A Real-Time Java Virtual Machine with Applications in 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 Prochazka
SciSys
and
Jan Vitek
Purdue University

1. INTRODUCTION

The Real-Time Specification for Java (RTSJ) [Bollella et al. 2000] was designed for the construction of large-scale Real-time Embedded systems [Sharp 2001; Dvorak et al. 2004]. The RTSJ 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 large scale systems are being developed in Java, e.g.

This work was supported under a DARPA PCES contract, NSF 501-1398-1086 and NSF CSR-AES 501-1398-1588. A preliminary version of this paper appeared in the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006).

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

for avionics, shipboard computing and simulation [Child 2004; 2003; Benowitz and Niessner 2003a; Sharp et al. 2003]. 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. A number of research projects are working on open source implementations [Timesys Inc 2003; Corsaro and Schmidt 2002a; Purdue University - S3 Lab 2005; Nilsen 1998; Buytaert et al. 2002; Tryggvesson et al. 1999; Gleim 2002; Siebert 1999]. This paper discusses our implementation of the RTSJ in the Ovm customizable virtual machine framework and its use within the DARPA PCES project.

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. Boeing and Purdue University demonstrated autonomous navigation capabilities on an Unmanned Aerial Vehicle (UAV) known as the ScanEagle (Fig. 1).

The ScanEagle 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 avionics systems. These product scenarios contain representative component configurations and interactions.

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 flew in tests conducted in April 2005. The system was awarded a 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. Overall, the Real-time Java Specification has proven to be more than adequate for the tasks at hand and, considering the multiple design constraints placed on the RTSJ, it represents a good engineering compromise.

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 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 by using 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 [Brosgol et al. 2002]. This facilitates early termination while preserving the safety of code that does not expect such interruptions.

Additional introductory material includes the evaluation of the expressive power of the RTSJ done by Wellings and Puschner [Wellings and Puschner 2003] and a number of other papers by Wellings.

3. THE OVM VIRTUAL MACHINE

Ovm is a generic framework for building virtual machines with different features. It supports components that provide a wide variety of alternate implementations of core VM features. 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 and good predictability 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 threads 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 the Ovm intermediate representation and bytecode editing and analysis framework as well as to [Flack et al. 2003] for a discussion of idioms used to express unsafe operations within Ovm.

3.1 System Architecture

The overall architecture of Ovm consists of an *executive domain* (ED) core, which serves as the kernel of the virtual machine, around which multiple *user domain* (UD) "personalities" can execute. The executive domain consists of a 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 name space. Although the executive domain is written in Java, it is not subject to regular Java semantics (as described in Section 3.4). We use a set of idioms known to our compiler, in the form of pragmas and intrinsic methods, to allow ED code to perform type-unsafe operations such as pointer arithmetic and unchecked stores within scoped memory. As the executive domain is in a separate name space, its notion of basic `java.lang` classes such as `Object`, `String`, and `Throwable` is different from those defined in the UD. For instance, the representation of strings within the ED need not be identical to that of Java Strings in the UD.

Fig. 2 illustrates the architecture of the Ovm, from the low-level kernel components up to the GNU Classpath library, which we use as the standard library. The executive domain implements the core functionality of Java, such as monitors, memory allocation, 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 how to translate these instructions into appropriate executive domain method calls. This is achieved via a glue layer called the `CoreServicesAccess` (CSA). For example, instructions such

as MONITORENTER or ATHROW are translated 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 call into the CSA.

Domains in 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 LibraryImports. The Ovm compiler recognizes UD 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 method calls in the GNU Classpath [FSF 2005] library are translated into calls to regular Java methods which then use library imports to access system functionality. This process of translating user domain native methods is called LibraryGlue. 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.

3.2 Assumptions and Requirements

The implementation of Ovm was driven by the requirements of our main client, the PRiSMj application. We give some of the main assumptions here.

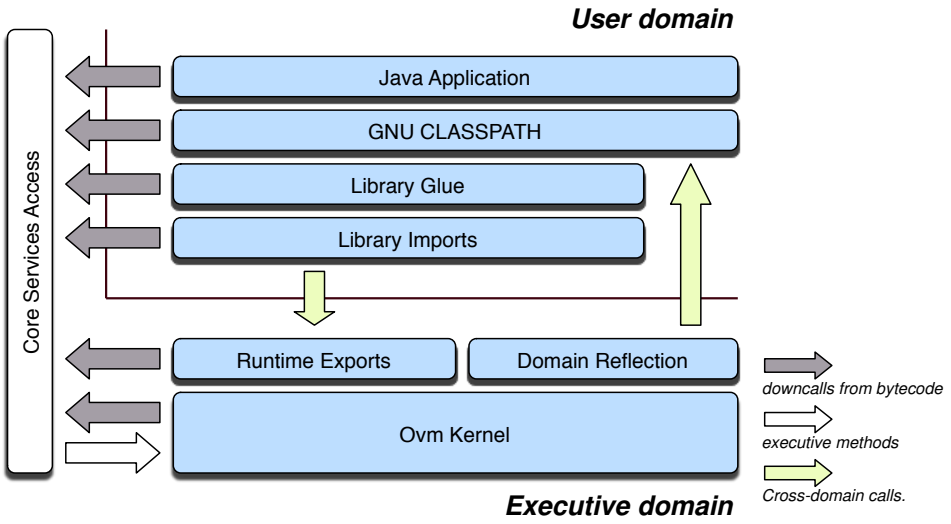


Fig. 2: Overview of the Ovm architecture. The virtual machine consists of two domains, an executive domain (ED) and a user domain (UD). The interaction between the domains is mediated by the CoreServicesAccess class. It is defined in ED and accessed from the UD either by direct calls inserted by the compiler or by calls to a distinguished LibraryImports class. Standard Java libraries are provided by the GNU CLASSPATH project.

The current Ovm implementation assumes a single operating-system thread which means that it will not be able to take advantage of multi-core/multi-processor systems. This was not deemed to be a serious limitation in the target application domain and we plan to lift the restriction in future implementations. The main RTSJ configuration is based on an optimizing ahead-of-time compiler and does not include a just-in-time compiler. This means that we assume that applications will not request dynamic class loading. Furthermore, the compiler requires that reflection targets be declared at virtual machine construction. While this restriction is not strictly necessary, it does lead to better optimization and higher performance. We assume that the GNU GCC compiler is available for the target platform. This has not proven a problem so far and has given us a degree of hardware portability. Ovm has very little dependence on the operating system as it does its own scheduling, synchronization and preemption. We assume that sufficient memory is available (at least 64MB). Optimizing the virtual machine for space has not been a requirement for the avionics application and was not a priority. Ovm routinely supports task rates of 1KHz. We are currently experimenting with rates of 22KHz.

3.3 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.

3.4 Java in Java

Ovm is implemented in Java, with extended semantics to express low-level operations, and 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 150,000 lines of Java code and 15,000 lines of C code.

In order to express low-level operations which are outside of the boundaries of the Java language, such as object allocation and initialization, garbage collection, direct memory access, it is necessary to extend Java semantics. We do this by defining a number of idioms that are recognized by the Ovm compiler and translated into efficient low-level operations. When this is combined with static analysis to remove some overheads such as dynamic binding, the resulting code is close to what one would write in a C++ virtual machine. We illustrate the process with a prototypical example of low-level behavior – the bump pointer allocator used to allocate memory in scoped memory areas. Fig. 3 gives the Java code for the allocation routine. The operations on the `VM_Address` class have no meaning in Java; in fact we will never allocate instances of `VM_Address`. Instead this class will be translated

```
VM_Address getMem(int size)
      throws PragmaNoPollcheck, PragmaNoBarriers {
   VM_Address ret = base().add(offset);
   offset += size;
   Mem.the().zero(ret.add(ALIGN), offset == rsize?size-ALIGN:size);
   return ret;
 }
```

Fig. 3: Java implementation of `getMem()`, the bump pointer allocator in class `TransientArea`. Method `base()` returns the base pointer in area and the field `offset` is the number of allocated bytes. This allocator is used by scoped memory areas and ensure allocation times linear in the size of the allocated object (due to zeroing). Notice the use of the `VM_Address` types to represent native memory locations.

```
 static VM_Address* getMem(TransientArea* area, jint size){
   jint s1 = area + area->offset;
   area->offset += size;
   jint s2 = s1 + (&SplitRegionManager)->ALIGN;
   jint s3 = (area->offset == area->rsize)?
     (size-(&SplitRegionManager)->ALIGN) : size;
   PollingAware_zero(roots->values[57]), s2, s3);
   return s1;
 }
```

Fig. 4: The C++ translation of the `getMem()` method performed by the j2c ahead-of-time compiler. (Type casts are omitted, and names shortened for readability.) This method is not virtual and can be inlined by the GCC backend. The receiver object is made explicit in the translation as an additional argument to the method. Address operation are performed by pointer arithmetic. The call to the `zero()` method is a statically determined call. In fact, after translation all occurrence of dynamic method invocation have been eliminated.

down to native operations. But, even in this case we still get benefits from writing the code in Java. As `VM_Address` is expressed as a Java class, we can give it methods and have the Java type system make sure that only those methods will be used on instance of that class. Ovm defines a hierarchy of memory related classes: `VM_Word` for basic bit manipulation and `VM_Address` for pointer arithmetic. Ovm also has a class `Oop` which stands for an address that points to the start of an object. Garbage collectors further extend `Oop`. For instance a moving garbage collector would define `MovingGC` as a particular kind of `Oop` that has a field for storing forwarding pointers.

Ovm also supports a number of compiler pragmas, which are expressed in Java as exception types. Methods can be annotated by a pragma by mentioning it in the method's `throws` clause[1]. The main pragmas are given here:

`PragmaInline`. This pragma instructs the Ovm compiler to inline the method. The inlining can currently be done either at the bytecode level while bootstraping the VM or, in the case of ahead-of-time compilation, by the GNU GCC backend. A `PragmaNoInline` is also provided to prevent inlining of certain methods. It is mostly used to avoid code bloat.

`PragmaNoBarriers`. There should be no RTSJ read or write barriers. This pragma is needed for some system operations that run within a region but must manipulate data allocated outside of that region. It is also used to optimize executive domain code.

[1]This design predates Java 5.0. One can now use meta-data to this end.

