

# The Design and Performance of Real-time Java Middleware

Angelo Corsaro and Douglas C. Schmidt  
Electrical and Computer Engineering Department  
University of California, Irvine, CA 92697  
{corsaro, schmidt}@ece.uci.edu

## Abstract

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

*This paper provides two contributions to the study of languages and middleware for real-time and embedded applications. We first describe the architecture of jRate, which is an open-source ahead-of-time-compiled implementation of the Real-time Specification for Java (RTSJ) middleware. We then show performance results obtained using RTJPerf, which is an open-source benchmarking suite that illustrates how well RTSJ middleware implementations perform.*

*This paper shows that while research remains to be done to make RTSJ a bullet-proof technology, the initial results are promising. The performance and predictability of jRate provides a baseline for what can be achieved by using ahead-of-time compilation. Likewise, RTJPerf enables researchers and practitioners to evaluate the pros and cons of RTSJ middleware systematically as implementations mature.*

**Keywords** Real-time Middleware, Real-time Java, QoS-enabled Middleware Platforms, Object-Oriented Languages, Real-time Resource Management, Performance Evaluation.

## 1 Introduction

### 1.1 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 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, companies are now also faced with the problem of developing and delivering new products in short time frames. It is therefore essential that the production of real-time embedded systems can take advantage of languages, middleware, tools, and methods that enable higher software productivity, without unduly degrading quality of service (QoS).

### 1.2 The State of the Art

Many real-time embedded systems are still developed in C, and increasingly in C++. While writing in C/C++ is more productive than assembly code, they are not the most productive or error-free 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 [1]. 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. Java [2] has become an attractive choice for the fol-

lowing reasons:

- It has a large and rapidly growing programmer base and is taught in many universities.
- It is simpler than C++, yet programmers experienced in C++ can learn it easily.
- It has an architecture—the Java Virtual Machine (JVM)—that allows Java applications to run on any platform that supports a JVM.
- It has a portable standard library that can reduce programming time and costs.
- It offloads many tedious and error-prone programming details, particularly memory management, from developers into the language runtime system.
- It has powerful language features, such as strong typing, dynamic class loading, and reflection/introspection.
- It supports concurrency and synchronization in the language.
- Its bytecode representation is more compact than native code, which can reduce memory usage for embedded systems.

Conventional Java implementations are unsuitable for developing real-time embedded systems, however, due to the following problems:

- The scheduling of Java threads is purposely underspecified to make it easy to develop JVMs for new platforms.
- The Java Garbage Collector (GC) has higher execution eligibility than any other Java thread, which means that a thread could experience unbounded preemption latency while waiting for the GC to run.
- Java provides coarse-grained control over memory allocation and access, *i.e.*, it allows applications to allocate objects on the heap, but provides no control over the type of memory in which objects are allocated.
- 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) [3], 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.

Until recently, there was no implementation of the RTSJ, which hampered the adoption of Java in real-time embedded systems. It also hampered systematic empirical analysis of

the pros and cons of the RTSJ programming model. Several implementations of RTSJ are now available, however, including the RTSJ Reference Implementation (RI) from TimeSys [4] and jRate from the University of California, Irvine (UCI).

This paper significantly extends our earlier work [5, 6] by providing:

- More extensive coverage of jRate capabilities, such as its memory regions implementation and its support for custom allocators.
- A detailed analysis and comparison of scoped memory that quantifies the tradeoffs associated with different types of scoped memory.
- Extensive new test results based on the commercial version of Linux/RT from TimeSys, which provides many features (such as higher resolution timer, support for priority inversion control via priority inheritance and priority ceiling protocols, and resource reservation) not supported by the GPL version of the TimeSys Linux/RT.

The main contribution of this paper is to describe the techniques used by jRate to implement the RTSJ middleware, as well as to empirically illustrate the performance pitfalls and tradeoffs that certain RTSJ design decisions incur.

### 1.3 Paper Organization

The remainder of the paper is organized as follows: Section 2 provides a brief overview of the RTSJ; Section 3 describes the architecture and design rationale of jRate; Section 4 provide an overview RTJPerf [5], which is the benchmark used to evaluate RTSJ compliant implementations. Section 5 analyzes the empirical results obtained by benchmarking jRate and the TimeSys RTSJ Reference Implementation (RI) using our RTJPerf benchmarking suite; Section 6 compares our work on jRate with related research; and Section 7 summarizes our work and outlines future plans for improving the next generation of RTSJ middleware for real-time embedded applications.

## 2 Overview of the Real-Time Specification for Java

The RTSJ extends the Java API and refines the semantics of certain constructs to support the development of real-time applications. The guiding principles followed by the expert group who created the RTSJ specification included [3]:

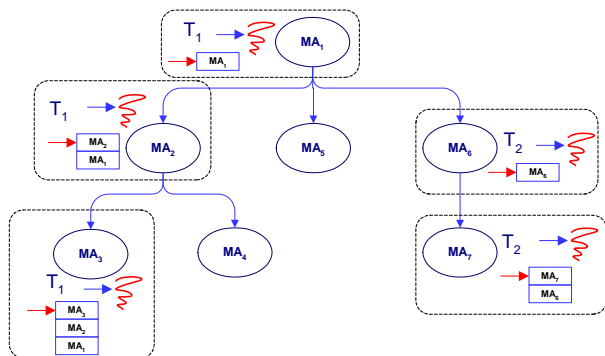
- Backward compatibility with the Java 2 platform
- No syntactic extension to the Java language, *i.e.* no new keywords
- Write once carefully, run anywhere conditionally
- Enable predictable execution and

- Balance between current practice and advanced features.

This section presents an overview of the language and middleware extensions provided by the RTSJ.

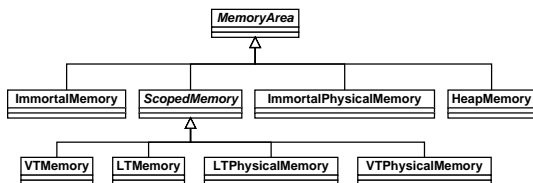
## 2.1 Memory

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. Figure 1 shows how the scope stack for threads  $T_1$  and  $T_2$  evolve while moving from one memory area to another. As shown in Figure 2, the RTSJ specifi-



**Figure 1. Thread Scope Stack in the RTSJ Memory Model**

cation provides scoped memories with linear and variable allocation times (LTMemory, LTPhysicalMemory and VTMemory, VTPhysicalMemory, respectively). For



**Figure 2. Hierarchy of Classes in the RTSJ Memory Model**

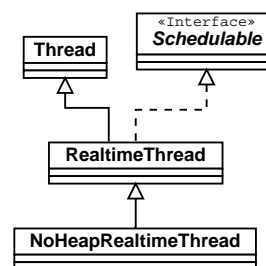
linear allocation time scoped memory, the RTSJ requires that the time needed to allocate the  $n > 0$  bytes to hold the class instance must be bounded by a polynomial function  $f(n) \leq Cn$  for some constant  $C > 0$ .<sup>1</sup> The RTSJ also

<sup>1</sup>This bound does not include the time taken by an object's constructor or a class's static initializers.

introduces the concept of *Immortal Memory*. Objects allocated within this memory area have the same lifetime of the JVM, *i.e.* are never collected. Another addition to the Java memory model provided by the RTSJ allows direct access to raw memory, as well as to allocate Java objects at specific memory locations.

## 2.2 Threads

The RTSJ extends the existing Java threading model with two new types of real-time threads: `RealtimeThread` and `NoHeapRealtimeThread`. The relation of these new classes with respect to the regular Java thread class is depicted in Figure 3.



**Figure 3. RTSJ Real-time Thread class Hierarchy**

The `NoHeapRealtimeThread` can have an execution eligibility higher than the garbage collector.<sup>2</sup> A `NoHeapRealtimeThread` can therefore neither allocate nor reference any heap objects. The scheduler controls the *execution eligibility*<sup>3</sup> of the instances of this class by using the `SchedulingParameters` associated with it.

