Language constructs for transactional memory

Tim Harris

Disclaimer: these are my personal opinions
Untangling “atomic” from TM

Hiding TM from programmers

Current performance
Example: double-ended queue

- Support push/pop on both ends
- Allow concurrency where possible
- Avoid deadlock
Implementing this: TM

Class Q {
    QElem leftSentinel;
    QElem rightSentinel;

    void pushLeft(int item) {
        QElem e = new QElem(item);
        do {
            StartTx();
            TxWrite(&e.right, TxRead(&this.leftSentinel.right));
            TxWrite(&e.left, this.leftSentinel);
            TxWrite(&TxRead(&this.leftSentinel.right).left, e);
            TxWrite(&this.leftSentinel.right, e);
        } while (!CommitTx());
    }

    ...
}

Broadly based on word-based STM from “Concurrent programming without locks”
Keir Fraser & Tim Harris, ACM TOCS
Implementing this: atomic blocks

```java
Class Q {
    QElem leftSentinel;
    QElem rightSentinel;

    void pushLeft(int item) {
        do {
            StartTx();
            QElem e = new QElem(item);
            TxWrite(&e.right, TxRead(&this.leftSentinel.right));
            TxWrite(&e.left, this.leftSentinel);
            TxWrite(&TxRead(&this.leftSentinel.right).left, e);
            TxWrite(&this.leftSentinel.right, e);
        } while (!CommitTx());
    }

    ...
}
```
Class Q {
    QElem leftSentinel;
    QElem rightSentinel;
    void pushLeft(int item) {
        atomic {
            QElem e = new QElem(item);
            e.right = this.leftSentinel.right;
            e.left = this.leftSentinel;
            this.leftSentinel.right.left = e;
            this.leftSentinel.right = e;
        }
    }
}

“What about I/O?”

“What if another thread tries to access one of these fields without being in an atomic block?”

“What about memory access violations, exceptions, security error logs, ...?”

“What if another atomic block updates one of these fields? Will I see the value change midway through my atomic block?”

“What happens to this object if the atomic block is rolled back?”

“What happens if I single-step through this, and there’s a conflict with a concurrent transaction?”

“What happens if this fails with an exception; are the other updates rolled back?”

“What about I/O?”

“What about memory access violations, exceptions, security error logs, ...?”

“What if another atomic block updates one of these fields? Will I see the value change midway through my atomic block?”

“What happens to this object if the atomic block is rolled back?”

“What happens if I single-step through this, and there’s a conflict with a concurrent transaction?”

“What happens if this fails with an exception; are the other updates rolled back?”

“What about I/O?”

“What about memory access violations, exceptions, security error logs, ...?”
“Atomic blocks are transactions”

Consider the question of external interaction through storage systems or databases, not as we address distributed systems.

Even in this setting, concurrent programming is extremely difficult. The dominant programming technique is based on locks, an approach that is simple but direct, but that simply does not scale with program size and complexity. To make concurrent programs work, we must identify which operations conflict, to ensure, however, that there are no deadlocks, to ensure good performance, the blocking behavior on which locking is performed against the costs of lock overhead. Perhaps the most fundamental objection, though, is that lock-based programs do not compose correctly, and fragile systems may fail when composed.

As a result, consider a basic idea: transactions demand a different level of abstraction to support transactions. Instead of being able to commit individual parts of a transaction, the transaction must be completed or failed.For example, consider a basic idea: transactions with read and delete operations. Now suppose that we want to define a transaction on some variable, and insert into the transaction, but the intermediate state is such that the variable can remain in an intermediate state, and then make it visible to other threads. If the transaction is in the middle of the lock, and the intermediate state, then it can remain in an intermediate state, which makes it visible to other threads. If the transaction is in the middle of the lock, and the intermediate state, then it can remain in an intermediate state, which makes it visible to other threads.

In this setting, atomic blocks are transactions. Consider the question of external interaction through storage systems or databases, not as we address distributed systems.

