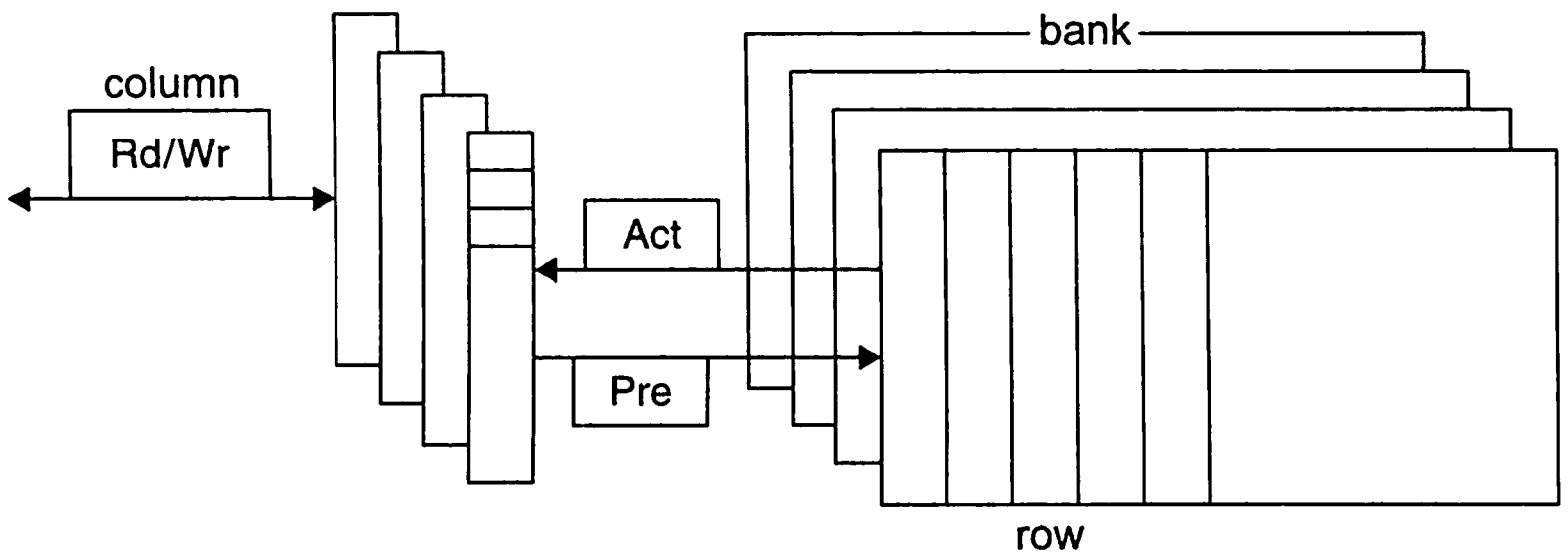

图 5-4 DRAM 的内部结构。现代 DRAM 以 bank 的方式来组织结构。典型地，DDR3 被划分为四个 bank。每个 bank 包括一系列行（row）。发送预充电（Pre，pre-charge）命令可以打开或关闭某个 bank。激活（Act，activate）命令发送行地址，并将对应某行的数据传输到缓冲区中。当某行写入缓冲后，无论 DRAM 的宽度是多少（DDR3 中典型的数据传输宽度为 4、8、16 位），可以通过发送后续列地址进行数据传输，或者通过指定起始地址进行块传输。与块传输一样，每个命令都与时钟同步

生产年份	芯片容量	美元/GB	新行/列的访问时间（ns）	缓冲行的平均列访问时间（ns）
1980	64 Kibibit	1 500 000	250	150
1983	256 Kibibit	500 000	185	100
1985	1 Mebibit	200 000	135	40
1989	4 Mebibit	50 000	110	40
1992	16 Mebibit	15 000	90	30
1996	64 Mebibit	10 000	60	12
1998	128 Mebibit	4 000	60	10
2000	256 Mebibit	1 000	55	7
2004	512 Mebibit	250	50	5
2007	1 Gibibit	50	45	1.25
2010	2 Gibibit	30	40	1
2012	4 Gibibit	1	35	0.8

图 5-5 1996 年后，DRAM 容量以大约每三年四倍的速度增长，之后趋势放缓。访问时间的改善已经放缓，但仍在持续。虽然存储芯片的成本受到其他因素如可用性和需求的影响，但大体上仍保持随密度的增加而降低。每 GB 的成本不随通货膨胀进行调整

行（row）结构有助于 DRAM 的刷新，也有助于改善性能。为提高性能，DRAM 中缓存了行数据以便重复访问。该缓冲区有点类似 SRAM：通过改变地址，可以访问缓冲区中任意位置的数据，直到换行。这个功能显著改善了访问时间，因为确定行数据的访问时间被大幅度降低。让芯片变得更宽也能改善存储带宽。当某行已在缓冲区中，无论 DRAM 的数据宽度是多少（典型的为 4、8、16 位），可以通过发送后续列地址进行数据传输，或者通过指定缓冲区中的起始地址进行块传输。

为更好地优化与处理器的接口，DRAM 添加了时钟，因此被称为同步 DRAM（Synchronous DRAM，SDRAM）。SDRAM 的好处在于，使用时钟消除了内存和处理器之间的同步问题。同步 DRAM 的速度优势来自于，进行突发传输（burst transfer）时无须指定额外地址位，而是通过时钟来突发传输后续数据。速度最快的结构称为*双倍数据传输率*（Double Data Rate，DDR）SDRAM，这名字意味着在时钟的上升沿和下降沿都可以进行数据传输。因此，如果根据时钟频率和数据位宽测算，使用该结构可以获得预想中双倍的数据带

宽。该技术的最新架构称为 DDR4，DDR4-3200 DRAM 能够在 1.6GHz 工作频率下，每秒进行 32 亿次的数据传输。

要维持这样的高带宽，需要对 DRAM 的内部结构进行精心组织。DRAM 可以在内部组织对多个 bank 进行读或写，每个 bank 对应各自的行缓冲，而不仅仅是添加一个快速行缓冲。对多个 bank 发送一个地址允许同时对这些 bank 进行读或写。例如，对于 4 个 bank 的结构，只需要一次访问时间，之后以轮转的方式对这 4 个 bank 进行访问，这样就获得了 4 倍的带宽。这种轮转的访问方式称为交叉地址访问（address interleaving）。

虽然如 iPad 这样的个人移动设备（详见第 1 章）都会使用单独的 DRAM，但是服务器的存储通常都是集成在小电路板上售卖，这种结构被称为双列直插式内存模块（Dual Inline Memory Modules，DIMM）。典型的 DIMM 包括 4 ～ 16 个 DRAM 颗粒，在服务器系统中每个 DRAM 颗粒通常被组织成 8 字节宽度。一个使用了 DDR4-3200 SDRAM 的 DIMM 每秒能够传输 $8 \times 3200 = 25\ 600$ MB 的数据，这样的 DIMM 结构使用它的带宽来命名：PC25600。一个 DIMM 可以有许多 DRAM 芯片，但是在特定的传输中只有其中的一部分芯片会被使用。因此需要一个术语来表示 DIMM 中共享地址总线的芯片集合。为避免与 DRAM 内部的行和 bank 混淆，使用 rank 来表示这一芯片子集。

详细阐述 衡量高速缓存之外的存储系统性能的方法之一，就是使用流式基准测试程序（Stream benchmark）[McCalpin, 1995]。该程序主要测量长向量操作的性能，这些操作没有时间局部性，访问的数组大小比待测机器中的 cache 容量要大。

5.2.3　闪存

闪存是一种电可擦除的可编程只读存储器（Electrically Erasable Programmable Read-only Memory，EEPROM）。

与磁盘和 DRAM 不同，但与其他 EEPROM 技术相似，闪存的写操作会对器件本身产生磨损。为应对这种限制，大多数闪存产品都包括一个控制器，用来将发生多次写的块重新映射到较少被写的块，从而使得写操作尽量分散。该技术被称为耗损均衡（wear leveling）。通过该技术，个人移动设备不太可能超过闪存的写次数限制。这样的技术虽然降低了闪存的潜在性能，但却是非常必要的，除非使用更高层次的软件来监控损耗情况。具有损耗均衡的闪存控制器也能够通过重新映射，将生产制造中出现故障的存储单元屏蔽掉，从而改善产品的良率（yield）。

