Shared Memory Multiprocessors

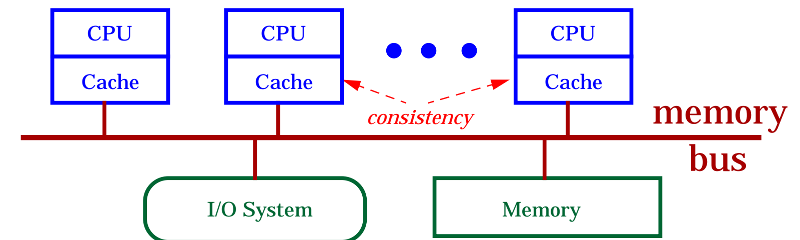
step 1: SMP's

- Symmetric multiprocessor characteristics
  - global physical address space
    - ~ symmetric access to all memory from any processor
  - each processor has it's own cache (1 or more levels)
    - start with single level to simplify the initial discussion
  - physically shared main memory
    - hence easy to export shared memory programming model
  - message passing model requires a thin software layer
    - often faster than real message passing machines since OS isn't involved
- Dominant parallel architecture today
  - server class throughput engines
  - PE count is modest = 2 to 128 processors
    - 2 and 4-way SMP boards are relatively cheap

Bus Based Shared Memory

center of commercial multiprocessor focus

- Low cost interconnect
  - small-way: usually 2 to 4 processors
  - market for these high end *servers* is increasing dramatically
  - heavy influence/limitation imposed by the bus design
  - note simple broadcast capability



Private vs. Shared Data

SMP must support both

- Private
  - normal cache policies and benefits
- Shared
  - simplifies communication between the multiple processors
    - simple reads and writes to memory
    - obvious synchronization requirement however
  - introduces consistency & coherence problems
    - shared data may be replicated in the multiple caches
    - in order to preserve uniprocessor (***sequential consistency***) semantics all copies must be identical
    - writes are clearly the source of potential confusion
  - consider the drawbacks of conventional cache wisdom
    - write-back, large block/line sizes, tag bit semantics - *problems?*
    - write-buffers - *problems?*

Consistency and Coherence

- Informal coherence model
  - any read of a data item should return the value that was most recently written
  - appealing but way too simple in the SMP world
- Simple model contains 2 behavioral aspects
  - coherence
    - defines ***what*** value is returned by a read
  - consistency
    - defines ***when*** a written value will be returned by a read
    - this is a problem since a write at processor 1 may have happened but still not have left the processor by the time a read at processor 2 happens
    - note that this *when* problem gets more difficult as the physical extent of the multiprocessor system increases
  - both critical to writing correct shared memory programs

Coherence

A memory system is coherent if:

- given no other writes, a processor P will get its last written value to X on a read of X
  - simply preserves program order and is the normal expectation
- given no other writes, a read by P1 of X gets the value written by P2 if the read and write are *sufficiently* separated
  - we'll defer the issue of what *sufficient* means and how it is controlled for now
- writes to the same location are serialized
  - fundamental need to avoid the concurrent writer problem

❑ Implication

- reordering reads is OK
  - similar to uniprocessor world
- writes must finish in program order (*write serialization*)
  - definitely not the same as in the uniprocessor world
  - this restriction will be relaxed later

Enforcing Coherence

2 common styles

❑ Directory Based

- keep sharing and block status in a directory
  - directory may be centralized or distributed
- appropriate for DSM

❑ Snooping

- appropriate for SMP's (our focus for today)
- take advantage of the common connection to the bus
- caches monitor transactions on the bus
  - see writes to shared data
  - modify contents of their copy if they have one
  - what options are there for modifications??
- what are the costs of this approach?

Protocols

2 common choices

❑ Write-invalidate

- writer needs to get exclusive copy
- write forces other copies to be invalidated
- subsequent reads get new copy from the writer
  - what state change is required for the writer's copy
- if 2 writers - then one wins the *race*
  - removing races requires some form of synchronization
  - in single bus case we get it by default - e.g. via bus arbitration

❑ Write-update

- broadcast writes on bus - snoopers update blocks if they have a copy
  - knowing which lines are shared helps minimize bus contention
- how do we deal with the multiple writer race?

