Transactional memory & atomic blocks

Tim Harris


AMD quad-core
Sun Niagara-2
Example: double-ended queue

<table>
<thead>
<tr>
<th>Left sentinel</th>
<th></th>
<th></th>
<th>Thread 1</th>
<th></th>
<th>Thread 2</th>
<th></th>
<th>Right sentinel</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td>20</td>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>

Thread 1

Thread 2
Example: coarse-grained locking

```java
class Q {
    Lock qLock = new Lock();
    QElem leftSentinel;  
    QElem rightSentinel;

    void pushLeft(int item) {
        QElem e = new QElem(item);
        qLock.Acquire();
        e.right = this.leftSentinel.right;
        e.left = this.leftSentinel;
        this.leftSentinel.right.left = e;
        this.leftSentinel.right = e;
        qLock.Release();
    }
}
```

Thread 1

Thread 2
Example: fine-grain locking

```java
public class Q {
    Lock leftLock = new Lock();
    Lock rightRlock = new Lock();
    QELEM leftSentinel;
    QELEM rightSentinel;

    void pushLeft(int item) {
        QELEM e = new QELEM(item);
        leftLock.Acquire();
        e.right = this.leftSentinel.right;
        e.left = this.leftSentinel;
        this.leftSentinel.right.left = e;
        this.leftSentinel.right = e;
        leftLock.Release();
    }

    ...
}
```
Example: fine-grain locking

Left sentinel

Right sentinel

leftLock

rightLock
What we want

Libraries build layered concurrency abstractions

Concurrency primitives

Hardware
What we have

Locks and condition variables
(a) are hard to use and
(b) do not compose
Atomic blocks

Atomic blocks built over transactional memory
3 primitives: atomic, retry, orElse
Atomic memory transactions

Item PopLeft() {
    atomic { ... sequential 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 `PopLeft` code)

Like database transactions

ACID
Atomic blocks compose (locks do not)

```c
void GetTwo() {
    atomic {
        i1 = PopLeft();
        i2 = PopLeft();
    }
    DoSomething( i1, i2 );
}
```

- Guarantees to get two consecutive items
- PopLeft() is unchanged
- Cannot be achieved with locks (except by breaking the PopLeft abstraction)
Blocking: how does PopLeft wait for data?

```java
Item PopLeft() {
    atomic {
        if (leftSentinel.right == rightSentinel) {
            retry;
        } else {
            ...remove item from queue...
        }
    }
}
```

- **retry** means “abandon execution of the atomic block and re-run it (when there is a chance it’ll complete)”
- No lost wake-ups
- No consequential change to GetTwo(), even though GetTwo must wait for there to be two items in the queue
Choice: waiting for either of two queues

- **do** {...this...} *orelse* {...that...} tries to run “this”
- If “this” retries, it runs “that” instead
- If both retry, the do-block retries. GetEither() will thereby wait for there to be an item in *either* queue

```c
void GetEither() {
    atomic {
        do { i = Q1.Get(); }
        orelse { i = Q2.Get(); }
        R.Put( i );
    }
}
```
Programming with atomic blocks

With locks, you think about:

• Which lock protects which data? What data can be mutated when by other threads? Which condition variables must be notified when?
• None of this is explicit in the source code

With atomic blocks you think about

• What are the **invariants** (e.g. the tree is balanced)?
• Each atomic block maintains the invariants
• **Purely sequential reasoning** within a block, which is dramatically easier
• Much easier setting for static analysis tools
Summary so far

- 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;
  - just aimed at shared memory.
- But the improvement is very substantial
State of the art ~ 2003

Normalised execution time

- Coarse-grained locking (1.13x)
- Sequential baseline (1.00x)
- Fine-grained locking (2.57x)
- Traditional STM (5.69x)

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

- **Direct-update STM**
  - Allow transactions to make updates in place in the heap
  - Avoids reads needing to search the log to see earlier writes that the transaction has made
  - Makes successful commit operations faster at the cost of extra work on contention or when a transaction aborts

