Memory Consistency Models

Outline

• Review of multi-threaded program execution on uniprocessor
• Need for memory consistency models
• Sequential consistency model
• Relaxed memory models
  – weak consistency model
  – release consistency model
• Conclusions

Multi-threaded programs on uniprocessor

• Processor executes all threads of program
  – unspecified scheduling policy
• Operations in each thread are executed in order
• Atomic operations: lock/unlock etc. for synchronization between threads
• Result is as if instructions from different threads were interleaved in some order
• Non-determinacy: program may produce different outputs depending on scheduling of threads (eg)

Thread 1
  
  x := 1;  print(x);
  x := 2;

Thread 2

Multi-threaded programs on multiprocessor

• Each processor executes one thread
  – let’s keep it simple
• Operations in each thread are executed in order
• One processor at a time can access global memory to perform load/store/atomic operations
  – no caching of global data
• You can show that running multi-threaded program on multiple processors does not change possible output(s) of program from uniprocessor case

More realistic architecture

• Two key assumptions so far:
  1. processors do not cache global data
     • improving execution efficiency:
       – allow processors to cache global data
       – leads to cache coherence problem, which can be solved using coherent caches as explained before
  2. instructions within each thread are executed in order
     • improving execution efficiency:
       – allow processors to execute instructions out of order subject to data/control dependences
       – surprisingly, this can change the semantics of the program
       – preventing this requires attention to memory consistency model of processor

Recall: uniprocessor execution

• Processors reorder operations to improve performance
• Constraint on reordering: must respect dependences
  – data dependences must be respected: in particular, loads/stores to a given memory address must be executed in program order
  – control dependences must be respected
• Reorderings can be performed either by compiler or processor
Permitted memory-op reorderings

- Stores to different memory locations can be performed out of program order:
  - store v1, data
  - store b1, flag
  - store b1, flag
  - store v1, data

- Loads from different memory locations can be performed out of program order:
  - load flag, r1
  - load data, r2
  - load data, r2
  - load flag, r1

- Load and store to different memory locations can be performed out of program order

Example of hardware reordering

Problem in multiprocessor context

- Canonical model:
  - operations from given processor are executed in program order
  - memory operations from different processors appear to be interleaved in some order at the memory

- Question:
  - If a processor is allowed to reorder independent operations in its own instruction stream, will the execution always produce the same results as the canonical model?
  - Answer: no. Let us look at some examples.

Execution Sequence for (I)

Code:
- Initially A = Flag = 0
- P1
  - A = 23;
  - Flag = 1;

Possible execution sequence on each processor:
- P1
  - Write A 23
  - Write Flag 1

- P2
  - Read Flag //get 0

Problem: If the two writes on processor P1 can be reordered, it is possible for processor P2 to read 0 from variable A. Can happen on most modern processors.

Example (I)

Code:
- Initially A = Flag = 0

P1
- A = 23;
- Flag = 1;

P2
- while (Flag != 1) {};
- ... = A;

Idea:
- P1 writes data into A and sets Flag to tell P2 that data value can be read from A.
- P2 waits till Flag is set and then reads data from A.

Example II

Code: (like Dekker's algorithm)
- Initially Flag1 = Flag2 = 0

P1
- Flag1 = 1;
- If (Flag2 == 0) critical section
- ... = A;

P2
- Flag2 = 1;
- If (Flag1 == 0) critical section
- ... = A;

Possible execution sequence on each processor:
- P1
  - Write Flag1, 1
  - Read Flag2 //get 0
  - Read A //what do you get?

- P2
  - Write Flag2, 1
  - Read Flag1 //get 0
  - Read Flag1 //what do you get?
Execution sequence for (II)

Code: (like Dekker's algorithm)

Initially Flag1 = Flag2 = 0

P1
Flag1 = 1; P2
Flag2 = 1;

If (Flag2 == 0)                          If (Flag1 == 0)
    critical section  critical section

Possible execution sequence on each processor:
P1 Write Flag1, 1
Read Flag2 //get 0
P2 Write Flag2, 1
Read Flag1, ??

