中国科学技术大学
University of Science and Technology of China

计算机体系结构

Topic IV: Memory Hierarchy

第4章　存储层次结构设计

- 存储层次结构
- Cache基本知识
- 基本的Cache优化方法
- 高级的Cache优化方法
- 存储器技术与优化
- 虚拟存储器－基本原理

Cache 性能分析

- **CPU time = (CPU execution clock cycles +**
  Memory stall clock cycles) x clock cycle time

- **Memory stall clock cycles =**
  (Reads x Read miss rate x Read miss penalty +
  Writes x Write miss rate x Write miss penalty)

- **Memory stall clock cycles =**
  Memory accesses x Miss rate x Miss penalty

- **Different measure: AMAT**

  Average Memory Access time (AMAT) =
  Hit Time + (Miss Rate x Miss Penalty)
- **Note:** ***memory hit time is included in execution cycles.***

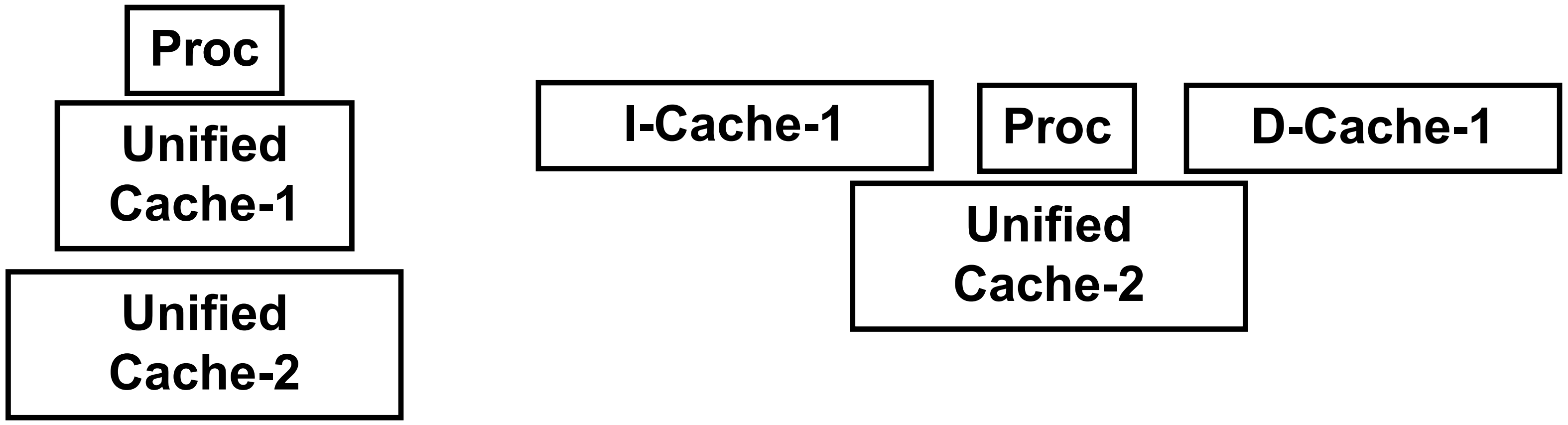
性能分析举例

- Suppose a processor executes at
  - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control
- Miss Behavior:
  - 10% of memory operations get 50 cycle miss penalty
  - 1% of instructions get same miss penalty
- 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 + .5) cycle/ins = 3.1
- 65% (2/3.1) of the time the processor is stalled waiting for memory!
- AMAT=(1/1.3)x[1+0.01x50]+(0.3/1.3)x[1+0.1x50]=2.54

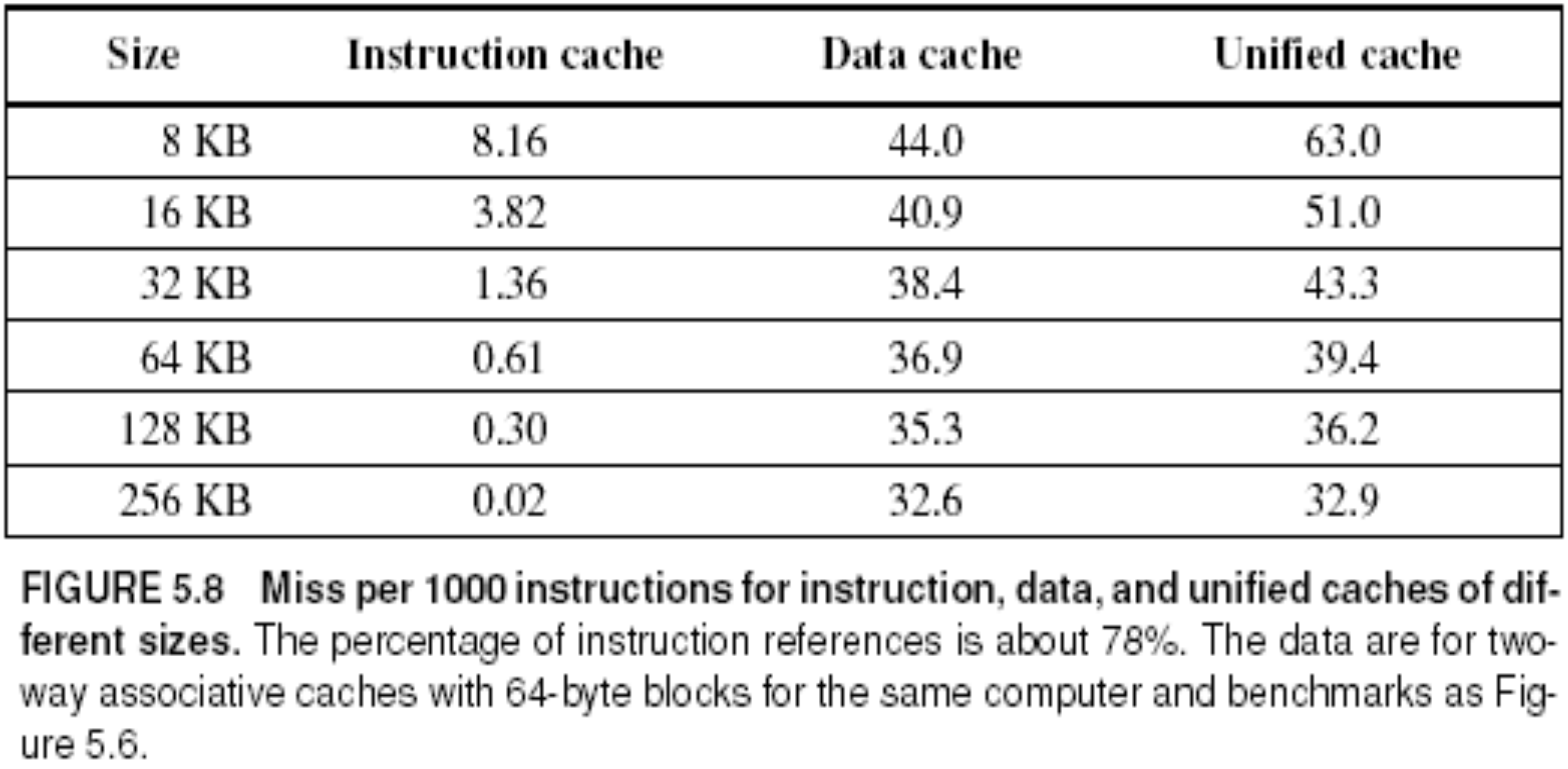
Example: Harvard Architecture

- **Unified vs Separate I&D (Harvard)**



- **Statistics (given in H&P):**
  - 16KB I&D: Inst miss rate=0.64%, Data miss rate=6.47%
  - 32KB unified: Aggregate miss rate=1.99%
- **Which is better (ignore L2 cache)?**
  - Assume 33% data ops ⇒ 75% accesses from instructions (1.0/1.33)
  - hit time=1, miss time=50
  - Note that data hit has 1 stall for unified cache (only one port)

- **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**



Size	Instruction cache	Data cache	Unified cache
8 KB	8.16	44.0	63.0
16 KB	3.82	40.9	51.0
32 KB	1.36	38.4	43.3
64 KB	0.61	36.9	39.4
128 KB	0.30	35.3	36.2
256 KB	0.02	32.6	32.9

FIGURE 5.8 Miss per 1000 instructions for instruction, data, and unified caches of different sizes. The percentage of instruction references is about 78%. The data are for two-way associative caches with 64-byte blocks for the same computer and benchmarks as Figure 5.6.

- 以顺序执行的计算机 UltraSPARC III为例. 假设Cache失效开销为100 clock cycles，所有指令忽略存储器停顿需要1个cycle, Cache失效可以用两种方式给出

（1）假设平均失效率为2%，平均每条指令访存1.5次

（2）假设每1000条指令cache失效次数为30次

分别基于上述两种条件计算处理器的性能

- 结论：

- CPUtime = IC * (1 + 2%*1.5*100) * T = IC * 4 * T

(1) CPI越低，固定周期数的Cache失效开销的相对影响就越大

(2) 在计算CPI时，失效开销的单位是时钟周期数。因此，即使两台计算机的存储层次完全相同，时钟频率较高的CPU的失效开销会较大，其CPI中存储器停顿部分也就较大。

因此 Cache对于低CPI，高时钟频率的CPU来说更加重要

考虑不同组织结构的Cache对性能的影响:

B-19例题：直接映像Cache 和两路组相联Cache，试问他们对CPU性能的影响？先求平均访存时间，然后再计算CPU性能。分析时请用以下假设：

（1）理想Cache(命中率为100%）情况下CPI 为1.0，时钟周期为0.35ns，平均每条指令访存1.4次

（2）两种Cache容量均为128KB，块大小都是64B

（3）采用组相联时，由于多路选择器的存在，时钟周期增加到原来的1.35倍

（4）两种结构的失效开销都是65ns (在实际应用中，应取整为整数个时钟周期)

（5）命中时间为1个cycle，128KB直接映像Cache的失效率为2.1%，相同容量的两路组相联Cache 的失效率为1.9%

- AMAT_da =1+0.021*65=2.365
- AMAT_sa = 1+0.019*65=2.235
- CPI_da = 1+1.4*0.021*65=2.911
- CPI_sa = 1+1.4*0.019*65=2.729
- CPU Time_da = 2.911*0.35=1.0188ns
- CPU Time_sa = 2.729*0.35*1.35=1.2894ns

Summary of performance equations in this chapter

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

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

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

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

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

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

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

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

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

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

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

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

FIGURE 5.9 Summary of performance equations in this chapter. The first equation calculates with cache index size, but the rest help evaluates performance. The final two equations deal with multilevel caches, is explained early in the next section. They are included here to help make the figure a useful reference.

改进Cache 性能的方法

- 平均访存时间＝命中时间＋失效率×失效开销
- 从上式可知，基本途径
  - 降低失效率
  - 减少失效开销
  - 缩短命中时间

基本Cache优化方法

- **降低失效率**
  - 1、增加Cache块的大小
  - 2、增大Cache容量
  - 3、提高相联度
- **减少失效开销**
  - 4、多级Cache
  - 5、使读失效优先于写失效
- **缩短命中时间**
  - 6、避免在索引缓存期间进行地址转换

降低失效率

Cache失效的原因 可分为三类 3C

- **强制性失效 (Compulsory or cold)**
  - 第一次访问某一块，只能从下一级Load，也称为冷启动或首次访问失效
- **容量失效（Capacity)**
  - 如果程序执行时，所需块由于容量不足，不能全部调入Cache， 则当某些块被替换后，若又重新被访问，就会发生失效。
  - 可能会发生“抖动”现象
- **冲突失效（Conflict (collision))**
  - 组相联和直接相联的副作用
  - 若太多的块映像到同一组（块）中，则会出现该组中某个块被别的块替换（即使别的组或块有空闲位置)，然后又被重新访问的情况，这就属于冲突失效

各种类型的失效率

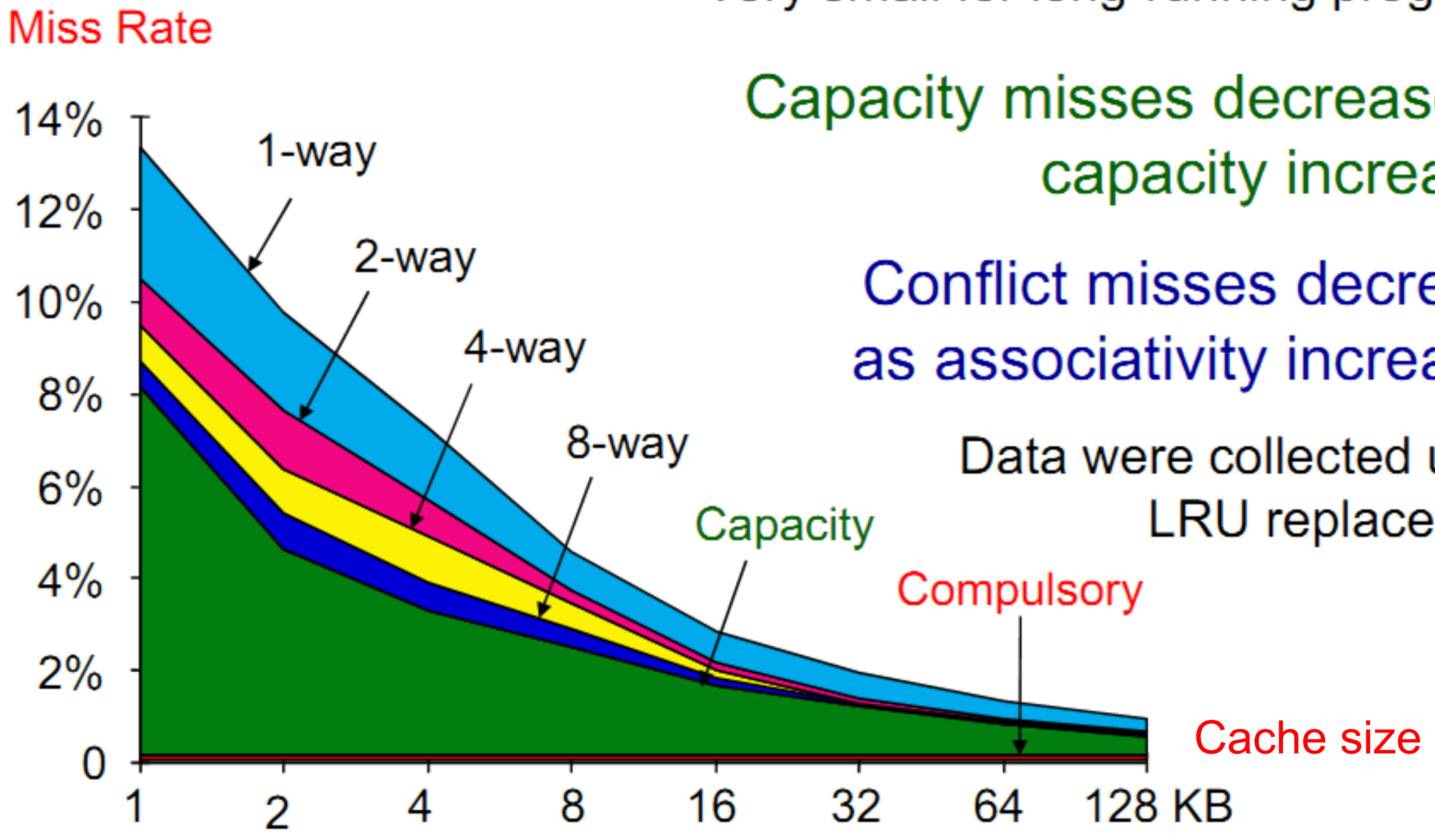
Compulsory misses are independent of cache size

