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

Dan Grossman
University of Washington

8 September 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
(but no starvation)

Why now?

Multicore unleashing small-scale parallel computers on the programming masses

Threads and shared memory 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. 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)

key semantic decisions may depend on emphasis

Research should be guiding:

- New hardware support
- Language implementation with existing ISAs

“is this a hardware or software question or both”

Our view

SCAT (*) 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
2. “Strong” atomicity is key
3. If 1 thread runs at a time, strong atomicity is easy & fast
4. Else static analysis can improve performance

* (Scalable Concurrency Abstractions via Transactions)

Non-outline

Paper trail:

- Added to OCaml [ICFP05; Ringenburg]
- Added to Java via source-to-source [MSPC06; Hindman]
- Memory-model issues [MSPC06; Manson, Pugh]
- Garbage-collection analogy [TechRpt, Apr06]
- Static-analysis for barrier-removal
  [TBA; Balensiefer, Moore, Intel PSL]

Focus on UW work, happy to point to great work at

Sun, Intel, Microsoft, Stanford, Purdue, UMass, Rochester, Brown, MIT, Penn, Maryland, Berkeley, Wisconsin, …

Outline

- Why (local reasoning)
  - Example
  - Case for strong atomicity
  - The GC analogy
- What (tough semantic “details”)
  - Interaction with exceptions
  - Memory-model questions
- How (usually the focus)
  - In a uniprocessor model
  - Static analysis for removing barriers on an SMP

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
(but no starvation)

Code evolution

Having chosen "self-locking" yesterday,

hard to add a correct transfer method tomorrow

```
void deposit(…)  { synchronized(this) { … }}
void withdraw(…) { synchronized(this) { … }}
int  balance(…)  { synchronized(this) { … }}
void transfer(Acct from, int amt) {

   //race
   if(from.balance()>=amt) {
     from.withdraw(amt);
     this.deposit(amt);
   }

}
```

Code evolution

Having chosen “self-locking” yesterday,

hard to add a correct transfer method tomorrow

```
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) {
     from.withdraw(amt);
     this.deposit(amt);
   }
  }
}
```

Code evolution

Having chosen "self-locking" yesterday,

hard to add a correct transfer method tomorrow

```
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) {
     from.withdraw(amt);
     this.deposit(amt);
   }
  }}
}
```

Code evolution

Having chosen "self-locking" yesterday,

hard to add a correct transfer method tomorrow

```
void deposit(…)  { atomic { … }}
void withdraw(…) { atomic { … }}
int  balance(…)  { atomic { … }}

void transfer(Acct from, int amt) {

    //race
    if(from.balance()>=amt) {
      from.withdraw(amt);
      this.deposit(amt);
    }

}
```

