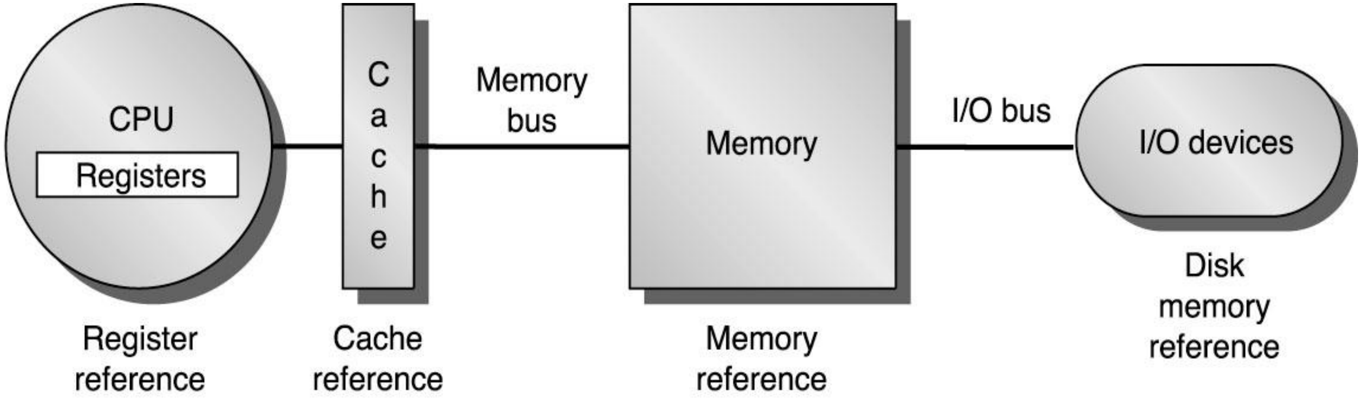
Chapter 2
Memory-Hierarchy Design

Multilevel Memory Hierarchy

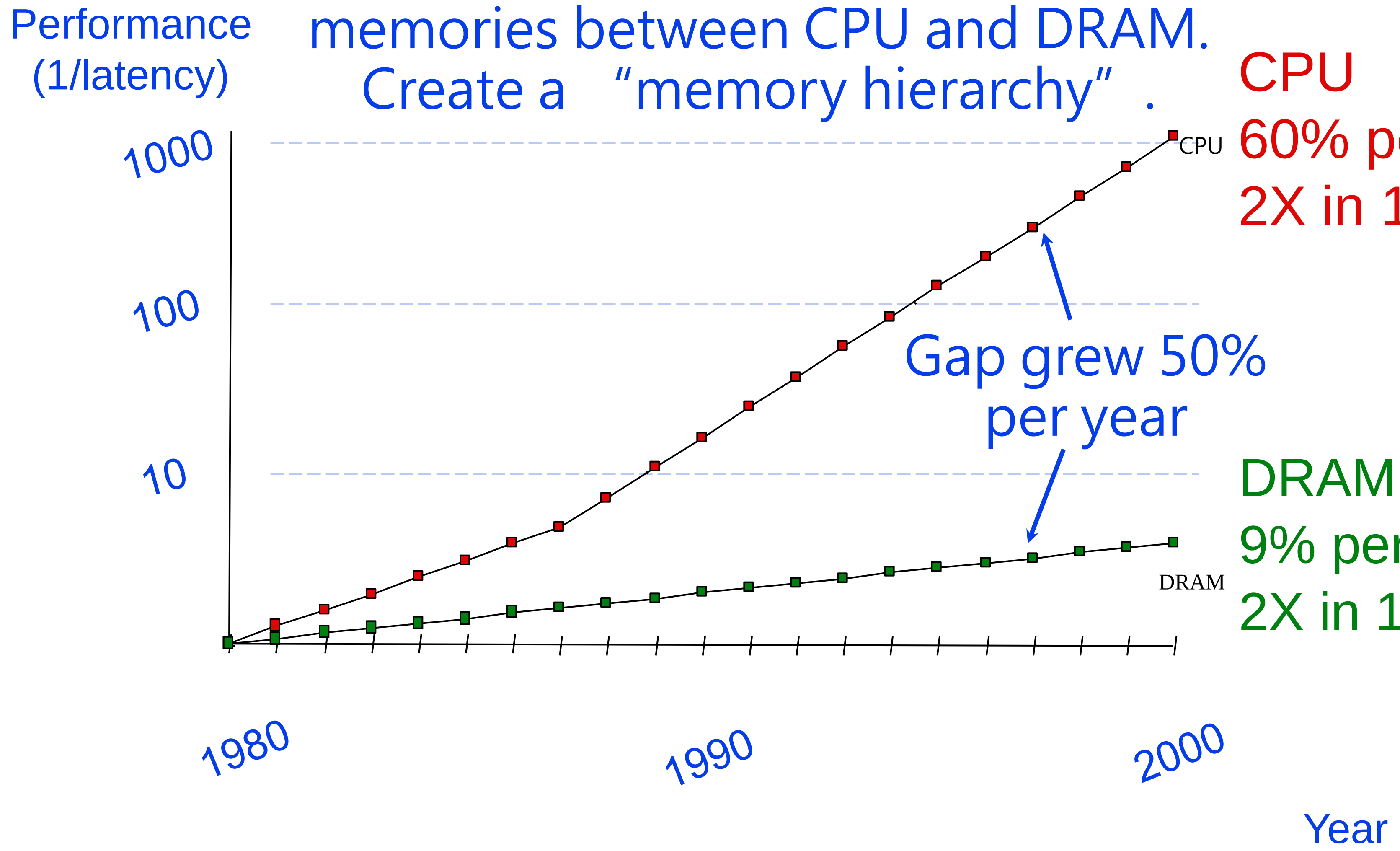


Size:	500 bytes	64 KB	512 MB	100 GB
Speed:	0.25 ns	1 ns	100 ns	5 ms

Since 1980, CPU has outpaced DRAM ...

Q. How do architects address this gap?

A. Put smaller, faster “cache” memories between CPU and DRAM. Create a “memory hierarchy”.



Gap grew 50% per year

CPU
60% per yr
2X in 1.5 yrs

DRAM
9% per yr
2X in 10 yrs

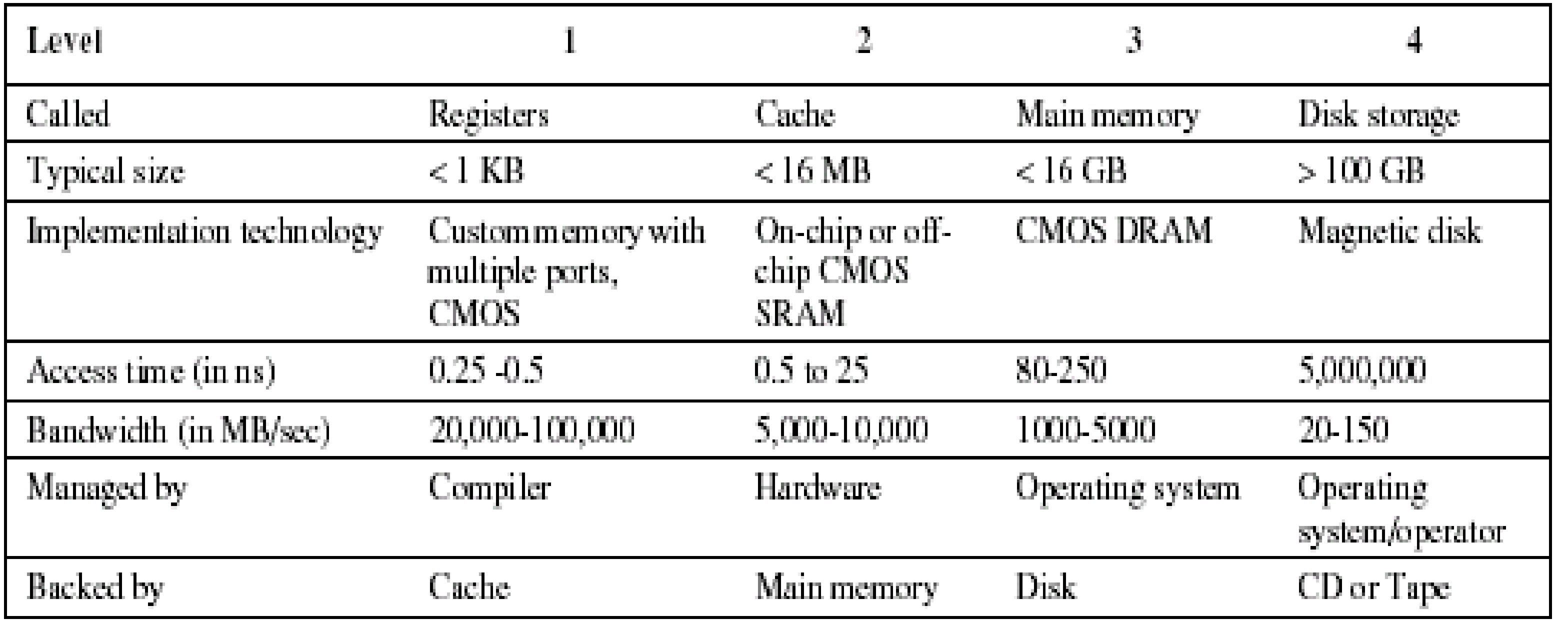
Memory Hierarchy of Embedded Computers vs Desktop's

- Embedded computers are often used in real-time applications, and hence programmers must worry about worst case performance
- Embedded applications are often concerned about power and battery life
- The protection role of the memory hierarchy is often diminished
- The main memory itself may be quite small–less than one megabyte–and there is often no disk storage

Locality

- Temporal locality:
  - It’s likely to need this word again in the near future
- Spatial locality:
  - There is high probability that the other data in the block will be needed soon

Cache Performance Review



Level	1	2	3	4
Called	Registers	Cache	Main memory	Disk storage
Typical size	< 1 KB	< 16 MB	< 16 GB	> 100 GB
Implementation technology	Custom memory with multiple ports, CMOS	On-chip or off-chip CMOS SRAM	CMOS DRAM	Magnetic disk
Access time (in ns)	0.25 -0.5	0.5 to 25	80-250	5,000,000
Bandwidth (in MB/sec)	20,000-100,000	5,000-10,000	1000-5000	20-150
Managed by	Compiler	Hardware	Operating system	Operating system/operator
Backed by	Cache	Main memory	Disk	CD or Tape

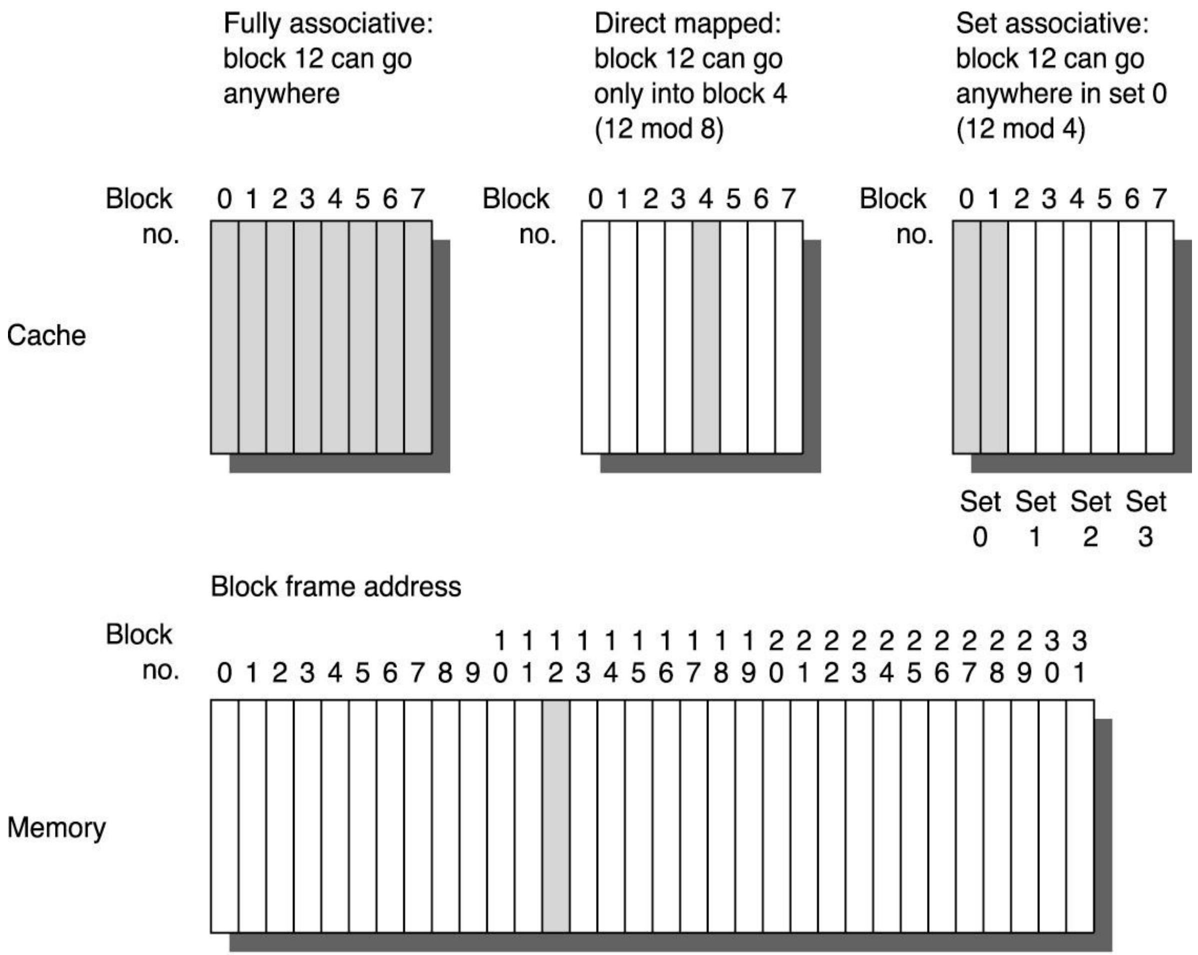
Four Memory Hierarchy Questions

- Q1: Where can a block be placed in the upper level? (Block placement)
- Q2: How is a block found if it is in the upper level? (Block identification)
- Q3: Which block should be replaced on a miss? (Block replacement)
- Q4: What happens on a write? (Write strategy)

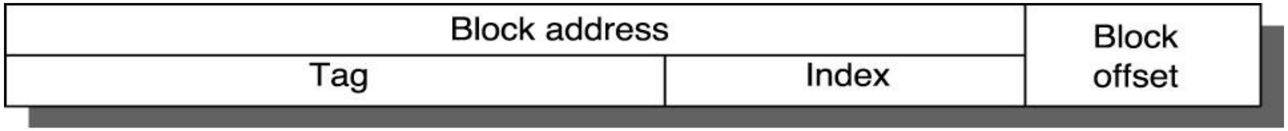
Q1: Where Can a Block Be Placed in a Cache?

- Direct mapped
- Fully associative
- Set associative

Example


Fully associative:
block 12 can go
anywhere
Direct mapped:
block 12 can go
only into block 4
(12 mod 8)
Set associative:
block 12 can go
anywhere in set 0
(12 mod 4)
Block no.
0 1 2 3 4 5 6 7
Cache
Set 0
Set 1
Set 2
Set 3
Block frame address
Block no.
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Memory

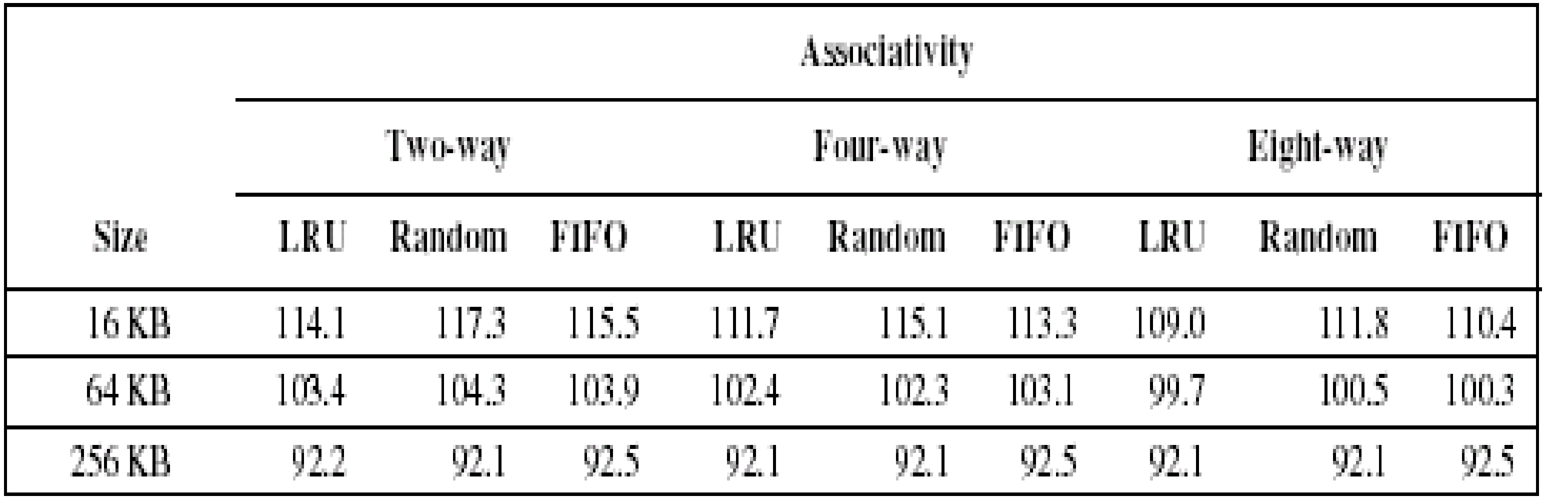
Q2: How Is a Block Found If It Is in the Cache?



Q3: Which Block Should Be Replaced on a Cache Miss?

- Random
- LRU
- FIFO

Miss Rates between LRU, Random, and FIFO replacement



	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

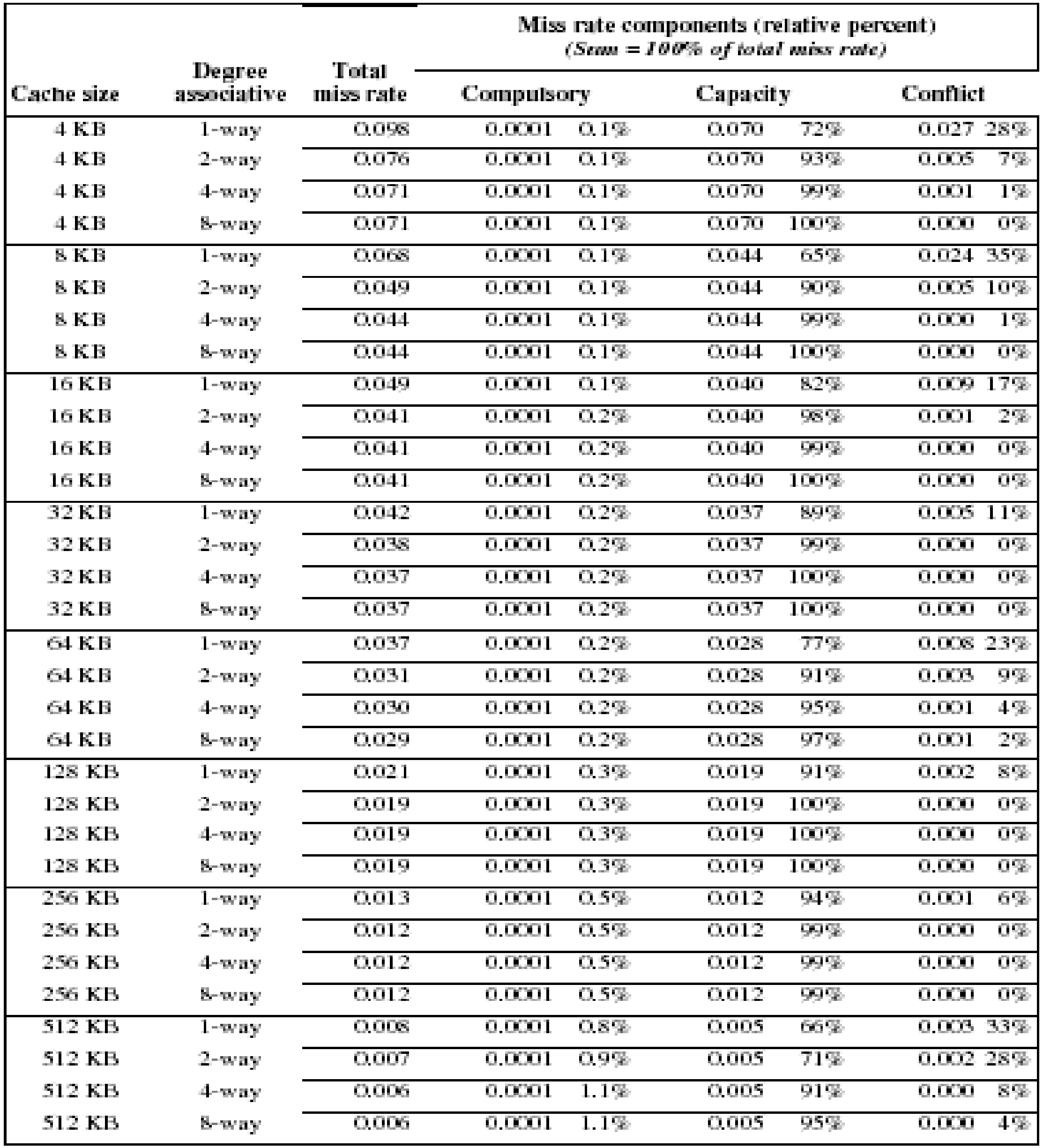
Q4: What Happens on a Write? (1/2)

- There are two basic options when writing to the cache:
  - Write through
  - Write back
- Dirty bit
- Write stall
- Write buffer

Q4: What Happens on a Write? (2/2)

- Since the data are not needed on a write, there are two are two options on a write miss:
  - —Write allocate: the block is allocated on a write miss, followed by the write it actions above.
  - —No-write allocate: blocks stays out of the cache in no-write allocate until the program tries to read the blocks