Very small for long-running programs

Capacity misses decrease as capacity increases

Conflict misses decrease as associativity increases

Data were collected using LRU replacement



从统计规律中得到的一些结果

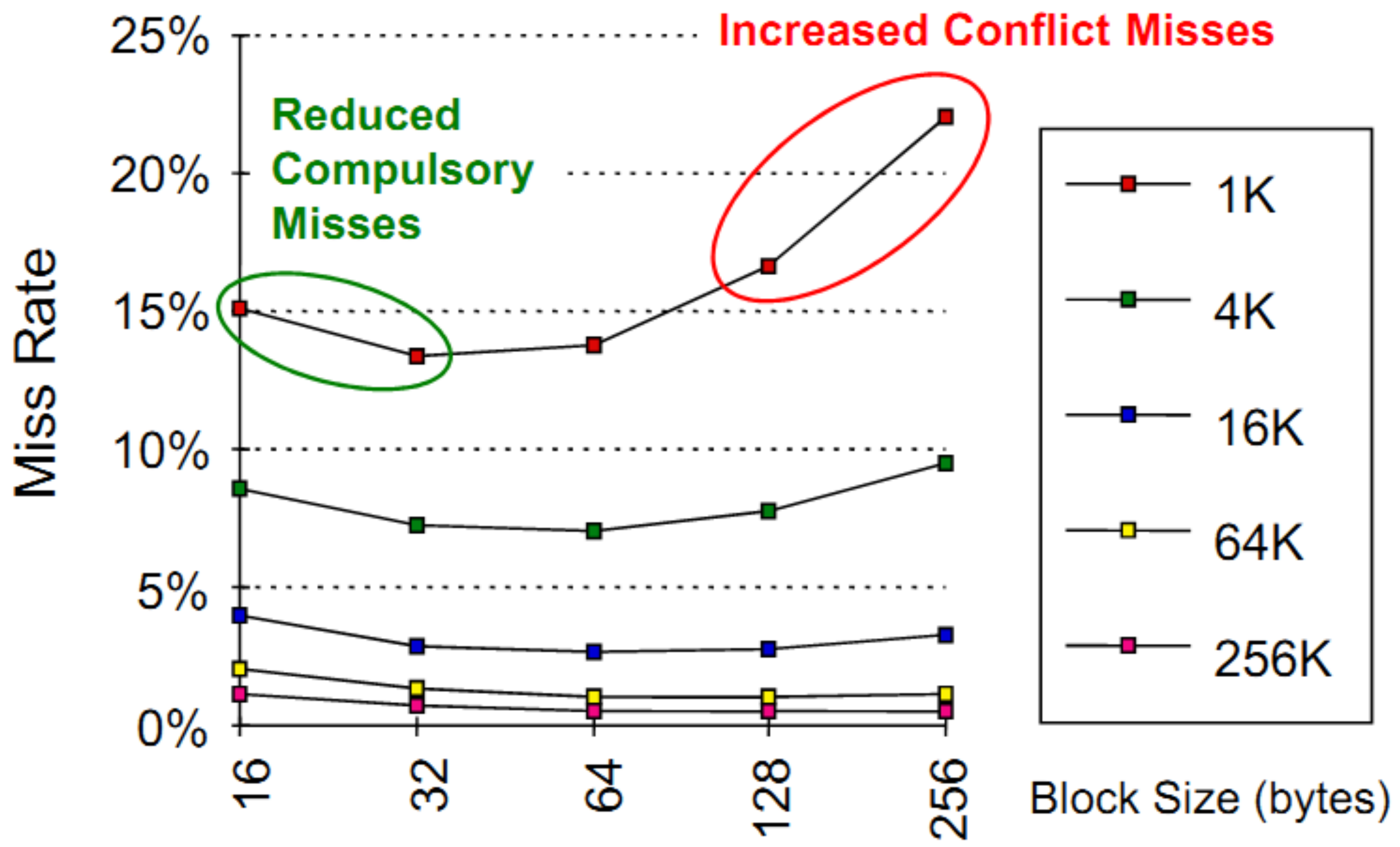
- 相联度越高，冲突失效就越小
- 强制性失效和容量失效不受相联度的影响
- 强制性失效不受Cache容量的影响
- 容量失效随着容量的增加而减少
- 符合2:1Cache经验规则
  - 即大小为N的直接映象Cache的失效率约等于大小为N/2的两路组相联的Cache失效率

减少3C的方法

从统计规律可知

- **增大Cache容量**
  - 对冲突和容量失效的减少有利
- **增大块**
  - 减缓强制性失效
  - 可能会增加冲突失效（因为在容量不变的情况下，块的数目减少了）
- **通过预取可帮助减少强制性失效**
  - 必须小心不要把你需要的东西换出去
  - 需要预测比较准确（对数据较困难，对指令相对容易）

增加块的大小



64-byte blocks are common in L1 caches

128-byte blocks are common in L2 caches

块大小、容量对失效率的影响

- **Given: miss rates for different cache sizes & block sizes**

Block Size	Cache = 4 KB	Cache = 16 KB	Cache = 64 KB	Cache = 256 KB
16 bytes	8.57%	3.94%	2.04%	1.09%
32 bytes	7.24%	2.87%	1.35%	0.70%
64 bytes	7.00%	2.64%	1.06%	0.51%
128 bytes	7.78%	2.77%	1.02%	0.49%
256 bytes	9.51%	3.92%	1.15%	0.49%

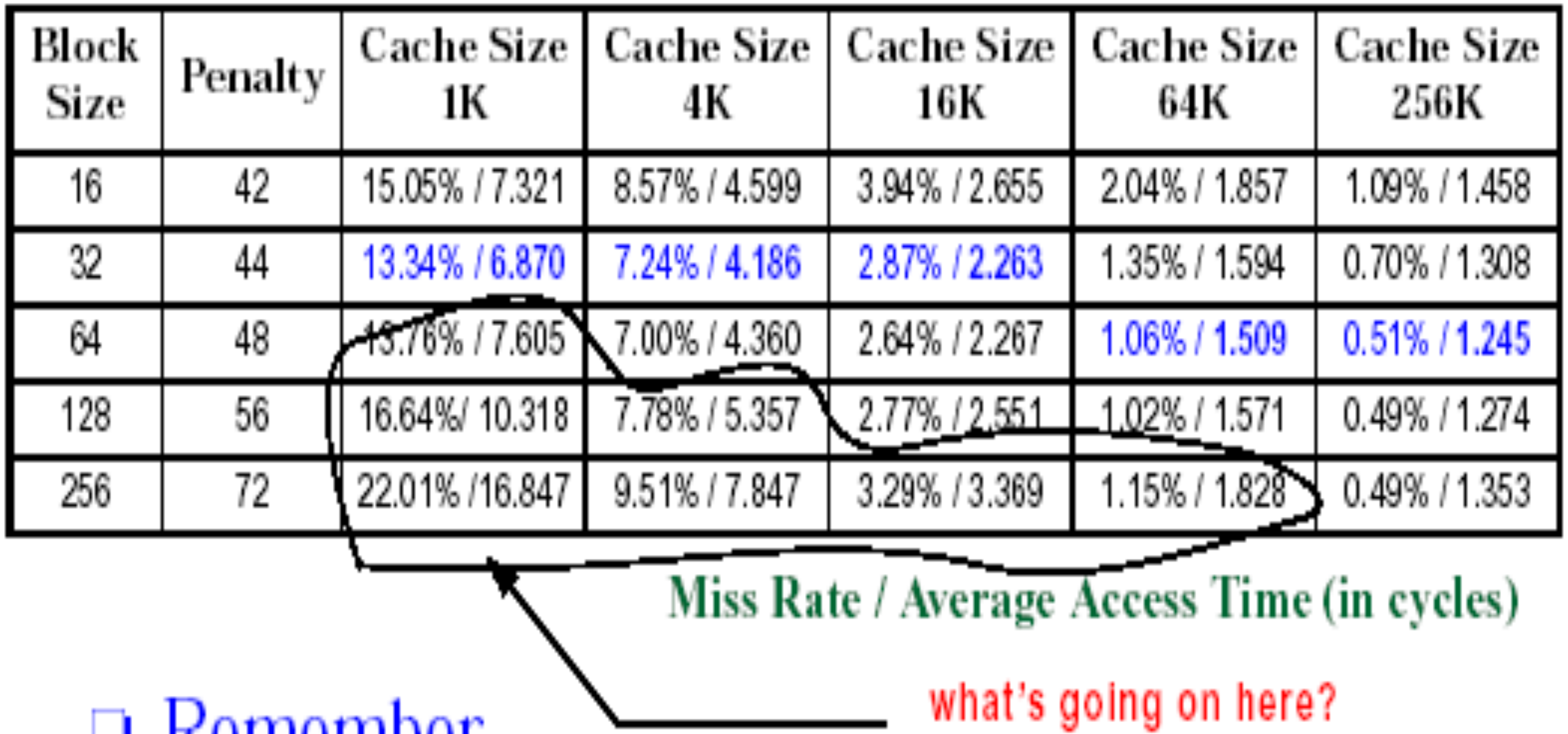
- **Memory latency = 80 cycles + 1 cycle per 8 bytes**
  - Latency of 16-byte block = 80 + 2 = 82 clock cycles
  - Latency of 32-byte block = 80 + 4 = 84 clock cycles
  - Latency of 256-byte block = 80 + 32 = 112 clock cycles
- **Which block has smallest AMAT for each cache size?**

块大小、容量对AMAT的影响

- **Solution: assume hit time = 1 clock cycle**
  - Regardless of block size and cache size
- **Cache Size = 4 KB, Block Size = 16 bytes**
  - AMAT = 1 + 8.57% × 82 = 8.027 clock cycles
- **Cache Size = 256 KB, Block Size = 256 bytes**
  - AMAT = 1 + 0.49% × 112 = 1.549 clock cycles

Block Size	Cache = 4 KB	Cache = 16 KB	Cache = 64 KB	Cache = 256 KB
16 bytes	AMAT = 8.027	AMAT = 4.231	AMAT = 2.673	AMAT = 1.894
32 bytes	AMAT = **7.082**	AMAT = 3.411	AMAT = 2.134	AMAT = 1.588
64 bytes	AMAT = 7.160	AMAT = **3.323**	AMAT = **1.933**	AMAT = **1.449**
128 bytes	AMAT = 8.469	AMAT = 3.659	AMAT = 1.979	AMAT = 1.470
256 bytes	AMAT = 11.65	AMAT = 4.685	AMAT = 2.288	AMAT = 1.549

块大小、容量对AMAT的影响



Block Size	Penalty	Cache Size 1K	Cache Size 4K	Cache Size 16K	Cache Size 64K	Cache Size 256K
16	42	15.05% / 7.321	8.57% / 4.599	3.94% / 2.655	2.04% / 1.857	1.09% / 1.458
32	44	13.34% / 6.870	7.24% / 4.186	2.87% / 2.263	1.35% / 1.594	0.70% / 1.308
64	48	13.76% / 7.605	7.00% / 4.360	2.64% / 2.267	1.06% / 1.509	0.51% / 1.245
128	56	16.64%/ 10.318	7.78% / 5.357	2.77% / 2.551	1.02% / 1.571	0.49% / 1.274
256	72	22.01% /16.847	9.51% / 7.847	3.29% / 3.369	1.15% / 1.828	0.49% / 1.353

Miss Rate / Average Access Time (in cycles)

what's going on here?

- Remember
  - Avg-access-time= hit-time + miss-rate x miss-penalty

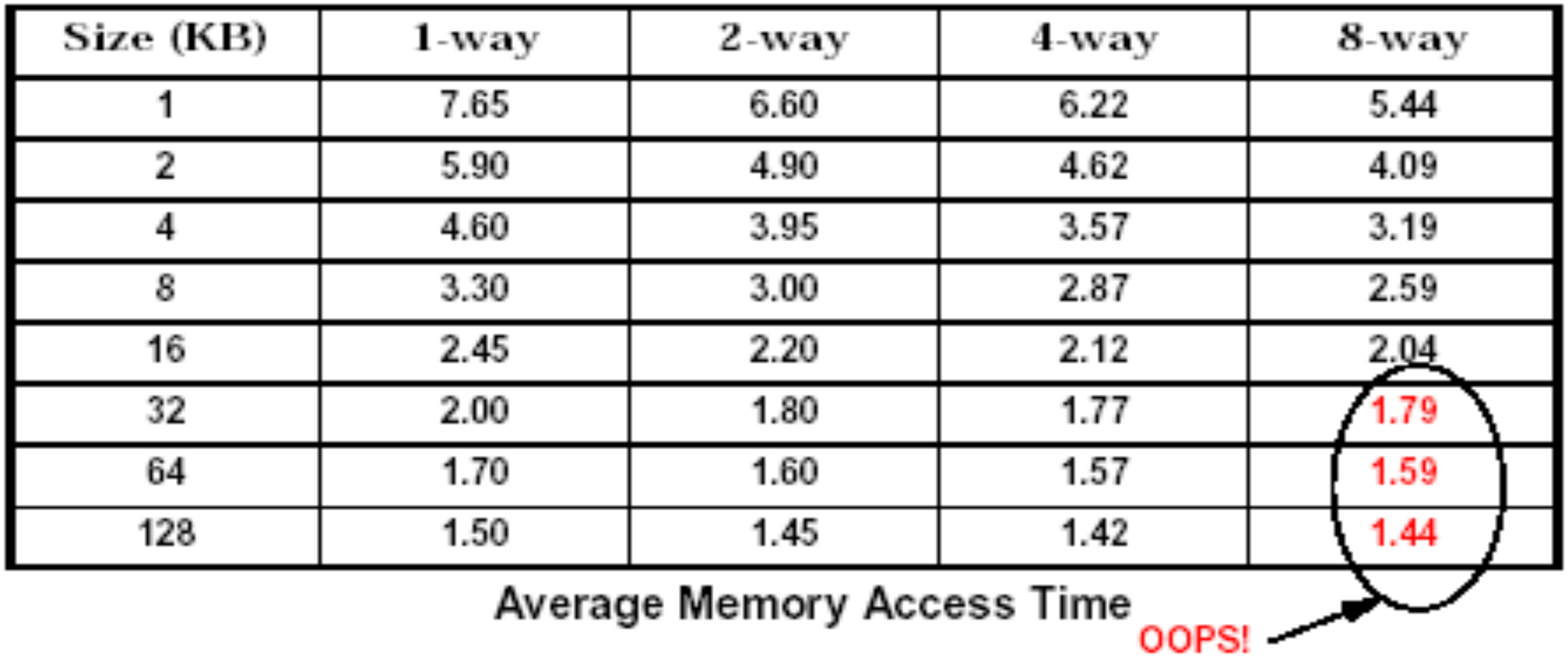
- 降低失效率最简单的方法是增加块大小；统计结果如图所示
- 假定存储系统在延迟40个时钟周期后，每2个时钟周期能送出16个字节，即：经过42个时钟周期，它可提供16个字节；经过44个四周周期，可提供32个字节；依此类推。试根据图5-6列出的各种容量的Cache，在块大小分别为多少时，平均访存时间最小?

