AFRL-VA-WP-TP-2006-319

A REAL-TIME JAVA VIRTUAL MACHINE FOR AVIONICS (PREPRINT)

Austin Armbruster, Edward Pla, Jason Baker, Antonio Cunei, Chapman Flack, Filip Pizlo, Jan Vitek, Marek Procházka, and David Holmes

FEBRUARY 2006

Approved for public release; distribution is unlimited.

STINFO COPY

This work, resulting in whole or in part from Department of the Air Force contract number F33615-03-C-3317, has been submitted to ACM for publication in ACM Transactions on Embedded Computing. If published, ACM may assert copyright. The United States has for itself and others acting on its behalf an unlimited, paid-up, nonexclusive, irrevocable worldwide license to use, modify, reproduce, release, perform, display, or disclose the work by or on behalf of the Government. All other rights are reserved by the copyright owner.

AIR VEHICLES DIRECTORATE
AIR FORCE MATERIEL COMMAND
AIR FORCE RESEARCH LABORATORY
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7542

NOTICE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Wright Site (AFRL/WS) Public Affairs Office (PAO) and is releasable to the National Technical Information Service (NTIS). It will be available to the general public, including foreign nationals.

PAO Case Number: AFRL/WS 06-0526, 23 Feb 2006.

THIS TECHNICAL REPORT IS APPROVED FOR PUBLICATION.

*//Signature//

DANIEL J. SCHREITER
Project Engineer

//Signature//

MICHAEL P. CAMDEN, Chief
Control Systems Development and
Applications Branch
Control Sciences Division

//Signature//

BRIAN W. VAN VLIET, Chief
Control Sciences Division
Air Vehicles Directorate

This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS**.

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
February 2006	Journal Article Preprint	04/30/2003– 04/15/2005

4. TITLE AND SUBTITLE
A REAL-TIME JAVA VIRTUAL MACHINE FOR AVIONICS (PREPRINT)

5a. CONTRACT NUMBER
F33615-03-C-3317

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
0602201

6. AUTHOR(S)
Austin Armbruster and Edward Pla (The Boeing Company)
Jason Baker, Antonio Cunei, Chapman Flack, Filip Pizlo, and Jan Vitek (Purdue University)
Marek Procházka (SciSys)
David Holmes (DLTeCH)

5d. PROJECT NUMBER
A041

5e. TASK NUMBER

5f. WORK UNIT NUMBER
0E

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The Boeing Company
P.O. Box 516
St. Louis, MO 63166

Purdue University
SciSys
DLTeCH

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Vehicles Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson Air Force Base, OH 45433-7542

Defense Advanced Research Projects Agency (DARPA)
3701 N. Fairfax Drive
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL-VA-WP

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-VA-WP-TP-2006-319

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
This work, resulting in whole or in part from Department of the Air Force contract number F33615-03-C-3317, has been submitted to ACM for publication in ACM Transactions on Embedded Computing. If published, ACM may assert copyright. The United States has for itself and others acting on its behalf an unlimited, paid-up, nonexclusive, irrevocable worldwide license to use, modify, reproduce, release, perform, display, or disclose the work by or on behalf of the Government. All other rights are reserved by the copyright owner.

PAO Case Number: AFRL/WS 06-0526 (cleared February 23, 2006). Report contains color.

14. ABSTRACT

The authors report on their 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.

15. SUBJECT TERMS
Embedded systems, Distributed real-time embedded systems, software development, real time Java, Java virtual machine

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Daniel J. Schreiter
Unclassified	Unclassified	Unclassified	SAR	38	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

A Real-time Java Virtual Machine for Avionics

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

1. INTRODUCTION

The Real-Time Specification for Java (RTSJ) [Bollella et al. 2000] was designed to be used to construct large-scale Distributed Real-time Embedded (DRE) systems [Sharp 2001; Dvorak et al. 2004]. 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

This work was supported under a DARPA PCES contract, NSF 501-1398-1086 and NSF CSR-AES 501-1398-1588.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

PREPRINT

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 [Timesys Inc 2003; Beebee, Jr. and Rinard 2001; Corsaro and Schmidt 2002; Purdue University - S3 Lab 2005; Nilsen 1998; Buytaert et al. 2002; Tryggvesson et al. 1999; Gleim 2002; Siebert 1999].

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 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" [Bollella et al. 2000] 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. The RTSJ covers five main areas related to real-time programming:

—*Scheduling*: Priority based scheduling guarantees that the highest priority schedulable object is always the one that is running. The scheduler must also support the periodic release of real-time threads, and the sporadic release of asynchronous event handlers that

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

can be attached to asynchronous event objects that themselves are triggered by actual events in the execution environment.

—*Admission control and cost enforcement*: Schedulable objects can be assigned parameter objects that characterize their temporal requirements in terms of start times, deadlines, periods, and cost. This information can be used to prevent the admission of a schedulable object if the resulting system would not be feasible from a scheduling perspective. Schedulable objects can also have handlers that are released in the event of a deadline miss.

—*Synchronization*: Priority inversion through the use of Java's synchronization mechanism (monitors) must be controlled through the use of the priority inheritance protocol (PIP), or optionally, the priority ceiling emulation protocol (PCEP). This applies to both application code and the virtual machine itself.

—*Memory Management*: Time-critical threads must not be subject to delays caused by garbage collection. To facilitate this, NoHeapRealtimeThread are prohibited from touching heap allocated objects, and so can preempt garbage collection at any time. Instead of using heap memory, these threads can use special, limited-lifetime, memory areas known as *scoped memory areas*, or an immortal memory area from which objects are never reclaimed.

—*Asynchronous Transfer of Control*: It is sometimes desirable to terminate a computation at an arbitrary point. The RTSJ allows for the asynchronous interruption of methods that are marked as allowing asynchronous interruption. This facilitates early termination while preserving the safety of code that does not expect such interruptions.

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 [Purdue University - S3 Lab 2005], and the reader is referred to [Palacz et al. 2005] for further discussion of Ovm.

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 to execute Java code. As such, it provides services to both itself and 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. Although the executive domain is written in Java, it is not subject to regular Java semantics. We use compiler tricks such as pragmas and intrinsic methods to allow executive domain code to perform type-unsafe operations such as pointer arithmetic, unrestricted raw memory access, and unchecked pointer writes even within scoped memory. Further, the executive domain notion of such classes as `java.lang.Object`, `java.lang.String`, and `java.lang.Throwable` is quite separate from, and quite different to, that defined in the user domain for a Java compatible virtual machine.

