Transactional Memory and concurrency

Simon Peyton Jones
Microsoft Research, Cambridge
The Context

Multicore

Parallel programming essential

Task parallelism
- Explicit threads
- Synchronise via locks, messages

Data parallelism
Operate simultaneously on bulk data

This talk
1.30pm today...
Task parallelism: state of play

• The state of the art in concurrent programming is 30 years old: locks and condition variables. (In Java/C#: synchronised methods, lock statements, Monitor.wait.)

• Locks and condition variables are fundamentally flawed: it’s like building a sky-scraper out of bananas.

• This presentation describes significant recent progress: bricks and mortar instead of bananas.
What’s wrong with locks?

A 30-second review:

- **Races**: due to forgotten locks
- **Deadlock**: locks acquired in “wrong” order.
- **Lost wakeups**: forgotten notify to condition variable
- **Diabolical error recovery**: need to restore invariants and release locks in exception handlers

- These are serious problems. But even worse...
Locks are absurdly hard to get right

Scalable double-ended queue: one lock per cell

No interference if ends “far enough” apart

But watch out when the queue is 0, 1, or 2 elements long!
Locks are absurdly hard to get right

<table>
<thead>
<tr>
<th>Coding style</th>
<th>Difficulty of concurrent queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential code</td>
<td>Undergraduate</td>
</tr>
</tbody>
</table>
Locks are absurdly hard to get right

<table>
<thead>
<tr>
<th>Coding style</th>
<th>Difficulty of concurrent queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential code</td>
<td>Undergraduate</td>
</tr>
<tr>
<td>Locks and condition variables</td>
<td>Publishable result at international conference</td>
</tr>
</tbody>
</table>
### Atomic memory transactions

<table>
<thead>
<tr>
<th>Coding style</th>
<th>Difficulty of concurrent queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential code</td>
<td>Undergraduate</td>
</tr>
<tr>
<td>Locks and condition variables</td>
<td>Publishable result at international conference</td>
</tr>
<tr>
<td><strong>Atomic blocks</strong></td>
<td><strong>Undergraduate</strong></td>
</tr>
</tbody>
</table>
Atomic memory transactions

atomic { ... sequential get code ... }

- To a first approximation, just write the sequential code, and wrap atomic around it
- All-or-nothing semantics: Atomic commit
- Atomic block executes in Isolation
- Cannot deadlock (there are no locks!)
- Atomicity makes error recovery easy (e.g. exception thrown inside the get code)
How does it work?

atomic { ... <code> ... }

One possibility:

- Execute <code> without taking any locks
- Each read and write in <code> is logged to a thread-local transaction log
- Writes go to the log only, not to memory
- At the end, the transaction tries to commit to memory
- Commit may fail; then transaction is re-run
Blocking

```plaintext
atomic { if n_items == 0 then retry else ...remove from queue... }
```

- **retry** says “abandon the current transaction and re-execute it from scratch”
- The implementation waits until n_items changes
- No condition variables, no lost wake-ups!
Blocking composes

```java
atomic { x = queue1.getItem(); queue2.putItem( x ) }
```

- If either `getItem` or `putItem` retries, the whole transaction retries.
- So the transaction waits until `queue1` is not empty AND `queue2` is not full.
- No need to re-code `getItem` or `putItem`.
- (Lock-based code does not compose.)
atomic { x = queue1.getItem()
    ; choose
    queue2.putItem(x)
orElse
    queue3.putItem(x) }

- **orElse** tries two alternative paths
- If the first retries, it runs the second
- If both retry, the whole **orElse** retries.
atomic { x = queue1.getItem();
    ; choose
        queue2.putItem(x)
    orElse
        queue3.putItem(x) }

- So the transaction waits until
  - queue1 is non-empty, AND
  - EITHER queue2 is not full OR queue3 is not full

without touching getItem or putItem
Performance

There’s a run-time cost for TM, but

- Compiler technology and hardware support can reduce it a lot
- A “faster” program that doesn’t work right is useless
- TM allows much finer-grain locking without losing correctness, so performance may be better than when using locks
Results: concurrency control overhead

- Sequential baseline (1.00x)
- Coarse-grained locking (1.13x)
- Fine-grained locking (2.57x)
- Traditional STM (5.69x)
- Direct-update STM (2.04x)
- Direct-update STM + compiler integration (1.46x)

Workload: operations on a red-black tree, 1 thread, 6:1:1 lookup:insert:delete mix with keys 0..65535

Scalable to multicore
Results: scalability

- Fine-grained locking
- Direct-update STM + compiler integration
- Coarse-grained locking
- Traditional STM

Microseconds per operation vs. #threads
Remember this

- Atomic blocks (atomic, retry, orElse) are a real step forward
- **It’s like using a high-level language instead of assembly code**: whole classes of low-level errors are eliminated.
- Not a silver bullet:
  - you can still write buggy programs;
  - concurrent programs are still harder to write than sequential ones;
  - aimed at shared memory
- Very hot research area – expect developments
- Available in STM Haskell today: http://haskell.org/ghc
Backup slides
Starvation

- A worry: could the system “thrash” by continually colliding and re-executing?
- No: one transaction can be forced to re-execute only if another succeeds in committing. That gives a strong progress guarantee.
- But a particular thread could perhaps starve.
- No automatic solution can possibly be adequate
No I/O inside transactions

atomic \{ \text{if} \ (x>y) \ \text{then launchMissiles} \}

- The transaction might see \((x>y)\) because it pauses between reading \(x\) and reading \(y\)
- So we must not call \text{launchMissiles} until the transaction commits
- Simple story: no I/O inside transactions
• Transactional output is easy:

• Input is a bit harder, because of the need to make sure the transactional input buffer is filled enough.

Transactional output → Shared (transational) memory → I/O thread