Code evolution

Having chosen "self-locking" yesterday,

hard to add a correct transfer method tomorrow

```
void deposit(…)  { atomic { … }}
void withdraw(…) { atomic { … }}
int  balance(…)  { atomic { … }}

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

Moral

- Locks do not compose
  - Leads to hard-to-change design decisions
  - Real-life example: Java's StringBuffer

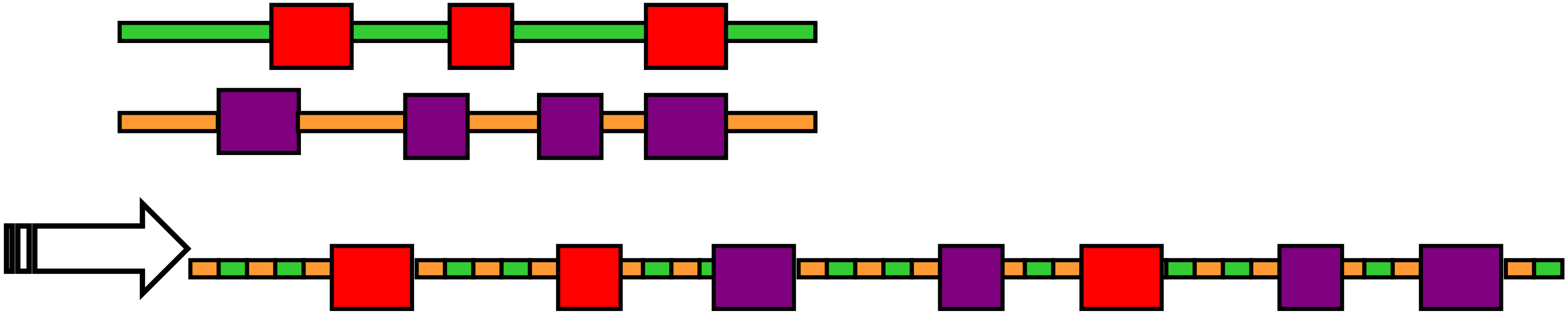
- Transactions have other advantages

- But we assumed "wrapping transfer in atomic" prohibited *all* interleavings…
  - **`transfer`** implemented with *local knowledge*

Strong atomicity

(behave as if) no interleaved computation

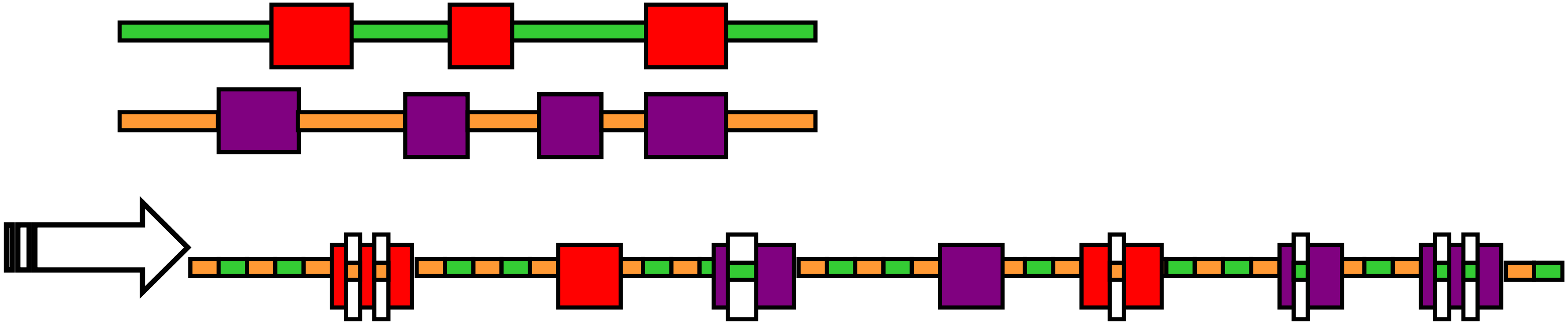
- Before a transaction "commits"
  - Other threads don't "read its writes"
  - It doesn't "read other threads' writes"
- This is just the semantics
  - Can interleave more unobservably



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. Local (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

Caveat

Need not *implement* strong atomicity to get it, given weak

For example:

Sufficient (but unnecessary) to ensure all mutable thread-shared data accesses are in transactions

Doable via:

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

Outline

- Why (local reasoning)
  - Example
  - Case for strong atomicity
  - The GC analogy
- What (tough semantic “details”)
  - Interaction with exceptions
  - Memory-model questions
- How (usually the focus)
  - In a uniprocessor model
  - Static analysis for removing barriers on an SMP

Why an analogy

- Already hinted at crisp technical reasons why atomic is better than locks
  - Locks weaker than weak atomicity
- Analogies aren't logically valid, but can be
  - Convincing
  - 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 synchronization] needs subtle whole-program invariants

- So does [Garbage-collection / software-transactions] but they are localized in the run-time system
  - Complexity doesn't increase with size of program
  - Can use compiler and/or hardware cooperation

Old way still there

Alas:

"stubborn" programmers can nullify many advantages

- GC: application-level object buffers
- Transactions: application-level locks…

