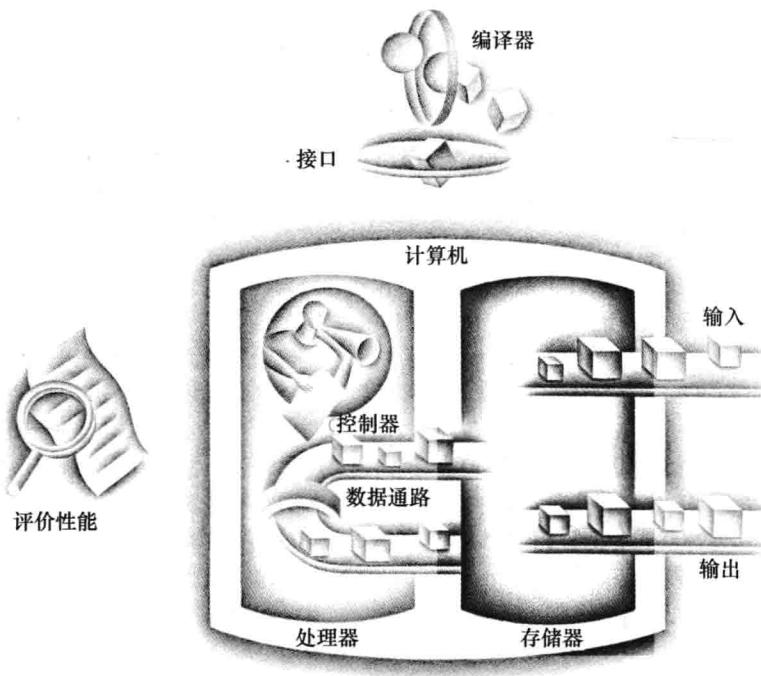


第5章 |

Computer Organization and Design: The Hardware/Software Interface

大容量和高速度：开发存储器层次结构



计算机的 5 个经典部件

在理想情况下，我们希望存储器容量可以无限大，这样，在任何特定的情况下……都可以立刻得到需要用到的字……在实际中，我们需要构建一个具有层次结构的存储器，其中的每一层都比它的上一层拥有更大的容量，但访问速度更慢。

—— A. W. Burks, H. H. Goldstine 和 J. von Neumann

《Preliminary Discussion of the Logical Design of an Electronic Computing Instrument》, 1946

5.1 引言

从最早期的计算开始，程序员就希望快速存储器的容量可以无限大。这一章主要探讨如何帮助程序员构建一个容量无限大的虚拟快速存储器。在这之前，让我们通过简单的类比方式来介绍将要使用的关键原理和机制。

假如你正在完成一份关于计算机硬件重要历史性发展的论文，你可以从图书馆的书架上精心挑选一些经典计算机书籍，并将它们放在书桌上。你从这些书中找到了需要写的几种重要的计算机，但是没有找到关于 EDSAC 的，因此，你返回书架去寻找其他书，并在早期的英国计算机书籍中找到了一本有关 EDSAC 的书。一旦在你的书桌上有了选好的一些书，你就有可能从这些书中找到你需要的许多内容，这样一来，你的大部分时间只需花在阅读这些书上，而无需返回书架。试比较这两种情况：一种是在你的书桌上有好几本书；另一

种是书桌上只有一本书，你不得不频繁地返回书架，进行还书后取另一本书。在书桌前放一些书会更节省时间。

同样，我们可以构建一个大容量的虚拟存储器，它能像小容量的存储器那样被快速访问。就像你不会同时以相同的概率查阅图书馆中的每一本书那样，一个程序也不会同时以相同的概率访问它全部的代码或数据。否则，不可能让存储器在保持大容量的同时又能快速访问，就像不能既要求你把图书馆中所有的图书放在书桌上，还要求你能保持快速查找一样。

局部性原理不仅适用于在图书馆查找资料的工作方式，而且适用于程序执行的方式。局部性原理表明了在任何时间内，程序访问的只是地址空间相对较小的一部分内容。以下是两种不同类型的局部性：

- **时间局部性** (temporal locality)：如果某个数据项被访问，那么在不久的将来它可能再次被访问。就如刚拿了一本书到书桌上查阅，那么很可能你会很快地再次查阅它。
- **空间局部性** (spatial locality)：如果某个数据项被访问，与它地址相邻的数据项可能很快也将被访问。例如，当你找到一本关于 EDSAC 的早期经典计算机的书籍时，也许紧挨着它的另一本关于早期工业计算机的书籍里同样有你所需的材料，因为图书馆通常将主题相同的书放在同一个书架上以提高空间定位效率。后面我们将看到空间局部性原理如何应用于存储器层次结构。

