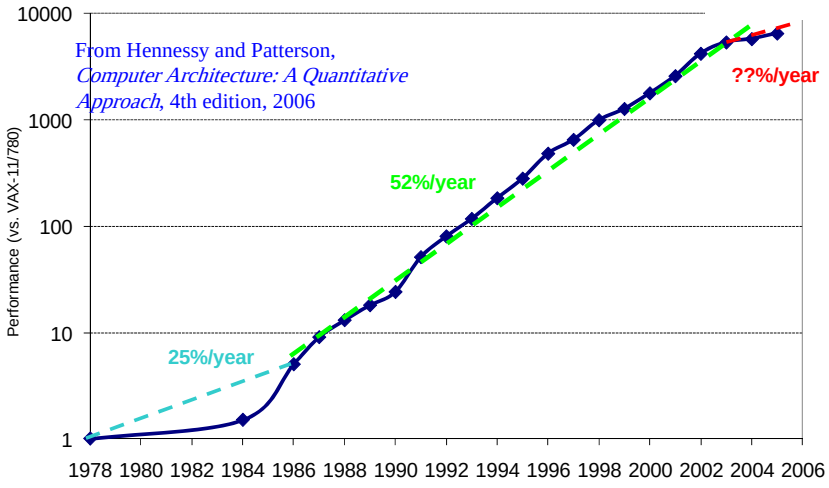
Multiprocessors and Thread-Level Parallelism

CS 282 – KAUST – Spring 2010

Muhamed Mudawar

Original slides created by: David Patterson

Uniprocessor Performance (SPECint)



- **VAX : 25%/year 1978 to 1986**
- **RISC + x86: 52%/year 1986 to 2002**
- **RISC + x86: <20%/year 2002 to present**

2

Déjà vu all over again?

"… today's processors … are nearing an impasse as technologies approach the speed of light.."

David Mitchell, *The Transputer: The Time Is Now* (1989)

- Transputer had bad timing (Uniprocessor performance↑)
  ⇒ Procrastination rewarded: 2X sequential performance / 1.5 years
- "We are dedicating all of our future product development to multicore designs. … This is a sea change in computing"

  Paul Otellini, President, Intel (2005)
- All microprocessor companies switch to MP (2X CPUs / 2 yrs)
  ⇒ Procrastination penalized: 2X sequential performance / 5 yrs

Manufacturer/Year	AMD/'05	Intel/'06	IBM/'04	Sun/'05
Processors/chip	2	2	2	8
Threads/Processor	1	2	2	4
Threads/chip	2	4	4	32

Other Factors ⇒ Multiprocessors

- Growth in data-intensive applications
  - Databases, file servers, …
- Growing interest in servers, server performance
- Increasing desktop performance is less important
  - Outside of graphics
- Improved understanding in how to use multiprocessors effectively
  - Especially server where significant natural TLP
- Advantage of leveraging design investment by replication
  - Rather than unique design

4

Flynn's Taxonomy

- Flynn classified by data and instruction streams in 1966

Single Instruction Stream Single Data Stream (SISD) (Uniprocessor)	**Single Instruction Stream Multiple Data Stream SIMD (single PC: Vector, CM-2)**
Multiple Instruction Stream Single Data Stream (MISD) (No commercial computer)	**Multiple Instruction Stream Multiple Data Stream MIMD (Clusters, SMP servers)**

- SIMD ⇒ Data Level Parallelism
- MIMD ⇒ Thread Level Parallelism
- MIMD popular because
  - Flexible: N programs or 1 multithreaded program
  - Cost-effective: same microprocessor used in desktop & Multiprocessor

5

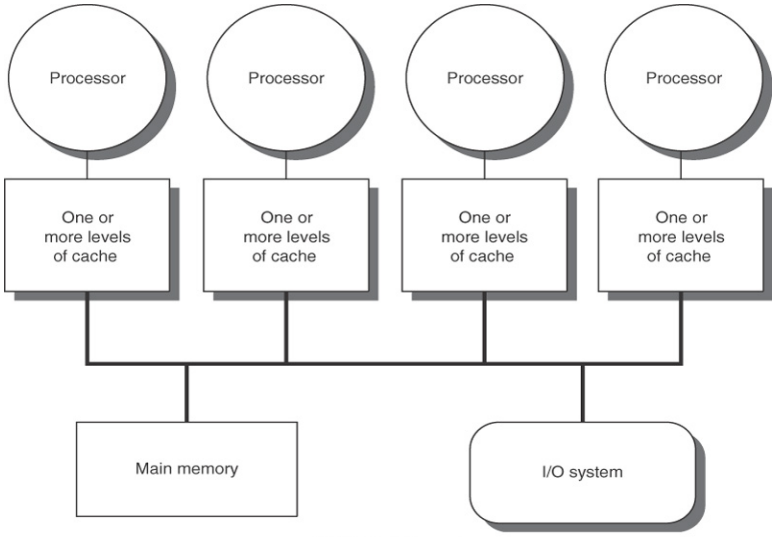
Back to Basics

- "A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast."
- Parallel Architecture = Computer Architecture + Communication Architecture
- 2 classes of multiprocessors WRT memory:

1. Centralized Memory Multiprocessor
   - < few dozen processor chips (and < 100 cores) in 2006
   - Small enough to share single, centralized memory
2. Physically Distributed-Memory multiprocessor
   - Larger number chips and cores.
   - Increasing BW demands ⇒ Memory distributed among processors

6

Basic Structure of a Centralized Shared-Memory Multiprocessor

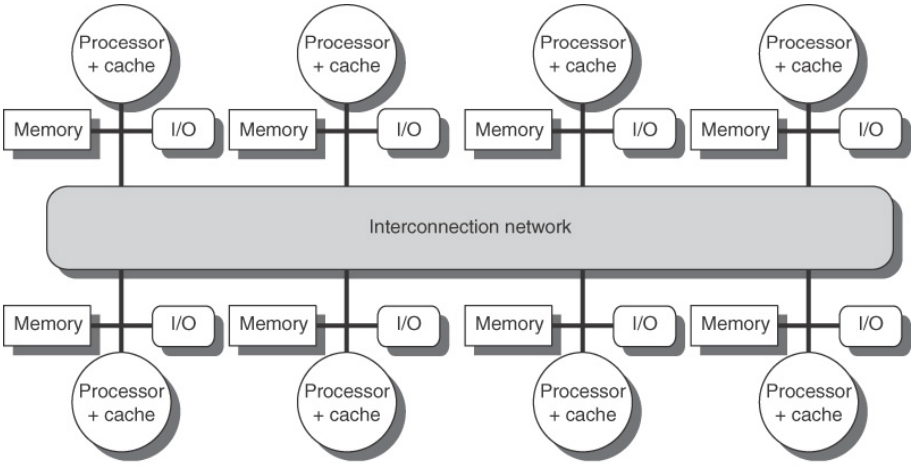


Centralized Memory Multiprocessor

- Also called symmetric multiprocessors (SMPs) because single main memory has a symmetric relationship to all processors
- Multilevel caches ⇒ single memory can satisfy memory demands of small number of processors
- Can scale to a few dozen processors by using a switch and by using many memory banks
- Although scaling beyond that is technically conceivable, it becomes less attractive as the number of processors sharing centralized memory increases

8

Basic Architecture of a Distributed Memory Multiprocessor



Distributed Memory Multiprocessor

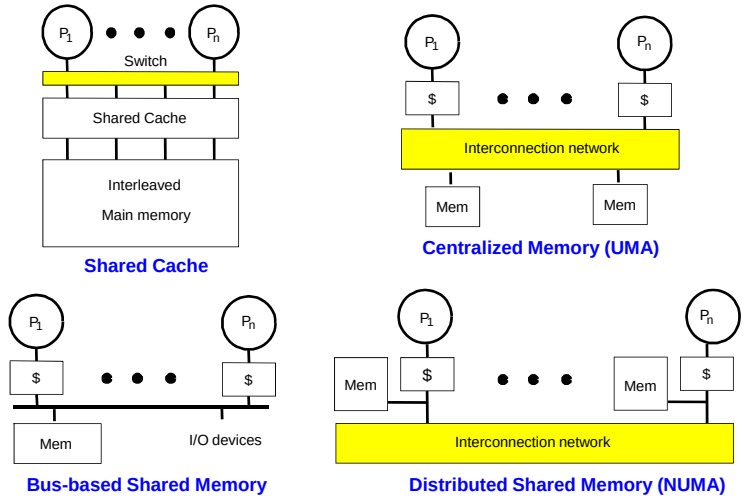
- Pro: Cost-effective way to scale memory bandwidth
  - If most accesses are to local memory
- Pro: Reduces latency of local memory accesses
- Con: Communicating data between processors more complex
- Con: Must change software to take advantage of local memory references