Even in this setting, concurrent programming is extremely difficult. The dominant programming technique is based on locks, an approach that is simple but direct, but that simply does not scale with program size and complexity. To make concurrent programs work, we must identify which operations conflict, to ensure, however, that there are no deadlocks, to ensure good performance, the blocking behavior on which locking is performed against the costs of lock overhead. Perhaps the most fundamental objection, though, is that lock-based programs do not compose correctly, and fragile systems may fail when composed.

As a result, consider a basic idea: transactions demand a different level of abstraction to support transactions. Instead of being able to commit individual parts of a transaction, the transaction must be completed or failed. For example, consider a basic idea: transactions with read and delete operations. Now suppose that we want to define a transaction on some variable, and insert into the transaction, but the intermediate state is such that the variable can remain in an intermediate state, and then make it visible to other threads. If the transaction is in the middle of the lock, and the intermediate state, then it can remain in an intermediate state, which makes it visible to other threads. If the transaction is in the middle of the lock, and the intermediate state, then it can remain in an intermediate state, which makes it visible to other threads.

In this setting, atomic blocks are transactions.
“Atomic blocks are locks”

- Consequences explored methodically by Menon et al (Transact ’08, SPAA ’08)
Abstractions vs implementations

- Transactional memory API
- AME
- Atomic blocks
- Synchronized blocks

- TM
- Locks
Defining “atomic” without saying “TM”

• “Strong semantics”
  – Simple interleaved execution of threads
  – If a thread starts an atomic block then only it can take steps
  – Blocking operations (e.g. “retry”, “orElse”, “blockUntil”) can be incorporated – see refs below

• This means:
  – Atomic blocks are atomic wrt normal memory accesses
  – Do not need to model conflict detection / resolution
  – Can choose whether or not to retain the effects of an atomic block that raises an exception

“Composable memory transactions”, Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy. PPoPP ’05
“Semantics of Transactional Memory and Automatic Mutual Exclusion”, Martín Abadi, Andrew Birrell, Tim Harris, Michael Isard. POPL ’08
Example: a privatization idiom

```
x_shared = true;  x = 0;

atomic {
    if (x_shared) {
        x = 100;
    }
}
```

```
atomic {
    x_shared = false;
}
x++;
```

```
atomic {
    x_shared = false;
}
x++;
```
Example: a privatization idiom

```c
x_shared = true;  x = 0;

atomic {
    if (x_shared) {
        x = 100;
    }
}

atomic {
    x_shared = false;
}

x++;
```
Example: a privatization idiom

```cpp
x_shared = true;  x = 100;

atomic {
    if (x_shared) {
        x = 100;
    }
}

x++;  

atomic {
    x_shared = false;
}  

x++;  
```
Example: a privatization idiom

```
x_shared = false;  x = 100;
```

```
atomic {
  if (x_shared) {
    x = 100;
  }
}
```

```
atomic {
  x_shared = false;
}
```

```
x++;  
```

Example: a privatization idiom

```c
x_shared = false;    x = 101;

atomic {
    if (x_shared) {
        x = 100;
    }
}

x++;

atomic {
    x_shared = false;
}
```
Example: a privatization idiom

\[
x_{\text{shared}} = \text{true}; \quad x = 0;
\]

\[
\begin{align*}
\text{atomic} \{ \\
\quad \text{if } (x_{\text{shared}}) \{ \\
\quad\quad x = 100; \\
\quad\} \\
\} \\
\text{atomic} \{ \\
\quad x_{\text{shared}} = \text{false}; \\
\} \\
\quad x++; 
\end{align*}
\]
Example: a privatization idiom

```
x_shared = true; x = 0;
```

```
atomic {
    if (x_shared) {
        x = 100;
    }
}
```

```
atomic {
    x_shared = false;
}
x++;  
```
Example: a privatization idiom

```c
atomic {
    if (x_shared) {
        x = 100;
    }
}
```

```c
atomic {
    x_shared = false;
}
```

```c
x++; x = 0;
```
Example: a privatization idiom

```c
x_shared = false;  
x = 0;
```

```
atomic {
    if (x_shared) {
        x = 100;
    }
}
```
Example: a privatization idiom

```c
x_shared = false;  x = 1;

atomic {
    if (x_shared) {
        x = 100;
    }
}

atomic {
    x_shared = false;
}

x++;```

