Lecture 20: 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                        P2
A = 1                     B = 1
if (B == 0)                if (A == 0)
   critical section           critical section
```

Initially, A = B = 0

```
 P1              P2              P3
A = 1
             if (A == 1)
               B = 1
                              if (B == 1)
                                register = A
```

Initially, Head = Data = 0

```
 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 …..

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            P2                  P3
A = 1
            if (A == 1)
              B = 1
                              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 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

Example Programs

Initially, A = B = 0

```
  P1                      P2
A = 1                   B = 1
if (B == 0)             if (A == 0)
   critical section        critical section
```

Initially, A = B = 0

```
 P1          P2           P3
A = 1
          if (A == 1)
            B = 1
                        if (B == 1)
                          register = A
```

Initially, Head = Data = 0

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

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                          P2
{                           {
  Region of code              Region of code
  with no races               with no races
}                           }

Fence                       Fence
Acquire_lock                Acquire_lock
Fence                       Fence

{                           {
   Racy code                   Racy code
}                           }

Fence                       Fence
Release_lock                Release_lock
Fence                       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 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
```

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

Title

- Bullet