



# Master Informatics Eng.

2019/20

*A.J.Proença*

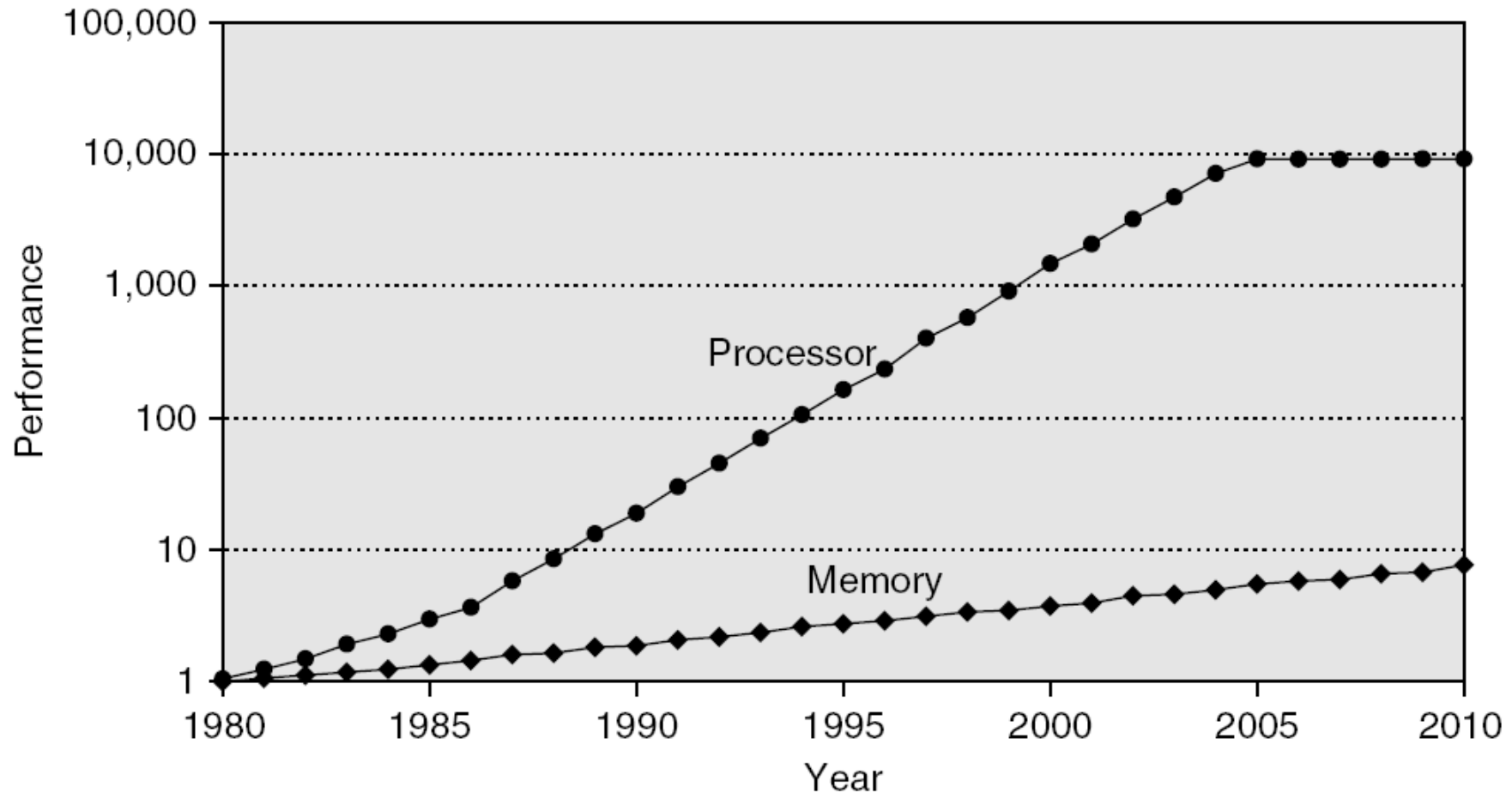
## Memory Hierarchy

*(most slides are borrowed)*

# Introduction

- Programmers want unlimited amounts of memory with low latency
- Fast memory technology is more expensive per bit than slower memory
- Solution: organize memory system into a hierarchy
  - Entire addressable memory space available in largest, slowest memory
  - Incrementally smaller and faster memories, each containing a subset of the memory below it, proceed in steps up toward the processor
- Temporal and spatial locality insures that nearly all references can be found in smaller memories
  - Gives the illusion of a large, fast memory being presented to the processor

# Memory Performance Gap



# Memory Hierarchy Design

- Memory hierarchy design becomes more crucial with recent multi-core processors:
  - Aggregate peak bandwidth grows with # cores:
    - Intel Core i7 can generate two references per core per clock
    - Four cores and 3.2 GHz clock
      - 25.6 billion\* 64-bit data references/second +
      - 12.8 billion\* 128-bit instruction references
      - = 409.6 GB/s!
  - DRAM bandwidth is only 6% of this (25 GB/s)
  - Requires:
    - Multi-port, pipelined caches
    - Two levels of cache per core
    - Shared third-level cache on chip

