東北林業大學
NORTHEAST FORESTRY UNIVERSITY

# 计算机系统结构

## 第4章：存储系统

# 4.1 存储系统的基本概念

# 4.1 存储系统的基本概念

## 4.1.1 存储系统的层次结构

1. 计算机系统结构设计中关键的问题之一 ：

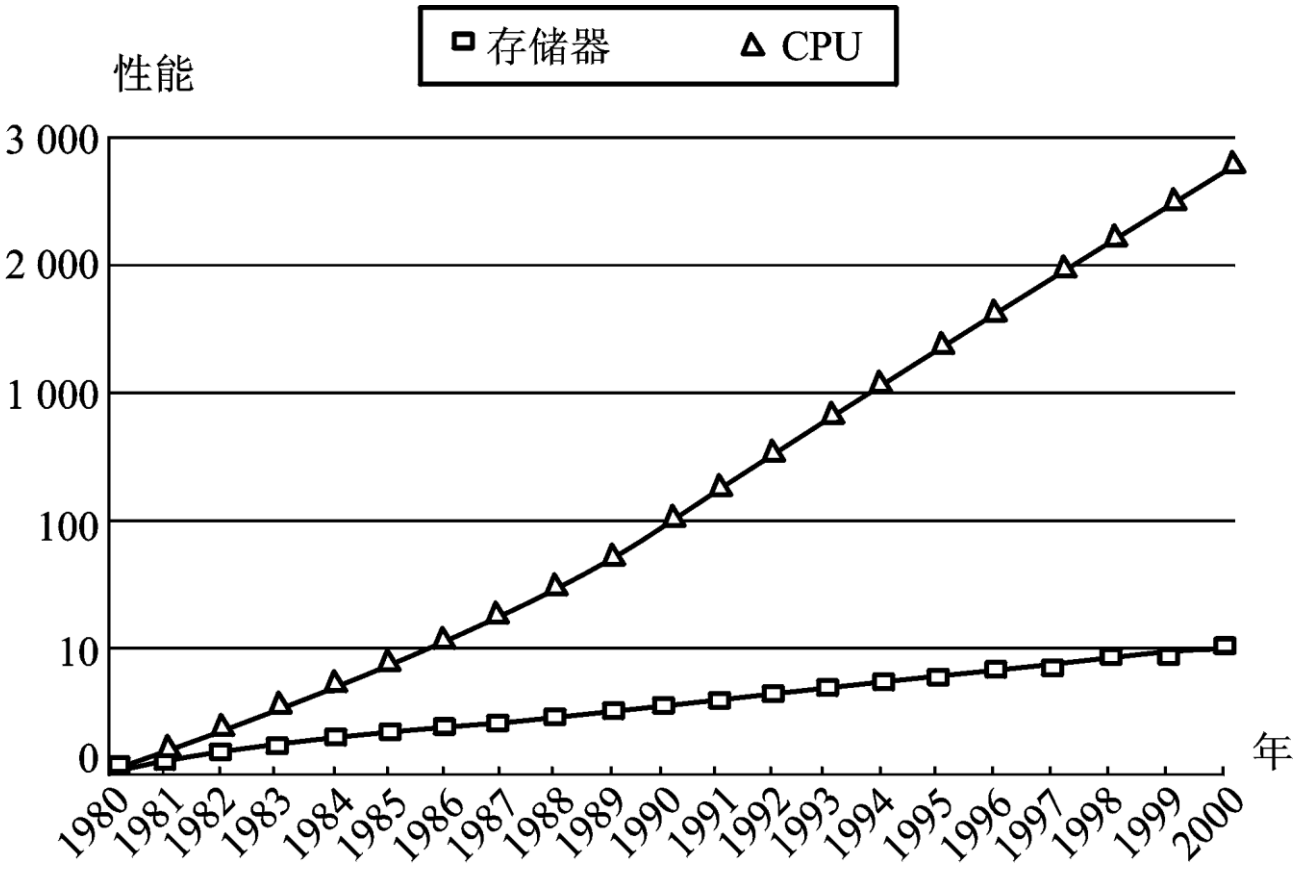
如何以合理的价格，设计容量和速度都满足计算机系统要求的存储器系统？

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。




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）

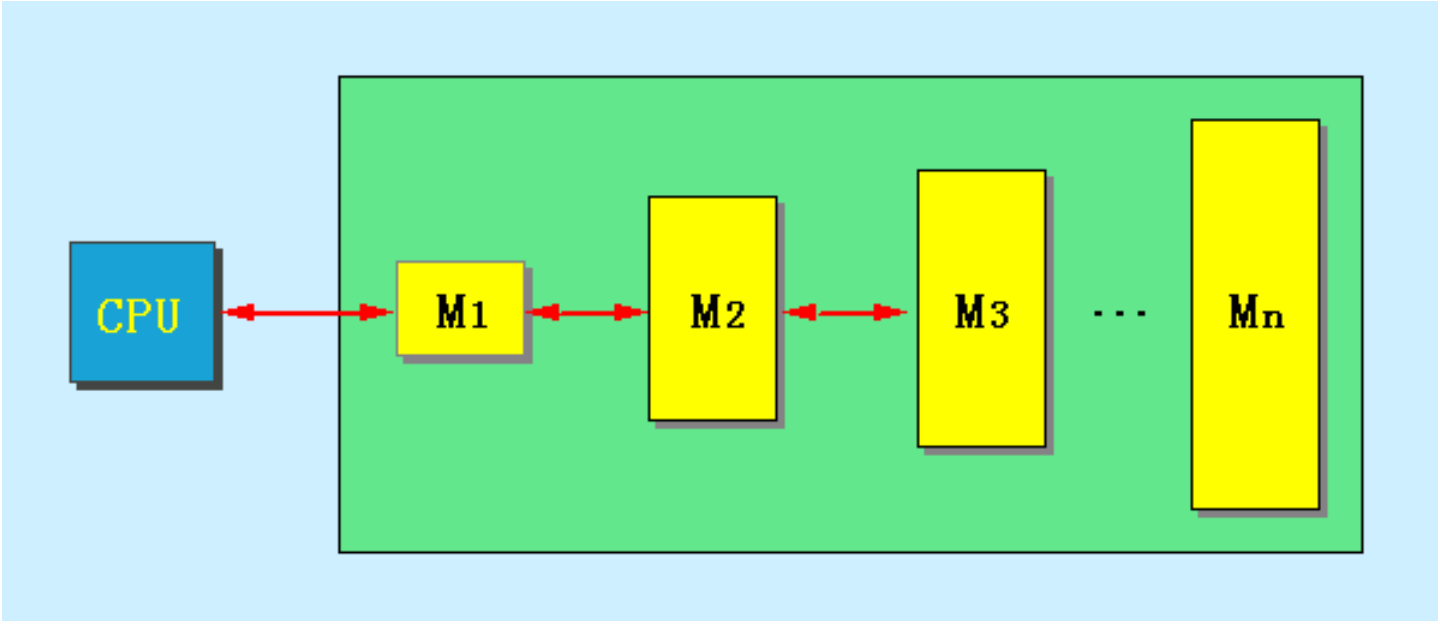
4. 解决方法：采用多种存储器技术，构成多级存储层次结构。

- 程序访问的局部性原理：对于绝大多数程序来说，程序所访问的指令和数据在地址上不是均匀分布的，而是相对簇聚的。 程序局部性原理
- 程序访问的局部性包含两个方面

• 时间局部性：程序马上将要用到的信息很可能就是现在正在使用的信息。

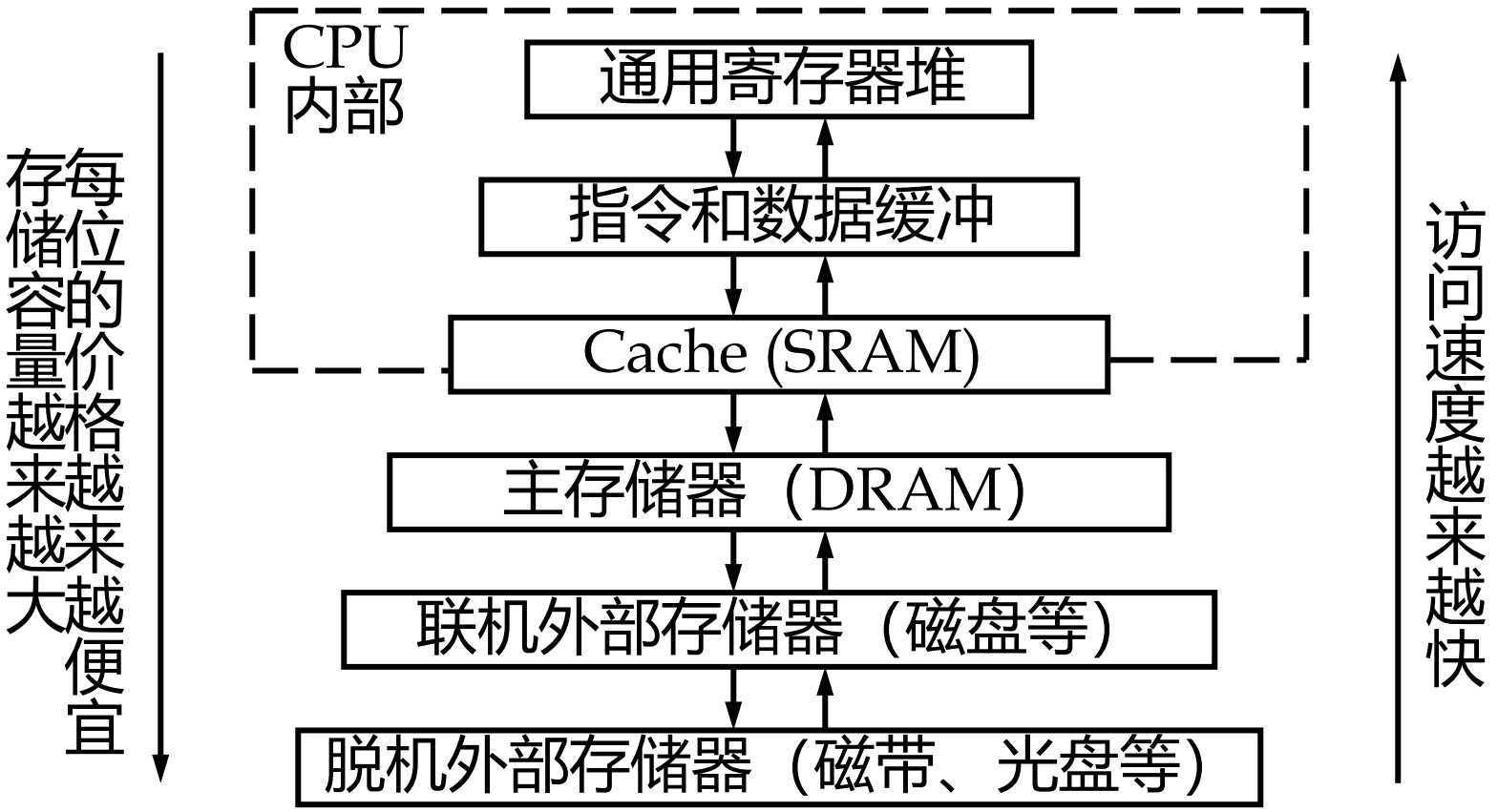
• 空间局部性：程序马上将要用到的信息很可能与现在正在使用的信息在存储空间上是相邻的。

## 5. 存储系统的多级层次结构




多级存储层次

存储器的层次结构

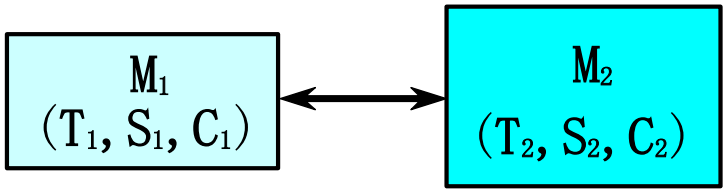

CPU
内部
通用寄存器堆
指令和数据缓冲
Cache (SRAM)
主存储器（DRAM）
联机外部存储器（磁盘等）
脱机外部存储器（磁带、光盘等）
每位容量越来越大
价格越来越便宜
访问速度越来越快

- 假设第i个存储器$M_i$的访问时间为$T_i$，容量为$S_i$，平均每位价格为$C_i$，则
  - 访问时间：$T_1 < T_2 < \ldots < T_n$
  - 容量：$S_1 < S_2 < \ldots < S_n$
  - 平均每位价格：$C_1 > C_2 > \ldots > C_n$
- 整个存储系统要达到的目标：从CPU来看，该存储系统的速度接近于$M_1$的，而容量和每位价格都接近于$M_n$的。
  - 存储器越靠近CPU，则应该让CPU对它的访问频度越高，而且最好大多数的访问都能在$M_1$完成。

## 4.1.2 存储层次的性能参数

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：$T_1$，$S_1$，$C_1$
- $M_2$的参数：$T_2$，$S_2$，$C_2$



1. 存储容量S

- 一般来说，整个存储系统的容量即是第二级存储器$M_2$的容量，即$S=S_2$。

每位价格C

$$C=\frac{C_1S_1+C_2S_2}{S_1+S_2}$$

当$S_1<<S_2$时，$C\approx C_2$。

## 2 命中率H 和不命中率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

- 不命中率 ：F＝1－H

3 平均访存时间$T_A$

$$T_A = HT_1 + (1-H)(T_1 + T_M)$$
$$= T_1 + (1-H)T_M$$

即 $T_A$ = 命中时间 + 不命中率×不命中开销

分两种情况来考虑CPU的一次访存：

- 当命中时，访问时间即为$T_1$（命中时间）
- 当不命中时，情况比较复杂。

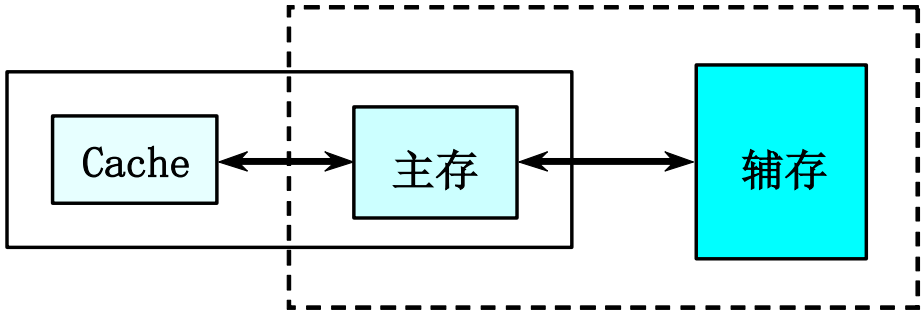
不命中时的访问时间为：$T_2 + T_B + T_1 = T_1 + T_M$

