A Real-time Java Virtual Machine for Avionics
An Experience Report

Jason Baker, Antonio Cunei, Chapman Flack, Filip Pizlo, Marek Prochazka, Jan Vitek
Purdue University
Austin Armbruster, Edward Pla
The Boeing Company
David Holmes
DLTeCH

Abstract

We report on our experience with the implementation of the Real-time Specification for Java (RTSJ) in the DARPA Program Composition for Embedded System (PCES) program. Within the scope of PCES, Purdue University and the Boeing Company collaborated on the development of Ovm, an open source implementation of the RTSJ virtual machine. Ovm was deployed on a ScanEagle Unmanned Aerial Vehicle and successfully flight tested during the PCES Capstone Demonstration.

1. Introduction

The Real-Time Specification for Java (RTSJ) [3] was designed to be used to construct large-scale Distributed Real-time Embedded (DRE) systems [14, 6]. The key benefits of the RTSJ are, first, that it allows programmers to write real-time programs in a type-safe language, thus reducing many opportunities for catastrophic failures; and second, that it allows hard-, soft- and non-real-time code to interoperate in the same execution environment. This is becoming increasingly important as multi-million line DRE systems are being developed in Java, e.g. for avionics, shipboard computing and simulation. The success of these projects hinges on the RTSJ's ability to combine plain Java components with real-time ones. As of this writing commercial implementations of the specification have been released by IBM, SUN, Aonix, Aicas, and Timesys, and a number of research projects are working on open source implementations [8, 2, 5, 1, 10, 4, 17, 7, 16].

The DARPA PCES project's Capstone Demonstration integrated several independently developed real-time software systems into a live demonstration of their combined functionality, using both real and simulated components. As part of that demonstration Boeing and Purdue University demonstrated autonomous navigation capabilities on an Unmanned Air Vehicle (UAV) known as the ScanEagle (Fig. 1). The ScanEagle is a low-cost, long-endurance UAV developed by Boeing and the Insitu Group. This UAV is four-feet long, has a 10-foot wingspan, and can remain in the air for more than 15 hours. The primary operational use of the ScanEagle vehicle is to provide intelligence, surveillance and reconnaissance data. The ScanEagle software, called PRiSMj, was developed using the Boeing Open Experiment Platform (OEP) and associated development tool set. The OEP provides a number of different run-time product scenarios which illustrate various combinations of component interaction patterns found in actual Bold Stroke avionics systems. These product scenarios contain representative component configurations and interactions. These product scenarios were developed using three rate group priority threads (20Hz, 5Hz, and 1Hz) and an event notification mechanism.

The PCES project was a success. PRiSMj with Ovm was

Figure 1. A ScanEagle UAV with the Boeing PRiSMj software and the Ovm Real-time JVM.

the first Real-time Specification for Java system to pass Boeing's internal qualification tests. Ovm and PRiSMj met all of Boeing's operational requirements and the flight test conducted in April 2005 was a success. The system was awarded the Java 2005 *Duke's Choice Award* for innovation in Java technology.

This paper reports on our experience working with and implementing the Real-time Specification for Java. While our experience is limited to a single application on one virtual machine, we view these results are encouraging.

2. Real-time Java

The Real-Time Specification for Java (RTSJ) was developed within the Java Community Process as the first Java Specification Request (JSR-1). Its goal was to "provide an Application Programming Interface that will enable the creation, verification, analysis, execution, and management of Java threads whose correctness conditions include timeliness constraints" [3] through a combination of additional class libraries, strengthened constraints on the behavior of the JVM, and additional semantics for some language features, but without requiring special source code compilation tools.

3. The Ovm Virtual Machine

Ovm is a generic framework for building virtual machines with different features. It supports components that implement core VM features in a wide variety of ways. While Ovm was designed to allow rapid prototyping of new VM features and new implementation techniques, its current implementation was driven by the requirements of the PCES project, namely to execute production code written to the RTSJ at an acceptable level of performance. While Ovm's internal interfaces have been carefully designed for generality, much of the coding effort has focused on implementations that achieve high runtime performance with low development costs. The real-time support in Ovm is compliant with version 1.0 of the RTSJ in the following areas:

– *Real-time thread and priority scheduler support*: This is the basic priority-preemptive scheduler defined for real-time threads, and providing for deadline monitoring of those threads.

– *Priority inheritance monitors*: All monitor locks support the priority-inheritance protocol.

– *Periodic and one-shot timers*: These utility classes are used to release time-triggered asynchronous event handlers.

– *General asynchronous event handler support*: These handlers support the release of schedulable entities in response to system, or application defined events.

– *Memory management*: Scoped memory areas are fully supported along with the necessary checks on their usage. The use of NoHeapRealtimeThread objects is supported. Full preemption of the garbage collector is not yet implemented.

Sources and documentation for Ovm are available from [1], and the reader is referred to [11] for further discussion of Ovm.

3.1. Design and Implementation of a RTSJ VM

The overall architecture of Ovm consists of an executive-domain core around which multiple user-domain "personalities" can execute. The executive domain consists of a core set of system services that provide the functionality needed by the user domain. This includes code translation and execution, memory management, threading and synchronization, and other services typically implemented in native code, or delegated to the operating system, in other virtual machines. The executive domain is isolated from the user domain and has its own type system and type name space (it has its own notion of `java.lang.Object` which is quite separate from, and quite different to, that defined in the user domain for a Java compatible virtual machine).

Real-time support in Ovm consists of both an RTSJ-compatible implementation of the user domain `javax.realtime` runtime library, and realtime variants of many core VM services defined in the executive domain. We discuss some of the main design choices and their implications.

Java in Java. Ovm is implemented almost exclusively in Java with only small amounts of C for the bootloader and low level facilities. Even though we have not conducted a thorough study, we have anecdotal evidence of higher developer productivity and lower defect rates. The entire system is comprised of approximately 250,000 lines of Java code and 15,000 lines of C code.

