第5章 存储层次

5.1 存储器的层次结构

5.1.1 从单级存储器到多级存储器

1. 从用户的角度来看，存储器的三个主要指标：

   容量、速度和价格（指每位价格）

2. 人们对这三个指标的要求

   容量大、速度快、价格低

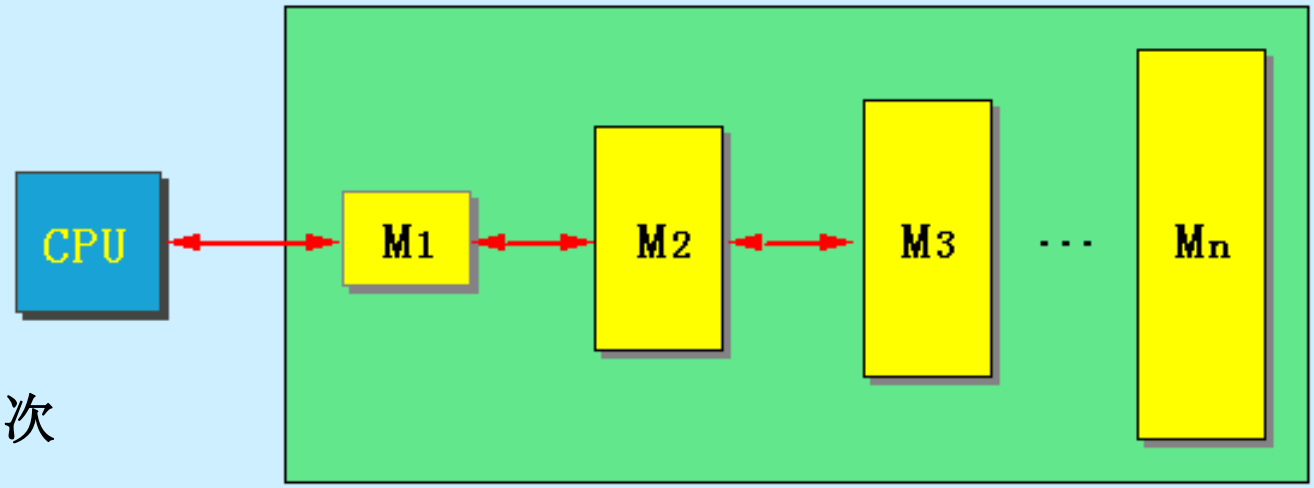
3. 三个要求是相互矛盾的

   - 速度越快，每位价格就越高；
   - 容量越大，每位价格就越低；
   - 容量越大，速度越慢。

4. 解决方法

采用多种存储器技术，构成所谓的存储层次。

演示 I　演示 II　（局部性原理）



多级存储层次

对于绝大多数程序而言，所访问的指令和数据在地址上不是均匀的，而是相对聚集的. 包含时间局部性和空间局部性. 前者指程序将用到的信息很可能就是现在使用的信息. 后者指程序将用到的信息很可能与现在正在使用的信息在存储空间上是相邻的. 多级存储器之间以块或页面传送数据，其目标是：从CPU来看，存储系统速度接近M1的速度，而容量和位价格都接近Mn的容量和价格.

5.1.2 存储层次的性能参数

C，H，T_A

假设：S —— 容量

T_A —— 访问时间

C —— 每位价格

下面仅考虑由M_1和M_2构成的两级存储层次：

- M_1的参数：S_1，T_{A1}，C_1
- M_2的参数：S_2，T_{A2}，C_2

$S_1 < S_2$

$T_{A1} < T_{A2}$

$C_1 > C_2$

1. 每位价格C

2. 命中率*H* 和失效率*F*

- 命中率：CPU访问存储系统时，在M_1中找到所需信息的概率。

$$H=\frac{N_1}{N_1+N_2}$$

  - N_1 —— 访问M_1的次数
  - N_2 —— 访问M_2的次数

- 失效率：F＝1－H

3. 平均访问时间T_A

$$T_A = HT_{A1} + (1-H)(T_{A1}+T_M)$$
$$= T_{A1} + (1-H)T_M$$
$$或\ T_A = T_{A1} + FT_M$$

分两种情况来考虑CPU的一次访存：

- 当命中时，访问时间即为T_{A1}（命中时间）
- 当不命中时，情况比较复杂。

  不命中时的访问时间为：$T_{A2}+T_B+T_{A1}=T_{A1}+T_M$

  $$T_M = T_{A2}+T_B$$

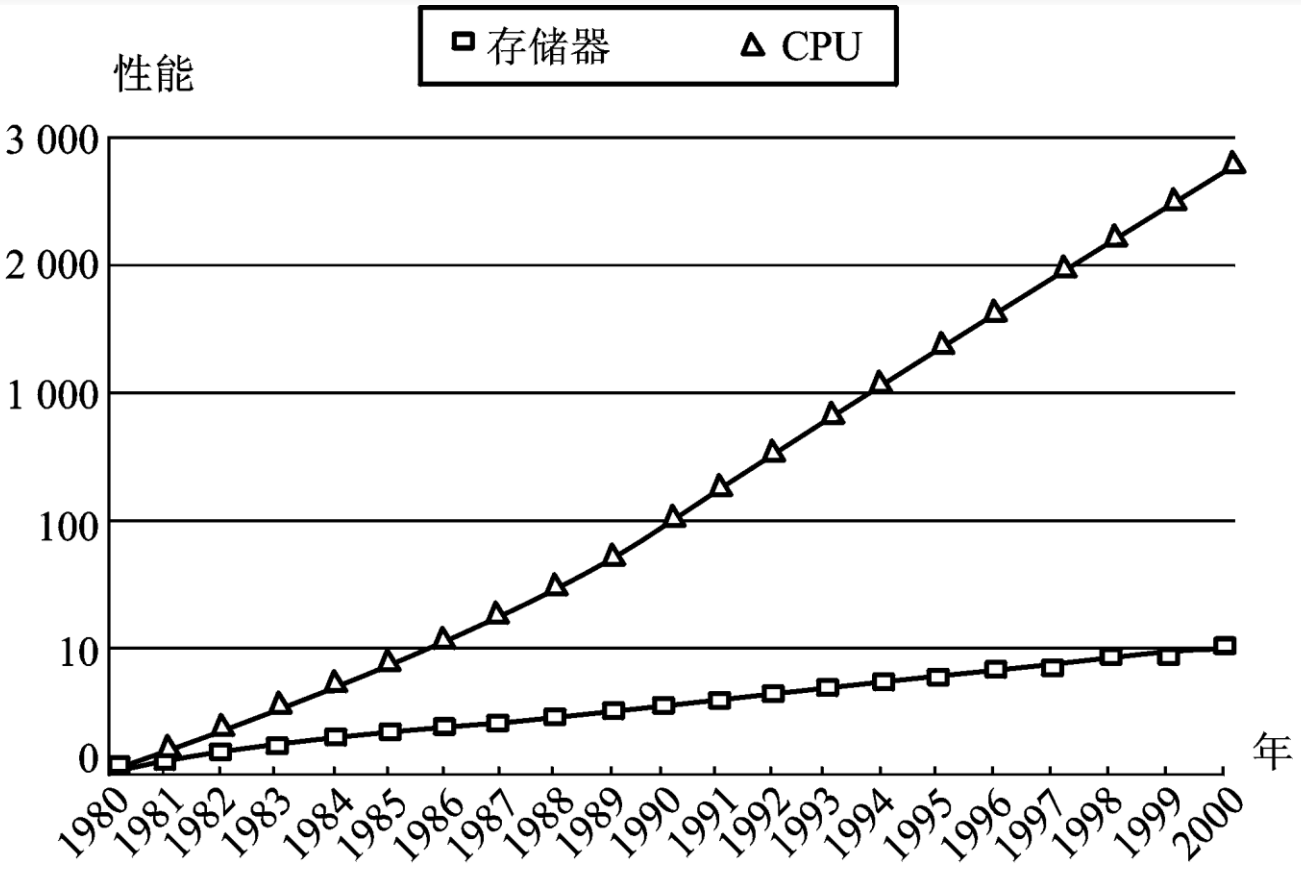
  - 失效开销T_M：从向M_2发出访问请求到把整个数据块调入M_1中所需的时间。
  - 传送一个信息块所需的时间为T_B。

5.1.3 “Cache－主存”和“主存－辅存”层次

从主存的角度来看

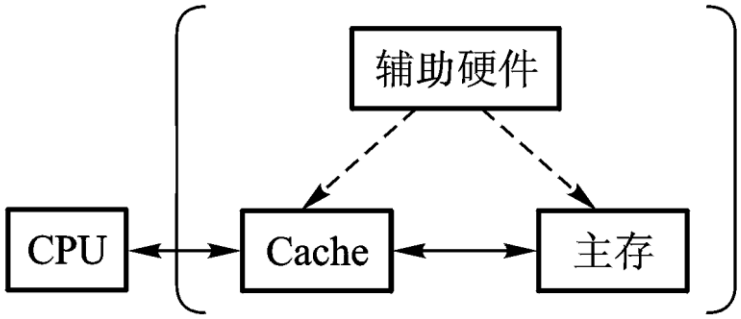
- “Cache－主存”层次：弥补主存速度的不足
- “主存－辅存”层次： 弥补主存容量的不足

1. “Cache—主存”层次
   - 主存与CPU的速度差距
   - “Cache － 主存”层次
2. “主存－辅存”层次

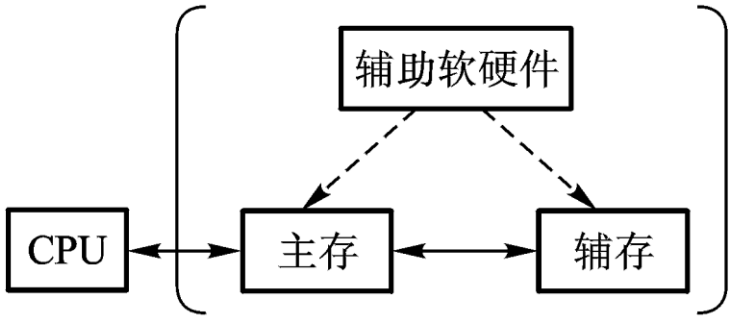


1980年以来存储器和CPU性能随时间而提高的情况

（以1980年时的性能作为基准）



(a) “Cachc-主存”层次



(b) “主存-辅存”层次

两种存储层次

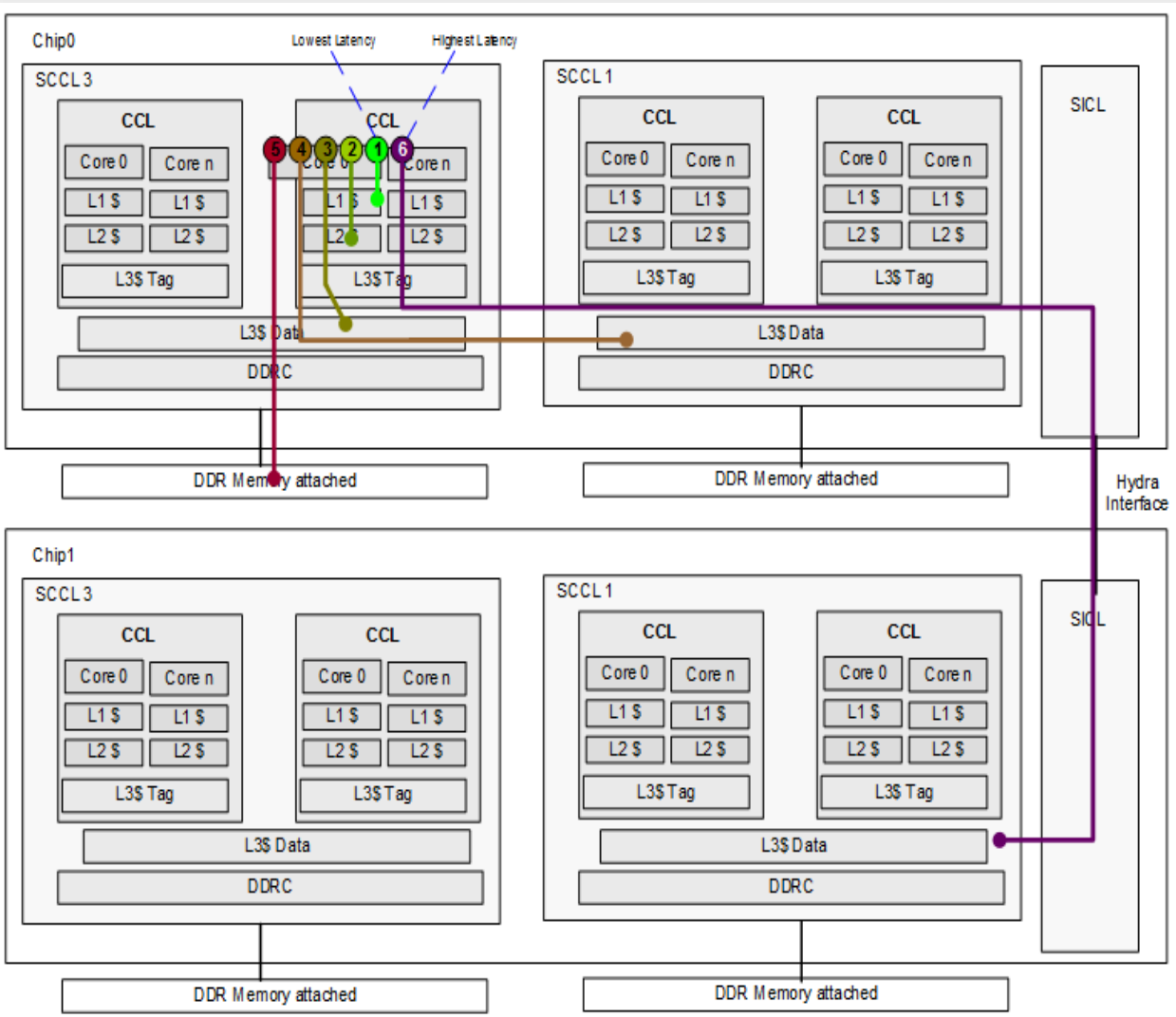
借助辅助硬件,使CACHE与主存形成一个有机整体,弥补主存速度不足.由硬件实现,对应用与系统程序员都透明.

依靠辅助软硬件,使主存与辅存形成一个有机整体,弥补主存容量不足. 用来实现虚拟存储器,提供程序空间.

“Cache－主存”与“主存－辅存”层次的区别

比较项目 \ 存储层次	“Cache －主存”层次	“主存－辅存”层次
目　　的	为了弥补主存速度的不足	为了弥补主存容量的不足
存储管理实现	主要由专用硬件实现	主要由软件实现
访问速度的比值(第一级和第二级)	几比一	几百比一
典型的块(页)大小	几十个字节	几百到几千个字节
CPU对第二级的访问方式	可直接访问	均通过第一级
失效时CPU是否切换	不切换	切换到其他进程

鲲鹏920系列芯片架构——内存子系统 (2)



- 一个CPU Die包含4个DDR Channel
- 一个Socket包含2个CPU Die，8个DDR Channel
- 每个控制器支持2DPC 2933
- 本地内存访问均在本地进行，不走片间互联总线，因此访存时延最小，总体性能最好。

时延	ARM CPU Cycles	Skylake Cycles
Register	1	1
L1 cache	4	4
L2 cache	8	14
L3 cache	40	55
DRAM	71 -221(ns)	83-143(ns)

5.1.4 存储层次的四个问题

1. 当把一个块调入高一层(靠近CPU)存储器时，可以放在哪些位置上?

   （映像规则）

2. 当所要访问的块在高一层存储器中时，如何找到该块?

   （查找算法）

3. 当发生失效时，应替换哪一块？

   （替换算法）

4. 当进行写访问时，应进行哪些操作?

