Reflexes: Abstractions for Integrating Highly Responsive Tasks into Java Applications

Jesper Honig Spring[1], Filip Pizlo[2], Jean Privat[2], Rachid Guerraoui[1] and Jan Vitek[2]

[1] Ecole Polytechnique Fédérale de Lausanne
[2] Purdue University

Achieving sub-millisecond response times in a managed language environment such as Java or C# requires overcoming significant challenges. In this paper we propose Reflexes, a programming model and runtime system infrastructure that lets developers seamlessly mix highly responsive tasks and timing-oblivious Java applications. Thus enabling gradual addition of real-time features, to a non-real-time application without having to resort to recoding the real-time parts in a different language such as C or Ada. Experiments with the Reflex prototype implementation show that it is possible to run a real-time task with a period of 45 μs with an accuracy of 99.996% (only 0.001% worse than the corresponding C implementation) in the presence of garbage collection and heavy load ordinary Java threads.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*run-time environments*

General Terms: Languages, Experimentation

Additional Key Words and Phrases: Real-time systems, Java virtual machine, Memory management

1. INTRODUCTION

The state of the art in real-time system development is that most real-time system programming models are defined as a function of the hardware, operating system and libraries available to the programmer. Not surprisingly this state of affairs leads to non portable codes, and low rates of reuse across project based on different infrastructures. The last decade, there has been a push by industry to switch to high-level programming languages such as Java and C# which have benefits such as memory safety and protability across operating systems and hardware platforms. These languages have seen widespread adoption for multiple reasons, not the least of which being higher developper productivity. Unfortunately, these benefits come at a price, namely the heavy-weight runtime infrastructure needed to support the execution of those languages. For instance, in Java the absence of memory errors is predicated on the use of complex garbage collection algorithms that can introduce pauses in the hundreds of milliseconds in a high-performance implementation. Other popular runtime services such as dynamic class loading and just-in-time compilation can also introduce pathological behaviors.

The goal of this paper is not to propose Java as a replacement for system's programming languages such as C or Ada. Instead, we take the position that there are many systems that are predominently made up of timing-oblivious code with little nuggets of soft- or hard-real-time behavior. The question we are investigating is how to seamlessly extend a non-real-time application written in a high-level language with real-time tasks without having to switch languages. We would like, as much as possible, to retain the benefits of memory safety, portability, reusability of Java, yet support the definition of real-time components. The solution space is

constrained by our desire to keep the semantics of plain Java system intact. We investigate a solution that relies on novel runtime support and static type checking.

Different approaches have already been explored to bring real-time capabilities to Java. At one end of the spectrum, one can envision running unmodified Java application in a virtual machine carefully engineered to avoid pausing user-code and employ a real-time garbage collector to bound the latencies due to memory management. The state of the art in real-time garbage collection is below 1 millisecond maximum pause times and a 2× slow down due to garbage collection overheads [Pizlo and Vitek 2008]. This approach has the benefit of requiring no changes to programs and thus being perfectly backwards compatible. At the other end of the spectrum, one could amend the Java language to better support real-time. This is the approach chosen in the Real-time Specification for Java (RTSJ) [Gosling and Bollella 2000] which changes and extends the semantics of Java to provide strong real-time guarantees. RTSJ programs can execute without interference from the garbage collector and thus potentially run much faster and with better response times. The drawbacks of the RTSJ is that it is invasive; the whole program and all libraries have to be aware that they may be executed in a real-time context. Furthermore, some of the design choices underlying the RTSJ entail runtime overheads and the possibility of memory access errors that are not caught by the type system. We defer to [Pizlo et al. 2004] for a discussion of some of these drawbacks and to [Auerbach et al. 2007; Dawson and Thwaite 2008; Armbuster et al. 2007; Bollella et al. 2005] for a discussion of the challenges faced by implementers of the RTSJ.

This paper proposes a different approach. We introduce *Reflexes,* a programming model for mixing highly-responsive tasks with timing-oblivious Java programs. This work is based on the first author's phd thesis [Spring 2008], and our VEE'07 [Spring et al. 2007] and OOPSLA'07 [Spring et al. 2007] papers. While Reflexes require a modified virtual machine, it does not entail changes to the libraries or user programs. Software components that have no time constraints are left untouched and will be unaware that they are not on plain Java virtual machine. Real-time tasks, however, are written using the Reflex abstractions. Reflex defines a restricted subset of the Java language and libraries extended with a facility for safe region-based memory management, obstruction-free atomic regions and real-time preemptive scheduling. Reflex enforce strong static memory partitioning between data belonging to plain Java threads and data used by Reflex tasks. This partitioning is done by the compiler and does not incur run-time overheads.

We report on two implementation of Reflex. The first implementation effort was carried out on top of the Ovm virtual machine [Armbuster et al. 2007] which provides support for real-time Java on uni-processor systems. Ovm is an ahead-of-time compiler, the code of the entire application is translated to C and compiled with an off-the-shelf compiler such as gcc. The second implementation uses a commercial real-time Java virtual machine with chip-level multiprocessor support. Our experiments show that Reflexes provide better latency than either a real-time collector or the commercial implementation of the RTSJ. We have run tasks with periods as low as 45 μs without background noise due to plain Java task and the Java garbage collection and obtained an accuracy of 99.996%. This is only 0.001% worse than the corresponding C implementation. We argue that Reflexes are a promising approach to incorporate real-time processing in the Java language.

2. RELATED WORK

The most closely related work to this paper is the Eventron [Spoonhower et al. 2006] and Exotask [Auerbach et al. 2007] real-time programming models developped in parallel by Auerbach *et al.* at IBM Research. Both models have the goal of extending Java in a non-intrusive way with real-time features. They differ in the constraints they impose on programs and the real-time guarantees that can be achieved.

Eventrons provide strong responsiveness guarantees at the expense of expressiveness. In the Eventron model, a real-time task cannot allocate new objects. Furthermore, it is prevented, by load-time compiler inserted checks, from writing to reference variables or even reading reference variables that may be modified by other threads. The constraints on reference variables are particularly stringent and entail that computation in an Eventron is limited to modification of scalar variables. The motivation for these constraints is that they enable Eventron tasks to preempt the garbage collector at any time, even when the heap is in the process of being compacted and all references are not up-to date. This is the key to being able to achieve response times in the microseconds on Java platform. Reflexes have similar responsiveness but are less restrictive, we take advantage of our the type system partition memory. In the memory partition that belongs to a Reflex allocation and reading/writing reference variables is supported.

In later work [Auerbach et al. 2007], Eventrons were generalized to form a graph of tasks called Exotasks. Like Reflex, the tasks partition the memory of the virutal machine in disjoint areas which are kept disjoint. The main innovation in Exotasks was that tasks could be garbage collected. As the memory used by individual tasks is disjoint, the collection is task-local and can usually be carried out in very little time. Tasks communicate by exchanging messages by deep-copy.

In a recent collaboration with IBM Research, we have successfully transitioned the key ideas of Reflex (and it's follow up called StreamFlex [Spring et al. 2007]) in the context of the IBM production virtual machine. The resulting envionment, referred to as FlexoTask [Auerbach et al. 2008], adopts the ownership type system introduced in this paper as well our obstruction-free atomic region abstraction.

3. PROGRAMMING MODEL OVERVIEW

A Reflex program consists of a graph of Reflex tasks[1] connected according to some topology through a number of unidirectional *communication channels*. This relates directly to graph-based modeling systems, such as Simulink and Ptolemy [Lee 2003], that are used to design real-time control systems, and to stream-based programming languages like StreamIt [Thies et al. 2002]. A Reflex graph is constructed as a Java program, following standard Java programming conventions, and using standard Java development tools.

Reflexes can run in isolation or as part of a larger Java application. To interact with ordinary Java threads, Reflex provides special methods which will ensure that real-time activities do not block for normal Java threads. Fig. 1 illustrates a Reflex program and its interaction with an ordinary Java thread.

[1] Note, we use the term *Reflexes* to denote both the programming model as well as the tasks.

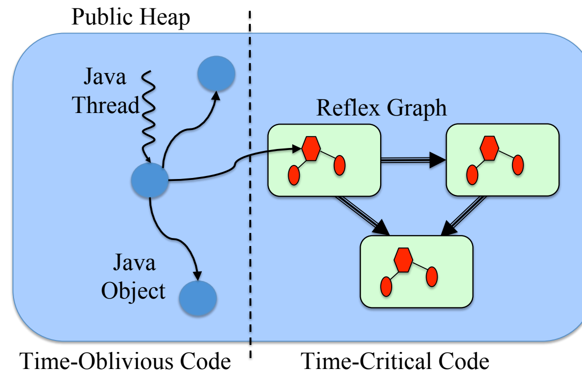

Fig. 1. Illustration of a Java application consisting of time-oblivious code (blue) and a time-critical Reflex graph with three connected tasks.

