

# 第4章 存储器-层次结构设计

- ❖ 4.1 引言
- ❖ 4.2 Cache 性能
- ❖ 4.3 Cache优化技术
- ❖ 4.4 主存储器技术
- ❖ 4.5 虚拟存储器
- ❖ 4.6 谬误和易犯的错误
- ❖ 4.7 小结

## 4.3 改善cache性能

平均访存时间 = 命中时间 + 缺失率 × 缺失代价

因此，**17种 cache 优化措施**分为**4类**：

### 1.减少缺失率 -- 4

——增加块大小，增大 cache 容量，更高相联度，编译优化

### 2.减少缺失代价 -- 5

——多级 caches，关键字优先，读缺失优于写缺失，  
合并写缓冲，牺牲缓冲

### 3.通过并行减少缺失代价和缺失率--3

——非阻塞 caches，硬件预取，编译预取

### 4.减少cache的命中时间 -- 5

——小和简单的 caches，避免地址转换，流水线 cache 访问，  
路预测，踪迹 caches

## 4.3.1 减少Cache缺失率

### ➤ 减少缺失率——4种技术

增大块容量  
增大cache容量  
更大的相联度  
编译器优化

### ➤ 减少缺失代价

### ➤ 利用并行减少缺失代价或缺失率

### ➤ 减少 cache命中时间

平均访存时间 = 命中时间 + 缺失率 × 缺失代价

# 缺失来自哪里？

- 缺失分类：3 Cs

- **强制缺失**—第1次访问一个块，它一定不在cache中，因此这一块必须被装入cache。这也称为**冷启动缺失**或**首次访问缺失**。

- （甚至在一个无限 Cache中也有强制缺失）

- **容量缺失**—如果 cache容纳不了一个执行程序的所有块，由于一些块被替换出去后又要被访问引起**容量缺失**（除了强制缺失外）。

- (Misses in Fully Associative Size X Cache)

- **冲突缺失**—如果块放置策略是**直接映像**或**组相联**，如果有太多块映射到一组，一块可能被替换后又要被访问从而引起冲突缺失（除了强制和容量缺失外）。也称为 **碰撞缺失**或**干涉缺失**。

- 4th “C”:

- **一致性** - 缺失：由于cache 一致性引起。

## Figure B.8 部分内容

| Cache size (KB) | Degree associative | Total miss rate | Miss rate components (relative percent)<br>(sum = 100% of total miss rate) |      |          |      |          |     |
|-----------------|--------------------|-----------------|--|------|----------|------|----------|-----|
|                 |                    |                 | Compulsory   |      | Capacity |      | Conflict |     |
| 16              | 2-way              | 0.041           | 0.0001   | 0.2% | 0.040    | 98%  | 0.001    | 2%  |
| 32              | 1-way              | 0.042           | 0.0001   | 0.2% | 0.037    | 89%  | 0.005    | 11% |
| 32              | 2-way              | 0.038           | 0.0001   | 0.2% | 0.037    | 99%  | 0.000    | 0%  |
| 64              | 1-way              | 0.037           | 0.0001   | 0.2% | 0.028    | 77%  | 0.008    | 23% |
| 64              | 2-way              | 0.031           | 0.0001   | 0.2% | 0.028    | 91%  | 0.003    | 9%  |
| 128             | 1-way              | 0.021           | 0.0001   | 0.3% | 0.019    | 91%  | 0.002    | 8%  |
| 128             | 2-way              | 0.019           | 0.0001   | 0.3% | 0.019    | 100% | 0.000    | 0%  |
| 256             | 1-way              | 0.013           | 0.0001   | 0.5% | 0.012    | 94%  | 0.001    | 6%  |

①强制性缺失与**Cache**大小无关。

②容量缺失随容量增加而减少，与相联度无关。

③相联度越高，冲突失效就越少。

④ 当**Cache** 的总容量  $N < 128\text{K}$  时，大小为  $N$  的直接映象**Cache**的缺失率约等于大小为  $N/2$  的两路组相联**Cache**的缺失率。

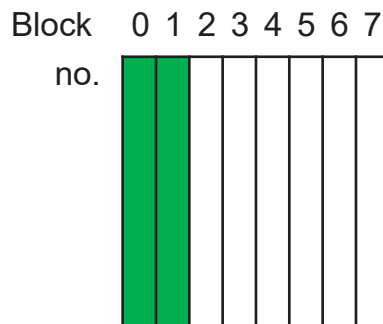
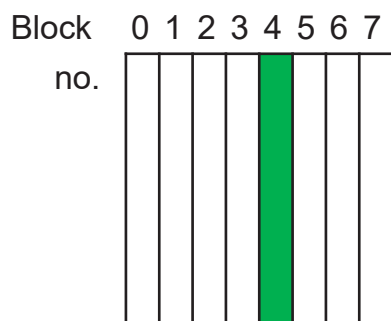
当**Cache** 的总容量  $N \geq 128\text{K}$  时，上述结论不成立。

## 例：直接映像与2路组相联的缺失

读主存的顺序号：① ② ③ ④

假设，读主存的块序列为：**4**、**12**、**4**、**12**

Cache为空，分析后图按以上块序列访问时，直接映像与2路组相联Cache的缺失次数。



# 例：直接映像与2路组相联的缺失

读主存的顺序号：① ② ③ ④

假设，读主存的块序列为：**4、12、4、12**

Cache为空，分析后图按以上块序列访问时，直接映像与2路组相联Cache的缺失次数。

解：访问直接映像Cache的缺失：① ② ③ ④

缺失次数：4

强制缺失次数2：① ②

冲突缺失次数2：③ ④

访问2路组相联像Cache的缺失：① ②

缺失次数：2

强制缺失次数2：① ②

冲突缺失次数0

# 减小 Cache缺失率

- 要减小**cache**缺失率，必须消除**3C's**引起的某些缺失。
- 要减小 **容量缺失**，只有增大**cache**容量。
- 采用什么方式能够减少 **冲突缺失**和 **强制缺失**？



# Cache 组织?

- 假设总的 cache 容量不变
- 如果进行以下变化，会发生什么呢？

1) 改变块的容量；

2) 改变相联度；

3) 改变编译器；

3Cs 中的哪些缺失会被明显影响？

# 第1种缺失率减小技术：增大块容量 (cache容量和相联度固定)

## 方法

- 更大的块减小了强制缺失率，这是利用了空间局部性。

## 绘制的曲线是 U-形的

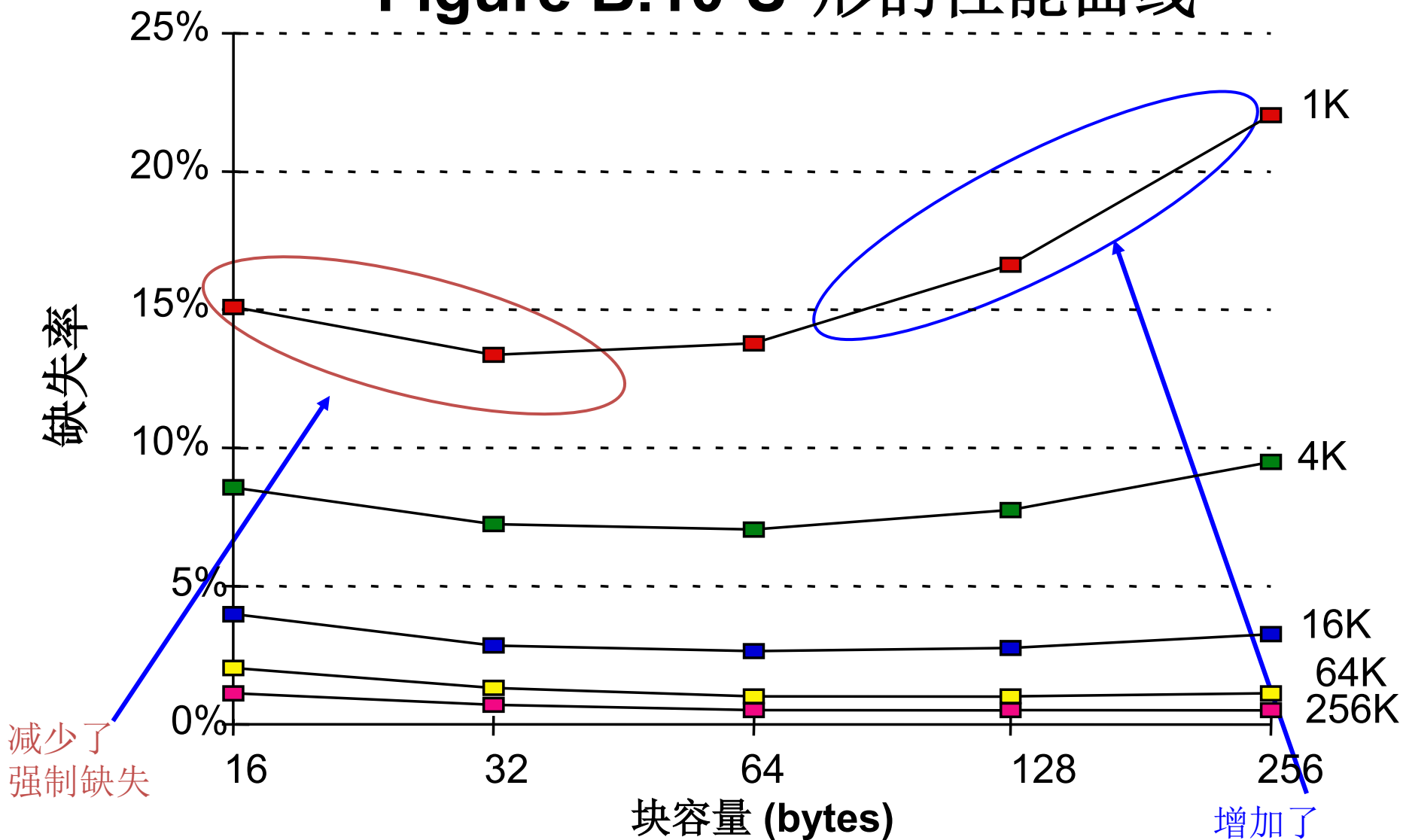
- 会增加缺失代价，这是由于每次缺失需要取更多的数据。
- 几乎会确定增加冲突缺失，这是因为cache中的块数更少。
  - 在小容量cache，甚至可能增加容量缺失。

## 权衡

- 试图既要减小缺失率也要减小缺失代价。
- 块容量的选择取决于下一级存储器的延迟和带宽。

$$\text{平均访存时间} = \text{命中时间} + \text{缺失率} \times \text{缺失代价}$$

# Figure B.10 U-形的性能曲线



块大小的变化会产生什么影响？

# 例：增大块容量

假设：实际缺失率如下表 Figure B.11

| Block size | Cache 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% |

命中时间：1CLK  
(独立于块容量)

