Memory consistency models

Songhao Zhou
Introduction

• What can the programmer assume about the servicing of memory operations?

Initially all pointers = null, all integers = 0.

P1

while (there are more tasks) {
    Task = GetFromFreeList();
    Task → Data = ...
    insert Task in task queue
}

Head = head of task queue;

P2, P3, …, Pn

while (MyTask == null) {
    Begin Critical Section
    if (Head != null) {
        MyTask = Head;
        Head = Head → Next;
    }
    End Critical Section
}

... = MyTask → Data;

Figure 1: What value can a read return?
Expected Behavior: the value read from the data field within a dequeued record should be the same as that written by P1 in that record.

However, in many commercial shared memory systems, it is possible for processors to observe the old value of the data field. Programmers need a precise notion of how memory behaves with respect to read and write operations from multiple processors in order to write correct and efficient shared memory programs.
Memory consistency model:

- A formal specification of how the memory system on a shared memory multiprocessor will appear to the programmer
- Eliminate the gap between the behavior expected by the programmer and the actual behavior supported by a system
Memory model for multiprocessors?

- Intuitively, a read of a memory location should return the value of its “last” write
- Natural for uniprocessors
- Not obvious what this means for multiprocessors with concurrent operations
- Idea: require that all memory operations appear to execute one at a time, and the operations of a single processor appear to execute in the order described by that processor’s program
Sequential consistency (SC)

Result of an execution appears as if

- All operations appear as if executed in some **sequential order**
- Memory operations of each thread in **program order**

[Lamport 1979]
Two aspect of sequential consistency

1. Program Order
2. Atomicity

Citation: http://preshing.com/20130618/atomic-vs-non-atomic-operations/
Why does program order matter?

What do we expect from this code?

Initially Flag1 = Flag2 = 0

P1
Flag1 = 1
if (Flag2 == 0)  
critical section

P2
Flag2 = 1
if (Flag1 == 0)  
critical section
Why does program order matter?

Execution:

P1
(Operation, Location, Value)
Write, Flag1, 1
Read, Flag2, 0

P2
(Operation, Location, Value)
Write, Flag2, 1
Read, Flag1, 1
Why does program order matter?

Both threads could enter critical section if
● if the hardware allows a thread’s read to complete before a prior write completes
● if the compiler reorders the thread’s read and write
Why does atomicity matter?

What do we expect from this code?

Initially $A = B = 0$

P1 P2 P3

$A = 1$

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

if ($B == 1$)
$\text{register1} = A = 1$
Why does atomicity matter?

What actually could happen...

- Can happen if read returns new value before all copies see it
- We have a cycle here
- Leads to unintuitive behavior
Why does atomicity matter?

- Atomicity allows us to assume the effect of P1’s write is seen by the entire system at the same time.
- With atomicity, P3 would see the effect of P1’s write and must return the value 1 for its read of A.
Atomic vs non-atomic operations

An operation acting on shared memory is atomic if:

- it completes in a single step relative to other threads
- when an atomic store is performed on a shared variable, no other thread can observe the modification half-complete
- when an atomic load is performed on a shared variable, it reads the entire value as it appeared at a single moment in time

Non-atomic loads and stores do not make those guarantees
Implementing sequential consistency on modern hardware is non-obvious

3 canonical hardware optimizations (without caching):

- Write buffers with bypassing capability
- Overlapping write operations
- Non-blocking read operations
Write buffer with read bypassing

- Write buffer
  - On a write, processor inserts a value into a write buffer
  - Proceeds without waiting for the write to complete
  - Why? Hide the latency of write operations
- Read bypassing
  - Subsequent reads may bypass any previous writes in a write buffer for faster completion
Write buffer with read bypassing

What do we expect from this code?

**P1**
- Flag1 = 1
- if (Flag2 == 0)
  - critical section

**P2**
- Flag2 = 1
- if (Flag1 == 0)
  - critical section
Write buffer with read bypassing

- Can happen if read early from write buffer
Overlapping write operations

- Multiple write operations issued by the same processor may be simultaneously serviced by different memory modules.
Overlapping write operations

What do we expect from this code?

P1
- Data = 2000
- Head = 1

P2
- while (Head == 0) {};
- ... = Data
- Assume Data and Head reside in different memory modules
- Write to Head may be injected into the network before the write to Data has reached its memory module
  - The two writes could complete out of program order
- Possible for another processor to observe the new value of Head and the old value of Data
Non-blocking reads

- Many processors have the capability to proceed past a read operation by using techniques such as non-blocking (lockup-free) caches, speculative execution, and dynamic scheduling
Non-blocking reads

What do we expect from this code?

P1
Write Head t3
Write Data t2

P2
Read Head t4
Read Data t1

Head: 0
Memory

Data: 0

P1
Data = 2000
Head = 1

P2
while (Head == 0) {;}
... = Data
- Assume P1 ensures that its writes arrive at their respective memory modules in program order
- A problem if P2 is allowed to issue its read operations in an overlapped fashion
- Possible for the read of Data to arrive at its memory module before the write from P1 while the read of Head reaches its memory module after the write from P1
- Leads to a non-sequentially-consistent outcome

```
P1
Data = 2000
Head = 1

while (Head == 0) {};
...
= Data

P2
```
Architecture with caches

- Use Cache to store data
- A mechanism needed to propagate a write to other copies ⇒ Cache coherence protocol
Three additional issues with shared data

1. The presence of multiple copies requires a cache coherence protocol to propagate a newly written value to all cached copies of the modified location.

