

Lecture: Consistency Models, TM

- Topics: consistency models, TM intro (Section 5.6)

Coherence Vs. Consistency

- Recall that coherence guarantees (i) that a write will eventually be seen by other processors, and (ii) write serialization (all processors see writes to the same location in the same order)
- The consistency model defines the ordering of writes and reads to different memory locations – the hardware guarantees a certain consistency model and the programmer attempts to write correct programs with those assumptions

Example Programs

Initially, $A = B = 0$

P1

$A = 1$

if ($B == 0$)

critical section

P2

$B = 1$

if ($A == 0$)

critical section

Initially, $Head = Data = 0$

P1

$Data = 2000$

$Head = 1$

P2

while ($Head == 0$)

{ }

... = $Data$

Initially, $A = B = 0$

P1

$A = 1$

P2

if ($A == 1$)

$B = 1$

P3

if ($B == 1$)

register = A

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

Problem 1

- What are possible outputs for the program below?

Assume $x=y=0$ at the start of the program

Thread 1

$x = 10$

$y = x+y$

Print y

Thread 2

$y=20$

$x = y+x$

Problem 1

- What are possible outputs for the program below?

Assume $x=y=0$ at the start of the program

Thread 1		Thread 2	
A	$x = 10$	a	$y=20$
B	$y = x+y$	b	$x = y+x$
C	Print y		

Possible scenarios: 5 choose 2 = 10

ABCab	ABaCb	ABabC	AaBCb	AaBbC
10	20	20	30	30
AabBC	aABCb	aABbC	aAbBC	abABC
50	30	30	50	30

Sequential Consistency

- Programmers assume SC; makes it much easier to reason about program behavior
- Hardware innovations can disrupt the SC model
- For example, if we assume write buffers, or out-of-order execution, or if we drop ACKS in the coherence protocol, the previous programs yield unexpected outputs

Consistency Example - I

- An ooo core will see no dependence between instructions dealing with A and instructions dealing with B; those operations can therefore be re-ordered; this is fine for a single thread, but not for multiple threads

```
Initially A = B = 0
P1          P2
A ← 1      B ← 1
...
if (B == 0)  if (A == 0)
  Crit.Section  Crit.Section
```

The consistency model lets the programmer know what assumptions they can make about the hardware's reordering capabilities

Consistency Example - 2

Initially, $A = B = 0$

P1
 $A = 1$

P2
if ($A == 1$)
 $B = 1$

P3
if ($B == 1$)
 register = A

If a coherence invalidation didn't require ACKs, we can't confirm that everyone has seen the value of A.

Sequential Consistency

- A multiprocessor is sequentially consistent if the result of the execution is achievable 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

Relaxed Consistency Models

- We want an intuitive programming model (such as sequential consistency) and we want high performance
- We care about data races and re-ordering constraints for some parts of the program and not for others – hence, we will relax some of the constraints for sequential consistency for most of the program, but enforce them for specific portions of the code
- Fence instructions are special instructions that require all previous memory accesses to complete before proceeding (sequential consistency)

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

Relaxing Constraints

- Sequential consistency constraints can be relaxed in the following ways (allowing higher performance):
 - within a processor, a read can complete before an earlier write to a different memory location completes (this was made possible in the write buffer example and is of course, not a sequentially consistent model)
 - within a processor, a write can complete before an earlier write to a different memory location completes
 - within a processor, a read or write can complete before an earlier read to a different memory location completes
 - a processor can read the value written by another processor before all processors have seen the invalidate
 - a processor can read its own write before the write is visible to other processors

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
- 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)

“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

Title

- Bullet