A Reflex acts as the basic computational unit in the graph, consisting of user-defined persistent data structures, typed input and output channels, and user-specific logic implementing the functional behavior of the task. In order to ensure low latency, each Reflex lives in a partition of the virtual machine's memory outside of the control of the garbage collector. Furthermore, Reflexes are executed with a priority higher than ordinary Java threads. This allows the Reflex scheduler to safely preempt any Java thread, including the garbage collector. Memory partitioning also prevents synchronization hazards, such as a task blocking on a lock held by an ordinary Java thread, which in turn can be blocked by the garbage collector.

In terms of memory management, a Reflex is composed of *stable* objects, *transient* objects and *capsules*. Stable objects include the Reflex itself as well as any internal data structure that must have a lifetime equal to that of the graph. Transient objects live only while the Reflex is active. This reflects the split between temporary data needed during one activation of a periodic activity and data that persists for the duration of a program. Capsules are data objects used as messages between Reflexes. They persist as long as they are referenced by a channel or task. Their contents is severely restricted. Specifying whether an object is stable, transient or capsule is done at the class level.

The Reflex runtime infrastructure includes a scheduler that is responsible for releasing Reflexes. While a Reflex can become schedulable any time new data appears on one of its input channels, the scheduler does not, in general, attempt to guarantee timeliness; only that each task will eventually be released. If the programmer requires timely execution, *clocks* must be used. When a Reflex is connected to a clock, the scheduler arranges for the task to be released according to the clock's period. Hence, with this scheme, a periodic activity is modeled with a single Reflex connected to a clock. While multiple threads can drive the execution of a graph, individual Reflexes are single-threaded.

3.1 Reflex Graph

A graph is constructed by extending the built-in abstract `ReflexGraph` class, and implementing at least one of its constructors. The graph is, in turn, responsible

of creating tasks and connecting them according to the desired topology. Once a graph is fully constructed, the `validate` is invoked to check the well-formedness of the graph. Safety of memory operations is checked statically as part of compilation of the Reflex classes.

```
public abstract class ReflexGraph {
    public ReflexGraph(int priority, int commAreaSize);
    public final void start();
    public final void stop();
    protected final void validate() throws ValidationException;
    protected final ReflexTask create(Class taskClass);
    protected final ReflexTask create(Class taskClass, int stableSz, int transientSz);
    protected final Clock createClock(int periodInMicros);
    protected final void connect(Clock source, ReflexTask target, String targetField);
    protected final void connect(ReflexTask source, String sourceField,
                                 ReflexTask target, String targetField, int size);
}
```

Fig. 2. An excerpt of the abstract `ReflexGraph` class to be subclassed by the programmer in order to create and connect tasks in the graph according to user-specific requirements.

Validation involves verifying that (1) channels are connected to fields of the proper types; (2) that sufficient space is available within the private memory areas of the tasks; and (3) the communication area (whose size is set in the `ReflexGraph` constructor), and that clocks are configured with periods supported by the underlying virtual machine. Cyclic graphs are allowed in the validation phase as they do not necessarily run indefinitely. Once validated, the graph's topology is fixed and the `start()` method can be invoked to schedule the Reflex in the graph.

Fig. 2 shows the methods for the reflective creation of Reflexes and channels. Reflection is needed because the data structures representing these abstractions must be allocated under the control of the runtime infrastructure in the proper memory regions to make sure that they are not traversed by the garbage collector.

3.2 Reflexes

A Reflex is the computational unit in our model, and is constructed by extending the built-in abstract `ReflexTask` class. The programmer must implement an `execute()` method, which defines the functional behavior of the task. Fig. 3 shows an excerpt of this class.

The `execute()` method is invoked by the Reflex scheduler when the Reflex is schedulable. This occurs upon the arrival of data on one of its input channels according to the specified rate on the channels. More specifically, the rate specifies how much data the task needs on its individual input channels in order to execute. By default, each channel's rate is set to one, but this can optionally be overridden.

By convention, the `execute()` method is expected to yield and give control back to the runtime infrastructure – in most cases, it would be a programming error for an activity to fail to yield as this could block all tasks in the graph and cause deadline misses.

```
public abstract class ReflexTask implements Stable {
  public ReflexTask(int transientSize, int stableSize) {...}
  public abstract void execute();
  public void initialize() {}
  protected final Capsule makeCapsule(Class c) {...}
}
```

Fig. 3. An excerpt of the `ReflexTask` class to be subclassed by the programmer. Its `initialize()` and `execute()` methods provide the user-specific functional behavior.

A Reflex can also provide an `initialize()` method that is invoked by the infrastructure to initialize the task before it starts (but after the tasks in the graph have been connected).

3.3 Memory Management

Reflexes execute in complete isolation from the Java heap, instead they run in their own (heap-allocated) private memory region, illustrated in Fig. 4. The `ReflexTask` instance itself is allocated within its own private memory region to shield it from the garbage collector.

The memory region of a Reflex is partitioned between a *stable heap* and a *transient area*. The sizes of both regions are chosen at startup as illustrated in Fig. 3. As the stable heap has a fixed size, the allocation of stable objects must be managed carefully to avoid running out of memory. The transient area is also fixed in size and serves as a per-invocation scratchpad. Once the `execute()` method returns, all allocations made in the transient area during its execution will be reclaimed in constant time; any allocations made on the stable heap will remain. Our design assumes that the allocation of persistent state is the exception. Specifying whether an object is stable or transient is done at the class level. By default, data allocated by or within a task is directed to the transient area. Only objects of classes im-

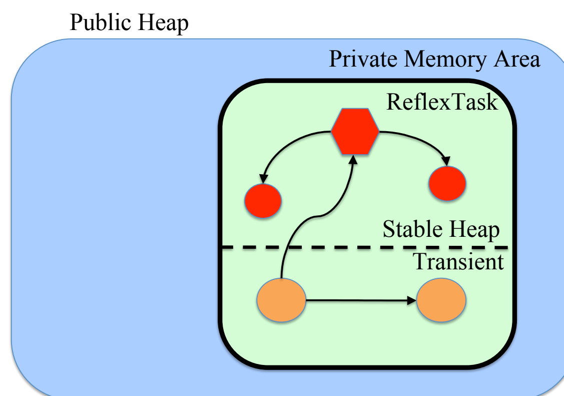

Fig. 4. Illustration of the memory model of a Reflex task (hexagon) in its own private memory area with its object graphs of stable (red) and transient (orange) objects (circles).

plementing the `Stable` marker interface will be put on the stable heap and persist between invocations.

The `ReflexTask` class is declared stable and will always be allocated in the stable heap of its own private memory area. Given different lifetimes different the objects, for memory safety reasons, stable objects are restricted from referencing transient objects; whereas transient objects can reference both transient and stable types. Referencing a transient object from a stable one could lead to a dangling pointer once the transient area has been reclaimed. Finally, allocations made by a Reflex are never directed to the public heap.

By using class granularity for distinguishing between stable and transient objects, we relinquish the possibility of using the same class in both memory contexts. The alternative approach would be to introduce some per-object annotation, e.g., one could write code like `@stable HashMap h = @stable new HashMap()` where the annotation `@stable` is used to denote data that resides in stable memory. Unfortunately that is not sufficient. Specifically, the problem is that the code within the `HashMap` class may itself perform allocations, and those allocations would have to be destined in the exact same stable memory context to be consistent. Here, an approach treating the annotation as a type parameter, e.g. `new HashMap<@stable>()`, would help. However, this approach is notationally cumbersome and requires retrofitting all library classes with generic parameters. The added effort and complexity does not seem warranted.

Another design choice is that the transient area is the default allocation context. Unlike for stable classes, transient classes 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).

3.3.1 *Stable Arrays*

As mentioned, the default allocation context is the transient area. Following this design choice, primitive array objects allocated using statements such as `int[] ia = new int[10]` are thus transient. It follows that stable objects cannot reference standard array objects. It is not reasonable to forbid stable arrays, Reflex provides two special cases. The Reflex API introduces a `StableArray` base class and provides a set of subclasses for each of the available primitive types. These classes encapsulate the different primitive arrays, and, as their names indicate, enable the allocation of these arrays in the stable heap. For reference types, Reflex takes the position that an array of `Stable` types is considered stable.

3.4 Exceptions

Given this distinction between object lifetime, exception handling within a Reflex requires special attention. When an exception is thrown within a Reflex, the object and its stack trace are created in the transient area, and will be reclaimed like any other object following the completion of the invocation of the `execute()` method. If the exception propagates out of the `execute()` method, the stack trace is logged and the graph is terminated.

