High-Performance Transactional Event Processing

Antonio Cunei[1], Rachid Guerraoui[1], Jesper Honig Spring[1], Jean Privat[2], Jan Vitek[3]

Ecole Polytechnique Fédérale de Lausanne (EPFL)[1], [2]Université du Québec à Montréal (UQÀM) and Purdue University[2]

Abstract. This paper presents a transactional framework for low-latency, high-performance, concurrent event processing in Java. At the heart of our framework lies Reflexes, a restricted programming model for highly responsive systems. A Reflex task is an event processor that can run at a higher priority and preempt any other Java thread, including the garbage collector. It runs in an obstruction-free manner with time-oblivious code. We extend Reflexes with a publish/subscribe communication system, itself based on an optimistic transactional event processing scheme, that provides efficient coordination between time-critical, low-latency tasks.We report on the comparison with a commercial JVM, and show that it is possible for tasks to achieve 50 μs response times with way less than 1% of the executions failing to meet their deadlines.

1 Introduction

Performing real-time processing in a managed language environment, such as Java, is very appealing but introduces two significant implementation challenges: memory management and inter-task communication.

Typically, garbage collectors used in commercial Java virtual machines are designed to maximize the performance of applications at the expense of predictability. Consequently, with these garbage collectors it is non-deterministic *when* and for *how long* they will run. As a consequence garbage collection introduces execution interference that can easily reach hundreds of milliseconds, preventing the timeliness requirements of the real-time systems from being satisfied. High performance real-time Java virtual machines have somewhat reduced this challenge through advances in real-time garbage collection algorithms, reducing the latency to approximately 1 ms. However, some applications have latency/throughput real-time requirements that cannot be met by current real-time garbage collection technology. For these applications, having scheduling latency requirements below a millisecond, any interference from the virtual machine is likely to result in deadline misses.

Another source of interference that can easily cause deadline misses, relates to communication between the time-critical real-time tasks, including any interaction they might have with the rest of the application. Typically, time-critical tasks only account for a fraction of the code of an entire application, the rest

being either soft- or non-real-time code. For instance, the US Navy's DD-1000 Zumwalt class destroyer is rumored to have million lines of code in its shipboard computing system, of which only small parts have real-time constraints. Typical programming practices for sharing data would involve synchronizing access to the data. In a real-time system, this might lead to unbounded blocking of the real-time thread, so-called *priority inversion*, causing serious deadlines infringements.

One of the key design decisions of the Real-time Specification for Java (RTSJ) [9] was to address these problems with a programming model that restricts expressiveness to avoid unwanted interactions with the virtual machine and the garbage collector in particular. The RTSJ introduced the NoHeapRealtimeThread for this purpose, and also proposed solutions to cope with priority inversion. As we discuss in the related work, however, experience implementing [5, 13, 21, 2] and using [8, 20, 7, 22, 24] the RTSJ revealed a number of serious deficiencies. More recently, alternatives to NoHeapRealtimeThread have been proposed, such as Eventrons [26] and Exotasks [3] from IBM Research as well as Reflexes [27] and StreamFlex [28].

This work builds on our experience with Reflexes [27], a simple, statically type-safe programming model that makes it easy write and integrate simple periodic tasks observing real-time timing constraints in the sub-millisecond range, into larger time-oblivious Java applications. Reflex tasks are written in a subset of Java with special features for (1) safe region-based memory management preventing interference from the garbage collector, (2) obstruction-free atomic regions avoiding any priority inversion problems when communicating with time-oblivious code, and (3) real-time preemptive scheduling allowing the Reflex task to preempt any lower-priority Java thread, including the garbage collector. Finally, Reflexes rely on a set of safety checks, based on our previous work for Real-time Java [1, 32], to ensure safety of memory operations. These checks are enforced statically by an extension of the standard Java compiler. The Reflex safe regions provide better latency than a real-time collector.

In StreamFlex [28], we extended Reflexes to support low-latency stream processing by introducing graphs of tasks that communicate through non-blocking transactional communication channels, allowing tasks to communicate in a zero-copy fashion. While these transactional channels are effective for communication, they fall short when it comes to coordination between time-critical tasks. In particular, coordinating transactions in a multi-core environment turns out to be challenging when striving for low-latency.

Publish/subscribe systems are a special case of event-based programming where a number of computational components are allowed to register, or subscribe, to events published by other components in the system [15]. This programming model has been applied in different context; in distributed systems, publish/subscribe is a convenient way to decouple producers from consumers, and to provide a simple resource discovery protocol via a registration mechanism. On a single node, publish/subscribe offers a convenient way to program dynamic systems where new rules can be added/removed dynamically and events

processed in parallel. Example of applications can be found in the financial sector where events are the movement of stocks and computational elements implement trading rules. Some examples of event-based systems are Gryphon, JEDI and JavaSpaces [16, 29, 14].

This paper presents an extension of the original Reflex programming model with a publish/subscribe substrate that allows for coordination and communication between highly time-critical, low-latency Reflex tasks by registering for, and publishing, user-defined events. This publish/subscribe system is itself built on top of a transactional tuple space implementation that abides by the semantics described in [17] and uses the data structures described in [30]. While the original Reflex implementation used a limited form of software transactional memory based on [18] for the obstruction-free interaction with ordinary Java threads, in the extension presented in this paper, all the computation performed by a Reflex task is transactional. Thus, access to the shared space containing events and subscriptions is transactional, as are the actions performed during event processing.