Frequency of Cache Misses

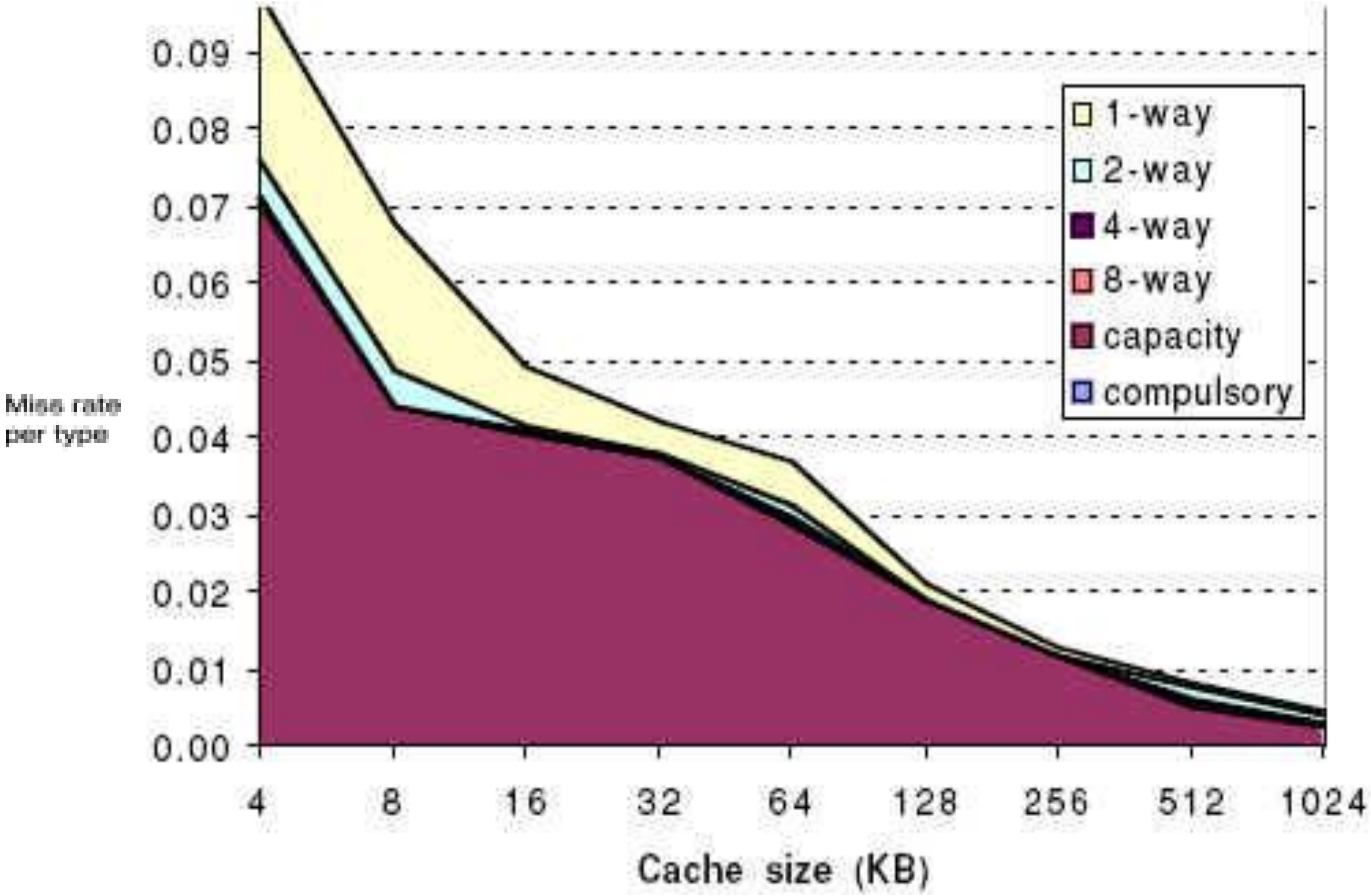


Cache size	Degree associative	Total miss rate	Miss rate components (relative percent) (*Sum = 100% of total miss rate*)					
			Compulsory		Capacity		Conflict	
4 KB	1-way	0.098	0.0001	0.1%	0.070	72%	0.027	28%
4 KB	2-way	0.076	0.0001	0.1%	0.070	93%	0.005	7%
4 KB	4-way	0.071	0.0001	0.1%	0.070	99%	0.001	1%
4 KB	8-way	0.071	0.0001	0.1%	0.070	100%	0.000	0%
8 KB	1-way	0.068	0.0001	0.1%	0.044	65%	0.024	35%
8 KB	2-way	0.049	0.0001	0.1%	0.044	90%	0.005	10%
8 KB	4-way	0.044	0.0001	0.1%	0.044	99%	0.000	1%
8 KB	8-way	0.044	0.0001	0.1%	0.044	100%	0.000	0%
16 KB	1-way	0.049	0.0001	0.1%	0.040	82%	0.009	17%
16 KB	2-way	0.041	0.0001	0.2%	0.040	98%	0.001	2%
16 KB	4-way	0.041	0.0001	0.2%	0.040	99%	0.000	0%
16 KB	8-way	0.041	0.0001	0.2%	0.040	100%	0.000	0%
32 KB	1-way	0.042	0.0001	0.2%	0.037	89%	0.005	11%
32 KB	2-way	0.038	0.0001	0.2%	0.037	99%	0.000	0%
32 KB	4-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
32 KB	8-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
64 KB	1-way	0.037	0.0001	0.2%	0.028	77%	0.008	23%
64 KB	2-way	0.031	0.0001	0.2%	0.028	91%	0.003	9%
64 KB	4-way	0.030	0.0001	0.2%	0.028	95%	0.001	4%
64 KB	8-way	0.029	0.0001	0.2%	0.028	97%	0.001	2%
128 KB	1-way	0.021	0.0001	0.3%	0.019	91%	0.002	8%
128 KB	2-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128 KB	4-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128 KB	8-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
256 KB	1-way	0.013	0.0001	0.5%	0.012	94%	0.001	6%
256 KB	2-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256 KB	4-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256 KB	8-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
512 KB	1-way	0.008	0.0001	0.8%	0.005	66%	0.003	33%
512 KB	2-way	0.007	0.0001	0.9%	0.005	71%	0.002	28%
512 KB	4-way	0.006	0.0001	1.1%	0.005	91%	0.000	8%
512 KB	8-way	0.006	0.0001	1.1%	0.005	95%	0.000	4%

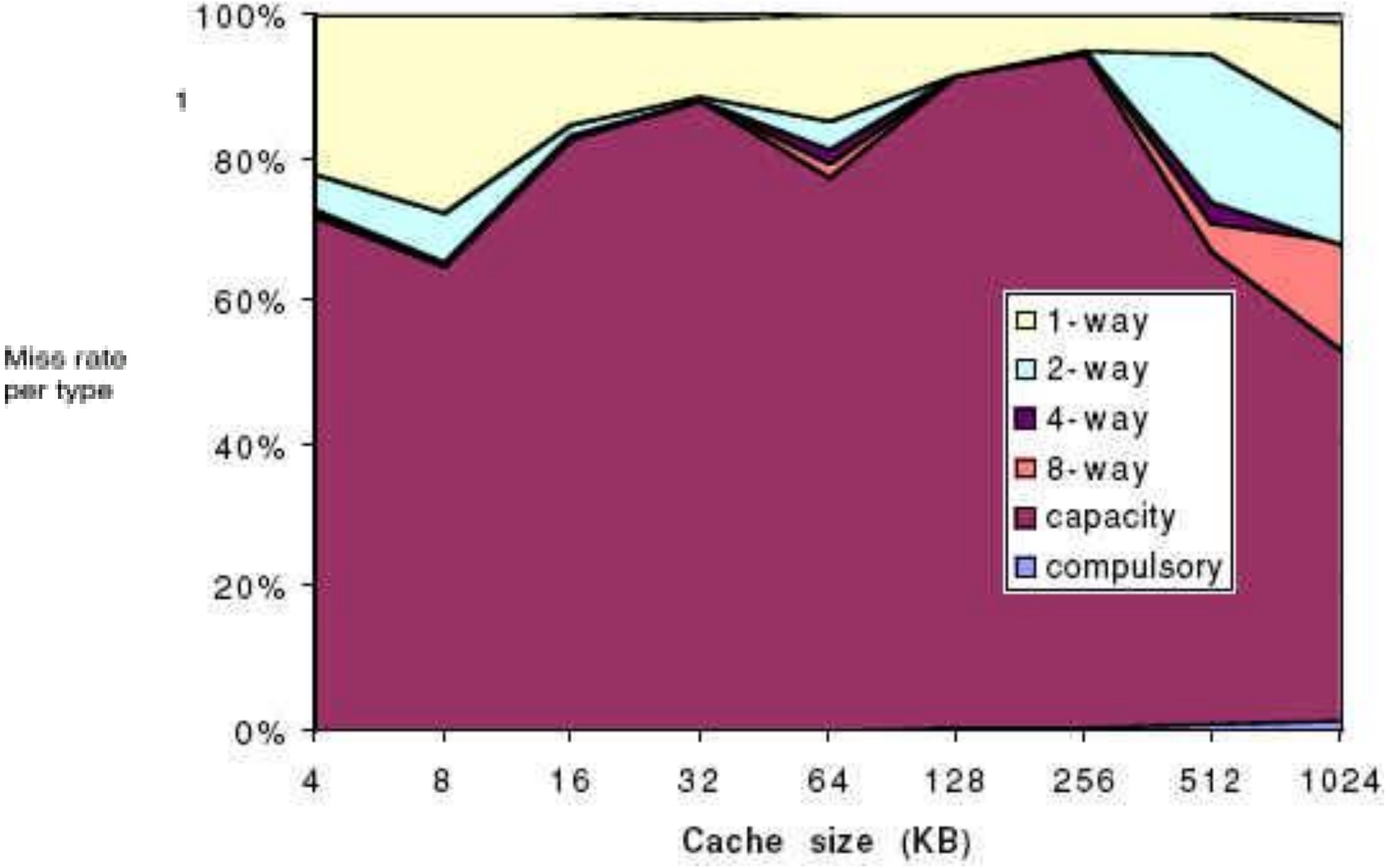
Divisions of Misses

- Compulsory misses are those that occur in an infinite cache
- Capacity misses are those that occur in a fully associative cache
- Conflict misses
  - —Eight-way—conflict misses due to going from fully associative to eight-way associative
  - —Four-way—conflict misses due to going from eight-way associative to four-way associative
  - —Two-way—conflict misses due to going from four-way associative to two-way associative
  - —One-way—conflict misses due to going from two-way associative to one-way associative (direct mapped)

Total Miss Rate



Distribution of Miss Rate



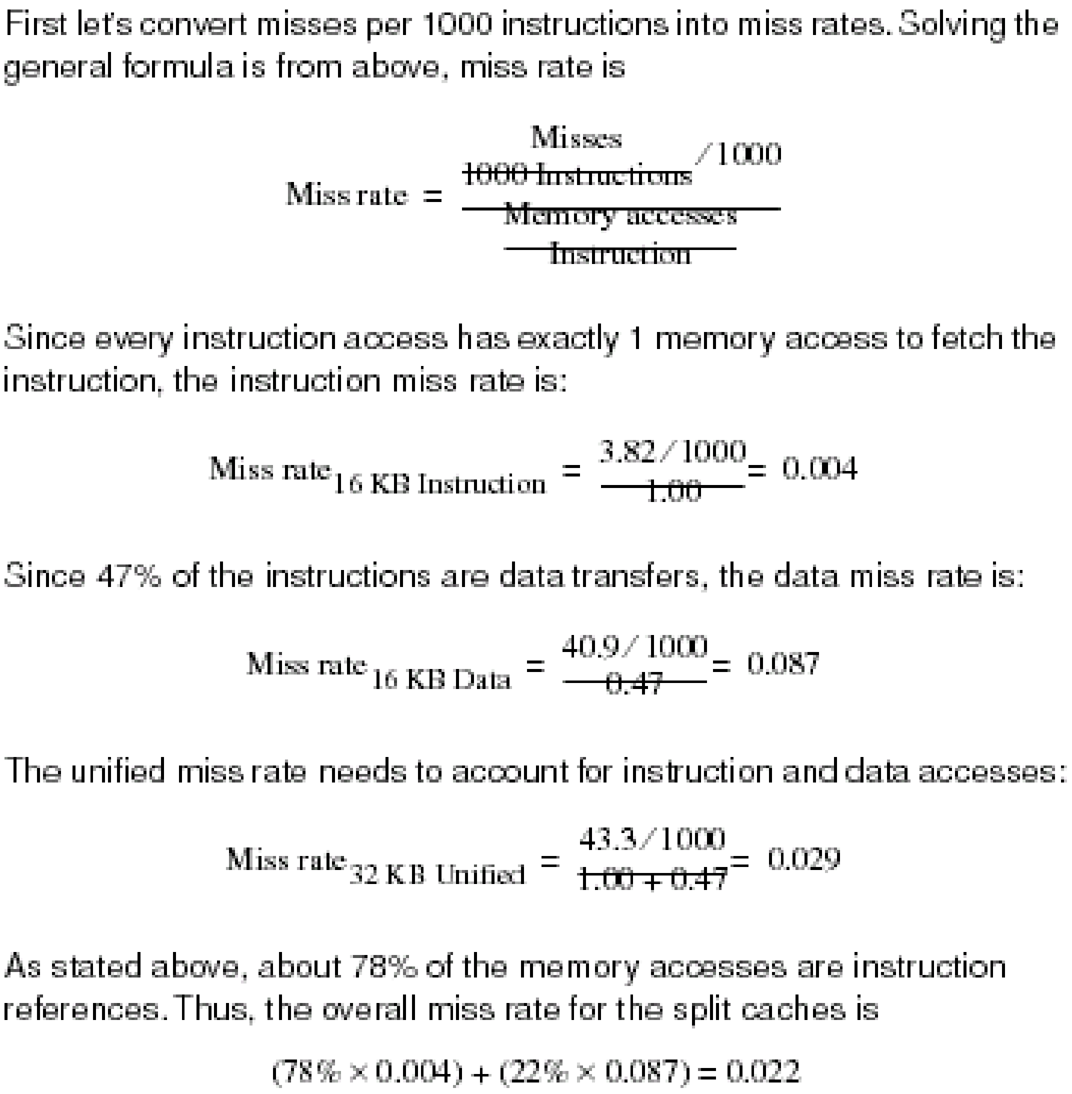
Cache Performance

- Average memory access time = Hit time + Miss rate × Miss penalty

Example

Which has the lower miss rate: a 16-KB instruction cache with a 16-KB data cache or a 32-KB unified cache? Use the miss rates to help calculate the correct answer assuming 47% of the instructions are data transfer instructions. Assume a hit takes 1 clock cycle and the miss penalty is 100 clock cycles. A load or store hit takes 1 extra clock cycle on a unified cache if there is only one cache port to satisfy two simultaneous requests. Using the pipelining terminology of the previous chapter, the unified cache leads to a structural hazard. What is the average memory access time in each case? Assume write-through caches with a write buffer and ignore stalls due to the write buffer.

Answer



First let's convert misses per 1000 instructions into miss rates. Solving the general formula is from above, miss rate is

$$\text{Miss rate} = \frac{\frac{\text{Misses}}{\text{1000 Instructions}} / 1000}{\frac{\text{Memory accesses}}{\text{Instruction}}}$$

Since every instruction access has exactly 1 memory access to fetch the instruction, the instruction miss rate is:

$$\text{Miss rate}_{\text{16 KB Instruction}} = \frac{3.82 / 1000}{1.00} = 0.004$$

Since 47% of the instructions are data transfers, the data miss rate is:

$$\text{Miss rate}_{\text{16 KB Data}} = \frac{40.9 / 1000}{0.47} = 0.087$$

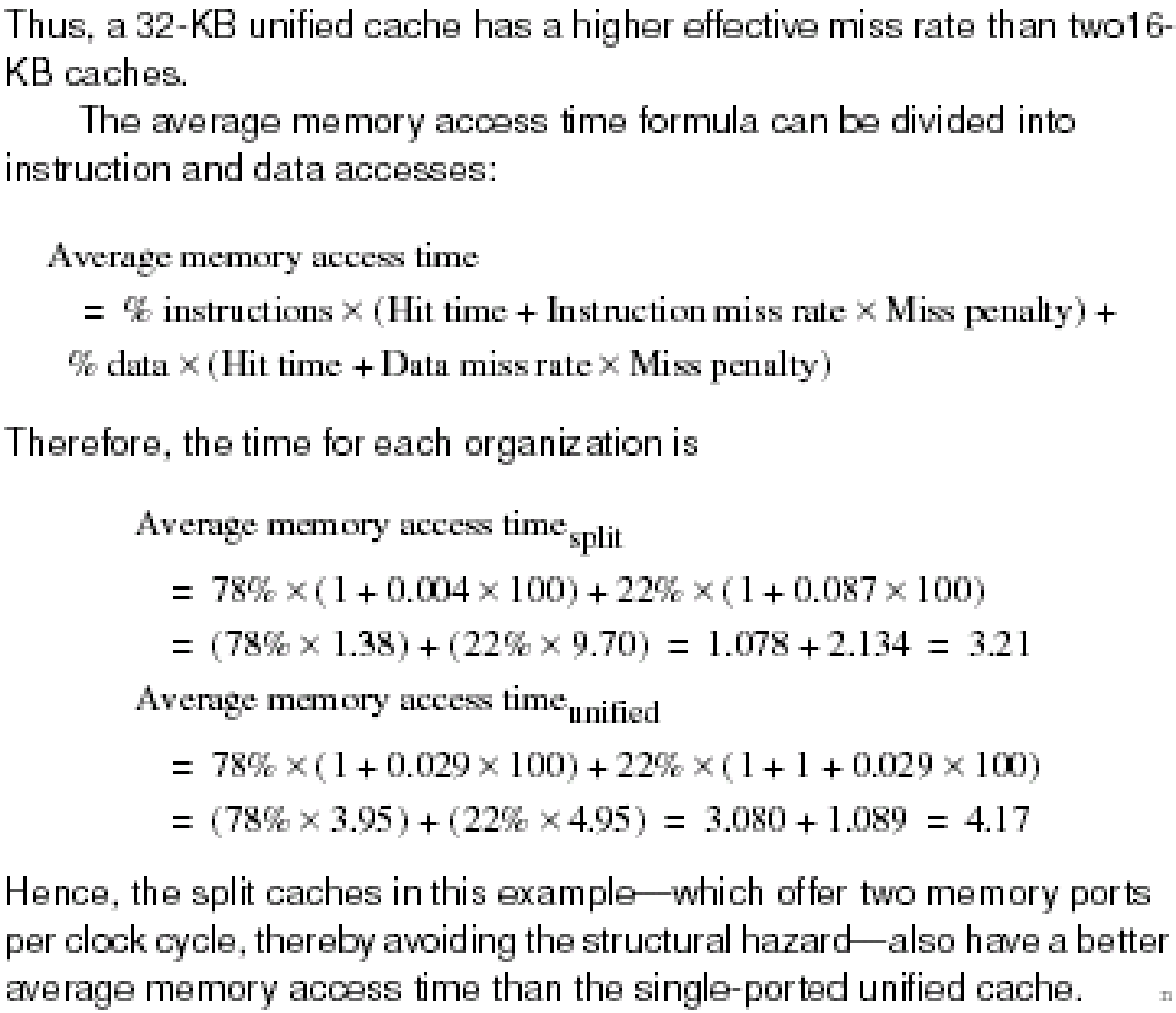
The unified miss rate needs to account for instruction and data accesses:

$$\text{Miss rate}_{\text{32 KB Unified}} = \frac{43.3 / 1000}{1.00 + 0.47} = 0.029$$

As stated above, about 78% of the memory accesses are instruction references. Thus, the overall miss rate for the split caches is

$$(78\% \times 0.004) + (22\% \times 0.087) = 0.022$$

Answer (Cont'd)



Thus, a 32-KB unified cache has a higher effective miss rate than two16-KB caches.

