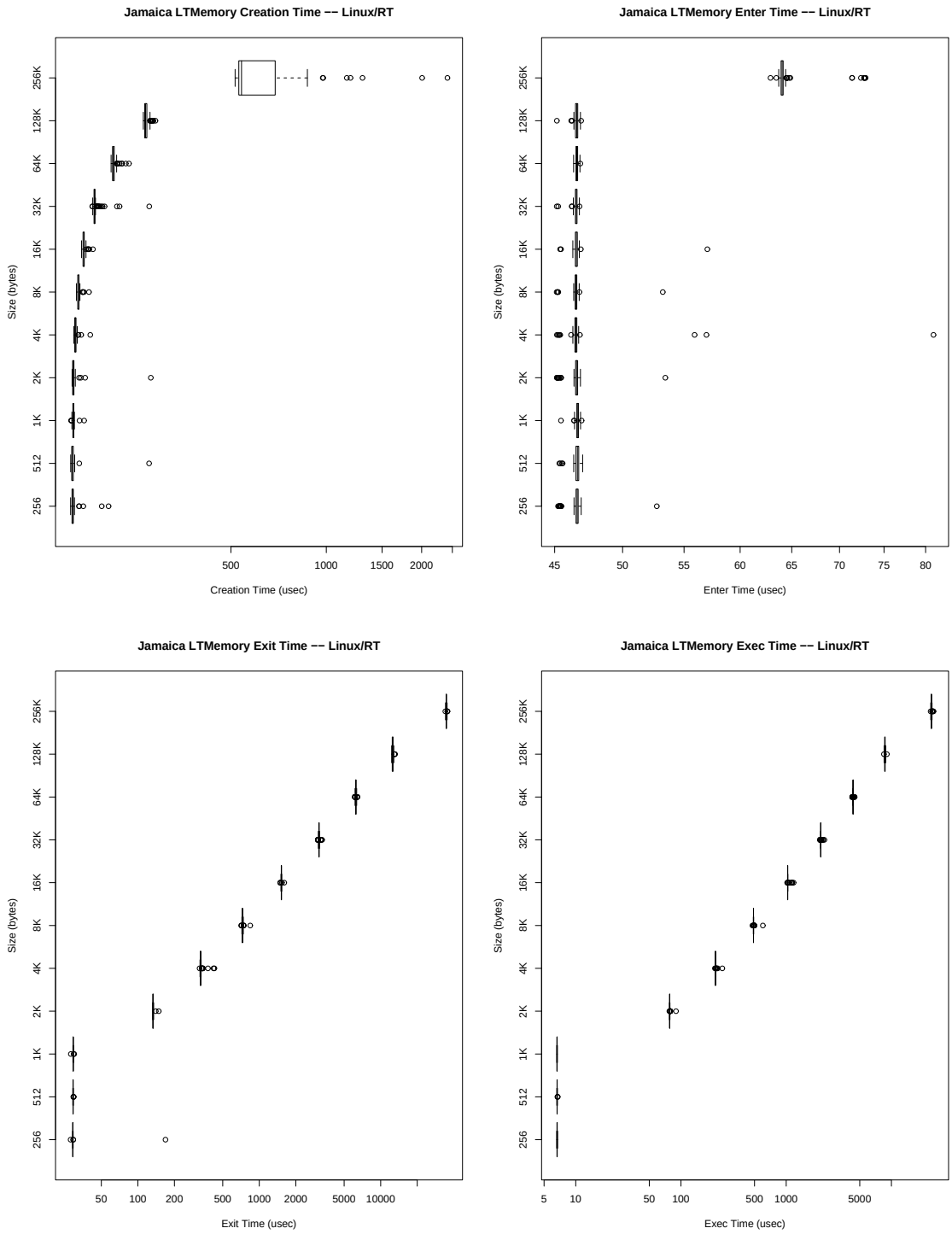


Figure 6.20: Jamaica LTMemory Timings.

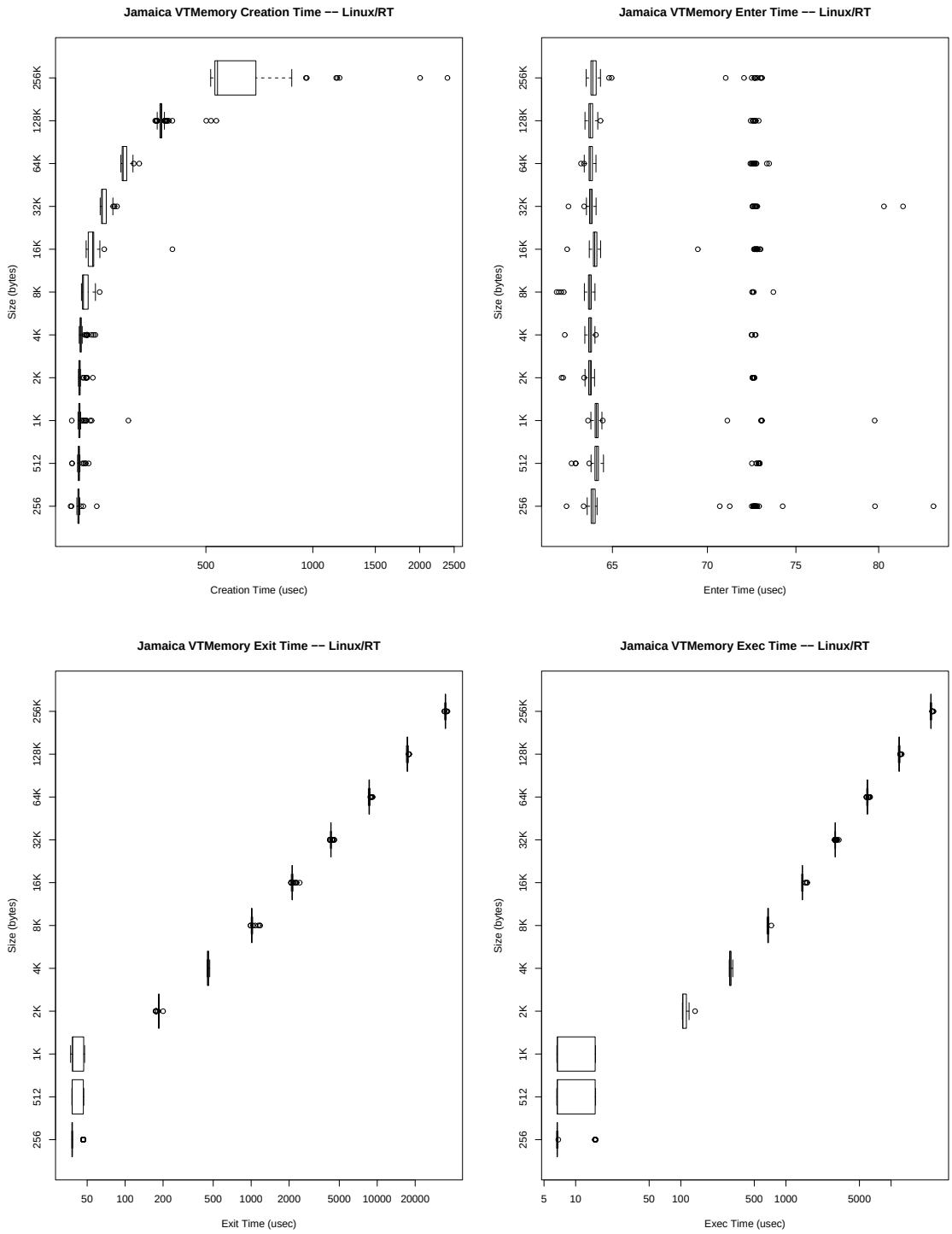


Figure 6.21: Jamaica VTMemory Timings.

