DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

5008: Computer Architecture

Chapter 4 - Multiprocessors and Thread-Level Parallelism --II

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)

Coherency Misses

1. True sharing misses arise from the communication of data through the cache coherence mechanism
   - Invalidates due to 1st write to shared block
   - Reads by another CPU of modified block in different cache
   - Miss would still occur if block size were 1 word
2. False sharing misses when a block is invalidated because some word in the block, other than the one being read, is written into
   - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
   - Block is shared, but no word in block is actually shared $\Rightarrow$ miss would not occur if block size were 1 word

Example: True vs. False Sharing vs. Hit?

- **Assume x1 and x2 in same cache block. P1 and P2 both read x1 and x2 before.**

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		True miss; invalidate x2 in P1

Outline

- Review
- Directory-based protocols and examples
- Synchronization
- Relaxed Consistency Models
- Conclusion

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)

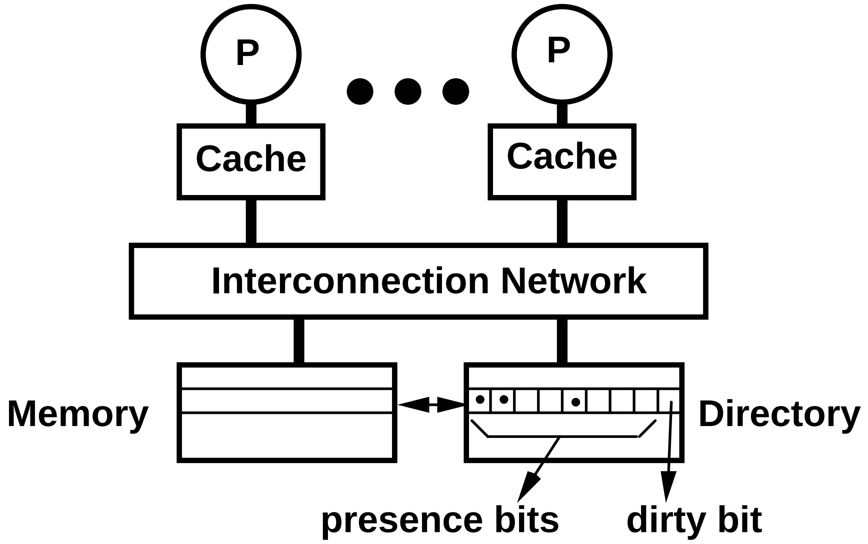
Bus-based Coherence

- 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: $\geq$ 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

Directory Protocol Messages (Fig 4.22)

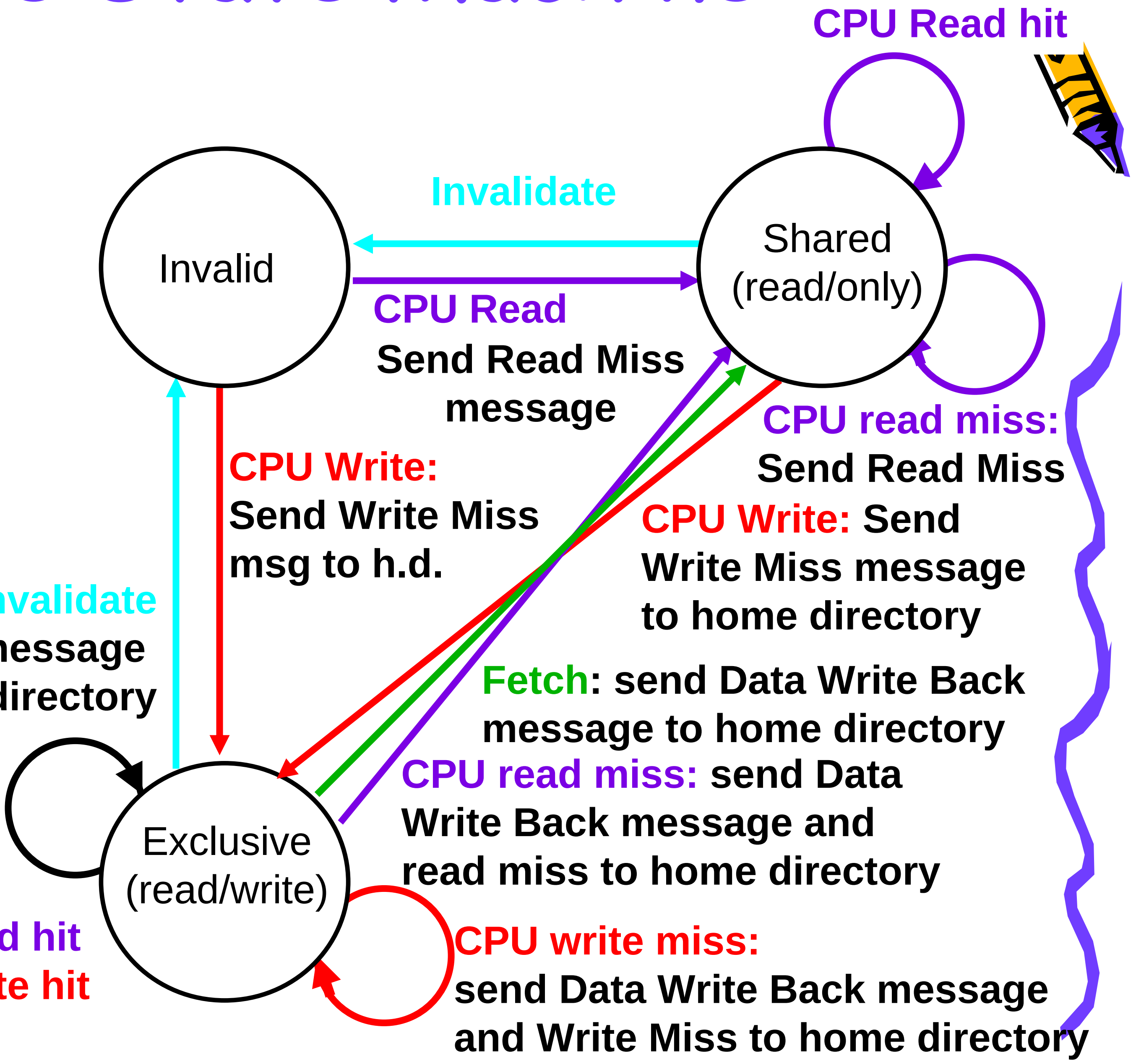
Message type	*Source*	*Destination*	*Msg Content*
Read miss	**Local cache**	**Home directory**	**P, A**
– *Processor P reads data at address A; make P a read sharer and request data*			
Write miss	**Local cache**	**Home directory**	**P, A**
– *Processor P has a write miss at address A; make P the exclusive 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