The average memory access time formula can be divided into instruction and data accesses:

$$\begin{aligned}&\text{Average memory access time}\\&\quad = \%\ \text{instructions} \times (\text{Hit time} + \text{Instruction miss rate} \times \text{Miss penalty}) +\\&\quad \%\ \text{data} \times (\text{Hit time} + \text{Data miss rate} \times \text{Miss penalty})\end{aligned}$$

Therefore, the time for each organization is

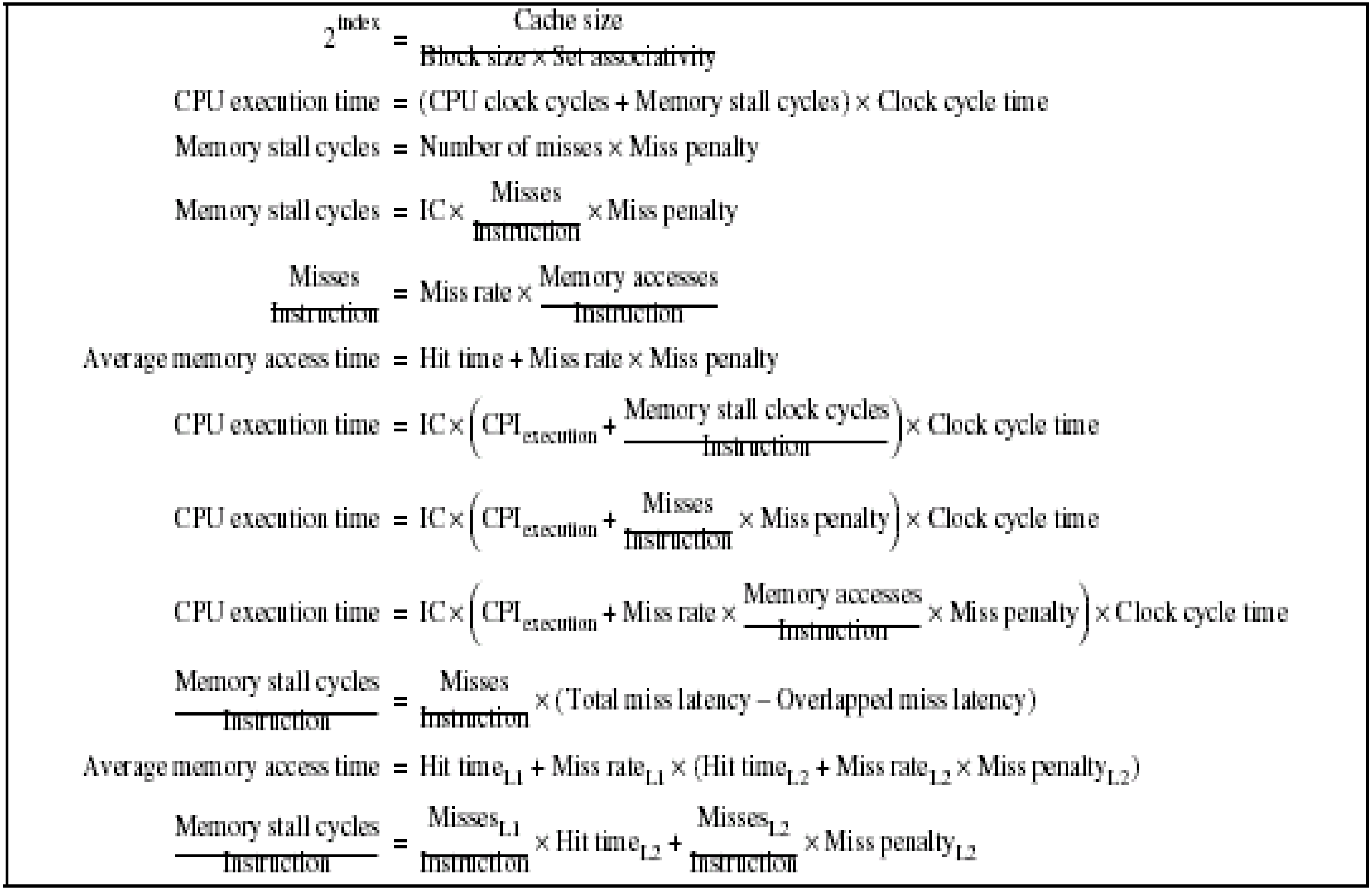
$$\begin{aligned}&\text{Average memory access time}_{\text{split}}\\&\quad = 78\% \times (1 + 0.004 \times 100) + 22\% \times (1 + 0.087 \times 100)\\&\quad = (78\% \times 1.38) + (22\% \times 9.70) = 1.078 + 2.134 = 3.21\\&\text{Average memory access time}_{\text{unified}}\\&\quad = 78\% \times (1 + 0.029 \times 100) + 22\% \times (1 + 1 + 0.029 \times 100)\\&\quad = (78\% \times 3.95) + (22\% \times 4.95) = 3.080 + 1.089 = 4.17\end{aligned}$$

Hence, the split caches in this example—which offer two memory ports per clock cycle, thereby avoiding the structural hazard—also have a better average memory access time than the single-ported unified cache.

Average memory access time and Processor Performance

- An obvious question is whether average memory access time due to cache misses predicts processor performance
- First, there are other reasons for stalls, such as contention due to I/O devices using memory
- Second, the answer depends also on the CPU

Summary of Performance Equations



$$2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$

$$\text{Memory stall cycles} = \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

$$\frac{\text{Misses}}{\text{Instruction}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}}\right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cycle time}$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

$$\text{Average memory access time} = \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \times (\text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}})$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}_{\text{L1}}}{\text{Instruction}} \times \text{Hit time}_{\text{L2}} + \frac{\text{Misses}_{\text{L2}}}{\text{Instruction}} \times \text{Miss penalty}_{\text{L2}}$$

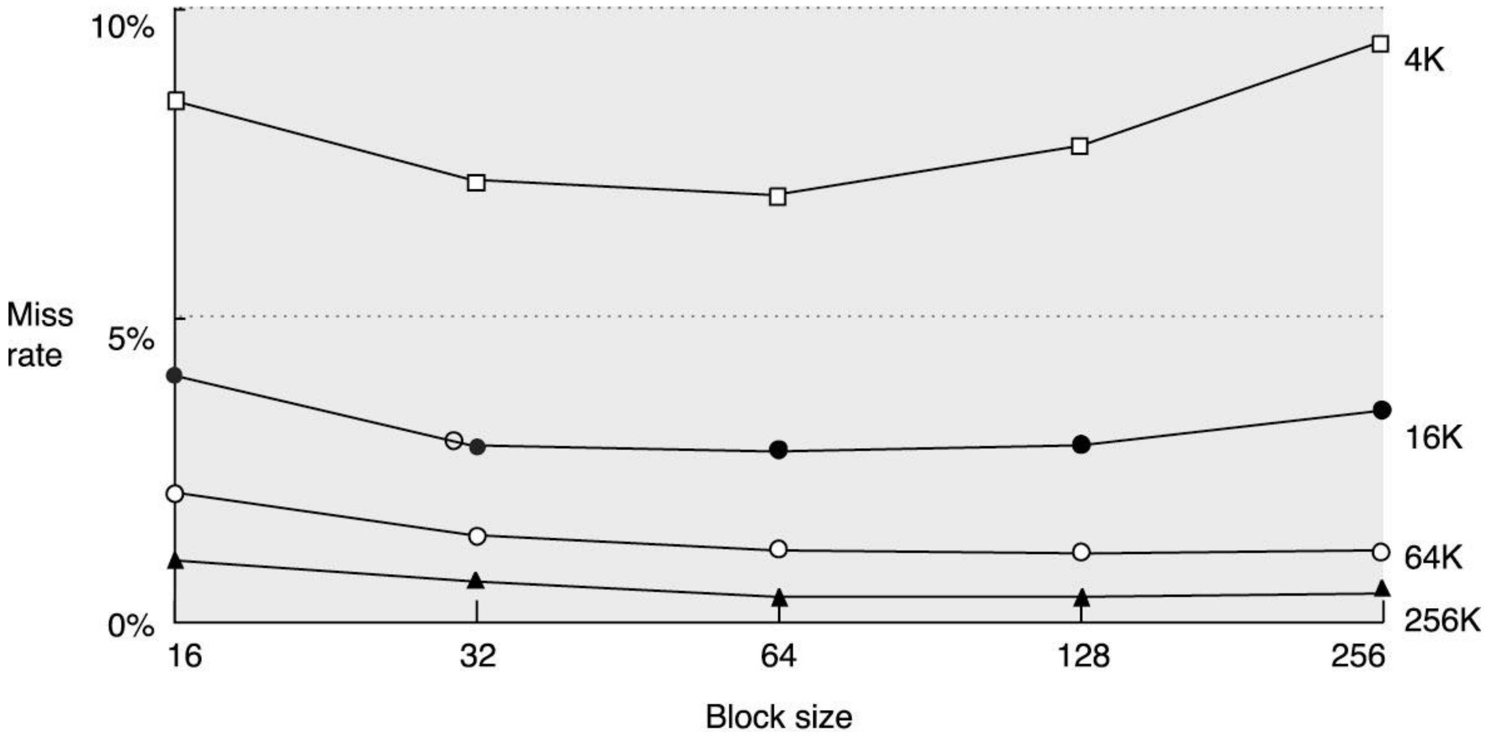
Improving Cache Performance

- Average memory access time = Hit time + Miss rate × Miss penalty
- Four categories of cache optimizations :
  - Reducing the miss penalty : multilevel caches, critical word first, read miss before write miss, merging write buffers, victim caches
  - Reducing the miss rate : larger block size, larger cache size, higher associativity, pseudo-associativity, and compiler optimizations
  - Reducing the miss penalty or miss rate via parallelism : nonblocking caches, hardware prefetching, and compiler prefetching;
  - Reducing the time to hit in the cache : small and simple caches, avoiding address translation, and pipelined cache access.

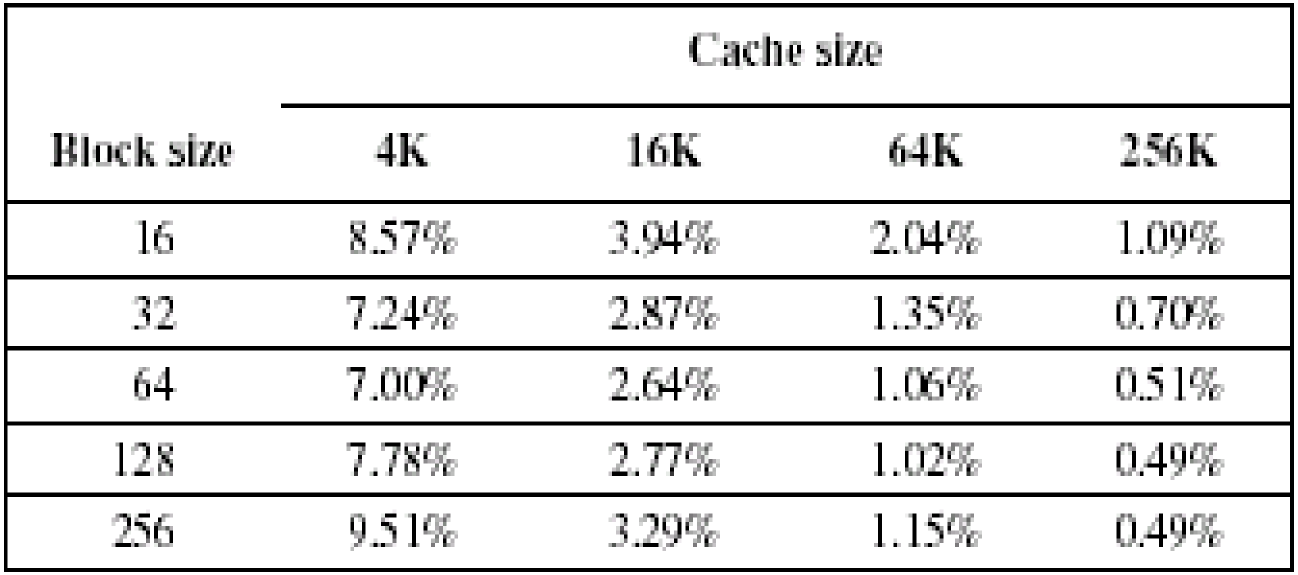
Review: 6 Basic Cache Optimizations

1. Larger block size to reduce miss rate (Compulsory misses)
2. Larger cache size to reduce miss rate (Capacity misses)
3. Higher associativity to reduce miss rate (Conflict misses)
4. Multilevel caches to reduce miss rate
5. Giving reads priority over Writes
    - E.g., Read complete before earlier writes in write buffer
6. Avoiding address translation during cache indexing

First Miss Rate Reduction Technique: Larger Block Size



Actual Miss Rate VS Block Size for Five Different-Sized Caches



	Cache size			
Block size	4K	16K	64K	256K
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

Example

- The table shows the actual miss rates. Assume the memory system takes 80 clock cycles of overhead and then delivers 16 bytes every 2 clock cycles. Thus, it can supply 16 bytes in 82 clock cycles, 32 bytes in 84 clock cycles, and so on. Which block size has the smallest average memory access time for each cache size in the table?

Answer

Average memory access time is

Average memory access time = Hit time + Miss rate × Miss penalty

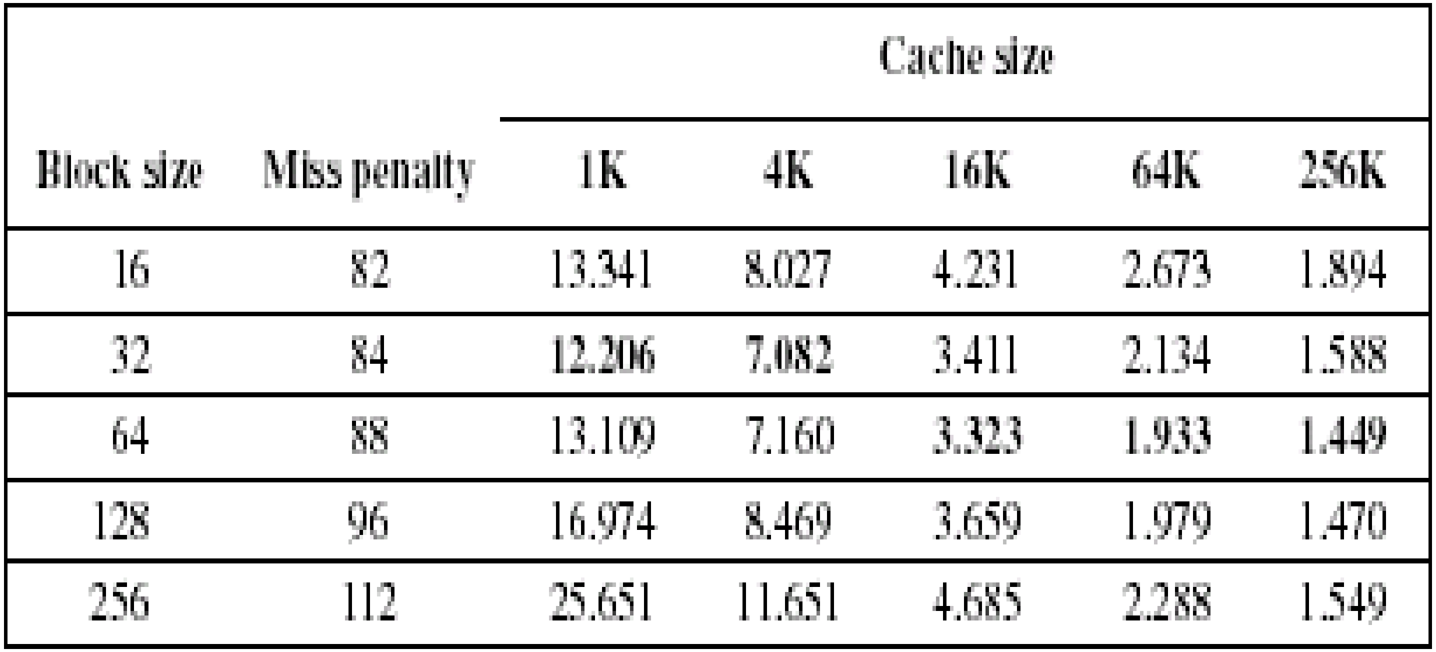
If we assume the hit time is one clock cycle independent of block size, then the access time for a 16-byte block in a 1-KB cache is

Average memory access time = 1 + (15.05% × 82) = 13.341 clock cycles

and for a 256-byte block in a 256-KB cache the average memory access time is

Average memory access time = 1 + (0.49% × 112) = 1.549 clock cycles

Average Memory Access Time vs Block Size



		Cache size				
Block size	Miss penalty	1K	4K	16K	64K	256K
16	82	13.341	8.027	4.231	2.673	1.894
32	84	12.206	7.082	3.411	2.134	1.588
64	88	13.109	7.160	3.323	1.933	1.449
128	96	16.974	8.469	3.659	1.979	1.470
256	112	25.651	11.651	4.685	2.288	1.549

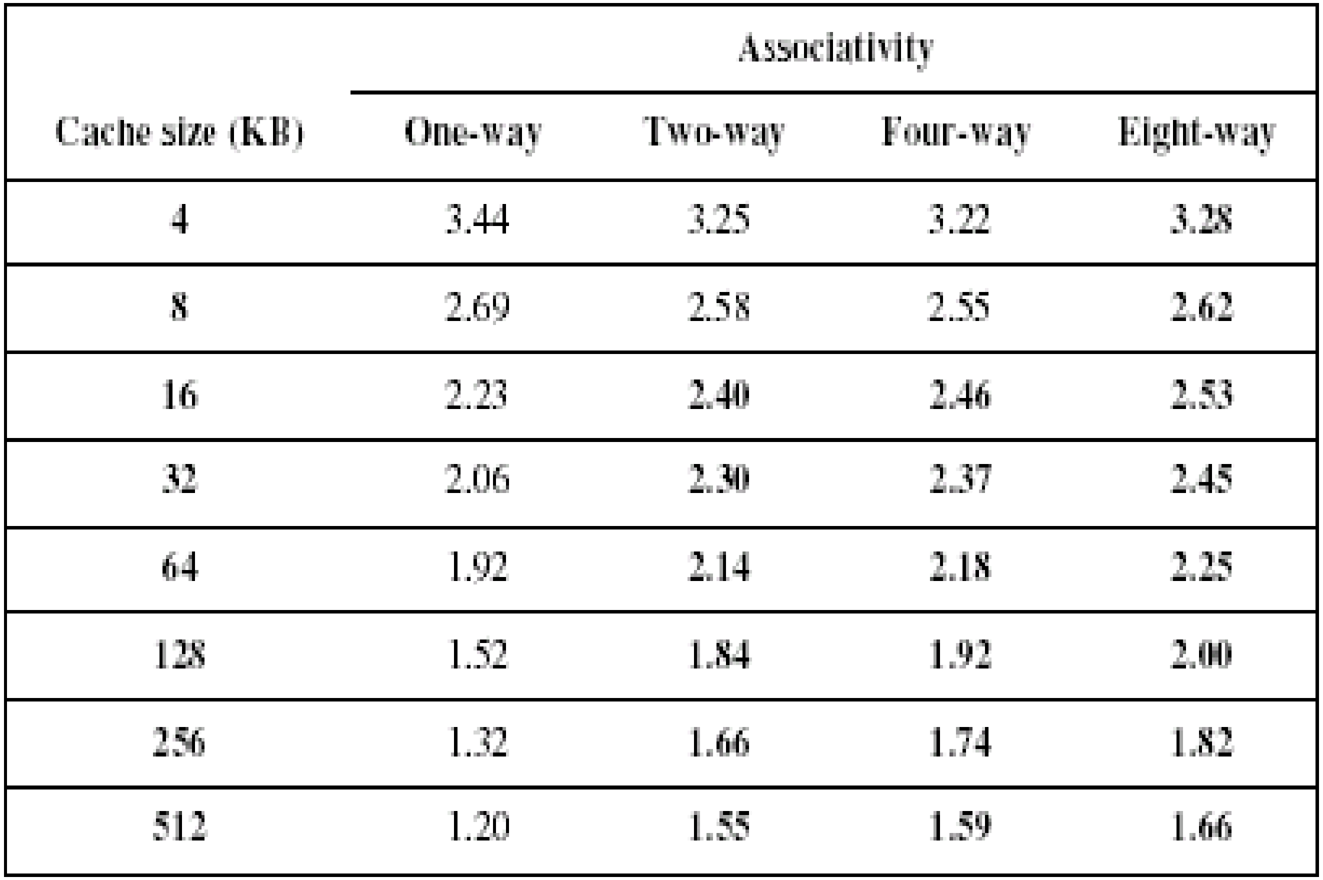
Larger caches

- The obvious way to reduce capacity misses is to increases capacity of the cache
- The obvious drawback is longer hit time and higher cost
- This technique has been especially popular in off-chip caches:
  - The size of second or third level caches in 2001 equals the size of main memory in desktop computers

Higher Associativity

- Two general rules
  - Eight-way set associative is for practical purposes as effective in reducing misses for these sized caches as fully associative
  - 2:1 cache rule of thumb: a direct-mapped cache of size N has about the same miss rate as a 2-way set associative cache of size N/2

Average Memory Access Time Using Miss Rates



Cache size (KB)	Associativity			
	One-way	Two-way	Four-way	Eight-way
4	3.44	3.25	3.22	3.28
8	2.69	2.58	2.55	2.62
16	2.23	2.40	2.46	2.53
32	2.06	2.30	2.37	2.45
64	1.92	2.14	2.18	2.25
128	1.52	1.84	1.92	2.00
256	1.32	1.66	1.74	1.82
512	1.20	1.55	1.59	1.66

Reducing Cache Miss Penalty

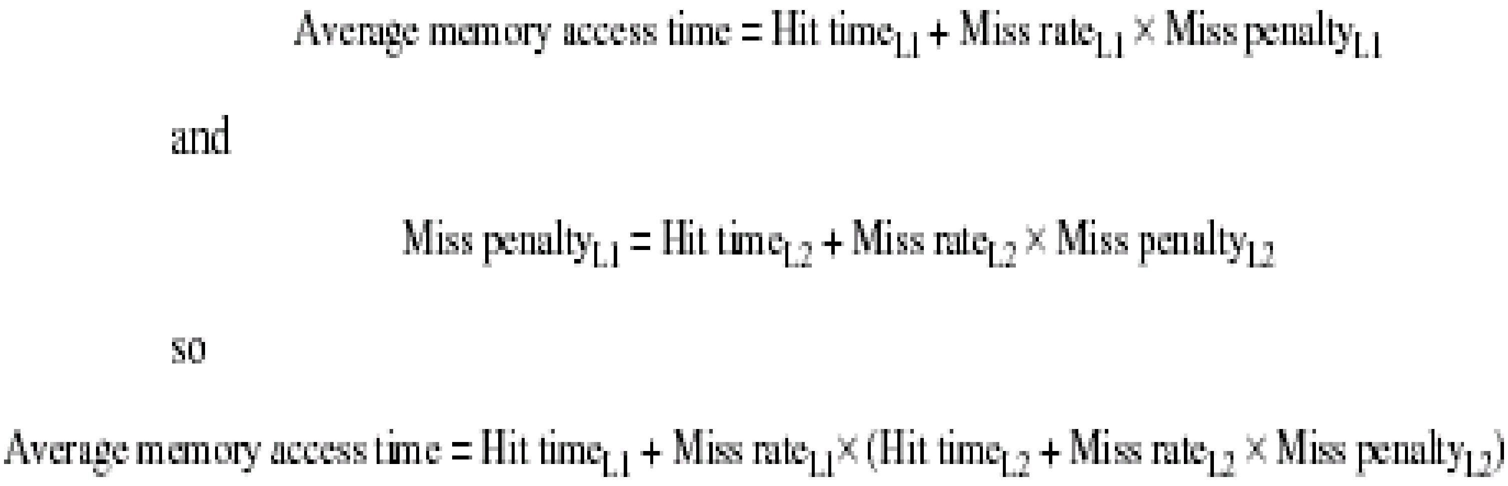
- Reducing cache misses has been the traditional focus of cache research
- Cache performance formula assures us that improvements in miss penalty can be just as beneficial as improvements in miss rate
- The speed of processors is faster than DRAMs, making the relative cost of miss penalties increase over time