```
class SpinLock {
  private boolean b = false;
  void acquire() {
    while(true)
      atomic {
       if(b) continue;
       b = true;
       return;
      }
  }
  void release() { atomic { b = false; }}
}
```

Much more

- 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

- Why (local reasoning)
  - Example
  - Case for strong atomicity
  - The GC analogy
- What (tough semantic “details”)
  - Interaction with exceptions
  - Memory-model questions
- How (usually the focus)
  - In a uniprocessor model
  - Static analysis for removing barriers on an SMP

Basic design

With higher-order functions, no need to change to parser and type-checker

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

```
external atomic : (unit->α)->α = “atomic”
```

In atomic (dynamically):

- **`retry : unit->unit`** causes abort-and-retry
- No point retrying until relevant state changes
  - Can view as an implementation issue

Exceptions

What if code in atomic raises an exception?

```
atomic { … f(); /* throws */ …}
```

Options:

1. Commit
2. Abort-and-retry
3. Abort-and-continue

Claim:

"Commit" makes the most semantic sense…

Abort-and-retry

Abort-and-retry does not preserve sequential behavior

- Atomic should be about restricting interleaving
- Exceptions are just an “alternate return”

```
atomic {throw new E();} //infinite loop?
```

Violates this *design goal*:

In a single-threaded program,
adding atomic has no observable behavior

“But I want abort-and-retry”

The abort-and-retry lobby says:

“in good code, exceptions indicate bad situations”

- That is not the semantics
- Can build abort-and-retry from commit, not vice-versa

```
atomic {
 try { … }
 catch(Throwable e) { retry; }
}
```

- Commit is the primitive; sugar for abort-and-retry fine

Abort-and-continue

Abort-and-continue has even more semantic problems

- “Abort is a blunt hammer, rolling back all state”
- Continuation needs “why it failed”, but cannot see state that got rolled back (integer error codes?)

```
Foo obj = new Foo();
atomic {
  obj.x = 42;
  f();//exception undoes unreachable state
}
assert(obj.x==42);
```

Outline

- Why (local reasoning)
  - Example
  - Case for strong atomicity
  - The GC analogy
- What (tough semantic “details”)
  - Interaction with exceptions
  - Memory-model questions
- How (usually the focus)
  - In a uniprocessor model
  - Static analysis for removing barriers on an SMP

Relaxed memory models

Modern languages don't provide sequential consistency

- Lack of hardware support
- Prevents otherwise sensible & ubiquitous compiler transformations (e.g., common-subexpression elim)

So safe languages need complicated definitions:

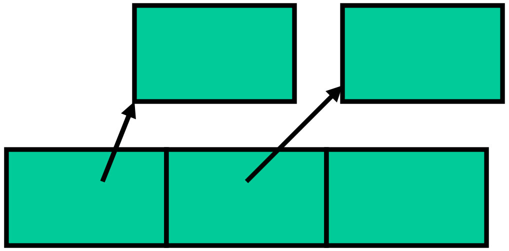
1. What is "properly synchronized"?
2. What "happens-before events" must compiler obey?

A flavor of simplistic ideas and the consequences…

Data-handoff okay?

"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++;
```

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

Happens-before

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;
assert(s>=r);//invalid
```

Happens-before

A total "happens-before" order among all transactions

Consequence: atomic has barrier semantics, making dubious code correct

```
initially x=y=0
```

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

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

Happens-before

A total "happens-before" order among transactions with conflicting memory accesses

Consequence: "memory access" now in the language definition; affects dead-code elimination

```
initially x=y=0
```

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

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

Outline

- Why (local reasoning)
  - Example
  - Case for strong atomicity
  - The GC analogy
- What (tough semantic “details”)
  - Interaction with exceptions
  - Memory-model questions
- How (usually the focus)
  - In a uniprocessor model
  - Static analysis for removing barriers on an SMP

Interleaved execution

The “uniprocessor (and then some)” assumption:

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

Important special case:

- Many language implementations assume it (e.g., OCaml, DrScheme)
- Many concurrent apps don’t need a multiprocessor (e.g., many user-interfaces)
- Uniprocessors still exist

Implementing atomic

Key pieces:

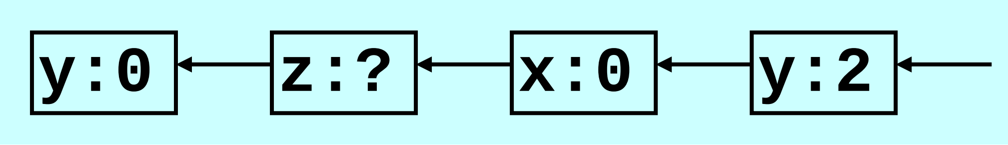
- Execution of an atomic block logs writes
- If scheduler pre-empts a thread in atomic, rollback the thread
- Duplicate code so non-atomic code is not slowed by logging
- Smooth interaction with GC

Logging example

