Lecture 21: Transactional Memory

- Topics: Hardware TM basics, different implementations

Transactions

- New paradigm to simplify programming
  - instead of lock-unlock, use transaction begin-end
  - locks are blocking, transactions execute speculatively in the hope that there will be no conflicts
- Can yield better performance; Eliminates deadlocks
- Programmer can freely encapsulate code sections within transactions and not worry about the impact on performance and correctness (for the most part)
- Programmer specifies the code sections they'd like to see execute atomically – the hardware takes care of the rest (provides illusion of atomicity)

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
```

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

Hash table implementation

```
transaction begin
    index = hash(key);
    head = bucket[index];
    traverse linked list until key matches
    perform operations
transaction end
```

Most operations will likely not conflict → transactions proceed in parallel

Coarse-grain lock → serialize all operations

Fine-grained locks (one for each bucket) → more complexity, more storage, concurrent reads not allowed, concurrent writes to different elements not allowed

TM Implementation

- Caches track read-sets and write-sets
- Writes are made visible only at the end of the transaction
- At transaction commit, make your writes visible; others may abort

Detecting Conflicts – Basic Implementation

- 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)

Summary of TM Benefits

- As easy to program as coarse-grain locks
- Performance similar to fine-grain locks
- Avoids deadlock

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 (reduces work during commit)

“Lazy” Implementation

- 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

“Lazy” Implementation

- 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)

“Lazy” Implementation

- 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 – must simply invalidate other readers of these cache 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)

“Lazy” Implementation

- 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

“Lazy” Implementation – 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

“Eager” Implementation

- A write is made permanent immediately (we do not wait until the end of the transaction)
- This means that if some other transaction attempts a read, the latest value is returned and the memory may also be updated with this latest value
- 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

“Eager” Implementation

- 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

“Eager” Implementation

- 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

“Eager” Implementation

- Note that if Tr-B is doing a write, it may be forced to stall because Tr-A may have done a read and does not want to invalidate its cache line just yet

- If new reading transactions keep emerging, Tr-B may be starved – again, need other sw/hw mechanisms to handle starvation

- Since logs are stored in virtual memory, there is no cache overflow problem and transactions can be large

- Commits are inexpensive (no additional step required); Aborts are expensive, but rare (must reinstate data from logs)

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?
- What if a transaction overflows out of cache?

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!

Discussion

- “Eager” optimizes the common case and does not waste energy when there’s a potential conflict
- TM implementations require relatively low hardware support
- Multiple commercial examples: Sun Rock, AMD ASF, IBM BG/Q, Intel Haswell

Title

- Bullet