- ② **时间局部性**：某个数据项在被访问之后可能很快被再次访问的特性。
- ② **空间局部性**：某个数据项在被访问之后，与其地址相近的数据项可能很快被访问的特性。

正如查阅书桌上的资料体现了自然的局部性，程序的局部性起源于简单自然的程序结构。例如，大多数程序都包含了循环结构，因此这部分指令和数据将被重复地访问，呈现出了很高的时间局部性。由于指令通常是顺序执行的，因此程序也呈现了很高的空间局部性。对数据的访问同样显示了一种自然的空间局部性。例如，对数组或者记录中的元素进行顺序访问都体现了高度的空间局部性。

我们可以利用局部性原理将计算机存储器组织成为**存储器层次结构** (memory hierarchy)。存储器层次结构由不同速度和容量的多级存储器构成。快速存储器每比特的成本要比慢速存储器高很多，因而通常它们的容量也比较小。

- ② **存储器层次结构**：一种由多存储器层次组成的结构，存储器的容量和访问时间随着离处理器距离的增加而增加。

如图 5-1 所示，较快的存储器靠近处理器，而较慢的、便宜的存储器层次较低。其目的是以最低的价格向用户提供尽可能大的存储容量，同时存取速度与最快的存储器相当。

同样，数据也可以组织成层次化结构：靠近处理器那一层中的数据是那些较远层次中的子集，所有的数据则被存在最慢的底层。我们依然使用图书馆的例子来进行类比，书桌上的书籍是图书馆藏书的一个子集，进而也是学校中所有图书馆藏书的一个子集。而且，离处理器越远的层次访问时间也越长，就像我们在学校图书馆系统中可能遇到的情况一样。

存储器层次结构可以由多层构成，但是数据每次只能在相邻的两个层次之间进行复制。因此我们将注意力重点集中在两个层次上。高层的存储器靠近处理器，比低层存储器容量小但访问速度更快，这是因为它采用了成本更高的技术来实现。如图 5-2 所示，我们将一个两级层次结构中存储信息交换的最小单元称为块 (block) 或行 (line)，就像在图书馆中，一个信息块就是一本书。

速度	处理器	尺寸	价格(美元/位)	当前技术
最快	存储器	最小	最高	SRAM
	存储器			DRAM
最慢	存储器	最大	最低	磁盘

图 5-1 存储器层次的基本结构。存储系统采用层次结构后，用户对于存储器的认识就是：它的容量和容量最大的那层存储器相同，而访问速度和最快的那层存储器相当。在很多个人移动终端设备中，闪存已经代替了磁盘，对于台式计算机和服务器来说可能会在存储器层次中引入新的一层；见 5.2 节

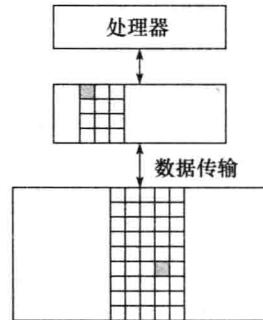


图 5-2 存储器层次结构中的每两个层次可以被认为一个是高层次，一个是低层次。在每一层中，存储信息交换的最小单元称为块或者行。通常在层次之间复制时按整块进行传输

如果处理器需要的数据存放在高层存储器中的某个块中，则称为一次命中（这就好像正好从书桌上的一本书中找到所需的信息一样）。如果在高层存储器中没有找到所需的数据，这次数据请求则称为一次缺失。随后访问低层存储器来寻找包含所需数据的那一块（如同从书桌旁走到书架前去寻找所需的书籍）。命中率 (hit rate) 或命中比率 (hit ratio)，是在高层存储器中找到数据的存储访问比例，通常被当成存储器层次结构性能的一个衡量标准。缺失率 (miss rate) ($1 - \text{命中率}$) 则是在高层存储器中没有找到数据的存储访问比例。

