Lec-11

Multi-Threading Concepts: Coherency, Consistency, and Synchronization

ECE-720-t4: Innovations in Processor Design
2008t1 (Winter)
Mark Aagaard
University of Waterloo

Introduction

Coherency vs Consistency

- Memory coherency and consistency are major concerns in the design of shared-memory systems.

Consistency

- guarantees that a **single processor** can execute a **sequential program** correctly
- defines the relative order of ***read and write operations*** from multiple processors to a ***single*** memory location.

Coherency

- guarantees that **multiple processors** can execute a **concurrent program** correctly
- defines the relative order of ***write operations*** from multiple processors to ***multiple*** memory locations.

P1	P2	P3
1: Wr A $\leftarrow \alpha_1$	1: Wr A $\leftarrow \alpha_2$	1: Wr A $\leftarrow \alpha_3$
2: Wr B $\leftarrow \beta_1$	2: Rd A	2: Rd A
3: Wr A $\leftarrow \alpha_4$	3: Rd B	3: Rd B
4: Wr B $\leftarrow \beta_2$		

Simple Memory Coherency

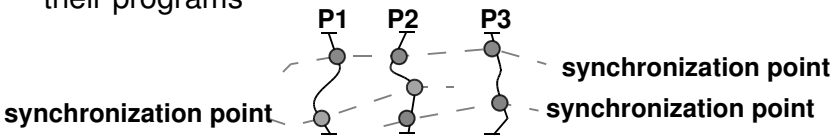
- Three requirements for simple coherency
  - If P_i writes α to A, and then reads from A without any intervening writes by P_i or another processor, then the read will return α.
  - If P_i writes α to A, and then after a sufficiently long time without any intervening writes by P_i or another processor, then if P_j reads A, the read will return α.
  - If P_i writes α_1 to A and P_j writes α_2 to A, then all processors will agree on the relative order of the two write operations.

Simple Memory Consistency

- Two requirements for simple memory consistency
  - A write operation is defined to be ***completed*** when all processors have seen the effect of the write
  - Each processor preserves the order of writes with respect to both reads and writes

Synchronization

- Basic idea: force multiples processors to stop at particular points in their programs



- Purpose of synchronization in hardware:
  Provide basic building blocks for messages and critical sections in software.

Coherency Introduction

Terminology for Shared Memory

	Symmetric	Distributed
Location of memory		
Uniform access time		
Communication mechanism		

Multiprocessor Memory Models

	Symmetric	Distributed
Shared	•UMA •Ld/St •HW coherency •Snooping protocol	•NUMA •Ld/St •HW coherency •Directory protocol
Private		•NUMA •Messages •SW coherency

Distributed Memory Systems

- Distributed shared memory vs distributed private memory
- Shared
  - HW is responsible for coherency
  - HW is complicated
  - SW uses normal load and store instructions to communicate
- Private
  - SW is responsible for coherency
  - HW is simple
  - SW uses messages and remote procedure calls to communicate
- ***Q:*** *Why shared = hw-coherence and private = sw-coherence?*

HW vs SW for Coherence

- HW has ***dynamic*** view of program, SW has ***static*** view
- HW view
  - small “horizon” (can see only a few instructions at a time)
  - fine detail (e.g. know physical addresses)
- SW view
  - large “horizon” (can see many instructions at a time)
  - coarse detail (e.g. don't know physical addresses)
- To guarantee coherence, SW must treat any piece of data that might be shared as if it were shared. HW knows at each instant whether a piece of data is shared.
- SW must behave conservatively. For example, generate invalidate messages that might not be needed.

