DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Computer Architecture

Lecture 10: Thread-Level Parallelism--II (Chapter 5)

Chih-Wei Liu 劉志尉

National Chiao Tung University

cwliu@twins.ee.nctu.edu.tw

Review

- Caches contain all information on state of cached memory blocks
- Snooping cache over shared medium for smaller MP by invalidating other cached copies on write
- Sharing cached data

  ⇒ Coherence (values returned by a read),

  ⇒ Consistency (when a written value will be returned by a read)

A Cache Coherent System Must:

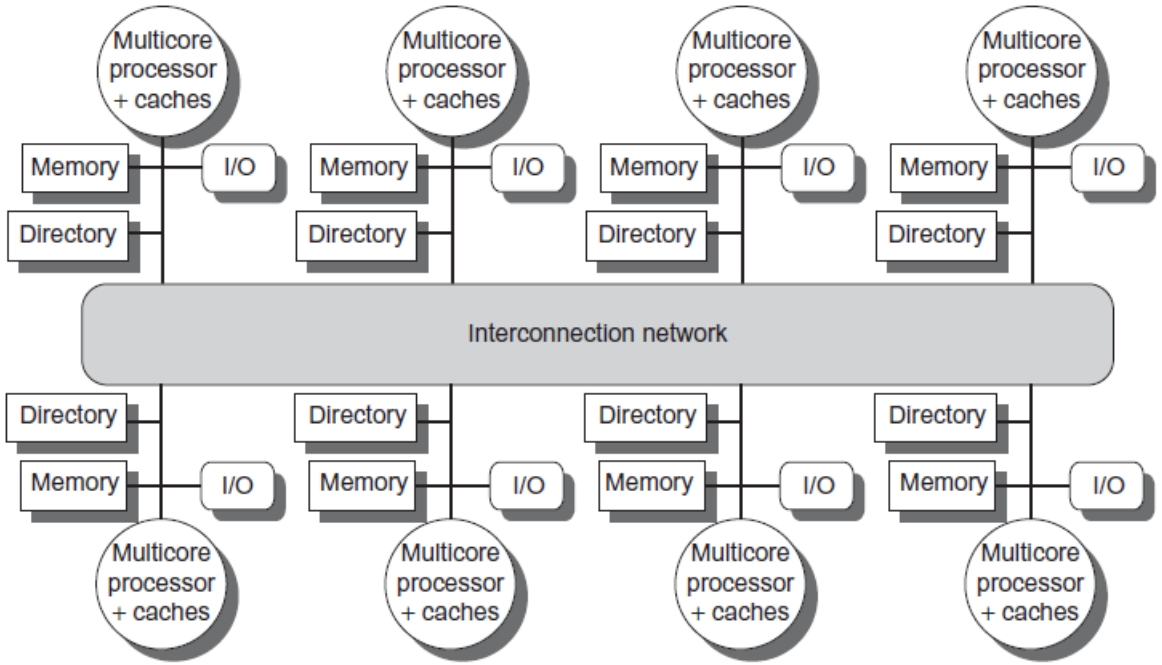
- Provide set of states, state transition diagram, and actions
- Manage coherence protocol
  - (0) Determine when to invoke coherence protocol
  - (a) Find info about state of block in other caches to determine action
    - whether need to communicate with other cached copies
  - (b) Locate the other copies
  - (c) Communicate with those copies (invalidate/update)
- (0) is done the same way on all systems
  - state of the line is maintained in the cache
  - protocol is invoked if an "access fault" occurs on the line
- Different approaches distinguished by (a) to (c)

Bus-based Coherence

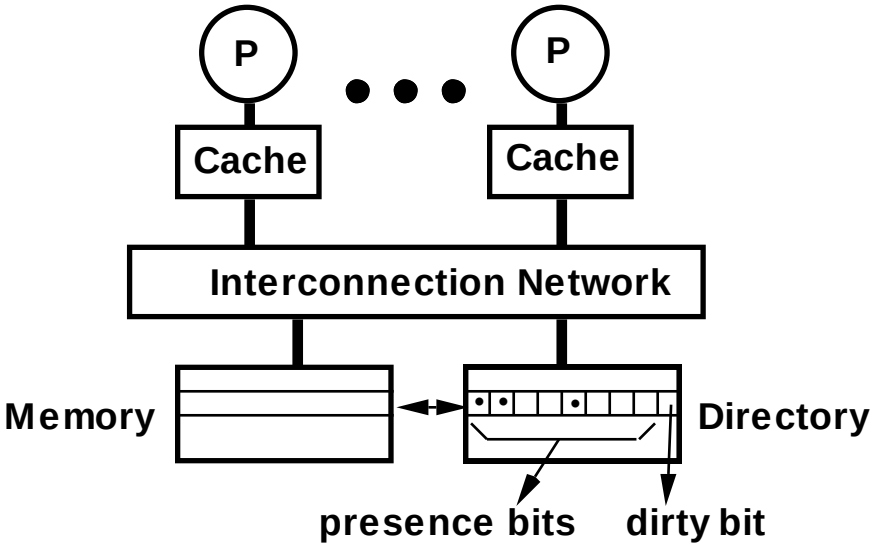
- All of (a), (b), (c) done through broadcast on bus
  - faulting processor sends out a “search”
  - others respond to the search probe and take necessary action
- Could do it in scalable network too
  - broadcast to all processors, and let them respond
- Conceptually simple, but broadcast doesn’t scale with p
  - on bus, bus bandwidth doesn’t scale
  - on scalable network, every fault leads to at least p network transactions
- Scalable coherence:
  - can have same cache states and state transition diagram
  - different mechanisms to manage protocol

Scalable Approach: Directories

- Every memory block has associated directory information (may be cached)
  - keeps track of copies of cached blocks and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information



Basic Operation of Directory



- k processors.
- With each cache-block in memory:
  k presence-bits, 1 dirty-bit
- With each cache-block in cache:
  1 valid bit, and 1 dirty (owner) bit

- Read from main memory by processor i:
  - If dirty-bit OFF then { read from main memory; turn p[i] ON; }
  - if dirty-bit ON then { recall line from dirty proc (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i;}
- Write to main memory by processor i:
  - If dirty-bit OFF then { supply data to i; send invalidations to all caches that have the block; turn dirty-bit ON; turn p[i] ON; ... }
  - ...

