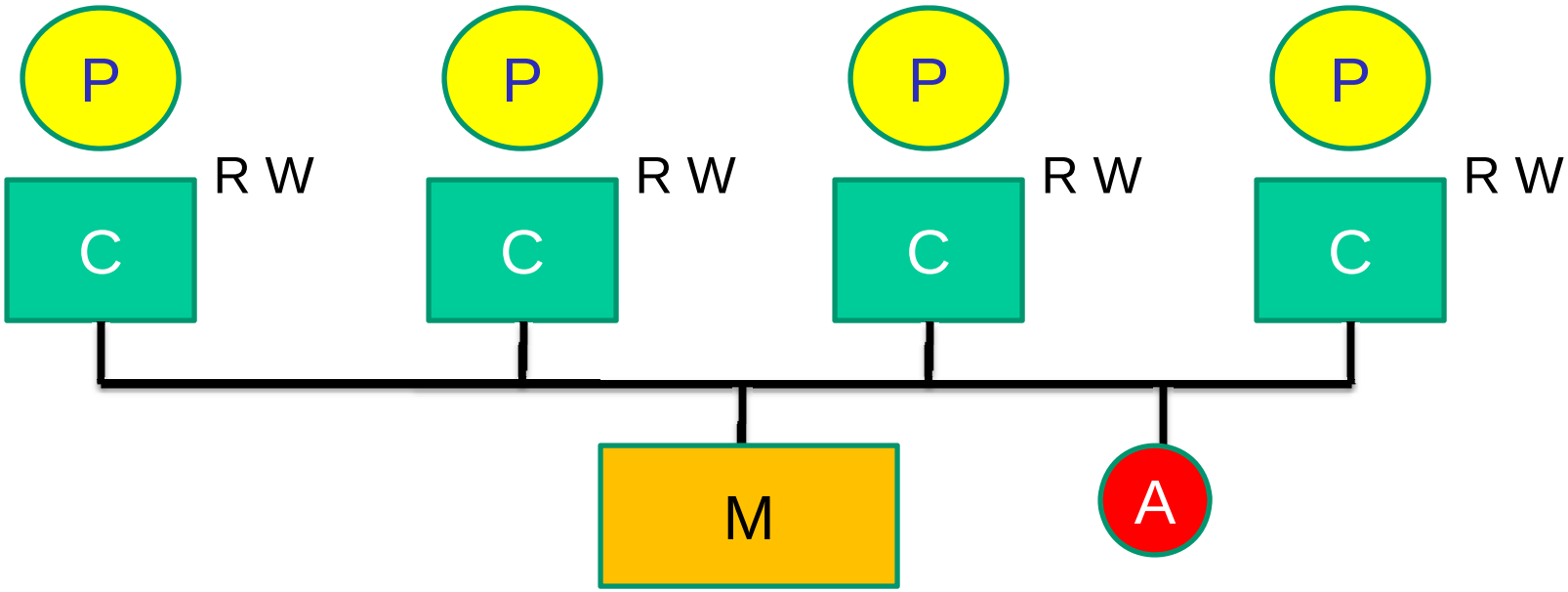
Lecture 7: Lazy & Eager Transactional Memory

- Topics: details of “lazy” TM, scalable lazy TM, implementation details of eager TM

Lazy Overview

Topics:
- Commit order
- Overheads
- Wback, WAR, WAW, RAW
- 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

TCC Features

- All transactions all the time (the code only defines transaction boundaries): helps get rid of the baseline coherence protocol
- When committing, a transaction must acquire a central token – when I/O, syscall, buffer overflow is encountered, the transaction acquires the token and starts commit
- Each cache line maintains a set of “renamed bits” – this indicates the set of words written by this transaction – reading these words is not a violation and the read-bit is not set

TCC Features

- Lines evicted from the cache are stored in a write buffer; overflow of write buffer leads to acquiring the commit token
- Less tolerant of commit delay, but there is a high degree of “coherence-level parallelism”
- To hide the cost of commit delays, it is suggested that a core move on to the next transaction in the meantime – this requires “double buffering” to distinguish between data handled by each transaction
- An ordering can be imposed upon transactions – useful for speculative parallelization of a sequential program