2 Models for Communication and Memory Architecture

1. Communication occurs by explicitly passing messages among the processors: message-passing multiprocessors
2. Communication occurs through a shared address space (via loads and stores): shared memory multiprocessors either
   - UMA (Uniform Memory Access time) for shared address, centralized memory MP
   - NUMA (Non Uniform Memory Access time multiprocessor) for shared address, distributed memory MP
- In past, confusion whether "sharing" means sharing physical memory (Symmetric MP) or sharing the address space (which can be distributed)

11

Shared Memory Organizations



Challenges of Parallel Processing

- First challenge is % of program inherently sequential
- Suppose we would like to achieve 80X speedup from 100 processors. What fraction of original program can be sequential?

  a. 20%

  b. 10%

  c. 5%

  d. 1%

  e. <1%

13

Amdahl's Law Answers

$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}}}$$

$$80 = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{100}}$$

$$80 \times \left(\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{100}\right) = 1$$

$$79 = 80 \times \text{Fraction}_{\text{parallel}} - 0.8 \times \text{Fraction}_{\text{parallel}}$$

$$\text{Fraction}_{\text{parallel}} = 79 / 79.2 = 99.75\%$$

14

Challenges of Parallel Processing

- Second challenge is long latency to remote memory
- Suppose 32 CPU MP, 2GHz, 200 ns remote memory access time, all local accesses hit in the cache hierarchy and all remote accesses stall the CPU. Base CPI is 0.5 (Remote access = 200 ns/0.5 ns per cycle = 400 clocks.)
- What is the performance impact on the CPI if 0.2% of instructions involve remote memory access?
  a. Increases the CPI by 1.5X
  b. Increases the CPI by 2.0X
  c. Increases the CPI by 2.5X

15

CPI Equation

- CPI = Base CPI + stall cycles per instr for remote access
  CPI = Base CPI + Remote access per instruction x
  Remote access penalty
- CPI = 0.5 + 0.2% x 400 = 0.5 + 0.8 = 1.3
- Impact on the CPI = 1.3/0.5 = 2.6X
- Processor is 2.6 times slower because 0.2% of instructions involve remote access that stalls the CPU

16

Challenges of Parallel Processing

1. Application parallelism ⇒ primarily via new algorithms that have better parallel performance
2. Long remote latency impact ⇒ both by architect and by the programmer

- For example, reduce frequency of remote accesses either by
  - Caching shared data (HW)
  - Restructuring the data layout to make more accesses local (SW)

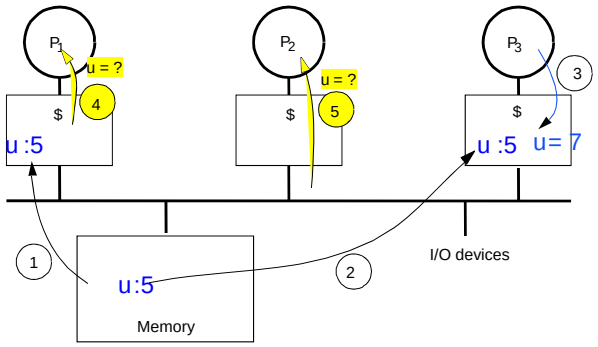
17

Symmetric Shared-Memory Architectures

- From multiple boards on a shared bus to multiple processors inside a single chip
- Caches both
  - Private data are used by a single processor
  - Shared data are used by multiple processors
- Caching shared data
  ⇒ reduces latency to shared data, memory bandwidth for shared data, and interconnect bandwidth
  ⇒ cache coherence problem

18

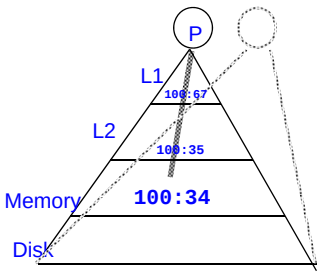
Example Cache Coherence Problem



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

19

Intuitive Memory Model



- **Reading an address should return the last value written to that address**
  - **Easy in uniprocessors, except for I/O**
  - **More difficult in multiprocessors**

- Too vague and simplistic; there are 2 issues
1. Coherence defines values returned by a read
2. Consistency determines when a written value will be returned by a read
- Coherence defines behavior of reads and writes to same location, while Consistency defines behavior of reads and writes with respect to accesses to other locations

0

Defining Coherent Memory System

1. Preserve Program Order: A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P
2. Coherent view of memory: Read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses
3. Write serialization: 2 writes to same location by any 2 processors are seen in the same order by all processors
   - For example, if the values 1 and then 2 are written to a location X by P1 and P2, processors can never read the value of the location X as 2 and then later read it as 1

21

Basic Schemes for Enforcing Coherence

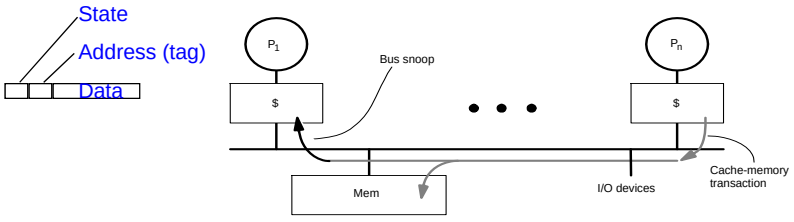
- Program on multiple processors will normally have copies of the same data in several caches
- Rather than trying to avoid sharing in SW, SMPs use a HW protocol to maintain coherent caches
  - Migration and Replication key to performance of shared data
- Migration - data can be moved to a local cache and used there in a transparent fashion
  - Reduces both latency to access shared data that is allocated remotely and bandwidth demand on the shared memory
- Replication – for shared data being simultaneously read, since caches make a copy of data in local cache
  - Reduces both latency of access and contention for reading shared data

22

2 Classes of Cache Coherence Protocols

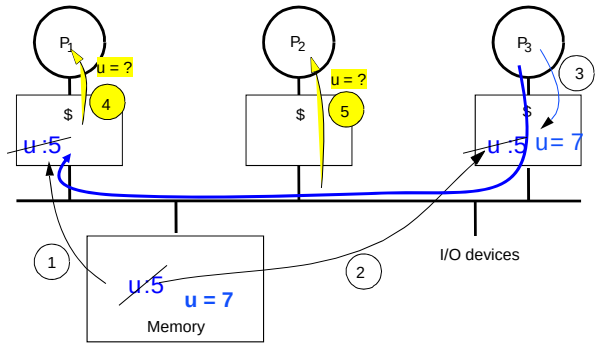
1. Snooping — Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept
   - All caches are accessible via some broadcast medium (a bus or switch)
   - All cache controllers monitor or snoop on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access
2. Directory based — Sharing status of a block of physical memory is kept in just one location, the directory

Snoopy Cache-Coherence Protocols



- Cache Controller "snoops" all transactions on the shared medium (bus or switch)
  - 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
- Either get exclusive access before write via write invalidate or update all copies on write

Example: Write-thru Invalidate



- Must invalidate before step 3
- Write update uses more broadcast medium BW
  ⇒ all recent MPUs use write invalidate

25

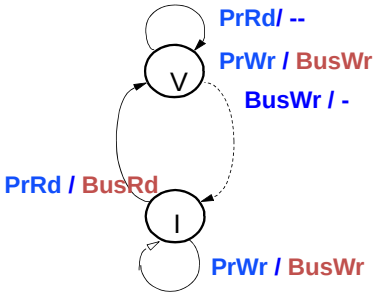
Example Protocol

- Snooping coherence protocol is usually implemented by incorporating a finite-state controller in each node
- Logically, think of a separate controller associated with each cache block
  - That is, snooping operations or cache requests for different blocks can proceed independently
- In implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion
  - that is, one operation may be initiated before another is completed, even through only one cache access or one bus access is allowed at time

26

Write-through Invalidate Protocol

- 2 states per block in each cache
  - as in uniprocessor
  - state of a block is a 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 cache copies
  - can have multiple simultaneous readers of block,but write invalidates them

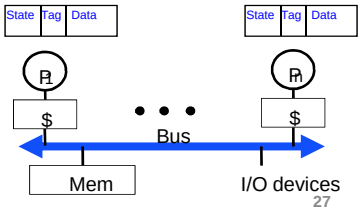


PrRd: Processor Read

PrWr: Processor Write

BusRd: Bus Read

BusWr: Bus Write



27

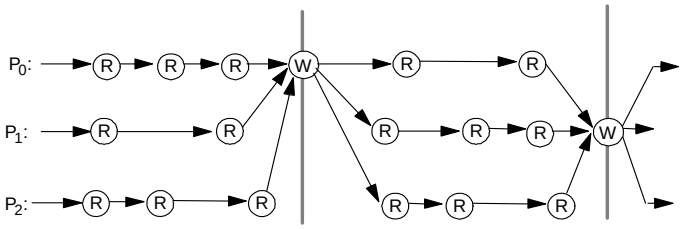
Is 2-state Protocol Coherent?

