Lecture 4: Directory Protocols and TM

- Topics: corner cases in directory protocols, lazy TM
Handling Reads

• When the home receives a read request, it looks up memory (speculative read) and directory in parallel

• Actions taken for each directory state:
  ➢ shared or unowned: memory copy is clean, data is returned to requestor, state is changed to excl if there are no other sharers
  ➢ busy: a NACK is sent to the requestor
  ➢ exclusive: home is not the owner, request is forwarded to owner, owner sends data to requestor and home
Inner Details of Handling the Read

• The block is in exclusive state – memory may or may not have a clean copy – it is speculatively read anyway

• The directory state is set to busy-exclusive and the presence vector is updated

• In addition to forwarding the request to the owner, the memory copy is speculatively forwarded to the requestor
  - Case 1: excl-dirty: owner sends block to requestor and home, the speculatively sent data is over-written
  - Case 2: excl-clean: owner sends an ack (without data) to requestor and home, requestor waits for this ack before it moves on with speculatively sent data
Inner Details II

• Why did we send the block speculatively to the requestor if it does not save traffic or latency?
  ➢ the R10K cache controller is programmed to not respond with data if it has a block in excl-clean state
  ➢ when an excl-clean block is replaced from the cache, the directory need not be updated – hence, directory cannot rely on the owner to provide data and speculatively provides data on its own
Handling Write Requests

• The home node must invalidate all sharers and all invalidations must be acked (to the requestor), the requestor is informed of the number of invalidates to expect

• Actions taken for each state:
  ➢ shared: invalidates are sent, state is changed to excl, data and num-sharers are sent to requestor, the requestor cannot continue until it receives all acks (Note: the directory does not maintain busy state, subsequent requests will be fwned to new owner and they must be buffered until the previous write has completed)
Handling Writes II

• Actions taken for each state:
  ➢ unowned: if the request was an upgrade and not a read-exclusive, is there a problem?
  ➢ exclusive: is there a problem if the request was an upgrade? In case of a read-exclusive: directory is set to busy, speculative reply is sent to requestor, invalidate is sent to owner, owner sends data to requestor (if dirty), and a “transfer of ownership” message (no data) to home to change out of busy
  ➢ busy: the request is NACKed and the requestor must try again
Handling Write-Back

- When a dirty block is replaced, a writeback is generated and the home sends back an ack

- Can the directory state be shared when a writeback is received by the directory?

- Actions taken for each directory state:
  - exclusive: change directory state to unowned and send an ack
  - busy: a request and the writeback have crossed paths: the writeback changes directory state to shared or excl (depending on the busy state), memory is updated, and home sends data to requestor, the intervention request is dropped
Writeback Cases

This is the “normal” case. D3 sends back an Ack.
If someone else has the block in exclusive, D3 moves to busy
If Wback is received, D3 serves the requester
If we didn’t use busy state when transitioning from E:P1 to E:P2, D3 may not have known who to service (since ownership may have been passed on to P3 and P4…) (although, this problem can be solved by NACKing the Wback and having P1 buffer its “strange” intervention requests… this could lead to other corner cases… )
Writeback Cases

If Wback is from new requester, D3 sends back a NACK
Floating unresolved messages are a problem
Alternatively, can accept the Wback and put D3 in some new busy state

Conclusion: could have got rid of busy state between E:P1 \(\rightarrow\) E:P2, but
with Wback ACK/NACK and other buffering
could have kept the busy state between E:P1 \(\rightarrow\) E:P2, could
have got rid of ACK/NACK, but need one new busy state
Transactions

• Access to shared variables is encapsulated within transactions – the system gives the illusion that the transaction executes atomically – hence, the programmer need not reason about other threads that may be running in parallel with the transaction.

Conventional model:

```plaintext
... 
lock(L1);
    access shared vars
unlock(L1);
... 
```

TM model:

```plaintext
... 
trans_begin();
    access shared vars
trans_end();
... 
```
Transactions

• Transactional semantics:
  ▪ when a transaction executes, it is as if the rest of the system is suspended and the transaction is in isolation
  ▪ the reads and writes of a transaction happen as if they are all a single atomic operation
  ▪ if the above conditions are not met, the transaction fails to commit (abort) and tries again

```
transaction begin
  read shared variables
  arithmetic
  write shared variables
transaction end
```
Why are Transactions Better?