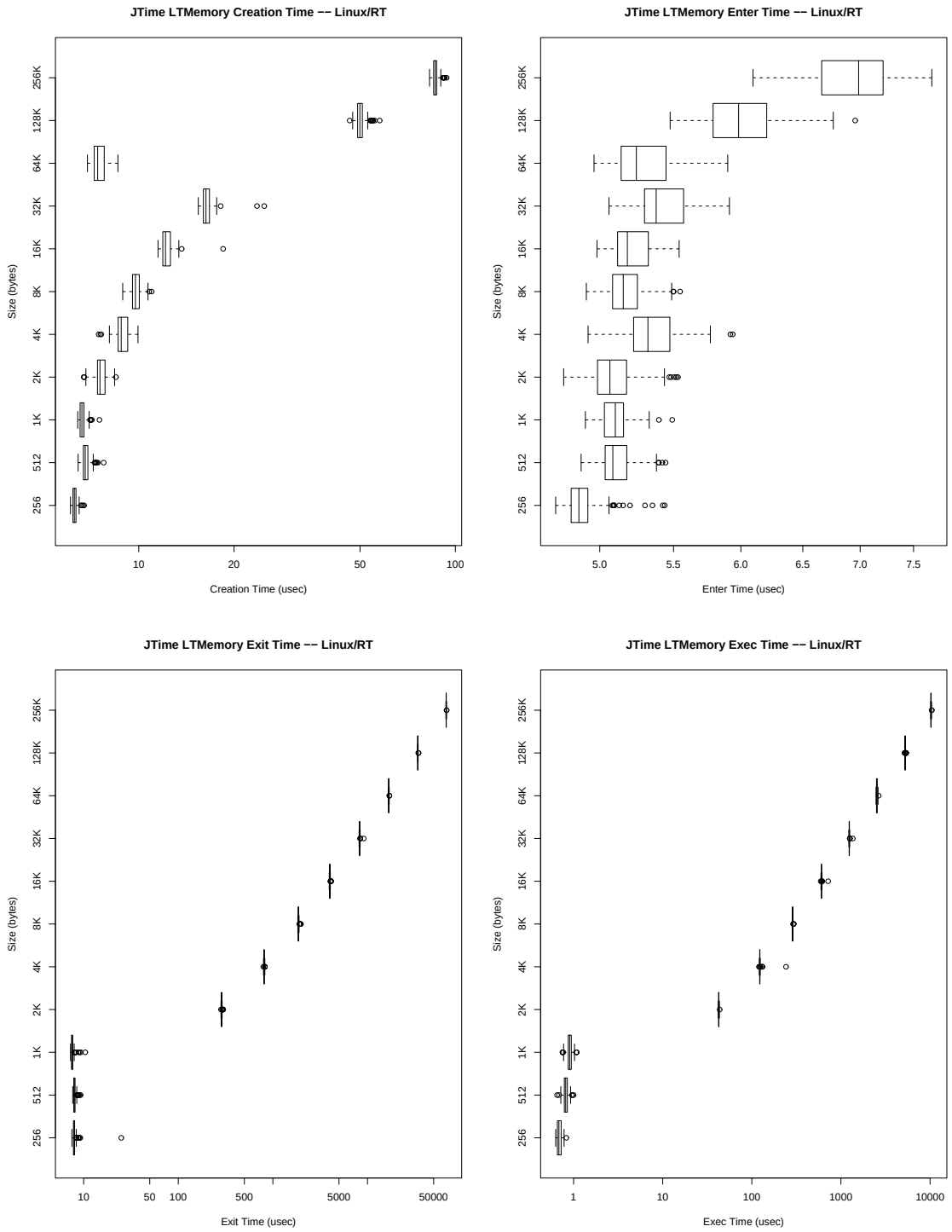


Figure 6.22: JTime LTMemory Timings Allocation Time.

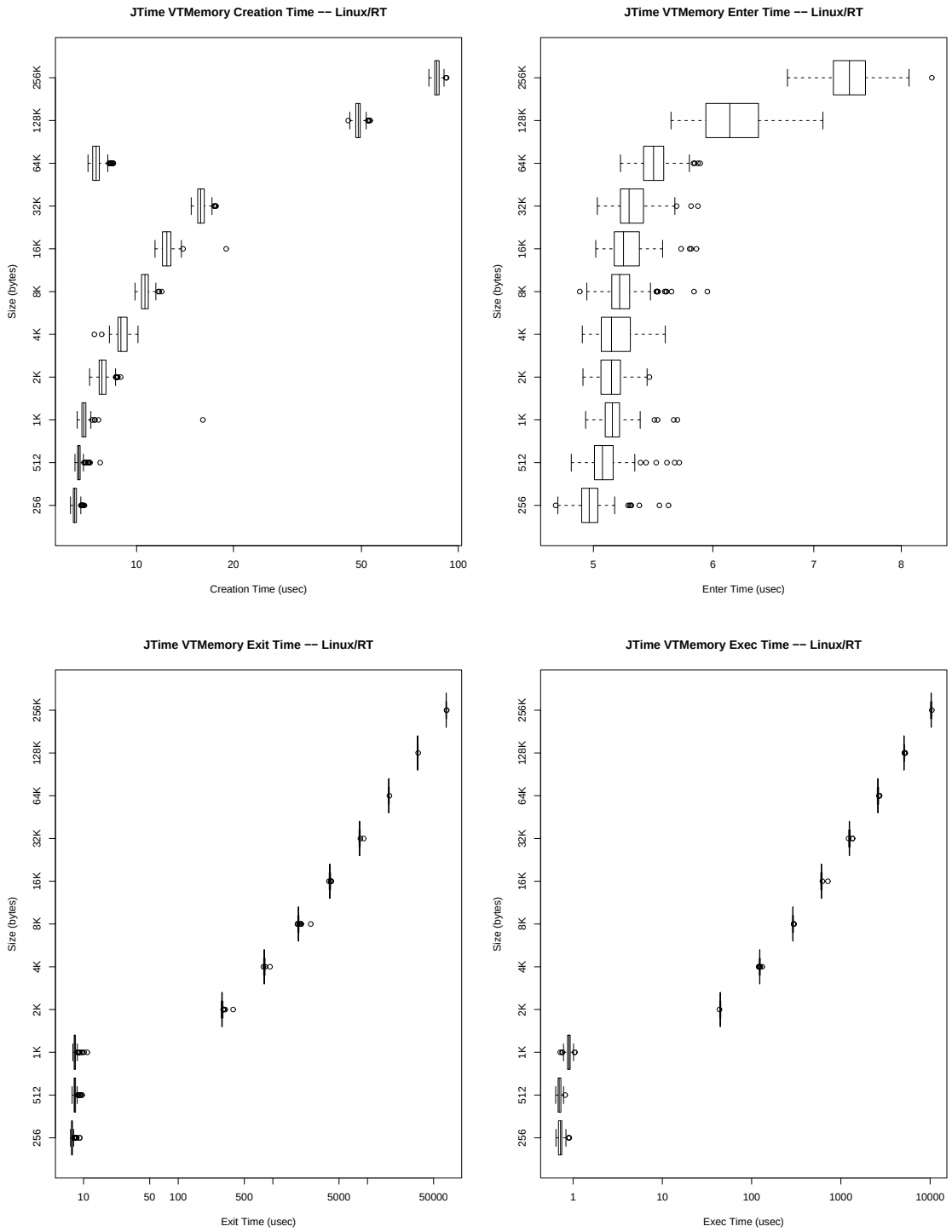


Figure 6.23: JTime VTMemory Timings Allocation Time.

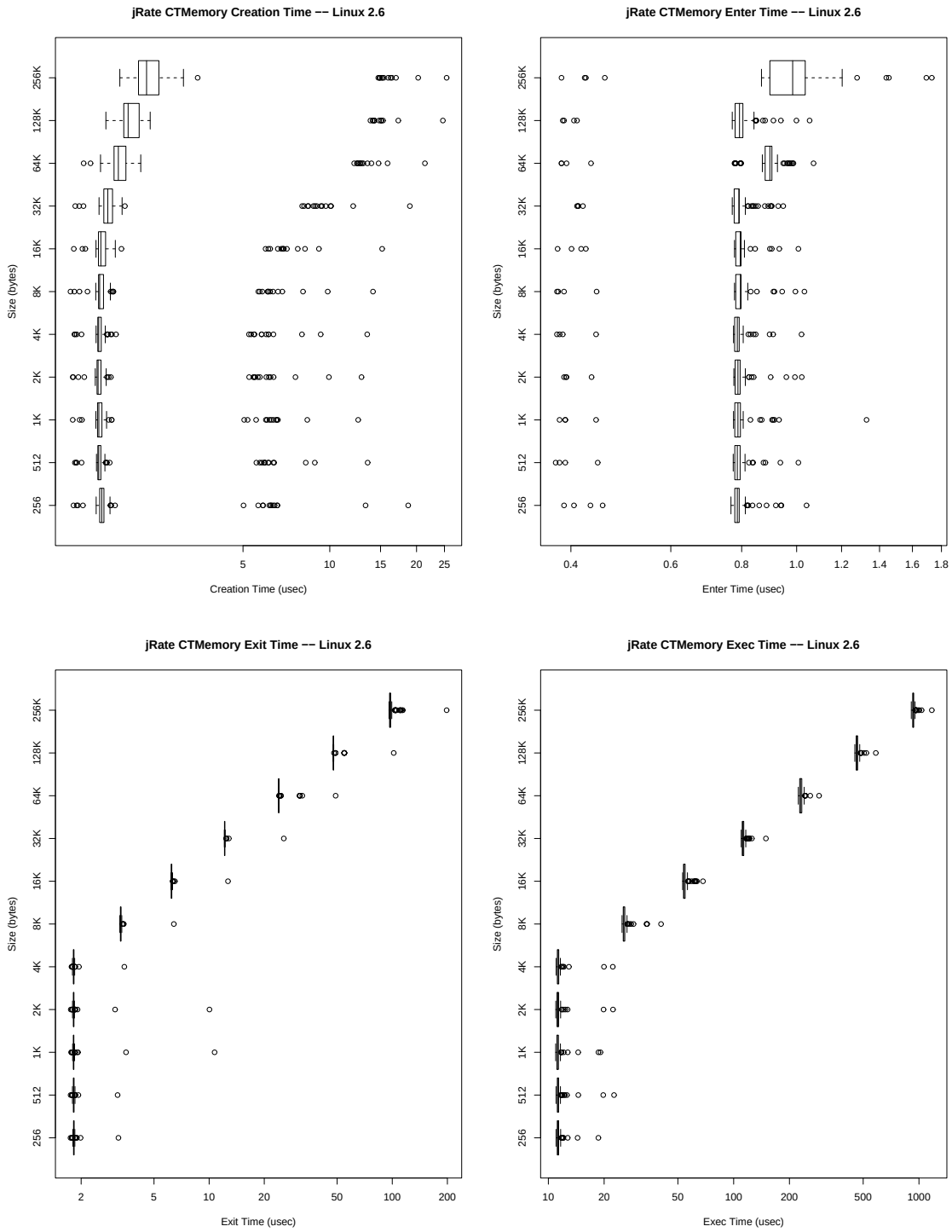


Figure 6.24: jRate CTMemory Timings, Linux 2.6.

