Eventrons: A Safe Programming Construct for High-Frequency Hard Real-Time Applications

Daniel Spoonhower
Carnegie Mellon University
spoons@cs.cmu.edu

Joshua Auerbach
IBM Research
josh@us.ibm.com

David F. Bacon
IBM Research
bacon@us.ibm.com

Perry Cheng
IBM Research
perryche@us.ibm.com

David Grove
IBM Research
groved@us.ibm.com

Abstract

While real-time garbage collection has achieved worst-case latencies on the order of a millisecond, this technology is approaching its practical limits. For tasks requiring extremely low latency, and especially periodic tasks with frequencies above 1 KHz, Java programmers must currently resort to the NoHeapRealtimeThread construct of the Real-Time Specification for Java. This technique requires expensive run-time checks, can result in unpredictable low-level exceptions, and inhibits communication with the rest of the garbage-collected application. We present *Eventrons*, a programming construct that can arbitrarily preempt the garbage collector, yet guarantees safety *and* allows its data to be visible to the garbage-collected heap. Eventrons are a strict subset of Java, and require no run-time memory access checks. Safety is enforced using a *data-sensitive* analysis and simple run-time support with extremely low overhead. We have implemented Eventrons in IBM's J9 Java virtual machine, and present experimental results in which we ran Eventrons at frequencies up to 22 KHz (a 45 μs period). Across 10 million periods, 99.997% of the executions ran within 10 μs of their deadline, compared to 99.999% of the executions of the equivalent program written in C.

Categories and Subject Descriptors C.3 [*Special-Purpose and Application-Based Systems*]: Real-time and embedded systems; D.3.2 [*Programming Languages*]: Java; D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection); D.4.7 [*Operating Systems*]: Organization and Design—Real-time systems and embedded systems

General Terms Experimentation, Languages, Measurement, Performance

Keywords Scheduling, Allocation, Real-time

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.

PLDI'06 June 10–16, 2006, Ottawa, Ontario, Canada
Copyright © 2006 ACM 1-59593-320-4/06/0006. . . $5.00.

1. Introduction

The complexity of real-time systems is increasing rapidly as the isolated real-time controllers of the past give way to complex systems in which many coordinated systems interact to create an integrated multi-level real-time system, such as a car or a ship. In the face of this huge increase in complexity, traditional real-time methodologies based on low-level coding styles and fixed data structures are no longer tenable.

Java's software engineering benefits have made it a compelling choice for modular application development: memory safety, a standardized model of concurrency, and a host of common libraries make it possible to develop large and complex applications in a piecewise fashion. With the development of Metronome real-time garbage collection technology [2] and the Real-Time Specification for Java (RTSJ) standard [4], Java has become a viable platform for the creation of such complex real-time systems. However, there are many problems remaining before a dynamic, garbage collected language like Java can provide a complete real-time solution down to the lowest-level, highest-frequency applications.

Real-time garbage collection, as exemplified by the Metronome system [2], is currently able to achieve worst-case latencies on the order of a millisecond. This latency bound may scale down by another factor of two to four as a result of improved implementation techniques. However, latency is limited by the need for atomic processing of certain data structures, and by the overhead of context switching, especially since the collector tends to evict much of the application data from the cache.

As a result, real-time Java programmers with response time requirements below a millisecond are currently forced to resort to the RTSJ's NoHeapRealtimeThread (NHRT) construct. As its rather verbose moniker implies, code executed by an NHRT is isolated from the garbage-collected heap and may therefore be run concurrently with the collector or interrupt it arbitrarily. However, NHRTs may only create and access objects in the manually-managed Immortal and Scoped memory areas.

Unfortunately, this requires fundamental changes to the semantics of the Java programming language. Scopes essentially provide region-based memory management with neither explicit region qualifiers [12] nor implicit region inference [22]. As a result, the region restrictions of a method are not documented in its interface and may vary dynamically based on its inputs. This greatly lowers the level of abstraction of the language, requires expensive run-time checks, and makes it difficult to reason about the behavior of an object without understanding its implementation.

Most perniciously, NHRTs require dynamic checks on both reads and writes of reference fields to guarantee that region constraints are not violated. These dynamic checks will throw exceptions when they fail, while reading or writing reference fields of an object in the heap will never throw an exception. This represents a deep change to the language semantics and a significant obstacle to code re-use.

In this work, we introduce *Eventrons*, a programming construct for tasks that can arbitrarily preempt the garbage collector and yet are comprised entirely of standard Java code without extra run-time memory access checks. Eventrons are simple to design, develop, and understand. They can also be terminated in the event of cost over-run without causing deadlocks or data structure corruption.

Eventrons are both more and less restrictive than NHRTs and Scopes. Allocation within Eventrons is not permitted; neither is modification of pointers. These restrictions are enforced at *validation time* using a simple bytecode analysis. Once an Eventron is validated, it will not throw any exceptions except those resulting from normal Java semantics such as array bounds checking.

Eventrons are less restrictive than Scopes because the Eventron's data structures can be accessed by normal Java threads running in the garbage collected heap. As a result, communication of data between high-frequency Eventron tasks, and lower frequency, garbage-collected tasks is greatly simplified.

Validation occurs at run-time, examining both the code and the data of the Eventron, but must be performed before an Eventron may be scheduled. By choosing this intermediate point between fully static and fully dynamic checking, we are able to perform a simple analysis whose restrictions are easy for programmers to understand, and yet obtain a fairly precise call graph since it is based on a known set of previously instantiated objects. The separation of the program into an "initialization" and a "mission" phase is also natural to most real-time applications.

Eventrons are well suited to tasks like sensor processing, in which sensor data must be sampled at a very high frequency, buffered, and then processed *en masse* by a lower-frequency task. They are also well-suited to simple high-frequency control loops that read a limited amount of data, perform some simple processing (such as FFTs) and then produce actuator control values. The ability to share data between high- and low-frequency tasks allows the use of more complex, dynamically allocated, garbage-collected data structures in the low-frequency tasks that make higher-level planning decisions requiring more complex processing.

The combination of Eventrons for high-frequency tasks and real-time garbage collection for medium- to low-frequency tasks provides an integrated programming methodology that enables the use of heap-allocated data structures in even the most demanding real-time applications, and yet does not change the Java programming model in any fundamental way.

2. Example: Music Generation

We begin with an example to illustrate the use of Eventrons to implement an integrated real-time application. The example combines both low- and high-frequency tasks to generate and play back music without relying on the operating system or underlying hardware to buffer the output. A low-frequency task performs the computation necessary to compose a musical score and generate the raw audio data, while a high-frequency task sends the raw audio data to the unbuffered output device, one sample at a time. To provide seamless audio playback at the standard frequency of 22.05 KHz, the high-frequency task must be run every 45 μs.

Implementing a Low-Frequency Task Figure 1 defines the low-frequency task. This task uses a music composition library (not shown here) to generate a musical score one note at a time. Each note is then converted to a set of samples and written to a shared channel. The low-frequency task (including any libraries it uses) may rely on a garbage collector to automatically manage memory; the code used to implement this task is only constrained at the point where it interacts with the high-frequency task.

```
class LowFrequencyTask extends Thread {
  EventronReadChannelOfShort channel;
  ...
  public void run() {
    short[] samples = new short[MAX_DURATION];
    while (composing) {
      Note note = composer.nextNote();
      int length = note.toSamples(samples);
      channel.write(samples, length);
    }
    channel.close();
  }
}
```

Figure 1. The low-frequency task. This task generates the next note of a musical score, converts the note to a array of samples, and then writes the samples to a shared channel.

```
class HighFrequencyTask implements Runnable {
  final EventronReadChannelOfShort channel;
  final RuntimeException errorStop =
    new RuntimeException("Write error occured");
  final RuntimeException normalStop =
    new RuntimeException("Channel closed");
  int missedSamples = 0;
  ...
  private static native boolean playSample(short sample);

  public void run() {
    short sample = channel.read();
    if (sample == channel.CHANNEL_EMPTY) {
      missedSamples++;
      sample = Composer.SILENCE_SAMPLE;
    } else if (sample == channel.CHANNEL_CLOSED) {
      throw normalStop;
    }
    if (!playSample(sample)) {
      throw errorStop;
    }
  }
}
```

Figure 2. The high-frequency task. This tasks reads a single 16-bit sample from the shared channel and plays it using an interface to the raw audio device.