主存延迟：80 CLK  
带宽：16 bytes/2 CLK

这相当于主存能够在82 CLK提供16个字节，  
在84 CLK 提供32个字节，以此类推。

对于上表中各cache容量，哪一种块大小具有最小平均访存时间？

# 例：增大块容量

**答案：** 平均访存时间是：

平均访存时间 = 命中时间 + 缺失率 × 缺失代价

16-byte/块，4KB cache

平均访存时间 =  $1 + 8.57\% \times 82 = 8.027$  CLK

256-byte/块，256KB cache

平均访存时间 =  $1 + 0.49\% \times (80 + 2 \times 256/16)$

=  $1 + 0.49\% \times 112 = 1.549$  CLK

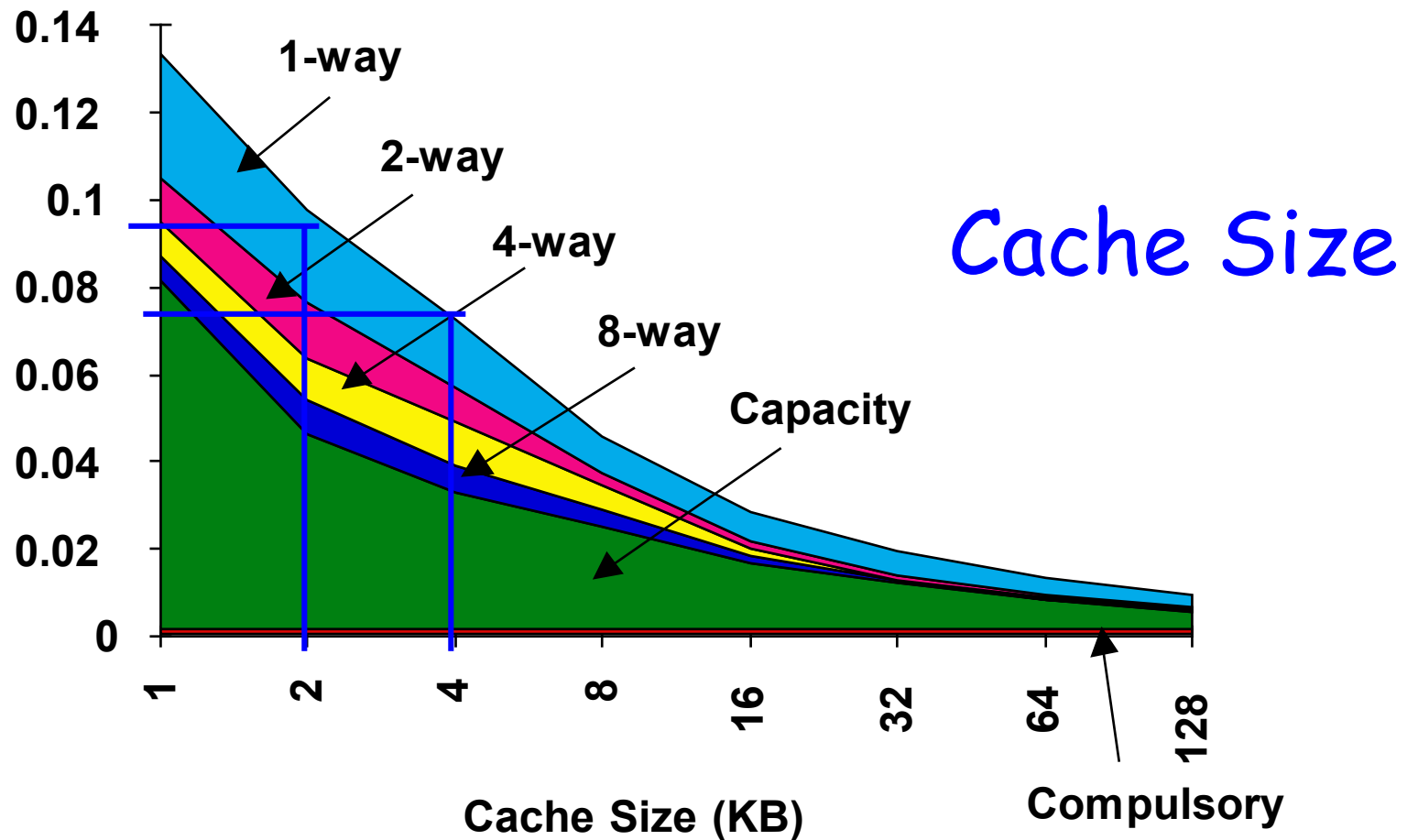
所有计算出的平均访存时间如下表

Figure B.11 部分内容

| Block size | Cache size |       |
|------------|------------|-------|
|            | 4K         | 256K  |
| 16         | 8.57%      | 1.09% |
| 256        | 9.51%      | 0.49% |

| 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 |

## 第2种缺失率减小技术：增大 Caches容量



- 老的经验法则：2 x size => 减小25% 的缺失率
- 增大cache容量减小了什么？容量缺失和冲突缺失
- 缺点：更长的命中时间；更高的价格

# 第3种缺失率减小技术：更高的相联度

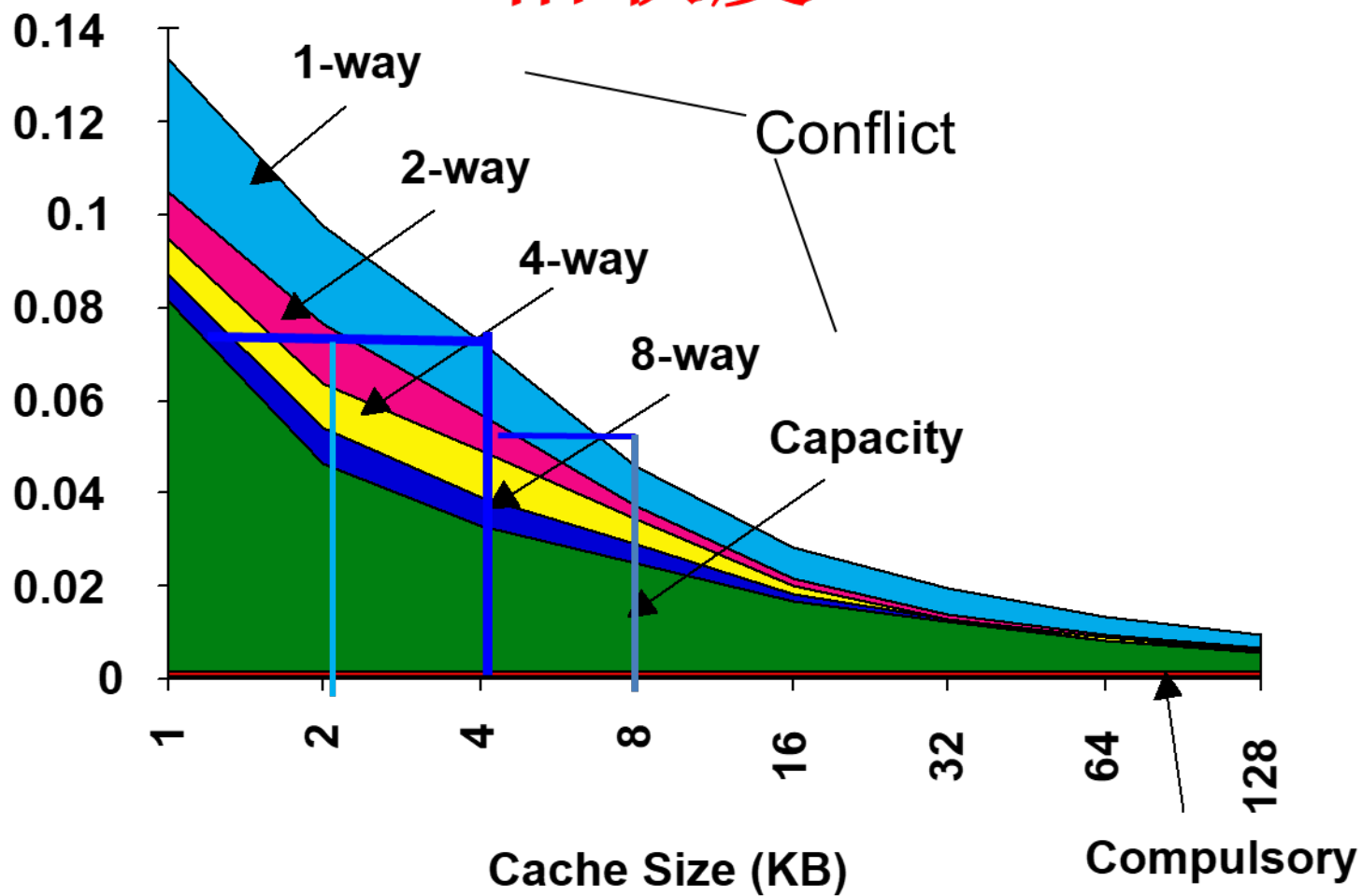
## 方法

- 对于低相联度（尤其是直接映像）cache，冲突缺失是一个问题。
- 更高相联度减少了冲突缺失，从而改善了缺失率。

## Cache经验法则

- 2:1 经验法则 一个容量小于128KB为  $N$  直接映像cache与容量为  $N/2$  的2-路组相联cache具有相同的缺失率。
- 从实用角度，8-路组相联cache与相同容量全相联cache有效减少冲突缺失的效果几乎相当。

# 相联度





# 相联度与时钟周期时间

- ❖ 必须意识到：执行时间是唯一最终的评价指标！
- ❖ 时钟周期时间与命中时间紧密相关。
- ❖ 相联度增加，时钟周期时间会增加吗？
  - Hill [1988] 提出，2-路相对1-路的命中时间：  
外部cache +10%  
内部cache + 2%

## 例： 更高相联度

假设提高相联度将会延长时钟周期时间， 如下所示：

$$\text{时钟周期时间}_{2\text{-路}} = 1.36 \times \text{时钟周期时间}_{1\text{-路}}$$

$$\text{时钟周期时间}_{4\text{-路}} = 1.44 \times \text{时钟周期时间}_{1\text{-路}}$$

$$\text{时钟周期时间}_{8\text{-路}} = 1.52 \times \text{时钟周期时间}_{1\text{-路}}$$

命中时间<sub>1路</sub>： 1 CLK

缺失代价<sub>1路</sub>： 25 CLK

**L2 cache** 不产生缺失， 且不要求缺失代价是时钟周期整倍数。  
使用图B.8中的缺失率， 哪种容量的**cache** 才能使以下三个表达式成立？

$$\text{平均访存时间}_{8\text{-路}} < \text{平均访存时间}_{4\text{-路}}$$

$$\text{平均访存时间}_{4\text{-路}} < \text{平均访存时间}_{2\text{-路}}$$

$$\text{平均访存时间}_{2\text{-路}} < \text{平均访存时间}_{1\text{-路}}$$

## 例： 更高相联度 Figure B.8 部分内容

答案：

平均访存时间<sub>8-路</sub>

$$\begin{aligned} &= \text{命中时间}_{8\text{-路}} + \text{缺失率}_{8\text{-路}} \times \text{缺失代价}_{8\text{-路}} \\ &= \mathbf{1.52} + \text{缺失率}_{8\text{-路}} \times 25 \end{aligned}$$

$$\text{平均访存时间}_{4\text{-路}} = \mathbf{1.44} + \text{缺失率}_{4\text{-路}} \times 25$$

$$\text{平均访存时间}_{2\text{-路}} = \mathbf{1.36} + \text{缺失率}_{2\text{-路}} \times 25$$

$$\text{平均访存时间}_{1\text{-路}} = \mathbf{1.00} + \text{缺失率}_{1\text{-路}} \times 25$$

缺失代价在以上式子中是相同的，假定为25个时钟周期。

4KB直接映像cache的平均访存时间是：

$$\text{平均访存时间}_{1\text{-路}} = 1.00 + 0.098 \times 25 = 3.45$$

512KB, 8-路组相联是：

$$\text{平均访存时间}_{8\text{-路}} = 1.52 + 0.006 \times 25 = 1.67$$

| Cache size (KB) | Degree associative | Total miss rate |
|-----------------|--------------------|-----------------|
| 4               | 1-way              | 0.098           |
| 512             | 8-way              | 0.006           |

Figure B.13

| Associativity   |       |              |              |              |
|-----------------|-------|--------------|--------------|--------------|
| Cache size (KB) | 1-way | 2-way        | 4-way        | 8-way        |
| 4               | 3.45  | 3.26         | 3.215        | <b>3.295</b> |
| 8               | 2.7   | 2.585        | 2.54         | <b>2.62</b>  |
| 16              | 2.225 | <b>2.385</b> | <b>2.465</b> | <b>2.545</b> |
| 32              | 2.05  | <b>2.31</b>  | <b>2.365</b> | <b>2.445</b> |
| 64              | 1.925 | <b>2.135</b> | <b>2.19</b>  | <b>2.245</b> |
| 128             | 1.525 | <b>1.835</b> | <b>1.915</b> | <b>1.995</b> |
| 256             | 1.325 | <b>1.66</b>  | <b>1.74</b>  | <b>1.82</b>  |
| 512             | 1.20  | <b>1.535</b> | <b>1.59</b>  | <b>1.67</b>  |

注:5e,6e的部分  
数据不准确, 已  
更正

**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.**

以图B-8中的缺失率作为本例中参数得出的存储器平均访问时间。**粗体**表示这一时间高于左侧的数值, 即较高的相联度延长了存储器平均访问时间。

该表显示, 对于不大于8KiB、不超过四路相联度的缓存, 本例中的公式成立。从16KiB开始, 较高相联度的较长命中时间超过了因为缺失降低所节省的时间。

注意, 在本例中, 我们没有考虑较慢时钟频率对程序其余部分的影响, 因此低估了直接映射缓存的优势。

# 第4种缺失率减小技术：编译器优化

## 方法

无需修改硬件、利用编译器对指令程序、数据重排序就可以减小缺失率。

### ❖ 指令

- 如，编译器预测转移发生，可以将转移目标基本块与转移指令后的基本块互换

### ❖ 数据

- **合并数组**：将二个连续数组合并为一个数组，改善空间和时间局部性
- **循环交换**：改变嵌套循环顺序以便按照数据的存储顺序来访问数据
- **循环融合**：将有相同循环和一些变量重叠的2个独立循环组合成一个循环
- **分块**：利用重复访问数据“块”（相对于整列或整行访问）改善空间局部性。

## ① 合并数组 (merged arrays)

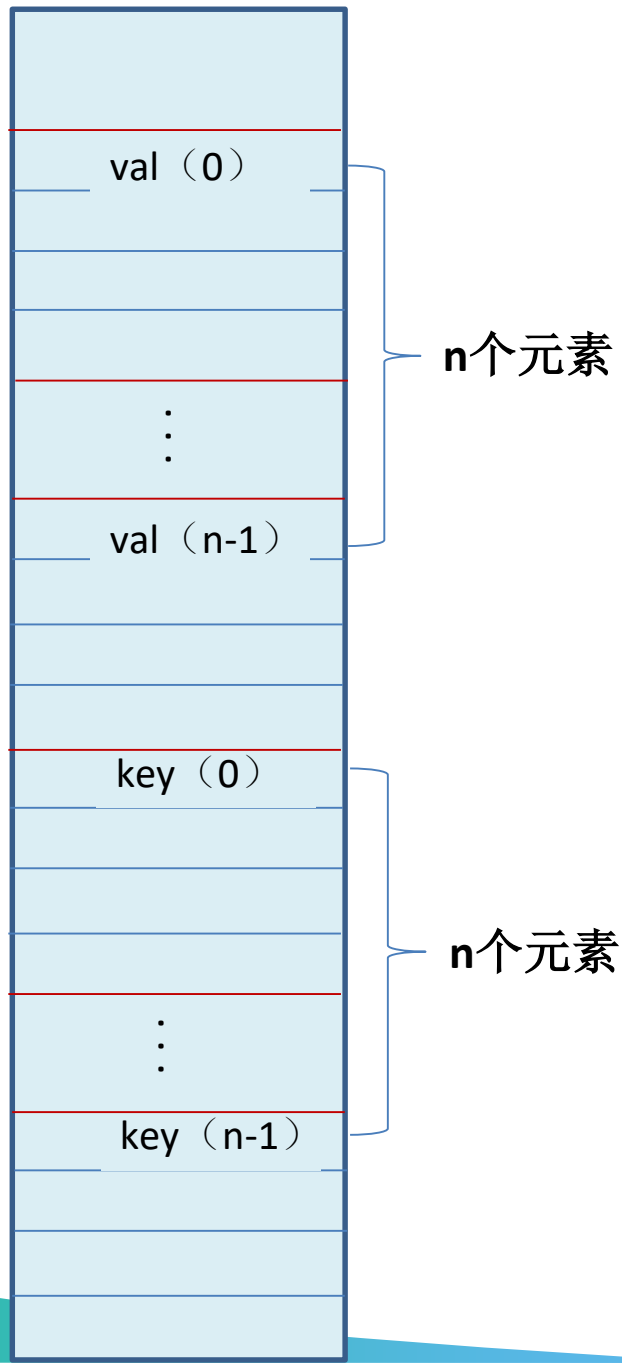
```
/* Before: 2 sequential arrays */  
int val[SIZE];  
int key[SIZE];
```