Ahead of time compilation. The high performance real-time configuration of Ovm relies on ahead of time compilation. The entire program is processed to maximize the opportunities for optimization and an executable image is generated for a particular Java application. The quality of the optimization is further discussed in section 3.2.5. The Ovm optimizing compiler (called j2c) translates the entire application and virtual machine code into C++ which is then processed by gcc. The advantages of this approach is that we obtain portability at almost no cost and that we can offload some of the low level optimizations to the native compiler. The main drawback is that by going to C++, we lose some control over the generated code. So, for instance, some care has to be taken to avoid code bloat due to overeager inlining (balancing the inlining that is essential for performance). Another issue is that the C++ compiler hinders precise garbage collection, which has forced us to rely on a mostly-copying collector. This has not proven to be a significant problem for the implementation of the RTSJ – but

does complicate the task of implementing real-time garbage collection algorithms.

User-level threading. Threading is implemented in the VM by using user-level *contexts* that are executed within a single native operating system thread, with all scheduling and preemption controlled by the VM. Asynchronous event processing, such as timer interrupts and I/O completion signals, is implemented synchronously within the VM by the means of compiler inserted cheap *poll checks*. The cost associated with the polling is small (see Section 3.2.4) and can be reduced further by more aggressive compiler analysis, for instance loop unrolling can decrease the number of poll-checks needed as it reduces the number of back-branches. An advantage of explicit poll-checks is that the compiler knows when a context switch may occur and also when a sequence of instructions is atomic. This simplifies code generation and allows for some operations performed by the VM to forgo explicit synchronization. Since we do our own scheduling and synchronization, we need not rely on particular operating system features, and so do not require the use of proprietary, commercial real-time operating systems. With Ovm it is possible to run RTSJ programs on any OS and have the application threads behave correctly with respect to each other; this guarantee is not extended to other processes running on the same machine, of course. System-level blocking I/O calls such as `read` or `write` will stall the whole VM. The Ovm I/O subsystem solves this by restricting system calls to the non-blocking and asynchronous varieties. In order to properly implement Java's blocking I/O APIs, Ovm simulates standard POSIX semantics by doing its own scheduling. In effect, the Ovm I/O subsystem includes much of the same scheduling machinery that would be found in an operating system kernel.

3.2. Performance Evaluation

We have evaluated Ovm on a number of benchmarks and report some of these results here. All benchmarks in this section were run on an AMD Athlon(TM) XP1900+ running at 1.6GHz, with 1GB of memory. The operating system is Real-time Linux with a kernel release number of 2.4.7-timesys-3.1.214.

3.2.1. Throughput Benchmarks. We evaluate the raw performance of Ovm on the SpecJVM98 benchmark suite and compare with the Timesys jTime RTSJVM (compiled), and the GCJ compiler. The goal of this experiment is to provide a performance baseline. We measure two versions of Ovm: one which is the standard Ovm and the other (Ovm w. bars) including the read and write barriers on memory operation mandated by the RTSJ. jTime, likewise, has read/write barriers turned on. All three systems are ahead-of-time compiled.

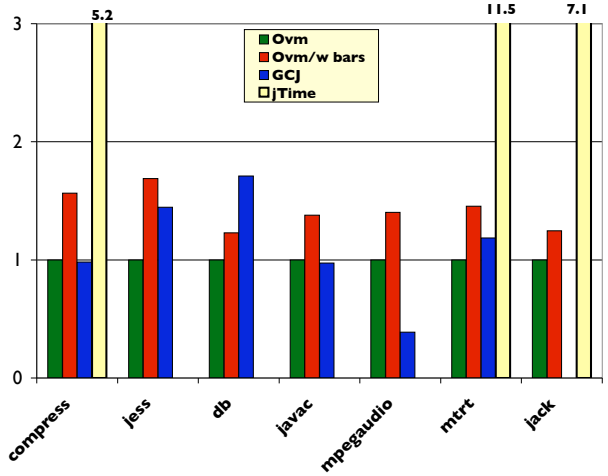


Figure 2. SpecJVM98. (normalized wrt. Ovm)

The results, given in Fig. 2 show that performances of Ovm and GCJ are close. Typically, Ovm is slightly faster with the exception of mpegaudio where massive slowdown is due in part to our treatment of floating point numbers, this will be addressed in forthcoming releases. The figure also illustrates the costs of RTSJ barriers (up to 50%). Now, clearly SpecJVM is by no means representative of a real-time application, but it gives a worst case estimate. GCJ did not execute jack successfully, and jTime could not run jess, db, javac and mpegaudio.

3.2.2. Startup Latency. We measure the startup time of Ovm on a 300MHz PPC. Fig. 3 gives the distribution of the time required to load the virtual machine from disk, perform any initialization and up to and excluding the first instruction in the user's main() method. The image used here is that of PRiSMj (22 MB of data, and 11 MB of code).

3.2.3. Boeing Latency Benchmarks. Early on in the project Boeing developed a number of latency benchmarks to compare implementations of the RTSJ [15]. Fig. 4 shows the latency of a number of basic RTSJ operations and compare to the jTime virtual machine on Timesys Linux. The figure shows the minimum, average and maximum latencies of 100 runs.
Event Latency: We create an event handler and periodically fire an event in a thread. We measure latency between the time of firing the event and the time the event handler is invoked.
Periodic Thread Jitter: We run a single periodic thread with a given period and with no computation performed. We measure jitter of period starts.
Preemption Latency: We start two threads, a low-priority one and a periodic high-priority one, which perform no computation. In the low-priority non-periodic thread we repeatedly get the current time. Once the high-priority thread

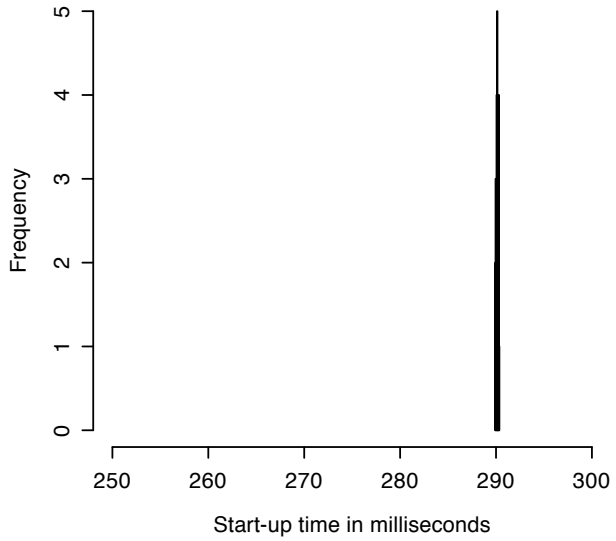


Figure 3. Ovm Startup Latency.

is scheduled, it gets the current time. We are interested in measuring the time interval between these times as it approximates the preemption latency.

