View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

Reflexes: Abstractions for Highly Responsive Systems

Jesper Honig Spring
Ecole Polytechnique Fédérale de Lausanne (EPFL)
jesper.spring@epfl.ch

Filip Pizlo
Purdue University
pizlo@purdue.edu

Rachid Guerraoui
Ecole Polytechnique Fédérale de Lausanne (EPFL)
rachid.guerraoui@epfl.ch

Jan Vitek
IBM T.J Watson
Purdue University
jv@cs.purdue.edu

Abstract

Commercial Java virtual machines are designed to maximize the performance of applications at the expense of predictability. High throughput garbage collection algorithms, for example, can introduce pauses of 100 milliseconds or more. We are interested in supporting applications with response times in the tens of microseconds and their integration with larger timing-oblivious applications in the same Java virtual machine. We propose Reflexes, a new abstraction for writing highly responsive systems in Java and investigate the virtual machine support needed to add Reflexes to a Java environment. Our implementation of Reflexes was evaluated on several programs including an audio-processing application. We were able to run a Reflex at 22.05KHz with less than 0.2% missed deadlines over 10 million observations, a result that compares favorably to an implementation written in C.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—interpreters, run-time environments; D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects; D.4.7 [*Operating Systems*]: Organization and Design—real-time systems and embedded systems.

General Terms Languages, Experimentation.

Keywords Real-time systems, Java virtual machine, Memory management, Ownership types.

1. Introduction

The nature and role of real-time processing is evolving. The complexity of embedded real-time systems has increased to the point that, today, code bases in the million lines of code are common place in avionics and shipboard computing [12]. At the same time, systems with strong timeliness requirements in finance and banking have created a market for real-time application servers [7]. What these applications have in common is that they do not look like traditional real-time systems– they are large, often in the million lines, and complex, yet they place stringent predictability requirements on the execution environment and language implementation. The majority of these systems can be written in Java, the real-time requirements are limited to a small number of subsystems. Using Java is attractive from a software engineering point of view, it brings the benefits of a high-level, type-safe, language with large library of standard components. But Java does not come with support for real-time programming, this leaves developers with the challenge of interfacing hard real-time subsystems with the main body of the application.

Our goal is to facilitate the development of such systems by integrating abstractions for highly responsive systems into mainstream Java virtual machines. As real-time applications may have very different constraints, different approaches can provide the required bounds on latency and response time. At one end of the spectrum, standard JVMs trade throughput for predictability. Memory management related jitter on a JVM can be in the hundreds of milliseconds. This can be reduced by advances in real-time garbage collection algorithms to approximately 1 millisecond [5, 22].[1] We focus on applications that are clearly beyond the reach of state-of-the-art real-time garbage collection techniques. For applications with latency requirements in the tens of microseconds, any synchronous interaction between the real-time code and the virtual machine or another non-real-time task will cause a high-frequency task to miss its deadline. Our goal is to design language constructs that permit mixed mode execution between hard real-time and non-real-time tasks. We also want to identify the minimal set of features that must be supported by the underlying virtual machine for this to be possible. Some of our inspiration comes from our experience implementing the Real-time Specification for Java (RTSJ) which required profound changes to the virtual machine [3, 6] and, to a lesser extent, the semantics of Java [8]. In our experience, most of the problems with Real-time Java came from its region-based memory model which is error-prone and requires expensive run-time checks [21, 19, 14, 17, 11]. Yet avoiding garbage collection pauses is crucial to achieve submillisecond latencies. Eventrons [23] propose a different solution to the problem. They provide a much simpler programming model than Real-time Java and one that is both efficient and provably safe. These benefits come at the expense of expressive power and require a powerful interprocedural static analysis to be carried at run-time, or more precisely at application startup time.

[1] Garbage collection pause times in the hundreds of microseconds have been reported by Henriksson [15] for C programs running with small heaps. There are no published results for a Java implementation of his algorithm.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VEE'07, June 13–15, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-630-1/07/0006. . . $5.00

Reflexes are a new programming abstraction for highly-responsive computing in Java. They provide a simple, statically type-safe programming model that makes it easy to integrate hard real-time tasks into larger Java applications. A Reflex consists of (1) a thread that is scheduled by a real-time scheduler and (2) a partition of the memory that is not touched by the garbage collector. To avoid priority inversion when interacting with non-real-time threads, the Reflex API provides a limited form of software transactional memory [18]. Finally, safety of memory operations is enforced by a type system based on our previous work for Real-time Java [2, 24]. The design of the Reflex API was driven by the following forces:

- **Safety:** Real-time programs should not experience run-time errors. With the RTSJ, any operation on a reference variable can result in an exception. Reflexes and Eventrons provide stronger memory safety guarantees. If a program is successfully verified, no memory error will ever occur.
- **Expressivity:** Static safety often comes at the expense of expressive power. The RTSJ has a rich API which supports many different real-time programming styles. In contrast, Eventrons are rather restrictive as they preclude allocation and mutation of references. Reflexes fall in between; they are strictly more expressive than Eventrons but clearly less so than RTSJ.
- **Simplicity:** Correctness is often correlated with simplicity. In that respect both Eventrons and Reflexes provide an API that is simpler and easier use than the RTSJ. The Reflex approach to static safety is arguably better as it relies on a small extension to the Java type system. An invalid Reflex results in compiler errors. Correctness for Eventrons is only ascertained after deployment and it may be harder for end-users to interpret error messages produced by a sophisticated static program analysis.
- **Efficiency:** Eventrons and Reflexes can be expected to outperform implementations of the RTSJ as they do not need run-time checks on reads/writes of references. Reflexes do not have to support priority inversion avoidance and have simpler memory region semantics. They improve on Eventrons in term of startup times and JVM footprint as Eventrons must perform data flow analysis and compilation of the bytecode at startup.

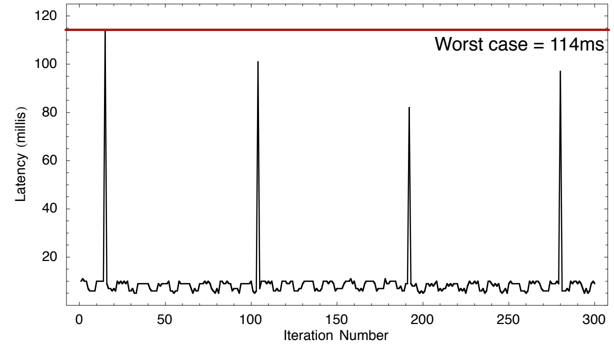
This paper validates our claims by reporting on an implementation of Reflexes. Safety is achieved by a conservative extension of the Java type system integrated in the Java 5.0 compiler. The type system is such that any valid Reflex program is also a valid Java program, but the converse is not necessarily the case. Expressivity of a programming model can only be measured on a large base of successful applications. Unfortunately, there are only a handful of RTSJ programs available in open source form and, exactly one Eventron application [23]. We have refactored the Eventron code and use it as one of our benchmarks. As another measure of expressivity we refactored the standard Java collection classes so that they can be used in Reflexes. To evaluate efficiency, we implemented Reflexes on top of the Ovm open source Real-time Java virtual machine [3]. As Ovm is a highly configurable virtual machine, we were able to start from a plain Java configuration and, with little effort add only the features needed for supporting Reflexes. Efficiency was evaluated with respect to our implementation of Real-time Java and to a C implementation of the benchmark programs.