```
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:



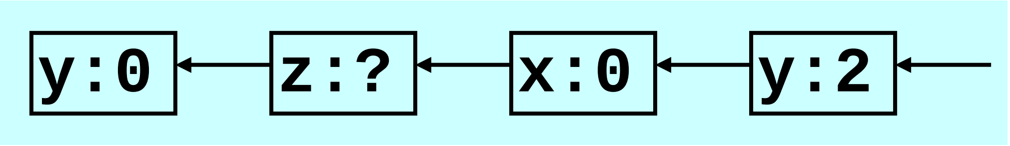
Rollback on pre-emption:

- 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 when log has “many” (50) entries

Code duplication

```
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();
  }
}
```

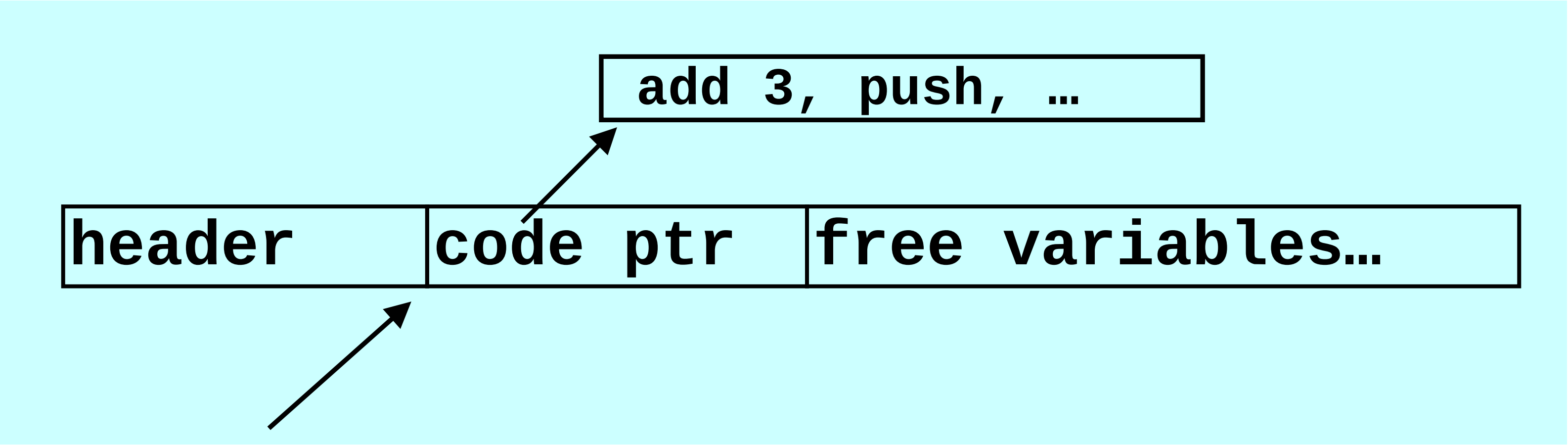
Duplicate code so callees know to log or not:

- 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

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

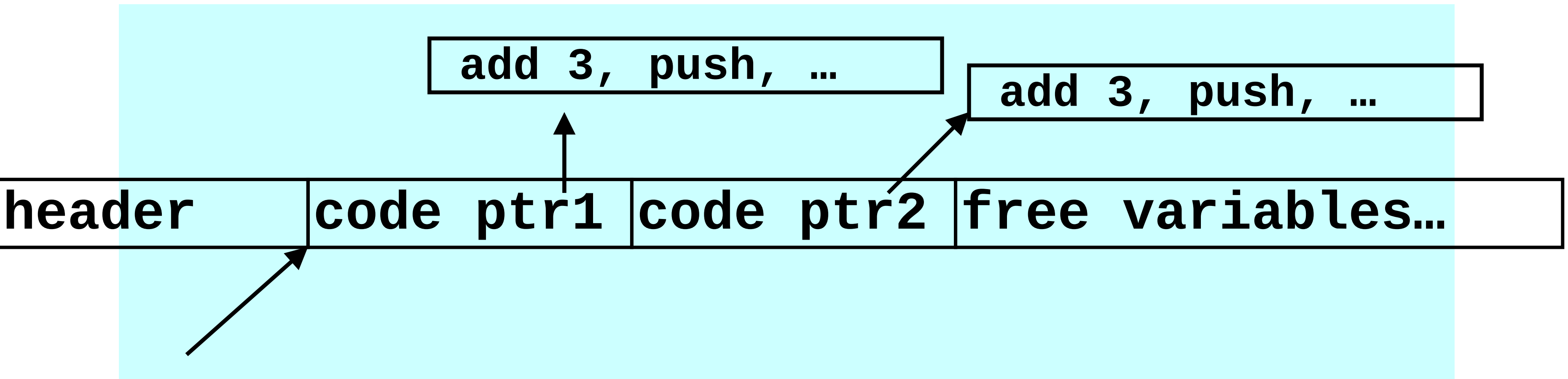
OCaml:



Representing closures

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

One approach: bigger closures

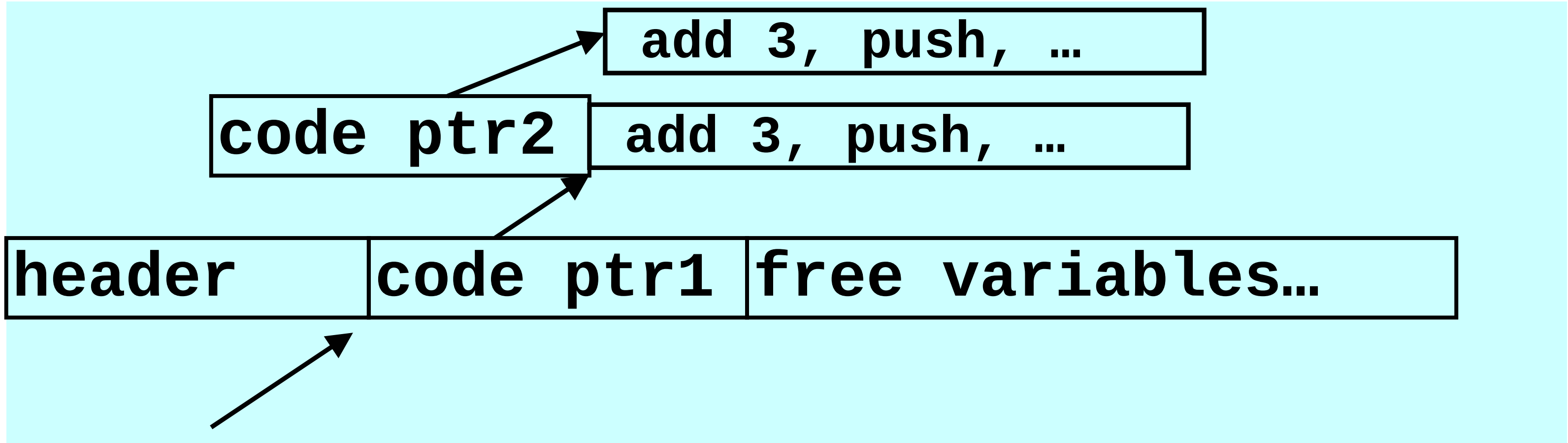


Note: atomic is first-class, so it is one of these too!

