Preemptible Atomics

Jan Vitek

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

NSF grant CCF-0341304 and DARPA PCES.
Why not Lock-based Synchronization?

Challenges of programming with mutual exclusion locks:

- ✓ avoiding data races
- ✓ choosing lock granularity
- ✓ enforcing lock acquisition order
- ✓ dealing with modularity and abstraction

& in hard real-time systems:

- ✓ bounding blocking time
- ✓ avoiding priority inversion
Preemptible Atomics

- Transactional concurrency control construct

  Designed for commodity uniprocessor embedded systems

  Alternative to locks with, e.g., priority inheritance (PIP)

- Atomicity

  All statements will execute, or none.

- Strong Isolation

  High priority threads (HPT) preempt Atomics in LPTs

  HPT execute without observing changes performed by LPT
Example with Locks

class ThreadPoolLane {
    synchronized leaderExec(Request task) {
        if (borrowThreadAndExec(task)) {
            synchronized(rQueue) {
                rQueue.enqueue(task);
                numBuffered++;
            }
        }...
    }
}

class Queue {
    final Object sObject = new Object();
    void enqueue(Object data) {
        QueueNode node=getNode();
        node.value=data;
        synchronized(sObject) {
            // enqueue the object
        }
    }
}

from the UCI Zen  real-time ORB
(c) Jan Vitek 2006
Example with Atomics

```java
class ThreadPoolLane {
    @Atomic leaderExec(final Request task) {
        if (borrowThreadAndExec(task)) {
            rQueue.enqueue(task);
            numBuffered++;
        }
    }
}

class Queue {
    @Atomic void enqueue(final Object data) {
        QueueNode node=getNode();
        node.value=data;
        // enqueue the object
    }
}
```
Related Work

- Bershad, Redell, Ellis.
  – no undo

- Anderson, Ramamurthy, Jeffay,
  – non-blocking algorithms, no language support

- Herlihy+, Harris+, Welc+,
  – weak isolation

- Ringenburg, Grossman,
  AtomCaml First-Class Atomicity with Rollback, ICFP, 2005.
  – no real-time guarantees, simpler environment
Semantics

```java
@Atomic method(...) { B }
```

- B logically atomic
- B can be preempted by a higher-priority thread
- If preempted, B’s updates not be observed by HPT
- Nesting coalesced in a single atomic.
PIP locks vs Atomics

Locks with Priority Inheritance Protocol

Atomics
Schedulability

Assuming tasks scheduled with a rate monotonic scheme:

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

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

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

- **Priority Inheritance Protocol:**
  A HPT may block for multiple LPT
  Deadlock and data races
  Non-real-time LPTs may cause unbounded blocking
  *programmer error, but an easy one to make.*

- **Priority Ceiling Protocol:**
  HPTs may still have to wait for completion of a LPT
  Hard to assign ceilings with libraries, changing thread priorities

- **Preemptible Atomic Region:**
  HPTs only block for higher-level tasks.
  At most one abort per context switch.
  no dead-locks & no live-locks
  *if schedulable*
Refactoring Legacy Code

- Locks ⇒ Atomics = ~straightforward

- All uses of a particular lock must be made into atomic

- Consider:

```java
public class Vector extends AbstractList ... {
    @Atomic public void insertElementAt(Object o ...) ...
    @Atomic public int size() { ...
```

- N.B. requires preemptible & logged System.arraycopy
Locks and Atomics

- Atomic must coexist with PIP-locks
- Lock long lived, write-intensive methods
- HPT in an atomic needs to acquire lock held by a LPT:
  \[ \text{undo} \Rightarrow \text{boost and execute LPT} \Rightarrow \text{reexecute HPT} \]
- Wait / Notify can be used when needed
IMPLEMENTATION
A method “@Atomic f(){ x++; B(); }” is translated to:

```java
while (true) {
    try{
        try {  Transaction.start();
            log(x);
            x++
            B_T();
        } finally { Transaction.commit(); break; }
    } catch(Retry _) { } // undo performed by aborting thread
}
```

- finally implemented by catching all subclasses of Throwable
- Retry not a subclass of Throwable, not get caught by finally
Scaling-Up

- **I/O** - How do you undo a write to the screen? You don’t. Could support buffering of output/replay of input or using compensations.

- **Garbage collection** - Addresses stored in log need to be updated. GC must be preemptible and cannot preempt RT task. Now - Rollback the Atomic if a GC is triggered.

- **Dynamic class loading** - Could generate transactional versions of methods on the fly. Now - RT does not require dynamic class loading.

- **Reflection** - Methods invoked reflectively from an Atomic must be transactional. Simple check in the implementation of the reflection package.

- **Regions** - Memory allocated within a region must be returned on abort to avoid leaks.

- **Asynchronous Transfer of control** - Defer until interruptible, then abort.
Optimizations

- Turn an atomic into a nop
  \[
  \text{@Atomic } m() \implies \text{ @Uninterruptible } m()
  \]

- Safe iff execution time is bounded

- Heuristic: short, non-looping methods

- (n.b. not safe for lock-based sync)
Extensions

- Prescient commits
  
  *exception throwing code does not affect or rely on user allocated heap data*

- Open nesting
  
  *string interning requires that strings not be undone as the VM kernel has pointer on char array*

- Exposed regions
  
  *operations are immediately made visible, aborts are deferred, e.g. for debugging*
Evaluation
SpecJVM98

- Ovm performance is competitive.

AMD Athlon XP1900+, 1.6GHz, 1GB RTLlinux, 2.4.7-timesys-3.1.214
Microbenchmarks

- HTP response times

2 threads, performing mix of get/put ops into a HashMap
300Mhz PPC, 256MB memory, Embedded Planet Linux
Ovm RTSJ VM, AOT, priority preemptive, PIP locks
UCI’s RT-ZEN

- Real-time CORBA ORB written in RTSJ, 179,000 LOC,
- ~600 synchronized stmts mechanically translated to atomics

30 HPT/70 LPT. Measure time to process a request

AMD Athlon XP1900+, 1.6GHz, 1GB RTLinux
PRiSMj

- Avionics applications from the Boeing Company
- Benchmark scenarios w. different workloads / components
- Oscillating modal behavior
- ~100 periodic threads in three main rate groups: 1, 5, 20Hz
- 953 Java classes, 6616 methods.
- Deployed on a ScanEagle
PRiSMj: 1X

- High responsiveness, small workloads

- 300Mhz PPC, 256MB memory, Embedded Planet Linux
- Ovm RTSJ VM, AOT, priority preemptive, PIP locks

Atomics (aborts): 3'180 (0)
Reads Max (median): 514 (6)
Writes Max (media): 115 (3)
Monitor inflated: 1338
PRiSMj: 100X

- **Large workloads**

300Mhz PPC, 256MB memory, Embedded Planet Linux
Ovm RTSJ VM, AOT, priority preemptive, PIP locks
Conclusions

- Easier to write reusable correct concurrent real-time code
- Improve responsiveness with little impact on throughput
- Not a replacement for locks, another tool in the box

source code at http://ovmj.org

[Manson+. Preemptible Atomic Regions for Real-time Java. RTSS’05]
[Baker+. A Real-time Java Virtual Machine for Avionics. RTAS’06]