In this example, that interaction is managed by an EventronReadChannelOfShort. This general purpose data structure provides a form of impedance matching between the low- and high-frequency tasks and can be configured to hide the latency and jitter of the low-frequency task. While the code shown above allocates an array from which the raw audio samples are copied into the channel queue, the channel queue also provides a mechanism to avoid this copy. Section 9 discusses the mechanisms by which low- and high-frequency tasks may communicate in more detail.

Implementing a High-Frequency Task Figure 2 defines the high-frequency task that is implemented using the Eventron abstraction. During each iteration, the task reads a single 16-bit sample from the shared channel and writes that sample out to the audio device.

We enforce a number of constraints on Eventron code during initialization using an on-line analysis. For example, note that ref-

```
1  final long AUDIO_FREQ_HZ = 22050L;
2  final long AUDIO_PERIOD_NS = 1000000000L / AUDIO_FREQ_HZ;
3  final long GC_LATENCY_NS = GarbageCollector.maximumLatency();
4  final int BUFFER_SIZE = (int) (GC_LATENCY_NS/AUDIO_PERIOD_NS);
5  final int BUFFERS = 3;
6  final int PRELOAD_SAMPLES = BUFFER_SIZE * (BUFFERS−1);
7
8  public void playMusic(Composer composer) {
9    FileOutputStream soundDevice = composer.openSoundDevice();
10
11   EventronReadChannelOfShort channel =
12       new EventronReadChannelOfShort(BUFFER_SIZE, BUFFERS);
13   channel.write(Composer.SILENCE_SAMPLE, PRELOAD_SAMPLES);
14
15   Runnable runnable = new HighFrequencyTask(channel, soundDevice);
16   Eventron player = Eventron.validate(runnable);
17
18   Thread synthesizer = new LowFrequencyTask(channel);
19   synthesizer.start();
20
21   player.schedulePeriodic(AUDIO_PERIOD_NS);
22   ...
23   player.unschedule();
24   Throwable status = player.getExceptionState();
25 }
```

Figure 3. Task Integration. The two tasks are instantiated, initialized with a shared channel queue, and the high frequency task is validated before being passed to the scheduler.

erence fields are declared final on lines 2-6. We have found that the use of final is a natural and familiar way to ensure that the Eventron data structure remains fixed over time. Our analysis is described in Section 5.

The high-frequency task uses the non-blocking interface of the EventronChannel to read individual samples. This operation returns either a sample or a value indicating that the channel is empty or closed. Alternatively, programmers can avoid reserving special indicator values and instead use exceptions. However, all exceptions must be preallocated, as also illustrated in the example.

This task also uses a native method (lines 9 and 19) to communicate with the audio playback device. Many applications of Eventrons may require native methods to communicate with hardware. The Eventron run-time takes steps to ensure the relative safety of JNI code that is invoked by an Eventron, as discussed in Section 5.3.2.

Notice that the high-frequency task contains no looping construct. Instead, the scheduler will invoke this task periodically as indicated during initialization. The Eventron will continue to be invoked until it is explicitly removed from the scheduling queue (by some other thread), or when an exception is thrown from the Eventron run() method.

Integrating Tasks Figure 3 shows the integration of the low- and high-frequency tasks. On line 11, the programmer instantiates a shared data structure which is the sole mechanism for interaction between the low- and high-frequency tasks. The low-frequency task is scheduled on lines 18 and 19 by instantiating an ordinary Java thread.

Before the high-frequency task can be executed, it must first be validated (line 16) to ensure that it can safely preempt the collector. While validation may fail and cause a checked exception to be thrown in the main() method, no memory access checks are performed while executing the Eventron. Thus errors will occur during the initialization phase of the application instead of during the execution of the Eventron itself. This is a natural time to detect any problems in the configuration of the Eventron, as the applica-

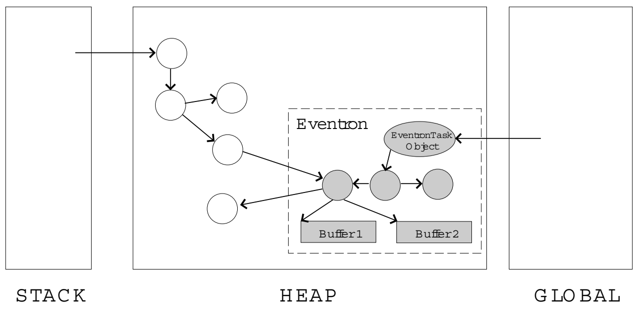

Figure 4. Interaction of Eventrons with the heap. Objects associated with an Eventron (gray objects) reside in the garbage-collected heap but are pinned for the life of the Eventron. They may be referenced by other heap objects, and will be subject to garbage collection once the Eventron terminates.

tion must also ensure that any required hardware is properly configured and that sufficient resources are available to run the Eventron (though we have elided these checks in our example).

Validation also causes the run-time system to pin down any data structures accessible by the Eventron: these objects will not be collected or even moved to another location in the heap while the Eventron is active. However, once the Eventron is disabled, these objects can be moved and, if they are otherwise unreachable, reclaimed.

3. The Life and Times of an Eventron

An Eventron is defined by a set of Java methods that will be executed as a high-frequency task, the data structures manipulated by those methods, and a thread that will execute them. We guarantee that Eventrons can safely preempt the virtual machine, and in particular the garbage collector, at any point by eliminating any synchronization between Eventrons and the collector. First, we ensure that Eventrons cannot change the reachability of any object managed by the collector. While this condition might be enforced by additional run-time checks, we instead impose constraints on the code and data structures defining an Eventron that can be verified before the Eventron is executed. These constraints ensure that the set of objects accessible to an Eventron is fixed for the duration of its execution, while allowing objects used by an Eventron to be accessed by ordinary Java threads. We enforce these constraints using a *run-time static* analysis: we perform an analysis of the Eventron bytecode while the program is executing, but before the Eventron itself is invoked. This allows an efficient and precise analysis.

Secondly, we eliminate the need for synchronization between the collector and an Eventron by precluding any collector operations that, if preempted by an Eventron, might expose the heap in an inconsistent state. Specifically, we *pin* every object accessible to an Eventron. Thus the collector will never be interrupted while moving one of the Eventron's objects.

The lifetime of an Eventron is split into five phases. First, an Eventron is *initialized* in the same way an ordinary Java structure is constructed. During the second phase, *validation*, we verify that the Eventron code can safely preempt the collector and simultaneously discover the set of objects that must be pinned. The third phase is *instantiation*, in which the Eventron's data structure is pinned and an executable thread is created. The fourth phase is *execution*, during which the Eventron is invoked one or more times, either at specific times, with a predetermined period, or in response to certain external events. Finally, an Eventron may be *destroyed*, at which point the objects in its data structure are unpinned and

Bytecode	Illegal Unless...
new, newarray	
anewarray, multianewarray	
monitorenter, monitorexit	
invokevirtual	non-synchronized method
invokestatic	non-synchronized method
invokespecial	non-synchronized method
invokeinterface	non-synchronized method
putfield, putstatic	Primitive type
getfield, getstatic	Primitive type or final
aaload	From FinalArray
aastore	

(a) Bytecodes that are illegal in Eventron code

Class	Illegal Methods
java.lang.Object	clone, wait, notify, notifyAll
java.lang.Class	newInstance
java.lang.reflect.Constructor	newInstance
java.lang.reflect.Field	get, set
java.lang.reflect.Array	newInstance, set, get
java.lang.reflect.Method	invoke

(b) Methods that are illegal in Eventron code

Figure 5. Operations that are illegal in Eventron code. The first table includes illegal bytecodes and the second table lists illegal method invocations.

eventually garbage collected, unless they are reachable from the heap or other Eventrons.

