

Issues in the Design and Implementation of Efficient Interfaces Between Hard and Soft Real-Time Java Components

Kelvin Nilsen, Andrew Klein

Aonix, NA, 877 S. Alvernon Way, Suite 100, Tucson, AZ 85711
{kelvin, klein}@aonix.com

Abstract. Almost all modern computer systems involve multiple layers of software, each manifesting different requirements for performance, security, and real-time predictability. Interfaces between layers are required to efficiently and reliably share information without compromising the requirements of the layers that individually participate in the interface. This paper discusses principles and empirical results in the comparison of three available techniques for interfacing between hard and soft real-time components: the Java Native Interface, the Real-Time Specification for Java, and Real-Time Core Extensions.

1 Background

For decades, principles of good software engineering have dictated strong encapsulation and partitioning of responsibility so that independently developed software components integrate cleanly into larger software systems. The implementations of modern computing environments typically include multiple layers of responsibility and trust. At the lowest layers, the operating system may itself be divided into a multi-tasking micro-kernel and interrupt handlers with the highest levels of trust, and various operating system services (e.g. caching file system, virtual memory paging, network protocol stacks) implemented as protected threads running on top of the micro-kernel. Typical application code, which runs on top of the operating system, is the least trusted. Modern high-level operating systems will protect their own integrity from violation by application code. They will also protect individual applications from being violated by each other.

Often, developers of embedded real-time systems must work at the very lowest layers of this software hierarchy. Certain hard-real-time mission-critical and safety-critical applications require that the application developer implement his own interrupt handlers and perform other operations that might compromise the integrity of the real-time operating system. Clearly, these developers must be trusted to do the right thing.

The challenges we face as providers of developer tools and real-time frameworks are to help these developers do the right thing by:

1. Making the job of developing hard-real-time components as easy and safe as possible. We do this by providing tools and reusable software components that reduce

the programmer's burden of responsibility and automatically verify that particular global invariants are always satisfied.

2. Providing mechanisms that enforce strong partitioning of responsibility and trust between layers of the software hierarchy. Whatever the mechanism, it must support independent realms of "trust" that are protected from one another. Any real-time development methodology that requires equal trust in all developers will never scale to the size and complexity required by today's embedded systems.
3. Assuring that the resulting technologies provide adequate efficiency in terms of memory footprint and execution speed. If a bullet-proof methodology runs so inefficiently that programmers frequently step outside its boundaries in order to get adequate performance, it is no longer bullet proof.

It needs to be emphasized that this paper focuses exclusively on issues related to the flow of information and control between the hard real-time and traditional Java environments. There are many other criteria by which available hard-real-time technologies could also be evaluated, including, for example, ease of development, portability, scalability, and maintainability of software.

1.1 Java Native Interface (JNI)

The Java Native Interface is a set of protocols that allow mixing of Java code with traditional native-compiled code written in, for example, C or C++ [1]. Using these protocols, threads started within the Java virtual machine environment can call methods implemented as native code. And operating system threads started outside the Java virtual machine environment can attach to the virtual machine and invoke methods implemented in Java.

Because of inherent differences between the Java and native programming models, the interface between these worlds is fairly expensive. For example, Java objects that are made visible to the native code must be associated with registered handles to ensure that the garbage collector does not reclaim their memory. And native code that is traversing Java data structures needs to coordinate with the garbage collector to assure data coherency between what is seen by native code, Java code, and the garbage collector itself.

In some cases, the JNI protocol makes the addresses of particular pinned Java objects visible to native code. The native code is then free to manipulate the referenced objects using traditional C-style pointer operations. When native programmers accidentally (or maliciously) reach beyond the ends of objects or overwrite certain object fields with inappropriate values, the Java virtual machine itself can be compromised. At some future time following one of these storage trampler events, the Java virtual machine may crash because its internal data structures have been corrupted.

1.2 The Real-Time Specification for Java (RTSJ)

The Real-Time Specification for Java [5] is a collection of APIs in combination with a tightening of the semantic requirements for standard Java APIs. The RTSJ requires, for example, that traditional Java threads honor strict priority scheduling as determined by the Java thread priorities and not by operating system heuristics that occasionally age the priorities of certain threads, boosting their priorities if they have not recently run and lowering their priorities if they are seen to be consuming more than their fair share of CPU resources. The RTSJ also requires the implementation of Java's synchronization constructs to provide a means of avoiding priority inversion.

