Shared Memory
Consistency Models

Authors: Sarita V. Adve and Kourosh Gharachorloo

Presented by Arvindh Shriram

Motivations

- Programmer is required to reason about consistency to ensure data race conditions don’t occur
- Programmer is required to understand the underlying memory consistency model when porting code across architectures
- Heavy performance penalty imposed by sequential consistency
- Relaxed consistency models make programming tougher but improve performance

Optimizations for Performance

- Hardware
  - Write Buffers
  - Out-of-Order Instruction issue
  - Pipelining of Memory Operations
  - Lock-Up free Caches
  - Non-Atomic Memory operations
- Software
  - Data dependence analysis
  - Instruction reordering
  - Register allocation

Memory Consistency

Memory Consistency is an alias for Memory Semantics

To write correct and efficient programs the programmer uses the MC to reason about the reordering of reads and writes to the memory. The programmer uses MC to reason when the reads/writes become visible to other processors in the system.
Outline

- Sequential Consistency – commonly assumed
  - Architectures with caches
  - Architectures without caches
- Relaxed Consistency – not intuitive
  - Taxonomy
  - IBM 370
  - TSO, PC
  - RC (RCsc, RCpc)
  - Alpha, Power
- Performance
- Programmer Centric Framework

Sequential Consistency (SC) [Lamport 79]

- Result of any execution is the same as if operations of all processors were executed in a sequential order, and operation of each processor appeared in program order.
  - Maintaining program order within a processor
  - Maintaining a sequential order among operations from processors

SC Example

Flag1=0, Flag2=0
Flag1=1, Flag2=1
If (Flag2=0) If (Flag1=0)
Critical Section Critical Section

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=1</td>
<td>if (A=1)</td>
<td>R1=A</td>
</tr>
<tr>
<td>B=1</td>
<td>if (B=1)</td>
<td></td>
</tr>
</tbody>
</table>

Final Value - R1=1

Cache Independent Optimizations
Write Buffers

Write Buffer with bypassing
- Reads bypass writes. Reads are blocking
- Write, Operation placed into buffer and processor continues

Flag1=0, Flag2=0
P1
P2
Flag1= 1  Flag2=1
P1            P2
If (Flag2=0)  If (Flag1=0)
Critical Section  Critical Section

Bypass violates sequential consistency.

Pipelined Writes

Writes may bypass other writes in write buffer
- Improves buffer utilization
- Utilizes bandwidth of multiple memory banks.
- Point-Point interconnection network alleviates bus bottlenecks.

Non-Blocking Reads

Reads are allowed to bypass reads and writes
- Dynamic Scheduling, Lockup-free caches, Speculative execution
- Point-Point interconnection network alleviates bus bottlenecks.

Architectures with Caches
Multiprocessor Systems

- Requires cache coherence protocol to ensure that newly written values are propagated to other processors
- Write completion detection to prevent inconsistency between various cached copies
- Maintain illusion of atomic write operations
- Cache coherency assures that newly written values are propagated to the multiple cached copies in system
- Memory Consistency policy places an early and late bound on when a new value is propagated to a processor

Non-Atomic Writes

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=1</td>
<td>B=1</td>
<td>Spin(B);Spin(C)</td>
<td>R1=A</td>
</tr>
<tr>
<td>A=2</td>
<td>C=1</td>
<td></td>
<td>R1=A</td>
</tr>
<tr>
<td>R2=A</td>
<td>R2=A</td>
<td>Spin(B);Spin(C)</td>
<td>R2=A</td>
</tr>
</tbody>
</table>

Result – R1=1, R2=2 - Is it possible?
Yes

Solution
- All updates by processors P1,P2 reach P3 and P4 in different order
- P3 and P4 read different values due to general interconnect network

Cache Coherence Vs SC

- Cache coherence ensures
  - Serialization of writes to same locations
  - Write visible to all processors
- Sequential consistency stronger condition requires
  - All processors to see all writes to all locations in the same order
  - All operations in a program to be in same order
- Sequential consistency implies Cache Coherence

P1
 X=1

P2
 X=2

Result R1=1, R2=2, R3=2, R4=1
Does Cache Coherence allow result? No
Does SC permit the result? No

P1
 X=1

P2
 Y=2

Result R1=1, R2=0, R3=2, R4=0
Does Cache Coherence allow result? Yes
Does SC permit the result? No

P1
 X=1

P2
 R1=X;R2=Y

Result R1=X, R2=Y, R3=X, R4=X

P3
 R3=X;R4=X

P4
 R3=X;R4=X
Outline
- Sequential Consistency – commonly assumed
  - Architectures with caches
  - Architectures without caches
- Relaxed Consistency – not intuitive
  - Taxonomy
  - IBM 370
  - TSO,PC
  - RC (RCsc,RCpc)
  - Alpha, Power
- Performance
- Programmer Centric Framework

RMM Classification
Based on Program Order requirement
- Writes to Reads
- Writes to Writes
- Read to Reads/Writes
Based on Write Atomicity
- Read other processor’s writes early
- Read own write early

Memory Access Restrictions
<table>
<thead>
<tr>
<th>Dependency preserved</th>
<th>Processor Consistency (PC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Read</td>
</tr>
<tr>
<td>Read</td>
<td>Write</td>
</tr>
</tbody>
</table>

Read Others write early
Read own write early

RMM Classification (contd….)
Weak Consistency (WC)
- Acquire A
- Acquire B
- Release A
- Release B

Read Consistency (RL)
- Acquire A
- Acquire B
- Release A
- Release B

Read Other’s write Early
Read Own Write Early
PC: W -> RW + R -> RW

(Contd : W -> RW + R -> RW)

Acquire – Synchronization to gain access to Shared Memory
Release – Release Access to Shared Memory
Why does RMM make sense?