Try SW-Coherence for Shared Memory

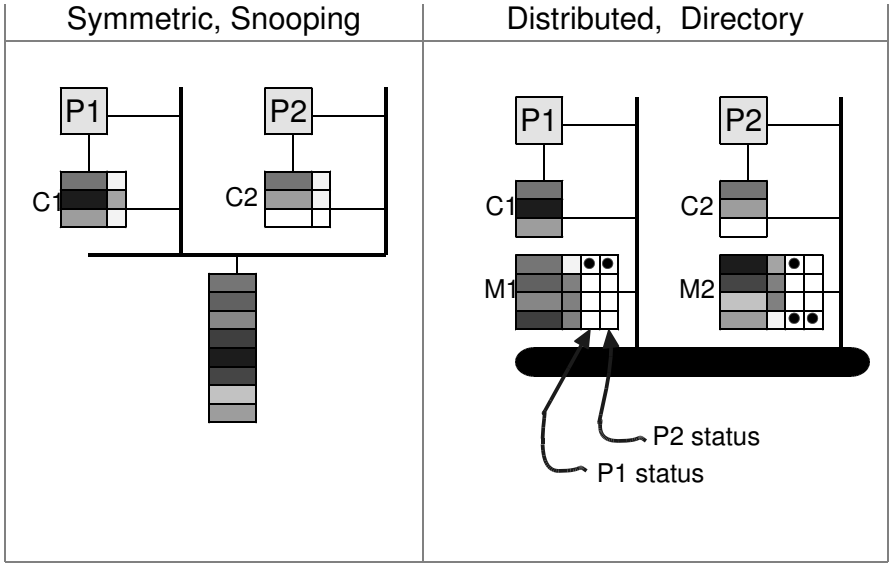
- Caches introduce coherency problems
- Option 1: remove caches
  - Exactly one copy of each memory block
  - No coherency problems! No performance!
- Option 2: only private data may be cached
  - Same effect as option 1
- Option 3: software controls cache
  - 3a: compiler generates cache control code
    - Feasible only for specialized applications (e.g. arrays)
  - 3b: programmer controls cache
    - For efficiency, want to move blocks of memory
    - Normal ISAs provide load/store for at most double words

Summary of HW vs SW

- Shared vs Private
  - Shared has simple programming model
- Symmetric vs Distributed
  - Symmetric uses snooping protocol, which doesn't scale
  - SW coherency is difficult
- Goal: Find HW-coherency protocol for distributed shared memory
  - Simple programming model
  - Scales to 64 – 256 processors
- Answer: ***Directory protocol***
  - HW-based coherency protocol for distributed memory

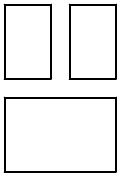
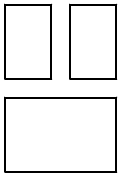
Coherence Protocols: Snoop vs Directory

- Two major categories of memory coherence protocols, distinguished by location of status information for block of physical memory



Coherence Protocols: Snoop vs Directory

- Two major categories of memory coherence protocols, distinguished by location of status information for block of physical memory

	Snooping	Directory
Location of status		
Update status		
Shared mem type		

Coherency: Snooping Protocols

Maintaining Coherence on Write

- Applicable to both snooping and directory protocols
- Write invalidate
  - Before P_i writes to memory location A, all other processors must invalidate location A.
  - This guarantees that P_i has exclusive access to A
- Write update
  - When P_i writes to memory location A, it broadcasts the write to all other processors
- Write invalidate usually has better performance than write update
- Write invalidate much more popular than write update

Snooping Protocols

- Each cache keeps copy of ***its status*** for each memory block that it contains.
- Status:
  - Invalid: must refetch block on next access
  - Shared: block held by multiple caches, all are clean
  - Modified: local processor has modified block (other procs should be invalid, main mem out of date)
  - Exclusive: local processor is only proc that contains block (block is clean, main mem is up to date)
  - Owned: local processor owns block and must service all requests
- Options for status: MSI, MESI, MOSI, MOESI

Snooping Protocol Example

- H&P Fig 6.8: Write-through, write-invalidate, Status = {Valid, Invalid}

P1	C1	P2	C2	Bus	Mem[A]
1: Rd A	Miss A			$P_1 \leftarrow A$	α
		1: Rd A	Miss A	$P_2 \leftarrow A$	
	α				
			α		
2: Wr A $\leftarrow \beta$				A $\leftarrow \beta$	
			INV		β
	β				

Snooping Protocol Example

- Write-through, write-invalidate, Status = {Valid, Invalid}
- Illustrate competing writes

