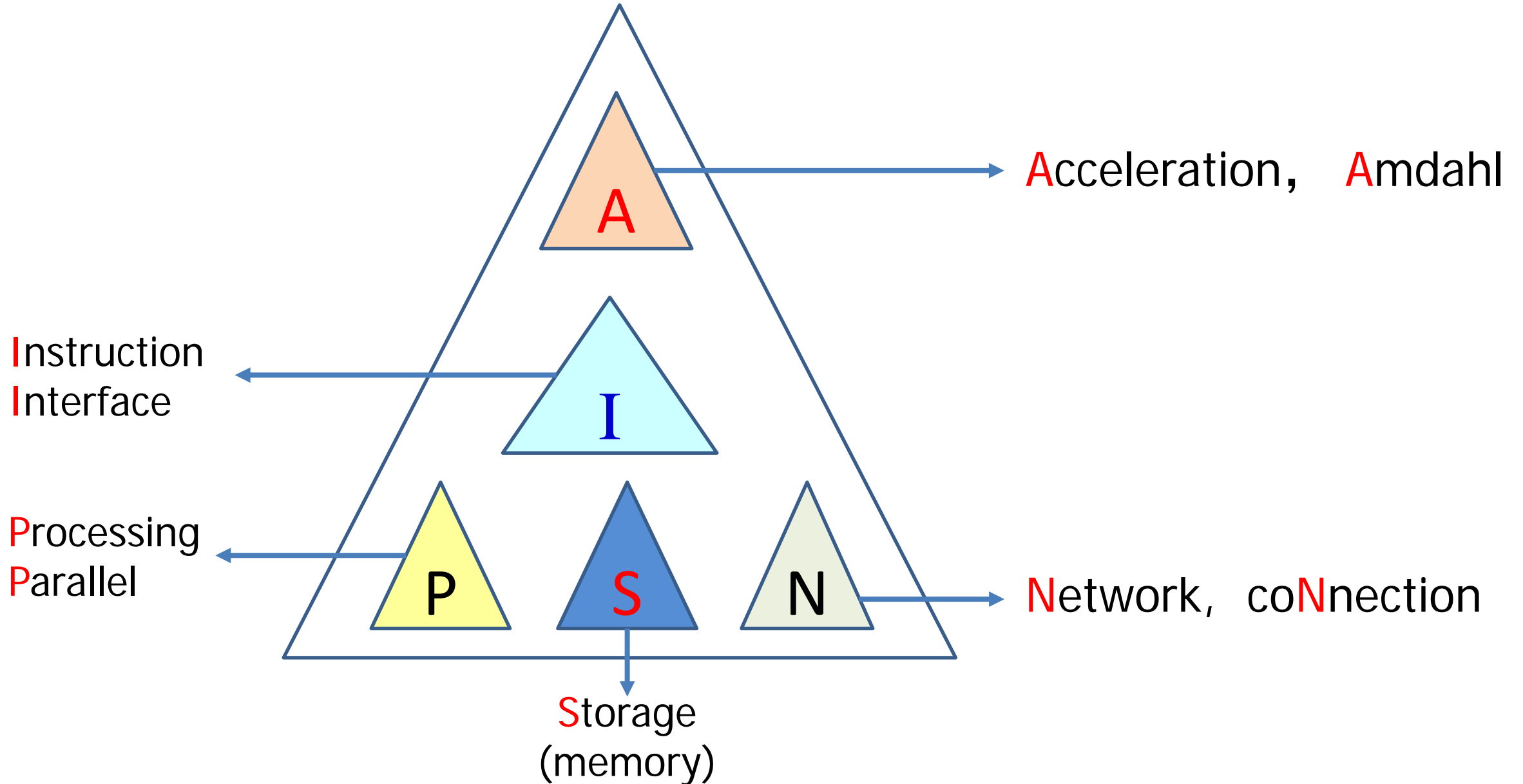


大A: Architecture



第7章 存储系统 (Memory Hierarchy)

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

7.7 虚拟存储器

7.8 实例：AMD Opteron的存储器层次结构

本章作业

7.9

7.10

7.11

7.14

7.1 存储系统的基本知识

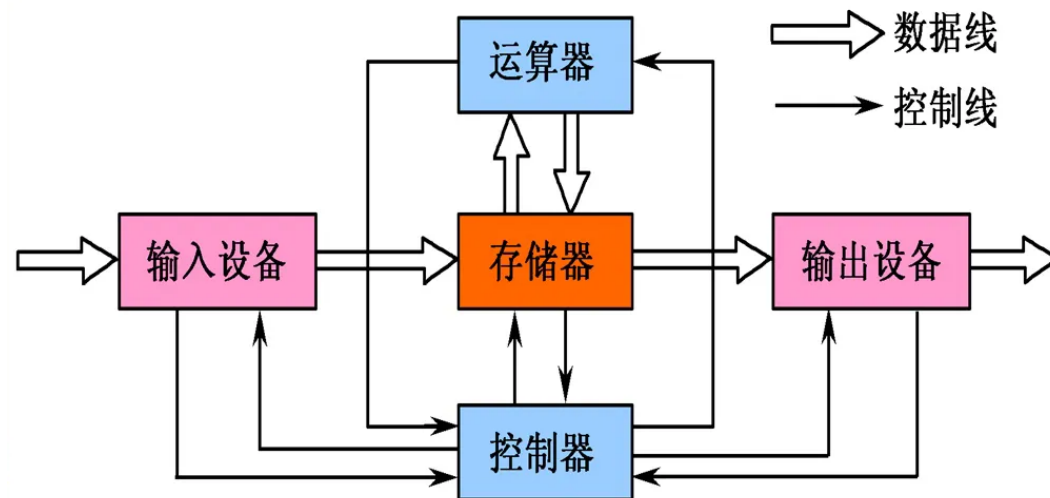
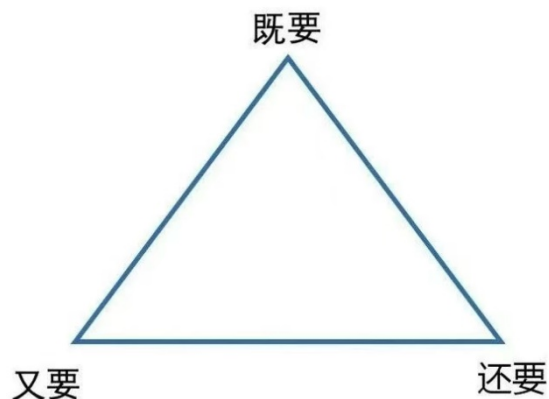
7.1.1 存储系统的层次结构（Memory Hierarchy）

1. 理想存储系统：

容量足够大、**速度**足够快、**价格**足够低

2. 三个要求是相互矛盾的

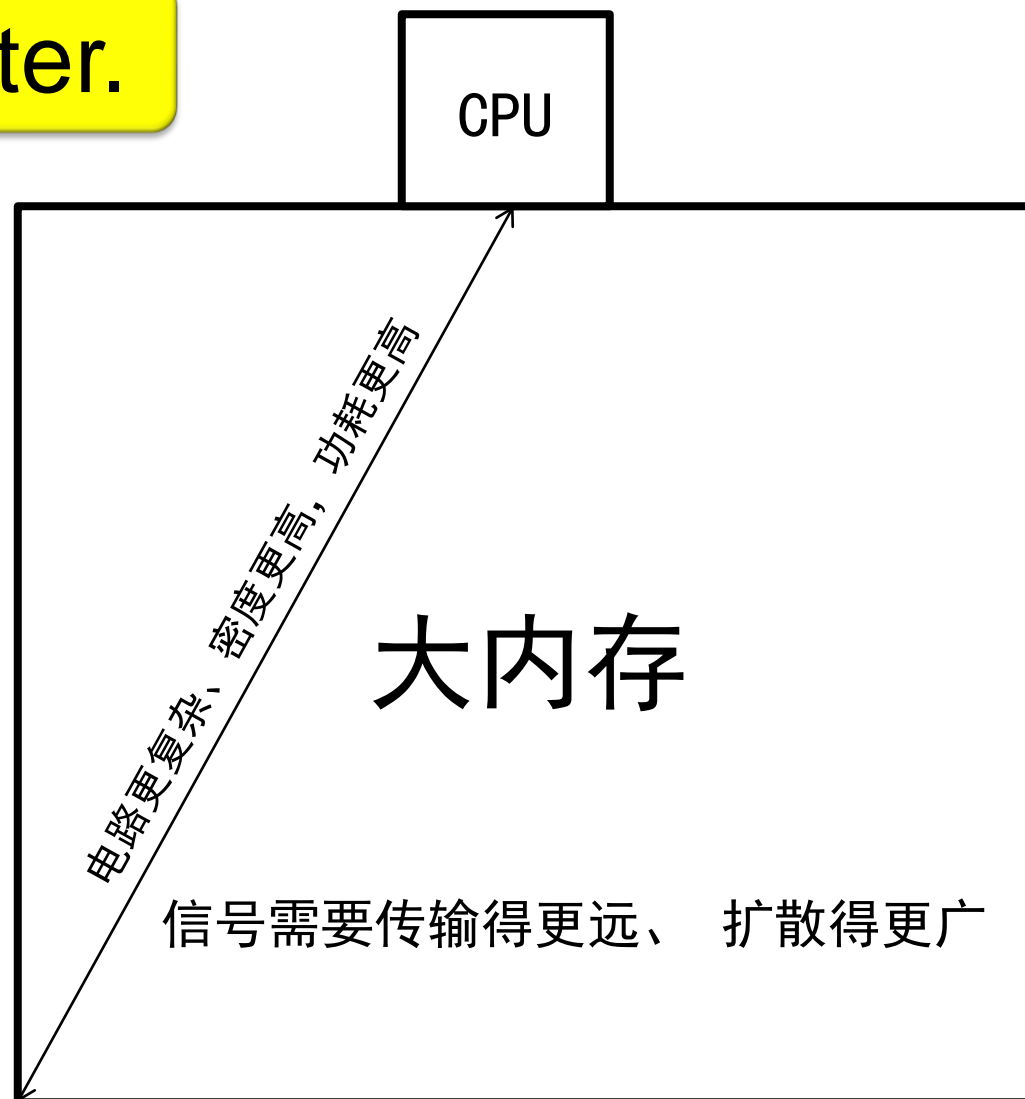
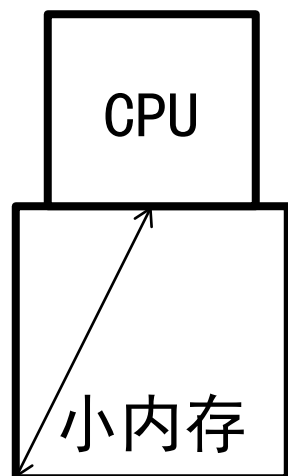
- 速度越快，每位价格就越高
- 容量越大，每位价格就越低
- 容量越大，速度越慢



冯.诺依曼结构

处理器物理尺寸影响延迟时间

The Smaller is faster.



CPU与Memory之间的性能差异越来越大

1. CPU性能提高大约60%/year

2. DRAM 性能提高大约 9%/year

	Capacity	Speed (latency)
Logic:	2x in 3 years	2x in 3 years
DRAM:	4x in 3 years	2x in 10 years
Disk:	4x in 3 years	2x in 10 years

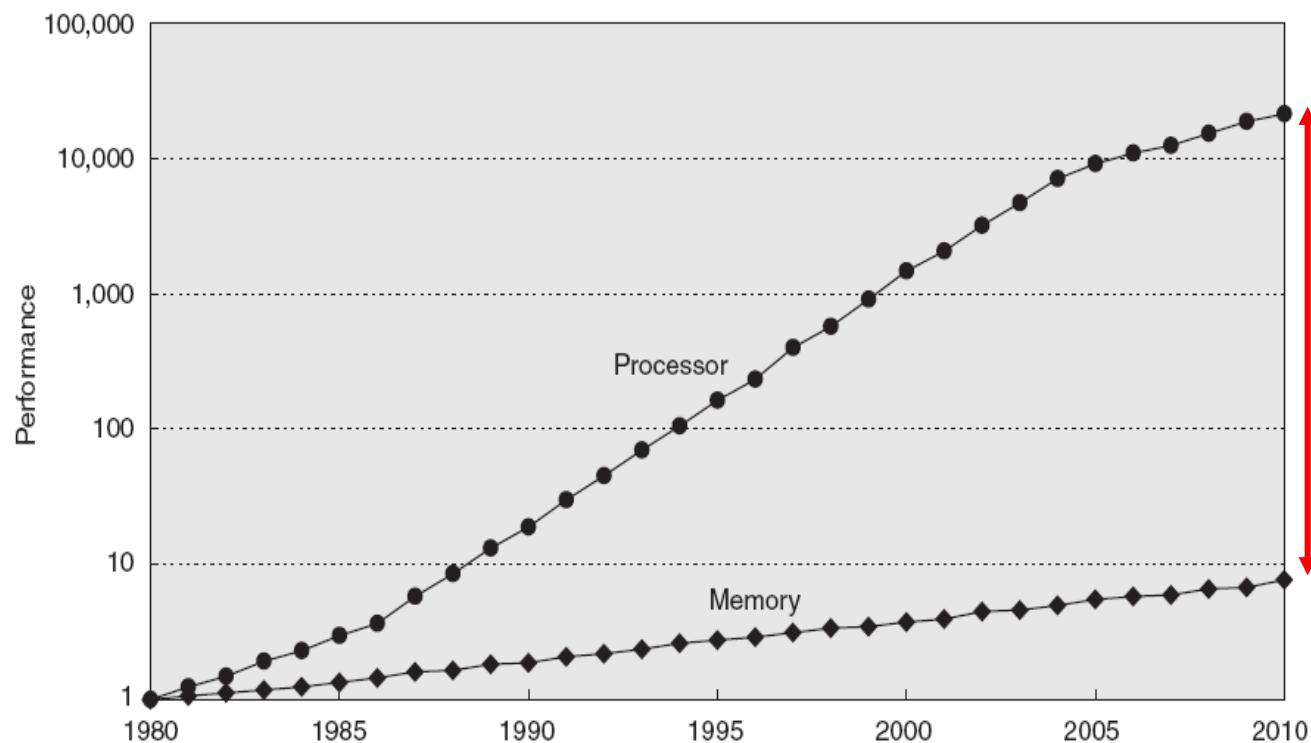
Year	DRAMSize	Cycle Time
1980	64 Kb	250 ns
1983	256 Kb	220 ns
1986	1 Mb	190 ns
1989	4 Mb	165 ns
1992	16 Mb	145 ns
1995	64 Mb	120 ns

1000:1!

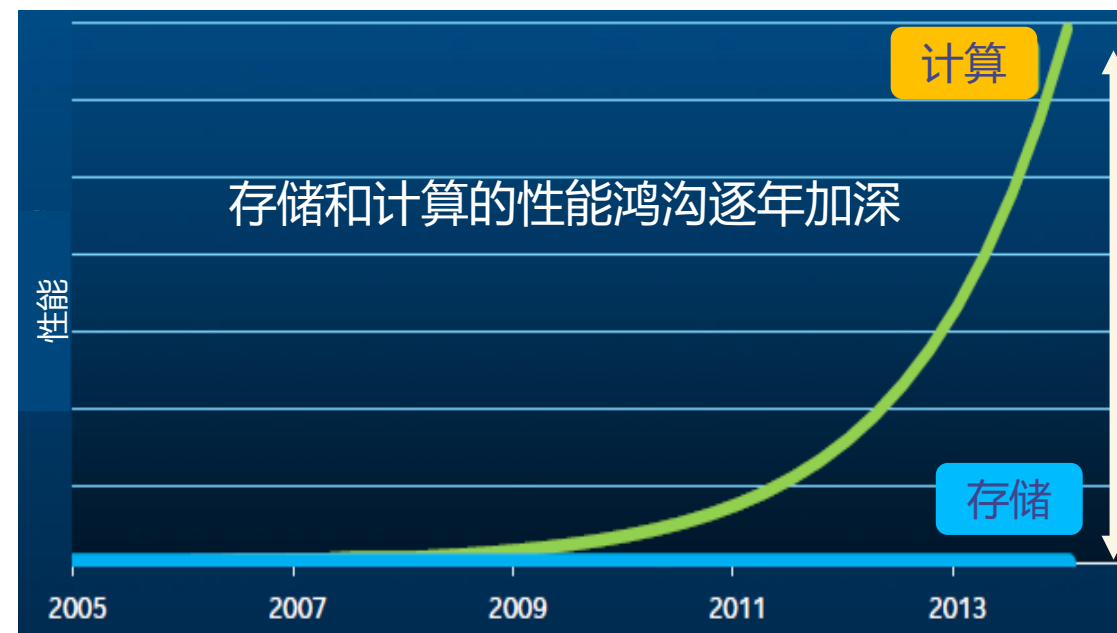
2:1!

7.1 存储系统的基本知识

Processor-Memory Performance Gap Growing



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)