4. Initialization

To create an Eventron, the programmer constructs an instance of a class that implements the standard Java Runnable interface. This object forms the root of the Eventron data structure, and its run() method forms the entrypoint for the execution of the Eventron.

The programmer also builds the data structures necessary to support communication and synchronization between the Eventron and low-frequency threads. Typically this involves the use of one or more *channels* as demonstrated in the example in Section 2 and described in detail in Section 9.

5. Validation

During the validation phase, we ensure that an Eventron may be safely executed without synchronizing with the collector. Validation takes place while the application is running but before the Eventron is invoked. This allows the validation process to take advantage of information that is present at run-time and to detect potentially misbehaving code while the task is being set up rather than during the execution of the Eventron itself.

Validation must ensure three features of the Eventron code: first, that it does not allocate any memory, second, that it does not perform any blocking operations, and third, that the extent of the heap that it accesses remains fixed.

These restrictions are enforced by creating a *data-specific* call graph for the particular Eventron being validated, and performing a set of checks on the bytecodes of the resulting methods. This process can be viewed as a kind of partial evaluation.

5.1 Enforcement of Restrictions

We will begin by assuming that we have created the call graph, and describe the enforcement of the restrictions, and then describe how the call graph is constructed.

Operations that are illegal within an Eventron are summarized in Figure 5. When any of these operations are encountered, an EventronValidationException is thrown reporting the source code location of the offending instruction.

As shown in Figure 5(a), all bytecodes that perform allocation are prohibited, as well as bytecodes that perform synchronization either explicitly (via monitor bytecodes) or implicitly (via synchronized methods). As further shown in Figure 5(b), reflective access to such operations is also prohibited, as is the use of the wait and notify methods of Object (which would necessarily generate exceptions since the Eventron thread is never able to enter a monitor).

The last group of restrictions in Figure 5(a) ensures that Eventrons neither read nor update mutable heap references. Writes to object fields, static variables, and array elements are only permitted for primitive types. Reads *are* permitted for non-primitive types, but only if the field is declared final.

There are two complications with respect to final fields: first, Java does not provide any mechanism to specify the immutability of array elements. To this end we have provided a special class, ImmutableArray<T>, which stores an array of immutable references to objects of class T.

An ImmutableArray is initialized by passing it an array of references, which is copied into a private instance array of the ImmutableArray object. The only methods provided are get(index) and length(). Since the finality of the array elements within an ImmutableArray is enforced programmatically, the validator allows their use within an Eventron.

The reflective analogues get and set of the proscribed pointer-accessing bytecodes are also prohibited. Note that unlike the bytecode operations, the reflective operations are more restricted, since we do not know at validation time which fields or methods are being accessed. In particular, neither reflective access to final pointer fields nor reflective method invocation is permitted within an Eventron.

The second complication with respect to finality has to do with exposure of incompletely constructed objects. We solve this problem using a simple run-time assisted mechanism described in Section 5.3.1.

In practice, we found that rejecting all methods that perform allocation inhibits the reuse of some existing library methods inside Eventrons. This is often due to the common Java idiom of allocating exception objects ("throw new FooException"). In many cases, the programmer knows that the offending code will never be executed but the validator conservatively assumes otherwise. An optional feature of the validator allows programmers to exempt certain methods from the normal check for allocating bytecodes. To keep this feature from compromising safety, the run-time allocator throws an immediate OutOfMemoryError whenever an allocation occurs on an Eventron thread.

5.2 Data-Sensitive Call Graph Analysis

The validator's analysis explores the set of reachable methods and the set of reachable objects simultaneously, letting each set inform the other. Rather than build a complete call graph, the validator analyzes only those methods that could be invoked given the set of objects reachable by the Eventron. In the course of its analysis, the validator will conservatively determine sets of objects O, field signatures F, methods M, and virtual method signatures V reachable to the Eventron. These are the structures which may be accessed by the Eventron during its execution.

Initially, O contains only the object implementing the Runnable interface, and the only method in M is its run() method. For each method added to the call graph, the validator considers each bytecode instruction of that method in turn. Note that each method is analyzed at most one time, and therefore the validator will terminate in time linear in the size of the program and the size of the heap.

- getfield – The signature of target field f is added to the validation set F. In addition, the validator considers every object in O that defines a field matching that signature. For every such object, the validator uses reflection to determine the referent of its field f; this referent is then added to O.

As objects are added to O, each one must be inspected for fields that are already found in F, and the referents of these fields must be added to O. Therefore, in the course of processing a single getfield instruction, only one field signature will be added to F, but many objects may be added to O.

- getstatic – The validator immediately adds the referent of the static field to O. Any fields of the referent that appear in F are considered as above.
- invokestatic, invokespecial – For method invocations whose targets can be statically determined, the validator adds the target method to M.
- invokeinterface, invokevirtual – For virtual method invocations, the signature of the target method is added to V. The validator then considers each object in O implementing that signature. For each such object, its implementation is added to M and subsequently validated.

Just as the fields of objects added to O must be compared to the signatures in F, the methods of those objects must be compared to the signatures in V. That is, for each object added to O, and for each method m of that object, if m matches a signature in V, then m must be added to M.

Finally, we enforce the restrictions described above as we process each method. If a prohibited bytecode instruction or method invocation appears in an analyzed method body, the analysis terminates immediately, throwing an exception that indicates where the offending code can be found.

If the validation succeeds, the objects in O are those that may be accessed by the Eventron and therefore must be pinned.

Recall that any reference field (instance or static) accessed by an Eventron must be declared final. This restriction is used not only to ensure that synchronization with the collector can be safely omitted, it is also necessary for the soundness of the above analysis. Without this restriction, the validator cannot safely assume which objects may occupy a given field or which implementations of a given method signature might be invoked.

Because we require only those fields that might be referenced by an Eventron to be declared final, some reference fields in the Eventron data structure may not be final. As a result, pinned objects may hold (mutable) references to unpinned objects, as shown in Figure 4. This is safe since our analysis has shown that these fields will not be accessed by the Eventron.

5.2.1 Data-Sensitive Analysis versus Other Analyses

Note that our analysis is neither flow-sensitive nor context-sensitive. It is, however, *data-sensitive*: the call graph is being constructed for the invocation of the run() method of a particular concrete Eventron instance. In general, flow- and context-sensitivity can increase the precision of call graph analysis [13]. However, even fairly aggressive algorithms still make some conservative approximations, especially about the contents of instance fields, which can induce imprecision in the call graph.

JIT compilers often compensate for the imprecision of static call graph construction algorithms by exploiting profile data, online class hierarchy analysis, and other optimistic and speculative optimizations [1]. This often works well in practice, but if the assumption turns out to be invalid, the code must be recompiled (an operation that does not mesh well with real-time execution).

In principle, a data-sensitive analysis may be more or less precise than any of the above techniques, depending on the particular class hierarchies, call graphs, and execution orders. However, we believe that a data-sensitive analysis will often yield the most precise result in practice.

Regardless of the precision, however, there is a more compelling reason to use data-sensitive analysis: it is simple enough that it can be easily described to a programmer. More complex analyses are easily perturbed by small changes in the code, and a seemingly trivial change can cause two variables to become possibly aliased. Our analysis avoids this form of speculation and relies on two forms of information specified directly by the user: the object graph and the use of final fields. In contrast to other analyses, our data-sensitive analysis uses mechanisms that are familiar to Java programmers and yields an induced call graph that is quite stable in the face of changes to the program.

5.2.2 Implementation

We implemented our validation phase using a combination of standard Java reflection (to explore the graph of reachable objects) and specialized VM-specific code to consult the bytecodes and constant pool of already loaded and validated classes (validation must run against the in-memory classes to prevent security attacks that change the class files). It is straightforward to implement the same validation algorithm using a class file inspection package, with or without a static analysis framework (for example, an earlier prototype used the Domo program analysis library [11]). However, such a validator would only be safe to use for development-time exploration of Eventron validity. A run-time validator must ensure that it is working with the actual bytecodes that have been loaded into the running VM.