- **Compiler integration**
  - Decompose the transactional memory operations into primitives
  - Expose the primitives to compiler optimization (e.g. to hoist concurrency control operations out of a loop)

- **Runtime system integration**
  - Integration with the garbage collector or runtime system components to scale to atomic blocks containing 100M memory accesses
  - Memory management system used to detect conflicts between transactional and non-transactional accesses
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
Direct update STM

• Transactional write:
  – Lock objects before they are written to (abort if another thread has that lock)
  – Log the overwritten data – we need it to restore the heap case of retry, transaction abort, or a conflict with a concurrent thread

• Transactional read:
  – Log a version number we associate with the object

• Commit:
  – Check the version numbers of objects we’ve read
  – Increment the version numbers of object we’ve written
Example: contention between transactions

Thread T1

```c
int t = 0;
atomic {
    t += c1.val;
    t ++;
    t += c2.val;
}
```

T1’s log:

```
ver = 200
val = 40
c2
```

Thread T2

```c
atomic {
    t = c1.val;
    t ++;
    c1.val = t;
}
```

T2’s log:

```
ver = 100
val = 10
c1
```

```
ver = 200
val = 40
c2
```
Example: contention between transactions

Thread T1

```java
int t = 0;
atomic {
    t += c1.val;
    t ++;
    c1.val = t;
}
```

T1’s log:

c1.ver=100

c1: version 100

T2

```java
atomic {
    t = c1.val;
    t ++;
    c1.val = t;
}
```

T2’s log:

T1 reads from c1: logs that it saw version 100

c1: version 100

c1: val = 10

c2: ver = 200

c2: val = 40
Example: contention between transactions

Thread T1

int t = 0;
atomic {
    t += c1.val;
    t += c2.val;
}

T1’s log:
c1.ver=100

d1

Thread T2

atomic {
    t = c1.val;
    t ++;
    c1.val = t;
}

d2

T2’s log:
c1.ver=100

T2 also reads from c1: logs that it saw version 100
Example: contention between transactions

Thread T1

```c
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

T1’s log:

- c1.ver=100
- c2.ver=200

Thread T2

```c
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```

T2’s log:

- c1.ver=100

Suppose T1 now reads from c2, sees it at version 200
Example: contention between transactions

Thread T1

```c
int t = 0;
atomic {
    t += c1.val;
    t += c2.val;
}
```

T1’s log:

c1.ver=100

c2.ver=200

Thread T2

```c
atomic {
    t = c1.val;
    t ++;
    c1.val = t;
}
```

T2’s log:

c1.ver=100

lock: c1, 100

Before updating c1, thread T2 must lock it: record old version number
Example: contention between transactions

Thread T1

int t = 0;
atomic {
    t += c1.val;
    t += c2.val;
}

T1’s log:
c1.ver=100
c2.ver=200

Thread T2

atomic {
    t = c1.val;
    t ++;
    c1.val = t;
}

T2’s log:
c1.ver=100
lock: c1, 100
c1.val=10

(1) Before updating c1.val, thread T2 must log the data it’s going to overwrite

(2) After logging the old value, T2 makes its update in place to c1

T1’s log:
locked:T2
c1
val = 11
val = 40

T2’s log:
ver = 200
c2

(1) Before updating c1.val, thread T2 must log the data it’s going to overwrite

(2) After logging the old value, T2 makes its update in place to c1
Example: contention between transactions

Thread T1

```c
int t = 0;
atomic {
    t += c1.val;
    t ++;
    c1.val = t;
}
```

T1’s log:

- c1.ver=100
- c2.ver=200

Thread T2

```c
atomic {
    t = c1.val;
    t ++;
    c1.val = t;
}
```

T2’s log:

- c1.ver=100
- lock: c1, 100
- c1.val=10

(1) Check the version we locked matches the version we previously read

(2) T2’s transaction commits successfully. Unlock the object, installing the new version number
Example: contention between transactions

Thread T1

```c
int t = 0;
atomic {
    t += c1.val;
    t ++;
    c1.val = t;
}
```

Thread T2

```c
atomic {
    t = c1.val;
    t += c2.val;
    c1.val = t;
}
```

T1’s log:

- c1.ver=100
- c2.ver=100

T2’s log:

(1) T1 attempts to commit. Check the versions it read are still up-to-date.