## 2.3 Scheduling

The RTSJ introduces the concept of a `Schedulable` object. The execution of `Schedulable` entities is managed by the scheduler that holds a reference to them. The RTSJ provide a scheduling API that is sufficiently general to implement commonly used scheduling algorithms, such as Rate Monotonic (RM), Earliest Deadline First (EDF), Least Laxity First (LLF), Maximum Urgency First (MAU), etc.

<sup>2</sup>RTSJ v1.0 states that a `NoHeapRealtimeThread` always has execution eligibility higher than the GC, but this has been changed in RTSJ v1.01, where a `NoHeapRealtimeThread` can have execution eligibility higher than the GC.

<sup>3</sup>Execution eligibility is defined as the position of a schedulable entity in a total ordering established by a scheduler over the available entities. The total order depends on the scheduling policy. The only scheduler required by the RTSJ is a priority scheduler, which uses the `PriorityParameters` to determine the execution eligibility of a `Schedulable` entity, such as threads or event handlers.

However, the only required scheduler for a RTSJ-compliant implementation is a priority preemptive scheduler that can distinguish 28 different priorities.

## 2.4 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 asynchronous event handler to some application-specific or external events. As shown in Figure 4, there are two types

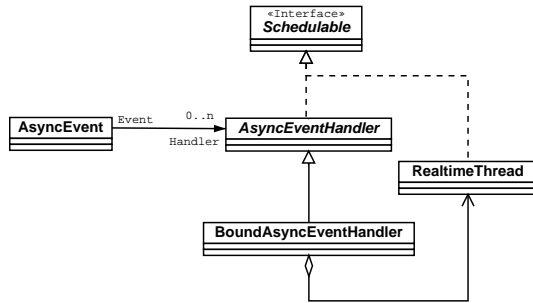


Figure 4. RTSJ Asynchronous Event Class Hierarchy

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 AsyncEventHandlers will be dispatched.
- 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.

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.

## 2.5 Time and 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. As shown in Figure 5, the RTSJ provides two types of timers:

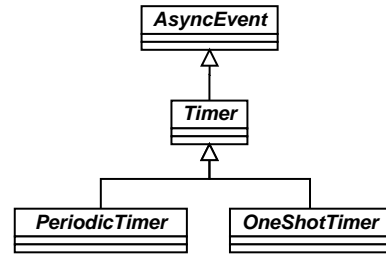


Figure 5. RTSJ Timer Class Hierarchy

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

OneShotTimers and PeriodicTimers events are handled by AsyncEventHandlers. The RTSJ also supports high resolution timers and high resolution clocks.

## 3 jRate Overview

jRate is an open-source RTSJ-based real-time Java implementation that we are developing at UCI. jRate extends the open-source GNU Compiler for Java (GCJ) runtime system [7] to provide an ahead-of-time compiled middleware platform for developing RTSJ-compliant applications. The

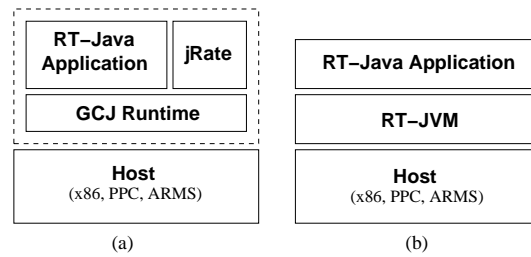


Figure 6. The jRate Architecture

jRate architecture shown in Figure 6(a) differs from the JVM model shown in Figure 6(b) since there is no JVM interpreting the Java bytecode. Instead, jRate 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.

jRate supports most of the RTSJ features described in Section 2. We describe these features below and indicate where the design and performance of these features is discussed in subsequent sections of this paper.

### 3.1 Memory Areas

jRate supports scoped memory and immortal memory. Figure 7 shows how these memory areas are implemented in jRate. This diagram shows how each memory area is

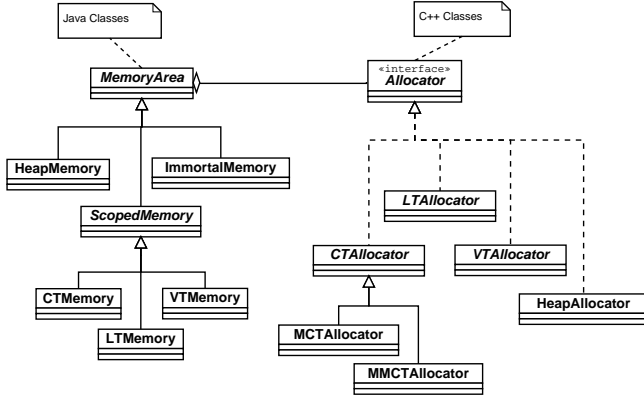


Figure 7. The jRate Memory Region Structure

associated with an allocator. There are two parallel class hierarchies: one set of Java classes for the memory areas and one set of C++ classes for the allocators. The memory management strategy is delegated to native allocators and the binding between memory area and type of allocator can be deferred until creation time. This design provides users with the flexibility to experiment with different allocators strategies and to choose the type of allocator that best fits their application usage patterns.

jRate also provides a strategy that enables users to decide which type of memory (such as linear time memory or variable time memory) should implement immortal memory. Although the RTSJ does not mandate how immortal memory is implemented, we believe it is important to allow users to specify which type of implementation to configure. jRate’s immortal memory implementation can be configured when an application is launched. Its scoped memory implementation also exposes a non-standard extension that uses non-thread safe allocators to avoid the overhead of unnecessary locks if a memory area is always accessed by a single thread.

Figure 7 illustrates a new type of scoped memory – called CTMemory – provided by jRate. CTMemory trades off allocation time for the memory area creation time. This memory area is zeroed at initialization time and the amount used is also zeroed each time the memory reference count drops to zero.<sup>4</sup> This feature provides constant time allocation for objects created within the CTMemory,

<sup>4</sup>The reference count associated with a scoped memory is represented by the number of real-time threads in it that are currently *active*, i.e., have entered the scoped memory but have not yet exited.

which is useful for real-time applications. The structure of CTMemory is depicted in Figure 8. The *type* field distin-

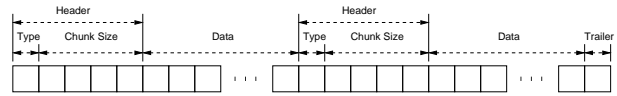


Figure 8. The jRate CTMemory Structure

guishes different types of objects. Different types of objects must be treated differently, e.g., some must be finalized, whereas others need not be finalized. The performance of jRate scoped memory implementation is analyzed in Section 5.3.1.

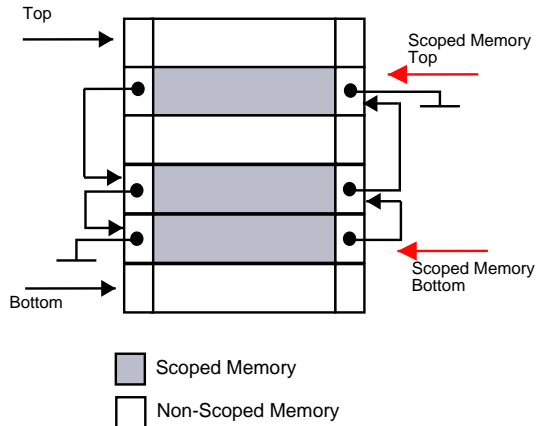
### 3.2 Real-time Threads and Scheduling

jRate supports real-time threads of type RealtimeThread using a priority preemptive scheduler based on the underlying real-time OS priority preemptive scheduler. An interesting characteristic of the RTSJ is that each RealtimeThread is associated with a scope stack that (1) keeps track of the set of memory regions that have been entered by the thread and (2) can detect cycles in the entered scopes. jRate’s scope stack implementation uses data structures which allow to perform all scope stack operations in constant time.