P1	C1	P2	C2	Bus	Mem[A]
1: Rd A	Miss A			$P_1 \leftarrow A$	α
		1: Rd A	Miss A	$P_2 \leftarrow A$	
	α				
			α_1		
2: Wr A $\leftarrow \beta$		2: Wr A $\leftarrow \gamma$		A $\leftarrow \beta$	
			INV		β
	β			A $\leftarrow \gamma$	
	INV		γ		γ

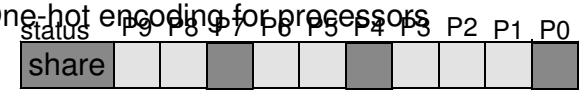
Advantages of Rich Status

- Basic protocol provides Invalid status
- ***Q:*** *Why add more status values?*
- Shared:
- Modified:
- Exclusive:
- Owned:

Coherency: Directory Protocols

Memory Block Status

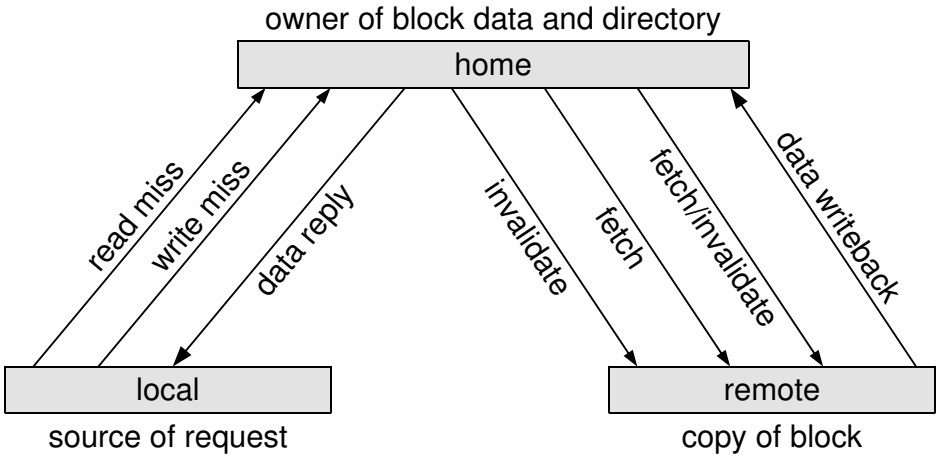
- Shared: memory block held by multiple caches, all are clean
  - Need to know ***which*** processors have memory block
- Uncached: no processor has copy of memory block
- Exclusive: exactly one processor has copy of memory block, processor might have written to block
  - Need to know ***which*** processor has memory block
  - Need to know if processor has ***written*** to block
- Implementation of status
  - A few bits for status
  - One-hot encoding for processors



Directory Protocol

- Two fundamental operations: read miss, write to non-exclusive block
- Read miss
- Write to non-exclusive block
  - Handle as write miss (e.g. read miss; write)
  - Wait until have exclusive access before changing memory
- Complication
  - Have complex interconnection network
  - No longer have a single bus
  - Snooping uses bus to provide consistent ordering of write operations

Directory Messages (Simple Version)



- Possible to have:
  - Local = home, Local = remote, or Remote = home

Directory Protocol Example

- Same scenario as first snooping protocol example in Lec-18

		M1		M2[A]		
P1	C1	Val Stat P1 P2	Net	Val Stat P1 P2	C2	P2C1
				α unc		
Rd A	Miss A		P_1 RdMiss A			
			$P_1 \leftarrow$ A:α	α exc ✔		
	A:α					
			P_2 RdMiss A		Miss A	Rd A
			$P_2 \leftarrow$ A:α	α sh ✔ ✔		
					A:α	

Directory Protocol Example

- P2 writes to A when A is shared

		M1		M2[A]		
P1	C1[A]	Val Stat P1 P2	Net	Val Stat P1 P2	C2[A]	P2C1
	A:α			α sh ✔ ✔	A:α	
			P_2 WrMiss A			Wr A $\leftarrow$ β
			Invalidate A	α unc		
	INV				INV	
			$P_2 \leftarrow$ A:α	α exc ✔		
					A:α	
					A:β	

