课程编号180086085404P2008H 2024-2025学年秋季学期

计算机体系结构

第 8 讲 缓存一致性、内存一致性

主讲教师： 刘珂

2024年10月29日

Coherent Protocols

Multi-thread programming models

Cache Coherence

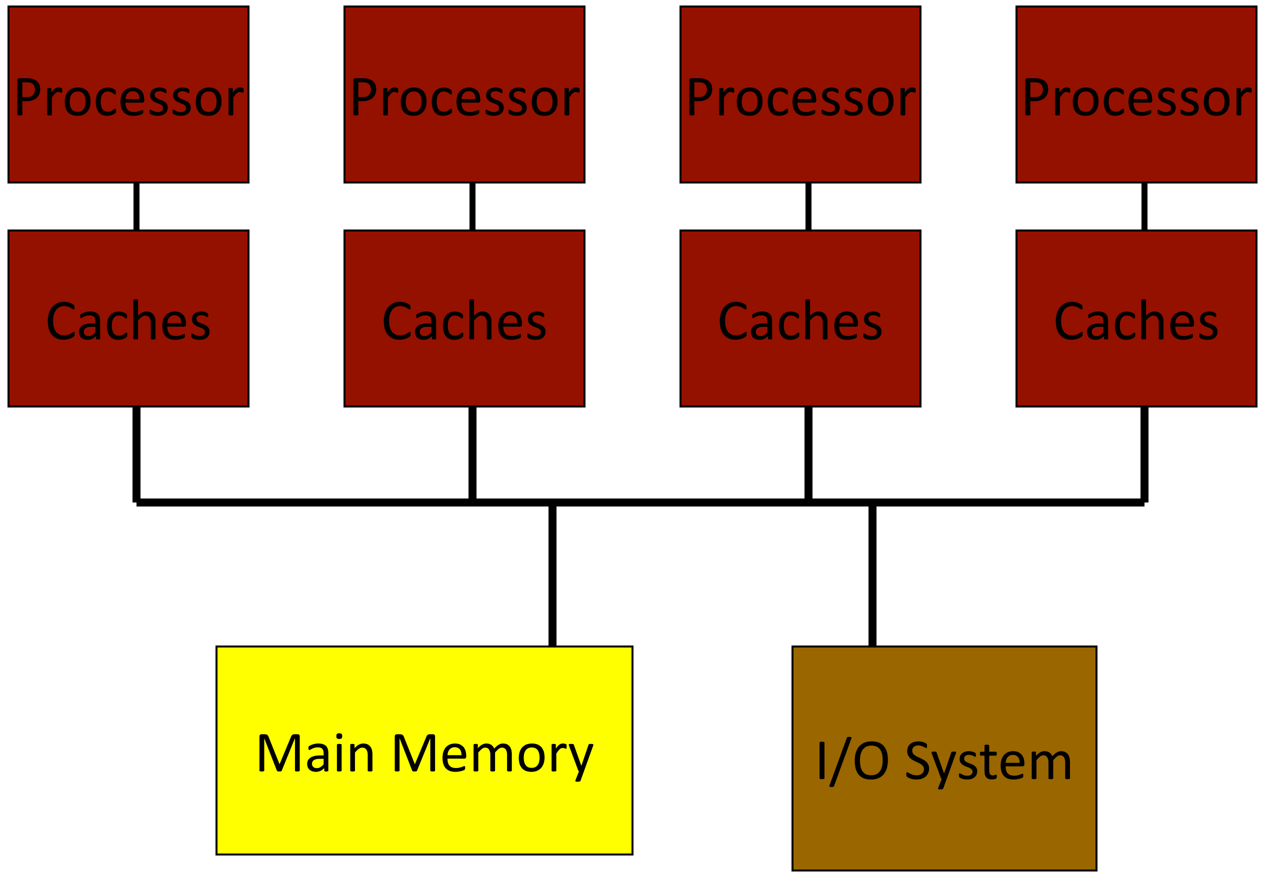
Memory Consistence

Multi-processors Memory Organization

Multiprocs -- Memory Organization - I

- Centralized shared-memory multiprocessor or Symmetric shared-memory multiprocessor (SMP)
- Multiple processors connected to a single centralized memory – since all processors see the same memory organization -> uniform memory access (UMA)
- Shared-memory because all processors can access the entire memory address space
- Can centralized memory emerge as a bandwidth bottleneck? – not if you have large caches and employ fewer than a dozen processors

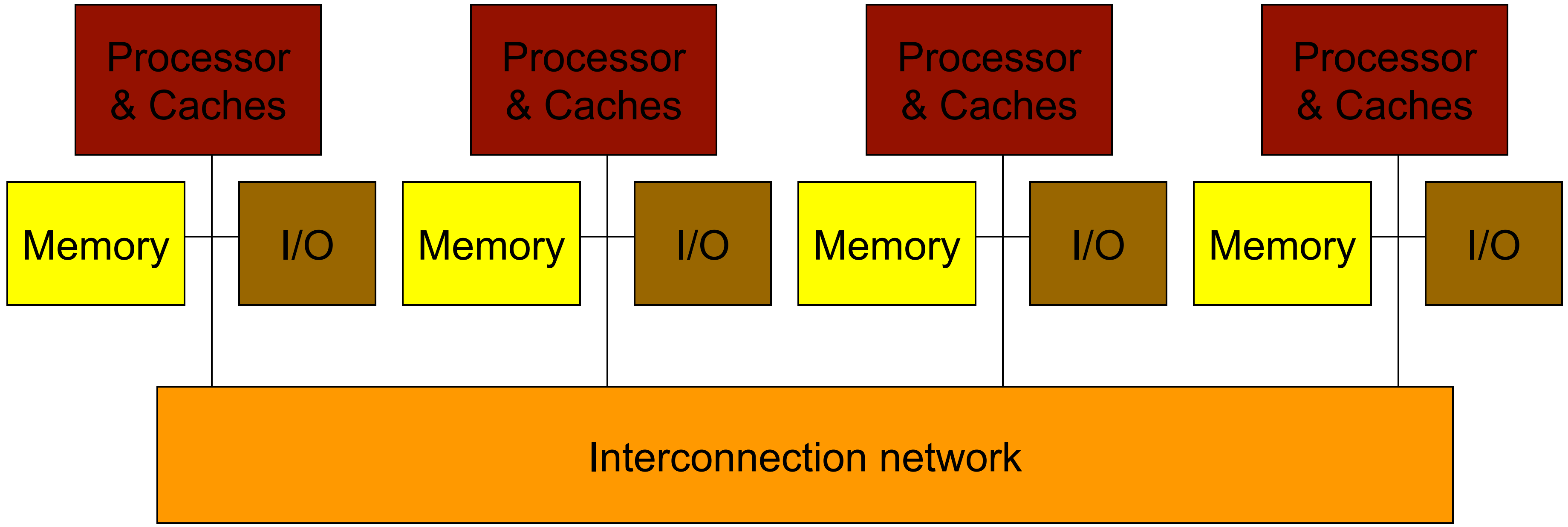
SMPs or Centralized Shared-Memory



Multiprocs -- Memory Organization - II

- For higher scalability, memory is distributed among processors -> distributed memory multiprocessors
- If one processor can directly address the memory local to another processor, the address space is shared -> distributed shared-memory (DSM) multiprocessor
- If memories are strictly local, we need messages to communicate data -> cluster of computers or multicomputers
- Non-uniform memory architecture (NUMA) since local memory has lower latency than remote memory

Distributed Memory Multiprocessors



Cache Coherence Protocols

Shared-Memory

Shared-memory:

- Well-understood programming model
- Communication is implicit and hardware handles protection
- Hardware-controlled caching

Message-passing:

SMPs

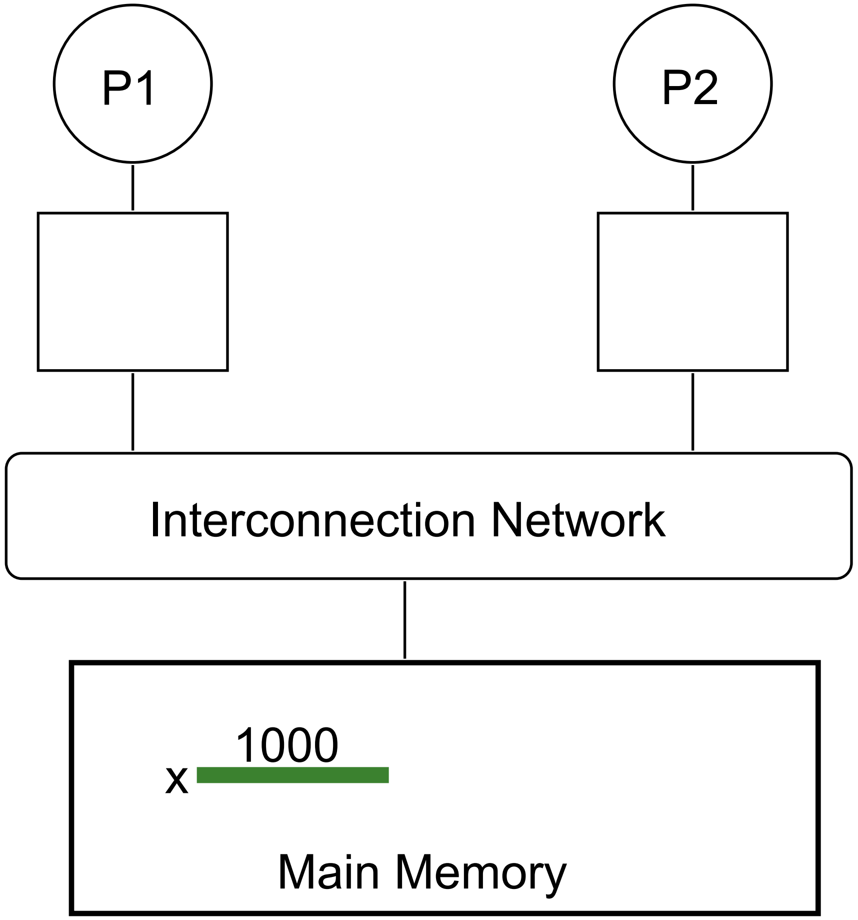
- Centralized main memory and many caches -> many copies of the same data

- A system is cache coherent if a read returns the most recently written value for that word

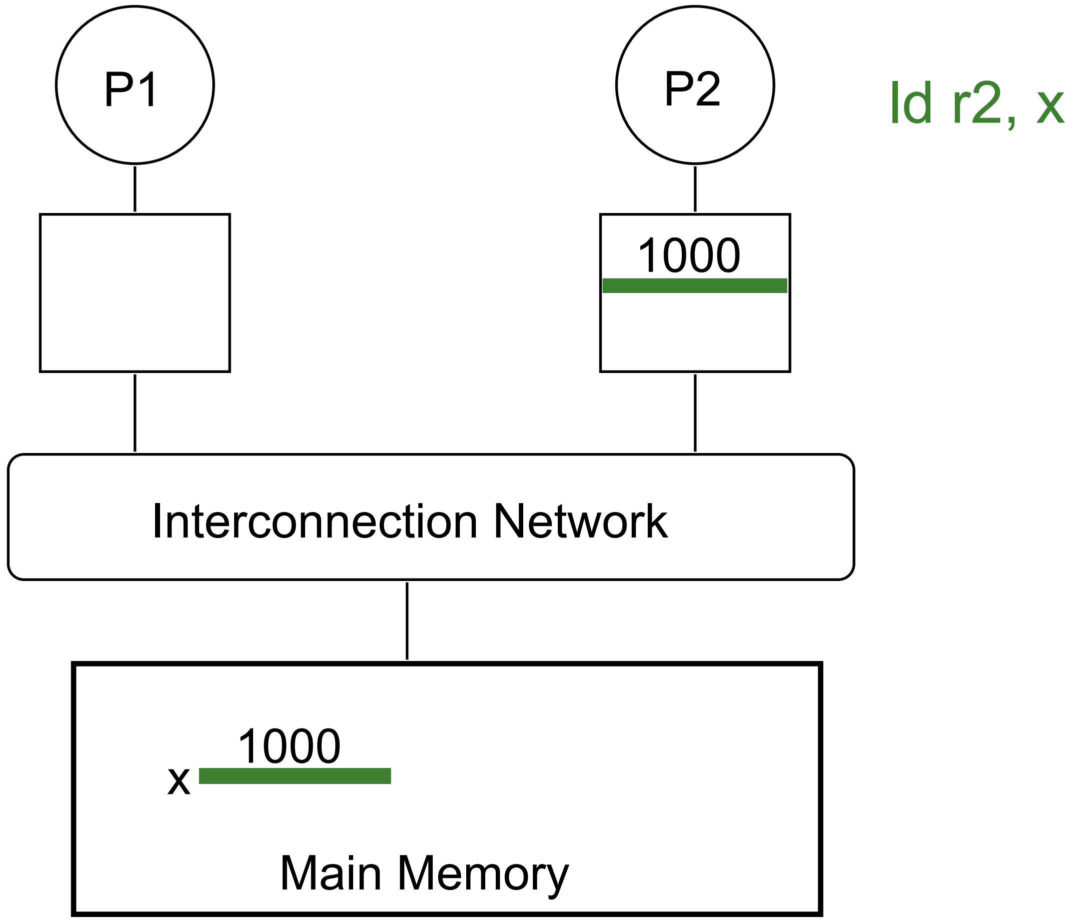
Time	Event	Value of X in Cache-A	Cache-B	Memory
0		-	-	1
1	CPU-A reads X	1	-	1
2	CPU-B reads X	1	1	1
3	CPU-A stores 0 in X	0	1	0

Cache Coherence

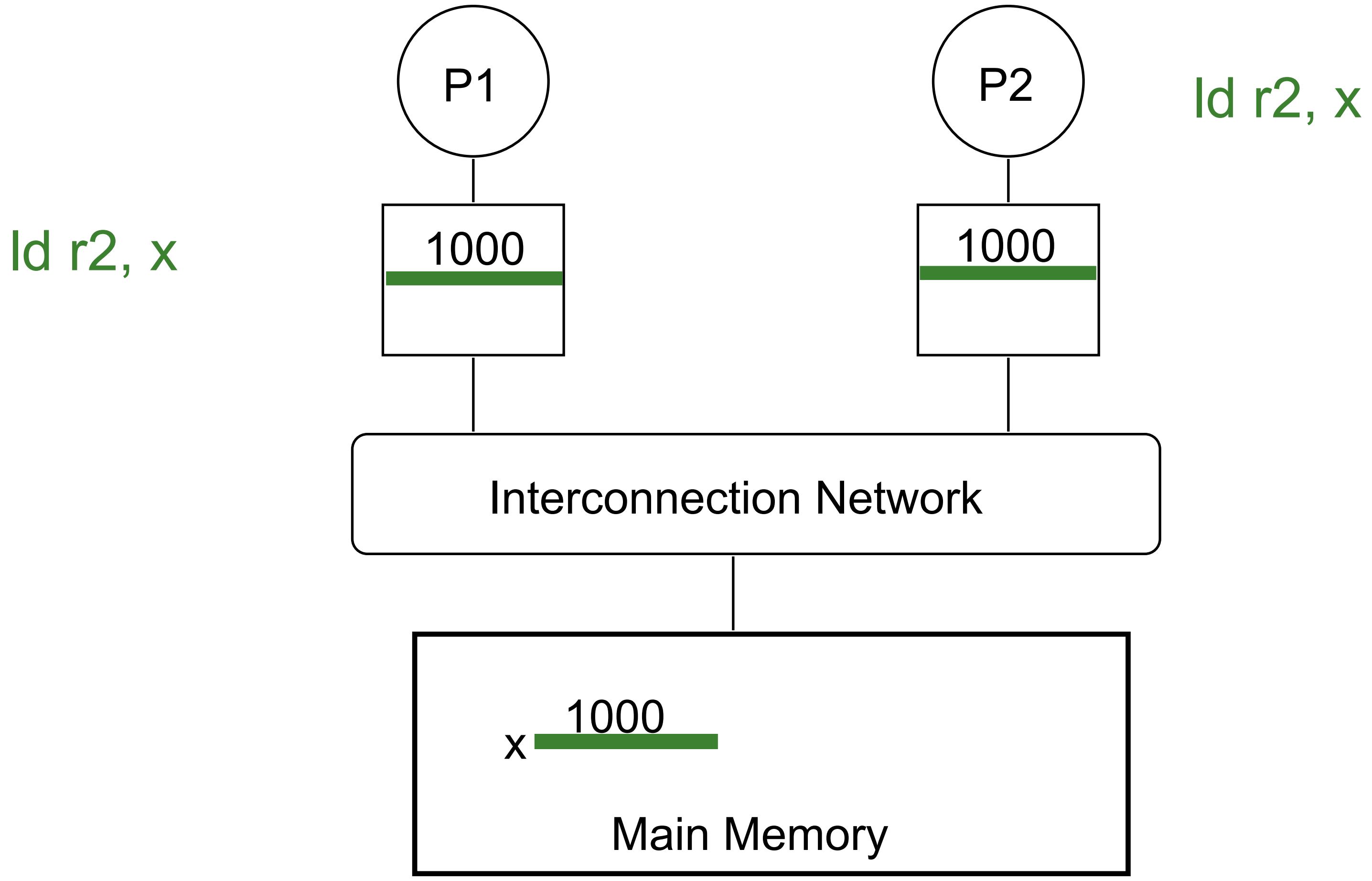
- Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?