Directory Protocol

- Similar to Snoopy Protocol: Three states
  - Shared: ≥ 1 processors have data, memory up-to-date
  - Uncached (no processor has it; not valid in any cache)
  - Exclusive: 1 processor (owner) has data; memory out-of-date
- In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)
- Keep it simple(r):
  - Writes to non-exclusive data => write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent

Directory Protocol

- No bus and don't want to broadcast:
  - interconnect no longer single arbitration point
  - all messages have explicit responses
- Terms: typically 3 processors involved
  - Local node where a request originates
  - Home node where the memory location of an address resides
  - Remote node has a copy of a cache block, whether exclusive or shared
- Example messages on next slide: P = processor number, A = address

Possible Messages in Directory Protocol

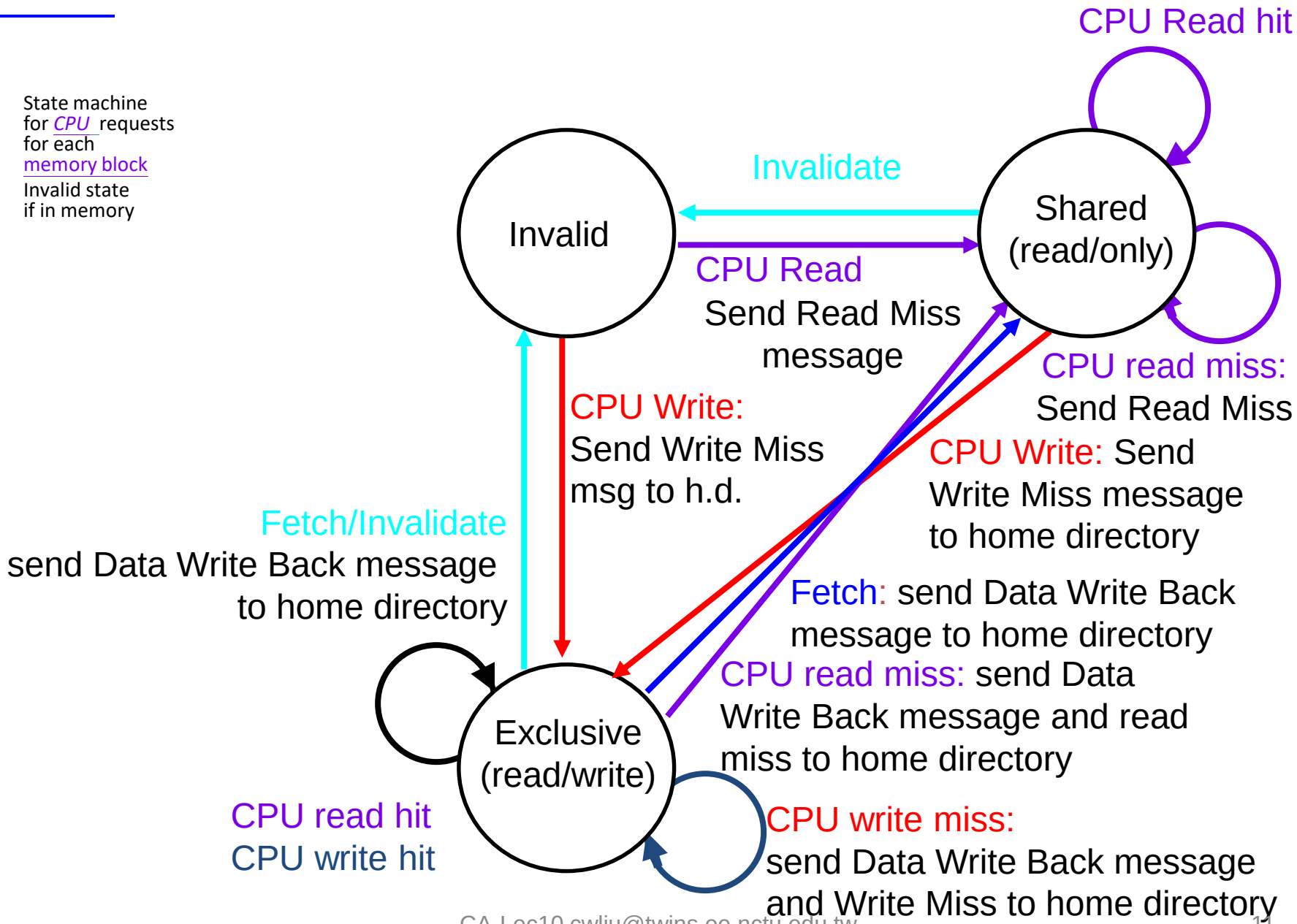
Message type	Source	Destination	Msg Content
Read miss	Local cache	Home directory	P, A
– *Processor P reads data at address A; make P a read sharer and request data*			
Write miss	Local cache	Home directory	P, A
– *Processor P has a write miss at address A; make P the exclusive owner and request data*			
Invalidate	Home directory	Remote caches	A
– *Invalidate a shared copy at address A*			
Fetch	Home directory	Remote cache	A
– *Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared*			
Fetch/Invalidate	Home directory	Remote cache	A
– *Fetch the block at address A and send it to its home directory; invalidate the block in the cache*			
Data value reply	Home directory	Local cache	Data
– *Return a data value from the home memory (read miss response)*			
Data write back	Remote cache	Home directory	A, Data
– *Write back a data value for address A (invalidate response)*			