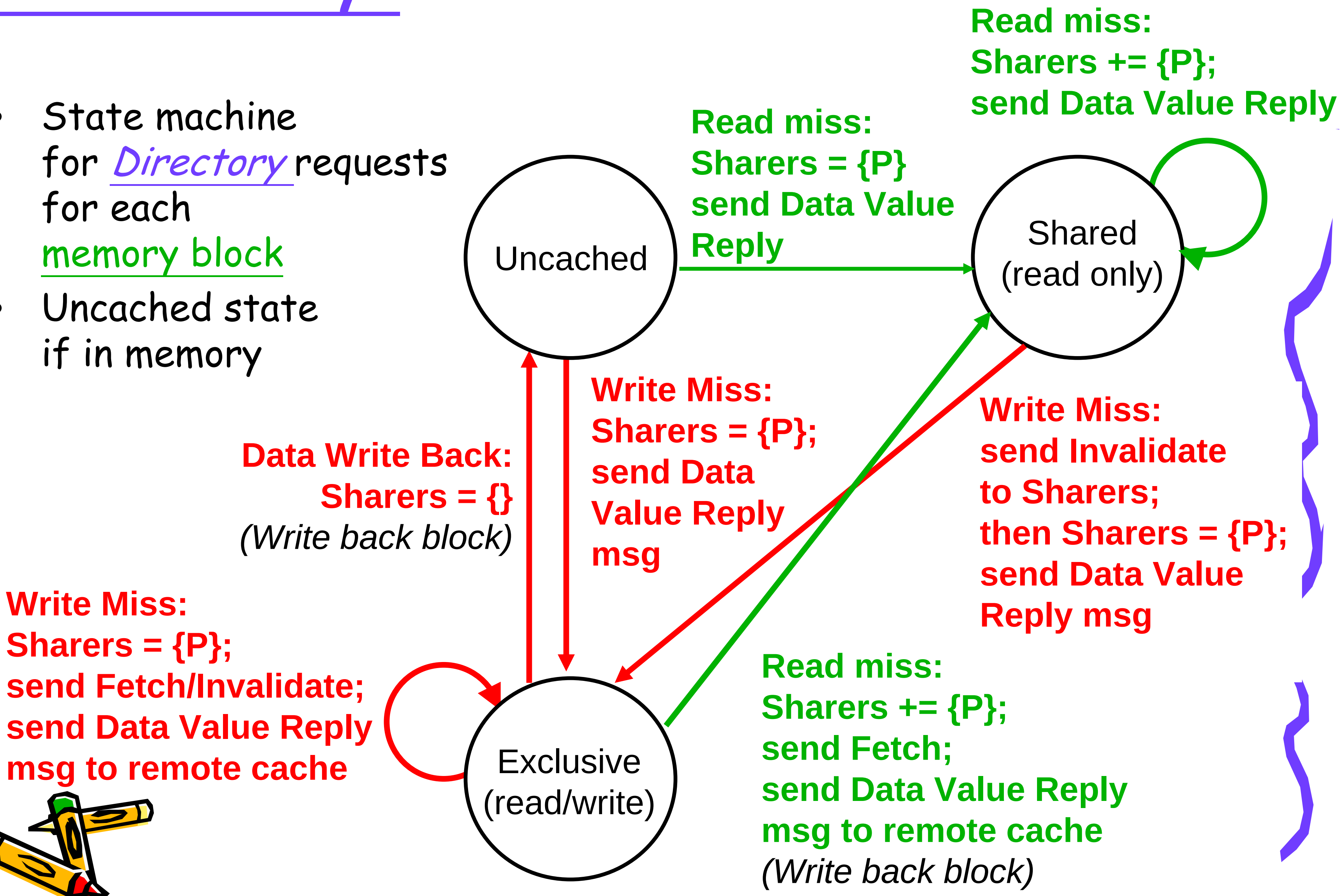
CPU Read hit
Invalid
Shared (read/only)
Invalid
CPU Read
Send Read Miss message
CPU read miss:
Send Read Miss
CPU Write:
Send Write Miss msg to h.d.
CPU Write: Send Write Miss message to home directory
Fetch/Invalidate
send Data Write Back message to home directory
Fetch: send Data Write Back message to home directory
CPU read miss: send Data Write Back message and read miss to home directory
Exclusive (read/write)
CPU read hit
CPU write hit
CPU write miss:
send Data Write Back message and Write Miss to home directory

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



Uncached
Shared (read only)
Exclusive (read/write)
Read miss: Sharers = {P} send Data Value Reply
Read miss: Sharers += {P}; send Data Value Reply
Write Miss: Sharers = {P}; send Data Value Reply msg
Data Write Back: Sharers = {} (Write back block)
Write Miss: send Invalidate to Sharers; then Sharers = {P}; send Data Value Reply msg
Write Miss: Sharers = {P}; send Fetch/Invalidate; send Data Value Reply msg to remote cache
Read miss: Sharers += {P}; send Fetch; send Data Value Reply msg to remote cache (Write back block)

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

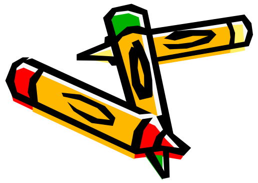
- Block is Exclusive: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) => three possible directory requests:
  - Read miss: owner processor sent data fetch message, causing state of block in owner's cache to transition to Shared and causes owner 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



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														

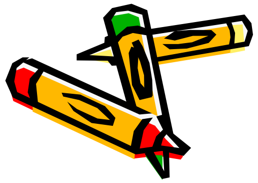
A1 and A2 map to the same cache block

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



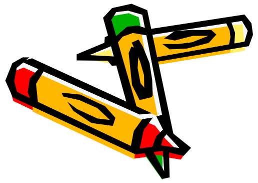
Example

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

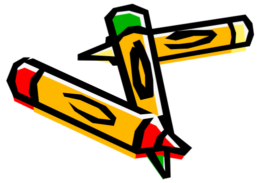


Example

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**

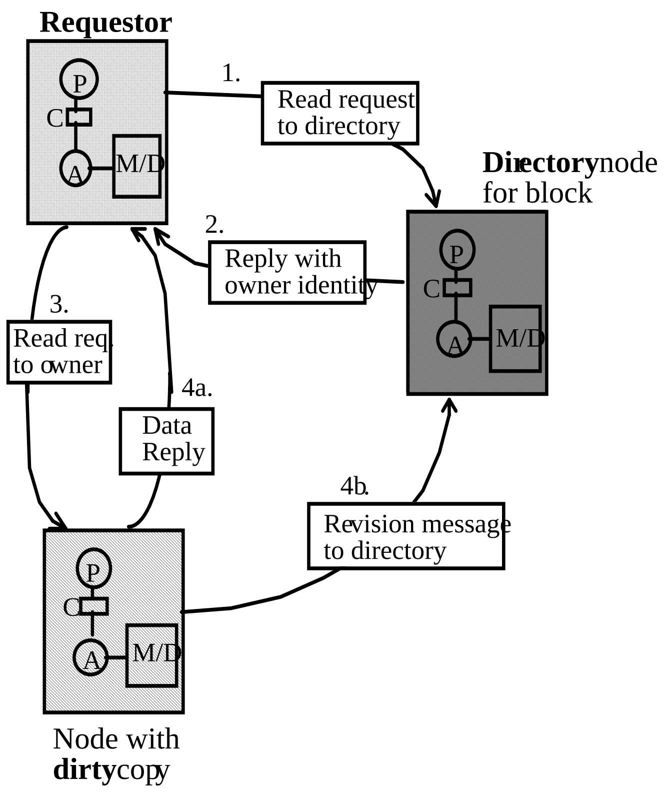
	P1			*P2*			*Bus*				*Directory*			*Memory*
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*State*	*{Procs}*	*Value*
P1: Write 10 to A1							*WrMs*	P1	A1		*A1*	*Ex*	*{P1}*	
	Excl.	*A1*	*10*				*DaRp*	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				*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 $\neq$ A2)

