

第7章 存储系统

张晨曦 刘依

www.GotoSchool.net

xzhang2000@sohu.com

- 7.1 [存储系统的基本知识](#)
- 7.2 [Cache基本知识](#)
- 7.3 [降低Cache不命中率](#)
- 7.4 [减少Cache不命中开销](#)
- 7.5 [减少命中时间](#)
- 7.6 [并行主存系统](#)
- 7.7 [虚拟存储器](#)
- 7.8 [实例：AMD Opteron的存储器层次结构](#)

7.1 存储系统的基本知识

7.1.1 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

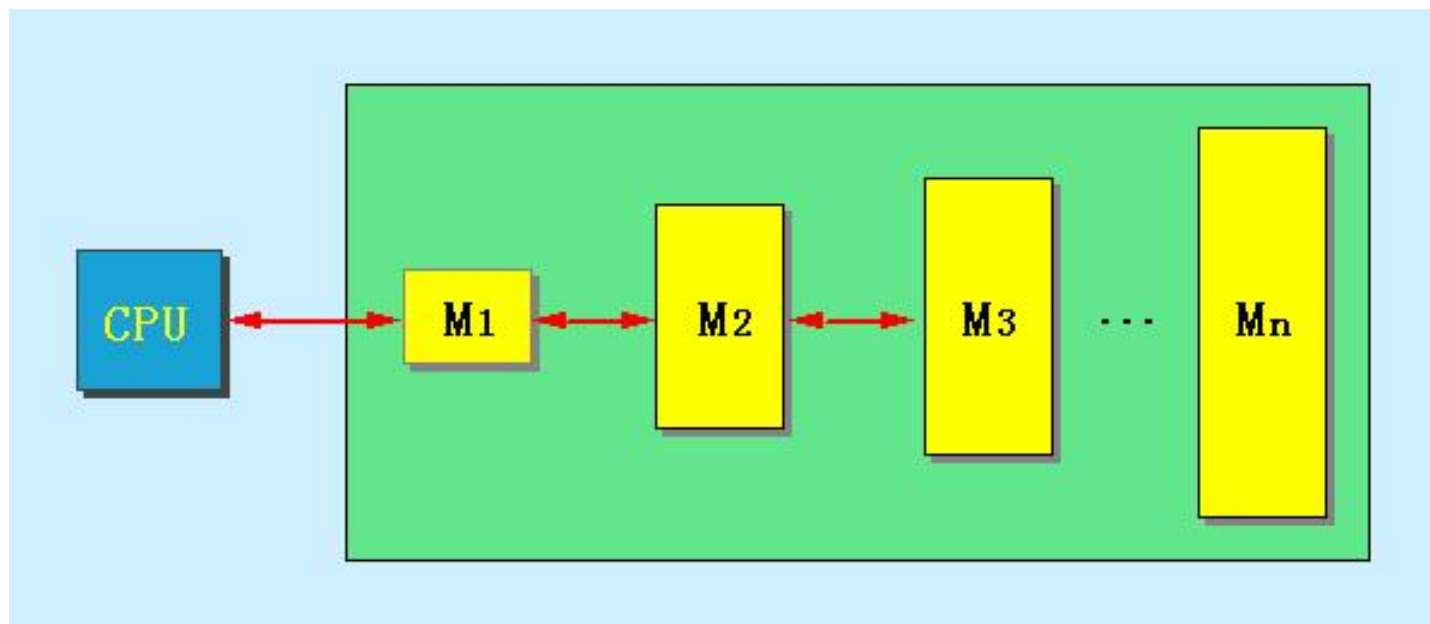
- 速度越快，每位价格就越高；
- 容量越大，每位价格就越低；
- 容量越大，速度越慢。

1. 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。 (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。

1. 存储系统的多级层次结构

演示 I 演示 II



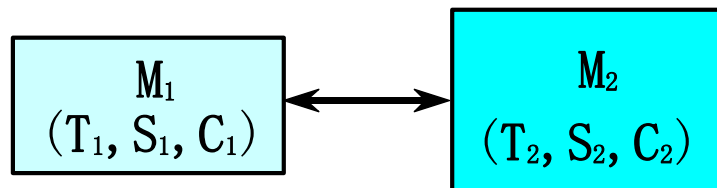
多级存储层次

- 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： $T_1 < T_2 < \dots < T_n$
 - 容量： $S_1 < S_2 < \dots < S_n$
 - 平均每位价格： $C_1 > C_2 > \dots > C_n$
- 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。

7.1.2 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： S_1 , T_1 , C_1
- M_2 的参数： S_2 , T_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 $S=S_2$ 。

2. 每位价格C

$$C = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2}$$

当 $S_1 \ll S_2$ 时， $C \approx C_2$ 。

1. 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**： $F = 1 - H$

1. 平均访问时间 T_A

$$\begin{aligned}T_A &= HT_1 + (1-H)(T_1 + T_M) \\ &= T_1 + (1-H)T_M\end{aligned}$$

$$\text{或 } T_A = T_1 + FT_M$$

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_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： $T_2 + T_B + T_1 = T_1 + T_M$

$$T_M = T_2 + T_B$$

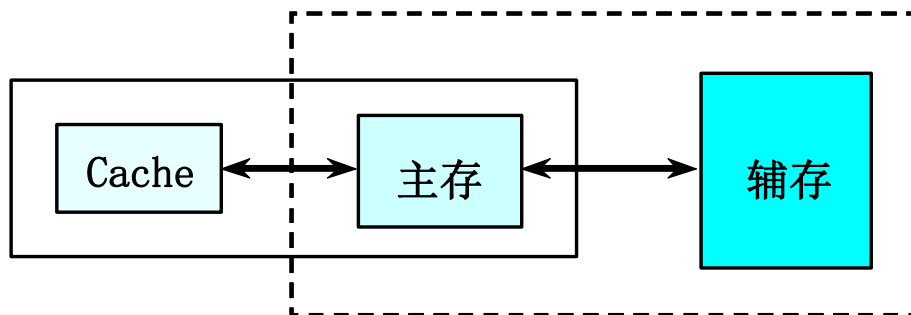
- 不命中开销 T_M ：从向 M_2 发出访问请求到把整个数据块调入 M_1 中所需的时间。
- 传送一个信息块所需的时间为 T_B 。

7.1.3 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。



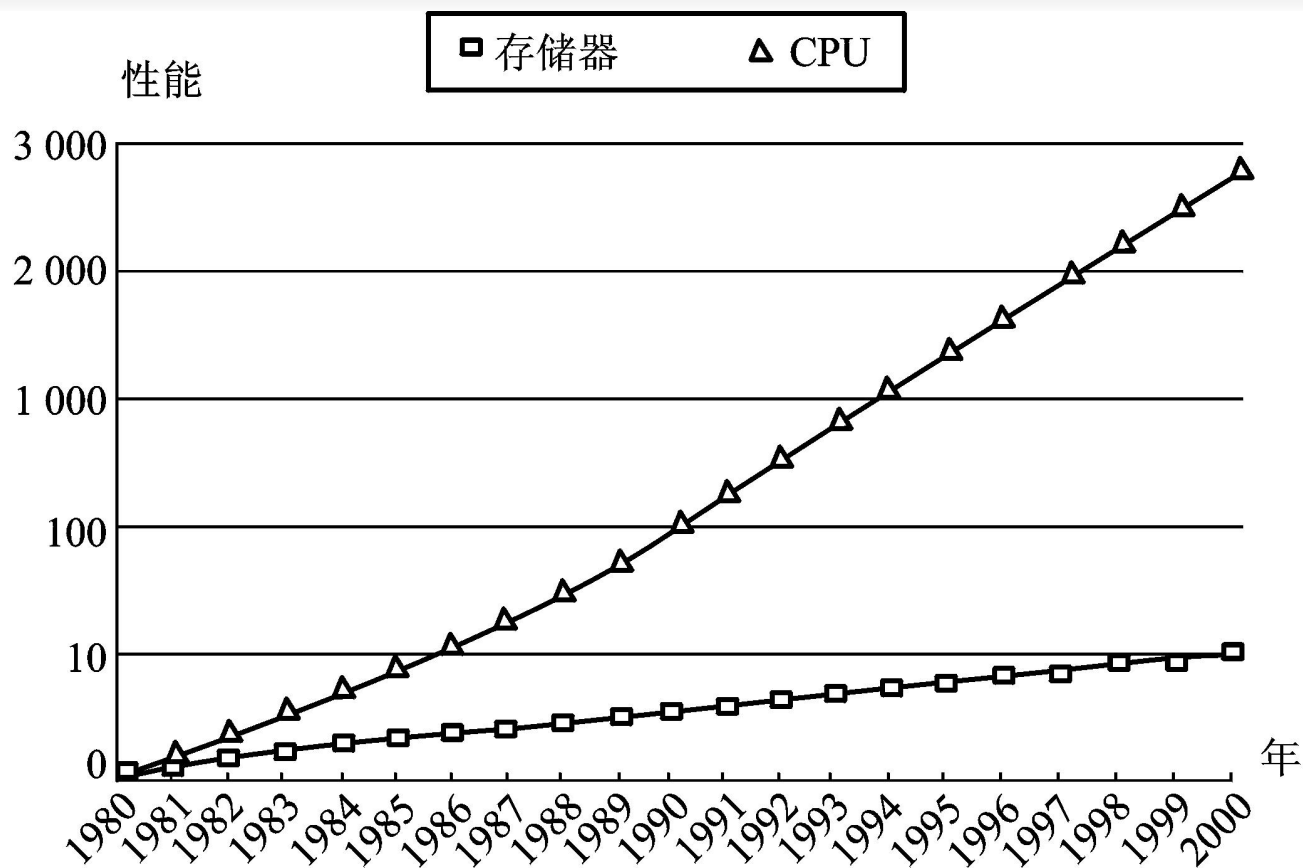
从主存的角度来看

- “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

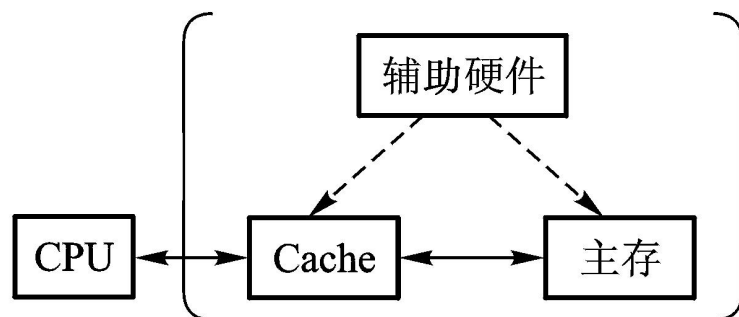
1. “Cache—主存”层次

- 主存与CPU的速度差距
- “Cache - 主存”层次

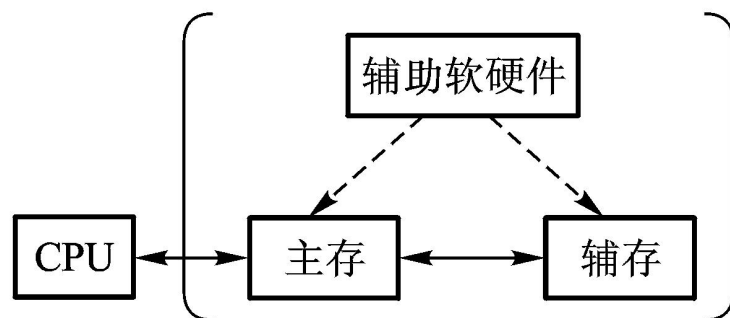
2. “主存—辅存”层次



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) “Cache-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

7.1.4 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)

7.2 Cache基本知识

7.2.1 基本结构和原理

1. 存储空间分割与地址计算

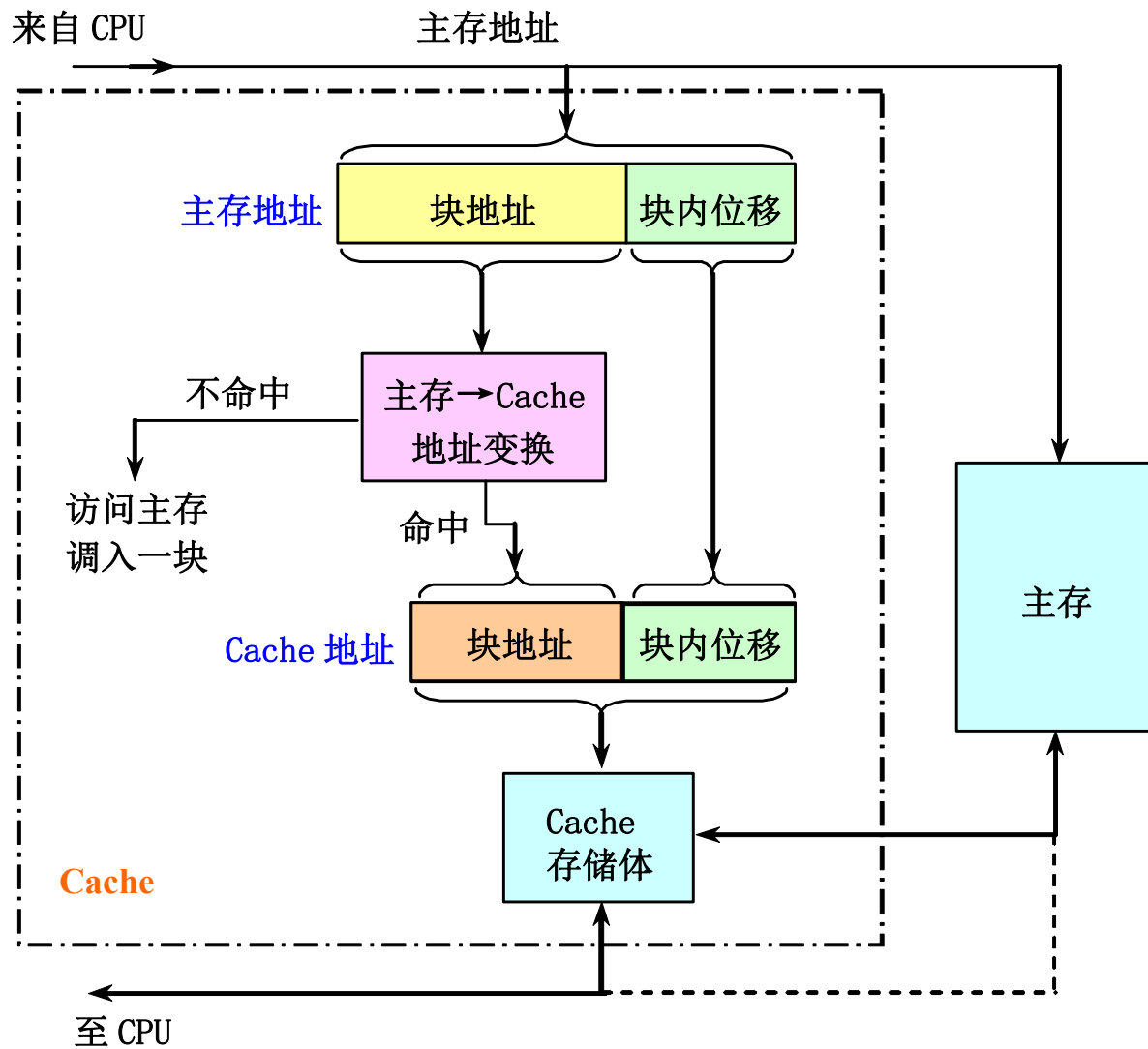
2. Cache和主存分块

- 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：

块地址	块内位移
-----	------

1. Cache的基本工作原理示意图

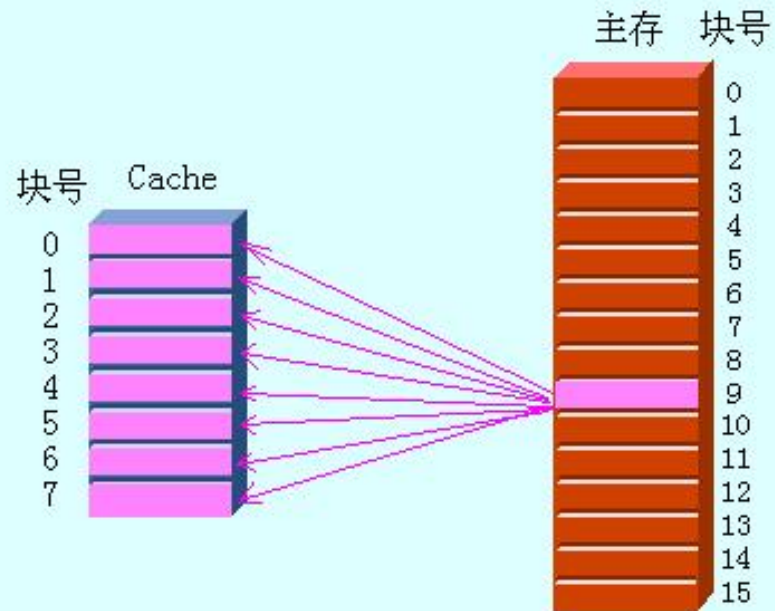


7.2.2 映象规则

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)