\* US billion =  $10^9$

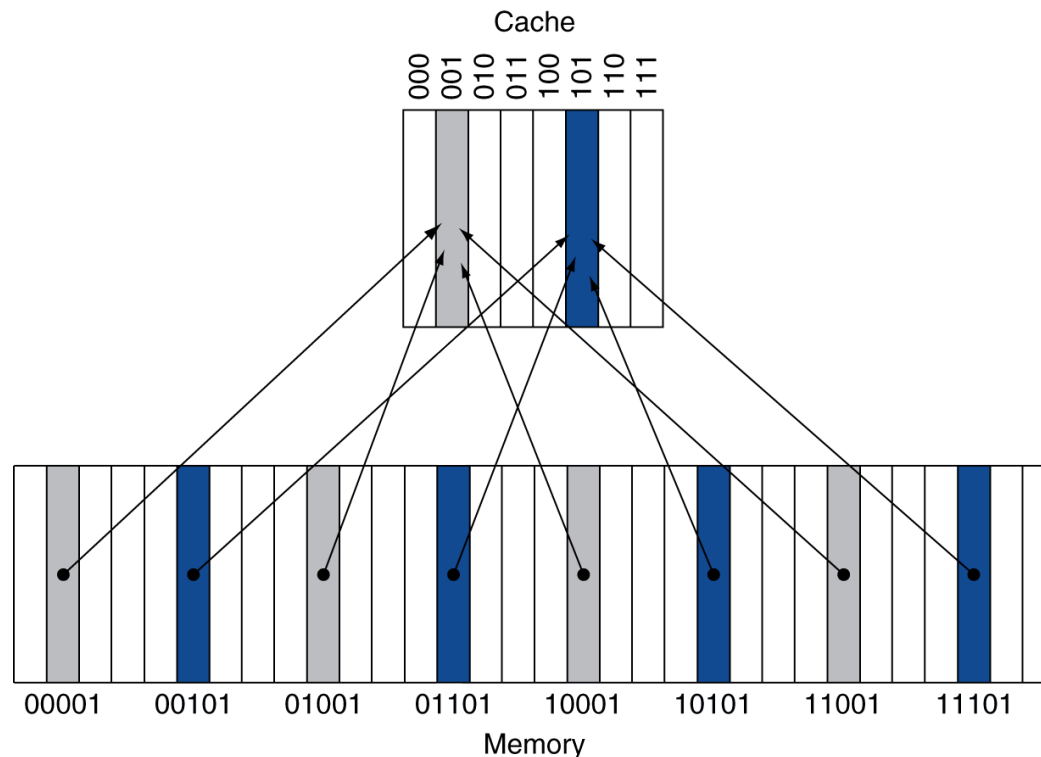
# The Memory Hierarchy

## The BIG Picture

- Common principles apply at all levels of the memory hierarchy
  - Based on notions of caching
- At each level in the hierarchy
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy

# Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)



- #Blocks is a power of 2
- Use low-order address bits

# Associative Caches

- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- $n$ -way set associative
  - Each set contains  $n$  entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - $n$  comparators (less expensive)

# How Much Associativity

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%



# Block Placement

- Determined by associativity
  - Direct mapped (1-way associative)
    - One choice for placement
  - n-way set associative
    - n choices within a set
  - Fully associative
    - Any location
- Higher associativity reduces miss rate
  - Increases complexity, cost, and access time

# Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity

# Write Policy

- Write-through
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer
- Write-back
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state
- Virtual memory
  - Only write-back is feasible, given disk write latency

# Memory Hierarchy Basics

- $n$  sets  $\Rightarrow$   $n$ -way set associative
  - *Direct-mapped cache*  $\Rightarrow$  one block per set
  - *Fully associative*  $\Rightarrow$  one set
- Writing to cache: two strategies
  - *Write-through*
    - Immediately update lower levels of hierarchy
  - *Write-back*
    - Only update lower levels of hierarchy when an updated block is replaced
  - Both strategies use *write buffer* to make writes asynchronous

# Memory Hierarchy Basics

$$\text{CPU}_{\text{exec-time}} = (\text{CPU}_{\text{clock-cycles}} + \text{Mem}_{\text{stall-cycles}}) \times \text{Clock cycle time}$$

$$\text{CPU}_{\text{exec-time}} = (\text{IC} \times \text{CPI}_{\text{CPU}} + \text{Mem}_{\text{stall-cycles}}) \times \text{Clock cycle time}$$

$$\text{Mem}_{\text{stall-cycles}} = \text{IC} \times \dots \text{Miss rate} \dots \text{Mem accesses} \dots \text{Miss penalty} \dots$$

# Memory Hierarchy Basics

$$\text{CPU}_{\text{exec-time}} = (\text{CPU}_{\text{clock-cycles}} + \text{Mem}_{\text{stall-cycles}}) \times \text{Clock cycle time}$$

$$\text{Mem}_{\text{stall-cycles}} = \text{IC} \times \text{Misses/Instruction} \times \text{Miss Penalty}$$

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

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

- Note1: miss rate/penalty are often different for reads and writes
- Note2: speculative and multithreaded processors may execute other instructions during a miss
  - Reduces performance impact of misses

# Cache Performance Example

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache:
  - D-cache:
- Actual CPI =  $2 + ?? + ?? = ??$