2. Prior Work

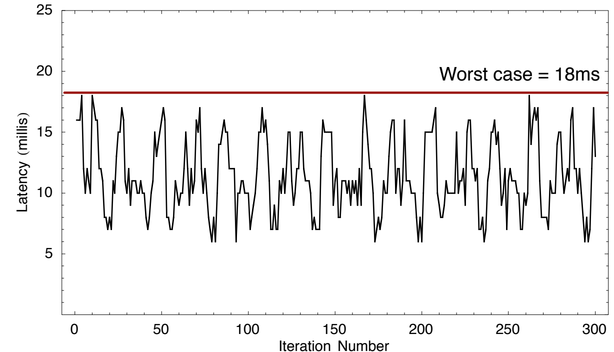
The Real-time Specification for Java (RTSJ) is designed to extend Java for real-time processing. To do so, significant changes to the Java virtual machine are needed in the areas of scheduling, synchronization and memory management. These changes affect the internals of the JVM, its performance and the semantics of Java programs (see [3] for a description of one particular implementation). The most controversial feature of the RTSJ is its memory model which allows programmers to side-step garbage collection by using nested regions, or scopes, that are de-allocated in constant time. The challenge of programming with scopes is that a simple statement such as:

```
this.str = (String) x.str;
```

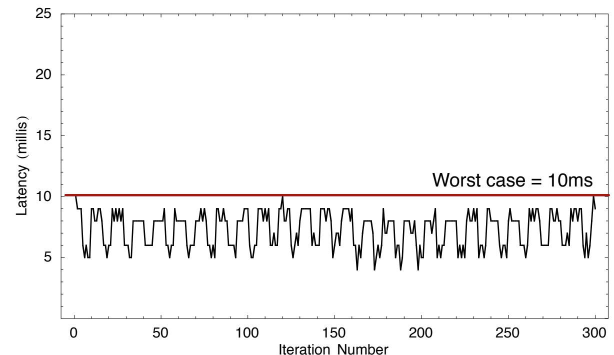
may throw two different memory exceptions: `MemoryAccessError` if the object pointed to by `x` lies in the garbage collected heap, and `IllegalAssignmentError` if the object that is the target of the assignment could outlive the string `x.str`. The virtual machine must insert non-trivial read and write barriers around all



(a) Latency of a JVM with a copying collector.



(b) Latency of a JVM with RTGC.



(c) Latency of a RTSJVM with Scoped Memory.

Figure 1. Comparing approaches to memory management in Java virtual machines. We compare a standard JVM, (a), with a real-time collector, (b), and an implementation of the RTSJ, (c), on a benchmark performing collision detection. The x-axis represent the number of iterations (input frames) and the y-axis the time to process a frame in milliseconds.

reference operations to detect such unsafe accesses. In our experience, this model is error-prone and leads to brittle code [21].

An alternative to region-based memory is to use a real-time garbage collection algorithms that enforce deterministic bounds on pause time. Recent work on time-based real-time garbage collection (RTGC) [5] has shown that it is possible to guarantee pause times around 1 millisecond in a high-performance Java virtual machine. RTGC has the advantage that the semantics of programs is not affected, but it (1) reduces throughput, and (2) requires programmers to estimate the maximum allocation rate of *all* threads in order to compute the time quantas allocated to the GC.

Eventrons [23], much like Reflexes, circumvent interference from the garbage collector by using only objects that will not be moved or modified during collection. This allows for an Eventron to be scheduled periodically and even preempt the garbage collector. To ensure safety Eventrons impose restrictions on the programming model; an Eventron must not access objects that are in the garbage collected heap as these may be in an inconsistent state, there can be no allocation within an Eventron, an Eventron is not allowed to store into a reference field as this may invalidate the result of the static analysis, and finally it is not allowed to perform blocking operations. The constraints are enforced by a startup-time data-sensitive inter-procedural analysis. So the statement:

```
this.str = (String) x.str;
```

is not valid in an Eventron as it performs reference assignment. Reflexes attempt to lift the restrictions on allocation and assignment and provide compile-time error messages instead of startup-time messages. They also reduce the footprint as the static analysis infrastructure does not need to be included in the virtual machine.

Fig. 1 illustrates some of the tradeoffs between the different approaches to memory management. The figure, from [21], shows the *latency* of processing one frame of input data in an airplane collision detection algorithm. We ran the same code with Java-GC, RTGC and RTSJ scoped memory. The mutator thread performs around 8 ms of useful work in each iteration (the exact number depends on the relative positions of the planes). Java-GC causes some iterations to take up to 114 ms causing multiple deadline misses. With a real-time collector, Fig. 1(b), the worst case observed time is 18 ms. This is interesting because, even if the bound on any individual pause is 1 ms, the mutator thread takes twice as long to complete because it is interrupted multiple times. This clearly shows that pause times are only part of the cost of RTGC, one has to account for the overhead of barriers and the frequency of pauses. With the RTSJ, there are no GC pauses as shown by Fig. 1(c). But scoped memory still has an overhead due to read/write barriers. In this example, we measured a 20% run time overhead for barriers. A deeper analysis of this benchmark and of the differences between the approaches can be found in [21].

3. Programming with Reflexes

Reflexes target small time-critical tasks that must execute free of interference from the execution environment. Examples of such tasks include data acquisition or communication with external devices. In their design, Reflexes emphasize simplicity and ease of use. The `Reflex` class is a subclass of `Thread` that is treated specially by the virtual machine, it runs at a priority higher than any other Java thread, including the garbage collector, and is scheduled in a priority preemptive manner. Thus, when a Reflex starts running, it will run to completion without interruption – with the possible exception of preemption by a higher priority Reflex.

Reflexes avoid interference from the garbage collector through a combination of runtime support and static constraints. All data allocated within a Reflex, and the Reflex object itself, is placed in a part of memory that is not subject to garbage collection. Furthermore, the Reflex type system enforces constraints that prevent Reflexes from referring to heap allocated data. The Reflex thread can thus preempt the garbage collector without having to worry about observing objects in an inconsistent state. Furthermore, since a Reflex does not allocate on the heap it cannot trigger a garbage collection on its own.

The Reflex programming model allows for a bimodal distribution of object lifetimes. An object allocated within a Reflex can be either *stable*, in which case the lifetime of the object is equal to that of the Reflex, or it can be *transient* in which case the virtual machine will reclaim it before the next invocation of the Reflex. Specifying whether an object is stable or transient is done at the class level, the programmer annotates classes which are to be allocated in stable memory with `@stable` meta-data tag. By default, data allocated by a Reflex thread is transient, only objects of classes marked `@stable` will persist between invocations. Stable objects must be managed carefully by the programmer as the size of the stable area is fixed and the area is not garbage collected.

```
class HighFreqReader extends Reflex {
   private State state;
   private Channel channel;
   int idx;
   public void initialize(int sz) {
      state = new State(sz); }
   void periodic() {
      Vector data = new Vector();
      for(int i=0;i<state.size();i++)
         data.put(channel.read());
      state.update(data);
      state.current = state.channels[idx++ % 4];
   }
}

@stable class State {
   Channel current;
   Channel[] channels = new Channel[4];
   void update(Vector v) {...}
}

@stable class Channel {}
```

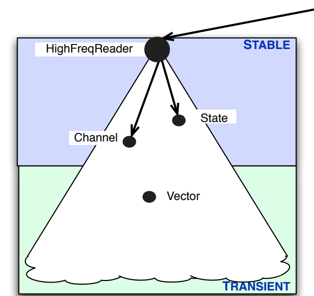


Figure 2. `HighFreqReader` is a Reflex which periodically processes data. Transient data (e.g. the `Vector` instance) is reclaimed automatically at the end of each period. Dangling pointers to transient objects from stable storage are prevented statically by the type system. Stable classes are explicitly annotated. The type system enforces the following three properties on the object graph: (a) objects allocated by a Reflex do not point into the heap, (b) object allocated outside of a Reflex can only point to the Reflex object and not its internal state, and (c) stable objects do not point to transient ones through instance fields. The type system and Reflex runtime support only affects objects allocated within a Reflex.

