Summer School on
Language-Based Techniques for
Concurrent and Distributed Software

Software Transactions: Software Implementations

Dan Grossman
University of Washington
18 July 2006

Atomic

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

```
withLk:
 lock->(unit->α)->α

let xfer src dst x =
withLk src.lk (fun()->
withLk dst.lk (fun()->
 src.bal <- src.bal-x;
 dst.bal <- dst.bal+x
))
```

```
atomic:
 (unit->α)->α

let xfer src dst x =
atomic (fun()->
 src.bal <- src.bal-x;
 dst.bal <- dst.bal+x
)
```

lock acquire/release

(behave as if)
no interleaved computation

Implementation issues

- How to start, commit, and abort a transaction
- How to do a read/write in a transaction
- How to do a read/write outside a transaction
- How to detect and/or avoid *conflicts*
  - How *optimistically* (go now, maybe abort later)
- What *granularity* to use for conflicts
- What about "really long" transactions?

Will mostly skim over important details:
- Obstruction-free?
- Support for strong atomicity?

Our plan

- Atomicity on a *uniprocessor* (AtomCaml)
- Sketch seminal language work: Harris/Fraser's WSTM
  - Optimistic reads and writes
  - More recent RSTM is faster (Dwarkadas lecture 3)
- Sketch more recent approaches: PLDI06
  - Optimistic reads, pessimistic writes
- Optimizations to avoid read/write overhead
  - Particularly strong atomicity

Interleaved execution

The "uniprocessor" assumption:

Threads communicating via shared memory don't execute in "true parallel"

Important special case:
- Many language implementations assume it (e.g., OCaml)
- Many concurrent apps don't need a multiprocessor (e.g., a document editor)
- Uniprocessors are dead? Where's the funeral?
- The O/S may give an app one core (for a while)

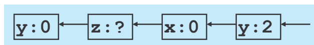
Implementing atomic

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

```
let x = ref 0
let y = ref 0
let f() =
 let z =
  ref((!y)+1)
 in
  x := !z
let g() =
 y := (!x)+1
let h() =
 atomic(fun()->
   y := 2;
   f();
   g())
```

- Executing atomic block in `h` builds a LIFO log of old values:

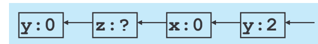


Rollback on pre-emption:

- Pop log, doing assignments
- Set program counter and stack to beginning of atomic

On exit from atomic: drop log

Logging efficiency



Keeping the log small:

- Don't log reads (key uniprocessor optimization)
- 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

```
let x = ref 0
let y = ref 0
let f() =
 let z =
  ref((!y)+1)
 in
  x := !z;
let g() =
 y := (!x)+1
let h() =
 atomic(fun()->
   y := 2;
   f();
   g())
```

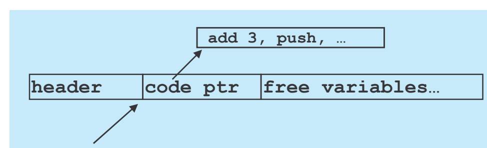
Duplicate code so callees know to log or not:

- For each function `f`, compile `f_atomic` and `f_normal`
- Atomic blocks and atomic functions call atomic functions
- Function pointers compile to pair of code pointers

Representing closures/objects

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

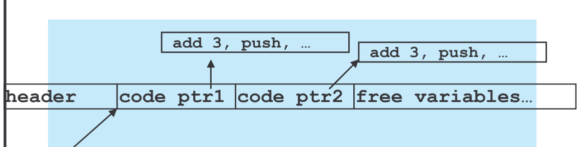
OCaml:



Representing closures/objects

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

AtomCaml: bigger closures

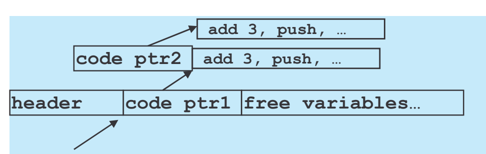


Note: atomic is first-class, so it is one of these too!

Representing closures/objects

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

AtomCaml alternative: slower calls in `atomic`



Note: Same overhead as OO dynamic dispatch

Interaction with GC

What if GC occurs mid-transaction?