   （写策略）

5.2 Cache的基本知识

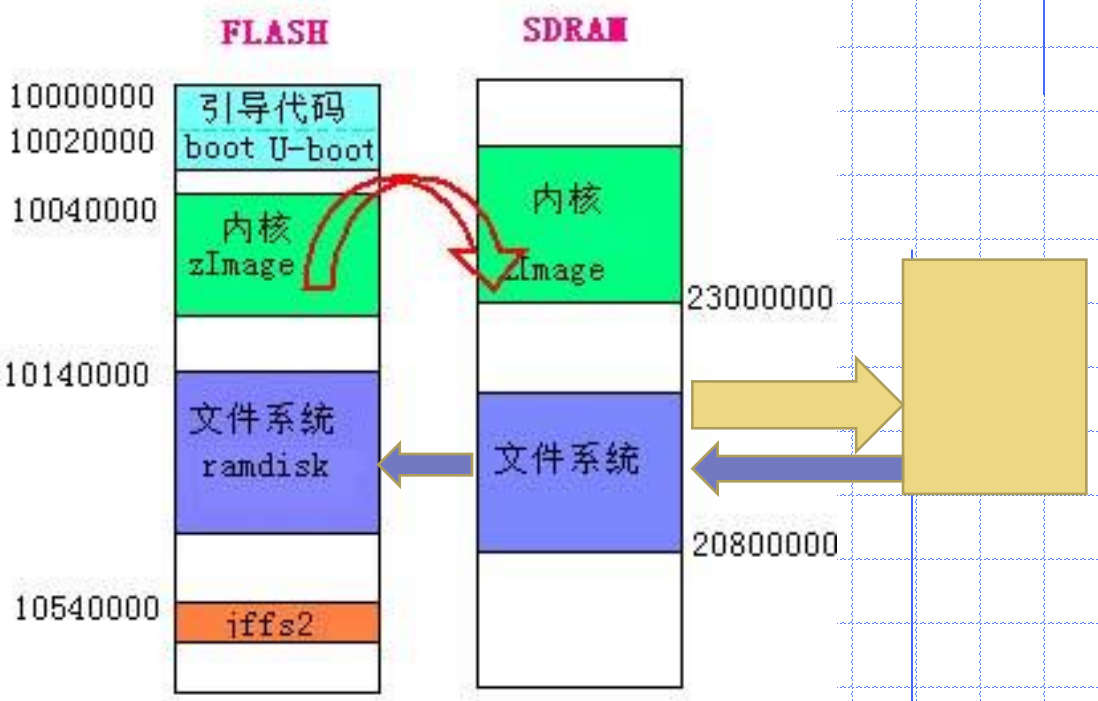
1. 存储空间分割与地址计算
2. Cache和主存分块

5.2.1 映象规则

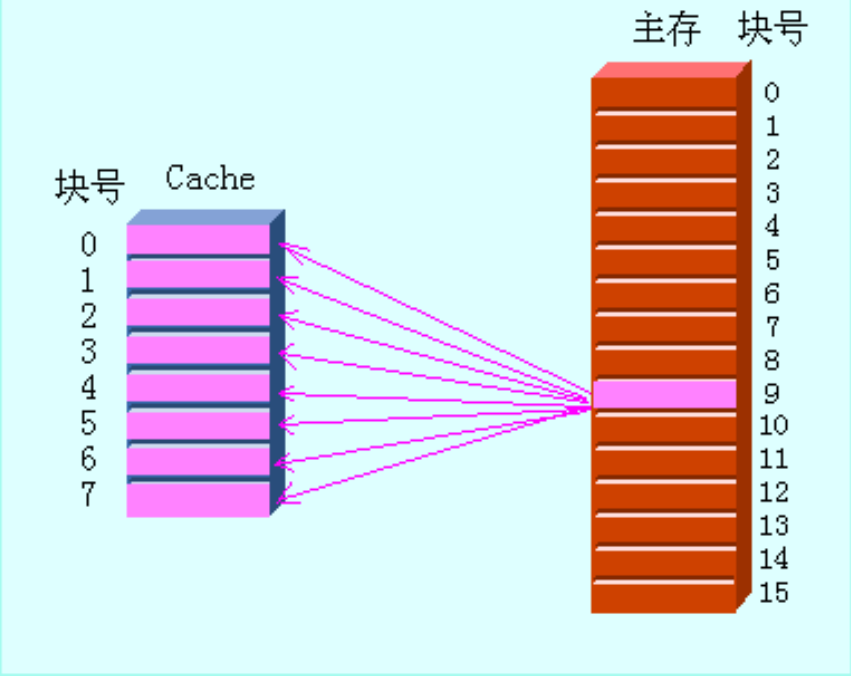
1. 全相联映象

- 全相联：主存中的任一块可以被放置到Cache中的任意一个位置。 举例
- 对比：阅览室位置 —— 随便坐
- 特点：空间利用率最高，冲突概率最低，实现最复杂。

#xdownload 20000000 boot.bin
#fl 10000000 20000000 20000
#xdownload 20000000 u-boot.bin
#fl 10020000 20000000 20000
#xdownload 23000000 Zimage
#fl 10040000 23000000 f0000
#xdownload 20800000 ramdisk
#fl 10140000 20800000 500000
#xdownload 20000000 jffs2
#fl 10540000 20000000 20000



全相联映射
（举例）



2. 直接映象

- 直接映象：主存中的每一块只能被放置到Cache中唯一的一个位置。 举例

  （循环分配）

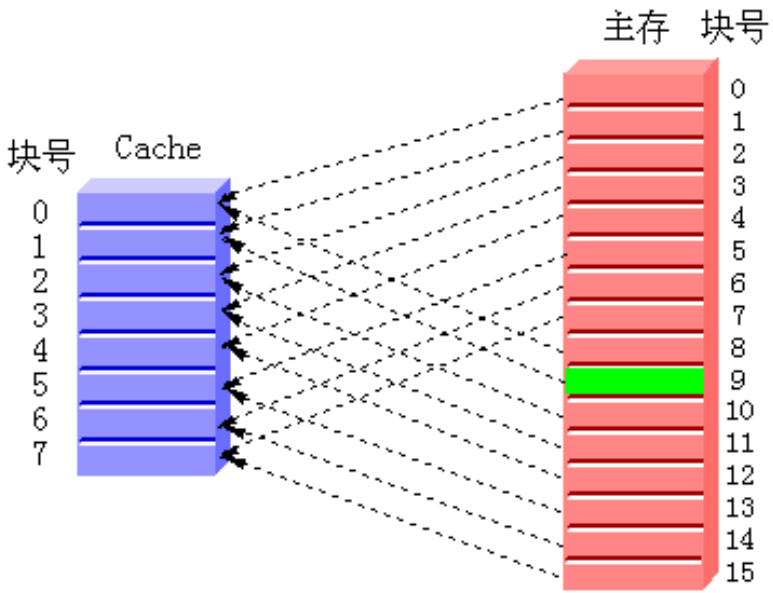
- 对比：阅览室位置 —— 只有一个位置可以坐
- 特点：空间利用率最低，冲突概率最高，

  实现最简单。
- 对于主存的第 i 块，若它映象到Cache的第 j 块，则

  $j = i \bmod (M)$ （M为Cache的块数）

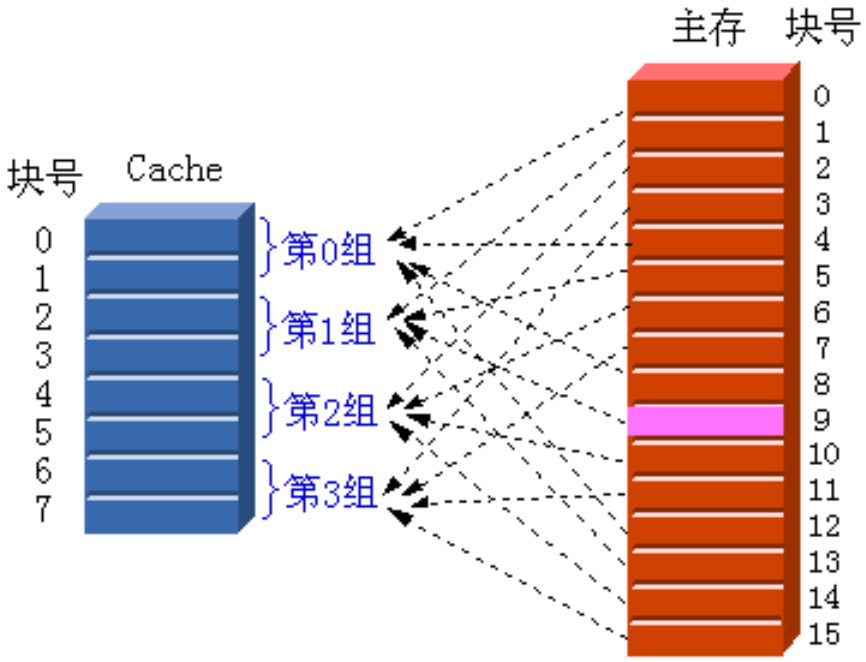
- 设$M=2^m$，则当表示为二进制数时，j实际上就是i的低m位：

直 接 映 射
(举 例)



3. 组相联映象

- 组相联：主存中的每一块可以被放置到Cache中唯一的一个组中的任何一个位置。 举例
- 组相联是直接映象和全相联的一种折中



- 组的选择常采用位选择算法
  - 若主存第i 块映象到第k 组，则

    $k=i \bmod (G)$ （G为Cache的组数）
  - 设$G=2^g$，则当表示为二进制数时，k 实际上就是i的低 g 位：

- 低g位以及直接映象中的低m位通常称为索引。

- *n* 路组相联：每组中有n个块（$n=M/G$ ）。

  n 称为相联度。

  相联度越高，Cache空间的利用率就越高，块冲突概率就越低，失效率也就越低。

	n（路数）	*G*（组数）
全相联	***M***	**1**
直接映象	**1**	***M***
组相联	$1<n<M$	$1<G<M$

- 绝大多数计算机的Cache：$n \leqslant 4$

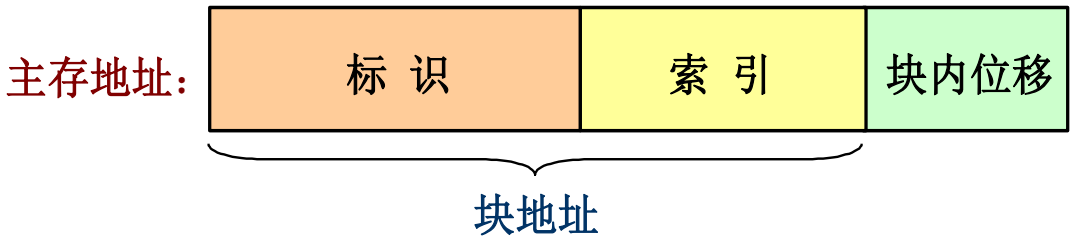
  想一想：相联度一定是越大越好？

5.2.2 查找算法

- 当CPU访问Cache时，如何确定Cache中是否有所要访问的块？
- 若有，如何确定其位置？

1. 通过查找目录表来实现

- 目录表的结构
  - 主存块的块地址的高位部分，称为标识 。
  - 每个主存块能唯一地由其标识来确定



主存块被调入CACHE中某一个块位置时,其标识被填入目录表与该CACE块相对应的项中,其有效位置”1”

- 只需查找候选位置所对应的目录表项

2. 并行查找与顺序查找
   - 提高性能的重要思想：主候选位置(MRU块)

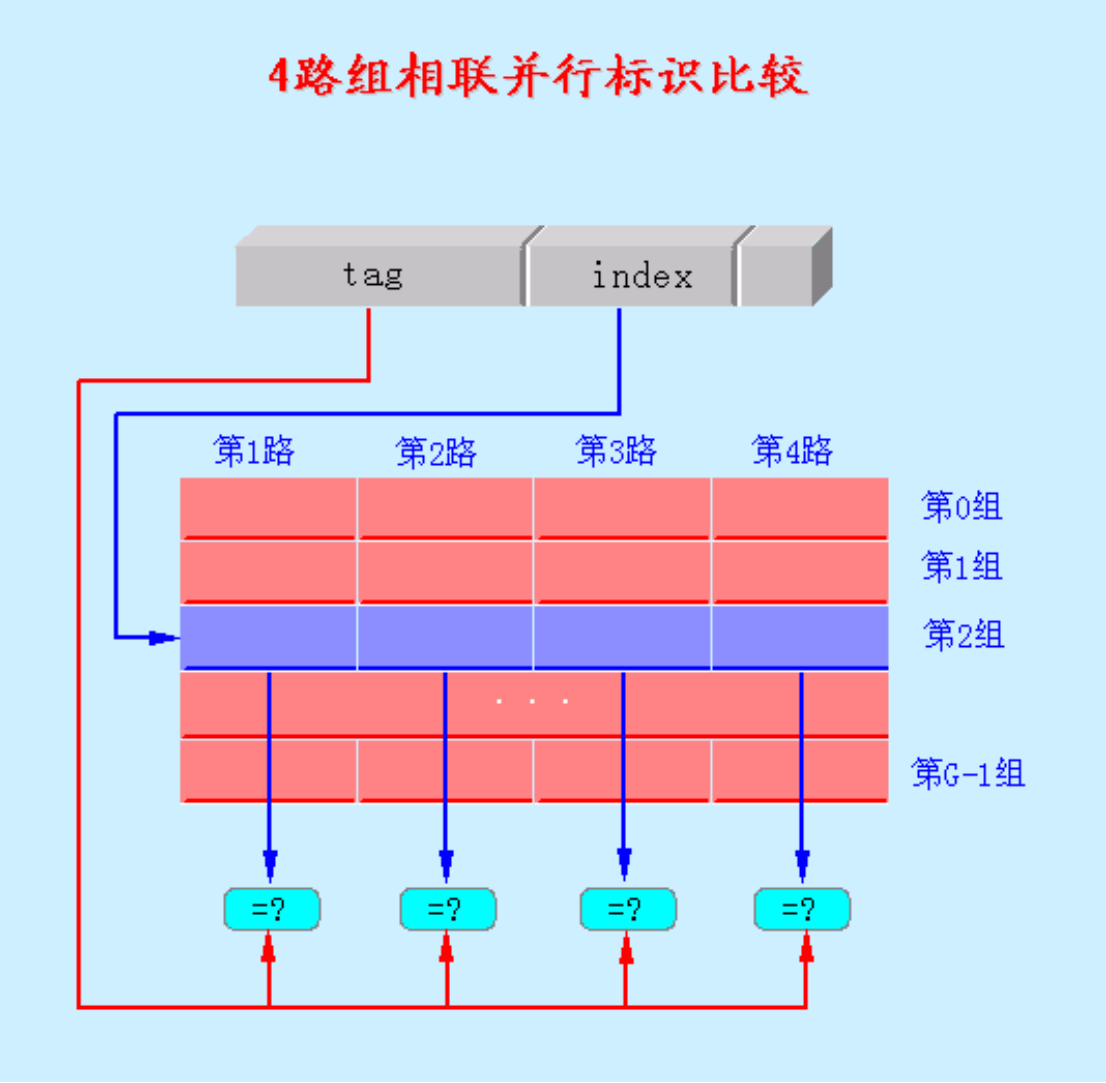
     （前瞻执行）

     根据映像规则不同,一个主存块可能映像多个CACHE块(候选位置),访问主存时,查找候选位置目录表,有效位为”1”,则对应的CACHE块是所找的块.各候选位置的标识可并行检查.

     先进行”主存→CACHE”的地址变换,得到CACHE地址后再去访问CACHE存储体.为提高速度,把地址变换和访问CACHE存储体安排同时进行.因此把所有候选位置中的内容都读出,在地址变换完成后,再选择有效位为”1”的送给CPU.

3. 并行查找的实现方法
   - 相联存储器
   - 单体多字存储器＋比较器
     - 举例： 4路组相联并行标识比较

       （比较器的个数及位数）
     - 4路组相联Cache的查找过程
     - 直接映象Cache的查找过程


4路组相联并行标识比较
tag
index
第1路
第2路
第3路
第4路
第0组
第1组
第2组
第G-1组
=?
=?
=?
=?

5.2.3 替换算法

1. 所要解决的问题：当新调入一块，而Cache又已被占满时，替换哪一块？
   - 直接映象Cache中的替换很简单

     因为只有一个块，别无选择。
   - 在组相联和全相联Cache中，则有多个块供选择。
2. 主要的替换算法有三种
   - 随机法

     优点：实现简单
   - 先进先出法（FIFO）

- 最近最少使用法LRU
  - 选择近期最少被访问的块作为被替换的块。
    （实现比较困难）
  - 实际上：选择最久没有被访问过的块作为被替换的块。
  - 优点：失效率低。反映程序的局部性原理,但实现复杂,硬件成本较高.随着组数的增加,实现代价越来越高.
- LRU和随机法的失效率的比较

LRU和随机法的失效率的比较

Cache容量	相联度					
	2路		4路		8路	
	LRU	随机替换	LRU	随机替换	LRU	随机替换
16KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

LRU和随机法分别因其失效率低和实现简单而被广泛采用。

对于容量大的CACHE,LRU与随机法的命中率差别不大.

5.2.4 写策略

1. “写”在所有访存操作中所占的比例

- 统计结果表明，对于一组给定的程序：
  - load指令：26%
  - store指令：9%
- “写”在所有访存操作中所占的比例：

  9%/(100%+26%+9%)≈7%
- “写”在访问数据Cache操作中所占的比例：

  9%/(26%+9%)≈25%

2. “写”操作必须在确认是命中后才可进行
3. “写”访问有可能导致Cache和主存内容的不一致
4. 两种写策略

写策略是区分不同Cache设计方案的一个重要标志。

- 写直达法
  - 执行“写”操作时，不仅写入Cache，而且也写入下一级存储器。
- 写回法（也称为拷回法）
  - 执行“写”操作时，只写入Cache。仅当Cache中相应的块被替换时，才写回主存。（设置“修改位”）

5. 两种写策略的比较

- 写回法的优点：速度快，所使用的存储器带宽较低。
- 写直达法的优点：易于实现，一致性好。

6. 采用写直达法时，若在进行“写”操作的过程中CPU必须等待，直到“写”操作结束，则称CPU写停顿。

- 减少写停顿的一种常用的优化技术：

  采用写缓冲器.CPU把数据写入缓冲器,马上可以继续执行,使下一级存储器的更新与CPU的执行重叠起来.