Fig. 2 illustrates the use of Reflexes. The programmer defines class HighFreqReader as a subclass of Reflex. It is a normal Java class with instance variables and methods. It must implement the periodic() method which is invoked every time the Reflex is scheduled. In this example, the body of the method allocates a Vector, reads data from a channel and finally updates the internal state of the Reflex.

```
Vector data = new Vector();
...
state.current = state.channels[idx++ % 4];
```

When periodic() is invoked, the VM automatically switches the allocation area to that of the Reflex's transient memory. Thus, the Vector instance, an instance of a standard Java class, is transient and will be reclaimed as soon as the method returns. An instance of this Reflex is created and started as follows:

```
Reflex rflx = Reflex.make(HighFreqReader.class,
              stableSize, transientSize, 45000);
rflx.initialize(100);
rflx.start();
```

The call to make() takes a class name, sizes for the stable and transient areas, and a period[2]. The virtual machine will invoke the Reflex's periodic() method every 45 μs from the time the Reflex is started. The initialize() method can be invoked from plain Java code and will cause allocation of an instance of class State in the Reflex's stable area. The in-memory graphical representation for the Reflex in Fig. 2 shows that the Reflex object can be referred to from Java code, but all objects allocated by the Reflex are under the Reflex's ownership – no external reference to them is allowed and, conversely, they cannot refer to any object allocated in the heap.

4. The Reflex Type System

We present a type system that ensures memory safety by preventing dangling pointers and preventing a Reflex from observing objects in an inconsistent state as they are, for example, when being copied by a garbage collector. The presentation is informal, we refer interested readers to [24] where we formalized an ownership type system for a variant of the FeatherweightJava object calculus and proved its type soundness. The Reflex type system is an extension of that work with the novel notion of implicit ownership.

The Reflex type system is an ownership type system. As in other ownership type systems [13, 9] there is a notion of a dominator that encapsulates access to a subgraph of objects – in our case every Reflex instance encapsulates all objects allocated within its stable and transient memory regions. The type system we present here ensures that: (1) references to objects owned by a Reflex are never accessed from outside the Reflex; (2) a Reflex will not refer to objects subject to garbage collection; and finally, (3) a Reflex prevents dangling pointer errors caused by references to transient objects from stable ones.

Reflex ownership is *implicit* because, unlike e.g. [13, 9], no ownership parameters are needed on class declarations. This reduces the annotation burden and increases opportunities for reuse. Previous systems with explicit ownership have the significant drawback that *no* existing code can be reused without refactoring, for instance the Vector class would have to be refactored to something like Vector<owner> in order to record the owner of each vector.

We implemented a type checker for Reflexes by extending the Java 5.0 compiler. No changes to the Java syntax were required and all error messages are returned by the javac compiler in a format programmers can easily understand.

4.1 Partially Closed-world Assumption

A key requirement for type-checking a Reflex is that all classes that will be used within it must be verified. We refer to this as a *partially* closed-world assumption, as we place no constraints on code that is outside of a Reflex. The type checker thus takes as input a collection of class files. Classes used within a Reflex fall in one of two categories: those whose instances are allocated in *stable* and those in *transient* memory. We use $\mathcal{W}$ to denote the union of these categories for a given Reflex. The first task of the checker is to ensure that no class outside of $\mathcal{W}$ can be instantiated within the Reflex. This can be done in a straightforward fashion by inspecting the methods of the classes in $\mathcal{W}$ and checking that new objects are instances of classes in $\mathcal{W}$.

$\mathcal{R}0$: *Any class annotated with the meta-data tag* @stable *is in* $\mathcal{W}$. *Any subclass of* Reflex *is in* $\mathcal{W}$.

$\mathcal{R}1$: *Given a class instance creation expression* new C(...) *occurring in class* C'*, if* C' *or a subclass of* C' *is in* $\mathcal{W}$ *then* C *must be in* $\mathcal{W}$. □

The type checker will validate all classes in $\mathcal{W}$ and their parent classes. Classes that are not in $\mathcal{W}$ need not be type checked, the rules are purely local to the Reflex. *Thus our type system does not prevent dynamic loading – it only ensures that no dynamically loaded class will be used within a Reflex.* The type checker will also ensure that static methods invoked within a Reflex belong to classes in $\mathcal{W}$. Taken together rule $\mathcal{R}1$ and $\mathcal{R}2$ ensure that no object of a class that is not in $\mathcal{W}$ will ever be created while evaluating code in $\mathcal{W}$.

$\mathcal{R}2$: *Any invocation of a* static *method* C.m() *occurring in class* C'*, if* C' *or a subclass of* C' *is in* $\mathcal{W}$ *then* C *must be in* $\mathcal{W}$. □

Reflective invocation and reflective instantiation are restricted. The Reflex API lets the virtual machine know if the current thread is within a Reflex and grants access to $\mathcal{W}$, so that it is possible to check that the method being invoked or the object being created are allowed in that particular Reflex. Native methods are currently allowed on a case by case basis as they must be validated by hand.

4.2 Implicit Ownership

A Reflex instance acts as an owner for all objects allocated within it. The type system ensures that code running outside of a Reflex cannot acquire a reference to an object allocated within it. The simplest way to achieve this without having to check all client code is to leverage the visibility rules of Java to enforce confinement.

$\mathcal{R}3$: *A reference type cannot appear in the signature of a non-*private *member (method or field) of any subclass of* Reflex. □

The rule ensures that the methods and fields visible to clients of a Reflex will not leak references across the Reflex boundary. Likewise, as we saw previously, the rules $\mathcal{R}1$ and $\mathcal{R}2$ ensure that the Reflex cannot acquire a reference to an object outside of $\mathcal{W}$. Dangling pointers to transient objects are prevented by segregating stable references from transients ones. This is done at the class granularity. If a class is declared stable, then it can only refer to other stable classes.

$\mathcal{R}4$: *The type* T *of an instance field declaration in a* @stable *class or a parent of a* @stable *class is legal if* T *is a primitive type, a*

[2] The full Reflex creation API is richer as it allows to set parameters such as start time and priority, but these are not essential for this discussion.

`@stable` *class or an array of* `T'` *where* `T'` *is* `@stable`. □

Since the type system tracks classes, it is critical to prevent instances of transient classes from masquerading as stable objects. This is achieved by mandating that descendents of stable classes are stable.

$\mathcal{R}5$: *Any subclass of a* `@stable` *must be annotated* `@stable`. □

4.3 Static Reference Isolation

The rules introduced so far are only part of what is needed. Enforcing encapsulation also requires that communication through `static` variables be controlled. Without any limitations, static variables can be used for unrestricted sharing of references across Reflex boundaries and can thus provide opportunities for all kinds of programming errors. A drastic solution is simply to prevent code in $\mathcal{W}$ from reading or writing static reference variables.