$$T_M = T_2 + T_B$$

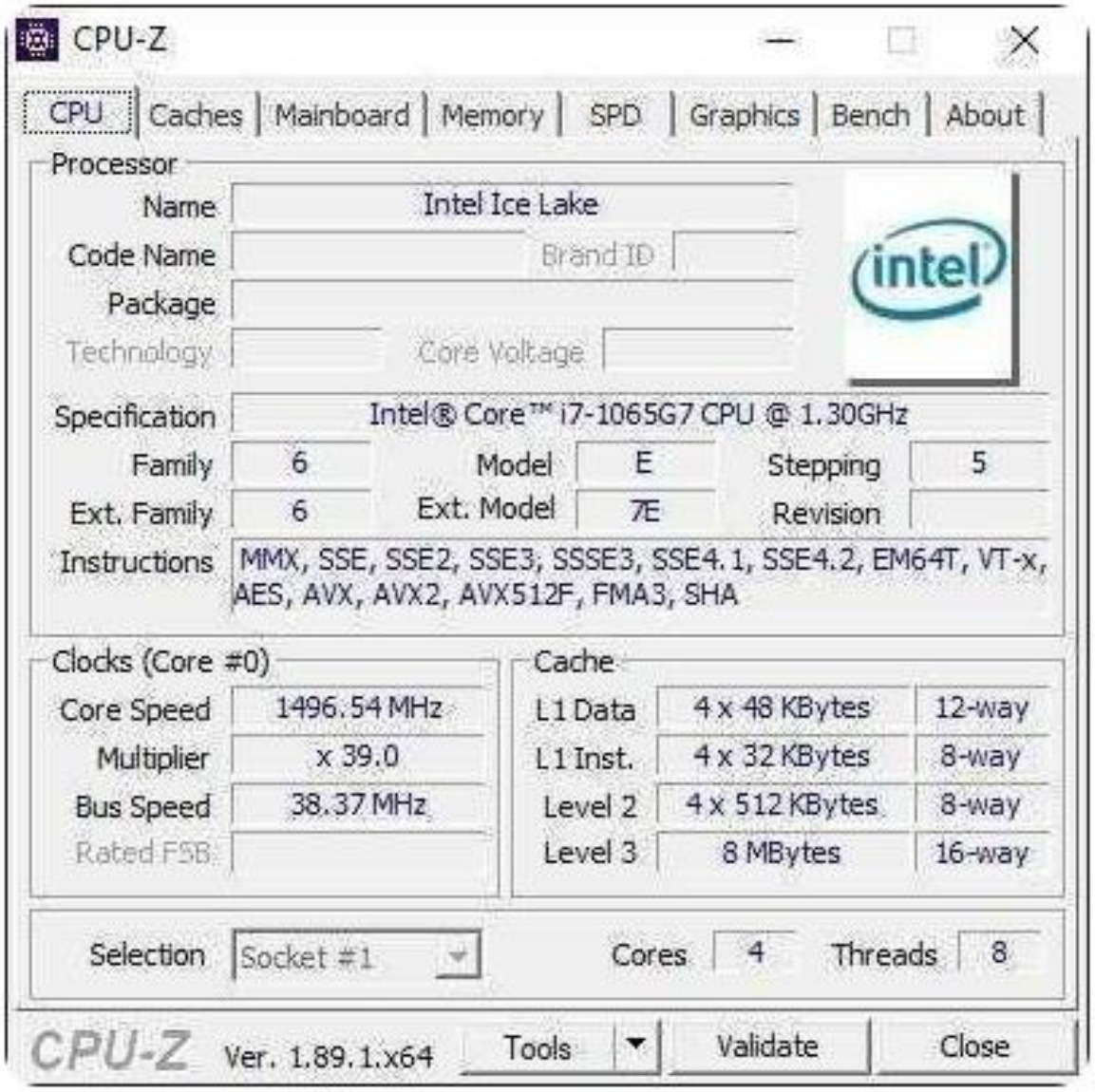
- 不命中开销$T_M$：从向$M_2$发出访问请求（T2）到把整个数据块调入$M_1$中所需的时间。
- 传送一个信息块所需的时间为$T_B$。


## 4.1.3 三级存储系统

- 三级存储系统
  - Cache（高速缓冲存储器）
  - 主存储器
  - 磁盘存储器（辅存）
- 可以看成是由“Cache—主存”层次和“主存—辅存”层次构成的系统。



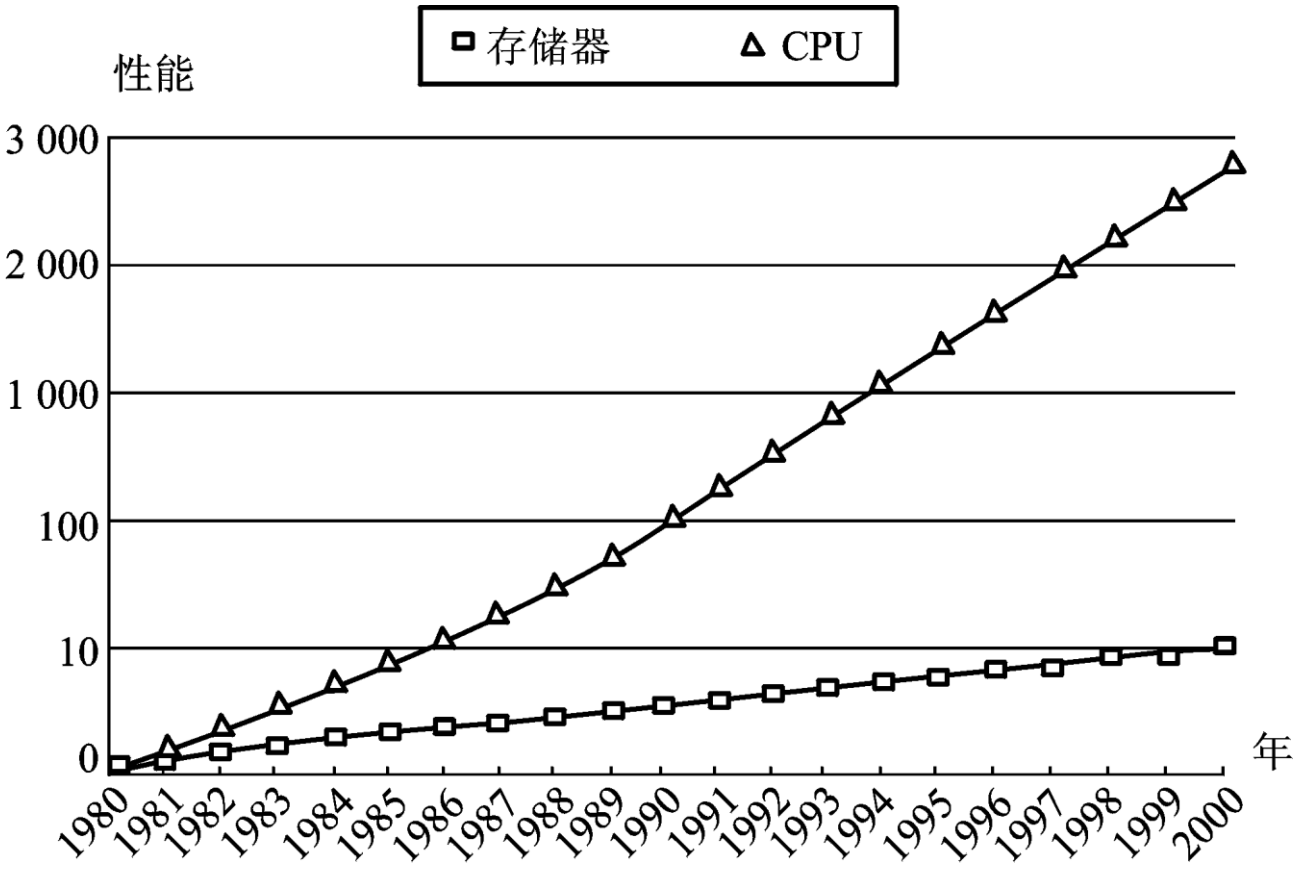
# 4.1 存储系统的基本概念



从主存的角度看

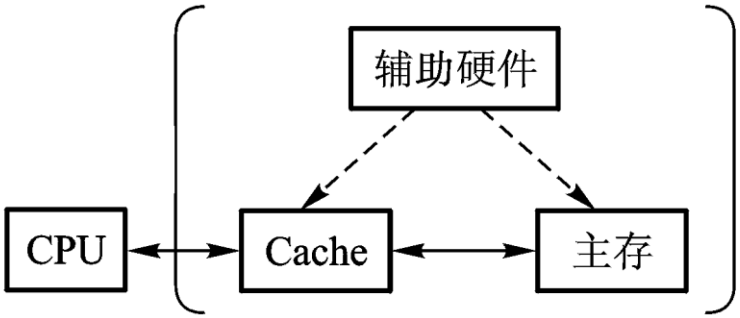
“Cache-主存”层次 ：弥补主存速度的不足，
完全由硬件实现，
对程序员是透明的

- “主存-辅存”层次： 弥补主存容量的不足，
依靠辅助软硬件，特别是操作系统，
常用来实现虚拟存储器

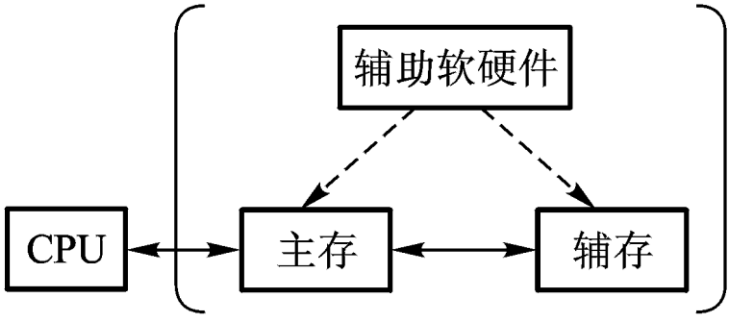



1980年以来存储器和CPU性能随时间而提高的情况

（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

“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 | 几万比一 |
| 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 |
| 不命中时CPU是否切换 | 不切换 | 切换到其他进程 |

## 4.1.4 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?

（映象规则）

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. 当发生不命中时，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）

# 4.2 Cache基本知识

# 4.2 Cache基本知识

## 4.2.1 基本结构和原理

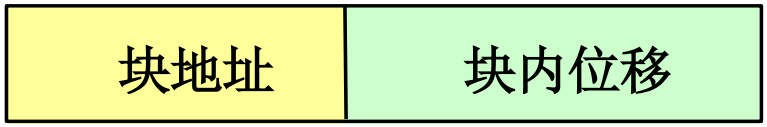
1. 存储空间分割与地址计算

2. Cache和主存分块

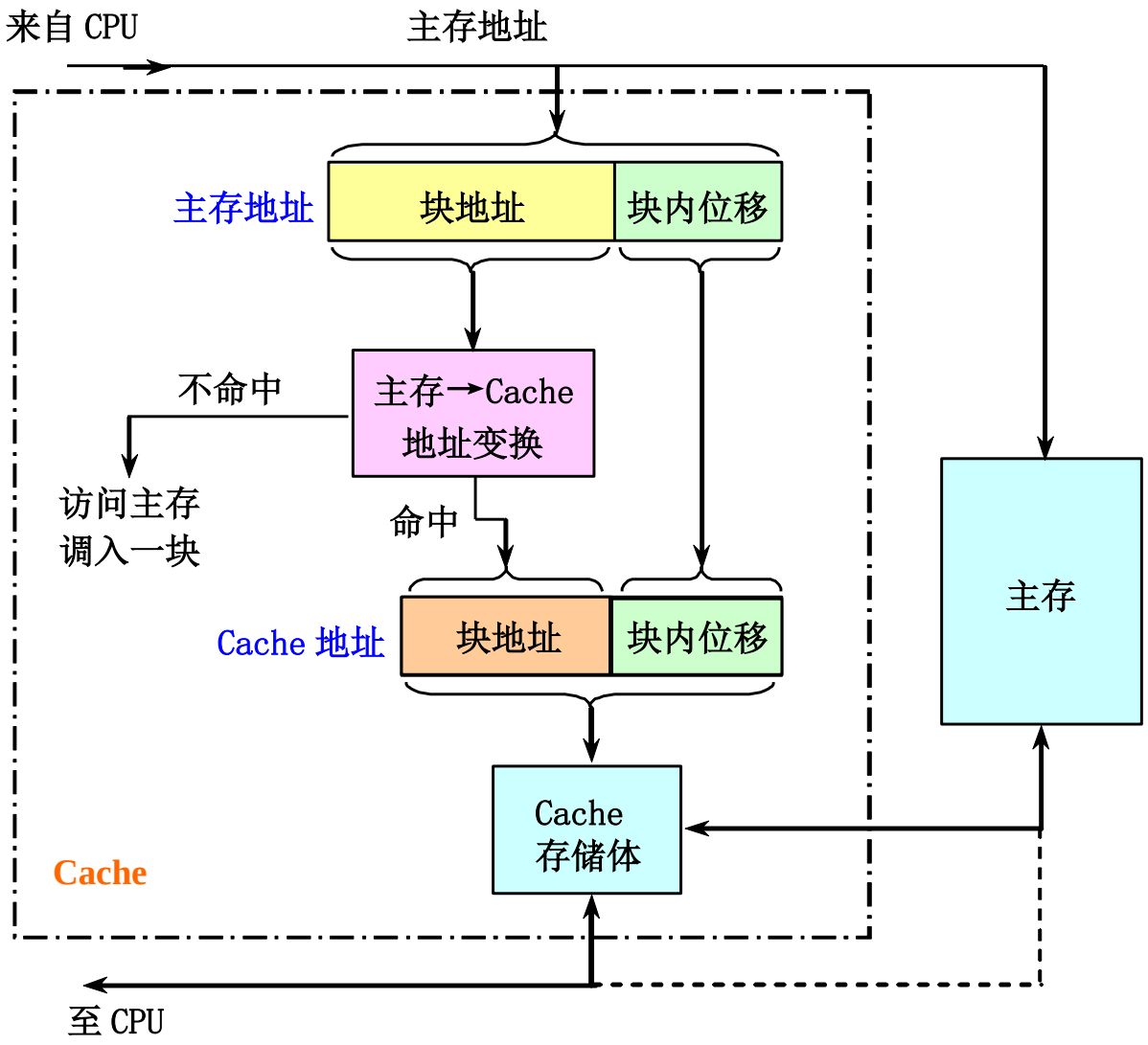
Cache是按块进行管理的。Cache和主存均被分割成大小相同的块。信息以块为单位调入Cache。

