Shared Memory Consistency

CSE 661 – Parallel and Vector Architectures

Prof. Muhamed Mudawar

Computer Engineering Department

King Fahd University of Petroleum and Minerals

Outline of this Presentation

- Uniprocessor Memory Consistency
- Shared Memory Consistency
- Sequential Consistency
- Relaxed Memory Consistency Models
  - PC: Processor Consistency
  - TSO: Total Store Order
  - PSO: Partial Store Order
  - WO: Weak Ordering
  - RMO: Relaxed Memory Ordering
  - RC: Release Consistency

Uniprocessor Memory Consistency

- Simple and intuitive sequential-memory semantics
  - Presented by most high-level programming languages
  - All memory operations assumed to execute in program order
    - Each read must return the last write to the same address
- Sequential execution can be supported efficiently, while
  - Ensuring data dependences
    - When two memory operations access the same memory location
  - Ensuring control dependences
    - When one operation controls the execution of another
- Compiler or hardware can reorder unrelated operations
  - Enabling several compiler optimizations
  - Allowing a wide range of efficient processor designs

Shared Memory Consistency

- In a shared memory multiprocessor …
  - Multiple processors can read and write shared memory
  - Shared memory might be cached in more than one processor
  - Cache coherence ensures **same view** by all processors
- But cache coherence does not address the problem of
  - **How Consistent** the view of shared memory must be?
    - When should processors see a value that has been updated?
  - Is **reordering** of reads/writes to **different** locations allowed?
    - In a uniprocessor, it is allowed and not considered an issue
    - But in a multiprocessor, it is considered an issue
- Memory consistency specifies constraints on the …
  - Order in which memory operations can appear to execute

Shared Memory Consistency: Example

- Consider the code fragments executed by P1 & P2

```
P1:  A = 0;                  P2:  B = 0;
     ...                          ...
     A = 1;                       B = 1;
L1:  if (B == 0) ...         L2:  if (A == 0) ...
```

- Can both if statements L1 & L2 be true?
  - Intuition says **NO**, it can't be
  - At least **A** or **B** must have been assigned **1** before **if**
- But reading of **B** and **A** might take place before writing 1
  - Reading of **B** in P1 is independent of writing **A = 1**
    - Read hit on **B** might take place before Bus Upgrade on writing **A**
  - Same thing might happen when reading **A** in P2
- Should this behavior be allowed?

Another Example

- Initial values of **A** and **flag** are assumed to be 0

P1	P2
`A = 1;`	`while (flag == 0); /*spin*/`
`flag = 1;`	`print A;`

- Programmer Intuition
  - Flag is set to **1** after writing **A** in P1, so P2 should print **1**
- However, memory coherence does not guarantee it!
  - Coherence says nothing about the **order** in which
    - Writes to **A** and **flag** (**different memory locations**) become visible
  - **A = 1** may take place after **flag = 1**, **not in program order!**
- Coherence only guarantees that …
  - New value of **A** will eventually become visible to P2
    - But not necessarily before the new value of **flag** is observed

Shared Memory Consistency Model

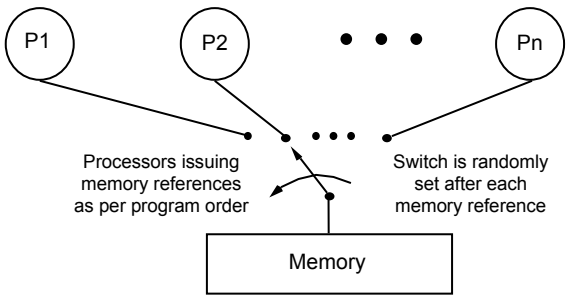
- Specifies constraints on order of memory operations
  - Which memory operation orders are preserved?
  - Enables programmers to reason about correctness and results
- Is an interface between the programmer and the system
  - Interface at the high-level language
    - Which optimizations can the compiler exploit?
  - Interface at the machine-code
    - Which optimizations can the processor exploit?
- Influences many aspects of parallel system design
  - Affects hardware, operating system, and parallel applications
- Affects performance, programmability, and portability
  - Lack of consensus on a single model

Sequential Consistency

"A multiprocessor 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, and the operations of each individual processor occur in this sequence in the order specified by its program" (Lamport 1979)

Programmer Abstraction of the Memory System

Model completely hides underlying concurrency in memory system hardware



Lamport's Requirements for SC

1. Each processor issues memory requests in the …
   - Order specified by its program ⇒ Program Order
2. Memory requests issued from all processors are
   - Executed in some sequential order
   - As if serviced from a single FIFO queue

