Applying Java to the Domain of Hard Real-Time Systems

Kelvin Nilsen

Applying Java to the Domain of Hard Real-Time Systems

Kelvin Nilsen, Ph.D.

Aonix North America
125 E. Main St., #501
American Fork, UT 84003
(+1) 801-756-4821
kelvin@aonix.com

Abstract: Organizations are attracted to Java because the language has proven more economical than C and C++. Companies that have made the switch to Java typically find that they are twice as productive during development of new functionality and five to ten times as productive during reuse of existing code. Organizations that develop in Java also observe decreased software error rates, increased software reuse and longevity, and improved recruitment of competent developers.

Special hard real-time Java development practices enable proofs of resource needs and determinism. Early analysis demonstrates that the hard real-time Java platform runs in less than a tenth the memory footprint and up to three times faster than traditional Java for typical hard real-time tasks. Determinism is on par with typical C code, offering more than a 20-fold improvement over the timing predictability of traditional Java.

Keywords: hard real time, safety certification, Java, real-time specification for Java

1. Introduction

In the traditional information processing domain, Java's high-level support for scalable composition of software modules is among its most highly valued benefits in comparison with C and C++ [2, 3].

As an object-oriented programming language, the natural style of programming in Java requires dynamic allocation of various temporary objects. In traditional Java, these objects are allocated in the heap, and their memory is reclaimed automatically by a garbage collection system after the objects are no longer in use. Real-time garbage collectors offering sub-ms preemption latencies are available now from multiple vendors. But real-time garbage collection adds complexity, jitter, and performance overhead that are usually considered inappropriate for hard real-time software. In 2000, the Java Community Process introduced the Real-Time Specification for Java (RTSJ) [1]. The RTSJ introduced the notion of stacked memory scopes to serve the temporary allocation needs of hard real-time software. By allocating temporary objects within stacked scopes, the allocation and deallocation becomes deterministic. But using memory scopes is very difficult and error prone.

This paper presents an approach that layers additional abstraction and static analysis on top of the RTSJ scope stacks in order to simplify development, ease integration of independently developed components, and enable proofs that code components will not fail because they violate scoped-memory protocols. The approaches are similar to the use of SPARK abstractions with the Ada programming language [4]. Several important factors combine to motivate the design of this technology:

1. The safety-critical development community desires to leverage commercially popular technologies such as Java rather than specialized niche technologies such as Ada because free-market competition among a broader supplier base delivers higher quality technologies for lower prices.
2. Safety-critical systems need to pass stringent certification requirements. Certification authorities expect proofs that programs function correctly. Certification guidelines also recommend elimination of all dead code, and require independent proofs that any deployed dead code is "deactivated", meaning the code will never be executed. Run-time assignment checks are very problematic, because they add significantly to the burden of proof associated with certification of safety-critical software.
3. The size and complexity of safety-critical software continues to grow. Because the costs of developing and certifying safety-critical software modules are so high, there is increasing pressure to reuse safety-critical modules and their certification artifacts. This demands a strong separation of concerns between independently developed software modules. Traditional whole-program analysis tech-

niques are problematic because a small change to a single module may invalidate all of the certification artifacts in the system. For many development organizations, this is cost prohibitive.

4. The desire to reuse safety-critical software modules also scales to the domain of mission-critical systems that do not require safety certification. Whenever there are common functional requirements between safety- and mission-critical systems, the industry generally prefers solutions that allow them to maintain a single implementation of the common functionality. Thus, there is a general desire to allow code written to the stringent safety-critical requirements to be deployed in mission-critical systems which may be larger, more complex, and more dynamic than typical safety-critical deployments. For this reason, there is a need to support dynamic loading of safety-certified real-time modules.

2. Related Work

Others have examined the challenge of using type systems to make the use of scope- or stack-based memory allocation safe. The Cyclone system seeks to establish a safe alternative to C [5]. Cyclone introduces a type system to support safe stack allocation and deallocation of memory, and combines this with explicit safe deallocation of heap objects, along with automatic conservative garbage collection. A project at MIT has also developed a type system for safe region-based memory management of RTSJ programs [6]. Though the MIT system has many similarities with the approach described in this paper, a key difference deals with identification of scope levels in argument lists. The MIT approach requires that every scoped argument be associated with a particular named scope, whereas the approach described in this paper, for the most part, simply identifies that certain referenced objects may reside in scoped memory, without requiring programmers to differentiate one scope from another. For certain special circumstances, the type system described in this paper allows programmers to require that certain incoming scoped arguments reside in more outer-nested scopes, or at the same scope level, as certain other scopes that are relevant to execution of a particular method or constructor. While the MIT approach is capable of describing more complicated algorithms and data structures, the MIT type system imposes far greater restrictions on the generality of individual software modules. This specialization of code interfaces hinders software reuse and increases the volume of code that must be written and certified. A third approach to type-safe scoped memory has been described in reference [7]. This approach makes use of existing aspect-oriented programming tools to analyze and transform stylized Java source code.

