

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

4.1 存储层次结构

- 存储系统设计是计算机体系结构设计的关键问题之一
 - 价格，容量，速度的权衡
- 用户对存储器的“容量，价格和速度”要求是相互矛盾的
 - 速度越快，每位价格就高
 - 容量越大，每位价格就低
 - 容量越大，速度就越慢
 - 目前主存一般由DRAM构成
- Microprocessor与Memory之间的性能差异越来越大
 - CPU性能提高大约60%/year
 - DRAM 性能提高大约 9%/year

技术发展趋势

	Capacity	Speed (latency)
Logic:	2x in 3 years	2x in 3 years
DRAM:	4x in 3 years	2x in 10 years
Disk:	4x in 3 years	2x in 10 years

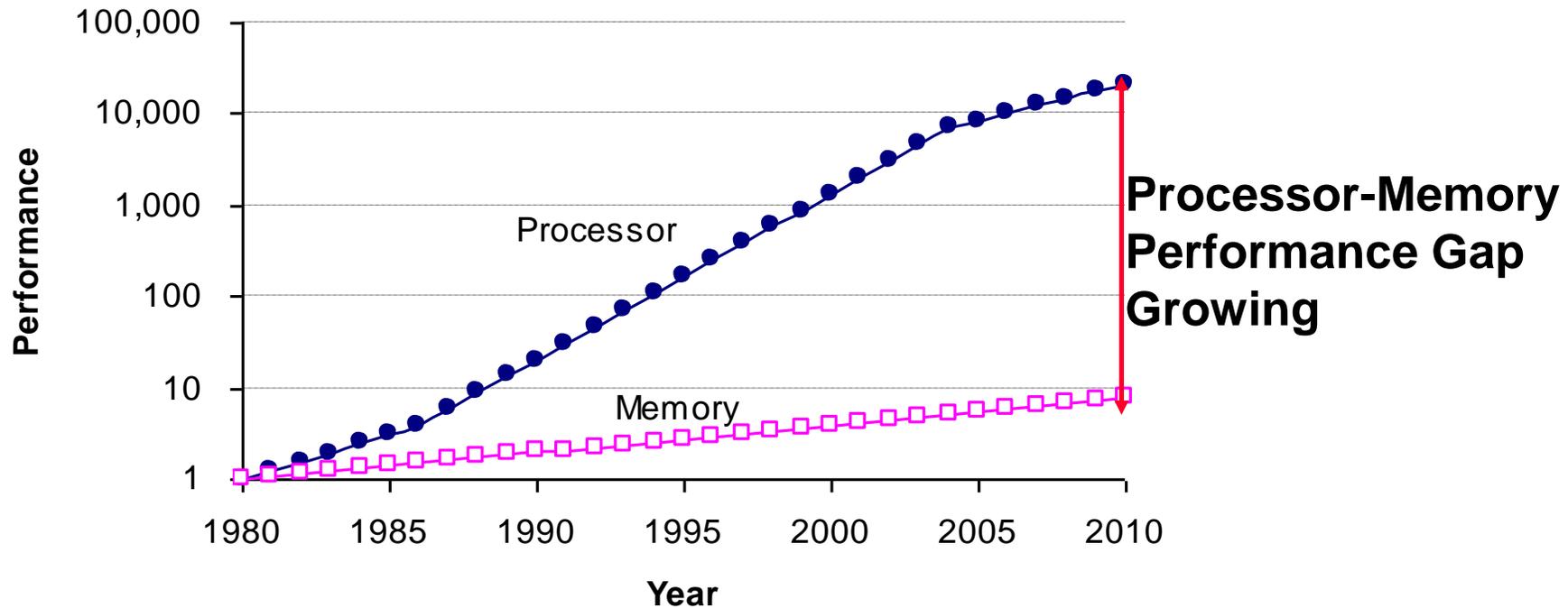
Year	DRAM Size	Cycle Time
1980	64 Kb	250 ns
1983	256 Kb	220 ns
1986	1 Mb	190 ns
1989	4 Mb	165 ns
1992	16 Mb	145 ns
1995	64 Mb	120 ns

Annotations: 1000:1! (between 1980 and 1995 DRAM Size), 2:1! (between 1980 and 1995 Cycle Time)

2009 8192 (8 Gbi)

微处理器与DRAM 的性能差异

Processor-DRAM Memory Gap (latency)



Microprocessor-DRAM性能差异

□利用caches来缓解微处理器与存储器性能上的差异

□Microprocessor-DRAM 性能差异

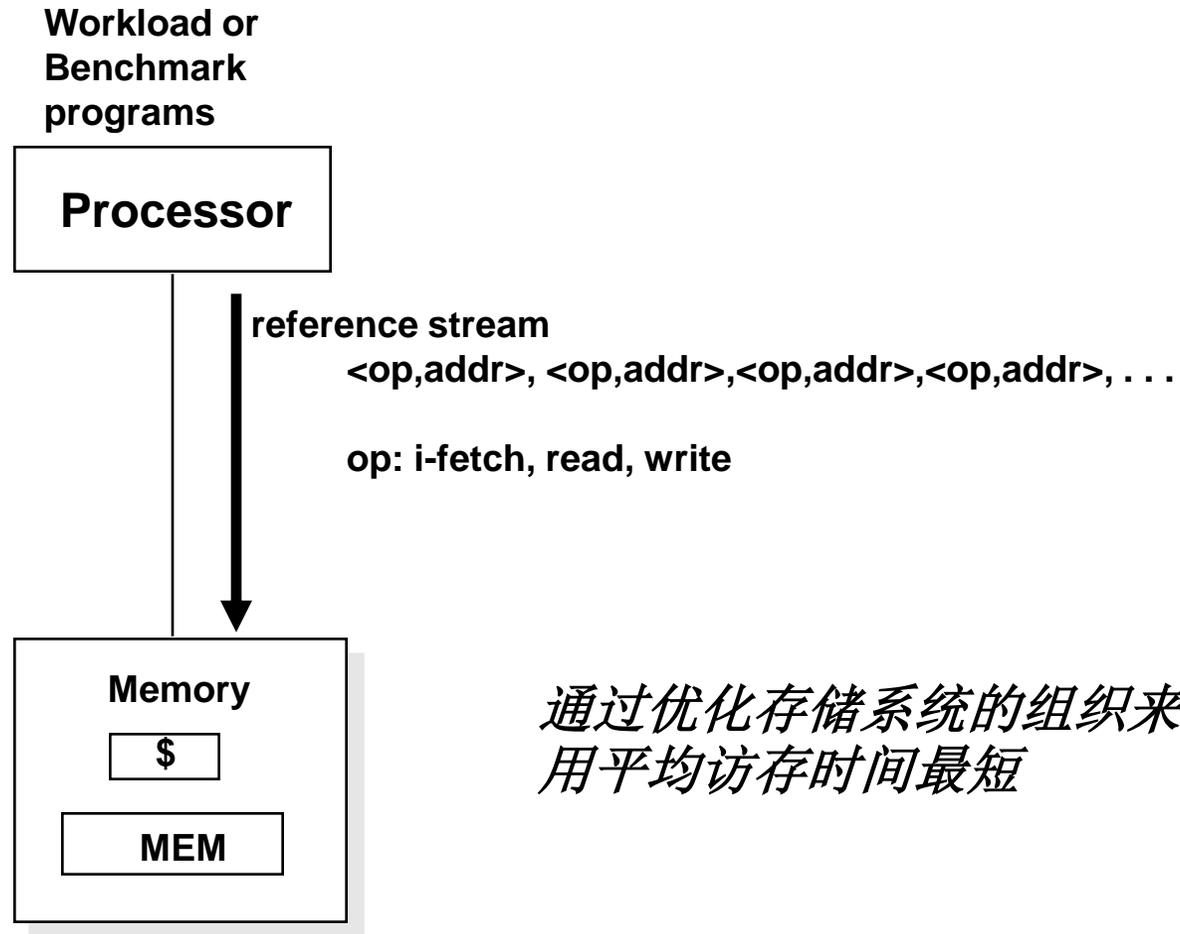
● time of a full cache miss in instructions executed

1st Alpha : 340 ns/5.0 ns = 68 clks x 2 or 136 instructions

2nd Alpha : 266 ns/3.3 ns = 80 clks x 4 or 320 instructions

3rd Alpha : 180 ns/1.7 ns =108 clks x 6 or 648 instructions

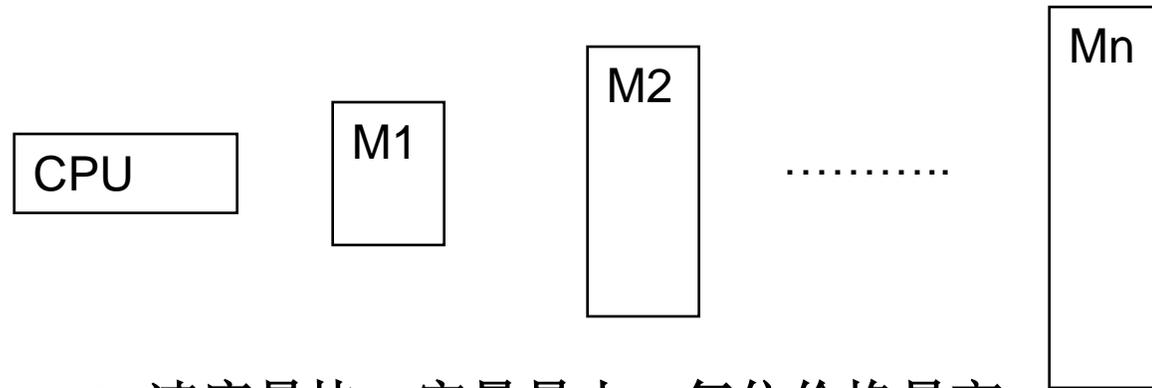
存储系统的设计目标



通过优化存储系统的组织来使得针对典型应用平均访存时间最短

基本解决方法：多级层次结构

□ 多级分层结构

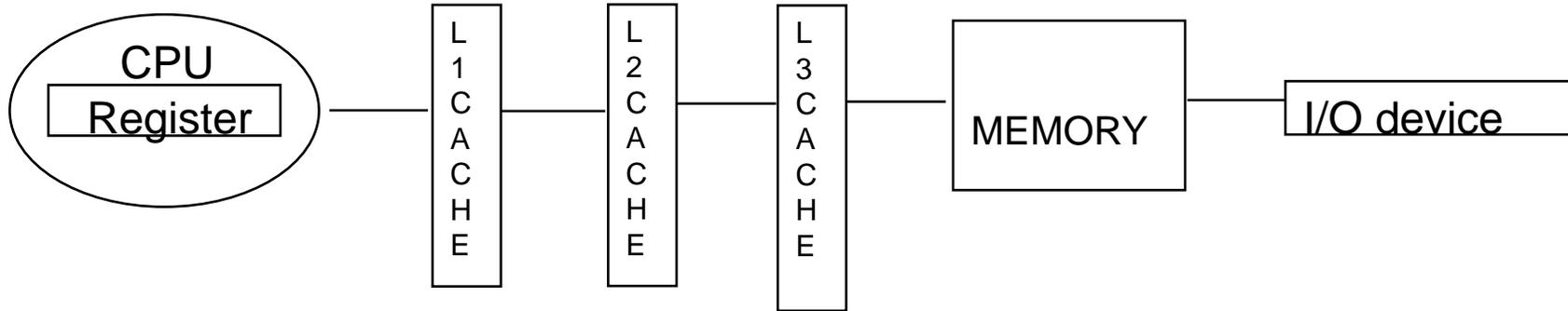


- **M1** 速度最快，容量最小，每位价格最高
- **Mn**速度最慢，容量最大，每位价格最低

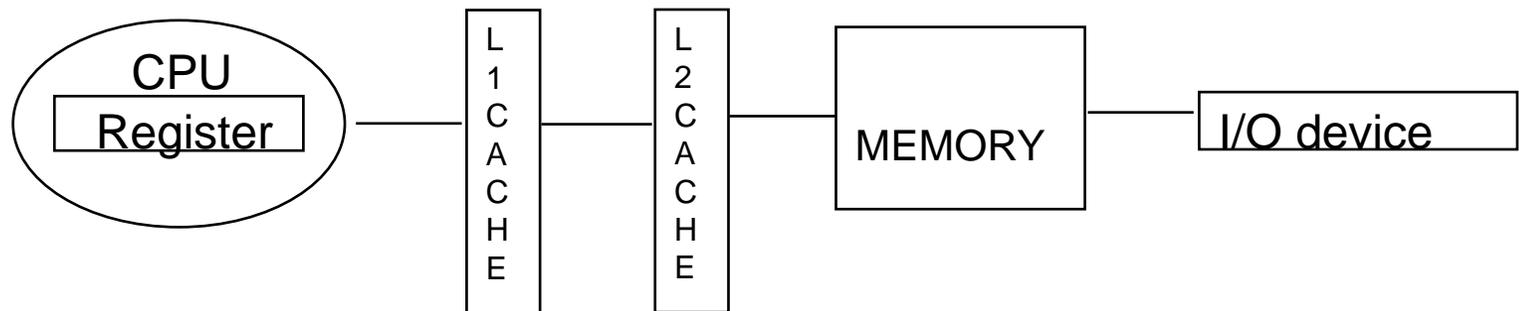
□ 并行

□ 存储系统接近M1的速度，容量和价格接近Mn

现代计算机系统的多级存储层次



300ps	1ns	3-10ns	10-20ns	50-100ns	5-10ms
1000B	64KB	256K	2-4MB	4-16GB	4-16TB

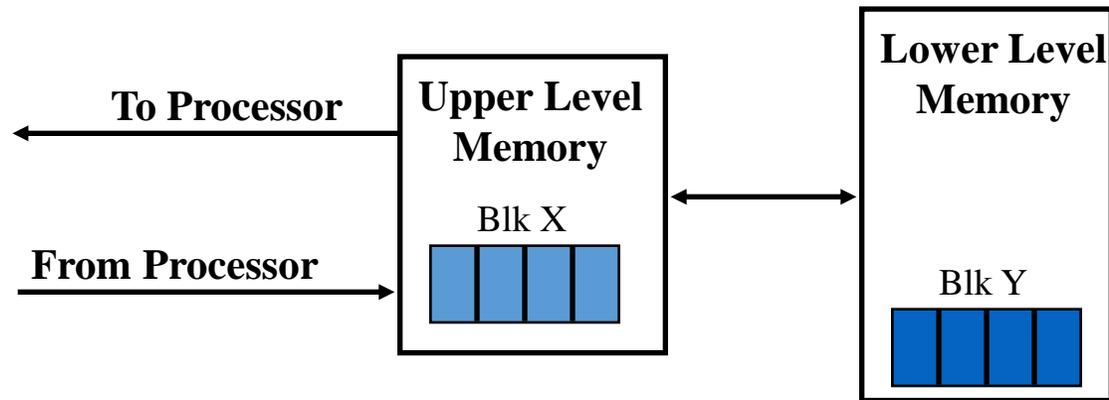


500ps	2ns	10-20ns	50-100ns	25-50μs
500B	64KB	256K	256-512GB	4-8GB

- 应用程序局部性原理: 给用户
- 一个采用低成本技术达到的存储容量. (容量大, 价格低)
 - 一个采用高速存储技术达到的访问速度. (速度快)

存储层次工作原理： Locality!

- **Temporal Locality** (时间局部性):
=>保持最近访问的数据项最接近微处理器
- **Spatial Locality** (空间局部性):
=>以由地址连续的若干个字构成的块为单位，从低层复制到上一层



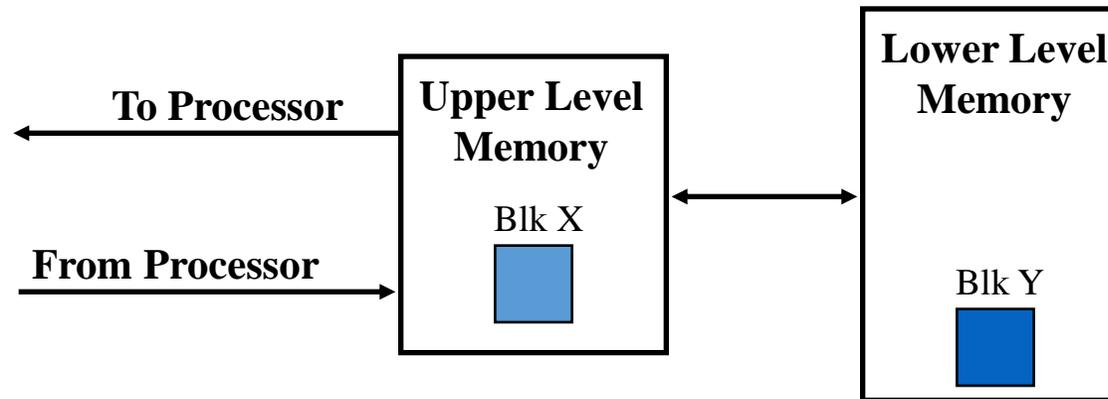
存储层次结构涉及的基本概念

- Block
 - Block : 不同层次的Block大小可能不同
 - 命中和命中率
 - 失效和失效率
- 镜像和一致性问题
 - 高层存储器是较低层存储器的一个镜像
 - 高层存储器内容的修改必须反映到低层存储器中
 - 数据一致性问题
- 寻址：不管如何组织，我们必须知道如何访问数据
- 要求：我们希望不同层次上块大小是不同的
 - 在L0 cache 可能以Double, Words, Halfwords, 或bytes
 - 在L1cache仅以cache line 或 slot为单位访问
 - 在更低层.....
 - 因此总是存在地址映射问题
 - 物理地址格式 $\text{Block Frame Address} + \text{Block Offset}$

存储层次的性能参数 (1/2)

假设采用二级存储：M1和M2

- M1和M2的容量、价格、访问时间分别为：
 S_1 、 C_1 、 T_{A1} S_2 、 C_2 、 T_{A2}



存储层次的性能参数 (2/2)

□ 存储层次的平均每位价格C

- $C=(C_1*S_1+C_2*S_2)/(S_1+S_2)$

□ 命中(Hit): 访问的块在存储系统的较高层次上

- 若一组程序对存储器的访问，其中 N_1 次在M1中找到所需数据， N_2 次在M2中找到数据 则
- Hit Rate (命中率): 存储器访问在较高层命中的比例 $H= N_1/(N_1+N_2)$
- Hit Time (命中时间): 访问较高层的时间, T_{A1}

□ 失效(Miss):访问的块不在存储系统的较高层次上

- Miss Rate (失效) = $1 - (\text{Hit Rate}) = 1 - H = N_2/(N_1+N_2)$
- 当在M1中没有命中时: 一般必须从M2中将所访问的数据所在块搬到M1中, 然后CPU才能在M1中访问。
- 设传送一个块的时间为 T_B , 即不命中时的访问时间为: $T_{A2}+T_B+T_{A1} = T_{A1}+T_M$
TM 通常称为失效开销

□ 平均访存时间:

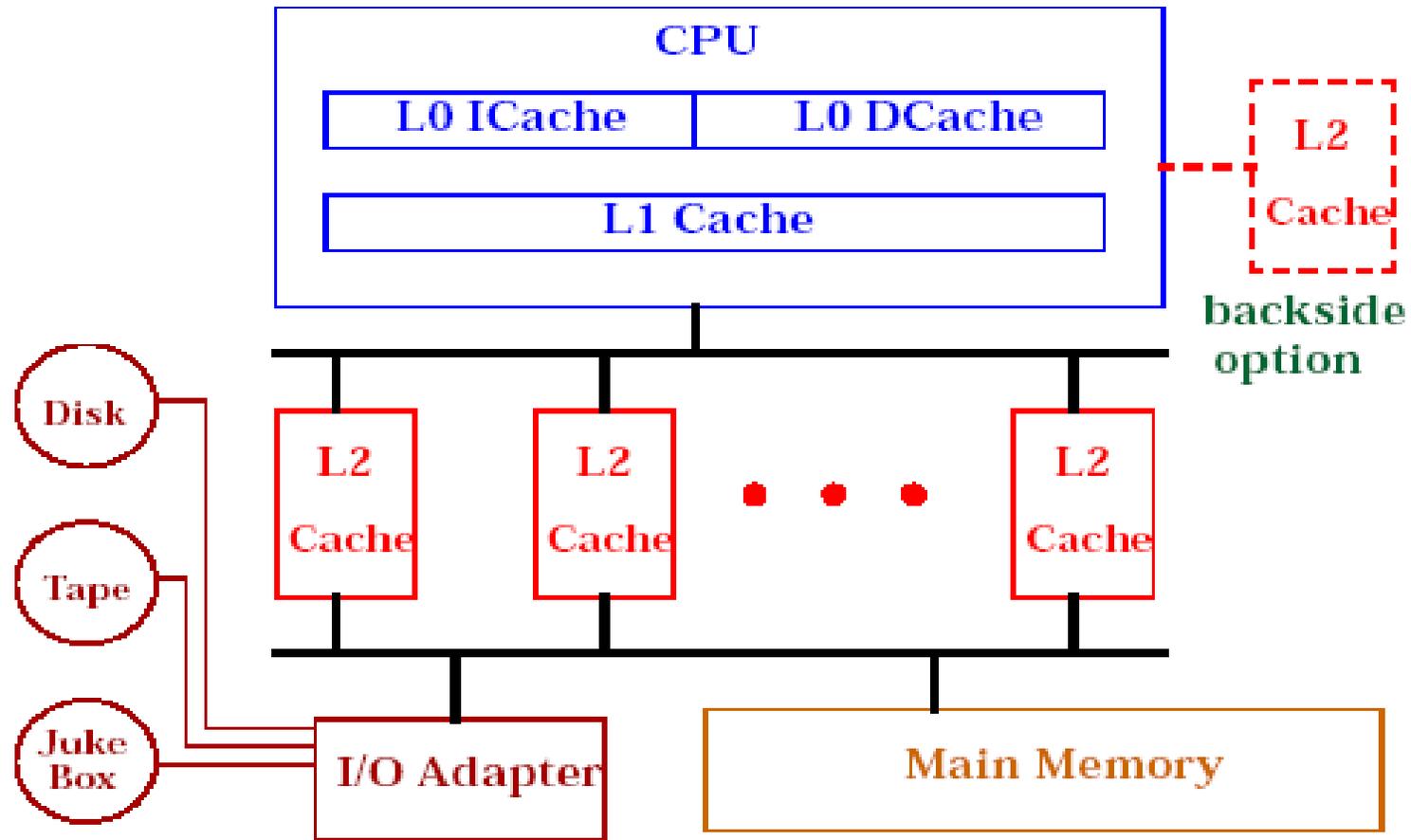
- 平均访存时间 $T_A = HT_{A1}+(1-H)(T_{A1}+T_M) = T_{A1}+(1-H)T_M$

常见的存储层次的组织

- Registers <-> Memory
 - 由编译器完成调度
- cache <-> memory
 - 由硬件完成调度
- memory <-> disks
 - 由硬件和操作系统（虚拟管理）
 - 由程序员完成调度

4.2 Cache基本知识

Sample Memory Hierarchy



Q1: 映象规则

- 当要把一个块从主存调入Cache时，如何放置问题
- 三种方式
 - 全相联方式：即所调入的块可以放在cache中的任何位置
 - 直接映象方式：主存中每一块只能存放在cache中的唯一位置
一般，主存块地址*i*与cache中块地址*j*的关系为：

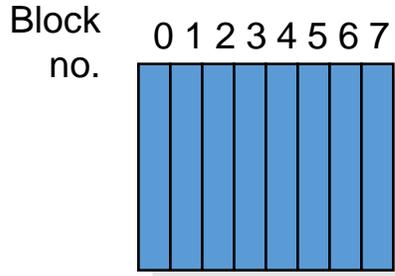
$$j = i \bmod (M), \quad M \text{ 为 cache 中的块数}$$

- 组相联映象：主存中每一块可以被放置在Cache中唯一的一个组中的任意一个位置，组由若干块构成，若一组由*n*块构成，我们称*N*路组相联
 - 组间直接映象
 - 组内全相联若cache中有*G*组，则主存中的第*i*块的组号*K*
 $K = i \bmod (G),$

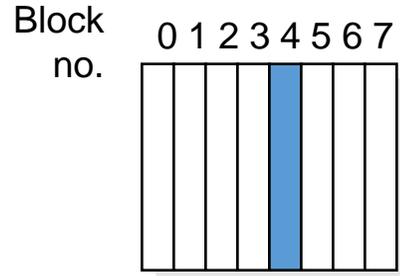
Q1: Where can a block be placed in the upper level?

- Block 12 placed in 8 block cache:
 - Fully associative, direct mapped, 2-way set associative
 - S.A. Mapping = Block Number Modulo Number Sets

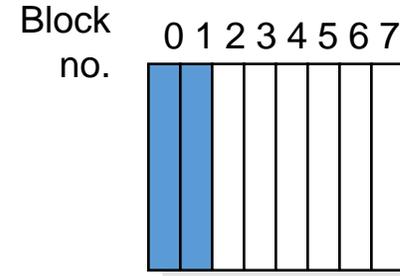
Fully associative:
block 12 can go
anywhere



Direct mapped:
block 12 can go
only into block 4
(12 mod 8)



Set associative:
block 12 can go
anywhere in set 0
(12 mod 4)



Block-frame address



Q1的讨论

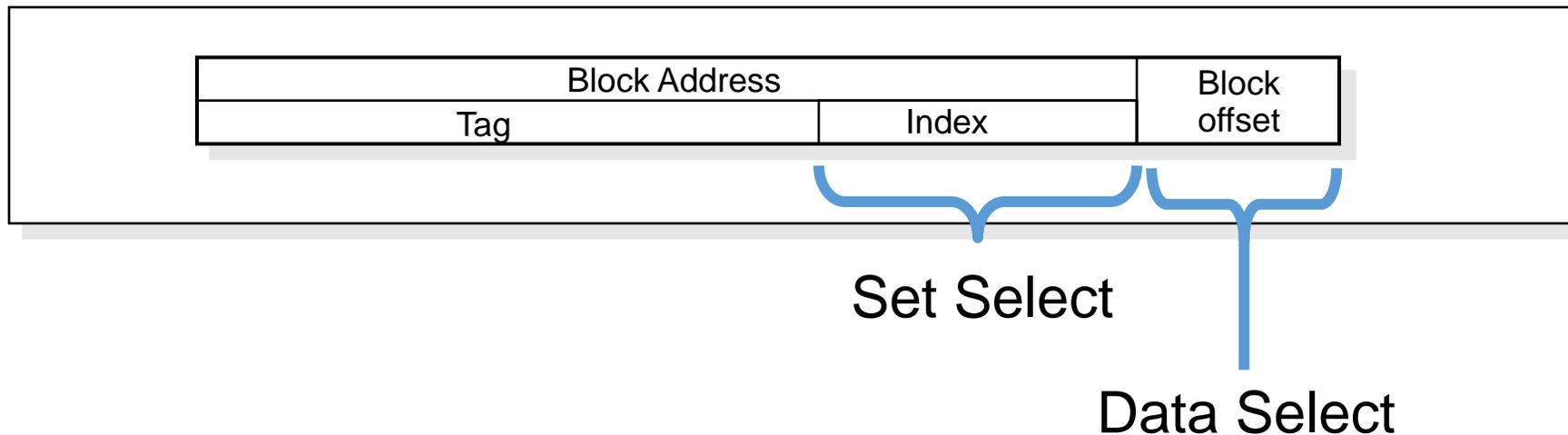
- N-Way组相联：如果每组由N个块构成，cache的块数为M，则cache的组数G为M/N
- 不同相联度下的路数和组数

	路数	组数
全相联	M	1
直接相联	1	M
其他组相联	$1 < N < M$	$1 < G < M$

- 相联度越高，cache空间利用率就越高，块冲突概率就越小，失效率就越低
- N值越大，失效率就越低，但Cache的实现就越复杂，代价越大
- 现代大多数计算机都采用直接映象，两路或四路组相联。

Q2 (1/2) : 查找方法

- 在CACHE中每一block都带有tag域（标记域），标记分为两类
 - Address Tags: 标记所访问的单元在哪一块中，这样物理地址就分为三部分：Address Tags ## Block index## block Offset
全相联映象时，没有Block Index
显然 Address tag越短，查找所需代价就越小
 - Status Tags: 标记该块的状态，如Valid, Dirty等

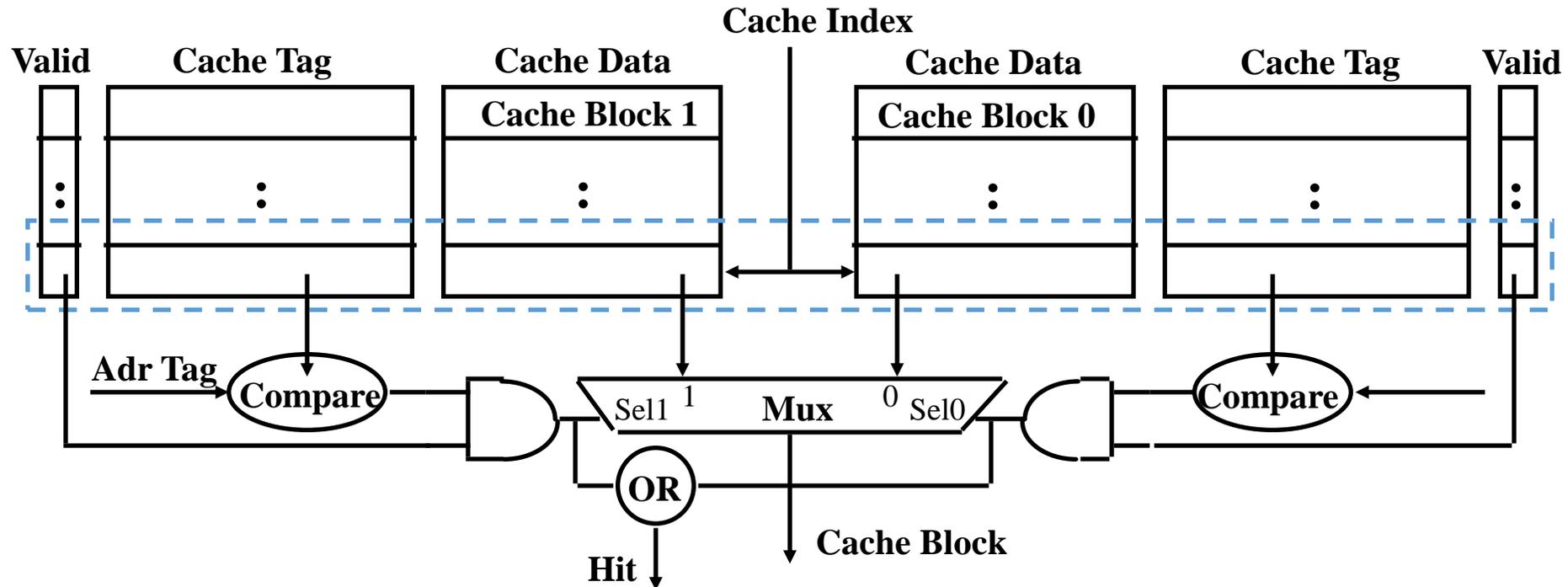


Q2 (2/2) 查找方法

- 原则：所有可能的标记并行查找，cache的速度至关重要，即并行查找
- 并行查找的方法
 - 用相联存储器实现，按内容检索
 - 用单体多字存储器和比较器实现
- 显然相联度 N 越大，实现查找的机制就越复杂，代价就越高
- 无论直接映象还是组相联，查找时，只需比较 tag，index 无需参加比较

Example: Set Associative Cache

- **N-way set associative:** 每一个cache索引对应N个cache entries
 - 这N个cache项并行操作
- **Example: Two-way set associative cache**
 - Cache index 选择cache中的一组
 - 这一组中的两块对应的Tags与输入的地址同时比较
 - 根据比较结果选择数据



Q3: 替换算法

- 主存中块数一般比cache中的块多，可能出现该块所对应的一组或一个Cache块已全部被占用的情况，这时需强制腾出其中的某一块，以接纳新调入的块，替换哪一块，这是替换算法要解决的问题：
 - 直接映象，因为只有一块，别无选择
 - 组相联和全相联有多种选择
- 替换方法
 - 随机法 (Random)，随机选择一块替换
 - 优点：简单，易于实现
 - 缺点：没有考虑Cache块的使用历史，反映程序的局部性较差，失效率较高
 - FIFO—选择最早调入的块
 - 优点：简单
 - 虽然利用了同一组中各块进入Cache的顺序，但还是反映程序局部性不够，因为最先进入的块，很可能是经常使用的块
 - 最近最少使用法 (LRU) (Least Recently Used)
 - 优点：较好的利用了程序的局部性，失效率较低
 - 缺点：比较复杂，硬件实现较困难

LRU和Random的比较（失效率）

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

Data Cache misses per 1000 instructions comparing LRU, Random, FIFO replacement for several sizes and associativities.

These data were collected for a block size of 64 bytes for the Alpha architecture using 10 SPEC2000 benchmarks. Five are from SPECint2000(gap, gcc, gzip, mcf and perl) and five are from SPECfp2000(applu, art, equake, lucas and swim)

□ 观察结果（失效率）

- 相联度高，失效率较低。
- Cache容量较大，失效率较低。
- LRU在Cache容量较小时，失效率较低
- 随着Cache容量的加大，Random的失效率在降低

Q4: 写策略

- 程序对存储器读操作占26%，写操作占9%
 - 写所占的存储器访问比例 $9/(100+26+9)$ 大约为7%
 - 占访问数据Cache的比例: $9/(26+9)$ 大约为25%
- 大概率事件优先原则—优化Cache的读操作
- Amdahl定律: 不可忽视“写”的速度
- “写”的问题
 - 读出标识, 确认命中后, 对Cache写 (串行操作)
 - Cache与主存内容的一致性问题
- 写策略就是要解决: 何时更新主存问题

两种写策略

□ 写直达法 (Write through)

- 优点: 易于实现, 容易保持不同层次间的一致性
- 缺点: 速度较慢

□ 写回法

- 优点: 速度快, 减少访存次数
- 缺点: 一致性问题

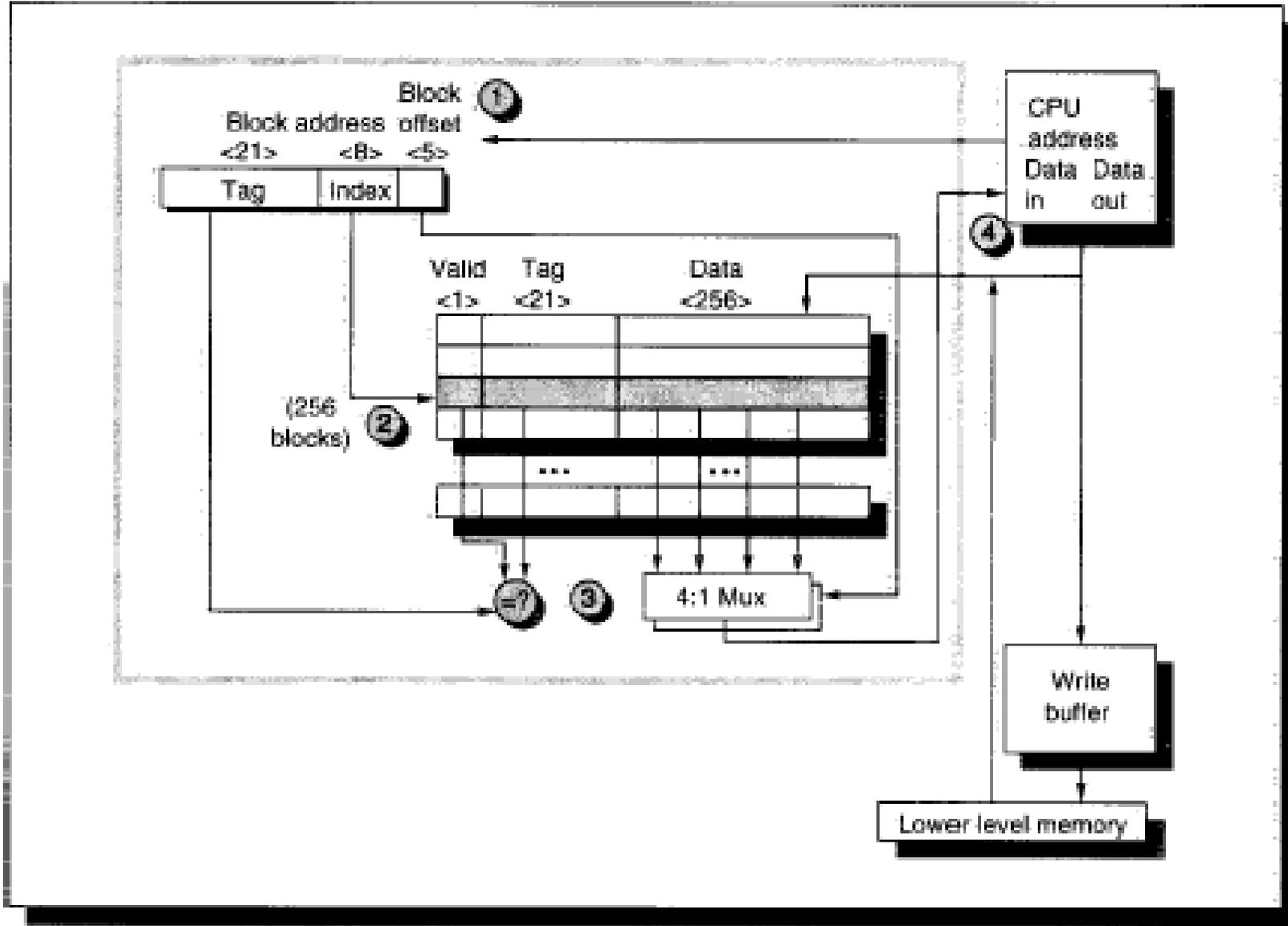
□ 当发生写失效时的两种策略

- 按写分配法(Write allocate): 写失效时, 先把所写单元所在块调入Cache, 然后再进行写入, 也称写时取 (Fetch on Write)方法
- 不按写分配法 (no-write allocate): 写失效时, 直接写入下一级存储器, 而不将相应块调入Cache, 也称绕写法 (Write around)
- 原则上以上两种方法都可以应用于写直达法和写回法, 一般情况下
 - Write Back 用Write allocate
 - Write through 用no-write allocate

Alpha AX 21064 Cache结构（数据Cache）

- 基本技术特性
 - 容量 8KB，Block 32B，共256个Block
 - 直接映象
 - 写直达法，写失效时，no-write allocate 方法
 - 每个字为8个字节
- 21064物理地址34位
 - 21位tag##8位index ##5位块内偏移
- Cache命中的步骤
 - 读命中
 - 写命中
- Cache失效
 - Cache向CPU发暂停信号
 - 块传送，21064 Cache与下一级存储器之间数据通路16字节，传送全部32字节需要10个cycles

Alpha AX 21064 Cache结构 (数据Cache)



Alpha 21264 Data Cache

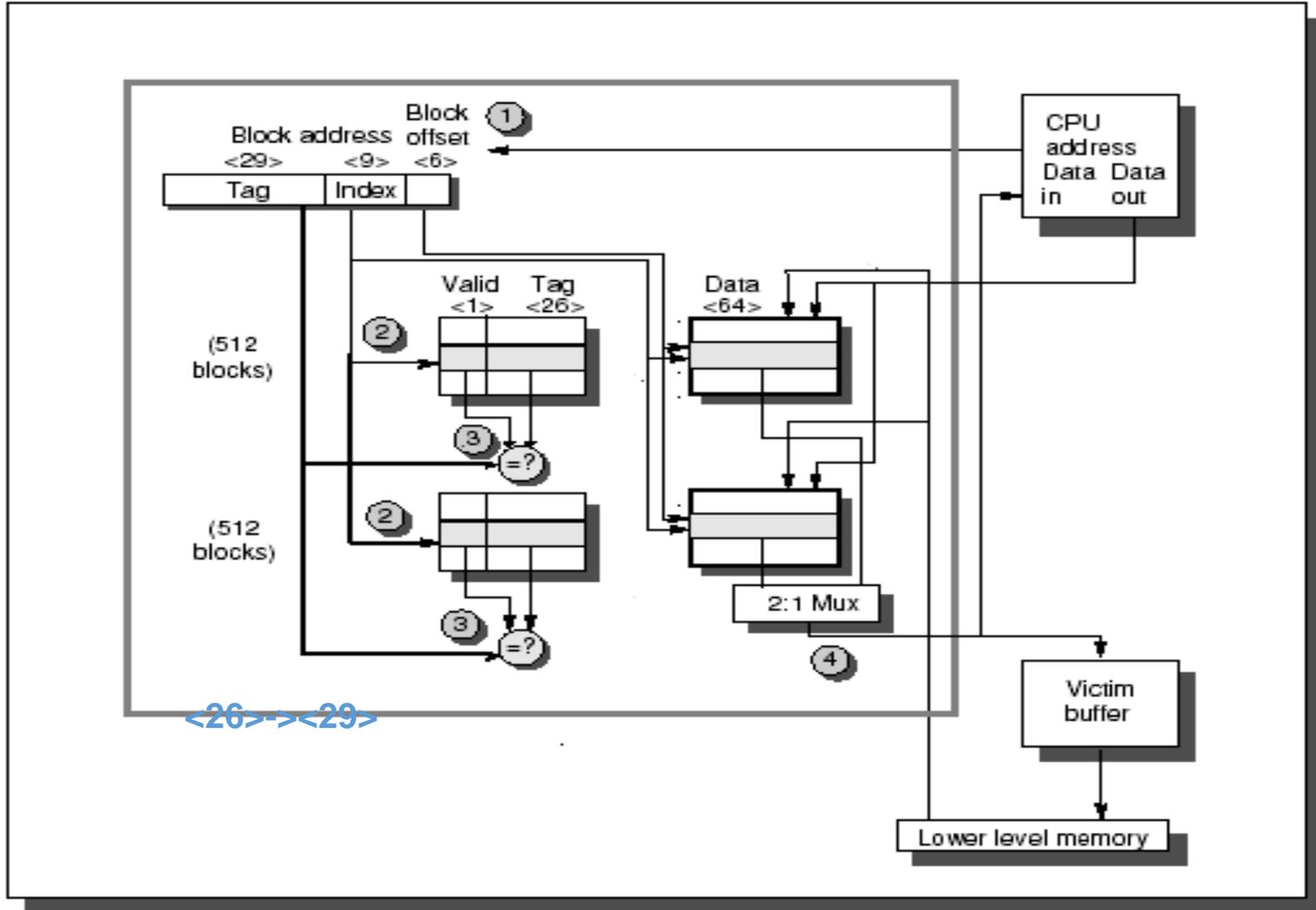
□ 基本技术特征

- Cache size: 64Kbytes
- Block size: 64-bytes
- Two-way 组相联
- Write back, 写失效时, write allocate

□ 21264 48-bits 虚拟地址, 虚实映射为44-bits的物理地址, 也支持 43-bits虚拟地址, 虚实映射为41-bits的物理地址

- 29位tags##9位index##6位 Block offset

Alpha 21264 Data Cache



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.*

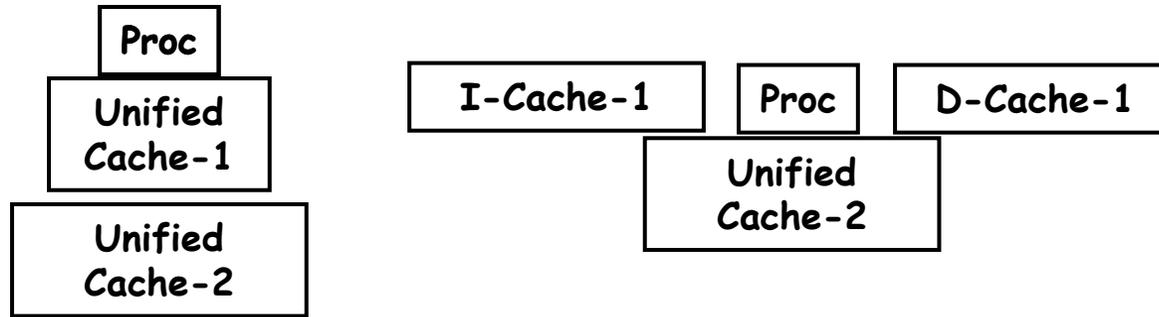
性能分析举例

- 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

$$\begin{aligned}
 &= 1.1(\text{cycles/ins}) + \\
 &\quad [0.30 (\text{DataMops/ins}) \\
 &\quad \quad \quad \times 0.10 (\text{miss/DataMop}) \times 50 (\text{cycle/miss})] + \\
 &\quad [1 (\text{InstMop/ins}) \\
 &\quad \quad \quad \times 0.01 (\text{miss/InstMop}) \times 50 (\text{cycle/miss})] \\
 &= (1.1 + 1.5 + .5) \text{ cycle/ins} = 3.1
 \end{aligned}$$
- 65% of the time the proc is stalled waiting for memory!
- $AMAT = (1/1.3) \times [1 + 0.01 \times 50] + (0.3/1.3) \times [1 + 0.1 \times 50] = 2.54$

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 \Rightarrow 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)

$$AMAT_{\text{Harvard}} = 75\% \times (1 + 0.64\% \times 50) + 25\% \times (1 + 6.47\% \times 50) = 2.05$$

$$AMAT_{\text{Unified}} = 75\% \times (1 + 1.99\% \times 50) + 25\% \times (1 + 1 + 1.99\% \times 50) = 2.24$$

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:

(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：

- 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2.0，时钟周期为2ns，平均每条指令访存1.3次
 - (2) 两种Cache容量均为64KB，块大小都是32B
 - (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1.1倍
 - (4) 两种结构的失效开销都是70ns（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，64KB直接映像Cache的失效率为1.4%，相同容量的两路组相联Cache 的失效率为1.0

失效开销与Out-of-Order执行的处理器

$$\frac{\text{MemoryStalCycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{TotalMissLatency} - \text{OverlappedMissLatency})$$

- 需要确定两个参数：
 - Length of memory latency
 - Length of latency overlap
- 例如：在前面的例子中，若假设允许处理器乱序执行，则对于直接映射方式，假设30%的失效开销可以覆盖（overlap），那么原来的70ns失效开销就变为49ns.

Summary of performance equations in this chapter

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

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

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

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

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

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

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

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

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

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

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

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

FIGURE 5.9 Summary of performance equations in this chapter. The first equation calculates with cache index size, but the rest help evaluate performance. The final two equations deal with multilevel caches, as 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

4.3 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)

- 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

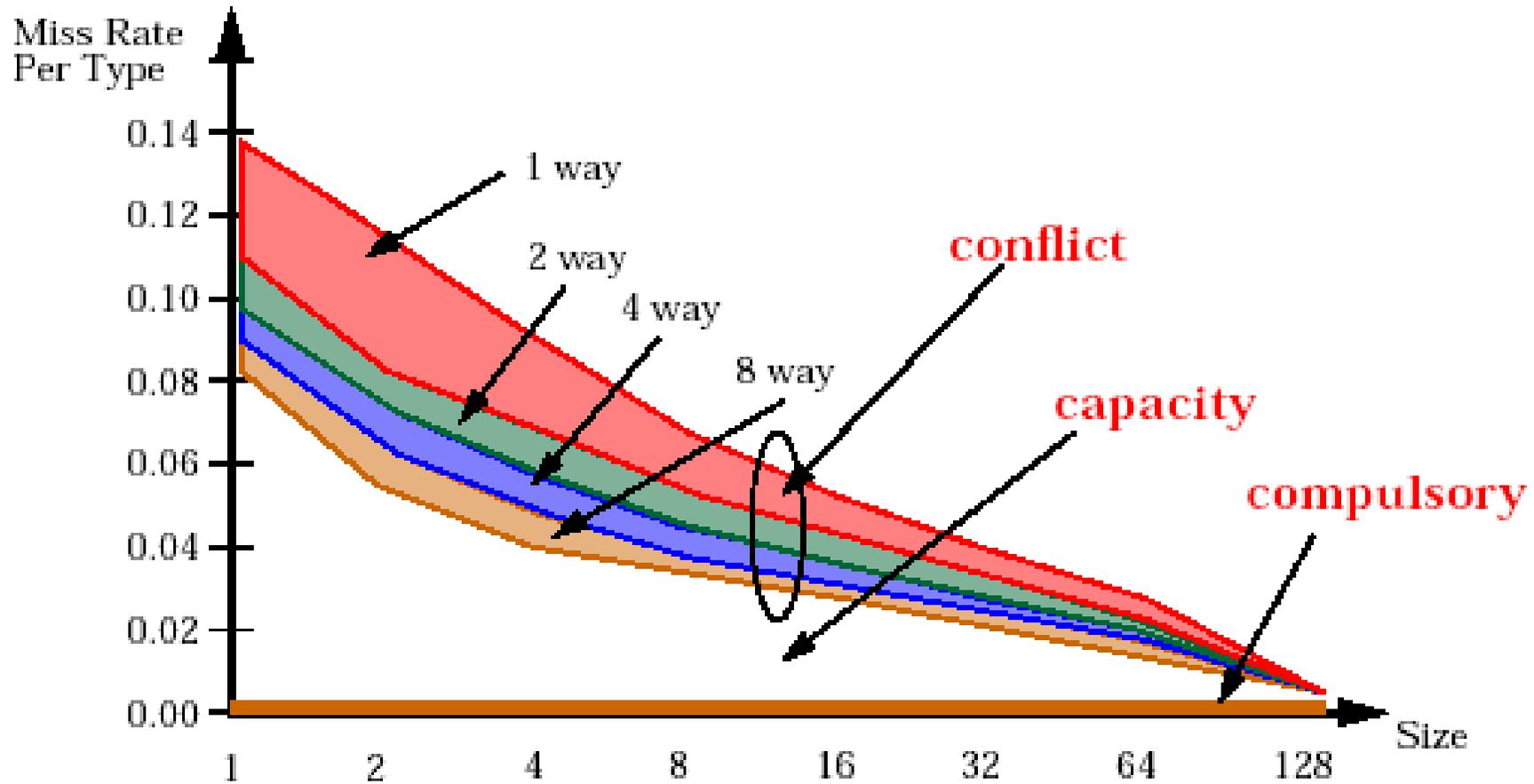
❑ 容量失效 (Capacity)

- 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
- 若太多的块映射到同一组（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



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Cache块大小 (1/2)

- 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？

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	11.76% / 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

增加Cache块大小 (2/2)

□ 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（因为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： $\text{平均访存时间} = \text{命中时间} + \text{失效率} \times \text{失效开销}$

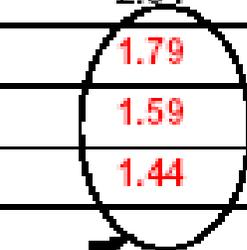
提高相联度

- ❑ 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

OOPS!



Victim Cache (1/2)

- 在Cache和Memory之间增加一个小的全相联Cache

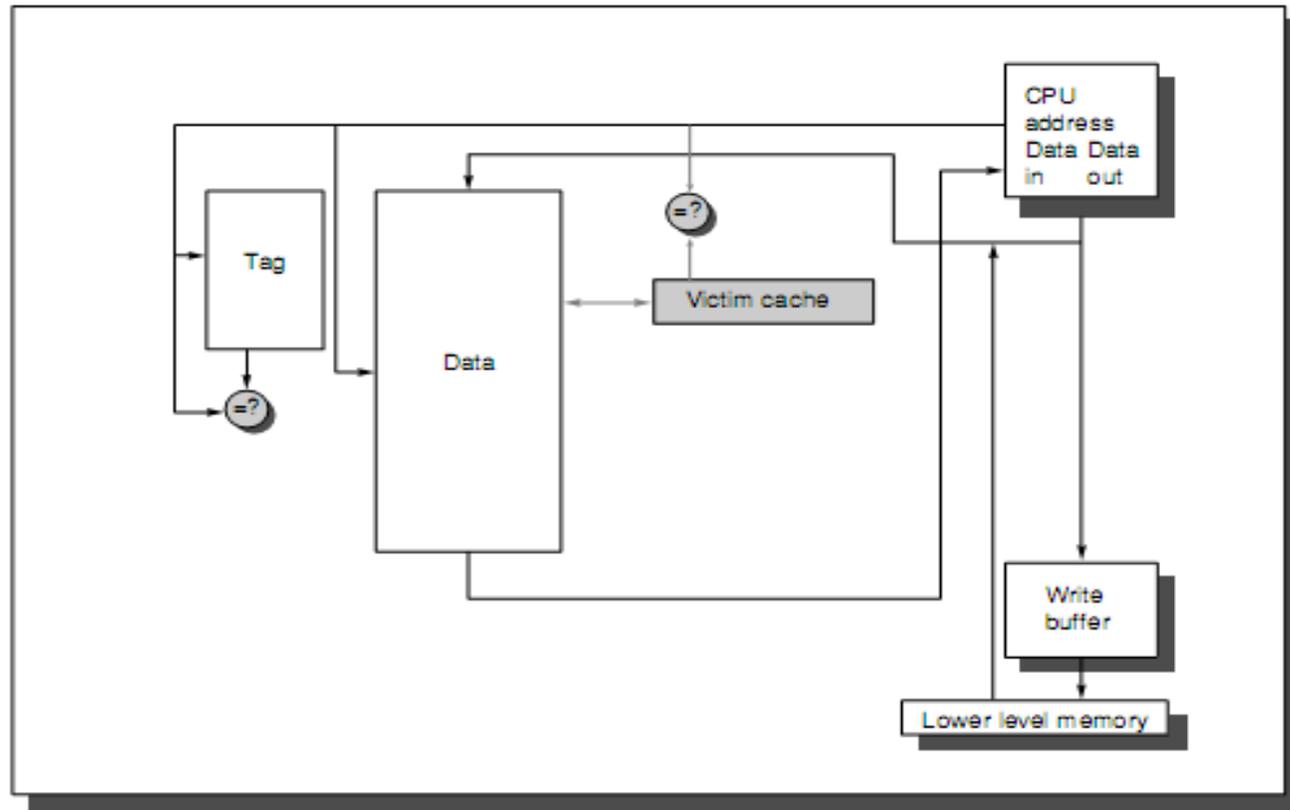


FIGURE 5.13 Placement of victim cache in the memory hierarchy. Although it reduces miss penalty, the victim cache is aimed at reducing the damage done by conflict misses, described in the next section. Jouppi [1990] found the four-entry victim cache could reduce the miss penalty for 20% to 95% of conflict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

Victim Cache (2/2)

□ 基本思想

- 通常Cache为直接映象，因而冲突失效率较大
- 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

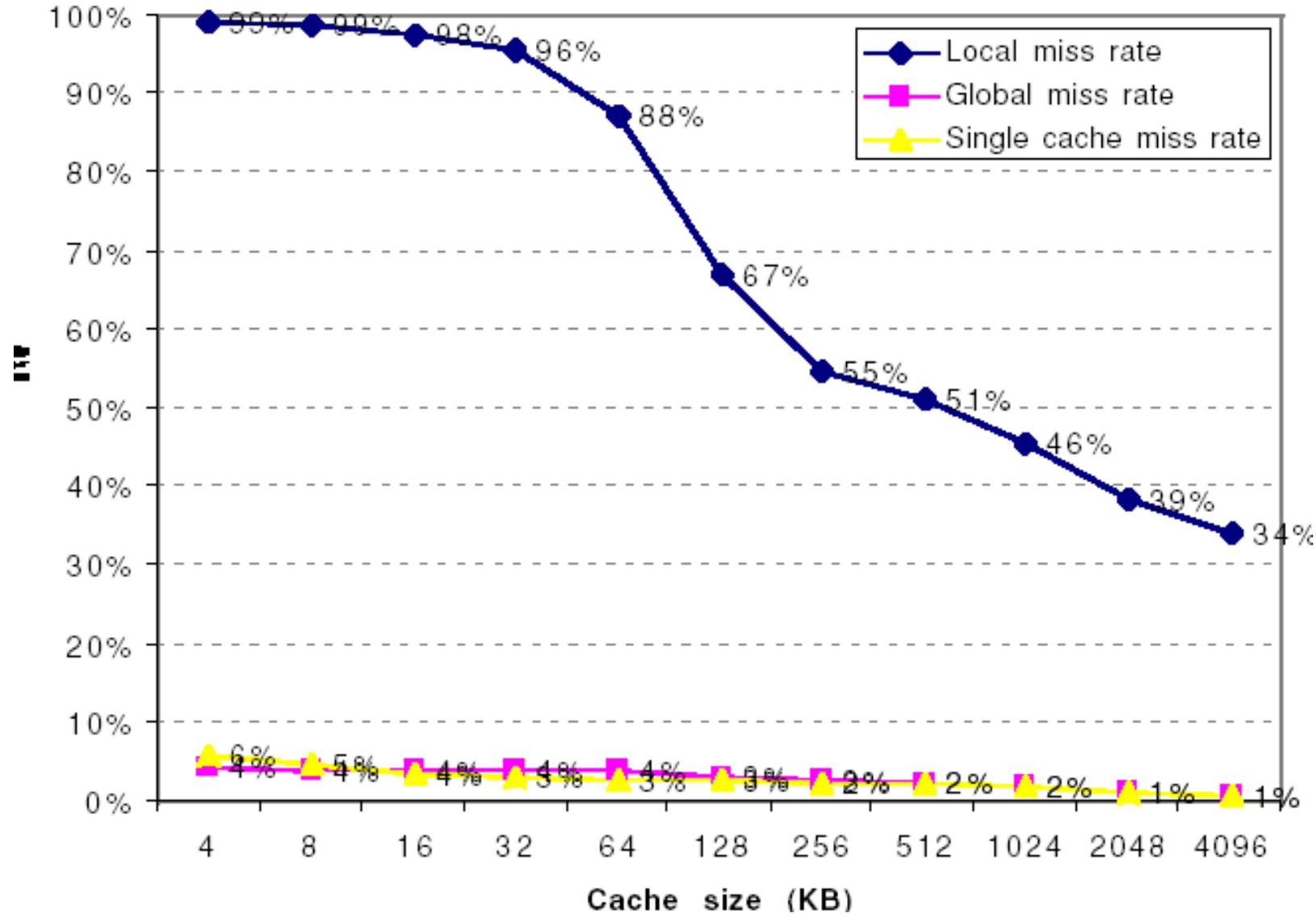
- 减少CPU与存储器间性能差异的重要手段
 - 平均访存时间 = 命中时间 + 失效率 × 失效开销

- 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

- 与CPU无关，重点是Cache与Memory之间的接口
- 问题：为了使Memory-CPU性能匹配，到底应该把Cache做的更快，还是应该把Cache做的更大
- 答案：两者兼顾。二级Cache –降低失效开销（减少访问存储器的次数。
- 带来的复杂性：性能分析问题
- 性能参数
 - 平均访存时间 = 命中时间L1 + 失效率L1 × 失效开销L1
失效开销L1 = 命中时间L2 + 失效率L2 × 失效开销L2
 - $AMAT = Hit\ Time_{L1} + Miss\ rate_{L1} \times (Hit\ Time_{L2} + Miss\ rate_{L2} \times Miss\ penalty_{L2})$
 - 对第二级Cache，系统所采用的术语
 - 局部失效率：该级Cache的失效次数 / 到达该级Cache的访存次数
 - 全局失效率：该级Cache的失效次数 / CPU发出的访存总次数

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的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

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 cycles 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} = 100 clock cycles

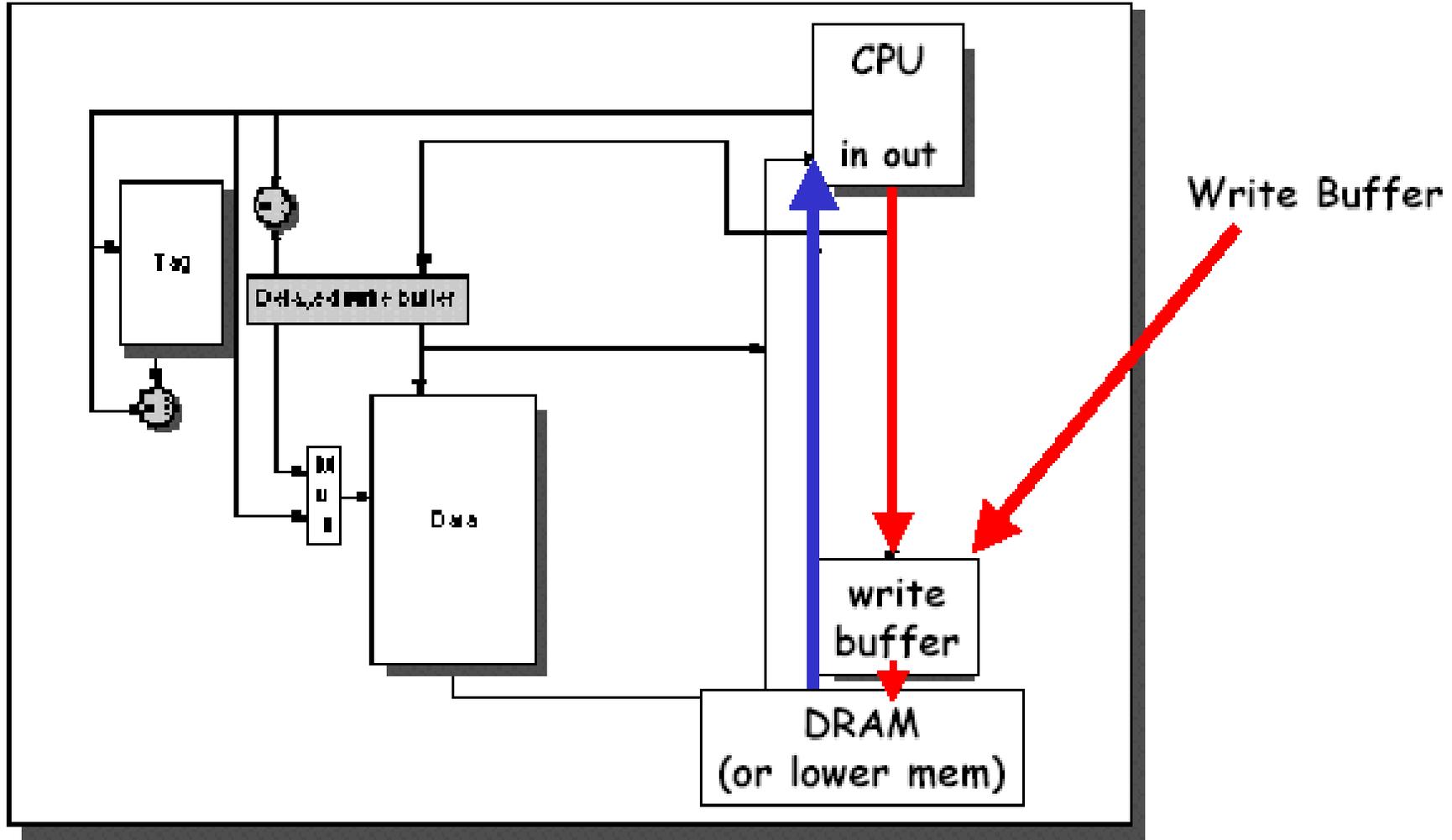
结论：提高相联度，可减少第一级Cache的失效开销

- ❑ 第二级Cache特点：容量大，高相联度，块较大，重点减少失效次数。

让读失效优先于写

- Write Buffer (写缓冲), 特别对写直达法更有效
 - 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
- 原因: 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映射到同一快。则
 - SW R3, 512(R0)
 - LW R1, 1024(R0) 失效
 - LW R2, 512(R0) 失效
- 解决问题的方法
 1. 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。
 2. 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
- 由于读操作为大概率事件, 需要读失效优先, 以提高性能
- 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

让读优先于写图示



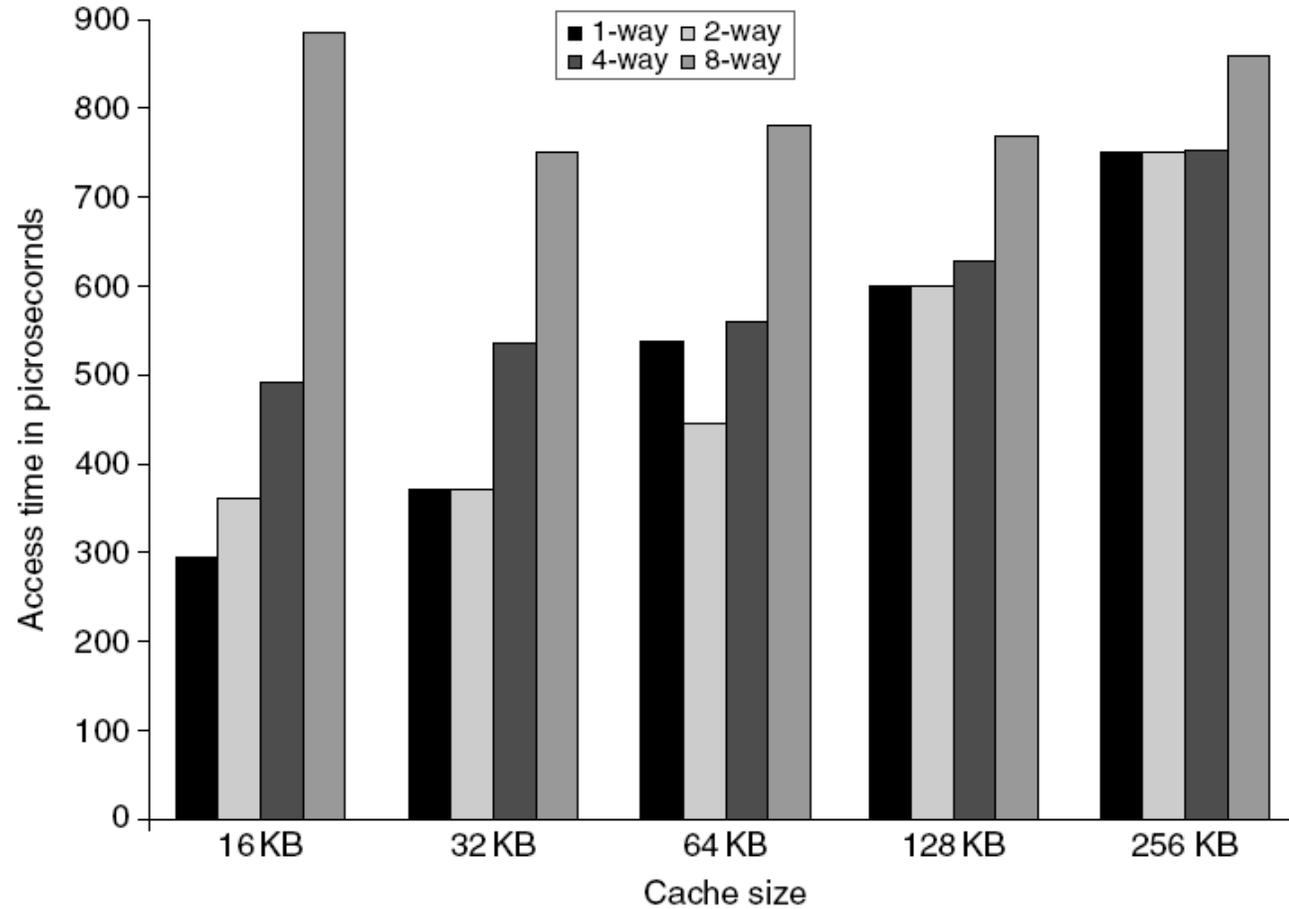
4.4 高级Cache优化方法

- 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
- 减小失效开销
 - 5、多体Cache
 - 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
- 通过并行降低失效代价或失效率
 - 9、硬件预取
 - 10、编译器控制的预取

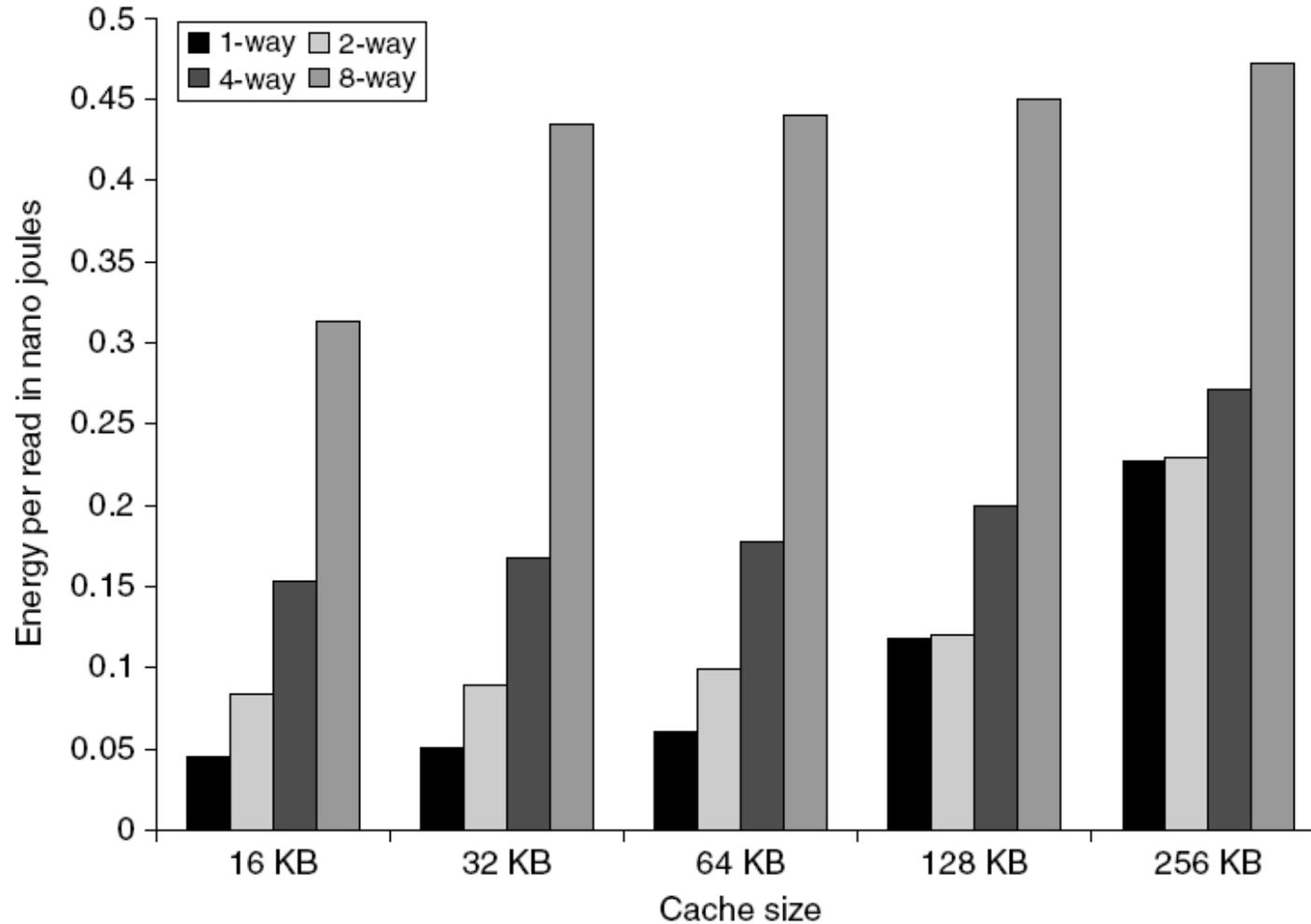
1、Small and simple first level caches

- 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(tag), 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



2、Way Prediction

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