(2) Object c1 was updated from version 100 to 101, so T1’s transaction is aborted and re-run.
Zombie transactions

Initially: \( x==y==z==0 \)

- \( temp==0 \) is the only correct result here if these blocks really are atomic
Zombie transactions

Direct update, lazy conflict detection

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

```
atomic {
    if (x != y) z = 1;
}
```

```
temp = z;
```

- $x == 0$
- $y == 0$
- $z == 0$
Zombie transactions

Direct update, lazy conflict detection

atomic {
    x = 1;
    y = 1;
}

atomic {
    if (x != y) z = 1;
}

temp = z;

• $x = 0$
• $y = 0$
• $z = 0$
Zombie transactions

Direct update, lazy conflict detection

atomic {
    x = 1;
    y = 1;
}

atomic {
    if (x != y) z = 1;
}

temp = z;

• x == 1
• y == 1
• z == 0
Zombie transactions

Direct update, lazy conflict detection

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

```
atomic {
  if (x != y) z = 1;
}
```

```
temp = z;
```

- $x == 1$
- $y == 1$
- $z == 1$
Zombie transactions

Direct update, lazy conflict detection

atomic {
    x = 1;
    y = 1;
}

atomic {
    if (x != y) z = 1;
}

temp = z;

• x == 1
• y == 1
• z == 1
Strong isolation

• Add a mechanism to detect conflicts between tx and normal accesses
  – e.g. ‘z’ in this example

• We would like:
  – Implementation flexibility – e.g. different STMs
  – No overhead on non-transactional accesses
  – Predictable performance
  – Little overhead over weak atomicity
Strong isolation: implementation

- Physical address space
- Virtual address space

Normal-heap

Tx-heap

Normal memory accesses

Memory accesses from atomic blocks
Writes from atomic blocks

1. Atomic block attempts to write to a field of an object
2. Revoke direct access to the page holding the direct view of the object
Writes from atomic blocks

3. Use underlying STM write primitives

Physical address space

Virtual address space

Normal-heap

Tx-heap

Normal memory accesses

Memory accesses from atomic blocks
Writes from atomic blocks

4A. Restore direct access once the underlying transaction has finished.
Conflicting normal access

4B. Access violation (AV) delivered to a normal thread accessing that page: wait for TX
Performance 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 (TDRF)?
- **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

1-thread, normalized to seq. baseline

11.86
Sequential overhead

<table>
<thead>
<tr>
<th></th>
<th>STM</th>
<th>Dynamic filtering</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st level</td>
<td>11.86</td>
<td>3.14</td>
</tr>
<tr>
<td>2nd level</td>
<td>1.99</td>
<td>1.71</td>
</tr>
<tr>
<td>3rd level</td>
<td>1.71</td>
<td>1.71</td>
</tr>
</tbody>
</table>

Dynamic filtering to remove redundant logging

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

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

```assembly
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
Scaling – Labyrinth

Execution time / seq. baseline

- Weak isolation
- Strong isolation

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

![Graph showing scaling behavior with Delaunay triangulation. The x-axis represents the number of threads, ranging from 1 to 8. The y-axis represents the execution time relative to the sequential baseline. Two lines are plotted: one for weak isolation and one for strong isolation. The graph demonstrates how execution time decreases as the number of threads increases.](image-url)
Scaling – Genome

![Graph showing the execution time ratio of parallel to sequential baseline for different numbers of threads. The graph compares weak and strong isolation conditions.]
Scaling – Vacation

![Graph showing execution time versus number of threads with lines for weak and strong isolation.](image)

- Weak isolation
- Strong isolation

Execution time / seq. baseline vs. #Threads
Conclusion

• What are atomic blocks good for?
  – Shared memory data structures

• Implementations involve work throughout the software stack
  – Language design
  – Compiler
  – Language runtime system
  – OS-runtime-system interfaces

• Two different experiences
  – STM-Haksell
  – STM.Net