3. Thread Stack Memory Model

This paper's approach to hard real-time Java builds on the notion of scoped memory, but hides the RTSJ APIs that manipulate memory scopes. Instead, hard real-time Java programmers describe their intentions with respect to use of scoped memory by using annotations which can be statically analyzed and enforced at compile time. The hard real-time Java profile also supports the notion of immortal memory, which represents the outer-most memory scope. Objects allocated within the immortal memory region will not be reclaimed.

The need to support temporary memory allocation within real-time programs has been well motivated. At the same time, there is general agreement that developers of hard real-time modules do not require the full generality and flexibility offered by automatic garbage collection.

Within the RTSJ, hard real-time programmers are encouraged to use the *LTMemory* abstraction. This service makes it possible to allocate new memory within a dynamic scope in time that is proportional to the size of the allocation request. A developer of hard real-time software must face several significant difficulties with the use of this abstraction:

1. Knowing how big to make an *LTMemory* region in order to reliably support execution of a particular real-time module is quite difficult and error prone. Furthermore, it is non-portable between different compliant RTSJ implementations.

1. Instantiation of an *LTMemory* region is not a hard real-time operation. There is no bound on how much time this will take, and there is, in fact, no guarantee that a request to instantiate an *LTMemory* region will succeed even if there is sufficient available memory in the system at the time of the request. This is because memory may become fragmented during the course of a program's execution.

Many RTSJ programmers overlook these difficulties with use of the *LTMemory* abstraction. They regularly allocate and discard *LTMemory* objects, and successful execution of test programs instills confidence that the code will work reliably in the field. This is a dangerous practice, because it is not generally possible to test all of the different ways that the allocation pool might become fragmented. Further, the program may not behave the same way if it is moved to a different vendor's compliant RTSJ implementation, or even if the

same vendor provides a new maintenance release of the same RTSJ implementation.

RTSJ programmers who understand and appreciate the risks of memory fragmentation find that the only way to reliably and safely use *LTMemory* abstractions in their hard real-time code is to allocate all of the *LTMemory* objects that their application might need during initialization of the application. This adds significantly to the difficulty of implementing and maintaining the software, and adds considerably to the amount of memory required for reliable execution of the application since many of the *LTMemory* instances allocated during startup sit idle throughout most of the program's execution.

This paper's hard-real-time Java profile addresses these issues by providing static analysis tools to determine the amount of memory required to execute particular program modules and by requiring all creation and destruction of scopes to follow a strict LIFO (stack) ordering.

At startup, all of the workspace memory available to support execution of the program is set aside as the run-time stack for the main hard real-time Java thread. If the application needs to support more than a single thread, the main program carves memory from its run-time stack to represent the run-time stacks for each of the threads it spawns. Figure 1 illustrates the organization of the main thread's run-time stack immediately after it has spawned three new threads. This illustration assumes that all three threads were spawned from the same context within the main thread. Note that space has been reserved within the main thread's stack to allow the main thread to continue to populate its run-time stack. Note also that it is essential at this point in the program's execution to know the amount of memory that must be reserved to represent each of the spawned thread's run-time stacks. These stacks need not be the same size. In a typical application, the size of each stack is custom-tailored to the needs of the given thread.

As execution proceeds, each of the three spawned threads and the main thread will continue to populate their respective stacks. Assume the stack memory is organized as shown in Figure 2 at a subsequent execution point.

The scoped memory usage guidelines, as defined in the RTSJ, allow inner-nested objects to refer to objects residing in more outer-nested scopes, but forbids references that go in the opposite direction. These scope-nesting restrictions guarantee that there will never exist a dangling pointer from an outer-nested object to an inner-nested memory location that

