Chapter 1
Atomicity in Real-time Computing

Jan Vitek
Purdue University

Abstract Writing safe and correct concurrent programs is a notoriously error-prone and difficult task. In real-time computing, the difficulties are aggravated by stringent responsiveness requirements. This paper reports on three experimental language features that aim to provide atomicity while bounding latency. The context for our experiments is the real-time extension of the Java programming language.

1.1 Introduction

Adding concurrency to any software system is challenging. It entails a paradigm shift away from sequential reasoning. Programmers must reason in terms of the possible interleavings of operations performed by the different threads of their program. This shift affects all aspects of the software lifecycle. During development, synchronization commands must be added to prevent data races. During verification and analysis, more powerful techniques must deployed to validate code. During testing, code coverage metrics must be revisited. Lastly, debugging becomes more complex as bugs become harder to reproduce. Aiming to simplify the task of writing correct concurrent algorithms, Herlihy and Moss proposed the idea of *transactional memory*, an alternative to lock-based mutual exclusion [8] that ensures *atomicity* and *isolation*. Atomicity means that either all of a given transaction's updates will be visible or none will. Isolation means that a transaction's data will not be observed in an intermediate state. While transactional memory has been studied extensively for general purpose computing, real-time systems have their own set of constraints. In particular, providing bounds on the execution time of any code fragment is key to being to ensure that a real-time task will meet its deadline and that a set of tasks is schedulable.

The tension between concurrency and real-time has been studied extensively. Practitioners are trained to keep critical sections short and to enforce strict programming protocols to avoid timing hazards. This requires a thorough understanding of the program's control flow. Unfortunately, the global view of the program required by these approaches does not mesh with modern software engineering practices which emphasize component-based systems and code reuse. The data abstraction and information hiding principles that are key to reusability in object-oriented programming tend to obscure the control flow of programs. Consider the Java code fragment:

```
synchronized ( obj ) {   obj.f = tgt.get();   }
```

Ensuring that the critical section is short requires non-local knowledge. The programmer must be able to tell which method is invoked in the call to `get()`, this in turn depends on the class of the object referenced by variable `tgt`. The question whether acquiring a lock on `obj` is sufficient is tricky too. It depends on which shared memory location are accessed by `get()`. Real-time scheduling theory refers to the time spent in a critical section as *blocking time*. This is the time a, possibly higher-priority, thread may have to wait until it can acquire the lock on `obj` and execute. For predictability the blocking time has to be bounded (and preferably small). A transactional memory equivalent of the above code, with appropriate language support, would be:

```
atomic {   obj.f = tgt.get();   }
```

The difference with the lock-based code is that it is not necessary for the programmer to specify a lock. Instead, it is up to the implementation to ensure atomicity and isolation of all of the operations performed by the critical section. Much like in a database, if two atomic sections attempt to perform conflicting changes to the memory, one of them will be aborted, its changes undone, and rescheduled for execution. The software engineering benefits of transactional memory are striking, especially in an object-oriented setting where it is often difficult to know which locks should be acquired to protect a particular sequence of memory operations and method invocations. The main disadvantage lies in the need to reexcute aborted transactions which entails maintaining a log of memory updates. Unless one can provide a hard bound on the number of aborts and on all of the implementation overheads, the approach may not be of use in a real-time context.

The notion of adding transactional facilities to a programming can be traced back all the way to Lomet [13]. As mentioned above, Herlihy and Moss introduced the concept of software transactional memory [8, 19], and were among the many to provide implementations for Java and other languages [7, 6]. Bershad worked on short atomic sections [4, 3] for performing compare-and-swaps without hardware support. Anderson et al. [1] described lock free objects for real-time systems, but did not explore compiler support.

Welc et al. investigated the interaction of preemption and transactions on a multi-processor [22], but did not provide any real-time guarantees. Finally, Rigneburg and Grossman have used similar techniques to the ones developed here to add transactions to the Caml language on uniprocessors [16].

1.2 Preemptible Atomic Regions for Uniprocessor Systems

The first concurrency control abstraction we will review is the *preemptible atomic region* (PAR) introduced by Manson, Baker, Cunei, Jagannathan, Prochazka, Xin and Vitek in [14]. This work was done within the Ovm [2] project that implemented the first open source virtual machine for the Real-Time Specification for Java (RTSJ) [5].

PARs are a restricted form of software transactional memory that provide an alternative to mutual exclusion monitors for uniprocessor priority-preemptive real-time systems. A PAR consists of a sequence of instructions guaranteed to execute atomically. If a task is executing in a PAR and a higher-priority task is released by the scheduler, then the higher-priority task will get to execute right away. It will preempt the lower priority task and in the process, abort all memory operations performed within the atomic region. Once the lower-priority task is scheduled again, the PAR is transparently re-executed. The advantage of this approach is that high-priority tasks get to execute quickly, no special priority inversion avoidance protocol is needed, while atomicity and isolation of atomic regions is enforced.