For example, the findFirstScope() operation defined in the RTSJ, scans the scope stack from top to bottom and returns the first scoped memory area found. If implemented as suggested by the RTSJ specification, this operation would have a time complexity of  $O(n)$ , where  $n$  is the length of the stack. jRate enhances the stack data structure suggested in the RTSJ by maintaining a doubly-linked list of the scoped memory in the stack, along with the index of the topmost scoped memory, as shown in Figure 9.

This design allows a constant time implementation of both findFirstScope(), push() and pop(). In fact, findFirstScope() simply has to return the value of the pointer to the top scoped memory, while push and pop have respectively to detect if the memory area being pushed or popped is a scoped memory, and if so, update the top pointer for the scoped memory and the pointer in the doubly linked list. Something else that is worth noticing, is jRate avoids using *instanceof* in order to determine the type of a the memory area being pushed or popped, since this is usually implemented as a linear time operation in the height of the class hierarchy. To have a constant time type identification, jRate uses its own type encoding for all those classes that often require an *instanceof*.

When a scope stack is destroyed and the reference counts of all scoped memory areas still on the stack must be decremented, jRate knows exactly which entries are scoped memory and which are not. This knowledge enables it to



**Figure 9. The jRate Scope Stack structure.**

elide a test on the type of memory area and avoids blindly searching for scoped memories on the stack.

The size of the jRate scope stack is fixed after the real-time thread is created. This design make jRate more efficient by avoiding the use of pointers to implement the doubly-linked list. The performance of jRate thread implementation is analyzed in Section 5.3.2.

### 3.3 Asynchrony

jRate provides a robust and efficient asynchronous event handling implementation that avoids priority inversion and provides lock free dispatch on most platforms.<sup>5</sup> jRate uses priority queues ordered by the execution eligibility of the handlers to dispatch asynchronous events. Execution eligibility is the ordering mechanism used throughout jRate, e.g., it is used to achieve total ordering of schedulable entities whose QoS are expressed in various ways. The performance of jRate thread implementation is analyzed in Section 5.3.3.

### 3.4 High Resolution Time and Clock

jRate implements the RTSJ high resolution time API. Different implementations of real-time clocks are provided. Depending on the underlying hardware and OS platform, resolution from nanoseconds up to microseconds can be obtained. In particular, on Pentium platforms, high resolution time with a resolution close to the processor frequency is obtained by using the read time stamp counter (RDTSC) register.

<sup>5</sup>On certain platforms, such as Compaq Alpha, the assumptions that we rely upon to avoid locking do not hold, so for those platforms jRate must use locks.

## 3.5 Timers

jRate implements periodic and one-shot timers in accordance to the RTSJ. A thread is associated with each timer (the RI takes a similar approach). Periodic timers are implemented by relying on the behavior provided by periodic threads, where as one-shot timers use a real-time thread with custom logic to generate the event at the right time. The priority of the thread is inherited by the priority of the most eligible handler registered with the timer. An analysis of the performance of the jRate timers implementation appears in [5].

## 4 Overview of RTJPerf

This section presents an overview of the RTJPerf benchmarking suite and shows which RTSJ features from Section 2 are covered by these benchmarks.

### 4.1 Assessing RTSJ Effectiveness

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

- **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 provide 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 paper 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 benchmarking suite called RTJPerf that we have developed at UCI.

### 4.2 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 of this paper.

### 4.2.1 Memory

RTJPerf provides the following benchmarks that measure 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. Section 5.3.1 present the results of this test for two RTSJ implementations.

**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.<sup>6</sup>

---

<sup>6</sup>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-specific way, *e.g.*, using `malloc()` or `mmap()`.

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. The results of this test are presented in Section 5.3.1.

### 4.2.2 Asynchrony

RTJPerf provides the following benchmarks 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 5.3.3.

**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, RTJPerf can determine the degree of priority inversion present in the underlying RTSJ event dispatching implementation. Section 5.3.3 provides an analysis of the implementation of the current RI and presents how `jRate` overcomes some RI shortcomings.

### 4.2.3 Threads

Since the `NoHeapRealtimeThread` can have execution eligibility higher than the garbage collector<sup>7</sup>, it cannot allocate nor reference any heap objects. The scheduler controls the execution eligibility. `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 predictability. 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 can cause a context switch in one of the following two 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 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 the first of these tests is presented in Section 5.3.2. The results for the second type of test appear in [5].

**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 its APIs for scheduling real-time thread execution 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

<sup>7</sup>The `RTSJ v1.0` specification states that the `NoHeapRealtimeThread` have always execution eligibility higher than the GC, but this has been changed in the `v1.01`

executed is important since excessive jitter is detrimental to most real-time applications.

`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 5.3.2.

**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 5.3.2.

### 4.2.4 Timers

`RTJPerf` provides the following benchmarks 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. Performances results for these tests appear in [5].

**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. Performances results for these tests appear in [5].

## 4.3 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<sup>8</sup> instruction [8] 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 10, 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 mea-



Figure 10. Time Measurement in RTSJ.

surement will 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, the timing measurement are largely independent of the underlying JNI implementation.

## 5 Performance Results and Analysis

This section first describes our real-time Java testbed and outlines the various Java implementations used for the tests. We then present and analyze the results obtained running the RTJPerf test cases discussed in Section 4.2 in our testbed.

<sup>8</sup>The RDTSC is a 64 bit counter that can be read with the single x86 assembly instruction RDTSC.

### 5.1 Overview of the Hardware and Software Testbed

The test results reported in this section were obtained on an Intel Pentium III 733 MHz with 256 MB RAM, running Linux RedHat 7.3 with the TimeSys Linux/RT 3.1 kernel [9]. This is the TimeSys Linux/RT NET commercial version that implements priority inheritance protocols, high resolution timers, and a resource kernel that supports resource reservation. The Java platforms used to test the RTSJ features described in Section 4 are described below:

**TimeSys RTSJ RI.** TimeSys has developed the official RTSJ Reference Implementation (RI) [4], which is a fully compliant implementation of Java [2, 10] that implements all the mandatory features in the RTSJ. The RI is based on a Java 2 Micro Edition (J2ME) JVM and supports an interpreted execution mode *i.e.*, there is no just-in-time (JIT) compilation. Run-time performance was intentionally not optimized since the main goal of the RI was predictable real-time behavior and RTSJ-compliance. The RI runs on all Linux platforms, but the priority inversion control mechanisms are available to the RI only when running under TimeSys Linux/RT [9], *i.e.*, the commercial version.

Figure 6(b) shows the structure of the resulting platform. As the figure shows, this is the classical Java approach in which bytecode is interpreted by a JVM that was written for the given host system. The TimeSys RI was designed as a proof of concept for the RTSJ, rather than as a production JVM. The production-quality TimeSys jTime that will be released later this year should therefore have much better performance.

**UCI jRate.** jRate is an open-source RTSJ-based extension of GCJ runtime systems that we are developing at UCI. By relying on GCJ, jRate provides an ahead-of-time compiled platform for the development of RTSJ-compliant applications. The research goal of jRate is to explore the use of Aspect-Oriented Programming (AOP) [11] techniques to produce a high-performance, scalable, and predictable RTSJ implementation. AOP enables developers to select only the RTSJ *aspects* they use, thereby reducing the jRate runtime memory footprint.

The jRate model shown in Figure 6(a) is different than the JVM model depicted in Figure 6(b) since there is no JVM interpreting Java bytecode. Instead, the Java application is ahead-of-time compiled into native code. The Java and RTSJ services, such as garbage collection, real-time threads, scheduling etc., are accessible via the GCJ and jRate runtime systems, respectively.

### 5.2 Compiler and Runtime Options

The following options were used when compiling and running the benchmarks described in Section 4.

**TimeSys RTSJ RI.** The Java code for the tests was compiled with `jikes` [12] using the `-O` option. The TimeSys RTSJ RI JVM was always run using the `-Xverify:none` option. The environment variable that controls the size of the immortal memory was set as `IMMORTAL_SIZE=9000000`.

