


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

`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

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

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

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

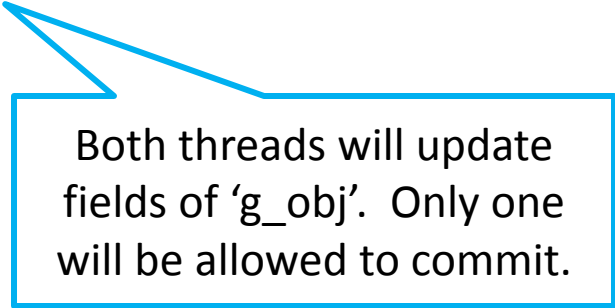
Both threads will update 'g_temp'. Only one will be allowed to commit.

- Transactional version of 'xlistp'
- Red-black tree sentinel node fields
- Haskell-STM identifies transactionally-silent stores

#2 – False sharing

```
atomic { // Tx-1
    g_obj.x ++;
    // Private work
}
```

```
atomic { // Tx-2
    g_obj.y ++;
    // Private work
}
```



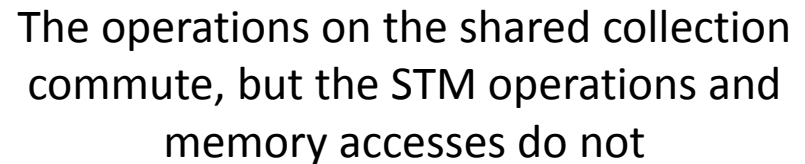
Both threads will update fields of 'g_obj'. Only one will be allowed to commit.

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

#3 – Commutativity & layering

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

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



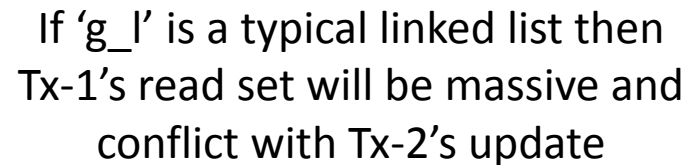
The operations on the shared collection commute, but the STM operations and memory accesses do not