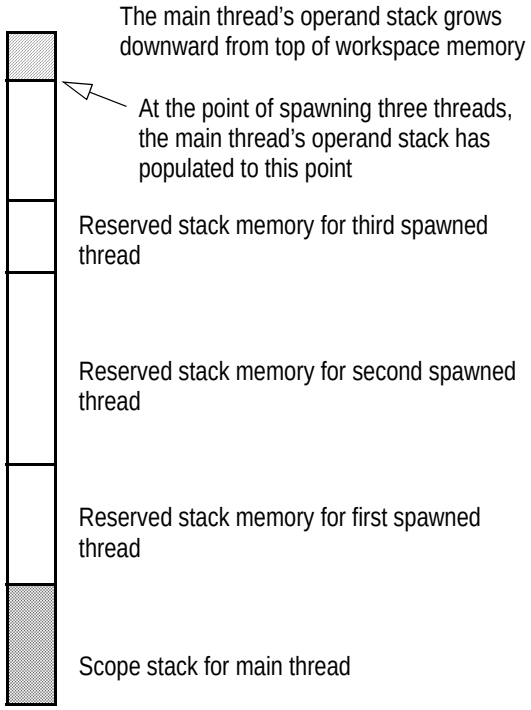

Figure 1. Main Thread Stack After Spawning Three Threads

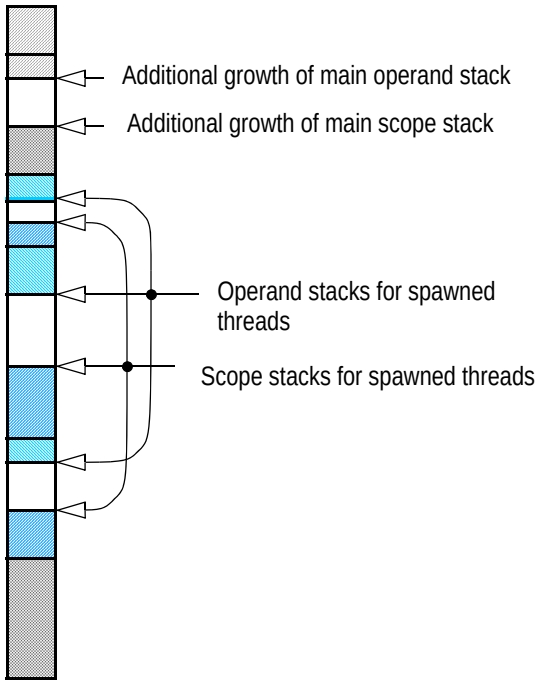

Figure 2. Stack Organization After Each Thread Has Populated Its Stack

no longer exists because its inner-nested scope has been reclaimed.

Given these conventions, any data structures that must be shared between multiple threads need to reside either in immortal memory (the outer-most scope, which is never reclaimed) or must exist within the parent or some other ancestor thread's stack below the point at which the descendent thread was spawned. Shared objects do not necessarily need to exist at the time the subthreads are spawned, but the memory allocation contexts within which the shared objects will eventually be allocated must be set aside and entered within the parent thread's stack before the point at which the child thread is spawned.

A special protocol must be enforced to support these conventions. In particular, whenever an ancestor thread desires to unwind its thread stack beyond the point at which it had spawned descendent threads, it must wait for all of the threads that were spawned from this context to terminate execution before it returns from this context. Otherwise, we might create a situation in which a spawned child thread refers to non-existent objects that once resided within the parent's scope.

This special protocol is enforced by the hard real-time Java compiler. Whenever a method spawns the execution of another thread, the code generator inserts into the finalization code for this method a request to unconditionally wait for termination of the spawned thread. This assures that any scope-resident objects shared between this method and its spawned subthread persists as long as the subthread is running.

4. Type Attributes

Every reference variable within a scope-safe program is classified as having one or more of the following five attributes.

The *scoped* attribute marks variables that may, but need not, hold references to objects residing in scoped memory. A variable that does not have the *scoped* attribute is not allowed to hold references to objects residing in scoped memory.

The *immortal* attribute marks variables that are known to hold references to immortal memory, or *null*. The *immortal* attribute is mutually exclusive with the *scoped* attribute.

The *array* attribute is used in combination with the *scoped* attribute to identify variables that may hold references to arrays residing in scoped memory, the elements of which are considered to be *scoped* variables. In the case that an element of a *scoped-array* object is itself a reference to another array, the referenced array is considered to have the *scoped-array* attribute. Note that some reference arrays may have the *scoped* attribute without having the *array* attribute. Such an array may reside in scoped memory, but its array elements must reside in immortal memory.

The *captive* attribute is used in combination with the scoped attribute to identify variables whose contents cannot be copied into static or instance fields. *Captive* values are only allowed in local variables and arguments.

The *result* attribute identifies local reference variables whose value might be returned from the currently executing method.

5. Meta-Data Annotations

The hard real-time Java system uses Java Standard Edition 5.0 style meta-data annotations to characterize the type attributes associated with method and constructor arguments, method return values, instance variables, and static variables. Marking static variables as *@Scoped* anticipates that certain classes may be dynamically loaded and unloaded within inner-nested scopes. In these circumstances, a dynamically loaded class might have static variables that refer to objects allocated in more outer-nested scoped memory regions. The annotations described in this section are a small subset of the full set of annotations designed for the scope-safe type system. Space does not allow presentation of the full system. See reference [9] for more detail.

The *@Scoped* annotation applies to instance and static fields, methods, and method and constructor argument declarations to indicate that the corresponding variable has the *scoped* attribute. When applied to a method, this annotation denotes that the method may return a reference to an object that resides in scoped memory.

