03-2014

RTSJ Memory Areas and Their Affects on the Performance of a Flight-like Attitude Control System

Albert F. Niessner and Edward G. Benowitz

Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
{Al.Niessner,Edward.G.Benowitz}@jpl.nasa.gov

Abstract. The two most important factors in improving performance in any software system, but especially a real-time, embedded system, are knowing which components are the low performers and knowing what can be done to improve their performance. The word performance with respect to a real-time, embedded system does not necessarily mean fast execution, which is the common definition when discussing non real-time systems. It also includes meeting all of the specified execution deadlines and executing at the correct time without sacrificing non real-time performance. Using a Java prototype of an existing control system used on Deep Space 1[1], the effects from adding memory areas are measured and evaluated with respect to improving performance.

1 Introduction

1.1 Goal

Many features are added to the Java virtual machine and libraries through the Real-time Specification for Java[2] (RTSJ) with the intent to improve the real-time performance of Java. One of the more interesting features is the addition of scoped memory, where the intent is to separate functional regions of the user's application from interactions with the garbage collector (GC). In theory, a scoped memory area is entered prior to the execution of a functional region and exited after the region. The functional region does not interact with the GC because reclamation occurs when the thread count in a scoped memory region goes to zero and is performed without the aid of the GC. Of course, allocating and reclaiming memory without the aid of the GC means that there are rules on cross referencing memory areas that are detailed in the RTSJ. The goal of this work is to demonstrate that the intent of the scoped memory area holds true with "real world" software.

1.2 Approach

A Java prototype of the Deep Space 1 attitude control system was developed in order to show that Java is usable in the production of spacecraft software. The

prototype was profiled on the desktop to measure memory and processor usage. The profile information shows that almost all of the garbage collected memory is allocated in the portion of the system that computes the output response from the sensor input – the control law. The excessive allocation is due to the choice of architecture for the prototype which forces an immutable implementation of a physical units package. The physical units package is used extensively when converting from the input of angular velocity to the command thrust of each individual thruster. Hence, the control law allocates enough data to activate the GC in about 3 cycles. The software was then adapted to include placing the control law within a scoped memory block whose size was larger than required as measured from the profiling and placing instrumentation in critical parts of the system. The latter adaptation was done because there are no tools available for profiling with Timesys's Reference Implementation[3] of the RTSJ (RI).

1.3 Tools

COTS graphical development tools were used extensively in this project. Specifically, the open-source Eclipse[4] integrated development environment provided graphical code editing, browsing, debugging, and refactoring capabilities. Headway's Review[5] product was used to graphically inspect our design, plan refactoring, and allow us to maintain a consistent architecture. Additionally, JProbe[6] was used to examine memory usage and to identify critical regions for future optimization. Lastly, Timesys's Reference Implementation[3] of the RTSJ is used because it is the only fully functional, freely available implementation of the RTSJ.

2 Measuring Time

The original intent was to use the clocks and date functions provided by Sun's JVM and the RI. However, both of these implementations limit the time resolution to a millisecond and the smallest time scales being measured are hundreds of microseconds. Therefore, a simple JNI interface to the Linux time services was developed which has a microsecond resolution. This section is devoted to the characterization of the JNI interface with Sun's JVM and the RI, using the same shared libraries and byte code for all tests.

2.1 Using Sun's JVM 1.4.1

As can be seen from the histogram (see Fig. 1), using either the Mandrake 9.0 stock kernel or the Timesys real-time Linux kernel version 3.2 has little affect on the latency where almost all calls are between 3 and 25 microseconds. It can also be seen in figure 3 that there appears to be a periodic delay associated with the Sun JVM which corresponds to the third mode around 1.3 milliseconds in the histogram. An attempt was made to profile this test in order to determine if the third mode was related to the GC, but the profiler measurement interfered

with the test and the results were less than clear. However, an analysis of the test software indicates with a good level of confidence that the object allocation rate corresponds to the second mode, but the analysis is not complete and is too speculative to say conclusively that it is the GC. There are only minor differences in results between the two kernels and they all appear in the distribution of higher ordered modes.

