Memory models:
Making sense of thread level parallelism and shared memory access

Aaron Schutza
COMP 522
9/18/2014
Outline

• Introduction to memory models
  • Why do we care about memory models today?
  • How do they affect parallel applications?
  • Sequential consistency

• Important issues
  • Correctness: data races
  • Performance: hardware optimizations

• Weak memory models
  • Categorization of relaxation methods
  • Examples of relaxed memory models

• Conclusion
  • Examples of synchronization
  • Why should we care about memory models in parallel computing?
Shared Memory Parallelism

- Shared memory parallel programs use multiple threads and a set of shared variables to perform a task
- Why are threads such a popular way of organizing software execution?
  - Supported by mainstream operating systems
  - Sharing data between threads makes it easy to use sophisticated data structures
  - Allows incremental parallelization without a complete redesign of code and data structures
- To assign meaning to a computation by multiple threads sharing memory, a program needs a well defined *memory concurrency model*
What is a Memory Model (MM)?

- Memory models define a contract between a program and any hardware or software that reorders operations in a program execution.
- Choice of a particular memory model affects ease of use, allowable reorderings, and performance.
- In the context of parallelism, it describes interactions between threads and shared memory:
  - Atomicity, ordering, visibility.
  - Determines the semantics of reads and writes subject to program order and synchronization.
Memory Models For Single Threads

• Memory operations of a program can be reordered to improve performance
  • Changes to memory ordering are generated by the compiler by producing reordered, optimized machine code
  • The hardware also reorders memory operations
    • e.g., out of order execution by a core

• Reorderings are performed with one golden rule:
  Any memory reordering will not affect the behavior of a single threaded application as long as it respects data dependences

• Operation reorderings operate silently in the background, are unobserved by developers, and improve performance
Memory Models for Parallel Threads

- Memory models describe the semantics of shared variables
  - It’s difficult to understand the semantics of a multithreaded program if we don’t know exactly how our memory model is interacting with each thread
- High level programming languages specify memory models that define semantic meaning
  - e.g., what must be ordered?
- Hardware vendors implement memory models to deliver high performance
  - e.g., how can we improve performance by reordering?
- Over the past decade this effort has exposed fundamental shortcomings of languages intended for single threaded applications
  - The advent of multiple hardware threads in consumer products has motivated the need for well-defined hardware and software MMs
  - Many of these definitions change as hardware and software evolve
Strong vs. Weak Memory Model

• **Strong**: Every machine instruction will implicitly have *acquire and release semantics* and load/store operations will be performed in program order.

• **Weak**: Any load/store operation can be reordered with another as long as that transformation doesn’t affect single thread execution.

http://preshing.com/20120930/weak-vs-strong-memory-models/
Sequential Consistency

• Definition: A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order
  • The operations of each individual processor appear in the sequence in the order specified by its program
  • This requires a strong MM
  • The most straightforward method of ensuring sequential consistency is to interleave thread executions, retaining program order in each thread

• Implications
  • No reordering of memory operations is allowed
  • All operations on shared memory execute one at a time
  • Hardware and compiler optimizations on these shared memory operations are disallowed, so there is a loss in performance
Example of Sequential Consistency

- Two threads are sharing a task with an implementation of Dekker’s algorithm with arbitrary critical sections
- Flag1 & Flag2 are shared variables
- If sequential consistency is maintained, the critical section will never be executed by both threads

Initially Flag1 = Flag2 = 0

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

P2
Flag2 = 1
if (Flag1 == 0)
    *critical section*
Importance of Atomicity

- Operation atomicity is the quality of appearing discrete to the system.
- If sequential consistency is maintained, the result of the read of A for P3 will be 1 since each read and write is simultaneously visible to P1, P2, and P3.
- In that sense, the read and write operations are *atomic*, meaning that they appear to execute discretely to the entire system.

