Strong Atomicity for Today's Programming Languages

Dan Grossman
University of Washington

10 October 2005
Atomic

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

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

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

semantics:
lock acquire/release

semantics:
(behavior 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*
• …
Overview

- The case for atomic
- Previous approaches to atomic
- AtomCaml
  - Logging-and-rollback
  - Uniprocessor implementation
  - Programming experience
- AtomJava
  - Logging-and-rollback
  - Source-to-source implementation (unchanged JVM)
- Condition variables via atomic (time permitting)
Locks in high-level languages

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

```java
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]

```java
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
```
Detecting locking errors

• Data-race detectors
  – Dynamic (e.g., what locks held when)
  – Static (e.g., type systems for what locks to hold)
  – *Little work on higher-level races*
• Deadlock detectors
  – Static (e.g., program-wide partial-order on locks)
• Atomicity checkers
  – Static (treat “atomic” as a type annotation)

  *Can catch bugs, but the tough programming model remains!*

[Savage97, Cheng98, von Praun01, Choi02, Flanagan, Abadi, Freund, Qadeer99-05, Boyapati01-02, Grossman03, …]
Atomic

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

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

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

semantics:
lock acquire/release

semantics:
(behavior as if)
no interleaved execution

No fancy hardware, code restrictions, deadlock, or unfair scheduling (e.g., disabling interrupts)
6.5 ways atomic is better

1. Atomic makes deadlock less common

```java
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 \(\Rightarrow\) 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)

```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
6.5 ways atomic is better

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

```java
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 synchronized Java methods 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() { «race» } }  
class B { int m() { return 3; } }
6. Atomic **can efficiently implement locks**

```java
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 per table, per bucket, or per entry?

**atomic** doesn’t solve the problem, but makes it easier to mix coarse- and fine-grained operations
Overview

• The case for atomic

• Previous approaches to atomic

• AtomCaml
  – Logging-and-rollback
  – *Uniprocessor implementation*
  – Programming experience

• AtomJava
  – Logging-and-rollback
  – *Source-to-source implementation* (unchanged JVM)

• Condition variables via atomic
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
STMs

- Software Transactional Memory
  - Compute using private version of memory
  - Commit via sophisticated protocols (version #s, etc)
- Java [OOPSLA03]:
  - Guard expressions: \texttt{atomic(e)\{s\}}
  - Weak guarantee: only atomic w.r.t. other atomics!
- Haskell [PPoPP05]:
  - Composition: “if s1 aborts, try s2”
  - Strong guarantee via purely functional language
- C#:
  - Just a library
  - Thread-shared data has many restrictions, must be created by factories, …

[Herlihy, Harris, Fraser, Marlow, Peyton-Jones,…]
Warning:

Next slide criticizes the work of the audience.

Why?

Provoke good conversation (later?)

Strong belief:

Long-term solutions will be hw + sw, but we’re still learning the pure hw and pure sw solutions
HTMs

Hardware Transactional Memory
• extend ISA with “xstart” and “xend”
• cache for logging-and-rollback
• contention similar to cache-coherence (pay once!)
• long-running transactions lock the bus [ASPLOS04] or use hardware to log in RAM [HPCA05]

I am skeptical (and biased):
• need a software answer too (legacy chips, etc.)
• logs things that need not be logged
  – immutable fields
  – a garbage collection triggered in atomic
• ISA’s semantics won’t match a language’s atomic
  – compilers want building blocks
Claim

We can realize suitable implementations of strong atomicity 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
  - Not yet clear what hardware should provide
Overview

- The case for atomic
- Previous approaches to atomic
- **AtomCaml**
  - Logging-and-rollback
  - *Uniprocessor implementation*
  - Programming experience
- **AtomJava**
  - Logging-and-rollback
  - *Source-to-source implementation* (unchanged JVM)
- Condition variables via atomic
Interleaved execution

The “uniprocessor” assumption:

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

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)
• 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

• In an atomic block, buffer output and log input
  – Necessary for rollback but may be inconvenient
  – A general native-code API

Note: Similar idea for RTSJ by Manson et al. [Purdue TR 05]
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:

```
y:0  z:?  x:0  y:2
```

Rollback on pre-emption:
• Pop log, doing assignments
• Set program counter and stack to beginning of atomic

On exit from atomic: drop log
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
Duplicate code so callees know to log or not:

- For each function \( f \), compile \( f_{\text{atomic}} \) and \( f_{\text{normal}} \)
- 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 \( f_{\text{atomic}} \)