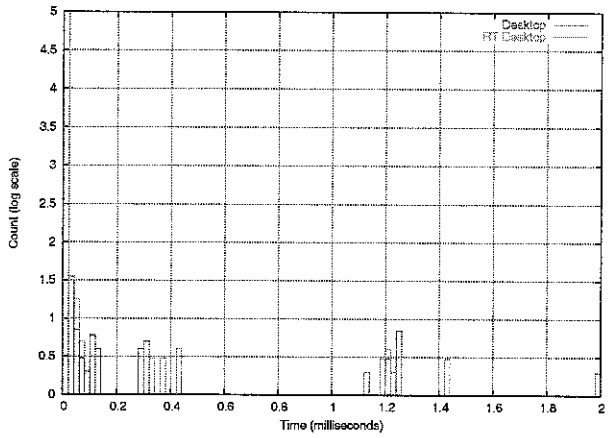


Fig. 1. JVM 1.4.1 Latency Histogram
This diagram illustrates the latency behavior differences between the Mandrake 9.0 stock kernel and Timesys's real-time kernel with respect to retrieving the current time using the JNI and a call to gettimeofday from ¡sys/time.h¿.

2.2 Using the RI

As can be seen from the histogram (see Fig. 2), the RI behaves wildly different from the Sun JVM, where the main difference for the little bit of code being executed is the HotSpot compiler. Given a very aggressive GC, the allocation rate links the second or third mode to the garbage collector. However, there is still an undetermined, systematic error just as influential as the GC which is causing the other mode. The large difference can be seen in the first mode of the histogram, where the RI's distribution is much more Poisson than is Sun's JVM counterpart[1] and is centered at a much higher value. The best explanation for the constant offset and wider distribution is the RI's lack of HotSpot or any other just-in-time compiler technology. Each iteration of the loop in the RI is

[1] It is expected that the first mode of the histogram for Sun's JVM is also Poisson, but at a much smaller scale like 2 to 50 microseconds.

interpreted, while only the first few in Sun's JVM are interpreted before they are compiled. However, expecting transient delays from the HotSpot compiler, the first 100 were ignored in all the tests, allowing the JVMs to reach a steady-state condition before measuring began. Hence, none of the delays seen should be associated with HotSpot.

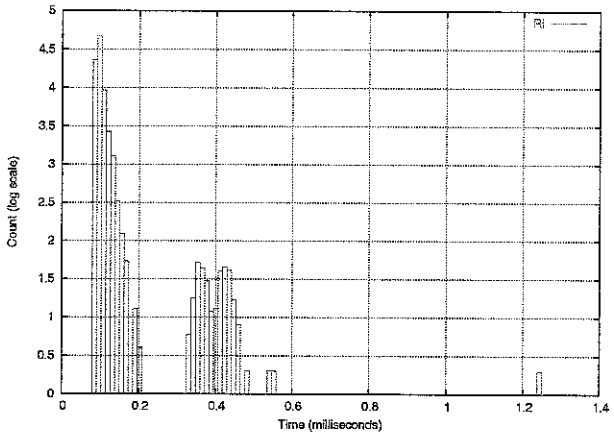


Fig. 2. RI Latency Histogram
This diagram illustrates the latency behavior of the RI with Timesys's real-time kernel with respect to retrieving the current time using the JNI and a call to gettimeofday from ¡sys/time.h¿.

3 Measuring Performance

The Java prototype being measured can be broken into three distinct parts: the full software loop, the control law, and the thruster command processing. The thruster command processing portion takes as input the desired thrust from the control law and commands the individual thrusters on the spacecraft to exert the specified force. The control law takes as input the latest sensor readings and the desired attitude requirements, allowing it to compute the desired force to meet its requirements. The full loop does all of the management related tasks, making sure that the other two components have their necessary inputs and that all other management tasks are made aware of what is happening.