Furthermore, the paper reports on a number of encouraging performance results by comparing equivalent executions with a commercial JVM, and documents on the ability for tasks to achieve 50 μs response times with way less than 1% of the executions failing to meet their deadlines.

This paper focuses on the extended programming model and its performance characteristics. We explicitly do not address issues of distribution, fault-tolerance, event correlation and event lifetimes. While these are important from a usability point of view, we leave their investigation to future work.

2 Programming with Events and Reflexes

Reflexes are small time-critical tasks that are intended to execute free from interference from their environment. The task is an object of a user-defined subclass of ReflexTask with its own private memory area and that is executed by a real-time thread. The main responsibility of a Reflex task is to implement execute(), a method that will be invoked whenever the Reflex task's trigger condition evaluates to true. In this paper we extend the notion of time-triggered tasks from Reflexes [27] with registration-triggered Reflex tasks. A purely time-triggered Reflex task is one whose execute() method is executed according to a period specified at task instantiation. A registration-triggered Reflex task is released by the scheduler when an event that matches one of the Reflex task's registered templates is inserted in the shared space (by another Reflex task).

Fig. 1 illustrates a Reflex application consisting of a time-oblivious part and three time-critical tasks. A single transactional event space is shared by the three Reflex tasks (R1, R2, R3). The tasks can register for events, take events and write events to the shared space. Standard, time-oblivious Java code can run in the same virtual machine but has restricted ability to interact with Reflex tasks, see [27] for details.

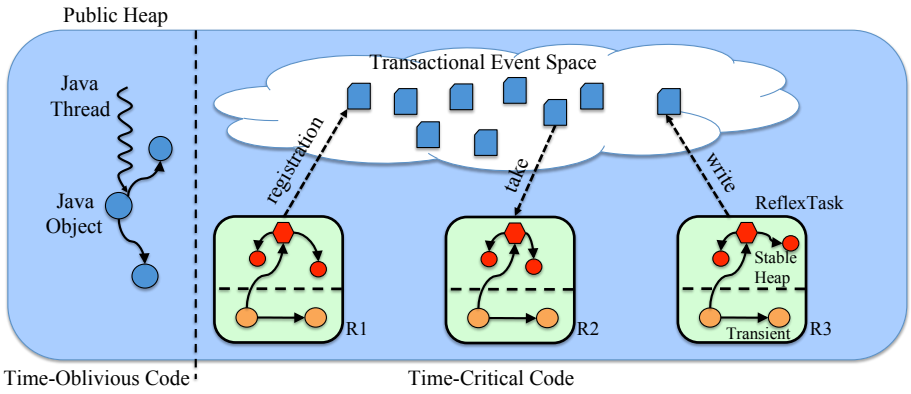

Fig. 1. Transactional Event Processing with Reflexes. Three Reflex tasks (R1, R2, R3) are concurrently using a shared event space. Reflex tasks can register for events, in which case they get to execute as soon as an event matching their registered template is put in the space (R1). Reflex task can take events from the space (R2) or put new events into it (R3). Reflex tasks execute in private memory areas and are unreachable to the public heap garbage collector. They are composed of two parts, a stable heap that is not reclaimed, and transient storage that is reclaimed after each invocation of the task.

In order to minimize latency, the Reflex programming model sports a bimodal distribution of object lifetimes for tasks. An object allocated within a Reflex task can be either *stable*, in which case the lifetime of the object is equal to that of the task, or it can be *transient* in which case the object only lives for the duration of the invocation of the task's execute method after which the virtual machine will reclaim it in constant time before the next invocation of the task. Specifying whether an object is stable or transient is done at the class level. By default, data allocated by a Reflex thread is transient, while only objects of classes implementing the Stable marker interface will persist between invocations. Stable objects must be managed carefully by the programmer, as the size of the stable heap is fixed and the area is not garbage collected. The distinction between stable and transient objects is enforced statically by a set of static safety checks [27]. It is noteworthy that code running in transient does not require special annotations, and we can thus reuse many classes from the standard Java libraries.

Example. Fig. 2 presents a simple Reflex task. The class StockBuyer is declared to extend the abstract class ReflexTask. As such it has to implement the method execute(), which runs every time the task is scheduled. The state of the Reflex task consists of two fields, maxPrice, a double, and handle, a reference to a Handle. Being instance fields of the Reflex task, these fields persist across invocations of the task, they are the roots for its stable data. Primitive types are stable by default, and object are stable only if their defining class implement the Stable

```
class StockBuyer extends ReflexTask {
  double maxPrice;
  Handle handle;

  public void initialize() {
    handle = subscribe(new Event("sell", null));
  }

  public void execute() {
    for (Event offer : handle) {
      if (isExpired(offer.get("expiry")) return;
      double price = asDouble(offer.get("price"));
      if (price * 2 <= maxPrice)
        write(new Event("type", "buy")
          .add("price", price));
      else
        if (price > maxPrice) maxPrice = price;
    }
  }
}
```

Fig. 2. Stock Trading with Reflexes.

marker interface (as is the case for Handle). The role of a Handle is to represent a subscription that has been registered with the event space. In this case, handle will be notified if an event with the key sell is inserted in the space.

The execute() method is invoked after an event matching the subscription has been inserted in the space. By the time the Reflex task starts to execute, more matching events may have become available, or someone else may have been quicker than the Reflex task and no matching event may be present in the space any longer. Thus, execute() will iterate over the events that match the query and, if the price is right, it will write buy the events back to the space.

