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

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 some of 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

Until recently, there was no implementation of the Real-Time Specification for Java (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 [11].

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 at UC Irvine. In this paper, 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; 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[2] (which is an ahead-of-time compiled implementation of RTSJ we are developing); and Section 4 presents concluding remarks.

2 Overview of RTJPerf

RTJPerf provide benchmarks for most of the RTSJ features that are critical to real-time embedded systems. A complete description of the tests currently available in RTJPerf can be found in [2]. Below, we describe a subset of these benchmark tests and reference where we present the results of the tests in subsequent sections of this paper. In addition to describing what RTSJ features RTJPerf measures, we summarize the key RTSJ features themselves.

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

[1] RTJPerf is freely available at `http://tao.doc.wustl.edu/~corsaro/periscope.html`.

[2] jRate is freely available at `http://tao.doc.wustl.edu/~corsaro/jRate`.

Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)
0-7695-1739-0/02 $17.00 © 2002 IEEE

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. 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 $C > 0$.[3]

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

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 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. There are two types of asynchronous event handlers defined in RTSJ:

- The `AsyncEventHandler` class, which does not have a thread permanently bound to it, nor is it guaranteed that there will be a separate thread for each `AsyncEventHandler`. The RTSJ simply requires that after an event is fired the execution of all its associated `AsyncEventHandlers` 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 [4], 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 3.3.2.

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 3.3.2, provides an analysis of the implementation of the current RI and presents an implementation that overcomes some shortcomings of the RI.

2.3 Threads

The RTSJ extends the Java threading model with two new types of real-time threads: `RealtimeThread` and `NoHeapRealtimeThread`.

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

Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)
0-7695-1739-0/02 $17.00 © 2002 IEEE

Since the NoHeapRealtimeThread can have execution eligibility higher than the garbage collector [4], it cannot allocate nor reference any heap objects. The scheduler controls the *execution eligibility* [3] [5] of the instances of this class 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 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 3.3.3, while the reader interested in the results for the second type of test is remanded to [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 3.3.3.

[4] The RTSJ v1.0 specification states that the NoHeapRealtimeThread have always execution eligibility higher than the Garbage Collector (GC), but this has been changed in the v1.01

[5] Execution eligibility is defined as the position of a schedulable entity in a total ordering established by a scheduler over the available entities [3]. 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.

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. The RTSJ provides two types of timers:

- 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. 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. Performances results for these tests are reported in Section 3.3.4. In [2] results for periodic timers are presented as well.

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 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

Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)
0-7695-1739-0/02 $17.00 © 2002 IEEE

Linux RedHat 7.2 with the TimeSys Linux/RT 3.0 GPL[6] kernel [12]. 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) [11], which is a fully compliant implementation of Java [1] that implements all the mandatory features in the RTSJ. The RI is based on a Java 2 Micro Edition (J2ME) Java Virtual Machine (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 [12], *i.e.*, the commercial version.

Figure 1b 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 the GNU Compiler for Java (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) [6] 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 1a is different than the JVM model depicted in Figure 1b 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

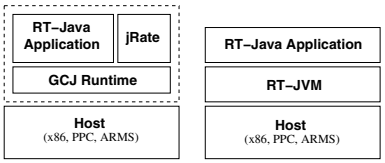

Figure 1. The jRate and RI Architectures.

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 [10] is a J2ME platform targeted for embedded and consumer electronic devices. CVM has relatively small footprint and is designed 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** [5] 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 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.

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

[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.

Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)
0-7695-1739-0/02 $17.00 © 2002 IEEE

3.3.1 Memory Benchmark Results

Below, we present and analyze the results of the allocation time test that was described in Section 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. jRate's `CTMemory` implements an RTSJ scoped memory such as the `LTMemory` or the `VTMemory`.

This test only examines jRate and 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]

Figure 2 shows the resulting average allocation time for the different test runs and Figure 3 shows the standard deviation of the allocation time measured in the various test settings. Figure 4 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 2, 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 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 4 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 3. 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 5 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 3 that shows the standard deviation, provides insights on how the measured allocation time is dispersed and distributed.
- **Worst-case Measures**—Figure 6 and Figure 7 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 6 illustrates how the worst-case execution time for jRate's `CTMemory`is at most ∼1.4 times larger than its average execution time.

  Figure 7 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 6, Figure 7 and Figure 5 also characterize the distribution of the allocation time. Figure 5 shows how for some allocation sizes, the allocation time for the RI `LTMemory` is centered around two points.

3.3.2 Asynchrony Benchmark Results

Below we present and analyze the results of the asynchronous event handler dispatch delay and asynchronous

[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.

Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)
0-7695-1739-0/02 $17.00 © 2002 IEEE

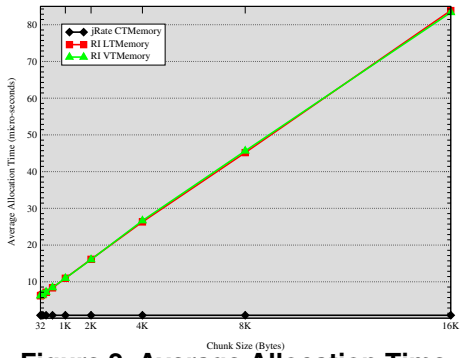

Figure 2. Average Allocation Time

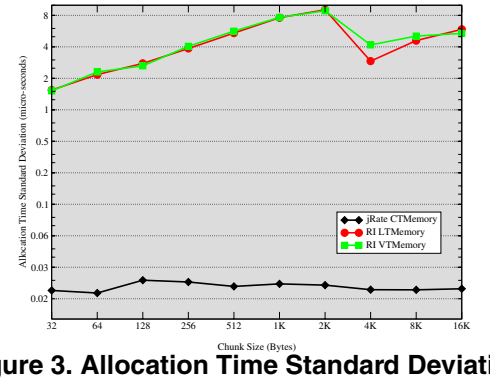

Figure 3. Allocation Time Standard Deviation

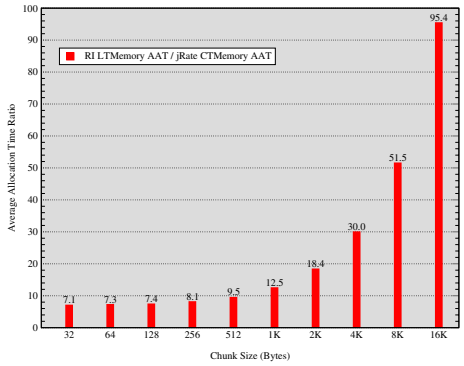

Figure 4. `CTMemory` AAT vs. LTMemory AAT Speedup

	AsycnEventHandler	BoundAsycnEventHandler
Avg.	36.57 μs	34.00 μs
Std. Dev.	0.11 μs	0.14 μs
Max	39.40 μs	35.55 μs
99%	36.94 μs	34.47 μs

Table 1. jRate Event Handler's Dispatch Latency Statistics Indexes

	AsycnEventHandler	BoundAsycnEventHandler
Avg.	2373.0 μs	56.10 μs
Std. Dev.	909.9 μs	0.84 μs
Max	3950.8 μs	70.46 μs
99%	3892.5 μs	56.69 μs

Table 2. RI Event Handler's Dispatch Latency Statistical Indexes

event handler priority inversion tests, which were described in Section 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 2.2 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.

Test Results. Figure 8 shows the trend of the dispatch latency for successive event firings.[9] The data obtained by running the dispatch delay tests were processed to obtain average worst-case and dispersion measure of the dispatch latency. Table 1 and Table 2 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 2 illustrates the large average dispatch latency incurred by the RTSJ RI `AsyncEventHandler`. The results in Figure 9 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

[9] Since The RI's `AsyncEventHandler` trend is completely off the scale, it is omitted in this figure and depicted separately in Figure 9.

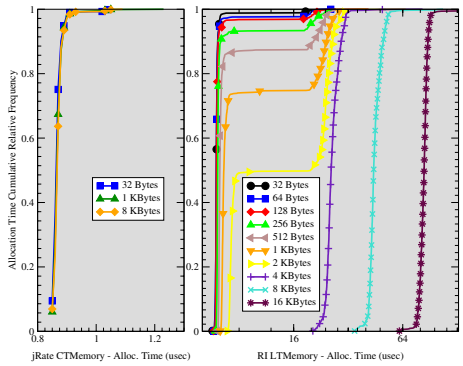

Figure 5. Allocation Time Cumulative Relative Frequency Distribution

Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)
0-7695-1739-0/02 $17.00 © 2002 **IEEE**

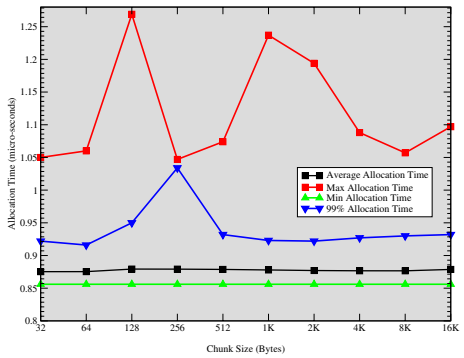

Figure 6. CTMemory's Allocation Time Statistical Indexes

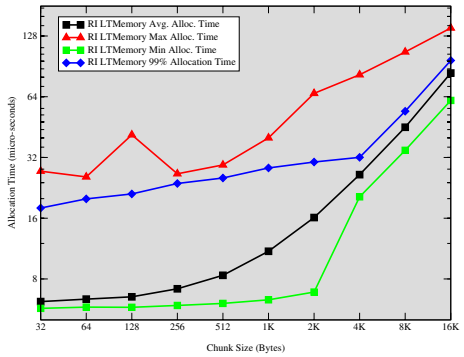

Figure 7. LTMemory's Allocation Time Statistical Indexes

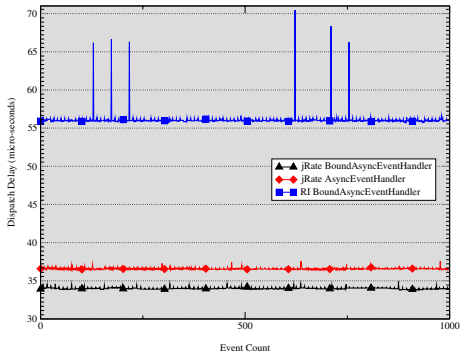

Figure 8. Dispatch Latency Trend for Successive Event Firing

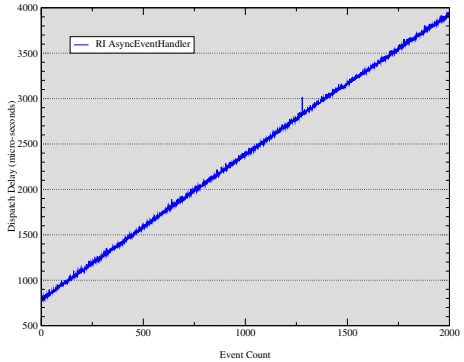

Figure 9. `AsyncEventHandler` Dispatch Latency Trend

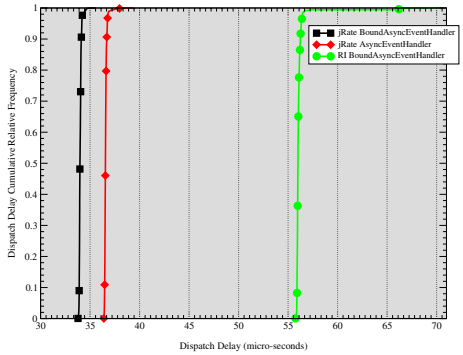

Figure 10. Cumulative Dispatch Latency Distribution

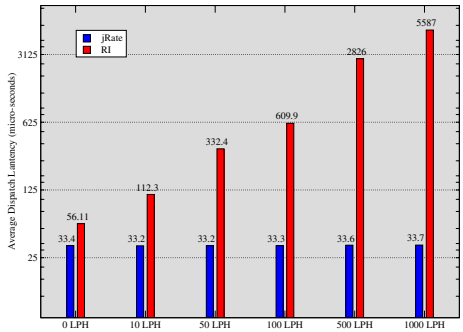

Figure 11. H's Average Dispatch Latency

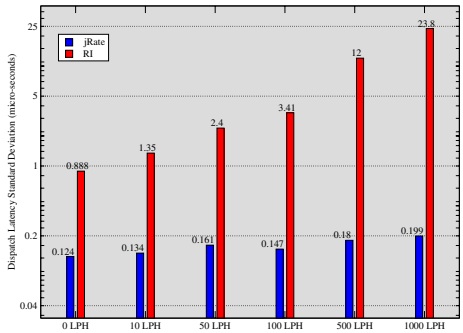

Figure 12. H Dispatch Latency's Standard Deviation

	Avg.	Std. Dev.	Max	99%
0 LP	33.37 μs	0.12 μs	34.87 μs	34.11 μs
10 LP	33.15 μs	0.13 μs	34.90 μs	33.79 μs
50 LP	33.20 μs	0.16 μs	36.06 μs	33.82 μs
100 LP	33.26 μs	0.14 μs	35.95 μs	33.85 μs
500 LP	33.63 μs	0.18 μs	37.14 μs	34.28 μs
1000 LP	33.73 μs	0.19 μs	37.56 μs	34.45 μs

Table 3. jRate's Dispatch Delay Statistical Indexes

	Avg.	Std. Dev.	Max	99%
0 LP	56.10 μs	0.88 μs	70.46 μs	56.70 μs
10 LP	112.33 μs	1.34 μs	133.90 μs	122.18 μs
50 LP	332.41 μs	2.39 μs	353.17 μs	344.86 μs
100 LP	609.92 μs	3.41 μs	631.51 μs	624.96 μs
500 LP	2826.40 μs	12.00 μs	2884.00 μs	2862.10 μs
1000 LP	5587.00 μs	23.76 μs	5672.70 μs	5650.30 μs

Table 4. RI's Dispatch Delay Statistical Indexes

Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)
0-7695-1739-0/02 $17.00 © 2002 **IEEE**

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 8 and Table 1 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* [7] 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 2, Table 1, Figure 8, and Figure 10 illustrate how jRate's `BoundAsyncEventHandler` dispatch latency incurs the least jitter. The dispatch latency value dispersion 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 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 [8].
- **Worst-case Measures**—Table 1 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 9), 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 3 and Table 4 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 11 and Figure 12 provide a graphical representation for the average and dispersion measures.

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 11 and Tables 3 and 4 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,

Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)
0-7695-1739-0/02 $17.00 © 2002 IEEE

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 [3].

- **Dispersion Measures**—Figure 12 and Tables 3 and 4 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 3 and 4 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.

3.3.3 Thread Benchmark Results

Below, we present and analyze the results from the yield and context switch test and the periodic thread test which were described in Section 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 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 5 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 Measures**—Table 5 shows how the RI and CVM perform fairly well in this test, *i.e.*, their context switch time is only $\sim$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 5 reports the results of a C++-based context switch test described in [9]. 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 5 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 5, 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.

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.

Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)
0-7695-1739-0/02 $17.00 © 2002 IEEE

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()`.

- **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.

	Average	Std. Dev.	Max	99%
CVM	2.90 μs	0.02 μs	3.18 μs	2.97 μs
JDK 1.4	3.74 μs	12.63 μs	402.02 μs	3.40 μs
jRate	1.30 μs	0.01 μs	1.33 μs	1.32 μs
RI	2.89 μs	0.01 μs	3.06 μs	2.97 μs
C++	1.30 μs	0.02 μs	N/A	N/A

Table 5. Yield Context Switch Statistical Indexes

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 2.4.

	Avg.	Std. Dev.	Max	99%
T=1 ms	0.93 ms	0.147 ms	0.95 ms	0.95 ms
T=5 ms	3.94 ms	7.890 ms	19.75 ms	19.73 ms
T=10 ms	9.95 ms	9.527 ms	19.48 ms	19.47 ms
T=30 ms	29.95 ms	0.004 ms	29.96 ms	29.96 ms
T=50 ms	49.95 ms	0.004 ms	49.96 ms	49.95 ms
T=100 ms	99.95 ms	0.004 ms	99.96 ms	99.96 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 Statistical Indexes

	Avg.	Std. Dev.	Max	99%
T=5 ms	19.81 ms	0.07 ms	19.90 ms	19.83 ms
T=10 ms	19.82 ms	0.00 ms	19.90 ms	19.83 ms
T=30 ms	39.81 ms	0.01 ms	39.93 ms	39.82 ms
T=50 ms	59.81 ms	0.02 ms	59.92 ms	59.82 ms
T=100 ms	109.81 ms	0.07 ms	109.94 ms	109.83 ms
T=300 ms	309.81 ms	0.06 ms	309.93 ms	309.92 ms
T=500 ms	509.81 ms	0.03 ms	509.92 ms	509.92 ms

Table 7. Aperiodic Timer Results

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 7 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 7 reports the average value of the time interval after which the `OneShotTimer` generates an event, while the first column represents the desired time interval. As shown in Section 3.3.3'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 7 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

Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)
0-7695-1739-0/02 $17.00 © 2002 IEEE

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 7 are also consistent with the resolution provided by `nanosleep()` on the tested platform.

- **Dispersion Measures**—The third column of Table 7 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 3.3.3), 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 7 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 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.

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

[3] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

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

Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)
0-7695-1739-0/02 $17.00 © 2002 **IEEE**