追求高性能是我们使用存储器层次结构的主要目的，因而命中时间和缺失时间就显得尤为重要。命中时间 (hit time) 是指访问存储器层次结构中的高层存储器所需要的时间，包括了判断当前访问是命中还是缺失所需的时间（相当于浏览书桌上书籍所花费的时间）。缺失代价 (miss penalty) 是将相应的块从低层存储器替换到高层存储器中，以及将该信息块传送给处理器的时间之和（也就是从书架上取另一本书并将它放到桌上的时间）。由于较高存储器层次容量较小并且使用了快速的存储器部件，因此比起对存储器层次中较低层的访问，命中时间要少得多，这也是缺失代价的主要组成部分。（同样，查找书桌上书籍的时间比站起来到书架前查找一本新书所需的时间要少得多。）

- ② 块或行：可存在于或不存在于 cache 中的信息的最小单元。
- ③ 命中率：在高层存储器中找到目标数据的存储访问比例。
- ④ 缺失率：在高层存储器中没有找到目标数据的存储访问比例。
- ⑤ 命中时间：访问某存储器层次结构所需要的时间，包括了判断当前访问是命中还是缺失所需的时间。
- ⑥ 缺失代价：将相应的块从低层存储器替换到高层存储器所需的时间，包括访问块、将数据逐层传输、将数据插入发生缺失的层和将信息块传送给请求者的时间。

在这一章中我们也将看到，用来构建存储器层次结构的这些概念将影响计算机的许多其他方面，包括操作系统如何管理存储器和 I/O，编译器如何产生代码，甚至对应用程序如何使用计算机也产生一定影响。当然，由于所有程序花费大量时间访问存储器，因而存储系统必然成为评估机器性能的一个主要指标。利用存储器层次结构来达到性能的提升，意味着在过去程序员可以把存储器看成是一个线性的随机访问的存储设备，而现在必须理解存储器层次结构如何

工作才能获得良好的性能。稍后我们将在图 5-18 的示例中说明其重要性，在 5.14 节说明如何使矩阵乘法加倍性能。

由于存储系统对性能至关重要，计算机设计人员在这些系统上花费了大量精力，并致力于开发复杂的机制来提高存储系统的性能。本章我们主要讨论概念性的观点，为了不至于使篇幅过长和使内容太复杂，对许多概念进行了简化和抽象。

01 重点 程序不仅表现出时间局部性，即重复使用最近被访问的数据项的趋势，同时也表现出了空间局部性，即访问与最近被访问过的数据项地址空间相近的数据项的趋势。存储器层次结构利用了时间局部性，将最近被访问的数据项放在靠近处理器的地方；同时它也利用了空间局部性，将一些包含连续字的块移至存储器层次结构的较高层次。

如图 5-3 所示，在存储器层次结构中，离处理器越近的层次容量越小，速度越快。因此，数据在层次结构中的最高层命中能被很快处理。而缺失后，需要访问容量大但速度慢的低层存储器层次。如果命中率足够高，存储器层次结构就会拥有接近最高（而且最快）层次的访问速度和接近最低（也是最大）层次的容量。

在很多系统中，存储器是一个真实的层次结构，这意味着除非数据在第 $i+1$ 层存在，否则绝不可能在第 i 层存在。

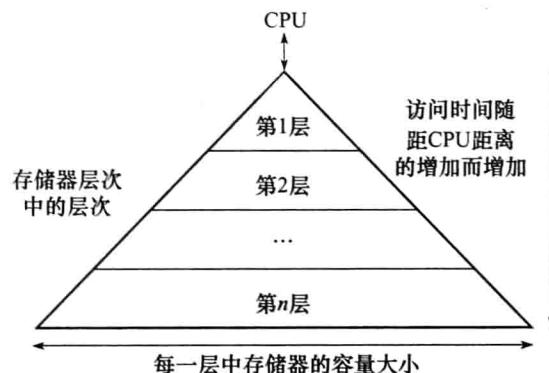


图 5-3 这幅图说明了存储器层次结构：离处理器越远，容量越大。当采用合适的操作机制时，这种结构允许处理器的访问时间主要由层次结构中的第 1 层来决定，而整个存储器的容量则和第 n 层一样大。本章的主题就是要实现这种结构。尽管本地磁盘一般位于存储器层次结构的底层，但是一些系统会使用磁带或者局域网内的文件系统作为层次结构的更下一层

01 小测验

下面哪些表述通常是正确的？

1. 存储器层次利用了时间局部性。
2. 在一次读操作中，返回的值取决于哪些块在 cache 中。
3. 存储器层次结构的大部分成本处于最高层。
4. 存储器层次结构的大部分容量处于最低层。