# Cache Performance Example

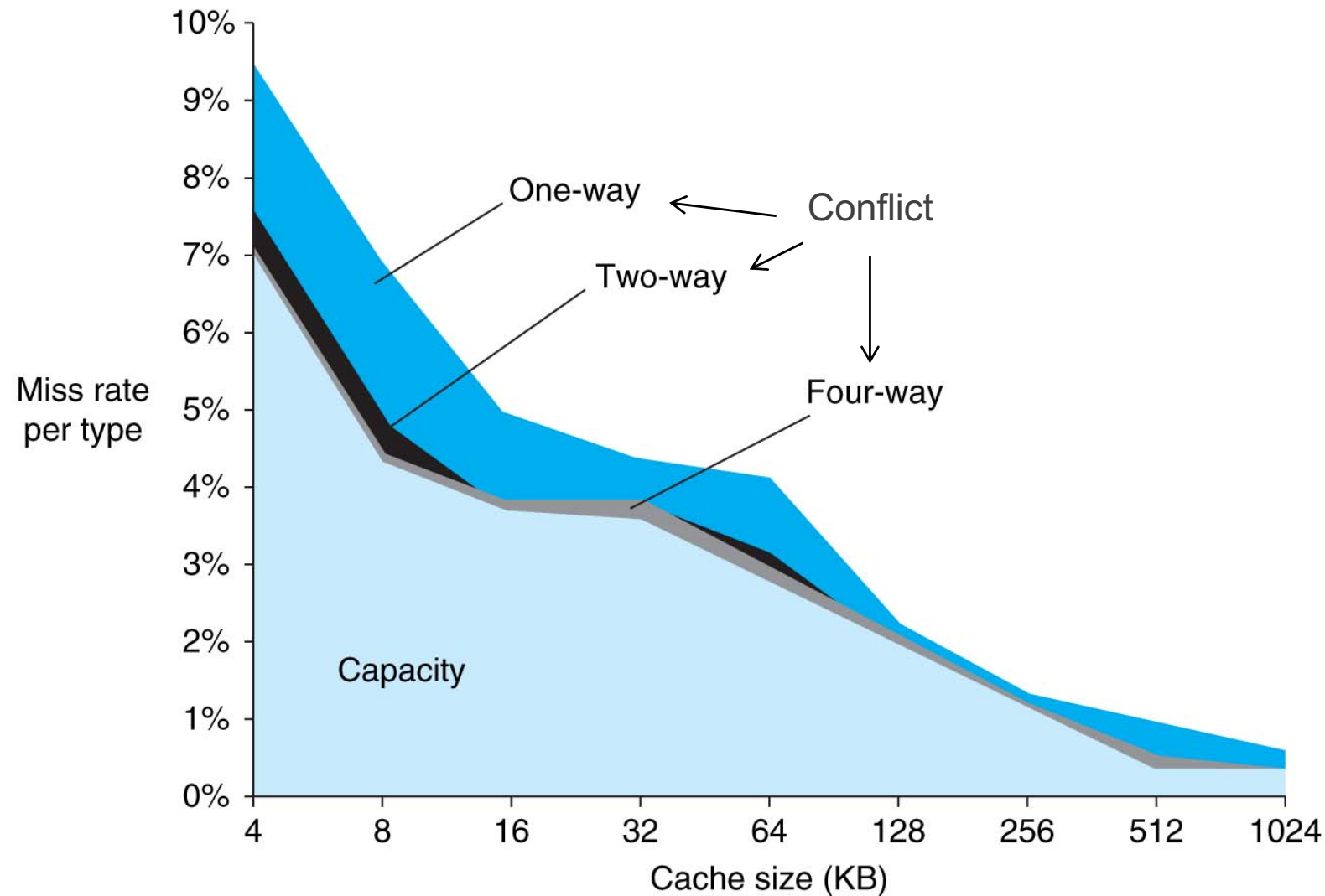
- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache:  $0.02 \times 100 = 2$
  - D-cache:  $0.36 \times 0.04 \times 100 = 1.44$
- Actual CPI =  $2 + 2 + 1.44 = 5.44$



# Memory Hierarchy Basics

- Miss rate
  - Fraction of cache access that result in a miss
- Causes of misses (3C's +1)
  - Compulsory
    - First reference to a block
  - Capacity
    - Blocks discarded and later retrieved
  - Conflict
    - Program makes repeated references to multiple addresses from different blocks that map to the same location in the cache
  - Coherency
    - Different processors should see same value in same location

# The 3C's in diff cache sizes



# The cache coherence pb

- Processors may see different values through their caches:

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

# Cache Coherence

## ■ Coherence

- All reads by any processor must return the most recently written value
- Writes to the same location by any two processors are seen in the same order by all processors

*(Coherence defines the behaviour of reads & writes to the same memory location)*

## ■ Consistency

- When a written value will be returned by a read
- If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

*(Consistency defines the behaviour of reads & writes with respect to accesses to other memory locations)*

# Enforcing Coherence

- Coherent caches provide:
  - *Migration*: movement of data
  - *Replication*: multiple copies of data
- Cache coherence protocols
  - Directory based
    - Sharing status of each block kept in one location
  - Snooping
    - Each core tracks sharing status of each block

# Memory Hierarchy Basics

- Six basic cache optimizations:
  - Larger block size
    - Reduces compulsory misses
    - Increases capacity and conflict misses, increases miss penalty
  - Larger total cache capacity to reduce miss rate
    - Increases hit time, increases power consumption
  - Higher associativity
    - Reduces conflict misses
    - Increases hit time, increases power consumption
  - Multilevel caches to reduce miss penalty
    - Reduces overall memory access time
  - Giving priority to read misses over writes
    - Reduces miss penalty
  - Avoiding address translation in cache indexing
    - Reduces hit time

# Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

# Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty = ??? = 400 cycles
  - Effective CPI = 1 + ??? = 9
- Now add L-2 cache ...



