6 • Transactional Memory

Chip Multiprocessors (ACS MPhil)
Robert Mullins
Overview

- Limitations of lock-based programming
- Transactional memory
  - Programming with TM
  - Software TM (STM)
  - Hardware TM (HTM)
Lock-based programming

• Lock-based programming is a low-level model
  – Close to basic hardware primitives
  – For some problems lock-based solutions that perform well are complex and error-prone
    • difficult to write, debug, and maintain
    • Not true of all problems

• Parallel programming for the masses
  – The majority of programmers will need to be able to produce highly parallel and robust software
Lock-based programming

• Challenges:
  – Must remember to use (the correct) locks
    • Careful to avoid when not required (for performance)
  – Coarse-grain vs. fine-grain locks
    • Simplicity
    • Unnecessary serialisation of operations
    • Also consider time taken to acquire and release locks!
  – Worries
    • Deadlock, priority inversion, convoying
    • lost wake-ups (wait on CV, but forget to signal)
  – Horribly complicated error recovery
  – Cannot even easily compose lock based programs
Lost wake-up example

**push**

```cpp
mutex::scoped_lock....
queue.push(item)
if (queue.size() == 1) m_emptyCond.notify_one()
```

**pop**

```cpp
mutex::scoped_lock....
while (queue.empty())
  m_emptyCond.wait()
Item = queue.front()
queue.pop()
return item
```
Lock-based programming

- **Deadlock**
  - We are free to do anything when we hold a lock, even take a lock on another mutex
  - This can quickly lead to deadlock if we are not careful
    - Limiting ourselves to only being able to take a single lock at a time would force us to use coarse-grain locks
    - e.g. consider maintaining two queues. These are each accessed by many different threads. We are infrequently required to transfer data from one queue to the other (atomically)

```c
// Trivial deadlock example
// Thread 1                      // Thread 2
a.lock();                      b.lock();
b.lock();                      a.lock();
...                            ...
```
Lock-based programming

- Avoiding deadlock
  - Requires programmer to adopt some sort of policy (although this isn't automatically enforced)
  - Often difficult to maintain/understand
- Lock hierarchies (and lock chaining)
  - All code must take locks in the same order
- Try and back off
  - More flexible than imposing a fixed order
  - Get first lock
  - Then try and lock additional mutexes in the required set. If we fail release locks and retry
    - `pthread_mutex_trylock`
Lock-based programming

- Composing lock-based programs
  - Consider our example of two queues
  - There is no simple way of dequeuing from one and enqueuing to the other in an atomic fashion
    - We would need to expose synchronization state and force caller to manage locks
  - Can't compose methods that block either (wait/notify)
    - How do we describe the operation where we want to dequeue from either queue, whichever has data
    - Each queue implementation blocks internally
Transactions

atomic {
    x=q0.deq();
    q1.enq(x);
}

- Focus on where atomicity is necessary rather than specific locking mechanisms
- The transactional memory system will ensure that the transaction is run in isolation from other threads
  - Transactions are typically run in parallel *optimistically*
  - If transactions perform conflicting memory accesses, we must abort and ensure none of the side-effects of the abandoned transactions are visible
Transactions

• **Atomicity** (all-or-nothing)
  – We guarantee that it appears that either all the instructions are executed or none of them are (if the transaction fails, **failure atomicity**)
  – The transaction either **commits** or **aborts**

• Transactions execute in **isolation**
  – Other operations cannot access a transaction's intermediate state.
  – The result of executing concurrent transactions must be identical to a result in which the transactions executed sequentially (**serializability**).
Transactions

void Queue::enq (int v) {
    atomic {
        // queue is full
        if (count==MAX_LEN) retry;
        buf[tail]=v;
        if (++tail == MAX_LEN) tail=0;
        count++;
    }
}

• Retry
  – Abandon transaction and try again
  – An implementation could wait until some changes occur in memory locations read by the aborted transaction
    • Or specify a specific watch set [Atomos/PLDI'06]