❖ Assumes memory operations execute atomically
   - With respect to all processors
   - Each memory operation completes before next one is issued

❖ Total order on interleaved memory accesses
   - As if there were no caches, and a single shared memory module

Dubois Requirements for SC

1. Each processor issues memory requests in …
   - The order specified by the program ⇒ Program Order
2. After a store operation is issued …
   - Issuing processor should wait for the store to complete
     - Before issuing its next memory operation
3. After a load operation is issued …
   - Issuing processor should wait for the load to complete
     - Before issuing its next memory operation

❖ Last 2 points ensure atomicity of all memory operations
   - With respect to all processors

What Really is Program Order?

- Intuitively, order of memory operations in source code
  - As seen by the programmer
- Straightforward translation of source code to assembly
  - Order in assembly/machine code is same as in source code
- However, optimizing compiler might reorder operations
  - Uniprocessors care only about dependences to same location
  - Independent memory operations might be reordered
  - Loop transformations, register allocation
  - Compiler tries to improve performance on uniprocessors
  - So compiler optimization must be taken into consideration

SC is Different from Cache Coherence

- Requirements for cache coherence
  - **Write propagation**
    - A write must eventually be made visible to all processors
    - Either by invalidation or updating each copy
  - **Write serialization**
    - Writes to the same location
    - Appear in the same order to all processors
- Cache coherence is only part of Sequential Consistency
- The above conditions are not sufficient to satisfy SC
  - Program order on memory operations is not specified
  - Whether memory operations execute atomically is not specified

Some Optimizations that Violate SC

- Write Buffers with Read Bypassing
  - Processor inserts a write into a write buffer and proceeds
    - Without waiting for the write to complete
  - Subsequent unrelated reads can bypass the write in buffer
  - Optimization gives priority to reads to reduce their latency
- Non-Blocking Reads
  - Recent processors can proceed past a read miss
  - Subsequent unrelated memory operation can bypass read miss
  - Using a non-blocking cache and dynamic scheduling
- Out-of-Order Writes
  - Multiple writes may be serviced concurrently
  - Writes may complete out of program order

Write Buffers with Read Bypassing

- Following example shows importance of …
  - Maintaining order between a write and a following read
  - Even when there is no data or control dependence between them

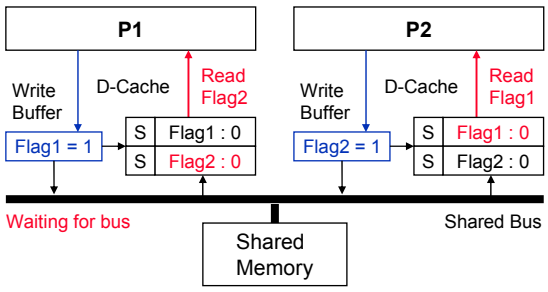
Decker's Algorithm for ensuring mutual exclusion

P1

```
Flag1 = 1
if (Flag2 == 0) {
  Critical Section
}
```

P2

```
Flag2 = 1
if (Flag1 == 0) {
  Critical Section
}
```

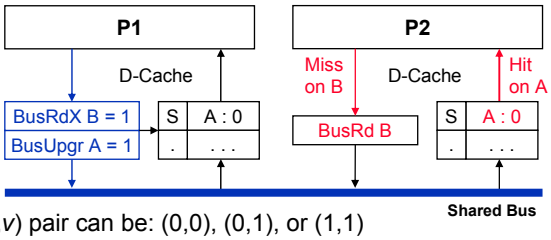


Non-Blocking Reads

- Following example shows importance of …
  - Maintaining order between a read and a following operation
  - Even when there is no data or control dependence between them

A and B are initially 0

P1	P2
A = 1	u = B
B = 1	v = A
. . .	. . .



- Possible values for (*u*,*v*) pair can be: (0,0), (0,1), or (1,1)
- However, (*u*,*v*) cannot be (1,0) under Sequential Consistency
- With a non-blocking cache read, (*u*,*v*) = (1,0) is possible
- Read hit on *A* bypasses a read miss on *B*
- The two write transactions in P1 might take place before BusRd B

Out-of-Order Writes

- Writes to the same block are sometimes combined
  - A common optimization to reduce bus transactions
  - Merged writes might complete out-of-order