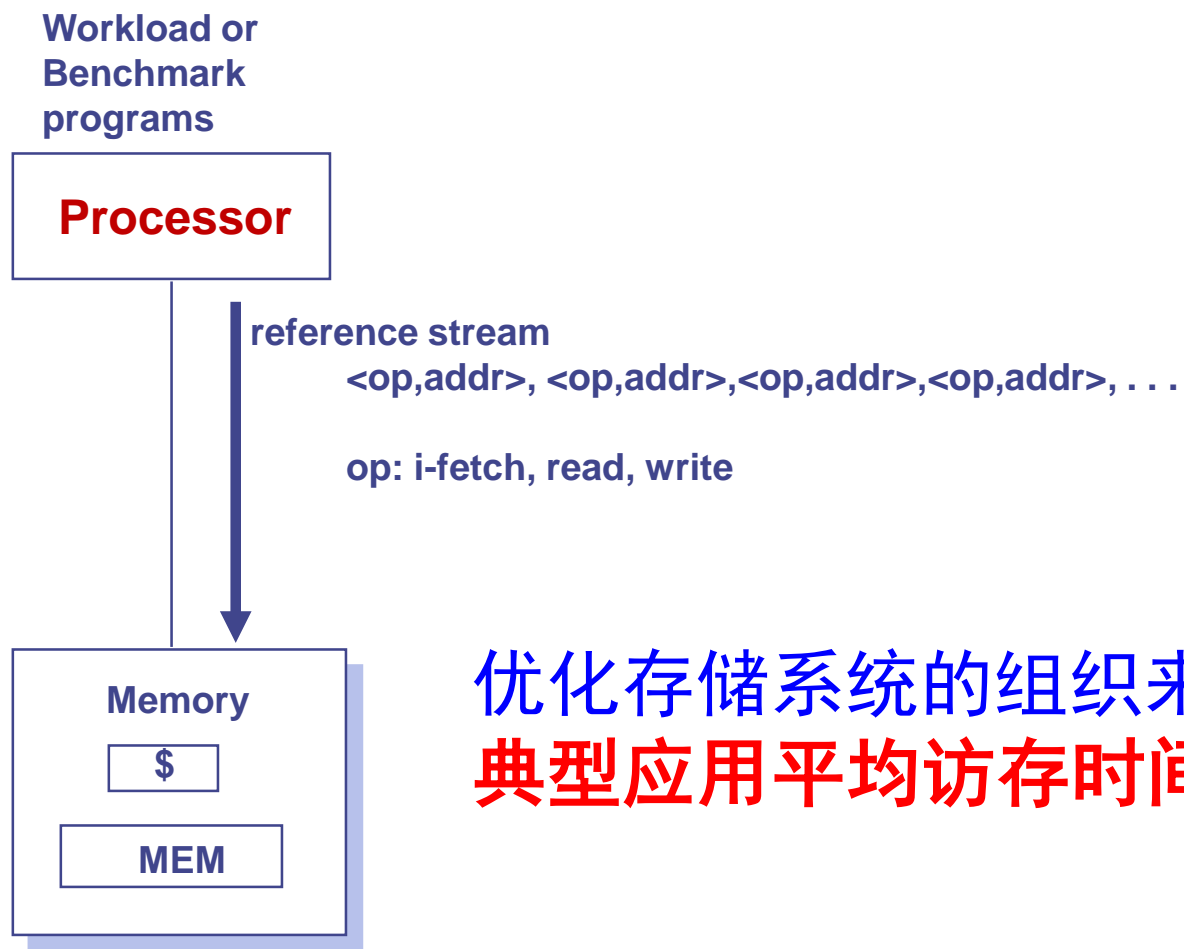


源自：英特尔新闻中心

存储系统的设计目标

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**都满足计算需求的存储器系统？



优化存储系统的组织来使得针对**典型应用平均访存时间最短**

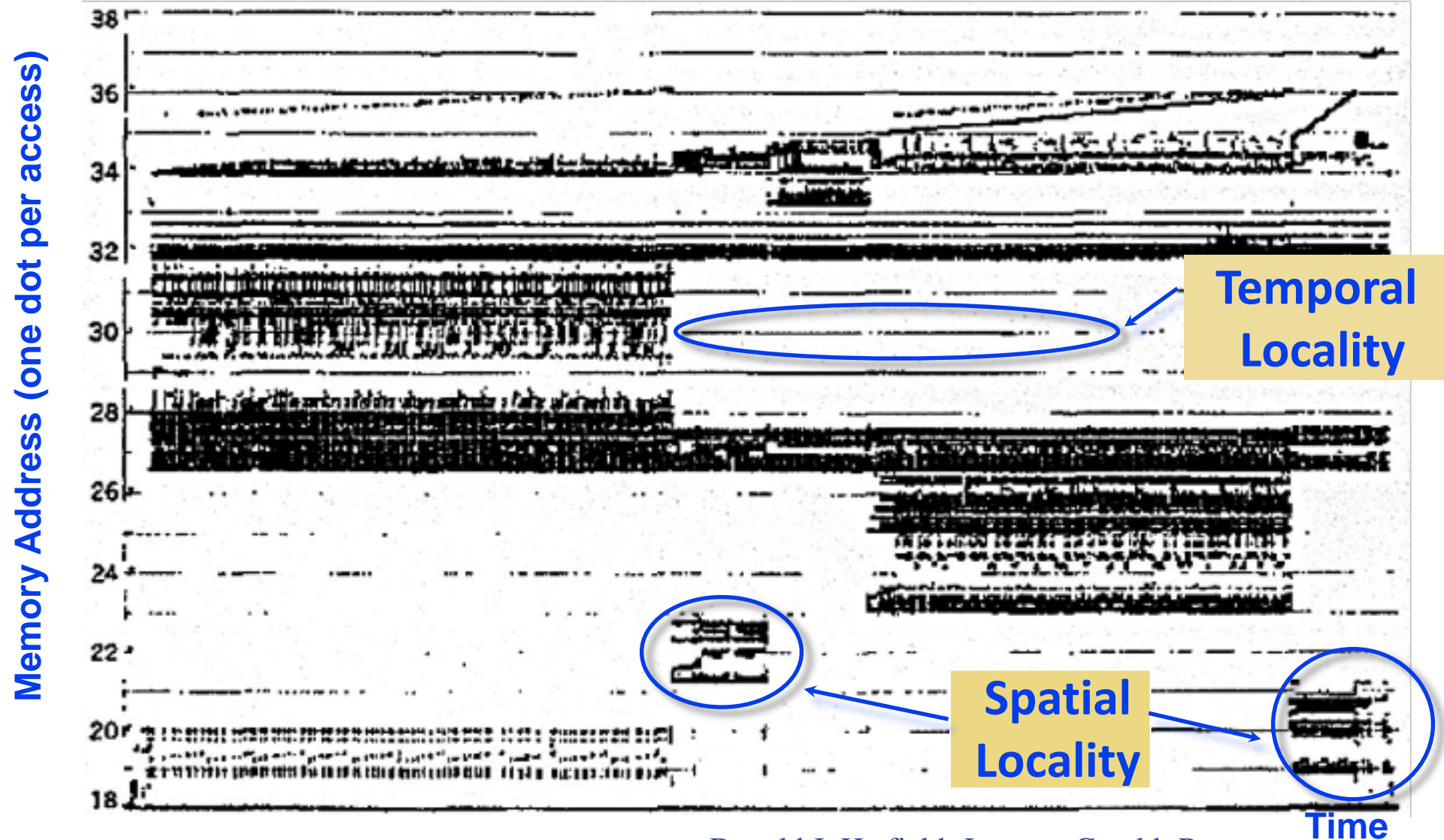
7.1 存储系统的基本知识

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

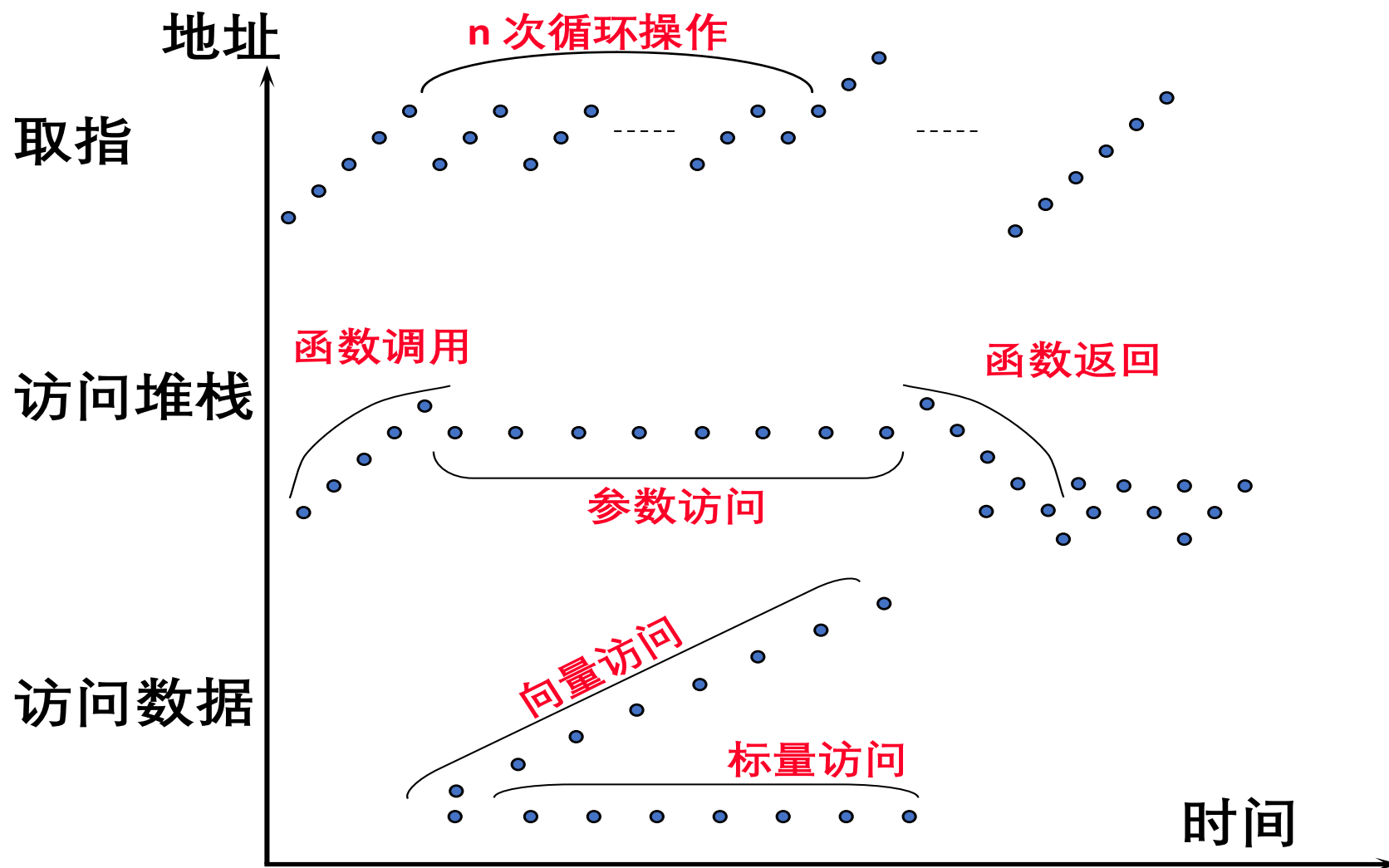
原理：加快经常性事件

Memory Reference Patterns



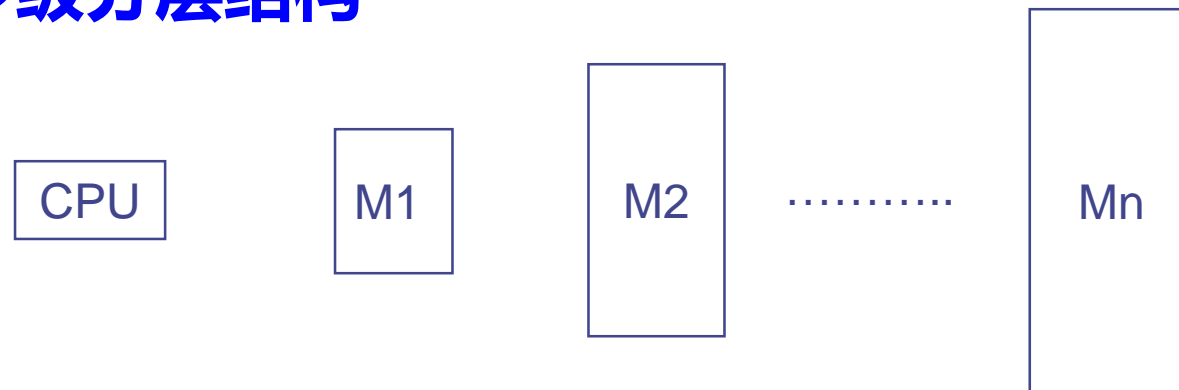
Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

访存局部性的主要来源



基本解决方法：多级层次结构

- 多级分层结构



- M1 速度最快，容量最小，每位价格最高
- Mn速度最慢，容量最大，每位价格最低

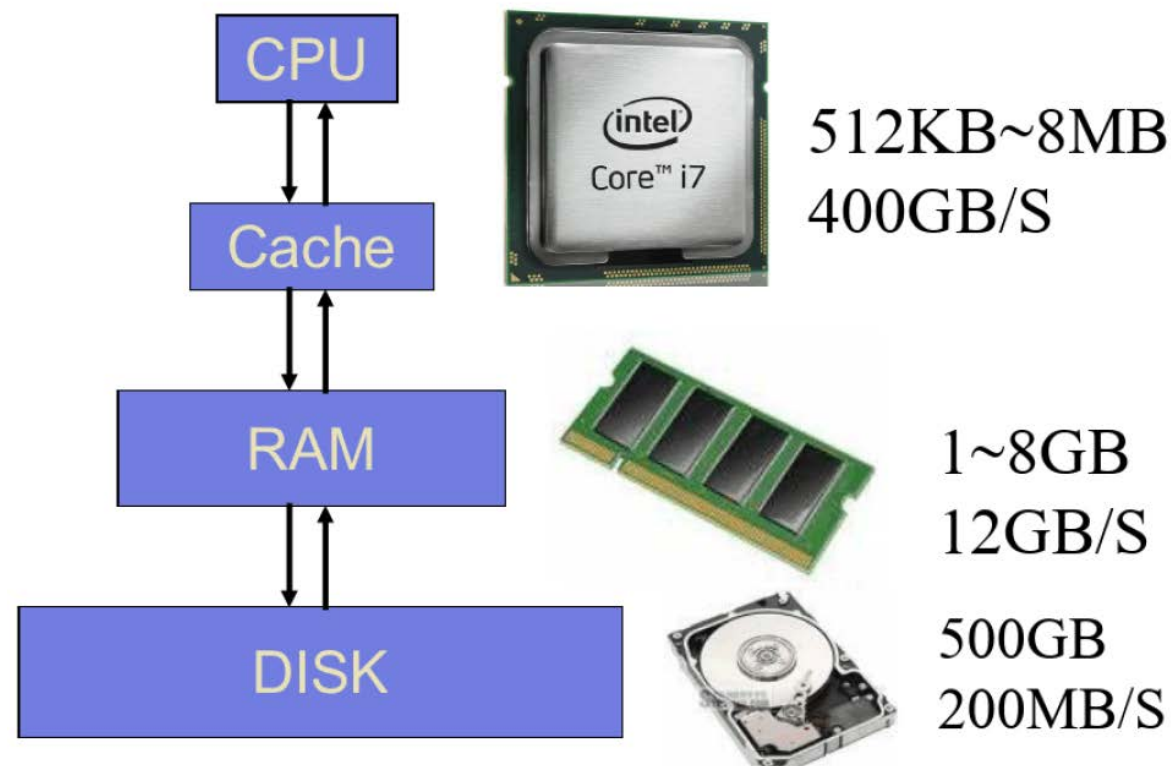
- 并行

- 存储系统接近M1的速度，容量和价格接近Mn

多级层次结构

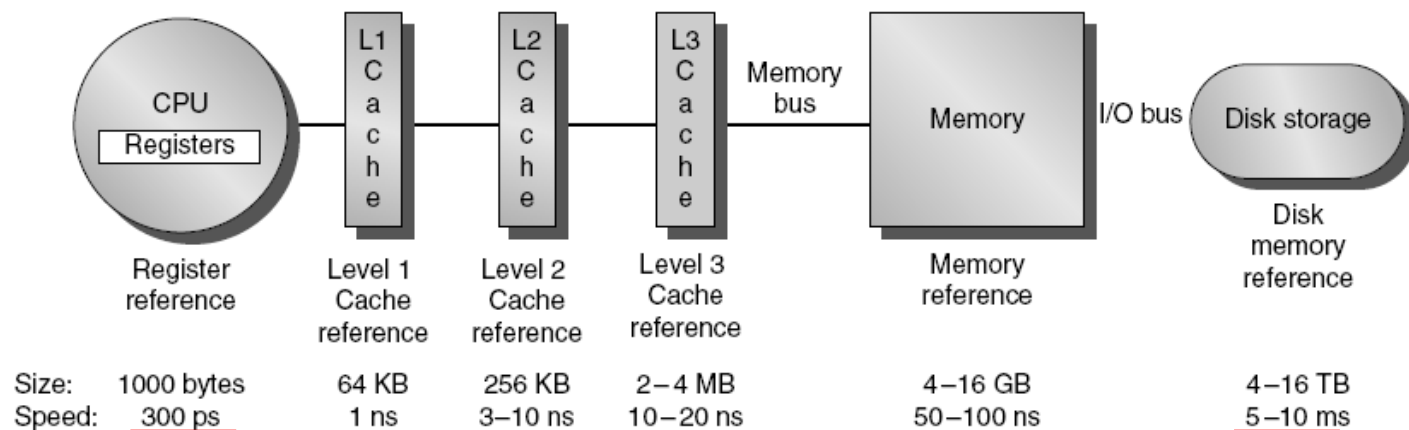


存储器的层次结构

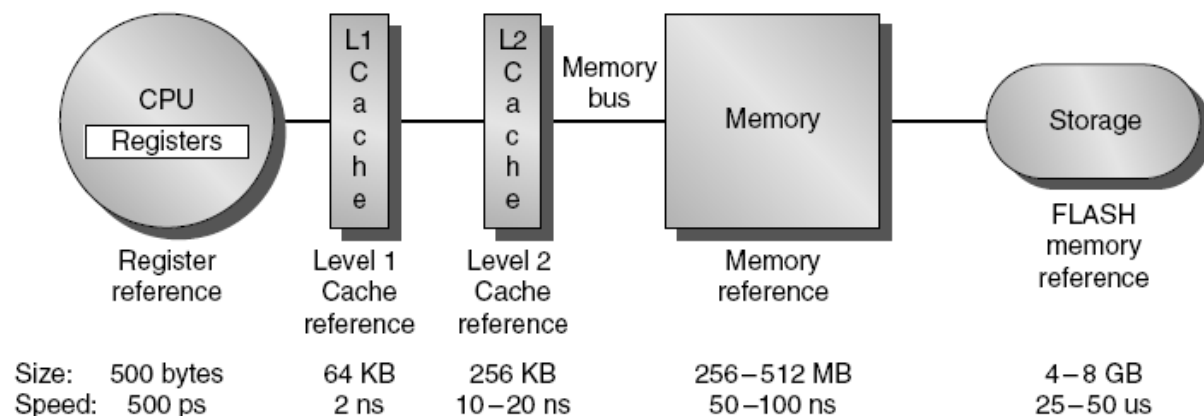


7.1 存储系统的基本知识

5. 存储系统的多级层次结构



(a) Memory hierarchy for server

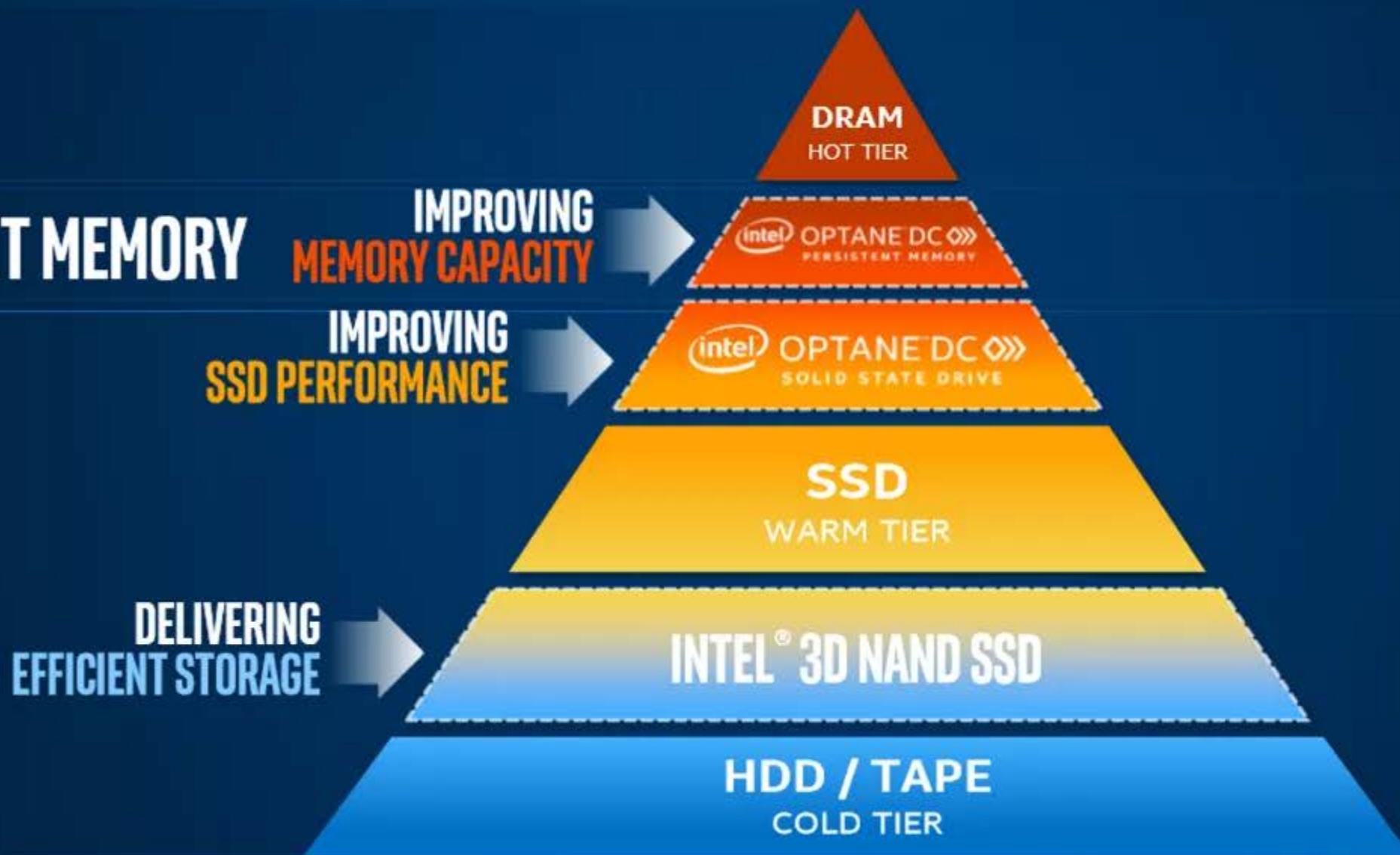


(b) Memory hierarchy for a personal mobile device

MEMORY

PERSISTENT MEMORY

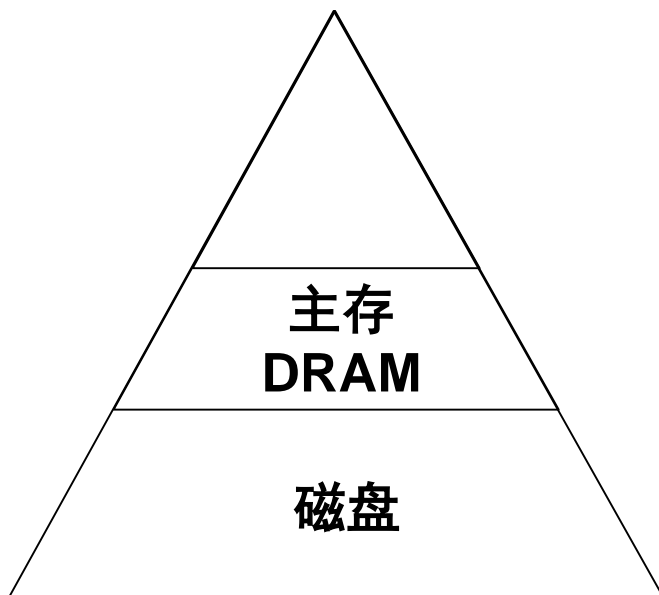
STORAGE



存储层间融合

传统存储

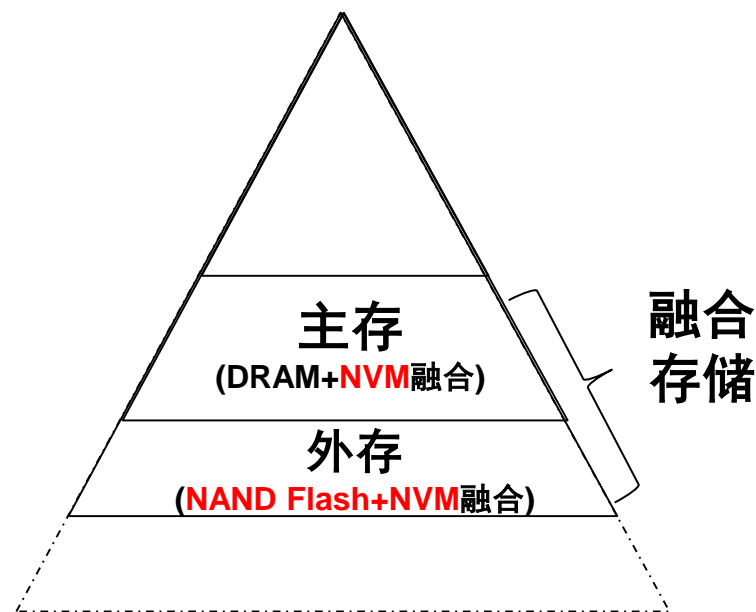
- 由于内存易失，突然断电后数据很容易丢失
- 主机访问磁盘速度慢
- 磁盘和主存的能耗高



传统存储结构模式

融合存储

- 采用灵活的软件定义存储结构
- 采用新型非易失存储介质
- 读写速度快，性能高
- 能耗低



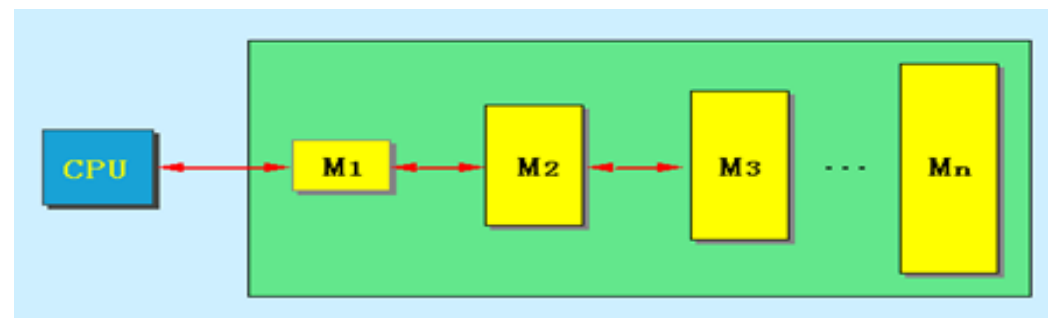
融合存储结构模式

7.1 存储系统的基本知识

层次结构存储器

- 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: $C_1 > C_2 > \dots > C_n$



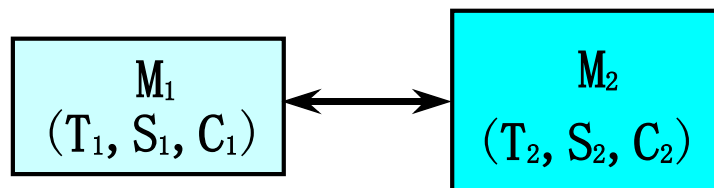
- 整个存储系统要达到的目标: 从CPU来看，速度接近于 M_1 的，而容量和每位价格都接近于 M_n
 - 存储器越靠近CPU，则CPU对它的访问频度越高，而且最好大多数的访问都能在 M_1 完成

7.1 存储系统的基本知识

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



7.1 存储系统的基本知识

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, $C \approx C_2$

7.1 存储系统的基本知识

3. 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**： $F = 1 - H$

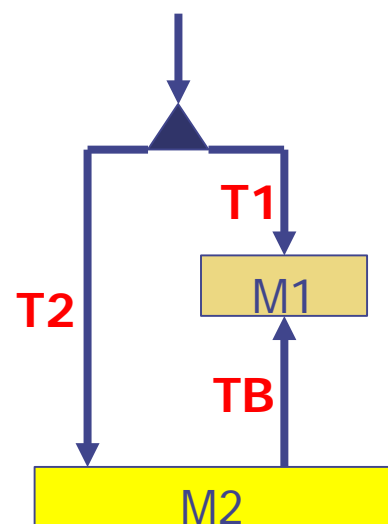
7.1 存储系统的基本知识

4. 平均访问时间 T_A

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

$$\text{或 } T_A = T_1 + FT_M$$

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： $T_2 + T_B + T_1 = T_1 + T_M$