$\mathcal{R}6$: *Let* `C` *be a class in* $\mathcal{W}$ *or a parent of a class in* $\mathcal{W}$. *A field access expression occurring in* `C` *is illegal if the field is a* `static` *reference type. An assignment statement occurring in* `C` *is illegal if the left-hand side of the assignment is a* `static` *field of reference type.* □

This is clearly sufficient as the only static variables that a Reflex is allowed to use are ones of primitive types. But it is a bit too restrictive in practice. We can loosen the type system a little bit with *reference-immutable* classes. Recall that what we are trying to prevent is that a Reflex attempts to read from or write to an object that is being moved or whose fields are being updated by the garbage collector. With some help from the VM, it is possible to make sure that designated immutable objects are never moved – and thus can be used safely from within a Reflex. We say that a class is reference-immutable if it is transitively immutable in its reference fields. The following two rules are use to determine if a class is reference-immutable.[3]

$\mathcal{R}7$: *A class is reference-immutable if all non-primitive instance fields in the class and parent classes are declared* `final` *and are of reference-immutable types.* □

$\mathcal{R}8$: *Any class in* $\mathcal{W}$ *can read a* `static` *reference variable of reference-immutable type.* □

Rule $\mathcal{R}6$ is thus relaxed by allowing Reflexes to access static reference-immutable variables.

4.4 Encapsulating Arrays

Arrays and especially arrays of primitive types are needed in virtually all applications of Reflexes (communicating with devices, signal processing, etc). For performance reasons, it is crucial to be able to exchange arrays without copying. The rules as stated above allow use of arrays without restrictions in transient memory, but not in stable memory. We extend the type system to support the common case of primitive arrays.

$\mathcal{R}9$: *A field of a uni-dimensional array type is allowed in a stable class if it is declared* `private final` *and is assigned to a freshly allocated array in all constructors.* □

$\mathcal{R}10$: *An instance field of array type in a stable class can be used only in array read and array write expressions.* □

[3] Interestingly, the restriction enforced by data flow analysis in Eventrons are similar. The difference is that an Eventron can manipulate objects with some mutable state, as long as it does not read/write the mutuable parts.

This rules codifies the idiom of Fig. 3. This allows the allocation and use of arrays provided that they are wrapped in a stable class. For multi-dimensional arrays, each dimension must be wrapped independently.

```
@stable class ImmutableIntArray {
   private final int[] data;
   ImmutableIntArray(int sz) { data = new int[sz]; }
   void set(int p,int v) { data[p]=v; }
   int get(int p) {  return data[p]; }
}
```

Figure 3. Encapsulated Arrays. Primitive arrays can be allocated in a stable class as long as they are encapsulated, e.g. `data` holds a reference to the encapsulated array.

In order to achieve zero-copy communication, the type system supports the notion of borrowed array references. A borrowed reference is a reference that can be used to read or write but that cannot be stored. Thus, a primitive array can be passed in as argument to a method without fear that the reference will be stored within the method.

$\mathcal{R}11$: *A primitive array parameter annotated with* `@borrow` *can be used in array read and array write expressions and as argument of a method if the corresponding parameter is annotated as* `@borrow`. □

Borrowing could be made less restrictive, for example to allow user-defined types, at the price of additional rules. In our application domains, the restrictive version appears sufficient for most needs.

4.5 Checking Java Collections

The reader may rightly wonder how restrictive the type system is, and in particular, how it compares with the data flow analysis approach adopted by [23]. In an Eventron all data must be reference immutable. It is thus generally not possible to reuse standard Java classes. Does the same hold for Reflexes?

As an experiment, we tried to type-check the collection classes, such as `Vector` and `HashMap` in the `java.util` package for Java 1.4 (the GNUClasspath open source implementation). When inner classes are counted, there are 126 classes and about 22,000 lines of code. We decided to treat the collection classes as transient and ran the type checker. The first set of errors were due to the use of classes such as `String`, `StringBuffer` and `Random`. We declared them as intrinsics – special types that are treated as transient by the type checker but not validated. After declaring these classes safe, there were still over 200 type errors due to the use of static reference variables. The collection classes have a total of 66 static fields, out of which only 10 fields are of reference type. They hold various singletons used to represent empty collections, empty slots and empty keys. The key observation is that these statics are not dangerous as they are never modified and they are never reclaimed. This suggested extending the type system with the notion of reference-immutable types. Adding rules $\mathcal{R}7$ and $\mathcal{R}8$ eliminated all but a handful of errors.

Rooting out the last errors would require some refactoring of the collection classes. The problem arises from the fact that some of the singletons, while they are in practice immutable, have non-final fields. One can take care of those errors by refactoring some of the collection classes to introduce immutable singleton classes. There is only one class that we did not attempt to include in the experiment, `WeakHashMap`, as it drags in extra libraries and has

no use within a Reflex since transient objects are not garbage collected.

In conclusion, we found that the majority of Java collection classes can be used without any changes within a Reflex and a small number of classes require small modifications to be usable.

4.6 Design Choices

The type system is affected by several fundamental design choices. First, by choosing class granularity for distinguishing between stable and transient objects, we relinquish using the same class in both contexts. The alternative would be to have per-object annotations. So one could write code like `@stable HashMap = @stable new HashMap()`. But unfortunately that is not enough, as the code within `HashMap` may need to allocate, it is necessary to treat the annotation as a type parameter, e.g. `new HashMap<@stable>()`. While object granularity allows a greater degree of reuse, it is more heavyweight and requires retrofitting all library classes with generic parameters. The added effort and complexity does not seem warranted. One distinct benefit of avoiding annotations on stable (or transient for that matter) classes is that the same classes can be used within different Reflexes as either stable or transient, and outside of any Reflex.

Another choice is that transient is the default case. Unlike for stable classes, transients have no restrictions on the types of their fields. This choice reflects the hypothesis that stable code is the smaller part of a Reflex and that it is less likely that we need to reuse legacy libraries in stable classes (part of the reason is that the allocation behavior of many library classes is not appropriate for an environment where objects are not reclaimed).

Borrowing has been discussed before [10]. We considered allowing borrowing of objects but this adds two issues: a reference retrieved from a borrowed object must be treated as borrowed (borrowing is 'sticky') and calling a method of a borrowed object is allowed only if the method does not leak a reference to the receiver. Once again we chose to go with the simpler solution: Borrowing only allows the exchange of arrays. Since most examples of Reflexes that we have been able to identify only deal with primitive values and arrays this is a sensible choice. Another possible approach for communication between plain Java and Reflexes would be to pass objects by deep copy. This is safe but the cost of copying is too high for many applications.

5. Virtual Machine Support

Reflexes have been implemented on top of the Ovm Java virtual machine. We leveraged real-time support in the VM to implement some of the key features of the API. The virtual machine configuration described here uses an optimizing ahead-of-time compiler to achieve performance competitive to commercial virtual machines [21]. The VM was designed for resource constrained uniprocessor embedded devices and has been successfully deployed on a ScanEagle Unmanned Aerial Vehicle in collaboration with the Boeing Company [3].

Scheduling. Scheduling is implemented in the VM. Priority-preemptive scheduling is available for real-time threads. The complete priority range is from 1-42, where the subrange 12-39 are real-time priorities used by Reflexes and the remaining are used for Java threads. The VM's mostly-copying collector is run in a Java thread. In a priority preemptive scheduled system, real-time threads must yield explicitly (i.e. the `periodic()` method must return).

The abstract `Reflex` class is implemented as a thread with real-time priorities. The thread is started as a result of an invocation of `start()`. This causes the thread to be scheduled at a start time that may either be the current time, or a user defined future time. The ability to set the start time is essential to prevent the Reflex from being released at times that are out of phase with the rest of the system.

Memory hierarchy. For each Reflex instance, the implementation allocates a fixed size continuous memory region for the Reflex's stable area and another region for its transient area. Furthermore, a buffer is set aside for the transactional log. The size of each of the above is set programmatically in the Reflex API. The Reflex object, its thread and all other implementation specific data structures are allocated in the Reflex's stable area. These regions have the key property that they are not garbage collected. The collector does trace the stable area because it contains implementation specific objects that may have references into the heap.

