Abstract nested transactions

Tim Harris (MSR Cambridge)
Srđan Stipić (BSC)
Atomic blocks and scalability

• Typical implementations of atomic blocks let them execute concurrently as long as there are no conflicts at an object level

```plaintext
atomic { tmp1 = o1.x; }  atomic { tmp2 = o1.x; }
atomic { o1.x = 17; }    atomic { o2.x = 42; }
atomic { o1.x = 42; }    atomic { o1.y = 17; }
atomic { tmp1 = o1.x; }  atomic { o1.y = 17; }
```
Problem: benign conflicts

• This provides a way for programmers to anticipate which transactions will run concurrently and which cannot
  – Unlike hashing on heap addresses

• Programs can suffer from ‘benign’ conflicts
  – Informally: conflicting operations where “the conflict doesn’t really matter”
#1 – Shared temporaries

```java
atomic { // Tx-1
    workOn(g_o1);
}
atomic { // Tx-2
    workOn(g_o2);
}

void workOn(Object o) {
    g_temp = o;
    // Work on ‘g_temp’
}
```

- Transactional version of ‘xlisp’
- Red-black tree sentinel node fields
- Haskell-STM identifies transactionally-silent stores
#2 – False sharing

atomic {
    g_obj.x ++;
    // Private work
}

atomic {
    g_obj.y ++;
    // Private work
}

Both threads will update fields of ‘g_obj’. Only one will be allowed to commit.

• Different perf-counter fields
• Can be avoided by restructuring code...
• ...or by a finer-granularity of conflict detection
The operations on the shared collection commute, but the STM operations and memory accesses do not commute.

- Updates to the same perf counter
- Can avoid with open-nesting (ONTs) – but care required to retain serializability of transactions
#4 – Low-level conflicts

atomic { // Tx-1
    f = g_l.Find(1000);
} atomic { // Tx-2
    g_l.Insert(10);
}

If ‘g_l’ is a typical linked list then Tx-1’s read set will be massive and conflict with Tx-2’s update

• STM-specific hooks to trim the read set
• Need great care to ensure correctness (suppose we add DeleteFrom...)
#5 – Arbitrary choices

```java
while (true) {
    atomic { // Tx-1
        t = getAny(g_in);
        if (t == null) break;
        // Work on t
        put(g_out, t);
    }
}
```

- Open-nesting can be used directly (taking care with empty lists)
- Use randomization

Run two loops in parallel – they’ll both pick the same items and conflict...
Discussion

• Some cases can/could be handled automatically
  – Shared temporaries: recognise as a form of silent store
  – False conflicts due to granularity

• Some cases are handled by ONTs
  – Commutative operations on a collection
  – Arbitrary removal from a work-queue

• Other cases use manual optimization interfaces
  – Low-level conflicts in linked-list operations

Do this transparently in the implementation
Use randomization?
Develop analyses or new dynamic techniques?
Overview: abstract nested transactions

- ANTs identify possible benign conflicts in the source code
  - We do this manually
  - It could be automated in the future
- Our new syntax is semantically transparent
  - Impacts the program’s performance, not possible behavior
  - Poor usage of ANTs may slow down a program; it won’t make it crash
#2 – False sharing

Both threads will update fields of ‘g_obj’. Only one will be allowed to commit.

- Different perf-counter fields
- Can be avoided by restructuring code...
- ...or by a finer-granularity of conflict detection
Commutativity & layering

atomic { // Tx-1
    ant { g.Insert(100, v1); }
    // Private work
}

atomic { // Tx-2
    ant { g.Insert(200, v2); }
    // Private work
}

The operations on the shared collection commute, but the STM operations and memory accesses do not

• Updates to the same perf counter
• Can avoid with open-nesting (ONTs) – but care required to retain serializability of transactions
What does this actually do?

- **Goal is to**
  - Detect conflicts experienced by an ANT
  - Upon conflict just re-execute the ANT, not whole tx

- **Do this by**
  - Tracking the inputs to the ANT (values it reads from the heap, variables it reads from)
  - Tracking the outputs from the ANT (values it writes to the heap, variables it updates, result value/exception)

- **In case of conflict**
  - Re-execute the ANT with the same inputs
  - Check it produces the same outputs
Why can it help?

- Before:
  - First ANT execution
  - ANT invalidated by physical conflicts in this interval
  - Fail to commit

- After:
  - First ANT execution
  - Re-executed ANT recovers from physical conflict
  - Commit
  - Re-execute ANTs
Basic implementation (GHC)

```haskell
r0 = <LOTS-OF-WORK> ;
ant { o1.ctr ++ } ;
r1 = <LOTS-OF-WORK> ;
ant { o2.ctr ++ } ;
```

### Ordinary transaction log

<table>
<thead>
<tr>
<th>Addr</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>0x6004</td>
<td>400</td>
<td>500</td>
</tr>
</tbody>
</table>

### ANT log

<table>
<thead>
<tr>
<th>Addr</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2000</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>0x3000</td>
<td>40</td>
<td>41</td>
</tr>
</tbody>
</table>

### ANT action list

- Closure to run `{ o1.ctr ++; }`
- Closure to run `{ o2.ctr ++; }`
Commit: refresh the ANT log

- Validate the ANT log
  - OK? We’re done
  - Invalid? Discard the log and re-run ANTs

<table>
<thead>
<tr>
<th>Addr</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>0x6004</td>
<td>400</td>
<td>500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addr</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2000</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>0x3000</td>
<td>40</td>
<td>41</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addr</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>0x6004</td>
<td>400</td>
<td>500</td>
</tr>
</tbody>
</table>

- ANT log

<table>
<thead>
<tr>
<th>Addr</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2000</td>
<td>340</td>
<td>341</td>
</tr>
<tr>
<td>0x3000</td>
<td>500</td>
<td>501</td>
</tr>
</tbody>
</table>

- ANT action list

  Closure to run
  `{ o1.ctr ++; }`
  Closure to run
  `{ o2.ctr ++; }`
Commit

- Finish the commit operation:
  - Commit the ANT log into the ordinary log
  - Commit the resulting log to the heap

<table>
<thead>
<tr>
<th>Addr</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>0x6004</td>
<td>400</td>
<td>500</td>
</tr>
<tr>
<td>0x2000</td>
<td>340</td>
<td>341</td>
</tr>
<tr>
<td>0x3000</td>
<td>500</td>
<td>501</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addr</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2000</td>
<td>340</td>
<td>341</td>
</tr>
<tr>
<td>0x3000</td>
<td>500</td>
<td>501</td>
</tr>
</tbody>
</table>

ANT action list:
- Closure to run `{ o1.ctr ++; }`
- Closure to run `{ o2.ctr ++; }`
Prototype perf

7.8% of time spent in ANT sections

Execution time [s]

Number of threads

Regular Transaction
AN Transaction
ON Transaction
Conclusion

- Prototype implementation in progress in GHC
  - Fall-back to direct execution in complex cases
  - Several ideas for perf improvements
- Key argument for this approach:
  - Deal with *some* of the uses of open nesting
  - Guarantee *atomic means atomic*
  - Provide reasonable perf, good scalability