Chapter 6: Multiprocessors and Thread-Level Parallelism

Parallel Architectures

Flynn's Four Categories

- SISD (Single Instruction Single Data)
  - Uniprocessors
- MISD (Multiple Instruction Single Data)
  - ???; multiple processors on a single data stream
- SIMD (Single Instruction Multiple Data)
  - Examples: Illiac-IV, CM-2
    - Simple programming model
    - Low overhead
    - Flexibility
    - All custom integrated circuits
  - (Phrase reused by Intel marketing for media instructions ~ vector)
- MIMD (Multiple Instruction Multiple Data)
  - Examples: Sun Enterprise 5000, Cray T3D, SGI Origin
    - Flexible
    - *Use off-the-shelf micros*
- MIMD current winner: Concentrate on major design emphasis <= 128 processor MIMD machines

MIMD

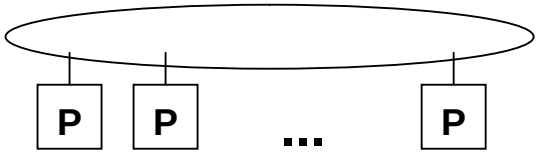
- In order to take advantage of an MIMD multiprocessor with n processors, we must have at least n threads or processes to execute.
  - Thread-level parallelism

Major MIMD Styles

1. Centralized shared memory architectures ("Uniform Memory Access" or "Shared Memory Processor")
   - For multiprocessors with small processor count, it is possible to share a single centralized memory and to interconnect the processors and memory by a bus.
2. Decentralized memory (memory module with CPU)
   - get more memory bandwidth, lower memory latency
   - Drawback: Longer communication latency
   - Drawback: Software model more complex
   - Shared memory model
   - Message passing model

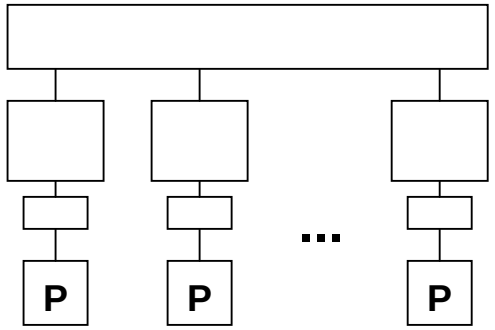
What is a Multiprocessor?

- A collection of communicating processors
  - Goals: balance load, reduce inherent communication and extra work

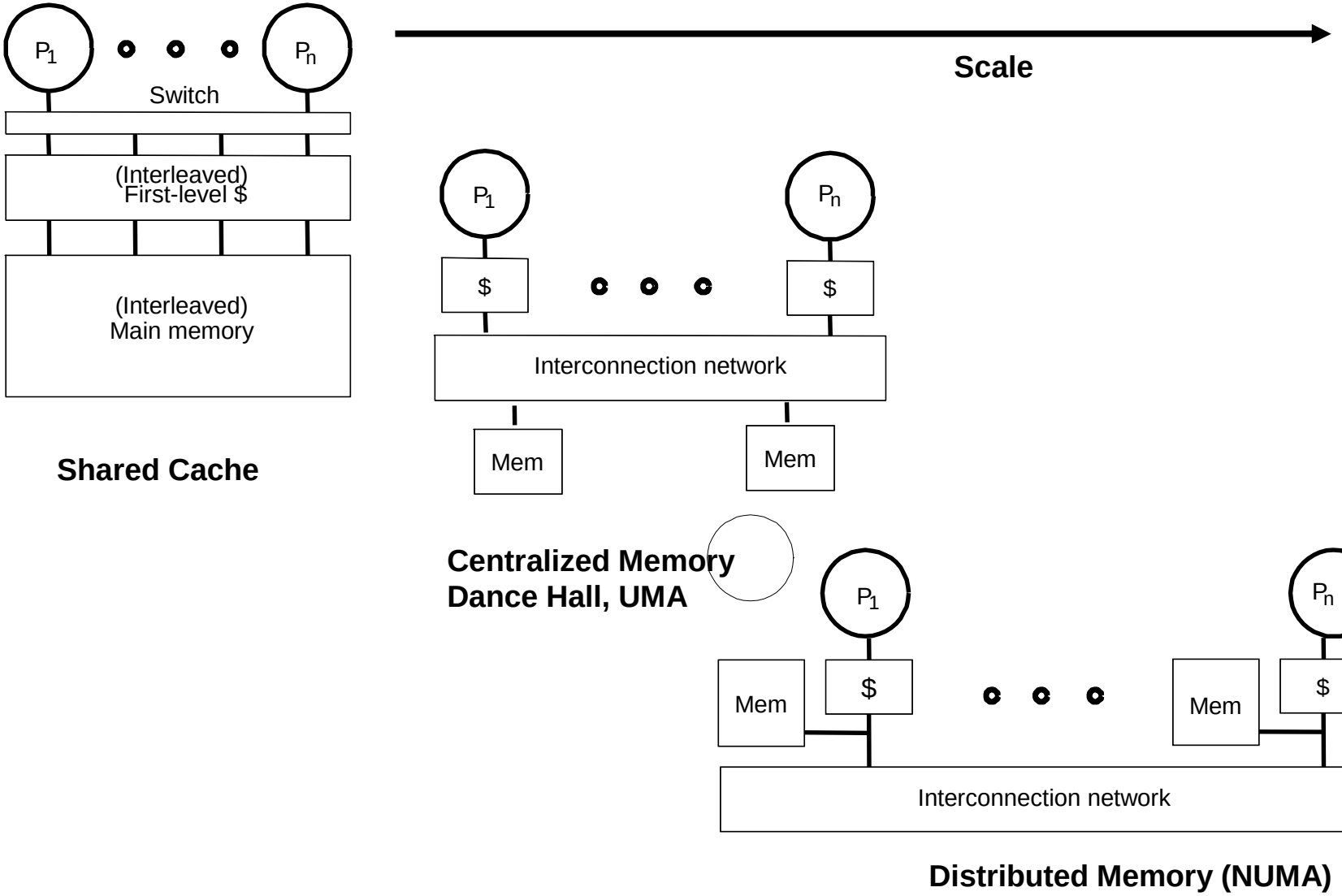


A multi-cache, multi-memory system

- Role of these components essential regardless of programming model
- Prog. model and comm. abstr. affect specific performance tradeoffs



Natural Extensions of Memory System



Shared Cache

Centralized Memory
Dance Hall, UMA

Distributed Memory (NUMA)

Decentralized Memory versions

1. Shared Memory with "Non Uniform Memory Access" time (NUMA)
2. Message passing "multicomputer" with separate address space per processor
   - Can invoke software with Remote Procedue Call (RPC)
   - Often via library, such as MPI: Message Passing Interface
   - Also called "Synchronous communication" since communication causes synchronization between 2 processes

Parallel Architecture

- Parallel Architecture extends traditional computer architecture with a communication architecture
  - abstractions (HW/SW interface)
  - organizational structure to realize abstraction efficiently

Parallel Framework

- Layers:
  - Programming Model:
    - Multiprogramming : lots of jobs, no communication
    - Shared address space: communicate via memory
    - Message passing: send and receive messages
    - Data Parallel: several agents operate on several data sets simultaneously and then exchange information globally and simultaneously (shared or message passing)
  - Communication Abstraction:
    - Shared address space: e.g., load, store, atomic swap
    - Message passing: e.g., send, receive library calls
    - Debate over this topic (ease of programming, scaling)
      => many hardware designs 1:1 programming model

Shared address space model

Shared Address Model Summary

- Each processor can name every physical location in the machine
- Each process can name all data it shares with other processes
- Data transfer via load and store
- Data size: byte, word, ... or cache blocks
- Uses virtual memory to map virtual to local or remote physical
- Memory hierarchy model applies: now communication moves data to local processor cache (as load moves data from memory to cache)
  - Latency, BW, scalability when communicate?

Message passing model

Message Passing Model

- Whole computers (CPU, memory, I/O devices) communicate as explicit I/O operations
  - Essentially NUMA but integrated at I/O devices vs. memory system
- Send specifies local buffer + receiving process on remote computer
- Receive specifies sending process on remote computer + local buffer to place data
  - Usually send includes process tag
    and receive has rule on tag: match 1, match any
  - Sync: when send completes, when buffer free, when request accepted, receive wait for send
- Send+receive => memory-memory copy, where each supplies local address, AND does pairwise sychronization!

Performance Metrics: Latency and Bandwidth

1. Bandwidth
   - Need high bandwidth in communication
   - Match limits in network, memory, and processor
   - Challenge is link speed of network interface vs. bisection bandwidth of network
2. Latency
   - Affects performance, since processor may have to wait
   - Affects ease of programming, since requires more thought to overlap communication and computation
   - Overhead to communicate is a problem in many machines
3. Latency Hiding
   - How can a mechanism help hide latency?
   - Increases programming system burden
   - Examples: overlap message send with computation, prefetch data, switch to other tasks

Data parallel model

Data Parallel Model

- Operations can be performed in parallel on each element of a large regular data structure, such as an array
- 1 Control Processor broadcast to many PEs
- When computers were large, could amortize the control portion of many replicated PEs
- Condition flag per PE so that can skip
- Data distributed in each memory
- Early 1980s VLSI => SIMD rebirth:
  32 1-bit PEs + memory on a chip was the PE
- Data parallel programming languages lay out data to processor

Data Parallel Model

- Vector processors have similar ISAs, but no data placement restriction
- SIMD led to Data Parallel Programming languages (High Performance Fortran and friends)
- Advancing VLSI led to single chip FPUs and whole fast µProcs (SIMD less attractive)
- SIMD programming model led to Single Program Multiple Data (SPMD) model
  - All processors execute identical program
- Data parallel programming languages still useful, do communication all at once: "Bulk Synchronous" phases in which all communicate after a global barrier

Comparing models of parallelism

Advantages shared-memory communication model

- Compatibility with SMP hardware
- Ease of programming when communication patterns are complex or vary dynamically during execution
- Ability to develop apps using familiar SMP model, attention only on performance critical accesses
- Lower communication overhead, better use of BW for small items, due to implicit communication and memory mapping to implement protection in hardware, rather than through I/O system
- HW-controlled caching to reduce remote comm. by caching of all data, both shared and private.

Advantages message-passing communication model

- The hardware can be simpler (esp. vs. NUMA)
- Communication explicit => simpler to understand; in shared memory it can be hard to know when communicating and when not, and how costly it is
- Explicit communication focuses attention on costly aspect of parallel computation, sometimes leading to improved structure in multiprocessor program
- Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization
- Easier to use sender-initiated communication, which may have some advantages in performance

Review: Communication Models

- Shared Memory
  - Processors communicate with shared address space
  - Easy on small-scale machines
  - Advantages:
    - Model of choice for uniprocessors, small-scale MPs
    - Ease of programming
    - Lower latency
    - Easier to use hardware controlled caching
- Message passing
  - Processors have private memories, communicate via messages
  - Advantages:
    - Less hardware, easier to design
    - Focuses attention on costly non-local operations
- Can support either SW model on either HW base

Amdahl's Law and Parallel Computers

- Amdahl's Law (FracX: original % to be speed up)
  Speedup = 1 / [(FracX/SpeedupX + (1-FracX)]
- A portion is sequential => limits parallel speedup
  - Speedup <= 1/ (1-FracX)
- Ex. What fraction sequential to get 80X speedup from 100 processors? Assume either 1 processor or 100 fully used

80 = 1 / [(FracX/100 + (1-FracX)]

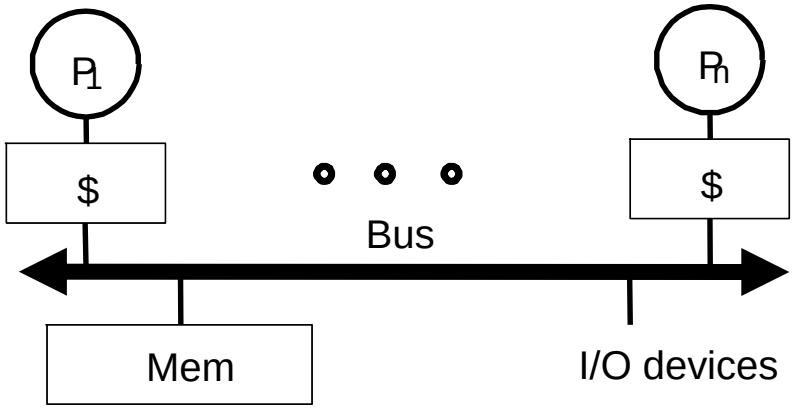
0.8*FracX + 80*(1-FracX) = 80 - 79.2*FracX = 1

FracX = (80-1)/79.2 = 0.9975

- Only 0.25% sequential!

