Validating Java™ for Safety-Critical Applications

Jean-Marie Dautelle[*]
Raytheon Company, Marlborough, MA, 01752

With the real-time extensions, Java can now be used for safety critical systems. It is therefore primordial to be able to guarantee that virtual machine implementations not only conform to the Real-Time Specification for Java (RTSJ) but also that efficiency and predictability is up to a certain standard. In particular, if the overhead incurred by RTSJ implementations are beyond a certain threshold, they may not suitable for safety critical systems. With these objectives in mind, we developed and maintained a test suite which addresses conformance as well as performance. Our performance metrics include latency measurements (e.g. context switch and dispatch latencies), execution jitter (e.g. jitter caused by Just-In-Time compilation, concurrent garbage collection or lower priority thread cache effect) and network distribution delays.

Nomenclature

JVM	=	Java™ Virtual Machine
RTSJ	=	Real-Time Specification for Java™
AOT	=	Ahead Of Time
JIT	=	Just In Time
GC	=	Garbage Collection
TCK	=	Technology Compatibility Kit

I. Introduction

WITH regard to performance we cannot distinguish between the Java Virtual Machine (**JVM**), the Operating System (**OS**) and the hardware. All have a direct impact on timing, latencies and precision. In other words, changing any of these three elements should require validating the whole platform again (regression tests).

In this paper, we will address the principal causes of latencies and delays when running a Java program and how their impact can be assessed in order to produce pass/fail criteria for our safety critical test suite. It is anticipated that there is no "one size fits all" set of pass/fail criteria and tailoring to project specific requirements has to be performed.

II. Just-In-Time Compilation

In the current Java execution model, all methods are compiled to JVM byte-code instructions (it is still unusual for the Java Virtual Machine to be implemented directly in hardware). The byte-code is then executed using one of the following techniques:

1. Interpretation
2. Just-in-Time (JIT) or hybrid/adaptive compilation
3. Native/Ahead-of-Time (AOT) compilation

Interpretation has good real-time characteristics but results in slow program execution. Just-In-Time and dynamic compilation increases execution speed but execution time is unpredictable and sensitive to the fluctuating execution profile. Furthermore, the "worst-case" execution time is often significantly worst than with a pure interpreted solution due to the compiler interference. Native/ahead of time compilation provides good determinism, but performance might be affected by the lack of dynamic optimization performed by adaptive compilers.

[*] DD(X) Software Architect, Dept. 30047, 1001 Boston Post Road, Marlborough, MA, 01752

It should be noted that modern JVMs with JIT-enabled may recompile the same code several times during program execution (see Figure 1), “warming up” the code to force compilation at start-up will not prevent further dynamic compilations from occurring.

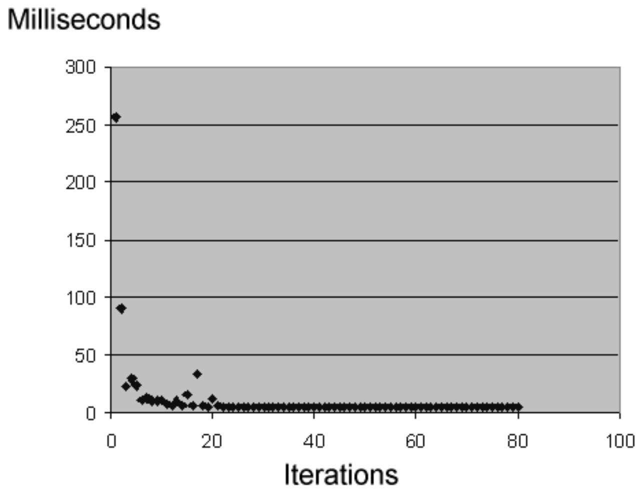


Figure 1: Execution Time - JIT Enabled

Our JIT tests allow us to evaluate different techniques which can be employed to reduce this particular source of jitter such as pre-compilation, interpreted mode or concurrent byte-code compilations (on multi-processors machines). The tests themselves consist of Java collections manipulated through the extensive use of interfaces. Indeed, one of the main accelerating effects of adaptive compilation is aggressive code in-lining to avoid expensive virtual dispatching calls. Our tests indicate that AOT compilation is not far behind JIT/Adaptive compilation in term of overall performance (10-30% slower execution speed) and has the smallest “worst-case” execution time by several orders of magnitude.