1. 直接映象

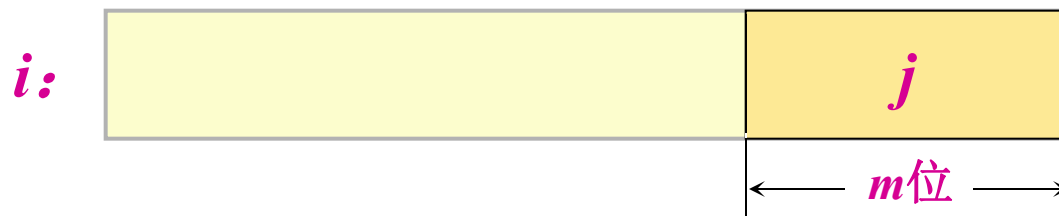
- **直接映象**：主存中的每一块只能被放置到Cache中唯一的一个位置。 举例

（循环分配）

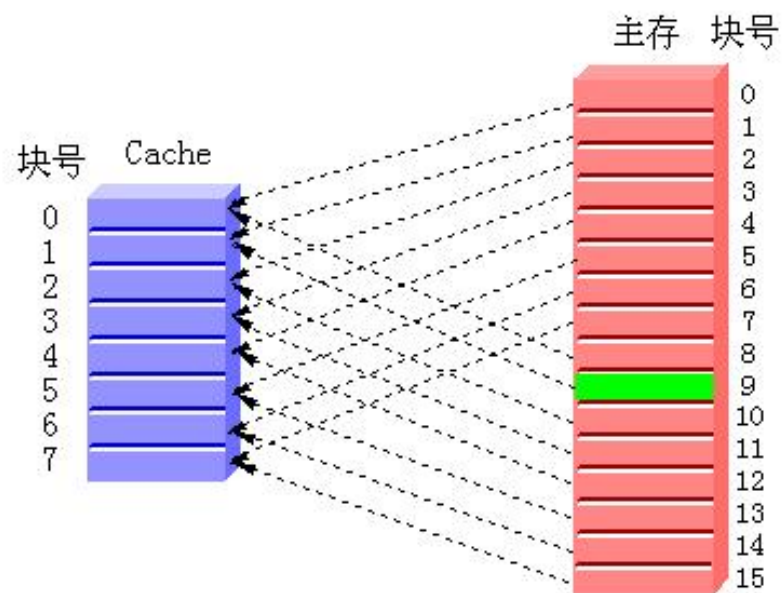
- **对比**：阅览室位置 —— 只有一个位置可以坐
- **特点**：空间利用率最低，冲突概率最高，实现最简单。
- 对于主存的第 i 块，若它映象到Cache的第 j 块，则：

$$j = i \bmod (M) \quad (M \text{ 为Cache的块数})$$

- 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：

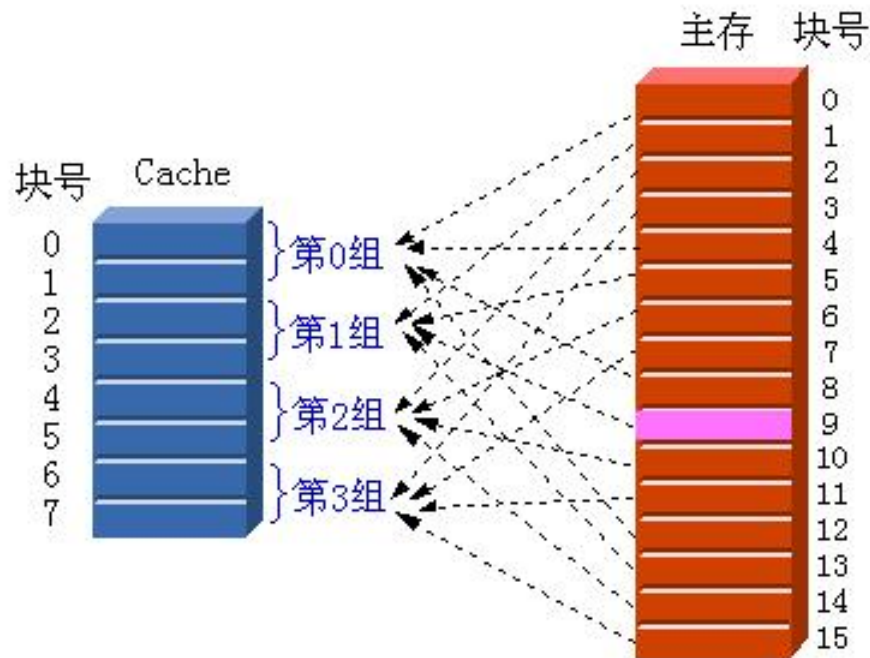


直接映射 (举例)



1. 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衷](#)

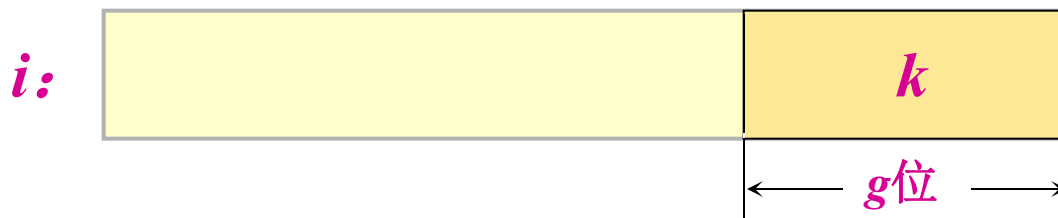


➤ 组的选择常采用位选择算法

- 若主存第 i 块映像到第 k 组，则：

$$k = i \bmod (G) \quad (G \text{ 为 Cache 的组数})$$

- 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 位以及直接映像中的低 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，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: $n \leq 4$

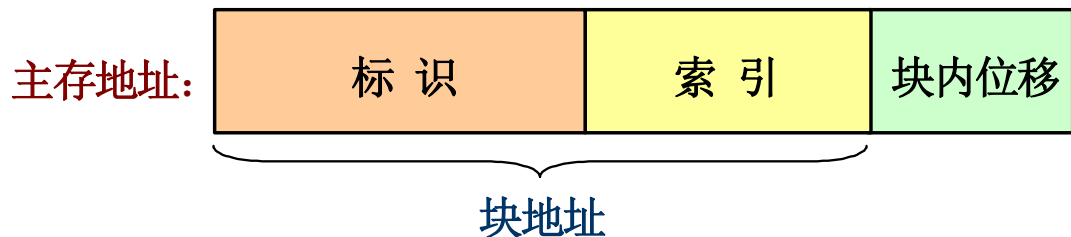
想一想：相联度一定是越大越好？

7.2.3 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？

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



- 只需查找候选位置所对应的目录表项

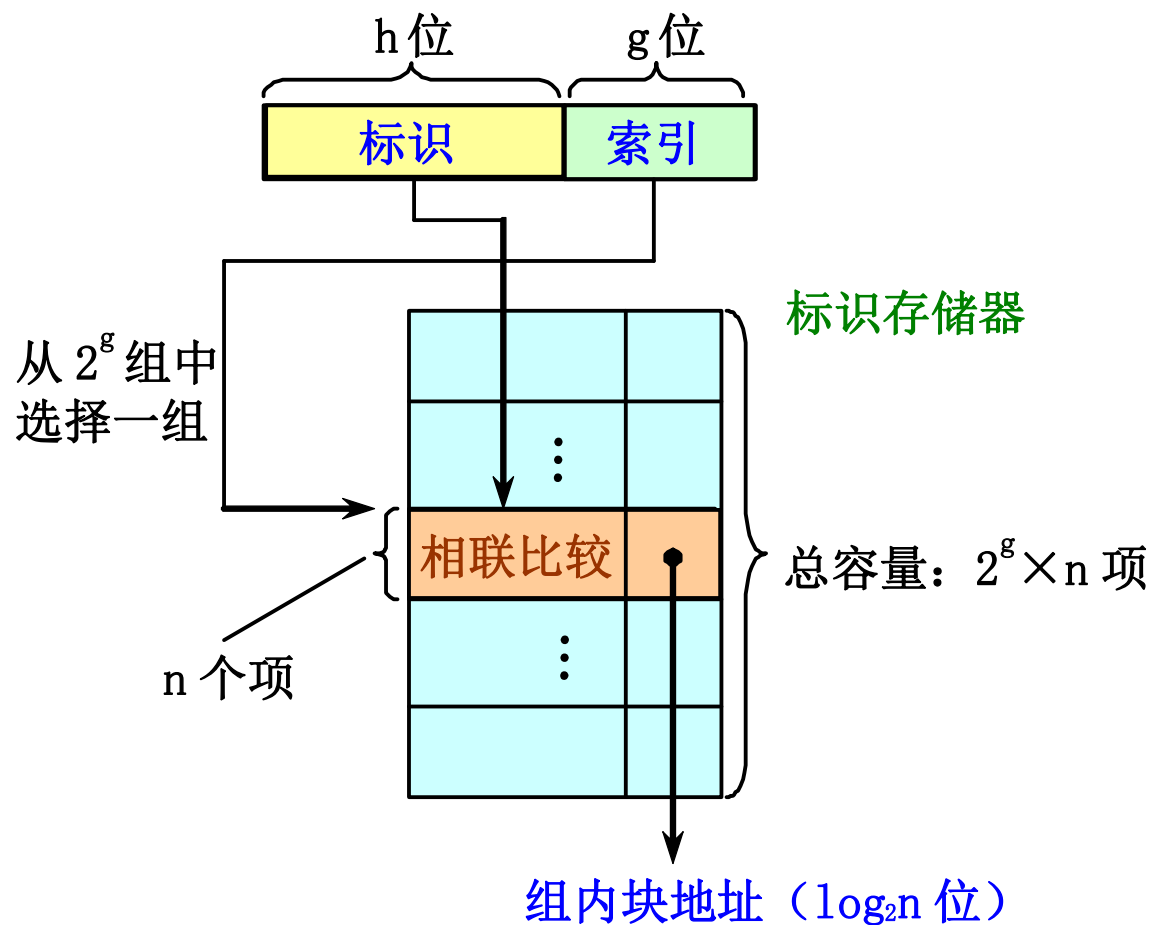
2. 并行查找与顺序查找

- 提高性能的重要思想：主候选位置(MRU块)
(前瞻执行)

1. 并行查找的实现方法

- 相联存储器

- 目录由 2^g 个相联存储区构成，每个相联存储区的大小为 $n \times (h + \log_2 n)$ 位。
- 根据所查找到的组内块地址，从Cache存储体中读出的多个信息字中选一个，发送给CPU。



➤ 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)

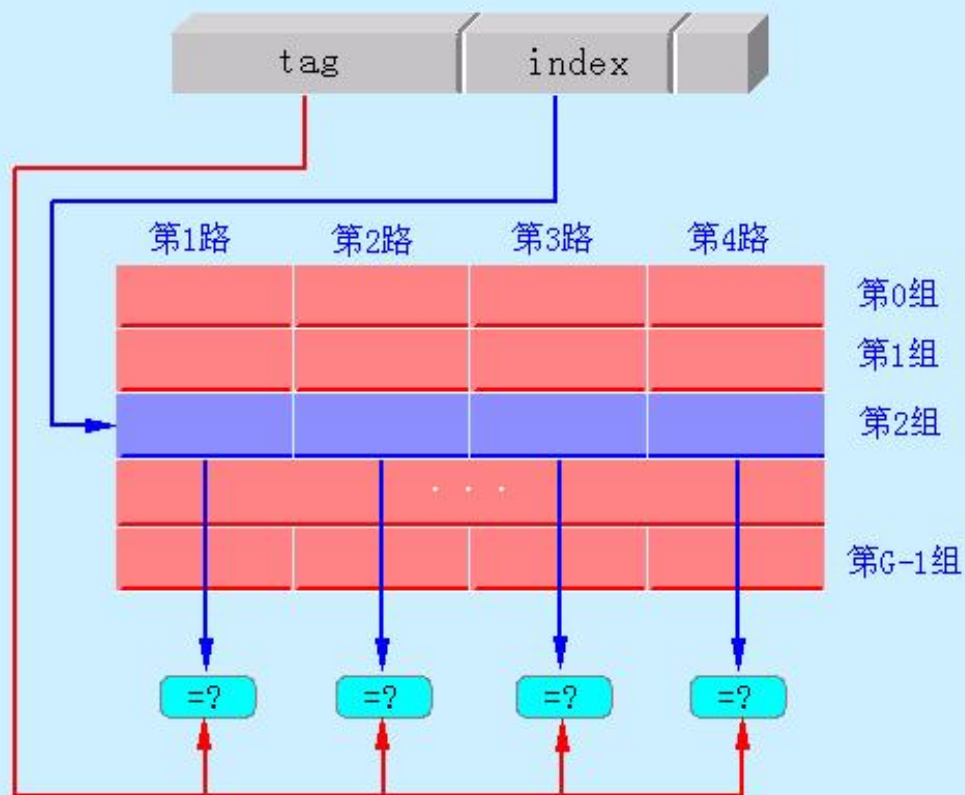
- 4路组相联Cache的查找过程

- 直接映象Cache的查找过程

- 优缺点

- 不必采用相联存储器，而是用按地址访问的存储器来实现。
- 所需要的硬件为：大小为 $2^g \times n \times h$ 位的存储器和 n 个 h 位的比较器。
- 当相联度 n 增加时，不仅比较器的个数会增加，而且比较器的位数也会增加。

