体系结构 第五章

by xjc

5.1 存储系统的基本知识

5.1.1 存储系统的层次结构

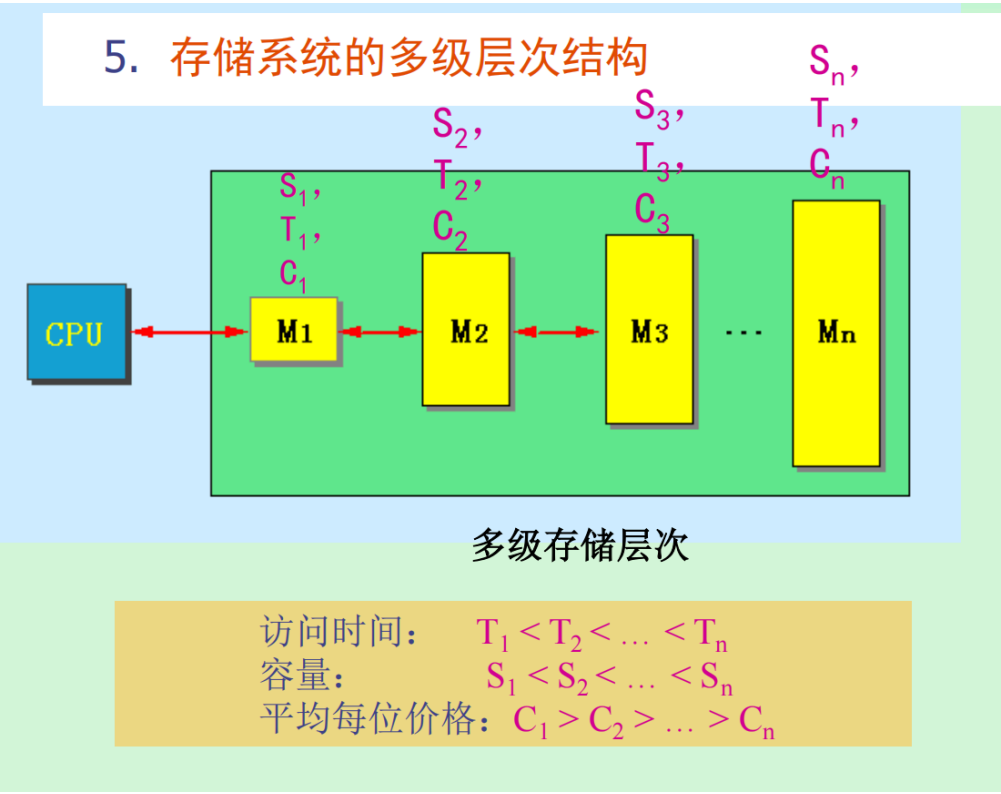
（1）计算机系统结构设计的关键问题

问题：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?
三种指标：容量大、速度快、价格低
变化趋势

- 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

（2）解决方法——多级存储结构

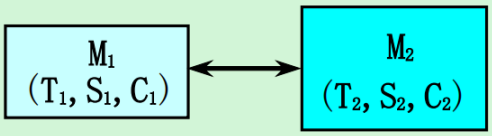
方法：采用多种存储器技术，构成多级存储层次结构
多级层次结构：见图，*非常重要*
目标：整个存储系统要达到的目标：从CPU来看，该存储系统的速度接近于M1的，而容量和每位价格都接近于Mn的



