Chap.5 Large and Fast：Exploiting Memory Hierarchy

- 5.1 Introduction
- 5.2 The Basics of Caches
- 5.3 Measuring and Improving Cache
- 5.4 Performance
- 5.5 Virtual Memory
- 5.6 A Common Framework for Memory Hierarchies

本次课程主要内容

- 层次结构的存储器
  - cache

层次结构的存储器设计

为什么存储器的设计会引入层次化设计方法？

因为，使用者：

- 希望存储器的访问速度越来越快
- 希望存储器的存储容量越来越大

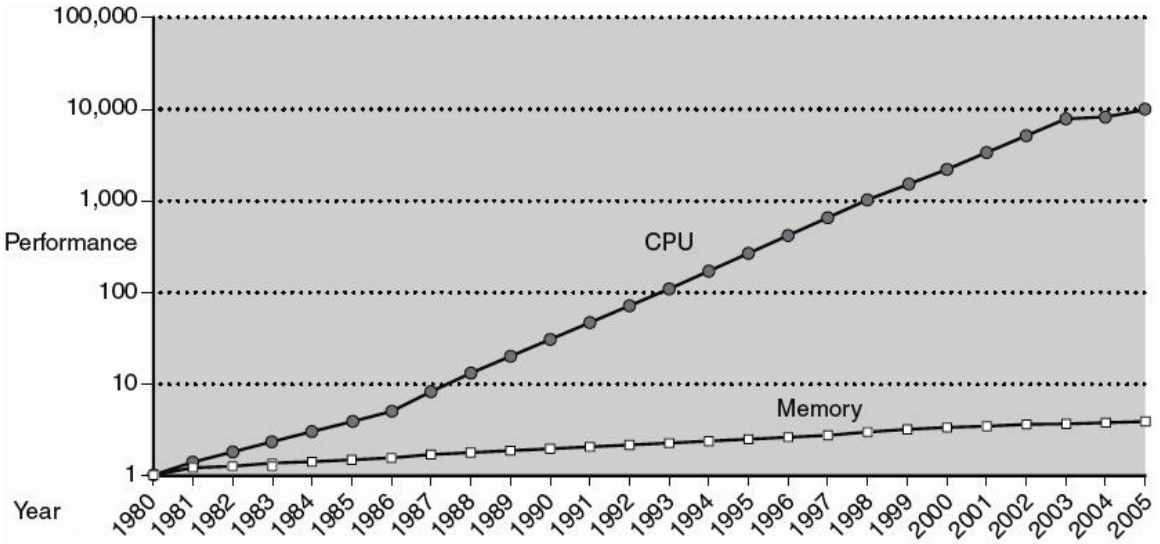
- 而且，该需求还会一直变本加厉地膨胀……

层次结构的存储器设计

真实情况是：

- 随着半导体技术的不断发展，CPU的处理能力及其处理速度在相对飞速发展，而需要与之协调工作的半导体存储器技术，其相对发展速度是严重滞后的。
- 人们对存储器要求其又大又快，这两个指标本身是一对矛盾。而且人们还要求其价格尽可能低廉。

真实情况是：CPU的处理速度要远大于存储器响应速度



- 以1980年为基准，DRAM当年为64KB，96年前每3年更新一代，之后每2年更新一代，访问时间平均每年改进7%；
- 处理器在86年之前每年提高35%，而后到2003年之前每年大约提高55%。之后速度有所减缓。

真实情况

存储器的访问速度是制约计算机处理能力的瓶颈

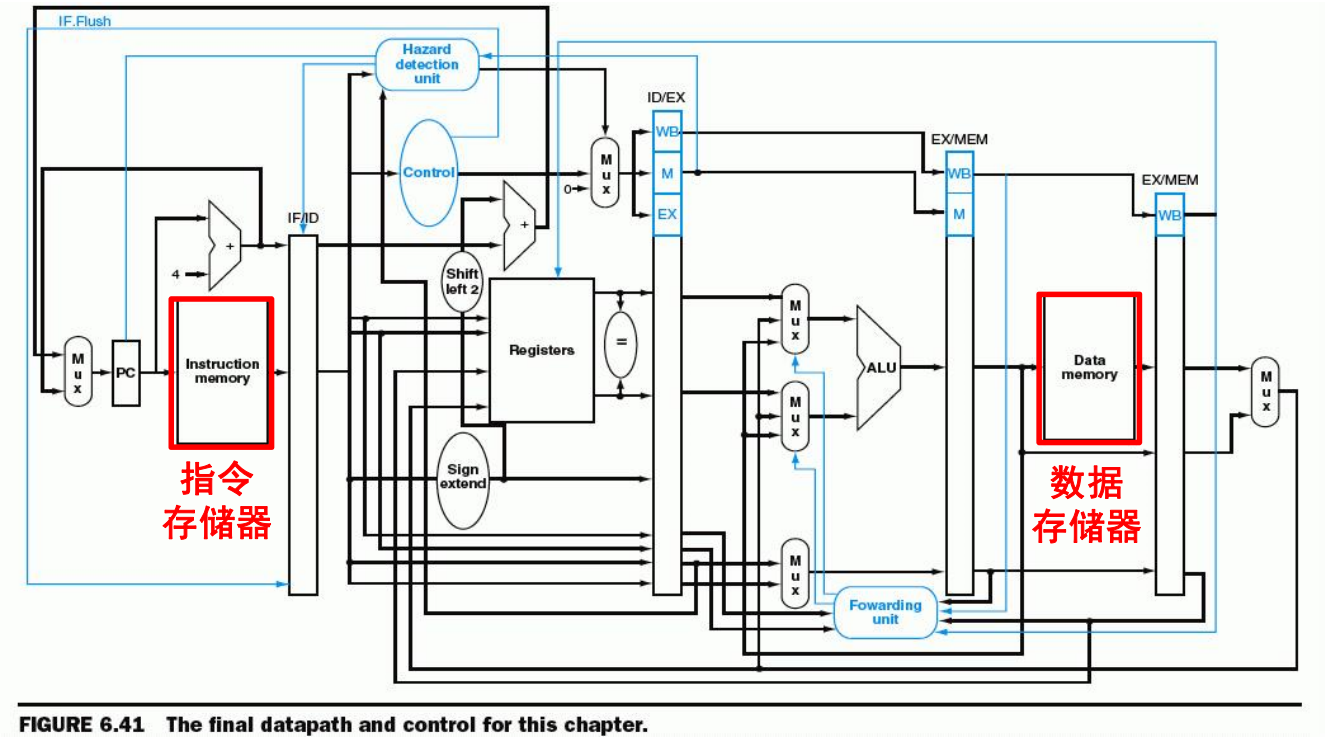


FIGURE 6.41 The final datapath and control for this chapter.