4路组相联并行标识比较



7.2.4 Cache的工作过程

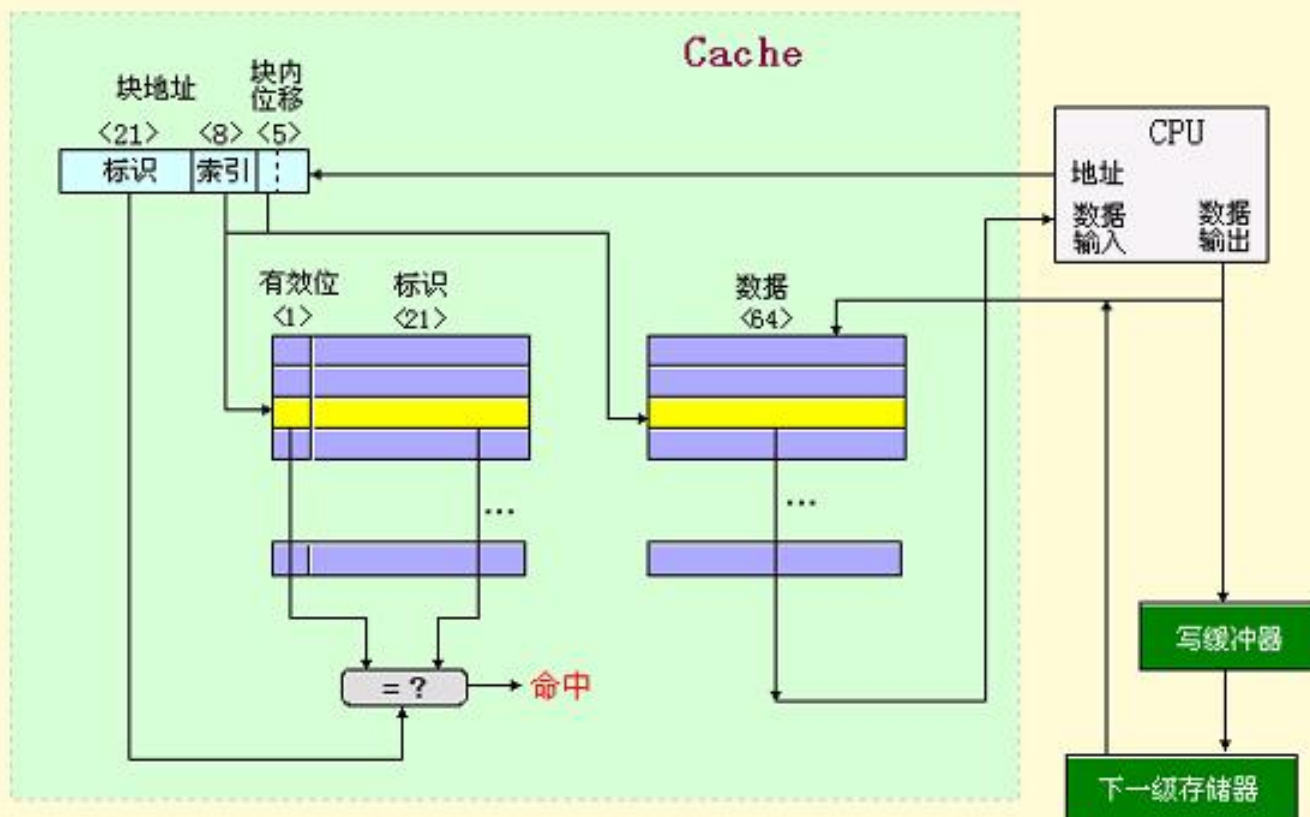
例子：DEC的Alpha AXP21064中的内部数据Cache

1. 简介

- 容量：8KB
- 块大小：32B
- 块数：256
- 映象方法：直接映象
- 写缓冲器大小：4个块

1. 结构图

Alpha AXP 21064中数据Cache的结构



1. 工作过程

➤ “读”访问命中

（完成4步需要2个时钟周期）

- Cache的容量与索引index、相联度、块大小之间的关系

Cache的容量= 2^{index} ×相联度×块大小

- 把容量为8192、相联度为1、块大小为32（字节）代入：

索引index：8位 标识： $29-8=21$ 位

➤ “写”访问命中

➤ 设置了一个写缓冲器

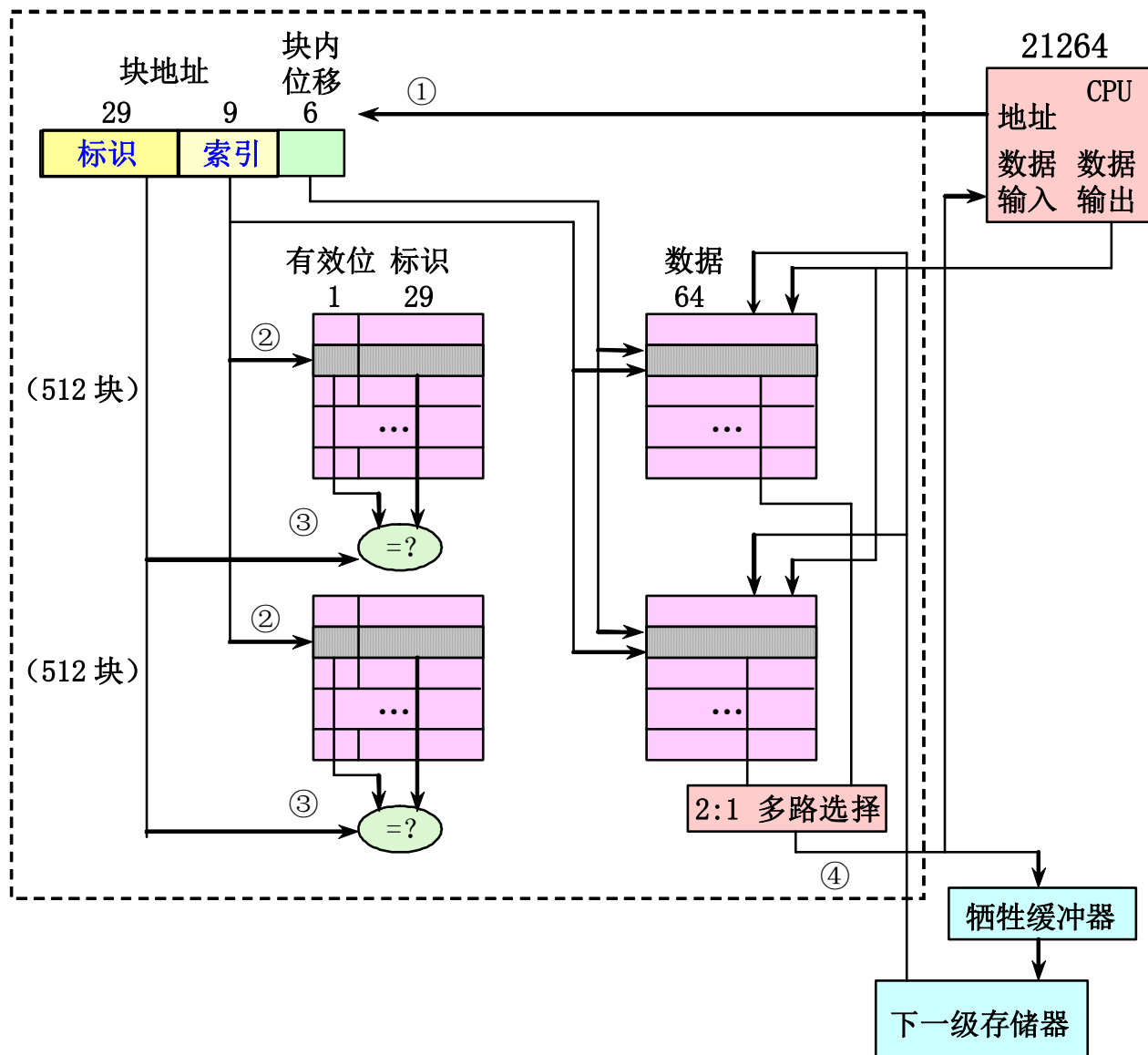
(提高“写”访问的速度)

- 按字寻址的，它含有4个块，每块大小为4个字。
- 当要进行写入操作时，如果写缓冲器不满，那么就把数据和完整的地址写入缓冲器。对CPU而言，本次“写”访问已完成，CPU可以继续往下执行。由写缓冲器负责把该数据写入主存。
- 在写入缓冲器时，要进行写合并检查。即检查本次写入数据的地址是否与缓冲器内某个有效块的地址匹配。如果匹配，就把新数据与该块合并。

- 发生读不命中与写不命中时的操作
 - 读不命中：向CPU发出一个暂停信号，通知它等待，并从下一级存储器中新调入一个数据块（32字节）。
 - 写不命中：将使数据“绕过”Cache，直接写入主存。

对比：Alpha AXP 21264的数据Cache结构

- 容量：64KB 块大小：64字节 LRU替换策略
- 主要区别
 - 采用2路组相联
 - 采用写回法
- 没有写缓冲器



7.2.5 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

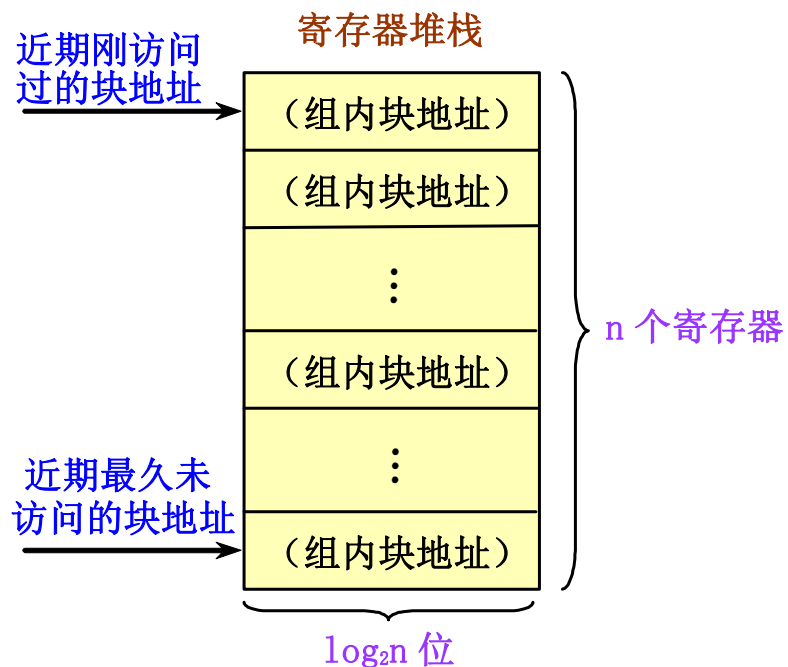
➤ LRU和随机法分别因其不命中率低和实现简单而被广泛采用。

- 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。

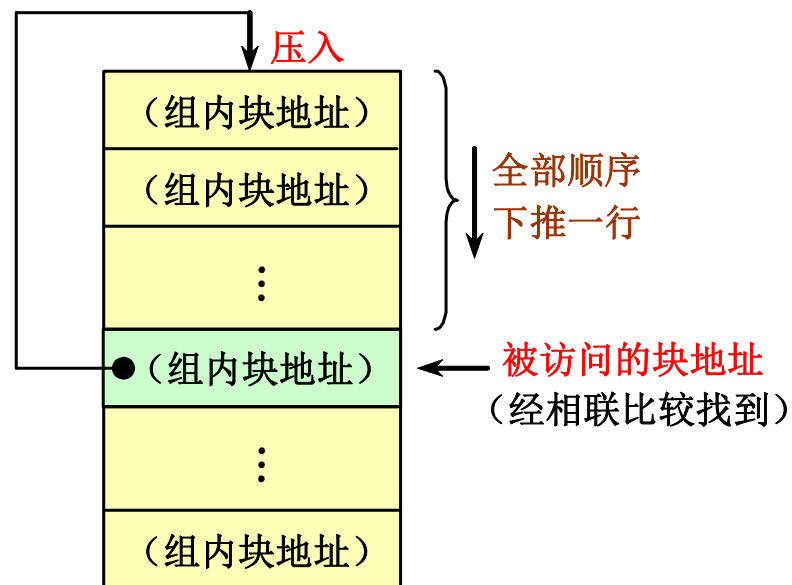
1. LRU算法的硬件实现

➤ 堆栈法

- 用一个堆栈来记录组相联Cache的同一组中各块被访问的先后次序。
- 用堆栈元素的物理位置来反映先后次序
 - 栈底记录的是该组中最早被访问过的块，次栈底记录的是该组中第二个被访问过的块，...，栈顶记录的是刚访问过的块。
 - 当需要替换时，从栈底得到应该被替换的块（块地址）。



(a) 用位置记录访问的先后次序



(b) 发生访问时所进行的操作

- 堆栈中的内容必须动态更新
 - 当Cache访问命中时，通过用块地址进行相联查找，在堆栈中找到相应的元素，然后把该元素的上面的所有元素下压一个位置，同时把本次访问的块地址抽出来，从最上面压入栈顶。而该元素下面的所有元素则保持不动。
 - 如果Cache访问不命中，则把本次访问的块地址从最上面压入栈顶，堆栈中所有原来的元素都下移一个位置。如果Cache中该组已经没有空闲块，就要替换一个块。这时从栈底被挤出去的块地址就是需要被替换的块的块地址。

- 堆栈法所需要的硬件
 - 需要为每一组都设置一个项数与相联度相同的小堆栈，每一项的位数为 $\log_2 n$ 位。
- 硬件堆栈所需的功能
 - 相联比较
 - 能全部下移、部分下移和从中间取出一项的功能
- 速度较低，成本较高（只适用于相联度较小的LRU算法）

➤ 比较对法

- 基本思路

让各块两两组合，构成比较对。每一个比较对用一个触发器的状态来表示它所相关的两个块最近一次被访问的远近次序，再经过门电路就可找到LRU块。

例如：假设有A、B、C三个块，可以组成3对：AB、AC、BC。每一对中块的访问次序分别用“对触发器” T_{AB} 、 T_{AC} 、 T_{BC} 表示。

- T_{AB} 为“1”，表示A比B更近被访问过；
- T_{AB} 为“0”，表示B比A更近被访问过。
- T_{AC} 、 T_{BC} 也是按这样的规则定义。

显然，当 $T_{AC}=1$ 且 $T_{BC}=1$ 时，C就是最久没有被访问过了。

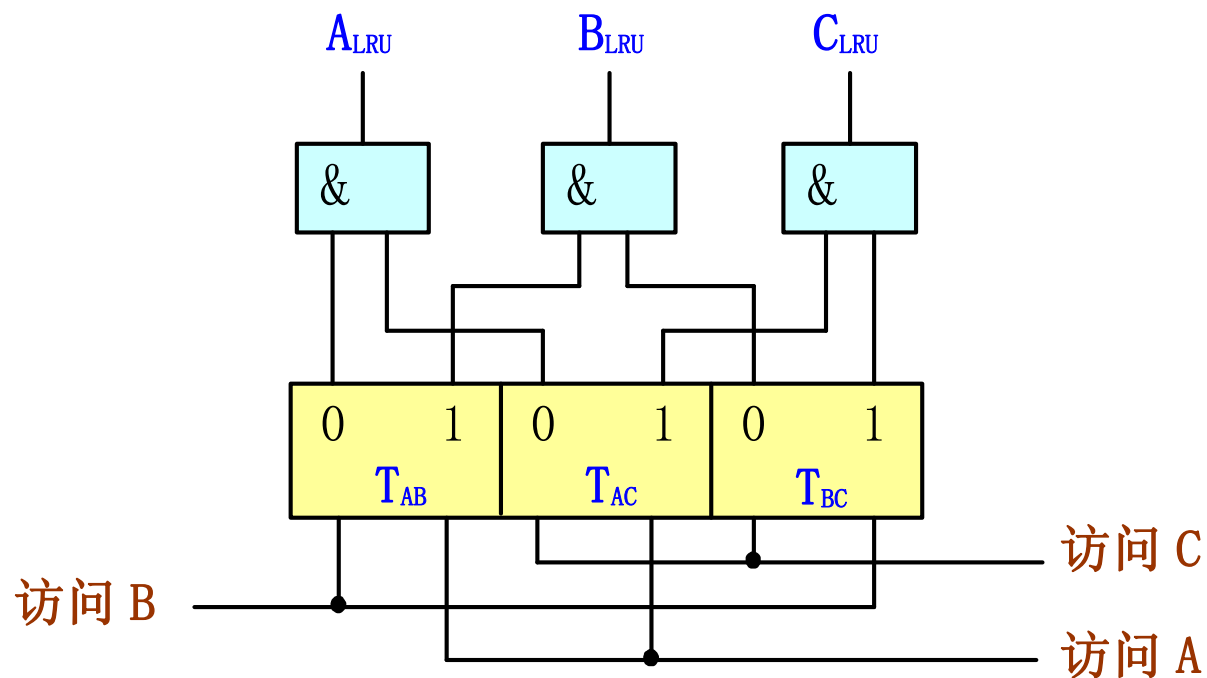
（A比C更近被访问过、且B比C也是更近被访问过）

即： $C_{LRU} = T_{AC} \cdot T_{BC}$

同理可得：

$$B_{LRU} = T_{AB} \cdot \bar{T}_{BC} \quad A_{LRU} = \bar{T}_{AB} \cdot \bar{T}_{AC}$$