5.1.2 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. 每位价格C

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

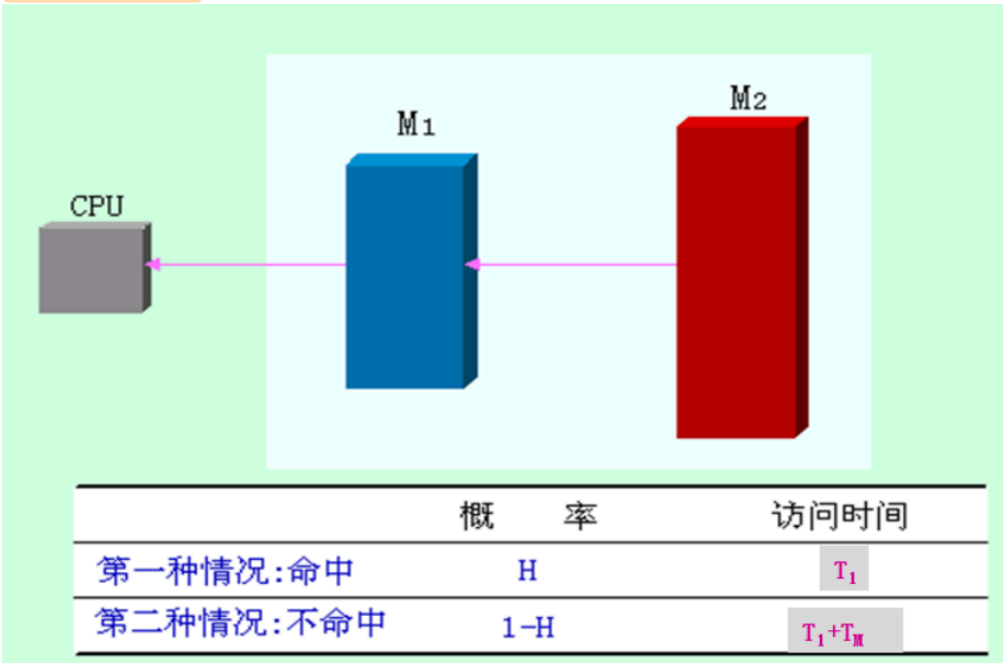
2. 命中率

定义：CPU访问存储系统时，在M_1中找到所需信息的概率（一定要注意是哪一级！）

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

其中N_1为命中M_1的次数，N_2为命中M_2的次数

3. 平均访问时间



	概　率	访问时间
第一种情况:命中	H	T_1
第二种情况:不命中	1-H	T_1+T_M

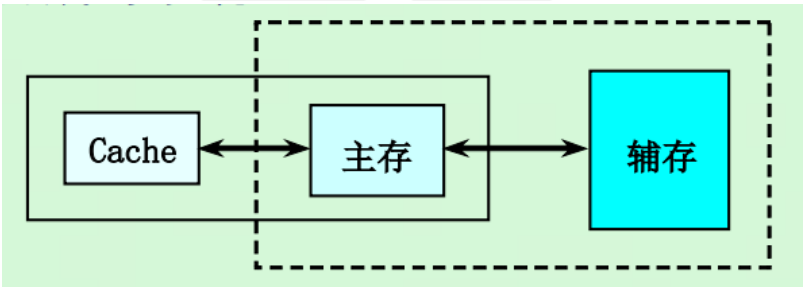
$$T_A = \frac{[N_1 * T_1 + N_2 * (T_1 + T_M)]}{(N_1 + N_2)} = HT_1 + (1 - H)(T_1 + T_M)$$

其中，T_M是不命中开销，其他参数的含义见命中率部分

5.1.3 三级存储系统

（1）分级内容

1. Cache（高速缓冲存储器）
2. 主存储器
3. 磁盘存储器（辅存）
   可以看成是由**Cache-主存**和**主存-赋存**两个层次构成的系统



（2）两种层次

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

5.1.4 存储层次的四个问题

1. 映像规则: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?
2. 查找算法:当所要访问的块在高一层存储器中时，如何*找到该块*?
3. 替换算法:当发生*不命中*时，应*替换*哪一块?
4. 写策略:当进行*写访问*时，应进行哪些操作?