5.2.4　磁盘

如图 5-6 所示，磁性硬盘由一堆盘片组成，这些盘片绕着轴心每分钟转动 5400 ～ 15000 周。这些金属盘片的每一面都被磁性记忆材料所覆盖，与磁带上的磁性材料相似。为从硬盘上读写信息，一个可移动的转臂正位于这些盘面的上方，其中包括一个称为读

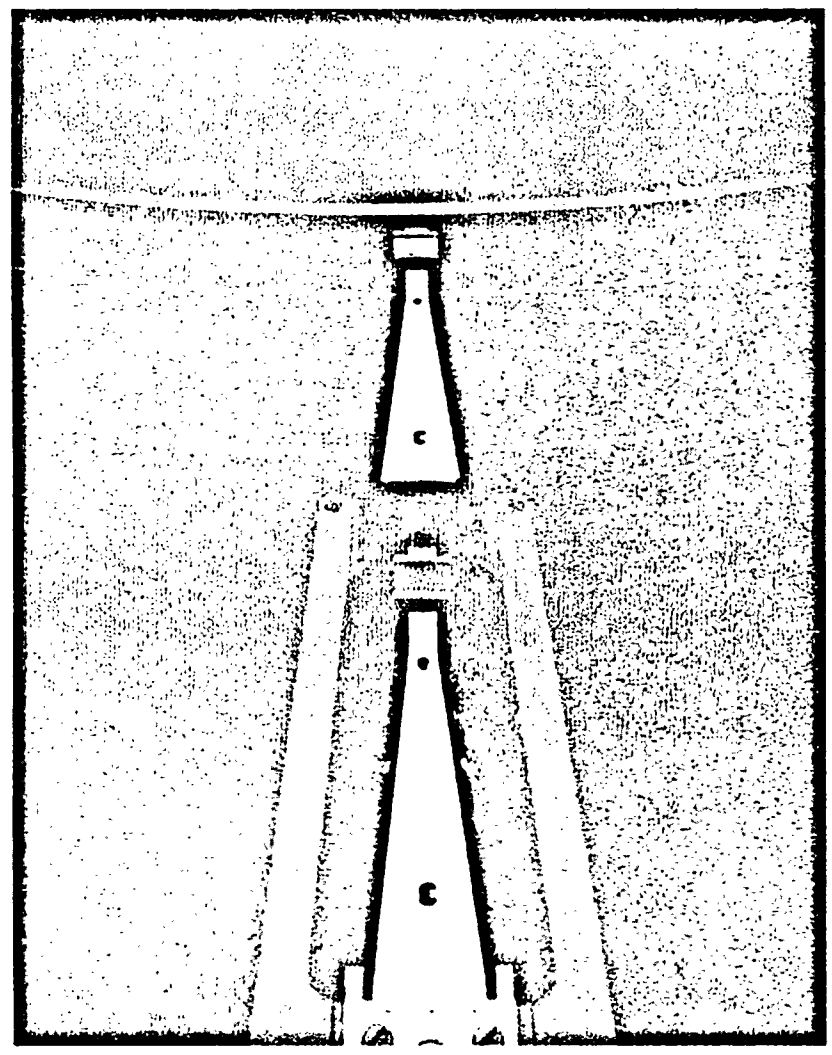

图 5-6　具有 10 个盘片和读写头的磁盘。现代磁盘的直径为 2.5 ～ 3.5 英寸，每个驱动器通常控制 1 ～ 2 个盘片

写头的小型电磁线圈。整个驱动器严格密封，以控制驱动器内部的环境，从而使磁头更接近驱动器表面。

每个磁盘表面被分为若干的同心圆，称为**磁道**（track）。每个盘面上通常有几万条磁道。每条磁道按序划分为上千个保存信息的**扇区**（sector）。扇区的容量一般为 512 ～ 4096 字节。记录在磁介质上的内容依次为：扇区号，间隙，该扇区的信息（包括纠错码，详见 5.5 节），间隙，下一个扇区的扇区号，等等。

磁道：磁盘表面上千同心圆中的一个。

扇区：磁盘磁道上的一段，扇区是读写磁盘信息的最小单位。

每个盘面都配有一个磁头，这些磁头互相连接并一起移动。每个磁头都可以读写每个盘面的每一条磁道。术语柱面（cylinder）用来表示某磁头在给定点能够访问到的所有磁道集合。

操作系统通过三步完成对磁盘的数据访问。第一步，将磁头定位在正确的磁道上方。这个操作称为**寻道**（seek），将磁头移动到所需磁道上方的时间称为寻道时间（seek time）。

寻道：将读写头定位到磁盘上正确的磁道上方的过程。

磁盘制造商在他们的手册中提供最小寻道时间、最大寻道时间和平均寻道时间。前两者很容易测量，但是平均寻道时间却因为与寻道距离有关而难以解释。工业界计算平均寻道时间的方法通常是：对所有可能的寻道时间求和并计算平均值，（这样计算出来的）平均寻道时间通常为 3ms ～ 13ms。但是，由于应用及磁盘请求调度策略的不同，同时磁盘访问存在局部性，事实上平均寻道时间可能仅有上述数值的 25% ～ 33%。如果考虑到对同一个文件做连续访问，且操作系统也会尽量对这样的访问进行集中调度，那么局部性还会增加。

一旦磁头到达正确的磁道，我们需要等待所需扇区旋转到读写磁头下，这段时间被称为**旋转延时**（rotational latency 或 rotational delay）。获得所需信息的平均延时为磁盘旋转半周的时间。磁盘以 5400 转 / 分钟（Rotation Per Minute，RPM）～ 15000 转 / 分钟的速度旋转，当转速为 5400 转 / 分钟时，平均旋转延时为：

旋转延时：也称为旋转延迟（rotational delay），即磁盘上所需扇区旋转到读写磁头下的时间。通常假设为旋转半周的时间。

$$
\begin{aligned}
\text{平均旋转延时} &= \frac{0.5\ \text{转}}{5400\ (\text{转} / \text{分钟})} \\
&= \frac{0.5\ \text{转}}{5400\ (\text{转} / \text{分钟}) / 60\ (\text{秒} / \text{分钟})} \\
&= 0.0056\ \text{秒} \\
&= 5.6\ \text{毫秒}
\end{aligned}
$$

磁盘访问的最后一部分是传输时间（transfer time），即传输数据块（block）的时间。传输时间是扇区大小、旋转速度和磁道记录密度的函数。2012 年，磁盘的传输速率大约在 100 ～ 200MB/s。

复杂的是，大多数磁盘控制器内置一个缓存，用来保存刚刚读取过的扇区的数据。从该缓存中读取数据的传输速率会高得多，2012 年达到了 750MB/s（即 6Gbit/s）。

现在，块号的位置不能再凭直觉了。上述的扇区 – 磁道 – 磁柱模型有如下假设：相邻的数据块在同一磁道上；由于无须寻道时间，在同一磁柱上的数据块的访问开销更少；不同的磁道与磁头的距离也不同。如此变化的原因在于磁盘接口的层次在提升。为加速磁盘的串行传输速度，这些高层次接口将磁盘组织得更像磁带，而不是随机访问设备。逻辑块（沿圆周）以弯曲的方式排列在盘片表面，尽可能使每个扇区的记录密度相同，以获得最佳性能。因

而，顺序的数据块可能会在不同的磁道上。

综上所述，磁盘与半导体存储器件的两个主要区别是：由于存在机械装置，磁盘的访问速度更慢。例如，闪存的速度是磁盘的1000倍，DRAM是磁盘的100 000倍。但是，由于可以最低的成本获得非常高的存储容量，磁盘的单位成本更低，一般便宜10到100倍。与闪存相同，磁盘也是非易失性存储。但是与闪存不同的是，磁盘没有写损耗问题。不过闪存更坚固，因此更适合个人移动设备。

5.3　cache基础

在上文的图书馆例子中，书桌扮演了cache（高速缓存）的角色：一个存储考试所需信息（书籍）的安全场所。在第一台商用计算机中，cache就被用来表示位于处理器和主存之间的特殊存储层次。第4章数据通路中的存储都可以简单地替换成cache。如今，虽然这些仍是cache这个词的主要使用场合，但它也被用来表示利用访问局部性进行管理的其他存储。早在20世纪60年代早期，cache就第一次出现在计算机研究中，随后又出现在计算机产品中。如今，从服务器到低功耗嵌入式处理器，每一台通用计算机都包含cache。