用触发器和与门实现上述逻辑的电路：



用比较对法实现LRU算法

- 比较对法所需的硬件量

- 与门

- 与门

有多少个块，就要有多少个与门；每个与门的输入端要连接所有与之相关的触发器。

- 与门

对于一个具有 P 块的组中的任何一个块来说，由于它可以跟除了它自己以外的所有其他的块两两组合，所以与该块相关的比较对触发器个数为 $P-1$ ，因而其相应的与门的输入端数是 $P-1$ 。

- 触发器

- 触发器

所需要的触发器的个数与两两组合的比较对的数目相同。

比较对触发器个数、与门的个数、与门的输入端数与块数P的关系

组内块数	3	4	8	16	64	256	...	P
触发器个数	3	6	28	120	2016	32640	...	$\frac{P(P-1)}{2}$
与门个数	3	4	8	16	64	256	...	P
与门输入端个数	2	3	7	15	63	255	...	P-1

- 块数少时，所需要的硬件较少，
- 随着组内块数 P 的增加，所需的触发器的个数会以平方的关系迅速增加，门的输入端数也线性增加。

（硬件实现的成本很高）

- 当组内块数较多时，可以用多级状态位技术减少所需的硬件量。

例如：在IBM 3033中

组内块数为16，可分成群、对、行3级。

先分成4群，每群两对，每对两行。

选LRU群需6个触发器；

每群中选LRU对需要一个触法器，4个群共需要4个触发器；

每行中选LRU块需要一个触发器，8个行共需要8个触发器。
所需的触发器总个数为：

$$6（选群）+4（选对）+8（选行）=18（个）$$

以牺牲速度为代价的。

7.2.6 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：
$$9\% / (100\% + 26\% + 9\%) \approx 7\%$$
- “写” 在访问Cache操作中所占的比例：
$$9\% / (26\% + 9\%) \approx 25\%$$

1. “写”操作必须在确认是命中后才可进行
2. “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”）

1. 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。

2. 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

1. “写”操作时的调块

➤ 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。

2. 写策略与调块

➤ 写回法 —— 按写分配

➤ 写直达法 —— 不按写分配

7.2.7 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

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

1. 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

$$\begin{aligned} \text{CPU时间} &= IC \times \left(CPI_{\text{execution}} + \frac{\text{访存次数}}{\text{指令数}} \times \text{不命中率} \times \text{不命中开销} \right) \times \text{时钟周期时间} \\ &= IC \times (CPI_{\text{execution}} + \text{每条指令的平均访存次数} \times \text{不命中率} \times \text{不命中开销}) \times \text{时钟周期时间} \end{aligned}$$

例7.1 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

$$\begin{aligned}\text{CPU时间}_{\text{有cache}} &= IC \times (CPI_{\text{execution}} + \text{每条指令的平均访存次数} \\ &\quad \times \text{不命中率} \times \text{不命中开销}) \times \text{时钟周期时间} \\ &= IC \times (2.0 + 1.33 \times 2\% \times 50) \times \text{时钟周期时间} \\ &= IC \times 3.33 \times \text{时钟周期时间}\end{aligned}$$

考虑Cache的不命中后，性能为：

$$\begin{aligned}\text{CPU时间}_{\text{有cache}} &= IC \times (2.0 + 1.33 \times 2\% \times 50) \times \text{时钟周期时间} \\ &= IC \times 3.33 \times \text{时钟周期时间}\end{aligned}$$

实际CPI : 3.33

$$3.33 / 2.0 = 1.67 (\text{倍})$$

CPU时间也增加为原来的1.67倍。

但若不采用Cache, 则：

$$CPI = 2.0 + 50 \times 1.33 = 68.5$$

1. 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：

- $CPI_{\text{execution}}$ 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7.2 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：

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

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

两路组相联Cache的平均访存时间比较低。

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

因此：

$$\begin{aligned}\text{CPU时间}_{1\text{路}} &= IC \times (2.0 \times 2 + (1.3 \times 0.014 \times 70)) \\ &= 5.27 \times IC\end{aligned}$$

$$\begin{aligned}\text{CPU时间}_{2\text{路}} &= IC \times (2.0 \times 2 \times 1.10 + (1.3 \times 0.010 \times 70)) \\ &= 5.31 \times IC\end{aligned}$$

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

直接映象Cache的平均性能好一些。

7.2.8 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17种Cache优化技术
 - 8种用于降低不命中率
 - 5种用于减少不命中开销
 - 4种用于减少命中时间

7.3 降低Cache不命中率

7.3.1 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。

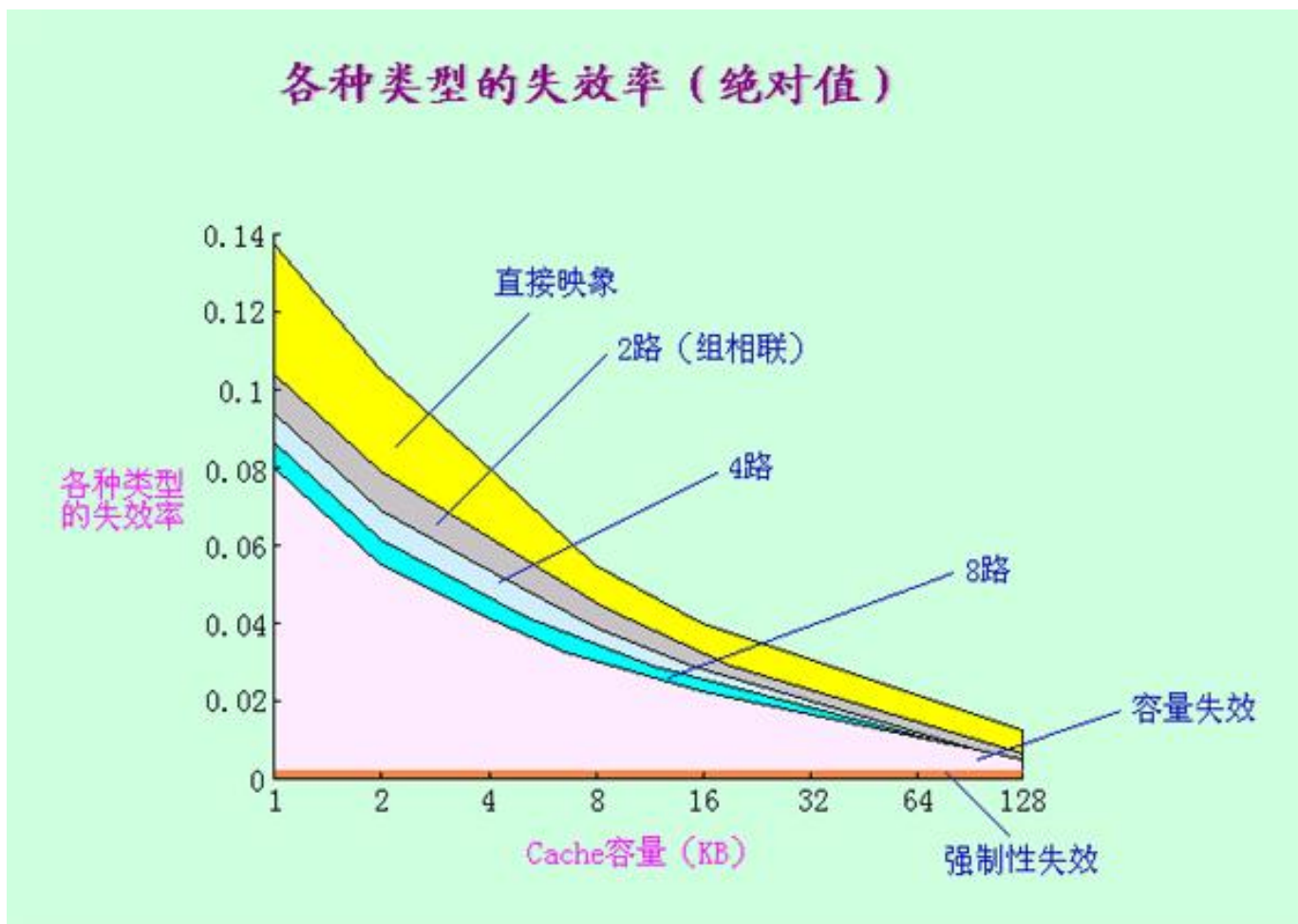
➤ **冲突不命中** (Conflict miss)

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

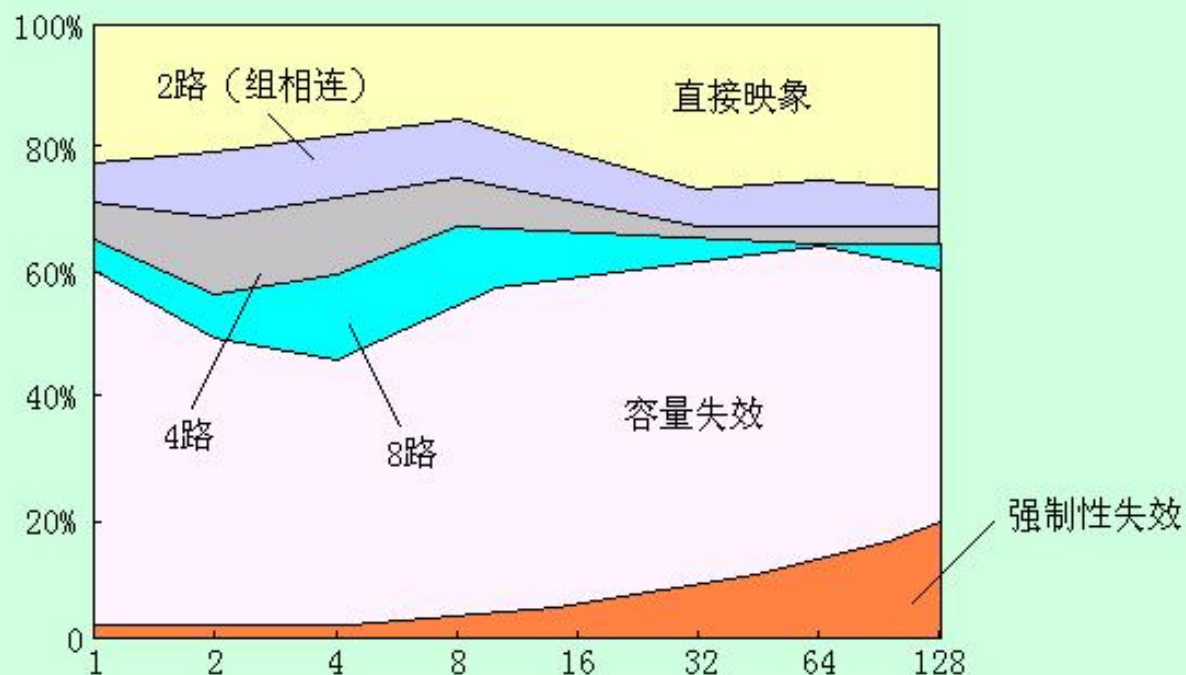
(碰撞不命中，干扰不命中)

1. **三种不命中所占的比例**

- 图示I (绝对值)
- 图示II (相对值)



