Preemptible Atomic Regions for Real-time Java

Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan,
Marek Prochazka, Bin Xin, Jan Vitek
Purdue University

Abstract

We present a new concurrency control abstraction for real-time systems called preemptible atomic regions *(PARs). PARs a transactional mechanism that improves upon lock-based mutual exclusion in several ways. First, and foremost, PARs provide strong correctness guarantees. Any sequence of operations declared atomic will not suffer interference from other threads, even in the presence of programmer errors. In spite of this, PARs can be preempted by high priority tasks; this is essential to the minimization of blocking times. We have implemented PARs in a uniprocessor real-time Java virtual machine and evaluated their utility on a number of programs. The results suggest that programs that use PARs, depending on their semantics, can run faster and experience less jitter than those that use locks.*

1 Introduction

The Real-Time Specification for Java (RTSJ) [4] is designed to allow programmers to engineer large scale real-time systems in a modern, type-safe programming environment. Features such as memory safety, checked exceptions, and a rigorously specified memory model, make Java a good programming language for developing mission critical applications. In spite of these benefits, concurrency control remains one area where Java has not substantially advanced the state of the art. To build concurrent programs, it is still necessary to use lock-based critical sections, which are widely recognized as too complex. Data races, deadlocks and violations of atomicity provide a wealth of opportunities for programmers to make mistakes; furthermore, coarse-grained use of locks increases blocking time unnecessarily.

The difficulties in using concurrency in real-time settings have been studied extensively. Programmers are trained to keep critical sections short. They must rely on strict programming and runtime protocols to avoid difficulties; for example, priority inheritance is used to avoid priority inversion [16]. Unfortunately, these approaches do not scale well when working with modern, mission critical, distributed real-time embedded (DRE) systems. DRE applications in domains such as avionics and onboard computing are configured from millions of lines of source code; this makes it difficult for them to provide the kind of assurances that small-scale systems can typically make.

In this paper, we propose a concurrency control abstraction which we call a *preemptible atomic region* (PAR). PARs are a restricted form of software transactional memory [9, 17, 8] that provide a convincing alternative to mutual exclusion monitors. It consists of a sequence of instructions which is guaranteed to execute atomically. If a higher-priority task is released, the effects of the PAR are undone and the high-priority task gets to execute as if the lower-priority task never ran at all. 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. In fact, the blocking time of a thread is, at worst, equal to the longest critical section in a lower-priority thread. Other significant advantages of PARs include the absence of data-races and the fact that no other priority inversion avoidance technique is needed.

The PAR design leverages the uniprocessor nature of the majority of real-time embedded systems to achieve a number of benefits. Code within a PAR can manipulate memory in place. The original contents of heap locations written within a PAR are logged in an undo buffer; this is the only additional overhead during their execution. When an abort occurs, the aborting thread must block while the undo buffer is written back to memory. The PAR design provides two important guarantees. First, a thread may, at most, trigger a single abort; the overall number of aborts is restricted to, at most, one per context switch. Another important property of PARs is that deadlocks cannot occur. A worst case execution time analysis of atomic regions can verify that all threads make progress and that no deadline will be missed.

2 An Introduction to PARs

This section introduces preemptible atomic regions and contrasts them with lock-based concurrency control mechanisms.

Figure 1 is a simplified extract from a queue-based thread pool implementation. The method `leaderExec()` in the class `ThreadPoolLane` places an incoming `Request` onto the queue `requestBuffer` (a.4). If a processor is free, it will dequeue (and execute) the

```
   class ThreadPoolLane {
1.    synchronized leaderExec(Request task){
2.       if (borrowThreadAndExec(task))
3.          synchronized(requestBuffer) {
4.             requestBuffer.enqueue(task);
5.             numBuffered++;
          }
       ...
   }  }

   class Queue {
7.    final Object sObject = new Object();
8.    void enqueue(Object data) {
9.       QueueNode node=getNode();
10.      node.value=data;
11.      synchronized(sObject) {
12.         // enqueue the object
   }  }  }
```

(a) With Monitors.

```
   class ThreadPoolLane {
1.    @PAR leaderExec(Request task){
2.       if (borrowThreadAndExec(task))
3.          requestBuffer.enqueue(task);
4.          numBuffered++;
       ...
   }  }  }

   class Queue {
5.    @PAR void enqueue(Object data) {
6.       QueueNode node=getNode();
7.       node.value=data;
8.       // enqueue the object
   }  }
```

(b) With Preemptible Atomic Regions

Figure 1. Example: A `ThreadPoolLane` from the Zen ORB. (Simplified)

`Request` when it is next scheduled. The code is taken from the Zen real-time ORB [12].

