Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems

Angelo Corsaro, 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, and the behavior of many of these applications is constrained by the physical world. Higher-level programming languages and middleware are needed to robustly and productively design, implement, compose, integrate, validate, and enforce real-time constraints along with conventional functional requirements and reusable components.

This paper provides two contributions to the study of programming languages and middleware for real-time and embedded applications. First, we present the empirical results from applying the RTJPerf benchmarking suite to evaluate the efficiency and predictability of several implementations of the Real-time Specification for Java (RTSJ). Second, we describe the techniques used to develop jRate, which is an open-source ahead-of-time-compiled implementation of RTSJ we are developing. Our results indicate that RTSJ implementations are maturing to the point where they can be applied to a variety of real-time embedded applications.

1 Introduction

Over 90 percent 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), telemedicine (*e.g.*, remote surgery), manufacturing process automation (*e.g.*, hot rolling mills), and defense applications (*e.g.*, avionics mission computing systems). Many of these real-time embedded systems use several, hundreds, or even thousands of processors that interoperate via networks and interconnects.

Creating high quality software for real-time embedded systems is a hard problem. Although these systems have historically operated with limited memory and used processors with limited capacity, the trend is toward increasing memory and computational power. Moreover, increasingly complex application software is being developed to control and utilize these processors.

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, tools, and methods that enable higher software productivity.

Many real-time embedded systems are still developed in C, and increasingly also 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++ is associated with their *memory management* mechanisms, which require programmers to allocated and deallocate memory manually. Moreover, C++ is a feature rich, complex language that has a steep learning curve, which makes it hard to find and retain experienced real-time embedded developers trained in this language.

Real-time embedded software should ultimately be synthesized from high-level specifications expressed with domain-specific modeling tools [21]. 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 a programming language that shields them from many accidental complexities, such as type errors, memory management, and steep learning curves. The Java [1] programming language has become an attractive choice for the following rea-

*This work was supported in part by Siemens MED, SAIC, and ATD.

sons:

- 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 a virtual machine architecture—the Java Virtual Machine (JVM)—that allows Java applications to run on any platform that supports a JVM.
- It has a powerful, 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 desirable language features, such as strong typing, dynamic class loading, and reflection/introspection.
- It defines portable support for concurrency and synchronization.
- 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 that 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 JVMs 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) [2], 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 [24]. In this paper, therefore, we empirically analyze most of the RTSJ features and compare the performance of the RTSJ RI with other popular and emerging real-time Java implementations.

The remainder of the paper is organized as follows: Section 2 describes RTJPerf, which is an open-source benchmarking suite we developed to evaluate the performance of RTSJ implementations; Section 3 presents the results obtained by applying RTJPerf to measure the performance of the RTSJ RI and compare/contrast these results with the performance of JDK 1.4.0 and jRate (which is an ahead-of-time compiled implementation of RTSJ we are developing); Section 4 compares our work with related research; and Section 5 summarizes the results we obtained and outlines how they can be used to improve the support of next-generation implementations of RTSJ for real-time embedded software.

2 Overview of RTJPerf

This section presents an overview of the RTJPerf benchmarking suite. In addition to describing what RTSJ features RTJPerf measures, we summarize the key RTSJ features themselves.

2.1 Assessing Real-Time (RT)–JVM Effectiveness

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 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[1] benchmarking suite called RTJPerf that we are developing.

2.2 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.

2.2.1 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 the thread T_1 and T_2 evolves while the thread moves from one memory area to another. As shown in Figure 2,

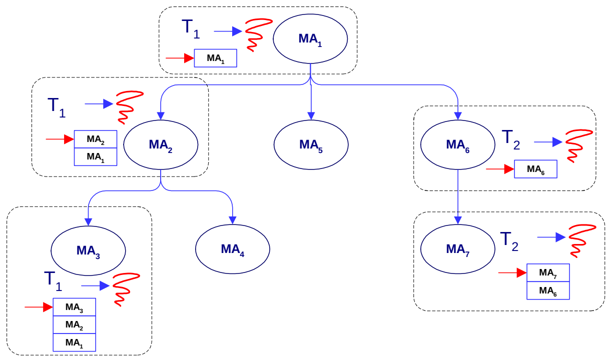

Figure 1. Thread Scope Stack in the RTSJ Memory Model.

the RTSJ specification provides scoped memories with linear and variable allocation times (`LTMemory`, `LTPhysicalMemory` and `VTMemory`, `VTPhysicalMemory`, respectively). For 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 $C > 0$.[2]

RTJPerf provides the following test that measures key performance properties of RTSJ memory area implementations.

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

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

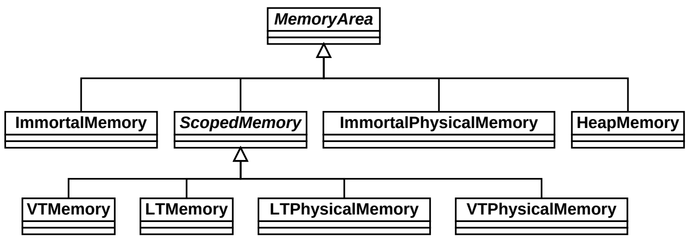

Figure 2. Hierarchy of Classes in the RTSJ Memory Model.

Allocation Time Test. To minimize memory leaks, latency, and non-determinism, the use of dynamic memory allocation is forbidden or strongly discouraged in many real-time embedded systems. The scoped memory specified by the RTSJ is designed to provide a relatively fast and safe way to allocate memory that has much of the flexibility of dynamic memory allocation, but much of the efficiency 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 *time efficiency* of the 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. By running this test with different allocation sizes, it is possible to determine the allocation time associated with each type of scoped memory. Section 3.3.1 present the results of this test for several Java implementations.

2.2.2 Threads

The RTSJ extends the 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.

Since the `NoHeapRealtimeThread` can have execution eligibility higher than the garbage collector[3], it cannot allocate nor reference any heap objects. The scheduler controls the *execution eligibility*[4] of the instances of this class

[3]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

[4]Execution eligibility is defined as the position of a schedulable entity in a total ordering established by a scheduler over the available entities [5]. 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.

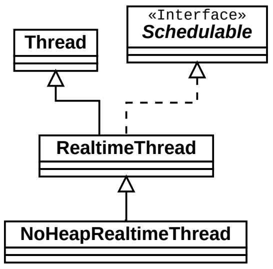

Figure 3. RTSJ Real-time Thread class hierarchy.

by using the `SchedulingParameters` associated with it.

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 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 3.3.2.

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 3.3.2.

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 3.3.2.