块大小、容量的权衡

- **从统计数据可得到如下结论**
  – 对于给定Cache容量，块大小增加时，失效率开始是下降，但后来反而上升
  – Cache容量越大，使失效率达到最低的块大小就越大
- **分析**
  – 块大小增加，可使强制性失效减少（空间局部性原理）
  – 块大小增加，可使冲突失效增加（Cache中块数量减少）
  – 失效开销增大（上下层间移动，数据传输时间变大）
- **设计块大小的原则，不能仅看失效率**
  – 原因：平均访存时间 ＝ 命中时间＋失效率×失效开销

提高相联度

- 8路组相联在降低失效率方面的作用已经和全相联一样有效
- **2:1Cache经验规则**
  - 容量为N的直接映象Cache失效率与容量为N/2的两路组相联Cache的失效率差不多相同
- 提高相联度，会增加命中时间



Size (KB)	1-way	2-way	4-way	8-way
1	7.65	6.60	6.22	5.44
2	5.90	4.90	4.62	4.09
4	4.60	3.95	3.57	3.19
8	3.30	3.00	2.87	2.59
16	2.45	2.20	2.12	2.04
32	2.00	1.80	1.77	1.79
64	1.70	1.60	1.57	1.59
128	1.50	1.45	1.42	1.44

Average Memory Access Time

Victim Cache(1/2)

- 在Cache和Memory之间增加一个小的全相联Cache

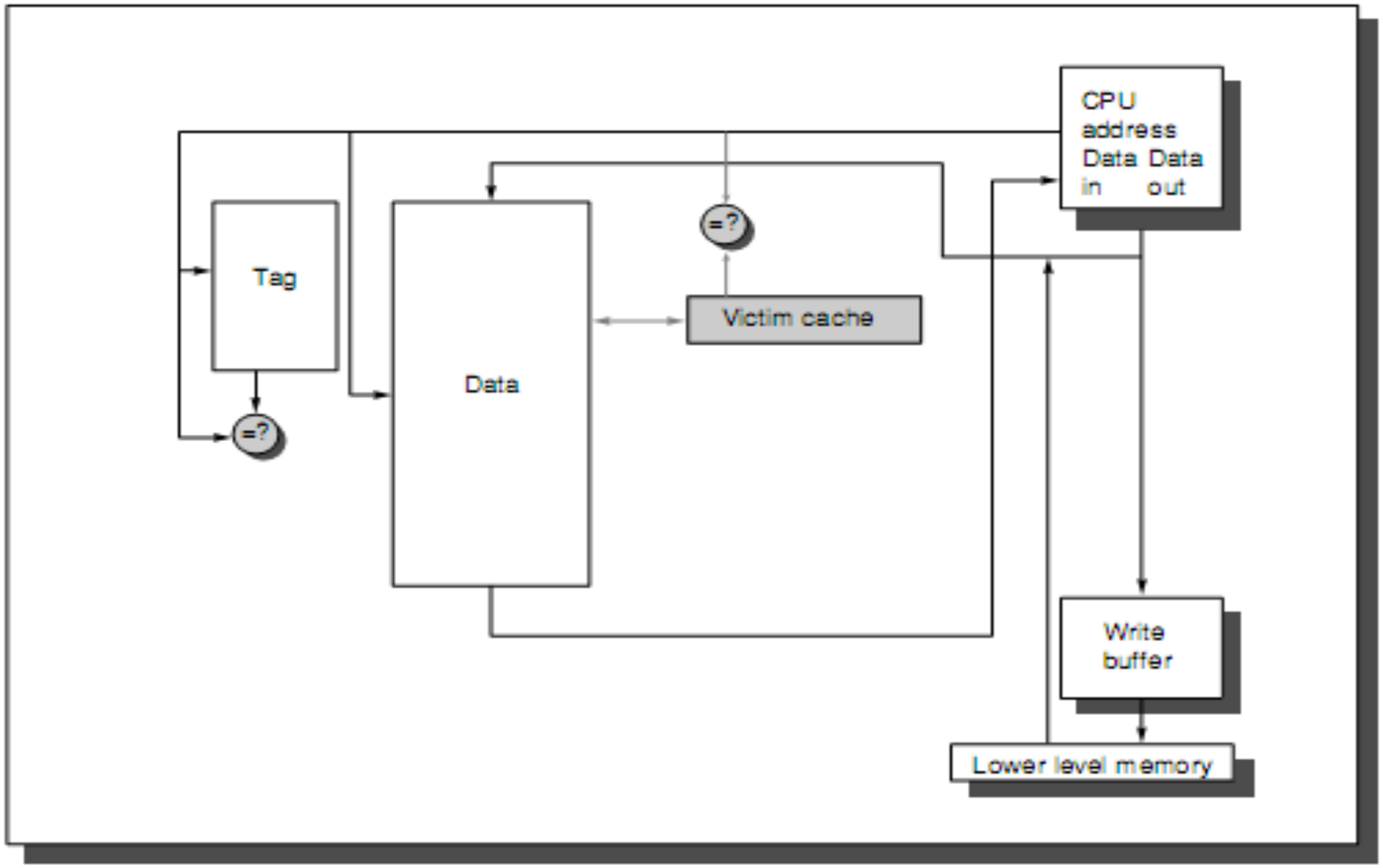


FIGURE 5.13 Placement of victim cache in the memory hierarchy. Although it reduces miss penalty, the victim cache is aimed at reducing the damage done by conflict misses, described in the next section. Jouppi [1990] found the four-entry victim cache could reduce the miss penalty for 20% to 95% of conflict misses.

Victim Cache(2/2)

- **基本思想**
  - 通常Cache为直接映象时冲突失效率较大
  - Victim cache采用全相联－失效率较低
  - Victim cache存放由于（冲突）失效而被丢弃的那些块
  - 失效时，首先检查Victim cache是否有该块，如果有就将该块与Cache中相应块比较。
- **Jouppi (DEC SRC)发现，含1到5项的Victim cache对减少失效很有效，尤其是对于那些小型的直接映象数据Cache 。测试结果，项为4的Victim Cache能使4KB直接映象数据Cache冲突失效减少20%-90%**

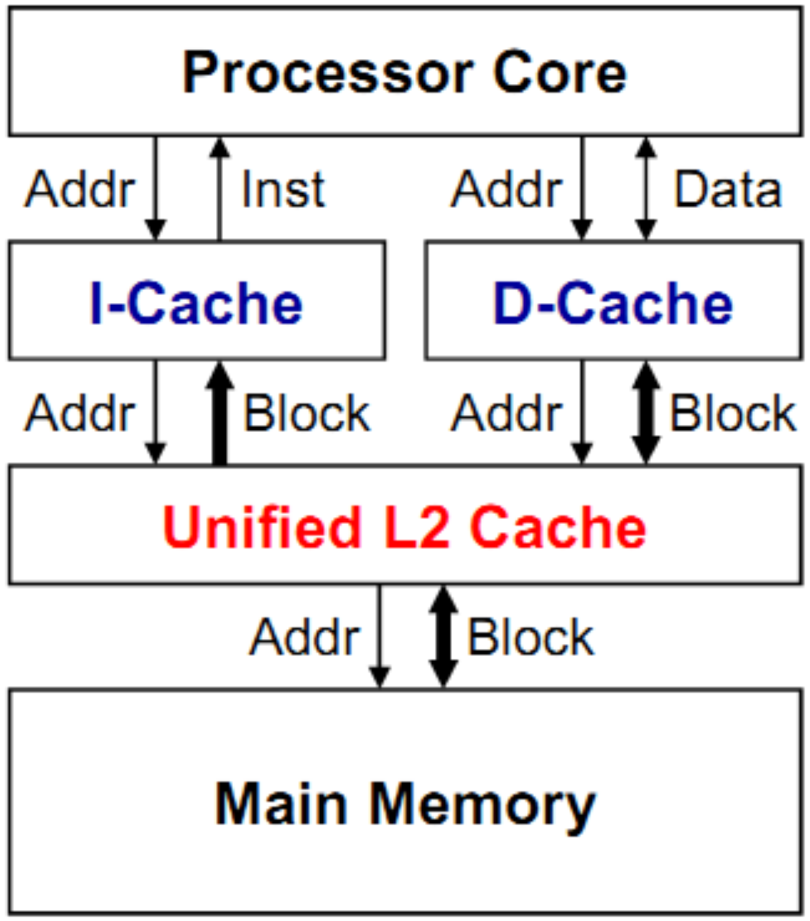
减少失效开销

- **减少CPU与存储器间性能差异的重要手段**
  – 平均访存时间＝命中时间＋失效率×失效开销

- **基本手段：**
  – 4、多级Cache技术(Multilevel Caches)
  – 5、让读优先于写(Giving Priority to Read Misses over Writes)

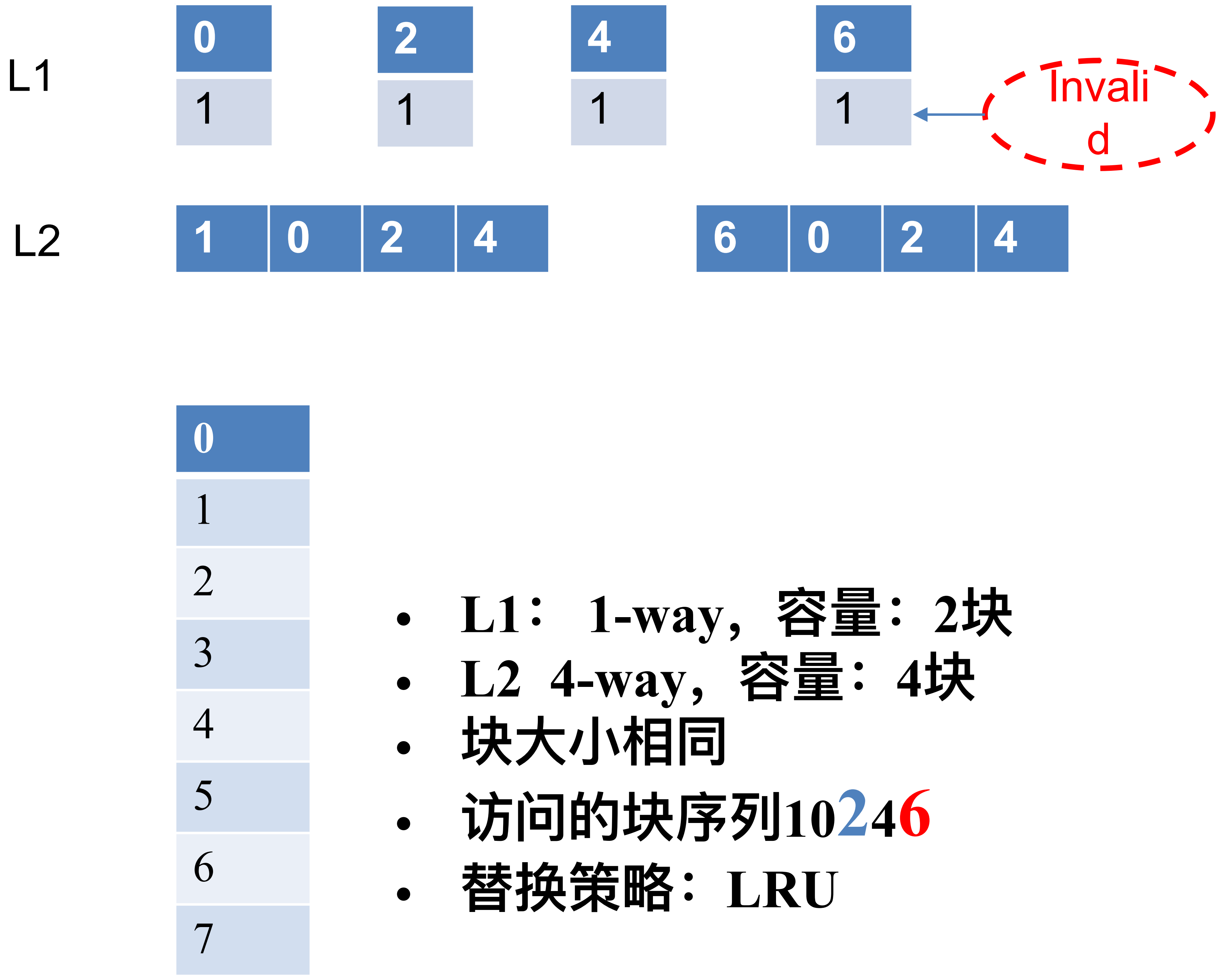
采用多级Cache

- **一级cache保持较小容量**
  - 降低命中时间
  - 降低每次访问的能耗
- **增加二级cache**
  - 减少与存储器的gap
  - 减少存储器总线的负载
- **多级cache的优点**
  - 减少失效开销
  - 缩短平均访存时间（AMAT,
- **较大容量的L2 cache可以捕捉许多L1 cache的失效**
  - 降低全局失效率

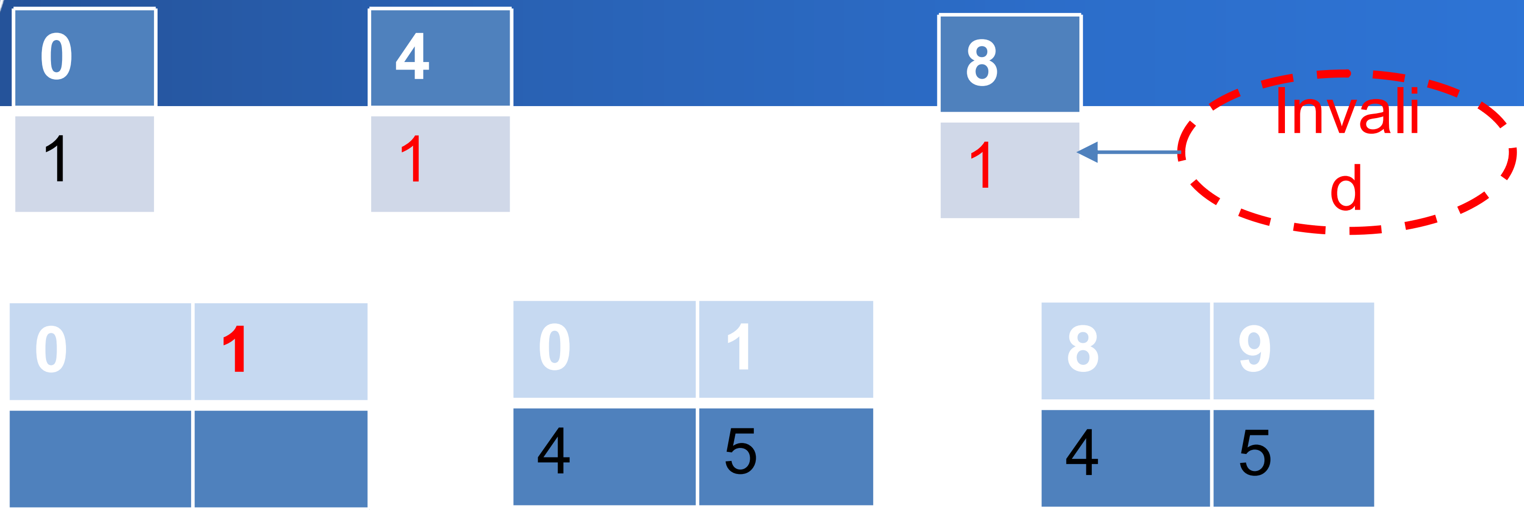


多级包容性(multilevel inclusive)

- **L1 cache 的块总是存在于L2 cache中**
  - 浪费了L2 cache 空间，L2 还应当有存放其他块的空间
- **L1中miss, 但在L2中命中，则从L2拷贝相应的块到L1**
- **在L1和L2中均miss, 则从更低级拷贝相应的块到L1和L2**
- **对L1写操作导致将数据同时写到L1和L2**
- **Write-through 策略用于L1到L2**
- **Write-back 策略可用于L2 到更低级存储器，以降低存储总线的数据传输压力**
- **L2的替换动作（或无效)对L1可见**
  - 即L2的一块被替换出去，那么其在L1中对应的块也要被替换出去。



- **L1： 1-way，容量：2块**
- **L2 4-way，容量：4块**
- **块大小相同**
- **访问的块序列10246**
- **替换策略：LRU**



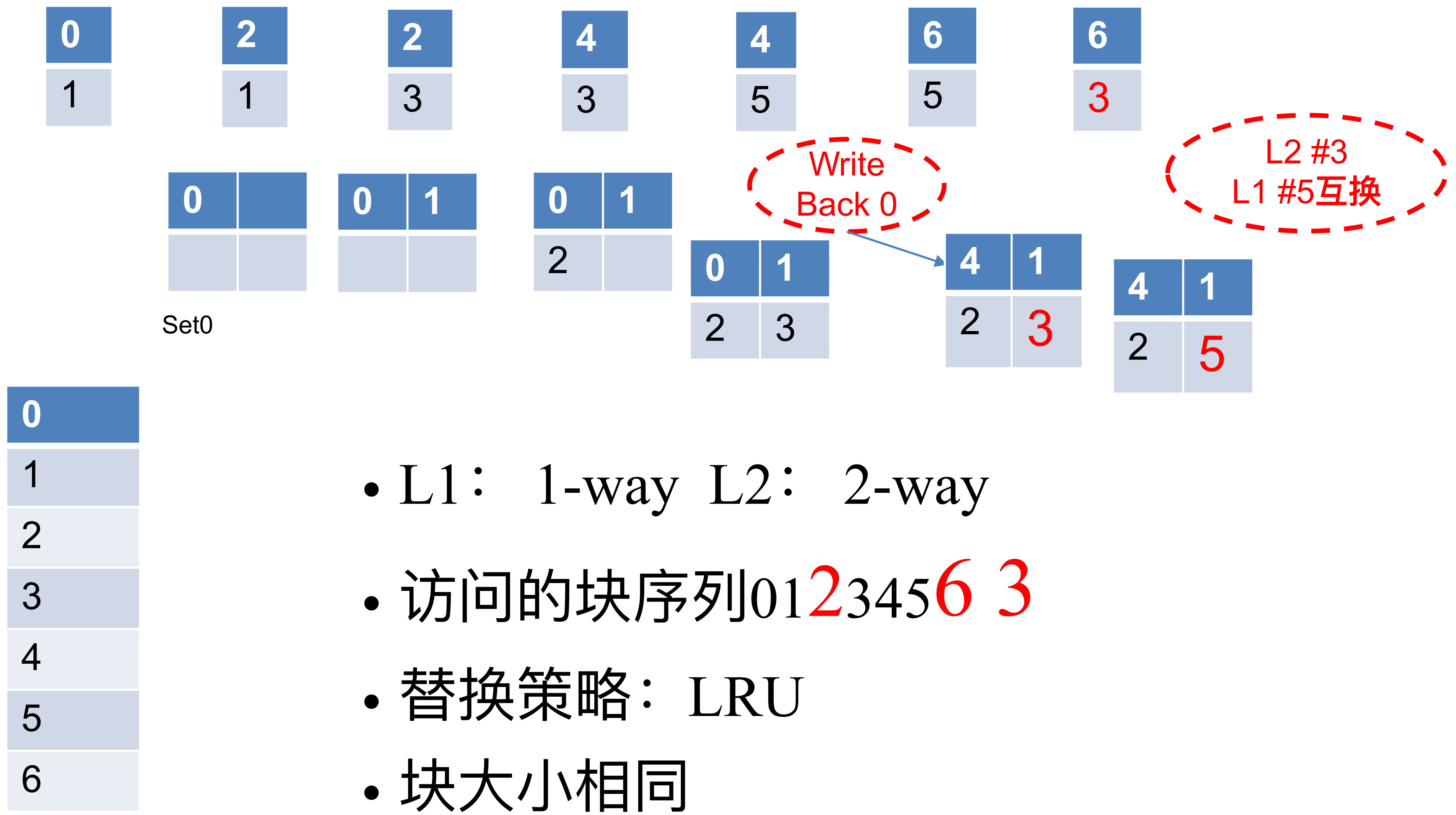
0	0(0)
1	0(1)
2	1(2)
3	1(3)
4	2(4)
5	2(5)
6	3(6)
7	3(7)
8	4(8)
9	4(9)