Preemptible atomic regions differ from most transactional memory implementations in that tasks are aborted eagerly. This has the advantage forgoing conflict detection, but, on the other hand, results in more roll-backs than strictly necessary. Conflict detection is work that has to be performed on every read or write, while preemption happens relatively unfrequently. We believe that this was the right tradeoff.

We contrast lock-based code and PARs with an example, simplified code taken from the Zen real-time ORB [11]. Figure 1.1(a) makes extensive use of synchronization. Method `exec()` is synchronized to protect `ThreadPoolLane` instances against concurrent access. The lock on line 3 protects the shared `queue`. The `Queue` object relies on a private lock (line 6 and 8) to protect itself from missuse by client code. Atomic regions are declared by annotating a method as `@PAR`; they are active for the dynamic scope of the method, so all methods invoked by a method declared `@PAR` are transitively atomic. In Figure 1.1.b, we use two atomic sections: one for the `exec()` method (10) and another for the `enqueue()` method (14). The first PAR is sufficient to prevent all data races within `exec()`; it is therefore unnecessary to obtain a lock on the queue. If `enqueue()` were only called from `exec()`, it would not need to be declared atomic (but declaring atomic does not hurt, as nested

PARs are treated as a single atomic region). This solution is simpler to understand, as it does not rely on multiple locking granularities. A single PAR will protect all objects accessed within the dynamic extent of the annotated method. Contrast this with the lock-based solution, where all potentially exposed objects must be locked. Furthermore, the order of lock acquisition is critical to prevent deadlocks. On the other hand, PARs cannot deadlock: they do not block waiting for each other to finish.

The PAR-based mechanism avoids costs found in typical locking protocols. When a contended lock is acquired, one or more allocations may need to be performed. Additionally, whenever a lock is acquired or released, several locking queues need to be maintained; these determine who is "next in line" for the lock. In contrast, a PAR entrance only needs to store a bookkeeping pointer to the current thread. When a PAR exits, the only overhead is the reset of the log; this consists of a single change to a pointer. Lock-based implementations also tend to have greater context-switching overhead. Consider the code in Figure 1.1.a with three threads: `t1`, `t2` and a higher-priority thread `t3`. Thread `t1` can acquire the lock on `sync` and be preempted by Thread `t2`, which then synchronizes on `queue`. Now, assume that Thread `t3` attempts to execute `exec()`. This scenario can result in five context switches. The first one occurs when `t3` preempts `t2`. The second and third occur when the system switches back to `t2` so that it can release the lock on `queue`. Finally, the fourth and fifth switches occur when the system schedules `t1` so that it can release the lock on `sync`. Under the same conditions, the use of PARs only requires one context switch. If `t2` preempts `t1` while it is in

```
class ThreadPoolLane {

  synchronized void exec(Req t){
     if (borrow(t)) {
        synchronized(queue) {
           queue.enqueue(t);
           buffered++;
        }
     ...

class Queue {

  final Object sync=new Object();

  void enqueue(Object d) {
     QueueNode n=getNode();
     n.value=d;
     synchronized(sync) {
        // enqueue the object
     ...
```

(a) With Monitors.

```
class ThreadPoolLane {

   @PAR void exec(Req t){
     if (borrow(t)) {
        queue.enqueue(t);
        buffered++;
     }
   ...

class Queue {

   @PAR void enqueue(Object d) {
     QueueNode n=getNode();
     n.value=d;
     // enqueue the object
   ...
```

(b) With Preemptible Atomic Regions.

Fig. 1.1 Example: A `ThreadPoolLane` from the Zen ORB. (Simplified)

an atomic section, then `t1` will be aborted, and any changes it might have made will be undone. When `t3` is scheduled, it needs only undo the changes performed by `t2` to make progress. This does not require a context switch, as `t3` has access to the log. It is worth pointing out that roll-backs are never preempted.

PAR-based mechanisms incur two major costs that lock-based implementations do not. First, all writes to memory involve a log operation that records the current contents of the location being written. Second, if another thread preempts a thread that is executing a PAR, all changes performed by that thread will have to be undone; the heap will be restored based on the values stored in the log. Therefore, whenever writes are sparse, the overheads for a lock-based solution will be higher than those of the PAR-based solution. In our experience, aborts are cheap, because critical sections typically perform few writes.