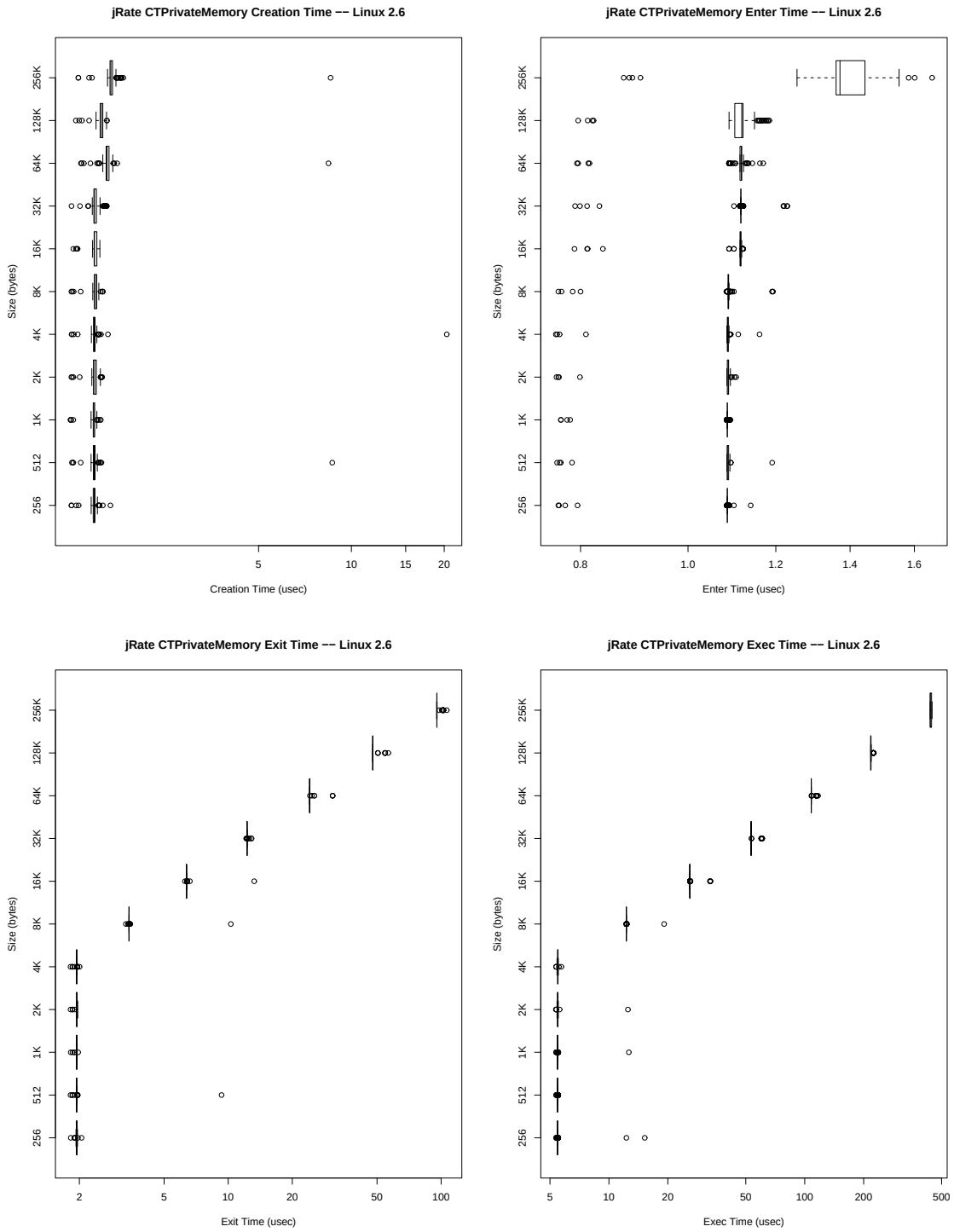


Figure 6.25: jRate CTPrivateMemory Timings, Linux 2.6.

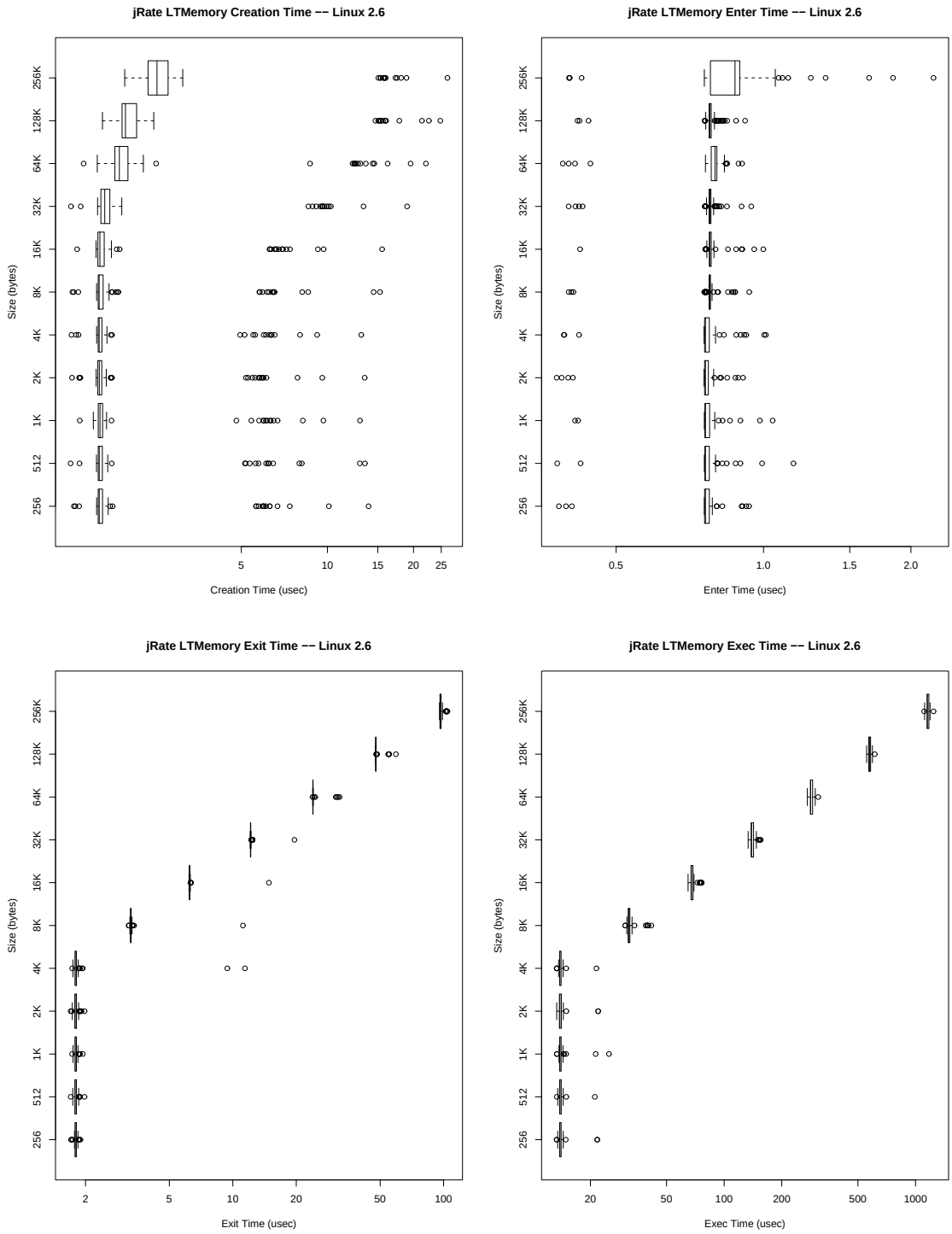


Figure 6.26: jRate LTMemory Timings, Linux 2.6.

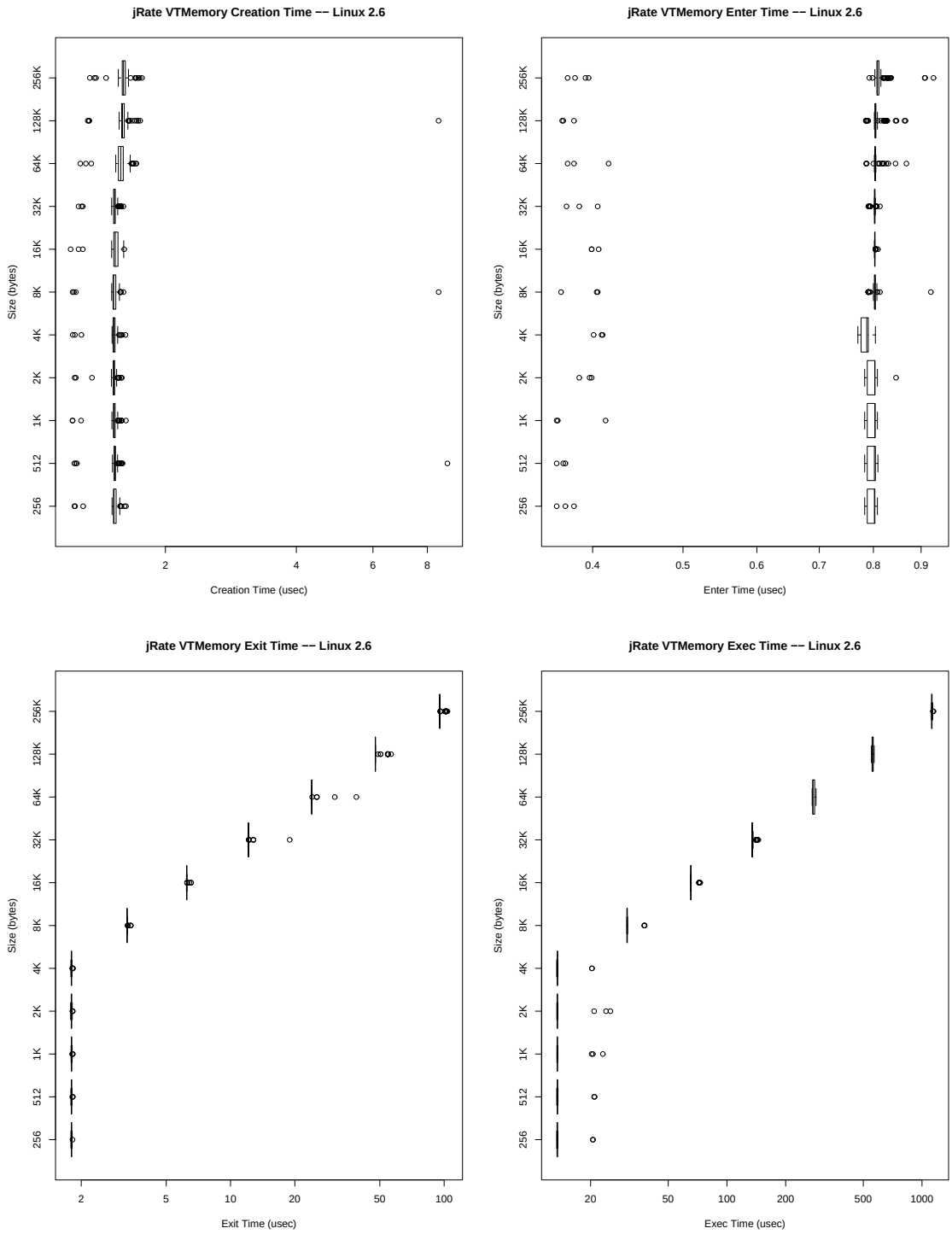


Figure 6.27: jRate VTMemory Timings, Linux 2.6.