3.5 Reflex Communication

Reflex provides type-safe, non-blocking communication between the individual Reflexes and between ordinary, time-oblivious Java threads and time-critical Reflex tasks.

3.5.1 *Non-Blocking Channels*

Inter-task communication is designed with a key requirement in mind; enabling non-blocking, zero-copy messaging between Reflexes. A Reflex communicates with other tasks solely through non-blocking channels. A channel is a fixed-sized, typed buffer connecting two Reflexes. The infrastructure supports primitive type channels (all of Java's primitive types), time channels holding periodic time-stamps, and a restricted set of objects belonging to subclasses of the `Capsule` base class.

Fig. 5 gives an overview of the `CapsuleChannel` class which is straightforward. The `TimeChannel` class is different in order to avoid storing, potentially large, numbers of clock ticks. Hence, it has two methods, one to put a current clock tick in microseconds on the channel, and one to return the latest unread clock tick.

```
public class CapsuleChannel extends Channel {
  public int size();
  public put(Capsule val);
  public Capsule take();
}
```

Fig. 5. Excerpts of the `CapsuleChannel` classe for transferring `Capsule` type data.

The operations performed on a channel during a given release are atomic. Once a Reflex starts executing, its channels are logically frozen, no other task is allowed to modify them. All changes to channels are published when the `execute()` method returns. Channels are created when two `ReflexTask` instances are connected by a call to the `connect()` method on the `ReflexGraph` class. The method will create the channel with its given size, and set the fields of the Reflexes. The connection is done by reflection based on the name provided as argument.

Channels are allocated in a memory region separate from any of the Reflexes using them – the communication area, as depicted in Fig. 6. Capsules are also allocated in this region. The region is, like the private memory area, free of interference from the garbage collector. It is fixed-sized, and so the programmer has to carefully size the communication area to account for channels and capsules it holds. While this at first appears limiting, the actual number of capsule types used in an application as well as the instances created of each type, in our experience, are typically bounded. The actual allocation of the memory area is performed by the Reflex runtime as part of the instantiation of the `ReflexGraph`.

Reflex does not support growable channels and, in case of overflow, silently drops packets. Other policies have been considered but have not been implemented. Variable sized channel, for example, can be added if users are willing to take the chance that put operations take variable time.

3.5.2 *Capsules*

Given the goal of zero-copy messaging, using capsules on channels turns out to be challenging in the absence of a garbage collector. Indeed, the question of where to allocate capsules, and when to deallocate them is a difficult one. They cannot be allocated in the transient memory of a Reflex as they would be deallocated as soon as the `execute()` method completes. Likewise, they should not be allocated in stable memory for fear of running out of space. Instead, as mentioned above, capsules are allocated from a pool managed by the infrastructure. The invocation to `makeCapsule()` causes for a capsule of the requested type to be returned, this is either an existing capsule or a newly allocated one. Capsules are returned to the pool as soon they are not referenced by any task or channel. The Reflex type system and runtime infrastructure ensure that a capsule can be accessed by at most one task and be on at most one queue at a time. For pragmatic reasons, there is one loophole in the zero-copy semantics, if a Reflex needs to put a capsule on multiple output channels, the capsule will be copied in order to preserve the single-reference invariant.

To guarantee memory safety, capsule classes are restricted. Specifically, to preserve isolation between tasks, a Reflex must not retain a reference to a capsule that has been pushed to its output channel, and a capsule should not retain references to the task's stable data. In fact, a capsule must not leak references. This is achieved by restricting capsules to reference-immutable data types. Informally, an object is reference-immutable if none of its reference fields, of transitively reachable reference fields, can modified. For pragmatic reasons, we restrict capsules a bit further and limit their fields to primitive types and arrays of primitive types. While these constraints have proved acceptable so far, one could loosen them if they prove to be too stringent, e.g., by allowing capsules to have `final` fields of any reference-immutable types. However, this will come at the price of more complex set of static checks.

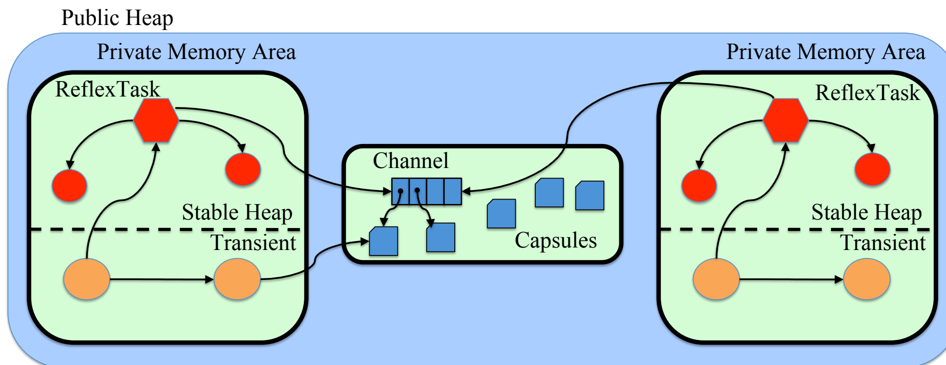

Fig. 6. Reflex tasks communicate by passing around references to capsules on a shared channel. Channels and capsules are allocated in a separate memory area managed by the Reflex run-time.

3.5.3 *Obstruction-free Communication with Atomic Methods*

Reflexes prevent synchronous operations by replacing lock-based synchronization with an obstruction-free communication scheme based on atomic methods [Manson et al. 2005]. The principle behind atomic methods is to let an ordinary Java thread invoke certain methods on the time-critical task. Once inside the atomic method, the ordinary Java thread can access the data it shares with the Reflex. These methods ensure that any memory mutations made by the ordinary Java thread to objects allocated within a Reflex's stable memory will only be visible if the atomic method runs to completion. Again, given the default allocation context, any transient objects allocated during the invocation of the atomic method will be reclaimed when the method returns. If the ordinary Java thread is preempted by the Reflex scheduler, all of the changes will be discarded and the atomic method will be scheduled for re-execution. The semantics ensures that time-critical tasks can run obstruction-free without blocking.

```
public class PacketReader extends ReflexTask {
  ...
  @atomic public void write(byte[] b) {...}
}
```

Fig. 7. Example of declaration of method on `ReflexTask` class to be invoked with transactional semantics by ordinary Java threads.

Atomic methods to be invoked by ordinary Java threads are required to be declared on a subclass of the `ReflexTask` and must be annotated `@atomic` as demonstrated with the `write()` method in Fig. 7. Methods annotated with `@atomic` are implicitly synchronized, preventing concurrent invocation of the method by multiple ordinary Java threads.

For reasons of type-safety, parameters of atomic methods are limited to types allowed in capsules, i.e. primitives and primitive array types. Return types are even more restricted, atomic method may only return primitives. This further restriction is necessary to prevent returning a transient object, which would lead to a dangling pointer, or a stable object, which would breach isolation.

3.5.4 *Communicating through Static Variables*

In addition to atomic methods, Reflexes can communicate with ordinary Java threads through static variables. However, static variables pose a particular type-safety problem as references to objects allocated in different Reflexes or on the heap could easily breach isolation. Thus, their use is restricted to primitive and reference-immutable types. Objects referenced from static variables must not be moved by the garbage collector throughout the lifetime of the Reflex graph.

3.6 Scheduling

The Reflex programming model specifies a *time triggered* scheduling policy embodied in the `Clock` task which causes connected tasks to be executed periodically. A graph must have at least one `Clock` in order to execute. The `Clock` is connected

to a `ReflexTask` by a `TimeChannel` as shown in Fig. 8. Upon firing, the `Clock` publishes a time stamp on its output time channel, causing the Reflex attached to this channel to become schedulable.

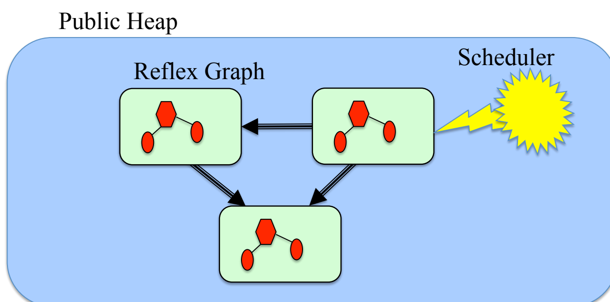

Fig. 8. Each graph is under the purview of a time triggered scheduler. Threads are not bound to tasks. At a minimum, a thread is assigned to each `Clock` but the infrastructure can choose to increase the thread count in order to improve parallelism.

Threads are not required to be mapped to tasks following a one-to-one scheme. However, as a minimum threads are assigned to `Clock` tasks that then, within the period, simply traverse as far down in the graph possible and execute all schedulable tasks, a simple scheme that makes sense on a uni-processor machine. On a uni-processor platform, executing the Reflex graph using multiple threads would not contribute to any true parallelism, but rather extend the total execution time of the graph by introducing an execution overhead of context switching between the threads. Contrary, on a multi-processor machine applying multiple threads would be beneficial to parallelism as different threads are run by multiple processors.