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

#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

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

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

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

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

```
atomic { // Tx-1  
    ant { g_obj.x ++; }  
    // Private work  
}
```

```
atomic { // Tx-2  
    ant { 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

– 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

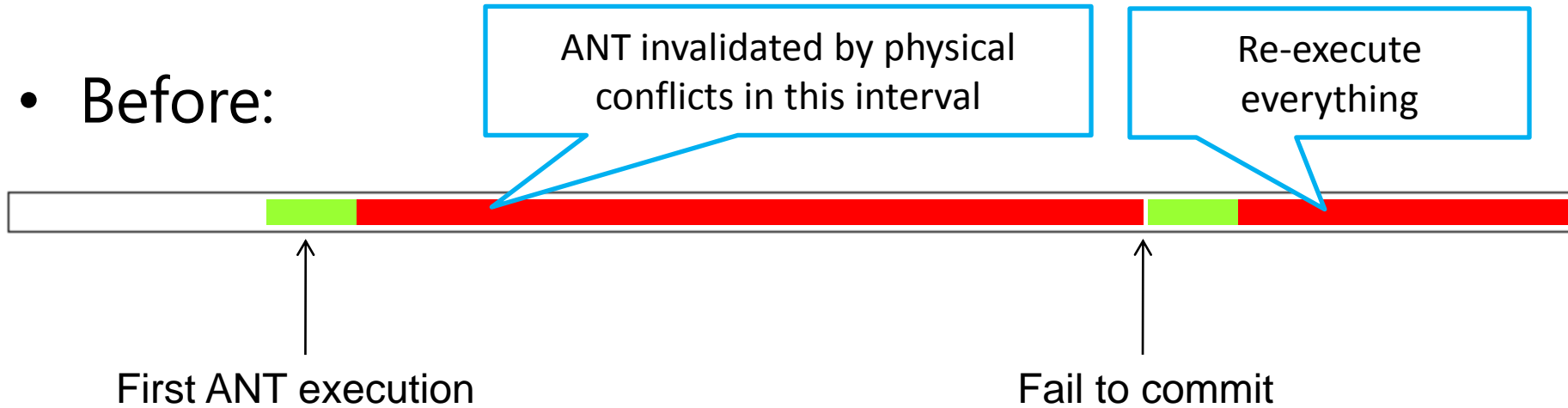
- Updates to the same perf counter
- Can avoid with open-nesting (ONTs) – but care required to retain serializability of transactions

What does this actually do?

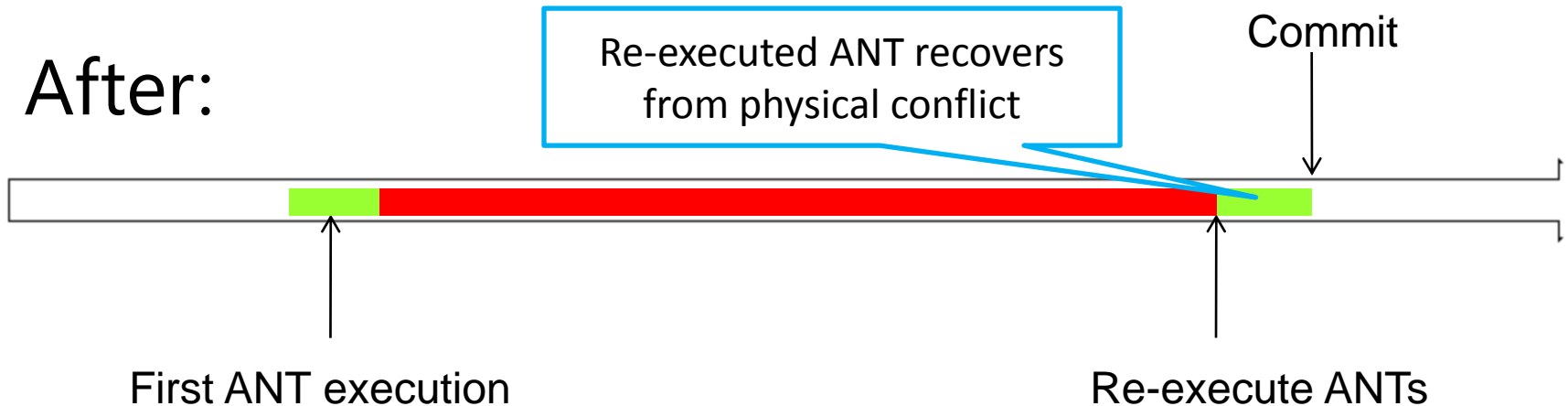
- Goal is to
 - Detect conflicts experienced by an ANT
 - Upon conflict just re-execute the ANT, not whole tx
- Do this by
 - Tracking the inputs to the ANT (values it reads from the heap, variables it reads from)
 - Tracking the outputs from the ANT (values it writes to the heap, variables it updates, 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?

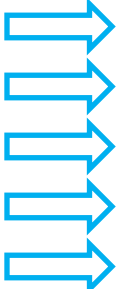
• Before:



After:



Basic implementation (GHC)



