IC-UNICAMP

MO401

IC/Unicamp
2013s1
Prof Mario Côrtes

Capítulo 2:
Hierarquia de Memória

IC-UNICAMP

Tópicos

- Desempenho de Cache: 10 otimizações
- Memória: tecnologia e otimizações
- Proteção: memória virtual e máquinas virtuais
- Hierarquia de memória
- Hierarquia de memória do ARM Cortex-A8 e do Intel Core i7

2.1 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 Hierarchy

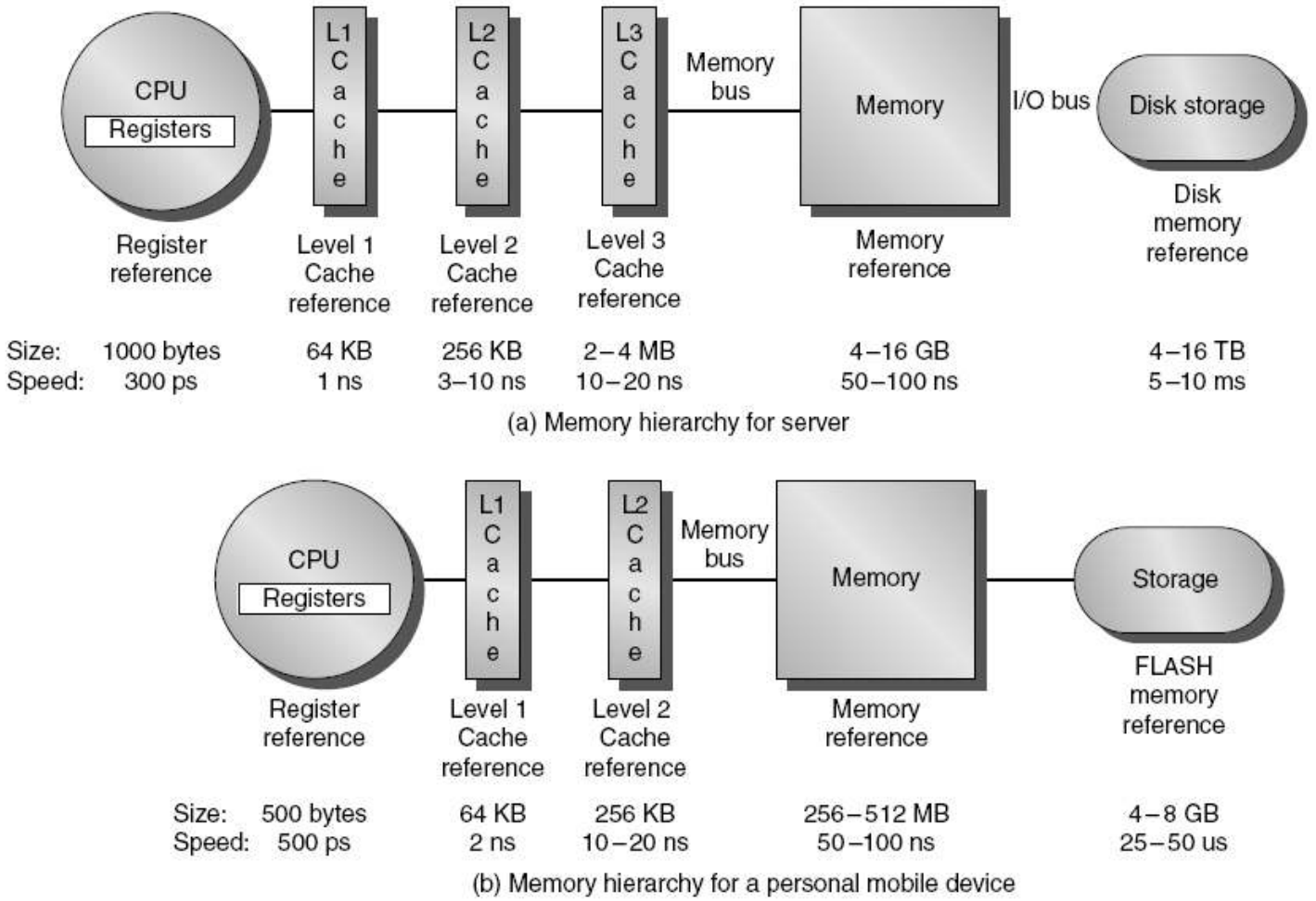


Figure 2.1 The levels in a typical memory hierarchy in a server computer shown on top (a) and in a personal mobile device (PMD) on the bottom (b). As we move farther away from the processor, the memory in the level below becomes slower and larger. Note that the time units change by a factor of 10^9—from picoseconds to milliseconds—and that the size units change by a factor of 10^{12}—from bytes to terabytes. The PMD has a slower clock rate and smaller caches and main memory. A key difference is that servers and desktops use disk storage as the lowest level in the hierarchy while PMDs use Flash, which is built from EEPROM technology.

Memory Performance Gap

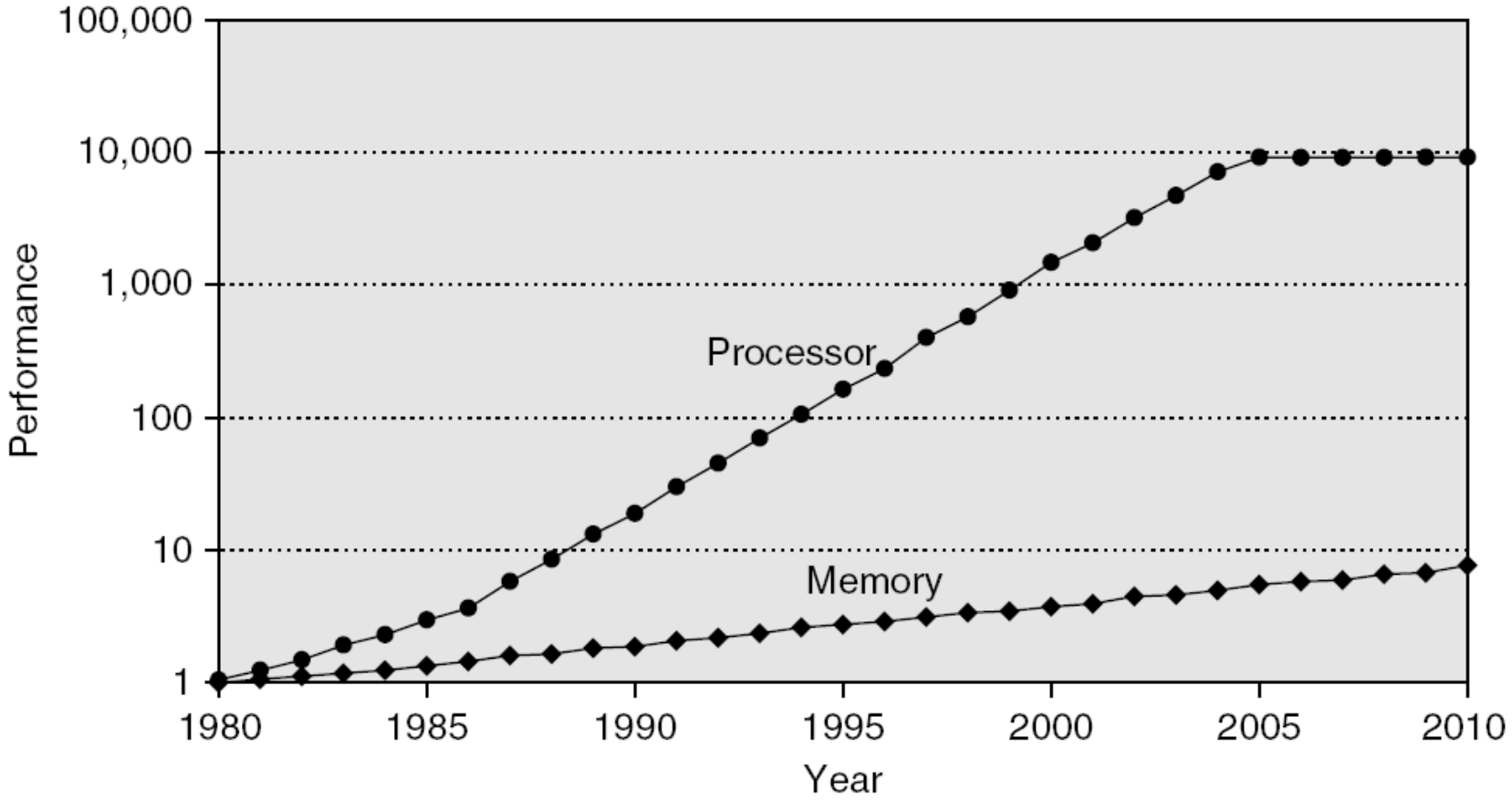


Figure 2.2 Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time. Note that the vertical axis must be on a logarithmic scale to record the size of the processor–DRAM performance gap. The memory baseline is 64 KB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 2.13 on page 99). The processor line assumes a 1.25 improvement per year until 1986, a 1.52 improvement until 2000, a 1.20 improvement between 2000 and 2005, and no change in processor performance (on a per-core basis) between 2005 and 2010; see Figure 1.1 in Chapter 1.

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

Performance and Power

- High-end microprocessors have >10 MB on-chip cache
  - Consumes large amount of area and power budget
  - Consumo de energia das caches
    - inativa (leakage)
    - ativa (potência dinâmica)
  - Problema ainda mais grave em PMDs: power budget 50x menor
    - caches podem ser responsáveis por 25-50% do consumo

Memory Hierarchy Basics

- When a word is not found in the cache, a *miss* occurs:
  - Fetch word from lower level in hierarchy, requiring a higher latency reference
  - Lower level may be another cache or the main memory
  - Also fetch the other words contained within the *block*
    - Takes advantage of spatial locality
  - Place block into cache in any location within its *set*, determined by address
    - block address MOD number of sets

Memory Hierarchy Basics

- *n* sets => *n-way set associative*
  - *Direct-mapped cache* => one block per set
  - *Fully associative* => 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

- Miss rate
  - Fraction of cache access that result in a miss

- Causes of misses
  - 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

Fig B9: Miss Rate vs tipo de Miss

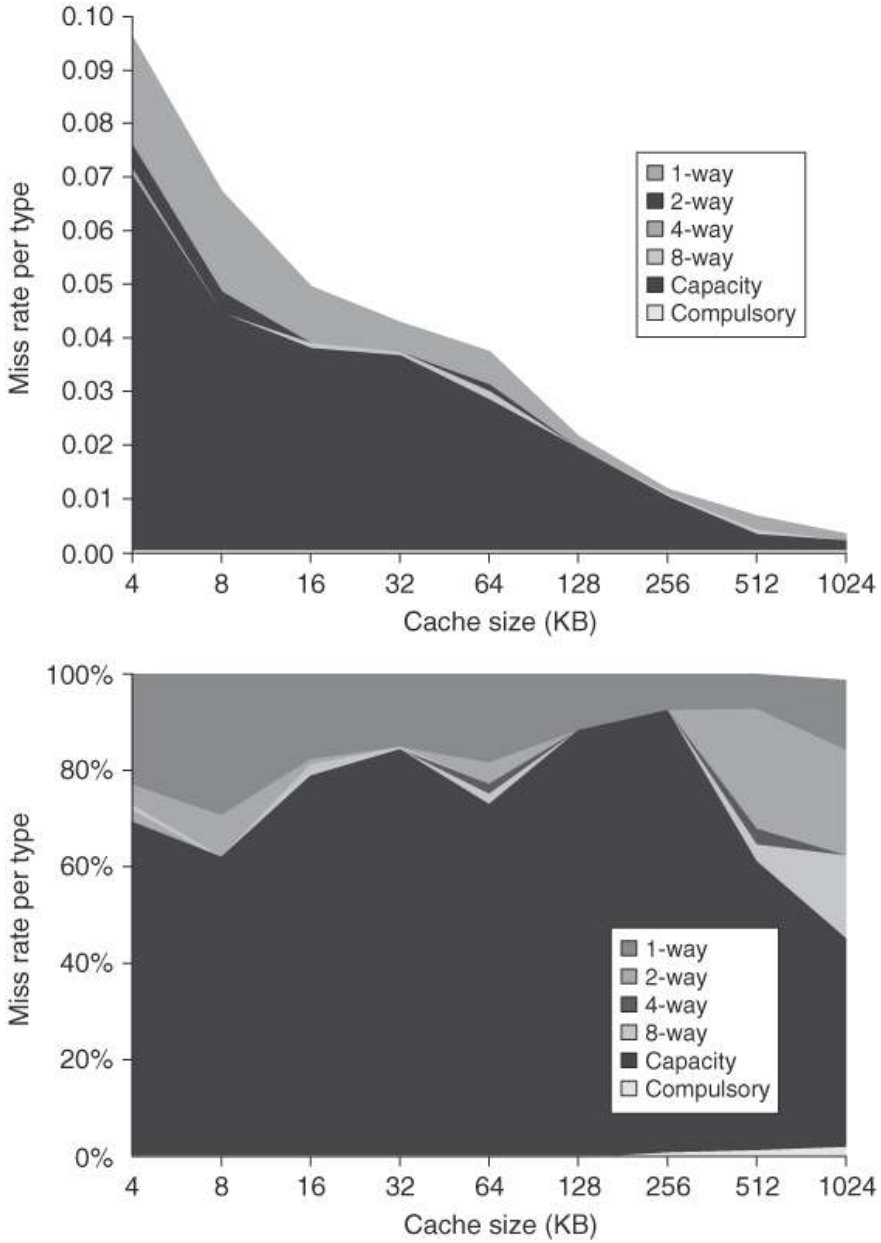


Figure B.9 Total miss rate (top) and distribution of miss rate (bottom) for each size cache according to the three C's for the data in Figure B.8. The top diagram shows the actual data cache miss rates, while the bottom diagram shows the percentage in each category. (Space allows the graphs to show one extra cache size than can fit in Figure B.8.)

Memory Hierarchy Basics

- Miss Rate pode ser “misleading”; alguns preferem Miss/Instruction

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

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

- Note that speculative and multithreaded processors may execute other instructions during a miss
  - Reduces performance impact of misses

- AMAT (Ave Mem Access Time)
  - AMAT = Hit time + Miss Rate x Miss Penalty

Introdução: Desempenho da Cache

- **Orientado ao Miss no acesso à Memória:**

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemAccess}{Inst} \times MissRate \times MissPenalty \right) \times CycleTime$$

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemMisses}{Inst} \times MissPenalty \right) \times CycleTime$$

  - $CPI_{Execution}$ inclui instruções da ALU e de acesso à Memória

- **Isolando o acesso à Memória**
  - AMAT = Average Memory Access Time
  - CPI_{ALUOps} não inclui as instruções de memória

$$CPUtime = IC \times \left(\frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime$$

$$\begin{aligned} AMAT &= HitTime + MissRate \times MissPenalty \\ &= (HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst}) + \\ &\quad (HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data}) \end{aligned}$$

Impacto no Desempenho

IC-UNICAMP

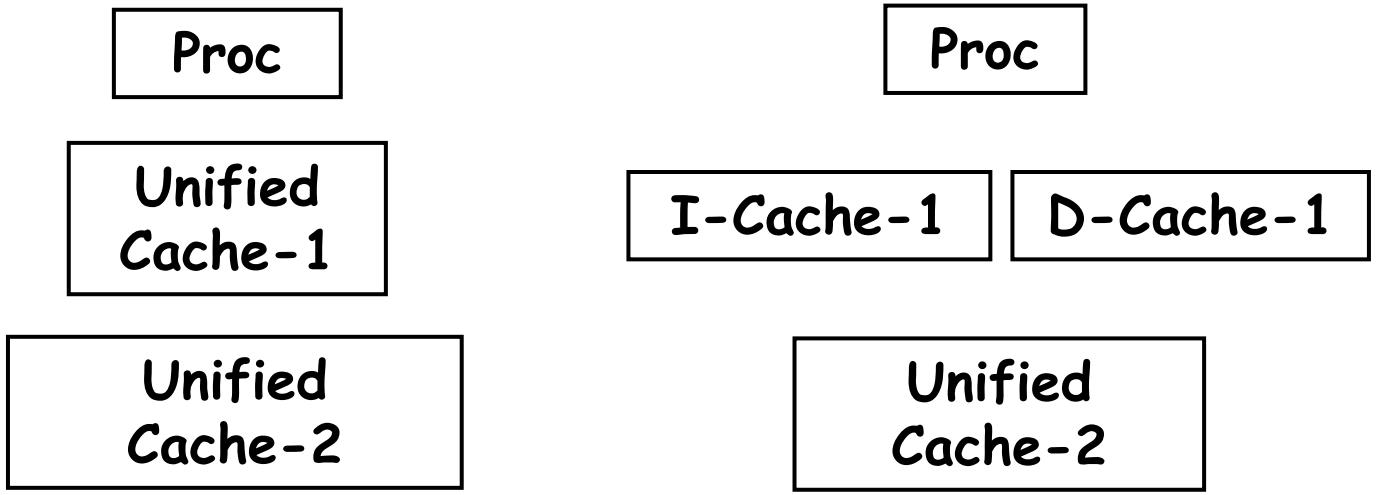
- Suponha um processador executando:
  - Clock Rate = 200 MHz (5 ns por ciclo), CPI Ideal (sem misses) = 1.1
  - 50% aritmética/lógica, 30% ld/st, 20% controle
  - Miss rate cache de dados = 10%, Miss penalty = 50 ciclos; Hit Time = 1 ciclo
  - Miss rate cache instrução = 1%, Miss penalty = 50 ciclos ; Hit Time = 1 ciclo
- CPI = ideal CPI + average stalls per instruction
  1.1(cycles/ins) +
  [0.30 (DataMops/ins) x 0.10 (miss/DataMop) x 50 (cycle/miss)] +
  [1 (InstMop/ins) x 0.01 (miss/InstMop) x 50 (cycle/miss)]
  = (1.1 + 1.5 + 0.5) cycle/ins = 3.1
- 64% (=(3.1-1.1)/3.1) do tempo do processador é devido a stall esperando pela memória!
- 48% (=1.5/3.1) devido a espera por dados!
- AMAT=(1/1.3)x[1+0.01x50]+(0.3/1.3)x[1+0.1x50]=2.54

instruções dados

IC-UNICAMP

Exemplo: Arquitetura Harvard

- **Cache Unificada vs Separada I&D (Harvard)**



IC-UNICAMP

Exemplo: Arquitetura Harvard

- **Suponha:**
  - **16KB I&D: Inst miss rate = 0.64%, Data miss rate = 6.47%**
  - **32KB unificada: miss rate = 1.99% (agregado)**
  - **33% das instruções são ops de dados (load ou store)**
- **Qual é melhor (ignore cache L2)?**
  - **33% ops de dados $\Rightarrow$ 75% dos acessos são devidos a fetch das instruções (1.0/1.33)**
  - **hit time = 1, miss time = 50**
  - **Note que *data* hit tem 1 stall para a cache unificada (somente uma porta)**
- **AMATHarvard = 75%x(1+0.64%x50) + 25%x(1+6.47%x50) = 2.05**
- **AMATUnified = 75%x(1+1.99%x50) + 25%x(1+1+1.99%x50)= 2.24**