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

减少 val 与 key的Cache缺失;

改善了空间局部性

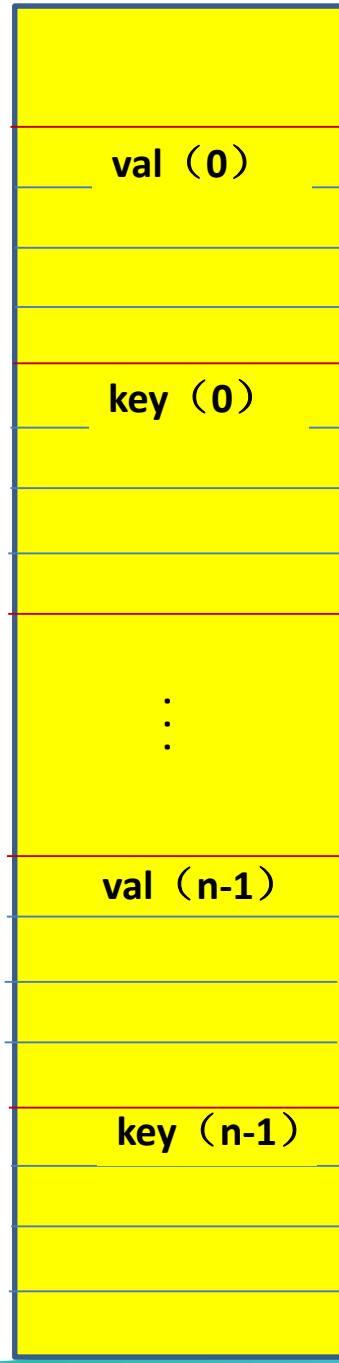
主存



合并前

合并后

主存



例:  $x[i] = val[i] + key[i]$

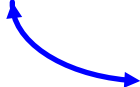
对应的指令:

```
lw r1, 0(r2)
lw r3, 4(r2)
add r4, r1, r3
sw r4, 100(r2)
```

## ② 循环交换 (loop interchange)

交换循环的执行顺序，可以减少缺失次数，这是因为改善了空间局部性。

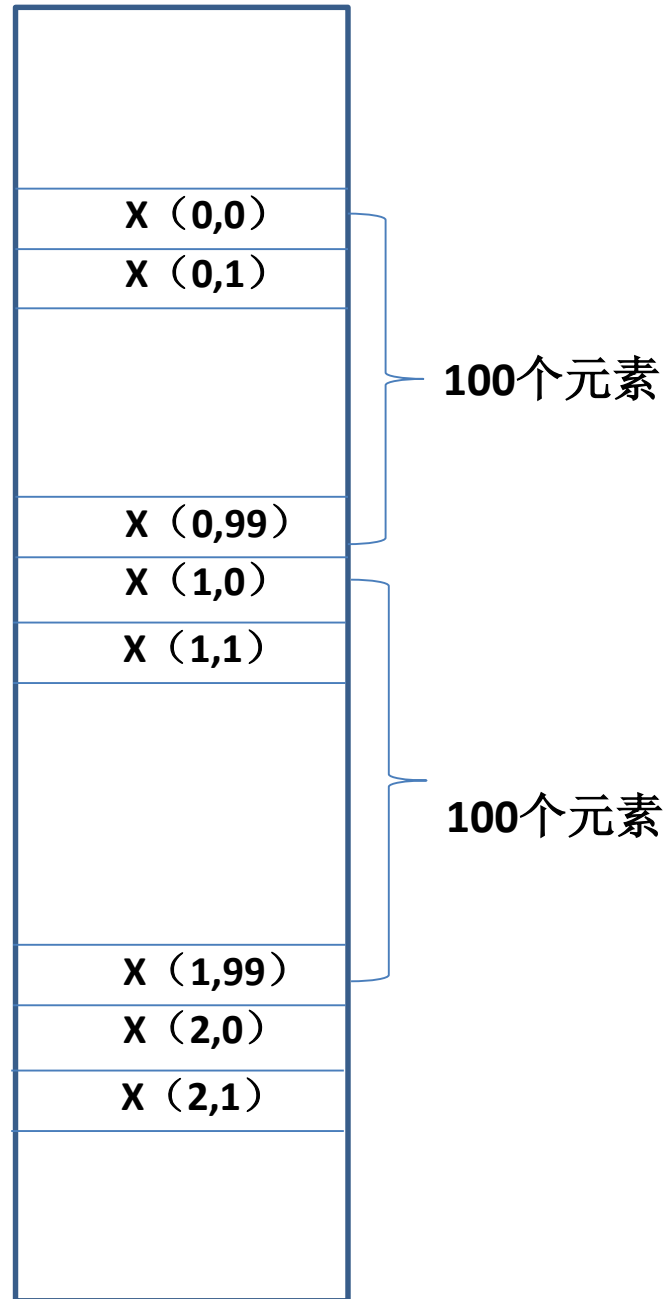
```
/* Before */  
for (k = 0; k < 100; k = k+1)  
    for (j = 0; j < 100; j = j+1)  
        for (i = 0; i < 5000; i = i+1)  
            x[i][j] = 2 * x[i][j];  
  
/* After */  
for (k = 0; k < 100; k = k+1)  
    for (i = 0; i < 5000; i = i+1)  
        for (j = 0; j < 100; j = j+1)  
         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



用顺序访问替代了以100个字为步长的跳跃访问。



主存



### ③ 循环融合 (loop fusion)

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

对于 **a** 和 **c** :

- 融合前, 每次计算访问**2**次缺失;
- 融合后, 平均每次计算访问**1**次缺失。利用了时间局部性。

## ④ 未优化的矩阵乘法

/\* 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;
        };
```

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
|   | j |   |   |   |   |   |
| x | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 |   |   |   |   |   |   |
| 1 |   |   |   |   |   |   |
| 2 |   |   |   |   |   |   |
| 3 |   |   |   |   |   |   |
| 4 |   |   |   |   |   |   |
| 5 |   |   |   |   |   |   |

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
|   | k |   |   |   |   |   |
| y | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 |   |   |   |   |   |   |
| 1 |   |   |   |   |   |   |
| 2 |   |   |   |   |   |   |
| 3 |   |   |   |   |   |   |
| 4 |   |   |   |   |   |   |
| 5 |   |   |   |   |   |   |

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
|   | j |   |   |   |   |   |
| z | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 |   |   |   |   |   |   |
| 1 |   |   |   |   |   |   |
| 2 |   |   |   |   |   |   |
| 3 |   |   |   |   |   |   |
| 4 |   |   |   |   |   |   |
| 5 |   |   |   |   |   |   |

x[i][j]

y[i][k]

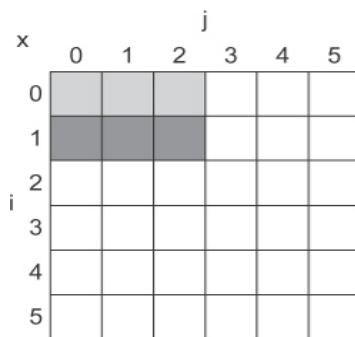
z[k][j]

- 两个内部循环：
  - 写x一行的N个元素
  - 重复读 Y[] 一行的N个元素
  - 读Z[] Nx1 个元素
- 容量缺失依赖于N与cache的容量：
  - $2N^3 + N^2$  (假定不考虑冲突缺失)
- 思路：采用分块矩阵进行计算

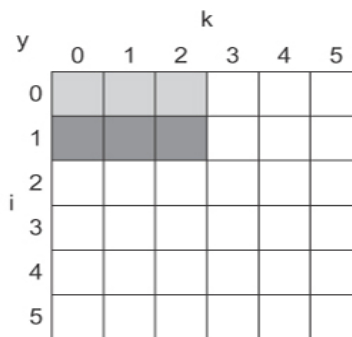
$((N+N)+1)N = 2N^3 + N^2$   
 $N^3$ 级别的操作访问次数

# 优化分块矩阵乘法 (blocking)

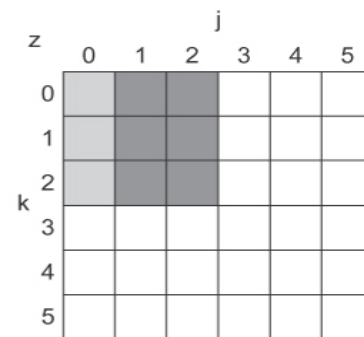
- 矩阵乘法是先利用子矩阵乘法来完成的



BN



BN



B×B

B 称为分块因子

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

Y 受益于空间局部性

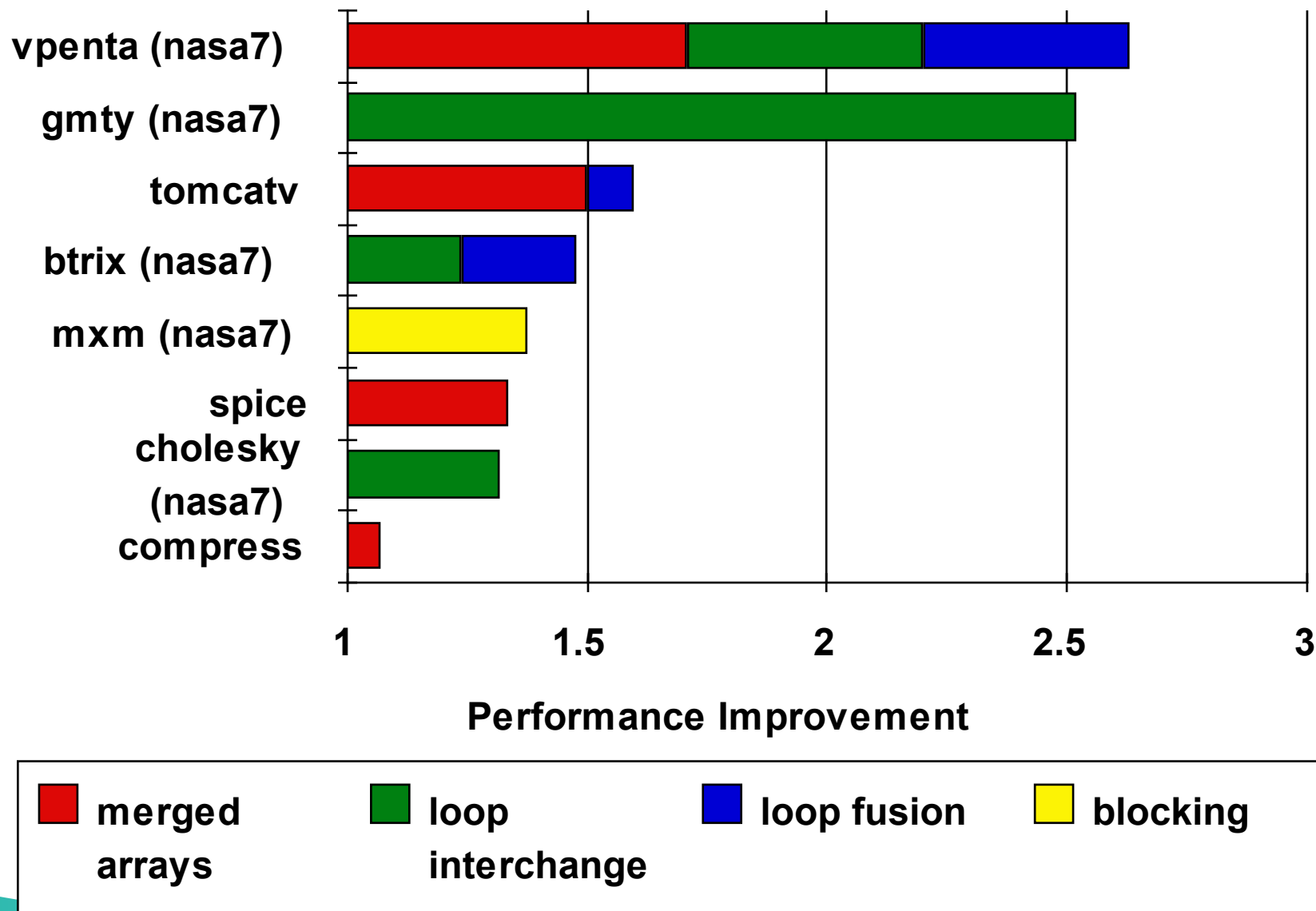
Z 受益于时间局部性

容量缺失从  $2N^3 + N^2$  到  $2N^3/B + N^2$

$$((NB + NB) + B^2) \times (N/B)^2 = 2N^3/B + N^2$$

$N^3$  级别的操作访问次数

# 编译器优化减少 Cache 缺失总结



# 小结：减小缺失率

$$CPUtime = IC \times \left( CPI_{Execution} + \frac{Memory\ accesses}{Instruction} \times \text{Miss rate} \times Miss\ penalty \right) \times Clock\ cycle\ time$$

