Lecture 23: Transactional Memory

 Topics: consistency model recap, introduction to transactional memory

Example Programs

Initially, A = B = 0**P1** P2 A = 1B = 1if (B == 0)if (A == 0)critical section critical section Initially, A = B = 0**P1 P2 P3** A = 1if (A == 1)B = 1 if (B == 1)register = A

P1 P2 Data = 2000 while (Head == 0) Head = 1 { } ... = Data

Sequential Consistency

P1	P2
Instr-a	Instr-A
Instr-b	Instr-B
Instr-c	Instr-C
Instr-d	Instr-D

We assume:

- Within a program, program order is preserved
- Each instruction executes atomically
- Instructions from different threads can be interleaved arbitrarily

Valid executions: abAcBCDdeE... or ABCDEFabGc... or abcAdBe... or aAbBcCdDeE... or

- A multiprocessor is sequentially consistent if the result of the execution is achieveable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion
- Can implement sequential consistency by requiring the following: program order, write serialization, everyone has seen an update before a value is read – very intuitive for the programmer, but extremely slow
- This is very slow... alternatives:
 - Add optimizations to the hardware
 - Offer a relaxed memory consistency model and fences

Fences

P1 { Region of code with no races }

Fence Acquire_lock Fence

{ Racy code }

Fence Release_lock Fence

P2 Region of code with no races Fence Acquire_lock Fence Racy code Fence Release_lock Fence

Transactions

- New paradigm to simplify programming
 - instead of lock-unlock, use transaction begin-end
 - locks are blocking, transactions execute speculative 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

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

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

No apparent advantage to using transactions (apart from fault resiliency)

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 tailDequeue
transaction begin
if (head->next == NULL)
update head and tail
else
update tail
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

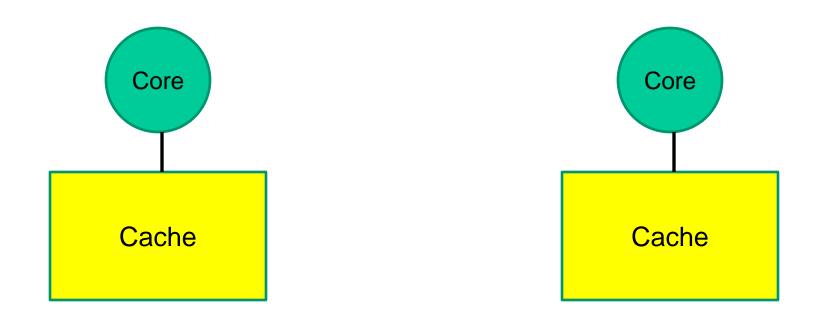
perform operations

transaction end
```

Most operations will likely not conflict \rightarrow 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
- Speculative parallelization
- Avoids deadlock
- Resilient to faults

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)

- Transactions can be viewed as an extension of LL-SC
- LL-SC ensures that the read-modify-write for a single variable is atomic; a transaction ensures atomicity for all variables accessed between trans-begin and trans-end

Vers-1		Vers-2		Vers-3		
Ш	a	II	а	tra	trans-begin	
ld	b	- 11	b	ld	a	
st	b	SC	b	ld	b	
SC	a	SC	а	st	b	
				st	а	
				trans-end		



Bullet