377

5.2 存储器技术

目前，构建存储器层次结构主要有 4 种技术。主存储器由 DRAM（动态随机存取存储器）实现，靠近处理器的那层（cache）则由 SRAM（静态随机存取存储器）来实现。DRAM 每比特成本要低于 SRAM，但是速度比 SRAM 慢。价格的差异源于 DRAM 每比特占用的存储器空间较少，因此等量的硅制造的 DRAM 的容量会比 SRAM 的要大。速度的差异则由多种因素造成，我们将在附录 B 的 B.9 节中介绍。第三种技术是闪存，这种非易失存储器用作个人移动设备中的二级存储器。第四种技术是磁盘，它通常是服务器中容量最大且速度最慢的一层。以上这些技术的访问时间和每比特的成本变化很大，如下表所示（表中使用的是 2012 年的典型数据）。

存储器技术	典型访问时间 (ns)	2012 年每 GiB 的价格 (美元)
SRAM	0.5 ~ 2.5	500 ~ 1 000
DRAM	50 ~ 70	10 ~ 20
Flash	5 000 ~ 50 000	0.75 ~ 1.00
磁盘	5 000 000 ~ 20 000 000	0.05 ~ 0.10

378

本节的余下部分将分别讲授每种存储器技术。

5.2.1 SRAM 技术

SRAM 是一种组织成存储阵列结构的简单集成电路，通常具有一个读写端口。虽然读写访问时间可能不同，但 SRAM 对任何数据访问时间都是固定的。

SRAM 不需要刷新，并且其访问时间与周期时间非常相近。为了防止读操作时信息丢失，SRAM 的一个基本存储单元通常由 6 ~ 8 个晶体管组成。在空闲模式下，SRAM 只需要最小的功率来保持电荷。

过去，在大多数 PC 和服务器系统中通常将 SRAM 芯片从它们的一级、二级，甚至三级 cache 中分离出来。由于摩尔定律的推动，当今的处理器芯片中集成了多层次的 cache，因此独立的 SRAM 芯片几乎在市场上消失了。

5.2.2 DRAM 技术

只要给 SRAM 加电，其中的数值就会保持。而在动态 RAM (DRAM) 中，存储单元使用电容保存电荷的方式来存储数据。为了对保存的电荷进行读取或写入，使用一个晶体管对该电容进行访问。因为 DRAM 存储每一位都只使用一个晶体管，所以它比 SRAM 密度要高得多，且价格也要便宜很多。由于 DRAM 在电容上保存电荷，因此不能长久地保持数据，从而必须周期性地刷新。与静态存储器 SRAM 相比，这就是将该存储结构称为动态的原因。

为了对单元进行刷新，只需要读出其内容然后写回即可。DRAM 单元中的电荷可以保持几微秒。如果 DRAM 中的每个比特位需要独立的读出后写回，则必须不停地进行刷新操作，这将导致没有时间可用于正常的访问操作。幸运的是，DRAM 采用了一种两级译码结构，可以通过在一个读周期后紧跟一个写周期的方式一次刷新一行（一行单元共用一个字线）。

图 5-4 给出了一个 DRAM 的内部组织结构，图 5-5 给出了多年来 DRAM 的密度、成本、访问时间的变化。

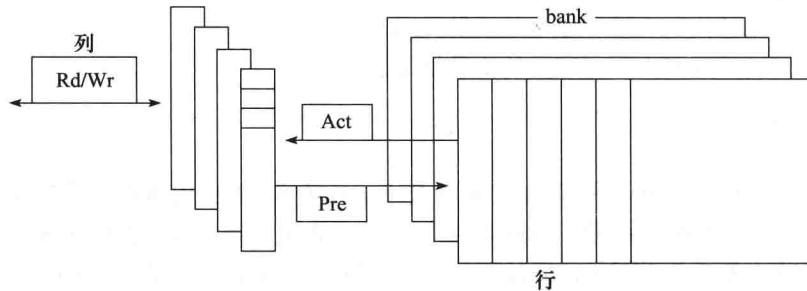


图 5-4 DRAM 的内部组织。现代 DRAM 以 bank (存储块) 方式组织，典型的 DDR3 中有 4 个 bank。每个 bank 由多个行组成。发送一条 Pre (预充电) 命令能够打开或者关闭一个 bank。使用 Act (激活) 命令发送一个行地址，将对应的行中的数据传送到一个缓冲器中。当一行数据在缓冲器中时，无论 DRAM 数据宽度（典型情况为 4、8 或 16 位）是多少，都可以通过指定要传送的数据块大小和数据块在缓冲器中的起始地址的方式连续传送相邻地址的数据。与数据块的传送一样，每条命令使用时钟进行同步。