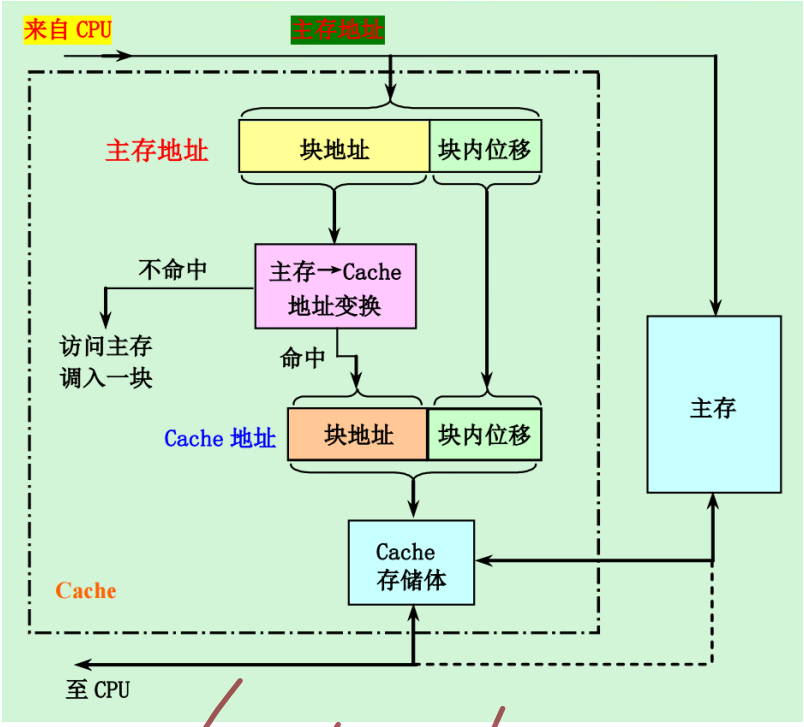
5.2 Cache基本知识

5.2.1 基本结构和原理

存储空间划分：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

Cache的基本工作原理



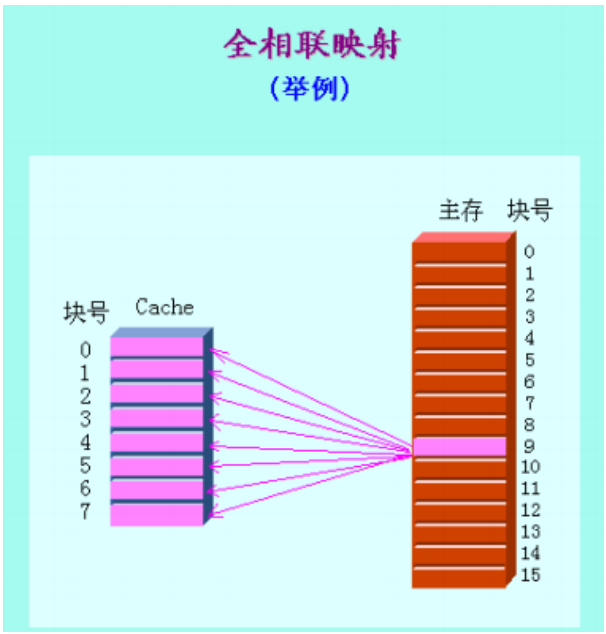
5.2.2 映像规则

(1) 全相联映像

定义：主存中的任一块可以被放置到Cache中的任意一个位置

特点：空间利用率最高，冲突概率最低，实现最复杂

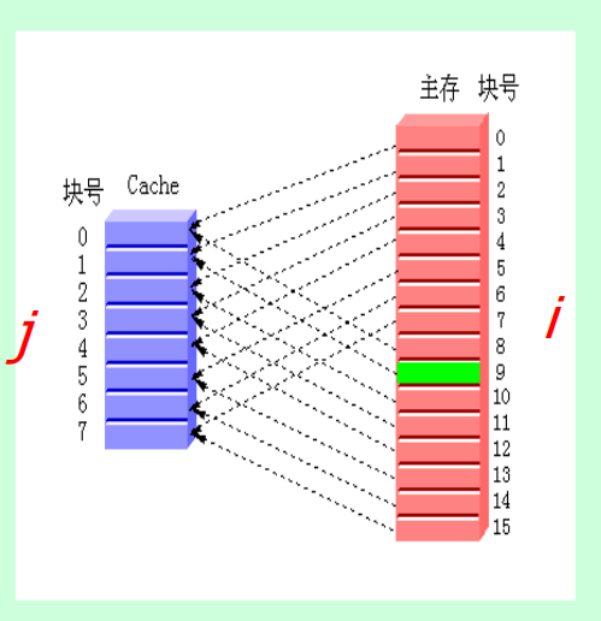
任意主存块→任意Cache块



（2）直接映像

定义：主存中的每一块只能被放置到Cache中唯一的一个位置
特点：空间利用率最低，冲突概率最高，实现最简单
映像方法：很简单，主存第i块，映射到Cache第j块，Cache共M块，则

$$j = i \ mod \ M$$



特殊映射：如果$M = 2^m$，那么j就是i的低m位，低m位为索引

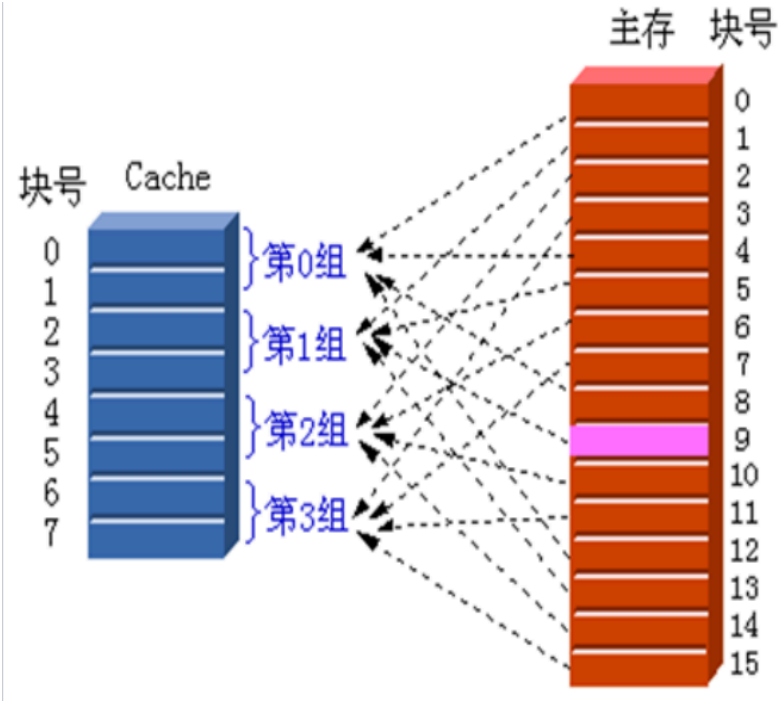
详细目录表

编号	cache块号	主存块号
000	000	001000
001	001	110001
010	010	101010
011	011	100011
100	100	101100
101	101	010101
110	110	001110
111	111	000111

（3）组相连映像

定义：*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*。

- 组相联是直接映像和全相联的一种折中



映射：主存第i块，映射到第k组，$k = i \ mod \ G$，G为Cache的组数
如果$G = 2^g$，k就是i的低g位，低g位通常成为索引
n路组相连：每组中有n个块（n=M/G），n称为相连度，相连度n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

