Transactional Memory Should be an Implementation Technique, Not a Programming Interface

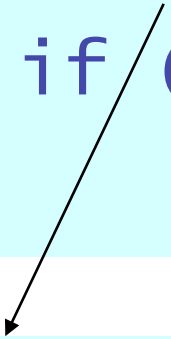
Hans-J. Boehm

Why Transactional Memory?

- A mechanism for providing failure atomicity?
  - Largely subject of last talk. Part of answer?
  - Interacts with parallelism, but
  - Just as interesting without parallelism.
  - Hard for reasons that have nothing to do with parallelism. (next slide)
  - For the purposes of this talk, it's
    a) Not central.
    b) Too hard for us.

Why failure atomicity is hard:

```
atomic {
   x.foo();
   if (…) abort();
}
```



```
foo() {
   launch_missile_now();
}
```

How do you prevent this?

- Dynamically?
  - Problematic
- Type/effect systems?
  - Too complex?

Thanks to Intel authors of last paper & Tatiana Shpeisman.

Why TM? (Second, final try)

- A simpler synchronization mechanism than locks.

- (There are other possible, usually less ambitious, answers, but they're *not* the subject of this talk.)

But threads with locks already have (superficially?) simple semantics

- Java 1.5+ and C++0x both support "sequential consistency for data-race-free programs".*
- Multithreaded execution can be viewed as interleaving:
  - Canonical example (everything initially zero):

Thread 1	*Thread 2*
`x = 1;`	`y = 1;`
`r1 = y;`	`r2 = x;`

  - Might be executed as:

    x = 1; y = 1; r2 = x; r1 = y; or

    x = 1; y = 1; r1 = y; r2 = x;
  - *Provided there are no data races:*
    - No such interleaving has conflicting adjacent non-atomic/volatile operations from different threads.
  - No `lock(l)` can appear in the interleaving unless prior calls from other threads balance.

*Provided certain esoteric library calls are avoided.

Handling locks

Thread 1

```
lock(l);
r1 = x;
x = r1+1;
unlock(l);
```

Thread 2

```
lock(l);
      r2 = x;
x = r2+1;
unlock(l);
```

– can only be executed as

```
lock(l); r1 = x; x = r1+1; unlock(l); lock(l);
r2 = x; x = r2+1; unlock(l);
```

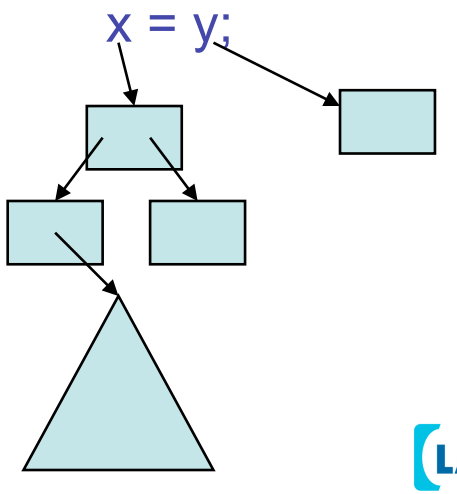
or

```
lock(l); r2 = x; x = r2+1; unlock(l); lock(l);
r1 = x; x = r1+1; unlock(l);
```

since second lock(l) must follow first unlock(l)

So what's complicated about locks?

- Code needs to be designed to avoid deadlocks.
- Usually locks are acquired in a fixed order.
- This complicates interfaces.
- And it's basically incompatible with some common programming practices, e.g.
  - `x = y;` in C++ may be implemented as a reference counted assignment.
  - May synchronously deallocate a large opaque data structure previously referenced by `x`, acquiring many unknown locks.
  - Avoid assignments in critical sections.



LABS hp

Obvious solution:

- Only use one (reentrant) lock.
- Semantics are even (slightly) simpler.
- Lock-based deadlocks are impossible.
- No issues with:
  - Strong vs. weak isolation
  - Publication and privatization safety
  - Irreversible IO actions
  - Interactions with existing lock-based code
  - …

So what are we missing?

- Nice syntax
  - `atomic { }` *vs.* `synchronized(the_lock) { }`
  - View `atomic {}` as abbreviation!
- Scalability
  - Fixable with clever implementations (?)
  - Some TM systems (e.g. Intel's) already provide "Single Global Lock" semantics.
  - Semantically we want a single global lock.
  - TM is potentially a great implementation technique.
  - But not the only one.

What about thread communication and `retry`?

- Inherently doesn't compose correctly:

```
f() { // in library
  do something;
  if (error) while (msgs to log) {
    atomic {
      try to add msg to buffer;
      if (buffer was full)
        retry;
    } } }
main() { atomic { f(); }}
```

- Retrying at inner level violates isolation.
- Retrying at outer level doesn't work.

Solution to `retry` problem

- ~~`retry`~~
- Use locks and condition variables!
- Needed for legacy code anyway.
- Locks provide the right kind of partial isolation
  - Transactions don't.
- Simpler than open nesting. (No undo actions.)
- Can usually be hidden in libraries?
- For examples like this, TM-like implementations still work for most calls.

What about guaranteed parallel progress?

Thread 1:

```
atomic {
  while (true);
}
```

Thread 2:

```
atomic { }
print “Hello”;
```

(Similar to Luchangco example.)

- Conventional TM view (?): “Hello” is always printed.
- With atomic as syntactic sugar for lock: “Hello” not always printed.
- According to JLS, with `atomic` removed: “Hello” not always printed!
- Fully portable code can’t tell a lock-based implementation from a roll-back based one, without nested synchronization (?)
- Does real code care?

Can distinguish with nested synchronization!

v is declared volatile/atomic

Thread 1:

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

Thread 2:

```
while (!v);
atomic { }
x = 2;
```

- In our view, no data race on x.
- Hard to handle in a purely roll-back-based implementation anyway.

LABS hp

What about faster TMs with weaker semantics?

- Either destroys simple interleaving-based view of threads, and/or
- Adds unintuitive, unproven restrictions, *e.g.*
  - No movement of code *into* critical sections.
  - Need separate shared & private versions of data types.
- Parallel programming is hard enough!
  - Let's not make it harder!

Conclusions

- Atomic sections defined as a simple short-hand for lock acquisition give us:
  - All the synchronization benefits of TM.
  - Simple semantics.
  - Support for roll-back based implementations.
- Open questions:
  - What can we do about performance?
  - Is failure atomicity practically feasible and worth the added complexity?

Questions?