The *@ScopedThis* annotation applies to instance methods and constructors to indicate that within the corresponding method or constructor body, the *this* variable has the *scoped* attribute. In order to construct an object within scoped memory, the constructor's *this* variable must have the *scoped* attribute.

The following implementation of a complex number abstraction demonstrates the use of both annotations:

```
public class Complex {
   float real, imaginary;

   @ScopedThis public Complex(float r, float i) {
      real = r;
      imaginary = i;
   }

   public static boolean lengthGT(@Scoped Complex c1,
                                  @Scoped Complex c2) {
```

```
        return (Math.sqrt(c1.real * c1.real +
                          c1.imaginary * c1.imaginary) >
                Math.sqrt(c2.real * c2.real +
                          c2.imaginary * c2.imaginary));
    }
}
```

The *@CallerAllocatedResult* annotation applies to methods to denote that the caller specifies the context within which the method's return object will be allocated. Consider introduction of a *multiply()* method into the *Complex* class as an example of this annotation:

```
@CallerAllocatedResult @ScopedPure
public Complex multiply(Complex arg) {
    float r, i;
    r = this.real * arg.real - this.imaginary * arg.imaginary;
    i = this.real * arg.imaginary + arg.real * this.imaginary;
    return new Complex(r, i);
}
```

This example introduces the *@ScopedPure* annotation, which is shorthand to denote *@ScopedThis* and *@Scoped* for all reference arguments.

There are certain situations under which a constructor may desire to return an object which contains references to other more outer-nested objects. Consider, for example, the following code:

```
public class String {
    @Scoped StringBuilder data;
    int offset, length;

    @ScopedThis
    public String(@Scoped StringBuilder sb, int off, int len) {
        data = sb[1];
        offset = off;
        length = len;
    }
}
```

The type system requires that any *@Scoped* arguments to a *@ScopedThis* constructor reside in scopes that are at equal or more outer-nested scope level than the object to be constructed[2]. In most constructor invocations, demonstrating compliance with this requirement is trivial, because the typical object is instantiated in the inner-most scope. However, invocation of constructors from within caller-allocated result methods, for example, may introduce violations of this rule.

In general, the type checker enforces that any *@Scoped* arguments passed to inner-nested caller-allocated result methods, or to a constructor that initializes an object to be returned from a caller-allocated result method originate outside the current method's scope. Through inductive reasoning, the hard real-time Java verifier proves that the *@Scoped* arguments passed to any *@CallerAllocatedResult* method and to any constructor reside in scopes that enclose the object to be constructed.

In certain situations, the code that invokes a caller-allocated result method may choose to request that the caller-allocated result be placed in immortal memory. In this circumstance, the hard real-time Java verifier requires that all *@Scoped* arguments passed to the caller-allocated-result method refer to objects residing in immortal memory.

Programmers have an alternative annotation to mark the *scoped* arguments of constructors and caller-allocated-result methods which might refer to objects residing in scopes that are nested internal to the scope of result object. These arguments are marked with the *@CaptiveScoped* or *@CaptiveScopedThis* annotations. The type system forbids constructors from copying *captive-scoped* arguments to the fields of the newly constructed object.

Another special annotation, *@NestedReentrantScope*, marks classes for which certain private *@Scoped* instance variables and all *@Scoped* arguments sent to, and @Scoped values returned from the instance methods are known by the type system to reside at the same scope that holds *this* object instance. The type checker enforces these invariants by restricting allocation of the object itself, restricting allocations performed by the object's instance methods, and restricting assignments to the object's private instance fields. This annotation is useful when building data structures to represent, for example, balanced binary trees, arbitrary precision numbers, or variants of the *java.util.collections* libraries. Space constraints do not allow for a complete description of the type checking associated with use of the *@NestedReentrantScope* or *@ReentrantScope* annotations. See reference [9] for a more thorough explanation.

1. The hard real-time Java API introduces a variant of *StringBuilder* that lacks methods to perform mutation on the contents of the character string array. Thus, this implementation is valid.

2. The complete type system described in reference [9] suports an *@AllowCheckedScopedLinks* annotation, which marks methods and constructors that are allowed to perform assignments that require run-time checks. For methods and constructors that carry this annotation, the type system does not enforce that incoming scoped arguments reside in scopes that are assignment compatible with the constructed object.

6. Inference of Local Attributes

Though Java 5.0 meta-data annotations may be associated with the declarations of local (auto) variables, these annotations are not preserved in the class file representation. Since a design goal is to enforce the scope-safe type system through byte-code verification, the scope-safe type attributes for local variables must be inferred from context. The type inference system implements the algorithm described below.