This example make extensive use of synchronization. The method `leaderExec()` is synchronized (a.1) to ensure that multiple threads cannot concurrently access the method of the `ThreadPoolLane` on which it will be invoked. The second use of locks is around lines a.4 and a.5; it ensures that the length of the queue is consistent with `numBuffered`. This cannot be accomplished with the lock on the `ThreadPoolLane` because there may be other methods (not pictured) that are not synchronized on the `ThreadPoolLane` object, but that access the `requestBuffer` queue and `numBuffered`. The final use of locking in this example occurs inside of the implementation of the `Queue` class: the `enqueue()` method relies on a private object (a.7) to protect the updates to the queue (a.12)[1].

We contrast this with an implementation that uses preemptible atomic regions. As mentioned above, a PAR executes atomically. While it is executing, it logs the original contents of locations to which it writes; these values are then restored if the thread is preempted before the PAR ends. To the preempting thread, it appears as if the code had not executed at all. Aborted atomic regions are silently re-executed until they successfully commit. The programming model is intentionally simple; in most cases, monitors can be exchanged for atomic regions with minimal changes to the program. 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.b, we use two atomic sections: one for the `leaderExec()` method (b.1) and another for the `enqueue()` method (b.5). The first PAR is sufficient to prevent all data races within `leaderExec()`; it is therefore unnecessary to obtain a lock on the queue. If `enqueue()` were only called from `leaderExec()`, it would not need to be declared atomic; however, as mentioned above, it is declared atomic to allow use in a non-atomic calling context.

The solution that uses atomic regions is simpler and easier to prove correct, 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.

Moreover, PAR-based mechanisms avoid three major costs found in typical locking protocols:

- **Lock Acquisition Overhead.** The first time a 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 book-keeping 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.

- **Nesting Overhead.** Every nested lock that needs to be acquired incurs an additional overhead. For example, in Figure 1.a, the program will perform three lock acquisitions and three lock releases for each invocation of `leaderExec()`. On the other hand, because PARs can only conflict with other PARs, nested PAR entrances and exits may be ignored. In Figure 1.b, only

[1] This is a fairly common idiom in Java: an internal object is used for synchronization internal to the object because external code needs to use the `Queue` object for (unrelated) synchronization.

two PAR operations (an enter and a commit) will be performed on a call to `leaderExec()`.

- **Context Switching Overhead.** Lock-based implementations also tend to have greater context-switching overhead. Consider the code in Figure 1.a with three threads: `t1`, `t2` and a higher-priority thread `t3`. Thread `t1` can acquire the lock on `sObject` and be preempted by Thread `t2`, which then synchronizes on `requestBuffer`. Now, assume that Thread `t3` attempts to execute `leaderExec()`. 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 `requestBuffer`. Finally, the fourth and fifth switches occur when the system schedules `t1` so that it can release the lock on `sObject`.

  Under the same conditions, the use of PARs only requires one context switch. If `t2` preempts `t1` while it is in 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.

By comparison, 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.

PARs are not a solution to every concurrency control problem. Critical sections that contain long sequences of updates will perform better with conventional locks. Furthermore, input/output operations cannot readily be reexecuted. In spite of these shortcomings, we have found that in the majority of cases we have studied, applications that are written to use PARs outperform the same application using locking protocols.

Perhaps more importantly, we have found that PARs provide greater assurances against programmer error than locks do. Programmers are faced with the need to include more and more functionality in real-time systems. This necessitates the use of "black box"-style component use, which makes it difficult to reason about the semantics of a given program. As can be seen from our example, PARs are easier to compose than locks are: it is much easier to reason about the interaction of PARs across multiple program components than it is for locks. By making it easier to analyze the interaction of components, PARs can mitigate some of these difficulties.

3 Real-time Java

The RTSJ was designed by Sun Microsystems and a consortium of over 40 companies [4]. While the first release of the RTSJ specification appeared in 2000, it is only recently that production implementations have become available.

One of the notable advantages of the RTSJ is that it is possible to implement mixed-mode systems in which real-time and non-real-time tasks can co-exist. The integration of the two programming models, while not seamless, represents a pragmatic engineering compromise. The real-time extensions are backward-compatible with the rest of the Java programming language and require no changes to the tool chain (*e.g.*, the IDE or the compiler). Thus, adopting real-time Java does not require forsaking libraries or legacy code. Instead, it is possible to implement the (typically small) real-time portion of an application using the real-time extensions, and to use standard Java for the rest.

