Atomicity for Today's Programming Languages

Dan Grossman
University of Washington

24 March 2005

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)

Overview

- Language and language-tool support for locks
- The case for atomic
- Other approaches to atomic
- Logging-and-rollback for a uniprocessor
  - AtomCaml implementation
  - Programming experience
- Logging-and-rollback for a multiprocessor
  - High-level design only

Locks in high-level languages

Java a reasonable proxy for state-of-the-art

```
synchronized e { s }
```

Related features:

- Reentrant locks (no self-deadlock)
- Syntactic sugar for acquiring **`this`** for method call
- Condition variables (release lock while waiting)
- …

Java 1.5 features:

- Semaphores
- Atomic *variables* (compare-and-swap, etc.)
- Non-lexical locking

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 also synchronized
```

Detecting concurrency errors

Dynamic approaches

- Lock-sets: Warn if:
  - An object's accesses come from > 1 thread
  - Common locks held on accesses = empty-set
- Happens-before: Warn if an object's accesses are reorderable without
  - Changing a thread's execution
  - Changing memory-barrier order

neither sound nor complete

(happens-before more complete)

[Savage97, Cheng98, von Praun01, Choi02]

Detecting concurrency errors

Static approaches: lock types

- Type system ensures:

  For each shared data object, there exists a lock that a thread must hold to access the object

- Polymorphism essential
  - fields holding locks, arguments as locks, …
- Lots of add-ons essential
  - read-only, thread-local, unique-pointers, …
- Deadlock avoiding partial-order possible

incomplete, sound only for single objects

[Flanagan,Abadi,Freund,Qadeer99-02, Boyapati01-02,Grossman03]

Enforcing Atomicity

- Lock-based code often enforces atomicity (or tries to)
- Building on lock types, can use Lipton's theory of movers to detect [non]atomicity in locking code
- `atomic` becomes a *checked type annotation*
- Detects StringBuffer race (but not deadlock)

- Support for an inherently difficult task
  – the *programming* model remains tough

[Flanagan,Qadeer,Freund03-05]

Overview

- Language and language-tool support for locks
- The case for atomic
- Other approaches to atomic
- Logging-and-rollback for a uniprocessor
  - AtomCaml implementation
  - Programming experience
- Logging-and-rollback for a multiprocessor
  - High-level design only

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)

Target

Applications that use threads to:

- mask I/O latency
- provide GUI responsiveness
- handle multiple requests
- structure code with multiple control stacks
- …

Not:

- *high-performance scientific computing*
- *backbone routers*
- *…*

6.5 ways atomic is better

1. Atomic makes deadlock less common

```
transfer(Acct that,
         int x){
synchronized(this){
synchronized(that){
  this.withdraw(x);
  that.deposit(x);
}}}
```

- Deadlock with parallel "untransfer"
- Trivial deadlock if locks not re-entrant
- 1 lock at a time ⇒ race with "total funds available"

6.5 ways atomic is better

2. Atomic allows modular code evolution
   – Race avoidance: global object→lock mapping
   – Deadlock avoidance: global lock-partial-order

```
// x, y, and z are
// globals
void foo() {
synchronized(???){
 x.f1 = y.f2 + z.f3;
}}
```

- Want to write **`foo`** to be race and deadlock free
  – What locks should I acquire? (Are **`y`** and **`z`** immutable?)
  – In what order?

6.5 ways atomic is better

3. Atomic localizes errors

   (Bad code messes up only the thread executing it)

```
void bad1(){
 x.balance = -1000;
}

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

6.5 ways atomic is better

4. Atomic makes abstractions thread-safe without committing to serialization

```
class Set { // synchronization unknown
 void insert(int x) {…}
 bool member(int x) {…}
 int  size  ()      {…}
}
```

To wrap this with synchronization:

Grab the same lock before any call. But:

- Unnecessary: no operations run in parallel (even if **`member`** and **`size`** could)
- Insufficient: implementation may have races

6.5 ways atomic is better

5. Atomic is usually what programmers want
   [Flanagan, Qadeer, Freund]

- Many Java methods marked **`synchronized`** are actually atomic
- Of those that aren't, many races are application-level bugs
- **`synchronized`** is an implementation detail
  - does not belong in interfaces (atomic does)!

```
interface I { /* thread-safe? */ int m(); }
class A { synchronized int m() {
            <<call code with races>>
          }}
class B { int m() { return 3; }}
```