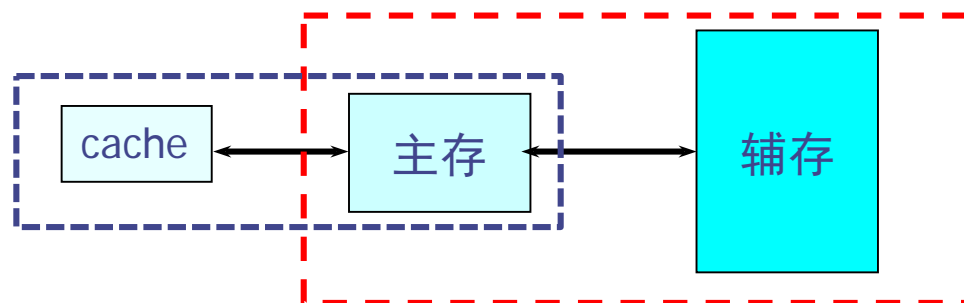
$$T_M = T_2 + T_B$$

- 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 存储系统的基本知识

7.1.3 三级存储系统

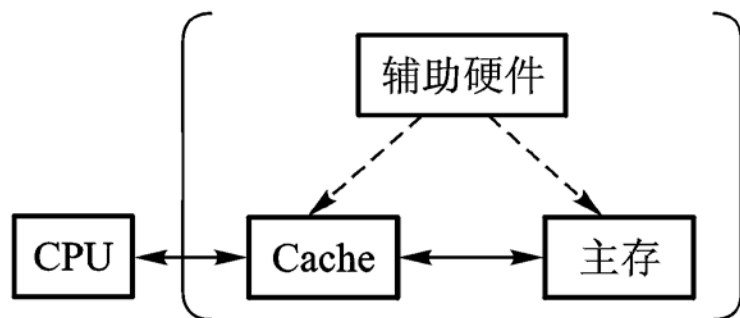
- 三级存储系统
 - ❑ Cache（高速缓冲存储器）
 - ❑ 主存储器
 - ❑ 外存储器（辅存）
- 可以看成是由“Cache—主存”层次和“主存—辅存”层次构成的系统。



7.1 存储系统的基本知识

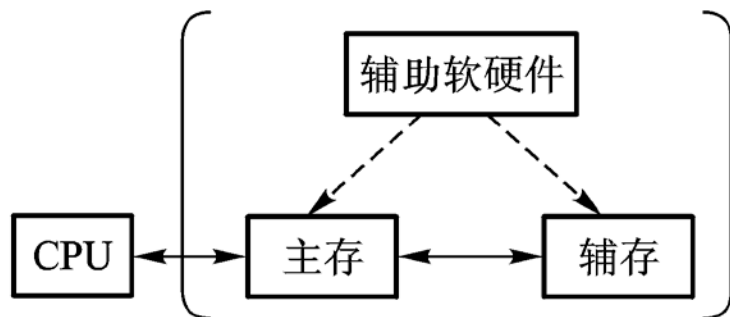
主存的角度来看

两种存储层次



(a) “Cache-主存” 层次

➤ 弥补主存速度的不足



(b) “主存-辅存” 层次

➤ 弥补主存容量的不足

7.1 存储系统的基本知识

“Cache—主存”与“主存—辅存”层次的区别

<div>存储层次</div> <div>比较项目</div>	Cache — 主存	主存—辅存
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 存储系统的基本知识

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)

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

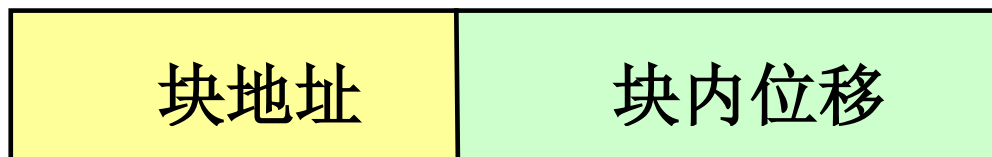
Cache是按块进行管理的

Cache和主存均被分割成大小相同的块（Cache Line）

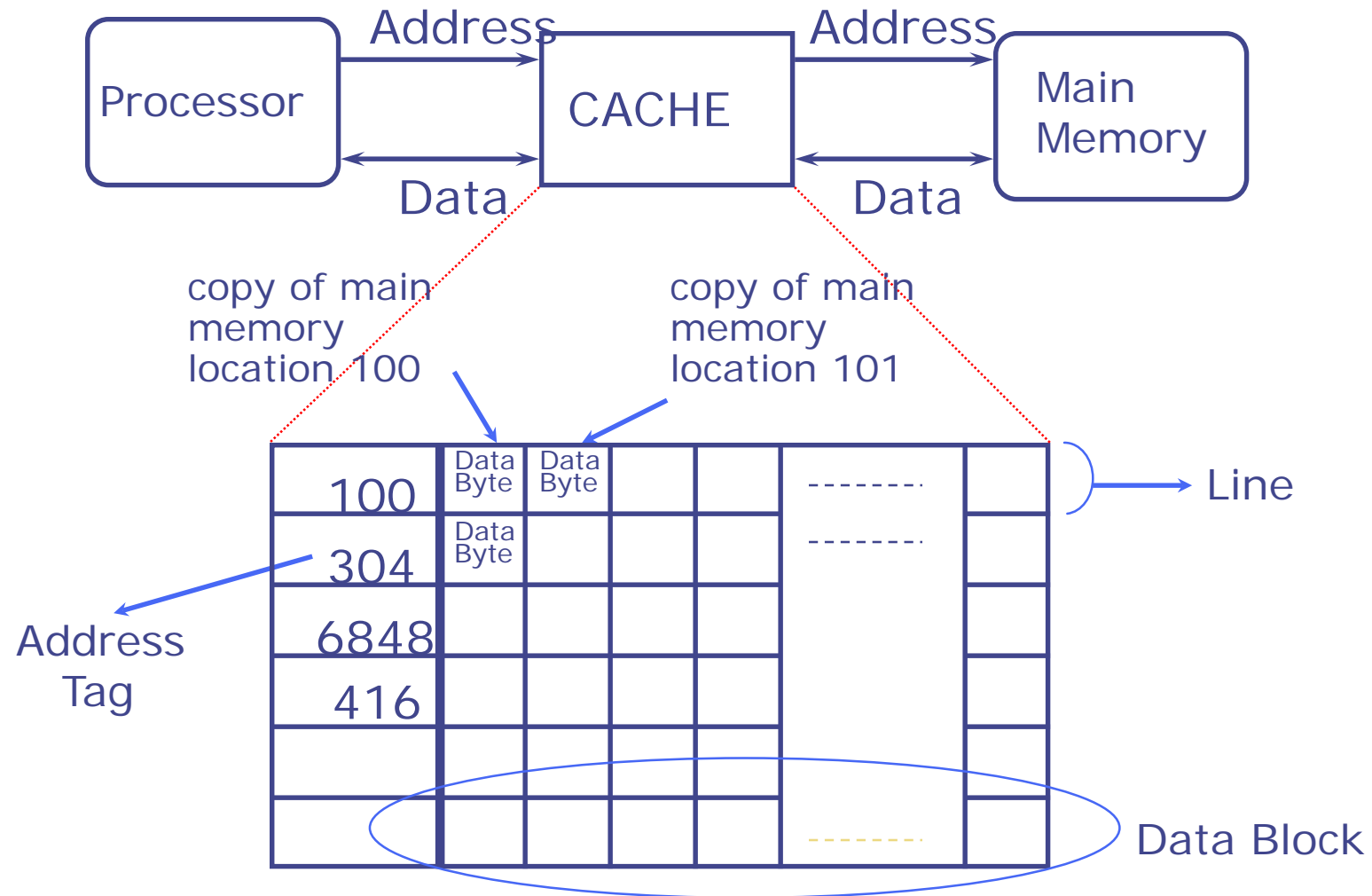
信息以块为单位调入Cache

- **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：



Inside a Cache



Cache的基本工作原理示意图

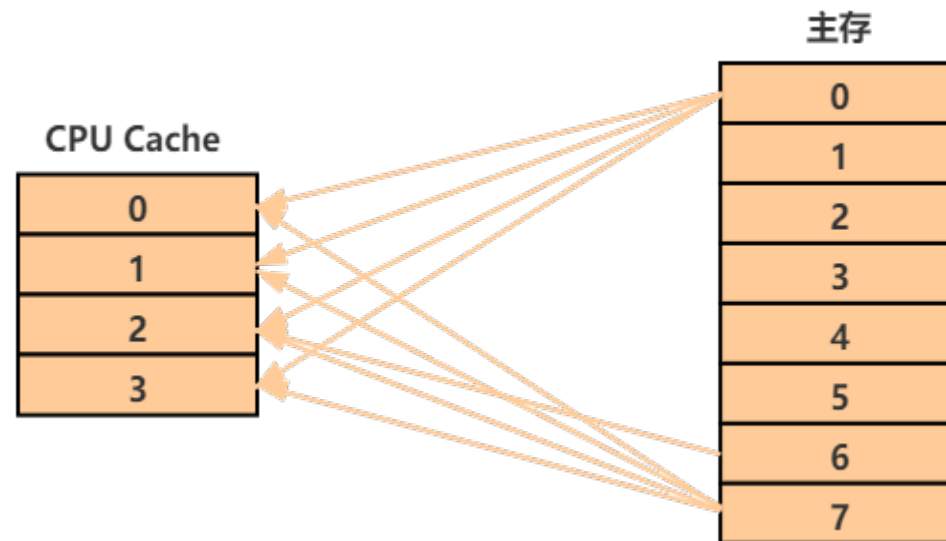


7.2 Cache基本知识

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
- **对比：**大学教室位置 —— 随便坐
- **特点：**空间利用率最高，冲突概率最低，实现最复杂，**查找速度慢**

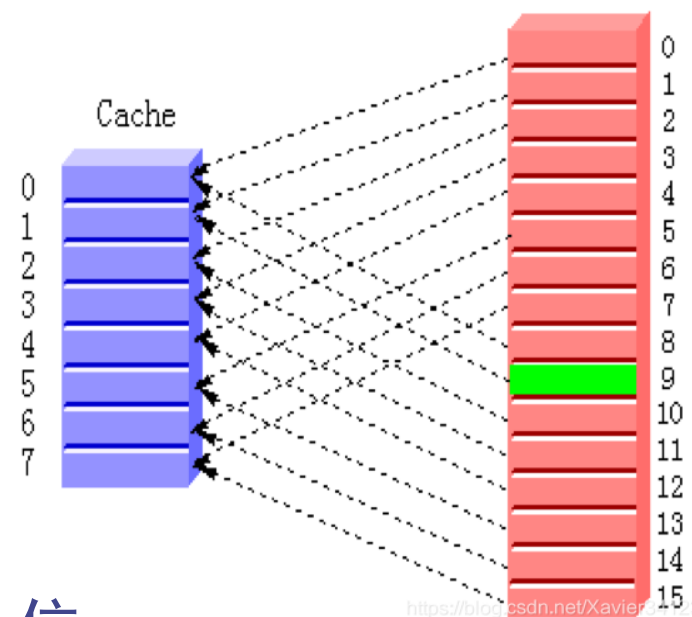


7.2 Cache基本知识

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
- **对比**：中学教室位置 —— 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，**查找速度最快**
- 对于主存的第 i 块，若它映象到Cache的第 j 块，则：

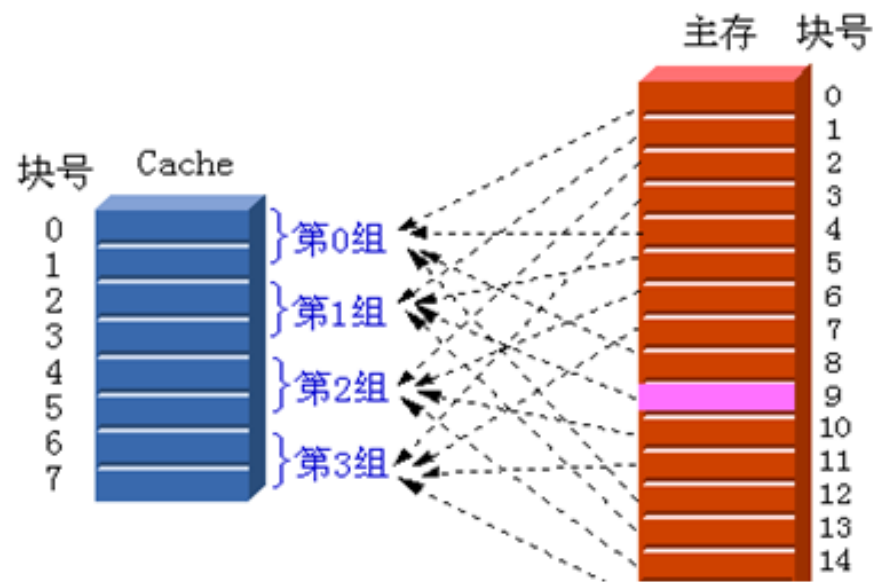
$$j = i \bmod (M) \quad (M \text{ 为Cache的块数})$$
- 设 $M=2^m$ ，则当表示为二进制数时， j 实际上就是 i 的低 m 位：



7.2 Cache基本知识

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。(直接映像到组，组内全相联)
- 组相联是直接映象和全相联的一种折衷



7.2 Cache基本知识

➤ 组的选择

- 若主存第 i 块映象到第 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 位以及直接映象中的低 m 位通常称为**索引 (Index)**

7.2 Cache基本知识

- **n 路组相联**：每组中有 n 个块 ($n = M/G$)。

n 称为**相联度**。

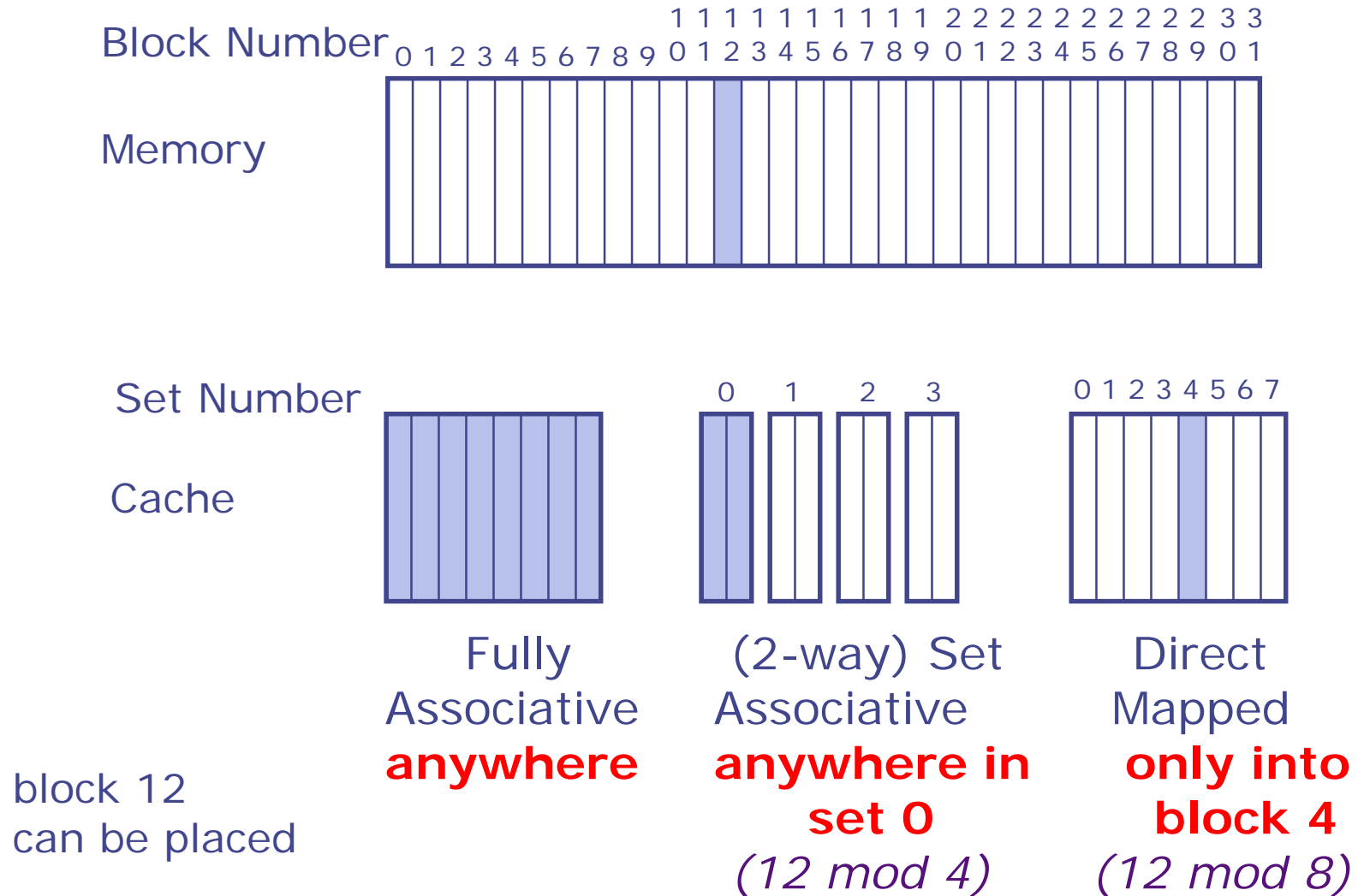
相联度越高，Cache空间的利用率就越高，块冲突概率就越低，不命中率也就越低。

Q：相联度是否越大越好？

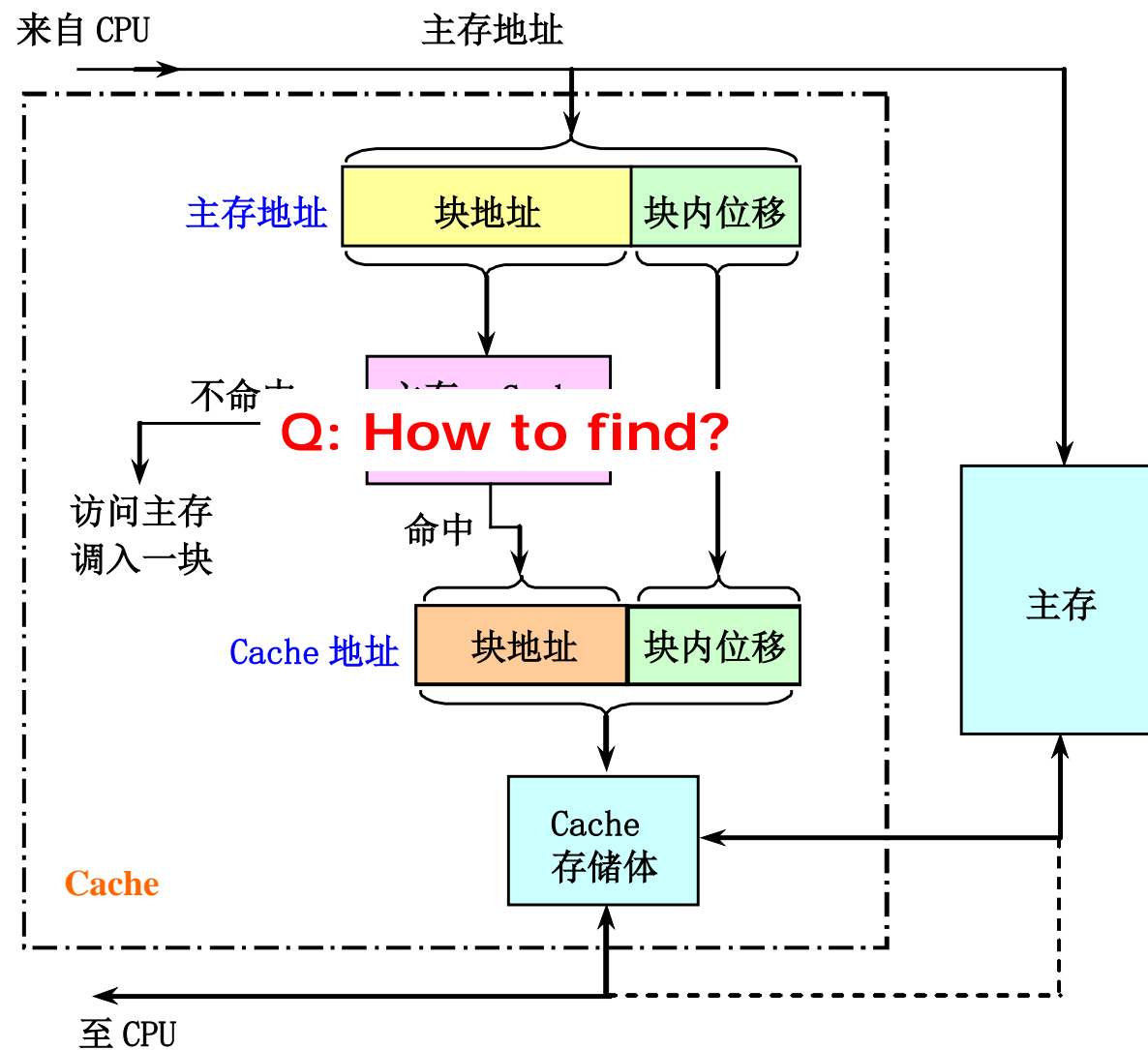
	n (路数)	G (组数)
全相联	M	1
直接映象	1	M
组相联	$1 < n < M$	$1 < G < M$

- 绝大多数计算机的Cache： $n \leq 16$

Placement Policy



Cache的基本工作原理示意图



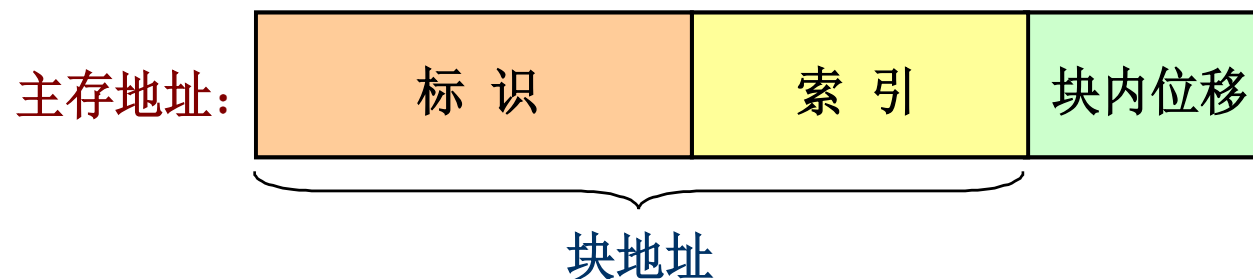
7.2 Cache基本知识

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 (Tag)**
 - 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



