Design and Implementation Issues for Atomicity

Dan Grossman

University of Washington

Workshop on Declarative Programming Languages for Multicore Architectures

15 January 2006

Atomicity Overview

- Atomicity: what, why, and why relevant
- Implementation approaches (hw & sw, me & others)
- 3 semi-controversial language-design claims
- 3 semi-controversial language-implementation claims
- Summary and discussion (experts are lurking)

Atomic

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

```
withLock:
  lock->(unit->α)->α

let dep acct amt =
withLock acct.lk
 (fun()->
  let tmp=acct.bal in
  acct.bal <- tmp+amt)
```

lock acquire/release

```
atomic:
  (unit->α)->α

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

(behave as if)
no interleaved execution

No deadlock or unfair scheduling (e.g., disabling interrupts)

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)

Declarative control

For programmers who will see:

threads & shared-memory & parallelism

`atomic` directly *declares* what schedules are allowed

(without sacrificing pre-emption and fairness)

Moreover, implementations perform better with immutable data, encouraging a functional style

Implementing atomic

Two basic approaches:

1. Compute using "shadow memory" then *commit*
   - Fancy optimistic-concurrency protocols for parallel commits with progress (STMs)

   [Harris et al. OOPSLA03, PPoPP05, ...]

2. Lock data before access, log changes, *rollback* and back-off on contention
   - My research focus
   - Key performance issues: locking granularity, avoiding unneeded locking
   - Non-issue: *any* granularity is *correct*

An extreme case

One extreme:

- One lock for all data
- Acquire lock on context-switch-in
- Release lock only on context-switch-out
  - (after rollback if necessary)

Per data-access overhead:

	Not in atomic	In atomic
Read	none	none
Write	none	logging

Ideal on *uniprocessors* [ICFP05, Manson et al. RTSS05]

In general

Naively, locking approach with parallelism looks bad

(but note: no communication if already hold lock)

	Not in atomic	In atomic
Read	lock	lock, maybe rollback
Write	lock	lock, maybe rollback, logging

Active research:

1. Hardware: lock = cache-line ownership

   [Kozyrakis, Rajwar, Leiserson, …]

2. Software (my work-in-progress for Java):
   - Static analysis to avoid locking
   - Dynamic lock coarsening/splitting

Atomicity Overview

- Atomicity: what, why, and why relevant
- Implementation approaches (hw & sw, me & others)
- 3 semi-controversial language-design claims
- 3 semi-controversial language-implementation claims
- Summary and discussion

Claim #1

"Strong" atomicity is worth the cost

"Weak" says only atomics not interleaved with each other

– Says nothing about interleaving with non-atomic

So:

	Not in atomic	In atomic
Read	none	lock, maybe rollback
Write	none	lock, maybe rollback, logging

But back to bad synchronization breaking good code!

Caveat: Weak=strong if all thread-shared data accessed within atomic (other ways to enforce this)

Claim #2

Adding atomic shouldn't change "sequential meaning"

That is, ***e*** and **`atomic (fun()-> e)`** should be equivalent in a single-threaded program

But it means exceptions must commit, not rollback!

– Can have "two kinds of exceptions"

Caveats:

- Tough case is "input after output"
- Not a goal in Haskell (already a separate monad for "transaction variables")

Claim #3

Nested transactions are worth the cost

Allows parallelism within **`atomic`**

– "Participating" threads see uncommitted effects

Currently most prototypes (mine included) punt here, but I think many-many-core will drive its need

Else programmers will hack up buggy workarounds

Claim #4

Hardware implementations are too low-level and opaque

Extreme case: ISA of “start_atomic” and “end_atomic”

Rollback does not require RAM-level rollback!

- Example: logging a garbage collection
- Example: rolling back thunk evaluation

All I want from hardware: fast conflict detection

Caveats:

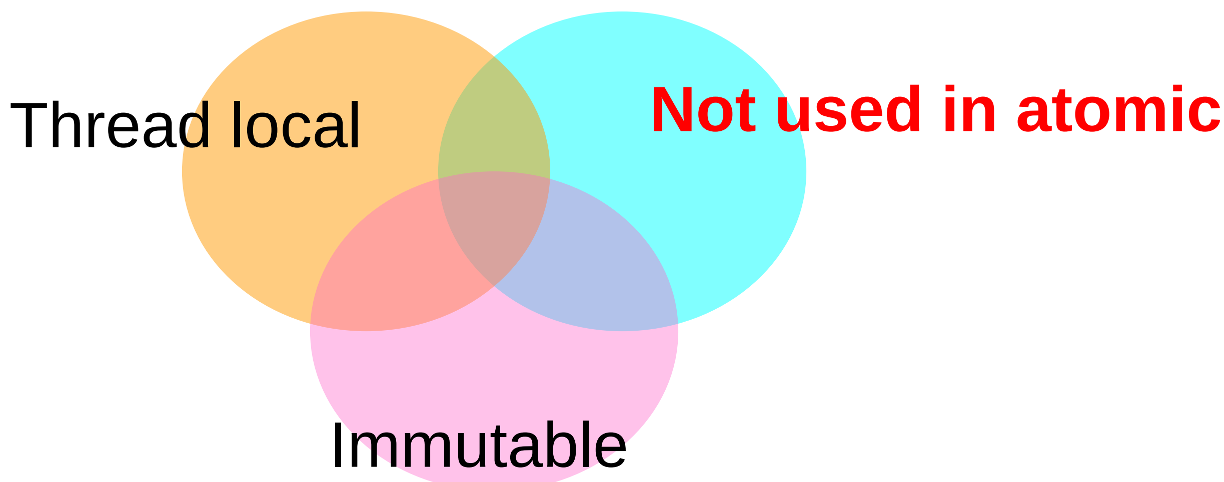
- Situation improving fast (we’re talking!)
- Focus has been on chip design (orthogonal?)

Claim #5

Simple whole-program optimizations can give strong atomicity for close to the price of weak

Lots of data doesn't need locking:

(2/3 of diagram well-known)



Caveat: unproven; hopefully numbers in a few weeks

Claim #6

Serialization and locking are key tools
for implementing atomicity

- Particularly in low-contention situations

- STMs are great too
  - I predict best systems will be hybrids
  - Just as great garbage collectors do some copying, some mark-sweep, and some reference-counting

Summary

1. Strong atomicity is worth the cost
2. Atomic shouldn't change sequential meaning
3. Nested transactions are worth the cost

4. Hardware is too low-level and opaque
5. Program analysis for "strong for the price of weak"
6. Serialization and locks are key implementation tools

Lots omitted: Alternative composition, wait/notify idioms, logging techniques, …

www.cs.washington.edu/homes/djg

Plug

Relevant workshop before PLDI 2006:

TRANSACT:
First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing

www.cs.purdue.edu/homes/jv/events/TRANSACT/