- 主存块地址（块号）用于查找该块在Cache中的位置。
- 块内位移用于确定所访问的数据在该块中的位置。

主存地址：



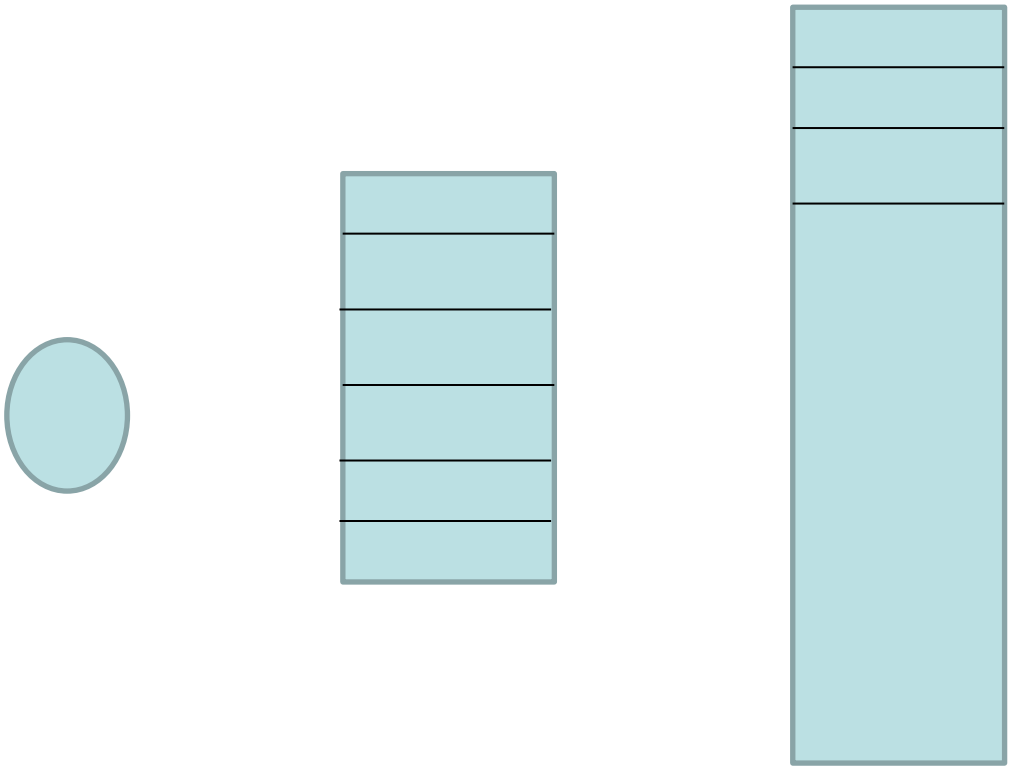
# 3. Cache的基本工作原理示意图



## 4.2.2 映象规则

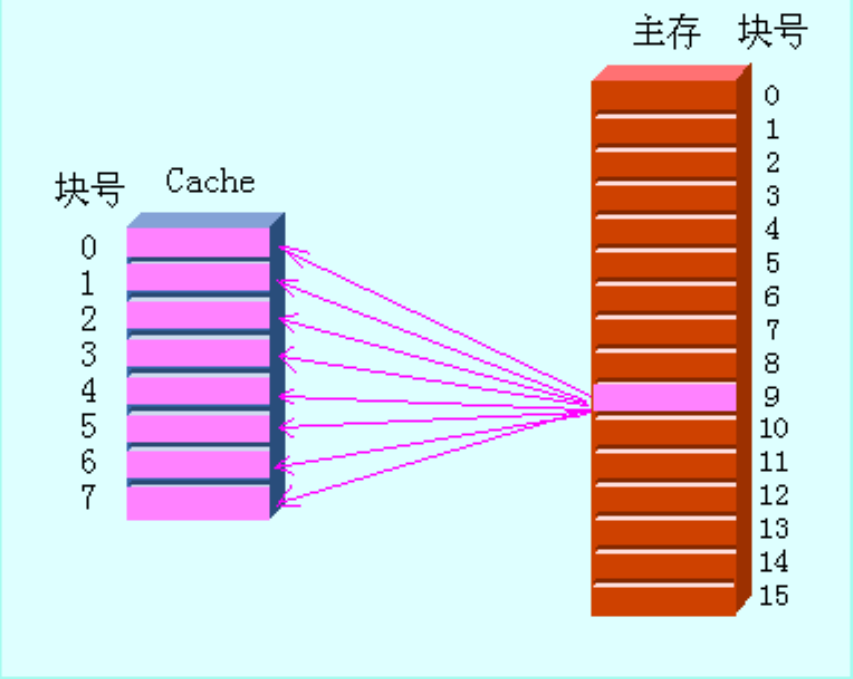
### 1. 全相联映象

- 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。



## 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）

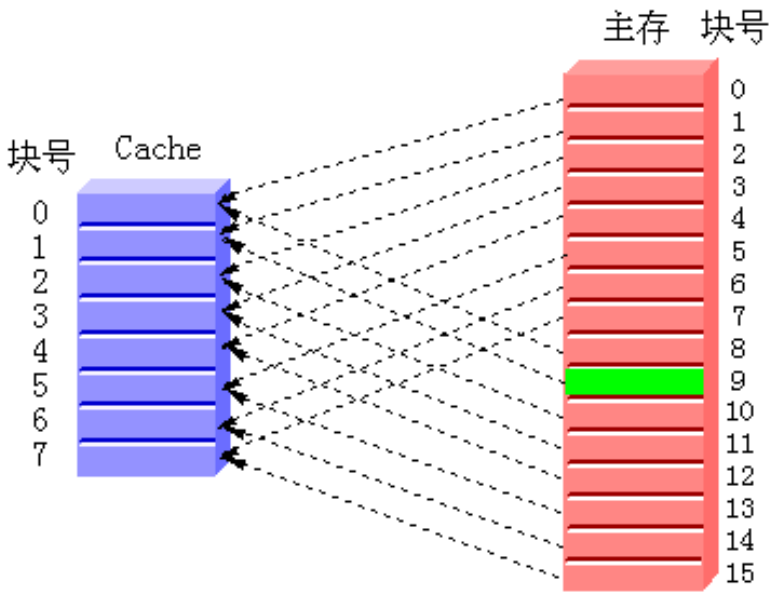
- 对比：阅览室位置 —— 只有一个位置可以坐
- 特点：空间利用率最低，冲突概率最高，

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

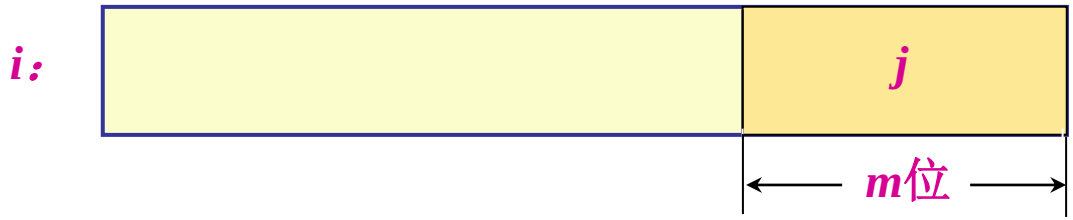
  $j = i \bmod (M)$ （$M$为Cache的块数）

## 直 接 映 射

### (举 例)

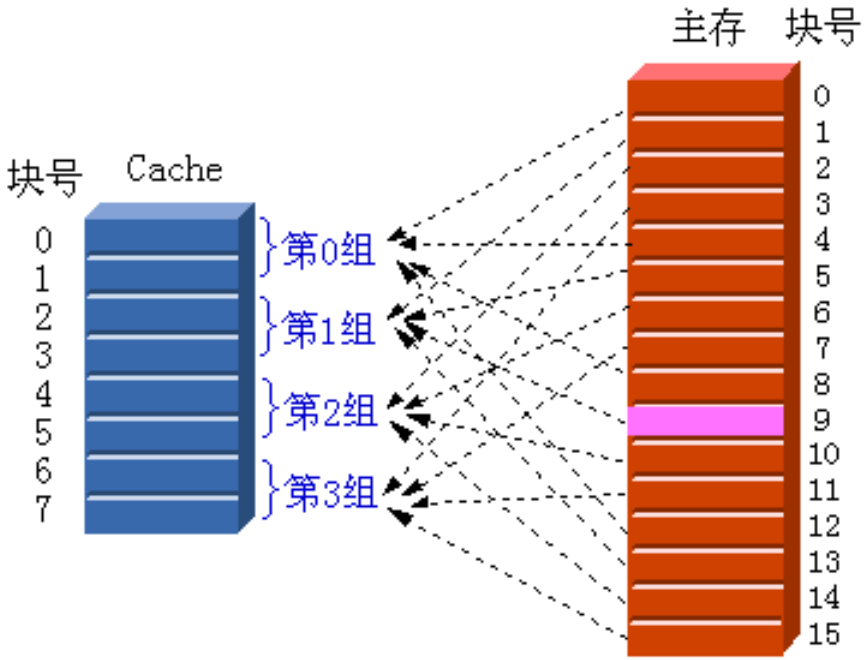


- 设$M=2^m$，则当表示为二进制数时，j实际上就是i的低m位：



## 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

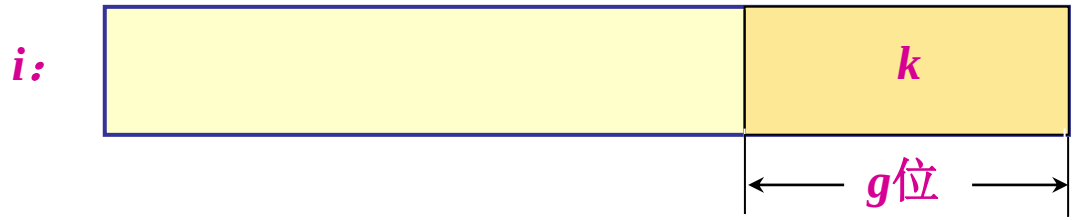


➢ 组的选择常采用位选择算法

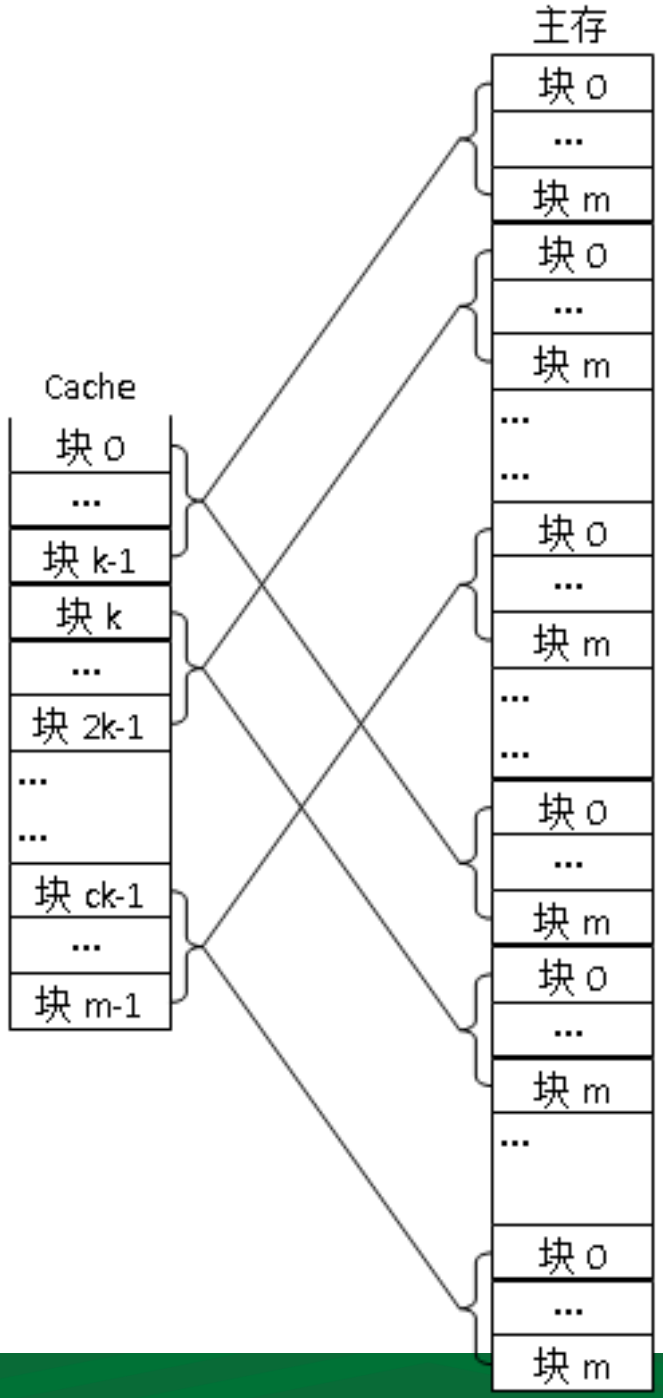
- 若主存第$i$ 块映象到第$k$ 组，则：

$$k = i \bmod (G)$$ （$G$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。


主存
块 0
...
块 m
块 0
...
块 m
...
...
块 0
...
块 m
...
...
块 0
...
块 m
块 0
...
块 m
...
块 0
...
块 m
Cache
块 0
...
块 k-1
块 k
...
块 2k-1
...
...
块 ck-1
...
块 m-1

- *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，不命中率也就越低。

| | *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：$n \leqslant 4$

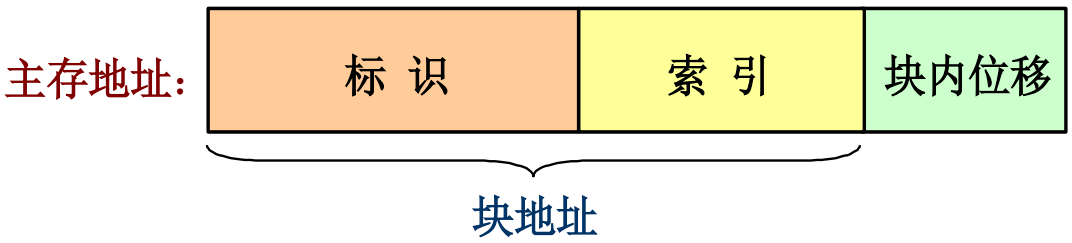
想一想：相联度一定是越大越好？

## 4.2.3 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?
- 若有的话，如何确定其位置?

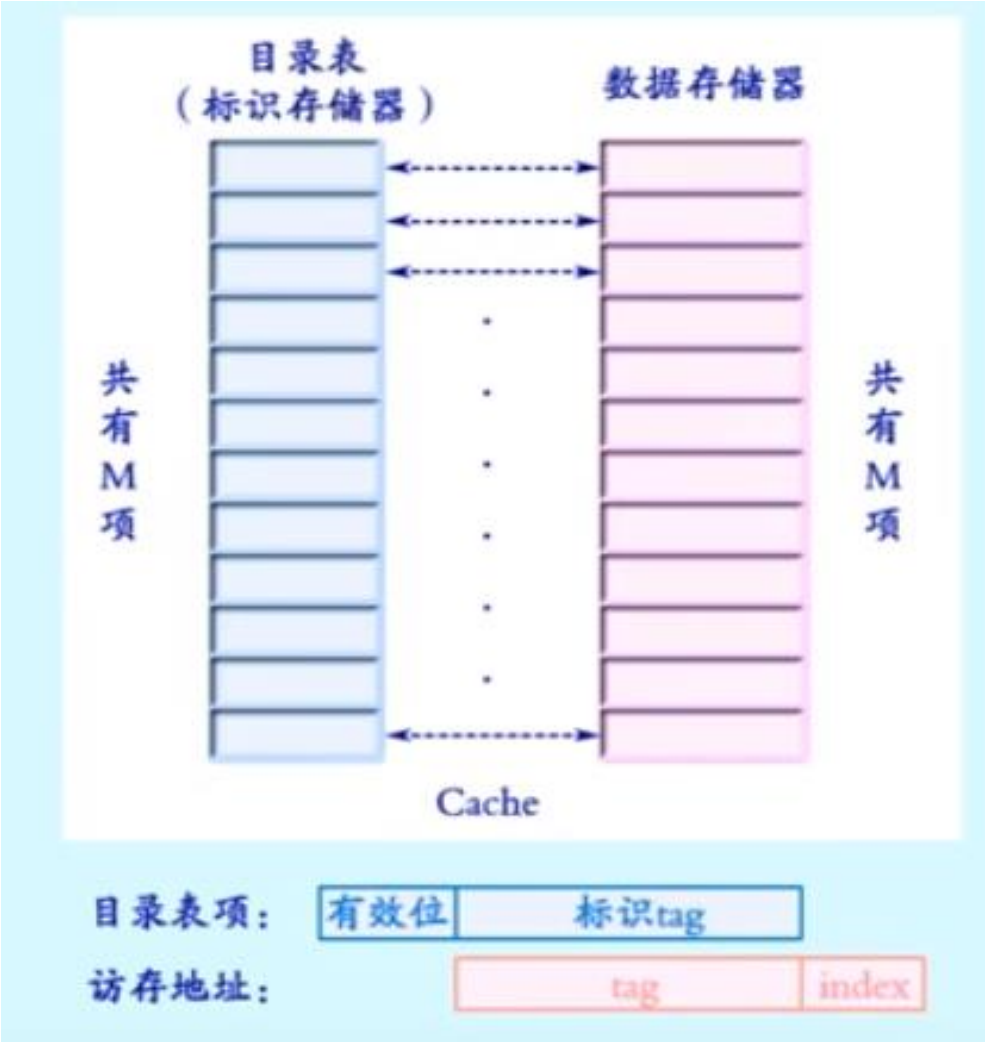
通过查找目录表来实现

- 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




- 只需查找候选位置所对应的目录表项

# 4.2 Cache基本知识



## 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）

## 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

## 4.2.4 替换算法

### 1. 替换算法

- 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。
- 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：命中率较高

➢ LRU和随机法分别因其不命中率低和实现简单而被广泛采用。

- 模拟数据表明，对于容量很大的Cache，LRU和随机法的命中率差别不大。

## 表4.3　LRU和随机替换法的失效率比较

| 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 |
| 16K | 5.18% | 5.56% | 4.67% | 5.29% | 4.39% | 4.96% |
| 64K | 1.88% | 2.01% | 1.54% | 1.66% | 1.39% | 1.53% |
| 256K | 1.15% | 1.15% | 1.13% | 1.13% | 1.12% | 1.12% |

## 4.2.5 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

“写”访问有可能导致Cache和主存内容的不一致

3. 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（也称为存直达法）
  - 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”）

4. 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。

5. 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把数据写入缓冲器后就可以继续执行，由缓冲器负责把数据写入主存。

6. “写”操作时的调块(写不命中)

- 按写分配(写时取)

  写不命中时，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)

  写不命中时，直接写入下一级存储器而不调块。

7. 写策略与调块

- 写回法 —— 按写分配
- 写直达法 —— 不按写分配

## 4.2.6 Cache的性能分析

1. 不命中率
   - 与硬件速度无关
   - 容易产生一些误导，不能直观反映性能
2. 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