不同的存储器具有不同的读写特性和性价比

不同技术的存储器其访问时间和位价格差异很大

Memory technology	Typical access time	$ per GB in 2004
SRAM	0.5–5 ns	$4000–$10,000
DRAM	50–70 ns	$100–$200
Magnetic disk	5,000,000–20,000,000 ns	$0.50–$2

（2004年的典型数据（第三版教材））

Memory technology	Typical access time	$ per GB in 2008
SRAM	0.5–2.5 ns	$2000–$5000
DRAM	50–70 ns	$20–$75
Magnetic disk	5,000,000–20,000,000 ns	$0.20–$2

（2008年的典型数据（第四版教材））

存储器的层次结构

由于在价格和访问时间上的差异，建立存储器系统的层次结构是有利的

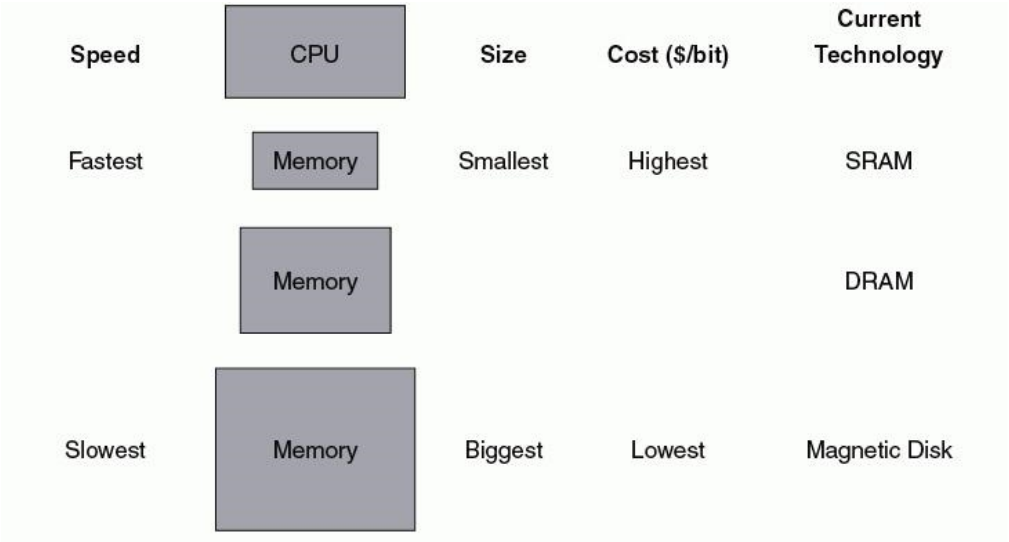


FIGURE 7.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory.

目标: 整个存储器系统的容量和最大一层相同，而速度尽量接近最高一级

存储器的层次结构

如何实现这样一个存储系统的设计目标？

【目标】：整个存储器系统的容量和最大一层相同，而速度尽量接近最高一级、也就是最快一级的访问速度。

联想一下我们日常生活中的购物过程：

楼下的便利店 → 较远一点的超市 → 更远一点的大超市 → 更大的商业中心

存储器的层次结构

CPU周围的典型存储器层次结构

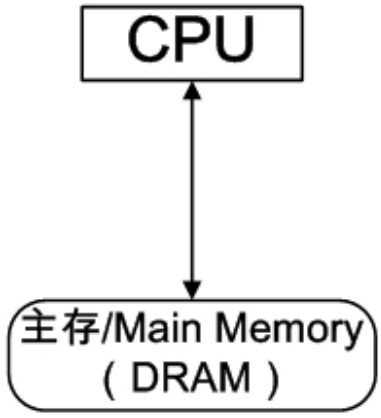
Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 512 GB	> 1 TB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25–0.5	0.5–25	50–250	5,000,000
Bandwidth (MB/sec)	50,000–500,000	5000–20,000	2500–10,000	50–500
Managed by	compiler	hardware	operating system	operating system/ operator
Backed by	cache	main memory	disk	CD or tape

Figure C.1 The typical levels in the hierarchy slow down and get larger as we move away from the processor for a large workstation or small server. Embedded computers might have no disk storage, and much smaller memories and caches. The access times increase as we move to lower levels of the hierarchy, which makes it feasible to manage the transfer less responsively. The implementation technology shows the typical technology used for these functions. The access time is given in nanoseconds for typical values in 2006; these times will decrease over time. Bandwidth is given in megabytes per second between levels in the memory hierarchy. Bandwidth for disk storage includes both the media and the buffered interfaces.

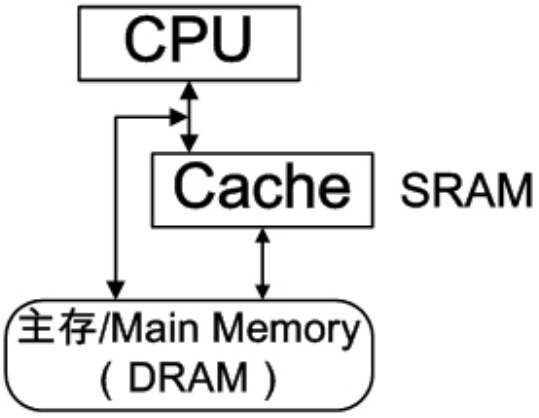
存储器的层次结构

Cache

Cache通常是存储器层次结构中第一层的名字，是距离处理器最近的存储层次。



CPU直接访问主存



CPU和主存间加入高速缓存Cache

存储器的层次结构 —— Cache

Cache 工作原理

Cache本身是位于处理器和主存之间的一块容量不算大，只是访问速度堪比CPU的一块SRAM，为何能够提高CPU访问存储器的性能呢？

- 局部性原理
  - 时间局部性
  - 空间局部性

局部性原理

- 当处理器在Cache中找到要访问的数据项时，称为Cache命中（*Cache hit*）。
- 当处理器在Cache中找不到要访问的数据项时，称为Cache缺失（*Cache miss*）。
- 包含所需要的固定大小的数据集成为一个块（*Block*），它来源于主存并被放置于Cache中。
- 时间局部性表明，该字很可能在不久的将来再次被用到，所以将它放到处理器能很快访问到的Cache中是很有意义的。
- 块内的其它数据也可能在不久的将来被访问到，这就是空间局部性原理。

例题

【例题】假定有一台计算机，当所有存储器访问操作都能在Cache中命中时，每条指令的时钟周期数（CPI）为1.0。数据访问只包含load和store，这些指令占全部指令的50%。缺失代价为25个时钟周期，缺失率为2%。当所有指令都在Cache中命中时，比较理想的全部命中情况下和真实情况下计算机的性能提高多少？

如将有cache情况和无cache情况相比，其计算机性能提高多少？

利用Cache提高存储器访问性能

- 缺失率是Cache访问中产生缺失所占访问总数的百分比，也就是产生Cache缺失的访问次数除以访问总次数。
- 读、写Cache操作的缺失率和缺失代价往往是不同的。当发生Cache缺失时，Cache下一层的存储器由于上一层的请求和 存储器刷新而变得异常忙碌，并且在处理器、总线和存储器之间相互传输数据时，时钟周期数也是不同的。因此，通常把缺失代价作为一个常量处理，是一种简化。把读、写缺失率不加区分，也是一种近似简化。