The Cache Coherence Problem



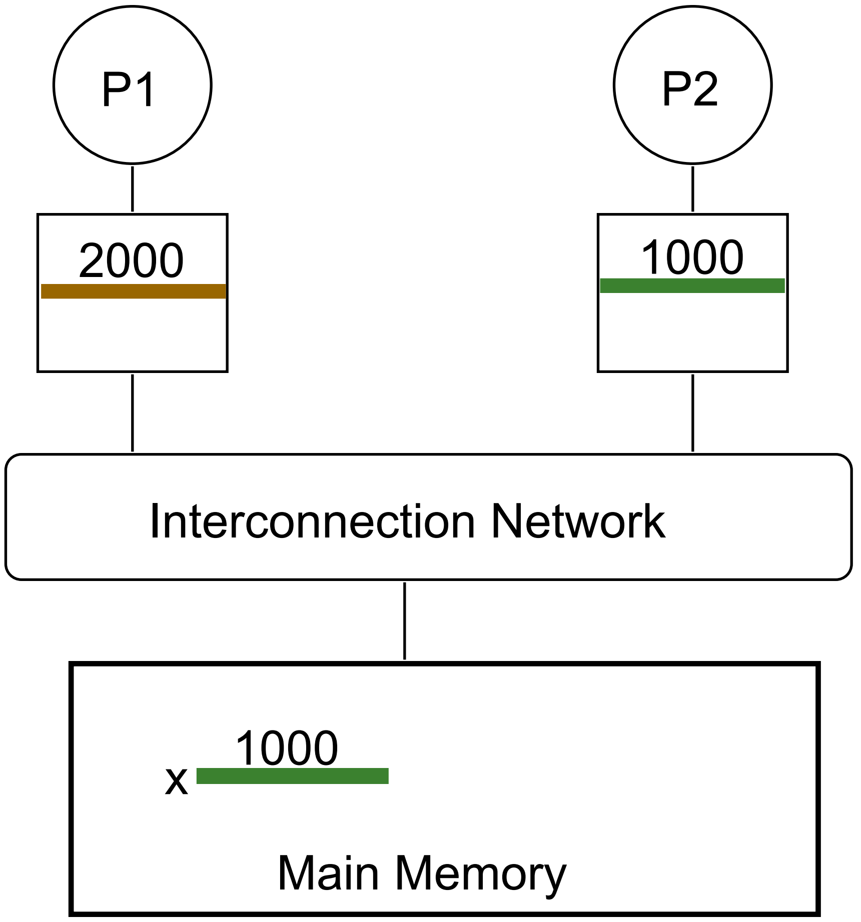
The Cache Coherence Problem



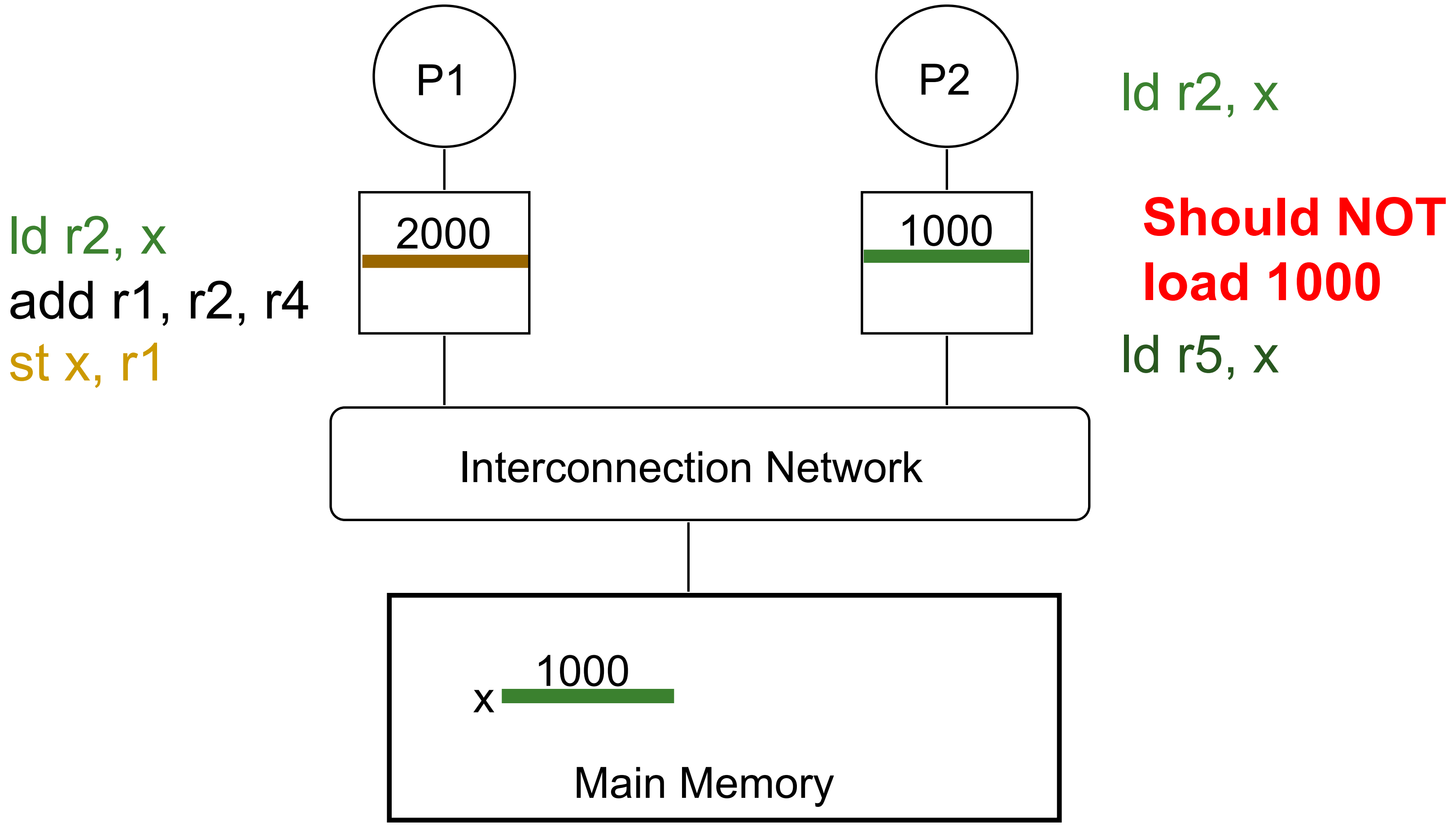
The Cache Coherence Problem

ld r2, x
add r1, r2, r4
st x, r1

ld r2, x



The Cache Coherence Problem



Cache Coherence

- A memory system is coherent if:
  - Write propagation: P1 writes to X, sufficient time elapses, P2 reads X and gets the value written by P1
  - Write serialization: Two writes to the same location by two processors are seen in the same order by all processors
  - The memory *consistency* model defines "time elapsed" before the effect of a processor is seen by others and the ordering with R/W to other locations (loosely speaking – more later)