Initially $A = B = 0$

- **P1**: $A = 1$
- **P2**: if ($A == 1$) $B = 1$
- **P3**: if ($B == 1$) register1 = $A$
Drawbacks of Sequential Consistency

- Slow: difficult to hide load and store latencies when assignments are atomic
- It’s difficult to predict the result of every possible interleaving of thread operations
- Unordered accesses, with at least one being a write, to the same shared data can produce unintended behavior (This is called a data-race)
  - If we are interleaving threads in program order that are accessing the same data, then each permutation of thread execution might give different results
  - Even though we are sequentially consistent, we get unpredictable results because there is no priority given to any particular thread (it’s actually up to the scheduler)
Data-Race-Free Programs

• We can define rules that will give consistent results for conflicting operations between threads, where at least one thread is attempting a write on a conflicting variable

• These are implemented using synchronization mechanisms to limit ambiguous orderings of reads and writes

• Some memory models will not allow data-races and in others it’s up to the programmer to develop data-race-free code
Sequential Consistency With Optimizations - I

- Write buffers are used to hide store latency
- How could that be bad?
- Suppose that at $t0$, P1 & P2 execute writes to Flag1 and Flag2 respectively
Sequential Consistency With Optimizations - I

• P1 inserts a write operation on Flag1 at $t_0$ to the write buffer and continues without waiting for it to complete.
• P2 similarly inserts a write to the buffer and continues.
• The read of Flag2 then completes at $t_1$ returning 0.
Sequential Consistency With Optimizations - 1

- Notice that P2 may execute its read before the write to Flag1 is completed at $t_3$
- At $t_2$ the read of Flag1 is performed, but the write is still waiting in the buffer so it reads a value of 0
Sequential Consistency With Optimizations - I

- Both threads are allowed to continue to the critical section
- At $t_3$ the write to Flag1 is complete
Sequential Consistency With Optimizations - I

- At $t_4$ the write to Flag2 completes
- Both threads may encounter errors if they both execute the critical section
- This violates sequential consistency
Sequential Consistency With Optimizations - II

• Below we have a general interconnect architecture where memory operations proceed in program order but don’t wait for writes to complete.
Sequential Consistency With Optimizations - II

- At $t_0$ P1 executes writes to Head and Data
- Because they are held in different memory modules, it takes different amounts of time for those writes to complete
- At $t_1$ the write to Head completes
Sequential Consistency With Optimizations - II

- At $t_2$ P2 reads Head
- Since it reads the updated value for Head, the while loop ends and P2 continues
Sequential Consistency With Optimizations - II

- At $t_3$ P2 reads Data, returning a value of 0
Sequential Consistency With Optimizations - II

- At $t4$ the write completes, updating Data to the new value of 2000
- This violates sequential consistency
Dekker’s Algorithm and the Impact of Interleavings

- If we adhere to strict atomicity of memory operations, we will never get a result of $r_1 = 0 \& r_2 = 0$ and sequential consistency is maintained.
- On the other hand, what if we have a weak MM where these executions are not atomic? Could $r_1 = 0 \& r_2 = 0$?

| Initially X=Y=0 | \[\begin{array}{c|c}
\text{Red Thread} & \text{Blue Thread} \\
X = 1; & Y = 1; \\
r_1 = Y; & r_2 = X;
\end{array}\] |
|----------------|---------------------|

<table>
<thead>
<tr>
<th>Execution 1</th>
<th>Execution 2</th>
<th>Execution 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X = 1;$</td>
<td>$Y = 1;$</td>
<td>$X = 1;$</td>
</tr>
<tr>
<td>$r_1 = Y;$</td>
<td>$r_2 = X;$</td>
<td>$Y = 1;$</td>
</tr>
<tr>
<td>$Y = 1;$</td>
<td>$X = 1;$</td>
<td>$r_1 = Y;$</td>
</tr>
<tr>
<td>$r_2 = X;$</td>
<td>$r_1 = Y;$</td>
<td>$r_2 = X;$</td>
</tr>
<tr>
<td>(/ r_1 == 0)</td>
<td>(/ r_1 == 1)</td>
<td>(/ r_1 == 1)</td>
</tr>
<tr>
<td>(/ r_2 == 1)</td>
<td>(/ r_2 == 0)</td>
<td>(/ r_2 == 1)</td>
</tr>
</tbody>
</table>
Relaxed Memory Models

• An alternative to sequential consistency is a relaxed MM

• Two main characteristics of relaxed MMs:
  • 1) How they relax the sequential program order requirement
    • Relax order between write to a following read
    • Relax ordering of two writes
    • Relax ordering of read to a following write/read
    • Only applies to operation pairs accessing different addresses
  • 2) How they relax write atomicity requirement
    • e.g., relax the visibility of one thread’s write before all cached copies have been updated or invalidated

• Both program order and atomicity can be relaxed
  • e.g., a core is allowed to read it’s own write before that write is visible to other cores
Relaxed Memory Models