7. “写”操作不命中时的调块

- 按写分配(写时取)

  写失效时，先把所写单元所在的块调入Cache，再行写入。

- 不按写分配(绕写法)

  写失效时，直接写入下一级存储器而不调块。

8. 写策略与调块

- 写回法 —— 按写分配
- 写直达法 —— 不按写分配

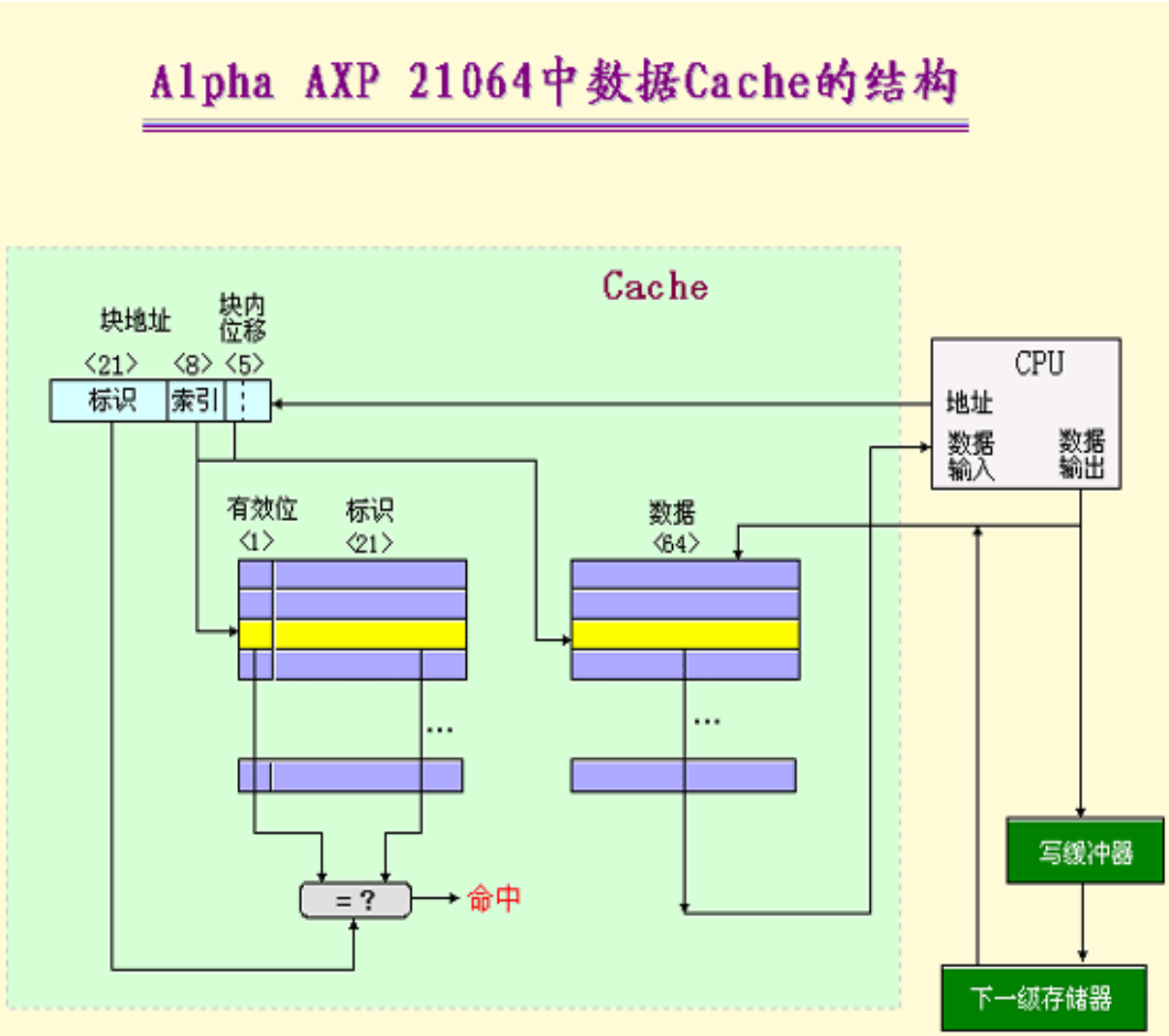
5.2.5 Cache的结构

例：DEC的Alpha AXP21064中的内部数据Cache

1. 简介

- 容量：8KB
- 块大小：32B
- 块数：256
- 采用不按写分配
- 映象方法：直接映象
- “写”策略：写直达
- 写缓冲器大小：4个块

2. 结构图



物理地址34位,分为块地址29位和块内位移5位.
CACHE容量=2^index*相联度*块大小
2^index= CACHE容量/(相联度*块大小
=8192/32=2^8
index=8
标识为29-8=21位

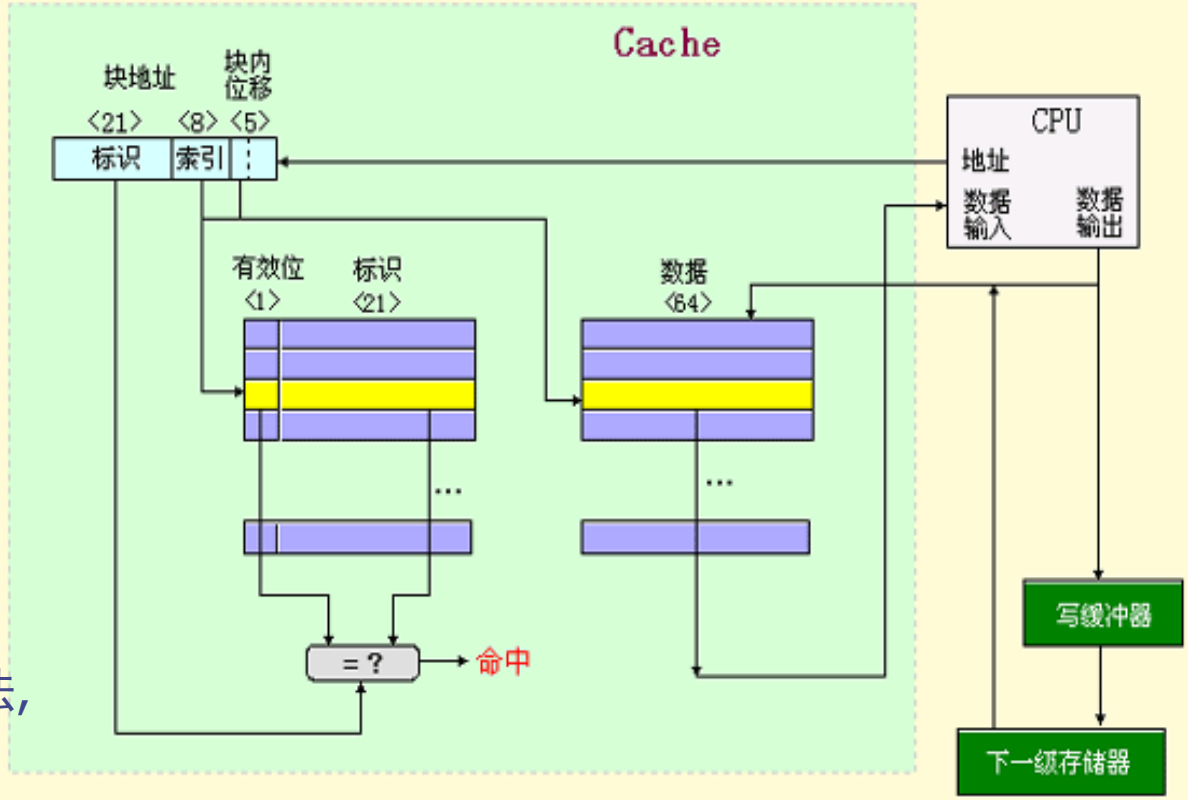
2. 结构图

Alpha AXP 21064中数据Cache的结构

1)处理器送给CACHE的物理地址34位.块地址29位与块内位移5位.容量=2^index*相联度*块大小

2)用索引作地址从256个目录项中选择一项,读出相应标识和有效位.

3)把上一步读出的标识与CPU送来的物理地址中的标识比较.”1”为有效.采用写直达法,还应将数据写入写缓冲器.



4) 比较结果匹配,标识位”1”有效,本次CACHE访问命中,通知CPU取走数据.

5) 失效:一是读不命中,CACHE向CPU发出一个暂停信号,通知等待,从下一级存储器读入32B数据,数据总线16B,需要读取2次.CACHE与主存采用直接映像,只替换一个块,更新该块的数据,及块目录表的标识、有效位。二是写不命中，采用不按写分配规则。CPU使数据“绕过”CACHE，直接写入主存。

3. 工作过程
   - “读”访问命中
   - “写”访问命中
   - 失效情况下的操作
4. 混合Cache与分离Cache
   - 优缺点
   - 失效率的比较

失 效 率 的 比 较

容　量	指令 Cache	数据 Cache	混合 Cache
1 KB	3.06%	24.61%	13.34%
2 KB	2.26%	20.57%	9.78%
4 KB	1.78%	15.94%	7.24%
8 KB	1.10%	10.19%	4.57%
16 KB	0.64%	6.47%	2.87%
32 KB	0.39%	4.82%	1.99%
64 KB	0.15%	3.77%	1.36%
128 KB	0.02%	2.88%	0.95%

5.2.6 Cache的性能分析

1. 失效率
   - 与硬件速度无关
   - 容易产生一些误导
2. 平均访存时间

   平均访存时间 = 命中时间＋失效率×失效开销

例5.1 假设Cache的命中时间为1个时钟周期，失效开销为50个时钟周期，在混合Cache中一次load或store操作访问Cache的命中时间都要增加1个时钟周期（因为混合Cache只有一个端口，无法同时满足两个请求，会导致结构冲突），根据表5.4所列的失效率，试问指令Cache和数据Cache容量均为16 KB的分离Cache和容量为32 KB的混合Cache相比，哪种Cache的失效率更低？又假设采用写直达策略，且有一个写缓冲器，并且忽略写缓冲器引起的等待。请问上述两种情况下平均访存时间各是多少？

条件：根据统计得出，指令CACHE的访问占全部访问的100%/（100%+26%+9%）=75%，对数据CACHE的访问占全部访问的（26%+9%）/（100%+26%+9%）=25%。

解　如前所述，约75%的访存为取指令。

因此，分离Cache的总体失效率为：

（75%×0.64%）＋（25%×6.47%）＝2.10%

根据表5.4，容量为32 KB的混合Cache的失效率略低一些，只有1.99%。

平均访存时间公式可以分为指令访问和数据访问两部分：

平均访存时间　＝

指令所占的百分比×（指令命中时间＋指令失效率×失效开销）

＋ 数据所占的百分比×（数据命中时间＋数据失效率×失效开销）

所以，两种结构的平均访存时间分别为：

$$\text{平均访存时间}_{\text{分离}} = 75\% \times (1+0.64\% \times 50) + 25\% \times (1+6.47\% \times 50)$$

$$= (75\% \times 1.32) + (25\% \times 4.325)$$

$$= 0.990 + 1.059 = 2.05$$

$$\text{平均访存时间}_{\text{混合}} = 75\% \times (1+1.99\% \times 50) + 25\% \times (1+1+1.99\% \times 50)$$

$$= (75\% \times 1.995) + (25\% \times 2.995)$$

$$= 1.496 + 0.749 = 2.24$$

因此，尽管分离Cache的实际失效率比混合Cache的高，但其平均访存时间反而较低。分离Cache提供了两个端口，消除了结构冲突。

3. 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×读失效率×读失效开销＋“写”的次数×写失效率×写失效开销
- 存储器停顿时钟周期数＝访存次数×失效率×失效开销

$$CPU\text{时间}=IC\times\left(CPI_{execution}+\frac{\text{访存次数}}{\text{指令数}}\times\text{失效率}\times\text{失效开销}\right)\times\text{时钟周期时间}$$

$$\text{每条指令的平均失效次数}=\frac{\text{失效次数}}{\text{指令数}}=\frac{\text{访存次数}\times\text{失效率}}{\text{指令数}}$$

$$CPU\text{时间}=IC\times\left(CPI_{execution}+\frac{\text{存储器停顿周期数}}{\text{指令数}}\right)\times\text{时钟周期时间}$$

例5.2 我们用一个和Alpha AXP类似的机器作为第一个例子。假设Cache失效开销为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失效率为2%，平均每条指令访存1.33次。试分析Cache对性能的影响。

解

$$\text{CPU 时间} = IC \times \left(CPI_{exe} + \frac{\text{存储器停顿周期数}}{\text{指令数}}\right) \times \text{时钟周期时间}$$

考虑Cache的失效后，性能为：

$$CPU时间_{有cache}=IC\times(2.0+1.33\times2\ \%\times50)\times时钟周期时间$$
$$=IC\times3.33\times时钟周期时间$$

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：

$$CPI=2.0+50\times1.33=68.5$$

4. Cache失效对于一个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失效开销的相对影响就越大。
- 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5.3 考虑两种不同组织结构的Cache：直接映象Cache和2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：

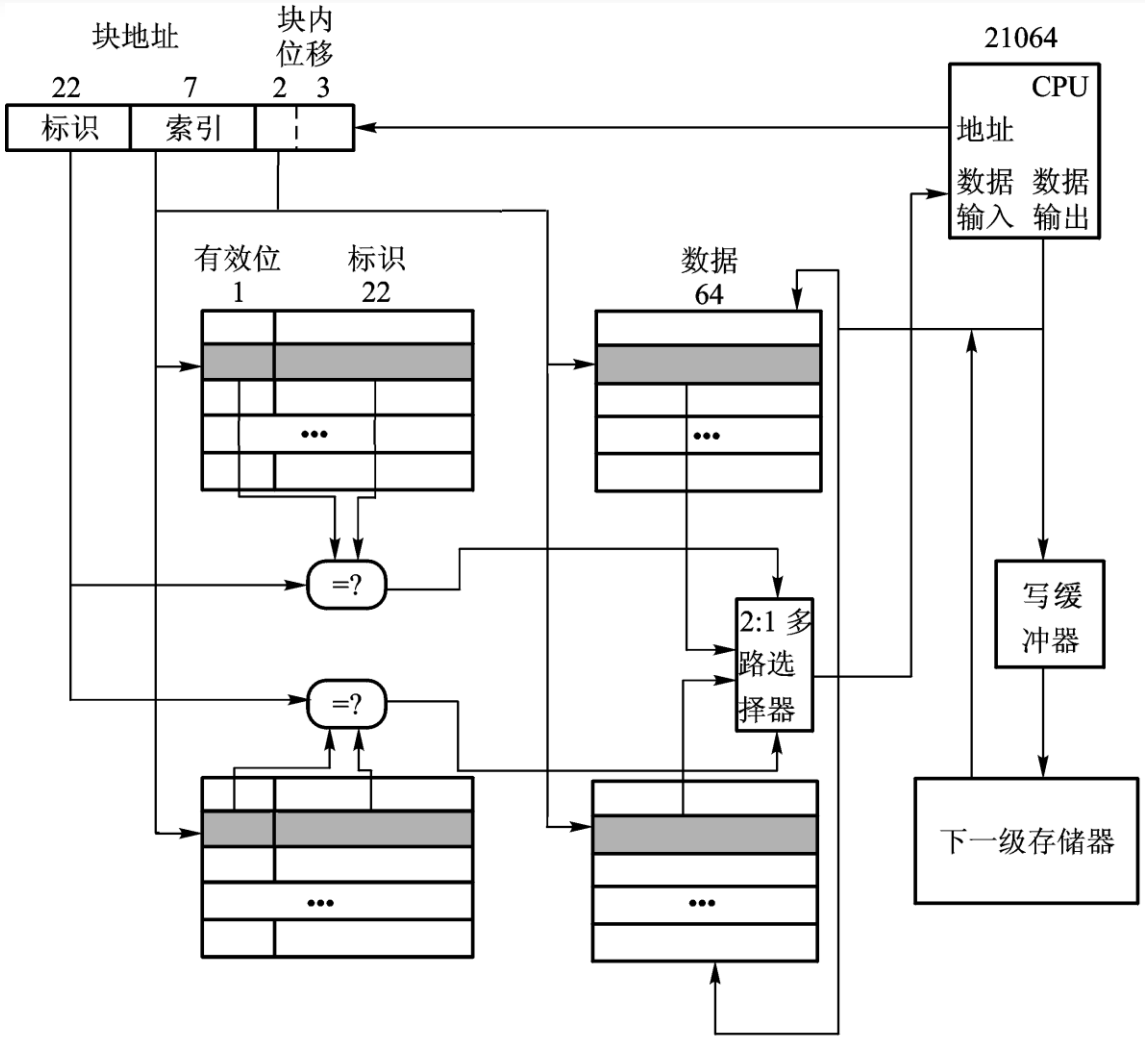
(1)理想Cache（命中率为100%）情况下的CPI为2.0，时钟周期为2ns，平均每条指令访存1.3次。

(2)两种Cache容量均为64KB，块大小都是32字节。

(3)图5.8说明，在组相联Cache中，必须增加一个多路选择器，用于根据标识匹配结果从相应组的块中选择所需的数据。因为CPU的速度直接与Cache命中的速度紧密相关，所以对于组相联Cache，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。

（4）这两种结构Cache的失效开销都是70 ns。（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，64 KB直接映象Cache的失效率为1.4%，相同容量的2路组相联Cache的失效率为1.0%。


块地址
块内
位移
22
7
2
3
标识
索引
21064
CPU
地址
数据
输入
数据
输出
有效位
1
标识
22
数据
64
=?
=?
2:1 多
路选
择器
写缓
冲器
下一级存储器

解 平均访存时间为：

平均访存时间＝命中时间＋失效率×失效开销

因此，两种结构的平均访存时间分别是：

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

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

2路组相联Cache的平均访存时间比较低。

$$CPU\text{ 时间}=IC\times(CPI\text{exe}+\text{每条指令的平均存储器停顿周期数})\times\text{时钟周期时间}$$

$$=IC\times(CPI\text{exe}\times\text{时钟周期时间}+\text{每条指令的平均存储器停顿时间})$$

因此：

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

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

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

直接映象Cache的平均性能好一些。

5.2.7 改进Cache的性能

1. 平均访存时间＝命中时间＋失效率×失效开销
2. 可以从三个方面改进Cache的性能：
   - 降低失效率
   - 减少失效开销
   - 减少Cache命中时间
3. 下面介绍17种Cache优化技术
   - 8种用于降低失效率
   - 5种用于减少失效开销
   - 4种用于减少命中时间

5.3 降低Cache失效率的方法

1. 三种失效(3C)

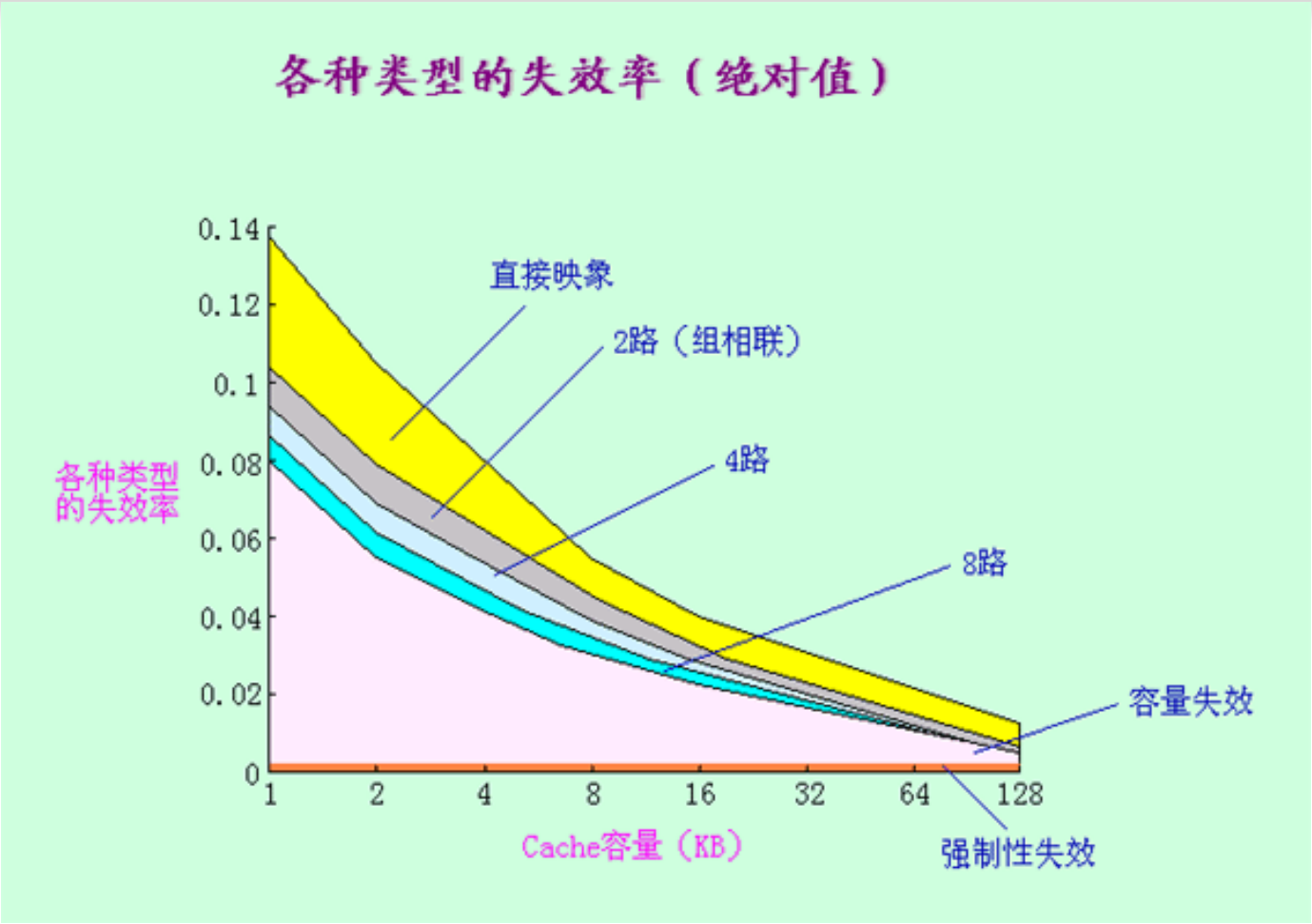
- 强制性失效(Compulsory miss)
  - 当第一次访问一个块时，该块不在Cache中，需从下一级存储器中调入Cache，这就是强制性失效。