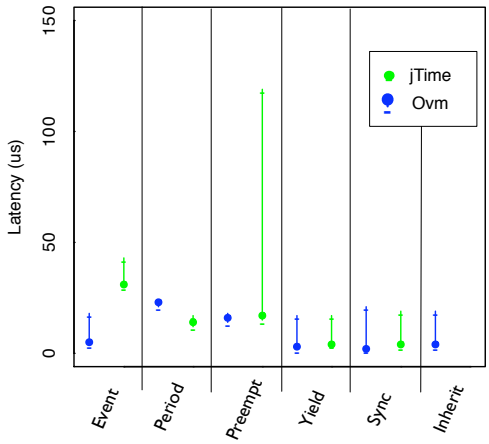


Figure 4. Boeing RTSJ Latency benchmarks.

Yield Latency: Two threads with the same priority are started. The first one repeatedly gets the current time and yields. The second thread gets the current time once it is scheduled. We measure the interval between the first thread yields and the second thread is scheduled.

Synchronization Latency: *n* threads are started and each of them tries in a loop to enter a synchronized block. In each iteration, we get the owner of the lock and the time of acquisition. The synchronization latency is measured as the time interval between the time the previous thread left the synchronized block and the time the next thread entered the synchronized block.

Priority Inheritance Latency: We start *n* lower-priority threads with priorities 1, ..., n, and we use n different locks. We also start a mid-priority and a high-priority thread. The lower-priority threads are started in the way that a thread with a priority i is waiting for a thread with priority $i-1$ to release a lock l_i. But none of those threads are in fact scheduled, since they are blocked by mid-priority thread. We measure boost/unboost times.

Overall the Ovm latencies are in line with those observed in the jTime VM running on Timesys Linux. Preemption latency is much better in Ovm as context switches are performed within the VM and are lightweight and jTime must call into the OS.

3.2.4. The effect of poll-checks. Compiler inserted poll-checks are essential to Ovm's scheduling infrastructure. Poll-checks are the only points in the program code where scheduling actions can occur. The benefit of this approach is that it simplifies the implementation of synchronization primitives. The downsides are (*i*) performance overhead, both from the time spent executing the poll-check and from compiler optimizations impeded by their presence, and (*ii*) potential increase in latency. Latency may increase if there is a long span of code without poll-checks. While the code runs, interrupts received will be deferred. To help developers understand the nature of latencies due to poll-checks, Ovm includes a profiler that produces a distribution of interrupt-to-poll-check latencies. Fig. 5 includes such a distribution for the PRiSMj 100X scenario. Other benchmarks exhibit similar behavior. It is easy to see that the current implementation of poll-checks is unlikely to have an adverse effect on latency.

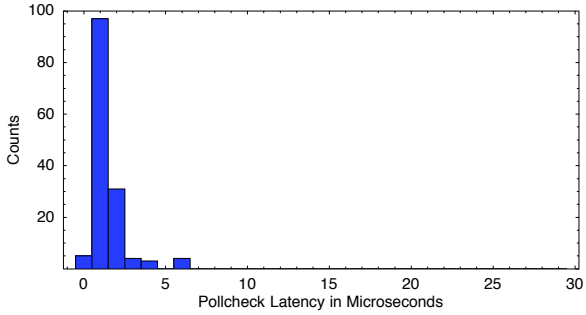


Figure 5. Distribution of poll check latencies for the PRiSMj 100X scenario. Poll-check latency is the time between an interrupt and a poll-check that services that interrupt.

To estimate the impact of poll-checks on throughput, we run

Ovm on SpecJVM98 with all poll-checks deactivated. See Fig. 6 for percent overheads measured for poll-checks in the Spec benchmarks. The overheads were computed based on the median of 20 runs with and without poll checks. All benchmarks exhibit under 10% overhead. The `javac` benchmark actually runs slightly faster with poll checks activated.

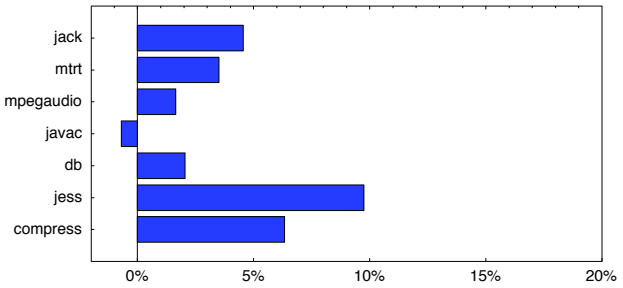


Figure 6. Percent overhead of poll checks in SpecJVM98 benchmarks. In this graph, 0% overhead indicates that enabling poll checks did not slow down the benchmark.

3.2.5. Effectiveness of optimizations. When building an Ovm image for an embedded system, we require developers to provide all Java sources in advance as well as a list of all reflective methods that may be invoked. This information is used to by the optimizing compiler to improve code quality. We give the example of two applications, PRiSMj and RT-Zen (both are described later). Fig. 7 gives the size of all components that can potentially go into an image: the application source code, the JDK classes, the source code of the virtual machine and the implementation of the RTSJ.