- Processor only observes state of memory system by issuing memory operations
- Assume bus transactions and memory operations are atomic and a one-level cache
  - all phases of one bus transaction complete before next one starts
  - processor waits for memory operation to complete before issuing next
  - with one-level cache, invalidations are applied during bus transaction
- All writes go to bus + atomicity
  - Writes serialized by order in which they appear on bus (bus order)

  => invalidations applied to caches in bus order
- How to insert reads in this order?
  - Important since processors see writes through reads, so determines whether write serialization is satisfied
  - But read hits may happen independently and do not appear on bus or enter directly in bus order
- Let's understand other ordering issues

28

Ordering



- Writes establish a partial order
- Doesn't constrain ordering of reads, though shared-medium (bus) will order read misses too
  - any order among reads between writes is fine, as long as in program order

29

Locate up-to-date copy of data

- Write-through: get up-to-date copy from memory
  - Write through simpler if enough memory BW
- Write-back: most recent copy can be in a cache
  - harder to implement
- Can use same snooping mechanism
  1. Snoop every address placed on the bus
  2. If a processor has dirty copy of requested cache block, it provides it in response to a read request and aborts the memory access
  - Complexity of retrieving cache block from a processor cache
- Write-back needs lower memory bandwidth
  ⇒ Support larger numbers of faster processors
  ⇒ Most multiprocessors use write-back

Architectural Building Blocks

- Cache block state transition diagram
  - FSM specifying how state of block changes
    - invalid, valid, modified
- Broadcast Medium Transactions (e.g., bus)
  - Fundamental system design abstraction
  - Logically single set of wires connect several devices
  - Protocol: arbitration, command/addr, data
  - ⇒ Every device observes every transaction
- Broadcast medium enforces serialization of read or write accesses ⇒ Write serialization
  - 1st processor to get medium invalidates others copies
  - Implies cannot complete write until it obtains bus
  - All coherence schemes require serializing accesses to same cache block
- Also need to find up-to-date copy of cache block

31

Cache Resources for WB Snooping

- Normal cache tags can be used for snooping
- Valid bit per block makes invalidation easy
- Read misses easy since rely on snooping
- Writes ⇒ Need to know whether any other copies of the block are cached
  - No other copies ⇒ No need to place write on bus for WB
  - Other copies ⇒ Need to place invalidate on bus

32

Cache Resources for WB Snooping

- To track whether a cache block is shared, add extra state bit associated with each cache block, like valid bit and dirty bit
  - Write to Shared block ⇒ Need to place invalidate on bus and mark cache block as private (if an option)
  - No further invalidations will be sent for that block
  - This processor called owner of cache block
  - Owner then changes state from shared to unshared (or exclusive)

33

Cache behavior in response to bus

- Every bus transaction must check the cache-address tags
  - could potentially interfere with processor cache accesses
- A way to reduce interference is to duplicate tags
  - One set for caches access, second set for bus accesses
- Another way to reduce interference is to use L2 tags
  - Since L2 less heavily used than L1

  ⇒ Every entry in L1 cache must be present in the L2 cache, called the inclusion property
  - If Snoop gets a hit in L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor

34

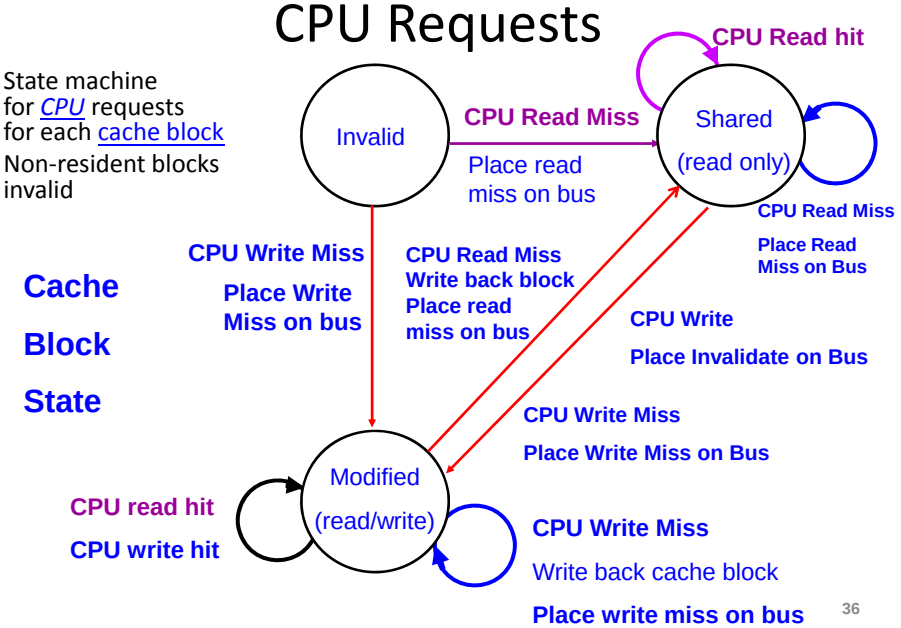
Example Write Back Snoopy Protocol

- Invalidation protocol, write-back cache
  - Snoops every address on bus
  - If it has a modified copy of requested block, provides that block in response to the read request and aborts the memory access
- Each memory block is in one state:
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Modified in exactly one cache (Modified)
  - OR Not in the cache (Invalid)
- Each cache block is in one state (track these):
  - Shared : block can be read and can be replicated in multiple caches
  - OR Modified : only this cache has a copy, it is writeable, and modified
  - OR Invalid : block contains invalid data (cannot be read)
- Read misses: cause all caches to snoop bus
- Writes to shared blocks are treated as misses

35

Write-Back State Machine CPU Requests

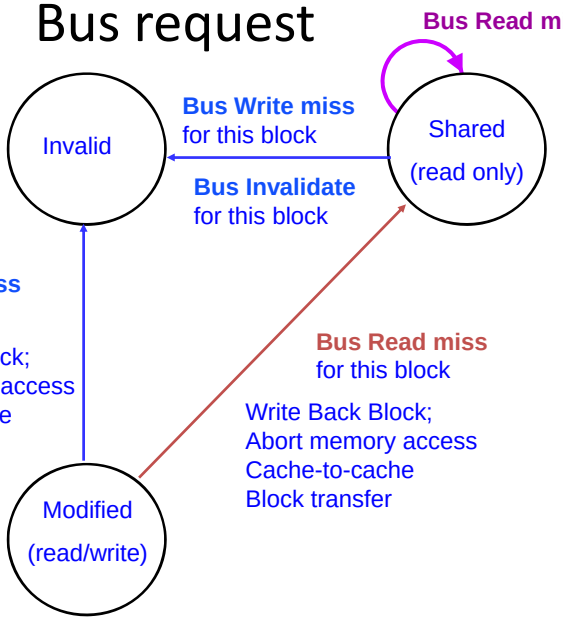
- State machine for *CPU* requests for each cache block
- Non-resident blocks invalid