Just as with the previous tests, the software is moved from a Mandrake 9.0 system with a stock kernel to the Timesys real-time kernel using Sun's JVM as a baseline. It is then tested with RI JVM using none of the RTSJ features. The RI scheduler is then added, followed by scoped memory areas. Some of the results will then be compared to highlight the most interesting details.

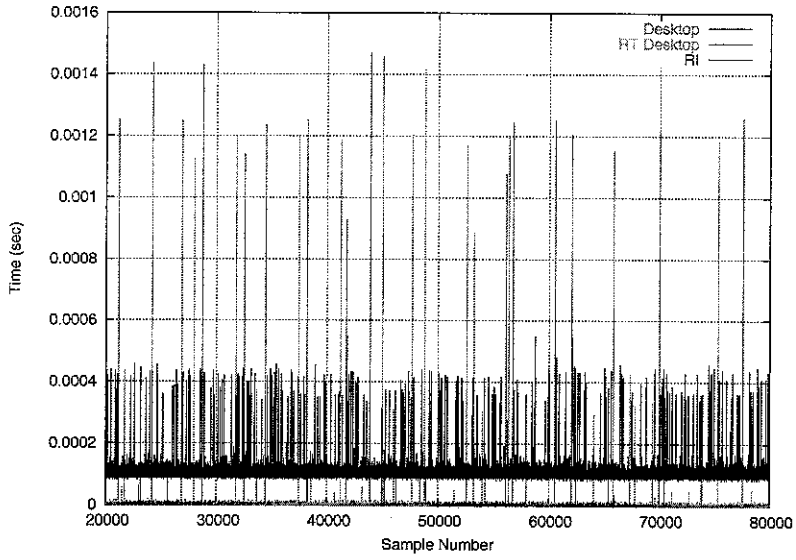


Fig. 3. Latency Times
This diagram is the measured latency from a call through the JNI to gettimeofday from ¡sys/time.h¿ for all JVM's and kernels used. It is the data that was used to construct the histograms in figures 1 and 2.

In order to execute the same code on both the desktop and with the RI, an abstraction layer was added to the system to separate the problem being solved from any implementation of the desired solution. There are two abstraction layers that are used during this test to change the behavior or the Java prototype: a scheduler abstraction layer and a memory area abstraction layer. The scheduler abstraction layer basically delegates either to the RI default scheduler or to a home-grown scheduler. The home-grown scheduler was written not as a serious scheduler but, rather, as a tool to allow the Java tools available on the desktop to used with the prototype. The memory area abstraction layer either delegates to the RI memory areas when on that platform or simply uses the heap when using Sun's JVM, and the scoped memory is allocated in the immortal space when delegating to the RI.

3.1 On the Desktop

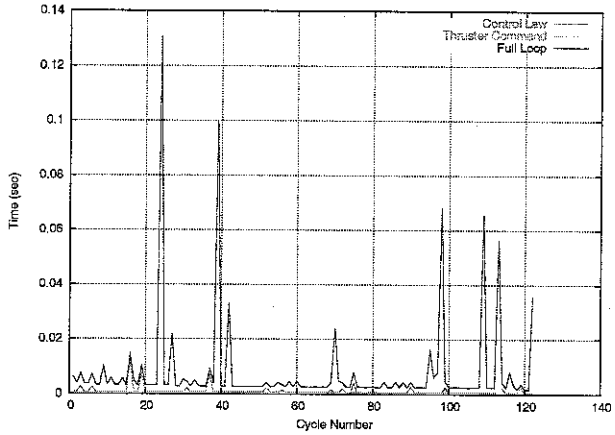
The performance chart (see Fig. 4) shows a distinct improvement (roughly 4x) in the maximum processing time between the stock kernel and Timesys's real-time kernel. In any case, all cycles were processed within their alloted time. It is also important to notice that none of the delays over two milliseconds occurred while the control law was in operation, which is the heaviest allocation portion of the system. Profiling also suggests the GC caused the difference, but it cannot be stated conclusively. If it really was the GC, the expectation is the GC would activate during the heaviest allocation portion of the system because memory was exhausted. Otherwise, the processor duty cycle is less than 25%, which leaves plenty of processing power for the GC to complete. If the GC did not complete in time, then the software would either process as normal or report that a complete cycle was missed. The jitter from the GC and the scheduler is ignored as long as the cycle boundaries are not exceeded.