Figure 2 illustrates the architecture of the Ovm. The executive domain implements the core functionality of Java, such as monitors, memory allocation for the `new` expression, type casts, and exceptions. Because all of this functionality is normally accessed by Java programs using ordinary bytecode instructions, the Ovm's compiler must know to translate these instructions into appropriate executive domain method calls. This is achieved via a glue layer called the *Core Services Access*, or CSA. The compiler translates instructions such as *MONITORENTER* or *ATHROW* into calls to CSA methods. Because a CSA call leads to execution of Java code, recursive CSA calls are possible. For example, the implementation of monitor entry may allocate memory using the *NEW* instruction, which then causes another calls into the CSA.

Domains in the Ovm are firmly segregated. The executive domain can only call into the user domain using a reflection API. On the other hand, the user domain can only call into

the executive domain using *Library Imports*. The Ovm compiler recognizes user domain classes that have the name `LibraryImports`. Any `native` methods in a library imports class are translated into calls to methods of the same name in the executive domain `RuntimeExports` object.

Because Ovm is written in Java, the ordinary Java notion of native code does not apply. Native code in the GNU CLASSPATH library is translated into calls to user domain `LibraryGlue` classes. These classes then typically use library imports to access VM functionality. As such, the typical calling sequence for Java native methods goes like this: native method → library glue → library imports → runtime exports → Ovm kernel method that implements the requested functionality.

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.

3.1 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

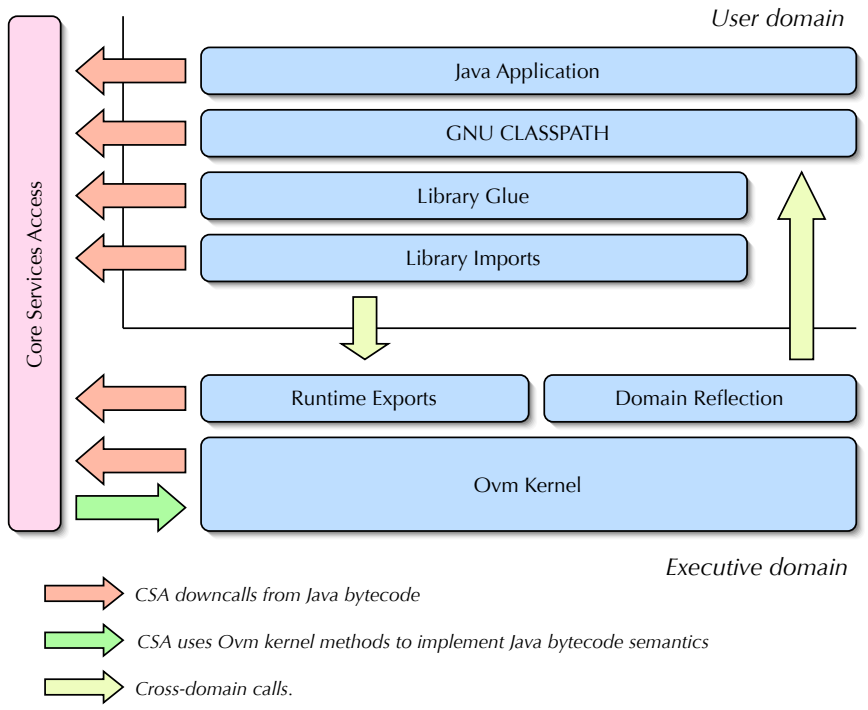


Fig. 2. Overview of the Ovm architecture.

of C code.

3.2 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 4.5. 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 are that we obtain portability at almost no cost and 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. 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.

3.3 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 *poll checks*. The cost associated with the polling is small (see Section 4.4) 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 backbranches. 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.

In designing our poll checking scheme we had the following goals:

(1) Cheap. A poll check should not require more than a load and compare on a single 32-bit word at a well-known location. In the future, we will explore register allocating this word, making the check even cheaper.

(2) Thread- and signal-safe. A signal handler may interrupt the Ovm at any instruction boundary. Whatever action the signal handler takes to cause the next poll check to trigger must be correct even if the point of interruption was another poll check.
Additionally, we have explored having the Ovm start OS threads to perform auxiliary tasks such as I/O. Such threads may wish to notify Java code of new conditions. The only facility for doing so is to cause a poll check to trigger. The procedure for doing this should work even in SMP environments.

(3) Fast dynamic deactivation. Often, it is necessary to run code that was compiled with poll checks in an uninterruptible mode. For example, our linked list library is used both from uninterruptible code in the scheduler and event manager, and in interruptible user code. Being able to dynamically disable poll checks means that this code only needs to be compiled once.

We achieve all three goals by using simple atomic operations over a 32-bit polling word. The C++ definition of this word is shown in Fig. 3.

```
union {
    struct {
        volatile int16_t notSignaled;
        volatile int16_t notEnabled;
    } s;
    volatile int32_t pollWord;
} pollUnion;
```

Fig. 3. Definition of the 32-bit polling word.

The `s.notSignaled` field is set to 1 by default, and is set to 0 using a simple store operatioon whenever a signal occurs. This may happen from a signal handler, or from an OS thread running in concurrently to the Ovm Java thread. The `s.notEnabled` field is set to 0 when poll checking is dynamically enabled, or 1 when it is disabled.

A poll check then becomes a simple matter of comparing `pollWord` to 0. If it is 0, we know that both a signal occurred and poll checking is enabled. The full code for a poll check is shown in Fig. 4

```
if (pollUnion.pollWord == 0) {
    pollUnion.s.notSignaled = 1;
    pollUnion.s.notEnabled = 1;
    handleEvents();
}
```

Fig. 4. Code for a poll check.

Notice that the fast path is just a load and a compare. The slow path involves disabling poll checks and clearing the signal, and then entering the event handling code. Poll checks must be disabled because the event handler may call into common code that was compiled with injected poll checks.

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.

3.4 I/O scheduling

In a user-level threaded system, blocking I/O calls such as `read` or `write` will stall the whole virtual machine. Unfortunately, Java class libraries expect POSIX I/O calls with blocking semantics.

To get around this we developed the Async-I/O framework. Clients, such as the Ovm POSIX I/O emulator, are restricted to using asynchronous operations. Calling a blocking method in the framework will only arrange for the operation to take place at some point in the future, callbacks are used to notify the client when the operation is ready to make progress. For example, a `write` call may be processed as follows.

(1) A thread calls *FileOutputStream.write*. This method calls the POSIX I/O emulator's `write` method which has a fast path, where a simple non-blocking `write` is executed. Failing this, a *BlockingCallback* is allocated, and the Async-I/O write method

is called with the callback as argument. The thread is removed from the ready queue and other threads are free to execute. The *BlockingCallback* will be responsible for making the thread ready once progress can be made.

(2) At some point in the future, the operating system notifies Ovm that it is ready for writing. The Async-I/O infrastructure dispatches a call to the *BlockingCallback*, passing an *AsyncFinalizer* object that encapsulates the next action that must be taken to process the write. The *BlockingCallback* places the Java thread back on the ready queue and arranges for it to call into the *AsyncFinalizer*.