36

Write-Back State Machine- Bus request

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



37

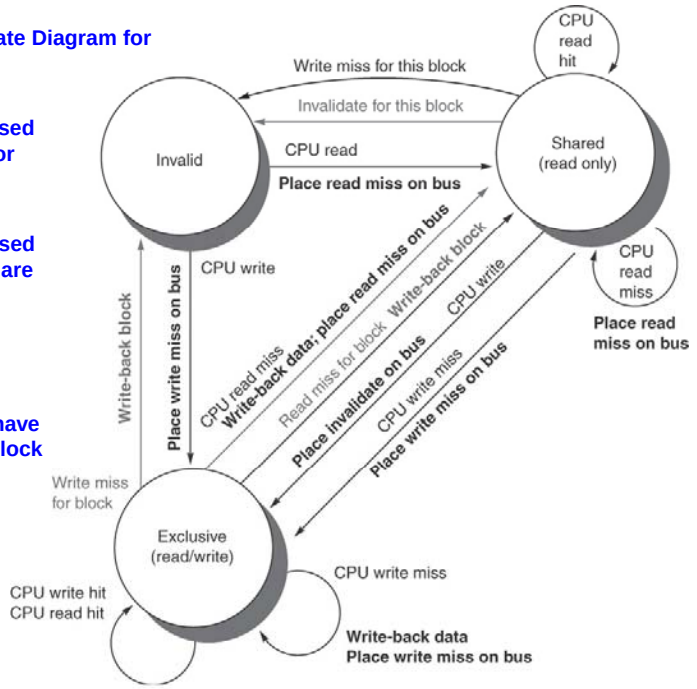
Cache Coherence State Diagram for a Write-Back Cache

State transitions caused by the local processor are shown in black

State transitions caused by the Bus activities are shown in grey

Exclusive state here means Modified

No other cache can have a copy of the same block



Example

Step	P1			P2			Bus			Memory	
	State	Addr	Value	State	Addr	Value	Proc	Addr	Action	Addr	Value
P1: Write 10 to A1										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

39

Example

Step	P1			P2			Bus			Memory	
	State	Addr	Value	State	Addr	Value	Proc	Addr	Action	Addr	Value
P1: Write 10 to A1							P1	A1	WrMiss	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

40

Example

Step	P1			P2			Bus			Memory	
	State	Addr	Value	State	Addr	Value	Proc	Addr	Action	Addr	Value
P1: Write 10 to A1							P1	A1	WrMiss	A1	?
	M	A1	10								
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

41

Example

Step	P1			P2			Bus			Memory	
	State	Addr	Value	State	Addr	Value	Proc	Addr	Action	Addr	Value
P1: Write 10 to A1							P1	A1	WrMiss	A1	?
	M	A1	10								
P1: Read A1 (Hit)	M	A1	10								
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

42

Example

Step	P1			P2			Bus			Memory	
	State	Addr	Value	State	Addr	Value	Proc	Addr	Action	Addr	Value
P1: Write 10 to A1							P1	A1	WrMiss	A1	?
	M	A1	10								
P1: Read A1 (Hit)	M	A1	10								
P2: Read A1							P2	A1	RdMiss		
P2: Write 20 to A1											
P2: Write 40 to A2											

Assumes A1 and A2 map to same cache block,

Initial cache state is invalid

43

Example

Step	P1			P2			Bus			Memory	
	State	Addr	Value	State	Addr	Value	Proc	Addr	Action	Addr	Value
P1: Write 10 to A1							P1	A1	WrMiss	A1	?
	M	A1	10								
P1: Read A1 (Hit)	M	A1	10								
P2: Read A1							P2	A1	RdMiss		
	S	A1	10				P1	A1	Wrback	A1	10
P2: Write 20 to A1											
P2: Write 40 to A2											

Assumes A1 and A2 map to same cache block,

Initial cache state is invalid

44

Example

Step	P1			P2			Bus			Memory	
	State	Addr	Value	State	Addr	Value	Proc	Addr	Action	Addr	Value
P1: Write 10 to A1							P1	A1	WrMiss	A1	?
	M	A1	10								
P1: Read A1 (Hit)	M	A1	10								
P2: Read A1							P2	A1	RdMiss		
	S	A1	10				P1	A1	Wrback	A1	10
				S	A1	10	P2	A1	Transfer		
P2: Write 20 to A1											
P2: Write 40 to A2											

Assumes A1 and A2 map to same cache block,

Initial cache state is invalid

45

Example

Step	P1			P2			Bus			Memory	
	State	Addr	Value	State	Addr	Value	Proc	Addr	Action	Addr	Value
P1: Write 10 to A1							P1	A1	WrMiss	A1	?
	M	A1	10								
P1: Read A1 (Hit)	M	A1	10								
P2: Read A1							P2	A1	RdMiss		
	S	A1	10				P1	A1	Wrback	A1	10
				S	A1	10	P2	A1	Transfer		
P2: Write 20 to A1							P2	A1	Inv		
P2: Write 40 to A2											

Assumes A1 and A2 map to same cache block,

Initial cache state is invalid

46

Example

Step	P1			P2			Bus			Memory	
	State	Addr	Value	State	Addr	Value	Proc	Addr	Action	Addr	Value
P1: Write 10 to A1							P1	A1	WrMiss	A1	?
	M	A1	10								
P1: Read A1 (Hit)	M	A1	10								
P2: Read A1							P2	A1	RdMiss		
	S	A1	10				P1	A1	Wrback	A1	10
				S	A1	10	P2	A1	Transfer		
P2: Write 20 to A1							P2	A1	Inv		
	I	A1	10	M	A1	20				A1	10
P2: Write 40 to A2											

Assumes A1 and A2 map to same cache block,

Initial cache state is invalid

47

Example

Step	P1			P2			Bus			Memory	
	State	Addr	Value	State	Addr	Value	Proc	Addr	Action	Addr	Value
P1: Write 10 to A1							P1	A1	WrMiss	A1	?
	M	A1	10								
P1: Read A1 (Hit)	M	A1	10								
P2: Read A1							P2	A1	RdMiss		
	S	A1	10				P1	A1	Wrback	A1	10
				S	A1	10	P2	A1	Transfer		
P2: Write 20 to A1							P2	A1	Inv		
	I	A1	10	M	A1	20				A1	10
P2: Write 40 to A2				M	A2	40				A2	?

Assumes A1 and A2 map to same cache block,

Initial cache state is invalid

48

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 Modified state
    - Must track and prevent multiple misses for one block
- Must support interventions and invalidations

49

Limitations of Symmetric Shared-Memory Multiprocessors and Snooping Protocols

- Single memory accommodate all CPUs
  $\Rightarrow$ Multiple memory banks
- Bus-based multiprocessor, bus must support both coherence traffic & normal memory traffic

$\Rightarrow$ Multiple buses or interconnection networks (cross bar or small point-to-point)

- Opteron
  - Memory connected directly to each dual-core chip
  - Point-to-point connections for up to 4 chips
  - Remote memory and local memory latency are similar, allowing OS Opteron as UMA computer

50

Performance of Symmetric Shared-Memory Multiprocessors

- Cache performance is combination of
1. Uniprocessor cache miss traffic
2. Traffic caused by communication and cache coherence
   - Results in invalidations and subsequent cache misses
- 4th C: *Coherence miss*
   - Joins Compulsory, Capacity, Conflict

51

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
     ⇒ miss would not occur if block size were 1 word

52

Example: True vs. False Sharing?

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

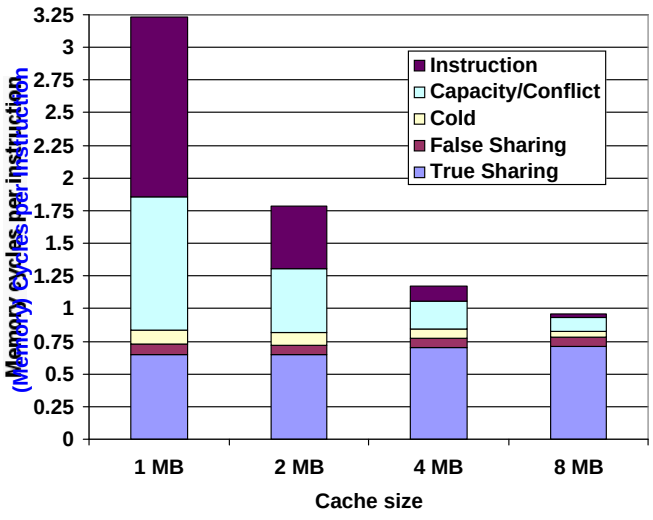
Time	P1	P2	**True, False Sharing? Why?**
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		True miss; invalidate x2 in P1

53

Performance of 4 Processor System
Commercial Workload: OLTP (Oracle Database)