平均访存时间 = 命中时间 + 缺失率 × 缺失代价

- **3 Cs: 强制缺失, 容量缺失, 冲突缺失**
  1. 增大块容量减少强制缺失
  2. 增大 cache 容量减少容量缺失和冲突缺失
  3. 更高相联度减少冲突缺失
  4. 编译器优化减少缺失

## 4.3.2 减少 Cache 缺失代价

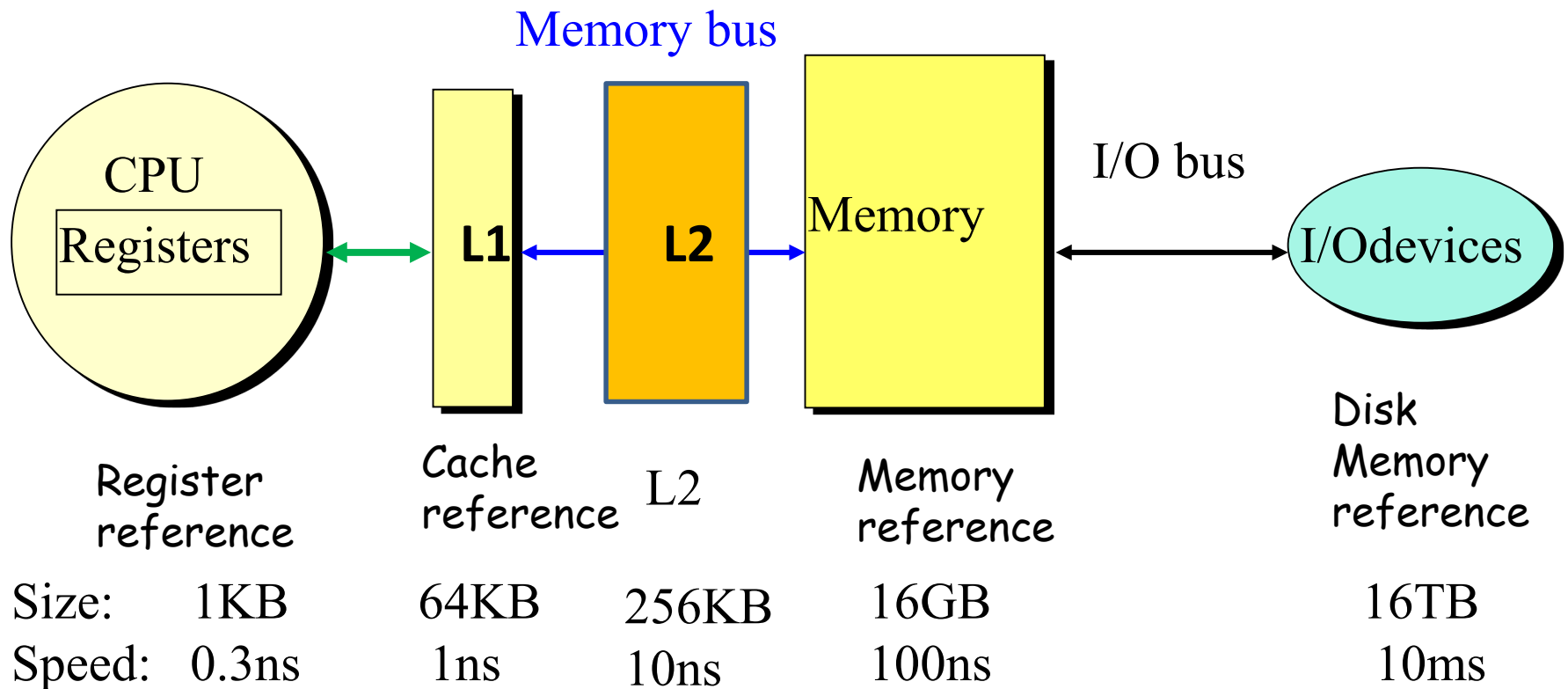
- 减少缺失率——4种技术
- 减少缺失代价——5种技术
- 利用并行减少缺失代价或缺失率
- 减少 cache命中时间

多级cache  
关键字优先  
读缺失优先于写  
合并写缓冲  
牺牲缓存

平均访存时间 = 命中时间 + 缺失率 x 缺失代价

# 第1种缺失代价减少技术：多级 Caches

处理器-存储器之间的性能差距





# 第1种缺失代价减少技术：多级 Caches

## 方法

- 在一个 **小而快的一级cache** 与主存之间增加一个 **二级 cache**
  - 帮助满足 cache 快而大
- **一级 cache :**
  - 一级 cache 是快得足够匹配CPU的时钟周期时间，而且小，以减少 命中时间。
- **二级 cache :**
  - 更大的二级cache大得足以捕捉很多本该到主存的访问，从而有效地减少 缺失代价。

## 二级cache的参数

- 参照单级cache性能，计算二级cache的性能
- L2 公式

$$\text{平均访存时间} = \text{命中时间}_{L1} + \text{缺失率}_{L1} \times \text{缺失代价}_{L1}$$

$$\text{缺失代价}_{L1} = \text{命中时间}_{L2} + \text{缺失率}_{L2} \times \text{缺失代价}_{L2}$$

$$\text{平均访存时间} = \text{命中时间}_{L1} + \text{缺失率}_{L1} \times (\text{命中时间}_{L2} + \text{缺失率}_{L2} \times \text{缺失代价}_{L2})$$

- 定义：

- 局部缺失率 ---- 这一级 cache 的缺失次数除以访问这级cache的总次数（缺失率<sub>L2</sub>）
- 全局缺失率 ---- 这级 cache的缺失次数除以CPU产生的访存总次数
- L1和L2的全局缺失率？

## 二级cache的参数

- 定义：

- **局部缺失率** — 这一级 cache 的缺失次数除以访问这级cache的总次数（缺失率<sub>L2</sub>）
- **全局缺失率** — 这级 cache的缺失次数除以CPU产生的访存总次数

L1和L2的全局缺失率？

按照定义，一级cache的全局缺失率与局部缺失率是相同的为缺失率<sub>L1</sub>；但是第二级 cache 的全局缺失率却是：

$$\begin{aligned}\text{缺失率}_{\text{全局}L2} &= \frac{\text{缺失次数}_{L2}}{M} = \text{缺失率}_{L1} \times \frac{\text{缺失次数}_{L2}}{M \times \text{缺失率}_{L1}} \\ &= \text{缺失率}_{L1} \times \text{缺失率}_{L2}\end{aligned}$$

# 例：二级cache的缺失率计算

假设： 1000次访存中

L1 cache: 40次缺失

L2 cache: 20 次缺失

求L1的缺失率，L2的局部与全局缺失率？

答案： 计算局部和全局缺失率

$$\text{缺失率}_{L1} = \frac{\text{缺失次数}_{L1}}{\text{访存总次数}} = \frac{40}{1000} = 4\%$$

$$\text{缺失率}_{L2} = \frac{\text{缺失次数}_{L2}}{\text{访存次数}_{L2}} = \frac{20}{40} = 50\%$$

$$\text{缺失率}_{G2} = \frac{\text{缺失次数}_{L2}}{\text{访存总次数}} = \frac{20}{1000} = 2\%$$

\* 二级cache的局部缺失率大， 是因为一级cache存储的是最易命中的数据。

\* 全局缺失率： 给出了CPU访问不同级存储器缺失所占的比例。

# 每条指令的存储器停顿

每条指令的存储器停顿需要考虑二级cache的影响:

$$\text{平均访存时间} = \text{命中时间}_{L1} + \text{缺失率}_{L1} \times \text{缺失代价}_{L1}$$


$$\text{平均访存停顿时间} = \text{缺失率}_{L1} \times \text{缺失代价}_{L1}$$

$$\text{每条指令的存储器停顿} = \text{每条指令的访存次数} \times \text{平均访存停顿时间}$$

$$= \text{每条指令的访存次数} \times \text{缺失率}_{L1} \times \text{缺失代价}_{L1}$$

$$= \text{每条指令的访存次数} \times \text{缺失率}_{L1} \times (\text{命中时间}_{L2} + \text{缺失率}_{L2} \times \text{缺失代价}_{L2})$$

$$= \text{每条指令的访存次数} \times \text{缺失率}_{L1} \times \text{命中时间}_{L2} +$$

$$\text{每条指令的访存次数} \times \text{缺失率}_{L1} \times \text{缺失率}_{L2} \times \text{缺失代价}_{L2}$$

$$= \text{每条指令的缺失次数}_{L1} \times \text{命中时间}_{L2} +$$

$$\text{每条指令的缺失次数}_{L2} \times \text{缺失代价}_{L2}$$

# 例：二级cache的平均访存时间

假设：1000次访存

L1 cache: 40 次缺失

命中时间—— 1 clock cycles

每条指令访存次数—— 1.5

L2 cache:: 20 次缺失

命中时间——10 clock cycles

缺失代价—— 200 clock cycles

求平均访存时间、平均每条指令存储器停顿周期数

答案： 计算局部和全局缺失率

$$\text{缺失率}_{L1} = \frac{\text{缺失次数}_{L1}}{\text{访存总次数}} = \frac{40}{1000} = 4\% \quad \text{缺失率}_{L2} = \frac{\text{缺失次数}_{L2}}{\text{访存次数}_{L2}} = \frac{20}{40} = 50\%$$

$$\text{缺失率}_{G2} = \frac{\text{缺失次数}_{L2}}{\text{访存总次数}} = \frac{20}{1000} = 2\%$$

$$\begin{aligned} \text{平均访存时间} &= \text{命中时间}_{L1} + \text{缺失率}_{L1} \times (\text{命中时间}_{L2} + \text{缺失率}_{L2} \times \text{缺失代价}_{L2}) \\ &= 1 + 4\% \times (10 + 50\% \times 200) = 1 + 4\% \times 110 = 5.4 \text{ clock cycle} \end{aligned}$$

只有一级Cache的平均访存时间是：  $1 + 4\% \times 200 = 9 \text{ clock cycle}$

假设：1000次访存

L1 cache: 40 次缺失

命中时间—— 1 clock cycles

每条指令访存次数—— 1.5

L2 cache: 20 次缺失

命中时间—— 10 clock cycles

缺失代价—— 200 clock cycles

例：二级cache的平均每条指令的存储器停顿

每 1000 指令的缺失次数：

$$L1（全局）： 1.5 \times 4\% \times 1000 = 60$$

$$L2（全局）： 1.5 \times 2\% \times 1000 = 30$$

平均每条指令的存储器停顿

= 每条指令缺失次数<sub>L1(global)</sub> × 命中时间<sub>L2</sub> + 每条指令缺失次数<sub>L2(global)</sub>

$$\times \text{缺失代价}_{L2} = (60/1000) \times 10 + (30/1000) \times 200$$

$$= 0.060 \times 10 + 0.030 \times 200 = 6.6 \text{ clock cycles}$$

如果从平均访存时间减去 L1 命中时间，然后乘以 平均每条指令访存次数，将获得同样的结果：

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

只有一级Cache的平均每条指令的存储器停顿是：12 clock cycle

# 例：L2相联度对缺失代价影响

假定：直接映像 命中时间 $t_{L2} = 10$  clock cycles

2路组相联 命中时间 $t_{L2} = 10.1$  clock cycles

直接映像 局部缺失率 $r_{L2} = 25\%$

2路组相联 局部缺失率 $r_{L2} = 20\%$

缺失代价 $C_{L2} = 200$  clock cycles

求二级Cache相联度对一级cache缺失代价的影响？



# 例：L2相联度对缺失代价影响

假定：直接映像 命中时间 $t_{L2} = 10$  clock cycles

2路组相联 命中时间 $t_{L2} = 10.1$  clock cycles

直接映像 局部缺失率 $r_{L2} = 25\%$

2路组相联 局部缺失率 $r_{L2} = 20\%$

缺失代价 $C_{L2} = 200$  clock cycles

求二级Cache相联度对一级cache缺失代价的影响？

$$\text{缺失代价}_{L1} = \text{命中时间}_{L2} + \text{缺失率}_{L2} \times \text{缺失代价}_{L2}$$

答案：对于直接映像 L2 cache, L1 cache 缺失代价是：

$$\text{L1缺失代价}_{1\text{路}L2} = 10 + 25\% \times 200 = 60.0 \text{ clock cycles}$$

增加相联度的代价是命中时间增加 0.1 clock cycles，对于2路组相联 L2，L1 cache 缺失代价是：

$$\text{L1缺失代价}_{2\text{路}L2} = 10.1 + 20\% \times 200 = 50.1 \text{ clock cycles}$$

答案： 对于直接映像 L2 cache, L1 cache 缺失代价是：

$$\text{L1缺失代价}_{1\text{路L2}} = 10 + 25\% \times 200 = 60.0 \text{ clock cycles}$$