`PragmaNoPollcheck`. This pragma is used to get 'shallow' atomicity. The compiler will not emit poll-check instructions within the body of the annotated method, but poll-checks can still occur in methods called from it. If real atomicity is required, `PragmaAtomic` will be used. It causes the compiler to emit instructions that turn off scheduling for the duration of the method (including other methods called from it).

`PragmaTransformCall`. This pragma is the parent class of large number of specialized pragmas. They are used to instruct the compiler to replace calls to a method with simpler operations. For instance, in the case of `address.add(offset)`, rather than emitting a method call, the compiler will notice a pragma attached to the `add()` method and replace the invocation with pointer arithmetic.

3.5 User-level Threading

In Ovm, threads are implemented in one process and are run by a single operating system thread. User-level threading decouples the virtual machine from the underlying operating system and allows different configurations of Ovm to offer applications different scheduling and synchronization semantics. For real-time programs, this approach increases portability as the virtual machine can mask differences in primitives provided by different real-time operating systems.

Java threads are represented in the VM by so-called *contexts* which are run by one operating system thread. The preemption model is a form of cooperative-multithreading. Threads relinquish control at special instructions called *poll-checks*. The processing of asynchronous events such as timer interrupts and I/O completion occurs at poll-checks. In order to prevent starvation, poll-checks are inserted automatically by the compiler as shown in Fig. 5. One of the key invariants that comes with the combination of user-level threading and cooperative preemption is that at any given point all but one of the Java-level threads are stopped at a poll-check instruction. This has the advantage that the compiler can ensure that a sequence is atomic by simply omitting poll-check instructions. The policy for emitting poll-checks can be tuned. Typically the compiler will emit poll-checks at the entry of every method and on every back-branch in the control flow graph. The compiler's objective is to guarantee that each thread will eventually hit a poll-check. The cost associated with polling is usually small (see Section 4.4) and can be reduced further by using compiler optimizations. Loop unrolling, for instance, decreases the number of poll-checks by reducing the frequency of back-branches.

In designing our polling scheme we had the following three goals. (*a*) Cheap: A poll-check should not require more than a load and a compare on a single 32-bit word at a well-known location. Register allocating this word would make the check even cheaper. (*b*) Signal-safe: A signal handler may interrupt Ovm at any instruction boundary. Whatever action the signal handler takes must be correct even if the point of interruption was a poll-check. (*c*) Fast dynamic deactivation: `PragmaAtomic` disables interruptions due to poll-checks. Many critical path methods use `PragmaAtomic`. Hence, it should be possible to rapidly disable and re-enable poll-checks.

Ovm uses simple atomic operations over a 32-bit polling word, shown in Fig. 6. The `s.notSignaled` field is one by default, and set to zero whenever a signal occurs. `s.notEnabled` is zero when polling is enabled, and is one when disabled. A poll-check then is a simple matter of comparing `pollWord` to zero. If they are equal, a signal occurred and polling is enabled. The fast path is a load and a compare. The slow path involves disabling

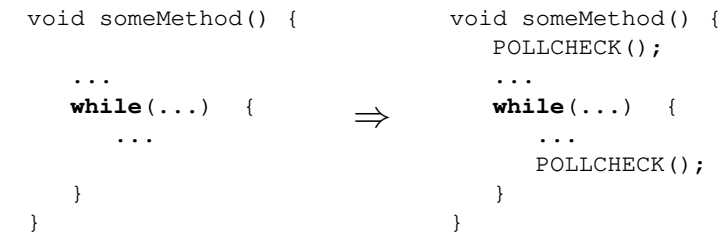


Fig. 5: Compiler inserted cooperative multi-threading. In Ovm rescheduling can only occur at poll-check instructions inserted by the compiler in the bytecode. All code paths must eventually encounter a poll-check instruction. Bounding the number of instructions between poll-checks is a key part in reducing preemption latency.

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


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

Fig. 6: Definition of the 32-bit polling word and the compiler inserted code fragment implementing the check. No synchronization is required as rescheduling can only occur when `handleEvents()` is executed and there is only one thread active at any given time.

checks and clearing the signal, and then entering the event handling code. Polling must be disabled because the event handler may call into common code that was compiled with injected poll-checks.

3.6 Memory Management and the Executive Domain

In a Java-in-Java virtual machine, the most natural approach to memory management is for the executive domain to rely on the very same garbage collection (GC) algorithm used to reclaim user domain objects. However this is not sufficient to meet hard real-time requirements, as the ED must be able to preempt the collector at any time. The ED can thus not rely on GC, or at least not entirely. In Ovm, some critical ED data structures are allocated outside of the reach of the collector. In fact, we use the same code used to implement memory areas within the executive. There is a single instance of `MemoryManager` class which provides an interface to the garbage collector and to memory areas. The ED uses objects of the type `VM_Area` to represent memory areas internally, even for heap and immortal memory.

`MemoryManager` is an interface with multiple implementations. For example, in an Ovm configuration tuned for throughput, the memory manager serves as glue for the

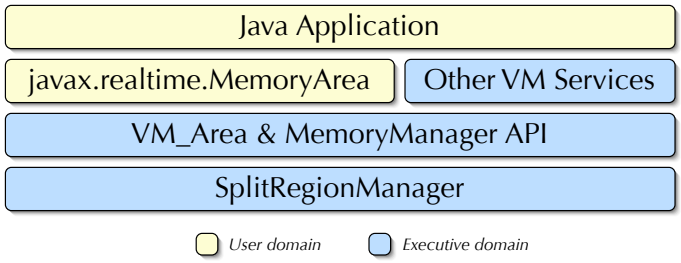


Fig. 7: Ovm memory region stack. Java applications use memory regions directly via the `javax.realtime.MemoryArea` API as well as indirectly when accessing other VM services. The `SplitRegionManager` class provides support for regions and moving garbage collection for heap allocated data.

Name	Description
Exception Safe Area	Used to throw an exception without risk of memory access violation. Dynamically selected to be either the heap or immortal depending on the parent area.
Monitor Area	The area where monitors are allocated is selected dynamically depending on the monitor's owner object.
Mirror Area	This is the area where globally accessible objects, such as instances of class `Class`, are allocated. Used for both UD and ED objects and visible in the UD.
Meta-data Area	Area where data referenced from static fields is allocated.
Interned String Area	Whenever `String.intern()` is called, the string object is placed in this area.
Class Init Area	Area in which static class 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	Thread-local areas used for allocation of short-lived temporary data.

Fig. 8: Memory areas that the Ovm executive domain is aware of. Whenever the ED performs an operation that may require changing memory areas, it requests the appropriate area using the `MemoryPolicy` API. These are logical areas, there need not be a one-to-one correspondence to real memory areas. The appropriate memory area is often selected dynamically.

generic memory management toolkit MMTk [Blackburn et al. 2004]. In an RTSJ configuration, we use our own split region mostly-copying garbage collector, which implements a conservative semi-space collector for lower priority threads and provides region-based memory management for real-time threads. The term 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.

The primary goal of using regions is to ensure timely completion of ED/UD operations. However, two additional goals can be identified. Firstly, performing an ED call should not leak memory into the caller's scope, with the exception of operations that are commonly understood to require allocation (such as expanding a monitor, or allocating a file descriptor for I/O). Secondly, ED operations should never cause a memory access violation, i.e. `MemoryAccessError` or `IllegalAssignmentError` to be thrown.

A requirement for a memory API is to be able to change the effective allocation context efficiently. As Ovm supports many configurations with very different memory management policies, it is important to be able to specify a logical mapping of data to memory independently from its actual implementation. So, for instance, in a RTSJ configuration

some data must be allocated in non-GCed memory, while in a plain Java all data can be allocated in GCed space. Ovm defines an interface, named `MemoryPolicy`, which insulates the source code of the VM from the configuration's memory management implementation. This interface exposes a number of logical allocation contexts, shown in Fig. 8, and hides their physical mapping. A non-real-time configuration may map all of these areas to the heap. Some areas, like the repository query area and the scratch pad area, are usually one and the same.

The ED memory interface is strictly more expressive that the Memory Management interface provided by the RTSJ to user code. First, it streamlines the process of entering memory areas; second, it introduces tread-local areas; and third, it allows scope checks to be elided. Entering an area in ED code does not require a `Runnable` (which is the case for the RTSJ API), instead the allocation context can be set directly as shown in Fig. 9. This can reduce the likelihood of memory leaks and is generally more convenient. On the other hand, it requires disciplined use of the *try/finally* idiom. Thread-local areas allow a thread to allocate temporary objects without having to worry about sharing issues or concurrent access.

3.6.1 *Examples.* The needs for and use of the memory API is illustrated with the implementation of cross-domain calls. A cross-domain call is a method invocation that straddles the ED/UD boundary. Since domains are to be kept disjoint and are in different namespaces cross-domain invocation is implemented with a lightweight form of reflection. In order to invoke a method, the source domain will allocate an `InvocationMessage` object, this object is then handed over to the target domain and executed there. Once the call returns, the source domain can query the message for its return value.

The memory management issues here are related to allocation of the reflective data structures. Consider the case where a cross-domain call is initiated while in scoped memory area (or immortal). Allocating the message object in the current area will cause a memory leak and possibly an out-of-memory error. This is not acceptable as this occurs in a place where the user does not expect allocation to happen. The solution relies on one important property of the cross-domain invocation protocol, namely that the message object becomes garbage after the call returns. The obvious solution is thus to allocate the object in a thread-local scratch pad as shown in Fig. 10. The scratch pad can be cleared when the method returns.

Thread scheduling poses another problem. The state of threads includes queue nodes. Whenever an instance of class `Thread` or one of its real-time subclasses is allocated, all the data structures are created in the current allocation context. If the thread is created while in a scoped memory area, then the seemingly simple operation of placing the thread onto the ready queue becomes error prone as the queue node is also allocated in scoped

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

Fig. 9: Entering a memory area using the ED area API. The call to `setCurrentArea()` changes the allocation context to that of its argument returning the previous area. Once the operation completes, the code switches back to the previous context.

```
public ValueUnion call(Oop recv) {
    VM_Area area = MemoryManager.the().getCurrentArea();
    Object r1 = MemoryPolicy.the().enterScratchPadArea();
    try {
        InvocationMessage msg = makeMessage();
        VM_Area r2 = MemoryManager.the().setCurrentArea(area);
        try {
            ReturnMessage ret = msg.invoke(recv);
            ret.rethrowWildcard();
            return ret.getReturnValue();
        } finally { MemoryManager.the().setCurrentArea(r2); }
    } finally { MemoryPolicy.the().leave(r1); }
}
```

Fig. 10: Scratch Pad. A scratch pad is a thread-local temporary memory area with a clearly delimited lifetime. The `call()` method implements the semantics of a cross-domain call. It starts in the caller's area, `area`, before entering the scratch pad with `enterScratchPadArea()`. The message object is allocated there, then the associated method is invoked and the return value or exception is retrieved. Before returning, the allocation context is restored, `setCurrentArea()`, and scratch memory is reclaimed with `leave()`. The `rethrowWildcard()` method is used to propagate exception which must cross domain boundaries.

memory and thus may cause a memory access violation. The solution here is based on careful scope lifetime management and memory access check elision.