- **True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)**

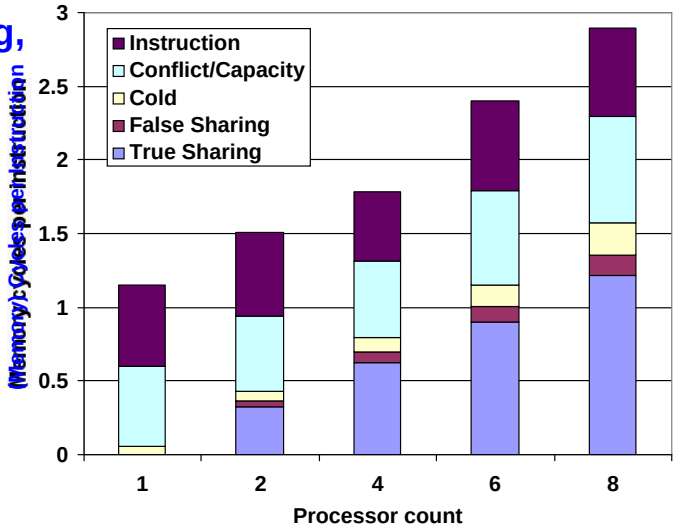
- **Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)**



54

MP Performance 2MB Cache
Increasing Processor Count (Commercial Workload)

• True sharing, false sharing increase going from 1 to 8 CPUs



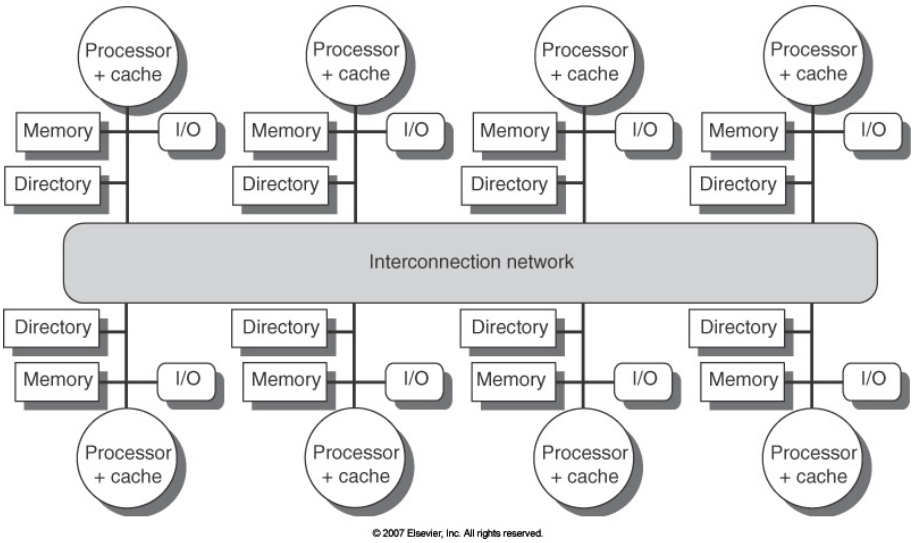
55

Scalable Approach: Directories

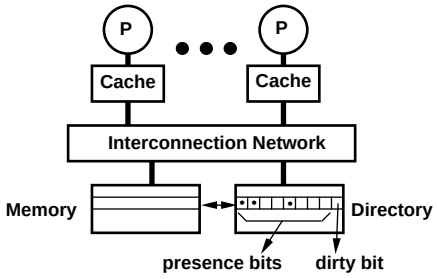
- Every memory block has associated directory information
  - keeps track of copies of cached blocks and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is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

56

Directory added to each Node



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 hasit; 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:
  - Writes to non-exclusive data => write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent

59

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

60

Directory Protocol Messages

Message type	*Source*	*Destination*	*Msg Content*
Read miss	Local cache	Home directory	P, A
– *Processor P has a read miss at address A; Request data and make P a read sharer*			
Write miss	Local cache	Home directory	P, A
– *Processor P has a write miss at address A; Request data &make P the exclusive owner*			
Invalidate	Local cache	Home directory	A
– *Request to invalidate all remote caches that are caching block at address A*			
Invalidate	Home directory	Remote cache	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)*			

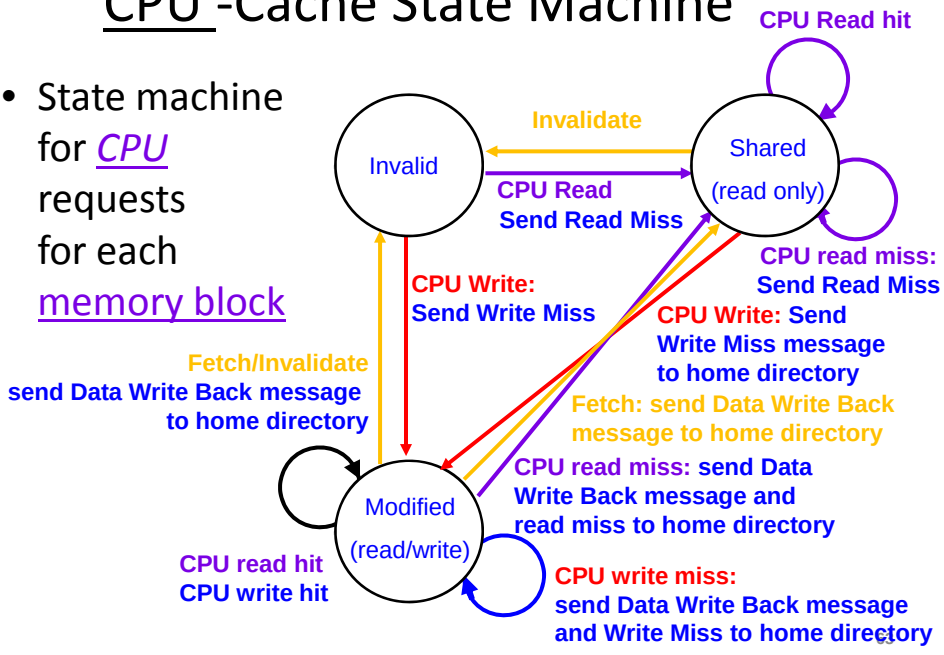
State Transition Diagram for One Cache Block in Directory Based System

- 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

62

CPU -Cache State Machine

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



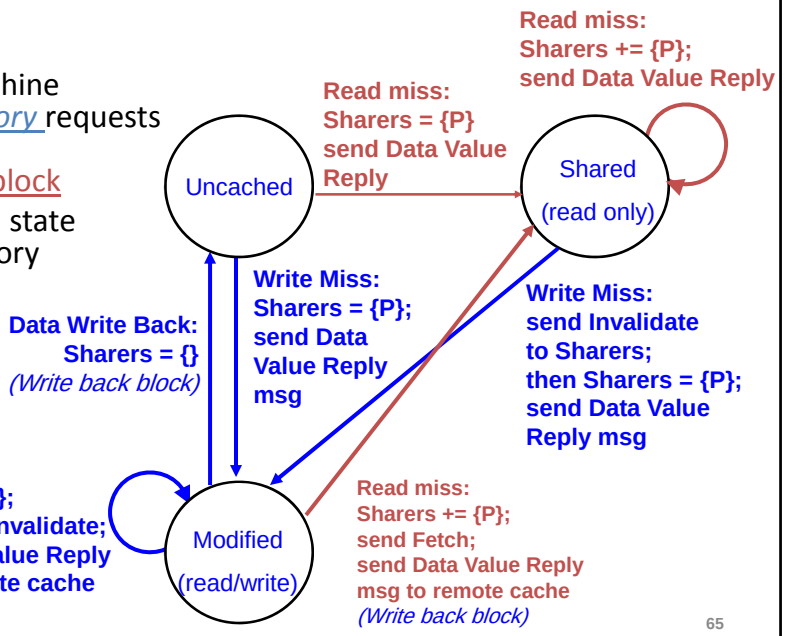
State Transition Diagram for Directory

- 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

64

Directory State Machine

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



65

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 Modifying node. The block is made Modified 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 Modified.

66

Example Directory Protocol

- Block is Modified: 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: directory 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 kept Modified.

