Lecture 19: Syncrhonization, Memory Consistency Models, and MP Example

Professor David A. Patterson

Computer Science 252

Fall 1996

Review

- Caches contain all information on state of cached memory blocks
- Snooping and Directory Protocols similar; bus makes snooping easier because of broadcast

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	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		10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1			10
P2: Write 40 to A2							WrMs	P2	A2			10
				Excl.	A2	40	WrBk	P2	A1	20	A1	20

Assumes A1 and A2 map to same cache block

Implementation Complications

- Write Races:
  - Cannot update cache until bus is obtained
    - Otherwise, another processor may get bus first, and 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**

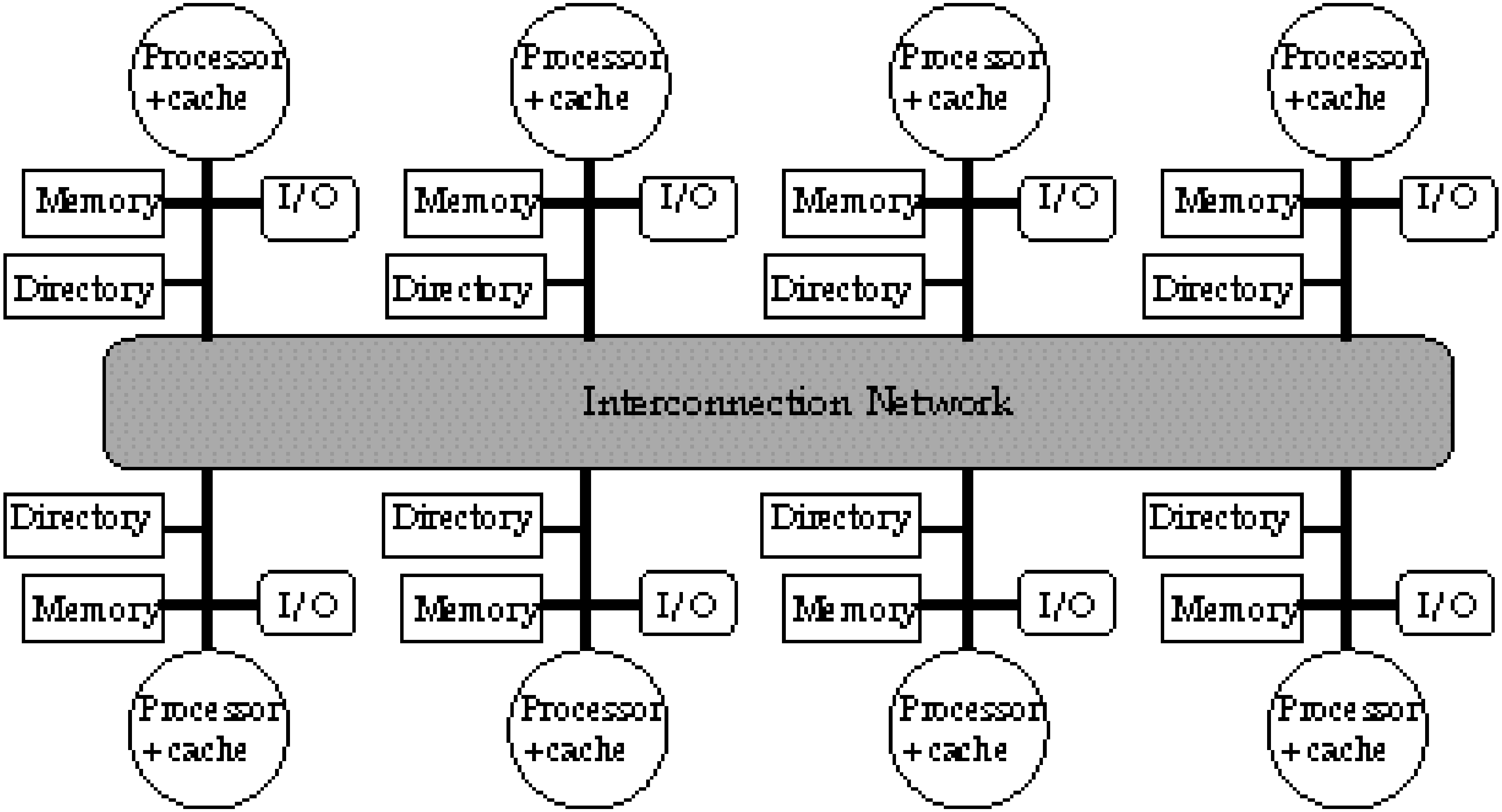
Implementing Snooping Caches

- 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
- 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 that obeys inclusion with L1 cache

Larger MPs

- **Separate Memory per Processor**
- **Local or Remote access via memory controller**
- **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 *ƒ*(memory size) vs. *ƒ*(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 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(r):
  - Writes to non-exclusive data => write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent

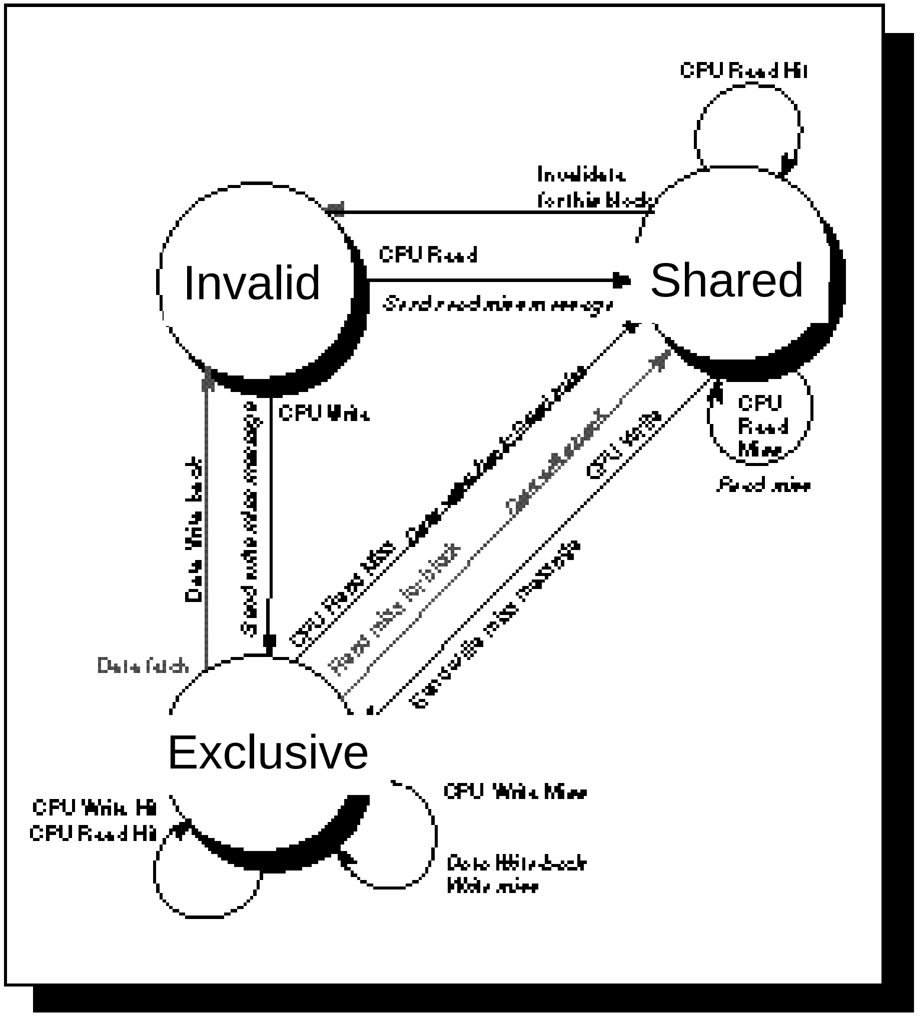
Directory Protocol

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

Directory Protocol Messages

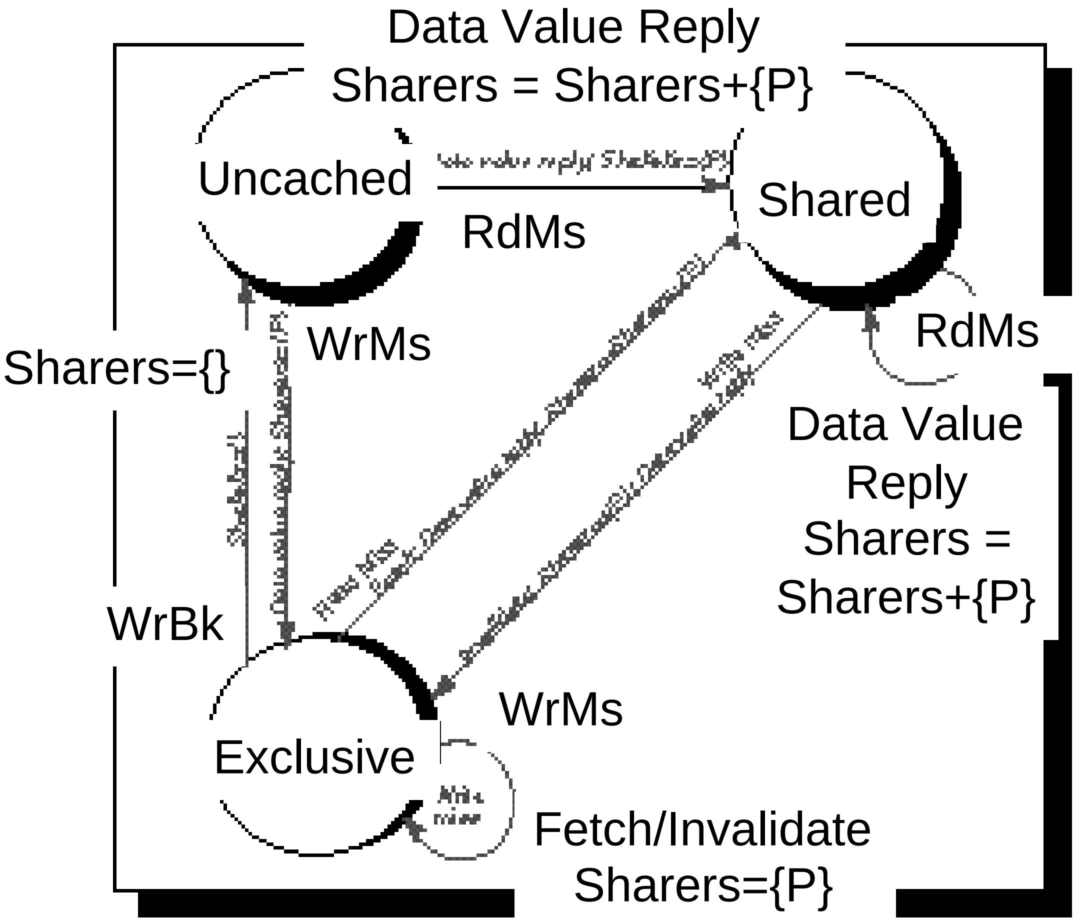
Message type	*Source*	*Destination*	*Msg*
Read miss	Local cache	Home directory	P, A
– *Processor P reads data at address A; send data and make P a read sharer*			
Write miss	Local cache	Home directory	P, A
– *Processor P writes data at address A; send data and make P the exclusive owner*			
Invalidate	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*			
Data write-back	Remote cache	Home directory	A, Data
– *Write-back a data value for address A*			

State Transition Diagram for an Individual Cache Block in a Directory Based System



- States identical to snoopy case; transactions very similar.
- Tranistions caused by read misses, write misses, invalidates, data fetch req.
- Generates read miss & write miss msg to home directory.
- Write misses that were broadcast on the bus => explicit invalidate & data fetch requests.

State Transition Diagram for the Directory



Data Value Reply
Sharers = Sharers+{P}

Uncached

RdMs

Shared

RdMs

WrMs

Sharers={}

Data Value
Reply
Sharers =
Sharers+{P}

WrBk

Exclusive

WrMs

Fetch/Invalidate
Sharers={P}

- **Same states & structure as the transition diagram for an individual cache**
  - **2 actions: update of directory state & send msgs to statisfy req.**
  - **Tracks all copies of memory block.**
  - **Also indicate an action that updates the sharing set, Sharers, as opposed to sending a message.**

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, which causes 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).
  - Data write-back: owner processor is replacing the block and hence must write it back. This makes the 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 bufffers in network (see Appendix E)**
- **Optimizations:**
  - **read miss or write miss in Exclusive: send data directly to requestor from owner vs. 1st to memory and then from memory to requestor**

Example

	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Actio	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

	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Actio	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

	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Add	Value	Actio	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

	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Actio	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				10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	{P1,P2}	10
P2: Write 20 to A1														10
														10
P2: Write 40 to A2														10

A1 and A2 map to the same cache block

Example

	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Actio	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				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														10