1.2.1 Real-time Guarantees

To ensure that a set of real-time tasks meet their deadlines, one must conduct a response time analysis. The theory for dealing with mutual exclusion has been developed and is well understood. We now consider how to plug in PARs into schedulability equations. Assume a set of n periodic tasks scheduled according to the *rate monotonic scheme* [9], which is widely used scheduling technique for fixed priority preemptive real-time systems. Tasks share a set of locks $\ell_1 \dots \ell_k$. At most one task can be executing at any instant. Each task τ_i performs a job J_i. A job has period p_i such that $\forall i < n, p_i < p_{i+1}$. There is one critical section per job, and the critical section always ends before the job finishes. For each job, W_i is the maximal execution time spent in a critical section and U_i is the maximal time needed to perform an undo. R_i is the worst case response time of a job J_i. C_i is the worst-case execution time of job J_i. Tasks with higher priority π than τ_i are $hp(i) = \{j \mid \pi_j > \pi_i\}$, and ones with lower priority are $lp(i) = \{j \mid \pi_j < \pi_i\}$. Given that a task τ_i suffers *interference* from higher priority tasks and *blocking* from lower priority tasks, the response time is computed as $R_i = C_i + B_i + I_i$, where I_i is the maximum *interference time* and B_i the maximum *blocking factor* that J_i can experience [10]. The schedulability theorem is the following.

Theorem 1. *A set of n periodic tasks $\tau_i, 0 \leq i < n$ is schedulable in RM, iff*

$$\forall i \leq n, \exists R_i : R_i \leq p_i$$

$$R_i = C_i + \max_{j \in lp(i)} U_j + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{p_j} \right\rceil (C_j + U_i + W_i)$$

The intuition behind Theorem 1 is as follows. The expression $\max_{j\in lp(i)} U_j$ represents the worst case delay caused by rolling back a critical section executed by any task with priority lower than τ_i. The worst case interference of J_i with higher priority tasks, plus extra execution time needed to reexecute some critical sections, are computed as follows. Given that $\left\lceil \frac{R_i}{p_j} \right\rceil$ is the maximal number of releases of a higher priority task τ_j that can interfere with a task τ_i, we can compute the number of releases of τ_j in J_i as $\sum_{j\in hp(i)} \left\lceil \frac{R_i}{p_j} \right\rceil$. The most pessimistic approximation of how many rollbacks can occur is to assume that every interference implies a rollback of a critical section in J_i. Hence, every time a higher priority task τ_j preempts J_i, C_j is the worst case execution time of τ_j during which J_i is preempted and thus not progressing, and U_i+W_i is the worst case time necessary to undo and reexecute the critical section of J_i preempted. As a critical section is undone by the higher priority task, $\sum_{j\in hp(i)} \left\lceil \frac{R_i}{p_j} \right\rceil U_i$ is a part of J_i's interference with higher priority tasks, while $\sum_{j\in hp(i)} \left\lceil \frac{R_i}{p_j} \right\rceil W_i$ is an extra execution time. The worst case for undo times is $U_i = W_i$; this occurs if all operations within a PAR are memory writes.

Let us compare PAR with the original priority inheritance protocol (PIP) by Sha *et al.* [18].

Theorem 2 (PIP Schedulability). *A set of n periodic tasks $\tau_i, 0 \le i < n$ is schedulable in RM, iff*

$$\forall i \le n, \exists R_i : R_i \le p_i$$

$$R_i = C_i + m_i \max_{j\in lp(i)} W_j + \sum_{j\in hp(i)} \left\lceil \frac{R_i}{p_j} \right\rceil C_j$$

where m_i is the number of critical sections in J_i.

In PIP, the worst case delay of a job J_i caused by lower priority tasks sharing locks with J_i is proportional to the number of critical sections in J_i. Moreover, the worst case delay to enter to a single critical section is bounded by the worst case execution time of the conflicting critical section. In PAR, the worst case delay to enter a single critical section is bounded by the number of updates to be undone. In addition, the overall delay of J_i caused by lower priority tasks is not dependent on the number of critical sections in J_i. The cost of PAR is that the overall throughput is reduced due to the cost of undoing and reexecuting lower priority threads. While in all the priority inheritance schemes the cost of sharing locks degrades response times of higher priority threads through their blocking factor, in PAR the cost is paid by the lower priority threads in their higher execution time and interference.

In Java the number of threads (n) tends to be small, on the other hand the number of critical sections (m) is typically large. Assuming $U_i < W_i$, the trade-off between the PIP and PAR formulas is a question of comparing $m_i W_j$ with $2 \sum_{j \in hp(i)} \left\lceil \frac{R_i}{p_j} \right\rceil W_i$.