例4.1　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一个时钟周期（因为混合Cache只有一个端口，无法同时满足两个请求。按照前面有关流水线的术语，混合Cache会导致结构冲突），根据表4.4所列的失效率，

（1）试问指令Cache和数据Cache容量均为16KB的分离Cache和容量为32KB的混合Cache相比，哪种Cache的失效率更低？

（2）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？

# 表4.4指令Cache、数据Cache以及混合Cache失效率的比较

| 容量 | 指令 Cache | 数据 Cache | 混合 Cache |
|---|---|---|---|
| 1KB | 3.06% | 24.61% | 13.34% |
| 2KB | 2.26% | 20.57% | 9.78% |
| 4KB | 1.78% | 15.94% | 7.24% |
| 8KB | 1.10% | 10.19% | 4.57% |
| 16KB | 0.64% | 6.47% | 2.87% |
| 32KB | 0.39% | 4.82% | 1.99% |
| 64KB | 0.15% | 3.77% | 1.35% |
| 128KB | 0.02% | 2.88% | 0.95% |

解 （1） 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

$$(75\% \times 0.64\%)+(25\% \times 6.47\%)=2.10\%$$

根据表4.4，容量为32KB的混合Cache的失效率略低一些，只有1.99%。

（2）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，消除了结构相关。

## 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

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

$$= IC \times （CPI\text{execution} + \text{每条指令的平均访存次数} \times \text{不命中率} \times \text{不命中开销}） \times \text{时钟周期时间}$$

例4.2 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

$$\text{CPU时间}_{\text{有cache}} = IC \times (CPI\text{execution} + \text{每条指令的平均访存次数} \times \text{不命中率} \times \text{不命中开销}) \times \text{时钟周期时间}$$

$$= IC \times (2.0 + 1.33 \times 2\% \times 50) \times \text{时钟周期时间}$$

$$= IC \times 3.33 \times \text{时钟周期时间}$$

考虑Cache的不命中后，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4.3 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*×（*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*×（*CPI*execution× 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：

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

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

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

直接映象Cache的平均性能好一些。

## 4.2.6 改进Cache的性能

1. 平均访存时间＝命中时间＋不命中率×不命中开销
2. 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
3. 下面介绍15种Cache优化技术
   - 7种用于降低不命中率
   - 5种用于减少不命中开销
   - 3种用于减少命中时间

# 4.3 降低Cache不命中率

# 4.3 降低Cache不命中率

## 4.3 三种类型的不命中

### 1. 三种类型的不命中(3C)

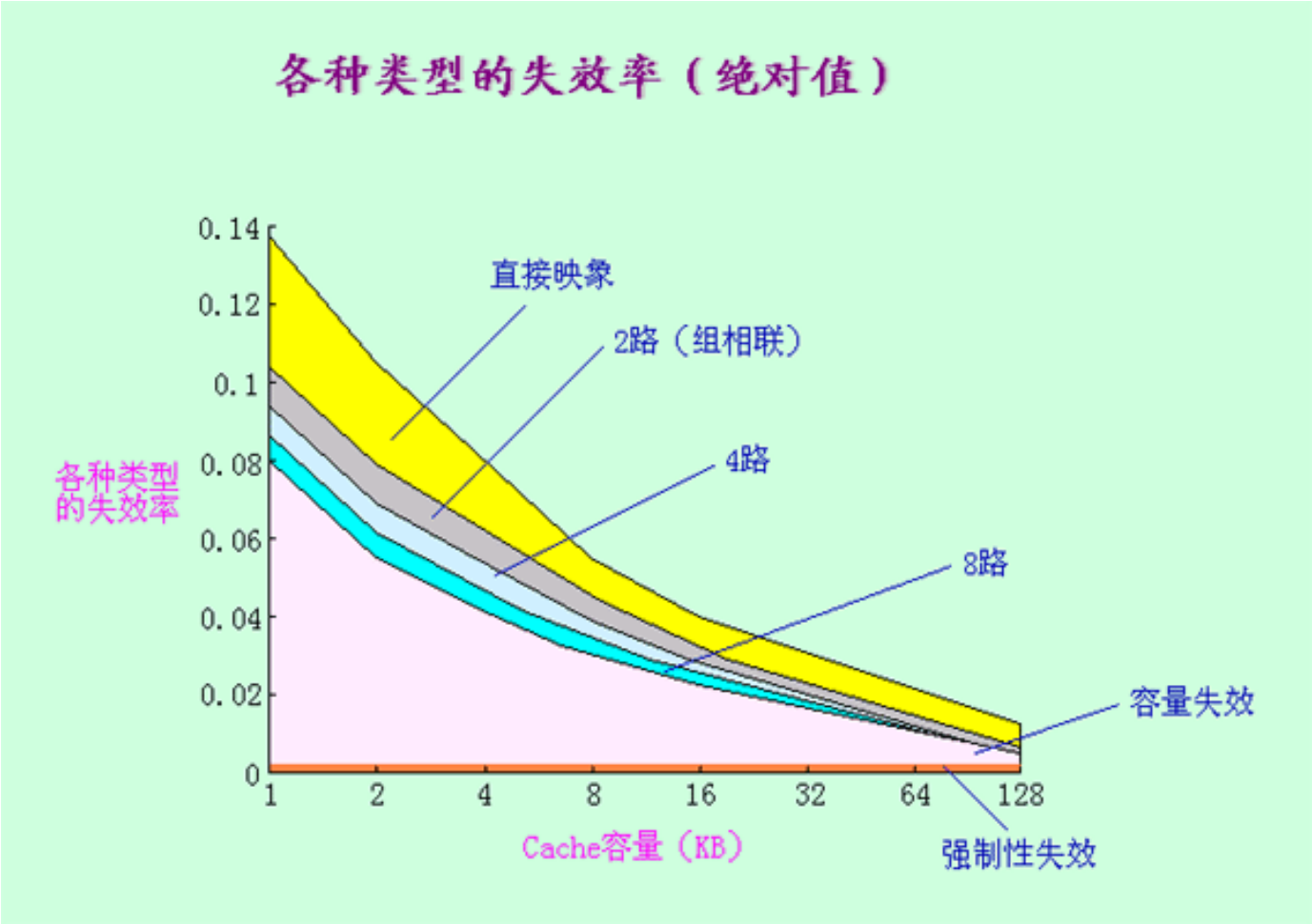
- **强制性不命中(Compulsory miss)**
  - 当第一次访问一个块时，该块不在Cache中，需从下一级存储器中调入Cache，这就是强制性不命中。（冷启动不命中，首次访问不命中）
- **容量不命中(Capacity miss )**
  - 如果程序执行时所需的块不能全部调入Cache中，则当某些块被替换后，若又重新被访问，就会发生不命中。这种不命中称为容量不命中。

➢ 冲突不命中(Conflict miss)

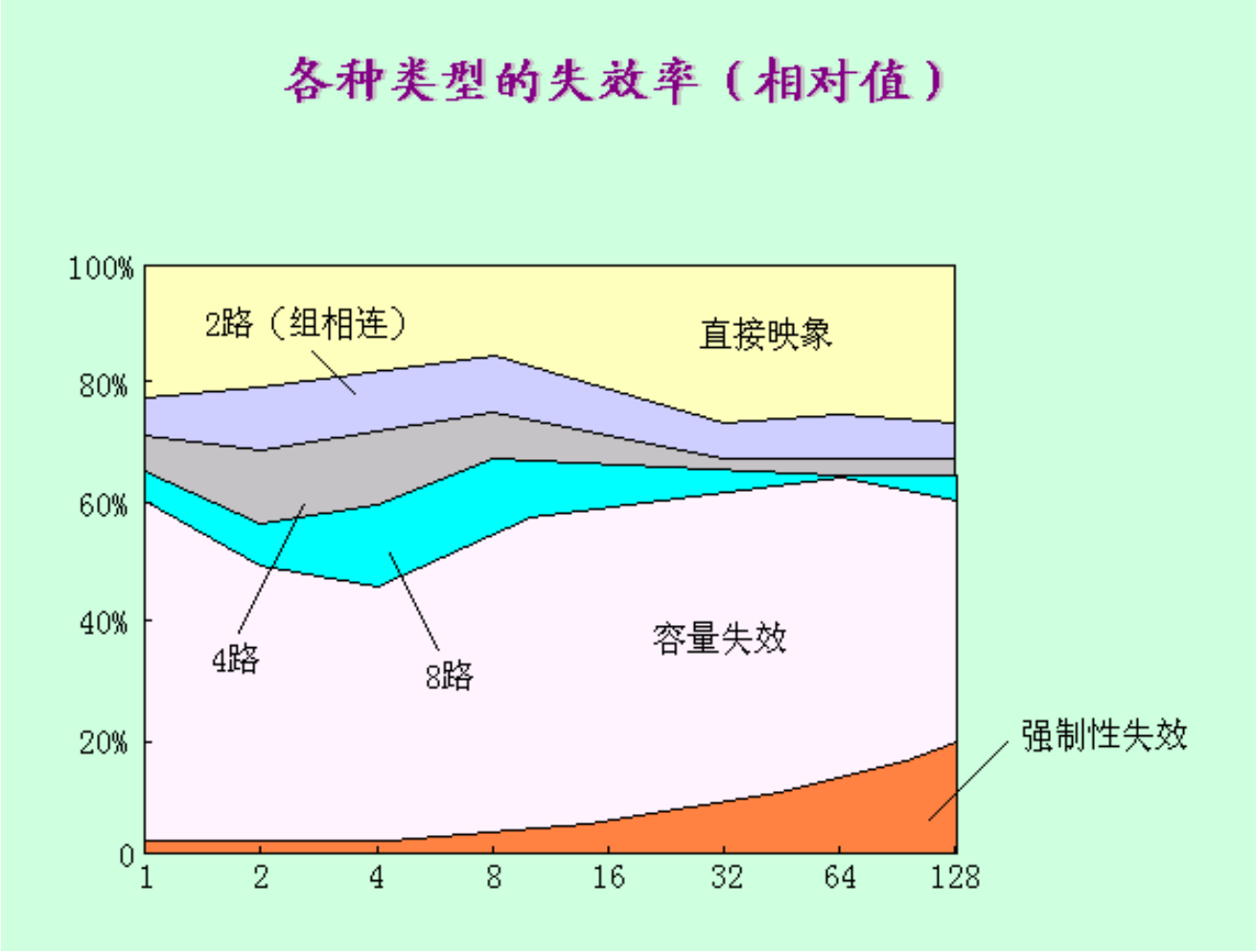
- 在组相联或直接映象Cache中，若太多的块映象到同一组(块)中，则会出现该组中某个块被别的块替换(即使别的组或块有空闲位置)，然后又被重新访问的情况。这就是发生了冲突不命中。

(碰撞不命中，干扰不命中)

# 4.3 降低Cache不命中率



# 4.3 降低Cache不命中率



➢ 可以看出：

- 相联度越高，冲突不命中就越少；
- 强制性不命中和容量不命中不受相联度的影响；
- 强制性不命中不受Cache容量的影响，但容量不命中却随着容量的增加而减少。

# 4.3 降低Cache不命中率



- 减少三种不命中的方法
  - 强制性不命中：增加块大小，预取 （本身发生很少）
  - 容量不命中：只能增加Cache容量
  - 冲突不命中：提高相联度 （理想情况：全相联）
- 许多降低不命中率的方法会增加命中时间或不命中开销

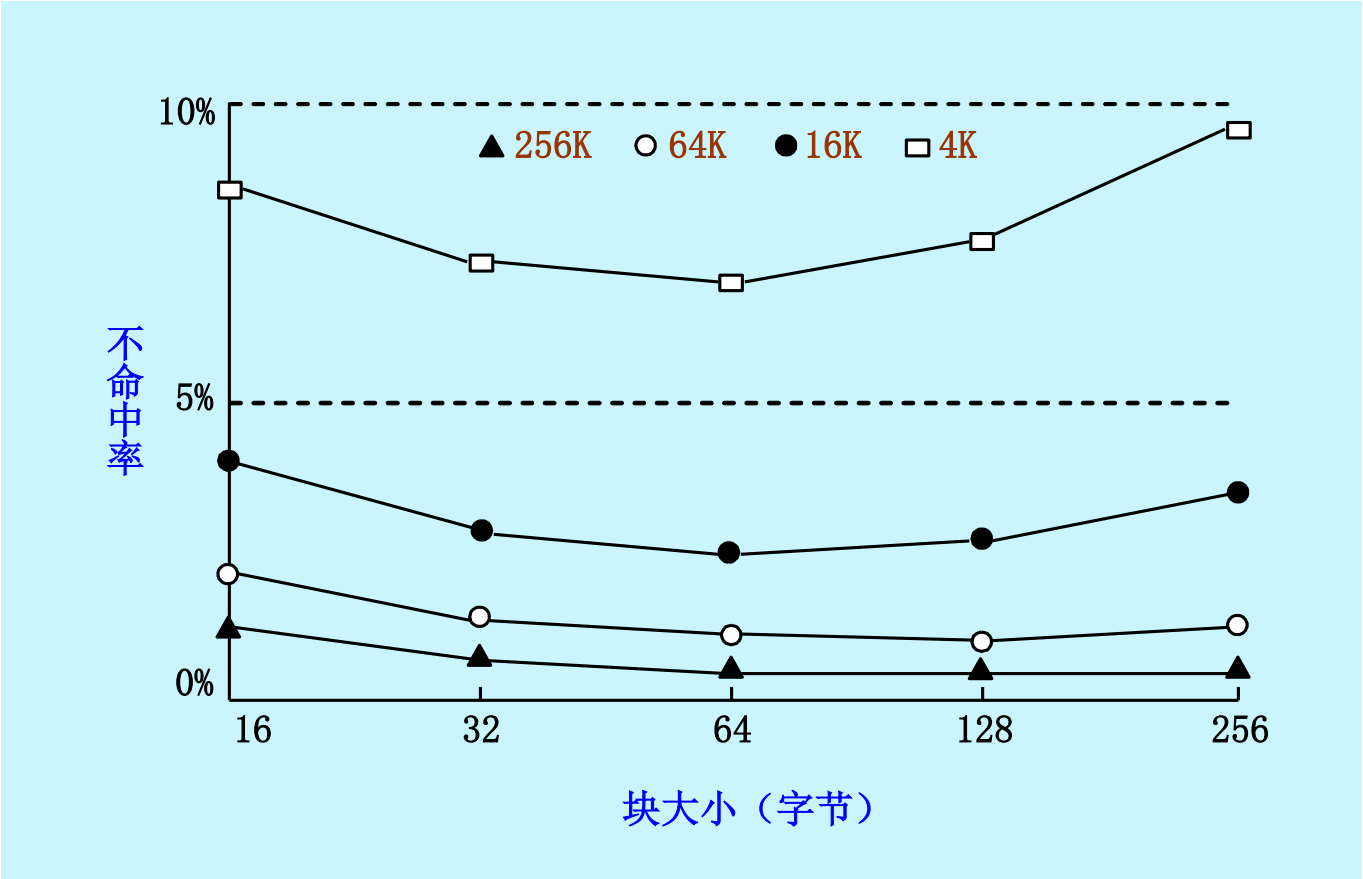
## 4.3.1 增加Cache块大小

1. 不命中率与块大小的关系，表4－6
   - 对于给定的Cache容量，当块大小增加时，不命中率开始是下降，后来反而可能上升了。

     原因：

     - 一方面它减少了强制性不命中；
     - 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。
   - Cache容量越大，使不命中率达到最低的块大小就越大。
2. 增加块大小会增加不命中开销

# 4.3 降低Cache不命中率




不命中率随块大小变化的曲线（表4—6）

# 4.3 降低Cache不命中率



- 各种块大小情况下Cache的不命中率. 表4 - 6

| 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% |

例4.4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4.6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:平均访存时间为:

平均访存时间=命中时间十失效率×失效开销

假设命中时间与块大小无关，为1个时钟，那么对于一个块大小为16字节（42T）、容量为1KB 的Cache来说：

平均访存时间=1+(15.05%×42)=7.321个时钟周期

而对于块大小为256字节（失效开销72T＝40T+32T） 、容量为256KB的Cache来说，平均访存时间为：

平均访存时间=1+(0.49%×72)=1.353个时钟周期

表4.7列出了在这两种极端情况之间的各种块大小和各种Cache容量的平均访存时间。粗体字的数字为速度最快的情况：Cache容量为1KB、4KB、16KB的情况下块大小为32B时速度最快；容量为64KB和256KB时，64字节最快。实际上，这些块大小都是当今处理机Cache中最常见的。

| 块大小/B | 失效开销/时钟周期 | Cache 容量/B | | | | |
|---|---|---|---|---|---|---|
| | | 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 | 1.174 |
| 256 | 72 | 10.847 | 7.847 | 3.369 | 1.828 | 1.353 |

从失效开销的角度来讲，块大小的选择取决于下一级存储器的延迟和带宽两个方面。高延迟和高带宽时，宜采用较大的Cache块，与之相反，宜采用较小的Cache块，块的数量多，就有可能减少冲突失效。

## 4.3.2 增加Cache的容量（进而增加块大小）

1. 降低不命中率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. 增加Cache容量会降低不命中率，但当Cache容量已经很大时，再增大Cache的效果也会不明显
3. 这种方法在片外Cache中用得比较多

## 4.3.3 提高相联度

1. 采用相联度超过8的方案的实际意义不大。已经等效于全相联

2. 2:1 Cache经验规则

   容量为N的直接映象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。

3. 提高相联度是以增加命中时间为代价。

例4.5 假定提高相联度全按下列比例增大处理器时钟周期：

$$时钟周期_{2路}=1.10\times 时钟周期_{1路}$$

$$时钟周期_{4路}=1.12\times 时钟周期_{1路}$$

$$时钟周期_{8路}=1.14\times 时钟周期_{1路}$$

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。使用表4.5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

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

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

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

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

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直接映象Cache的平均访存时间为

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

容量为128KB的8路组相联Cache的平均访存时间为

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

利用这些公式和表4.5中给出的失效率，可得各种容量和相联度情况下Cache的平均访存时间，如表4.8所示。

| 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 | 3.75 | 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** |

表中的数据说明，当Cache容量不超过16KB时，上述三个不等式成立。从32KB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。

表4.5

Cache 的总不命中率以及 3C 所占的比例

| Cache 容量 | 相联度 | 总不命中率 | 不命中率组成(相对百分比) | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 强制性不命中 | | 容量不命中 | | 冲突不命中 | |
| 1KB | 1 路 | 0.133 | 0.002 | 1% | 0.080 | 60% | 0.052 | 39% |
| 1KB | 2 路 | 0.105 | 0.002 | 2% | 0.080 | 76% | 0.023 | 22% |
| 1KB | 4 路 | 0.095 | 0.002 | 2% | 0.080 | 84% | 0.013 | 14% |
| 1KB | 8 路 | 0.087 | 0.002 | 2% | 0.080 | 92% | 0.005 | 6% |

| Cache 容量 | 相联度 | 总不命中率 | 强制性不命中 | | 容量不命中 | | 冲突不命中 | |
|---|---|---|---|---|---|---|---|---|
| 2KB | 1 路 | 0.098 | 0.002 | 2% | 0.044 | 45% | 0.052 | 53% |
| 2KB | 2 路 | 0.076 | 0.002 | 2% | 0.044 | 58% | 0.030 | 39% |
| 2KB | 4 路 | 0.064 | 0.002 | 3% | 0.044 | 69% | 0.018 | 28% |
| 2KB | 8 路 | 0.054 | 0.002 | 4% | 0.044 | 82% | 0.008 | 14% |
| 4KB | 1 路 | 0.072 | 0.002 | 3% | 0.031 | 43% | 0.039 | 54% |
| 4KB | 2 路 | 0.057 | 0.002 | 3% | 0.031 | 55% | 0.024 | 42% |
| 4KB | 4 路 | 0.049 | 0.002 | 4% | 0.031 | 64% | 0.016 | 32% |
| 4KB | 8 路 | 0.039 | 0.002 | 5% | 0.031 | 80% | 0.006 | 15% |
| 8KB | 1 路 | 0.046 | 0.002 | 4% | 0.023 | 51% | 0.021 | 45% |
| 8KB | 2 路 | 0.038 | 0.002 | 5% | 0.023 | 61% | 0.013 | 34% |
| 8KB | 4 路 | 0.035 | 0.002 | 5% | 0.023 | 66% | 0.010 | 28% |
| 8KB | 8 路 | 0.029 | 0.002 | 6% | 0.023 | 79% | 0.004 | 15% |
| 16KB | 1 路 | 0.029 | 0.002 | 7% | 0.015 | 52% | 0.012 | 42% |
| 16KB | 2 路 | 0.022 | 0.002 | 9% | 0.015 | 68% | 0.005 | 23% |
| 16KB | 4 路 | 0.020 | 0.002 | 10% | 0.015 | 74% | 0.003 | 17% |
| 16KB | 8 路 | 0.018 | 0.002 | 10% | 0.015 | 80% | 0.002 | 9% |

表4.5

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 32KB | 1路 | 0.020 | 0.002 | 10% | 0.010 | 52% | 0.008 | 38% |
| 32KB | 2路 | 0.014 | 0.002 | 14% | 0.010 | 74% | 0.002 | 12% |
| 32KB | 4路 | 0.013 | 0.002 | 15% | 0.010 | 79% | 0.001 | 6% |
| 32KB | 8路 | 0.013 | 0.002 | 15% | 0.010 | 81% | 0.001 | 4% |
| 64KB | 1路 | 0.014 | 0.002 | 14% | 0.007 | 50% | 0.005 | 36% |
| 64KB | 2路 | 0.010 | 0.002 | 20% | 0.007 | 70% | 0.001 | 10% |
| 64KB | 4路 | 0.009 | 0.002 | 21% | 0.007 | 75% | 0.000 | 3% |
| 64KB | 8路 | 0.009 | 0.002 | 22% | 0.007 | 78% | 0.000 | 0 |
| 128KB | 1路 | 0.010 | 0.002 | 20% | 0.004 | 40% | 0.004 | 40% |
| 128KB | 2路 | 0.007 | 0.002 | 29% | 0.004 | 58% | 0.001 | 14% |
| 128KB | 4路 | 0.006 | 0.002 | 31% | 0.004 | 61% | 0.001 | 8% |
| 128KB | 8路 | 0.006 | 0.002 | 31% | 0.004 | 62% | 0.000 | 7% |

# 本章小考

根据例4.5，计算32K的cache在各种相联度情况下的平均访存时间？

假定提高相联度全按下列比例增大处理器时钟周期：

$$时钟周期_{2路}=1.10\times 时钟周期_{1路}$$

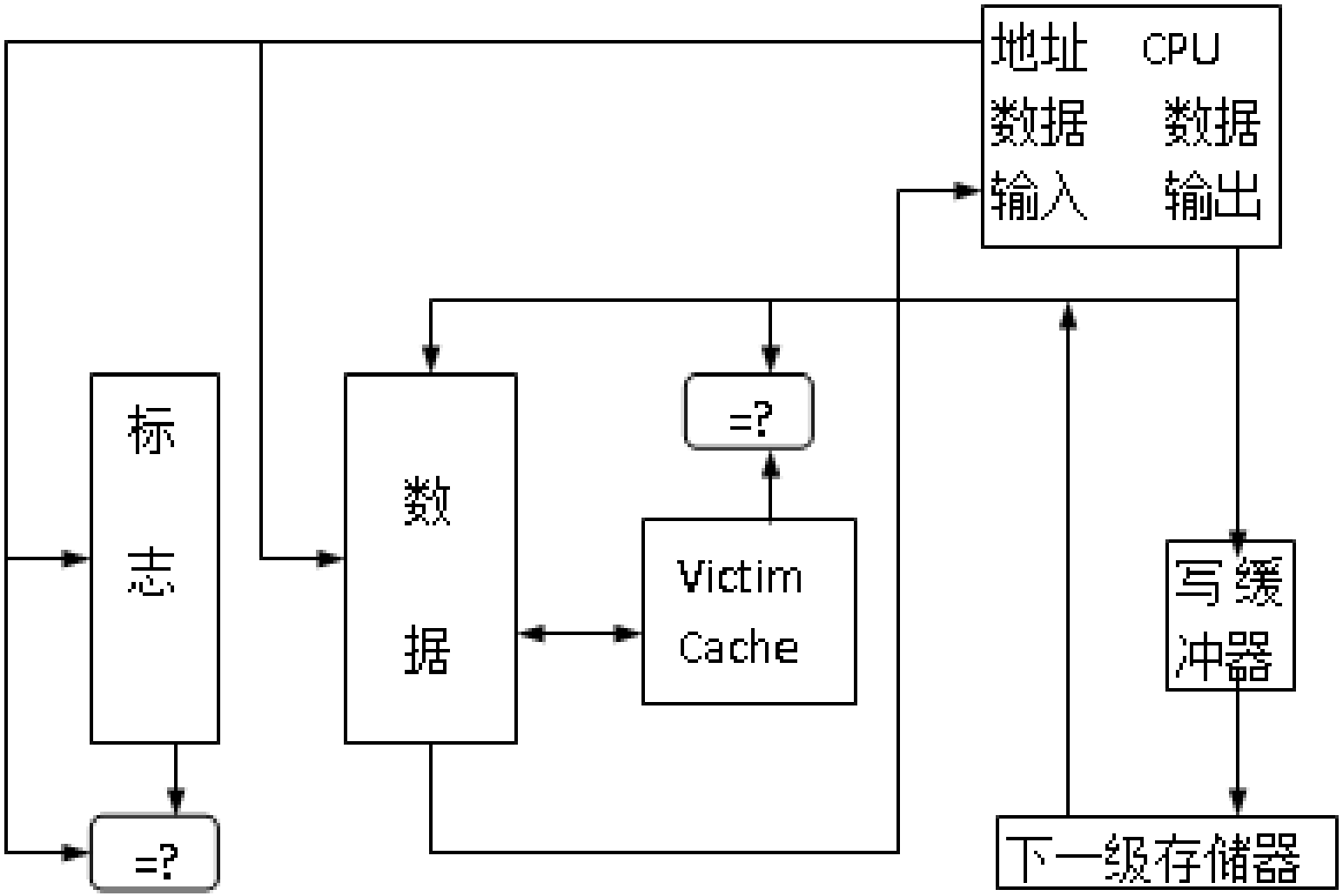
$$时钟周期_{4路}=1.12\times 时钟周期_{1路}$$

$$时钟周期_{8路}=1.14\times 时钟周期_{1路}$$

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。使用表4.5中的失效率，32K时的失效率$_{1，2，4，8路}$分别为0.02，0.014，0.013，0.013

## 4.3.4 “牺牲” Cache

1. 一种能减少冲突不命中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，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。
   - 每当发生不命中时，先检查牺牲Cache中有无所需的块，若没有，再访问主存。
   - 工作过程


地址
数据
输入
CPU
数据
输出
标志
=?
数据
=?
Victim
Cache
写缓冲器
下一级存储器

## 4.3.5 伪相联 Cache (列相联 Cache )

1. 多路组相联的低不命中率和直接映象的命中速度

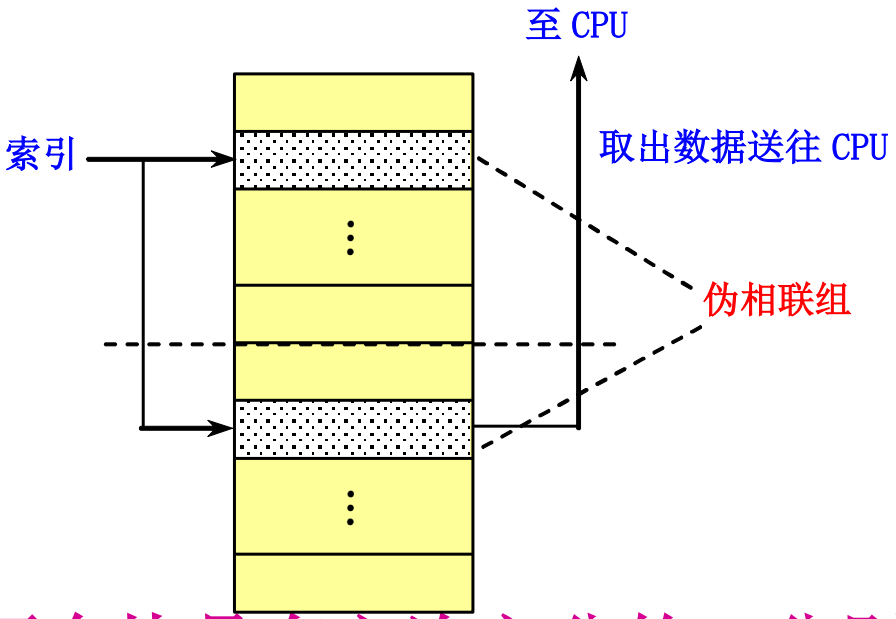
| | 优　点 | 缺　点 |
|---|---|---|
| 直接映象 | **命中时间小** | **不命中率高** |
| 组相联 | **不命中率低** | **命中时间大** |

2. 伪相联Cache的优点

- 命中时间小
- 不命中率低

## 3. 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。若正常命中，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。




伪相联组中的两个块是有主次之分的，分别称主位置和候选位置

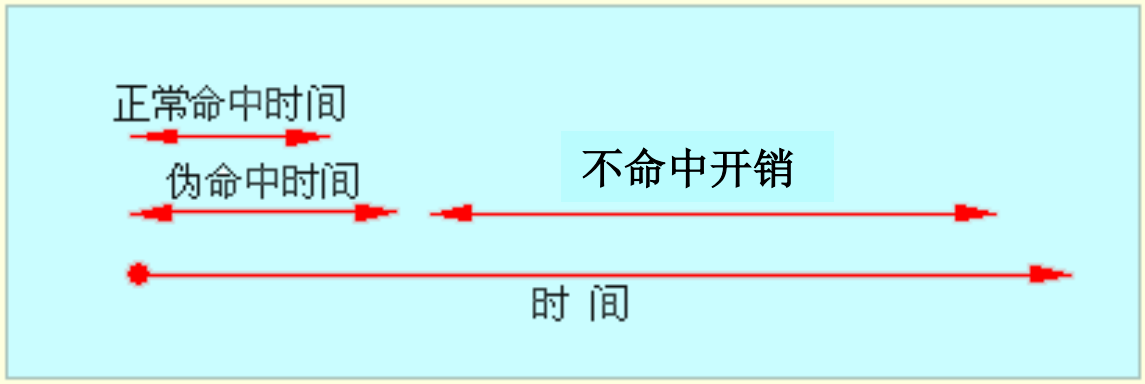
- 0000
- 0001
- 0010

- 0111
- 1000
- 1001
- 。 。 。 。

- 1111

4. 快速命中（正常命中）与慢速命中（伪命中）

要保证绝大多数命中都是快速命中。




5. 缺点：

多种命中时间会使CPU流水线设计复杂化，伪相联往往用在离处理器比较远的Cache上（如二级Cache ）。

例4.6

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)时，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2KB和128KB时，直接映象、两路组相联和伪相联这三种组织结构中，哪一种速度最快？

解:标准的平均访存时间公式为：

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

注：（1）失效开销总是相同的。

（2）“伪相联”组中的两块是用同一个索引选择得到的（正地址反地址），两路组相联Cache也是用同一个索引选择组（每组两块），因而它们的失效率相同，即：$\text{失效率}_{\text{伪相联}}=\text{失效率}_{\text{2路}}$

（3）伪相联Cache的命中时间等于直接映象Cache的命中时间加上在伪相联查找过程中命中(即伪命中)的百分比乘以该命中所需的额外时间开销，即

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

（4）$\text{伪命中率}_{\text{伪相联}}=\text{命中率}_{\text{2路}}-\text{命中率}_{\text{1路}}$（2路“组－地址取反的块”相联的硬件配置和组间直接映像，但“组”内却直接映像，而不是全相联映像。源于直接映像的改进，保持接近1路的命中速度，获得2路的低失效率）

$$=(1-\text{失效率}_{\text{2路}})-(1-\text{失效率}_{\text{1路}})\qquad=\text{失效率}_{\text{1路}}-\text{失效率}_{\text{2路}}$$