A1 and A2 map to the same cache block

Example

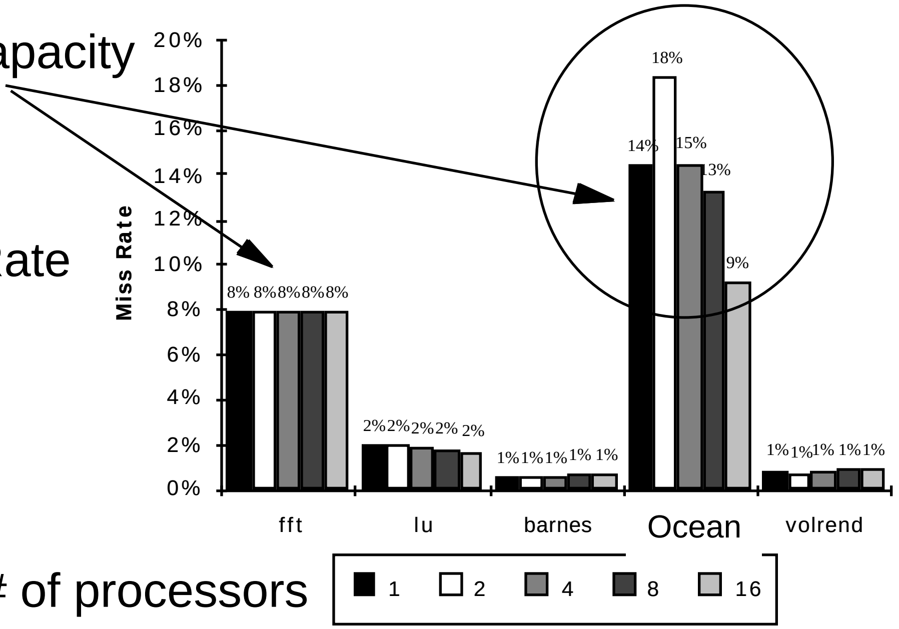
	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Actio	Proc.	Addr	Value	Addr	State	{Procs	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10				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

Miss Rates for Snooping Protocol

- 4th C: Conflict, Capacity, Compulsory and Coherency Misses
- More processors: increase coherency misses while decreasing capacity misses since more cache memory (for fixed problem size)
- Cache behavior of Five Parallel Programs:
  - FFT Fast Fourier Transform: Matrix transposition + computation
  - LU factorization of dense 2D matrix (linear algebra)
  - Barnes-Hut n-body algorithm solving galaxy evolution probem
  - Ocean simluates influence of eddy & boundary currents on large-scale flow in ocean: dynamic arrays per grid

Miss Rates for Snooping Protocol



High Capacity Misses

Miss Rate

# of processors

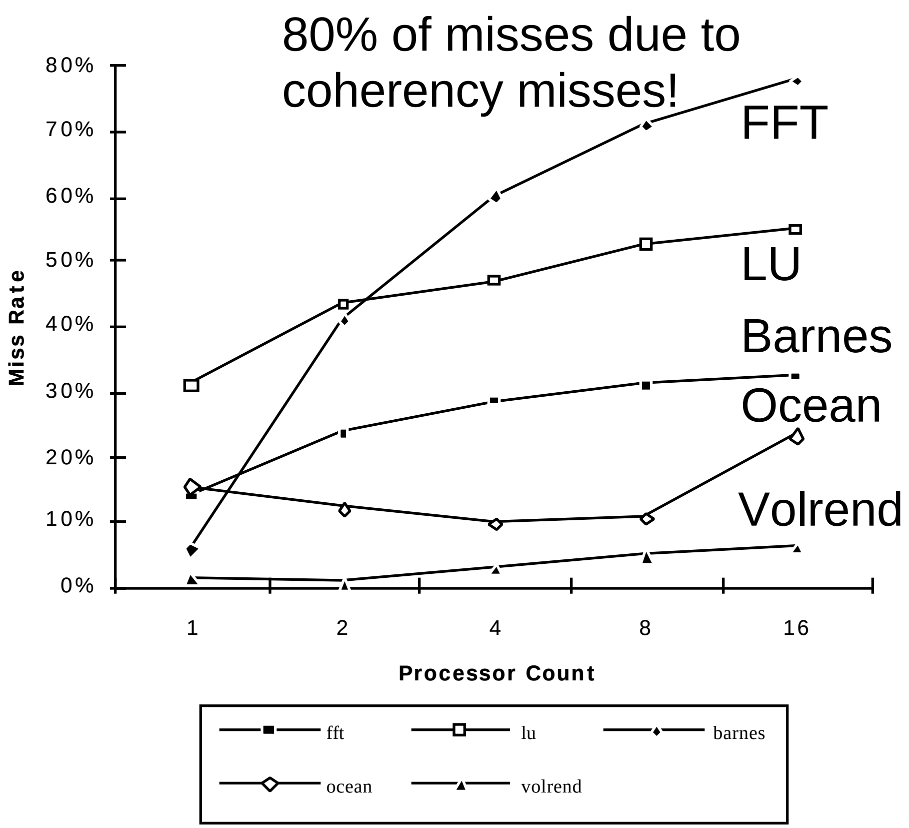
Big differences in miss rates among the programs

– **Cache size is 64KB, 2-way set associative, with 32B blocks.**

– **Misses in these applications are generated by accesses to data that is potentially shared.**