The transactional infrastructure must keep track of two kinds of events: the operations on the shared event space, and the mutations of the stable state of the Reflex task. Each iteration of the loop in execute() performs (1) a test to see if more data is available, then (2) it does a destructive read, and (3) in some cases a write into the space. The transactional infrastructure will record all of these operations, and manage conflict detection and rollback. On the side of the Reflex task, the only meaningful operation is the possible update to the field maxPrice, which will be recorded in case a rollback is necessary.

It is useful to consider the potential sources of aborts. The Reflex programming model is such that tasks 'own' all the data in both stable and transient state. Thus, all of the objects that make up a Reflex task are guaranteed to be accessed only by a single thread. This means that there can be no conflict on task data. Conflicts (and aborts) can come about in only two ways: concurrent

operations on the shared event space, and explicit calls to abort() from a Reflex task itself.

3 Transactional Reflex API and Semantics

We present here the semantics of the shared event space (Sec. 3.1), and then explain how the shared space integrates with transactional Reflex tasks (Sec. 3.2).

3.1 Shared Event Space

An event space is a multiset of events that are shared between Reflex tasks. An event is a function from keys – represented as interned String objects – to values – Java objects such as boxed primitives, arrays, and certain user-defined data structures. The basic operations on a shared space are limited to three basic, non-blocking, operations: take(), write(), and test(), which respectively, remove and insert a deep-copied version of the provided event, and check for the availability of an event in the shared space. The arguments to all those methods are events; in the case of take() and test(), the argument is used as a template for finding matching events in the space.

The semantics of matching is the following: a template matches an event if it contains the same or fewer keys, and for each key it contains a value that is either the same of the event or null, null being a wildcard. Fig. 3 depicts a few examples on matching between templates and events.

Template		Event
['stock':'APPL', 'value':3]	matches	['stock':'APPL', 'value':3]
['stock':null, 'value':3]	matches	['stock':'APPL', 'value':3]
['stock':'APPL']	matches	['stock':'APPL', 'value':3]
['stock':'APPL', 'value':3]	!match	['stock':'APPL']

Fig. 3. Matching between templates and events.

Transactional semantics of the shared space follow from [17]. Informally, a sequence of operations performed within a transaction is conflict-free if the same sequence would succeed at the time of commit.

3.2 Transactional Reflexes

An excerpt of the extended Reflex API supporting transactional event processing is given in Fig. 4. To implement a Reflex task, the programmer must provide a subclass of ReflexTask class. The operations available include two versions of the shared space operations, e.g. write() and writeNow(). The latter bypasses the transactional layer and updates the space directly. This is a form of open

```
public abstract class ReflexTask implements Stable {
  public ReexTask(int transientSize, int stableSize) {...}
  public abstract void execute();
  public void initialize() {};

  final void write(Event ev) {...};
  final void writeNow(Event ev) {...};
  final Event take(Event template) {...};
  final Event takeNow(Event template) {...};
  final boolean test(Event template) {...};
  final boolean testNow(Event template) {...};
  final Handle subscribe(Event template) {...};
  final void unsubscribe(Handle hndl) {...};
  final void abort() {...};
}

public final class Handle implements Stable {
  Event next();
  Event nextNow();
  boolean hasNext();
  boolean hasNextNow();
}

public class Event {
  Event();
  Event add(String key, Object value);
  Object get(String key);
}
```

Fig. 4. An excerpt of the extended Reflex API.

nesting [19], and is needed in combination with user-initiated abort() to post an event describing the reasons of the abort or containing partial results.

The semantics of abort() is to discard all changes to the shared space, terminate the current invocation of the execute() method, discard all data allocated in the transient area and finally rollback all changes in the stable heap.

In order to be notified of the insertion of an event matching some template, a Reflex task must subscribe() to that event. When it does, it receives an instance of Handle which is always allocated in stable heap of the task (and thus can be retained between invocation of execute()). Each handle refers to one subscription in the shared event space. A Reflex task may have multiple handles listening on different kinds of events. When an event is inserted in the space all Reflex tasks with matching subscription will be notified. Handles support an iterator interface to query and read matching events.

Reflex tasks can be active or passive. An active Reflex has an associated real-time thread with a priority and a period. The semantics of an active Reflex is

that every period, the implementation checks if one of the handles has witnessed insertion of a matching event. If so, the Reflex's execute() method is invoked. A Reflex with a period of 0 does not sleep between invocations of execute(). If an active Reflex has no subscriptions, the execute() method is invoked every period. A passive Reflex is run without timeliness guarantees by threads taken from a thread pool.

3.3 Legacy Reflex Communication Schemes

Besides allowing Reflex tasks to communicate and coordinate through the shared event space, as described so far, the original Reflex model also allows for tasks to communicate through static variables and for ordinary Java threads to communicate with Reflex tasks.

Communication with ordinary Java threads has to be managed carefully to avoid introducing execution interferences that could cause the Reflex task to miss its deadlines. Typical programming practices for sharing data between threads involve lock-based synchronization. In a real-time system this might lead to *priority inversion* and serious deadline misses. To encounter this, Reflexes propose a scheme based on a limited form of transactional memory in the form of obstruction-free transactional methods ensuring that the Reflex task meets its temporal requirements.

In enforcing isolation of a Reflex task, static variables pose a particular type-safety problem as references to objects allocated in different tasks or on the heap, could easily pass the isolation boundaries. To circumvent this, Reflexes restrict the use of static variables to primitive and *reference- immutable* types. Informally speaking, an object of reference-immutable type provides access to an object graph connected by references that cannot change but containing other fields that can change, i.e., primitive types.

