CS252
Graduate Computer Architecture

Lecture 13:
Multiprocessor 3: Measurements, Crosscutting Issues, Examples, Fallacies & Pitfalls

March 2, 2001

Prof. David A. Patterson

Computer Science 252

Spring 2001

Review

- Caches contain all information on state of cached memory blocks
- Snooping and Directory Protocols similar
- Bus makes snooping easier because of broadcast (snooping => Uniform Memory Access)
- Directory has extra data structure to keep track of state of all cache blocks
- Distributing directory
  => scalable shared address multiprocessor
  => Cache coherent, Non Uniform Memory Access (NUMA)

Parallel App: Commercial Workload

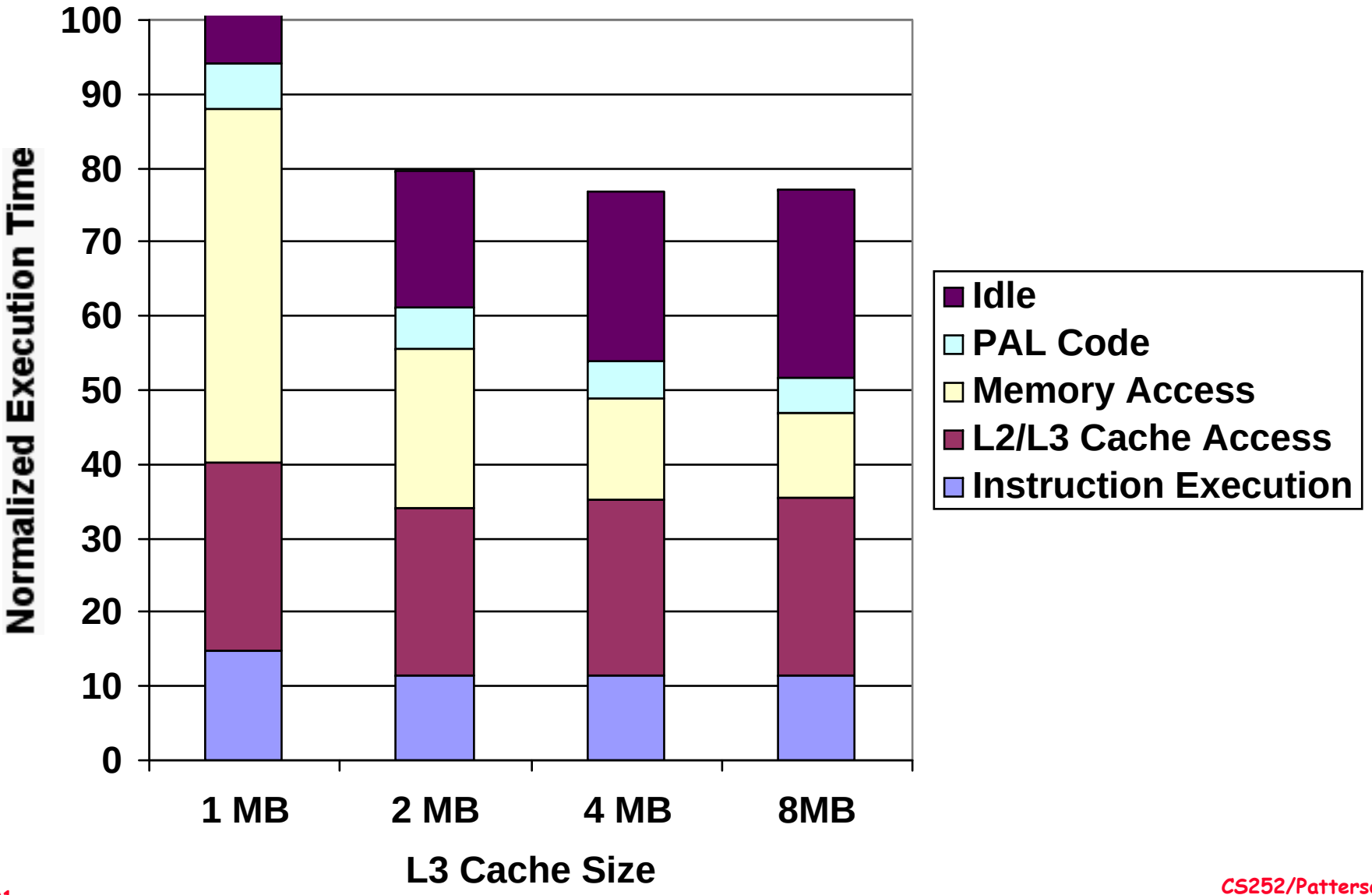
- Online transaction processing workload (OLTP) (like TPC-B or -C)
- Decision support system (DSS) (like TPC-D)
- Web index search (Altavista)

Benchmark	% Time User Mode	% Time Kernel	% Time I/O time (CPU Idle)
OLTP	71%	18%	11%
DSS (range)	82-94%	3-5%	4-13%
DSS (avg)	87%	4%	9%
Altavista	> 98%	< 1%	<1%