Directory Protocol Example

- P1 and P2 try to write to A simultaneously

		M1			M2[A]				
P1	C1[A]	Val	Stat	P1 P2	Net at M2	Val	Stat	P1 P2	C2[A]	P2C1
						α	unc			
Wr A ← γ				P1 WrMiss A	P2 WrMiss A					Wr A ← β
				P1 WrMiss A						
				P1 WrMiss A	P2 ← A:α	α	exc	✔		
	INV				P1 WrMiss A				A:α	
					FetInv P1 A				A:β	
					A ← β				INV	
	A:β				P1 ← A:β	β	exc	✔		
	A:γ									

Performance Example

- H&P 6.12: Use distributed mem with bus communication
- 64B cache blocks, negligible I-Cache misses
- Total D-Cache miss rate 2%
- % misses to private data 60%
- % shared-data-misses that are uncached 20%
- % shared-data-misses that are dirty 80%
- Local memory hit: 100 ns
- Remote access to uncached data: 1800ns
- Remote access to dirty data: 2200ns
- 1GHz clock, CPI w/ 100% cache hit = 1.0, 40% instrs ld or st
- Find real CPI

Conistency

Quagmire of Consistency

- **P1**

```
A ← 0;
...
A ← 1;
if (B == 0) {
    print "P2 is slow"
}
```

- **P2**

```
B ← 0;
...
B ← 1;
if (A == 0) {
    print "P1 is slow"
}
```

- Q: possible for both P1 and P2 to think that the other is slow?

- Issue is ***consistency***: relative order of accesses to A and B by a single processor.

Tradeoffs in Consistency

- There are many definitions of consistency
- Strict (strong) definitions: make everything sequential
  - easy to understand
  - less parallelism, hurts performance
- Loose (weak) definitions: lots of parallelism and out-of-orderedness
  - hard to understand
  - improves performance

Sequential Consistency

- Overal order of operations is consistent with the order seen by each individual processor
- Example:
  - P1: **a; b; c; d; e; f;**
  - P2: **m; n; o; p; q;**
  - OK: **a; b; c; d; e; f; m; n; o; p; q;**
  - OK: **a; m; n; b; o; c; p; d; q; e; f;**
  - **BAD: a; m; n; b; p; c; o; d; q; e; f;**
- Sequential consistency is incompatible with loads-passing-stores in out-of-order execution.
- Idea: weak ordering models of consistency

Weak Orderings

- Idea:
  - allow operations to different addresses to complete out of order
  - use synchronization where need to enforce order
- Relaxation modes:
  - total-store ordering (relax RAW order)
  - partial-store ordering (relax WAW order)
  - weak ordering (relax WAR and RAR order)
    Alpha ordering
    PowerPC ordering
- Programming with relaxed ordering is very stressful

Speculation with Sequential Ordering

- An alternative to weak ordering
- Speculate that performing memory operations out-of-order operations will produce the same result as preserving sequential order
- **P1**

  A ← 0;

  A ← 1;

  Ld R3 **B**

  BNZ R3 end

  print “P2 is slow”

  end:

- **P2**

  B ← 0;

  B ← 1;

  Ld R3 **A**

  BNZ R3 end

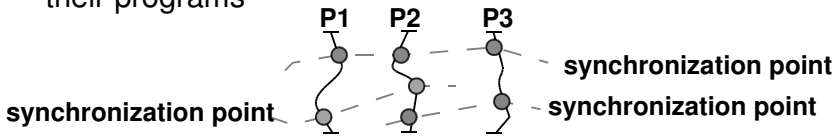
  print “P1 is slow”

  end:

Synchronization

Synchronization

- Basic idea: force multiples processors to stop at particular points in their programs



- Purpose of synchronization in hardware:
  Provide basic building blocks for messages and critical sections in software.
- Requirements on primitive operation for synchronization:
  - Read a memory location
  - Change the value of the memory location
  - Either guarantee that, or test if, (read;change) is atomic

Challenges

- Correctness (Solution to bugs)
  - Hardware provides simple, fast primitives
  - Low-level programmers build libraries on top of ISA-specific primitives that provide high-level constructs to programmers
- Performance
  - When no contention: minimize latency
  - When high contention: minimize communication