The RTSJ introduces certain new APIs that are not in standard Java. Among the extensions, RTSJ introduces two new thread types: **RealtimeThread** and **NoHeapRealtimeThread**. Unlike traditional Java threads, these threads respond to asynchronous interruption and cancellation requests. These threads are also able to run at priorities higher than the traditional virtual machine's 10 thread priorities. And the **NoHeapRealtimeThread** is guaranteed to never be preempted by garbage collection.

Within the RTSJ specification, the scheduling behavior of **RealtimeThread** is not well defined. If a particular RTSJ implementation offers some form of real-time garbage collection, then the developer can expect predictable scheduling behavior. But if it doesn't, **RealtimeThread** threads will experience unpredictable interference from garbage collection activities.

NoHeapRealtimeThread threads achieve highly predictable real-time scheduling behavior by avoiding all access to heap-allocated objects. Whenever they need access to a dynamically allocated object, they must allocate this object either from an **ImmortalMemory** or **ScopedMemory** region. **ImmortalMemory** objects live permanently and can never be reclaimed. For applications that are expected to run reliably for years at a time, the only objects that can be allocated in **ImmortalMemory** are the ones allocated during initial startup. Objects allocated within a **ScopedMemory** region are all reclaimed simultaneously, at the moment the reference count for the **ScopedMemory** region itself is decremented to zero. **ScopedMemory** regions may nest, and objects within one **ScopedMemory** region may refer to objects in outer-nested **ScopedMemory** regions, but not the other way around. Run-time checks accompany every assignment to reference fields of objects to make sure that these constraints are not violated.

Synchronization between **NoHeapRealtimeThread** and traditional Java threads is problematic given that a traditional Java thread may be preempted by garbage collection while it holds a lock on a shared object. If a **NoHeapRealtimeThread** subsequently requests access to the same synchronization lock, it may be forced to wait for garbage collection to complete before the traditional Java thread can relinquish its lock to grant access to the **NoHeapRealtimeThread**. For this reason, the RTSJ programmer is told not to synchronize between **NoHeapRealtimeThread** and tradi-

tional Java threads. Instead, any information sharing between these two domains must be realized by copying the data into **ImmortalMemory** objects and passing the copies to the other domain by way of built-in wait-free queues.

The RTSJ designers have commented in discussing the rationale for various design tradeoffs that “real time is not real fast.” The primary criterion in designing the RTSJ was to enable predictable and deterministic execution. Some of the performance compromises that were implemented in order to achieve this include the requirement to impose run-time checks on reference assignments and to require copying of data between real-time and non-real-time domains whenever information sharing is necessary.

1.3 Real-Time Core Extensions by the J Consortium (Core)

The Real-Time Core Extensions specification is similar to RTSJ in that it adds real-time programming capabilities to the standard Java virtual machine environment [6]. Both approaches partition objects so that hard-real-time threads do not depend on garbage collection activities. Insofar as is relevant to interface issues, the Core specification differs from RTSJ in the following regards:

1. The Core specification focuses only on hard-real-time applications. The intent is to serve the needs of soft-real-time developers with complementary but distinct technologies. Contrast this with the RTSJ design which promotes the use of the same APIs to serve both hard-real-time and soft-real-time requirements.
2. The Core attaches to existing Java virtual machines without requiring any changes to the existing virtual machines, or can alternatively deploy as a stand-alone execution environment. Note that RTSJ requires significant changes to the synchronization, thread scheduling, and JIT code generation models of any virtual machine it extends. To date, RTSJ extensions have only been demonstrated on J2ME.
3. The Core allows sharing of objects between the hard-real-time and traditional Java domains. All such sharing uses a carefully constructed protocol patterned after Ada’s protected objects [7]. This protocol avoids the garbage collection synchronization difficulties faced by the designers of the RTSJ.
4. The Core supports neither immortal memory nor scoped memory. It offers allocation regions that are similar to immortal memory except that memory allocated in such regions can be reclaimed under programmer control. It also offers stack-allocated memory which is similar in concept to scoped memory except that the compiler enforces the restrictions required for safe operation. The design of stack allocation supports polymorphic inheritance and code reuse, guarantees absence of dangling pointers, and performs all enforcement at compile time without any requirement for run-time checks.
5. Though designers of the Core agree with RTSJ designers in observing that “real-time is not real fast”, they sought to address a broader audience for whom execution speed, memory footprint, and deterministic execution were all very important.