• High performance with little programming effort
  ➢ Transactions proceed in parallel most of the time if the probability of conflict is low (programmers need not precisely identify such conflicts and find work-arounds with say fine-grained locks)
  ➢ No resources being acquired on transaction start; lesser fear of deadlocks in code
  ➢ Composability
Example

Producer-consumer relationships – producers place tasks at the tail of a work-queue and consumers pull tasks out of the head

Enqueue
transaction begin
  if (tail == NULL)
    update head and tail
  else
    update tail
transaction end

Dequeue
transaction begin
  if (head->next == NULL)
    update head and tail
  else
    update head
transaction end

With locks, neither thread can proceed in parallel since head/tail may be updated – with transactions, enqueue and dequeue can proceed in parallel – transactions will be aborted only if the queue is nearly empty
Example

Is it possible to have a transactional program that deadlocks, but the program does not deadlock when using locks?

```c
flagA = flagB = false;
thr-1
lock(L1)
    while (!flagA) {};
    flagB = true;
    *
unlock(L1)

thr-2
lock(L2)
    flagA = true;
    while (!flagB) {};
    *
unlock(L2)
```

• Somewhat contrived
• The code implements a barrier before getting to *
• Note that we are using different lock variables
Atomicity

• Blindly replacing locks-unlocks with tr-begin-end may occasionally result in unexpected behavior

• The primary difference is that:
  ▪ transactions provide atomicity with every other transaction
  ▪ locks provide atomicity with every other code segment that locks the same variable

• Hence, transactions provide a “stronger” notion of atomicity – not necessarily worse for performance or correctness, but certainly better for programming ease
Other Constructs

- Retry: abandon transaction and start again
- OrElse: Execute the other transaction if one aborts
- Weak isolation: transactional semantics enforced only between transactions
- Strong isolation: transactional semantics enforced between transactions and non-transactional code
Useful Rules of Thumb

• Transactions are often short – more than 95% of them will fit in cache

• Transactions often commit successfully – less than 10% are aborted

• 99.9% of transactions don’t perform I/O

• Transaction nesting is not common

• Amdahl’s Law again: optimize the common case!
  → fast commits, can have slightly slow aborts, can have slightly slow overflow mechanisms
Design Space

• Data Versioning
  ▪ Eager: based on an undo log
  ▪ Lazy: based on a write buffer
    Typically, versioning is done in cache;
    The above two are variants that handle overflow

• Conflict Detection
  ▪ Optimistic detection: check for conflicts at commit time
    (proceed optimistically thru transaction)
  ▪ Pessimistic detection: every read/write checks for
    conflicts (so you can abort quickly)
Basic Implementation – Lazy, Lazy

• Writes can be cached (can’t be written to memory) – if the block needs to be evicted, flag an overflow (abort transaction for now) – on an abort, invalidate the written cache lines

• Keep track of read-set and write-set (bits in the cache) for each transaction

• When another transaction commits, compare its write set with your own read set – a match causes an abort

• At transaction end, express intent to commit, broadcast write-set (transactions can commit in parallel if their write-sets do not intersect)
Lazy Overview

Topics:
• Commit order
• Overheads
• Wback, WAW
• Overflow
• Parallel Commit
• Hiding Delay
• I/O
• Deadlock, Livelock, Starvation
“Lazy” Implementation (Partially Based on TCC)

• An implementation for a small-scale multiprocessor with a snooping-based protocol

• Lazy versioning and lazy conflict detection

• Does not allow transactions to commit in parallel
Handling Reads/Writes

• When a transaction issues a read, fetch the block in read-only mode (if not already in cache) and set the rd-bit for that cache line

• When a transaction issues a write, fetch that block in read-only mode (if not already in cache), set the wr-bit for that cache line and make changes in cache

• If a line with wr-bit set is evicted, the transaction must be aborted (or must rely on some software mechanism to handle saving overflowed data) (or must acquire commit permissions)
Commit Process

• When a transaction reaches its end, it must now make its writes permanent

• A central arbiter is contacted (easy on a bus-based system), the winning transaction holds on to the bus until all written cache line addresses are broadcasted (this is the commit) (need not do a writeback until the line is evicted or written again – must simply invalidate other readers of these lines)

• When another transaction (that has not yet begun to commit) sees an invalidation for a line in its rd-set, it realizes its lack of atomicity and aborts (clears its rd- and wr-bits and re-starts)
Miscellaneous Properties

- While a transaction is committing, other transactions can continue to issue read requests

- Writeback after commit can be deferred until the next write to that block

- If we’re tracking info at block granularity, (for various reasons), a conflict between write-sets must force an abort
Summary of Properties

• Lazy versioning: changes are made locally – the “master copy” is updated only at the end of the transaction

• Lazy conflict detection: we are checking for conflicts only when one of the transactions reaches its end

• Aborts are quick (must just clear bits in cache, flush pipeline and reinstate a register checkpoint)

• Commit is slow (must check for conflicts, all the coherence operations for writes are deferred until transaction end)

• No fear of deadlock/livelock – the first transaction to acquire the bus will commit successfully

• Starvation is possible – need additional mechanisms
Title

• Bullet