To ensure backward compatibility with library classes, synchronized statements and `wait/notify` are allowed. However, they are essentially treated as *no-ops* as there is at most one thread active within a task.

4. EXAMPLE: INTRUSION DETECTOR SYSTEM

To illustrate the applicability of Reflexes, we have implemented a stream processing application in the form of an Intrusion Detection System (IDS), inspired by [Sekar et al. 1999], which analyzes a stream of raw network packets and detects intrusions by pattern matching.

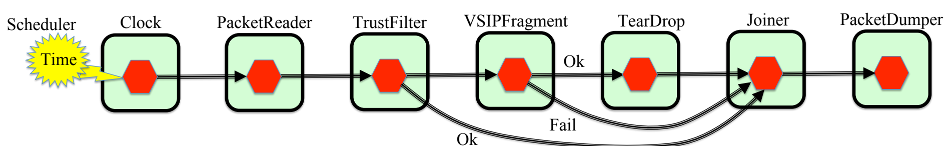

Fig. 9. Graphical representation of the Reflex graph of an Intrusion Detection System consisting of six tasks and a clock task triggered periodically by a time triggered scheduler.

Fig. 10 shows the declaration of the Reflex graph class `IDSGraph`, which instantiates and connects the tasks that combined implement the intrusion detection system. The argument to the `IDSGraph` constructor is the period in microseconds provided to the clock. Fig. 9 provides a graphical illustration of the same Reflex graph, its tasks, and how the tasks are connected.

```
public class IDSGraph extends ReflexGraph {
  private Clock clock;
  private PacketReader packetReader;
  private ReflexTask trustFilter, vsipFragment, tearDrop, joiner, packetDumper;

  public IDSGraph(int periodInMicrosecs) throws ValidationException {
    super(PRIORITY, COMMAREASIZE);
    clock = createClock(periodInMicrosecs);
    packetReader = (PacketReader) create(PacketReader.class);
    trustFilter = create(TrustFilter.class);
    ...
    packetDumper = create(PacketDumper.class);
    connect(clock, packetReader);
    connect(packetReader, trustFilter, 10);
    connect(trustFilter, vsipFragment, 10);
    connect(trustFilter, "ok", joiner, 10);
    ...
    connect(joiner, packetDumper, 10);
    validate();
  }
}
```

Fig. 10. The `IDSGraph` class extends the abstract `ReflexGraph` class, declares a constructor for setting up the graph with default priority and communication area. Note, how at the end of the constructor the `validate` method is invoked, causing the graph to be validated.

The capsules being passed around the system represent different network packets: Ethernet, IP, TCP and UDP. Object-oriented techniques are useful in the implementation as nested structures of protocol headers are modeled by inheritance. For instance, the IP capsule class (`IP_Hdr`) is a subclass of the Ethernet capsule class (`Ether_Hdr` seen in Fig. 11) with extra fields to store IP protocol information.

```
public class Ether_Hdr extends Capsule {
  ...
  public final int ETH_LEN = 6;
  final byte[] e_dst = new byte[ETH_LEN]
  final byte[] e_src = new byte[ETH_LEN];
  ...
}
```

Fig. 11. An excerpt of the `Ether_Hdr` capsule containing primitive byte arrays.

Fig. 12 shows the `PacketReader` class that creates capsules representing network packets from a raw stream of bytes. For our experiments, we simulate the network with the `Synthesizer` class. The synthesizer runs as an ordinary Java thread, and feeds the `PacketReader` task instance with a raw stream of bytes to be analyzed. Communication between the synthesizer and the `PacketReader` is done by invoking the `write` method on the `PacketReader`. This method takes a reference to a buffer of data (primitive byte array) allocated on the heap and parses it to create packets. The `write` method is annotated `@atomic` to give it transactional semantics, thereby ensuring that the task can safely preempt the synthesizer thread at any time.

```
public class PacketReader extends ReflexTask {
  private Channel out;
  private Buffer buffer = new Buffer(16384);
  private int underruns;

  public void execute() {
    TCP_Hdr p = (TCP_Hdr) makeCapsule(TCP_Hdr.class);
    if (readPacket(p) < 0) underruns++; else out.put(p);
  }

  @atomic public void write(byte[] b) {
    buffer.write(b);
  }

  private int readPacket(TCP_Hdr p) {
    try {
      buffer.startRead();
      for (int i=0; i<Ether_Hdr.ETH_LEN; i++) p.e_dst[i] = buffer.read_8();
      ...
      return buffer.commitRead();
    } catch (UnderrunEx e) { buffer.abortRead(); ... }
  }
}
```

Fig. 12. An excerpt of the `PacketReader` task that reads packets received from the ordinary Java thread and pushes them down in the graph. The `write` method, invoked by the ordinary Java thread, is declared to have transactional semantics.

The `PacketReader` buffers data in its stable memory with the `Buffer` class, shown in Fig. 13. Being referred from an instance field of the `PacketReader` task, the `Buffer` class itself is declared stable (by implementing the `Stable` interface), and in addition contains a primitive array of bytes. To satisfy the static safety constraints, we use the `StableByteArray` class to represent the primitive array within the stable class.

The reader uses the `readPacket` method to initialize capsules from the data stored in the buffer. The capsule instance itself in which to read the data is retrieved from the capsule pool through the `makeCapsule` call. The methods `startRead`, `commitRead`, and `abortRead` are used to ensure that only whole packets are read

```
public class Buffer implements Stable {
  private final StableByteArray data;
  ...
  public Buffer(int cap) {
    data = new StableByteArray(cap);
  }
  ...
}
```

Fig. 13. An excerpt of the `Buffer` class shared by the ordinary Java thread and the `PacketReader` to exchange data. Note, that the class is declared stable as it is used as an instance field on the `PacketReader` task (which inherently is stable), and that it uses the `StableByteArray` type to represent a primitive byte array.

from the buffer. They do not need synchronization since (1) potential higher priority tasks have no way to access the buffer (thanks to the isolation), and (2) ordinary Java threads, that can access the buffer through the `write` method, cannot preempt the Reflex task execution, assuming a priority-preemptive scheduling policy where the task runs at higher priorities than ordinary Java threads.

The packets first go to the `TrustFilter`, which looks for packets that match a trusted pattern; these packets will not require further analysis. Other packets are forwarded to the `VSIPFragment` task. This task detects IP fragments that are smaller than TCP headers. These are dangerous as they can be used to bypass packet-filtering firewalls. The `TearDrop` task recognizes attacks that involves IP packets that overlap.

The three tasks, `TrustFilter`, `VSIPFragment`, and `TearDrop` have a similar structure: an input channel (`in`) for incoming packets to analyze and two output channels, one for packets caught by the tasks (`ok` or `fail`), the other one for uncaught packets (`out`). These tasks also mark caught packets with meta-data that can be used in further treatment, logging or statistics. The task implementations rely on an automaton stored in stable space to recognize patterns on packet sequences that correspond to attacks.

The `Joiner` is used to transform a stream of data from multiple input tasks to a single stream of data. The last Reflex task, `PacketDumper`, gathers statistics of the whole intrusion detection process thanks to the meta-data written on packed by the previous tasks.

5. STATIC SAFETY CHECKS

Reflexes use an approach inspired by previous work on ownership type systems to statically enforce isolation, prevent dangling pointers or access to heap-allocated objects. Ownership types were first proposed in [Noble et al. 1998] as a way to control aliasing in object-oriented systems. Typically, these systems track aliasing by imposing a tree-shaped ownership structure on object graphs. Objects belonging to one particular owner can only be accessed through that owner, direct references that bypass the owner are disallowed. Most ownership type systems require fairly extensive annotations which tend to be cubersome and require invasive changes to legacy code.

In contrast, Reflexes rely on an implicit ownership type system [Zhao et al. 2008] in which no ownership parameters need to be added to variable and method declarations. Instead, the ownership is defaulted using straight-forward rules; every task encapsulates and owns all objects allocated within its private memory region. Given this ownership, the static checks ensure that references to objects owned by a Reflex are never accessed from outside, that Reflexes cannot reference heap-allocated objects (with a few exceptions), and that stable objects cannot reference transient ones. Fig. 14 illustrates legal and illegal object references.

An important property of the static safety checks is that the restrictions they enforce only apply to the time-critical parts of the application code. In other words, the legacy code interacting with Reflexes is not affected by the restrictions. One exception here is the data being shared between the time-oblivious code and Reflexes; since such data is referenced from a Reflex context it will be checked.