增加相联度的代价是命中时间增加 0.1 clock cycles, 对于2路组相联 L2, L1 cache 缺失代价是：

$$\text{L1缺失代价}_{2\text{路L2}} = 10.1 + 20\% \times 200 = 50.1 \text{ clock cycles}$$

事实上, L2 cache 几乎总是与L1 cache 和 CPU同步的。因此, L2命中时间一定是整数时钟周期数。

- L2命中时间或削整到 10 cycles或凑足到 11 cycles。这样改动后对L2是2路组相联的L1 cache 的缺失代价是：

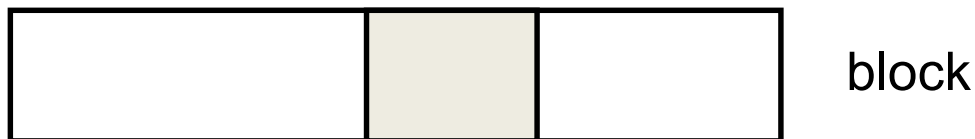
$$\text{L1缺失代价}_{2\text{路L2}} = 11 + 20\% \times 200 = 51.0 \text{ clock cycles}$$

结论：通过增加二级Cache的相联度以减少二级cache的缺失率, 来减少一级cache的缺失代价。

## 第2种缺失代价减少技术：关键字优先和提前重启动

方法：CPU只需要块中的一个字

- 不要等到取到整个块后才重新启动CPU
  - 关键字优先—首先从存储器请求缺失的字并尽可能快地送到CPU；让CPU继续执行同时填放块中的其余字。也称为 *wrapped fetch* 和 *requested word first*
  - 提前重启动—以正常顺序取块，只要块中所请求的字到达，就送到CPU，让CPU继续执行。
- 通常用在大块中
- 空间局部性 => 趋向于将需要下一个连续的字，应该说提前重启动是有利的



# 例：关键字优先

假设： cache 块=64-byte

L2: 花费 11 CLK 得到关键 8 bytes, (AMD Athlon)

然后 以8byte/2 CLK速度取块的其余部分。

对于块其余部分将没有任何其他访问。

计算关键字优先的平均缺失代价。

比较有关关键字优先和没有关键字优先所花费的时间。

# 例：关键字优先

假设： cache 块=64-byte

L2： 花费 11 CLK 得到关键 8 bytes, (AMD Athlon)

然后 以8byte/2 CLK速度取块的其余部分。

对于块其余部分将没有任何其他访问。

计算关键字优先的平均缺失代价。

比较有关关键字优先和没有关键字优先所花费的时间。

**答案： 假设读取8字节**

关键字优先的平均缺失代价是：CPU读关键的8字节, **11clock cycles**

没有关键字优先：CPU连续读一个整cache块

$$11 + (8-1) \times 2 = 25 \text{ clock cycle}$$

会花费 25 clock cycles 取到整块。

# 例

- ❖ 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.
- ❖ 假定主存储器的访问时间为36ns，存储器系统的持续传输速率为16 GiB/s。设块大小为64字节。如果在给定请求流的情况下能够保持峰值带宽，而且访问永远不会冲突，则需要支持的最大未处理缺失数目为多少？如果一次访问与前4次访问发生冲突的概率为50%，并且每次访问都要等待更早的访问完成，请估计最大未完成访问数目。为简单起见，忽略缺失之间的时间。

# 例

- ❖ 假定主存储器的访问时间为36ns，存储器系统的持续传输速率为16 GiB/s。设块大小为64字节。如果在给定请求流的情况下能够保持峰值带宽，而且访问永远不会冲突，则需要支持的最大未处理缺失数目为多少？如果一次访问与前4次访问发生冲突的概率为50%，并且每次访问都要等待更早的访问完成，请估计最大未完成访问数目。为简单起见，忽略缺失之间的时间。
- ❖ 解：在第一种情况下，假定我们可以保持峰值带宽，存储器系统支持每秒  $(16 \times 10^9) / 64 = 2.5$  亿次访问。由于每次访问耗时36ns，因此可以支持  $2.5 \times 10^8 \times 36 \times 10^{-9} = 9$  次访问。如果发生冲突的概率大于0，我们就会面临更多的未完成访问，因为如果访问存在冲突，就无法正常工作；存储器系统需要更多的独立访问！为了简单估计这一数目，假定有一半存储器访问不需要发送到存储器。这就意味着必须支持两倍的未完成访问，即 18 次。

## 第3种缺失代价减少技术： 读缺失的优先级高于写—潜在问题

### 注意

- 如果系统有写缓冲，写操作能够延迟到读后。
- 但是，系统**必须**小心检查写缓冲中是否有**读缺失**要读的值。

• **写直达** 当读缺失产生读主存时，有可能要读的数据在写缓冲中，还没有写入主存：

数据在**写缓冲**中=> **RAW** 冒险



## 例：读缺失优先级高于写

假设：缓存容量128块，直接映像，每块1个字。有以下代码序列：

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)

直接映像cache能够映射地址512 和 1024到同一块，写直达有4个字的写缓冲。

R2 的值总是与R3的值相同吗？

## 例：读缺失优先级高于写

假设：缓存容量128块，直接映像，每块1个字。有以下代码序列：

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)

直接映像cache能够映射地址512 和 1024到同一块，**写直达**有4个字的写缓冲。

**R2 的值总是与R3的值相同吗？**

**答案：**在主存中有一个read-after-write数据冒险。

通过跟踪一次 **cache** 访问来分析这个冒险。

在**store**操作后，**R3**的值放入了**写缓冲（写命中）**。

其后 **load** 使用相同的**cache**索引，不命中，会引起一次读缺失（优先级高于写）。

第2条**load** 指令想把位于 **512**处的值写入寄存器**R2**，这也导致一次读缺失。

如果**写缓冲没有完成**将**512**处的值写入主存，对主存**512**处的读将导致将主存中老的、错误的值写入**cache**块，然后送入**R2**。**R3** 的值将不等于**R2**的值！

**这种情况下**，读缺失的地址与写缓冲中的地址有**冲突**，就不能先读主存。

## 第3种缺失代价减少技术： 读缺失的优先级高于写—潜在问题

### 解决方法

- 如果简单等待**写缓冲空**，可能增加读缺失代价（老的 MIPS 1000 会增加 50% ）
- 在**读之前**检查写缓冲数据地址：如地址不相同，让**读缺失**优先。

### •写回

- 读缺失替换脏块
- 通常操作：写脏块到主存，然后进行读操作。

如果有**写缓冲**：复制脏块到写缓冲，然后**先读**，再写。

有读缺失时，也要检查写缓冲，看地址是否相同，没有则可以读缺失优先。

# 第4种缺失代价减少技术：合并写缓冲

## 方法

- 用写多个字合并为写一个字，可以改善写缓冲的效率。
- **写直达**，如果写缓冲包含其他修改的块，检查新数据的地址是否与写缓冲中的其他有效项地址匹配。如果匹配，**新数据合并到那一项中**。
- 合并写优化技术有时可以减少**写缓冲满**时导致的停顿。

| 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 |  |

| 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 |          |

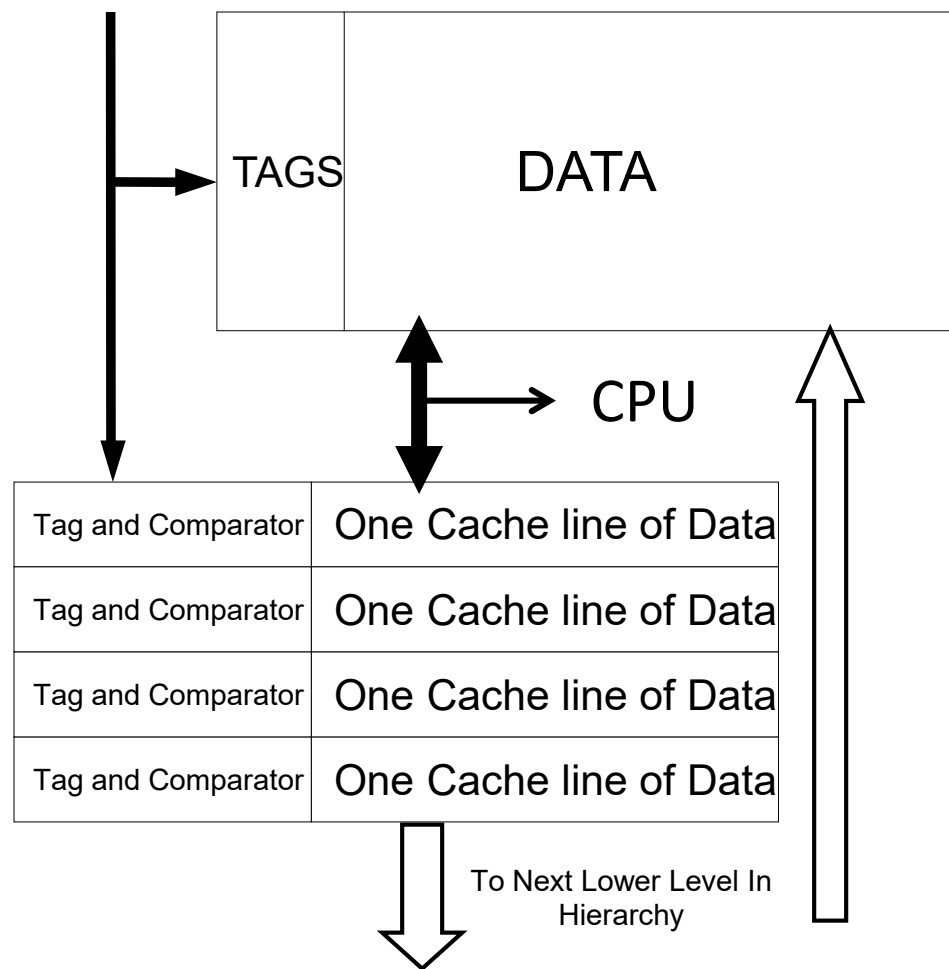
# 第5种缺失代价减少技术：牺牲缓存

## 方法

- 牺牲缓存是一个 **小的全相联 cache**，它存放 几个最近被替换出的块。该技术 **并非必须**。
- 在发生 **缺失** 要访问下一级存储器以前，**先检查牺牲 cache**：
  - 看是否有缺失的数据
  - 如果有，交换牺牲块与 cache 块。
  - AMD Athlon 有一个8项的 victim caches。

# 怎样组织高效牺牲Cache

- 怎样组织牺牲Cache既加快直接映像cache的命中时间，又减少冲突缺失？
- 增加缓冲区大小及比较逻辑
- Jouppi [1990]: 一个4-项victim cache能使一个4KB直接映象数据Cache的冲突缺失减少20%~90%。
- 用于 Alpha、HP 机器



# 小结：减少缺失代价

$$CPUtime = IC \times \left( CPI_{Execution} + \frac{Memory\ accesses}{Instruction} \times \text{Miss rate} \times \text{Miss penalty} \right) \times Clock\ cycle\ time$$

1. 减少缺失代价通过多级Caches
2. 减少缺失代价通过关键字优先
3. 减少缺失代价通过读缺失优先于写缺失
4. 减少缺失代价通过合并写缓冲区
5. 减少缺失代价通过牺牲 Caches

## 4.3.3 利用并行减少 Cache 缺失代价或缺失率

- 减少缺失率——4种技术
  - 减少缺失代价——5种技术
  - 利用并行减少缺失代价或缺失率——3种
  - 减少 cache命中时间
- 非阻塞cache  
硬件预取  
编译器控制预取

$$\text{平均访存时间} = \text{命中时间} + \text{缺失率} \times \text{缺失代价}$$

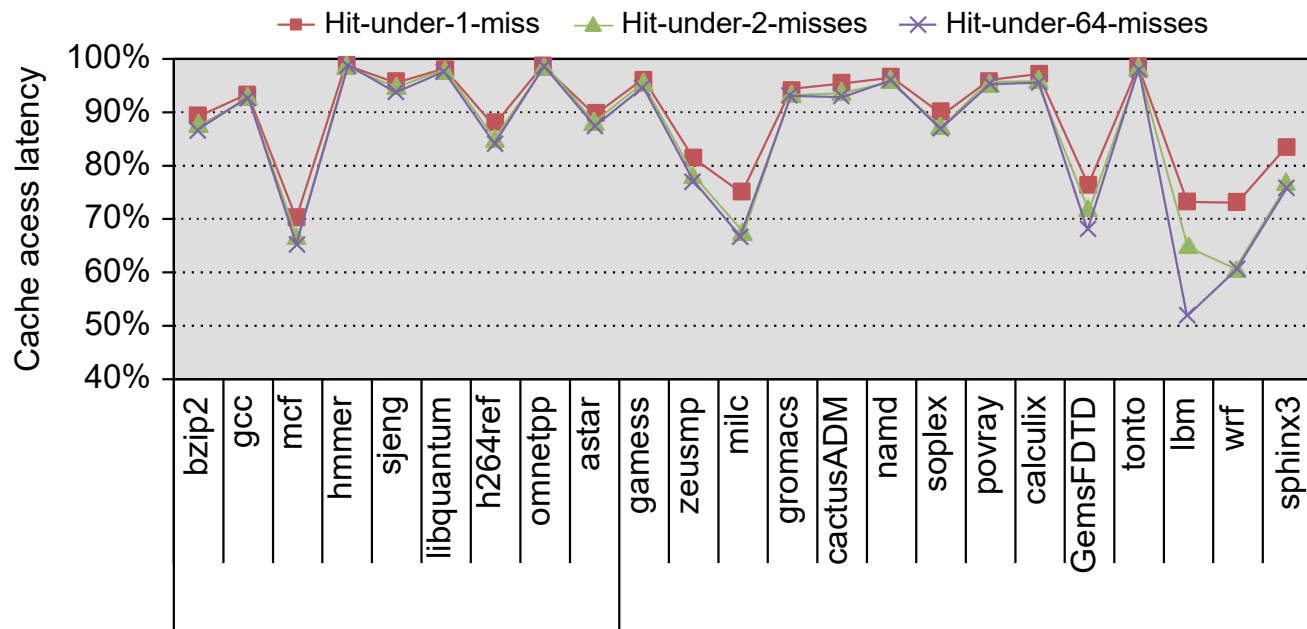


# 第1种缺失代价/缺失率降低减少技术： 非阻塞cache增加cache带宽

## 方法——减少缺失代价

- 乱序执行的流水线处理器，在Cache缺失停顿时，仍然可以执行其他无关指令。如果其他指令又要访问Cache？
- 非阻塞cache（无锁 cache）：在处理读缺失过程中，允许cache 继续提供命中（“缺失下命中”，“多重缺失下命中”）
- 复杂 caches 甚至能够重叠多个缺失（“缺失下缺失”），将进一步有效地降低缺失代价。