	LOC	Classes	Data	Code
Boeing PRiSMj	108'004	393	22'944 KB	11'396 KB
UCI RT-Zen	202'820	2447	26'847 KB	12'331 KB
GNU classpath	479'720	1974		
Ovm framework	219'889	2618		
RTSJ libraries	28'140	268		

Figure 7. Footprint. Lines of code computed over-all all Java sources files (w. comments). Data/Code measure the executable Ovm image for PPC.

The compiler performs a Reaching Types Analysis to discover the call graph of the application and in the process prune dead methods and dead classes. The result are shown in Fig. 8. The number of classes loaded refers to the classes that are inspected by the compiler (the majority of classes are never referenced by the application). The number of classes used is the number of classes that are determined to be live, i.e. may be accessed at runtime. The number of methods defined is the sum of all methods of live classes. The number of method used is the subset of those methods which may be invoked. Methods that are not used need not be compiled.

	classes loaded/used	methods defnd / used	call sites (devirt)	casts (removed)
RTZEN	3266 / 941	20608 / 9408	67514 (89.7%)	5519 (37.7%)
PRiSMj	3446 / 953	13473 / 6616	46564 (89.8%)	73408 (96.9%)

Figure 8. Impact of compiler optimizations.

Finally, Fig. 8 measures the opportunities for devirtualization and type casts removal. In Java, every method is virtual by default, we show that in the two applications at hand 90% of call sites can be devirtualized. Type casts (e.g. instanceof) are frequent operation in Java. The compiler is able to determine that a large portion of them are superfluous and can be optimized away.

3.2.6. Application level benchmarking. RT-Zen is a freely available, open-source middleware component developed at UC Irvine and written to the RTSJ API's. The system is about 50,000 lines of code. For this experiment, we use an application which implements a server for a distributed multi-player action game. The application allows players to register with the server, update location information, and find the position of all of the other players in the game. RT-Zen has a pool of worker threads that it uses to serve client requests. In our experiment, we have implemented a small server for a multi-player interactive game, the application runs with a low priority and a high priority real-time thread. Fig. 9 reports on the time taken to process a request.
The jitter for the high priority thread is approximately 7 milliseconds, this is almost entirely due to interaction between the two threads. Both of them try to acquire a shared lock and priority inheritance kicks in when the low priority threads cause the high priority thread to block. When the same benchmark is run without synchronization, as one would expect, the jitter on the high priority thread disappears.

4. Experiences Implementing the RTSJ

Each of the real-time programming areas addressed by the RTSJ presents its own implementation challenges to the VM. Ideally the implementation of different aspects of behavior would be essentially independent, and allow modular composition of system services. In practice this is not the case and in particular memory management and support for

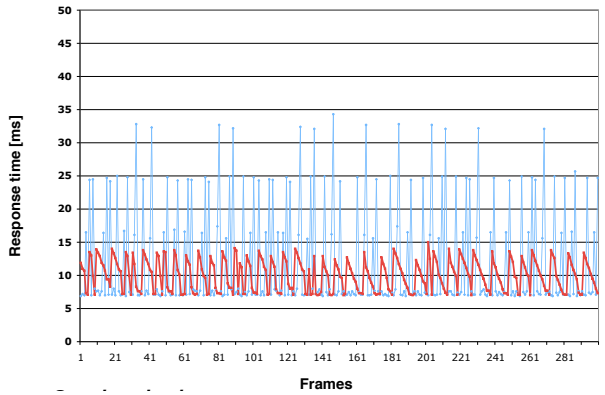


Figure 9. RT-Zen Results. Comparing the response time for an application running on top of a RTSJ CORBA ORB. Two thread groups (low and high) handle 300 requests each. The y-axis indicates the time to process a request.

NoHeapRealtimeThread objects infects much of the VM design. The following sections discuss some of the more interesting implementation issues and how we dealt with them.

4.1. Priority Scheduling

Priority scheduling is not enforced by traditional operating systems, which generally employ time-sharing or time-sliced based preemption models. These models are fair in a general sense but unsuitable for real-time systems because of the need to ensure higher priority threads always run in preference to lower priority ones. Priority preemptive scheduling is typically provided in commercial real-time operating systems, and may be available as an option for other operating systems that support the POSIX Real-time Extensions, like Linux, but often only when executing as the superuser.

While it had been the intent to make Ovm work with a native threading model, the initial use of the user-level threading model quickly demonstrated how easily real-time scheduling requirements could be implemented in Ovm independently of the operating system. This freed Ovm from any dependency on commercial real-time operating systems, or the need for privileged execution rights (where an errant real-time thread could easily hang an entire machine and necessitate a hard reboot!). Additionally, the scheduling requirements of the RTSJ need not match those provided by an OS. For example, they may differ on how a yielding thread is replaced in the ready queue: the RTSJ says it goes to the tail of the set of threads with the same priority, while the OS scheduler might place it at the head. If the VM uses native threads on such a system then it will have to take additional steps to ensure that the RTSJ execution semantics are adhered to. For Ovm, user-level scheduling allows us to easily implement any semantics required by the RTSJ.

The use of real-time Ovm on a non-realtime operating system, achieving real-time execution characteristics, was demonstrated in its use on the payload board of the ScanEagle. This single-processor embedded computer board ran Ovm as the single non-system process, with only minimal operating system services running.

4.2. Priority Inheritance

The Priority Inheritance Protocol (PIP) is well known in the real-time literature as a means of bounding priority inversion. It is also an optional component of the POSIX Real-time Extensions and supported by many commercial real-time operating systems. However, support for PIP is harder to find on non-real-time operating systems, even those that support priority scheduling. Further, the POSIX specification for how priority inheritance operates is unclear on the interaction between priority inheritance and the explicit setting of priority values, allowing for differences in how a particular implementation behaves.

So again, the use of user-level threading in Ovm allowed us to easily implement the PIP as required by the RTSJ without any reliance on operating system support.