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

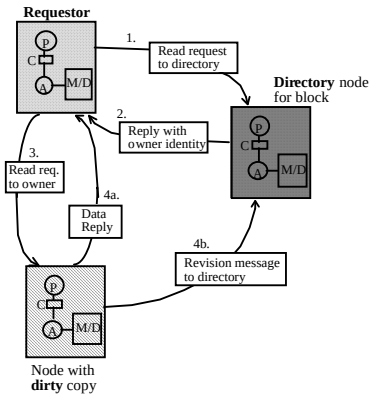
	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10							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

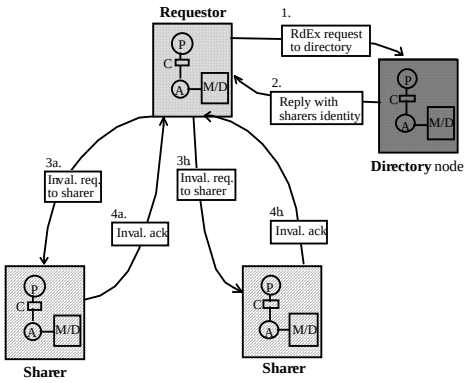
Implementing a Directory

- 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

Basic Directory Transactions

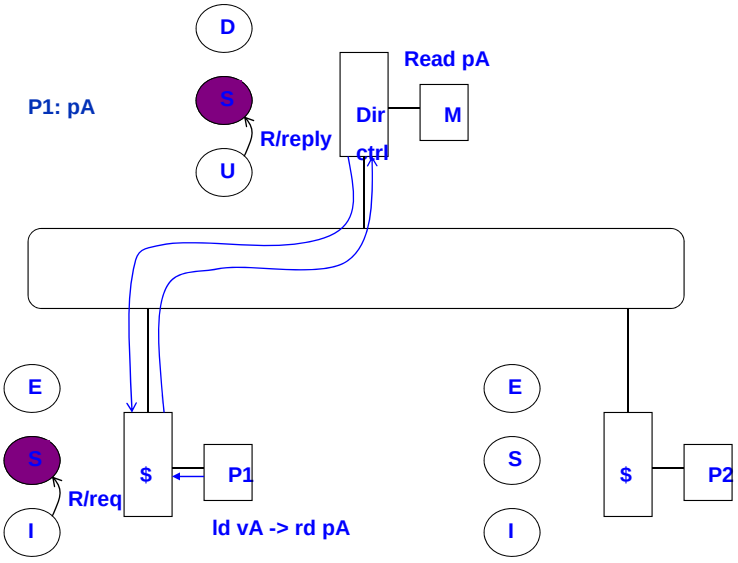


(a) Read miss to a block in dirty state

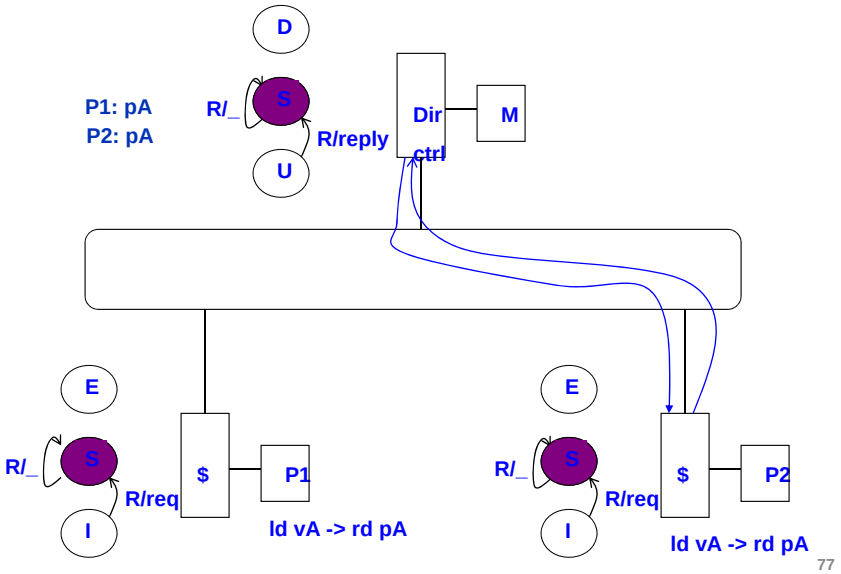


(b) Write miss to a block with two sharers

Example Directory Protocol (1st Read)

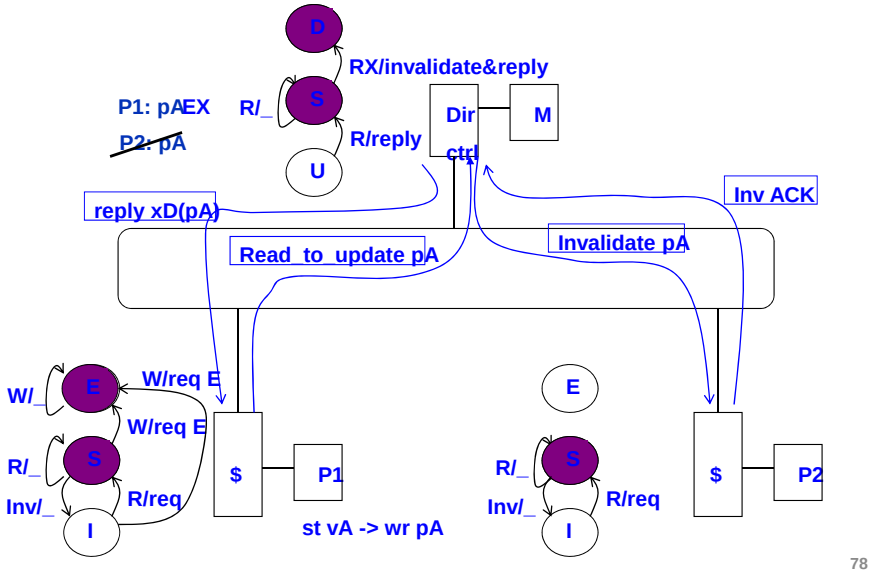


Example Directory Protocol (Read Share)



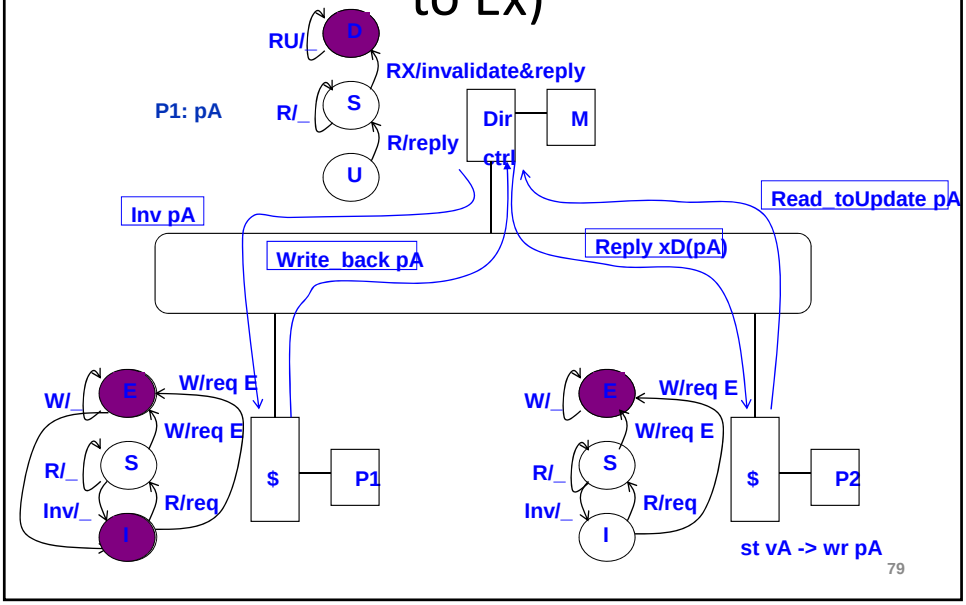
77

Example Directory Protocol (Wr to shared)



78

Example Directory Protocol (Wr to Ex)



79

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?