6.3 Symmetric Shared-Memory Architectures

Bus-Based Symmetric Shared Memory



- Dominate the server market
  - Building blocks for larger systems; arriving to desktop
- Attractive as throughput servers and for parallel programs
  - Fine-grain resource sharing
  - Uniform access via loads/stores
  - Automatic data movement and coherent replication in caches
  - Cheap and powerful extension
- Normal uniprocessor mechanisms to access data
  - Key is extension of memory hierarchy to support multiple processors
- Now Chip Multiprocessors

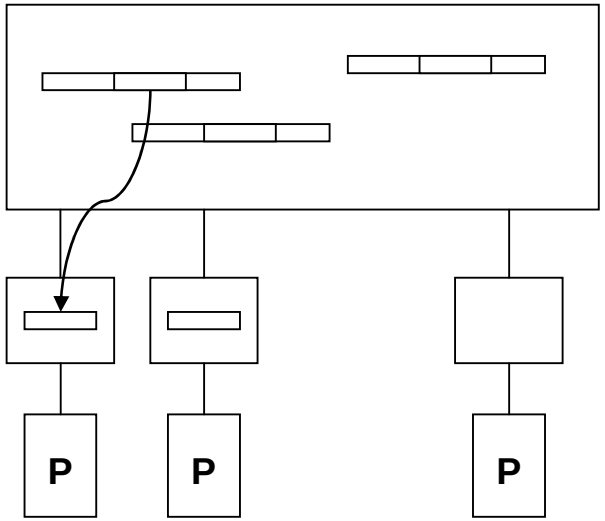
Symmetric Multi-Processors

- The large, multilevel caches reduce the memory demand of a processor
- This allows for more than one processor to share the bus to the memory
- Different organizations:
  - Processor and cache on an extension board
  - Integrated on the mainboard (most popular)
  - Integrated on the same chip
- The new problem: cache coherence

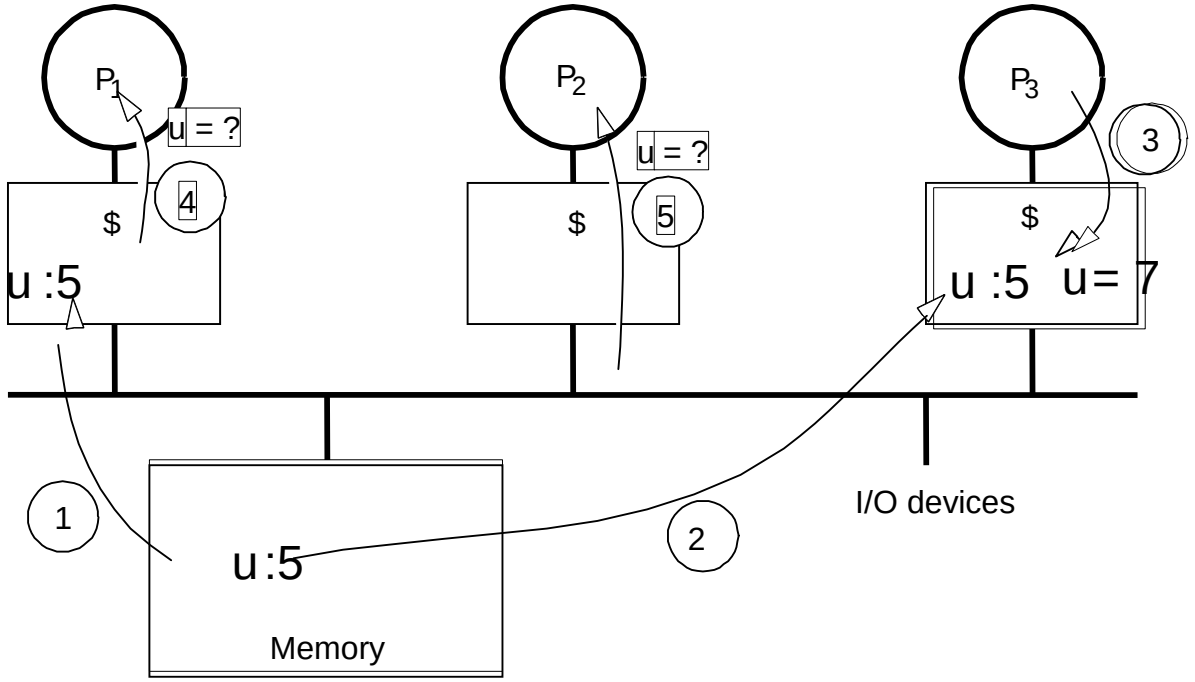
Cache coherence

Caches are Critical for Performance

- Reduce average latency
  - automatic replication closer to processor
- Reduce average bandwidth
- Data is logically transferred from producer to consumer to memory
  - store reg --> mem
  - load reg <-- mem
- Many processors can shared data efficiently
- What happens when store & load are executed on different processors?



Example Cache Coherence Problem



- Processors see different values for u after event 3
- With write-back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  - Processes accessing main memory may see very stale value
- Unacceptable to programs, and frequent!

Caches and Cache Coherence

- Caches play key role in all cases
  - Reduce average data access time
  - Reduce bandwidth demands placed on shared interconnect

- Private processor caches create a problem
  - Copies of a variable can be present in multiple caches
  - A write by one processor may not become visible to others
    - They'll keep accessing stale value in their caches

  $\Rightarrow$ *Cache coherence* problem

- What do we do about it?
  - Organize the mem hierarchy to make it go away
  - Detect and take actions to eliminate the problem

Cache Coherence

Shared-memory multiprocessors:

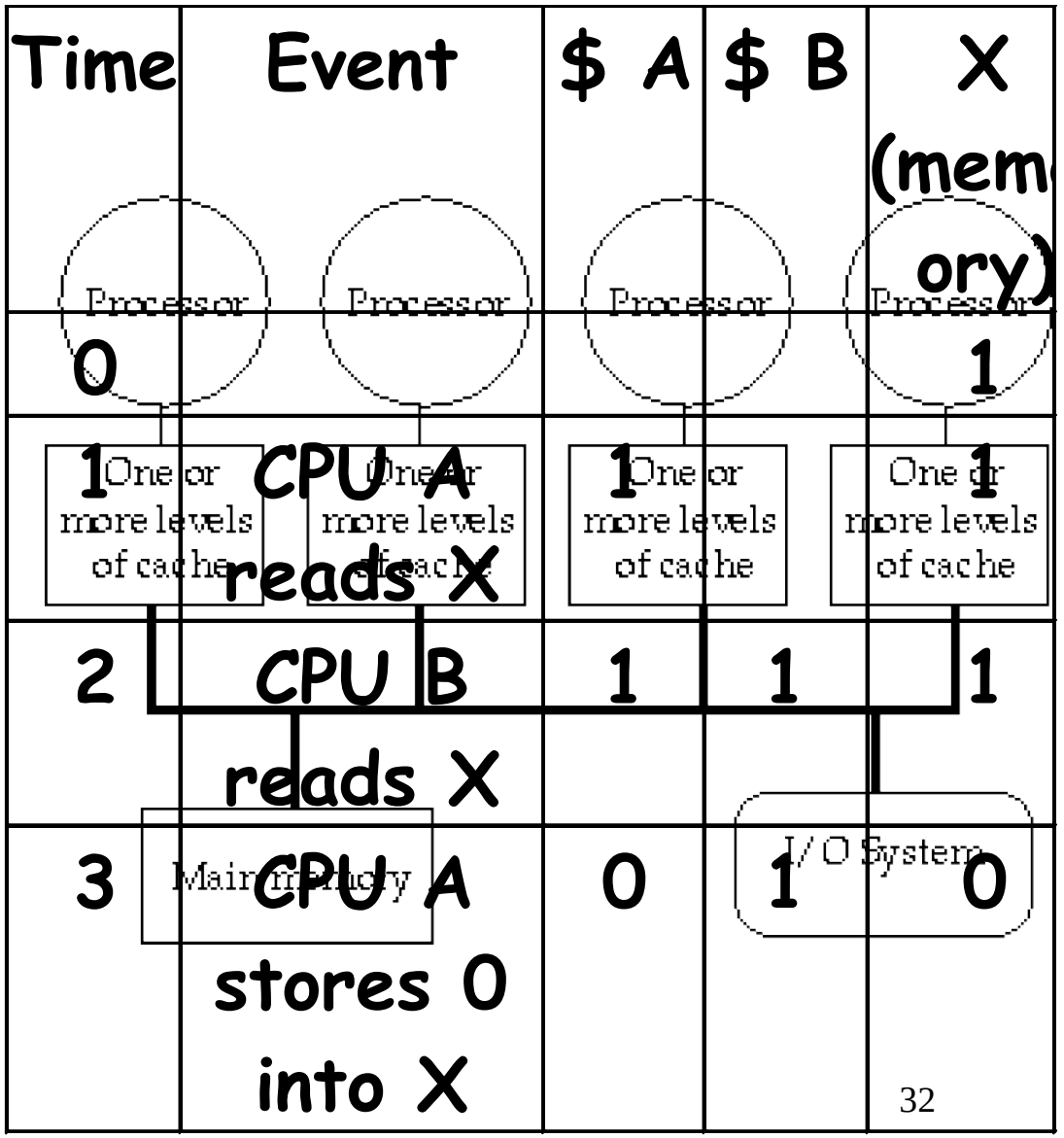
- Processes in different processors can use the same virtual address space
- Two types of data:
  - Private: only accessed by one processor
  - Shared: can be accessed by all processors

Problem: cache coherence

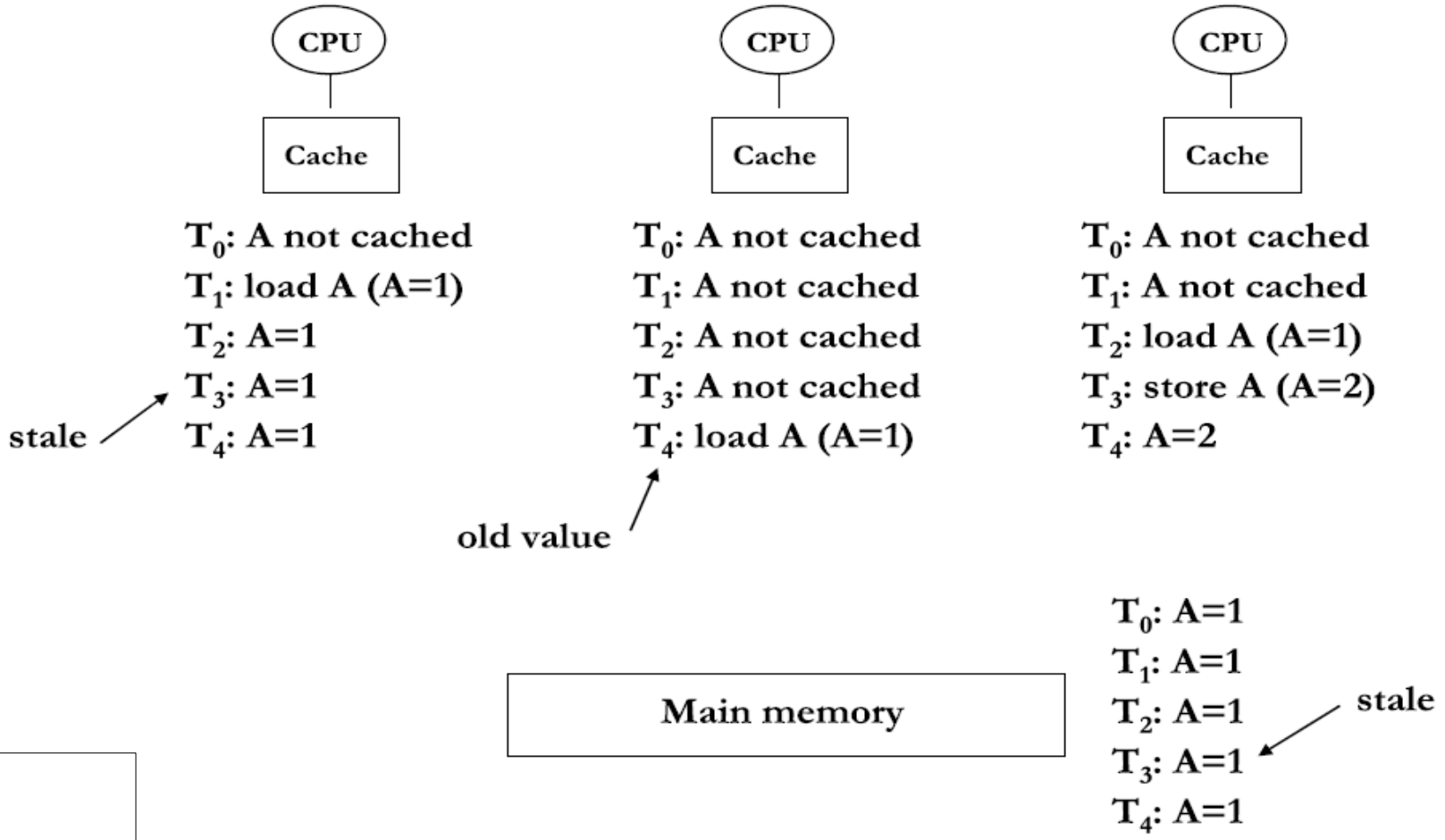
- Shared data may be cached in two different caches
- After a write by one processor the cached data in another processor becomes stale
- With write-back caches data in main memory also becomes stale after a write
- Incorrect program execution if processor uses stale data

Small-Scale—Shared Memory

- Caches serve to:
  - Increase bandwidth versus bus/memory
  - Reduce latency of access
  - Valuable for both private data and shared data
- What about cache consistency?