**Figure 2.5** 针对18个SPEC2006 程序，在缺失数量变化时，“缺失下命中”策略占阻塞cache平均访存延迟的比例。



这个数据存储器系统以Intel i7建模, 它包括 32KiB L1缓存, 访问延迟为4个周期。L2缓存(与指令共享)为256KiB, 访问延迟为10个周期。L3为2MiB, 访问延迟为36个周期。所有缓存都是八路组相联, 块大小为 64 字节。在缺失时允许1次命中, 可以使整数基准测试的缺失代价降低9%, 浮点基准测试的缺失代价降低12.5%。允许2次命中会将这些结果提高到10%和16%, 而允许命中64次基本不会进一步改进结果

$$\text{平均访存时间} = \text{命中时间} + \text{缺失率} \times \text{缺失代价}$$

## 例：非阻塞cache

假设： 针对图2.5 描述的32KB cache；

浮点直接映像cache缺失率： 5.2%

浮点 2-路 cache缺失率： 4.9%

整数直接映像cache缺失率： 3.5%

整数 2-路 cache缺失率： 3.2%

假设缺失代价都是10个时钟周期

- ① 对于浮点程序，用平均访存时间，判断哪种策略更好：  
阻塞cache 2-路组相联还是 “一次缺失下命中”？
- ② 对于整数程序情况又怎么样？

**假设：** 针对图2.5 描述的32KB cache；

浮点直接映像cache缺失率： 5.2%

浮点 2-路 cache缺失率： 4.9%

整数直接映像cache缺失率： 3.5%

整数 2-路 cache缺失率： 3.2%

假设缺失代价都是10个时钟周期

① 对于浮点程序，用平均访存时间，判断哪种策略更好：

阻塞cache 2-路组相联还是“直接映像一次缺失下命中”？

② 对于整数程序情况又怎么样？

**答案：**

浮点程序的平均访存时间是：

$$\text{缺失率}_{\text{直接映像}} \times \text{缺失代价} = 5.2\% \times 10 = 0.52$$

$$\text{缺失率}_{2\text{-路}} \times \text{缺失代价} = 4.9\% \times 10 = 0.49$$

2-路的访存停顿时间与直接映像cache的时间之比为  $0.49/0.52$  或  $94.2\%$  。

在缺失时允许1次命中, 如果可以使浮点基准测试的缺失代价降低12.5%，则“直接映像一次缺失下命中”更好。

**例：非阻塞cache**

假设：针对图2.5 描述的32KB cache；

浮点直接映像cache缺失率： 5.2%

浮点 2-路 cache缺失率： 4.9%

整数直接映像cache缺失率： 3.5%

整数 2-路 cache缺失率： 3.2%

假设缺失代价都是10个时钟周期

② 对于整数程序情况又怎么样？

答案：

整数程序的平均访存停顿时间是：

$$\text{缺失率}_{\text{直接映像}} \times \text{缺失代价} = 3.5\% \times 10 = 0.35$$

$$\text{缺失率}_{2\text{-路}} \times \text{缺失代价} = 3.2\% \times 10 = 0.32$$

2-路的访存时间与直接映像cache的时间之比为  $0.32/0.35$  或 **91.4%**。

在缺失时允许1次命中, 如果可以使整数基准测试的缺失代价降低9%，则这两种策略基本相同。

## 第2种缺失代价/缺失率降低技术： 指令和数据的硬件预取

### 方法——减少缺失

- 在CPU实际需要访存数据以前，提前从主存取数据。
- 在产生访存请求之前获取数据可减少强制缺失和缺失代价。
- 可能增加其他缺失，如从cache中移走了有用的块。
  - 因此，很多 caches 会增加一个特殊的缓冲器中保存预取的块
- 例如：指令预取
  - Alpha 21064设置一块32B的“流缓冲器”。
  - 缺失时先访问流缓冲器，命中读出该块，发下一块预取请求；如不命中，则从下级存储器取该块和下一块到Cache和流缓冲器
- 预取与CPU执行可以同时进行

## Alpha21064两级cache。

- 一级cache：指令cache、数据cache。这两个caches **8KB**容量，256块（**8位索引**），每块**32B**（**5位块内位移**），直接映像，写直达法，不按写分配，写合并缓冲。
- 二级cache是一个外部cache，有**128KB~8MB** 容量。  
本例是2MB，65536块（**16位索引**），每块**32B**，直接映像，需要**29位块地址**。
- 二级**Cache**采用写回法，按写分配：失效可能导致把旧块（被替换出的块）写回存储器。被替换出的块（如果为脏）放入**Victim缓冲器**，新数据一到达就立即被装入Cache，然后才把旧数据从Victim写回存储器（**读失效优先于写**）。
- **指令预取流缓冲器**，可存放**1块32B**。

# 第3种缺失代价/缺失率降低技术： 编译器控制的预取

## 方法——减少缺失

- 编译器插入**预取指令**请求数据（在实际使用这些数据之前）
- 有两种预取方式：
  - 捆绑预取Binding prefetch：请求预取的值直接装入寄存器。  
(HP PA-RISC loads)
  - 非捆绑预取：将数据预取到 cache，不放入寄存器。  
(MIPS IV, PowerPC, SPARC v. 9)
- 通常特殊的预取指令不会引起异常：一种特殊的执行形式
- 发射执行预取指令需要花费时间
  - 预取产生的开销 < 减少缺失节省的开销？
  - 超标量支持多指令发射有助于提高预取指令执行效率



- ❖ 对于以下代码, 判断哪些访问可能导致数据缓存缺失。然后, 插入预取指令, 以减少缺失。最后, 计算所执行的预取指令数和通过预取避免的缺失数。假设有一个 8KiB 直接映射的数据缓存, 块大小为 16 字节, 它是一个执行写分配的写回缓存。a 和 b 是双精度浮点数组, 所以它们的元素长 8 字节。a 有 3 行、100 列, b 有 101 行、3 列。另外假定在程序启动时, 这些数据不在缓存中。

```
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];
```

- ❖ 对于以下代码, 判断哪些访问可能导致数据缓存缺失。然后, 插入预取指令, 以减少缺失。最后, 计算所执行的预取指令数和通过预取避免的缺失数。假设有一个 8KiB 直接映射的数据缓存, 块大小为 16 字节, 它是一个执行写分配的写回缓存。a 和 b 是双精度浮点数组, 所以它们的元素长 8 字节。a 有 3 行、100 列, b 有 101 行、3 列。另外假定在程序启动时, 这些数据不在缓存中。

```
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];
```

- ❖ 编译器首先判断哪些访问可能导致缓存缺失, 否则我们可能会对那些能够命中的数据发出预取指令, 白白浪费时间。a 的元素是以它们在存储器中的存储顺序写入的, 所以 a 可以受益于空间局部性: j 的偶数值会缺失, 奇数值会命中。由于 a 有 3 行、100 列, 所以对它的访问将会导致  $3 \times (100/2) = 150$  次缺失。

- ❖ 对于以下代码, 判断哪些访问可能导致数据缓存缺失。然后, 插入预取指令, 以减少缺失。最后, 计算所执行的预取指令数和通过预取避免的缺失数。假设有一个 8KiB 直接映射的数据缓存, 块大小为 16 字节, 它是一个执行写分配的写回缓存。a 和 b 是双精度浮点数组, 所以它们的元素长 8 字节。a 有 3 行、100 列, b 有 101 行、3 列。另外假定在程序启动时, 这些数据不在缓存中。

```
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];
```

- ❖ 数组 b 不会从空间局部性中获益, 因为对它的访问不是按照存储顺序执行的。数组 b 可以从时间局部性中获得双重受益: 每次对 i 进行迭代时会访问相同的元素, 每次对 j 进行迭代时使用的 b 元素值与上一次迭代相同。忽略可能存在的冲突缺失, 由 b 导致的缺失将在 j=0 时访问 b[j+1][0] 时出现, 以及在 j=0 时首次访问 b[j][0] 时出现。当 i=0 时, j 从 0 增至 99, 所以对 b 的访问将导致 100+1=101 次缺失。
- ❖ 因此, 这次循环将会出现的数据缓存缺失大约包括 a 的 150 次和 b 的 101 次, 也就是 251 次缺失。

- ❖ 对于以下代码, 判断哪些访问可能导致数据缓存缺失。然后, 插入预取指令, 以减少缺失。最后, 计算所执行的预取指令数和通过预取避免的缺失数。假设有一个 8KiB 直接映射的数据缓存, 块大小为 16 字节, 它是一个执行写分配的写回缓存。a 和 b 是双精度浮点数组, 所以它们的元素长 8 字节。a 有 3 行、100 列, b 有 101 行、3 列。另外假定在程序启动时, 这些数据不在缓存中。

```
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];
```

- ❖ 为了简化优化过程, 我们不用费心为循环中的第一次访问进行预取。这些内容可能已经放在缓存中了, 或者我们需要为 a 或 b 的前几个元素承担缺失代价。在到达循环末尾时, 预取操作会尝试提前获取超出 a 末端之外的内容 (a[i][100]... a[i][106]) 和 b 末端之外的内容 (b[101][0] ...b[107][0]), 我们也不需要费心来禁止这些预取。如果这些是故障性预取, 那我们也许不能承担如此之大的开销。我们假定缺失代价非常大, 需要至少提前 (比如) 7 次迭代开始预取。(换句话说, 我们假定在第 8 次迭代之前进行预取不会带来任何好处。) 以下代码中加下划线的部分, 是为了添加预取优化而对前面代码所做的修改。

```
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];
```

❖ 修改为

```
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];}
```

- ❖ 第一次循环中访问元素  $b[0][0], b[1][0], \dots, b[6][0]$  时的 7 次缺失
- ❖ 第一次循环中访问元素  $a[0][0], a[0][1], \dots, a[0][6]$  时的 4 次缺失 ( $\lceil 7/2 \rceil$ ) (利用空间局部性将缺失数减少为每 16 字节缓存块一次缺失)
- ❖ 第二次循环中访问元素  $a[1][0], a[1][1], \dots, a[1][6]$  的 4 次缺失 ( $\lceil 7/2 \rceil$ )
- ❖ 第二次循环中访问元素  $a[2][0], a[2][1], \dots, a[2][6]$  的 4 次缺失 ( $\lceil 7/2 \rceil$ )
- ❖ 即总共 19 次非预取缺失。避免 232 次缓存缺失的成本是执行了 400 条预取指令, 这很划算。

```
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];  
        }
```

- ❖ 计算上个例题中节省的时间。忽略指令缓存缺失, 并假定数据缓存中没有冲突缺失或容量缺失。假定预取过程可以相互重叠, 并能与缓存缺失重叠。因此可以用最高存储带宽进行传输。下面是忽略缓存缺失的关键循环次数: 原循环每次迭代需要 7 个时钟周期, 第一次预取循环每次迭代需要 9 个时钟周期, 第二次预取循环每次迭代需要 8 个时钟周期 (包含循环外部的开销)。一次缺失需要 100 个时钟周期。

- ❖ 计算上个例题中节省的时间。忽略指令缓存缺失, 并假定数据缓存中没有冲突缺失或容量缺失。假定预取过程可以相互重叠, 并能与缓存缺失重叠。因此可以用最高存储带宽进行传输。下面是忽略缓存缺失的关键循环次数: 原循环每次迭代需要 7 个时钟周期, 第一次预取循环每次迭代需要 9 个时钟周期, 第二次预取循环每次迭代需要 8 个时钟周期 (包含循环外部的开销)。一次缺失需要 100 个时钟周期。
- ❖ 解: 原来的双层嵌套循环执行  $3 \times 100 = 300$  次。由于该循环每次迭代需要 7 个时钟周期, 所以总共需要  $300 \times 7 = 2100$  个时钟周期再加上缓存缺失。缓存缺失增加  $251 \times 100 = 25100$  个时钟周期, 所以总共需要 27200 个时钟周期。第一次预取循环迭代 100 次, 每次迭代需要 9 个时钟周期, 所以总共需要 900 个时钟周期再加上缓存缺失。加上缓存缺失的  $11 \times 100 = 1100$  个时钟周期, 总共需要 2000 个时钟周期。第二次循环执行  $2 \times 100 = 200$  次, 每次迭代需要 8 个时钟周期, 所以一共需要 1600 个时钟周期, 再加上缓存缺失的  $8 \times 100 = 800$  个时钟周期, 总共需要 2400 个时钟周期。由上个例子可知, 这段代码为了执行这两个循环, 在  $2000 + 2400 = 4400$  个时钟周期内执行了 400 条预取指令。如果假定这些预取操作完全与其他执行过程相重叠, 那么这段预取代码要快  $27\ 200 / 4400 = 6.18$  倍。