Performance Issues

3 characteristic differences

❑ Multiple writes to the same word

- less bus traffic in invalidate
  - first write causes a transaction with invalidate
  - every write causes a transaction with update

❑ Multiple word cache blocks

- same issue but applied at block granularity
  - invalidate = 1 bus transaction per line per new writer
  - update = 1 bus transaction for every write

❑ Reading a remotely written value

- update wins here since it has a local copy

Result

- bus bandwidth issues are critical - hence most common winner is invalidate
- OK for small way SMP - (still true when CPU's get faster?)
- does every line need to use the same protocol?

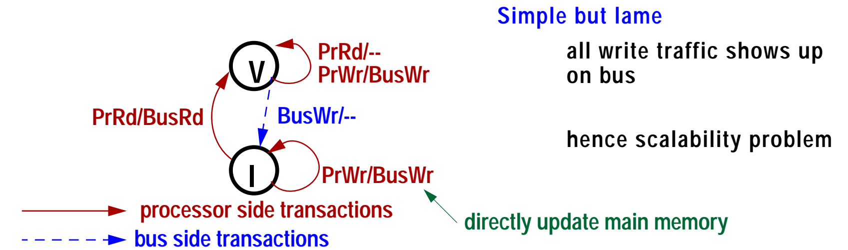
Cache Implementation

- Single cache
  - each line has a state: invalid, valid, dirty, etc. (*status tags*)
  - processor generates read and write transactions
  - controller
    - reacts to processor transactions
    - matches *address tags* to determine hit or miss
    - on a hit: modifies the state of the line as needed (finite state machine model)
    - generates bus transactions to main memory: read or write
- Snooping coherent cache
  - similar game but with multiple distributed controllers
  - plus a new source of transactions
    - controllers must watch the bus as well as the processor
    - any potential problems w/ this dual master situation?

Simple Example

write-through no-allocate write-invalidate coherence

- Basics
  - Processor read and write ::= PrRd & PrWr transactions
  - Bus read and write: BusRd & BusWr
  - Line state I or V
    - write-through implies all writes are seen by the memory & other processors
    - hence no dirty state is necessary



Simple but lame

all write traffic shows up on bus

hence scalability problem

Consistency

- Sequential consistency
  - Lamport's 1979 definition
    - SC if the result of any execution is the same as if the operations were executed in some sequential order …
    - really is saying that the multiple total orders of all threads/programs can be arbitrarily interleaved
    - one interleaved order must be the one observed
- Sufficient conditions
  - every process issues memory requests in program order
  - after a write the issuing process waits until the write is complete before proceeding
  - after a read, the issuing process waits for the read to complete
    - implies that previous write to this location must also complete to all processors
    - this write atomicity can be quite demanding in modern processors where out of order and speculative everything is the norm

Snooping Protocol Design Space

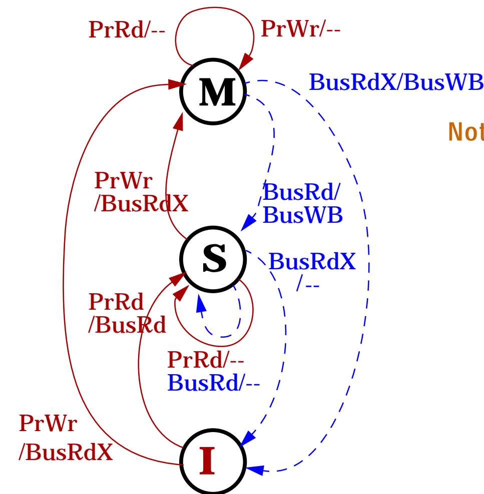
- The Beauty
  - single bus owner and snooping broadcast used to enforce write atomicity
  - only a small amout of additional effort is required to permit multiprocessor cache coherent operation
- and The Beast
  - bus design is now critical
    - must support some additional transactions
  - cache controller has dual masters
    - contention possibility may require *I'm not ready* signals
    - e.g. bus and CPU try for the same line - tag and line contention
  - new cache states will be necessary - for example
    - exclusive: owner (OK to write) but currently clean
    - dirty: owner (OK to write) but must respond to reads by other processors
    - many actual options