The Core is designed to offer speed, footprint, and latency comparable to C or C++ running with a modern real-time operating system.

2 Evaluation Criteria

The focus of this paper is on understanding issues related to the interface between hard-real-time and soft-real-time components. In evaluating the strengths and weaknesses of each proposed approach, we consider the issues discussed below. Since one of this paper's authors served as the editor of the J Consortium's Real-Time Core Specification, this list of evaluation criteria closely mirrors the objectives of those who drafted that specification. We are not intentionally overlooking alternative criteria which might favor alternative interface designs. Rather, we are not fully aware of the objectives that motivated the designs of the RTSJ and of the JNI. We invite open discussion on these topics so that the general-purpose infrastructure technologies that we are developing will have broadest possible appeal and so that developers who might need to select between alternative infrastructure solutions can do so based in part on distinctions in the objectives that each infrastructure approach attempts to satisfy.

Software Engineering Ideals. Software engineering principles are intended to reduce development costs, improve software reliability, and increase generality (in order to improve reuse). The following guidelines capture our sentiment with respect to these concerns. In each case, the benefits of each objective are assumed to be independent of all others. We make no attempt to quantify the importance of individual objectives, and recognize that different situations will place different weights on the relative importance of each objective.

1. An interface design that allows the developer to think about fewer details is preferred over one that requires the developer to address more details.
2. An interface design that allows the developer to write less code is preferred over one that requires the developer to write more code.
3. An interface design that encourages composability and object-oriented inheritance is preferred over one that does not.

Certification Issues. Often, layered software systems are partitioned so as to protect more secure components from less secure components. We have commented already that there is a tendency to "trust" the developers of the lowest layer software and to put protection mechanisms in place to make sure that the lower-layer software is not compromised by the less trusted higher-level software. In order to earn the trust required of the software comprising the lowest layers of the system hierarchy, developers of that lowest-layer software usually adopt development processes that are much more rigorous than processes typical of higher level software. These processes include enforcement of rigid code style guidelines, code inspections by peers, and extensive testing driven by code coverage analysis. In some cases, such as when this software controls the flight of a commercial aircraft, the software processes include certification by an external auditing agency. Whether code is externally or internally certified, the guiding

principles remain the same. Here we identify several additional criteria by which interface techniques are compared.

4. An interface that protects the low-level software from being compromised by high-level software is preferred over one that offers no such protection.
5. An interface that reduces interference by low-level software into issues pertaining to high-level software is preferred over one that does nothing to reduce such interference.

Performance. For many applications, performance is a critical development objective. This is one of the reasons that the C language is used by over 70% of embedded developers even though the language is over 30 years old and clearly does not scale well to the large embedded projects typical of today's market. Thus, we identify two more evaluation criteria:

6. An interface technique that imposes very little, if any, overhead at the point that information passes through the interface is preferred over an interface technique that imposes a high overhead.
7. An interface technique that imposes very little, if any, overhead on the execution of typical code running in any of the software layers connected by the interface is preferred over an interface technique that does impose an overhead on execution of typical code within the independent software layers.

3 Analytical Evaluation

Each of the three alternatives for interface implementation is evaluated here according to the criteria discussed in Section 2. Table 1 reports our analysis of the scores, using the encodings Good, Fair, and Poor. The remainder of this section discusses the rationale for the assigned ratings.

Table 1. Subjective Scores for Available Interface Techniques

Evaluation Criterion	1	2	3	4	5	6	7
JNI	Poor	Poor	Poor	Good	Poor	Poor	Fair
RTSJ	Fair	Fair	Good	Fair	Good	Fair	Poor
Core	Good	Good	Good	Good	Good	Good	Good

3.1 JNI

We rate JNI poorly for all three of the software engineering criteria because JNI developers have to think very carefully about many issues when connecting native code with Java code. For example, the JNI programmer must understand non-portable details regarding the priority mappings between Java threads and native threads and must understand the relationship between the implementation of Java synchronization

and the native RTOS synchronization techniques. The JNI programmer needs to carefully design techniques to avoid priority inversion. Accessing Java objects from C code requires strict adherence to special protocols designed to abstract Java object representations. This requires much more code than would be required if the C program were dealing with traditional C structures. Finally, the JNI interface does not enforce any type checking or encapsulation and does not support any sort of object-oriented inheritance.