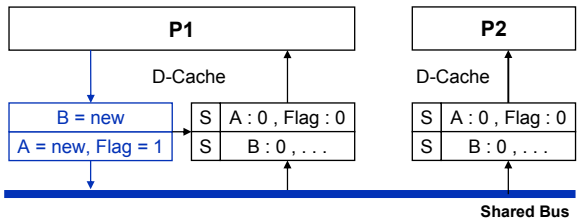
Event Synchronization

P1

A = *new value*
B = *new value*
Flag = 1

P2

while (Flag == 0) {}
Use A
Use B



- *A* and *Flag* might reside in same block
- Writes to *A* and *Flag* might be combined
- Write to *Flag* might occur before write to *B*
- Processor P2 might see old value of B

Out-of-Order Writes – cont'd

- Consider a distributed shared memory multiprocessor
  - Writes are issued in-order, but might complete out-of-order
- Following example shows importance of write completion
  - To maintaining program order between two writes

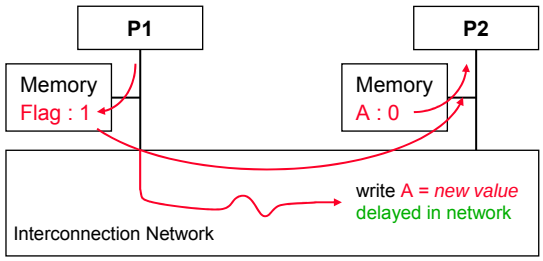
Event Synchronization

P1

A = *new value*
Flag = 1
. . .

P2

while (Flag == 0) {}
Use A
. . .

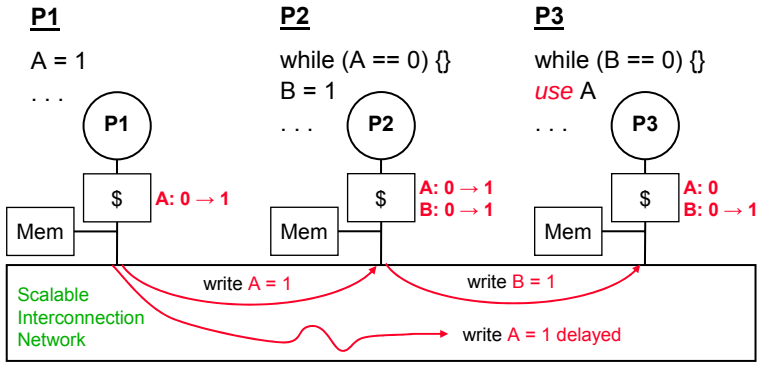


Write Atomicity

- Sequential Consistency requires that all memory ops …
  - Execute atomically with respect to all processors
  - In addition to program order within a process
- Write atomicity is an important issue
  - Writes to all locations must appear to all processors in same order
- Extends write serialization of cache coherence
  - Write serialization requires that …
    - Writes to same location only appear in same order
  - But write atomicity requires that for all memory locations

Violation of Write Atomicity

- Consider a distributed shared memory multiprocessor
  - Write atomicity can be easily violated if a write is made visible to some processors before making it visible to others
- Importance of write atomicity to SC is shown below



Implementing Sequential Consistency

1. Every process issues memory operations in program order
   - Even when memory operations address different memory locations
2. Wait for a write to complete before next memory operation
   - In a bus-based system, write completes as soon as bus is acquired
     - Bus Read Exclusive, Bus Upgrade, Bus Update
   - In a scalable multiprocessor, with a scalable interconnect
     - A write requires explicit acknowledgements if multiple copies exist
     - Each processor acknowledges an invalidate or update on receipt
3. Maintain write atomicity
   - Wait for a write to complete with respect to all processors
   - No processor can use new value until it is visible to all processors
   - Challenging with update protocol and scalable non-bus network

- Conditions are very restrictive for performance

Compilers

- Compilers that reorder shared memory operations …
  - Cause sequential consistency violations
  - Similar to hardware generated reordering
- Compiler must preserve program order …
  - Among shared memory operations
  - But this prohibits compiler optimizations
- Simple optimizations that violate SC include
  - Register allocation to eliminate memory access
  - Eliminating common sub-expressions
- Sophisticated optimizations that violate SC include
  - Instruction reordering
  - Software pipelining

Example on Register Allocation

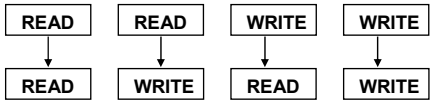
- Register allocation can violate sequential consistency
- Can cause the elimination of shared memory access
- In the following example …
  - Compiler might easily allocates r1 to B in P1 and r2 to A in P2