利用Cache提高存储器访问性能

解题分析

- 通常将处理器暂停工作、等待一次存储器访问的周期数称为存储器停顿周期数，性能可以表示为处理器处理时钟周期数与处理器停顿周期数的和，再乘以时钟周期时间：

CPU执行时间

=（CPU时钟周期数+存储器停顿周期数）×时钟周期时间

利用Cache提高存储器访问性能

解题分析

- 存储器停顿周期数由缺失次数和每一次的缺失代价所决定：

$$\text{存储器停顿周期数} = \text{缺失次数} \times \text{缺失代价}$$

$$= \text{执行指令数} \times \frac{\text{缺失次数}}{\text{指令数}} \times \text{缺失代价}$$

$$= \text{执行指令数} \times \frac{\text{存储器访问次数}}{\text{指令数}} \times \text{缺失率} \times \text{缺失代价}$$

利用Cache提高存储器访问性能

解题分析

- 如将读和写区分开，则有：

$$\text{存储器停顿周期数} = \text{执行指令数} \times \text{每条指令读次数} \times \text{读缺失率} \times \text{读缺失代价} + \text{执行指令数} \times \text{每条指令写次数} \times \text{写缺失率} \times \text{写缺失代价}$$

如忽略读和写的差别，上式也可简化为：

$$\text{存储器停顿周期数} = \text{执行指令数} \times \frac{\text{存储器访问次数}}{\text{指令数}} \times \text{缺失率} \times \text{缺失代价}$$

利用Cache提高存储器访问性能

【题解】Cache始终命中时的机器性能：

CPU执行时间

=（CPU时钟周期数+存储器停顿周期数）×时钟周期时间

=(执行指令数×指令执行周期数+0)×时钟周期时间

=执行指令数×1.0×时钟周期时间

利用Cache提高存储器访问性能

【题解】对于实际Cache时的机器性能，先计算存储器的停顿周期数：

存储器停顿周期数

$$= \text{执行指令数} \times \frac{\text{存储器访问次数}}{\text{指令数}} \times \text{缺失率} \times \text{缺失代价}$$

=执行指令数×(1+0.5)×0.02×25

=执行指令数×0.75

上式中，（1+0.5）表示每条指令的平均访问存储器次数，包括每条指令的1次访存取指令和50%的load和store指令的1次存取数据操作，因此，每条指令的平均存储器访问次数为1.5。

利用Cache提高存储器访问性能

【题解】因此，对于实际Cache时的机器性能，其总性能为：

$$\text{CPU执行时间}_{Cache}$$
$$=(\text{执行指令数}\times 1.0+\text{执行指令数}\times 0.75)\times\text{时钟周期时间}$$
$$=\text{执行指令数}\times 1.75\times\text{时钟周期时间}$$

利用Cache提高存储器访问性能

【题解】而对于没有采用Cache的机器，可简单地将上式计算中的缺失率等效为100%，从而可简单计算出其性能。也可直接计算：

$$\text{CPU执行时间}_{No\_Cache}$$
$$=(\text{执行指令数}\times 1.0+\text{执行指令数}\times(1+0.5)\times 25)\times\text{时钟周期时间}$$
$$=\text{执行指令数}\times 38.5\times\text{时钟周期时间}$$

利用Cache提高存储器访问性能

【题解】那么，这三种情况的机器性能可表征如下：

$$\text{CPU执行时间}_{\text{理想/全命中}} = \text{执行指令数} \times 1.0 \times \text{时钟周期时间}$$

$$\text{CPU执行时间}_{Cache} = \text{执行指令数} \times 1.75 \times \text{时钟周期时间}$$

$$\text{CPU执行时间}_{No\_Cache} = \text{执行指令数} \times 38.5 \times \text{时钟周期时间}$$

有cache的性能比没有cache的性能提高了：

$$\frac{38.5}{1.75} = 22\text{倍}$$

利用Cache提高存储器访问性能

平均每条指令缺失次数
与 平均每次存储器访问缺失次数

$$\text{平均每条指令缺失次数} = \frac{\text{总缺失次数}}{\text{总指令数}} = \frac{\text{缺失率} \times \text{总存储器访问次数}}{\text{总指令数}}$$

$$= \text{缺失率} \times \frac{\text{总存储器访问次数}}{\text{总指令数}}$$

$$= \text{缺失率} \times \text{指令平均访存次数}$$

【注】谨注意表述上的差别

利用Cache提高存储器访问性能

存储器层次结构的四个问题

1. 块的放置：一个从主存来的块，可以被放置在Cache的哪里？
2. 块的标记：如何找到放在Cache中的块？
3. 块的替换：如果块发生缺失，Cache中哪个块应该被替换出去？
4. 写时策略：CPU执行写操作时，该怎么处理？

Cache

Q1：一个块可以被放置到Cache的什么地方？

- 如果每个块在Cache中只能出现在唯一的位置上，那么这种映射就称为直接映射。
- 如果一个块可以放到Cache中的任何一个地方，那么这种映射就称为全相联映射。
- 如果一个块必须严格地放置到Cache中某组（即某一些）位置里，那么这种映射就称为组相联映射。

（载入方法不外乎以上三种）

Cache – 载入方法分析（8个Cache块，32个主存块）

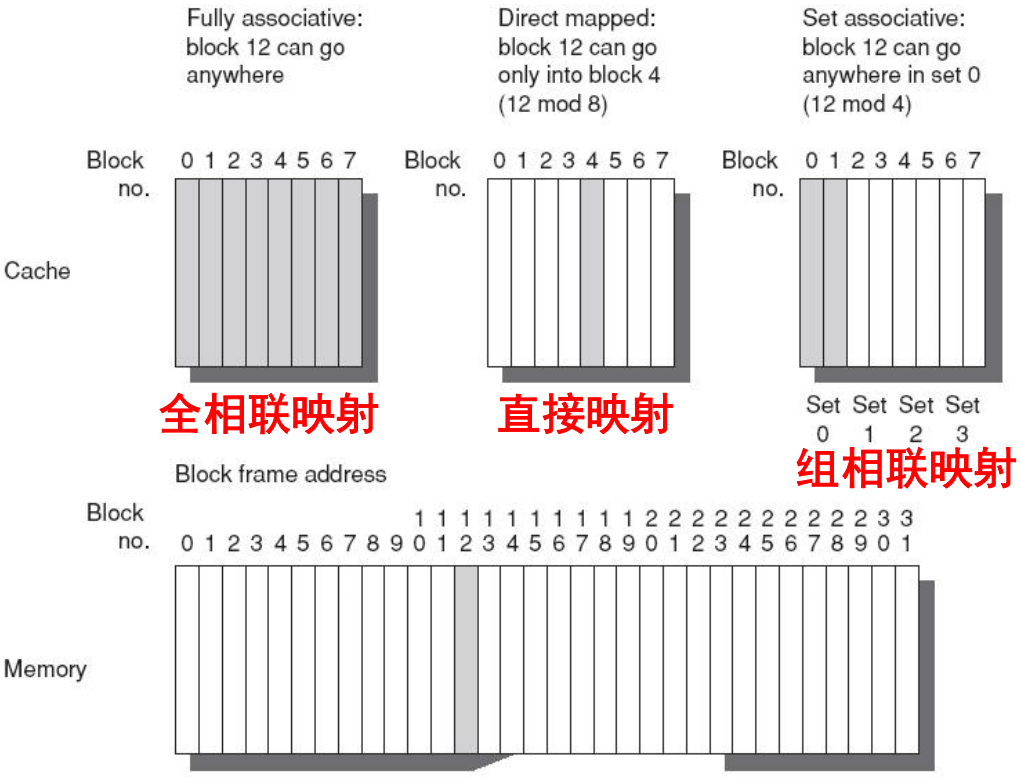


Figure C.2 This example cache has eight block frames and memory has 32 blocks.

