The Why, What, and How of Software Transactions for More Reliable Concurrency

Dan Grossman
University of Washington

17 November 2006
Atomic

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

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

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

lock acquire/release

(behave as if)

no interleaved computation;

no unfair starvation
Why now?

You are unleashing small-scale parallel computers on the programming masses

Threads and shared memory remaining a key model
  – Most common if not the best

Locks and condition variables not enough
  – Cumbersome, error-prone, slow

Transactions should be a hot area, and it is...
A big deal

Software-transactions research broad…

• Programming languages
  PLDI, POPL, ICFP, OOPSLA, ECOOP, HASKELL, …

• Architecture
  ISCA, HPCA, ASPLOS, MSPC, …

• Parallel programming
  PPoPP, PODC, …

… and coming together
  TRANSACT (at PLDI06)
Viewpoints

Software transactions good for:
• Software engineering (avoid races & deadlocks)
• Performance (optimistic “no conflict” without locks)

Research should be guiding:
• New hardware with transactional support
• Inevitable software support
  – Legacy/transition
  – Semantic mismatch between a PL and an ISA
  – May be fast enough
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-implemention 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
Atomic

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

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

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

lock acquire/release

(behave as if)

no interleaved computation;
no unfair starvation
Code evolution

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);
        }
    }
}
```
Code evolution

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

```c
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 (for any field maxXfer)
        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

Common belief:

• “Weak” means nontransactional code can interpose reads/writes with transactions
• Same bugs arise with lock-based code
• Strict segregation of transactional vs. non-transactional data sufficient to avoid races

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

Initially \(y == 0\)

```java
x = 2;
print(y); // 1? 2?
```

Joint work with Intel PSL
Segregation

Segregation is not necessary in lock-based code
  – Even under relaxed memory models

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

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

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

Joint work with Intel PSL
“Weak” atomicity redux

“Weak” really means nontransactional code bypasses the transaction mechanism…

Weak STMs violate isolation on example:
• Eager-updates (one update visible before abort)
• Lazy-updates (one update visible after commit)

Imposes correctness burdens on programmers that locks do not

Joint work with Intel PSL
Lesson

“Weak” is worse than most think; it can require segregation where locks do not

Corollary: “Strong” has easier semantics
– especially for a safe language

Joint work with Intel PSL
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)

So safe languages need two complicated definitions
1. What is “properly synchronized”?
2. What can compiler and hardware do with “bad code”?
(Unsafe languages need (1))

A flavor of simplistic ideas and the consequences…

Joint work with Manson, Pugh
Simplistic ideas

“Properly synchronized” ➔ All thread-shared mutable memory accessed in transactions

Consequence: *Data-handoff* code deemed “bad”

```
//Producer
tmp1=new C();
tmp1.x=42;
atomic {
    q.put(tmp1);
}

//Consumer
atomic {
    tmp2=q.get();
}
tmp2.x++;
```

*Joint work with Manson, Pugh*
Simplistic ideas

There exists a total “happens-before” order among all transactions

Consequence: atomic has barrier semantics, making dubious code correct

initially $x==y==0$

$x = 1;$
$y = 1;$
$r = y;$
$s = x;$
\[
\text{assert}(s>=r); //invalid
\]

Joint work with Manson, Pugh
Simplistic ideas

There exists a total “happens-before” order among all transactions.

Consequence: atomic has barrier semantics, making dubious code correct and real implementations wrong.

Initially $x==y==0$

```c
x = 1;
atomic { }
y = 1;
```

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

Joint work with Manson, Pugh
Simplistic ideas

There exists a total “happens-before” order among transactions with conflicting memory accesses.

Consequence: “memory access” now in the language definition; dead-code elim must be careful.

Initially $x == y == 0$

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

Joint work with Manson, Pugh
Lesson

It is not clear when transactions are ordered, but languages need memory models

Corollary: This could/should delay adoption of transactions in well-specified languages

Shameless provocation:
architectures need memory models too! (Please?!)

Joint work with Manson, Pugh
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
- Multicore may assign one core to an app
- Many concurrent apps don’t need a multiprocessor (e.g., a document editor)
- Many language implementations assume it (e.g., OCaml, DrScheme)
Implementing atomic

Key pieces:

• Execution of an atomic block logs writes

• If scheduler pre-empts a thread in atomic, rollback the thread

• Duplicate code so non-atomic code is not slowed by logging

• Smooth interaction with GC
Logging example

```c
int x=0, y=0;
void f() {
    int z = y+1;
    x = z;
}
void g() {
    y = x+1;
}
void h() {
    atomic {
        y = 2;
        f();
        g();
    }
}
```

Executing atomic block:
- build LIFO log of old values:

```
y:0  z:?  x:0  y:2
```

Rollback on pre-emption:
- Pop log, doing assignments
- Set program counter and stack to beginning of atomic