State Transition Diagram for One Cache Block in Directory Based System

- States identical to snoopy case; transactions very similar.
- Transitions caused by read misses, write misses, invalidates, data fetch requests
- Generates read miss & write miss msg to home directory.
- Write misses that were broadcast on the bus for snooping => explicit invalidate & data fetch requests.
- Note: on a write, a cache block is bigger, so need to read the full cache block

CPU -Cache State Machine

- State machine for *CPU* requests for each memory block
- Invalid state if in memory

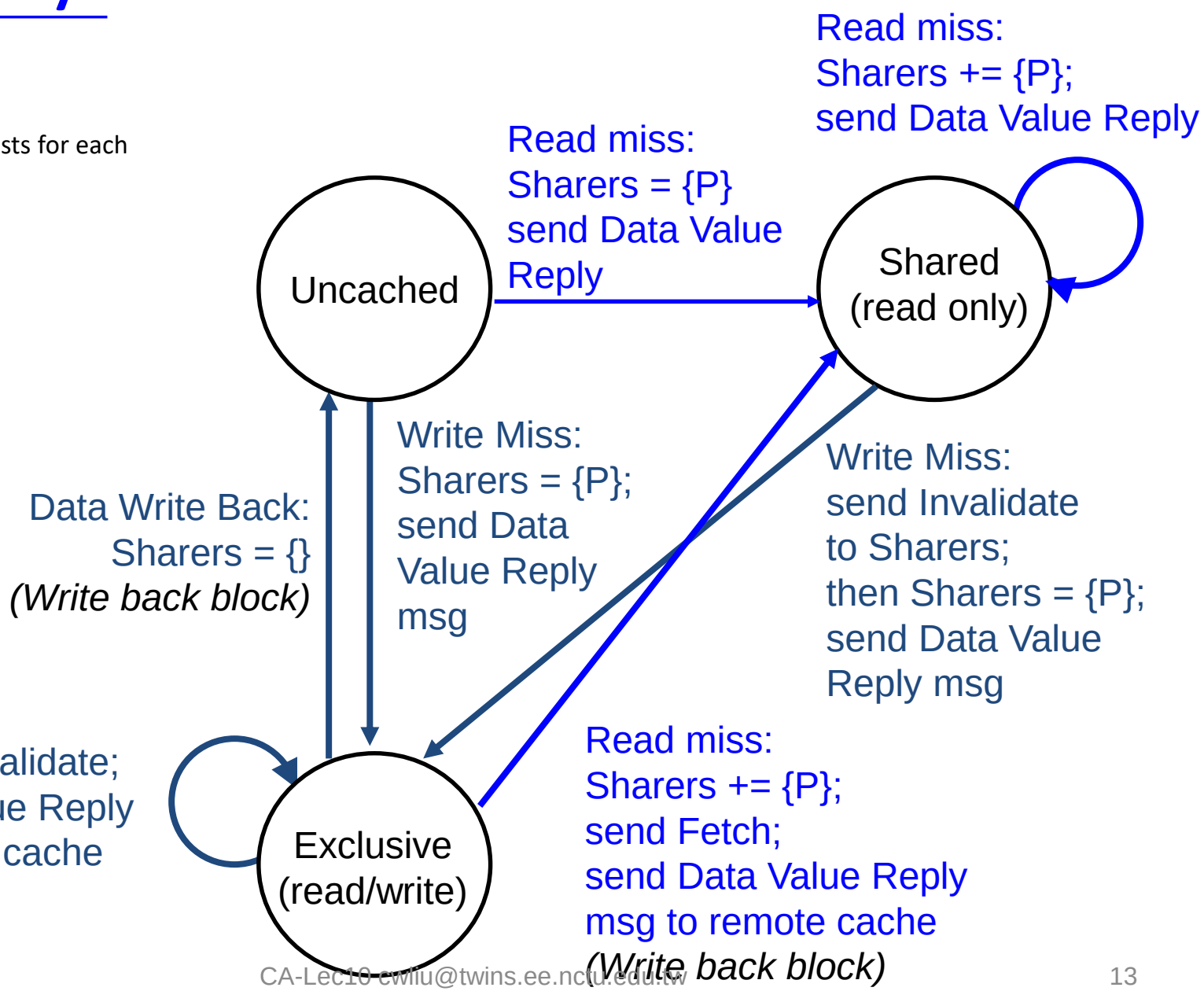


State Transition Diagram for Directory

- Same states & structure as the transition diagram for an individual cache
- 2 actions: update of directory state & send messages to satisfy requests
- Tracks all copies of memory block
- Also indicates an action that updates the sharing set, Sharers, as well as sending a message

Directory State Machine

- State machine for *Directory* requests for each memory block
- Uncached state if in memory



Example Directory Protocol

- Message sent to directory causes two actions:
  - Update the directory
  - More messages to satisfy request
- Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:
  - Read miss: requesting processor sent data from memory &requestor made only sharing node; state of block made Shared.
  - Write miss: requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.
- Block is Shared => the memory value is up-to-date:
  - Read miss: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
  - Write miss: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.

Example Directory Protocol

- Block is Exclusive: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) => three possible directory requests:
  - Read miss: owner processor sent data fetch message, causing state of block in owner's cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor.
    Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy).
    State is shared.
  - Data write-back: owner processor is replacing the block and hence must write it back, making memory copy up-to-date
    (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.
  - Write miss: block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			*P2*			*Bus*				*Directory*			*Memory*
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*State*	*{Procs}*	*Value*
P1: Write 10 to A1														
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block
(but different memory block addresses A1 ≠ A2)

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1			10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1														
P2: Write 40 to A2														

Write Back

A1 and A2 map to the same cache block

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1			10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2														