- Pointers in log are roots (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

Qualitative evaluation

Strong atomicity for Caml at little cost

- Already assumes a uniprocessor

- Mutable data overhead

	not in atomic	in atomic
read	none	none
write	none	log (2 more writes)

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

Performance

Cost of synchronization is all in the noise

- Microbenchmark: *short* atomic block 2x slower than same block with lock-acquire/release
  - Longer atomic blocks = less slowdown
  - Programs don't spend all time in critical sections
- PLANet: 10% faster to 7% slower (noisy)
  - Closure representation mattered for only 1 test
- Sequential code (e.g., compiler)
  - 2% slower when using bigger closures

See paper for (boring) tables

Our plan

- Atomicity on a *uniprocessor* (AtomCaml)
- Sketch seminal language work: Harris/Fraser's WSTM
  - Optimistic reads and writes
  - More recent RSTM is faster (Dwarkadas lecture 3)
- Sketch more recent approaches: PLDI06
  - Optimistic reads, pessemistic writes
- Optimizations to avoid read/write overhead
  - Particularly strong atomicity

The Set-Up

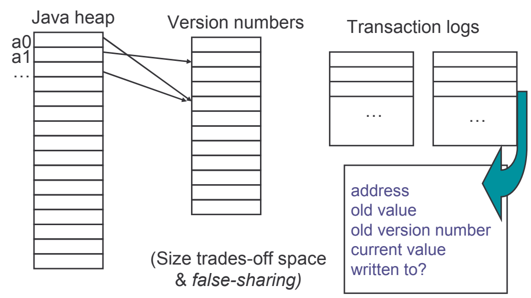
Caveats:

- Some simplifications (lies & omissions)
- Weak atomicity only

Key ideas:

- For every word, there exists a version number
- Transactions don't update memory or version numbers until commit
  - Must consult/update thread-local log

Memory picture



Most operations easy

- Read/write outside transaction: no change
- Read/write inside transaction: consult/modify log
- Abort: drop log, reset control
- Start: new log
- Commit: hmm… *Conceptually*
  - If any version # in log is out-of-date, abort
  - Else do all the updates, incrementing version #s for writes

Nice properties: parallel reads don't cause aborts, no synchronization until commit

Ah, commit

All the fanciness to allow parallel commits
- Scalable parallelism must avoid a "choke point"

Simple version:
- Replace *all* relevant version #s with thread-id
  - (low-order bit to distinguish)
- (Change to read/write: abort if find a thread-id)
- Then update heap values
- Then write back (new) version #s

Commit point:
The last change from version # to thread-id

Actually…

Many TM implementations, WSTM included, are *obstruction-free*:

Any thread can make progress in absence of contention (even if another thread dies/gets-unscheduled)

So we "can't" wait for a version # to return
- Instead, go into the log and get the "right" value
- Old value if before commit point
- New value if after commit point

Algorithm similar to multiword CAS from single CAS

Optimism

This algorithm has *optimistic* reads and writes
- Just get the data and version #
- Hope it won't get bumped before commit
- Else abort

(Backoff to avoid livelock)

But there's actually a subtle problem…

Hint: A bound-to-fail transaction may be operating on an inconsistent view of the world

Needing validate

```
     // x and y start 0
atomic {        atomic {
  if(x!=y)        ++x;
   for(;;) ;      ++y;
}               }
```


Needing validate

```
     // x and y start 0
atomic {        atomic {
  if(x!=y)        ++x;
   for(;;) ;      ++y;
}               }
```

Punch-line: Can't wait until end to abort, if you might never get there due to need to abort

Fix: Periodically *validate*: check that you could commit, but do not commit

Our plan

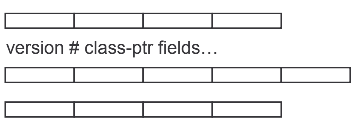
- Atomicity on a *uniprocessor* (AtomCaml)
- Sketch seminal language work: Harris/Fraser's WSTM
  - Optimistic reads and writes
  - More recent RSTM is faster (Dwarkadas lecture 3)
- Sketch more recent approaches: PLDI06
  - Optimistic reads, pessimistic writes
- Optimizations to avoid read/write overhead
  - Particularly strong atomicity