We rate JNI as good in category 4 because Java's security model is very good at protecting native components from being compromised by Java application software. On the other hand, we rate JNI as poor in category 5 because there are no protections in place to prevent native code from compromising the integrity of the Java virtual machine.

We rate the interface efficiency as poor because each time control passes between native and Java code, significant marshalling of information must take place. The typical JNI call is more than ten times as expensive as a typical Java method invocation.

In general, it would appear that the performance of traditional C code and traditional Java code are unaffected by the presence of the JNI interface. Nevertheless, we rate this performance as mediocre for two reasons. First, when C code is accessing objects shared with the Java virtual machine, it runs much more slowly than when it is accessing traditional C structures residing in its own heap. This is because the C code must follow special protocols involving macros and function invocations to access the Java objects. Second, the requirement to efficiently support the JNI interface imposes implementation tradeoffs on the Java virtual machine architecture which slows execution of traditional Java code even if that code is not making use of the JNI interface.

3.2 RTSJ

RTSJ represents an improvement over JNI in the details that must be considered and in the code that must be written to implement both sides of the interface. Thus, we rate RTSJ as fair for both criteria 1 and 2. One reason we do not offer good ratings is because **NoHeapRealtimeThreads** cannot use traditional Java techniques to synchronize with traditional **java.lang.Threads** or with **RealTimeThreads**. Instead, they must devise ad hoc sharing protocols that allow certain objects to exist at some times within the domain of the hard-real-time world and at other times within the domain of the soft-real-time or non-real-time world. RTSJ allows objects to be passed between domains by placing them onto specially designed wait-free queues. The only objects that can be shared between these two worlds are objects residing in **ImmortalMemory**. Since these objects are not subject to automatic garbage collection, developers must take responsibility for manually managing the allocation and recycling of such objects as they move in and out of the hard-real-time domain. Most developers of hard real-time code program defensively. They will generally take extra precautions to protect the integrity of their hard real-time software from interference by other software components. Because of this, most will be wary of releasing their

critical data structures to the control of non-real-time software components. Instead, they will often desire to keep their critical data structures to themselves and will only copy information into shared objects when necessary to communicate with non-real-time components. Finally, note that RTSJ programmers must invent their own ad hoc techniques for throttling the flow of information through a wait-free queue. What is the real-time program to do if it cannot place critical information into the queue because the queue is full, or if a reusable buffer does not come back from the non-real-time world in sufficient time to fill it with the next requested information?

We rate RTSJ as good in category 3 because information is passed between hard-real-time and non-real-time domains in the form of Java objects. We emphasize that these are special objects (residing in **ImmortalMemory**) and special care must be taken to ensure that the methods associated with these objects can be invoked both from within a **NoHeapRealtimeThread** and a **java.lang.Thread**. We do not penalize RTSJ in the category 3 rating for these issues because we consider RTSJ to have already paid the price of these sins in its category 1 and 2 ratings.

We rate RTSJ as mediocre in protecting the hard-real-time domain from mistakes made within the traditional Java domain. On the one hand, RTSJ uses standard Java encapsulation techniques to protect hard-real-time components from traditional components. On the other hand, there is no other partitioning beyond this to protect the hard-real-time domain from the Java domain. The two domains are so tightly integrated that if the Java virtual machine crashes, the hard-real-time environment also crashes. If a traditional Java thread consumes all of the virtual machine's memory, the hard-real-time environment will not be able to find memory to represent scoped memory regions. Also, if the traditional Java code fails to return the immortal objects that carry information by way of wait-free queues from the hard-real-time domain to the Java domain, the hard-real-time application software will not be able to continue to send information to the traditional Java domain.

With the RTSJ, **NoHeapRealtimeThreads** are not allowed to access traditional heap objects. This helps to prevent **NoHeapRealtimeThreads** from interfering with the activities of the traditional Java application code. Thus we give category 5 a good rating. Unfortunately, most real-time developers would rather have a higher rating in category 4 than in category 5.

We rate category 6 as fair because the protocols for sharing information between the hard-real-time and non-real-time domain are much heavier than traditional Java-synchronized sharing. Passing information between domains usually requires allocation of a reusable buffer from an existing pool and copying of information into the reusable buffer. Then the object must be inserted onto one end of a wait-free queue and subsequently removed from the other end of the wait-free queue. Even so, this interface is more efficient than the typical JNI implementation.