6.5 ways atomic is better

6. Atomic can efficiently implement locks

```
class SpinLock {
  bool b = false;
  void acquire() {
    while(true) {
      while(b) /*spin*/;
      atomic {
       if(b) continue;
       b = true;
       return; }
    }
  }
  void release() {
   b = false;
  }
}
```

- Cute O/S homework problem
- In practice, implement locks like you always have?
- Atomic and locks peacefully co-exist
  - Use both if you want

6.5 ways atomic is better

6.5 Concurrent programs have the granularity problem:

- Too little synchronization:
  non-determinism, races, bugs
- Too much synchronization:
  poor performance, sequentialization

Example: Should a chaining hashtable have one lock, one lock per bucket, or one lock per entry?

`atomic` doesn't solve the problem, but makes it easier to mix coarse-grained and fine-grained operations

Overview

- Language and language-tool support for locks
- The case for atomic
- Other approaches to atomic
- Logging-and-rollback for a uniprocessor
  - AtomCaml implementation
  - Programming experience
- Logging-and-rollback for a multiprocessor
  - High-level design only

A classic idea

- Transactions in databases and distributed systems
  - Different trade-offs and flexibilities
  - Limited (not a general-purpose language)
- Hoare-style monitors and conditional critical regions
- Restartable atomic sequences to implement locks
  - Implements locks w/o hardware support [Bershad]
- Atomicity for individual persistent objects [ARGUS]
- Rollback for various recoverability needs
- Disable interrupts

Rapid new progress

- **`atomic`** for Java
  - Uses Software Transactional Memory (STM) [Herlihy, Israeli, Shavit]
  - shadow-memory, version #s, commit-phase, …
- C*omposable* **`atomic`** for Haskell
  - Explicit **`retry`**: abort/retry after world changes
  - Sequential composition: “do s1 then s2”
  - Alternate composition: “do s1, but if aborts, do s2”
  - Leave transactions “open” for composition (atomic “closes” them)

[Harris, Fraser, Herlihy, Marlow, Peyton-Jones]
OOPSLA03, PODC04, PPoPP05

Rapid new progress

Closely related notions:

- Hardware for transactions
  - Instead of cache coherence, locking primitives, …
  - Programming: explicit forks and parallel loops
  - Long transactions may lock the bus

  [Hammond et al. ASPLOS04]

- *Transactional monitors* for Java
  - Most but not all of **`atomic`**'s advantages
  - Encouraging performance results

  [Welc et al. ECOOP04]

- Improve lock performance via transactions

  [Rajwar, Goodman ASPLOS02]

Claim

We can realize suitable implementations of atomic on today's hardware using a purely software approach to logging-and-rollback

- Alternate approach to STMs; potentially:
  - better guarantees
  - faster common case
- No need to wait for new hardware
  - A solution for today
  - A solution for backward-compatibility
  - Not yet clear what hardware should provide

Overview

- Language and language-tool support for locks
- The case for atomic
- Other approaches to atomic
- Logging-and-rollback for a uniprocessor
  - AtomCaml implementation
  - Programming experience
- Logging-and-rollback for a multiprocessor
  - High-level design only

Interleaved execution

The "uniprocessor" assumption:

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

Actually more general than uniprocessor: threads on different processors can pass messages

An important special case:

- Many language implementations make this assumption
- Many concurrent apps don't need a multiprocessor (e.g., a document editor)
- If uniprocessors are dead, where's the funeral?

Implementing atomic

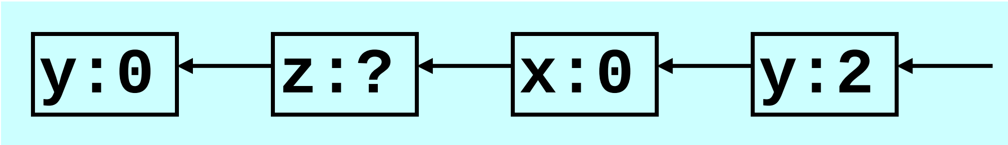
Key pieces:

- Execution of an atomic block logs writes
- If scheduler pre-empts a thread in an atomic block, rollback the thread
- Duplicate code so non-atomic code is not slowed down by logging/rollback
- In an atomic block, buffer output and log input
  - Necessary for rollback but may be inconvenient
  - A general native-code API

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:

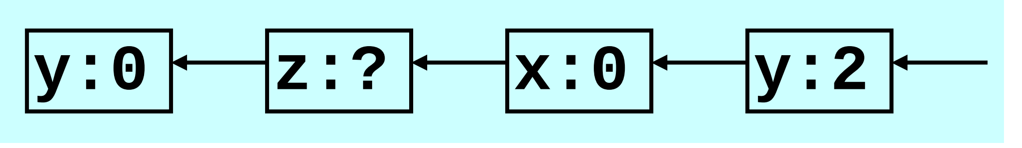


Rollback on pre-emption:

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

On exit from atomic: drop log

Logging efficiency



Keeping the log small:

- Don't log reads (key uniprocessor optimization)
- Don't log memory allocated after atomic was entered (in particular, local variables like **z**)
- No *need* to log an address after the first time
  - To keep logging fast, switch from an array to a hashtable only after "many" (50) log entries
  - Tell programmers non-local writes cost more

Duplicating code

```
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:

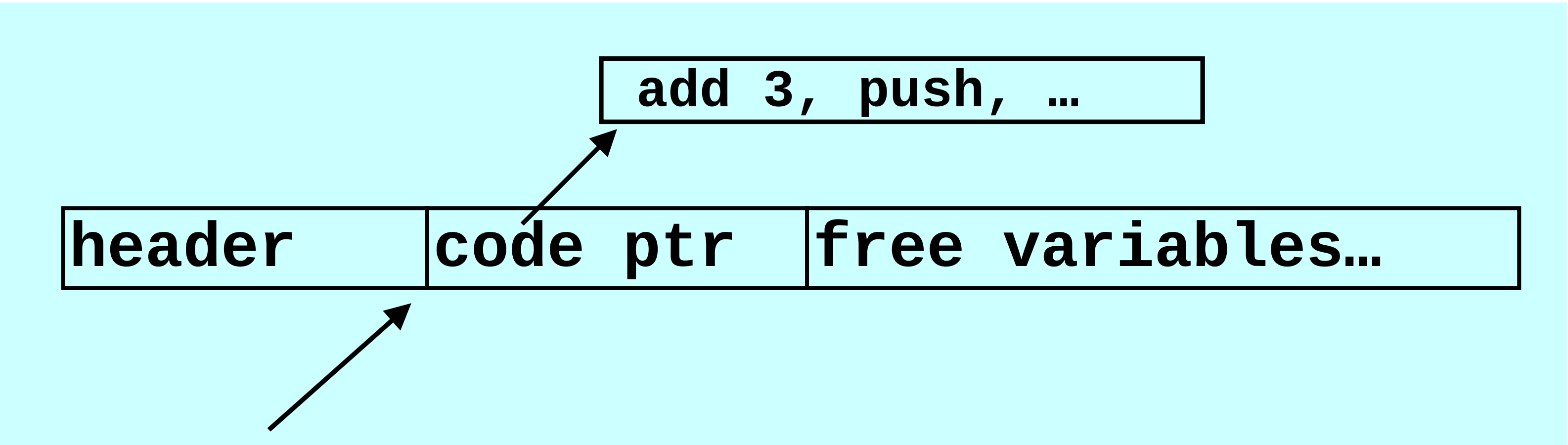
- For each function **f**, compile **f_atomic** and **f_normal**
- Atomic blocks and atomic functions call atomic functions
- Function pointers (e.g., vtables) compile to pair of code pointers

Cute detail: compiler erases any atomic block in **f_atomic**

Representing closures/objects

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

