The Atomos Transactional Programming Language

Brian D. Carlstrom, Austen McDonald, Hassan Chafi,
JaeWoong Chung, Chi Cao Minh,
Christos Kozyrakis, Kunle Olukotun

Computer Systems Laboratory
Stanford University
http://tcc.stanford.edu
Transactional Memory

• Reasons to use Transactional Memory (TM)
  ▪ Replace mutual exclusion with concurrent transactions
  ▪ Remove challenges to programming with locks

• Challenges
  ▪ Long running transactions without lower level violations
    • Easier to use one big transaction than having to split into chunks
    • Application libraries and runtimes want to update encapsulated state
  ▪ Transactional conditional waiting with hardware support
    • Software transactional memory (STM) systems have an arbitrary number of transactional contexts in memory, allowing some to be idle
    • Hardware transactional memory (HTM) systems have a fixed number of transactional contexts in silicon, don’t want to busy wait
The Atomos Programming Language

• Atomos derived from Java
  - atomic replaces synchronized
  - retry replaces wait/notify/notifyAll

• Atomos design features
  - Open nested transactions
    - open blocks committing nested child transaction before parent
    - Useful for language implementation but also available for applications
  - Watch Sets
    - Extension to retry for efficient conditional waiting on HTM systems

• Atomos implementation features
  - Violation handlers
    - Handle expected violations without rolling back in all cases
    - Not part of the language, only used in language implementation
synchronized **versus** atomic

Java

```java
... synchronized (hashMap) {
    hashMap.put(key, value);
}
...```

Atomos

```java
... atomic {
    hashMap.put(key, value);
}
...```

**Transactional memory advantages**

- No association between a lock and shared data
- Non-conflicting operations can proceed in parallel
The counter problem

Application

```java
atomic {
  ...
  this.id = getUID();
  ...
}

static long getUID () {
  atomic {
    globalCounter++;
  }
}
```

JIT Compiler

```java
// method prolog
...
invocationCounter++;
...
// method body
...
// method epilogue
...
```

• Lower-level updates to global data can lead to violations
• General problem not confined to counters:
  ▪ Application level caching
  ▪ Cooperative scheduling in virtual machine
Open nested solution to the counter problem

- **Solution**
  - Wrap counter update in open nested transaction

  ```java
  atomic {
    ...
    this.id = getUID();
    ...
  }
  
  static long getUID () {
    open {
      globalCounter++;
    }
  }
  ```

- **Benefits**
  - Violation of counter just replays open nested transaction
  - Open nested commit discards child’s read-set preventing later violations

- **Issues**
  - What happens if parent rolls back after child commits?
  - Okay for statistical counters and UID
  - Not okay for SPECjbb2000 object allocation counters
    - Need to some way to *compensate* if parent rolls back
Transaction Commit and Abort Handlers

- Programs can specify callbacks at end of transaction
  - Separate interfaces for commit and abort outcomes
    ```java
    public interface CommitHandler { boolean onCommit();}
    public interface AbortHandler { boolean onAbort();}
    ```
  - DB technique for delaying non-transactional operations
  - Harris brought the technique to STM for solving I/O problem
    - See *Exceptions and side-effects in atomic blocks*.
      - Buffer output until commit, rewind input on abort
  - In Atomos, commit of open nested transaction can register abort handler for parent transaction
    - This allows for compensating transaction for object counter example
Java

```java
public int get (){
    synchronized (this) {
        while (!available)
            wait();
        available = false;
        notifyAll();
        return contents;
    }
}
```

Atomos

```java
public int get (){
    atomic {
        if (!available)
            retry;
        available = false;
        return contents;
    }
}
```

Transactional memory advantages

- Automatic reevaluation of `available` condition
- No need for explicit `notifyAll`
Transactional Conditional Waiting

- When condition false, wait until read set violated
  - Leverage violation detection for efficient wakeup
  - When violation happens
    - Rollback waiting transaction
    - Move thread from waiting to ready

- Approach scales well in STM
  - No practical limit on number of transactional contexts

- However HTM has limited number of hardware contexts
  - Can we overcome this issue?
Hardware Transactional Conditional Waiting

- Instead of using one HW context per waiting transaction
  - Merge waiting read sets into one shared context
- Our VM already has dedicated VM scheduler thread
  - Use as shared context
- Challenges
  - How can we communicate read set between threads?
  - How can shared context handle violations for others?