(3) Once the Java thread awakens, it calls into the *AsyncFinalizer*. This may return an error, or it may return success, or it may indicate that the request was not yet completed. In the latter case, the thread is removed from the ready queue and the process repeats. Otherwise, the operation is complete.

Figure 6 shows the actual code in the POSIX I/O emulator for handling this part.

This design has several interesting consequences. First, it minimizes the amount of work that the Ovm must do while in the interrupt service routine. Everything except for the basic dispatching is stuffed into an *AsyncFinalizer*. Second, actions offloaded into the *AsyncFinalizer* are executed under the priority of the Java thread. This means that I/O operations done on behalf of low priority threads will only be executed once high priority threads yield. Third, it allows for certain I/O operations to interact with the garbage collector—for example, if the operation is given a heap buffer. If all of the work done for I/O operations was done in interrupt context, interaction with the collector would have to be limited.

The Async I/O framework enables multiple implementations of the same I/O operation, to be selected depending on client requirements and the type of resource being accessed. Current implementations include SIGIO and Select, which enable interrupt-driven scheduling; the Polling configuration, in which the VM repeatedly attempts I/O operations until they succeed; and the Stalling configuration, in which I/O stalls the whole VM. All of these configurations can be made active in the VM at once, with configurations being selected automatically every time a file is opened or a socket is created. Figure 5 shows the Ovm I/O stack implemented using the Async framework.

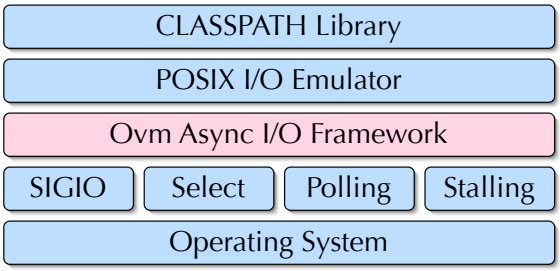


Fig. 5. Ovm I/O subsystem stack. An asynchronous implementation of the POSIX I/O API is provided to prevent the entire virtual machine from blocking during file or network operations.

```
BlockingCallback bc = new BlockingCallback();
descriptor.write(buffer, byteCount, bc);
bc.waitOnDone();
if (bc.getError() != null) {
    ... // handle error
}
return bc.getFinalizer().getNumBytes();
```

Fig. 6. Code for implementing the slow path of `write` in the Ovm POSIX I/O emulator.

3.5 Scoped Memory and Region Based Memory Management

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 [Palacz and Vitek 2003].

Ovm supports region based memory management. We use our memory region support for implementing RTSJ scoped memory as well as for managing memory in the Ovm executive-domain. To reduce code duplication while giving the executive-domain all of the hooks it needs, we designed our own memory area API. In this API, a single `MemoryManager` object represents the capabilities of the garbage collector and region allocator and objects of type `VM_Area` represent memory areas, including the heap and immortal memory. See Figure 8 for an illustration.

```
//the write barrier's fast path is always inlined.
void storeCheck(VM_Address src, int offset, VM_Address tgt)
        throws PragmaNoPollcheck, PragmaNoBarriers, PragmaInline {
    int sb = src.asInt() >>> blockShift;
    int tb = tgt.asInt() >>> blockShift;
    if (sb != tb)   storeCheckSlow(sb, tb);
    // Succeed if objects are allocated in the same block.
}

// The slow path perform a scope inclusion check
void storeCheckSlow(int sb, int tb)
        throws PragmaNoPollcheck, PragmaNoBarriers {
    VM_Word tidx = VM_Word.fromInt(tb - scopeBaseIndex);
    if ( !tidx.uLessThan(scopeBlocks))   return;
    // sucess if the target is in immortal or heap memory.
    Area ta = scopeOwner[ tidx.asInt() ];
    VM_Word sidx = VM_Word.fromInt(sb - scopeBaseIndex);
    if ( !sidx.uLessThan(scopeBlocks))   fail();
    // fail if we are trying to store a scope location into heap/immortal.
    Area sa = scopeOwner[sidx.asInt()];
    if (sa == ta)   return;
    if ((ta.prange - sa.crange) & MASK) != RES)   fail();
    // fail if the location being assigned to is not in a parent/same scope.
}

// Read barriers are always inlined.
void readBarrier(VM_Address src)
        throws PragmaInline, PragmaNoBarriers, PragmaNoPollcheck {
    if (!doLoadCheck)   return;
    // doLoadCheck is true if the running thread is a NHRT
    if ( src.diff(heapBase).uLessThan(heapSize))   fail();
    // fail if the object is in the heap.
}
```

Fig. 7. Implementation of Store and Load checks in Ovm. Like most of the virtual machine the code is expressed in Java with some extensions to perform low-level operations. In this case, the type VM_Address and VM_Word are special types that are compiled to native pointer operations.

`MemoryManager` is an interface with multiple implementations. For example, in configurations tuned for throughput, the memory manager object can serve as glue for MMTk. In the RTSJ configuration, we currently use our own *Split Region Mostly Copying* garbage collector, which implements a conservative semi-space algorithm with pinning collector for lower priority threads and provides region-based memory management for no-heap threads. The name *Split Region* refers to the fact that memory is split into heap and non-heap parts which are managed separately, thereby allowing scoped memory management

to occur even when the garbage collector is running.

3.6 Region Based Memory Management and the Executive Domain

Ovm being written in Java, the most natural approach to memory management would be to rely on the very gabage collector used to reclaim user objects. However to meet the hard real-time requirements of the RTSJ, the Ovm executive-domain must be able to preempt the collector. We solve this by using memory regions. In fact, Ovm's implementation of `javax.realtime.MemoryArea` is based on the same region API that the executive-domain uses for its own memory management.

The primary goal of using regions is to ensure timely completion of executive-domain operations. However, two additional goals can be identified. First, performing a executive-domain call should not leak memory into the caller's scope, with the only exception being for operations that are commonly understood to require allocation (such as expanding a monitor, or allocating a file descriptor for I/O). Second, system operations should never fail due to scope checks.

Consider the following examples of system operations where these goals pose non-trivial problems: reflection, thread scheduling, and POSIX I/O emulation.

CLASSPATH implements Java reflection routines such as `Class.getDeclaredMethods` by using lazy initialization. The `Class` object contains initially null pointers to arrays of methods, constructors, and fields. Invoking a method such as `getDeclaredMethods` may cause a new array to be allocated and stored into the `declaredMethods` field. Hence, if the first time that this method is executed is from a scope, this naive implementation would lead to an illegal assignment error. To solve this problem, we must either arrange to run the initialization code in the appropriate memory area, or else guarantee that initialization happens eagerly.