Representing closures

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

GC Interaction

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

	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

Outline

- Why (local reasoning)
  - Example
  - Case for strong atomicity
  - The GC analogy
- What (tough semantic "details")
  - Interaction with exceptions
  - Memory-model questions
- How (usually the focus)
  - In a uniprocessor model
  - Static analysis for removing barriers on an SMP

Performance problem

Recall uniprocessor overhead:

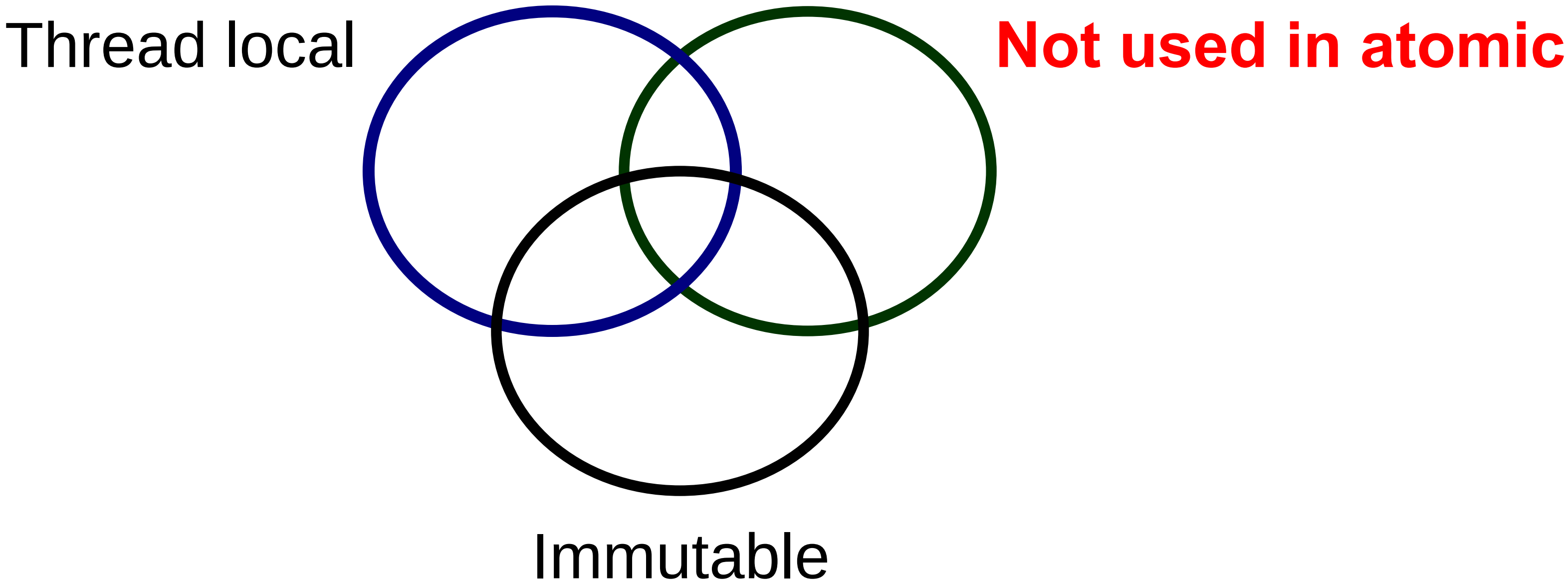
	not in atomic	in atomic
read	none	none
write	none	some

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)

Optimizing away barriers



New: static analysis for not-used-in-atomic…

Not-used-in-atomic

Revisit overhead of not-in-atomic for strong atomicity, given 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

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

Our view

SCAT (*) 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
2. "Strong" atomicity is key
3. If 1 thread runs at a time, strong atomicity is easy & fast
4. Else static analysis can improve performance

* (Scalable Concurrency Abstractions via Transactions)

Credit and other

OCaml: Michael Ringenburg

Java via source-to-source: Benjamin Hindman (B.S., Dec06)

Static barrier-removal: Steven Balensiefer, Katherine Moore

Transactions 1/n of my current research

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

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

[Presentation ends here; additional slides follow]

