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

Dan Grossman
University of Washington

26 May 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

Why now?

Multicore 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

Atomicity should be a hot area, and it is…

A big deal

Software-transactions research broad…

- Programming languages
  PLDI 3x, POPL, ICFP, OOPSLA, ECOOP, HASKELL
- Architecture
  ISCA, HPCA, ASPLOS
- Parallel programming
  PPoPP, PODC

… and coming together, e.g.,
TRANSACT & WTW at PLDI06

Viewpoints

Software transactions good for:

- Software engineering (avoid races & deadlocks)
- Performance (optimistic “no conflict” without locks)

key semantic decisions depend on emphasis

Research should be guiding:

- New hardware with transactional support
- Language implementation for expected platforms

“is this a hw or sw question or both”

Our view

SCAT (Scalable Concurrency Abstractions via Transactions) project at UW is motivated by

"reliable concurrent software without new hardware"

Theses:

1. Atomicity is better than locks, much as garbage collection is better than malloc/free [Tech Rpt Apr06]
2. "Strong" atomicity is key, with minimal 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]

Outline

- Motivation
  - Case for strong atomicity
  - The GC analogy
- Related work
- Atomicity for a functional language on a uniprocessor
- Optimizations for strong atomicity on multicore
- Conclusions

Atomic, again

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
)
```

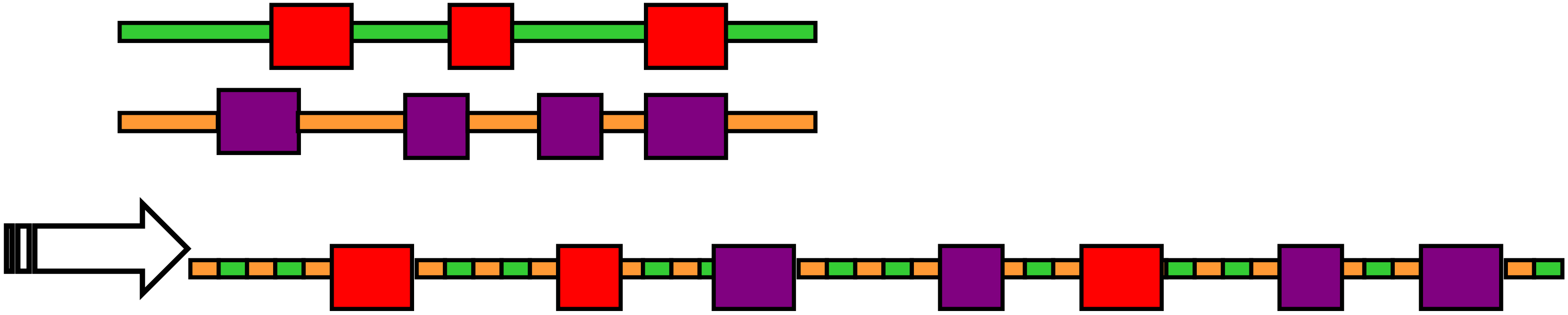
lock acquire/release

(behave as if)
no interleaved computation

Strong atomicity

(behave as if) no interleaved computation

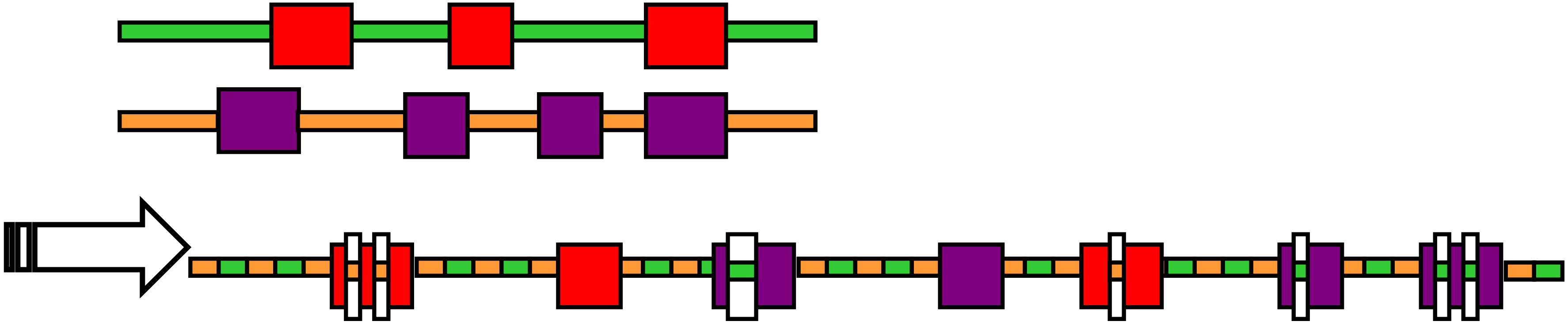
- Before a transaction "commits"
  - Other threads don't "read its writes"
  - It doesn't "read other threads' writes"
- This is just the semantics
  - Can interleave more unobservably



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



Wanting strong

Software-engineering advantages of strong atomicity

1. Sequential reasoning in transaction
   - Strong: sound
   - Weak: only if all (mutable) data is not simultaneously accessed outside transaction
2. Transactional data-access a local code decision
   - Strong: new transaction “just works”