“Composable memory transactions”, Harris et al.
Transactions

• **Choice**
  – Try to dequeue from q0 first, if this retries (i.e. queue is empty), then try the second
  – If both retry, retry the whole orElse block

```
atomic {
    x = q0.deq();
} orElse {
    x = q1.deq();
}
```

“Composable memory transactions”, Harris et al.
Critical sections ≠ transactions

• Converting critical sections to transactions
  – pitfall: “A critical section that was previously atomic only with respect to other critical sections guarded by the same lock is now atomic with respect to all other critical sections.”

```
proc1 {
  acquire (m1)
  while (!flagA) {}
  flagB = true
  ....
  release(m1)
}

proc2 {
  acquire (m2)
  flag A=true
  while (!flagB) {}
  ....
  release(m2)
}
```

“Deconstructing Transactional Semantics: The Subtleties of Atomicity”
Colin Blundell. E Christopher Lewis. Milo M. K. Martin, WDDD, 2005)
Implementing a TM system

- Transaction granularity
  - Object, word or block
- How do we provide isolation?
  - Direct or deferred update?
    - Update object directly and keep undo log
    - Update private copy, discard or replace object
  - Also called eager and lazy versioning
- When and how do we detect conflicts?
  - Eager or lazy conflict detection?
- A software or hardware-supported implementation?
Hardware support for TM

- An introduction to hardware mechanisms for supporting transactional memory
  - See Larus/Rajwar book for a more complete survey
  - We'll look at:
    - A simple HTM with lazy conflict detection
    - Herlihy/Moss (1993)
  - Discuss others in reading group
Hardware support for TM

• 1. Tom Knight (1986)
  – Not really a TM scheme, Knight describes a scheme for parallelising the execution of a single thread
  – Blocks are identified by the compiler and executed in parallel assuming there are no memory carried dependencies between them
  – Hardware support is provided to detect memory dependency violations
  – This work introduces the basic ideas of using caches and the CC protocol to support TM

Larus/Rajwar book p.140
Hardware support for TM

[Diagram showing a processor connected to cache blocks and a bus, with notes on dependency cache and confirm block counter].

[Knight86]
Hardware support for TM

- **Confirm Cache**
  - A block executes to completion and then commits. Blocks are committed in the original program order
    - Any data written in the block is temporarily held in the confirm cache (not visible to other processors). This is swept and written back during commit.
    - On a processor read, priority is given to the data in the commit cache
      - The block needs to see any writes it has made

[Knight86]
Hardware support for TM

- **Dependency Cache**
  - The dependency cache holds data read from memory. Data read during a block is held in state D (Depends)
    - A memory dependency violation is detected if a bus write (made by a block that is currently committing) updates a value in a dependency cache in state D
    - This indicates that the block read the data too early and must be aborted

[Knight86]
Hardware support for TM

- Simplified state transition diagram for the dependency cache
  - In the real scheme there are also “predict” states to handle conditional execution and loops

[Knight86]
Hardware support for TM

Original Program Order

n

Load R1, X

n+1

Store R1, X

n+2

block completes....

confirm and write-back "side-effects"

confirm block counter = n

X is in state D in dependency cache

Bus write to X, new value, abort block

blocks or transactions

[Knight86]
Hardware support for TM

• 2. A simple HTM scheme
  – Lazy Conflict Detection
  – Lazy Version Management
  – Committer Wins

• Similar to Knight's scheme
  – The TCC and Bulk HTMs also take a similar approach

• Many thanks to Christos Kozyrakis at Stanford
Hardware support for TM

• **Lazy Conflict Detection**
  – We will only start looking for conflicts when we try to commit
  – We will only allow one transaction to commit at once
    • *Serialised commit*
  – To commit, we request exclusive access for all locations in our write set
    • Active transactions abort if data in their read set is invalidated
    • This is known as “**committer wins**” and guarantees forward progress
Hardware support for TM

• **Lazy Conflict Detection**
  – We will only start looking for conflicts when we try to commit
  – We will only allow one transaction to commit at once
    • **Serialised commit**
  – To commit, we request exclusive access for all locations in our write set*
    • Active transactions abort if data in their read set is invalidated
    • This is known as “committer wins” and guarantees forward progress