Each thread in the VM has a default allocation area. This area is the heap for all Java threads and the respective transient area for all Reflex threads. The VM exposes low-level functionality for setting allocation areas through the class `ReflexSupport`. The static method `setCurrentArea()` allows the Reflex implementation to change the allocation area for the current thread. Regions are reference counted, each call to `setCurrentArea()` increases the count of active threads by one. The static method `reclaimArea()` decreases the counter by one for that area, if the counter is zero all objects in the area are reclaimed. The method `reclaimAreaAndWait()` is a blocking version of the above. Essentially, the allocation pointer is reset to the start of the area. Thus the behavior of a Reflex's `start()` method can be expressed abstractly as:

```
start() {
   while(this.waitForNextPeriod()) {
      ReflexSupport.setCurrentArea(transient);
      this.periodic();
      ReflexSupport.reclaimAreaAndWait(transient);
   }
}
```

We use bytecode rewriting to bracket all invocations of Reflex methods from Java code with `setCurrentArea/reclaimArea` pairs to ensure that when a Reflex method is called from Java code the allocation area is set to the transient region.

The VM exposes another method, `setAllocKind(reflex, class)` for identifying stable classes. Whenever an instance of `class` is created by a thread running in `reflex` (which is obtained from the thread's allocation context), the stable region is used instead to allocate the object. `setAllocKind()` is invoked at the creation of each Reflex for each class annotated `@stable`. The allocation of arrays encapsulated within the constructor of a stable class is rewritten to add code that checks if the thread is running within a Reflex and, if yes, allocates the array in stable memory.

The Ovm virtual machine also supports allocation policies for meta-data. In particular, we rely on a policy for lock inflation that ensures that a monitor is always allocated in the same area as the object with which it is associated, regardless of the current allocation area.

Pinning. The garbage collector supports pinning for objects. Pinned objects are guaranteed not to move during a garbage collection. Thus they can safely be accessed from a Reflex. The allocation policy for classes and static initializers ensures that all objects allocated at initialization are pinned (in the current implementation the class objects and static fields are actually allocated in a non-GCed immortal region). There is one other case where pinning is required. Borrowed variables may be heap-allocated objects, so these are pinned before invoking a Reflex method from plain Java and unpinned when the method returns.

Calls from Java. Invoking a method of an instance of a Reflex from plain Java requires changing the allocation context from the

heap to that of the Reflex. Furthermore, if one or more of the arguments to the call are annotated `@borrow`, the VM will try to acquire the monitor for each of them. This has two goals, it forces lock-inflation before entering the Reflex and ensures that the Reflex will not have to block for a Java thread if it tries to synchronize on a borrowed object.[4]

Reclaiming Reflexes. A Reflex and its memory can be reclaimed if the Reflex thread is not active and the Reflex object is unreachable. The current implementation does not garbage collect Reflexes. This is not a problem as Reflexes are not created dynamically in any of the applications that we have considered so far.

Exceptions. When an exception is thrown within a Reflex, the object is created with normal Java semantics. By default the exception object and its stack trace are created in transient memory. If the exception propagates out of the `periodic()` method, the stack trace is printed and the thread is terminated. If an exception escapes from a call to a Reflex method from plain Java, the exception object is translated into a `ReflexException` that is allocated in the heap and rethrown.

Synchronization. Synchronization and `wait/notify` protocols can be used within a Reflex. We do not guarantee priority inheritance semantics for locks, thus if users want to avoid priority inversion they are encouraged to use `@atomic`. The implementation of `wait/notify` does not require allocation and thus can be safely used in a Reflex. If a Reflex notifies a lower priority Java thread, the priority preemptive scheduler ensures that the Reflex thread continues until its explicit yield point before scheduling the Java thread.

Finalization. The VM does not invoke finalizers of objects allocated in a Reflex. While this may seem harsh, observe that since objects in a Reflex have no reference to operating system's resources and have a very predictable lifetime, the value of finalization is quite low.

5.1 Priority Inversion and Atomicity

Communication between Java thread and Reflexes must be managed carefully as it may cause blocking in the Reflex. The only way for Reflexes and Java to interact is through public methods of the Reflex class. A Java thread can call any one of these methods, in which case it executes within the Reflex. The semantics is that any allocation performed within that invocation is done in transient and allocated objects will be cleared before the next invocation of the `periodic()` method. There can thus be multiple threads active within a Reflex and they may need to regulate access to resources.

There are two issues here. First, if a plain Java thread grabs a lock on an object and the Reflex thread tries to acquire the same lock, the Reflex will have to wait for the Java thread to release it. Second, as long as the Java thread is active within the Reflex, the Reflex's transient memory cannot be reclaimed. This, again, may cause the Reflex to block. Priority inversion may occur in either scenario if the Java thread is preempted by another thread (that thread may be further preempted or, worse, trigger a garbage collection).

We avoid some of the most egregious cases of priority inversion by increasing the priority of any Java thread entering a Reflex to a priority that is higher than all normal Java threads but lower than all Reflexes for the duration of the call. This has the effect of ensuring that the Java thread will not be preempted by a heap-allocating thread. (Note that since all threads that are at the same priority level or higher are running within Reflexes, none of them will trigger a GC).

[4] A preferable solution would be to ignore any use of the `synchronized` within a Reflex.

It remains that we need to provide synchronization support that reduces blocking between Java and Reflexes. We supplement Java monitors with a simple transactional facility inspired by the preemptible atomic regions of [18]. We allow users to annotate methods as `@atomic`. The semantics is simple: the method will execute atomically, unless another thread in the same Reflex is released. If another thread starts executing within the Reflex, all changes performed within the atomic method are undone and the method will be automatically re-executed. The implementation follows [18] with one difference, there is one transactional log per Reflex instead of a single global log. For each write within an atomic the VM records the original value and address of field in the log. An abort boils down to replaying the log in reverse order. No conflict detection is needed as aborts are performed eagerly at context switches. Enters and commits are constant time, aborts are proportional in the number of writes performed in the atomic section.

5.1.1 Implementing Channels

We illustrate atomicity and some of the features of the Reflex type system with a simplified version of the AudioPlayer benchmark used in the evaluation section. Fig. 4 shows two classes `AudioPlayer` and `Channel`, the latter declared stable. The player has a `periodic()` method that sends data to an audio device. It also has a public `write()` method which allows a Java synthesizer thread to send it audio data.

The AudioPlayer also has an instance of class `Channel` in its stable state. The type system forces us to declare the variable private. For the outside to be able to write into the channel, it is necessary to add a public `write()` method to the AudioPlayer. This method takes an array of shorts and calls the channel's write method. The channel copies the data into its buffer. There are two noteworthy things about this implementation. The array of notes is passed by reference from the synthesizer to the player. We use a `@borrow` annotation to ensure that the player does not retain a reference to the object. Secondly the channel's `write()` method is

```
class AudioPlayer extends Reflex {
   private Channel chan;
   public AudioPlayer(...) {
      super(...);
      chan = new Channel();
   }
   public void periodic() { ... }
   public void write(@borrow short[] samples) {
      chan.write(samples);
   }
}

@stable class Channel {
   final private short[] data;
   Channel(int size) {
      data = new short[size];
   }
   @atomic public int write(@borrow short[] b) {
      for (int i=0;i<b.length;i++)
         data[i]=b[i];
   }
}
```

Figure 4. An AudioPlayer Reflex containing a Channel. Both AudioPlayer and Channel are stable. An external synthesizer thread communicates with the AudioPlayer by calling its `write()` method. The channel's `write()` is declared `@atomic`.

declared `@atomic`. This means that if the Reflex's thread is released while the plain Java thread is writing to the channel, the changes performed by the writer will be undone. Once the periodic method returns, the plain Java thread is automatically restarted.