Historical Approaches

- Hardware provides single instruction that is guaranteed to be atomic
  - test-and-set *reg addr*
    - reg ← M[addr]
    - M[addr] ← #1
  - fetch-and-increment *reg addr*
    - reg ← M[addr]
    - M[addr] ← M[addr] + 1
  - atomic exchange *reg addr*
    - tmp ← M[addr]
    - M[addr] ← reg
    - reg ← tmp

Historical Approaches

- Disadvantage: requires that a read-write sequence is atomic
  - Memory coherence requires that no other memory operations occur between the read and the write
  - Potential problems with deadlock

		M1				M2[A]					
P1	C1	Val	Stat	P1	P2	Net	Val	Stat	P1	P2	C2	P2
R1 = sqrt												
Ld R3 (R1)												
Ex R3 A	Miss A					P_1 WrMiss A						
			P_2 RdMiss (R1)									

Q: *what's the problem?*

Modern Approaches

- Don't ***guarantee*** atomicity of (read; write)
- Instead: provide two operations
  - read
  - write and ***test*** if (read;write) was atomic
  - If test returns FALSE,
    then (read;write) was not atomic so try (read;write) again

Load-Link + Store-Conditional (Idea)

- Load-linked: load mem value into reg and special link register
  - LL *reg addr*

    reg ← M[*addr*]

    link ← *reg*
- Store conditional:
  if mem value = link-reg then store reg value in memory else fail
  - SC *reg addr*

    if M[*addr*] = link {
      M[*addr*] ← *reg*
    } else {
      reg ← #0
    }

Q: *what's the problem with SC?*

Use of Load-Link

- Assembly code to implement atomic exchange with load-link
- The code is the equivalent of:
  atomic-exchange R4 (R1)
- Assembly code

```
try:  R3 ←  R4          --  R3 holds working copy of R4
      LL    R2   (R1)   --  R2 ← M[R1]
      SC    R3   (R1)   --  M[R1] ← R3 (if succeed)
      BEQZ  R3   try    --  jump to try if failed
      R4 ←  R2          --  R4 ← M[R1]
```

- ***Q:*** *why do we need R3, rather than using just R4?*

Load-Link + Store-Conditional (Reality)

- Load-linked: load mem value into reg
  and copy ***address*** into special link register
  - LL *reg addr*

    reg ← M[*addr*]

    link ← *addr*
- Store conditional:
  if ***addr*** = link-reg then store reg value in memory, else fail
  - SC *reg addr*

    if *addr* = link {
      M[*addr*] ← *reg*
    } else {
      reg ← #0
    }

Atomicity of LL+SC

- Link register is special
- Processor must reset link register if the memory location that the link register points to might have been modified

status of M[A] = ___

P1	Link1	C1[A]	Net	M[A]	C2[A]	Link2	P2
				α			
LL R2 A			P1 RdMiss A				
	A	α	P1 ← A:α				
			P2 WrMiss A				ST A β
	0	INV	Inv P_1 A				
SC R3 A			P_2 ← A:α		α		
					β		

R3 = ___

status of M[A] = ___

Spin-Lock

- Locks are used to guarantee mutual exclusion
- With spin-lock, process spins in tight loop until can acquire lock
- Pseudo code for use of a spin-lock in mutual exclusion:

```
while ( locked = #1 ) {};        -- loop waiting for locked = #0
-- know that locked = #0
locked ← #1;                     -- set locked to prevent others
--
-- critical section
--
locked ← #0;                     -- release locked
```


Spin-Lock with LL+SC

```
-------------------------------------------------
loop: LL      R2    locked   -- loop waiting for locked = #0
      BNEZ    R2    loop
-------------------------------------------------
      R2    ← #1               -- set locked to prevent others
      SC      R2    locked
-------------------------------------------------
      BEQZ    R2    loop     -- jump back to top if store failed
-------------------------------------------------
      --
      -- critical section
      --
-------------------------------------------------
      ST      locked #0        -- release locked
```

