Lecture 8: Transactional Memory – TCC

- Topics: “lazy” implementation (TCC)

Other Issues

- Nesting: when one transaction calls another
  - flat nesting: collapse all nested transactions into one large transaction
  - closed nesting: inner transaction's rd-wr set are included in outer transaction's rd-wr set on inner commit; on an inner conflict, only the inner transaction is re-started
  - open nesting: on inner commit, its writes are committed and not merged with outer transaction's commit set
- What if a transaction performs I/O? (buffering can help)

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!

Design Space

- Data Versioning
  - Eager: based on an undo log
  - Lazy: based on a write buffer

- 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)

Detecting Conflicts – Overview

- 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” 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)

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

Title

- Bullet