**UCI jRate.** The Java code for the test was compiled with GCJ with the `-O3` flag and statically linked with the GCJ and `jRate` runtime libraries. The immortal memory size was set to the same value as the RI. `jRate` v0.3 was used along with GCJ 3.2.1.

### 5.3 RTJPerf Benchmarking Results

This section presents the results obtained when running the tests discussed in Section 4.2 in the RTSJ testbed described above. We analyze the results and explain why the TimeSys RI and `jRate` RTSJ implementations performed differently.<sup>9</sup>

We provide average- and worst-case behavior, along with dispersion indexes, for all the RTSJ Java features we measured. The standard deviation indicates the dispersion of the values of features we measured. For certain tests, we provide sample traces that are representative of all the measured data. The measurements performed in the tests reported in this section are based on *steady state* observations, where the system is run to a point at which the transitory behavior effects of *cold starts* are negligible before executing the tests.

#### 5.3.1 Memory Benchmark Results

Below we present and analyze the results of the RTJPerf memory benchmarks that were described in Section 4.2.1.

**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.** To measure the average allocation time incurred by the RI implementation of `LTMemory` and `VMemory`, we ran the RTJPerf allocation time test for allocation sizes ranging from 32 to 16,384 bytes. Each test samples 1,000 values of the allocation time for the given allocation size. This test also measured the average allocation time of `jRate`'s `CTMemory` implementation described in Section 3.1. Figure 7 shows how `jRate`'s `CTMemory` implementation relates to the memory areas defined by the RTSJ, which are depicted in Figure 2.

<sup>9</sup>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.

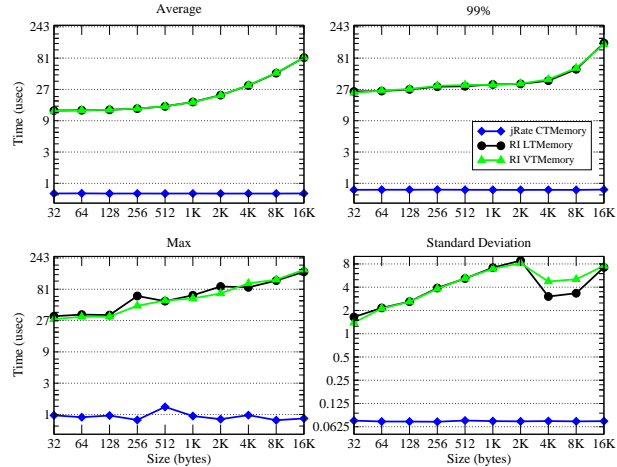


Figure 11. Scoped Memory Allocation Time Statistics.

**Test Results.** The data obtained by running the allocation time tests were processed to obtain an average, dispersion, and worst-case measure of the allocation time. We compute both the average and dispersion indexes since they indicate the following information:

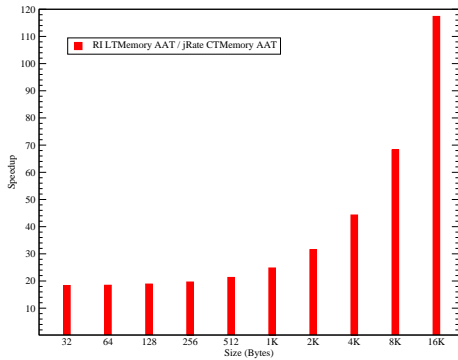
- How predictable is the behavior of a scope memory implementation
- How much variation in allocation time can occur and
- How the worst-case behavior compares to the average-case and to the case that provides a 99% upper bound.<sup>10</sup>

Figure 11 shows the resulting average allocation time for the different test runs, and it also shows the standard deviation of the allocation time measured in the various test settings. Figure 12 shows the performance ratio between `jRate`'s `CTMemory`, and the RI `LTMemory`. This ratio indicates how many times smaller the `CTMemory` average allocation time is compared to the average allocation time for the RI `LTMemory`.

**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 Figure 11, both `LTMemory` and `VMemory` provide linear time allocation with respect to the allocated memory size. Since similar results were found for other measured statistical parameters we infer that the RI implementation of `LTMemory` and `VMemory` are similar, so we focus primarily on the `LTMemory` since our results also apply to `VMemory`. `jRate` has an average allocation time that is independent of the allocated

<sup>10</sup>By “99% upper bound” we mean that value that represents an upper bound for the measured values in the 99th percentile of the cases.



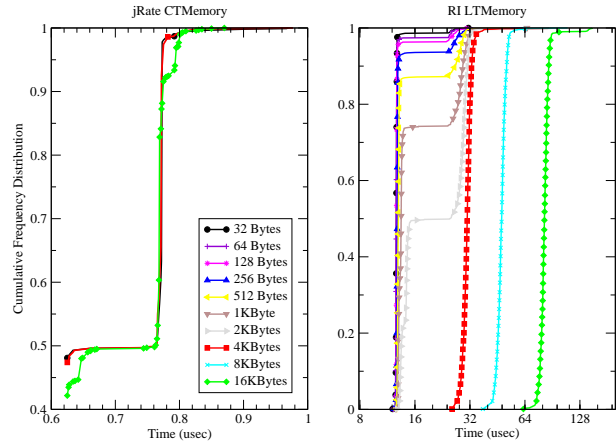
**Figure 12. Speedup of the CTMemory Average Allocation Time Over the LTMemory Average Allocation Time.**

chunk, which helps analyze the timing of RTSJ code, even without knowing the amount of memory that will be needed. Figure 12 shows that for small memory chunks the jRate memory allocator is nearly twenty times faster than RI’s LTMemory. For the largest chunk we tested, jRate’s CTMemory is ~120 times faster RI’s LTMemory.

- **Dispersion Measures**—The standard deviation of the different allocation time cases is shown in Figure 11. This deviation increases with the chunk size allocated for both LTMemory and VTMemory until it reaches 4 Kbytes, where it suddenly drops and then it starts growing again. On Linux, a virtual memory page is exactly 4 Kbytes, but when an array of 4 Kbytes is allocated the actual memory is slightly larger to store freelist management information. In contrast, the CTMemory implementation has the smallest variance and the flattest trend.

The plots in Figure 13 show the cumulative relative frequency distribution of the allocation time for some of the different cases discussed above. These plots illustrate how the allocation time is distributed for different types of memory and different allocation sizes. For any given point  $t$  on the  $x$  axis, the value on the  $y$  axis indicates the relative frequency of allocation time for which  $AllocationTime \leq t$ . The standard deviations shown in Figure 13 and Figure 11 provide insights on how the measured allocation time is dispersed and distributed. It is interesting to note that the cumulative frequency distribution for jRate is S-shaped. Moreover, the values are mostly concentrated in the two flat regions of the “S”.

The shape of distribution in Figure 13 reveals a characteristic of jRate’s allocator implementation. The GCJ runtime requires all the objects to be allocated at the



**Figure 13. Allocation Time Cumulative Relative Frequency Distribution.**

8-byte boundaries.<sup>11</sup> As a result, padding may need to be computed, depending whether the size of the object plus the size of the header is a multiple of 8 or not (see Figure 8). The two regions in the cumulative distribution show the two different cases.

- **Worst-case Measures**—Figure 11 shows the bounds on the allocation time for jRate’s CTMemory and the RI LTMemory. Each of these graphs depicts the average- and worst-case allocation times, along with the 99% upper bound of the allocation time. Figure 11 illustrates how the worst-case execution time for jRate’s CTMemory is at most ~1.8 times larger than its average execution time.

Figure 11 shows how the maximum, average, and the 99% case, for the RI LTMemory, converge as the size of the allocated chunk increases. The minimum ratio between the worst- and average-case allocation times is ~1.8 for a chunk size of 16K. Figure 11 and Figure 13 also characterize the distribution of the allocation time. Figure 13 shows how for some allocation sizes, the allocation time for the RI LTMemory is centered around two points.