4.3. Scoped Memory

The RTSJ identifies three different kinds of memory that can be used: heap, immortal and scoped memory. Scoped memory operates using a reference counting scheme such that when no thread is actively using a scope, the scope can be cleared of objects and reclaimed. As scopes can be reclaimed it is essential that no references to objects in a scope are stored in variables (fields or array elements) that have a longer lifetime than the object being referred to. Otherwise, when the scope was reclaimed the reference would be left “dangling”. This requires that all stores to variables be checked at runtime to ensure that they are allowed.

A further runtime check is needed when variables are loaded to ensure that a NoHeapRealtimeThread does not acquire a reference to a heap allocated object. These two runtime checks can have a serious impact on performance, so there is a strong motivation to make the checks as fast as possible, and to find ways to elide the checks when it is safe to do so. In Ovm both kinds of runtime memory checks execute in constant time and involve simple comparisons. Ovm divides memory into three slices: one for heap, one for immortal and one from which all scoped memory will be allocated. With heap in the top slice, a heap check simply involves comparing the address held in a reference with the address of the bottom of the heap.

Scope store checks are more complicated. The RTSJ restricts use of scopes such that one scope can only ever be

entered from a single other scope. This is known as the *single parent rule*. The effect of this rule is that a safe store requires that the destination variable exist in a child scope of the scope in which the referenced object is allocated. All of the scopes that are in use at any moment at runtime form a tree, that is rooted in a conceptual object known as the primordial scope. By carefully assigning upper and lower bounds to each scope in the tree, such that a child's range is a subrange of the parent's range, then a scope check consists of finding the scope in which the destination variable and the target object exist, and checking that the destination range is a subrange of the target. The resulting check operates in constant time and requires two comparisons. This technique was adapted from a similar technique used for identifying subtype relationships in Ovm [12].

4.4. Garbage Collection

The RTSJ does not require real-time garbage collection, so the garbage collector in the VM can use whatever techniques are normally available. However, the garbage collector can not be implemented without consideration of the other parts of the memory system and the existence of NoHeapRealtimeThread objects.

First, the additional immortal and scoped memory areas must all be considered GC roots (though there is an optimization to ignore a scope that has only been used by NoHeapRealtimeThread objects). Second, the garbage collector (depending on type) has to be aware that a field that held a reference to a heap object when the GC started, may not hold a heap reference late in the GC pass, due to the actions of a NoHeapRealtimeThread. This is particularly an issue for copying collectors that move an object during GC and then go through and fix up all references to the object. For immortal memory this can be fixed by using an atomic compare-and-set operation that only updates the reference if it hasn't changed (a reference field that exists in immortal memory can only be changed by a NoHeapRealtimeThread to either contain a reference to an immortal object, or null). For scoped memory it is a little more complicated. Between the time that the GC sees a heap reference and goes back to update it, the scope could have been reclaimed and reused. So the address that previously held a reference may now be a completely different type, but might coincidentally hold the same value. This can not be detected by using a compare-and-set operation (and is the commonly known ABA problem). In this case the GC must be informed that the scope has been reused and should be ignored.

4.5. Real-time Scope-aware Class Libraries

The general Java class libraries provided by proprietary virtual machines, or created by projects such as GNU Classpath (which is used by Ovm), are not written to support real-time. At the simplest level this often means that they don't have sufficiently predictable performance characteristics to be used by real-time, especially hard real-time, threads. An additional failing, however, is that many classes will cause store check failures if instances of those classes are used from scoped memory. There are two common programming techniques that typically result in these failures: lazy initialization and dynamic data structures. Lazy initialization delays the creation of an object until it is actually needed. For example, if you create a `HashMap` you can ask it for a set that allows access to all the keys or values in the map. This set is typically a view into the underlying map and is only created when asked for. But when it is created the reference is stored so that later requests for the view simply return the same object and don't create another one. If the original map is created in heap or immortal memory, and the set is first asked for when executing within scoped memory, then the set will be created in scoped memory. The attempt to store a reference to the scope allocated set into the heap or immortal allocated map, will then fail. Dynamic data structures grow (and shrink) as needed based on their usage. If a linked list allocates a node object for each entry added to the list, then adding to an immortal allocated list from scope memory will require linking an immortal node to a scoped node. This is not permitted so the attempt will fail.

We must either accept these limitations and work within them in our applications, or else rewrite libraries to ensure they always change to an allocation context that is compatible with the main object. Such changes however are detrimental to the performance of non-real-time code that also uses the libraries; and represent significant development effort. A third option may be to define a real-time library that contains a subset of the general library classes, written to be predictable, scope-aware, and perhaps even asynchronously interruptible.

5. The PCES Experiments

In the design of the test experiments both small scale prototypes and full-scale prototypes were considered. Small-scale prototypes provide an early indication of the predicted behavior of a full-scale system. Unfortunately, costly problems sometimes occur when these prototypes are extrapolated to large-scale systems. Potential problems include unexpected increases of execution times and memory utilization. On the other hand, full-scale systems can require a significant amount of manpower to develop.

To balance these forces, various size scenarios were developed by combining a number of slightly modified small-scale test scenarios into larger scale scenarios with the aid of automation tools. This collection of scenarios provided sufficient test coverage for predicting the behavior of a full-scale mission critical embedded system at reduced development costs. Leveraging technology from the

DARPA Model-Based Integration of Embedded Software (MoBIES) program [13], allowed for rapid development of large scale scenarios. MoBIES program products included a component-based real-time Open Experiment Platform (OEP) and associated development tool set with well-defined XML based interfaces. For benchmarking purposes, a modified version of a MoBIES Product Scenario with oscillating modal behavior was selected. This product scenario has been identified as the "1X" scenario and is illustrated in Fig. 10. The original version provided use of three rate group priority threads (20Hz, 5Hz, and 1Hz), event correlation, and modal behavior.