3.7 Scoped Memory and Region Based Memory Management

The RTSJ identifies three different kinds of memory: *heap, immortal* and *scoped memory*. Heap memory has the traditional Java semantics with a garbage collector. Immortal memory is a globally accessible sequence of memory locations used to hold objects that are never reclaimed. Scoped memory supports reclamation of individual scopes. It relies on a reference counting scheme such that when no thread is in 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 is reclaimed, a reference could be left "dangling". The RTSJ prevents this by requiring that all stores to variables be checked at runtime. 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 (see Fig. 26), 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. The Ovm source code is shown in Fig. 13.

Store checks are more complex. 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

```
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);
 }
```

Fig. 11: RTSJ write barrier fast path. `storeCheck()` verifies if the two objects are allocated in the same block. If not, follow the slow path. The method is inserted before every write to a reference field and is always inlined. The variable `blockShift` is used to obtain the block in which the objects are allocated.

```
 void storeCheckSlow(int sb, int tb)
    throws PragmaNoPollcheck, PragmaNoBarriers, PargamNoInline {
    VM_Word tidx = VM_Word.fromInt(tb - scopeBaseIndex);
    if (!tidx.uLessThan(scopeBlocks)) return;
    Area ta = scopeOwner[ tidx.asInt() ];
    VM_Word sidx = VM_Word.fromInt(sb - scopeBaseIndex);
    if (!sidx.uLessThan(scopeBlocks)) fail();
    Area sa = scopeOwner[sidx.asInt()];
    if (sa == ta)  return;
    if ((ta.prange - sa.crange) & MASK) != RES) fail();
  }
```

Fig. 12: RTSJ write barrier slow path. `storeCheckSlow()` fails, throwing an exception, if the assignment would violate RTSJ memory semantics. There are two failure cases: if we try to store a a reference to a scope-allocated object into the heap or immortal, and if we try to store a reference to an object with shorter or disjoint lifetime. This method is never inlined. `MASK` and `RES` are constants described in [Palacz and Vitek 2003].

```
void readBarrier(VM_Address src)
      throws PragmaInline, PragmaNoBarriers, PragmaNoPollcheck {
    if (!doLoadCheck)  return;
    if (src.diff(heapBase).uLessThan(heapSize))  fail();
 }
```

Fig. 13: RTSJ read barrier. `readBarrier()` fails if the current thread is a `NoHeapRealtimeThread` and the target of the reference is a heap location. This code is inserted before every load of a reference field and is always inlined. `doLoadCheck` is true if the current thread requires read barriers.

```
 VM_Area areaOf(Oop mem) {
   VM_Word off = VM_Address.fromObject(mem).diff(heapBase);
   if (off.uLT(VM_Word.fromInt(heapSize))) return heapArea;
   off = VM_Address.fromObject(mem).diff(scopeBase);
   if (!off.uLT(VM_Word.fromInt(scopeSize))) return immortalArea;
   int idx = off.asInt() >>> blockShift;
   return scopeOwner[idx];
}
```

Fig. 14: The `areaOf()` method returns the memory area of an object which can be either heap, immortal or scoped. We keep a mapping from memory blocks to scoped memory instances, `scopeOwner`, to speed up discovery.

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. This tree 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, a scope check boils down to a pair of bound tests. The store check can thus be done in constant time [Palacz and Vitek 2003].

Fig. 11 and Fig. 12 details the Ovm implementation of store checks (fast and slow paths) and Fig. 14 gives the implementation of `areaOf()` operation. The code refers to an auxiliary data structure `scopeOwner` which maps blocks to `Area` objects which are ED representatives of scopes. Variables `crange` and `prange` are allocated as in [Palacz and Vitek 2003].

3.7.1 *Execute In Area.* The RTSJ API provides a way to execute a method in a different memory area with `MemorArea.executeInArea()`. Its semantics is similar to the `get/setCurrentArea()` idiom used within the ED. Due to the intricacies of the RTSJ scope semantics the implementation is more tricky. The call `area.executeInArea(logic)` invokes the `run()` method of `logic` within the scope `area`.

The implementation is given in Fig. 15. Some of the complexity is due to the fact that a thread keeps a stack of scopes that it has entered and this scope stack must be updated to reflect the allocation context switch. This is done by creating and manipulating a copy of the thread's scope stack. But, this copy must be allocated somewhere! To avoid memory leaks, the copy is allocated in a scratch pad that will be torn down when the call returns. Thus, the implementation copies and unnwinds the scope stack to make `area` the current area. It then executes the `logic` and, finally, restores the original scope stack and current allocation context. `makeExplicitArea()` creates a temporary area where the copy of the scope stack will be allocated (the copy is created by `unwindUpToMe()`.

```
void execInArea(Runnable logic) {
  RealtimeThread current = getCurrentThread();
  Opaque temp = LibraryImports.makeExplicitArea(size);
  try {
    Opaque currentMA = LibraryImports.setCurrentArea(temp);
    try {
      ScopeStack oldStack = current.getScopeStack();
      ScopeStack newStack = unwindUpToMe(current);
      try {
        current.setScopeStack(newStack);
        try {
          LibraryImports.setCurrentArea(area);
          logic.run();
        } finally { current.setScopeStack(oldStack); }
      } finally { newStack.free(); }
    } finally { LibraryImports.setCurrentArea(currentMA); }
  } finally { LibraryImports.destroyArea(temp); }
}
```

Fig. 15: Implementing `executeInArea`. The `run()` method of `logic` will be executed in the memory area denoted by `this`. The current real-time thread's scope stack is saved and then unwound to the target area. Upon completion the scope stack is restored.

The RTSJ also provides a method for reflective object allocation in a different scope, `MemoryArea.newInstance(Class klass)`. Its implementation is identical to that of `executeInArea()` with reflective allocation replacing the call to `run()`.

3.7.2 *Scope Boundary Exceptions.* There are subtle ways to cause a memory access violation. One particularly tricky issue arises with exceptions, either those explicitly created by user code or thrown by the virtual machine. Consider a typical throw expression, such as `throw new Error()`. If this expression is evaluated within a scoped memory area, the lifetime of the exception object is tied to that of the scope. If the exception escapes the scope there is a risk of memory corruption – reclaiming the scope will create a dangling pointer to the exception object. Allocating the exception in the heap is not possible as this would make real-time code dependent on the GC and using immortal memory would cause a permanent leak.

The solution adopted in Fig. 16 is to copy exceptions that cross a scope boundary. To be precise an exception crossing a boundary will be converted into a `ThrowBoundaryError`. To do this, it is necessary to step out of the current memory area and allocate an error in the outer area. The `ThrowBoundaryError` instance and the string value constructed from the original throwable must be allocated in the outer memory area. The implementation assumes that `e.toString()` will actually create new values and not return an existing scope-allocated value. If that occurs we fall back to trying to extract `e`'s type.

```
static ThrowBoundaryError wrapTBE(Throwable e,
                                  MemoryArea area, Opaque outer){
  Opaque cur = LibraryImports.setCurrentArea(outer);
  String msg = null;
  try {
    try {
      msg = e.toString();
      if (getMemoryArea(msg) == area)
          msg = e.getClass().getName();
    } catch (Throwable t) { ... secondary error processing }
    return new ThrowBoundaryError(msg);
  } finally { LibraryImports.setCurrentArea(cur); }
}
```

Fig. 16: Boundary crossing. Exceptions that cross a scope boundary are copied to prevent references into an inner scope.

3.7.3 *Scope Life cycle Management.* The life cycle of a memory area requires careful management. Each scope maintains a reference count of the number of active threads in scope. A dedicated scope finalizer thread is in charge of executing the `finalize()` methods of objects before their reclamation. The critical part of the life cycle is the handling of enter and exit events. The work is split in three parts. Method `upRefForEnter()` is invoked before a thread enters a scope. `downRefAfterEnter()` is invoked after the thread exits. This method is in charge of the first part of the exit protocol; the second part of the exit protocol, `completeDownRef`, can be done at the same time or deferred to a separate finalizer thread.

Fig. 17 gives the implementation of scope entry. To ensure mutual exclusion and allow for notification, each scope has a dedicated lock, `joiner`. Once the entering thread acquires that lock, it checks if the scope is currently in use. If no other thread is active within it, the scope is reclaimed with `resetArea()` and nested below the top of the thread's scope stack. If there are threads already active in this scope, it is necessary to check that the thread's current scope is the same as the parent of the memory area we are about to enter. If they differ, the single parent rules has been violated and an exception is thrown. The method ends with an increment to the reference count to indicate that a new thread is live in the scope.

```
void upRefForEnter(RealtimeThread t) throws ScopedCycleException {
  synchronized (joiner) {
    if (refCount == 0) {
      if (!LibraryImports.hasChildArea(this.area)) {
        if (LibraryImports.areaOf(t) == this.area)
          ... // finalizer tries to reclaim this thread's scope
        LibraryImports.resetArea(this.area);
        setParent(t.getScopeStack().getCurrentArea());
      }
    } else {
      MemoryArea current = t.getScopeStack().getCurrentArea();
      if ((current instanceof ScopedMemory && current != parent) ||
        (!(current instanceof ScopedMemory) &&
           parent != primordialScope))
        throw new ScopedCycleException();
    }
    refCount++;
    joiner.ready = false;
  }
}
```

Fig. 17: Increment Scope Reference Count upon Entry. If the scope is not in use, this operation will reclaim memory and set the scope's parent to be the current scope. If the scope is in use, the scope's parent must be that same as the current scope. This method is defined in the UD, all ED accesses are mediated by calls to `LibraryImports`.