Snoop-based Cache Coherence Protocol

Cache Coherence Protocols

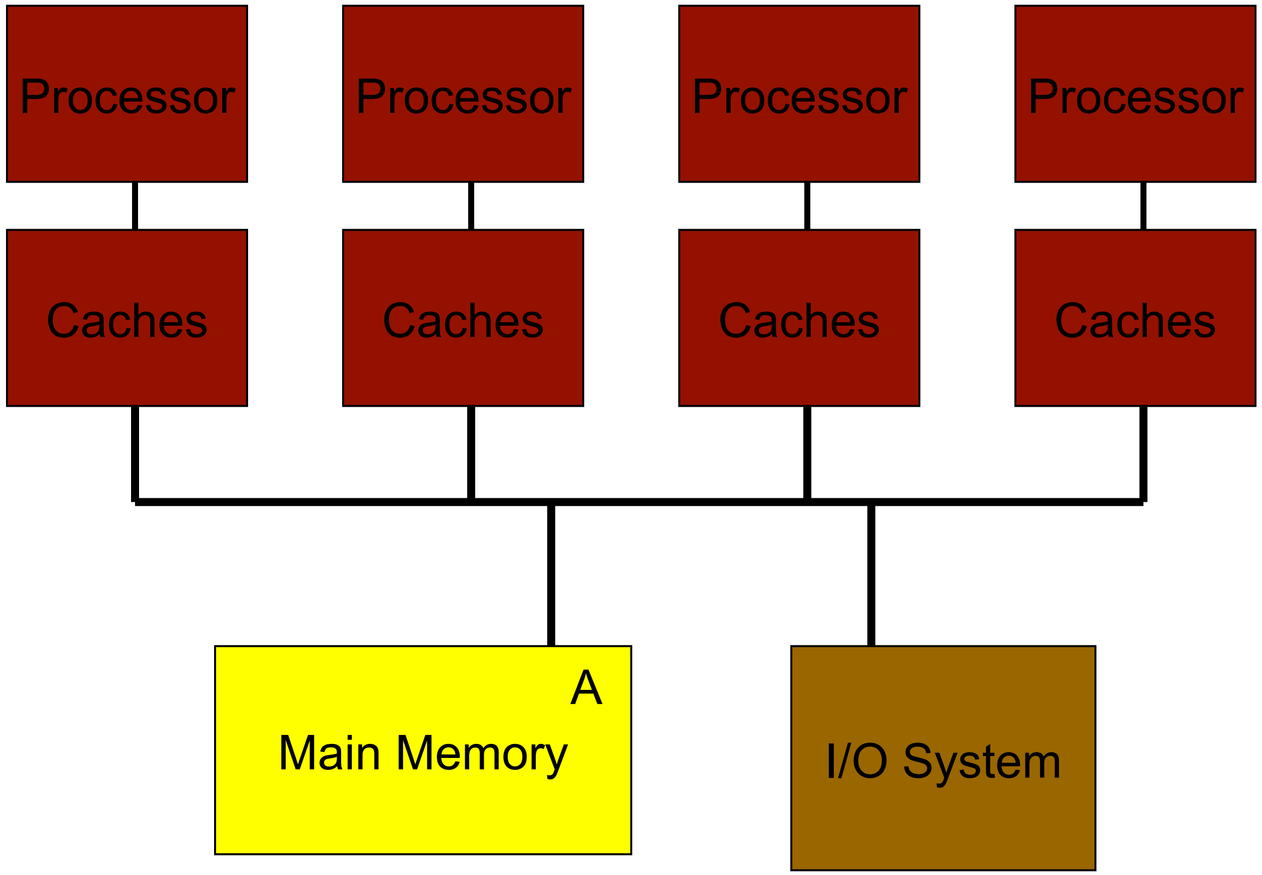
- Directory-based: A single location (directory) keeps track of the sharing status of a block of memory

- Snooping: Every cache block is accompanied by the sharing status of that block – all cache controllers monitor the shared bus so they can update the sharing status of the block, if necessary

  - Write-invalidate: a processor gains exclusive access of a block before writing by invalidating all other copies
  - Write-update: when a processor writes, it updates other shared copies of that block

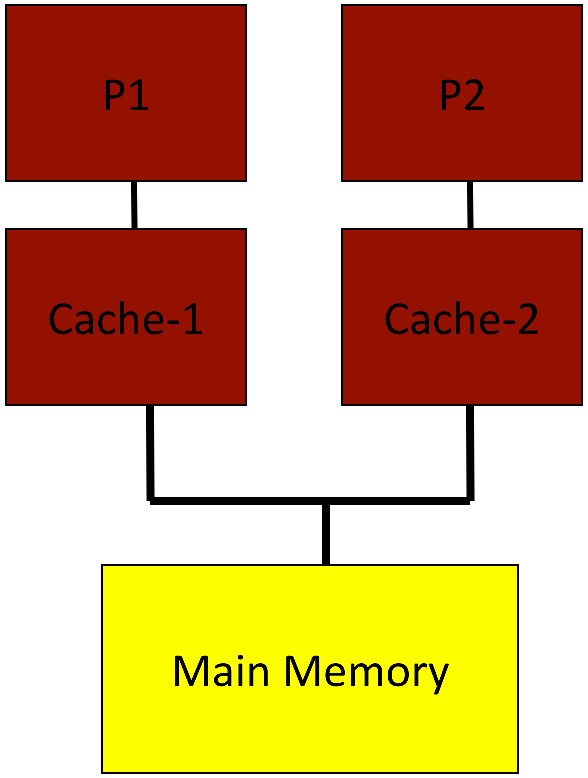
SMPs or Centralized Shared-Memory

P1: Rd A
P4: Rd A
P3: Wr A
P1: Rd A
P1: Wr A
P4: Wr A



SMP Example

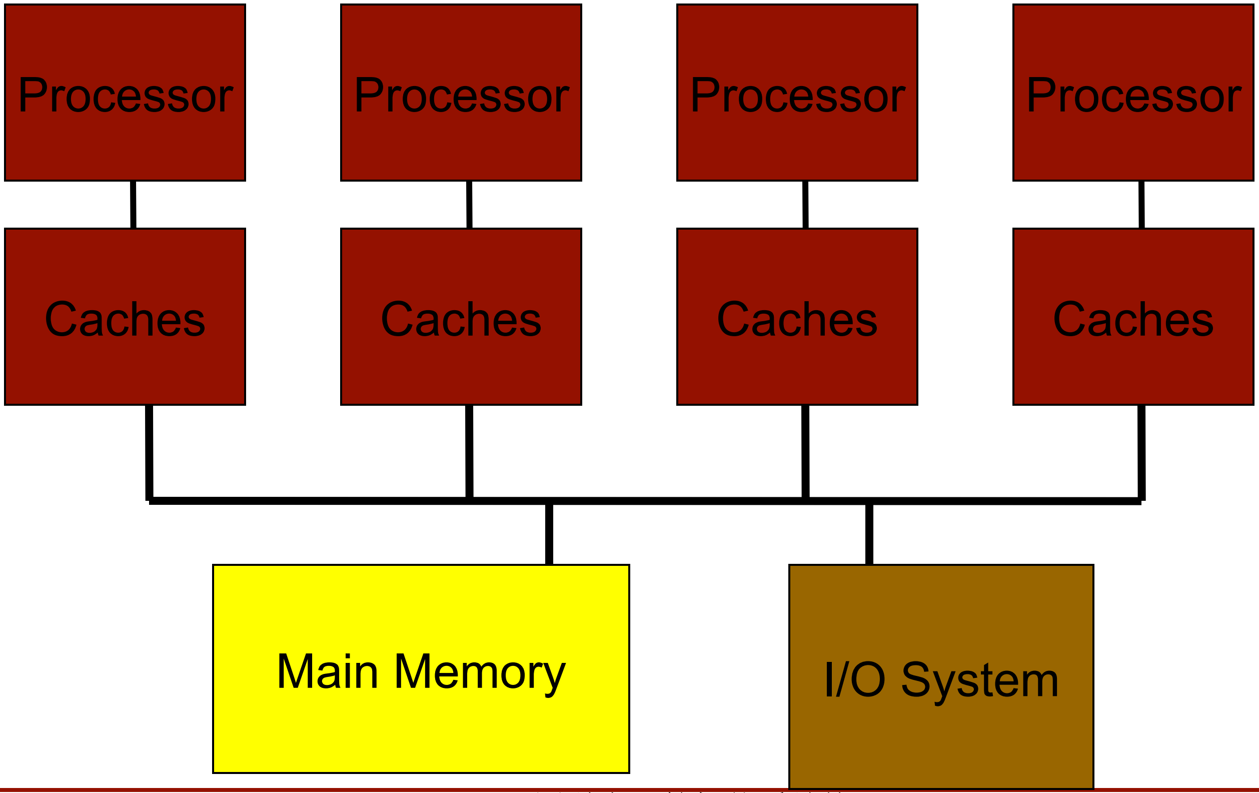
- P1 reads X: not found in cache-1, request sent on bus, memory responds, X is placed in cache-1 in shared state
- P2 reads X: not found in cache-2, request sent on bus, everyone snoops this request, cache-1 does nothing because this is just a read request, memory responds, X is placed in cache-2 in shared state