MSI Protocol

write-invalidate write-back

- Line states: Invalid, Shared, Modified
- Processor transactions: PrRd, PrWr
- Bus transactions
  - BusRd - asks for a copy of the line with no intent to modify
    - generated as a result of a PrRd miss
    - line may be supplied by another cache or by the main memory
  - BusRdX - asks for an exclusive copy of the line
    - generated as a result of a PrWr miss
    - or from a PrWr hit on a line that is not in the M state
    - note new bus transaction that is imposed by the need for cache coherence
  - BusWB - writeback or flush
    - imposed by the write-back cache policy
    - a minor extension to the BusWr idea but for lines rather than smaller data chunk size

MSI State Diagram



Note:

1: Bus WB supplies data to cache as well as to memory

2. writing to shared block is a problem

BusRdX can be used in 2 situations

Normal

BusUpgr - bus upgrade is one common optimization - in this case no bus occupancy and bandwidth is lost since a data return is not needed for this transaction

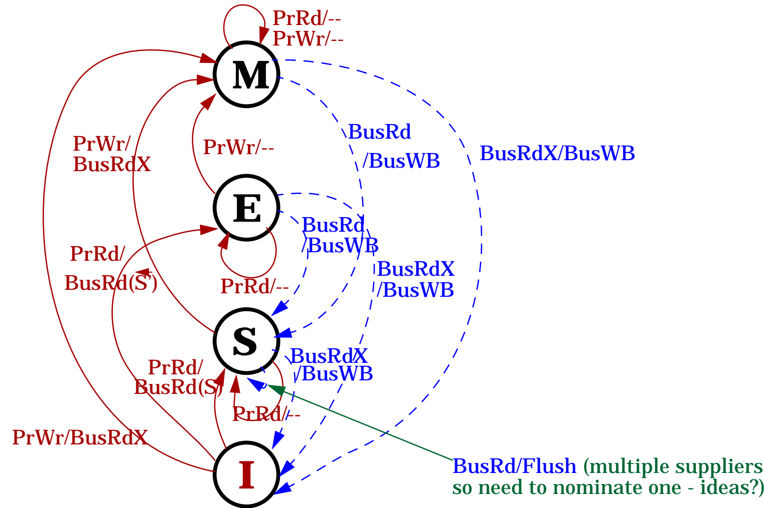
MSI Analysis

- SC
  - write completion is detected when BusRdX is seen on the bus
    - and the data return is interned in the cache and the pending write is issued
  - bus ownership guarantees atomicity
    - but note the possible delayed data return to an immediate next BusRdX
- Options
  - BusRd from M goes to S
    - could also have gone to I (choice for the Synapse machine) = migratory
  - tradeoff
    - if new processor is likely to write soon then going to I is better
    - if old processor is likely to read again soon then going to M is better
  - hybrid is possible based on a protocol bit
    - Sequent Symmetry Model B & MIT Alewife made this choice
  - protocol flexibility adds both performance and cost
    - the real question is how much of each??

MESI Protocol

- Deals with PrRd followed by PrWr problem
  - which generates 2 bus transactions in the MSI protocol
    - even when no sharers exist
  - adds E (exclusive state)
    - intermediate binding between the S and M states
    - means exclusive clean - e.g. memory is consistent
    - M - now means exclusive dirty - e.g. memory is inconsistent
    - S now means 2 or more sharers and memory is consistent
    - I is the same
- S semantics has an additional implication
  - a shared signal must be added to the bus
    - single wire is sufficient, implemented via a wired OR
    - BusRd(S) - indicates the shared signal is asserted on a Bus read
    - Bus Rd(S') - indicates the shared signal is not asserted on a Bus read
    - Bus Rd - means we don't care about the shared signal