For programmers, the main difference between Java and the RTSJ is that within real-time code, memory management is performed using a region-based memory model in which regions can be deallocated in constant time without requiring a garbage collector. Standard Java objects are still garbage collected, but they live in a segregated portion of memory. This has a number of implications for concurrency control, some of which are discussed below.

In our experience, RTSJ applications can contain up to several hundred threads, all of which have to be scheduled carefully to ensure that all deadlines are met. As usual, in order to ensure schedulability, it is necessary to bound both the time required to execute the thread up to the end of the current period, as well as the thread's *blocking time* (*i.e.*, the time a thread can spend while waiting for locks held by other threads). Computing blocking time requires considering a number of factors:

- **Critical section execution time.** It is necessary to estimate the longest time a thread may block by bounding the length of any given critical section. Object-oriented language features such as dynamic binding, together with the use of components, complicate the task of accurately estimating worst case execution time.

- **Priority inversion.** Priority inversion [13, 5] can be prevented by a number of well known techniques. In the RTSJ, every Java object is equipped with a lock that implements priority inheritance and can optionally support priority ceiling emulation. Supporting priority inheritance is not trivial. Let us assume that a high priority thread τ_h wants to acquire a mutual exclusion lock ℓ_1, and a low priority thread τ_l that currently holds it. Assume also that τ_l is waiting on a lock ℓ_2. Priority inheritance requires that the thread that holds ℓ_2 have its priority raised. In addition to this, priority inheritance must be applied transitively to **any** thread that holds a lock waited for by any priority boosted thread. This creates a ripple effect: boosting the priority of any

blocked thread implies finding the lock on which it is waiting and boosting the priority of the thread holding that lock. These problems are compounded in Java, where applications and library code use locking frequently; as a result of this, the code needed to support priority inheritance imposes a non-negligible runtime penalty.

- **Blocking on Plain Java Threads** In real-time Java, a real-time thread may (accidentally or deliberately) have to wait for a lock held by a non-real-time thread. Because very few Java libraries have been implemented with predictability in mind, a real-time program that uses them may block for an arbitrary length of time.

- **Blocking on Garbage Collection.** Real-time Java distinguishes hard real-time threads from (softer) real-time threads: the former are not allowed to read references to heap objects. This restriction is meant to ensure that a hard real-time thread will never have to wait for the garbage collector. Unfortunately, it is possible to set up a scenario in which a hard real-time thread blocks on a lock held by a soft real-time thread, which then blocks on a lock held by plain Java thread. If memory is exhausted while the plain Java thread is executing, the hard real-time thread will be blocked for the duration of garbage collection.

The motivation for our work is to simplify the task of reasoning about critical sections by providing a concurrency control abstraction that minimizes these problems and attempts to avoid undue blocking delays and catastrophic interference between the real-time and the non-real-time parts of a RTSJ environment.

4 Preemptible Atomic Regions

In Section 2, we introduced PARs and described their high-level semantics. Here, we elaborate on some of the ideas introduced in that section, and provide more detail on their semantics and operation. We also describe some of their shortcomings.

As seen in Figure 1, PARs can be declared by annotating a method `@PAR`. A PAR is active during the dynamic scope of such a method. Nested PARs are permitted, but no additional action is taken when program control enters or exits them. If a thread within one or more nested atomic regions is aborted, all of the changes performed by the outermost PAR are rolled back, and program execution will restart from the outermost PAR.

Since no other threads can see the effects of a PAR in progress, it is safe to abort a thread at any time. In our implementation, atomic methods are aborted every time a higher priority thread is released. Note that this reflects an extremely conservative view of what it means for two critical sections to conflict. This approach has two major implications. First, it reduces blocking time. Specifically, when a high-priority thread is released, it will only ever block for as long as it takes to abort one atomic region. Second, it implies that only one PAR will ever be active at any given point. Thus, an implementation only needs to maintain a single undo log. If an implementation needed to increase opportunities for concurrency, it could abort a PAR only when two threads actually interfere, i.e., read from or write to the same locations. This would make it harder to bound blocking time.

As observed in Section 2, PARs have several other significant benefits. For example, unlike lock-based concurrency control mechanisms, they cannot suffer from deadlock. In addition, there can be, at most, one abort per context switch.

In our implementation, the use of atomic regions introduces several costs. There is only one significant memory overhead: a single system wide log is preallocated with a user defined size (with a default of 10KB). There are several computational overheads. First, 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 exception. This process is described in more detail in Section 5.