Decrementing the reference count after a thread returns from a scope is performed by `downRefAfterEnter()` as shown in Fig. 18. This may take the count to zero, resulting in the finalization of objects in the scope. The actual clearing of the scope occurs on the next entry – if any. We do not need to check if the parent needs to be finalized as that cannot be the case. The parent is either another scope that we entered directly (in which case we do not finalize until we exit it directly), or it is an upper scope that we revisited through `executeInArea`. If it is an upper scope reentered through `executeInArea()` then we are guaranteed that either it has a child other than this scope (part of the scope stack prior to the `executeInArea` call), or it has a non-zero reference count (because this thread already entered it explicitly). If the scope has other children, but these are in the process of being finalized, we hand the current scope to the finalizer thread for delayed processing. Similarly, if the current thread is a `NoHeapRealtimeThread` we hand the

scope to the finalizer thread. In other cases the finalization is performed right away by the `completeDownRef()` method as shown in Fig. 19.

```
void downRefAfterEnter() {
  boolean needExit = true;
  LibraryImports.monitorEnter(joiner);
  try {
    if (refCount > 1){ refCount--; return; }
    // refCount == 1
    if(LibraryImports.hasChildArea(this.area)) {
      if (ScopeFinalizerThread.instance.isProcessingChildOf(this)) {
        ScopeFinalizerThread.instance.add(this);
        needExit = false;
        return;
      }
      refCount--;
    } else {
      if (Thread.currentThread() instanceof NoHeapRealtimeThread) {
        ScopeFinalizerThread.instance.add(this);
        needExit = false;
        return;
      }
     completeDownRef();
    }
  } finally {if (needExit) LibraryImports.monitorExit(joiner);}
}
```

Fig. 18: Decrement Reference Count Upon Exit. This UD method will decrement the reference count and perform finalization either directly, or by handing the scope to the finalizer thread. The lock on the `joiner` object is obtained by an ED call and can be handed to the finalizer thread if necessary.

```
void completeDownRef() {
  if (LibraryImports.hasChildArea(this.area)){refCount--; return;}
  boolean more = false;
  do { more = LibraryImports.runFinalizers(this.area); }
  while (moreToDo && refCount == 1);
  if (refCount-- > 1) return;
  if (!LibraryImports.hasChildArea(this.area)
      && parent != primordialScope)
    parent.checkLastChild(true);
  resetParent();
  joiner.ready = true;
  doSetPortal(null);
}
```

Fig. 19: Finalize Scope Upon Exit. This UD method will call the ED `runFinalizers()` method to finalize all live objects (iterations are needed because, in Java, objects can create finalizable objects while being finalized!). Once finalization is complete the parent scope is reset and the portal is cleared.

```
class ScopeFinalizerThread extends RealtimeThread.VMThread {
  ScopeFinalizerThread() {
     super(HeapMemory.instance());
     rt_priority = RealtimeJavaDispatcher.MIN_RT_PRIORITY;
     setDaemon(true);
  }

  synchronized ScopedMemory take() throws PragmaNoBarriers {
      while (head == null)
        try { wait(); } catch(InterruptedException ex){ error();}
     return pop();
  }

  public void run() throws PragmaNoBarriers {
     while (true) {
       synchronized(this) { current = take(); }
       Opaque currentMA = LibraryImports.setCurrentArea(current.area);
       try {
          current.completeDownRef();
          LibraryImports.setCurrentArea(currentMA);
       } catch(Throwable t) { ... // error
       } finally {
         LibraryImports.monitorExit(current.joiner);
         restoreOriginalScopeStack();
       }
    }
  }
}
```

Fig. 20: Scope Finalizer Thread. This UD class performs scope finalization on behalf of `NoHeapRealtimeThreads` and when the scope being excited has children being finalized. The `take()` method blocks on a queue of scopes. `run()` changes the area to the scoped memory area to be finalized, and performs the second part of the scope exit protocol.

The scope finalizer thread is responsible for executing the finalizers of objects allocated in scoped memory, on behalf of `NoHeapRealtimeThread` instances that might encounter heap references if they did the finalization themselves. In addition, due to the need to process scope "exits" in a consistent order – child then parent – it will also process a scope whose child is being processed. Unlike other system threads this one runs at low-priority by default, only gaining in priority when some other thread needs to (re)enter a scope that is being finalized, or is joining a scope. This priority gain occurs through the normal priority inheritance mechanism (with a `notifyAll` thrown in for the joiners). The key parts of the implementation are shown in Fig. 20.

The scope finalizer thread maintains a queue of scoped memory areas waiting to be finalized and blocks on this queue while it is empty. When a thread adds a scope to the queue it first assigns the scope lock for that scope to the finalizer thread, then signals that a scope is in the queue. The scope finalizer thread acts like a normal heap-using thread and is subject to GC delays as normal. It follows that a `NoHeapRealtimeThread` that wants to reenter or join a scope that had finalizers, may also be delayed by the GC. This is unfortunate but it is an artifact of the RTSJ semantics and not of the Ovm implementation.

When the finalizers execute, the current allocation context must be the scope being finalized and the scope stack must be valid with respect to the single parent rule. The simplest way to achieve this would be to copy the scope stack of the thread that hands off the scoped memory area for finalization – however that would be rather expensive in both time and memory use, particulary as the same thread might be handing off multiple scopes in succession as its call stack unwinds. Instead we lazily create a temporary scope stack by considering the parent of the current area being finalized.

3.8 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 [Borger and Rajkumar 1989] is that the RTSJ allows threads to change their base priority dynamically. This feature impacts the implementation of PIP monitors as they must accurately reflect any changes in the priorities of blocked threads.

3.8.1 *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 bequeaths its priority to the monitor owner. If the bequeathed 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.

3.8.2 *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 case 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:* $t.ownedSet.size() = 0 \wedge$ $t.inhertanceQueue.size() = 0\ \wedge\ t.activePriority = t.basePriority$.

3.8.3 *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 its active priority invariant is met, and if not to change its active priority so that the invariant is met. An implementation can optimize things by checking for actual changes in active priority.

PropagatePriority: Has the following effect on a thread t:

```
if t blocked acquiring monitor m then
    m.lockSet.reposition(t); // ensure the lockSet is correctly ordered
    if m.lockSet.top ≠ 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

3.8.3.1 *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**
$m.owner.inheritanceQueue.remove(\textbf{old}\ \ m.lockSet.top)$
$m.owner.inheritanceQueue.insert(t)$
$m.owner.maintainPriority()$
$m.owner.propagatePriority()$

When t eventually acquires the monitor then the following happens:

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

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

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

3.8.3.3 *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.

3.9 I/O Scheduling and the AsyncIO Framework

In a user-level threaded system like Ovm, blocking I/O calls stall the whole virtual machine. To get around this problem Ovm has a POSIX I/O emulator which schedules I/O operations inside the VM. Included in Ovm is the AsyncIO framework, which provides asynchronous I/O scheduling primitives that are used to emulate ordinary I/O calls. This section discusses the design and implementation of the AsyncIO component of Ovm.

There are four implementations of AsyncIO, as shown in Fig. 21. The most trivial is the *stalling* implementation, in which asynchronous calls are blocking. This implementation is only provided as a fall-back, or for cases where the resource being used is known not to block for long (such as a file descriptor that refers to a RAM disk). The *polling* implementation is non-blocking and attempts periodically to perform an I/O request by having

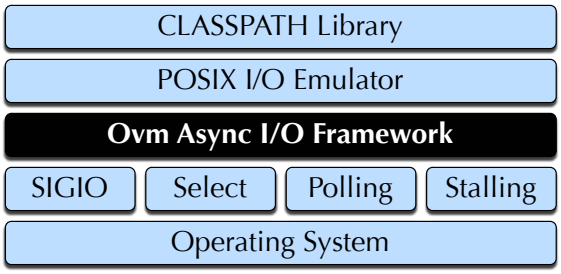


Fig. 21: Ovm I/O subsystem stack. An implementation of the POSIX I/O API based on AsyncIO is provided to prevent the entire virtual machine from blocking during file or network operations. There are four different implementations of AsyncIO with different characteristics.

a timer interrupt invoke the `ready()` methods of all pending operations. The last two implementations use the operating system's I/O notification mechanisms, namely SIGIO and `select`.

Two more implementations of AsyncIO are planned. One will use a pool of native threads, running outside of the virtual machine's direct control, to service I/O requests. Each request will be handed to one of those threads and would be executed with native blocking I/O calls. A second implementation will use the operating system's asynchronous I/O API. This approach is more involved as the mapping of the semantics of Ovm AsyncIO to the operating system's API may be subtle. We are investigating using Kernel Asynchronous I/O on Linux as well as Win32's asynchronous I/O primitives for an eventual Win32 port.

As mentioned above, scheduling of I/O operations is controlled by the AsyncIO framework. Clients, such as the POSIX emulator, are restricted to asynchronous operations. Upon reception of an I/O request, the AsyncIO framework arranges for the operation to take place at some point in the future with callbacks being used to notify the client when the operation has made progress. Fig. 22 illustrates the components of a prototypical asynchronous call. Such calls typically involve instances of five classes, `IODescriptor`, `AsyncMemoryCallback` (if memory buffers are used), `AsyncCallback` , `AsyncFinalizer` , and `AsyncHandle` , which are described next.

3.9.1 *IODescriptor.* An `IODescriptor` is analogous to a POSIX file descriptor. It contains both asynchronous and immediate non-blocking methods. For example, write comes in two flavors: asynchronous, `write()`, and immediate, `tryWriteNow()`. The latter performs the operation immediately or, if it can not complete right away, returns an error code. `write()` takes three arguments: a buffer, the number of bytes to output, and a callback. Notice that the buffer passed to `write()` is of type `AsyncMemoryCallback` , which allows the client of the call and the AsyncIO implementation to negotiate the best way of doing buffer management in a garbage collected and scope-aware environment.

3.9.2 *AsyncMemoryCallback.* Ovm supports multiple garbage collectors with different characteristics. Most relevant to this discussion is the ability to pin objects. Pinning an object prevents the collector from moving it until it is unpinned. If the implementation of an asynchronous operation, such as `write()`, needs to pass a pointer to code outside of the virtual machine's control while continuing to execute Java code, that pointer must be pinned to prevent the GC from moving the object while it is being used by the native

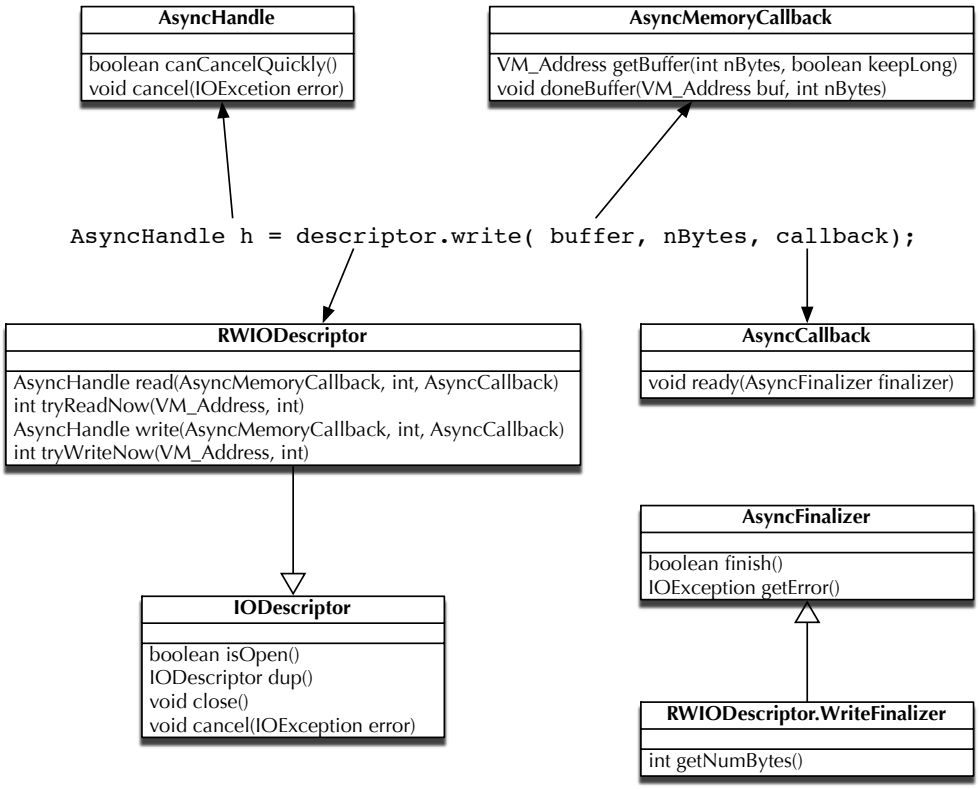


Fig. 22: Anatomy of a write operation in Ovm's AsyncIO framework.

operation. Some Ovm collectors support efficient pinning, while others do not. Additionally, some pointers (such as those allocated using `java.nio.ByteBuffer`, or any objects allocated in a scope) do not need to be pinned. Thus, we would like to be able to prevent pinning from taking place if it is unnecessary or inefficient. The `AsyncMemoryCallback` interface accomplishes this goal by having AsyncIO inform the client whether the buffer will need to be pinned for long (see the `keepLong` argument to `getBuffer()`). This allows the client to decide whether to pin (if it is efficient to do so), allocate a bounce buffer, or do nothing (if the object is already in non-movable space).

3.9.3 *AsyncCallback.* A callback scheme is used to notify the client of completion of an asynchronous operation, but with a twist: processing of the request is offloaded to the client, even if the operation is not guaranteed to complete. The client can, for example, be asked to optimistically perform a non-blocking read. Offloading has the advantage that operations are scheduled with the client's priority. When an asynchronous operation completes, the `ready()` method of the `AsyncCallback` is called with an instance of `AsyncFinalizer`.

3.9.4 *AsyncFinalizer.* When `ready()` is called on an `AsyncCallback`, it is incumbent on the client to call the `AsyncFinalizer`'s `finish()` method, possibly from a different thread. If `finish()` returns `true`, the operation completed (either by way of an error or success) and the client can determine the status of the operation using the `Async-`