Q1：一个块可以被放置到Cache的什么地方？

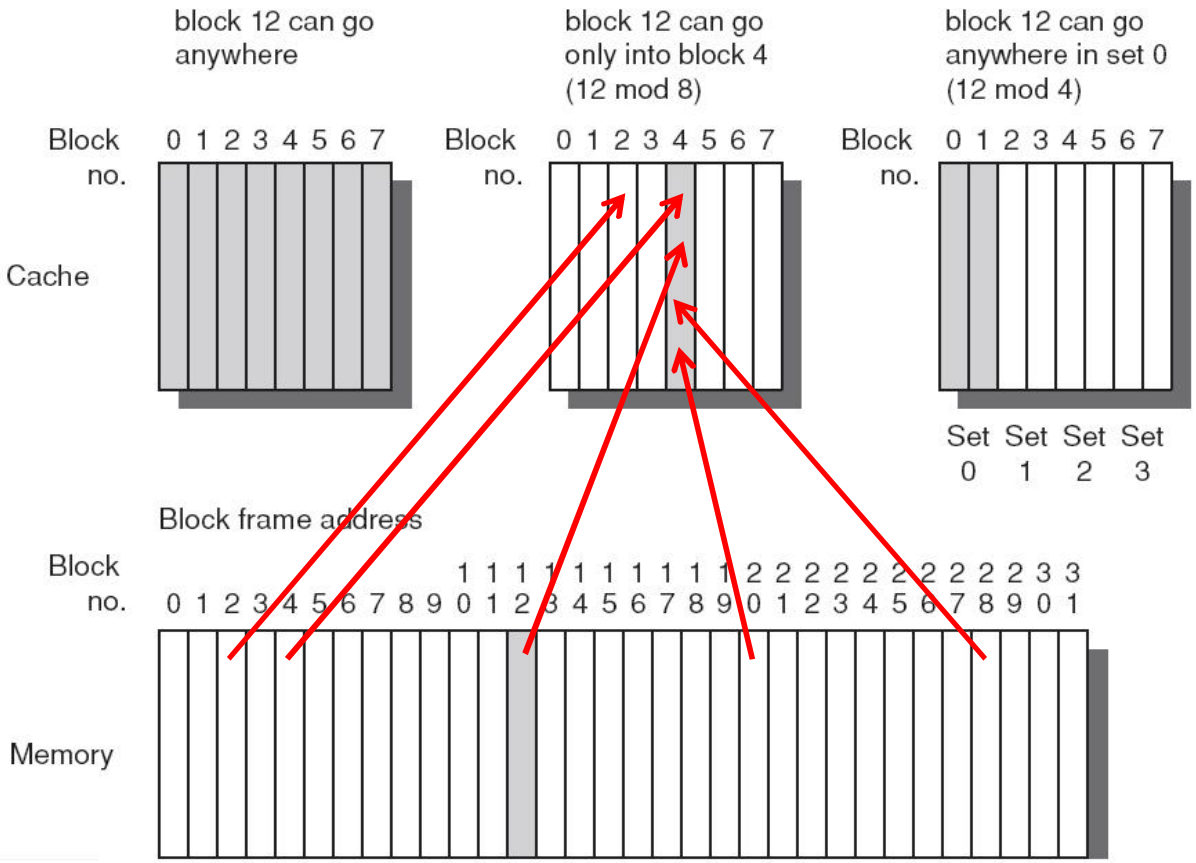
直接映射

映射方法：

建立一个将主存中的每一个块对应到Cache中的唯一位置的一个映射方法

（块地址） MOD （Cache中的块数）

直接映射：方法简单，但易导致冲突



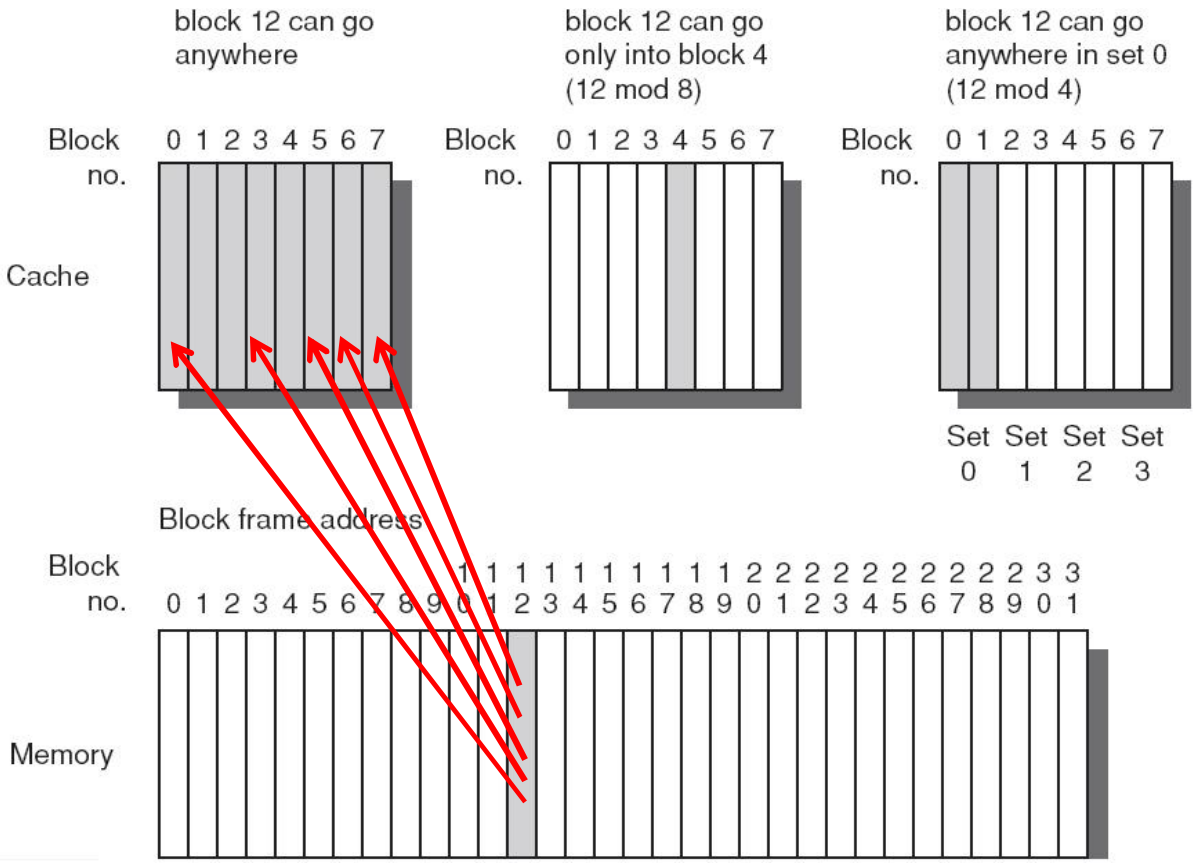
Q1：一个块可以被放置到Cache的什么地方？

全相联映射

映射方法：

哪里空就可以放到哪里，或者按某种算法剔除某位置就可以放到那里。

全相联映射：不易冲突，但映射方法复杂



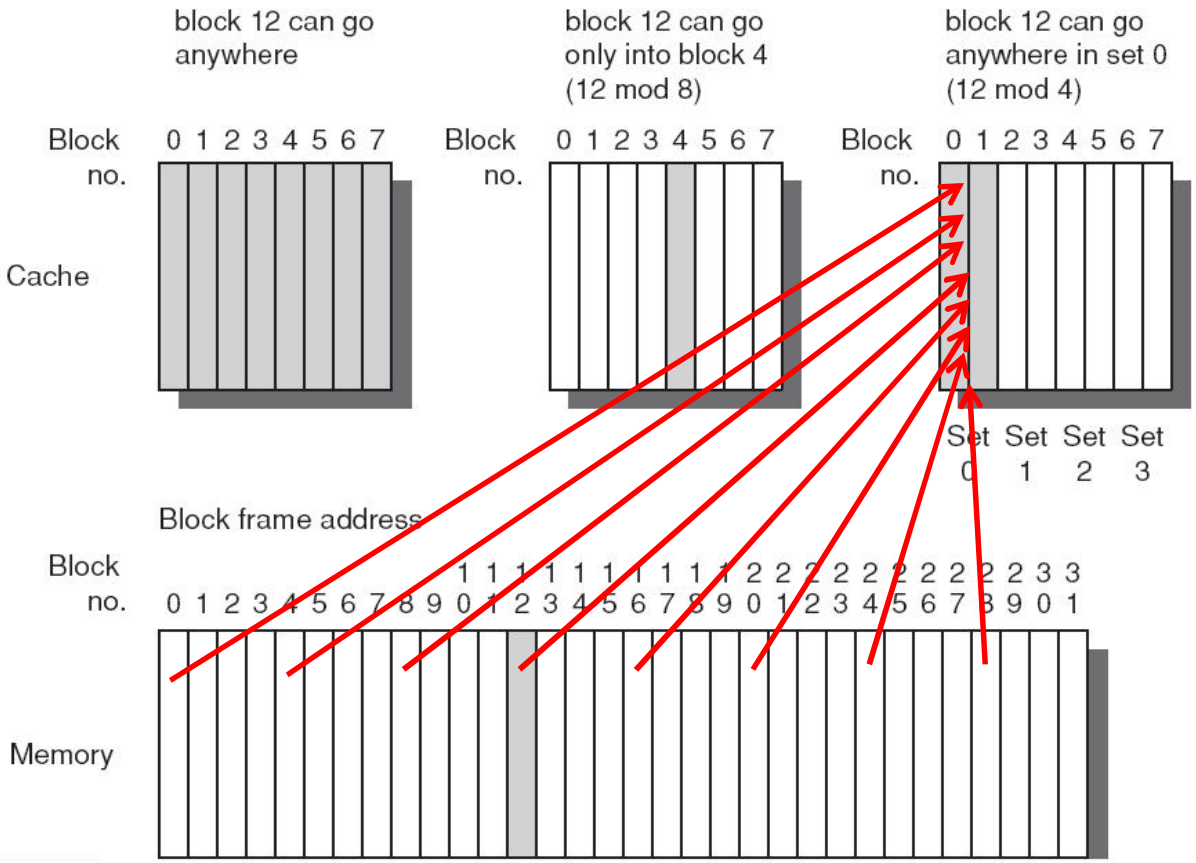