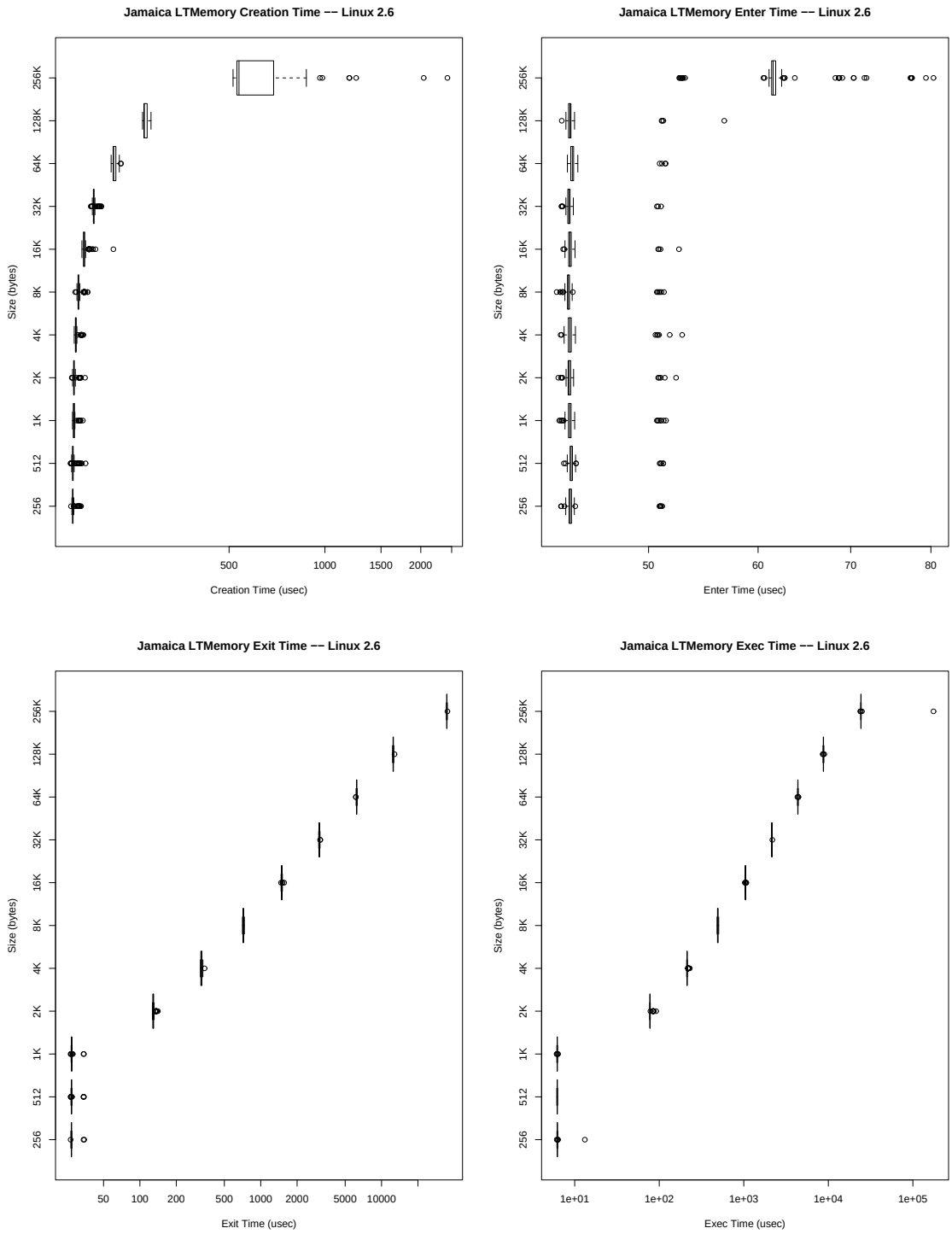


Figure 6.28: Jamaica LTMemory Timings, Linux 2.6.

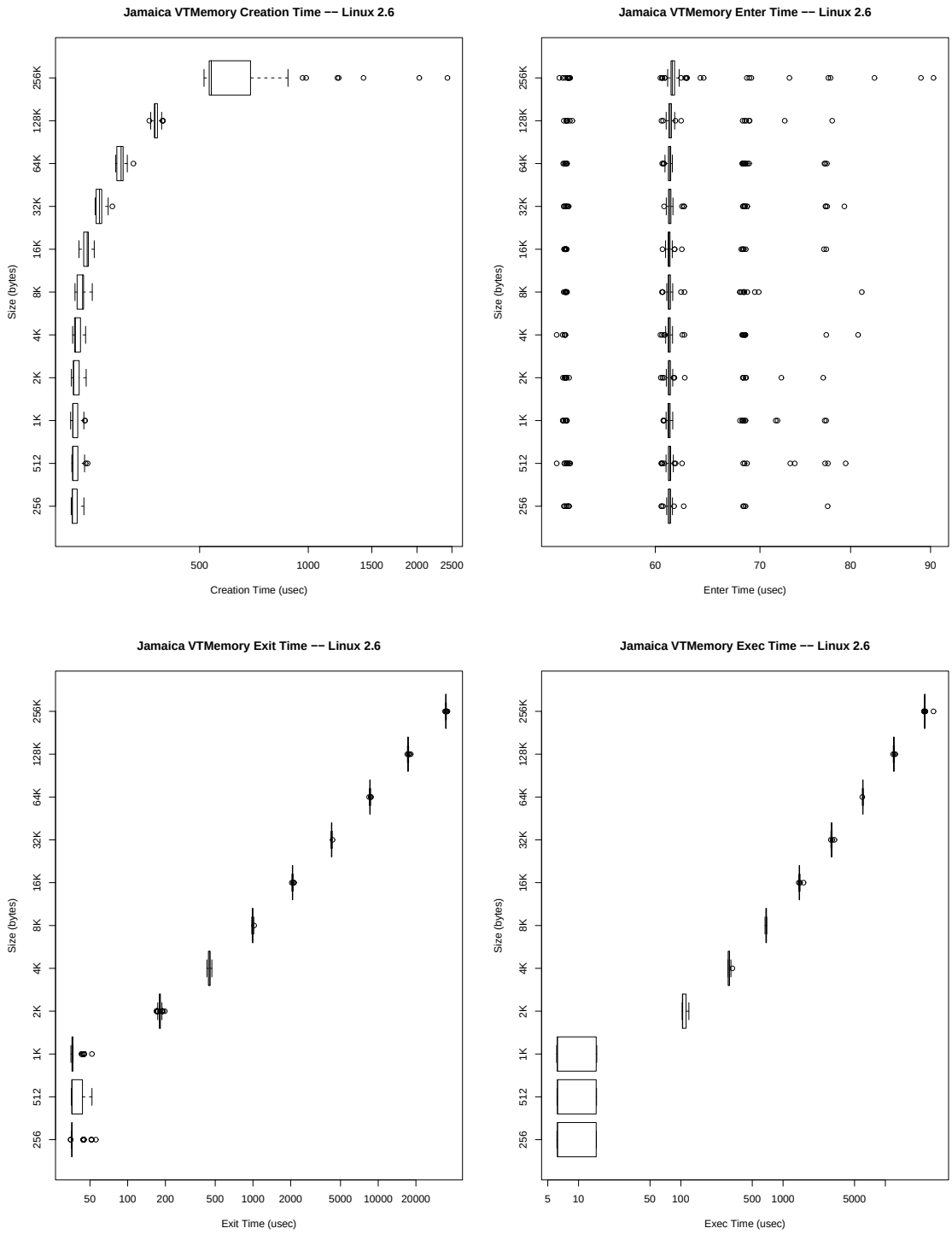


Figure 6.29: Jamaica VTMemory Timings, Linux 2.6 Allocation Time.

area framework, and the use of $O(1)$ segregated allocator as buffer provider for memory areas. Moreover, as it can be seen from the box plots reported in Figure 6.16-6.29, jRate's creation time is much smaller than the one provided by Jamaica and JTime, and the performance gain grows linearly with the size of the memory being allocated.

If we consider now the enter time, we can see that it is practically independent of the memory size for both jRate and Jamaica, while JTime shows some minor dependency on the memory size.