2.2.3 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. As shown in Figure 4, there are two types

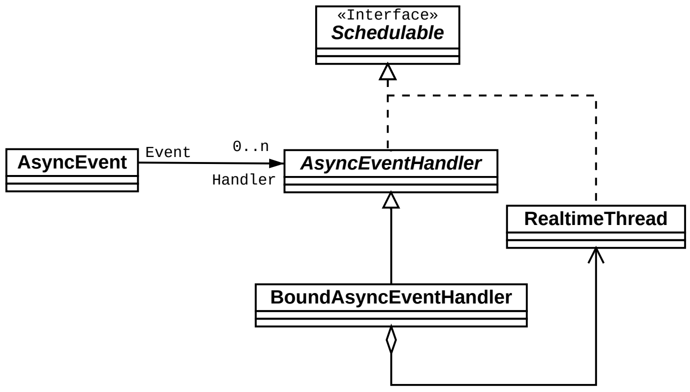

Figure 4. RTSJ Asynchronous Event Class Hierarchy.

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 `AsyncEventHandler`s will be dispatched.

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

Event handlers can also be specified to be *no-heap*, which means that the thread used to handle the event must be a `NoHeapRealtimeThread`.

Since event handling mechanisms are commonly used to develop real-time embedded systems [8], 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 3.3.3.

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, we can determine the degree of priority inversion present in the underlying RTSJ event dispatching implementation. Section 3.3.3, provides an analysis of the implementation of the current RI and presents an implementation that overcomes some shortcomings of the RI.

2.2.4 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:

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

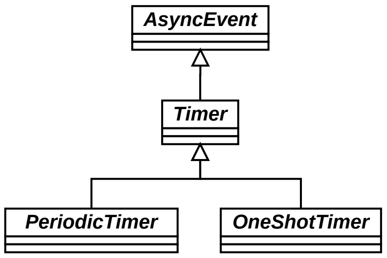

Figure 5. RTSJ Timer Class Hierarchy.

- `PeriodicTimer`, which generates events periodically.

`OneShotTimer`s and `PeriodicTimer`s events are handled by `AsyncEventHandler`s. 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. Performances results for these tests are reported in Section 3.3.4

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[5] instruction [4] 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

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

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, 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-

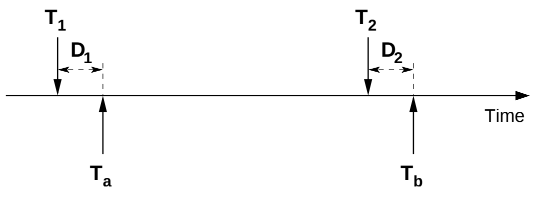

Figure 6. Time Measurement in RTSJ.

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, we can assume that the timing measurement are largely independent of the underlying JNI implementation.

3 Performance Results

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 2.2 in our testbed.

3.1 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 7.2 with the TimeSys Linux/RT 3.0 GPL[6] kernel [25]. The Java platforms used to test the RTSJ features described in Section 2 are described below:

TimeSys RTSJ RI. TimeSys has developed the official RTSJ Reference Implementation (RI) [24], which is a fully compliant implementation of Java [1, 7] 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 [25], *i.e.*, the commercial version.

[6]This OS is the freely available version of TimeSys Linux/RT and is available under the GNU Public License (GPL).

Figure 7 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

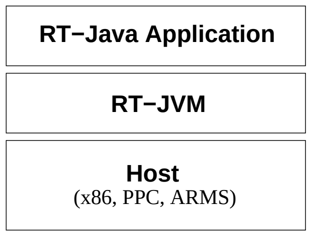

Figure 7. The Real-time JVM Architecture.

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 the GNU Java Compiler (GCJ) runtime systems that we are developing at the University of California, Irvine (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 8 is different than the JVM model depicted in Figure 7 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. One downside of ahead-of-time compiled RTSJ implementations like jRate, however, is that they can hinder portability since applications must be recompiled each time they are ported to a new architecture.

The C Virtual Machine (CVM). CVM [20] is a J2ME platform targeted for embedded and consumer electronic devices. CVM has relatively small footprint and is designed

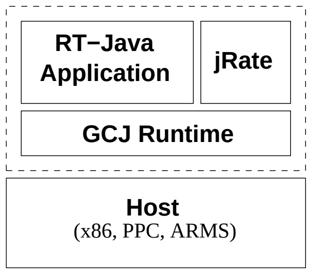

Figure 8. The jRate **Architecture.**

to be portable, RTOS-aware, deterministic, and space-efficient. It has a precise—as opposed to conservative—generational garbage collector.

JDK 1.4 JVM. Where appropriate, we compare the performance of the real-time Java implementations against the JVM shipped with the Sun's JDK 1.4, which is the latest version of Java that provides many performance improvements over previous JDK versions. Although JDK 1.4 was clearly not designed for real-time embedded systems, it provides a baseline to measure the real-time Java implementation improvements in predictability and efficiency.

3.2 Compiler and Runtime Options

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

CVM and JDK 1.4. The Java code for the tests was compiled with **jikes** [9] using the `-O` option. These JVM were always run using the `-Xverify:none` option.

TimeSys RTSJ RI. The settings used were the same as the one for CVM and JDK 1.4, additionally the environment variable that controls the size of the immortal memory was set as `IMMORTAL_SIZE=6000000`.

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

3.3 RTJPerf **Benchmarking Results**

This section presents the results obtained when running the tests discussed in Section 2.2 in the testbed described above. We analyze the results and explain why the various Java implementations performed differently.[7]

Average and worst-case behavior, along with dispersion indices, are provided for all the real-time 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.

[7]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.

3.3.1 Memory Benchmark Results

Below, we present and analyze the results of the allocation time test that was described in Section 2.2.1.

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 `VTMemory`, 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. Figure 9 shows how jRate's `CTMemory` implementation relates to the memory areas defined by the RTSJ, which are depicted in Figure 2. This test only examines jRate and

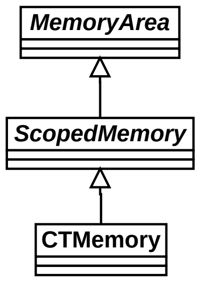

Figure 9. `CTMemory` **class hierarchy.**

the RI since the other Java platforms do not support scoped memories. We felt that comparing platforms with scoped memory against platform that lack them would be unfair since the latter would perform so poorly

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 indices since they indicate the following information:

- How predictable the behavior of an implementation is
- 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.[8]

[8]By "99% upper bound" we mean that value that represents an upper bound for the measured values in the 99th percentile of the cases.