Example


CPU
Cache
T_0: A not cached
T_1: load A (A=1)
T_2: A=1
T_3: A=1
T_4: A=1
stale
CPU
Cache
T_0: A not cached
T_1: A not cached
T_2: A not cached
T_3: A not cached
T_4: load A (A=1)
old value
CPU
Cache
T_0: A not cached
T_1: A not cached
T_2: load A (A=1)
T_3: store A (A=2)
T_4: A=2
Main memory
T_0: A=1
T_1: A=1
T_2: A=1
T_3: A=1
T_4: A=1
stale

What Does Coherency Mean?

- Informally:
  - "Any read must return the most recent write"
  - Too strict and too difficult to implement
- Better:
  - "Any write must eventually be seen by a read"
  - All writes are seen in proper order ("serialization")
- Two rules to ensure this:
  - "If P writes x and P1 reads it, P's write will be seen by P1 if the read and write are sufficiently far apart"
  - Writes to a single location are serialized: seen in one order
    - Latest write will be seen
    - Otherwise could see writes in illogical order (could see older value after a newer value)

Potential HW Coherency Solutions

- Snooping Solution (Snoopy Bus):
  - Send all requests for data to all processors
  - Processors snoop to see if they have a copy and respond accordingly
  - Requires broadcast, since caching information is at processors
  - Works well with bus (natural broadcast medium)
  - Dominates for small scale machines (most of the market)
- Directory-Based Schemes (discuss later)
  - Keep track of what is being shared in 1 centralized place (logically)
  - Distributed memory => distributed directory for scalability (avoids bottlenecks)
  - Send point-to-point requests to processors via network
  - Scales better than Snooping
  - Actually existed BEFORE Snooping-based schemes

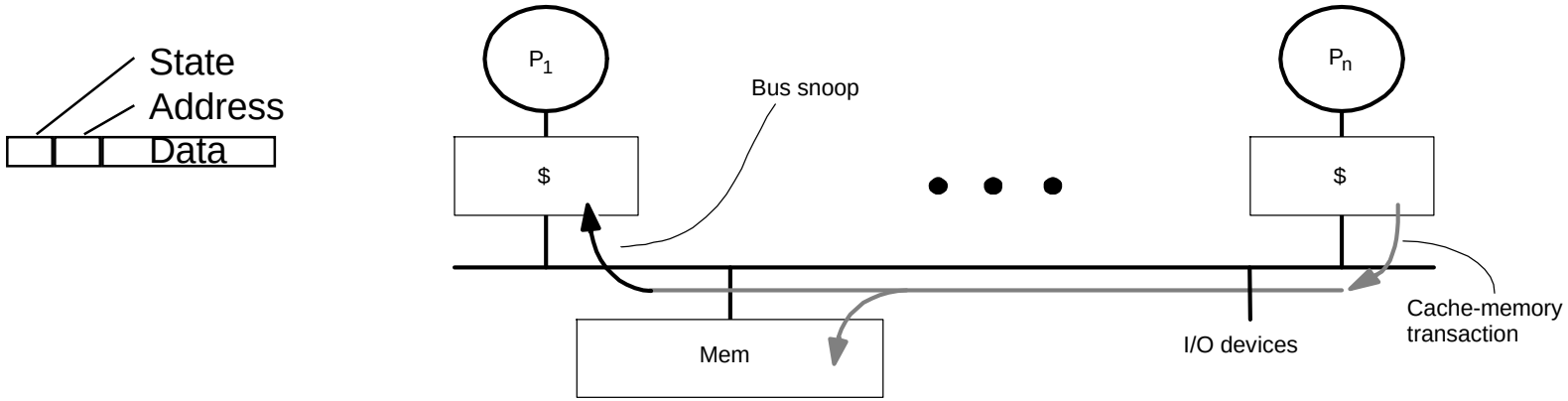
Snooping Protocols- Write Invalidate

- Invalidate other copies on a write
  - All other cached copies of the item are invalidated.

Snooping Protocols- Write Update

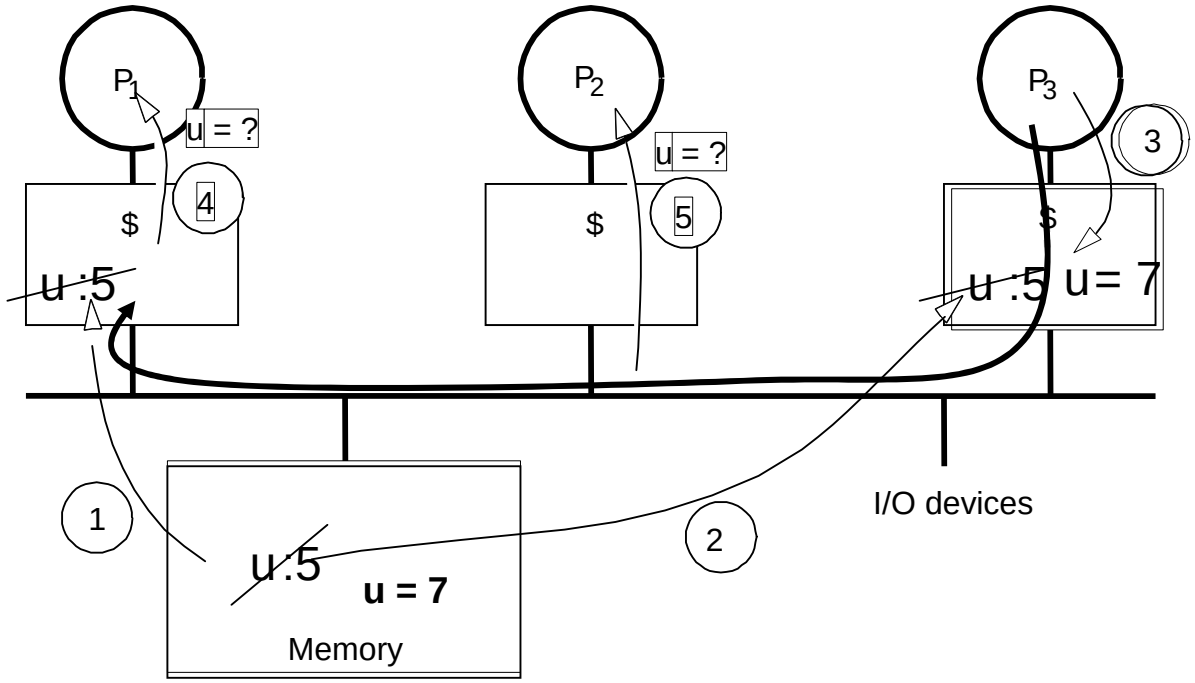
- Update all the cached copies of a data item when that item is written.

Snoopy Cache-Coherence Protocols



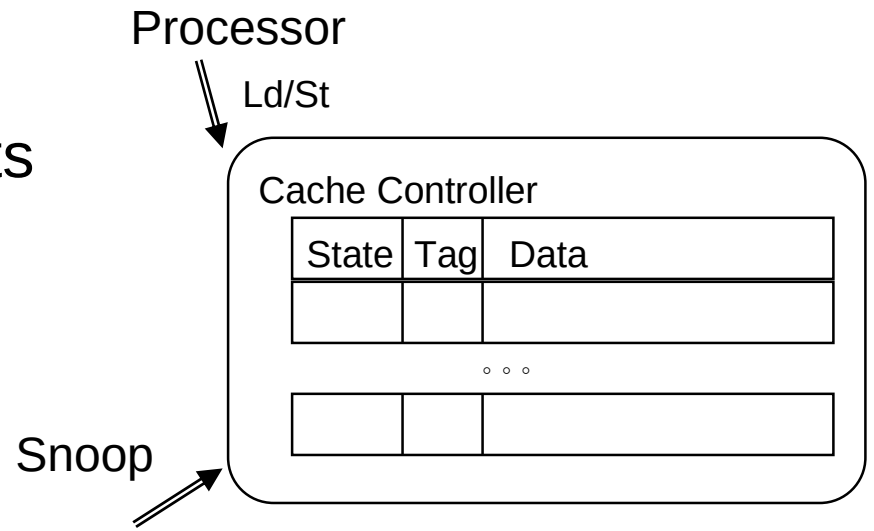
- Bus is a broadcast medium & Caches know what they have
- Cache Controller “snoops” all transactions on the shared bus
  - relevant transaction if for a block it contains
  - take action to ensure coherence
    - invalidate, update, or supply value
  - depends on state of the block and the protocol

Example: Write-thru Invalidate



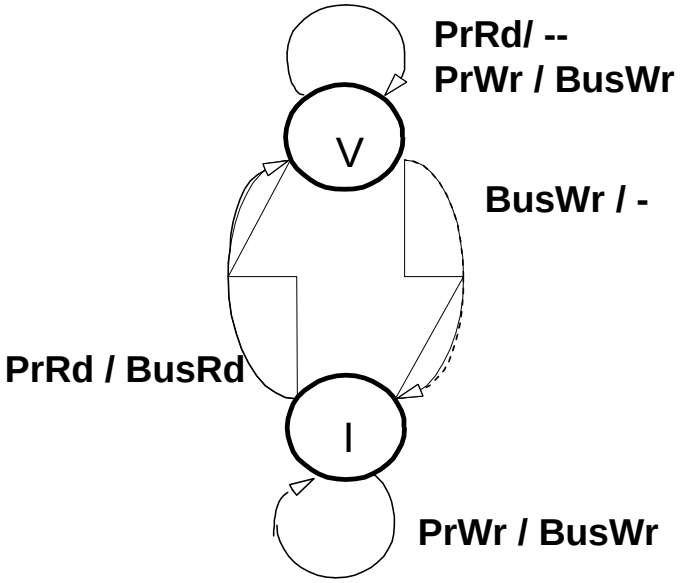
Design Choices

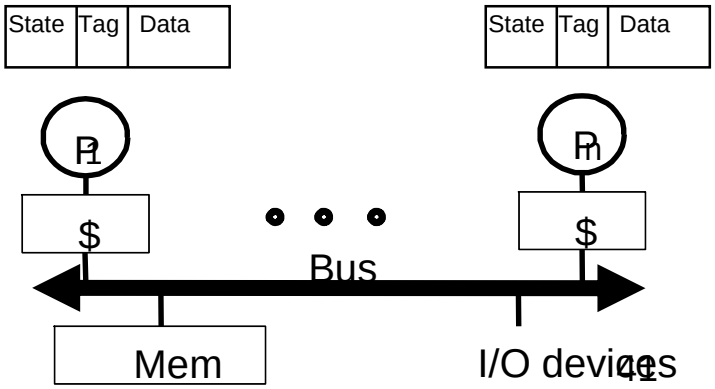
- Controller updates state of blocks in response to processor and snoop events and generates bus transactions
- Snoopy protocol
  - set of states
  - state-transition diagram
  - actions
- Basic Choices
  - Write-through vs Write-back
  - Invalidate vs. Update