   - Weak: what data “is transactional” is global
3. Fairness: Long transactions don’t starve others
   - Strong: true; no other code sees effects
   - Weak: maybe false for non-transactional code

Caveat

Need not *implement* strong atomicity to get it

With weak atomicity, suffices to put all mutable thread-shared data accesses in transactions

Can do so via

- "Programmer discipline"
- Monads [Harris, Peyton Jones, et al]
- Program analysis [Flanagan, Freund et al]
- "Transactions everywhere" [Leiserson et al]
- …

Outline

- Motivation
  - Case for strong atomicity
  - The GC analogy
- Related work
- Atomicity for a functional language on a uniprocessor
- Optimizations for strong atomicity on multicore
- Conclusions

Why an analogy

- Already gave some of the crisp technical reasons why atomic is better than locks
  - Locks are weaker than weak atomicity
- An analogy isn't logically valid, but can be
  - Convincing and memorable
  - Research-guiding

Software transactions are to concurrency as garbage collection is to memory management

Hard balancing acts

memory management

correct, small footprint?

- free too much:
  dangling ptr
- free too little:
  leak, exhaust memory

non-modular

- deallocation needs "whole-program is done with data"

concurrency

correct, fast synchronization?

- lock too little:
  race
- lock too much:
  sequentialize, deadlock

non-modular

- access needs "whole-program uses same lock"

Move to the run-time

- Correct [manual memory management / lock-based synhronization] requires subtle whole-program invariants

- [Garbage-collection / software-transactions] also requires subtle whole-program invariants, but localized in the run-time system
  - With compiler and/or hardware cooperation
  - Complexity doesn't increase with size of program

Old way still there

Despite being better, “stubborn” programmers can nullify most of the advantages

```
type header = int
let t_buf : (t *(bool ref) array =
 …(*big array of ts and false refs*)
let mallocT () : header * t =
 let i = … (*find t_buf elt with false *)in
 snd t_buf[i] := true;
 (i,fst t_buf[i])
let freeT (i:header,v:t) =
 snd t_buf[i] := false
```

Old way still there

Despite being better, “stubborn” programmers can nullify most of the advantages