Larger-scale scenarios were created incrementally by duplicating component classes and instances from the 1X scenario. For example, a 20X scenario was created by duplicating the eight application component instances above the Physical Device layer twenty times. In addition to duplicating component instances, component types were also increased via a simple copy/renaming approach to also scale the associated code base. The 100X scenario contains a representative number of components and events in a typical single processor avionics system, while executing within a representative multi-rate cyclical context, and is therefore used to evaluate success criteria. Success criteria is based on Boeing's experience with mission critical large scale avionics systems. Fig. 11 illustrates the flight configuration.

5.1. Experiments

Experiments were run on flight hardware used on the ScanEagle UAV: an Embedded Planet PowerPC 8260 processor running at 300MHz with 256Mb SDRAM and 32 Mb FLASH. The operating system is Embedded Linux. An illustration of the 1X modal scenario is shown in Fig. 12. The

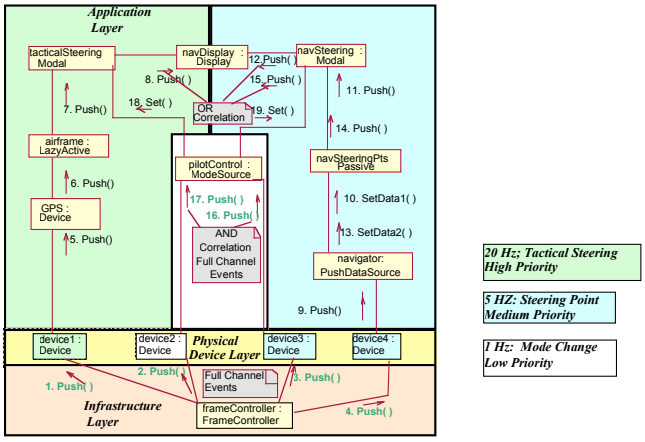


Figure 10. The overview of the Boeing PRiSMj 1X scenario.

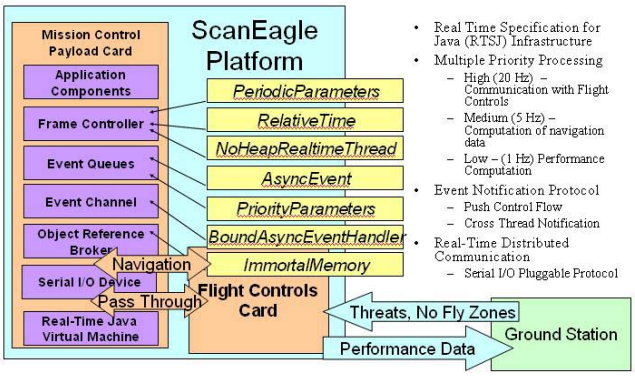


Figure 11. ScanEagle Flight Product Scenario RTSJ Architecture.

test results indicated low jitter in the order of 10's of microseconds and provided the expected behavior as demonstrated previously with the reference implementation on the desktop.

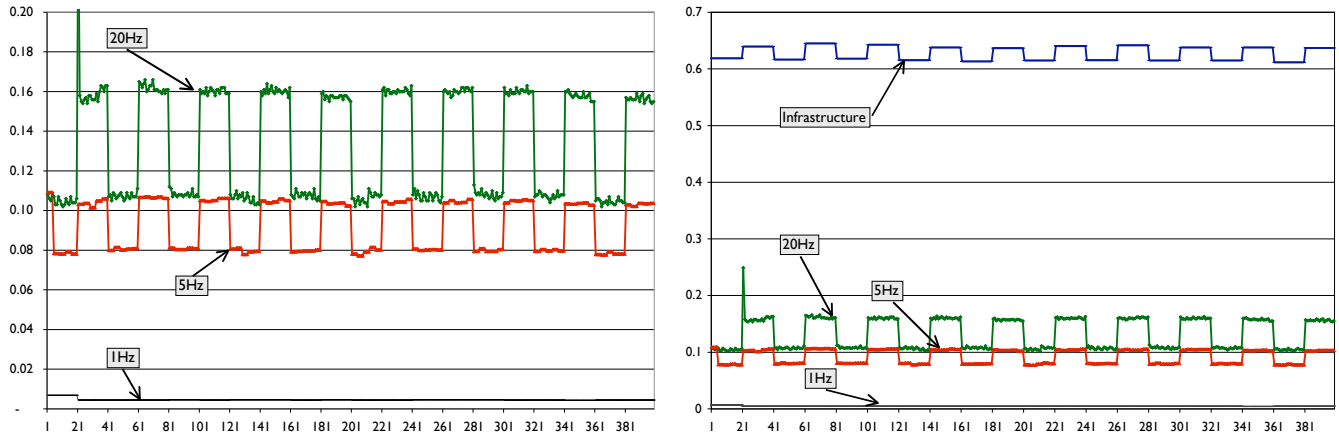
The Purdue University Ovm implementation was the first Real-Time Java application to qualify on the flight hardware. Other implementations considered included jTime, which did not support PPC, and jRate and Flex, but these could not be made ready in time. The 100X scenario test was used for the formal testing. The success criteria was that the variability in the initiation of periodic processing frames shall not exceed 1% of the associated period. For example, during the 50 millisecond period, the maximum allowable jitter is 500 microseconds. The jitter measured at approximately 100 microseconds during the 50 millisecond period. This was well within the 1% success criteria. The results are illustrated in Fig. 12.

6. ScanEagle Flight Demonstration

Ovm was used as the Java Virtual Machine for the Real-Time Java Open Experiment Platform in demonstrations at Chicago in June 2004; St. Louis, for a ground demonstration in December 2004; and White Sands Missile Range, NM, for the capstone demonstration in April 2005.

6.1. ScanEagle Flight Product Scenario

The flight product scenario was added to the OEP in order to support the ScanEagle flight demonstration using a real avionics asset. The ScanEagle using the Ovm was designated as the Reconnaissance UAV (RUAV). This ScanEagle's main function was surveillance of real-time targets during the mockup mission. The flight product scenario was responsible for providing autonomous auto-routing and health monitoring by (1) communicating with the flight controls card, (2) computing navigational cues for the flight controls based on threats and no fly zone data from the



(a) PRiSMj 1x workload. (b) PRiSMj 100x workload.