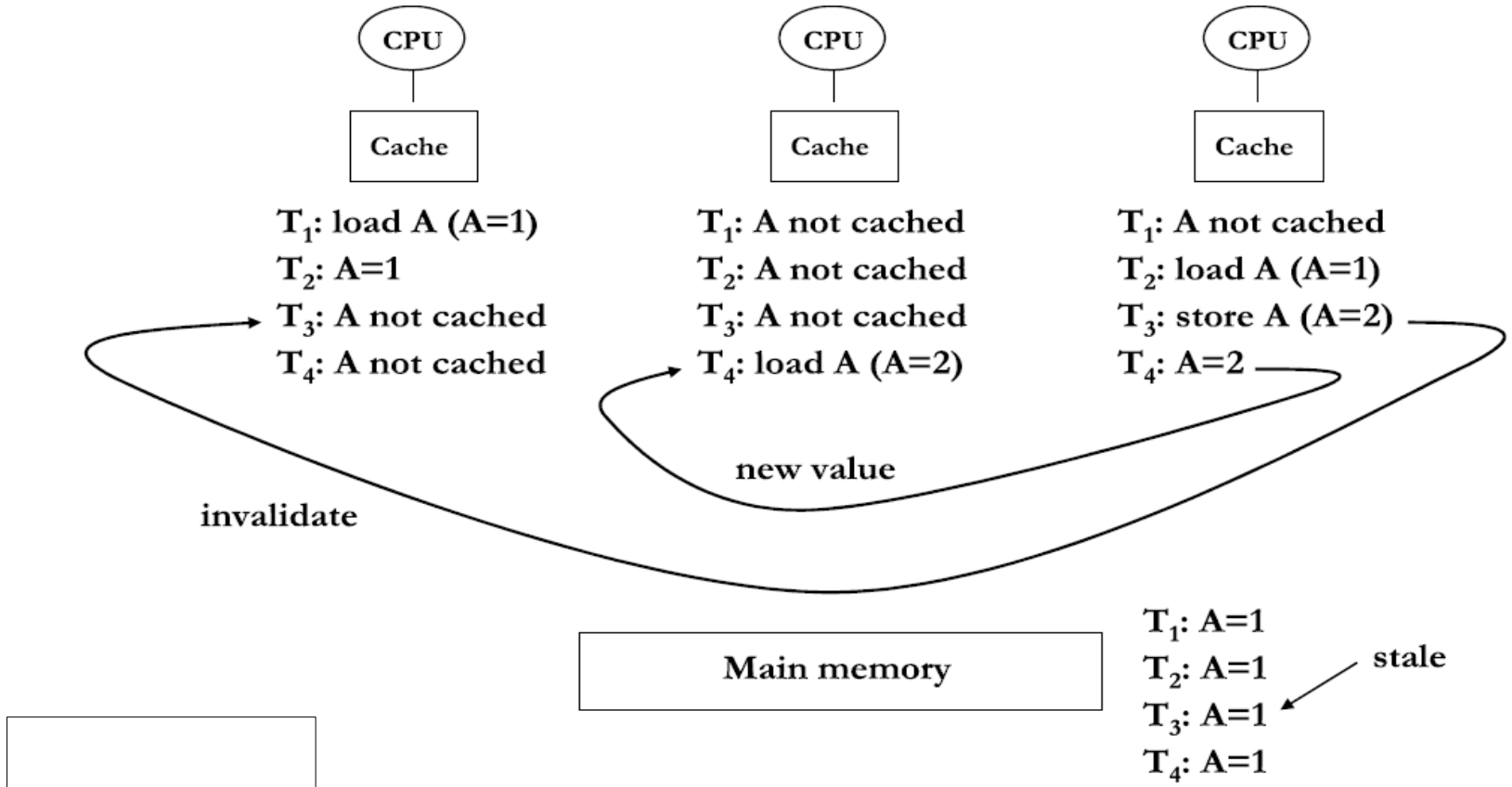
Write-through Invalidate Protocol

- Two states per block in each cache
  - as in uniprocessor
  - state of a block is a *p*-vector of states
  - Hardware state bits associated with blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache
- Writes invalidate all other caches
  - can have multiple simultaneous readers of block,but write invalidates them

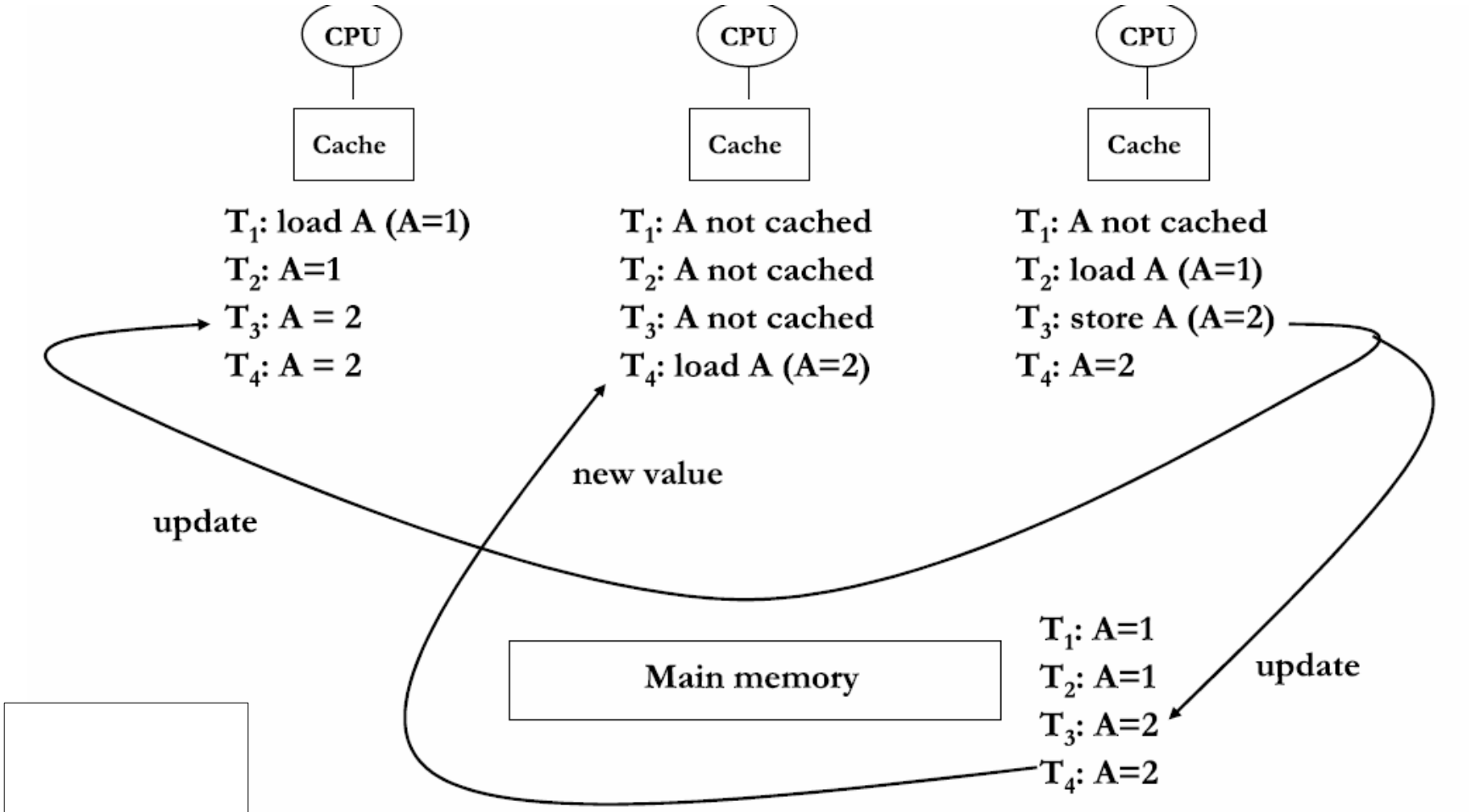




Write-Invalidate Example



Write-Update Example



Write-through vs. Write-back

- Write-through protocol is simple
  - every write is observable
- Every write goes on the bus

  => Only one write can take place at a time in any processor
- Uses a lot of bandwidth!

Example: 200 MHz dual issue, CPI = 1, 15% stores of 8 bytes

=> 30 M stores per second per processor

=> 240 MB/s per processor

1GB/s bus can support only about 4 processors without saturating

Invalidate vs. Update

- Basic question of program behavior:
  - Is a block written by one processor later read by others before it is overwritten?
- Invalidate.
  - yes: readers will take a miss
  - no: multiple writes without addition traffic
    - also clears out copies that will never be used again
- Update.
  - yes: avoids misses on later references
  - no: multiple useless updates
    - even to pack rats

=> Need to look at program reference patterns and hardware complexity

but first - correctness

Invalidate vs. Update Protocols

Invalidate:

+ Multiple writes by the same processor to the cache block only require one invalidation
+ No need to send the new value of the data (less bandwidth)
– Caches must be able to provide up-to-date data upon request
– Must write-back data to memory when evicting a modified block

Usually used with write-back caches (more popular)

Update:

+ New value can be re-used without the need to ask for it again
+ Data can always be read from memory
+ Modified blocks can be evicted from caches silently
– Possible multiple useless updates (more bandwidth)

Usually used with write-through caches (less popular)

Basic Snoopy Protocols

- Write Invalidate Protocol:
  - Multiple readers, single writer
  - Write to shared data: an invalidate is sent to all caches which snoop and *invalidate* any copies
  - Read Miss:
    - Write-through: memory is always up-to-date
    - Write-back: snoop in caches to find most recent copy
- Write Broadcast Protocol (typically write through):
  - Write to shared data: broadcast on bus, processors snoop, and *update* any copies
  - Read miss: memory is always up-to-date
- Write serialization: bus serializes requests!
  - Bus is single point of arbitration

Basic Snoopy Protocols

- Write Invalidate versus Broadcast:
  - Invalidate requires one transaction per write-run
  - Invalidate uses spatial locality: one transaction per block
  - Broadcast has lower latency between write and read

Snooping Cache Variations

Basic Protocol	Berkeley Protocol	Illinois Protocol	MESI Protocol
Exclusive	Owned Exclusive	Private Dirty	Modfied (private,!=Memory)
	Owned Shared	Private Clean	eXclusive (private,=Memory)
Shared	Shared	Shared	Shared (shared,=Memory)
Invalid	Invalid	Invalid	Invalid

Owner can update via bus invalidate operation
Owner must write back when replaced in cache

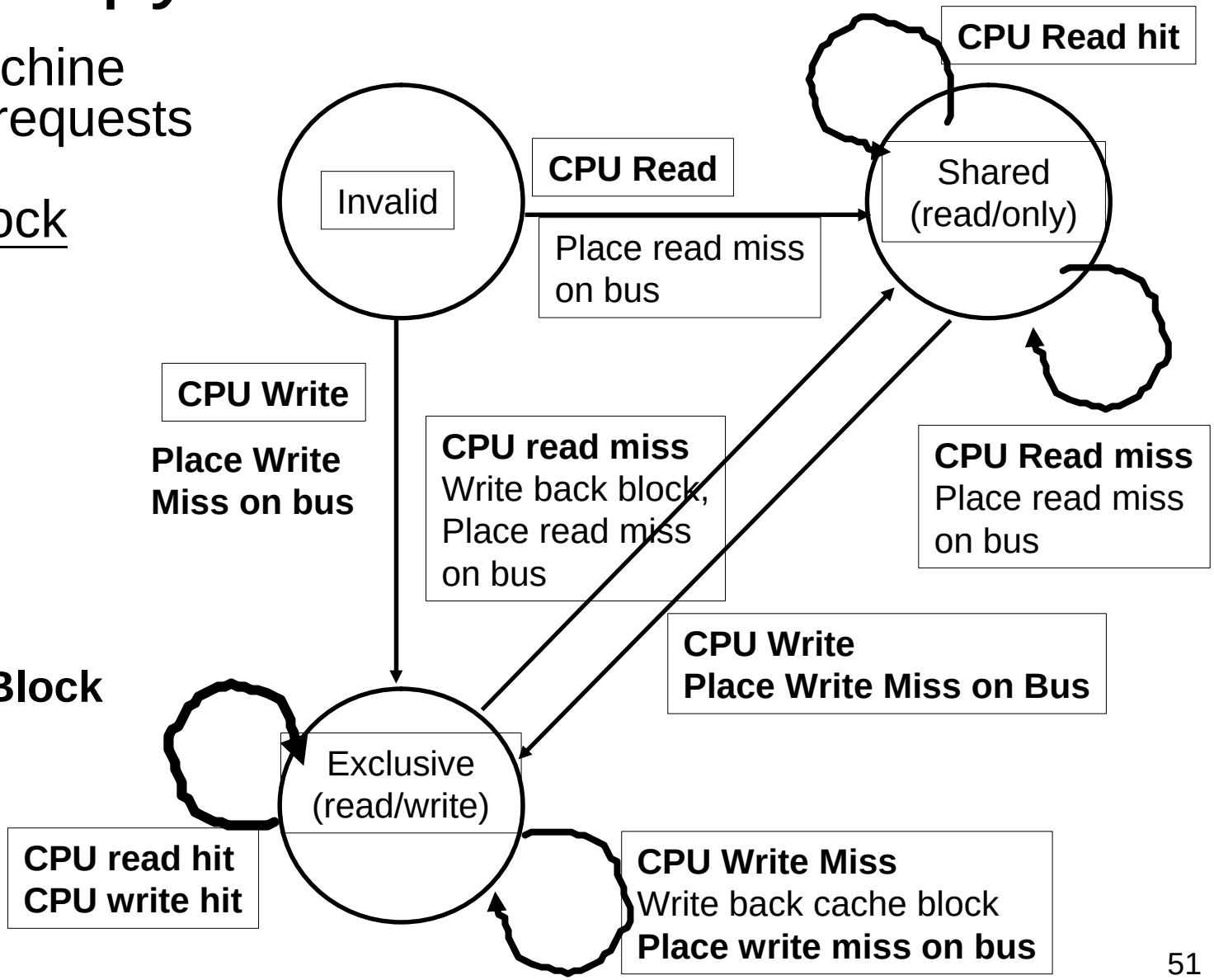
If read sourced from memory, then Private Clean
if read sourced from other cache, then Shared
Can write in cache if held private clean or dirty