Finally, if we consider now the exit and execution time, we can see that this is constant for jRate for memory areas smaller or equal to 4 Kbytes, while it is constant for Jamaica and JTime for memory areas smaller or equal to 1 Kbyte. The reason for this is that jRate rounds the size of a memory are to that of an OS page, which on Linux is 4 Kbytes. Apparently JTime and Jamaica round the size of a memory area so to be multiple of 1 Kbyte. Finally it should be noticed that the execution time closely resembles the exit time, since in the test, the exit time dominates the computation.

It is worth noticing that based on the results shown in Figure 6.16, 6.17, 6.18 and 6.19[7], `CTMemory` and `CTPrivateMemory` provide a clear gain in performance over `LTMemory` and `VTMemory`. Thus, we believe that this kind of memory should be added to the RTSJ specification.

- **Dispersion Measures**— Figure 6.16- 6.29 show how all the RTSJ implementations have very low dispersion values for small memory sizes. These dispersion values grow with the size of the scoped memory for both implementations. JTime's predictability seems to be the most susceptible to memory size.

- **Worst-case Measures**—The results for all the measured variables show that the worst-case measures are very close to the average-case for jRate and mostly for Jamaica. In contrast, the JTime's worst-case values can be quite large compared to its average-case values. The largest difference between average- and worst-case measures appeared in the creation time and in the execution time.

  We cannot give a precise answer to the reason of this behavior since the code of the JTime was not available for inspection. A reasonable guess, however, is that the JTime allocators rely directly on the system provided `malloc()` for each of the allocated objects. This explanation justifies both the relatively small creation time,

[7]The same applies for Linux 2.6, see , Figure 6.24, 6.25, 6.26 and 6.27

and also the degradation of the predictability when the allocator creates many objects to fill the scoped memory.

6.4.3 Asynchrony Benchmark Results

Below we present and analyze the results of the asynchronous event handler dispatch delay and asynchronous event handler priority inversion tests, which were described in Section 6.2.1.

Asynchronous Event Handler Dispatch Delay Test. This test measures the dispatch latency of the two types of asynchronous event handlers defined in the RTSJ. The results we obtained are presented and analyzed below[8].

Test Settings. To measure the dispatch latency provided by different types of asynchronous event handlers defined by the RTSJ, we ran the test described in Section 6.2.1 with a fire count of 5,000 for both Jamaica and jRate. To ensure that each event firing causes a complete execution cycle, we ran the test in "lockstep mode," where one thread fires an event and only after the thread that handles the event is done is the event fired again. To avoid the interference of the Garbage Collector (GC) while performing the test, the real-time thread that fires and handles the event uses scoped memory as its current memory area.

Test Results. Figures 6.30, and 6.31, and Tables 6.13, 6.14, 6.15, 6.16, shows the results for the dispatch latency for successive event firings.

Results Analysis. Below we analyze the results of the tests that measure the average-case and worst-case dispatch latency, as well as its dispersion, for the different test settings:

- **Average Measures**—From Tables 6.13, 6.14, 6.15, 6.16 it is possible to see that while both jRate and Jamaica have good predictability, the average dispatch latency measured for jRate is 10 times smaller than that measured for Jamaica. This is likely to be due to the fact that Jamaica uses user level threads. Finally it should also be noticed that for both RTSJ implementations there is only a small difference in term of

[8]This test could not be executed with JTime since it would systematically dump a core.

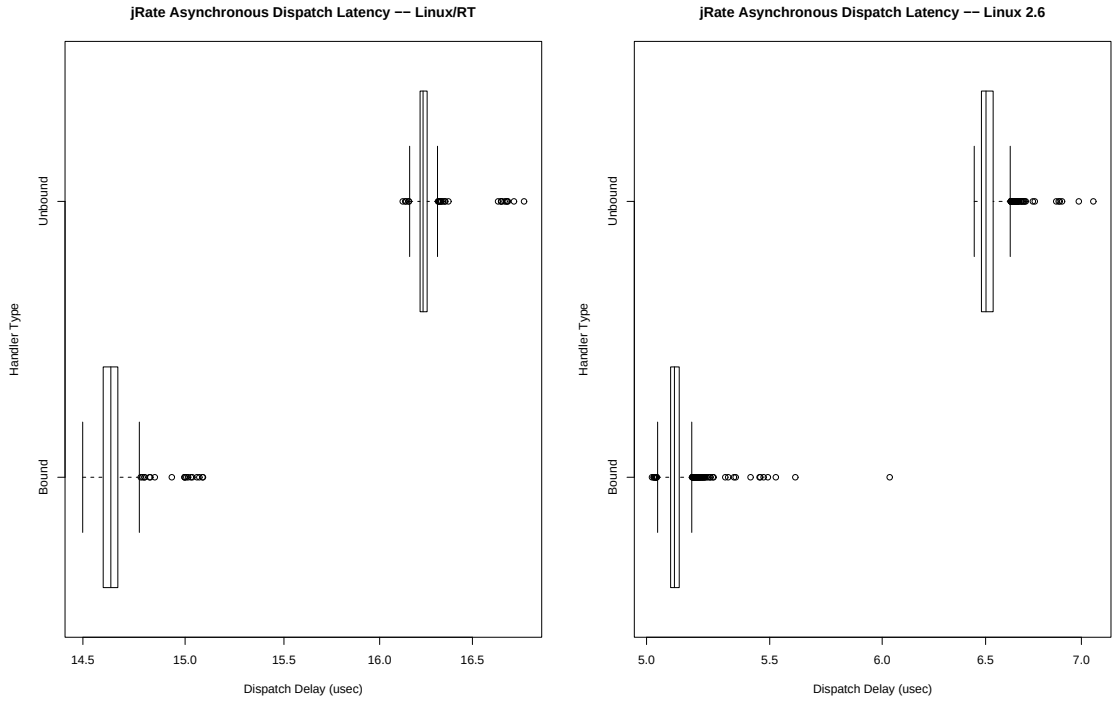


Figure 6.30: jRate Dispatch Delay.

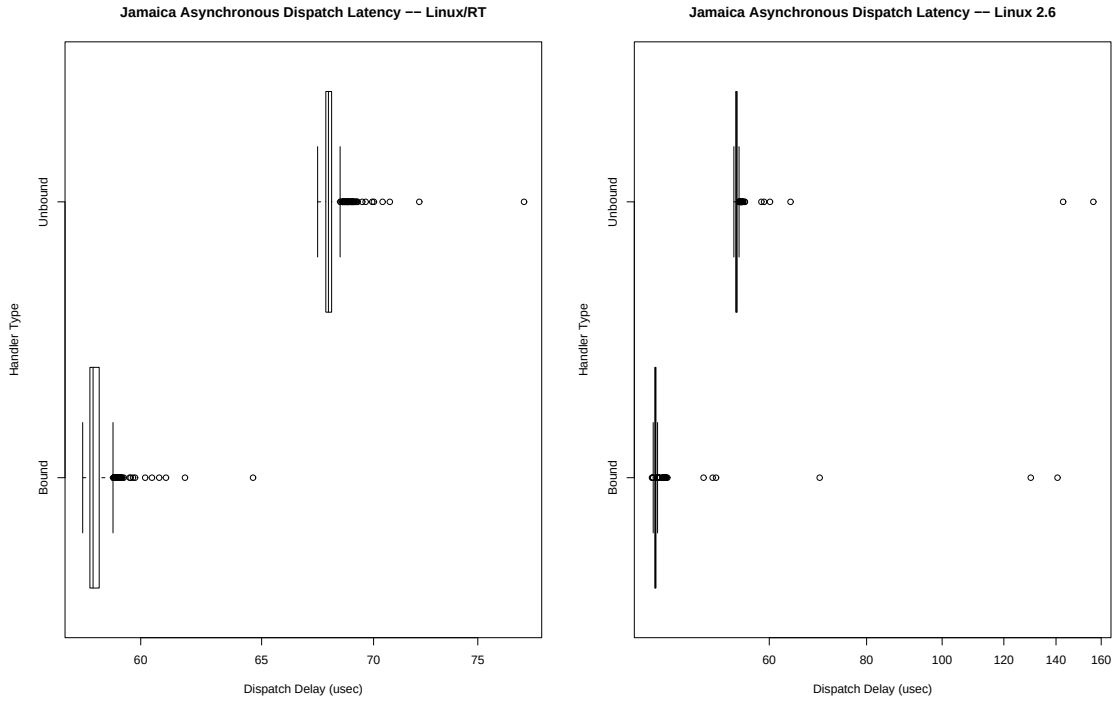


Figure 6.31: Jamaica Dispatch Delay Latency

Table 6.13: jRate Async. Event Handler Dispatch Latency, Linux/RT

jRate Bound Async. Handler Dispatch Latency – Linux/RT

Basic Statistics	
Mean	14.638 μs
STD	0.068 μs
Mode	14.647 μs
Median	14.635 μs
COV	0.005
Skew	2.35

Quantile	
0.10	14.567 μs
0.25	14.598 μs
0.75	14.669 μs
0.90	14.701 μs
0.99	14.998 μs
0.999	15.089 μs

Intervals	
IQR	0.071 μs
NTR	0.134 μs
Interval	[14.499, 15.089] μs
90% Conf. Int.	[14.635, 14.641] μs
99% Conf. Int.	[14.633, 14.643] μs
99.9% Conf. Int.	[14.631, 14.645] μs
Top Outlier	[14.781, 15.089] μs
Non Outlier	[14.491, 14.776] μs
Top Extremes	[14.934, 15.089] μs
Non Extremes	[14.385, 14.882] μs

jRate Async. Handler Dispatch Latency – Linux/RT

Basic Statistics	
Mean	16.238 μs
STD	0.054 μs
Mode	16.217 μs
Median	16.232 μs
COV	0.003
Skew	5.949