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

将表4.5中的数据代人上面的公式，得，

平均访存时间$_{伪相联}$，$T_{2KB}$=1+(0.098—0.076)×2+(0.076×50)=4.844

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

根据上一个例子中的表4.8，对于2KB的Cache，可得，

平均访存时间$_{1路}$=5.90个时钟

平均访存时间$_{2路}$=4.90个时钟

对于128K的Cache，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

| 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 | 3.75 | 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** |

可见，对于这两种Cache容量，伪相联Cache都是速度最快的

## 4.3.6 硬件预取

发生指令失效时取两个块（32字），被请求指令块放入Cache，下一块放入指令流缓冲器中；如果被请求的指令块恰好在指令流缓冲器中，直接取出，并预取下一指令块。

1. 指令和数据都可以在CPU提出访问请求之前进行预取。
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. 指令预取通常由Cache之外的硬件完成
4. 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。

1. Jouppi的研究结果表明：对于块大小为16字节、容量为4KB的直接映象指令Cache，指令流缓冲器拥有1个块可以捕获15%—25%的失效，4个块可以捕获大约50%的失效，而16个块则可以捕获72%的失效。
2. 经Jouppi统计，一个数据流缓冲器大约可以捕获4KB直接映象Cache的25%的失效。对于数据Cache，采用多个数据流缓冲器，分别从不同的地址预取数据。Jouppi发现，用4个数据流缓冲器可以将命中率提高到43%。
3. Palacharla和Kessler于1994年针对一组科学计算程序，研究了既能预取指令又能预取数据的流缓冲器。他们发现，对于一个具有两个64KB四路组相联Cache(分别用于指令和数据)的处理器来说，8个流缓冲器能够捕获其50%—70%的失效。

例4.7 Alpha AXP21064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 AXP21064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1个时钟周期。修改后的公式为：

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

假设预取命中率为25%，命中时间为1个时钟周期，失效开销为50个时钟周期。从表4.4可知，8KB指令Cache的失效率为1.10%，则

$$\text{平均访存时间}_{\text{预取}}=1+1.10\%\times 25\%\times 1+1.10\%\times(1-25\%)\times 50$$
$$=1+0.00275+0.413=1.415\text{时钟周期}$$

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=(1.415-1)/50=0.83%

计算结果说明，采用预取功能后，8KB的Cache的实际失效率为0.83%。由表4.4可知，16KB指令Cache的失效率为0.64%，所以采用预取后，8KB的Cache的失效率介于普通8KB Cache的失效率1.10%和16KB Cache的失效率0.64%之间。

预取建立在利用存储器的空闲频带的基础上。但是，如果它影响了对正常失效的处理，就可能会降低性能。

| 容量 | 指令 Cache | 数据 Cache | 混合 Cache |
|---|---|---|---|
| 1KB | 3.06% | 24.61% | 13.34% |
| 2KB | 2.26% | 20.57% | 9.78% |
| 4KB | 1.78% | 15.94% | 7.24% |
| 8KB | 1.10% | 10.19% | 4.57% |
| 16KB | 0.64% | 6.47% | 2.87% |
| 32KB | 0.39% | 4.82% | 1.99% |
| 64KB | 0.15% | 3.77% | 1.35% |
| 128KB | 0.02% | 2.88% | 0.95% |

## 4.3.7 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. 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。
2. 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。

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
   - 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
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。

## 4.3.8 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）

### 1. 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。
- 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。（提高大Cache块的效率）

- 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。
  - 对数据进行变换，以改善其空间局部性和时间局部性（例如，对数组的运算可变换为对存放在同一Cache块中的所有数据的操作）

## ➢ 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

## 2. 内外循环交换

举例：（原则：尽量按数据在主存中存储的顺序进行访问）

```
/* 修改前 */以100个字的跨距访问存储器，空间局部性不好
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 */顺序依次访问同一块中元素，再下一块；提高了局部性
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

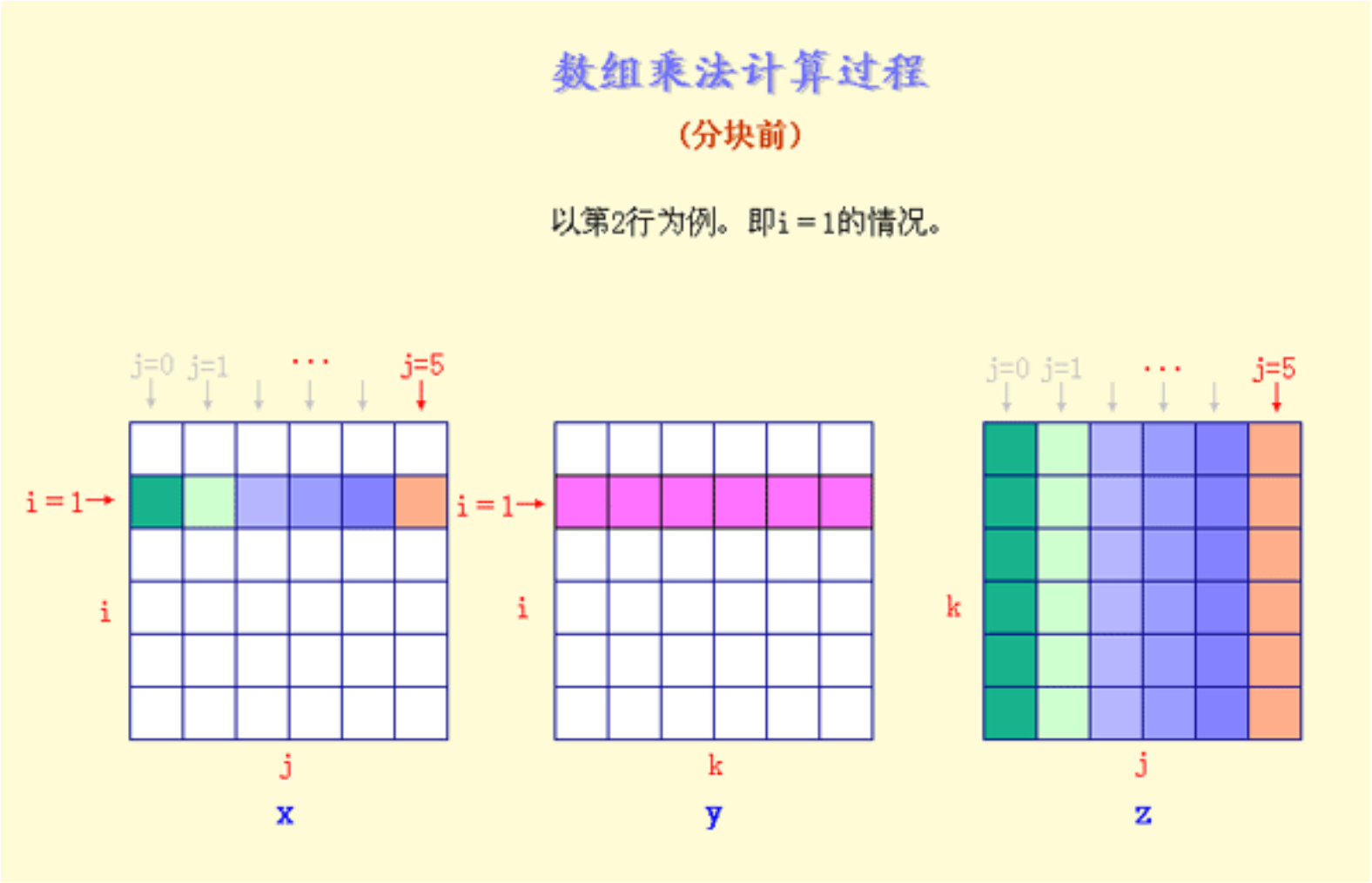
把对数组的整行或整列访问改为按块进行。

目的就是使一个Cache块被替换前最大限度利用它。

```
/*   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

两个内部循环读取了数组z的全部N×N个元素，并反复读取数组y的某一行中的N个元素，所产生的N个结果写入数组x的某一行。

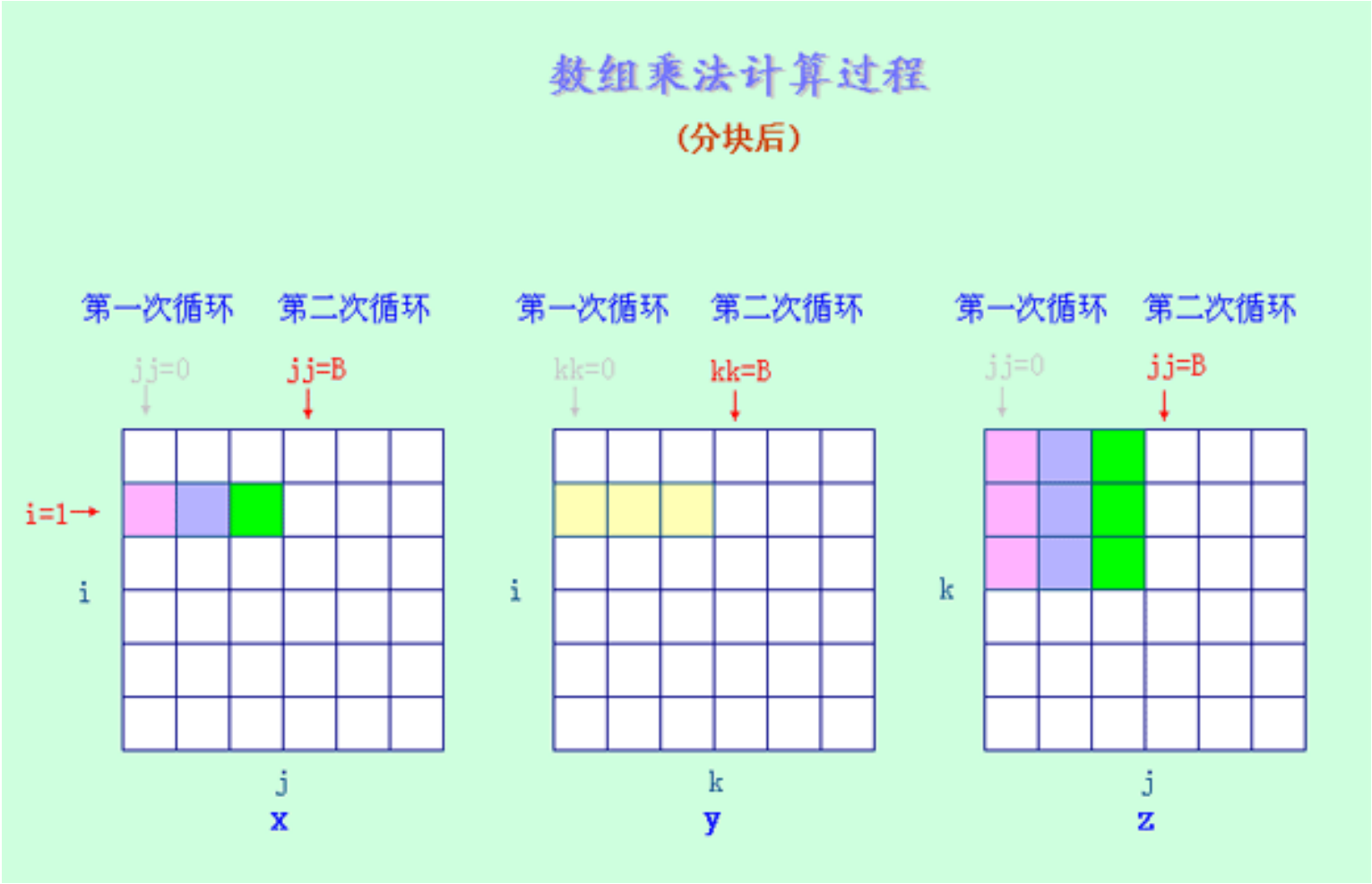



若Cache容量很小时，不命中次数会很频繁。

```
/*  修改后  */改为只对大小为B×B的块（子数组）进行计算
  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

# 4.3 降低Cache不命中率



# 4.4 减少Cache不命中开销

# 4.4 减少Cache不命中开销

## 4.4.1 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$_{L1}$＋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$_{L2}$）

3. 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

# 4.4 减少Cache不命中开销



- 全局不命中率$_{L2}$＝不命中率$_{L1}$×不命中率$_{L2}$

  评价第二级Cache时，全局不命中率这个指标更有用。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4.10 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

# 4.4 减少Cache不命中开销



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。
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。

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。

# 4.4 减少Cache不命中开销



例4.11 给出有关第二级Cache的以下数据：

（1）对于直接映象，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） 对于直接映象，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 对一个直接映象的第二级Cache来说，第一级Cache的不命中开销为：

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

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象。

➢ 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。

## 4.4.2 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

例4.12 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)
```

假设Cache采用写直达法和直接映象，并且地址512和1024映射到同一块，写缓冲器为4个字，试问寄存器R2的值总等于R3的值吗？

解:

（1）这是一个存储器写后读数据相关。

（2）分析：对Cache的访问来看，

在执行STORE指令之后，R3中的数据被放入写缓冲器。

第一条LOAD指令使用相同的Cache索引（地址512和1024映射到同一块），因而产生一次失效。

第二条LOAD指令欲把地址为512的存储单元的值读入寄存器R2中，这也会造成一次失效。如果此时写缓冲器还未将数据写入存储单元512中，那么第二条让LOAD指令将把错误的旧值读入Cache和寄存器中。

结论：如果不采取适当的预防措施，R2读的值就不会等于R3的值。

2. 解决问题的方法(读失效的处理)

**◆ 推迟对读失效的处理：读操作检查写缓冲器，为空才会对读失效处理。**

**(缺点：读失效的开销增加，如50%)**

**◆ 检查写缓冲器中的内容：如有需要的内容，先送给cpu，在写回到下一级存储。 (读失效优先于写)**

3. 在写回法Cache中，也可采用写缓冲器，减少不命中开销

**假定读失效将替换一个“脏”的存储块。可以不像往常那样先把“脏”块写回存储器，然后再读存储器，而是先把被替换的“脏”块拷入一个缓冲器，然后读存储器，最后再写存储器。这样CPU的读访间就能更快的完成。和上面的情况类似，发生读失效时，处理器既可以采用等待缓冲区清空的方法，也可以采用检查缓冲区中各字的地址是否有冲突的方法。**