MESI State Diagram



MESI Analysis

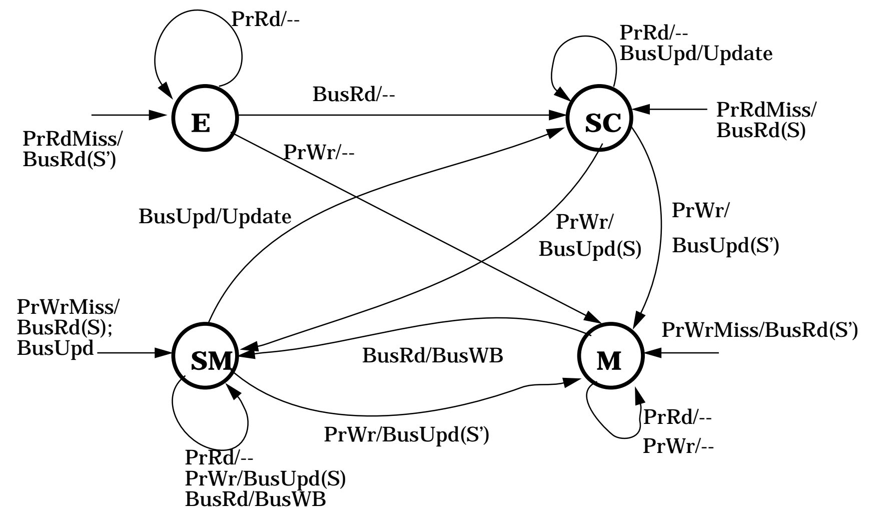
- Flush issues
  - clearly don't want to deal with redundant suppliers when a new sharer comes on line
  - one simple model
    - all Exclusive requests are seen by all
    - hence snooping arrangement means the last exclusive owner knows who they are
    - hence might as well use that
- cache to cache transfer or mem to cache
  - cache can supply requested line faster
    - hence used in machines such as DASH
    - but at the risk of cache interference on the supplier side big problem at today's speeds - may complicate bus design
  - memory is slower
    - but no interference and the flush issue disappears since memory is consistent

Dragon Protocol

write-back write-update

- History
  - first done in the Xerox PARC Dragon
  - then subsequently modified somewhat for Sun's SparcServer machines
- States
  - E - exclusive clean - only one cache has this copy - memory OK
  - SC - shared clean - two or more processors have a copy - this one is clean but one of the others may not be hence memory OK isn't known
  - SM - shared modified - 2 or more copies this one must be used to update main memory
  - M - modified - exclusive dirty
  - no I state: although it is implicitly there via an initialization hack and not necessary subsequently since the update protocol doesn't invalidate
- New bus transactions
  - BusUpd - update bus with the same shared S and S' distinction

Dragon Protocol



Dragon Analysis

- Cache Replacement
  - also ignored in the other protocols
    - what needs to be changed
  - should others be notified via a bus transaction or not?
    - think about what's in the critical path
    - since we care about overall performance not just a particular transaction
- General discussion
  - what happens as we go to multilevel caches?
  - what happens as we go to split transaction bus designs?
    - today's norm and imposed by 800MHz + bus clock speeds
  - what happens as the compiler reorders instructions?
  - what happens as the machine reorders and speculatively executes instructions?

2 Ported Cache Controller

look at CPU and the bus = problems

- Problem source = tag interference
  - either transaction requires checking the same tags
  - consider the meaning w.r.t. cache organization options
    - fully associative, direct mapped, and the hybrid set-associative ??
- 2 options - both can create stalls
  - duplicating tags or ports (which organization applies to each?)
    - when can the snoop completion be delayed
  - multilevel cache with *inclusion*
    - higher levels are subsets of lower levels
    - lower levels filter interference from L1
    - still when L1 copy exists the same snoop delay can occur = stall CPU
    - of course L2 cache tags could also be duplicated to further reduce contention but this starts to get expensive

Note: 2 key issues

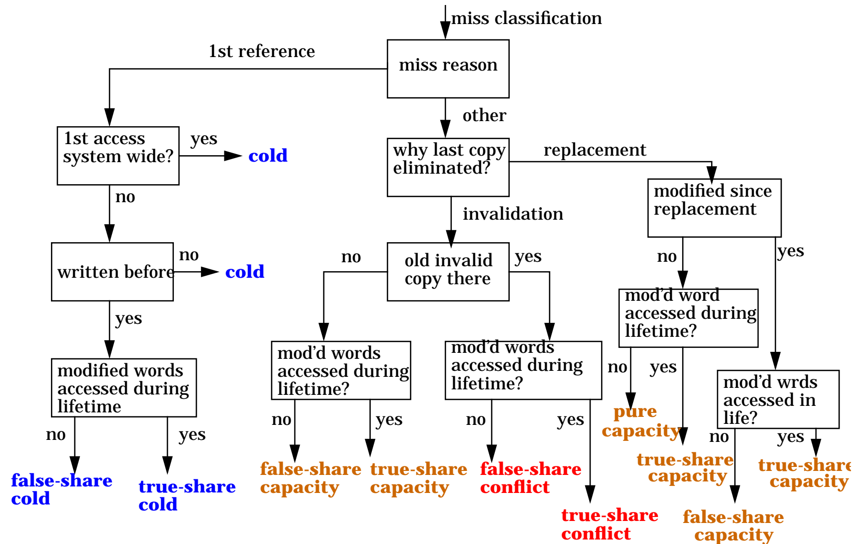
- Local cache state
  - is now extended to include MESI, MSI, etc. + V, D, I status tags
    - hence policy for shared vs. non-shared data may vary
    - e.g. write to a shared line effectively writes through
    - write to a private line may follow a write-back policy
    - unit of accounting is the cache line
- New miss source = coherence miss
  - 2 critical subtypes since hit/miss is to a line
    - true shared miss ==> miss caused by read and writes to same target
    - false sharing ==> sharing of line but not the actual datum