> **缓存**：一个隐藏或者存储信息的安全场所。
> *Webster's New World Dictionary of the American Language, Third College Edition, 1988*

在这一节中，我们从一个简单的cache开始，处理器每次请求为一个字，且每个数据块由单个字组成（已经非常熟悉cache基础理论的读者可以直接转到5.4节）。图5-7中给出了这样的简单cache在响应数据访问请求前后的情况，且被访问的数据初始时并不在cache中。在响应请求前，该cache中包含了最近访问的数据集合X_1，X_2，…，X_{n-1}，处理器请求的字X_n并不在cache中。这个请求会产生一次失效，字X_n从内存取到cache中。

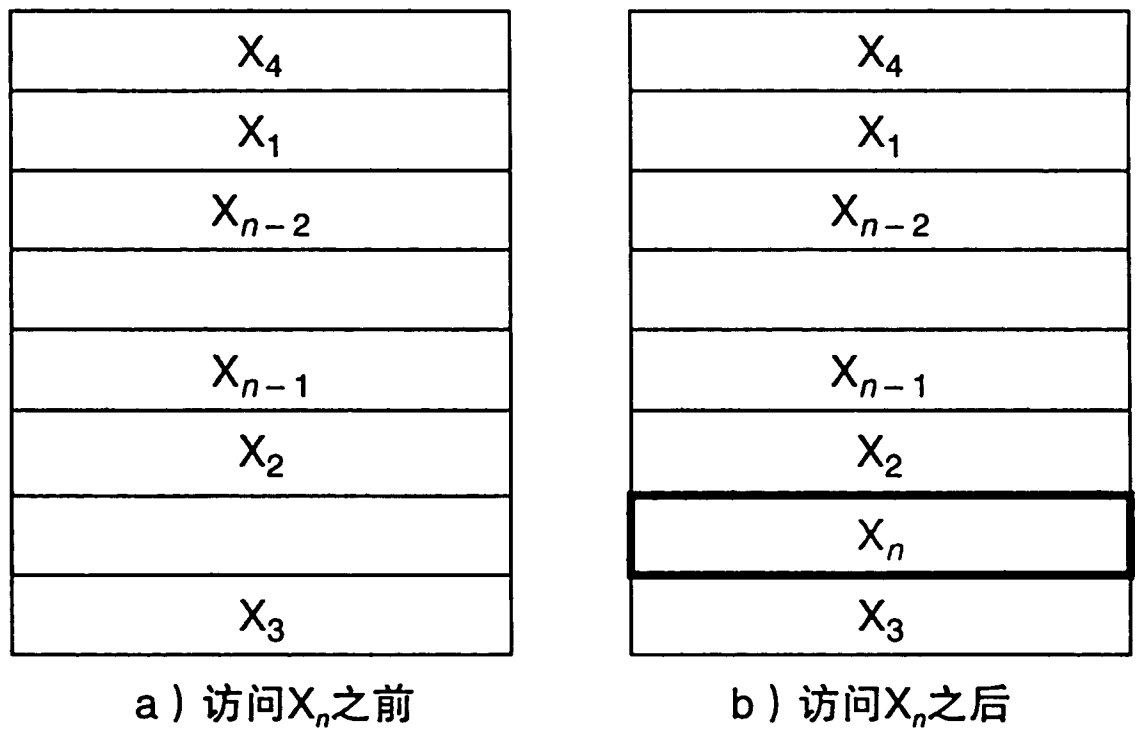

图5-7　访问字X_n前后的cache，且初始时X_n不在cache中。本次访问引起了cache失效，并强制将X_n从内存中取出并插入cache中

在对图5-7所述场景进行分析时，有两个问题需要回答：如何知道数据项是否存在于cache中？进一步来说，如果知道数据项存在于cache中，又该如何找到这个数据项呢？这两个答案是有关联的。如果每个字能够位于cache中的确定位置，只要它在cache中，那么找到它则是很简单的。在cache中为每个存储中的数据字进行位置分配的最简单方式，就是基于它在存储中的地址来分配cache中的位置。这种cache结构被称为**直接映射**（direct mapped），这是因为每个存储地址都被直接映射到cache中的确定位置。直接映射cache中，存储地址和cache位置之间的典型映射关系通常都非常简单。例如，几乎所有的直接映射

> **直接映射cache**：一种cache结构，其中每个存储地址都映射到cache中的确定位置。

cache 使用下述映射方法来找到对应的数据块：

（块地址）mod（cache 中的数据块数量）

如果 cache 的块数是 2 的幂，则取模运算很简单，只需要取地址的低 N 位即可。其中 $N = \log_2$(cache 的块数)。因此，一个 8 个数据块的 cache 使用地址的最低 3 位来查找。例如，图 5-8 中给出 1_{10}（00001_2）至 29_{10}（11101_2）之间的地址如何映射到容量为 8 个字的直接映射 cache 中的位置 1_{10}（001_2）和位置 5_{10}（101_2）。

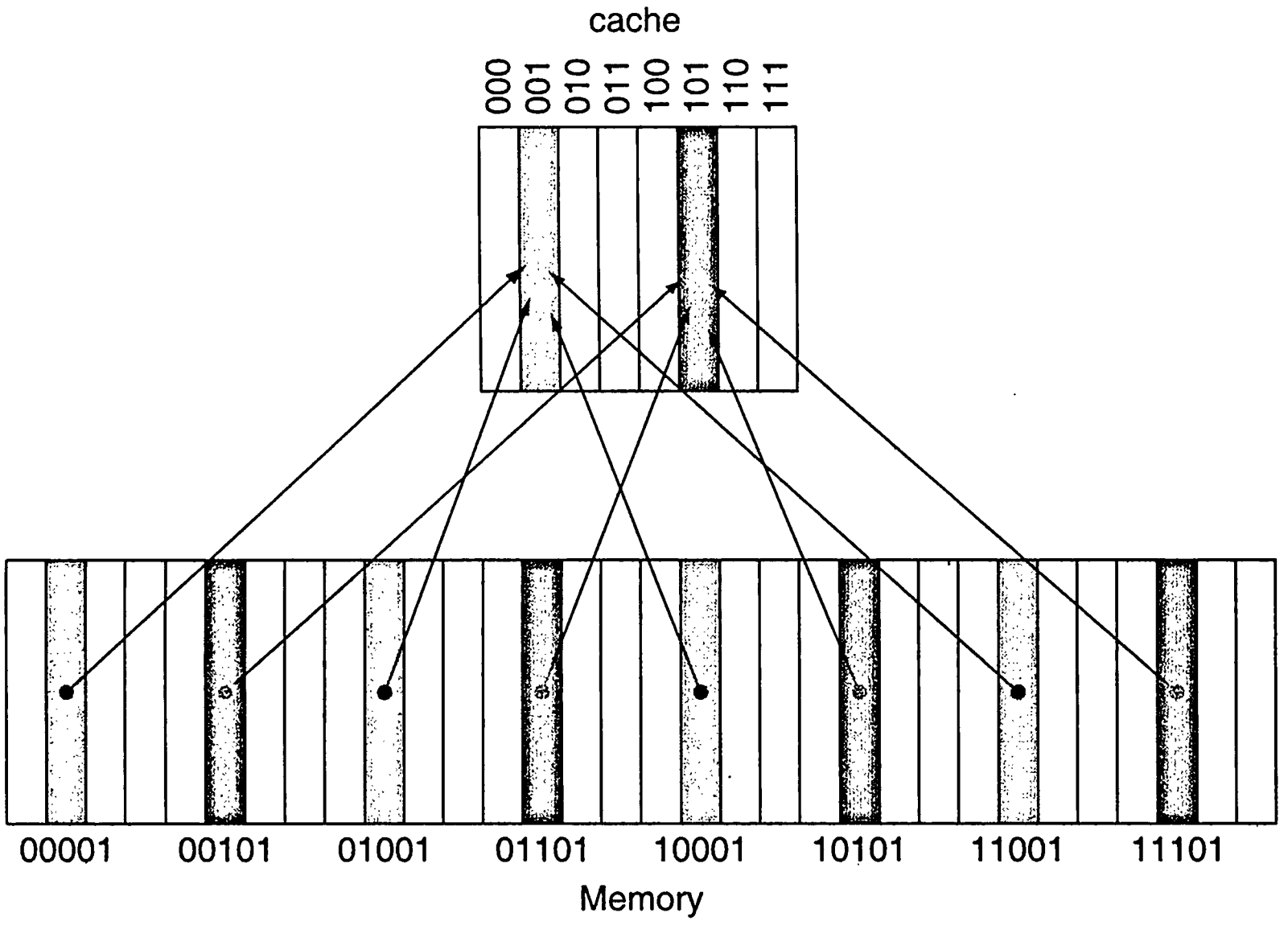

图 5-8　一个 8 个字的直接映射 cache，0 到 31 之间的存储字地址映射到相同的 cache 位置上。由于该 cache 中有 8 个字，地址 X 对应该直接映射 cache 的位置为 X mod 8。也就是说，低 3（$\log_2 8$）位被用来作为 cache 的索引（index）。因此，地址 00001_2、01001_2、10001_2 和 11001_2 都映射到 cache 的第 001_2 块，同时，地址 00101_2、01101_2、10101_2 和 11101_2 都映射到 cache 的第 101_2 块