OCaml:

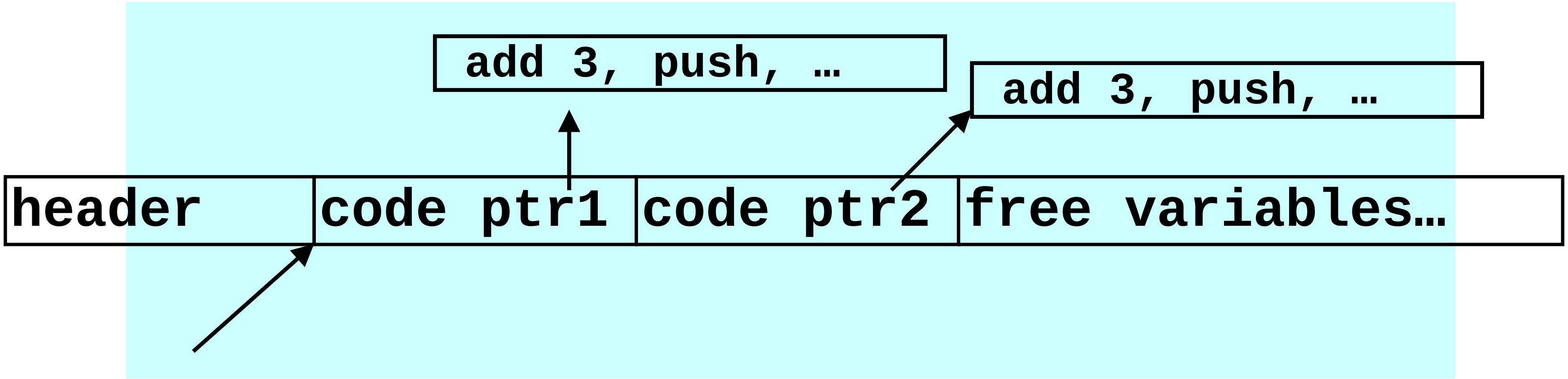


Representing closures/objects

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

AtomCaml prototype:

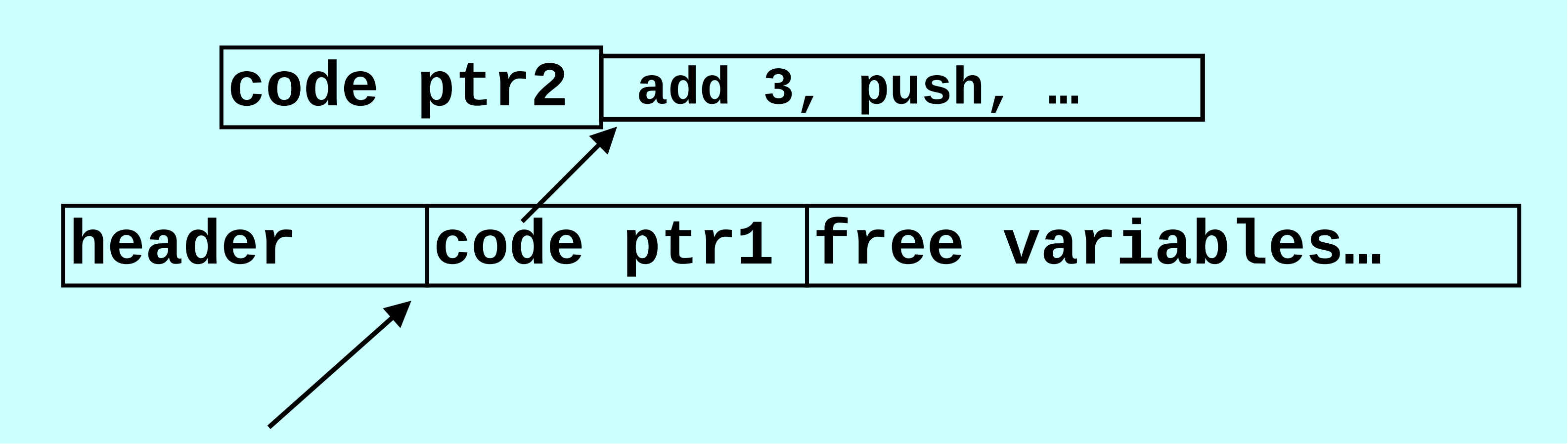
bigger closures (and related GC changes)



Representing closures/objects

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

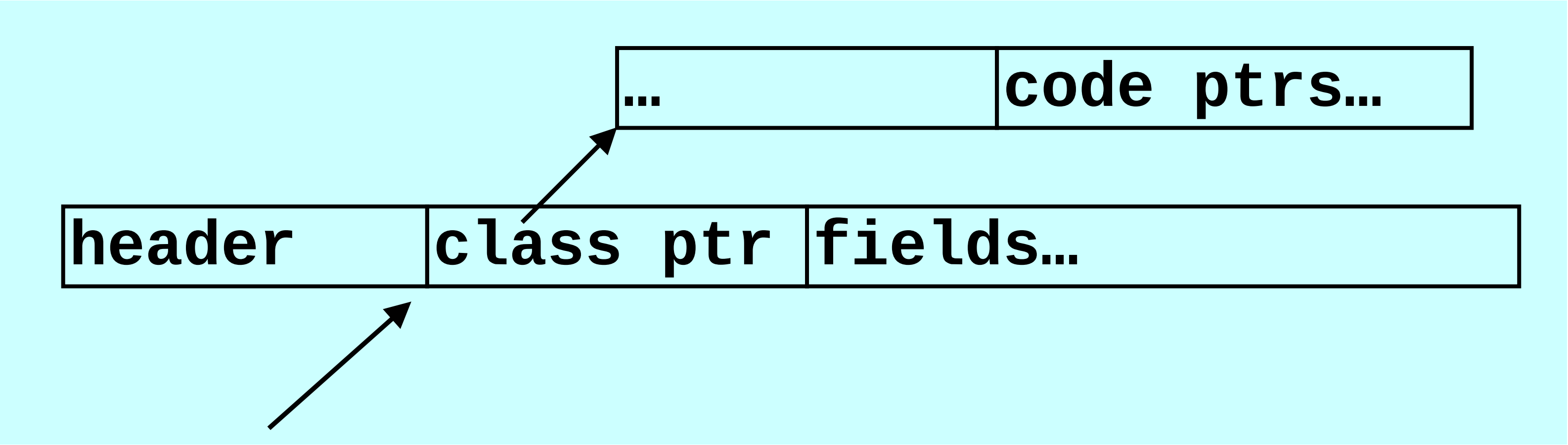
AtomCaml alternative:
(slower calls in **`atomic`**)



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)



Qualitative evaluation

- Non-atomic code executes unchanged
- Writes in atomic block are logged (2 extra writes)
- Worst case code bloat of 2x

- Thread scheduler and code generator must conspire

- Still have to deal with I/O
  - Atomic blocks probably shouldn't do much

Handling I/O

- Buffering sends (output) is easy and necessary
- Logging receives (input) is easy and necessary
  - And may as well rollback if the thread blocks
- But may miss subtle non-determinism:

```
void f() {
 write_file_foo(); // flushed?
 read_file_foo();
}
void g() {
  atomic {f();} // read won’t see write
  f();          // read may   see write
}
```

- Alternative: receive-after-send-in-atomic throws exception

General native mechanism

- Previous approaches: disallow native calls in **`atomic`**
  - raise an exception
  - obvious role for a static analysis or effect system
  - **`atomic`** no longer meaning preserving!
- We let the C library decide:
  - Provide two functions (in-atomic, not-in-atomic)
  - in-atomic can call not-in-atomic, raise-exception, or do something else
  - in-atomic can *register* commit-actions and rollback-actions (sufficient for buffering)
  - problem: if commit-action has an error “too late”

Overview

- Language and language-tool support for locks
- The case for atomic
- Other approaches to atomic
- Logging-and-rollback for a uniprocessor
  - AtomCaml implementation
  - Programming experience
- Logging-and-rollback for a multiprocessor
  - High-level design only

Prototype

- AtomCaml: modified OCaml bytecode compiler
- Advantages of mostly functional language
  - Fewer writes (don’t log object initialization)
  - To the front-end, **`atomic`** is just a function

```
atomic : (unit -> ’a) -> ’a
```

- Compiler bootstraps (single-threaded)
- Using atomic to implement locks, CML, …
- Planet active network [Hicks et al, INFOCOM99, ICFP98] “ported” from locks to **`atomic`**

Critical sections

- Most code looks like this:

```
try
  lock m;
  let result = e in
  unlock m;
  result
with ex -> (unlock m; raise ex)
```

- And often this is easier and equivalent:

```
atomic(fun()-> e)
```

- But not if e:
  - releases (and reacquires) **m**
  - calls native code
  - does something and "waits for response"

Condition Variables

- Idiom releasing/reacquiring a lock: Condition variable

```
lock m;
let rec loop () =
  if e1 then e3
  else (wait cv m; e2; loop())
in loop ();
unlock m;
```

Condition Variables

- Idiom releasing/reacquiring a lock: Condition variable

```
lock m;
let rec loop () =
  if e1 then e3
  else (wait cv m; e2; loop())
in loop ();
unlock m;
```

- This *almost* works

```
let f() = if e1 then Some e3 else None
let rec loop x =
  match x with
    Some y -> y
  | None -> wait' cv;
              loop(atomic(fun()-> e2; f()))
in loop(atomic f)
```

Condition Variables

- This *almost* works