As a result of these design decisions, computing the worst-case execution time (WCET) of a program that uses PARs is no harder than computing it for a program that relies on locks. The greatest difference is the need to obtain a bound on the number of writes performed within a critical section. Doing so will give a bound on both the undo costs and the logging overhead.

PARs are not necessarily appropriate for all cases in which concurrency control is necessary. They tend to be widely applicable for real-time code because the majority of critical sections found in real-time code are short. PARs are not suited to long-running critical sections, as an abundance of writes will cause the log to overflow. As we cannot detect which critical sections will be short, our implementation supports a degraded mode of execution in this case – the thread runs with interrupts turned off.

There are other limitations on the use of PARs. In our implementation, native code and I/O should not be executed within a transaction, as we do not have a way to undo their effects automatically. In many common cases, it is unclear what the semantics of an undo would be: what does it mean, for example, to undo a write to the terminal? Blocking operations (such as calls to `wait()` and `notify()`) should also be avoided within PARs.

In cases where PARs are inappropriate, programmers may still use traditional locks. The interaction between the two is straightforward.

5 Implementation

Preemptible atomic regions are a special case of a transactional memory system [9]. The four essential operations for any kind of transactional memory system are *reads* and *writes* of memory, *aborts* and *commits*.

In our implementation, a `read` can access memory directly. This is safe because there can be only one atomic region executing at a given time, though it may be nested. A memory `write` operation incorporates additional instructions that append the memory location and its original value to an undo buffer.

When a PAR is aborted, the corresponding `abort` operation goes through the undo buffer (atomically) in backward chronological order. As it does so, it writes the contents of the buffer out to memory, thus restoring memory to its original state. After this is done, the abort sets a pending `AbortedFault` for the current thread. The `commit` operation commits the actions performed in an atomic region. In our implementation this operation is free, as writes are performed directly on memory.

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 then 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. Thus, conflict detection is only required when a thread is ready to be released by the scheduler. The scheduler is invoked to switch from the currently executing thread `t1` to a new 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.

5.1 Integration with the Virtual Machine

The system described above has been integrated into the Ovm real-time Java virtual machine [1]. Ovm can execute code with an optimizing ahead-of-time compiler, a just-in-time (JIT) compiler, or an interpreter. Since we are mostly concerned with embedded systems, the discussion focuses on Ovm's optimizing ahead-of-time configuration. While PARs have a simple semantics, their integration into a feature complete RTSJ VM is not trivial.

The first phase of our implementation is VM independent: the Java bytecode of the application (the intermediate representation read by a Java virtual machine) is rewritten.

```
void f() {
   while (true) {
      try {
         try {
            PAR.start();
            f$();
         } finally { PAR.commit();
            PAR.exit(); }
      } catch (AbortedFault _) {
         continue; }
      break;
   }
}
```

Figure 2. Code transformation for a method `@PAR void f`. The body of the original method is moved into a new synthetic method named `f$`.

Ovm translates implicit PAR operations embedded in the bytecode into a low-level API, inserting explicit calls to operations such as commit and abort. After this, the optimizing ahead-of-time compiler is used to translate the bytecode into native code. Some changes had to be made to the virtual machine's kernel to support the transactional semantics required by PARs. This section describes these changes, as well as the support implemented in the underlying VM.

5.2 Bytecode rewriting

The first step in compiling an application that uses PARs is to rewrite its bytecode. In our implementation, adding the annotation `@PAR` to a method ensures that it will be executed in a PAR; the implementation employs meta-data annotations as introduced in Java 1.5. We transform any method `f()` with this annotation into a new method named `f$()`. A new `f()` method, as seen in Figure 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()` is executed. This method is enclosed in a `finally` clause to ensure that the transaction commits 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 and does not follow normal Java semantics. This avoids two problems. First, the `finally` clauses in the dynamic scope of a PAR must not execute if the code throws an `AbortedFault`. In Ovm, `finally` clauses are implemented as exception handlers that catch a pointer to any object that is a subtype of `Throwable`. In order to en-

sure that the `finally` clause is not executed when a PAR aborts, at the VM level, the type of `AbortedFault` is modified, by the runtime system, so it is not a subtype of `Throwable`. Second, methods that may be called within the dynamic scope of a PAR may have originally contained their own PARs. Those PARs will have their own `AbortedFault` handlers. Because an abort terminates every active PAR, these handlers must not be allowed to catch an `AbortedFault`. To avoid this, our whole-program analysis (described in the next section) removes any exception handlers that catch `AbortedFault` within the dynamic scope of a PAR.