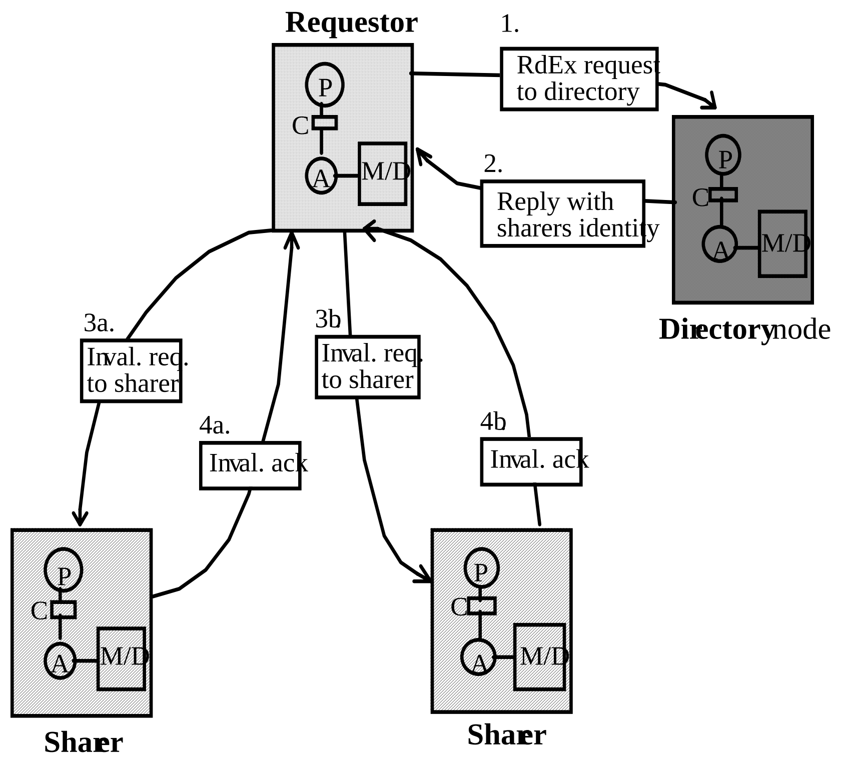
Implementing a Directory

- We assume operations atomic, but they are not; reality is much harder; must avoid deadlock when run out of buffers in network (see Appendix E)
- Optimizations:
  - read miss or write miss in Exclusive: send data directly to requestor from owner vs. 1st to memory and then from memory to requestor