生产年份	芯片容量	每GiB价格	访问新的 一行/一列的时间	访问已经在缓冲器 的行中的一列的时间
1980	64Kib	\$1 500 000	250ns	150ns
1983	256Kib	\$500 000	185ns	100ns
1985	1Mib	\$200 000	135ns	40ns
1989	4Mib	\$50 000	110ns	40ns
1992	16Mib	\$15 000	90ns	30ns
1996	64Mib	\$10 000	60ns	12ns
1998	128Mib	\$4 000	60ns	10ns
2000	256Mib	\$1 000	55ns	7ns
2004	512Mib	\$250	50ns	5ns
2007	1Gib	\$50	45ns	1.25ns
2010	2Gib	\$30	40ns	1ns
2012	4Gib	\$1	35ns	0.8ns

图 5-5 直到 1996 年，DRAM 芯片容量每 3 年增加为原来的 4 倍，之后增长速度下降非常快。访问时间的减少虽然很慢，但是仍然在持续减少。虽然价格受到其他诸如可用性和需求等因素的影响，但是也基本上按照存储密度增加的速度在降低。每 GiB 的价格没有按照通货膨胀进行调整

行组织结构不但有助于刷新，还有助于性能的提高。为了提高性能，DRAM 为了进行重复访问而对多行进行缓冲。缓冲器与 SRAM 类似；在下一行被访问之前，可通过改变地址来访问缓冲器中的任何一个比特位。由于访问该行中数据的时间短了很多，因此极大地减小了数据访问时间。更宽的芯片也可以增加芯片的存储器带宽。当一行数据在缓冲器中时，无论 DRAM 数据宽度（典型情况为 4、8 或 16 位）是多少，都可以通过指定要传送的数据块大小和数据块在缓冲器中的起始地址的方式连续传送相邻地址的数据。

为了进一步优化与处理器的接口，DRAM 增加了时钟，因此称之为同步 DRAM，简写为 SDRAM。SDRAM 的优势在于使用时钟对存储器和处理器保持同步。其速度上的优势主要源于不需要额外指定地址位以突发方式传送多个数据的能力，是在时钟的控制下以突发方式传送连续的数据。最快的版本称为双数据速率（DDR）SDRAM。该名称表示在时钟的上升沿和下降沿都要传送数据，因此可以获得双倍的数据带宽。该技术的最新版本是 DDR4。一个 DDR4-3200 DRAM 每秒可以传输 3 200 兆次，即其时钟频率为 1 600MHz。

要支持如此高的带宽需要在 DRAM 内部进行精心组织。与只有一个快速的行缓冲器不同，DRAM 内部可以组织成对多个 bank 进行读或写操作，每个 bank 都有自己的行缓冲器。向不同的 bank 发送一个地址可以允许同时对它们进行读或写操作。例如，对于 4 个 bank 而言，只需要一次访问时间，然后以轮转方式对这 4 个 bank 进行访问就可以提供 4 倍的带宽。这种轮转的访问方式成为地址交叉。

虽然 iPad（见第 1 章）之类的个人移动设备使用独立的 DRAM 颗粒，但服务器的存储器通常是以称为双列存储器模块（Dual Inline Memory Module，DIMM）的小电路板方式买卖。DIMM 通常含有 4~16 块 DRAM 芯片，它们组织成 8 字节位宽。一个使用 DDR4-3200 SDRAM 的 DIMM 每秒可以传送 $8 \times 3 200 = 25 600$ 兆字节。这类 DIMM 以其带宽进行命名：PC25600。一个 DIMM 可以有如此多的 DRAM 芯片，但是在特定的传送中只使用其中一部分，因此需要一个术语来表示 DIMM 上共享公共地址线的芯片子集。为了避免与 DRAM 内部的行和 bank 的名字混淆，使用存储器 rank 来表示 DIMM 中的芯片的一个子集。

01 精解 一种测试 cache 之外的存储器系统的性能的方法是使用流基准程序 [McCalpin, 1995]。它用来测试长向量操作的性能。它们没有时间局部性，并且它们访问的阵列比测试的计算机中的 cache 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 (EEPROM)。