An Example Snoopy Protocol

- Invalidation protocol, write-back cache
- Each block of memory is in one state:
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Dirty in exactly one cache (Exclusive)
  - OR Not in any caches
- Each cache block is in one state (track these):
  - Shared : block can be read
  - OR Exclusive : cache has only copy, its writeable, and dirty
  - OR Invalid : block contains no data
- Read misses: cause all caches to snoop bus
- Writes to clean line are treated as misses

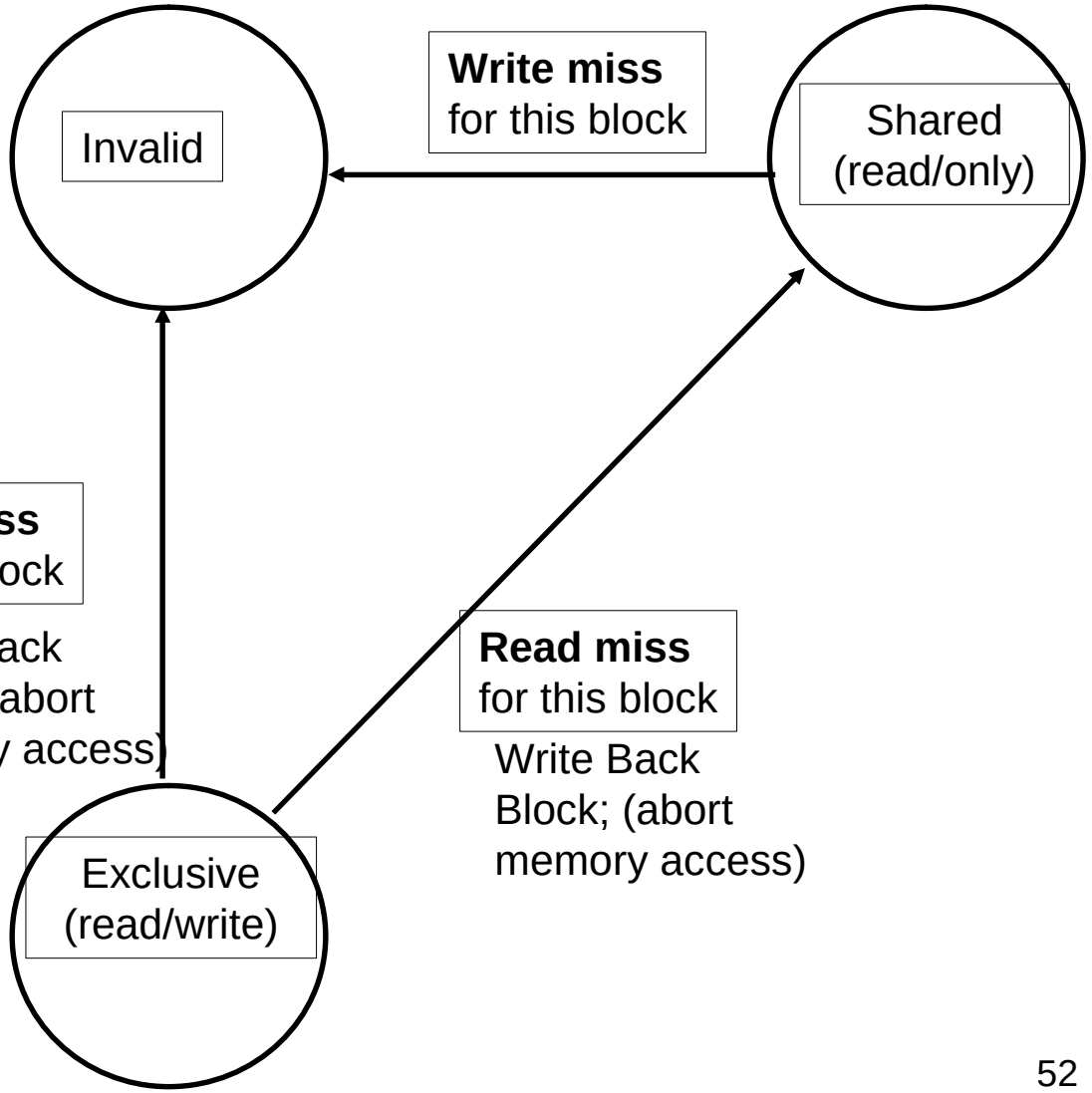
Snoopy-Cache State Machine-I

- State machine for *CPU* requests for each cache block



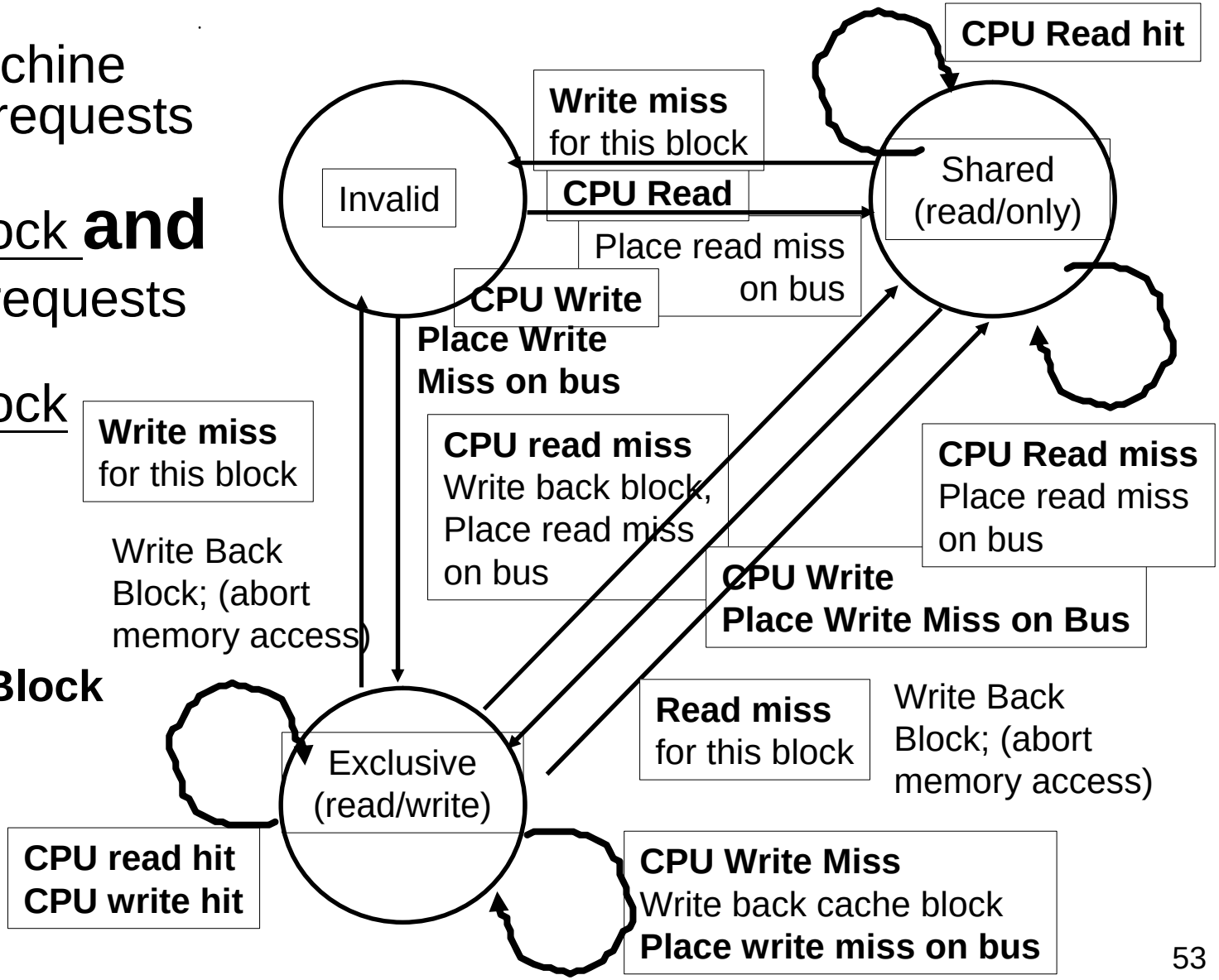
Snoopy-Cache State Machine-II

- State machine for *bus* requests for each cache block
- Appendix E? gives details of bus requests



Snoopy-Cache State Machine-III

- State machine for *CPU* requests for each cache block **and** for *bus* requests for each cache block



Example

step	P1 State	Addr	Value	P2 State	Addr	Value	Bus Action	Proc.	Addr	Value	Memory Addr	Memory Value
P1 Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block, initial cache state is invalid

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

step	P1 State	Addr	Value	P2 State	Addr	Value	Bus Action	Proc.	Addr	Value	Memory Addr	Memory Value
P1 Write 10 to A1	*Excl.*	*A1*	*10*				*WrMs*	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

step	P1 State	Addr	Value	P2 State	Addr	Value	Bus Action	Proc.	Addr	Value	Memory Add	Memory Valu
P1 Write 10 to A1	*Excl.*	*A1*	*10*				*WrMs*	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				*Shar.*	*A1*		*RdMs*	P2	A1			
	Shar.	A1	10				*WrBk*	P1	A1	10	A1	*10*
				Shar.	A1	*10*	*RdDa*	P2	A1	10	A1	10
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

step	P1 State	Addr	Value	P2 State	Addr	Value	Bus Action	Proc.	Addr	Value	Memory Add	Memory Valu
P1 Write 10 to A1	*Excl.*	*A1*	*10*				*WrMs*	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				*Shar.*	*A1*		*RdMs*	P2	A1			
	Shar.	A1	10				*WrBk*	P1	A1	10	A1	*10*
				Shar.	A1	*10*	*RdDa*	P2	A1	10	A1	10
P2: Write 20 to A1	*Inv.*			*Excl.*	A1	*20*	*WrMs*	P2	A1		A1	10
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Add	Valu
P1 Write 10 to A	*Excl.*	*A1*	*10*				*WrMs*	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				*Shar.*	*A1*		*RdMs*	P2	A1			
	Shar.	A1	10				*WrBk*	P1	A1	10	A1	*10*
				Shar.	A1	*10*	*RdDa*	P2	A1	10	A1	10
P2: Write 20 to A	*Inv.*			*Excl.*	A1	*20*	*WrMs*	P2	A1		A1	10
P2: Write 40 to A							*WrMs*	P2	A2		A1	10
				Excl.	*A2*	*40*	*WrBk*	P2	A1	20	A1	*20*

Assumes A1 and A2 map to same cache block,
but A1 != A2

Implementation Complications

- Write Races:
  - Cannot update cache until bus is obtained
    - Otherwise, another processor may get bus first, and then write the same cache block!
  - Two step process:
    - Arbitrate for bus
    - Place miss on bus and complete operation
  - If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
  - Split transaction bus:
    - Bus transaction is not atomic: can have multiple outstanding transactions for a block
    - Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
    - Must track and prevent multiple misses for one block
- Must support interventions and invalidations

