Lecture 6: Lazy Transactional Memory

- Topics: TM semantics and implementation details of "lazy" TM

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:

```
…
lock(L1);
  access shared vars
unlock(L1);
…
```

TM model:

```
…
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?

```
flagA = flagB = false;
   thr-1                                  thr-2
lock(L1)                                lock(L2)
   while (!flagA) {};                      flagA = true;
   flagB = true;                           while (!flagB) {};
      *                                       *
unlock(L1)                              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 beween transactions and non-transactional code

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!
  → 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

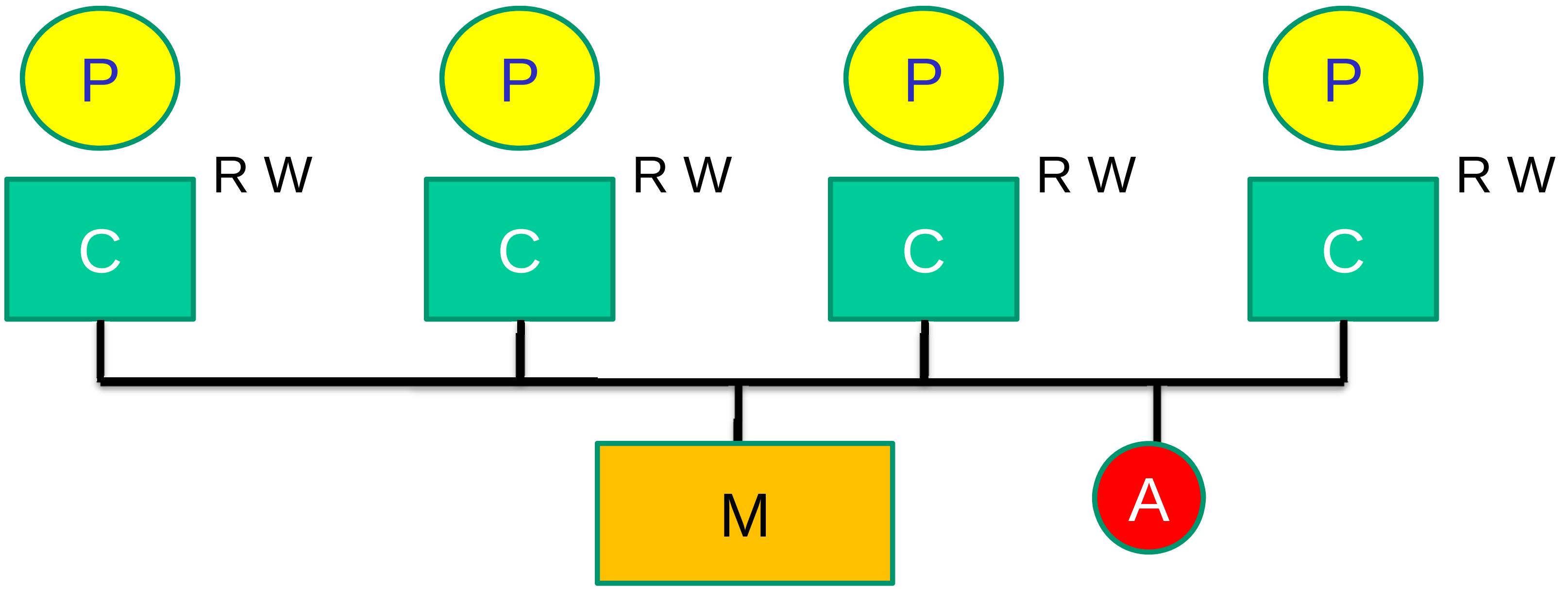
- Keep track of read-set and write-set (bits in the cache) for each transaction

- 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, 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

Title

- Bullet