1.2.2 Implementation sketch

As seen in Figure 1.1, PARs can be declared by annotating a method `@PAR`. In our implementation, atomic methods are aborted every time a higher priority thread is released. This approach reduces blocking time, when a high-priority thread is released it only blocks for as long as it takes to abort one atomic region, and we only need to maintain a single undo log. Atomic regions introduce several costs. A single system wide log is preallocated with a user defined size (default of 10KB). When control enters a PAR, it is necessary to store a reference to the current thread. Within the PAR, each time the application writes to memory, two additional writes are issued to the log: the original value of the location, and the location itself. The commit cost is limited to resetting the pointer into the log. The cost of undoing consists of traversing the log in reverse, which has the effect of undoing all writes performed within the critical section, and then throwing an internal exception to rewind the stack. For example, consider a program with two zero-initialized variables. If the instructions `x=1; y=1; x=2` were executed, the log would contain `(addressOf(x):0, addressOf(y):0, addressOf(x):1)`. If an abort took place, there would be a write of 1 to `x`, then a write of 0 to `y`, and finally a write of 0 to `x`; both variables would then contain their initial values. The runtime cost of an abort is thus $O(n)$, where n is the number of writes performed by the transaction.

In traditional transactional systems, a conflict manager is required to deal with issues such as deadlock and starvation prevention. PARs are not subject to these limitations. Conflict detection is only required when a thread is ready to be released by the scheduler. Assume the scheduler is invoked to switch from the currently executing thread `t1` to a new higher priority thread `t2`. First, the scheduler checks the status of `t1`. If it is in an atomic region, the scheduler releases `t2`, which then executes the `abort` operation. If `t1` is in an atomic region that is already in the process of aborting, the abort must complete before thread `t2` is released. In either case, the pending `AbortedFault` will be thrown when thread `t1` is scheduled again.

The implementation proceeds by bytecode rewriting. We transform any method `f()` with the PAR annotation into a new method named `f$()`. A new `f()` method, as seen in Figure 1.2, is added to the class; all of the original calls to `f()` will invoke this method instead of the original method. A transaction starts with an invocation of `start()`; this method enters a PAR and begins the logging process. The logged version of the original method is then executed. Upon successful completion, `commit()` resets the log. This method is enclosed in a `finally` clause to ensure that the transaction com-

```
void f() {
   RETRY: try {
      try { PAR.start(); f$(); }
      finally { PAR.commit(); }
   } catch (AbortedFault _) { goto RETRY; }
}
```

Fig. 1.2 Code transformation for a method `@PAR void f`. The body of the original method is moved into a new synthetic method named `f$`.

mits even if the method throws a Java exception. To deal with the consequences of an abort, we provide the class `AbortedFault`. When an abort occurs, the `AbortedFault` is thrown by the virtual machine. This exception class is treated specially by the virtual machine in that it can not be caught by normal catch clauses and sidesteps user defined `finally` clauses.

Rewriting extends down into the virtual machine; when a call is made that requires VM assistance, the invoked virtual machine code is rewritten as a PAR. Some calls can be rolled back. There are, effectively, four different types of code that can be encountered in the Ovm runtime: (a) Calls to Java methods that can be easily rolled back. This includes calls to allocators and garbage collectors. (b) Calls to Native methods that cannot be rolled back. This includes calls to system timers or to I/O methods. (c) Calls to Native methods that can easily be replaced with Java implementations. The `System.arraycopy` method, for example, calls the C function `memcpy`. `memcpy` is a native call, and so cannot be logged; however, it is trivial to write a Java implementation of this method. (d) Calls to Native methods that do not mutate system state. These methods can be handled individually, as well. Operations that must not be undone are called *non-retractable* operations. These include calls to I/O methods, as well as calls to user level data structures that are not specific to the thread currently running (such as timers or event counters), which must not be reset, as they are logically unrelated to the transaction. When encountered in a PAR, calls to such methods cause a compiler warning and are transformed to unconditionally throw an exceptions.

1.2.3 Experimental evaluation

We evaluated the response times of high-priority threads with a program that executes a low and a high priority thread which access the same data structure, a `HashMap` from the `java.util` package. The low priority thread continually executes critical sections that perform a fixed number of read, insert and delete operations on the `HashMap`. Periodically, the high-priority thread executes a similar number of operations. In one configuration, the accesses are protected by the default RTSJ priority inheritance lock implementation.