Lazy initialization is used throughout Java libraries. It is often essential to getting acceptable performance. As such, whatever solution is used to solve the initialization problem must be simple and clean, so that it can be reused throughout the virtual machine and its libraries.

Thread scheduling poses a unique set of problems. In Ovm, much of the state of threads is allocated using regular Java allocation routines. This includes the queue nodes that represent threads within the scheduler. Consider that a thread is allocated within a scope. Then consider the seemingly simple operation of placing the thread onto the ready queue. The ready queue is a linked list. Hence, there is likely to be a pointer to a thread's queue

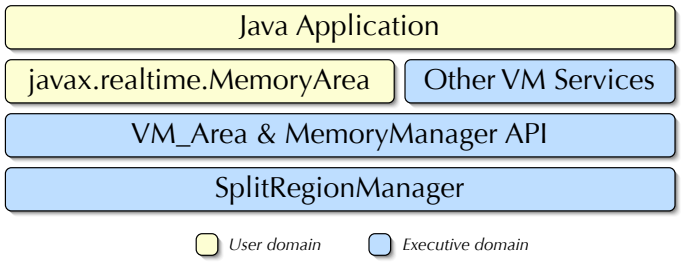


Fig. 8. Ovm memory region stack. Java applications use memory regions directly via the `javax.realtime.MemoryArea` API as well as indirectly when accessing other VM services.

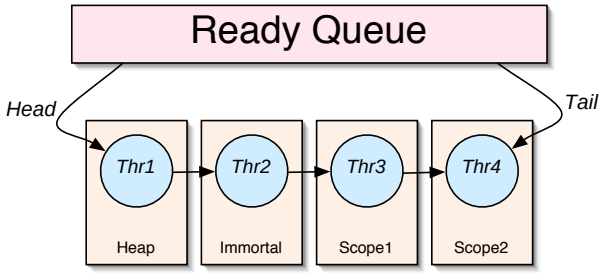


Fig. 9. Ovm thread ready queue with four threads allocated in four different memory areas. Notice that three of the references are illegal: *Thr2*'s reference to *Thr3* is illegal because immortal objects cannot point at scoped objects, *Thr3*'s reference to *Thr4* is only legal if Scope2 is a descendant of Scope1, and the ready queues *Tail* pointer to *Thr4* is illegal, since the ready queue must be immortal.

node from a node allocated in a different scope. Done naively, this would result in an illegal assignment error as soon as a scope allocated thread becomes ready.

Figure 0**??** shows an example of the ready queue problem. In RTSJ, the queue presented in this figure could not have been created. As noted in the figure, two of the references (from *Thr2* to *Thr3* and from the queue object to *Thr4*) cannot be legal, and a third reference (from *Thr3* to *Thr4*) would only be legal if we assume a certain scope hierarchy.

This particular example uncovers one of the great problems of the RTSJ's reference rules. The goal of these rules is to prevent dangling pointers, But in our example, the pointers in the linked list ready queue would never dangle, since for a scope to be reclaimed, all threads in it must terminated, and for a thread to terminate it must be removed from the ready queue. Hence, ready queue pointers will never point at a non-existant object.

Finally, consider what needs to be done to make the `write` call in Fig 6 scope safe. The main actors in this operation are the thread, the memory buffer being written, the I/O descriptor being written to, and temporary objects such as the *BlockingCallback*. In the implementation originally presented, every write operation would leak temporary objects into the caller's memory area. If we are operating in the heap, this is fine, since the garbage collector will clean up these temporary objects. But if scoped memory is being used, this leak puts an undue burden on the user: suddenly, the user must bound the number of I/O operations performed in a scope to prevent out-of-memory conditions. We believe that such a requirement would be unreasonable. For this reason, we introduce a per-thread memory area known as the *scratch pad* for allocating temporary objects.

But this does not solve all of the problems. Consider that the Async I/O framework has its own scheduler, not unlike the thread scheduler. The same exact linked list management problem occurs here. But by introducing the scratch pad, we have actually made this problem worse: now, even if the user does not use scoped memory, the Async I/O code may cause illegal assignment errors due to scratch pad usage.

Furthermore, the `write` call is not the only place where the scratch pad needs to be used. Hence, whatever idioms are used for the scratch pad, they must be easy to reuse.

It should be clear from these examples that to achieve our original goals (preventing leaks and illegal assignment errors), we need a clear, simple, and reusable facility for quickly entering the appropriate memory area for a given operation. Further, we see a need a facility for eliding scope checks.

Name	Description
Exception Safe Area	An area from which it is safe to throw an exception. Dynamically selected to be either the heap or immortal depending on the parent area.
Monitor Area	The area in which we allocate monitors. This is selected dynamically depending on the monitor's parent object.
Mirror Area	Area used to allocate objects referenced by system objects, such as objects of class `Class`. This area is used for allocating both user-level and executive-domain objects. As such, routines to enter and exit this abstract memory area are exported from the executive-domain.
Meta-data Area	Memory area for static data, such as `static` fields, `Class` objects, etc.
Interned String Area	Memory area used for string interning.
Class Init Area	Area in which static initializers are run.
Repository Query Area	Area in which to allocate temporary objects when performing queries for type information.
Repository Data Area	Area in which internal type information, as well as bytecode for all methods, is stored.
Scratch Pad Area	Per-thread area used for temporary data.

Fig. 10. Abstract memory areas that the Ovm executive-domain is aware of. Whenever the executive-domain performs an operation that may require changing memory areas, it requests the appropriate area using the *MemoryPolicy* API. As such, these areas do not have a one-to-one correspondence to real memory areas. Instead, the appropriate memory area is often selected dynamically.

Implementing these facilities is further complicated by the Ovm's need for configurability. In the RTSJ configuration, it is clear that memory areas must be used to meet the user's requirements. But in a regular Java configuration, we would like to have that same code use the heap. Even within the RTSJ configuration, it may be desirable to introduce multiple sub-configurations that differ in the arrangement of memory areas. To make this possible, we created an API known as the *MemoryPolicy* that provides abstract memory areas to the executive-domain. Whenever executive-domain code wishes to perform an operation that may require changing memory areas, it requests the appropriate abstract memory area. The *MemoryPolicy* responds by giving a concrete memory area. Fig. 10 shows a table of abstract memory areas that the Ovm executive-domain is aware of. Depending on the choice of memory management policy, these abstract areas may be linked to different actual areas. For example, a non-real-time configuration may link all of these areas to the heap. Some areas, like the repository query area and the scratch pad area, are typically the same.

To make using memory areas in the executive-domain as simple and reusable as possible, we have created our own memory API for use in the executive-domain and system libraries. This API trades in safety for increased power by introducing the following features: first, we streamline the process of entering memory areas; second, we introduce thread-local scoped memory areas; and third, we make it easy to elide scope checks. These features are discussed in detail below.