4 Static Safety Issues

To avoid interference from the public heap garbage collector, the Reflex programming model relies on strict isolation between: (1) the Reflex tasks themselves, and (2) the Reflex tasks and the time-oblivious Java code in which the Reflex tasks have been integrated and might or might not interact with. The goal of the safety checks is to statically guarantee this isolation by restricting unsafe code that would violate the memory integrity and allow access to heap-allocated objects in inconsistent states, and dangling pointers to be observed. Note, the scope of the restrictions enforced only apply to the time-critical parts of the Reflex application, including any data shared between the Reflex tasks and time-oblivious Java code; any legacy Java code is not subject to these restrictions. The details of the simple set of restrictions that we apply to ensure this isolation are described in [27]. Fig. 5 illustrates the valid and invalid references that are XX

In terms of function, events described here behave much like capsules used in StreamFlex [28] in that they are used as units of communication between tasks.

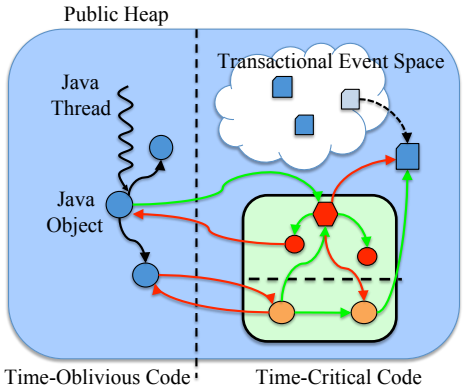

Fig. 5. Reflex Isolation. Enforcing Reflex task isolation through static safety checks that prevent illegal (red) references while permitting valid (green) references.

Likewise, as objects they both impose similar safety risk in that references could leak between tasks and break isolation through these objects. In StreamFlex these risks are addressed by letting capsules be treated specially (i.e., they are neither transient nor stable types), and any capsule instances are allocated from a special fixed size pool. This prevents the StreamFlex task from retaining a reference to the capsule once the execute method has completed. Furthermore, capsules are restricted in the types of fields that they can carry, allowing only fields of primitive or reference-immutable types. Together, the restrictions effectively prevents isolation from being violated and at the same time allow for zero-copy communication between tasks resulting in fast throughput, crucial for stream-processing applications, as demonstrated with StreamFlex [28].

In the extension described in this paper, the goal is not throughput of data processing but rather efficient and flexible coordination between time-critical, low-latency Reflex tasks. With this goal in mind, zero-copy communication is not strictly necessary, although desirable, from a performance point of view. Consequently, unlike capsules, events are not restricted in what object types they can carry. However, to ensure type safety by preventing references from leaking, when inserted into the shared space, or taken from here, the event objects (and the entire object graph they hold) are recursively deep-copied between the memory contexts of the task performing the operation and the shared event space. Furthermore, to ensure that no tasks retain a reference to an event, which could only happen if the Event class were to be declared stable, the Event class is treated as an ordinary transient type. This means that any event instances will only survive for the duration of the execute method (unlike the deep-copied event that following completion of the execute method will be present in the shared space). If events were treated as stable objects, any event ever used by the Reflex task throughout its lifetime would be allocated in its stable heap, with the probably risk of eventually running out of memory. Since, from the static safety

checks of Reflexes, stable types are prohibited from referencing transient ones, an event also cannot be assigned to a field on a Reflex task (as the ReflexTask is declared stable, see Fig. 4). With the treatment of events as normal transient types, and the deep-copying of events into the shared space, no additional static safety checks have to be defined than those specified in [27].

5 Implementation Highlights

Reflexes have been implemented on top of the Ovm [4] 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 (RTSJ). 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 [2]. We leveraged the real-time support in the VM to implement some of the key features of the API. The virtual machine configuration here described uses the ahead-of-time compiler to achieve performance competitive with commercial VMs [24].

We outline some of the key implementation issues of Reflexes; a more detailed description appears in [27]. Reflex tasks are run by real-time threads scheduled by a priority-preemptive scheduler. For each Reflex task instance, the implementation allocates a fixed size continuous memory region for the stable heap and another region for its transient area. The ReflexTask object, its thread, and all other implementation specific data structures are allocated in the Reflex task's stable heap. These regions have the key property that they are not garbage collected. We are in the process of investigating using hierarchical real-time garbage collector, described in [23], to garbage collect the stable heap. This collector can collect partitions of the heap independently and, due to the special structure of the event space, we expect to have compaction with pause time bounds less than 100 microseconds. Each thread in the VM has a default allocation area. This area is the heap for ordinary Java threads and the respective transient area for all real-time threads executing Reflex tasks. The garbage collector supports pinning for objects. Pinned objects are guaranteed not to move during a garbage collection. Thus they can safely be accessed from a Reflex. The allocation policy for classes and static initializers ensures that all objects allocated at initialization time are pinned.

Our implementation also relies on a simplified version of the RTSJ region API to ensure that sub-millisecond deadlines can be met. We depart from the RTSJ by our use of static safety checks in order to ensure memory safety. That has the major advantage of avoiding the brittleness of RTSJ applications, and also brings performance benefits as we do not have to implement run-time checks to prevent dangling pointers. The details of the static safety checks being enforced can be found in [27].