```
type lk = bool ref
let new_lk = ref true
let rec acquire lk =
 let done = atomic (fun () ->
             if !lk
             then (lk:=false;true)
             else false) in
 if done then () else acquire lk
let release lk = lk:=true
```

Much more

More similarities:

- Basic trade-offs
  - Mark-sweep vs. copy
  - Rollback vs. private-memory
- I/O (writing pointers / mid-transaction data)
- …

I now think “analogically” about each new idea!

Outline

- Motivation
  - Case for strong atomicity
  - The GC analogy
- Related work
- Atomicity for a functional language on a uniprocessor
- Optimizations for strong atomicity on multicore
- Conclusions

Related work, part 1

- Transactions a classic CS concept
- Software-transactional memory (STM) as a library
  - Even weaker atomicity & less convenient
- Weak vs. Strong: [Blundell et al.]
- Efficient software implementations of weak atomicity
  - MSR and Intel (latter can do strong now)
- Hardware and hybrid implementations
  - Key advantage: Use cache for private versions
  - Atomos (Stanford) has strong atomicity
- Strong atomicity as a type annotation
  - Static checker for lock code

Closer related work

- Haskell GHC
  - Strong atomicity via STM Monad
  - So can't "slap atomic around existing code"
    - By design (true with all monads)
- Transactions for Real-Time Java (Purdue)
  - Similar implementation to AtomCaml
- Orthogonal language-design issues
  - Nested transactions
  - Interaction with exceptions and I/O
  - Compositional operators
  - …

Outline

- Motivation
- Related work
- Atomicity for a functional language on a uniprocessor
  - Language design
  - Implementation
  - Evaluation
- Optimizations for strong atomicity on multicore
- Conclusions

Basic design

no change to parser and type-checker

- **`atomic`** a first-class function
- Argument evaluated without interleaving

```
external atomic : (unit->α)->α = "atomic"
```

In atomic (dynamically):

- **`yield : unit->unit`** aborts the transaction
- **`yield_r : α ref->unit`** yield & rescheduling hint
  - Often as good as a guarded critical region
  - Better: split "ref registration" & yield
  - Alternate: *implicit read sets*

Exceptions

If code in atomic raises exception caught outside atomic, does the transaction abort?

We say no!

- atomic = "no interleaving until control leaves"
- Else atomic changes sequential semantics:

```
let x = ref 0 in
atomic (fun () -> x := 1; f())
assert((!x)=1) (*holds in our semantics*)
```

A variant of exception-handling that reverts state might be useful and share implementation

- But not about concurrency

Handling I/O

- Buffering sends (output) easy and necessary
- Logging receives (input) easy and necessary

- But input-after-output does not work

```
let f () =
 write_file_foo();
 …
 read_file_foo()

let g () =
  atomic f; (* read won't see write *)
  f()       (* read may   see write *)
```

- I/O one instance of native code …

Native mechanism

- Previous approaches: no native calls in **`atomic`**
  - raise an exception
  - **`atomic`** no longer preserves meaning
- We let the C code decide:
  - Provide 2 functions (in-atomic, not-in-atomic)
  - in-atomic can call not-in-atomic, raise exception, or do something else
  - in-atomic can *register* commit- & abort- actions (sufficient for buffering)
  - a pragmatic, imperfect solution (necessarily)

Outline

- Motivation
- Related work
- Atomicity for a functional language on a uniprocessor
  - Language design
  - Implementation
  - Evaluation
- Optimizations for strong atomicity on multicore
- Conclusions

Interleaved execution

The “uniprocessor” assumption:

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

Actually more general:
threads on different processors can pass messages

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?

Implementing atomic

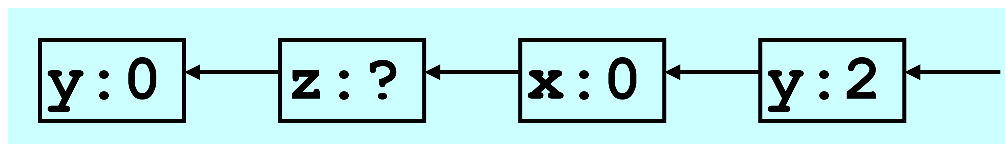
Key pieces:

- 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:

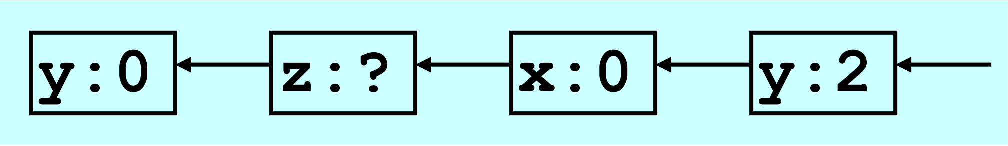


Rollback on pre-emption:

- 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())
```