We rate category 7 as poor because the RTSJ imposes a number of run-time checks on code executing both within the traditional Java domain and within the hard-real-time

domain. These checks are required to enforce that **NoHeapRealtimeThreads** never depend on or interfere with operation of the garbage collector.

3.3 Core

We assign good ratings for all three of the software engineering criteria. Using the Real-Time Core Extensions, the interface between hard-real-time and traditional Java objects is represented by a core-baseline method. To the traditional Java developer, this looks like a standard object-oriented Java method invocation. The hard-real-time developer who implements the core-baseline method writes this code the same way he would write all of the other code in his hard-real-time application. He inserts a special marker into the body of the method to inform the core compiler that it must translate this method according to the calling conventions of the traditional Java execution environment rather than the streamlined conventions of the hard-real-time environment. All of the implementation complexity is hidden behind the core compiler. Furthermore, because the interface is implemented using standard Java object-oriented disciplines, it takes full advantage of inheritance and polymorphism.

Categories 4 and 5 also earn good scores. There is a strong partition between the hard-real-time and traditional Java domains, so strong that the hard-real-time execution environment can continue to run even if the Java virtual machine crashes. Memory is permanently partitioned between the two domains so there is no risk that a runaway Java application will consume memory that ought to be reserved for hard-real-time activities. Further isolation is provided by compile-time enforcement that prevents traditional Java objects from directly manipulating core objects and prevents core objects from referring to traditional Java objects. A protocol is provided to allow traditional Java objects to obtain handles to specific core objects, but these handles can only be used to execute the core-baseline methods that were implemented by the trusted hard-real-time developer. There is no way for a traditional Java thread to access the instance variables of the core objects.

The strong partitioning that exists between the hard-real-time and traditional Java domains enables a very efficient interface implementation. Because the core domain never refers to traditional Java objects, there is no need to marshall object representations on this interface. And even though we prohibit traditional Java threads from directly accessing the instance variables of core objects, the core programmer can write very efficient accessor methods which will be compiled and even in-lined by the core compiler. This particular protocol was designed to allow traditional Java threads to run according to their standard code model, without any changes required to the Java virtual machine when it is attached to a core execution environment. Likewise, the protocol was designed to allow the hard-real-time Java threads to run at full speed comparable to C or C++ code. No run-time checks are required. Thus we give good ratings to the Core for categories 6 and 7 as well.

4 Empirical Evaluation

Aonix is currently implementing a subset of the Core specification. In this section, we provide a summary of empirical results obtained by measuring the alternative interface technologies as they .

4.1 Experimental Benchmarks

Two representative applications were selected for the purpose of evaluating the interface between hard-real-time and soft-real-time components. These are synthetic benchmarks based on real-world scenarios. We describe the two benchmarks below. Additional representative workloads are under development. Source code for the two benchmark applications described below and for the additional benchmark applications that are under development will be available from the authors of this paper.

Zero-Copy Network Stack. This application represents a simplification of the sort of work carried out by a network router or gateway.

The hard real-time component represents the lower level control plane of a network protocol stack. This routine repeatedly selects a random packet from a collection of 128 previously allocated and initialized packets. It examines certain fields of the packet to determine if this packet requires any special handling. In our simulation, an average of one packet in every 17 requires special handling and is forwarded to the soft-real-time management-plane software for further processing. The packets that do not need any special handling are queued directly for another hard-real-time thread to do further output processing.

The soft-real-time management plane examines each packet that is forwarded for special handling, makes some minor adjustments to the content of the packet, and then queues the packet for further output processing within the hard-real-time domain.

For our benchmark, we measure how much time is required to process 1,000 packets through the soft-real-time code. This corresponds to total processing of approximately 17,000 packets. Although our benchmark result is reported as an elapsed time, this value is inversely proportional to the maximum rate at which packets could be reliably handled by a network element implemented based on the respective interface technology.

Streaming Data Recorder. This application represents the sort of work that might be carried out in a streaming data recorder that is part of, for example, a seismic monitor or a black-box recorder for a commercial aircraft.