组相联：兼有前二者的优点

组相联

- 一个组是Cache中的一组块。
- 一个块首先被映射到一个组中，然后它可以被放置到组中的任何一个块中。
- 组通常利用位选择方式确定：

  （块地址）MOD（Cache的组数）
- 如果一个组里有*n*块，那么这种映射方法就称为*n*路组相联。

2路组相联映射：兼有前二者的优点



Q1：一个块可以被放置到Cache的什么地方？

三种映射方法

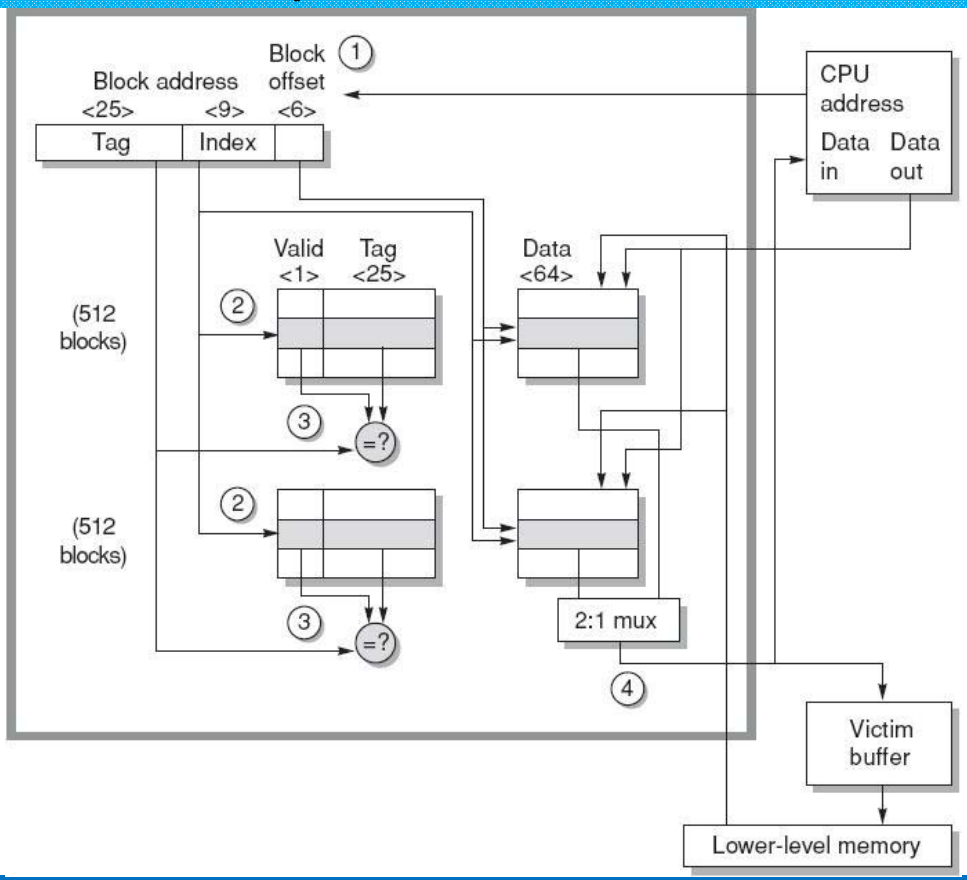
- 真正的Cache包括几千个块结构，而真正的主存则包含着成百万的块。
- 在实际应用中，今天大多数处理器的Cache采用直接映射、2路组相联映射、或4路组相联映射等。