```c
atomic {
    x_shared = false;
}

x++;```
Strong semantics

- We’ve not talked about “inconsistent reads”, “roll backs”, “in-place vs lazy updates”, “weak atomicity”, “strong atomicity”, ...
- We’ve not ruled out anything (e.g. I/O)
- We’ve not considered program transformations

- Is this a pipe-dream? Can we implement it?
Untangling “atomic” from TM

Hiding TM from programmers

Current performance
Example: a privatization idiom

```
x_shared = true;  x = 0;
```

```
atomic {
    if (x_shared) {
        x = 100;
    }
}
```

```
atomic {
    x_shared = false;
}
x++;  
```
Example: a privatization idiom

```c
x_shared = true;  x = 0;
```

```c
atomic {
    if (x_shared) {
        x = 100;
    }
}
```

```c
atomic {
    x_shared = false;
}
x++;`
Example: a privatization idiom

```c
atomic {
    if (x_shared) {
        x = 100;
    }
}
```

```c
x_shared = true;  x = 100;
```

```c
atomic {
    x_shared = false;
}
```

```c
x++;  
```
Example: a privatization idiom

```c
atomic {
    if (x_shared) {
        x = 100;
    }
}

x_shared = false;  x = 100;
```

atomic {
    x_shared = false;
}

x++;
Example: a privatization idiom

\[ x_{\text{shared}} = \text{false}; \quad x = 101; \]
Example: a privatization idiom

\[
x\_shared = \text{false}; \quad x = 0;
\]

```c
atomic {
    if (x\_shared) {
        x = 100;
    }
}
```

Not valid execution under strong semantics

```c
atomic {
    x\_shared = \text{false};
    x++; 
}
```
Example: a privatization idiom

```
x_shared = false;  x = 0;

atomic {
  if (x_shared) {
    x = 100;
  }
}

atomic {
  x_shared = false;
}
x++;```
Hiding TM from programmers

**Programming discipline(s)**
What does it mean for a program to use the constructs correctly?

**Strong semantics**
atomic, retry, ..... What, ideally, should these constructs do?

**Low-level semantics & actual implementations**
Transactions, lock inference, optimistic concurrency, program transformations, weak memory models, ...

Microsoft Research
Programming disciplines

• Based on a program’s execution under the strong semantics

<table>
<thead>
<tr>
<th>All programs</th>
<th>Violation-free programs</th>
<th>Obeying dynamic separation</th>
<th>Obeying static separation</th>
</tr>
</thead>
<tbody>
<tr>
<td>More programs satisfy the discipline</td>
<td>Fewer programs satisfy the discipline</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Based on a program’s execution under the strong semantics.
Static separation

- Atomic blocks can only access local variables and designated "atomic variables"
- "atomic variables" cannot be accessed outside atomic blocks

```java
Class Q {
    atomic QElem leftSentinel;
    atomic QElem rightSentinel;

    void pushLeft(int item) {
        atomic {
            QElem e = new QElem(item);
            e.right = this.leftSentinel.right;
            e.left = this.leftSentinel;
            this.leftSentinel.right.left = e;
            this.leftSentinel.right = e;
```
Delaunay triangulation

“Delaunay Triangulation with Transactions and Barriers”
Delaunay triangulation (2)

- Initialization
- Independent parallel work
- Possibly-conflicting parallel work
- Output