P1	P2	P1	P2
B=0	A=0	r1=0	r2=0
A=1	B=1	B=r1	A=r2
u=B	v=A	A=1	B=1
		u=r1	v=r2
(u,v) ≠ (0,0) under SC		(u,v) = (0,0) occurs here	

- Unfortunately, programming languages and compilers are largely oblivious to memory consistency models

Summary of Sequential Consistency

- Maintain order between shared access in each process
  - Reads or writes wait for previous reads or writes to complete
  - Total order on all accesses to shared memory



- Does SC eliminate synchronization?
  - No, still needs critical sections, barriers, and events
- SC only ensures interleaving semantics
  - Of individual memory operations

Relaxed Memory Models

- Sequential consistency is an intuitive programming model
  - However, disallows many hardware and compiler optimizations
- Many relaxed memory models have been proposed
- PC: Processor Consistency (Goodman 89)
- TSO: Total Store Ordering (Sindhu 90)
  - Relaxing the Write-to-Read Program Order
- PSO: Partial Store Ordering (Sindhu 91)
  - Relaxing the Write-to-Read and Write-to-Write Program Orders
- WO: Weak Ordering (Dubois 86)
- RC: Release Consistency (Charachorloo 1990)
- RMO: Relaxed Memory Ordering (Weaver 1994)
  - Relaxing all program orders for non-synchronization memory ops

PC and TSO : Relaxing Write-to-Read

- Allow a read to bypass an earlier incomplete write
- Motivation: hide latency of write operations
  - While a write-miss is placed in write buffer and not visible yet
  - Later reads that hit in the cache can bypass the write
- Most early multiprocessors supported PC or TSO
  - Sequent Balance, Encore Multimax, Vax 8800
  - SparcCenter 1000/2000, SGI Challenge, Pentium Pro quad
- Difference between PC and TSO is that …
  - TSO ensures write atomicity, while PC does not ensure it
- Many SC example codes still work under PC and TSO

Correctness of Results

P1	P2
A = 1;	while (Flag == 0) { }
Flag = 1;	Read A;

(a)

P1	P2
A = 1;	Read B;
B = 1;	Read A;

(b)

P1	P2	P3
A = 1;	while (A == 0) { }	while (B == 0) { }
	B = 1;	Read A;

(c)

P1	P2
A = 1;	B = 1;
read B;	Read A;

(d)

(a) and (b): Same for SC, TSO, and PC

(c) PC allows A to be read as 0 --- no write atomicity

(d) TSO and PC allow A and B to be read as (0,0)

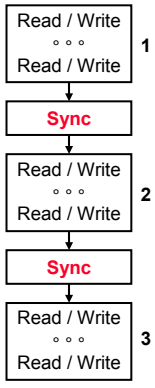
- Sequential Consistency can be ensured using
  - Special memory barrier or fence instructions – discussed later

PSO: Partial Store Ordering

- Processor relaxes write-to-read & write-to-write orderings
  - When addressing different memory locations
- Hardware Optimizations
  - Write-buffer merging
  - Enables multiple write misses to be fully overlapped
  - Retires writes out of program order
- But, even the simple use of flags breaks under this model
  - Violates our intuitive sequential consistency semantics
- PSO model is supported only by Sun Sparc (Sindhu 1991)
- Sparc V8 provides STBAR (store barrier) instruction
  - To enforce ordering between two store instructions

Weak Ordering

- Relaxes all orderings on non-synchronization operations
  - That address different memory locations
  - Retains only control and data dependences within each thread
- Motivation
  - Parallel programs use synchronization operations
    - To coordinate access to shared data
  - Synchronization operations await …
    - All previous memory operations to complete
  - Order of memory access need not be preserved
    - Between synchronization operations
- Matches dynamically scheduled processors
  - Multiple read misses can be outstanding
  - Enable compiler optimizations



Alpha and PowerPC

- Processor relaxes all orderings on memory operations
  - When addressing different memory locations
- However, Specific instructions enforce ordering
  - Called memory barriers or fences
- Alpha architecture: two kinds of fences (Sites 1992)
  - Memory Barrier (MB)
    - Wait for all previously issued memory operations to complete
  - Write Memory Barrier (WMB)
    - Imposes program order only between writes (like STBAR in PSO)
    - However, a read issued after WMB can still bypass
- IBM PowerPC provides only a single fence (May 1994)
  - SYNC equivalent to Alpha's MB, but writes are not atomic