Parallel Commits

- Writes cannot be rolled back – hence, before allowing two transactions to commit in parallel, we must ensure that they do not conflict with each other
- One possible implementation: the central arbiter can collect signatures from each committing transaction (a compressed representation of all touched addresses)
- Arbiter does not grant commit permissions if it detects a possible conflict with the rd-wr-sets of transactions that are in the process of committing
- The “lazy” design can also work with directory protocols

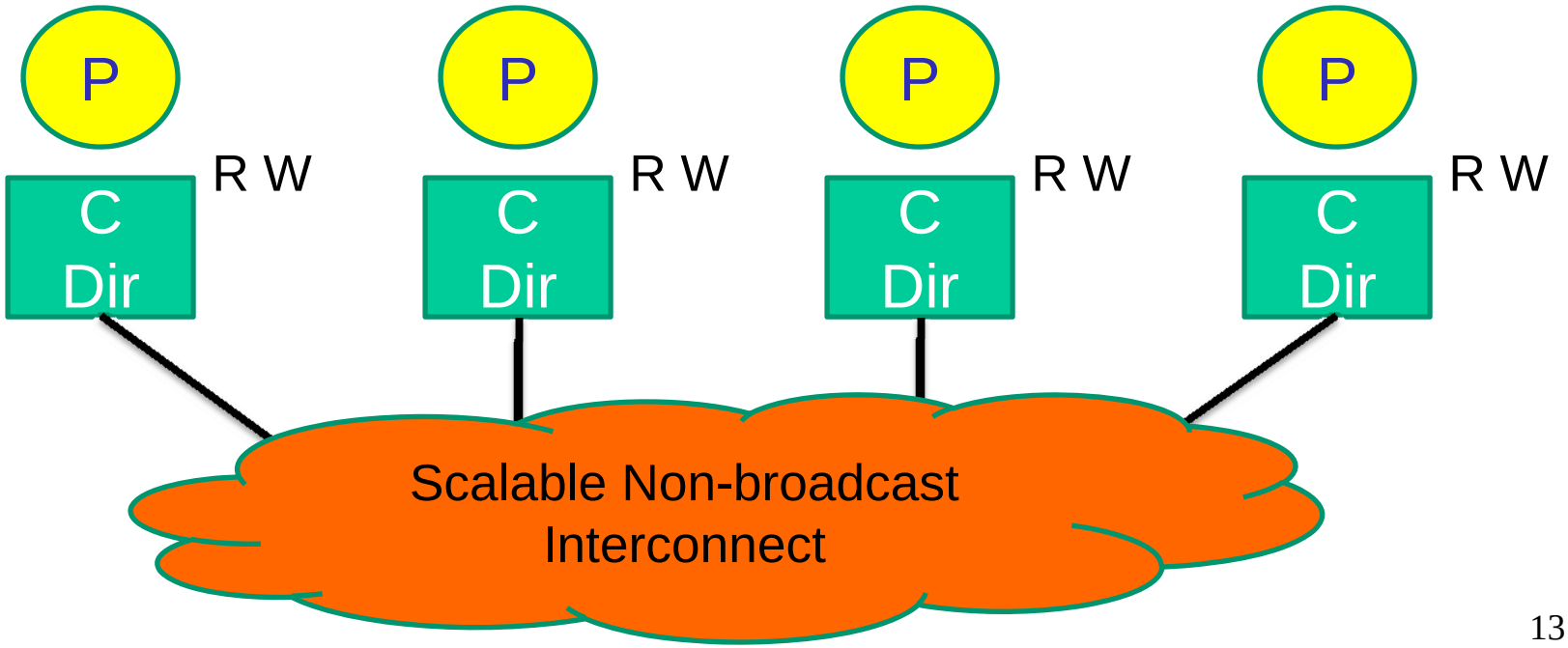
Scalable Algorithm – Lazy Implementation

- Data is distributed across several nodes/directories
- Each node has a token
- For a transaction to commit, it must first acquire all tokens corresponding to the data in its read and write set – this guarantees that an invalidation will not be received while this transaction commits
- After performing the writes, the tokens are released
- Tokens must be acquired in numerically ascending order for deadlock avoidance – can also allow older transactions to steal from younger transactions

Example

“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)

Title

- Bullet