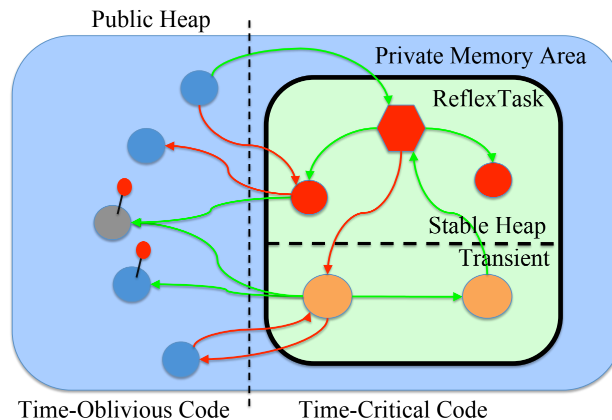

Fig. 14. The legal (green) and illegal (red) object references in and out of a Reflex task that static safety checks must ensure are respectively allowed and caught. The figure illustrates a `ReflexTask` with its stable and transient object graphs as well as a number of heap-allocated objects (blue) and static variables (black), of which some are pinned to the location on the heap.

The remainder of this section informally describes a set of rules that we argue are sufficient to ensure that neither Reflexes nor ordinary Java threads will experience dangling pointers nor observe heap-allocated objects in inconsistent states.

5.1 Partially Closed-World Assumption

A key requirement for type-checking a Reflex is that all classes used by it must be verified. To do so, we first construct a summary of classes, $\mathcal{W}$, used within a Reflex based on an approximation of the live class set. The classes in $\mathcal{W}$ are categorized in three disjoint sets: *stable*, *transient* and *capsule* classes.

The first thing the type checker has to ensure is that no class outside of $\mathcal{W}$ can be instantiated within any task in the program. 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 class in $\mathcal{W}$.

$\mathcal{R}1$: *Consider 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 checked. The checker will ensure that classes having static methods invoked from within a task belong 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 $\mathcal{W}$.

We add class whose static methods are used in Reflex to $\mathcal{W}$.

$\mathcal{R}2$: *Consider 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* $\mathcal{W}$*.* □

While $\mathcal{W}$ is clearly an over approximation of the code that will be used in a Reflex, we have not found the imprecision of the analysis to cause practical problems. Substituting a more precise analysis, such as done in [Auerbach et al. 2007], can be done without affecting the soundness of the type system.

5.2 Implicit Ownership

The key ownership property to be enforced is that objects allocated within a Reflex task are encapsulated. This means that no object allocated outside of a Reflex task may refer to a stable or transient object of that Reflex (except to the `ReflexTask` instance itself). Conversely, no stable or transient object may refer to an object allocated outside of the Reflex.

$\mathcal{R}3$: *The declaration of a non-private instance field of type* `T` *on a* `ReflexTask` *class, or subclass thereof, is only allowed if* `T` *is a primitive type.* □

Requiring that reference fields in a `ReflexTask` class to be `private`, ensures that isolation cannot be breached by accessing or updating fields of a Reflex.

$\mathcal{R}4$: *The declaration of a non-private method* `m` *on a* `ReflexTask` *class, or subclass thereof, is only allowed if* `m` *is declared* `@atomic`*.* □

The above rule ensures that interaction between Reflexes and ordinary Java thread will be non-blocking and data race free.

$\mathcal{R}5$: *Methods declared* `@atomic` *on a* `ReflexTask` *class, or subclass thereof, are restricted to declaring parameters of primitive and primitive array types, and can at most return primitive types.* □

Any method that can be invoked from ordinary Java runs the risk of leaking references through arguments or return values. Limiting signatures to primitive types is definitely safe but too restrictive. The type system allows arrays of primitive types in argument position, this does not create a breach of isolation by virtue of the treatment of arrays as transient, i.e. references that can not be retained in the Reflex.

Finally, the creation of Reflexes must be performed by the infrastructure and be initiated from outside of a Reflex. Without this rule, it would be possible for an instance of some subclass of `ReflexTask` to create another instance of the same class and breach isolation by accessing private reference fields.

$\mathcal{R}6$: *Calling* `new` *on a* `ReflexTask` *class, or subclass thereof, is illegal. Invoking methods of* `ReflexGraph` *within* $\mathcal{W}$ *is illegal.* □

Dangling pointers within the `ReflexTask` instance are prevented by segregating stable from transient references. No (long-living) stable object may acquire a reference to a (short-lived) transient object. This is done at the class granularity. If a class is declared stable, then it can only refer to other stable classes.

$\mathcal{R}7$: *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 or if* `T` *is a stable class.* □

Since the set of static safety checks tracks classes, it is critical to prevent instances of transient classes from masquerading as stable types. This is achieved by mandating that descendant of stable classes are stable.

$\mathcal{R}8$: *Assume* `C` *is a stable class in* $\mathcal{W}$*, for any class C' in* $\mathcal{W}$*. If* `C' extends C` *then* `C'` *must be stable.* □

Following from here, since the `ReflexTask` class is declared stable by implementing the `Stable` interface, any subclass thereof can only declare instance fields of primitive or stable types.

It should be noted that the above rule does not prevent a class declared stable in some $\mathcal{W}$ to have subclasses that do not respect $\mathcal{R}7$ (i.e., they are not valid stable classes as they, e.g., declare non-stable reference type fields). That is allowed as long as these are not in $\mathcal{W}$, i.e., not used from within a Reflex.

5.3 Static Reference Isolation

Enforcing encapsulation also requires `static` variables to be controlled. Without any limitations, they can be used for sharing references across encapsulation boundaries. A drastic solution would be to prevent the code in $\mathcal{W}$ from reading or writing static reference variables. Clearly this is safe, but is it too restrictive? While it may be possible for newly written code to replace static variables with objects that are threaded through constructors, the same can not be said for library classes that could be difficult to refactor. Furthermore, if one did, backwards compatibility would be lost.

The key observation here is that static variables are not dangerous if they are never modified. This suggest introducing the notion of *reference-immutable* types. These are basically types that are transitively immutable in their reference fields and mutable in their primitive fields.

$\mathcal{R}9$: *A field* `F` *in class* `C` *is effectively final if it is either (1) declared* `final` *and of reference-immutable type, or (2) declared* `private`*, of reference-immutable type, and not assigned in any non-constructor methods in class* `C` *and parent classes of* `C`*.* □

$\mathcal{R}10$: *A class* `C` *in* $\mathcal{W}$ *is reference-immutable if all non-primitive fields in the class and parent classes are effectively final.* □

Inference of which types must be immutable is based on the use of static variables.

$\mathcal{R}$11: *Let* `T` *be a class in* $\mathcal{W}$ *or a parent of a class in* $\mathcal{W}$*. A* `static` *field access expression occurring in* `T` *is legal if the field is a primitive or if the field is effectively final and it can be statically determined that it is assigned a null or a value of reference-immutable type. An assignment statement occurring in* `T` *is legal if the left-hand side of the assignment is a* `static` *field of a primitive type.* □

This last rule represents a pragmatic attempt to balance the desire for expressiveness, in particular to support the reuse of library code, with the ability to statically ensure type-safety. However, it turns out that enforcing this rule statically is non-trivial. Because of subtyping, it is not sufficient to look at the type of the declared field, but also the possible types of the values that can be assigned to the field. Thus, to declare a static field of type `T` safe, all values that can be assigned to it must be of reference-immutable type. If the type of the declared field is a final type (declared `final`) and is a reference-immutable type according to $\mathcal{R}$10, then it follows that reading from this static field is safe. For non-final types determining this property may not be possible statically.

We use an approximation on the set of live classes based on the following principles. This set of live types $\mathcal{W}$ for the static variable can be found by analyzing the class initializer, or `<clinit>`, of the class declaring it, and from here looking at all the types that are used directly or indirectly by the class initializer. To calculate this live set, we use a simple and conservative algorithm where all methods reachable from the class initializer are analyzed together with their bytecodes, thereby refraining from doing any control-flow and data-flow analysis. The algorithm is shown in Fig. 15.

function analyzeMethod(M)
set live set $S_{live} := \{\emptyset\}$
for all instructions i in method M **do**:
if (i is a `new` instruction of type T) **then** $S_{live} := S_{live} \cup \{T\}$
else if (i is an invocation of a method M' with return type T) **then**
if (T is `void`) **then** ignore, does not affect live set
if (T is declared `final`) **then** $S_{live} := S_{live} \cup \{T\}$
if (T is not declared `final`) **then** potential safety problem!
else if (i reads a `static` variable declared in type T) **then**
$S_{live} := S_{live} \cup$ {analyzeMethod(`<clinit>` method of T)}
else if (i reads an instance variable of type T)
if (T is declared `final`) **then** $S_{live} := S_{live} \cup \{T\}$
if (T is not declared `final`) **then** potential safety problem!