Blame analysis

Atomic localizes errors

(Bad code messes up only the thread executing it)

```
void bad1(){
 x.balance += 42;
}

void bad2(){
 synchronized(lk){
   while(true) ;
 }
}
```

- Unsynchronized actions by other threads are invisible to atomic
- Atomic blocks that are too long may get starved, but won't starve others
  - Can give longer time slices

Non-motivation

Several things make shared-memory concurrency hard

1. Critical-section granularity
   – Fundamental application-level issue?
   – Transactions no help beyond easier evolution?
2. Application-level progress
   – Strictly speaking, transactions avoid deadlock
   – But they can livelock
   – And the *application* can deadlock

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)

Granularity

Perhaps assume "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

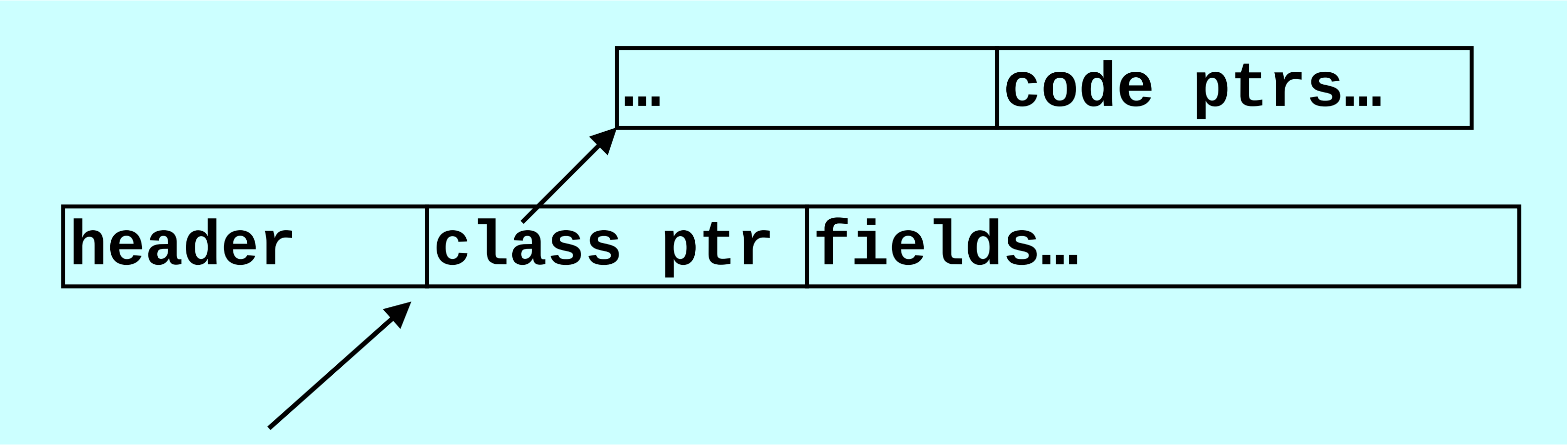
Conjecture: Profile-guided optimization can help

Note: Issue orthogonal to weak vs. strong

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 call thunk

Code evolution

Suppose StringBuffers are "self-locked" and you want to write append (JDK1.4, thanks to Flanagan et al)

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

Code evolution

Suppose StringBuffers are "self-locked" and you want to write append (JDK1.4, thanks to Flanagan et al)

```
int  length()    { synchronized(this) { … }}
void getChars(…) { synchronized(this) { … }}

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

Code evolution

Suppose StringBuffers are “self-locked” and you want to write append (JDK1.4, thanks to Flanagan et al)

```
int  length()    { atomic { … }}
void getChars(…) { atomic { … }}
void append(StringBuffer sb) {

  // race
  int len = sb.length();
  if(this.count + len > this.value.length)
    this.expand(…);
  sb.getChars(0,len,this.value,this.count);

}
```

Code evolution

Suppose StringBuffers are "self-locked" and you want to write append (JDK1.4, thanks to Flanagan et al)

```
int  length()    { atomic { … }}
void getChars(…) { atomic { … }}

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