IC-UNICAMP

Para melhorar o desempenho da cache

1. Reduzir miss rate
2. Reduzir miss penalty
3. Reduzir *tempo de hit na cache*

$$AMAT = HitTime + MissRate \times MissPenalty$$

Memory Hierarchy Basics (Ap B)

- Six basic cache optimizations:
  - 1 Larger block size
    - Reduces compulsory misses
    - Increases capacity and conflict misses, increases miss penalty
  - 2 Larger total cache capacity to reduce miss rate
    - Increases hit time, increases power consumption
  - 3 Higher associativity
    - Reduces conflict misses
    - Increases hit time, increases power consumption
  - 4 Higher number of cache levels
    - Reduces overall memory access time
  - 5 Giving priority to read misses over writes
    - Reduces miss penalty
  - 6 Avoiding address translation in cache indexing
    - Reduces hit time

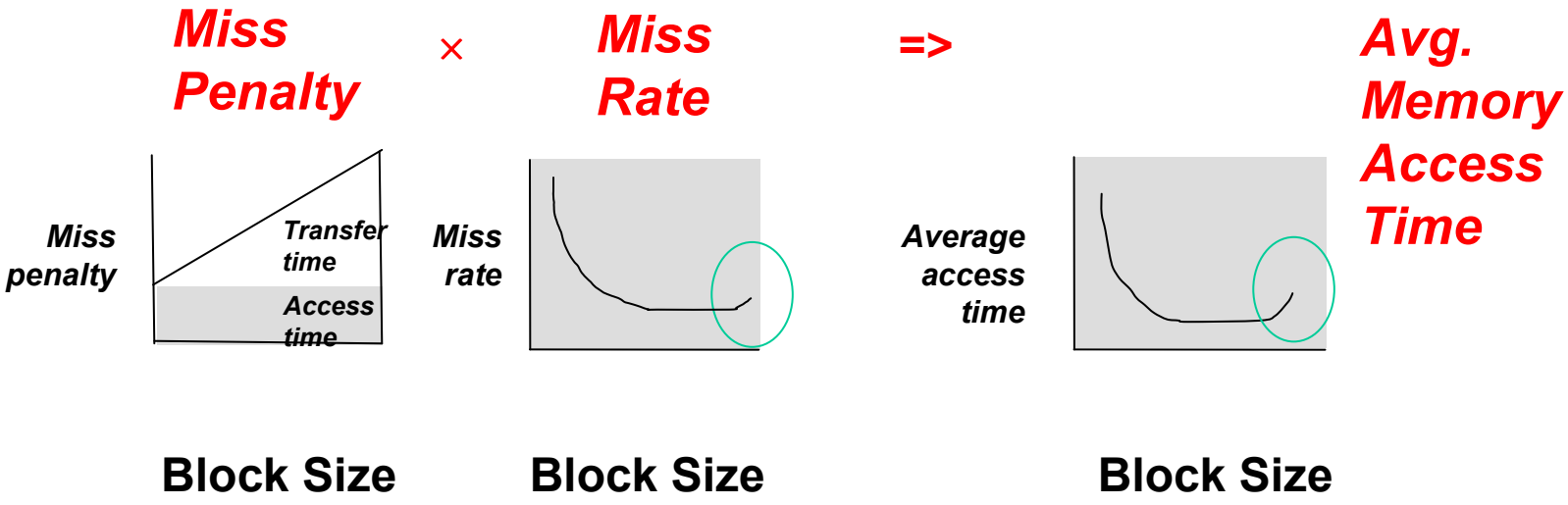
IC-UNICAMP

1- Larger block size

- Reduces compulsory misses
- Increases capacity and conflict misses, increases miss penalty

Miss Penalty e Miss Rate vs Tamanho do Bloco

- Maior tamanho do bloco →
  - → maior miss penalty
  - → menor miss rate



Miss rate x block size

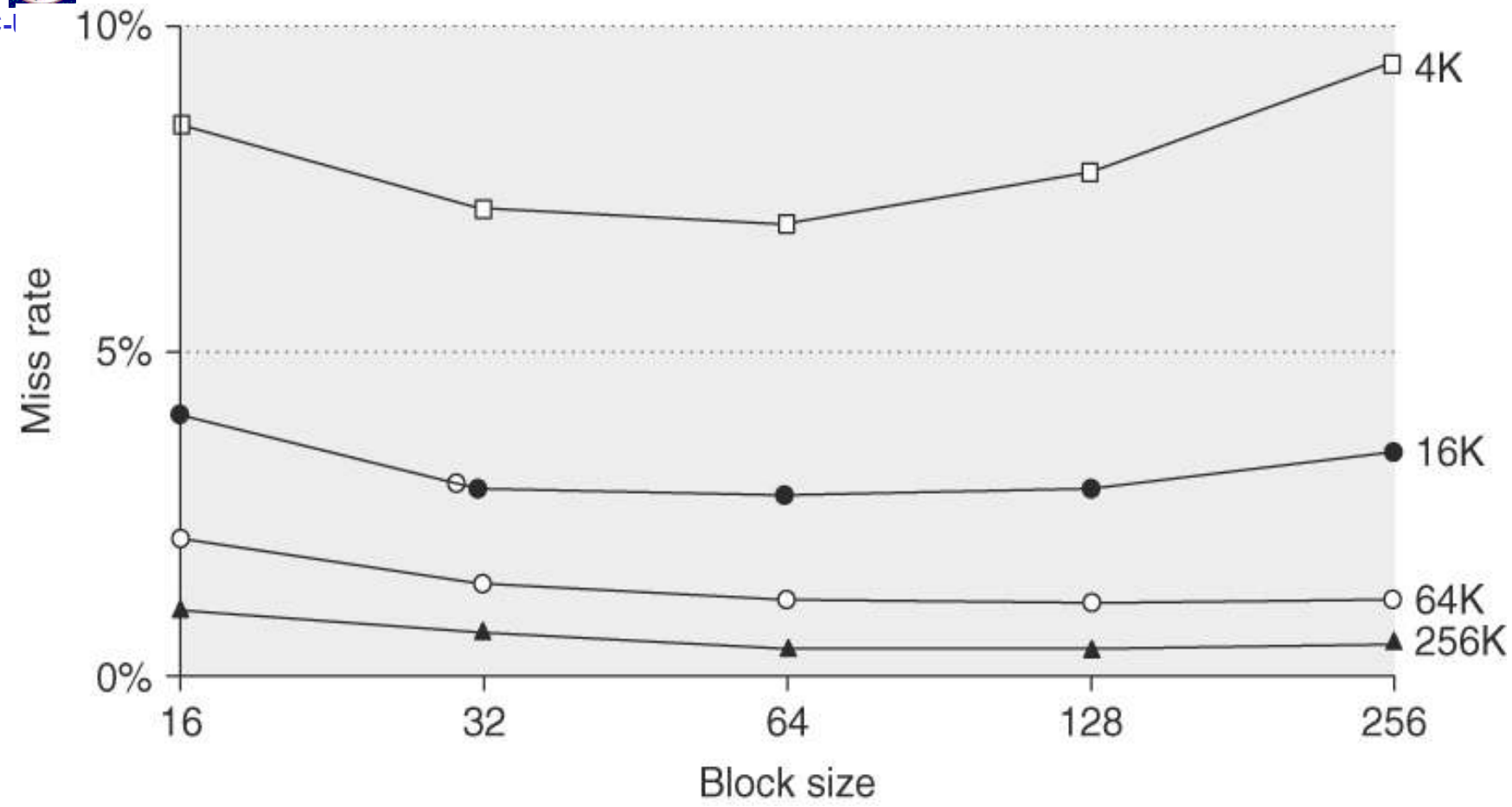


Figure B.10 Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. Figure B.11 shows the data used to plot these lines. Unfortunately, SPEC2000 traces would take too long if block size were included, so these data are based on SPEC92 on a DECstation 5000 [Gee et al. 1993].

IC-UNICAMP

Exmpl pag B-27: Miss rate e block size

Example Figure B.11 shows the actual miss rates plotted in Figure B.10. 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 Figure B.11?

Answer Average memory access time is

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

If we assume the hit time is 1 clock cycle independent of block size, then the access time for a 16-byte block in a 4 KB cache is

$$\text{Average memory access time} = 1 + (8.57\% \times 82) = 8.027 \text{ clock cycles}$$

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

$$\text{Average memory access time} = 1 + (0.49\% \times 112) = 1.549 \text{ clock cycles}$$

	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%

Figure B.11 Actual miss rate versus block size for the five different-sized caches in

Exmpl p B-27: Miss rate e block size (2)

IC-UNICAMP

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

Figure B.12 Average memory access time versus block size for five different-sized caches in Figure B.10. Block sizes of 32 and 64 bytes dominate. The smallest average time per cache size is boldfaced.

IC-UNICAMP

2- Larger total cache capacity to reduce miss rate

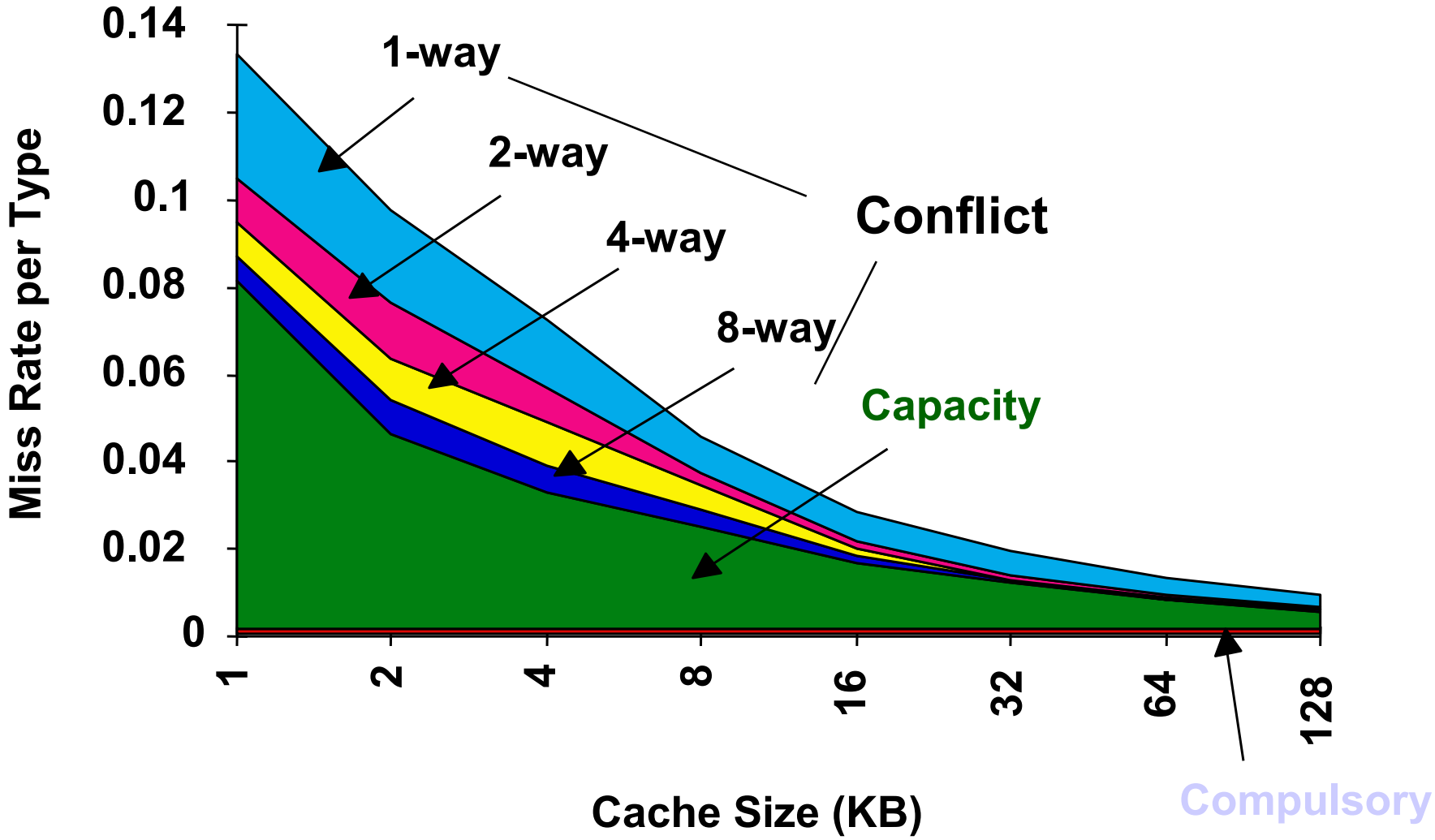
- Increases hit time, increases power consumption

IC-UNICAMP

3- Higher associativity

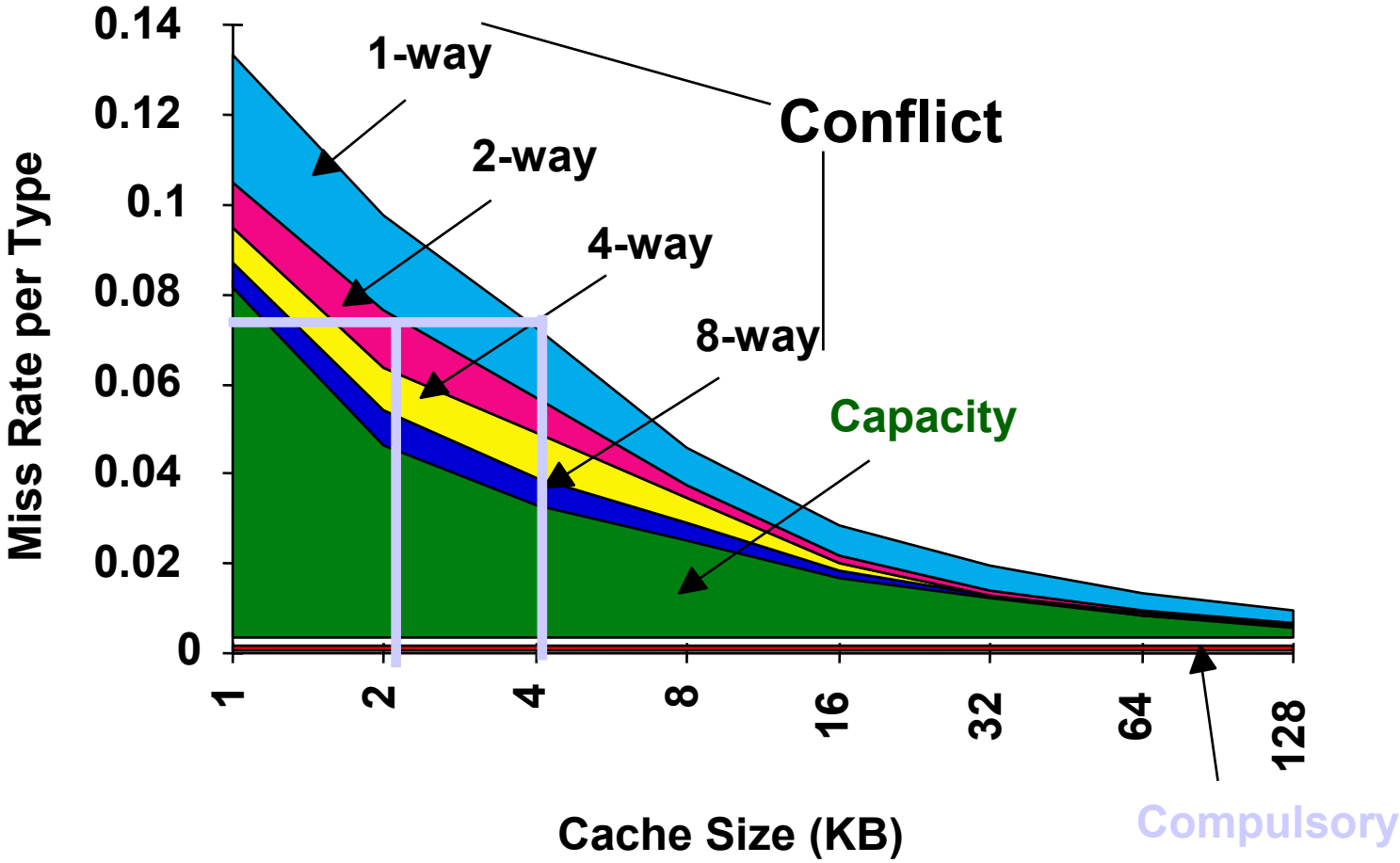
- Reduces conflict misses
- Increases hit time, increases power consumption

3Cs - Miss Rate Absoluto (SPEC92)



Cache Misses

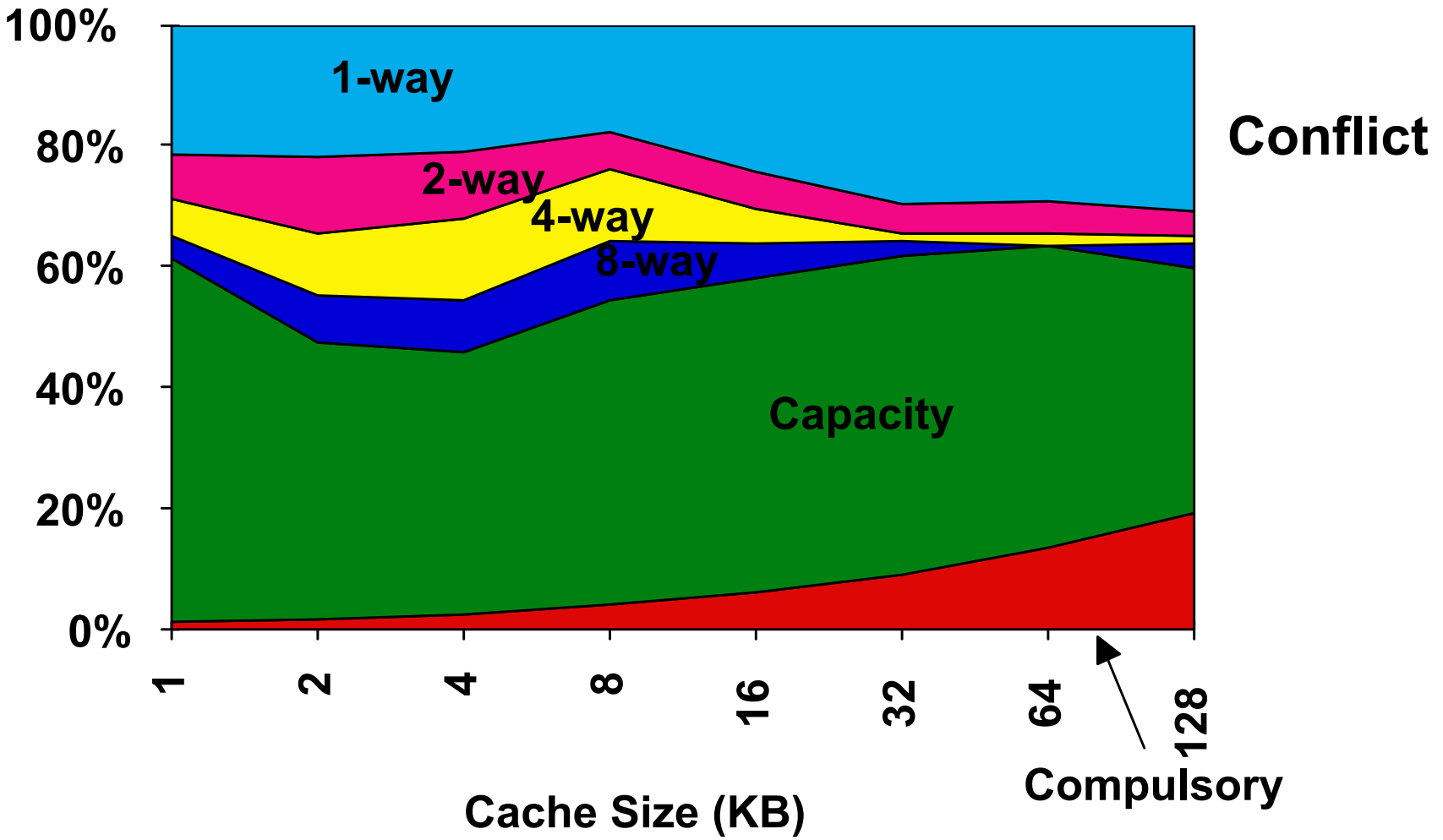
miss rate 1-way associative cache size X
= miss rate 2-way associative cache size X/2