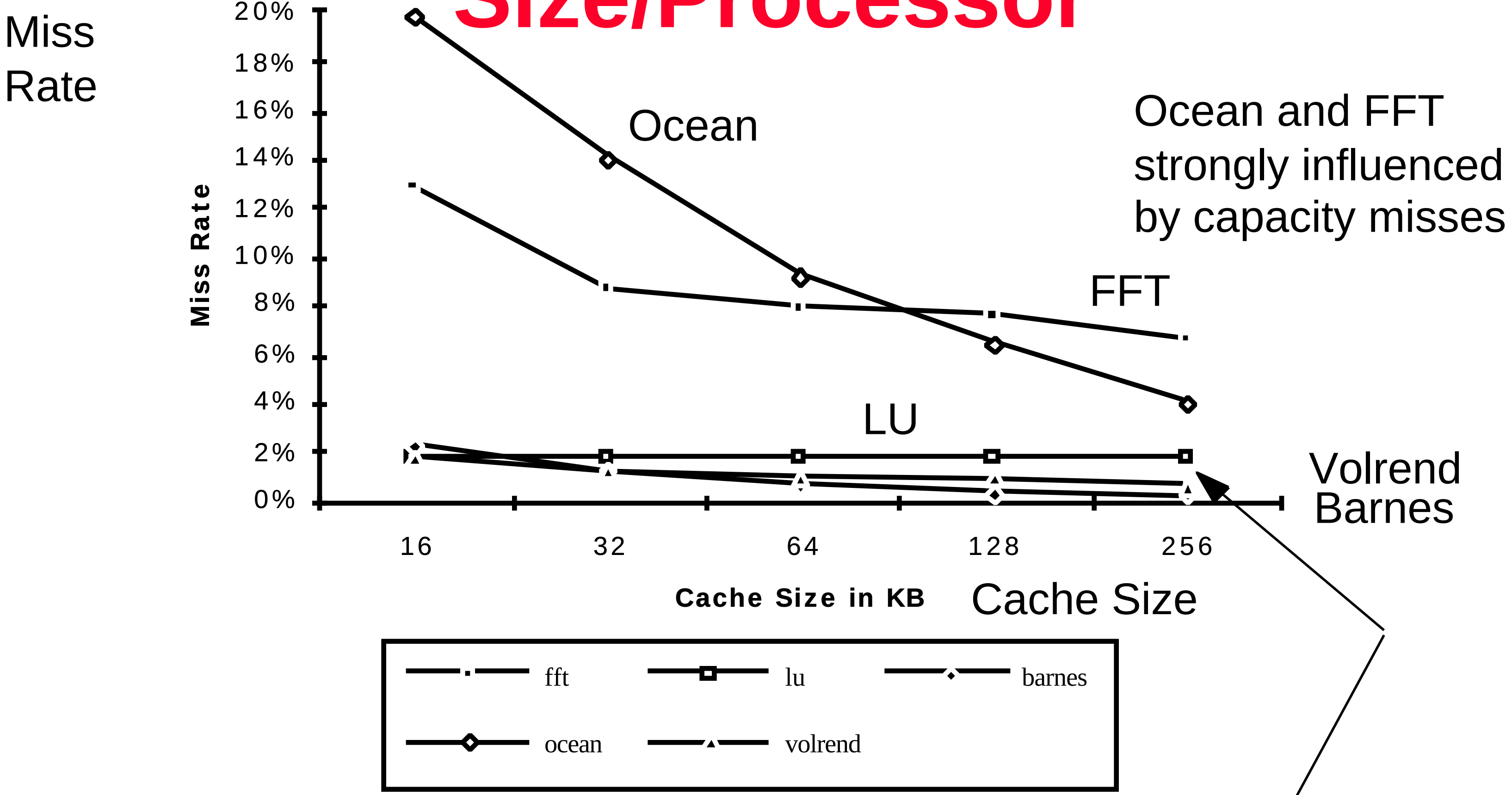
– **Except for Ocean, data is heavily shared; in Ocean only the boundaries of the subgrids are shared, though the entire grid is treated as a shared data object. Since the boundaries change as we increase the processor count (for a fixed size problem), different amounts of the grid become shared. The anamolous increase in miss rate for Ocean in moving from 1 to 2 processors arises because of conflict misses in accessing the subgrids.**

% Misses Caused by Coherency Traffic vs. # of Processors



- % cache misses caused by coherency transactions typically rises when a fixed size problem is run on more processors.
- The absolute number of coherency misses is increasing in all these benchmarks, including Ocean. In Ocean, however, it is difficult to separate out these misses from others, since the amount of sharing of the grid varies with processor count.
- Invalidations increases significantly; In FFT, the miss rate arising from coherency misses increases from nothing to almost 7%.

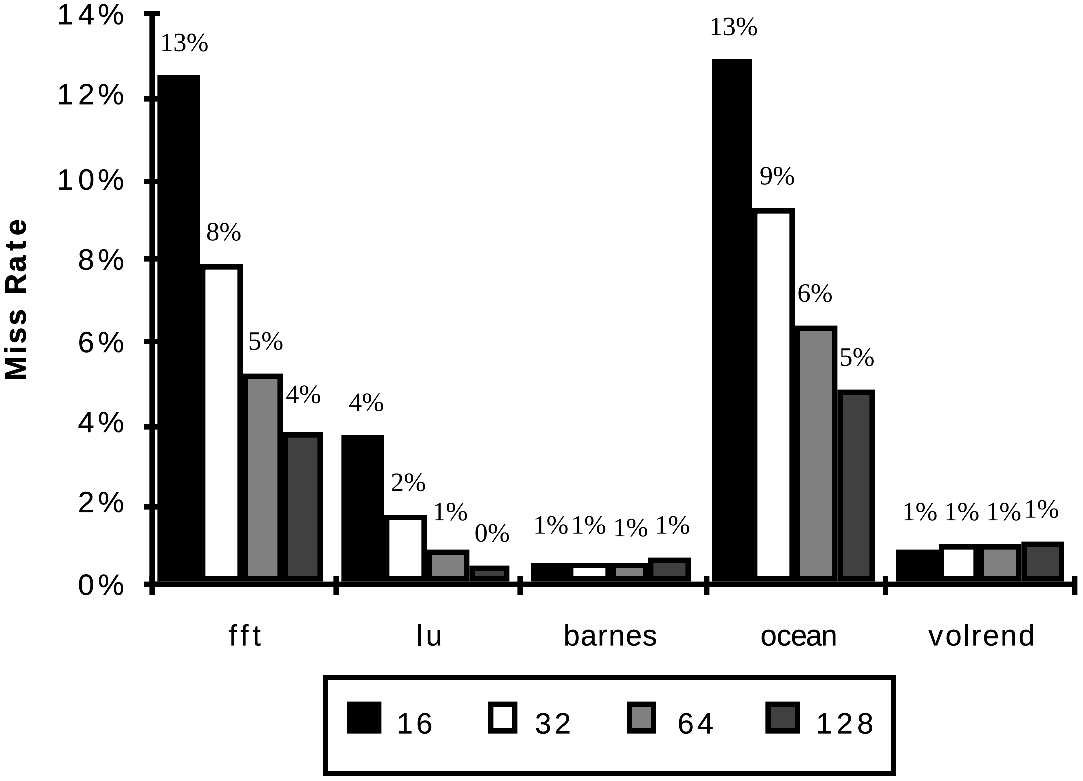
Miss Rates as Increase Cache Size/Processor



- **Miss rate drops as the cache size is increased, unless the miss rate is dominated by coherency misses.**
- **The block size is 32B & the cache is 2-way set-associative. The processor count is fixed at 16 processors.**