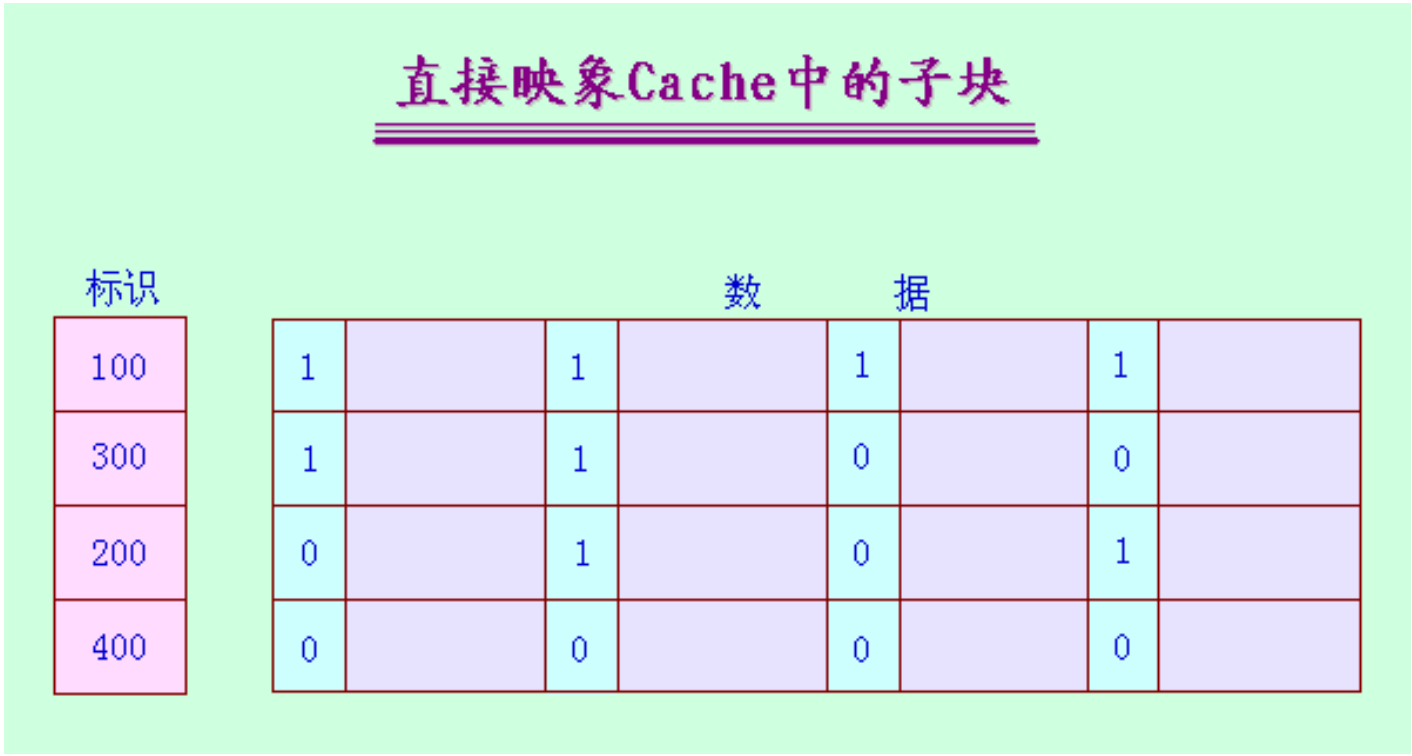
## 4.4.3 子块放置技术

1. 为减少标识的位数，可采用增加块大小的方法，但这会增加失效开销，故应采用子块放置技术。

2. 子块放置技术：把Cache块进一步划分为更小的块(子块)，并给每个子块赋予一位有效位，用于指明该子块中的数据是否有效。Cache与下一级存储器之间以子块为单位传送数据。但标识仍以块为单位。

3. 举例

# 4.4 减少Cache不命中开销




说明：

（1）标识匹配并不能意味着这个字在Cache中命中，只有当与该字对应的有效位也为“1”时才是。

（2）失效时只需从下一级存储器调入一个子块。

（3）一个Cache中就有可能有的子块有效，有的子块无效。

（4）子块的失效开销小于完Cache块的失效开销。

（5）子块可以被看作是地址标识之外的又一级寻址。

## 4.4.4 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，CPU启动。**

◆ **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的其余部分。也称为回绕读取(wrapped fetch)或关键字优先(chtical word first)。**

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（上）一部分**

## 4.4.5 非阻塞Cache技术

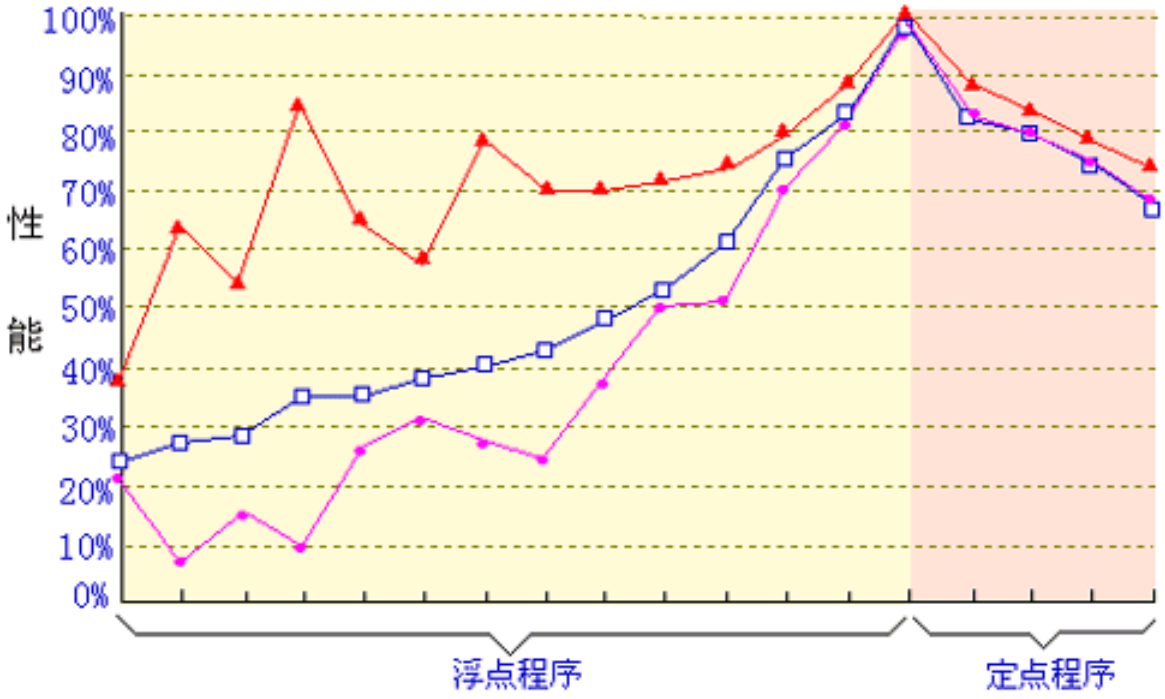
1. 非阻塞Cache：Cache失效时仍允许CPU进行其它的命中访问。即允许“失效下命中”。

2. 进一步提高性能：“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% |

# 非阻塞Cache平均存储器等待时间与阻塞Cache的比值

| 重叠失效个数 | 1 | 2 | 64 |
|---|---|---|---|
| 浮点程序 | 76% | 51% | 39% |
| 整数程序 | 81% | 78% | 78% |

例 4.13

**在两路组相联和“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%，两路组相联Cache为10.7%；**

**对于整数程序，直接映象Cache为7.4%，两路组相联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：

$$失效率_{直接映象} \times 失效开销 = 11.4\% \times 16 = 1.82$$

$$失效率_{两路组相联} \times 失效开销 = 10.7\% \times 16 = 1.71$$

$$1.71/1.82 = 0.94$$

4.4 减少Cache 失效开销

| 重叠失效个数 | 1 | 2 | 64 |
|---|---|---|---|
| 浮点程序 | 76% | 51% | 39% |
| 整数程序 | 81% | 78% | 78% |

即两路组相联Cache的平均存储器等待时间是直接映象Cache的94%，而支持“一次失效下命中”技术的直接映象Cache的平均存储器等待时间是直接映象Cache的76%，所以对于浮点程序来说，支持“一次失效下命中”的直接映象Cache比两路组相联Cache的性能更高。

对于整数程序：

失效率$_{直接映象}$×失效开销＝7.4 %×16＝1.18

失效率$_{两路组相联}$×失效开销＝6.0 %×16 ＝0.96

0.96/1.18=0.81

即整数计算中，两路组相联Cache的平均存储器等待时间是直接映象Cache的81%，而“一次失效下命中”技术把平均存储器等待时间降低到直接映象Cache的81%。因此，对于整数程序来说，两种技术的性能相同。

“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。

# 4.5 减少命中时间

# 4.5 减少命中时间

命中时间直接影响到处理器的时钟频率。往往是Cache的访问时间限制了处理器的时钟频率。

## 4.5.1 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，以便可与CPU放在同一块芯片上。

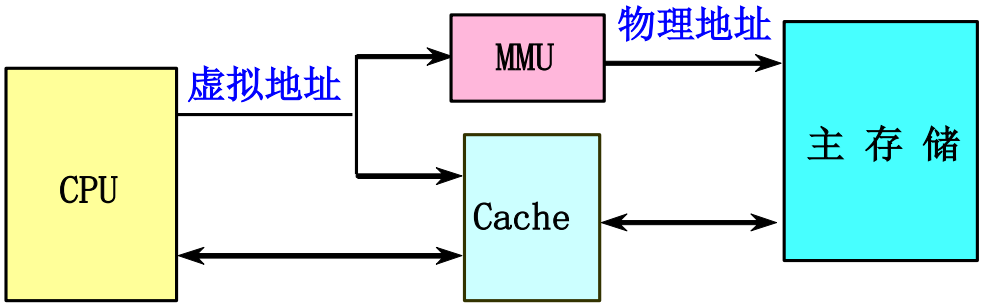
某些设计采用了一种折衷方案：

把Cache的标识放在片内，而把Cache的数据存储器放在片外。

## 4.5.2 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。



## 4.5.3 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

## 4.5.4 踪迹 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：采用踪迹 Cache

   存放CPU所执行的动态指令序列*1

3. 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。

## 4.5.5 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 | 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 |

# 4.6 并行主存系统

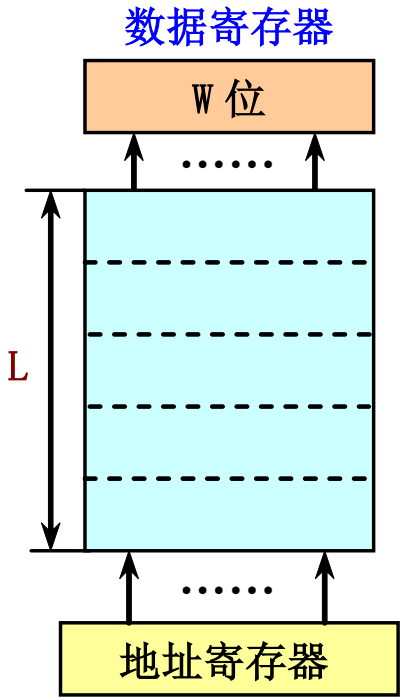
# 4.6 并行主存系统

- 以往：主存的主要性能指标：延迟和带宽
  - Cache主要关心延迟，I/O主要关心带宽。
- 现在：Cache既关心延迟，也非常关心带宽
  - 因为二级Cache的广泛应用，二级Cache的块比较大
- 并行主存系统是在一个访存周期内能并行访问多个存储字的存储器。
  - 能有效地提高存储器的带宽。

➢ 一个单体单字宽的存储器

- 字长与CPU的字长相同。
- 每一次只能访问一个存储字。
  假设该存储器的访问周期是$T_M$，
  字长为W位，则其带宽为：

$$B_M = \frac{W}{T_M}$$

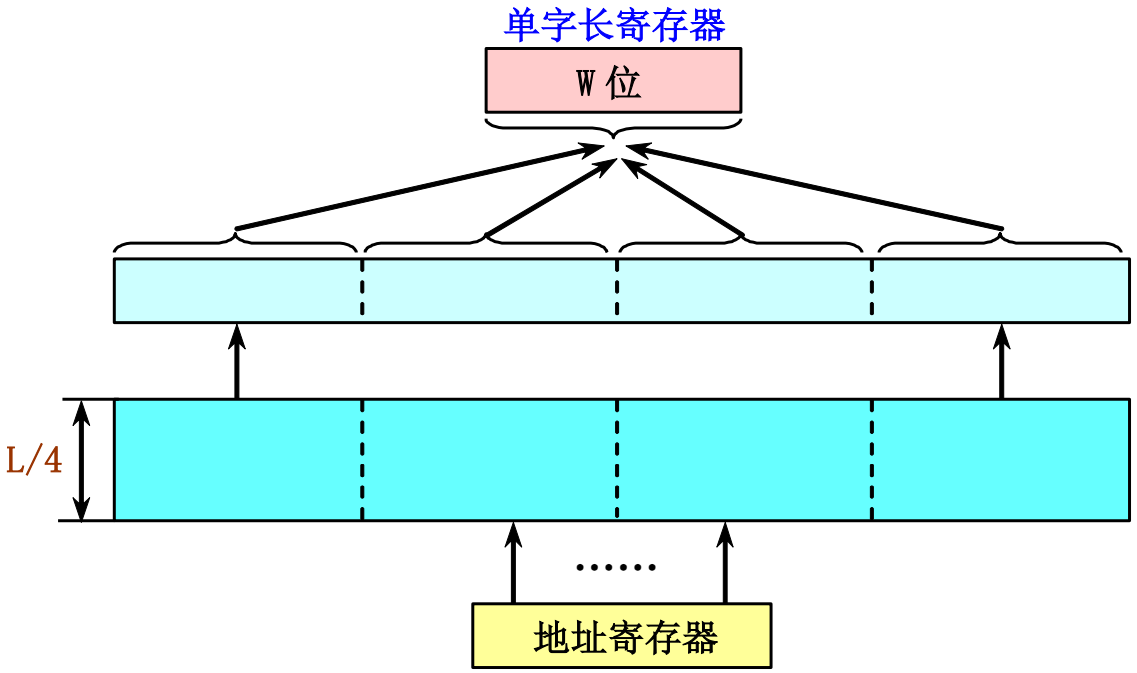



普通存储器

- 在相同的器件条件（即$T_M$相同）下，可以采用两种并行存储器结构来提高主存的带宽：
  - 单体多字存储器
  - 多体交叉存储器

## 4.6.1 单体多字存储器

### 1.一个单体m字（这里m=4）存储器



- 存储器能够每个存储周期读出m个CPU字。因此其最大带宽提高到原来的m倍 。

$$B_M = m \times \frac{W}{T_M}$$

- 单体多字存储器的实际带宽比最大带宽小

## 2. 优缺点

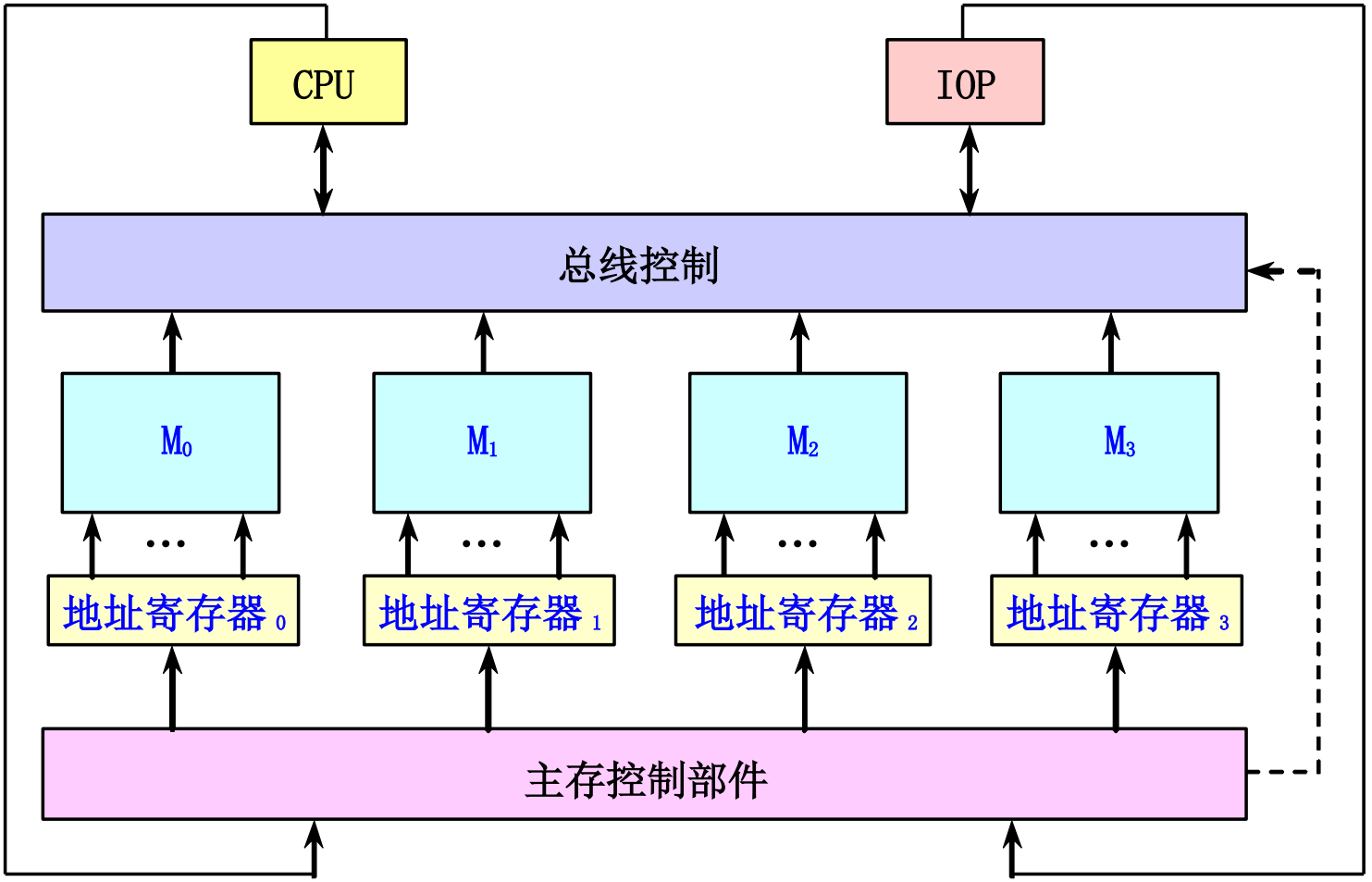
- 优点：实现简单
- 缺点：访存效率不高

原因 ：

- 如果一次读取的m个指令字中有分支指令，而且分支成功，那么该分支指令之后的指令是无用的。
- 一次取出的m个数据不一定都是有用的。另一方面，当前执行指令所需要的多个操作数也不一定正好都存放在同一个长存储字中。
- 写入有可能变得复杂。
- 当要读出的数据字和要写入的数据字处于同一个长存储字内时，读和写的操作就无法在同一个存储周期内完成。