IC-UNICAMP

3Cs Miss Rate Relativo

Flaws: for fixed block size
Good: insight => invention



IC-UNICAMP

Exmpl B-29: 3rd optimization (associativity)

Example Assume that higher associativity would increase the clock cycle time as listed below:

$$\text{Clock cycle time}_{2\text{-way}} = 1.36 \times \text{Clock cycle time}_{1\text{-way}}$$
$$\text{Clock cycle time}_{4\text{-way}} = 1.44 \times \text{Clock cycle time}_{1\text{-way}}$$
$$\text{Clock cycle time}_{8\text{-way}} = 1.52 \times \text{Clock cycle time}_{1\text{-way}}$$

Assume that the hit time is 1 clock cycle, that the miss penalty for the direct-mapped case is 25 clock cycles to a level 2 cache (see next subsection) that never misses, and that the miss penalty need not be rounded to an integral number of clock cycles. Using Figure B.8 for miss rates, for which cache sizes are each of these three statements true?

$$\text{Average memory access time}_{8\text{-way}} < \text{Average memory access time}_{4\text{-way}}$$
$$\text{Average memory access time}_{4\text{-way}} < \text{Average memory access time}_{2\text{-way}}$$
$$\text{Average memory access time}_{2\text{-way}} < \text{Average memory access time}_{1\text{-way}}$$

IC-UNICAMP

Exmpl B-29: 3rd optimization (cont)

Answer Average memory access time for each associativity is

$$\begin{aligned}
\text{Average memory access time}_{8\text{-way}} &= \text{Hit time}_{8\text{-way}} + \text{Miss rate}_{8\text{-way}} \times \text{Miss penalty}_{8\text{-way}} \\
&= 1.52 + \text{Miss rate}_{8\text{-way}} \times 25 \\
\text{Average memory access time}_{4\text{-way}} &= 1.44 + \text{Miss rate}_{4\text{-way}} \times 25 \\
\text{Average memory access time}_{2\text{-way}} &= 1.36 + \text{Miss rate}_{2\text{-way}} \times 25 \\
\text{Average memory access time}_{1\text{-way}} &= 1.00 + \text{Miss rate}_{1\text{-way}} \times 25
\end{aligned}$$

The miss penalty is the same time in each case, so we leave it as 25 clock cycles. For example, the average memory access time for a 4 KB direct-mapped cache is

$$\text{Average memory access time}_{1\text{-way}} = 1.00 + (0.098 \times 25) = 3.44$$

and the time for a 512 KB, eight-way set associative cache is

$$\text{Average memory access time}_{8\text{-way}} = 1.52 + (0.006 \times 25) = 1.66$$

Using these formulas and the miss rates from Figure B.8, Figure B.13 shows the average memory access time for each cache and associativity. The figure shows that the formulas in this example hold for caches less than or equal to 8 KB for up to four-way associativity. Starting with 16 KB, the greater hit time of larger associativity outweighs the time saved due to the reduction in misses.

Note that we did not account for the slower clock rate on the rest of the program in this example, thereby understating the advantage of direct-mapped cache.

IC-UNICAMP

Exmpl B-29: 3rd optimization (cont)

	Associativity			
Cache size (KB)	1-way	2-way	4-way	8-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**

Figure B.13 Average memory access time using miss rates in Figure B.8 for parameters in the example. Boldface type means that this time is higher than the number to the left, that is, higher associativity *increases* average memory access time.

IC-UNICAMP

4- Higher number of cache levels

- Reduces overall memory access time (supor 2 níveis)
- AMAT = Hit time$_{L1}$ + Miss Rate$_{L1}$ x Miss Penalty$_{L1}$
- Miss Penalty$_{L1}$ = Hit time$_{L2}$ + Miss Rate$_{L2}$ x Miss Penalty$_{L2}$
- AMAT = Hit time$_{L1}$ + Miss Rate$_{L1}$ x
  (Hit time$_{L2}$ + Miss Rate$_{L2}$ x Miss Penalty$_{L2}$
- Definições
  - Local Miss Rate = total de misses nesta cache / total de **acessos a esta cache**
    - para L1 = MissRate$_{L1}$ e para L2 = MissRate$_{L2}$
  - Global Miss Rate = total de misses nesta cache / total de **acessos de memória**
    - para L1 = MissRate$_{L1}$ e para L2 = MissRate$_{L1}$ x MissRate$_{L2}$
  - Ave Mem Stalls per Instruction = (Misses/Instruction$_{L1}$) x (HitTime$_{L2}$) + (Misses/Instruction$_{L2}$) x (MissPenalty$_{L2}$)

IC-UNICAMP

Exmpl B-31: multilevel caches

Example Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache. What are the various miss rates? Assume the miss penalty from the L2 cache to memory is 200 clock cycles, the hit time of the L2 cache is 10 clock cycles, the hit time of L1 is 1 clock cycle, and there are 1.5 memory references per instruction. What is the average memory access time and average stall cycles per instruction? Ignore the impact of writes.

Answer The miss rate (either local or global) for the first-level cache is 40/1000 or 4%. The local miss rate for the second-level cache is 20/40 or 50%. The global miss rate of the second-level cache is 20/1000 or 2%. Then

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$
$$= 1 + 4\% \times (10 + 50\% \times 200) = 1 + 4\% \times 110 = 5.4 \text{ clock cycles}$$

To see how many misses we get per instruction, we divide 1000 memory references by 1.5 memory references per instruction, which yields 667 instructions. Thus, we need to multiply the misses by 1.5 to get the number of misses per 1000 instructions. We have 40×1.5 or 60 L1 misses, and 20×1.5 or 30 L2 misses, per 1000 instructions. For average memory stalls per instruction, assuming the misses are distributed uniformly between instructions and data:

IC-UNICAMP

Exmpl B-31: multilevel caches

$$\begin{aligned}\text{Average memory stalls per instruction} &= \text{Misses per instruction}_{L1} \times \text{Hit time}_{L2} + \text{Misses per instruction}_{L2} \\ &\quad \times \text{Miss penalty}_{L2} \\ &= (60/1000) \times 10 + (30/1000) \times 200 \\ &= 0.060 \times 10 + 0.030 \times 200 = 6.6 \text{ clock cycles}\end{aligned}$$

If we subtract the L1 hit time from the average memory access time (AMAT) and then multiply by the average number of memory references per instruction, we get the same average memory stalls per instruction:

$$(5.4 - 1.0) \times 1.5 = 4.4 \times 1.5 = 6.6 \text{ clock cycles}$$

As this example shows, there may be less confusion with multilevel caches when calculating using misses per instruction versus miss rates.

Miss rate em cache (single e L2)

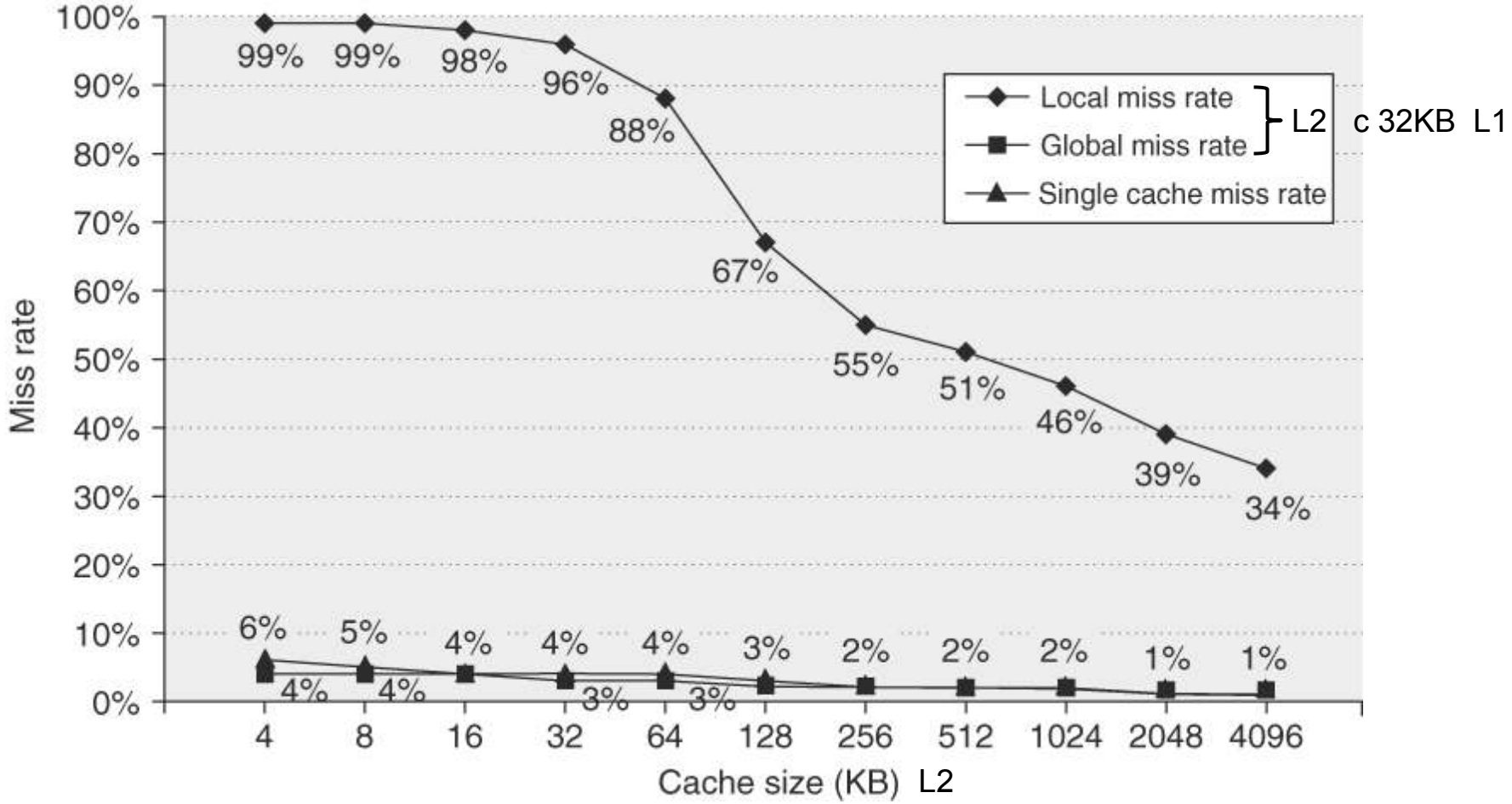


Figure B.14 Miss rates versus cache size for multilevel caches. Second-level caches *smaller* than the sum of the two 64 KB first-level caches make little sense, as reflected in the high miss rates. After 256 KB the single cache is within 10% of the global miss rates. The miss rate of a single-level cache versus size is plotted against the local miss rate and global miss rate of a second-level cache using a 32 KB first-level cache. The L2 caches (unified) were two-way set associative with replacement. Each had split L1 instruction and data caches that were 64 KB two-way set associative with LRU replacement. The block size for both L1 and L2 caches was 64 bytes. Data were collected as in Figure B.4

IC-UNICAMP

Exmpl B-33: multilevel caches

Example Given the data below, what is the impact of second-level cache associativity on its miss penalty?

- Hit time$_{L2}$ for direct mapped = 10 clock cycles.
- Two-way set associativity increases hit time by 0.1 clock cycle to 10.1 clock cycles.
- Local miss rate$_{L2}$ for direct mapped = 25%.
- Local miss rate$_{L2}$ for two-way set associative = 20%.
- Miss penalty$_{L2}$ = 200 clock cycles.

IC-UNICAMP

Exmpl B-33: multilevel caches (cont)

Answer For a direct-mapped second-level cache, the first-level cache miss penalty is

$$\text{Miss penalty}_{\text{1-way L2}} = 10 + 25\% \times 200 = 60.0 \text{ clock cycles}$$

Adding the cost of associativity increases the hit cost only 0.1 clock cycle, making the new first-level cache miss penalty:

$$\text{Miss penalty}_{\text{2-way L2}} = 10.1 + 20\% \times 200 = 50.1 \text{ clock cycles}$$

In reality, second-level caches are almost always synchronized with the first-level cache and processor. Accordingly, the second-level hit time must be an integral number of clock cycles. If we are lucky, we shave the second-level hit time to 10 cycles; if not, we round up to 11 cycles. Either choice is an improvement over the direct-mapped second-level cache:

$$\text{Miss penalty}_{\text{2-way L2}} = 10 + 20\% \times 200 = 50.0 \text{ clock cycles}$$
$$\text{Miss penalty}_{\text{2-way L2}} = 11 + 20\% \times 200 = 51.0 \text{ clock cycles}$$

IC-UNICAMP

5- Giving priority to read misses over writes

- Reduces miss penalty
  - Read posterior “esconde” miss penalty do write
- Em caches com write-through: write buffer
  - pode trazer complicações RAW
  - solução: read verifica se há writes pendentes no buffer
- Em caches com write-back
  - funcionamento idêntico: substituição de block dirty → write buffer

IC-UNICAMP

Exmpl B-35: priority to read miss

Example Look at this code sequence:

```
SW R3, 512(R0)      ;M[512] ← R3      (cache index 0)
LW R1, 1024(R0)     ;R1 ← M[1024]     (cache index 0)
LW R2, 512(R0)      ;R2 ← M[512]      (cache index 0)
```

Assume a direct-mapped, write-through cache that maps 512 and 1024 to the same block, and a four-word write buffer that is not checked on a read miss. Will the value in R2 always be equal to the value in R3?

Answer Using the terminology from Chapter 2, this is a read-after-write data hazard in memory. Let's follow a cache access to see the danger. The data in R3 are placed into the write buffer after the store. The following load uses the same cache index and is therefore a miss. The second load instruction tries to put the value in location 512 into register R2; this also results in a miss. If the write buffer hasn't completed writing to location 512 in memory, the read of location 512 will put the old, wrong value into the cache block, and then into R2. Without proper precautions, R3 would not be equal to R2!

IC-UNICAMP

6- Avoiding address translation in cache indexing

- Duas tarefas na tradução: indexar a cache e comparar tag
- Alternativa1: cache (com endereço) virtual:
  - tradução zero: na indexação e na comp. c tag
  - Problemas com cache virtual:
    - proteção: é verificada na tradução v → p (solução no App B)
    - na mudança de contexto, dados na cache são inúteis → flush the cache (solução no App B)
    - OS e user programs podem usar dois endereços virtuais para o mesmo endereço físico→ aliasing ou synonyms. Podem haver duas cópias do mesmo end. físico (“coerência”?) (solução no App B)
    - I/O usa endereço físico → necessária tradução em op de I/O
- Alternativa2, virtually indexed, physically tagged
  - indexação: usar parte da page offset do end virtual (sem tradução)
    - probl: cache size (direct) <= page size
  - simultaneamente, traduzir parte do end virtual e comparar com tag (end físico)

2.2 Ten Advanced Optimizations

- Redução do Hit Time (e menor consumo de potência)
  - 1: Small and simple L1
  - 2: Way prediction
- Aumento da cache bandwidth
  - 3: Pipelined caches
  - 4: Multibanked caches
  - 5: Nonblocking caches
- Redução da Miss Penalty
  - 6: Critical word fist
  - 7: Merging write buffers
- Redução da Miss Rate
  - 8: Compiler optimization
- Redução de Miss Rate/Penalty via paralelismo
  - 9: Hardware prefetching
  - 10: Compiler prefetching

1- Small and simple L1

- Reduce hit time and power (ver figuras adiante)
- Critical timing path:
  - addressing tag memory, then
  - comparing tags, then
  - selecting correct set (if set-associative)
- Direct-mapped caches can overlap tag compare and transmission of data (não é preciso selecionar os dados pois não associativo)
- Lower associativity reduces power because fewer cache lines are accessed
- Crescimento de L1 em uProcessadores era tendência; agora estabilizou
  - decisão de projeto
    - associatividade → redução de miss rate; mas
    - associatividade → aumento de hit time e power

IC-UNICAMP

Exmpl p80: associatividade

Example Using the data in Figure B.8 in Appendix B and Figure 2.3, determine whether a 32 KB four-way set associative L1 cache has a faster memory access time than a 32 KB two-way set associative L1 cache. Assume the miss penalty to L2 is 15 times the access time for the faster L1 cache. Ignore misses beyond L2. Which has the faster average memory access time?

Answer Let the access time for the two-way set associative cache be 1. Then, for the two-way cache:

$$\begin{aligned}\text{Average memory access time}_{\text{2-way}} &= \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \\ &= 1 + 0.038 \times 15 = 1.38\end{aligned}$$

For the four-way cache, the access time is 1.4 times longer. The elapsed time of the miss penalty is 15/1.4 = 10.1. Assume 10 for simplicity:

$$\begin{aligned}\text{Average memory access time}_{\text{4-way}} &= \text{Hit time}_{\text{2-way}} \times 1.4 + \text{Miss rate} \times \text{Miss penalty} \\ &= 1.4 + 0.037 \times 10 = 1.77\end{aligned}$$

Clearly, the higher associativity looks like a bad trade-off; however, since cache access in modern processors is often pipelined, the exact impact on the clock cycle time is difficult to assess.