Quantile	
0.10	16.202 μs
0.25	16.216 μs
0.75	16.254 μs
0.90	16.27 μs
0.99	16.641 μs
0.999	16.785 μs

Intervals	
IQR	0.038 μs
NTR	0.068 μs
Interval	[16.123, 16.785] μs
90% Conf. Int.	[16.236, 16.241] μs
99% Conf. Int.	[16.234, 16.242] μs
99.9% Conf. Int.	[16.233, 16.244] μs
Top Outlier	[16.313, 16.785] μs
Non Outlier	[16.159, 16.311] μs
Top Extremes	[16.369, 16.785] μs
Non Extremes	[16.102, 16.368] μs

Table 6.14: jRate Async. Event Handler Dispatch Latency, Linux 2.6.

jRate Bound Async. Handler Dispatch Latency – Linux 2.6

Basic Statistics	
Mean	5.118 μs
STD	0.057 μs
Mode	5.098 μs
Median	5.109 μs
COV	0.011
Skew	6.519

Quantile	
0.10	5.075 μs
0.25	5.094 μs
0.75	5.128 μs
0.90	5.163 μs
0.99	5.35 μs
0.999	6.035 μs

Intervals	
IQR	0.034 μs
NTR	0.088 μs
Interval	[5.021, 6.035] μs
90% Conf. Int.	[5.115, 5.12] μs
99% Conf. Int.	[5.114, 5.122] μs
99.9% Conf. Int.	[5.112, 5.123] μs
Top Outlier	[5.18, 6.035] μs
Non Outlier	[5.043, 5.179] μs
Top Extremes	[5.231, 6.035] μs
Non Extremes	[4.992, 5.23] μs

jRate Async. Handler Dispatch Latency – Linux 2.6

Basic Statistics	
Mean	6.518 μs
STD	0.059 μs
Mode	6.477 μs
Median	6.502 μs
COV	0.009
Skew	2.917

Quantile	
0.10	6.466 μs
0.25	6.479 μs
0.75	6.539 μs
0.90	6.587 μs
0.99	6.699 μs
0.999	7.065 μs

Intervals	
IQR	0.06 μs
NTR	0.121 μs
Interval	[6.443, 7.065] μs
90% Conf. Int.	[6.515, 6.52] μs
99% Conf. Int.	[6.513, 6.522] μs
99.9% Conf. Int.	[6.512, 6.523] μs
Top Outlier	[6.629, 7.065] μs
Non Outlier	[6.389, 6.629] μs
Top Extremes	[6.742, 7.065] μs
Non Extremes	[6.299, 6.719] μs

Table 6.15: Jamaica Async. Event Handler Dispatch Latency, Linux/RT

Jamaica Bound Async. Handler Dispatch Latency – Linux/RT

Basic Statistics	
Mean	58.248 μs
STD	0.421 μs
Mode	58.048 μs
Median	58.137 μs
COV	0.007
Skew	5.312

Quantile	
0.10	57.919 μs
0.25	58.014 μs
0.75	58.373 μs
0.90	58.72 μs
0.99	59.562 μs
0.999	64.631 μs

Intervals	
IQR	0.359 μs
NTR	0.801 μs
Interval	[57.738, 64.631] μs
90% Conf. Int.	[58.231, 58.266] μs
99% Conf. Int.	[58.217, 58.279] μs
99.9% Conf. Int.	[58.207, 58.29] μs
Top Outlier	[58.925, 64.631] μs
Non Outlier	[57.475, 58.911] μs
Top Extremes	[59.562, 64.631] μs
Non Extremes	[56.937, 59.45] μs

Jamaica Async. Handler Dispatch Latency – Linux/RT

Basic Statistics	
Mean	68.01 μs
STD	0.447 μs
Mode	67.874 μs
Median	67.938 μs
COV	0.007
Skew	10.81

Quantile	
0.10	67.728 μs
0.25	67.821 μs
0.75	68.082 μs
0.90	68.282 μs
0.99	69.247 μs
0.999	77.332 μs

Intervals	
IQR	0.261 μs
NTR	0.554 μs
Interval	[67.454, 77.332] μs
90% Conf. Int.	[67.992, 68.028] μs
99% Conf. Int.	[67.977, 68.043] μs
99.9% Conf. Int.	[67.967, 68.054] μs
Top Outlier	[68.478, 77.332] μs
Non Outlier	[67.429, 68.474] μs
Top Extremes	[68.891, 77.332] μs
Non Extremes	[67.038, 68.865] μs

Table 6.16: Jamaica Async. Event Handler Dispatch Latency, Linux 2.6

Jamaica Bound Async. Handler Dispatch Latency – Linux 2.6

Basic Statistics	
Mean	43.118 μs
STD	4.245 μs
Mode	42.835 μs
Median	42.861 μs
COV	0.098
Skew	20.922

Quantile	
0.10	42.724 μs
0.25	42.789 μs
0.75	42.929 μs
0.90	43.005 μs
0.99	44.204 μs
0.999	140.615 μs

Intervals	
IQR	0.14 μs
NTR	0.281 μs
Interval	[42.427, 140.615] μs
90% Conf. Int.	[42.946, 43.29] μs
99% Conf. Int.	[42.806, 43.43] μs
99.9% Conf. Int.	[42.703, 43.533] μs
Top Outlier	[43.141, 140.615] μs
Non Outlier	[42.579, 43.139] μs
Top Extremes	[43.351, 140.615] μs
Non Extremes	[42.369, 43.349] μs

Jamaica Async. Handler Dispatch Latency – Linux 2.6

Basic Statistics	
Mean	54.685 μs
STD	4.287 μs
Mode	54.391 μs
Median	54.44 μs
COV	0.078
Skew	22.074

Quantile	
0.10	54.268 μs
0.25	54.352 μs
0.75	54.565 μs
0.90	54.692 μs
0.99	55.501 μs
0.999	156.315 μs

Intervals	
IQR	0.213 μs
NTR	0.424 μs
Interval	[54.045, 156.315] μs
90% Conf. Int.	[54.512, 54.859] μs
99% Conf. Int.	[54.37, 55.001] μs
99.9% Conf. Int.	[54.266, 55.104] μs
Top Outlier	[54.885, 156.315] μs
Non Outlier	[54.032, 54.885] μs
Top Extremes	[55.214, 156.315] μs
Non Extremes	[53.713, 55.204] μs

performances between the `AsyncEventHandler` and the `BoundAsyncEvent-Handler`. For jRate the small difference is due to the fact that `AsyncEvent-Handler` are implemented using a thread pool, and it is likely that Jamaica does the same thing. Finally it is worth noticing how performances on Linux 2.6 are 3 times better for jRate, while they keep unvaried for Jamaica.

- **Dispersion Measures**—From Tables 6.13, 6.14, 6.15, 6.16 it is easy to see that both jRate and Jamaica have tight dispatch delays, with jRate showing an impressing predictability. This is mostly due to the way in which dispatching is implemented in jRate. To avoid this locking overhead, jRate uses a data structure that associates the event handler list with a given event and allows the contents of the data structure to be read without acquiring/releasing a lock. Only modifications to the data structure must be serialized. As a result, jRate's `AsyncEventHandler` dispatch latency is relatively predictable, even though the handler has no thread bound to it permanently. The jRate thread pool implementation uses LIFO queues for its executor, *i.e.*, the last executor that has completed executing is the first one reused. This technique is often applied in thread pool implementations to leverage cache affinity benefits [41].

  It is worth observing the the dispersion of data values are the same on Linux/RT and Linux 2.6 for jRate, while Jamaica shows more dispersed data points on Linux 2.6. As a final remark, it is worth noticing that the dispatch delay predictability is limited by the context switch predictability, thus it is worth comparing the results found here with the results shown in Section 6.4.1.

- **Worst-case Measures**—From the results shown in Tables 6.13, 6.14, 6.15, 6.16, it is possible to see how jRate has a very good worst case behaviour on both Linux/RT and Linux 2.6, thus showing a very appropriate behaviour for real-time systems. On the other hand, Jamaica suffers of a very bad worst case behaviour. The maximum dispatch delay is as much as three times bigger than the minimum. Again, this problem might stem from the fact that Jamaica uses user level threads.

Asynchronous Event Handler Priority Inversion Test. This test measures how the dispatch latency of an asynchronous event handler H is influenced by the presence of N others event handlers, characterized by a lower execution eligibility than H. In the ideal case, H's dispatch latency should be independent of N, and any delay introduced by the presence of other handlers represents some degree of priority inversion. The results we obtained are presented and analyzed below.

Test Settings. This test uses the same settings as the asynchronous event handler dispatch delay test. However, since the performances of `AsyncEventHandler` and `BoundAsyncEventHandler` are similar for the tested platforms, in this test, only the `BoundAsyncEventHandler` performance is measured. The current test uses the following two types of asynchronous event handlers:

- The first is identical to the one used in the previous test, *i.e.*, it gets a time stamp after the handler is called and measures the dispatch latency. This logic is associated with H.
- The second does nothing and is used for the lower priority handlers.

Test Results. Figure 6.32, and Figure 6.33, show the results found for this test.

Results Analysis. Below, we analyze the results of the tests that measure average-case and worst-case dispatch latency, as well as its dispersion, for jRate and the Jamaica.

- **Average Measures**—Figure 6.32, and Figure 6.33 illustrate that the average dispatch latency experienced by H is essentially constant for both jRate and Jamaica, regardless of the number of low-priority handlers.
- **Dispersion Measures**—As the results found for the previous test, Figure 6.32, and Figure 6.33 show that Jamaica has more dispersed dispatch delay than jRate, especially on Linux 2.6. It is worth noticing how the outliers shown in Figure 6.32 and Figure 6.33 are contributed by the unpredictability of the context switch (see Section 6.4.1) and the locking used to implements the thread pools.
- **Worst-Case Measures**—As the results found for the previous test, from Figure 6.32, and Figure 6.33 it is possible to see how jRate has a very good worst case behaviour on both Linux/RT and Linux 2.6, thus showing a very appropriate behaviour for real-time systems. On the other hand, Jamaica suffers of a not so good worst case behaviour. The maximum dispatch delay is as much as three times bigger than the minimum (on Linux 2.6). Again, this problem might stem from the fact that Jamaica uses user level threads.