Basic Directory Transactions

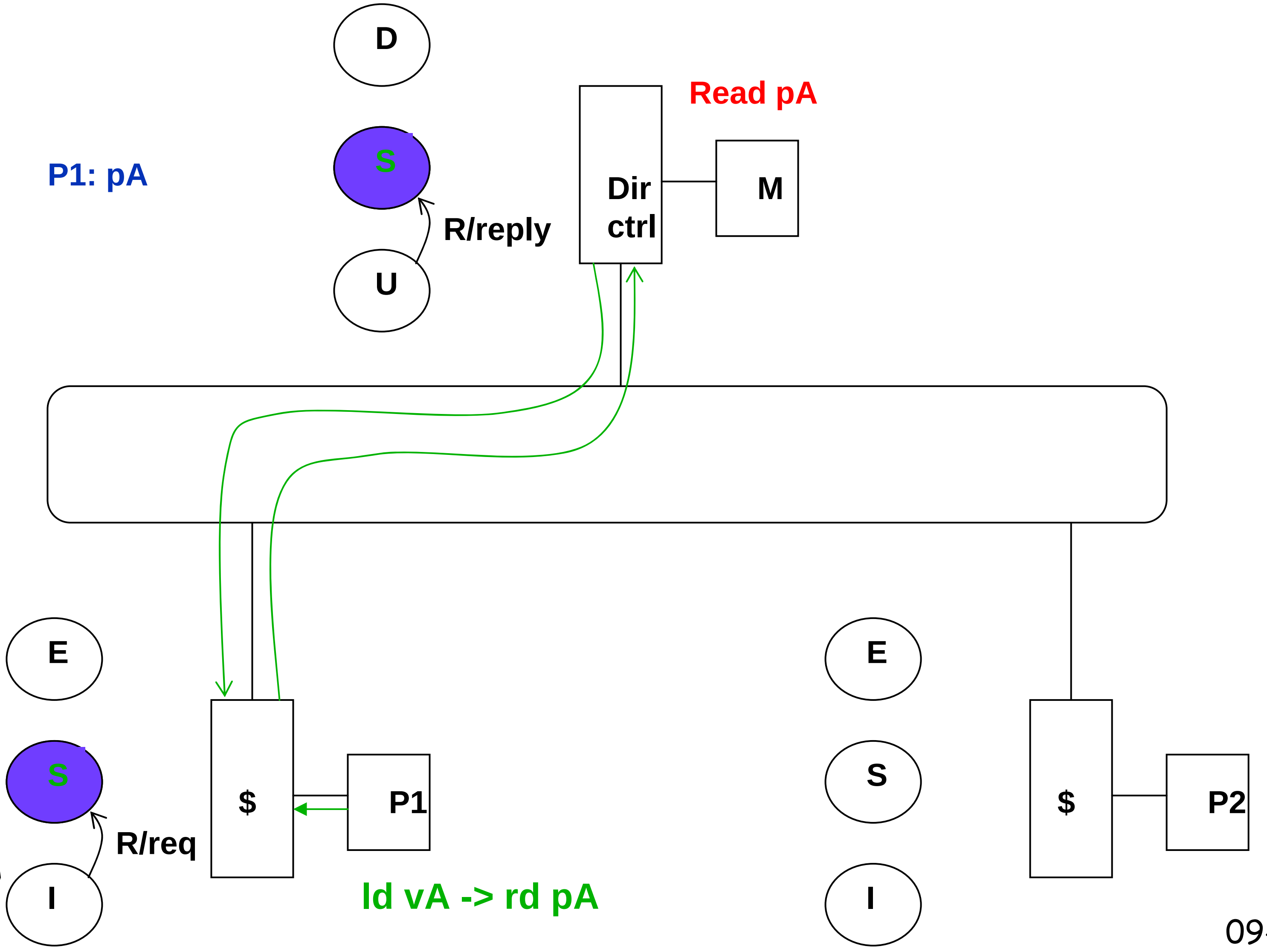


(a) Read miss to a block in dirty state

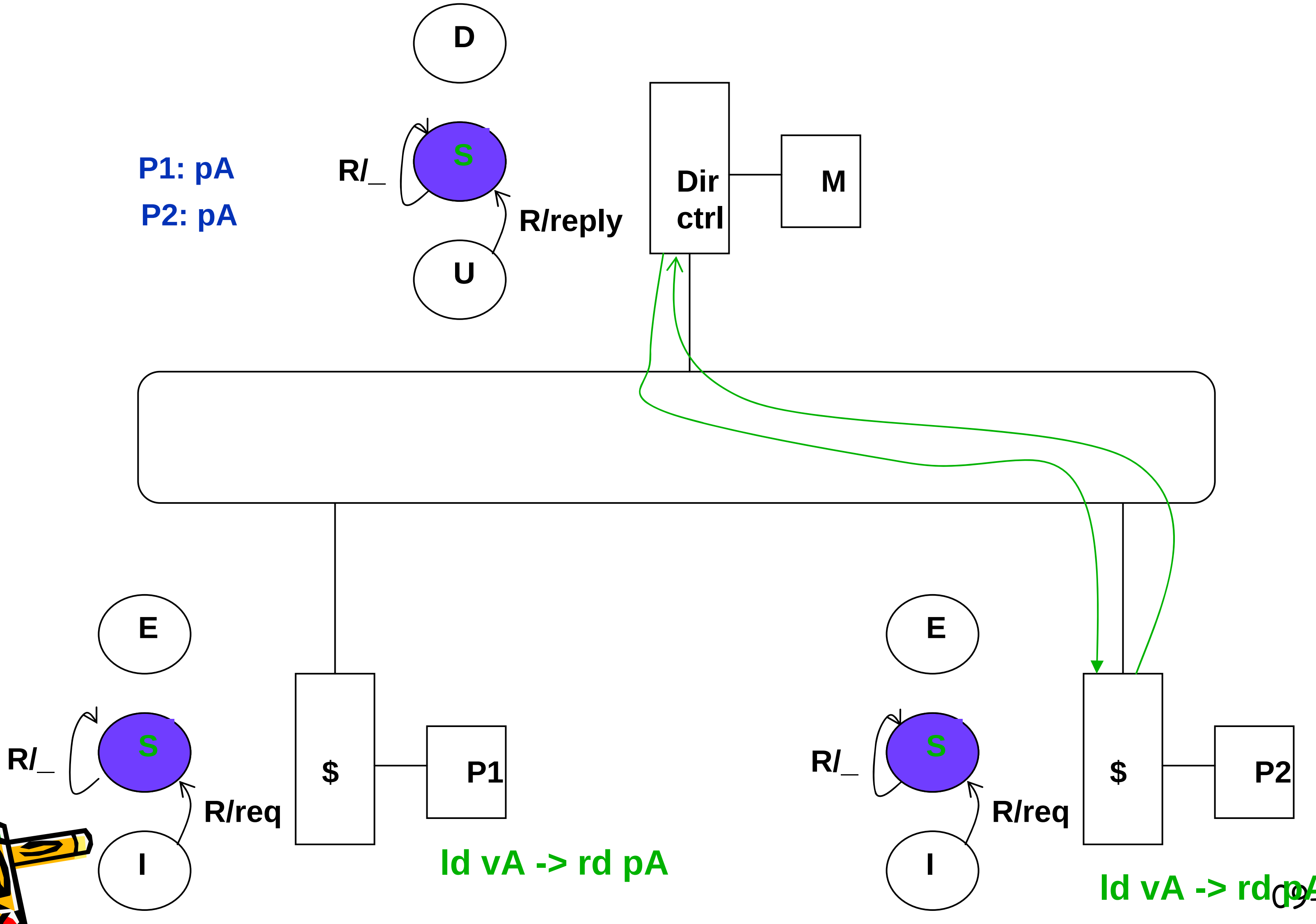


(b) Write miss to a block with two sharers

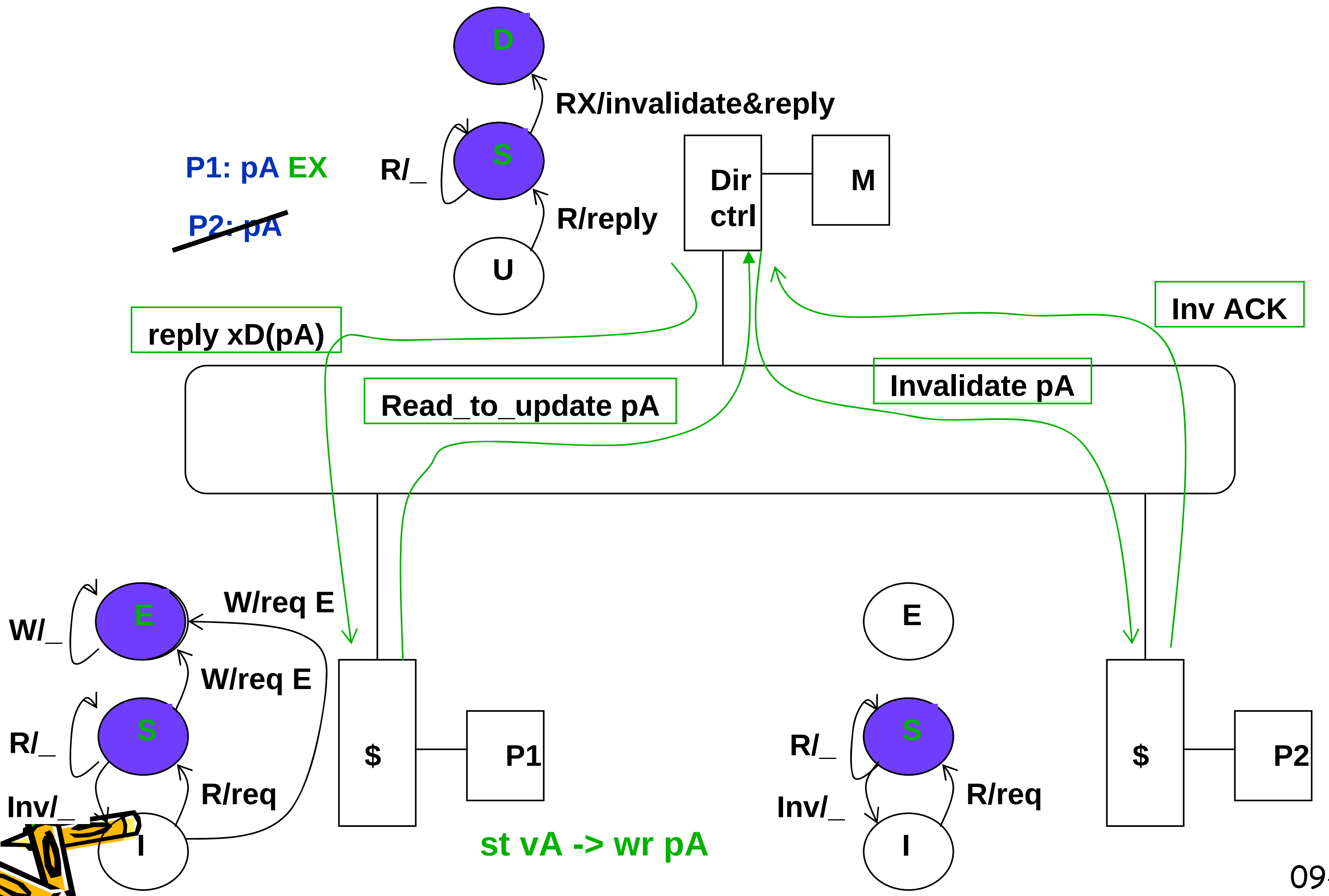
Example Directory Protocol (1st Read)



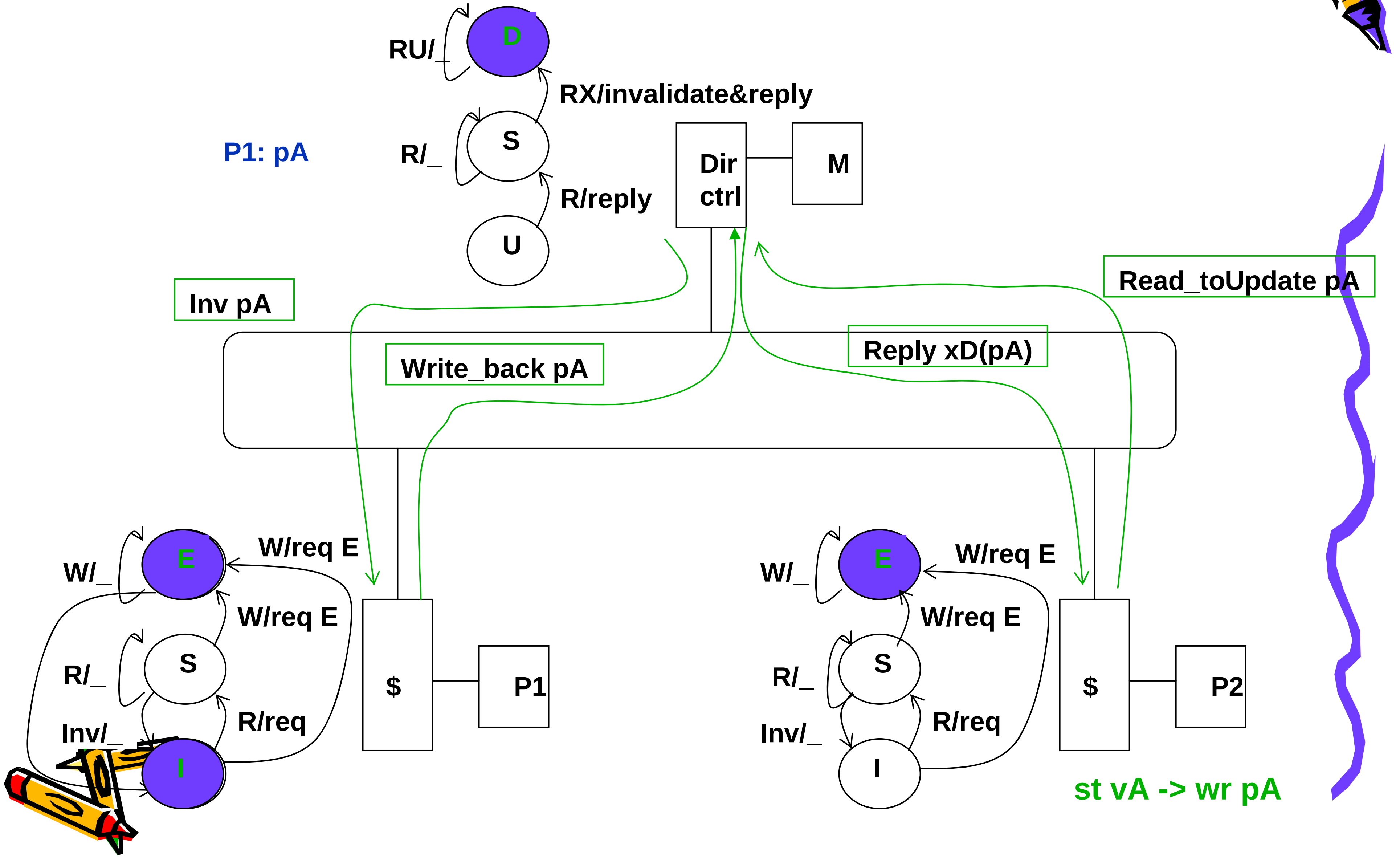
Example Directory Protocol (Read Share)