# Multilevel Cache Example

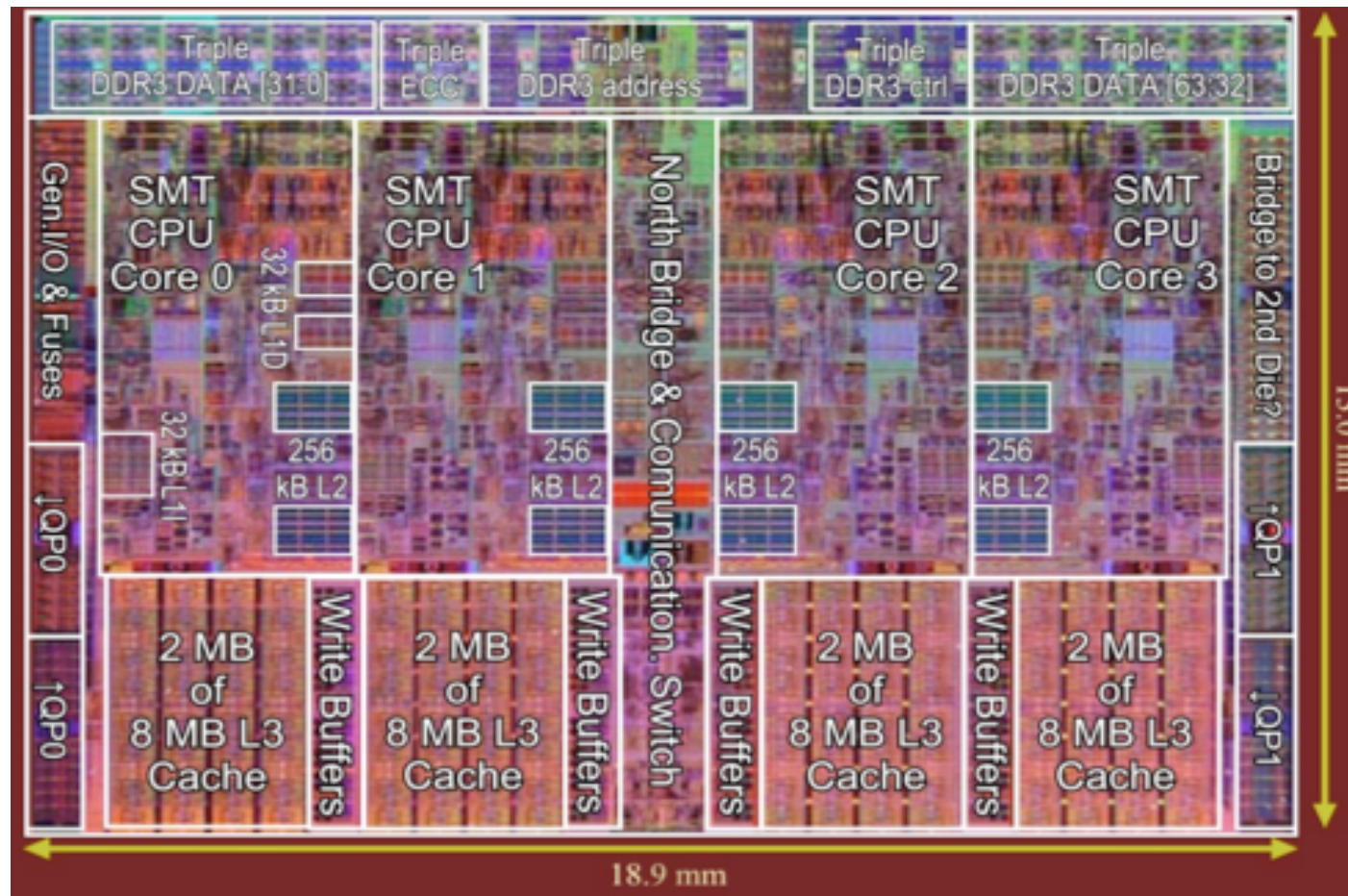
- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty =  $100\text{ns} / 0.25\text{ns} = 400$  cycles
  - Effective CPI =  $1 + 0.02 \times 400 = 9$
- Now add L-2 cache ...

# Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty =  $5\text{ns}/0.25\text{ns} = 20$  cycles
- Primary miss with L-2 miss
  - Extra penalty = 400 cycles
- $\text{CPI} = 1 + 0.02 \times 20 + 0.005 \times 400 = 3.4$
- Performance ratio =  $9/3.4 = 2.6$

# Multilevel On-Chip Caches

Intel Nehalem 4-core processor



Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache

# 3-Level Cache Organization

	Intel Nehalem	AMD Opteron X4
L1 caches (per core)	L1 I-cache: 32KB, 64-byte blocks, 4-way, <b>approx LRU replacement</b> , hit time n/a L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	L1 I-cache: 32KB, 64-byte blocks, 2-way, <b>approx LRU replacement</b> , hit time 3 cycles L1 D-cache: 32KB, 64-byte blocks, 2-way, <b>approx LRU replacement</b> , write-back/allocate, hit time 9 cycles
L2 unified cache (per core)	256KB, 64-byte blocks, 8-way, <b>approx LRU replacement</b> , write-back/allocate, hit time n/a	512KB, 64-byte blocks, 16-way, <b>approx LRU replacement</b> , write-back/allocate, hit time n/a
L3 unified cache (shared)	8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a	2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles

n/a: data not available



# Intel new cache approach with Skylake



## Re-Architected L2 & L3 Cache Hierarchy



- On-chip cache balance shifted from shared-distributed (prior architectures) to private-local (Skylake architecture):
  - Shared-distributed → shared-distributed L3 is primary cache
  - Private-local → private L2 becomes primary cache with shared L3 used as overflow cache
- Shared L3 changed from inclusive to non-inclusive:
  - Inclusive (prior architectures) → L3 has copies of all lines in L2
  - Non-inclusive (Skylake architecture) → lines in L2 **may not** exist in L3