- Relaxed models provide methods to override relaxation methods (*safety net*)
- These features can be used to enforce consistency in a code using a weak MM that would otherwise violate sequential consistency
Categorization of Relaxed MMs

<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>

Figure 8: Simple categorization of relaxed models. A ✓ indicates that the corresponding relaxation is allowed by straightforward implementations of the corresponding model. It also indicates that the relaxation can be detected by the programmer (by affecting the results of the program) except for the following cases. The “Read Own Write Early” relaxation is not detectable with the SC, WO, Alpha, and PowerPC models. The “Read Others’ Write Early” relaxation is possible and detectable with complex implementations of RCsc.
IBM Power Weak Memory Model

- **Global thread flag:**
  - A thread indicates that it’s done with a computation with a shared flag.
  - The setting of the flag should not occur until the computed data is stored.
  - To prevent the flag from being set and visible before the data is stored, a synchronization operation is required.

![Diagram showing computation and flag storage with and without synchronization](image)

Adapted from: http://www.ibm.com/developerworks/systems/articles/powerpc.html
IBM Power Weak Memory Model

• Waiting on a flag
  • We now switch perspective to a thread that is waiting for a flag to be set by another thread
  • In this memory model, speculative execution of branches is allowed
  • This would allow data access to the shared data before the flag is set

Adapted from: http://www.ibm.com/developerworks/systems/articles/powerpc.html
IBM Power Weak Memory Model

- **Waiting on a flag**
  - Now we insert a synchronization command into the *Yes* branch.
  - Placing `isync` at the beginning of the branch ensures that `load global flag` will complete before any loads subsequent to `isync`.
  - This will prevent any speculative execution beyond `isync`.

---

**With synchronization**

- **Loop:** load global flag
  - Has flag been set?
  - No: goto Loop
  - Yes: goto Next

- **Next:** `isync`
  - <Use data from shared block>
Conclusion

- Why are memory models important to us?
  - They define a contract between developer and system
  - Defines the semantics of multithreaded code
  - Multithreaded applications interact with memory operation reordering in non-trivial ways
  - Subtle bugs can be impossible to track down without a thorough understanding of the hardware or software MM
  - To optimize performance we allow reorderings to hide latency
  - Using weak MMs, we must construct our applications with sufficient synchronization to ensure desired behavior
- From a programmer’s perspective, MMs directly influence the design and meaning of multithreaded code
- From a system designer’s perspective, the MM defines constraints of shared memory access and optimization techniques
References


Details of Hardware Optimizations & Memory Models
Relax Write to Read Ordering

- Allow a read to be reordered with respect to writes from the same core
- IBM 370 model
- Prohibits a read from returning value of a write before the write is visible
- Safety net features:
  - Serialization instructions that are placed in between two operations
- Atomicity is not modified
Relax Write to Read Ordering

- SPARC V8 Total Store Ordering (TSO)
- Allows a read to return a value of its own core’s write before that write is visible
- Read is not allowed to return the value of another core’s write until visible
- No explicit safety net
  - Consistency is maintained through clever use of Read-Modify-Write (RMW) commands
- Atomicity is guaranteed by use of RMW commands
Relax Write to Read Ordering

- Processor Consistency (PC) model
- Read can return value of any write before write is visible or to other cores
- No explicit safety net
  - Consistency is maintained through use of RMW commands
- Atomicity is guaranteed by use of RMW commands
Initially $A = \text{Flag1} = \text{Flag2} = 0$

P1
Flag1 = 1
A = 1
register1 = A
register2 = Flag2

P2
Flag2 = 1
A = 2
register3 = A
register4 = Flag1

Result: register1 = 1, register3 = 2, register2 = register4 = 0

Initially $A = B = 0$

P1
A = 1
P2
if (A $\equiv$ 1) 
B = 1

P3
if (B $\equiv$ 1) 
register1 = A

Result: $B = 1$, register1 = 0

Figure 10: Differences between 370, TSO, and PC. The result for the program in part (a) is possible with TSO and PC because both models allow the reads of the flags to occur before the writes of the flags on each processor. The result is not possible with IBM 370 because the read of $A$ on each processor is not issued until the write of $A$ on that processor is done. Consequently, the read of the flag on each processor is not issued until the write of the flag on that processor is done. The program in part (b) is the same as in Figure 4(b). The result shown is possible with PC because it allows P2 to return the value of P1’s write before the write is visible to P3. The result is not possible with IBM 370 or TSO.
Relax *Write to Write/Read Ordering*