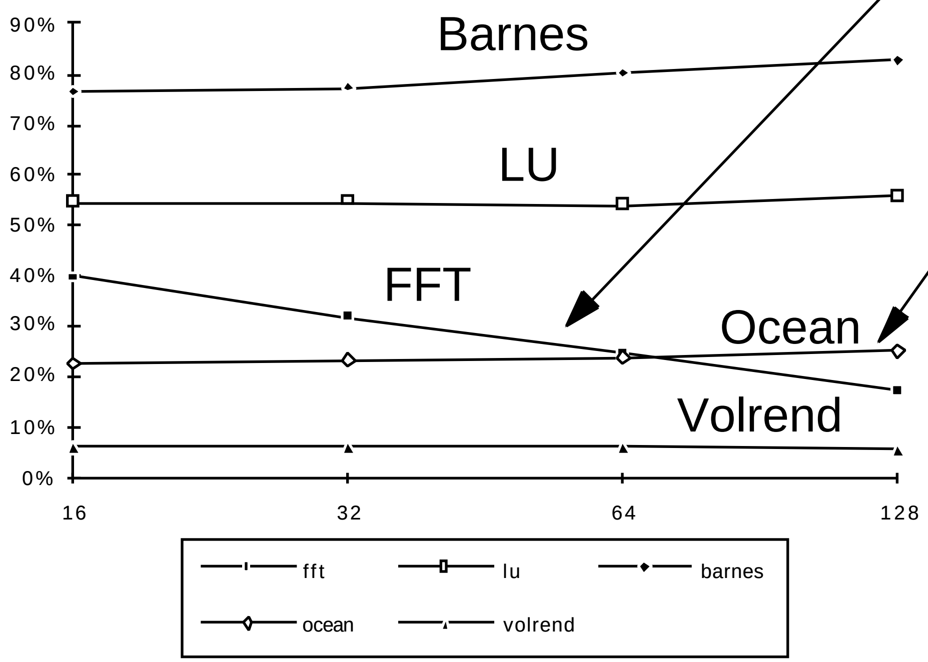
Miss Rate vs. Block Size

- **Since cache block hold multiple words, may get coherency traffic for unrelated variables in same block**
- **False sharing arises from the use of an invalidation-based coherency algorithm. It occurs when a block is invalidated (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written into.**



miss rates mostly fall with increasing block size

% Misses Caused by Coherency Traffic vs. Block Size

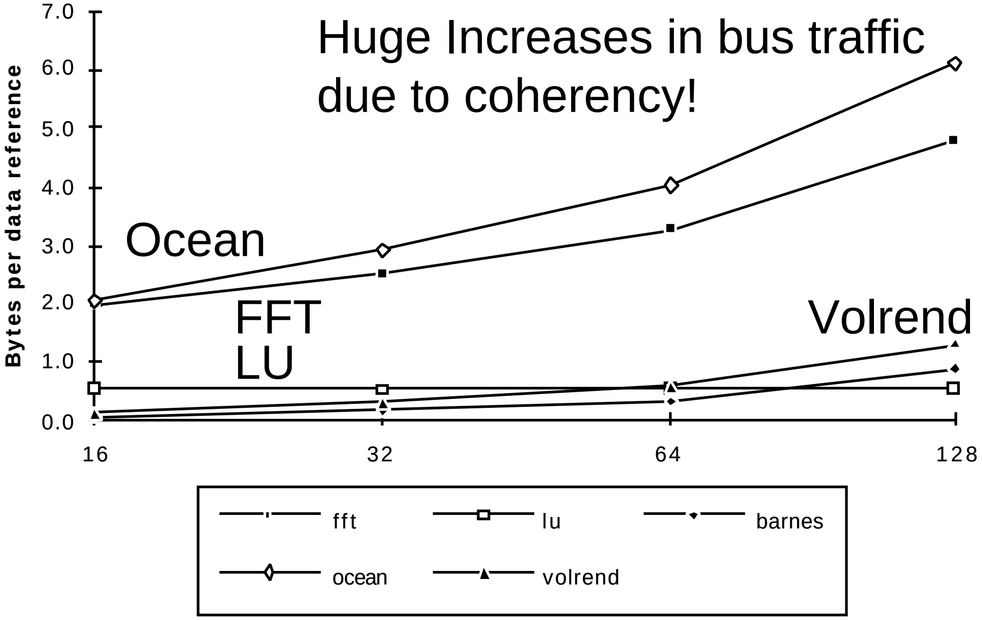


Behavior tracks cache size behavior
FFT: Coherence misses reduced faster than capacity misses!

- FFT communicates data in large blocks & communication adapts to the block size (it is a parameter to the code); makes effective use of large blocks.
- Ocean competing effects that favor different block size
  - – Accesses to the boundary of each subgrid, in one direction the accesses match the array layout, taking advantage of large blocks, while in the other dimension, they do not match. These two effects largely cancel each other out leading to an overall decrease in the coherency misses as well as the capacity misses.

Bus Traffic as Increase Block Size

Bytes per
data ref

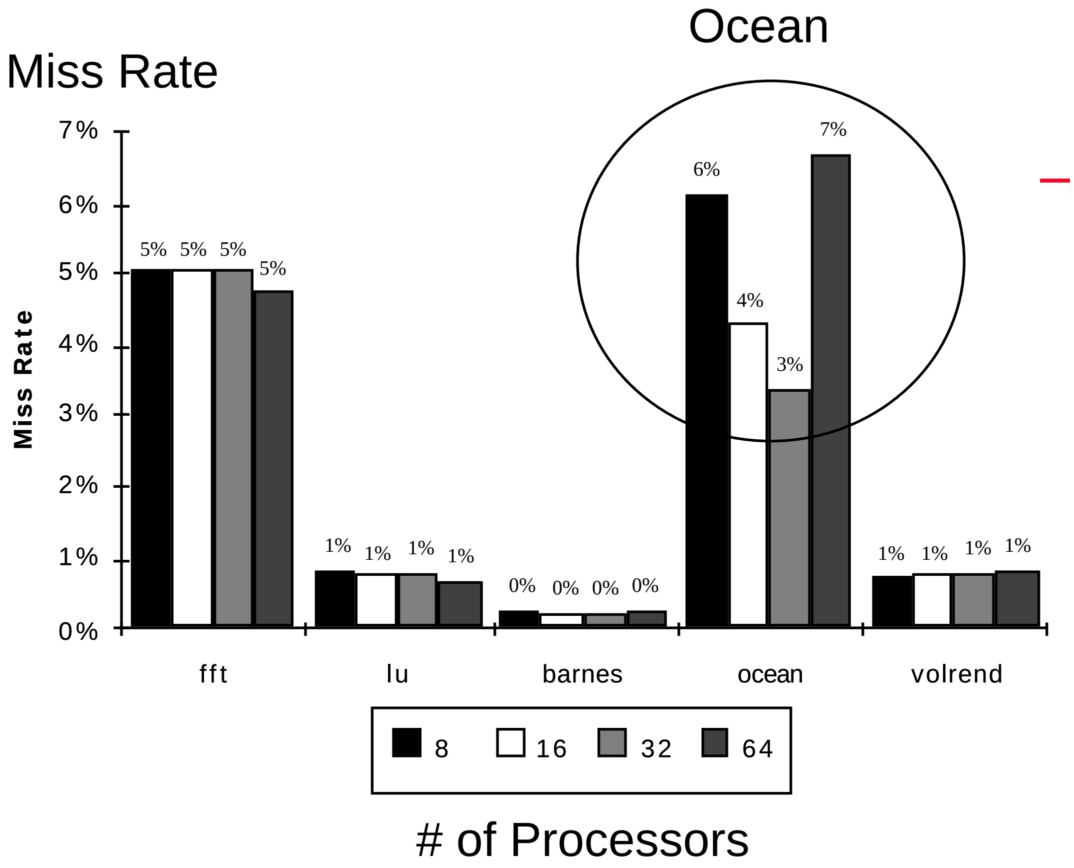


Huge Increases in bus traffic due to coherency!