80

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; techniques to reduce contention and latency of synchronization

81

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

82

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)
```

83

User Level Synchronization—Operation Using this Primitive

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

```
lockit:   li    R2,#1
          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?
```

84

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

85

Example on Memory Consistency

P_1	P_2
/*Assume initial value of A and flag is 0*/	
`A = g();`	`while (flag == 0);` /* spin */
`flag = 1;`	`print A;` /* 0 or value of g()? */

- Intuition not guaranteed by coherence
- Coherence is not enough!
  - Pertains only to single location
- Expect memory to respect order of accesses to *different* locations issued by a given process
  - To preserve order among accesses to same location by different processes

86

Write Consistency

- For now assume

1. A write does not complete (and allow the next write to occur) until all processors have seen the effect of that write
2. The processor does not change the order of any write with respect to any other memory access

$\Rightarrow$ if a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

- These restrictions allow the processor to reorder reads, but forces the processor to finish writes in program order

87

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

88

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

89

Mark Hill observation

- Instead, use speculation to hide latency from strict consistency model
  - If processor receives invalidation for memory reference before it is committed, processor uses speculation recovery to back out computation and restart with invalidated memory reference

1. Aggressive implementation of sequential consistency or processor consistency gains most of advantage of more relaxed models
2. Implementation adds little to implementation cost of speculative processor
3. Allows the programmer to reason using the simpler programming models

90

Cross Cutting Issues: Performance Measurement of Parallel Processors

- Performance: how well scale as increase Proc
- Speedup fixed as well as scaleup of problem
  - Assume benchmark of size n on p processors makes sense: how scale benchmark to run on m * p processors?
  - Memory-constrained scaling: keeping the amount of memory used per processor constant
  - Time-constrained scaling: keeping total execution time, assuming perfect speedup, constant
- Example: 1 hour on 10 P, time ~ $O(n^3)$, 100 P?
  - Time-constrained scaling: 1 hour $\Rightarrow 10^{1/3}n \Rightarrow$ 2.15n scale up
  - Memory-constrained scaling: 10n size $\Rightarrow 10^3/10 \Rightarrow$ 100X or 100 hours! 10X processors for 100X longer???
  - Need to know application well to scale: # iterations, error tolerance

91

Fallacy: Amdahl's Law doesn't apply to parallel computers

- Since some part linear, can't go 100X?
- 1987 claim to break it, since 1000X speedup
  - researchers scaled the benchmark to have a data set size that is 1000 times larger and compared the uniprocessor and parallel execution times of the scaled benchmark. For this particular algorithm the sequential portion of the program was constant independent of the size of the input, and the rest was fully parallel—hence, linear speedup with 1000 processors
- Usually sequential scale with data too

92

Fallacy: Linear speedups are needed to make multiprocessors cost-effective

- Mark Hill & David Wood 1995 study
- Compare costs SGI uniprocessor and MP
- Uniprocessor = $38,400 + $100 * MB
- MP = $81,600 + $20,000 * P + $100 * MB
- 1 GB, uni = $138k v. mp = $181k + $20k * P
- What speedup for better MP cost performance?
- 8 proc = $341k; $341k/138k $\Rightarrow$ 2.5X
- 16 proc $\Rightarrow$ need only 3.6X, or 25% linear speedup
- Even if need some more memory for MP, not linear

93

Fallacy: Scalability is almost free

- "build scalability into a multiprocessor and then simply offer the multiprocessor at any point on the scale from a small number of processors to a large number"
- Cray T3E scales to 2048 CPUs vs. 4 CPU Alpha
  - At 128 CPUs, it delivers a peak bisection BW of 38.4 GB/s, or 300 MB/s per CPU (uses Alpha microprocessor)
  - Compaq Alphaserver ES40 up to 4 CPUs and has 5.6 GB/s of interconnect BW, or 1400 MB/s per CPU
- Build apps that scale requires significantly more attention to load balance, locality, potential contention, and serial (or partly parallel) portions of program. 10X is very hard

94

Pitfall: Not developing SW to take advantage (or optimize for) multiprocessor architecture

- SGI OS protects the page table data structure with a single lock, assuming that page allocation is infrequent
- Suppose a program uses a large number of pages that are initialized at start-up
- Program parallelized so that multiple processes allocate the pages
- But page allocation requires lock of page table data structure, so even an OS kernel that allows multiple threads will be serialized at initialization (even if separate processes)

95

Answers to 1995 Questions about Parallelism

- In the 1995 edition of this text, we concluded the chapter with a discussion of two then current controversial issues.

1. What architecture would very large scale, microprocessor-based multiprocessors use?
2. What was the role for multiprocessing in the future of microprocessor architecture?

Answer 1. Large scale multiprocessors did not become a major and growing market $\Rightarrow$ clusters of single microprocessors or moderate SMPs