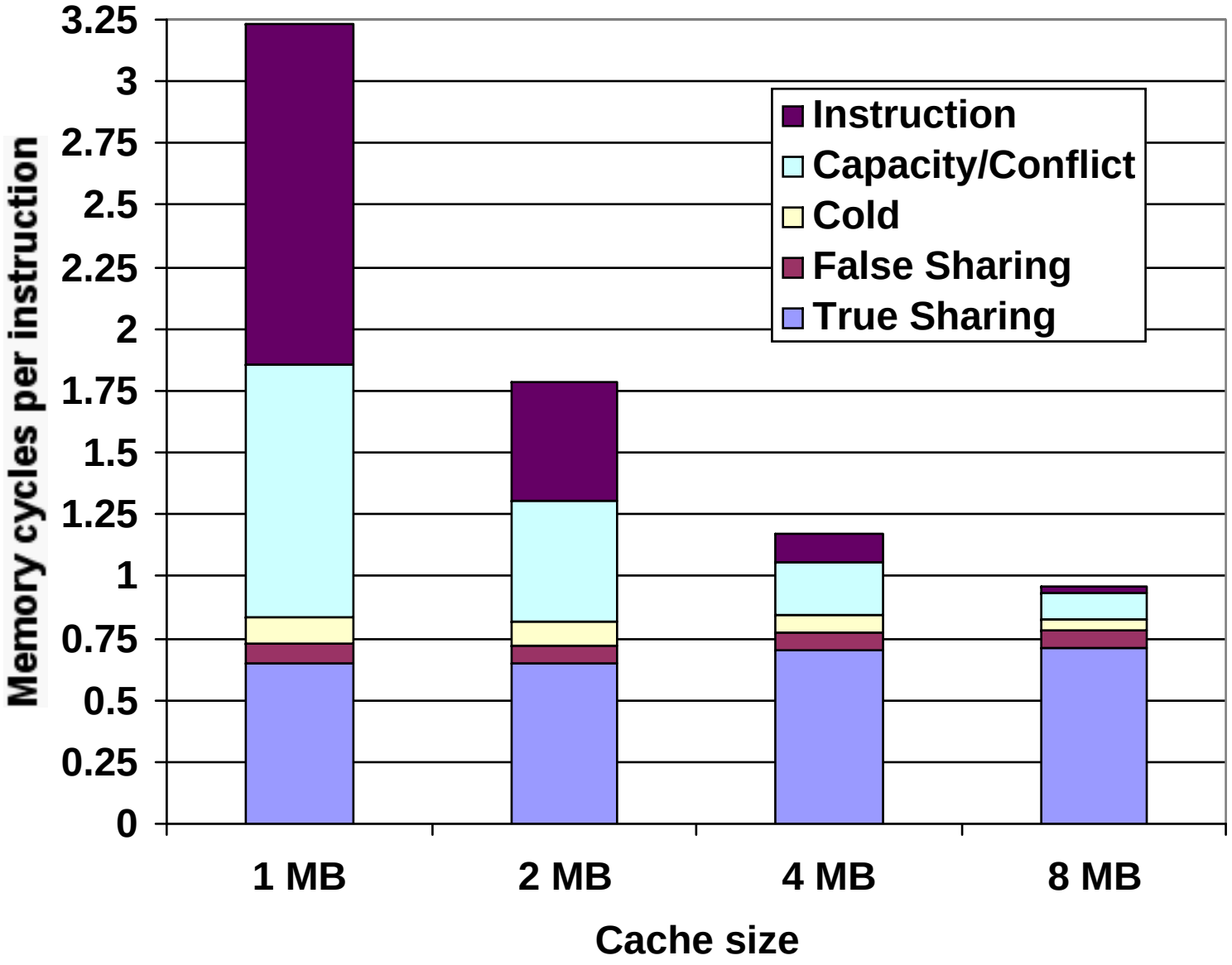
Alpha 4100 SMP

- 4 CPUs
- 300 MHz Apha 211264 @ 300 MHz
- L1$ 8KB direct mapped, write through
- L2$ 96KB, 3-way set associative
- L3$ 2MB (off chip), direct mapped
- Memory latency 80 clock cycles
- Cache to cache 125 clock cycles

