Lecture 10: Transactional Memory

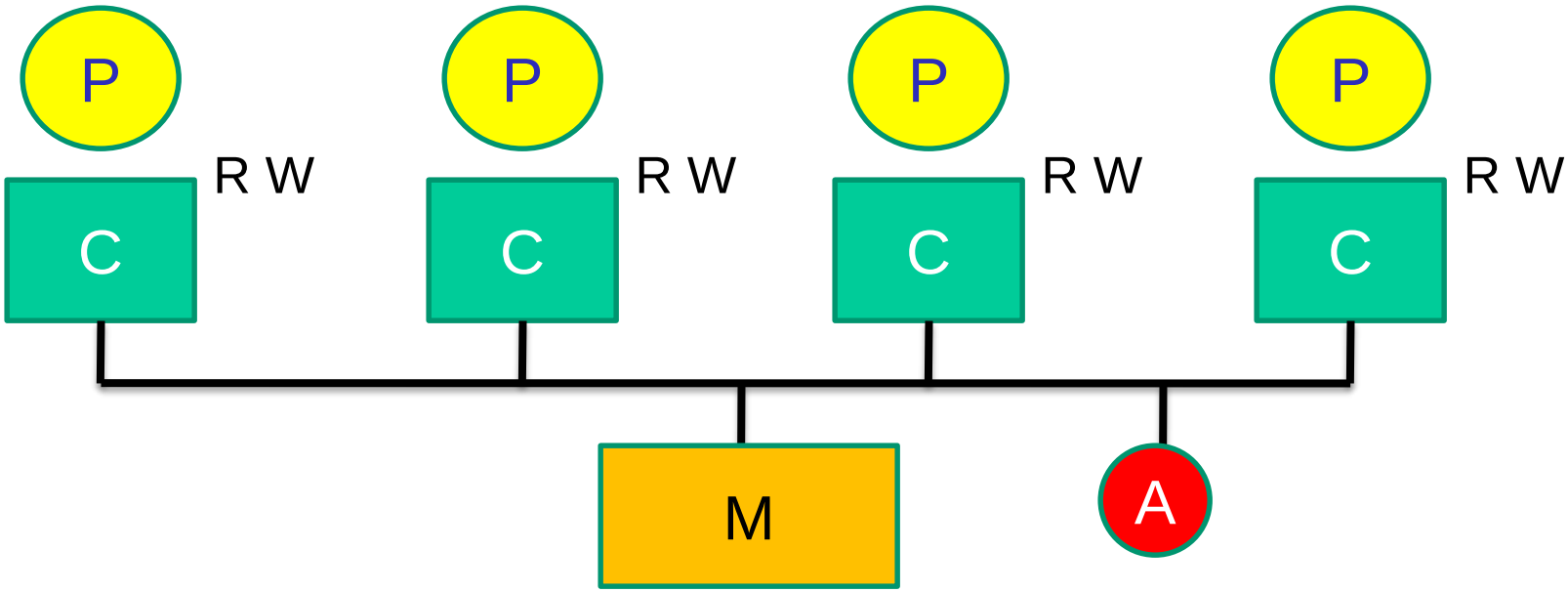
- Topics: lazy and eager TM implementations, TM pathologies

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
- Signatures



“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 broadcast (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
- Bloom filter signatures can be used to track blocks that have overflowed out of cache
- If we're tracking info at block granularity, (for various reasons), a conflict between write-sets must force an abort