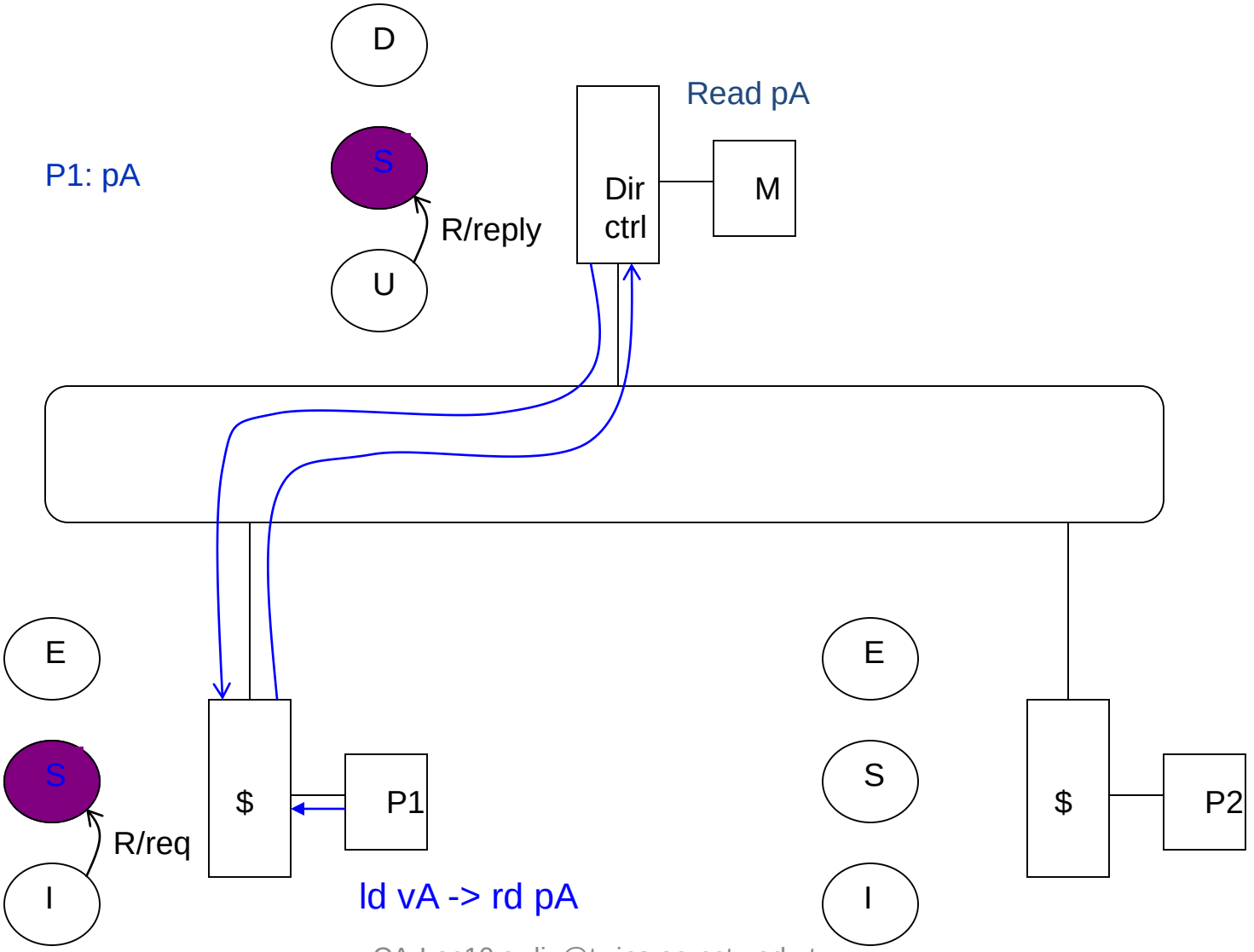
A1 and A2 map to the same cache block

Example

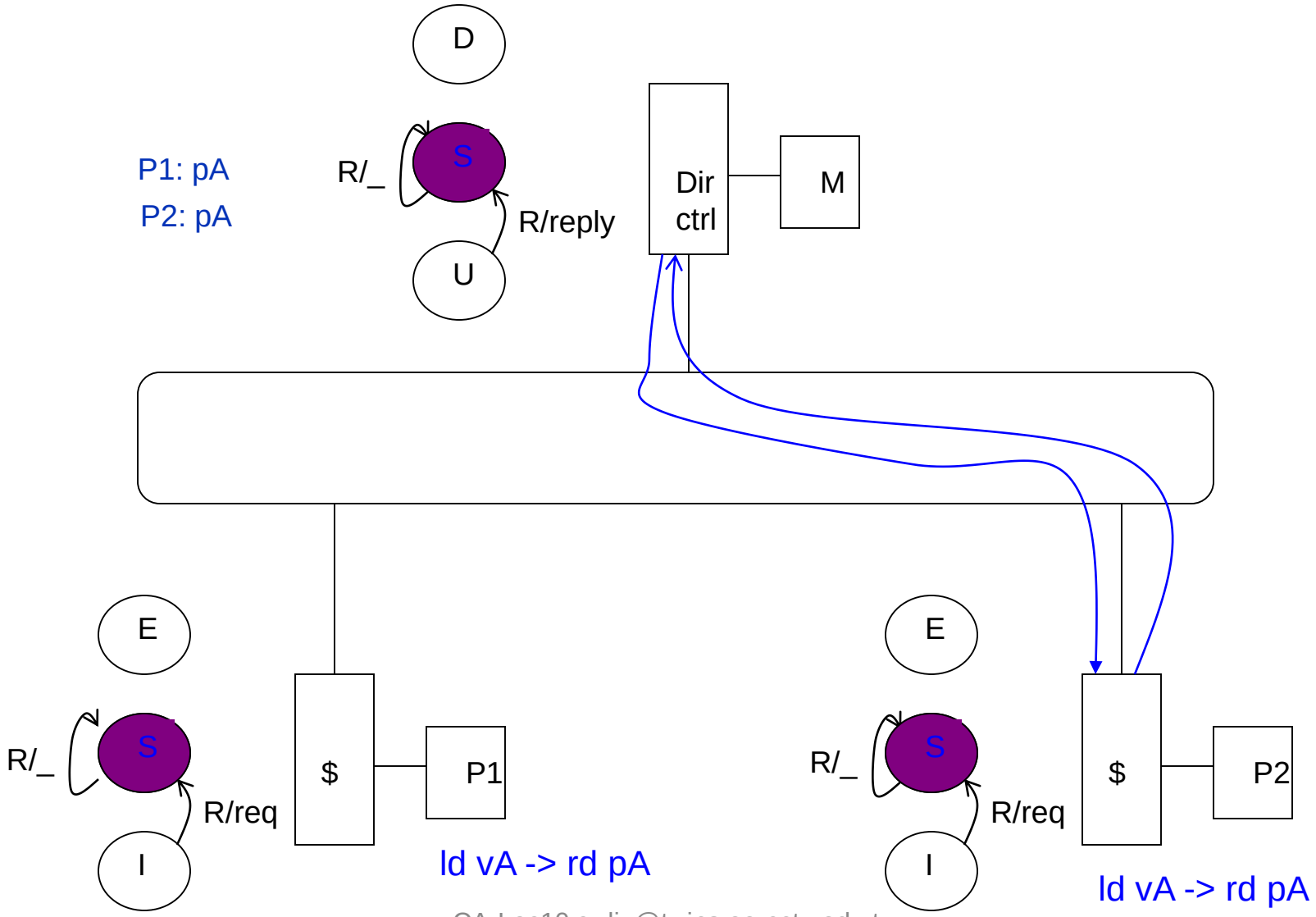
	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1			10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2							WrMs	P2	A2		A2	Excl.	{P2}	0
							WrBk	P2	A1	20	A1	Unca.	{}	20
				Excl.	A2	40	DaRp	P2	A2	0	A2	Excl.	{P2}	0