各种类型的失效率（相对值）



➤ 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。

➤ 减少三种不命中的方法

- 强制性不命中：增加块大小，预取

（本身很少）

- 容量不命中：增加容量

（抖动现象）

- 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

7.3.2 增加Cache块大小

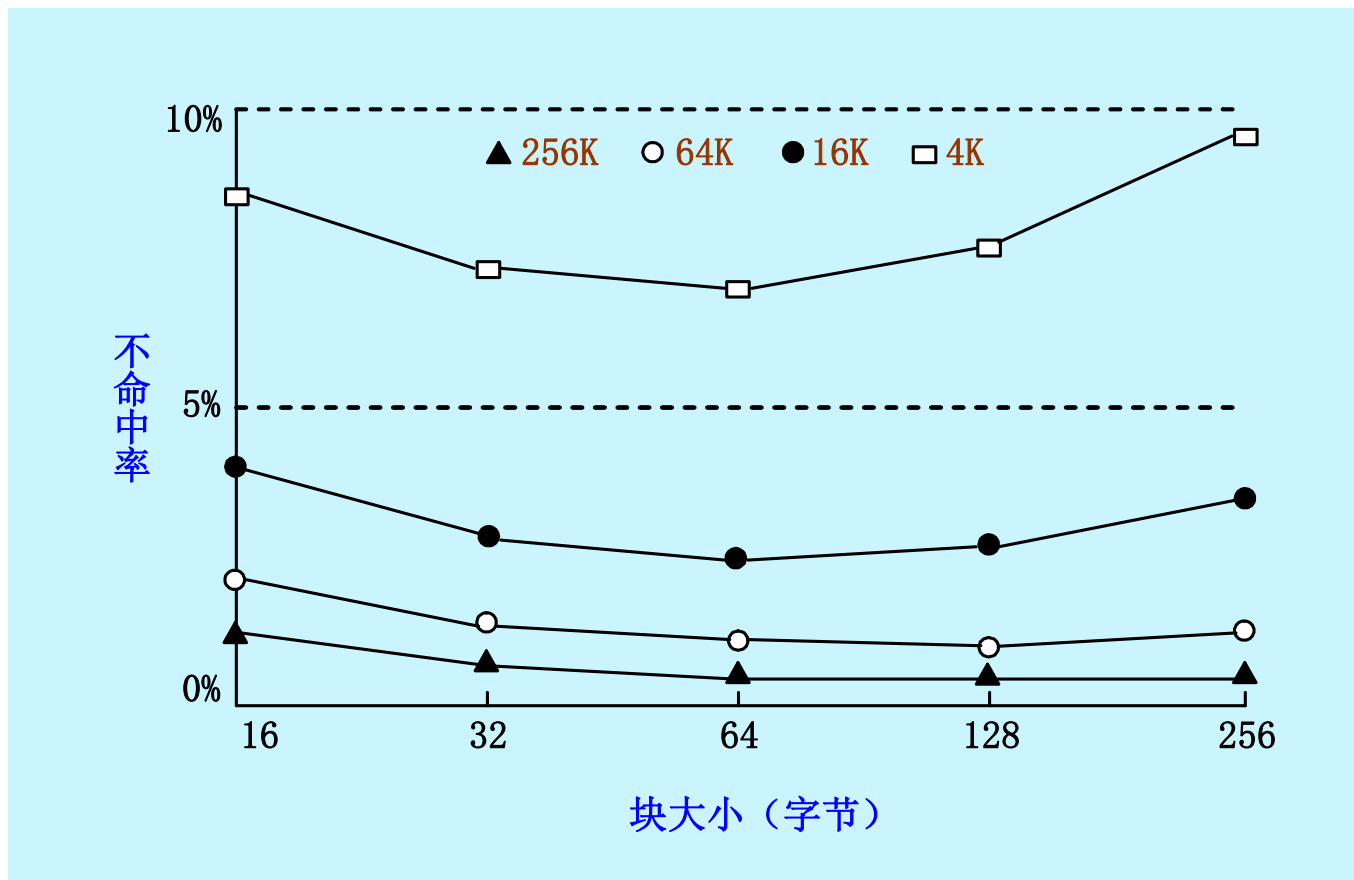
1. 不命中率与块大小的关系

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

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



不命中率随块大小变化的曲线

➤ 各种块大小情况下Cache的不命中率

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%

7.3.3 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

7.3.4 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。

1. 提高相联度是以增加命中时间为代价。

7.3.5 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

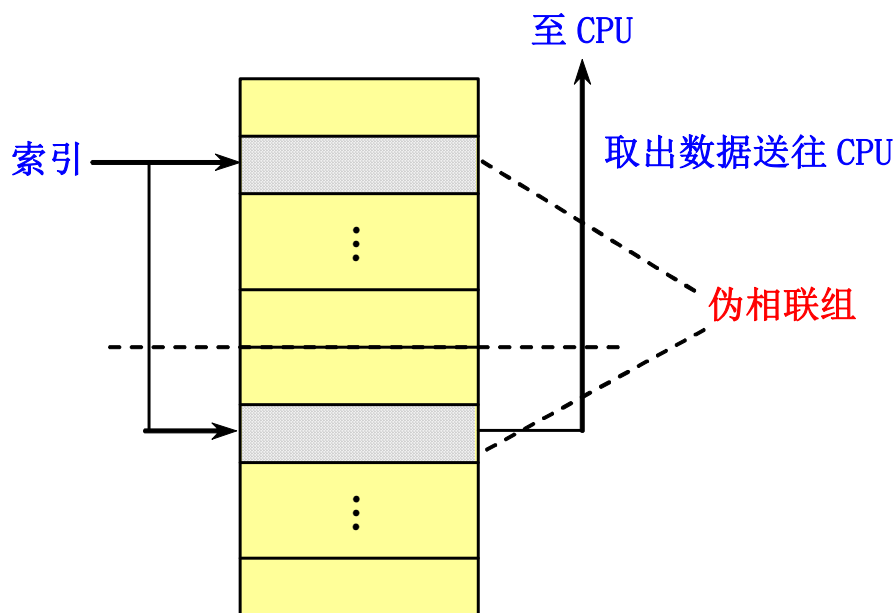
	优 点	缺 点
直接映象	命中时间小	不命中率高
组相联	不命中率低	命中时间大

2. 伪相联Cache的优点

- 命中时间小
- 不命中率低

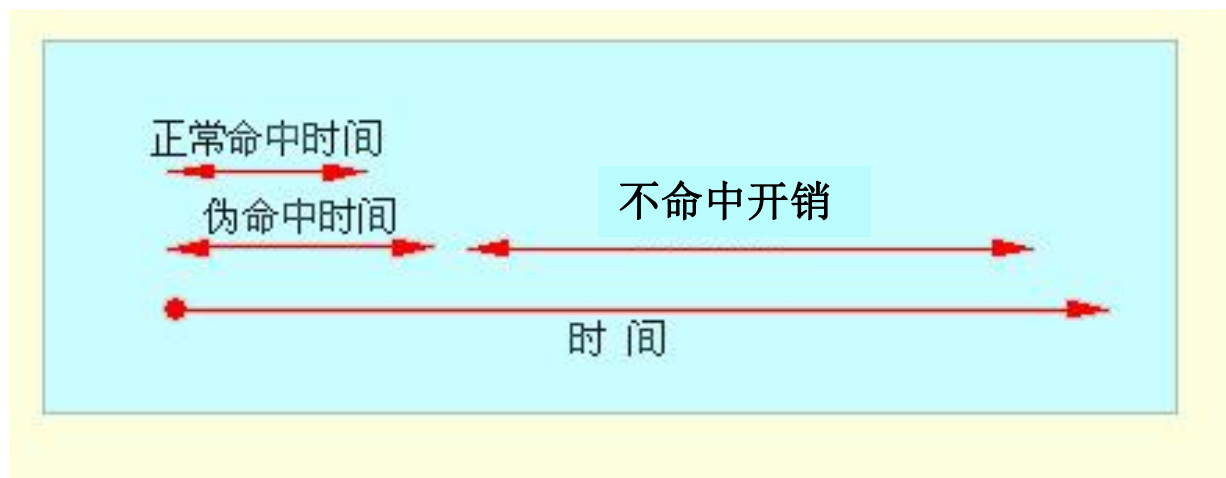
1. 基本思想及工作原理 (动画演示)

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。



1. 快速命中与慢速命中

要保证绝大多数命中都是快速命中。



1. 缺点:

多种命中时间

7.3.6 硬件预取

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. 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**（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%的不命中
 - 4个块的指令流缓冲器： 捕获50%
 - 16个块的指令流缓冲器： 捕获72%

- 数据预取（4KB, 直接映象Cache）
 - 1个数据流缓冲器：捕获25%的不命中
 - 还可以采用多个数据流缓冲器
- Palacharla和Kessler的研究结果
 - 流缓冲器：既能预取指令又能预取数据
 - 对于两个64KB四路组相联Cache来说：
 - 8个流缓冲器能捕获50%~70%的不命中
- 预取应利用存储器的空闲带宽，不能影响对正常不命中的处理，否则可能会降低性能。

7.3.7 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**。

1. **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）

2. **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。

3. **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

1. 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。

7.3.8 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。
 - McFarling研究了如何使用配置文件（profile）来进行这种优化。
- 把基本块对齐，使得程序的入口点与Cache块的

起始位置对齐，就可以减少顺序代码执行时所发生的Cache不命中的可能性。

（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。

➤ 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. 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 ];
```

1. 分块

把对数组的整行或整列访问改为按块进行。

/* 修改前 */

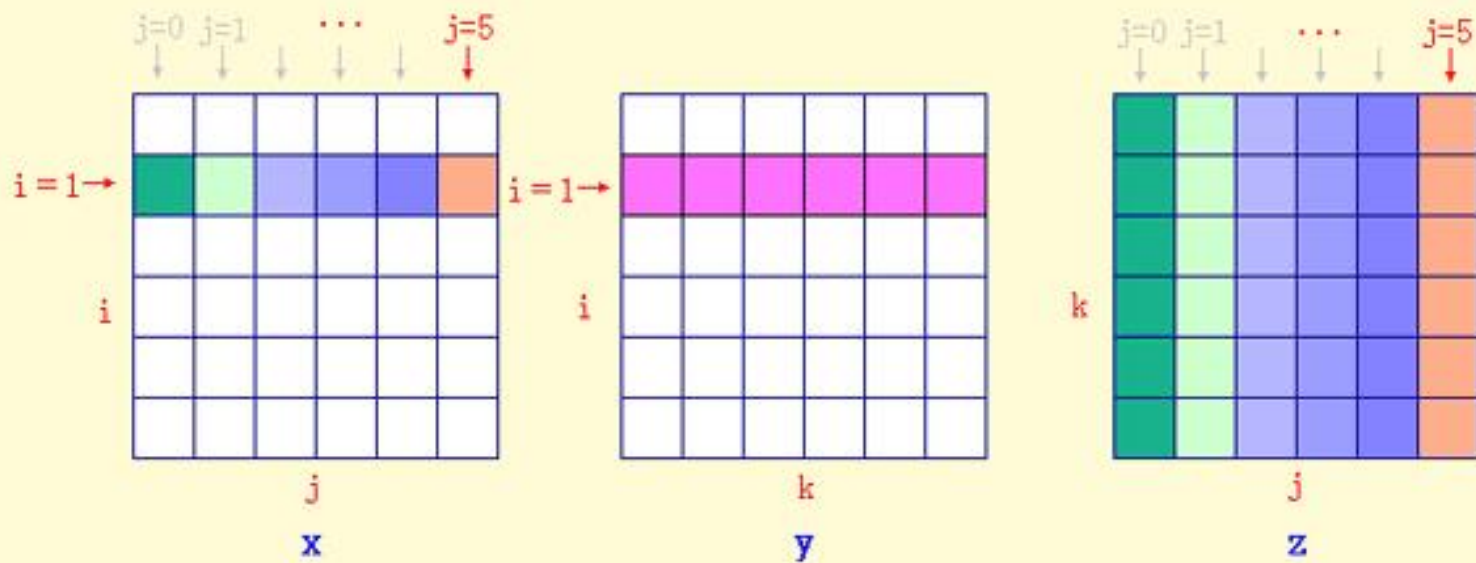
```
for ( i = 0; i < N; i = i+1 )  
for ( j = 0; j < N; j = j+1 ) {  
    r = 0;  
    for ( k = 0; k < N; k = k+1 ) {  
        r = r + y[ i ][ k ] * z[ k ][ j ];  
    }  
    x[ i ][ j ] = r;  
}
```

计算过程 (不命中次数: $2N^3 + N^2$)

数组乘法计算过程

(分块前)

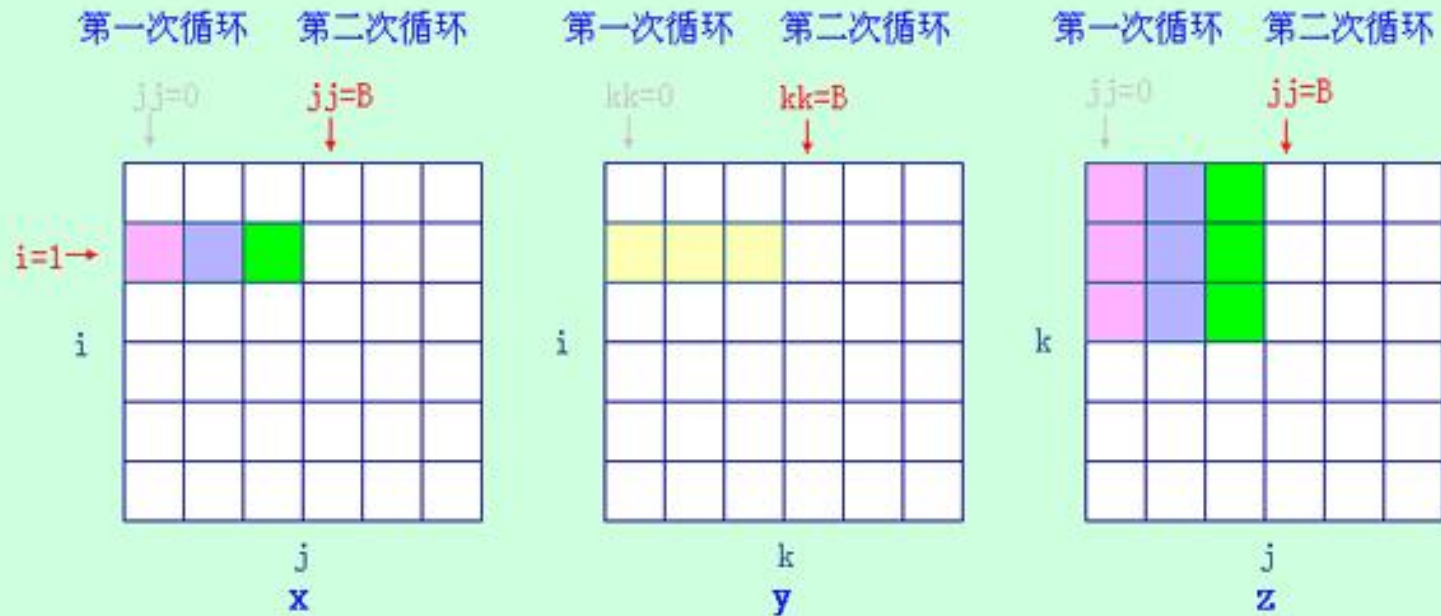
以第2行为例。即 $i = 1$ 的情况。



```
/* 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 (不命中次数: $2N^3 / B + N^2$)

数组乘法计算过程
(分块后)



7.3.9 “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。
 - 工作过程

1. 作用

- 对于减小冲突不命中很有效，特别是对于小容量的直接映象数据Cache，作用尤其明显。

- 例如

项数为4的Victim Cache:

能使4KB Cache的冲突不命中减少20%~90%

7.4 减少Cache不命中开销

7.4.1 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}

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

1. 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

➤ $\text{全局不命中率}_{L2} = \text{不命中率}_{L1} \times \text{不命中率}_{L2}$

评价第二级Cache时，应使用全局不命中率这个指标。
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。

1. 采用两级Cache时，每条指令的平均访存停顿时间：

每条指令的平均访存停顿时间

= 每条指令的平均不命中次数 $_{L1}$ × 命中时间 $_{L2}$ +

每条指令的平均不命中次数 $_{L2}$ × 不命中开销 $_{L2}$

例7.3 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%。

$$\begin{aligned} (2) \text{ 平均访存时间} &= \text{命中时间}_{L1} + \text{不命中率}_{L1} \times (\text{命中时间}_{L2} + \\ &\quad \text{不命中率}_{L2} \times \text{不命中开销}_{L2}) \\ &= 1 + 4\% \times (10 + 50\% \times 100) \\ &= 1 + 4\% \times 60 = 3.4 \text{ 个时钟周期} \end{aligned}$$

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：

$$\text{每条指令的平均停顿时间} = 2.4 \times 1.5 = 3.6 \text{ 个时钟周期}$$

1. 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。
2. 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？

1. 第二级Cache的参数

➤ 容量

第二级Cache的容量一般比第一级的大许多。

大容量意味着第二级Cache可能实际上没有容量不命中，只剩下一些强制性不命中和冲突不命中。

➤ 相联度

第二级Cache可采用较高的相联度或伪相联方法。

例7.4 给出有关第二级Cache的以下数据：

- (1) 对于直接映象，命中时间 t_{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。

7.4.2 让读不命中优先于写

1. Cache中的写缓冲器导致对存储器访问的复杂化

在读不命中时，所读单元的最新值有可能还在写缓冲器中，尚未写入主存。

1. 解决问题的方法(读不命中的处理)

➤ 推迟对读不命中的处理

(缺点：读不命中的开销增加)

➤ 检查写缓冲器中的内容

2. 在写回法Cache中，也可采用写缓冲器。

7.4.3 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，该写操作就算是完成了。

- 如果写缓冲器中已经有了待写入的数据，就要把这次的写入地址与写缓冲器中已有的所有地址进行比较，看是否有匹配的项。如果有地址匹配而

对应的位置又是空闲的，就把这次要写入的数据与该项合并。这就叫**写缓冲合并**。

- 如果写缓冲器满且又没有能进行写合并的项，就必须等待。

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

7.4.4 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

- **尽早重新启动**：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。

1. 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

7.4.5 非阻塞Cache技术

1. **非阻塞Cache**：Cache不命中时仍允许CPU进行其它的命中访问。即允许“不命中下命中”。
2. 进一步提高性能：
 - “多重不命中下命中”
 - “不命中下不命中”
(存储器必须能够处理多个不命中)
1. 可以同时处理的不命中次数越多，所能带来的性能上的提高就越大。(不命中次数越多越好?)