**Scoped Memory Lifetime Test.** This tests measures the time taken to create, enter, and exit a memory scope. Since the RI LTMemory and VTMemory are equivalent (as discussed in the previous test’s results), we only consider jRate CTMemory and the RI LTMemory.

**Test Settings.** To measure scoped memory creation, enter, and exit time, we ran the RTJPerf scoped memory timing test for memory sizes ranging from 4,096 to 1,048,576 bytes. The test was designed to ensure that the allocated objects overrode the finalizer, which enabled a

<sup>11</sup>This is done because the lowest 3 bit are used to store locking information.

worst-case measurement of the exit time. For each test, 500 values were sampled for each of the measured variables. This tests was run to get the relevant measures for the RI's LTMemory and jRate's CTMemory.

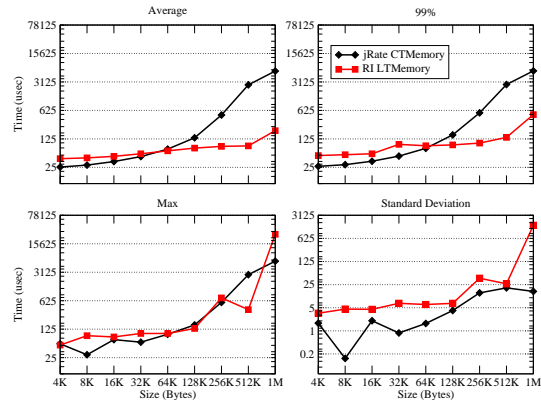
**Test Results.** Figure 14, 15, and 16 contain the average, 99%, maximum, and standard deviation trend for memory scope creation, enter, and exit time. Figure 17 shows the execution time, measured as the time needed to fill the memory with the objects.

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

- **Average Measures**—From Figure 14, we can see how the jRate CTMemory has better creation times on average than the RTSJ RI for scoped memory with size less or equal than 64 KBytes. After this point, jRate starts performing worse than the RI. The explanation of this behavior stems from the fact that the CTMemory maps the Linux /dev/zero device and then locks the allocated chunk. Simply locking the memory does not reserve physical memory for the calling process, however, since the pages might be copy-on-write. To improve the predictability of an application, therefore, it is usually good habit to make sure that at least a byte each page of the locked chunk of memory is accessed. The time taken to write at least one byte per each allocated page, has a time complexity of  $O(n)$  in the size of the scoped memory allocated, which makes creation time depend linearly on memory size.

Figure 15 shows the enter time trend for jRate's CTMemory and for the RI's LTMemory. From this diagram we can see that while the RI enter time varies only slightly with the scoped memory size, for jRate the enter time depends more on the size. This behavior stems from the fact that jRate's scoped memory puts pressure on the cache and induces cache misses in succeeding instructions, which degrades the execution time of the first enter (which is the one measured by the test).

Figure 16 shows the results for the exit time. From these diagram we see that even if jRate's CTMemory has to zero the allocated memory other than finalizing the allocated objects, its performance is close to the RI. Exit time depends primarily on the efficiency of JNI and on the efficiency of the native implementation that manages the scope stack, the finalization, and the cleanup of the memory, if any is needed. In this case, jRate's ahead-of-time compilation model plays a minor role, especially as the size of the memory grows.



**Figure 14. Average, 99%, Maximum and Standard deviation of the Creation Time ( $\mu$ sec).**

- **Dispersion Measures**—Figures 14, 15, 16 and 17 show how jRate and the RI have very low dispersion values for small memory sizes. These dispersion values grow with the size of the scoped memory for both implementations. The RI becomes less predictable as the size of the memory scope increases, culminating in pathological cases shown in Figure 14 and Figure 17.
- **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. In contrast, the RI'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 RI was not available for inspection. A reasonable guess, however, is that the RI allocators rely directly on the system provided `malloc()` for each of the allocated objects. This explanation justifies both the relatively small creation time, and also the degradation of the predictability when the allocator creates many objects to fill the scoped memory.

### 5.3.2 Thread Benchmark Results

Below, we present and analyze the results from the yield and thread creation latency benchmarks that were described in Section 4.2.3.

**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 RTSJ platform we evaluated, we collected 1,000 samples of the the context switch time, which we forced by explicitly yielding the CPU. Real-time threads were used for the RI and jRate, and immortal memory was used as allocation context for the threads.

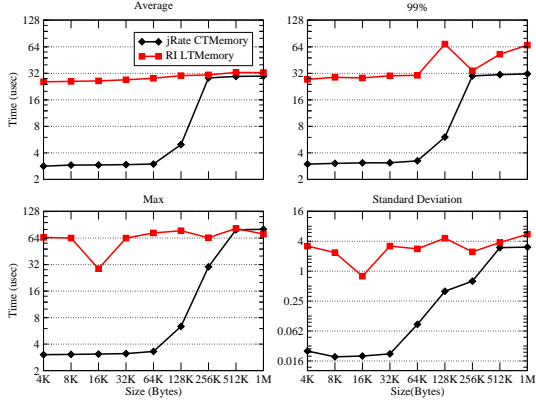


Figure 15. Average, 99%, Maximum and Standard deviation of the Enter Time ( $\mu\text{sec}$ ).

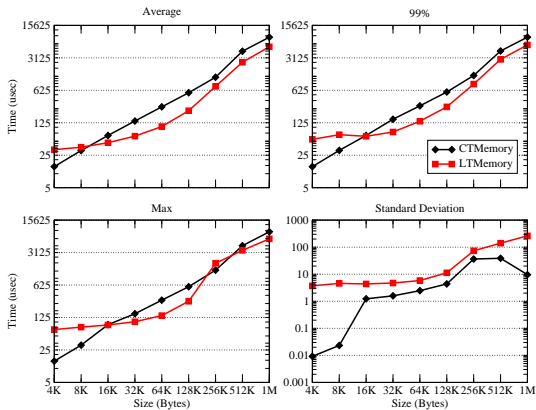


Figure 16. Average, 99%, Maximum and Standard deviation of the Exit Time ( $\mu\text{sec}$ ).

**Test Results.** Table 1 reports the average and standard deviation for the measured context switch in the various platforms.

	Average	Std. Dev.	Max	99%
<b>jRate</b>	1.449 $\mu\text{s}$	0.002 $\mu\text{s}$	1.475 $\mu\text{s}$	1.459 $\mu\text{s}$
<b>RI</b>	2.858 $\mu\text{s}$	0.0198 $\mu\text{s}$	3.052 $\mu\text{s}$	2.900 $\mu\text{s}$
<b>C++</b>	1.30 $\mu\text{s}$	0.002 $\mu\text{s}$	N/A	N/A

Table 1. Yield Context Switch Average, Standard Deviation, Max and 99% Bound.

**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 1 shows how the RI fairly well in this test, *i.e.*, its context switch time is only  $\sim 2 \mu\text{s}$  larger than jRate’s. The main reason for jRate’s better performance stems from its use of ahead-of-time compilation. The last row of Table 1 reports the results of a C++-based context switch test described in [13].

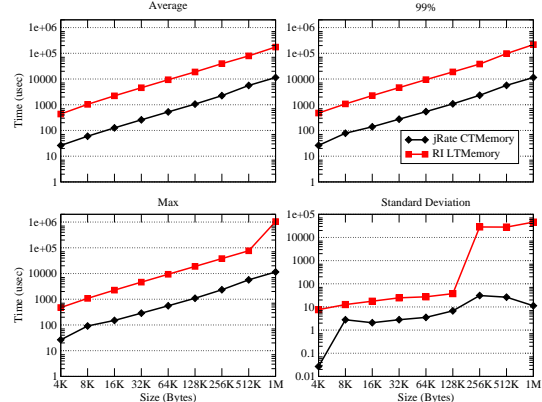


Figure 17. Average, 99%, Maximum and Standard deviation of the Execution Time ( $\mu\text{sec}$ ).

The table shows how the context switch time measured for the RI and jRate is similar to that for C++ programs on TimeSys Linux/RT. The context switching time for the RI is less than three times larger than that found for C++, whereas the times for jRate are roughly the same as those for C++.