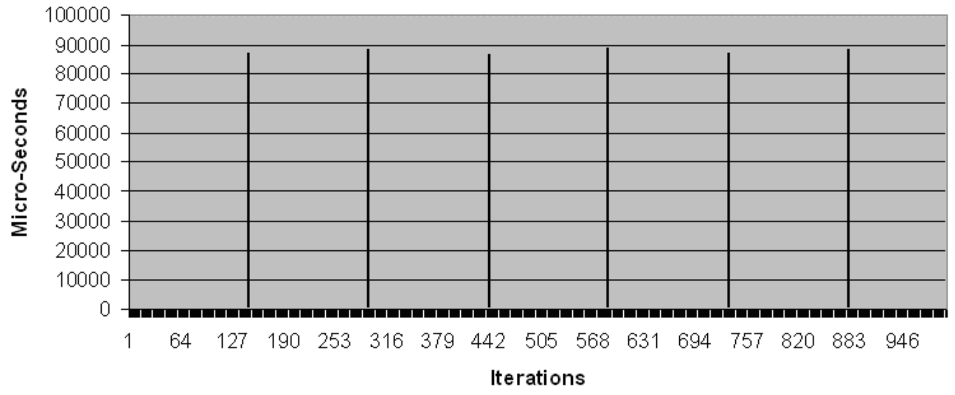
III. Real-Time Garbage Collection

Spreading the cost of garbage collection over time could suit real-time applications without resorting to a constrained programming environment (such as the one associated with RTSJ NoHeapRealtimeThread). In these past few years a significant effort has been accomplished by JVM designers with regard to “paced” (frequent interval for a short time) or “concurrent” (low-priority preemptible) garbage collections. Real-time garbage collection can considerably help in overcoming some of the RTSJ restrictions and may save significant development cost by taking most of the “real-time burden” off the shoulders of the developers.

Our GC tests simulate worst-case scenarios with regards to “garbage generation”. Our program creates a lot of dynamic objects for which life expectancy varies across the whole spectrum; with most of the objects dying fast and others surviving for a long time. Nonetheless, to allow for concurrent garbage collection, the program stays idle frequently during execution. Also, to avoid measuring JIT jitter, timing results of the first executions are discarded[†].

Our tests have shown that generational garbage collectors are extremely efficient in both allocating the objects and collecting them (most short-lived objects being in the CPU cache). Still, the worst case execution time is driven by major collections occurring regularly, as the tenured population increases see (Figure 2).

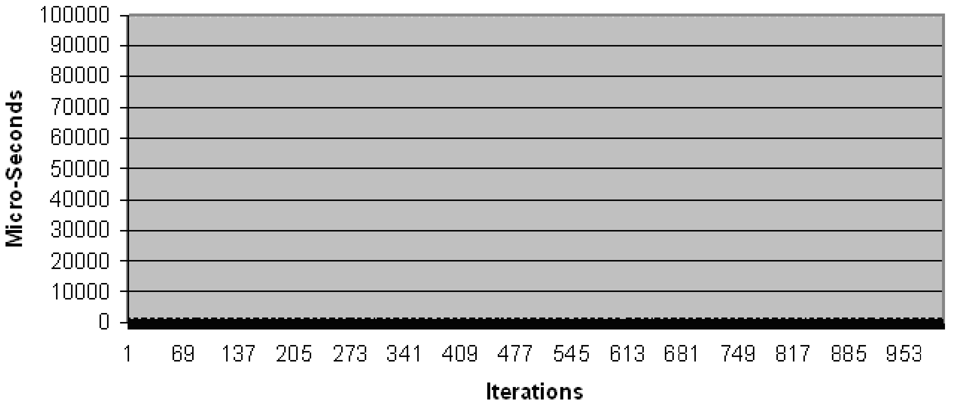
[†] JIT typically kicks in after the nth execution of a method. This default behavior can be modified through the use of JVM switches (e.g. option **–XX:CompileThreshold** for Sun™ HotSpot VM).