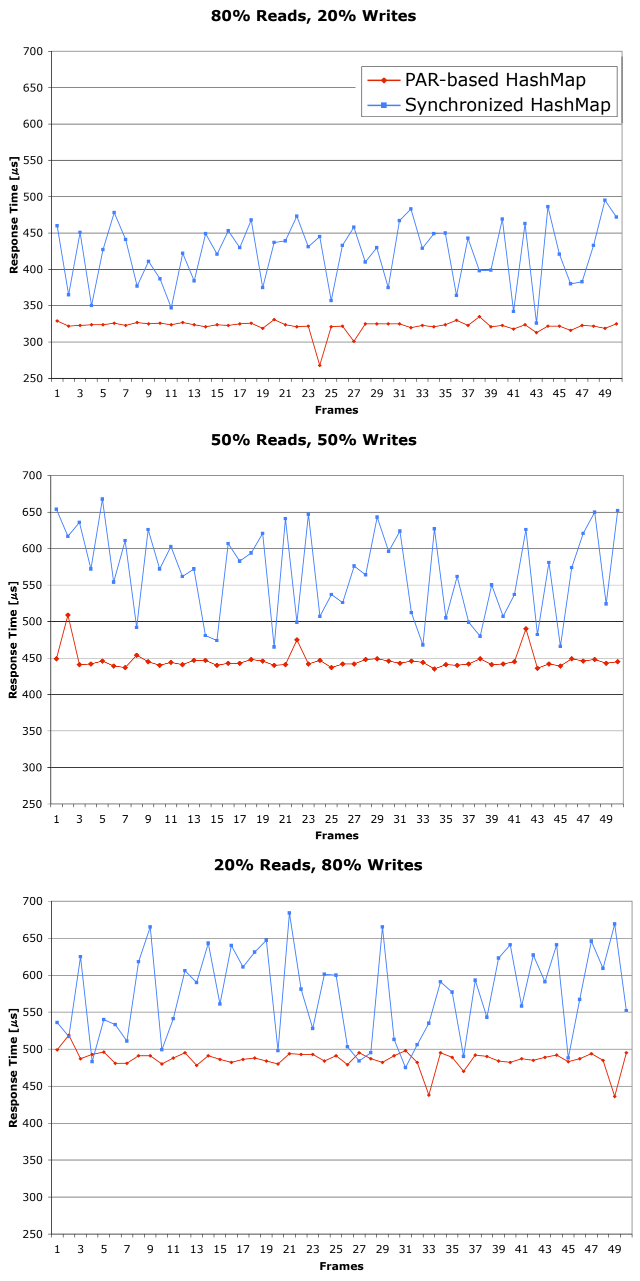

Fig. 1.3 Response time of a high-priority thread in the HashMap Microbenchmark. The x-axis indicates the number of periods (or frames) that have elapsed, and the y-axis indicates the response time of the high-priority thread (in microseconds).

In the other, the accesses are protected by a PAR. For a PAR-based `HashMap`, this produced a high likelihood of aborts. In fact, an abort occurred every time a high-priority thread is scheduled. These measurements were obtained with Ovm running on a 300Mhz Embedded Planet PowerPC 8260 board with 256MB SDRAM, 32 MB Flash, and Embedded Linux. Figure 1.3 shows the results of the test. Two points are noteworthy. First, the latency for the PAR-based HashMap is lower; this indicates that undoing the low priority thread's writes was faster than context switching to the other thread, finishing its critical section, and context switching back. Second, the response time of the PAR-based HashMap was more predictable; this is because it was not necessary to execute a indeterminately long critical section before executing the high-priority thread's PAR.

1.3 Obstruction-free Communication with Atomic Methods

The second abstraction we have experimented with is *Atomic Methods* in the Reflex programming model [20] (this work was done with Jesper Honig Spring, Filip Pizlo, Jean Privat and Rachid Guerraoui). Atomic methods were introduced in the *Reflex* programming model for mixing highly-responsive tasks with timing-oblivious Java programs. A Reflex program consists of a graph of Reflex tasks 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 [12], that are used to design real-time control systems, and to stream-based programming languages like StreamIt [21]. A Reflex graph is constructed as a Java program,

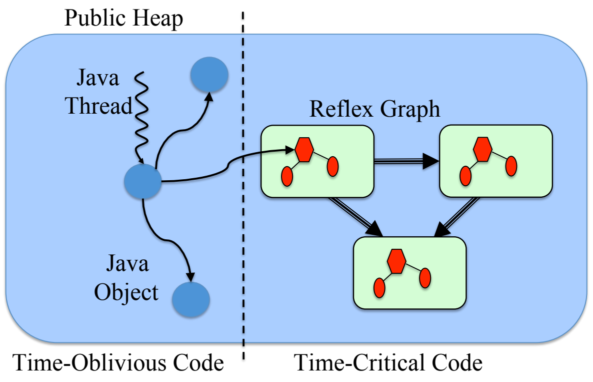

Fig. 1.4 Illustration of a Java application consisting of time-oblivious code (blue) and a time-critical Reflex graph with three connected tasks.

following standard Java programming conventions. Reflexes can run in isolation or as part of a larger Java application. To communicate with ordinary Java threads, Reflex provides special methods which will ensure that real-time activities do not block for normal Java threads and ensures atomicity of changes performed by plain Java. Figure 1.4 illustrates a Reflex program.

A Reflex acts as the basic computational unit in the graph, consisting of user-defined persistent data structures, typed input and output channels for communication between Reflexes, 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 the Reflex programming model, Reflexes have access to disjoint memory partitions and developers can choose between a real-time garabage collector and region-based allocation for memory managment within Reflexes.

1.3.1 Atomic Methods

Reflexes prevent synchronous operations by replacing lock-based synchronization with an obstruction-free communication. 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.

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 Figure 1.5. 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 (capsules are the special message objects that can be sent across channels between Reflexes), i.e. primitives and primitive array

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

Fig. 1.5 Example of declaration of method on `ReflexTask` class to be invoked with transactional semantics by ordinary Java threads.

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. If an atomic methods need to return more than a single a primitive, it can only do it by side-effecting an argument (i.e. an array).

1.3.2 Example

Figure 1.6 shows the `PacketReader` class that creates capsules representing network packets from a raw stream of bytes. This class is part of a network intrusion detection application written as a Reflex graph. 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.

1.3.3 Implementation sketch

To implement atomic methods, we exploit the preemptible atomic regions facility of the Ovm virtual machine as presented in Section 2. 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 which is invoked periodically by the Reflex framework) some of the

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

  @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. 1.6 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 `readPacket` method is invoked from the Reflex`execute` method.

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 rollback 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.

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. 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.

1.3.3.1 Multicore Implementation.

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. We 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.

1.3.4 Real-time guarantees

The real-time guarantees for atomic methods are slightly different then those of preemptible atomic regions. To start, there is a single real-time thread per Reflex which can interact with possibly multiple plain Java threads. So, the blocking time of the real-time thread is at most the time required to abort the atomic method (i.e. time proportional to the log size). On the other hand, no progress guarantee is given to the plain Java threads trying to communicate with a Reflex. The programming model allows for unbounded aborts in the worst case, though this has not occured in our experiments.

1.3.5 Experimental evaluation

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 when scheduled reads 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.

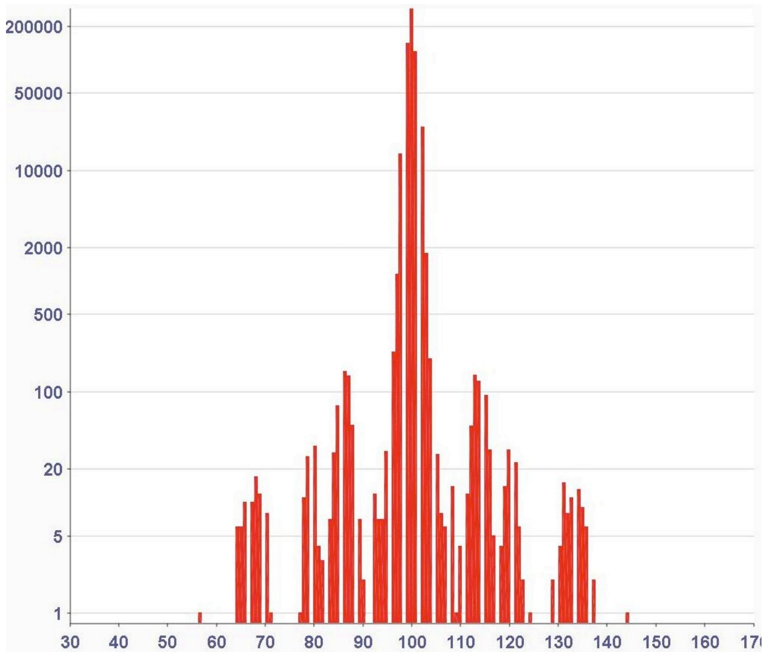

Fig. 1.7 Frequencies of inter-arrival times of a single Reflex task with a period of 100 μs continuously interrupted by an ordinary Java thread invoking an atomic method. The x-axis gives inter-arrival times in microseconds, the y-axis a logarithm of the frequency. The graph shows few departures from the ideal 100 μs inter-arrival times.

Figure 1.7 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.

1.4 Atomicity with Micro-transactions

The third real-time concurrency abstraction we have studied in [17, 15] is a hardware realization of transactional memory called RTTM for Real-Time

Transactional Memory (work done with Schoeberl, Brandner, Meawad, Iyer). The main design goals for the RTTM were: (a) simple programming model and (b) analyzable timing properties. Therefore, all design and architecture decisions were driven by their impact on real-time guarantees. In contrast to other transactional memory proposals RTTM does not aim for a high average case throughput, but for time-predictability. RTTM supports small atomic sections with a few read and write operations. Therefore, it is more an extension of the CAS instruction to simplify the implementation of non-blocking communication algorithms.

In our proposal each core is equipped with a small, fully associative buffer to cache the changed data during the transaction. All writes go only into the buffer. Read addresses are marked in a read set – a simplification that uses only tag memories. The write buffer and tag memory for the read set are organized for single word access. This organization ensures that no false positive conflicts are detected. For the same reason the transaction buffer has to be a fully associative cache with a FIFO replacement strategy. Fully associative caches are expensive and therefore the size is limited. We assume that real-time systems programmers are aware of the high cost of synchronization and will use small atomic sections where a few words are read and written. On a commit the buffer is written to the shared memory. During the write burst on commit all other cores listen to the write addresses and compare those with their own read set. If one of the write addresses matches a read address the transaction is marked to be aborted. The atomicity of the commit itself is enforced by a single global lock – the commit token. The commit token can also be used on a buffer overflow. When a transaction overflows the write buffer or the tag memory for the read set the commit token is grabbed and the transaction continues. The atomicity is now enforced by the commit token. Grabbing the commit token before commit is intended as a backup solution on buffer overflow. It effectively serializes the atomic sections. The same mechanism can also be used to protect I/O operations that usually cannot be rolled back. On an I/O operation within a transaction the core also grabs the commit token. Conflict detection happens only during a commit when all $n-1$ (on a n core multiprocessor) cores listen to the core that commits the transaction and not during local read/writes thus saving valuable CPU cycles. When a conflict is detected the transaction is aborted and restarted.

1.4.1 Example

We have implemented dynamically growing singly linked wait-free queues using different synchronization techniques. In our implementations, we use the Java synchronized and the annotation @atomic for micro-transactions. The implementation involves a 'head' node to keep track of the queue-empty condition. Both the head and the tail pointers point to the 'head' node at the beginning and when the queue is empty.

```
class SLQ {
  final static class Node {
    final Object value;
    volatile Node next;
  }

  volatile Node head = new Node(), tail = head;

  void insert(Node n) {
    tail.next = n; tail = n;
  }

  Object remove() {
     if (head.next == null) return null;
     head = head.next;
     return head.value;
  }
}
```

In the remove method, the removed node is retained as the special 'head' node until the next node is removed. This does not affect the number of retries.

1.4.2 Real-time guarantees

The real-time behavior of such transactions is established by bounding the number of retries r to $n-1$ on a n core multiprocessor. Assuming periodic threads, non-overlapping periods and execution deadline not exceeding the period, the worst case execution time (WCET) of any thread t is given by the equation

$$t = t_{na} + (r+1)t_{amax} \tag{1.1}$$

where t is the worst case execution time, t_{na} is the execution time of the non-atomic section of the thread and t_{amax} is the maximum of the execution times of the atomic sections of all the n threads in the system. Since r is bounded, the WCET of any thread is bounded.

1.5 Experimental evaluation

The experimentation environment is an FPGA programmed with a symmetric shared-memory multi-processor hardware system with four JOP cores. As hardware platform we us an Altera DE2-70 Development board consisting of a Cyclone II EP2C70 FPGA. The Altera board contains 64 MB SDRAM, 2 MB SSRAM and an 8 MB Flash Memory and I/O interfaces such as USB 2.0, RS232, and a ByteBlasterMV port. Each JOP core has a core local 4 KB instruction cache and 1 KB stack cache. The Cyclone FPGA was programmed to simulate the afore-mentioned symmetric shared-memory multi-processor environment.

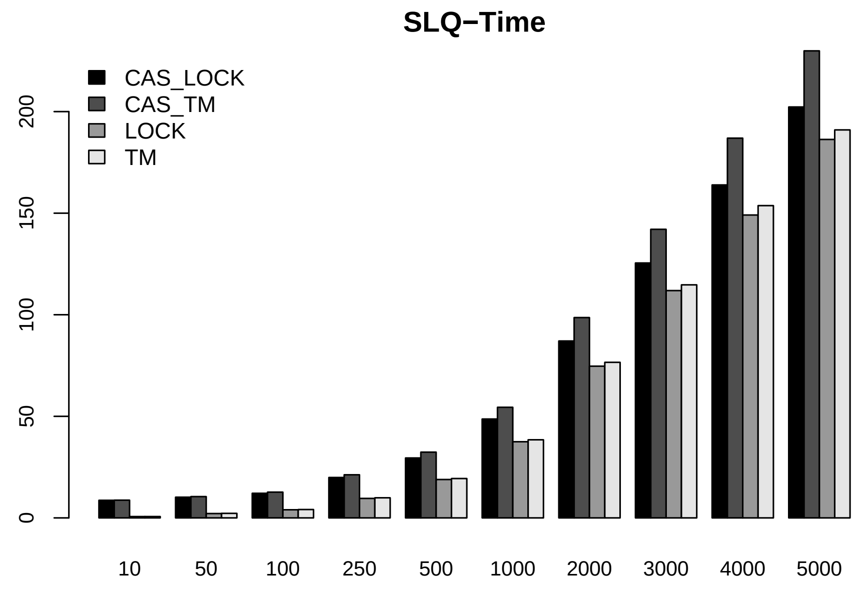

Fig. 1.8 SLQ Time The x-axis gives number of nodes and the y-axis gives the execution time in milli-seconds.