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 $\Rightarrow$ 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

Outline

- Review
- Directory-based protocols and examples
- Synchronization
- Relaxed Consistency Models
- Conclusion

Synchronization

- 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);
  - User level synchronization operation using this primitive;
  - For large scale MPs, synchronization can be a bottleneck;

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
  - 0 ⇒ synchronization variable is free

Uninterruptable Instruction to Fetch and Update Memory

- Hard to have read & write in 1 instruction: use 2 instead
- Load linked (or load locked) + store conditional
  - Load linked returns the initial value
  - Store conditional returns 1 if it succeeds (no other store to same memory location since preceding load) and 0 otherwise
- Example doing atomic swap with LL & SC:

```
try:  mov     R3,R4           ; mov exchange value
      ll      R2,0(R1)        ; load linked
      sc      R3,0(R1)        ; store conditional
      beqz    R3,try          ; branch store fails (R3 = 0)
      mov     R4,R2           ; put load value in R4
```

- Example doing fetch & increment with LL & SC:

```
try:  ll      R2,0(R1)        ; load linked
      addi    R2,R2,#1        ; increment (OK if reg–reg)
      sc      R2,0(R1)        ; store conditional
      beqz    R2,try          ; branch store fails (R2 = 0)
```

User Level Synchronization—Operation Using this Primitive

- Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock

```
                daddui  R2,R0,#1
lockit:         exch    R2,0(R1)        ;atomic exchange
                bnez    R2,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
- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange ("test and test&set"):

```
try:            li      R2,#1
lockit:         lw      R3,0(R1)        ;load var
                bnez    R3,lockit       ;≠ 0 ⇒ not free ⇒ spin
                exch    R2,0(R1)        ;atomic exchange
                bnez    R2,try          ;already locked?
```

Another MP Issue: Memory Consistency Models

- What is consistency? When must a processor see the new value? e.g., seems that

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

- Impossible for both if statements L1 & L2 to be true?
  - What if write invalidate is delayed & processor continues?
- Memory consistency models:
  what are the rules for such cases?
- Sequential consistency: 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

Memory Consistency Model

- Schemes faster execution to sequential consistency
- Not an issue for most programs; they are synchronized
  - A program is synchronized if all access to shared data are ordered by synchronization operations