L1 Size and Associativity

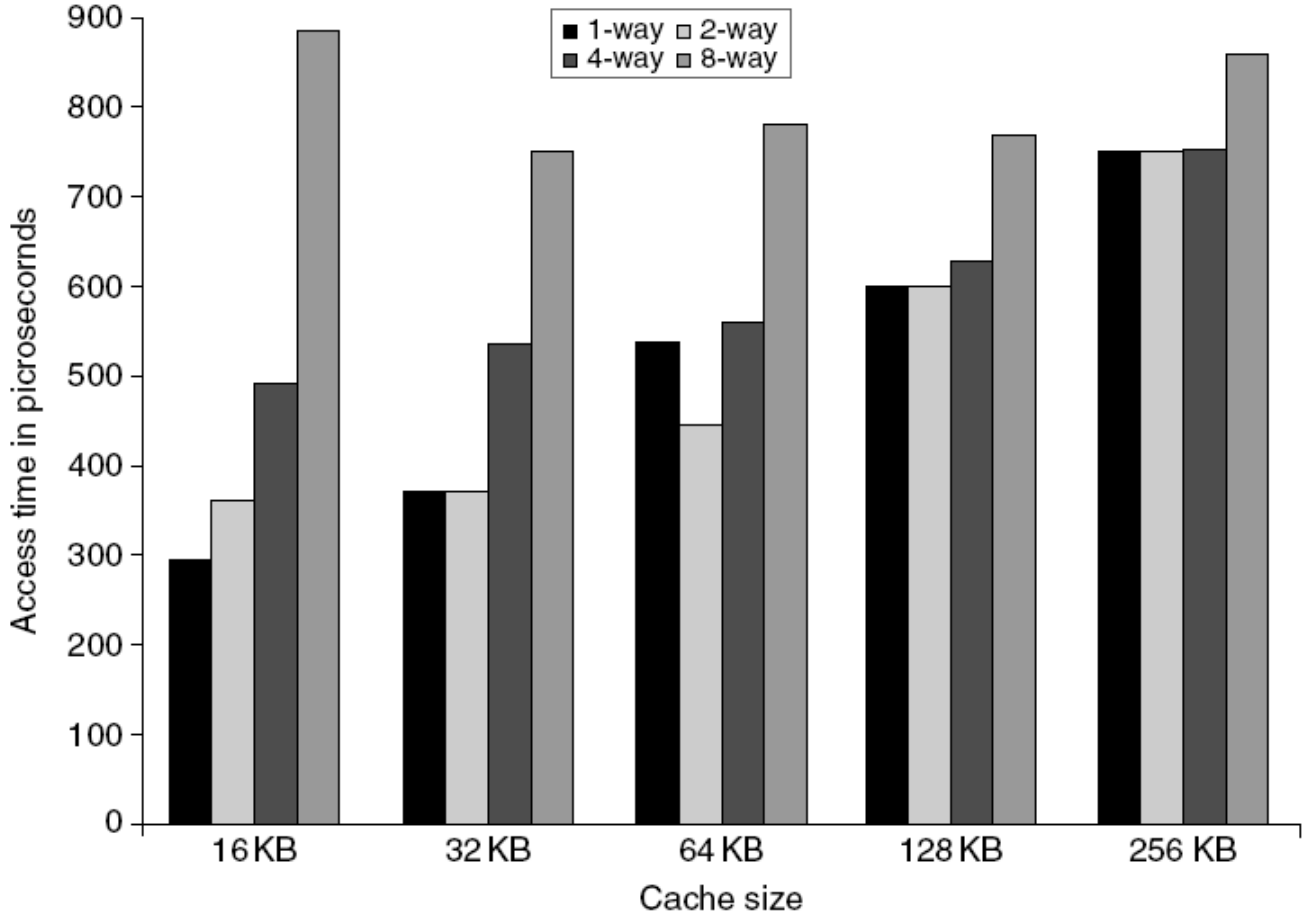


Fig 2.3: Access time vs. size and associativity

L1 Size and Associativity

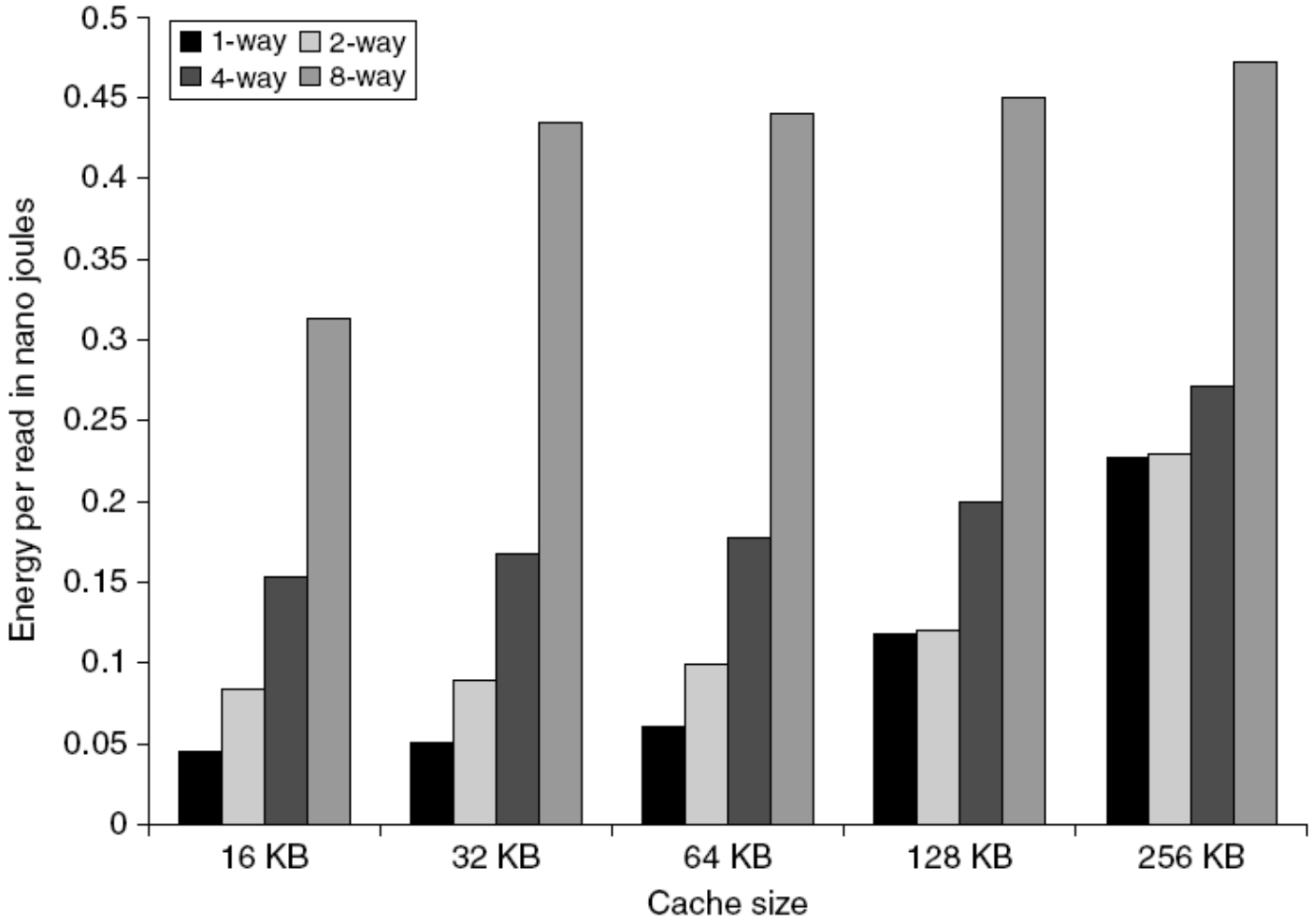


Fig 2.4: Energy per read vs. size and associativity

2- Way Prediction

- To improve hit time, predict the way to pre-set mux
  - Adicionar bits de predição do próximo acesso a cada bloco
  - 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

Exmpl p82: way prediction

IC-

Inoue, Ishihara, and Murakami [1999] estimated that using the way selection approach with a four-way set associative cache increases the average access time for the I-cache by 1.04 and for the D-cache by 1.13 on the SPEC95 benchmarks, but it yields an average cache power consumption relative to a normal four-way set associative cache that is 0.28 for the I-cache and 0.35 for the D-cache. One significant drawback for way selection is that it makes it difficult to pipeline the cache access.

Example Assume that there are half as many D-cache accesses as I-cache accesses, and that the I-cache and D-cache are responsible for 25% and 15% of the processor's power consumption in a normal four-way set associative implementation. Determine if way selection improves performance per watt based on the estimates from the study above.

Answer For the I-cache, the savings in power is $25 \times 0.28 = 0.07$ of the total power, while for the D-cache it is $15 \times 0.35 = 0.05$ for a total savings of 0.12. The way prediction version requires 0.88 of the power requirement of the standard 4-way cache. The increase in cache access time is the increase in I-cache average access time plus one-half the increase in D-cache access time, or $1.04 + 0.5 \times 0.13 = 1.11$ times longer. This result means that way selection has 0.90 of the performance of a standard four-way cache. Thus, way selection improves performance per joule very slightly by a ratio of $0.90/0.88 = 1.02$. This optimization is best used where power rather than performance is the key objective.

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 to increase BW

- Em processadores com execução for a de ordem e pipeline
  - Em um Miss, Cache (I e D) podem continuar com o próximo acesso e não ficam bloqueadas (hit under miss) → redução do Miss Penalty
- Idéia básica: hit under miss
  - Vantagens aumentam se hit “under multiple miss”, etc
- Nonblocking = lockup free

IC-UNICAMP

Exmpl p83: non blocking caches

Example Which is more important for floating-point programs: two-way set associativity or hit under one miss for the primary data caches? What about integer programs? Assume the following average miss rates for 32 KB data caches: 5.2% for floating-point programs with a direct-mapped cache, 4.9% for these programs with a two-way set associative cache, 3.5% for integer programs with a direct-mapped cache, and 3.2% for integer programs with a two-way set associative cache. Assume the miss penalty to L2 is 10 cycles, and the L2 misses and penalties are the same.

Answer For floating-point programs, the average memory stall times are

$$\text{Miss rate}_{DM} \times \text{Miss penalty} = 5.2\% \times 10 = 0.52$$

$$\text{Miss rate}_{2\text{-way}} \times \text{Miss penalty} = 4.9\% \times 10 = 0.49$$

IC-UNICAMP

Exmpl p83: non blocking caches (cont)

The cache access latency (including stalls) for two-way associativity is 0.49/0.52 or 94% of direct-mapped cache. The caption of Figure 2.5 says hit under one miss reduces the average data cache access latency for floating point programs to 87.5% of a blocking cache. Hence, for floating-point programs, the direct mapped data cache supporting one hit under one miss gives better performance than a two-way set-associative cache that blocks on a miss.

For integer programs, the calculation is

$$\text{Miss rate}_{\text{DM}} \times \text{Miss penalty} = 3.5\% \times 10 = 0.35$$

$$\text{Miss rate}_{\text{2-way}} \times \text{Miss penalty} = 3.2\% \times 10 = 0.32$$

The data cache access latency of a two-way set associative cache is thus 0.32/0.35 or 91% of direct-mapped cache, while the reduction in access latency when allowing a hit under one miss is 9%, making the two choices about equal.

Latência de nonblocking caches

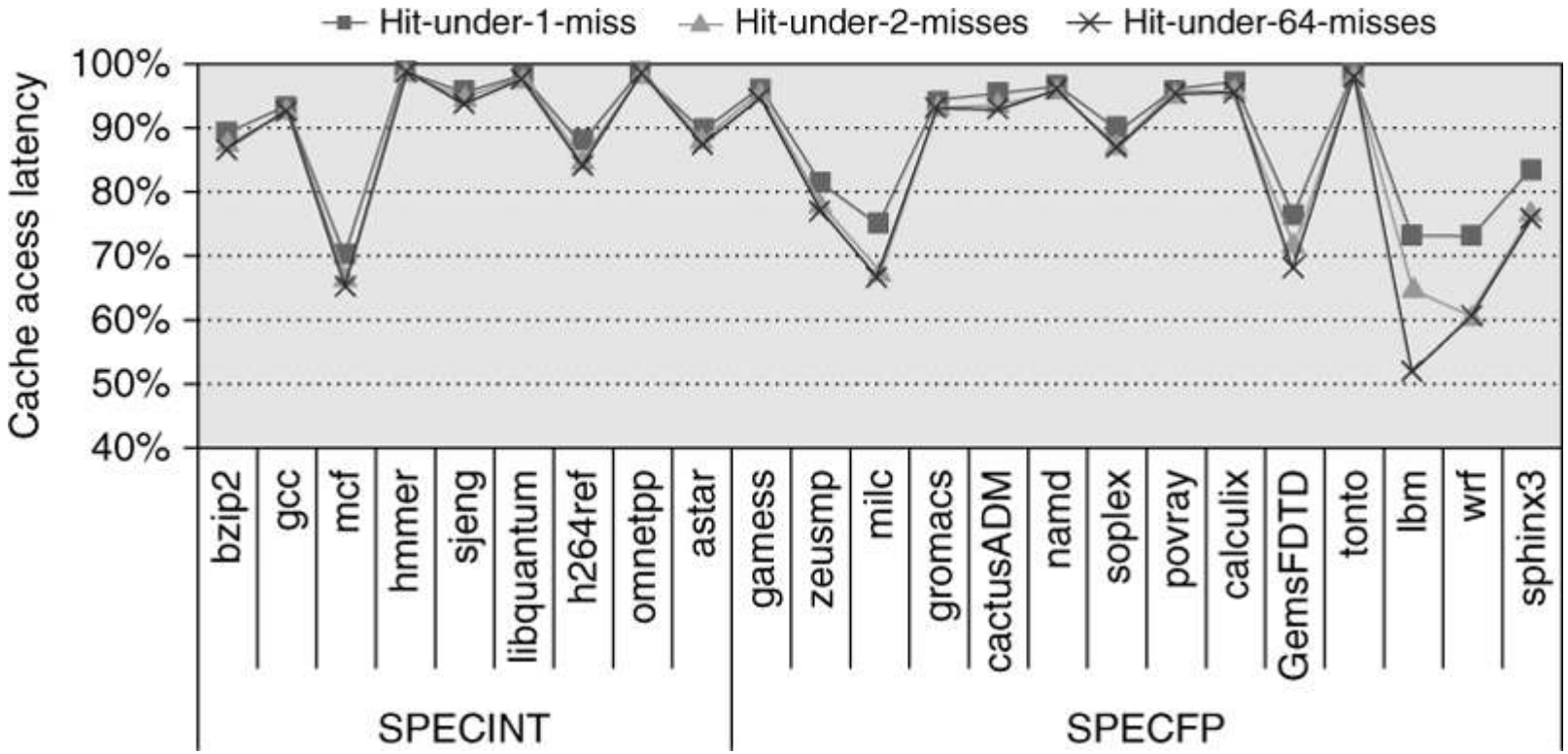
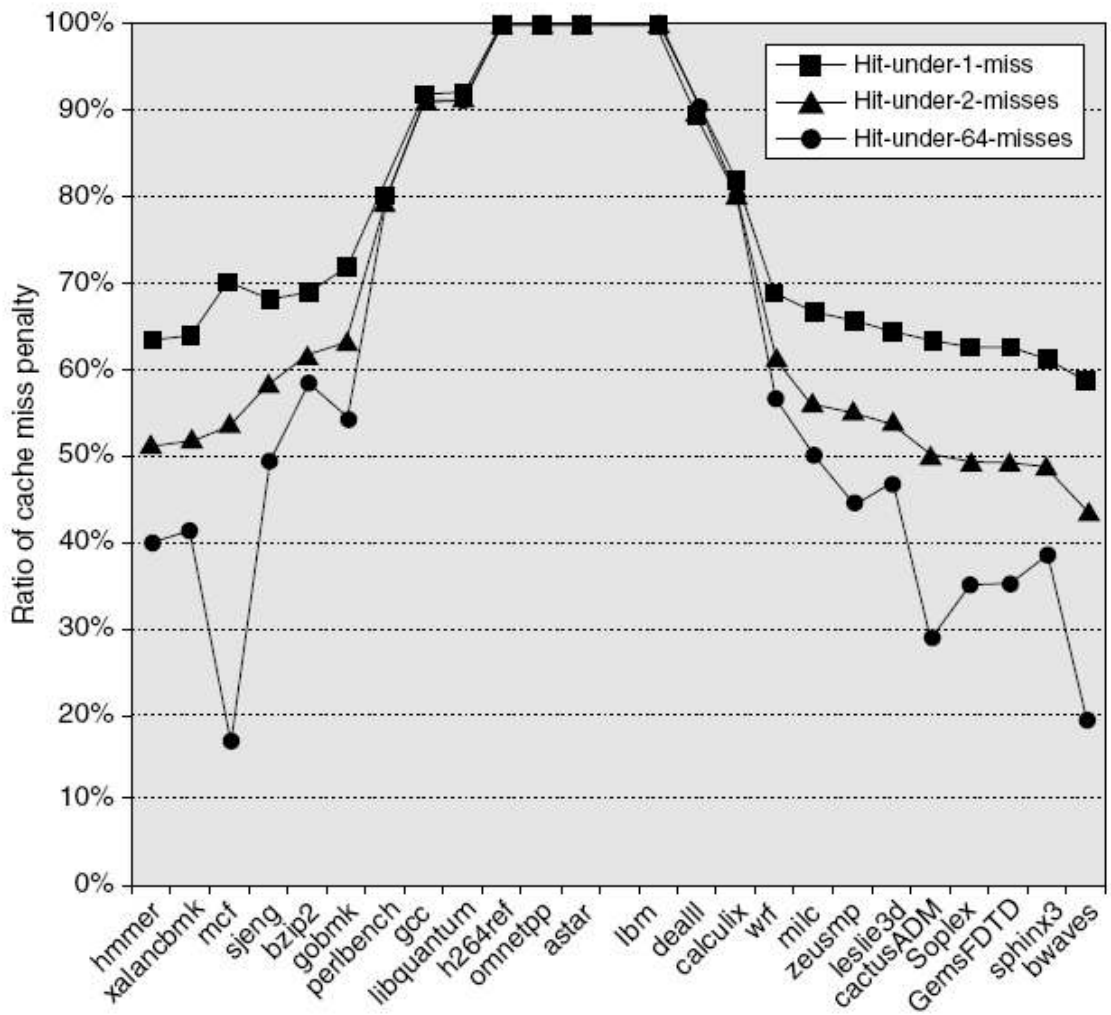


Figure 2.5 The effectiveness of a nonblocking cache is evaluated by allowing 1, 2, or 64 hits under a cache miss with 9 SPECINT (on the left) and 9 SPECFP (on the right) benchmarks. The data memory system modeled after the Intel i7 consists of a 32KB L1 cache with a four cycle access latency. The L2 cache (shared with instructions) is 256 KB with a 10 clock cycle access latency. The L3 is 2 MB and a 36-cycle access latency. All the caches are eight-way set associative and have a 64-byte block size. Allowing one hit under miss reduces the miss penalty by 9% for the integer benchmarks and 12.5% for the floating point. Allowing a second hit improves these results to 10% and 16%, and allowing 64 results in little additional improvement.

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



IC-UNICAMP

Exmpl p85: non blocking caches

Example Assume a main memory access time of 36 ns and a memory system capable of a sustained transfer rate of 16 GB/sec. If the block size is 64 bytes, what is the maximum number of outstanding misses we need to support assuming that we can maintain the peak bandwidth given the request stream and that accesses never conflict. If the probability of a reference colliding with one of the previous four is 50%, and we assume that the access has to wait until the earlier access completes, estimate the number of maximum outstanding references. For simplicity, ignore the time between misses.