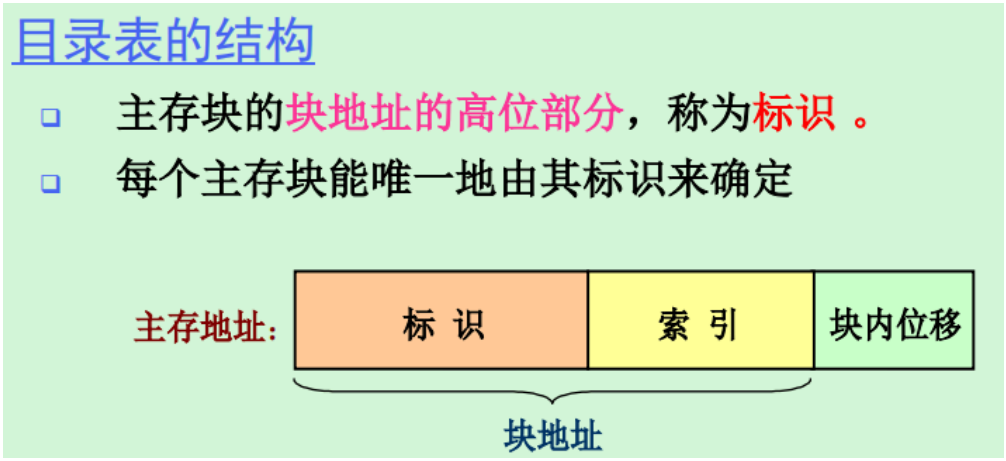
	n（路数）	G（组数）
全相联	M	1
直接映像	1	M
组相联	$1<n<M$	$1<G<M$

（4）部分结论

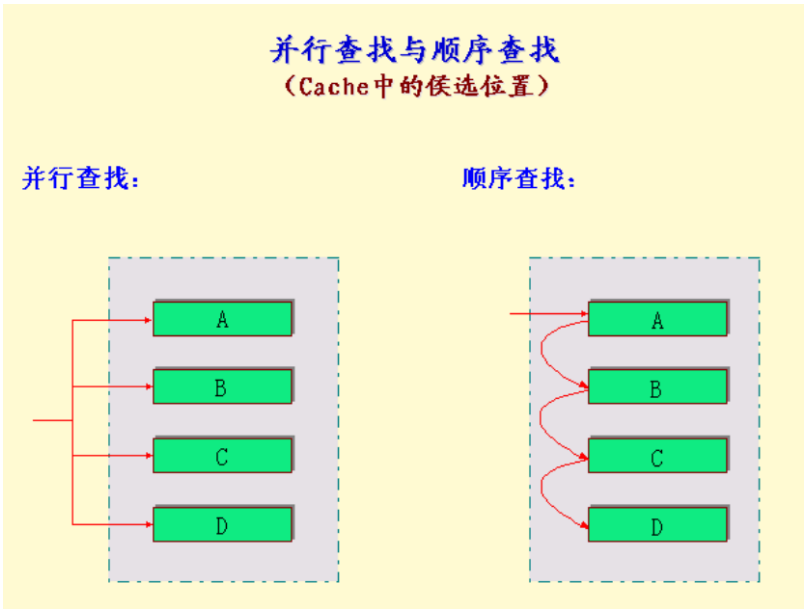
1. 块冲突是指一个主存块要进入已被占用的Cache块位置。
2. 显然，全相联的失效率最低，直接映象的失效率最高。
   虽然从降低失效率的角度来看，n的值越大越好，但在后面我们将看到，增大n值并不一定能使整个计算机系统的性能提高，而且还会使Cache的实现复杂度和代价增大。
3. 因此，绝大多数计算机都采用直接映象、两路组相联或4路组相联。特别是直接映象，采用得最多。

