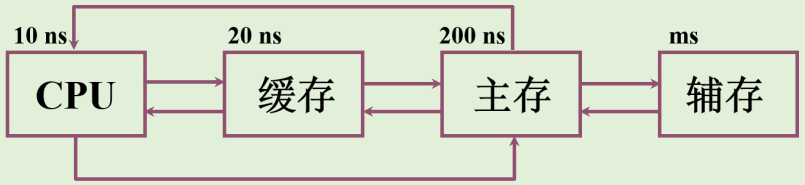
计算机系统结构

第二章 缓存优化

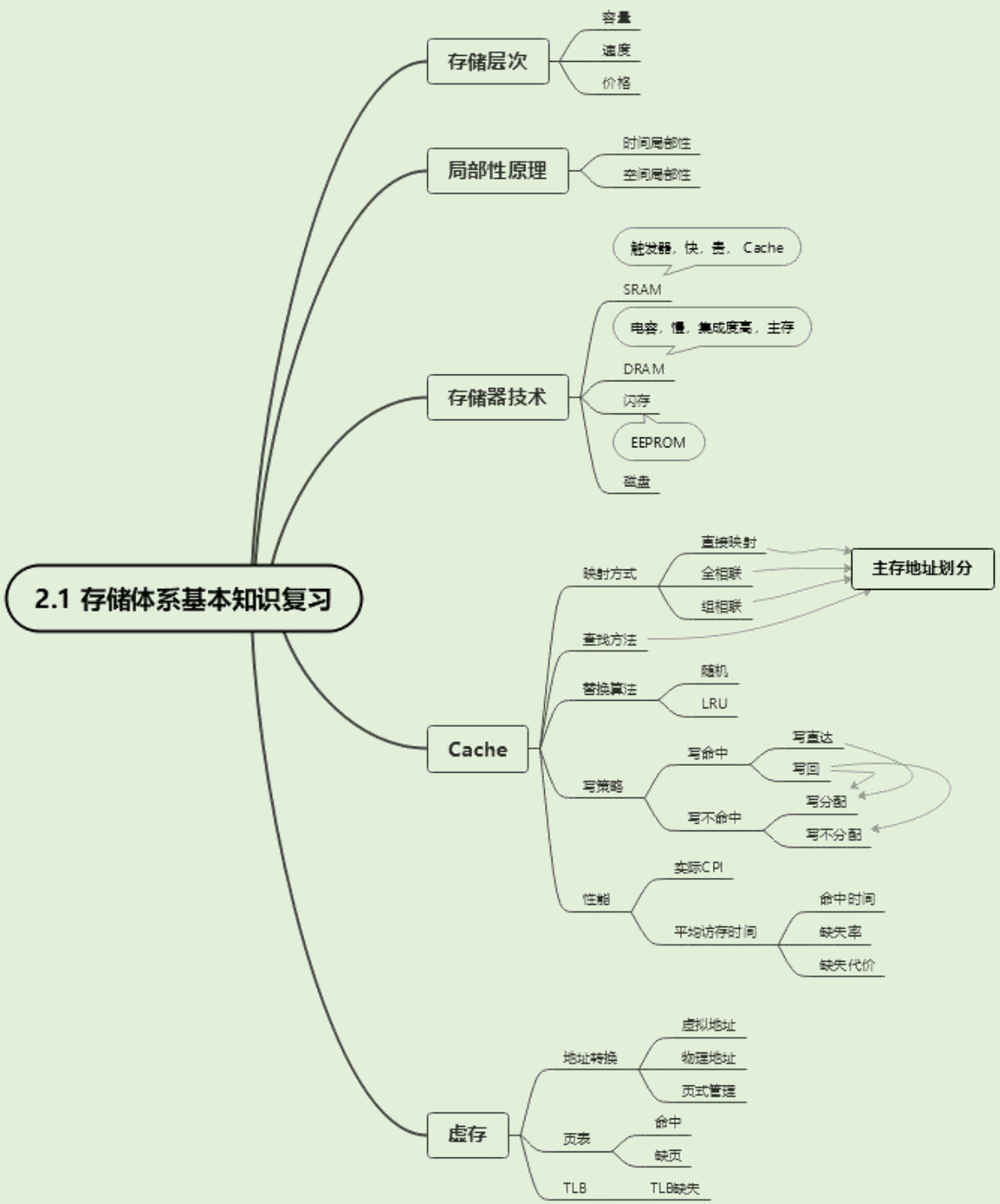
2.1 存储体系基础知识复习



（速度）　（容量）

缓存－主存　主存－辅存

主存储器	虚拟存储器
实地址	虚地址
物理地址	逻辑地址



2.2 Cache 性能分析与改进

2.2.1 平均访存时间与程序执行时间

- 不命中率 （失效率）
- 平均访存时间

$$\text{平均访存时间} = \text{命中时间} + \text{不命中率} \times \text{不命中开销}$$

- 程序执行时间

CPU时间＝（CPU执行周期数+存储器停顿周期数）× 时钟周期时间

其中：

存储器停顿时钟周期数＝“读”的次数×读不命中率×读不命中开销

＋“写”的次数×写不命中率×写不命中开销

存储器停顿时钟周期数＝访存次数×不命中率×不命中开销

CPU时间＝（CPU执行周期数+访存次数×不命中率×不命中开销）

× 时钟周期时间

=*IC*×（*CPI*execution＋每条指令的平均访存次数×不命中率

×不命中开销）× 时钟周期时间

例　用一个和Alpha AXP类似的机器作为第一个例子。假设Cache不命中开销为50个时钟周期，当不考虑存储器停顿时，所有指令的执行时间都是2.0个时钟周期，访问Cache不命中率为2%，平均每条指令访存1.33次。试分析Cache对性能的影响。

解　CPU时间$_{有cache}$＝*IC* ×（*CPI*execution＋每条指令的平均访存次数
×不命中率×不命中开销）× 时钟周期时间
＝IC ×（2.0＋1.33×2 %×50）× 时钟周期时间
＝IC × 3.33× 时钟周期时间

考虑Cache的不命中后，性能为：

CPU时间$_{有cache}$ ＝IC×（2.0＋1.33×2 %×50）×时钟周期时间
＝IC×3.33×时钟周期时间

实际*CPI* ： 3.33

3.33/2.0 = 1.67(倍)

CPU时间也增加为原来的1.67倍。

但若不采用Cache,则：

CPI＝2.0＋50×1.33＝68.5

- Cache不命中对于一个CPI较小而时钟频率较高的CPU来说，影响是双重的：
  - $CPI_{execution}$越低，固定周期数的Cache不命中开销的相对影响就越大。
  - 在计算CPI时，不命中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的不命中开销较大，其CPI中存储器停顿这部分也就较大。

因此Cache对于低CPI、高时钟频率的CPU来说更加重要。

例　考虑两种不同组织结构的Cache：直接映像Cache和两路组相联Cache，试问它们对CPU的性能有何影响？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%）情况下的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32字节。

(3)在组相联Cache中，由于多路选择器的存在而使CPU的时钟周期增加到原来的1.10倍。这是因为对Cache的访问总是处于关键路径上，对CPU的时钟周期有直接的影响。

(4) 这两种结构Cache的不命中开销都是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个时钟周期，64KB直接映像Cache的不命中率为1.4%，相同容量的两路组相联Cache的不命中率为1.0%。

(1)理想Cache（命中率为100%）情况下的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32字节。

(3)在组相联Cache中，由于多路选择器的存在而使CPU的时钟周期增加到原来的1.10倍。这是因为对Cache的访问总是处于关键路径上，对CPU的时钟周期有直接的影响。

(4) 这两种结构Cache的不命中开销都是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个时钟周期，64KB直接映像Cache的不命中率为1.4%，相同容量的两路组相联Cache的不命中率为1.0%。

解　平均访存时间为：

$$\text{平均访存时间} = \text{命中时间} + \text{不命中率} \times \text{不命中开销}$$

因此，两种结构的平均访存时间分别是：

$$\text{平均访存时间}_{1\text{路}} = 2.0 + (0.014 \times 70) = 2.98\text{ns}$$

$$\text{平均访存时间}_{2\text{路}} = 2.0 \times 1.10 + (0.010 \times 70) = 2.90\text{ns}$$

两路组相联Cache的平均访存时间比较低。

CPU时间＝IC×（CPIexecution＋每条指令的平均访存次数×不命中率×不命中开销）× 时钟周期时间

＝IC×（CPIexecution× 时钟周期时间＋每条指令的平均访存次数×不命中率×不命中开销×时钟周期时间）

因此：

$$\text{CPU时间}_{1路} = \text{IC}\times(2.0\times2+(1.3\times0.014\times70))$$
$$= 5.27\times\text{IC}$$

$$\text{CPU时间}_{2路} = \text{IC}\times(2.0\times2\times1.10+(1.3\times0.010\times70))$$
$$= 5.31\times\text{IC}$$

$$\frac{\text{CPU时间}_{2路}}{\text{CPU时间}_{1路}} = \frac{5.31\times\text{IC}}{5.27\times\text{IC}} = 1.01$$

直接映像Cache的平均性能好一些。

(1)理想Cache（命中率为100%）情况下的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32字节。

(3)在组相联Cache中，由于多路选择器的存在而使CPU的时钟周期增加到原来的1.10倍。这是因为对Cache的访问总是处于关键路径上，对CPU的时钟周期有直接的影响。

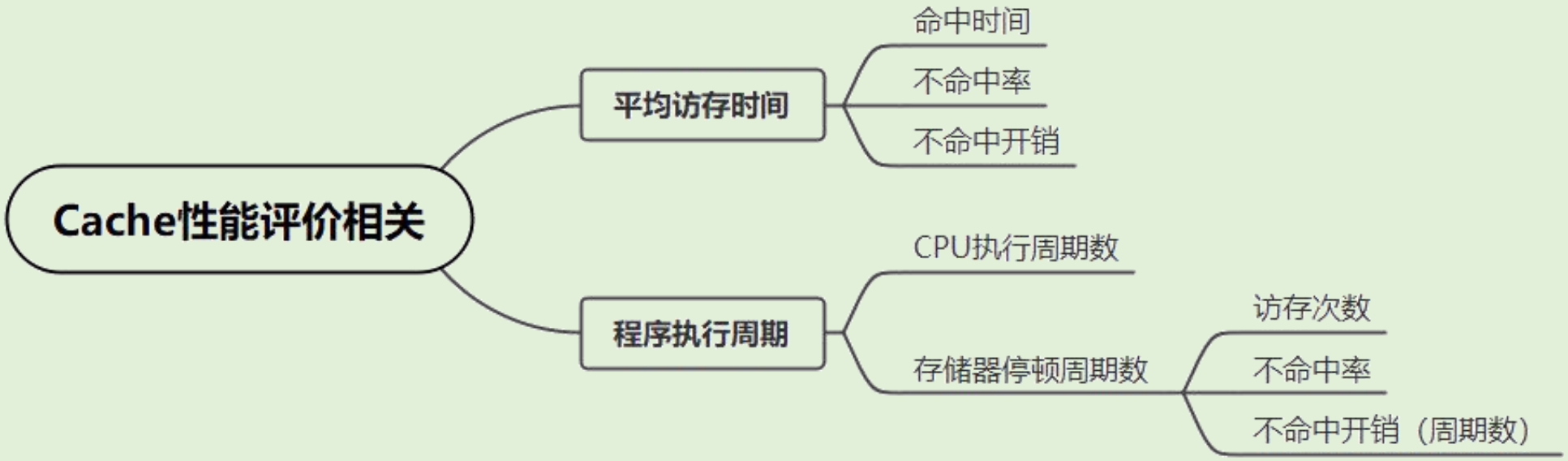
(4) 这两种结构Cache的不命中开销都是70ns。（在实际应用中，应取整为整数个时钟周期）

(5) 命中时间为1个时钟周期，64KB直接映像Cache的不命中率为1.4%，相同容量的两路组相联Cache的不命中率为1.0%。

2.2.2 Cache性能改进

- 平均访存时间＝命中时间＋不命中率×不命中开销
- 可以从三个方面改进Cache的性能：
  - 降低不命中率
  - 减少不命中开销
  - 减少Cache命中时间
- 下面介绍17种Cache优化技术
  - 8种用于降低不命中率
  - 5种用于减少不命中开销
  - 4种用于减少命中时间

总结


Cache性能评价相关
平均访存时间
命中时间
不命中率
不命中开销
程序执行周期
CPU执行周期数
存储器停顿周期数
访存次数
不命中率
不命中开销（周期数）

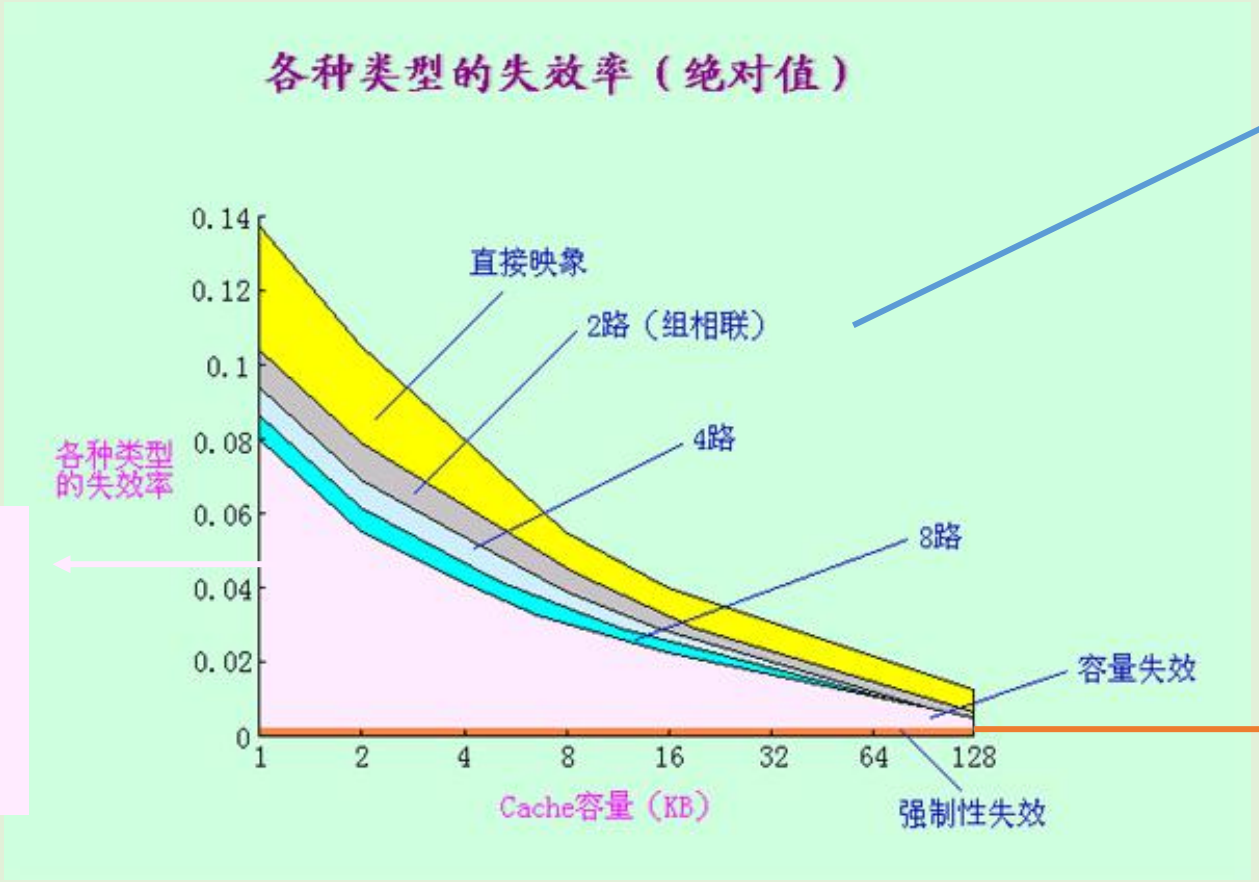
2.3 降低不命中率

2.3.1 三种类型的不命中

- 三种类型的不命中(3C)
  - 强制性不命中(Compulsory miss)
    - 当第一次访问一个块时，该块不在Cache中，需从下一级存储器中调入Cache，这就是强制性不命中。(冷启动不命中，首次访问不命中）
  - 容量不命中(Capacity miss)
    - 如果程序执行时所需的块不能全部调入Cache中，则当某些块被替换后，若又重新被访问，就会发生不命中。这种不命中称为容量不命中。
  - 冲突不命中(Conflict miss)
    - 在组相联或直接映像Cache中，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。这就是发生了冲突不命中。

      (碰撞不命中，干扰不命中)

三种类型的不命中



相联度越高，冲突不命中就越少

容量不命中不受相联度的影响；
但容量不命中却随着容量的增加而减少。

强制性不命中不受Cache容量的影响；
强制性不命中不受相联度的影响。

具体数据，请见学习通章节32.3 必读资料中的表5.3

2.3.2 降低不命中率的方法

- 针对三种类型的不命中的直接方法
  - 针对强制不命中 -- 增加块大小 （方法一）
  - 针对容量不命中 -- 增加Cache容量 （方法二）
  - 针对冲突不命中 -- 提高相联度 （方法三）

- 其他方法
  - 伪相联Cache （方法四）
  - 硬件预取 （方法五）
  - 编译器预取 （方法六）
  - 编译器优化 （方法七）
  - 牺牲Cache （方法八）

方法一：增加块大小

请先观察不命中率随块大小变化的曲线

- 对于给定的Cache容量，当块大小增加时，不命中率开始是下降，后来反而上升了。
- Cache容量越大，使不命中率达到最低的块大小就越大。（从上到下，曲线越来越平）

- 请分析：为什么增加块大小可以降低强制不命中？

- 请思考：为什么不命中率先下降后上升？
  - 一方面它减少了强制性不命中；（不命中率总体下降）
  - 另一方面，由于增加块大小会减少Cache中块的数目，所以有可能会增加冲突不命中。（不命中率又升高了）

- 请考虑：增加块大小还可能带来什么不良后果？

  会增加不命中开销

方法二：增加Cache的容量

- 最直接的方法是增加Cache的容量
  - 缺点:
    - 增加成本
    - 可能增加命中时间
- 这种方法在片外Cache中用得比较多

方法三：提高相联度

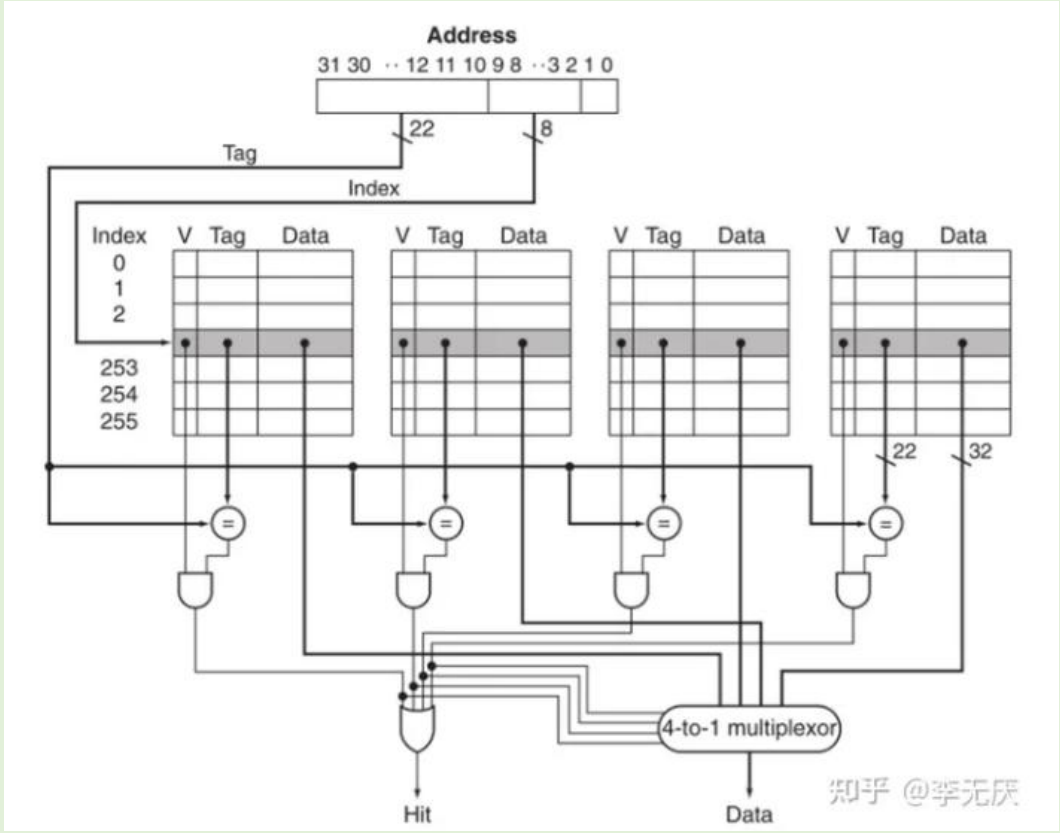
- 采用相联度超过8的方案的实际意义不大。
- 2:1Cache经验规则

  容量为N的直接映像Cache的不命中率和容量为N/2的两路组相联Cache的不命中率差不多相同。
- 提高相联度是以增加命中时间为代价。

方法四：伪相联 Cache（列相联 Cache ）

- 多路组相联 V.S. 直接映像

	优　点	缺　点
直接映像	命中时间小	不命中率高
组相联	不命中率低	命中时间大

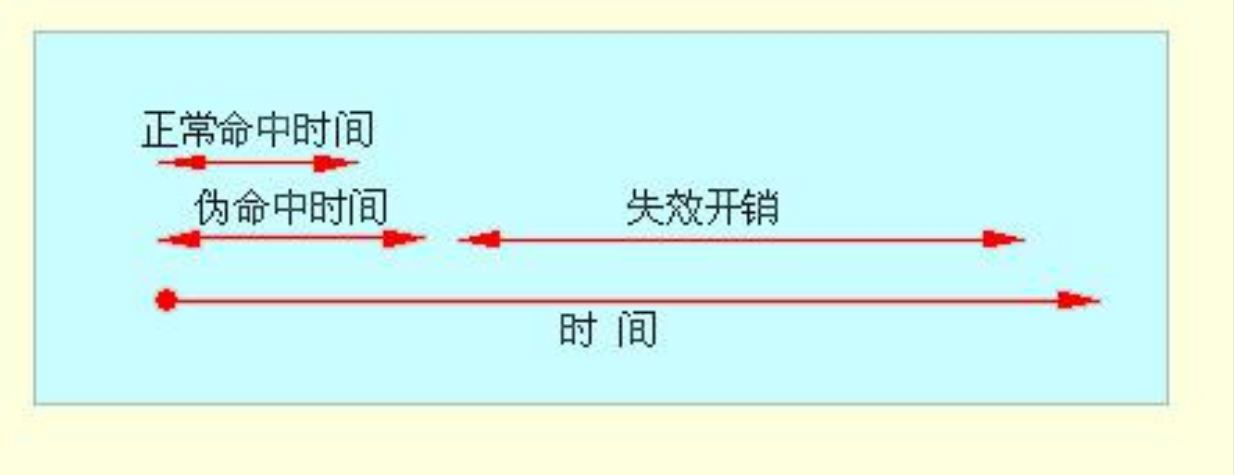


- 伪相联Cache的优点
  - 命中时间小
  - 不命中率低

- 基本思想及工作原理
  - 在逻辑上把直接映像Cache的空间上下平分为两个区。
  - 对于任何一次访问，伪相联Cache先按直接映像Cache的方式去处理。
    - 若命中，则其访问过程与直接映像Cache的情况一样。
    - 若不命中，则再到另一区相应的位置去查找。
      - 若找到，则发生了伪命中.
      - 否则就只好访问下一级存储器。

- 缺点 ：多种命中时间 -- 快速命中与慢速命中

  要保证绝大多数命中都是快速命中。



例 假设当在按直接映象找到的位置处没有发现匹配，而在另一个位置才找到数据（伪命中）需要2个额外的周期。问：当Cache容量分别为2 KB和128 KB时，直接映象、2路组相联和伪相联这三种组织结构中，哪一种速度最快？

Cache容量为2KB时，有：

	失效率	失效开销（周期）	平均访存时间（周期）
1路	0.098	50	5.90
2路	0.076	50	4.90

Cache容量为128KB时，1路、2路失效率分别为：0.010,0.007

	失效率	失效开销（周期）	平均访存时间（周期）
1路	0.010	50	1.50
2路	0.007	50	1.45

解 首先考虑标准的平均访存时间公式：

$$\text{平均访存时间}_{\text{伪相联}} = \text{命中时间}_{\text{伪相联}} + \text{失效率}_{\text{伪相联}} \times \text{失效开销}_{\text{伪相联}}$$

由于：

$$\text{失效率}_{\text{伪相联}} = \text{失效率}_{\text{2路}}$$

$$\text{命中时间}_{\text{伪相联}} = \text{命中时间}_{\text{1路}} + \text{伪命中率}_{\text{伪相联}} \times 2$$

伪相联查找的命中率等于2路组相联Cache的命中率和直接映象Cache命中率之差。

$$\begin{aligned}\text{伪命中率}_{\text{伪相联}} &= \text{命中率}_{\text{2路}} - \text{命中率}_{\text{1路}} \\ &= (1 - \text{失效率}_{\text{2路}}) - (1 - \text{失效率}_{\text{1路}}) \\ &= \text{失效率}_{\text{1路}} - \text{失效率}_{\text{2路}}\end{aligned}$$

综合上述分析，有：

$$\text{平均访存时间}_{\text{伪相联}} = \text{命中时间}_{\text{1路}} + (\text{失效率}_{\text{1路}} - \text{失效率}_{\text{2路}}) \times 2 + \text{失效率}_{\text{2路}} \times \text{失效开销}_{\text{1路}}$$

将前面表中的数据代入上面的公式，得：

$$\text{平均访存时间}_{\text{伪相联，2 KB}} = 1 + (0.098 - 0.076) \times 2 + (0.076 \times 50) = 4.844$$

对于2 KB Cache，有：

$$\text{平均访存时间}_{\text{1路}} = 5.90 \text{ 个时钟}$$

$$\text{平均访存时间}_{\text{2路}} = 4.90 \text{ 个时钟}$$

对于128KB的Cache有，可得：

$$\text{平均访存时间}_{\text{伪相联，128 KB}} = 1 + (0.010 - 0.007) \times 2 + (0.007 \times 50) = 1.356$$

$$\text{平均访存时间}_{\text{1路}} = 1.50 \text{ 个时钟}$$

$$\text{平均访存时间}_{\text{2路}} = 1.45 \text{ 个时钟}$$

可见，对于这两种Cache容量，伪相联Cache都是速度最快的。

方法五：硬件预取

- 指令和数据都可以预取
- 预取内容既可放入Cache，也可放在外缓冲器中。

  例如：指令流缓冲器
- 指令预取通常由Cache之外的硬件完成
  - 例如：Alpha AXP 21064微处理器在发生指令不命中时取两个块 - 被请求的指令块和顺序的下一指令块。被请求的指令块返回时放入Cache，而预取的指令块放入到缓冲器中。

- 预取效果
  - Joppi的研究结果
    - 指令预取 (4KB，直接映像Cache，块大小＝16字节)

      1个块的指令流缓冲器： 捕获15%～25%的不命中

      4个块的指令流缓冲器： 捕获50%

      16个块的指令流缓冲器：捕获72%
    - 数据预取 (4KB,直接映像Cache)

      1个数据流缓冲器，捕获25%的不命中

      还可以采用多个数据流缓冲器
  - Palacharla和Kessler的研究结果
    - 流缓冲器：既能预取指令又能预取数据

      对于两个64KB四路组相联Cache来说：

      8个流缓冲器能捕获50%～70%的不命中

- 预取应利用存储器的空闲带宽，不能影响对正常不命中的处理，否则可能会降低性能。

方法六：编译器控制的预取

主要思想：在编译时加入预取指令，在数据被用到之前发出预取请求。

假定：

(1) 使用一个容量为8 KB、块大小为16 B的直接映象Cache，它采用写回法并且按写分配。

(2) a、b分别为3×100（3行100列）和101×3的双精度浮点数组，每个元素都是8B。当程序开始执行时，这些数据都不在Cache内。

对于下面的程序：

```
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 ];
```

问：

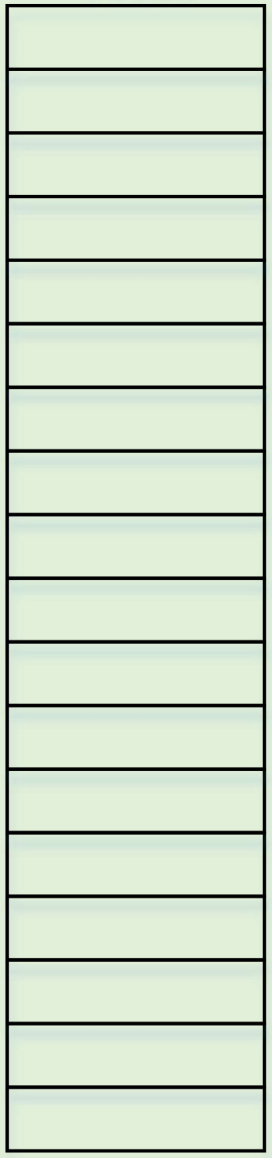
（1）首先，判断哪些访问可能会导致数据Cache失效；

（2）其次，加入预取指令以减少失效；

（3）最后，计算所执行的预取指令的条数以及通过预取避免的失效次数。

分析可能发生哪种失效

Cache
容量8KB
512块



数组a在
存储器中
占300*8B

a[0][0]
a[0][1]
a[0][2]
…
a[0][99]
a[1][0]
a[1][1]
a[1][2]
…
a[1][99]
a[2][0]
a[2][1]
a[2][2]
…
a[2][99]

数组b在
存储器中
占303*8B

b[0][0]
b[0][1]
b[0][2]
b[1][0]
b[1][1]
b[1][2]
b[2][0]
b[2][1]
b[2][2]
b[3][0]
b[3][1]
…
b[99][0]
b[99][1]
b[99][2]
b[100][0]
b[100][1]
b[100][2]

- 假设数组a和b在内存中**连续按行**存储；
- 1块16B，无论是a还是b，一块2个元素；
- 对于数组a，分布在150块中；
- 对于数组b，分布在152块中；
- a和b一共302块，少于Cache总块数，因此，不会发生容量失效；
- 直接映射，按照假定，a和b连续存储，不会发生冲突失效；
- 只能发生强制失效。

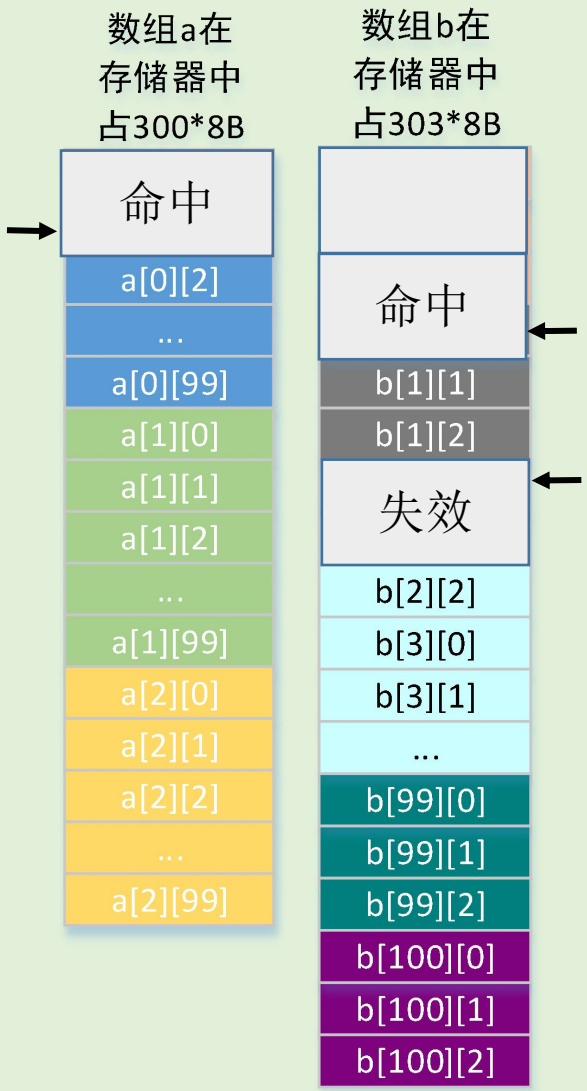
根据运算过程分析不命中和命中 a [i][j] = b[j][0] * b[j+1][0];

i=0;j=0

a [0][0] = b[0][0] * b[1][0];

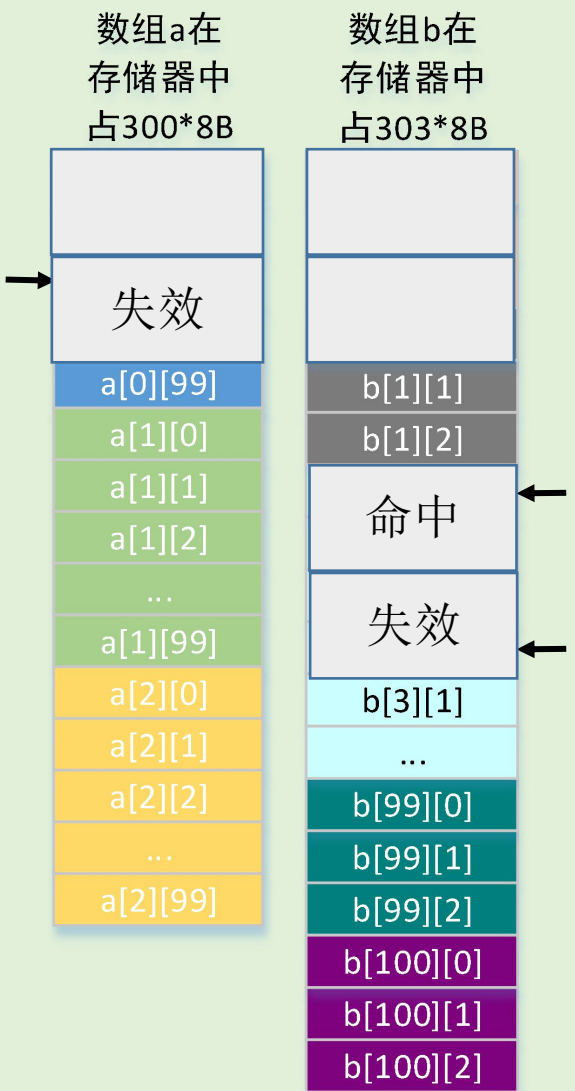
i=0;j=1

a [0][1] = b[1][0] * b[2][0];



i=0;j=2

a [0][2] = b[2][0] * b[3][0];



数组a的访问规律

a [0][0]　a [0][1]　a [0][2]　a [0][3]　等

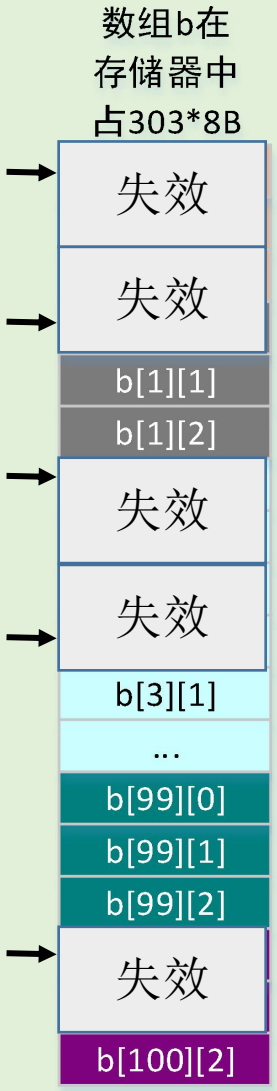
数组b的访问规律

b [0][0]　b [1][0]　b [2][0]　b [3][0]　…　b [100][0]

数组a：共150块，失效150次

数组b：i=0时，失效101次；i=1和2时，无失效

一共失效次数：150+101 = 251次



假设失效开销很大，预取必须至少提前7次循环进行。

```
for （ j = 0;  j < 100;  j = j+1 ）  {
        prefetch （ b[ j+7 ][ 0 ]）;
        /* 预取7次循环后所需的b （ j , 0 ） */
        prefetch （ a[ 0 ][ j+7 ]）;
        /* 预取7次循环后所需的a （0 , j ） */
        a [0][ j ] = b[ j ][ 0 ] * b[ j+1 ][ 0 ];
}
```

从 b [7][0] 开始预取

从 a [0][7] 开始预取

a [0][0] 至 a [0][6]可能产生数据访问失效

b [0][0] 至 b [6][0]可能产生数据访问失效

a失效 7/2 = 4； b失效7次

```
for （ i = 1;  i < 3;  i = i+1 ）  {
for （ j = 0;  j < 100;  j = j+1 ）
        prefetch （ a [ i ][ j+7 ]）;
        /* 预取7次循环后所需的a （ i , j ） */
        a [ i ][ j ] = b[ j ][ 0 ] * b[ j+1 ][ 0 ];
    } /* a失效 7/2 = 4, 两次循环共失效8次 */
```

与第一段类似，b已经无失效，不需要预取

一次循环a失效 7/2 = 4，两次循环共8次

总的失效次数＝4+7+4+4 = 19 次

- 按照预取数据所放的位置，可把预取分为两种类型：
  - 寄存器预取：把数据取到寄存器中。
  - Cache预取：只将数据取到Cache中。
- 按照预取的处理方式不同，可把预取分为：
  - 故障性预取：在预取时，若出现虚地址故障或违反保护权限，就会发生异常。
  - 非故障性预取：在遇到这种情况时则不会发生异常，因为这时它会放弃预取，转变为空操作。本节假定Cache预取都是非故障性的，也叫做非绑定预取。

- 在预取数据的同时，处理器应能继续执行。
  只有这样，预取才有意义。
  非阻塞Cache（非锁定Cache）
- 编译器控制预取的目的
  使执行指令和读取数据能重叠执行。
- 循环是预取优化的主要对象
  - 不命中开销小时：循环体展开1～2次
  - 不命中开销大时：循环体展开许多次
- 每次预取需要花费一条指令的开销
  - 保证这种开销不超过预取所带来的收益
  - 编译器可以通过把重点放在那些可能会导致不命中的访问上，使程序避免不必要的预取，从而较大程度地减少平均访存时间。

方法七：编译器优化

- 基本思想：通过对软件进行优化来降低不命中率。

  （特色：**无需对硬件做任何改动**）

- 程序代码和数据重组
  - 可以重新组织程序而不影响程序的正确性
    - 把一个程序中的过程重新排序，就可能会减少冲突不命中，从而降低指令不命中率。
      - McFarling研究了如何使用配置文件（profile）来进行这种优化。
    - 把基本块对齐，使得程序的入口点与Cache块的起始位置对齐，就可以减少顺序代码执行时所发生的Cache不命中的可能性。
  - 如果编译器知道一个分支指令很可能会成功转移，那么它就可以通过以下两步来改善**空间局部性**：
    - 将转移目标处的基本块和紧跟着该分支指令后的基本块进行对调；
    - 把该分支指令换为操作语义相反的分支指令。
  - 数据对存储位置的限制更少，更便于调整顺序。

这两点都是使得大概率执行的代码段放在分支失败处

- 编译优化技术包括

  （1）数组合并

  （2）内外循环交换

  （3）循环融合

  （4）分块

（1）数组合并

基本思想：将本来相互独立的多个数组合并成为一个复合数组，以提高访问它们的局部性。

举例：

```
/* 修改前 */
int x[100];
int y[100];
```

若x[i]和y[i]经常一起访问

```
/* 修改后 */
struct merge{
    int x;
    int y;
};
struct merge merged_array[100];
```

（2）内外循环交换

基本思想: 提高访问的局部性

举例：

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

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

（3）循环融合

基本思想：将若干个独立的循环融合为单个的循环。这些循环访问同样的数组，对相同的数据作不同的运算。这样能使得读入Cache的数据在被替换出去之前，能得到反复的使用 。

举例：

```
/* 修改前 */
for（j = 0; j < 100; j = j+1 ）
   x[i][j] = a[i][j] + b[i][j];
for（j = 0; j < 100; j = j+1 ）
   y[i][j] = a[i][j] - b[i][j];
```

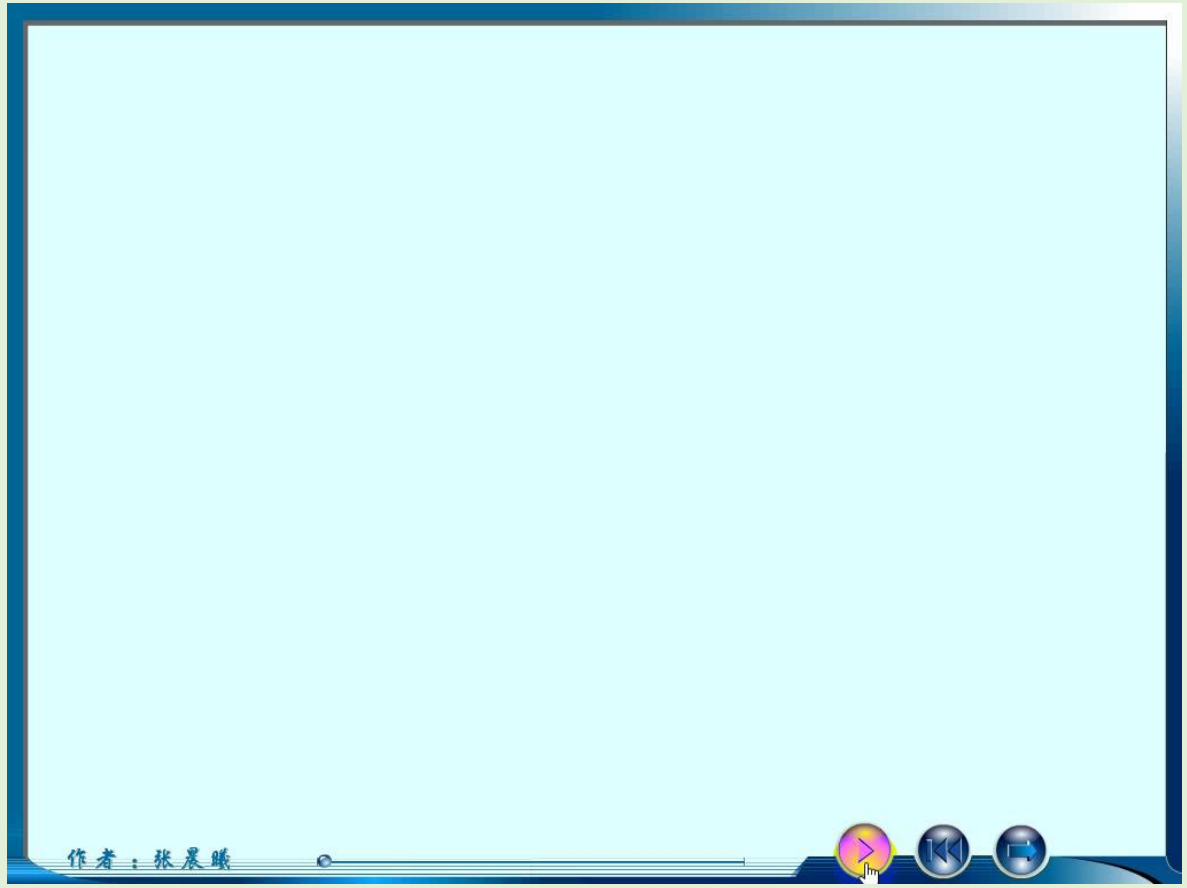
```
/* 修改后 */
for（j = 0; j < 100; j = j+1 ）{
   x[i][j] = a[i][j] + b[i][j];
   y[i][j] = a[i][j] - b[i][j];
}
```

（4）分块

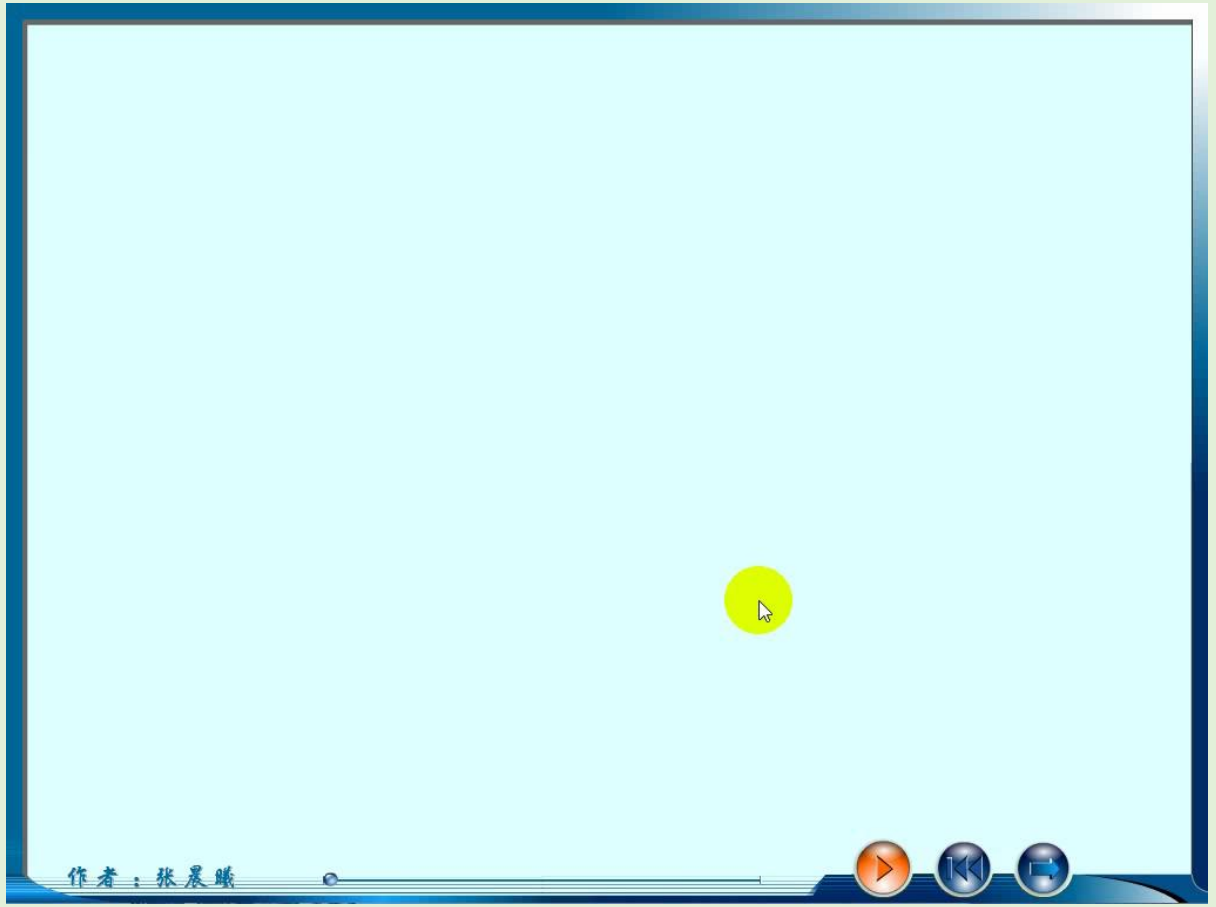
基本思想: 把对数组的整行或整列访问改为按块进行，尽量集中访问，减少替换，提高访问的局部性。

```
/* 修改前 */

    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;
    }
```

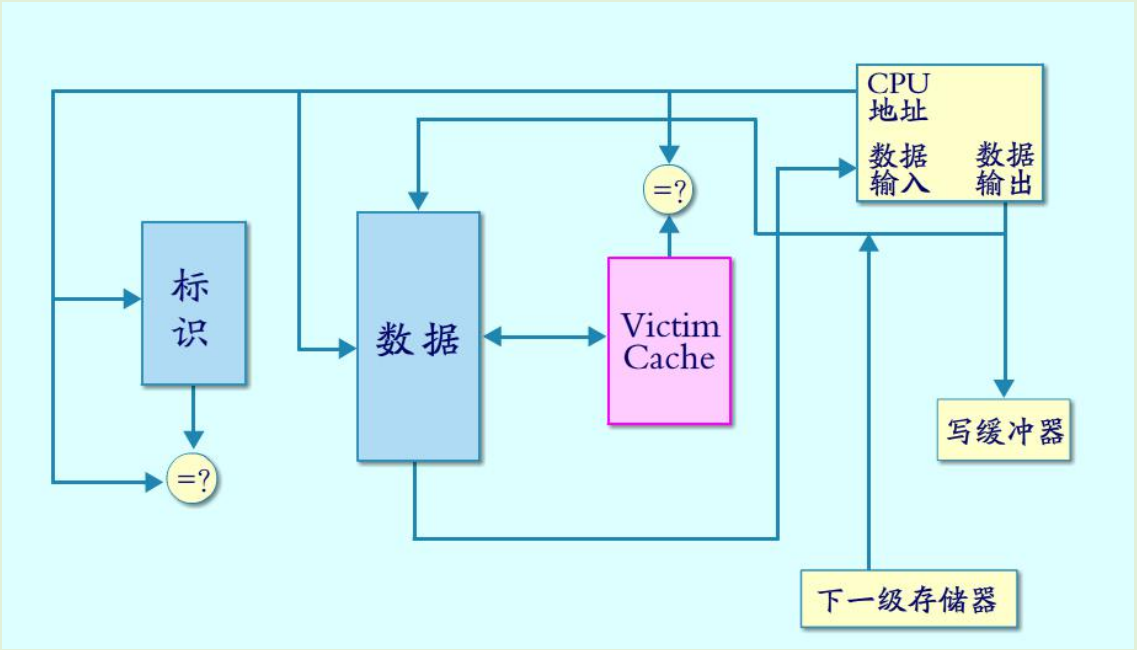


```
/* 修改后 */
  for （ jj = 0;  jj < N;  jj = jj+B ）
  for （ kk = 0;  kk < N;  kk = kk+B ）
  for （ i = 0;  i < N; i =i+1 ）
  for （ j = jj;  j < min（jj+B-1, N）;  j = j+1 ） {
     r = 0;
     for （ k = kk;  k < min（kk+B-1, N）;  k = k+1） {
       r = r + y[ i ][ k ] * z[ k ][ j ];
     }
      x[ i ][ j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



方法八：“牺牲”Cache

- 一种能减少冲突不命中次数而又不影响时钟频率的方法。
- 基本思想：在Cache和它从下一级存储器调数据的通路之间设置一个全相联的小Cache，称为“牺牲”Cache（Victim Cache）。用于存放被替换出去的块(称为牺牲者)，以备重用。
- 作用
  - 对于减小冲突不命中很有效，特别是对于小容量的直接映像数据Cache，作用尤其明显。
  - 例如项数为4的Victim Cache，能使4KB Cache的冲突不命中减少20%～90%



2.4 降低不命中开销

方法一：两级Cache

- 应把Cache做得更快？还是更大？

  答案：二者兼顾，再增加一级Cache

  - 第一级Cache(L1)小而快
  - 第二级Cache(L2)容量大

- 性能分析

  平均访存时间 ＝ 命中时间$_{L1}$＋不命中率$_{L1}$×不命中开销$_{L1}$

  不命中开销$_{L1}$ ＝ 命中时间$_{L2}$＋不命中率$_{L2}$×不命中开销$_{L2}$

  平均访存时间 ＝ 命中时间$_{L1}$＋不命中率$_{L1}$×（命中时间$_{L2}$＋不命中率$_{L2}$×不命中开销$_{L2}$）

- 局部不命中率与全局不命中率
  - 局部不命中率＝该级Cache的不命中次数/到达该级Cache的访问次数

    例如：上述式子中的不命中率L2
  - 全局不命中率＝该级Cache的不命中次数/CPU发出的访存的总次数
  - 全局不命中率L_2＝不命中率L_1×不命中率L_2
  - 全局不命中率比局部不命中率更有意义，它指出了在CPU发出的访存中，究竟有多大比例是穿过各级Cache，最终到达存储器的。
- 采用两级Cache时，每条指令的平均访存停顿时间：

  每条指令的平均访存停顿时间 ＝ 每条指令的平均不命中次数$_{L1}$×命中时间$_{L2}$

  ＋每条指令的平均不命中次数$_{L2}$×不命中开销$_{L2}$

例

考虑某一两级Cache：第一级Cache为L1，第二级Cache为L2。

（1）假设在1000次访存中，L1的不命中是40次，L2的不命中是20次。求各种局部不命中率和全局不命中率。

（2）假设L2的命中时间是10个时钟周期，L2的不命中开销是100时钟周期，L1的命中时间是1个时钟周期，平均每条指令访存1.5次，不考虑写操作的影响。问：平均访存时间是多少？每条指令的平均停顿时间是多少个时钟周期？

解 （1）第一级Cache的不命中率（全局和局部）是40/1000，即4%；

第二级Cache的局部不命中率是20/40，即50%；

第二级Cache的全局不命中率是20/1000，即2%。

（2）平均访存时间＝命中时间$_{L1}$＋不命中率$_{L1}$×（命中时间$_{L2}$＋不命中率$_{L2}$×不命中开销$_{L2}$）

＝1＋4%×（10＋50%×100）

＝1＋4%×60＝3.4个时钟周期

由于平均每条指令访存1.5次，且每次访存的平均停顿时间为：

3.4－1.0＝2.4

所以：

每条指令的平均停顿时间＝2.4×1.5＝3.6个时钟周期

- 对于第二级Cache，我们有以下结论：
  - 在第二级Cache比第一级 Cache大得多的情况下，两级Cache的全局不命中率和容量与第二级Cache相同的单级Cache的不命中率非常接近。
  - 局部不命中率不是衡量第二级Cache的一个好指标，因此，在评价第二级Cache时，应用全局不命中率这个指标。
- 第二级Cache不会影响CPU的时钟频率，因此其设计有更大的考虑空间。
  - 设计第二级Cache时有两个问题需要权衡：
    - 它能否降低CPI中的平均访存时间部分？
    - 它的成本是多少？
- 第二级Cache的参数
  - 容量：第二级Cache的容量一般比第一级的大许多。
    - 大容量意味着第二级Cache可能实际上没有容量不命中，只剩下一些强制性不命中和冲突不命中。
  - 相联度：第二级Cache可采用较高的相联度或伪相联方法。

例

给出有关第二级Cache的以下数据：

（1）对于直接映像，命中时间$_{L2}$ ＝ 10个时钟周期

（2）两路组相联使命中时间增加0.1个时钟周期，即为10.1个时钟周期。

（3） 对于直接映像，局部不命中率$_{L2}$ ＝ 25%

（4） 对于两路组相联，局部不命中率$_{L2}$ ＝ 20%

（5） 不命中开销$_{L2}$ ＝ 50个时钟周期

试问第二级Cache的相联度对不命中开销的影响如何？

解

对一个直接映像的第二级Cache来说，第一级Cache的不命中开销为：

$$\text{不命中开销}_{\text{直接映像，L1}} = 10 + 25\% \times 50 = 22.5 \text{ 个时钟周期}$$

对于两路组相联第二级Cache来说，命中时间增加了10%（0.1）个时钟周期，故第一级Cache的不命中开销为：

$$\text{不命中开销}_{\text{两路组相联，L1}} = 10.1 + 20\% \times 50 = 20.1 \text{ 个时钟周期}$$

把第二级Cache的命中时间取整，得10或11，则：

$$\text{不命中开销}_{\text{两路组相联，L1}} = 10 + 20\% \times 50 = 20.0 \text{ 个时钟周期}$$

$$\text{不命中开销}_{\text{两路组相联，L1}} = 11 + 20\% \times 50 = 21.0 \text{ 个时钟周期}$$

故对于第二级Cache来说，两路组相联优于直接映像。

- 块大小
  - 第二级Cache可采用较大的块，如 64、128、256字节
  - 为减少平均访存时间，可以让容量较小的第一级Cache采用较小的块，而让容量较大的第二级Cache采用较大的块。
  - 多级包容性

    需要考虑的另一个问题：

    第一级Cache中的数据是否总是同时存在于第二级Cache中。如果是，就说第二级Cache是具有多级包容性的。

方法二：让读不命中优先于写

- Cache中的写缓冲器导致对存储器访问的复杂化

  在读不命中时，所读单元的最新值有可能还在写缓冲器中，尚未写入主存。

- 解决问题的方法(读不命中的处理)
  - 推迟对读不命中的处理，直到写缓冲器清空

    （缺点：读不命中的开销增加）
  - 检查写缓冲器中的内容，若无相同，且存储器可用，继续处理读不命中（常用方案）
- 在写回法Cache中，也可采用写缓冲器。

方法三：写缓冲合并