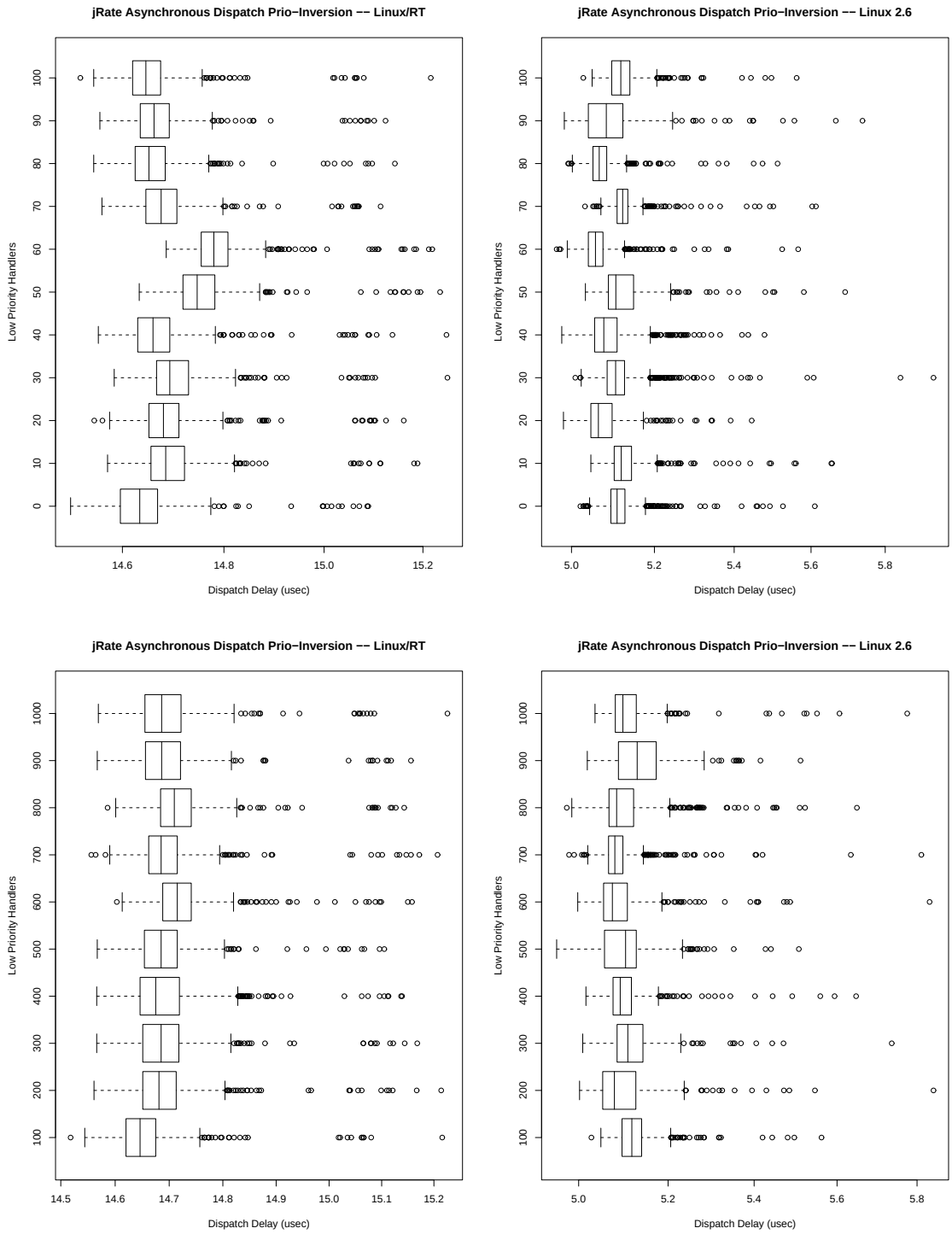


Figure 6.32: jRate Dispatch Delay.

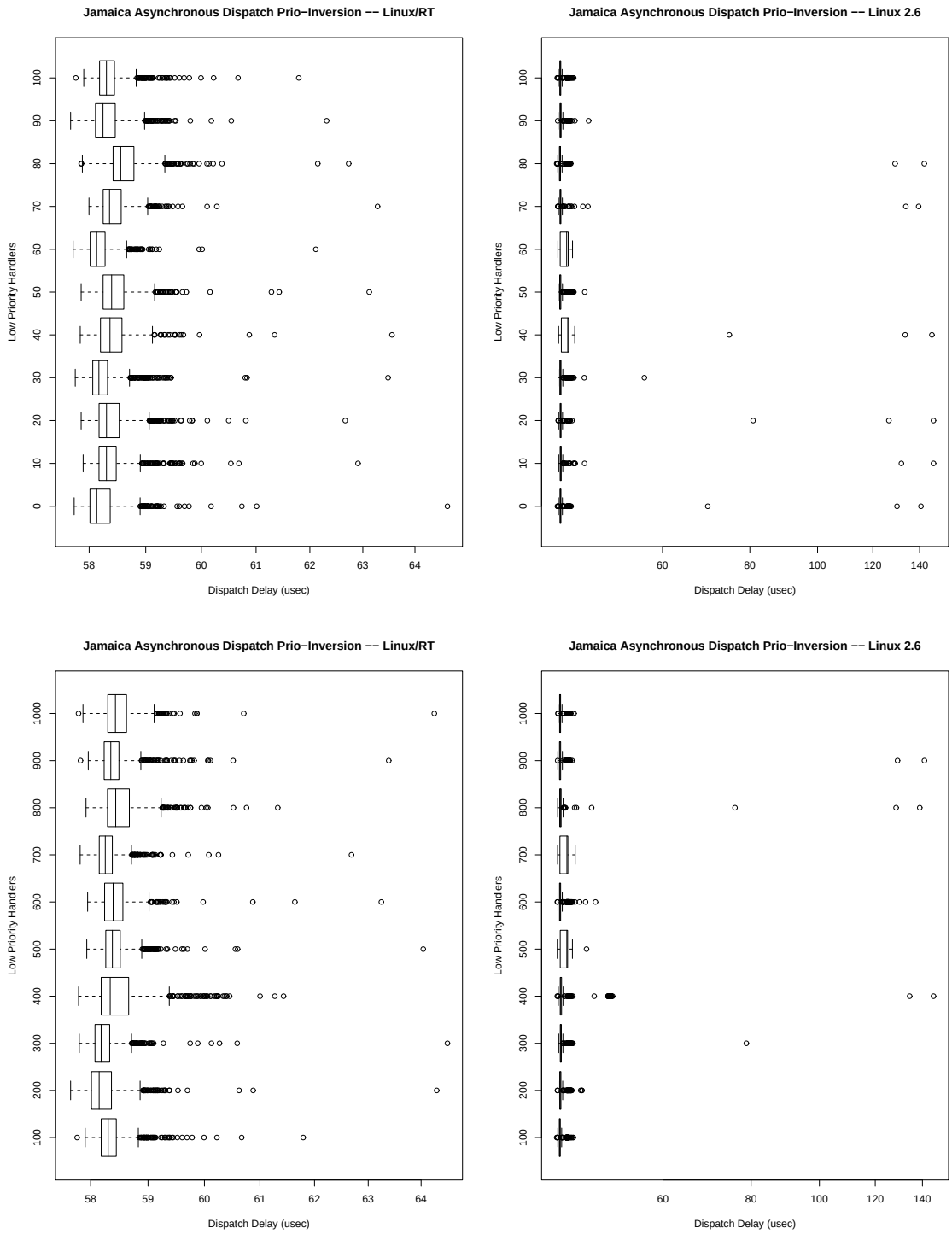


Figure 6.33: Jamaica Dispatch Delay Latency.

Chapter 7

Concluding Remarks

This thesis has focused on providing a set of guidelines for designing and optimizing safe and efficient real-time middleware. Specifically, in the context of the RTSJ this thesis has provided a road map for both RTSJ users and implementors on how to effectively architect RTSJ application and middleware. This thesis has systematically identified RTSJ's lacks, and the presence of features that could undermine its usage in real-time systems. It has then proposed solutions, which have been validated by means of empirical evaluation. The key contribution of this thesis can be summarized as follows.

- In this thesis we have provided a classification of RTSJ application along with a catalog of patterns that should help in developing more robust and efficient RTSJ applications.
- We have also shown how generative programming techniques can be used to design effective real-time middleware.
- We have provided a set of optimization techniques and optimal algorithms that can be used to improve the predictability and performance of RTSJ and similar middleware.
- This thesis has proposed a series of extension to the RTSJ. These extensions, have been validated empirically, when appropriate, and their effectiveness has been discussed.
- This thesis has provided a throughout empirical evaluation of three RTSJ implementations.

- Last but not the least, the work on which this thesis is based, has produced two successful open source projects: jRate and RTJPerf. The first one is an ahead of time compiled RTSJ platform, while the second is an RTSJ benchmarking suite.

References

[1] AICAS. Jamaica. `http://www.aicas.com/`, 2004.

[2] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, Boston, 2001.

[3] Angelo Corsaro and Douglas C. Schmidt. The Design and Performance of Real-time Java Middleware. *IEEE Transactions on Parallel and Distributed Systems*, 2003.

[4] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Boston, 2000.