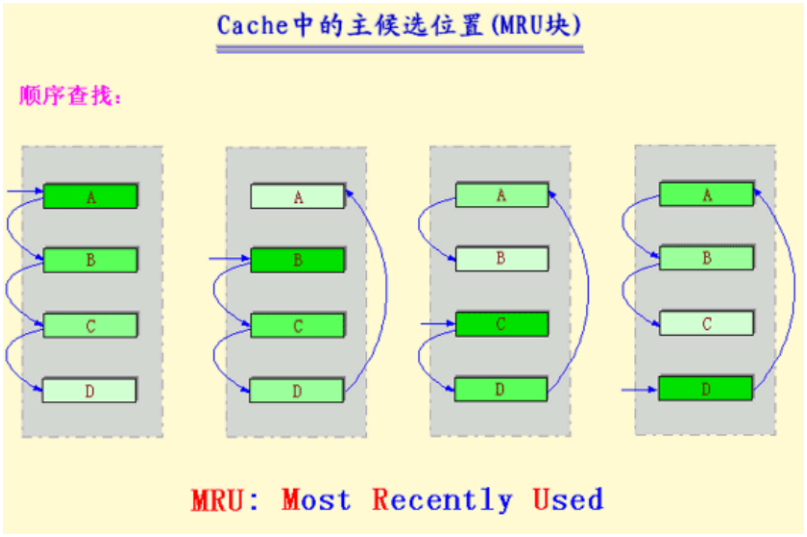
5.2.3 查找算法

查找方式：查找目录表



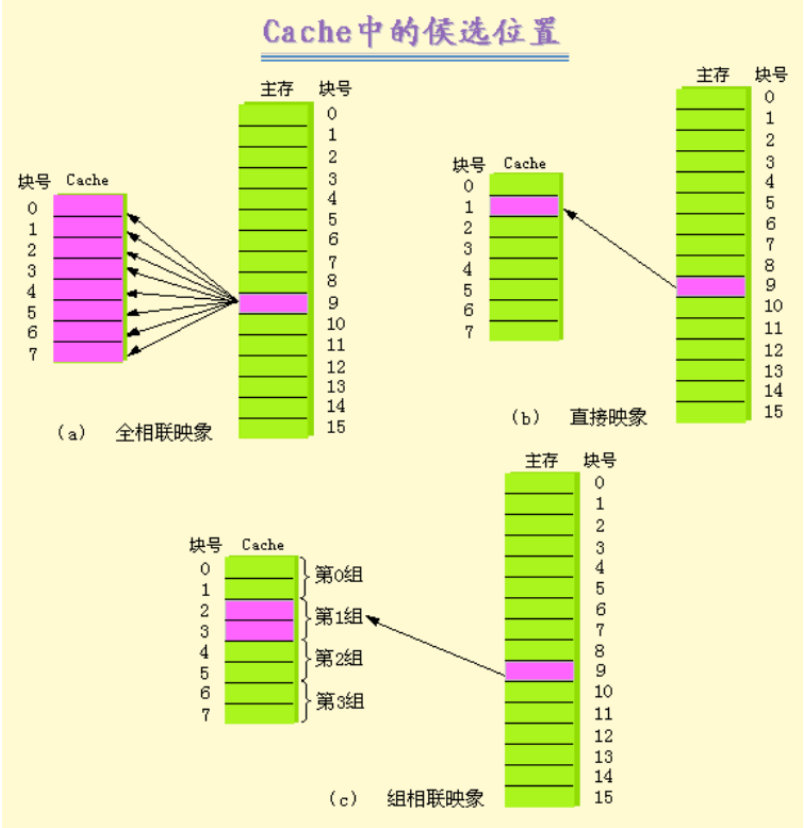
并行查找与顺序查找：提高性能的重要思想：主候选位置（MRU Most Recently used块）（前瞻执行）



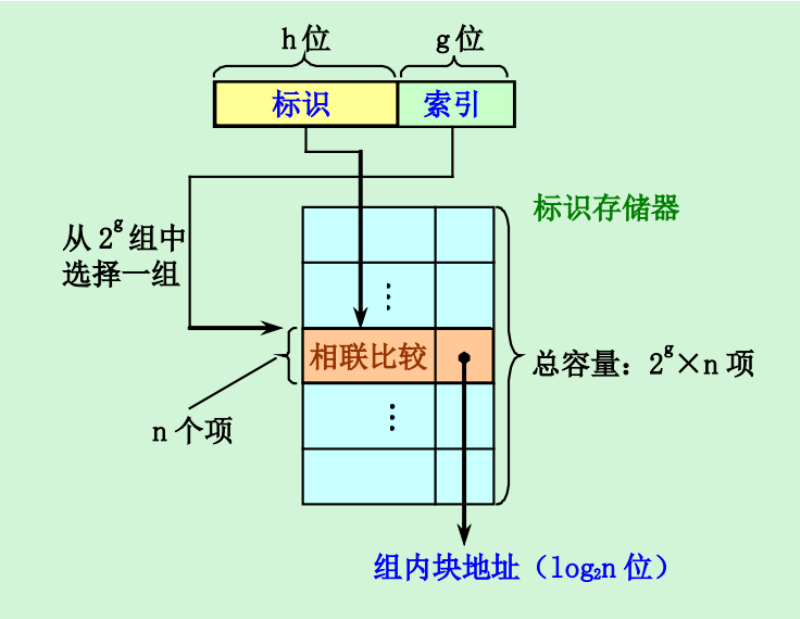


候选位置：根据映象规则不同，一个主存块可能映象到Cache中的一个或多个Cache块位置。为便于讨论，我们把它们称为**候选位置**（简单地来说就是，可以映射到的块）

- 直接映象：Cache的候选位置最少，只有一个；
- 全相联Cache：候选位置最多，为M个
- n路组相联：则介于两者之间，为n个



并行查找的实现方法：

1. 相连存储器
    - 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×（h+log2n）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
    - 

2. 单体多字存储器 + 比较器

- 举例： 4 路组相联并行标识比较
- 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
  - 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