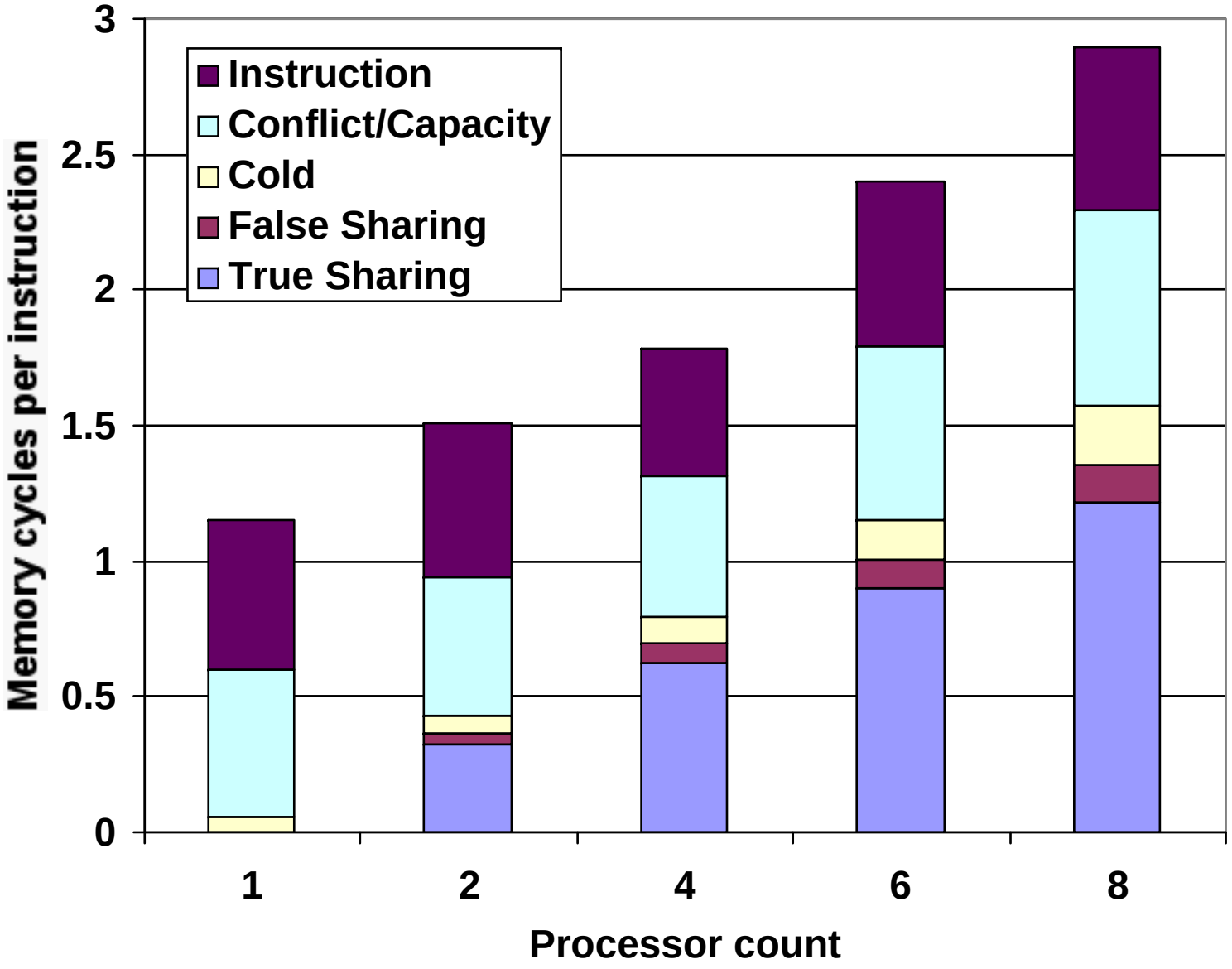
OLTP Performance as vary L3$ size



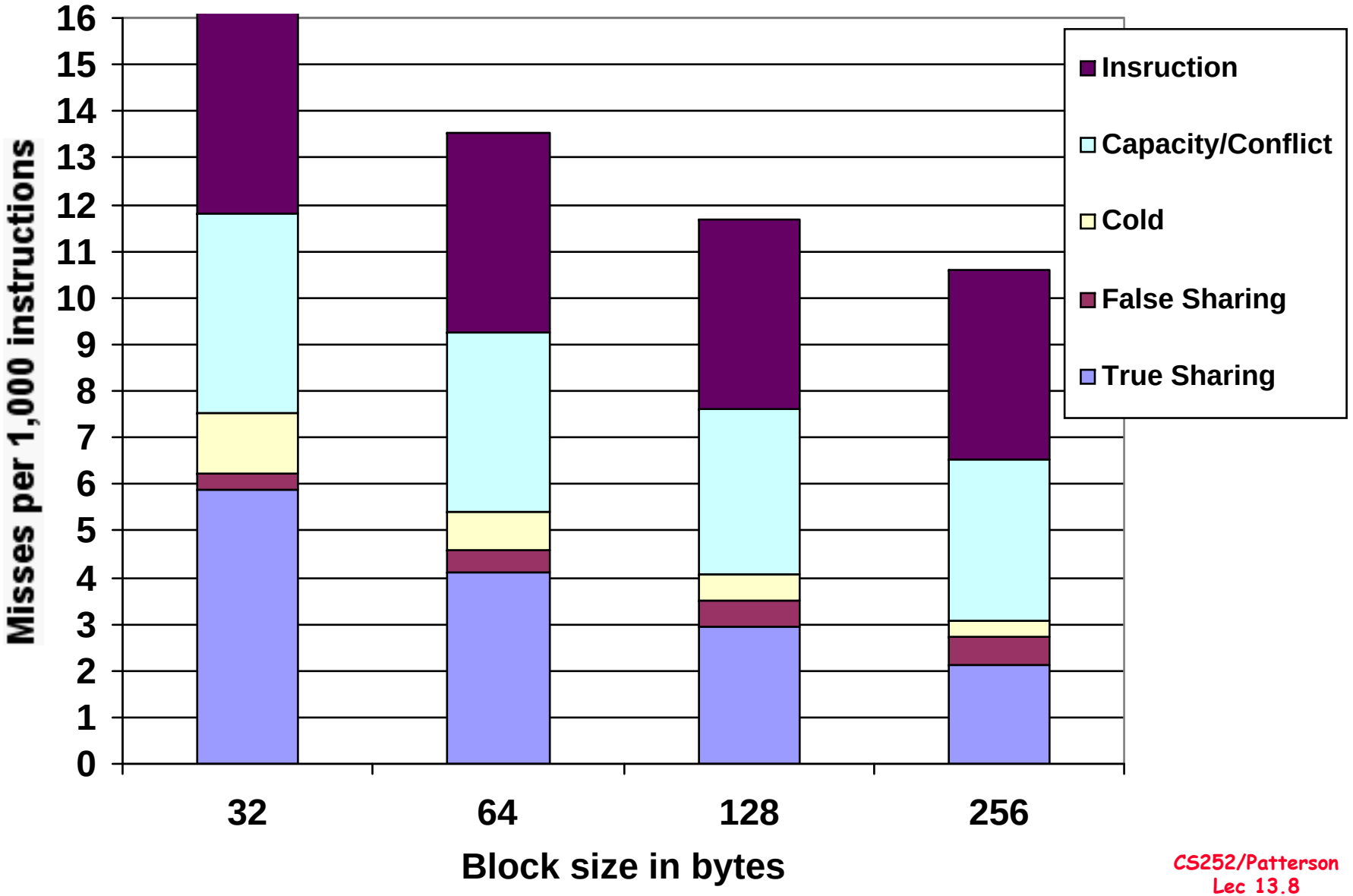
L3 Miss Breakdown



Memory CPI as increase CPUs



OLTP Performance as vary L3$ size



NUMA Memory performance for Scientific Apps on SGI Origin 2000

- Show average cycles per memory reference in 4 categories:
- Cache Hit
- Miss to local memory
- Remote miss to home
- 3-network hop miss to remote cache

CS 252 Administrivia

- Quiz #1 Wed March 7 5:30-8:30 306 Soda
  - No Lecture
- La Val's afterward quiz: free food and drink
- 3 questions
- Bring pencils
- Bring sheet of paper with notes on 2 sides
- Bring calculator (but don't store notes in calculator)