Summary of Properties

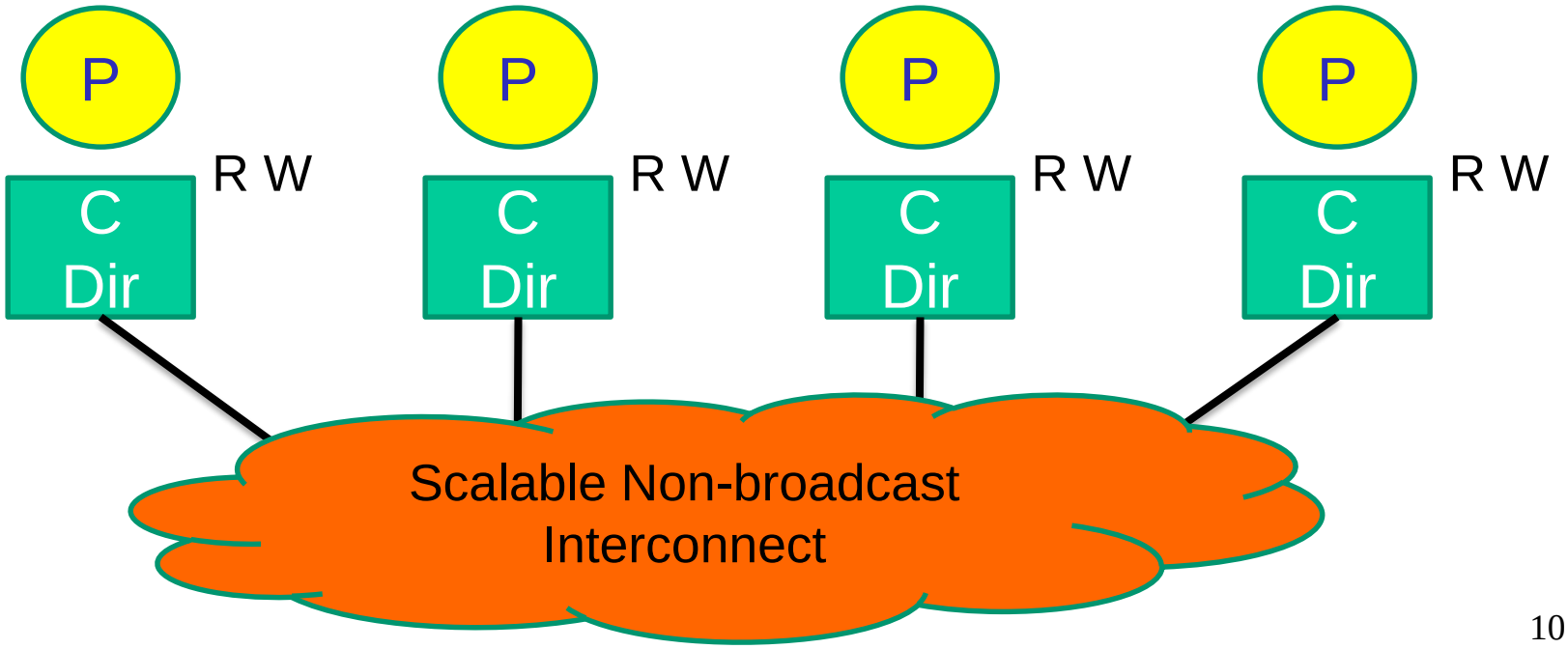
- Lazy versioning: changes are made locally – the “master copy” is updated only at the end of the transaction; on an overflow, the new version is saved in a log
- Lazy conflict detection: we are checking for conflicts only when one of the transactions reaches its end
- Aborts are quick (must just clear 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

Parallel Commits

- Each memory node has a token. Two transactions can commit in parallel if they deal with different data blocks, i.e., they need different tokens. Tokens are acquired in increasing order. (Pugsley et al., PACT'08)

“Eager” Overview

Topics:
- Logs
- Log optimization
- Conflict examples
- Handling deadlocks
- Sticky scenarios
- Aborts/commits/parallelism



“Eager” Implementation (Based Primarily on LogTM)

- A write is made permanent immediately (we do not wait until the end of the transaction)

- Can’t lose the old value (in case this transaction is aborted) – hence, before the write, we copy the old value into a log (the log is some space in virtual memory -- the log itself may be in cache, so not too expensive)
  This is eager versioning