2. Detecting when a write is complete (to preserve program order between a write and its following operations) requires more transactions

3. Propagating changes to multiple copies is inherently non-atomic
   ○ challenging to preserve the illusion of atomicity for writes with respect to other operations
How to propagate writes?

How to propagate writes?
Invalidate -- Remove old copies from other caches
Update -- Update old copies in other caches to new values
Consider a system with caches

What do we expect from this code?

- Assume that processor P2 initially has Data in its cache
- Suppose P1 proceeds with its write to Head after its previous write to Data reaches its target memory but before its value has been propagated to P2
● Possible for P2 to read the new value of Head and still return the old value of Data from its cache
  ○ violates sequential consistency
  ○ can be avoided if P1 waits for P2’s cache copy of Data to be updated or invalidated before proceeding with the write to Head
● Requires a mechanism to acknowledge the receipt of invalidation or update messages by the target cache and collected
Serialization of writes

What do we expect from this code?

Initially $A = B = C = 0$

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A = 1$</td>
<td>$A = 2$</td>
<td>while ($B \neq 1$) {;}</td>
<td>while ($B \neq 1$) {;}</td>
</tr>
<tr>
<td>$B = 1$</td>
<td>$C = 1$</td>
<td>while ($C \neq 1$) {;}</td>
<td>while ($C \neq 1$) {;}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>register1 = A $= 2$</td>
<td>register2 = A $= 2$</td>
</tr>
</tbody>
</table>
P3 and P4 can see different values for A if updates of A reach P3 and P4 in different order.

Coherence protocol must serialize writes to same location
- multiple writes to a location should be seen in same order by all

Delay an update or invalidate from being sent out until any updates or invalidates that have been issued on behalf of a previous write to the same location are acknowledged.
Compilers

- Compiler-generated reordering of loads and stores of shared values can lead to violations of sequential consistency similar to hardware-generated reorderings.
- A key requirement for compilers is to preserve program order among shared memory operations.
Relaxed memory models

- Relax the program order requirement
  - relax the order from a write to a following read
  - relax between two writes
  - relax from a read to a following read or write

- Relax the write atomicity requirement
  - allow to read the value of its own previous write before the write is visible to other processors
Example:
IBM power Weak Memory Model
(Producer/Consumer Synchronization)

Incorrect way: without attention to weak ordering

- Compute and store data
- Store flag
- Pending updates to data
- Data now visible to consumer
Example:
IBM power Weak Memory Model (Producer/Consumer Synchronization)

Correct way: ensure writes complete before setting flag

- Compute and store data
- pending updates
- lwsync ensures that all pending writes become visible before a store after lwsync can become visible
- store flag
- data now visible to consumer
Example: IBM power Weak Memory Model (Producer/Consumer Synchronization)

Incorrect way: without attention to weak ordering

- **Loop**: load global flag
  - has global flag been set?
    - no: go to **Loop**
    - yes: fall through to **Next**

- **Problem**: consumer can speculatively execute code at **Next** before flag is set

- **Next**: use data

- **Producer** stores flag
Example: IBM power Weak Memory Model (Producer/Consumer Synchronization)

Correct way: inhibit speculative reads until flag is set

Loop: load global flag
has global flag been set?
no: go to Loop
yes: fall through to Next

Next: isync
isync causes the processor to complete all previous instructions and discard instructions after the isync that may have begun execution

producer stores flag
Many memory models in use

<table>
<thead>
<tr>
<th>Relaxation</th>
<th>W → R Order</th>
<th>W → W Order</th>
<th>R → RW Order</th>
<th>Read Others’ Write Early</th>
<th>Read Own Write Early</th>
<th>Safety net</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC [16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM 370 [14]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>serialization instructions</td>
</tr>
<tr>
<td>TSO [20]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>RMW</td>
</tr>
<tr>
<td>PC [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>RMW</td>
</tr>
<tr>
<td>PSO [20]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>RMW, STBAR</td>
</tr>
<tr>
<td>WO [5]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>synchronization</td>
</tr>
<tr>
<td>RCsc [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>RCpc [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>Alpha [19]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>MB, WMB</td>
</tr>
<tr>
<td>RMO [21]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>various MEMBAR’s</td>
</tr>
<tr>
<td>PowerPC [17, 4]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SYNC</td>
</tr>
</tbody>
</table>
Take aways from today

● Memory model critical for correct software and performance
  ○ relaxed models improve performance

● Who should care about memory models?
  ○ programmers: need to understand ordering guarantees to write correct parallel software
    ■ Insufficiently synchronized programs may behave differently on different architectures
      ● E.g., Intel processors vs. PowerPC
  ○ language designers: need to choose what guarantees a language will provide
  ○ compiler writers: need to bridge the gap between language guarantees and hardware guarantees
  ○ computer architects: need to choose what orders hardware will guarantee, provide mechanisms for obtaining stronger guarantees