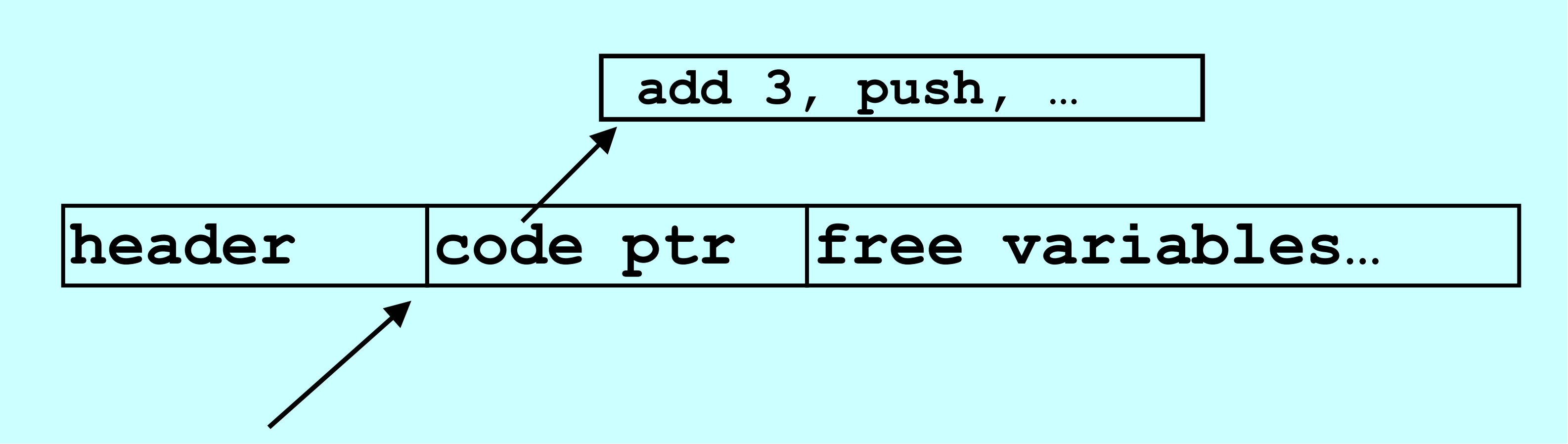
Duplicate code so callees know to log or not:

- 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

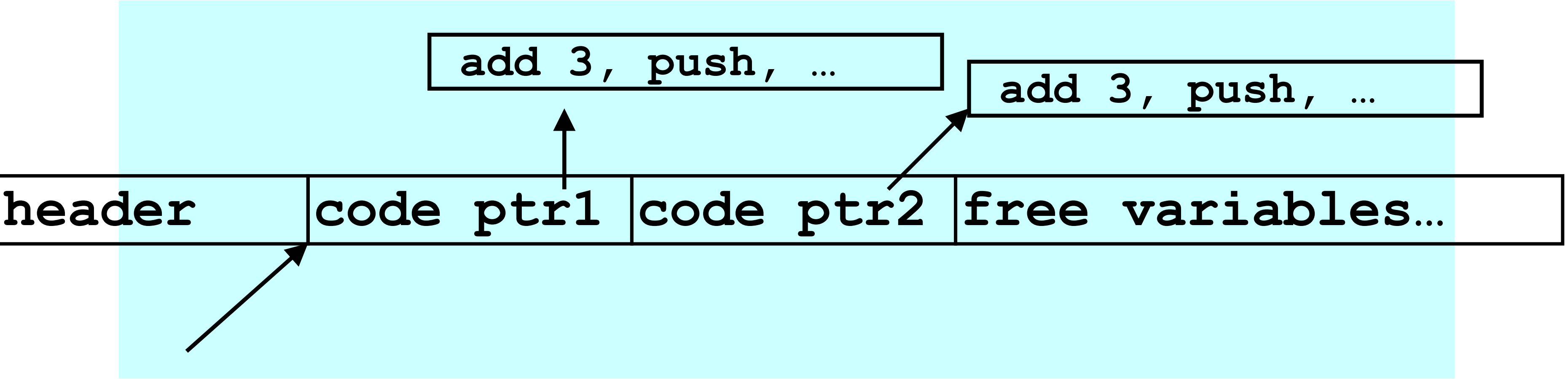
OCaml:



Representing closures/objects

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

AtomCaml: bigger closures

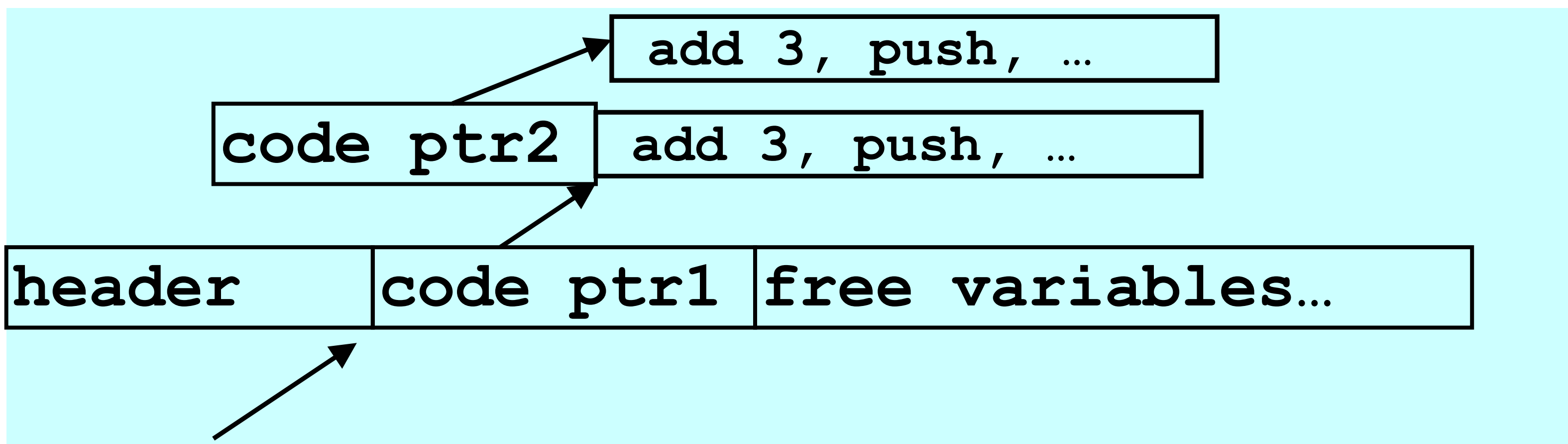


Note: atomic is first-class, so it is one of these too!

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

Outline

- Motivation
- Related work
- Atomicity for a functional language on a uniprocessor
  - Language design
  - Implementation
  - Evaluation
- Optimizations for strong atomicity on multicore
- Conclusions

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

PLANet program

Removed all locks from PLANet active-network simulator

- No large-scale structural changes
  - Condition-variable idioms via a 20-line library
- Found 3 concurrency bugs
  - 2 races in reader/writer locks library
  - 1 library-reentrancy deadlock (never triggered)
  - Turns out all implicitly avoided by atomic
- Dealt with 6 native calls in critical sections
  - 3: moved without changing application behavior
  - 3: used native mechanism to buffer output

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

Outline

- Motivation
  - Case for strong atomicity
  - The GC analogy
- Related work
- Atomicity for a functional language on a uniprocessor
- Optimizations for strong atomicity on multicore
- Conclusions