Architecture performance bounded by performance of memory system. Idle time consists of:
- Read access stalling for data
- Processor stalling for previous writes
- Write access when write buffer is full
- Acquire spinning until release obtained

RMM helps relax program memory access ordering and reduces idle time and stall

Outline

- Sequential Consistency – commonly assumed
  - Architectures with caches
  - Architectures without caches
- Relaxed Consistency – not intuitive
  - Taxonomy
  - IBM 370
  - TSO,PC
  - RC (RCsc,RCpc)
  - Alpha, Power
- Performance
- Programmer Centric Framework

Relaxing Writes to Reads

- IBM 370 stalls on read before write made visible to system
- TSO permits early read of write from own processor before serialization
- PC permits read of all writes without serialization.
- IBM 370 provides serialization instructions such as CAS and T&S.
- TSO provide RMW instructions. Replace Read and Write with RMW. Program order ensures serialization from W to R
- PC requires all accesses to memory location to be replaced with a RMW instruction since other processors may write to same location
- Primarily used to hide latency of write operations

Outline

- Sequential Consistency – commonly assumed
  - Architectures with caches
  - Architectures without caches
- Relaxed Consistency – not intuitive
  - Taxonomy
  - IBM 370
  - TSO,PC
  - RC (RCsc,RCpc)
  - Alpha, Power
- Performance
- Programmer Centric Framework

Relaxing Writes to Reads

- IBM 370 stalls on read before write made visible to system
- TSO permits early read of write from own processor before serialization
- PC permits read of all writes without serialization.
- IBM 370 provides serialization instructions such as CAS and T&S.
- TSO provide RMW instructions. Replace Read and Write with RMW. Program order ensures serialization from W to R
- PC requires all accesses to memory location to be replaced with a RMW instruction since other processors may write to same location
- Primarily used to hide latency of write operations

Outline

- Sequential Consistency – commonly assumed
  - Architectures with caches
  - Architectures without caches
- Relaxed Consistency – not intuitive
  - Taxonomy
  - IBM 370
  - TSO,PC
  - RC (RCsc,RCpc)
  - Alpha, Power
- Performance
- Programmer Centric Framework

Relaxing Writes to Reads

- IBM 370 stalls on read before write made visible to system
- TSO permits early read of write from own processor before serialization
- PC permits read of all writes without serialization.
- IBM 370 provides serialization instructions such as CAS and T&S.
- TSO provide RMW instructions. Replace Read and Write with RMW. Program order ensures serialization from W to R
- PC requires all accesses to memory location to be replaced with a RMW instruction since other processors may write to same location
- Primarily used to hide latency of write operations
**370 vs TSO vs PC**

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>A=1</td>
<td>A=2</td>
<td>A=2</td>
</tr>
<tr>
<td>R1=A</td>
<td>R3=A</td>
<td>R3=A</td>
</tr>
<tr>
<td>R2=F2</td>
<td>R4=F1</td>
<td></td>
</tr>
</tbody>
</table>

Result: R1=1; R3=2; R2=R4=0

Which Model permits the above result?
- 370 – No
- TSO – Yes
- PC – Yes

**Relaxing W to R and W to W (SPARC V8)**

- PSO= TSO + W => W relaxation
- Writes can be pipelined and can be allowed to complete out of order
- PSO is similar to TSO in write atomic requirements
- PSO will allow both previous examples
- Provides an STBAR (Store Barrier. I think!) inst.
- STBAR once inserted into Write buffer, prevents newer writes completing until all older writes lower in stack have completed

**Relaxing all program orders**

Classified as (1) Weak Consistency (2) Release Consistency

Weak Consistency – Weak Ordering
- Release Consistency – RCpc (PowerPC) , RCsc
- Every type of reordering is permitted
- Enables optimization of read operation and hide latency of reads using Speculative execution and Lockup-Free Caches
- Commodity processor provides special “MBAR” and “Sync” instructions to preserve ordering

**Outline**

- Sequential Consistency – commonly assumed
  - Architectures with caches
  - Architectures without caches
- Relaxed Consistency – not intuitive
  - Taxonomy
  - IBM 370
  - TSO, PC
  - WO
  - RC (RCsc,RCpc)
- Alpha, Power
- Performance
- Programmer Centric Framework
Weak Ordering