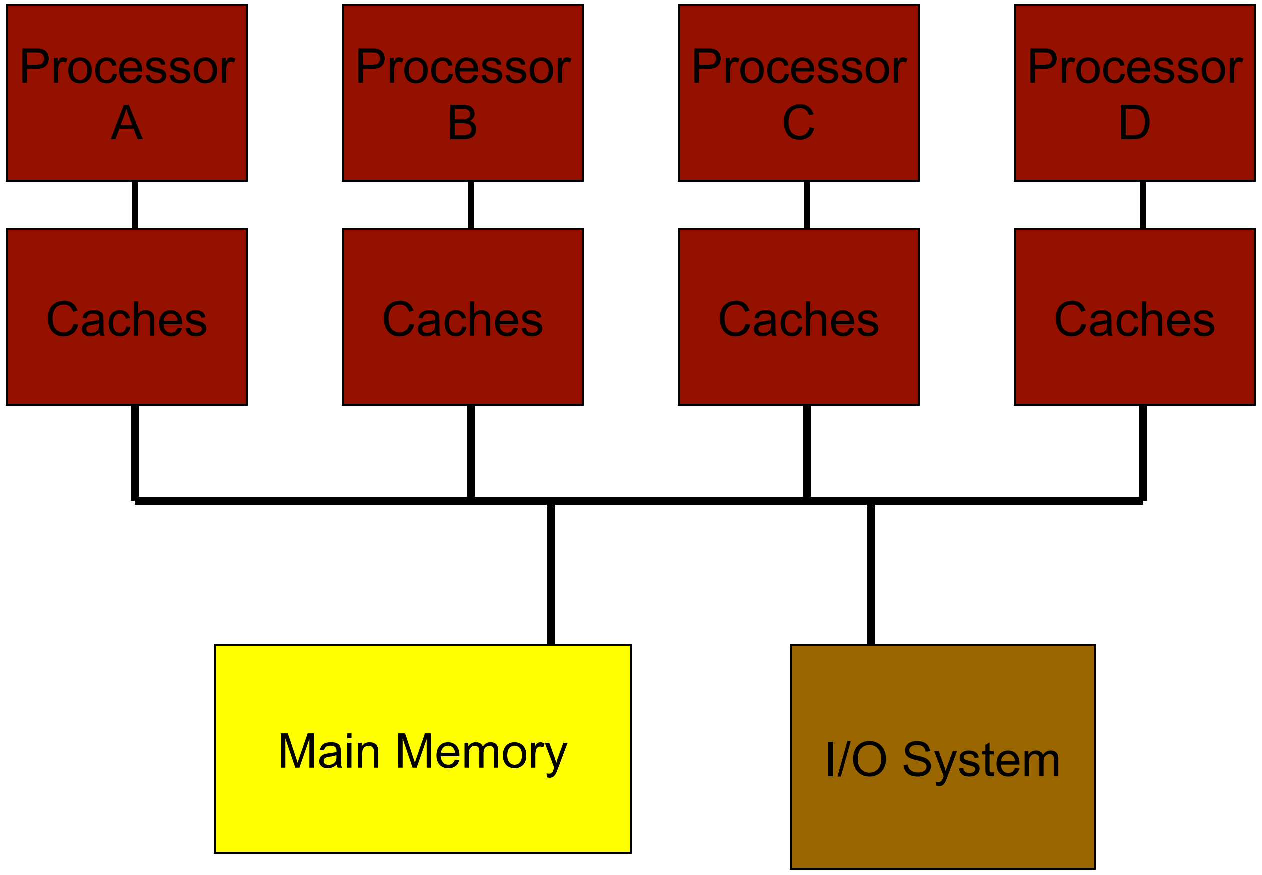
- P1 writes X: cache-1 has data in shared state (shared only provides read perms), request sent on bus, cache-2 snoops and then invalidates its copy of X, cache-1 moves its state to modified
- P2 reads X: cache-2 has data in invalid state, request sent on bus, cache-1 snoops and realizes it has the only valid copy, so it downgrades itself to shared state and responds with data, X is placed in cache-2 in shared state, memory is also updated

Design Issues

- Invalidate
- Find data
- Writeback/writethrough
- Cache block states
- Contention for tags
- Enforcing write serialization



SMP Example



A: Rd X
B: Rd X
C: Rd X
A: Wr X
A: Wr X
C: Wr X
B: Rd X
A: Rd X
A: Rd Y
B: Wr X
B: Rd Y
B: Wr X
B: Wr Y

SMP Example

	A	B	C	
A: Rd X	S			Rd-miss req; mem responds
B: Rd X	S	S		Rd-miss req; mem responds
C: Rd X	S	S	S	Rd-miss req; mem responds
A: Wr X	M	I	I	Upgrade req; no resp; others inv
A: Wr X	M	I	I	Cache hit
C: Wr X	I	I	M	Wr-miss req; A resp & inv; no wrtbk
B: Rd X	I	S	S	Rd-miss req; C resp; wrtbk to mem
A: Rd X	S	S	S	Rd-miss req; mem responds
A: Rd Y	S (Y)	S (X)	S (X)	Rd-miss req; X evicted; mem resp
B: Wr X	S (Y)	M (X)	I	Upgrade req; no resp; others inv
B: Rd Y	S (Y)	S (Y)	I	Rd-miss req; mem resp; X wrtbk
B: Wr X	S (Y)	M (X)	I	Wr-miss req; mem resp; Y evicted
B: Wr Y	I	M (Y)	I	Wr-miss req; mem resp; others inv; X wrtbk

Example Protocol

Request	Source	Block state	Action
Read hit	Proc	Shared/excl	Read data in cache
Read miss	Proc	Invalid	Place read miss on bus
Read miss	Proc	Shared	Conflict miss: place read miss on bus
Read miss	Proc	Exclusive	Conflict miss: write back block, place read miss on bus
Write hit	Proc	Exclusive	Write data in cache
Write hit	Proc	Shared	Place write miss on bus
Write miss	Proc	Invalid	Place write miss on bus
Write miss	Proc	Shared	Conflict miss: place write miss on bus
Write miss	Proc	Exclusive	Conflict miss: write back, place write miss on bus
Read miss	Bus	Shared	No action; allow memory to respond
Read miss	Bus	Exclusive	Place block on bus; change to shared
Write miss	Bus	Shared	Invalidate block
Write miss	Bus	Exclusive	Write back block; change to invalid

Directory-based Cache Coherence Protocols

Cache Coherence Protocols

- **Directory-based:** A single location (directory) keeps track of the sharing status of a block of memory

- Snooping: Every cache block is accompanied by the sharing status of that block – all cache controllers monitor the shared bus so they can update the sharing status of the block, if necessary

  - **Write-invalidate:** a processor gains exclusive access of a block before writing by invalidating all other copies
  - Write-update: when a processor writes, it updates other shared copies of that block