6. Evaluation

We conducted a number of experiments to evaluate the extent to which Reflexes achieve the response times needed by highly-responsive applications. All experiments were performed on an AMD Athlon 64 X2 Dual Core processor 4400+ with 2GB of physical memory running Linux. The kernel version used was 2.6.17 extended with high resolution timer (HRT) patches [1] configured with a tick period of 1 μs. We used an Ovm build with support for POSIX high resolution timers, and configured it with an interrupt rate of 1 μs. In addition, we disabled the run-time checks of violations of memory region integrity (read/write barriers) and configured it with a heap size of 512MB. The time-critical Reflex tasks were scheduled to run at a 45 μs period (equivalent to frequency of 22.05KHz, a standard audio frequency) and were executed over 10 million periods.

We evaluated Reflexes based on two metrics of predictability: precision of inter-arrival time, i.e., time between two successive executions of a Reflex, and number of missed deadlines when running Reflexes in isolation and in a mixed environment. It is important to properly characterize a deadline miss. A missed deadline occurs for the i'th firing of a Reflex with a period p if the actual completion time, α_i, comes after its expected completion time, ϵ_i, where $\epsilon_i = p(\lceil(\alpha_{i-1}/p)\rceil + 1)$. When counting missed deadlines, we will be conservative and consider both *real* deadline misses as well as *absolute* deadline misses; the difference being that for a given period p, a single absolute deadline miss might cover $i * p$ real deadlines, where $i > 1$. (So we will in this case count i deadline misses.)

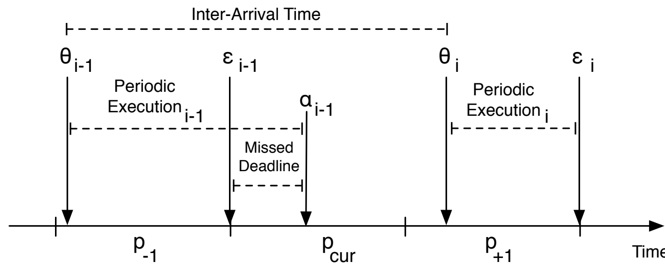


Figure 5. Timeline showing how a missed deadline can cause an inter-arrival time between two consecutive periodic executions to be larger than twice the period.

Note also that an inter-arrival time larger than twice the period p (but strictly less than three times the period) does not necessarily imply more than a single deadline miss. Fig. 5 shows that in the event of a deadline miss (when actual completion time, α_{i-1}, lies after the expected completion time, ϵ_{i-1}) of a $i-1$'th firing, the expected completion time, ϵ_i, of the subsequent i'th firing is set to be the end of the first-coming complete period, i.e., any time remaining in the current period is skipped. If the start of the subsequent periodic execution, i, is delayed (reflected in the actual start time, θ_i, lying after the period start) it can cause the inter-arrival time between the two consecutive periodic executions, $i-1$ and i, to be larger than twice the period p.

The results of the evaluation are encouraging when comparing Reflexes to equivalent implementations in native C running at 45 μs periods, both in terms of base performance as well as when running under significant workload. In both cases, Reflexes showed a comparable behavior to native C, though in the latter case with a small overhead; however, this is not unexpected when running with the extra layers of complexity that Java brings.

6.1 Virtual Machine Benchmarks

Working on a research virtual machine always raises questions about applicability of the results in the context of 'real' systems. We report on the raw performance of Ovm on the SpecJVM98 benchmark suite and compare it with Hotspot 1.5 and GCJ 4.0.2. We evaluate two Ovm configurations: the plain Java configuration and the RTSJ configuration which includes scoped memory access checks. Fig. 6 shows that Ovm outperforms GCJ and fares surprisingly well when compared to a production virtual machine.

The figure also illustrates the costs of RTSJ read/write barriers (up to 50%). SpecJVM is by no means representative of a real-time application, but it gives an estimate of the cost of memory access checks.

6.2 Base Performance

To evaluate the base performance of Reflexes, we implemented a single no-operation Reflex (and a similar C program), scheduled it for a 45 μs period, and let it execute over 10 million periods.

As depicted in Fig. 7 nearly all interesting observations centered around the 45 μs period, though the Reflexes appear to be slightly less timely than the C variant, because the spread in inter-arrival time is wider. Also note the observations clustered around 200-250 μs for both variants, which we attribute to perturbations in the underlying operating system. Similar observations for an equivalent base performance benchmark are reported in [23].

Fig. 8 depicts missed deadlines for both Reflexes and their C variant. More precisely, with Reflexes 99.996% of the periods are completed in time with no absolute deadline miss (99.993% in the case of real deadline misses). On the other hand, the equivalent for C is 99.997% (real: 99.993%). Interestingly, Fig. 8 indicates some pattern in deadline misses around 100-200 μs for both the Reflex and C, though for C there seems to more consistency in that pattern. Also, it appears that both versions experience an equivalent amount of deadline misses, but the Reflexes have more variation in the actual sizes of the misses than the C variant. In both cases, given the similar patterns in the missed deadlines lead us to believe that these must be caused by the underlying operating system.

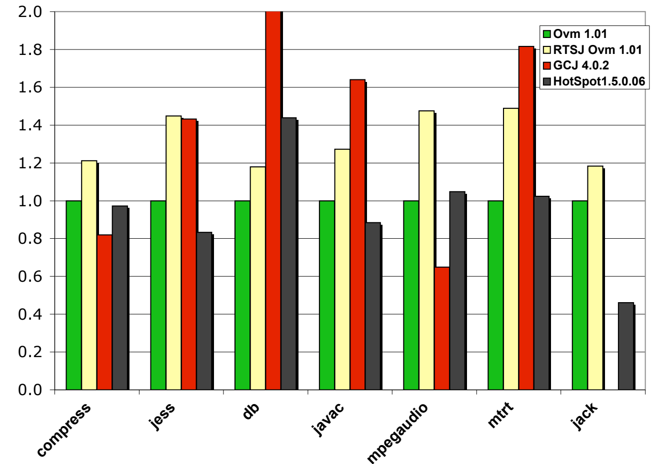


Figure 6. Comparing Java VMs on the SPECJVM98 benchmarks. The x-axis shows the individual benchmark tests and the y-axis the relative performance compared to Ovm (set to 1.0).

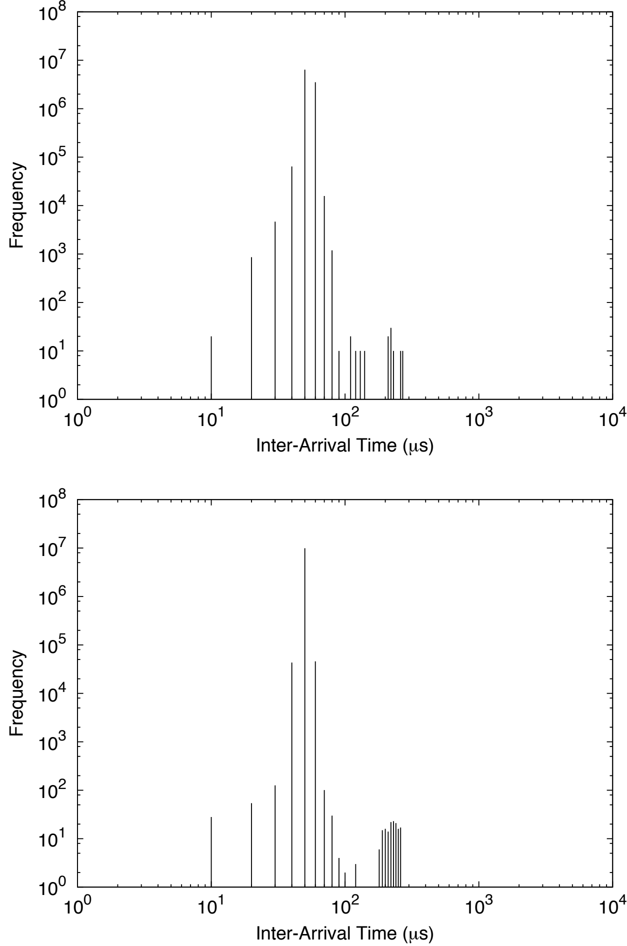


Figure 7. Histograms of inter-arrival time for Reflexes (top) and C (bottom) with a no-operation task scheduled for 45 μs periods. The x-axis shows the logarithm of the inter-arrival time in μs and the y-axis shows the logarithm of its frequency.