```
abstract class AsyncOpQueue {
   AsyncOpQueue(Object lock);
   void addToSignalManager(OpNode node);
   void removeFromSignalManager(OpNode node, Object byWhat);
   OpNode performOp(OpNode node);
   int cancelAll(IOException error);

   static abstract class OpNode implements IOSignalManager.Callback,
          AsyncFinalizer, AsyncHandle {

      OpNode(AsyncCallback cback);
      boolean canCancelQuickly();
      void cancel(IOException error);
      boolean doOp();
      boolean finish();
      boolean signal(boolean certain);
      IOException getError();
   }
}
```

Fig. 23: Sketch of the AsyncOpQueue class.

`Finalizer`'s `getError()` method. Subclasses of `AsyncFinalizer` can provide more functionality, e.g. `getNumBytes()`. If `finish()` is `false`, the operation did not complete, and the `ready()` method will be called again at some point in the future.

3.9.5 *AsyncHandle.* Any asynchronous operation can be cancelled. In the worst case, `cancel()` will wait for the operation to run its course[2]. In our implementations of AsyncIO, cancellation is immediate: all work on the operation stops. Every operation has an `AsyncHandle` object. The object has two methods: `canCancelQuickly()`, which is `true` if cancellation always completes in bounded time, and `cancel()`, which cancels the operation by causing it to complete with an `IOException`.

3.9.6 *I/O Scheduling with AsyncOpQueue and IOSignalManager.* I/O scheduling for the three non-stalling AsyncIO implementations relies on a common set of abstractions. In fact, most of the work for managing an in-flight request is done by a single class, `AsyncOpQueue`. Deciding when to proceed with a request is the realm of implementations of the `IOSignalManager` interface. The signal manager sends "signals" to the `AsyncOpQueue` whenever it believes progress can be made. We refer to the polling, select, and SIGIO implementations of AsyncIO as *signal-based*, and distinguish between them only through the choice of `IOSignalManager`.

The AsyncOpQueue provides a queueing mechanism for operations of a given type, on a given `IODescriptor`. This queue is used internally by the signal-based AsyncIO implementations. A sketch of the `AsyncOpQueue` class is shown in Fig. 23.

A typical use of the queue is as follows:

(1) Implement a subclass of `AsyncOpQueue.OpNode`, implementing the `doOp()` method. The `AsyncOpQueue` will pass the `OpNode` as the finalizer. As such, the `OpNode` should

[2]In all cases, `cancel()` itself is asynchronous: even if the process of canceling an operation takes an unbounded amount of time, this work is done in the background, and the method itself returns immediately.

```
public interface IOSignalManager {
   void addCallbackForRead(int fd, Callback cback);
   void addCallbackForWrite(int fd, Callback cback);
   void addCallbackForExcept(int fd, Callback cback);
   void removeCallbackFromFD(int fd, Callback cback, Object byWhat);
   void removeCallback(Callback cback, Object byWhat);
   void removeFD(int fd, Object byWhat);
   static interface Callback {
      static final Object BY_SIGNAL = new Object();
      boolean signal(boolean certain);
      void removed(Object byWhat);
   }
}
```

Fig. 24: Sketch of the IOSignalManager interface.

implement whatever finalizer interfaces (such as `RWIODescriptor.WriteFinalizer`) the client expects.

(2) Subclass `AsyncOpQueue` to provide an implementation of the `addToSignalManager()` and `removeFromSignalManager()` methods. Typically, implementations will simply forward calls to an appropriate `IOSignalManager`.

(3) Implement the asynchronous operation by calling the `performOp()` method of the queue, and passing it a freshly allocated instance of the custom OpNode class. The `OpNode` also serves as the `AsyncHandle`, and so can be returned as such. It is necessary to hold the queue's lock while calling `performOp()`.

It is possible to queue different operations on the same queue. For example, a write queue may contain `OpNode` s for `write()`, `send()`, `sendto()`, `sendmsg()`, and so on. This works because POSIX will use a common notification mechanism for these operations, and because it makes sense to have them queued rather than attempting them simultaneously. For example, the SignalRWDescriptor class allocates two queues: a `readQueue` and a `writeQueue`.

While the AsyncOpQueue class keeps track of in-flight operations, the `IOSignal-Manager` has the task of abstracting the system's I/O notification mechanisms. The `IOSignal-Manager` interface is shown in Fig. 24. The semantics of the signal manager is:

—`IOSignalManager` s are allowed to be optimistic: it is fine for the `signal()` method of the callback to be called even when the state of the file descriptor has not changed.

—Level-trigger semantics are adhered to. That is, if one of the `addCallback` methods is called with a file descriptor that is already ready for the given type of operation, then `signal()` should be called either immediately or only after some bounded pause. This is in contrast to SIGIO's edge trigger semantics, where the application is only notified when the state changes. (Our SIGIO-based implementation of the `IOSignalManager` does extra work to emulate level-trigger semantics.)

—If `signal()` returns `false`, the callback is removed from the signal manager.

—The following operations are guaranteed constant time: all of the `addCallback` methods, except when `signal()` is called, in which case the worst case depends on the implementation of `signal()`; removing the callback is constant time when done as a result of `signal()` returning `false`.

—When an interrupt comes in from the operating system, it is possible that the `IOSignal-Manager` will have to poll each file descriptor registered with it. As such, the worst-case interrupt handling latency of the system depends on the number of file descriptors.

3.9.7 *Clients of AsyncIO* . Currently, we use AsyncIO to emulate POSIX I/O, since this is what the GNU CLASSPATH library expects. We provide a PosixIO object that implements I/O operations with POSIX semantics. Methods in PosixIO include `open()`,