Miss Penalty Reduction Technique: Multi-Level Caches

- This technique ignores the CPU, concentrating on the interface between the cache and main memory
- Question: Should I make the cache faster to keep pace with the speed of CPUs, or make the cache larger to overcome the widening gap between the CPU and main memory?
- One answer is: both.
  - Adding another level of cache between the original cache
  - Memory simplifies the decision

Two-Level Caches

- The first-level cache can be small enough to match the clock cycle time of the fast CPU
- The second-level cache can be large enough to capture many accesses that would go to main memory



$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$$

and

$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

so

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

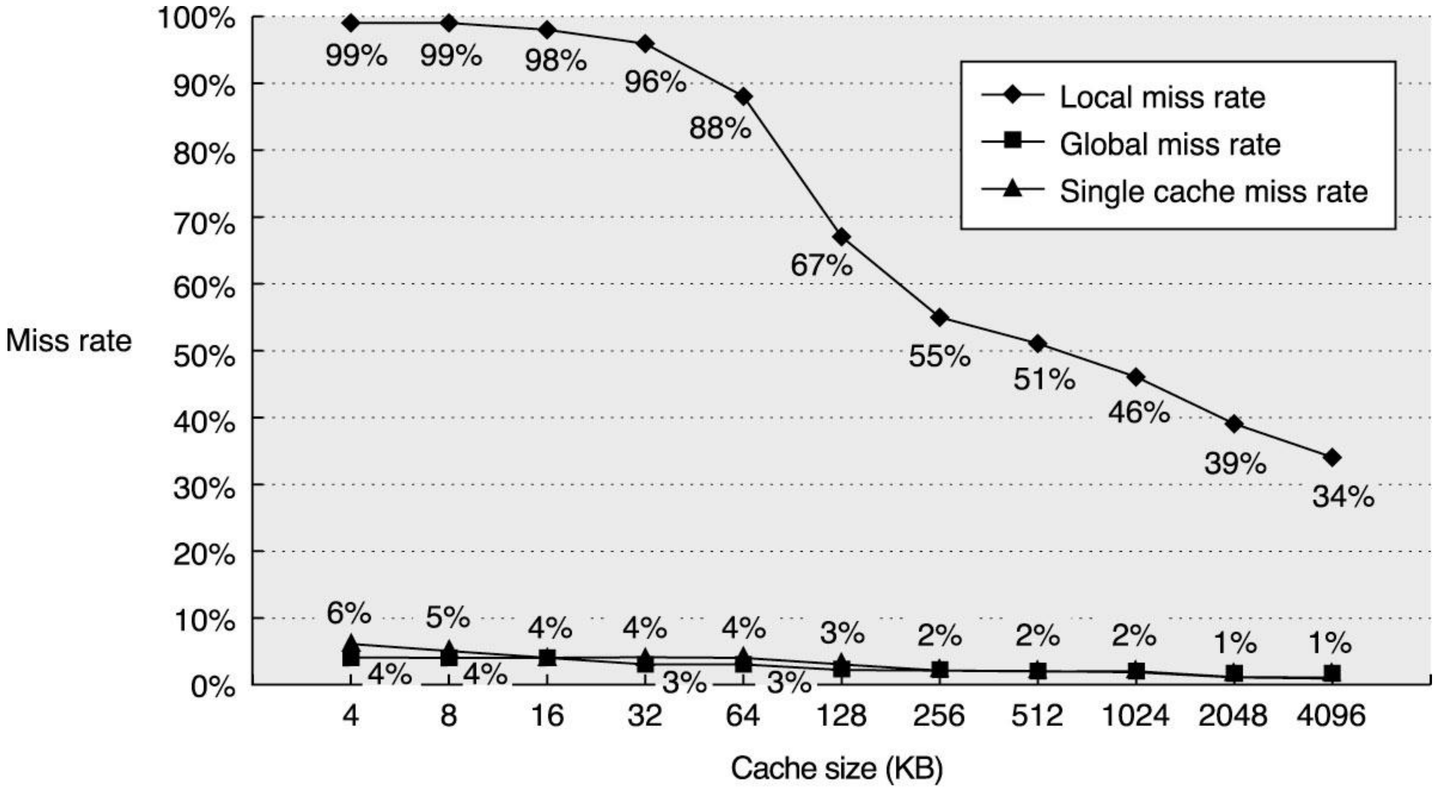
Two-Level Caches (Cont’d)

- Local miss rate—This rate is simply the number of misses in a cache divided by the total number of memory accesses to this cache
  - For the first-level cache it is equal to Miss rateL1 and for the second-level cache it is Miss rateL2
- Global miss rate—The number of misses in the cache divided by the total number of memory accesses generated by the CPU
  - The global miss rate for the first-level cache is still just Miss rateL1 but for the second-level cache it is Miss rateL1 × Miss rateL2

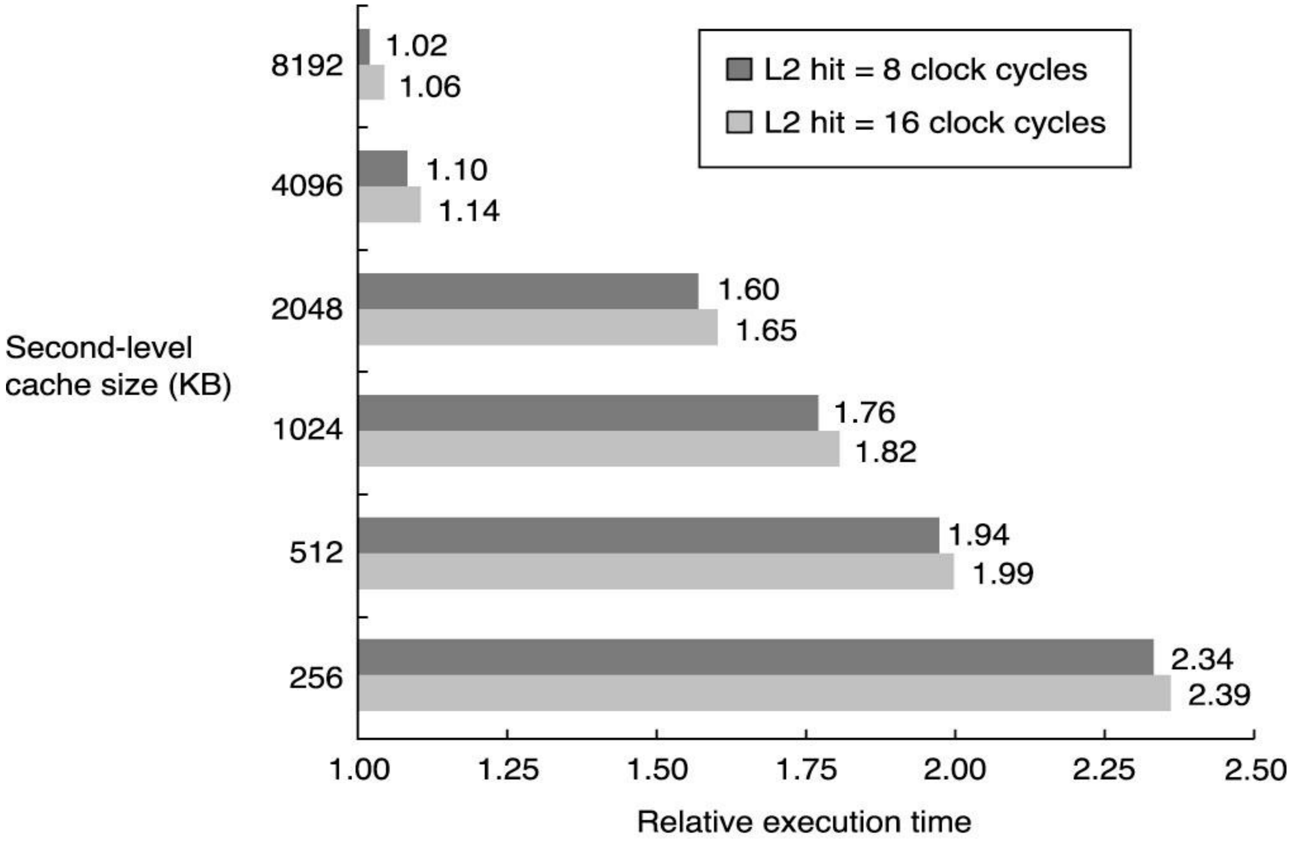
Two-Level Caches (Cont'd)

- Average memory stalls per instruction
  = Misses per instructionL1× Hit timeL2
  + Misses per instructionL2 × Miss penaltyL2

Miss Rates vs Cache Size for Multilevel Caches



Relative Execution Time by Second-level Cache Size



Multilevel Inclusion

- Multilevel inclusion is the natural policy for memory hierarchies: L1 data is always present in L2
- Inclusion is desirable because consistency between I/O and caches can be determined just by checking the second-level cache
- One drawback to inclusion is
  - Measurements can suggest smaller blocks for the smaller first-level cache and larger blocks for the larger second-level cache
  - For example, the Pentium 4 has 64-byte blocks in its L1 caches and 128-byte blocks in its L2 cache

Multilevel Exclusion

- Two questions:
  - What if the designer can only afford an L2 cache that is slightly bigger than the L1 cache?
  - Should a significant portion of its space be used as a redundant copy of the L1 cache?
- Multilevel exclusion: L1 data is never found in L2 cache
- This policy prevents wasting space in L2 cache.
  - For example, the AMD Athlon chip obeys the exclusion property since it has two 64 KB first level caches and only a 256 KB L2 cache
- The essence of all cache designs is balancing fast hits and few misses

Giving Priority to Read Misses over Writes

- With a write-through cache the most important improvement is a write buffer of the proper size
- Write buffers do complicate memory accesses in that they might hold the updated value of a location needed on a read miss
  - The read miss waits until the write buffer is empty
  - Check the contents of the write buffer on a read miss to ensure no conflicts occur and the memory system is available
- Virtually all desktop and server processors use the latter approach, giving reads priority over writes

Giving Priority to Read Misses over Writes(Cont'd)

- Instead of writing the dirty block to memory, and then reading memory, we could copy the dirty block to a buffer, then read memory, and then write memory
- This way the CPU read, for which the processor is probably waiting, will finish sooner
- If a read miss occurs, the processor can either stall until the buffer is empty or check the addresses of the words in the buffer for conflicts

Avoiding Address Translation During Indexing of the Cache (1)

- Caches must cope with the virtual address from the processor to a physical address to access memory
- A common optimization is to use the page offset to index the cache
- Even a small and simple cache must cope with the translation of a virtual address from the CPU to a physical address to access memory
- Processors treat main memory as just another level of the memory hierarchy
- Thus the address of the virtual memory that exists on disk must be mapped onto the main memory
- The guideline of making the common case fast suggests that we use virtual addresses for the cache
  - since hits are much more common than misses
- Such caches are termed *virtual caches*

Avoiding Address Translation During Indexing of the Cache (2)

- Virtual caches use virtual addresses for the cache
- Physical cache used to identify the traditional cache that uses physical addresses
- It is important to distinguish two tasks:
  - Indexing the cache
  - Comparing addresses
- Thus, the issues are whether a virtual or physical address is used to index the cache and whether a virtual or physical index is used in the tag comparison

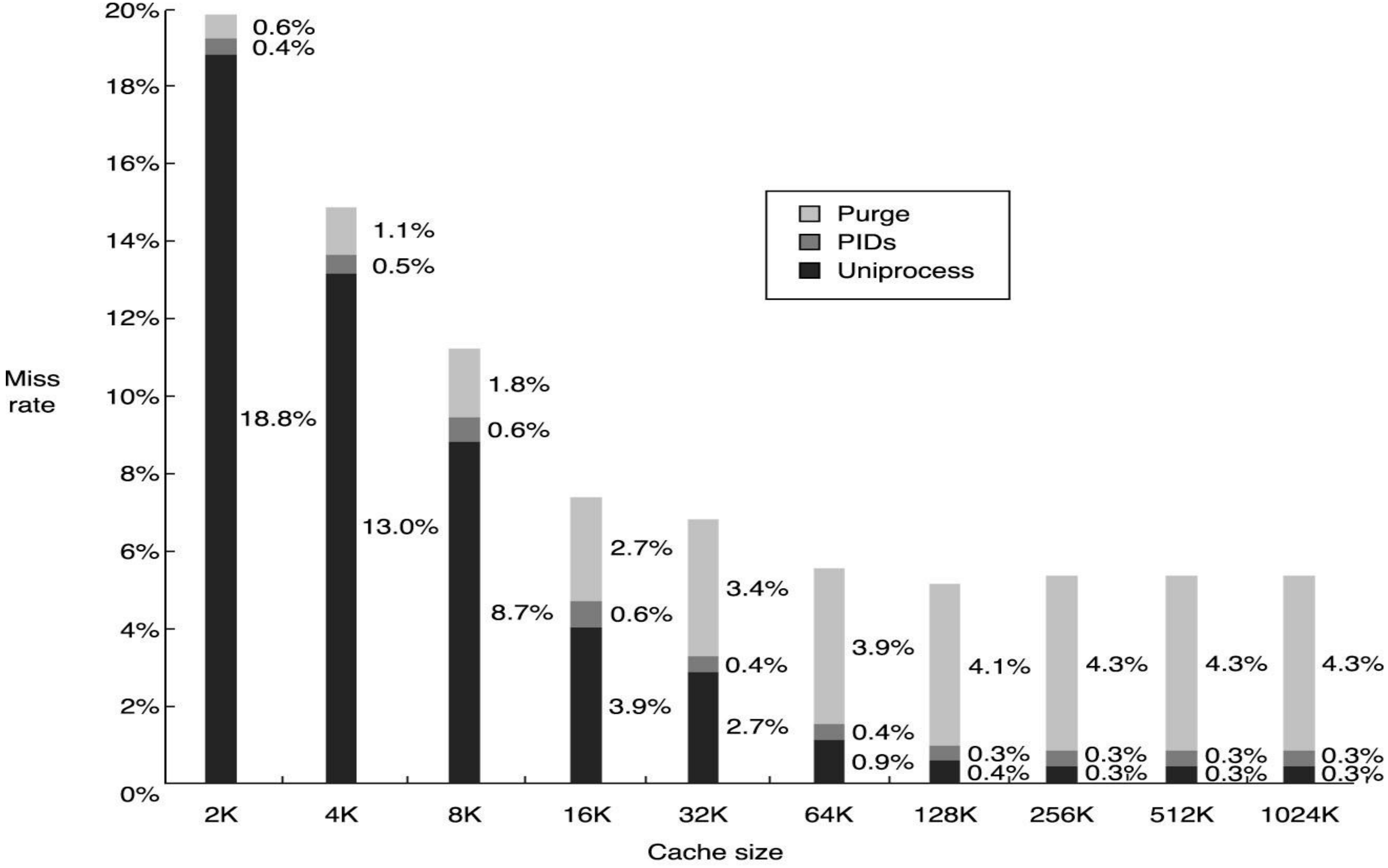
Avoiding Address Translation During Indexing of the Cache (3)

- Full virtual addressing for both index and tags eliminates address translation time from a cache hit
- Why doesn’t everyone build virtually addressed caches?
  - Protection
  - Cache flushing
  - Synonyms or aliases

Avoiding Address Translation During Indexing of the Cache (4)

- Protection
  - Page level protection is checked as part of the virtual to physical address translation, and it must be enforced no matter what
  - A solution is to copy the protection information from the TLB on a miss, add a field to hold it, and check it on every access to the virtually addressed cache
- Cache flushing
  - Every time a process is switched, the virtual addresses refer to different physical addresses, requiring the cache to be flushed
  - One solution is to increase the width of the cache address tag with a *process-identifier tag* (PID)
  - If the operating system assigns these tags to processes, it only need flush the cache when a PID is recycled

Impact on Miss Rates of Flushing



Avoiding Address Translation During Indexing of the Cache (5)

- *Synonyms* or *aliases*
  - —A third reason why virtual caches are not more popular is that operating systems and user programs may use two different virtual addresses for the same physical address
  - —These duplicate addresses, called *synonyms* or *aliases*, could result in two copies of the same data in a virtual cache; if one is modified, the other will have the wrong value
  - —With a physical cache this wouldn't happen, since the accesses would first be translated to the same physical cache block.

Avoiding Address Translation During Indexing of the Cache (6)

- Hardware solutions to the synonym problem, called *anti-aliasing*, guarantee every cache block a unique physical address
- Software can make this problem much easier by forcing aliases to share some address bits
  - —E.g., page coloring
- I/O typically uses physical addresses and thus would require mapping to virtual addresses to interact with a virtual cache
- One alternative to get the best of both virtual and physical caches is to use part of the page offset

11 Advanced Cache Optimizations

- Reducing hit time

1. Small and simple caches
2. Way prediction
3. Trace caches

- Increasing cache bandwidth

4. Pipelined caches
5. Multibanked caches
6. Nonblocking caches

- Reducing Miss Penalty

7. Critical word first
8. Merging write buffers

- Reducing Miss Rate

9. Compiler optimizations

- Reducing miss penalty or miss rate via parallelism

10. Hardware prefetching
11. Compiler prefetching

11 Advanced Cache Optimizations

- **Reducing hit time**

1. **Small and simple caches**
2. **Way prediction**
3. **Trace caches**

- Increasing cache bandwidth

4. Pipelined caches
5. Multibanked caches
6. Nonblocking caches

- Reducing Miss Penalty

7. Critical word first
8. Merging write buffers

- Reducing Miss Rate

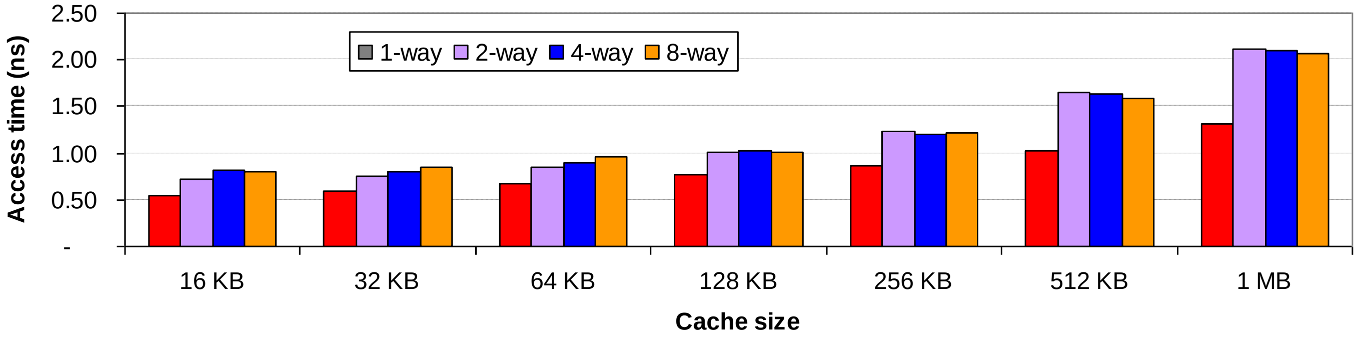
9. Compiler optimizations

- Reducing miss penalty or miss rate via parallelism

10. Hardware prefetching
11. Compiler prefetching

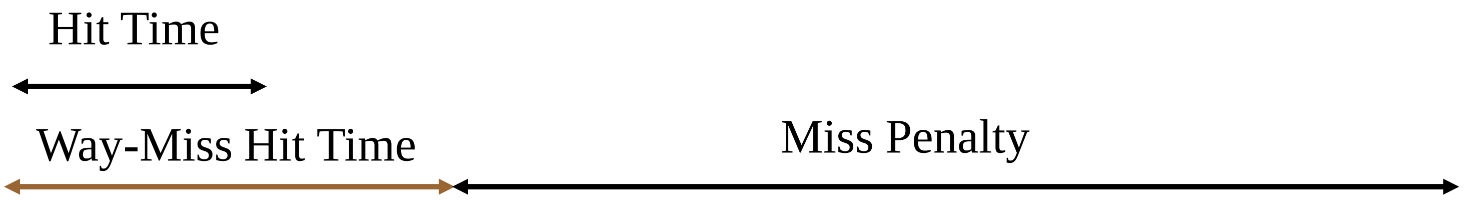
1. Fast Hit times via Small and Simple Caches

- Index tag memory and then compare takes time
- ⇒ Small cache can help hit time since smaller memory takes less time to index
  - E.g., L1 caches same size for 3 generations of AMD microprocessors: K6, Athlon, and Opteron
  - Also L2 cache small enough to fit on chip with the processor avoids time penalty of going off chip
- Simple ⇒ direct mapping
  - Can overlap tag check with data transmission since no choice
- Access time estimate for 90 nm using CACTI model 4.0
  - Median ratios of access time relative to the direct-mapped caches are 1.32, 1.39, and 1.43 for 2-way, 4-way, and 8-way caches



2. Fast Hit times via Way Prediction

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
- Way prediction: keep extra bits in cache to predict the "way," or block within the set, of next cache access.
  - Multiplexor is set early to select desired block, only 1 tag comparison performed that clock cycle in parallel with reading the cache data
  - Miss ⇒ 1st check other blocks for matches in next clock cycle



Hit Time

Way-Miss Hit Time

Miss Penalty

Way-Prediction

- In way-prediction, extra bits are kept in the cache to predict the set of the next cache access
- This prediction means the multiplexor is set early to select the desired set
- A miss results in checking the other sets for matches in subsequent clock cycles
- In addition to improving performance, way prediction can reduce power for embedded applications

Pseudoassociative (Column Associate)

- Accesses proceed just as in the direct-mapped cache for a hit
- On a miss before going to the next lower level of the memory hierarchy, a second cache entry is checked to see if it matches there
- A simple way is to invert the most significant bit of the index field to find the other block in the "pseudo set"
- Pseudo-associative caches then have one fast and one slow hit time—corresponding to a regular hit and a pseudo hit—in addition to the miss penalty

3. Fast Hit times via Trace Cache (Pentium 4 only; and last time?)

- Find more instruction level parallelism?
  How avoid translation from x86 to microops?
- Trace cache in Pentium 4