    (冷启动失效，首次访问失效)
- 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。

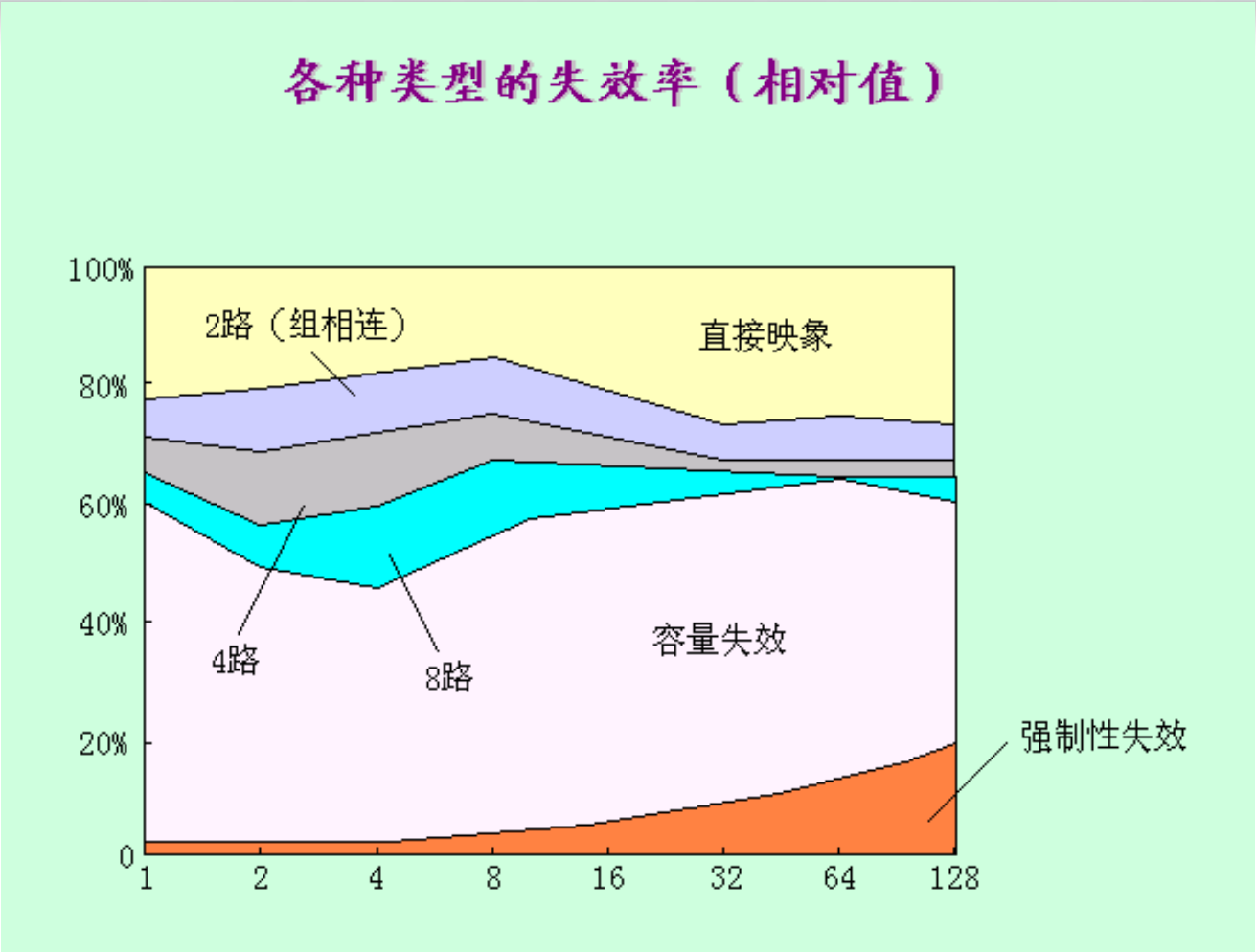
- 冲突失效(Conflict miss)
  - 在组相联或直接映象Cache中，若太多的块映象到同一组(块)中，则会出现该组中某个块被别的块替换(即使别的组或块有空闲位置)，然后又被重新访问的情况。这就是发生了冲突失效。

    (碰撞失效，干扰失效)

2. 三种失效所占的比例
   - 图示I(绝对值)
   - 图示II(相对值)



4路组相联的冲突失效对应区域为标注“4路”和“8路”两区域的合并。
2路组相联的冲突失效对应区域为标注“2路”、“4路”和“8路”三区域的合并。



3C失效中，只要采用全相联映像，就可克服冲突失效，但硬件成本昂贵。
通过增大CACHE的容量，减少容量失效。
增加块的大小，减少强制性失效。
降低失效率的方法一般会增加命中时间或失效开销。

在不同容量不同相联度的情况下，Cache 的总失效率以及“3C”所占的比例

Cache 容量(KB)	相联度	总失效率	失效率组成(相对百分比)					
			强制性失效		容量失效		冲突失效	
1	1 路	0.133	0.002	1%	0.080	60%	0.052	39%
1	2 路	0.105	0.002	2%	0.080	76%	0.023	22%
1	4 路	0.095	0.002	2%	0.080	84%	0.013	14%
1	8 路	0.087	0.002	2%	0.080	92%	0.005	6%
2	1 路	0.098	0.002	2%	0.044	45%	0.052	53%
2	2 路	0.076	0.002	2%	0.044	58%	0.030	39%
2	4 路	0.064	0.002	3%	0.044	69%	0.018	28%
2	8 路	0.054	0.002	4%	0.044	82%	0.008	14%
4	1 路	0.072	0.002	3%	0.031	43%	0.039	54%
4	2 路	0.057	0.002	3%	0.031	55%	0.024	42%
4	4 路	0.049	0.002	4%	0.031	64%	0.016	32%
4	8 路	0.039	0.002	5%	0.031	80%	0.006	15%
8	1 路	0.046	0.002	4%	0.023	51%	0.021	45%
8	2 路	0.038	0.002	5%	0.023	61%	0.013	34%

Cache 容量(KB)	相联度	总失效率	失效率组成(相对百分比)					
			强制性失效		容量失效		冲突失效	
8	4 路	0.035	0.002	5%	0.023	66%	0.010	28%
8	8 路	0.029	0.002	6%	0.023	79%	0.004	15%
16	1 路	0.029	0.002	7%	0.015	52%	0.012	42%
16	2 路	0.022	0.002	9%	0.015	68%	0.005	23%
16	4 路	0.020	0.002	10%	0.015	74%	0.003	17%
16	8 路	0.018	0.002	10%	0.015	80%	0.002	9%
32	1 路	0.020	0.002	10%	0.010	52%	0.008	38%
32	2 路	0.014	0.002	14%	0.010	74%	0.002	12%
32	4 路	0.013	0.002	15%	0.010	79%	0.001	6%
32	8 路	0.013	0.002	15%	0.010	81%	0.001	4%
64	1 路	0.014	0.002	14%	0.007	50%	0.005	36%
64	2 路	0.010	0.002	20%	0.007	70%	0.001	10%
64	4 路	0.009	0.002	21%	0.007	75%	0.000	3%
64	8 路	0.009	0.002	22%	0.007	78%	0.000	0%
128	1 路	0.010	0.002	20%	0.004	40%	0.004	40%
128	2 路	0.007	0.002	29%	0.004	58%	0.001	14%
128	4 路	0.006	0.002	31%	0.004	61%	0.001	8%
128	8 路	0.006	0.002	31%	0.004	62%	0.000	7%

- 可以看出：
  - 相联度越高，冲突失效就越少；
  - 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，但容量失效却随着容量的增加而减少；
  - 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2路组相联Cache的失效率。

- 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

    （抖动现象）高一级存储器容量比程序所需空间小得多，就出现抖动现象。
  - 冲突失效：提高相联度

    （理想情况：全相联，就不会发生冲突失效。但用硬件实现昂贵，并可能降低处理器的时钟频率，导致整体性能下降）
- 许多降低失效率的方法会增加命中时间或失效开销

5.3.1 增加Cache块大小

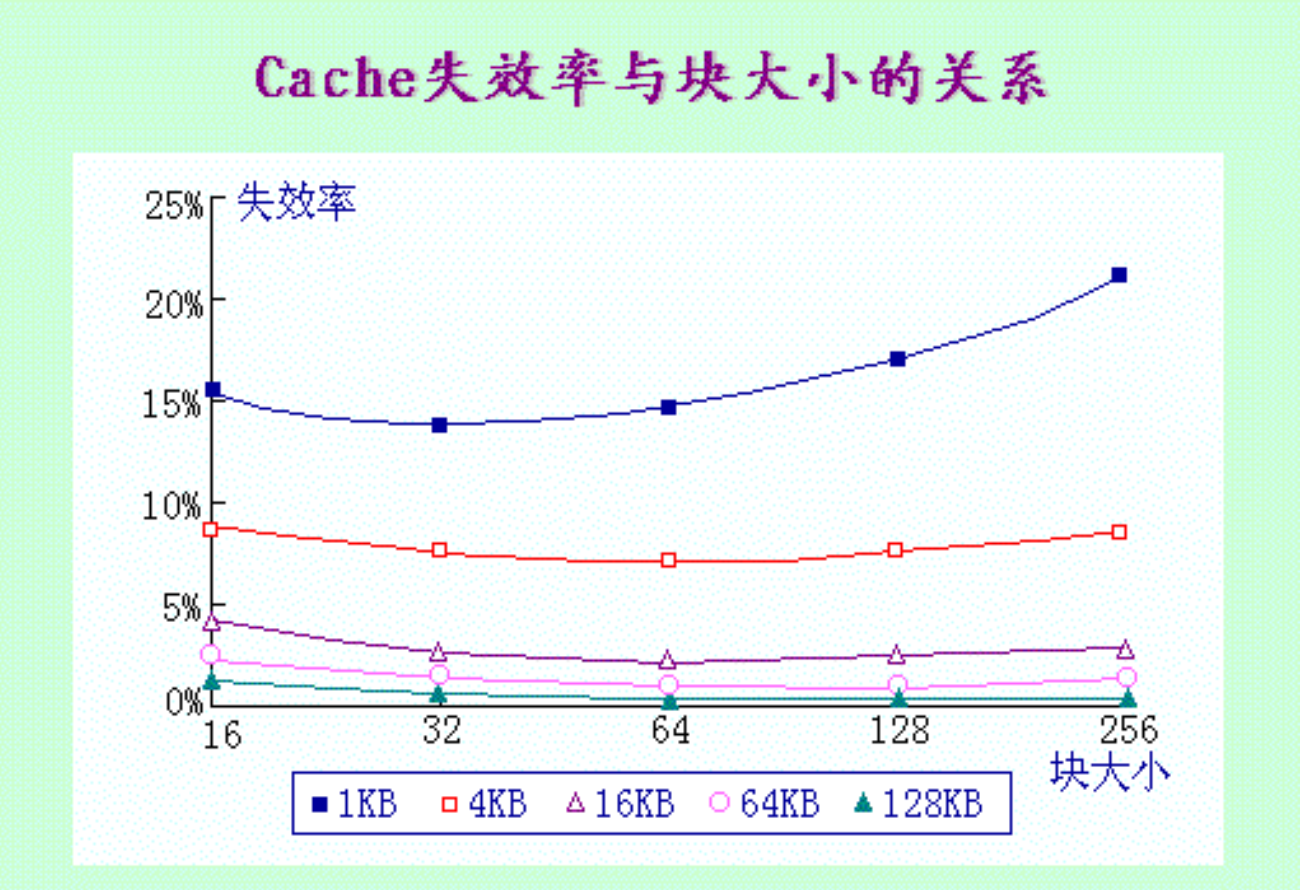
1. 失效率与块大小的关系

- 对于给定的Cache容量，当块大小增加时，失效率开始是下降，后来反而上升了。

  原因：

  - 一方面它减少了强制性失效；
  - 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失效。

- Cache容量越大，使失效率达到最低的块大小就越大。



增加Cache块的大小产生双重作用。减少强制失效，原因是时间与空间局部性，增加块大小利用空间局部性；但同时减少块的数目，增加了冲突失效的可能性。也增加失效的开销。

- 各种块大小情况下Cache的失效率

块大小（字节）	Cache容量（字节）				
	1K	4K	16K	64K	256K
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%

2. 增加块大小会增加失效开销

例5.4 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。请问对于表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？

解

解题过程

1 KB、4 KB、16 KB Cache：块大小＝32 B

64 KB、256 KB Cache：块大小＝64 B

块16B，容量1KB 平均访存时间=1+（15.05%*42)=7.321周期

块256B,容量256KB 平均访存时间=1+(0.49%*72)=1.353周期

各种块大小情况下Cache的平均访存时间

块大小（字节）	失效开销（时钟周期）	Cache容量（字节）				
		1K	4K	16K	64K	256K
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

对于失效开销，块大小取决于下一级存储器的延迟和带宽，大延迟和高带宽应选择大的CACHE块。失效时，增加一点开销，可以获得较多数据。

5.3.2 提高相联度

1. 采用相联度超过8的方案的实际意义不大。
2. 2:1 Cache经验规则

   容量为N的直接映象Cache的失效率和容量为N/2的2路组相联Cache的失效率差不多相同。
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，2路组相联：

     增加10%
   - 定制的CMOS Cache，2路组相联：

     增加2%

为实现高的处理器时钟频率，需要设计简单的CACHE，但频率高，失效开销所需时钟周期数就多。

例5.5 假定提高相联度会按下列比例增大处理器时钟周期：

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

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

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

假定命中时间为一个时钟周期，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。使用表5.5中的失效率，试问当Cache为多大时，以下不等式成立？

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

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

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

解 在各种相联度的情况下，平均访存时间分别为：

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

$$= 1.14 + \text{失效率}_{8\text{路}} \times 50$$

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

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

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

把相应的失效率代入上式，即可得平均访存时间。

例如，1 KB的直接映象Cache的平均访存时间为：

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

128 KB的8路组相联Cache的平均访存时间为：

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

在各种容量和相联度情况下Cache的平均访存时间

Cache容量	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

当Cache容量不超过16 KB时，上述三个不等式成立。

从32 KB开始，对于平均访存时间有：

4路组相联的平均访存时间小于2路组相联的；

2路组相联的小于直接映象的；

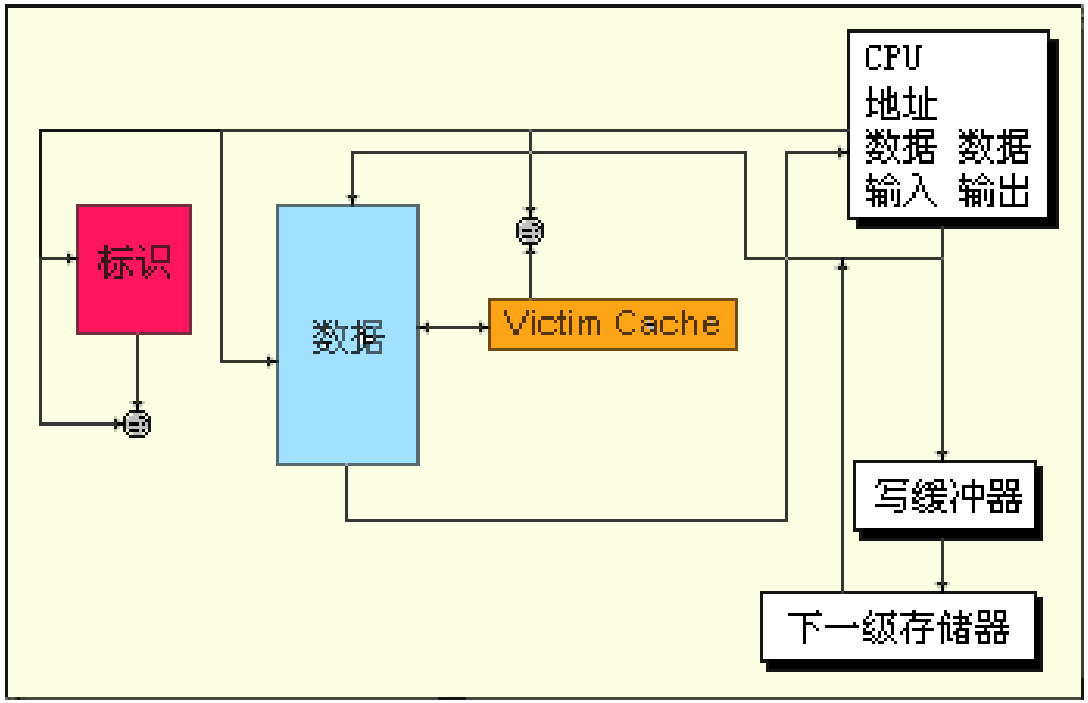
但8路组相联的却比4路组相联的大。

5.3.3 增加Cache的容量

1. 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
2. 这种方法在片外Cache中用得比较多

5.3.4 Victim Cache

1. 一种能减少冲突失效次数而又不影响时钟频率的方法。
2. 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