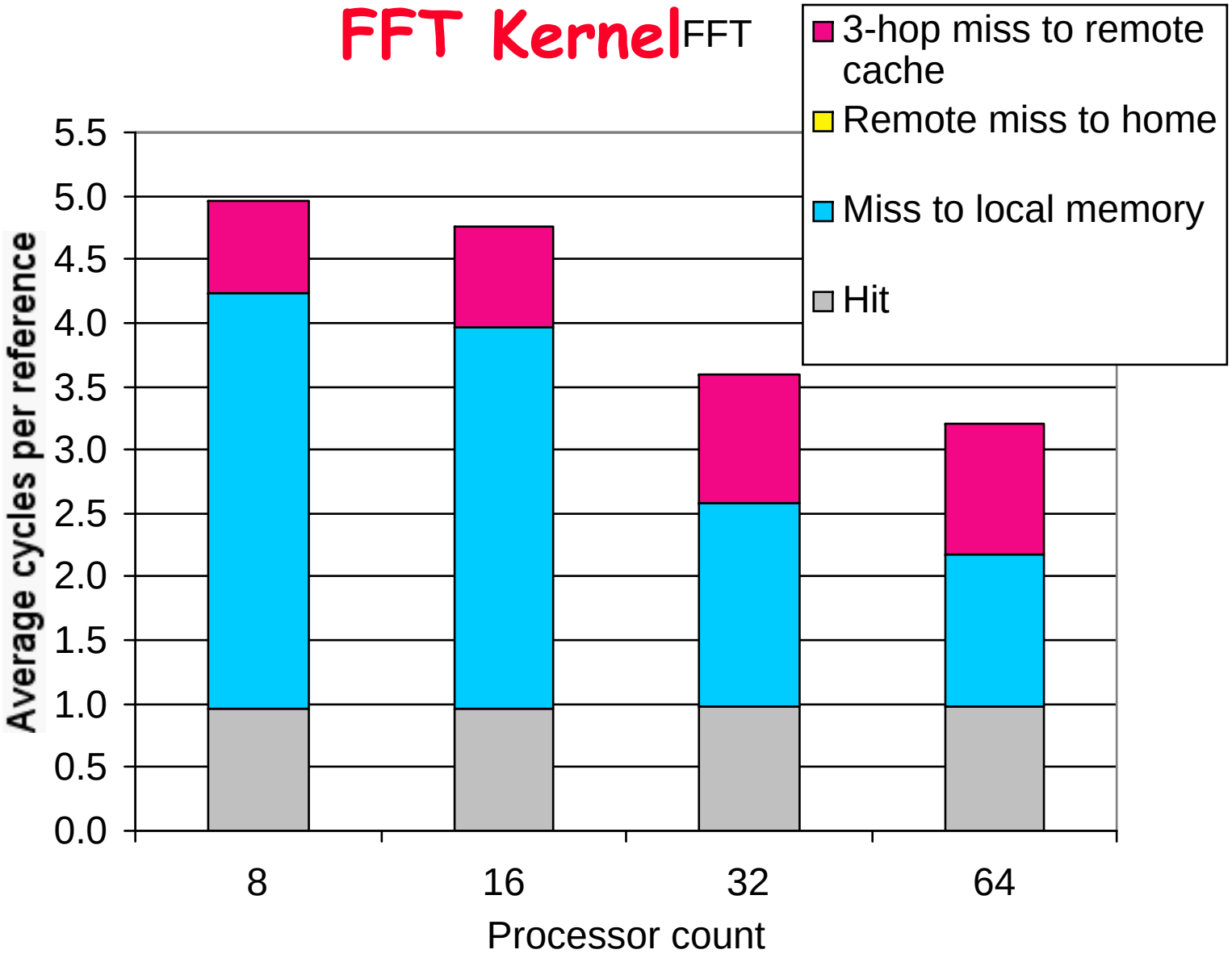
SGI Origin 2000

- a pure NUMA
- 2 CPUs per node,
- Scales up to 2048 processors
- Design for scientific computation vs. commercial processing
- Scalable bandwidth is crucial to Origin

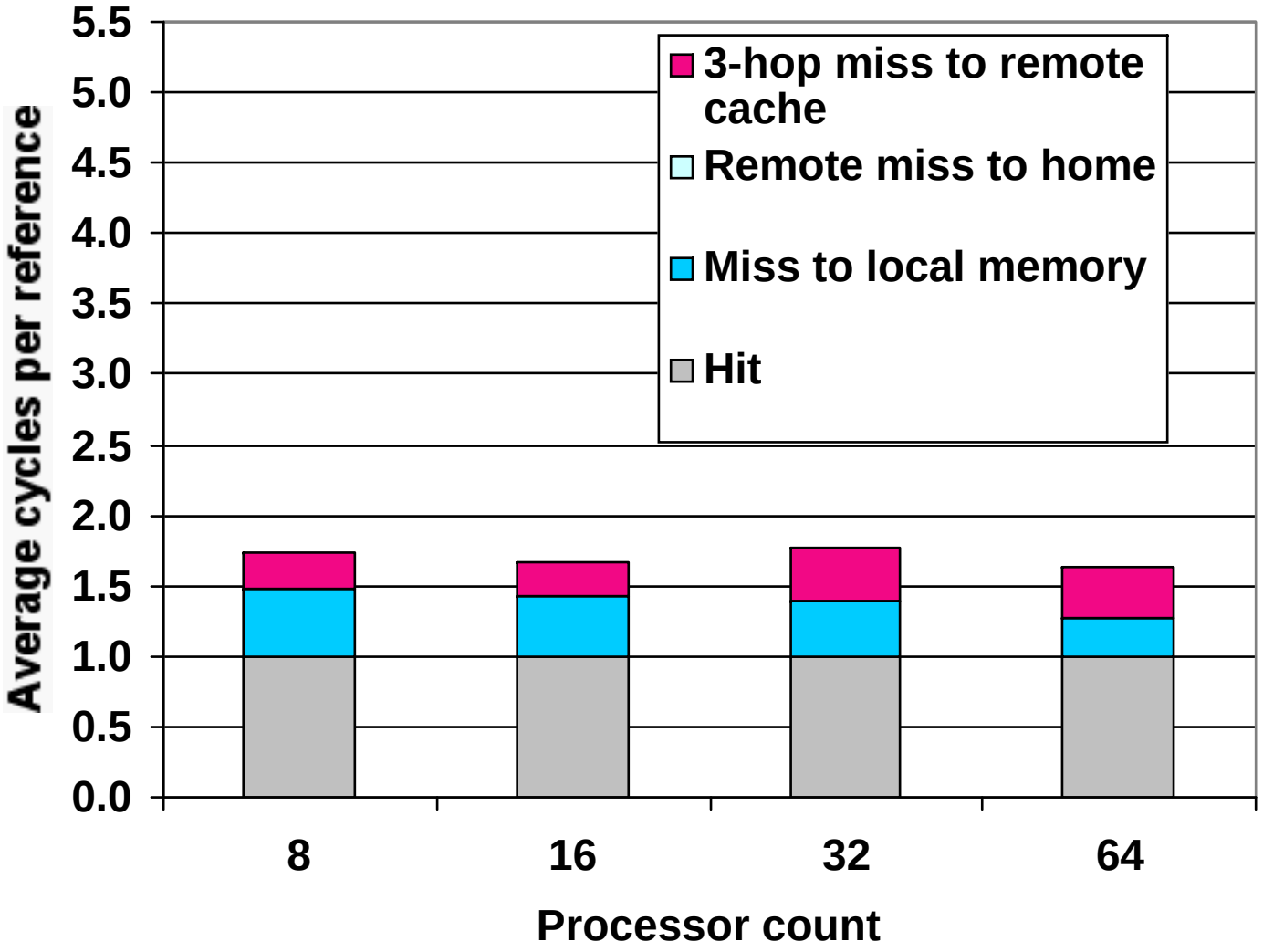
Parallel App: Scientific/Technical

- FFT Kernel: 1D complex number FFT
  - 2 matrix transpose phases => all-to-all communication
  - Sequential time for n data points: $O(n \log n)$
  - Example is 1 million point data set
- LU Kernel: dense matrix factorization
  - Blocking helps cache miss rate, 16x16
  - Sequential time for nxn matrix: $O(n^3)$
  - Example is 512 x 512 matrix