1. Dynamic traces of the executed instructions vs. static sequences of instructions as determined by layout in memory
   – Built-in branch predictor
2. Cache the micro-ops vs. x86 instructions
   – Decode/translate from x86 to micro-ops on trace cache miss

+ 1. ⇒ better utilize long blocks (don’t exit in middle of block, don’t enter at label in middle of block)

- 1. ⇒ complicated address mapping since addresses no longer aligned to power-of-2 multiples of word size

- 1. ⇒ instructions may appear multiple times in multiple dynamic traces due to different branch outcomes

11 Advanced Cache Optimizations

- Reducing hit time

1. Small and simple caches
2. Way prediction
3. Trace caches

- **Increasing cache bandwidth**

4. **Pipelined caches**
5. **Multibanked caches**
6. **Nonblocking caches**

- Reducing Miss Penalty

7. Critical word first
8. Merging write buffers

- Reducing Miss Rate

9. Compiler optimizations

- Reducing miss penalty or miss rate via parallelism

10. Hardware prefetching
11. Compiler prefetching

4: Increasing Cache Bandwidth by Pipelining

- Pipeline cache access to maintain bandwidth, but higher latency
- Instruction cache access pipeline stages:

  1: Pentium

  2: Pentium Pro through Pentium III

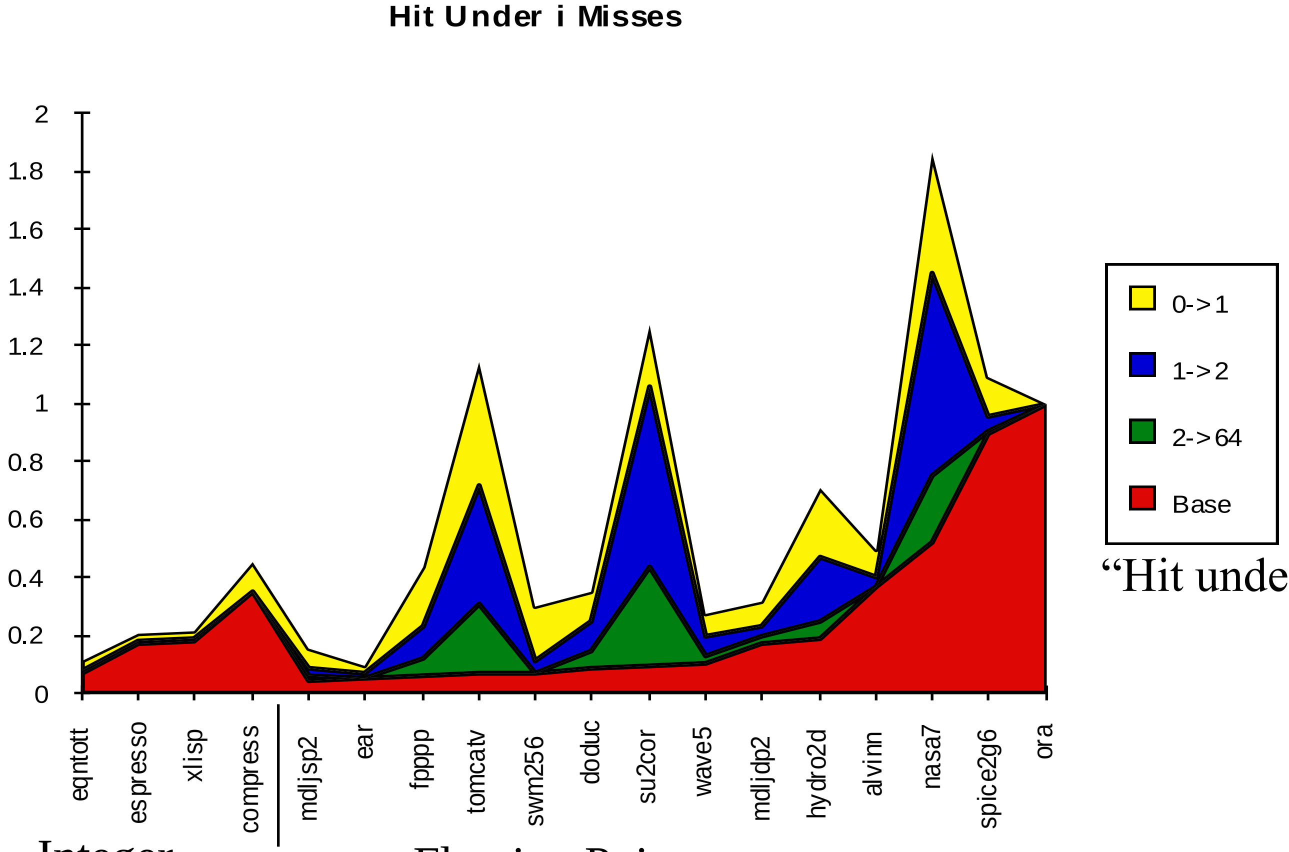
  4: Pentium 4

- ⇒ greater penalty on mispredicted branches
- ⇒ more clock cycles between the issue of the load and the use of the data

5. Increasing Cache Bandwidth: Non-Blocking Caches

- *Non-blocking cache* or *lockup-free cache* allow data cache to continue to supply cache hits during a miss
  - —requires F/E bits on registers or out-of-order execution
  - —requires multi-bank memories
- "*hit under miss*" reduces the effective miss penalty by working during miss vs. ignoring CPU requests
- "*hit under multiple miss*" or "*miss under miss*" may further lower the effective miss penalty by overlapping multiple misses
  - — Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
  - — Requires muliple memory banks (otherwise cannot support)
  - — Penium Pro allows 4 outstanding memory misses

Value of Hit Under Miss for SPEC (old data)



0->1
1->2
2->64
Base

"Hit under n Misses"

Integer

Floating Point

- FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26
- Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19
- 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss, SPEC 92

6: Increasing Cache Bandwidth via Multiple Banks

- Rather than treat the cache as a single monolithic block, divide into independent banks that can support simultaneous accesses
  - E.g.,T1 ("Niagara") L2 has 4 banks
- Banking works best when accesses naturally spread themselves across banks ⇒ mapping of addresses to banks affects behavior of memory system
- Simple mapping that works well is "sequential interleaving"
  - Spread block addresses sequentially across banks
  - E,g, if there 4 banks, Bank 0 has all blocks whose address modulo 4 is 0; bank 1 has all blocks whose address modulo 4 is 1; …

11 Advanced Cache Optimizations

- Reducing hit time

1. Small and simple caches
2. Way prediction
3. Trace caches

- Increasing cache bandwidth

4. Pipelined caches
5. Multibanked caches
6. Nonblocking caches

- **Reducing Miss Penalty**

7. **Critical word first**
8. **Merging write buffers**

- **Reducing Miss Rate**

9. **Compiler optimizations**

- Reducing miss penalty or miss rate via parallelism

10. Hardware prefetching
11. Compiler prefetching

7. Reduce Miss Penalty: Early Restart and Critical Word First

- Don’t wait for full block before restarting CPU
- *Early restart*— As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - — Spatial locality ⇒ tend to want next sequential word, so not clear size of benefit of just early restart
- *Critical Word First* —Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block
  - — Long blocks more popular today ⇒ Critical Word 1st Widely used

block

Example

- Let’s assume a computer has a 64-byte cache block, an L2 cache that takes 11 clock cycles to get the critical 8 bytes, and then 2 clock cycles per 8 bytes to fetch the rest of the block. Calculate the average miss penalty for critical word first, assuming that there will be no other accesses to the rest of the block until it is completely fetched. Then calculate assuming the following instructions reads data sequentially 8 bytes at a time from the rest of the block. Compare the times with and without critical word first.

Answer

- The average miss penalty is 11 clock cycles for critical word first. The Athlon can issue two loads per clock cycle, which is faster than the L2 cache can supply data. Thus, it would take 11 + (8-1) x 2 or 25 clock cycles for the CPU to sequentially read a full cache block. Without critical word first, it would take 25 clocks cycle to load the block, and then 8/2 or 4 clocks to issue the loads, giving 29 clock cycles total.

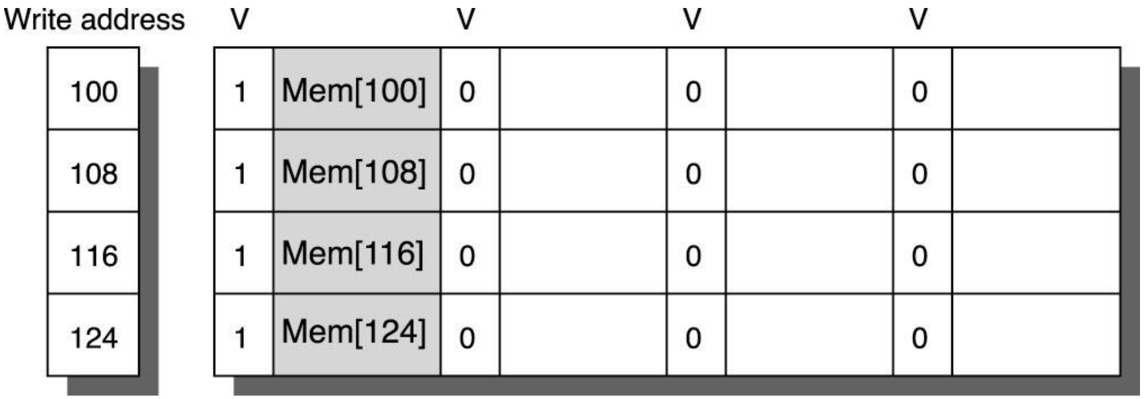
8. Merging Write Buffer to Reduce Miss Penalty

- Write buffer to allow processor to continue while waiting to write to memory
- If buffer contains modified blocks, the addresses can be checked to see if address of new data matches the address of a valid write buffer entry
- If so, new data are combined with that entry
- Increases block size of write for write-through cache of writes to sequential words, bytes since multiword writes more efficient to memory
- The Sun T1 (Niagara) processor, among many others, uses write merging

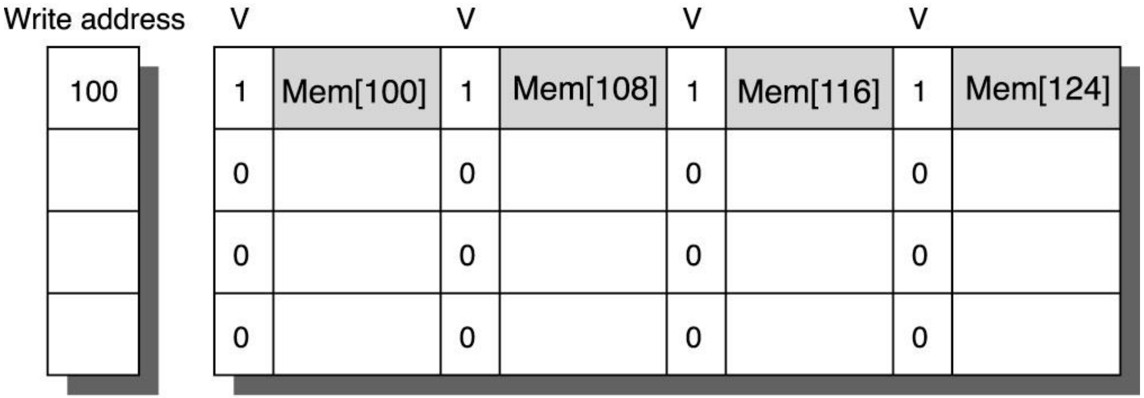
Merging Write Buffer

- Write through caches rely on write buffers, as all stores must be sent to the next lower level of the hierarchy
- If the write buffer is empty, the data and the full address are written in the buffer
- The CPU continues working while the write buffer prepares to write the word to memory
- If the buffer contains other modified blocks, the addresses must be checked
- If so, the new data are combined with that entry, called *write merging*
- If the buffer is full and there is no address match, the cache (and CPU) must wait until the buffer has an empty entry

Write Merging



Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

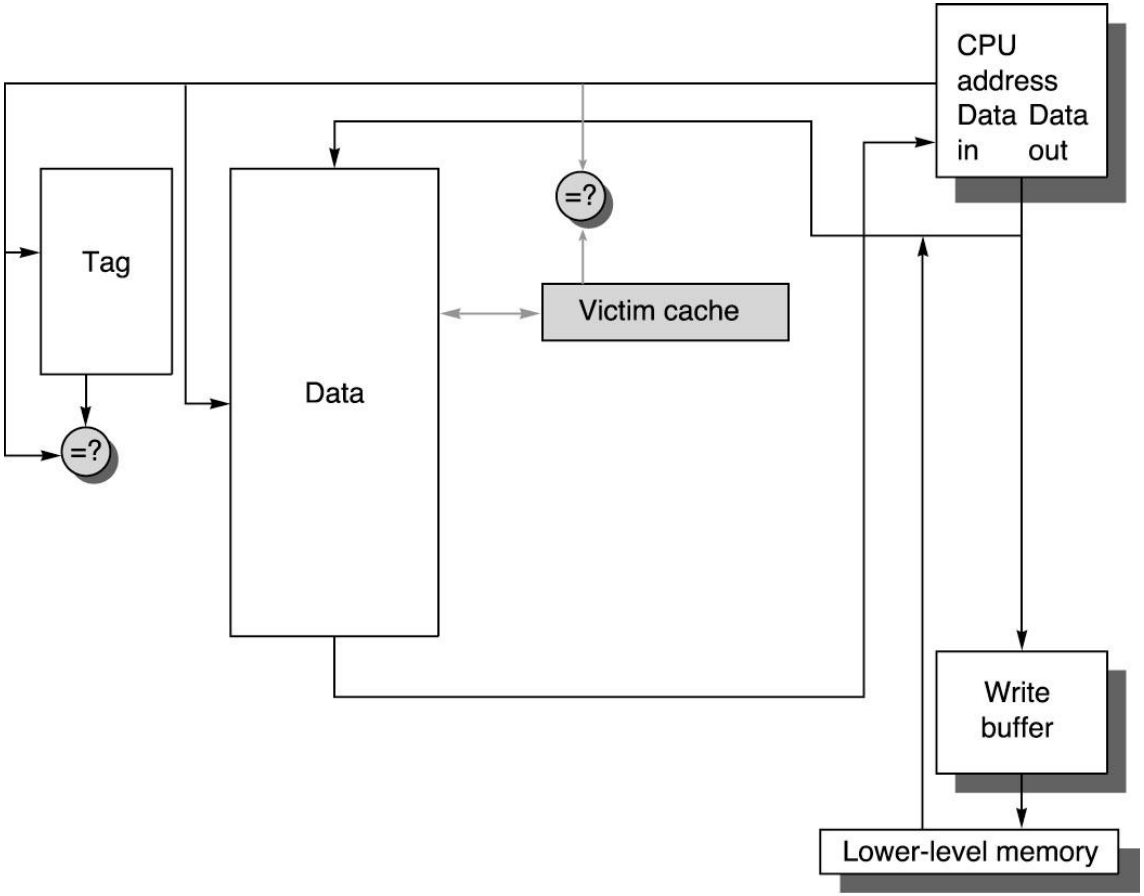


Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Victim Caches

- One approach to lower miss penalty is to remember what was discarded in case it is needed again
- Such "recycling" requires a small, fully associative cache between a cache and its refill path
- Victim cache contains only blocks that
  - —are discarded from a cache because of a miss
  - —are checked on a miss to see if they have the desired data before going to the next lower-level memory
- If it is found there, the victim block and cache block are swapped

Placement of Victim Cache in the Memory Hierarchy



Misses Categories

- Compulsory misses
- Capacity misses
- Conflict (collision or interference) misses

9. Reducing Misses by Compiler Optimizations

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- Instructions
  - Reorder procedures in memory so as to reduce conflict misses
  - Profiling to look at conflicts(using tools they developed)
- Data
  - *Merging Arrays*: improve spatial locality by single array of compound elements vs. 2 arrays
  - *Loop Interchange*: change nesting of loops to access data in order stored in memory
  - *Loop Fusion*: Combine 2 independent loops that have same looping and some variables overlap
  - *Blocking*: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows

Merging Arrays Example

```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of stuctures */
struct merge {
  int val;
  int key;
};
struct merge merged_array[SIZE];
```

Reducing conflicts between val & key; improve spatial locality

Loop Interchange Example

```
/* Before */
for (k = 0; k < 100; k = k+1)
   for (j = 0; j < 100; j = j+1)
      for (i = 0; i < 5000; i = i+1)
         x[i][j] = 2 * x[i][j];
/* After */
for (k = 0; k < 100; k = k+1)
   for (i = 0; i < 5000; i = i+1)
      for (j = 0; j < 100; j = j+1)
         x[i][j] = 2 * x[i][j];
```

Sequential accesses instead of striding through memory every 100 words; improved spatial locality

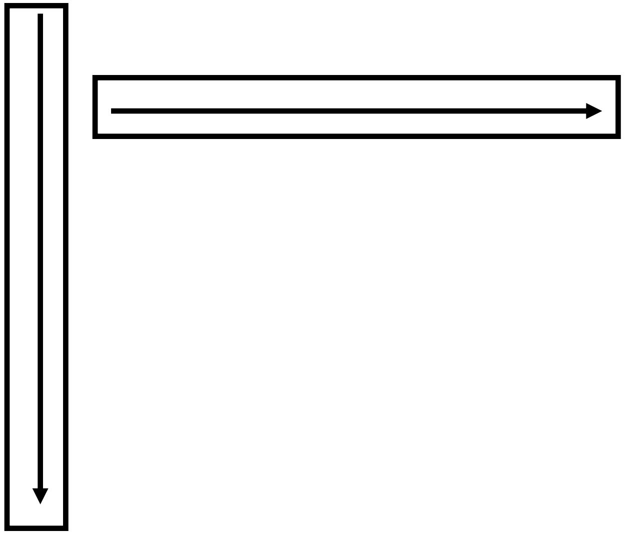
Loop Fusion Example

```
/* Before */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
       a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
       d[i][j] = a[i][j] + c[i][j];
/* After */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
   {   a[i][j] = 1/b[i][j] * c[i][j];
       d[i][j] = a[i][j] + c[i][j];}
```

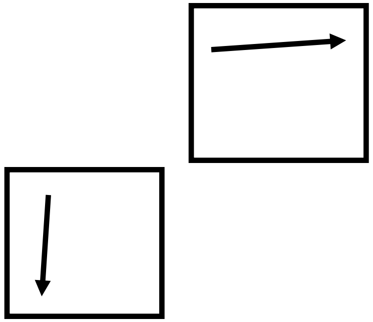
2 misses per access to `a` & `c` vs. one miss per access; improve spatial locality

Blocking Example

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
     {r = 0;
      for (k = 0; k < N; k = k+1){
        r = r + y[i][k]*z[k][j];};
      x[i][j] = r;
     };
```



- Two Inner Loops:
  - — Read all NxN elements of z[]
  - — Read N elements of 1 row of y[] repeatedly
  - — Write N elements of 1 row of x[]



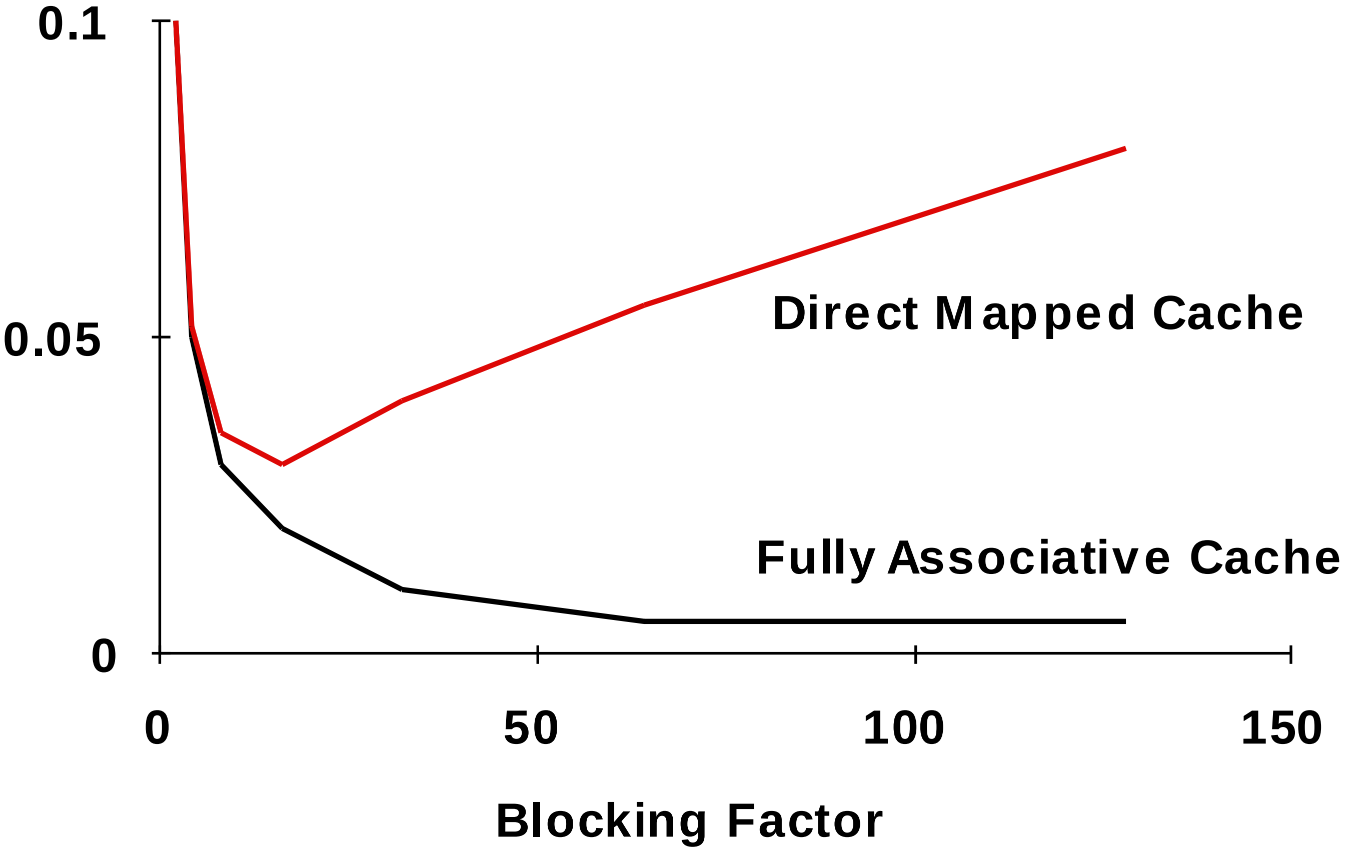
- Capacity Misses a function of N & Cache Size:
  - — $2N^3 + N^2$ => (assuming no conflict; otherwise …)
- Idea: compute on BxB submatrix that fits

Blocking Example

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
    for (j = jj; j < min(jj+B-1,N); j = j+1)
     {r = 0;
      for (k = kk; k < min(kk+B-1,N); k = k+1) {
        r = r + y[i][k]*z[k][j];};
      x[i][j] = x[i][j] + r;
     };
```