Figure 12. Response times of 100 threads split in three groups (high, medium, low) on a modal workload. The x-axis shows the number of data frames received by the UAV control, the y-axis indicates the time taken by by a thread to process the frame in milliseconds. (on flight hardware)

ground station, and (3) computing performance monitoring information to be transmitted to the ground station for real-time observation of jitter and priority processing. Synchronized communication with the flight controls was deemed as the most important mission critical function. This communication was assigned to highest priority and executed at a periodic rate of 20Hz. The navigational cue computation was deemed mission critical but not at the same level as the flight controls communication. The navigational cue computation was assigned a medium priority and computed at a periodic rate of 5Hz. The lowest priority was assigned to the computation of the performance data. This data was sent to the flight controls in the form of pass through messages and computed at a periodic rate of 1Hz. The flight product scenario is illustrated in Fig. 11. The italic yellow boxes are the RTSJ classes that were used during the demonstration.

A similar flight product scenario was developed using a C++ implementation. The ScanEagle using C++ was designated as the Tracking UAV (TUAV). This second ScanEagle was responsible for tracking a moving target. For this flight product scenario, the C++ code referenced the TimeSys real-time library functions in order to achieve the real-time performance.

6.2. ScanEagle Qualification Test

Before the mission computers could be flown, the software and hardware had to pass qualification. Both EP8260's, one loaded with the RUAV and the other the TUAV, along with the Serial UDP Bridge, had to pass the test specified by The Insitu Group. Each EP8260 was tested individually.

During February 2005, the on-board mission computer and ground base C2 systems were integrated with ScanEagle flight controls and ground station. The mission computer attached to the ScanEagle flight controls board, and the two communicated through a serial connection. The C2 system connected to the ScanEagle ground station through another serial connection. The ground station would pass appropriately formatted messages to the flight controller which would again check the message before passing it on to the mission computer. The mission computer would communicate with the C2 system by traversing the same path in the opposite direction and with the flight controls just over the direct serial connection. The integration effort was spent getting the hardware and software to accept appropriately formatted messages at the data rates that the information was supplied.

With a fully communicating system, ground qualification testing could commence. Insitu and Boeing had to demonstrate that adding the mission computer would not interfere with the flight controls in a way that the ground operator could not reassume control. The primary concerns were that a mission controller message would corrupt the flight controls or that the mating of the mission controller board to the flight controls board would cause a physical problem. To qualify the message traffic, the mission computer was installed into the hardware-in-the-loop test bed. The test bed was initialized using the ScanEagle standard operating procedure for pre-flight and take-off. Once the test-bed as in flight, the mission computer was turned on. Since the flight demonstration script was complete, the first test was to verify that the message traffic necessary to complete the script would not cause a problem. After completing that test, the test conductors, Insitu's head of software development and head of flight operations, requested a random sequence of

messages be sent. Testing continued with intermixing random messages, expected message sequences, and turning the board on and off. The test was successfully passed after both test conductors signed off on the experiment.
With the electronic qualifications complete, the boards were removed from the test bed and placed in the aircraft that were going to be used for the flight demonstration. One of the planes was taken out to a test facility for a physical check of the system. After the plane was subjected to simulated forces in flight, the plane was returned for additional electronic tests. The whole electronic system was tested to make sure the system could still execute during the demonstration. After passing both the electronic and physical test, the plane was qualified for flight tests.

6.3. ScanEagle Flight Test

On February 26, 2005, the Reconnaissance UAV (RUAV) and Tracking UAV (TUAV) were taken to the Boeing Boardman Test Facility to conduct flight tests. The first plane to fly was the RUAV. After a ground check of the systems, including the mission computer, the plane was launched. After the plane reached the preplanned reconnaissance route, the standard sequence of events was sent to the mission computer. Each step was allowed to complete before sending the next command. After successfully completing the test, the mission computer was turned off, and a test was conducted by The Insitu Group for a new part on the plane. Once the RUAV landed, the same ground tests were conducted on the TUAV, and it was launched. The only difficulty experienced during the flight tests was with the laptop used for the Serial UDP Bridge for the TUAV. The computer acted erratically during the pre-flight check and was replaced before the launch. In the end, all of the qualification tests resulted in a smooth, successful flight test.

6.4. Capstone Demonstration

On April 14, 2005, the live PCES Capstone Demonstration was conducted at White Sands Missile Range (WSMR). The demonstration consisted of a net centric demonstration of multiple kinds of systems distributed over a wide area, and networked together. Two live ScanEagles and four simulated ScanEagles with insufficient bandwidth to provide streaming video for all assets were positioned on the north end of the demonstration. The PCES program developed an end-to-end QoS technology to make optimum use of limited bandwidth communications stretching 100 miles across WSMR. The demonstration scenario started with multiple UAVs in the air in reconnaissance followed by the appearance of multiple pop up targets being prosecuted by the PCES operations center commander who has the ability to task UAVs and designate targets for track. Two of the UAVs were live ScanEagles. The RUAV played the role of an asset that has on-board autonomy supporting a variety of reconnaissance modes in support of finding and assessing damage of time sensitive targets, including support for real-time monitoring of weapon strikes against surface targets. The software on the RUAV hosted Real-Time Java technology from the PCES program. The other ScanEagle was the TUAV. The TUAV was responsible for tracking a moving target and deploying a virtual weapon capable of destroying that target.