In discussing this algorithm, the term "variable" represents temporary locations on the operand stack and dedicated slots within the method's activation frame. In both cases, a particular variable's lifetime begins when the value of that memory location is first defined, and ends immediately following the last use of that memory location's value. By this definition, a given memory location may represent different variables at different times.

1. All local reference array variables are assumed to be *captive-scoped-array* unless demonstrated otherwise.
2. All other local reference variables are assumed to be *captive-scoped* unless demonstrated otherwise.
3. If this method returns a reference result, mark each of the variables passed as an operand to the *areturn* byte-code instruction as having the *result* attribute.
4. By analysis of a method's data flow, analyze the usage of every *captive-scoped-array* variable. If there exists a path by which its value might be copied to a variable that is:
   a. *scoped-array* (but not *captive-scoped-array*), change the first variable's attribute to *scoped-array*.
   b. *captive-scoped* (but not *captive-scoped-array*), change the first variable's attribute to *scoped-local*.
   c. *scoped* (but not *captive-scoped-array* or *scoped-array*), change the first variable's attribute to *scoped*.
   d. not *scoped* in any form, change the first variable's attribute to *immortal*.
5. By analysis of a method's data flow, analyze the usage of every *scoped-array* variable. If there exists a path by which its value might be copied to a variable that is:
   a. *scoped* (but not *scoped-array*), change the first variable's attribute to *scoped*.
   b. not *scoped* in any form, change the first variable's attribute to *immortal*.
6. By analysis of a method's data flow, analyze the usage of every *captive-scoped* variable. If there exists a path by which its value might be copied to a variable that is:
   a. *scoped* (but not *captive-scoped-array* or *scoped-array*), change the first variable's attribute to *scoped*.
   b. not *scoped* in any form, change the first variable's attribute to *immortal*.
7. By analysis of a method's data flow, analyze the usage of every *scoped* variable.
   a. If there exists a path by which its value might be copied to a variable that is not *scoped* in any form, change the first variable's attribute to *immortal*.
8. By analysis of a method's data flow, analyze the usage of every reference variable that does not have the *result* attribute.
   a. If there exists a path by which its value might be copied to a variable that has the *result* attribute, add the *result* attribute to the first variable's type description.

In order to assure a consistent interpretation of program semantics, the above analysis is performed on the raw byte-code program before all code optimization.

Note that the algorithm described above may result in changes to the attributes of each local variable. When a local variable's attribute changes, the type inference engine must reconsider the type inference impact of all assignments to that variable. The type inference engine iterates in search of a fixed point. The upper bound on running time is n^3 in the number of local variables and assignments contained within a method. Since most methods contain a relatively small number of local variables and assignments, the execution time is generally tolerable.

7. Byte-Code Verification

Space constraints limit the discussion of byte-code verification to a high-level overview. A more complete description is available in reference [9]. Key concepts are discussed in this section.

Localizing references to scope-allocated objects: To enable compile-time assurance of scope safety, the verifier enforces that the content of a *scoped* variable is never assigned to another variable that is not *scoped*. Likewise, it enforces that *captive-scoped* values are never copied to variables that are not identified as *captive-scoped* variables.

Assuring scalable composition of modules: An important benefit that has made Java more popular than C and C++ for traditional information processing applications is the ease with which independently developed modules are assembled into large and complex software systems. An important goal of the hard real-time Java design is to assure that the scope semantics of individual modules are clearly identified in the annotated API definitions of the modules. These annotations make it possible for software developers and maintainers to easily determine whether particular modules compose simply by examining the interface declarations for those modules.

To enable scalable software development, the verifier enforces consistency between overriding method declarations in subclasses and superclasses. If, for example, the superclass definition of a method declares its first argument to have the *@Scoped* attribute, all subclass implementations of the same method must also declare the first argument to be *@Scoped* or *@CaptiveScoped*. And if a method is declared with *@ImmortalAllocation*, all overridden superclass methods must also carry the *@ImmortalAllocation* annotation. Also, *@ImmortalAllocation* methods and constructors may only be invoked from contexts that are themselves declared with the same annotation. This assures that programmers do not inadvertantly invoke a method that performs allocation in immortal memory.

Work is under way to integrate the byte-code verifier within the Eclipse development environment, as illustrated in Figure 3. When programmers save their files, the Eclipse build system automatically invokes the PERC Pico verifier and any errors detected by the enhanced byte-code verifier are immediately displayed within the Eclipse edit window.

Enabling code patterns that elide scope checks: In safety-critical systems, developers are required to offer certification artifacts that prove to the satisfaction of peer reviewers and certifying authorities that code runs correctly, without abnormal termination because a run-time assignment check detects an illegal assignment. In the vanilla RTSJ environment, a run-time check accompanies every reference assignment and an exception is thrown if the assignment would create a reference from an outer-nested object to an inner-nested object.