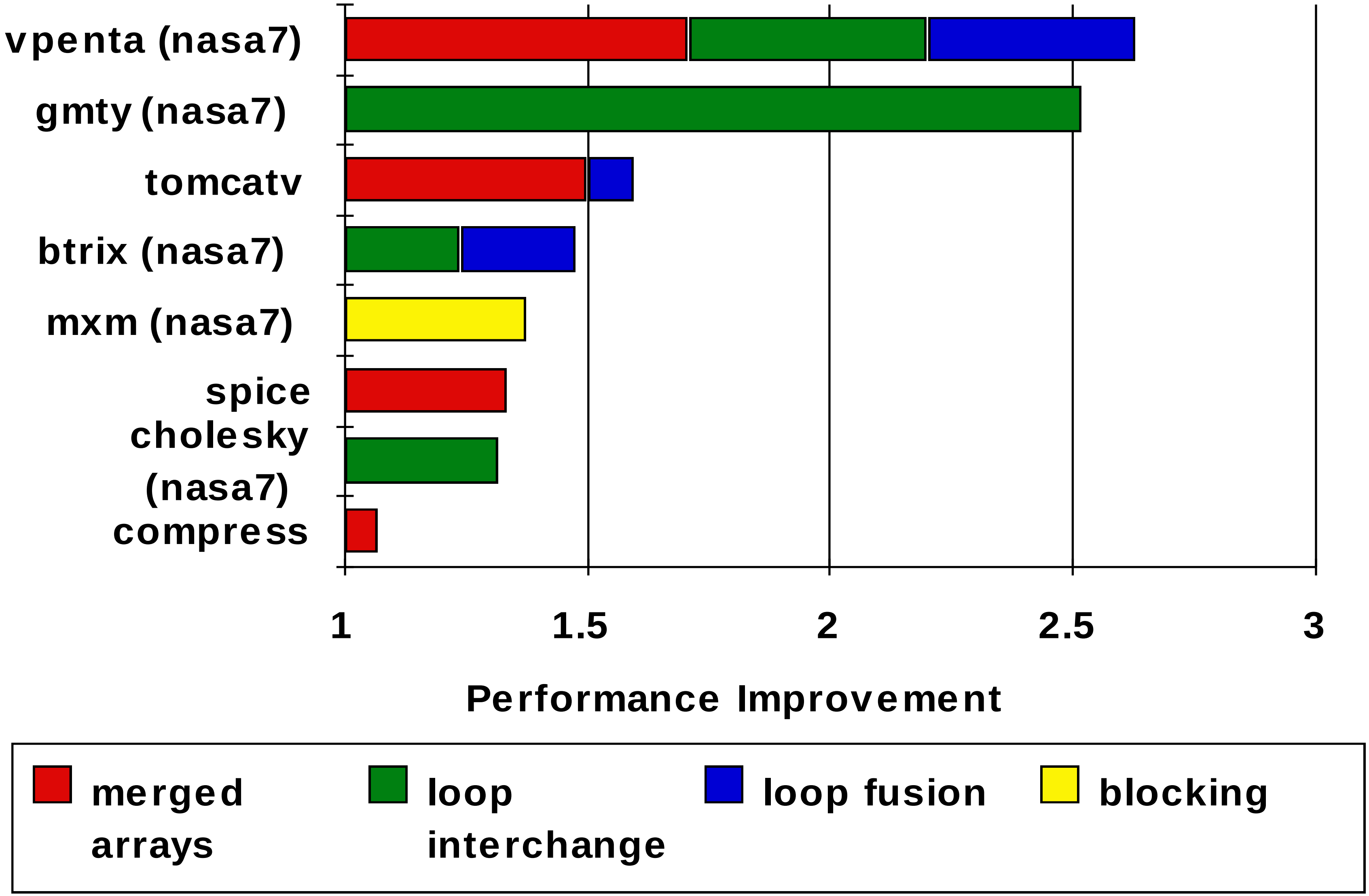
- B called *Blocking Factor*
- Capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$
- Conflict Misses Too?

Reducing Conflict Misses by Blocking



- Conflict misses in caches not FA vs. Blocking size
  - Lam et al [1991] a blocking factor of 24 had a fifth the misses vs. 48 despite both fit in cache

Summary of Compiler Optimizations to Reduce Cache Misses (by hand)



11 Advanced Cache Optimizations

- Reducing hit time

1. Small and simple caches
2. Way prediction
3. Trace caches

- Increasing cache bandwidth

4. Pipelined caches
5. Multibanked caches
6. Nonblocking caches

- Reducing Miss Penalty

7. Critical word first
8. Merging write buffers

- Reducing Miss Rate

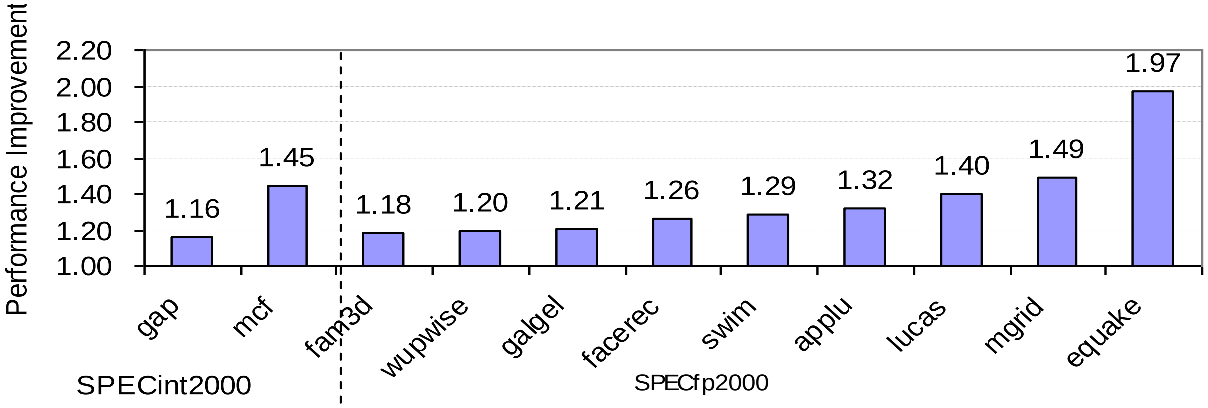
9. Compiler optimizations

- **Reducing miss penalty or miss rate via parallelism**

10. **Hardware prefetching**
11. **Compiler prefetching**

10. Reducing Misses by Hardware Prefetching of Instructions & Data

- Prefetching relies on having extra memory bandwidth that can be used without penalty
- Instruction Prefetching
  - Typically, CPU fetches 2 blocks on a miss: the requested block and the next consecutive block.
  - Requested block is placed in instruction cache when it returns, and prefetched block is placed into instruction stream buffer
- Data Prefetching
  - Pentium 4 can prefetch data into L2 cache from up to 8 streams from 8 different 4 KB pages
  - Prefetching invoked if 2 successive L2 cache misses to a page, if distance between those cache blocks is < 256 bytes



11. Reducing Misses by Software Prefetching Data

- Data Prefetch
  - Load data into register (HP PA-RISC loads)
  - Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
  - Special prefetching instructions cannot cause faults; a form of speculative execution
- Issuing Prefetch Instructions takes time
  - Is cost of prefetch issues < savings in reduced misses?
  - Higher superscalar reduces difficulty of issue bandwidth

Hardware Prefetching of Instructions and Data (1)

- Nonblocking caches effectively reduce the miss penalty by overlapping execution with memory access
- To have value, we need a processor that can allow instructions to execute out-of-order
- Another approach is to prefetch items before they are requested by the processor
- Both instructions and data can be prefetched, either directly into the caches or into an external buffer that can be more quickly accessed than main memory

Hardware Prefetching of Instructions and Data (2)

- The processor fetches two blocks on a miss: the requested block and the next consecutive block
- The requested block is placed in the instruction cache when it returns, and the prefetched block is placed into the instruction stream buffer
- If the requested block is present in the instruction stream buffer, the original cache request is canceled, the block is read from the stream buffer, and the next prefetch request is issued
- There could be multiple stream buffers beyond the data cache, each prefetching at different addresses

Hardware Prefetching of Instructions and Data (3)

- A single instruction stream buffer would catch 15% to 25% of the misses from a 4-KB direct-mapped instruction cache with 16-byte blocks
- With 4 blocks in the instruction stream buffer the hit rate improves to about 50%, and with 16 blocks to 72%

Hardware Prefetching of Instructions and Data (4)

- A similar approach can be applied to data accesses
  - A single data stream buffer caught about 25% of the misses from the 4-KB direct-mapped cache
  - Instead of having a single stream, there could be multiple stream buffers beyond the data cache, each prefetching at different addresses
  - Four data stream buffers increased the data hit rate to 43%
  - For scientific programs eight stream buffers could capture 50% to 70% of all misses from a processor with two 64-KB four-way set-associative caches, one for instructions and the other for data

Hardware Prefetching of Instructions and Data (5)

- The UltraSPARC III uses such a prefetch scheme
- A prefetch cache remembers the address used to prefetch the data
- If a load hits in prefetch cache, the block is read from the prefetch cache, and the next prefetch request is issued
- It calculates the “stride” of the next prefetched block using the difference between current address and the previous address
- There can be up to eight simultaneous prefetches in UltraSPARC III

Example

- What is the effective miss rate of the UltraSPARC III using instruction prefetching? How much bigger an data cache would be needed in the UltraSPARC III to match the average access time if prefetching were removed? It has an 64-KB data cache. Assume prefetching reduces data miss rate by 20%.

Answer (1)

- We assume it takes 1 extra clock cycle if the data misses the cache but is found in the prefetch buffer
- Here is our revised formula:

  Average memory access time$_{\text{prefetch}}$ = Hit time + Miss rate × Prefetch hit rate × 1 + Miss rate × (1– Prefetch hit rate) × Miss penalty

- Let's assume the prefetch hit rate is 50%. The number of misses per 1000 instructions for an 64-KB data cache is 36.9
- To convert to a miss rate, is we assume 22% data references, the rate is

  36.9/ (1000 x 22 /100) = 16.7%

Answer (2)

- Assume the he hit time is 1 clock cycles, and the miss penalty is 15 clock cycles since UltraSPARC III has an L2 cache:
- Average memory access time$_{\text{prefetch}}$ = 1 + (16.7% × 20% × 1) + (16.7% × (1 – 20%) × 15) = 1 + 0.034 + 2.013 = 3.046
- To find the effective miss rate with the equivalent performance, we start with the original formula and solve for the miss rate:
- Average memory access time = Hit time + Miss rate × Miss penalty

Answer (3)

- Average memory access time = Hit time + Miss rate x Miss penalty
- Thus, miss rate = 13.6%
- Our calculation suggests that the effective miss rate of prefetching with an 64-KB cache is 13.6%
- The number of misses per 1000 instructions of a 256-KB instruction cache is 32.6, yielding a miss rate of 32.6/(22% x1000) or 14.8%
- If the prefetching reduces miss rate by 20%, then a 64 KB data cache with prefetching outperforms a 256-KB cache without it

Summary for Hardware Prefetching

- Prefetching relies on utilizing memory bandwidth that otherwise would be unused
- But if it interferes with demand misses it can actually lower performance
- Help from compilers can reduce useless prefetching

Compiler-Controlled Prefetching (1)

- An alternative to hardware prefetching is for the compiler to insert prefetch instructions to request the data before they are needed
- Two flavors of prefetch:
  - Register prefetch will load the value into a register
  - Cache prefetch loads data only into the cache and not the register

Compiler-Controlled Prefetching (2)

- Either of these can be *faulting* or *nonfaulting* (*nonbinding prefetch*)
  - —That is, the address does or does not cause an exception for virtual address faults and protection violations
  - —Nonfaulting prefetches simply turn into no-ops if they would normally result in an exception
- The most effective prefetch is *semantically invisible* to a program:
  - —It doesn't change the contents of registers and memory and it cannot cause virtual memory faults

Compiler-Controlled Prefetching (3)

- Prefetching makes sense only if the processor can proceed while the prefetched data are being fetched
    - That is, the caches do not stall but continue to supply instructions and data while waiting for the prefetched data to return
- Like hardware-controlled prefetching, the goal is to overlap execution with the prefetching of data
- Loops are the important targets, as they lend themselves to prefetch optimizations
    - If the miss penalty is small, the compiler just unrolls the loop once or twice and it schedules the prefetches with the execution
    - If the miss penalty is large, it uses software pipelining or unrolls many times to prefetch data for a future iteration

Compiler-Controlled Prefetching (4)

- Issuing prefetch instructions incurs an instruction overhead, however, so care must be taken to ensure that such overheads do not exceed the benefits
- By concentrating on references that are likely to be cache misses, programs can avoid unnecessary prefetches while improving average memory access time significantly

Summary of Reducing Cache Miss Penalty/ Miss Rate via Parallelism

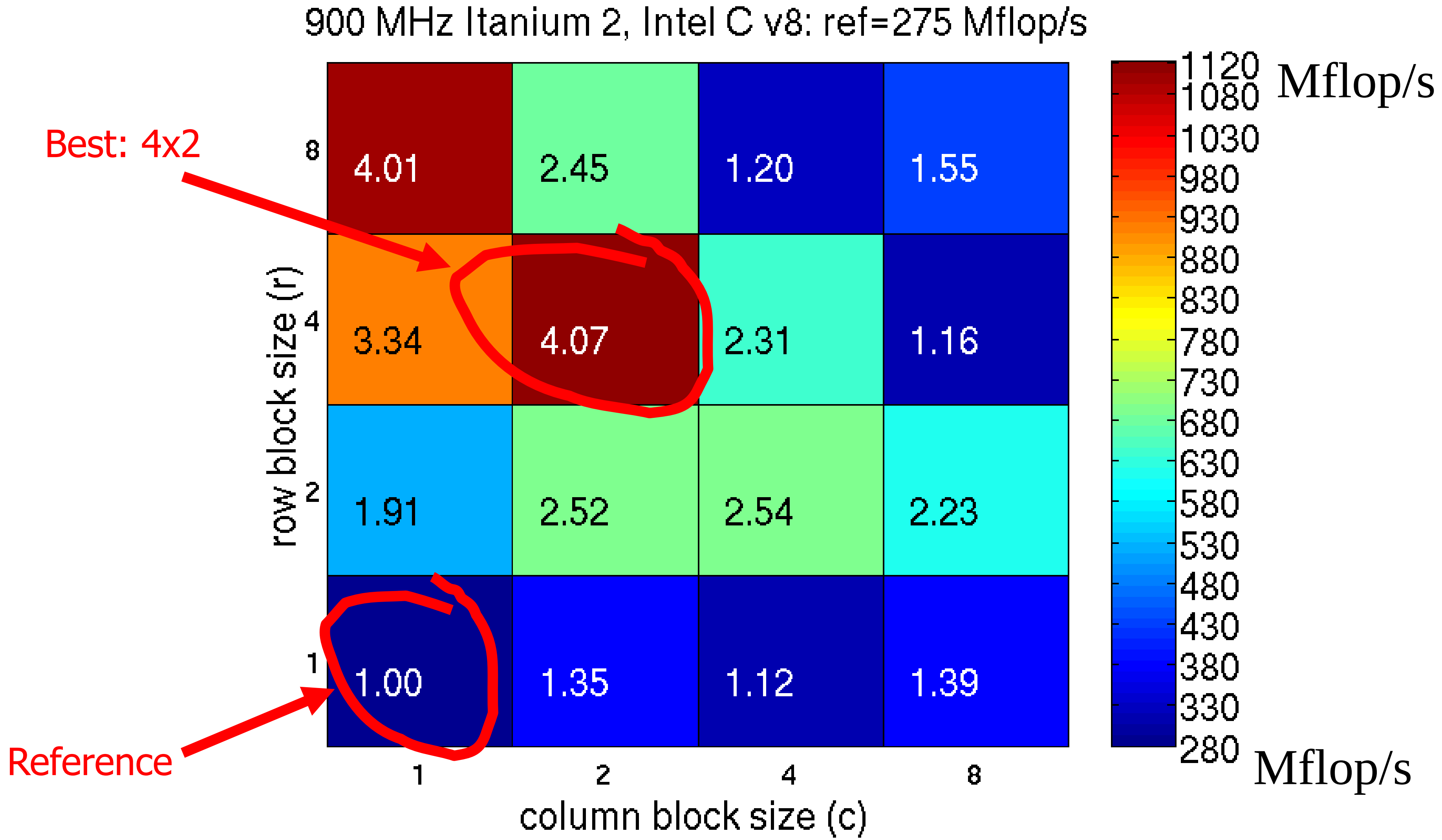
- Non-blocking caches enables out-of-order processors
  - In general such processors cache hide misses to L1 caches that hit in the L2 cache, but not a complete L2 cache miss
- However, if miss under miss is supported, nonblocking caches can utilize more bandwidth behind the cache with several outstanding misses operating at once for programs with sufficient ILP
- The HW/SW prefetching techniques leverage excess memory bandwidth for performance by using a cache
- The potential success of prefetching is either lower miss penalty, or if they are started far in advance of need, reduction of the miss rate

Compiler Optimization vs. Memory Hierarchy Search

- Compiler tries to figure out memory hierarchy optimizations
- New approach: "Auto-tuners" 1st run variations of program on computer to find best combinations of optimizations (blocking, padding, …) and algorithms, then produce C code to be compiled for *that* computer
- "Auto-tuner" targeted to numerical method
  - E.g., PHiPAC (BLAS), Atlas (BLAS),
    Sparsity (Sparse linear algebra), Spiral (DSP), FFT-W

Sparse Matrix – Search for Blocking

for finite element problem [Im, Yelick, Vuduc, 2005]



	1	2	4	8
8	4.01	2.45	1.20	1.55
4	3.34	4.07	2.31	1.16
2	1.91	2.52	2.54	2.23
1	1.00	1.35	1.12	1.39

Best Sparse Blocking for 8 Computers

row block size (r)				
8		Intel Pentium M		Sun Ultra 2, Sun Ultra 3, AMD Opteron
4	IBM Power 4, Intel/HP Itanium	Intel/HP Itanium 2	IBM Power 3	
2				
1				
	1	2	4	8

column block size (c)

- All possible column block sizes selected for 8 computers; How could compiler know?