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

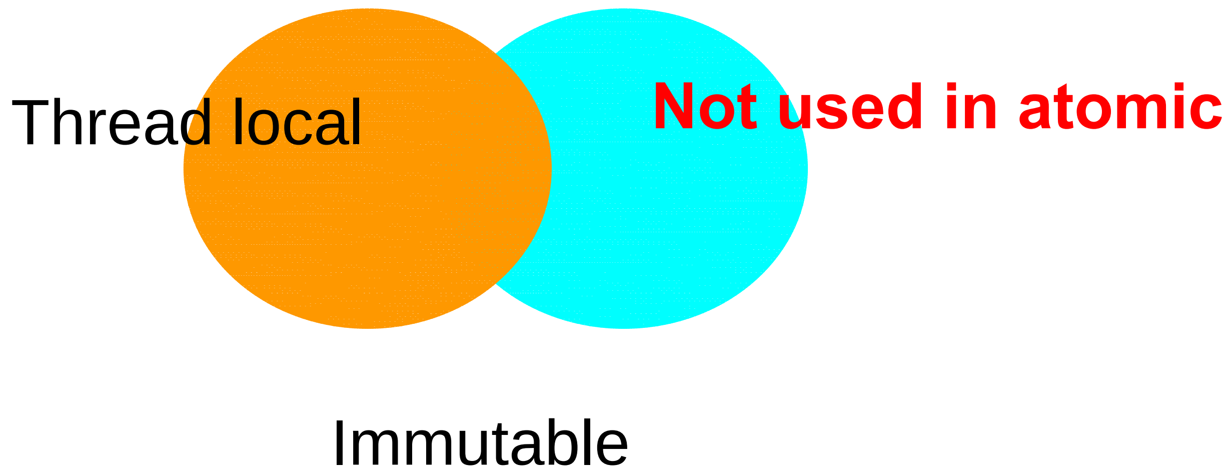
Start way behind in performance, especially in imperative languages (cf. concurrent GC)

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

“Type-based” alias analysis easily avoids many barriers:

- If field **f** never used in a transaction, then no access to field **f** requires barriers

Performance not there yet

- Some metrics give false impression
  - Removes barriers at most static sites
  - Removal speeds up programs almost 2x
- Must remove enough barriers to avoid sequentialization

Current results for TSP & no real alias analysis:

speedup over 1 processor

	lock code	weak	strong no-opt	strong opt
2 processors	1.7x	1.7x	1.7x	1.7x
8 processors	4.5x	2.7x	1.4x	1.5x

To do: Benchmarks, VM support, more optimizations

Outline

- Motivation
  - Case for strong atomicity
  - The GC analogy
- Related work
- Atomicity for a functional language on a uniprocessor
- Optimizations for strong atomicity on multicore
- Conclusions

Theses

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]

Credit and other

AtomCaml: Michael Ringenburg

AtomJava: Benjamin Hindman (B.S., Dec06)

Transactions are 1/4 of my current research

- Better type-error messages for ML: Benjamin Lerner
- Semi-portable low-level code: Marius Nita
- Cyclone (safe C-level programming)

More in the WASP group: wasp.cs.washington.edu

[Presentation ends here; additional slides follow]

Granularity

Previous discussion assumed "object-based" ownership

- Granularity may be too coarse (especially arrays)
  - False sharing
- Granularity may be too fine (object affinity)
  - Too much time acquiring/releasing ownership