- 提高写缓冲器的效率
- 写直达Cache依靠写缓冲来减少对下一级存储器写操作的时间。
  - 如果写缓冲器为空，就把数据和相应地址写入该缓冲器。从CPU的角度来看，该写操作就算是完成了。
  - 如果写缓冲器中已经有了待写入的数据，就要把这次的写入地址与写缓冲器中已有的所有地址进行比较，看是否有匹配的项。如果有地址匹配而对应的位置又是空闲的，就把这次要写入的数据与该项合并。这就叫写缓冲合并。
  - 如果写缓冲器满且又没有能进行写合并的项，就必须等待。
- 作用
  - 连续写入多个字的速度快于每次只写入一个字的操作；
  - 提高了写缓冲器的空间利用率，减少因写缓冲器满而要进行的等待时间。

例如　写缓冲区有4项，每项4个64位的字。

方法四：请求字处理技术

- 请求字

  从下一级存储器调入Cache的块中，只有一个字是立即需要的。这个字称为请求字。

- 应尽早把请求字发送给CPU
  - 尽早重启动：请求字没有到达时，CPU处于等待状态。一旦请求字到达，立即送给CPU，让等待的CPU尽早重启动，继续执行。
  - 请求字优先：调块时，让存储器首先提供CPU所要的请求字。请求字一旦到达，就立即送给CPU，让CPU继续执行，同时从存储器调入该块的其余部分。
- 在以下情况下，用不用差别不大
  - Cache块较小
  - 下一条指令正好访问同一Cache块的另一部分

方法五：非阻塞Cache技术

- 非阻塞Cache：采用记分牌或者Tomasulo类控制方式，允许指令乱序执行，CPU无需在Cache不命中时停顿；
- 允许多次不命中，进一步提高性能；
- 可以同时处理的不命中次数越多，所能带来的性能上的提高就越大。

问题：不命中次数越多越好？

- 模拟研究

  数据Cache的平均存储器等待时间（以周期为单位）与阻塞Cache平均存储器等待时间的比值。

  - 测试条件：8K直接映像Cache，块大小为32字节
  - 测试程序：SPEC92（14个浮点程序，4个整数程序）
  - 结果表明：在重叠不命中个数为1、2和64的情况下
    - 浮点程序的平均比值分别为：76%、51%和39%
    - 整数程序的平均比值则分别为：81%、78%和78%

对于整数程序来说，重叠次数对性能提高影响不大，简单的“一次不命中下命中”就几乎可以得到所有的好处。

2.5 减少命中时间

命中时间直接影响到处理器的时钟频率。在当今的许多计算机中，往往是Cache的访问时间限制了处理器的时钟频率。

方法一：使用小容量、结构简单的Cache

- 硬件越简单，速度就越快；
- 应使Cache足够小，以便可以与CPU一起放在同一块芯片上。
- 某些设计采用了一种折衷方案：

  把Cache的标识放在片内，而把Cache的数据存储器放在片外。
- 采用结构简单的Cache，比如直接映像Cache。

方法二：虚拟Cache

- 物理Cache
  - 使用物理地址进行访问的传统Cache。
  - 标识存储器中存放的是物理地址，进行地址检测也是用物理地址。
  - 缺点：地址转换和访问Cache串行进行，访问速度很慢。

物理Cache存储系统

- 虚拟Cache
  - 可以直接用虚拟地址进行访问的Cache。标识存储器中存放的是虚拟地址，进行地址检测用的也是虚拟地址。
  - 优点：在命中时不需要地址转换，省去了地址转换的时间。即使不命中，地址转换和访问Cache也是并行进行的，其速度比物理Cache快很多。

- 并非都采用虚拟Cache
  - 清空问题：由于新进程的虚拟地址可能与原进程相同；
  - 解决方法：在地址标识中增加PID(进程标识符)字段；
  - PID相关问题：为了减少位数，PID可能循环使用，所以也可能存在多个进程使用同一个PID，所以此时也需要清空Cache。

- 虚拟索引＋物理标识
  - 原理：使用虚地址中的页内位移生成Cache索引；虚实转换后的实页地址作为标志tag
  - 优点：兼得虚拟Cache和物理Cache的好处
  - 局限性：Cache容量受到限制

    Cache容量≤页大小×相联度
- 举例：IBM3033的Cache
  - 页大小＝4KB　　相联度＝16

Cache容量＝16×4KB＝64KB

方法三：Cache访问流水化

- 对第一级Cache的访问按流水方式组织
- 访问Cache需要多个时钟周期才可以完成
- 例如
  - Pentium访问指令Cache需要一个时钟周期
  - Pentium Pro到Pentium Ⅲ需要两个时钟周期
  - Pentium 4 则需要4个时钟周期

方法四：踪迹Cache

- 开发指令级并行性所遇到的一个挑战是：

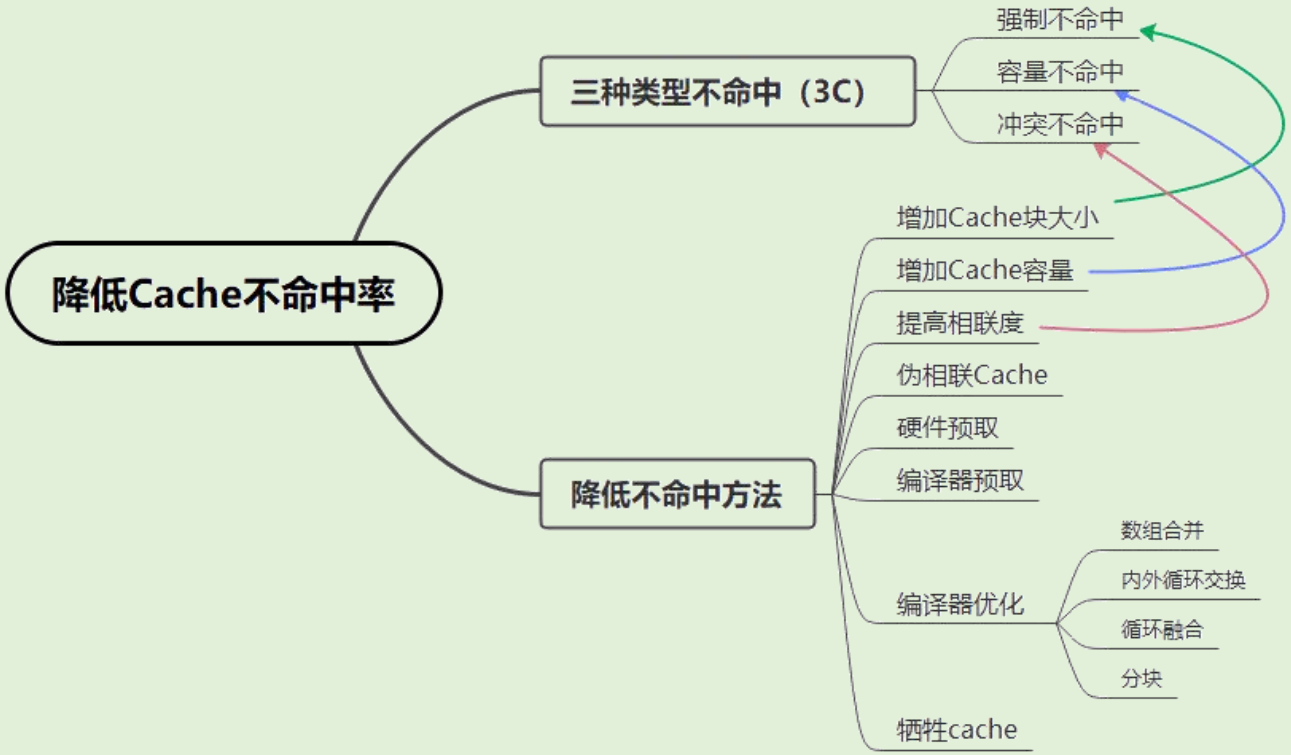
  当要每个时钟周期流出超过4条指令时，要提供足够多条彼此互不相关的指令是很困难的。

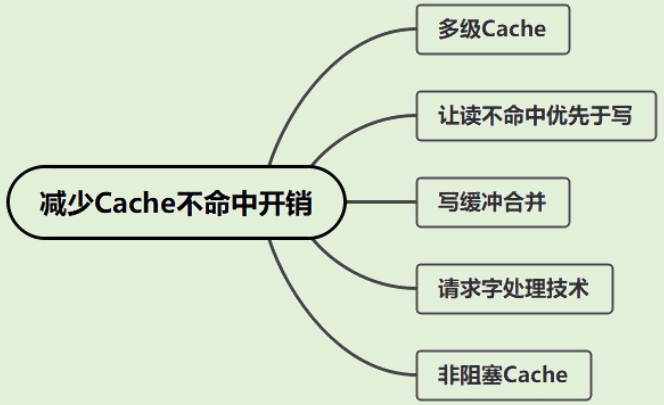
- 一个解决方法：采用踪迹 Cache

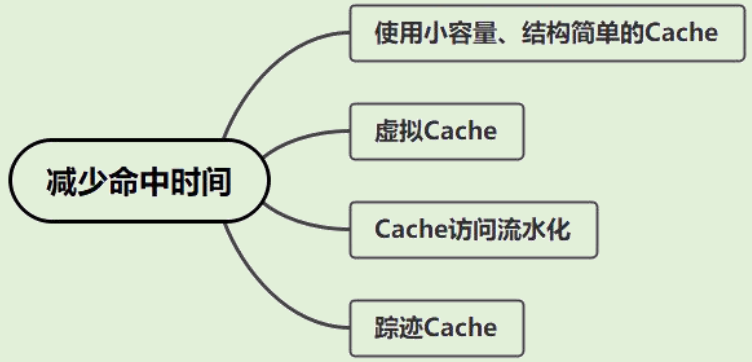
  存放CPU所执行的动态指令序列，包含了由分支预测展开的指令，该分支预测是否正确需要在取到该指令时进行确认。

- 优缺点
  - 地址映像机制复杂；
  - 相同的指令序列有可能被当作条件分支的不同选择而重复存放；
  - 能够提高指令Cache的空间利用率。

2.6 Cache优化技术总结







2.6 Cache优化技术总结

- Cache优化技术总结
  - “＋”号：表示改进了相应指标。
  - “－”号：表示它使该指标变差。
  - 空格栏：表示它对该指标无影响。
  - 复杂性：0表示最容易，3表示最复杂。

Cache优化技术总结

优化技术	不命中率	不命中开销	命中时间	硬件复杂度	说 明
增加块大小	+	–		0	实现容易；Pentium 4 的第二级Cache采用了128字节的块
增加Cache容量	+			1	被广泛采用，特别是第二级Cache
提高相联度	+		–	1	被广泛采用
牺牲Cache	+			2	AMD Athlon采用了8个项的Victim Cache
伪相联Cache	+			2	MIPS R10000的第二级Cache采用
硬件预取指令和数据	+			2～3	许多机器预取指令，UltraSPARC III预取数据

优化技术	不命中率	不命中开销	命中时间	硬件复杂度	说 明
编译器控制的预取	+			3	需同时采用非阻塞Cache；有几种微处理器提供了对这种预取的支持
用编译技术减少Cache不命中次数	+			0	向软件提出了新要求；有些机器提供了编译器选项
使读不命中优先于写		+	–	1	在单处理机上实现容易，被广泛采用
写缓冲归并		+		1	与写直达合用，广泛应用，例如21164，UltraSPARC III
尽早重启动和关键字优先		+		2	被广泛采用
非阻塞Cache		+		3	在支持乱序执行的CPU中使用

优化技术	不命中率	不命中开销	命中时间	硬件复杂度	说 明
两级Cache			+	2	硬件代价大；两级Cache的块大小不同时实现困难；被广泛采用
容量小且结构简单的Cache	–		+	0	实现容易，被广泛采用
对Cache进行索引时不必进行地址变换			+	2	对于小容量Cache来说实现容易，已被Alpha 21164和UltraSPARC III采用
流水化Cache访问			+	1	被广泛采用
Trace Cache			+	3	Pentium 4 采用