Answer In the first case, assuming that we can maintain the peak bandwidth, the memory system can support $(16 \times 10)^9/64 = 250$ million references per second. Since each reference takes 36 ns, we can support $250 \times 10^6 \times 36 \times 10^{-9} = 9$ references. If the probability of a collision is greater than 0, then we need more outstanding references, since we cannot start work on those references; the memory system needs more independent references not fewer! To approximate this, we can simply assume that half the memory references need not be issued to the memory. This means that we must support twice as many outstanding references, or 18.

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

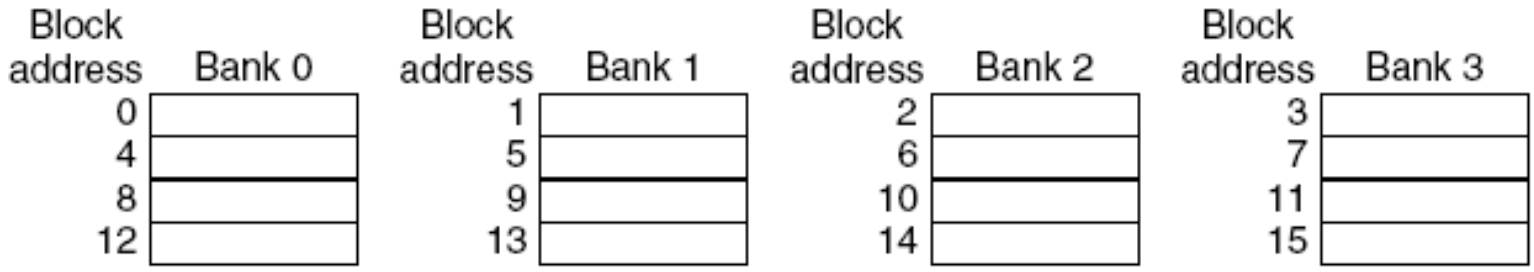


Figure 2.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

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 (e continua preenchendo o bloco da cache com as outras palavras)
- Early restart
  - Request words in normal order (dentro do bloco)
  - Send missed work to the processor as soon as it arrives (e continua preenchendo o bloco….)

- Effectiveness of these strategies depends on block size (maior vantagem se o bloco é grande) 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
  - mesma palavra ou outra palavra do bloco
- Reduces stalls due to full write buffer
- Do not apply to I/O addresses

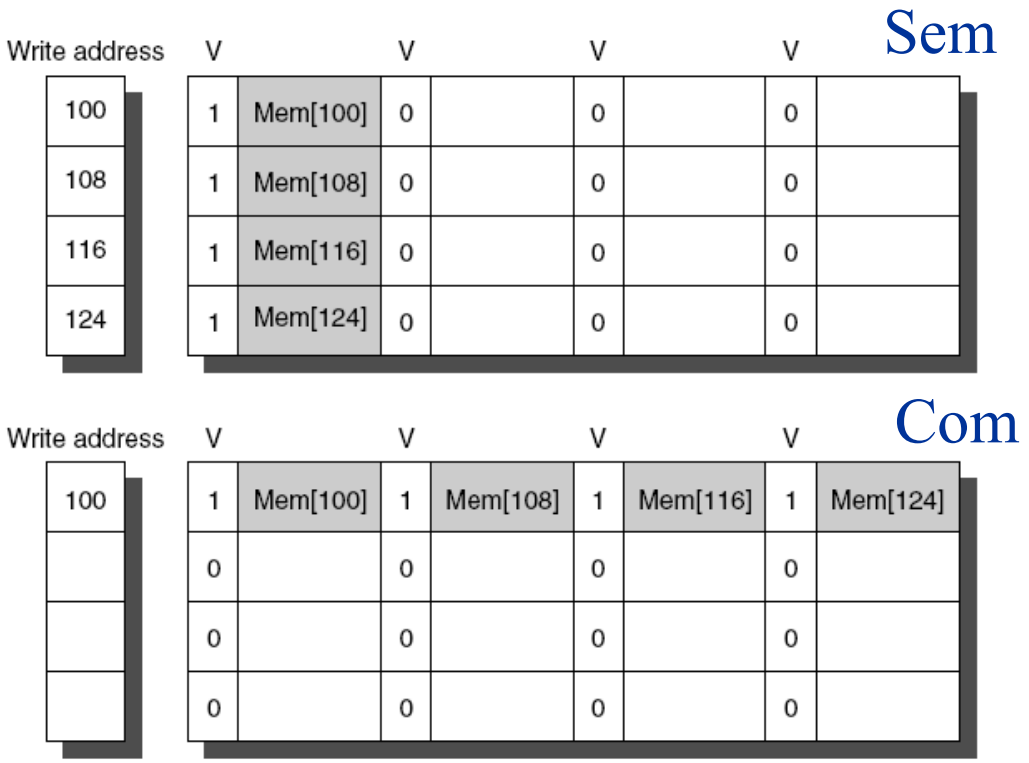


Figure 2.7 To illustrate write merging, the write buffer on top does not use it while the write buffer on the bottom does. The four writes are merged into a single buffer entry with write merging; without it, the buffer is full even though three-fourths of each entry is wasted. The buffer has four entries, and each entry holds four 64-bit words. The address for each entry is on the left, with a valid bit (V) indicating whether the next sequential 8 bytes in this entry are occupied. (Without write merging, the words to the right in the upper part of the figure would only be used for instructions that wrote multiple words at the same time.)

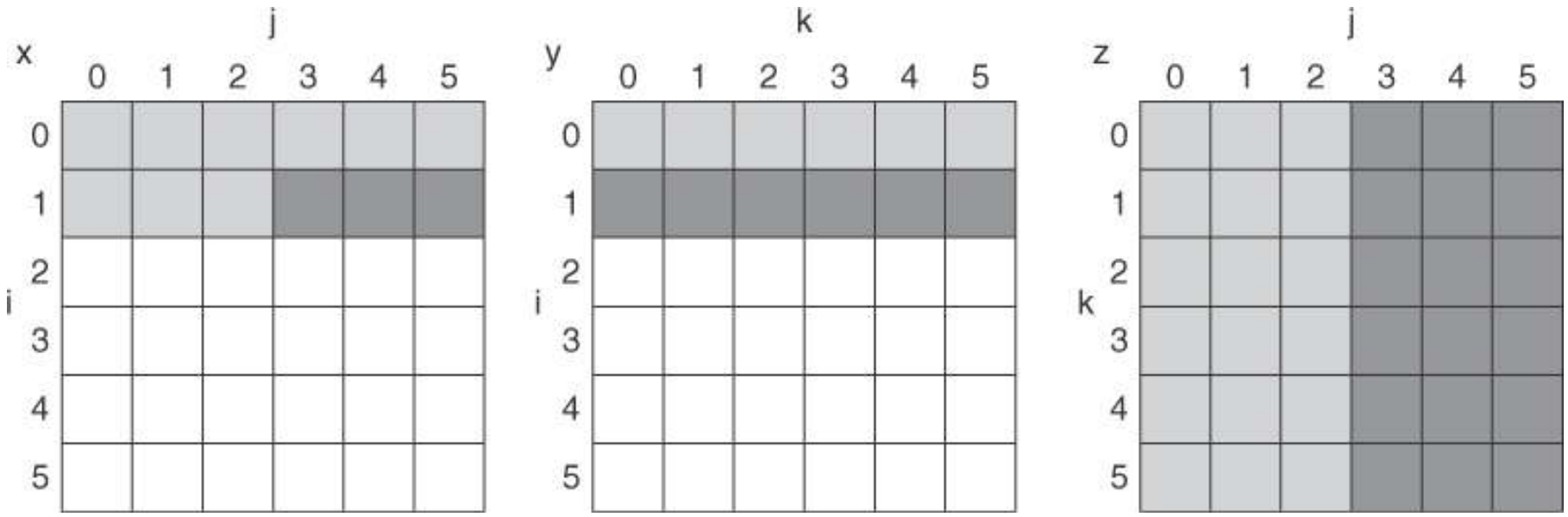
8- Compiler Optimizations

- Loop Interchange (→ localidade espacial)
  - Swap nested loops to access memory in sequential order
  - exemplo: matriz 5000 x 100, row major (x[i,j] vizinho de x[I,j+1])
    - nested loop: inner loop deve ser em j e não em i
    - senão “strides” de 100 a cada iteração no loop interno

- Blocking (→ localidade temporal)
  - Instead of accessing entire rows or columns, subdivide matrices into blocks
  - Requires more memory accesses but improves locality of accesses
  - exemplo multiplicação de matrizes NxN (só escolha apropriada de row or column major não resolve)
    - Problema é capacity miss: se a cache pode conter as 3 matrizes (X = Y x Z) então não há problemas
    - Sub blocos evitam capacity misses (no caso de matrizes grandes)

Multiplicação matrizes 6x6 sem blocking

IC-UNICAMP



X = Y x Z

Figure 2.8 A snapshot of the three arrays x, y, and z when *N* = 6 and i = 1. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. Compared to Figure 2.9, elements of y and z are read repeatedly to calculate new elements of x. The variables i, j, and k are shown along the rows or columns used to access the arrays.

IC-UNICAMP

Multiplicação matrizes 6x6 com blocking

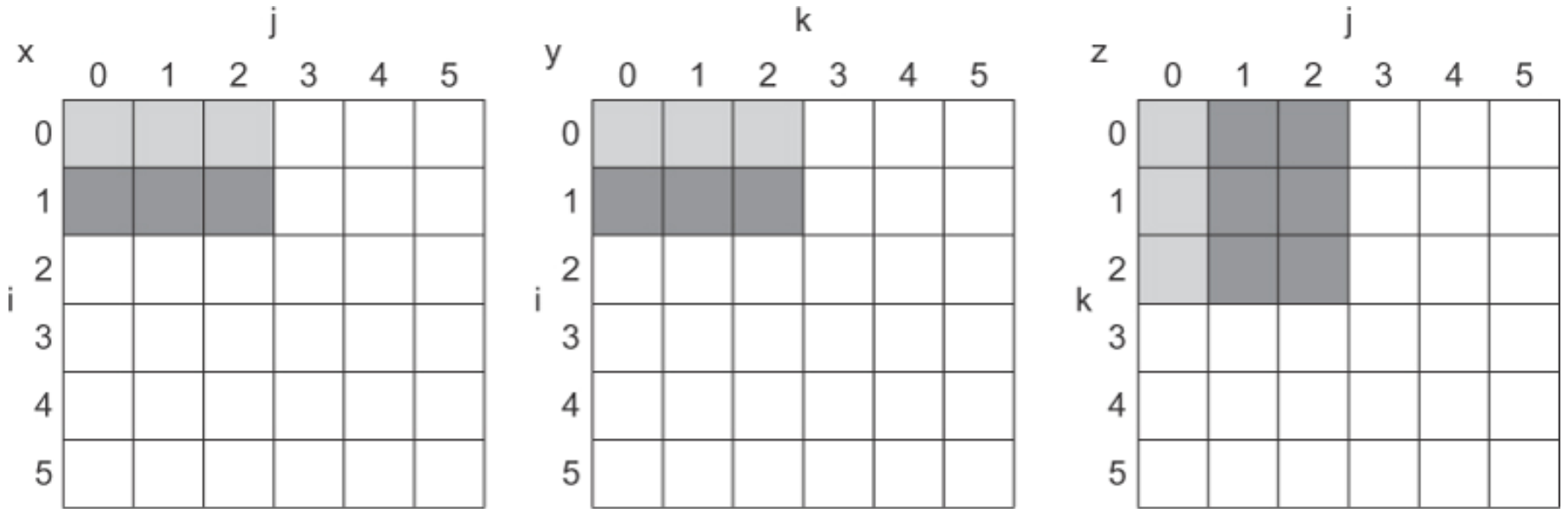


Figure 2.9 The age of accesses to the arrays x, y, and z when *B* = 3. Note that, in contrast to Figure 2.8, a smaller number of elements is accessed.

9- Hardware Prefetching

- Fetch two blocks on miss (include next sequential block)
  - para instruções (óbvio) e dados

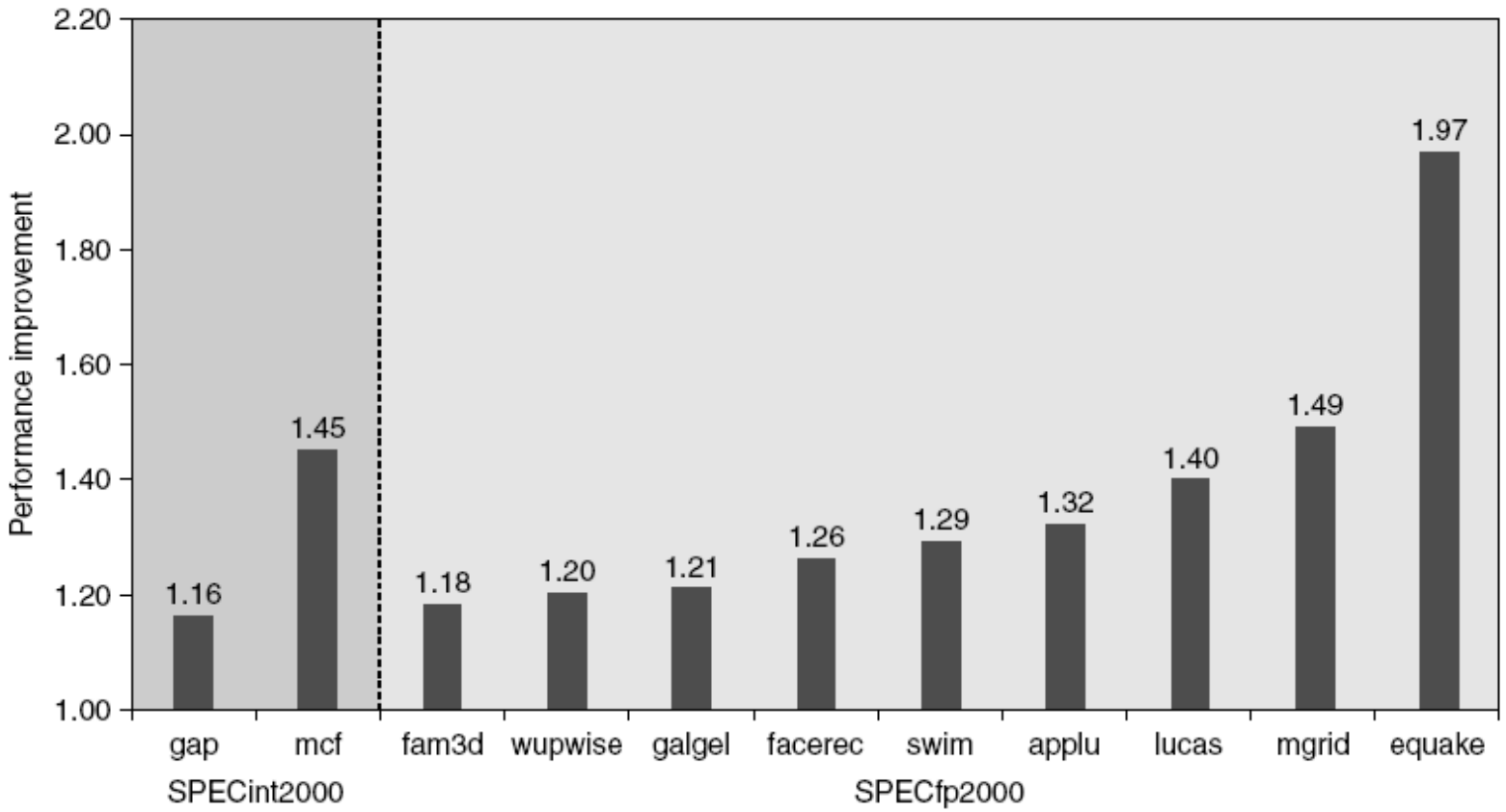


Figure 2.10 Speedup due to hardware prefetching on Intel Pentium 4 with hardware prefetching turned on for 2 of 12 SPECint2000 benchmarks and 9 of 14 SPECfp2000 benchmarks. Only the programs that benefit the most from prefetching are shown; prefetching speeds up the missing 15 SPEC benchmarks by less than 15% [Singhal 2004].

10- Compiler Prefetching

- Insert prefetch instructions before data is needed
- Non-faulting: prefetch doesn't cause exceptions (page fault or protection violation)
  - se fault → prefetch instruction transformada em no-op
- Register prefetch
  - Loads data into register
- Cache prefetch
  - Loads data into cache
- Pode ser "semantically invisible" → não afeta conteúdo de registradores e memória e não causa page fault
- Combine with loop unrolling and software pipelining

IC-UNICAMP

Exmpl p93: compiler inserted prefetch instructions

Example For the code below, determine which accesses are likely to cause data cache misses. Next, insert prefetch instructions to reduce misses. Finally, calculate the number of prefetch instructions executed and the misses avoided by prefetching. Let's assume we have an 8 KB direct-mapped data cache with 16-byte blocks, and it is a write-back cache that does write allocate. The elements of a and b are 8 bytes long since they are double-precision floating-point arrays. There are 3 rows and 100 columns for a and 101 rows and 3 columns for b. Let's also assume they are not in the cache at the start of the program.

```
for (i = 0; i < 3; i = i+1)
        for (j = 0; j < 100; j = j+1)
                a[i][j] = b[j][0] * b[j+1][0];
```

IC-UNICAMP

Exmpl p93: compiler inserted prefetch instructions (cont)

Answer The compiler will first determine which accesses are likely to cause cache misses; otherwise, we will waste time on issuing prefetch instructions for data that would be hits. Elements of a are written in the order that they are stored in memory, so a will benefit from spatial locality: The even values of j will miss and the odd values will hit. Since a has 3 rows and 100 columns, its accesses will lead to $3 \times (100/2)$, or 150 misses.

The array b does not benefit from spatial locality since the accesses are not in the order it is stored. The array b does benefit twice from temporal locality: The same elements are accessed for each iteration of i, and each iteration of j uses the same value of b as the last iteration. Ignoring potential conflict misses, the misses due to b will be for b[j+1][0] accesses when i = 0, and also the first access to b[j][0] when j = 0. Since j goes from 0 to 99 when i = 0, accesses to b lead to 100 + 1, or 101 misses.

Thus, this loop will miss the data cache approximately 150 times for a plus 101 times for b, or 251 misses.