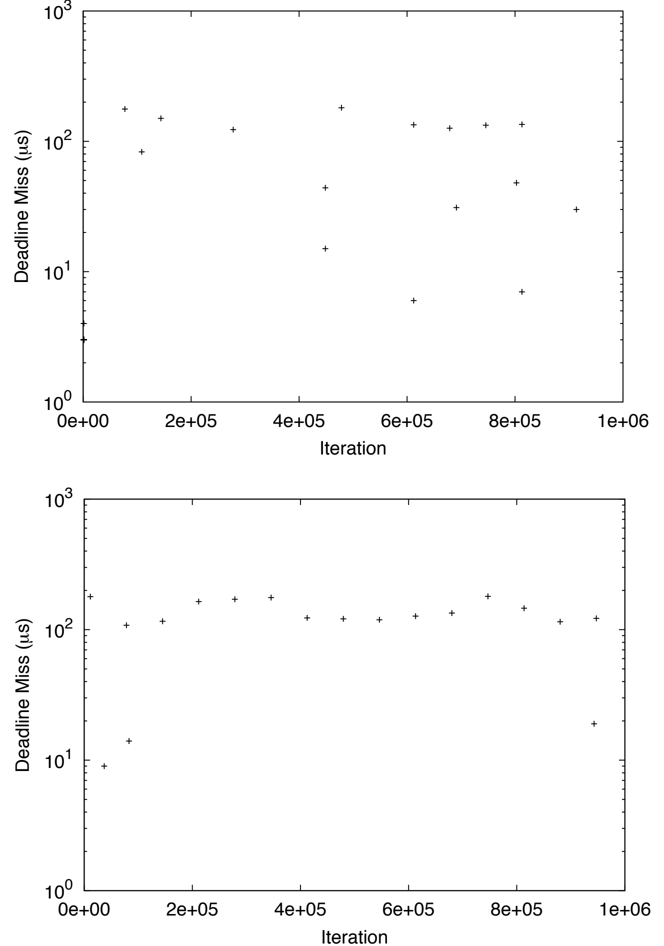


Figure 8. Missed deadlines over time for respectively Reflexes (top) and C (bottom) with a no-operation task scheduled for 45 μs periods. The x-axis shows the executions (only 1 million iterations shown) of the periodic task and the y-axis shows the logarithm of the size of the deadline misses.

6.3 Mixed Environment Performance

Having compared the base performance of Reflexes and corresponding C code, we next measured the performance of Reflexes under a workload stemming from a mixed environment with a plain Java thread and a (time-critical) Reflex task executed concurrently. We considered here a music synthesizer application, developed at IBM for Eventrons [23], which we modified to make use of Reflexes, including the lock-free transactional channel.[5] In short, the scenario involves a plain Java thread that generates music samples, and writes these to a channel. These are then periodically read by an audio player Reflex scheduled with 45 μs periods and which then writes the samples individually to the sound device for playing. For the sake of comparison, we implemented a corresponding C variant of the music synthesizer.

Fig. 9 depicts the inter-arrival time of the highly responsive audio player thread for both the Reflex and C variants. As already noted in Fig. 7, outlier clusters around the 200-300 μs range can also be seen in Fig. 9 for both Reflexes and its C variant. However, in Fig. 9 these outliers appear to have been enhanced, which we attribute to the effects of buffering congestion in the sound device to which the time-critical task is writing (twice per execution)[6].

The outlier clusters seen in Fig. 9 also seem to have a direct impact on the missed deadlines as seen in Fig. 10. Specifically, for Reflexes 99.869% (real: 99.698%) of them complete in time and do not cause deadline misses. For the C variant, this is the case in 99.949% (real: 99.799%) of the time. Of particular interest in Fig. 10 is to see how the perturbation causes regular deadline misses around 180 μs. We consider these anomalies to most likely be caused by buffering on the sound device or to stem from other interactions with the underlying operating system, and we have learned (through private conversations) from the Eventrons project that they experienced equivalent behavior when running at these frequencies. With Reflexes, however, there seems to be further frequent deadline misses in the ranges 2-3 μs, 5-6 μs and around 110-120 μs. These we attribute to the jitter in timeliness as described earlier and depicted in Fig. 7 and which also appears to cause similar missed deadlines as seen in Fig. 8.

[5] Eventrons have been released under the name *Expedited Real-Time Threads* and is available from `www.alphaworks.ibm.com`.

[6] First the upper 8 most significant bits of the short value are written to the sound device followed by the 8 least significant.

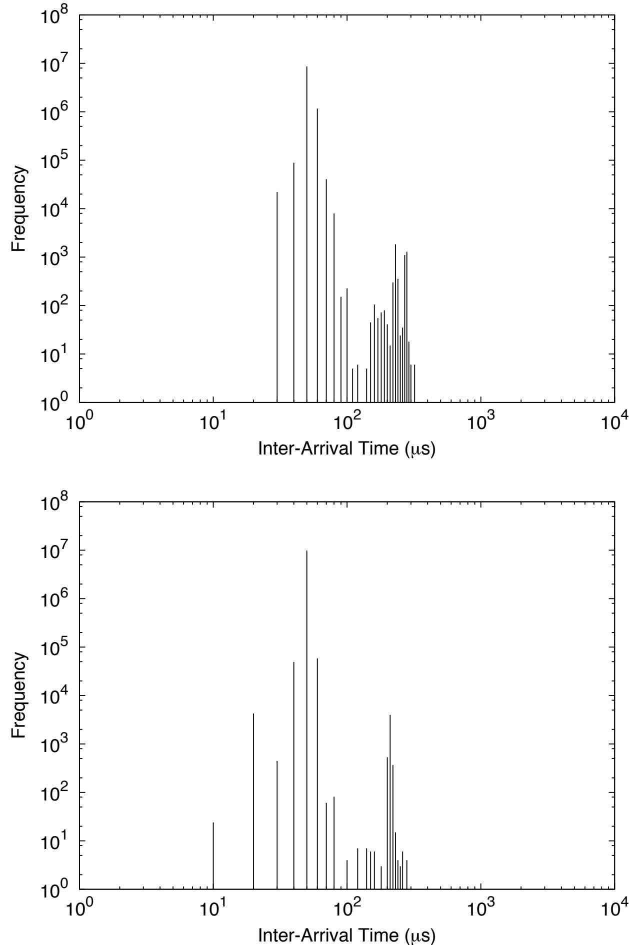


Figure 9. Histograms of inter-arrival time for respectively Reflexes (top) and C (bottom) with an audio player task scheduled for 45 μs periods. The x-axis shows the inter-arrival time in μs and the y-axis shows the logarithm of its frequency.