- **Bus traffic climbs steadily as the block size is increased.**
- **The factor of 3 increase in traffic for Ocean is the best argument against larger block sizes.**
- **Remember that our protocol treats ownership misses the same as other misses, slightly increasing the penalty for large cache blocks: in both Ocean and FFT this effect accounts for less than 10% of the traffic.**

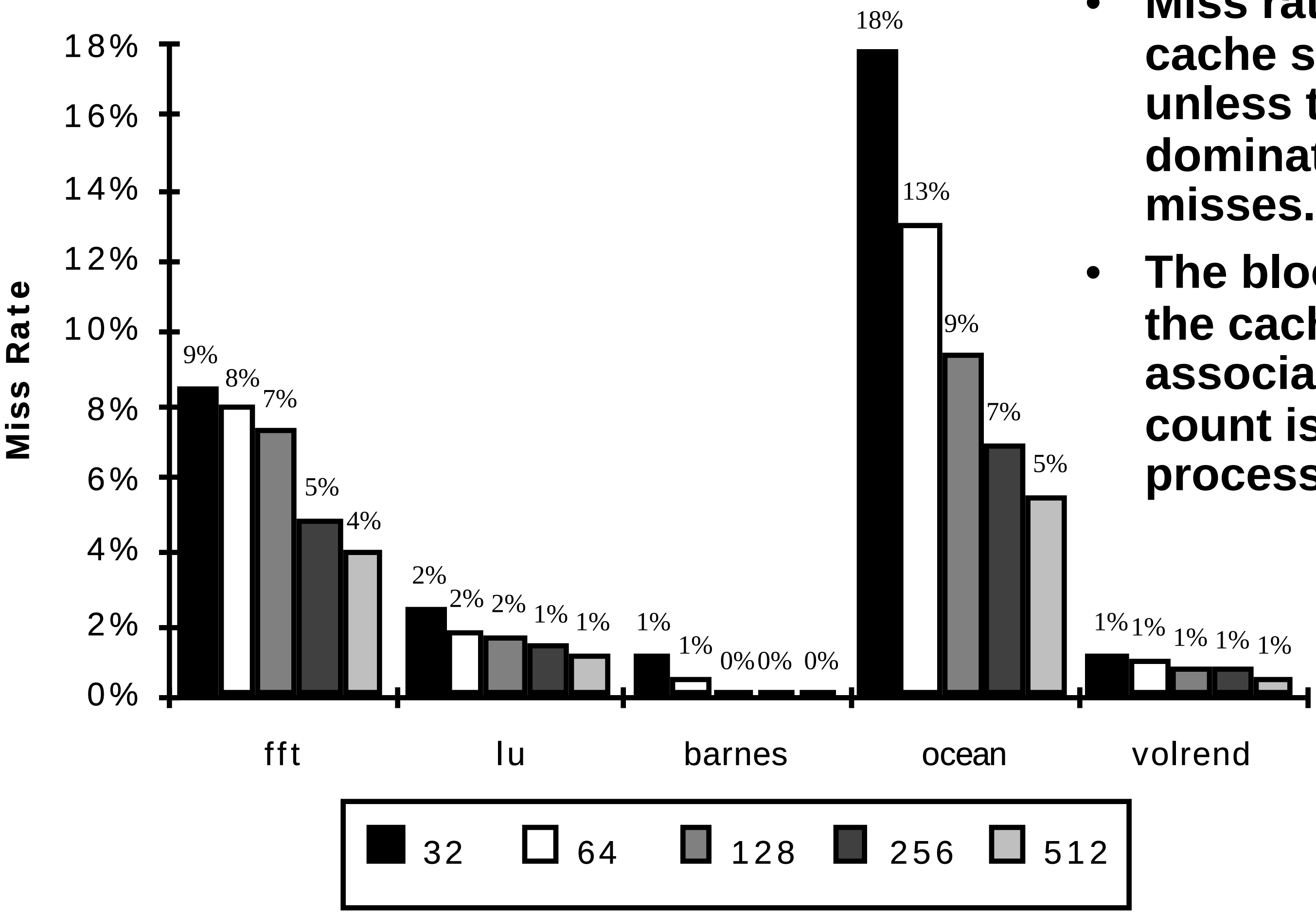
Miss Rates for Directory

Use larger cache to circumvent longer latencies to directories



- Cache size is 128 KB, 2-way set associative, with 64B blocks (cover longer latency)
- Ocean: only the boundaries of the subgrids are shared. Since the boundaries change as we increase the processor count (for a fixed size problem), different amounts of the grid become shared. The increase in miss rate for Ocean in moving from 32 to 64 processors arises because of conflict misses in accessing small subgrids & for coherency misses for 64 processors.