Victim Cache在存储层次中的位置

存放因冲突而被替换出去的那些块(Victim)，当发生失效时，在访问下一级存储器时，先检查Victim Cache是否有需要的块。有，就与Cache中的某个块交换。

3. 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:

能使4 KB Cache的冲突失效减少20%～90%

5.3.5 伪相联 Cache

1. 多路组相联的低失效率和直接映象的命中速度

	优　点	缺　点
直接映象	命中时间小	失效率高
组相联	失效率低	命中时间大

2. 伪相联Cache的优点：既有多路组相联映像的低失效率，又有直接映像的命中速度。

- 命中时间小
- 失效率低

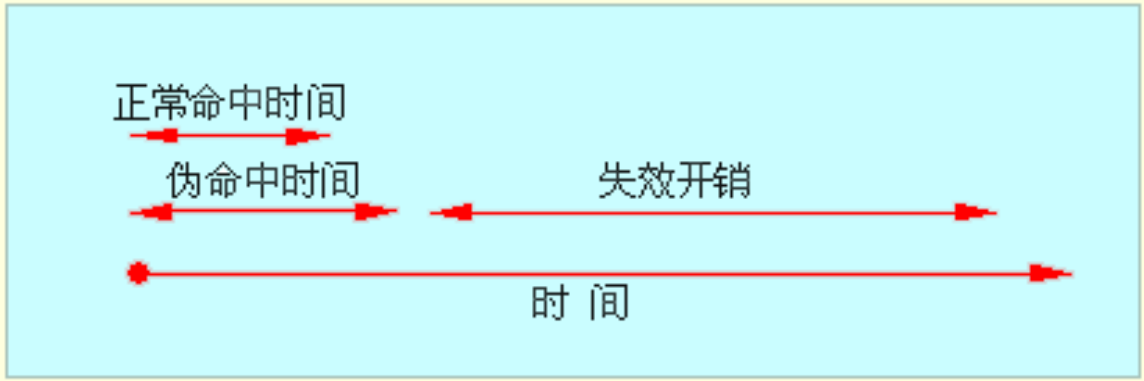
3. 基本思想及工作原理 （动画演示）

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。

4. 快速命中与慢速命中

要保证绝大多数命中都是快速命中。如何处理同一组中的两个块，哪一个是快命中，哪一个慢命中？

发生伪命中时，就交换两个块的内容，最近访问的内容放在第一个块位上。



例5.6 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。仍用上个例子中的数据，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？

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：

$$失效率_{伪相联}=失效率_{2路}$$

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

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。

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

综合上述分析，有：

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

将前面表中的数据代入上面的公式，得：

$$\text{平均访存时间}_{\text{伪相联，2 KB}}$$

$$=1+(0.098-0.076)\times 2+(0.076\times 50)=4.844$$

$$\text{平均访存时间}_{\text{伪相联，128 KB}}$$

$$=1+(0.010-0.007)\times 2+(0.007\times 50)=1.356$$

根据上一个例子中的表，对于2 KB Cache，可得：

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

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

对于128KB的Cache有，可得：

平均访存时间$_{1路}$ ＝1.50 个时钟

平均访存时间$_{2路}$ ＝1.45 个时钟

可见，对于这两种Cache容量，伪相联Cache都是速度最快的。

5. 缺点：

多种命中时间

使CPU流水线设计更加复杂，常使用在离处理器远的二级Cache。

5.3.6 硬件预取

1. 指令和数据都可以预取，即在处理器提出访问请求前
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. 指令预取通常由Cache之外的硬件完成
4. 预取效果
   - Joppi的研究结果
     - 指令预取（4 KB，直接映象Cache，块大小＝16 B）
       - 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（4 KB, 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

➢ 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64 KB四路组相联Cache来说：
  - 8个流缓冲器能捕获50%～70%的失效

例5.7 Alpha AXP 21064采用指令预取技术，其实际失效率是多少？若不采用指令预取技术，Alpha AXP 21064的指令Cache必须为多大才能保持平均访存时间不变？

解 假设当指令不在指令Cache里，而在预取缓冲器中找到时，需要多花一个时钟周期。

下面是修改后的公式：

$$\text{平均访存时间}_{\text{预取}} = \text{命中时间} + \text{失效率} \times \text{预取命中率} \times 1 + \text{失效率} \times (1 - \text{预取命中率}) \times \text{失效开销}$$

假设预取命中率为25%，命中时间为2个时钟周期，失效开销为50个时钟周期。查表可知8 KB指令Cache的失效率为1.10%。则：

平均访存时间$_{预取}$

＝2＋（1.10 %×25 %×1）＋1.10 %×（1－25 %）×50

＝2＋0.00275＋0.413 ＝2.415

为了得到相同性能下的实际失效率，由原始公式得：

平均访存时间 ＝命中时间＋失效率×失效开销

失效率 ＝（平均访存时间－命中时间）/失效开销

＝（2.415－2）/50＝0.83 %

失 效 率 的 比 较

容 量	指令 Cache	数据 Cache	混合 Cache
1 KB	3.06%	24.61%	13.34%
2 KB	2.26%	20.57%	9.78%
4 KB	1.78%	15.94%	7.24%
8 KB	1.10%	10.19%	4.57%
16 KB	0.64%	6.47%	2.87%
32 KB	0.39%	4.82%	1.99%
64 KB	0.15%	3.77%	1.36%
128 KB	0.02%	2.88%	0.95%

5.3.7 编译器控制的预取

在编译时加入预取指令，在数据被用到之前发出预取请求。

1. 预取有以下几种类型：

- 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。
- 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，也称非绑定预取。

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. 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。

4. 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，调度好预取与执行重叠
- 失效开销大时：循环体展开许多次，便于较远的循环预取数据

例5.8 对于下面的程序，首先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8 B。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

- 计算过程
- 失效情况：a有行100列，命中与失效各3*100/2=150次；b的失效为当i=0时，对所有b[j][0]的访问失效100次，当j=0时第一次对b[j+1][0]的访问失效1次。

  总的失效次数＝251次
- 改进后的程序
  - 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 ） */
      }
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 ];
      }
```

- 失效情况：预取数据从a[i][7]~a[i][99]，从b[7][0]~b[100][0]。失效次数：第一个循环a[0][0]~a[0][6]失效[7/2]=4次；b[0][0]~b[6][0]失效7次；第二个循环a[i][0]~a[i][6]失效[7/2]*2=8次。

总的失效次数＝19次

例5.9 在以下条件下，计算例5.8中所节约的时间：

(1) 忽略指令Cache失效，并假设数据Cache无冲突失效和容量失效。

(2) 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。

(3) 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）。

(4) 一次失效需50个时钟周期。

解

- 修改前：

  循环时间＝300×7 ＝2100

  失效开销＝251×50＝12550/14650

  2100＋12550＝14650

- 修改后：

  循环时间＝100×9＋200×8＝2500

  失效时间＝19×50＝950

  2500＋950＝3450

加速比＝14650/3450＝4.2

5.3.8 编译器优化

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。

McFaring 发现：

通过对指令进行重新排序，可有效地降低指令Cache的失效率。

- 2KB Cache：降低50%
- 8KB Cache：降低75%

数据对存储位置的限制比指令的少，因此更便于优化，调整顺序，改善数据的空间与时间局部性。

- 通过把数据重新组织，使得一块数据在被从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)。

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

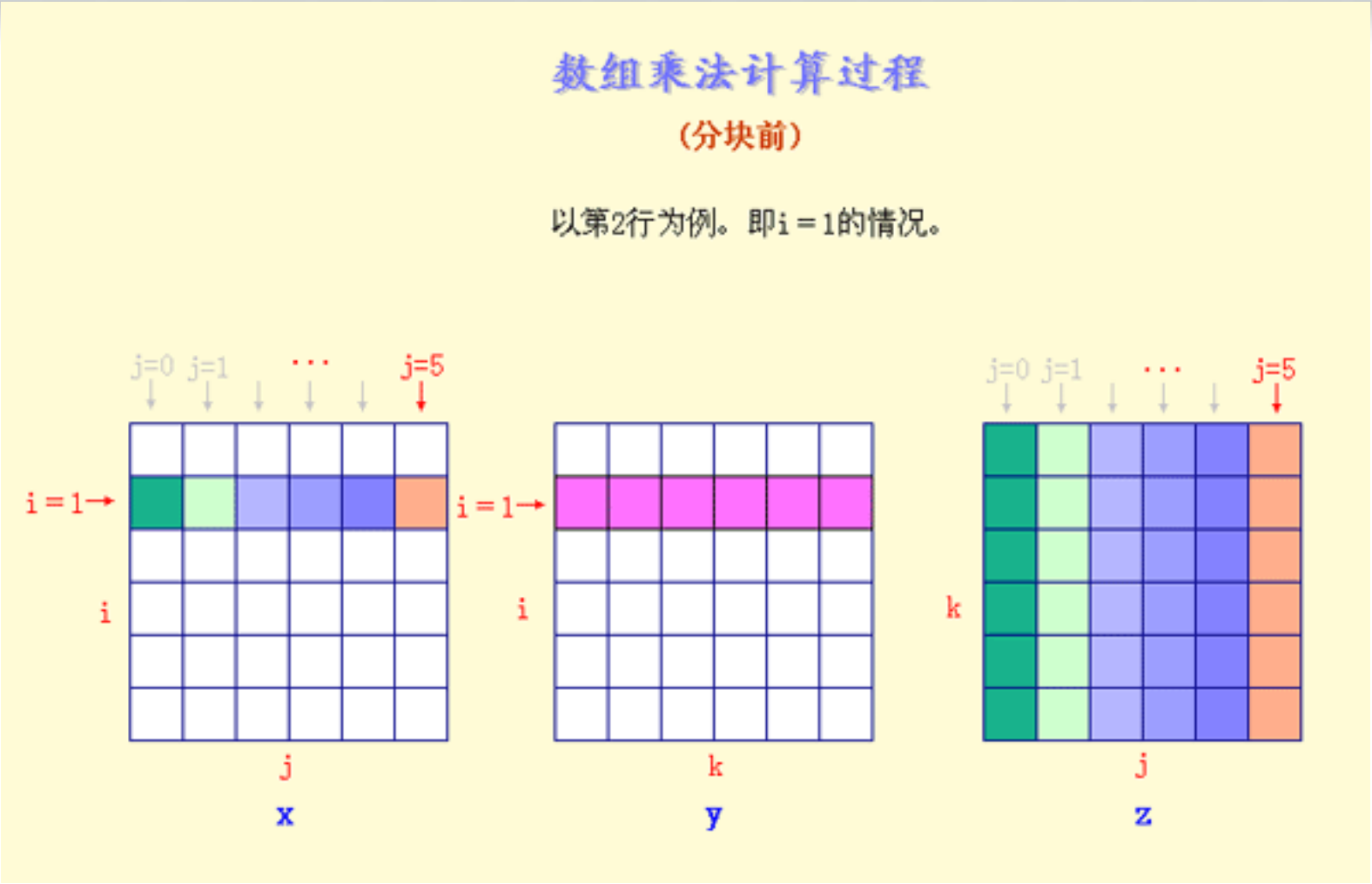
```
/* 修改前 */
    for （i = 0; i < N; i = i+1）
        for （j = 0; j < N; j = j+1）
            a[i][j] = 1/b[i][j] * c[i][j];
    for （i = 0; i < N; i = i+1）
        for （j = 0; j < N; j = j+1）
            d[i][j] = a[i][j] + c[i][j];
/* 修改后 */
    for （i = 0; i < N; i = i+1）
        for （j = 0; j < N; 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。

```
/*  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：$2N^3+N^2$）

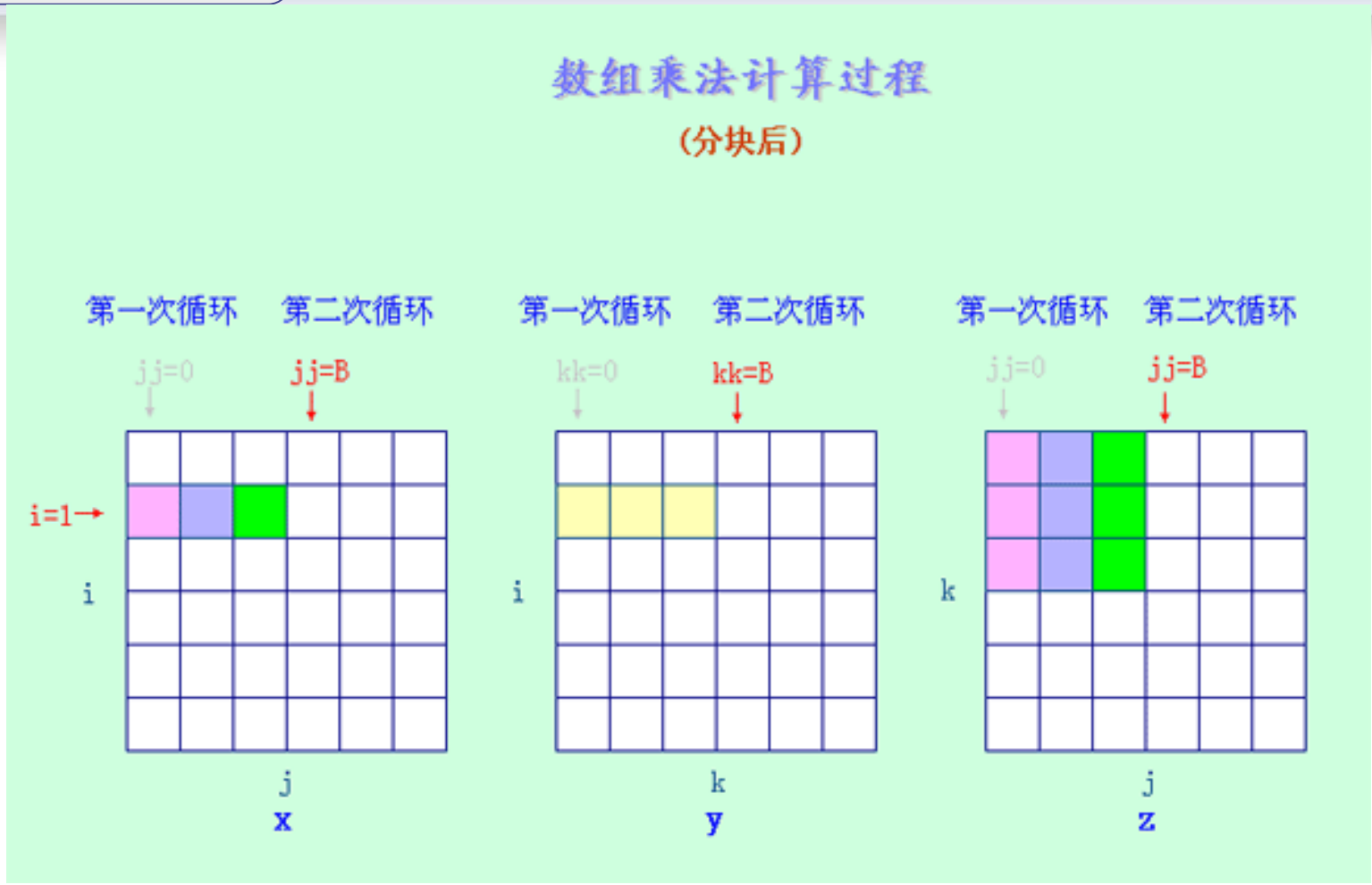


失效次数取决于N和CACHE容量，如CACHE只能放下一个N*N的数组和一行N个元素，则至少数组y的第i行和数组z的全部元素能同时放在CACHE中，如果容量太小，▲对数组x和z都可能失效。最坏失效次数达到 **2N^3＋N^2**

/* 修改后 */为保证元素在CACHE命中,大小改为B*B(分块因子)子数组进行计算,克服从x、z第一个元素到最后一个缺点.

```
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：$2N^3/B + N^2$）



访问元素个数减少。只会发生容量失效，访问存储器总次数**2N^3/B+N^2**,为原来的1/B。

分块技术利用空间与时间局部性，对y的访问利用空间局部性，z的访问利用时间局部性

5.4 减少Cache失效开销

5.4.1 让读失效优先于写

1. Cache中的写缓冲器导致对存储器访问的复杂化

写缓冲器进行的写入操作是滞后进行的，所以该缓冲器也被称为后行写数缓冲器。

例5.10 考虑以下指令序列：

```
SW   R3, 512（R0）        ；M[512]←R3     （Cache索引为0）
LW   R1, 1024（R0）       ；R1←M[1024]    （Cache索引为0）
LW   R2, 512（R0）        ；R2←M[512]     （Cache索引为0）
```

2. 解决问题的方法(读失效的处理)

- 推迟对读失效的处理,直至写缓冲器清空.

  （缺点：读失效的开销增加，如50%）

- 检查写缓冲器中的内容,如没有冲突且存储器可访问,就可继续处理失效.

3. 在写回法Cache中，也可采用写缓冲器。先把被替换的块复制到一个缓冲器中,然后读存储器,最后再写存储器.

5.4.2 写缓冲合并

1. 提高写缓冲器的效率
2. 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。

提高了写缓冲器的空间利用率，而且还能减少因写缓冲器满而要进行的等待时间。

写地址	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	

（a）不采用写合并

写地址	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	

（b）采用了写合并

5.4.3 请求字处理技术

1. 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**。

2. 应尽早把请求字发送给CPU

- **尽早重启动**：调块时，从块的起始位置开始读起。一旦请求字到达，就立即发送给CPU，让CPU继续执行。
- **请求字优先**：调块时，从请求字所在的位置读起。这样，第一个读出的字便是请求字。将之立即发送给CPU。同时从存储器调入该块其余部分.