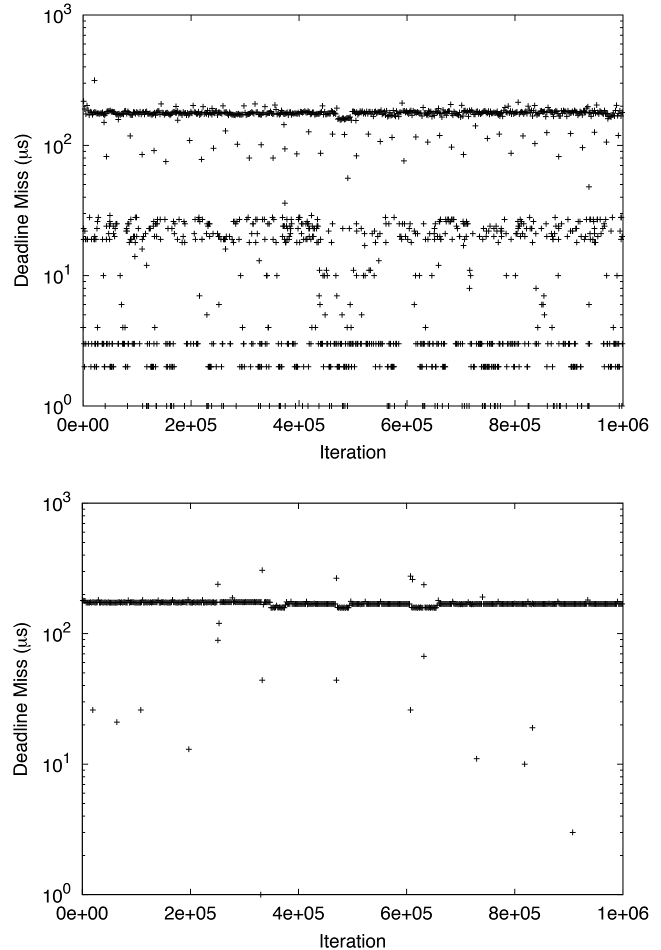


Figure 10. Missed deadlines over time for respectively Reflexes (top) and C (bottom) with an audio processing task scheduled for 45 μs periods. The x-axis shows the periodic executions (only 1 million iterations shown) of the time-critical task and the y-axis shows the logarithm of the size of the deadline misses.

7. Conclusion

We presented a new programming abstraction, *Reflexes*, for programming highly-responsive systems in Java. Reflexes combine control and data to provide high-frequency and predictable real-time tasks. They avoid garbage collection pauses with a region-based memory model that is both simple and statically type safe. A Reflex can thus be scheduled periodically by a priority preemptive scheduler running at higher priority than any other thread in a Java virtual machine including the garbage collection thread. While Reflexes are protected from interference they are not completed isolated, they can communicate with standard Java threads through a transactional memory abstractions that prevents priority inversion by preemption and roll-back of non-real-time tasks.

Our experimental evaluation demonstrated that the predictability of a Java virtual machine extended with support for Reflexes is competitive with that of a comparable C program. In a mixed environment, where Java threads and Reflexes had to interact, the number of deadline misses was 0.1% higher for Reflexes than for a comparable C program, a number which is not entirely unexpected as Java has some inherent overheads.

The Reflex approach can be extended in several ways. The ownership type system used to enforce memory safety could be extended with generics and thus allow more programs to be verified. Another exciting direction is would be to replace region-based memory with Reflex-local real-time garbage collection. One way to approach the problem would be to integrate the hierarchical real-time garbage collection technique of [20] to collect each reflex independently. Exotasks [4] go one step further, they aim to provide *time-portability*. Like reflexes, they are real-time components executing without interference from plain Java threads (and their garbage collector), instead they have a per-exotask real-time garbage collector. Furthermore Exotasks strive to stamp out non-determinism by adopting the notion of logical execution time [16].

Acknowledgments. We thank Jason Baker and the member of the Purdue Ovm team for their help fixing a critical bug in the compiler. We thank Joshua Auerbach, David Bacon for many helpful discussions and detailed feedback on this text. We thank Greg Bollella and Manuel Fähndrich for their comments. Finally, our work greatly benefited from the availability of the Eventron benchmark application, we are deeply grateful to Spoonhower, Auerbach, Bacon, Cheng, and Grove for making their source code available. This work is supported in part by NSF grants 501 1398-1086 and 501 1398-1600 as well as the EU 6th Framework Programme, project IST-002057.

References

[1] http://www.tglx.de/projects/hrtimers/2.6.17/.

[2] Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped Types and Aspects for Real-Time Java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2006)*, pages 124–147, Nantes, France, July 2006. Springer.

[3] Austin Armbuster, Jason Baker, Antonio Cunei, David Holmes, Chapman Flack, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems (TECS)*, 2006.

[4] Joshua Auerbach, , David F. Bacon, Daniel T. Iercan, Christoph M. Kirsch, V.T. Rajan, Harald Röck, and Rainer Trummer. Java takes flight: Time-portable real-time programming with exotasks. In *in the Proceeedings of ACM SIGPLAN/SIGBED 2007 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2007.

[5] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 285–298, January 2003.

[6] Jason Baker, Antonio Cunei, Chapman Flack, Filip Pizlo, Marek Prochazka, Jan Vitek, Austin Armbuster, Edward Pla, and David Holmes. A real-time Java virtual machine for avionics. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*. IEEE Computer Society, 2006.

[7] BEA. Weblogic real time. `www.bea.com.`, 2006.

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

[9] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 2002.

[10] John Tang Boyland. Alias killing: Unique variables without destructive reads. In *IWAOOS*, 1999.

[11] Benjamin M. Brosgol, Scott Robbins, and Ricardo J. Hassan II. Asynchronous transfer of control in the Real-Time Specification for Java. In *Proceedings of the Fifth International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)*, 2002.

[12] Jeff Child. DD(X) program leads navy's voyage toward cost-efficient computing. *COTS Journal*, June 2006.

[13] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 48–64. ACM, October 1998.

[14] Angelo Corsaro and Ron K. Cytron. Efficient memory reference checks for real-time Java. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, 2003.

[15] Roger Henriksson. *Scheduling Garbage Colection in Embedded Systems*. PhD thesis, Lund University, July 1998.

[16] Tom Henzinger, Christoph M. Kirsch, and T. Z. Horowitz. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, January 2003.

[17] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-Java: A high integrity profile for real-time Java. In *Joint ACM Java Grande/ISCOPE Conference*, November 2002.

[18] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time Java. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, December 2005.

[19] Filip Pizlo, Jason Fox, David Holmes, and Jan Vitek. 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, May 2004.

[20] Filip Pizlo, Athony L. Hosking, and Jan Vitek. Hiearchical real-time garbage collection. In *Proceeedings of ACM SIGPLAN/SIGBED 2007 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2007.

[21] Filip Pizlo and Jan Vitek. An empirical evalutation of memory management alternatives for Real-time Java. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, December 2006.

[22] Fridtjof Siebert. Real-time garbage collection in multi-threaded systems on a single processor. In *20th IEEE Real-Time Systems Symposium (RTSS'99)*, Phoenix, Arizona, 1999.

[23] Daniel Spoonhower, Joshua Auerbach, David F. Bacon, Perry Cheng, and David Grove. Eventrons: a safe programming construct for high-frequency hard real-time applications. In *Proceedings of the conference on Programming language design and implementation (PLDI)*, pages 283–294, 2006.

[24] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS04)*, Lisbon, Portugal, December 2004.