Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

It's Time for a Change: The Sun Java Real-Time System for Automation Systems

Greg Bollella, Ph.D.
Distinguished Engineer
Sun Microsystems

Abstract

The Sun Java Real-Time System (Java RTS) 2.0 was released as a produce in June of 2007. This release includes several innovative features new to Sun and Java; real-time garbage collection and initialization-time compilation. Additionally, this release offers customers remarkable precision in the control of the execution of logic in the temporal dimension.

The Java Platform requires, for any usable implementation, internal processes which automatically collect and return to the free pool unused blocks of memory. Typically called garbage collection this internal process executes in, what is essentially, opposition to the application in its access to the memory referencing data word topology (reference topology from now on) and in its movement and access to application defined blocks of memory (objects from now on). Garbage collectors need, at some level, to synchronize with the application to avoid mutually destructive simultaneous access to the reference topology and objects. Two points we can now assert, garbage collectors require processor cycles, i.e., overhead, to complete their work and garbage collectors contend with the application for access to application data structures; hence garbage collectors introduce inconsistencies in the execution of logic when viewed in the temporal dimension. For general-purpose applications these inconsistencies, also known as pause times, are currently small enough to be unnoticeable in all but the most demanding applications. For applications which control and monitor physical machines or processes, e.g., robots, cars, trains, oil refineries, manufacturing systems, financial markets, etc., these pause times are too large by *three orders of magnitude* and in the past have completely eliminated the possibility of using the Java Platform to develop and execute such applications.

Java Specification Request 01, finalized in 2001, known as The Real-Time Specification for Java (RTSJ), defines libraries and semantics which when implemented in a Java Virtual Machine (JVM) give developers and, consequently, applications precise control over temporal behavior. The semantics are a strict subset of the semantics of the general-purpose Java Platform which means that implementation of JSR-01 or the RTSJ are, in fact, also conforming implementations of the general-purpose Java Platform. The RTSJ offers developers a rich and deep set of abstractions in which logic can be executed but the one we are interested in is instances of RealtimeThread (RTT), (a class in the RTSJ which is a subclass of Thread). In all observable dimensions and function except the temporal domain logic executing in the context of an RTT is exactly equivalent to that logic executing in the context of a Thread.

978-1-1234-7890-2/08/$20.00 © 2008 IFAC 10.3182/20080706-5-KR-1001.3997

The real-time garbage collection process (RTGC) in Java RTS, which derives from work done and published at Lund University in Sweden, allows the developer to coordinate, with extreme temporal precision, the execution of the application logic and the execution of the RTGC logic. It is this coordination which allows the application logic to execute with pause times which can be, with correct configuration, under 100% processor utilization, three orders of magnitude smaller that pause times typically experienced in the general-purpose implementations of the Java Platform.

This result, I understand, sounds like magic, but I assure you, there is no magic, and there is no free lunch. My presentation will detail exactly how Java RTS 2.0 establishes and maintains the required coordination between the RTGC and the application.

The Java Language, as defined, is an 'interpreted' language, i.e., in addition to a source syntax and semantic it defines a machine language which does not match the machine language of any actual processor (subsequent realizations of Java processors notwithstanding). The Java Platform machine language is represented by a number of 'bytecodes', the correct execution of which are defined in the Java Virtual Machine Specification. "Interpreted' means that it is the JVM which is implemented in the native language of a processor and that this virtual machine executes the bytecodes. Of course, the obvious advantage is that the same set of bytecodes can be executed on any physical machine as long as functionally equivalent JVMs have been implemented.

However, the execution of bytecodes by the JVM is not nearly as efficient as the execution of the application logic rendered directly in the native language of the processor. So, implementations of the Java Platform often include what has come to be know as a, just-in-time-compiler (JIT), which during application execution and a various times, translates well-defined blocks of the application logic from bytecodes to the native language. Subsequent execution of these blocks is then at speeds comparable to those of languages compiled directly to the native language. When, what, and how to do this compilation is the subject of much research and various JITs perform with a wide variance in performance and optimization targets

The key observation with respect to the execution of the logic of the application is that the execution of the JIT, again, like garbage collection, introduces inconsistencies in the temporal domain. Here we have three issue with which to contend, overhead and synchronization as with garbage collection, but additionally we have variations in the actual execution time of application logical blocks. Fortunately, all three of these issues are handled easily compared to garbage collection.

Some work attempting to solve these three issues done outside of conformity to the RTSJ and not considered as valid Java Platforms by the Java Community is what has come to be known as ahead-of-time-compilation (AoT). This technique treats logic written in the Java Language syntax as though it did not have the intermediate form of bytecodes and directly compiles Java Language source to native processor language and links the result into a directly executable image for the operating system, processor pair. This technique, although solving the three issues mentioned above, bypasses three essential requirements of the Java Platform, class loading, class initialization, and verification. Without these three processes the resulting executable image cannot be considered conforming to the Java Platform.