3.2 Using the RI

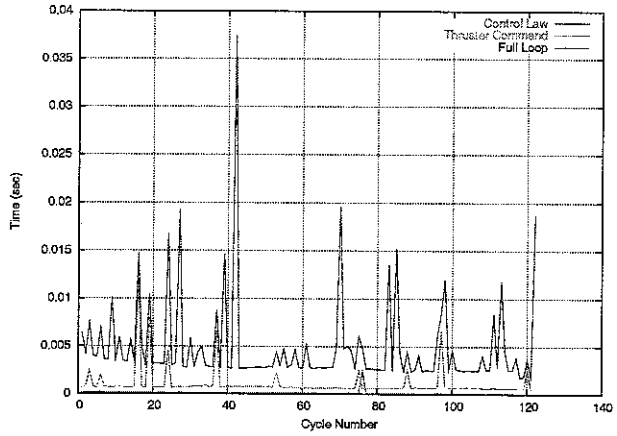
The performance chart (see Fig. 5) shows that the RI, even without memory areas and the built-in scheduler, is much more deterministic than Sun's JVM. The 100 millisecond spikes in the loop are interactions with the GC, and those interactions that are missing occurred in portions of the system that are beyond the scope of this paper – namely, the home-grown scheduler developed for the desktop and the spacecraft simulator. It is interesting that none of the GC delays occurred within the control law itself; the reason has still not been identified.

3.3 Using the RI Scheduler

The home-grown scheduler was then replaced with the RI scheduler. The performance chart (see Fig. 6) shows the full interaction of the system with the GC – the 120 millisecond spikes. Three of the GC executions occurred while commanding the thruster. There is a small bit of allocation (about 5% of one cycle)



(a) stock kernel



(b) Timesys real-time kernel

Fig. 4. Desktop Performance
This diagram illustrates, when comparing (a) to (b), the change in performance due to kernel changes. The control law takes such a small portion of the time that it is always near zero. The processing of the results and other management overhead is where the variations in processing time occur. Otherwise, the real-time kernel is somewhat slower, but otherwise performs like the stock Mandrake kernel.

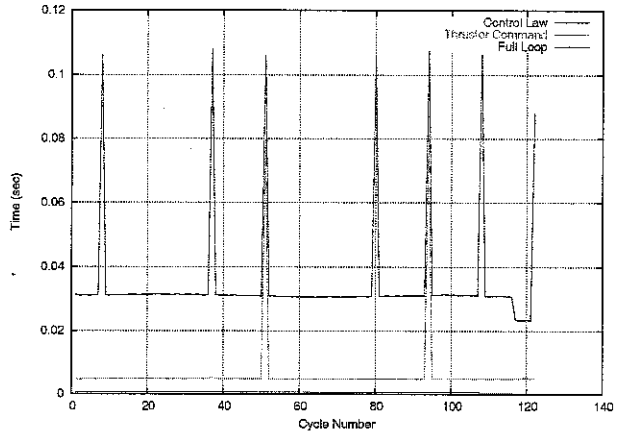


Fig. 5. RI Performance
This diagram illustrates, when comparing it to Fig. 4(b), the difference between the RI and Sun's JVM 1.4.1. The overall performance is slower, but seemingly more deterministic.

that takes place between the completion of the control loop and the start of the thruster command. Hence, it is not possible to determine if the GC execution occurred before or after the thruster command in the other cycles. However, the results from adding memory areas would imply that the delays come after the thruster command is executed and some cycle cleanup is in progress.