- **Dispersion Measures**—The third column of Table 1 reports the standard deviation for the context switch time. Both jRate and the RI exhibit tight dispersion indexes, indicating that context switching overhead is predictable for these implementations. In general, the context switch time for jRate is as predictable as C++ on our Linux testbed platform.
- **Worst-case Measures**—The fourth and fifth column of Table 1 represent the maximum and the 99% bound for the context switch time, respectively. jRate and the RI have 99% bound and worst-case context switching times that are close to their average-case values.

**Thread Creation Latency Test.** This test measures the time needed to create a thread, which consists of the time to create the thread instance itself and the time to start it. As described in Section 4.2.3, this test exists into two variants. Here we refer to the test that creates and starts a real-time thread from another real-time thread. The results we obtained are presented and analyzed below.

**Test Settings.** For each platform in our test suite, we collected 1,000 samples of the thread creation time and thread start time. Real-time threads were used for the RI and jRate. Since we are interested in measuring the time taken to create and start a thread – while limiting the effects of other delays – the threads had immortal memory as their allocation context, so to avoid garbage collection overhead.

**Test Results.** Table 2 and Table 3 report the average, standard deviation, maximum and 99% bound for the thread creation time and thread start time respectively.

	Average	Std. Dev.	Max	99%
<b>jRate</b>	17.690 $\mu s$	3.768 $\mu s$	56.526 $\mu s$	33.815 $\mu s$
<b>RI</b>	890.190 $\mu s$	227.260 $\mu s$	1306.550 $\mu s$	1276.15 $\mu s$

**Table 2. Thread Creation Time Statistical Indexes.**

	Average	Std. Dev.	Max	99%
<b>jRate</b>	118.671 $\mu s$	5.005 $\mu s$	169.552 $\mu s$	145.878 $\mu s$
<b>RI</b>	800.149 $\mu s$	29.310 $\mu s$	894.578 $\mu s$	867.250 $\mu s$

**Table 3. Thread Startup Time Statistical Indexes.**

**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**—The second column of Table 2 and Table 3 show that **jRate** has the best average thread creation time and thread start time. The **RI**'s creation time grows linearly in the **RT** test, which is unusual and may reveal a problem in how the **RI** manages the scope stack. The smallest creation time experienced by the **RI** is around 500  $\mu s$  (which is much higher than **jRate**) and one reason may be the **RI**'s scope stack implementation.
- **Dispersion Measures**—The third column of Table 2 and Table 3 present the standard deviation for thread creation time and thread start time, respectively. **jRate** has the smallest dispersion of values for the thread startup time and creation time.

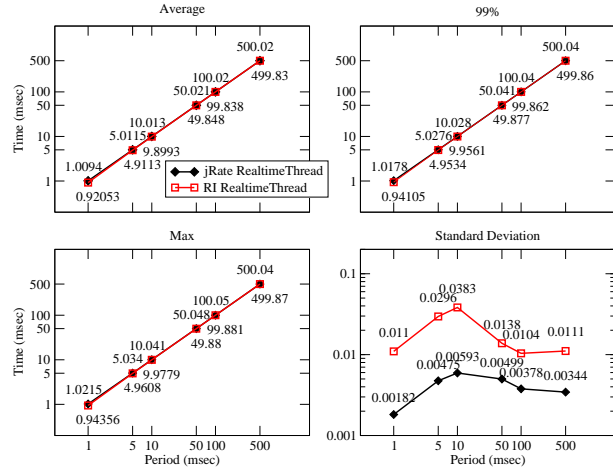
The standard deviation associated with the thread creation time is influenced by the predictability of the time the memory allocator takes to provide the memory needed to create the thread object. In contrast, the standard deviation of the thread start time depends largely on the OS, whereas the rest of thread start time depends on the details of the thread startup method implementation.

- **Worst-case Measures**—The fourth and fifth column of Table 2 and Table 3 present the maximum and the 99% bound for the thread creation time, and the thread startup time, respectively. These tables clearly show how **jRate** has a more predictable startup time than creation time. The jitter introduced in the creation time likely stems from the fact that **jRate** allocates several data structure via `malloc()`, and does not take yet advantage of custom allocators. The **RI** also has fairly good startup time, though we cannot make any comparison with its creation time due to an **RI** memory leak exposed by this test. The problem is related to **RI**'s management of the scope stack since it appears only when a `RealtimeThread` is created by another `RealtimeThread`.

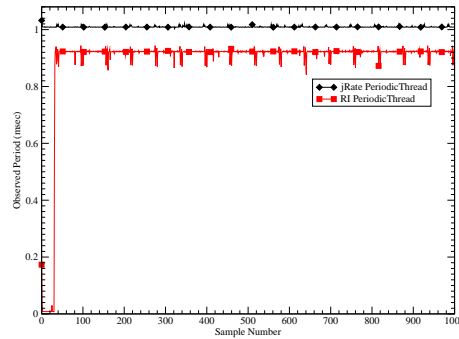
**Periodic Thread Test.** This test measures the accuracy with which the `waitForNextPeriod()` method in the `RealtimeThread` class schedules the thread's execution periodically. The results we obtained are presented and analyzed below.

**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  $1ms$ ,  $5ms$ ,  $10ms$ ,  $50ms$ ,  $100ms$ , and  $500ms$ .

**Test Results.** Figure 18 shows average and dispersion values that we measured for this test.<sup>12</sup>



**Figure 18. Measured Period Statistics.**



**Figure 19. Trace of the measured period for  $T = 1ms$ .**

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

<sup>12</sup>Whenever the plot for **jRate** and the **RI** overlap, the values for **jRate** are shown above the graph and the value for the **RI** are shown below the graph.

- **Average Measures**—Figure 18 shows that both jRate and the RI have an average period that is quite close to the target period. Whereas RI 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 RI 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 RI’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 slack time that should be considered as an approximation of the overhead of calling the `waitForNextPeriod()` and then getting the control back.

As stated near the beginning of Section 5.3, the measure for each test are made in stationary conditions. The Periodic Thread Test was interesting since the RI took a relatively long time to reach the steady state, particularly for small periods. Figure 19 shows the trace for a period of  $1ms$ . It is interesting to observe that while jRate shows no transitory behavior, the RI has a long period of transitory behavior, which runs for  $\sim 40$  samples.

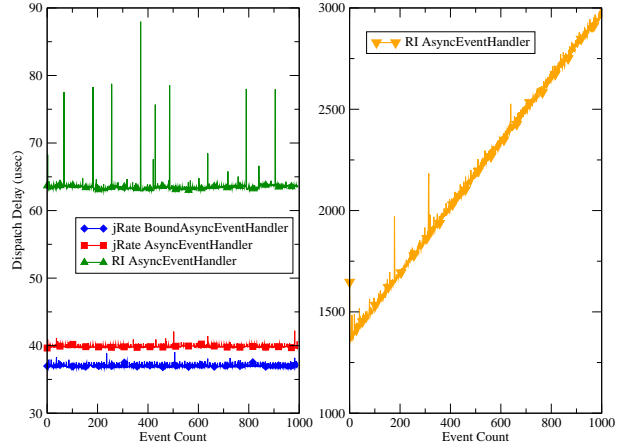
- **Dispersion Measures**—Figure 18 shows the dispersion of the measured period for both jRate and the RI has the same trend. While jRate generally has less dispersed values than the RI, both implementation are quite predictable.
- **Worst-case Measures**—As shown in Figure 18 both jRate and the RI have 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 RI values are not much further away.

