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

```
withLk:
lock->(unit->a)->a
let xfer src dst x =
withLk src.lk (fun l->)
withLk dst.lk (fun r->)
src.bal <- src.bal-x;
( dst.bal <- dst.bal+x)
```

```
atomic:
(unit->a)->a
let xfer src dst x =
atomic (fun l->)
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...

Our plan

• Motivation (and non-motivation)
  – With a “PL bias” and an overly skeptical eye
• Semantics semi-formally
• Language-design options and issues

Next lecture: Software-implementation approaches
  – No mention of hardware (see Dwarkadas lecture)
Metapoint: Much research focused on implementations,
  but let’s “eat our vegetables”
Note: Examples in Caml and Java (metapoint: it largely
doesn’t matter)

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

Motivation

• Flanagan gave two lectures showing why atomicity is
  a simple, powerful correctness property
  – Inside an atomic block, sequential reasoning is
    sound!
• Why check it if we can provide it
  – And he ignored deadlock
• Other key advantages of providing it
  – Easier for code evolution
  – Easier “blame analysis” at run-time
  – Avoid priority inversion
Code evolution

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

```java
// 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?

Code evolution, cont’d

Not just new code is easier: fixing bugs
Flanagan’s JDK example with atomics:

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

Blame analysis?

Atomic localizes errors
(Bad code messes up only the thread executing it)

```java
void bad1(){
    x.balance -= 100;
}
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

Priority inversion

- Classic problem:
  High priority thread blocked on lock held by low priority thread
  But medium priority thread keeps running, so low priority can’t proceed
  Result: medium > high
- Transactions are abortable “at any point”, so we can abort the low, then run the high

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
The clincher

"Bad" programmers can destroy every advantage transactions have over locks

class SpinLock {
    volatile boolean b = false;
    void acquire() {
        while(true) {
            //optional spin
            atomic {
                if(b) continue; //test and set
                b = true;
                return;
            }
        }
        void release() { atomic { b = false; } }
    }
}

Our plan

- Motivation (and non-motivation)
  - With a "PL bias" and an overly skeptical eye
  - Bonus digression: The GC analogy
- Semantics semi-formally
- Language-design options and issues

Next lecture: Software-implementation approaches

Next point: Much research focused on implementations, but let's "eat our vegetables"

Why an analogy

- Already gave some of the crisp technical reasons why atomic is better than locks
- 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 synchronization] 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
  - Can be "one-size-fits-most"

Much more

More similarities:
- Old way still there (reimplement locks or free-lists)
- 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!

See a "tech-report" on my web-page (quick, fun read)
Our plan

• Motivation (and non-motivation)
  – With a “PL bias” and an overly skeptical eye
  – Bonus digression: The GC analogy
• Semantics semi-formally
• Language-design options and issues

Next lecture: Software-implementation approaches
  – Brief mention of hardware (see Dwarkadas lecture)

Metapoint: Much research focused on implementations, but let’s “eat our vegetables”

Atomic

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

withLk:
lock->(unit->a)->a
let xfer src dst x =
withLk src.lk (fun() ->
withLk dst.lk (fun() ->
src.x <- src.x-x;
dst.x <- dst.x+x )
)

atomic:
( unit->a)->a
let xfer src dst x =
atomic (fun() ->
src.x <- src.x-x;
dst.x <- dst.x+x )

lock acquire/release (behave as if)
no interleaved computation

Strong atomicity

(behavior 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

Formalizing it

At the high-level, a formal small-step operational semantics is simple

• Atomic block “runs in 1 step” [Harris et al PPoPP05]
• Recall from intro lecture:
  “one thread, one step” $H,e \rightarrow H', e', o$
  “program, one step” to $H,e1, ..., en \rightarrow H', e1', ..., en'$

Wrong

$H,e \rightarrow H', e', o$
$H.atomic,e \rightarrow H', e', o$
$H.atomic,v \rightarrow H, v$. None

Closer to right

The essence of atomic is that it’s “all one step”
Note $\rightarrow^*$ is reflexive, transitive closure.

Ignoring fork

Claim (unproven): Adding atomic to fork-free program has no effect
About fork (exercise): One step could create n threads

Incorporating abort (a.k.a. retry)

An explicit abort (a.k.a. retry) is a very useful feature.
Tiny example:

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

Formally: $e ::= .. | abort$

Non-determinism is elegant but unrealistic!

Lower-level

We could also define an operational semantics closer to an actual implementation

• Versioning of objects
• Locking of objects

And prove such semantics equivalent to our “magic semantics”

See: [Vitek et al. ECOOP04]

Weak atomicity

(behavior 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

A lie

Bogus claim: ‘Under this ‘definition’, atomic blocks are still atomic w.r.t. each other’

Reality: Assuming no races with non-transactional code

```
// invariant: x and y are even
atomic {
  y=x;
  atomic {
    f();
    if(y%2==1)
      bad();
    } 
  =x;
}
```

Note: The transactions might even access disjoint memory.

Is that so bad?

Assumptions are fine if they’re true

• Programmer discipline
  – Good luck (cf. array-bounds in C)
• Race-detection technology
  – Whole-program analysis
• Type system
  – Much existing work should adapt
  – Avoiding code duplication non-trivial
  – Haskell uses a monad to segregate “transaction variables”

Our plan

• Motivation (and non-motivation)
  – With a “PL bias” and an overly skeptical eye
• Semantics semi-formally
• Language-design options and issues

Next lecture: Software-implementation approaches
  – Brief mention of hardware (see Dwarkadas lecture)

Metapoint: Much research focused on implementations, but let’s “eat our vegetables”

Language-design issues

“fancy features” & interaction with other constructs

As time permits, with bias toward AtomCaml [ICFP05]:

• Strong vs. weak vs. type distinction on variables
• Interaction with exceptions
• Interaction with native-code
• Condition-variable idioms
• Closed nesting (flatten vs. partial rollback)
• Open nesting (back-door or proper abstractions?)
• Parallel nesting (parallelism within transactions)
• The orelse combinator
• Memory-ordering issues
• Atomic as a first-class function (elegant, useless?)
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:

```haskell
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 (talk to Shinnar)
- But not about concurrency
- Has problems with the exception value

Exception

With “exception commits” and catch, the programmer can get “exception aborts”

```haskell
atomic {
  try { s }
  catch (Throwable e) {
    abort;
  }
}
```

Handling I/O

- Buffering sends (output) easy and necessary
- Logging receives (input) easy and necessary
- But input-after-output does not work

```haskell
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
- Can 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)
    - The "launch missiles problem"

Language-design issues

"fancy features" & interaction with other constructs
As time permits, with bias toward AtomCaml [ICFP05]:
- Strong vs. weak vs. type distinction on variables
- Interaction with exceptions
- Interaction with native-code
- Condition-variable idioms
- Closed nesting (flatten vs. partial rollback)
- Open nesting (back-door or proper abstraction?)
- Parallel nesting (parallelism within transactions)
- The orelse combinator
- Memory-ordering issues
- Atomic as a first-class function (elegant, useless?)

Critical sections

- Most code looks like this:

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

- And often this is easier and equivalent:

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

- But not always...
Non-atomic locking

Changing a lock acquire/release to atomic is wrong if it:
• Does something and “waits for a response”
• Calls native code
• Releasses and reacquires the lock:

```c
lock(m);
s1;
while(e){
  wait(m,cv);
s2;
}
s3;
unlock(m);
```

If s1 and e are pure, wait can become an abort, else we really have multiple critical sections

Wrong approach #1

```c
atomic {
s1;
  if(e) wait(cv);
  else {s3:return;}
}
while(true){
  atomic{
    s2;
    if(e) wait(cv);
    else {s3:return;}
  }

Cannot wait in atomic!
• Other threads can’t see what you did
• You block and can’t see signal
```

Wrong approach #2

```c
b=false;
atomic {
  s1;
  if(e) b=true;
  else {s3:return;}
}
if(b) wait(cv);
while(true){
  atomic{
    s2;
    if(!e){s3:return;}
  }
  wait(cv);
}
```

Cannot wait after atomic: you can miss the signal!

Solution: listen!

```c
b=false;
atomic {
  s1;
  if(e) {
    ch=listen(cv);
b=true;
  } else {s3:return;}
} if(b) wait(ch);
/* similar for the loop */
```

You wait on a channel and can listen before blocking (signal chooses any channel)

The interfaces

With locks:
```c
condvar new_condvar();
void wait(lock,condvar);
void signal(condvar);
```

With atomic:
```c
condvar new_condvar();
channel listen(condvar);
void wait(channel);
void signal(condvar);
```

A 20-line implementation uses only atomic and lists of mutable booleans
Language-design issues

"fancy features" & interaction with other constructs
- Strong vs. weak vs. type distinction on variables
- Interaction with exceptions
- Interaction with native-code
- Condition-variable idioms
- Closed nesting (flatten vs. partial rollback)
- Open nesting (back-door or proper abstraction?)
- Parallel nesting (parallelism within transactions)
- The orelse combinator
- Memory-ordering issues
- Atomic as a first-class function (elegant, useless?)

Closed nesting

One transaction inside another has no effect!

void f() { ... atomic { ... g() ... } }
void h() { ... atomic { ... } }

- AtomCaml literally treats nested atomic "as a no-op"
- Abort to outermost (a legal interpretation)
- Abort to innermost ("partial rollback") could avoid some recomputation via extra bookkeeping [Intel, PLDI06]
- Recall in reality there is parallelism
- Claim: This is not an observable issue, "just" an implementation question.

Open nesting

An open (open { s; }) is a total cheat/back-door
- Its effects happen even if the transaction aborts
- So can do them "right away"

Arguments against:
- It's not a transaction anymore!
- Now caller knows nothing about effect of "wrapping call in atomic"

Arguments for:
- Can be correct at application level and more efficient
  - (e.g., caching, unique-name generation)
- Useful for building a VM (or O/S) w/ only atomic
  [Atomos, PLDI06]

A compromise?

- Most people agree the code in the open should never access memory the "outer transaction" has modified.
- So could detect this conflict and raise a run-time error.
- But... this detection must not have false positives from false sharing
  - E.g., a different part of the cache line

Parallel nesting

- Simple semantics: A fork inside an atomic is delayed until the commit
  - Compatible with "no scheduling guarantees"
- But then all critical sections must run sequentially
  - Not good for many-core
- Semantically, could start the threads, let them see transaction state, kill them on abort
  - Now nested transactions very interesting!
  - It all works out [Moss, early 80s]
  - Implementation more complicated (what threads should see what effects of what transactions)
    - Must maintain/discard fork/transaction trees

Language-design issues

"fancy features" & interaction with other constructs
- Strong vs. weak vs. type distinction on variables
- Interaction with exceptions
- Interaction with native-code
- Condition-variable idioms
- Closed nesting (flatten vs. partial rollback)
- Open nesting (back-door or proper abstraction?)
- Parallel nesting (parallelism within transactions)
- The orelse combinator
- Memory-ordering issues
- Atomic as a first-class function (elegant, useless?)
Why orelse?

• Sequential composition of transactions is easy:

```c
void f() { atomic { ... } }
void g() { atomic { ... } ; }
void h() { atomic { f(); g(); } ; }
```

• But what about alternate composition

• Example: “get something from either of two buffers, failing only if both are empty”

```c
void get(buf){
    atomic{if(empty(buf)) abort; else ...}
void get2(buf1,buf2) { ??? }
```

orelse

• Only "solution" so far is to break abstraction
  – The greatest sin in programming
• Better:
  – `atomic{get(buf1); orelse{get(buf2);}
  – Semantics: On abort, try alternative, if it also aborts, the whole thing aborts
• Early similar to something Flatt just showed you?

Memory-Ordering issues

• As Dwarkadas and Cartwright have told you, sequential consistency is often not provided by hardware or a language implementation
  – For a compiler, can prevent “basic” optimizations like dead-code elimination
• Locking: Acquires and releases of the same lock must be ordered ("happens before")
• Transactions: There are no locks!
  – No great solution known ("accesses same memory" prohibits changing memory accesses)
  – Ongoing work with Pugh & Manson

Language-design issues

“fancy features” & interaction with other constructs
As time permits, with bias toward AtomCam [ICFP05]:
  – Strong vs. weak vs. type distinction on variables
  – Interaction with exceptions
  – Interaction with native-code
  – Condition-variable idioms
  – Closed nesting (flatten vs. partial rollback)
  – Open nesting (back-door or proper abstraction?)
  – Parallel nesting (parallelism within transactions)
  – The orelse combinator
  – Memory-ordering issues
  – Atomic as a first-class function (elegant, useless?)

Basic design

no change to parser and type-checker
  – `atomic` a first-class function
  – Argument evaluated without interleaving

```c
external atomic : (unit->a)->a = "atomic"
```

Advantages:
  • Elegant
  • Simplifies implementation (next time)
  • "Same old" functional-language sermon?
  • Not actually useful to programmers?

Our plan

• Motivation (and non-motivation)
  – With a "PL bias" and an overly skeptical eye
  • Semantics semi-formally
  • Language-design options and issues

Next lecture: Software-implementation approaches
  – Brief mention of hardware (see Dwarkadas lecture 3)
Metapoint: Much research focused on implementations, but let's "eat our vegetables"
Note: Examples in Caml and Java (metapoint: it largely doesn't matter)