5.2.4 Cache的工作过程（略）

读访问命中，写访问命中

5.2.5 替换算法

（1）替换算法需要解决的问题

当新调入一块，而Cache又已被占满时，替换哪一块？需要解决组相联和全相联cache的替换问题

- 原因
  - 直接映像：只能选1个块
  - 组相联/全相联：有多个cache可选

（2）可以使用的替换方法

1. 随机法
   优点：实现简单

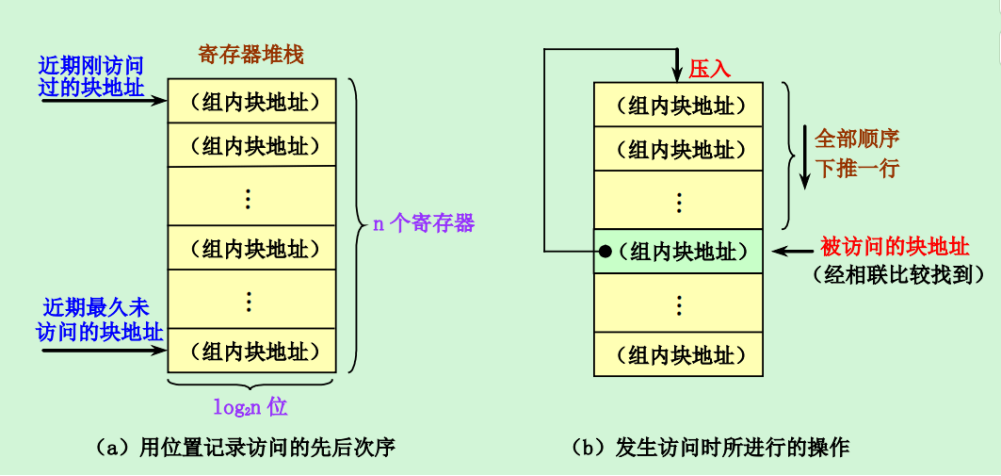
2. 先进先出法（FIFO）

3. 最近最少使用法（LRU法）
   原本的目标：选择近期最少被访问的块作为被替换的块。（实现比较困难）
   实际的实现方法：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

上述方法中，**LRU法和随机法**分别因其不命中率低和实现简单而被广泛采用

LRU的物理实现：
1. **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
- 栈底记录的是该组中最早被访问过的块，当需要替换时，从栈底得到应该被替换的块
- 堆栈中的内容需要动态更新
- 速度低，成本高

-



2. **比较对法**：

- 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
- *触发器电路和个数计算是否会成为考点呢？感觉不会*

-

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：

$$B_{LRU} = T_{AB} \bullet \overline{T}_{BC} \qquad A_{LRU} = \overline{T}_{AB} \bullet \overline{T}_{AC}$$

5.2.6 写策略

（1）概念：

- “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

（2）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

1. 写直达法（存直达法）

   方法：执行“写”操作时，不仅写入Cache，而且也写入下一级存储器（*同时写入cache和下一级存储器*）

   优点：速度快，所使用的存储器带宽较低

   CPU写停顿：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CPU写停顿的优化技术位采用写缓冲器

调块方式： 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

2. **写回法（拷回法）**

   方法： 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”)，*先cache，替换后再写入主存*

   优点： 易于实现，一致性好

   调块方式： 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

5.2.7 Cache性能分析

（1）不命中率

与硬件速度无关

（2）平均访存时间和CPI

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

$$CPI_{\text{实际}} = CPI_{ex} + \text{各指令平均访存次数} \times \text{不命中率} \times \text{不命中开销}$$

（3）程序执行时间

1. **CPU时间基础公式**

$$CPU\text{时间} = (CPU\text{执行周期数} + \text{停顿周期数}) \times \text{时钟周期时间}$$

   其中停顿周期数是需要计算的

2. **停顿周期数的计算**

$$\text{停顿周期数} = (\text{读次数} + \text{写次数}) \times \text{不命中率} \times \text{不命中开销}$$

3. **综合计算公式**

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

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

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

4. 例题：第11次课p4，例5.1和例5.2

5.2.8 改进cache的性能

平均访存时间 = 命中时间 + 不命中率×不命中开销

可以从三个方面改进Cache的性能：

1. *降低不命中率*
2. *减少不命中开销*
3. *减少**Cache**命中时间*

5.3 降低Cache不命中率

5.3.1 三种类型的不命中

（1）三种类型

1. 强制性不命中(Compulsory miss)：从来没用过，第一次无法命中
   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)
2. 容量不命中(Capacity miss)：容量不够导致的
   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。
3. 冲突不命中(Conflict miss)：Cache中块的映射太多导致的
   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)

（2）三种不命中所占的比例

1. 相联度越高，冲突不命中就越少；
2. 强制性不命中和容量不命中不受相联度的影响；
3. 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

（3）减少三种不命中的方法

1. 强制性不命中：增加块大小，预取（本身很少）
2. 容量不命中：增加容量（抖动现象）
3. 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

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

5.3.2 增加Cache块大小

（1）不命中率与块大小的关系：

1. 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*。
2. 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

5.3.3 增加Cache的容量

最直接的方法是增加Cache的容量，这种方法在片外Cache中用得比较多

缺点:

1. 增加成本
2. 可能增加命中时间

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

5.3.5 伪相联Cache（列相联Cache）

思想/原理：在逻辑上把*直接映像Cache*的空间上下平分为两个区。

- 对于任何一次访问，伪相联Cache先按*直接映像Cache*的方式去处理。
  - 若*命中*，则其访问过程与*直接映像Cache*的情况一样。
  - 若*不命中*，则再到另一区相应的位置去查找。*若找到，则发生了伪命中*，否则就只好访问下一级存储器

  优点：

1. 命中时间小
2. 不命中率低

   命中时间：要保证证绝大多数命中都是快速命中，缺点是存在多种命中方式

5.3.6 指令预取

预取：指令和数据都可以预取，预取内容既可放入Cache，也可放在外缓冲器中。（指令流缓冲器）
执行硬件：指令预取通常由Cache之外的硬件完成
注意事项：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
结构组成：先行读数站和后行写数站

5.3.7 编译器控制的预取

定义：在*编译时加入预取指令*，在数据被用到之前发出预取请求。
在预取数据的同时，处理器应能继续执行
分类：

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

  目的：使执行指令和读取数据能重叠执行

预取优化的主要对象：循环，影响不命中开销
每次预取的开销：一条指令

- 保证开销不超过预取的收益
- 编译器重点放在导致不命中的访问上，使程序避免不必要的预取，从而减少平均访存时间

例题：第11次课，p36，循环失效判定

5.3.8 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。

（1）方法：*程序代码和数据重组*

1. 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*，从而减少冲突不命中，降低指令不命中率。
- McFarling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，可降低顺序代码执行时Cache不命中率。（提高大Cache块的效率）

2. 编译器知道分支指令会成功转移，则可以通过如下两步改善空间局限性

- 将*转移目标处的基本块*和*该分支指令后的基本块*进行对调；
- 把该分支指令换为*操作语义相反的分支指令*。

3. 数据对存储位置的限制更少，更便于调整顺序。

（2）编译优化技术

类型：数组合并、内外循环交换、循环融合、分块

1. 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。
这种技术通过提高空间局部性来减少失效次数

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

```
/* 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

定义：将*多个独立的循环融合为单一的循环*，能使读入Cache的数据在*被替换出去之前，得到反复的使用*。这是通过改进时间局部性来减少失效次数。