Classifying Misses

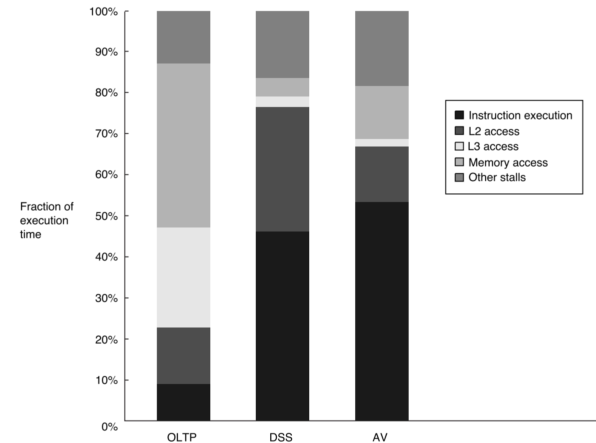
in a particular reference stream

- Idea
  - define the *lifetime* for a block in the cache
  - do per word accounting
    - this line invalidated due to word FOO reference in another processor
  - then distinguish between the various miss types

Miss Classification



Commercial Workload Execution Time Profile

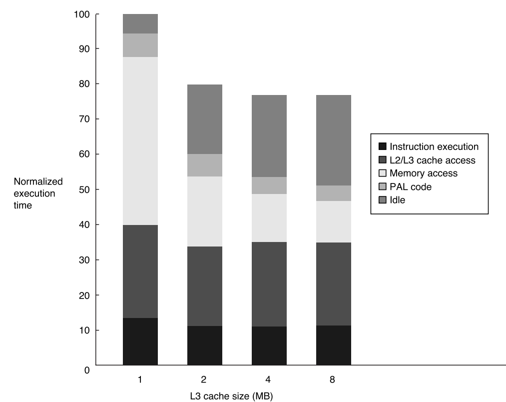


note: legend is inverted for this and subsequent figures

© 2003 Elsevier Science (USA). All rights reserved.

Commercial: Varying External L3 Size

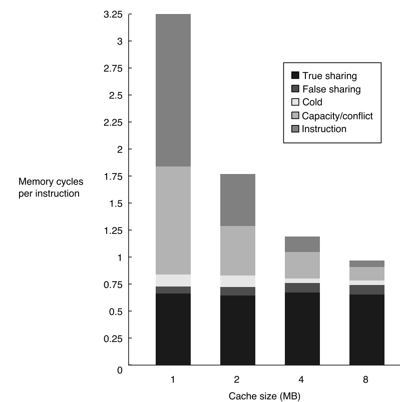
modeled as 2 way set-associative



© 2003 Elsevier Science (USA). All rights reserved.

Commercial: Memory Cycle Components

vs. L3 size



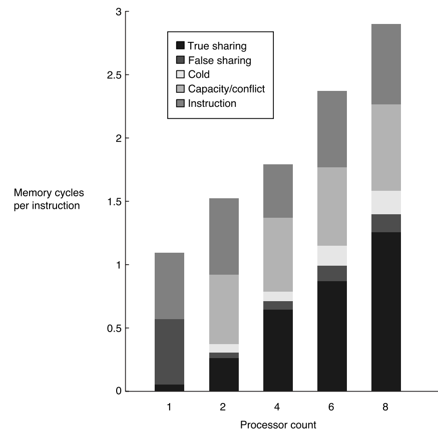
Instruction and capacity/conflict references dominate for small L3 and become insignificant with larger L3

major cause is that instruction footprint has poor spatial locality

© 2003 Elsevier Science (USA). All rights reserved.