The event space uses an event-tree data structure for fast access based on the fingerprinting scheme described in [30]. Registrations are maintained by a reverse

```
ReflexSupport.setCurrentArea(transientArea);

while (true) {
  waitForNextPeriod();
  if ((subscriptions.size() > 0) &&
      (!subscriptions.hasMatch())) continue;

  delta = new Transaction(space);
  reflex.startLogging();
  try {
    execute();
  }
  catch (Abort a) {
    reflex.undo();
    ReflexSupport.reclaimArea(transientArea);
    continue;
  }
  if (delta.validate(space)) {
    delta.commit(space);
    reflex.commit();
  } else {
    reflex.undo();
    delta.abort();
  }
  ReflexSupport.reclaimArea(transientArea);
}
```

Fig. 6. Time-triggered Reflex (pseudo code).

event-tree that takes advantage of the duality between templates and events. The empty template (an event with no fields) is not allowed in a registration.

Fig. 6 shows pseudo code for the implementation of a time-triggered periodic Reflex task. The implementation of transactions is done at two levels:

- *Event space transactions* are managed by interposing a Delta between each Reflex task and the space. The Delta records all operations and will try to publish the changes when the execute() method returns. In Fig. 6, event space transaction code is related to the delta object.
- *Reflex-level transactions* are implemented by logging all memory mutations in the stable heap (as described in [18]) – transient objects can be ignored because they will be discarded when the execute() method returns. The log is used to undo the operations performed during an invocation of the execute() method. As there can be only one thread executing within a Reflex task and no other thread may observe the internals of a task (this is ensured by the static safety checks [27]), memory operations can be performed on the main

memory while retaining strong atomicity. In Fig. 6, Reflex-level transactions code is related to the ReflexTask object.

6 Performance Evaluation

We conducted a number of empirical experiments to evaluate the performance and behavior of the proposed system. All experiments were performed on an AMD Athlon 64 X2 Dual Core processor 4400+ with 2GB of physical memory running Linux 2.6.17 extended with high resolution timer (HRT) patches [25] configured with a tick period of 1 μs. We used an Ovm build with support for POSIX high resolution timers, and configured it with an interrupt rate of 1 μs. In addition, we disabled the run-time checks of violations of memory region integrity (read/write barriers), and configured it with a heap size of 512MB. The version of the HotSpot client JVM used in our benchmarks is 1.5.0_09.

6.1 Throughput

We developed some micro-benchmarks to test the raw performance of our system.

Empty A single Reflex task that just increments a counter. No operations are performed on the event space.

Solo A single Reflex task subscribes and takes events it sends to itself. A take() and a write() are performed during each execute().

Duo The version of 'solo' with two Reflex tasks. Each one subscribes and takes events it sends to the other one. A take() and a write() are performed during each execute().

Max Before starting the Reflex task, the event space is filled with events containing integers. The body of the execute() of the Reflex task takes two integer events from the space and re-writes the one having the largest value back into the space.

Table 1 shows the performance of these micro-benchmarks.

Benchmark	Time (ms)	Operations/ms
Empty	688	—
Solo	4 553	439,000
Duo	5 775	346,000
Max	6 463	464,000

Table 1. Execution time for 1 million iterations, and number of take/write operations per second.

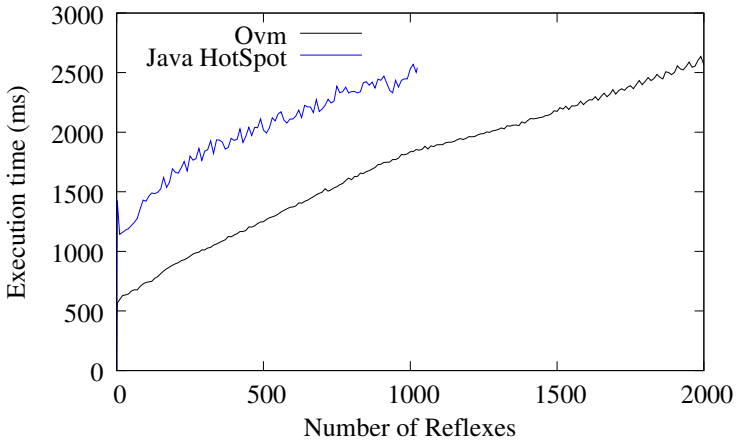

Fig. 7. Execution time as a function of the number of Reflex tasks in the hand-off chain. The x-axis is the number of executing Reflex tasks, the y-axis is the time in microseconds for one event to traverse the entire chain.

6.2 Scalability

How does a system based on Reflexes scale in the presence of an increasing number of event processing elements? From a software engineering point of view, it is advantageous to represent different business rules with different Reflexes as they execute independently and can be added/removed at any time. But is it feasible to have hundreds of Reflex tasks in the same JVM? The overheads come from the memory regions and thread associated to Reflexes.

We set up a benchmark which implements a chained hand-off between Reflex tasks. Each Reflex task in the benchmark is given a unique identifier, it takes an event with its id and writes back a copy of the event with the identity of the next Reflex task. As we increase the number of tasks, the chain gets longer.

Fig. 7 shows the time it takes for an event to travel down the chain. We compare numbers for HotSpot and Ovm from one to 2,000 threads. As expected, the execution time increases with the number of Reflexes. In term of numbers of Reflex tasks, Ovm is limited only by the available memory. Thus, we were able to run with 2,000 Reflex tasks, while HotSpot fails with a Java exception if we try to create more than 1,024 threads. Interestingly, in the comparable range Reflex tasks appear more efficient on Ovm than on HotSpot. The worst case for Ovm is 1,849 microseconds while it is 2,541 for HotSpot (thus making HotSpot 27% slower for one thousand threads).