[5] William S. Beebee and Martin Rinard. An implementation of scoped memory for real-time Java. *Lecture Notes in Computer Science*, 2211, 2001.

[6] Ali Behforooz and Frederick J. Hudson. *Software Engineering Fundamentals*. Oxford University Press, 198 Madison Avenue, New York, 1996.

[7] Greg Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Daniel Dvorak, Brian Giovannoni, Kenny Meyer, Alex Murray, and Kirk Reinholtz. Programming with Non-Heap Memory in RTSJ. In *Proceedings of the 18th 2003 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'03)*. ACM Press, 2003.

[8] Greg Bollella, James Gosling, Ben Brosgol, Peter Dibble, Steve Furr, David Hardin, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[9] Intel Coorporation. Using the RDTSC Instruction for Performance Monitoring. Technical report, Intel Coorporation, 1997.

[10] Angelo Corsaro. jRate. `http://www.cs.wustl.edu/~corsaro/jRate/`, 2004.

[11] Angelo Corsaro. Python Statistics. `http://www.cs.wustl.edu/~corsaro/jRate/`, 2004.

[12] Angelo Corsaro. RTJPerf. `http://www.cs.wustl.edu/~corsaro/jRate/`, 2004.

[13] Angelo Corsaro and Ron K. Cytron. Efficient Memory-reference Checks for Real-Time Java. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, Compiler, and Tools for Embedded Systems*, pages 51–58. ACM Press, 2003.

[14] Angelo Corsaro and Ron K. Cytron. Efficient memory-reference checks for real-time java. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems*, 2003.

[15] Angelo Corsaro and Ron K. Cytron. Implementing and optimizing real-time java. In *Proceedings of The 11th International Workshop on Parallel and Distributed Real-Time Systems*. IEEE, 2003.

[16] Angelo Corsaro and Corrado Santoro. A C++ Native Interface for Intrepreted JVMs. In *International Workshop on Java Technologies for Real-time and Embedded Systems*. Springer-Verlag, November 2003.

[17] Angelo Corsaro and Douglas C. Schmidt. Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems. In *Proceedings of the* 8th *IEEE Real-Time Technology and Applications Symposium*, San Jose, September 2002. IEEE.

[18] Angelo Corsaro and Douglas C. Schmidt. Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems. Technical Report 2002-001, University of Califoria, Irvine, 2002.

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

[20] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.

[21] Miguel A. de Miguel. QoS-Aware Component Frameworks. In *The* 10th *International Workshop on Quality of Service (IWQoS 2002)*, Miami Beach, Florida, May 2002.

[22] Steven M. Donahue, Matthew P. Hampton, Morgan Deters, Jonathan M. Nye, Ron K. Cytron, and Krishna M. Kavi. Storage allocation for real-time, embedded systems. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software: Proceedings of the First International Workshop*, pages 131–147. Springer Verlag, 2001.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

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

[25] GNU is Not Unix. Octave 2.1.57 . `http://www.octave.org/`, 2004.

[26] James Gosling, Bill Joy, and Guy Steele. *The Java Programming Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[27] Grace Development Team. xmgrace 5.1.14. `http://plasma-gate.weizmann.ac.il/Grace/`, 2004.

[28] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, October 1997. ACM.

[29] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley & Sons, New York, 1991.

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

[31] Michael B. Jones and Richard F. Rashid. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. Technical Report CMU-CS-87-150, Carnegie Mellon University, September 1986.

[32] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *Java Language Specification (2nd Edition)*. Addison–Wesley, 2000.

[33] 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.

[34] Mark Lutz. *Programming Python*. O'Reilly, 2nd edition, 2001.

[35] Miguel A. de Miguel-Cabello M. Teresa Higuera-Toledano. Dynamic Detection of Access Errors and Illegal References in RTSJ. In *Proceedings of the* 8th *IEEE Real-Time Technology and Applications Symposium*, San Jose, September 2002. IEEE.

[36] M.Rinard et al. FLEX Compiler Infrastructure. `http://www.flex-compiler.lcs.mit.edu/Harpoon/`, 2002.

[37] Norman H. Cohen. Type-Extension Type Tests Can Be Performend In Constant Time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1991.

[38] OVM/Consortium. OVM An Open RTSJ Compliant JVM. `http://www.ovmj.org/`, 2002.

[39] Geoffrey Phipps. Comparing observed bug and productivity rates for java and c++. *Software Practice & Experience*, 29(4):345–358, 1999.

[40] J. Regehr. Inferring scheduling behavior with hourglass, 2002.

[41] James D. Salehi, James F. Kurose, and Don Towsley. The Effectiveness of Affinity-Based Scheduling in Multiprocessor Networking. In *IEEE INFOCOM*, San Francisco, USA, March 1996. IEEE Computer Society Press.

[42] 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.

[43] David C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. In *Proceedings of the 10th Annual Software Technology Conference*, April 1998.

[44] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–112, April 1997.

[45] A. S. Tanenbaum. *Modern Operating Systems. Second Edition.* McGraw Hill, 1998.

[46] The R Project for Statistical Computing. . `http://www.r-project.org/`, 2003.

[47] TimeSys. JTime. `http://www.timesys.com/`, 2004.

[48] Krzysztof Palacz Jan Vitek. Java subtype tests in real-time, 2003.

[49] Wolfram Research. Math World . http://mathworld.wolfram.com/, 2004.

Vita

Angelo Corsaro

Date of Birth	July 18, 1974
Place of Birth	Bergamo, Italy
Degrees	Doctor of Science in Computer Science, Washington University, Dec 2004
Degrees	Master in Computer Science, Washington University, Aug 2001
Degrees	Laurea Magna Cum Laude, Computer Engineering, Jul 1999
Professional Societies	Association for Computing Machinery
Book Chapters	Arvind S. Krishna, Douglas C. Schmidt, Ray Klefstad, and Angelo Corsaro (2003). "Real-time CORBA Middleware" *Middleware for Communications, edited by Qusay Mahmoud, Wiley and Sons, New York.*
	Michael Kircher, Prashant Jain, Angelo Corsaro, David Levine (2002). "Distributed eXtreme Programming" *Extreme Programming Perspectives, edited by Michele Marchesi, Addison Wesley.*
Journal Publications	Angelo Corsaro and Douglas C. Schmidt (2003). "The Design and Performance of Real-time Java Middleware", *IEEE Transactions on Parallel and Distributed Systems.*
	S. Cavalieri, A. Corsaro, S. Monforte, G. Scapellato (2002). "Multicycle Polling Scheduling Algorithms for Fieldbus Networks", *IEEE Journal of Realtime Systems.*

Refereed Conference Publications

Angelo Corsaro and Ron K. Cytron, (2003). "Efficient Memory-reference Checks for Real-Time Java", *In Proceedings of the 2003 ACM SIGPLAN conference on Language, Compiler, and Tools for Embedded Systems.*

Arvind S. Krishna, Douglas C. Schmidt, Raymond Klefstad, and Angelo Corsaro (2003). "Towards Predictable Real-time Java Object Request Brokers", *Proceedings of the 9th Real-time Embedded Technology and Applications Symposium (RTAS).*

Angelo Corsaro and Douglas C. Schmidt (2002). "Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems", *Proceedings of the* 8th *IEEE Real-Time Technology and Applications Symposium.*

Angelo Corsaro and Douglas C. Schmidt (2002). "The Design and Performance of the jRate Real-Time Java Implementation", *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE, Berlin, 2002. Lecture Notes in Computer Science 2519, Springer Verlag.*

Angelo Corsaro, Douglas C. Schmidt, Raymond Klefstad, and Carlos O'Ryan (2002). "Virtual Component: a Design Pattern for Memory-Constrained Embedded Applications", *Proceedings of the* 9th *Annual Conference on the Pattern Languages of Programs.*

M. Kiercher, P. Jain, A. Corsaro (2002). "XP + AOP = Better Software?", *Proceeding of the 3nd International Conference on Extreme Programming and Flexible Processes in Software Engineering.*

A. Corsaro, D. Schmidt, C. Gill, R. Cytron (2001). "Formalizing Meta Programming Techniques to Reconcile Heterogeneous Scheduling Disciplines in Open Distributed Real-Time Systems", *Proceeding of the 3rd International Symposium on DistributedObjects and Applications.*

M. Kiercher, P. Jain, A. Corsaro, D. Levine (2001). Distributed eXtreme Programming, *Proceeding of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering.*

S. Cavalieri, A. Corsaro, O. Mirabella, G. Scapellato (1998). Scheduling Periodic Information Flow in Fieldbus and Multi-Fieldbus Environment, *Proceeding of the International Conference on Automation.*

Refereed Workshop Publications

Angelo Corsaro and Corrado Santoro (2003). "A C++ Native Interface for Intrepreted JVMs", *International Workshop on Java Technologies for Real-time and Embedded Systems. Springer-Verlag, November 2003.*

Angelo Corsaro and Ron K. Cytron, (2003). "Implementing and Optimizing Real-Time Java", *In Proceedings of The 11th IEEE International Workshop on Parallel and Distributed Real-Time Systems.*

M. Mousavi, G. Russello, M. Chaudron, M. Reniers, T. Basten, A. Corsaro, S. Shukla, R. Gupta, D.C. Schmidt (2002) "Using Aspect-GAMMA in Design and Verification of Embedded Systems", *The IEEE International High Level Design Validation and Test Workshop.*

M. Mousavi, G. Russello, M. Chaudron, M. Reniers, T. Basten, A. Corsaro, S. Shukla, R. Gupta, D.C. Schmidt (2002) "Aspects + GAMMA = AspectGAMMA: A Formal Framework for Aspect Oriented Specification", *Early Aspects: Apect Oriented Requirements Engineering and Architecture Requirements Workshop.*

December 2004