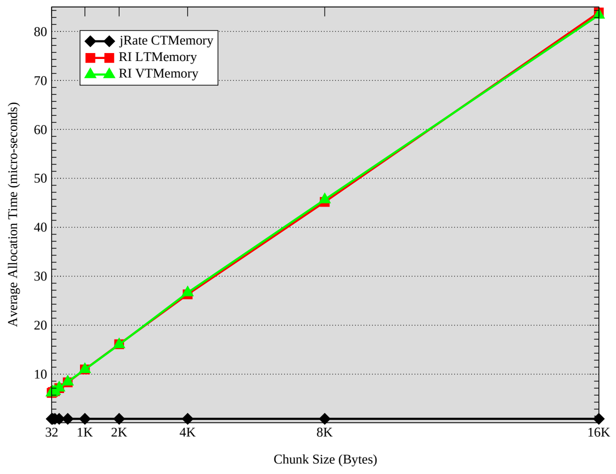

Figure 10. Average Allocation Time.

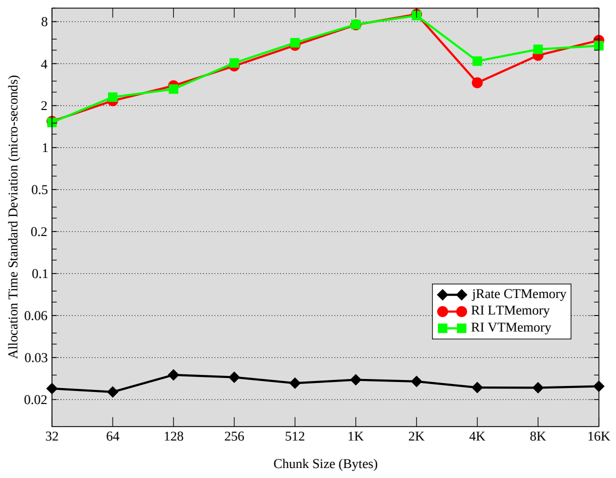

Figure 11. Allocation Time Standard Deviation.

Figure 10 shows the resulting average allocation time for the different test runs and Figure 11 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 10, both `LTMemory` and `VTMemory` provide linear time allocation with respect to the allocated memory size. Matching results were found for the other measured statistical parameter, based on this, we infer that the

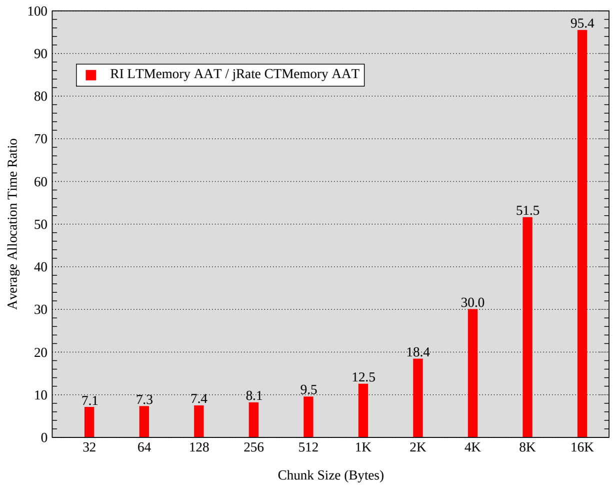

Figure 12. Speedup of the `CTMemory` Average Allocation Time Over the LTMemory Average Allocation Time.

RI implementation of `LTMemory` and `VTMemory` are similar, so we mostly focus on the `LTMemory` since our results also apply to `VTMemory`. jRate has an average allocation time that is independent of the allocated chunk, which helps analyze the timing of real-time Java 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 ten times faster than RI's `LTMemory`. For the biggest chunk we tested, jRate's `CTMemory` is ∼95 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 graphs 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 $AllocationTime \leq t$. This graph, along with Figure 11 that shows the standard deviation, provides insights on how the measure allocation time is

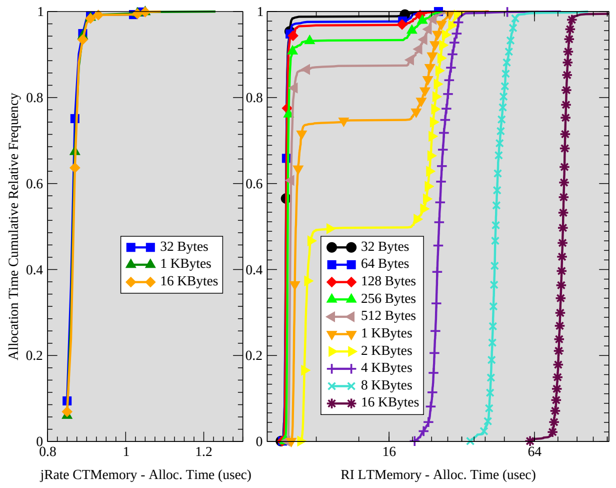

Figure 13. Allocation Time Cumulative Relative Frequency Distribution.

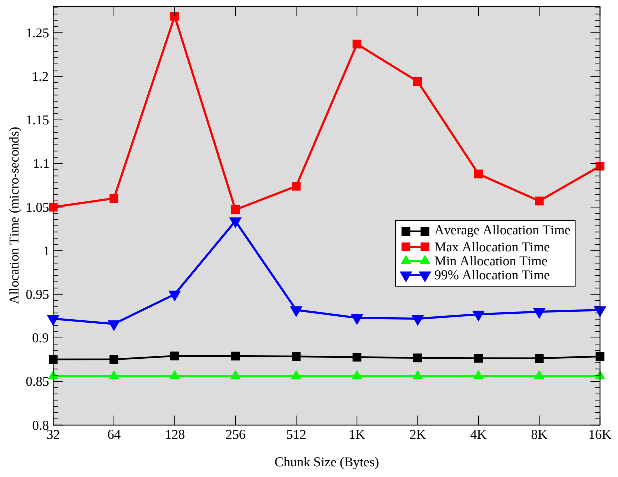

Figure 14. CTMemory Worst, Best, Average and 99% Allocation Time.