3. 这种技术在以下情况下效果不大：

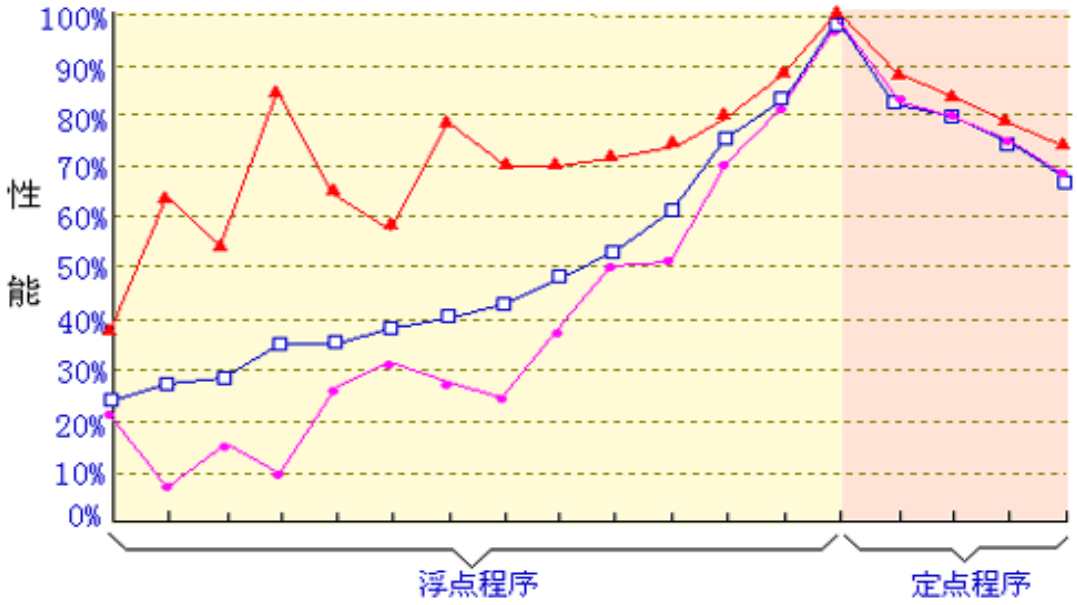
- Cache块较小
- 下一条指令正好访问同一Cache块的另一部分，只能节省一个时针周期.

5.4.4 非阻塞Cache技术

1. 非阻塞Cache：Cache失效时仍允许CPU进行其他的命中访问。即允许“失效下命中”。
2. 进一步提高性能,减少失效的开销：
   - “多重失效下命中”
   - “失效下失效”

   （存储器必须能够处理多个失效）
3. 重叠失效个数对平均访问时间的影响

“多重失效下命中”Cache的平均存储器等待时间与阻塞Cache的平均等待时间的比值



	Hit under 1 miss	Hit under 2 miss	Hit under 64 miss
浮点程序(平均值)	76%	51%	39%
定点程序(平均值)	81%	78%	78%

1. 对于浮点程序,重叠失效个数越多,性能提高就越多
2. 但对于整数程序,重叠次数对性能提高影响不大.

例5.11 对于上图描述的Cache，在2路组相联和“一次失效下命中”这两种措施中，哪一种对浮点程序更重要？对整数程序的情况如何？

假设8KB数据Cache的平均失效率为：对于浮点程序，直接映象Cache为11.4%，而2路组相联Cache为10.7%；对于整数程序，直接映象Cache为7.4%，2路组相联Cache为6.0%。并且假设平均存储器等待时间是失效率和失效开销的积，失效开销为16个时钟周期。

解 对于浮点程序，平均存储器等待时间为：

$$失效率_{直接映象} \times 失效开销 = 11.4\% \times 16 = 1.84$$

$$失效率_{2路组相联} \times 失效开销 = 10.7\% \times 16 = 1.71$$

1.71/1.84≈0.93（2路的等待时间是直接的93%）

对于浮点程序,支持”一次失效下命中”的直接映像CACHE等待时间是直接映像的76%，因此支持”一次失效下命中”的直接映像比2路组相联的CACHE性能更高.

对于整数程序：

$$失效率_{直接映象} \times 失效开销 = 7.4\% \times 16 = 1.18$$

$$失效率_{2路组相联} \times 失效开销 = 6.0\% \times 16 = 0.96$$

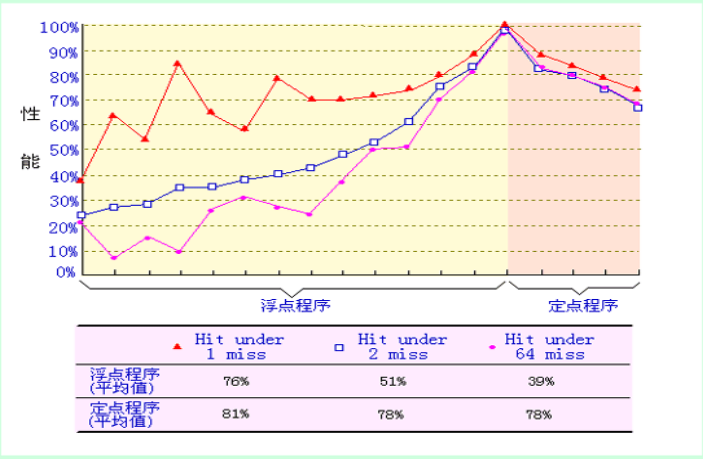
0.96/1.18≈0.81

对于整数程序,”一次失效下命中”把存储时间降低到直接映像CACHE的81%. 性能相同.

“失效下命中”方法有一个潜在优点：

它不会影响命中时间，而组相联却会。

“多重失效下命中”Cache的平均存储器等待时间与阻塞Cache的平均等待时间的比值



	Hit under 1 miss	Hit under 2 miss	Hit under 64 miss
浮点程序(平均值)	76%	51%	39%
定点程序(平均值)	81%	78%	78%

5.4.5 采用两级Cache

1. 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

2. 性能分析

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$_{L2}$

平均访存时间 ＝ 命中时间$_{L1}$＋失效率$_{L1}$×

（命中时间$_{L2}$＋失效率$_{L2}$×失效开销$_{L2}$）

3. 局部失效率与全局失效率

- 局部失效率＝该级Cache的失效次数/到达该级Cache的访问次数

  例如：上述式子中的失效率L2

- 全局失效率＝该级Cache的失效次数/CPU发出的访存的总次数

- 全局失效率$_{L2}$＝失效率$_{L1}$×失效率$_{L2}$

  评价第二级Cache时，应使用全局失效率这个指标。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。

例5.12 假设在1000次访存中，第一级Cache失效40次， 第二级Cache失效20次。试问：（1）在这种情况下，该Cache系统的局部失效率和全局失效率各是多少？（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

第一级Cache的失效率（全局和局部）是40/1000，即4%；

第二级Cache的局部失效率是20/40，即50%；

第二级Cache的全局失效率是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$_{L2}$）-------访存的平均停顿时间

＝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＝平均每条指令访存1.5次*每次访存的平均停顿时间2.4个时钟周期=2.4×1.5＝3.6个时钟周期

4. 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的全局失效率和容量与第二级Cache相同的单级Cache的失效率非常接近。
   - 局部失效率不是衡量第二级Cache的一个好指标，因此，在评价第二级Cache时，应用全局失效率这个指标。
5. 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。
   - 两个问题：

- 它能否降低CPI中的平均访存时间部分？
- 它的成本是多少？

6. 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。消除容量失效.存在强制性失效和冲突失效.

如512 KB，1024 KB

➢ 相联度

第二级Cache可采用较高的相联度或伪相联方法。

例5.13 给出有关第二级Cache的以下数据：

（1） 2路组相联使命中时间增加10%×CPU时钟周期

（2） 对于直接映象，命中时间$_{L2}$ ＝ 10个时钟周期

（3） 对于直接映象，局部失效率$_{L2}$ ＝ 25%

（4） 对于2路组相联，局部失效率$_{L2}$ ＝ 20%

（5） 失效开销$_{L2}$ ＝ 50个时钟周期

试问第二级Cache的相联度对失效开销的影响如何？

解 对一个直接映象的第二级Cache来说，第一级Cache的失效开销为：

$$失效开销_{直接映象，L1} = 10+25\%\times 50 = 22.5 \text{ 个时钟周期}$$

对于2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的失效开销为：

$$失效开销_{2路组相联，L1} = 10.1+20\%\times 50 = 20.1 \text{ 个时钟周期}$$

把第二级Cache的命中时间取整，得10或11，则：

$$失效开销_{2路组相联，L1} = 10+20\%\times 50 = 20.0 \text{ 个时钟周期}$$

$$失效开销_{2路组相联，L1} = 11+20\%\times 50 = 21.0 \text{ 个时钟周期}$$

故对于第二级Cache来说，2路组相联优于直接映象。

- 块大小
  - 第二级Cache可采用较大的块

    如 64 B、128 B、256 B
  - 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。

    为减少平均访存时间,容量小的第一级CACHE采用较小的块,容量大的第二级CACHE采用较大的块.可以实现包容性,但处理第二级失效时,即替换第二级CACHE的块时,必须作废第一级对应的块.

5.5 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。

5.5.1 容量小、结构简单的Cache

1. 硬件越简单，速度就越快。
2. 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中方案：

把Cache的标识放在片内，而把Cache的数据存储器放在片外。如直接映像CACHE让标识检测和数据传送重叠执行，减少命中时间

5.5.2 虚拟Cache

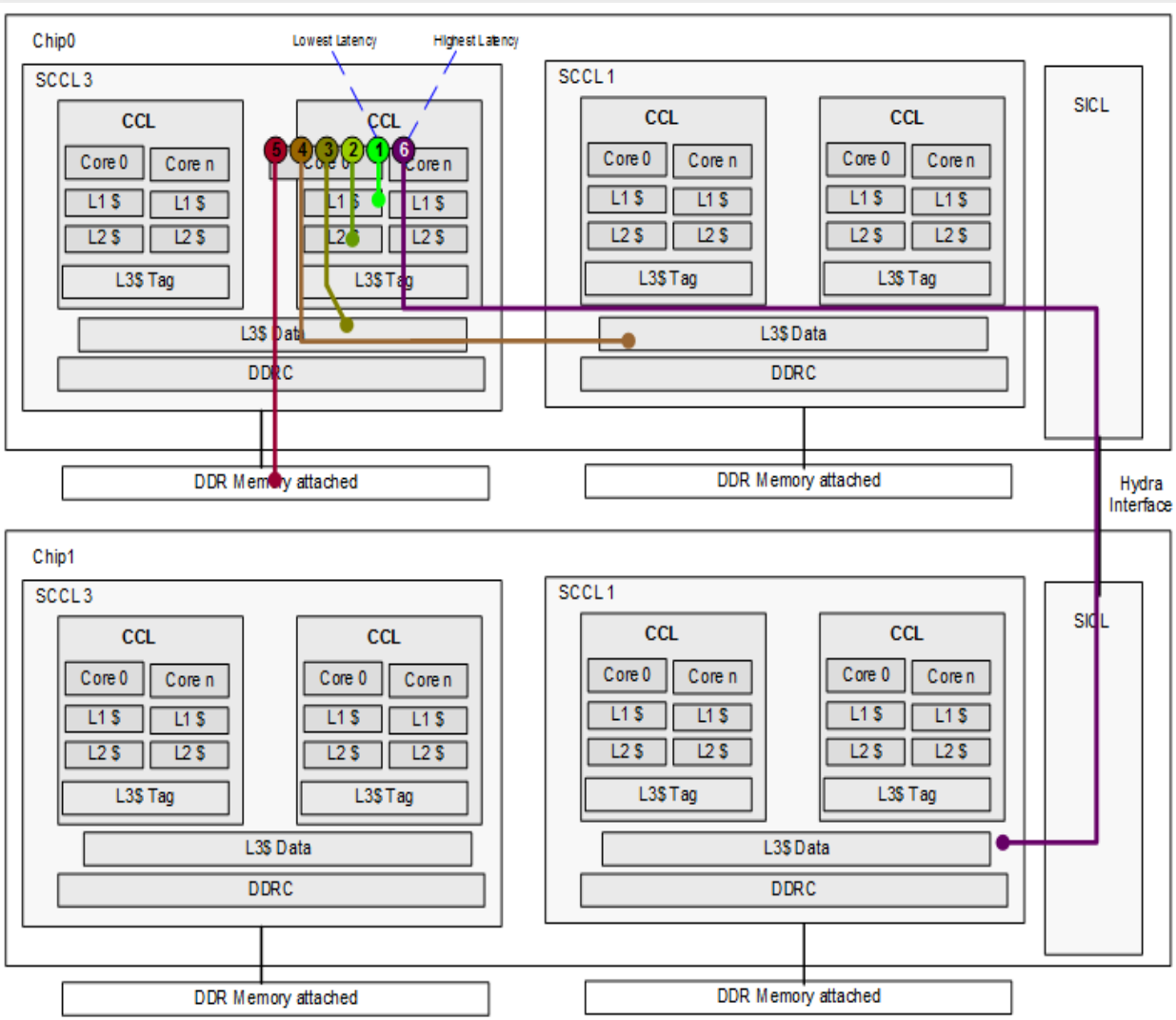
1. 虚拟Cache

访问Cache的索引以及Cache中的标识都是虚拟地址（一部分）。

2. 物理Cache

使用物理地址的传统Cache。

鲲鹏920系列芯片架构——内存子系统 (2)



- 一个CPU Die包含4个DDR Channel
- 一个Socket包含2个CPU Die，8个DDR Channel
- 每个控制器支持2DPC 2933
- 本地内存访问均在本地进行，不走片间互联总线，因此访存时延最小，总体性能最好。

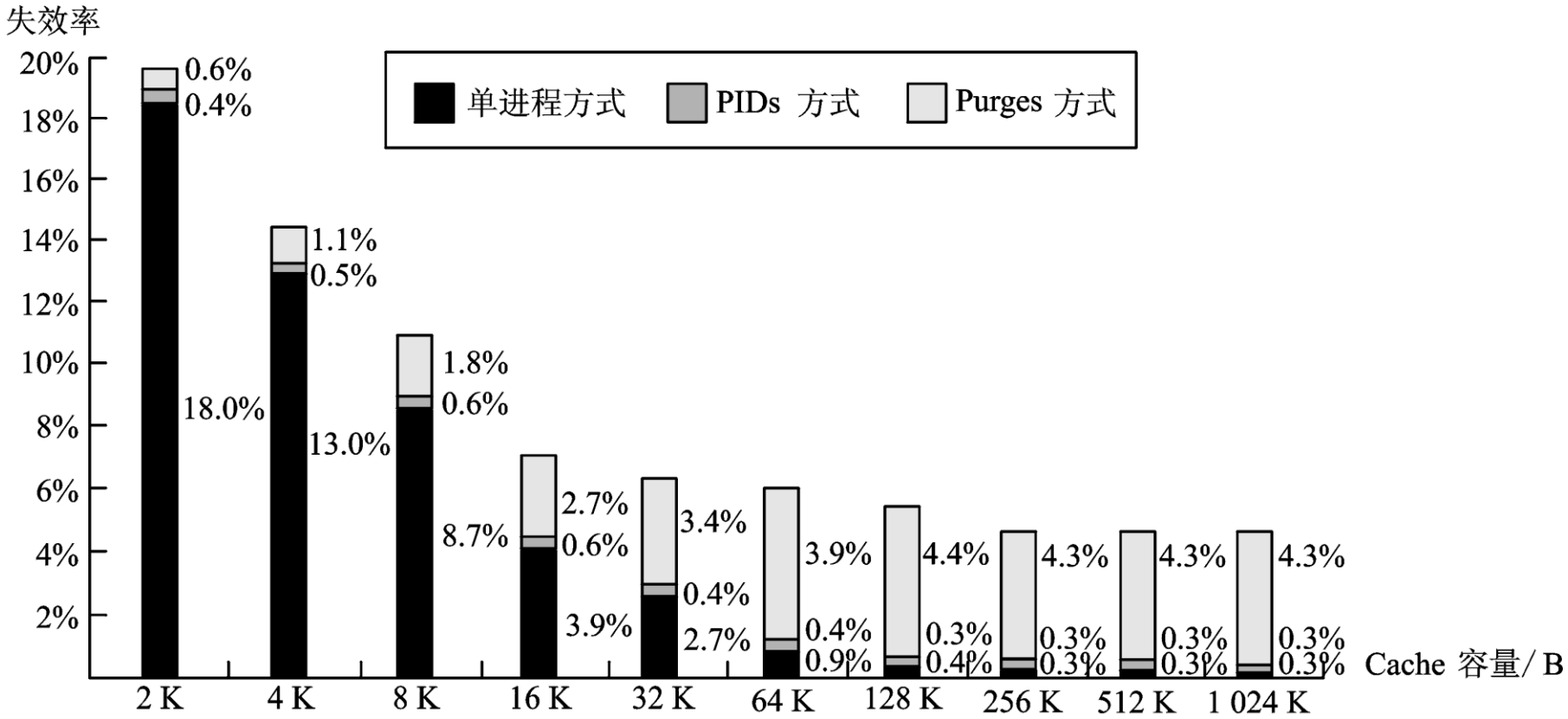
时延	ARM CPU Cycles	Skylake Cycles
Register	1	1
L1 cache	4	4
L2 cache	8	14
L3 cache	40	55
DRAM	71 -221(ns)	83-143(ns)

3. 并非都采用虚拟Cache(为什么?)

- 虚拟Cache的清空问题
  - 解决方法：在地址标识中增加PID字段
    (进程标识符)
  - 三种情况下失效率的比较
    - 单进程，PIDs，清空
    - PIDs与单进程相比：＋0.3%～＋0.6%
    - PIDs与清空相比：　－0.6%～－4.3%
- 同义和别名

  解决方法：反别名法、页着色

  同一个物理地址可能采用两种或以上的虚拟地址,造成同一数据在虚拟**CACHE**中有多个副本,形成不一致情况,为保证不发生错误,就要求把虚拟地址转换为物理地址,找到物理**CACHE**块.