6.3 Predictability of Event Processing

Predictability is important in applications which require very low-latency responses to events. There are two challenges for a JVM here: scheduling threads

periodically and preemption of non-critical threads. In this benchmark we demonstrate that Reflexes can be scheduled with sub-millisecond accuracy without interference from other concurrently running threads. To establish a worst case scenario, we consider a Reflex with a 50 μs period that performs a take() followed by a write() of the same event. Concurrently, a low-priority thread performs reads and writes to the space in a tight loop.

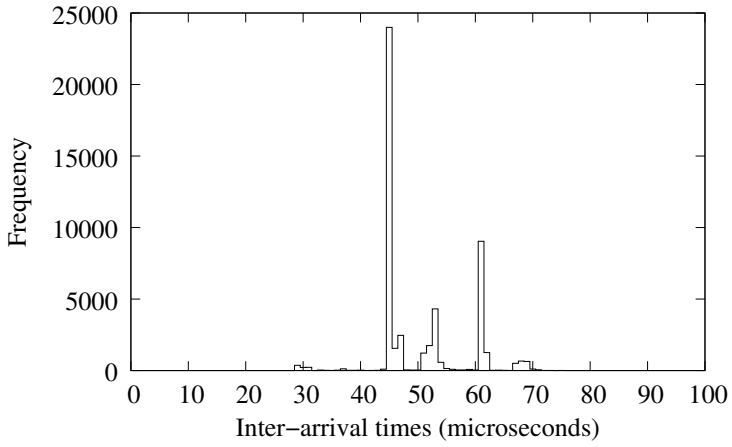

Fig. 8. Frequency of inter-arrival time for a Reflex periodically scheduled with a period of 50 μs when executed (over 50,000 iterations) concurrently with a noise maker thread. The x-axis is the inter-arrival time of two consecutive executions in microseconds. The y-axis is a frequency.

Fig. 11 shows the time between two invocations of the execute() method in a Reflex. The results are clearly concentrated around the request period with remarkably few outliers.

Fig. 9 show that when a Reflex misses a deadline the order of magnitude is usually less than 50 μs. Interestingly, Fig. 9 also shows a few extreme deadline misses going as high as around 1,500 μs (not shown). We believe these to be bugs in the implementation. In terms of precision, out of 50,000 iterations only 18 periods were missed, which correspond to a deadline miss rate of 0.03%.

Fig. 10 shows the inter-arrival and processing time of the Reflex from stock trade example when executed on Ovm with a period of 80 μs. Specifically, the Reflex is responsible for generating real-time stock offers in a constant flow and writing them to the event space. As can be seen from the figure, the processing time of the stock seller Reflex lies constantly around 10 μs throughout the shown execution period. Likewise, the inter-arrival time represents a stable level of predictability – centered around the scheduled 80 μs period, and with very little variation. In fact, in our experiments covering 100,000 periodic executions, we only found 65 deadline misses (equivalent to a 99.93% met deadlines).

6.4 Reflexes on Multi-core 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 transactions 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, 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 transaction 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 heap locations updated. We rely on a combination of program transformations and minimal native extensions to the VM to achieve this.

We evaluate the impact of transactions 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 transaction on the task every 20 ms. To evaluate the influence of computational

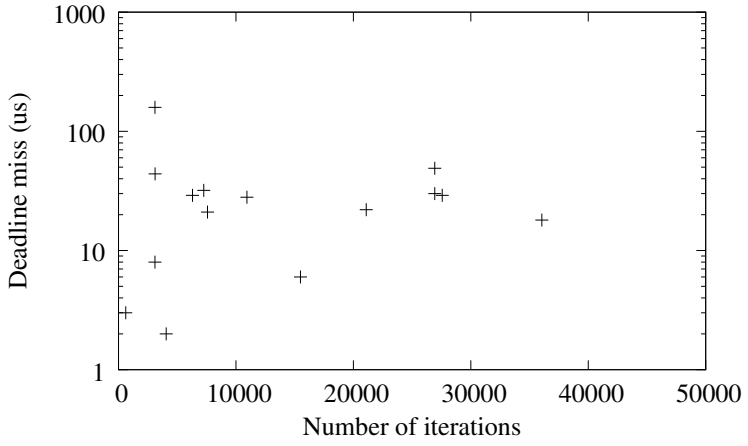

Fig. 9. Deadline misses over time for a Reflex periodically scheduled with a period of 50 μs when executed concurrently with a noise maker thread. The x-axis depicts the periodic executions over time whereas the y-axis depicts the logarithm of deadline misses in microseconds.