3.4 Adding Memory Areas

Lastly, scoped memory areas were included in the test system. The performance chart (see Fig. 7) does not contain all of the GC interactions, which is a minor mystery. The RI scheduler is being used, but there is a more processing taking place because of the scoped memory area. Hence, it is not too surprising that the interaction between the test system and the GC changes. There are minor spikes at the correct periodicity to be the GC is another minor mystery. There is no reason to measure such minor delay from the GC unless the GC only did a partial reclamation before returning control to the test software since the threads are all at the normal priority and none of them are no-heap real-time threads.

3.5 Comparing the Results

Comparing the full loop performance of each platform (see Fig. 8) highlights four very interesting details. First, the best loop time, even if it is less deterministic, is using Sun's JVM with Timesys's real-time kernel. Second, the loop processing time increases as RTSJ features are used. Third, the periodicity of

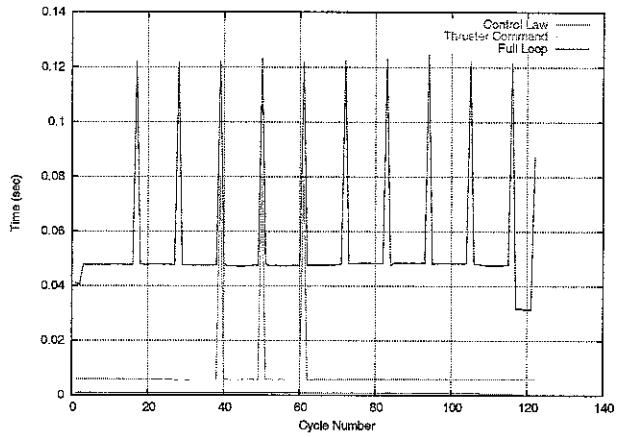


Fig. 6. RI Scheduler Performance
This diagram illustrates, when comparing it to Fig. 5, the performance changes from using the RI default scheduler.

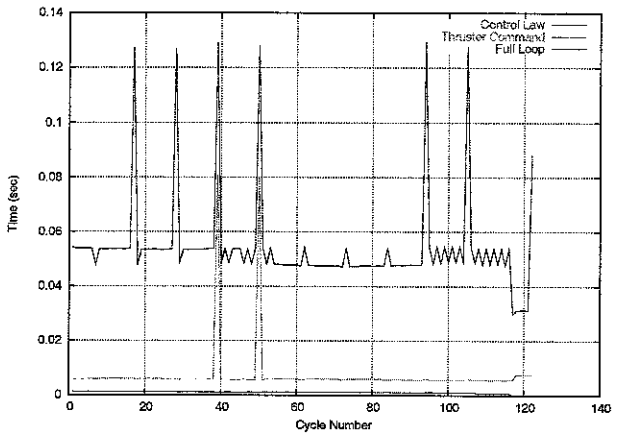


Fig. 7. RI Memory Area Performance
This diagram illustrates, when comparing it to Fig. 6, the performance changes from using the RI default scheduler and memory areas.

the GC changes with schedulers – (3) and (4) in Fig. 8 – and the period is larger with the home-grown scheduler even though the allocation rate and volume are larger. Fourth, the periodicity of the GC activity is the same between (4) and (5) in figure 8, even though there is at least a 30% reduction in allocation on the heap.

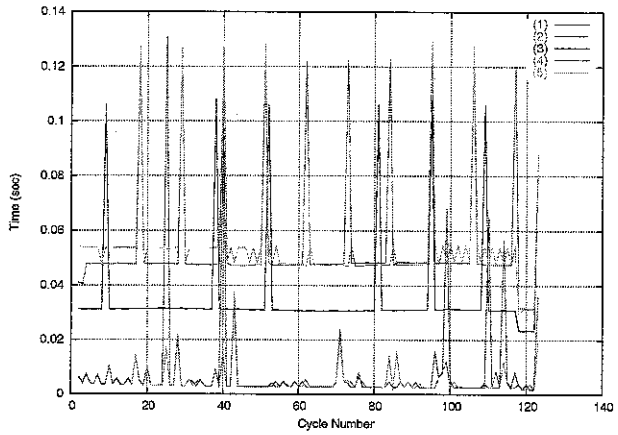


Fig. 8. Full Loop Comparison
(1) is Sun's JVM 1.4.1 with Mandrake's 9.0 stock kernel.
(2) is Sun's JVM 1.4.1 with Timesys's real-time kernel.
(3) is the RI with the same home-grown scheduler used in (1) and (2) and no RTSJ other features either.
(4) is the RI with its default scheduler and no other RTSJ features.
(5) is the RI with its default scheduler and memory areas covering the control law.
This diagram simply collects the data from the previous diagrams for a more direct comparison. Not all of the lines are easily visible as (1) and (2) coincide nearly everywhere as do (4) and (5).