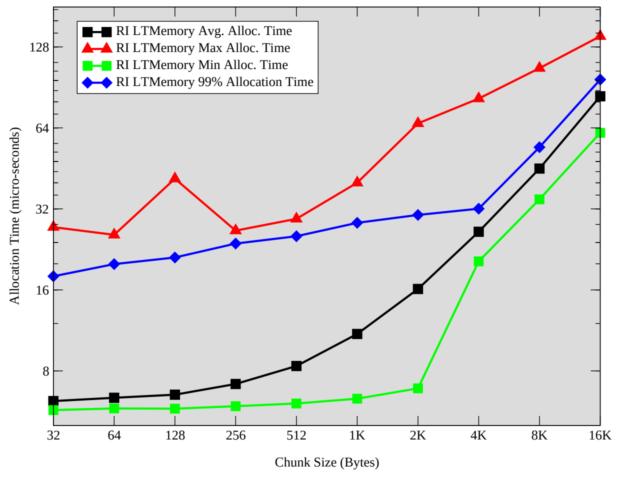

Figure 15. LTMemory Worst, Best, Average and 99% Allocation Time.

dispersed and distributed.

- **Worst-case Measures**—Figure 14 and Figure 15 show the bounds on the allocation time for jRate's `CTMemory` and the RI `LTMemory`. Each of these graphs depicts the worst, best, and average allocation times, along with the 99% upper bound of the allocation time. Figure 14 illustrates how the worst-case execution time for jRate's `CTMemory`is at most ∼1.4 times larger than its average execution time.

  Figure 15 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-case allocation time and the average-case is ∼1.6 for a chunk size of 16K. Figure 14, Figure 15 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.

3.3.2 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 2.2.2.

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 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, whereas regular threads were used for JDK 1.4 and CVM. To avoid GC overhead on platforms that do not support memory areas, we ensured the heap was large enough so that the GC would never be invoked. With Sun's JDK 1.4 JVM, either the option `-verbose:gc`, or the option `-Xloggc:<filename>` can be used to detect if the garbage collector is run. We used this option to set the value of the heap size to prevent the GC execution during the test.

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

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	Std. Dev.	Max	99%
CVM	2.905 μs	0.021 μs	3.181 μs	2.973 μs
JDK 1.4	3.743 μs	12.633 μs	402.02 μs	3.403 μs
jRate	1.301 μs	0.011 μs	1.338 μs	1.325 μs
RI	2.897 μs	0.019 μs	3.064 μs	2.974 μs
C++	1.30 μs	0.02 μs	N/A	N/A

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

- **Average Measures**—Table 1 shows how the RI and CVM perform fairly well in this test, *i.e.*, their context switch time is only ∼2 μs 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 [18]. 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, the RI, and CVM exhibit tight dispersion indexes, indicating that context switch overhead is predictable for these implementations. In general, the context switch time for jRate, the RI and CVM is as predictable as C++ on our Linux testbed platform. Conversely, JDK 1.4 exhibits less predictability, *i.e.*, due to the fact that it is not designed to have real-time behavior.
- **Worst-case Measures**—The fourth and fifth column of Table 1, represent respectively the maximum and the 99% bound for the context switch time. jRate, the RI, and CVM have 99% bound and worst-case context switch that is close to their average values. The JDK 1.4 worst-case context switch time is very high, though its 99% bound is fairly good, *i.e.*, JDK 1.4 has fairly good context switch time most of the time, but not all the time.

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.

Test Results. Table 2 shows the aggregate context switch time[9] average and standard deviation for the different Java implementations we tested.

	Average	Std. Dev.	Max	99%
CVM	12.945 μs	3.834 μs	19.278 μs	16.545 μs
JDK 1.4	25.998 μs	19.556 μs	400.140 μs	31.669 μs
jRate	7.231 μs	0.029 μs	7.683 μs	7.297μs
RI	10.379 μs	0.017 μs	10.537 μs	10.404 μs
C++	7.478 μs	0.012 μs	N/A	N/A

Table 2. Aggregate Synchronized Context Switch Switch Average, Standard Deviation, Max and 99% Bound.

Table 3 and Table 4 report the average, standard deviation, maximum and 99% bound of the time taken to respectively enter and exit a monitor. Table 5 depicts the extrapolated average and worst case of the net context switch, which is obtained by subtracting the minimum monitor enter and exit times from the aggregate synchronized context switch time.

	Average	Std. Dev.	Max	99%
CVM	1.385 μs	0.017 μs	1.637 μs	1.468 μs
JDK 1.4	1.024 μs	0.208 μs	5.352 μs	1.218 μs
jRate	0.284 μs	0.005 μs	0.353 μs	0.290 μs
RI	1.864 μs	0.022 μs	2.062 μs	1.932 μs

Table 3. Monitor Enter time Average, Standard Deviation, Max and 99% Bound.

	Average	Std. Dev.	Max	99%
CVM	1.294 μs	0.019 μs	1.435 μs	1.352 μs
JDK 1.4	1.632 μs	16.858 μs	377.22 μs	0.972 μs
jRate	0.254 μs	0.006 μs	0.329 μs	0.263 μs
RI	1.723 μs	0.386 μs	13.881 μs	1.748 μs

Table 4. Monitor Exit time Average, Standard Deviation, Max and 99% Bound.

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

- **Average Measures**—Table 2 illustrates that although jRate has the best aggregate context switch, the RI actually has the best maximum net context switch, with jRate close behind. Moreover, the RI context switch time is close to the C++ context switch time.
- **Dispersion Measures**—The third column of Table 2 depicts the standard deviation for the aggregate and net context switch. All the real-time Java platforms have low dispersion, which indicates a predictable context switch time. CVM also behaves quite well, while the JDK 1.4 dispersion is substantially higher than the real-time Java dispersions.
- **Worst-case Measures**—Both jRate and the RI have maximum and 99% bound values that are close to the

[9]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.

	Average	Max
CVM	10.266 μs	16.654
JDK 1.4	23.342 μs	398.60 μs
jRate	6.693 μs	7.151 μs
RI	6.792 μs	7.137 μs
C++	6.917 μs	N/A

Table 5. Net Synchronized Context Switch Average and Maximum.

average. CVM exhibits good worst-case behavior as well, though not as good as the real-time Java implementations. In contrast, JDK 1.4 has an upper bound for the synchronized context switch test that is far off its average value.

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 500 of these measurements were made. We just ran this test on the RI since only it supports this feature. Although jRate is based on the RTSJ it does not yet support periodic threads.

Test Results. Table 6 shows average and dispersion values that we measured for this test.

	Avg.	Std. Dev.	Max	99%
T=1 ms	0.933 ms	0.147 ms	0.959 ms	0.959 ms
T=5 ms	3.949 ms	7.890 ms	19.756 ms	19.739 ms
T=10 ms	9.955 ms	9.527 ms	19.480 ms	19.478 ms
T=30 ms	29.955 ms	0.004 ms	29.963 ms	29.960 ms
T=50 ms	49.955 ms	0.004 ms	49.960 ms	49.959 ms
T=100 ms	99.955 ms	0.004 ms	99.965 ms	99.960 ms
T=300 ms	299.95 ms	0.004 ms	299.96 ms	299.96 ms
T=500 ms	499.96 ms	0.004 ms	499.96 ms	499.96 ms

Table 6. Periodic Thread Period Average, Standard Deviation, Max and 99% Bound.