5.3 Preventing Violations of Finality

To ensure complete safety of Eventron execution, some additional checks are required for code that executes outside of the Eventron threads.

5.3.1 Exposure of Incompletely Constructed Objects

Unfortunately, simply using final does not guarantee that heap structures will be immutable. A constructor may expose its this pointer before its fields are completely initialized, for instance by storing this into a static field. If a partially constructed object is present in an Eventron data structure, the validator may see a final field as null even though it will be subsequently initialized to a non-null value. In this case, the validator will have based its analysis on incomplete information, and the safety of the Eventron could be compromised.

Attempts to prevent such operations via static analyses tend to be grossly over-conservative because the targets of virtual calls are not available.

Our solution is to add a "fully constructed" flag to the header of every object. The interpreter sets this flag as a constructor returns (the return of any constructor for a given class is sufficient to guarantee that final fields of that class and its superclasses are initialized). We provide a run-time interface allowing the flag to be queried, and the validator throws an exception whenever it encounters an object for which this flag is not set. This results in a simple programming model and an efficient implementation.

The specification to the programmer is simple: all Eventron objects must be completely constructed.

5.3.2 Reflection and JNI

It is also possible to undermine the restrictions of final using reflection. The method invocation restrictions on Eventron code (Figure 5(b)) already rule out any reflective access to pointers in the Eventron data structure from the Eventron itself. However, other threads have full access to reflection, and in Java 1.5, reflection even allows the modification of final fields (!) after a call to setAccessible(). According to the specification, this functionality is only intended to be used during deserialization and other uses may have "unpredictable effects." However, any violation of finality would render Eventrons unsafe, which we do not consider to be an acceptable "unpredictable effect," so this must be prevented.

Our preferred approach to this problem is to modify the Field.set() implementation of java.lang.reflect to check whether the field being set is a final field of an object pinned by an Eventron. In that case, a run-time exception is thrown. Similarly, Array.set() can be modified to prevent reflective modification of pointers inside the arrays that implement the ImmutableArray class.

Restrictions on reflective field access should not cause run-time exceptions in code that uses the interface as intended, since Eventron data structures are constructed before validation, and so by definition are not being de-serialized.

Similar to reflection, JNI code can circumvent Eventron restrictions and violate validation assumptions. For example, native code can execute Java methods that were not part of the validation-time call graph, read and write reference fields that the validator did not know were to be read or written, enter monitors, or allocate objects. Despite this, many Eventrons must use native code to interface with hardware devices. One could argue that JNI code is inherently unsafe since it is capable of directly modifying any word in memory (indeed, no JNI access control restrictions are enforced in the Java programming language [18] as a deliberate design decision). However, we judged that JNI code that was completely safe to execute in other contexts ought not to produce arbitrary results simply because it was invoked by an Eventron.

We currently catch native code violations with inexpensive dynamic checks in those JNI helper functions that should not be called by Eventrons. Each helper function that would be unsafe to execute on an Eventron thread will immediately throw an exception if executed on an Eventron thread. This allows problems to be caught early in the testing phase (albeit not at validation time).

6. Instantiation

Once an Eventron has been validated, it must be instantiated. This requires either creating an executable thread that will invoke the Eventron repeatedly at high frequency, or associating the Eventron with an interrupt handler (though the latter mode of instantiation has not yet been fully implemented or studied). In either case, the run-time system must also track the set of objects accessible to the Eventron and pin these objects so that they are not moved by the garbage collector.

The set of Eventron-reachable objects is created during the validation phase (to avoid redundant object traversals) and is now installed into the run-time data structures. This set is external to the Eventron itself and persists until the Eventron is destroyed.

6.1 Pinning of Eventron Objects

For Eventrons to work properly in conjunction with a collector that moves objects (whether or not the collector is incremental), the objects must be pinned before the Eventron is allowed to execute.

There are various collector architectures that move objects, among them page-based defragmenting collectors [3], generational

```
class EventronSupport
{
  static native void pinObjects(Object[] o);
  static native void unpinObjects(Object[] o);
  static native void becomeEventronThread();
  static native void becomeOrdinaryThread();
}
```

Figure 6. Interfaces to run-time support operations added to the virtual machine to support Eventrons

collectors [23], and semi-space copying collectors [7], as well as their concurrent variants. There are also various approaches to pinning objects suited to the different kinds of collector architectures. For instance, in a generational collector, objects should not be pinned in the nursery, so a nursery evacuation should be forced as part of the Eventron instantiation procedure.

In this paper we will confine ourselves to describing the design used in our actual implementation that runs together with the Metronome [2] collector, a real-time incremental collector that organizes memory into page-based segregated free lists. Defragmentation is performed on an as-needed basis and compacts objects within a size class.

To minimize changes to the Java run-time system, we chose to implement the run-time support for Eventrons in Java whenever possible. One exception is support for pinning objects. We extend the interface to the run-time system with the functions shown in Figure 6.

Invoking pinObjects prevents the objects from being moved by the collector. Invoking unpinObjects reverts this state change. Only the Eventron run-time support code has access to the pinning and unpinning methods. The native support is wrapped by Java code that aggregates pinning information from multiple Eventrons. These native methods are only called with collections of objects the first time those objects are used by an Eventron or once all Eventrons using those objects have terminated.

This interface is similar to the Create/DestroyGlobalRef functions found in the Java Native Interface [18]. While those functions ensure that a given object will not be freed, they do not guarantee that the referent will not be relocated by the collector. Though no standard support for pinned objects exists in Java, several other languages already provide similar functionality. For example, the above interface can be easily implemented using existing run-time support in either the Common Language Runtime [19] or the Haskell Foreign Function Interface [14].

Since Metronome uses a page-based memory architecture, our implementation of pinning maintains a count of the number of pin operations applied to the objects of each page. At instantiation time, pinObjects() is invoked with the set of objects that are accessible to the Eventron but that are not already pinned. Since pinning is atomic with respect to interleavings with the garbage collector, once the entire data structure has been pinned, we are guaranteed that the collector will not move any of the objects.

6.1.1 Compensating for Races with Compaction

However, this does not yet guarantee collector independence: there may be pointers among the objects that were pinned early in this stage that refer to objects that were later pinned but moved by the collector in the intervening time. Thus there may be stale pointers in the Eventron data structure. For the garbage-collected heap, this is not a problem since those pointers will be fixed during the subsequent tracing phase of the collector before the old versions of objects are reaped.

Unlike ordinary Java threads, the Eventron's stack is not scanned to identify roots or fix stale pointers once it begins execution. Thus an Eventron could potentially load a stale pointer into a stack variable and dereference it after the old object (with its forwarding pointer) had been reaped by the collector and re-allocated.

This problem is addressed by performing a second iteration over the Eventron objects after pinning is completed. The two-phase nature of object pinning and forwarding is part of the implementation of pinObjects. The second iteration checks whether any pointers in the object are stale and immediately updates them to the forwarded copy of the referent.

This second pass guarantees that no stale pointers remain in any Eventron object. After this point, no Eventron object will be moved by the collector nor will any Eventron object contain pointers that will be modified by the collector. Therefore, the Eventron stack will never contain stale references and does not need to be examined, and at no point in the Eventron's execution must it synchronize with the collector.

Once all Eventron objects are pinned and their pointers are forwarded, an Eventron thread may be created. The run-time ensures that becomeEventronThread is invoked for each Eventron thread so that it may be distinguished from ordinary Java threads. This distinction is discussed in the following section.

7. Execution

7.1 Thread Priorities

Once an Eventron is released, it may arbitrarily preempt the garbage collector. Eventrons by default are created with a priority just higher than the thread that schedules the garbage collector (which itself runs at a priority just higher than the collector threads themselves).

Eventron priorities may be set to any value allowed by the underlying system. Eventrons co-exist with RTSJ NoHeapRealTimeThreads, and their relative scheduling is handled by the operating system based on their respective priorities.