First, we change the way that memory areas are entered. Instead of passing an *Runnable* callback to an *enter* method, we have methods for directly setting the current area. See Fig 11 for a code snippet that illustrates this approach.

Not having to allocate a callback whenever a memory area is to be entered reduces the likelihood of memory leaks. On the other hand, it requires the programmer to follow the *try/finally* idiom. We also introduce an explicit call to reset the area. It is up to the user to decide when this call is made, which further increases the utility of memory areas.

Second, we have thread-local memory areas. If a thread requires a temporary memory

```
Object oldArea = setCurrentArea(area);
try {
    ... // perform operations inside the area
} finally {
    setCurrentArea(oldArea);
    area.reset();
}
```

Fig. 11. Entering a memory are using the Ovm Executive Domain memory area API.

area, it does not need to retrieve one from a pool, or risk using one that another thread is using. Instead, each thread has memory areas attached to it, so when that thread needs some temporary memory, it can use its own private area.

Third, scope checks can be elided using a special pragma. This allows parent scopes to contain pointers into child scopes. Although potentially dangerous, such pointers pose no problems as long as they are managed with care. As mentioned previously, we have found numerous cases where eliding scope checks was necessary to make the code work; without this feature, we would have had to resort to object pools.

3.7 Implementing Priority Inheritance Monitors

The RTSJ enriches the semantics of Java monitors with monitor control policies. The default policy is the Priority Inheritance Protocol (PIP) [Sha et al. 1990; Locke et al. 1988] and, optionally, implementations of the RTSJ may provide Priority Ceiling Emulation [Goodenough and Sha 1988]. Ovm has built-in support for PIP monitors. In this section we discuss an implementation approach for the PIP within a real-time Java environment. One of the main departures from previous implementations of the basic priority inheritance protocol, such as [Borger and Rajkumar 1989], is that RTSJ allows threads to change their base priority dynamically. This feature impacts the implemantion of PIP monitors as they must accurately reflect any changes in priorities of blocked threads.

3.7.1 *Basic Concepts and Definitions.* We start by summarizing some of the key concepts of an implementation of PIP:

—The *basePriority* of a thread is the priority assigned to the thread by the application program. In terms of the RTSJ, this is the priority defined in the PriorityParameters object bound to the realtime thread and which we assume can be changed at any time.

—The *activePriority* of a thread is its current execution priority as seen by the scheduler. This is the priority value used to establish execution eligibility and to order the thread on any system queues that are ordered by priority (such as monitor lock acquisition queues, and monitor wait-set queues).

—The *owner* of a monitor is the thread that currently holds the monitor lock.

—The *lockSet* of a monitor is the set of all threads blocked trying to acquire that monitor (ordered by active priority). The *top* of the lock-set is the thread with the highest active priority.

—The *lockSet.top* thread bequests its priority to the monitor owner. If the bequested priority is greater than the owners active priority then the owner inherits the bequested

priority as its active priority. The *lockSet.top* thread is known as a *priority source* for the monitor owner.

—The *ownedSet* is the set of monitors owned by a thread.

—The *inheritanceQueue* of a thread is the ordered set of priority sources for that thread. The top of the inheritance-queue is the thread with the highest active priority.

3.7.2 *Properties and Invariants.* A correct implementation of PIP monitors must maintain the following invariants.

—**Invariant 1:** $\forall\, t \in threads, t.activePriority \geq t.basePriority.$

—**Invariant 2:** $\forall\, t \in threads,$
$t.activePriority = max(t.basePriority, t.inheritanceQueue.top.activePriority).$

—**Invariant 3:** $\forall\, t \in threads, \forall\ monitor\ m \in t.ownedSet,$
$t.inheritanceQueue.contains(m.lockSet.top).$

—**Invariant 4:** *a thread can exist in only one lockSet at a time, and thus in only one inheritance queue.*

—**Invariant 5:** $\forall\, m \in monitors, m.owner.activePriority \geq m.lockSet.top.activePriority.$

—**Invariant 6:** $\forall\, t \in threads, t.ownedSet.size() \geq t.inheritanceQueue.size().$

Furthermore, it will be the cases that threads are started and terminated in consistent states.

—**Property:** *when a thread t is started:* $t.ownedSet.size() = 0$
$\wedge\ t.inhertanceQueue.size() = 0\ \wedge\ t.activePriority = t.basePriority.$

—**Property:** *when a thread t terminates:* $t.ownedSet.size() = 0 \wedge$
$t.inhertanceQueue.size() = 0\ \wedge\ t.activePriority = t.basePriority.$

3.7.3 *Basic Operations.* There are four actions that affect the operation of the priority inheritance protocol:

(1) A thread blocks trying to acquire a monitor (either directly through entry to a synchronized method or statement, or indirectly when returning from a call to Object.wait()) and so may become a priority source for the owning thread.

(2) A thread moves to the top of the lockSet for a monitor (because the previous top thread has either acquired the monitor or abandoned its attempt) and so becomes a priority source for the current owner.

(3) A thread releases a monitor lock (and so loses the priority source from that monitor).

(4) A thread has its priority changed. Depending on the state of the thread this might cause it to become a priority source, or cease to be a priority source; or simply require a change to the active priority of the thread for which it is a priority source.

In all cases correct operation simply involves maintaining the invariants that were previously listed, for all threads. We define two helper functions to express the basic actions that must occur in each case: maintainPriority and propagatePriority.

MaintainPriority: Causes a thread to check that it's active priority invariant is met, and if not to change its active priority so that the invariant is met. An implementation can optimise things by checking for actual changes in active priority.

PropagatePriority: For a thread t this has the following effect:

if t *blocked acquiring monitor* m **then**
  $m.lockSet.reposition(t)$; // *ensure the lockSet is correctly ordered*
  if $m.lockSet.top \neq$ **old** $m.lockSet.top$ **then**
    $m.owner.inheritanceQueue.remove($**old** $m.lockSet.top)$
    $m.owner.inheritanceQueue.insert(m.lockSet.top)$
    $m.owner.maintainPriority()$
    $m.owner.propagatePriority()$
  else if $t = m.lockSet.top$ **then**
    $m.owner.inheritanceQueue.reposition(t)$
    $m.owner.maintainPriority()$
    $m.owner.propogatePriority()$
else if t **is** *runnable/running* **then**
  reorder ready queue

3.7.3.1 *Monitor Acquisition.* If a thread t tries to acquire a monitor m and that monitor already has an owner other than t, then t is placed in the lockSet of m and the following occurs:

if $t = m.lockSet.top$ **then**
  $m.owner.inheritanceQueue.remove($**old** $m.lockSet.top)$
  $m.owner.inheritanceQueue.insert(t)$
  $m.owner.maintainPriority()$
  $m.owner.propagatePriority()$

When t eventually acquires the monitor then the following happens:

$t.inheritanceQueue.insert(m.lockSet.top)$
$t.maintainPriority()$
$t.propagatePriority()$

3.7.3.2 *Monitor Release.* When a thread t releases a monitor m, such that t is no longer the owner of m, then the following occurs:

$t.inheritanceQueue.remove(m.lockSet.top)$
$t.maintainPriority()$
$t.propagatePriority()$

3.7.3.3 *Priority Change.* If a thread t has its priority changed to a value p then the following occurs:

$t.basePriority = p$
$t.maintainPriority()$
$t.propagatePriority()$

The Ovm framework currently does not provide an implementation of PCE monitors. In Ovm, it is possible to implement PIP monitors as either fat- or thin-locks [Bacon et al. 1998], the choice is a configuration option of the virtual machine.