### 5.3.3 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 4.2.2.

**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.

**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 4.2.2 with a fire count of 1,000 for both RI and jRate. To ensure



**Figure 20. Dispatch Latency Trend for Successive Event Firing.**

	AsyncEventHandler	BoundAsyncEventHandler
<b>Avg.</b>	39.884 $\mu s$	36.809 $\mu s$
<b>Std. Dev.</b>	0.2115 $\mu s$	0.231 $\mu s$
<b>Max</b>	42.211 $\mu s$	39.376 $\mu s$
<b>99%</b>	40.665 $\mu s$	37.504 $\mu s$

**Table 4. jRate Event Handler’s Dispatch Latency Average, Standard Dev., Max and 99% bound for the Different Settings**

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 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.** Figure 20 shows the trend of the dispatch latency for successive event firings (since the RI’s `AsyncEventHandler` trend is completely off the scale, it is reported in a separate plot in Figure 20). The data obtained by running the dispatch delay tests were processed to obtain average and worst-case behavior, and the dispersion measure of the dispatch latency. Table 4 and Table 5 shows the results found for jRate and the RI, respectively.

**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**—Table 5 illustrates the large average dispatch latency incurred by the RTSJ RI `AsyncEventHandler`. The results in Figure 20 show how the actual dispatch latency increases as the event count increases. By tracing the memory used when running the test using heap memory, we found

	AsyncEventHandler	BoundAsyncEventHandler
Avg.	2177.3 $\mu s$	63.600 $\mu s$
Std. Dev.	463.59 $\mu s$	1.4813 $\mu s$
Max	2997.8 $\mu s$	87.968 $\mu s$
99%	2953.7 $\mu s$	68.297 $\mu s$

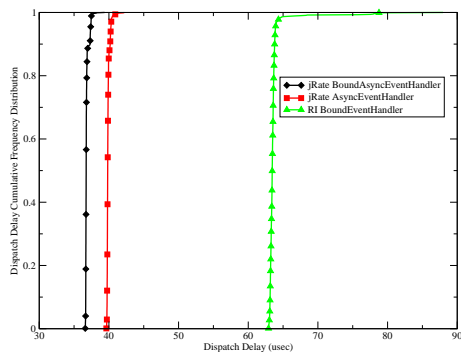
**Table 5. RI Event Handler’s Dispatch Latency Average, Standard Dev., Max and 99% bound for the Different Settings**

that not only did memory usage increased steadily, but even invoking the GC explicitly did not free any memory.

These results reveal a problem with how the RI manages the resources associated to threads. The RI’s AsyncEventHandler creates a new thread to handle a new event, and the problem appears to be a memory leak in the underlying RI memory manager associated with threads, rather than a limitation with the model used to handle the events. In contrast, the RI’s BoundAsyncEventHandler performs quite well, *i.e.*, its average dispatch latency is slightly less than twice as large as the average dispatch latency for jRate.

Figure 20 and Table 4 show that the average dispatch latency of jRate’s AsyncEventHandler is the same order of magnitude as its BoundAsyncEventHandler. The difference between the two average dispatch latency stems from jRate’s AsyncEventHandler implementation, which uses an *executor* [14] thread from a pool of threads to perform the event firing, rather than having a thread permanently bound to the handler.

- **Dispersion Measures**—The results in Table 5, Table 4, Figure 20, and Figure 21 illustrate how jRate’s BoundAsyncEventHandler dispatch latency incurs the least jitter. The dispatch latency value disper-



**Figure 21. Cumulative Dispatch Latency Distribution.**

sion for the RTSJ RI BoundAsyncEventHandler

is also quite good, though its jitter is higher than jRate’s AsyncEventHandler and BoundAsyncEventHandler. The higher jitter in RI may stem from the fact that the RI stores the event handlers in a `java.util.Vector`. This data structure achieves thread-safety by synchronizing all method that `get()`, `add()`, or `remove()` elements from it, which acquires and releases a lock associated with the vector for each method.

To avoid the locking overhead incurred by the RI, 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 itself 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 [15].

- **Worst-case Measures**—Table 4 illustrates how the jRate’s BoundAsyncEventHandler and AsyncEventHandler have worst-case execution time that is close to its average-case. The worst-case dispatch delay of the RI’s BoundAsyncEventHandler is not as low as the one provided by jRate, due to differences in how their event dispatching mechanisms are implemented. The 99% bound differs only on the first decimal digit for both jRate and the RI (we do not consider the RI’s AsyncEventHandler since no bound can be put on its behavior, as discussed below).

### 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. Only the BoundAsyncEventHandler performance is measured, however, because the RI’s AsyncEventHandlers are essentially unusable since their dispatch latency grows linearly with the number of event handled (see Figure 20), which masks any priority inversions. Moreover,

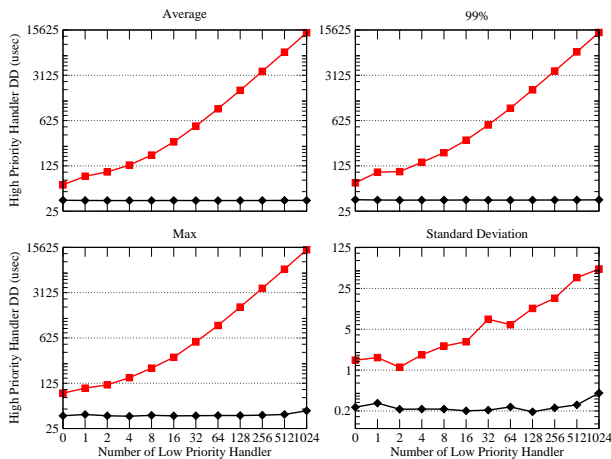


Figure 22.  $H$ 's Dispatch Latency Statistics.

`jRate`'s `AsyncEventHandler` performance is similar to its `BoundAsyncEventHandler` performance, so the results obtained from testing one applies to the other. 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 22 shows how the average, standard deviation, maximum and 99% bound of the dispatch delay changes for  $H$  as the number of low-priority handlers increase.

**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 RI.

- **Average Measures**—Figure 22 illustrates that the average dispatch latency experienced by  $H$  is essentially constant for `jRate`, regardless of the number of low-priority handlers. It grows rapidly, however, as the number of low-priority handlers increase for the RI. The RI's event dispatching priority inversion is problematic for real-time applications and stems from the fact that its queue of handlers is implemented with a `java.util.Vector`, which is not ordered by the *execution eligibility*. In contrast, the priority queues in `jRate`'s event dispatching are ordered by the execution eligibility of the handlers.

Execution eligibility is the ordering mechanism used throughout `jRate`. For example, it is used to achieve total ordering of schedulable entities whose QoS are expressed in different ways.

- **Dispersion Measures**—Figure 22 illustrates how  $H$ 's dispatch latency dispersion grows as the number of low-priority handlers increases in the RI. The dispatch latency incurred by  $H$  in the RI therefore not only grows with the number of low-priority handlers, but its variability increases *i.e.*, its predictability decreases. In contrast, `jRate`'s standard deviation increases very little as the low-priority handlers increase. As mentioned in the discussion of the average measurements above, the difference in performance stems from the proper choice of priority queue.
- **Worst-Case Measures**—Figure 22 illustrates how the worst-case dispatch delay is largely independent of the number of low-priority handlers for `jRate`. In contrast, worst-case dispatch delay for the RI increases as the number of low-priority handlers grows. The 99% bound is close to the average for `jRate` and relatively close for the RI.

## 6 Related Work

Although the RTSJ was adopted as a standard fairly recently [3], there are already a number of related research projects. The following projects are particularly interesting:

- The **FLEX** [16] 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** [17] 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, field access, and speed. OVM allows dynamic updates of the implementation of instructions on a running VM.
- Work on real-time storage allocation and collection [18] 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 [19] 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 [20].

- Wellings and Burns [21] propose an architecture for implementing asynchronous event handling, and performing feasibility analysis on the schedulability of the event handlers. The architecture proposed is similar to the architecture used to implement one type of `jRate` Asynchronous Event Handler (specifically those that rely on pool of executors described in Section 5.3.3), though `jRate` does not yet include feasibility analysis.