由于每个 cache 块中能够保存不同存储地址的内容，如何知道对应请求的数据字是否在 cache 中呢？回答这个问题前，需要在 cache 中添加一组**标签**（tag）。这些标签保存了所需的地址信息，这些信息用来确定请求字是否在 cache 中。标签中只需要保存地址的高位部分，这部分地址不会用来作为 cache 的索引。例如，图 5-8 中，只需要使用 5 位地址中的高 2 位作为标签，低 3 位作为地址的索引字段用来选择数据块。按照定义，对于任何可以放入相同 cache 块中的数据字，其地址的索引域必定是对应的 cache 块号，因此标签中无须记录这些冗余的索引。

标签：存储层次中的表项位，用来记录对应请求字的地址信息，这些信息用来确定所需数据块是否在该存储层次中。

同时，还需要一种方法能够判断 cache 中的数据块中是否保存有效信息。例如，当处理器启动时，cache 中没有有效数据，标签位都是无意义的。即使在执行了许多指令后，cache 中的一些表项内容仍可能为空，如图 5-7。因此，对于这些表项的标签可以不考虑。最常用的方法就是添加**有效位**（valid bit），用来表示该表项中是否保存有效的数据。如果该位未被置位，则对应的数据块不能使用。

有效位：存储层次中的表项位，用来表示该层次的对应数据块中是否保存了有效数据。

本节的后续部分将会重点阐述 cache 如何处理读操作。通常，由于读操作无须改变 cache 内容，因此处理读操作比处理写操作要简单些。在对读操作和 cache 失效处理的基本原理进行分析后，我们将会详细讲解 cache 的写操作和真实计算机中的 cache 设计。

重点 缓存可能是预测技术中最重要的例子。该技术依赖于局部性原理，尝试在更高的存储层次中找到所需的数据，并提供了一套机制，保证当预测错误时能够从下一级存储中找到并使用正确的数据。现代计算机的 cache 预测命中率通常在 95% 以上（详见图 5-46）。

5.3.1 cache 访问

下面是对一个大小为 8 个数据块的空 cache 进行 9 次存储访问的操作序列，包括每次存储访问的具体操作。图 5-9 中给出每次失效后 cache 中的内容变化。由于该 cache 中有 8 个数据块，地址的低 3 位用来表示数据块号。

访存地址（十进制表示）	访存地址（二进制表示）	cache命中或失效	分配的cache数据块（存放数据的位置）
22	10110_2	失效（图5-9b）	$(10110_2 \bmod 8) = 110_2$
26	11010_2	失效（图5-9c）	$(11010_2 \bmod 8) = 010_2$
22	10110_2	命中	$(10110_2 \bmod 8) = 110_2$
26	11010_2	命中	$(11010_2 \bmod 8) = 010_2$
16	10000_2	失效（图5-9d）	$(10000_2 \bmod 8) = 000_2$
3	00011_2	失效（图5-9e）	$(00011_2 \bmod 8) = 011_2$
16	10000_2	命中	$(10000_2 \bmod 8) = 000_2$
18	10010_2	失效（图5-9f）	$(10010_2 \bmod 8) = 010_2$
16	10000_2	命中	$(10000_2 \bmod 8) = 000_2$

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a）上电后的cache初始状态

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_2	Memory (10110_2)
111	N		

b）地址（10110_2）访问失效处理后

图 5-9　每次访问请求失效后 cache 中的内容，其中索引和标签使用二进制表示。cache 初始化为空，所有有效位（V）为无效（N）。处理器按照下列地址发出请求：10110_2（失效），11010_2（失效），10110_2（命中），11010_2（命中），10000_2（失效），00011_2（失效），10000_2（命中），10010_2（失效），10000_2（命中）。图中给出了该访存序列每次失效处理后 cache 的情况。当地址 10010_2（18）访问结束后，地址 11010_2（26）对应的数据项被替换，后续对地址 11010_2 的访问将引发失效。数据块的标签位仅记录地址的高位部分。存储地址中包含有 cache 数据块号 i、标签位 j，则对于该 cache 存储地址为 $j \times 8 + i$，或者等同于标签位 j 与索引位 i 的拼接。例如，上述的 cache f 中，索引 010_2 拼接上标签 10_2，对应地址为 10010_2

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_2	Memory (11010_2)
011	N		
100	N		
101	N		
110	Y	10_2	Memory (10110_2)
111	N		

c）地址（11010_2）访问失效处理后

Index	V	Tag	Data
000	Y	10_2	Memory (10000_2)
001	N		
010	Y	11_2	Memory (11010_2)
011	N		
100	N		
101	N		
110	Y	10_2	Memory (10110_2)
111	N		

d）地址（10000_2）访问失效处理后

Index	V	Tag	Data
000	Y	10_2	Memory (10000_2)
001	N		
010	Y	11_2	Memory (11010_2)
011	Y	00_2	Memory (00011_2)
100	N		
101	N		
110	Y	10_2	Memory (10110_2)
111	N		

e）地址（00011_2）访问失效处理后

Index	V	Tag	Data
000	Y	10_2	Memory (10000_2)
001	N		
010	Y	10_2	Memory (10010_2)
011	Y	00_2	Memory (00011_2)
100	N		
101	N		
110	Y	10_2	Memory (10110_2)
111	N		

f）地址（10010_2）访问失效处理后

图 5-9 （续）

由于 cache 初始为空，许多地址的首次访问都为失效。图 5-9 中描述了每一次访存后的具体操作。在第 8 次访存后产生了数据块的地址冲突。地址 18（10010_2）的数据字应放入 cache 的数据块 2（010_2）中。因此，它需要将已经在数据块 2 中的数据，也就是地址 26（11010_2）的数据替换掉。这个策略使得 cache 可以有效利用时间局部性：使用最近访问的数据替换最近不常访问的数据。

这种情况类似于，从书架上取下一本所需书籍，但你的书桌上没有多余的地方可放，必须将一些已经在书桌上的书重新放回书架。在直接映射 cache 中，只有一个位置来存放最新访问的数据项，因此替换项也只有一个选择。

对于每一个可能的地址，都需要在 cache 中进行如下查找：使用地址低位找到对应的唯一 cache 数据块。图 5-10 显示了访存地址被划分为：

- 标签字段：用来和 cache 中存放数据的标签位进行比较。
- 索引字段：用来选择数据块。

cache 数据块的索引和标签唯一确定了应放于该块的数据字的存储地址。由于索引字段被用来作为访问 cache 的地址，而一个 n 位的二进制数有 2^n 个数值，因此直接映射 cache 的数据块总数应为 2 的幂。数据字的地址是 4 字节对齐的，每个地址的最低两位用来表示对应的字节地址。因此，如果存储都是字对齐的，那么在访问数据字时地址的最低两位可以忽略不计。本节中，假设存储中的数据都是对齐存放，在“详细阐述”中会讨论如何处理 cache 的非对齐访问。

访问 cache 所需的所有数据，是 cache 容量和存储地址大小的函数。这是因为 cache 中既保存了数据，也保存了标签。其中，单个数据块的大小是 4 字节（单字），但是通常都会比它大。对于如下情况：