Sparc V9 RMO Model

- RMO: Relaxed Memory Order (Weaver 1994)
  - Processor relaxes all orderings on memory operations
  - When addressing different memory locations
- Provides a memory barrier instruction called MEMBAR
  - Similar to Alpha and PowerPC but with different flavors
- Sparc V9 MEMBAR has 4 flavor bits
  - Each bit indicates a particular type of ordering to be enforced
  - LoadLoad bit enforces read-to-read ordering
  - LoadStore bit enforces read-to-write ordering
  - StoreLoad bit enforces write-to-read ordering
  - StoreStore bit enforces write-to-write ordering
  - Any combination of these 4 bits can be set

Examples on Memory Barriers

❖ Sparc V9 MEMBAR is used in the following examples

Event Synchronization

P1

A = *new value*
membar #StoreStore
Flag = 1

P2

while (Flag == 0) {}
membar #LoadLoad
Use A

Decker's Algorithm for ensuring mutual exclusion

P1

Flag1 = 1
membar #StoreLoad
if (Flag2 == 0) {
  Critical Section
}

P2

Flag2 = 1
membar #StoreLoad
if (Flag1 == 0) {
  Critical Section
}

Release Consistency

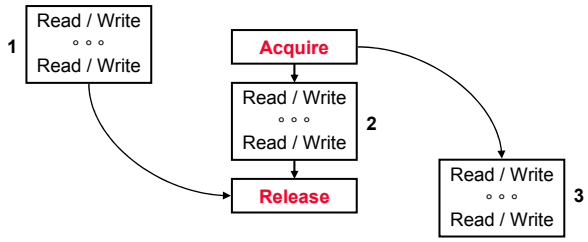
❖ Extends weak ordering model
- ✸ Distinguishes among types of synchronization operations
- ✸ Further relaxing ordering constraints

❖ Acquire: read or read-modify-write operation
- ✸ Gain access to a set of operations on shared variables
- ✸ Delay memory accesses that follow until acquire completes
- ✸ Has nothing to do with memory accesses that precede it

❖ Release: write operation
- ✸ Grant access to another processor to the …
  - ✧ New set of data values that are modified before in program order
- ✸ Release must wait for preceding memory accesses to complete
- ✸ Has nothing to do with memory accesses that follow it

Release Consistency – cont'd

- In the example shown below …
  - Block 1 precedes acquire and block 3 follows release
  - Acquire can be reordered with respect to accesses in block 1
  - Release can be reordered with respect to accesses in block 3
  - Blocks 1 and 2 have to complete before release
  - Blocks 2 and 3 cannot begin until acquire completes



Examples on Acquire and Release

P1 , P2 , … , Pn

```
. . .
Lock(TaskQ);
newTask->next = Head;
if (Head != NULL)
  Head->prev = newTask;
Head = newTask;
Unlock(TaskQ);
. . .
```

P1

```
TOP: while(flag2==0);
     A = 1;
     u = B;
     v = C;
     D = B * C;
     flag2 = 0;
     flag1 = 1;
     goto TOP;
```

P2

```
TOP: while(flag1==0);
     x = A;
     y = D;
     B = 3;
     C = D / B;
     flag1 = 0;
     flag2 = 1;
     goto TOP;
```

- Examples on acquire
  - Lock(TaskQ) in the first example
  - Reading of flag1 and flag2 within the while loop conditions
- Examples on release
  - Unlock(TaskQ) in the first example
  - Setting of flag1 and flag2 to 1 in the second example

Summary of Various Models

Model	W→R Reorder	W→W Reorder	R→RW Reorder	Read Other's Write Early	Read Own Write Early	Ordering Operations
SC					yes	
TSO	yes				yes	membar, rmw
PC	yes			yes	yes	membar, rmw
PSO	yes	yes			yes	stbar, rmw
WO	yes	yes	yes		yes	sync
RC	yes	yes	yes	yes	yes	acq, rel, rmw
RMO	yes	yes	yes		yes	membar #
Alpha	yes	yes	yes		yes	mb, wmb
PowerPC	yes	yes	yes	yes	yes	sync

- RMW are read-modify-write operations
- ACQ and REL are the acquire and release operations
- MEMBAR # is the memory barrier with various flavors

Summary of Various Models – cont'd

- Read Own Write Early relaxation
  - Processor is allowed to read its own previous write …
    - Before the write completes
    - Write can be still waiting in write buffer
  - Optimization can be used with SC and other models …
    - Without violating their semantics
- Read Other's Write Early relaxation
  - This is the non-atomic write
  - Processor is allowed to read result of another processor write
    - Before the write completes globally with respect to all processors