*In practice, many systems check against both the read set and the write set. This is due to granularity issues (we are working with cache lines not individual words) and in order to support strong isolation
Hardware support for TM

\[
\text{atomic} \{ \quad \%t1 \\
\text{write } x \\
\text{read } t \\
\text{write } y \\
\text{write } z \\
\}
\]

\[
\text{atomic} \{ \quad \%t2 \\
\text{write } z \\
\text{read } x \\
\}
\]
Hardware support for TM

- Registers are saved (checkpointed) in case we need to abort
- R and W bits in the cache track each transaction's read/write sets
  - Each transaction's write set is only visible locally
Hardware support for TM

**Diagram:**

<table>
<thead>
<tr>
<th>R</th>
<th>W</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Let's assume t1 gets to commit first
- Validate: request exclusive access to write-set lines
- Commit: reset R/W bits, turn write set into valid (dirty) data
Hardware support for TM

• BusUpgrX transactions arrive at t2's cache
  – Check: exclusive requests against its read-set
  – Abort: (if necessary) invalidate write-set, reset R/W bits, restore registers
Hardware support for TM

- 3. Herlihy/Moss (1993)
  - Coined the term “transaction memory”
  - Eager Conflict Detection
  - Lazy Version Management
Hardware support for TM

- Their scheme exploits **Eager Conflict Detection**
  - It detects possible conflicts as each transaction executes
  - This reduces the amount of computation lost by an aborted transaction
  - It may also abort transactions that could have committed
    - We don't have the luxury of knowing anything about the order in which transactions will actually commit
    - In addition, we are not committing transactions one-at-a-time (serialised commit). So have to worry about write-write conflicts too.
Hardware support for TM

- If we adopt eager (pessimistic) conflict detection we don't know the order in which T1 and T2 will commit when we are detecting conflicts
  - We have to assume the worst-case situation. In this case, that T2 will commit first and T1 must be aborted
  - If T1 actually commits first and we use lazy (optimistic) conflict detection, it is not necessary to abort either transaction
Hardware support for TM

- Similar setup to our simple HTM scheme
- But now, all caches will snoop our read/write requests
  - What happens if our request hits in another cache?
    - If we are performing a bus read and the data is only in the read set of the other transaction - no problem
    - Any other situation where we access the data set of another transaction will cause the remote cache to initiate a bus transaction to indicate that we should abort
      - This covers all situations where there is a potential conflict: two transactions accessing the same memory location where at least one operation is a write
      - We assume that the requester aborts, but there are other policies
Hardware support for TM

- Herlihy & Moss' Implementation
  - Their implementation isn't as simple as described!

- Extensions (see paper)
  - Store old value in cache
    - XCOMMIT (Old value)
    - XABORT (New value)
    - We discard one depending on the outcome, commit or abort
  - Dual caches
    - They don't use a single cache, they have a regular and transactional cache
    - Why is this advantageous?
Hardware support for TM

- **Problems**
  - Assumes transactions are short lived and have small data sets
    - Maximum transaction size bounded by size of transactional cache
    - They suggest trapping to software to support large transactions (as in the limitLESS directory protocol)
  - Contention management
    - How do we stop transactions repeatedly aborting each other?
    - As described the scheme doesn't guarantee forward progress
    - They suggest addressing this at the software level, by having aborted transactions execute exponential backoff in SW
Hardware support for TM

Other approaches to build TM systems:
- Unbounded HTMs
- Use of bloom filters (SigTM)
- Hybrid TM schemes
Retaining locks

- **Speculative Lock Elision (SLE)**
  - Rajwar and Goodman (MICRO 2001)
  - Retains lock-based programming model but exploits optimistic execution

- Another possibility is to identify critical sections (transactions) but construct a set of locks automatically by analysing the whole program
  - Pessimistic rather than optimistic concurrency
  - “Lock inference”
Software TM (STM)

• Software TM systems are important too
  – Hardware is useful in accelerating most frequent or expensive operations
• We won't cover software TM systems in detail here
  – Read Chapter 3. of Larus/Rajwar book