Lessons

- Unprocessors can reorder instructions subject only to control and data dependence constraints
- These constraints are not sufficient in shared-memory context
  - simple parallel programs may produce counter-intuitive results
- Question: what constraints must we put on uniprocessor instruction reordering so that
  - shared-memory programming is intuitive
    - but we do not lose uniprocessor performance?
- Many answers to this question
  - answer is called memory consistency model supported by the processor

Consistency models

- Consistency models are not about memory operations from different processors.
- Consistency models are not about dependent memory operations in a single processor’s instruction stream (these are respected even by processors that reorder instructions).
- Consistency models are all about ordering constraints on independent memory operations in a single processor’s instruction stream that have some high-level dependence (such as flags guarding data) that should be respected to obtain intuitively reasonable results.

Sequential Consistency

- SC constrains all memory operations:
  - Write → Read
  - Write → Write
  - Read → Read, Write
- Simple model for reasoning about parallel programs
  - You can verify that the examples considered earlier work correctly under sequential consistency.
  - However, this simplicity comes at the cost of uniprocessor performance.
- Question: how do we reconcile sequential consistency model with the demands of performance?

Simplest Memory Consistency Model

- Sequential consistency (SC) [Lamport]
  - our canonical model: processor is not allowed to reorder reads and writes to global memory

Relaxed consistency model: Weak consistency

- Programmer specifies regions within which global memory operations can be reordered
- Processor has fence instruction:
  - all data operations before fence in program order must complete before fence is executed
  - all data operations after fence in program order must wait for fence to complete
  - fences are performed in program order
- Implementation of fence:
  - processor has counter that is incremented when data op is issued, and decremented when data op is completed
- Example: PowerPC has SYNC instruction
- Language constructs:
  - OpenMP/Flush
  - All synchronization operations like lock and unlock act like a fence
Weak ordering picture

Example (I) revisited

Code:
Initially A = Flag = 0
P1
A = 23;
flush;
while (Flag != 1) {}  
Flag = 1;
... = A;
P2

Execution:
- P1 writes data into A
- Flush waits till write to A is completed
- P1 then writes data to Flag
- Therefore, if P2 sees Flag = 1, it is guaranteed that it will read the correct value of A even if memory operations in P1 before flush and memory operations after flush are reordered by the hardware or compiler.
- Question: does P2 need a flush between the two statements?

Another relaxed model: release consistency

- Further relaxation of weak consistency
- Synchronization accesses are divided into
  - Acquires: operations like lock
  - Releases: operations like unlock
- Semantics of acquire:
  - Acquire must complete before all following memory accesses
- Semantics of release:
  - all memory operations before release are complete
- However,
  - acquire does not wait for accesses preceding it
  - accesses after release in program order do not have to wait for release
    - operations which follow release and which need to wait must be protected by an acquire

Example

Which operations can be overlapped?

Implementations on Current Processors

<table>
<thead>
<tr>
<th>Processor</th>
<th>Implementations on Current Processors</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sun</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compaq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- In the literature, there are a large number of other consistency models
  - processor consistency
  - total store order (TSO)
  - ...

- It is important to remember that these are concerned with reordering of independent memory operations within a processor.
- Easy to come up with shared-memory programs that behave differently for each consistency model.
- Emerging consensus that weak/release consistency is adequate.
Summary

- Two problems: memory consistency and memory coherence
- Memory consistency model
  - what instructions is compiler or hardware allowed to reorder?
  - nothing really to do with memory operations from different processors/threads
  - sequential consistency: perform global memory operations in program order
  - relaxed consistency models: all of them rely on some notion of a fence operation that demarcates regions within which reordering is permissible
- Memory coherence
  - Preserve the illusion that there is a single logical memory location corresponding to each program variable even though there may be lots of physical memory locations where the variable is stored