The byte-code verifier is required to perform certain analyses in order to justify that particular assignments are scope safe. For example, when assigning to a *@Scoped* field of an object, the verifier recognizes the following conditions as not requiring a run-time assignment check:

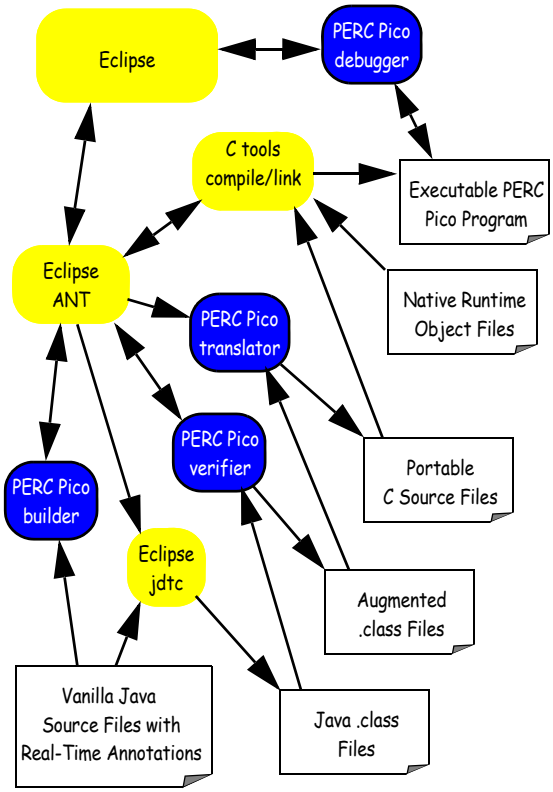

Figure 3. Hard Real-Time Java Development Environment

1. If the object that contains the field to be assigned was just allocated within this thread's inner-most local scope context, the verifier recognizes that any values assigned to this field reside in the same or outer-nested contexts.
2. If the value to be assigned was copied from another reference field (or array element) of the same object, the verifier recognizes that the assigned reference value refers to an object residing in the same or outer-nested context.
3. If the value to be assigned was copied from another reference field (or array element) of an object that was reachable from the object that is being assigned to, the verifier recognizes that the assigned reference value refers to an object residing in the same or outer-nested context.
4. If a reference value being returned from a method was passed in as an argument to the method, or was reachable from one of the objects passed in as arguments to the method, the verifier recognizes that the returned value is visible in the scope of the caller's method.

All of the above-described analyses are based on data-flow analysis of reaching definitions within the

unoptimized byte-code representation of each software module.

More sophisticated byte-code analysis rules deal with more complex software constructs. For example, the verifier enforces at the point of invocation that all of the *@Scoped* (but not necessarily the *@CaptiveScoped*) arguments of a method declared with the *@CallerAllocatedResult* annotation reside in scopes that enclose the scope that holds the caller-allocated result. By inductive reasoning, the byte-code verifier allows the values of incoming *@Scoped* arguments to be assigned to the fields of the caller-allocated result object without requiring a run-time assignment check. Similar analysis enables certain check-free assignments within the instance methods of *@NestedReentrantScope* classes.

8. Layers of Real-Time Abstraction

In systems that require combinations of hard real-time, soft real-time, and non-real-time functionality, it is possible to combine hard real-time Java application components with components deployed on traditional Java or soft real-time Java virtual machines. The mechanism described here allows a hard real-time Java application to be paired with a traditional Java virtual machine, enabling efficient and robust implementation of layered architectures comprised of a combination of soft real-time Java components that use standard edition Java libraries and real-time garbage collection and hard real-time Java components that perform all temporary object allocations with a stack of allocation scopes.

Unlike the wait-free queue mechanism that is provided by the RTSJ, this alternative offers the following benefits:

1. All hard real-time Java components reside in a different virtual name space than all soft real-time (or traditional) Java components. This means there is never any confusion between which class libraries are designed to support the hard real-time execution model (without heap) and which libraries are intended to make use of garbage collection.
2. The protocol prohibits direct sharing of objects between the hard real-time and soft real-time domains. Thus, there is no way for an errant or malicious traditional Java component to synchronize at the wrong time or in the wrong way on a shared hard real-time Java object, thereby introducing priority inversion that could possibly compromise the timeliness constraints of all hard real-time components.
3. Traditional Java components can block, waiting for notification from hard real-time components. The mechanism uses a disciplined form of Java's familiar *wait*/*notify* services. In order to acquire a lock on a hard real-time Java object, a traditional Java thread must transform itself into a hard real-time thread. As such, the thread is under the full control of the hard real-time Java environment's priority ceiling and priority inheritance implementations. While a traditional Java thread holds a lock on a hard real-time Java object, it is only allowed to execute code that was written by the hard real-time Java developer.
4. Entirely under the control of the hard real-time programmer, selected hard real-time Java threads can block, waiting for notification from traditional Java components. The mechanism also uses a disciplined form of Java's familiar *wait*/*notify* services.
5. Besides offering improved encapsulation and separation of concerns, this mechanism avoids wasting high-priority CPU cycles in busy-wait loops and avoids the scheduling latency introduced by waiting for a timer tick in an I/O polling loop.