The challenge for the Java RTS team was to invent a system which resolved the three issues while retaining the essential three processes. We have defined, initialization-time-compilation (ITC). The concept is simple. The JVM is given a list of method signatures (which also specify the enclosing class). When the class containing a given method had been correctly loaded, verified, and initialized the method will be compiled from bytecodes to native opcodes for execution within one of the executions contexts specified by the RTSJ. ITC requires that the developer know, a priori, which methods will be executed in which contexts and of those which are required to be temporally precise. Java RTS provides some limited tooling (more is being considered) to help the developers

learn which methods must be given to the ITC. Additionally, ITC requires that the classes be loaded, verified, and initialized prior to entry into any block of logic which requires temporally precise execution. The Java Language specifies, lazy initialization, and in typical implementations classes are not initialized until first use. Java RTS then imposes the additional programmatic structure on the application of having to use, or at least touch, classes earlier than what the strict logic demands. In practice this is straightforward and has not been a noticeable impediment to adoption of Java RTS. My talk will explain some details of ITC.

Java RTS gives developers the ability to control and understand the temporal behavior of their application with a precision three orders of magnitude greater than general purpose implementations of the Java Platform. As mentioned previously this increase in predictability in the temporal dimension does not come for free. It is well understood in the real-time community that execution platforms trade off raw throughput for predictability. However, this is a very complex and subtle area and requires much explanation and careful thought before one can fully appreciate the differences and value of the various Java Platform implementations. My talk will begin an explanation of this area.

Here are a couple of examples to keep in mind as we discuss the performance of Java RTS. The Java RTS JVM derives from the HotSpot JVM (HS) (Sun's Java Standard Edition JVM). And although these two JVMs share upwards of 95% of code they are completely different engineering artifacts and cannot usefully be compared by the same metrics. I often use the following analogy. Think of HS as a dragster and Java RTS as a Formula One race car. Clearly, on a ¼ mile straight strip of road the dragster will beat the Formula One car to the finish. But, on a typical Formula One course the dragster would be lucky to make it around the first corner. How does one compare these two, although very similar (both have four wheels, a steering mechanism, an internal combustion engine, etc.), but very different engineering artifacts? Clearly maximum speed is an interesting metric but such comparison would require much qualification.

Much of the difference between HS and Java RTS comes from the three orders of magnitude difference in temporal execution predictability. The computer science and engineering communities often consider a change in some metric of only a single order of magnitude to cause a fundamental shift in the industries considering that metric important (e.g., consider a 10x decrease in nanometer technology in chip fabrication techniques and how that would impact the industry). Now, extrapolate that to three orders of magnitude! It's the difference between a family with two children and a family with two *thousand* children. Concretely conceptualizing and fully understanding a change of three orders of magnitude is probably beyond the limits of the human mind. Yet, this is the difference between HS and Java RTS in a metric crucially important to *some* applications.

Thus, with the above characteristics developers who write control system for industrial automation systems, including advance robots, PLC based systems, transportation systems can now use a powerful, modern, fully-featured programming language with complete assurance that the timeliness requirements of the system are met.

References

Bollella, G., et al, "The Real-Time Specification for Java", Addison Wesley, 2000.

Bollella, G., Delsart, B., Guider, R., Lizzi, C., Parain, F., "Mackinac: Making HotSpot Real-Time", http://java.sun.com/javase/technologies/realtime/index.jsp, 2005

Bruno, E., "The Sun Java Real Time System on Wall Street",
http://java.sun.com/javase/technologies/realtime/index.jsp

Bruno, E., "Go Inside the Java Real-Time System" 01/03/2007
http://www.devx.com/Java/Article/33475

Dibble, P., "Real-Time Java Platform Programming", Sun Microsystems Press, 2002.

Hofert, D., "Simplify Your Real-Time Programming", 07/01/2007
http://adtmag.com/features/article.aspx?id=1990

Locke, D., "Finally! Robust Embedded Java Using the RTSJ", 05/2003
http://www.rtcmagazine.com/home/article.php?id=100215

Mikhalenko, P., "Real-Time Java: An Introduction", 05/10/2006
http://www.onjava.com/pub/a/onjava/2006/05/10/real-time-java-introduction.html

Mikhalenko, P., "Developing real-time applications with Java RTS 2.0", 03/07/2008
http://blogs.techrepublic.com.com/programming-and-development/?p=628

Wellings, A., "Concurrent and Real-Time Programming in Java", Wiley, 2004.

Technical Documentation, Sun Java Real-Time System,
http://java.sun.com/javase/technologies/realtime/reference/rts_productdoc.html

http://www.rtsj.org

http://java.sun.com/javase/technologies/realtime/index.jsp