Some better trade-offs

- Better to update memory in place
  - No log-lookup on read/write
  - Worth giving up obstruction-freedom
- Better to keep version #s "nearby"
  - Better caching, but avoid space blow-up
  - One version # for objects' fields
- Obstruction-freedom not necessary (debatable)
  - Optimistic reads, pessimistic writes
  - Rollback (cf. AtomCaml) on abort

Memory picture (simplified)

Objects



Transaction logs:

- Read log: object-address, version #
- Write log: object-address, old-value, version #

Most operations easy

- Read inside transaction: adjust read log
  - Grab the current version #
- Write inside transaction: get ownership, adjust write log
  - abort if version # is another transaction-id
- Abort: rollback, reset control
- Commit: hmm
  - If any version # in log is out-of-date, abort
  - Else do all the updates, incrementing version #s for writes
  - Easy: Already own everything

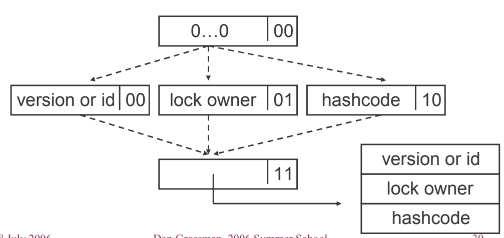
Nice property: parallel reads don't cause aborts

Some details

- Version # wraparound an A-B-A problem:
  - Check on commit unsound (value may be wrong)
  - Fix: Once every 2^29 transactions, validate all active transactions (abuse GC)

Some details

- Avoid extra 32-bits per object
  - Use same word for hashcode and lock
  - Modern version of an old trick…



Our plan

- Atomicity on a *uniprocessor* (AtomCaml)
- Sketch seminal language work: Harris/Fraser's WSTM
  - Optimistic reads and writes
  - More recent RSTM is faster (Dwarkadas lecture 3)
- Sketch more recent approaches: PLDI06
  - Optimistic reads, pessimistic writes
- Optimizations to avoid read/write overhead
  - Particularly strong atomicity

Strong performance problem

Recall AtomCaml overhead:

	not in atomic	in atomic
read	none	none
write	none	some

In general, with parallelism:

	not in atomic	in atomic
read	none iff weak	some
write	none iff weak	some

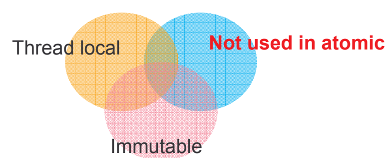
Start way behind in performance, especially in imperative languages (cf. concurrent GC)

AtomJava

Novel prototype recently completed

- Source-to-source translation for Java
  - Run on any JVM (so parallel)
  - At VM's mercy for low-level optimizations
- Atomicity via locking (object ownership)
  - Poll for contention and rollback
  - No support for parallel readers yet ☹
- Hope whole-program optimization can get "strong for near the price of weak"

Optimizing away barriers



Want static (no overhead) and dynamic (less overhead)
Contributions:

- Dynamic thread-local: never release ownership until another thread asks for it (avoid synchronization)
- Static not-used-in-atomic…

Not-used-in-atomic

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

	not in atomic			in atomic
	no atomic access	no atomic write	atomic write	
read	none	none	some	some
write	none	some	some	some

Analysis sketch

This is a novel use of conventional analysis results:

0 (At least conceptually) do code-duplication so each new, read, and write is "in-atomic" or "not-in-atomic"

1. For each read/write, compute (approximation of) which news could have produced the object whose field is being accessed.
   - Classic pointer-analysis problem
   - See Foster's lecture
2. In one pass over "atomic" code, use results of (1) to compute in-atomic access for each new
3. In one pass over "non-atomic" code, use results of (2) to compute whether a barrier is needed

Theses/conclusions

1. Atomicity is better than locks, much as garbage collection is better than malloc/free [Tech Rpt Apr06]
2. “Strong” atomicity is key, preferably w/o language restrictions
3. With 1 thread running at a time, strong atomicity is fast and elegant [ICFP Sep05]
4. With multicore, strong atomicity needs heavy compiler optimization; we’re making progress [Tech Rpt May06]