A1 and A2 map to the same cache block
(but different memory block addresses A1 ≠ A2)

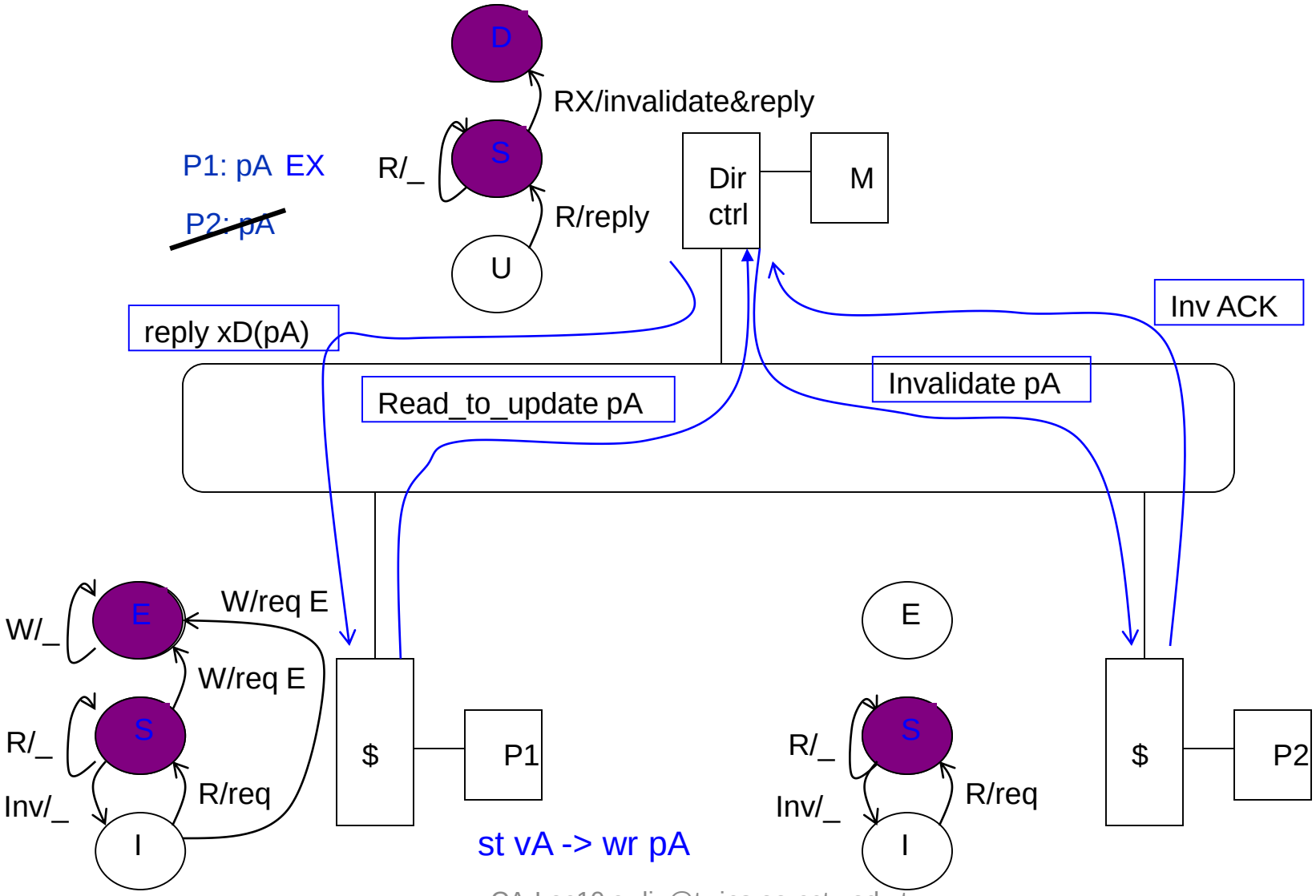
Example Directory Protocol (1st Read)