‡

Figure 2: Execution Time - Generational GC

On the other hand virtual machines with real-time garbage collectors, although not as effective (about 3x penalty for object creation/collection), can easily be tuned[§] to avoid “stop the world” collections (Figure 3).



**

Figure 3: Execution Time - Real-time GC

With regard to real-time garbage collection “the proof of the pudding is in the eating”. Our tests have shown that real-time garbage collection is no longer an oxymoron! Concurrent garbage collection can be extremely valuable when a jitter of a few milliseconds is tolerated. For a jitter below the millisecond level, there is not much choice other than to use NoHeapRealtimeThread.

IV. Class Initialization

By default, class initialization is performed at first use and may cascade into hundreds of classes being initialized at an inappropriate time (resulting in unexpected delays incompatible with real-time requirements). Our test suite has shown that even for empty classes (classes with no static member) this process takes up to 1ms per class.
At this time JVM vendors do not provide an easy way to initialize classes at start-up and ad-hoc tools have to be written to force class initialization using profiling data from previous executions. A capability to initialize all compiled classes at start-up would be extremely valuable. This should not be difficult for the AOT compiler as long as there is no class initialization circularity. Furthermore, for safety reasons, static class circularities should be detected, as they could result in illegal states (e.g. final class members not set).

[‡] Major GCs occur regularly producing up to 90 ms delays in execution time.
[§] Tuning consists in setting the heap size and the CPU time reserved for the garbage collector.
[**] GC has almost no impact on execution time (less than 1 ms).

V. RTSJ Implementation

All real-time systems need consistent performance. The RTSJ does not require that a conforming Java platform be unusually fast. But having poor consistent performance is not better than good performance with some limited fluctuations (often the “worst-case” execution time is actually what matters the most).
To assess the efficiency of the RTSJ implementation, our test suite measures the VM performance when RTSJ features (such as real-time threads, priority inheritance, memory model, high resolution clock, scheduling, etc.) are employed.

Concurrency Jitter

This test measures the impact of lower priority threads on the determinism of higher priority real-time threads. It is expected that lower priority threads have only a very limited impact (e.g. cache effect) on higher priority real-time threads (or even no impact at all on SMP systems). This test does not check for “priority inversion” which is covered by the priority inheritance test.

Context Switch Delay

This test measures the time incurred for a synchronized context switch between two instances of real-time threads.

Clock Accuracy

This test measures the accuracy of the real-time clock.

Scheduling Latency

This test measures the accuracy with which the “waitForNextPeriod()” method in the real-time thread class schedules threads executions periodically.

Dispatch Latency

This test measures both scheduling and dispatching of asynchronous event handlers defined in the RTSJ (AsyncEventHandler and BoundAsyncEventHandler).

Memory Allocation

This test measures the allocation time for different type of memory: Heap, Scoped and Immortal

Multi-Processors

This test measures how efficient the real-time threads are in taking advantage of multi-processors.

Floating-Point

This test measures the floating-point maximum throughput when using real-time threads.

Priority Inheritance

This test ensures that priority inheritance is efficiently implemented (compliance being tested by the RTSJ TCK).

Memory Check Penalty

RTSJ requires that any illegal memory access is detected (e.g. NoHeapRealtimeThread accessing heap objects). This test measures the cost of the run-time access checks on the program execution.