Synchronization barriers
Dynamic separation

- Add explicit operations to indicate whether data is accessed inside atomic blocks, or accessed outside them
- Correctly synchronized: data is always in the correct mode when it is accessed
- Robust dynamic checking is possible:
  - Either the program runs with strong semantics
  - Or it fails with an error
Violation freedom (VF)

- Allow data’s access mode to change implicitly
- To be correctly synchronized:
  - Conflicting data accesses must not be attempted concurrently inside & outside atomic blocks
- Reminiscent of rules for programs to be free from data races
Example: a privatization idiom

```c
x_shared = true;  x = 0;
```

```c
class {
    x_shared = false;
    x++;
}
```
Programming with violations

Copy to a thread-local map (no concurrency control)

Atomically: route on the local map and merge back

C# version of Labyrinth, derived from “STAMP: Stanford Transactional Applications for Multi-Processing” Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun, IISWC ’08
Strong atomicity

• Similar to typical HTM behavior
• Trade off implementation complexity for (hopefully) scalability & straight-line speed,
• Two recent approaches:
  – “Dynamic Optimization for Efficient Strong Atomicity”, Schneider et al, OOPSLA ’08
  – “Transactional memory with strong atomicity using off-the-shelf memory protection hardware”, Abadi et al, PPoPP ’09
Strong atomicity \implies strong semantics

- Can tmp1==true, tmp2==0?
- Under strong semantics: no
- Under plausible implementations with strong atomicity: yes

Example from “What do high-level memory models mean for transactions?”
Dan Grossman, Jeremy Manson, William Pugh, MSPC ’06
Class Q {
    QElem leftSentinel;
    QElem rightSentinel;
    void pushLeft(int item) {
        atomic {
            QElem e = new QElem(item);
            e.right = this.leftSentinel.right;
            e.left = this.leftSentinel;
            this.leftSentinel.right.left = e;
            this.leftSentinel.right = e;
        }
    }
}

“Atomic blocks are for building shared memory data structures; use explicit synchronization for I/O

“In correctly synchronized programs, any use of speculation must be hidden by the implementation

“Again, ...”

“Again, ...”

“If it's a conflicting access, then the program is not correctly synchronized

“In correctly synchronized programs, speculation won’t be revealed by the implementation

“This depends on the language (Personally: no roll back, to avoid overhead on lock-inference impl's)
Programming abstraction is “atomic blocks”. Just shared memory operations (including allocation, including GC).

Implementation may use system calls, e.g. allocating memory.

Open nesting, boosting are “TM-level” operations, possibly used in the implementation of allocation during atomic blocks. Mark uses as “unsafe” if explicit in applications.
Untangling “atomic” from TM

Hiding TM from programmers

Current performance
Perf. figures depend on...

- **Workload**: What do the atomic blocks do? How long is spent inside them?
- **Baseline implementation**: Mature existing compiler, or prototype?
- **Intended semantics**: Support static separation? Violation freedom? Strong atomicity?
- **STM implementation**: In-place updates, deferred updates, eager/lazy conflict detection, visible/invisible readers?
- **STM-specific optimizations**: e.g. to remove or downgrade redundant TM operations
- **Integration**: e.g. dynamically between the GC and the STM, or inlining of STM functions during compilation
- **Implementation effort**: low-level perf tweaks, tuning, etc.
- **Hardware**: e.g. performance of CAS and memory system
Labyrinth

- STAMP v0.9.10
- 256x256x3 grid
- Routing 256 paths
- Almost all execution inside atomic blocks
- Atomic blocks can attempt 100K+ updates
- C# version derived from original C
- Compiled using Bartok, whole program mode, C# -> x86 (~80% perf of original C with VS2008)
- Overhead results with Core2 Duo running Windows Vista

“STAMP: Stanford Transactional Applications for Multi-Processing”
Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun, IISWC 2008
Sequential overhead