➤ 模拟研究

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

- 测试条件：8K直接映象Cache，块大小为32字节
- 测试程序：SPEC92（14个浮点程序，4个整数程序）
- 结果表明

在重叠不命中个数为1、2和64的情况下

浮点程序的平均比值分别为：76%、51%和39%

整数程序的平均比值则分别为：81%、78%和78%

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

7.5 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。

7.5.1 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：

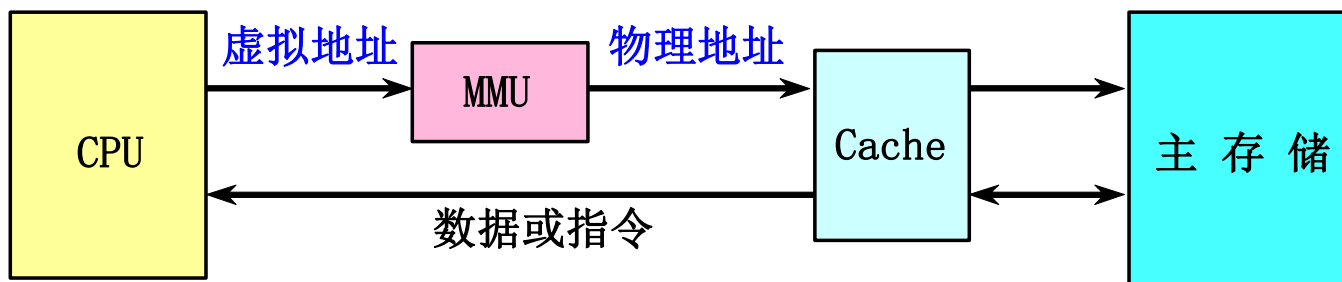
把Cache的标识放在片内，而把Cache的数据存储器放在片外。

7.5.2 虚拟Cache

1. 物理Cache

- 使用物理地址进行访问的传统Cache。
- 标识存储器中存放的是物理地址，进行地址检测也是用物理地址。

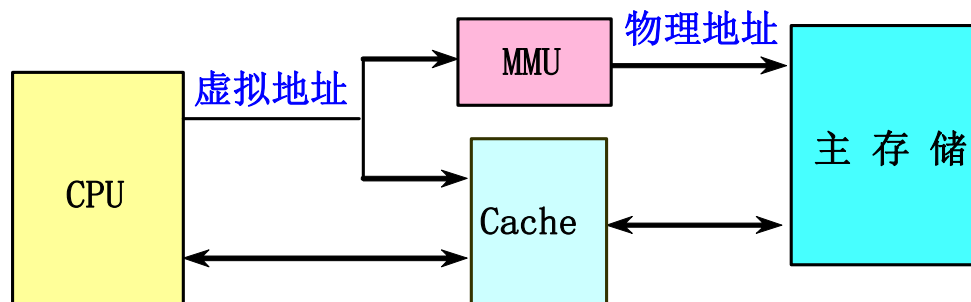
- **缺点：** 地址转换和访问Cache串行进行，访问速度很慢。



物理Cache存储系统

1. 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。



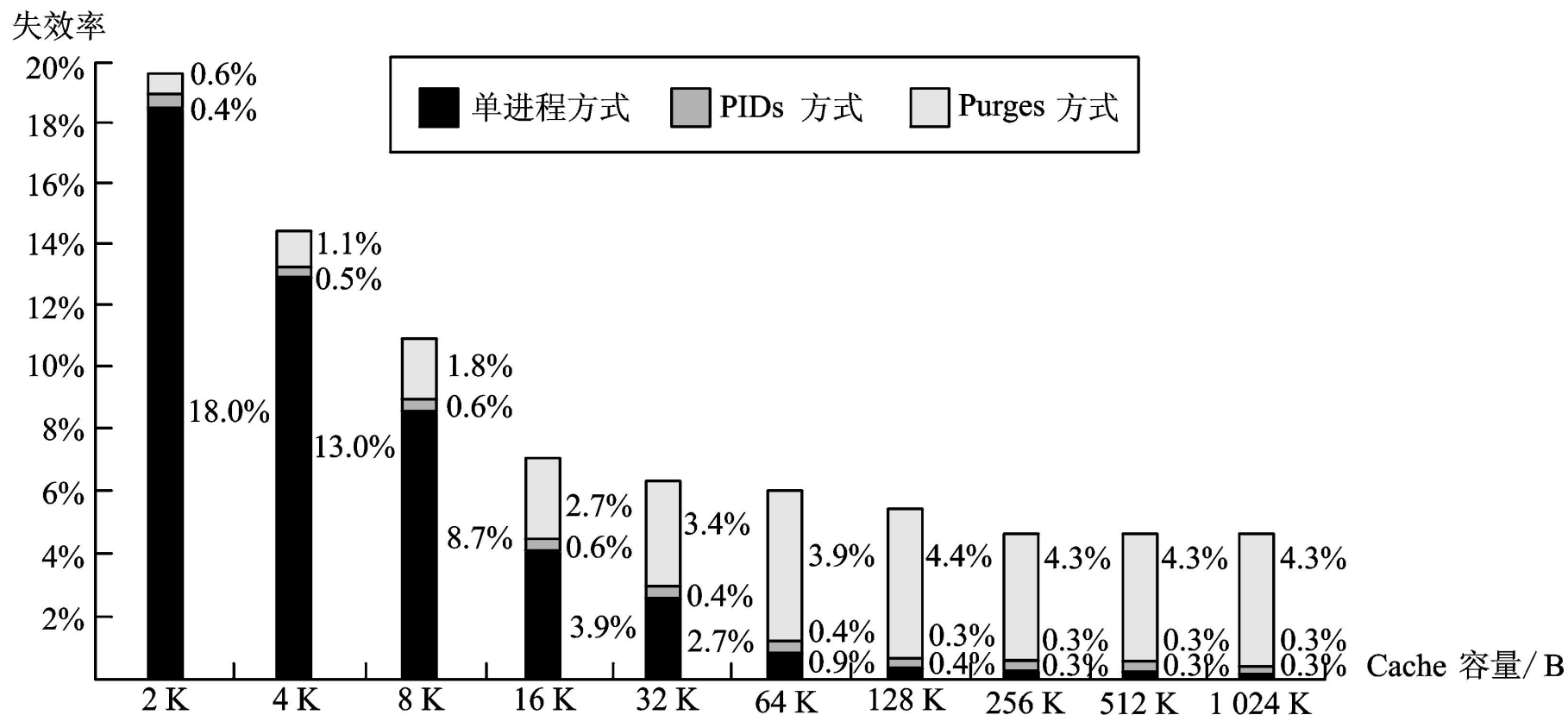
1. 并非都采用虚拟Cache(为什么?)

➤ 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)
- 三种情况下不命中率的比较
 - 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



1. 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 \leq 页大小 \times 相联度

2. 举例：IBM3033的Cache

- 页大小=4KB 相联度=16

31	12	11	0
页地址		页内位移	
地址标识		索引	块内位移

➤ $\text{Cache容量} = 16 \times 4\text{KB} = 64\text{KB}$

1. 另一种方法：硬件散列变换

7.5.3 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 则需要四个时钟周期

7.5.4 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

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。

1. 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。

7.5.5 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

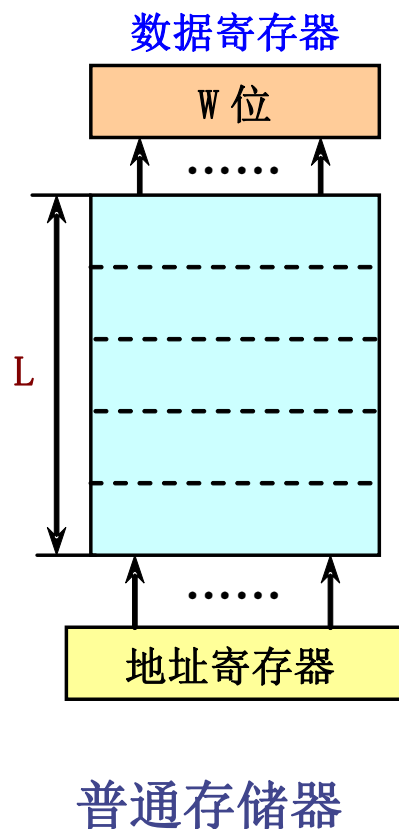
7.6 并行主存系统

- 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
- 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：

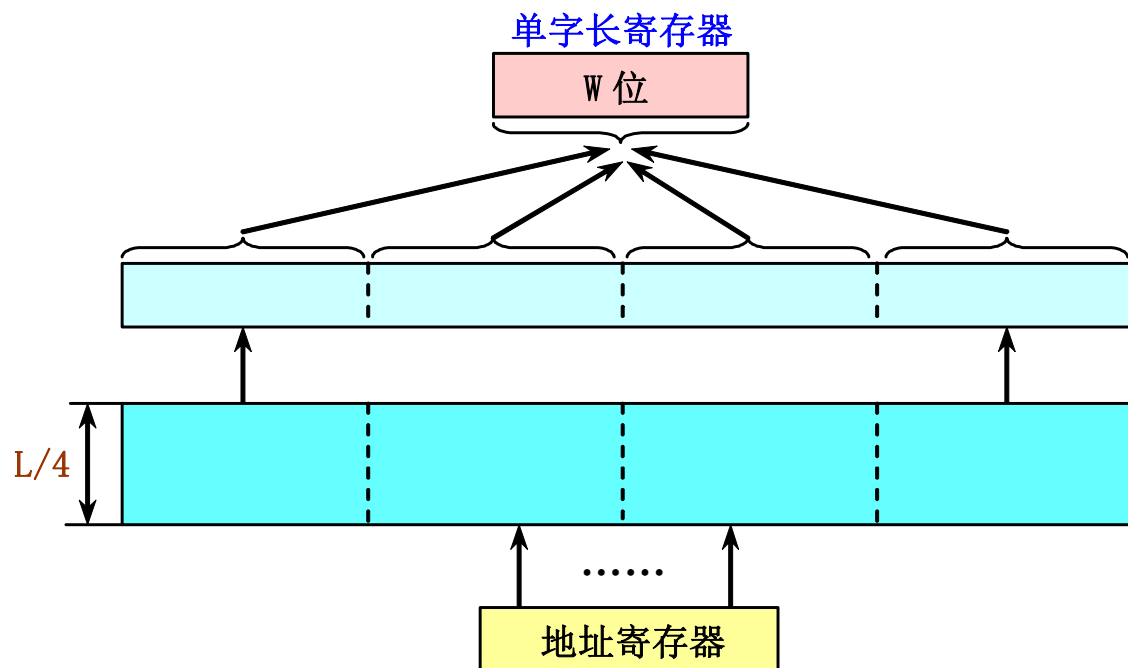
$$B_M = \frac{W}{T_M}$$



- 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

7.6.1 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

1. 优缺点

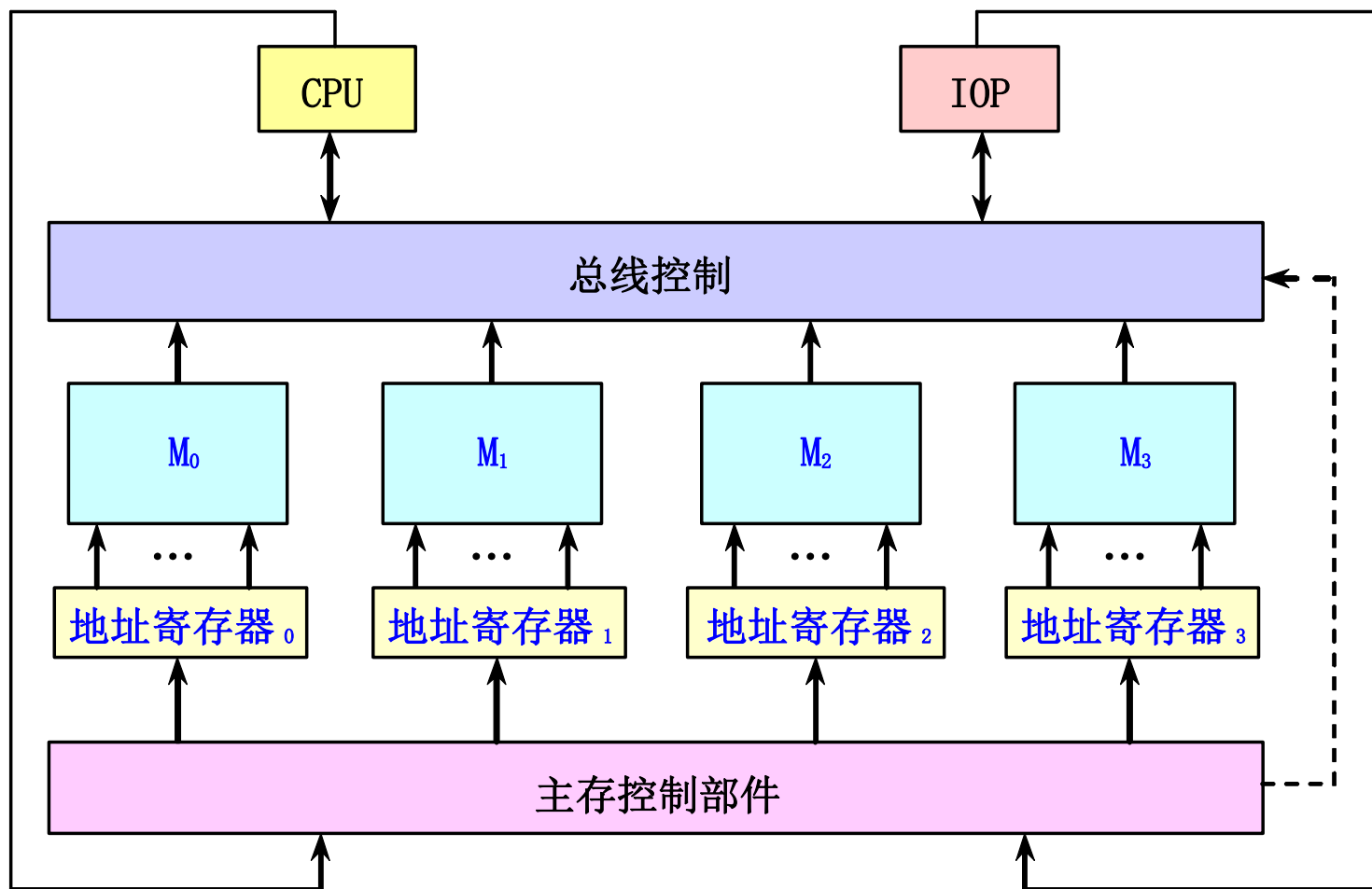
- 优点：实现简单
- 缺点：访存效率不高

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。

7.6.2 多体交叉存储器

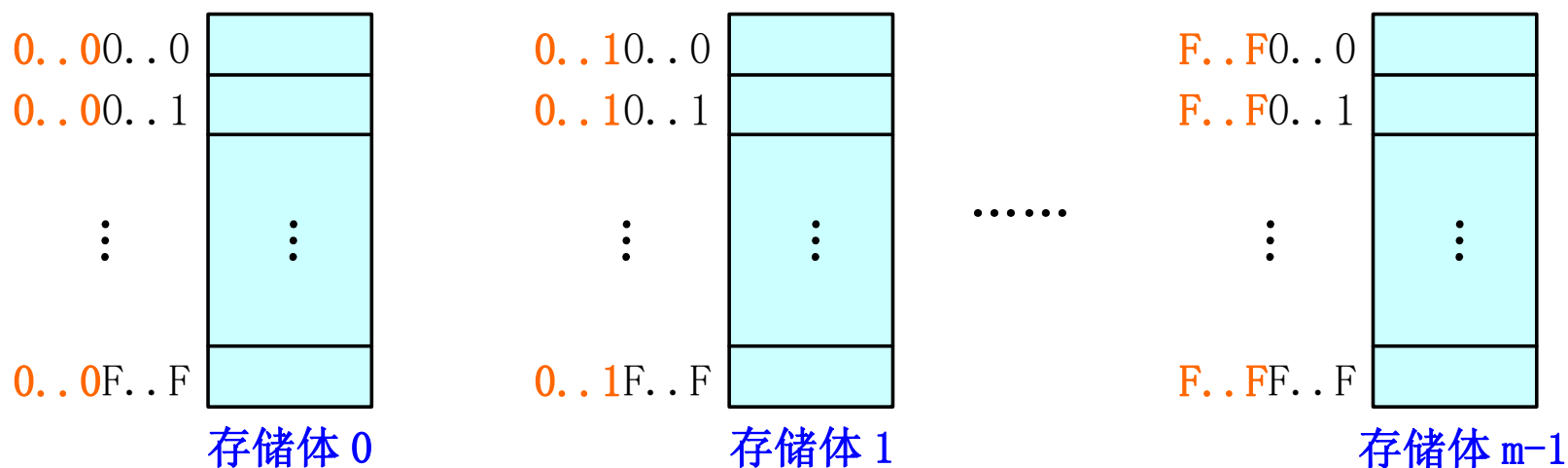
1. **多体交叉存储器**：由多个单字存储体构成，每个体都有自己的地址寄存器以及地址译码和读/写驱动等电路。
2. **问题**：对多体存储器如何进行编址？
 - 存储器是按顺序线性编址的。如何在二维矩阵和线性地址之间建立对应关系？
 - 两种编址方法
 - 高位交叉编址
 - 低位交叉编址（有效地解决访问冲突问题）