There are several ways in which we plan to leverage our work on `jRate` and the work being done in the FLEX, OVM, and real-time allocator projects outlined above. For instance, the `jRate` RTSJ library implementation could become the library used by the OVM. This is possible because `jRate` has been designed to port easily from one Java platform to another. `jRate` could be used as the RTSJ library on which FLEX relies. Likewise, the work on real-time allocators and garbage collectors could be to implement `jRate`'s scoped memory with different characteristics than its current `CTMemory` design.

## 7 Concluding Remarks

The RTSJ is an important emerging middleware platform that defines a standard, high-level, and productive environment for developing real-time embedded applications. RTSJ encapsulates much of the complexity and platform-specific details in the middleware, and provides a powerful set of programming abstractions to application developers. This paper presented the architecture of `jRate`, which is an open-source ahead-of-time compiled RTSJ middleware that we have created at UC Irvine. This paper also used the open-source `RTJPerf` benchmarking suite to empirically evaluate the performance of RTSJ middleware features in `jRate` and the TimeSys RTSJ RI that are crucial to the development of real-time embedded applications.

`RTJPerf` is one of the first open-source benchmarking suites designed to evaluate RTSJ-compliant Java implementations empirically. We believe it is important to have an open benchmarking suite to measure the quality of service of RTSJ implementations. `RTJPerf` not only helps guide application developers to select RTSJ features that are suited to their requirements, but also helps developers of RTSJ implementations evaluate and improve the performance of their products.

Although much work remains to ensure predictable and efficient performance under heavy workloads and high contention, our test results indicate that real-time Java is maturing to the point where it can be applied to certain types of real-time applications. In particular, the performance and predictability of `jRate` is approaching C++ for some tests. The TimeSys RTSJ RI also performed relatively well in some aspects, though it has problems with `AsyncEventHandler` dispatching delays and priority inversion.

Our future work on `RTJPerf` is described below:

- Since the TimeSys RI does not provide a Just In Time (JIT) or ahead-of-time compiler, its not surprising that its overall performance is generally lower than `jRate`, which is compiled ahead-of-time. In some cases, however, the main source of variation is the native implementation of certain features, such as allocation time and scoped memory entry/exit time. In such cases, `jRate` has the edge over the RI since `jRate` uses the Cygnus Native Interface (CNI) instead of the less efficient JNI used in the TimeSys RI. We look forward to measuring the performance of the commercial TimeSys RTSJ product that will be released later this year, which should have much better performance since it will contain an ahead-of-time compiler.
- `RTJPerf` test's are based on synthetic workload, which is a time-honored way of isolating key factors that affect performance. One area of future work is therefore to add tests based on representative operational real-time embedded applications. Our first target platform is Boeing Bold Stroke [20], which is a framework for avionics mission computing applications. We are collaborating with researchers from Boeing and the AFRL *Real-Time Java for Embedded Systems (RTJES)* program [19] to define a comprehensive benchmarking suite for RTSJ-based real-time Java platforms.
- `RTJPerf` currently focuses on measuring time efficiency. Clearly, however, measuring memory efficiency and predictability under heavy workloads and contention is also critical for real-time embedded systems. We are working with the Boeing Bold Stroke researchers and the RTJES team at AFRL to provide a comprehensive series of benchmarks test that quantify many QoS properties of RTSJ-compliant implementations.

In addition to implementing the remaining RTSJ features, our future work on `jRate` is focusing on the following topics:

- Developing an extensible user-level scheduling framework that leverages the simple priority-based scheduling provided by real-time operating systems to provide advanced scheduling algorithms, such as EDF, LLF, and MAU.

- Partitioning the Java and C++ parts of jRate into sets of aspects that can be woven together at compile-time to configure custom real-time Java implementations that are tailored for specific application needs.
- Providing a meta-object protocol as one of the aspects to support both computational and structural reflection. Reflection is useful for real-time applications that must manage resources dynamically. Moreover, it enables developers to customize the behavior of jRate's implementation at run-time.

By using AOP tools, such as AspectJ [22], we are decomposing jRate's into a series of aspects that can be configured by developers and tools to generate RTSJ implementations that are customized for specific needs.

The latest version of jRate is 0.3a and the latest version of RTJPerf is 0.1.1a. Both jRate and RTJPerf are available at <http://tao.doc.wustl.edu/~corsaro/jRate>. Since jRate and RTJPerf are open-source projects, we encourage researchers and developers to provide us feedback and help improve their quality and capabilities. jRate and RTJPerf use the same open-source model we use for our ACE and TAO middleware, which has proved to be successful to produce high-quality open-source software.

## Acknowledgments

We would like to thank Ron Cytron, Peter Dibble, David Holmes, Doug Lea, Doug Locke, Carlos O'Ryan, John Regehr, and Gautam Thaker for their constructive suggestions that helped to improve earlier drafts of this paper.

## References

- [1] J. Sztipanovits and G. Karsai, "Model-Integrated Computing," *IEEE Computer*, vol. 30, pp. 110–112, Apr. 1997.
- [2] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*. Boston: Addison-Wesley, 2000.
- [3] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, 2000.
- [4] TimeSys, "Real-Time Specification for Java Reference Implementation." [www.timesys.com/rtj](http://www.timesys.com/rtj), 2001.
- [5] A. Corsaro and D. C. Schmidt, "Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems," in *Proceedings of the 8<sup>th</sup> IEEE Real-Time Technology and Applications Symposium*, (San Jose), IEEE, Sept. 2002.
- [6] A. Corsaro and D. C. Schmidt, "The Design and Performance of the jRate Real-Time Java Implementation," in *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE* (R. Meersman and Z. Tari, eds.), (Berlin), pp. 900–921, Lecture Notes in Computer Science 2519, Springer Verlag, 2002.
- [7] GNU is Not Unix, "GCJ: The GNU Compiler for Java." <http://gcc.gnu.org/java>, 2002.
- [8] I. Corporation, "Using the RDTSC Instruction for Performance Monitoring," tech. rep., Intel Corporation, 1997.
- [9] TimeSys, "TimeSys Linux/RT 3.0." [www.timesys.com](http://www.timesys.com), 2001.
- [10] J. Gosling, B. Joy, and G. Steele, *The Java Programming Language Specification*. Reading, Massachusetts: Addison-Wesley, 1996.
- [11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.
- [12] IBM, "Jikes 1.17." <http://www.research.ibm.com/jikes/>, 2001.
- [13] D. C. Schmidt, M. Deshpande, and C. O'Ryan, "Operating System Performance in Support of Real-time Middleware," in *Proceedings of the 7<sup>th</sup> Workshop on Object-oriented Real-time Dependable Systems*, (San Diego, CA), IEEE, Jan. 2002.
- [14] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns, Second Edition*. Boston: Addison-Wesley, 2000.
- [15] J. D. Salehi, J. F. Kurose, and D. Towsley, "The Effectiveness of Affinity-Based Scheduling in Multiprocessor Networking," in *IEEE INFOCOM*, (San Francisco, USA), IEEE Computer Society Press, Mar. 1996.
- [16] M. Rinard et al., "FLEX Compiler Infrastructure." <http://www.flex-compiler.lcs.mit.edu/Harpoon/>, 2002.
- [17] OVM/Consortium, "OVM An Open RTSJ Compliant JVM." <http://www.ovmj.org/>, 2002.
- [18] S. M. Donahue, M. P. Hampton, M. Deters, J. M. Nye, R. K. Cytron, and K. M. Kavi, "Storage allocation for real-time, embedded systems," in *Embedded Software: Proceedings of the First International Workshop* (T. A. Henzinger and C. M. Kirsch, eds.), pp. 131–147, Springer Verlag, 2001.
- [19] Jason Lawson, "Real-Time Java for Embedded Systems (RTJES)." <http://www.opengroup.org/rtforum/jan2002/slides/java/lawson.pdf>, 2001.
- [20] D. C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," in *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.
- [21] A. Wellings and A. Burns, "Asynchronous event handling and real-time threads in the real-time specification for java," in *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 81–89, 2002.
- [22] The AspectJ Organization, "Aspect-Oriented Programming for Java." [www.aspectj.org](http://www.aspectj.org), 2001.