Q2：如果某一块在Cache中，如何找到它？

- Cache的每一个块结构都有一个地址标志给出该块的主存块地址信息。
- 查找目标信息是通过对每一个Cache块标志进行检查，看它是否与来自CPU的访问地址中的主存块地址相匹配来实现的。
- 为判断某一个块是否在当前是有效的，需要在标志中加入一个有效位，来标明该块是否有效。如果该位未被置位，则该地址不能进行匹配。如刚上电，Cache内容还为空时；或作废后。
- 为追求速度，所有可能的标志都要被并行检查。

一个实例：AMD Opteron的数据Cache

Opteron微处理器中数据Cache的组织结构



一个实例：AMD Opteron的数据Cache

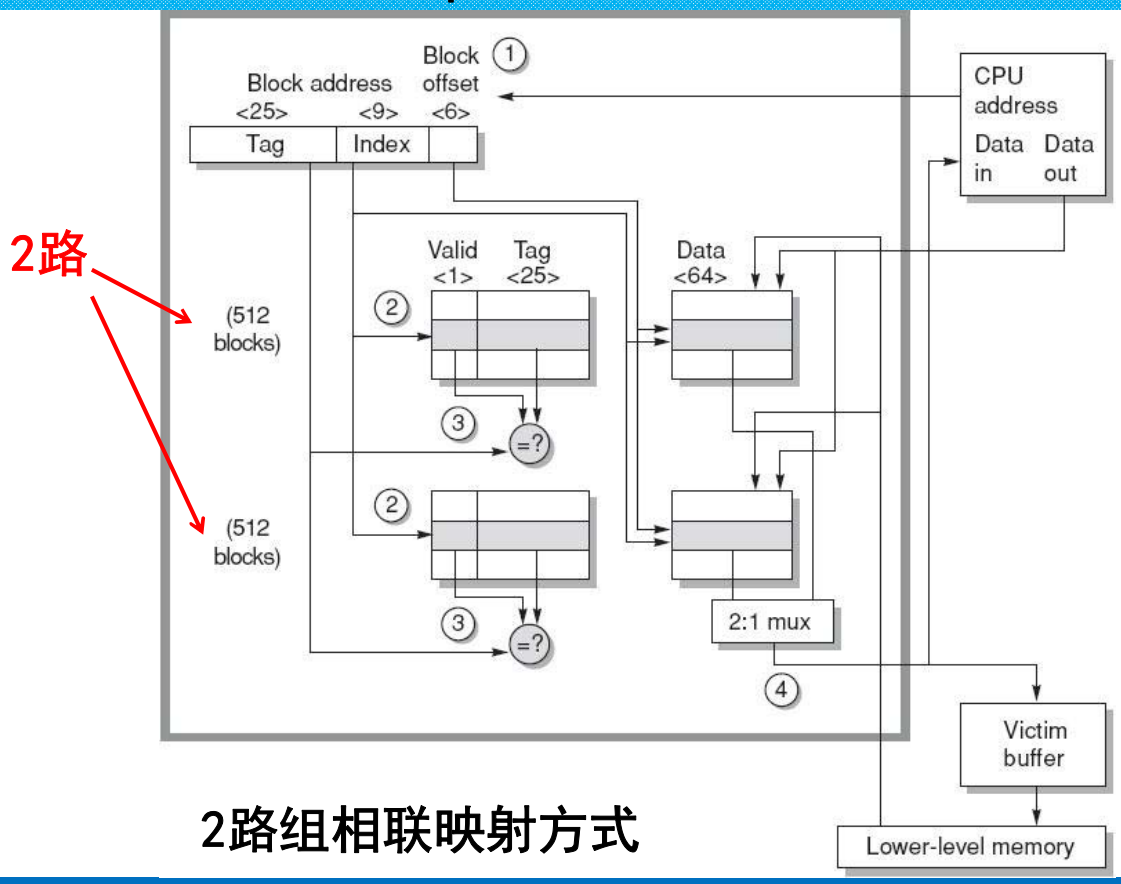
- 64KB的Cache使用64字节块，采用2路组相联映射。即每组2路，共有512组。
- 利用9位索引可以对512组进行选择。
- CPU发出的访存物理地址为40位。
- 在图中，读命中的四个步骤以圈起来的数字显示，按照发生的次序标明了组织顺序。

一个实例：AMD Opteron的数据Cache

标志字段的含义

- 以前述32个主存块、以2路组相联方式映射到8个Cache块为例，主存块号 0，4，8，12，16，20，24，28共8个块可以被放入Cache第0组。
- 对应这里的组号索引字段都为二进制00_b，哪Cache里第0组的2个块里到底放的是谁呢？
- 事实上，Cache块的标志字段放的就是主存块号的高位字段。
- 如，$28_d=11100_b$；$16_d=10000_b$；$4_d=00100_b$；如果 Cache块的标志字段=100，则在Cache块中放的是主存的第16块；如=001，则放的是主存第4块。

一个实例：AMD Opteron的数据Cache



2路组相联映射方式

一个实例：AMD Opteron的数据Cache

例题

如果上图中的映射方式改为4路组相联，若电路结构不变，会有哪些地方发生变化？

Q2：如果某一块在Cache中，如何找到它？

组相联或直接映射中地址的三个部分

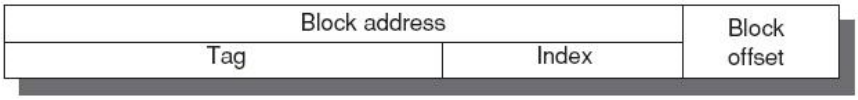


Figure C.3 The three portions of an address in a set-associative or direct-mapped cache. The tag is used to check all the blocks in the set, and the index is used to select the set. The block offset is the address of the desired data within the block. Fully associative caches have no index field.

Q3：如果Cache缺失，哪个块应该被替换？

替换策略

- 当缺失发生时，Cache控制器必须选择Cache中的一个块，并用欲获得的数据来替换它。
- 使用直接映射替换方法的优点是硬件决策简单，事实上，它不需要做任何选择，只需要检查一个块是否命中，当不命中时，也只有这个块可用于替换。
- 如是全相联或组相联，则发生缺失时，就要从多个块中作出适当选择。有三种基本的替换策略。

Q3：如果Cache缺失，哪个块应该被替换？

三种基本的替换策略

- 随机替换策略：为了均匀分配，候选块将被随机选择。
- 最近最少使用（LRU，Least-Recently Used）：如果一个最近被使用过的块很可能被再次访问，那么最好替换最近最少使用的块。（这是局部性原理的一个推论）。
- 先进先出（FIFO）替换策略：由于计算LRU比较复杂，该替换策略将最早进入Cache的块作为替换块。

三种基本的替换策略

LRU、随机和FIFO三种替换策略的性能比较
（在不同Cache大小和相联度的情况下，每千条指令的缺失率）

	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Figure C.4 Data cache misses per 1000 instructions comparing least-recently used, random, and first in, first out replacement for several sizes and associativities. There is little difference between LRU and random for the largest-size cache, with LRU outperforming the others for smaller caches. FIFO generally outperforms random in the smaller cache sizes. These data were collected for a block size of 64 bytes for the Alpha architecture using 10 SPEC2000 benchmarks. Five are from SPECint2000 (gap, gcc, gzip, mcf, and perl) and five are from SPECfp2000 (applu, art, equake, lucas, and swim). We will use this computer and these benchmarks in most figures in this appendix.