```
/* 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（不命中次数：$2N^3+N^2$）

N很大时，cache装不下。

```
/* 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：$2N^3/B + N^2$）

5.3.9 "牺牲"cache

功能： 一种能减少冲突不命中次数而又不影响时钟频率的方法。

原理： 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

使用过程： 每当发生失效时，在访问下一级存储器之前，先*检查VictimCache中是否含有所需的块*。如果有，就将该块与Cache中某个块做交换

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

5.4 减少Cache不命中开销

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
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

5.4.1 采用两级Cache

1. 原理：再增加一级Cache，第一级Cache(L1)小而快，第二级Cache(L2)容量大
2. 性能指标：例题：ppt p58

- *平均访存时间*

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

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

- *局部不命中率和全局不命中率*

$$局部不命中率 = \frac{当前级别Cache不命中次数}{当前级别Cache访存数}$$

$$全局不命中率 = \frac{当前级别Cache不命中次数}{CPU总访存次数}$$

特殊例子：$全局不命中率_{L2} = 不命中率_{L1} \times 不命中率_{L2}$

- *平均访存停顿时间*

$$各指令平均访存停顿时间 = 各指令平均不命中次数_{L1} \times 命中时间_{L2} + 各指令平均不命中次$$

3. 第二级Cache相关结论

- 在第二级Cache比第一级 Cache大得多的情况下，两级Cache的全局不命中率和容量与第二级Cache相同的单级Cache的不命中率非常接近。
- 局部不命中率不是衡量第二级Cache的一个好指标，因此，在评价第二级Cache时，应用全局不命中率这个指标
- 第二级Cache不会影响CPU的时钟频率，因此其设计有更大的考虑空间

4. 第二级Cache的参数

- **容量：** 第二级比第一级大很多（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
- **块大小：** 第二级Cache可采用较大的块（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。

5.4.2 让读不命中优先于写

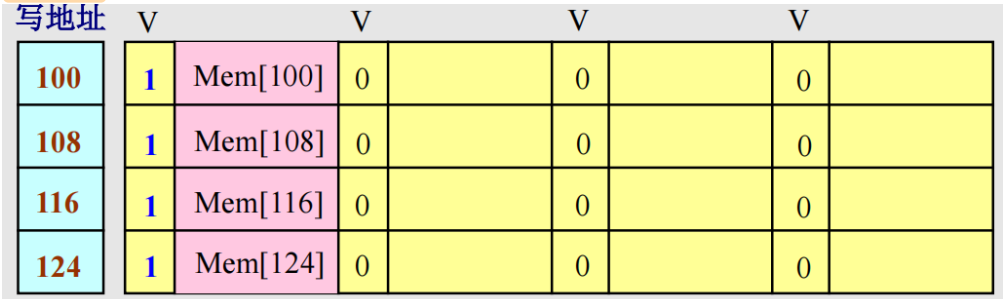
问题：Cache中的写缓冲器导致对存储器访问的复杂化，在读不命中时，所读单元的最新值有可能还在写缓冲器中，尚未写入主存
解决问题的方法：

- 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

5.4.3 写缓冲合并

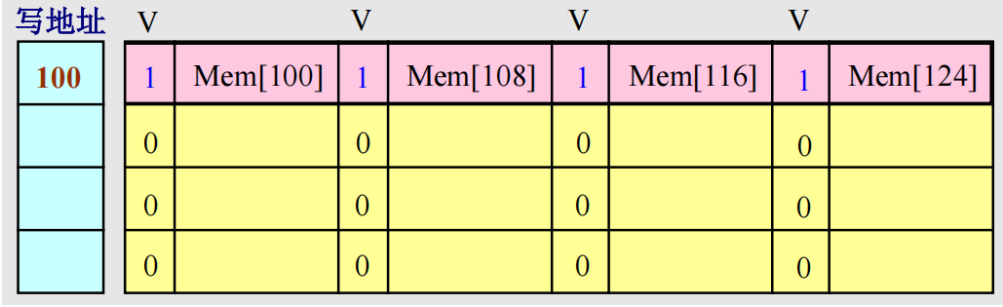
目的：提高写缓冲器的效率
实现：对于写直达Cache，依靠写缓冲来减少对下一级存储器写操作的时间

- 如果*写缓冲器为空*，就把数据和相应地址写入该缓冲器
- 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
  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
  效果：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	

5.4.4 请求字处理技术

请求字：从下一级存储器调入Cache的块中，只有一个字是立即需要的。这个字称为请求字
处理方法：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。
  效果不大的情况：

1. Cache块较小
2. 下一条指令正好访问同一Cache块的另一部分

5.4.5 非阻塞Cache技术

非阻塞Cache：*Cache不命中*时仍允许CPU进行*其他的命中访问*。即允许“不命中下命中”（不阻塞）
提高性能的方法：

- “多重不命中下命中”

- “不命中下不命中”
  规律：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。

5.5 减少命中时间

优化技术	不命中率	不命中开销	命中时间	硬件复杂度	说 明
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

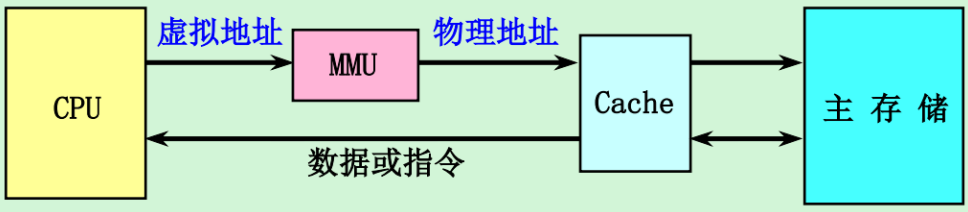
5.5.1 容量小、结构简单的Cache

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

5.5.2 虚拟Cache

（1）概念

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)
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
物理Cache存储系统缺点：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

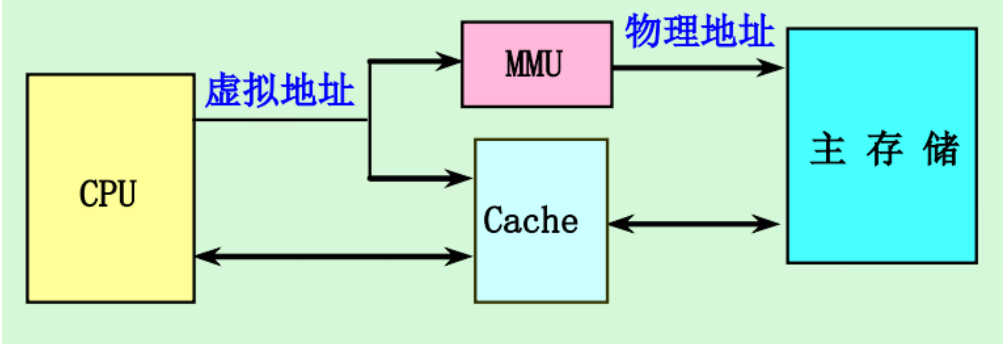
（2）虚拟Cache

定义：

- 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



存在的问题：

- *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)
- *同义和别名*：解决方法：反别名法、页着色

（3）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≤页大小×相联度

（4）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

- 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

5.5.4 踪迹 Cache

挑战：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
解决方法：采用**踪迹Cache**，存放CPU所执行的*动态指令序列*，包含了由分支预测展开的指令，

该分支预测是否正确需要在取到该指令时进行确认。

优缺点：

1. 地址映像机制复杂，
2. 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，
3. 能够提高指令Cache的空间利用率。

5.5.5 Cache优化技术总结

1. “+”号：表示改进了相应指标。
2. “-”号：表示它使该指标变差。
3. 空格栏：表示它对该指标无影响。
4. 复杂性：0表示最容易，3表示最复杂

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