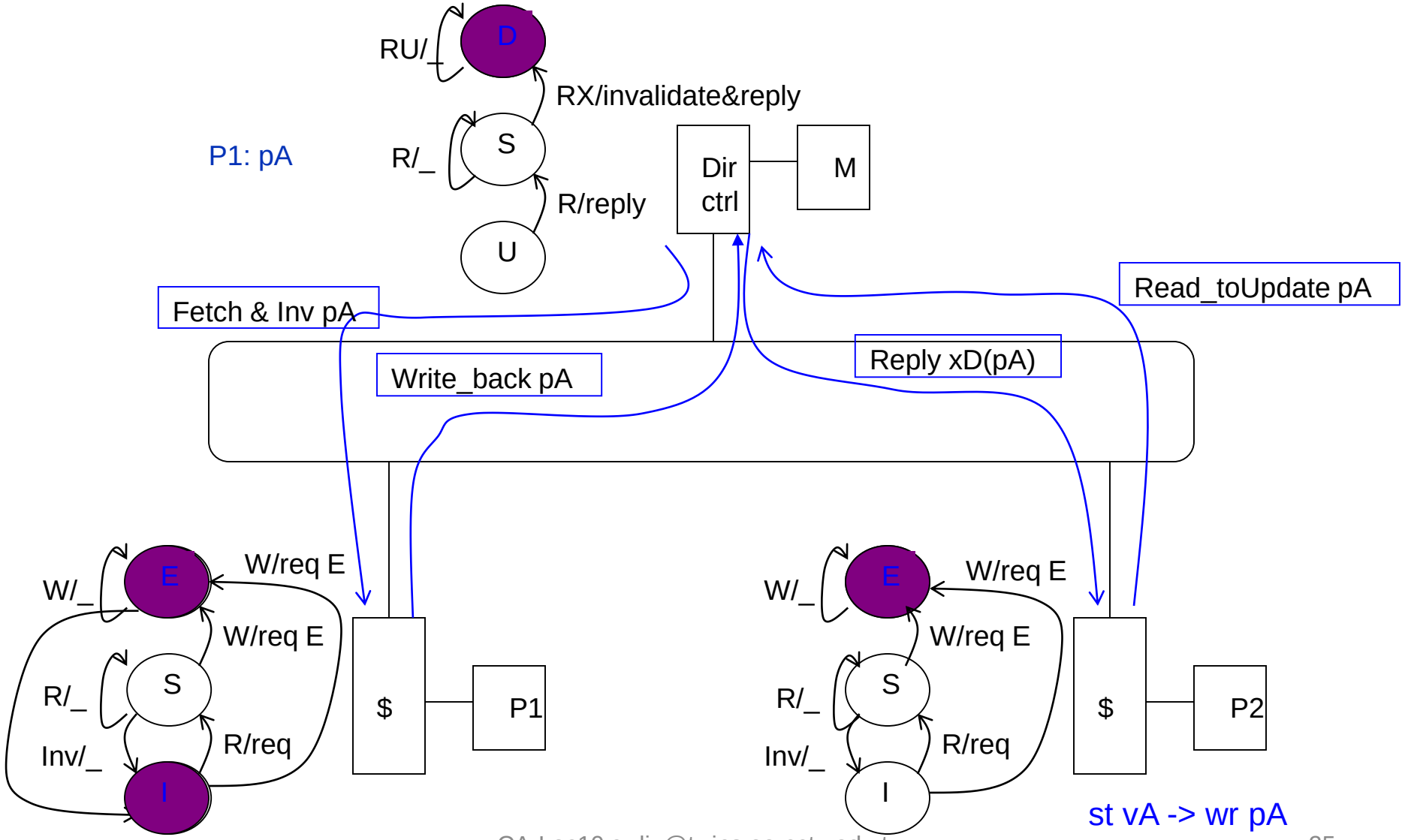
Example Directory Protocol (Read Share)



Example Directory Protocol (Wr to shared)



Example Directory Protocol (Wr to Ex)



A Popular Middle Ground

- Two-level “hierarchy”
- Individual nodes are multiprocessors, connected non-hiearchically
  - e.g. mesh of SMPs
- Coherence across nodes is directory-based
  - directory keeps track of nodes, not individual processors
- Coherence within nodes is snooping or directory
  - orthogonal, but needs a good interface of functionality
- SMP on a chip directory + snoop?

And in Conclusion …

- Caches contain all information on state of cached memory blocks
- Snooping cache over shared medium for smaller MP by invalidating other cached copies on write
- Sharing cached data ⇒ Coherence (values returned by a read), Consistency (when a written value will be returned by a read)
- Snooping and Directory Protocols similar; bus makes snooping easier because of broadcast (snooping => uniform memory access)
- Directory has extra data structure to keep track of state of all cache blocks
- Distributing directory => scalable shared address multiprocessor => Cache coherent, Non uniform memory access

Synchronization: The Basics

- Why Synchronize? Need to know when it is safe for different processes to use shared data

- Issues for Synchronization:
  - Uninterruptable instruction to fetch and update memory (atomic operation); i.e. the basic hardware primitive
  - User level synchronization operation using this primitive;
  - For large scale MPs, synchronization can be a bottleneck;
    - Appendix I.