4. OVM 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.

4.1 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.

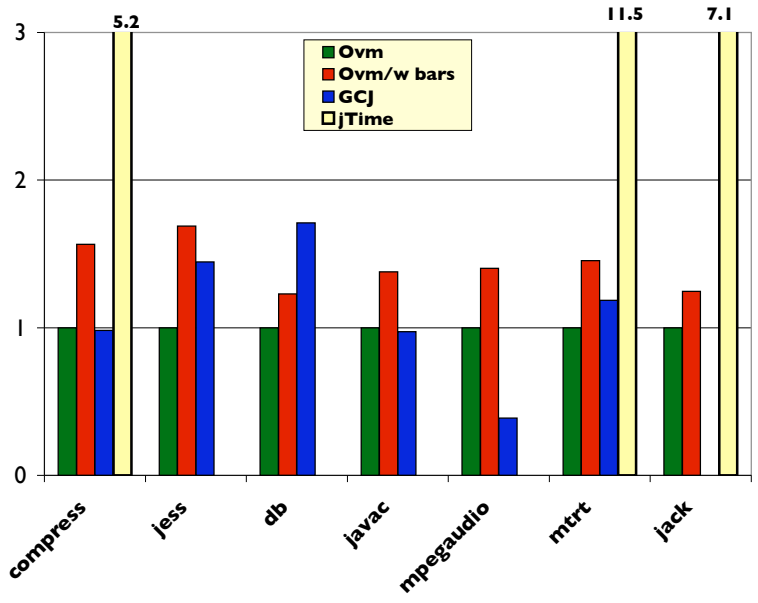


Fig. 12. SpecJVM98. (normalized wrt. Ovm)

The results, given in Fig. 12 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.

4.2 Startup Latency.

We measure the startup time of Ovm on a 300MHz PPC. Fig. 13 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).

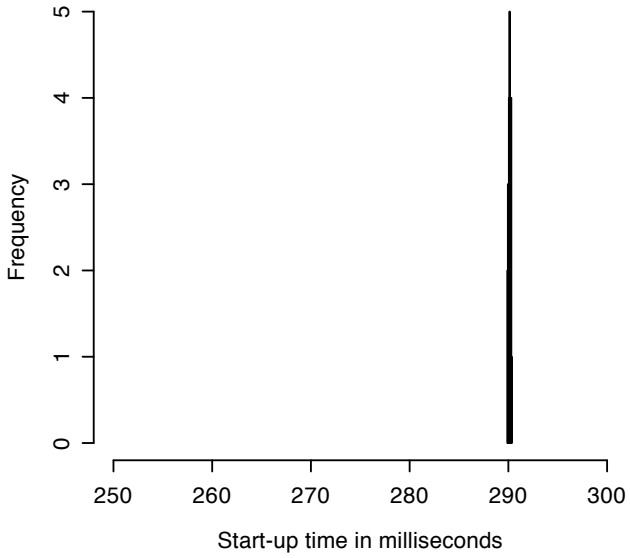


Fig. 13. Ovm Startup Latency.

4.3 Boeing Latency Benchmarks.

Early on in the project Boeing developed a number of latency benchmarks to compare implementations of the RTSJ [Sharp et al. 2003]. Fig. 14 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 is scheduled, it gets the current time. We are interested in measuring the time interval between these times as it approximates the preemption latency.

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.

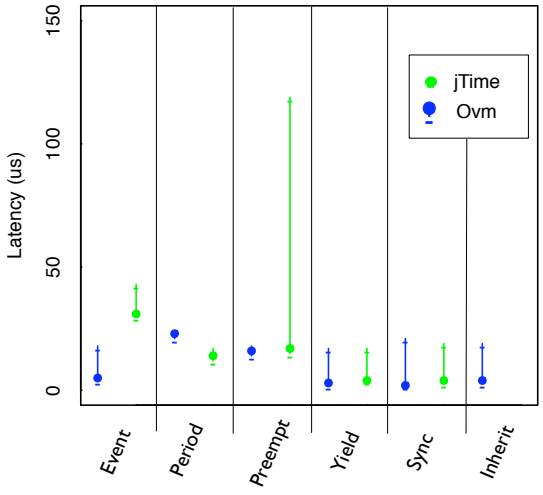


Fig. 14. Boeing RTSJ Latency benchmarks.

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.

4.4 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. 15 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.

To estimate the impact of poll-checks on throughput, we run Ovm on SpecJVM98 with all poll-checks deactivated. See Fig. 16 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.

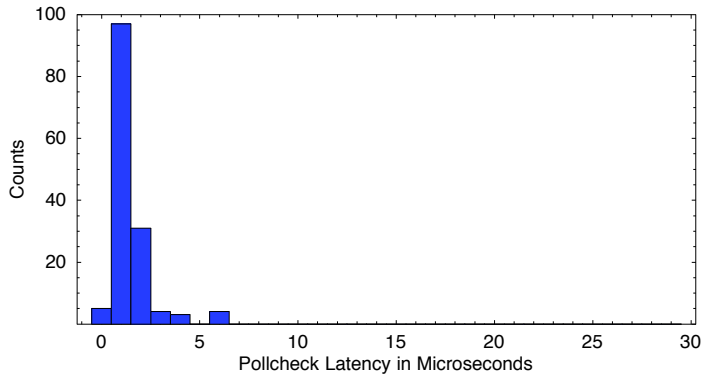


Fig. 15. Distribution of poll check latencies for the PRiSMj 100X scenario. Poll-check latency is the time between an interrupt and a poll-check that services that interrupt.