与磁盘和 DRAM 不同，而与其他 EEPROM 技术类似，对闪存的写操作可以使存储位损耗。为了应对该限制，大多数闪存产品都有一个控制器，用来将写操作从已经写入很多次的块中映射到写入次数较少的块中，从而使写操作尽量分散。这种技术称为损耗均衡 (wear leveling)。采用损耗均衡技术，个人移动设备很难超过闪存的写极限。这种均衡技术虽然降低了闪存的潜在性能，但是不需要在高层次的软件中监控块的损耗情况。闪存控制器的这种损耗均衡也将制造过程中出错的存储单元屏蔽掉，从而提高其成品率。

5.2.4 磁盘存储器

如图 5-6 所示，一个磁质硬盘包含一组圆形磁盘片，它们绕着轴心每分钟转动 5 400 ~ 15 000 周。金属盘片的两侧均被磁性存储材料覆盖，其磁性材料与盒式磁带和录像带的材料相同。为了对硬盘上的信息进行读写，每层的表面有一个包含小的电磁线圈的读写磁头。整个驱动器被永久地密封起来以控制驱动器中的环境，从而使得磁头可以距离驱动器表面非常近。

每个磁盘的表面划分为同心圆盘，称为磁道 (track)。每个面通常有几万条磁道。每条磁道同样被划分为用于存储信息的扇区 (sector)；每条磁道有几千个扇区。每个扇区的容量通常是 512 ~ 4 096 字节。信息在磁介质上保存的顺序为扇区号、一个间隙、包含该扇区纠错码 (见 5.5 节) 的信息、一个间隙、下一扇区的扇区号。

380
381

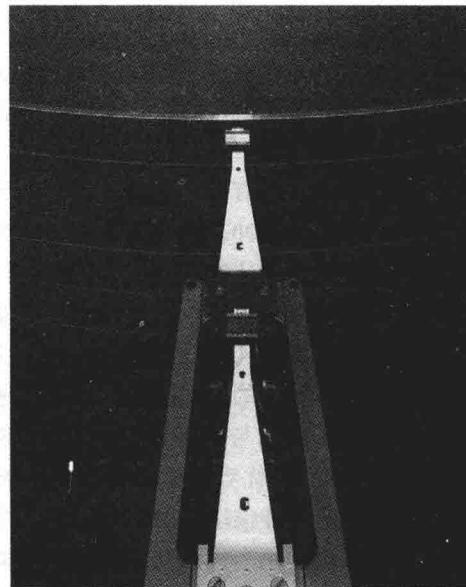


图 5-6 具有 10 个盘面和读写头的磁盘。当今磁盘的直径是 2.5 ~ 3.5 英寸，并且每个驱动器通常有 1 ~ 2 个圆形磁盘片

- ② 磁道：位于磁盘表面的数万个同心圆环中的任意一个圆环称为一个磁道。
- ② 扇区：构成磁盘上磁道的基本单位，是磁盘上数据读写的最小单位。

访问每个盘面的磁头连在一起相互协调运动，因此每个盘面的磁头位于相同的扇区。术语柱面用来表示磁头在给定点时访问到所有盘面上的所有扇区的集合。

为了访问数据，操作系统必须对磁盘进行三步操作。第一步是将磁头移动到适当的磁道之上，这称为寻道 (seek)，将磁头移动到目标磁道所需的时间称为寻道时间。

- ② 寻道：把读写磁头移动到磁盘上适当的磁道上面的过程。

磁盘供应商在他们的手册中报告寻道的最小、最大和平均时间。前面两个寻道时间数据比较容易测量，但是平均寻道时间却因与寻道距离相关而难测量。工业界计算平均寻道时间的方

法是对所有可能的寻道时间取平均值。平均寻道时间通常在 3 ~ 13ms，但是，由于应用程序以及磁盘访问调度策略的不同，且磁盘数据具有局部性，所以实际的平均寻道时间通常只有标称数据的 25% ~ 33%。由于对同一文件通常会做连续访问，其操作系统也会尽量把这些访问一起进行调度，所以这种局部性会增加。

一旦磁头到达了正确的磁道，就必须等待要访问的扇区转动到读写头下面。该等待时间称为旋转延时（rotational latency）。平均延时通常是磁盘转动一周时间的一半。磁盘每分钟转动 5 400 ~ 15 000 周。5 400 周的磁盘的平均旋转延时为

$$\text{平均旋转延时} = \frac{0.5 \text{ 周}}{5400 \text{ RPM}} = \frac{0.5 \text{ 周}}{5400 \text{ RPM} / (60 \frac{\text{秒}}{\text{分钟}})} = 0.0056 \text{ 秒} = 5.6 \text{ 毫秒}$$