FFT Kernel FFT



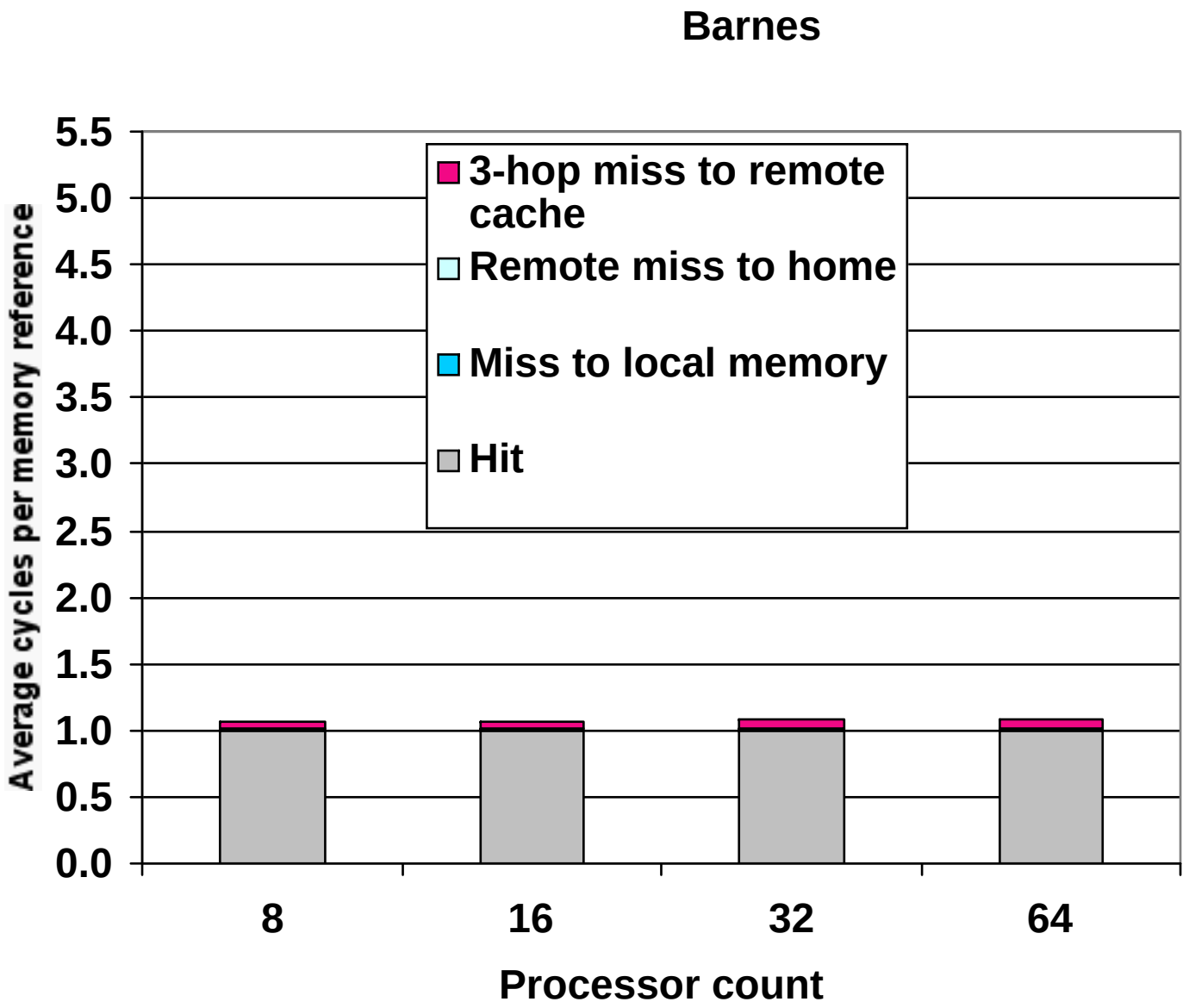
LU kernel LU


Average cycles per reference
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
3-hop miss to remote cache
Remote miss to home
Miss to local memory
Hit
8
16
32
64
Processor count

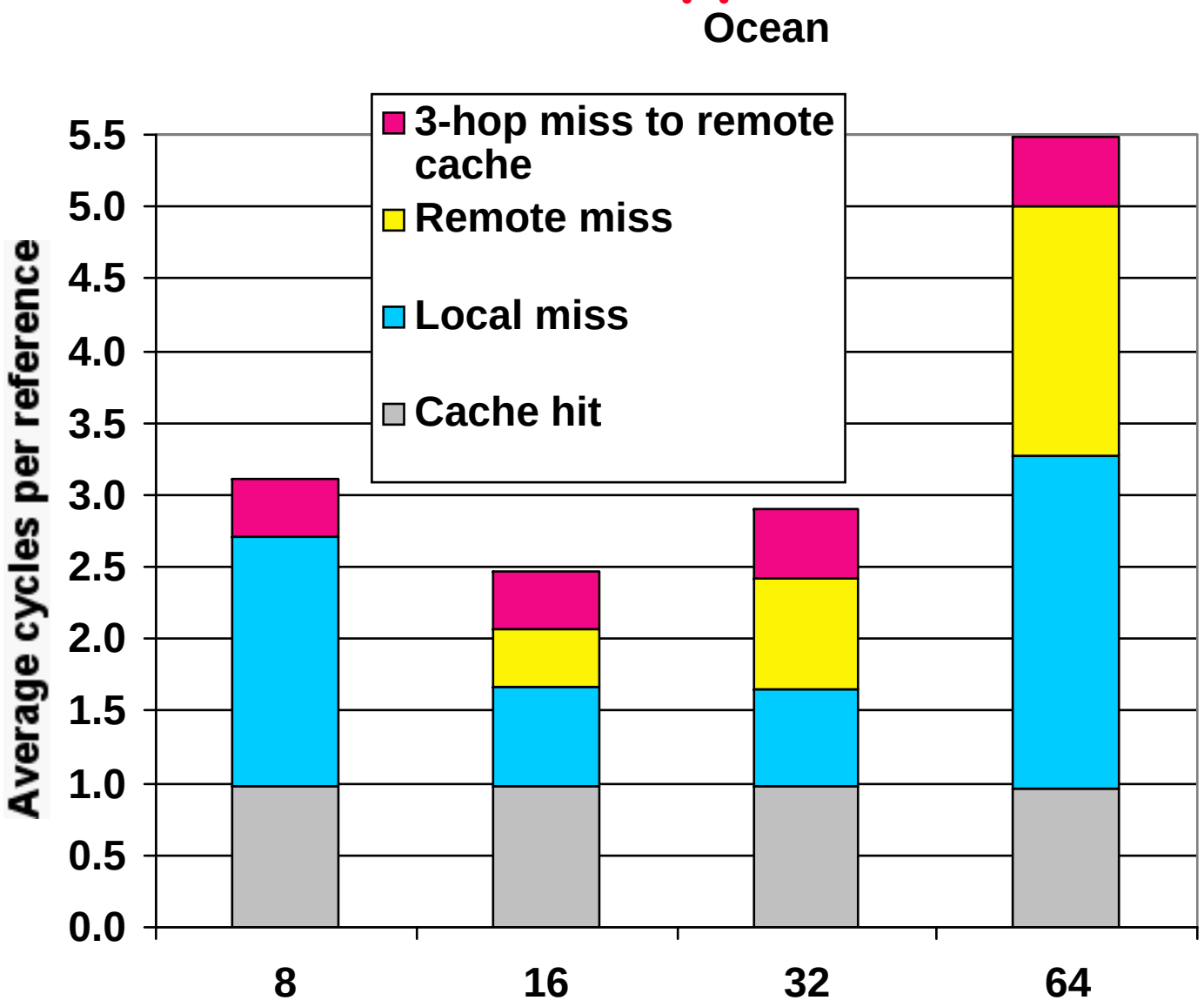
Parallel App: Scientific/Technical

- Barnes App: Barnes-Hut n-body algorithm solving a problem in galaxy evolution
  - n-body algs rely on forces drop off with distance; if far enough away, can ignore (e.g., gravity is $1/d^2$)
  - Sequential time for n data points: O(n log n)
  - Example is 16,384 bodies
- Ocean App: Gauss-Seidel multigrid technique to solve a set of elliptical partial differential eq.s'
  - red-black Gauss-Seidel colors points in grid to consistently update points based on previous values of adjacent neighbors
  - Multigrid solve finite diff. eq. by iteration using hierarch. Grid
  - Communication when boundary accessed by adjacent subgrid
  - Sequential time for nxn grid: $O(n^2)$
  - Input: 130 x 130 grid points, 5 iterations