```
public int write(
        int fd, Oop buf, int byteOffset, int byteCount, boolean block) {
    if (byteCount == 0) return 0;
    if (!verifyPointer(buf, byteOffset, byteCount)) {
        setErrno(NativeConstants.EFAULT);
        return -1;
    }
    IODescriptor io = getIOD(fd);
    if (io == null) {
        setErrno(NativeConstants.EBADF);
        return -1;
    }
    if (!(io instanceof RWIODescriptor)) {
        setErrno(NativeConstants.EINVAL);
        return -1;
    }
    try {                                                    (a) Non-blocking
        int result = ((RWIODescriptor)io).tryWriteNow(           fast path
                getPointer(buf, byteOffset, byteCount),
                byteCount);
        if (result >= 0) return result;
    } catch (IOException e) {
        setErrno(e);
        return -1;
    }
    if (!block) {
        setErrno(NativeConstants.EWOULDBLOCK);
        return -1;
    }
    Object r1 = MemoryPolicy.the().enterScratchPadArea();
    try {                                                    (b) Slow path that uses
        BlockingCallback bc = new BlockingCallback(bm,tm);       asynchronous I/O
        ((RWIODescriptor)io).write(
                new ForWriteMemoryCallback(buf,byteOffset,byteCount),
                byteCount, bc);
        bc.waitOnDone();
        IOException error = bc.getFinalizer().getError();
        if (error != null) {
            setErrno(error);
            return -1;
        }
        return ((RWIODescriptor.WriteFinalizer)bc.getFinalizer()).getNumBytes();
    } finally {
        MemoryPolicy.the().leave(r1);
    }
}
```

Fig. 25: Implementation of the POSIX I/O `write()` method.

`socket()`, `read()`, and so on. To illustrate the use of the AsyncIO framework, consider the implementation of `write()` shown in Fig. 25. Its signature is similar to the standard POSIX version with two differences: the buffer is a Java array which is passed as an `Oop` – a pointer to an any object. And second, there is an extra boolean to indicate whether the operation should block.

The implementation is close to its POSIX counterpart. It starts with some validity checks. If the tests succeed, the method attempts to perform the write with a non-blocking fast path, shown in Fig. 25(a). The fast path is always followed, regardless of whether other threads are blocking on writes to the same resource. While this violates fair queueing, it is consistent with the semantics of Java, since `java.nio` does not require fair queueing and `java.io` prevents concurrent access to file descriptors by using synchronization. Note that the fast path is executed with garbage collector disabled.

The slow path, in Fig. 25(b), is more expensive as it requires changing allocation context and creating some objects. The memory used for those objects should not come from the scoped memory area associated with the caller as this could lead to a memory leak if the current allocation context is not garbage collected. So, allocation is performed in a so-called scratch pad. Allocation contexts are changed by calls to method of `MemoryPolicy` (see Sec. 3.6 for details).

A `BlockingCallback` is allocated next. It is an `AsyncCallback` that blocks the client thread during the processing of the request. We then call the `write()` method of the `IODescriptor`, passing the blocking callback, and a `ForWriteMemoryCallback` to wrap the buffer. Once the asynchronous `write()` call returns, we call the `Blocking-Callback`'s `waitOnDone()` method, which blocks the thread until the asynchronous operation is complete.

3.9.8 *Complexity of AsyncIO Operations.* The complexity of AsyncIO operations depends on the AsyncIO implementation used. In this section we consider the polling implementation, since the bounds on the SIGIO and select implementations are no better (both operating system facilities may return optimistically, which in the worst case deteriorates to polling) and in fact may be much worse (we do not know the bounds on the system calls used for SIGIO and select; for all we know these calls may be very slow). The polling implementation makes only very limited use of operating system facilities, which leads to a simple analysis.

Consider the sequence of events that take place in the worst-case POSIX I/O read or write operation implemented using AsyncIO with polling. First, we experience failure in the non-blocking fast path seen in Fig. 25(a). The worst-case performance of the Java code has constant time, but the bound on the non-blocking operation depends on the operating system. Next, we proceed with allocating objects in the slow path. The allocated objects have fixed size and are allocated in linear-time scoped memory. Inserting the `OpNode` into the `AsyncOpQueue` and the `IOSignalManager` takes constant time. The `BlockingCallback` removes the thread from the ready queue, which takes time proportional to the number of threads.

Thereafter, on every timer poll, the client thread will be scheduled to attempt the non-blocking version of the requested operation. Hence, if there are N threads blocked on I/O, every timer poll will require $O(N)$ threads to be placed on the thread manager's ready queue. This takes $O(N)$ time, since making a thread ready takes constant time. Once allowed to run, each thread will attempt the non-blocking operation. This is a system call,

the bounds of which depend on the operating system. Once the operation returns, either the thread is put back to sleep (which requires time proportional to the number of threads) and scheduled again on the next timer call, or else the operation is complete and the thread is allowed to continue executing.

3.10 Complexity of Basic Java and RTSJ operations

We now informally discuss the complexity of some of the basic Java and RTSJ operations. Method invocation is performed in constant time. In most cases the j2c compiler will devirtualize method, and thus turn them into standard function calls. When not possible the invocation is done through a function pointer retrieved from a statically known offset of a dispatch table. Subtype test are implemented using the technique described in [Palacz and Vitek 2003] which guarantees constant time test for classes and interfaces. The complexity of exception throwing in plain Java is computed in two parts: first construction of the string which contains the error message. This done in time proportional to the depth of the call stack. Then finding the handler is proportional to the depth of the call stack and the number of handlers in each frame. The RTSJ adds to this the cost of copying the exception message at each scope boundary. Memory barriers (read/write) are performed in constant time, and so is the `areaOf()` operation. Allocation in a scoped memory area is proportional to the size of the object being allocated due to the need of zeroing it. The complexity of entering and leaving scopes depends on a number of issues: whether the scope was used by multiple threads, whether it contains finalizable object, whether it has multiple child scope. To compute the worst case execution time of an enter – we need the WCET of the exit (as the entering thread may have to block for an exiting thread). The WCET of the exit depends on the WCET of runing the finalization methods (which are user code) and, possibly, that of the garbage collector.

4. OVM PERFORMANCE EVALUATION

We have evaluated Ovm on a number of benchmarks and report some of these results here. Unless otherwise specified, 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.

There is a small but growing body of work on measuring performance characteristics of Real-time Java [Higuera-Toledano and Issarny 2002; Corsaro and Schmidt 2002a; 2002b; Niessner and Benowitz 2003; Bollella et al. 2003]. Unfortunately comparing different implementations is difficult due to the proprietary nature of many systems. We only have copies of jRate and jTime at our disposal as of this writing.

4.1 Throughput Benchmarks.

We evaluate the raw performance of Ovm on the SpecJVM98 benchmark suite and compare it with the Timesys jTime RTSJVM 1.0 (compiled), Hotspot 1.5 and GCJ 4.0.2. jTime, Ovm and GCJ are ahead-of-time compiled, Hotspot is using just-in-time compilation. The goal of this experiment is to provide a performance baseline. We evaluate two Ovm configurations: the plain Java configuration and the RTSJ configuration which includes scoped memory access checks. jTime, likewise, has read/write barriers turned on.

The results, given in Fig. 26, show that performances of Ovm and GCJ are close. Typically, Ovm is slightly faster with the exception of mpegaudio where the slowdown is due in part to our treatment of floating point numbers. This will be addressed in forthcoming

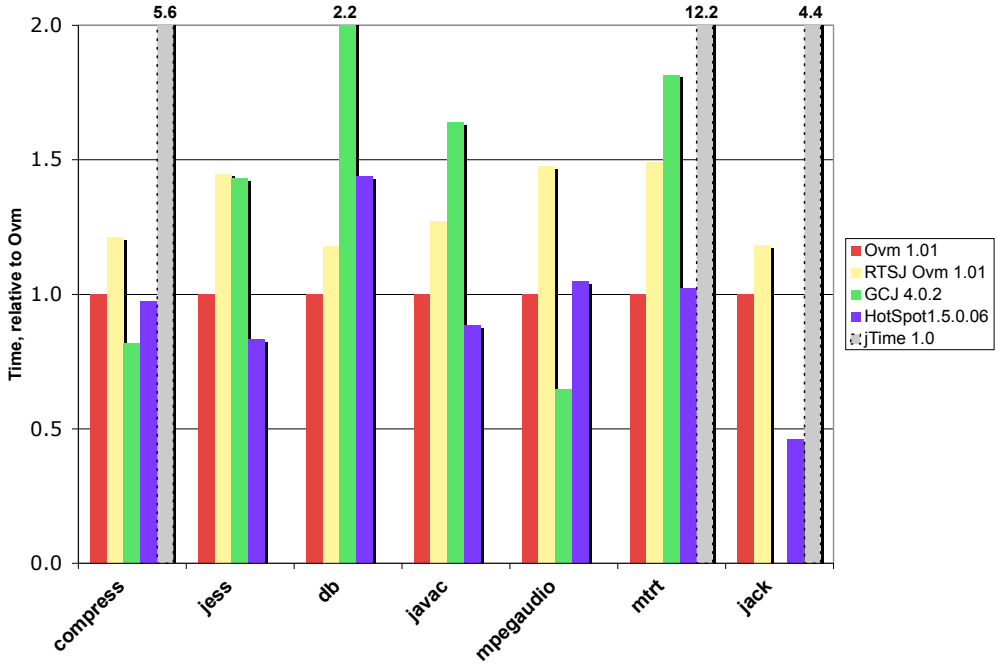


Fig. 26: SpecJVM98. (normalized wrt. Ovm) "Ovm" is the Java configuration without scope checks. RTSJ-Ovm outperforms the jTime RTSJVM. Ovm outperforms, on average, GCJ and is competitive with Hotspot.

releases. GCJ did not execute jack successfully, and jTime could not run jess, db, javac and mpegaudio. The figure also illustrates the costs of RTSJ barriers (up to 50%). SpecJVM is by no means representative of a real-time application, but it gives a worst case estimate of the cost of memory access checks.

For completeness, we tried to compare the performance of Ovm against two other RTSJ implementations: JamaicaVM [AICAS 2005], a commercial implementation, and jRate [Corsaro and Schmidt 2002a], a project that adds many RTSJ features to GCJ. In both cases we encountered problems building a statically linked and optimized binary on our execution platform. In the case of JamaicaVM, an unexpected linking error prevented us from obtaining a working fully optimized static binary. The non-optimized version worked correctly, but with considerably lower performance (compress required about 11 minutes to run against about 16 seconds for Ovm), therefore the comparison was deemed to be unfair. Our attempts to run SPECjvm98 on jRate were also unsuccessful, as we could not find an obvious way to force the jRate build process (which involves patching a GCJ distribution and regenerating binaries and libraries) to include the required `awt` support, seemingly necessary to compile statically the SPECjvm98 suite.

4.2 Startup Latency.

We measure the startup time of Ovm on a 300MHz PPC. The time required to load the virtual machine from disk and perform any initialization and up to and excluding the first instruction in the user's `main()` method is 290 milliseconds with very little jitter. The image used here is that of PRiSMj (22 MB of data, and 11 MB of code).

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. 27 shows the latency of a number of basic RTSJ operations and compares Ovm 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.

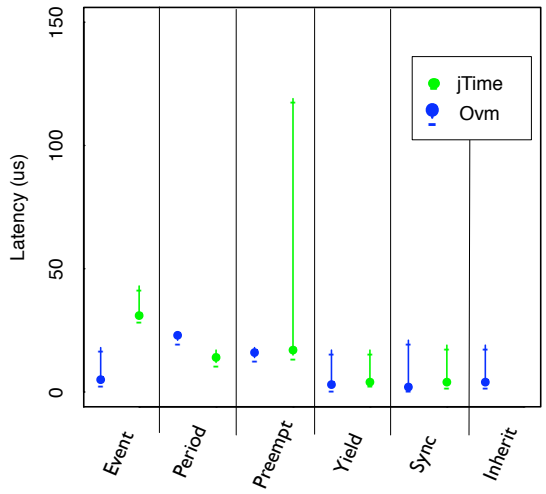


Fig. 27: Boeing RTSJ Latency benchmarks. Comparing Ovm with jTime. (in microseconds on a 1.6GHz AMD.)

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 the 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 while jTime must call into the OS.

4.4 Latency and Throughput Impact of Poll-checks.

Compiler inserted poll-checks are essential to Ovm's scheduling infrastructure as they are the only points where a thread can be preempted. Polling has the advantage of simplifying the implementation of synchronization primitives. The downsides are (*i*) performance overheads, both from the time spent executing the poll-check and from compiler optimizations impeded by their presence, and (*ii*) potential increases in preemption latency. Decreasing the frequency of poll-checks means that there will be longer segments of code with interrupts deferred.

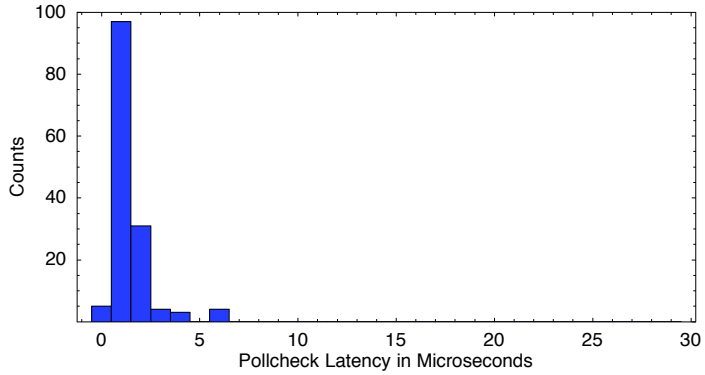


Fig. 28: Distribution of poll-check latencies for the PRiSMj 100X scenario. Poll-check latency is the time between an interrupt and a poll-check that services that interrupt. Worst case observed latency is 6 microseconds.

Fig. 28 gives the distribution of interrupt-to-check latencies for the PRiSMj 100X benchmark. We record the time difference between each event occurring outside of an atomic region –where polling is turned off– and the time of the next poll-check. The maximum latency is consistently six microseconds, other benchmarks exhibit similar behavior. The current implementation of polling does not have an adverse effect on preemption latency.

To estimate the impact of poll-checks on throughput, we run Ovm on SpecJVM98 benchmark suite with and without poll-checks. See Fig. 29 for percent overheads measured for poll-checks in the Spec benchmarks. The overheads were computed based on the median of 20 runs. All benchmarks exhibit under 3% overhead.

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 by the optimizing compiler to improve code quality. We give the example of two applications, PRiSMj and RT-Zen (both are described later). Fig. 30 gives

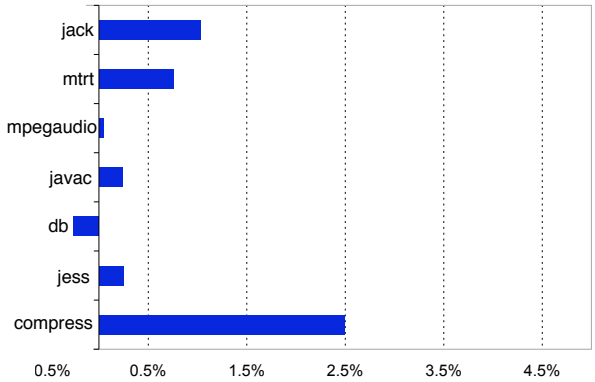


Fig. 29: Percent overhead of poll-checks in SpecJVM98 benchmarks. In this graph, 0% overhead indicates that enabling polling did not slow down the benchmark. Overhead is under 3%.

the size of all the 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	–	–

Fig. 30: Footprint. Lines of code computed overall all Java sources files (w. comments). Data/Code measure the executable Ovm image for two complete application (PRiSMj and Zen) on a PPC.

The compiler performs a Reaching Types Analysis to discover the call graph of the application and to prune, in the process, dead methods and dead classes. The results are shown in Fig. 31. 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 used classes is the number of classes that are determined to be live, i.e. may be accessed at runtime. The number of defined methods is the sum of all methods of live classes. The number of used methods is the subset of those methods which may be invoked. Methods that are not used need not be compiled.

	classes loaded / used	methods defined / 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. 31: Impact of compiler optimizations.

Finally, Fig. 31 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. `instance of`) 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 University of California Irvine and written to the RTSJ API's. 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, the application runs with a low priority and a high priority real-time thread. Fig. 32 reports on the time taken to process a request.

The jitter for the high priority thread is approximately 7 milliseconds, this is 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 `NoHeapRealtimeThread` objects affect 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 gen-

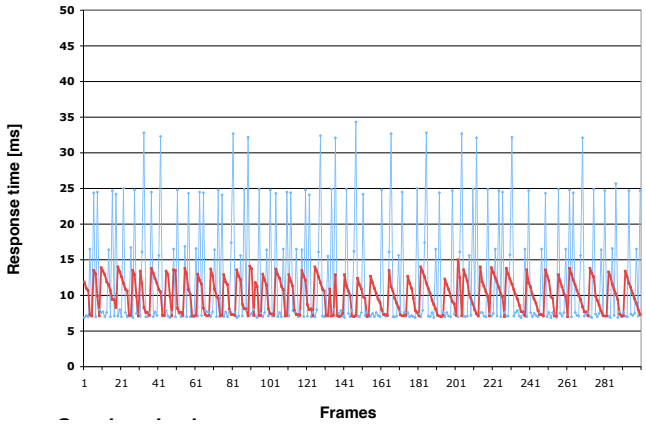


Fig. 32: RT-Zen Results. Comparing the response time for an application running on top of a RTSJ CORBA ORB. Two thread groups (low and high) handle 300 requests each. The y-axis indicates the time to process a request. (milliseconds)

eral 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 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

In our experience, Scoped Memory is one of the harder features of the RTSJ to implement and use correctly. In Ovm, support for scopes permeates much of the virtual machine implementation. While many of the semantic issues related to scopes have been resolved, there are still some disturbing corners cases. For instance, since both `RealtimeThread` and `NoHeapRealtimeThread` threads can share a scope - it is possible to set up a situation where a `NoHeapRealtimeThread` blocks for the GC. This occurs if a `NoHeapRealtimeThread` blocks on a monitor held by a `RealtimeThread` which tries to allocate an object in the heap.

Another issue is finalization of objects in a scope. The first point is that the last thread out of the scope will pay the price of finalizing objects– even if it did not allocate them. But, even worse, a scope may contain objects which, as part of their finalization, manipulate heap-allocated data structures. If the last thread out happens to be a `NoHeapRealtimeThread` , finalization will fail – this is probably not a desirable outcome as these objects were most likely allocated by a `RealtimeThread` thread. A partial solution is to turn off the "no-heap"-ness of the exiting thread to allow it to touch heap-allocated data. But, doing this means that the `NoHeapRealtimeThread` is subject to preemption by the GC.

As for usage of scopes, it is notably difficult. Avoiding memory access errors impose burdens on programmers and increases the potential for bugs as they require reasoning about localities [Pizlo et al. 2004]. With the RTSJ, programmers must be aware where an object has been allocated, a piece of information that cannot be obtained straightforwardly by inspection of the program text. The result is that scopes are, in our experience, the main source of program errors in RTSJ programs.

Scoped Memory has given rise to a large body of related work. While we do not claim exhaustiveness, we will mention some notable papers. [Bollella and Reinholtz 2002] presents a rationale and defense of Scoped Memory. Several authors have attempted to catalog best practice programming idioms and design patterns [Pizlo et al. 2004; Benowitz and Niessner 2003b] for using scoped memory in a disciplined fashion. But, this is not enough to ensure correctness. Wellings et al. designed the Ravenscar Java profile, a set of restrictions intended to make the RTSJ suitable for high-integrity applications. One of its recommendations is to reduce the expressive power of scoped memory areas: areas must be preallocated and can not be nested [Kwon et al. 2005]. While this goes some way towards simplifying the programming model, the basic issues of runtime assignment errors remain. In [Zhao et al. 2003] a lightweight static discipline based on ownership types was proposed to give static correctness guarantees for RTSJ programs. Like all static disciplines it entails restrictions in expressiveness – whether the proposed programming model is sufficiently expressive remains an open question. Kwon and Wellings proposed another approach to ease the task of programming with scoped memory, by using annotation on methods that indicate their memory-related characteristics [Kwon and Wellings 2004]. Another proposal involved using weak references [Borg and Wellings 2003]. The first investigation of the performance of scoped memory appeared in 2001 [Higuera-Toledano et al. 2001] in the context of an interpreted virtual machine. Implementations of scope checks were described in [Fox and Welc 2003; Corsaro and Cytron 2003; Palacz and Vitek 2003; Beebee, Jr. and Rinard 2001].

A partial alternative to Scoped Memory is to use a real-time garbage collector (RTGC) such as [Bacon et al. 2003; Siebert 1999; Schmidt and Nilsen 1994]. But, even the propo-

nents of RTGC admit that there will likely always be a subset of hard real-time codes for which the overheads of garbage collection are not acceptable.

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 has not 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 moment in which the GC sees a heap reference and the moment in which it updates the same reference, 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 (the scope can not hold any new heap references because only a `NoHeapRealtimeThread` could be accessing it).

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 [FSF 2005] (which is used by Ovm), are not written to support real-time. At the simplest level this often means that they do not 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 do not 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. The implementation of a

scope-aware `Vector.ensureCapacity()` method is shown in Fig. 33.

