Lecture 7: Transactional Memory Intro

- Topics: introduction to transactional memory,
  “lazy” implementation

Transactions

- New paradigm to simplify programming
  - instead of lock-unlock, use transaction begin-end

- Can yield better performance; Eliminates deadlocks

- Programmer can freely encapsulate code sections within transactions and not worry about the impact on performance and correctness

- 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
```

Applications

- A transaction executes speculatively in the hope that there will be no conflicts

- Can replace a lock-unlock pair with a transaction begin-end
  - the lock is blocking, the transaction is not
  - programmers can conservatively introduce transactions without worsening performance

lock (lock1)	transaction begin
read A	read A
operations	operations
write A	write A
unlock (lock1)	transaction end

Example 1

```
lock (lock1)
   counter = counter + 1;
unlock (lock1)
```

```
transaction begin
   counter = counter + 1;
transaction end
```

Is the transactional code any better?

Example 2

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 3

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

Example 4

Is it possible to have a transactional program that deadlocks?

Example 4

Is it possible to have a transactional program that deadlocks?

```
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

Summary of TM Benefits

- As easy to program as coarse-grain locks
- Performance similar to fine-grain locks
- Speculative parallelization
- Avoids deadlock
- Resilient to faults

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)

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)

Design Issues and Challenges

- Nested transactions
  - Closed nesting: nested transaction's read/write set are included in parent's read/write set on inner commit; on inner conflict, only nested transaction is re-started; easier for programmer
  - Open nesting: on inner commit, writes are committed and not merged with outer read/write set

- I/O – buffering can help

- Interaction with other non-TM applications (OS)

- Large transactions that cause overflows (less than 1% of all transactions are large)

- Low overheads for rollback and commit

Title

- Bullet