Software Transactions: A Programming-Languages Perspective

Dan Grossman
University of Washington

5 December 2006
A big deal

Research on software transactions broad...

- Programming languages
  PLDI, POPL, ICFP, OOPSLA, ECOOP, HASKELL, ...

- Architecture
  ISCA, HPCA, ASPLOS, MSPC, ...

- Parallel programming
  PPoPP, PODC, ...

... and coming together
  TRANSACT (at PLDI06 and PODC07)
Why now?

Small-scale multiprocessors unleashed on the programming masses

Threads and shared memory remains a key model

Locks + condition-variables cumbersome & error-prone

Transactions should be a hot area
  An easier to use and harder-to-implement synchronization primitive:

\[
\text{atomic} \{ s \}
\]
PL Perspective

Key complement to the focus on “transaction engines” and low-level optimizations

Language design:
  interaction with rest of the language
    – Not just I/O and exceptions (not this talk)

Language implementation:
  interaction with the compiler and today’s hardware
    – Plus new needs for high-level optimizations
Today

Issues in language design and semantics
1. Transactions for software evolution
2. Transactions for strong isolation [Nov06]*
3. The need for a memory model [MSPC06a]**

Software-implementation techniques
1. On one core [ICFP05]
2. Without changing the virtual machine [MSPC06b]
3. Static optimizations for strong isolation [Nov06]*

* Joint work with Intel PSL
** Joint work with Manson and Pugh
Having chosen “self-locking” today, hard to add a correct transfer method tomorrow

```java
void deposit(...) { synchronized(this) { ... }}
void withdraw(...) { synchronized(this) { ... }}
int balance(...) { synchronized(this) { ... }}
void transfer(Acct from, int amt) {
    synchronized(this) {
        //race
        if(from.balance()>=amt && amt < maxXfer) {
            from.withdraw(amt);
            this.deposit(amt);
        }
    }
}
```
Having chosen “self-locking” today, hard to add a correct transfer method tomorrow

```java
void deposit(...) { synchronized(this) { ... }}
void withdraw(...) { synchronized(this) { ... }}
int balance(...) { synchronized(this) { ... }}
void transfer(Acct from, int amt) {
    synchronized(this) {
        synchronized(from) { //deadlock (still)
            if(from.balance()>=amt && amt < maxXfer) {
                from.withdraw(amt);
                this.deposit(amt);
            }
        }
    }
}
```
Code evolution

Having chosen “self-locking” today, hard to add a correct transfer method tomorrow

```java
void deposit(...) { atomic { ... } }
void withdraw(...) { atomic { ... } }
int balance(...) { atomic { ... } }
void transfer(Acct from, int amt) {
    //race
    if(from.balance()>=amt && amt < maxXfer) {
        from.withdraw(amt);
        this.deposit(amt);
    }
}
```
Code evolution

Having chosen “self-locking” today, hard to add a
correct transfer method tomorrow

```c
void deposit(...) { atomic { ... } }
void withdraw(...) { atomic { ... } }
int balance(...) { atomic { ... } }
void transfer(Acct from, int amt) {
    atomic {
        //correct
        if(from.balance()>=amt && amt < maxXfer){
            from.withdraw(amt);
            this.deposit(amt);
        }
    }
}
```
Lesson

Locks do not compose; transactions do
Today

Issues in language design and semantics
1. Transactions for software evolution
2. Transactions for strong isolation [Nov06]*
3. The need for a memory model [MSPC06a]**

Software-implementation techniques
1. On one core [ICFP05]
2. Without changing the virtual machine [MSPC06b]
3. Static optimizations for strong isolation [Nov06]*

* Joint work with Intel PSL
** Joint work with Manson and Pugh
“Weak” atomicity

Widespread misconception:

“Weak” atomicity violates the “all-at-once” property of transactions only when the corresponding lock code has a data race

(May still be a bad thing, but smart people disagree.)

Initially \( y = 0 \)

```java
atomic {
    y = 1;
    x = 3;
    y = x;
}
```

\( x = 2; \)

print(y); //1? 2?
It’s worse

This lock-based code is correct in Java

Initially \( \text{ptr.f} == \text{ptr.g} \)

\[
\text{sync(lk) \{ }
\text{r = ptr; }
\text{ptr = new C(); }
\text{assert(r.f==r.g); }
\]

(Example from [Rajwar/Larus] and [Hudson et al])
It’s worse

But every published weak-atomicity system allows the assertion to fail!

- Eager- or lazy-update

```plaintext
initially ptr.f == ptr.g

atomic {
  r = ptr;
  ptr = new C();
}
assert(r.f==r.g);

atomic {
  ++ptr.f;
  ++ptr.g;
}

(Example from [Rajwar/Larus] and [Hudson et al])
```
Lesson

“Weak” is worse than most think and sometimes worse than locks
Today

Issues in language design and semantics
1. Transactions for software evolution
2. Transactions for strong isolation [Nov06]*
3. The need for a memory model [MSPC06a]**

Software-implementation techniques
1. On one core [ICFP05]
2. Without changing the virtual machine [MSPC06b]
3. Static optimizations for strong isolation [Nov06]*

* Joint work with Intel PSL
** Joint work with Manson and Pugh
Relaxed memory models

Modern languages don’t provide sequential consistency
1. Lack of hardware support
2. Prevents otherwise sensible & ubiquitous compiler transformations (e.g., copy propagation)

One tough issue: When do transactions impose ordering constraints?
Can get “strange results” for bad code
– Need rules for what is “good code”

initially $x==y==0$

```c
x = 1;
y = 1;
r = y;
s = x;
assert(s>=r);//invalid
```
Ordering

Can get “strange results” for bad code
– Need rules for what is “good code”

\textbf{initially} \quad x==y==0

\begin{align*}
x &= 1; \\
sync(lk) &{} \\
y &= 1; \\
sync(lk) &{} //same\ lock \\
s &= x; \\
assert(s\geq r); //valid
\end{align*}
Ordering

Can get “strange results” for bad code
  – Need rules for what is “good code”

initially $x==y==0$

```c
x = 1;
atomic{}
y = 1;
r = y;
atomic{}
s = x;
assert(s>=r); // ???
```

If this is good code, existing STMs are wrong
Ordering

Can get “strange results” for bad code
  – Need rules for what is “good code”

initially $x==y==0$

```
x = 1;
atomic{z=1;}
y = 1;
r = y;
atomic{tmp=0*z;}
s = x;
assert(s>=r); //???
```

“Conflicting memory” a slippery ill-defined slope
It is unclear when transactions should be ordered, but languages need memory models.