Results Analysis. Below we analyze the results of the test that measure accuracy with which periodic thread's logic are activated:

- **Average Measures**—Table 6 shows that for periods $>$ 10 ms, the average actual period is close to the nominal period, which is represented by the values in the first column. For periods $<$ 10 ms, however, the actual value is not always close to the desired or nominal value. 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. Our analysis indicates, however, that the behavior observed for periods $\leq$ 10 ms does not result from the use of the `nanosleep()` system call. This observation is based on the output of `ptrace` (described in Section 3.3.4), which indicated that the RI timer implementation uses `nanosleep()`. Similar traces obtained for the tests reported in Tables 13 and 14 indicated that for time $\leq$ 10 ms the timer results are not consistent with the periodic thread results. Based on these results, we conclude that the implementation of periodic threads uses a different mechanism than `nanosleep()`.

- **Dispersion Measures**—The third column of Table 6 shows that for a period $>=$ 30 ms, the actual period with which the thread logic is activated is close to the nominal value and is quite predictable, *i.e.*, it has low jitter. In contrast, for periods of 5 and 10 ms the mean is close to the nominal value and the measured values are highly dispersed. Based on the results shown in Table 6, the RI behaves unpredictably for periods $<=$ to 10 ms.

- **Worst-case Measures**—The forth and fifth columns of Table 6 show the maximum period experienced and the 99% bound on the period experienced by the real-time thread. The worst-case behavior is bad, however, only in the case of T=5 ms and T=10 ms. In other cases, the worst-case behavior is close to the average-case behavior and provides a predictable and regular period.

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 2.2.2, this test exists into two variants. We refer to the test that creates and starts a real-time thread from another real-time thread as the *RT test*, as opposed to the test that creates a real-time thread (or regular for platform that lack real-time threads) from a regular Java thread. The results we obtained are presented and analyzed below.

Test Settings. For each java platform in our test suite, we collected 400 samples of the thread creation time and thread start time. Real-time threads were used for the RI and jRate, whereas regular threads were used for JDK 1.4 and CVM. To avoid garbage collection overhead we ensured the heap was large enough so that the garbage collector would not be invoked since we are interested in measuring the time taken to create and start a thread, while limiting the effects of other delays.

Test Results. Table 7 and Table 8 report the average, standard deviation, maximum and 99% bound for the thread creation time and thread start time respectively. Figure 16 presents the aggregate average, max and 99% bound of the thread creation and startup time.

	Average	Std. Dev.	Max	99%
CVM	70.222 μs	4.233 μs	100.680 μs	92.463 μs
JDK 1.4	175.250 μs	92.631 μs	921.970 μs	303.930 μs
jRate	146.780 μs	8.237 μs	194.410 μs	180.540 μs
jRate RT	18.808 μs	6.958 μs	55.270 μs	42.136 μs
RI	234.750 μs	16.977 μs	509.880 μs	276.540 μs
RI RT	412.380 μs	89.851 μs	582.240 μs	571.710 μs

Table 7. Thread Creation Time Statistical Indexes.

	Average	Std. Dev.	Max	99%
CVM	460.670 μs	12.678 μs	528.320 μs	496.669 μs
JDK 1.4	390.450 μs	54.232 μs	771.940 μs	520.720 μs
jRate	70.308 μs	6.019 μs	97.906 μs	90.389 μs
jRate RT	66.012 μs	5.755 μs	99.075 μs	92.627 μs
RI	587.240 μs	38.866 μs	694.030 μs	668.450 μs
RI RT	601.940 μs	13.770 μs	647.570 μs	640.810 μs

Table 8. Thread Startup Time Statistical Indexes.

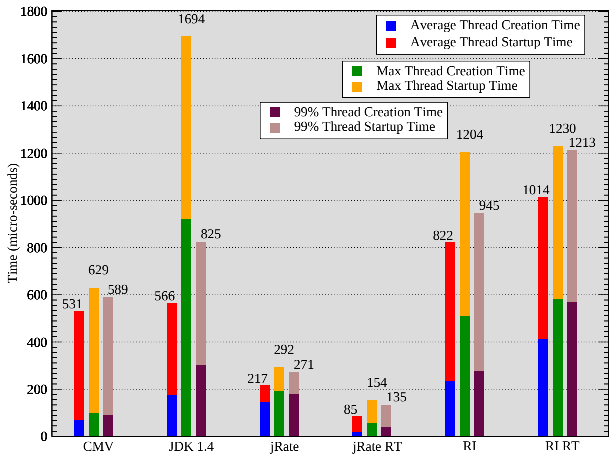

Figure 16. Aggregate Thread Creation/Startup.

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 first column of Table 7 and Table 8 show that jRate has the best average thread creation time and thread start time, and both value are obtained in the RT test case, where a real-time thread creates and starts other real-time threads. Conversely, for the other case, *i.e.*, where a non-RT thread creates a real-time thread, CVM provides the best average creation time. The RI has the largest values in both cases, and the JDK 1.4 values are usually close to the RI and to CVM. The RI's creation time grows linearly in the RT test, which is unusual and may reveal a problem in the RI management of real-time thread resources.

  Both the JDK 1.4 and the RI exhibit similar average performance, though the RI is fully interpreted whereas the JDK 1.4 is JIT-compiled. CVM has the second best average values. The difference in the thread creation time and thread start time experienced on the various Java platforms may stem from different implementations dividing the amount of work that is performed differently between the two components.

- **Dispersion Measures**—The third column of Table 7 and Table 8 present the standard deviation for thread creation time and thread start time, respectively. CVM has the smallest dispersal of values for the thread creation time, whereas jRate has the smallest dispersal of values for the thread startup time. The RI has always smaller standard deviation than the JDK 1.4.

  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 forth and fifth column of Table 7 and Table 8 present the maximum and the 99% bound for the thread creation time, and the thread startup time, respectively. These tables and Figure 16 clearly show how jRate, CVM, and the RI have worst case behavior that are much closer to the average and the 99% bound than JDK 1.4.

3.3.3 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 2.2.3.

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 2.2.3 with a fire count of 2,000 for both RI 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 GC while performing the test, the real-time thread that fires and handles the event uses scoped memory as its current memory area.