- 64 位地址。
- 直接映射 cache。
- cache 大小为 2^n 数据块，因此索引字段为 n 位。
- 数据块大小为 2^m 个单字（2^{m+2} 字节），因此在单个数据块中使用 m 位来索引单字，使用地址的最低 2 位来索引字节。

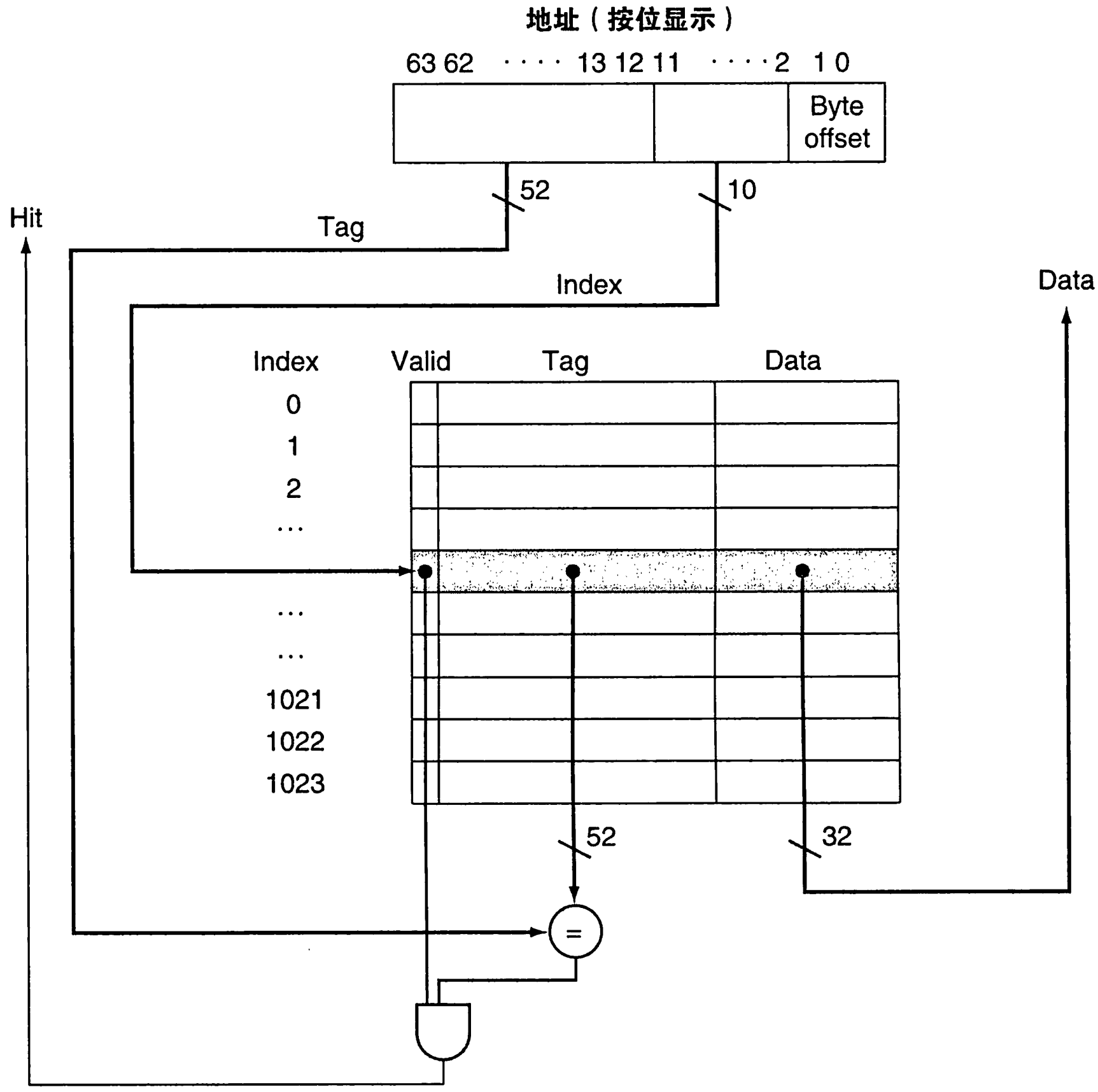

图 5-10　对于上述 cache，地址低位用来选择 cache 的数据块，该数据块包括数据和标签。上述 cache 容量为 1024 个单字（4KiB）。在本章中，我们假设使用 64 位地址。cache 中的标签需要与地址的高位进行比较，以判断请求所需的数据是否在 cache 中。由于 cache 容量为 2^{10}（或 1024）单字，而每个数据块大小为单字，因此使用 10 位地址来索引 cache，剩余的 64−10−2 = 52 位与标签进行比较。如果某数据块的标签与地址的高 52 位相等，同时对应的有效位有效，则该访存请求在 cache 中命中，处理器所需数据将被读出。否则，出现 cache 失效

标签字段的大小为：

$$64-(n+m+2)$$

该直接映射 cache 的总容量为：

$$2^n \times (\text{单个数据块容量} + \text{标签字段大小} + \text{有效位大小})$$

由于数据块的大小为 2^n 单字（2^{m+5} 位），使用 1 位来表示有效位，因此以上 cache 的容量大小为：

$$2^n \times (2^m \times 32 + (64-n-m-2) + 1) = 2^n \times (2^m \times 32 + 63 - n - m)$$

虽然这是 cache 的真实容量，但 cache 命名规范中一般只考虑数据的大小，并不考虑标签和有效位的大小。因此，图 5-10 中的 cache 被称为 4KiB cache。

例题 | cache 的容量

假设 64 位的存储地址，对于直接映射 cache，如果数据大小为 16KiB，每个数据块为 4 字大小。该 cache 容量多大（用 bit 表示）？

答案 | 已知 16KiB 为 4096（2^{12}）个字。如果单个数据块的大小为 4 字，则共有 1024（2^{10}）个数据块。每个数据块有 4×32 或 128 位数据，并加上标签和有效位，其中标签大小为（64–10–2–2）位。因此，完整的 cache 容量为：

$$2^{10} \times (4 \times 32 + (64 - 10 - 2 - 2) + 1) = 2^{10} \times 179 = 179\ \text{Kib}$$

或数据大小为 16KiB 的 cache 实际总容量为 22.4KiB。对于 cache 来说，总容量是用于数据存储的容量的 1.4 倍。

例题 | 地址映射至多字大小的 cache 块

假设该 cache 有 64 个基本块，每个基本块的大小为 16 字节。对于字节地址 1200，会映射到哪个基本块呢？

答案 | 据 5.3 节开始处的公式，块号的确定是：

（块地址）mod（cache 中的数据块数量）

其中，

$$\text{块地址 A} = \frac{\text{字节地址}}{\text{每块中的字节数}}$$

注意，这个块地址中包含了所有从（[A]× 每块字节数）到（[A]× 每块字节数 +（每块字节数 −1））的地址。（注：[A] 表示对 A 进行取整。）

因此，如果每块中的字节数为 16，则字节地址 1200 对应的块地址为

$$\frac{1200}{16} = 75$$

该块地址映射到的 cache 块号为（75 mod 64）= 11。事实上，从 1200 到 1215 的字节地址都映射到这个块号。

容量更大的块可以通过挖掘空间局部性来降低失效率。图 5-11 中，随着块大小的增长，失效率通常都在下降。如果单个块大小占 cache 容量的比例增加到一定程度，失效率最终会随之上升。这是因为 cache 中可存放的块数变少了。最终，某个数据块会在它的大量数据被访问之前就被挤出 cache。另一方面，对于一个较大的数据块，块中各字之间的空间局部性也会随之降低，失效率降低带来的好处也就随之减少。

增大块容量时，另一个更严重的问题是失效损失。失效损失是从下一级存储获得数据块并加载到 cache 的时间。该时间分为两部分：访问命中的时间和数据的传输时间。很明显，除非我们改变存储系统，否则传输时间（也可以说是失效损失）将随着数据块容量的增大而增大。而且，随着数据块容量的增大，失效率改善带来的收益开始降低。最终的结果是，失效损失增大引起的性能下降超过了失效率降低带来的收益，cache 性能当然随之下降。当然，