Uninterruptable Instruction to Fetch and Update Memory

- Atomic exchange: interchange a value in a register for a value in memory

  0 ⇒ synchronization variable is free

  1 ⇒ synchronization variable is locked and unavailable

  – Set register to 1 & swap

  – New value in register determines success in getting lock

    0 if you succeeded in setting the lock (you were first)

    1 if other processor had already claimed access

  – Key is that exchange operation is indivisible

- Test-and-set: tests a value and sets it if the value passes the test

- Fetch-and-increment: it returns the value of a memory location and atomically increments it

RISC V Uninterruptable Instructions

- Hard to have read & write in 1 instruction: use 2 instead
- *Load reversed* (or load linked, load locked) + *store conditional*
  - **lr** loads the contents of memory given by **rs1** into **rd** and creates a reservation on that memory address
  - **sc** stores the value in **rs2** into the memory address given by **rs1** and **returns 1 if it succeeds** (no other store to same memory location since preceding load) and **0 otherwise**
- Atomic exchange (EXCH) on the memory location specified by the contents of x1 with the value in x4

```
try: mov     x3,x4          ;mov exchange value
     lr      x2,0(x1)       ;load reserved from
     sc      x3,0(x1)       ;store conditional
     beqz    x3,try         ;branch store fails (x3 = 0)
     mov     x4,x2          ;put load value in x4
```

- Atomic fetch & increment with lr & sc:

```
try: lr      x2,0(x1)       ;load reserved 0(x1)
     addi    x3,x2,#1       ;increment
     sc      x3,0(x1)       ;store conditional
     beqz    x2,try         ;branch store fails (x2 = 0)
```

Implementing Locks Using Coherence

- Spin locks: locks that a processor continuously tries to acquire, spinning around a loop until it succeeds.
- Lock a spin lock (no cache coherence)

```
            addi    x2,R0,#1
lockit:     EXCH    x2,0(x1)        ;atomic exchange
            bnez    x2,lockit       ;already locked?
```

- What about MP with cache coherency?
  - Want to spin on cache copy to avoid full memory latency
  - Likely to get cache hits for such variables
- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

Spin Lock Example in Directory Protocol

multiple processes trying to lock a variable using an atomic swap

Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0.
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write-back from P0; state shared.
4		(Waits while bus/ directory busy)	Lock = 0 test succeeds	Shared	Cache miss for P2 satisfied.
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied.
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

P0 starts with the lock (step 1), and the value of the lock is 1 (i.e., locked); it is initially exclusive and owned by P0 before step 1 begins. P0 exits and unlocks the lock (step 2). P1 and P2 race to see which reads the unlocked value during the swap (steps 3–5). P2 wins and enters the critical section (steps 6 and 7), while P1's attempt fails, so it starts spin waiting (steps 7 and 8).

Remarks

- The example shows that once the processor with the lock stores a 0 into the lock, all other caches are invalidated and must fetch the new value to update their copy of the lock.
  - Advantage of the `lr/sc` primitives: the read and write operations are explicitly separated.
- Optimized version using exchange (The lr need not cause any bus traffic):

  Lock a spin lock (with cache coherence)

```
lockit:     lr     x2,0(x1)      ;load reserved
            bnez   x2,lockit     ;not available-spin
            addi   x2,R0,#1      ;locked value
            sc     x2,0(x1)      ;store
            bnez   x2,lockit     ;branch if store fails
```

  - The first branch forms the spinning loop;
  - The second branch resolves races when two processors see the lock available simultaneously.

Another MP Issue: Memory Consistency Models

- What is consistency? When must a processor see the new value?

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

- Assume that the processes are running on different processors, and that locations A and B are originally cached by both processors with the initial value of 0.
- Should be Impossible for both if statements to be evaluated as true
  - Reaching the if statement means that either A or B must have been assigned the value 1.
  - What if write invalidate is delayed & processor continues?
- Sequential consistency (SC): result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved

  ⇒ assignments must be completed before the if statements are initiated
  - SC: delay all memory accesses until all invalidates done
  - SC reduces system performance, especially in a large number of processors or long interconnect delays

Example

SC reduces system performance

Problem

- Suppose we have a processor where a write miss takes 50 cycles to establish ownership, 10 cycles to issue each invalidate after ownership is established, and 80 cycles for an invalidate to complete and be acknowledged once it is issued.
- Assuming that four other processors share a cache block, how long does a write miss stall the writing processor if the processor is sequentially consistent?
- Assume that the invalidates must be explicitly acknowledged before the coherence controller knows they are completed. Suppose we could continue executing after obtaining ownership for the write miss without waiting for the invalidates; how long would the write take?

Answer:

- When we wait for invalidates, each write takes the sum of the ownership time plus the time to complete the invalidates.
- Because the invalidates can overlap, we need only worry about the last one, which starts 10+10+10+10=40 cycles after ownership is established. Therefore the total time for the write is 50+40+80=170 cycles.
- In comparison, the ownership time is only 50 cycles. With appropriate write buffer implementations, it is even possible to continue before ownership is established.

Memory Consistency Model

- A more efficient scheme (faster than sequential consistency) is to assume that programs are *synchronized*.
- A program is synchronized if all access to shared data are ordered by synchronization operations

```
write (x)
...
release (s) {unlock}        ……  "Unlock" after write
...
acquire (s) {lock}          ……  "Lock" before read
...
read(x)
```

- Cases where variables may be updated without ordering by synchronization are called *data race* and the execution outcome is unpredictable (depends on the relative speed of the processors).
- The *relaxed consistency model* is to allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering so that a synchronized program behaves as though the processor were sequentially consistent.

Relaxed Consistency Models (1/2)

- SC requires maintaining all four possible orderings: R → W, R → R, W→R, W → W.
- Relaxed Consistency Model:
  - Key idea: To specify the orderings of the form X → Y, meanings that operation X must complete before operation Y is done.
    - The relaxed models are defined by the subset of four orderings they relax
  - 3 major sets of relaxed orderings:

1. Relaxing only W→R ordering (all writes completed before next read)
   - Because retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization. Called processor consistency
2. Relaxing both the W→R and the W → W ordering (all writes completed before next write) . Called partial store order.
3. Relaxing all four ordering. Called weak ordering or release consistency. A variety of models depending on ordering restrictions and how synchronization operations enforce ordering