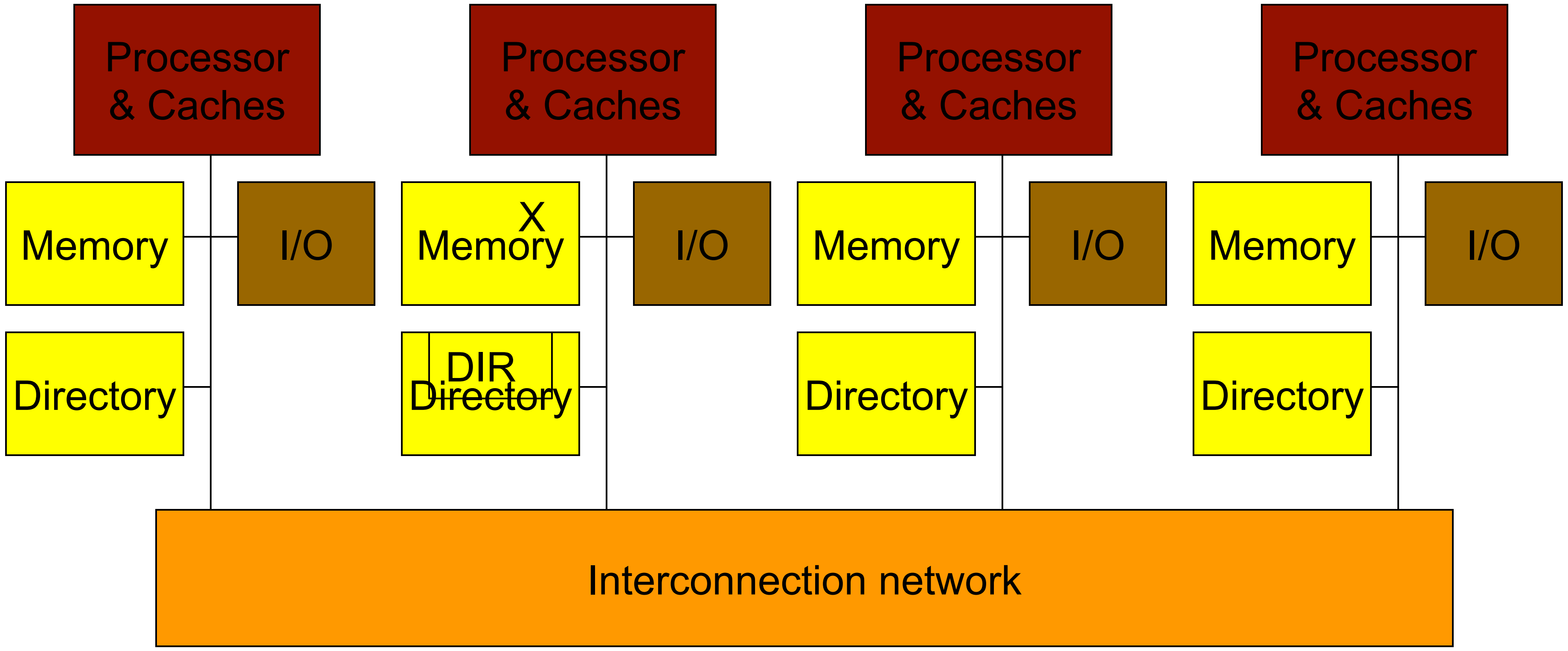
Directory-Based Cache Coherence

- The physical memory is distributed among all processors
- The directory is also distributed along with the corresponding memory
- The physical address is enough to determine the location of memory
- The (many) processing nodes are connected with a scalable interconnect (not a bus) – hence, messages are no longer broadcast, but routed from sender to receiver – since the processing nodes can no longer snoop, the directory keeps track of sharing state

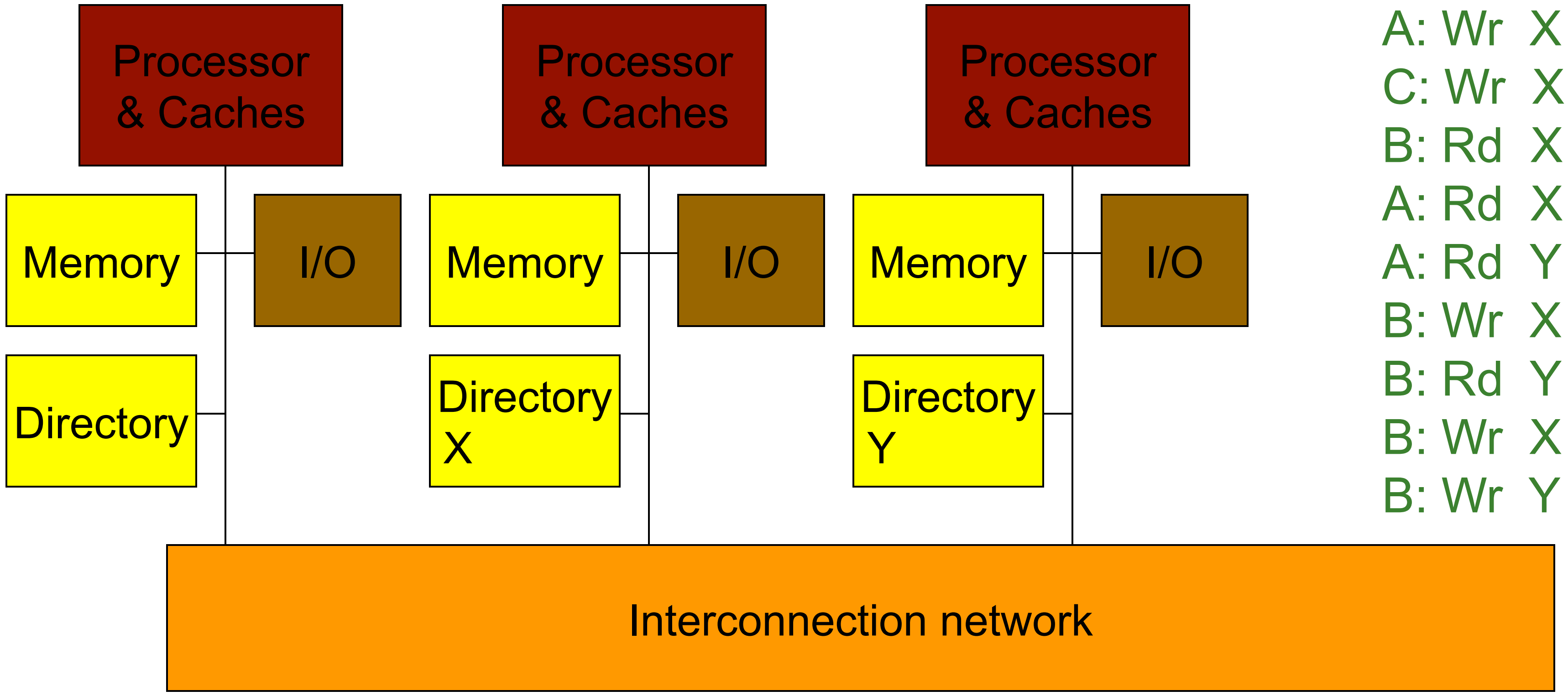
Distributed Memory Multiprocessors

- X – 1.5 G

N1: Rd X, N4: Rd X, N3: Wr X, N4: Rd X



Directory-based Example



A: Rd X
B: Rd X
C: Rd X
A: Wr X
A: Wr X
C: Wr X
B: Rd X
A: Rd X
A: Rd Y
B: Wr X
B: Rd Y
B: Wr X
B: Wr Y

Directory Example

	A	B	C	Dir	Comments
A: Rd X					
B: Rd X					
C: Rd X					
A: Wr X					
A: Wr X					
C: Wr X					
B: Rd X					
A: Rd X					
A: Rd Y					
B: Wr X					
B: Rd Y					
B: Wr X					
B: Wr Y					

Directory Example

	A	B	C	Dir	Comments
A: Rd X	S			S: A	Req to dir; data to A
B: Rd X	S	S		S: A, B	Req to dir; data to B
C: Rd X	S	S	S	S: A,B,C	Req to dir; data to C
A: Wr X	M	I	I	M: A	Req to dir;inv to B,C;dir recv ACKs;perms to A
A: Wr X	M	I	I	M: A	Cache hit
C: Wr X	I	I	M	M: C	Req to dir;fwd to A; sends data to dir; dir to C
B: Rd X	I	S	S	S: B, C	Req to dir;fwd to C;data to dir;dir to B; wrtbk
A: Rd X	S	S	S	S:A,B,C	Req to dir; data to A
A: Rd Y	S(Y)	S	S	X:S: A,B,C (Y:S:A)	Req to dir; data to A
B: Wr X	S(Y)	M	I	X:M:B	Req to dir; inv to A,C;dir recv ACK;perms to B
B: Rd Y	S(Y)	S(Y)	I	X: - Y:S:A,B	Req to dir; data to B; wrtbk of X
B: Wr X	S(Y)	M(X)	I	X:M:B Y:S:A,B	Req to dir; data to B
B: Wr Y	I	M(Y)	I	X: - Y:M:B	Req to dir;inv to A;dir recv ACK; perms and data to B;wrtbk of X