如果能设计出高效的传输大数据块的存储，就可以增大数据块的容量，并得到更大的性能改善。我们将在下一节讨论这个问题。

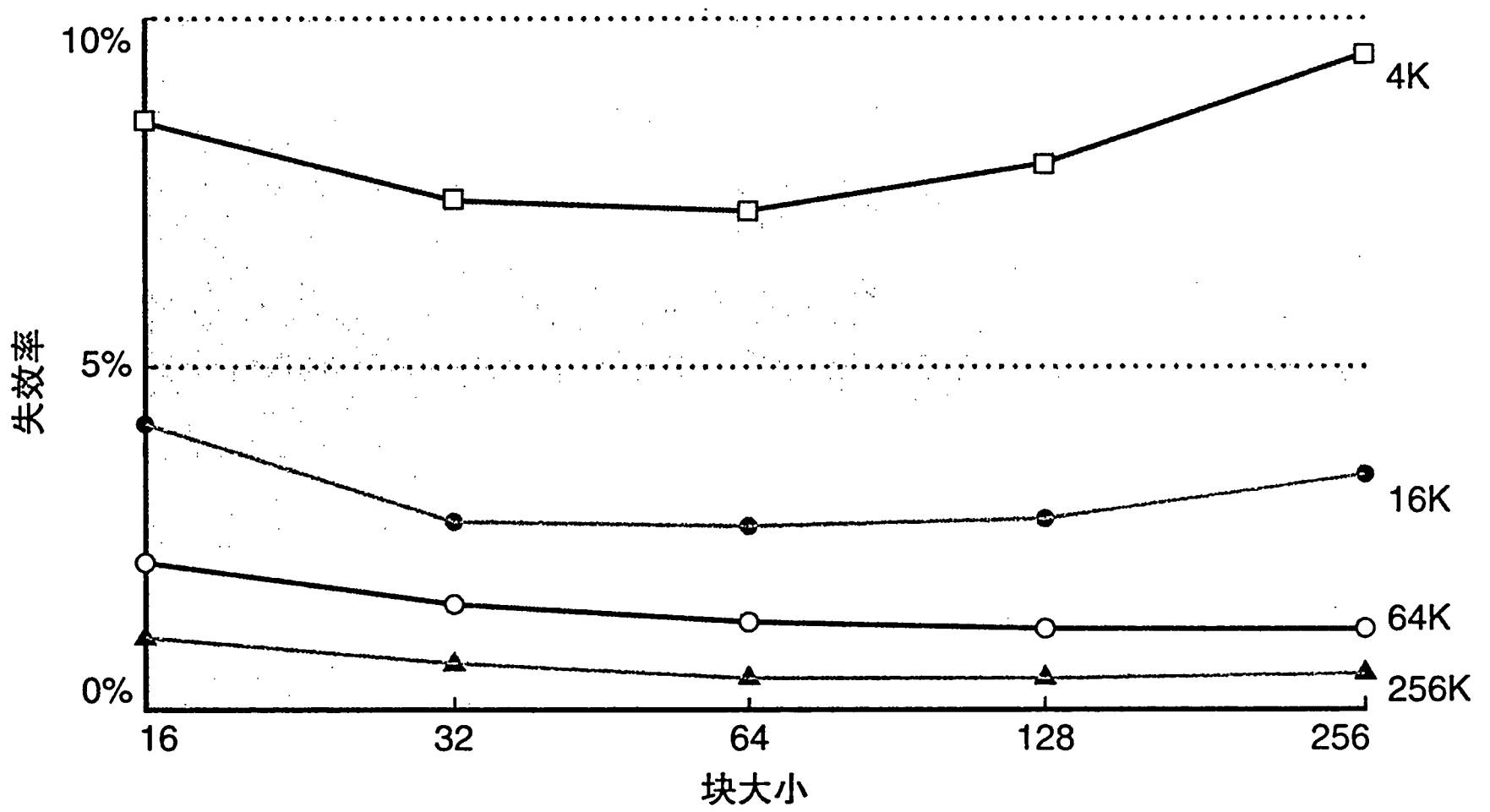

图 5-11　失效率与块容量。注意，如果相对于 cache 大小，块容量过大，实际上失效率会逐渐上升。图中的每条折线表示不同容量的 cache（本图与相联度无关，该内容之后讨论）。不走运的是，如果包含了块容量，运行 SPEC CPU2000 的时间过长，因此图上这些数据都是基于 SPEC92 的

详细阐述　虽然对于大数据块很难解决失效损失中的长延迟问题，但可以隐藏部分传输时间来有效地减少失效损失。最早采用该思想的技术称为“提早重启”（early restart），即只要数据块中的所需数据返回来，就继续执行，无须等到该数据块中所有数据都完成传输。许多处理器将该技术用于指令访问，效果显著。取指一般都是顺序的，因此，如果存储系统能够每周期传输一个字，且能及时传输新指令字，那么当所需数据返回时处理器就能重启操作。对于数据 cache，该技术通常效果不好。这是因为所需数据的访问顺序很难预测，在数据传输完成之前，下一个所需数据来自另一个数据块的概率比较大。如果由于当前数据传输未完成导致处理器不能继续访问数据 cache，那么流水线就必须停顿。

另一种更为复杂的方案是，重新组织存储，让所需的数据首先从存储传输到 cache 中，再继续传输数据块中的剩余部分。从所需数据之后的地址开始，直到该数据块的开头。该技术被称为请求字先行（requested word first）或关键字先行（critical word first），比“提早重启”技术性能稍好。但与“提早重启”技术相同，会因为同样的问题而受到限制。

5.3.2　处理 cache 失效

在考虑真实系统中的 cache 之前，讨论一下控制单元如何处理 **cache 失效**（5.9 节将详细描述 cache 控制器）。控制单元必须能够检测到失效，然后通过从内存（或者，按照我们的理解，从下一级 cache）中取得所需的数据来处理失效。如果 cache 命中，计算机将继续使用数据，就像什么都没有发生过一样。

cache 失效：由于所需数据不在 cache 中，对 cache 发出的数据请求不能被响应。

当 cache 命中时，对处理器的控制逻辑进行修改并没有那么重要。不过，cache 失效时，则需要一些额外的工作。cache 的失效处理与两部分协同工作：一部分是处理器的控制单元；

另一部分是单独的控制器，用来初始化内存访问和重填 cache。cache 的失效处理会引发流水线的停顿（具体见第 4 章），这与例外或者中断处理不同，后者需要保存所有寄存器的状态。cache 失效将会停顿整个处理器来等待内存（返回数据），特别是冻结临时寄存器和程序员可见寄存器的内容。更为复杂的是，乱序执行的处理器在等待 cache 失效处理时允许继续执行指令。不过，本节中的按序处理器都假设在 cache 失效时停顿流水线。

进一步仔细考虑如何处理指令失效，相同的方法可以方便地扩展到处理数据失效。如果一条指令访问引发了失效，那么指令寄存器的内容将被置为无效。为了将正确的指令写入 cache 中，必须能够对下一级存储发出读操作。由于程序计数器是在执行的第一个时钟周期递增，引发指令 cache 失效的指令地址就等于程序计数器的数值减 4。一旦确定了地址，就需要指导主存进行读操作。等待内存响应（因为该访问将耗费多个时钟周期），然后将含有所需指令的（指令）字写入指令 cache 中。

一旦发生指令 cache 失效，可以定义如下处理步骤：

1. 将 PC 的原始值（当前 PC−4）发送到内存。

2. 对主存进行读操作，等待主存完成本次访问。

3. 写 cache 表项，将从内存获得的数据写入到该表项的数据部分，将地址的高位（来自于 ALU）写入标签字段，并将有效位置为有效。

4. 重启指令执行。这将会重新取指，本次取指将在指令 cache 中命中。

与上述相比，数据访问的 cache 控制本质上是相同的。一旦失效，简单地暂停处理器，直到内存返回数据。

5.3.3 处理写操作

写操作有一些不同。例如，对于存储指令，只把数据写入数据 cache（不需要改变主存）。完成写入 cache 的操作后，主存中的数据将和 cache 中的数据不同。在这种情况下，cache 和主存称为不一致（inconsistent）。保持 cache 和主存一致的最简单方法是，总是将数据写回内存和 cache。这样的写策略称为**写穿透**或者**写直达**（write-through）。