磁盘访问的最后一部分是传输时间，即传输一块数据需要的时间。传输时间是扇区大小、旋转速度和磁道信息密度的一个函数。2012 年的传输速率在每秒 100 ~ 200MB。

② 旋转延时：在磁头定位后，指定扇区通过读写头的所需时间。通常是磁盘转动一周时间的一半。

大多数磁盘控制器的一个复杂问题是，它有一个仅保存最近传输过的扇区数据的 cache，从 cache 中传输数据的速率通常更高，在 2012 年到了每秒 750MB（每秒 6Gb）。

现在，块号存放在哪里不能再凭直觉了。前面所述的扇区 - 磁道 - 柱面模型有如下假定：邻近的块在同一磁道上；因为访问同一柱面上的块不需要寻道时间，所以访问时间较短；一些磁道与其他磁道距磁头更近。变化的原因是磁盘接口层次的提升。为了加速数据传输，高层次的接口将磁盘组织地更像磁带，而不像随机访问设备。在一个磁面上，逻辑块以弯曲形式顺序排列，尽可能使所有扇区的数据密度相同，从而获得最好的性能。因此，顺序的块可能在不同的磁道上。

概括起来，磁盘和半导体存储器技术的主要差别是磁盘的访问速度慢，这主要是因为它们是机械器件——闪存比磁盘快 1000 倍，DRAM 比磁盘快 100 000 倍——但是它们却因为使用适度的成本即可获得很大的存储容量而使得每比特位的成本低了许多——磁盘会便宜 10 ~ 100 倍。与闪存类似，磁盘是非易失的，但却不存在写损耗问题。然而，闪存更加坚固，因此更加适用于个人移动设备。

5.3 cache 的基本原理

cache: 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

在前面介绍的图书馆例子中，书桌就好比是高速缓存（cache）——一个存放待用事物（书籍）的安全场所。在早期的商业计算机中，cache 是处理器和主存之间的特殊层次。在第 4 章的数据通路中，存储器就被 cache 简单地替代了。现在，尽管 cache 的使用占据了主导地位，但该术语也用来指代那些基于局部性原理来管理的存储器。cache 最早出现在 20 世纪 60 年代早期的研究型计算机中，后期则被应用于产品型计算机。如今生产的每一台通用计算机，从服务器到低功耗嵌入式处理器，都含有 cache。

在这一节中，我们先来看一个简单的 cache，处理器每次请求一个字，每个块也由一个单

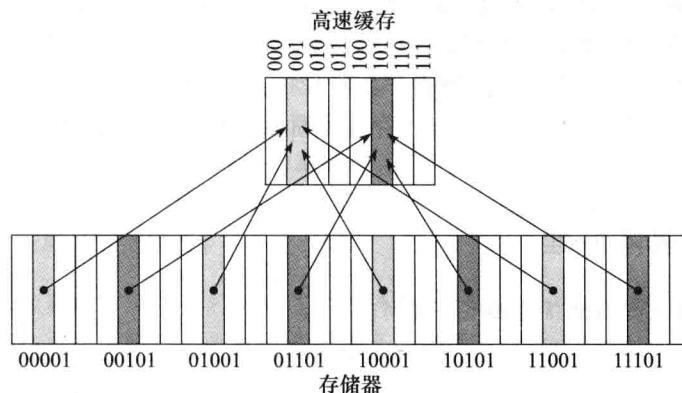
独的字组成（已经熟悉 cache 基本原理的读者可以跳至 5.4 节）。图 5-7 就是一个简单的 cache，要访问的数据项最初不在 cache 中。在请求发出之前，cache 中保存了最近所访问过的数据项 X_1, X_2, \dots, X_{n-1} 的集合，而当前处理器所要访问的数据项 X_n 并不在 cache 中。该请求导致了一次缺失， X_n 被从主存调入 cache 之中。