On exit from atomic:
- drop log
Logging efficiency

Keep the log small:
- Don’t log reads (key uniprocessor advantage)
- Need not log memory allocated after atomic entered
  - Particularly *initialization writes*
- Need not log an address more than once
  - To keep logging fast, switch from array to hashtable after “many” (50) log entries
Duplicating code

Duplicate code so callees know to log or not:

- For each function \( f \), compile \( f_{\text{atomic}} \) and \( f_{\text{normal}} \)
- Atomic blocks and atomic functions call atomic functions
- Function pointers compile to pair of code pointers

```c
int x=0, y=0;
void f() {
    int z = y+1;
    x = z;
}
void g() {
    y = x+1;
}
void h() {
    atomic {
        y = 2;
        f();
        g();
    }
}
```
Representing closures/objects

Representation of function-pointers/closures/objects an interesting (and pervasive) design decision

OCaml:

```
header code ptr free variables...
```

```
add 3, push, ...
```
Representing closures/objects

Representation of function-pointers/closures/objects
an interesting (and pervasive) design decision

One approach: bigger closures

Note: atomic is first-class, so it is just one of these too!
Representing closures/objects

Representation of function-pointers/closures/objects
an interesting (and pervasive) design decision

Alternate approach: slower calls in atomic

Note: Same overhead as OO dynamic dispatch
Interaction with GC

What if GC occurs mid-transaction?
• The log is a root (in case of rollback)
• Moving objects is fine
  – Rollback produces *equivalent* state
  – Naïve hardware solutions may log/rollback GC!

What about rolling back the allocator?
• Don’t bother: after rollback, objects allocated in transaction are unreachable
  – Naïve hardware solutions may log/rollback initialization writes!
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: Don’t run other multicore services on a uni either
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

Note: Preserves separate compilation
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
- **Polling** for releasing ownership
  - Transactions rollback before releasing
- 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)
}
```

Field accesses become method calls

- Read/write barriers that acquire ownership
  - Then do the read or write
  - In transaction, log between acquire and write
- Calls simplify/centralize code (JIT will inline)
Read-barrier

//field in class C
D x;

static D get_x(C o) {
  o.acq();
  return o.x;
}
//also two setters

//some field-read
e.x

D x;
C.get_x(e)
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
Releasing ownership

- Must “periodically” check “to release” set
  - If in transaction, first rollback
    - Retry later (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,
  …
Lesson

Transactions for high-level programming languages do not need low-level implementations

But good performance does tend to need 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>

*Joint work with Intel PSL*
Optimizing away barriers

Thread local

Not accessed in transaction

Immutable

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

Joint work with Intel PSL
Not-accessed-in-transaction

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>no atomic write</td>
<td>atomic write</td>
</tr>
<tr>
<td>read</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>write</td>
<td>none</td>
<td>some</td>
</tr>
</tbody>
</table>

Yet another client of pointer-analysis

Joint work with Intel PSL
Analysis details

- Whole-program, context-insensitive, flow-insensitive
  - Scalable, but needs whole program
- Can be done before method duplication
  - Keep lazy code generation without losing precision
- Given pointer information, just two more passes
  1. How is an “abstract object” accessed transactionally?
  2. What “abstract objects” might a non-transactional access use?

Joint work with Intel PSL
### Static counts

Not the point, but good evidence
- Usually better than thread-local analysis

<table>
<thead>
<tr>
<th>App</th>
<th>Access</th>
<th>Total</th>
<th>NAIT or TL</th>
<th>NAIT only</th>
<th>TL only</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpecJVM98</td>
<td>Read</td>
<td>12671</td>
<td>12671</td>
<td>8796</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>9885</td>
<td>9885</td>
<td>7961</td>
<td>0</td>
</tr>
<tr>
<td>Tsp</td>
<td>Read</td>
<td>106</td>
<td>93</td>
<td>89</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>36</td>
<td>17</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>JBB</td>
<td>Read</td>
<td>804</td>
<td>798</td>
<td>364</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>621</td>
<td>575</td>
<td>131</td>
<td>344</td>
</tr>
</tbody>
</table>

Joint work with Intel PSL
Experimental Setup

High-performance strong STM from Intel PSL

- 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

Joint work with Intel PSL
Benchmarks

Tsp

Joint work with Intel PSL
Benchmarks

JBB

Joint work with Intel PSL
Lesson

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

1. Locks do not compose; transactions do
2. “Weak” is worse than most think; can require segregation where locks do not
3. Unclear when transactions are ordered, but languages need memory models
4. Strong 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
Related work

Work at UW complements other pieces of the puzzle…

• Efficient transaction “engines” in hw, sw, hybrid

• Semantics of closed, open, parallel nesting

• Irrevocable actions (e.g., I/O)
  – We provide and use a pragmatic transaction-aware foreign-function interface [ICFP05]
Credit

Uniprocessor: Michael Ringenburg
Source-to-source: Benjamin Hindman
Barrier-removal: Steven 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