```
write (x)
...
release (s) {unlock}
...
acquire (s) {lock}
...
read(x)
```

- Only those programs willing to be nondeterministic are not synchronized: "data race": outcome f(proc. speed)
- Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW
  to different addresses

Relaxed Consistency Models: The Basics

- Key idea: allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering, so that a synchronized program behaves as if the processor were sequentially consistent
  - By relaxing orderings, may obtain performance advantages
  - Also specifies range of legal compiler optimizations on shared data
  - Unless synchronization points are clearly defined and programs are synchronized, compiler could not interchange read and write of 2 shared data items because might affect the semantics of the program
- 3 major sets of relaxed orderings:

1. 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. W → W ordering (all writes completed before next write)
3. R → W and R → R orderings, a variety of models depending on ordering restrictions and how synchronization operations enforce ordering

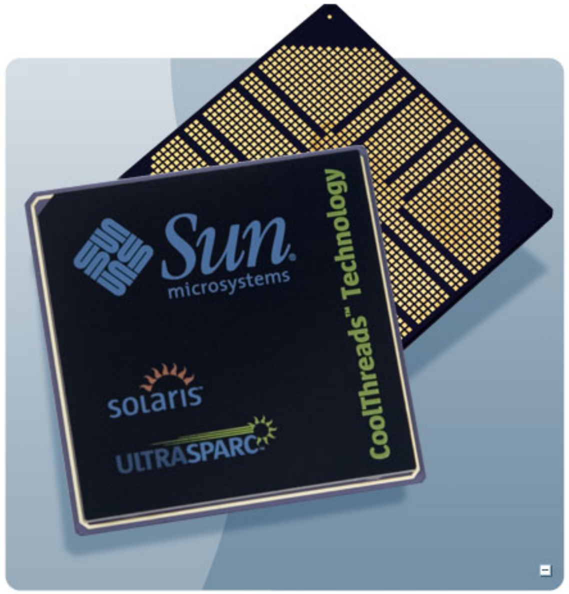
- Many complexities in relaxed consistency models; defining precisely what it means for a write to complete; deciding when processors can see values that it has written

Outline

- Review
- Directory-based protocols and examples
- Synchronization
- Relaxed Consistency Models
- Conclusion
- T1 (“Niagara”) Multiprocessor

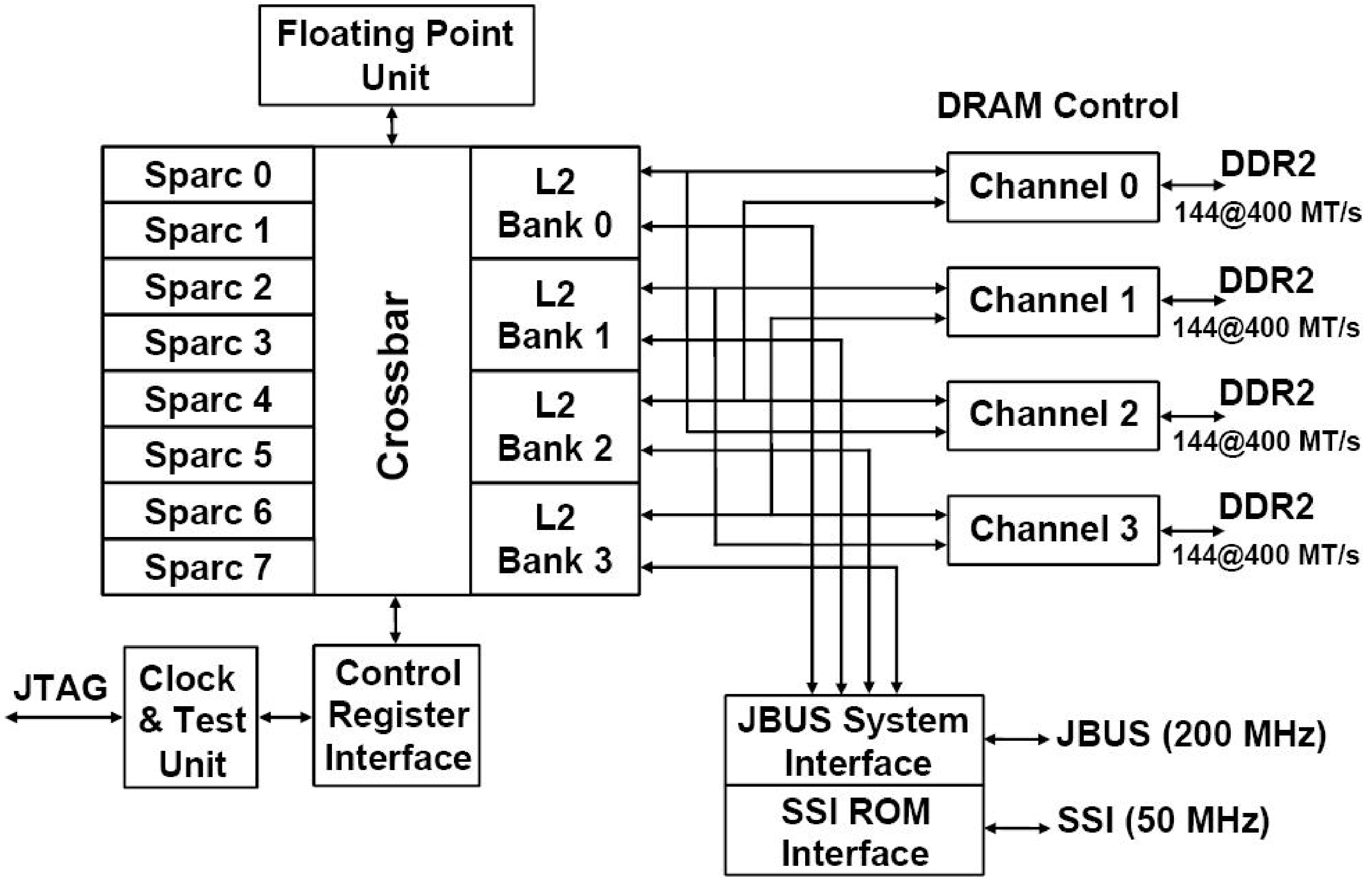
T1 ("Niagara")

- Target: Commercial server applications
  - **High thread level parallelism (TLP)**