Cache Block States

- What are the different states a block of memory can have within the directory?
- Note that we need information for each cache so that invalidate messages can be sent
- The block state is also stored in the cache for efficiency
- The directory now serves as the arbitrator: if multiple write attempts happen simultaneously, the directory determines the ordering

Directory Actions

- If block is in uncached state:
  - Read miss: send data, make block shared
  - Write miss: send data, make block exclusive
- If block is in shared state:
  - Read miss: send data, add node to sharers list
  - Write miss: send data, invalidate sharers, make excl
- If block is in exclusive state:
  - Read miss: ask owner for data, write to memory, send data, make shared, add node to sharers list
  - Data write back: write to memory, make uncached
  - Write miss: ask owner for data, write to memory, send data, update identity of new owner, remain exclusive

Multi-Threading Programming Model

Shared-Memory Vs. Message-Passing

- Shared-memory:
  - Well-understood programming model
  - Communication is implicit and hardware handles protection
  - Hardware-controlled caching

- Message-passing:
  - No cache coherence -> simpler hardware
  - Explicit communication -> easier for the programmer to restructure code
  - Sender can initiate data transfer

Ocean Kernel

```
Procedure Solve(A)
begin
  diff = done = 0;
  while (!done) do
      diff = 0;
      for i ← 1 to n do
         for j ← 1 to n do
            temp = A[i,j];
            A[i,j] ← 0.2 * (A[i,j] + neighbors);
            diff += abs(A[i,j] – temp);
         end for
      end for
      if (diff < TOL) then done = 1;
  end while
end procedure
```

Shared Address Space Model

```
int  n, nprocs;
float  **A, diff;
LOCKDEC(diff_lock);
BARDEC(bar1);


main()
begin
   read(n); read(nprocs);
   A ← G_MALLOC();
   initialize (A);
   CREATE (nprocs,Solve,A);
   WAIT_FOR_END (nprocs);
end main
```

```
procedure Solve(A)
    int i, j, pid, done=0;
    float temp, mydiff=0;
    int mymin = 1 + (pid * n/procs);
    int mymax = mymin + n/nprocs -1;
    while (!done) do
       mydiff = diff = 0;
       BARRIER(bar1,nprocs);
       for i ← mymin to mymax
          for j ← 1 to n do
             …
          endfor
       endfor
       LOCK(diff_lock);
       diff += mydiff;
       UNLOCK(diff_lock);
       BARRIER (bar1, nprocs);
       if (diff < TOL) then done = 1;
       BARRIER (bar1, nprocs);
    endwhile
```

Message Passing Model

```
main()
   read(n); read(nprocs);
   CREATE (nprocs-1, Solve);
   Solve();
   WAIT_FOR_END (nprocs-1);

procedure Solve()
   int i, j, pid, nn = n/nprocs, done=0;
   float temp, tempdiff, mydiff = 0;
   myA ← malloc(…)
   initialize(myA);
   while (!done) do
       mydiff = 0;
       if (pid != 0)
         SEND(&myA[1,0], n, pid-1, ROW);
       if (pid != nprocs-1)
         SEND(&myA[nn,0], n, pid+1, ROW);
       if (pid != 0)
         RECEIVE(&myA[0,0], n, pid-1, ROW);
       if (pid != nprocs-1)
         RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
         for i ← 1 to nn do
            for j ← 1 to n do
               …
            endfor
         endfor
         if (pid != 0)
           SEND(mydiff, 1, 0, DIFF);
           RECEIVE(done, 1, 0, DONE);
         else
           for i ← 1 to nprocs-1 do
              RECEIVE(tempdiff, 1, *, DIFF);
              mydiff += tempdiff;
           endfor
           if  (mydiff < TOL)  done = 1;
           for i ← 1 to nprocs-1  do
              SEND(done, 1, I, DONE);
           endfor
         endif
      endwhile
```

The following slides will be covered in the next lectures

Synchronization

Constructing Locks

- Applications have phases (consisting of many instructions) that must be executed atomically, without other parallel processes modifying the data

- A lock surrounding the data/code ensures that only one program can be in a critical section at a time

- The hardware must provide some basic primitives that allow us to construct locks with different properties

- Lock algorithms assume an underlying cache coherence mechanism – when a process updates a lock, other processes will eventually see the update

Synchronization

- The simplest hardware primitive that greatly facilitates synchronization implementations (locks, barriers, etc.) is an atomic read-modify-write
- Atomic exchange: swap contents of register and memory
- Special case of atomic exchange: test & set: transfer memory location into register and write 1 into memory
- lock:

```
lock:    t&s   register, location
         bnz   register, lock
         CS
         st    location, #0
```

Caching Locks

- Spin lock: to acquire a lock, a process may enter an infinite loop that keeps attempting a read-modify till it succeeds

- If the lock is in memory, there is heavy bus traffic => other processes make little forward progress

- Locks can be cached:
  - Cache coherence ensures that a lock update is seen by other processors
  - The process that acquires the lock in exclusive state gets to update the lock first
  - Spin on a local copy – the external bus sees little traffic

Coherence Traffic for a Lock

- If every process spins on an exchange, every exchange instruction will attempt a write
  - many invalidates and the locked value keeps changing ownership
- Hence, each process keeps reading the lock value – a read does not generate coherence traffic and every process spins on its locally cached copy

- When the lock owner releases the lock by writing a 0, other copies are invalidated, each spinning process generates a read miss, acquires a new copy, sees the 0, attempts an exchange (requires acquiring the block in exclusive state so the write can happen), first process to acquire the block in exclusive state acquires the lock, others keep spinning

Test-and-Test-and-Set

```
lock:   test    register, location
        bnz     register, lock
        t&s     register, location
        bnz     register, lock
        CS
        st      location, #0
```

Load-Linked and Store Conditional

- LL-SC is an implementation of atomic read-modify-write with very high flexibility
- LL: read a value and update a table indicating you have read this address, then perform any amount of computation
- SC: attempt to store a result into the same memory location, the store will succeed only if the table indicates that no other process attempted a store since the local LL (success only if the operation was "effectively" atomic)
- SC implementations do not generate bus traffic if the SC fails – hence, more efficient than test&test&set

Spin Lock with Low Coherence Traffic

```
lockit:   LL       R2, 0(R1)    ; load linked, generates no coherence traffic
          BNEZ     R2, lockit   ; not available, keep spinning
          DADDUI R2, R0, #1 ; put value 1 in R2
          SC       R2, 0(R1)    ; store-conditional succeeds if no one
                                ; updated the lock since the last LL
          BEQZ     R2, lockit   ; confirm that SC succeeded, else keep trying
```

- If there are i processes waiting for the lock, how many bus transactions happen?