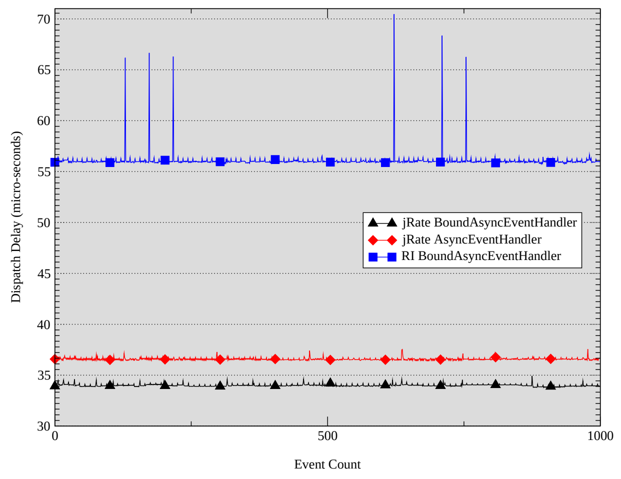

Figure 17. Dispatch Latency Trend for Successive Event Firing.

Test Results. Figure 17 shows the trend of the dispatch latency for successive event firings.[10] The data obtained by running the dispatch delay tests were processed to obtain average worst-case and dispersion measure of the dispatch latency. Table 9 and Table 10 shows the results found for jRate and the RI respectively.

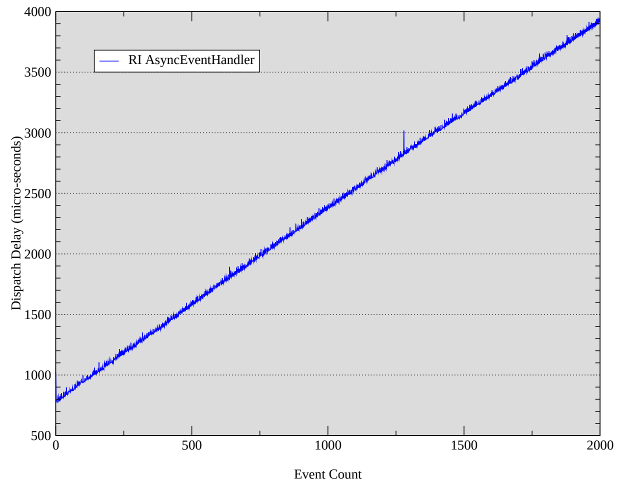

Figure 18. `AsyncEventHandler` Dispatch Latency Trend.

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:

[10]Since The RI's `AsyncEventHandler` trend is completely off the scale, it is omitted in this figure and depicted separately in Figure 18.

	AsycnEventHandler	BoundAsycnEventHandler
Avg.	36.574 μs	34.004 μs
Std. Dev.	0.113 μs	0.148 μs
Max	39.400 μs	35.555 μs
99%	36.945 μs	34.472 μs

Table 9. jRate Event Handler's Dispatch Latency Average, Standard Dev., Max and 99% bound for the Different Settings

	AsycnEventHandler	BoundAsycnEventHandler
Avg.	2373.0 μs	56.100 μs
Std. Dev.	909.92 μs	0.848 μs
Max	3950.8 μs	70.462 μs
99%	3892.5 μs	56.692 μs

Table 10. RI Event Handler's Dispatch Latency Average, Standard Dev., Max and 99% bound for the Different Settings

- **Average Measures**—Table 10 illustrates the large average dispatch latency incurred by the RTSJ RI `AsyncEventHandler`. The results in Figure 18 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 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 17 and Table 9 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* [13] 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 10, Table 9, Figure 17, and Figure 19 illustrate how jRate's `BoundAsyncEventHandler` dispatch latency incurs the least jitter. The dispatch latency value dispersion for the RTSJ RI `BoundAsyncEven-`

`tHandler` 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.

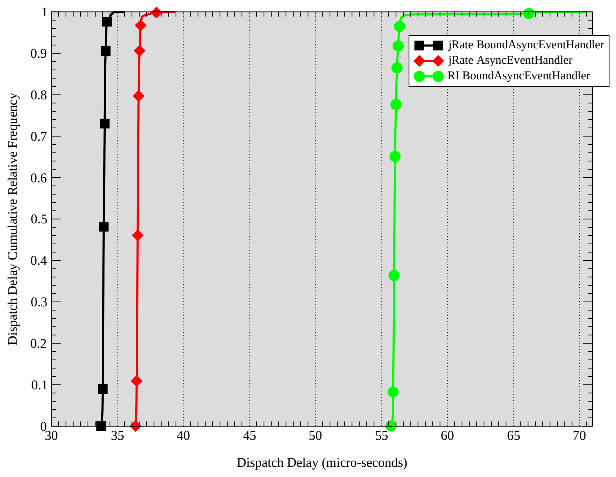

Figure 19. Cumulative Dispatch Latency Distribution.

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 [16].

- **Worst-case Measures**—Table 9 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 provided by the RI's `BoundAsyncEventHandler` is not as good 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 (clearly we do not consider the RI's `AsyncEventHandler` since no bound can be put on its behavior).

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 18), which masks any priority inversions. Moreover, 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. Table 11 and Table 12 report how the average, standard deviation, maximum and 99% bound of the dispatch delay changes for H as the number of low-priority handlers increase. Figure 20 and Figure 21 provide a graphical representation for the average and dispersion measures.

	Avg.	Std. Dev.	Max	99%
0 LP	33.375 μs	0.124 μs	34.877 μs	34.116 μs
10 LP	33.154 μs	0.134 μs	34.903 μs	33.797 μs
50 LP	33.205 μs	0.161 μs	36.063 μs	33.825 μs
100 LP	33.264 μs	0.147 μs	35.959 μs	33.851 μs
500 LP	33.632 μs	0.180 μs	37.149 μs	34.283 μs
1000 LP	33.739 μs	0.199 μs	37.565 μs	34.458 μs

Table 11. jRate**'s Average, Standard Deviation, Maximum and 99% bound of the Dispatch Delay.**

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 20 and Tables 11 and 12 illustrate that the average dispatch latency experienced by H is essentially constant for jRate, regardless of the number of low-priority handlers. It grows rapidly,

	Avg.	Std. Dev.	Max	99%
0 LP	56.106 μs	0.887 μs	70.462 μs	56.706 μs
10 LP	112.33 μs	1.346 μs	133.90 μs	122.18 μs
50 LP	332.41 μs	2.396 μs	353.17 μs	344.86 μs
100 LP	609.92 μs	3.410 μs	631.51 μs	624.96 μs
500 LP	2826.4 μs	12.005 μs	2884.0 μs	2862.1 μs
1000 LP	5587.0 μs	23.768 μs	5672.7 μs	5650.3 μs

Table 12. RI's Average, Standard Deviation, Maximum and 99% bound of the Dispatch Delay.