多体 (m=4) 交叉存储器

1. 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)



- 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：

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

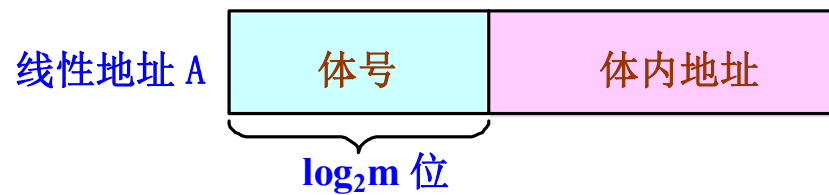
其中： $j = 0, 1, 2, \dots, m-1$

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

- 一个单元的线性地址为*A*，则其体号*j*和体内地址*i*为：

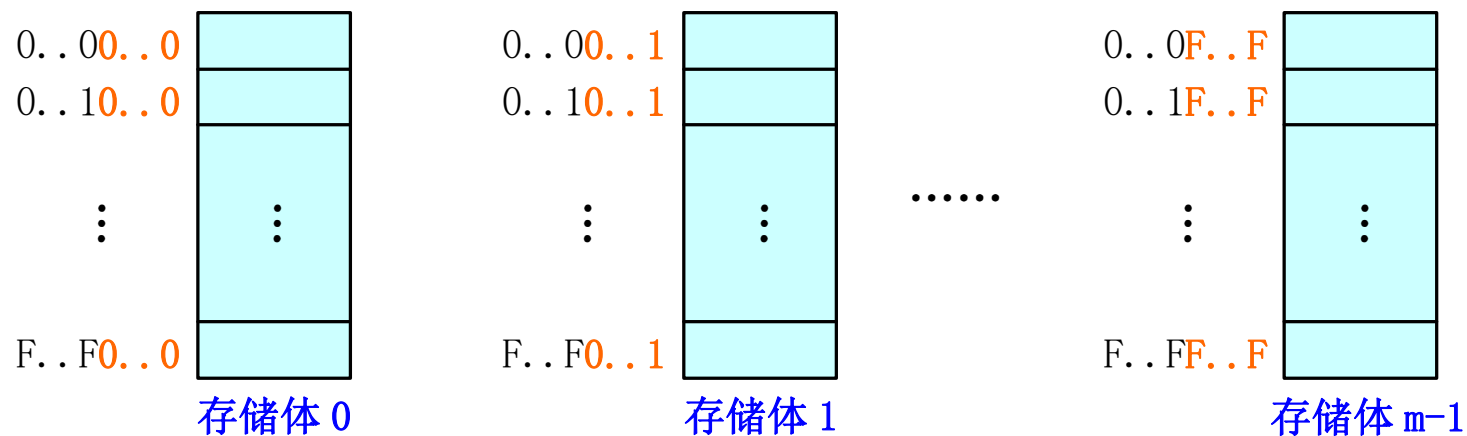
$$j = \left\lfloor \frac{A}{n} \right\rfloor \qquad i = A \bmod n$$

- 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。



1. 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：

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

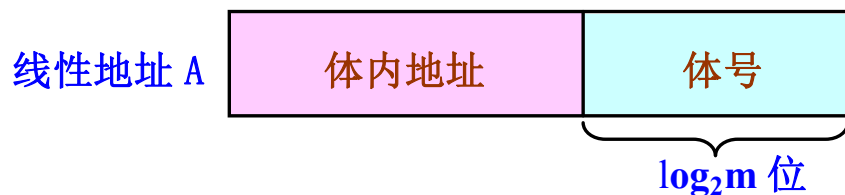
其中： $i = 0, 1, 2, \dots, n-1$

$j = 0, 1, 2, \dots, m-1$

- 一个单元的线性地址为A，则其体号j和体内地址i为：

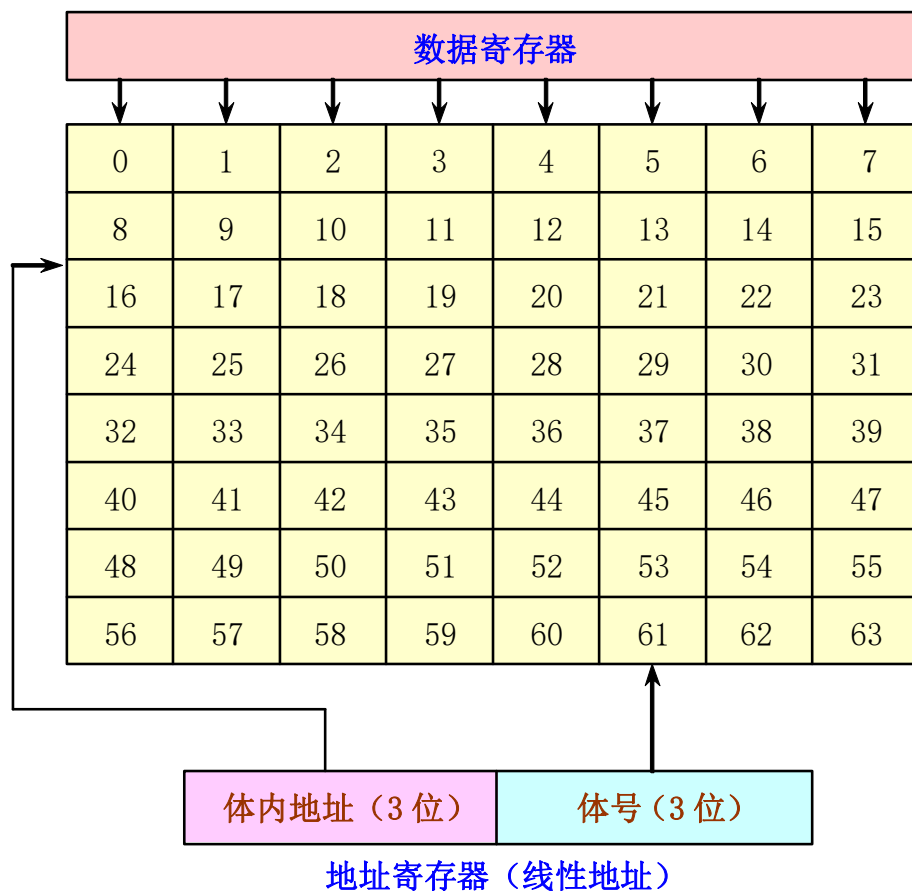
$$i = \left\lfloor \frac{A}{m} \right\rfloor \quad j = A \bmod m$$

- 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。

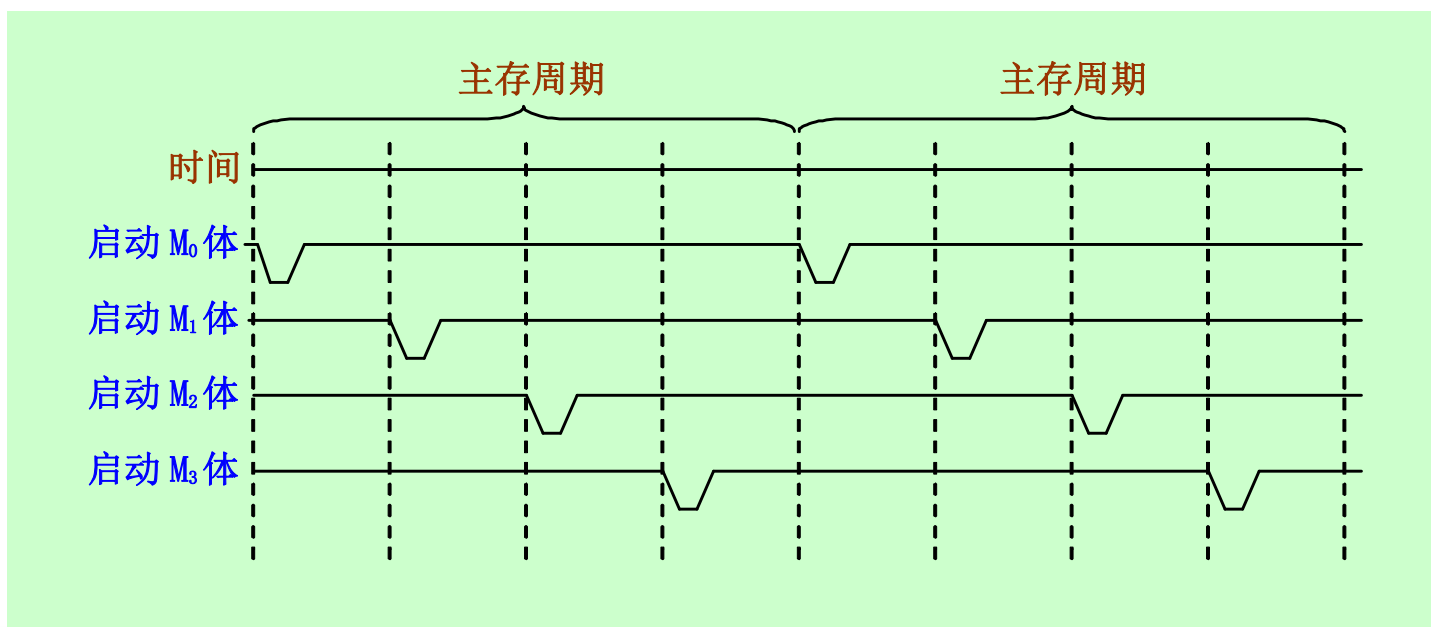


➤ 例：采用低位交叉编址的存储器

由8个存储体构成、总容量为64。格子中的编号为线性地址。



- 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： $t=T_M/m$ 。



- 增加 m 的值就能够提高主存储器的带宽。但是，由于存在访问冲突，实际加速比小于 m 。

1. 通过一个模型分析并行主存系统的实际带宽

- 一个由 m 个独立分体组成的主存系统
- CPU发出的一串地址为 A_1, A_2, \dots, A_q 的访存申请队列
- 存储控制器扫描这个队列，并截取从头起的 A_1, A_2, \dots, A_k 序列作为申请序列。
 - **申请序列**是满足以下条件的最长序列： k 个地址所访问的存储器单元都处在不同的分体中。

- $A_1 \sim A_k$ 不一定是顺序地址，只要它们之间不出现分体冲突。
- k 越接近于 m ，系统的效率就越高。
- 设 $P(k)$ 表示申请序列长度为 k 的概率，用 B 表示 k 的平均值，则

$$B = \sum_{k=1}^m k \cdot P(k)$$

其中： $k=1, 2, \dots, m$

每个主存周期所能访问到的字数的平均值，正比于主存实际带宽。

- $P(k)$ 与具体程序的运行状况密切相关。如果访存申请队列都是指令的话，那么影响最大的是转移概率 λ 。
 - **转移概率 λ** ：给定指令的下条指令地址为非顺序地址的概率。
 - 当 $k=1$ 时，所表示的情况是：第一条就是转移指令且转移成功。
- $P(1) = \lambda = (1-\lambda)^0 \cdot \lambda$
- 当 $k=2$ 时，所表示的情况是：第一条指令没有转移（其概率为 $1-\lambda$ ），第二条是转移指令且转移成功。所以有：

$$P(2) = (1-\lambda)^1 \cdot \lambda$$

- 同理, $P(3) = (1-\lambda)^2 \cdot \lambda$
- 依此类推, $P(k) = (1-\lambda)^{k-1} \cdot \lambda$

其中: $1 \leq k < m$

- 如果前 $m-1$ 条指令均不转移, 则不管第 m 条指令是否转移, k 都等于 m , 因此有:

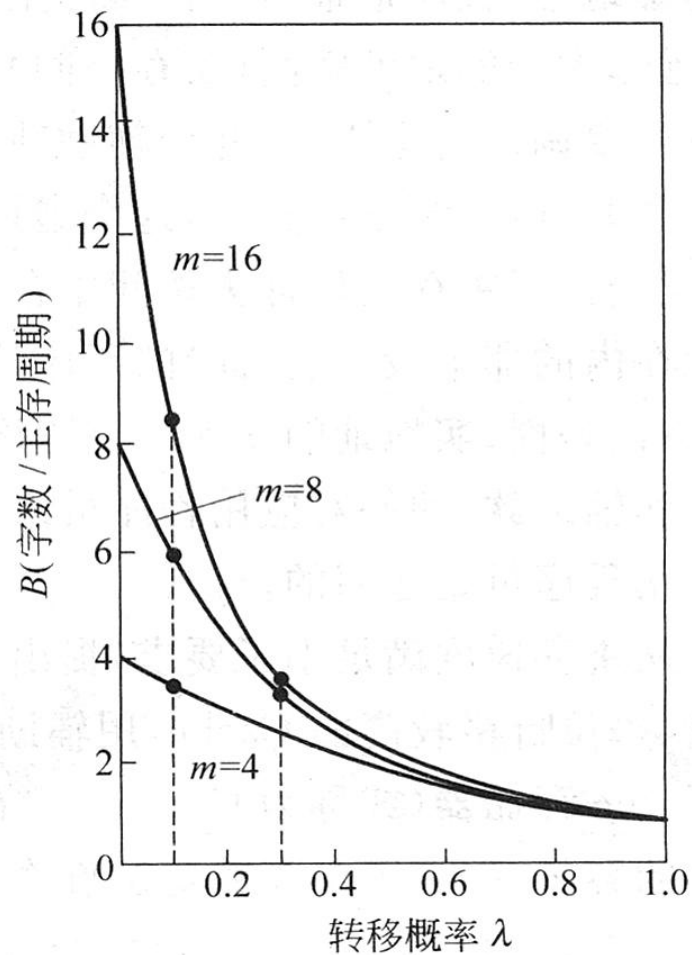
$$P(m) = (1-\lambda)^{m-1}$$

$$B = \sum_{k=1}^m k \cdot P(k) = 1 \cdot \lambda + 2 \cdot (1-\lambda) \cdot \lambda + 3 \cdot (1-\lambda)^2 \cdot \lambda + \cdots + (m-1)(1-\lambda)^{m-2} \cdot \lambda + m(1-\lambda)^{m-1}$$

$$B = \sum_{i=0}^{m-1} (1-\lambda)^i$$

$$B = \frac{1 - (1-\lambda)^m}{\lambda}$$

- m 等于4、8、16时， B 与 λ 的关系曲线



- 对于数据来说，由于其顺序性差， m 值的增大给 B 带来的好处就更差一些。
 - 若机器主要是运行标量运算的程序，一般取 $m \leq 8$ 。
 - 如果是向量处理机，其 m 值可以取大些。
- 单纯靠增大 m 来提高并行主存系统的带宽是有限的，而且性能价格比还会随 m 的增大而下降。