LRU、随机和FIFO三种替换策略的性能比较

（在不同Cache大小和相联度的情况下，每千条指令的缺失率）

	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

- 对于大容量Cache，LRU和随机替换算法几乎没有差别；
- 对于小容量Cache，LRU替换算法最优，FIFO算比随机替换算法性能略好一些。

Q4：写操作时会发生什么？

关于读和写

- 读操作在处理器Cache的访问中占大多数。所有的指令都是通过读操作来获得的，而且大部分指令并不进行存储器写操作。
- 《量化》教材附录B的图B.27给出的实验统计结果表明，MIPS程序所产生的写操作，占全部存储器通信量的大约7%，占数据Cache通信量的大约28%。
- 这样，加快经常性事件的速度就意味着要优化读Cache的时间。
- 但Amdahl定律告诉我们，高性能的设计不能忽视写操作速度对系统性能的影响。

写Cache策略

通常有两种写Cache的基本策略

- 写直达法（write through）：信息被同时写到Cache块和更低一层的存储器块中；
- 写回法（write back）：信息只被写入Cache块。只有Cache中该块被替换出去时，信息才会被写回到主存中。

- 为了减少替换时块的写回频率，通常使用一个称为重写（脏）位（dirty）的特征信息位。以便Cache中的内容在没有被改写的情况下，不必回写。

写Cache策略

写回法与写直达法

- 在写回法中，写操作和Cache存储器的速度一致，而且对同一块中的多次写操作仅需对下层存储器进行一次写操作。因此，较小的带宽。
- 写直达法相对更易实现，其它的读操作缺失时不会导致替换写操作。其另一个优点是，下一级存储器具有当前最新的数据副本，有良好的数据一致性。

写Cache缺失策略 – 当要写的数据不在Cache中时

写缺失时的两种策略

当写一个不在Cache中的数据时，有两种策略：

- 写分配：在发生写缺失时，将内存的块读到Cache中，然后执行写。（期望接下来对该块的写操作能够在Cache中命中）
- 不按写分配：仅修改底层存储器中的数据，不将该块调入Cache。也就不影响当前Cache内容。

写Cache缺失策略 – 当要写的数据不在Cache中时

写缺失时的两种策略：写分配与不按写分配

- 虽然这两种写缺失策略都可以应用到写回法或写直达法中，但是：
- 写回法Cache通常利用写分配策略，期望接下来对该块的写操作能够在Cache中命中。
- 写直达法经常与不按写分配策略配合使用，因为接下来对该块的写操作仍必须写入到下一层存储器中，所以采用写分配，将其调入Cache，并不能获得额外的好处。

Cache结构实例

Intel P4和AMD Opteron的一级和二级高速缓存

Characteristic	Intel Pentium P4	AMD Opteron
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	8 KB for data, 96 KB trace cache for RISC instructions (12K RISC operations)	64 KB each for instructions/data
L1 cache associativity	4-way set associative	2-way set associative
L1 replacement	Approximated LRU replacement	LRU replacement
L1 block size	64 bytes	64 bytes
L1 write policy	Write-through	Write-back
L2 cache organization	Unified (instruction and data)	Unified (instruction and data)
L2 cache size	512 KB	1024 KB (1 MB)
L2 cache associativity	8-way set associative	16-way set associative
L2 replacement	Approximated LRU replacement	Approximated LRU replacement
L2 block size	128 bytes	64 bytes
L2 write policy	Write-back	Write-back

FIGURE 7.35 First-level and second-level caches in the Intel Pentium P4 and AMD Opteron. The primary caches in the P4 are physically indexed and tagged; for a discussion of the alternatives, see the Elaboration on page 527.

一体Cache和独立的指令和数据Cache

指令Cache比数据Cache有更低的缺失率

Size	Instruction cache	Data cache	Unified cache
8 KB	8.16	44.0	63.0
16 KB	3.82	40.9	51.0
32 KB	1.36	38.4	43.3
64 KB	0.61	36.9	39.4
128 KB	0.30	35.3	36.2
256 KB	0.02	32.6	32.9

Figure C.6 Miss per 1000 instructions for instruction, data, and unified caches of different sizes. The percentage of instruction references is about 74%. The data are for two-way associative caches with 64-byte blocks for the same computer and benchmarks as Figure C.4.

注：数据为每千条指令缺失数。指令访问频率大约为74%，数据访问频率26%。性能对比时需以同等总容量做对比。

Cache性能评价

对存储器层次结构较合理的评价方法是采用**平均存储器访问时间**：

平均存储器访问时间＝命中时间+缺失率×缺失代价

【例题】一个16KB指令Cache加一个16KB数据Cache，与一个32KB的一体Cache相比较，哪一个具有更低的缺失率？

假设36%的存储器访问是数据访问，使用前图C.6中的缺失率来计算正确的结果。假设Cache命中需要一个时钟周期，缺失代价是200个时钟周期，在一体Cache中，load或store命中额外需要一个时钟周期，因为只有一个Cache端口来满足这两个同时发生的请求（结构冒险）。每种情况下的平均存储器访问时间是多少？假定是带有写缓存的写直达Cache，而且写缓存的停顿时间可以忽略不计。

Cache性能评价——例题

【题解】C.6图中给出的是每千条指令的访问缺失率。

平均每条指令缺失次数 = 缺失率×指令平均访存次数

一条指令只进行一次取指令的访存操作，故指令的访问缺失率为：

$$缺失率_{16KB指令Cache} = \frac{3.82/1000}{1.0} = 0.004$$

由题目知36%的指令是数据操作指令，则数据缺失率为

$$缺失率_{16KB数据Cache} = \frac{40.9/1000}{0.36} = 0.114$$

注意：0.36/(1+0.36)=0.26=26%，C6和本题意是一致的。
1/(1+0.36)=0.74=74%

Cache性能评价——例题

【题解】一体Cache的缺失率包括指令和数据的缺失率：

$$缺失率_{32KB一体Cache} = \frac{43.3/1000}{1.0+0.36} = 0.0318$$

由于74%的存储器访问是取指令，因此，分立Cache的全局缺失率为：

$$(74\% \times 0.004) + (26\% \times 0.114) = 0.0326$$

因此，32KB的一体Cache比2个16KB单体Cache有相对较低的缺失率。

Cache性能评价——例题

【题解】求平均存储器访问时间：

平均存储器访问时间
＝指令所占比例×(命中时间+指令缺失率×缺失代价)
＋数据所占比例×(命中时间+数据缺失率×缺失代价)

两种情况分别解算如下：

$$\text{平均存储器访问时间}_{\text{分立Cache}}$$
$$=74\%\times(1+0.004\times200)\ +26\%\times(1+0.114\times200)$$
$$=74\%\times1.8+26\%\times23.8=1.332+6.118=7.52$$

$$\text{平均存储器访问时间}_{\text{一体Cache}}$$
$$=74\%\times(1+0.0318\times200)\ +26\%\times(1+1+0.0318\times200)$$
$$=74\%\times7.36+26\%\times8.36=5.446+2.174=7.61$$