The fact that Sun's JVM performs better than the RI in these tests is not too disturbing. To put this observation into perspective we need to investigate three anomalies: One, the large variation between (1) and (2) in Fig. 8 can be attributed to the difference in background services that were running simultaneously, which could have interfered with the test; most notably, X11 was not active with the Timesys real-time kernel. Two, the main difference between (2) and (3) in Fig. 8 is just-in-time compilation technology that is present in Sun's JVM and non-existent in the RI. Third, the test does not stress the capabilities of the hardware that the test was performed on; which is to say, the load is less than 50% of the hardware's capability. Hence, while Sun's JVM did better

in this instance, it probably would not scale with loading which a full implementation of the RTSJ presumably would. This observation does imply that the performance, both in the real-time and non real-time sense, can be significantly improved from compilation.

Using RTSJ features adds performance penalties. Increased time with the addition of memory areas is understandable because more checks are required in order to detect illegal assignments across memory barriers. The increase between choices of schedulers is a bit more perplexing. The home-grown scheduler is a very poor scheduler that uses very short sleep intervals, the smallest available period, as interrupts for yielding control to the rest of the test system. The sleep interval is notoriously poor with jitter and this, of course, bleeds over to the home-grown scheduler, which is why I called it very poor. However, the difference between (3) and (4) in Fig. 8 indicates the home-grown scheduler requires less processing power than the RI's scheduler, which surprised me because the home-grown scheduler is poorly written, is a high allocation rate, and is interpreted. The RI scheduler, on the other hand, appears to be part of the binary distribution and is interfaced through the JNI, which suggests that the scheduler should be fast and efficient. Hence, the penalty observation implies that one should measure every feature before using it to improve performance because it may not have the expected outcome.

The home-grown scheduler GC period is smaller than the RI scheduler's period. Again, since the RI's default scheduler appears to be a binary distribution accessed via the JNI, then less allocation should be taking place. Less allocation means that it would take more cycles to allocate enough trash to activate the GC. The incorrect change of periodicity is simply an extension of the penalty observation and implies the same consequences as well.

Significantly reducing heap allocations through the use of scoped memory increases overall processing time and the GC's activation periodicity does not change. This observation is contrary to the intent of scoped memory regions and is more than just an extension of the penalty observation. As Fig. 9 shows, the processing time of the control law is affected by the GC when running with Sun's JVM, but the RI performance contains all of the same features which are more than one cycle in width with only a constant between them. The control law is straight forward code with a single branch that is clearly present around cycle 110, where the type of compensator is changed. In the two cases (3) and (4) in Fig. 9, it is believable that they both have the same features since they both use the heap. However, (5) in Fig. 9 has the same structure as (3) and (4), which implies one of the following:

1. The structure is a function of reading the time through the JNI and therefore appears in all the RI runs. Fig. 3 contradicts this implication as there is no frequency of delay and, since the structure is defined by more than a single cycle, it is unlikely for a random process to repeat so well over an extended time.

2. The use of the scoped memory area is erroneous and the heap is being used. This is unlikely because an illegal assignment exception had to be fixed prior to the test working.
3. Scoped memory is strongly related to the heap and therefore exhibits some of the same features, but, at the end of the day, it is independent of the heap and the intended benefit can be realized. Fig. 8 clearly shows that the scoped memory area and the heap are not independent because the periodicity of the two tests, (4) and (5), are identical when there is at least a 30% reduction of allocations to the heap.