In comparison with the alternative of writing low-level code in C and combining this with Java by way of the JNI protocol, this alternative mechanism offers:

1. Improved robustness and scalability, because object-oriented protocols are used on both sides of the interface, and byte-code verification ensures that proper protocols are followed on each side of the interface.
2. Improved performance, because the byte-code verification performed on both the soft real-time and hard real-time Java components enables optimizations, including in-lining, that are not possible with JNI.

A hard real-time Java method that can be invoked from a traditional Java virtual machine is known as a traditional-Java method. The hard real-time Java byte-code verifier imposes certain restrictions on traditional-Java methods. These restrictions include

- A traditional-Java method cannot be declared to return its result in the caller's scope.
- A traditional-Java method is not allowed to perform object allocations in immortal memory.
- A traditional-Java method must treat all of its incoming arguments as potentially residing in scoped memory.

9. Experience

The byte-code verifier for this paper's hard real-time Java system was released commercially in the spring of 2007 as part of the PERC Pico product. The PERC Ultra product, which first shipped in 1997, uses real-

time garbage collection to support soft real-time applications. An integration of these two products supports hybrid applications.

The hard real-time library subset of standard edition Java comprises approximately 20,000 lines of reusable real-time Java software. To demonstrate the use of this technology, the author has implemented a proprietary *RegionMatch* Java benchmark based on algorithms provided by one of our aerospace customers (about 2,500 lines) and a single-board computer simulation involving an interrupt-handling device driver (approximately 4,000 lines). We have also ported several benchmarks to the scope-safe type system, including the FFT benchmark of the Java Grande Forum and and the CaffeineMark benchmark. These experiments have demonstrated that the type system is sufficiently expressive to support scope-safe solutions to a broad variety of real-time programming challenges. Evaluation of several synthetic benchmarks demonstrates that hard real-time Java programs run significantly faster than traditional Java. This is because they incur neither the run-time overheads of real-time garbage collection, nor the run-time overheads of scope enforcement.

The effort required to convert vanilla RTSJ and vanilla Java code to the scope-safe type system is facilitated by enforcement of the scope-safe type system. The porting process typically consists of:

1. Copying the standard-edition or RTSJ Java source code into a directory appropriate for use of the Eclipse development environment with the special byte-code verification tools;
2. Setting up a *Makefile* or Eclipse to build the desired program using the safety-critical Java tool chain; and
3. Repeatedly:
    a. compiling the code,
    b. examining the error messages produced by the byte-code verifier, and
    c. adding appropriate annotations and/or refactoring the code to address the problems identified in the error messages.

This is a very different process than the typical effort required to port traditional Java to vanilla RTSJ. In order to port traditional Java code, programmers generally need to understand the program's memory allocation behavior (a non-trivial task for programs that might consist of tens if not hundreds of thousands of lines of code), and need to map all memory allocation operations to appropriate RTSJ scopes (or to the immortal region). With traditional RTSJ porting, programmers are rarely confident that they've done the port correctly, and generally depend on extensive testing to make sure the ported code does not result in illegal assignments, illegal fetches, or scope cycles. Even then, there often remains a lingering uncertainty that testing has not fully exercised all of the paths and data values that might possibly lead to violation of scoped-memory protocol rules.

For many applications, the process of porting to the scope-safe type system is straightforward. For example, the effort required to properly annotate the vanilla Java Grande FFT benchmark was roughly half a day for one software engineer.

In various experiments conducted on this hard real-time Java solution, we have found that Java code structured according to the hard real-time Java conventions executes within 15% of the speed, memory footprint, and determinism of optimized C code. Table 1 compares the efficiency of optimized C with commercial products supporting both soft real-time (PERC Ultra) and hard real-time (PERC Pico) execution of Java. The measured workload was patterned after a real-world defense-system application which a customer used to evaluate the comparative performance of C and Java. Though the Sun HotSpot VM is not designed to support real-time operation, we evaluated the benchmark on Sun HotSpot as well. Interestingly, the average time on Sun Hotspot is even better than optimized C, 294 ns. However, the maximum time for the HotSpot implementation was 8,089 ns and the standard deviation was 228.8 ns. After profiling the code in real time, the Sun HotSpot virtual machine transformed the code, in-lining various method calls in order to optimize the efficiency of the typical path through the code. This optimization approach serves the traditional non-real community very well, but it violates fundamental requirements, such as predictable timing behavior and traceability of code, of most real-time systems.