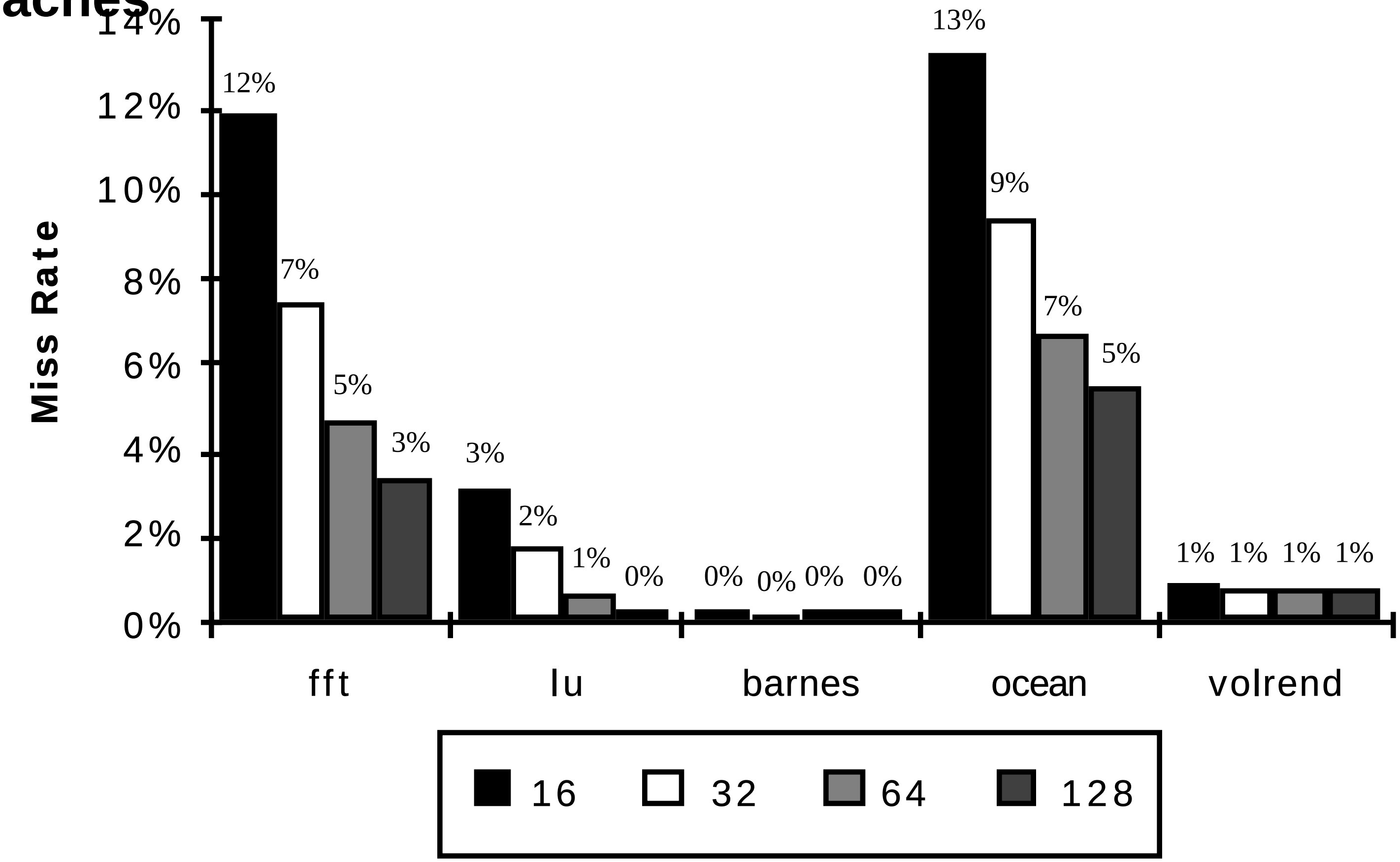
Miss Rates as Increase Cache Size/Processor for Directory



- Miss rate drops as the cache size is increased, unless the miss rate is dominated by coherency misses.
- The block size is 64B and the cache is 2-way set-associative. The processor count is fixed at 16 processors.

Block Size for Directory

- **Assumes 128 KB cache & 64 processors**
  - **Large cache size to combat higher memory latencies than snoop caches**



Summary

- Caches contain all information on state of cached memory blocks
- Snooping and Directory Protocols similar; bus makes snooping easier because of broadcast
- Directory has extra data structure to keep track of state of all cache blocks
- Distributing directory => scalable shared address multiprocessor

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

Uninterruptable Instruction to Fetch and Update Memory

- **Atomic exchange: interchange a value in a register for a value in memory**
  - 0 => synchronization variable is free
  - 1 => synchronization variable is locked and unavailable
  - – Set register to 1 & swap
  - – New value in register determines success in getting lock
    0 if you succeeded in setting the lock (you were first)
    1 if other processor had already claimed access
  - – Key is that exchange operation is indivisible
- **Test-and-set: tests a value and sets it if the value passes the test**
- **Fetch-and-increment: it returns the value of a memory location and atomically increments it**
  - – 0 => synchronization variable is free