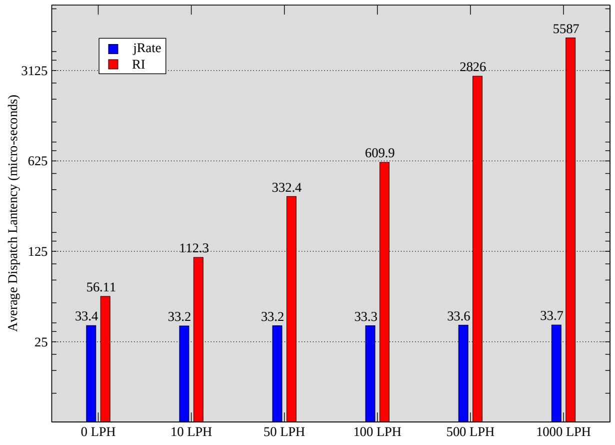

Figure 20. H**'s Average Dispatch Latency.**

however, as the number of low-priority handlers increase for the RI. The RI's event dispatching priority inversion is problematic for real-time systems 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. This approach is an application of the formalisms presented in [5].

- **Dispersion Measures**—Figure 21 and Tables 11 and 12 illustrate 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**—Tables 11 and 12 illustrate 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.

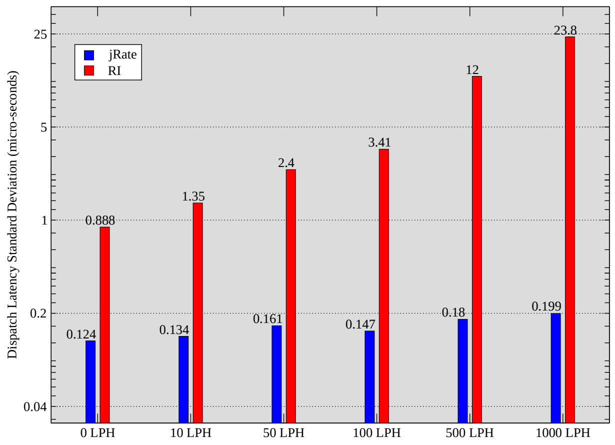

Figure 21. H **Dispatch Latency's Standard Deviation.**

3.3.4 Timer Benchmark Results

Below, we present and analyze the results from the one shot and periodic timer tests, which were described in Section 2.2.4.

One Shot Timer Test. This test measures how precisely a `OneShotTimer` can fire events, relative to the time interval for which it was programmed.

Test Settings. The test ran a `OneShotTimer` that generated an event for time intervals ranging from 5 ms to 500 ms. The event was handled by a `BoundAsyncEventHandler` that was registered as the timer timeout handler. For each timeout interval we collected 500 samples. The time interval was specified by using `RelativeTime`. We also tried using `AbsoluteTime`, but it behaved so similarly that we only present the `RelativeTime` results for brevity. We just ran this test on the RI since only it supports this feature. Although jRate is based on the RTSJ it does not yet support timers.

Test Results. Table 13 shows the average and standard deviation for the actual timeout interval produced by the `OneShotTimer`.

Results Analysis. Below we analyze the results of the test that measures the accuracy with which `OneShotTimer` fire their events with respect to the requested interval:

- **Average Measures**—The second column of Table 13 reports the average value of the time interval after

	Avg.	Std. Dev.	Max	99%
T=5 ms	19.818 ms	0.07 ms	19.903 ms	19.830 ms
T=10 ms	19.820 ms	0.007 ms	19.901 ms	19.830 ms
T=30 ms	39.819 ms	0.013 ms	39.930 ms	39.828 ms
T=50 ms	59.814 ms	0.02 ms	59.925 ms	59.829 ms
T=100 ms	109.81 ms	0.07 ms	109.94 ms	109.83 ms
T=300 ms	309.81 ms	0.067 ms	309.93 ms	309.92 ms
T=500 ms	509.81 ms	0.036 ms	509.92 ms	509.92 ms

Table 13. Aperiodic Timer Results.

which the `OneShotTimer` generates an event, while the first column represents the desired time interval. As shown in Section 3.3.2's analysis of the periodic thread test results, the average firing interval performs poorly for time intervals < 10 ms. Conversely, for time interval > 10 ms, the results in Table 13 show a strange, yet consistent behavior, where the average time interval generated by the timer is exactly equal to the desired one plus 9.82 ms.

By inspecting the implementation of the RI timer, we found that the time interval after which the timer is fired is generated by having a thread associated with the timer that waits on a dummy Java object for the specified amount of time. The resolution of timer is therefore essentially the same as the one provided by the `Object.wait(long msec, int nsec)` method in the RI implementation. By using `ptrace` we traced the system call made by the RI when the `Object.wait(long msec, int nsec)` method in invoked. We found that `nanosleep()` is used to implement this method. The result shown on Table 13 are also consistent with the resolution provided by `nanosleep()` on the tested platform.

- **Dispersion Measures**—The third column of Table 13 shows the standard deviation of the measured time intervals generated by the firing of the timer. The standard deviation is small, which indicates that the generated interval is quite predictable. These results are inconsistent with the periodic thread results (see Table 6 in Section 3.3.2), where the standard deviation was quite large for periods $<= 10$ ms. This difference in the dispersion of the value for periods $<= 10$ ms is ascribable to the different mechanism by periodic threads and timers.

- **Worst-case Measures**—The forth and fifth columns of Table 14 show the maximum period experienced and the 99% bound on the period experienced by the real-time thread. Both of these values are close to the average behavior, which demonstrates good worst-case behavior.

Periodic Timer Test. This test measures how precisely a `PeriodicTimer` can fire events, relative to the period for which it was programmed.

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

Test Results. Table 14 shows the average-case, worst-cast, and the standard deviation for the actual period produced by the `PeriodicTimer`.

	Avg.	Std. Dev.	Max	99%
T=5 ms	19.953 ms	0.004 ms	19.968 ms	19.961 ms
T=10 ms	19.955 ms	0.010 ms	20.083 ms	19.962 ms
T=30 ms	39.954 ms	0.01 ms	40.068 ms	39.965 ms
T=50 ms	59.955 ms	0.010 ms	60.069 ms	59.963 ms
T=100 ms	109.95 ms	0.015 ms	110.08 ms	109.97 ms
T=300 ms	309.95 ms	0.024 ms	310.06 ms	310.06 ms
T=500 ms	509.95 ms	0.028 ms	510.06 ms	510.06 ms

Table 14. Periodic Timer Results.

