Atomicity via Source-to-Source Translation

Benjamin Hindman  Dan Grossman
University of Washington

22 October 2006
Atomic

An easier-to-use and harder-to-implement primitive

```
void deposit(int x) {
    synchronized(this) {
        int tmp = balance;
        tmp += x;
        balance = tmp;
    }
}
```

```
void deposit(int x) {
    atomic {
        int tmp = balance;
        tmp += x;
        balance = tmp;
    }
}
```

lock acquire/release (behave as if)
no interleaved computation
Why the excitement?

• Software engineering
  – No brittle object-to-lock mapping
  – Composability without deadlock
  – Simply easier to use

• Performance
  – Parallelism unless there are dynamic memory conflicts

But how to implement it efficiently…
This Work

Unique approach to “Java + atomic”

1. Source-to-source compiler (then use any JVM)
2. Ownership-based (no STM/HTM)
   - Update-in-place, rollback-on-abort
   - Threads retain ownership until contention
3. Support “strong” atomicity
   - Detect conflicts with non-transactional code
   - Static optimization helps reduce cost
Outline

• Basic approach
• Strong vs. weak atomicity
• Benchmark evaluation
• Lessons learned
• Conclusion
Our "run-time"

Our compiler

Polyglot

foo.ajava

javac

AThread.java

…

Note: Separate compilation or optimization

class files
Key pieces

• A field read/write first *acquires ownership* of object
  – In transaction, a write also *logs the old value*
  – No synchronization if already own object
• Some Java cleverness for efficient logging
• *Polling* for releasing ownership
  – Transactions rollback before releasing
• Lots of omitted details for other Java features
Acquiring ownership

All objects have an owner field

```java
class AObject extends Object {
    Thread owner;  // who owns the object
    void acq(){...}  // owner=caller (blocking)
}
```

Field accesses become method calls

• Read/write barriers that acquire ownership
• Calls simplify/centralize code (JIT will inline)
Field accessors

D x; // field in class C
static D get_x(C o){
    o.acq(); return o.x;
}
static D set_nonatomic_x(C o, D v) {
    o.acq(); return o.x = v;
}
static D set_atomic_x(C o, D v) {
    o.acq();
    ((AThread)currentThread()).log(...);
    return o.x = v;
}

Note: Two versions of each application method, so know which version of setter to call
Important fast-path

If thread already owns an object, no synchronization

```java
void acq(){
    if(owner==currentThread()) return;
    ...
}
```

• Does not require sequential consistency
• With “owner=currentThread()” in constructor, thread-local objects never incur synchronization

Else add object to owner’s “to release” set and wait
  – Synchronization on owner field and “to release” set
  – Also fanciness if owner is dead or blocked
Logging

• Conceptually, the log is a stack of triples
  – Object, “field”, previous value
  – On rollback, do assignments in LIFO order
• Actually use 3 coordinated arrays
• For “field” we use singleton-object Java trickery:

```java
D x; // field in class C
static Undoer undo_x = new Undoer() {
    void undo(Object o, Object v) {
        ((C)o).x = (D)v;
    }
}
...
...currentThread().log(o, undo_x, o.x);...
```
Releasing ownership

- Must “periodically” check “to release” set
  - If in transaction, first rollback
    - Retry later (after backoff to avoid livelock)
  - Set owners to null
- Source-level “periodically”
  - Insert call to `check()` on loops and non-leaf calls
  - Trade-off synchronization and responsiveness:

```c
int count = 1000; //thread-local
void check()
{
  if(--count >= 0) return;
  count=1000; really_check();
}
```
But what about…?

Modern, safe languages are big

See paper & tech. report for:
   constructors, primitive types, static fields,
   class initializers, arrays, native calls,
   exceptions, condition variables, library classes,
   …
Outline

• Basic approach
• Strong vs. weak atomicity
• Benchmark evaluation
• Lessons learned
• Conclusion
Strong vs. weak

- **Strong**: atomic not interleaved with any other code
- **Weak**: semantics less clear
  - “If atomic races with non-atomic code, undefined”
    - Okay for C++, non-starter for safe languages
  - Atomic and non-atomic code can be interleaved
    - For us, remove read/write barriers outside transactions
- One common view: strong what you want, but too expensive in software
  - Present work offers (only) a glimmer of hope
Examples

```java
atomic { 
    if(x!=null) 
        x.f=42;
}
```

```java
x=null;
```

```java
atomic { 
    x=secret_password;
    //compute with x
    x=null;
}
```

```java
print(x);
```

```java
x=null;
```
Optimization

Static analysis can remove barriers outside transactions
• In the limit, “strong for the price of weak”

Thread local

Not used in atomic

Immutable

• This work: Type-based alias information
• Ongoing work: Using real points-to information
Outline

- Basic approach
- Strong vs. weak atomicity
- Benchmark evaluation
- Lessons learned
- Conclusion
Methodology

• Changed small programs to use atomic (manually checking it made sense)
  – 3 modes: “weak”, “strong-opt”, “strong-noopt”
  – And original code compiled by javac: “lock”

• All programs take variable number of threads
  – Today: 8 threads on an 8-way Xeon with the Hotswap JVM, lots of memory, etc.
  – More results and microbenchmarks in the paper

• Report slowdown relative to lock-version and speedup relative to 1 thread for same-mode
A microbenchmark

crypt:
- Embarrassingly parallel array processing
- No synchronization (just a main Thread.join)