原因：

 - 程序的转移概率不会很低
 - 数据分布的离散性较大

7.6.3 避免存储体冲突

1. **体冲突**：两个请求要访问同一个体。
2. **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：

假如我们有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不命中时暂停。

1. 解决体冲突的方法

➤ 软件方法(编译器)

- ❑ 循环交换优化
- ❑ 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A 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

7.7 虚拟存储器

7.7.1 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。

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

7.7.2 快速地址转换技术

1. 地址变换缓冲器TLB

- 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

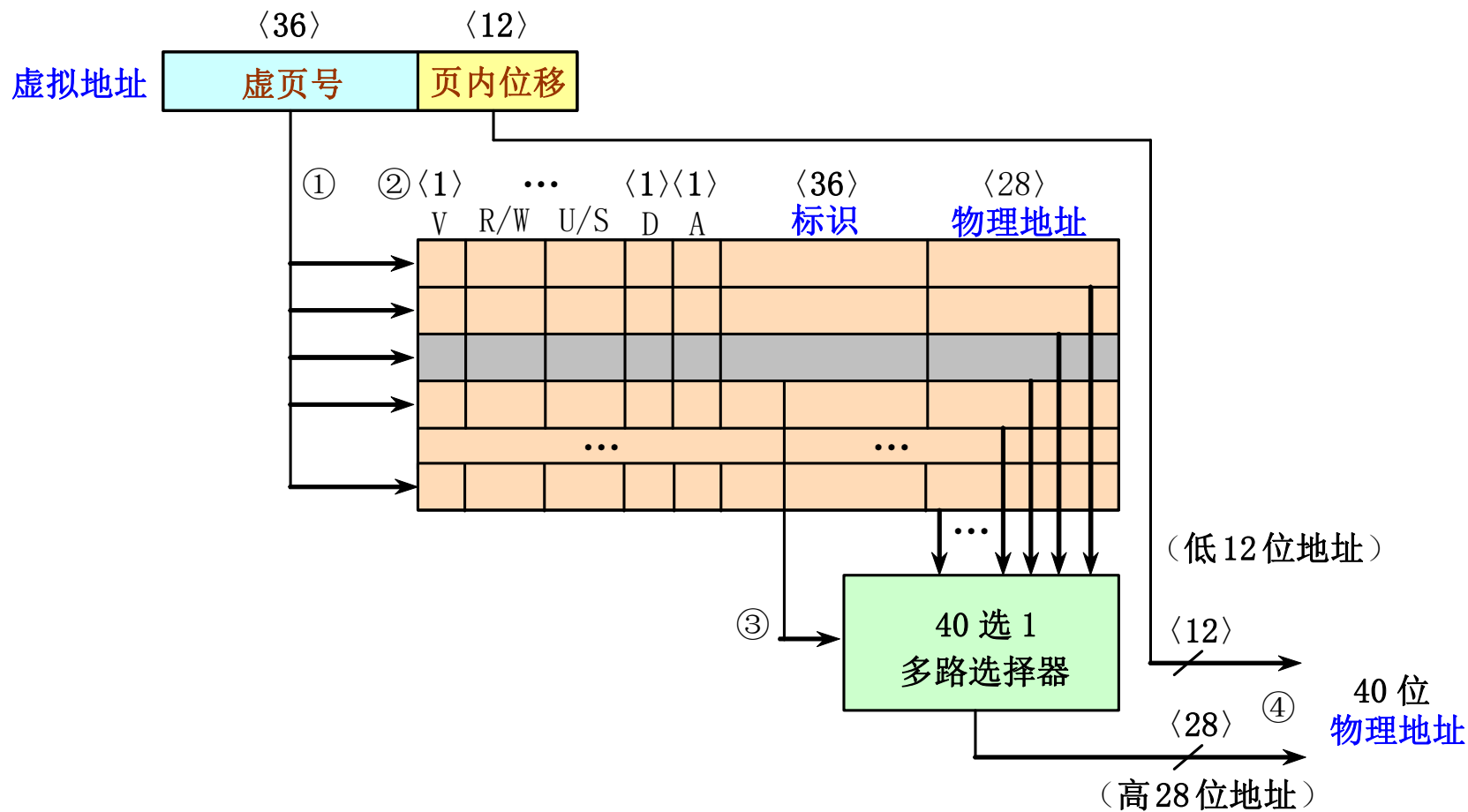
- 标识中存放的是虚地址的一部分。
- 数据部分中存放的则是物理页帧号、有效位、存储保护信息、使用位、修改位等。

1. 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

2. 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。



7.7.3 页式虚拟存储器实例：

64位Opteron的存储管理

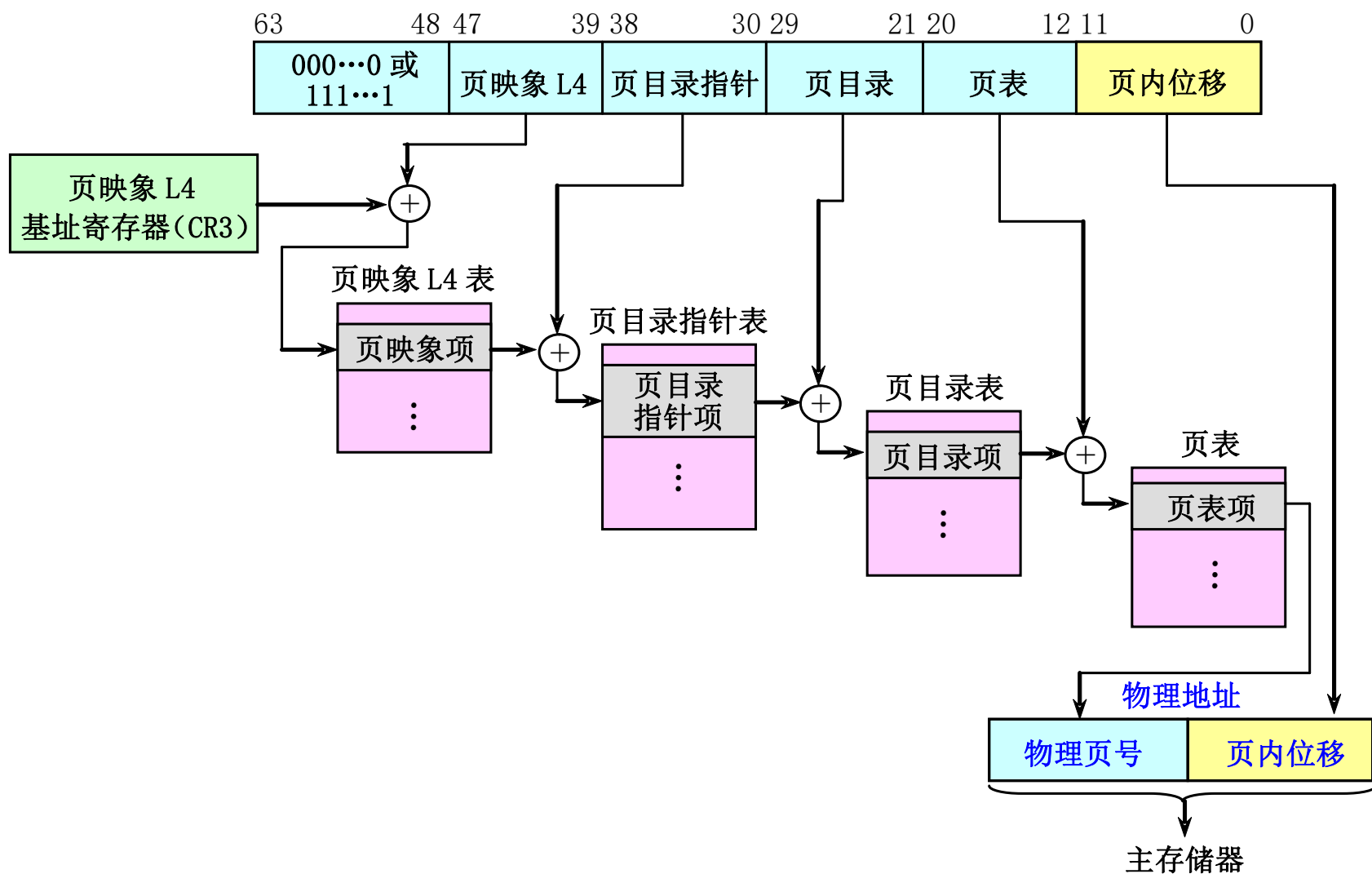
1. Opteron的页面大小：4KB，2MB和4MB。
2. AMD64系统结构

虚拟地址：64位 物理地址：52位

- 进行虚→实地址转换时，是把64位的虚拟地址映射到52位的物理地址。
- 要求：64位虚拟地址中的高16位是由低48位进行符号位扩展而来的

规范格式

1. 采用多级分层页表结构来映射地址空间，以便使页表大小合适。
 - 分级的级数取决于虚拟地址空间的大小
 - Opteron的48位虚拟地址的4级转换
 - 每个分级页表的偏移量分别来自4个9位的字段
2. Opteron的每一级页表都采用64位的项
其中：
 - 前12位留给将来使用
 - 随后的52位是物理页号



➤ 最后的12位包括保护和使用信息。

不同级的页表中有所不同，但大都包含以下基本字段：

- ❑ **存在位**：说明该页面在存储器中。
- ❑ **读/写位**：说明该页面是只读还是可读写。
- ❑ **用户/管理位**：说明用户是否能访问此页或只能由上面的3个特权级所访问。
- ❑ **修改位**：说明该页面已被修改过。
- ❑ **访问位**：说明自上次该位被清0后到现在，该页面是否被读或写过。
- ❑ **页面大小**：说明最后一级页面是4KB还是4MB；如果是4MB，则Opteron仅使用三级页表而非四级。

- **非执行位**：在有些页面中用来阻止代码的执行。
- **页级Cache使能**：说明该页面能否进入Cache。
- **页级写直达**：说明该页是允许对数据Cache进行写回还是写直达。

1. Opteron通常在TLB不命中时要遍历所有四级页表，故有3个位置可以进行保护限制的检查。
 - 仅遵从底层的PTE，而在其他级上只需确认有效位是有效的即可。
2. 在保护方面，如何避免用户进行非法的地址转换？
 - 页表本身已经被保护，用户程序无法对它们进行写操作。

- 操作系统通过控制页表项来控制哪些物理地址可以被访问，哪些不能访问。
- 多个进程共享存储器是通过使各自的地址空间中的一个页表项指向同一个物理页面来实现的。

1. Opteron使用4个TLB以减少地址转换时间

- 两个用于访问指令，两个用于访问数据。

2. 和多级Cache类似，Opteron通过采用两个更大的第二级TLB来减少TLB不命中。

- 一个用于访问指令
- 另一个用于访问数据

➤ Opteron中第一级和第二级指令、数据TLB的参数

参数	描述
块大小	1个 PTE (8字节)
L1命中时间	1个时钟周期
L2命中时间	7个时钟周期
L1 TLB大小	指令和数据TLB都是40个PTE，其中32个用于4KB大小的页面，8个用于2MB或4MB页面。
L2 TLB大小	指令和数据TLB都是512个PTE，用于4KB页面
块选择	LRU
L1映象规则	全相联
L2映象规则	4路组相联

7.8 实例：AMD Opteron的存储器层次结构

➤ 一个乱序执行处理器

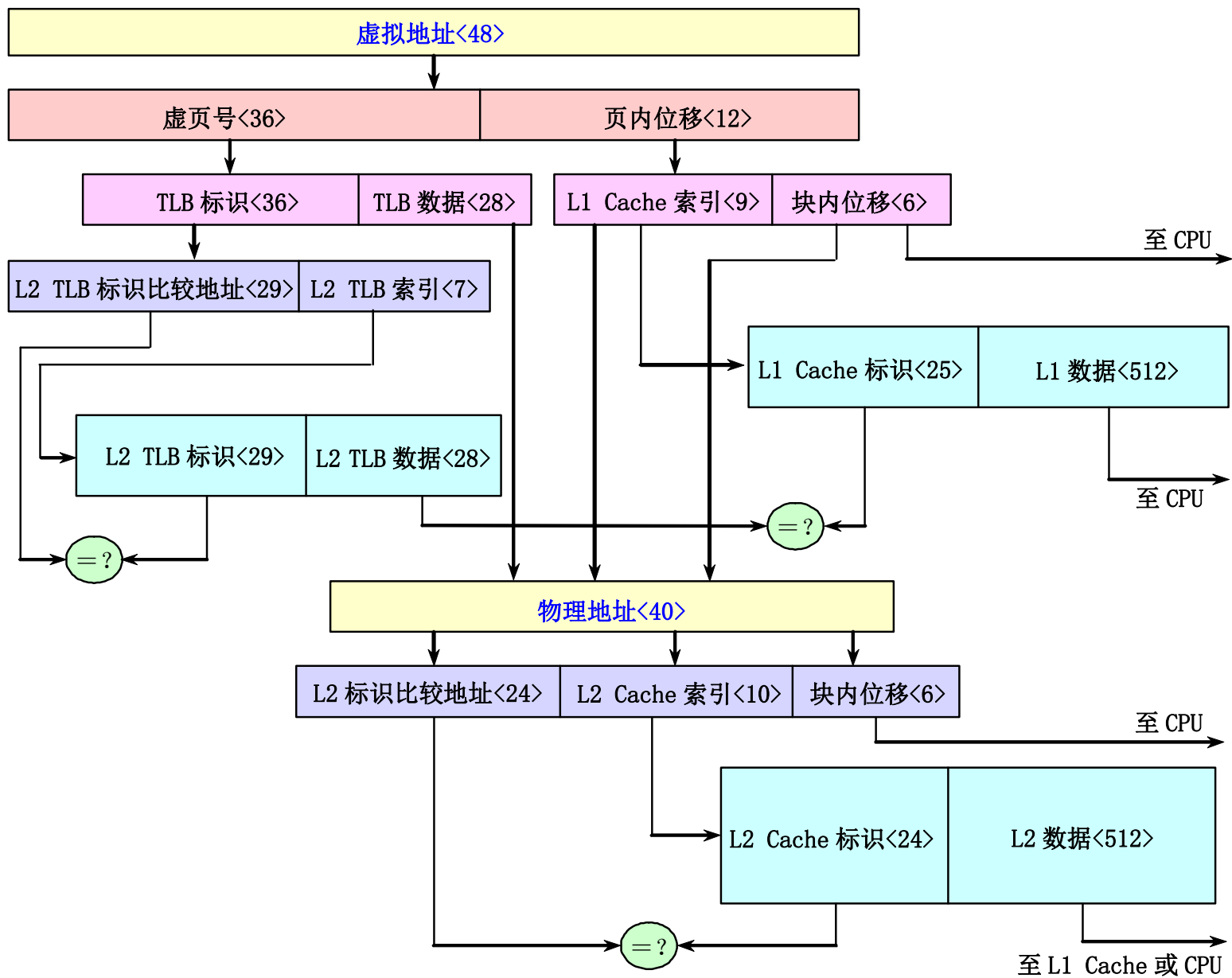
- 每个时钟周期最多可以取出3条80x86指令，并将之转换成类RISC操作，然后以每个时钟周期3个操作的速率流出。

➤ 有11个并行的执行部件

➤ 在2006年，其12级定点流水线使得该处理器的最高时钟频率达到了2.8GHz。

➤ 虚地址：48位 物理地址：40位

1. 通过两级TLB实现的从虚拟地址到物理地址的转换以及对两级数据Cache的访问情况



1. AMD Opteron存储器层次结构图