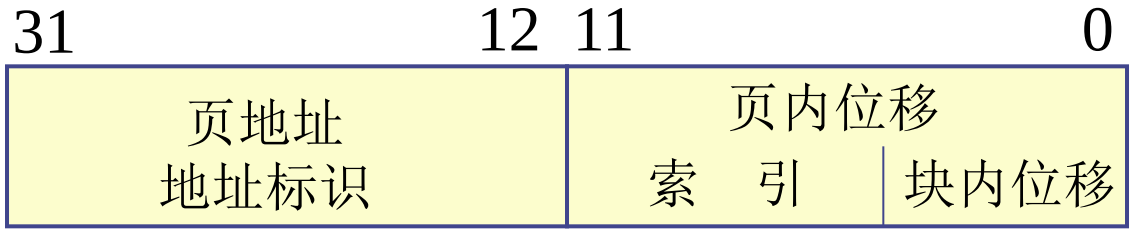

失效率
20%
18%
16%
14%
12%
10%
8%
6%
4%
2%
单进程方式
PIDs 方式
Purges 方式
0.6%
0.4%
18.0%
1.1%
0.5%
13.0%
1.8%
0.6%
8.7%
2.7%
0.6%
3.9%
3.4%
0.4%
2.7%
3.9%
0.4%
0.9%
4.4%
0.3%
0.4%
4.3%
0.3%
0.3%
4.3%
0.3%
0.3%
4.3%
0.3%
0.3%
2 K
4 K
8 K
16 K
32 K
64 K
128 K
256 K
512 K
1 024 K
Cache 容量/ B

4. 虚拟索引＋物理标识
   - 优点：兼得虚拟Cache和物理Cache的好处
   - 局限性：Cache容量受到限制

(页内位移)

Cache容量=2^index*相联度*块大小≤页大小×相联度

5. 举例：IBM 3033的Cache
   - 页大小＝4 KB　　相联度＝16



- Cache容量＝16×4KB＝64KB

6. 另一种方法：硬件散列变换

5.5.3 Cache访问流水化

1. 对第一级Cache的访问按流水方式组织
2. 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

提高时钟频率,实际上不能真正减少CACHE的命中时间,但提高访问CACHE的带宽.

5.5.4 Trace Cache

1. 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。

2. 一个解决方法：采用Trace 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。

3. 优缺点

- 地址映象机制复杂。
- 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。

5.5.5 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	失效率	失效开销	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 B的块
增加Cache容量	+				被广泛采用，特别是第二级Cache
提高相联度	+		–	1	被广泛采用
Victim 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	失效率	失效开销	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失效次数	+			0	向软件提出了新要求；有些机器提供了编译器选项
使读失效优先于写		+	–	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	失效率	失效开销	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

5.6 主 存

- 主存的主要性能指标：延迟和带宽
- 以往：
  - Cache主要关心延迟，I/O主要关心带宽。
- 现在：Cache关心两者
- 下面讨论几种能提高主存性能的存储器组织技术
  - 我们以处理Cache失效为例来说明各种存储器组织结构的好处。

- 在下面的讨论中，假设基本存储器结构的性能为：
  - 送地址需要4个时钟周期
  - 每个字的访问时间为24个时钟周期
  - 传送一个字（32位）的数据需4个时钟周期

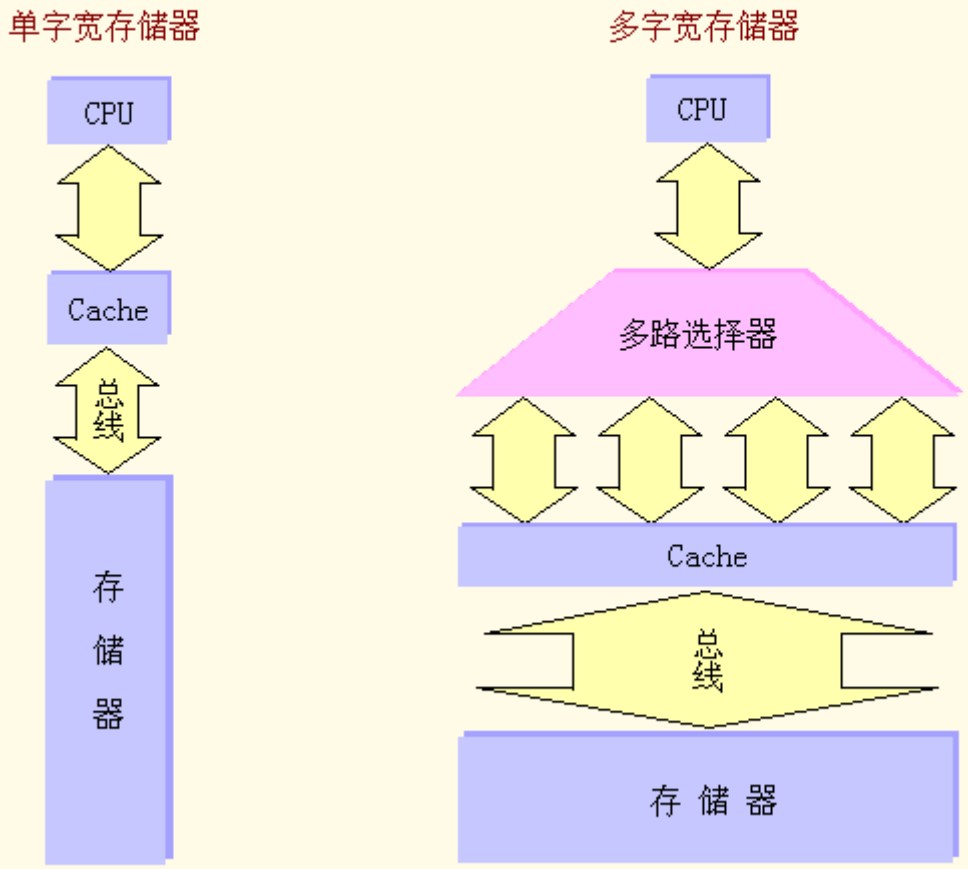
如果Cache块大小为4个字，则

- 失效开销为4×（4＋24＋4）＝128个时钟周期
- 存储器的带宽为每个时钟周期1/8（16/128）字节

1. 增加存储器的宽度

- 性能举例 (参照前面的假设)
  - 当宽度为4个字时：
    - 失效开销＝1×32(周期)
    - 带宽＝0.5(字节/周期)
- 缺点：
  - 增加CPU和存储器之间的连接通路的宽度
  - CPU和Cache之间有一个多路选择器
  - 扩充主存的最小增量增加了相应的倍数
  - 写入有可能变得复杂
- 举例：
  - DEC的Alpha Axp21064：256位宽

增加存储器的宽度



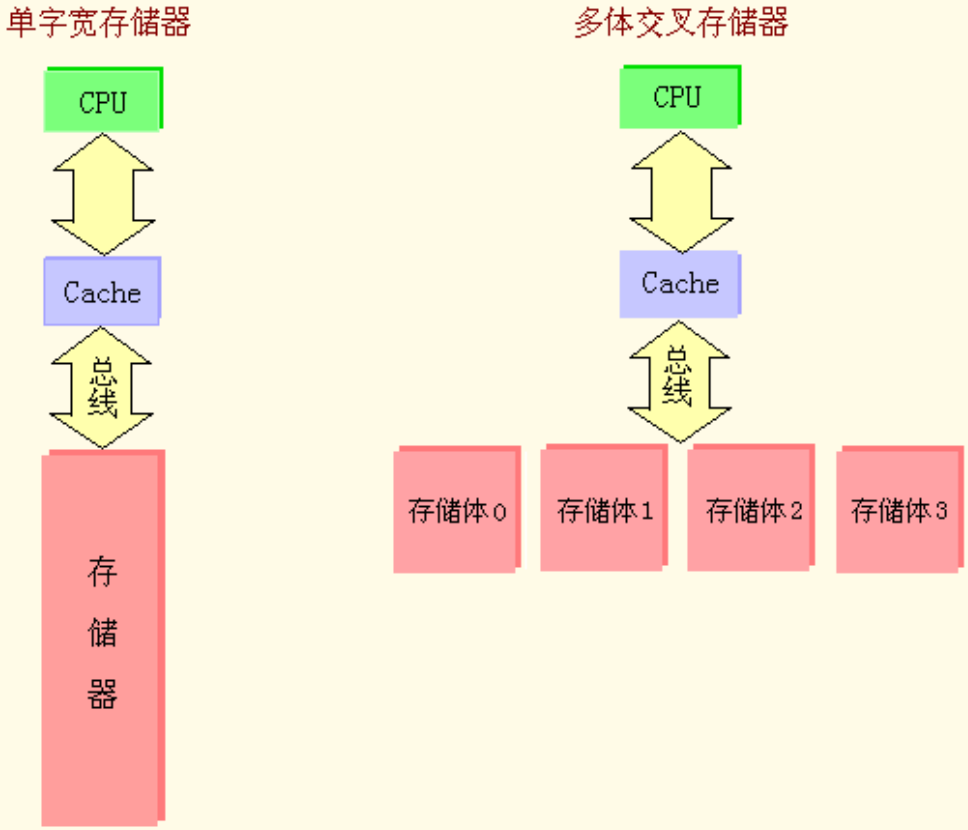
2. 采用简单的多体交叉存储器

- 在存储系统中采用多个DRAM，并利用它们潜在的并行性。
- 性能举例：(参照前面的假设)
  - 失效开销＝4＋24＋4×4＝44(周期)
  - 带宽＝0.4(字节/周期)
- 存储器的各个体一般是按字交叉的,每个存储体宽度一般一个字

  交叉存储器（interleaved memory）
  - 通常是指存储器的各个体是按字交叉的
  - 字交叉存储器非常适合于处理：

    Cache读失效，原因是所调的块中各个字是顺序读出的；写回法Cache中的写回，读和写都是顺序进行的。

多体交叉存储器


单字宽存储器
CPU
Cache
总线
存储器
多体交叉存储器
CPU
Cache
总线
存储体0
存储体1
存储体2
存储体3

假设四个存储体的地址是在字一级交叉的，即存储体0中每个字的地址对4取模都是0，体1中每个字的地址对4取模都是1，依此类推。 按字寻址, 如按字节寻址, 且每字为4B, 则把地址*4.

例5.14 假设某台计算机的特性及其Cache的性能为：

(1) 块大小为1个字；

(2) 存储器总线宽度为1个字；

(3) Cache失效率为3%；

(4) 平均每条指令访存1.2次；

(5) Cache失效开销为32个时钟周期；

(6)平均CPI（忽略Cache失效）为2。

试问多体交叉和增加存储器宽度对提高性能各有何作用？

如果当把Cache块大小变为2个字时，失效率降为2%；块大小变为4个字时，失效率降为1%。根据前面给出的访问时间，求在采用2路、4路多体交叉存取以及将存储器和总线宽度增加一倍时，性能分别提高多少？

解 在改变前的计算机中，Cache块大小为一个字，

其CPI为：

$2+(1.2\times3\%\times32) = 3.15$

- 当将块大小增加为2个字时，在下面三种情况下的CPI分别为
  - 32位总线和存储器，不采用多体交叉：

$2+(1.2\times2\%\times2\times32) = 3.54$

- 32位总线和存储器，采用多体交叉：

  2＋1.2×2%×（4＋24＋8）＝ 2.86

  性能提高了10%

- 64位总线和存储器，不采用多体交叉：

  2＋（1.2×2%×1×32） ＝ 2.77

  性能提高了14%

➢ 将块大小增加到4个字节，可以得到以下数据：

- 32位总线和存储器，不采用多体交叉：

  2＋（1.2×1%×4×32） ＝ 3.54

- 32位总线和存储器，采用多体交叉：

  2＋1.2×1%×（4＋24＋16）＝ 2.53

  性能提高了25%

- 64位总线和存储器，不采用多体交叉：

  2＋（1.2×1%×2×32） ＝ 2.77

  性能提高了14%. 因此, 多体交叉存储器在逻辑上是一种宽存储器, 只是为共享内部资源(总线), 把各存储器的访问安排在不同的时间段进行.

存储器中应该含有多少个体呢？

向量计算机采用以下衡量标准：

体的数目 ≥ 访问体中一个字所需的时钟周期数

- 存储系统的设计目标

对于顺序访问，每个时钟周期都能从一个存储体中送出一个数据。

3. 独立存储体

- 设置多个存储控制器，使多个体能独立操作，以便能同时进行多个独立的访存。
- 例如
  - 一台输入设备可能会使用某个存控，访问某个存储体；
  - Cache读操作可能在使用另一个存控，访问另一个存储体；
  - Cache写操作则可能在使用第三个存控，访问第三个存储体。
- 每个体需要有独立的地址线和独立的数据总线。

- 非阻塞Cache与多体结构
- 采用多体结构的另一个原因

  共享公共存储器多处理机系统的需求
- 独立存储体和按字交叉的多体结合起来使用
  - 将存储器分为若干个独立的存储体；
  - 而每个独立存储体内部又划分为若干个按字交叉方式工作的体。

4. 避免存储体冲突

- 体冲突：两个请求要访问同一个体。
- 减少体冲突次数的一种方法：采用许多体

  例如，NEC SX/3最多可使用128个体

这种方法存在问题：

假如有128个存储体，按字交叉方式工作，并执行以下程序：

```
int  x [ 256 ][ 512 ];
for  ( j = 0;   j < 512;   j = j+1 )
for  ( i = 0;   i < 256;   i = i+1 )
   x [ i ][ j ] = 2 * x [ i ][ j ];
```

因为512是128的整数倍，同一列中的所有元素都在同一个体内，无论CPU或存储系统多么高级，该程序都会在数据Cache失效时暂停。

- 解决体冲突的方法
  - 软件方法(编译器)
    - 循环交换优化,避免对同一个体的访问
    - 扩展数组的大小，使之不是2的幂
  - 硬件方法
    - 使体数为素数

      体内地址＝地址 mod（存储体中的字数）

      可以直接截取
    - 举例

顺序交叉和取模交叉的地址映象举例

体内地址	存储体					
	顺序交叉			取模交叉		
	0	1	2	0	1	2
0	0	1	2	0	16	8
1	3	4	5	9	1	17
2	6	7	8	18	10	2
3	9	10	11	3	19	11
4	12	13	14	12	4	20
5	15	16	17	21	13	5
6	18	19	20	6	22	14
7	21	22	23	15	7	23

5. DRAM专用交叉结构

- 介绍几种利用DRAM特性的技术

采用更宽的存储器；交叉存储器；多体结构存储器及避免冲突方法。

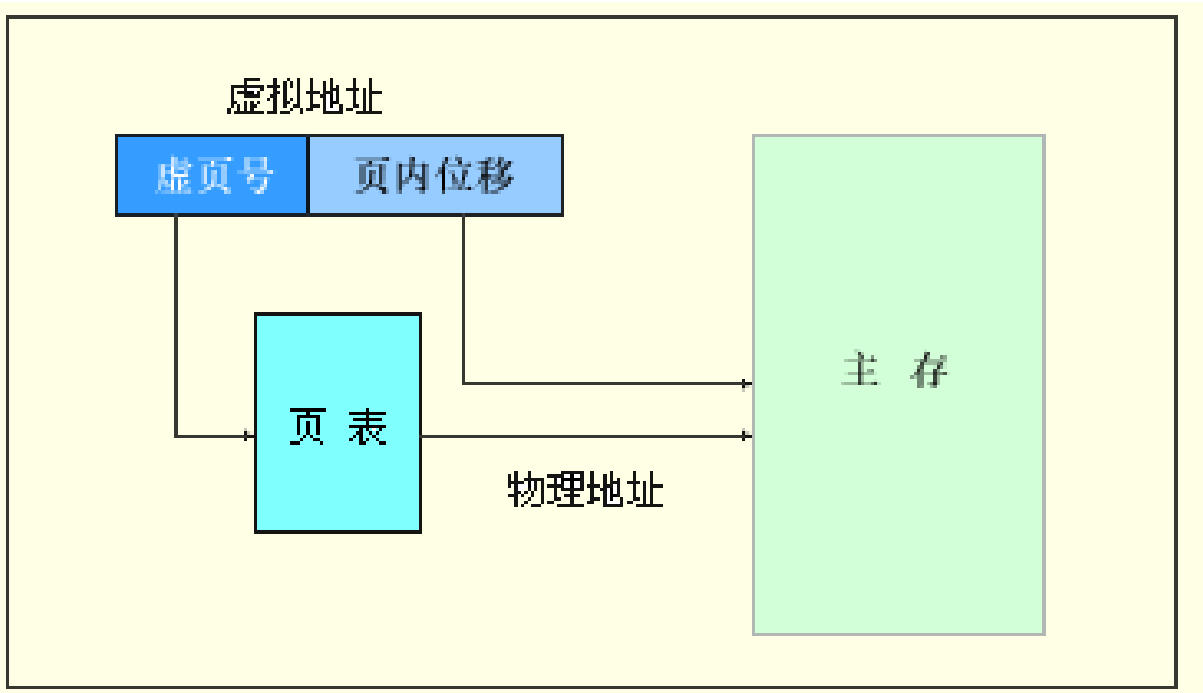
- 对DRAM的访问分为行访问和列访问，每行的位数是DRAM容量的平方根。若64Mb的DRAM,一行为8Kb；256Mb的DRAM，一行为16Kb.
- 三种优化方式
  - Nibble方式 ：每次进行行访问时，DRAM除能够给出所需的位以外，还能给出其后的3位。
  - Page方式：缓冲器以SRAM的方式工作：通过改变列地址，可以随机地访问缓冲器内的任一位。
  - Static column方式： 和Page方式类似，只是在列地址改变时，无需触发列访问选通线。

5.7 虚拟存储器

5.7.1 虚拟存储器的基本原理

1. 虚拟存储器是“主存—辅存”层次进一步发展的结果。
2. 虚拟存储器可以分为两类：页式和段式,页式虚拟存储器把空间划分为大小相同的块—页面，机械划分空间，地址单一固定，由页号和页内位移组成；段式虚拟存储器把空间划分为可变长的块—段，按逻辑意义划分，地址由两个字组成，段号和段内位移。现代计算机采用段页式管理，每段划分为若干页面。保持段为逻辑单位，简化替换。段不必整体一次调入主存，以页面为单位调入。
3. 有关虚拟存储器的四个问题
   - 映象规则：全相联,即操作系统允许信息块放在主存任一位置。
   - 查找算法：页表，段表，TLB
   - 替换算法：LRU
   - 写策略：写回法