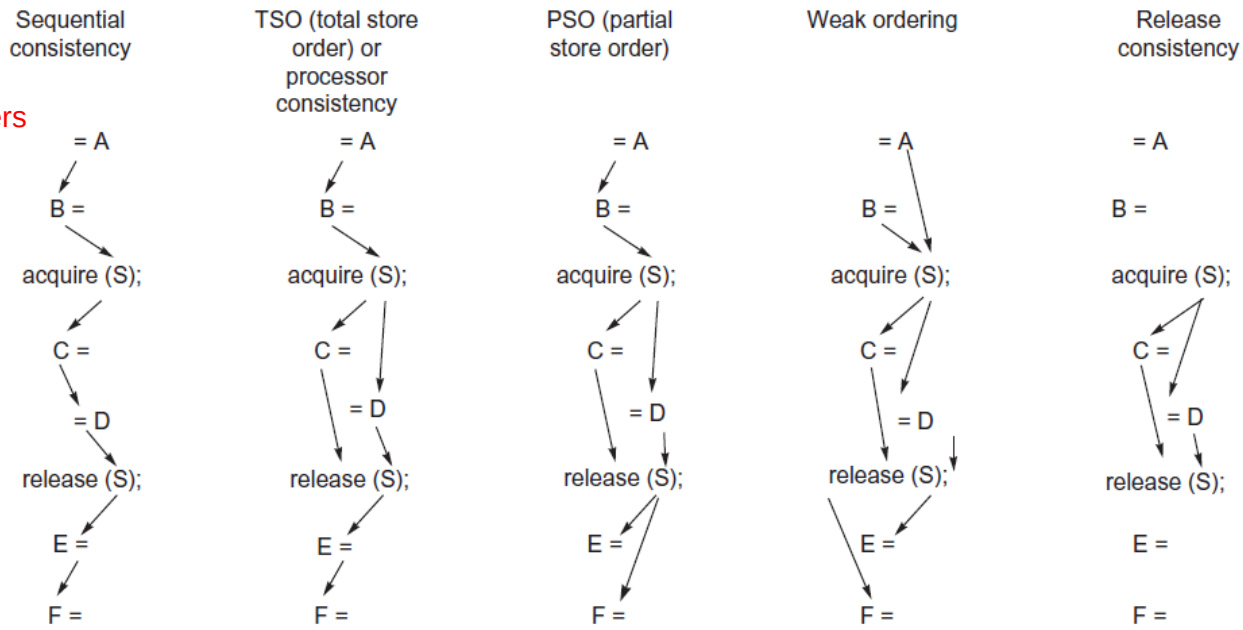
  - Many complexities in relaxed consistency models; defining precisely what it means for a write to complete; deciding when processors can see values that it has written

Relaxed Consistency Models (2/2)

Model	Used in	Ordinary orderings	Synchronization orderings
Sequential consistency	Most machines as an optional mode	R → R, R → W, W → R, W → W	S → W, S → R, R → S, W → S, S → S
Total store order or processor consistency	IBMS/370, DEC VAX, SPARC	R → R, R → W, W → W	S → W, S → R, R → S, W → S, S → S
Partial store order	SPARC	R → R, R → W	S → W, S → R, R → S, W → S, S → S
Weak ordering	PowerPC		S → W, S → R, R → S, W → S, S → S
Release consistency	MIPS, RISC V, Armv8, C, and C++ specifications		$S_A \rightarrow W$, $S_A \rightarrow R$, $R \rightarrow S_R$, $W \rightarrow S_R$, $S_A \rightarrow S_A$, $S_A \rightarrow S_R$, $S_R \rightarrow S_A$, $S_R \rightarrow S_R$

- S_A and S_R stand for acquire and release operations, respectively
- If we use S_A and S_R for each S consistently, each ordering with one S would become two orderings (e.g., S→W becomes $S_A \rightarrow W$, $S_R \rightarrow W$), and each S→S would become the four orderings shown in the last line of the bottom-right table entry

Only the minimum orders are shown with arrows.



Using Speculation to Hide Latency in Strict Consistency Models

- Speculation gives much of the performance advantage of relaxed models with sequential consistency
  - Executing the memory references out of order may generate violations of sequential consistency
  - Solution: (by using the delayed commit feature of a speculative processor)
    1. if an invalidation arrives for a result that has not been committed, use speculation recovery
    2. If the exact outcome differs from what would have been seen under sequential consistency, the processor will redo the execution.
- *One open question is how successful compiler technology will be in optimizing memory references to shared variables.*

Remarks: The Future of Multicore Scaling

- ILP scaling failed because of both limitations in the ILP available and the efficiency of exploiting that ILP
- Energy and power scale up indeed from generation to generation. E.g., 22nm vs. 11nm

Device count scaling (since a transistor is 1/4 the size)	4
Frequency scaling (based on projections of device speed)	1.75
Voltage scaling projected	0.81
Capacitance scaling projected	0.39
Energy per switched transistor scaling (CV^2)	0.26
Power scaling assuming fraction of transistors switching is the same and chip exhibits full frequency scaling	1.79

Figure 5.36 A comparison of the 22 nm technology of 2016 with a future 11 nm technology, likely to be available sometime between 2022 and 2024. The characteristics of the 11 nm technology are based on the International Technology Roadmap for Semiconductors, which has been recently discontinued because of uncertainty about the continuation of Moore's Law and what scaling characteristics will be seen.

- If we assume the heat dissipation limit remains, then only partial number of cores can be active.

Multicore Scaling Example

- Suppose we have a 96-core processor, but on average only 54 cores can be busy. Suppose that 90% of the time, we can use all available cores; 9% of the time, we can use 50 cores; and 1% of the time is strictly serial. How much speedup might we expect?
- Answer

$$\text{Average Processor Usage} = 0.09 \times 50 + 0.01 \times 1 + 0.90 \times \text{Max processor}$$
$$54 = 4.51 + 0.90 \times \text{Max processor}$$
$$\text{Max processor} = 55$$

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{55}}{55} + \frac{\text{Fraction}_{50}}{50} + (1 - \text{Fraction}_{55} - \text{Fraction}_{50})}$$

$$\text{Speedup} = \frac{1}{\frac{0.90}{55} + \frac{0.09}{50} + 0.01} = 35.5$$

➔ Multicore does not magically solve the power problem, indeed