5.3 Code generation

Our implementation performs a whole program analysis to determine which methods may be called within the dynamic scope of a PAR.[2] It then duplicates these methods and appends the $ character to their name. Method invocations within the scope of the PAR block are then rewritten to call these duplicate methods. Finally, all code that may be executed within the dynamic scope of a PAR is rewritten so that every write to memory is also logged into an undo buffer.

At runtime, at each write in a PAR, the original value of the location being written is stored in an array; the writes to the array are performed in ascending order. When a PAR is aborted, the contents of the log are restored to memory in descending order. This has the effect of "undoing" the writes performed by the thread (as described in Section. 4).

A logging operation is *redundant* if it logs a location whose value has already been stored. In practice, Ovm need not emit logging instructions for redundant stores. However, because experimental results do not show much potential performance improvement from this optimization (because there are typically few writes in PARs), we did not implement it.

Our PAR implementation does not log local variables. Doing so would introduce a great deal of overhead, especially as locals are frequently stored in registers. Methods marked `@PAR` will be reexecuted in their entirety, resulting in the automatic reinitialization of local variables.

5.4 Scheduling

Ovm performs its own priority preemptive scheduling without assistance from the operating system. We adapted the scheduler to support PARs. When the scheduler initiates a context switch, the contention manager is invoked. As described above, when the manager aborts an ongoing transaction, it flags the low-priority thread as requiring an abort. The scheduler is responsible for throwing an `AbortedFault` when a flagged thread is scheduled.

[2] As discussed earlier, every method invocation can dispatch to multiple implementations. We use a reaching type analysis and dead code elimination to obtain a conservative approximation of the actual set of called methods that can be called within a PAR. For our benchmark suite we have observed approximately 10% code blow up due to method duplication. We believe that this could be reduced with more sophisticated static analysis.

Implicit in our design is the notion that only one PAR may be active at a time. This implies that our implementation of PARs would not work on a multi-processor machine. However, this is not tremendously limiting in a real-time context.

5.5 Non-retractable Operations

One of the challenges for the implementation of PARs on top of a virtual machine is that some of the operations performed by the kernel of the VM must not be undone. We call such operations *non-retractable*.

First, user level data structures that are not specific to the thread currently running (such as timers or event counters) must not be reset, as they are logically unrelated to the transaction.

Second, much of the modification of kernel state that is internal to the VM must be treated as non-retractable. For example, in several places, Ovm employs self-modifying code. Fortunately, Ovm's design maintains a clear separation between kernel and user code; it is therefore possible to identify kernel code and compile it without logging.

Finally, there are subtle cases of interactions between kernel and application data structures which require special handling. As an example, consider string *interning*: the creation of a single canonical version of a string for use by the entire VM. String construction takes place in user code, so interning cannot simply be compiled without logging. If an abort takes place after a String is interned, it leaves a pointer from the kernel to an ill-formed (rolled-back) user object.

There are only a small number of similar situations in Ovm (another is class initialization). We deal with these on a case-by-case basis by introducing *partial commits*. When a partial commit starts, it checks if the thread is currently in a transaction; if it is, the position in the undo log is recorded. When it exits, it again checks if a transaction is in progress; if it is, the undo log position will be restored to its earlier value. The net effect is that any user-level changes performed within the dynamic scope of a partial commit will not be undone when an abort occurs.

5.6 Reflective method invocation

In real-time Java, reflection is relatively pervasive. As a result, it is necessary to log reflective methods that may be invoked within a PAR. Ovm relies on an explicit list of methods that may be called reflectively. Using this information, the system creates logged versions of all reflectively invoked methods. The call sites of reflective methods within PARs are altered to invoke these logged methods. If we were to support JIT compilation the logging versions of reflective methods would be generated on-demand. However, JIT compilation is not an option for our target applications.

5.7 Memory management

Our implementation of PARs employs a single, system-wide undo log. The log is preallocated in *immortal* memory

and is not resized; objects allocated in immortal memory live until the end of the application and are never subject to garbage collection. This requirement implies that it is necessary for the size of the log to be known *a priori*; however, since PARs are designed for a real-time environment, we do not consider awareness of the memory constraints to be a drawback. Our implementation leaves room for the log size to be determined by the programmer.

Real-time threads execute within memory regions that are not garbage collected. The size of allocation regions is fixed. If an object is allocated within a PAR, and the PAR is aborted, the memory will be leaked. If a transaction is repeatedly aborted, it is conceivable that the region may run out of memory entirely. A solution to this problem is to undo the effect of allocation. All memory allocated within a transaction can be returned when the transaction exits. What is needed here is for the implementation of `start` to record the value of the allocation pointers in all regions that are accessible to the currently executing thread. When a thread enters a new region while a transaction is active, the allocation pointer of that region is also recorded. The `abort` operation resets the allocation pointers to their previous value. This procedure does not interact with the partially committed transactions mentioned above, because classes and interned string objects are allocated in immortal memory.

Ordinary Java threads run in the garbage collected heap. If a similar leak occurs in such a thread, we can rely on the garbage collector to reclaim the lost memory. For these threads, the GC may be triggered within the scope of a PAR. If this is the case, the transaction is aborted before the GC is run.

6 Response Time Analysis

We outline a response time analysis for PARs for a priority preemptive scheduler. Assume a set of n periodic tasks scheduled according to the *rate monotonic scheme* [10]. Each task τ_i performs a job J_i. A job has period p_i such that $\forall i < n, p_i < p_{i+1}$ and a worst case execution time C_i. 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. 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 [11]. 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 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.

6.1 Response Time Evaluation

In order to compare respective worst case response times, we created a microbenchmark that runs three tasks: a high-priority task τ_{hp}, a medium-priority task τ_{mp}, and a low-priority task τ_{lp}. Each task performs a fixed number of updates and reads of a shared Hashtable; the setup is similar to the one in Section 7.1, but has three tasks instead of two, and is not designed to execute an abort in each period.

We measured C, W and U for these tasks, and used them to compare the response time analysis for PARs with that of the priority inheritance (PIP) and priority ceiling protocols (PCE) [16].

Each task has a single critical section that occupies its entire runtime. As a result of this, $C = W$; therefore, only C is listed. Figure 3 shows the results. As can easily be seen, the response time of the high-priority thread in the PAR configuration is improved at the expense of that of the low-priority thread.

	High	**Medium**	**Low**
C	2300	2350	2450
P	13000	14000	15000
R (PAR)	2316	7032	12048
R (PIP)	4750	7100	7100
R (PCE)	4750	7100	7100

Figure 3. Response Time Analysis (in microseconds) for each priority task of Microbenchmark using PAR, PIP and PCE. Maximum measured abort time (U) is 16 microseconds

7 Experimental Validation

We used a number of benchmark applications to evaluate the usefulness and performance of our implementation of PARs. These include a microbenchmark (Section. 7.1), a 110,000 line real-time avionics application developed by the Boeing company (Section. 7.2), and a real-time CORBA server (Section. 7.3). All measurements were obtained with Ovm running on a 300Mhz Embedded Planet PowerPC 8260 board with 256MB SDRAM, 32 MB Flash, and Embedded Linux.

7.1 Microbenchmark

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. 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 (once per frame).

Figure 4 shows the results of the test. The reader will note two points. First, the latency for the PAR-based HashMap was 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.

Figure 4. Response time of a high-priority thread in the HashMap Microbenchmark. The x-axis indicates the number of periods that have elapsed (frames), and the y-axis indicates the response time of the high-priority thread (in microseconds). Lower is better. The graph compares RTSJ locks with PARs, and indicates that using PARs provides consistently better performance.

7.2 A Real-time Avionics Application

PRISMj is a Real-time Java application developed in a collaboration between the Boeing Company and Purdue University. PRISMj is designed to run on a ScanEagle Unmanned Aerial Vehicle (UAV), a low-cost, high-endurance UAV developed by Boeing and the Insitu Group. PRISMj controls components of the UAV dedicated to the Global Positioning System, the airframe, tactical steering, and navigation steering. It runs over 100 threads in three rate groups (20Hz, 5Hz, and 1Hz). These threads perform different tasks. There is a single infrastructure thread which acts as a cyclic executive and pushes events to components in the physical device layer. Based on those events, 5Hz and 20Hz threads implement steering and route computation. The 1 Hz thread simulates the pilot control component and periodically switches all components in the system between tactical a navigation steering. The source code for PRISMj represents approximately 110 KLoc; this number does not include libraries.