Q: *what would happen if replaced SC with ST?*

Caches and Spin-Lock with LL+SC

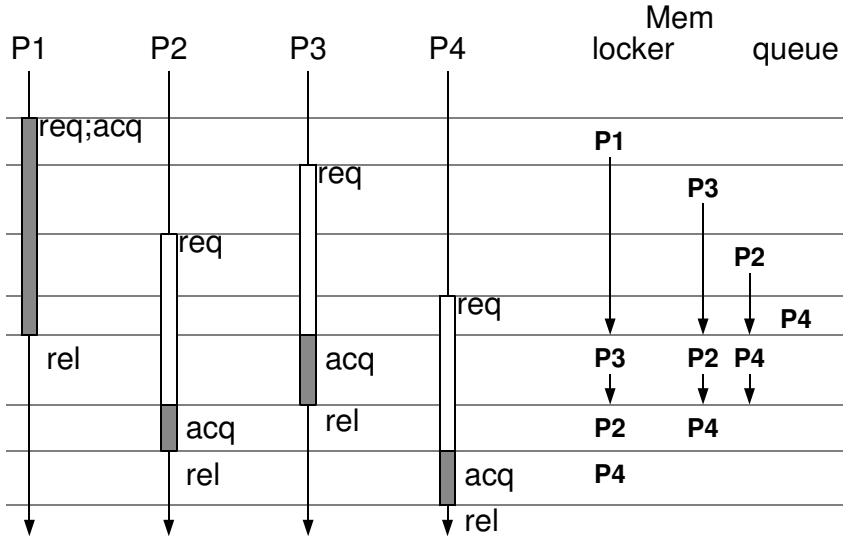
P1	Link1	C1[A]	Net	M[A]	C2[A]	Link2	P2
				1			
LL R2 A			P1 RdMiss A				
	A	**1**	P1 ← A:1				
LL R2 A							
LL R2 A							
LL R2 A			P2 WrMiss A				ST A **0**
LL R2 A	0	INV	Inv P_1 A				
P1 RdMiss A			P_2 ← A:1		**1**		
P1 RdMiss A					**0**		
P1 RdMiss A			P1 RdMiss A				

Performance Analysis

- Bus and memory traffic low while spinning (good)
- Lots of communication when transfer lock (bad)
  - Must invalidate all copies of lock when released (ST)
  - Must invalidate all copies of lock when acquired (SC)
- Observation: only one processor will acquire lock
- For losing processor:
  - locked=0
  - INV
  - RdMiss
  - locked=0
- Idea: Pick winning processor, don't talk to losers (Queuing lock)

Queuing Lock

- Augment memory directory with queue of processors waiting for lock



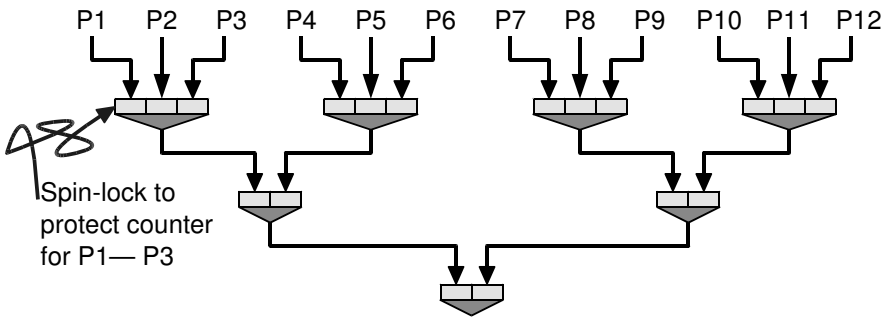
Exponential Backoff

- Another technique to reduce memory/bus traffic with spin locks
- Each time that fail to aquire lock, wait twice as long before re-trying
- Implement in low-level software library, no change to hardware

Barrier Synchronization

- Barrier: point in a program where all processors must synchronize
- Force all processors to wait at barrier
- Release all processors after last has arrived at barrier
- Implement barrier with two spin-locks
  - protect critical section that counts arrivals
  - processors spin-lock at barrier waiting for barrier to be unlocked
- Performance problem with many processors
  - Queuing lock: forces serialization of arrival counter
- Idea: Use ***combining-tree*** of locks, rather than a single lock

Combining Tree



Can also use combining tree for spin-lock to allow processors past barrier.
Q: *when beneficial?*