Fig. 15. A simple and conservative algorithm for inferring the possible value types that can be assigned to a static variable of reference-immutable type.

Having analyzed the class initializers and calculated a live set of classes from here, all classes that are type incompatible with the type of the static field being read are discarded from the live set as they can never be assigned to the field. The remaining classes in the live set are then checked for reference-immutability following the rule

in $\mathcal{R}10$, and in the event that one or more types are not reference-immutable, there is a safety problem, and the violating code statement will have to be rejected.

Finally, we assume that all static variables are either initialized eagerly before the instantiations of the Reflex graph or that the infrastructure ensure that class initializers never allocate in transient memory.

5.4 Capsules

A capsule is manipulated in a linear fashion. At any given time, the following must be enforced: there is at most one reference to the capsule from a data channel, and at most one Reflex can have references to the capsule from its stack or transient objects. With these invariants, the implementation can achieve zero copy management of capsules.

$\mathcal{R}12$: *A capsule is an instance of a subclass of* `Capsule`*, which declares only fields of primitive types and* `final` *primitive array types, and declares only* `private` *constructors.* □

The above rule is a pragmatic choice that effectively and easily ensures that capsules are reference-immutable (without permitting general reference-immutable data structures) and can only be instantiated by the Reflex infrastructure. This has two purposes: (1) it prevents creation of capsules in transient memory which could lead to dangling pointers; and (2) it ensures that all capsules are allocated off one infrastructure-controlled memory pool.

$\mathcal{R}13$: *Capsule types in $\mathcal{W}$ are transient types.* □

From the point of view of stable and transient classes, a capsule is “just” like any other transient class. Thus, we inherit the guarantee that when `execute()` method returns there will be no reference to the capsule in the state of a Reflex.

5.5 Arrays

Primitive arrays are by default transient types. Reflexes must use the set of provided array wrappers for storing primitive arrays in the stable heap, as described in Sec. 3.3.1. Array of reference types have the same allocation context as their element type. Thus any array of `T` is stable if `T` is stable and transient otherwise. So, for instance, assuming that `S` is a stable class, the statement `Object[] ms = new S[1]` is valid since `ms` is transient variable referring to a stable array.

5.6 Further Restrictions

Furthermore, the static checker restrict classes in $\mathcal{W}$ from the use of finalization as this would hamper the constant time deallocation guarantee of the transient area, thread creation as scheduling is controlled by the infrastructure, and the use of weak, soft, and phantom references. While not restricted, native code invoked from a Reflex poses problems. Likewise, the use of reflection to, e.g., load of classes, is illegal as such classes would not be statically checked. As it is not verified, native code could perform operations that impact predictability, e.g. by blocking, or memory safety, e.g., through JNI callbacks. We currently rely programmer to manually ensure that native code is fit for use in a Reflex.

6. IMPLEMENTATION

Our implementation builds on the Ovm [Armbuster et al. 2007] real-time Java virtual machine, which comes with an optimizing ahead-of-time compiler and provides an implementation of the Real-time Specification for Java. The virtual machine was designed for resource constrained uni-processor embedded devices and has been successfully deployed on a ScanEagle Unmanned Aerial Vehicle in collaboration with the Boeing Company.

We leverage the RTSJ support in Ovm to implement some of the key features of the Reflex API. For instance, the stable and transient memory areas in a Reflex are implemented by reusing the Ovm RTSJ `ScopedMemory` implementation, and the threads executing the Reflex tasks are subclasses of the standard `RealtimeThread` construct. The virtual machine configuration described here uses an optimizing ahead-of-time compiler to achieve performance competitive to commercial virtual machines [Pizlo and Vitek 2006]. Furthermore, in our implementation, we switched off memory boundary checks on the `ScopedMemory` that are normally performed by RTSJ-compliant virtual machines, as these guarantees are provided statically through our static safety checks.

6.1 Scheduling

Scheduling is implemented in the Ovm virtual machine supporting priority-preemption for real-time threads with a complete range of priorities from 1-42, the subrange 12-39 are real-time priorities used by Reflex tasks and the remaining are used for ordinary Java threads. The virtual machine's mostly-copying garbage collector is run in an ordinary Java thread.

The `ReflexTask` instances in each Reflex graph are executed by a single thread with real-time priorities according to the priority of the graph it belongs to. The thread is started as a result of an invocation of `start` on the Reflex graph, which basically causes the thread of the `Clock` task to start. Having started, upon reaching its period, the `Clock` will put the latest time stamp on channel, and traverse downstream in the graph and execute any schedulable tasks.

6.2 Memory Management

For each `ReflexTask`, the implementation allocates a fixed size continuous memory region for the Reflex's stable area and another region for its transient area. The size of each of the above is set programmatically in the Reflex API, as shown in Fig. 3. Furthermore, a buffer is set aside for the transactional log. In our prototype implementation, the size of the transaction log is growable, but not shrinkable, but the log can be reset and already allocated entries reused between transactions. However, note that the transaction log only holds mutations to stable objects. The `ReflexTask` object, the transaction log and all other implementation specific data structures are allocated in the Reflex task's stable area, and thus not subject to garbage collection.

The default allocation area for ordinary Java threads is of course the public heap. For real-time threads executing the `execute` method, this area is the transient area of the task. When an ordinary Java thread invokes a transactional method on a Reflex, the memory area has to be switched to the transient area of the task throughout the invocation, and reset once the invocation returns. To enable this,

the bytecode rewriter of the Ovm compiler has been modified to bracket all invocations of atomic methods declared on the `ReflexTask` subclass with invocations to the native `setCurrentArea/reclaimArea` methods to switch between regions. Whereas the method `setCurrentArea` changes the allocation area for the current thread, the method `reclaimArea` causes for the objects in the provided area to be reclaimed by resetting the allocation pointer to the start of the area in constant time.

The virtual machine also is responsible for redirecting the allocation of stable classes into the stable heap. For this purpose, another native method, `setAllocKind(graph, class)`, is exposed for internally identifying stable classes. This method is invoked by the Reflex run-time once for each stable class used by the tasks in the graph to be executed. The list of stable classes is provided to the Reflex run-time engine as a result of the type checking of the given Reflex graph.

Finally, the virtual machine supports allocation policies for meta-data. In particular, we rely on a policy for lock inflation ensuring that a lock is allocated in the same area as the object with which it is associated.

6.3 Atomic Methods

To implement the atomic methods, we exploit the *preemptible atomic regions* [Manson et al. 2005] facility of the Ovm virtual machine, a non-standard facility not supported by standard compliant commercial Java virtual machines. Any method annotated `@atomic` is treated specially by the Ovm compiler. More specifically, the compiler will privatize the call-graph of a transactional method, i.e., recursively generate a transactional variant of each method reachable from the transactional method. This transactionalized variant of the call-graph is invoked by the ordinary Java thread, whereas the non-transactional variant is kept around as the Reflex task might itself invoke (from the `execute()` method) some of the methods, and those should not be invoked with transactional semantics.

We have applied a subtle modification to the preemptible atomic region implementation. Rather than having a single global transaction log, a transactional log is created per `ReflexTask` instance in the graph, assuming that it declares atomic methods. This change ensures the encapsulation of each `ReflexTask` instance, and enables concurrent invocation of different atomic methods on different `ReflexTask` instances.

The preemptible atomic regions use a roll-back approach in which for each field write performed by an ordinary Java thread on a stable object within the transactional method, the virtual machine inserts an entry in the transaction log and records the original value and address of field. With this approach, a transaction abort boils down to replaying the entries in the transaction log in reverse order. Running on a uni-processor virtual machine, no conflict detection is needed. Rather, the transaction aborts are simply performed eagerly at context switches. Specifically, the transaction log is rolled back by the high-priority thread before it invokes the `execute` method of the schedulable Reflex. Whereas the complexity of transaction aborts is proportional with the number of writes performed in the transactional method at the time of preemption, starting and committing a transaction can be done in constant time. Upon resuming, the ordinary Java thread will

discover that it was preempted, and will subsequently retry the invocation of the transactional method.

6.4 Exceptions

Several exception cases need to be considered:

—If an exception occurs within the transient area of a Reflex during the invocation of the `execute()` method, we rely on standard Java semantics causing the exception object and its stack trace to be allocated in transient memory.

—If the exception propagates out of the `execute` method, the stack trace is printed and the task's computation terminates.

—If the exception occurs during an ordinary Java thread's invocation of an atomic method and the exception propagates out of the outermost atomic method, we rely on standard RTSJ behavior. The problem here is that the exception object is allocated in the transient area within the task, and thus is out of reach of the receiving Java thread allocated on the public heap. Leveraging RTSJ specific behavior, rather than receiving the specific exception object, the ordinary Java thread will receive an unchecked `ThrowBoundaryError` with a `String` based description of the actual thrown exception.