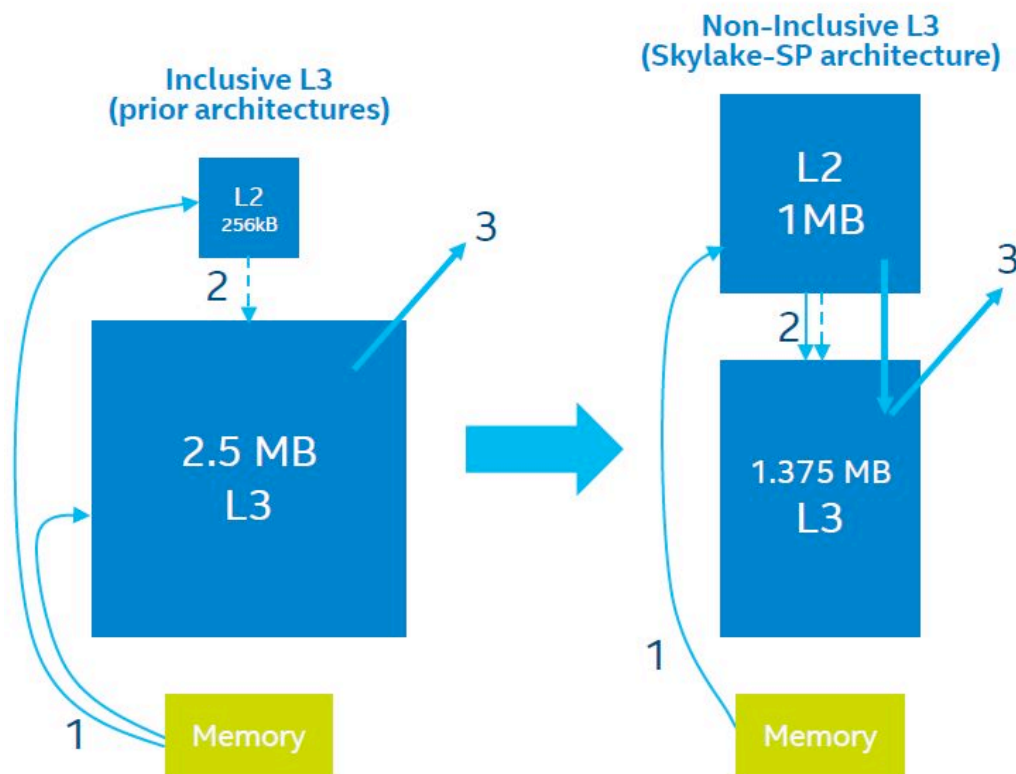
**SKYLAKE-SP CACHE HIERARCHY ARCHITECTED SPECIFICALLY FOR DATA CENTER USE CASE**



# Intel new cache approach with Skylake



## Inclusive vs Non-Inclusive L3



1. Memory reads fill directly to the L2, no longer to both the L2 and L3
2. When a L2 line needs to be removed, both modified and unmodified lines are written back
3. Data shared across cores are copied into the L3 for servicing future L2 misses

### Cache hierarchy architected and optimized for data center use cases:

- Virtualized use cases get larger private L2 cache free from interference
- Multithreaded workloads can operate on larger data per thread (due to increased L2 size) and reduce uncore activity