Answer 2. Astonishingly clear. For at least for the next 5 years, future MPU performance comes from the exploitation of TLP through multicore processors vs. exploiting more ILP

96

Cautionary Tale

- Key to success of birth and development of ILP in 1980s and 1990s was software in the form of optimizing compilers that could exploit ILP
- Similarly, successful exploitation of TLP will depend as much on the development of suitable software systems as it will on the contributions of computer architects
- Given the slow progress on parallel software in the past 30+ years, it is likely that exploiting TLP broadly will remain challenging for years to come

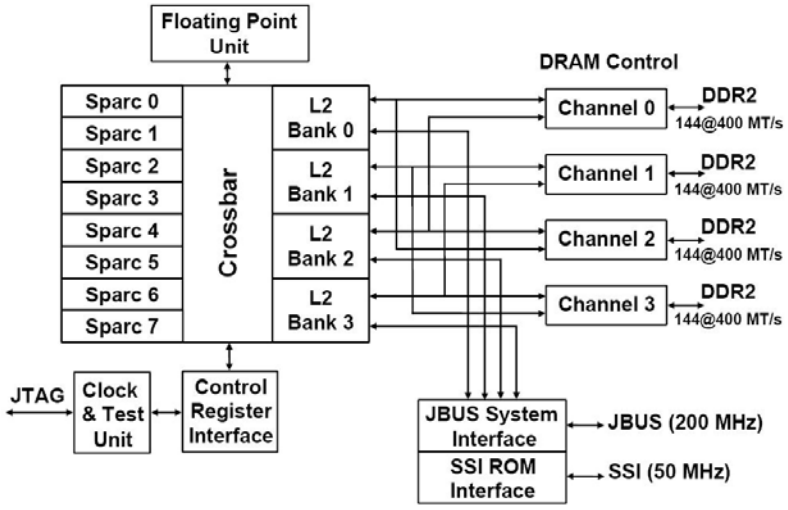
97

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

98

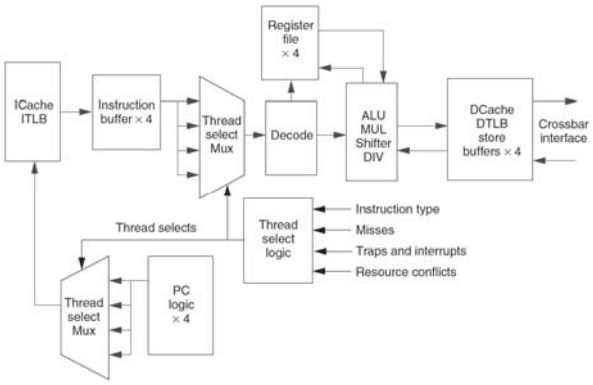
T1 Architecture



99

T1 pipeline

- Single issue, in-order, 6-deep pipeline: F, S, D, E, M, W
- 3 clock delays for loads & branches.
- Shared units:
  - L1 $, L2 $
  - TLB
  - X units
  - pipe register
- Hazards:
  - Data
  - Structural



100

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

101

Memory, Clock, Power

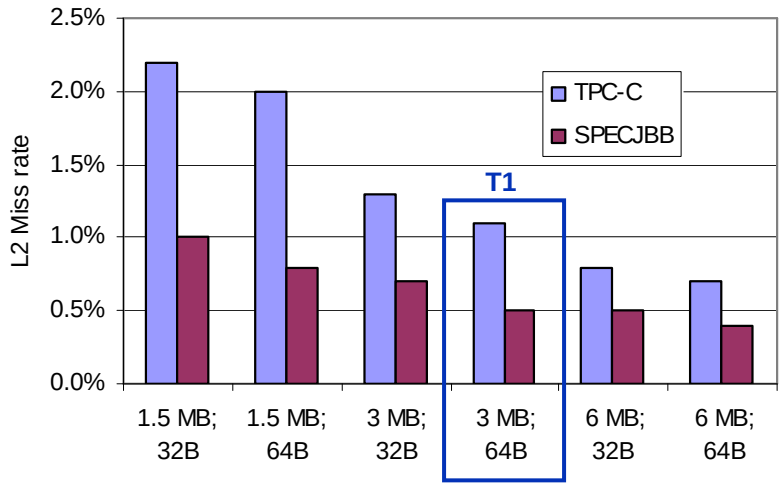
- 16 KB 4 way set assoc. I$/ core
- 8 KB 4 way set assoc. D$/ core
- 3MB 12 way set assoc. L2 $ shared
  - 4 x 750KB independent banks
  - crossbar switch to connect
  - 2 cycle throughput, 8 cycle latency
  - Direct link to DRAM & Jbus
  - Manages cache coherence for the 8 cores
  - CAM based directory

Write through
- allocate LD
- no-allocate ST

- 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

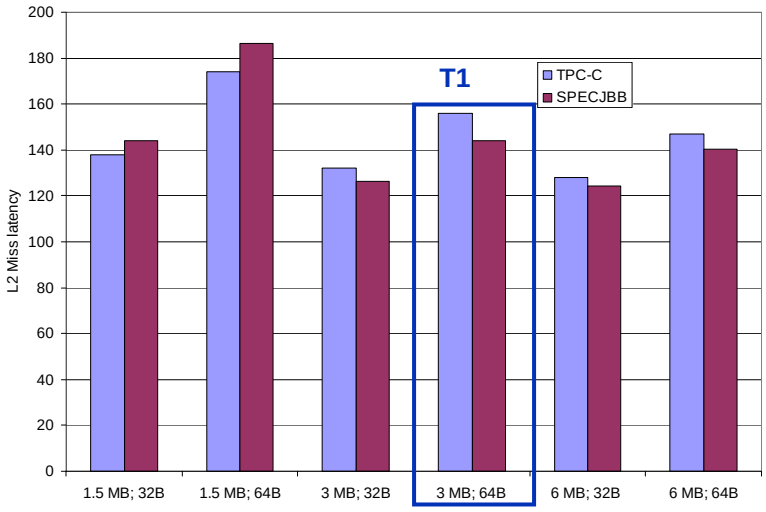
102

Miss Rates: L2 Cache Size, Block Size



103

Miss Latency: L2 Cache Size, Block Size



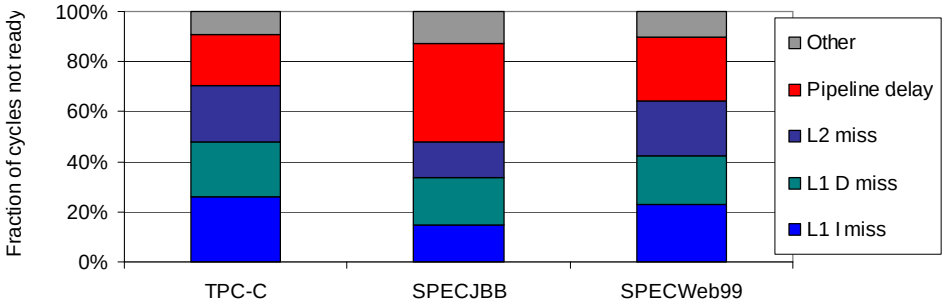
104

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

105

Not Ready Breakdown



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

106

Performance: Benchmarks + Sun Marketing

Benchmark\Architecture	Sun Fire T2000	IBM p5-550 with 2 dual-core Power5 chips	Dell PowerEdge
SPECjbb2005 (Java server software) business operations/ sec	63,378	61,789	24,208 (SC1425 with dual single-core Xeon)
SPECweb2005 (Web server performance)	14,001	7,881	4,850 (2850 with two dual-core Xeon processors)
NotesBench (Lotus Notes performance)	16,061	14,740	

SPECjappServer 2004 Dual Node	Sun Fire T2000	HP rx4640
Space (RU)	2	4
Watts	320	1,303
Performance (SPECjapp JOPs)	615	471
Performance / Watt	1.922	0.361
SWaP	0.96	0.09

Space, Watts, and Performance

107

HP marketing view of T1 Niagara

1. Sun's radical UltraSPARC T1 chip is made up of individual cores that have much slower single thread performance when compared to the higher performing cores of the Intel Xeon, Itanium, AMD Opteron or even classic UltraSPARC processors.
2. The Sun Fire T2000 has poor floating-point performance, by Sun's own admission.
3. The Sun Fire T2000 does not support commerical Linux or Windows® and requires a lock-in to Sun and Solaris.
4. The UltraSPARC T1, aka CoolThreads, is new and unproven, having just been introduced in December 2005.
5. In January 2006, a well-known financial analyst downgraded Sun on concerns over the UltraSPARC T1's limitation to only the Solaris operating system, unique requirements, and longer adoption cycle, among other things. [10]

- Where is the compelling value to warrant taking such a risk?
- http://h71028.www7.hp.com/ERC/cache/280124-0-0-0-121.html

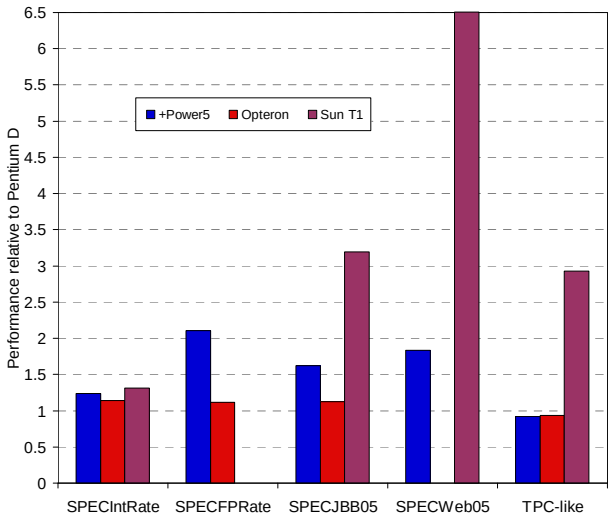
108

Microprocessor Comparison

Processor	SUN T1	Opteron	Pentium D	IBM Power 5
Cores	**8**	2	2	2
Instruction issues / clock / core	1	3	3	4
Peak instr. issues / chip	**8**	6	6	**8**
Multithreading	Fine-grained	No	SMT	SMT
L1 I/D in KB per core	16/8	**64/64**	12K uops/16	64/32
L2 per core/shared	**3 MB** shared	1MB / core	1MB/ core	1.9 MB shared
Clock rate (GHz)	1.2	2.4	**3.2**	1.9
Transistor count (M)	**300**	233	230	276
Die size (mm^2)	379	199	206	**389**
Power (W)	**79**	110	130	125

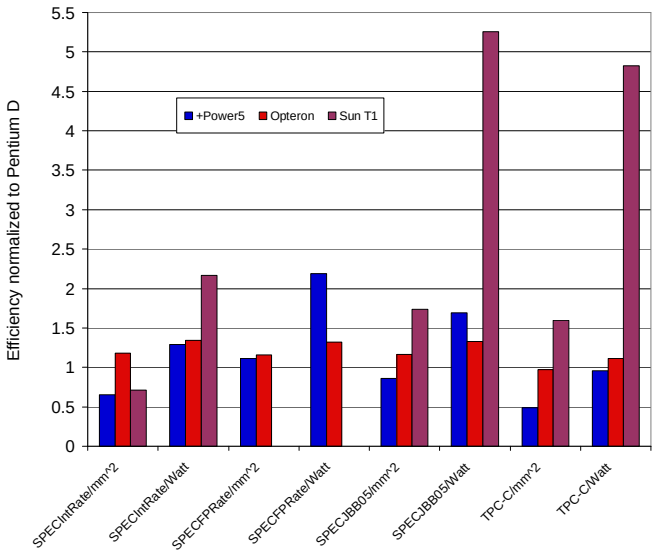
109

Performance Relative to Pentium D



110

Performance/mm², Performance/Watt



111

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.

Kevin Krewell

"Sun's Niagara Begins CMT Flood - The Sun UltraSPARC T1 Processor Released"

Microprocessor Report, January 3, 2006

112

Amdahl's Law Paper

- Gene Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings, (30), pp. 483-485, 1967.
- How long is paper?
- How much of it is Amdahl's Law?
- What other comments about parallelism besides Amdahl's Law?

Parallel Programmer Productivity

- Lorin Hochstein *et al* "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers." International Conference for High Performance Computing, Networking and Storage (SC'05). Nov. 2005
- What did they study?
- What is argument that novice parallel programmers are a good target for High Performance Computing?
- How can account for variability in talent between programmers?
- What programmers studied?
- What programming styles investigated?
- How big multiprocessor?
- How measure quality?
- How measure cost?

Parallel Programmer Productivity

- Lorin Hochstein *et al* "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers." International Conference for High Performance Computing, Networking and Storage (SC'05). Nov. 2005
- What hypotheses investigated?
- What were results?
- Assuming these results of programming productivity reflect the real world, what should architectures of the future do (or not do)?
- How would you redesign the experiment they did?
- What other metrics would be important to capture?
- Role of Human Subject Experiments in Future of Computer Systems Evaluation?

115