In this application, the hard-real-time component is responsible for writing buffered data to the physical non-volatile media. The soft-real-time component is responsible for gathering data samples to be recorded. In this simulation, the soft-real-time component simply generates random integer values and stores them into a buffer holding 256 entries. Each time the buffer fills, the soft-real-time component hands the buffer over

to the hard-real-time domain and begins filling a second buffer. The hard-real-time component simulates the streaming data write operation by summing the integers in the buffer and then sleeping for 100 microseconds before returning the emptied buffer back to the soft-real-time component.

For this benchmark, we measure how much time is required to process 16,384 buffers of 256 entries each. As with the other benchmark, this result is also reported as an elapsed time. The reported value is inversely proportional to the maximum rate at which information could be collected and streamed to non-volatile media using the respective interface technology.

Comparison Between Workloads. The zero-copy network stack is very typical of layered network protocol implementations in which 90% of the CPU time is spent in 10% of the code. As is typical in such systems, the 10% of the code that is most performance critical is the lower level hard-real-time code. In this example, only 1 out of 17 packets is bubbled up to the soft-real-time management layer. And even when a packet is sent to the management layer for special processing, the management layer only examines and manipulates selected fields of the packet's header. In contrast, the lower layer software processes every packet and deals with the entire data payload of each packet. This benchmark places greatest emphasis on performance of the low-level hard-real-time code. The need to pass information from the hard-real-time to soft-real-time world is relatively rare, as is the need to examine or manipulate shared objects from the soft-real-time domain.

The streaming data recorder provides a much more balanced sharing of computation between the soft-real-time and hard-real-time domains. For each entry in the buffer, the soft-real-time thread generates a random number and inserts this into the buffer. Similarly, the hard-real-time component examines each entry in the buffer as a simulated step in the streaming write operation. Thus, this benchmark places approximately equal emphasis on efficiency of soft-real-time and hard-real-time access to shared objects. The shared buffer objects themselves are passed between the hard-real-time and soft-real-time domains relatively rarely in comparison to the frequency of operations that access the fields of the shared objects.

4.2 Measured Platforms

We implemented the two sample applications using each of the three interface techniques. We ran all of our measurements on the same computer, a dual-processor Pentium II processor running at 350 MHz. The system has 256 MBytes of RAM. It is running Timesys Linux version 3.2.214smp, configured to use only one of the processors. We chose to run in single-processor configuration so as to reduce uncertainty as to what exactly is being measured. Except for the measured workload, each measurement was taken with the processor in a quiescent state.

JNI. To measure the cost of the JNI interface, we ran the JNI versions of the two applications on a pre-release version of PERC 4.1. All shared data objects were allocated in

the Java heap. We chose to measure JNI performance on PERC rather than Sun's JDK product because the Java threads are to represent soft-real-time behavior.

RTSJ. We evaluated the efficiency of the RTSJ interface by measuring the RTSJ version of the two applications running on RTSJ reference implementation 1.0. For this test workload, all of the shared objects were allocated from within **ImmortalMemory**. Hard-real-time activities were represented by **NoHeapRealtimeThread** objects. We chose to represent soft-real-time activities with **java.lang.Thread** objects. This represents a compromise from the original intent of the demonstration, forced by the lack of real-time garbage collection in existing RTSJ implementations.

The hard-real-time components trusted the soft-real-time Java code to deal appropriately with the packet and buffer objects that were shared between the two domains. Thus, it was not considered necessary to copy data out of private hard-real-time buffers before communicating their contents to firm-real-time Java components.

We are aware that TimeSys has now completed a commercial implementation of the RTSJ, named JTime, and we would have preferred to evaluate the efficiency of the RTSJ interface by measuring that implementation. Unfortunately, Timesys refused our repeated requests to license the JTime implementation for the purposes of conducting this research. Individuals interested in the performance of JTime may obtain test programs directly from the authors and conduct their own tests of the JTime platform.

Core. We have not yet completed implementation of the Real-Time Core Extensions so the measurements reported here are only approximations based on the currently available partial implementation. In the Core programs, all shared objects are allocated from within the core domain. In the simulation, the core components are represented by C code written to follow the sharing protocol that was designed for integration of core components. We consider this a fair representation of core performance, especially in light of the fact that one of our next implementation steps is to automate the translation of real-time Core code into the equivalent C code.

For purposes of this simulation, we have instrumented a version of PERC 4.0 to recognize classes that implement a particular interface and were loaded by a particular class loader as having core-baseline methods. These methods are treated like JNI methods but with a less costly (more efficient) interface implementation.