# Ten Advanced Optimizations



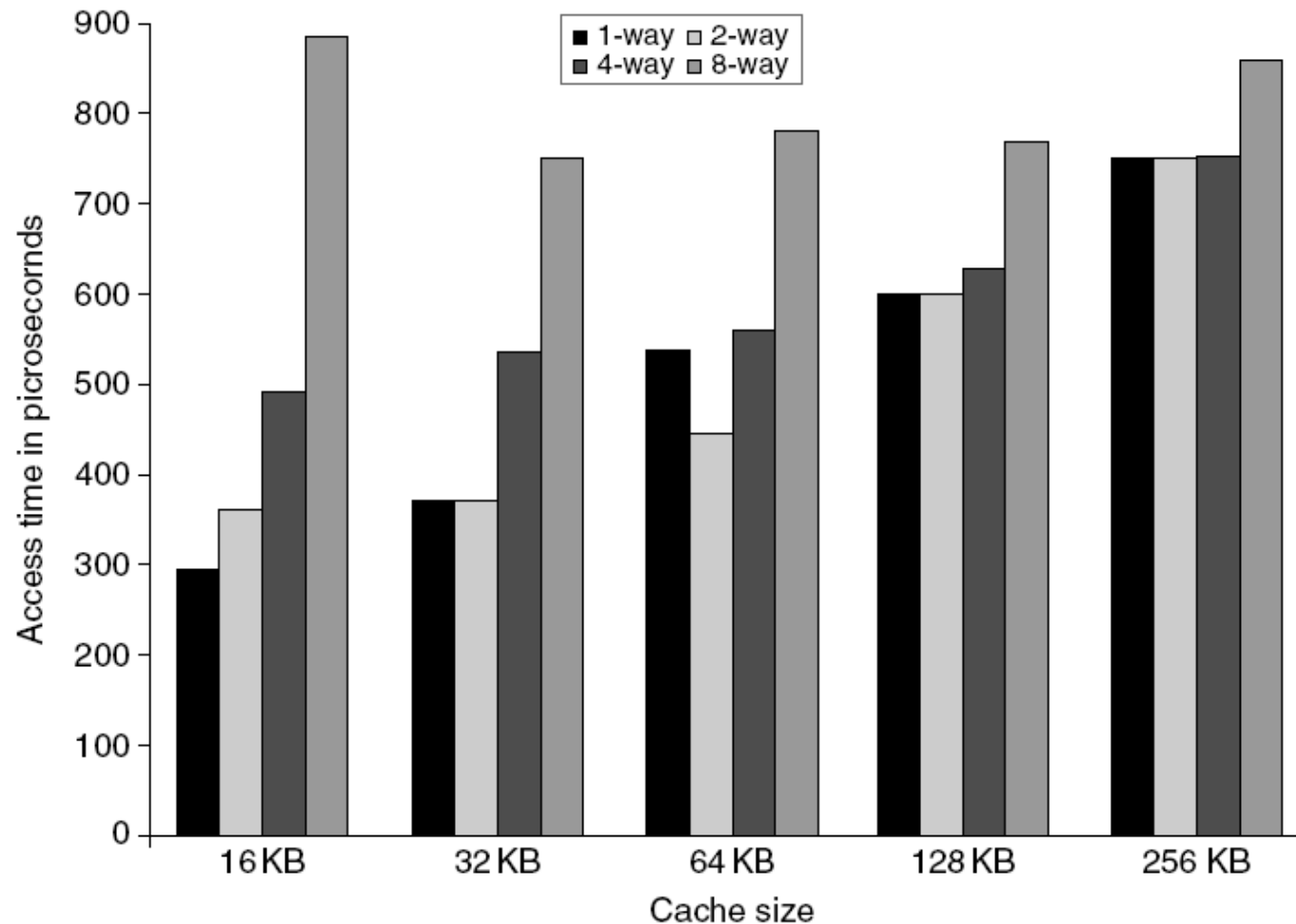
- Reducing the hit time
  1. Small & simple first-level caches
  2. Way-prediction
- Increase cache bandwidth
  3. Pipelined cache access
  4. Nonblocking caches
  5. Multibanked caches
- Reducing the miss penalty
  6. Critical word first
  7. Merging write buffers
- Reducing the miss rate
  8. Compiler optimizations
- Reducing the miss penalty or miss rate via parallelism
  9. Hardware prefetching of instructions and data
  10. Compiler-controlled prefetching

# 1. Small and simple 1<sup>st</sup> level caches

- Small and simple first level caches
  - Critical timing path:
    - addressing tag memory, then
    - comparing tags, then
    - selecting correct set
  - Direct-mapped caches can overlap tag compare and transmission of data
  - Lower associativity reduces power because fewer cache lines are accessed