    - Large numbers of parallel client requests
  - **Low instruction level parallelism (ILP)**
    - High cache miss rates
    - Many unpredictable branches
    - Frequent load-load dependencies
- Power, cooling, and space are major concerns for data centers
- Metric: Performance/Watt/Sq. Ft.
- Approach: Multicore, Fine-grain multithreading, Simple pipeline, Small L1 caches, Shared L2



T1 Architecture

- Also ships with 6 or 4 processors



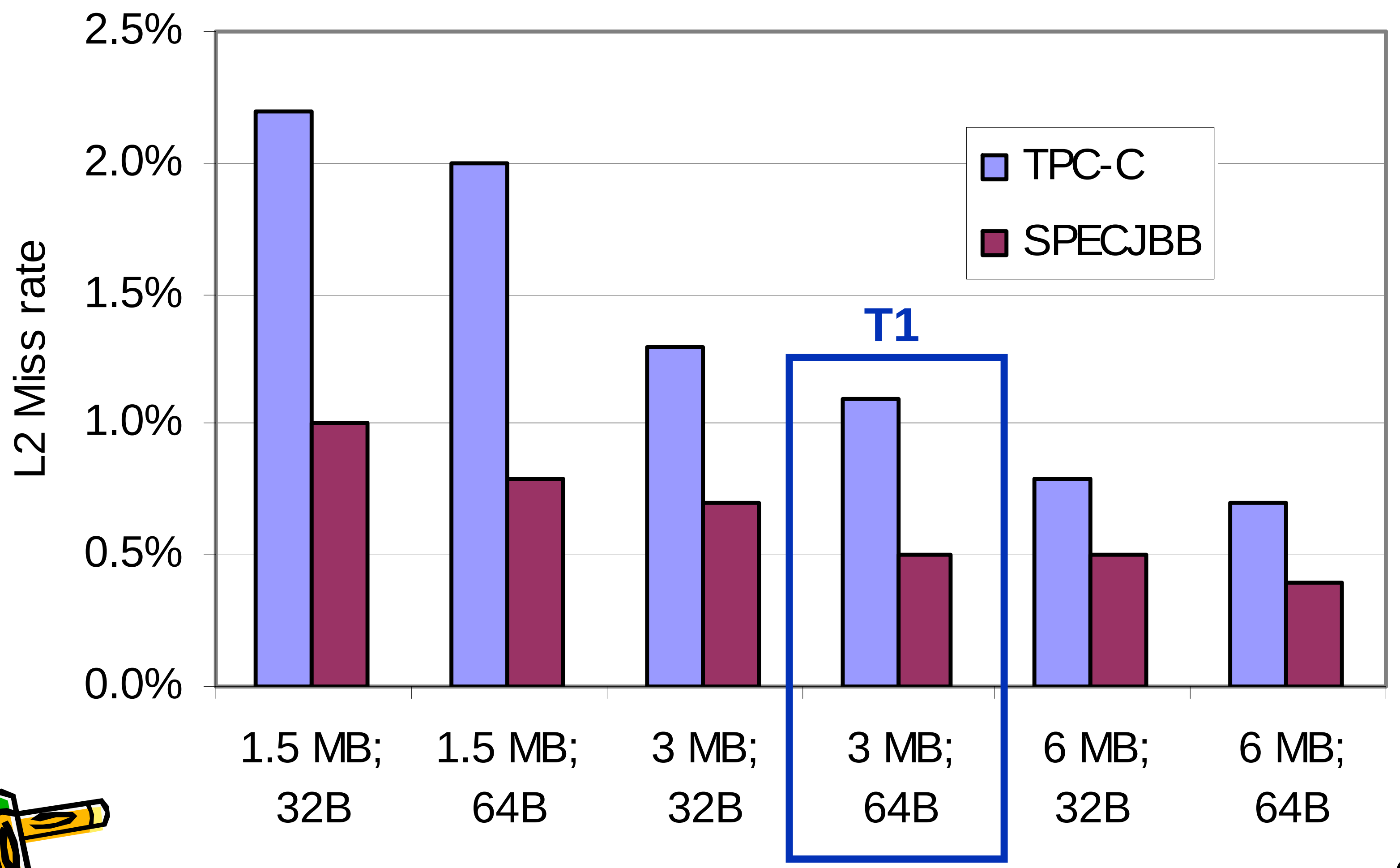
T1 Fine-Grained Multithreading

- Each core supports four threads and has its own level one caches (16KB for instructions and 8 KB for data)
- Switching to a new thread on each clock cycle
- Idle threads are bypassed in the scheduling
  - Waiting due to a pipeline delay or cache miss
  - Processor is idle only when all 4 threads are idle or stalled
- Both loads and branches incur a 3 cycle delay that can only be hidden by other threads
- A single set of floating point functional units is shared by all 8 cores
  - floating point performance was not a focus for T1

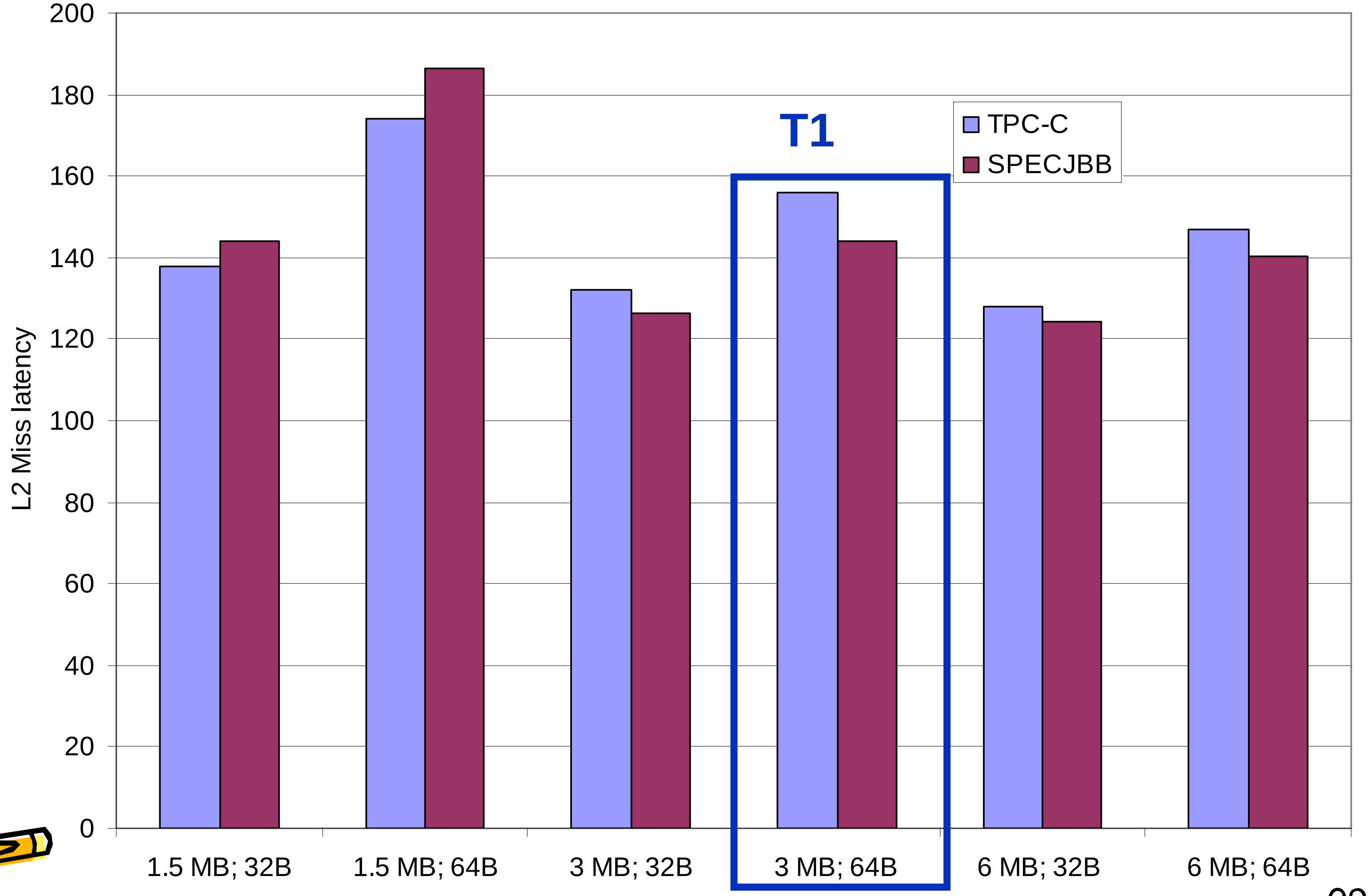
Memory, Clock, Power

- 16 KB 4 way set assoc. I$/ core
- 8 KB 4 way set assoc. D$/ core
- 3MB 12 way set assoc. L2 $ shared
  - 4 x 750KB independent banks
  - crossbar switch to connect
  - 2 cycle throughput, 8 cycle latency
  - Direct link to DRAM & Jbus
  - Manages cache coherence for the 8 cores
  - CAM based directory
- Coherency is enforced among the L1 caches by a directory associated with each L2 cache block
- Used to track which L1 caches have copies of an L2 block