7.2 Distinguishing Eventron Threads

Eventron threads must not yield to garbage collection threads at the beginning of each collector quantum. Therefore, we distinguish Eventron threads using a flag in a thread-specific data structure. This flag is used by the run-time scheduler to determine which threads must reach a safe point before a collection quantum can begin. In addition, as the collector is scanning the stack of each application thread, it must ignore the stacks of Eventron threads since these threads are not necessarily at safe points.

Because the stacks of Eventron threads are not scanned during collection, we must ensure that any object accessible to an Eventron is also reachable from some other root. The set of pinned objects associated with each Eventron serves to maintain this invariant: each set of pinned objects is added to a global table for the duration of the execution of the Eventron.

Eventron threads share some characteristics with NoHeapRealtimeThreads, for example, neither is suspended during the execution of the collector. However, we distinguish these two types of threads as Eventrons are not subject to the same checks on heap access.

7.3 Run-Time Exceptions

While our analysis has ruled out any allocation that may occur because of code intentionally written by the programmer, we have allowed the programmer to relax this restriction for code included from libraries, meaning that unanticipated allocations may occur, causing an immediate OutOfMemoryError. In addition, the system may perform allocation while handling an exceptional condition that arises during normal execution (*e.g.,* NullPointerException). We found it impractical to change all of the code paths involved, and typically the run-time does much more than allocate an exception object (*e.g.,* it also allocates a string of message text). In our implementation, these allocations also cause an immediate OutOfMemoryError. Despite this, our implementation ensures that all such exceptions contain valid traceback information, which simplifies debugging.

Exceptions may be caught by an Eventron, including both those exceptions that are preallocated and those that arise from allocation. Unhandled exceptions result in the termination of the execution phase. The Eventron will not be re-run until it is explicitly rescheduled.

8. Termination

Finally, once a Eventron has run its course, either after it has been disabled by a call from another thread, or because it threw or failed to catch an exception, its data structures are unpinned and the Eventron thread reverts to the ordinary state for a brief interval before terminating. During this brief interval, the Eventron thread may store into reference fields, and this explains how exceptions thrown during the execution phase are preserved for inspection by other threads.

When a Eventron is destroyed, the destruction routine unpins the associated data structure, so that Eventron objects that remain live due to pointers from the mutable heap are no longer pinned, and there is no "pinning leak."

8.1 Killing, Restarting, and Cost Enforcement

Since Eventrons do not modify pointer structures and do not hold locks (because all operations must be non-blocking), it should be possible to terminate an Eventron at any instruction without negative effects on the rest of the system.

This avoids another major problem with other approaches to high-frequency tasks in Java, namely that they may hold resources which make the semantics of early termination poorly defined.

The only circumstance under which killing Eventrons is unsafe is when the program is using a custom synchronization protocol in which the Eventron might expose partially updated state non-atomically. However, as long as our channel abstractions are used this will not happen.

This capability can be used to implement either asynchronous termination of the entire Eventron or of one particular execution. This meshes well with cost enforcement, since it allows the early termination of an iteration that misses its deadline, and immediately resets it to a state in which it can handle the next event in a timely fashion.

9. Eventron Channels

To aid programmers in transferring data between tasks of different priorities and frequencies, we provide an implementation of wait-free, allocation-free channels. These channels allow data to be streamed to and from Eventron tasks. Though we intend EventronChannels to be common utility classes, they are still checked by the validation process described above. We also provide the lower level Notifier object (discussed below as part of the implementation of EventronChannels). The Notifier, along with the properties of the Java memory model and/or the facilities of the java.util.concurrent.atomic package, can be used to implement other wait-free utilities.

EventronChannels must be wait-free to avoid priority inversion. We provide two versions of our channel abstraction: one that provides wait-free access to the reader and a blocking interface to the writer (called an EventronReadChannel) and another that provides a symmetric interface (EventronWriteChannel). (We also provide wait-free access for the low-priority half of these channels.) This allows for relatively straightforward implementations of both high- and low-frequency tasks: high-frequency tasks are driven by the scheduler, and if they avoid looping and use only wait-free structures such as EventronChannels, they should have predictable execution times. Low-frequency tasks can be designed to produce or process data as quickly as possible and can rely on the channel to throttle their execution.

Channels must also be free of allocation and pointer mutation to avoid synchronization with the collector. Like other Eventron data structures, they must perform all allocation during initialization. We specialize EventronChannels for the single reader/single writer case to allow efficient implementation in the context of these constraints.

Each EventronChannel provides a wait-free communication mechanism by implementing a circular queue or ring of primitive values. These values may be grouped together into buffers to provide more efficient access. EventronChannels come in seven "flavors," one for each of the each of the primitive Java types.

9.1 Programming Interface

At their core, EventronChannels are similar to input and output streams, providing methods to read and write individual elements and arrays of them. In the most basic interface, reading (respectively, writing) an array of values causes the elements of the array to be copied from (respectively, to) the internal data structure.

However, we also offer a slightly more complex, but lower overhead, alternative. A reader or writer may also acquire access to the internal buffer array. This allows data to be transfered directly between tasks without any additional copies. Programmers must be careful to relinquish all references to a buffer once it has been released for use by the other task. Failure to do so may lead to unexpected values being read from the channel.

In many applications, when a channel overrun or underrun occurs, the high-frequency task must continue to execute either by discarding data (in the former case) or using a default value (in the latter), while the error itself should be handled by some other task. EventronChannels provide a mechanism to automatically communicate these exceptional conditions back to the low-frequency task: a subsequent write (read) performed by the low-frequency task will throw an UnderrunException (OverrunException).

9.2 Configuration

The two most important channel configuration parameters are the number of buffers and the size of each buffer. By dividing the channel storage into a fixed set of buffers, we allow the programmer to control the rate of synchronization between the reader and the writer. Each buffer is owned by at most one of the two processes, allowing efficient access in the common case. Only when moving from one buffer to the next are atomic memory accesses (*e.g.*, compare-and-swap) required. The size and number of buffers must be determined given the relative jitter of the two tasks. The size of each buffer determines the minimum latency between when values are written to and read from the channel, and the number of buffers determines the maximum such latency.

In the audio playback example, the primary source of latency for the low-frequency task is the garbage collector. Suppose in this example that we are able to guarantee a maximum pause time of 5 ms and a minimum mutator utilization of 50% over any 10 ms window. Then the proper configuration is three buffers each containing 5 ms of data or, equivalently, 111 samples (recall that our example uses 22.05 kHz audio). Initially the (low-frequency) writer will be 10 ms ahead of the reader, but due to interruptions by the garbage collector, it may lag behind to the point where it is only 5 ms ahead. Such a scenario will only occur if the collector has consumed a significant portion of the previous scheduling quanta: by our minimum utilization guarantee, the collector will consume less of the subsequent quanta and allow the synthesizer to speed ahead once again.

The EventronWriteChannel has one additional configuration parameter. In the case of an overrun, some data will be lost: either the oldest data must be overwritten, or the newest must be discarded. Our implementation supports both, leaving the programmer to determine which is most appropriate for a given application.

9.3 Implementation

Each EventronChannel is implemented using an ImmutableArray of BufferState objects, where each such object encapsulates a buffer and an integer state variable. The state of each buffer determines both the state of the content (whether the buffer contains data or not) and who owns the buffer. For example, each buffer in an EventronReadChannel may be *empty*, *full*, *reading*, or *writing*, the last two indicating ownership by the reader or writer, respectively. There are two additional states, *underrun* and *closed*, that are used to communicate changes in the status of the channel as a whole. When an underrun occurs, the reader wrests ownership of the current buffer from the writer, setting its state to *underrun*. The next access by the writer will discover the state change and throw the an exception. Analogously, the writer may indicate the end of the data stream by setting the state of its current buffer to *closed*.

EventronChannels depend on atomic operations provided by the java.util.concurrent.atomic package. Atomic compare-and-swap operations are used to update the state of each buffer. For example, an underrun may occur at the same moment that the writer finishes writing to a buffer. Both tasks will attempt to update the state of this buffer (to *underrun* and *full*, respectively). An atomic update ensures that both tasks agree on the winner of the race.