7.2 Cache基本知识

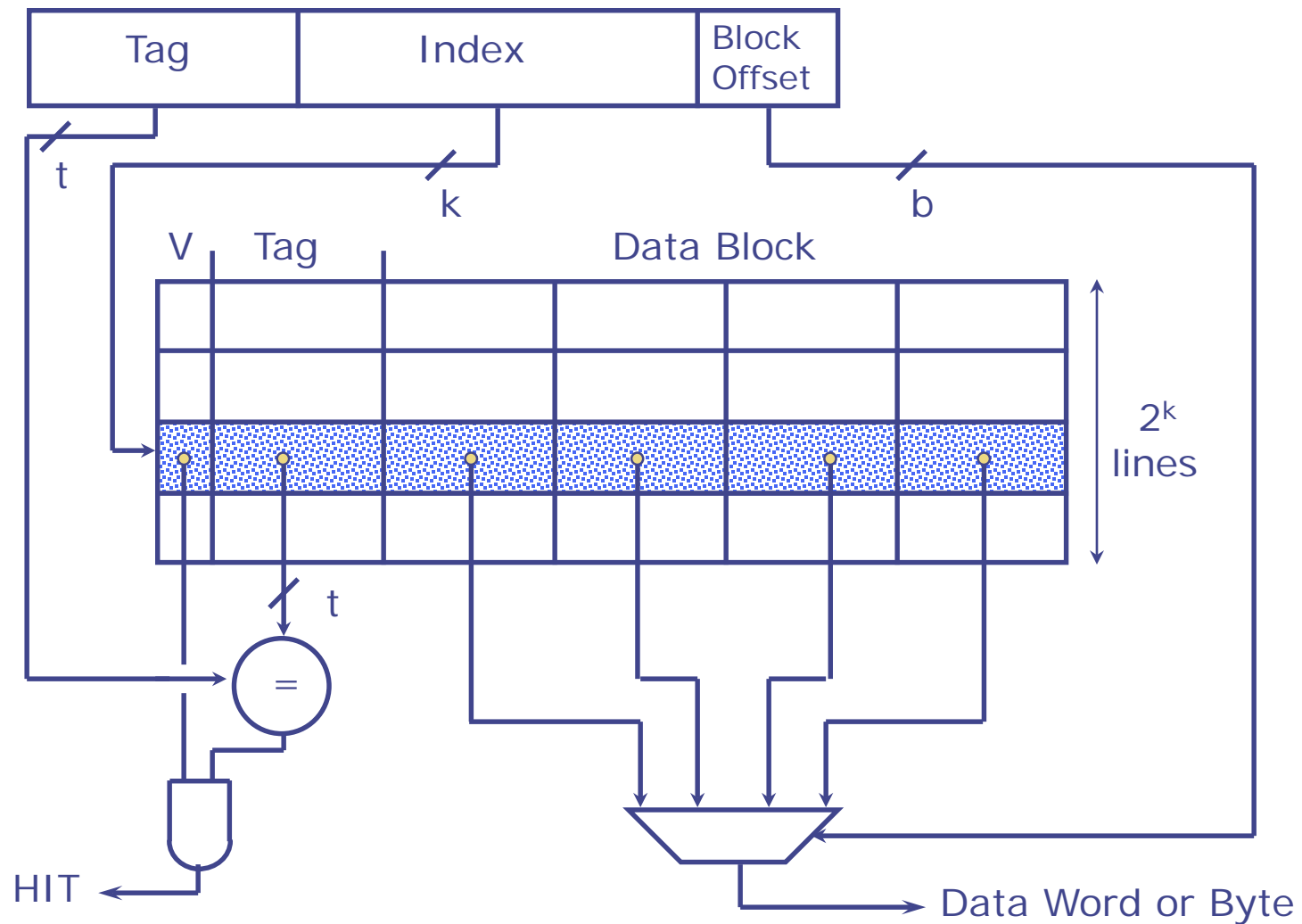
2. 并行查找的实现方法

➤ 相联存储器(P196, 图7.9)

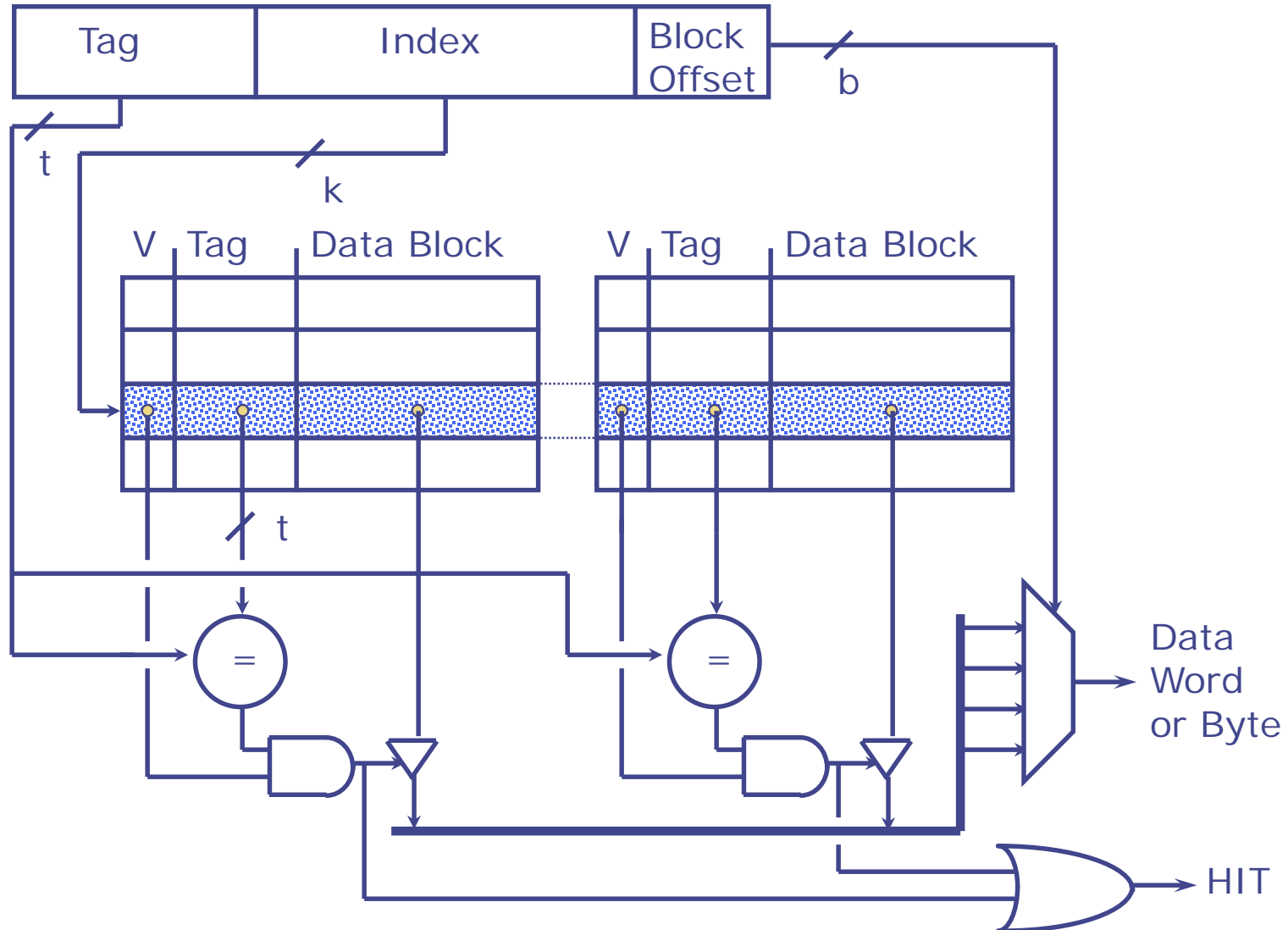
- 目录由 2^g 个相联存储区构成, 每个相联存储区的大小为 $n \times (h + \log_2 n)$ 位
- 根据所查找到的组内块地址, 从Cache存储体中读出的多个信息字中选一个, 发送给CPU

➤ 单体多字的按地址访问的存储器和比较器

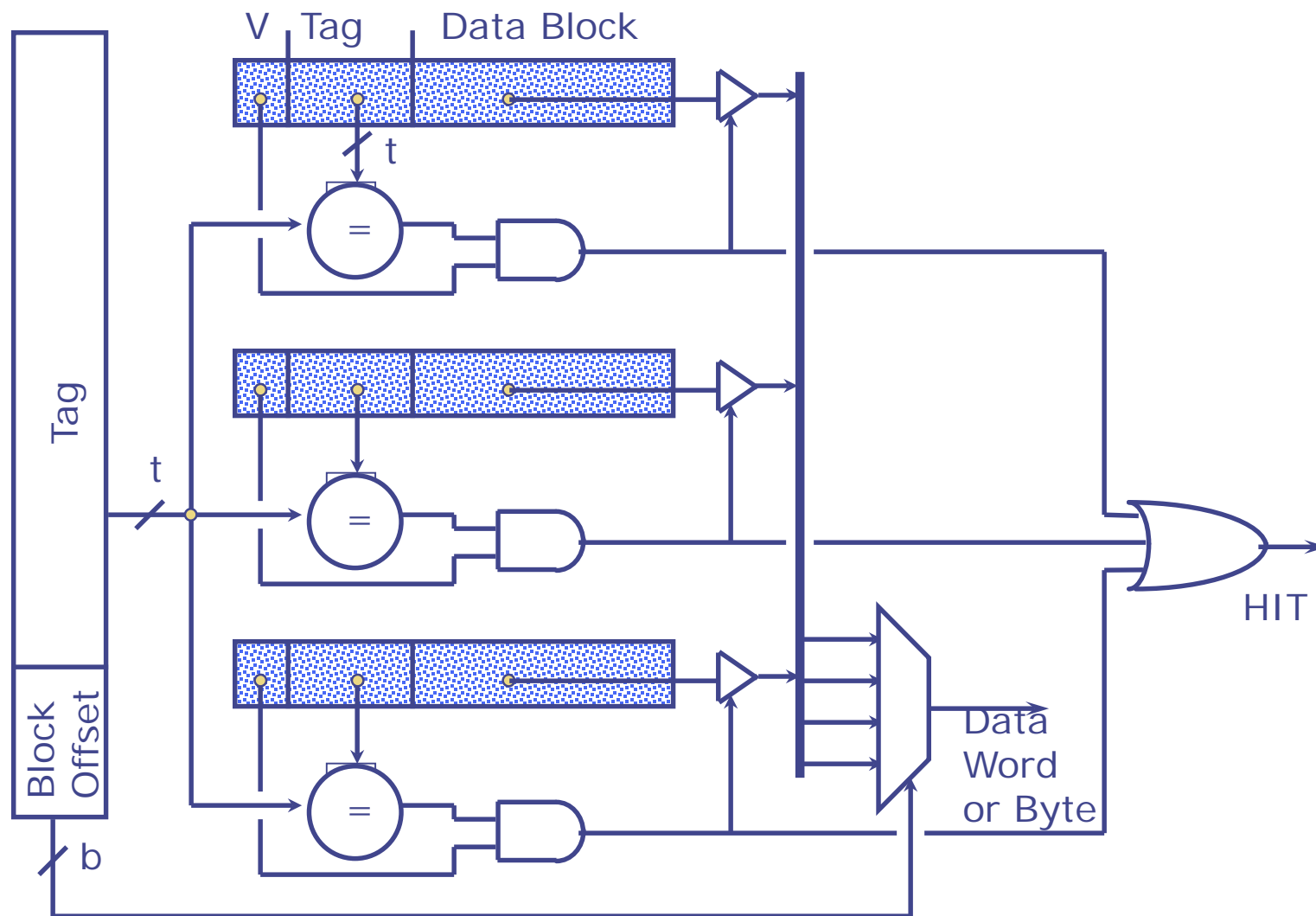
Direct-Mapped Cache



2-Way Set-Associative Cache



Fully Associative Cache



7.2 Cache基本知识

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

公 平

7.2 Cache基本知识

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

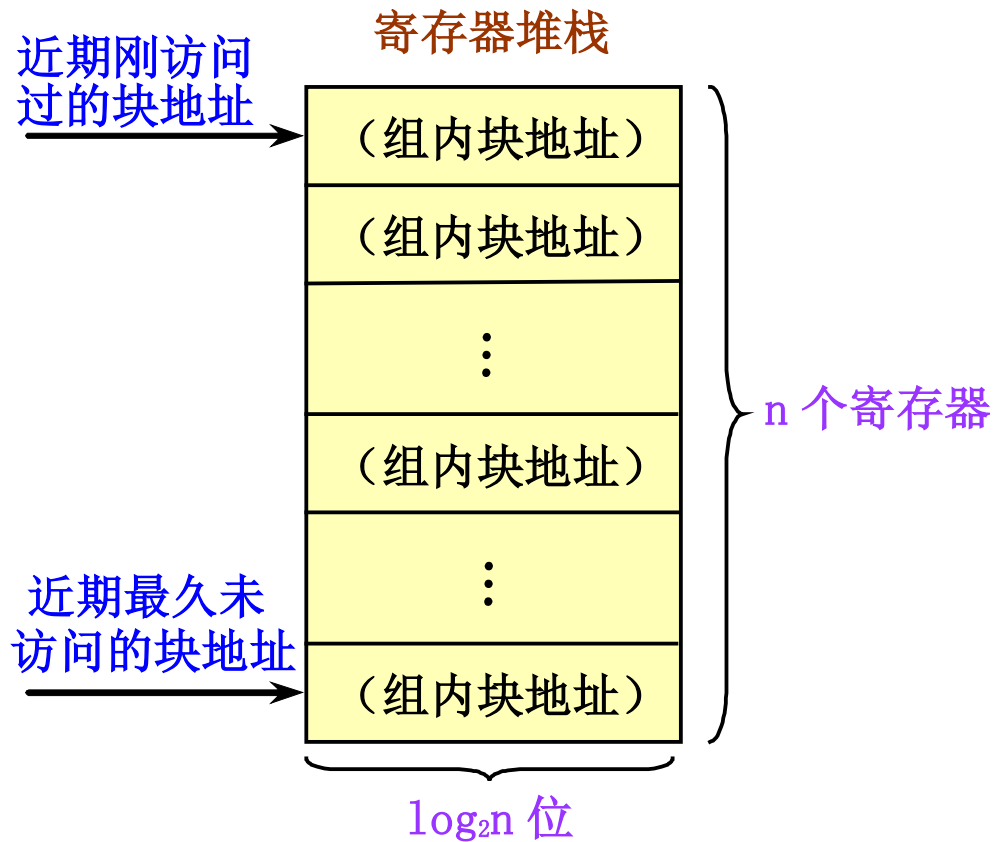
7.2 Cache基本知识

2. LRU算法的硬件实现

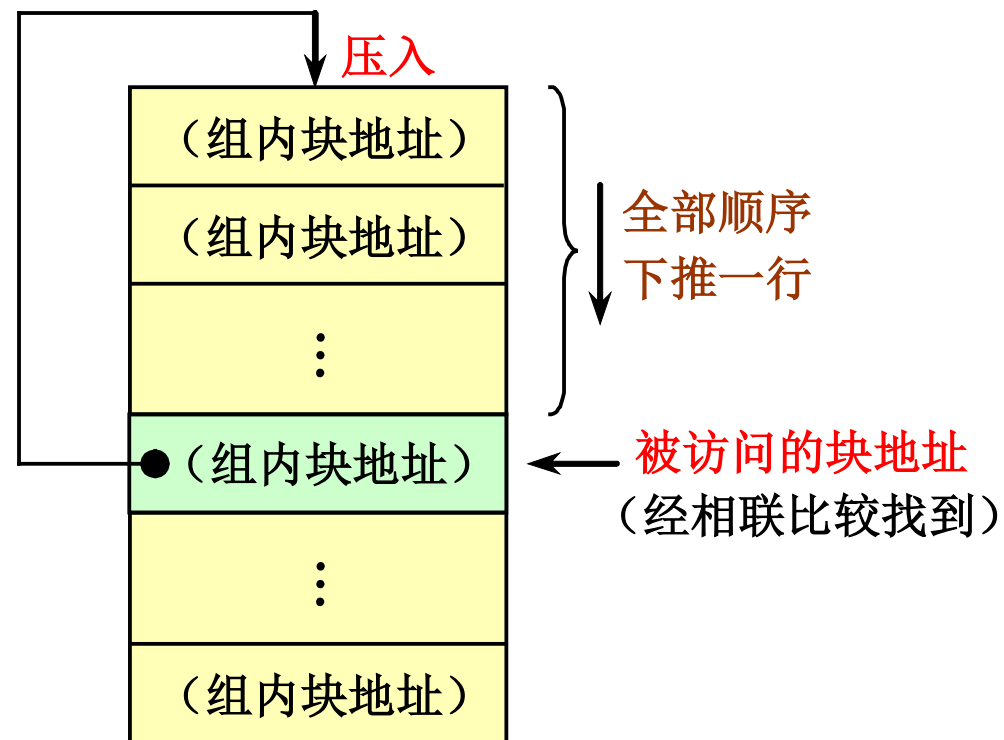
➤ 堆栈法

- 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）

7.2 Cache基本知识



(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

7.2 Cache基本知识

- 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

7.2 Cache基本知识

- 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)

缓存操作	初始状态	调入2	命中块4	替换块1
寄存器0	3	2	4	5
寄存器1	4	3	2	4
寄存器2	1	4	3	2
寄存器3	空	1	1	3

7.2 Cache基本知识

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”在访问数据Cache操作中所占的比例：

$$9\% / (26\% + 9\%) \approx 25\%$$

占比例不大，影响力极强

7.2 Cache基本知识

2. “写”操作必须在确认是命中后才可进行（读可以并行）

3. “写”访问有可能导致Cache和主存内容的不一致

4. 两种写策略 → 何时更新？

➤ **写直达法**（也称为存直达法、写穿，write through）

□ 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，write back）

□ 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”）

写策略是区分不同Cache设计方案的一个**重要标志**

7.2 Cache基本知识

5. 两种写策略的比较

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

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

- 减少写停顿的一种常用的优化技术：
 - 采用写缓冲器

7.2 Cache基本知识

7. “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

8. 写策略与调块

- **写回法 —— 按写分配**

- **写直达法 —— 不按写分配**

实例分析1

(2018) 矩阵行列转换是信号处理与科学计算中常见的操作。有如下矩阵行列转换C语言程序代码：

```
void transpose (int dst[2][2], int src[2][2])
{
    int i, j;
    for(i = 0; i < 2; i++)
        for(j = 0; j < 2; j++)
            dst[j][i] = src[i][j];
}
```

该代码运行在具有如下配置参数的计算机上：

整型数为4字节（sizeof（int） = 4）；

源数组src在内存中存储起始地址为0，目的数组dst起始地址为16（十进制）；CPU具仅有一级数据cache（不考虑取指令访存），采用直接相联的映射方式（direct-mapped），写穿（write-through）且写分配（write-allocate）策略，cache容量为16字节，块大小为8字节；初始cache内容为空；

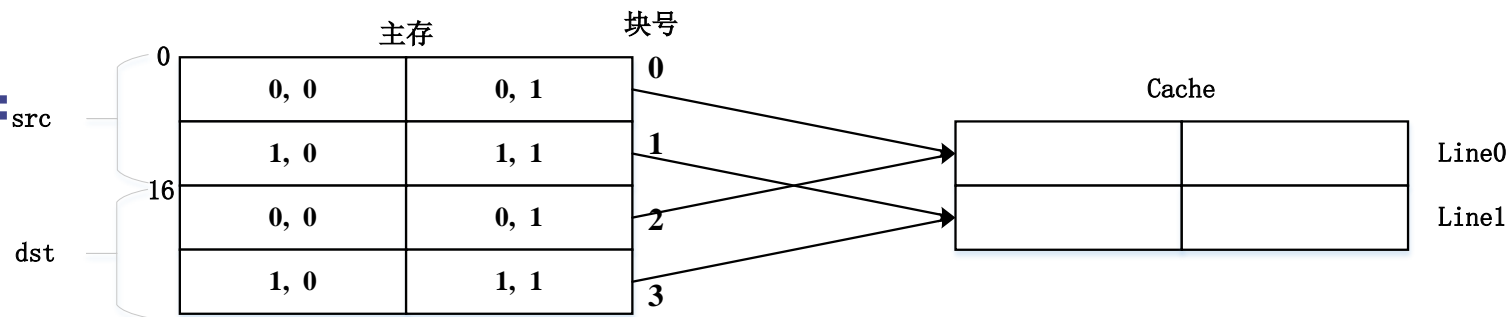
假设程序运行期间仅有读/写src、dst数组产生数据访问。

试计算第一次运行该矩阵转换程序数据cache的命中率。若cache容量增大到32字节则命中率又是多少？

实例分析1

解：

主存与cache映射关系如下图：



数据访问顺序：

scr[0][0], dst[0][0], scr[0][1], dst[1][0], scr[1][0], dst[0][1], scr[1][1], dst[1][1]

	scr[0][0]	dst[0][0]	scr[0][1]	dst[1][0]	scr[1][0]	dst[0][1]	scr[1][1]	dst[1][1]
Cache/tag	Line0/0	Line0/2	Line0/0	Line1/3	Line1/1	Line0/2	Line1/1	Line1/3
Miss/Hit	miss	miss	miss	miss	miss	miss	hit	miss

故命中率 = $1/8 = 12.5\%$

若cache增大到32 bytes，访问的数据恰好全部可装入cache中，首次访问一个cache line时不命中，第二次访问时命中，故命中率为50%。

实例分析2

(2019) 设有一个“cache-主存”二级存储器，主存共分8个块（地址编号0~7），cache为4个块（0~3），采用2路组相联映像，替换算法为先进先出（FIFO）。

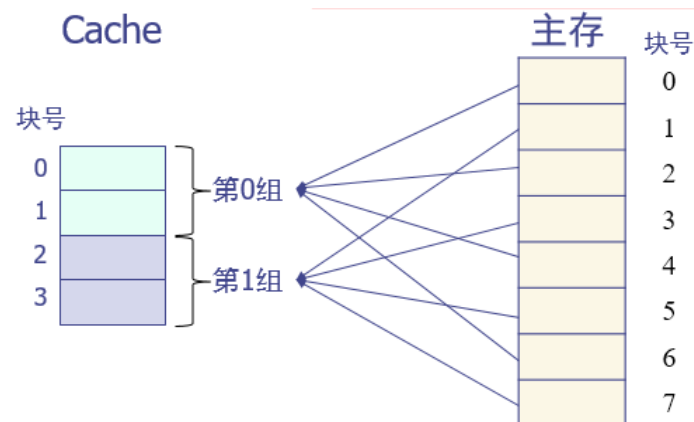
1. 画出主存、cache空间块的映像对应关系示意图；
2. 对于主存访问块地址流：1, 2, 4, 1, 3, 7, 0, 1, 2, 5, 4, 6, 4, 7, 2，如果cache初始内容为空，请列出cache中各块随时间的使用状况，在表中标出当前访问结束时各cache块中缓存的数据对应的主存块号（空闲cache块不用填），并用“*”标出备选淘汰块；
3. 对于（2），计算此期间cache的命中率。

访问系列:	1	2	4	1	3	7	0	1	2	5	4	6	4	7	2
Cache 块号															
0															
1															
2															
3															