- 用页表实现虚拟地址到物理地址的映射

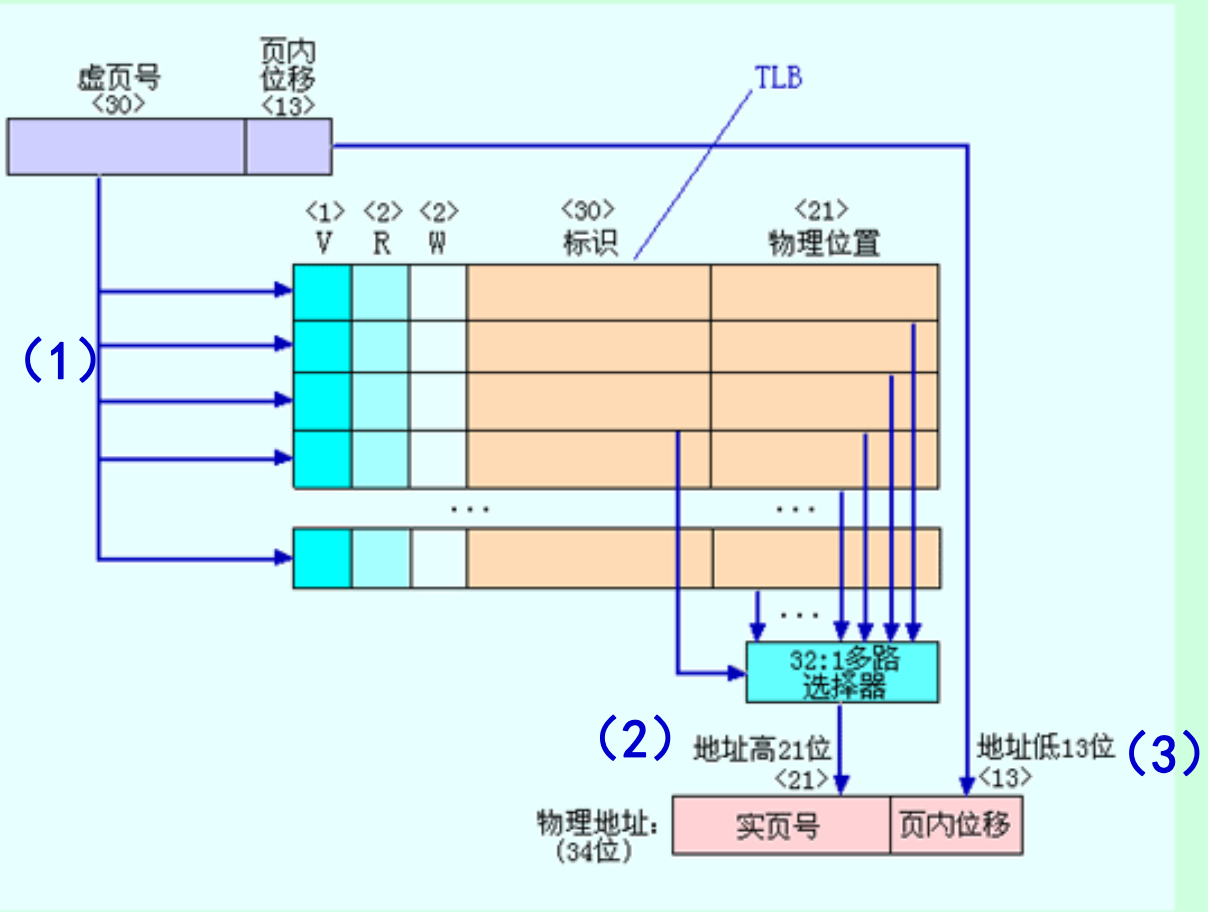


页式和段式管理使用一个由页号或段号作为索引的数据结构—页表和段表，含有所需要查找块的物理地址；对于段式系统，段内位移加上段的物理地址即为最终物理地址；对于页式系统，将页内位移拼接在相应页面的物理地址之后即可。页表用虚拟页号作索引，包含的项数与虚拟地址空间的总页面数相同。为了减少地址转换时间，常设置一个专门的地址转换高速缓存（TLB或TB）

5.7.2 快表（TLB）

1. 地址变换缓冲器TLB：每次访存需两次访问主存：读取页表项，访问数据。
   - TLB是一个专用的高速缓冲器，用于存放近期经常使用的页表项；
   - TLB中的内容是页表部分内容的一个副本；进行地址变换时，一般直接查TLB表；偶尔不命中时，才访问主存中的页表。
   - TLB也利用了局部性原理。TLB的表项与CACHE类似，由标识和数据构成。标识存放的是虚拟地址一部分；数据部分存放物理页帧号、有效位、保护信息及辅助信息等。
2. Alpha Axp 21064 的地址转换过程
3. TLB一般比Cache的标识存储器更小、更快

Alpha AXP 21064的地址转换过程



（1）把虚拟地址送往各标识，进行比较；

（2）如存在匹配的标识，则多路选择器把相应的TLB项中的物理地址选出；

（3）选出的物理地址与页内位移拼接成完整的34位物理地址。

采用小容量的CACHE，把访问CACHE的索引限制在页内范围内；虚拟地址一到，立即索引CACHE。读CACHE标识同时，把虚页号送给TLB进行地址变换。把从TLB得到物理地址与CACHE标识比较。

V=“1”，TLB有效位；R=“1”，对页面读允许

W=“1”，对页面写允许

5.8 进程保护与虚存实例

进程：程序呼吸的空气和生存的空间。

也就是说，一个正在运行的程序加上它继续执行所需的任何状态就是进程。

5.8.1 进程保护

1. 最简单的保护机制是用一对寄存器来检查每一个地址，以确保地址在两个界限之间。
   - 基地址　　上界地址
   - 检测条件：

     基地址≤地址≤上界地址

- 在有些系统中，地址被看作是无符号的整数。检测条件就变为：

  （基地址＋地址）≤上界地址

2. 计算机硬件设计者为操作系统设计者提供支持

- 提供至少两种模式
  - 区分正在运行的进程是用户进程还是操作系统进程
  - 称后者为内核进程、超级用户进程或管理进程
- 使CPU状态的一部分成为用户进程，可读但不可写

  包括：基地址/上界地址寄存器

  用户/管理模式位

  异常许可/禁止位

- 提供一种机制，使得CPU能从用户模式进入管理模式和从管理模式进入用户模式。第一种通过系统调用完成

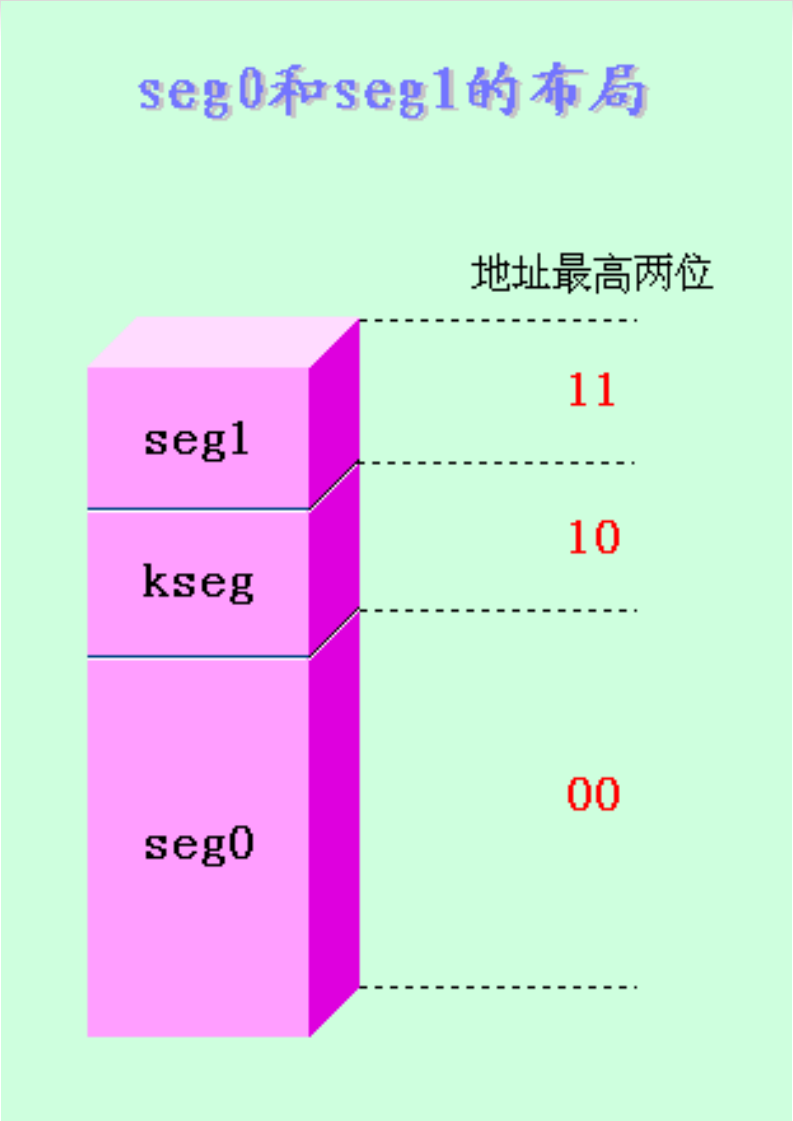
3. 虚拟存储器
   - 给每个页或段增加许可标志。页面级的保护可通过增加用户/内核保护，防止用户程序访问属于内核的页。
4. 环形保护。将访存保护由两极扩展到多级。
5. 加锁和解锁。一个程序得到钥匙后才能解锁并访问数据，并能显式地将他们从一个程序传输到另一个程序，防止程序进行伪造。

5.8.2 页式虚存举例：

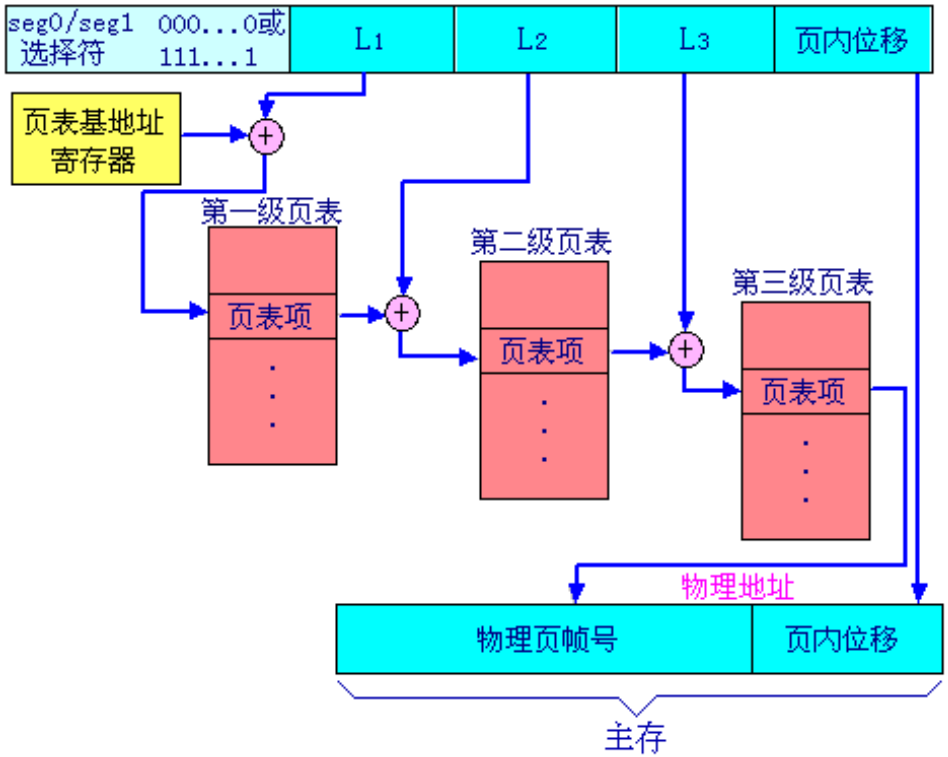
Alpha Axp的存储管理和21064的TLB

Alpha Axp体系结构采用段页相结合的方式

1. Alpha的地址空间64位分为3段：
   - kseg(地址最高两位：10) (内核)
   - seg0(最高位：0) (用户)
   - seg1(最高两位：11) (用户)
   - sego和seg1的布局：seg0地址由0向上生长， seg1从最高地址向下生长。现代系统采用空间预分段和页式管理相结合。
2. Alpha采用三级页表实现地址空间映射，虚地址三个域：I1，I2，I3，分别查找三级页表。
   - 地址变换过程
   - Alpha的页表项(PTE)


seg0和seg1的布局
地址最高两位
11
10
00
seg1
kseg
seg0

Alpha机器中虚地址的变换过程



1）将虚拟地址I1字段的值和页表基地址寄存器的值相加，再用结果作地址访问存储器，得到第二级页表的基地址；

2）将虚拟地址I2字段的值和第二级页表基地址相加，再用结果作地址访问存储器，得到第三级页表的基地址；

3）将虚拟地址I3字段的值和第三级页表基地址相加，再用结果作地址访问存储器，得到所访问页面的物理地址；

4）将所访问页面的物理地址和页内位移拼接，得到实际的物理地址。

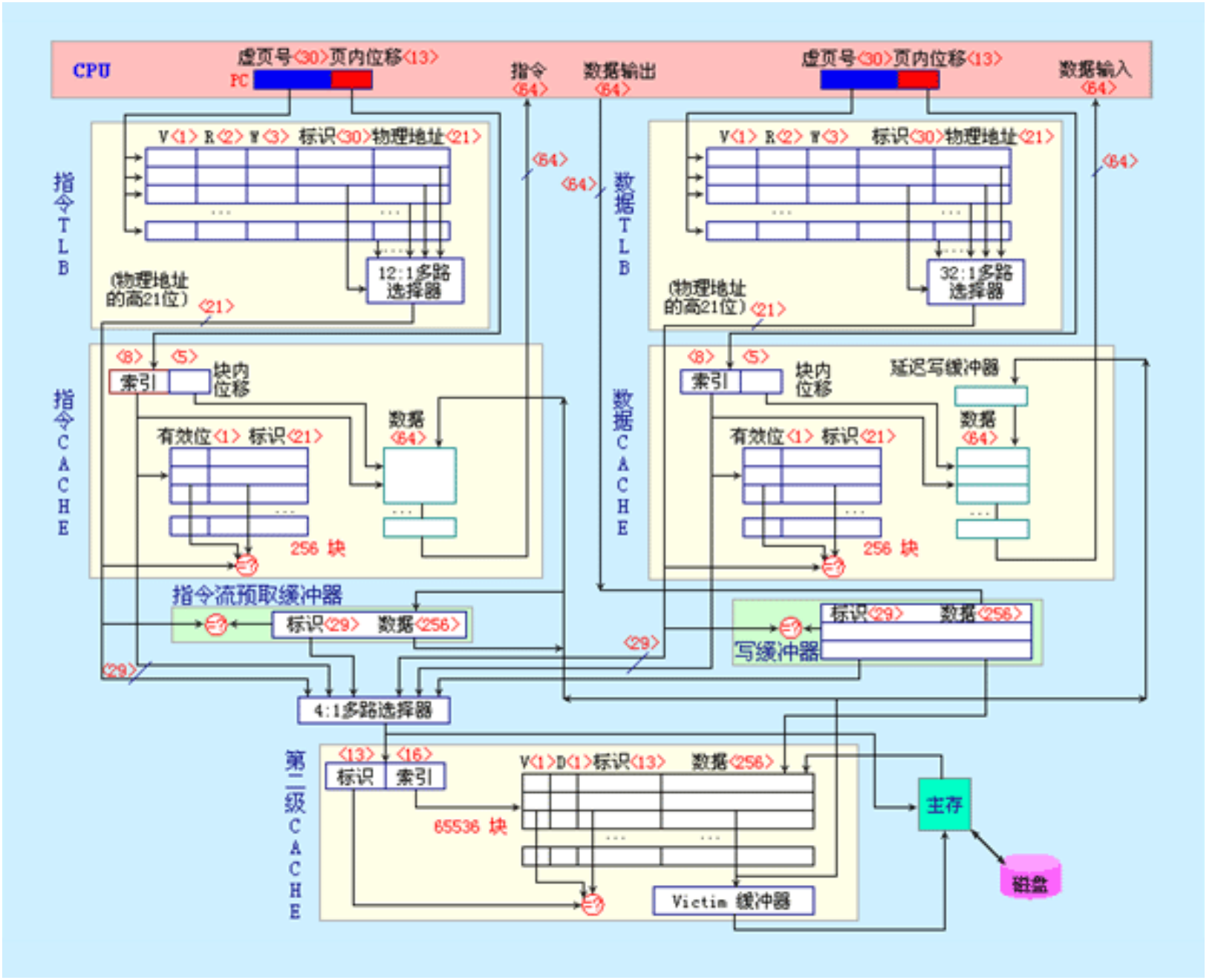
64位中后32位包含5个保护字段：有效字段、用户读许可字段、内核读许可字段、用户写许可字段、内核写许可字段。两个 TLB，一个指令访问TLB，一个数据访问TLB。

3. Alpha AXP 21064 TLB的存储层次参数

参　数	描　　述
块 大 小	1 PTE (8 B)
命 中 时 间	1 个时钟周期
平均失效开销	20个时钟周期
TLB 容 量	指令TLB：8 个 PTE 用于大小为 8 KB的页，4个PTE 用于大小为 4 MB 的页(共 96 B) 数据TLB：32 个 PTE 用于大小为 8 KB、64 KB、512 KB 和 4 MB 的页(共 256 个字节)
块替换策略	随 机
写 策 略	不适用
块映象策略	全相联

5.9 Alpha AXP 21064存储层次

1. 简介
2. 工作过程


CPU
虚页号<30>页内位移<13>
PC
指令
<64>
数据输出
<64>
虚页号<30>页内位移<13>
数据输入
<64>
指令TLB
V<1> R<2> W<3> 标识<30>物理地址<21>
12:1多路选择器
(物理地址的高21位)
<21>
<64>
数据TLB
V<1> R<2> W<3> 标识<30>物理地址<21>
32:1多路选择器
(物理地址的高21位)
<21>
指令CACHE
<8> <5>
索引
块内位移
有效位<1> 标识<21>
数据<64>
256 块
数据CACHE
<8> <5>
索引
块内位移
延迟写缓冲器
有效位<1> 标识<21>
数据<64>
256 块
指令流预取缓冲器
标识<29> 数据<256>
写缓冲器
标识<29> 数据<256>
<29>
4:1多路选择器
第二级CACHE
<13> <16>
标识 索引
65536 块
V<1>D<1>标识<13> 数据<256>
Victim 缓冲器
主存
磁盘