Diagram:
- Consumer
  - A
  - available? No
  - xfer read-set
  - rollback
- Scheduler
  - reschedule A
  - available? Yes!
- Producer
  - available=true
  - commit
  - violation
Violation Handlers

• Violation Handlers solve both challenges
  ▪ Thread can register handler for violation callbacks
    public interface ViolationHandler {
        boolean onViolation (Address violatedAddress);}

• How can we communicate read set between threads?
  ▪ Use open nested transaction to send command to scheduler
  ▪ Scheduler ViolationHandler receives commands

• How can shared context handle violations for others?
  ▪ Scheduler maintains map of addresses to interested threads
  ▪ non-command violation moves threads from waiting to ready
Common case transactional waiting

• Issues with transferring the read-set on retry
  ▪ Need HW interface to enumerate read-set
  ▪ Want to minimize size the number of addresses
  ▪ Want to prevent overflow of HW transactional context

• Solution
  ▪ Program usually only cares about changes to a small subset of its read-set
  ▪ This watch-set will usually only be a single address

```java
public int get (){  
    atomic {  
      if (!available){  
        watch available;  
        retry;  
      }
      available = false;  
      return contents; }
```
Hardware and Software Environment

- The simulated chip multiprocessor TCC Hardware (See PACT 2005)

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>1-32 single issue PowerPC core</td>
</tr>
<tr>
<td>L1</td>
<td>64-KB, 32-byte cache line, 4-way associative, 1 cycle latency</td>
</tr>
<tr>
<td>Victim Cache</td>
<td>8 entries fully associative</td>
</tr>
<tr>
<td>Bus width</td>
<td>16 bytes</td>
</tr>
<tr>
<td>Bus arbitration</td>
<td>3 pipelined cycles</td>
</tr>
<tr>
<td>Transfer Latency</td>
<td>3 pipelined cycles</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>8MB, 8-way, 16 cycles hit time</td>
</tr>
<tr>
<td>Main Memory</td>
<td>100 cycles latency, up to 8 outstanding transfers</td>
</tr>
</tbody>
</table>

For detailed semantics of open nesting, handlers, etc., see ISCA 2006

- Atomos built on top of Jikes RVM
  - Derived from Jikes RVM 2.4.2+CVS using GNU Classpath 0.19
  - All necessary code precompiled before measurement
  - Virtual machine startup excluded from measurement
Transactions keep data structures simple

- **TestHashtable**
  - 50%-50% mix of reads and write to Map implementations
- **Comparison of Map performance**
  - **Java HashMap**
    - No built in synchronization
    - Collections.synchronizedMap
  - **Java Hashtable**
    - Singe coarse lock
  - **Java ConcurrentHashMap**
    - Fine grained locking
  - **Atomos HashMap**
    - Simple HashMap with transactions scales better than than ConcurrentHashMap
Transactional conditional waiting evaluation

- **TestWait benchmark**
  - Pass tokens in circle
  - Uses blocking queues
  - 32 CPUs, vary token count

- **Purpose**
  - Used by Harris and Fraser to measure Conditional Critical Region (CCR) performance

- **Results**
  - Atomos similar scalability to Java with few tokens
  - As token count nears CPU count, violation detection short circuits wait code, avoiding context switch overhead
The Atomos Programming Language

- Atomos derived from Java
  - **Transactional Memory for concurrency**
    - `atomic` blocks define basic nested transactions
    - Removed `synchronized`
  - **Transaction based conditional waiting**
    - Derivative of Conditional Critical Regions and Harris `retry`
    - Removed `wait`, `notify`, and `notifyAll`
    - `Watch sets` for efficient implementation on HTM systems
  - **Open nested transactions**
    - `open` blocks committing nested child transaction before parent
    - Useful for language implementation but also available for applications
  - **Violation handlers**
    - Handle expected violations without rolling back in all cases
    - Not part of the language, only used in language implementation
- Finally, *atomos* is the classical Greek word for indivisible
  - “a” prefix means “not” and “tomos” root means “cuttable”