STM implementation supporting static separation
In-place updates
Lazy conflict detection
Per-object STM metadata
Addition of read/write barriers before accesses
  Read: log per-object metadata word
  Update: CAS on per-object metadata word
  Update: log value being overwritten
Sequential overhead

Log size grows with #locations accessed
Consequential reduction in validation time
1st level: per-thread hashtable (1024 entries)
2nd level: per-object bitmap of updated fields
**Sequential overhead**

1. STM: 11.86
2. Dynamic filtering: 3.14
3. Dataflow opts: 1.99

**Data-flow optimizations**
- Remove repeated log operations
- Open-for-read/update on a per-object basis
- Log-old-value on a per-field basis
- Remove concurrency control on newly-allocated objects
Sequential overhead

1-thread, normalized to seq. baseline

STM: 11.86
Dynamic filtering: 3.14
Dataflow opts: 1.99
Filter opts: 1.71

Inline optimized filter operations

```
mov eax <- obj_addr
and eax <- eax, 0xffc
mov ebx <- [table_base + eax]
cmp ebx, obj_addr
```

Re-use table_base between filter operations
Avoids caller save/restore on filter hits
Sequential overhead

1-thread, normalized to seq. baseline

- STM: 11.86
- Dynamic filtering: 3.14
- Dataflow opts: 1.99
- Filter opts: 1.71
- Re-use logs: 1.71

Re-use STM logs between transactions
Reduces pressure on per-page allocation lock
Reduces time spent in GC
Scaling – Labyrinth

Execution time / seq. baseline

#Threads

Static separation
Strong atomicity

1.0 = wall-clock execution time of sequential code without concurrency control
Scaling – Delaunay

- Execution time / seq. baseline
- #Threads
- Static separation
- Strong atomicity

Graph showing performance comparison between Static separation and Strong atomicity across different thread counts.
Scaling – Genome

![Graph showing execution time relative to sequential baseline across different thread counts for different separation methods.]

- Static separation
- Strong atomicity

Execution time / seq. baseline vs. #Threads.
Scaling – Vacation

![Graph showing execution time vs. number of threads for Static separation and Strong atomicity.](image)
Untangling “atomic” from TM

Hiding TM from programmers

Current performance
Abstractions vs implementations

- Transactional memory API
- AME
- Atomic blocks
- Synchronized blocks
- TM
- Locks
Future directions

- Which programming discipline should we settle on
  - ...in a language like C#?
  - ...in future languages?
- H/W acceleration based on mature optimized S/W implementations
- Progress guarantees, interactions with implementation techniques and performance
- What asymptotic bounds on STM performance can we give when supporting different programming disciplines?
- How do we define correctness of an STM interface, as opposed to the whole language implementation?
Acknowledgements

- Most of the work described in these slides has been collaborative; I’d like to thank colleagues at MSR Cambridge, MSR Redmond, MSR Mountain View, the Microsoft Parallel Computing Platform Group, the University of Cambridge Computer Lab, and the MSR-BSC joint research centre.

- Material is drawn from the following publications:
  - Martín Abadi, Andrew Birrell, Tim Harris, Johnson Hsieh, Michael Isard. “Implementation and use of transactional memory with dynamic separation”. Compiler Construction, March 2009
  - Martín Abadi, Tim Harris, Mojtaba Mehrara. “Transactional memory with strong atomicity using off-the-shelf memory protection hardware”. PPoPP, February 2009
  - Martín Abadi, Tim Harris, Katherine Moore. “A model of dynamic separation for transactional memory”. CONCUR, August 2008
  - Martín Abadi, Andrew Birrell, Tim Harris, Michael Isard. “Semantics of Transactional Memory and Automatic Mutual Exclusion”. POPL, January 2008
  - Keir Fraser, Tim Harris. “Concurrent programming without locks”. ACM TOCS, May 2007
  - Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy. “Composable memory transactions” PPoPP, June 2005

- The material on performance is current at Jan 2009, and reflects a slightly later more optimized implementation than that described in the PPoPP 2009 paper.