Commercial: Sharing Cost

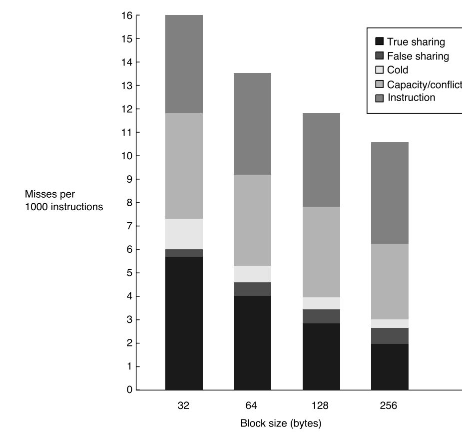
tends to go up with processor count



© 2003 Elsevier Science (USA). All rights reserved.

Commercial: L3 Block Size

false sharing increases & instruction misses decrease

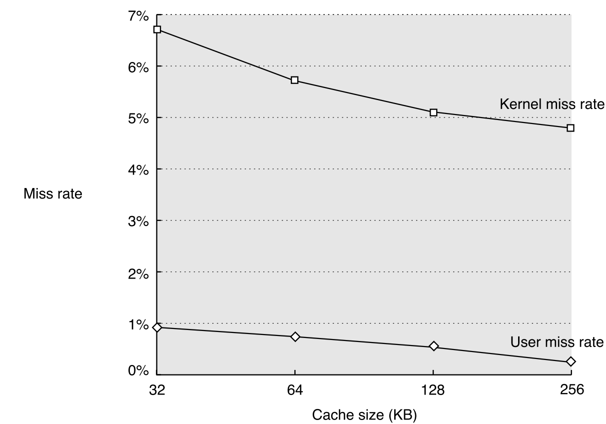


© 2003 Elsevier Science (USA). All rights reserved.

Multiprogrammed Workload Performance

- **Model**
  - 2 independent makes on 8 processors
- **2 components**
  - kernel vs. user code
  - interference increases cold, capacity, and conflict miss rates

Kernel & User Code Miss Rate vs. Size



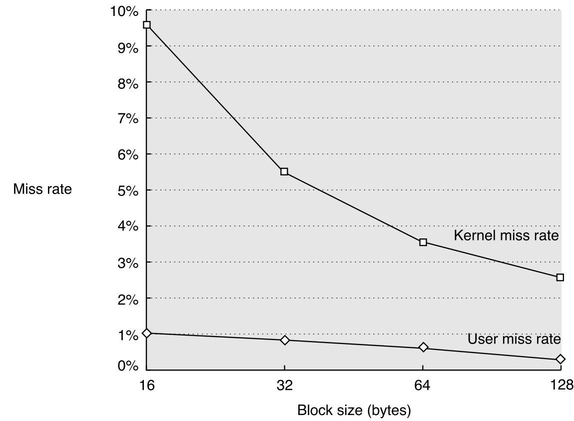
8 CPU's

user = 3x

kernel = 1.3x

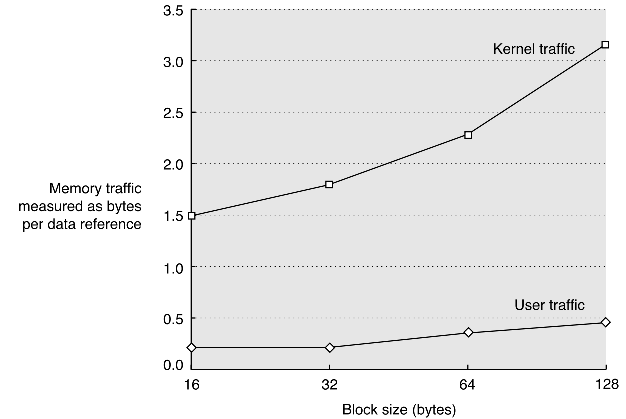
© 2003 Elsevier Science (USA). All rights reserved.

Miss Rates vs. Block Size



© 2003 Elsevier Science (USA). All rights reserved.