IC-UNICAMP

Exmpl p93: compiler inserted prefetch instructions (cont)

To simplify our optimization, we will not worry about prefetching the first accesses of the loop. These may already be in the cache, or we will pay the miss penalty of the first few elements of a or b. Nor will we worry about suppressing the prefetches at the end of the loop that try to prefetch beyond the end of a (a[i][100] ... a[i][106]) and the end of b (b[101][0] ... b[107][0]). If these were faulting prefetches, we could not take this luxury. Let's assume that the miss penalty is so large we need to start prefetching at least, say, seven iterations in advance. (Stated alternatively, we assume prefetching has no benefit until the eighth iteration.) We underline the changes to the code above needed to add prefetching.

```
for (j = 0; j < 100; j = j+1) {
        prefetch(b[j+7][0]);
        /* b(j,0) for 7 iterations later */
        prefetch(a[0][j+7]);
        /* a(0,j) for 7 iterations later */
        a[0][j] = b[j][0] * b[j+1][0];};
for (i = 1; i < 3; i = i+1)
        for (j = 0; j < 100; j = j+1) {
                prefetch(a[i][j+7]);
                /* a(i,j) for +7 iterations */
                a[i][j] = b[j][0] * b[j+1][0];}
```

IC-UNICAMP

Exmpl p93: compiler inserted prefetch instructions (cont)

This revised code prefetches `a[i][7]` through `a[i][99]` and `b[7][0]` through `b[100][0]`, reducing the number of nonprefetched misses to

- 7 misses for elements `b[0][0]`, `b[1][0]`, . . . , `b[6][0]` in the first loop
- 4 misses ([7/2]) for elements `a[0][0]`, `a[0][1]`, . . . , `a[0][6]` in the first loop (spatial locality reduces misses to 1 per 16-byte cache block)
- 4 misses ([7/2]) for elements `a[1][0]`, `a[1][1]`, . . . , `a[1][6]` in the second loop
- 4 misses ([7/2]) for elements `a[2][0]`, `a[2][1]`, . . . , `a[2][6]` in the second loop

or a total of 19 nonprefetched misses. The cost of avoiding 232 cache misses is executing 400 prefetch instructions, likely a good trade-off.

IC-UNICAMP

Exmpl p94: compiler inserted prefetch instructions

Example Calculate the time saved in the example above. Ignore instruction cache misses and assume there are no conflict or capacity misses in the data cache. Assume that prefetches can overlap with each other and with cache misses, thereby transferring at the maximum memory bandwidth. Here are the key loop times ignoring cache misses: The original loop takes 7 clock cycles per iteration, the first prefetch loop takes 9 clock cycles per iteration, and the second prefetch loop takes 8 clock cycles per iteration (including the overhead of the outer for loop). A miss takes 100 clock cycles.

IC-UNICAMP

Exmpl p94: compiler inserted prefetch instructions

Answer The original doubly nested loop executes the multiply 3 × 100 or 300 times. Since the loop takes 7 clock cycles per iteration, the total is 300 × 7 or 2100 clock cycles plus cache misses. Cache misses add 251 × 100 or 25,100 clock cycles, giving a total of 27,200 clock cycles. The first prefetch loop iterates 100 times; at 9 clock cycles per iteration the total is 900 clock cycles plus cache misses. Now add 11 × 100 or 1100 clock cycles for cache misses, giving a total of 2000. The second loop executes 2 × 100 or 200 times, and at 8 clock cycles per iteration it takes 1600 clock cycles plus 8 × 100 or 800 clock cycles for cache misses. This gives a total of 2400 clock cycles. From the prior example, we know that this code executes 400 prefetch instructions during the 2000 + 2400 or 4400 clock cycles to execute these two loops. If we assume that the prefetches are completely overlapped with the rest of the execution, then the prefetch code is 27,200/4400, or 6.2 times faster.

Summary: 10 optimizations

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

2.3 Memory Technology

- Performance metrics
  - Latency is concern of cache
  - Bandwidth is concern of multiprocessors and I/O
  - Access time
    - Time between read request and when desired word arrives
  - Cycle time
    - Minimum time between unrelated requests to memory
- DRAM used for main memory, SRAM used for cache

Memory Technology

- SRAM
  - Requires low power to retain bit
  - Requires 6 transistors/bit

- DRAM
  - Must be re-written after being read
  - Must also be periodically refeshed
    - Every ~ 8 ms
    - Each row can be refreshed simultaneously
    - Goal: tempo gasto em refreshing ≅ 5% tempo total
  - One transistor/bit
  - Address lines are multiplexed:
    - Upper half of address: row access strobe (RAS)
    - Lower half of address: column access strobe (CAS)

DRAM: organização interna

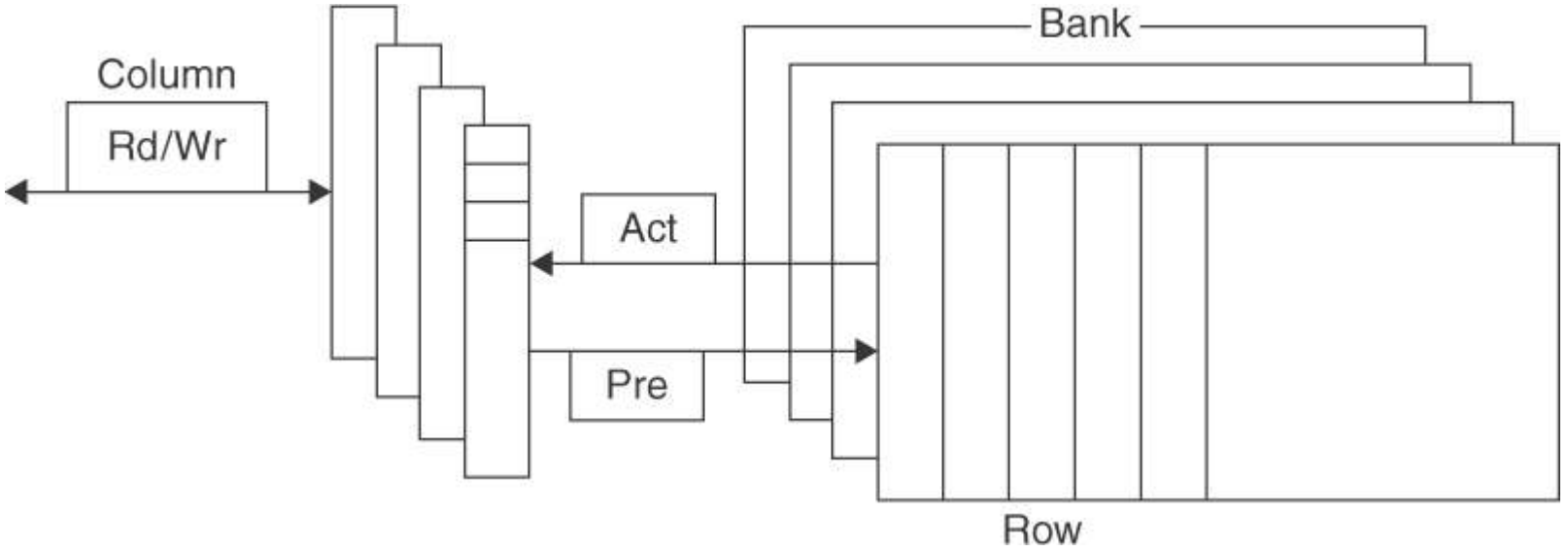


Figure 2.12 Internal organization of a DRAM. Modern DRAMs are organized in banks, typically four for DDR3. Each bank consists of a series of rows. Sending a PRE (precharge) command opens or closes a bank. A row address is sent with an Act (activate), which causes the row to transfer to a buffer. When the row is in the buffer, it can be transferred by successive column addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR3) or by specifying a block transfer and the starting address. Each command, as well as block transfers, are synchronized with a clock.

Memory Technology

- Amdahl:
  - Memory capacity should grow linearly with processor speed
  - Unfortunately, memory capacity and speed has not kept pace with processors (fig 2.13). Aumento anual: 4x (até1996) e 2x depois

- Some optimizations:
  - Multiple accesses to same row (buffer pode manter linha armazenada)
  - Synchronous DRAM → SDRAM
    - Added clock to DRAM interface (tratou overhead de sincronização nas assincr.)
    - Burst mode with critical word first (enviar pacote de dados)
  - Wider interfaces (4bits; depois em 2010 DDR2 e DDR3 → 16 bits)
  - Double data rate (DDR): data transfer on rising and falling edges of clock
  - Multiple banks (2-8) on each DRAM device: vantagens de interleaving e gestão de energia
    - endereço: banco, row, column. Acesso subsequente no mesmo banco é mais rápido

Memory Evolution

			Row access strobe (RAS)			
Production year	Chip size	DRAM Type	Slowest DRAM (ns)	Fastest DRAM (ns)	Column access strobe (CAS)/ data transfer time (ns)	Cycle time (ns)
1980	64K bit	DRAM	180	150	75	250
1983	256K bit	DRAM	150	120	50	220
1986	1M bit	DRAM	120	100	25	190
1989	4M bit	DRAM	100	80	20	165
1992	16M bit	DRAM	80	60	15	120
1996	64M bit	SDRAM	70	50	12	110
1998	128M bit	SDRAM	70	50	10	100
2000	256M bit	DDR1	65	45	7	90
2002	512M bit	DDR1	60	40	5	80
2004	1G bit	DDR2	55	35	5	70
2006	2G bit	DDR2	50	30	2.5	60
2010	4G bit	DDR3	36	28	1	37
2012	8G bit	DDR3	30	24	0.5	31

Figure 2.13 Times of fast and slow DRAMs vary with each generation. (Cycle time is defined on page 95.) Performance improvement of row access time is about 5% per year. The improvement by a factor of 2 in column access in 1986 accompanied the switch from NMOS DRAMs to CMOS DRAMs. The introduction of various burst transfer modes in the mid-1990s and SDRAMs in the late 1990s has significantly complicated the calculation of access time for blocks of data; we discuss this later in this section when we talk about SDRAM access time and power. The DDR4 designs are due for introduction in mid- to late 2012. We discuss these various forms of DRAMs in the next few pages.

Memory Optimizations

- DDR:
  - Packaging DIMM
    - Nome = DIMM bw; ex: DIMM PC2100 → 133Mz x 2 x8B = 2100 MB/s
    - Nome DDR = bits/sec; ex DDR de 133Mz → DDR266
  - DDR2
    - Lower power (2.5 V -> 1.8 V)
    - Higher clock rates (266 MHz, 333 MHz, 400 MHz)
  - DDR3
    - 1.5 V e 800 MHz
  - DDR4
    - 1-1.2 V e 1600 MHz
- GDDR5 is graphics memory based on DDR3

Tipos DDR, nome e velocidades

Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /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	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1066–1600	2133–3200	DDR4-3200	17,056–25,600	PC25600

Figure 2.14 Clock rates, bandwidth, and names of DDR DRAMS and DIMMs in 2010. Note the numerical relationship between the columns. The third column is twice the second, and the fourth uses the number from the third column in the name of the DRAM chip. The fifth column is eight times the third column, and a rounded version of this number is used in the name of the DIMM. Although not shown in this figure, DDRs also specify latency in clock cycles as four numbers, which are specified by the DDR standard. For example, DDR3-2000 CL 9 has latencies of 9-9-9-28. What does this mean? With a 1 ns clock (clock cycle is one-half the transfer rate), this indicate 9 ns for row to columns address (RAS time), 9 ns for column access to data (CAS time), and a minimum read time of 28 ns. Closing the row takes 9 ns for precharge but happens only when the reads from that row are finished. In burst mode, transfers occur on every clock on both edges, when the first RAS and CAS times have elapsed. Furthermore, the precharge in not needed until the entire row is read. DDR4 will be produced in 2012 and is expected to reach clock rates of 1600 MHz in 2014, when DDR5 is expected to take over. The exercises explore these details further.

Memory Optimizations

- Graphics memory:
  - Achieve 2-5 X bandwidth per DRAM vs. DDR3
    - Wider interfaces (32 vs. 16 bit)
    - Higher clock rate
      - Possible because they are attached via soldering instead of socketted DIMM modules

- Reducing power in SDRAMs:
  - Lower voltage
  - Uso de bancos: acesso a uma linha de um único banco por RD
  - Low power mode (ignores clock, continues to refresh)

Memory Power Consumption

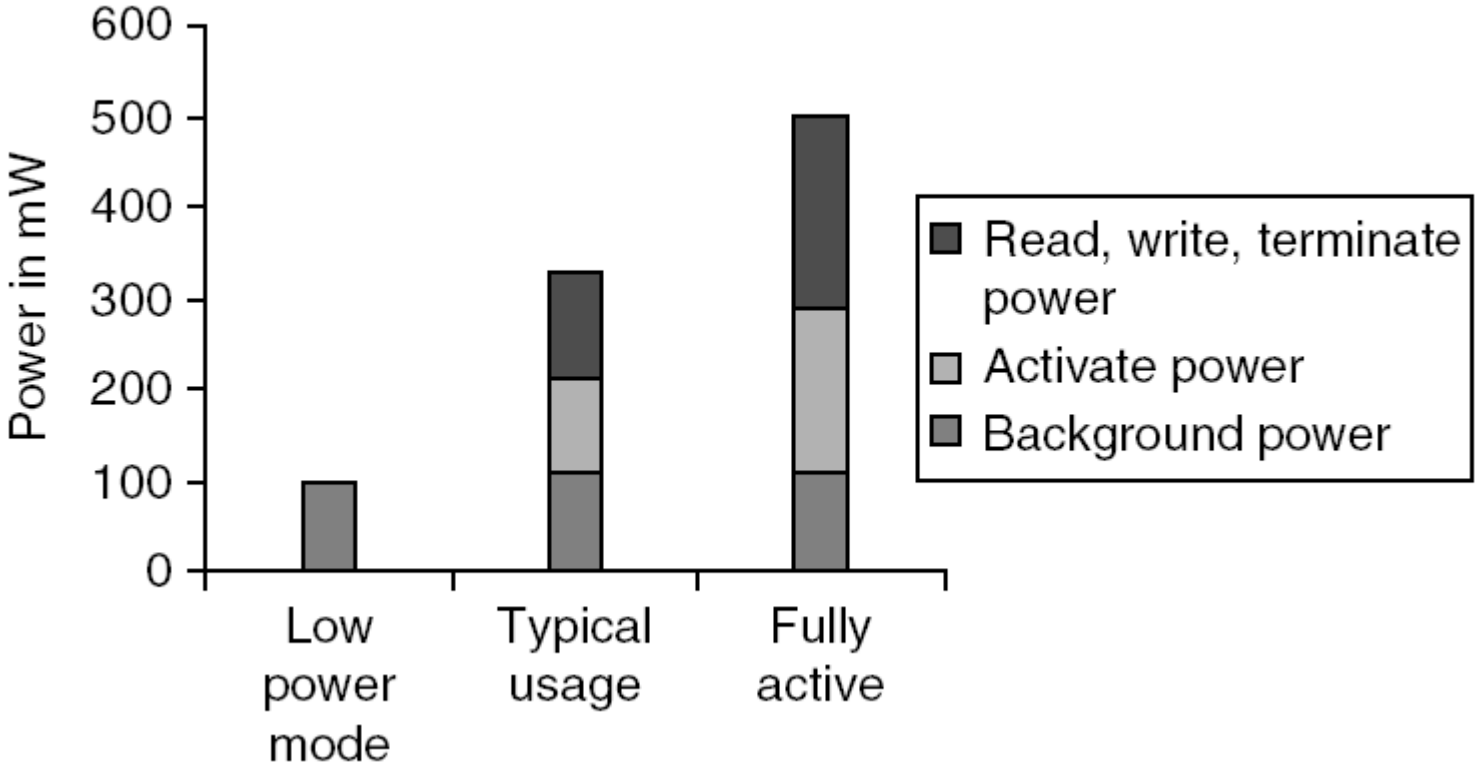


Figure 2.15 Power consumption for a DDR3 SDRAM operating under three conditions: low power (shutdown) mode, typical system mode (DRAM is active 30% of the time for reads and 15% for writes), and fully active mode, where the DRAM is continuously reading or writing when not in precharge. Reads and writes assume bursts of 8 transfers. These data are based on a Micron 1.5V 2Gb DDR3-1066.

Flash Memory

- Type of EEPROM
- Must be erased (in blocks) before being overwritten
- Non volatile
- Consome pouca (ou nenhuma) energia se inativa
- Limited number of write cycles (+- 100 000)
  - há medidas para distribuir uso e evitar desgaste localizado
- Cheaper than SDRAM, more expensive than disk
- Slower than SRAM, faster than disk

Memory Dependability

- Memory is susceptible to cosmic rays
- *Soft errors*: dynamic errors
  - Detected and fixed by error correcting codes (ECC)
- *Hard errors*: permanent errors
  - Use sparse rows to replace defective rows

- Chipkill: a RAID-like error recovery technique

- Exemplo: 10000 servidores, 4GB/servidor → MTBF
  - somente com paridade: 17 minutos
  - ECC: 7.5 horas
  - Chipkill: 2 meses

IC-UNICAMP

2.4 Virtual Memory

- Protection via virtual memory
  - Keeps processes in their own memory space
- Role of architecture:
  - Provide user mode and supervisor mode
  - Protect certain aspects of CPU state
  - Provide mechanisms for switching between user mode and supervisor mode
  - Provide mechanisms to limit memory accesses
  - Provide TLB to translate addresses

Virtual Machines

- Velhas VMs: 1960´s, nos mainframes IBM
- Ignorada posteriormente, volta agora porque:
  - Supports isolation and security
  - Maior segurança do que a obtida com OS tradicionais
  - Sharing a computer among many unrelated users
  - Enabled by raw speed of processors, making the overhead more acceptable
- Allows different ISAs and operating systems to be presented to user programs (emulation)
  - “System Virtual Machines”: matching ISA (VM and host hardware)
    - usuário: ilusão de ter uma máquina inteira sob seu controle
  - SVM software is called “virtual machine monitor

IC-UNICAMP

Impact of VMs on Virtual Memory

- Each guest OS maintains its own set of page tables
  - VMM adds a level of memory between physical and virtual memory called “real memory”
  - Guest OS maps virtual memory to real memory (its page table)
  - VMM page table maps real memory to physical memory
  - To avoid extra level of indirection, VMM maintains shadow page table that maps guest virtual addresses to physical addresses
    - Requires VMM to detect guest’s changes to its own page table
    - Occurs naturally if accessing the page table pointer is a privileged operation