## 小结：利用并行减少 Cache 缺失代价或缺失率

$$CPUtime = IC \times \left( CPI_{Execution} + \frac{Memory\ accesses}{Instruction} \times \text{Miss rate} \times Miss\ penalty \right) \times Clock\ cycle\ time$$

1. 减小缺失代价：非阻塞 Caches
2. 减少缺失次数：硬件预取
3. 减少缺失次数：编译器控制预取

## 4.3.4 减少Cache命中时间

- 减少缺失率——4种技术
- 减少缺失代价——5种技术
- 利用并行减少缺失代价或缺失率——3种
- 减少 cache命中时间——5种技术

- 小和简单的cache
- 在cache索引时避免地址转换
- 流水线化cache访问
- 路预测
- 踪迹cache

平均访存时间 = 命中时间 + 缺失率 × 缺失代价

# 第1种命中时间减少技术： 小和简单的L1 Caches

## 方法

使用小的和直接映像 cache

- 实现一个cache必要的硬件越少，通过硬件的关键路径就越短。
- **直接映像**无论是读还是写都快于组相联cache。还能降低功耗。
- 命中的cache与CPU在同一芯片上靠的很近，对于加快访问时间是非常重要的。

## Figure 2.3 访问时间的对比

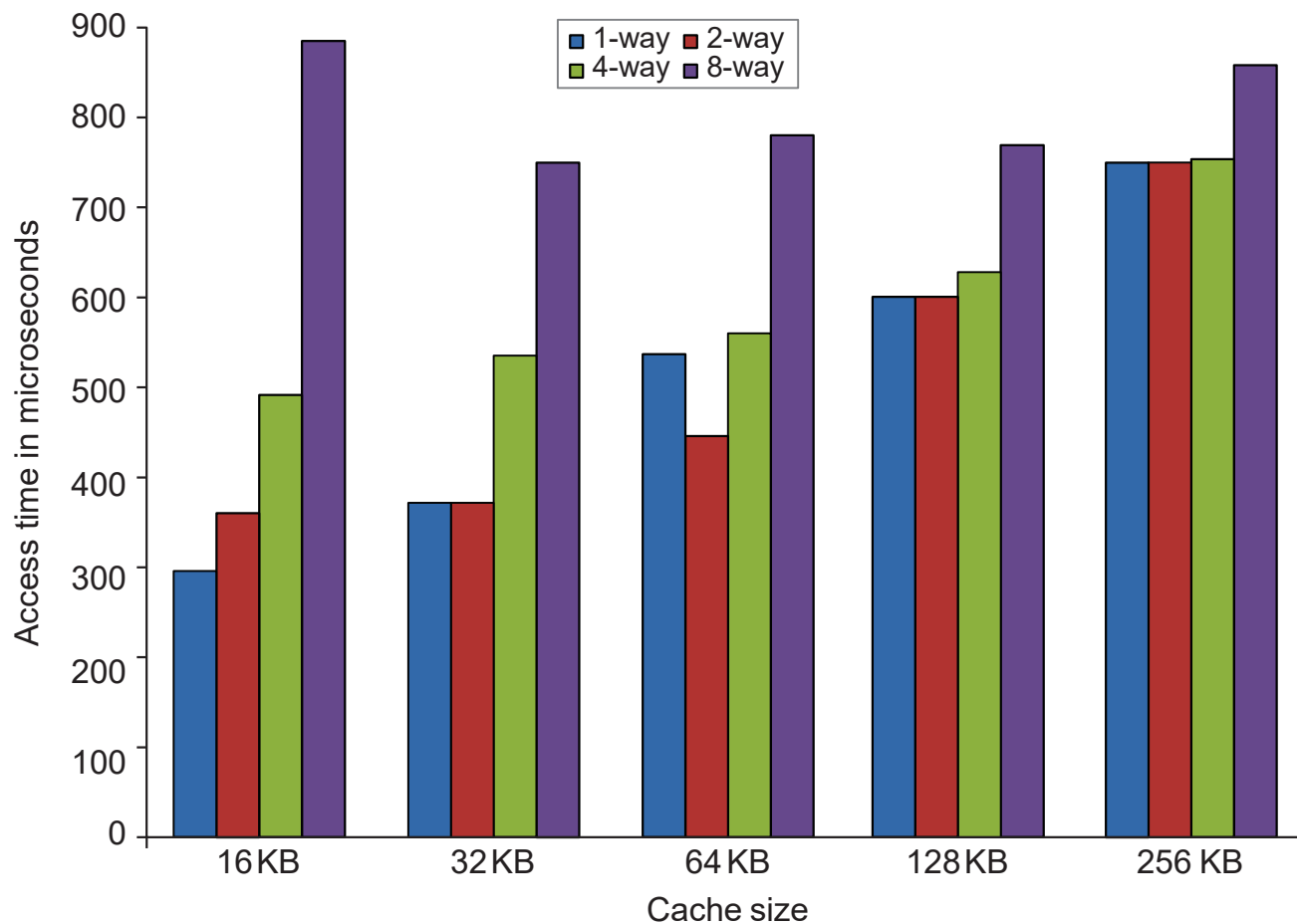


Figure 2.3 Access times generally increase as cache size and associativity are increased.  
访问时间通常会随缓存大小和相联程度的增大而增加

## Figure 2.4 能耗的对比

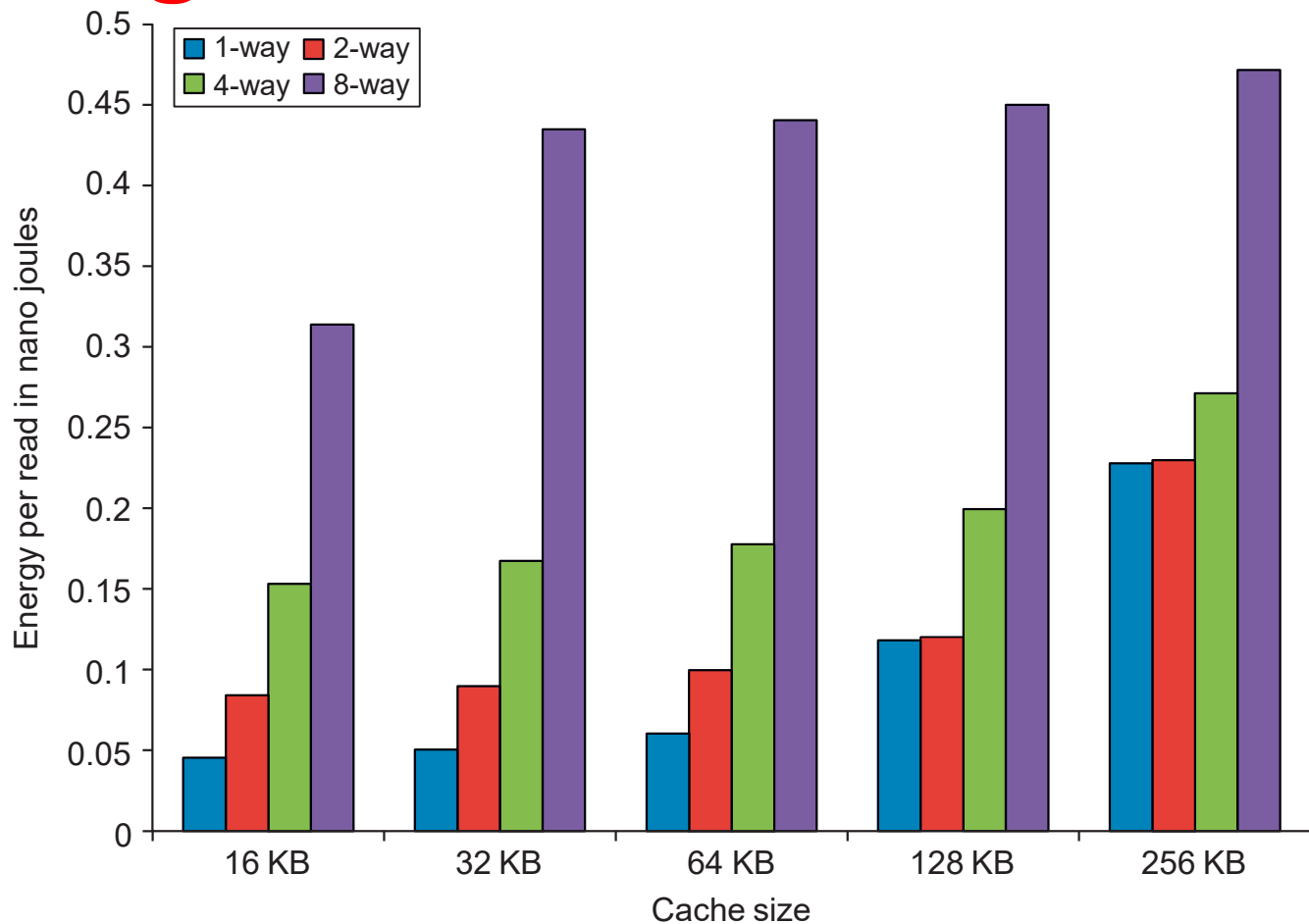


Figure 2.4 Energy consumption per read increases as cache size and associativity are increased

每次读操作的能耗通常会随缓存大小和相联程度的增大而增加

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?

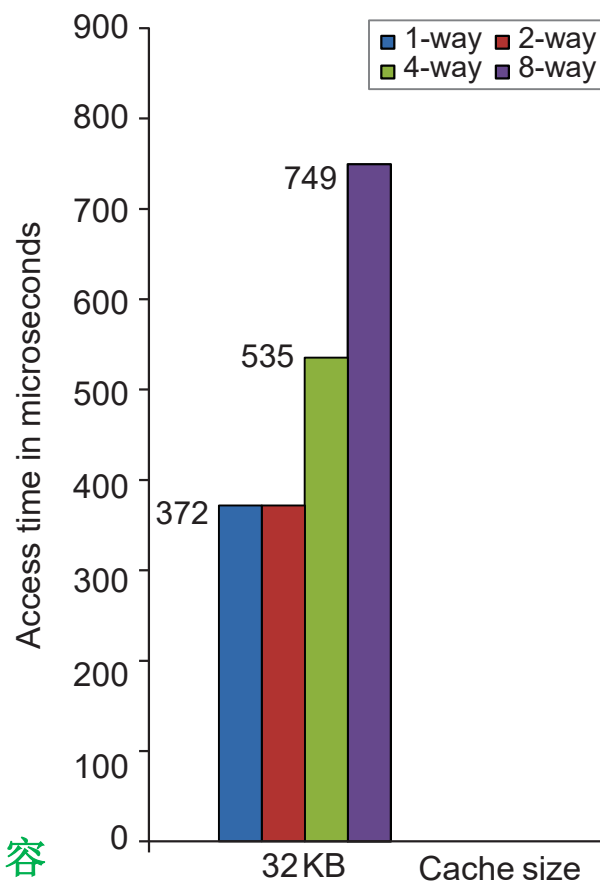
利用图2-3和附录B中图B-8中的数据，判断32KiB四路组相联L1缓存的存储器访问时间是否快于32KiB两路组相联L1缓存。假定L2的缺失代价是快速L1缓存访问时间的15倍。忽略L2后续存储层次的缺失。哪种缓存的存储器平均缓存时间较短？

Figure B.8 部分内容

| Cache size (KB) | Degree associative | Total miss rate |
|-----------------|--------------------|-----------------|
| 32              | 2-way              | 0.038           |
| 32              | 4-way              | 0.037           |

Figure 2.3 部分内容

# Example



**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:

$$\begin{aligned}\text{Average memory access time}_{2\text{-way}} &= \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \\ &= 1 + 0.038 \times 15 = 1.57\end{aligned}$$

For the four-way cache, the access time is  $535/372=1.438$  times longer. Average memory access time<sub>4-way</sub> = Hit time<sub>2-way</sub> × 1.438 + Miss rate × Miss penalty

$$= 1.438 + 0.037 \times 15 = 1.993$$

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.