One notable weakness of the existing partial implementation is that all access to shared Core objects from within the Java domain must include an invocation of a native method. A future version of the Core implementation will in-line the implementation of many core-baseline methods.

4.3 Experimental Results

The results of our empirical evaluation are tabulated in Table 2.

Table 2. Benchmark measurements (ms)

Benchmark		Trial 1	Trial 2	Trial 3	Average
Zero-Copy Network Stack	JNI	27,944	21,315	23,159	24,139
	RTSJ	10,395	10,359	10,309	10,354
	Core	2,160	2,090	2,060	2,103
Streaming Data Recorder	JNI	4,802	4,757	4,749	4,769
	RTSJ	191,694	191,563	191,516	191,591
	Core	8,920	9,000	8,850	8,923

Discussion of Results. From the first benchmark, it is interesting to note that even though C code is known generally to be much faster than Java code, the JNI implementation runs over 10 times slower than the Core solution and over twice as slow as the RTSJ approach. This is presumably because of the very high cost of “crossing the barrier” between the JNI and JVM domains. The Core hard-real-time components are also written in C, but the protocol for sharing objects between the Core and JVM domains is much simpler. This Core sharing protocol is independent of JVM technologies so the benefits shown here would accrue to integration with other JVMs as well.

In the second benchmark, the costs of crossing the interface between hard- and soft-real-time domains are less significant. Much more significant is the efficiency with which each approach independently executes soft-real-time and hard-real-time components. RTSJ is over 20 times slower than Core. This is presumably because all of the RTSJ components are interpreted, whereas the Core solution benefits from the PERC JIT compiler. Note that interpretation of RTSJ code runs even slower than interpretation of traditional Java code because RTSJ requires extra run-time checks to enforce memory partitioning and dynamic scoping.

Presumably, the commercial JTime implementation would perform better than the RTSJ reference implementation. However, we were not given an opportunity to evaluate that environment. It is our understanding that performance even of JTime, which offers limited forms of ahead-of-time compilation, is generally disappointing, and this may be one of the reasons Timesys has elected to protect the product from public scrutiny. Our difficulties obtaining access to the commercial product hint of larger issues that must be addressed in the real-time marketplace. Discussion of economic issues that accompany dependency on proprietary technical products focused on narrow market niches are beyond the scope of this paper.

Given the huge penalty paid by JNI in the first benchmark, it is surprising to see it perform significantly better than the other two alternatives for this benchmark. Our analysis of this anomaly is that this particular benchmark was particularly sensitive to a shortcoming in our prototype Core implementation. In particular, for each value inserted into the buffer, the soft-real-time component must make a core-native method

call. This is because the data buffer is a core object which does not live in the Java heap and does not follow the traditional Java heap-access protocols. Even though the protocol for core-native method calls is much more efficient than the protocol for traditional JNI native method calls, it is still a fairly expensive operation. A future improvement to our Core implementation will replace the core-native method calls with in-lined machine code. This in-lined machine code will run even faster than JIT-translated Java code because it will circumvent the protocols required for coordination with garbage collection and relocation of objects. Thus, we are confident that a future version of the Core will run significantly faster than the JNI implementation on the Streaming Data Recorder application.

There is one last point we desire to emphasize relating to these two benchmarks. It is important to note that the alternative technical approaches yield radically different performance, spanning a range of over 40-fold. Even more noteworthy, the best technology for one of these challenges was the worst for the other! These observations underscore the importance of (a) recognizing that different technologies are better suited to different problems, (b) it is very difficult to predict the performance of a given technology on your specific problem based on the experience of others with other problems, and finally (c) the stakes are quite high as are the risks (we're not talking about performance variations of only plus or minus 30%; we're talking about orders of magnitude). This work is ongoing. Aonix will continue to expand its repertoire of test applications in order to better understand the tradeoffs represented by particular interface technologies.

Qualitative Observations. This was the authors' first experience developing software using the RTSJ APIs. Briefly, we summarize here some of the lessons we learned from this experience.