The experiment we ran measured the response time of the special configuration of the PRISMj components that was instrumented to produce benchmarking data. In our setup, we refactored the program to use preemptible atomic regions. The refactoring involved turning 157 synchronized sections into atomic regions. We measured the response time of jobs in the three rate groups for the Boeing `1x` workload, which is a simulation of the workload on the UAV. Figure 5 shows the *worst* response time for each kind of thread. Tasks are modal and the workload varies every 20 frames; the change in workload is clearly visible on the graph. Outliers in the high priority task were consistent across versions

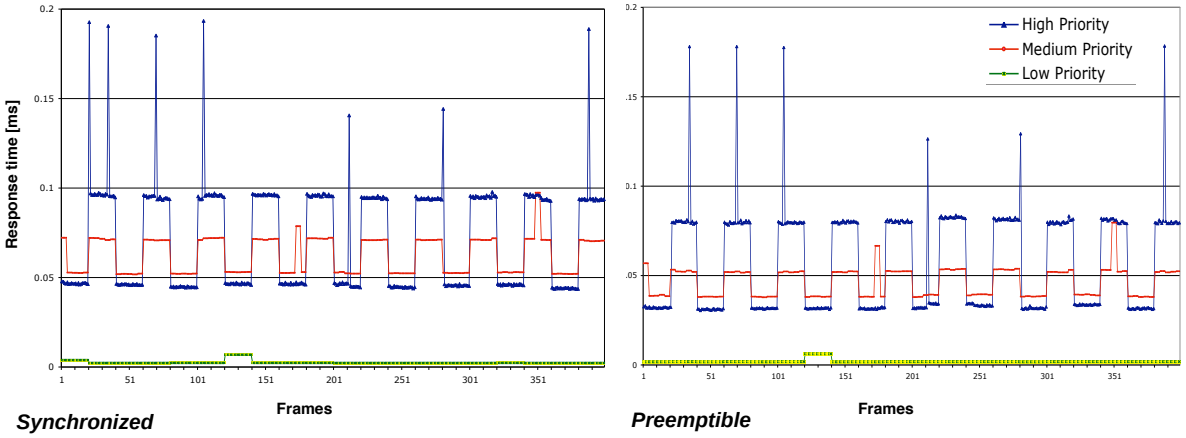

Figure 5. PRISMj Results. Comparing the response times of 100 threads split in three groups (high, medium, low) on a modal workload. The x-axis shows the number of data frames received by the UAV control, the y-axis indicates the time taken by a thread to process the frame. Lower is better.

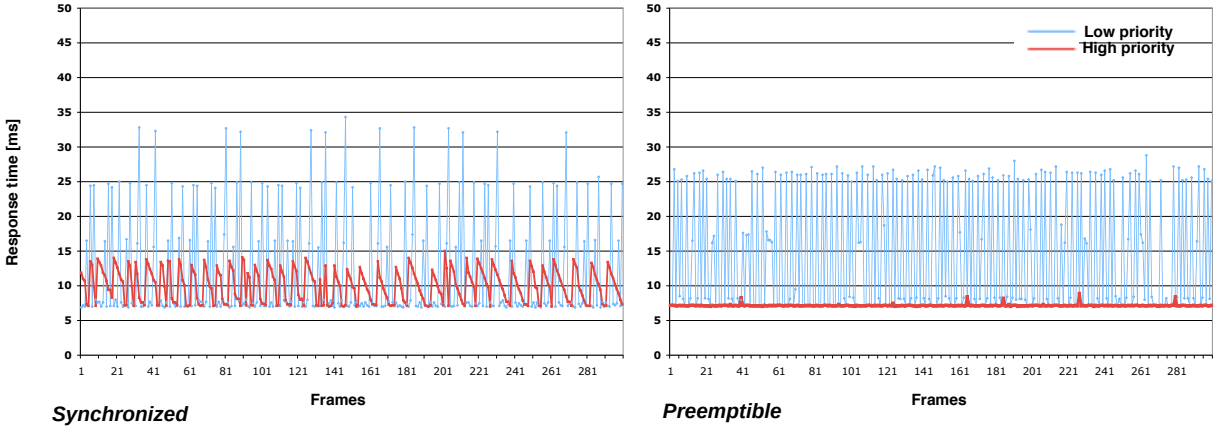

Figure 6. RT-Zen Results. Comparing the response time for a game server running on top of a Real-time Java CORBA implementation. There are two thread groups (low and high) handling 300 requests each. The y-axis indicates the time taken by the application code to process the request. Lower is better.

of the VM and remain within acceptable ranges.

Figure 5 shows that the response times of the high and medium priority threads were consistently better with preemptible atomic regions. There are few runtime aborts in this run. One explanation for the improved performance is that the cost of implementing priority inheritance is high (and that overhead has to paid frequently as Java programs acquire locks often). The low priority thread was mostly unaffected; we assume this is because it does very little work, making it less likely to be preempted.