6.5 Pinning of Objects

The Ovm garbage collector supports pinning for objects such that the objects are not moved or removed during a collection, and will therefore always be in a consistent state when observed by referent objects from other memory areas, including a Reflex task. We do not pin static variables, but instead for convenience allocated them in `ImmortalMemory` area giving us the same guarantees. In contrast, arguments to atomic methods are heap-allocated objects and must be pinned when the ordinary Java thread invokes a transactional method and unpinned again when the invocation exits. We have modified the bytecode rewriter of the Ovm compiler to instrument the method bodies of the atomic methods to pin any reference type objects passed in upon entry and unpin again upon exit.

7. EVALUATION

The conducted experiments were performed using the Ovm virtual machine built with support for POSIX high resolution timers, and configured it with an interrupt rate of 1 μs, disabled the run-time memory region integrity checks (read/write barriers) and set the heap size to 512MB. Finally, non-determinism due to just-in-time (JIT) compilation is avoided through Ovm's ahead-of-time compiler. As execution platform we used an AMD Athlon 64 X2 Dual Core processor 4400+ with 2GB of physical memory running Linux with kernel version 2.6.17 extended with high resolution timer (HRT) patches and configured with a tick period of 1 μs.

7.1 Predictability

To evaluate predictability of Reflexes, we implemented a simple Reflex graph containing a single null task, scheduled it for a 45 μs period (equivalent to frequency of 22.05 KHz, a standard audio frequency), and let it execute over 10 million periods.

We also implemented a C variant of the same code, though the C variant relies on POSIX real-time extensions.

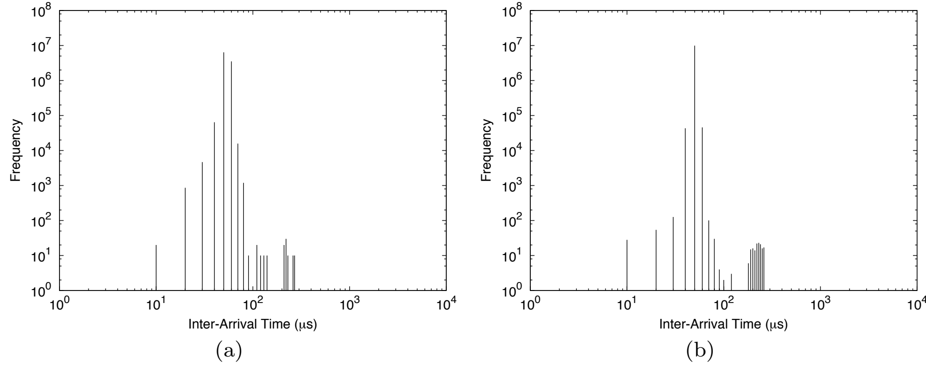

Fig. 16. Histograms of inter-arrival time for (a) Reflex graph with a null task, and (b) a C variant both scheduled for 45 μs periods. The x-axis shows the logarithm of the inter-arrival time in μs and the y-axis shows the logarithm of its frequency.

As depicted in Fig. 16 nearly all interesting observations centered around the 45 μs period, though the Reflex variant appears 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 [Spoonhower et al. 2006].

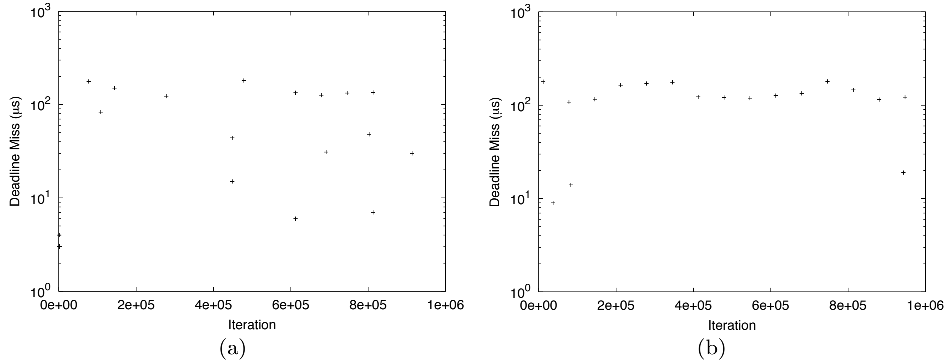

Fig. 17. Missed deadlines over time for (a) a Reflex graph with a null task, and (b) a C variant both scheduled for 45 μs periods. The x-axis shows the executions (1 million shown) of the periodic task and the y-axis shows the logarithm of the size of the deadline misses.

Fig. 17 depicts missed deadlines for both Reflex and the C variants. More precisely, with Reflexes 99.996% of the periods are completed in time compared to 99.997% for the C variant. Interestingly, Fig. 17 indicates some pattern in deadline misses around 100-200 μs for both Reflex and the C variants, though for the C

variant there seems to more consistency in that pattern. Also, it appears that both versions experience an equivalent amount of deadline misses, but 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.

7.2 Performance

We next measured the performance of Reflexes using a music synthesizer application, developed for Eventrons [Spoonhower et al. 2006], which we modified to make use of Reflexes, including a transactional method. In short, the scenario involves an ordinary Java thread that generates music samples, and writes these to a buffer on the `ReflexTask` instance through a transactional method. These samples 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.

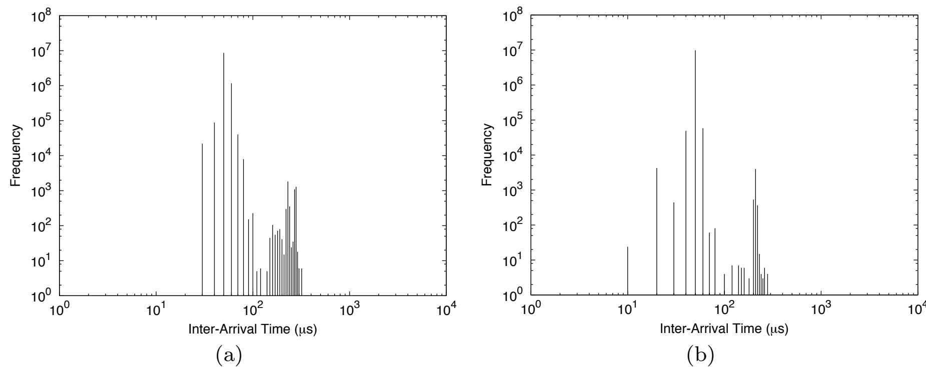

Fig. 18. Histograms of inter-arrival time for (a) a Reflex and (b) a C variant of an audio player task scheduled for 45 μs periods. The x-axis shows the inter-arrival time in μs and the y-axis shows the logarithm of its frequency.

Fig. 18 depicts the inter-arrival time of the time-critical audio player thread for both the Reflex and C variants. As already noted in Fig. 16, outlier clusters around the 200-300 μs range can also be seen in Fig. 18 for both the Reflex and its C variants. However, in Fig. 18 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)[2].

The outlier clusters seen in Fig. 18 also seem to have a direct impact on the missed deadlines as seen in Fig. 19. Specifically, for Reflexes 99.869% of the periods complete in time and do not cause deadline misses compared to 99.949% for the C variant. Of particular interest in Fig. 19 is to see how the perturbation causes regular deadline misses around 180 μs. We consider these anomalies to most

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

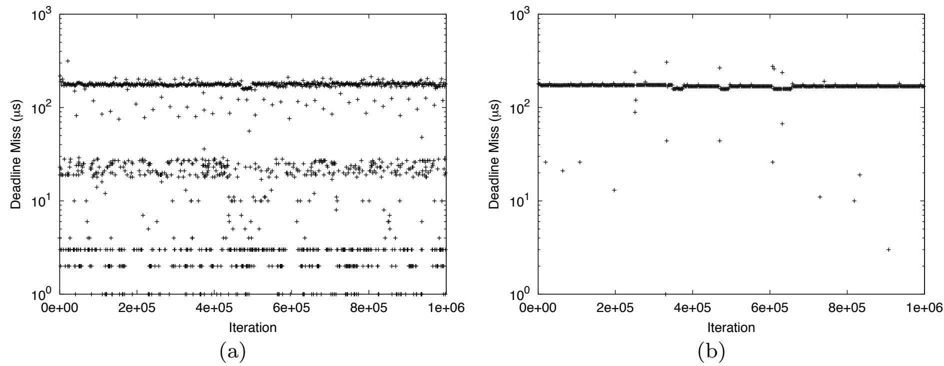

Fig. 19. Missed deadlines over time for (a) a Reflex and (b) a C variant of an audio processing task scheduled for 45 μs periods. The x-axis shows the periodic executions (1 million shown) of the time-critical task and the y-axis shows the logarithm of the size of the deadline misses.