6.5. Evaluation

This milestone marked the first flight using the RTSJ on an Unmanned Air Vehicle and received the Java 2005 *Duke's Choice Award* for innovation in Java technology.
The Embedded Planet EP8260 on board mission computer was integrated with ScanEagle flight controls in order to insure the C++ and Real-Time Java software were ready for flight. During this time, both applications needed similar changes to the flight controls interface, so the benefits and difficulties of working with each language were apparent.
Converting the OEP code from C++ to Java was fairly straight forward. The RTSJ extensions mapped well to the fully developed in-house infrastructure features with minor wrapper modifications. For example, the event channel service was developed with the underlying RTSJ BoundAsyncEventHandler class and the frame controller was developed with a periodic NoHeapRealtimeThread. Porting the C++ code to the TimeSys Linux from VxWorks presented more of a challenge. In order to get acceptable deterministic performance, the C++ frame controller had to be modified to use the TimeSys Linux specific real-time libraries instead of using the standard POSIX libraries. This required some research and debugging to determine this solution.
The development environment associated with the Java code consisted of compiling bytecodes on a desktop and connecting the desktop directly to the ground station via a serial connection. On the C++ side, the software required compilation on the desktop, perform initial unit testing on the desktop, cross compilation for the target hardware, and final testing on the ground station. These additional steps on the C++ side were due to byte ordering differences in the development x86 desktop environment and the PowerPC target platform environment combined with use of proprietary libraries to communicate with the flight controls that prevented global macro solutions. Also important to note that compiling bytecodes was in the order of 10 times faster than compiling C++ code. Thus during the majority of the integration, the C++ flight scenario product required more effort to prototype new functionality.
The C++ development suffered from tool incompatibilities. Developer studio 6.0 was used for desktop C++ develop-

ment. Developer studio provides a rich set of development and debug features. Unfortunately, developer studio is not compatible with the target TimeSys Linux O/S. In order to generate the target executable, the GNU g++ compiler was selected. Unexpected compilation and executable errors propagated to the target executable due to macro definitions (`#DEFINE`) not being set properly, missing precompiled headers, and accidental use of win32 specific libraries. With the Java development, the Eclipse tool set was used. Eclipse also provides a rich set of development and debug features. In contrast, the same Eclipse tool could be used for both the development and target environment thereby eliminating tool set incompatibility errors.

7. Conclusion

Overall our experience implementing and using the Real-time Specification for Java was positive. The implementation of the virtual machine presented a number of challenges which were resolved. We uncovered some ambiguities in the Specification which are being addressed in the upcoming revision of the RTSJ. From the user's perspective, the RTSJ extensions mapped well to the infrastructure services already developed on Boeing avionics platforms. Given the same constraints placed on large scale real-time embedded C++ applications, the Ovm running RTSJ classes provided comparable performance. In general, the Java language itself offered better portability and productivity over a traditional language such as C++. The main concern expressed was about the level of maturity of tools and vendor support.

Acknowledgments. The authors thank Kenn Luecke from Boeing and Chip Jones from Open Computing, Inc. for collaboration and development of the Boeing OEP. The authors thank James Liang, Krista and Christian Grothoff, Andrey Madan, Gergana Markova, Jeremy Manson, Krzystof Palacz, Jacques Thomas, Ben Titzer, Hiroshi Yamauchi for their contributions to the Ovm framework. We also thank Doug Lea and Bill Pugh for their feedback and advice. Doug Schmidt and Joe Cross for their continued support. This work was funded under a DARPA PCES contract.

References

[1] The Ovm virtual machine, `www.ovmj.org`, 2005.

[2] William S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for real-time Java. *Emsoft - LNCS*, 2211, 2001.

[3] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[4] Dries Buytaert, Frans Arickx, and Johan Vos. A profiler and compiler for the Wonka Virtual Machine. In *USENIX JVM'02 Work in Progress*, San Francisco, CA, August 2002.

[5] Angelo Corsaro and Doug Schmidt. The design and performace of the jRate Real-Time Java implementation. In *The 4th International Symposium on Distributed Objects and Applications (DOA'02)*, 2002.

[6] Daniel Dvorak, Greg Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Brian Giovannoni, Mark Indictor, Kenny Meyer, Alex Murray, and Kirk Reinholtz. Project Golden Gate: Towards Real-Time Java in Space Missions. In *Proceedings of the Seventh International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, 2004.

[7] Urs Gleim. JaRTS: A portable implementation of real-time core extensions for Java. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '02): August 1–2, 2002, San Francisco, California, US*, Berkeley, CA, USA, 2002. USENIX.

[8] Timesys Inc. jTime, 2003.

[9] Joseph P. Loyall, Richard E. Schantz, David Corman, James L. Paunicka, and Sylvester Fernandez. A distributed real-time embedded application for surveillance, detection, and tracking of time critical targets. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 88–97. IEEE Computer Society, 2005.

[10] Kelvin Nilsen. Adding real-time capabilities to Java. *Communications of the ACM*, 41(6):49–56, June 1998.

[11] Krzysztof Palacz, Jason Baker, Chapman Flack, Christian Grothoff, Hiroshi Yamauchi, and Jan Vitek. Engineering a common intermediate representation for the Ovm framework. *The Science of Computer Programming*, 57(3):357–378, September 2005.

[12] Krzysztof Palacz and Jan Vitek. Java subtype test in real-time. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP03)*, Darmstadt, Germany, 2003.

[13] Wendy Roll. Towards model-based and ccm-based applications for real-time systems. In *ISORC*, pages 75–82, 2003.

[14] David Sharp. Real-time distributed object computing: Ready for mission-critical embedded system applications. In *Proceeding of the Third International Symposium on Distribtued-Objects and Applications (DOA'01)*, 2001.

[15] David C. Sharp, Edward Pla, Kenn R. Luecke, and Ricardo J. Hassan II. Evaluating Real-Time Java for mission-critical large-scale embedded systems. In *IEEE Real Time Technology and Applications Symposium*, pages 30–36, 2003.

[16] Fridtjof Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, 1999.

[17] Jörgen Tryggvesson, Torbjörn Mattsson, and Hansruedi Heeb. Jbed: Java for real-time systems. *Dr. Dobb's Journal of Software Tools*, 24(11):78, 80, 82–84, 86, November 1999.