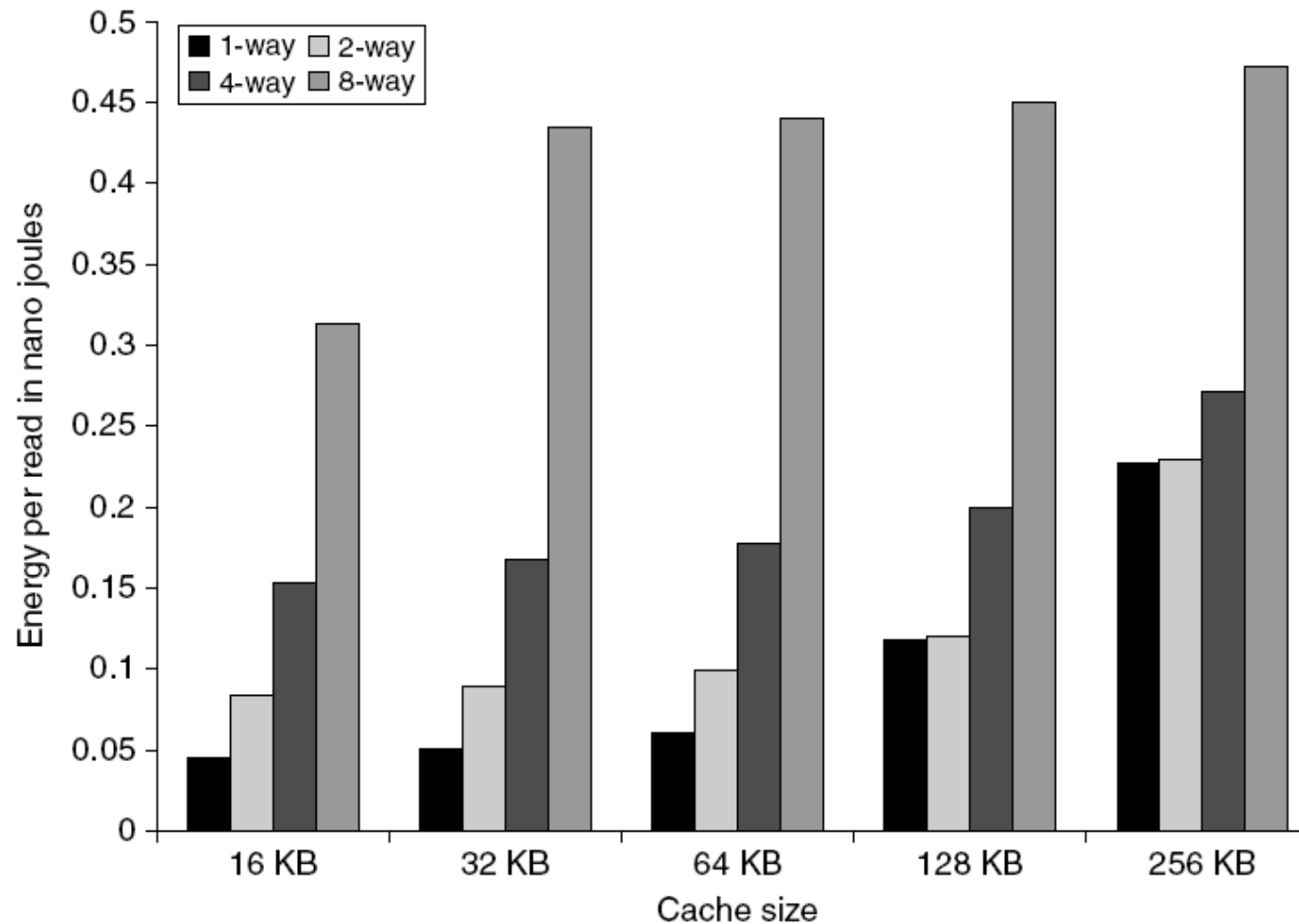


# L1 Size and Associativity



Access time vs. size and associativity

# L1 Size and Associativity



Energy per read vs. size and associativity

## 2. Way Prediction

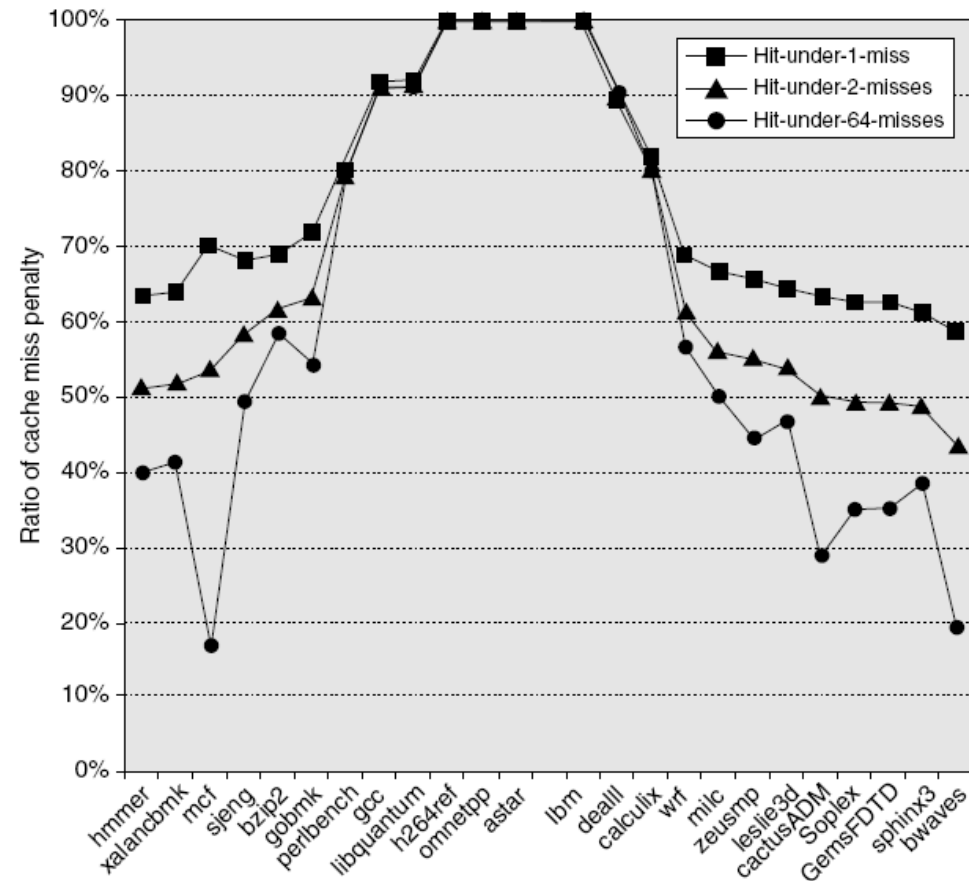
- To improve hit time, predict the way to pre-set mux
  - Mis-prediction gives longer hit time
  - Prediction accuracy
    - > 90% for two-way
    - > 80% for four-way
    - I-cache has better accuracy than D-cache
  - First used on MIPS R10000 in mid-90s
  - Used on ARM Cortex-A8
- Extend to predict block as well
  - “Way selection”
  - Increases mis-prediction penalty

### 3. Pipelining Cache

- Pipeline cache access to improve bandwidth
  - Examples:
    - Pentium: 1 cycle
    - Pentium Pro – Pentium III: 2 cycles
    - Pentium 4 – Core i7: 4 cycles
- Increases branch mis-prediction penalty
- Makes it easier to increase associativity

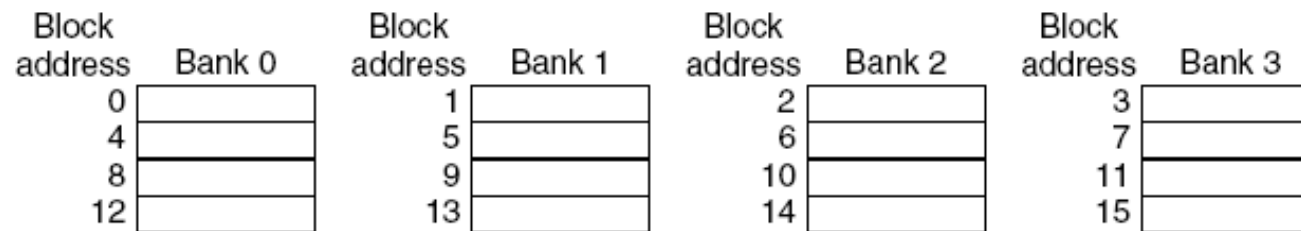
## 4. Nonblocking Caches

- Allow hits before previous misses complete
  - “Hit under miss”
  - “Hit under multiple miss”
- L2 must support this
- In general, processors can hide L1 miss penalty but not L2 miss penalty