Results Analysis. Below we analyze the results of the test that measures the accuracy with which a `PeriodicTimer` fires events with respect to the requested period:

- **Average Measures**—The second column of Table 14 reports the average value of the period at which the `PeriodicTimer` generate events, while the first column represents the desired period. For time intervals < 10 ms, the average firing interval is much worse than the desired one, which is consistent with the periodic thread test results in Section 3.3.2. In contrast, for time intervals > 10 ms, the results in Table 13 show a strange, yet consistent behavior, where the average time interval generated by the timer is exactly equal to the desired one plus 9.6 ms. This behavior is consistent with the `OneShotTimer` results.

  The implementation of periodic timers is the same as the one shot timers, so that same observation made earlier for one shot timers applies to periodic timers. The difference that we have found with respect to the periodic thread behavior relies on the different implementation. In fact, the `waitForNextPeriod()` implementation relies on a native method call to wait a period. Without access to the RI JVM source code, however, we cannot exactly say how it is implemented and why there is such a difference in the behavior of the two.

- **Dispersion Measures**—The third column of Table 14 shows the standard deviation of the measured time intervals generated by firing the timer. The standard deviation is small, which indicates the generated intervals are highly predictable. This result is also inconsistent with the periodic thread behavior shown in see Table 6 in Section 3.3.2, where the standard deviation was large when the period was $<= 10$ ms. Moreover, for most of the cases reported in Table 6, the standard deviation is slightly smaller than the corresponding `OneShotTimer` case shown in Table 13. This small difference can be ascribed to the fact that one shot timers must be restarted each time they fire.

- **Worst-case Measures**—The forth and fifth columns of Table 14 show the maximum period experienced and the 99% bound on the period experienced by the real-time thread. Both of these values are close to the average behavior, which demonstrates good worst-case behavior.

4 Related Work

Although the RTSJ was adopted fairly recently [2], there are already a number of research projects related to our work on jRate and RTJPerf. The following projects are particularly interesting:

- The **FLEX** [14] 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 [15] 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 [6] 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 [10] 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 [19].

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.

5 Concluding Remarks

This paper presented an empirical evaluation of the performance of RTSJ features 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.

This paper applies RTJPerf to measure the performance of the RTSJ RI, jRate, CVM, and JDK 1.4. 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 many aspects, though it has several problems with `AsyncEventHandler` dispatching delays and priority inversion. While CVM is not an RTSJ-compliant implementation, it performed well for many tests.

Our future work on RTJPerf is outlined below:

- Since the TimeSys RI does not provide a Just In Time (JIT) compiler, its not surprising that its overall performance is lower than the ahead-of-time compiled jRate approach. We look forward to measuring the performance of the TimeSys jTime product that will be released later this year, which should have much better performance.
- We also plan to run the RTJPerf benchmarks on the commercial TimeSys Linux/RT. This version provides many features that are lacking in the GPL version of the TimeSys Linux/RT, such as higher resolution timer, support for priority inversion control via priority inheritance and priority ceiling protocols, and resource reservation.
- 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 [19], 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 [10] 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.

Although jRate implements many core RTSJ features, the following omissions will be addressed in our future work:

- Add support for the remaining RTSJ features, such as timers, POSIX signal handling, periodic and no-heap real-time threads, and physical memory access. Some feature that we don't plan to implement in the first release of jRate is the memory reference checking, and the Asynchronous transfer control. Since jRate will be the primary Real-Time Java platform used by ZEN [12], jRate's implementation is being driven by the features that are most important for real-time ORBs.
- Provide a user-level scheduling framework that leverages the simple priority-based scheduling provided by the underlying real-time operating systems to provide advanced scheduling services.
- Completely partition 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.
- Provide 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 [23] and AspectC++ [22], we are decomposing jRate's into a series of aspects that can be configured by developers.

The first public version of jRate will be released by the end of June, 2002. Information on its current status and availability can be found at `http://tao.doc.wustl.edu/~corsaro/jRate`. Since jRate is an open-source project, we encourage researchers and developers to provide us feedback and help improve its quality and capabilities. jRate will use the same open-source model we use for ACE [17] and TAO [3], which has proved to be successful to produce high-quality open-source middleware.

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] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, Boston, 2000.

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

[3] Center for Distributed Object Computing. The ACE ORB (TAO). www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

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

[5] A. Corsaro, D. C. Schmidt, R. K. Cytron, and C. Gill. Formalizing Meta-Programming Techniques to Reconcile Heterogeneous Scheduling Disciplines in Open Distributed Real-Time Systems. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications.*, pages 289–299, Rome, Italy, September 2001. OMG.

[6] 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 T. A. Henzinger and C. M. Kirsch, editors, *Embedded Software: Proceedings of the First International Workshop*, pages 131–147. Springer Verlag, 2001.

[7] J. Gosling, B. Joy, and G. Steele. *The Java Programming Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[8] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, October 1997. ACM.

[9] IBM. Jikes 1.14. `http://www.research.ibm.com/jikes/`, 2001.

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

[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] R. Klefstad, D. C. Schmidt, and C. O'Ryan. The Design of a Real-time CORBA ORB using Real-time Java. In

Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing. IEEE, Apr. 2002.

[13] D. Lea. *Concurrent Java: Design Principles and Patterns, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1999.

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

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

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

[17] D. C. Schmidt. The ADAPTIVE Communication Environment (ACE). www.cs.wustl.edu/∼schmidt/ACE.html, 1997.

[18] D. C. Schmidt, M. Deshpande, and C. O'Ryan. Operating System Performance in Support of Real-time Middleware. In *Proceedings of the* 7^{th} *Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA, Jan. 2002. IEEE.

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

[20] Sun. The C Virtual Machine (CVM). `http://java.sun.com/products/cdc/cvm/`, 2001.

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

[22] The AspectC++ Organization. Aspect-Oriented Programming for C++. `www.aspectc.org`, 2001.

[23] The AspectJ Organization. Aspect-Oriented Programming for Java. `www.aspectj.org`, 2001.

[24] TimeSys. Real-Time Specification for Java Reference Implementation. `www.timesys.com/rtj`, 2001.

[25] TimeSys. TimeSys Linux/RT 3.0. `www.timesys.com`, 2001.