2.5 Crosscutting issues: design of hierar.

IC-UNICAMP

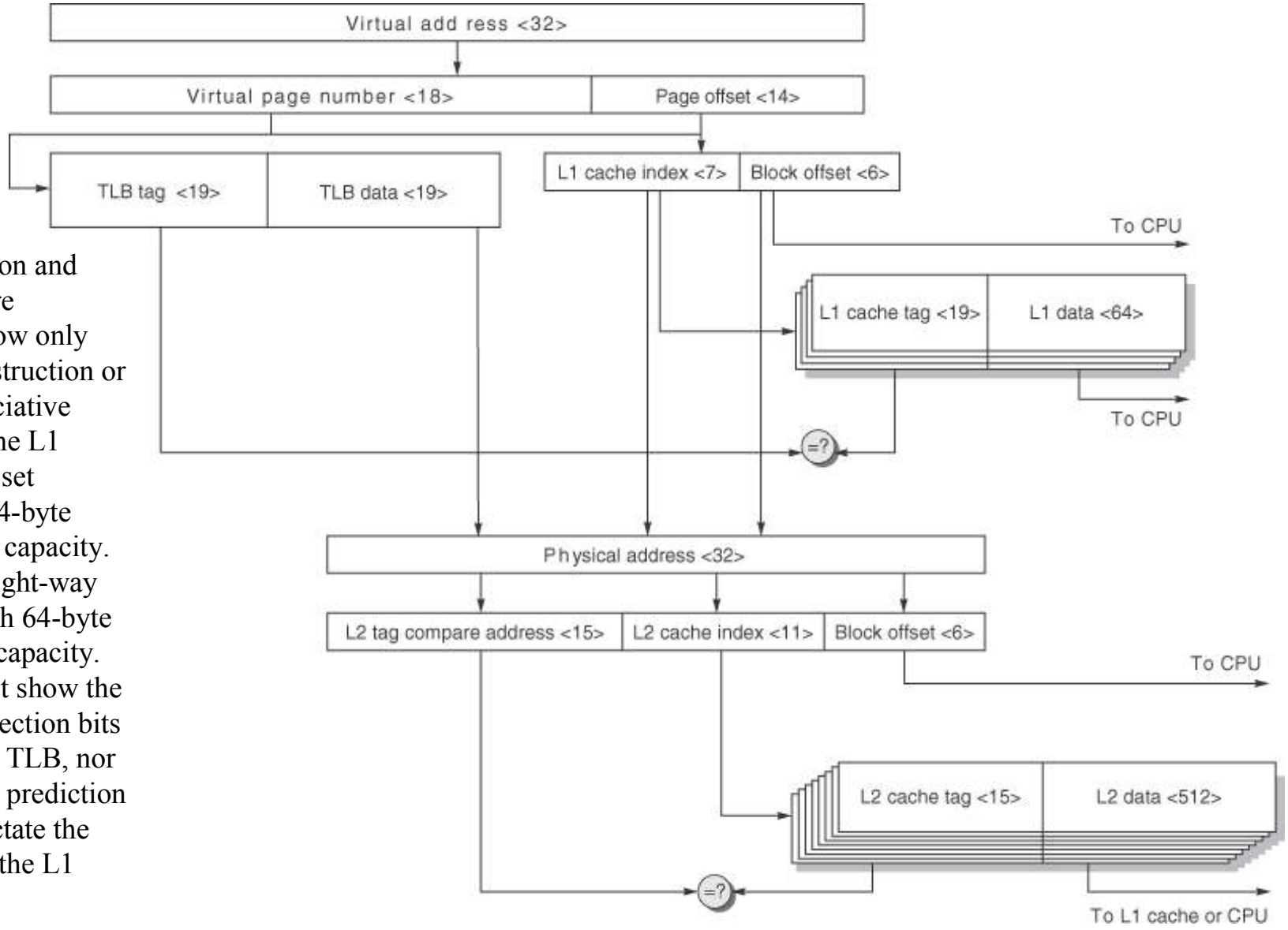
- Protection and ISA
  - Proteção: trabalho conjunto do OS e arquitetura
  - Mas pode haver interferência do ISA
  - Ex: problemas com Interrupt Enable e virtualização no 80x86
- Coherency of cached data in I/O operations
  - I/O ⇔ cache ou I/O ⇔ Memória ?
  - Se cache → processador vê dado atualizado, mas I/O interfere na cache (acesso ou substituição) → stall
  - Muitos sistemas preferem I/O ⇔ Memória (que serve como I/O buffer). Se write through:
    - operações de IO out veem dado atualizado
    - operações de IO in : a) flush página da cache; ou b) marcar pag de IO como uncacheable

IC-UNICAMP

2.6 Putting it all Together

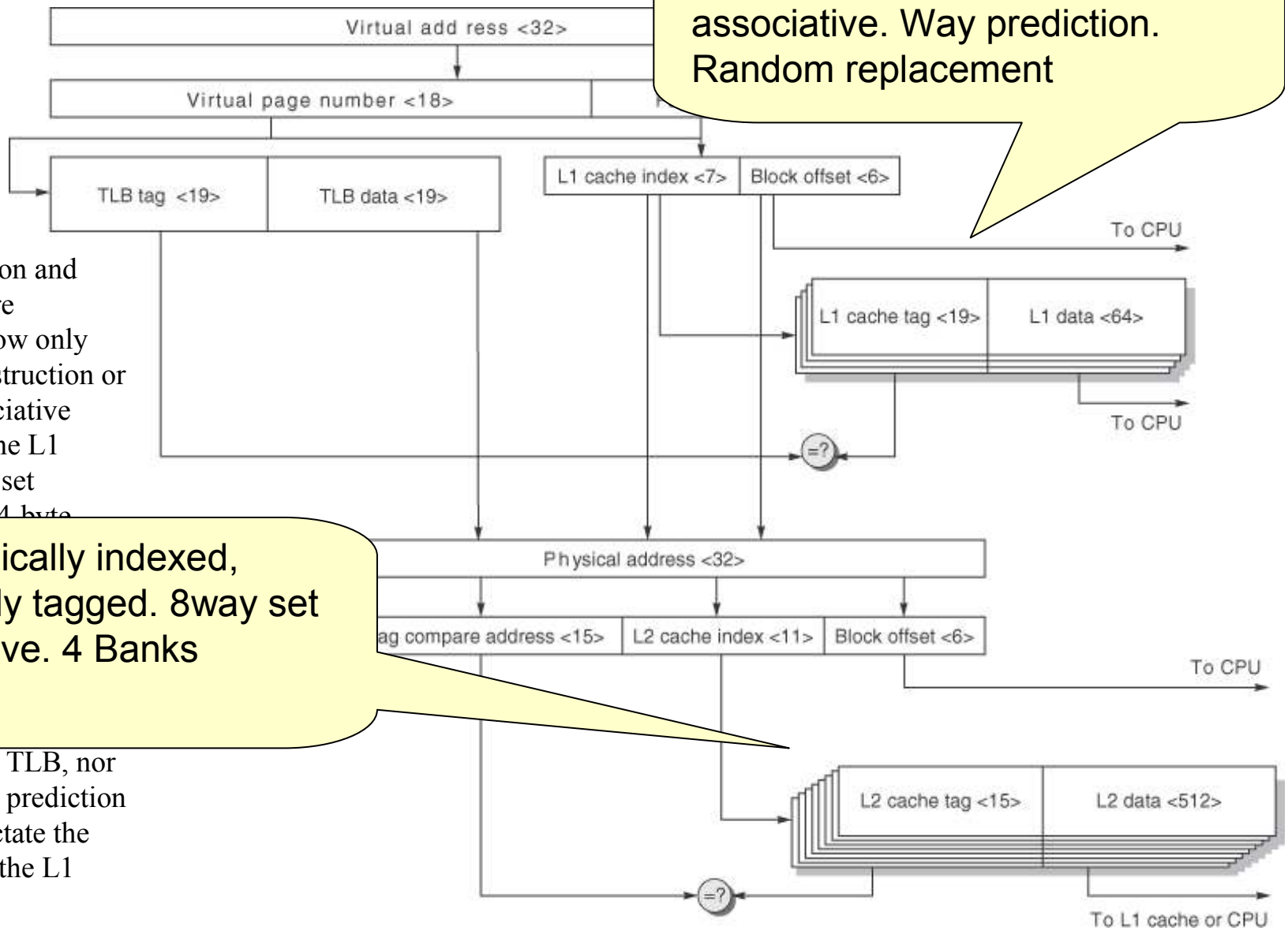
- Exemplos de Hierarquias de Memória
  - ARM Cortex-A8
  - Intel Core i7

Fig 2.16: o ARM Cortex A8



Since the instruction and data hierarchies are symmetric, we show only one. The TLB (instruction or data) is fully associative with 32 entries. The L1 cache is four-way set associative with 64-byte blocks and 32 KB capacity. The L2 cache is eight-way set associative with 64-byte blocks and 1 MB capacity. This figure doesn't show the valid bits and protection bits for the caches and TLB, nor the use of the way prediction bits that would dictate the predicted bank of the L1 cache.

Fig 2.16: o ARM C



L1: Virtually indexed, Physically tagged. 16 or 32 KB. 4way set associative. Way prediction. Random replacement

L2: Physically indexed, Physically tagged. 8way set associative. 4 Banks

Since the instruction and data hierarchies are symmetric, we show only one. The TLB (instruction or data) is fully associative with 32 entries. The L1 cache is four-way set associative with 64-byte blo
Th
set
blo
Th
vali
for the caches and TLB, nor the use of the way prediction bits that would dictate the predicted bank of the L1 cache.

Fig 2.17: Miss Rate (A8)

The data miss rate for ARM with a 32 KB L1 and the global data miss rate for a 1 MB L2 using the integer Minnespec benchmarks are significantly affected by the applications. Applications with larger memory footprints tend to have higher miss rates in both L1 and L2. Note that the L2 rate is the global miss rate, that is counting all references, including those that hit in L1. Mcf is known as a cache buster.

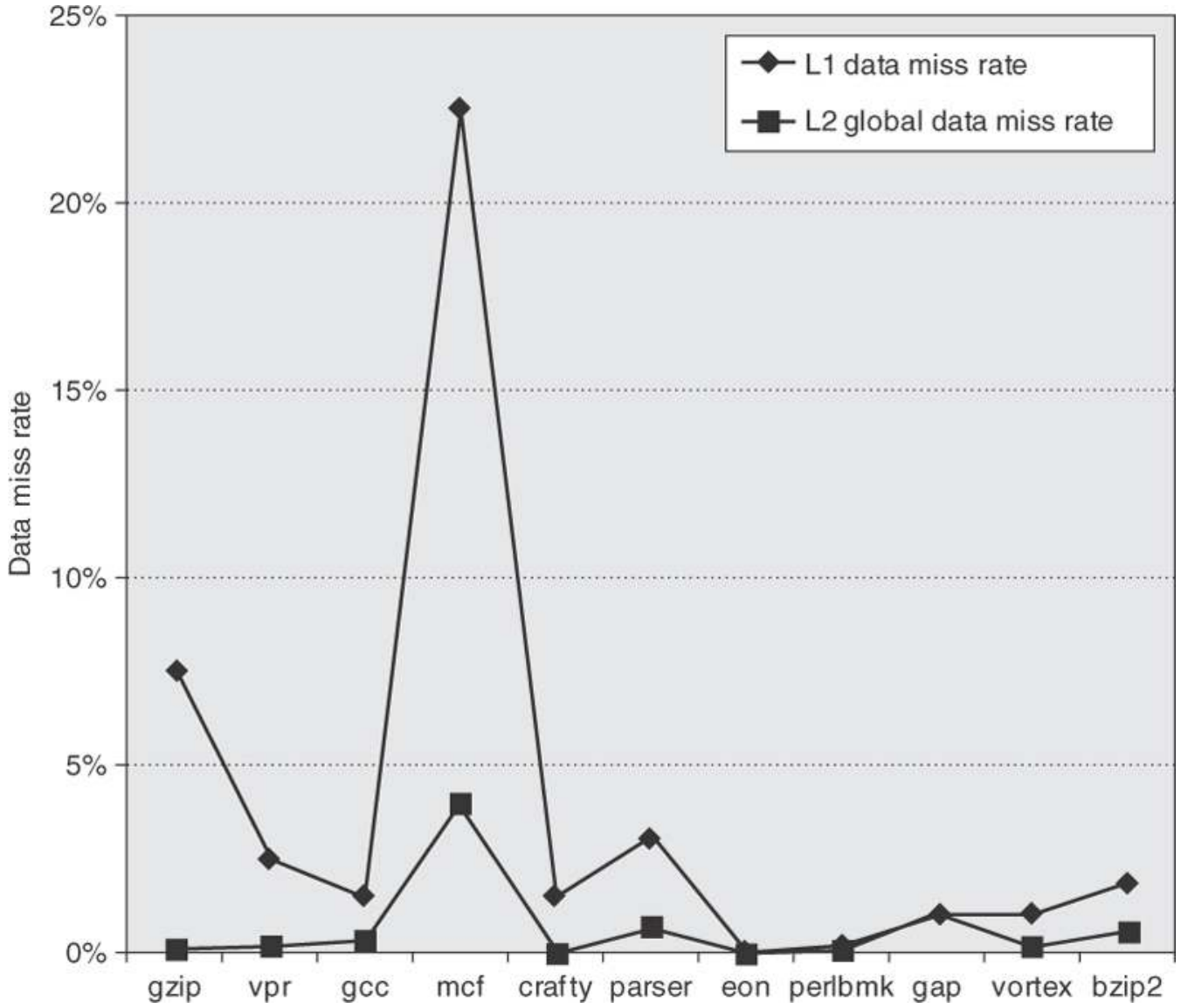
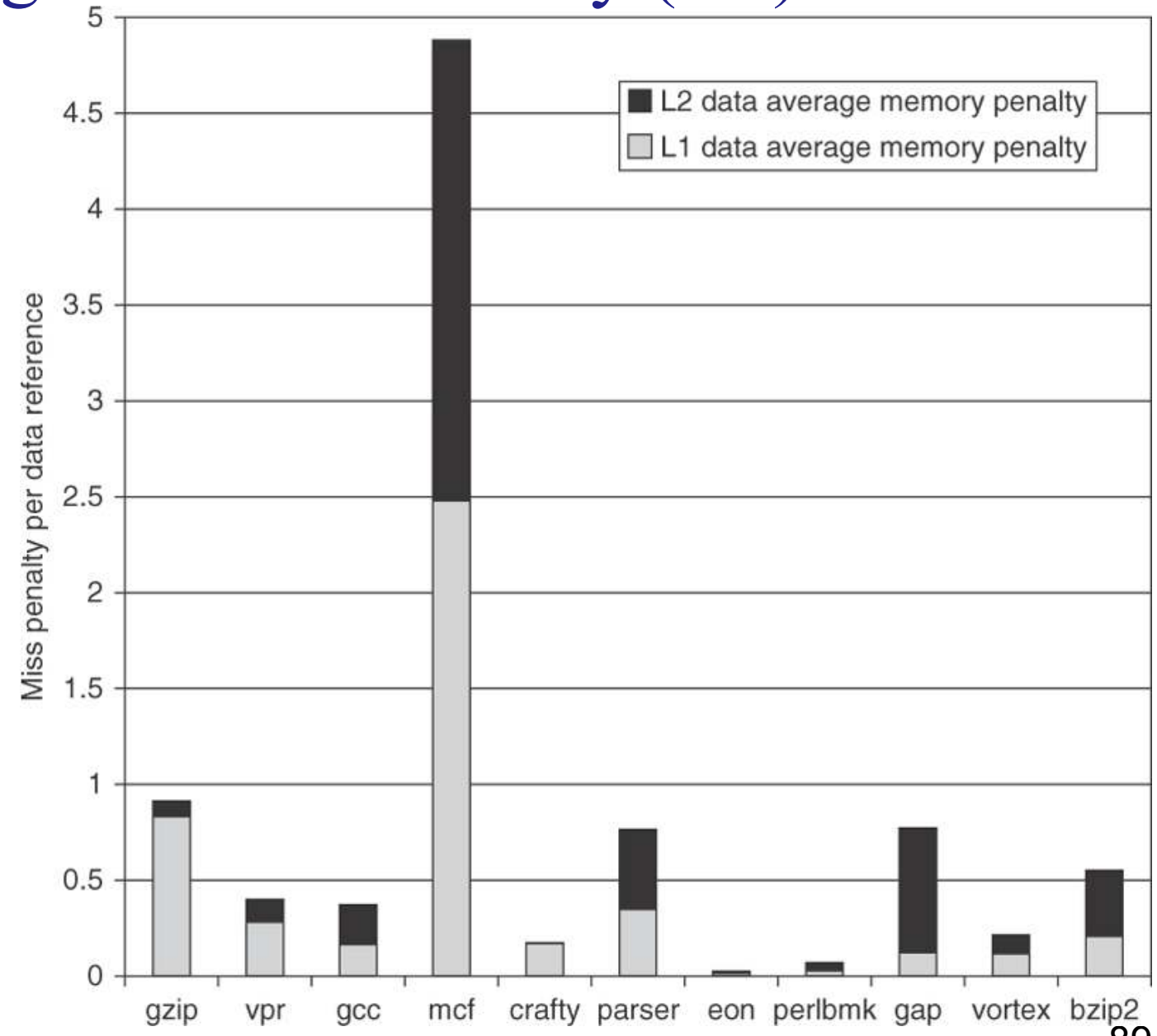


Fig 2.18: Miss Penaly (A8)

The average memory access penalty per data memory reference coming from L1 and L2 is shown for the ARM processor when running Minniespec. Although the miss rates for L1 are significantly higher, the L2 miss penalty, which is more than five times higher, means that the L2 misses can contribute significantly.



IC-UNICAMP

O Intel i7: características

TLB

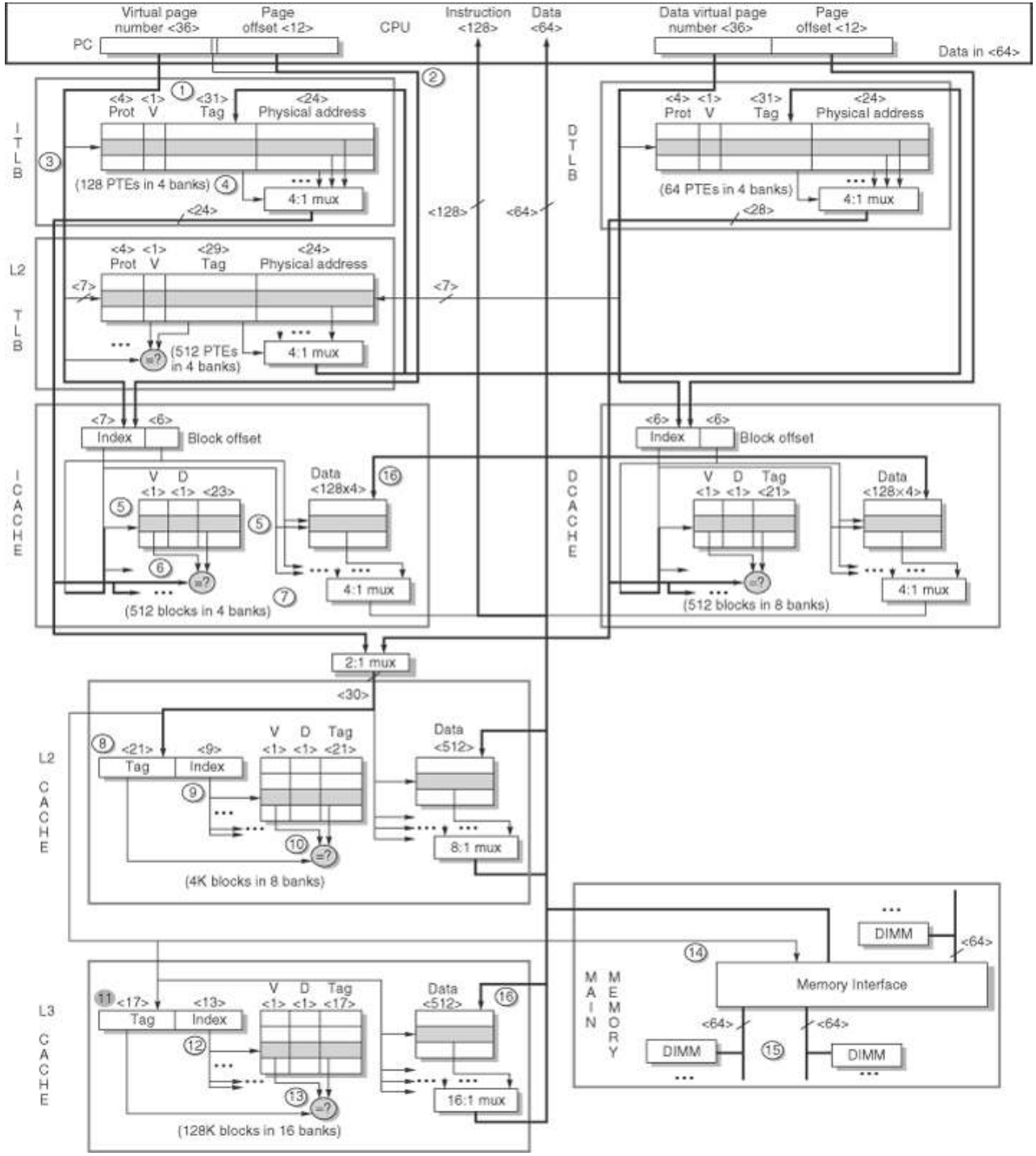
Characteristic	Instruction TLB	Data TLB	Second Level TLB
Size	128	64	512
Associativity	4-way	4-way	4-way
Replacement	pseudo LRU	pseudo LRU	pseudo LRU
Access Latency	1 cycle	2 cycle	6 cycle
Miss	7 cycles	8 cycles	>100 (page table)

Caches

Characteristic	L1	L2	L3
Size	32 kb (I and D)	256 KB	2 MB per core
Associativity	4-way (I), 8-way (D)	8-way	16-way
Access Latency	4 cycles, pipelined	10 cycles,	35 cycles,
Replacement	pseudo LRU	pseudo LRU	pseudo LRU

IC-UNICAMP

Fig 2.21: Hier. de Mem. do Intel i7



The Intel i7 memory hierarchy and the steps in both instruction and data access. We show only reads for data. Writes are similar, in that they begin with a read (since caches are write back). Misses are handled by simply placing the data in a write buffer, since the L1 cache is not write allocated.