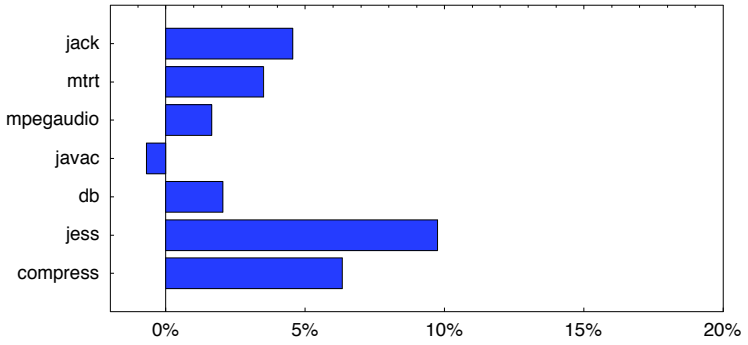


Fig. 16. Percent overhead of poll checks in SpecJVM98 benchmarks. In this graph, 0% overhead indicates that enabling poll checks did not slow down the benchmark.

4.5 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. 17 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.

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. 18. 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

	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		

Fig. 17. Footprint. Lines of code computed overall all Java sources files (w. comments). Data/Code measure the executable Ovm image for PPC.

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%)

Fig. 18. Impact of compiler optimizations.

Finally, Fig. 18 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.

4.6 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. 19 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.

5. 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

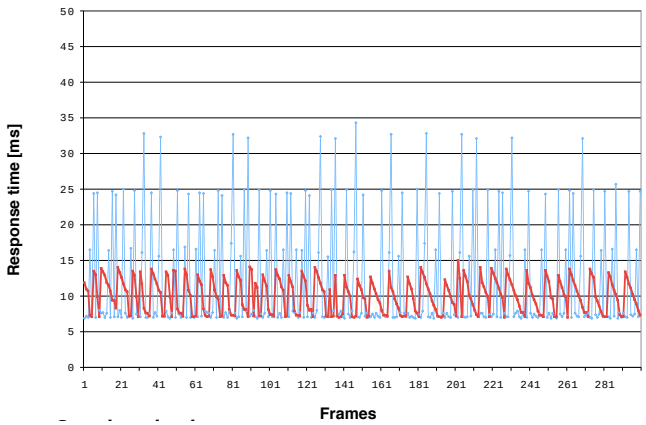


Fig. 19. RT-Zen Results. Comparing the response time for an application running on top of a RTSJ CORBA ORB. Two thread groups (low and high) handle 300 requests each. The y-axis indicates the time to process a request.

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.

5.1 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.

5.2 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.

One reason for delaying the implementation of Protocol Ceiling Emulation is the added complexity of having to support both PCE and PIP in the same program. Implementation issues aside, reasoning about programs with mixed protocols seems difficult.

Another question is which of fat- or thin-locks [Bacon et al. 1998] should be the default in a real-time virtual machine. Thin locks provide much greater throughput but at the cost of predictability. While the worst case execution time of locking is sensibly the same, it is conceivable that programs perform very differently from one run to the next. So for the sake of a simpler performance model, the default synchronization mechanism is based on fat-locks. Thin-locks remain available for application where higher throughput is required. As for their space requirements, both kinds of locks require the same data structure. The difference is that fat-locks are allocated the first time the lock is acquired, while thin-locks are only allocated on blocking.

5.3 Scoped Memory

5.4 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.

5.5 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.

6. 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 [Roll 2003], 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. 20. 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. 21 illustrates the flight configuration.

6.1 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. 22. The 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

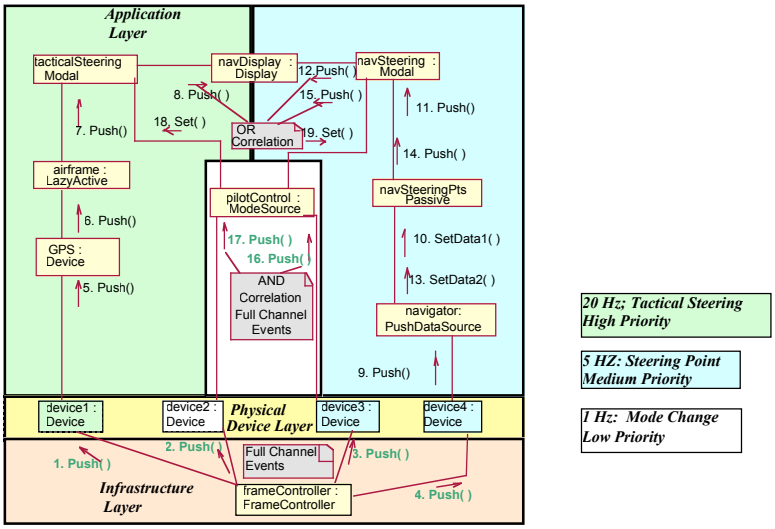


Fig. 20. The overview of the Boeing PRiSMj 1X scenario.

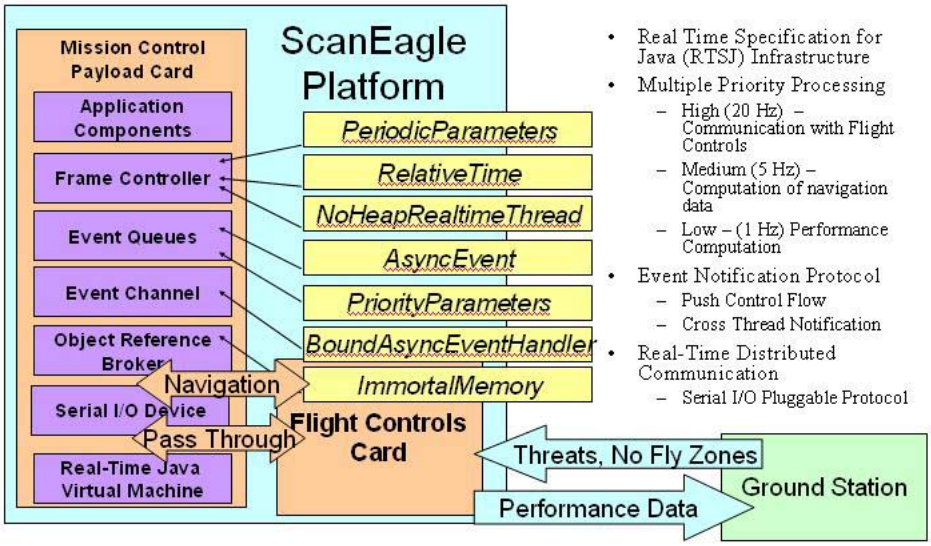


Fig. 21. ScanEagle Flight Product Scenario RTSJ Architecture.

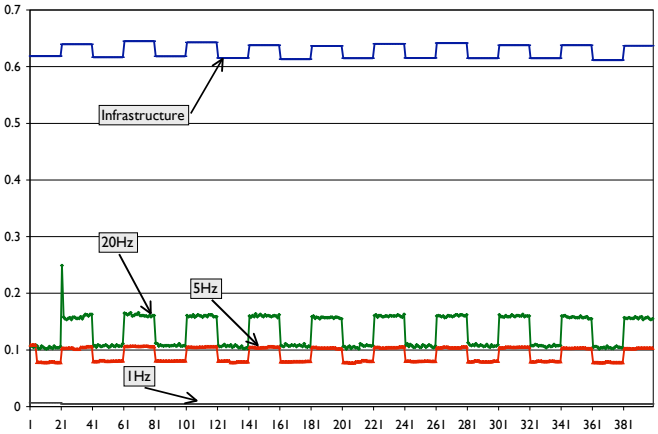
500 microseconds. The jitter measured at approximately 100 microseconds during the 50 millisecond period. This was well within the 1% success criteria. The results are illustrated in Fig. 22.

7. SCANEAGLE FLIGHT DEMONSTRATION

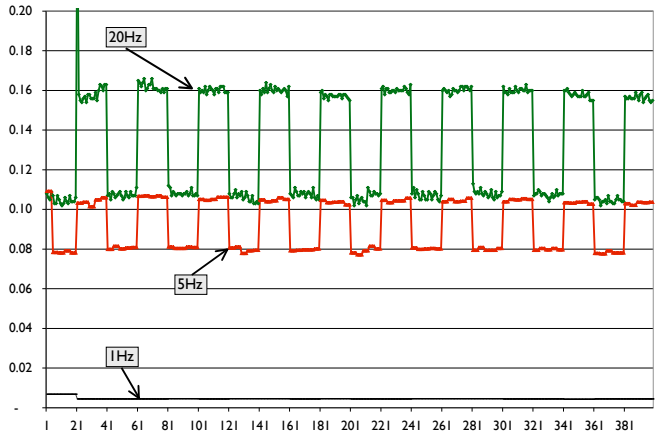
Ovm was used as the Java Virtual Machine for the Real-Time Java Open Experiment Platform in demonstrations at Chicago in June 2004; St. Louis, for a ground demonstration in December 2004; and White Sands Missile Range, NM, for the capstone demonstration in April 2005.

7.1 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 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



(b) PRiSMj 100x workload.



(a) PRiSMj 1x workload.

Fig. 22. 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)

at a periodic rate of 1Hz. The flight product scenario is illustrated in Fig. 21. 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.

7.2 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.

7.3 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.

7.4 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.

7.5 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++ development. 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.

8. 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, Bin Xin, 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.

REFERENCES

BACON, D., KONURU, R., MURTHY, C., AND SERRANO, M. 1998. Thin locks: Featherweight synchronization for Java. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 258–268.

BEEBEE, JR., W. S. AND RINARD, M. 2001. An implementation of scoped memory for Real-Time Java. *Emsoft - LNCS 2211*, 289–305.

BOLLELLA, G., GOSLING, J., BROSGOL, B., DIBBLE, P., FURR, S., AND TURNBULL, M. 2000. *The Real-Time Specification for Java*. Addison-Wesley, Reading, MA, USA.

BORGER, M. AND RAJKUMAR, R. 1989. Implementing priority inheritance algorithms in an Ada runtime system. Tech. Rep. CMU/SEI-89-TR-15, Software Engineering Institute, Carnegie Mellon University. April.

BUYTAERT, D., ARICKX, F., AND VOS, J. 2002. A profiler and compiler for the Wonka Virtual Machine. In *USENIX JVM'02 Work in Progress, San Francisco, CA*. USENIX, Berkeley, CA.

CORSARO, A. AND SCHMIDT, D. C. 2002. The design and performance of the jRate Real-Time Java implementation. *Lecture Notes in Computer Science 2519*, 900–921.

DVORAK, D., BOLLELLA, G., CANHAM, T., CARSON, V., CHAMPLIN, V., GIOVANNONI, B., INDICTOR, M., MEYER, K., MURRAY, A., AND REINHOLTZ, K. 2004. Project Golden Gate: Towards Real-Time Java in Space Missions. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004), 12-14 May 2004, Vienna, Austria* (12–14). IEEE Computer Society Press, Silver Spring, MD 20910, USA, 15–22.

GLEIM, U. 2002. 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*. USENIX, Berkeley, CA, USA.

GOODENOUGH, J. B. AND SHA, L. 1988. The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks. *ACM SIGADA Ada Letters 8*, 7 (Fall), 20–31.

LOCKE, D., SHA, L., RAJKUMAR, R., LEHOCZKY, J., AND BURNS, G. 1988. Priority inversion and its control: An experimental investigation. *ACM SIGADA Ada Letters 8*, 7 (Fall), 39–42.

NILSEN, K. 1998. Adding real-time capabilities to Java. *Communications of the ACM 41*, 6 (June), 49–56.

PALACZ, K., BAKER, J., FLACK, C., GROTHOFF, C., YAMAUCHI, H., AND VITEK, J. 2005. Engineering a common intermediate representation for the Ovm framework. *The Science of Computer Programming 57*, 3 (September), 357–378.

PALACZ, K. AND VITEK, J. 2003. Java subtype test in real-time. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP03)*. Lecture Notes in Computer Science. Springer-Verlag, Darmstadt, Germany, 378–404.

PURDUE UNIVERSITY - S3 LAB. 2005. The Ovm Virtual Machine homepage, `http://www.ovmj.org/`.

ROLL, W. 2003. Towards model-based and ccm-based applications for real-time systems. In *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003), 14-16 May 2003, Hakodate, Hokkaido, Japan*. IEEE Computer Society Press, Silver Spring, MD 20910, USA, 75–82.

SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput. 39*, 9 (Sept.), 1175–1185.

SHARP, D. C. 2001. Real-time distributed object computing: Ready for mission-critical embedded system applications. In *Proceeding of the 3rd International Symposium on Distributed Objects and Applications, DOA 2001, 17-20 September 2001, Rome, Italy*. IEEE Computer Society Press, Silver Spring, MD 20910, USA, 3–4.

SHARP, D. C., PLA, E., LUECKE, K. R., AND II, R. J. H. 2003. Evaluating Real-Time Java for mission-critical large-scale embedded systems. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003), May 27-30, 2003, Toronto, Canada*. IEEE Computer Society Press, Silver Spring, MD 20910, USA, 30–36.

SIEBERT, F. 1999. Hard real-time garbage collection in the Jamaica Virtual Machine. In *Proceedings of the 6th International Workshop on Real-Time Computing and Applications Symposium (RTCSA '99), 13-16 December 1999, Hong Kong, China*. IEEE Computer Society Press, Silver Spring, MD 20910, USA.

TIMESYS INC. 2003. The jTime Virtual Machine, `http://www.timesys.com/`.

TRYGGVESSON, J., MATTSSON, T., AND HEEB, H. 1999. Jbed: Java for real-time systems. *Dr. Dobb's Journal of Software Tools 24*, 11 (Nov.), 78, 80, 82–84, 86.