Implementing Snooping Caches

- Multiple processors must be on bus, access to both addresses and data
- Add a few new commands to perform coherency, in addition to read and write
- Processors continuously snoop on address bus
  - If address matches tag, either invalidate or update
- Since every bus transaction checks cache tags, could interfere with CPU just to check:
  - solution 1: duplicate set of tags for L1 caches just to allow checks in parallel with CPU
  - solution 2: L2 cache already duplicate, provided L2 obeys inclusion with L1 cache
    - block size, associativity of L2 affects L1

Implementing Snooping Caches

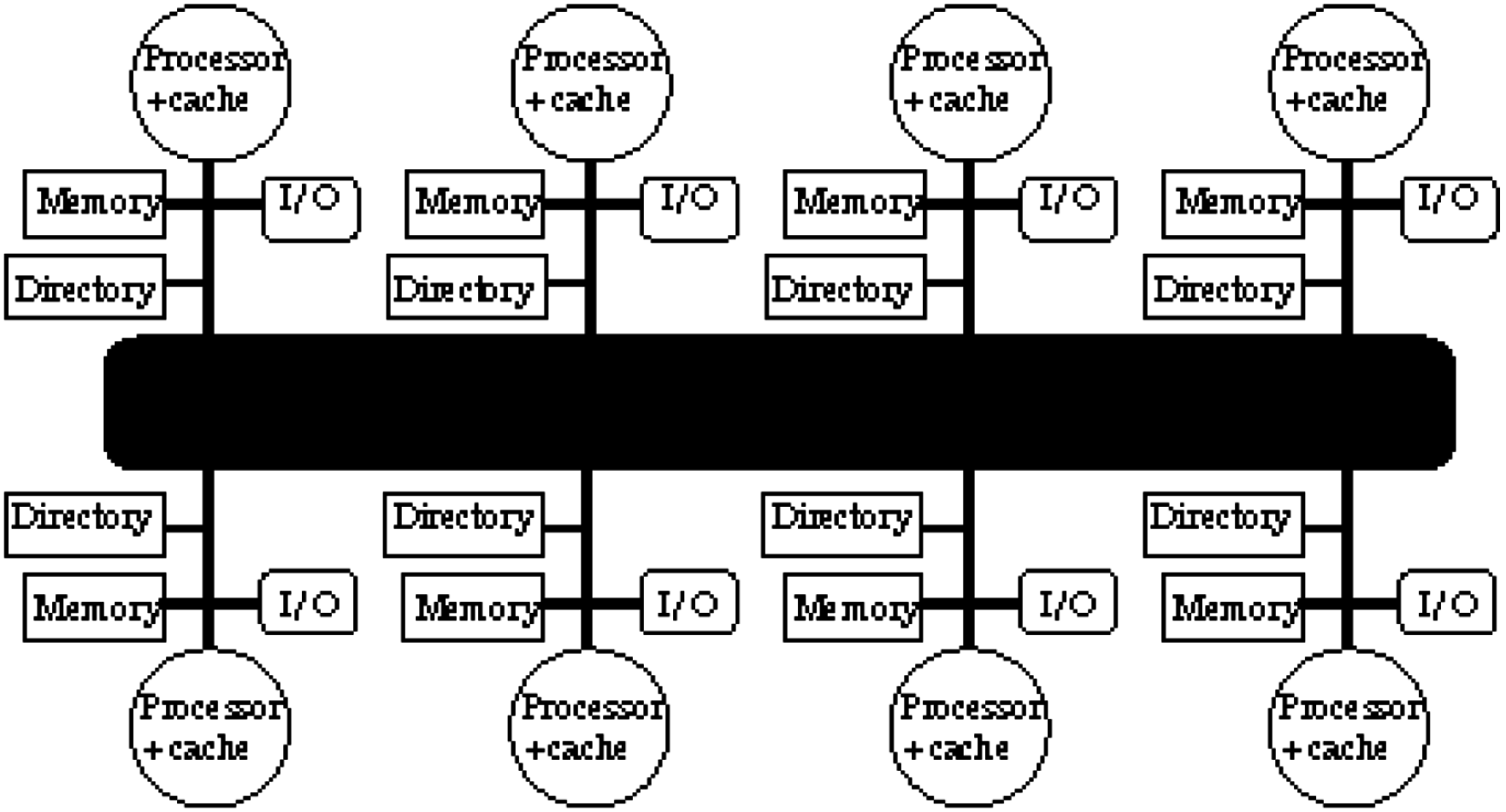
- Bus serializes writes, getting bus ensures no one else can perform memory operation
- On a miss in a write back cache, may have the desired copy and its dirty, so must reply
- Add extra state bit to cache to determine shared or not

6.5 Distributed Shared-Memory Architectures

Larger MPs

- Separate Memory per Processor
- Local or Remote access via memory controller
- 1 Cache Coherency solution: non-cached pages
- Alternative: directory per cache that tracks state of every block in every cache
  - Which caches have a copies of block, dirty vs. clean, ...
- Info per memory block vs. per cache block?
  - PLUS: In memory => simpler protocol (centralized/one location)
  - MINUS: In memory => directory is *f*(memory size) vs. *f*(cache size)
- Prevent directory as bottleneck?
  distribute directory entries with memory, each keeping track of which Procs have copies of their blocks

Distributed Directory MPs



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

Directory Protocol Messages

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 arrange to send data back*

Write miss	Local cache	Home directory	P, A

– *Processor P writes data at address A; make P the exclusive owner and arrange to send data back*

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*

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

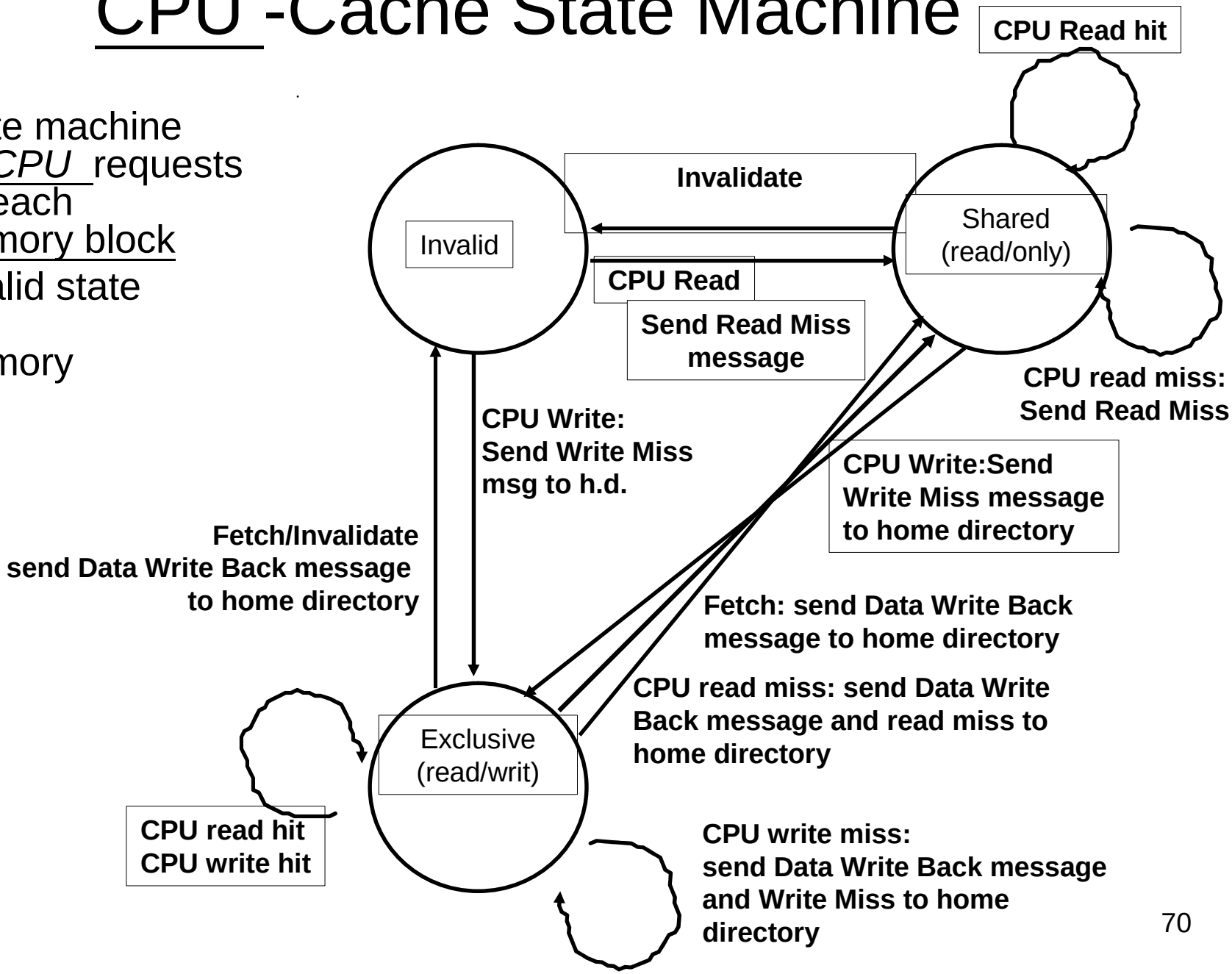
– *Write-back a data value for address A (invalidate response)*

State Transition Diagram for an Individual Cache Block in a 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

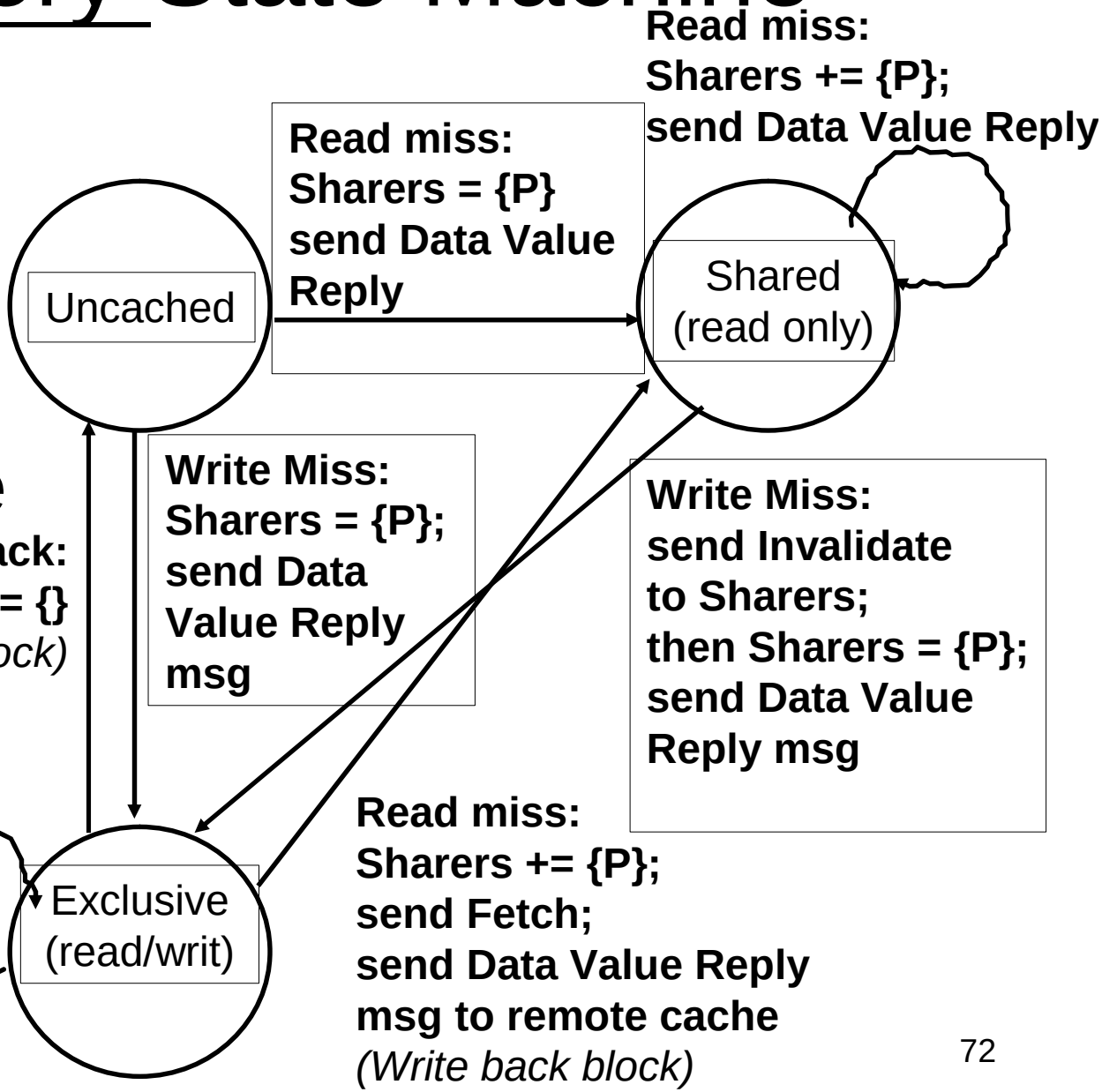


State Transition Diagram for the Directory

- Same states & structure as the transition diagram for an individual cache
- 2 actions: update of directory state & send msgs to statisfy 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.