Corollary: Could/should delay adoption of transactions in real languages.
Today

Issues in language design and semantics
1. Transactions for software evolution
2. Transactions for strong isolation [Nov06]*
3. The need for a memory model [MSPC06a]**

Software-implementation techniques
1. On one core [ICFP05]
2. Without changing the virtual machine [MSPC06b]
3. Static optimizations for strong isolation [Nov06]*

* Joint work with Intel PSL
** Joint work with Manson and Pugh
Interleaved execution

The “uniprocessor (and then some)” assumption:

*Threads communicating via shared memory don't execute in “true parallel”*

Important special case:

* Uniprocessors still exist
* Many language implementations assume it (e.g., OCaml, DrScheme)
* Multicore may assign one core to an application
Uniprocessor implementation

- Execution of an atomic block logs updates
  - No overhead outside transaction nor for reads nor for initialization writes
- If scheduler preempts midtransaction, rollback
  - Else commit is trivial
- Duplicate code to avoid logging overhead outside transactions
  - Closures/objects need double code pointers
- Smooth interaction with GC
  - The log is a root
  - No need to log/rollback the GC (unlike hardware)
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>

• Rare rollback
Lesson

Implementing (strong) atomicity in software for a uniprocessor is so efficient it deserves special-casing

Note: The O/S and GC special-case uniprocessors too
Today

Issues in language design and semantics
1. Transactions for software evolution
2. Transactions for strong isolation [Nov06]*
3. The need for a memory model [MSPC06a]**

Software-implementation techniques
1. On one core [ICFP05]
2. Without changing the virtual machine [MSPC06b]
3. Static optimizations for strong isolation [Nov06]*

* Joint work with Intel PSL
** Joint work with Manson and Pugh
System Architecture

Our ‘run-time’

javac

class files

Note: Preserves separate compilation

Polyglot extensible compiler

Our compiler

foo.java

AThread.java

…

Our “run-time”

Our compiler

Polyglot extensible compiler

javac

class files

Note: Preserves separate compilation
Key pieces

• A field read/write first *acquires ownership* of object
• *Polling* for releasing ownership
  – Transactions rollback before releasing
• In transaction, a write also *logs the old value*
• Read/write barriers via method calls
  (JIT can inline them later)
• Some Java cleverness for efficient logging
• Lots of 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)
        if (owner == currentThread())
            return;
        ... // complicated slow-path
        return;
    }
}
```

- Synchronization only when contention
- With “owner=currentThread()” in constructor, thread-local objects *never* incur synchronization
Transactions for high-level programming languages do not need low-level implementations.

But good performance often needs parallel readers, which is future work. 😞
Today

Issues in language design and semantics
1. Transactions for software evolution
2. Transactions for strong isolation [Nov06]*
3. The need for a memory model [MSPC06a]**

Software-implementation techniques
1. On one core [ICFP05]
2. Without changing the virtual machine [MSPC06b]
3. Static optimizations for strong isolation [Nov06]*

* Joint work with Intel PSL
** Joint work with Manson and Pugh
**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>
Optimizing away barriers

Thread local

Not accessed in transaction

Immutable

New: static analysis for not-accessed-in-transaction …
Experimental Setup

UW: static analysis using whole-program pointer analysis
• Scalable (context- and flow-insensitive) using Paddle/Soot

Intel PSL: high-performance strong STM via compiler and run-time
• StarJIT
  – IR and optimizations for transactions and isolation barriers
  – Inlined isolation barriers
• ORP
  – Transactional method cloning
  – Run-time optimizations for strong isolation
• McRT
  – Run-time for weak and strong STM
Benchmarks

Tsp

![Graph showing benchmark results for different thread counts and options.](image-url)
Benchmarks

![Graph showing performance comparison of different locking mechanisms and compilation options across varying numbers of threads. The y-axis represents the average time per 10,000 operations in seconds, and the x-axis represents the number of threads (1, 2, 4, 8, and 16). Bars are labeled for Synch, Weak Atom, Strong Atom No Opts, +JIT Opts, +DEA, and +Static Opts. The JBB notation is also present.]
Lesson

The cost of strong isolation is in nontransactional barriers and compiler optimizations help a lot

Note: The first high-performance strong software transaction implementation for a multiprocessor
Credit

Uniprocessor: Michael Ringenburg
Source-to-source: Benjamin Hindman (undergrad)
Barrier-removal: Steve Balensiefer, Kate Moore

Memory-model issues: Jeremy Manson, Bill Pugh
High-performance strong STM: Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha

wasp.cs.washington.edu
Lessons

1. Locks do not compose; transactions do
2. “Weak” is worse than most think and sometimes worse than locks
3. It is unclear when transactions should be ordered, but languages need memory models

4. Implementing atomicity in software for a uniprocessor is so efficient it deserves special-casing
5. Transactions for high-level programming languages do not need low-level implementations
6. The cost of strong isolation is in nontransactional barriers and compiler optimizations help a lot