## 5. Multibanked Caches

- Organize cache as independent banks to support simultaneous access
  - ARM Cortex-A8 supports 1-4 banks for L2
  - Intel i7 supports 4 banks for L1 and 8 banks for L2
  
- Interleave banks according to block address



**Figure 2.6** Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

## 6. Critical Word First, Early Restart

- Critical word first
  - Request missed word from memory first
  - Send it to the processor as soon as it arrives
- Early restart
  - Request words in normal order
  - Send missed work to the processor as soon as it arrives
- Effectiveness of these strategies depends on block size and likelihood of another access to the portion of the block that has not yet been fetched

# 7. Merging Write Buffer

- When storing to a block that is already pending in the write buffer, update write buffer
- Reduces stalls due to full write buffer
- Do not apply to I/O addresses

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

No write buffering

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

Write buffering

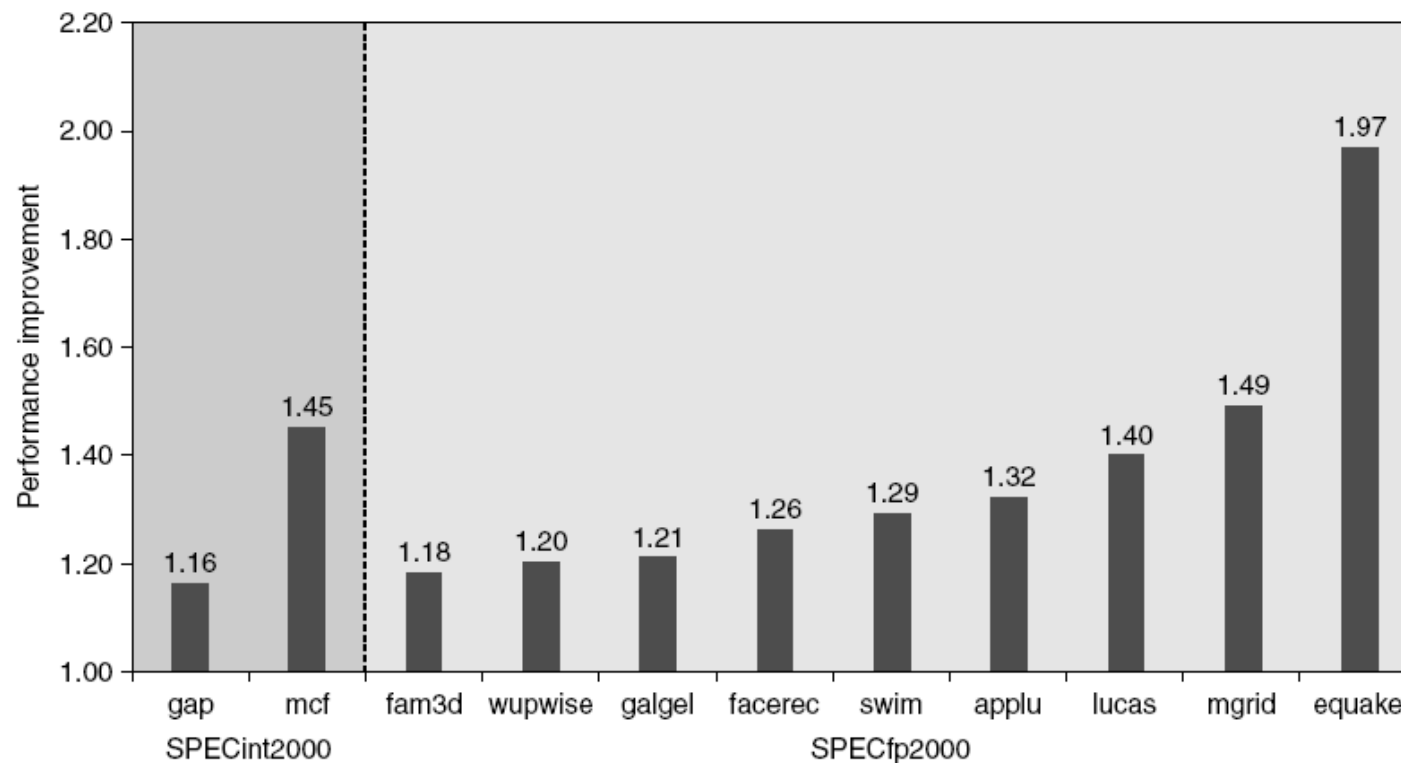


## 8. Compiler Optimizations

- Loop Interchange
  - Swap nested loops to access memory in sequential order
- Blocking
  - Instead of accessing entire rows or columns, subdivide matrices into blocks
  - Requires more memory accesses but improves locality of accesses

# 9. Hardware Prefetching

- Fetch two blocks on miss (include next sequential block)



Pentium 4 Pre-fetching

# 10. Compiler Prefetching

- Insert prefetch instructions before data is needed
- Non-faulting: prefetch doesn't cause exceptions
- Register prefetch
  - Loads data into register
- Cache prefetch
  - Loads data into cache
- Combine with loop unrolling and software pipelining

# Summary

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined cache access	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Banked caches		+			+	1	Used in L2 of both i7 and Cortex-A8
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, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware.
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs