计算机组织与体系结构

第二十一讲

计算机科学与技术学院

张展

第八章　存储层次

8.2.6 性能分析

- **不命中率（缺失率）**
  - 与硬件速度无关
  - 容易产生一些误导
  - 平均访存时间比不命中率更好
- **平均访存时间**

平均访存时间 = 命中时间＋不命中率×不命中开销

- 可以采用绝对时间——如一次命中需0.25-1.0纳秒
- 或是用CPU等待存储器的时钟周期数——如缺失代价用75-100个时钟周期表示
- 仍旧不能代替执行时间

8.2.6 性能分析

CPU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

存储器停顿周期数＝访存次数×失效率×失效开销

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

CPU时间＝IC×[CPIexe＋访存次数/指令数×
失效率×失效开销]×时钟周期时间

CPU时间＝IC×[CPIexe＋每条指令的平均存储器停顿周期数]×时钟周期时间

例8.3 考虑两种不同组织结构的Cache：直接映象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直接映象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：

平均访存时间＝命中时间＋不命中率×不命中开销

因此，两种结构的平均访存时间分别是：

平均访存时间1路＝2.0＋（0.014×70）＝2.98ns

平均访存时间2路＝2.0×1.10＋（0.010×70）＝2.90ns

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：

CPU时间1路 ＝ IC×(2.0×2＋(1.3×0.014×70))

＝ 5.27×IC

CPU时间2路 ＝ IC×(2.0×2×1.10＋(1.3×0.010×70))

＝ 5.31×IC

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

与平均存储器存取时间的比较结果相反，直接映象Cache的平均性能好一些。

2-路组相联情况下，尽管它的缺失率低一些，但是所有指令时钟周期都延长了。

8.2.7 改进Cache性能

平均访存时间＝命中时间＋失效率×失效开销

可以从三个方面改进Cache的性能：

(1) 降低失效率

(2) 减少失效开销

(3) 减少Cache命中时间

15种Cache优化技术

共7种

增加块大小

提高相联度

Victim Cache

伪相联 Cache

硬件预取

编译器控制的预取

用编译器技术减少Cache失效

降低失效率

共5种

使读失效优于写

子块放置技术

尽早重启动和关键字优先

非阻塞Cache

第二级Cache

减少失效开销

共3种

容量小且结构简单的Cache

对Cache索引时，不必进行地址变换

流水化写

减少命中时间

本章内容

8.1 存储器的层次结构

8.2 Cache基本知识

8.3 降低Cache失效率的方法

8.4 减少Cache失效开销

8.5 减少命中时间

8.6 主存

8.7 虚拟存储器

8.3 降低Cache失效率的方法

1. 三种失效(3C)

(1) 强制性失效(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，这就是强制性失效

(冷启动失效，首次访问失效)

(2) 容量失效(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，就会发生失效。这种失效称为容量失效。

(3) 冲突失效(Conflict miss)

在组相联或直接映象Cache中，若太多的块映象到同一组(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。这就是发生了冲突失效。

(碰撞失效，干扰失效)

2. 三种失效所占的比例

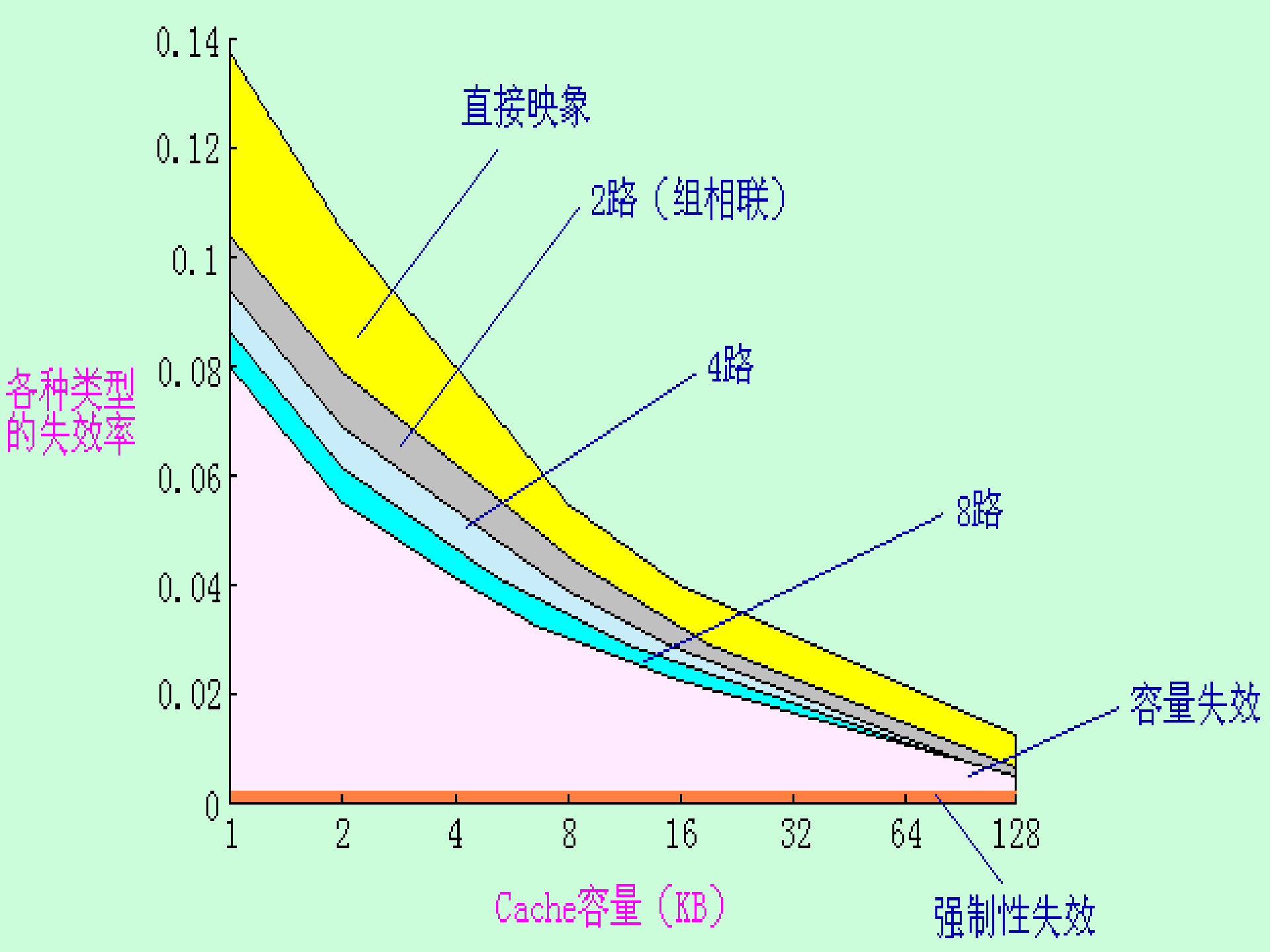
(SPEC92)　图示I(绝对值)

可以看出:

(1) 相联度越高，冲突失效就越少；

(2) 强制性失效不受Cache容量的影响，但容量失效却随着容量的增加而减少；强制性失效和容量失效不受相联度的影响.

(3) 表中的数据符合2:1的Cache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Cache的失效率。


0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
1
2
4
8
16
32
64
128
各种类型的失效率
Cache容量（KB）
直接映象
2路（组相联）
4路
8路
容量失效
强制性失效

3. 减少三种失效的方法

- 强制性失效：增加块大小，预取
  (本身很少)
- 容量失效：增加容量
  (防止出现抖动现象)
- 冲突失效：提高相联度
  (理想情况：全相联)

4. 许多降低失效率的方法会增加命中时间或失效开销

8.3.1 增加Cache块大小

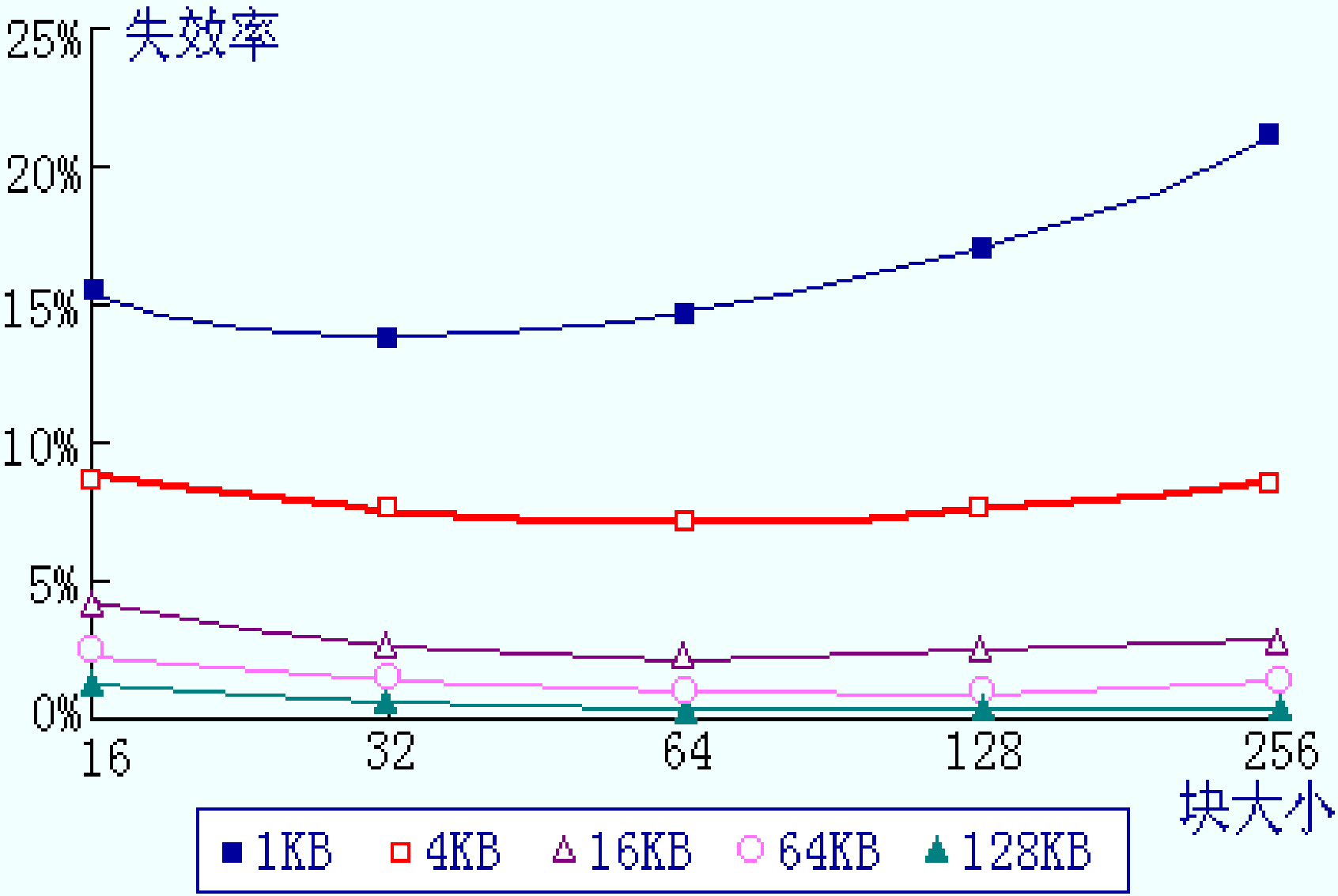
1. 失效率与块大小的关系

(1) 对于给定的Cache容量，当块大小增加失效率开始时下降，后来反而上升

(2) Cache容量越大，失效率达到最低的块大小就越大

2. 增加块大小会增加失效开销

Cache失效率与块大小的关系



块大小(字节)	Cache容量				
	1KB	4KB	16KB	64KB	256KB
16	15.05%	8.57%	3.94%	2.04%	1.09%
32	13.34%	7.24%	2.87%	1.35%	0.70%
64	13.76%	7.00%	2.64%	1.06%	0.51%
128	16.64%	7.78%	2.77%	1.02%	0.49%
256	22.01%	9.51%	3.29%	1.15%	0.49%

例 8.4

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个时钟周期，可提供32个字节；依此类推。试问：对于表5-6中列出的各种容量的Cache，在块大小分别为多少时，平均访存时间最小？假设命中时间为一个时钟周期

解:

解题过程

1KB、4KB、16KB Cache: 块大小＝32字节

64KB、256KB Cache: 块大小＝64字节

平均访问时间 = 1 + 失效率 × 失效开销

块大小为16时： 平均访存时间 = 1 + 失效率×42

= 1 + 15.05%×42

= 7.321

块大小（字节）	Cache容量				
	1KB	4KB	16KB	64KB	256KB
16	15.05%	8.57%	3.94%	2.04%	1.09%
32	13.34%	7.24%	2.87%	1.35%	0.70%
64	13.76%	7.00%	2.64%	1.06%	0.51%
128	16.64%	7.78%	2.77%	1.02%	0.49%
256	22.01%	9.51%	3.29%	1.15%	0.49%

块大小（字节）	失效开销（时钟周期）	Cache容量				
		1KB	4KB	16KB	64KB	256KB
16	42	7.321				
32	44					
64	48					
128	56					
256	72					

平均访存时间

块大小（字节）	失效开销（时钟周期）	Cache容量				
		1KB	4KB	16KB	64KB	256KB
16	42	7.321	4.599	2.655	1.857	1.458
32	44	**6.870**	**4.186**	**2.263**	1.594	1.308
64	48	7.605	4.360	2.267	**1.509**	**1.245**
128	56	10.318	5.357	2.551	1.571	1.274
256	72	16.847	7.847	3.369	1.828	1.353

8.3.2 提高相联度

1. 采用相联度超过8的方法实际意义不大
2. 2:1 Cache经验规则

   失效率：容量为*N* 的直接映象Cache

   ≈容量为*N*/2的两路组相联Cache
3. 提高相联度是以增加命中时间为代价

   例如：

   TTL或ECL板级Cache，两路组相联：增加10％

   定制的CMOS Cache, 两路组相联：增加2％

例 8.5

假定提高相联度会按下列比例增大处理器时钟周期：

$$\text{时钟周期}_{2\text{路}} = 1.10 \times \text{时钟周期}_{1\text{路}}$$

$$\text{时钟周期}_{4\text{路}} = 1.12 \times \text{时钟周期}_{1\text{路}}$$

$$\text{时钟周期}_{8\text{路}} = 1.14 \times \text{时钟周期}_{1\text{路}}$$

假定命中时间为1个时钟，直接映象情况下失效开销为50个时钟周期，而且假设不必将失效开销取整。使用教材（张晨曦教材）表5-3中的失效率，试问当Cache为多大时，以下不等式成立?

$$平均访存时间_{8路} < 平均访存时间_{4路}$$
$$平均访存时间_{4路} < 平均访存时间_{2路}$$
$$平均访存时间_{2路} < 平均访存时间_{1路}$$

解：

在各种相联度的情况下，平均访存时间分别为：

$$平均访存时间_{8路} = 命中时间_{8路} + 失效率_{8路} \times 失效开销_{8路}$$
$$= 1.14 + 失效率_{8路} \times 50$$

$$平均访存时间_{4路} = 1.12 + 失效率_{4路} \times 50$$

$$平均访存时间_{2路} = 1.10 + 失效率_{2路} \times 50$$

$$平均访存时间_{1路} = 1.00 + 失效率_{1路} \times 50$$

在每种情况下的失效开销相同，都是50个时钟周期。
把相应的失效率代入上式，即可得平均访存时间。

例如，1KB的直接映象Cache的平均访存时间为：

$$\text{平均访存时间}_{1\text{路}} = 1.00 + (0.133 \times 50)$$
$$= 7.65$$

容量为128KB的8路组相联Cache的平均访存时间为：

$$\text{平均访存时间}_{8\text{路}} = 1.14 + (0.006 \times 50)$$
$$= 1.44$$

Cache 容量 (KB)	相联度（路）			
	1	2	4	8
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

8.3.3 Victim Cache

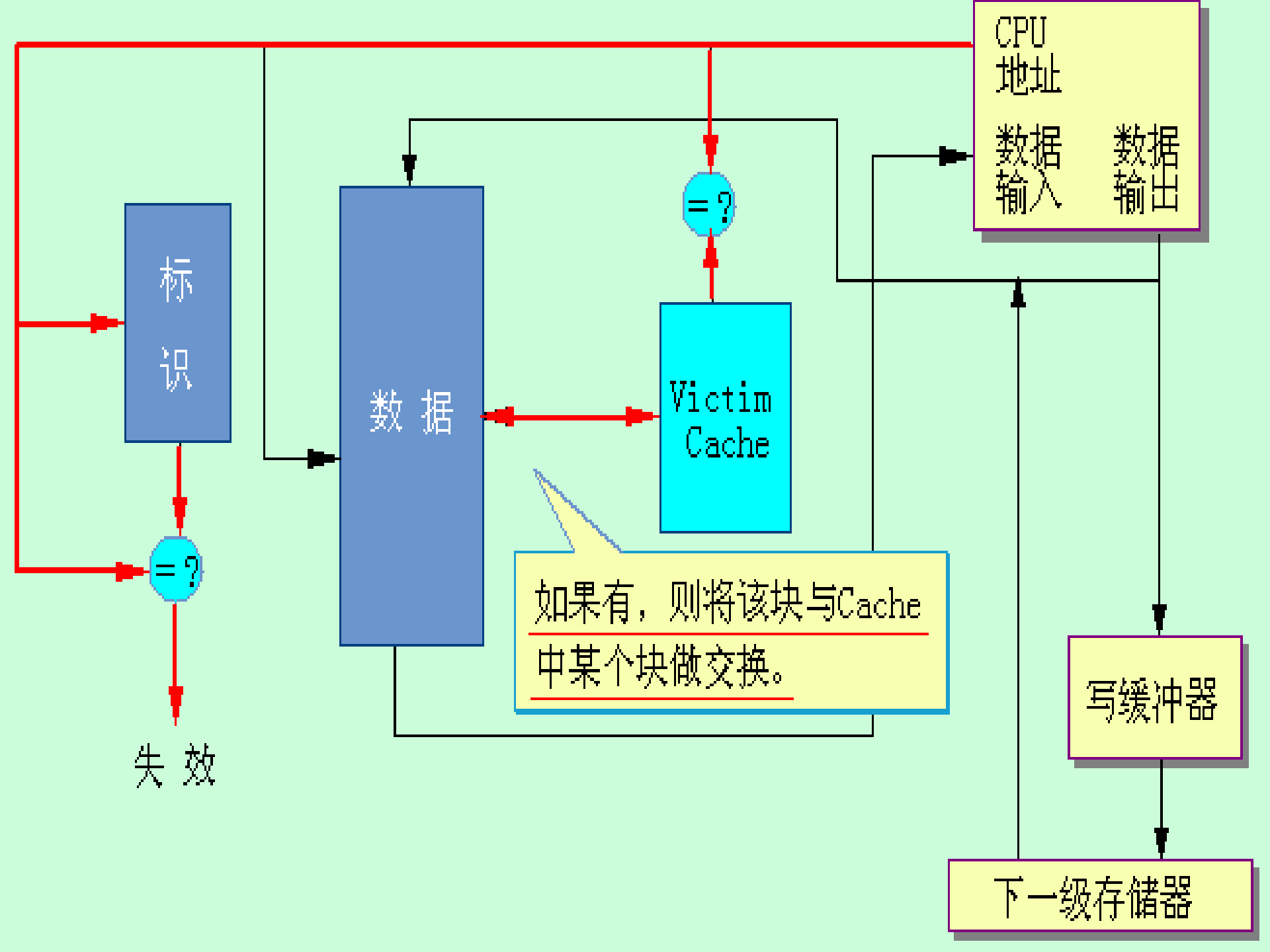
1. 基本思想

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，用于存放被替换出去的块(称为Victim)，以备重用。

工作过程

2. 作用

对于减小冲突失效很有效，特别是对于小容量的直接映象数据Cache，作用尤其明显。例如，项数为4的Victim Cache: 使4KB Cache的冲突失效减少20%～90%


CPU
地址
数据
输入
数据
输出
标
识
=?
失 效
数 据
=?
Victim
Cache
如果有，则将该块与Cache
中某个块做交换。
写缓冲器
下一级存储器

2. 伪相联Cache

取直接映象及组相联两者的优点：

命中时间小，失效率低

缺点：

多种命中时间会使CPU流水线的设计复杂化

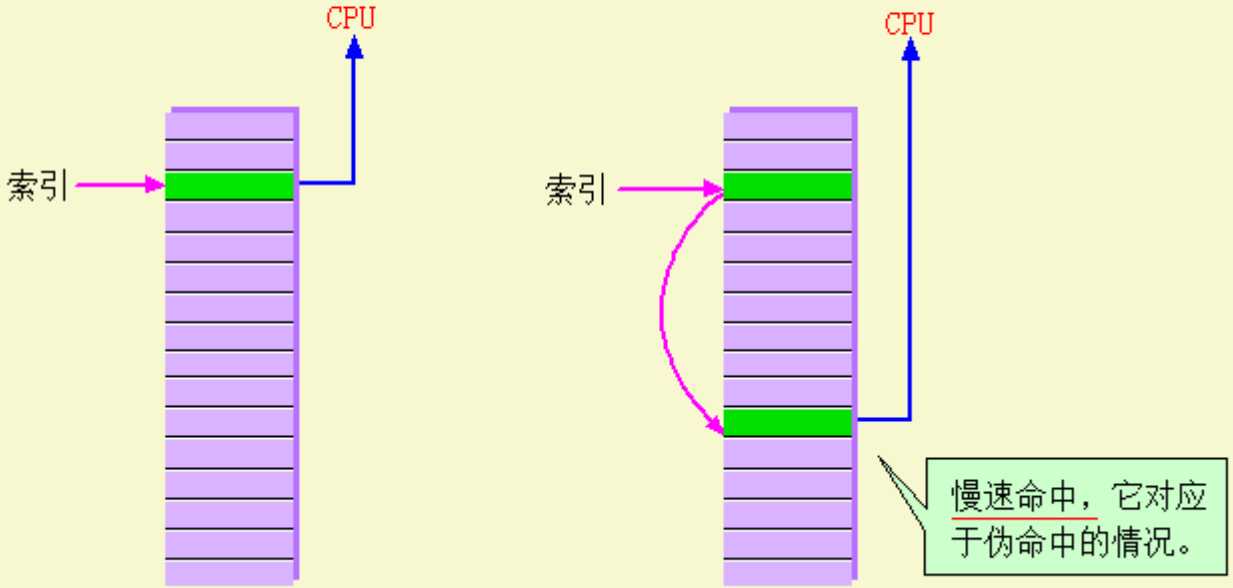
(1) 基本思想及工作原理

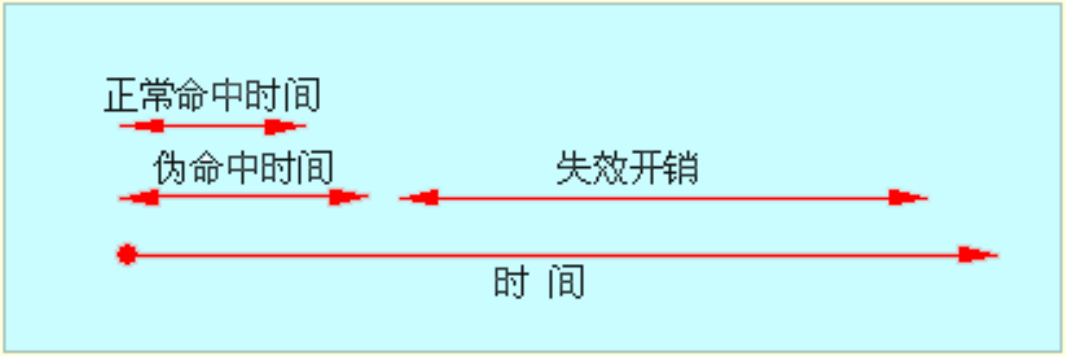
在逻辑上把直接映象Cache的空间上下平分为两个区。对于任何一次访问，伪相联Cache先按直接映象Cache的方式去处理。若命中，则其访问过程与直接映象Cache的情况一样。若不命中，则再到另一区相应的位置去查找。若找到，则发生了伪命中，否则就只好访问下一级存储器。

(2) 快速命中与慢速命中

要保证绝大多数命中都是快速命中。

快速命中与慢速命中





8.3.4 硬件预取技术

1. 指令和数据都可以预取

2. 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

3. 预取效果

   (1) Joppi的研究结果

   - 指令预取：(4KB，直接映象Cache,块大小＝16字节)

     1个块的指令流缓冲器： 捕获15％～25％的失效

     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,直接映象Cache)1个数据流缓冲器：捕获25％的失效,还可以采用多个数据流缓冲器

(2) Palacharla和Kessler的研究结果

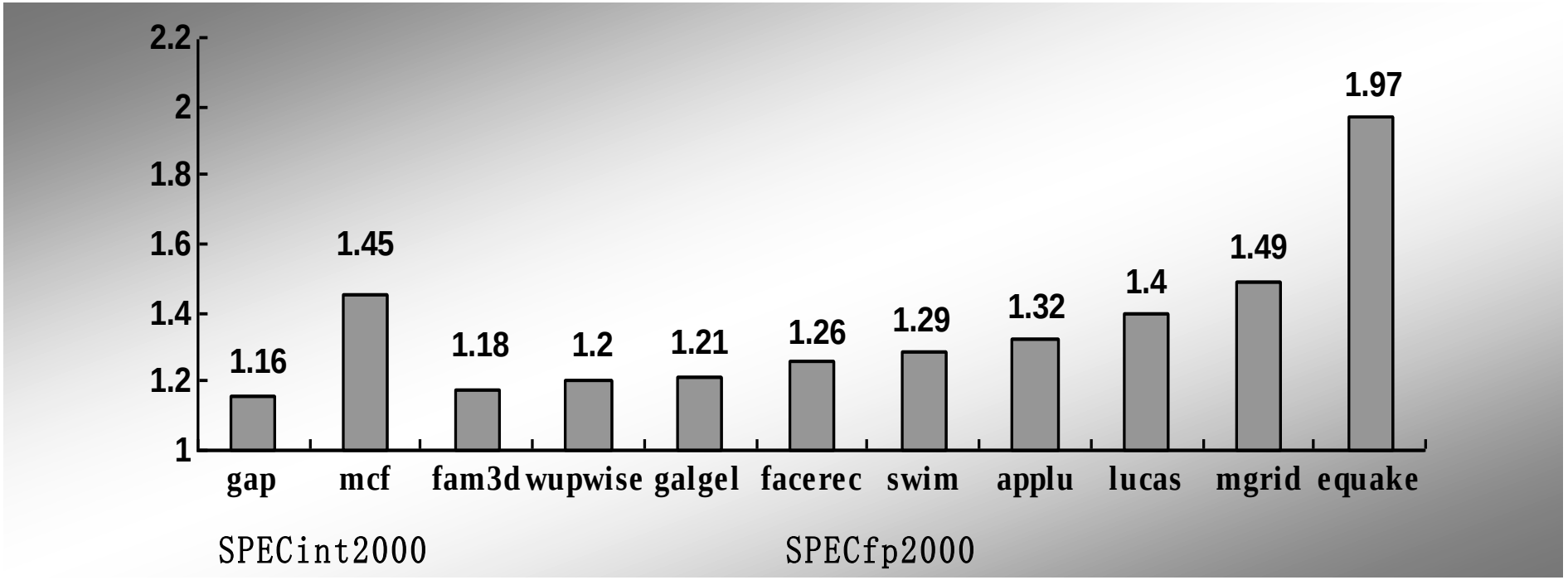
流缓冲器：既能预取指令又能预取数据

对于两个64KB四路组相联Cache来说：

8个流缓冲器能捕获50％～70％的失效。

（3）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。

- 一组利用硬件预取后整体性能得到较大提高的SPEC2000程序的性能增益
  （前2个为定点程序，其余为10个浮点程序）。



8.3.5 由编译器控制的预取

由编译器加入预取指令，在数据被用到之前发出预取请求。

1. 预取的类型

- 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
- 故障性预取：预取时，若出现虚地址故障或违反访问权限，就会发生异常。
- 非故障性预取：预取时，若出现虚地址故障或违反访问权限，并不会导致异常，只是转变为“不预取”。

2. 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)

3. 循环是预取优化的主要对象

失效开销小时：循环体展开1～2次

失效开销大时：循环体展开许多次

4. 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。

5. 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。

例8.7　对于下面的程序，判断哪些访问可能会导致数据Cache失效。然后，加入预取指令以减少失效。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。假定：

(1) 我们用的是一个容量为8KB、块大小为16B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8个字节。当程序开始执行时，这些数据都不在Cache内。

```
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];
```

解:

(1) 计算过程

(2) 失效情况

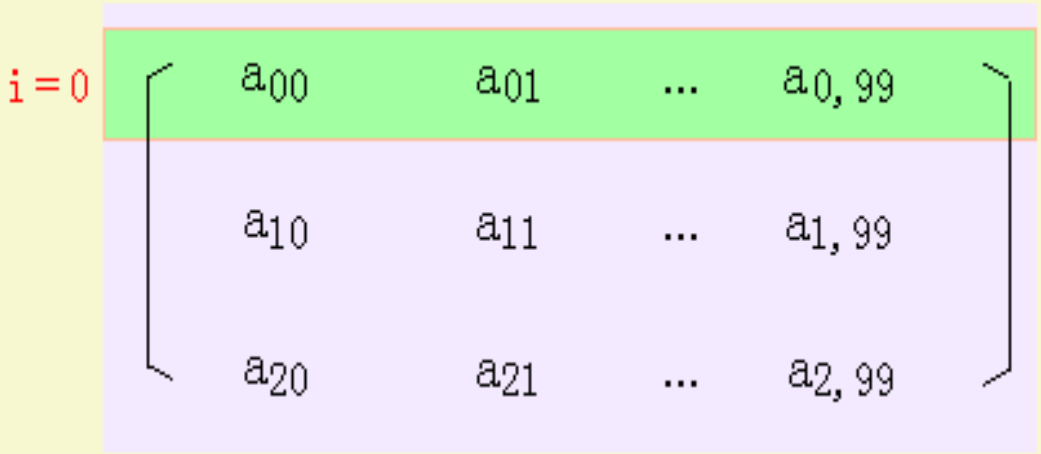
总的失效次数 = 251次

(3) 改进后的程序

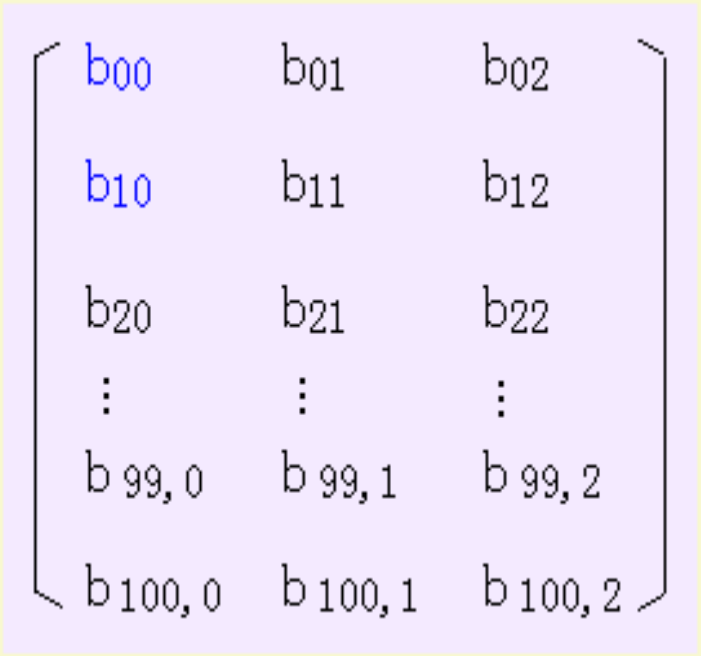
失效情况

总的失效次数 = 19次

$b_{00} * b_{10}$



a 数组



b 数组

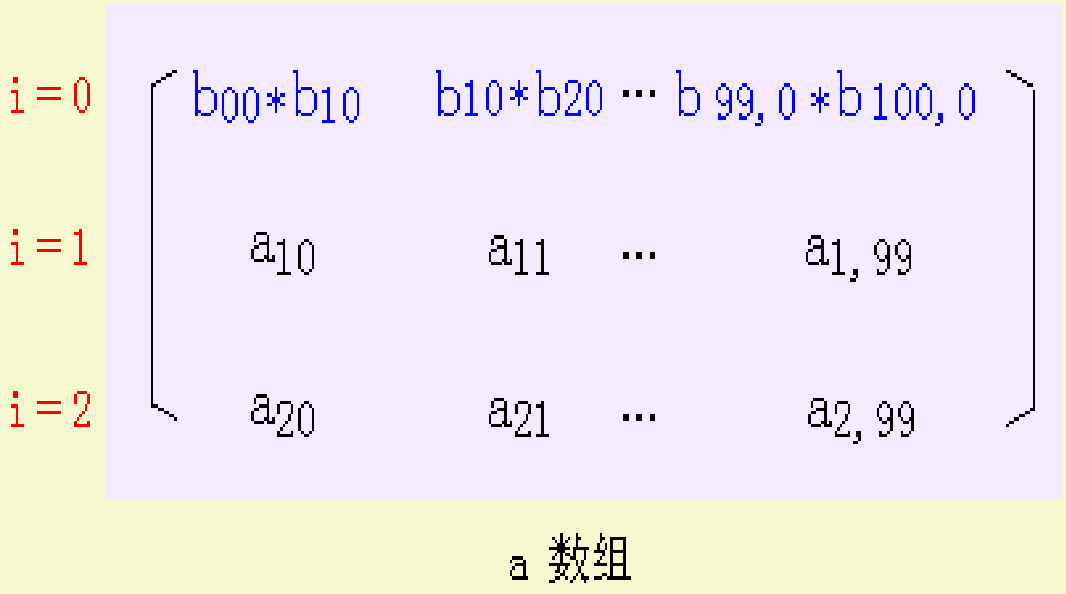
i=0

$$\begin{pmatrix} b_{00}*b_{10} & b_{10}*b_{20} & \cdots & a_{0,99} \\ a_{10} & a_{11} & \cdots & a_{1,99} \\ a_{20} & a_{21} & \cdots & a_{2,99} \end{pmatrix}$$

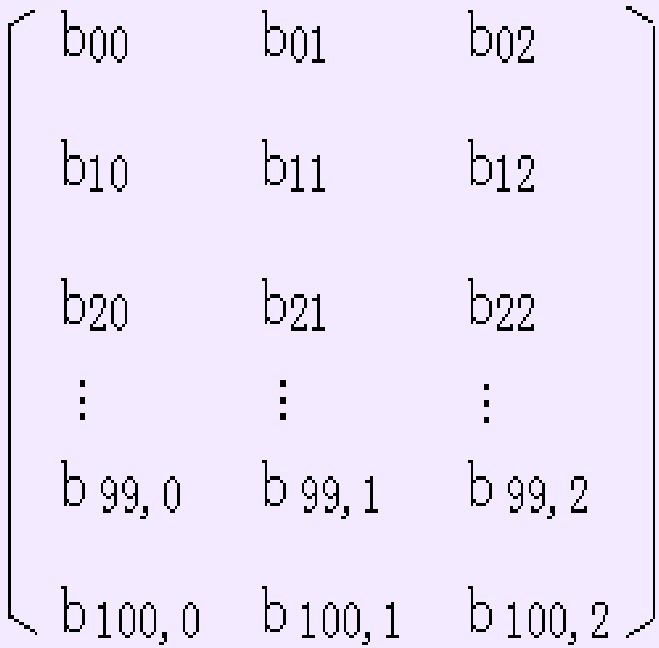
a 数组

$$\begin{pmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \\ \vdots & \vdots & \vdots \\ b_{99,0} & b_{99,1} & b_{99,2} \\ b_{100,0} & b_{100,1} & b_{100,2} \end{pmatrix}$$

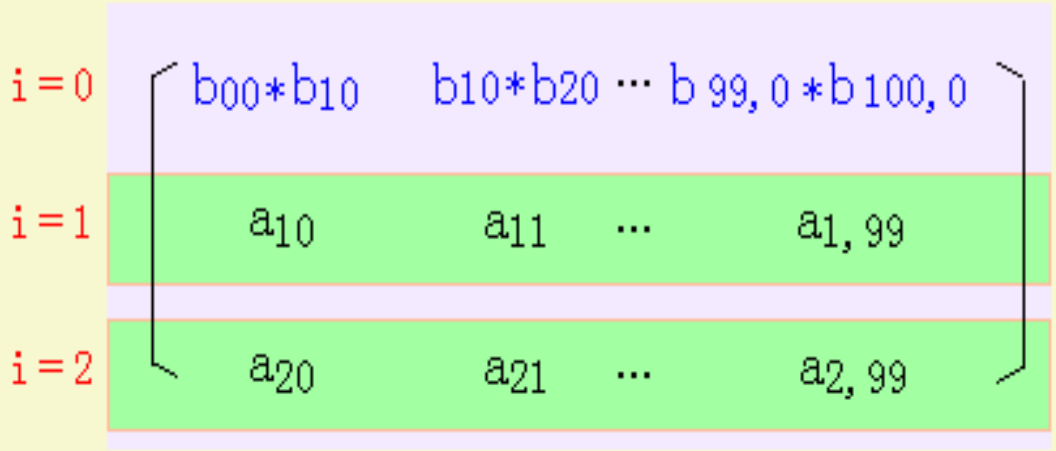
b 数组


i=0
i=1
i=2
b00*b10 b10*b20 … b99,0*b100,0
a10 a11 … a1,99
a20 a21 … a2,99

a 数组


b00 b01 b02
b10 b11 b12
b20 b21 b22
b99,0 b99,1 b99,2
b100,0 b100,1 b100,2

b 数组

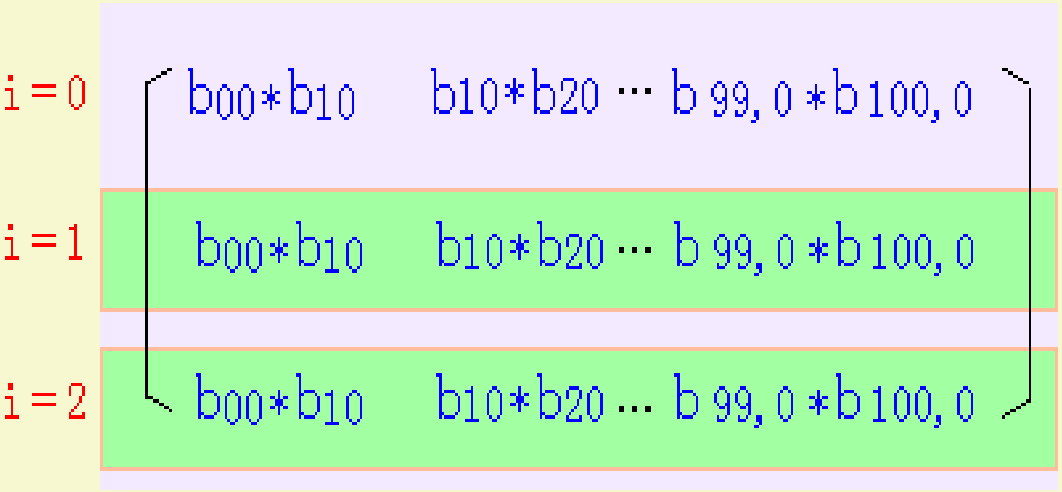


a 数组

用同样的计算过程
可得这两行的结果

$$\begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \\ \vdots & \vdots & \vdots \\ b_{99,0} & b_{99,1} & b_{99,2} \\ b_{100,0} & b_{100,1} & b_{100,2} \end{bmatrix}$$

b 数组



a 数组

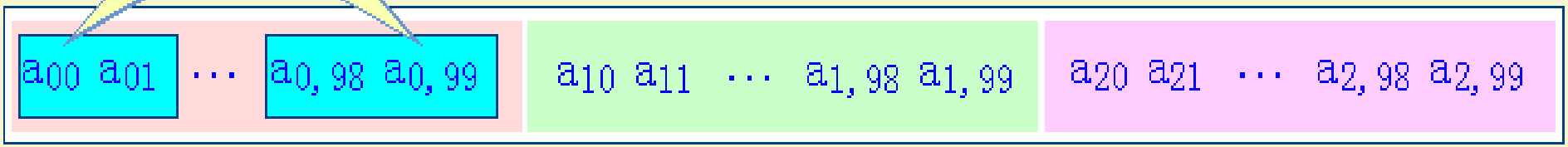
$$\begin{pmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \\ \vdots & \vdots & \vdots \\ b_{99,0} & b_{99,1} & b_{99,2} \\ b_{100,0} & b_{100,1} & b_{100,2} \end{pmatrix}$$

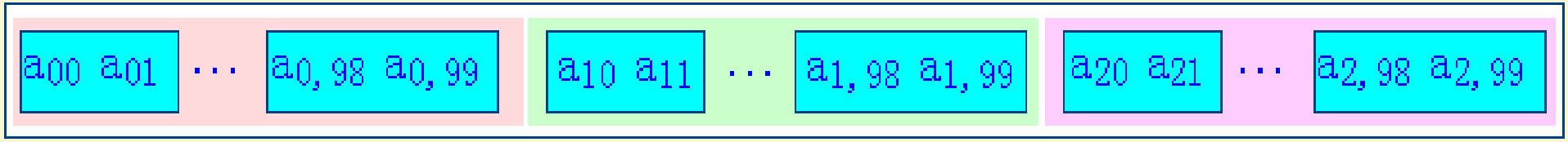
b 数组

1. 数组a受益于空间局部性

$$\begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0,98} & a_{0,99} \\ a_{10} & a_{11} & \cdots & a_{1,98} & a_{1,99} \\ a_{20} & a_{21} & \cdots & a_{2,98} & a_{2,99} \end{pmatrix}$$

块大小为16字节，每个元素为8字节，故每块中含两个元素。





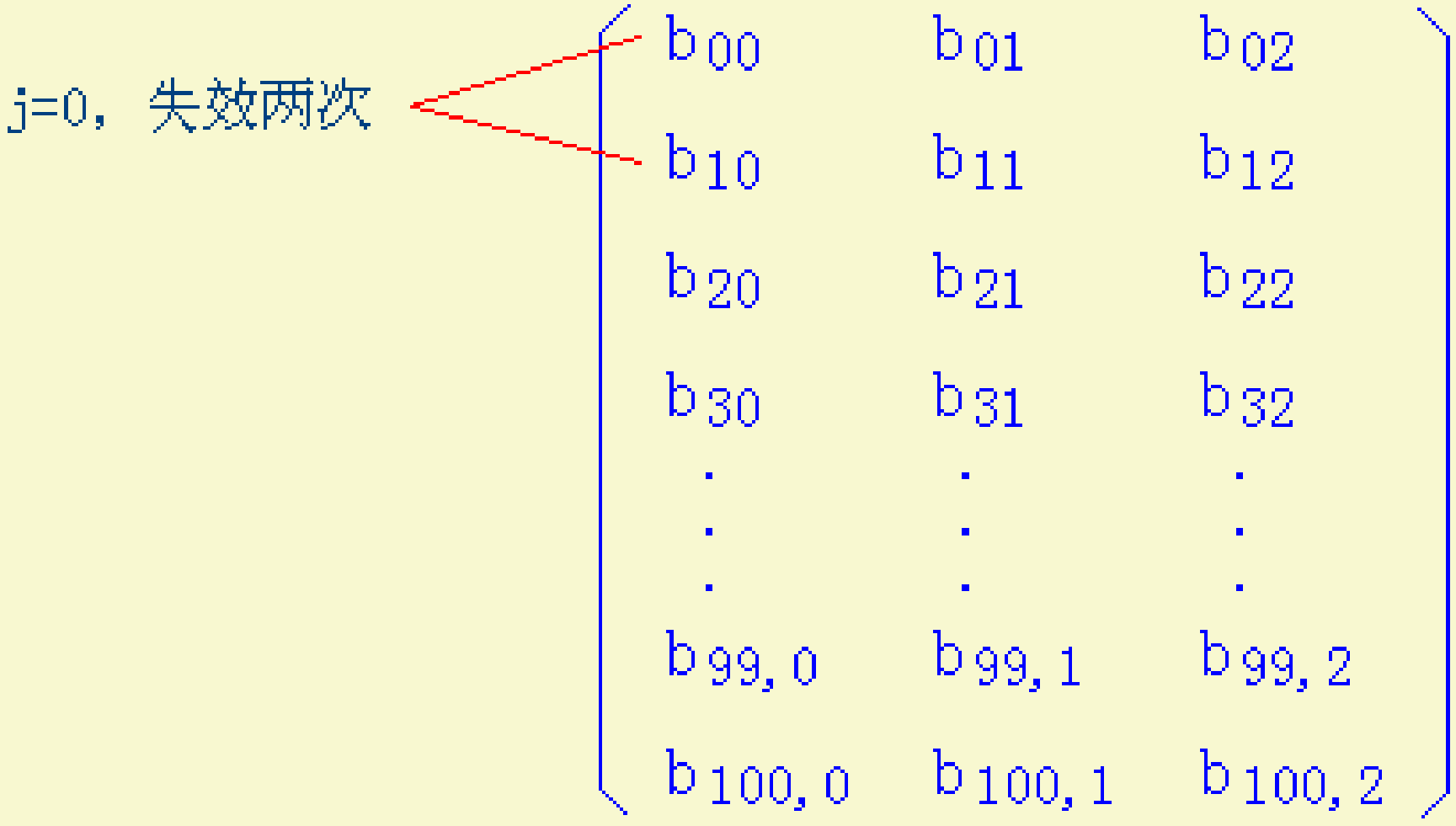
共3×100=300个元素

故失效次数=300/2=150

2. 数组b两次受益于时间局部性

(1) 对于i的每次循环，都访问同样的元素

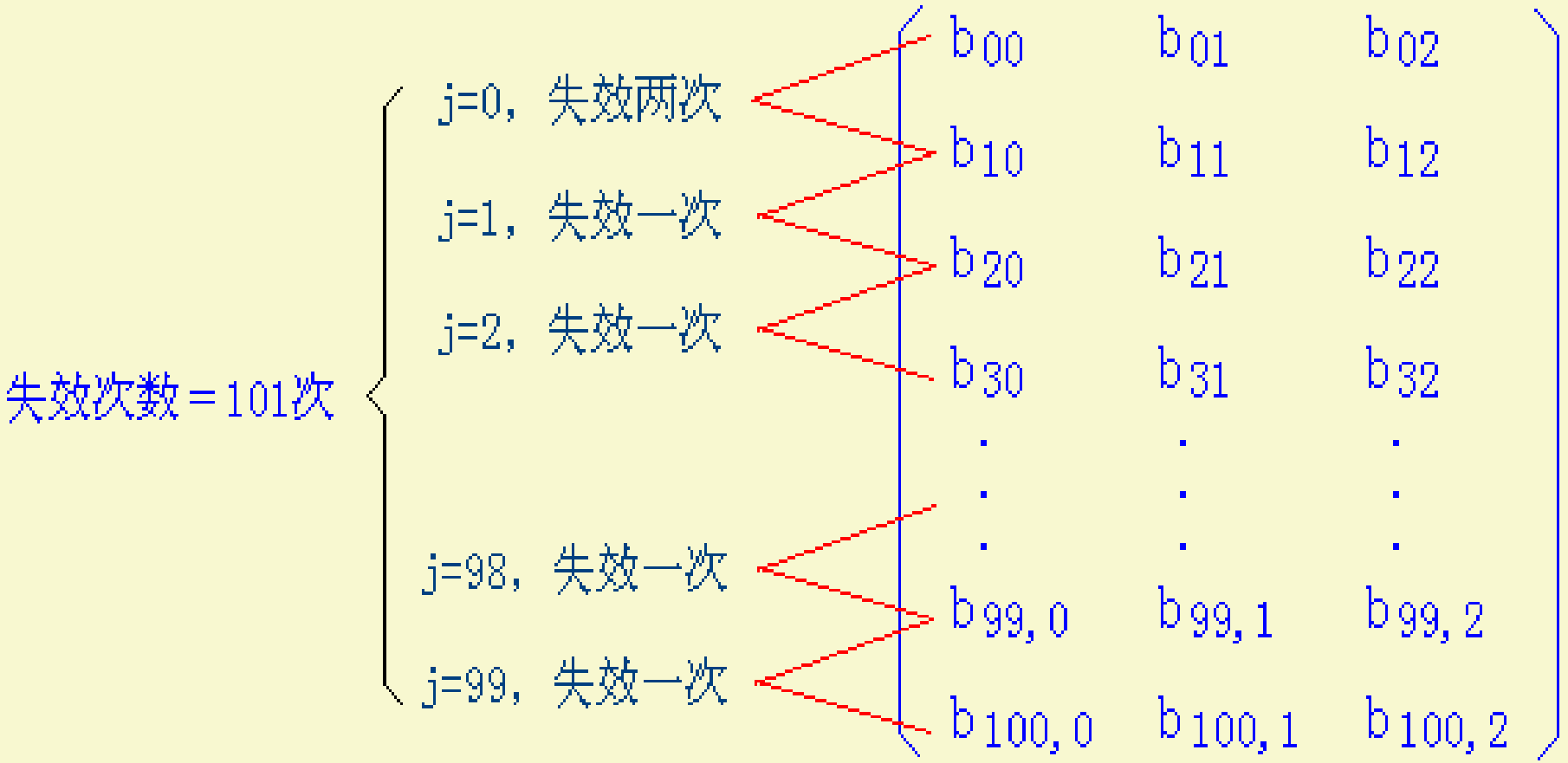
(2) 对于j的每次循环，都使用一次上一次循环用过的b元素



2．数组b两次受益于时间局部性

（1）对于i的每次循环，都访问同样的元素

（2）对于j的每次循环，都使用一次上一次循环用过的b元素



1．数组a受益于空间局部性

共3×100＝300个元素

故失效次数＝300/2＝150

2．数组b两次受益于时间局部性

（1）对于i的每次循环，都访问同样的元素

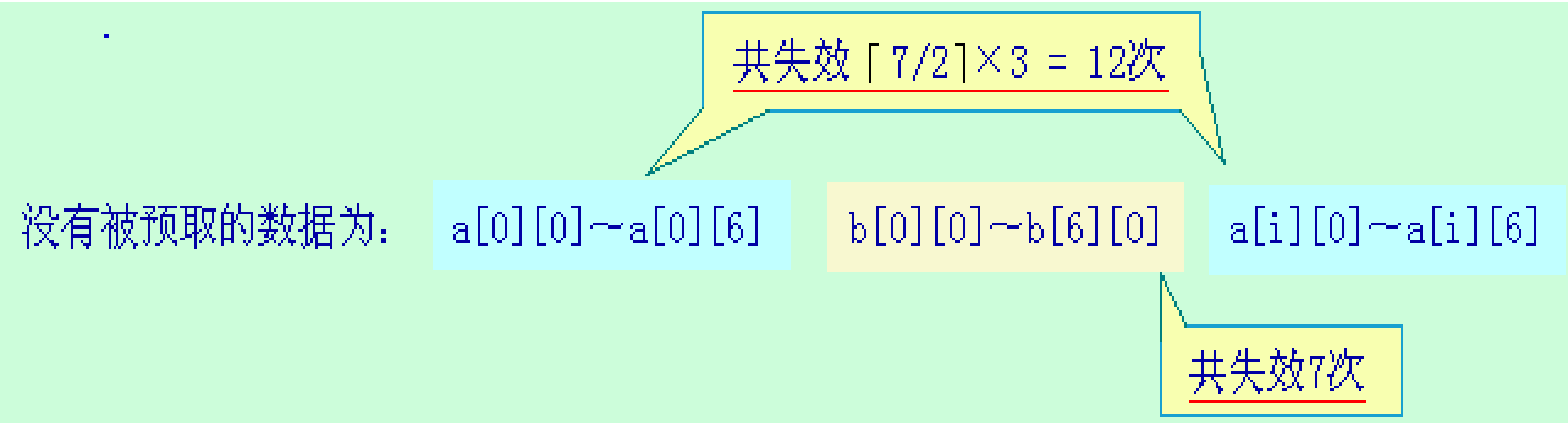
（2）对于j的每次循环，都使用一次上一次循环用过的b元素

失效次数＝101次

在没有预取功能时，这个循环引起的数据Cache失效次数为：

150＋101＝251

```
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]
    }
    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];
    }
```