- Classifies instructions into “Data” and “sync”
- Enables reordering Load/Stores between two sync instructions.
- Hardware implements WO using counters which ensure all operations up to acquire are completed.
- Requires programmer to make sure that no other process requires this data to be in a consistent state until release.
- Processor stalls for pending writes and release to complete
- Processor stalls at read for release to complete

Release Consistency

- Extends WO and makes distinction amongst the sync and non-sync operations.
- Two types
  - RCsc : Maintains SC between sync operations
    - acquire => all, all=>release and special => special
    - Write atomicity preserved by label “special”
  - RCpc : Maintains PC between sync operations
    - acquire => all, all => release and special => special
    - relaxes W => R because of PC
    - Allows processor to read other’s write early
    - RMW used to preserve write atomicity
- No optimization barred! except on RAW

Outline

- Sequential Consistency – commonly assumed
  - Architectures with caches
  - Architectures without caches
- Relaxed Consistency – not intuitive
  - Taxonomy
  - IBM 370
  - TSO,PC
  - WO
  - RC (RCsc, RCpc)
  - Alpha, Power
- Performance

Alpha, SPARC V9 and PowerPC

- Alpha processors employ the RCsc model. MB (Membar) set up barrier for memory operations. WMB instructions allow R=>W.
- SPARC V9 employs the RCsc model.
  - Types of MEMBAR instructions relax ordering. (you don’t need RMW to preserve W => R )
  - Write atomicity is maintained.
- PowerPc designed with RCpc.
  - Sync instruction provides ordering for all except R => R
  - RMW required to make writes atomic and preserve R => R
**SC Vs PC**

- PC allows reads to hide write latency
- PC beneficial when
  - Write misses high
  - Heavy write miss penalty
- PC increases read latencies since heavier workload on the memory system
- In PC higher probability of write buffer becoming full
  - Rarely happens as write and read misses are interleaved
  - Read miss stalls enable write misses to retire

**PC Vs WC**

- WC exploits knowledge of synchronization access
- WC writes are pipelined. Write completion doesn’t need ownership
- WC writes retire much faster
  - Write buffer will not overflow
  - Release can be observed sooner (spins are short)
  - Clustered writes have a benefit with WC
- WC reads and acquires are stalled for release
  - Performance degrades if application has fine-grain sync

**PC Vs RC**

- RC provides faster completion of synchronization
  Processor does not stall for pending acquires and releases
- RC requires more complex implementation
  Number of outstanding requests to be supported
- RC floods network and bank contention by indiscriminately increasing the number of overlapped memory operations

**Performance**
Discussion

- In LU processors synchronize over the pivot element
  - Write pipelining enables early issue of writes
  - Write pipelining lowers synchronization costs
  - Write buffer overflow does not occur
- PTHOR performance gain is RC due to faster synch
  - ‘Read’ and ‘Acquires’ are issued immediately after ‘Release’, which leads to frequent stalling in WC
- In MPID rate of synchronization is low, hence neither WC nor RC have any effect.
- Assumptions
  - Processor stalls on reads which limits gains of RC
  - If reads stall the processor then PC performance is nearly same as RC
  - Prefetching techniques improve performance of more consistent models
  - Architecture employs invalidation protocol, which leads to large number of read misses
  - Write Buffer will overflow if \( R_{\text{miss}} \times \text{Latency}_{\text{read}} \) > \( W_{\text{miss}} \times \text{Latency}_{\text{write}} \)

Programmer Interface

- Programmer ensures that all the synchronization operations are explicitly labeled
- Compiler/Runtime system translates these into explicit MemBAR and Fence instructions
- Identifying Synchronization Events
  - Conflicting operations
  - Competing operations
- A parallel program is synchronized if all competing operations have been labeled as synchronization operations.

Assumptions

- Processor stalls on reads which limits gains of RC
- If reads stall the processor then PC performance is nearly same as RC
- Prefetching techniques improve performance of more consistent models
- Architecture employs invalidation protocol, which leads to large number of read misses
- Write Buffer will overflow if \( R_{\text{miss}} \times \text{Latency}_{\text{read}} \) > \( W_{\text{miss}} \times \text{Latency}_{\text{write}} \)

Summary

- Laid out distinction between Reads/Writes and Sync operations ordering and overlapping leading to various Memory Consistency models
- Defined “Sequential Consistency” and discussed optimization effects leading to violation of SC
- Discussed Cache Coherency from perspective of SC
- Related Memory Models
  - Organized the memory consistency models based on (1) Program order relaxing (2) Write Atomicity Preservation
    - IBM 370, TSO, PC
    - SPARC 64
    - W
    - RC (RC64/RC74)
  - Compiling processors and ISA provisions for specifying order
- Analyzed performance of a set of shared memory applications
  - If Reads are blocking then PC is a better option over more complex RMMs
- Defined operations user has to identify in a programmer
  (For those who slept through the presentation. Will help you fill the questionnaire)

Choosing a Memory Model

- What is the target environment (application, programmers, and architecture)?
- How restrictive is the model with respect to different programming styles?
- What are the compiler optimizations that can be employed? How difficult is it to implement?
Thank You!