实例分析2

解：

1、主存、cache块的映像对应关系如图所示：



2、对于给定主存访问地址流，cache块使用情况如下图：

时间系号：1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

地址流：1, 2, 4, 1, 3, 7, 0, 1, 2, 5, 4, 6, 4, 7, 2

	2*	2*	2*	2*	2*	0	0	0*	0*	4	4*	4*	4*	2
		4	4	4	4	4*	4*	2	2	2*	6	6	6	6*
1*	1*	1*	1*	1*	7	7	7*	7*	5	5	5	5	5*	5*
				3	3*	3*	1	1	1*	1*	1*	1*	7	7

命中情况：M M M H M M M M M M M H M M

3、依上图命中情况统计，15次访问命中2次，命中率为： $2/15 = 13.3\%$ 。

7.2 Cache基本知识

7.2.7 Cache的性能分析

1. 不命中率

直觉，大部分时间很准

- 与硬件速度无关
- 容易产生一些误导

2. 平均访存时间

客观，根本评价标准

平均访存时间 = 命中时间 + 不命中率 × 不命中开销

7.2 Cache基本知识

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

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

7.2 Cache基本知识

例7.1 某计算机，假设Cache不命中开销为50个时钟周期，当不考虑存储器停顿，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：

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

7.2 Cache基本知识

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 (理想CPI为2.0)

但若不采用Cache, 则：

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

7.2 Cache基本知识

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%

7.2 Cache基本知识

解： 平均访存时间为：

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

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。

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

7.2 Cache基本知识

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 Cache基本知识

为什么Cache的性能如此重要？



7.2 Cache基本知识

改进访存性能

- **平均访存时间 = 命中时间 + 不命中率 × 不命中开销**
(Hit time) (Miss rate) (Miss penalty)
- 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**

7.3 降低Cache不命中率

7.3.1 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

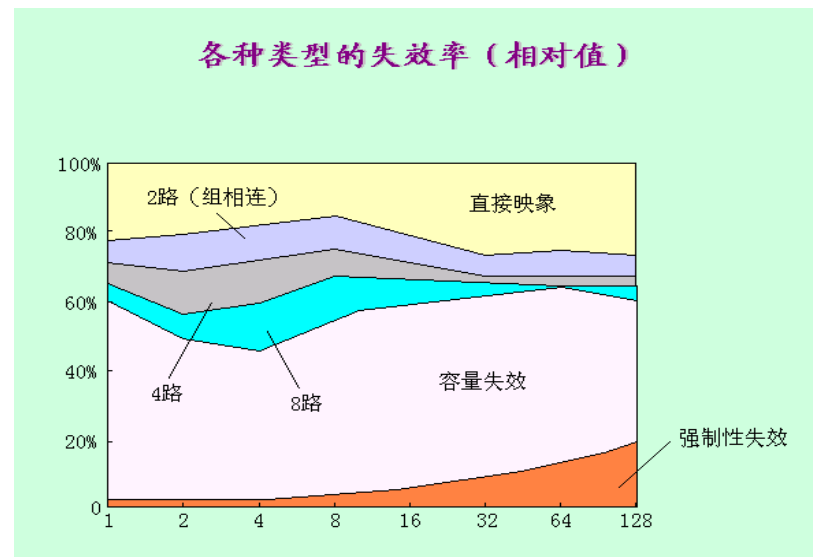
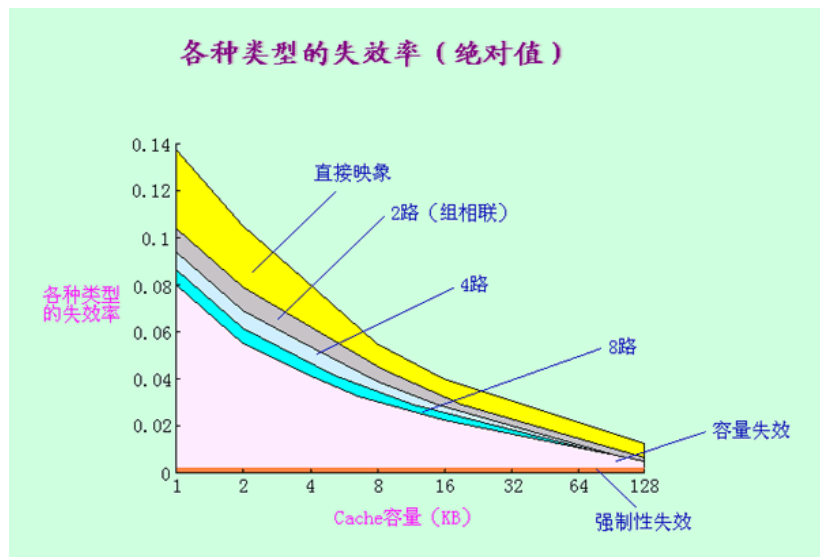
➤ 冲突不命中(Conflict miss)

- 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。称为冲突不命中（碰撞不命中，干扰不命中）

7.3 降低Cache不命中率

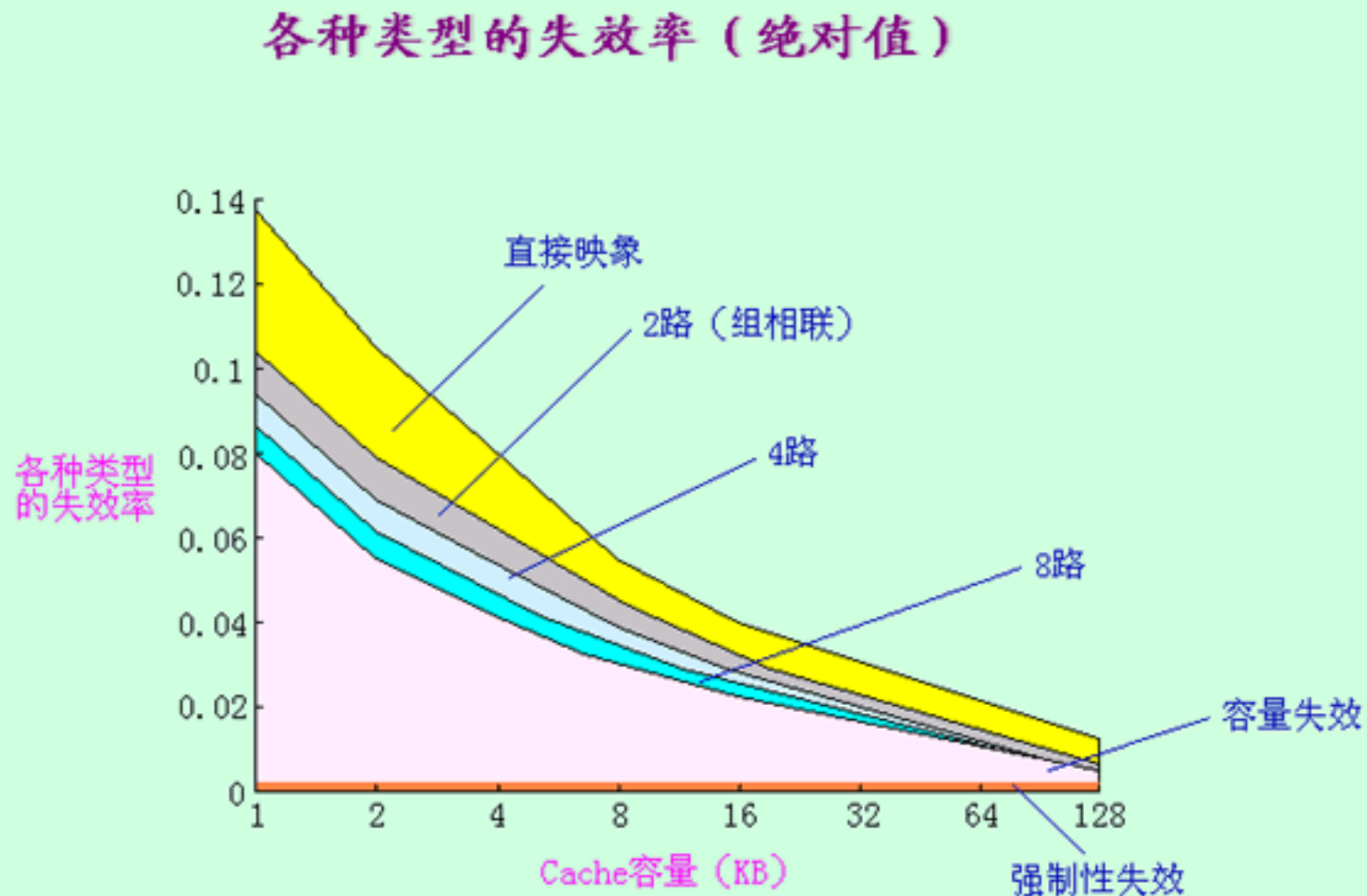
2. 三种不命中所占的比例

- 图示I(绝对值)
- 图示II(相对值)



7.3 降低Cache不命中率

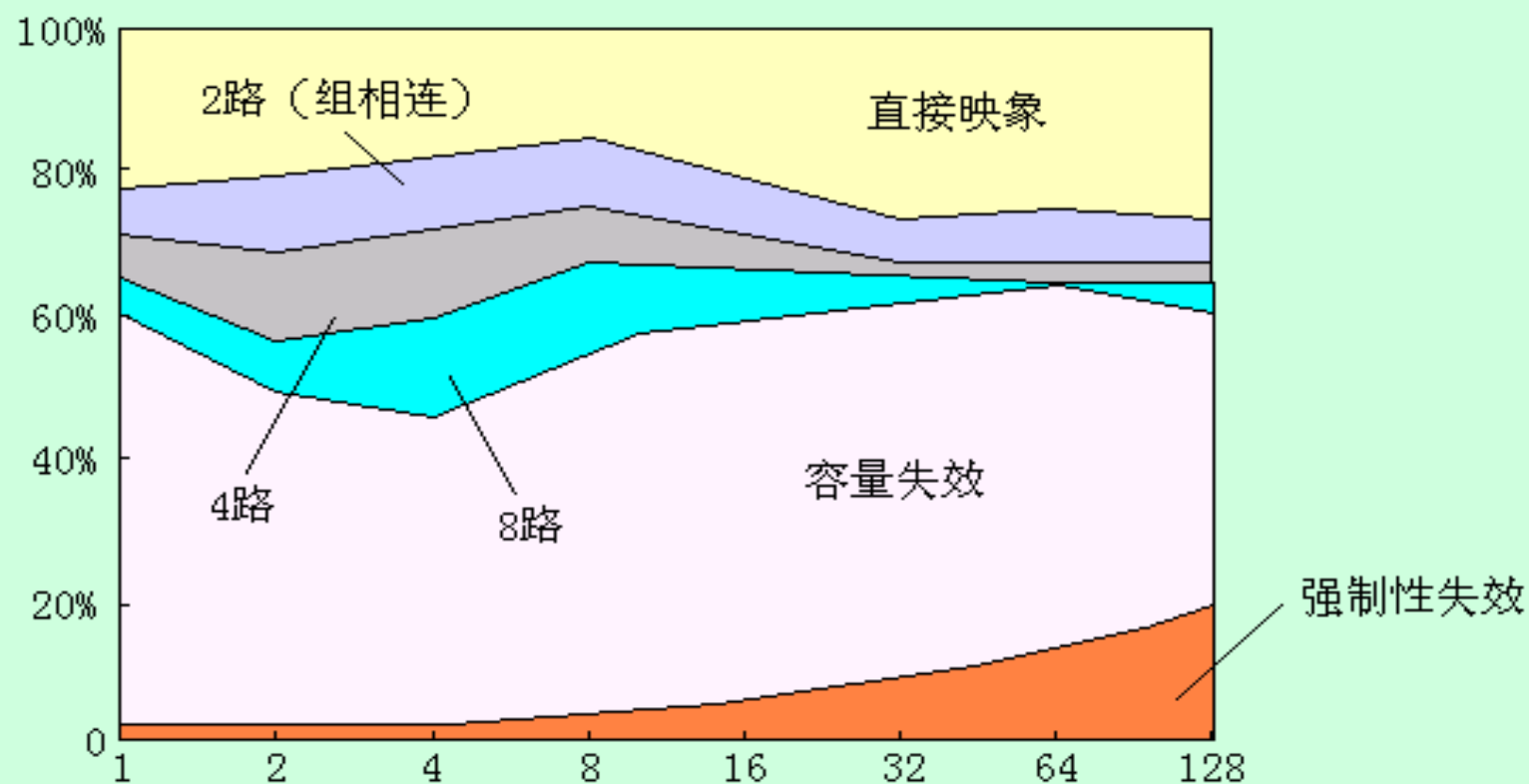
绝对值



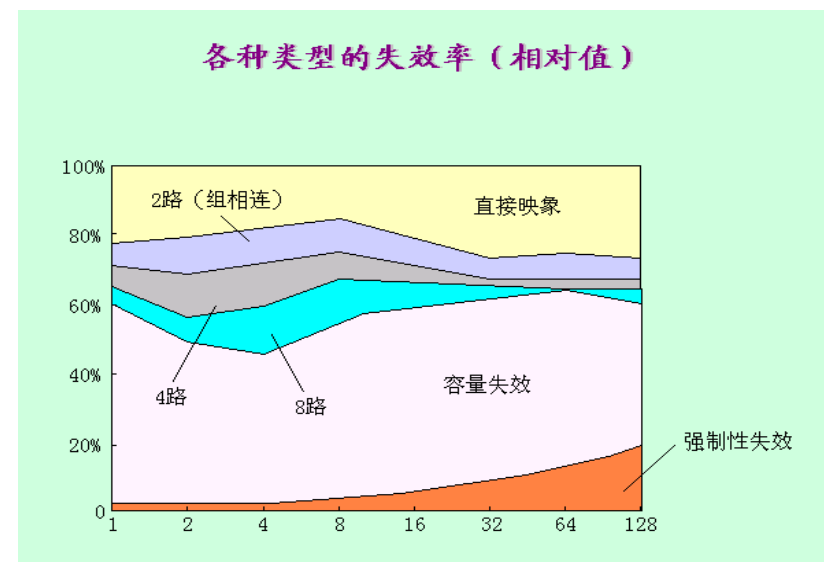
7.3 降低Cache不命中率

相对值

各种类型的失效率（相对值）



- **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**
- **容量不命中随着容量的增加而减少**



7.3 降低Cache不命中率

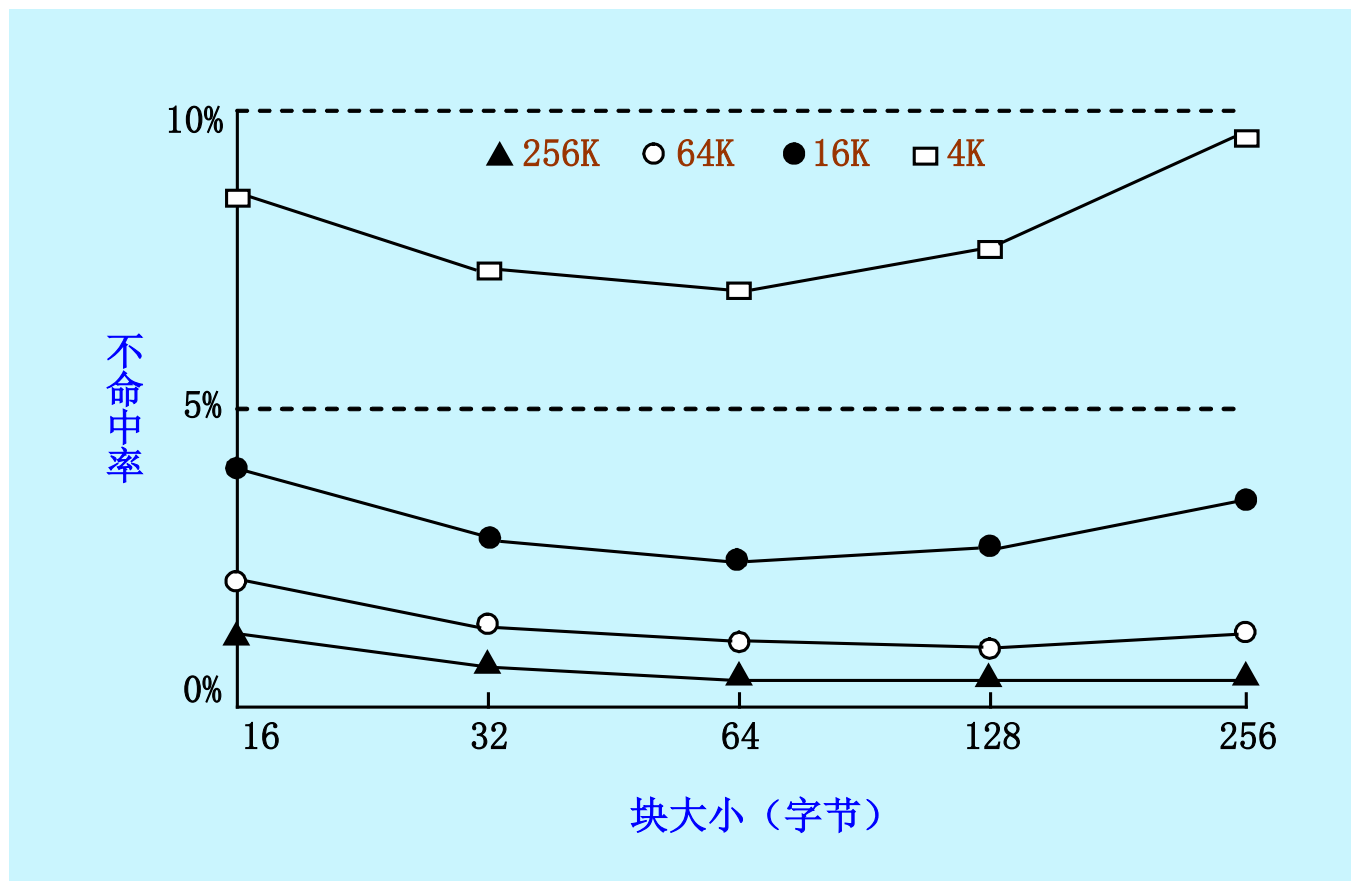
减少三种不命中的方法

- ❑ 强制性不命中：增加块大小，预取
(本身很少)
- ❑ 容量不命中：增加容量
(成本、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 降低Cache不命中率

7.3.2 增加Cache块大小



不命中率随块大小变化的曲线

7.3 降低Cache不命中率

➤ 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 降低Cache不命中率

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：增加冲突，增加不命中开销

7.3 降低Cache不命中率

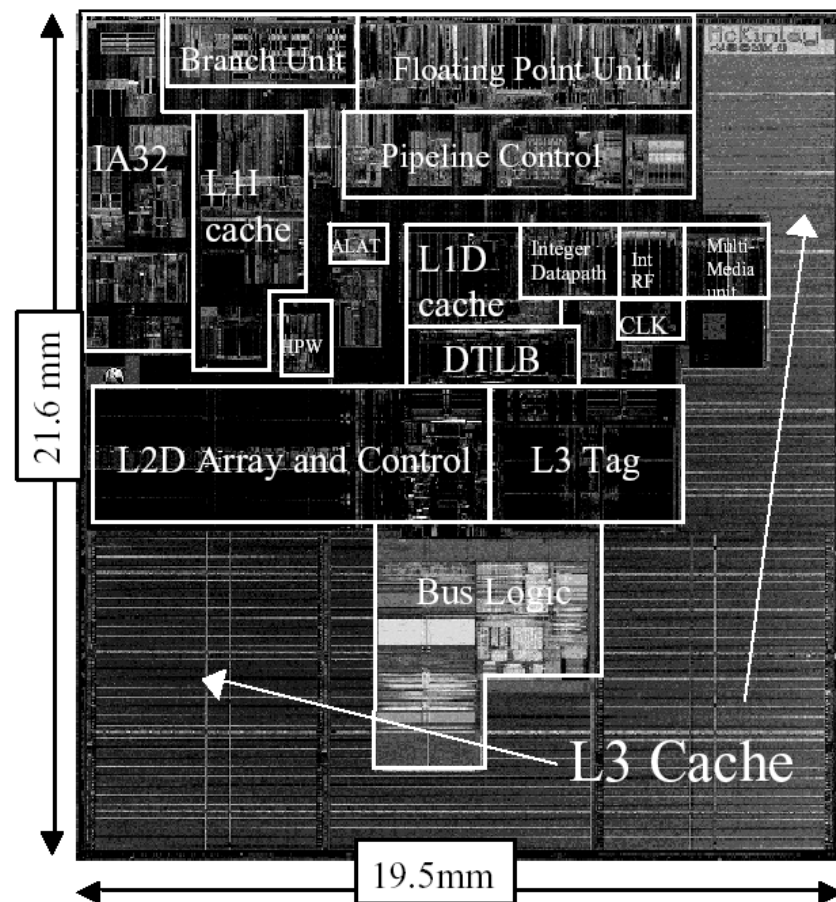
7.3.3 增加Cache的容量

1. 最直接的方法是增加Cache的容量

➤ **缺点:**

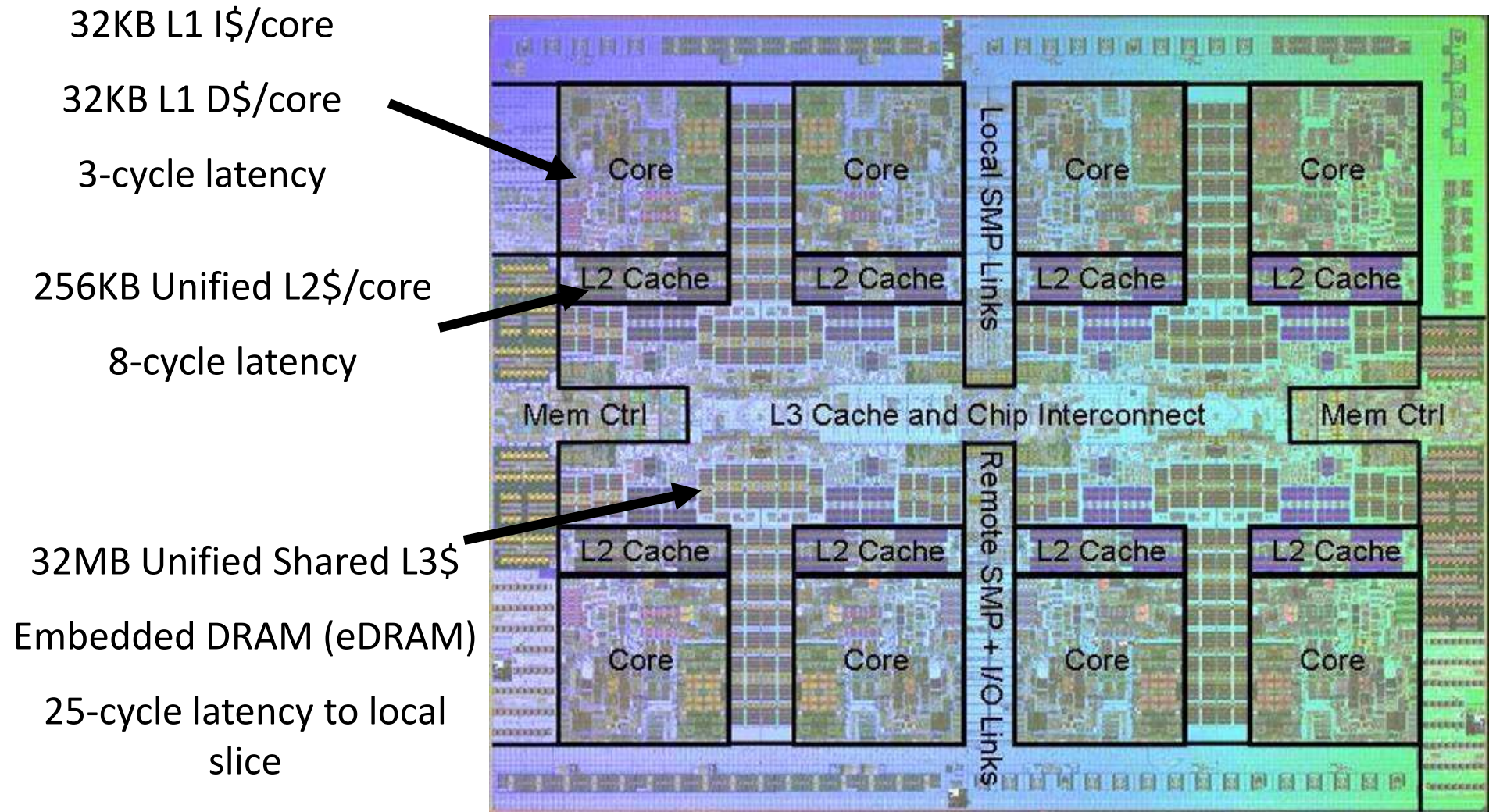
- 增加成本
- 可能增加命中时间

➤ 这种方法在片外Cache中用得比较多



Itanium-2 On-Chip Caches
(Intel/HP, 2002)