Fig 2.22: Data cache miss rate

The L1 data cache miss rate for 17 SPECCPU2006 benchmarks is shown in two ways: relative to the actual loads that complete execution successfully and relative to all the references to L1, which also includes prefetches, speculative loads that do not complete, and writes, which count as references, but do not generate misses.

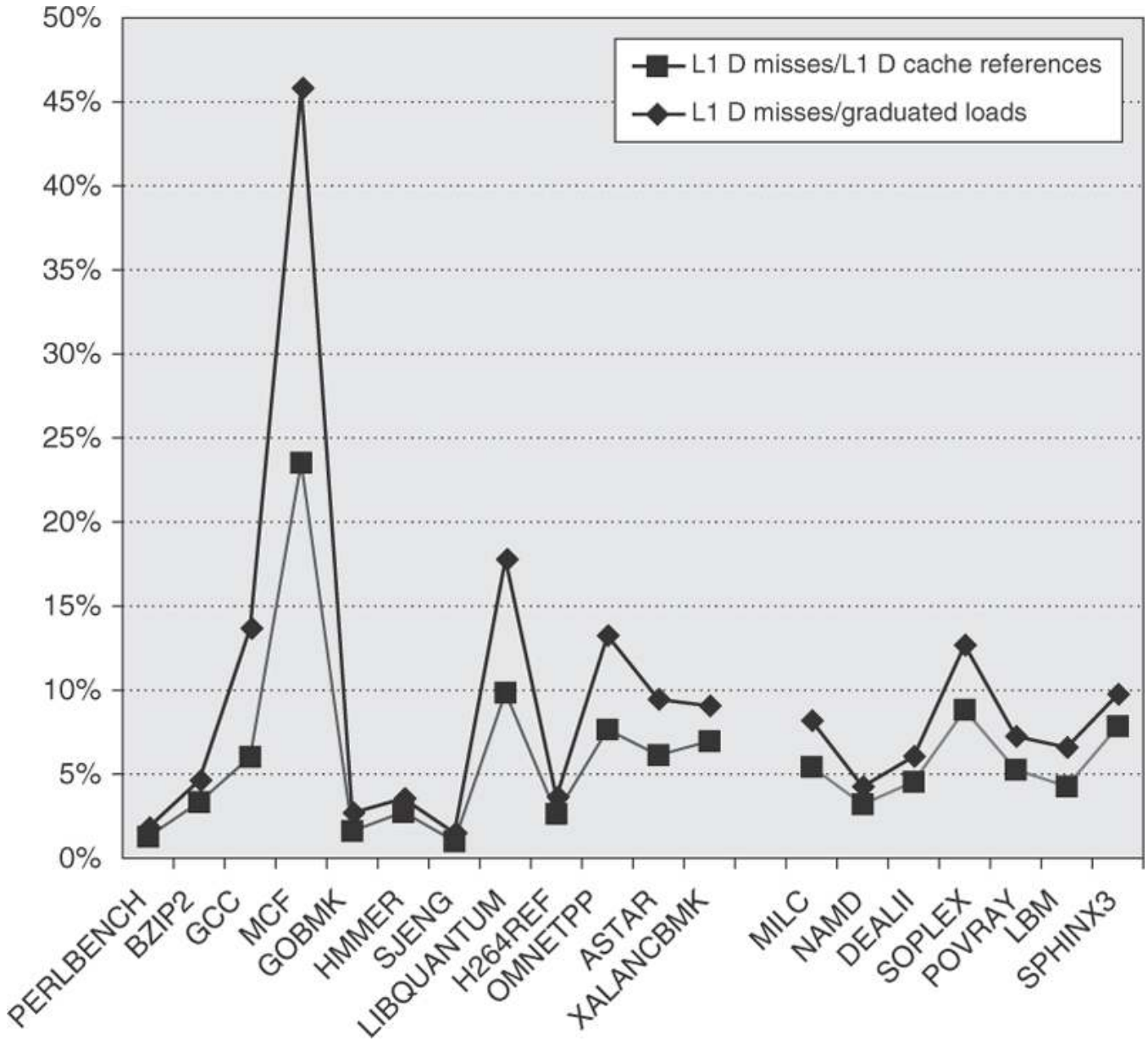
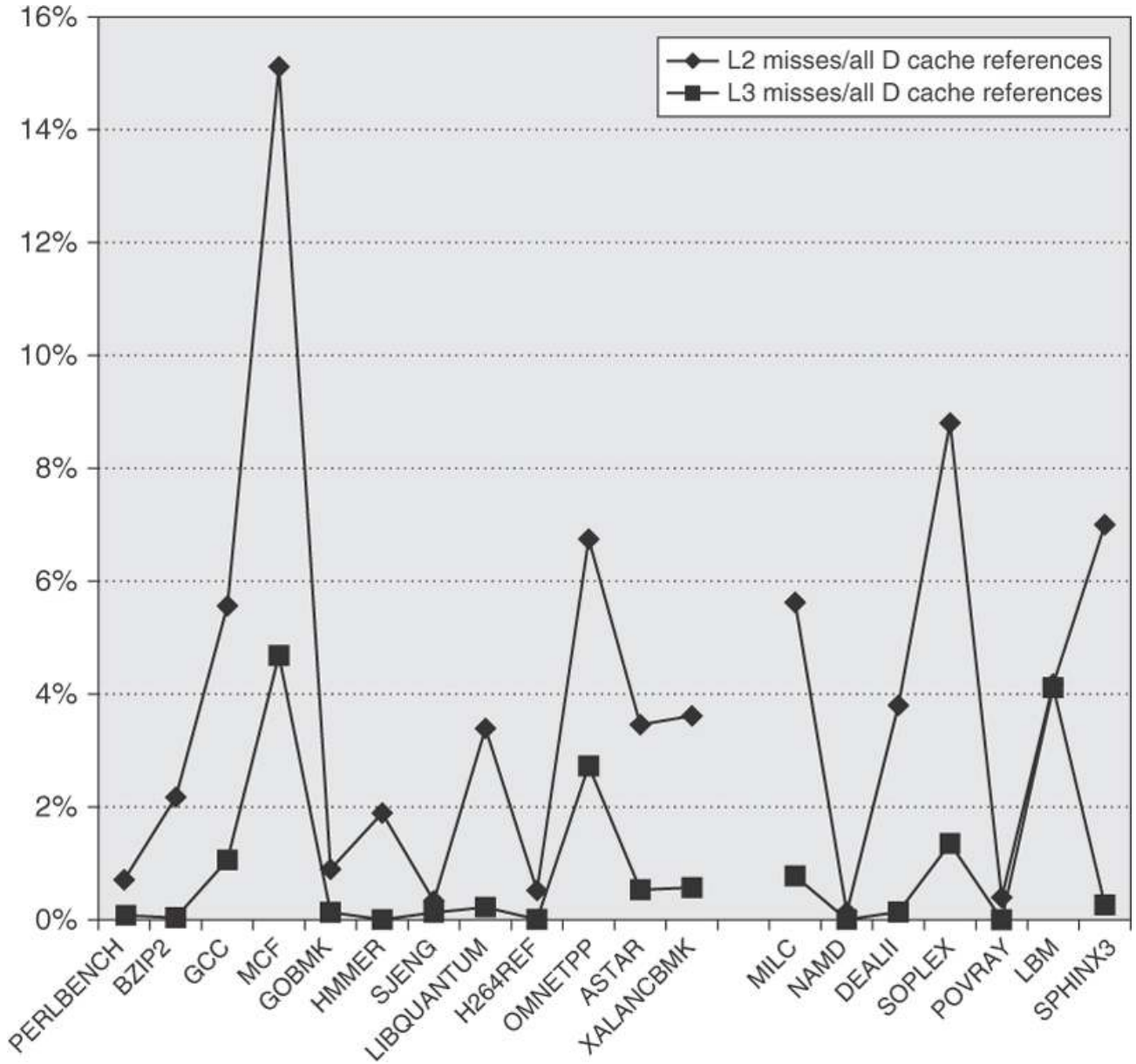


Fig 2.24: Miss Rate de L2 e L3

The L2 and L3 data cache miss rates for 17 SPECCPU2006 benchmarks are shown relative to all the references to L1, which also includes prefetches, speculative loads that do not complete, and program–generated loads and stores.



IC-UNICAMP

Fallacies and Pitfalls

- (ver texto)

Predizer cache performance de um programa com base em outro

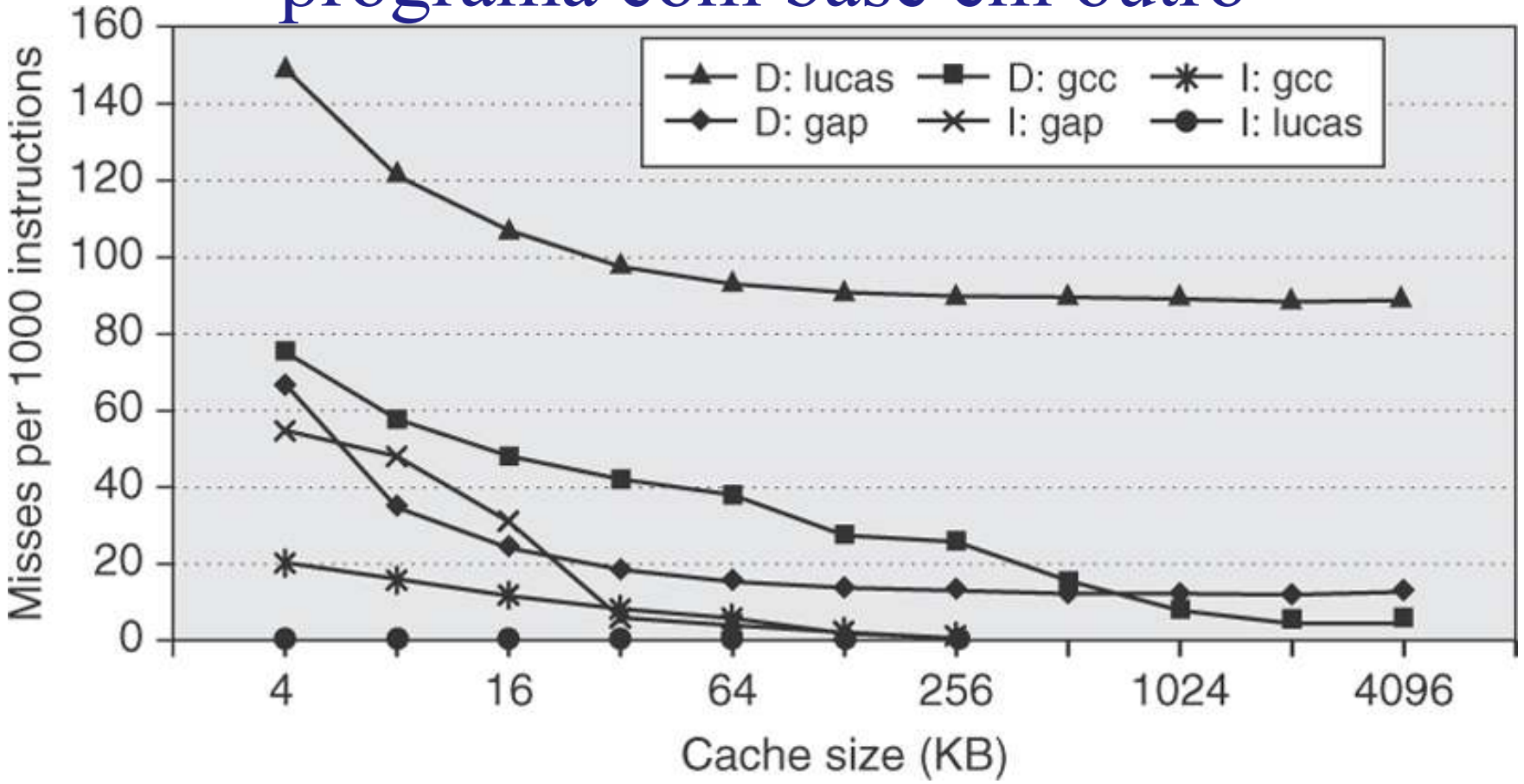


Figure 2.26 Instruction and data misses per 1000 instructions as cache size varies from 4 KB to 4096 KB. Instruction misses for gcc are 30,000 to 40,000 times larger than lucas, and, conversely, data misses for lucas are 2 to 60 times larger than gcc. The programs gap, gcc, and lucas are from the SPEC2000 benchmark suite.

Misses: basear em nº limitado de instruções

IC-UNICAMP

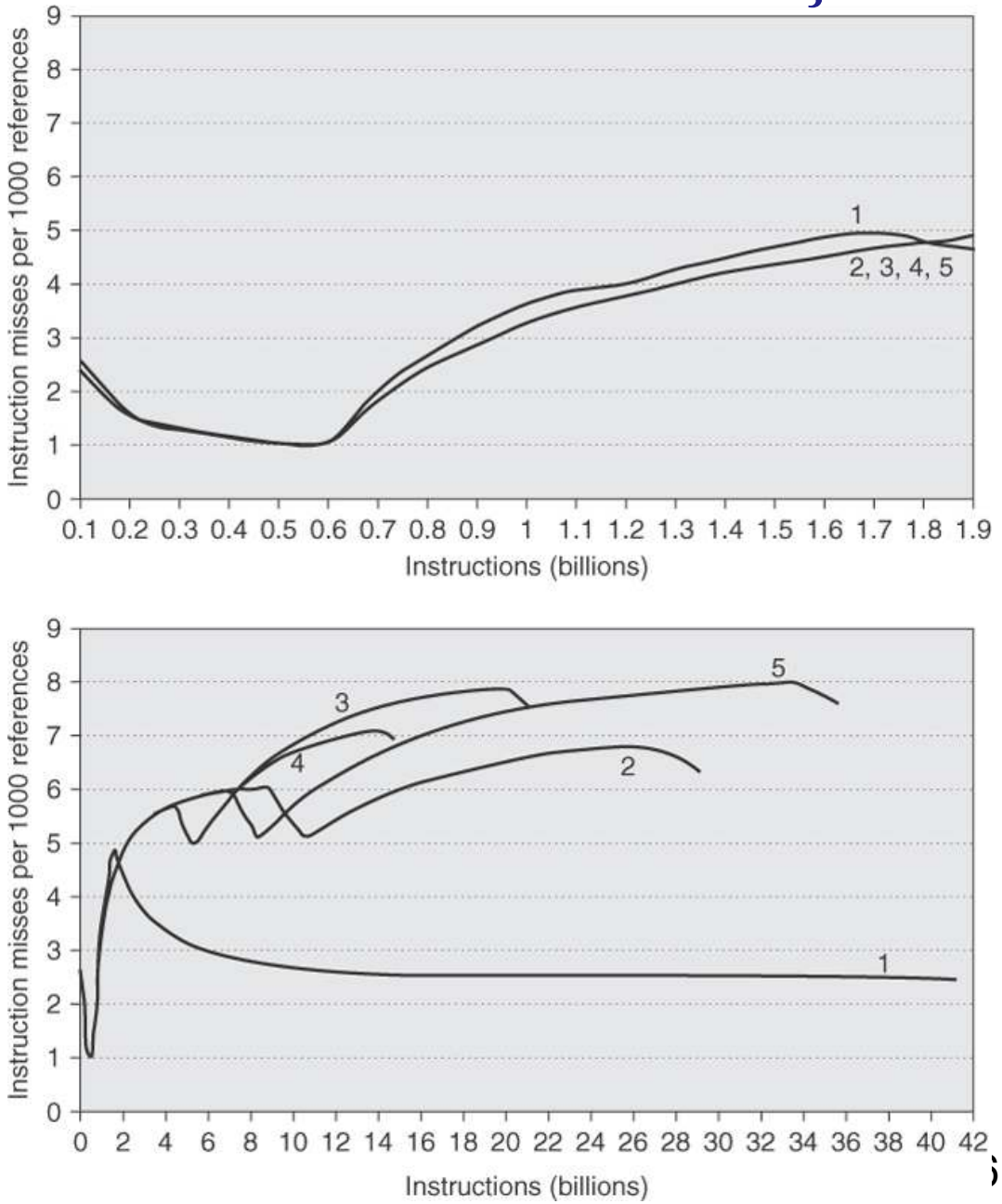


Figure 2.27 Instruction misses per 1000 references for five inputs to the perl benchmark from SPEC2000. There is little variation in misses and little difference between the five inputs for the first 1.9 billion instructions. Running to completion shows how misses vary over the life of the program and how they depend on the input. The top graph shows the running average misses for the first 1.9 billion instructions, which starts at about 2.5 and ends at about 4.7 misses per 1000 references for all five inputs. The bottom graph shows the running average misses to run to completion, which takes 16 to 41 billion instructions depending on the input. After the first 1.9 billion instructions, the misses per 1000 references vary from 2.4 to 7.9 depending on the input. The simulations were for the Alpha processor using separate L1 caches for instructions and data, each two-way 64 KB with LRU, and a unified 1 MB direct-mapped L2 cache.