Figure 1.8 plots the execution time to insert, move and remove a specified number of nodes from the singly linked queues. The x-axis indicates the number of nodes used in the experiment, we chose a sample of the small number of nodes followed by a linear increment starting from 1000. The bars indicate the time taken to complete the experiment when different synchronization methods are used. Time is measured from the instance when the insertion of the first node is started till the removal of the last node is completed. It can be noted that, as the number of nodes processed increase the average execution time increases almost linearly due to an increase in the number of locks, transactions and retries. The CAS numbers are simulation of hardware compare and swap as the architecture does not support such instructions. Execution times in the case of TM is 17% lower compared to the CAS cases, it is 2.5% higher than the LOCK case. The higher execution times of the TM implementation relative to that of LOCK can be attributed to the small sizes of the read/write sets and shorter atomic sections in singly linked queues. For example, in a singly linked list, an insert operation involves the modification only of a pointer and the queue tail. As a result, locks are held for a short period reducing overall waiting time. However, in the case of TM, retries, conflict detection and other transactional memory overhead is high as compared to the time lost in waiting for locks.

1.6 Conclusion

Correct and efficient design and implementation of concurrent programs is absolutely vital, but tremendously difficult to achieve. In this paper, we have reported on our experience with three concurrency control abstractions that leverage transactional memory ideas to provide time predictable performance. For uniprocessor systems, preemptible atomic regions provide a very good compromise between performance and responsiveness. They are fast and have simple semantics. They are arguably preferable to locks in most cases. If a restricted programming model is acceptable, then the atomic methods that are part of the Reflex programming model are a nice match that preserves most of the benefits of PARs but can be used in a multi-processor setting. Lastly, on a multi-core real-time system, a hardware implementation such as the one we have proposed with RTTM can provide the appropriate timing guarantee, but unfortunately our preliminary results suggest that there is a decrease in throughput due to the complexity of the hardware. Overall, we believe that transactional abstractions show promise to provide viable alternatives to lock based synchronization in the context of real-time systems.

References

1. James Anderson, Srikanth Ramamurthy, Mark Moir, and Kevin Jeffay. Lock-free transactions for real-time systems. In *Real-Time Database Systems: Issues and Applications*. Kluwer Academic Publishers, Norwell, Massachusetts, 1997.
2. Austin Armbuster, Jason Baker, Antonio Cunei, David Holmes, Chapman Flack, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems (TECS)*, 7(1):1–49, 2007.
3. Brian N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–273, May 1993.

4. Brian N. Bershad, David D. Redell, and John R. Ellis. Fast Mutual Exclusion for Uniprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–233, 1992.
5. Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
6. Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 388–402, Seattle, Washington, November 2003.
7. Maurice Herlihy, Victor Luchangco, Mark Moir, and William Scherer. Software transactional memory for dynamic-sized data structures. In *ACM Conference on Principles of Distributed Computing*, pages 92–101, 2003.
8. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300. ACM Press, 1993.
9. Ye Ding John P. Lehoczky, Lui Sha. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, 1989.
10. M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.
11. Arvind S. Krishna, Douglas C. Schmidt, and Raymond Klefstad. Enhancing Real-Time CORBA via Real-Time Java Features. In *24th International Conference on Distributed Computing Systems (ICDCS 2004), Hachioji, Tokyo, Japan*, pages 66–73, March 2004.
12. E.A. Lee. Overview of the Ptolemy project. Technical Report UCB/ERL M03/25, EECS Department, University of California, Berkeley, 2003.
13. D. B. Lomet. Process structuring, synchronisation and recovery using atomic actions. *Proceedings of the ACM Conference on Language Design for Reliable Software*, 12(3):128–137, 1977.
14. Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time Java. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS)*, December 2005.
15. Fadi Meawad, Karthik Iyer, Martin Schoeberl, and Jan Vitek. Real-time wait-free queues using micro-transactions. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, October 2011.
16. Michael F. Ringenburg and Dan Grossman. AtomCaml: First-class atomicity via rollback. In *Tenth ACM International Conference on Functional Programming*, Tallinn, Estonia, Septempter 2005.
17. Martin Schoeberl, Florian Brandner, and Jan Vitek. Rttm: Real-time transactional memory. In *Symposium on Applied Computing (SAC)*, pages 326–333, 2010.
18. Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.
19. Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 204–213, August 1995.
20. Jesper Honig Spring, Filip Pizlo, Jean Privat, Rachid Guerraoui, and Jan Vitek. Reflexes: Abstractions for integrating highly responsive tasks into Java applications. *ACM Transactions in Embedded Computing Systems (TECS)*, 2009.
21. William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC)*, April 2002.
22. Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Preemption-Based Avoidance of Priority Inversion for Java. In *33rd International Conference on Parallel Processing (ICPP 2004)*, pages 529–538, Montreal, Canada, August 2004.