Barnes App


Barnes
Average cycles per memory reference
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
3-hop miss to remote cache
Remote miss to home
Miss to local memory
Hit
8
16
32
64
Processor count

Ocean App


Ocean
3-hop miss to remote cache
Remote miss
Local miss
Cache hit
Average cycles per reference
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
8
16
32
64

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, => $10^{1/3}n$ => 2.15n scale up
  - Memory-constrained scaling: 10n size => $10^3/10$ => 100X or 100 hours! 10X processors for 100X longer???
  - Need to know application well to scale: # iterations, error tolerance

Cross Cutting Issues: Memory System Issues

- Multilevel cache hierarchy + multilevel inclusion—every level of cache hierarchy is a subset of next level—then can reduce contention between coherence traffic and processor traffic
  - Hard if cache blocks different sizes
- Also issues in memory consistency model and speculation, nonblocking caches, prefetching

Example: Sun Wildfire Prototype

- Connect 2-4 SMPs via optional NUMA technology
  - Use "off-the-self" SMPs as building block

1. For example, E6000 up to 15 processor or I/O boards (2 CPUs/board)
   - Gigaplane bus interconnect, 3.2 Gbytes/sec
- Wildfire Interface board (WFI) replace a CPU board => up to 112 processors (4 x 28),
  - WFI board supports one coherent address space across 4 SMPs
  - Each WFI has 3 ports connect to up to 3 additional nodes, each with a dual directional 800 MB/sec connection
  - Has a directory cache in WFI interface: local or clean OK, otherwise sent to home node
  - Multiple bus transactions

Example: Sun Wildfire Prototype

- To reduce contention for page, has Coherent Memory Replication (CMR)
- Page-level mechanisms for migrating and replicating pages in memory, coherence is still maintained at the cache-block level
- Page counters record misses to remote pages and to migrate/replicate pages with high count
- Migrate when a page is primarily used by a node
- Replicate when multiple nodes share a page

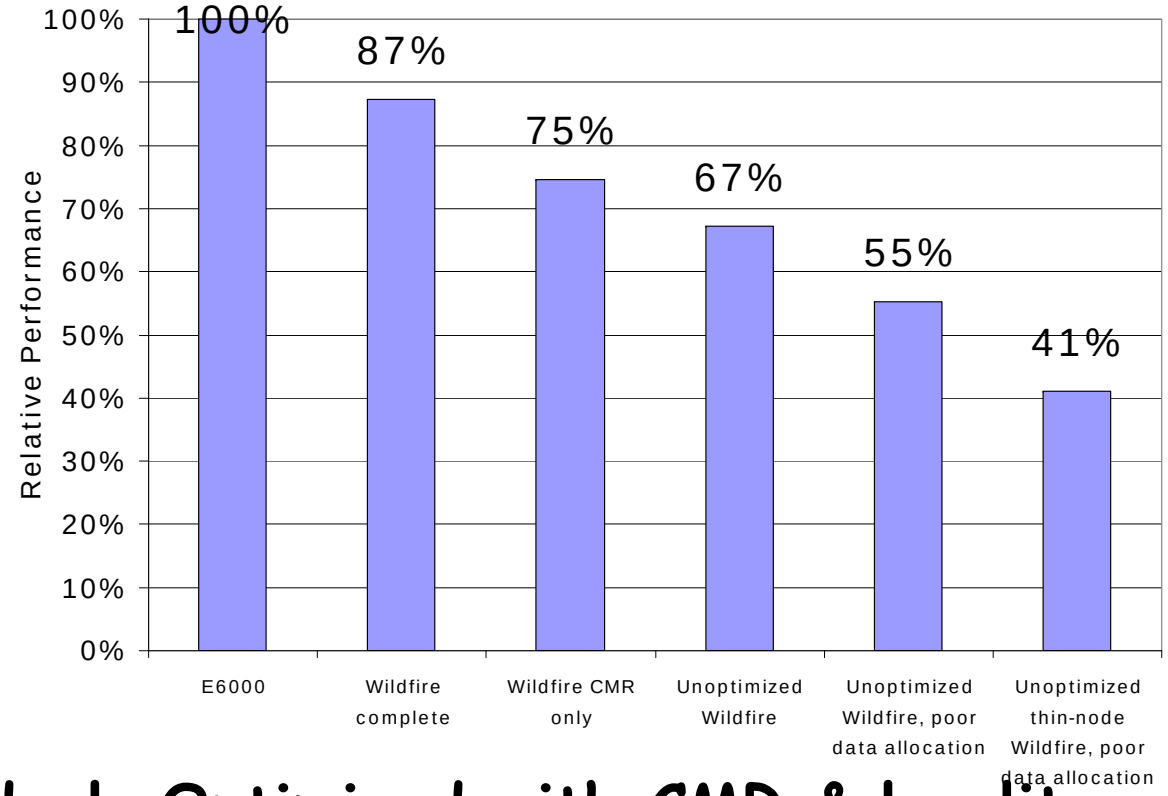
Memory Latency Wildfire v. Origin (nanoseconds)

Case	How?	Target?	Wildfire	Origin
Local mem.	Restart	Unowned	342	338
Local mem.	Restart	Dirty	482	892
Local mem.	Back-to-back	Dirty	470	1036
Avg. remote mem. (<128)	Restart	Unowned	1774	973
Avg. remote mem. (< 128)	Restart	Dirty	2162	1531
Avg. all mem. (< 128)	Restart	Unowned	1416	963
Avg. all mem. (< 128)	Restart	Dirty	1742	1520

Memory Bandwidth Wildfire v. Origin (Mbytes/second)