- We further the relaxation of program order by eliminating ordering constraints between writes to different locations
- SPARC V8 Partial Store Ordering model (PSO)
- Writes to different locations from the same core can be pipelined and are allowed to reach memory out of program order
  - Allows a read to return a value of its own processors write before that write is visible
  - Read is not allowed to return the value of another processor’s write until visible
- Safety net features:
  - STBAR (store barrier) to impose program order between writes
  - Atomicity is ensured by RMW commands (same as TSO)
Relaxing All Program Ordering

- Finally we may relax program order between all memory operations
- Memory operations following a read may be overlapped or reordered with respect to read operations
- Provides greater hardware flexibility for hiding latency of read operations
- A core may read it’s own write before it’s visible
- This scheme requires more robust safety net features:
  - Distinguishing memory operations and their type is crucial
  - Strict ordering constraints should be available for different types of operations
  - Use of fence instructions to impose ordering between specific memory operations
Weak Ordering (WO)

- Memory operations are split into two categories
  - Data operations
  - Synchronization operations
- Programmer must state what is a synchronization operation
- Each processor ensures no synchronization operation is issued until all previous operations are complete
- No operations are issued until the previous synchronization operation completes
- Memory operations between synchronous operations may still be reordered
- Writes are always atomic
Release Consistency (RCsc/RCpc)

- Release Consistency uses further distinctions of memory operations
  - Shared operations are split into special and ordinary (Synch./Data previously)
  - Sync are synchronous operations and nsync are asynchronous data operations
  - Sync is further specified by explicit acquire and release operations
- Two flavors correspond to special operations:
  - Sequential consistency is imposed (RCsc)
  - Processor consistency is imposed (RCpc)
Alpha, SPARC V9 RMO, and PowerPC

- Provide explicit fence instructions as a safety net
- Alpha includes a memory barrier and write memory barrier
- RMO model provides more fence instructions for greater control of program order
- PowerPC has a single fence instruction called SYNC
- Program order is manipulated with fence operations and RMW operations to ensure atomicity for certain memory operations
Example of Synchronization in Java

- Double check locking is a method of reducing lock acquisition overhead.
- A lock is a synchronization mechanism that enforces limits on resource access amongst threads.
- The following code segment won’t work for multithreaded applications:

```java
// Single-threaded version
class Foo {
    private Helper helper;
    public Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
        return helper;
    }

    // other functions and members...
}
```

- A lock must be obtained to prevent 2 threads from simultaneously calling `getHelper()` for the first time.

Adapted from: http://en.wikipedia.org/wiki/Double-checked_locking
• A lock is obtained with the following code:

```java
// Correct but possibly expensive multithreaded version
class Foo {
    private Helper helper;
    public synchronized Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
        return helper;
    }

    // other functions and members...
}
```

• Acquiring a lock every time `getHelper()` is called is costly so we may reorganize the code as follows:

```java
// Broken multithreaded version
// "Double-Checked Locking" idiom
class Foo {
    private Helper helper;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }

    // other functions and members...
}
```

Adapted from: http://en.wikipedia.org/wiki/Double-checked_locking
• A major flaw with this implementation is that other threads may try to access helper before its initialization is complete.

• This demonstrates a subtle impact of the language memory model that can cause unpredictable behavior in multithreaded applications.

• Determining if an implementation is broken can be difficult since it depends on many factors such as:
  • Compiler optimizations
  • Thread interleavings produced by the scheduler
  • Concurrent system activity
  • Intermittent reproducibility of bugs

Adapted from: http://en.wikipedia.org/wiki/Double-checked_locking
The following code updated for Java 5 fixes this problem because the `volatile` keyword ensures a global ordering to reads and writes of that variable:

```java
// Works with acquire/release semantics for volatile
// Broken under Java 1.4 and earlier semantics for volatile
class Foo {
    private volatile Helper helper;
    public Helper getHelper() {
        Helper result = helper;
        if (result == null) {
            synchronized(this) {
                result = helper;
                if (result == null) {
                    helper = result = new Helper();
                }
            }
        }
        return result;
    }
    // other functions and members...
}
```

- Access to volatiles act like fences
- They use `result` so that `helper` is accessed only once per call to `getHelper()` for optimization

Adapted from: http://en.wikipedia.org/wiki/Double-checked_locking