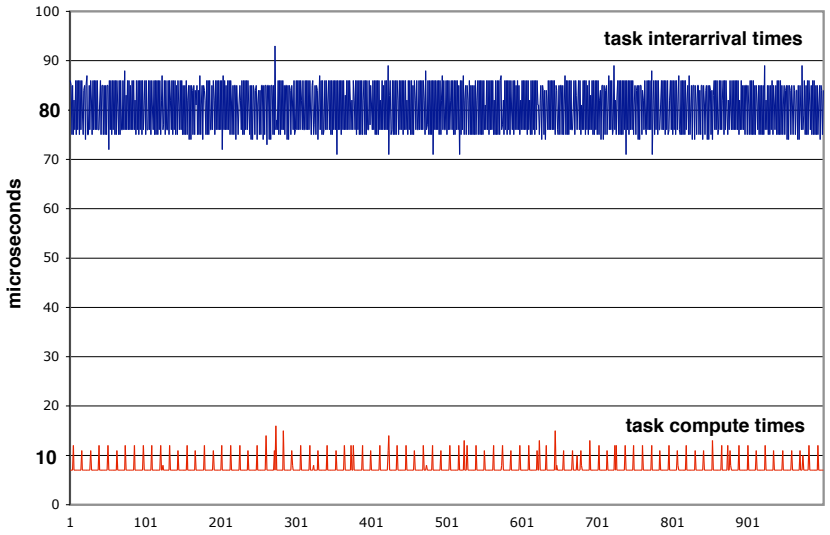

Fig. 10. Inter-arrival and processing time over time for a Reflex implementation of a stock seller scheduled with a period of 80 μs. The x-axis shows the periodic executions (only 1,000 shown) and the y-axis shows the inter-arrival and processing time (both in μs).

noise and garbage collection, another ordinary Java thread runs concurrently, continuously allocating at the rate of 2MB per second.

Fig. 11 shows a histogram of the frequencies of 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.

7 Related Work

The approach presented in this paper is closely related to independent work carried out at IBM Research, namely the Eventron [26] and Exotask [3] real-time programming models. 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 or modify the value of reference variables. Furthermore, they are prevented, by load-time checks, from reading mutable reference variables. The stringent re-

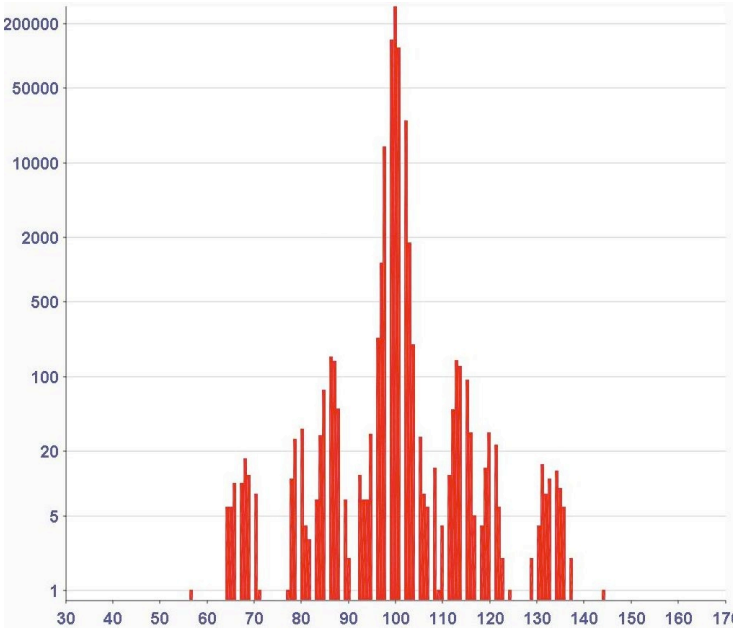

Fig. 11. Frequencies of inter-arrival times of a Reflex with a period of 100 μs continuously interrupted by an ordinary Java thread. The x-axis gives inter-arrival times in microseconds, the y-axis a logarithm of the frequency.

striction make it safe for an Eventron task to preempt the garbage collector or any other virtual machine service, and thus make it possible to run with periods in the microseconds. Reflexes have similar responsiveness but are less restrictive due to our combination of regions and ownership types.

Exotasks extend Eventrons on a number of accounts. Most importantly, Exotasks are organized in a graph connected by non-blocking point to point communication channels. As the task are isolated, the collection is task-local and can usually be carried out in very little time. Tasks communicate by exchanging messages by deep-copy, whereas StreamFlex adopts the zero-copy communication of [28]. Whereas the message exchange used in the extension presented in this paper also is based on deep-copy, it is not limited to point-to-point communication as the shared event space also facilitates one to many communication.

An interesting question is what advantages these programming models bring compared to RTSJ's NoHeapRealtimeThread which is, after all, supported by all RT JVMs. Experience implementing [5, 13, 21, 2] and using [8, 20, 7, 22, 24] the RTSJ revealed a number of serious deficiencies. In the RTSJ, interference from the garbage collection is avoided by allocating data needed by time critical real-time tasks from a part of the virtual machine's memory that is not subject to garbage collection, dynamically checked regions known as *scoped memory areas*. Individual objects allocated in a scoped memory area cannot be deallocated;

instead, an entire area is torn down as soon as all threads exit it. Dynamically enforced safety rules check that a memory scope with a longer lifetime does not hold a reference to an object allocated in a memory scope with a shorter lifetime and that a NoHeapRealtimeThread does not attempt to dereference a pointer into the garbage collected heap.

Another issue with RTSJ is that, due to a lack of isolation, it is possible for a NoHeapRealtimeThread to block on a lock held by a plain Java task. If this ever occurs, all bets are off in term of real-time guarantees as the blocking time cannot be bounded. Finally, dynamic memory access checks entail a loss of compositionality. Components may work just fine when tested independently, but break when put in a particular scoped memory context. This is because for a RTSJ program to be correct, developers must deal with an added dimension: *where* a particular datum was allocated. Design patterns and idioms for programming effectively with scoped memory have been proposed [22, 6, 8], but anecdotal evidence suggests that programmers have a hard time dealing with NoHeapRealtimeThreads and that resulting programs are brittle.