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
- 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

6.7 Synchronization

Synchronization

- Why Synchronize? Need to know when it is safe for different processes to use shared data
- Issues for Synchronization:
  - Uninterruptible instruction to fetch and update memory (atomic operation);
  - User level synchronization operation using this primitive;
  - For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

Uninterruptible 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

Uninterruptible 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 preceeding 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

```
                li      R2,#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   ;not free=>spin
                exch    R2,0(R1)    ;atomic exchange
                bnez    R2,try      ;already locked?
```

6.8 Models of Memory Consistency

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 before ifs above
  - SC: delay all memory accesses until all invalidates done

6.9 Multithreading

Concept

- Data Access Latency
  - Cache misses (L1, L2)
  - Memory latency (remote, local)
  - Often unpredictable
- Multithreading (MT)
  - Tolerate or mask long and often unpredictable latency operations by switching to another context, which is able to do useful work.

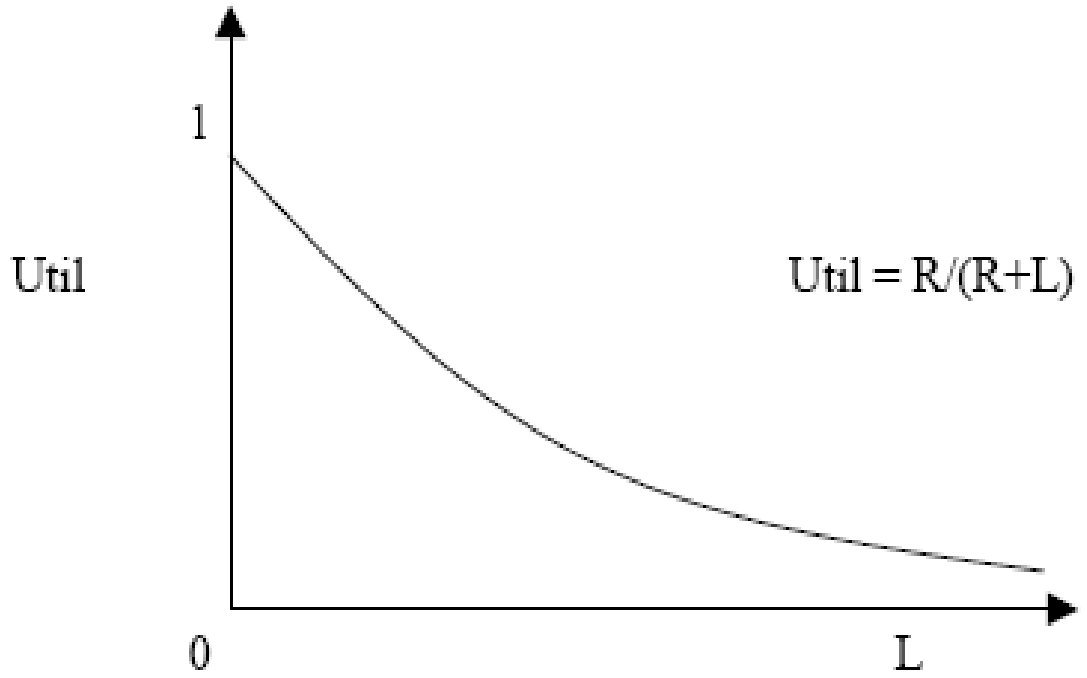
Why Multithreading Today?

- ILP is exhausted, TLP is in.
- Large performance gap bet. MEM and PROC.
- Too many transistors on chip
- More existing MT applications Today.
- Multiprocessors on a single chip.
- Long network latency, too.

Requirements of Multithreading

- Storage need to hold multiple context's PC, registers, status word, etc.
- Coordination to match an event with a saved context
- A way to switch contexts
- Long latency operations must use resources not in use

Processor Utilization vs. Latency



R = the run length to a long latency event

L = the amount of latency

How do the processors communicate?

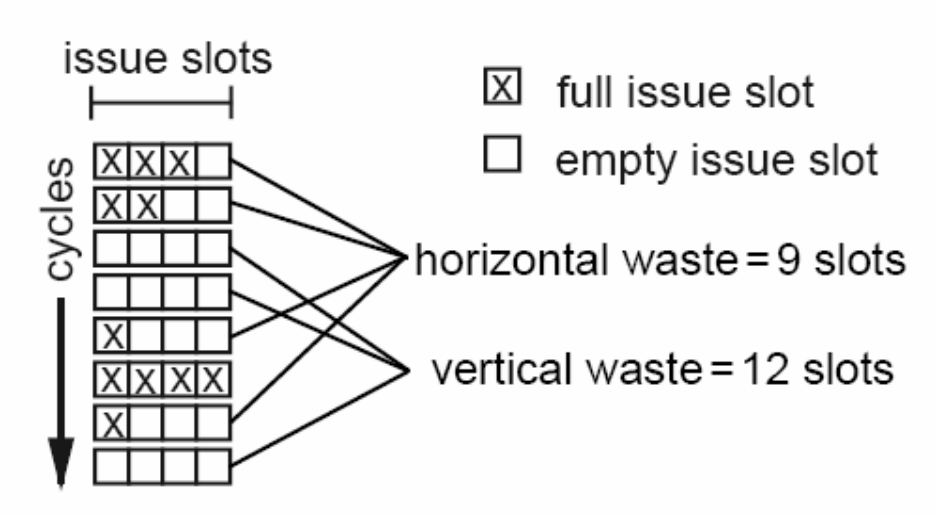
- Shared Memory
- Potential long latency on every load
  - Cache coherency becomes an issue
  - Examples include NYU's Ultracomputer, IBM's RP3, BBN's Butterfly, MIT's Alewife, and later Stanford's Dash.
  - Synchronization occurs through share variables, locks, flags, and semaphores.
- Message Passing
  - Programmer deals with latency. This enables them to minimize the number of messages, while maximizing the size, and this scheme allows for delay minimization by sending a message so that it reaches the receiver at the time it expects it.
  - Examples include Intel's PSC and Paragon, Caltech's Cosmic Cube, and Thinking Machines' CM-5
  - Synchronization occurs through send and receive

Improving Single Thread Performance

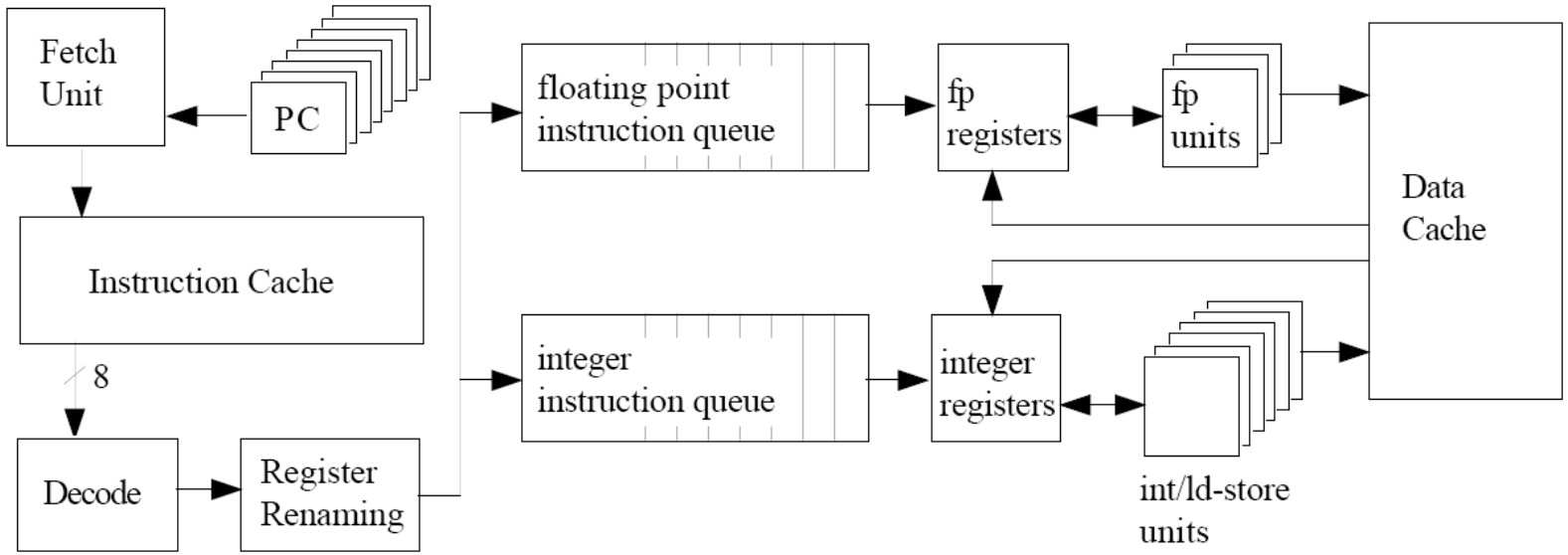
- Do more operations per instruction (VLIW)
- Allow multiple instructions to issue into pipeline from each context.
  - This could lead to pipeline hazards, so other safe instructions could be interleaved into the execution.
  - For Horizon & Tera, the compiler detects such data dependencies and the hardware enforces it by switching to another context if detected.
- Switch on load
- Switch on miss
  - Switching on load or miss will increase the context switch time.

Simultaneous Multithreading (SMT)

- Tullsen, et. al. (U. of Washington), ISCA ‘95
- A way to utilize pipeline with increased parallelism from multiple threads.



Simultaneous Multithreading



SMT Architecture

- Straightforward extension to conventional superscalar design.
  - multiple program counters and some mechanism by which the fetch unit selects one each cycle,
  - a separate return stack for each thread for predicting subroutine return destinations,
  - per-thread instruction retirement, instruction queue flush, and trap mechanisms,
  - a thread id with each branch target buffer entry to avoid predicting phantom branches, and
  - a larger register file, to support logical registers for all threads plus additional registers for register renaming.
    - The size of the register file affects the pipeline and the scheduling of load-dependent instructions.

Commercial Machines w/ MT Support

- Intel Hyperthreding (HT)
  - Dual threads
  - Pentium 4, XEON
- Sun CoolThreads
  - UltraSPARC T1
  - 4-threads per core
- IBM
  - POWER5

IBM Power5

http://www.research.ibm.com/journal/rd/494/mathis.pdf

Table 1 Workloads selected for the study.

Workload	*Computation type*	*SMT gain (%)*
Sentence passing	Integer	41.2
Data compression	Integer	38.6
Programming language	Integer	26.3
3D Multi-grid Solver	Floating-point	21.6
Circuit Routing	Integer	19.8
Seismic Wave Simulation	Floating-point	15.3
Object-oriented Database	Integer	12.5
Neural Network	Floating-point	11.2

IBM Power5

http://www.research.ibm.com/journal/rd/494/mathis.pdf

Table 5 Sources and percentages of data from each source by workload and mode.