相比较而言，分立的要好一些，虽然其缺失率稍大些。

Cache优化技术

使用多级Cache来降低缺失代价

- 减少缺失次数，提高命中率，是传统Cache的研究方向，但是Cache的性能公式表明，缺失代价的改善与缺失率的改善同样重要。
- 因此，可以将Cache的思想，继续应用于Cache和主存之间。
- 通过在原来的Cache和存储器之间，再增加一级Cache：使得第一级Cache可以小到足以与快速的处理器运行速度相匹配；而第二级Cache能够大到足以捕捉到对内存进行的多数访问，从而有效减少缺失代价。

使用多级Cache来降低缺失代价

- **局部缺失率**：这一级Cache的缺失数除以Cache的存储器访问总数。
  - 在一级Cache中，它等于缺失率$_{L1}$；
    在二级Cache中，它等于缺失率$_{L2}$。
- **全局缺失率**：这一级Cache的缺失数除以处理器产生的访存总数。
  - 使用上面的术语，一级Cache的全局缺失率仍为缺失率$_{L1}$，二级Cache的全局缺失率为 缺失率$_{L1}$×缺失率$_{L2}$。

多级Cache的平均存储器访问时间

$$\text{平均存储器访问时间} = \text{命中时间}_{L1} + \text{缺失率}_{L1} \times \text{缺失代价}_{L1}$$

其中：

$$\text{缺失代价}_{L1} = \text{命中时间}_{L2} + \text{缺失率}_{L2} \times \text{缺失代价}_{L2}$$

因此有：

$$\text{平均存储器访问时间}_{\text{2级Cache}} = \text{命中时间}_{L1} + \text{缺失率}_{L1} \times (\text{命中时间}_{L2} + \text{缺失率}_{L2} \times \text{缺失代价}_{L2})$$

使用多级Cache来降低缺失代价

多级Cache的平均存储器停顿周期

$$\text{每条指令的平均存储器停顿周期}_{\text{2级Cache}}$$
$$= \text{每条指令缺失次数}_{L1} \times \text{命中时间}_{L2}$$
$$+ \text{每条指令缺失次数}_{L2} \times \text{缺失代价}_{L2}$$

使用多级Cache来降低缺失代价

多级Cache

【例题】假定在1000次访存中，一级Cache中有40次缺失，二级Cache中有20次缺失。两种缺失率分别为多少？假设二级Cache到存储器的缺失代价为200个时钟周期，二级Cache的命中时间为10个时钟周期；一级Cache的命中时间是1个时钟周期，每条指令的存储器访问次数为1.5。求平均存储器访问时间和每条指令的平均停顿周期时间各为多少（写操作影响不计）。

使用多级Cache来降低缺失代价

例题解答

【解答】

一级Cache的缺失率（局部或是全局）＝ 40/1000=4%

二级Cache的局部缺失率 ＝ 20/40＝50%

二级Cache的全局缺失率 ＝ 20/1000=2%

因此:

平均存储器访问时间=

＝命中时间$_{L1}$+缺失率$_{L1}$×

（命中时间$_{L2}$+缺失率$_{L2}$×缺失代价$_{L2}$)

=1 +4%×(10+ 50%×200)

=1+4%×110

＝5.4个时钟周期

使用多级Cache来降低缺失代价

【解答】

- 为了得到每条指令的缺失次数，将1000次存储器访问除以每条指令的访存次数1.5，得667条指令。
- 因此，需要将缺失次数乘以1.5得到每1000条指令的缺失次数。
- 每1000条指令一级Cache的缺失数次为60次，二级Cache的缺失次数为30次。
- 假设数据和指令的缺失次数相等，则每条指令的平均存储器停顿周期数为：

每条指令的平均存储器停顿周期＝

＝每条指令缺失次数$_{L1}$×命中时间$_{L1}$

+每条指令缺失次数$_{L2}$×缺失代价$_{L2}$

＝(60/1000)×10+ (30/1000)×200

＝0.060×10+ 0.030×200 ＝ 6.6个时钟周期

使用多级Cache来降低缺失代价

【解答】

- 如果从平均存储器访问时间(AMAT)中减去一级Cache的命中时间，再乘上每条指令的平均访存（存储器访问）次数，也能得到相同的平均每条指令的存储器停顿周期数：

  （5.4 - 1.0）×1.5= 4.4×1.5= 6.6个时钟周期

- 由此可见，在多级Cache中，用每条指令缺失数来计算要比用缺失率进行计算更加清晰。

例：Intel Core i7的高速缓存层次结构

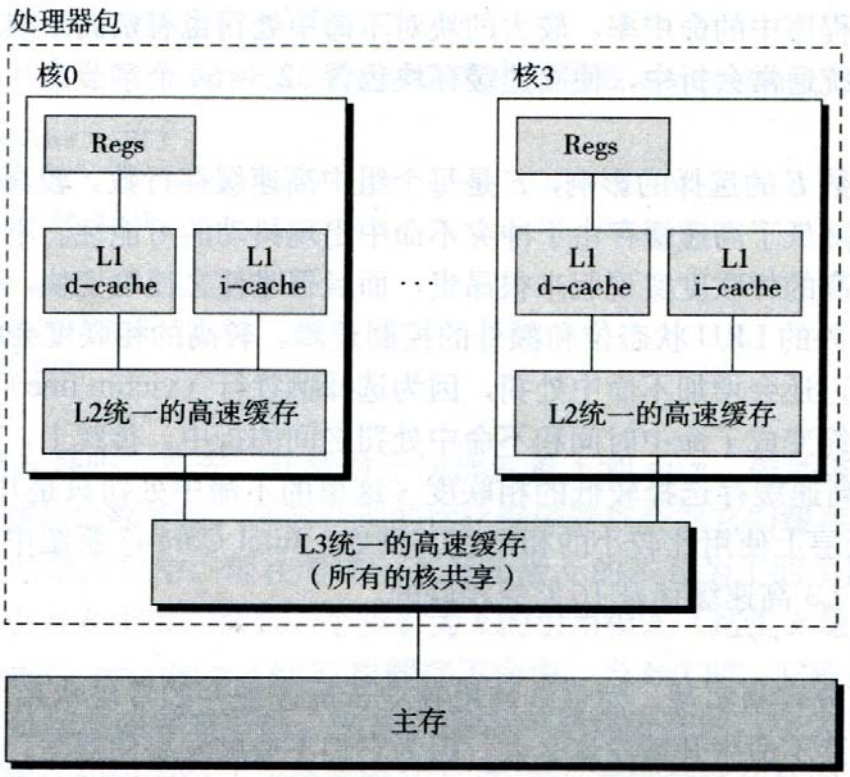


图 6-40 Intel Core i7 的高速缓存层次结构 来自《深入理解计算机系统》

例：Intel Core i7的高速缓存层次结构

Intel Core i7的高速缓存层次结构特性

高速缓存类型	访问时间（周期）	高速缓存大小（C）	相联度（E）	块大小（B）	组数
L1 i-cache	4	32KB	8	64B	64
L1 d-cache	4	32KB	8	64B	64
L2 统一的高速缓存	11	256KB	8	64B	512
L3 统一的高速缓存	30~40	8MB	16	64B	8192

图 6-41　Core i7 高速缓存层次结构的特性

来自《深入理解计算机系统（第二版）》