```
let f() = if e1 then Some e3 else None
let rec loop x =
  match x with
    Some y -> y
  | None -> wait' cv;
                loop(atomic(fun()-> e2; f()))
in loop(atomic(fun()-> f()))
```

- Unsynchronized **`wait'`** is a race:

  we could miss the **`signal`** (notify)
- Solution: split **`wait'`** into
  - "start listening" (called in **`f()`**, returns a "channel")
  - "wait on channel" (yields unless/until the signal)

Porting Planet

- Found bugs
  - Reader-writer locks unsound due to typo
  - Clock library deadlocks if callback registers another callback
- Most lock uses trivial to change to **`atomic`**
- Condition variables uses need only local restructuring
- Handful of “native calls in atomic”
  - 2 pure (so hoist before atomic)
  - 1 a clean-up action (so move after atomic)
  - 3 we wrote new C versions that buffered
- Note: could have left some locks in but didn’t
- Synchronization performance all in the noise

Overview

- Language and language-tool support for locks
- The case for atomic
- Other approaches to atomic
- Logging-and-rollback for a uniprocessor
  - AtomCaml implementation
  - Programming experience
- Logging-and-rollback for a multiprocessor
  - High-level design only

A multiprocessor approach

- Give up on zero-cost reads
- Give up on safe, unsynchronized accesses
  - All shared-memory access must be within atomic (conceptually; compiler can insert them)
- But: Try to minimize inter-thread communication

Strategy: Use locks to implement **`atomic`**

- Each *shared* object guarded by a lock
  - Key: many objects can share a lock
- Logging and rollback to prevent deadlock

Example redux

```
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();
  }
}
```

- Atomic code acquires lock(s) for **x** and **y** (1 or 2 locks)
- Release locks on rollback or completion
- Avoid deadlock automatically. Possibilities:
  - Rollback on lock-unavailable
  - Scheduler detects deadlock, initiates rollback
- Only 1 problem…

What locks what?

There is little chance any compiler in my lifetime will infer a decent object-to-lock mapping

- More locks = more communication
- Fewer locks = less parallelism

What locks what?

There is little chance any compiler in my lifetime will infer a decent object-to-lock mapping

- More locks = more communication
- Fewer locks = less parallelism
- Programmers can't do it well either, though we make them try

What locks what?

There is little chance any compiler in my lifetime will infer a decent object-to-lock mapping

When stuck in computer science, use 1 of the following:

a. Divide-and-conquer
b. Locality
c. Level of indirection
d. Encode computation as data
e. An abstract data-type

Locality

Hunch: Objects accessed in the same atomic block will likely be accessed in the same atomic block again

- So while holding their locks, change the object-to-lock mapping to share locks
  - Conversely, detect false contention and break sharing
- If hunch is right, future atomic block acquires fewer locks
  - Less inter-thread communication
  - And many papers on heuristics and policies ☺

Cheap Profiling

Can cheaply monitor the lock assignment

- Per shared object:
  “my current lock”
- Per lock (i.e., objects ever used for locking):
  “number of objects I lock”:
  optional: “how much recent contention on me?”
- Also: atomic log of objects accessed

Revisit STMs

- STMs or lock-based logging-rollback?
  - It's time to try out all the basics
  - What would hybrids look like?
  - Analogy: 1960s garbage-collectors
- STM advantage: more optimistic, …
- Locks advantage: spatial locality; less wasted computation, …

Summary

- Atomic is a big win for reliable concurrency
- Key is implementation techniques and properties
  - Disabling interrupts
  - Uniprocessor logging-rollback
  - STMs
  - Multiprocessor logging-rollback
  - Hardware support?
    - Even when it exists, we'll want pure software approaches
    - Too early even to know what we want

Acknowledgments

- Joint work with PhD student Michael Ringenburg
  - Thanks to Manuel Fähndrich and Shaz Qadeer (MSR) for motivating us
- For updates and other projects:

  www.cs.washington.edu/research/progsys/wasp/

[end of presentation; auxiliary slides follow]

Condition Variables

- This *really* works

```
type 'a attempt = Go   of 'a
                | Wait of channel
let f() = if   e1
          then Go e3
          else Wait (listen cv)
let rec loop x =
  match x with
    Go   y  -> y
  | Wait ch ->
    wait’ ch; loop(atomic(fun()->e2;f()))
in loop(atomic f)
```

- Note: These condition variables are implemented in AtomCaml on top of **atomic**
  - (in 20 lines, including broadcast)

Condition variables

```
type channel = bool ref
type condvar = channel list ref
let create () = ref []
let signal cv =
  atomic(fun()->
    match !cv with
      []     -> ()
    | hd::tl -> (cv := tl; hd := false))
let listen cv =
  atomic(fun()->
    let r = ref true in
    cv := r :: !cv;
    r)
let wait ch =
  atomic(fun()->
    if !ch then yield_r ch else ())
```