Since none of the implications are valid, the intent of the scoped memory area cannot be realized. The last observation implies that memory areas are not very effective in reducing GC interaction with the user's application.

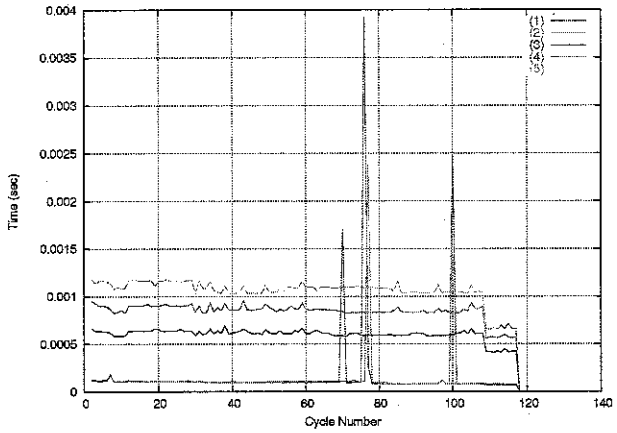


Fig. 9. Control Law Comparison
See Fig. 8 for line definitions. This diagram simply collects data from the previous diagrams for a more direct comparison. In this diagram, only (1) and (2) are nearly coincident.

4 Conclusion

The best performance enhancements came from compilation and not from trying to isolate the system from the GC through the use of memory areas. Memory areas require additional run-time processing time to ensure that dangling references and other problems do not occur. The use of static compliers[7,8] with automated scoped memory detection[9] would remove the necessity for some of

the run-time checking, and, perhaps, allow the intent of scoped regions to be realized.

Also, the lack of a tool API specification in the RTSJ and/or performance monitoring tools themselves makes it near impossible to gain conclusive data from the RI. If the RTSJ provided a required tools API for compliant JVMs, then the developer could use a generic tool or, in the worst of conditions, develop the required tools when manufacture does not supply them. The API would have to give the user visibility into GC, memory areas, the scheduler, and event handling.

As an aside, one of the problems associated with using the RTSJ memory area is the difficulty of moving data from one memory area to another; it has a viral affect on the design and architecture of the software (for further details see [10]). It was particularly time consuming and tedious to add memory areas in an architecture that uses the immutable object as a way of improving thread safety.

As a further aside, this is a single test case and its performance with RTSJ features may be improved by removing many of the abstraction layers that separate the prototype from any specific real-time extension to Java and fully embracing the RTSJ and it features, the existing benefits of the Java prototype would be compromised through increased complexity and the loss of the architectural, design, and maintenance benefits of Java.

5 Acknowledgments

The research in this paper was supported by and carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with NASA.

References

1. N. Rouquette, T. Neilson, and G. Chen, "The 13th Technology of DS1." *Proceedings of IEEE Aerospace Conference*, 1999.
2. G. Bollella , J. Gosling, B. Brosgol, P. Dibble, S. Furr, M. Turnbull, *The Real-Time Specification for Java,* Addison-Wesley, 2000.
3. Timesys Reference Implementation, http://www.timesys.com/index.cfm?bdy=java_bdy_ri.cfm, 2003.
4. "Eclipse.org" , http://www.eclipse.org/, 2003.
5. "Headway Software", http://www.headwaysoft.com/, 2003.
6. "Sitraka JProbe", http://www.sitraka.com/software/jprobe/, 2003.
7. M. Rinard et al., "FLEX Compiler Infrastructure", http://www.flex-compiler.lcs.mit.edu/, 2003.
8. A. Corsaro and D.C. Schmidt. "Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems." Technical Report 2002-001, University of California, Irvine, 2002.
9. Morgan Deters and Ron K. Cytron, "Automated Discovery of Scoped Memory Regions for Real-Time Java." In Proceedings of the 2002 International Symposium on Memory Management (Berlin, Germany), pp. 25-35. ACM, June 2002.
10. P. Dibble, *Real-Time Java Platform Programming*, Prentice Hall, 2002.