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 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. 16 and which also appears to cause similar missed deadlines as seen in Fig. 17.

7.3 Intrusion Detection

We evaluate the Intrusion Detector System implementation, described in Sec. 4. The graph was configured with a period of 80 μs, meaning that the `PacketReader` creates capsules at a rate of 12.5 KHz. At this rate, the packet synthesizer, an ordinary Java thread, is able to generate packets in to the attack detection pipeline without experiencing underruns, i.e., at a rate which matches the rate with which the IDS can analyze them. The time used to analyze a single network packet (from the capsule creation to the end of the `TearDrop` task) varies from 3 μs to 23 μs with an average of 6 μs. One reason for this variation is that some packets are identified as a possible suspects by one of the tasks, and thus require additional processing in the automata. If we consider raw bytes instead of network packets, our IDS implementation delivers an analysis rate of 77MB per second.

7.4 Atomics on Multicore Virtual Machine

One of the limitations of the Ovm implementation is that the virtual machine is optimized for uni-processor systems. In order to validate applicability of our approach we ported much of the functionality of Reflexes to the IBM WebSphere Real-Time VM, a virtual machine with multi-processor support and a RTSJ-implementation. The implementation of atomic methods in a multiprocessor setting is significantly different. They use a roll-forward approach in which an atomic method defers all memory mutations to a local log until commit time. Having reached commit time, it is mandatory to check if the state of the Reflex has changed during the method invocation, and if so abort the atomic method. The entries in the log can safely

be discarded, in constant time, as the mutations will not be applied. If the task state did not change, the atomic method is permitted to commit its changes with the Reflex scheduler briefly locked out for a time corresponding to $\mathcal{O}(n)$, where n is the number of stable memory locations updated by the atomic method. We rely on a combination of program transformations and minimal native extensions to the VM to achieve this.

We evaluate the impact of atomic methods on predictability using a synthetic benchmark on an IBM blade server with 4 dual-core AMD Opteron 64 2.4 GHz processors and 12GB of physical memory running Linux 2.6.21.4. A Reflex task is scheduled at a period of 100 μs, and reads at each periodic execute the data available on its input buffer in circular fashion into its stable state. An ordinary Java thread runs continuously and feeds the task with data by invoking an atomic method on the task every 20 ms. To evaluate the influence of computational noise and garbage collection, another ordinary Java thread runs concurrently, continuously allocating at the rate of 2MB per second. Fig. 20 shows a histogram of the frequencies of

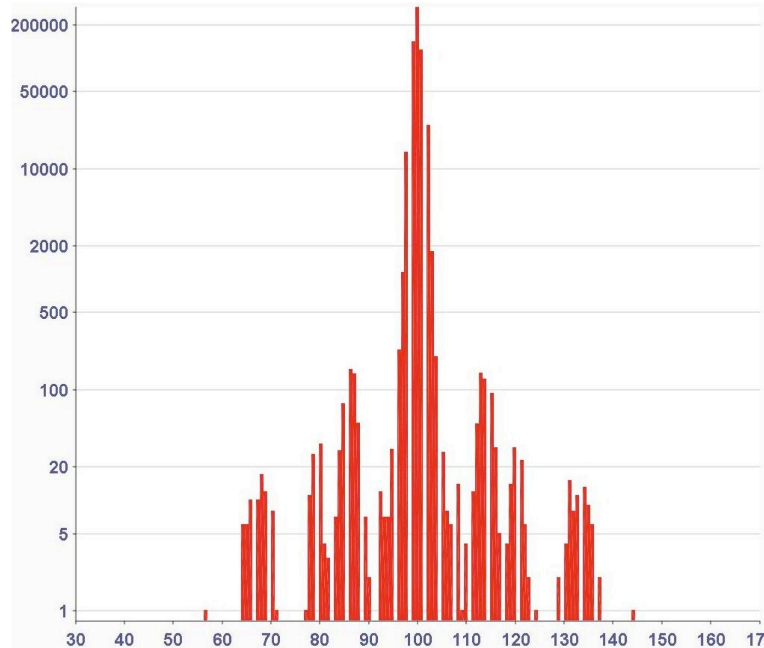

Fig. 20. Frequencies of inter-arrival times of a Reflex with a period of 100 μs continuously interrupted by an ordinary Java thread invoking an atomic method. The x-axis gives inter-arrival times in microseconds, the y-axis a logarithm of the frequency.

inter-arrival times of the Reflex. The figure contains observations covering almost 600,000 periodic executions. Out of 3,000 invocations of the atomic method, 516 of them aborted, indicating that atomic methods were exercised. As can be seen, all observations of the inter-arrival time are centered around the scheduled period of 100 μs. Overall, there are only a few microseconds of jitter. The inter-arrival times range from 57 to 144 μs.

8. CONCLUSIONS

We presented a new programming model, 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.

Source code for our Ovm implementation and examples can be found at: http://www.cs.purdue.edu/homes/jv/reflex.

Acknowledgments. We thank Jason Baker and Toni Cunei for their help with Ovm internals; Joshua Auerbach and David F. Bacon for their help with J9. This work was supported in part by NSF grants 501 1398-1086 and 501 1398-1600.

REFERENCES

Armbuster, A., Baker, J., Cunei, A., Holmes, D., Flack, C., Pizlo, F., Pla, E., Prochazka, M., and Vitek, J. 2007. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems (TECS) 7,* 1, 1–49.

Auerbach, J., Bacon, D. F., Iercan, D. T., Kirsch, C. M., Rajan, V. T., Roeck, H., and Trummer, R. 2007. Java takes flight: time-portable real-time programming with Exotasks. In Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES). *Proceedings of the 2007 LCTES conference 42,* 7, 51–62.

Auerbach, J. S., Bacon, D. F., Blainey, B., Cheng, P., Dawson, M., Fulton, M., Grove, D., Hart, D., and Stoodley, M. G. 2007. Design and implementation of a comprehensive real-time Java virtual machine. In *Proceedings of the 7th ACM & IEEE International conference on Embedded software (EMSOFT).* 249–258.

Auerbach, J. S., Bacon, D. F., Guerraoui, R., Spring, J. H., and Vitek, J. 2008. Flexible task graphs: a unified restricted thread programming model for Java. ACM, 1–11.

Bollella, G., Delsart, B., Guider, R., Lizzi, C., and Parain, F. 2005. Mackinac: Making HotSpot^{tm} real-time. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC).* 45–54.

Dawson, M. and Thwaite, P. 2008. Testing class libraries for rtsj safety. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES).* 61–67.

Gosling, J. and Bollella, G. 2000. *The Real-Time Specification for Java.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Lee, E. 2003. Overview of the Ptolemy project. Tech. Rep. UCB/ERL M03/25, EECS Department, University of California, Berkeley.

Manson, J., Baker, J., Cunei, A., Jagannathan, S., Prochazka, M., Xin, B., and Vitek, J. 2005. Preemptible atomic regions for real-time Java. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS).*

Noble, J., Vitek, J., and Potter, J. 1998. Flexible Alias Protection. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP).* Springer-Verlag, London, UK, 158–185.

Pizlo, F., Fox, J., Holmes, D., and Vitek, J. 2004. Real-time Java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC).* Vienna, Austria.

Pizlo, F. and Vitek, J. 2006. An emprical evaluation of memory management alternatives for Real-Time Java. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS).* IEEE Computer Society, Washington, DC, USA, 35–46.

Pizlo, F. and Vitek, J. 2008. Memory management for real-time java: State of the art. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC)*. Orlando, FL.

Sekar, R., Guang, Y., Verma, S., and Shanbhag, T. 1999. A high-performance network intrusion detection system. In *ACM Conference on Computer and Communications Security*. 8–17.

Spoonhower, D., Auerbach, J., Bacon, D. F., Cheng, P., and Grove, D. 2006. Eventrons: a safe programming construct for high-frequency hard real-time applications. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). *SIGPLAN Not. 41,* 6, 283–294.

Spring, J., Pizlo, F., Guerraoui, R., and Vitek, J. 2007. Reflexes: Abstractions for highly responsive systems. In *Proceedings of the 3rd International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*.

Spring, J., Privat, J., Guerraoui, R., and Vitek, J. 2007. StreamFlex: High-throughput stream programming in Java. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA)*.

Spring, J. H. 2008. Reflexes: programming abstractions for highly responsive computing in Java. Ph.D. thesis, Ecole Polytechnique Fédérale de Lausanne (EPFL).

Thies, W., Karczmarek, M., and Amarasinghe, S. 2002. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC)*.

Zhao, T., Baker, J., Hunt, J., Noble, J., and Vitek, J. 2008. Implicit ownership types for memory management. *Science of Computer Programming 71,* 3, 213–241.