- L1： 1-way，容量：2块
- L2 2-way，容量：2块
- L1和L2的块大小不同，L1的blockSize是L2的blockSize的1/2
- 替换策略：LRU
- 考察按L1的块划分的访问序列：

0148

多级不包含（Multilevel Exclusive）

- L1 cache 中的块不会在L2 cache中，以避免浪费空间
- 在L1中miss, 但在L2中命中，将导致Cache间块的互换
- 在L1和L2中均miss, 将仅仅从更低层拷贝相应的块到L1
- L1的被替换的块移至L2
  - L2 存储L1抛弃的块，以防后续L1还需要使用
- L1到L2的写策略为 Write-Back
- L2到更低级cache的写策略为 Write-Back
- L1和L2的块大小可以相同也可以不同
  - Pentium 4 had 64-byte blocks in L1 but 128-byte blocks in L2
  - Core i7 uses 64-byte blocks at all cache levels (simpler)



- L1： 1-way L2： 2-way
- 访问的块序列0123456 3
- 替换策略：LRU
- 块大小相同

多级cache的性能分析

- **局部失效率：该级Cache的失效次数 / 到达该级Cache的访存次数**
  - Miss rateL1 for L1 cache
  - Miss rateL2 for L2 cache
- **全局失效率：该级Cache的失效次数/ CPU发出的访存总次数**
  - Miss rateL1 for L1 cache
  - Miss rateL1 × Miss rateL2 for L2 cache
  - 全局失效率是度量L2 cache性能的更好方法
- **性能参数**
  - AMAT = Hit TimeL1+Miss rateL1×Miss penaltyL1
  - Miss penaltyL1 = HitTimeL2 + Miss rateL2×Miss penaltyL2
  - AMAT = Hit TimeL1+Miss rateL1× (Hit TimeL2+Miss rateL2×Miss penaltyL2)

L1 cache 失效率

- **对于I-Cache和D-Cache分开的L1 Cache**

  Miss RateL1 = %inst × Miss RateI-Cache + %data × Miss RateD-Cache

  %inst = Percent of Instruction Accesses = 1 / (1 + %LS)

  %data = Percent of Data Accesses = %LS / (1 + %LS)

  %LS = Frequency of Load and Store instructions

- **每条指令的L1 失效次数:**

  Misses per InstructionL1 = Miss RateL1 × (1 + %LS)

  Misses per InstructionL1 = Miss RateI-Cache + %LS × Miss RateD-Cache

具有二级Cache的AMAT举例

- **Problem: 计算AMAT**
  - I-Cache 失效率 = 1%， D-Cache失效率 = 10%
  - L2 Cache失效率 = 40%
  - L1 命中时间 = 1 cycle (I-Cache 和D-Cache相同）
  - L2 命中时间 = 8 cycles， L2 失效开销 = 100 cycles
  - Load + Store 指令频度 = 25%
- **Solution：**
  - 平均每条指令访存次数 = 1+25% = 1.25
  - 平均每条指令的失效次数 = 1%+25%× 10% = 0.035
  - L1的失效率= 0.035/1.25=0.028
  - L1的失效开销 = 8 + 0.4×100 = 48 cycles
  - AMAT = 1+0.028×48 = 2.344

Memory Stall Cycles Per Instruction

- **Memory Stall Cycles per Instruction**

  = Memory Access per Instruction × Miss RateL1 × Miss PenaltyL1

  = (1 + %LS) × Miss RateL1 × Miss PenaltyL1

  = (1 + %LS) × Miss RateL1 × (Hit TimeL2 + Miss RateL2 × Miss PenaltyL2)

- **Memory Stall Cycles per Instruction**

  = Misses per InstructionL1 × Hit TimeL2 +
  Misses per InstructionL2 × Miss PenaltyL2

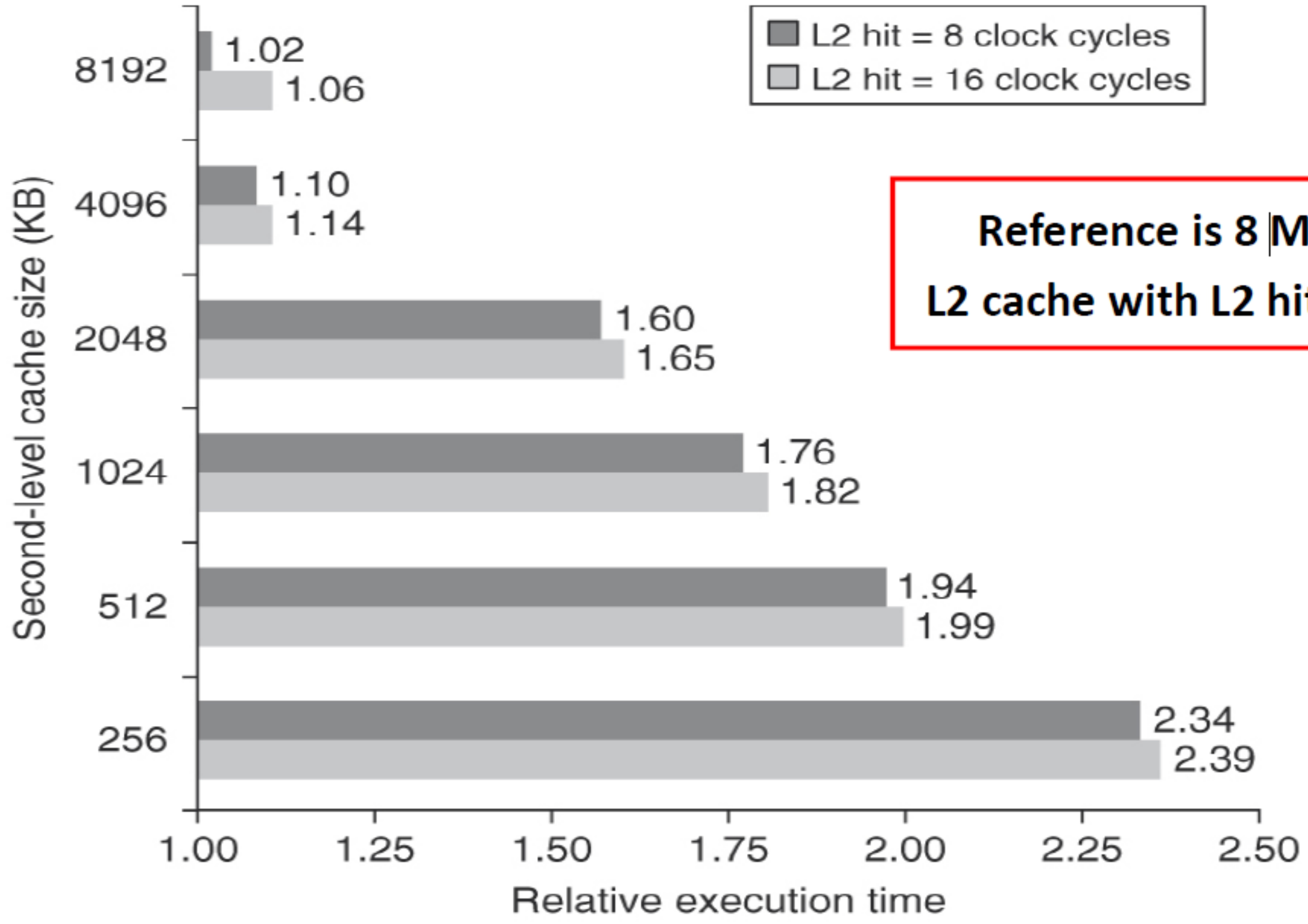
  Misses per InstructionL1 = (1 + %LS) × Miss RateL1

  Misses per InstructionL2 = (1 + %LS) × Miss RateL1 × Miss RateL2

两级Cache的性能（课后看）

- **Problem: 程序运行产生1000个存储器访问**
  - I-Cache misses = 5, D-Cache misses = 35, L2 Cache misses = 8
  - L1 Hit = 1 cycle, L2 Hit = 8 cycles, L2 Miss penalty = 80 cycles
  - Load + Store frequency = 25%, CPIexecution = 1.1 (perfect cache)
  - 计算memory stall cycles per instruction 和有效的CPI
  - 如果没有L2 cache, 有效的CPI是多少?
- **Solution：**
  - L1 Miss Rate = (5 + 35) / 1000 = 0.04 (or 4% per access)
  - L1 misses per Instruction =0.04 × (1 + 0.25) = 0.05
  - L2 misses per Instruction =(8 / 1000) × 1.25 = 0.01
  - Memory stall cycles per Instruction =0.05 × 8 + 0.01 × 80 = 1.2
  - CPIL1+L2 = 1.1 + 1.2 = 2.3, CPI/CPIexecution = 2.3/1.1 = 2.1x slower
  - CPIL1only =1.1 + 0.05 × 80 = 5.1 (worse)

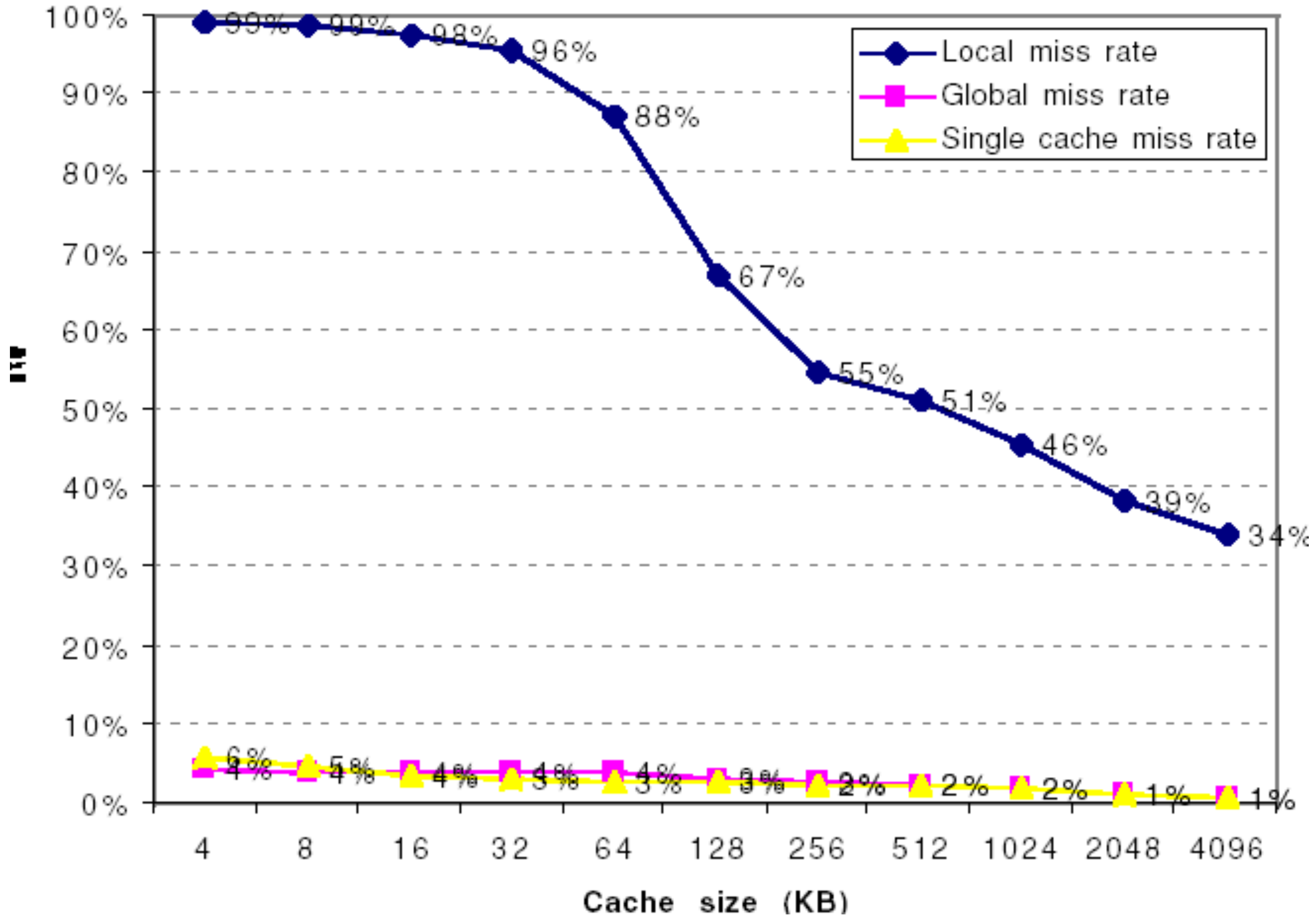
不同大小的L2cache对应的执行时间



Reference is 8 MB

L2 cache with L2 hit = 1

Miss rates versus cache size for multilevel caches



两级Cache的一些研究结论

- **在L2比L1大得多得情况下，两级Cache全局失效率 和容量与第二级Cache相同的单级Cache的失效率接近**
- **局部失效率不是衡量第二级Cache的好指标**
  - 它是第一级Cache失效率的函数
  - 不能全面反映两级Cache体系的性能
- **第二级Cache设计需考虑的问题**
  - 容量：一般很大，可能没有容量失效，只有强制性失效和冲突失效
    - 相联度对第二级Cache的作用
    - Cache可以较大，以减少失效次数
  - 多级包容性问题：第一级Cache中的数据是否总是同时存在于第二级Cache中。
    - 如果L1和L2的块大小不同，增加了多级包容性实现的复杂性

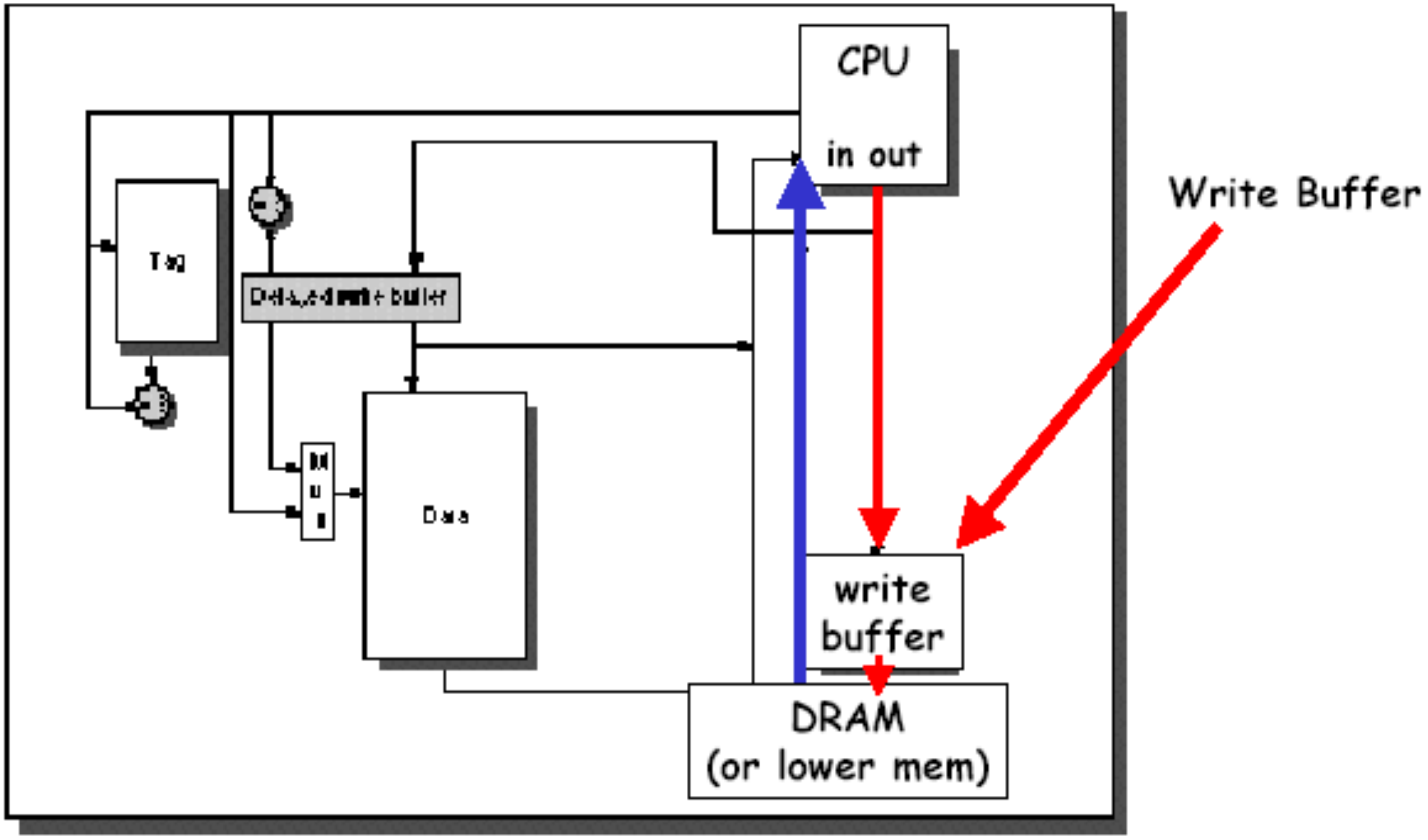
多级Cache举例（课后看）

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

- Hit timeL2 for direct mapped = 10 clock cycles
- Two-way set associativity increases hit time by 0.1 clock cycles to 10.1clock cycles
- Local miss rateL2 for direct mapped = 25%
- Local miss rateL2 for two-way set associative = 20%
- Miss penaltyL2 = 100 clock cycles

- **结论：提高相联度，可减少第一级Cache的失效开销**
- **第二级Cache特点：容量大，高相联度，块较大，重点减少失效次数。**

让读优先于写图示


CPU
in out
Tag
Delayed write buffer
MUX
Data
write buffer
DRAM
(or lower mem)
Write Buffer

让读失效优先于写

- **由于读操作为大概率事件，需要读失效优先，以提高性能**
- **Write-Through Cache ->Write Buffer (写缓冲)，特别对写直达法更有效**
  - Write Buffer: CPU不必等待写操作完成，即将要写的数据和地址送到Write Buffer后，CPU继续作其他操作。
  - 写缓冲导致对存储器访问的复杂化
    - 在读失效时写缓冲中可能保存有所读单元的最新值，还没有写回
    - 例如，直接映射、写直达、512和1024映射到同一块。则
    - SW R3, 512(R0)
    - LW R1, 1024(R0)　　失效
    - LW R2, 512(R0)　　失效
  - 解决问题的方法
    - 推迟对读失效的处理，直到写缓冲器清空，导致新的问题——读失效开销增大。
    - 在读失效时，检查写缓冲的内容，如果没有冲突，而且存储器可访问，就可以继续处理读失效
  - 写回法时，也可以利用写缓冲器来提高性能
    - 把脏块放入缓冲区，然后读存储器，最后写存储器
- **Write-Back Cache ->Victim Buffer**
  - 被替换的脏块放到了victim buffer
  - 在脏块被写回前，需要处理读失效
  - 问题: victim buffer 可能含有该读失效要读取的块
    - Solution: 查找victim buffer，如果命中直接将该块调入Cache

缩短命中时间 Virtually indexed cache

- **Cache index用virtual地址还是physical地址?**
- **Virtual index 可以避免在cache indexing时的virtual to physical address translation, 所以命中时间短。**
- **How about tags? A pure virtual cache (with virtual index + virtual tag) seems faster. But what is its problem?**
- **With multiple processes and process switching, a pure virtual cache is not practical**
  - May need to purge the entire cache on a process switch
  - Or can attach a PID field
- **Usually Virtually index, physically tagged**

虚拟地址Cache VS. 物理地址Cache

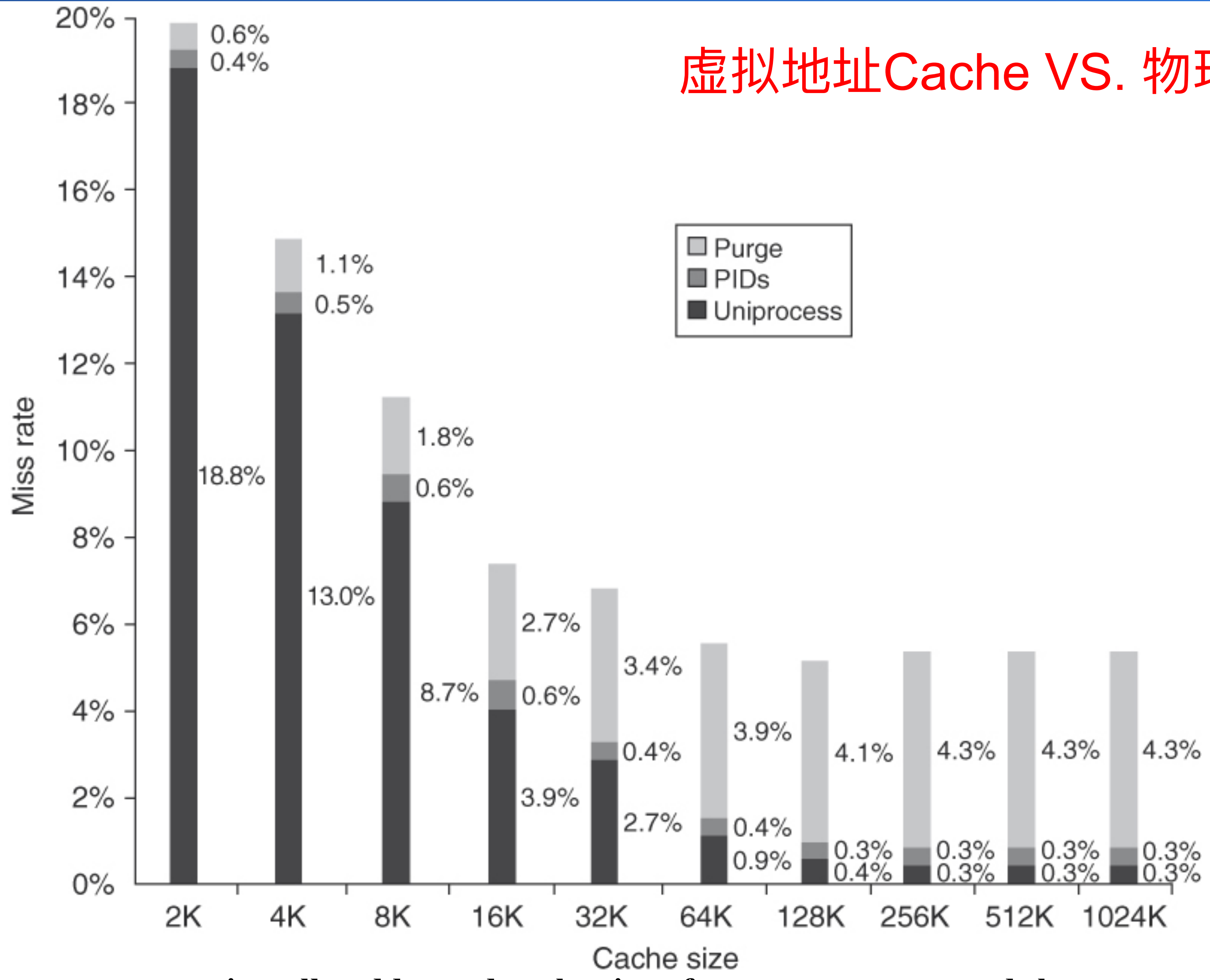
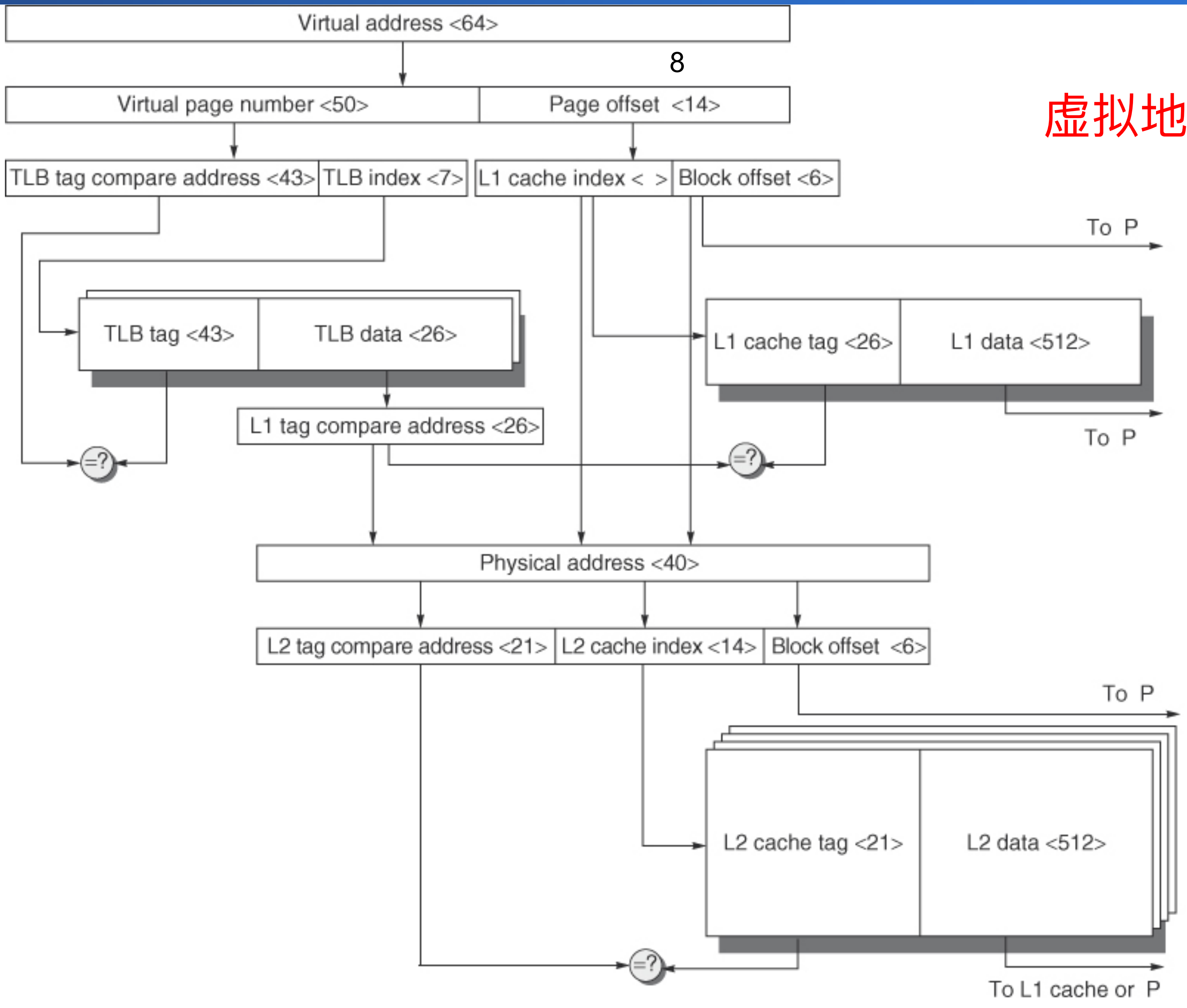


Figure B.16 Miss rate versus virtually addressed cache size of a program measured three ways: without process switches (uniprocess), with process switches using a process-identifier tag (PID), and with process switches but without PIDs (purge). PIDs increase the uniprocess absolute miss rate by 0.3% to 0.6% and save 0.6% to 4.3% over purging. Agarwal [1987] collected these statistics for the Ultrix operating system running on a VAX, assuming direct-mapped caches with a block size of 16 bytes. Note that the miss rate goes up from 128K to 256K. Such nonintuitive behavior can occur in caches because changing size changes the mapping of memory blocks onto cache blocks, which can change the conflict miss rate.



虚拟地址转换与Cache定位 并行

Virtually indexed, Physically tagged L1

Physically indexed, Physically tagged L2

Figure B.17 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 16 KB. The TLB is two-way set associative with 256 entries. The L1 cache is a direct-mapped 16 KB, and the L2 cache is a four-way set associative with a total of 4 MB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 40 bits.

高级Cache优化方法

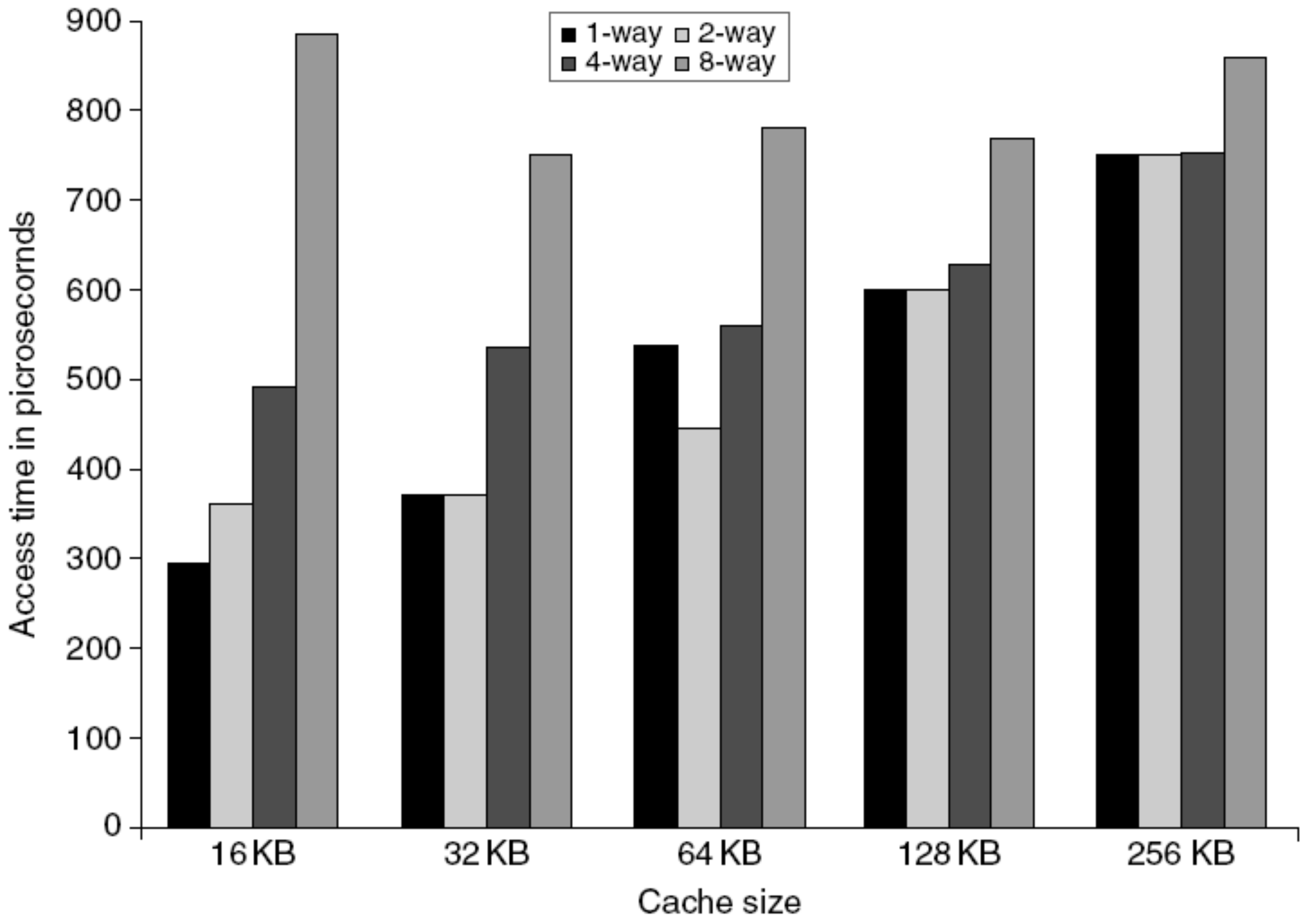
- 缩短命中时间
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- **增加Cache带宽**
  - 3、Cache访问流水化
  - 4、无阻塞Cache
  - 5、多体Cache
- **减小失效开销**
  - 6、关键字优先和提前重启
  - 7、合并写
- **降低失效率**
  - 8、编译优化
- **通过并行降低失效开销或失效率**
  - 9、硬件预取
  - 10、编译器控制的预取

1、Small and simple first level caches

- **Small and simple first level caches**
  - 容量小，一般命中时间短，有可能做在片内
  - 另一方案，保持Tag在片内，块数据在片外，如DEC Alpha
  - 第一级Cache应选择容量小且结构简单的设计方案
- **Critical timing path:**

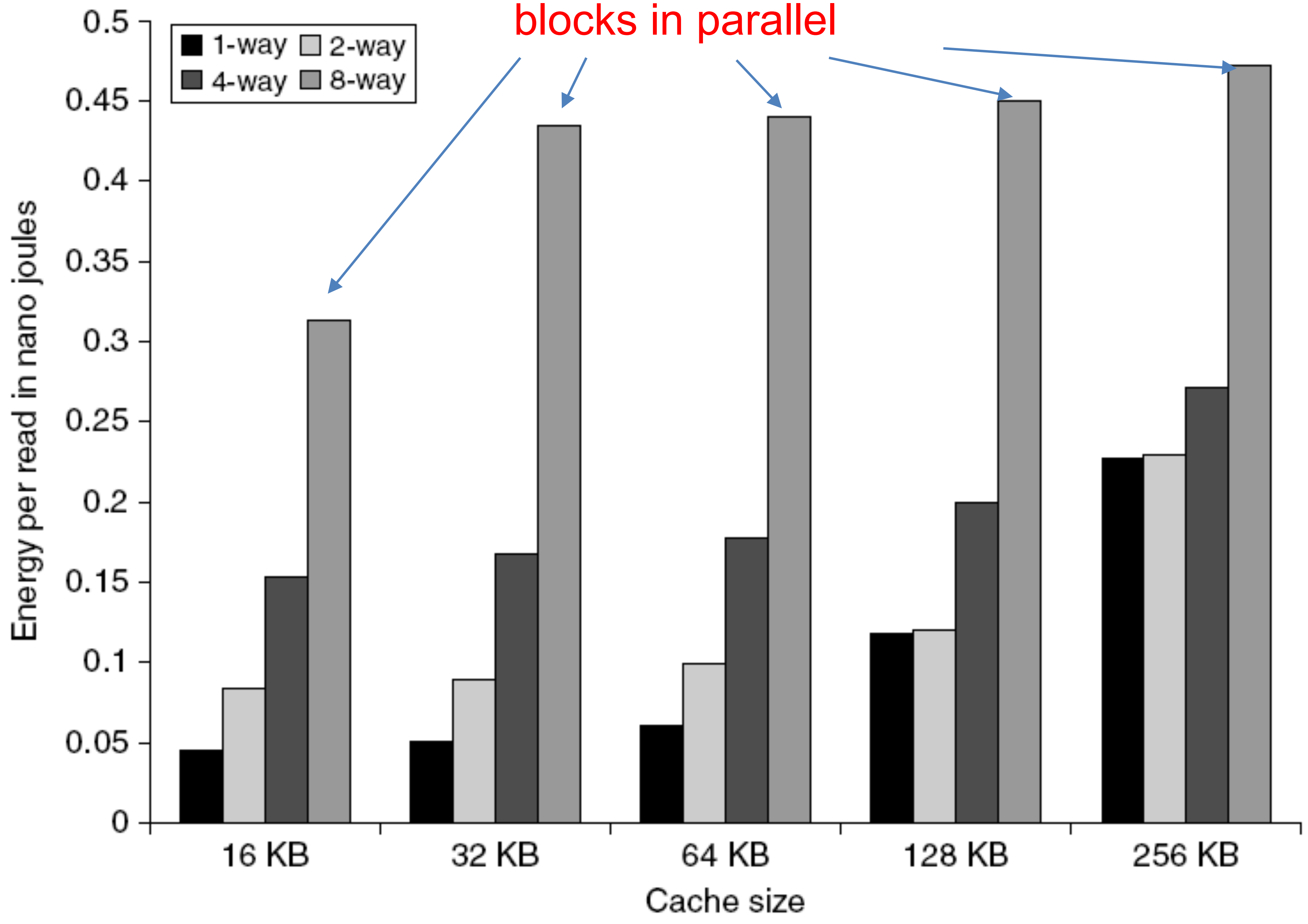
  1) 定位组, 确定tag的位置
  2) 比较tags,
  3) 选择正确的块
- **Direct-mapped caches can overlap tag compare and transmission of data**
  - 数据传输和tag 比较并行
- **Lower associativity reduces power because fewer cache lines are accessed**
  - 简单的Cache结构、可有效减少tag比较的次数，进而降低功耗

L1 Size and Associativity



L1 Size and Associativity

Reading out 8 tags and data blocks in parallel



:y

2、Way Prediction

- **为改进命中时间，预测被选中的路(way)**
  - 预测错误会导致更长的命中时间
  - 预测的准确性
    - 90%+ for two-way
    - 80%+ for four-way
    - I-cache比D-cache具有更好的准确性
  - 90年代中期第一次用于MIPS R10000
  - 用于ARM Cortex-A8
- **Way Prediction方法也可降低功耗：直接预测要访问的块**
  - 也称“路选择”“Way selection”
  - 可有效降低功耗，但一旦预测错误会有更长的命中时间

高级Cache优化方法

- **缩短命中时间**
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- **增加Cache带宽**
  - 3、Cache访问流水化
  - 4、无阻塞Cache
  - 5、多体Cache
- **减小失效开销**
  - 6、关键字优先和提前重启
  - 7、合并写
- **降低失效率**
  - 8、编译优化
- **通过并行降低失效开销或失效率**
  - 9、硬件预取
  - 10、编译器控制的预取

3、Pipelining Cache

- **实现Cache访问的流水化**
  - 提高Cache的带宽，有利于采用高相联度的缓存
  - L1 cache的访问由多个时钟周期构成
    - Pentium: 1 cycle
    - Pentium Pro – Pentium III: 2 cycles
    - Pentium 4 – Core i7: 4 cycles
    - IBM Power7：3 cycles
- **缺点：增加流水线的段数**
  - 增加了分支预测错误造成的额外开销 （I-cache）
  - 增加了Load指令与要使用其结果的指令间的latency
  - 增加了I-Cache和D-Cache的延时

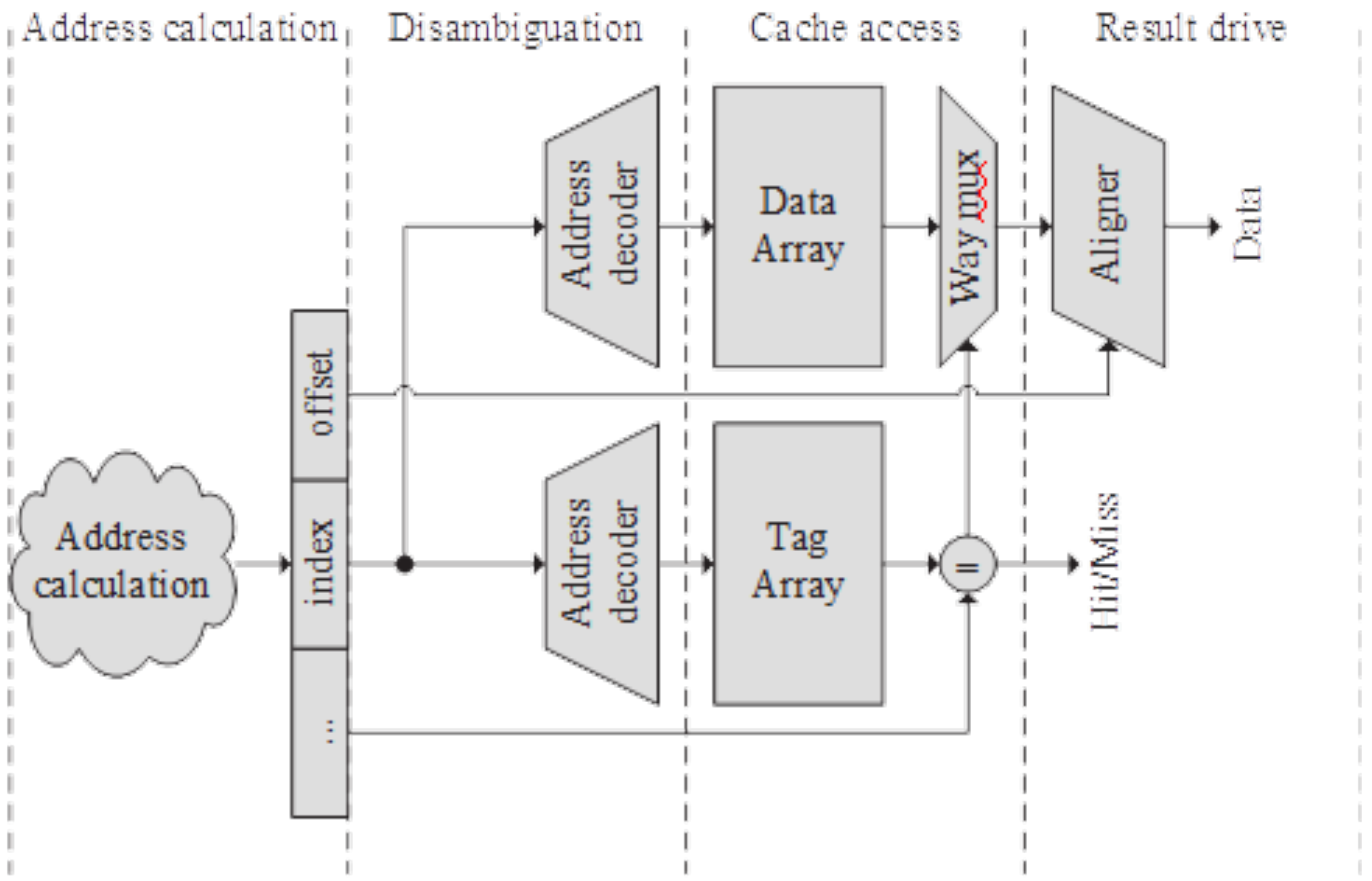


FIGURE 2.2: Parallel tag and data array access pipeline.

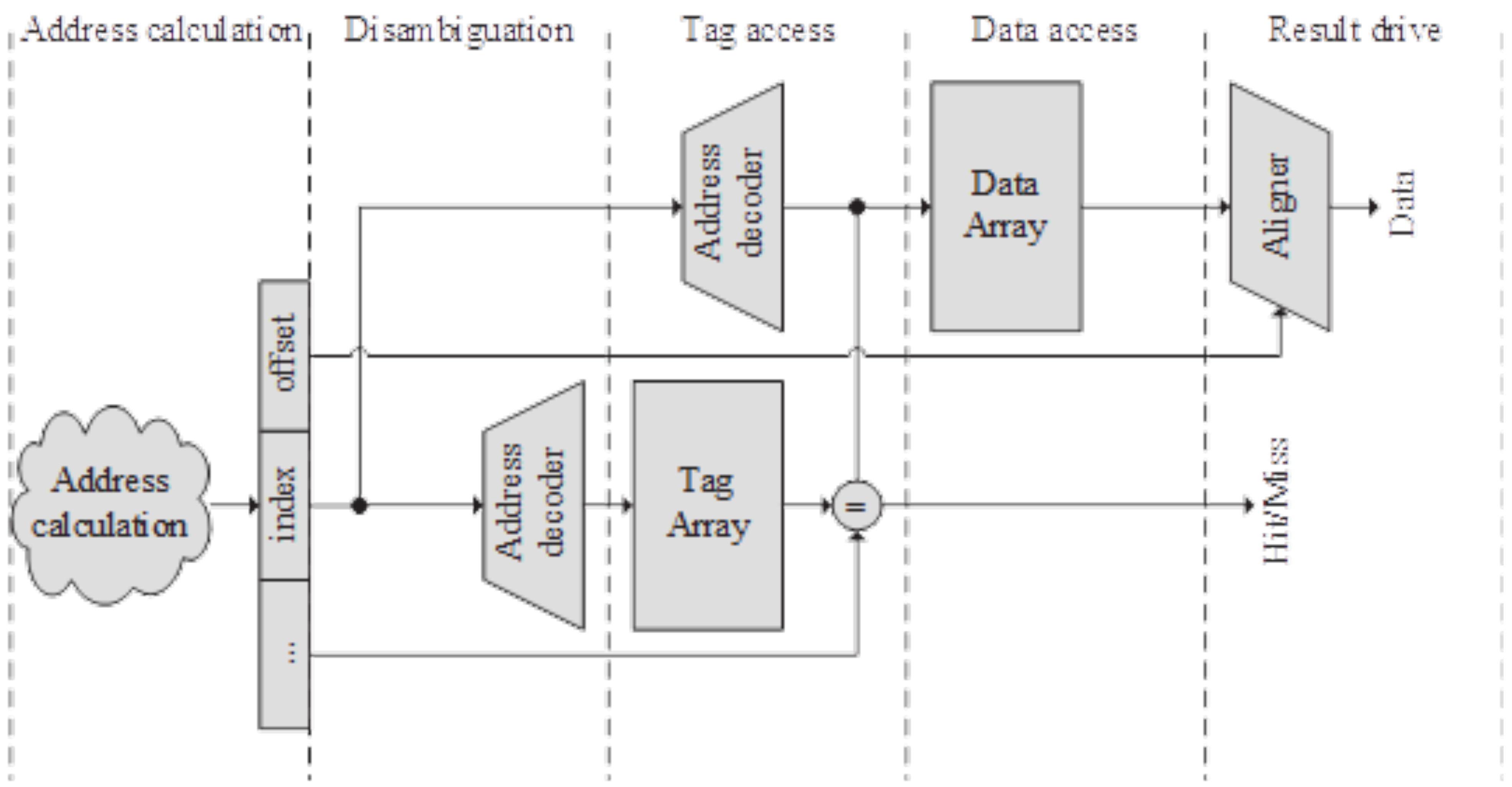
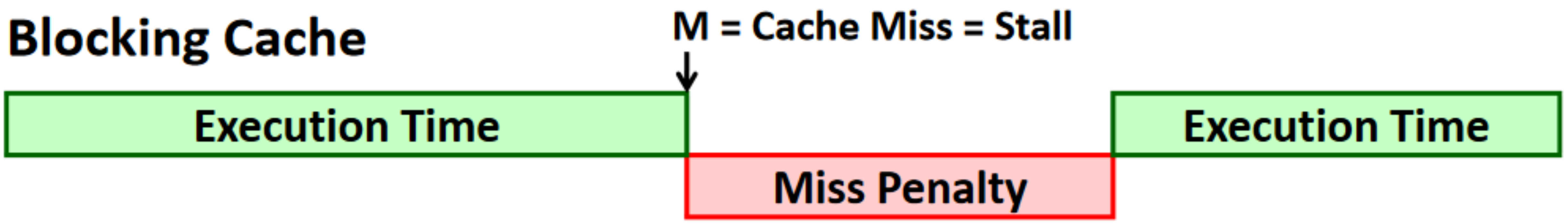


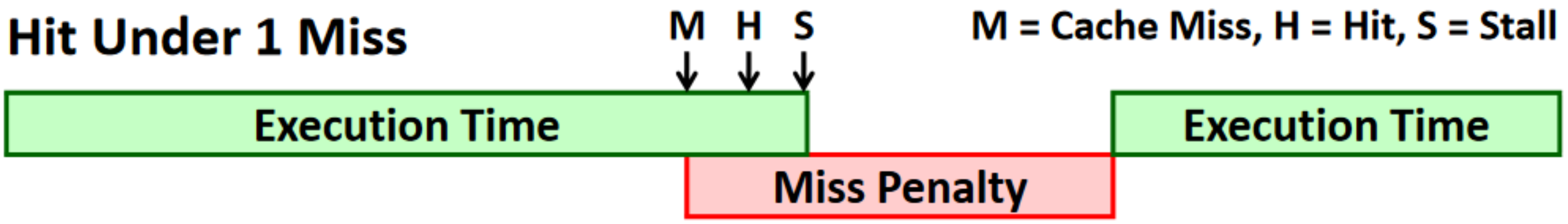
FIGURE 2.4: Serial tag and data array access pipeline.

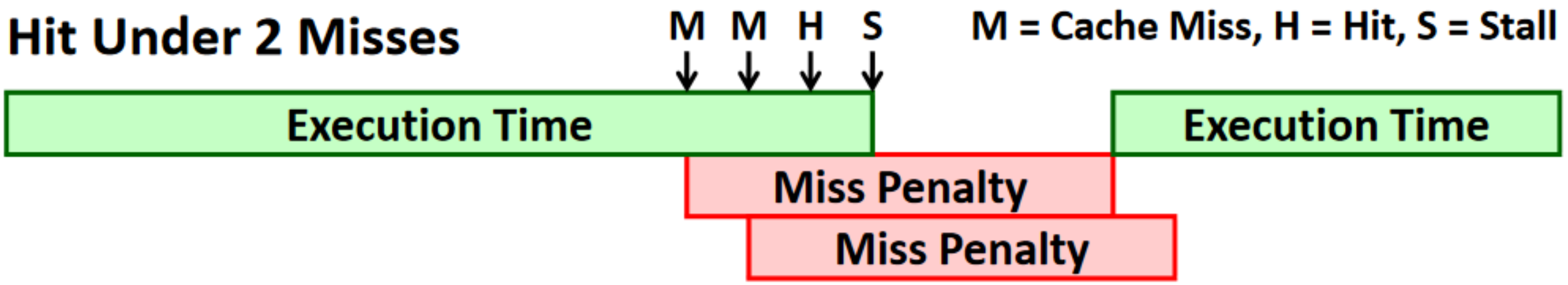
4、Nonblocking Caches

- **允许在Cache失效下继续命中**
  - 在Cache失效时，CPU无需stall
  - 主要用于乱序执行和多线程处理器
- **Hit under a Miss**
  - 减少有效的失效开销
  - 增加Cache的带宽
- **Hit under Multiple Misses**
  - 针对多个未解决的Cache失效
  - 可能会更多地减少有效的失效开销
  - 增加了Cache控制器的复杂性
  - 存储系统可以支持多个失效时的存储服务

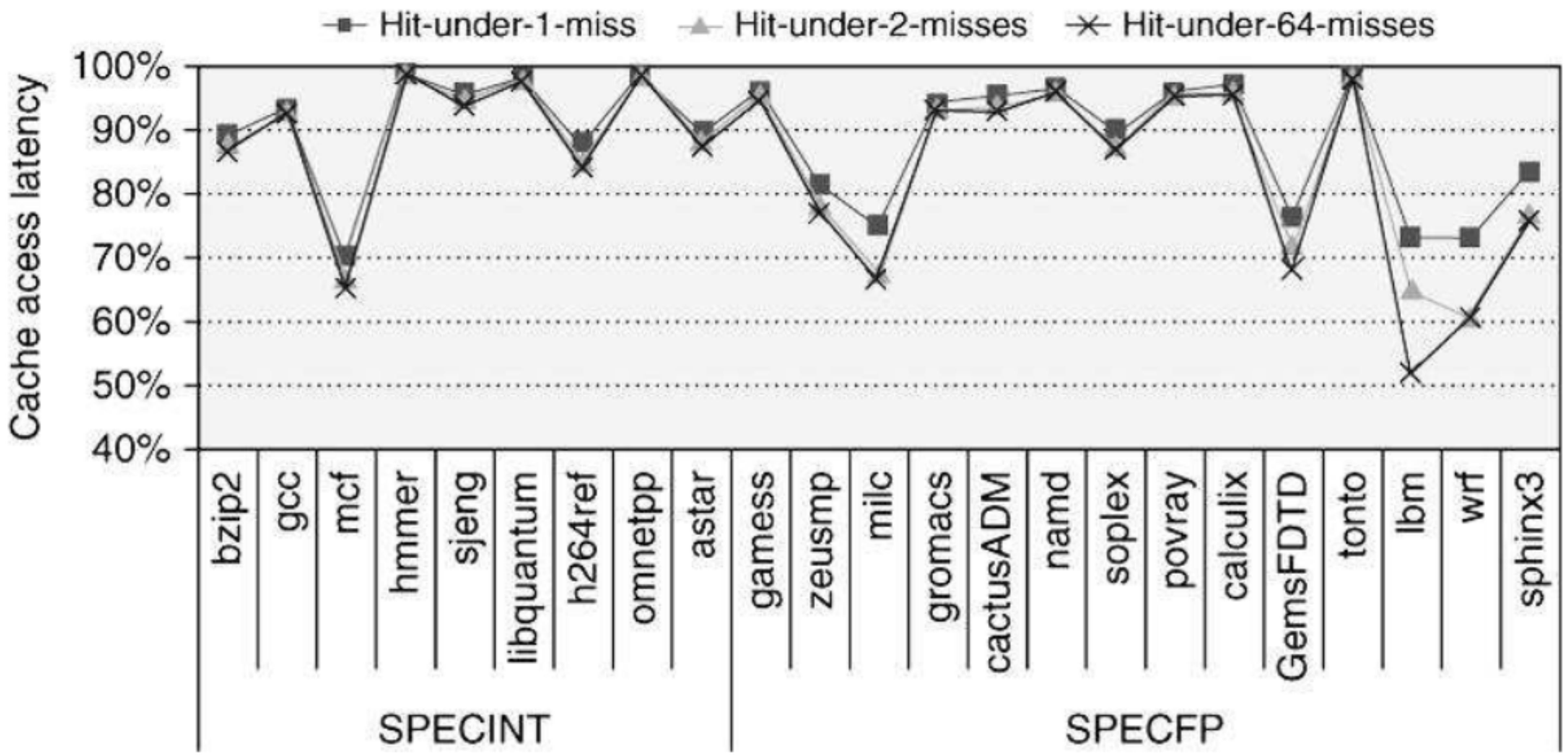
Nonblocking Cache Timeline







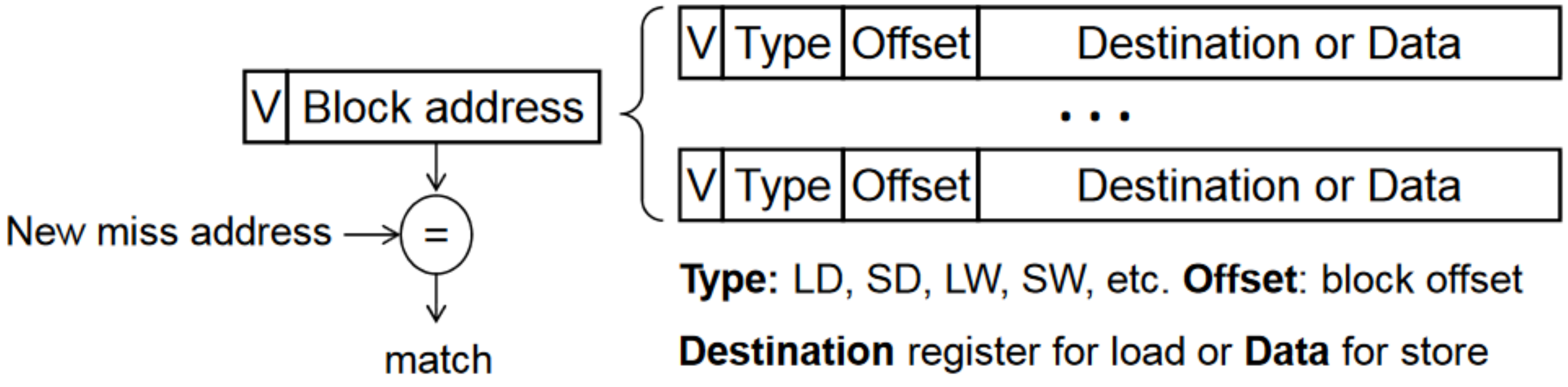
Effectiveness of Non-Blocking Cache



Reduces the miss penalty by not stalling the processor on a cache miss

Miss Status Holding Register (MSHR)

- **MSHR 包含正在等待处理的失效**
  - 相同的块可以包含多个未解决的Load/Store 失效
  - 可以有多个未解决的块地址
- **失效可以分为**
  - Primary: 第一次发起存取请求时的失效块
  - Secondary: 在后续过程中的失效
  - Structural Stall miss： MSHR 硬件资源耗尽



Type: LD, SD, LW, SW, etc. **Offset**: block offset

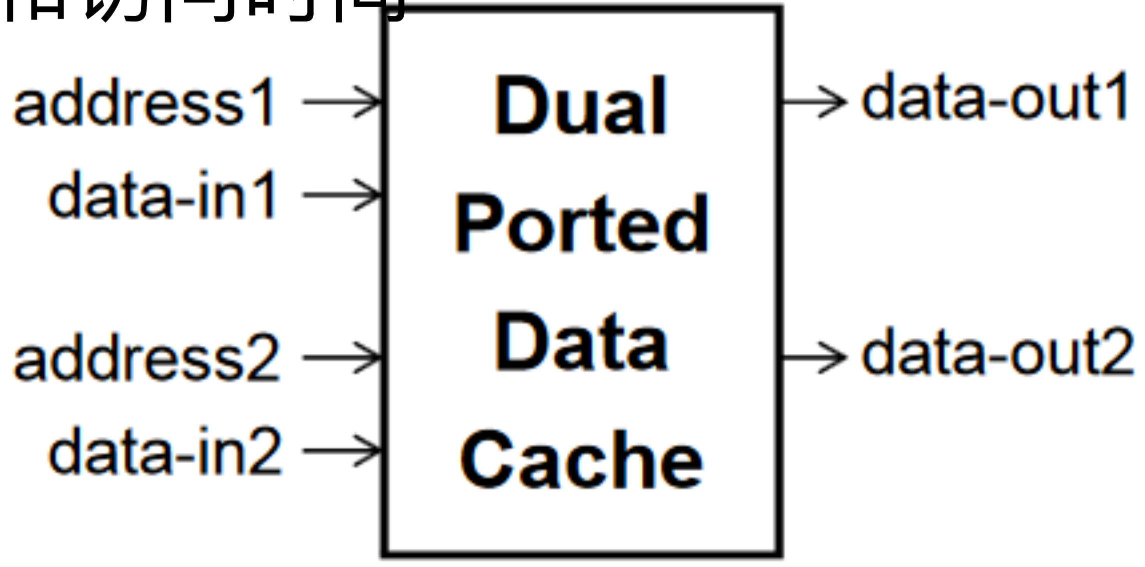
Destination register for load or **Data** for store

NonBlocking Cache Operation

- **当Cache失效时，检查MSHR是否有匹配的块地址**
  - 如果有，为该地址分配新的load/store 表项
  - 如果没有，分配新的MSHR和load/store表项
  - 如果所有MSHR资源都分配完，则Stall（结构相关）
- **当从底层传输Cache块时**
  - 处理该块中的Load和Store指令引起的失效
  - Load：根据block offset从该块中装载数据到寄存器
  - Store：根据block offset将数据写入该块指定位置
  - 完成该块所有的失效的Load/Store后，释放MSHR中的对应表项

Multi-Ported Cache

- **Dual-Ported Data Cache**
  - 两个地址端口（每个Cycle支持两条load/store 指令）
  - 在现代处理器中保持较高的指令吞吐率
- **True Multi-ported Cache Design**
  - 所有的控制和数据通路在Cache中是多份的
    - Address Decoder, way multiplexor, tag comparator, aligners
    - Tag Array, Data Array
  - 显著地增加了Cache的面积和访问时间
- **Multi-Banked Cache**
  - 将cache组织成多个banks
  - 每个bank是一个端口
  - 可以并行访问不同的Bank



5、Multibanked Caches

- 将Cache组织为多个独立的banks，以支持并行访问
  - ARM Cortex-A8 supports 1-4 banks for L2
  - Intel i7 supports 4 banks for L1 and 8 banks for L2
-

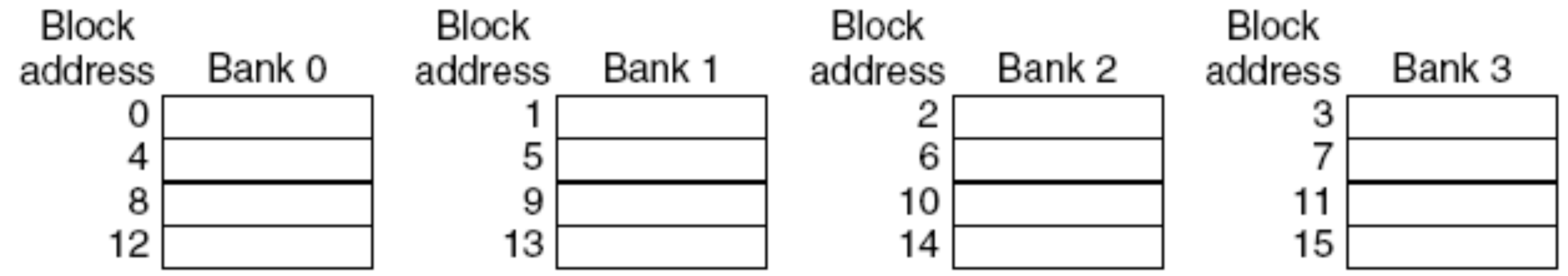


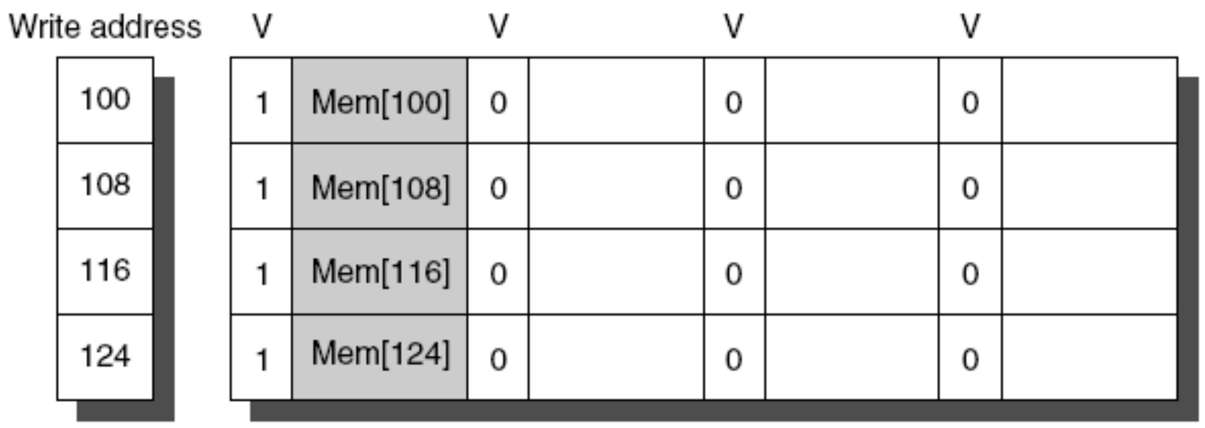
Figure 2.6 **Four-way interleaved cache banks using block addressing.** Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

6、Critical Word First, Early Restart

- **关键字优先**
  - 首先请求CPU所需要的字
  - 请求字一到达就发给CPU，使其继续执行，同时从存储器中调入其他字
- **提前重启**
  - 请求字的顺序不变
  - 请求字一到达就发给CPU，使其继续执行，同时从存储器中调入其他字
- **通常在块比较大时，这些技术才有效**

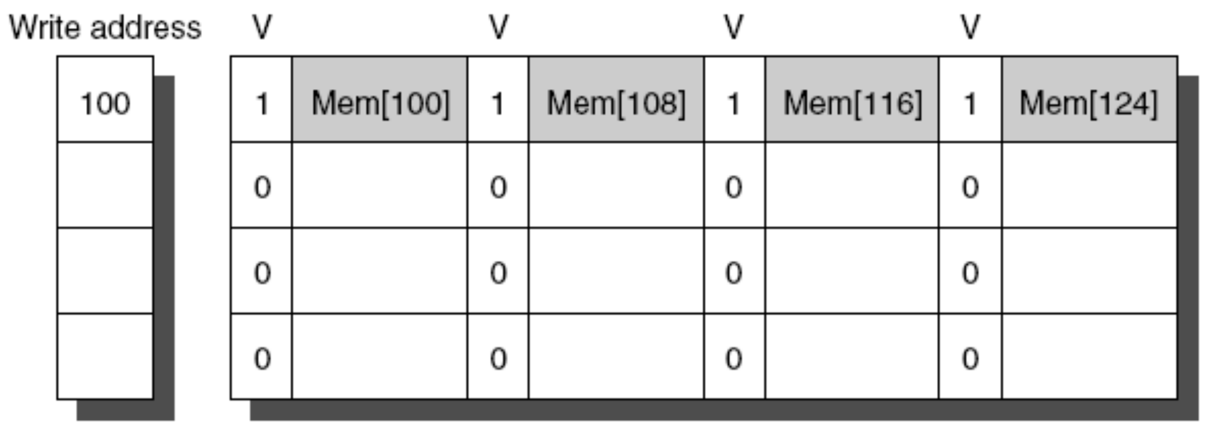
7、Merging Write Buffer

- 在向写缓冲器写入地址和数据时，如果写缓冲器中存在被修改过的块，就检查其地址，看看本次写入数据的地址是否与写缓冲器内的某个有效块地址匹配，如果匹配，就把新数据与该块合并，称为“合并写”
- 可以缓解由于写缓冲满而造成的CPU停顿



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 merging



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	

With Write merging

高级Cache优化方法

- **缩短命中时间**
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- **增加Cache带宽**
  - 3、Cache访问流水化
  - 4、无阻塞Cache
  - 5、多体Cache
- **减小失效开销**
  - 6、关键字优先和提前重启
  - 7、合并写
- **降低失效率**
  - 8、编译优化
- **通过并行降低失效开销或失效率**
  - 9、硬件预取
  - 10、编译器控制的预取

8、编译器优化

- **无需对硬件做任何改动，通过软件优化降低失效率**
- **研究从两方面展开：**
  - 减少指令失效
  - 减少数据失效
- **减少指令失效，重新组织程序（指令调度）而不影响程序的正确性**
  - 研究结果：通过使用profiling信息来判断指令组间可能发生的冲突，并将指令重新排序以减少失效。
  - 研究表明：
    - 容量为2KB, 块大小为 4Bytes的直接映象Icache，通过使用指令调度可以使失效率降低50%。容量增大到 8KB, 失效率可降低75%
    - 在有些情况下，当能够使某些指令不进入ICache时，可以得到最佳性能。即使不这样做，优化后（指令调度）的程序在直接映象Cache中的失效率也低于未优化程序在同样大小的8路组相联Cache中的失效率。
- **减少数据失效，主要通过优化来改善数据的空间局部性和时间局部性，基本方法为：**
  - 数据合并
  - 内外循环交换，循环融合
  - 分块

编译器优化方法举例之一：数组合并

```
// Original Code
for (i = 0; i < N; i++)
    a[i] = b[i] + c[i];
for (i = 0; i < N; i++)
    d[i] = a[i] + b[i] * c[i];
```

Blocks are replaced in first loop then accessed in second

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

Revised version takes advantage of temporal locality

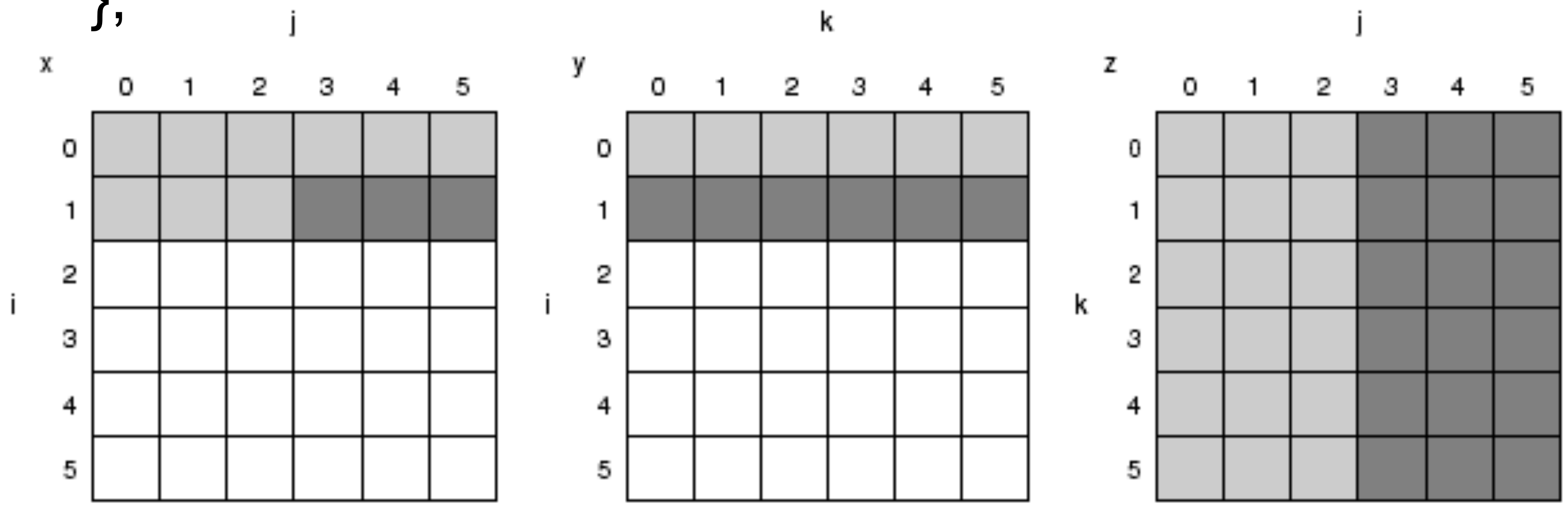
编译器优化方法举例之二：内外循环交换

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

Revised version takes advantage of spatial locality

编译器优化方法举例之三：分块（1/2）

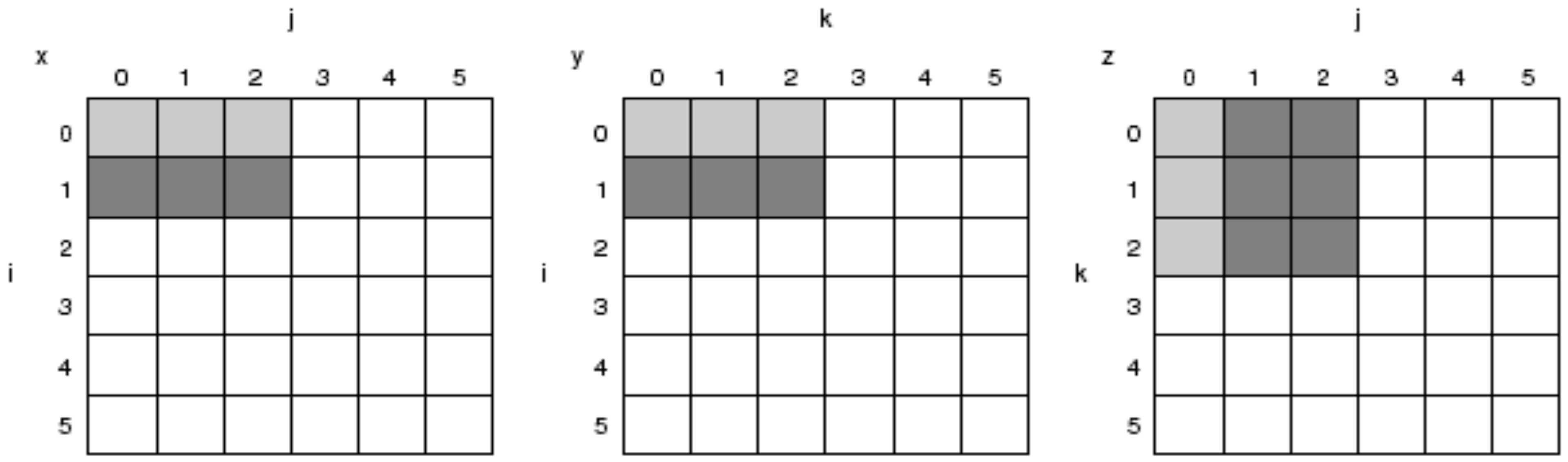
```
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
    {   r = 0;
        for (k = 0; k < N; k = k + 1)
            r = r + y[i][k]*z[k][j];
        x[i][j] = r;
    };
```



N^3次操作，产生的存储器访问次数为: $2N^3+N^2$

编译器优化方法举例之三：分块（2/2）

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



9、Hardware Prefetching

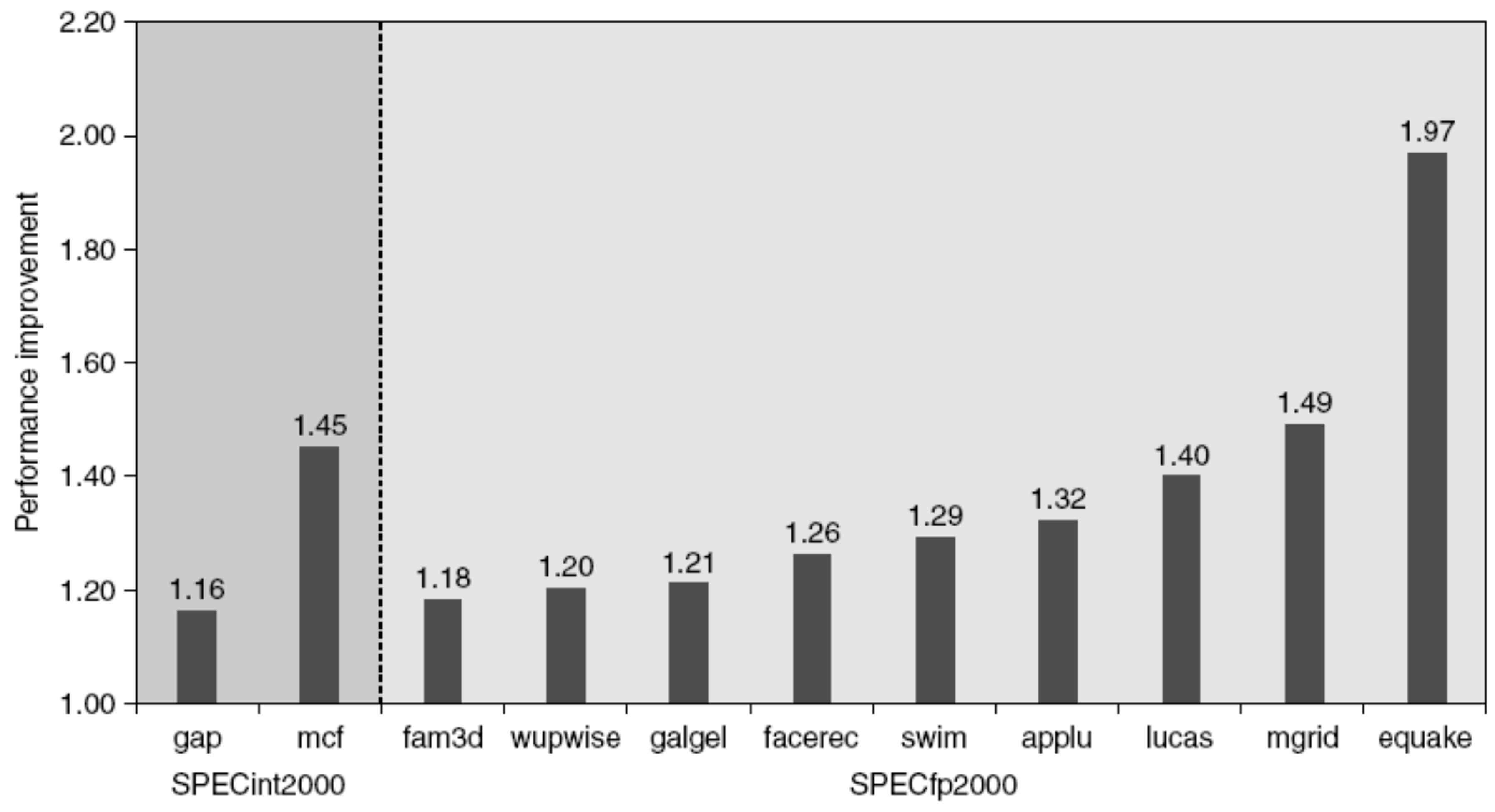
- **CPU在执行当前块代码时，硬件预取下一块代码**
  - CPU可能马上就要执行这块代码，这样可以降低或消除Cache的访问失效
- **当块中有控制指令时，预取失效**
- **预取的指令可以放在Icache中，也可以放在其他地方（存取速度比Memory块的地方）**
- **AXP21064失效时，取2块指令块**
  - 目标块放在Icache，下一块放在ISB(指令流缓冲）中
  - 如果访问的块在ISB中，取消访存请求，直接从ISB中读，并发出对下一指令块的预取访存请求
- **研究结果：块大小为16字节，容量为4KB的直接映象Cache，1个块的指令流缓冲器，可以捕获15%－25%的失效，4个块 ISB可捕获50%的失效，16块ISB可捕获72%的失效**

硬件预取

- **预取数据**
  - 出发点：CPU访问一块数据，可能马上要访问下一块数据
  - Jouppi研究结果：块大小16字节，4KB直接映象Cache， 1Block DSB－25%
  - 4Block DSB － 43%
  - Palacharla和Kessler 1994年研究
    - 一个具有两个64KB四路组相联(Icache, Dcache)的处理器来说，8Blocks流缓冲器能够捕获其50%－70%的失效
- **举例：Alpha AXP21064采用指令预取技术，其实际失效率是多少？若不采用指令预取技术，Alpha AXP 21064的指令Cache必须为多大才能保持平均访存时间不变？**
  - 假设当指令不在指令Cache中，在预取缓冲区中找到时，需要多花1个时钟周期。
  - 假设预取命中率为25%，命中时间为1个时钟周期，失效开销为50个时钟周期
  - 8KB指令Cache的失效率为1.10%，16KB指令cache的失效率为0.64%
  - AMAT（预取）= 命中时间+失效率*预取命中率*1+失效率*（1-预取命中率）*失效开销
- **注意：预取是利用存储器的空闲带宽，而不是与正常的存储器操作竞争。**

硬件预取

Fetch two blocks on miss (include next sequential block)



Pentium 4 Pre-fetching

10、Compiler Prefetching（1/2)

- **在ISA中增加预取指令，让编译器控制预取**
- **预取的种类**
  - 寄存器预取：把数据取到R中
  - Cache预取：只将数据取到Cache中，不放入寄存器
- **故障问题**
  - 两种类型的预取可以是故障性预取，也可以是非故障性预取
  - 所谓故障性预取指在预取时若出现虚地址故障，或违反保护权限，就会有异常发生
  - 非故障性预取，如导致异常就转化为空操作
- **只有在预取时，CPU可以继续执行的情况下，预取才有意义**
  - Cache在等待预取数据返回的同时，可以正常提供指令和数据，称为非阻塞Cache或非锁定Cache

由编译器控制预取 （2/2)

- **循环是预取优化的主要目标**
  - 失效开销较小时，Compiler简单的展开一两次，调度好预取与执行的重叠
  - 失效开销较大时，编译器将循环体展开多次，以便为较远的循环预取数据
  - 由于发出预取指令需要花费一条指令的开销，因此要避免不必要的预取
  - 重点放在那些可能导致失效的访问
- **举例1：P93（70） （Hennessy & Patterson 5th /中译本）**
- **举例2：P94（71）（Hennessy & Patterson 5th/中译本）**

举例1

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];
```

```
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];}
```

举例2

Example：

Calculate the time saved in the example above.

(1) Ignore instruction cache misses and assume there are no conflict or capacity misses in the data cache.

(2) Assume that prefetches can overlap with each other and with cache misses, thereby transferring at the maximum memory bandwidth.

(3) 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.

Acknowledgements

- **These slides contain material developed and copyright by:**
  - John Kubiatowicz (UCB)
  - Krste Asanovic (UCB)
  - David Patterson (UCB)
  - Chenxi Zhang (Tongji)
- **UCB material derived from course CS152、CS252、CS61C**
- **KFUPM material derived from course COE501、COE502**