Evaluating Real-Time Java for Mission-Critical Large-Scale Embedded Systems

David C. Sharp, Edward Pla, & Kenn R. Luecke
The Boeing Company, Saint Louis, Missouri, USA
{david.sharp,edward.pla,kenn.r.luecke}@boeing.com

Ricardo J. Hassan II
Jet Propulsion Laboratory,Pasadena, California, USA
ricardo.hassan@jpl.nasa.gov

Abstract

Many of the benefits of Java, including its portability, networking support, and simplicity, are of increasing importance to large-scale distributed real-time embedded (DRE) systems, but have been unavailable due to the lack of acceptable real-time performance. Recent work establishing the Real-Time Specification for Java (RTSJ) [1] has led to the emergence of Real-Time Java Virtual Machines (RT JVMs) that promise to bridge this gap. This paper describes benchmarking results on an RT JVM. This paper extends previously published results [2] by including additional tests, by being run on a recently available pre-release version of the first commercially supported RTSJ implementation, and by assessing results based on our experience with avionics systems in other languages.

1. Introduction

The Boeing Company is currently experimenting with Real-time (RT) Java as part of the Air Force Research Laboratory (AFRL) RT Java for Embedded System (RTJES) program [1]. This program investigates the use of Java in hard and soft large-scale distributed real-time embedded (DRE) avionic system applications. The program has two primary objectives: benchmarking RT Java implementations to assess their suitability for this domain, and demonstrating the operational benefit of RT Java features for network-centric applications. This paper describes results of a portion of the network-centric benchmarking effort on a pre-release version of the commercial JTime® RT JVM from TimeSys that implements the RTSJ.

[1] This work was sponsored by the Air Force Research Laboratory, Wright-Patterson Air Force Base, Information Directorate, under contract F33615-97-D-1155-0008.

The RTSJ defines a set of classes which provide capabilities supporting real-time operation in a Java environment, including threading, scheduling, event handling, synchronization, and memory management. Specially constructed RT JVMs support the real-time semantics defined in the class library specification.

Our benchmarking efforts focus on assessing the performance and determinism of systems using these RT JVM features via two sets of tests. The first set of tests (which are discussed in this paper) assesses the characteristics of individual RTSJ features. The second set of tests (which are not discussed in this paper) investigates performance within an environment that is representative of an actual avionics application, based on our experience with reusable component-based avionics systems on the Boeing Bold Stroke initiative [3]. We plan to publish these latter test results when complete.

The remainder of this paper is organized as follows. Section 2 describes the experimental system configuration. Section 3 describes the low-level RTSJ benchmarking results. Concluding remarks and acknowledgements follow in Sections 4 and 5, respectively.

2. Experimentation System

This section describes the configuration of both the hardware and software test platform.

2.1. Hardware System

A Dell GX 150 computer was used for Java benchmark development and execution. This computer has a 1.2 Gigahertz Pentium 4 single processor. It has a 12 GB hard drive, 256 MB of RAM, 256 KB of cache memory, and 900 MB of swap memory.

2.2. Software System

As of this writing, the only known commercial implementation of the RTSJ is from TimeSys, for which we received a pre-release version. Prior to availability of this product, we developed tests and measured results on the openly available Reference Implementation (RI), also from TimeSys. The RI was designed to investigate and demonstrate the semantics of the RTSJ, not for production-quality run-time performance. Prior RI benchmarking confirmed this [2], but these results are not included here due to space constraints.

The test platform was configured with Red Hat Linux version 7.2, with real-time support provided by TimeSys Linux/NET for X86 UNI platform operating system extensions, version 3.1.214c, and TimeSys RT JVM version 3.5.3.

The JVM was executed with a memory allocation pool of 50 MB (-Xms50M). The immortal memory size was set to 80 MB (IMMORTAL_SIZE=80000000).

The tests were developed with the Jakarta Ant version 1.4.1 build tool with javac from the Java Development Kit (JDK) version 1.2.2. This javac version was selected due to version compatibility issues in libraries used in the avionics application test set. No just in time or ahead of time compilation was performed for these tests.

3. RTSJ Testing

These tests focus on assessing the performance of specific RTSJ capabilities. These tests are being added to the RTJPerf open source RT JVM benchmarking suite established by Angelo Corsaro at the University of California, Irvine (UCI) [4] and Washington University in St. Louis. Taken together, tests were created to assess determinism, latency, and throughput associated with threads, scheduling, memory management, synchronization, time, timers, asynchrony, exceptions, and class loader and dynamic linking. Only the tests deemed of most importance are included here due to space constraints.

For each test, a description of the test is included, along with success criteria, and experimental results and analysis. The success criteria are based on requirements and experiences with avionic mission computing systems. While these criteria are intentionally domain specific, they do capture expectations for an important category of embedded systems. In all cases, the raw measured values are provided for comparison against criteria in other domains.

3.1. Throughput

The RTSJ introduces two new types of threads as shown in Figure 1: RT threads and No Heap RT (NHRT) threads. RT threads support, at a minimum, basic real-time preemptive scheduling. No Heap RT threads add the guarantee that execution will be independent of garbage collection but with the additional restriction that heap-based memory not be used. This section outlines tests assessing the throughput of these different thread types in varying execution environments.

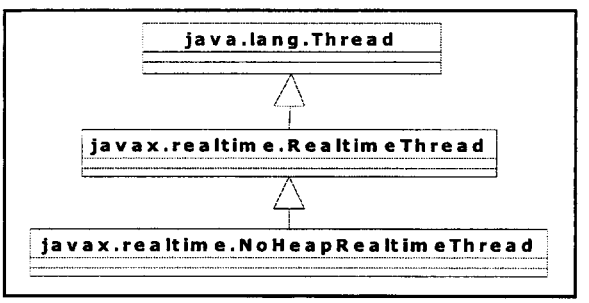

Figure 1. Thread Inheritance in RT Java

3.1.1 Thread Throughput

Description: Record the execution time of a computationally intensive algorithm, representative of avionics mission computing processing, when run in different thread types: NoHeapRealtimeThread (NHRT), RealtimeThread (RT), and java.lang.Thread. All three thread types will be processing in a 20 Hz frame. In the thread's `run()` method, log timestamps before and after the algorithm executes. This computationally intensive algorithm is a flight controls algorithm that is CPU intensive and reads data from two different input files. The flight controls algorithm performs all of its memory allocation and reference storage upon initiation. The NoHeapRealtimeThreads were executed using Immortal memory which has no garbage collection while the RealtimeThreads were executed from heap memory.

Success Criteria: Throughput in different threads shall not vary by more than 1%

Results: The throughput for NHRT, RT, and normal java threads on the selected algorithm was comparable. The largest percentage time difference between the three thread types was 0.643% for the NHRT and RT threads. See Table 1 for more throughput comparisons between NHRT, RT, and normal threads.

Table 1. Algorithm Execution Times Within Different Thread Types (milliseconds)

	Time in NHRT Threads	Time in RT Threads	Time in Normal Threads	Time (% Difference)
Avg	3.5201	3.5087	3.5174	0.324%
Max	3.5372	3.5601	3.5439	0.643%
Min	3.5109	3.4977	3.5076	0.376%

3.1.2 Thread Throughput With Contending Background Threads

Description: Record the execution time of the same mission computing algorithm, when run with different scheduling parameters and competing threads. The algorithm is CPU intensive and executes in a NoHeapRealtimeThread with a priority of 260. Measure how the same functionality scheduled with varying numbers of lower priority threads behaves. The contending threads execute in RealtimeThreads with lower priorities. The lower priority threads log timestamps and invoke `yield()` methods. All threads are periodic, executing at a 20 Hz frame rate.

Success Criteria: Difference between the tests with no background threads and the tests with 15 background threads shall be less than 5%.

Result: The first case schedules only the thread being analyzed, while the second case schedules the thread to be analyzed along with 15 background threads. The difference between the maximum, minimum, and average data points were all below 1%. This meets our success criteria. See Table 2 for detailed metrics.

Table 2. Algorithm Execution Times with Contending Background Threads (milliseconds)

	0 Other Threads	15 Other Threads	% Difference
Avg	3.5253	3.5290	0.1039%
Max	3.5395	3.5669	0.7673%
Min	3.5218	3.5244	0.0756%
Jitter	0.0177	0.0424	

3.2. Determinism

The RTSJ provides direct support for initiating functionality that needs to be run at periodic intervals either via thread scheduling or via events driven by timers. This section outlines tests investigating timing jitter associated with initiating and completing periodic activities.

3.2.1 Periodic Start of Frame Determinism

Description: Using the `PeriodicParameters` class, establish a periodic thread. Immediately after the `waitForNextPeriod()` call, log a timestamp and calculate the time between invocations. Two tests were conducted. The first test was executed with only a single 20 Hz NoHeapRealtimeThread being analyzed, while the second test was executed with the 20 Hz NoHeapRealtimeThread thread being analyzed while another fifteen lower priority RealtimeThread threads were executing at a 20 Hz frame rate. The lower priority threads log timestamps and invoke `yield()` methods. All threads are periodic, executing at a 20 Hz frame rate.

Success Criteria: Jitter shall be within 1% of the period. With a representative avionics processing rate of 20 Hz, the maximum allowable jitter is 0.5 milliseconds.

Results: The maximum jitter in both tests easily surpassed the success criteria of 0.5 milliseconds. See Table 3 for details.

Table 3. 20 Hz Frame Execution Times Measured at Frame Start Up (milliseconds)

	0 Other Threads	15 Other Threads
Avg	50.0	50.0
Max	50.0048	50.0241
Min	49.9954	49.9794
Difference	0.0094	0.0447

3.2.2 Periodic End of Frame Determinism

Description: Using the PeriodicParameters class, set up a periodic NoHeapRealtimeThread thread. Immediately after the `waitForNextPeriod()` call, execute an algorithm of significant duration but not longer than the period. After the algorithm completes, log a timestamp and calculate the difference between successive timestamps. Repeat with and without competing lower priority RealtimeThread threads as for the previous test. The lower priority threads log timestamps and invoke `yield()` methods. All threads are periodic, executing at a 20 Hz frame rate.

Success Criteria: Completion time differences shall be under 0.5 milliseconds. This represents 1% of a 20 Hz frame.

Results: The maximum jitter for 0 and 15 competing threads was 0.0282 milliseconds and 0.1441 milliseconds, respectively, which easily met the success criteria. See Table 4 for specific measurement results.

Table 4. Frame Execution Times Measured at Frame Completion (milliseconds)

	0 Other Threads	15 Other Threads
Avg	50.0000	50.0000
Max	50.0153	50.0688
Min	49.9870	49.9247
Difference	0.0282	0.1441

3.2.3 Periodic Event Determinism

Description: Measure the jitter in PeriodicTimer driven AsyncEvents. Immediately inside the handleAsyncEvent method, log a timestamp and calculate the time between invocations. The first test was executed with only a single 20 Hz NoHeapRealtimeThread being analyzed, while the second test was executed with the 20 Hz NoHeapRealtimeThread thread being analyzed while another fifteen lower priority RealtimeThread threads were executing at a 20 Hz processing rate also.

Success Criteria: Periodic event timing differences shall be under 0.5 milliseconds, 1% of a 20 Hz (50 millisecond) frame.

Results: The first case was run with the AsyncEventHandler analyzing a single thread while the second case was executed with the AsyncEventHandler analyzing a single thread with fifteen background threads. The third case was run with the BoundAsyncEventHandler analyzing one thread while the fourth case was executed with the BoundAsyncEventHandler analyzing a single thread with fifteen background threads. In all cases, the jitter met the success criteria. See Table 5 for more completion time comparisons with and without competing threads.

3.3. Latency

This section details tests assessing delays associated with context switching, synchronization, and event delivery. The RTSJ supports event-based programming for two types of event handlers: bound and unbound. A bound event handler creates one thread that is permanently bound to the handler and remains active for all event fires. An unbound event handler creates a new thread with each event fire.

Table 5. 20 Hz Frame Execution Times Measured at the Event Fire (milliseconds)

	0 Other Threads Unbound / Bound	15 Other Threads Unbound / Bound
Avg	50.0000 / 50.0000	50.0000 / 50.0000
Max	50.0667 / 50.0087	50.0908 / 50.0954
Min	49.9337 / 49.9920	49.9386 / 49.9024
Delta	0.1330 / 0.0167	0.1522 / 0.1930

3.3.1 Context Switch Latency

Description: Initiate a high priority thread and a lower priority thread. Both threads will be executing at a 20 Hz frame rate. In the higher priority thread, log a timestamp before the `yield()` in the run method. In the lower priority thread, log a timestamp after the `yield()` in the run method. Then compute the latency between the higher priority thread's timestamp and the lower priority thread's timestamp.

Success Criteria: Context switch latency shall be less than 10 microseconds.

Results: See Figure 2 for a graph of the context switch latency data samples. The results show a median of approximately 2.1 microseconds. Some of the samples spike to 2.3-2.8 microseconds, probably indicating that some processing in addition to the context switch is being run following the `yield()` call. Even with this, however, the maximum time to switch between threads was roughly 2.8 microseconds, which is better than the 10 microsecond success criteria.

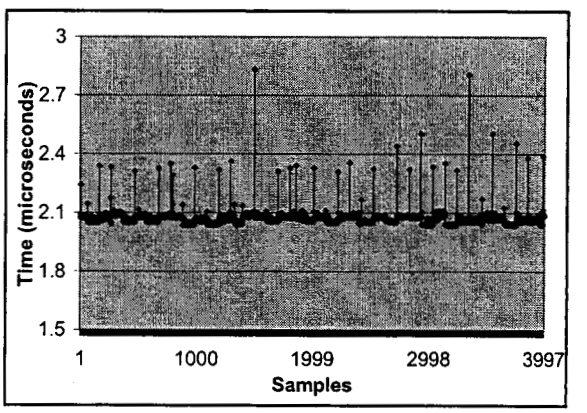

Figure 2. Context Switch Latency

3.3.2 Priority Inheritance Latency

Description: This test measures a relatively simple three thread case of priority inheritance. The low priority thread (LPT) starts and enters a synchronized method. While in that method, the medium priority thread (MPT) starts and preempts the LPT. While the MPT runs, a high priority thread (HPT) preempts the MPT and attempts to enter the same synchronized method the LPT presently has a lock on. According to priority inheritance, the LPT should get boosted up to the priority of the HPT so it can finish with the synchronized method, thus allowing the HPT to run as soon as possible. Log timestamps before and after the calls to the synchronized method. Also log timestamps at the first and last instructions inside the synchronized method. These timestamps are used to measure the boost, unboost, and total priority inheritance latency times. Each thread will execute at a 20 Hz frame rate.

Success Criteria: Priority latency shall be under 50 microseconds for boosting and unboosting priorities combined.

Results: For both cases the maximum latency was roughly 9.0 microseconds, thus the test passed the 50 microsecond success criteria. See Table 6 for more priority inheritance boost, unboost, and total latencies.

Table 6. Priority Inheritance Latency (microseconds)

	Boost	Unboost	Latency (Boost + Unboost)
Avg	5.1372	2.7735	7.9107
Max	6.2626	3.1574	8.9635
Min	4.4778	2.6517	7.1566
Delta	1.7849	0.5057	1.8070

3.3.3 Synchronization Latency

Description: Record the time elapsed to enter a synchronized method versus a non-synchronized method. Log timestamps prior to the method call and once inside the synchronized and non-synchronized methods. Each thread will execute at a 20 Hz frame rate.

Success Criteria: Synchronization latency shall be under 5 microseconds of overhead (difference between synchronized and non-synchronized).

Results: The test was executed for the synchronized and normal method latency cases. For each case the latency differences were less than 2 microseconds, thus this test passes the 5 microsecond threshold. See Table 7 for more synchronized and non-synchronized latencies.

Table 7. Synchronization Latency (microseconds)

	Non-Synchronized	Synchronized	Difference
Avg	1.3351	3.2385	1.9034
Max	1.7257	3.6962	1.9705
Min	1.3153	3.1975	1.8822

3.3.4 Event Latency

Description: Measure the latency from the firing of an AsyncEvent to the time it is handled. Log timestamps prior to the fire and once the event is handled. Each thread will execute at a 20 Hz frame rate.

Success Criteria: Event latency shall be under 100 microseconds.

Results: BoundAsyncEventHandler was used for the first case and AsyncEventHandler was used for the second. Both the BoundAsyncEventHandler and AsyncEventHandler were acceptable for our needs since all cases met the success criteria. Table 8 compares the BoundAsyncEventHandler and AsyncEventHandler latencies. Analysis of the data indicates that a relatively few measurement spikes were observed as in Section 3.3.1.

Table 8. Event Latency (microseconds)

	BoundAsyncEventHandler	AsyncEventHandler
Avg	14.675	14.584
Max	27.649	27.241
Min	14.355	14.339
Delta	13.294	12.902

3.4. Memory Management

The RTSJ defines a range of different memory types to address real-time aspects of memory management and garbage collection. This section details tests with allocation throughput, entry, and exit performance for the Heap, Immortal, Linear Time (LT) Memory, and Variable Time (VT) memory areas. The Allocation Time and Throughput Time tests were created by the Jet Propulsion Laboratory. See Figure 3 for a diagram of MemoryArea inheritance relationships in the RTSJ. In Figure 3, the Memory Area classes that are colored represent the classes with test results included herein.

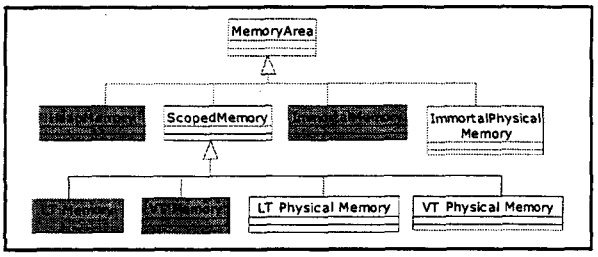

Figure 3. Memory Area Class Inheritance

3.4.1 Allocation Time vs Memory Area

Description: Measure the time required to allocate the same sized objects in different memory areas. Place time stamps before and after the memory allocation code. Then calculate the difference between before and after times for memory allocation. Perform this test for various object sizes from 4 to 16,384 bytes. Each thread executes at a 20 Hz periodic frame rate.

Success Criteria: Average allocation time less than 2 microseconds/byte shall be acceptable.

Results: The average time to create 64 byte objects took less than 16 microseconds for all memory areas, meeting the success criteria. The times for immortal, linear time, and variable time memory areas were nearly identical for this test. See Figure 4 for per-byte allocation times in the different memory areas.

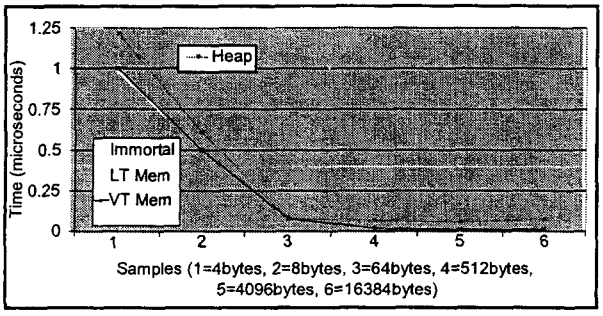

Figure 4. Maximum Memory Allocation Time per Byte for multiple byte objects

3.4.2 Throughput vs Memory Area

Description: Measure the time needed to execute a division, trigonometric, and no operation in each memory area. The division operation is a 'divide by 2' while the trigonometric operation takes the 'log of 5'. Place time stamps before and after the call to each operation. Each thread will execute at a 20 Hz frame rate.

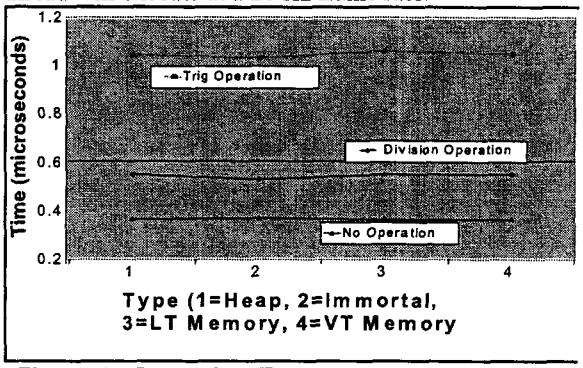

Figure 5. Operation Execution vs Memory Area

Success Criteria: The throughput values shall be within 5% across all memory types.

Table 9. Operation vs Memory Area

	Min	Max	% Delta
Float	0.5300	0.5500	4%
Trig	1.0400	1.0667	2.5%
No op	0.3667	0.3700	0.9%

Result: As tabulated in Table 9, the percent variation of operation execution across all memory areas was less than 4% thus meeting our success criteria.

3.4.3 Memory Area Entry/Exit

Description: Log timestamps before entering a MemoryArea and immediately upon entering. Also record timestamps prior to leaving the scope and immediately after leaving the scope. Each thread will execute at a 20 Hz frame rate.

Success Criteria: Average memory area entry time shall be 100 microseconds or less. Average memory area exit time shall be 100 microseconds or less.

Results: The average memory entry and exit times were under 20 microseconds for all measured memory types. Therefore this test passed the success criteria. The exit times for LT Memory and VT Memory was substantially more than for other memory areas because the garbage collector is executed on these memory areas when their scope is freed. See Figure 6 for a graph mapping the time to enter and exit the various MemoryAreas.

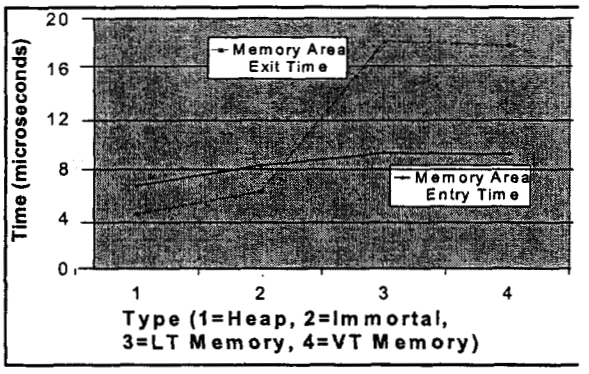

Figure 6. Average Memory Area Entry/Exit Times

4. Concluding Remarks

The experimental results in this paper indicate that emerging RTSJ implementations are capable of providing real-time characteristics with sufficient performance to meet key avionics system requirements.

There are still some areas that motivate further investigation. These areas include relative throughput to C++ or other languages, performance of memory management including garbage collection, and continued investigation into timing spikes as noted in the results. Our early results, however, indicate that the principal prerequisite real-time characteristics for mission-critical avionics systems are emerging in commercial implementations and hold promise in meeting the vision of bringing Java to large-scale DRE systems.

A second set of tests (which are not discussed in this paper) investigates performance within an environment that is representative of an actual avionics application, based on our experience with reusable component-based avionics systems on the Boeing Bold Stroke initiative. This paper will provide insight into a comparison between avionics mission computing applications that have been written in both RT Java and C++ that are based on a Bold Stroke application. We plan to publish these latter test results when complete.

5. Acknowledgements

This benchmarking effort has been a highly collaborative effort, with many contributors. We thank the US Air Force Research Laboratory Information Directorate, Wright-Patterson Air Force Base, for guiding and sponsoring this work. James M. Urnes-Jr. from Boeing created our initial set of RTSJ level tests. This paper benefited substantially from review and result interpretation insights provided by Peter Dibble at TimeSys. Ron Cytron and Ravi Pratap at Washington University in St. Louis contributed to this work, especially in the context of their aspect oriented event service named FACET, with ongoing integration results planned for future publication [5].

[1] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull, R. Belliardi, "The Real-Time Specification for Java". Addison-Wesley, 2000.

[2] A. Corsaro, D.C. Schmidt, "Evaluating Real-Time Features and Performance for Real-time Embedded Systems", Proceedings of the 8th IEEE Real-time Technology and Applications Symposium, September 2002..

[3] D.C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development", *Software Technology Conference*, May 1998.

[4] A. Corsaro, D.C. Schmidt, "Evaluating Real-Time Features and Performance for Real-time Embedded Systems", Proceedings of the 8th IEEE Real-time Technology and Applications Symposium, September 2002.

[5] F. Hunleth, R. Cytron, and C. Gill, "Building Customizable Middleware using Aspect Oriented Programming", *OOPSLA* 2001 Advanced Separation Of Concerns Workshop, Oct. 2001.