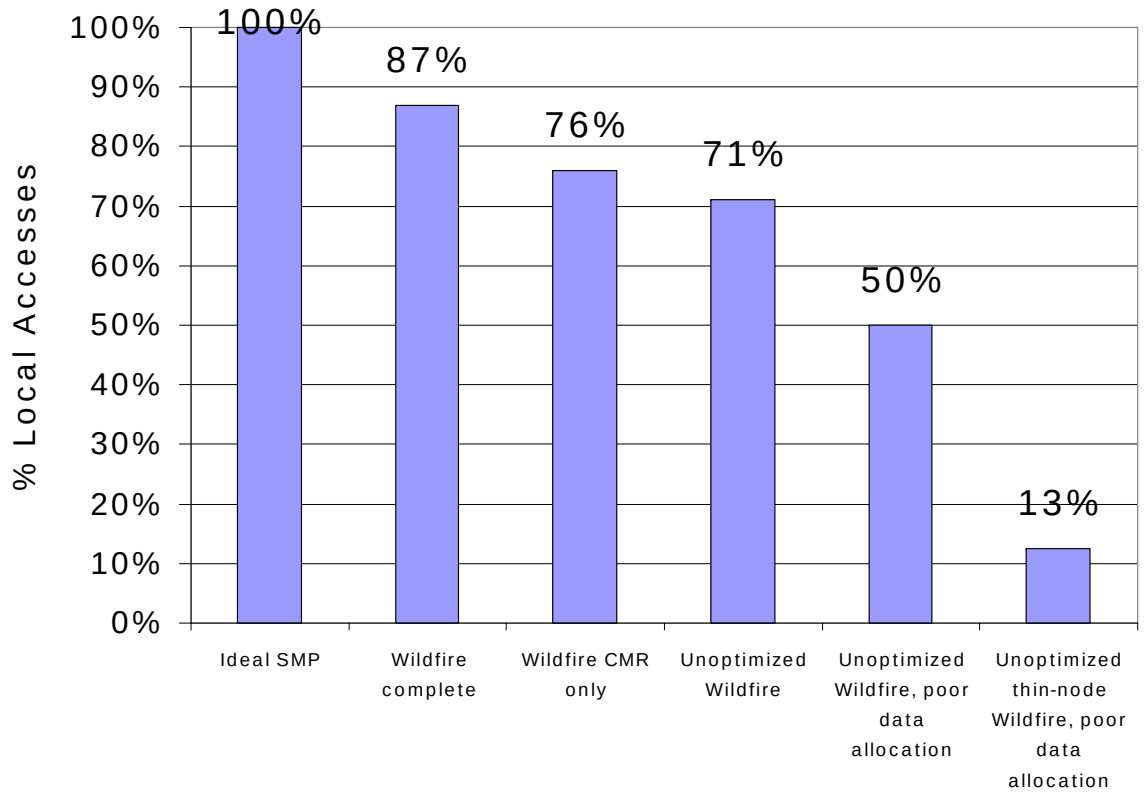
Characteristic	Wildfire	Origin
Pipelined local mem BW: unowned	312	554
Pipelined local mem BW: exclusive	266	340
Pipelined local mem BW: dirty	246	182
Total local mem BW (per node)	2,700	631
Local mem BW per proc	96	315
Aggregate local mem BW (all nodes, 112 proc)	10,800	39,088
Total bisection BW	9,600	25,600
Bisection BW per processor (112 proc)	86	229

E6000 v. Wildfire variations: OLTP Performance for 16 procs



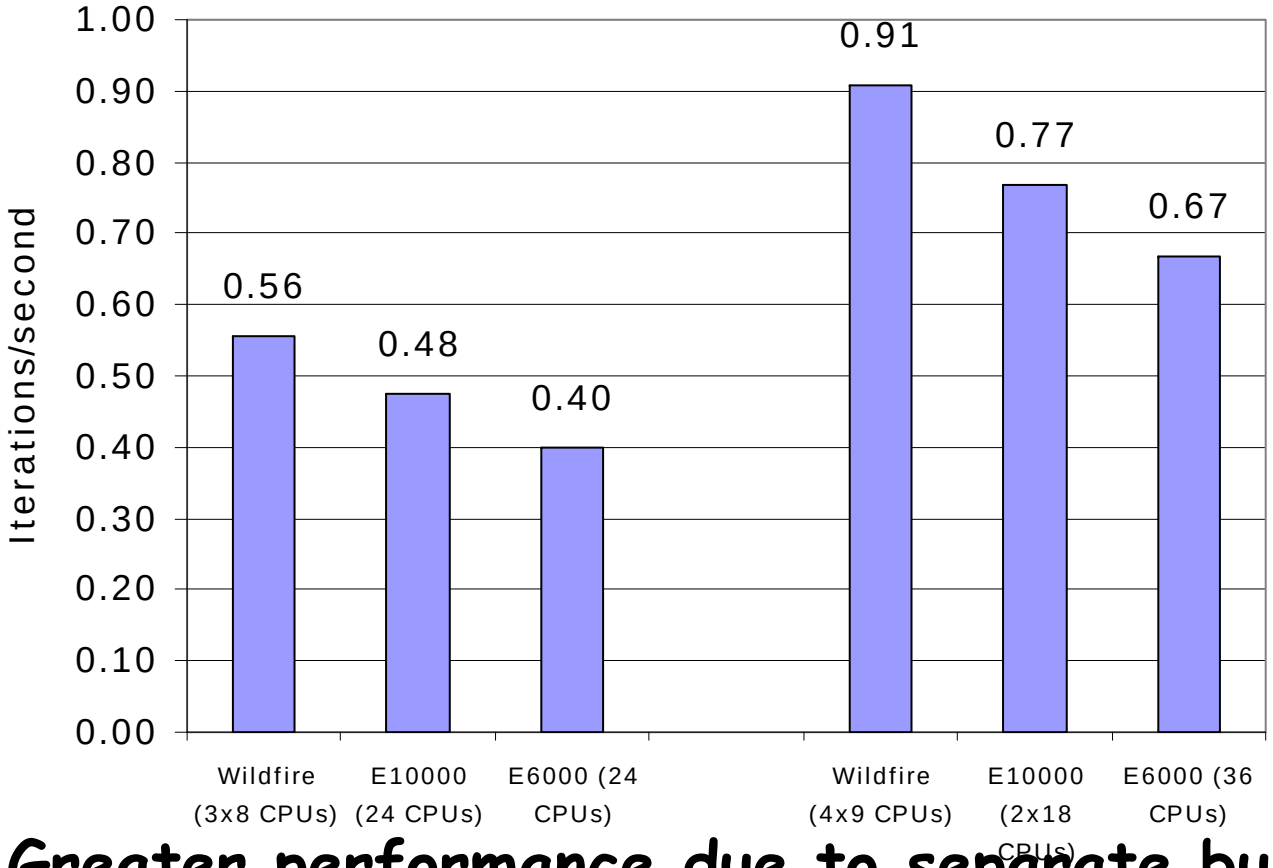
- Ideal, Optimized with CMR & locality scheduling, CMR only, unoptimized, poor data placement, thin nodes (2 v. 8 / node)

E6000 v. Wildfire variations: % local memory access (within node)