```
void ensureCapacity(int cap){
    ...
   Object[] arr=(Object[]) thisArea.newArray(Object.class, cap);
   System.arraycopy(elementData,0,arr,0,elementCount);
   elementData=arr;
}
```

Fig. 33: Scope-aware libraries: the `ensureCapacity()` method of the Vector class. Growing a generic data structure must be performed in the original allocation context of the object if we want to avoid memory access violations.

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. 34. 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. 35 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. 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 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. 36.

Similar experiments for the C++ implementation of the PCES benchmarks were reported in [Sharp et al. 2003]. Our results suggest that the Java version is faster. The readers should note that we were not able to obtain the C++ software to re-run the experiments on identical hardware configurations. Ovm ran on a PPC board while the C++ version ran on

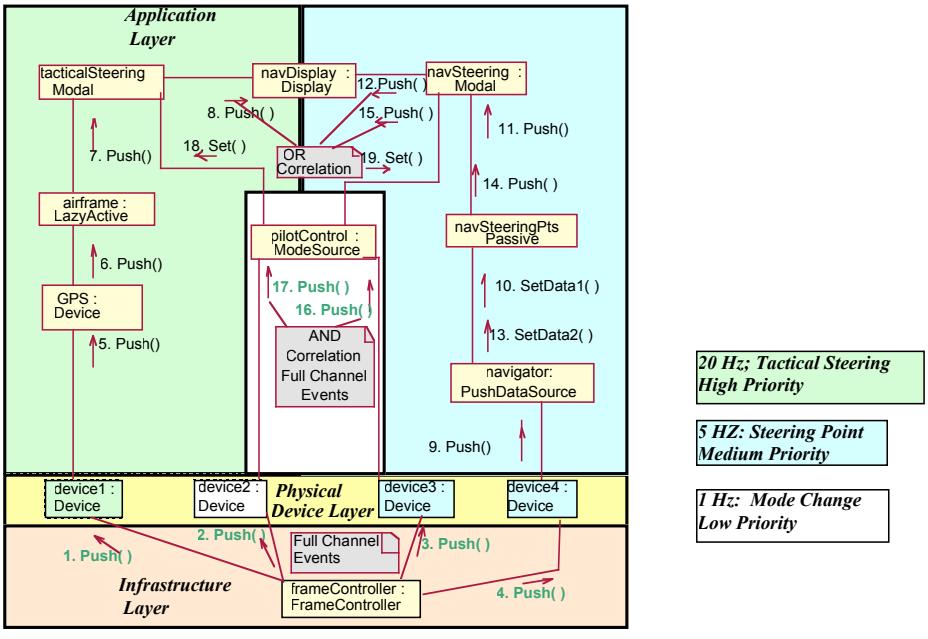


Fig. 34: The overview of the Boeing PRiSMj 1X scenario.

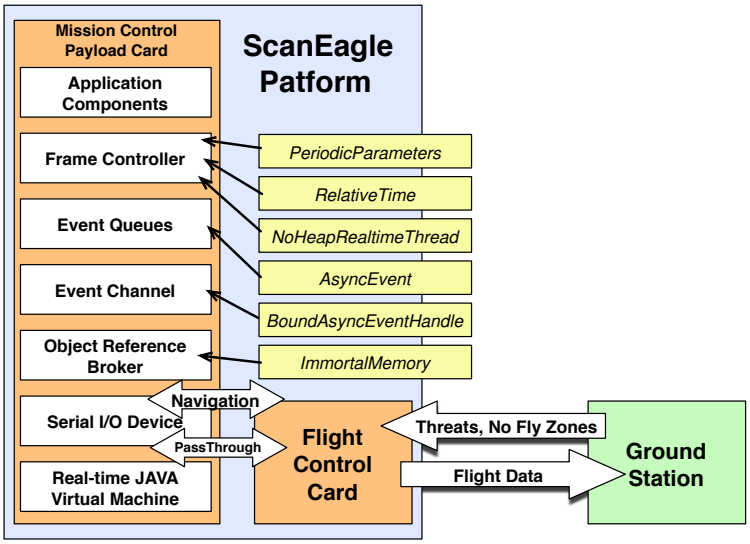
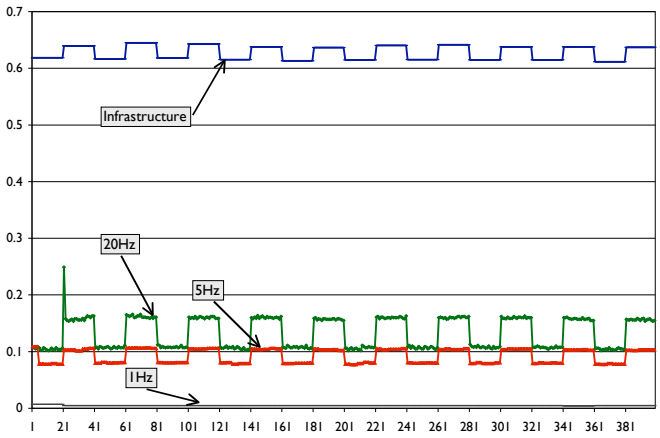


Fig. 35: ScanEagle Flight Product Scenario RTSJ Architecture.



PRiSMj 100x workload.

Fig. 36: 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)

more powerful 1.2GHz Pentium 4. The two implementations are comparable. The Boeing engineers who worked on the original C++ software translated it to Java with little need for adaption.

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 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 at a periodic rate of 1Hz. The flight product scenario is illustrated in Fig. 35. 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 a 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 could be resolved. The most complex implementation issue involved implementing the semantics of the RTSJ memory management model and in particular scoped memory areas. We uncovered some ambiguities in the specification which have been addressed in

the 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, Ovm running RTSJ classes provided comparable performance, in fact ran faster than the C++ version of the PRiSMj application. 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. Scoped memory clearly proved to be the single most difficult feature of the RTSJ to master.

Acknowledgments. The authors thank Ken 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, Krzysztof 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

AICAS. 2005. The Jamaica Virtual Machine homepage, `http://www.aicas.com`.

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.

BACON, D. F., CHENG, P., AND RAJAN, V. 2003. The metronome: A simpler approach to garbage collection in real-time systems. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*. 466–478.

BEEBEE, JR., W. S. AND RINARD, M. 2001. An implementation of scoped memory for Real-Time Java. In *Embedded Software Implementation Tools for Fully Programmable Application Specific Systems (EMSOFT)*. 289–305.

BENOWITZ, E. AND NIESSNER, A. 2003a. Experiences in adopting Real-Time Java for flight-like software. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*. 490–496.

BENOWITZ, E. G. AND NIESSNER, A. 2003b. A patterns catalog for RTSJ software designs. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*. 497–507.

BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S. 2004. Oil and water? high performance garbage collection in Java with MMTk. In *26th International Conference on Software Engineering (ICSE'04)*. 137–146.

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.

BOLLELLA, G., LOH, K., MCKENDRY, G., AND WOZENILEK, T. 2003. Experiences and benchmarking with JTime. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*. 534–549.

BOLLELLA, G. AND REINHOLTZ, K. 2002. Scoped memory. In *Proceedings of the Fifth International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC02)*.

BORG, A. AND WELLINGS, A. J. 2003. Reference objects for RTSJ memory areas. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*. 397–410.

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.

BROSGOL, B., ROBBINS, S., AND HASSAN II, R. 2002. Asynchronous transfer of control in the Real-Time Specification for Java. In *International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*. 101–108.

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.

CHILD, J. 2003. Java proving itself worthy for defense apps. *COTS Journal*.

CHILD, J. 2004. Real-time flavor completes the military Java puzzle. *COTS Journal*.

CORSARO, A. AND CYTRON, R. 2003. Efficient memory-reference checks for real-time Java. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*.

CORSARO, A. AND SCHMIDT, D. 2002a. The design and performace of the jRate Real-Time Java implementation. In *The 4th International Symposium on Distributed Objects and Applications (DOA'02)*.

CORSARO, A. AND SCHMIDT, D. 2002b. Evaluating Real-Time Java features and performance for real-time embedded systems. In *The 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

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.

FLACK, C., HOSKING, T., AND VITEK, J. 2003. Idioms in Ovm. Tech. Rep. CSD-TR-03-017, Purdue University Department of Computer Sciences.

FOX, J. M. AND WELC, A. 2003. Implementation of Real-Time Java scope access checks for JikesRVM. Tech. report, Purdue. may.

FSF. 2005. Free Software Foundation Inc, GNU Classpath, `www.gnu.org/software/classpath`.

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.

HIGUERA-TOLEDANO, M. T., ISSARNY, V., BANÂTRE, M., CABILLIC, G., LESOT, J.-P., AND PARAIN, F. 2001. Region-based memory management for Real-time Java. In *4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*. 387–394.

HIGUERA-TOLEDANO, T. AND ISSARNY, V. 2002. Analyzing the performance of memory management in RTSJ. In *Proceedings of the Fifth International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)*.

KWON, J., ANDY J. WELLINGS, AND KING, S. 2005. Ravenscar-Java: a high-integrity profile for real-time Java. *Concurrency - Practice and Experience 17,* 5-6, 681–713.

KWON, J. AND WELLINGS, A. 2004. Memory management based on method invocation in RTSJ. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*. 333–345.

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.

NIESSNER, A. AND BENOWITZ, E. 2003. RTSJ memory areas and their affects on the performance of a flight-like attitude control system. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*. 508–519.

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 tests in real-time. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2003)*. Lecture Notes in Computer Science, vol. 2743. Springer, Darmstadt, Germany, 378–404.

PIZLO, F., FOX, J., HOLMES, D., AND VITEK, J. 2004. Real-time java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC'04)*. Vienna, Austria.

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.

SCHMIDT, W. J. AND NILSEN, K. D. 1994. Performance of a hardware-assisted real-time garbage collector. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems. San Jose, California, 76–85.

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.

WELLINGS, A. AND PUSCHNER, P. 2003. Evaluating the expressive power of the Real-Time Specification for Java. *Real-Time Systems 24,* 3, 319–359.

ZHAO, T., PALSBERG, J., AND VITEK, J. 2003. Lightweight confinement for featherweight java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM Press, 135–148.