写穿透或写直达：一种写策略。写操作总是同时更新 cache 和下一级存储，保证两者之间的数据一致。

写操作的另一个关键点是写失效的处理。先从主存中取来对应数据块中的数据，之后将其写入 cache 中，覆盖引发失效的数据块中的数据。同时，也会使用完整地址将数据写回主存。

虽然上述设计方案能够简单地处理写操作，但是它的性能不佳。基于写穿透策略，每次的写操作都会引起写主存的操作。这些写操作延时很长，至少 100 个处理器时钟周期，这会大大降低处理器的性能。例如，假设 10% 的指令是存储指令（store）。如果不发生 cache 失效，处理器的 CPI 为 1。每次写操作需要 100 个处理器时钟周期。这会导致 CPI 变为 $1.0+100 \times 10\% = 11$，性能降低为原来的 1/10。

解决这个问题的方法之一是使用**写缓冲**（write buffer）。写缓冲中保存着等待写回主存的数据。数据写入 cache 的同时也写入写缓冲中，之后处理器继续执行。当写入主存的操作完成后，写缓冲中的表项将被释放。如果写缓冲满了，处理器必须停顿流水线直到写缓冲中出现空闲表项。当然，如果主存写操作的速率小于处理器产生写操作的速率，多大容量的缓冲都无济于事。因

写缓冲：一个保存等待写入主存的数据的队列。

为写操作的产生速度远远快于主存系统的处理速度。

即使写操作的产生速度小于主存的处理速度，还是会产生停顿。通常，当写操作成簇（burst）发生时，会发生上述现象。为减少这样的停顿发生，处理器通常会增多写缓冲的表项数。

*相对于写穿透或者写直达策略，另一种写策略称为*写返回（write-back）。基于写返回策略，当发生写操作时，新值只被写入cache中，被改写的数据块在替换出cache时才被写到下一级存储。写返回策略能够改善性能，尤其是当处理器写操作的产生速度等于或大于主存的处理速度时。不过，写返回策略的实现比写穿透策略要复杂得多。

写返回：一种写策略。处理写操作时，只更新cache中对应数据块的数值。当该数据块被替换时，再将更新后的数据块写入下一级存储。

在本章的后续部分中，将会阐述真实处理器中的cache，详细说明它们是如何处理读操作和写操作的。在5.8节中，还会更加详细地描述写操作的处理。

详细阐述 相对于读操作来说，写操作为cache引入了更多的复杂性。在此讨论其中的两点：写失效策略和写返回策略cache中写操作的高效实现。

考虑写穿透cache中的写失效处理。最常见的策略是，在cache中为其分配一个数据块，称为写分配（write allocate）。将该数据块从内存取入cache中，改写该数据块中相应部分的数据。另一种策略则是，在内存中更新相应部分的数据，并不将其取入cache。这种策略称为写不分配（no write allocate）。该策略的动机是，有时候程序需要写整个数据块，例如操作系统对某一页内存进行初始操作（全部写0）。在这样的情况下，初始写失效就将对应数据块取至cache里，这样的操作是不必要的。有些处理器允许以页为粒度来修改写分配策略。

事实上，在写返回cache中实现高效的写操作比在写穿透cache中要复杂得多。写穿透cache中，可以在比对标签的同时就将数据写入cache。如果标签比对不符，发生cache失效。由于cache是写穿透的，对应数据块的改写不会产生严重后果，因为主存和cache中的数据都是正确的。但是，在写返回cache中，如果cache中的数据已被修改并产生cache失效，必须先将对应数据块写回内存。如果在处理存储指令时，在不知道cache是否命中之前（就像对待写穿透cache那样），只是简单地改写cache中对应的数据块，那么可能会破坏数据块中的内容。这些内容还未及时备份到下一级存储中。

在写返回cache中，由于不能直接改写数据块，或者使用两个周期去处理存储指令（一个周期用来进行标签比对确认是否命中，之后再用一个周期进行真正的写操作），或者需要一个写缓冲来保存数据——通过流水化操作在一个周期内高效地处理存储指令。当使用存储缓冲器（store buffer）时，在正常的cache访问周期里，处理器进行cache查找，同时将数据放入store buffer中。如果cache命中，在下一个无用的cache访问周期里把新数据从store buffer写入cache中。

相比之下，写穿透cache中写操作可以在一个周期内完成，读取标签的同时将数据写入对应的数据块中。如果写操作的数据块地址与标签匹配，处理器继续正常执行，因为更新的是正确的数据块。如果标签不匹配，处理器产生写失效，将该地址对应的数据块剩余内容取到cache中。

写返回cache也有写缓冲。当cache发生失效替换修改过的数据块时，写缓冲可以用来减少失效损失。在这种情况下，在从主存读取所需数据块时，（被替换出去的）修改过的数据

块被放入 cache 的写返回缓冲（write-back buffer），之后再由写返回缓冲写回到主存中。如果下一个失效不会立即发生，当“脏”数据块被替换时，该技术可使失效损失减少一半。

5.3.4　cache 实例：Intrinsity FastMATH 处理器

Instrinsity 的 FastMATH 处理器是一款采用 MIPS 指令系统体系结构的嵌入式微处理器，内有一个简单的 cache 实现。在本章的结尾，将会介绍 ARM 和 Intel 处理器更为复杂的 cache 设计。根据教学规律，先从简单的、真实的样例入手。图 5-12 显示了 Intrinsity 的 FastMATH 处理器中数据 cache 的组织结构。注意，这款处理器的地址宽度是 32 位的，而非 64 位的。