## 4.6.2 多体交叉存储器

1. 多体交叉存储器：由多个单字存储体构成，每个体都有自己的地址寄存器以及地址译码和读/写驱动等电路。
2. 问题：对多体存储器如何进行编址？
   - 存储器是按顺序线性编址的。如何在二维矩阵和线性地址之间建立对应关系？
   - 两种编址方法
     - 高位交叉编址
     - 低位交叉编址 （有效地解决访问冲突问题 ）

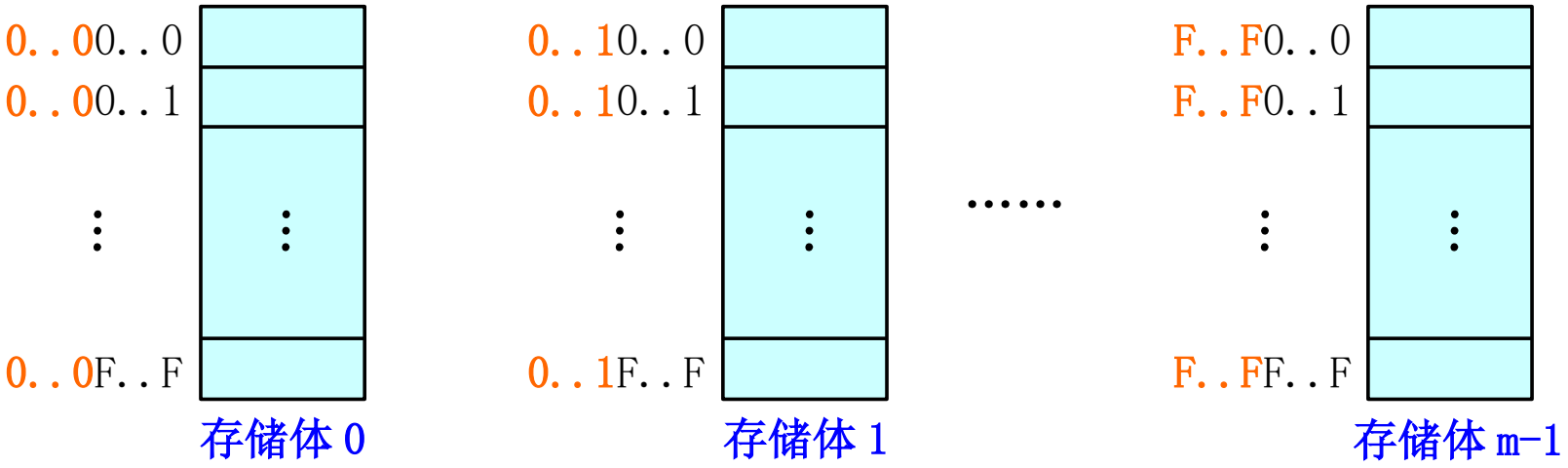



多体（m=4）交叉存储器

# 4.6 并行主存系统

## 3. 高位交叉编址

- 对存储单元矩阵按列优先的方式进行编址
- 特点：同一个体中的高$\log_2 m$位都是相同的（体号）




（共m个存储体，每个存储体n个存储单元）

# 4.6 并行主存系统

- 处于第i行第j列的单元，即体号为j、体内地址为i的单元，其线性地址为：

$$A = j \times n + i$$

其中：$j = 0, 1, 2, \ldots, m-1$

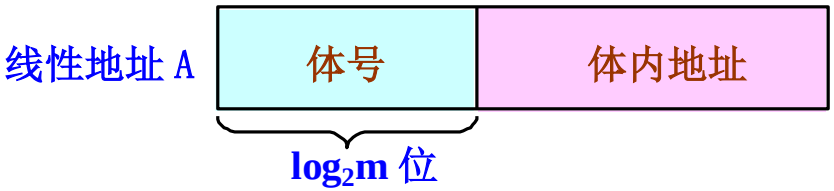
$i = 0, 1, 2, \ldots, n-1$

- 一个单元的线性地址为A，则其体号j和体内地址i为：

$$i = A \bmod n$$

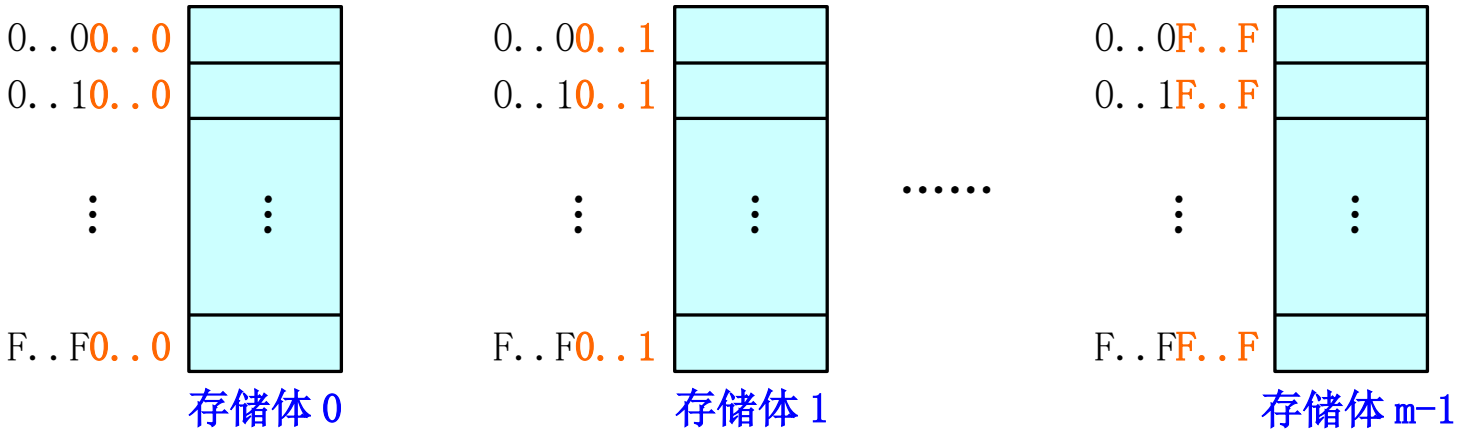
$$j = \left\lfloor \frac{A}{n} \right\rfloor$$

- 把A表示为二进制数，则其高$\log_2 m$位就是体号，而剩下的部分就是体内地址。




## 4. 低位交叉编址

- 对存储单元矩阵按行优先进行编址
- 特点：同一个体中的低$\log_2 m$位都是相同的（体号）




- 处于第i行第j列的单元，即体号为j、体内地址为i的单元，其线性地址为：

$$A=i\times m+j$$

其中：i＝0，1，2，…，n－1

j＝0，1，2，…，m－1

# 4.6 并行主存系统

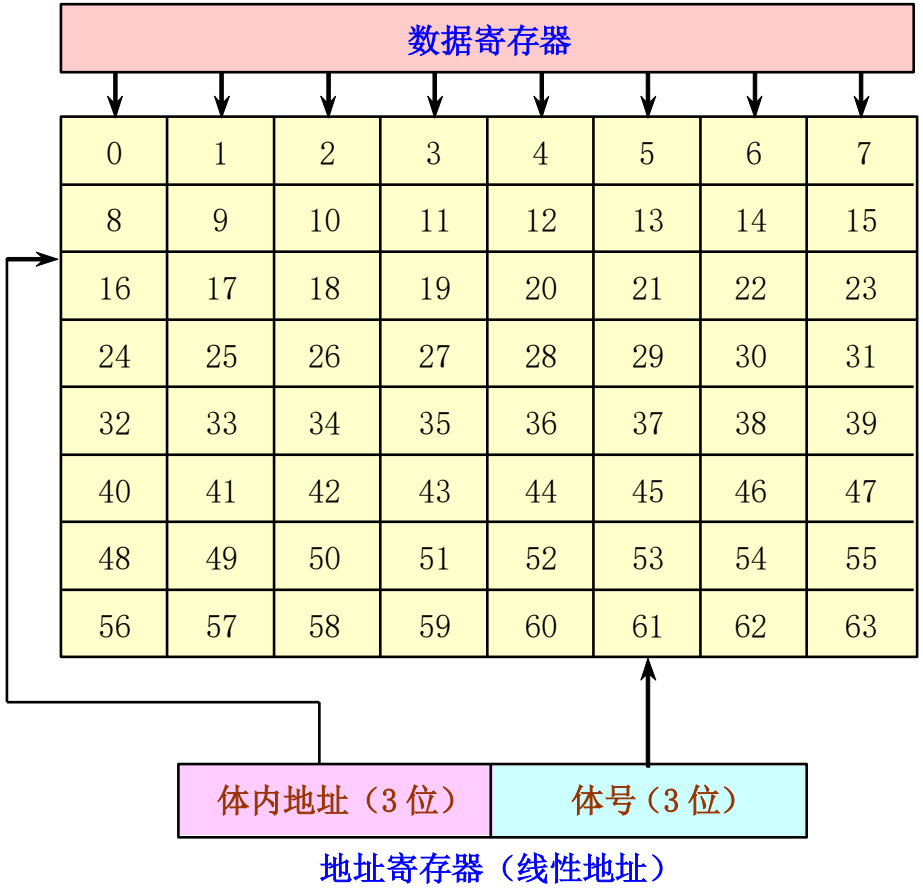
- 一个单元的线性地址为A，则其体号j和体内地址i为：

$$i = \left\lfloor \frac{A}{m} \right\rfloor \qquad j = A \bmod m$$

- 把A表示为二进制数，则其低$\log_2 m$位就是体号，而剩下的部分就是体内地址。

– 例：采用低位交叉编址的存储器

由8个存储体构成、总容量为64。格子中的编号为线性地址。

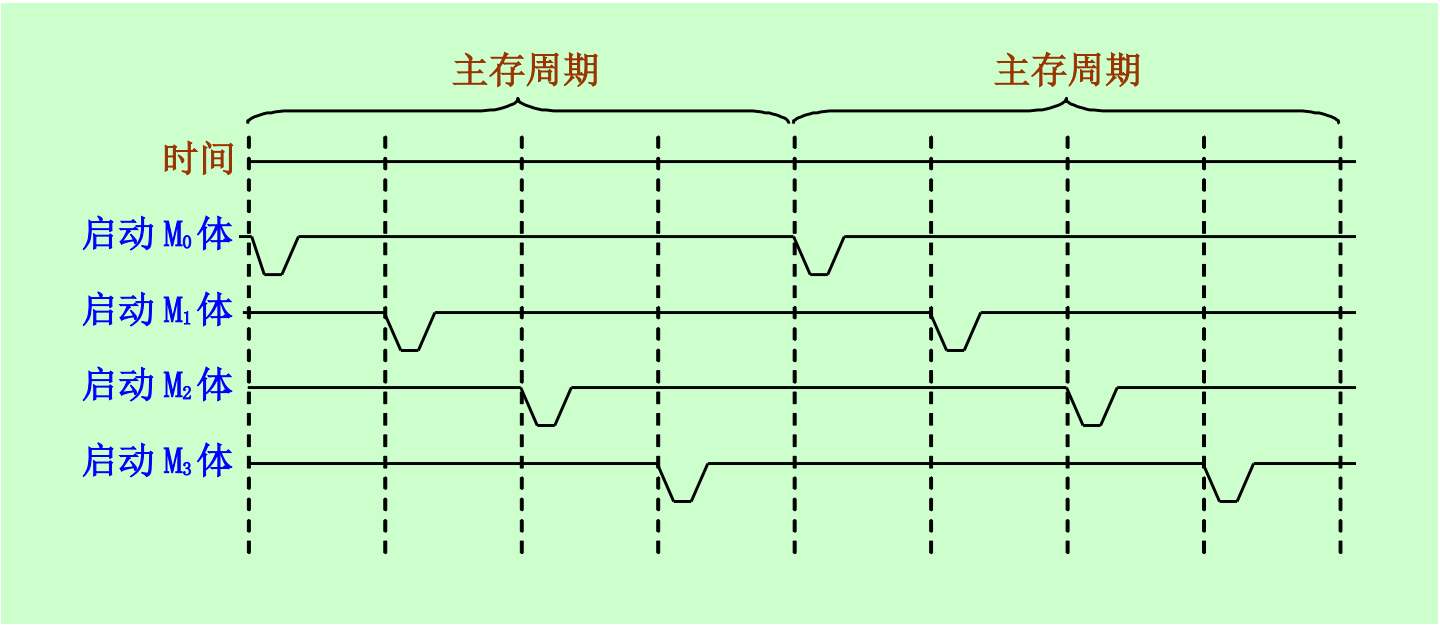


# 4.6 并行主存系统



- 为了提高主存的带宽，需要多个或所有存储体能并行工作。

  - 在每一个存储周期内，分时启动m个存储体。
  - 如果每个存储体的访问周期是$T_M$，则各存储体的启动间隔为： $t=T_M/m$。



# 4.7 虚拟存储器

# 4.7 虚拟存储器

## 4.7.1 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）
   - 段式虚拟存储器则把空间划分为可变长的块。（段）
   - 页面是对空间的机械划分，而段则往往是按程序的逻辑意义进行划分。
   - 多数计算机采用段式和页式的结合：段页式

## 3. Cache和虚拟存储器的参数取值范围

| 参数 | 第一级Cache | 虚拟存储器 |
|---|---|---|
| 块（页）大小 | 16-128字节 | 4096-65,536字节 |
| 命中时间 | 1-3个时钟周期 | 100-200个时钟周期 |
| 不命中开销 | 8-200个时钟周期 | 1,000,000-10,000,000个时钟周期 |
| （访问时间） | （6-160个时钟周期） | （800,000-8,000,000个时钟周期） |
| （传输时间） | （2-40个时钟周期） | （200,000-2,000,000个时钟周期） |
| 不命中率 | 0.1-10% | 0.00001-0.001% |
| 地址映象 | 25-45位物理地址到14-20位Cache地址 | 32-64位虚拟地址到25-45位物理地址 |

## 4.7.2 快速地址转换技术

### 1. 地址变换缓冲器TLB（Translation Look-aside Buffer）

- 页表一般都很大，主要存放在主存中。常使用快表TLB快速地完成虚实地址转换。
- 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；
- TLB也利用了局部性原理。

2. TLB中的项由两部分构成：标识和数据
   - 标识中存放的是虚地址的一部分。
   - 数据部分中存放的则是物理页帧号、有效位、存储保护信息、使用位、修改位等。
3. AMD Opteron的数据TLB的组织结构
   - 包含40个项
   - 采用全相联映象
   - AMD Opteron的地址转换过程
4. 一般TLB比Cache的标识存储器更小、更快。

   保证TLB的读出操作不会使Cache的命中时间延长。

東北林業大學
NORTHEAST FORESTRY UNIVERSITY

# THANK YOU!

2019.11.13