Spin Lock with Low Coherence Traffic

```
lockit:  LL      R2, 0(R1)   ; load linked, generates no coherence traffic
         BNEZ    R2, lockit   ; not available, keep spinning
         DADDUI R2, R0, #1 ; put value 1 in R2
         SC      R2, 0(R1)   ; store-conditional succeeds if no one
                             ; updated the lock since the last LL
        BEQZ    R2, lockit   ; confirm that SC succeeded, else keep trying
```

- If there are i processes waiting for the lock, how many bus transactions happen?
- 1 write by the releaser + i read-miss requests + i (or 1) responses + 1 write by acquirer + 0 (i-1 failed SCs) + i-1 read-miss requests + i-1 (or 1) responses
  - (The i and i-1 read misses and responses can be reduced to 1)

Further Reducing Bandwidth Needs

- Ticket lock: every arriving process atomically picks up a ticket and increments the ticket counter (with an LL-SC), the process then keeps checking the now-serving variable to see if its turn has arrived, after finishing its turn it increments the now-serving variable
- Array-Based lock: instead of using a "now-serving" variable, use a "now-serving" array and each process waits on a different variable – fair, low latency, low bandwidth, high scalability, but higher storage
- Queueing locks: the directory controller keeps track of the order in which requests arrived – when the lock is available, it is passed to the next in line (only one process sees the invalidate and update)

Memory Consistence Model

Coherence Vs. Consistency

- Recall that coherence guarantees (i) that a write will eventually be seen by other processors, and (ii) write serialization (all processors see writes to the same location in the same order)

- The consistency model defines the ordering of writes and reads to different memory locations – the hardware guarantees a certain consistency model and the programmer attempts to write correct programs with those assumptions

Example Programs

Initially, A = B = 0

```
  P1                      P2
A = 1                   B = 1
if (B == 0)             if (A == 0)
   critical section        critical section
```

Initially, Head = Data = 0

```
  P1                P2
Data = 2000     while (Head == 0)
Head = 1          { }
                ... = Data
```

Initially, A = B = 0

```
  P1         P2          P3
A = 1
           if (A == 1)
             B = 1
                       if (B == 1)
                         register = A
```

Sequential Consistency

P1	P2
Instr-a	Instr-A
Instr-b	Instr-B
Instr-c	Instr-C
Instr-d	Instr-D
…	…

We assume:

- **Within a program, program order is preserved**
- **Each instruction executes atomically**
- **Instructions from different threads can be interleaved arbitrarily**

Valid executions:

abAcBCDdeE… or ABCDEFabGc… or abcAdBe… or
aAbBcCdDeE… or …..

Sequential Consistency

- Programmers assume SC; makes it much easier to reason about program behavior

- Hardware innovations can disrupt the SC model

- For example, if we assume write buffers, or out-of-order execution, or if we drop ACKS in the coherence protocol, the previous programs yield unexpected outputs

Consistency Example - I

- An ooo core will see no dependence between instructions dealing with A and instructions dealing with B; those operations can therefore be re-ordered; this is fine for a single thread, but not for multiple threads

```
Initially A = B = 0
  P1                    P2
A ← 1                 B ← 1
 …                     …
if (B == 0)           if (A == 0)
  Crit.Section          Crit.Section
```

The consistency model lets the programmer know what assumptions they can make about the hardware's reordering capabilities

Consistency Example - 2

Initially, A = B = 0

```
  P1             P2                   P3
A = 1
            if (A == 1)
              B = 1
                                 if (B == 1)
                                   register = A
```

If a coherence invalidation didn't require ACKs, we can't confirm that everyone has seen the value of A.

Sequential Consistency

- A multiprocessor is sequentially consistent if the result of the execution is achievable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion

- Can implement sequential consistency by requiring the following: program order, write serialization, everyone has seen an update before a value is read – very intuitive for the programmer, but extremely slow

- This is very slow… alternatives:
  - Add optimizations to the hardware (e.g., verify loads)
  - Offer a relaxed memory consistency model and fences

Relaxed Consistency Models

- We want an intuitive programming model (such as sequential consistency) and we want high performance

- We care about data races and re-ordering constraints for some parts of the program and not for others – hence, we will relax some of the constraints for sequential consistency for most of the program, but enforce them for specific portions of the code

- Fence instructions are special instructions that require all previous memory accesses to complete before proceeding (sequential consistency)

Fences

```
P1                          P2
{                           {
  Region of code            Region of code
  with no races              with no races
}                           }

Fence                       Fence
Acquire_lock                Acquire_lock
Fence                       Fence

{                           {
   Racy code                Racy code
}                           }

Fence                       Fence
Release_lock                Release_lock
Fence                       Fence
```

Transaction Memory

Transactions

- New paradigm to simplify programming
  - instead of lock-unlock, use transaction begin-end
  - locks are blocking, transactions execute speculatively in the hope that there will be no conflicts
- Can yield better performance; Eliminates deadlocks
- Programmer can freely encapsulate code sections within transactions and not worry about the impact on performance and correctness (for the most part)
- Programmer specifies the code sections they'd like to see execute atomically – the hardware takes care of the rest (provides illusion of atomicity)

Example

- Producer-consumer relationships – producers place tasks at the tail of a work-queue and consumers pull tasks out of the head

```
Enqueue
  transaction begin
    if (tail == NULL)
      update head and tail
    else
      update tail
  transaction end
```

```
Dequeue
  transaction begin
    if (head->next == NULL)
      update head and tail
    else
      update head
  transaction end
```

With locks, neither thread can proceed in parallel since head/tail may be updated – with transactions, enqueue and dequeue can proceed in parallel – transactions will be aborted only if the queue is nearly empty

Transactions

- Transactional semantics:
  - when a transaction executes, it is as if the rest of the system is suspended and the transaction is in isolation
  - the reads and writes of a transaction happen as if they are all a single atomic operation
  - if the above conditions are not met, the transaction fails to commit (abort) and tries again

```
transaction begin
    read shared variables
    arithmetic
    write shared variables
transaction end
```

TM Implementation

Core
Core
Cache
Cache

- Caches track read-sets and write-sets
- Writes are made visible only at the end of the transaction
- At transaction commit, make your writes visible; others may abort

Detecting Conflicts – Basic Implementation

- Writes can be cached (can’t be written to memory) – if the block needs to be evicted, flag an overflow (abort transaction for now) – on an abort, invalidate the written cache lines
- Keep track of read-set and write-set (bits in the cache) for each transaction
- When another transaction commits, compare its write set with your own read set – a match causes an abort
- At transaction end, express intent to commit, broadcast write-set (transactions can commit in parallel if their write-sets do not intersect)

Summary of TM Benefits

- As easy to program as coarse-grain locks
- Performance similar to fine-grain locks
- Avoids deadlock

Design Space

- Data Versioning
  - Eager: based on an undo log
  - Lazy: based on a write buffer

- Conflict Detection
  - Optimistic detection: check for conflicts at commit time (proceed optimistically thru transaction)
  - Pessimistic detection: every read/write checks for conflicts (reduces work during commit)

“Lazy” Implementation

- An implementation for a small-scale multiprocessor with a snooping-based protocol
- Lazy versioning and lazy conflict detection
- Does not allow transactions to commit in parallel

“Lazy” Implementation

- When a transaction issues a read, fetch the block in read-only mode (if not already in cache) and set the rd-bit for that cache line
- When a transaction issues a write, fetch that block in read-only mode (if not already in cache), set the wr-bit for that cache line and make changes in cache
- If a line with wr-bit set is evicted, the transaction must be aborted (or must rely on some software mechanism to handle saving overflowed data)

“Lazy” Implementation

- When a transaction reaches its end, it must now make its writes permanent

- A central arbiter is contacted (easy on a bus-based system), the winning transaction holds on to the bus until all written cache line addresses are broadcasted (this is the commit) (need not do a writeback until the line is evicted – must simply invalidate other readers of these cache lines)

- When another transaction (that has not yet begun to commit) sees an invalidation for a line in its rd-set, it realizes its lack of atomicity and aborts (clears its rd- and wr-bits and re-starts)

"Lazy" Implementation

- Lazy versioning: changes are made locally – the "master copy" is updated only at the end of the transaction
- Lazy conflict detection: we are checking for conflicts only when one of the transactions reaches its end
- Aborts are quick (must just clear bits in cache, flush pipeline and reinstate a register checkpoint)

- Commit is slow (must check for conflicts, all the coherence operations for writes are deferred until transaction end)
- No fear of deadlock/livelock – the first transaction to acquire the bus will commit successfully
- Starvation is possible – need additional mechanisms

Acknowledgements

- EPFL, Onur Mutlu, Digital Design and Computer Architecture, 2020, 2023
- Utah University, Rajeev, CS/ECE 6810, Computer Architecture
- 计算机体系结构:量化研究方法(中文版第六版)
- UCSD CSE 240
- Washington University, CSE3 78
- 国科大, 胡伟武, 计算机体系结构
- CMU, CS 447 Computer Architecture
- UC Berkeley, CS 152 and CS 61C
- 国科大南京学院, 安学军等, 计算机体系结构
- 浙江大学, 计算机体系结构
- 南京大学, 计算机体系结构