Larger Block Size vs. Memory Traffic/Referenc



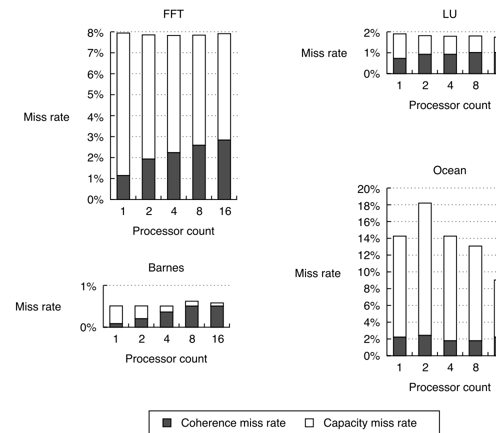
© 2003 Elsevier Science (USA). All rights reserved.

Scientific Workload

problem sizes

- Barnes-Hut
  - 16K bodies for 6 time steps
- FFT
  - 1M complex points
- LU
  - 512 x 512 matrix
  - block size = 16 x 16
- Ocean
  - 130 x 130 grid

Miss Rate vs. PE's



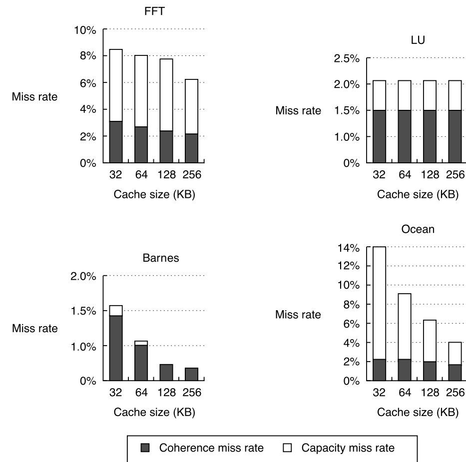
For 64 KB cache 2 way SA and 32 byte blocks

individual coherence patterns

small cache ==> capacity misses dominate for the problem sizes here

© 2003 Elsevier Science (USA). All rights reserved.

Miss Rate vs. Cache Size

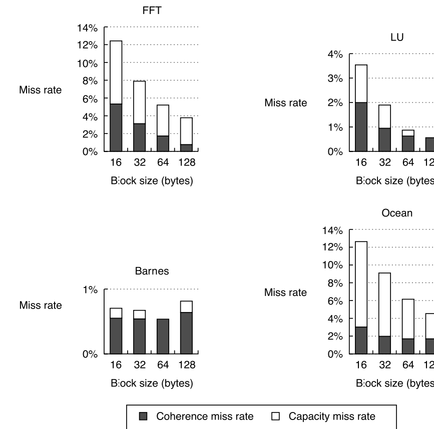


note scale difference

capacity miss rate improvement is consistent with what you'd expect but steady coherence miss rate (except Barnes) remains a performance anchor

© 2003 Elsevier Science (USA). All rights reserved.

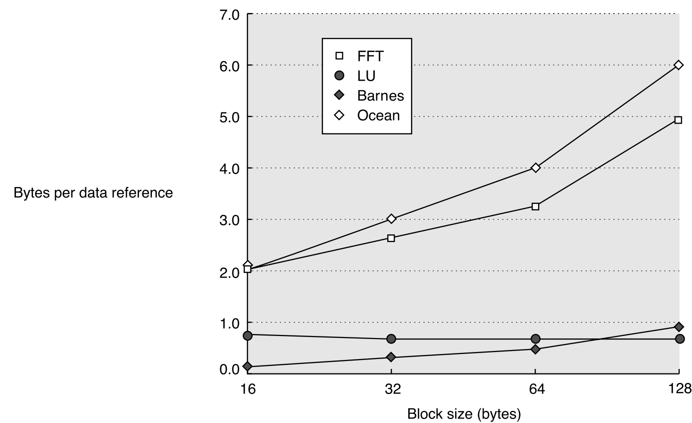
Miss Rate and Block Size



note false sharing effect on Barnes

© 2003 Elsevier Science (USA). All rights reserved.

Bus Traffic vs. Block Size



© 2003 Elsevier Science (USA). All rights reserved.

Snooping Caches

- Relatively simple step for the hardware
  - global atomicity point
  - sadly shared bus is a scalability problem at high clock rates
- Performance
  - similar to what you'd expect from uniprocessor cache experience
  - a few new wrinkles
    - coherence: true and false sharing
  - with lots of processors the shared bus ==> conflict point
    - PE idle time goes up
  - the shared memory costs are minimized here
    - primarily due to simplicity and symmetry
    - they will go up as we move to a distributed model