<table>
<thead>
<tr>
<th></th>
<th>lock</th>
<th>weak</th>
<th>strong-opt</th>
<th>strong-noopt</th>
</tr>
</thead>
<tbody>
<tr>
<td>slowdown vs. lock</td>
<td>--</td>
<td>1.1x</td>
<td>1.1x</td>
<td>15.0x</td>
</tr>
<tr>
<td>speedup vs. 1 thread</td>
<td>5x</td>
<td>5x</td>
<td>5x</td>
<td>0.7x</td>
</tr>
</tbody>
</table>

• Overhead 10% without read/write barriers
  - No synchronization (just a main Thread.join)
• Strong-noopt a false-sharing problem on the array
  - Word-based ownership often important
TSP

A small clever search procedure with irregular contention and benign purposeful data races
   – Optimizing strong cannot get to weak

<table>
<thead>
<tr>
<th></th>
<th>lock</th>
<th>weak</th>
<th>strong-opt</th>
<th>strong-noopt</th>
</tr>
</thead>
<tbody>
<tr>
<td>slowdown vs. lock</td>
<td>--</td>
<td>2x</td>
<td>11x</td>
<td>21x</td>
</tr>
<tr>
<td>speedup vs. 1 thread</td>
<td>4.5x</td>
<td>2.8x</td>
<td>1.4x</td>
<td>1.4x</td>
</tr>
</tbody>
</table>

Plusses:
• Simple optimization gives 2x straight-line improvement
• Weak “not bad” considering source-to-source
Outline

• Basic approach
• Strong vs. weak atomicity
• Benchmark evaluation
• Lessons learned
• Conclusion
Some lessons

1. Need multiple-readers (cf. reader-writer locks) and flexible ownership granularity (e.g., array words)
2. High-level approach great for prototyping, debugging
   – But some pain appeasing Java’s type-system
3. Focus on synchronization/contention (see (2))
   – Straight-line performance often good enough
4. Strong-atomicity optimizations doable but need more
5. Modern language features a fact of life
Related work

Prior software implementations one of:
• Optimistic reads and writes + weak-atomicity
• Optimistic reads, own for writes + weak-atomicity
• For uniprocessors (no barriers)
All use low-level libraries and/or code-generators

Hardware:
• Strong atomicity via cache-coherence technology
• We need a software and language-design story too
Conclusion

Atomicity for Java via source-to-source translation and object-ownership
  – Synchronization only when there’s contention

Techniques that apply to other approaches, e.g.:
• Retain ownership until contention
• Optimize strong-atomicity barriers

The design space is large and worth exploring
  – Source-to-source not a bad way to explore
To learn more

- Washington Advanced Systems for Programming
  wasp.cs.washington.edu

- First-author: Benjamin Hindman
  - B.S. in December 2006
  - Graduate-school bound
  - This is just 1 of his research projects
[ Presentation ends here ]
Not-used-in-atomic

This work: Type-based analysis for not-used-in-atomic
• If field $f$ never accessed in atomic, remove all barriers on $f$ outside atomic
• (Also remove write-barriers if only read-in-atomic)
• Whole-program, linear-time

Ongoing work:
• Use real points-to information
  – Present work undersells the optimization’s worth
• Compare value to thread-local
Strong atomicity

(behavior as if) no interleaved computation

• Before a transaction “commits”
  – Other threads don’t “read its writes”
  – It doesn’t “read other threads’ writes”

• This is just the semantics
  – Can interleave more unobservably
Weak atomicity

(behave as if) no interleaved transactions

• Before a transaction “commits”
  – Other threads’ transactions don’t “read its writes”
  – It doesn’t “read other threads’ transactions’ writes”

• This is just the semantics
  – Can interleave more unobservably
Evaluation

Strong atomicity for Caml at little cost
  – Already assumes a uniprocessor
  – See the paper for “in the noise” performance

• Mutable data overhead

<table>
<thead>
<tr>
<th></th>
<th>not in atomic</th>
<th>in atomic</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>write</td>
<td>none</td>
<td>log (2 more writes)</td>
</tr>
</tbody>
</table>

• Choice: larger closures or slower calls in transactions
• Code bloat (worst-case 2x, easy to do better)
• Rare rollback
### Strong performance problem

Recall uniprocessor overhead:

<table>
<thead>
<tr>
<th></th>
<th>not in atomic</th>
<th>in atomic</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>write</td>
<td>none</td>
<td>some</td>
</tr>
</tbody>
</table>

With parallelism:

<table>
<thead>
<tr>
<th></th>
<th>not in atomic</th>
<th>in atomic</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>none iff weak</td>
<td>some</td>
</tr>
<tr>
<td>write</td>
<td>none iff weak</td>
<td>some</td>
</tr>
</tbody>
</table>

Start way behind in performance, especially in imperative languages (cf. concurrent GC)
Not-used-in-atomic

Revisit overhead of not-in-atomic for strong atomicity, given information about how data is used in atomic

<table>
<thead>
<tr>
<th></th>
<th>not in atomic</th>
<th>in atomic</th>
</tr>
</thead>
<tbody>
<tr>
<td>no atomic access</td>
<td>none</td>
<td>atomic write</td>
</tr>
<tr>
<td>no atomic write</td>
<td>none</td>
<td>some</td>
</tr>
<tr>
<td>read</td>
<td>none</td>
<td>some</td>
</tr>
<tr>
<td>write</td>
<td>none</td>
<td>some</td>
</tr>
</tbody>
</table>

• Yet another client of pointer-analysis
• Preliminary numbers very encouraging (with Intel)
  – Simple whole-program pointer-analysis suffices