```

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

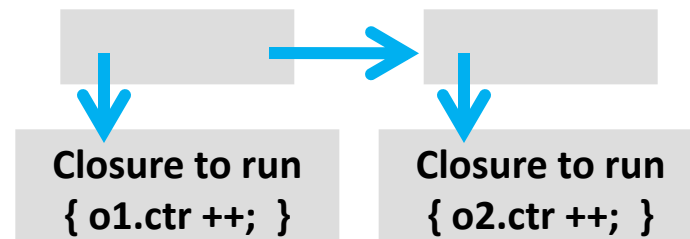
Ordinary transaction log

Addr	Old	New
0x1000	100	200
0x6004	400	500

ANT log

Addr	Old	New
0x2000	20	21
0x3000	40	41

ANT action list



Commit: refresh the ANT log

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

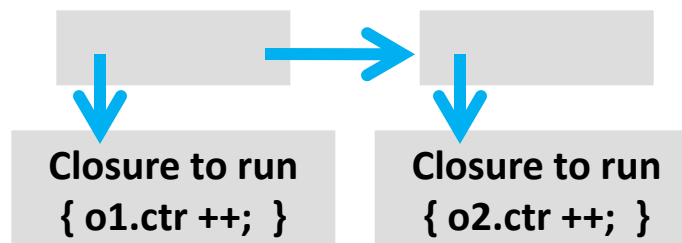
Ordinary transaction log

Addr	Old	New
0x1000	100	200
0x6004	400	500

ANT log

Addr	Old	New
0x2000	340	341
0x3000	500	501

ANT action list



Commit

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

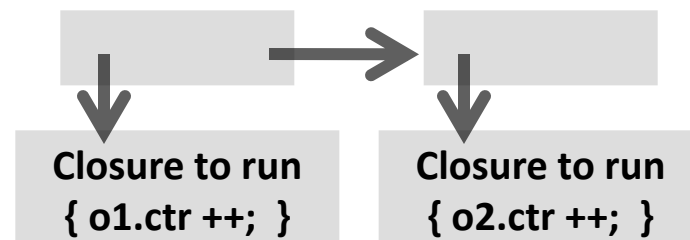
Ordinary transaction log

Addr	Old	New
0x1000	100	200
0x6004	400	500
0x2000	340	341
0x3000	500	501

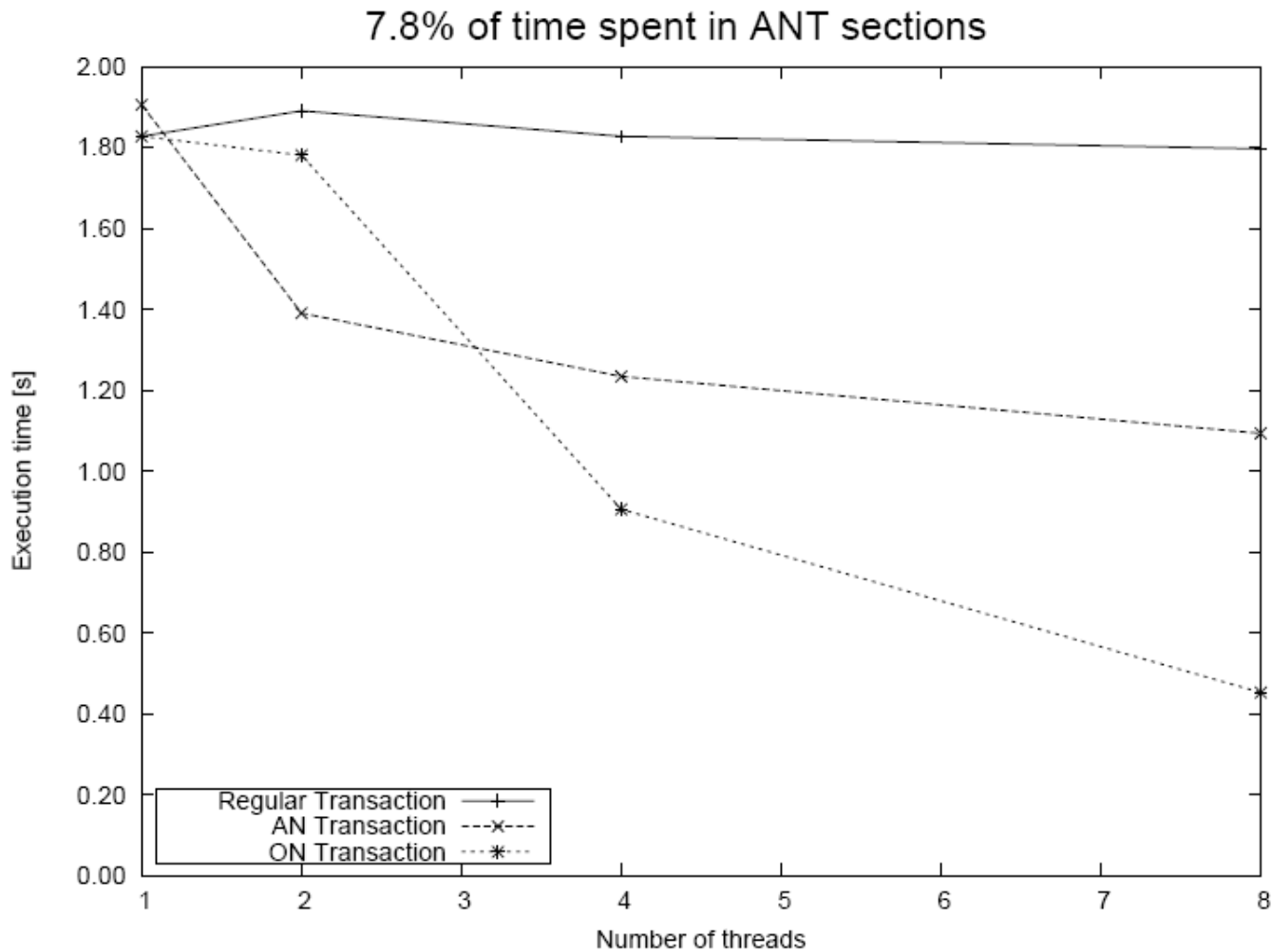
ANT log

Addr	Old	New
0x2000	340	341
0x3000	500	501

ANT action list



Prototype perf



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