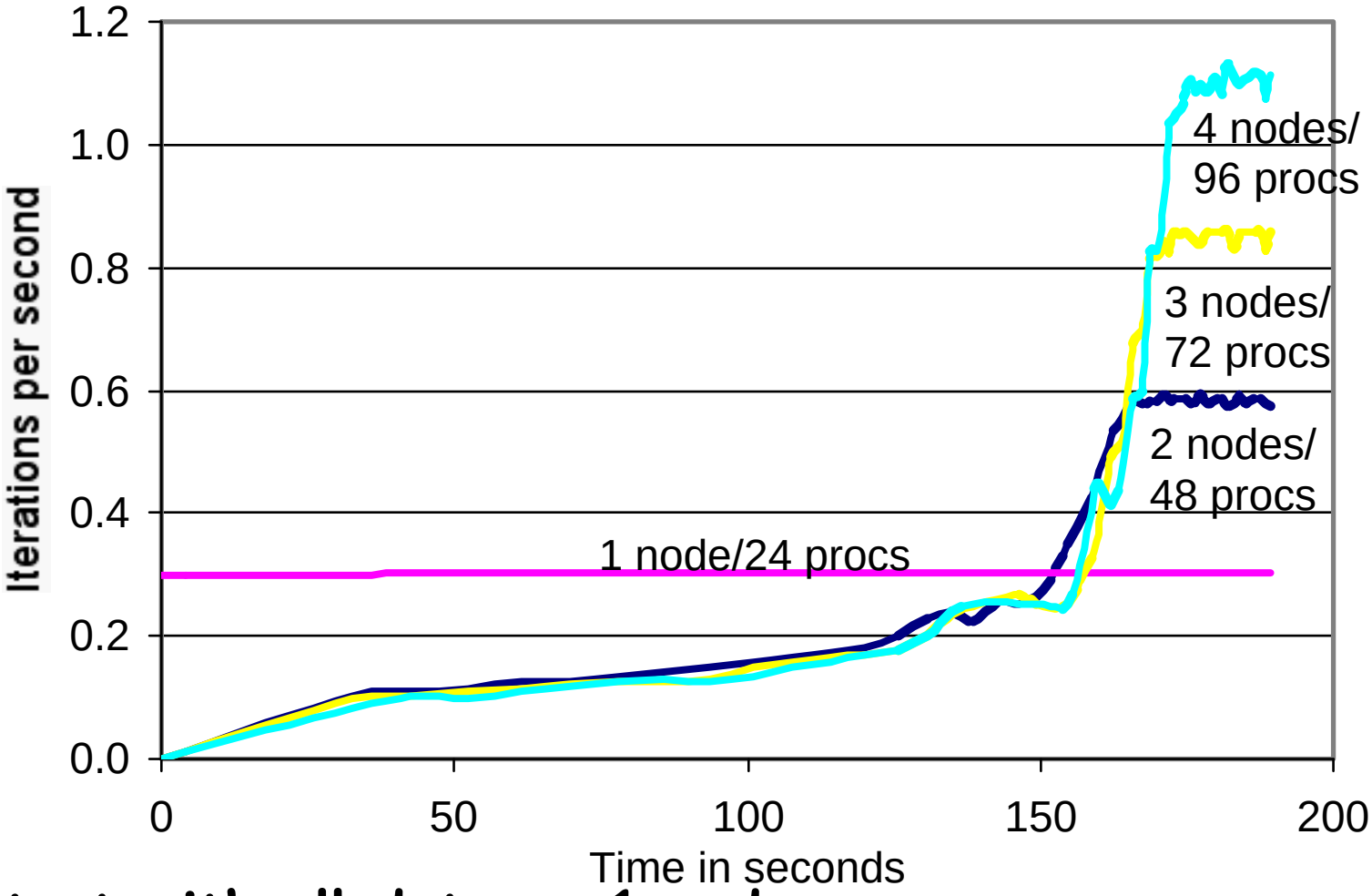
- Ideal, Optimized with CMR & locality scheduling, CMR only, unoptimized, poor data placement, thin nodes (2 v. 8 / node)

E10000 v. E6000 v. Wildfire: Red_Black Solver 24 and 36 procs



- Greater performance due to separate busses?
  - 24 proc E6000 bus utilization 90%-100%
  - 36 proc E10000 more caches => 1.7X perf v. 1.5X procs

Wildfire CMR: Red_Black Solver



- Start with all data on 1 node:
  500 iterations to converge (120-180 secs);
  what if memory allocation varied over time?

Wildfire CMR benefit: Migration vs. Replication Red_Black Solver

Policy	Iterations per sec	Iterations needed to reach stability	Number Migrations	Number Replications
No migration or replication	0.10	0	0	0
Migration only	1.06	154 sec.	992,51	
Replication only	1.15	61 sec.		98545
Migration + replication	1.09	151 sec.	98,543	85

- Migration only has much lower HW costs (no reverse memory maps)

Wildfire Remarks

- Fat nodes (8-24 way SMP) vs. Thin nodes (2-to 4-way like Origin)
- Market shift from scientific apps to database and web apps may favor a fat-node design with 8 to 16 CPUs per node
  - Scalability up to 100 CPUs may be of interest, but "sweet spot" of server market is 10s of CPUs. No customer interest in 1000 CPU machines key part of supercomputer marketplace
  - Memory access patterns of commercial apps have less sharing + less predictable sharing and data access
    => matches fat node design which have lower bisection BW per CPU than a thin-node design
    => as fat-node design less dependence on exact memory allocation an data placement, perform better for apps with irregular or changing data access patterns
    => fat-nodes make it easier for migration and replication

Embedded Multiprocessors

- EmpowerTel MXP, for Voice over IP
  - 4 MIPS processors, each with 12 to 24 KB of cache
  - 13.5 million transistors, 133 MHz
  - PCI master/slave + 100 Mbit Ethernet pipe
- Embedded Multiprocessing more popular in future as apps demand more performance
  - No binary compatability; SW written from scratch
  - Apps often have natural parallelism: set-top box, a network switch, or a game system
  - Greater sensitivity to die cost (and hence efficient use of silicon)

Pitfall: Measuring MP performance by linear speedup v. execution time

- "linear speedup" graph of perf as scale CPUs
- Compare best algorithm on each computer
- Relative speedup - run same program on MP and uniprocessor
  - But parallel program may be slower on a uniprocessor than a sequential version
  - Or developing a parallel program will sometimes lead to algorithmic improvements, which should also benefit uni
- True speedup - run best program on each machine
- Can get superlinear speedup due to larger effective cache with more CPUs

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

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 => 2.5X
- 16 proc => need only 3.6X, or 25% linear speedup
- Even if need some more memory for MP, not linear

Fallacy: Multiprocessors are "free."

- "Since microprocessors contain support for snooping caches, can build small-scale, bus-based multiprocessors for no additional cost"
- Need more complex memory controller (coherence) than for uniprocessor
- Memory access time always longer with more complex controller
- Additional software effort: compilers, operating systems, and debuggers all must be adapted for a parallel system

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 2,048 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 appls that scale requires significantly more attention to load balance, locality, potential contention, and serial (or partly parallel) portions of program. 10X is very hard

Pitfall: Not developing the software to take advantage of, or optimize for, a 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 initilization (even if separate processes)

Multiprocessor Conclusion

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

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?
```

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