Finally, the implementation of channels requires a mechanism to allow the high-frequency task to notify the low-frequency one when a buffer becomes available. The standard Java lock-notify-unlock is obviously unsuitable and while the implementation of locks in java.util.concurrent.locks includes tryLock and signal operations that avoid synchronization, they *do* perform allocation and pointer mutation, rendering them unusable by Eventrons. (This is partly because these locks support an unbounded number of waiters.) Instead, we provide a Notifier class that wraps any Object. It provides a notifyIfWaiting method, implemented internally to the virtual machine, which notifies threads (if any) waiting on the wrapped object's lock without blocking, allocating, or updating any reference field.

10. Evaluation

We have tested the performance of both validation and execution of Eventrons, in particular comparing them to equivalent C programs and evcaluating the interference from other (garbage collected) JVM threads.

All experiments were run on an IBM Intellistation Z Pro with 2 Pentium 4 Xeon processors (at 2.4 GHz and with 2-way hyperthreading) and 2 GB of physical memory. The operating system is Linux with the 2.6.14 kernel. Additionally, various patches to improve real-time performance were applied including high-resolution timers (HRT) [17] and robust mutexes for priority inheritance.

The base for the implementation is IBM's Real-Time edition of the J9 virtual machine, which includes the Metronome garbage collector, RTSJ, and an ahead-of-time compiler. The language level is 1.5 (with generics) using the J2SE libraries.

Our prototype implementation of Eventrons included the "fully constructed" check (Section 5.3.1) and safety checks for reflective field updates (Section 5.3.2). However, we have not finished integrating those aspects of the Eventron prototype into the production

Name	Methods	LOC	Validation Time (ms)
Null	1	1	0.11
Audible	4	87	4.16
FFT	8	139	4.58

Table 1. Validation Times for Several Eventrons. The Null Eventron has a trivial run() method, Audible is our music generation example, and FFT computes a fast Fourier transform.

VM used to gather the results reported below. But, based on our earlier experiments with the prototype implementation, we do not believe this materially impacts the results reported below.

All Eventrons were run at 22.05 KHz (a 45.35 μs period), a standard frequency for digital audio. Tests were run until 10,000,000 samples were obtained. These runs were sufficiently large to encompass periodic operating system activity which could perturb the results and to provide a statistically meaningful comparison. For example, the operating system flushes data to the disk every 30 seconds and performs timer interrupts every 1 millisecond.

10.1 Validation

We have measured the time required to validate a small set of examples. Table 1 shows the size of these examples and the time required to validate them. Each example includes any library code that may be executed by the Eventron, for example, some methods of EventronReadChannelOfShort in the Audible example. While these validation times are much longer than the execution of the Eventrons themselves, validation must only be performed once per instantiation of an Eventron.

10.2 Base Performance: Eventrons versus C

To establish the base timeliness results of Eventrons, we implemented a C program which uses the same OS function calls to establish a periodic task of the same 22.05 KHz frequency. Other than computation to establish the scheduling behavior, neither version performs any work. We measure the timeliness of both versions by measuring the interarrival times of the Eventron and C handler. Noting that the y-axis is logarithmic, figure 7 shows that nearly all of the data is centered tightly around the target period time of 45.3μs. More precisely, 99.64% of the samples lie within 5 μs of the target time. Within 10 μs of the target time, Eventrons include 99.997% of the data while C includes 99.999% of the samples.

Both the C version and the Eventron versions have nearly the same distribution, both in terms of the extent and count of both the outlier clusters and the main region. This curve represents the degree to which this particular version of Linux is real-time and represents the best we can achieve from user-level code.

In this graph, the histogram's x-axis ends at 120 μs although there is one additional group of outliers centered at 4.0 ms. In fact, all runs, regardless of whether it is C or Eventrons, which version of Eventrons, and the load under which it is run, suffer from the same effect. Thus, we believe that the effect stems from the operating system effect such as sync-ing the disk. In this particular figure, the C version has 12 outliers ranging from 3.93 to 4.06 ms while the Eventron version has 7 outliers ranging from 3.99 to 4.05 ms. Because this 4.0 ms effect is pervasive and stems from the operating system, we will not show nor discuss these outliers in the remaining sections.

10.3 Timeliness in the presence of GC

Having established that Eventrons can potentially achieve latency similar to that of a C application, we now test the main design feature of Eventrons by measuring the effect that garbage collection has on Eventron interarrival times. To do this, the Eventron is

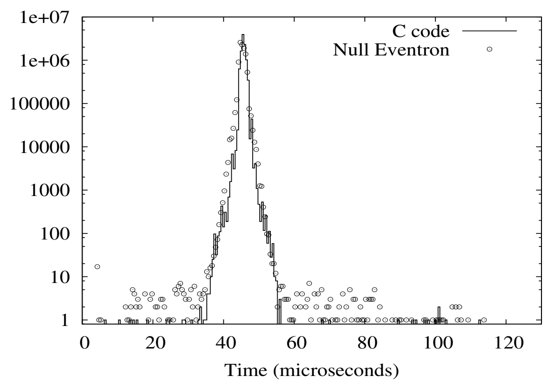

Figure 7. Histogram of interarrival times for a null Eventron scheduled every 45 μs and for a simple C program running at the same period. The Eventron and the C program run 99.997% and 99.999% of their executions within 10 μs of the target times, respectively.

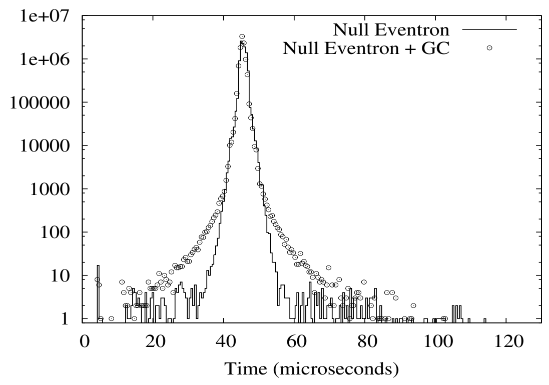

Figure 8. The effect of garbage collection on the interarrival times of an Eventron scheduled every 45 μs. Running a garbage collector increases the number of outliers between 10 and 25 μs from the target time but otherwise does not impact the shape of the distribution.

run together with a lower-priority Java thread that continually allocates, thus triggering periodic garbage collections. The Eventron's scheduling is potentially affected by both the mutator thread and the garbage collection thread. As we can see in figure 8, these two effects combine to slightly worsen the behavior of the Eventron. The distribution is not as tight as before so that at 10 μs, 99.990% of the data is included. However, nearly all of the distribution lies within 25 μs of the target time. In fact, the tail of the distribution is better when the mutator thread and the garbage collection threads are enabled.

To demonstrate the effectiveness with which Eventrons address the latency problem, we replace the Eventron with an equivalent Java version which runs at high priority. Figure 9 shows that the interarrival times of a regular Java thread are far worse than that of an Eventron. In fact, the outliers are so distant that we extended the x-axis from 130 μs to 1 ms. The number of outliers increases by a factor of 20, and these outliers are centered around 550 μs which is slightly greater than the time the garbage collection thread typically runs for. For non-realtime garbage collectors, the interarrival times would be even worse.

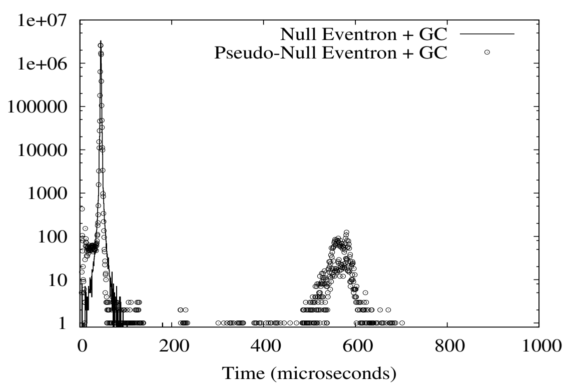

Figure 9. Histogram of interarrival times of the null Eventron and a null non-Eventron both scheduled every 45 μs. The non-Eventron, despite running at a higher priority, cannot preempt the garbage collector and is often interrupted for periods of approximately 550 μs.