Uninterruptable Instruction to Fetch and Update Memory

- Hard to have read & write in 1 instruction: use 2 instead
- Load linked (or load locked) + store conditional
  - Load linked returns the initial value
  - Store conditional returns 1 if it succeeds (no other store to same memory location since 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?
```

Steps for Invalidate Protocol

Step	*P0*	*$*	*P1*	*$*	*P2*	*$*	*Bus/Direct activity*
1.	Has lock	Sh	spins	Sh	spins	Sh	None
2.	Lock<– 0	Ex		Inv		Inv	P0 Invalidates lock
3.		Sh	miss	Sh	miss	Sh	WB P0; P2 gets bus
4.		Sh	waits	Sh	lock = 0	Sh	P2 cache filled
5.		Sh	lock=0	Sh	exch	Sh	P2 cache miss(WI)
6.		Inv	exch	Inv	r=0;l=1	Ex	P2 cache filled; Inv
7.		Inv	r=1;l=1	Ex	locked	Inv	WB P2; P1 cache
8.		Inv	spins	Ex		Inv	None

For Large Scale MPs, Synchronization Can Be a Bottleneck

- 20 procs spin on lock held by 1 proc, 50 cycles for bus

Read miss all waiting processors to fetch lock	1000
Write miss by releasing processor and invalidates	50
Read miss by all waiting processors	1000
Write miss by all waiting processors ,	
one successful lock, & invalidate all copies	1000
Total time for 1 proc. to acquire & release lock	3050

  – Each time one gets a lock, drops out of competition= 1525

  – 20 x 1525 = 30,000 cycles for 20 processors to pass through the lock

  – Problem is contention for lock and serialization of lock access: once lock is free, all compete to see who gets it

- Alternative: create a list of waiting processors, go through list: called a “queuing lock”

  – Special HW to recognize 1st lock access & lock release

- Another mechanism: fetch-and-increment; can be used to create barrier; wait until everyone reaches same point

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

Memory Consistency Model

- **Schemes faster execution to sequential consistency**
- **Not really 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**

CS 252 Administrivia

- Next reading is Chapter 8 of CA:AQA 2/e and Sections 1.1-1.4, Chapter 1 of upcoming book by Culler, Singh, and Gupta:

`www.cs.berkeley.edu/~culler/`

- Remzi Arpaci will talk Fri. 11/8 on Networks of Workstations and world record sort
- Dr. Dan Lenowski, architect of SGI Origin, talk in Systems Seminar Thur. 11/14 at 4PM in 306 Soda
- Next project review: survey due Mon. 11/11; 20 min. meetings moved to Fri. 11/15; signup Wed. 11/6

Key Issues for MPs

- Measuring Performance
  - Not just time on one size, but how performance scales with P
  - For fixed size problem (same memory per processor) and scaled up problem (fixed execution time)
  - What about accuracy of result as scale size of problem? e.g., may need more iteractions to converge
  - Care to compare to best uniprocessor algorithm, not just parallel program on 1 processor (unless its best)
- Multilevel Caches, Coherency, and Inclusion
  - Invalidation at L2 cache forces invalidation at higher levels if caches adher to the inclusion property
  - But larger L2 blocks lead to several L1 blocks getting invalidated

Key Issues for MPs

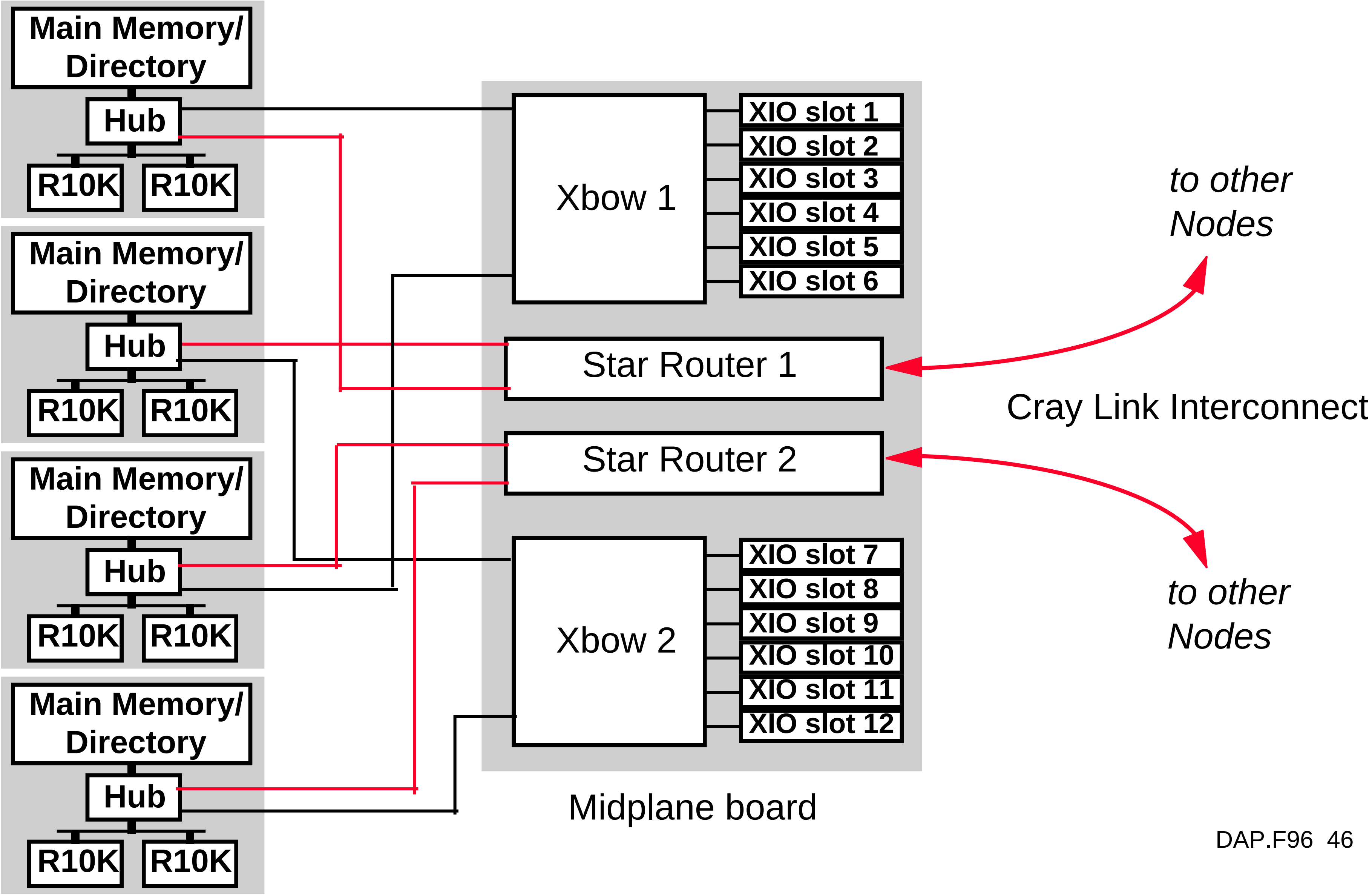
- **Nonblocking Caches and Prefetching**
  - More latency to hide (misses longer), so nonblocking caches even more important
  - Makes sense if there is available memory bandwidth; must balance bus utilization, false sharing (conflict w/ other processors)
  - Want prefetch to be coherent (“nonbinding” to local copy): in cache, not in a register, so that update can invalidate
- **Virtual Memory to get Shared Memory MP: Distributed VIrtual Memory (DVM) or Shared Virtual Memory (SVM)**
  - pages are units of coherency (false sharing bigger problem)
  - SW implements coherency (higher overhead)
  - Improve performance with HW to reduce overhead, variable page size, compiler optimizations

Example: SGI Origin2000

- Directory-based machine with network connecting up to 128 processors: “Scalable Shared memory Multiprocessor (S2MP)”
  - Deskside/Rack/Row does up to 8/16/128 procs
- Based on 3 Crossbar switches (ASICs)
  - Hub: 4-way DSM communication/coherence controller connects processor to memory cntrlr, network, I/O
  - Crossbow (“Xbox”): 8-way crossbar connects 6 I/O interfaces to 2 nodes
  - Router: 6-way crossbar forms the interconnection network at 800 MB/sec: : “Cray Link Interconnect”
- OS migrates pages near processor that uses data

Example: 4 Node SGI Origin

4 Node boards/ 8 processors



Example: SGI Origin 2000

- **Node:**
  - – Dual MIPS 195 MHz, R10000s connected by bus
  - – Each processor has own L1 and L2 cache
  - – One directory entry per memory block
  - – up to 2 GB of memory/node
  - – Directory information stored in local memory for up to 64 processors; 128 require extra DRAM per board
  - – Communication/coherence controller(“Hub”) is 4-way crossbar: proc. bus , memory controller, network, I/O

Example: SGI Origin

- Hub
  - Controller and Network Interface ASIC
  - Single chip crossbar-switch
  - sits on system bus
  - connects processors, local memory, network interface(“Router”), and I/O interface (“Xbow”)
    - Xbow is single chip cross-bar switch for I/O devices: up to 480 MB/sec
  - implements directory protocol
  - has block transfer engine
  - supports up to 12 oustanding memory requests per processor
  - supports load linked, store conditional

Example: SGI Origin2000

- Network:
  - Composed of other single-chip crossbar-switches: “Router”
  - Router has 6 ports, full duplex
  - Differential cabling up to 15 meters longs
  - Link Bandwidth: 800 MB/sec
  - Network has 4 virtual channels
  - Wormhole routing
  - Packet size is 128 bits
  - Topology is Hypercube

Example: SGI Origin2000

- Coherency:
  - Illinois/MESI cache states: Invalid, Shard, Dirty, Clean-Exclusive
  - Snoopy coherence on system bus for 2 processors
  - Directory states: Unowned, Shared, Exclusive, + 3 pending or busy states (home received request but not yet completed it)
  - Sequential consistency