First, we found it was easier than originally anticipated to share objects between the hard- and soft-real-time domains. However, there are subtle coordination issues, the importance of which should not be underestimated, and we expect that less "formal" developers would be likely to stumble into certain pitfalls, mentioned below. We used wait-free queues to pass objects between the two domains. We were confused by the documentation which states, for example, that the `read()` method of a `WaitFreeWriteQueue` may be called by more than one writer [*sic*], which seems to contradict the constructor API, which requires as one of its arguments a reference to the `java.lang.Thread` object that represents the writer. For our application, we needed only one writer, so this was not an issue for us. But we were left wondering how different RTSJ implementors might end up interpreting the specification in different ways, leading to incompatibilities between code developed for one implementation versus another.

For both of our applications, we would have liked to have had a built-in blocking queue upon which a soft-real-time thread could block waiting for information from the hard-real-time domain, and upon which a hard-real-time thread could block waiting for a slot in the queue to become available for transmission of information to the soft-

real-time domain. In the absence of this, we were forced to introduce busy loops that would repeatedly poll the queue status and sleep until the queue was ready. We were tempted to implement our own blocking queue. All of the facilities are in place to make the implementation quite straightforward. This, however, is where an unwary RTSJ programmer may create problems for himself, or even worse, for the folks down the hall who are writing code that has to run on the same platform. The problem is as follows:

1. Suppose you have a **NoHeapRealtimeThread** running at real-time priority 20, and you decide that you don't mind if this particular thread blocks waiting for certain coordination with a non-real-time thread. So you use a traditional Java monitor to arrange that the **NoHeapRealtimeThread** will block until the Java thread performs a certain action.
2. Suppose further that the **NoHeapRealtimeThread** attempts to enter this shared monitor in order to see if the Java thread is ready for a coordinated handoff, but the monitor is locked because the Java thread is already accessing certain shared data structures associated with this monitor.
3. Because of the way the RTSJ specification is written, the **NoHeapRealtimeThread** will endow its priority to the Java thread, allowing it to run at real-time priority 20 until it advances to the point of releasing its lock on the shared monitor.
4. And finally, suppose the Java thread cannot make forward progress because the Java VM is in the middle of garbage collection. The Java thread, and the garbage collector upon which it depends for its forward progress, are now running at effective priority 20 and will continue to do so until garbage collection completes.

The programmer who introduced this shared monitor thought it would be ok because, as far as he was concerned, his real-time task had nothing to do until the Java domain produced another value for him to process. The problem is that every other **NoHeapRealtimeThread** with priority equal or lower than 20 is also prevented from running while garbage collection completes. In our opinion, it's a little too easy for programmers to introduce these sorts of priority inversion bugs into an RTSJ system. These sorts of problems are even more troubling than the scoped-memory incompatibilities that RTSJ developers encounter when they attempt to invoke existing Java library code from within **NoHeapRealtimeThreads**. When library code invoked from a **NoHeapRealtimeThread** violates the scoped-memory restrictions, this throws an exception, so the programmer can figure out that these libraries are incompatible with **NoHeapRealtimeThreads**. No such exception is thrown when library code attempts to synchronize on objects that might be shared between the hard-real-time and non-real-time domains.

5 Conclusions

There are various ways to connect soft-real-time and hard-real-time software components together. Each has particular strengths and weaknesses. When evaluated according to the criteria detailed in this paper, the Real-Time Core approach is superior to

available alternatives. Further research may reveal alternative criteria that are of equal or greater importance to particular problem spaces and may reveal additional ways to evaluate alternative approaches in terms of the criteria already identified.

6 Acknowledgments

We thank the paper's reviewers for several insightful comments. We also acknowledge support from the U.S. Navy under contract N00178-08-C-3087.

References

1. Liang, S. "The Java Native Interface Programmer's Guide and Specification", Addison-Wesley Publishing Co. 320 pages. June 10, 1999.
2. Zukowski, J. "Mastering Java 2, J2SE 1.4", Sybex. 928 pages. April 2002.
3. Keogh, J. "J2EE: The Complete Reference", McGraw-Hill Osborne Media. 904 pages. Sept. 6, 2002.
4. Keogh, J. "J2ME: The Complete Reference", McGraw-Hill Osborne Media. 768 pages. Feb. 27, 2003.
5. Bollella, G. et al. "The Real-Time Specification for Java", Addison-Wesley Publishing Company. 195 pages. Jan. 15, 2000.
6. "Real-Time Core Extensions", J Consortium. 170 pages. Sept. 2, 2000.
7. Barnes, J. "Programming in Ada 95". Addison-Wesley Publishing Company. 720 pages. June 10, 1998.