Also noteworthy was HotSpot's virtual memory consumption of over 250 Mbytes. HotSpot improves upon C speed by profiling the code as it runs and in-lining the implementation of certain key methods in order to optimize their execution on the fly. The reason for the very high standard deviation is because HotSpot's non-real-time garbage collector and dynamic adaptive JIT compiler introduce unpredictable delays into the execution of application code.

In some cases, we have measured that hard real-time Java code runs even faster than comparable hand-written C code. Usually, this occurs when applications allocate large numbers of small temporary objects. The hard real-time Java's scoped-based allocation is much more efficient and deterministic than typical C

implementations of *malloc* and *free*. Contrast this with compliant full implementations of the RTSJ, which generally run slower and larger than traditional Java, and introduce the complexity of run-time enforcement of scoped memory protocols. This complexity adds significantly to the costs of development, maintenance, and validation of real-time software.

Table 1. Determinism and Footprint Tradeoffs

	Gnu C (gcc -O2)	PERC Ultra	PERC Pico
Min (ns)	**280**	**519**	**314**
Max (ns)	**571**	**1,060**	**633**
Average (ns)	**360**	**639**	**392**
Std Dev (ns)	**25.52465**	**48.66415**	**29.8434**
Memory (MB)	**2.32**	**24**	**2.5**

With the ability of hard real-time Java to match C performance, the motivation to use C for the implementation of low-level, performance critical code decreases significantly. Today, the majority of successfully deployed real-time Java applications incorporate a combination of Java, C, and JNI. The C code has been required to implement portions of the system that demand higher throughput, more determinism, or smaller (more economical) memory footprint than is feasible with traditional Java. In some cases, the use of C for low-level software is motivated by a need to directly interface to hardware devices.

Based on many years of experience building embedded systems using the Java language, developers have come to view JNI as the Achilles' heel of their system architectures. All of the security that is such an intrinsic part of the Java platform is compromised at the JNI boundary. Multiple customers have identified this interface as the single most common source of errors in their developed systems. This has motivated embedded Java developers to seek the ability to replace their low-level C code with high-performance Java code.

Besides offering improved abstraction and modularity, the mechanisms described in section 8 also offer significant performance benefits. We experimented with a Java Mandelbrot fractal image application, using "low-level" code to calculate the color of each pixel. Our experiment compared the efficiency of using hard real-time Java vs. optimized C for the low-level code. We found that the differences in performance depends on the size and zoom factor for the image rendered. Certain pixels require more computation than others. Pixels that require a lot of computation favor C over Pico, because the code generated currently by the Pico compiler is not quite as efficient as hand-written C code. The dominant performance factor, however, is the cost of making a JNI invocation vs. the cost of invoking PERC Pico code from within the PERC Ultra environment. This interface is much more efficient than JNI. Consistently, the integrated full-Java solution outperformed the HotSpot/C implementation. For certain rendered images, the total rendering time was over 2.3 times faster with the full-Java solution.

10. References

[1] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, M. Turnbull, "The Real-Time Specification for Java", Addison Wesley Longman, 195 pages, Jan. 15, 2000.

[2] K. Nilsen. "Applying COTS Java Benefits to Mission-Critical Real-Time Software", *Crosstalk The Journal of Defense Software Engineering*, pp. 19-24. June 2007.

[3] K. Nilsen. "Using Java for Reusable Embedded Real-Time Component Libraries", *Crosstalk The Journal of Defense Software Engineering*, pp. 13-18. December 2004.

[4] J. Barnes, "High Integrity Software: The SPARK Approach to Safety and Security", Addison-Wesley, Apr. 25, 2003.

[5] M. Hicks, G. Morrisett, D. Grossman, T. Jim, "Experience With Safe Manual Memory-Management in Cyclone", Proceedings of the 4th International Symposium on Memory Management, pp. 73-84, 2004.

[6] A. Salcianu, C. Boyapati, W. Beebee, Jr., M. Rinard, "A Type System for Safe Region-Based Memory Management in Real-Time Java", Proceedings of the ACM Conference on Programming Language Design and Implementation, San Diego, June 2003.

[7] C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, T. Zhao, "Scoped Types and Aspects for Real-Time Java", ECOOP, pp. 124-147, Springer-Verlag, 2006.

[8] K. Arnold, J. Gosling, D. Holmes. *The Java™ Programming Language, 4th edition*. 928 pages. Prentice Hall PTR. Aug, 2005.

[9] K. Nilsen. "Guidelines for Scalable Java Development of Real-Time Systems", March 2006, available at http://research.aonix.com/jsc.