Power 7 On-Chip Caches [IBM 2009]





Intel Core i9芯片布局

7.3 降低Cache不命中率

7.3.4 提高相联度

1. 采用相联度超过16的方案的实际意义不大
2. 2:1 Cache经验规则

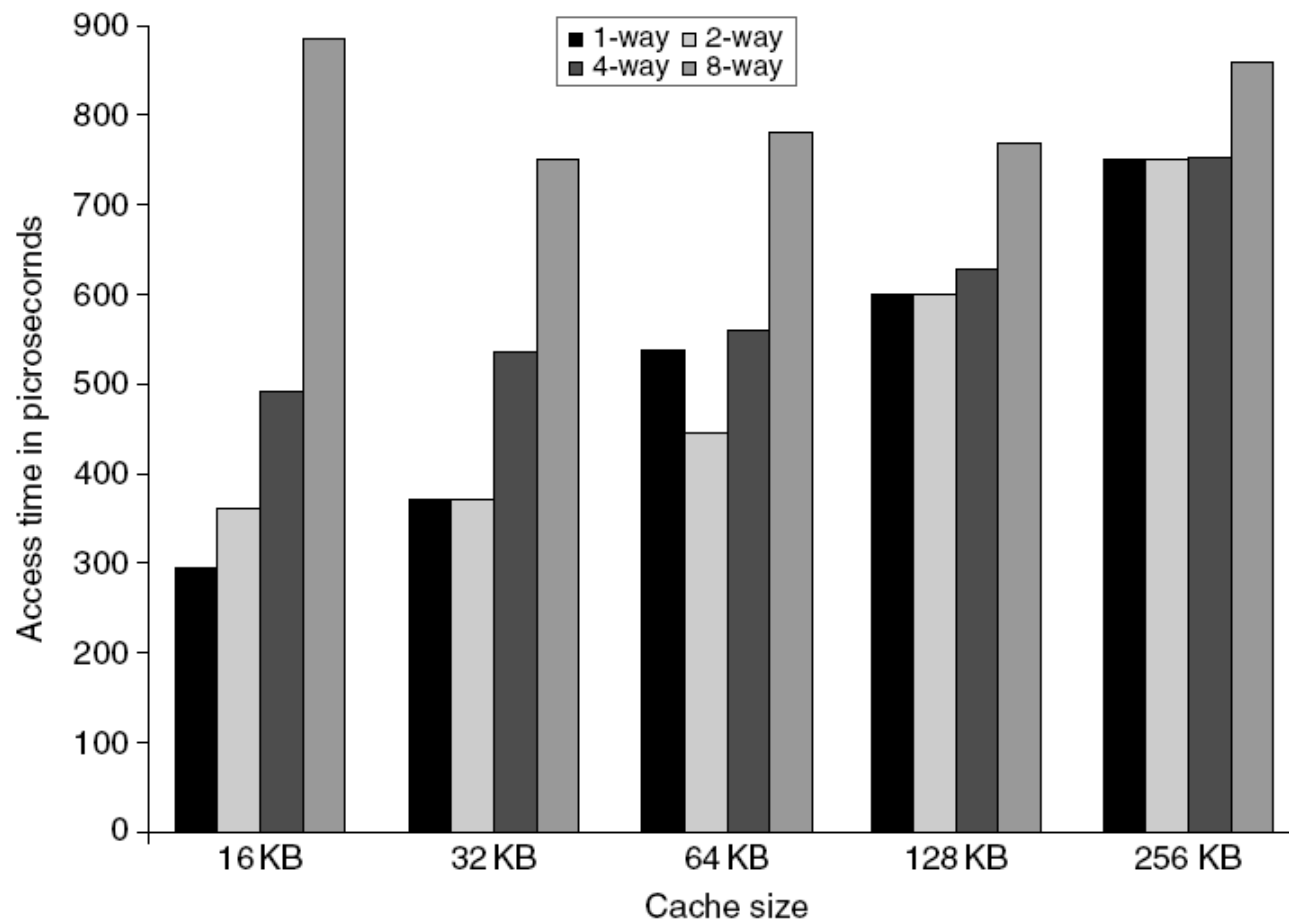
容量为N的直接映象

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

3. 提高相联度是以增加命中时间为代价

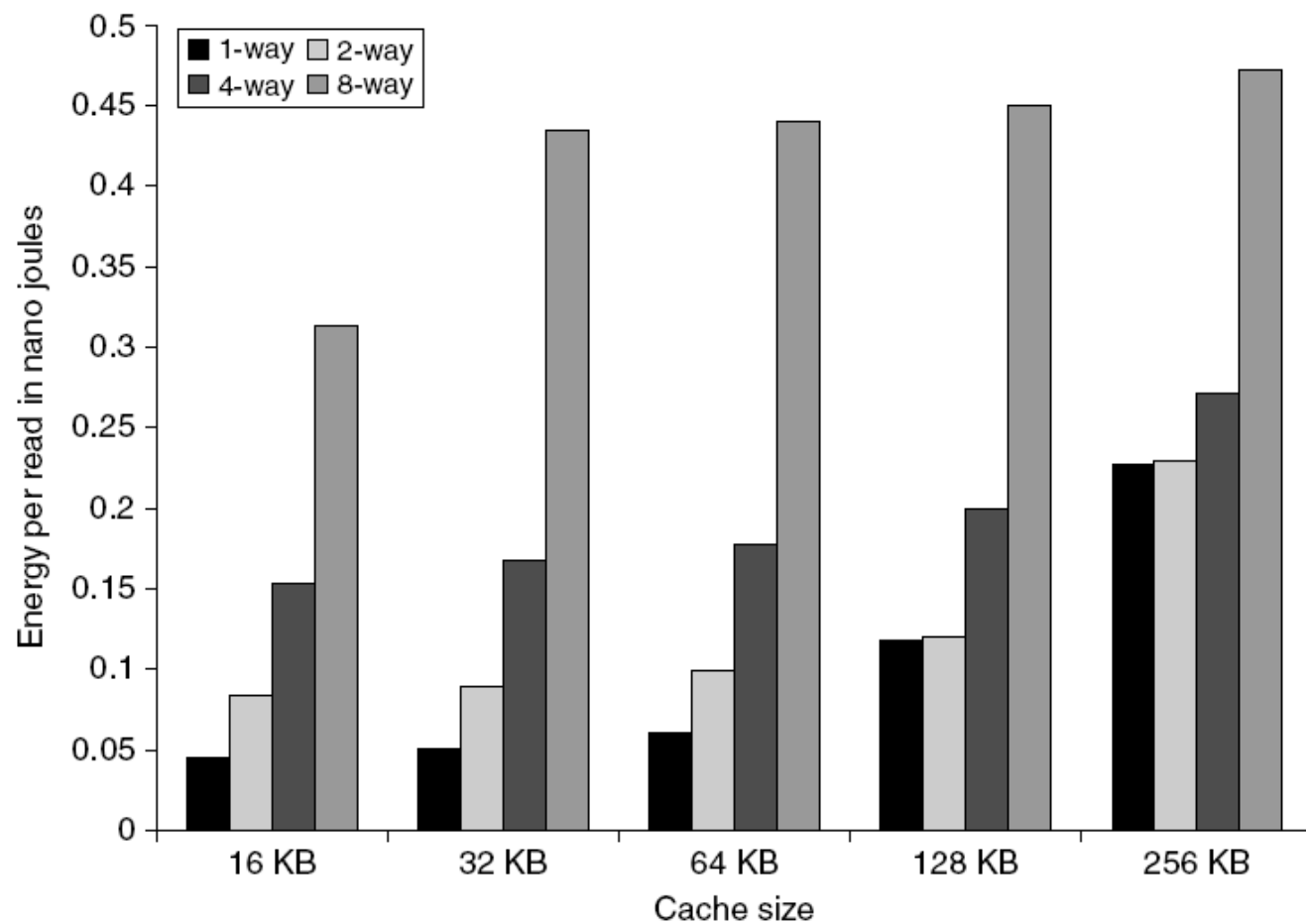
处理器	缓存	主板	内存	SPD	显卡	测试分数	关于																																																				
<div> <div>处理器</div> <table border="1"> <tr> <td>名字</td> <td colspan="3">Intel Core i7 10700</td> <td colspan="4" rowspan="4">  </td> </tr> <tr> <td>代号</td> <td>Comet Lake</td> <td>TDP</td> <td>65.0 W</td> </tr> <tr> <td>插槽</td> <td colspan="3">Socket 1200 LGA</td> </tr> <tr> <td>工艺</td> <td>14 纳米</td> <td>核心电压</td> <td>0.752 V</td> </tr> <tr> <td>规格</td> <td colspan="7">Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz</td> </tr> <tr> <td>系列</td> <td>6</td> <td>型号</td> <td>5</td> <td>步进</td> <td colspan="3">5</td> </tr> <tr> <td>扩展系列</td> <td>6</td> <td>扩展型号</td> <td>A5</td> <td>修订</td> <td colspan="3">Q0/G1</td> </tr> <tr> <td>指令集</td> <td colspan="7">MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3</td> </tr> </table> </div>								名字	Intel Core i7 10700							代号	Comet Lake	TDP	65.0 W	插槽	Socket 1200 LGA			工艺	14 纳米	核心电压	0.752 V	规格	Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz							系列	6	型号	5	步进	5			扩展系列	6	扩展型号	A5	修订	Q0/G1			指令集	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3						
名字	Intel Core i7 10700																																																										
代号	Comet Lake	TDP	65.0 W																																																								
插槽	Socket 1200 LGA																																																										
工艺	14 纳米	核心电压	0.752 V																																																								
规格	Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz																																																										
系列	6	型号	5	步进	5																																																						
扩展系列	6	扩展型号	A5	修订	Q0/G1																																																						
指令集	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3																																																										
<div>时钟 (核心 #0)</div> <table border="1"> <tr> <td>核心速度</td> <td>4589.88 MHz</td> </tr> <tr> <td>倍频</td> <td>x 46.0 (8 - 47)</td> </tr> <tr> <td>总线速度</td> <td>99.78 MHz</td> </tr> <tr> <td>额定 FSB</td> <td></td> </tr> </table>				核心速度	4589.88 MHz	倍频	x 46.0 (8 - 47)	总线速度	99.78 MHz	额定 FSB		<div>缓存</div> <table border="1"> <tr> <td>一级 数据</td> <td>8 x 32 KBytes</td> <td>8-way</td> </tr> <tr> <td>一级 指令</td> <td>8 x 32 KBytes</td> <td>8-way</td> </tr> <tr> <td>二级</td> <td>8 x 256 KBytes</td> <td>4-way</td> </tr> <tr> <td>三级</td> <td>16 MBytes</td> <td>16-way</td> </tr> </table>				一级 数据	8 x 32 KBytes	8-way	一级 指令	8 x 32 KBytes	8-way	二级	8 x 256 KBytes	4-way	三级	16 MBytes	16-way																																
核心速度	4589.88 MHz																																																										
倍频	x 46.0 (8 - 47)																																																										
总线速度	99.78 MHz																																																										
额定 FSB																																																											
一级 数据	8 x 32 KBytes	8-way																																																									
一级 指令	8 x 32 KBytes	8-way																																																									
二级	8 x 256 KBytes	4-way																																																									
三级	16 MBytes	16-way																																																									
<div>已选择 处理器 #1</div> <div>核心数 8</div> <div>线程数 16</div>																																																											

7.3 降低Cache不命中率



Access time vs. **size** and **associativity**

7.3 降低Cache不命中率



Energy per read vs. **size** and **associativity**

7.3 降低Cache不命中率

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

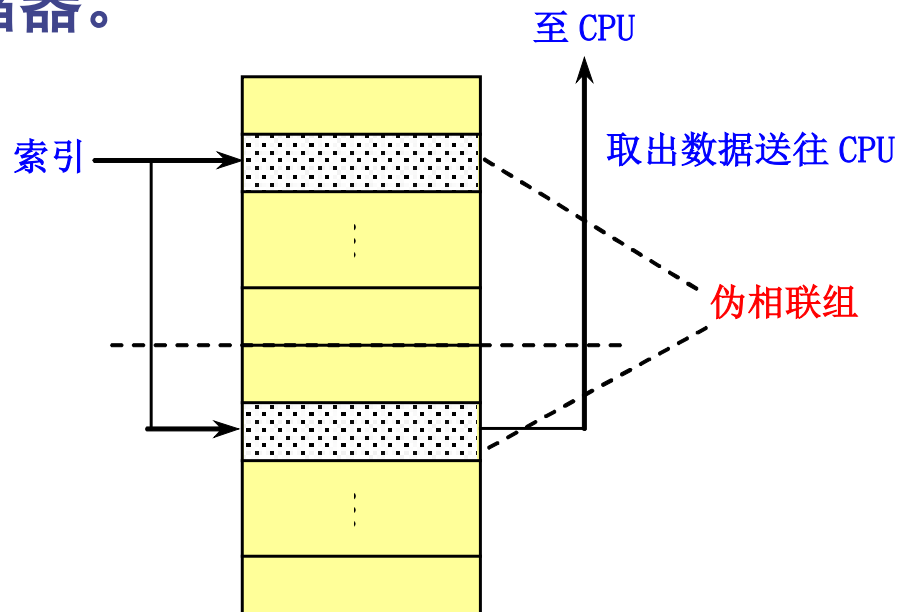
	优 点	缺 点
直接映象	命中时间小	不命中率高
组相联	不命中率低	命中时间长

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

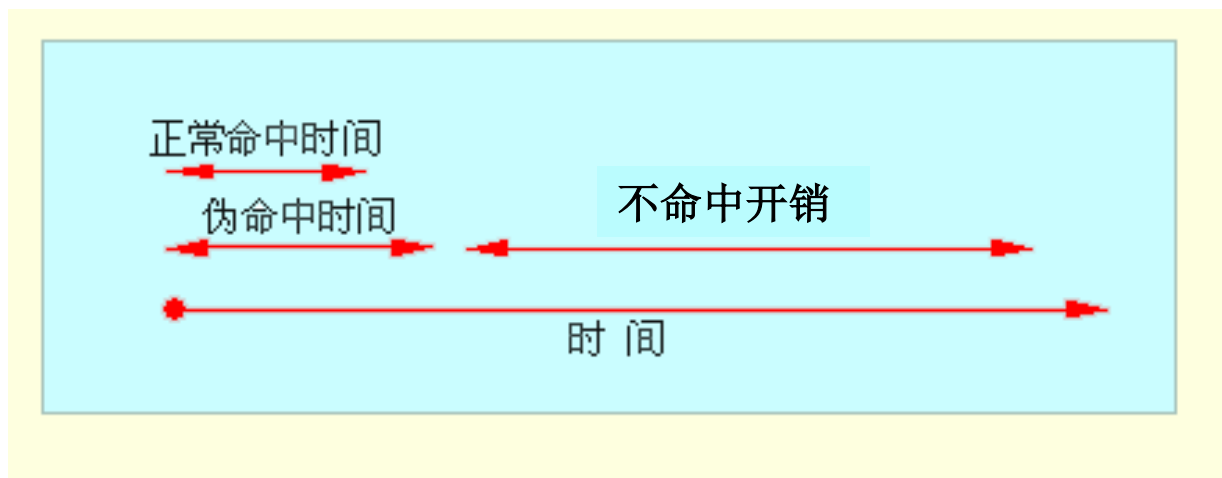
在逻辑上把直接映象Cache的空间上下平分为两个区。对于任何一次访问，伪相联Cache先按直接映象Cache的方式去处理。若命中，则其访问过程与直接映象Cache的情况一样。若不命中，则再到另一区相应的位置去查找。若找到，则发生了伪命中，否则就只好访问下一级存储器。



7.3 降低Cache不命中率

4. 快速命中与慢速命中

要保证绝大多数命中都是快速命中。how?



5. 缺点:

多种命中时间 → CPU 流水线复杂化

例： 在伪相联中，假设在直接映像位置没有发现匹配，而在另一位置找到数据（伪命中）时，不对这两个位置的数据进行交换。这时只需要1个额外的周期。假设不命中的开销为50个周期，2KB的直接映像cache不命中率为9.8%，两路组相联的不命中率为7.6%。试推导伪相联**平均访存时间的公式**，并利用其计算平均访存时间。

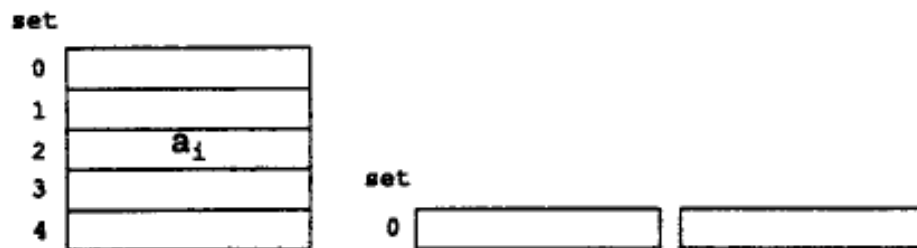
解：平均访存时间 = **命中时间** + 不命中率 × 不命中开销

命中时间 = **命中时间**_{1路} + 1路不中率 × 不命中额外开销

平均访存时间 = **命中时间**_{1路} + 不命中率_{1路} × 1 + **不命中率**_{2路} × 不命中开销

伪相联有效的条件：命中时间_{1路} + 不命中率_{1路} × 1 < 命中时间_{2路}

“伪相联”组中的两块都是用同一个索引得到的，因此失效率相同，
即：失效率_{伪相联} = 失效率_{2路}



Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches

Anant Agarwal and Steven D. Pudar
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

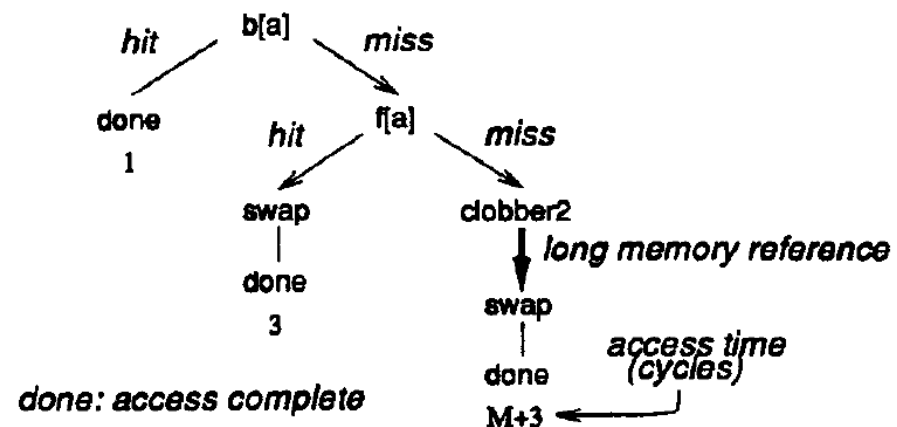
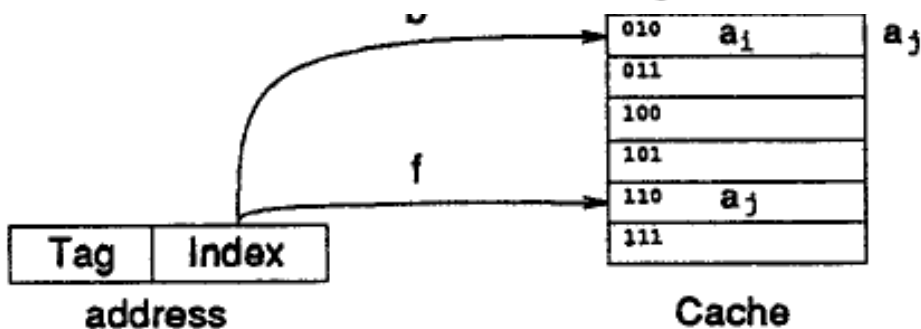


Figure 3: Indexing into a cache by bit selection and by bit flipping.
The conflict $b[a_i] = b[a_j]$ is resolved by the bit-flipping rehash.