Workload	*Mode*	*Data source*		
		L2 cache (%)	*L3 cache* (%)	*Main memory* (%)
Sentence parsing	ST	94	6	0
Sentence parsing	SMT	92	8	0
Data compression	ST	100	0	0
Data compression	SMT	100	0	0
Programming language	ST	94	5	0
Programming language	SMT	97	3	0
3D Multi-grid Solver	ST	95	2	3
3D Multi-grid Solver	SMT	88	6	6
Circuit Routing	ST	79	20	1
Circuit Routing	SMT	72	27	1
Seismic Wave Simulation	ST	86	6	8
Seismic Wave Simulation	SMT	82	4	14
Object-oriented Database	ST	85	14	1
Object-oriented Database	SMT	88	11	1
Neural Network	ST	50	50	0
Neural Network	SMT	49	51	0

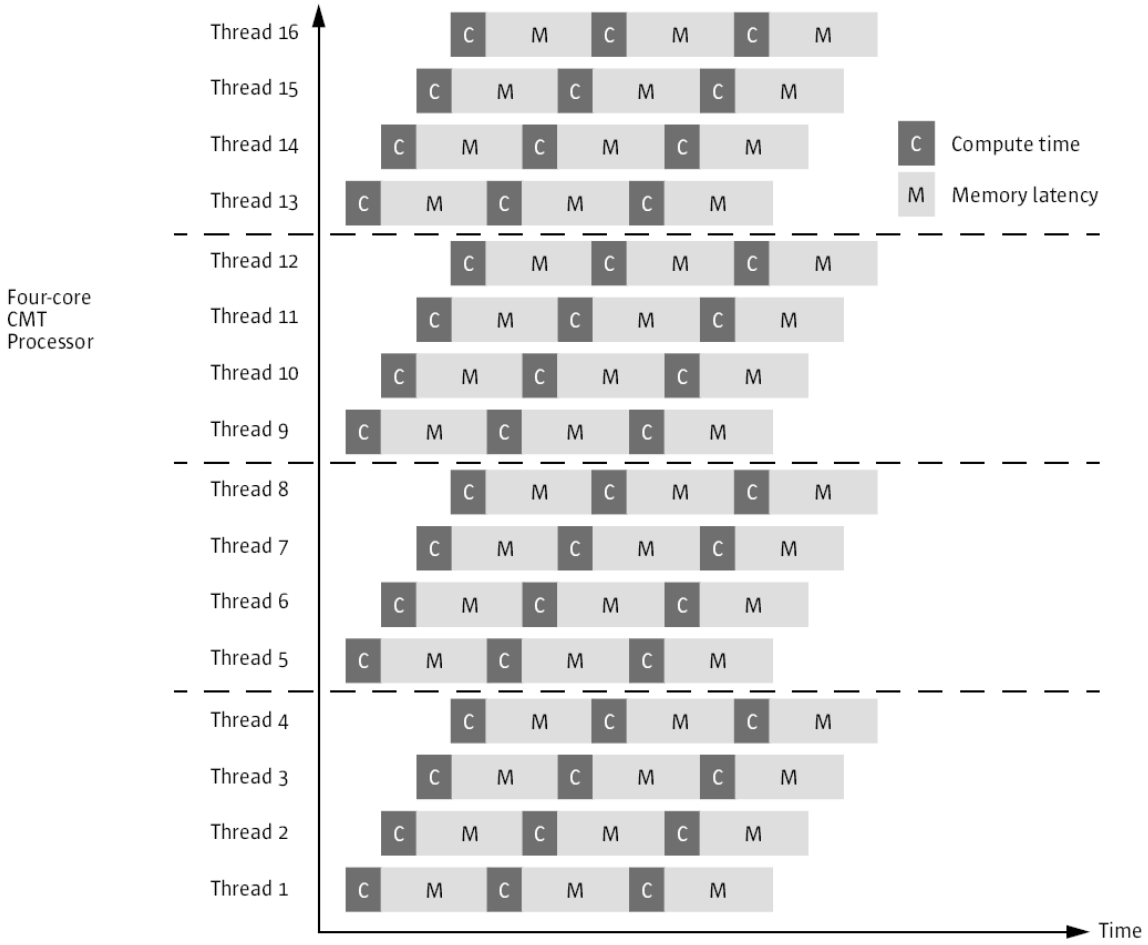
SMT Summary

- Pros:
  - Increased throughput w/o adding much cost
  - Fast response for multitasking environment
- Cons:
  - Slower single processor performance

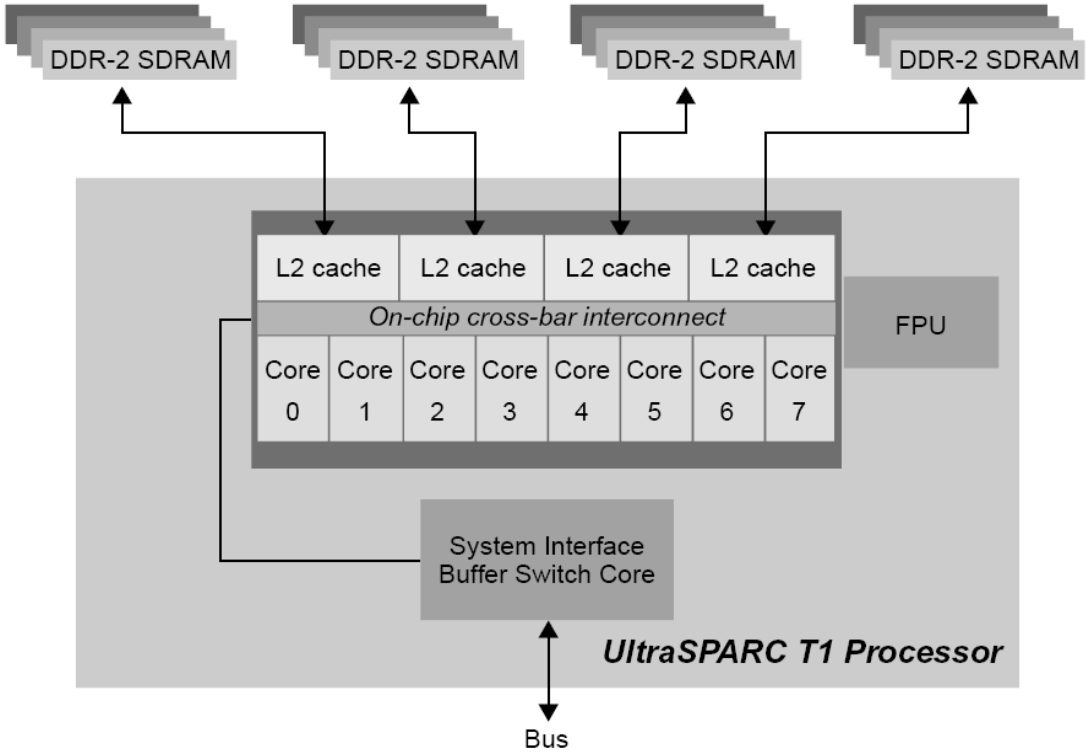
Multicore

- Multiple processor cores on a chip
  - Chip multiprocessor (CMP)
  - Sun's Chip Multithreading (CMT)
    - UltraSPARC T1 (Niagara)
  - Intel's Pentium D
  - AMD dual-core Opteron
- Also a way to utilize TLP, but
  - 2 cores ➔ 2X costs
  - No good for single thread performacne
- Can be used together with SMT

Chip Multithreading (CMT)



Sun UltraSPARC T1 Processor



http://www.sun.com/servers/wp.jsp?tab=3&group=CoolThreads%20servers

8 Cores vs 2 Cores

- Is 8-cores too aggressive?
  - Good for server applications, given
    - Lots of threads
    - Scalable operating environment
    - Large memory space (64bit)
  - Good for power efficiency
    - Simple pipeline design for each core
  - Good for availability
  - Not intended for PCs, gaming, etc

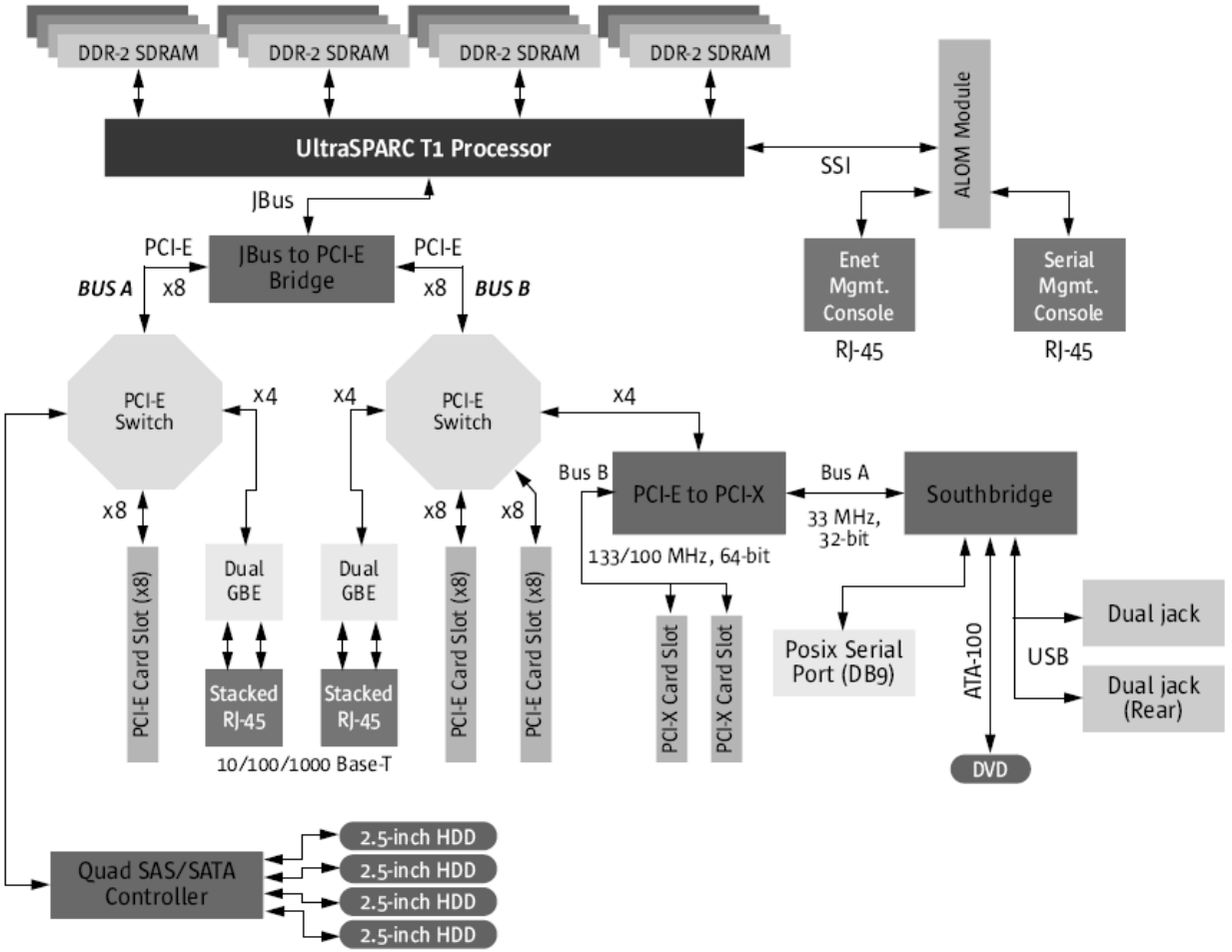
SPECWeb 2005

Feature	Sun Fire T2000 Server	IBM eServer x346	Sun Fire T2000 Server Advantage
Space (RU)	2	2	None
Watts	330	438	25%
Performance (composite)	14,001	4,348	3.22 times
Performance/watt	42.427	9.927	4.27 times
SWaP	21.2	5.0	4.24 times

IBM X346: 3Ghz Xeon

T2000: 8 core 1.0GHz T1 Processor

Sun Fire T2000 Server



Server Pricing

- UltraSPARC
  - Sun Fire T1000 Server
    - 6 core
      1.0GHz T1 Processor
    - 2GB memory,
      1x 80GB disk
    - List price: $5,745
  - Sun Fire T2000 Server
    - 8 core 1.0GHz
      T1 Processor
    - 8GB DDR2 memory,
      2 X 73GB disk
    - List price: $13,395
- X86
  - Sun Fire X2100 Server
    - Dual core
      AMD Opteron 175
    - 2GB memory,
      1x80GB disk
    - List price: $2,295
  - Sun Fire X4200 Server
    - 2x Dual core
      AMD Opteron 275
    - 4GB memory,
      2x 73GB disk
    - List price: $7,595

Multiprocessor Conclusion

- Some optimism about future
  - Parallel processing beginning to be understood in some domains
  - More performance than that achieved with a single-chip microprocessor
  - MPs are highly effective for multiprogrammed workloads
  - MPs proved effective for intensive commercial workloads, such as OLTP (assuming enough I/O to be CPU-limited), DSS applications (where query optimization is critical), and large-scale, web searching applications
  - On-chip MPs appears to be growing
    1) embedded market where natural parallelism often exists an obvious alternative to faster less silicon efficient, CPU.
    2) diminishing returns in high-end microprocessor encourage designers to pursue on-chip multiprocessing