```c
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();
    }
}
```
Representing closures/objects

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

OCaml:

```
header | code ptr | free variables...
```

```
add 3, push, ...
```
Representing closures/objects

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

AtomCaml:
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 \texttt{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

• But may miss subtle non-determinism:

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

- Previous approaches: disallow native calls in *atomic*
  - raise an exception
  - *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

• The case for atomic
• Previous approaches to atomic
• AtomCaml
  – Logging-and-rollback
  – *Uniprocessor implementation*
  – Programming experience
• AtomJava
  – Logging-and-rollback
  – *Source-to-source implementation* (unchanged JVM)
• Condition variables via atomic
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

\[
\text{atomic} : (\text{unit} \to 'a) \to 'a
\]

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

- And often this is easier and equivalent:
  
  ```ocaml
  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
- Releases and reacquires the lock:

```ml
lock m;
s1;
let rec loop () =
  if e
    then (wait cv m; s2; loop())
  else s3
in loop ();
unlock m
```
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
- Condition-variable uses need only local restructuring
- 6 “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

• The case for atomic
• Previous approaches to atomic
• AtomCaml
  – Logging-and-rollback
  – *Uniprocessor implementation*
  – Programming experience
• AtomJava
  – Logging-and-rollback
  – *Source-to-source implementation* (unchanged JVM)
• Condition variables via atomic
A multiprocessor approach

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

Less efficient straight-line code:
- All (even non-atomic) code must hold the correct lock to write or read a thread-shared object

But try to minimize inter-thread communication
- “Acquiring” a lock you hold needs no synchronization
Acquiring locks

Translate from AtomJava to Java:
• add getter/setter methods for each field
• code duplication and logging like in AtomCaml

• e.f becomes e.get_f()
  – acquire lock for e, then return e.f
• e1.f = e2 similar (and atomic version logs)
• Every object’s lock has a current-holder field
  – If the Thread “is me”, continue.
  – Else ask the holder to release the lock and wait
Releasing locks

- Threads *poll* to see if they hold requested locks
  - Rewrite source code to insert polling calls
  - To avoid deadlock, satisfy requests
  - If in atomic and you release a lock, rollback first

- Exponential backoff to avoid livelock

- For correctness, the rest is in the (many) details: arrays, primitive types, java.lang, class-loading, native calls, constructors, static fields, …
Optimizations

• Access does not need a lock if *any* of the following:
  – Data is thread-local
  – Data is immutable
  – Data is never accessed within an atomic block
  – You definitely hold the lock already
• Static and dynamic tricks to reduce polling costs
• … much, much more (make it a compiler problem!)

Only one problem… what is the object-to-lock mapping?
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 atomics acquire fewer locks
  – Less inter-thread communication
  – And many papers on heuristics and policies 😊

• Challenge is cheap profiling (future work)
Overview

• The case for atomic
• Previous approaches to atomic
• AtomCaml
  – Logging-and-rollback
  – *Uniprocessor implementation*
  – Programming experience
• AtomJava
  – Logging-and-rollback
  – *Source-to-source implementation* (unchanged JVM)
• Condition variables via atomic
Summary

• (Strong) atomic is a big win for reliable concurrency
• Key is implementation techniques and properties
  – Disabling interrupts
  – Software Transactional Memory
  – Hardware Transactional Memory
  – Uniprocessor logging-rollback
  – Multiprocessor logging-rollback
An analogy

Garbage collection is a big win for reliable memory management

• Programmers can usually ignore the implementation
• For 3 decades, perceived as “too slow”
  (and we tried hardware support)
• Manual memory management requires subtle, whole-program invariants

Is “TMs vs. rollback” like “copying vs. mark-sweep” (will the best systems be a hybrid)?
Hopefully < 30 years to find out
Acknowledgments

• Joint work with students Michael Ringenburg and Ben Hindman

• For updates and other projects:
  www.cs.washington.edu/research/progsys/wasp/
[end of presentation; auxiliary slides follow]
Condition variables: canonical use

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

- **wait** blocks until another thread *signals* cv
- signalling thread must hold m
Atomic w.r.t. code holding $m$:

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

```java
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);
```

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 implemention uses only atomic and lists of mutable booleans
[really, really auxiliary slides follow]
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 Praun 01, 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]
• Idiom releasing/reacquiring a lock: Condition variable

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

• This almost works

```ocaml
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

```ocaml
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)
Condition Variables

• This really works

```ml
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 ()
)
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…
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, …