Technique	Hit Time	Bandwidth	Miss penalty	Miss rate	HW cost/ complexity	Comment
Small and simple caches	+			–	0	Trivial; widely used
Way-predicting caches	+				1	Used in Pentium 4
Trace caches	+				3	Used in Pentium 4
Pipelined cache access	–	+			1	Widely used
Nonblocking caches		+	+		3	Widely used
Banked caches		+			1	Used in L2 of Opteron and Niagara
Critical word first and early restart			+		2	Widely used
Merging write buffer			+		1	Widely used with write through
Compiler techniques to reduce cache misses				+	0	Software is a challenge; some computers have compiler option
Hardware prefetching of instructions and data			+	+	2 instr., 3 data	Many prefetch instructions; AMD Opteron prefetches data
Compiler-controlled prefetching			+	+	3	Needs nonblocking cache; in many CPUs

Main Memory Background

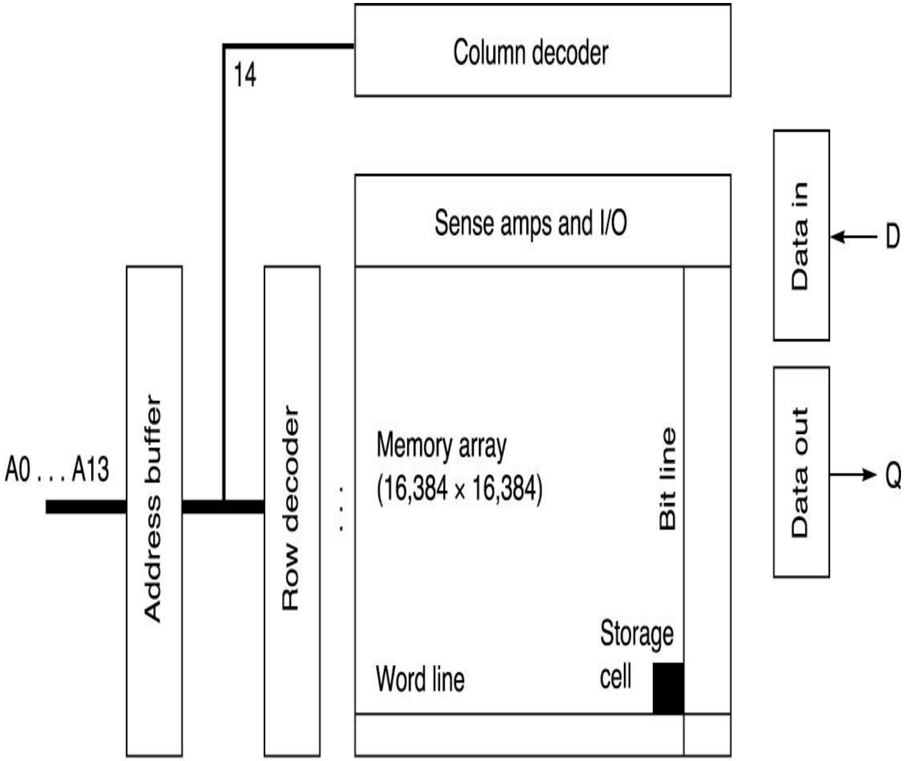
- Performance of Main Memory:
  - Latency: Cache Miss Penalty
    - *Access Time*: time between request and word arrives
    - *Cycle Time*: time between requests
  - Bandwidth: I/O & Large Block Miss Penalty (L2)
- Main Memory is *DRAM*: Dynamic Random Access Memory
  - Dynamic since needs to be refreshed periodically (8 ms, 1% time)
  - Addresses divided into 2 halves (Memory as a 2D matrix):
    - *RAS* or *Row Access Strobe*
    - *CAS* or *Column Access Strobe*
- Cache uses *SRAM*: Static Random Access Memory
  - No refresh (6 transistors/bit vs. 1 transistor
    Size: DRAM/SRAM - *4-8*,
    Cost/Cycle time: SRAM/DRAM - *8-16*

DRAM Technology

- The main memory of computers sold since 1975 is composed of DRAMs
- As early DRAMs grew in capacity, the cost of a package with all the necessary address lines was an issue
- The solution was to multiplex the address lines, thereby cutting the number of address pins in half

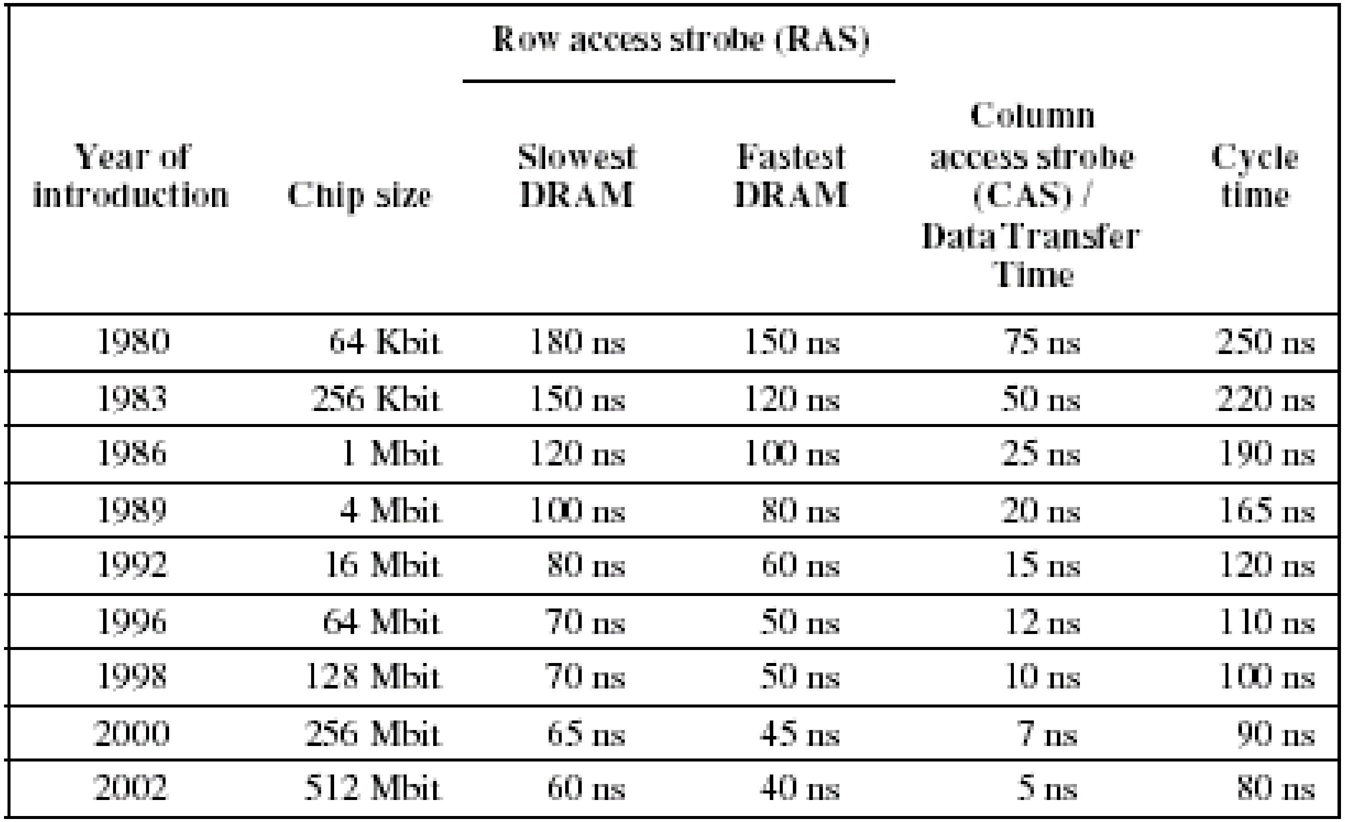
DRAM Technology (Cont'd)

- The memory is organized as a rectangular matrix addressed by rows and columns
- One half of the address is sent first, called the row access strobe or RAS
- It is followed by the other half of the address, sent during the column access strobe or CAS



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

Times of Fast and Slow DRAMs



		Row access strobe (RAS)			
Year of introduction	Chip size	Slowest DRAM	Fastest DRAM	Column access strobe (CAS) / Data Transfer Time	Cycle time
1980	64 Kbit	180 ns	150 ns	75 ns	250 ns
1983	256 Kbit	150 ns	120 ns	50 ns	220 ns
1986	1 Mbit	120 ns	100 ns	25 ns	190 ns
1989	4 Mbit	100 ns	80 ns	20 ns	165 ns
1992	16 Mbit	80 ns	60 ns	15 ns	120 ns
1996	64 Mbit	70 ns	50 ns	12 ns	110 ns
1998	128 Mbit	70 ns	50 ns	10 ns	100 ns
2000	256 Mbit	65 ns	45 ns	7 ns	90 ns
2002	512 Mbit	60 ns	40 ns	5 ns	80 ns

SRAM Technology

- SRAMs typically use six transistors per bit to prevent the information from being disturbed when read
- SRAM needs only minimal power to retain the charge in standby mode
- DRAMs must continue to be refreshed occasionally so as to not lose information
- SRAM designs are concerned with speed and capacity
- There is no difference between access time and cycle time

Embedded Processor Memory Technology: ROM and Flash

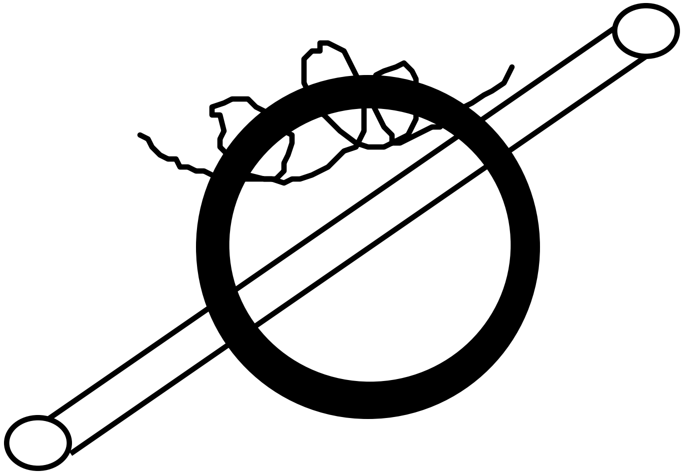
- Two memory technologies are found in embedded computers
  - —Read-Only Memory (ROM)
  - —Flash memory

Improving Memory Performance in a standard DRAM Chip

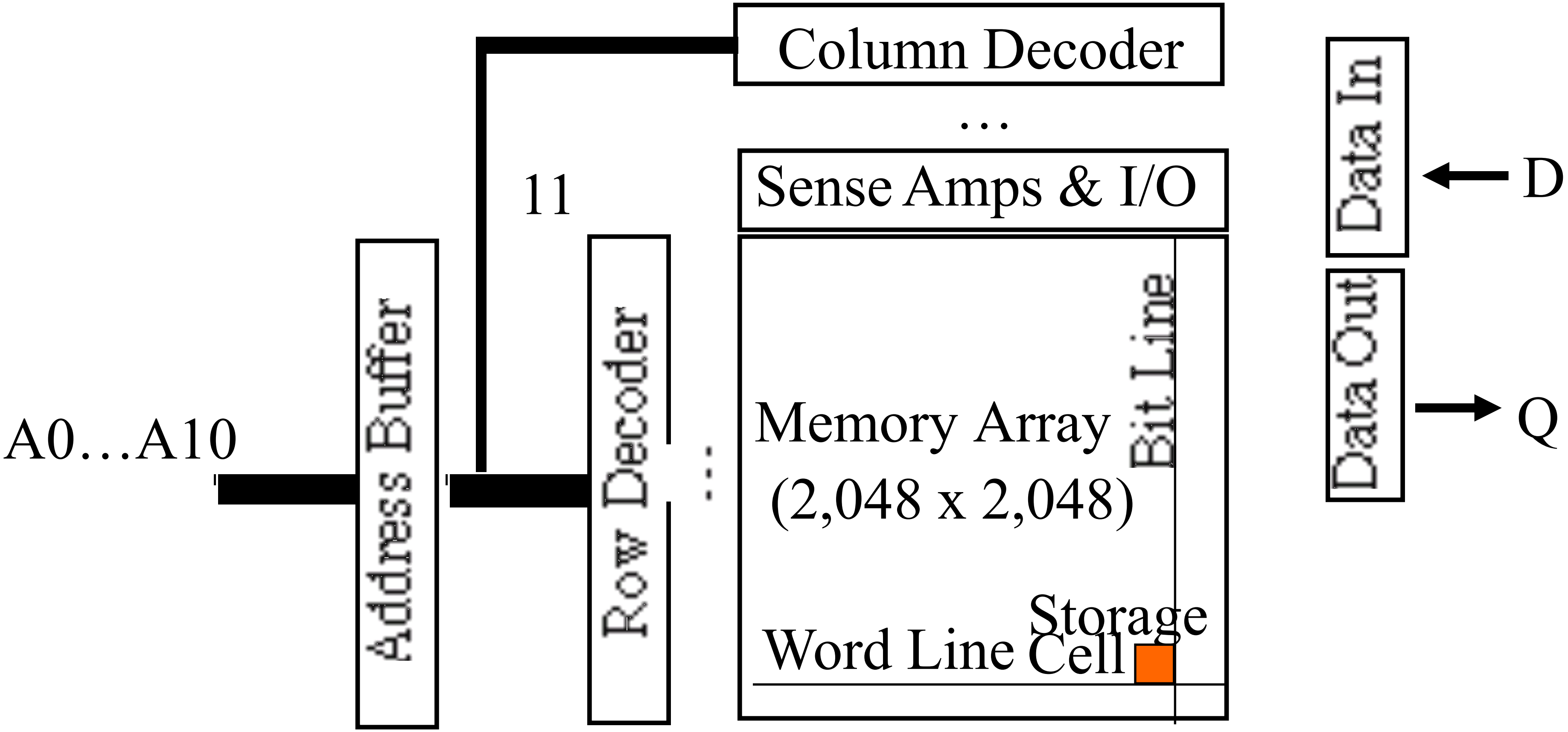
- The internal organization of DRAM actually consists of many memory modules
- These modules are usually 1 to 4 megabits
- There have been a variety of evolutionary innovations to improve bandwidth over time
    - —Fast page mode
    - —SDRAM
    - —DDR

Main Memory Deep Background

- "Out-of-Core", "In-Core," "Core Dump"?
- "Core memory"?
- Non-volatile, magnetic
- Lost to 4 Kbit DRAM (today using 512Mbit DRAM)
- Access time 750 ns, cycle time 1500-3000 ns



DRAM logical organization (4 Mbit)



- Square root of bits per RAS/CAS

Quest for DRAM Performance

1. Fast Page mode
   - Add timing signals that allow repeated accesses to row buffer without another row access time
   - Such a buffer comes naturally, as each array will buffer 1024 to 2048 bits for each access
2. Synchronous DRAM (SDRAM)
   - Add a clock signal to DRAM interface, so that the repeated transfers would not bear overhead to synchronize with DRAM controller
3. Double Data Rate (DDR SDRAM)
   - Transfer data on both the rising edge and falling edge of the DRAM clock signal ⇒ doubling the peak data rate
   - DDR2 lowers power by dropping the voltage from 2.5 to 1.8 volts + offers higher clock rates: up to 400 MHz
   - DDR3 drops to 1.5 volts + higher clock rates: up to 800 MHz

- Improved Bandwidth, not Latency

DRAM name based on Peak Chip Transfers/Sec
DIMM name based on Peak DIMM Mbytes/Sec

Standard	Clock Rate (MHz)	M transfers / second	DRAM Name	Mbytes/s/ DIMM	DIMM Name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10664	PC10700
DDR3	800	1600	DDR3-1600	12800	PC12800

Fastest for sale 4/06 ($125/GB)

x 2

x 8

Need for Error Correction!

- Motivation:
  - Failures/time *proportional* to number of bits!
  - As DRAM cells shrink, more vulnerable
- Went through period in which failure rate was low enough without error correction that people didn’t do correction
  - DRAM banks too large now
  - Servers always corrected memory systems
- Basic idea: add redundancy through parity bits
  - Common configuration: Random error correction
    - SEC-DED (single error correct, double error detect)
    - One example: 64 data bits + 8 parity bits (11% overhead)
  - Really want to handle failures of physical components as well
    - Organization is multiple DRAMs/DIMM, multiple DIMMs
    - Want to recover from failed DRAM and failed DIMM!
    - “Chip kill” handle failures width of single DRAM chip

Introduction to Virtual Machines

- VMs developed in late 1960s
  - Remained important in mainframe computing over the years
  - Largely ignored in single user computers of 1980s and 1990s
- Recently regained popularity due to
  - increasing importance of isolation and security in modern systems,
  - failures in security and reliability of standard operating systems,
  - sharing of a single computer among many unrelated users,
  - and the dramatic increases in raw speed of processors, which makes the overhead of VMs more acceptable

What is a Virtual Machine (VM)?

- Broadest definition includes all emulation methods that provide a standard software interface, such as the Java VM
- "(Operating) System Virtual Machines" provide a complete system level environment at binary ISA
  - Here assume ISAs always match the native hardware ISA
  - E.g., IBM VM/370, VMware ESX Server, and Xen
- Present illusion that VM users have entire computer to themselves, including a copy of OS
- Single computer runs multiple VMs, and can support a multiple, different OSes
  - On conventional platform, single OS "owns" all HW resources
  - With a VM, multiple OSes all share HW resources
- Underlying HW platform is called the host, and its resources are shared among the guest VMs

Virtual Machine Monitors (VMMs)

- Virtual machine monitor (VMM) or hypervisor is software that supports VMs
- VMM determines how to map virtual resources to physical resources
- Physical resource may be time-shared, partitioned, or emulated in software
- VMM is much smaller than a traditional OS;
  - —isolation portion of a VMM is ≈ 10,000 lines of code

VMM Overhead?

- Depends on the workload
- User-level processor-bound programs (e.g., SPEC) have zero-virtualization overhead
  - Runs at native speeds since OS rarely invoked
- I/O-intensive workloads ⇒ OS-intensive
  ⇒ execute many system calls and privileged instructions
  ⇒ can result in high virtualization overhead
  - For System VMs, goal of architecture and VMM is to run almost all instructions directly on native hardware
- If I/O-intensive workload is also I/O-bound
  ⇒ low processor utilization since waiting for I/O
  ⇒ processor virtualization can be hidden
  ⇒ low virtualization overhead

Other Uses of VMs

- Focus here on protection
- 2 Other commercially important uses of VMs

1. Managing Software
   - VMs provide an abstraction that can run the complete SW stack, even including old OSes like DOS
   - Typical deployment: some VMs running legacy OSes, many running current stable OS release, few testing next OS release
2. Managing Hardware
   - VMs allow separate SW stacks to run independently yet share HW, thereby consolidating number of servers
     - Some run each application with compatible version of OS on separate computers, as separation helps dependability
   - Migrate running VM to a different computer
     - Either to balance load or to evacuate from failing HW

Requirements of a Virtual Machine Monitor

- A VM Monitor
  - Presents a SW interface to guest software,
  - Isolates state of guests from each other, and
  - Protects itself from guest software (including guest OSes)
- Guest software should behave on a VM exactly as if running on the native HW
  - Except for performance-related behavior or limitations of fixed resources shared by multiple VMs
- Guest software should not be able to change allocation of real system resources directly
- Hence, VMM must control ≈ everything even though guest VM and OS currently running is temporarily using them
  - Access to privileged state, Address translation, I/O, Exceptions and Interrupts, …

Requirements of a Virtual Machine Monitor

- VMM must be at higher privilege level than guest VM, which generally run in user mode
  - ⇒ Execution of privileged instructions handled by VMM
- E.g., Timer interrupt: VMM suspends currently running guest VM, saves its state, handles interrupt, determine which guest VM to run next, and then load its state
  - — Guest VMs that rely on timer interrupt provided with virtual timer and an emulated timer interrupt by VMM
- Requirements of system virtual machines are ≈ same as paged-virtual memory:

1. At least 2 processor modes, system and user
2. Privileged subset of instructions available only in system mode, trap if executed in user mode
   - — All system resources controllable only via these instructions

ISA Support for Virtual Machines

- If VMs are planned for during design of ISA, easy to reduce instructions that must be executed by a VMM and how long it takes to emulate them
  - Since VMs have been considered for desktop/PC server apps only recently, most ISAs were created without virtualization in mind, including 80x86 and most RISC architectures
- VMM must ensure that guest system only interacts with virtual resources ⇒ conventional guest OS runs as user mode program on top of VMM
  - If guest OS attempts to access or modify information related to HW resources via a privileged instruction--for example, reading or writing the page table pointer--it will trap to the VMM
- If not, VMM must intercept instruction and support a virtual version of the sensitive information as the guest OS expects (examples soon)

Impact of VMs on Virtual Memory

- Virtualization of virtual memory if each guest OS in every VM manages its own set of page tables?
- VMM separates real and physical memory
  - Makes real memory a separate, intermediate level between virtual memory and physical memory
  - Some use the terms virtual memory, physical memory, and machine memory to name the 3 levels
  - Guest OS maps virtual memory to real memory via its page tables, and VMM page tables map real memory to physical memory
- VMM maintains a shadow page table that maps directly from the guest virtual address space to the physical address space of HW
  - Rather than pay extra level of indirection on every memory access
  - VMM must trap any attempt by guest OS to change its page table or to access the page table pointer

ISA Support for VMs & Virtual Memory

- IBM 370 architecture added additional level of indirection that is managed by the VMM
  - —Guest OS keeps its page tables as before, so the shadow pages are unnecessary
- To virtualize software TLB, VMM manages the real TLB and has a copy of the contents of the TLB of each guest VM
  - —Any instruction that accesses the TLB must trap
  - —TLBs with Process ID tags support a mix of entries from different VMs and the VMM, thereby avoiding flushing of the TLB on a VM switch

Impact of I/O on Virtual Memory

- Most difficult part of virtualization
  - Increasing number of I/O devices attached to the computer
  - Increasing diversity of I/O device types
  - Sharing of a real device among multiple VMs,
  - Supporting the myriad of device drivers that are required, especially if different guest OSes are supported on the same VM system
- Give each VM generic versions of each type of I/O device driver, and let VMM to handle real I/O
- Method for mapping virtual to physical I/O device depends on the type of device:
  - Disks partitioned by VMM to create virtual disks for guest VMs
  - Network interfaces shared between VMs in short time slices, and VMM tracks messages for virtual network addresses to ensure that guest VMs only receive their messages

Example: Xen VM

- Xen: Open-source System VMM for 80x86 ISA
  - Project started at University of Cambridge, GNU license model
- Original vision of VM is running unmodified OS
  - Significant wasted effort just to keep guest OS happy
- "paravirtualization" - small modifications to guest OS to simplify virtualization

3 Examples of paravirtualization in Xen:

1. To avoid flushing TLB when invoke VMM, Xen mapped into upper 64 MB of address space of each VM
2. Guest OS allowed to allocate pages, just check that didn't violate protection restrictions
3. To protect the guest OS from user programs in VM, Xen takes advantage of 4 protection levels available in 80x86
   - Most OSes for 80x86 keep everything at privilege levels 0 or at 3.
   - Xen VMM runs at the highest privilege level (0)
   - Guest OS runs at the next level (1)
   - Applications run at the lowest privilege level (3)

Xen changes for paravirtualization

- Port of Linux to Xen changed ≈ 3000 lines, or ≈ 1% of 80x86-specific code
  - Does not affect application-binary interfaces of guest OS
- OSes supported in Xen 2.0

OS	Runs as host OS	Runs as guest OS
Linux 2.4	Yes	Yes
Linux 2.6	Yes	Yes
NetBSD 2.0	No	Yes
NetBSD 3.0	Yes	Yes
Plan 9	No	Yes
FreeBSD 5	No	Yes