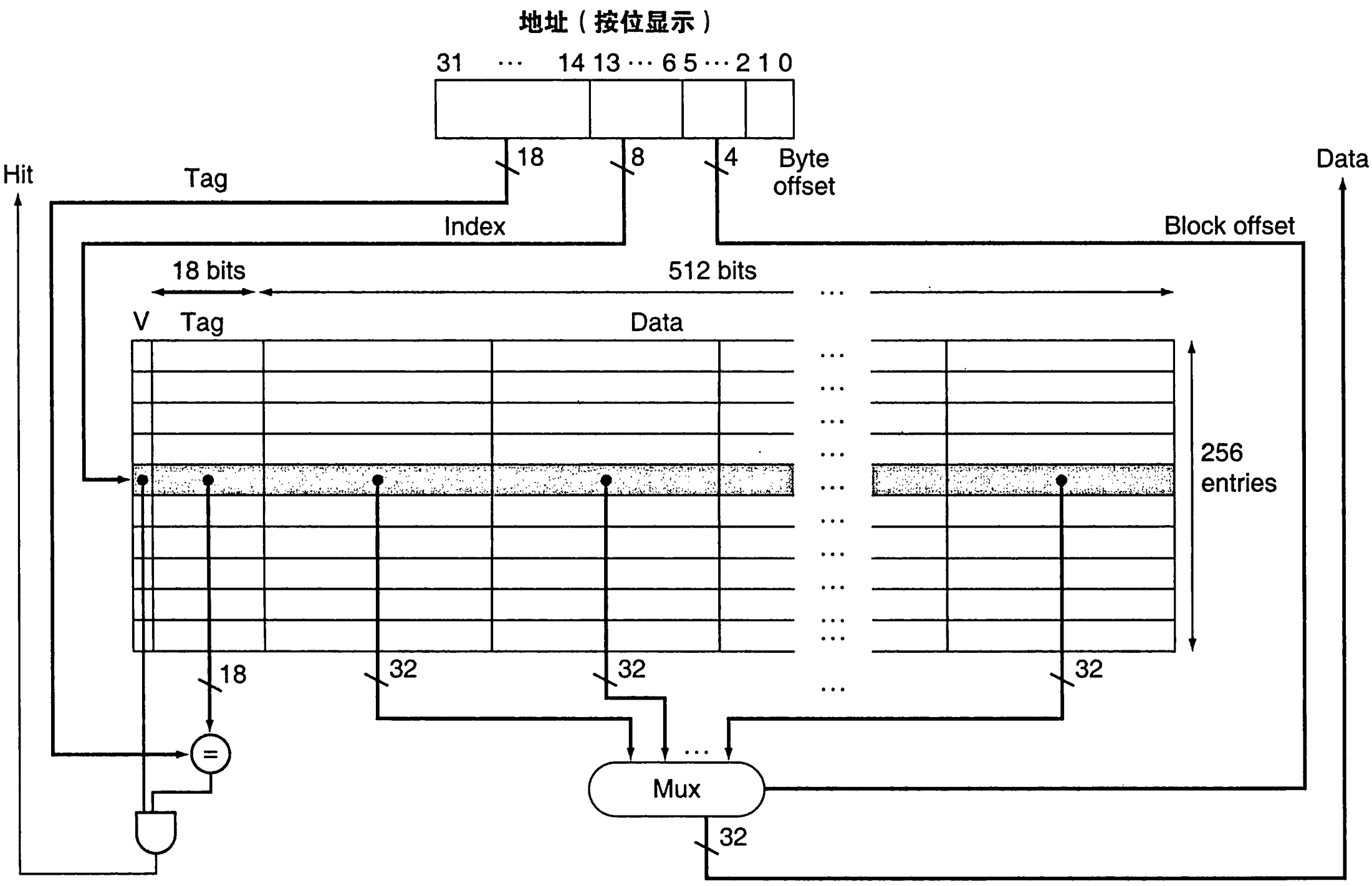

图 5-12　Instrinsity 的 FastMATH 处理器中的 16KiB cache，每个 cache 包含 256 个数据块，每个数据块有 16 个字。注意，地址宽度为 32 位。标签字段为 18 位，索引字段为 8 位，另外 4 位（5 ～ 2 位）用来在数据块中选择所需的字，可以使用 16 选 1 的多选器实现。事实上，为去掉这个多选器，cache 使用了一个独立的大容量 RAM 来存放数据，一个小容量 RAM 来存放标签。大容量的数据 RAM 所需的额外地址由块内偏移提供。这样，大容量 RAM 的字长为 32 位，字数是 cache 块数的 16 倍

该款处理器有 12 级流水线。峰值速度运行时，处理器可以在每个时钟周期同时取来一条指令和一个数据。为满足不停顿流水线的要求，指令 cache 和数据 cache 是分离的。每个 cache 为 16KiB 或 4096 字，数据块大小为 16 字。

cache 的读请求处理很简单。由于有独立的数据和指令 cache，需要为每个 cache 的读写提供单独的控制信号（记住：发生失效后指令 cache 需要更新）。因此，两个 cache 的读请求处理步骤如下：

1. 将地址发送到对应 cache。地址可能来自于 PC（读指令），也可能来自于 ALU（读数据）。

2. 如果 cache 命中，在数据块中可以找到请求的数据。由于每个数据块的容量为 16 个字，需要进行选择。使用多选器从这 16 个字中选择所需的数据，使用块索引作为多选器的控制信号（具体见图下方）。

3. 如果 cache 失效，将该地址发往主存。当主存返回数据时，将其写入 cache，（CPU）读取它并完成请求。

对于写请求，Intrinsity FastMATH 处理器提供了写穿透和写返回两种策略，由操作系统来决定为应用程序配置哪种策略，并含有 1 个表项的写缓冲。

使用像 Intrinsity FastMATH 中的 cache 结构，cache 的失效率会如何呢？图 5-13 给出了指令 cache 和数据 cache 的失效率。综合失效率（combined miss rate）指的是，考虑了指令访问和数据访问的不同频度后，每个程序的 cache 访问的失效率。

指令失效率	数据失效率	综合失效率
0.4%	11.4%	3.2%

图 5-13　使用 SPEC CPU2000 评测程序，Intrinsity FastMATH 处理器指令和数据 cache 的近似失效率。综合失效率指的是使用 16KiB 指令 cache 和 16KiB 数据 cache 时的实际失效率，通过将指令和数据失效率分别乘以指令和数据访存的频度获得

虽然失效率是 cache 设计的重要指标之一，但最重要的衡量指标仍然是存储系统对程序执行时间的影响。下面简要介绍一下失效率与执行时间之间的关系。

详细阐述　与两个分离的 cache 相比，同等容量的混合 cache 通常会有更高的命中率。原因在于，这种混合的 cache 不会严格把用于存放指令的表项数和用于存放数据的表项数区分开来。尽管如此，目前几乎所有的处理器都使用分离的指令 cache 和数据 cache，提高 cache 的带宽以满足现代流水线的需要。（这样冲突失效也会更少，具体见 5.8 节。）

分离的 cache：某一级存储由两个独立的 cache 组成，一个处理指令访存，一个处理数据访存，两者可以同时被访问。

下面是 Intrinsity FastMATH 处理器的 cache 配置，与一个同等容量的混合 cache 的失效率比较。

- 所有 cache 容量：32 KiB
- 分离的 cache 的失效率：3.24%
- 混合 cache 的失效率：3.18%

分离 cache 的失效率比混合 cache 的失效率略差。

通过同时支持指令和数据访问，cache 带宽加倍，但这一好处很容易被增高的失效率抵消。这提醒我们不能使用失效率作为衡量 cache 性能的唯一指标，具体见 5.4 节。

5.3.5　总结

本章从分析一个最简单的 cache 开始：直接映射，数据块容量为一个字。在这样的 cache 中，命中和失效都很简单，因为一个字正好一个数据块，每个字都有独立的标签。为保持 cache 和主存数据的一致性，可以使用写穿透的策略，这样每次对 cache 的写操作都会引发对主存的更新。相对写穿透，另一种策略是写返回，当数据块被替换时，才将其写入到主存中。这些将会在后续的章节中进行讨论。

为利用空间局部性，cache 的数据块容量应大于一个字。使用更大容量的数据块将会降低失效率，减少 cache 中与数据存储相关的标签存储，从而提高 cache 的效率。虽然更大的数据块容量可以降低失效率，但也会增加失效损失。如果失效损失随着数据块容量呈线性增长，那么更大的数据块容量很容易导致更低的性能。

为避免性能损失，加大主存带宽来更高效地传输 cache 数据块。通常加大 DRAM 带宽的方法是加宽主存和交叉访问。DRAM 设计者不断改善处理器和主存之间的接口，加大簇发传输模式下两者之间的带宽，减少大容量 cache 数据块的开销。

自我检测　存储系统的速度影响了设计者对于 cache 数据块容量的判断。针对 cache 设计者，下面哪一条准则通常是有效的？

1. 存储延迟越短，cache 的数据块容量越小。
2. 存储延迟越短，cache 的数据块容量越大。
3. 存储带宽越高，cache 的数据块容量越小。
4. 存储带宽越高，cache 的数据块容量越大。

5.4　cache 的性能评估和改进

在本节中，先从评估和分析 cache 性能的方法开始，之后介绍两种不同的改善 cache 性能的技术。第一项技术主要关注通过减少两个不同的内存块争夺同一缓存位置的发生概率来降低失效率。第二项技术是通过添加额外的一个存储层次来减少失效代价。这项技术被称为*多级缓存*（multilevel caching），出现在 1990 年售价高达 10 万美元的高端计算机中。此后，该技术被运用到售价仅需几百美元的个人移动设备中。

CPU 时间可被分成 CPU 用于执行程序的时间和 CPU 用来等待访存的时间。通常，假设 cache 命中的访问时间只是正常 CPU 执行时间的一部分。因此，

$$\text{CPU 时间} = (\text{CPU 执行的时钟周期数} + \text{等待存储访问的时钟周期数}) \times \text{时钟周期}$$

假设等待存储访问的时钟周期数主要来自于 cache 失效，同时，限制后续的讨论只针对简单的存储系统模型。在真实的处理器中，读写操作产生的停顿十分复杂，准确的性能预测通常需要对处理器和存储系统进行非常详细的模拟。

等待存储访问的时钟周期数可以被定义为，读操作带来的停顿周期数加上写操作带来的停顿周期数：

$$\text{等待存储访问的时钟周期数} = \text{读操作带来的停顿周期数} + \text{写操作带来的停顿周期数}$$

读操作带来的停顿周期数可以由每个程序的读操作次数、读操作失效率和读操作的失效代价来定义。

$$\text{读操作带来的停顿周期数} = \frac{\text{读操作数目}}{\text{程序}} \times \text{读失效率} \times \text{读失效代价}$$

写操作要更复杂些。对于写穿透策略，有两个停顿的来源：一个是写失效，通常在连续写之前需要将数据块取回（具体见 5.3.3 节的详细阐述部分）；另一个是写缓冲停顿，通常在写缓冲满时进行写操作会引发该停顿。因此，写操作带来的停顿周期数等于下面两部分的总和：