7.3 降低Cache不命中率

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

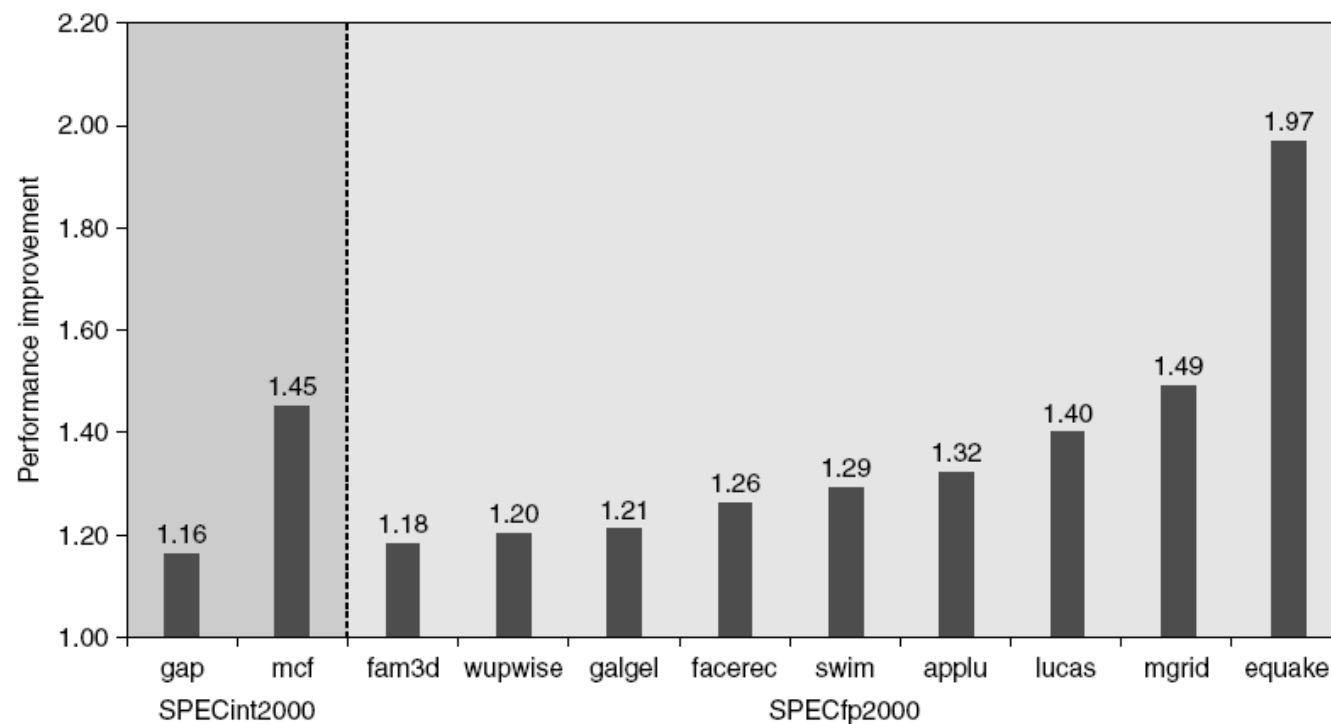
例如：指令流缓冲器

3. 指令预取通常由Cache之外的硬件完成

The Intel Pentium 4 can prefetch data into the second-level cache from up to eight streams from eight different 4 KB pages.

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

Fetch two blocks on miss (include next sequential block)



Pentium 4 Pre-fetching

7.3 降低Cache不命中率

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

本节假定Cache预取都是非故障性的，也叫做**非绑定预取**

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

7.3 降低Cache不命中率

3. 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）

4. 编译器控制预取的目的

使执行指令和读取数据能重叠执行

5. 每次预取需要花费一条指令的开销

- 保证这种开销不超过预取所带来的收益
- 避免不必要的预取（预取谁，何时预取）

7.3 降低Cache不命中率

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
- 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**)

7.3 降低Cache不命中率

- 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

7.3 降低Cache不命中率

2. 修改 C 代码减少cache misses---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 ];
```

7.3 降低Cache不命中率

3. 分块

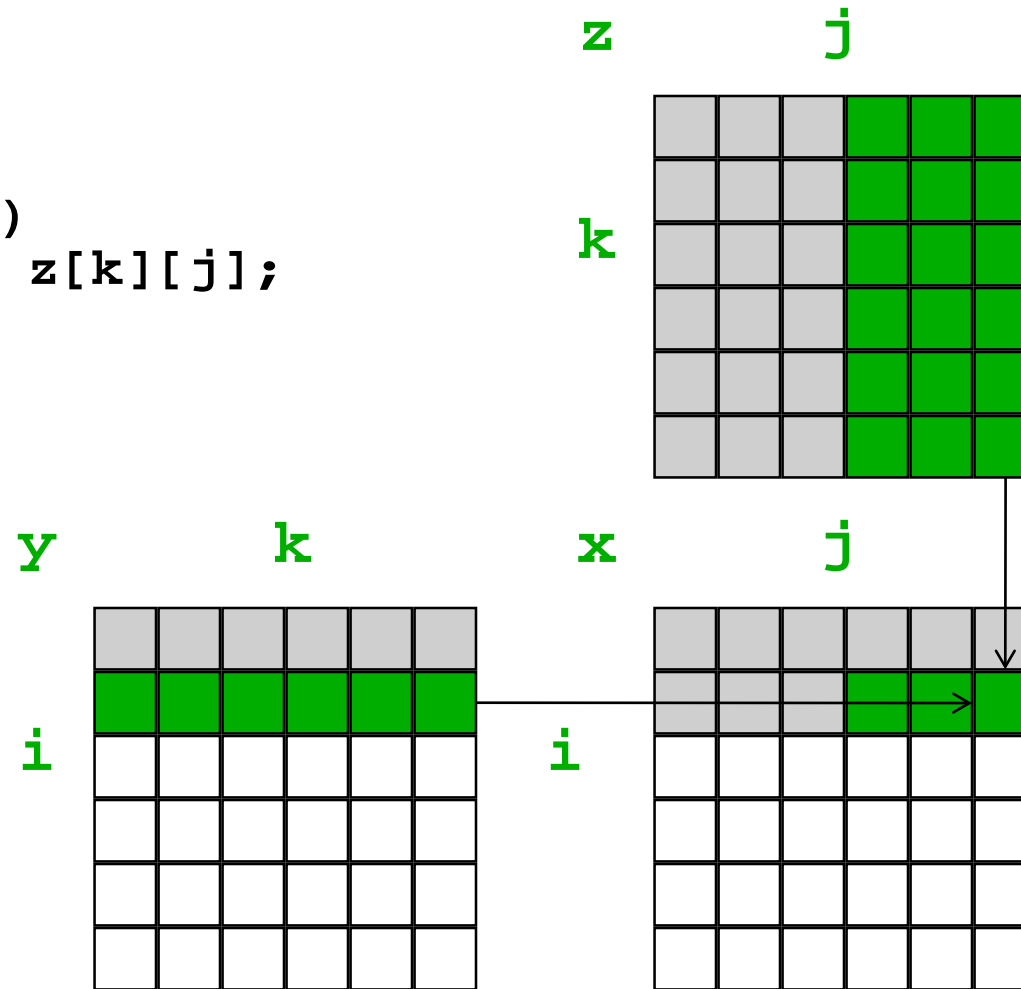
把对数组的整行或整列访问改为按块进行。

/* 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;
    }
```

Matrix Multiply, Naïve Code

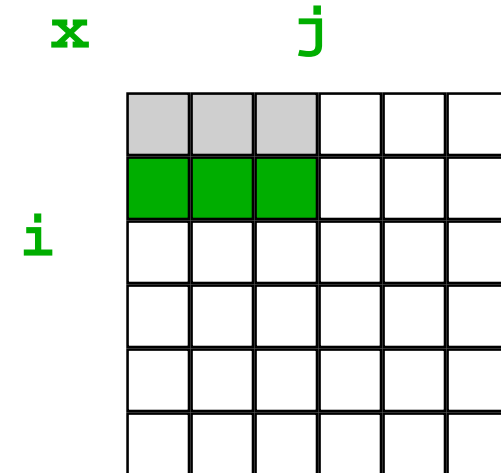
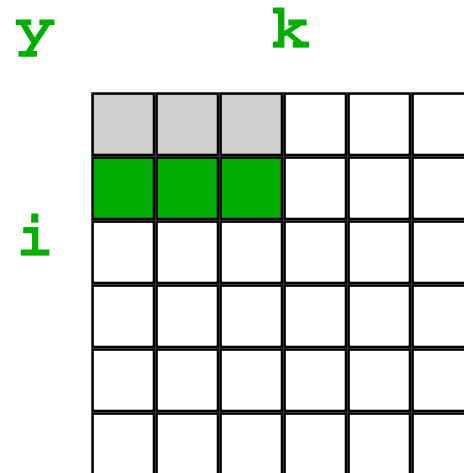
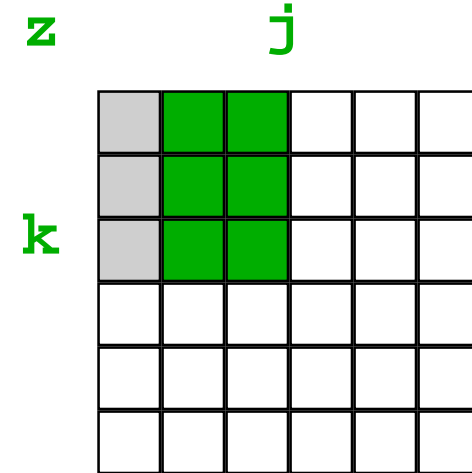
```
for(i=0; i < N; i++)  
  for(j=0; j < N; j++) {  
    r = 0;  
    for(k=0; k < N; k++)  
      r = r + y[i][k] * z[k][j];  
    x[i][j] = r;  
  }
```



□ *Not touched* □ *Old access* ■ *New access*

Matrix Multiply with Cache Tiling

```
for(jj=0; jj < N; jj=jj+B)
  for(kk=0; kk < N; kk=kk+B)
    for(i=0; i < N; i++)
      for(j=jj; j < min(jj+B,N); j++) {
        r = 0;
        for(k=kk; k < min(kk+B,N); k++)
          r = r + y[i][k] * z[k][j];
        x[i][j] = x[i][j] + r;
      }
}
```



What type of locality does this improve?

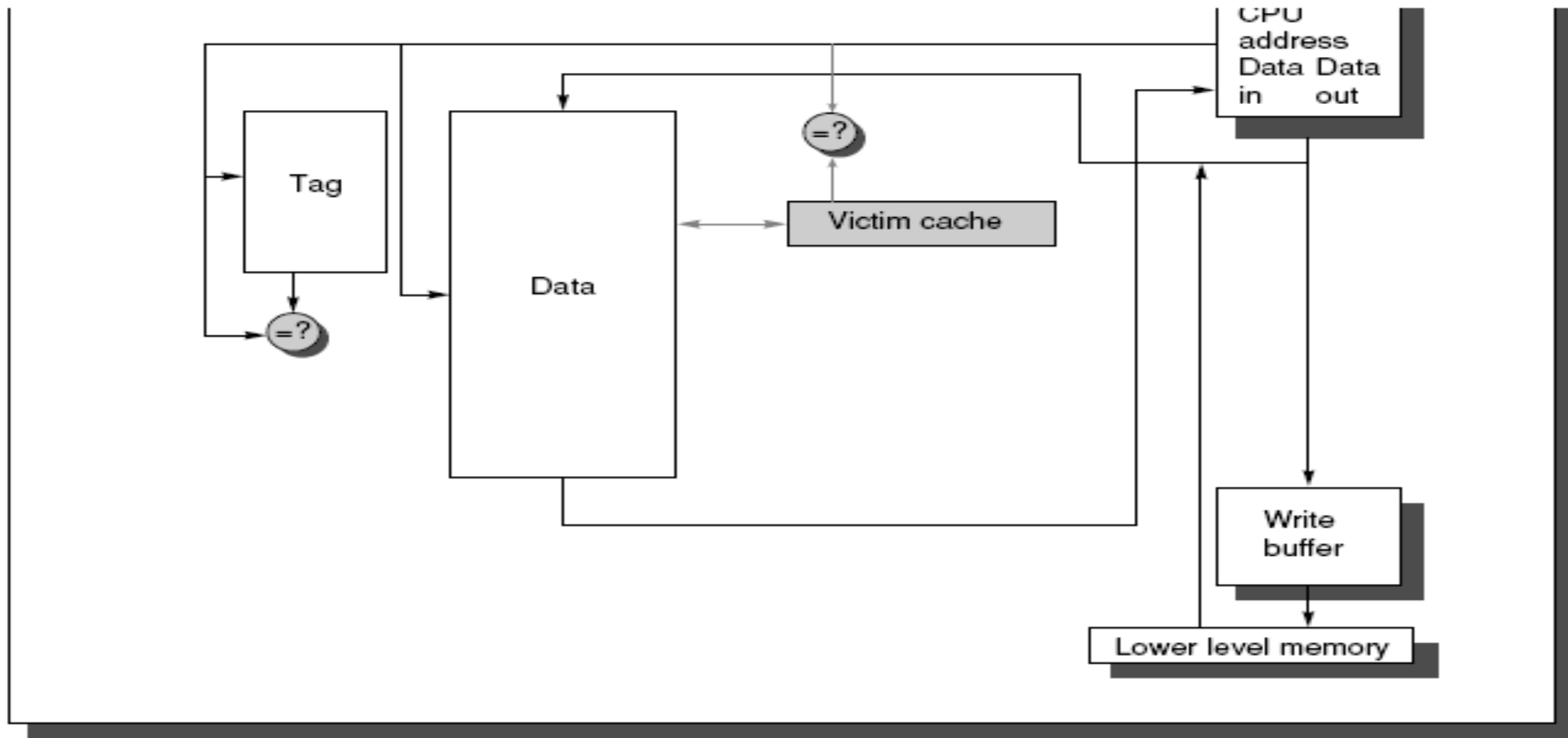
7.3 降低Cache不命中率

7.3.9 “牺牲” Cache

1. 基本思想

- 在Cache和它从下一级存储器调数据的通路之间设置一个全相联的小Cache，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。

7.3 降低Cache不命中率



例：Alpha ACP 21064中，16KB指令cache的不命中率为0.64%，命中时间为1个周期，不命中开销为60个周期。假设采用指令预取技术后，预取命中率为30%。当指令不在指令cache里，而在预取cache中找到时，需要多花1个周期。

问：其实际不命中率为多少？

解： 平均访存时间公式：平均访存时间 = 命中时间 + 不命中率 × 不命中开销
= 直接命中时间 + (不命中率 × 预取命中率 × 1) + (直接不命中率 × 预取不命中率) × 不命中开销
= 1 + (0.64% × 30% × 1) + (0.64% × 70% × 60) = 1.27

由平均访存时间 = 命中时间 + 不命中率 × 不命中开销

得：不命中率 = (平均访存时间 - 命中时间) / 不命中开销
= (1.27 - 1) / 60 = 0.45%

7.4 减少Cache不命中开销 Miss Penalty

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}

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})

7.4 减少Cache不命中开销

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 L_2

- **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

- **全局不命中率 L_2** = 不命中率 L_1 \times 不命中率 L_2

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。

4. 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

$$\begin{aligned} &= \text{每条指令的平均不命中次数}_{L_1} \times \text{命中时间}_{L_2} + \\ &\quad \text{每条指令的平均不命中次数}_{L_2} \times \text{不命中开销}_{L_2} \end{aligned}$$

7.4 减少Cache不命中开销

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%。

7.4 减少Cache不命中开销

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

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：

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

7.4 减少Cache不命中开销

5. 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可能消除容量不命中，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

处理器

缓存

主板

内存

SPD

显卡

测试分数

关于

处理器

名字

Intel Core i7 10700

代号

Comet Lake

TDP

65.0 W

插槽

Socket 1200 LGA

工艺

14 纳米

核心电压

0.752 V

规格

Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz

系列

6

型号

5

步进

5

扩展系列

6

扩展型号


A5

修订

Q0/G1

指令集

MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3



时钟 (核心 #0)

核心速度

4589.88 MHz

倍频

x 46.0 (8 - 47)

总线速度

99.78 MHz

额定 FSB

缓存

一级 数据

8 x 32 KBytes

8-way

一级 指令

8 x 32 KBytes

8-way

二级

8 x 256 KBytes

4-way

三级

16 MBytes

16-way

已选择

处理器 #1

核心数

8

线程数

16

7.4 减少Cache不命中开销

➤ 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 减少Cache不命中开销

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 L_2 = 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 L_2 = 25%
- (4) 对于两路组相联，局部不命中率 L_2 = 20%
- (5) 不命中开销 L_2 = 50个时钟周期

试问第二级Cache的相联度对L1不命中开销的影响如何？

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

7.4 减少Cache不命中开销

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

7.4 减少Cache不命中开销

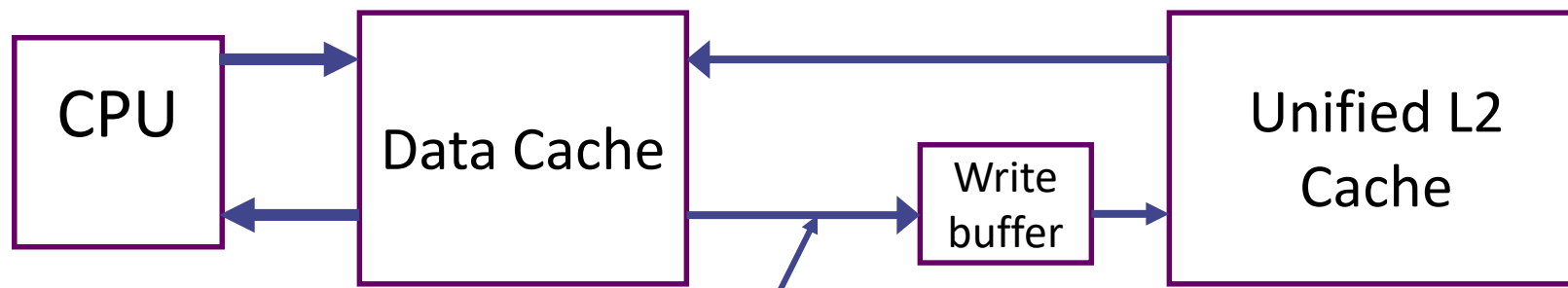
7.4.2 让读不命中优先于写

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

2. 解决问题的方法(读不命中的处理)

- 推迟对读不命中的处理，直至写缓冲器清空（**缺点**：读不命中的开销增加）
- 检查写缓冲器中的内容（没有冲突读继续）



Evicted dirty lines for **writeback cache** OR All writes in **writethrough cache**

7.4 减少Cache不命中开销

7.4.3 写缓冲合并

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

- 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**

7.4 减少Cache不命中开销

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 减少Cache不命中开销

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

3. 局限性在哪？

7.4 减少Cache不命中开销

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”

乱序

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)

7.5 减少命中时间

7.5.1 容量小、结构简单的Cache

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

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，同一核内

某些设计采用了一种折衷方案：

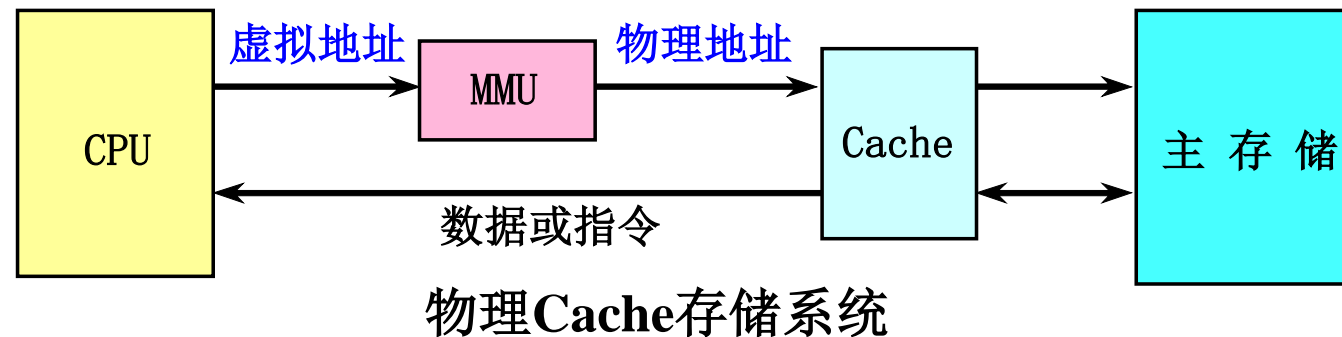
把Cache的标识放在片内，而把Cache的数据存储器放在片外

7.5 减少命中时间

7.5.2 虚拟Cache

1. 物理Cache

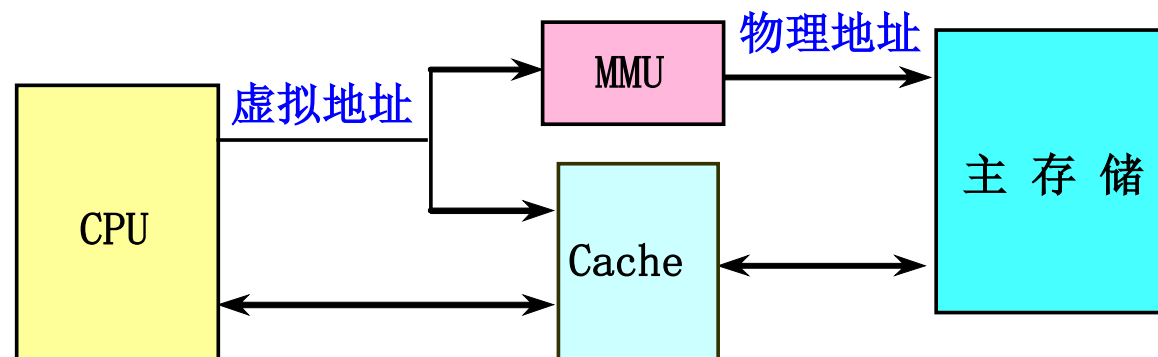
- 使用物理地址进行访问的传统Cache
- 标识存储器中存放的是物理地址，进行地址检测也是用物理地址
- **缺点：地址转换和访问Cache串行进行，访问速度很慢**



7.5 减少命中时间

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



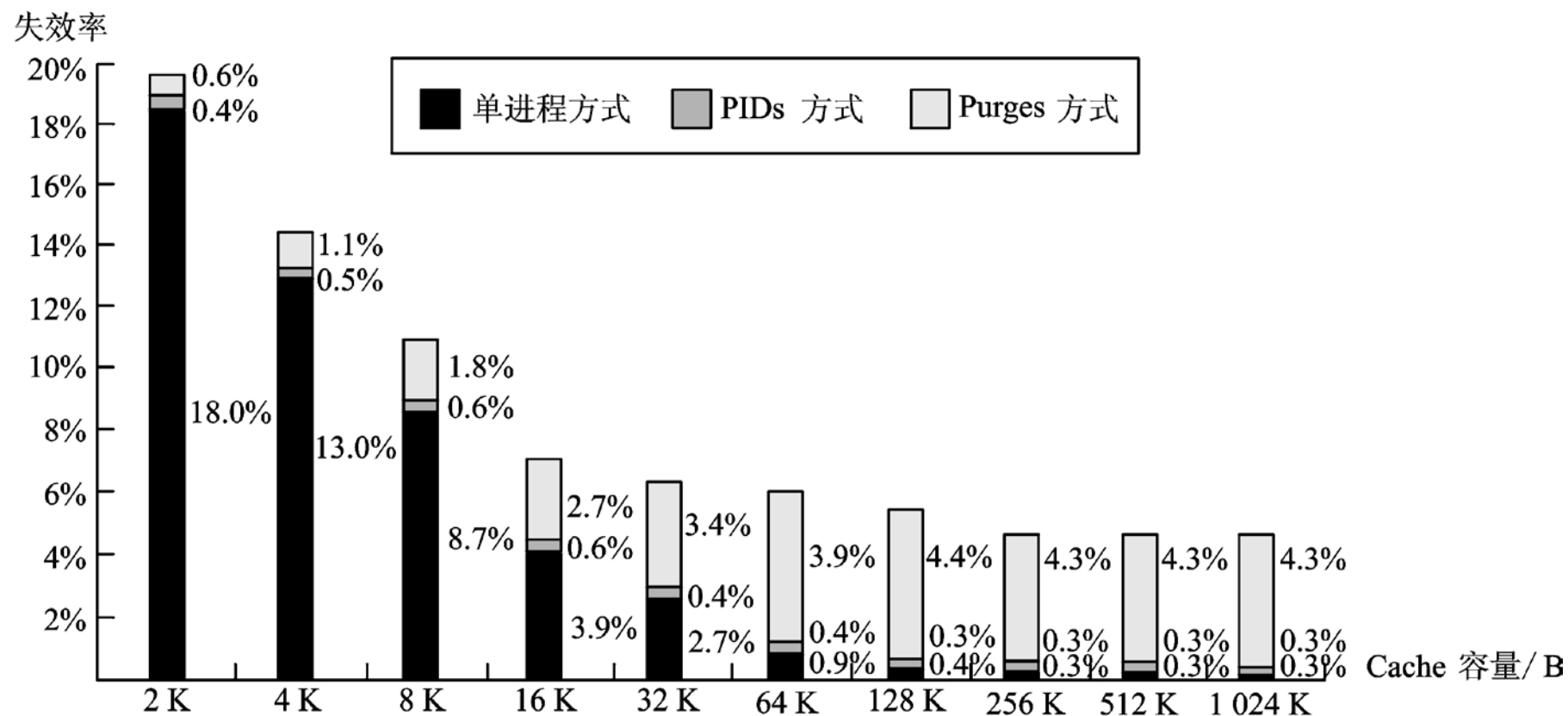
7.5 减少命中时间

3. 并非都采用虚拟Cache (为什么?)