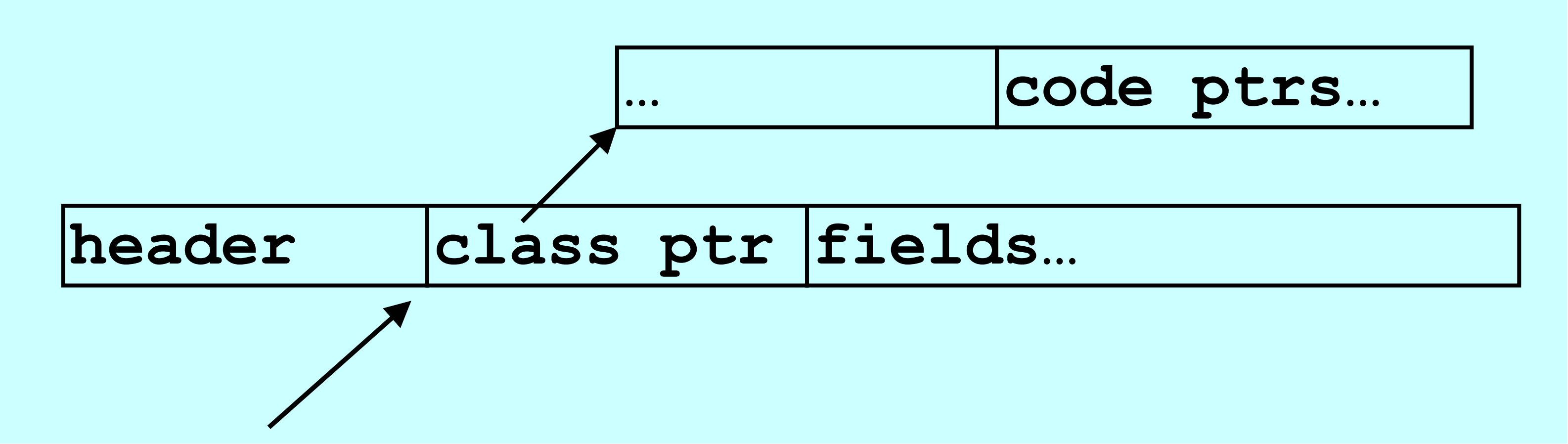
Conjecture: Profile-guided optimization can help

Note: Issue applies to weak atomicity too

Representing closures/objects

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

OO already pays the overhead atomic needs
(interfaces, multiple inheritance, … no problem)



Digression

Recall atomic a first-class function

- Probably not useful
- Very elegant

A Caml closure implemented in C

- Code ptr1: calls into run-time, then call thunk, then more calls into run-time
- Code ptr2: just calls thunk

Atomic

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

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

semantics:
lock acquire/release

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

semantics:
(behave as if)
no interleaved execution

No fancy hardware, code restrictions, deadlock, or unfair scheduling (e.g., disabling interrupts)

Common bugs

- Races
  - Unsynchronized access to shared data
  - Higher-level races: multiple objects inconsistent
- Deadlocks (cycle of threads waiting on locks)

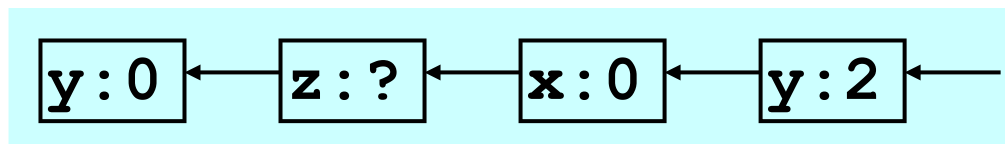
Example [JDK1.4, version 1.70, Flanagan/Qadeer PLDI2003]

```
synchronized append(StringBuffer sb) {
 int len = sb.length();
 if(this.count + len > this.value.length)
   this.expand(…);
 sb.getChars(0,len,this.value,this.count);
 …
}
// length and getChars are synchronized
```

Logging example

```
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 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

Why better

1. No whole-program locking protocols
   - As code evolves, use **`atomic`** with “any data”
   - Instead of “what locks to get” (races) and “in what order” (deadlock)
2. Bad code doesn’t break good atomic blocks:

```
let bad1() =
 acct.bal <- 123
let bad2() =
 atomic
 (fun()->«diverge»)
```

```
let good() =
atomic
(fun()->
 let tmp=acct.bal in
 acct.bal <- tmp+amt)
```

With atomic, “the protocol” is now the runtime’s problem (c.f. garbage collection for memory management)