显然, 采用较高的相联度看起来是一种糟糕的权衡选择; 不过, 由于现代处理器中的缓存访问通常都实现了流水化, 所以很难评估对时钟周期时间的具体影响。

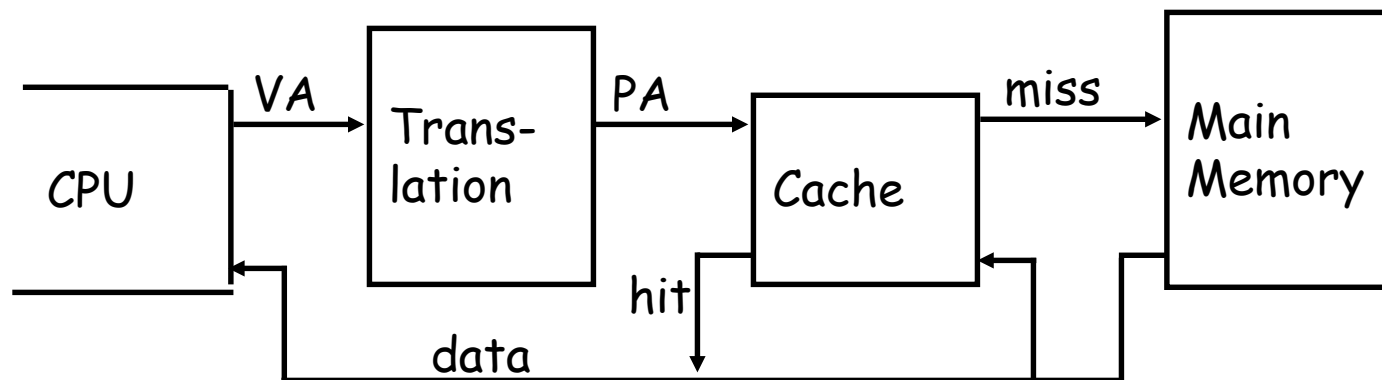
注: 教材5e的Average memory access time<sub>2-way</sub> 计算有误, 第6版已更正。

教材5e和6e在计算Average memory access time<sub>4-way</sub>时使用的Miss penalty为10, 怀疑有误。

## 第2种命中时间减少技术： 在cache索引时避免地址转换

### 传统物理地址Cache存在的问题：地址转换

- CPU使用虚拟地址VA，Cache使用物理地址PA。



物理Cache示意结构

- 页表在主存中，是一个大的数据结构
- 从主存取数据、存结果、取指令，需要2次访问主存！



# 变换旁路缓冲器TLBs

一种快速地址转换的方式是使用一个特殊的缓存cache  
存放最近用过的页表项 --

最常用的名字为 *Translation Lookaside Buffer or TLB*

| Virtual Address | Physical Address | Dirty | Ref | Valid | Access |
|-----------------|------------------|-------|-----|-------|--------|
|                 |                  |       |     |       |        |

实际上是页表映射的一个cache

TLB 的访问时间与cache 访问时间相当  
(大大少于主存访问时间)

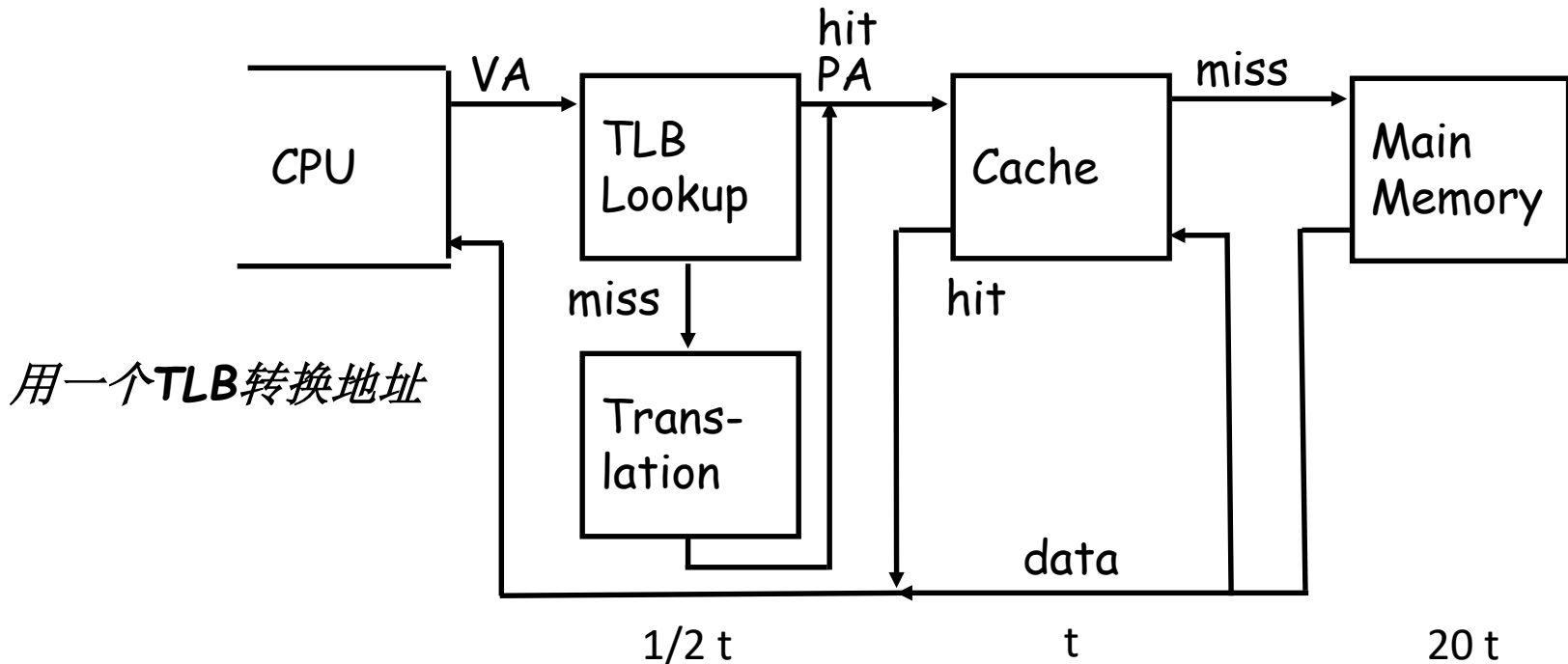
## TLBs

类似于 **cache**，**TLB** 可以采用全相联、组相联、直接映像组织结构。

**TLBs** 通常是小的，

- \* 在高性能处理器中典型大小不超过 **128 ~ 256** 项，采用全相联结构来检索。

- \* 中档处理器采用小的 **n-路组相联** 结构。



# 避免VA→PA地址转换的方法

Cache 也可以使用虚拟地址

- 一个 cache 用虚拟地址检索称为虚拟 cache （相对于物理 cache ）。
- 虚拟索引物理标识：地址转换与cache访问并行进行，可以减少命中时间。
  - 避免索引cache时的地址转换
- 虚拟cache虚拟标识：在命中时，不需要从虚拟地址转换为物理地址。
  - 同义问题：OS与用户程序使用不同的虚拟地址可能映射到同一物理地址，导致两个Cache副本

# 通过避免地址转换加快命中： 虚拟cache虚拟标识

- 虚拟Cache虚拟标识
  - 每次进程切换时，逻辑上必须刷新cache，否则会假命中
    - 代价：刷新时间 + 来自空cache的“强制”缺失
    - 解决方法：增加tag宽度，加上进程标识符PID (*process identifier tag*) 识别进程
- 处理别名或同义synonyms：两个不同VA映射到同一个PA
  - 硬件别名消去：保证每个cache块有唯一的地址
  - 软件方法：让所有别名地址的低位相同（称为页着色 page coloring）
- I/O 与cache交互：I/O一般使用物理地址，  
因此需要将物理地址PA映射为VA

## 第3种命中时间减少技术： 流水线化Cache 访问

### 方法

写命中比读命中花费更长时间，因为在检查标识后才能写入数据。

- 将写操作**分为两步**：第1步标识检查；  
第2步写数据。
- 本次写的第2步与下次标识比较的第1步同时进行。

## 第4种命中时间减小技术：路预测

### 路预测：针对组相联

- 在cache中预留特殊的位，用来预测下一次访问cache可能在组中会用到的路或块。
  - Alpha 21264 在2-路组相联指令cache中采用了路预测：
    - 如果预测正确，指令cache的延迟是 1 个时钟周期。
    - 如果预测错误，则选择其他的块，改变路预测器，需要3个时钟周期。
    - 使用 SPEC95的模拟表明组预测精度超过85%，，因此路预测在超过85%的取指令操作中可以节省流水线的步骤。
- 适合推测执行处理器，预测错误时操作已经被撤销。

*减少命中时间，减少功耗*

## 第4种命中时间减小技术：路预测

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 draw-back for way selection is that it makes it difficult to pipeline the cache access.

Inoue 等人 [1999]根据 SPEC95 基准测试进行估算, 对于四路组相联缓存使用路选择方法, 可以使指令缓存的平均访问时间增加 1.04 倍, 数据缓存增加 1.13 倍, 但与普通的四路组相联缓存相比, 指令缓存的平均缓存功耗降为原来的 0.28, 数据缓存降为原来的 0.35。路选择方法的一个重要缺点就是它增大了实现缓存访问流水化的难度。然而, 随着能耗问题受关注度的增加, 适时对缓存做低功耗处理的方案越来越有意义。

# 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.

假定在一个普通四路组相联实现中，数据缓存访问次数是指令缓存访问次数的一半，指令缓存和数据缓存分别占用该处理器功耗的25%和15%。根据上述研究的估计值，判断路选择方法是否提高了每瓦功耗的性能。



**Example** 假定在一个普通四路组相联实现中, 数据缓存访问次数是指令缓存访问次数的一半, 指令缓存和数据缓存分别占用该处理器功耗的25%和15%。根据上述研究的估计值, 判断路选择方法是否提高了每瓦功耗的性能。

**Answer** 对于四路组相联缓存使用路选择方法, 可以使指令缓存的平均访问时间增加 1.04 倍, 数据缓存增加 1.13 倍, 但与普通的四路组相联缓存相比, 指令缓存的平均缓存功耗降为原来的 0.28, 数据缓存降为原来的 0.35 。

对于指令缓存, 节省的功耗为总功耗的  $25\% \times 0.28 = 0.07$ ; 对于数据缓存, 节省的功耗为  $15\% \times 0.35 \approx 0.0525$ 。一共  $0.07 + 0.0525 = 0.1225$ 。路预测版本需要的功耗为标准四路缓存的  $1 - 0.1225 = 0.8775$ 。缓存访问时间的增加量等于指令缓存平均访问时间的增加量加上数据缓存访问时间增加量的一半, 即  $1.04 + 0.5 \times 0.13 = 1.105$  倍。这一结果意味着路选择的性能是标准四路缓存的  $1/1.105 = 0.905$ 。因此, 路选择方法略微提高了每焦功耗的性能, 比值为  $0.905/0.8775 = 1.031$ 。这种优化方法最适用于功耗比性能更重要的情景。

**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** A four-way set associative cache **using the way selection** approach 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.

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  $15\% \times 0.35 = 0.0525$  for a total savings of  $0.07 + 0.0525 = 0.1225$ . The way prediction version requires  $1 - 0.1225 = 0.8775$  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  $1.04 + 0.5 \times 1.13 = 1.105$  times longer. This result means that way selection has  $1/1.105 = 0.905$  of the performance of a standard four-way cache. Thus, way selection improves performance per joule very slightly by a ratio of  $0.905/0.8775 = 1.031$ . This optimization is best used where power rather than performance is the key objective.

# 第5种命中时间减少技术：踪迹 caches

## 方法

踪迹cache：块中是**动态指令序列**（包括发生分支），而不是限制指令在一个静态cache块中（空间局部性）。

Cache块中包含了**由CPU确定的要执行指令的动态踪迹**，而不是仅由存储器确定的静态指令序列。

- 转移预测操作需要加入到cache，预测的地址必须在流水线最后的步骤进行验证以证明是一次有效的取操作。

该方法在Pentium 4上使用，踪迹cache中存放的是译码后的微指令。

采用TC替换了指令L1-cache，按转移预测的指令序列进行译码，译码后的微指令序列存放在TC中。实现**微指令的乱序执行**，获得更高执行效率。

程序一个分支的微指令序列称为一个踪迹。

## 小结：减少命中时间

$$AMAT = \text{HitTime} + \text{MissRate} \times \text{MissPenalty}$$

1. 小和简单的Caches
2. 在cache索引时避免地址转换
3. 流水线化Cache 访问
4. 路预测
5. 踪迹 caches

# Cache 优化技术小结

|      | 技术                | 缺失代价 | 缺失率 | 命中时间 | 复杂度           |
|------|-------------------|------|-----|------|---------------|
| 缺失代价 | 1. 多级 caches      | +    |     |      | 2             |
|      | 2. 提前重启动和关键字优先    | +    |     |      | 2             |
|      | 3. 读缺失优先          | +    |     |      | 1             |
|      | 4. 合并写缓冲区         | +    |     |      | 1             |
|      | 5. 牺牲cache        | +    | +   |      | 2             |
| 缺失率  | 6. 更大的块容量         | -    | +   |      | 0             |
|      | 7. 更大的 cache 容量   |      | +   | -    | 1             |
|      | 8. 更高的相联度         |      | +   | -    | 1             |
|      | 9. 编译器优化          |      | +   |      | 0             |
| 并行性  | 10. 非阻塞Caches     | +    |     |      | 3             |
|      | 11. 硬件预取指令或数据     | +    | +   |      | 2instr./3data |
|      | 12. 编译器控制预取       | +    | +   |      | 3             |
| 命中时间 | 13. 小和简单的 Caches  |      | -   | +    | 0             |
|      | 14. 避免地址转换        |      |     | +    | 1             |
|      | 15. 流水线化 Cache 访问 |      |     | +    | 1             |
|      | 16. 路预测 cache     |      |     | +    | 1             |
|      | 17. 踪迹 cache      |      |     | +    | 3             |

“+” 表示该技术会对该项产生积极影响，“-”表示会对该项产生负面影响，空白则表示没有影响。复杂度的评估具有主观性，其中0表示最容易，3表示难度很大。