MANCHESTER 1824

The University of Manchester

Chip Multiprocessors COMP35112

Lecture 11 - Transactional Memory (I)

Graham Riley

Today's Lecture

- Thread Level Speculation (TLS) was discussed in Lecture 10: tries to speculate about parallelism without any help from the programmer. Not easy!
- The concept of a **Transaction** is something most programmers have met before and found to be helpful – used widely in:
  - Database Systems
  - Distributed Systems
- **Transactional Memory** (TM) provides the programmer with an implementation of transactions within a parallel data sharing context

Properties of Transactions

- In these other contexts, what are the properties of transactions? – usually summed up by the acronym ACID:
  - Atomicity – "all-or-nothing" behaviour
  - Consistency – one valid state leads to another valid state
  - Isolation – each transaction is isolated from all others
  - Durability – once committed, the outcome will endure
- Of these, only Atomicity and Isolation are directly applicable in the TM context

Atomicity and Isolation in Transactions

- Transactions are indivisible (atomic)
- So a transaction executes *either* as a whole *or* not at all
- While a transaction is executing, none of its changes can be observed from outside the transaction
- When it completes, any changes will become apparent outside *and* they will all become apparent at the same time

Notation

We want to be able to write, e.g.

```
atomic {                  // this is a transaction
        j = j * 2 ;       // in which j is shared
        k++ ;             // and k is also shared
        }
```

and know that, when this has executed, all the shared variables have been updated consistently

Before, we might have used a lock (or locks) for this

Why not use Locks?

- Writing correct code with locks is tricky in general. Transactions should be simpler
- Locks are not *composable* – if a library uses locks, the users of that library need to know about it in order to avoid deadlocks, etc. Transactions should not suffer from this problem (one facet of being simpler)

Commit and Abort

- So a transaction:
  - Starts
  - Reads some shared variables
  - Possibly writes to (copies of?) some shared variables
  - Finally it attempts to **commit**
  - This will **succeed** if there is nothing wrong (see next slide) – and then its writes become visible
  - It will fail (**abort**) if it cannot commit – and then the entire transaction must be repeated …

Reasons to Abort

- A transaction cannot commit successfully
  - if a variable it has read has since been written to from elsewhere (R-W clash), **or**
  - if a variable it has written to has been written to from elsewhere (W-W clash)

- Tolerating such clashes would allow programs to get results which violate atomicity and/or isolation

Implementation: Readsets and Writesets

- A transaction needs to keep track of all the shared variables it has read – its **readset**
- It also needs to keep track of all the shared variables to which it has written – its **writeset**
- It uses these two sets to determine clashes (conflicts) with other transactions

Real Application: Connection Routing

- Have a number of points in a plane and a number of connections to make between some of them
- Connections cannot cross each other
- Could be tracks on a Printed Circuit Board
- Representation:

  Plane of cells = 2D array of integer values that encode what lies in each cell (source, sink, route, power, null, etc.)

  It should be possible to find a number of routes in parallel – but need to defend against intersections

Routing – continued

- Coarse grain locking: lock whole 2D array – no parallelism
- Fine grain locking: lock individual cells – have to lock all the cells in a route, mark it, and then unlock them. Quite difficult to do correctly
- Transactions: very simple. Use one transaction for each track that is required. Get good parallel performance if tracks are not too congested

Versioning

- If a transaction writes to a variable and later reads it, it needs to see the value it wrote, not the original value
- If the transaction aborts, the original value needs to still be there
- So in general we will have more than one version of each shared variable …
- There are two ways of implementing this:
  - Direct Update (a.k.a. Eager Versioning)
  - Deferred Update (a.k.a. Lazy Versioning)

Direct Update (Eager Versioning)

- Change each shared variable that is written to, but keep a private log of the original values
- On commit, throw away the log
- On abort, restore the original values from the log
- Trust clash detection to preserve isolation
- Efficient if aborts are rare

Deferred Update (Lazy Versioning)

- Keep a private version of everything shared that a transaction changes
- Subsequent reads should access these private versions
- On abort, throw away private versions
- On commit, copy private version values into the shared variables
- Efficient if aborts are common (not rare)

Validation (a.k.a. Conflict Detection)

- Validation is the name used for the process of checking that transactions do not clash
- It too can be done two ways:
  - Lazy Validation
  - Eager Validation
- Can actually mix these up – using one way for reads and the other for writes – but first we need to discuss the two approaches

Lazy Validation

- Do the validation when the transaction tries to commit
- At this stage all the reads and writes to shared variables are known
- All conflict detection needs to do is work out whether it would be OK to commit this transaction's results

Eager Validation

- Every write operation to a shared variable by a transaction can trigger a check for clashing transactions amongst those currently running
- Having identified transactions which clash, can choose which one(s) to abort – or even delay some instead of aborting them
- Probably more work – but aborting transactions which would eventually fail anyway saves wasted effort!
- **But**, we need to guarantee progress …

Other Design Options

- Strong Isolation – i.e. shared variables can only be accessed within transactions
- Weak Isolation – shared variables can be used outside transactions as well as inside them!
- Nesting Semantics – various different ways of understanding what it means to nest transactions
  - Necessary for composability!

Next Lecture

- There is a lot more to say about how Transactional Memory can be implemented
- Three approaches:
  - Software Transactional Memory (STM)
  - Hardware Transactional Memory (HTM)
  - Hybrid
- In particular, how can the hardware help to make the implementation more efficient?