共失效⌈7/2⌉×3 = 12次
没有被预取的数据为：
a[0][0]~a[0][6]
b[0][0]~b[6][0]
a[i][0]~a[i][6]
共失效7次

例8.8

在以下条件下，计算例8.7中所节约的时间：

(1) 假设预取可以被重叠或与Cache失效重叠执行，从而能以最大的存储带宽传送数据。

(2) 不考虑Cache失效时，修改前的循环每7个时钟周期循环一次。修改后的程序中，第一个预取循环每9个时钟周期循环一次，而第二个预取循环每8个时钟周期循环一次(包括外层for循环的开销)。

(3) 一次失效需100个时钟周期。

解：

修改前：

循环时间 = 300×7 = 2100

失效开销 = 251×100 = 25100

2100 + 25100 = 27200

修改后：

循环时间 = 100×9 + 200×8 = 2500

失效时间 = (4+7)*100+8*100 = 1900

2500 + 1900 = 4400

加速比 = 27200/4400 = 6.2

8.3.7 编译器优化

1. 基本思想

在编译时，对程序中的指令和数据进行重新组织，以降低Cache失效率。

2. 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。
  - 若Cache容量为8K字节，则缺失率降低75％
- 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。
- 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。

3. 数据对存储位置的限制比指令的少，因此更便于优化

通过把数据重新组织，使一块数据被从Cache替换出去之前，能最大限度利用其中的数据(访问次数最多)

针对数据的编译优化技术包括:

- 数组合并

  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。

- 内外循环交换
- 循环融合

  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 。

- 分块

(1) 数组合并

举例：

```
        /* 修改前 */
    int  val [SIZE];
int  key [SIZE];
    /*  修改后 */
struct merge {
int val ;
int key ;
} ;
struct merge merged_array[size];
```

(2) 内外循环交换

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

举例：

```
/* 修改前 */
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];
```

```
  /* 修改后 */
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];
  }
```

(4) 分块

把对数组的整行或整列访问改为按块进行。

举例：

```
 /*  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;
 }
```

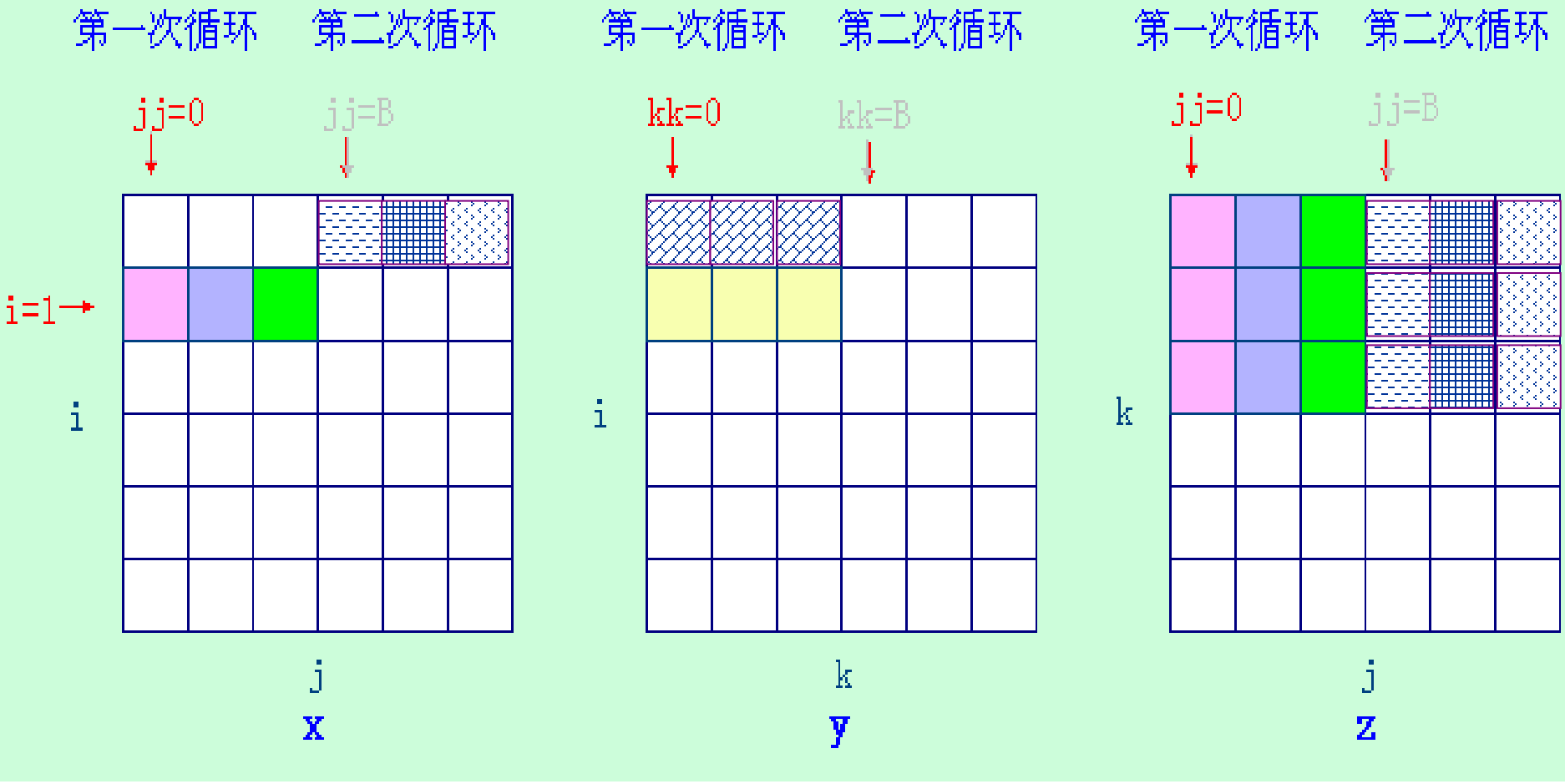
计算过程

失效次数：$2N^3 + N^2$

```
/* 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;
}
```

计算过程

失效次数：$(N/B)\times(N/B)\times N\times(2B) + N^2 = 2N^3/B+N^2$


第一次循环
第二次循环
jj=0
jj=B
i=1
i
j
x
第一次循环
第二次循环
kk=0
kk=B
i
k
y
第一次循环
第二次循环
jj=0
jj=B
k
j
z

8.3 降低Cache失效率的方法

增加块大小

提高相联度

Victim Cache

伪相联 Cache

硬件预取

编译器控制的预取

用编译器技术减少Cache失效

共7种

降低失效率

降低Cache失效率的方法

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
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
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

谢谢！