GC Impact

Because NoHeapRealtimeThread does not touch the heap, it should be able to run concurrently with the garbage collector (some RTSJ implementation may also allow RealtimeThread to run concurrently with the garbage collector as long as they do not access the heap). This test measures the impact of running the garbage collector on NoHeapRealtimeThread and RealtimeThread using Scoped/Immortal memory.

Network TCP/IP

This test measures the determinism of high priority real-time threads when communicating over a network used by lower priority threads.

VI. Standard Library

One may argue that the strength of the Java platform is its extensive standard library. Unfortunately, there is no RTSJ implementation of the core Java library at this time. Using the standard (non real-time) library could result in:

- Unexpected delays (due to lazy initialization, arrays resizing, maps rehashing, etc.)
- Memory clash (e.g. memory allocated for a static object while executing in scoped memory).
- Memory leak (e.g. surreptitious creation of temporary objects while executing in immortal memory).

Developers may have either to rewrite their own library or use third-party RTSJ compliant libraries such as the open-source ***J**avolution* library (http://javolution.org). As an example, the following figures illustrate the different timing behaviors when adding new elements to a random access list using the standard library (ArrayList) and using the ***J**avolution* equivalent (FastTable).

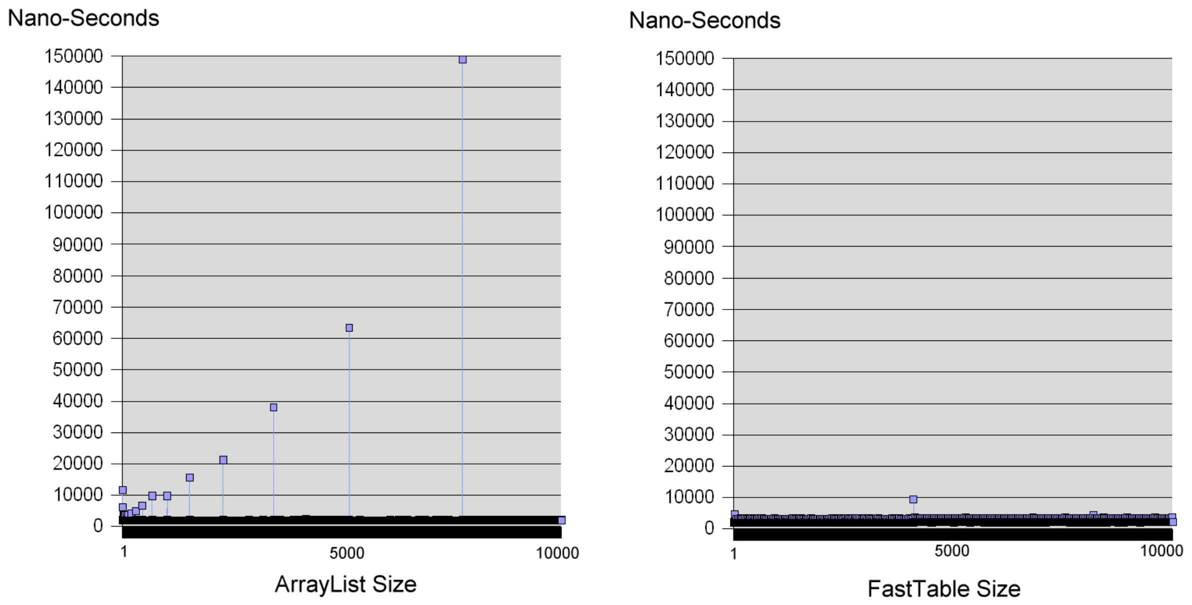


Figure 4: List.add(Object) execution time.

Our test suite does not validate the standard library for safety critical use at this time (this may change in the future when/if vendors provide a real-time implementation of the library).

VII. Conclusion

For real-time applications, compliance is not enough, good performance is also crucial. Our test suite proved to be extremely useful not only in the critical process of selecting the JVM Platform for use on our project, but also in the tuning of the JVM (e.g. configuration parameters) to best address the performance constraints of safety critical applications.