10.4 Performance Under Load: Eventrons with CPU-bound Threads

After having established that the base performance of Eventrons is similar to that of a similar C application, we test the performance of Eventrons when run in conjunction with regular Java threads in a single Java VM. To stress the system, we choose four benchmarks from the SPECjvm98 suite (`db`, `jack`, `javac`, and `mpegaudio`) which are CPU-bound batch applications so that the threads are almost always eligible to run.

As Figure 10 shows, the Eventrons are slightly less timely when the system is under load and the deviation is dependent on the load. With `db`, the deviation is small, and 99.990% of the data falls within a 10 μs window. On the other hand, `javac` causes greater perturbations, and a 10 μs window includes only 99.986% of the samples. While these numbers are still very high, the differences are significant and show that other threads do cause definite perturbations. Because this version of Linux does not have the pre-emption patches (which at the time were too unstable), various system calls can cause the kernel to enter critical sections in which it will not yield. For example, a lower-priority thread can request a non-blocking I/O operation which nonetheless will block an unrelated Eventron thread.

Figure 11 shows the passage of time as a series of strips or "scan lines" moving from left to right and then top to bottom. An Eventron (dark gray) is executed every 45 μs. The width of oscilloscope view is exactly four times the period of the Eventron, thus the Eventron executions line up vertically. The alarm task (part of the GC scheduler) is shown in black and runs with a period of 200 μs, slightly longer than the width of the view, resulting in the staggering effect. Finally the GC execution is shown in light gray. Note that the Eventron continues to run at fixed intervals despite the execution of the other tasks.

10.5 Interaction with Low-Frequency Tasks: Audible Eventrons

Finally, we test the performance of the audible Eventron that was described in section 2. In addition to showing perturbation from the lower-priority low-frequency task, this example demonstrates that the wait-free channels perform as described. Since the audible Eventron writes to /dev/dsp, timing anomalies may arise from operating system interactions. In fact, despite using non-blocking I/O, the system call write periodically takes quite a while. Because this anomaly is not central to the Eventron design but deals with lower-

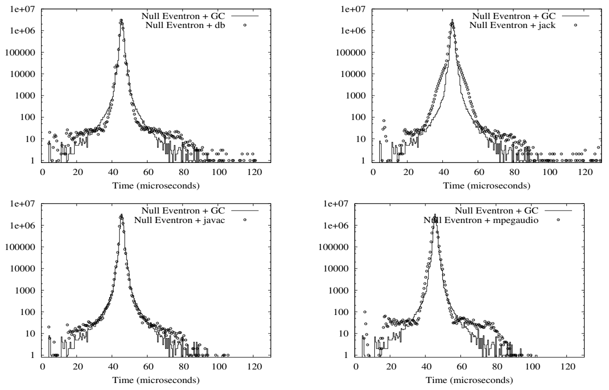

Figure 10. Histogram of interarrival times for an Eventrons scheduled every 45 μs in a virtual machine running one of several applications taken from the SPECjvm98 benchmark suite. Increasing the load within the virtual machine has small effects on Eventron performance, probably due to contention in the scheduler or uninterruptible kernel calls.

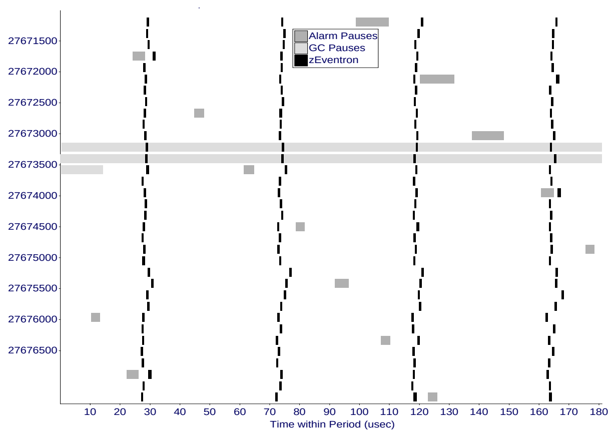

Figure 11. Oscilloscope view of an Eventron running every 45μs. Time moves from left to right and top to bottom. Notice that the Eventron (dark gray) continues to run concurrently with the collector (light gray). The "Alarm Run" (black) points denote the execution of the GC scheduler task.

level aspects of specific devices, we first present performance result without the write in Figure 12. There are no significant changes to the tail of the distribution although the central portion of the distribution grows slightly wider similar to effect of garbage collection in Figure 8. The similarity suggests that the operating system does not always promptly schedule high-priority threads even when there is no interaction (via mutexes, for example) between the high- and low-priority threads.

Finally, we show the pronounced effect of including the call to write in the audible Eventron in Figure 13. There is clearly a cluster of data centering around 200 μs. Examination of the raw data shows that every 93 ms, the call to write takes up to 220 μs instead of the usual 2 μs. The audio card performs double buffering and each buffer is 2048 samples in size. At 22050 Hz, this corresponds to 93 ms so we believe that, despite opening /dev/dsp with non-blocking I/O (which did improve matters), certain calls are still affected by buffering operations within the kernel.

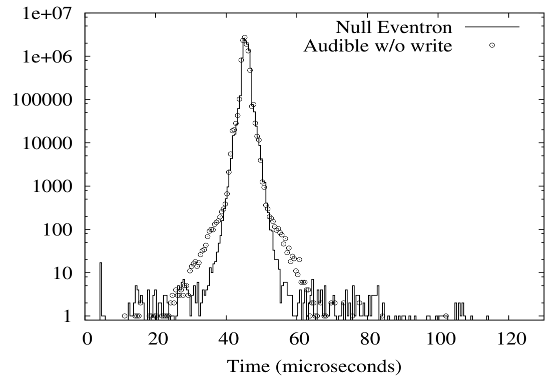

Figure 12. Histogram of interarrival times of the audible Eventrons without the write scheduled every 45 μs compared to a null Eventron running at the same frequency. Running in parallel with the low-frequency task has only a small impact on performance. This effect similar to that of the collector, indicating that it is an artifact of the OS scheduling algorithm and not the interaction between tasks.

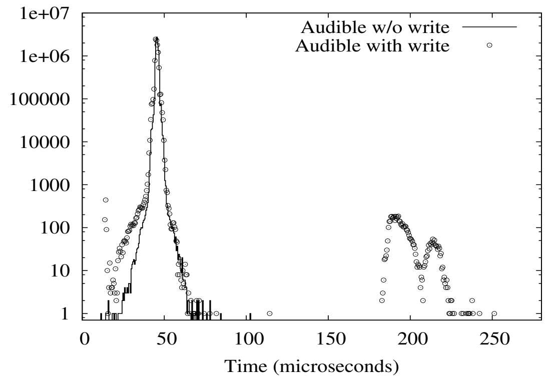

Figure 13. Histogram of interarrival times of the audible Eventron without the write and the audible Eventron with the write with both scheduled every 45 μs. Buffering by the audio driver (during a call to write) causes outliers to occur every 2048 samples or 93 ms.

11. Related Work

Eventron validation involves constructing a call graph and analyzing it to detect violations of the programming model. A wide range of call graph construction algorithms are described by Grove and Chambers [13]. Static analyses have also been adapted to operate on-line and handle Java language features such as reflection, dynamic class loading, and JNI. For example, Hirzel et al. [16] describe a pointer analysis that supports the full Java programming language. Similarly, Qian and Hendren [21] augment a standard call graph construction algorithm to support dynamic class loading. Unlike all of this previous work, our Eventron validator analyzes the code in the context of the exact object instances on which it will operate. This significantly simplifies analysis since the actual object instance can be reflectively examined to determine the types and contents of its fields.