- By associating each L2 with a particular memory bank and enforcing the subset property, T1 can place the directory at L2 rather than at the memory, which reduces the directory overhead
- L1 data cache is write-through, only invalidation messages are required; the data can always be retrieved from the L2 cache
- 1.2 GHz at ≈72W typical, 79W peak power consumption

Miss Rates: L2 Cache Size, Block Size



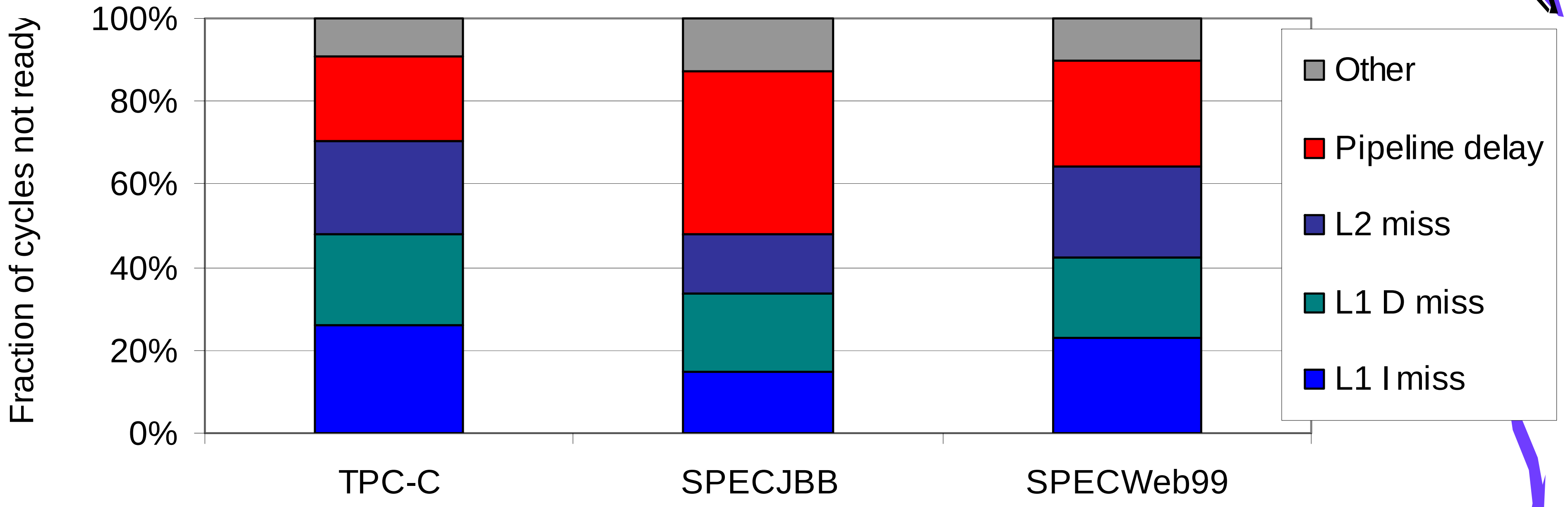
Miss Latency: L2 Cache Size, Block Size



CPI Breakdown of Performance

Benchmark	Per Thread CPI	Per core CPI	Effective CPI for 8 cores	Effective IPC for 8 cores
TPC-C	7.20	1.80	0.23	4.4
SPECJBB	5.60	1.40	0.18	5.7
SPECWeb99	6.60	1.65	0.21	4.8

Not Ready Breakdown

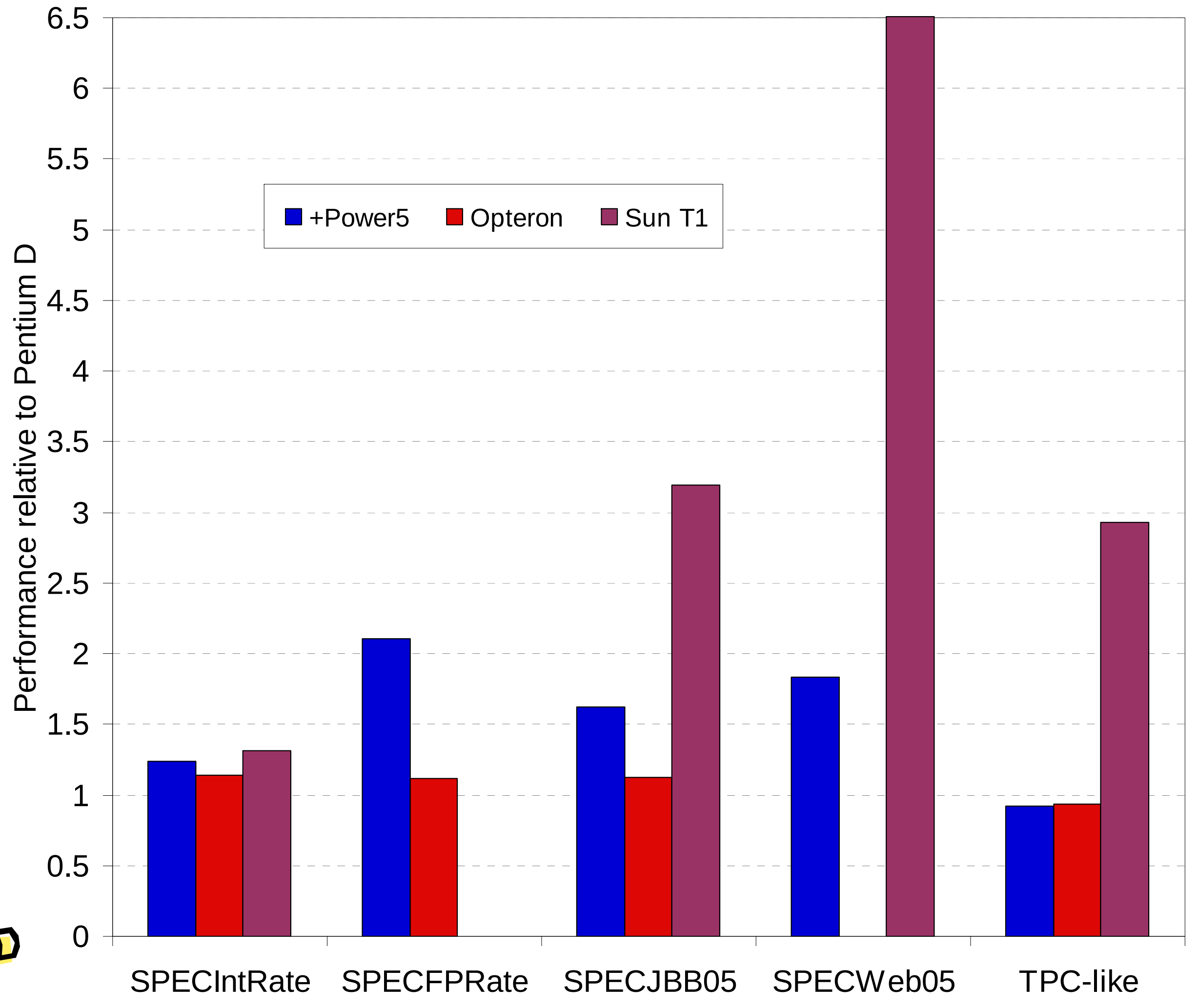


- TPC-C - store buffer full is largest contributor
- SPEC-JBB - atomic instructions are largest contributor
- SPECWeb99 - both factors contribute

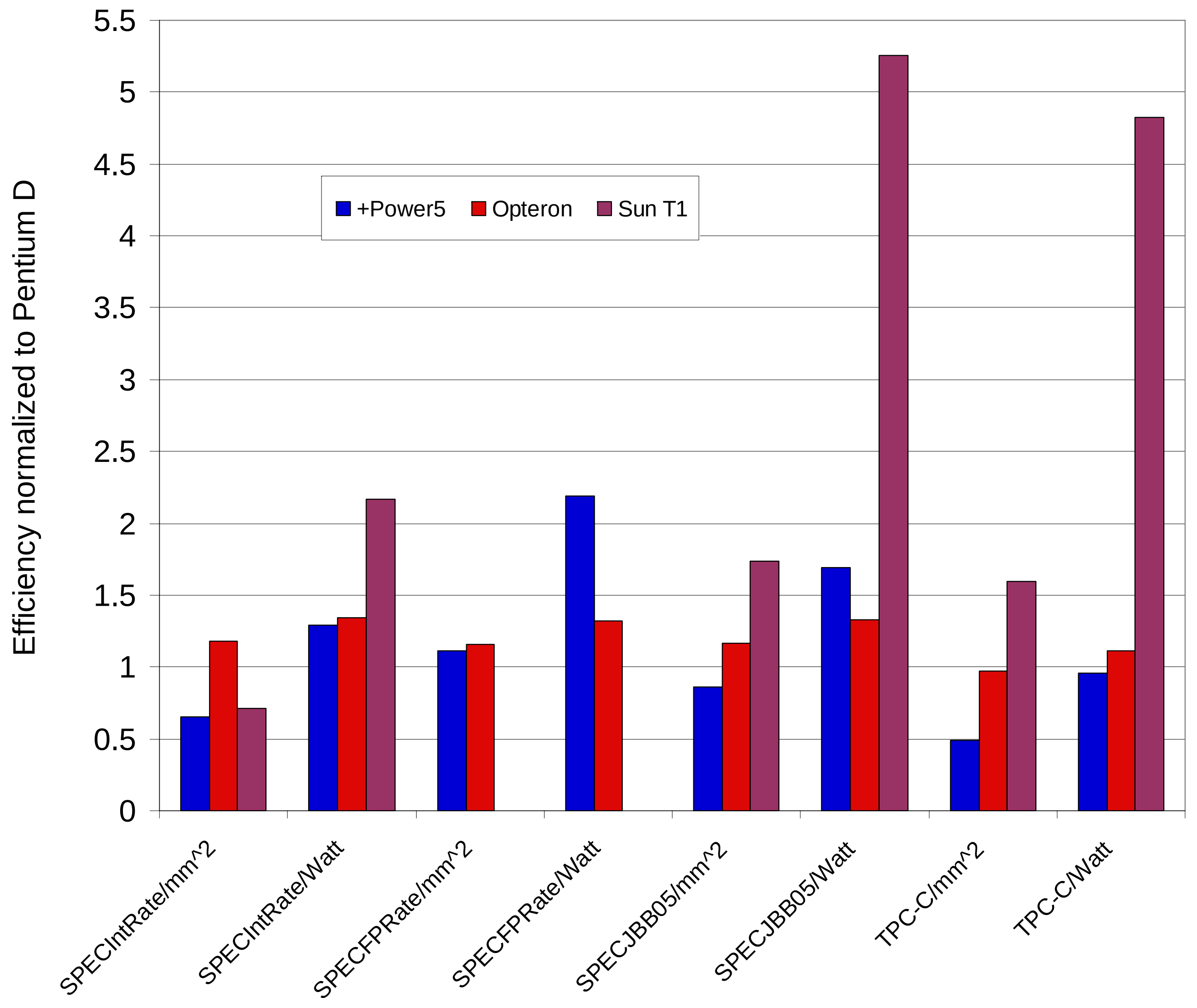
Microprocessor Comparison

- Please refer to Fig. 4.32

Performance Relative to Pentium D



Performance/mm^2, Performance/Watt



Niagara 2

- Improve performance by increasing threads supported per chip from 32 to 64
  - 8 cores * 8 threads per core
- Floating-point unit for each core, not for each chip
- Hardware support for encryption standards EAS, 3DES, and elliptical-curve cryptography
- Niagara 2 will add a number of 8x PCI Express interfaces directly into the chip in addition to integrated 10Gigabit Ethernet XAU interfaces and Gigabit Ethernet ports.
- Integrated memory controllers will shift support from DDR2 to FB-DIMMs and double the maximum amount of system memory.