➤ 虚拟Cache的清空问题

- 解决方法：在地址标识中增加PID字段(进程标识符)
- 三种情况下不命中率的比较
 - 单进程, PIDs, 清空
 - PIDs与单进程相比: $+0.3\% \sim +0.6\%$
 - PIDs与清空相比: $-0.6\% \sim -4.3\%$

7.5 减少命中时间



7.5 减少命中时间

4. 虚拟索引 + 物理标识

- **优点：** 兼得虚拟Cache和物理Cache的好处
- **局限性：** Cache容量受到限制

(页内位移)

Cache容量 \leq 页大小 \times 相联度

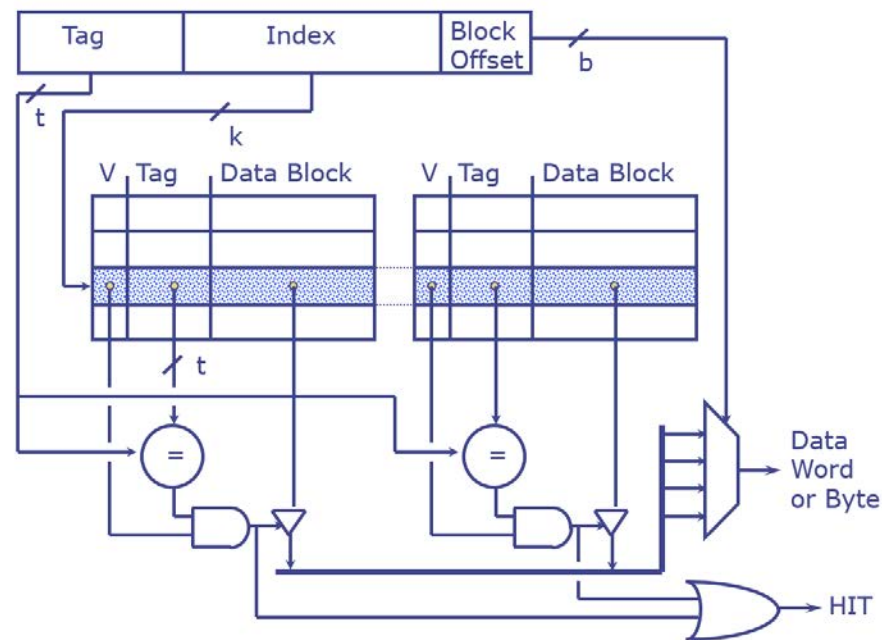
5. 举例：IBM3033的Cache

- 页大小 = 4KB 相联度 = 16



回顾查找算法

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



7.5 减少命中时间

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4个时钟周期

6种基本cache优化方法:

- 增加块大小(Larger block size)
 - 减少强制不命中 (compulsory misses ↓)
 - 增加容量和冲突不命中, 增加不命中开销 (miss penalty ↑)
- 增加cache容量
 - 增加命中时间(hit time ↓), 增加能耗(power consumption ↑)
- 增加相联度
 - 减少冲突不命中(conflict misses ↓)
 - 增加命中时间(hit time ↑), 增加能耗(power consumption ↑)
- 多级cache
 - 降低不命中时间, 优化平均访存时间
- 让读不命中优先于写
 - 降低不命中开销(miss penalty ↓)
- 虚拟Cache
 - 减少命中时间(hit time ↓)

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

(1) 假设在2000次访存中，第一级cache不命中150次，第二级cache不命中30次。试问：在这种情况下，该cache系统的局部不命中率和全局不命中率各是多少？（5分）

(2) 试比较两路组相联与直接映射cache的平均访存时间，假设：
<1>Cache容量为64KB，块大小为32字节；<2>直接映射Cache的CPU时钟周期是1ns，而二路组相联的组织方式会导致CPU时钟周期增加10%；<3>Cache的命中时间为1个时钟周期，失效开销是40ns；<4>直接映像Cache的失效率为2.2%，二路组相联Cache的失效率为1.3%。（5分）

(1) 一级cache全局不命中率=一级cache局部不命中率
 $= 150/2000 = 7.5\%$

二级cache全局不命中率 $= 30/2000 = 1.5\%$

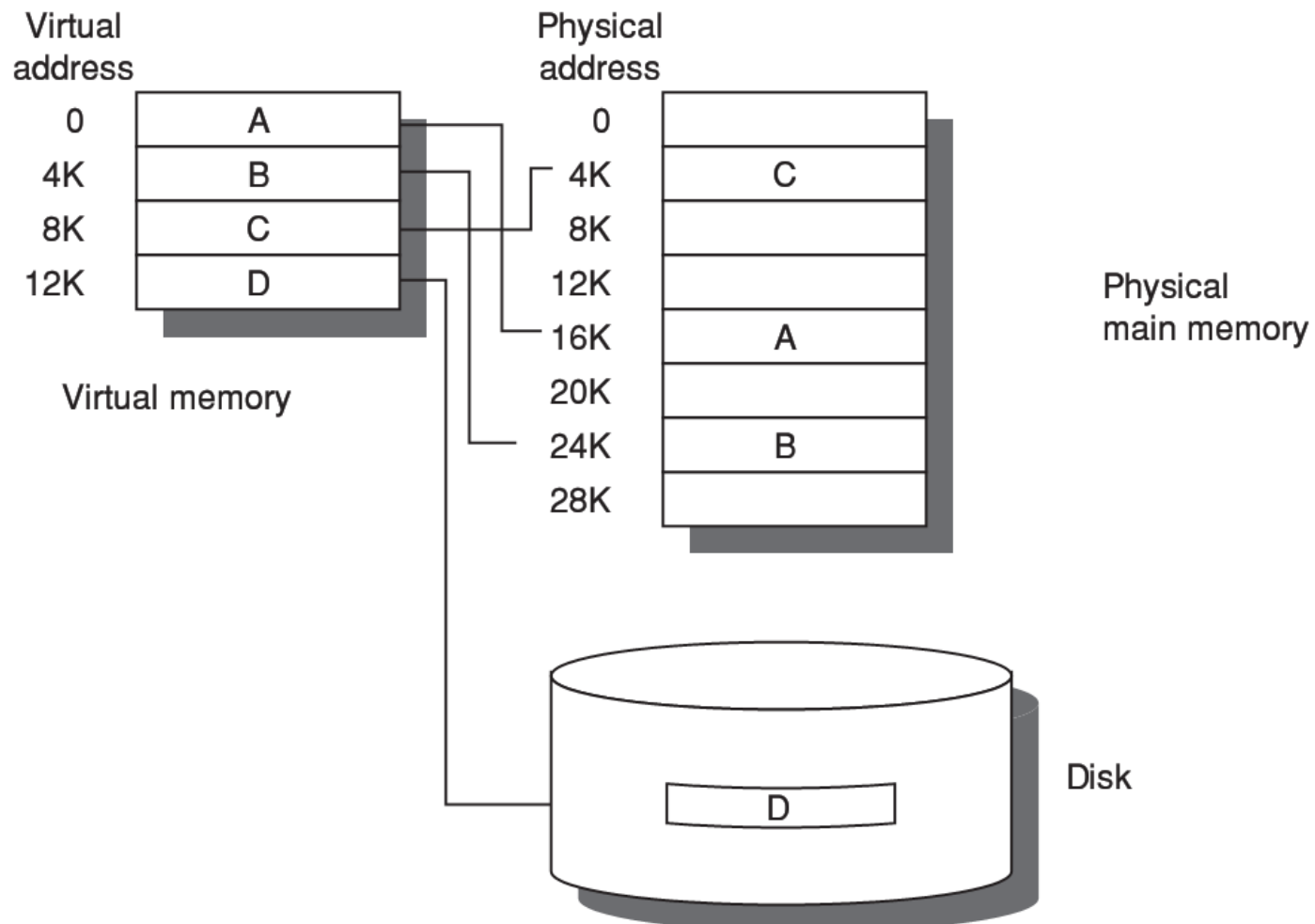
二级cache局部不命中率 $= 30/150 = 20\%$

(2) 平均访存时间一路 = 命中时间一路 + 失效率一路 * 失效开销
 $= 1\text{ns} + 2.2\% * 40\text{ns} = 1.88\text{ns}$

平均访存时间二路 = 命中时间二路 + 失效率二路 * 失效开销 $= 1\text{ns} * (1 + 10\%) \text{ns} + 1.3\% * 40\text{ns} = 1.62\text{ns}$

二路组相联Cache的平均访存时间更短。

7.7 虚拟存储器



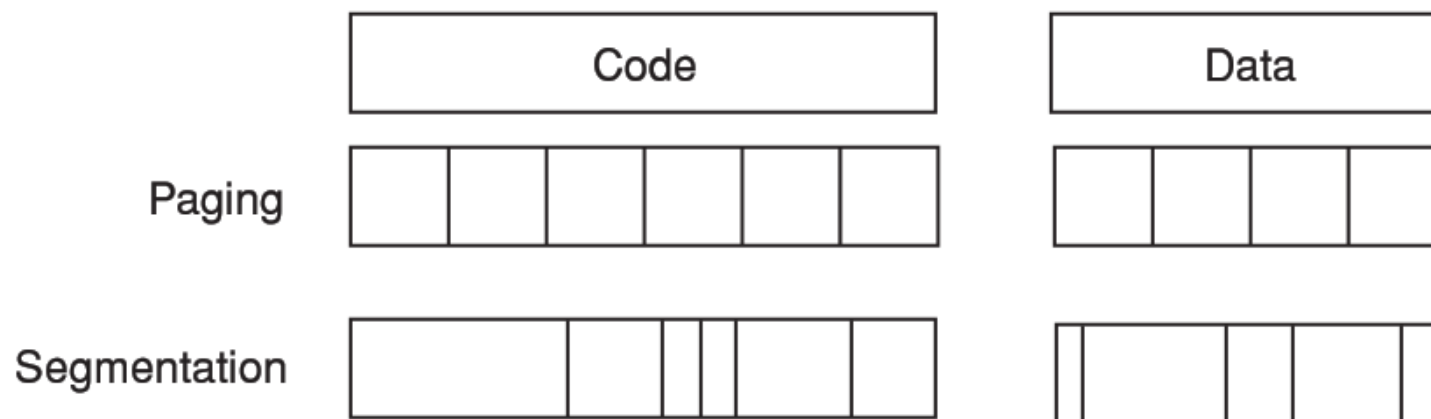
7.7 虚拟存储器

7.7.1 虚拟存储器的基本原理

1. 发明虚拟存储器的目的

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



Example of how paging and segmentation divide a program.

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	4KB-16MB
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

Four questions about any level of the hierarchy (Cache)

Q1: Where can a block be placed in the upper level?

(*Block placement*) 三种映像规则

Q2: How is a block found if it is in the upper level?

(*Block identification*) 目录表

Q3: Which block should be replaced on a miss?

(*Block replacement*) LRU

Q4: What happens on a write?

(*Write strategy*) 写回/写直达

Four questions about any level of the hierarchy (VM)

Q1: Where can a block be placed in the upper level?

(*Block placement*) **Full** (why?)

Q2: How is a block found if it is in the upper level?

(*Block identification*) **Page Table**

Q3: Which block should be replaced on a miss?

(*Block replacement*) **LRU** (how to implement?)

Q4: What happens on a write?

(*Write strategy*) **Write Back**

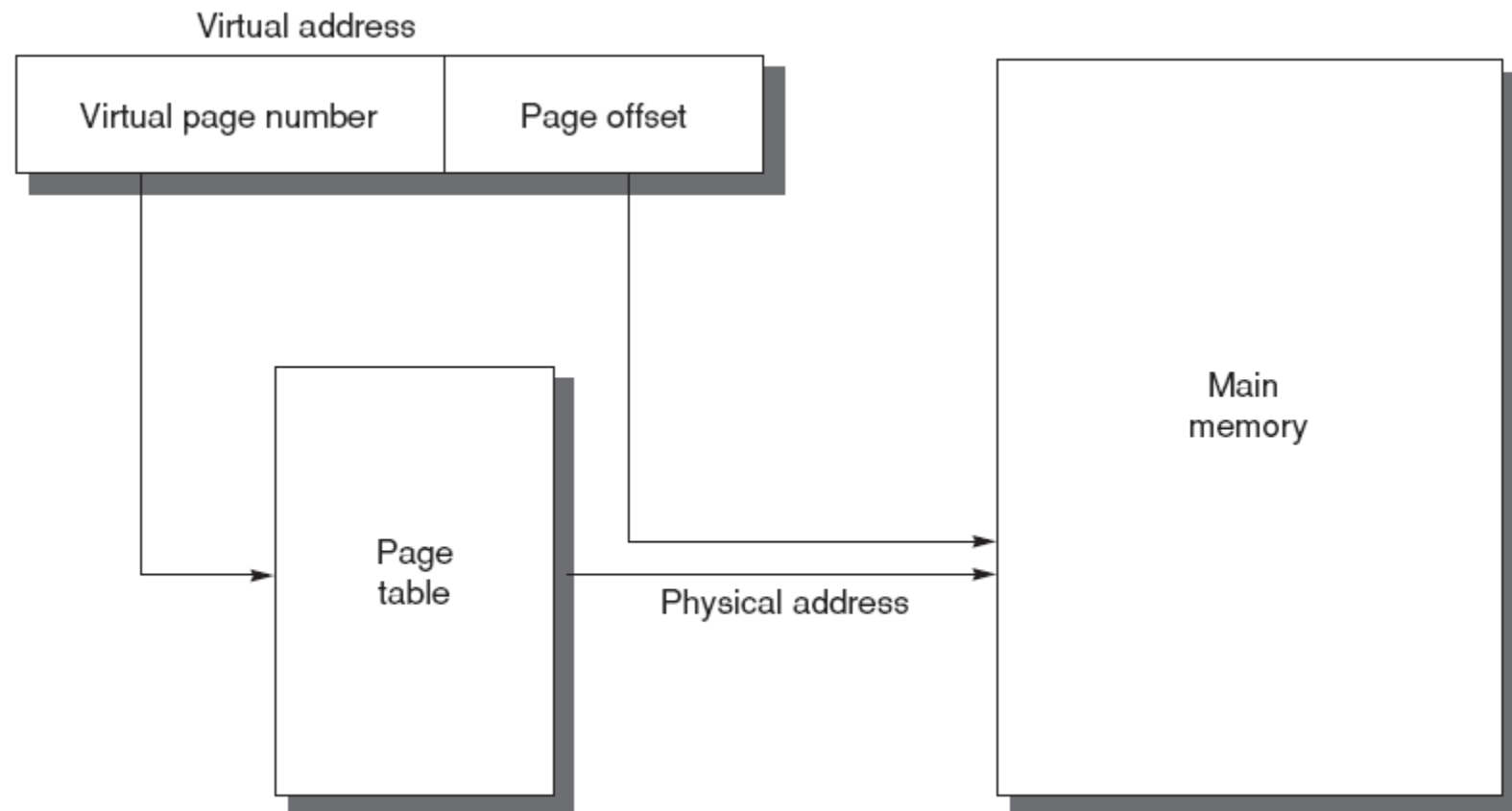


Figure C.22 The mapping of a virtual address to a physical address via a page table.

7.7 虚拟存储器

7.7.2 快速地址转换技术

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

- 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

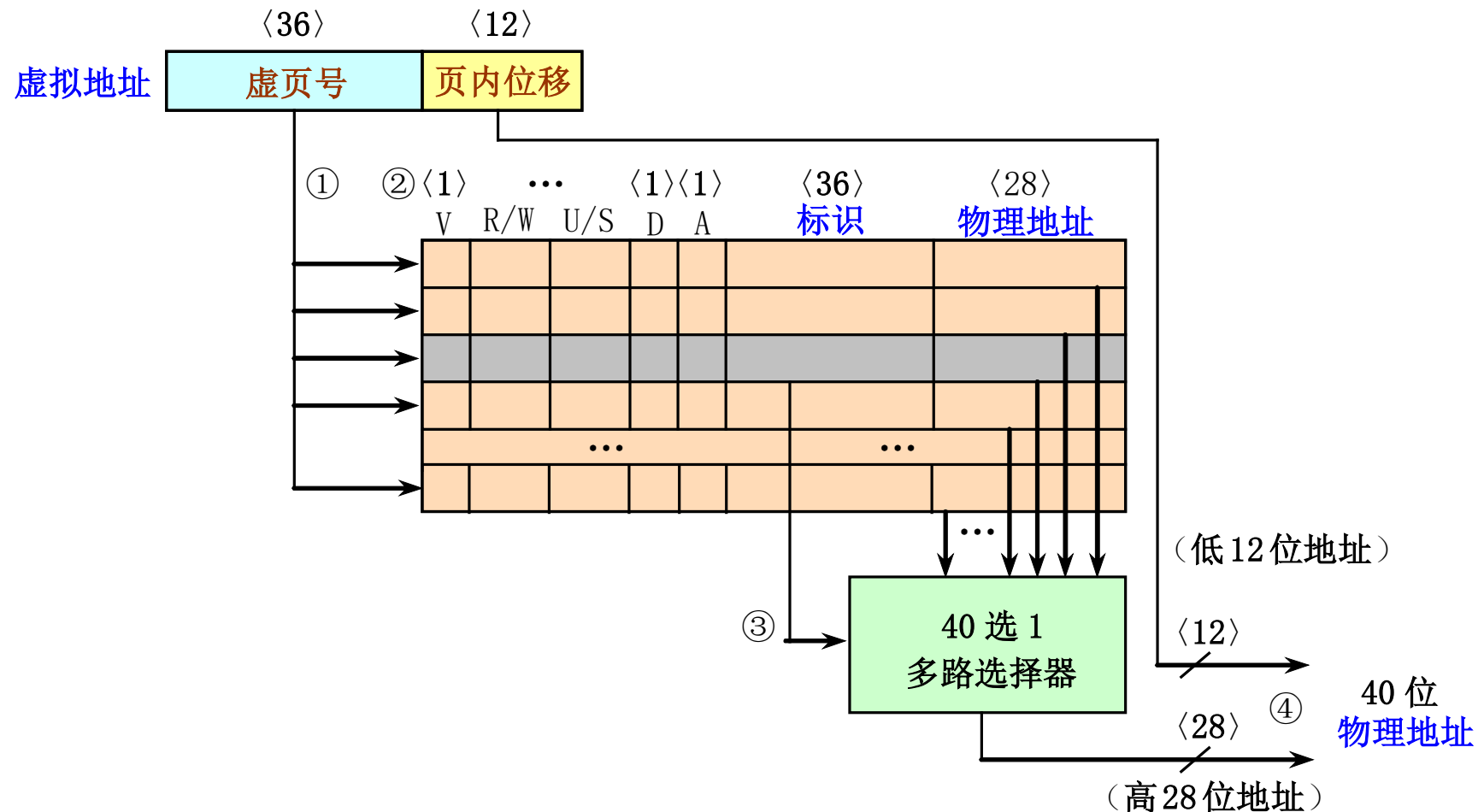
2. TLB中的项由两部分构成：标识和数据

- **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

7.7 虚拟存储器

3. AMD Opteron的数据TLB的组织结构

- 包含40个项
- 采用全相联映象



Cache与VM的区别

1. 目的不同

- Cache是为了提高访存速度
- VM是为了提高存储容量

2. 替换的控制者不同

- Cache失效由硬件处理
- VM的页失效通常由OS处理
 - 一般页失效开销很大，因此替换算法非常重要

3. 地址空间

- VM空间由CPU的地址尺寸确定
- Cache的大小与CPU地址尺寸无关

4. 下一级存储器

- Cache下一级是主存
- VM下一级是磁盘，大多数磁盘含有文件系统，文件系统寻址与主存不同，它通常在I/O空间中，VM的下一级通常称为**SWAP**空间

例：假定有一计算机，当所有存储器访问都在cache中命中时，其CPI为1.0。仅有载入和存储指令访问存储器数据，占总指令数的50%。如果cache访问缺失代价为25个时钟周期，缺失率为2%。试计算如果完全没有cache缺失，则计算机可以加快多少？

解：CPU性能公式：

CPU执行时间 = (CPU时钟周期 + 访存停顿周期) * 时钟周期

没有cache缺失，全部命中时（理想状态）：

CPU执行时间_{理想} = (IC * CPI + 0) * 时钟周期 = 1.0 * IC * 时钟周期

缺失率为2%时（实际）：

访存停顿周期 = IC * 平均每条指令访存数 * 缺失率 * 缺失代价
= IC * (1 + 0.5) * 2% * 25 = 0.75 * IC

CPU执行时间_{实际} = (IC * 1.0 + 0.75 * IC) * 时钟周期 = 1.75 * IC * 时钟周期

$$\text{每条指令缺失数} = \frac{\text{缺失数}}{\text{指令数}} = \frac{\text{缺失率} * \text{访存数}}{\text{指令数}} = \text{缺失率} * \frac{\text{访存数}}{\text{指令数}}$$

1. 假设对指令Cache的访问占全部访问的75%；而对数据Cache的访问占全部访问的25%。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，32KB的指令Cache的失效率为0.39%，32KB的数据Cache的失效率为4.82%，64KB的混合Cache的失效率为1.35%。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。试问指令Cache和数据Cache容量均为32KB的分离Cache和容量为64KB的混合Cache相比，哪种Cache的失效率更低？两种情况下平均访存时间各是多少？

解：（1）根据题意，约75%的访存为取指令。因此，分离Cache的总体失效率为： $(75\% \times 0.39\%) + (25\% \times 4.82\%) = 1.498\%$ ；

容量为64KB的混合Cache的失效率略低一些，只有1.35%。

（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 \times (读命中时间 + 读指令失效率 \times 失效开销) + 数据所占的百分比 \times (数据命中时间 + 数据失效率 \times 失效开销)

所以，两种结构的平均访存时间分别为：

分离Cache的平均访存时间 = $75\% \times (1 + 0.39\% \times 50) + 25\% \times (1 + 4.82\% \times 50) = 1.749$

混合Cache的平均访存时间 = $75\% \times (1 + 1.35\% \times 50) + 25\% \times (1 + 1 + 1.35\% \times 50) = 1.925$

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