7.3 Real-time CORBA

RT-Zen is a freely available, open-source, middleware component developed at UC Irvine [12] and written using the Real-time Specification for Java. For this experiment, we use an application which implements a server for a distributed multi-player action game. The application allows players to register with the server, update location information, and find the position of all of the other players in the game. RT-Zen has a pool of worker threads that it uses to serve client requests. Each worker thread is assigned either a high or low priority. The code of the RT-Zen ORB, as well as the demonstration application, were refactored to employ atomic regions in the place of synchronization. In total, 30 synchronized blocks were turned into preemptible methods.

Figure 6 shows the response time of the two categories of threads for the default version of Zen and our PAR version. We measure the time spent in the user code implementing the game server. Five client machines perform 300 invoca-

tions served by low priority threads and 300 served by high priority threads. The results show that high priority threads exhibit much better predictability in response times. Overall, even low priority threads have fewer outliers.

8 Related Work

Our approach is closely related to other work in transactional facilities for programming languages. Lomet presented an early design for atomic actions [14]. A number of later papers investigated the concept of software transactional memory [9, 17], and provided implementations with support for undoing operations. Harris and Fraser [6] described a lightweight transactional model for Java. Their model is more general than ours, but incurs overheads that are much higher, and does not provide real-time guarantees. Bershad investigated atomic sections [3]; however, the undos in that work were limited to short sequences without any user defined state. Anderson et al. [2] described a language independent notion of lock free objects in real-time systems. In contrast, our work leverages its integration with the language and compiler to achieve greater simplicity and efficiency. Harris and Fraser [7] investigated the concept of revocable locks for multi-processor systems. Their work does not consider rollback of state or real-time guarantees. Welc et al. investigated the interaction of preemption and transactions on a multi-processor [18], but did not provide any real-time guarantees. Finally, Rigneburg and Grossman have developed an atomic extension to the language Caml using similar implementation techniques and also targeting uniprocessors [15]. Their implementation does not address real-time constraints; it guarantees neither constant-time logging nor linear time rollbacks.

9 Conclusion

Preemptible atomic regions are an abstraction for controlling concurrency in real-time programs. They provide a model with stronger correctness guarantees than lock-based synchronization. Experimental results suggest that PARs incur smaller overhead and experience less jitter than comparable programs written with traditional lock-based mechanisms.

Acknowledgments. The authors thank Filip Pizlo and the Ovm development team, Chip Austin, Tim Harris, David Holmes, Doug Lea, Ed Pla, Andy Wellings, and the anonymous reviewers. This work was supported in part by the NSF grant CCF-0341304 and DARPA PCES.

References

[1] The Ovm virtual machine, `www.ovmj.org`, 2005.

[2] J. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay. Lock-free transactions for real-time systems. In *Real-Time Database Systems: Issues and Applications*. Kluwer Academic Publishers, Norwell, Massachusetts, 1997.

[3] B. N. Bershad, D. D. Redell, and J. 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.

[4] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[5] J. B. Goodenough and L. Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks. *ACM SIGADA Ada Letters*, 8(7):20–31, Fall 1988.

[6] T. Harris and K. Fraser. Language support for lightweight transactions. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 388–402, Seattle, Washington, Nov. 2003.

[7] T. L. Harris and K. Fraser. Revocable locks for non-blocking programming. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, June 2005.

[8] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *ACM Conference on Principles of Distributed Computing*, pages 92–101, 2003.

[9] M. Herlihy and J. E. 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.

[10] Y. D. 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.

[11] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.

[12] A. S. Krishna, D. C. Schmidt, and R. 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, Mar. 2004.

[13] D. Locke, L. Sha, R. Rajkumar, J. Lehoczky, and G. Burns. Priority inversion and its control: An experimental investigation. *ACM SIGADA Ada Letters*, 8(7):39–42, Fall 1988.

[14] 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.

[15] M. Ringenburg and D. Grossman. AtomCaml: First-class atomicity via rollback. In *Tenth ACM International Conference on Functional Programming*, Tallinn, Estonia, Septempter 2005.

[16] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, Sept. 1990.

[17] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 204–213, Aug. 1995.

[18] A. Welc, A. L. Hosking, and S. Jagannathan. Preemption-Based Avoidance of Priority Inversion for Java. In *33rd International Conference on Parallel Processing (ICPP 2004)*, pages 529–538, Montreal, Canada, Aug. 2004.