Versioning

- Every overflowed write first requires a read and a write to log the old value – the log is maintained in virtual memory and will likely be found in cache
- Aborts are uncommon – typically only when the contention manager kicks in on a potential deadlock; the logs are walked through in reverse order
- If a block is already marked as being logged (wr-set), the next write by that transaction can avoid the re-log
- Log writes can be placed in a write buffer to reduce contention for L1 cache ports

Conflict Detection and Resolution

- Since Transaction-A's writes are made permanent rightaway, it is possible that another Transaction-B's rd/wr miss is re-directed to Tr-A

- At this point, we detect a conflict (neither transaction has reached its end, hence, *eager conflict detection)*: two transactions handling the same cache line and at least one of them does a write

- One solution: requester stalls: Tr-A sends a NACK to Tr-B; Tr-B waits and re-tries again; hopefully, Tr-A has committed and can hand off the latest cache line to B → neither transaction needs to abort

Deadlocks

- Can lead to deadlocks: each transaction is waiting for the other to finish

- Need a separate (hw/sw) contention manager to detect such deadlocks and force one of them to abort

Tr-A	Tr-B
write X	write Y
…	…
read Y	read X

- Alternatively, every transaction maintains an “age” and a young transaction aborts and re-starts if it is keeping an older transaction waiting and itself receives a nack from an older transaction

Block Replacement

- If a block in a transaction's rd/wr-set is evicted, the data is written back to memory if necessary, but the directory continues to maintain a "sticky" pointer to that node (subsequent requests have to confirm that the transaction has committed before proceeding)

- The sticky pointers are lazily removed over time (commits continue to be fast); if a transaction receives a request for a block that is not in its cache and if the transaction has not overflowed, then we know that the sticky pointer can be removed; can also maintain signatures to track evicted lines

Paper on TM Pathologies (ISCA'08)

- LL: lazy versioning, lazy conflict detection, committing transaction wins conflicts
- EL: lazy versioning, eager conflict detection, requester succeeds and others abort
- EE: eager versioning, eager conflict detection, requester stalls

Pathology 1: Friendly Fire

- Two conflicting transactions that keep aborting each other
- Can do exponential back-off to handle livelock
- Fixable by doing requester stalls?
- Fixable by only letting the older transaction win

- VM: any
- CD: eager
- CR: requester wins

Pathology 2: Starving Writer

- A writer has to wait for the reader to finish – but if more readers keep showing up, the writer is starved (note that the directory allows new readers to proceed by just adding them to the list of sharers)

- VM: any
- CD: eager
- CR: requester stalls

Can allow requester wins for a potential starved writer

Pathology 3: Serialized Commit

- If there's a single commit token, transaction commit is serialized

- VM: lazy
- CD: lazy
- CR: any

- There are ways to alleviate this problem

Pathology 4: Futile Stall

- A transaction is stalling on another transaction that ultimately aborts and takes a while to reinstate old values
  -- no good workaround

- VM: any
- CD: eager
- CR: requester stalls

Pathology 5: Starving Elder

- Small successful transactions can keep aborting a large transaction

- The large transaction can eventually grab the token and not release it until after it commits

- VM: lazy
- CD: lazy
- CR: committer wins

Pathology 6: Restart Convoy

- A number of similar (conflicting) transactions execute together – one wins, the others all abort – shortly, these transactions all return and repeat the process

  Can do exponential back-off to reduce wasted work

- VM: lazy
- CD: lazy
- CR: committer wins

Pathology 7: Dueling Upgrades

- If two transactions both read the same object and then both decide to write it, a deadlock is created

- VM: eager
- CD: eager
- CR: requester stalls

- Exacerbated by the Futile Stall pathology

- Solution?

Four Extensions

- Predictor: predict if the read will soon be followed by a write and acquire write permissions aggressively
- Hybrid: if a transaction believes it is a Starving Writer, it can force other readers to abort; for everything else, use requester stalls
- Timestamp: In the EL case, requester wins only if it is the older transaction (handles Friendly Fire pathology)
- Backoff: in the LL case, aborting transactions invoke exponential back-off to prevent convoy formation

Title

- Bullet