The static safety checks used by Reflexes to guard against memory error, presented in [27], is an extension of the implicit ownership type system of [31], the latest in a line of research that emphasizes lightweight type systems for region-based memory [1, 32]. Ownership is *implicit* because, unlike e.g. [12, 10, 11], no ownership parameters are needed. Instead, ownership is assumed by default by using straightforward rules.

8 Conclusion

We have presented a new transactional framework in the context of event-based programming that builds on our previous work on real-time systems. We extend Reflexes, a restricted programming model for highly responsive systems, with a shared event space, that is accessed with transactional semantics, and transactionalize the execution of the Reflexes that operate on the shared space. The resulting model ensures strong atomicity and very low performance overheads.

Our evaluation is encouraging in terms of performance, scalability and predictability when comparing to equivalent executions on a commercial JVM. Also, we have shown that it is possible to achieve 50 μs response times with way less than 1% of the executions failing to meet their deadlines.

In the future, we plan to enrich the programming model with a language for expressing complex, and temporal, patterns of events. We also plan to integrate events with the stream processing paradigm that was explored in [28]. In terms of implementation, we intend to design a customized real-time garbage collector to manage the event space and to integrate the multi-processor extensions that are currently being added to our research infrastructure.

References

1. C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped Types and Aspects for Real-Time Java. In *Proceedings of the European Conference on Object-*

Oriented Programming (ECOOP), pages 124–147, Nantes, France, July 2006.
2. A. Armbuster, J. Baker, A. Cunei, D. Holmes, C. Flack, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems (TECS)*, 7(1):1–49, 2007.
3. J. Auerbach, D. F. Bacon, D. T. Iercan, C. M. Kirsch, V. T. Rajan, H. Roeck, and R. Trummer. 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)*, volume 42, pages 51–62, New York, NY, USA, 2007. ACM.
4. J. Baker, A. Cunei, C. Flack, F. Pizlo, M. Prochazka, J. Vitek, A. Armbruster, E. Pla, and D. Holmes. A real-time java virtual machine for avionics - an experience report. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 384–396, Washington, DC, USA, 2006. IEEE Computer Society.
5. W. S. Beebee and M. C. Rinard. An implementation of scoped memory for real-time java. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, pages 289–305, London, UK, 2001. Springer-Verlag.
6. E. G. Benowitz and A. Niessner. A patterns catalog for RTSJ software designs. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*, pages 497–507, 2003.
7. E. G. Benowitz and A. F. Niessner. Experiences in adopting real-time java for flight-like software. In *Proceedings of the International workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, pages 490–496, 2003.
8. G. Bollella, T. Canham, V. Carson, V. Champlin, D. Dvorak, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reinholtz. Programming with non-heap memory in the Real-time specification for Java. In *Companion of the 18th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 361–369, 2003.
9. G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.
10. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th Annual ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA)*, November 2002.
11. C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2003.
12. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, pages 48–64. ACM, Oct. 1998.
13. A. Corsaro and R. Cytron. Efficient memory reference checks for Real-time Java. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2003.
14. G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions Software Engineering*, 27(9):827–850, 2001.
15. P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

16. E. Freeman, S. Hüpfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, Reading, MA, 1999.
17. S. Jagannathan and J. Vitek. Optimistic concurrency semantics for transactions in coordination languages. In *Proceedings of the International Conference on Coordination Models and Languages (COORDINATION)*, Pisa, Italy, March 2004.
18. J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek. Preemptible atomic regions for real-time Java. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2005.
19. Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP*, pages 68–78, 2007.
20. A. F. Niessner and E. G. Benowitz. Rtsj memory areas and their affects on the performance of a flight-like attitude control system. In *Proceedings of the International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, pages 508–519, 2003.
21. K. Palacz and J. Vitek. Java subtype tests in real-time. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 378–404, Darmstadt, Germany, July 2003.
22. F. Pizlo, J. Fox, D. Holmes, and J. Vitek. Real-time Java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC)*, Vienna, Austria, May 2004.
23. F. Pizlo, A. Hosking, and J. Vitek. Hiearchical real-time garbage collection. In *Proceeedings of ACM SIGPLAN/SIGBED 2007 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 123–133, 2007.
24. F. Pizlo and J. Vitek. An empirical evalutation of memory management alternatives for Real-time Java. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2006.
25. H. Resolution Timers. `www.tglx.de/projects/hrtimers/2.6.17/`.
26. D. Spoonhower, J. Auerbach, D. F. Bacon, P. Cheng, and D. Grove. 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)*, volume 41, pages 283–294, New York, NY, USA, 2006. ACM.
27. J. Spring, F. Pizlo, R. Guerraoui, and J. Vitek. Reflexes: Abstractions for highly responsive systems. In *Proceedings of the 3rd International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, 2007.
28. J. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: High-throughput stream programming in Java. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA)*, 2007.
29. R. E. Strom, G. Banavar, T. D. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. C. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. *CoRR*, cs.DC/9810019, 1998.
30. J. Vitek, C. Bryce, and M. Oriol. Coordinating processes with secure spaces. *Science of Computer Programming*, 46(1-2):163–193, 2003.
31. T. Zhao, J. Baker, J. Hunt, J. Noble, and J. Vitek. Implicit ownership types for memory management. *Science of Computer Programming*, 71(3):213–241, 2008.
32. T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, 2004.