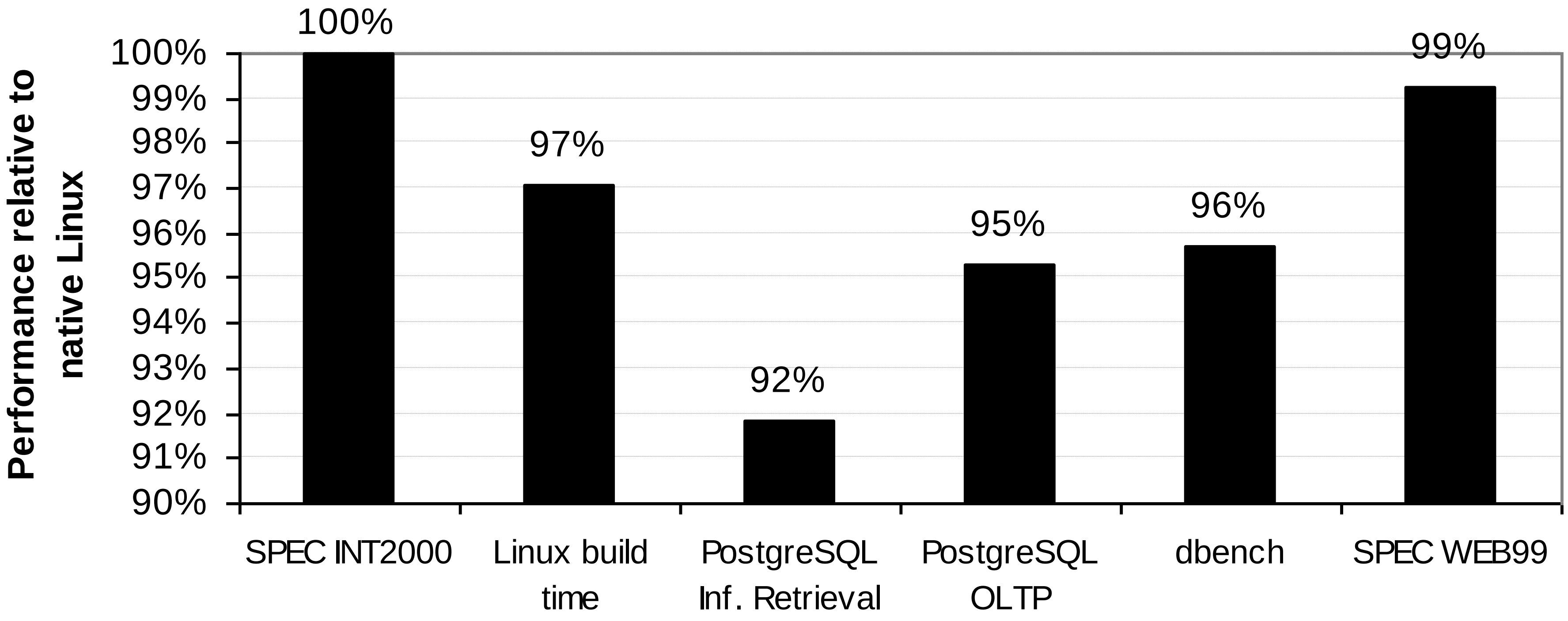
http://wiki.xensource.com/xenwiki/OSCompatibility

Xen and I/O

- To simplify I/O, privileged VMs assigned to each hardware I/O device: "driver domains"
  - Xen Jargon: "domains" = Virtual Machines
- Driver domains run physical device drivers, although interrupts still handled by VMM before being sent to appropriate driver domain
- Regular VMs ("guest domains") run simple virtual device drivers that communicate with physical devices drivers in driver domains over a channel to access physical I/O hardware
- Data sent between guest and driver domains by page remapping

Xen Performance

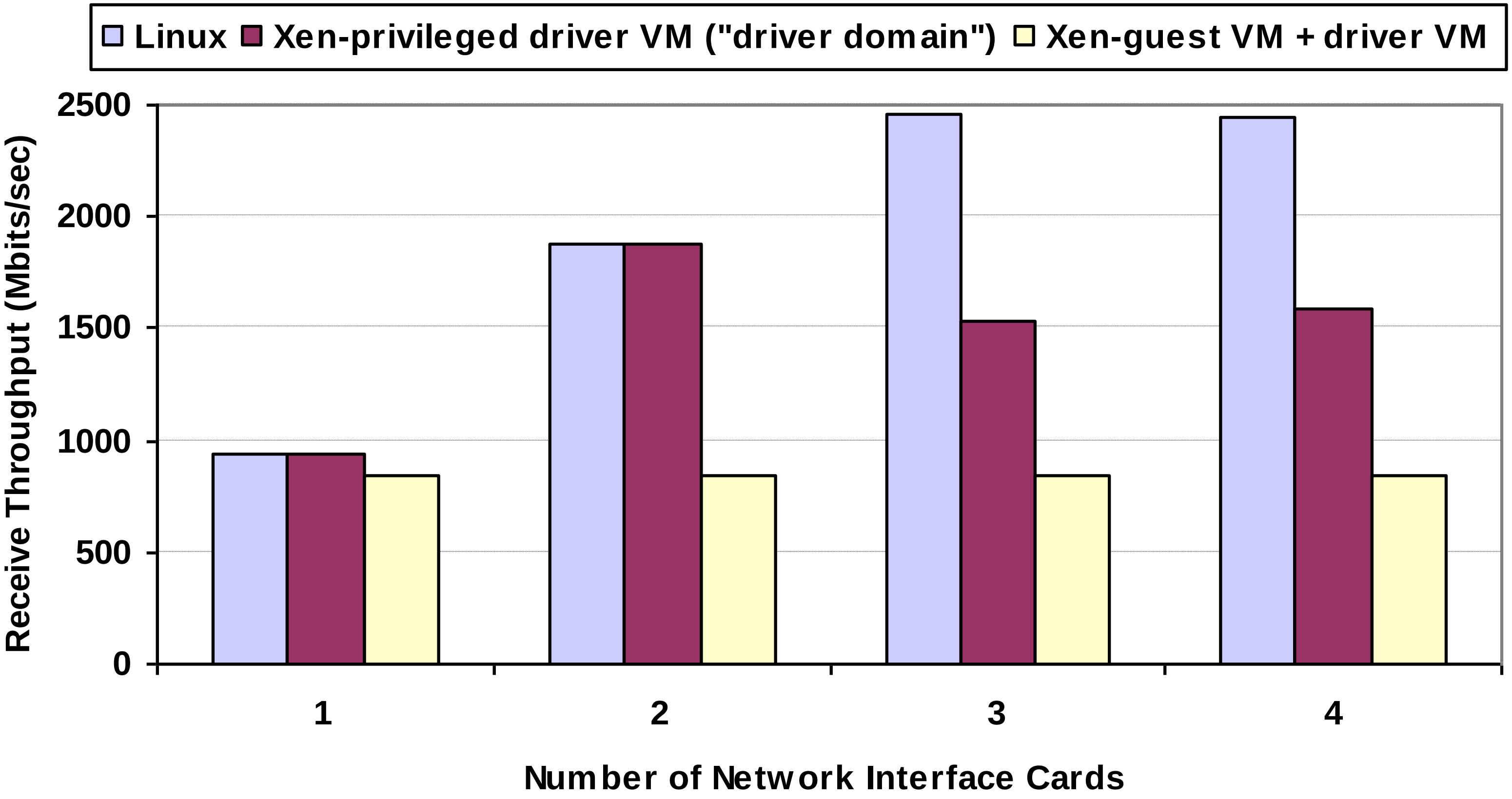
- Performance relative to native Linux for Xen for 6 benchmarks from Xen developers



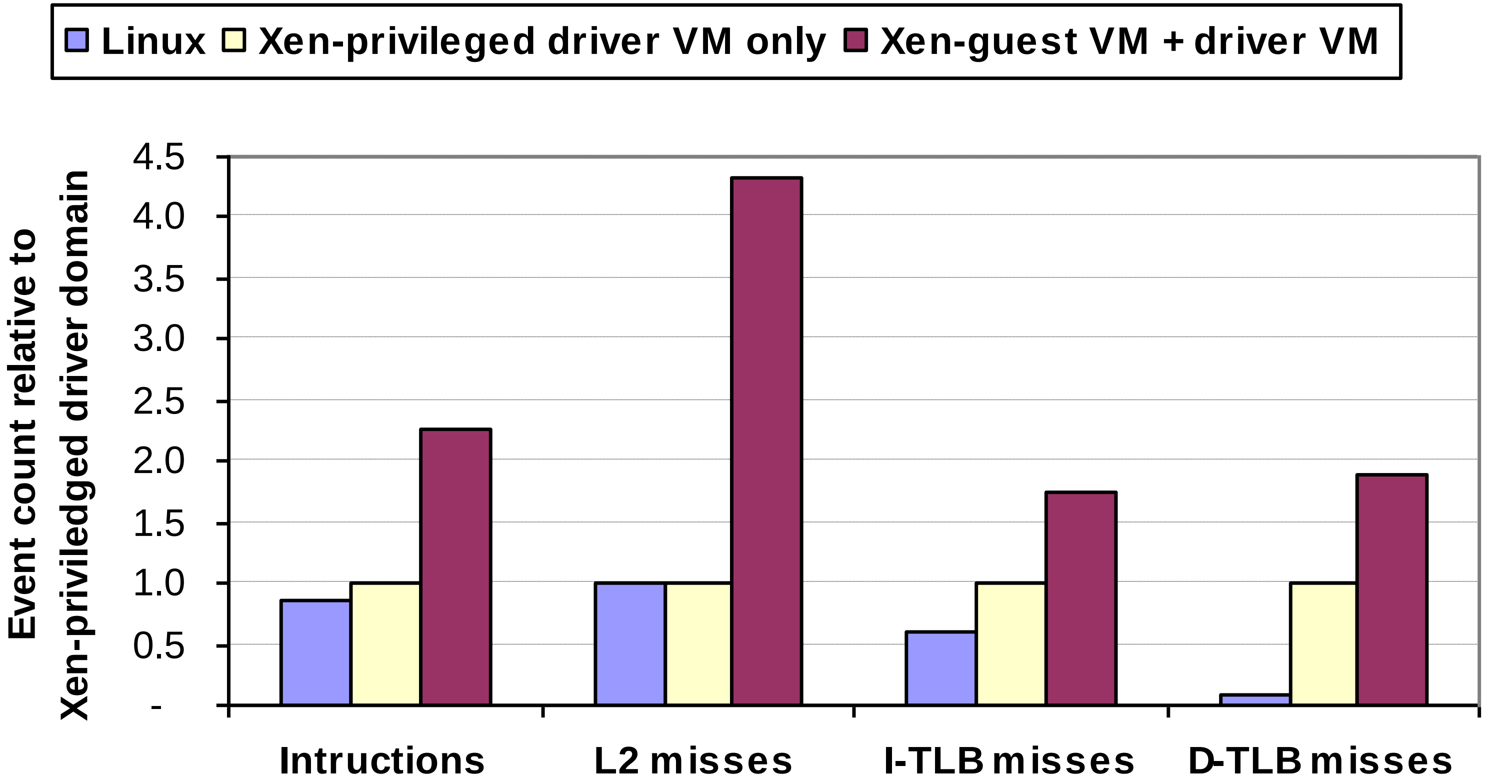
- Slide 40: User-level processor-bound programs? I/O-intensive workloads? I/O-Bound I/O-Intensive?

Xen Performance, Part II

- Subsequent study noticed Xen experiments based on 1 Ethernet network interfaces card (NIC), and single NIC was a performance bottleneck



Xen Performance, Part III



1. > 2X instructions for guest VM + driver VM
2. > 4X L2 cache misses
3. 12X – 24X Data TLB misses

Xen Performance, Part IV

1. > 2X instructions: page remapping and page transfer between driver and guest VMs and due to communication between the 2 VMs over a channel
2. 4X L2 cache misses: Linux uses zero-copy network interface that depends on ability of NIC to do DMA from different locations in memory
   - Since Xen does not support “gather DMA” in its virtual network interface, it can’t do true zero-copy in the guest VM
3. 12X – 24X Data TLB misses: 2 Linux optimizations
   - Superpages for part of Linux kernel space, and 4MB pages lowers TLB misses versus using 1024 4 KB pages. Not in Xen
   - PTEs marked global are not flushed on a context switch, and Linux uses them for its kernel space. Not in Xen

- Future Xen may address 2. and 3., but 1. inherent?

And in Conclusion [1/2] …

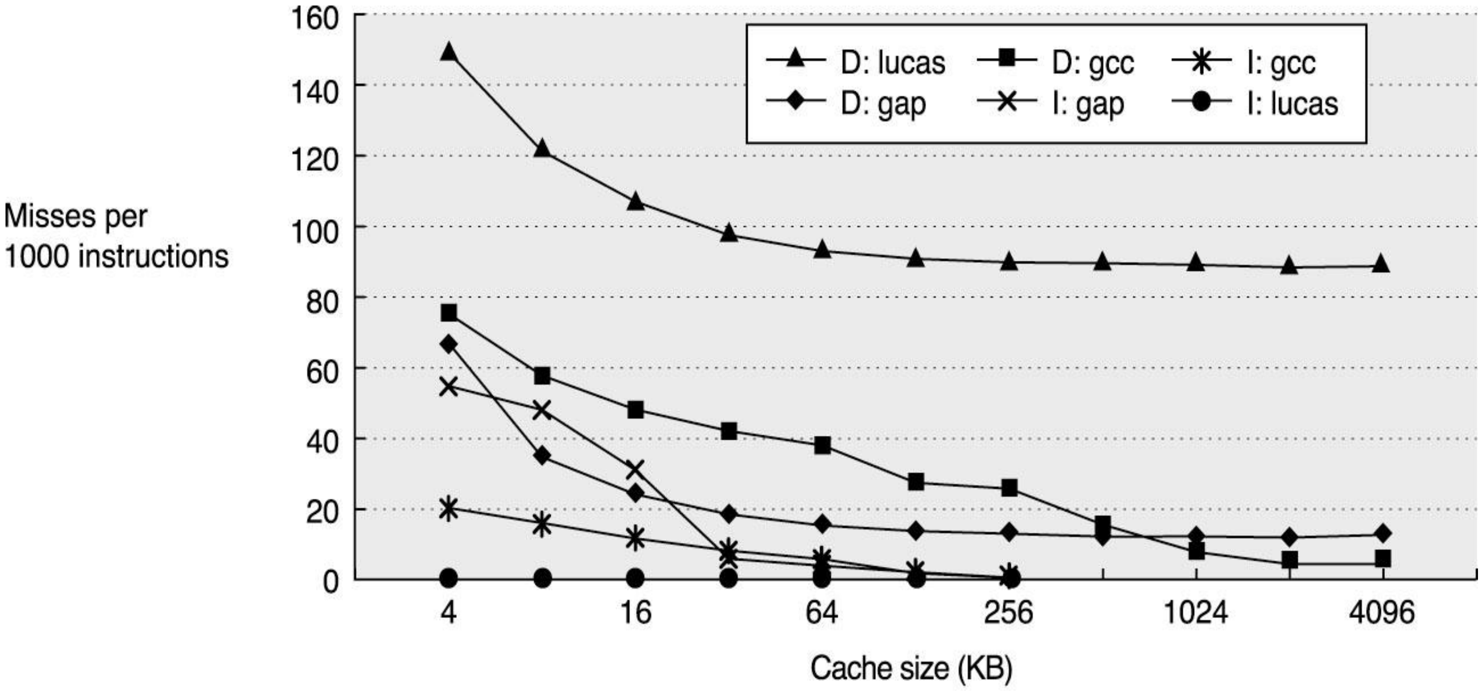
- Memory wall inspires optimizations since so much performance lost there
  - Reducing hit time: Small and simple caches, Way prediction, Trace caches
  - Increasing cache bandwidth: Pipelined caches, Multibanked caches, Nonblocking caches
  - Reducing Miss Penalty: Critical word first, Merging write buffers
  - Reducing Miss Rate: Compiler optimizations
  - Reducing miss penalty or miss rate via parallelism: Hardware prefetching, Compiler prefetching
- "Auto-tuners" search replacing static compilation to explore optimization space?
- DRAM – Continuing Bandwidth innovations: Fast page mode, Synchronous, Double Data Rate

And in Conclusion [2/2] ...

- VM Monitor presents a SW interface to guest software, isolates state of guests, and protects itself from guest software (including guest OSes)
- Virtual Machine Revival
  - Overcome security flaws of large OSes
  - Manage Software, Manage Hardware
  - Processor performance no longer highest priority
- Virtualization challenges for processor, virtual memory, and I/O
  - Paravirtualization to cope with those difficulties
- Xen as example VMM using paravirtualization
  - 2005 performance on non-I/O bound, I/O intensive apps: 80% of native Linux without driver VM, 34% with driver VM

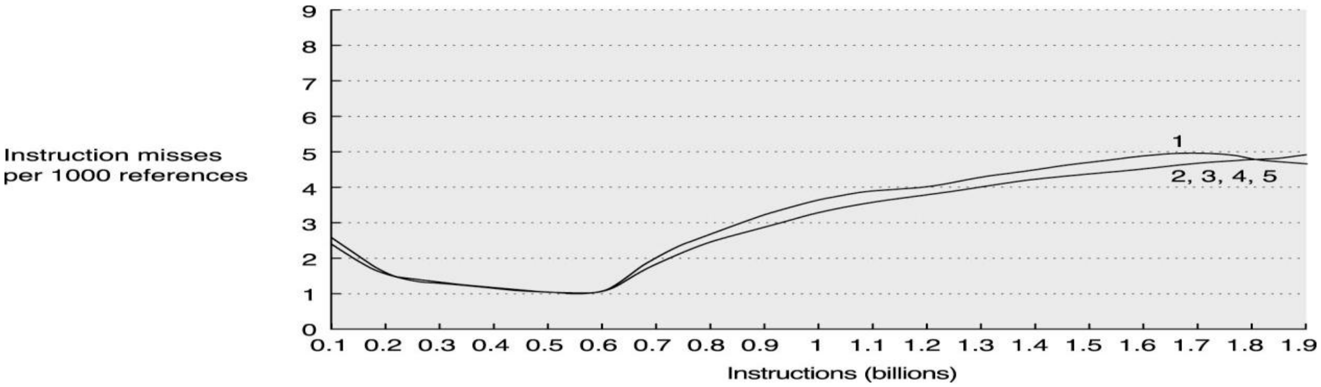
Fallacy 1

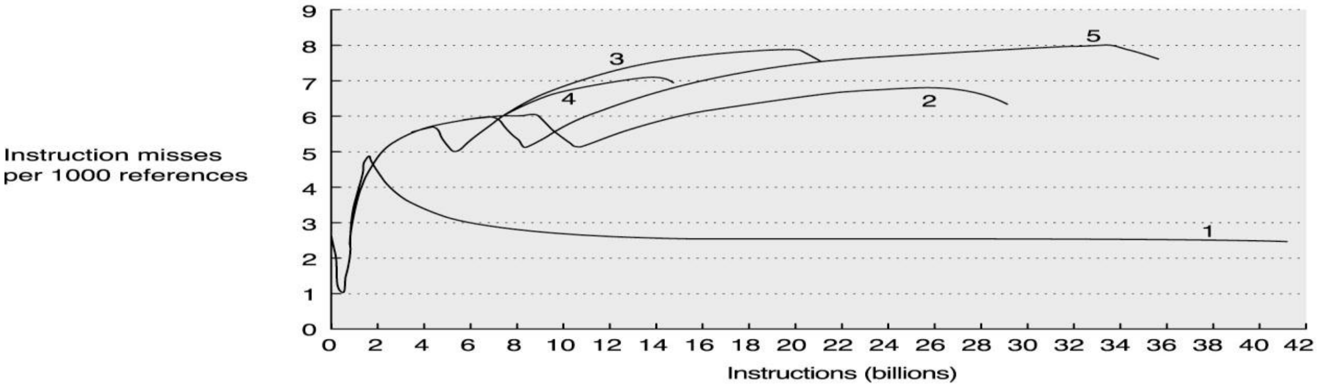
- Predicting cache performance of one program from another



Pitfall 1

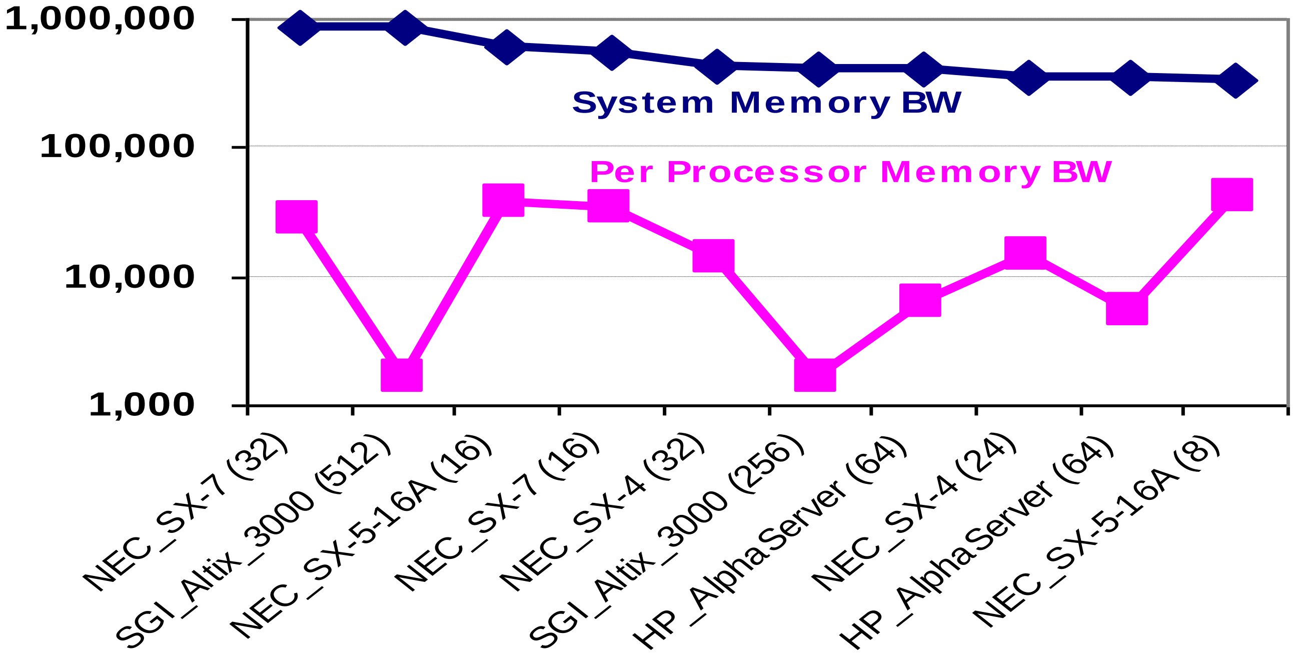
- Simulating enough instructions to get accurate performance measures of the memory hierarchy





Pitfall 2

- Not delivering high memory bandwidth in a cache-based system
- 10 Fastest computers at Stream benchmark
- Only 4/10 computers rely on data caches, and their memory BW per processor is 7X to 25X slower than NEC SX7


System Memory BW
Per Processor Memory BW
NEC_SX-7 (32)
SGI_Altix_3000 (512)
NEC_SX-5-16A (16)
NEC_SX-7 (16)
NEC_SX-4 (32)
SGI_Altix_3000 (256)
HP_AlphaServer (64)
NEC_SX-4 (24)
HP_AlphaServer (64)
NEC_SX-5-16A (8)