The Real-Time Specification for Java (RTSJ) is a standard that defines a number of extensions and changes to the virtual machine and standard Java libraries to support real-time application development. The RTSJ defines NoHeapRealtimeThreads (NHRTs) as the proper high frequency tasks. NHRTs are forbidden from accessing any object allocated in the garbage-collected heap. Instead, NHRTs may allocate from either the ImmortalMemoryArea or one of the ScopedMemoryAreas. Objects allocated in the former will be preserved for the remainder of the life of the virtual machine; thus the ImmortalMemoryArea is suitable for only those objects that are allocated during the initialization phase of an application.

Programming effectively and safely with RTSJ scoped memory is widely recognized as being quite challenging. Pizlo et al. [20] present a collection of programming idioms and design patterns for using scoped memory. Bollella et al. [5] describe the use of *restricted memory pools* and *scoped memory scratchpads* as key design patterns in their use of RTSJ to control loops in the Golden Gate/Mission Data System project. They advocate the use of a framework based on these patterns to insulate most programmers from the complexities of directly using RTSJ scoped memory.

Deters and Cytron [10] describe a dynamic analysis to facilitate the conversion of a standard Java program to one that uses RTSJ scoped memory. By instrumenting the program to observe object lifetimes and referencing patterns, their system can suggest where to place scoped memory regions. However, the system's suggestions may be unsound since they are based on a particular program execution.

The runtime checks required by RTSJ's scoped memory can be quite expensive. Higuera-Toledano and Issarny [15] quantify these overheads and analyze some solutions to reduce their impact. Corsaro and Cytron [8] present efficient algorithms for managing scoped memory and performing the required dynamic checks in addition to an empirical evaluation.

Static type systems have been proposed to make it easier to write correct programs using RTSJ's scoped memory abstractions. In these systems, correctly typed programs cannot cause memory access violations, therefore all of the RTSJ mandated dynamic memory checks can be eliminated. Boyapati et al. [6] describe one such system based on ownership types. Zhao et al. [24] develop a simpler, but more restrictive system which they prove sound. In effect, both of these type systems statically partition heap connectivity, but allow real time tasks to dynamically allocate memory (using RTSJ scoped memory) within their heap partition. In our Eventron programming model, heap connectivity is not restricted but dynamic memory allocation by the Eventron is forbidden.

Corsaro and Schmidt [9] report on a detailed performance comparison of their jRate RTSJ implementation and the RTSJ reference implementation (RI) from TimeSys. One relevant data point from their experiments is that they observed that the RTSJ RI Periodic Thread implementation was only able to achieve predictable behavior for tasks with periods greater than or equal to 30 ms. Higher frequency tasks suffered significant jitter. In contrast, our Eventron implementation is capable of achieving highly predictable behavior for tasks with periods as small as 45 μs.

12. Conclusion

Eventrons provide a simple and elegant means of writing tasks requiring extremely low latency while co-existing with a garbage-collected language. By taking advantage of Java's existing final mechanism, augmented with some run-time support to eliminate loopholes that could compromise safety, a simple *data-sensitive* analysis is able to guarantee safety before Eventrons begin executing. Eventrons are simpler to program, simpler to implement, more reliable, and more efficient than other approaches like scoped memory.

Eventrons make it feasible to program almost any real-time system entirely in pure Java code. We are currently investigating the use of Eventrons to directly generate CD-quality audio waveforms and to write device drivers. We believe that Eventrons will make it possible to replace even the lowest levels of system software with safe code in a managed run-time system.

Acknowledgements

We thank Daniel Frampton for invaluable assistance with J9 features required for our work, as well as Christoph Kirsch, Bob Harper, and Julian Dolby for helpful discussions. We also thank Graham Chapman, Mike Fulton, and Ryan Sciampacone for their help in incorporating the Eventron prototype into the J9 code base.

References

[1] ARNOLD, M., FINK, S. J., GROVE, D., HIND, M., AND SWEENEY, P. F. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE 93*, 2 (2005). Special issue on Program Generation, Optimization, and Adaptation.

[2] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 2003). *SIGPLAN Notices*, *38*, 1, 285–298.

[3] BOBROW, D. G., AND MURPHY, D. L. Structure of a LISP system using two-level storage. *Commun. ACM 10*, 3 (1967), 155–159.

[4] BOLLELLA, G., ET AL. *The Real-Time Specification for Java*. The Java Series. Addison-Wesley, 2000.

[5] BOLLELLA, G., ET AL. Programming with non-heap memory in the real time specification for Java. In *Companion Papers the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2003), pp. 361–369.

[6] BOYAPATI, C., SALCIANU, A., BEEBEE, JR., W., AND RINARD, M. Ownership types for safe region-based memory management in real-time Java. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, 2003), pp. 324–337.

[7] CHENEY, C. J. A nonrecursive list compacting algorithm. *Commun. ACM 13*, 11 (1970), 677–678.

[8] CORSARO, A., AND CYTRON, R. K. Efficient memory-reference checks for real-time Java. In *Proceedings of the 2003 ACM SIGPLAN conference on Languages, compilers, and Tools for Embedded Systems* (New York, NY, USA, 2003), pp. 51–58.

[9] CORSARO, A., AND SCHMIDT, D. C. Evaluating real-time Java features and performance for real-time embedded systems. In *Proceedings of the Eight IEEE Real-Time and Embedded Technology and Applications Symposium* (2002).

[10] DETERS, M., AND CYTRON, R. K. Automated discovery of scoped memory regions for real-time Java. In *Proc. of the Third International Symposium on Memory Management* (Berlin, Germany, June 2002). *SIGPLAN Notices*, *38*, 2 Supplement, 25–35.

[11] FINK, S., DOLBY, J., AND COLBY, L. Semi-automatic J2EE transaction configuration. Tech. Rep. RC23326, IBM Thomas J. Watson Research Center, Apr. 2004.

[12] GROSSMAN, D., MORRISETT, G., JIM, T., HICKS, M., WANG, Y., AND CHENEY, J. Region-based memory management in Cyclone. *SIGPLAN Notices 37*, 5 (May 2002), 282–293. In *Conference on Programming Language Design and Implementation (PLDI)*.

[13] GROVE, D., AND CHAMBERS, C. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst. 23*, 6 (Nov. 2001), 685–746.

[14] The Haskell 98 foreign function interface. www.cse.unsw.edu.au/-∼chak/haskell/ffi/.

[15] HIGUERA-TOLEDANO, M. T., AND ISSARNY, V. Analyzing the performance of memory management in RTSJ. In *Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time distributed Computing* (2002).

[16] HIRZEL, M., DIWAN, A., AND HIND, M. Pointer analysis in the pressence of dynamic class loading. In *18th European Conference on Object-Oriented Programming (ECOOP)* (June 2004), vol. 3086 of *LNCS*, pp. 96–122.

[17] High-Res Timer Kernel Patches. `i386-hrt-2.6.10.patch` and `hrt-common-2.6.10.patch` at http://easynews.dl.sourceforge.net/-sourceforge/high-res-timers.

[18] LIANG, S. *The Java Native Interface: Programmer's Guide and Specification*. Addison Wesley, 1999.

[19] MILLER, J. S., AND RAGSDALE, S. *The Common Language Infrastructure Annotated Standard*. Addison Wesley, 2004.

[20] PIZLO, F., FOX, J. M., HOLMES, D., AND VITEK, J. Real-time Java scoped memory: Design patterns and semantics. In *Proceedings of the 7th IEEE International Symposium on Object-oriented Real-time distributed Computing* (May 2004).

[21] QIAN, F., AND HENDREN, L. Towards dynamic interprocedural analysis in JVMs. In *3rd Virtual Machine Research and Technology Symposium (VM)* (May 2004), pp. 139–150.

[22] TOFTE, M., AND TALPIN, J.-P. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. of the Twenty-first ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, 1994), pp. 188–201.

[23] UNGAR, D. M. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, 1984), P. Henderson, Ed. *SIGPLAN Notices*, *19*, 5, 157–167.

[24] ZHAO, T., NOBLE, J., AND VITEK, J. Scoped types for real-time Java. In *Proc. of the Twenty-fifth IEEE International Real-Time Systems Symposium* (2004), pp. 241–251.