观察图 5-7 中的情景，有两个问题需要解决：怎样知道一个数据项是否在 cache 中？此外，如果数据项在 cache 中，如何找到它？这两个问题的答案是相关的。如果每个字都放在 cache 中确定的位置，那么只要它在 cache 中，我们就能直接找到它。在 cache 中为主存中每个字分配一个位置的最简单方法就是根据这个字的主存地址进行分配，这种 cache 结构称为直接映射（direct mapped）。每个存储器地址对应到 cache 中一个确定的地址。对直接映射 cache 来说，主存地址和 cache 位置之间的典型映射通常比较简单。例如，几乎所有的直接映射 cache 都使用以下的映射方法：

$$(块地址) \bmod (\text{cache 中的块数})$$

② 直接映射：一种 cache 结构，其中每个存储器地址仅仅对应到 cache 中的一个位置。

如果 cache 中的块数是 2 的幂，取模的计算就很简单，只需要取地址的低 \log_2 （块中的 cache 容量）位。因此，一个 8 块的 cache 可以使用块地址中最低的三位 ($8 = 2^3$)。例如，图 5-8 中，直接映射的 cache 块大小为 8 个字，存储器地址 $1_{10}(00001_2)$ 到 $29_{10}(11101_2)$ 被映射到 cache 中 $1_{10}(001_2)$ 到 $5_{10}(101_2)$ 的位置。



X_4	X_4
X_1	X_1
X_{n-2}	X_{n-2}
X_{n-1}	X_{n-1}
X_2	X_2
X_n	X_n
X_3	X_3

a) 访问 X_n 之前

b) 访问 X_n 之后

图 5-7 对字 X_n 访问前后 cache 中的内容，最初 X_n 不在 cache 中。这次访问引起了一次缺失，并强制 cache 从存储器中收回 X_n ，随后将 X_n 放入 cache 中

图 5-8 主存地址 0 ~ 31 被映射到 cache 中的相同位置，该 cache 中有 8 个字。由于 cache 中有 8 个字，地址 X 被映射到直接映射 cache 字 $X \bmod 8$ ，即低 $\log_2(8) = 3$ 位被用作 cache 索引。因此，地址 $00001_2, 01001_2, 10001_2$ 和 11001_2 都对应于 cache 中第 001_2 块，而地址 $00101_2, 01101_2, 10101_2$ 和 11101_2 都对应于 cache 中第 101_2 块。

由于 cache 中每个位置可能对应于主存中多个不同的地址，我们如何知道 cache 中的数据项是否是所请求的字呢？即如何知道所请求的字是否在 cache 中？我们可以在 cache 中增加一组标记（tag），标记中包含了地址信息，这些地址信息可以用来判断 cache 中的字是否就是所

请求的字。标记只需包含地址的高位，也就是没有用来检索 cache 的那些位。例如，在图 5-8 中，标记位只需使用 5 位地址中的高两位，地址低 3 位的索引域则用来选择 cache 中的块。按照定义，任何一个可以放入相同 cache 块中的字的地址的索引域一定是那个块的块号，因此标记位无需包含这些冗余的索引位。

- 标记：表中的一个字段，包含了地址信息，这些地址信息可以用来判断 cache 中的字是否就是所请求的字。

我们还需要一种方法来判断 cache 块中确实没有包含有效信息。例如，当一个处理器启动时，cache 中没有数据，标记域中的值没有意义。甚至在执行了一些指令后，cache 中的一些块依然为空，如图 5-7 所示。因此，在 cache 中，这些块的标记应该被忽略。最常用的方法就是增加一个有效位（valid bit）来标识一个块是否含有一个有效地址。如果该位没有被设置，则不能使用该块中的内容。[384]

- 有效位：表中的一个字段，用来标识一个块是否含有一个有效数据。

在本节的剩余部分，我们将重点说明如何在 cache 中进行读操作。通常来说，由于读操作不会改变 cache 中的内容，因而处理时比写操作要简单一些。在探讨了读操作和 cache 缺失如何处理的基本原理后，我们将介绍实际计算机中 cache 的设计以及 cache 如何处理写操作。

01 重点 cache 可能是预测技术中最重要的例子。它依赖于局部性原理尽可能在存储器层次结构的更高一层中寻找需要的数据，并且当预测错误时提供能够从存储器层次的更低一层中找到并获取正确数据的机制。现代计算机中 cache 预测命中率通常高于 95%（见图 5-47）。

5.3.1 cache 访问

下面是对一个容量为 8 块的空 cache 进行 9 次访问的一个序列，包括每次访问的行为。图 5-9 给出了每一次缺失后 cache 内容的变化。由于 cache 中有 8 个块，地址的低 3 位给出了块号。

访问的十进制地址	访问的二进制地址	在 cache 中命中/缺失	分配的 cache 块 (查找或者放置的位置)
22	10110_2	缺失（图 5-9b）	$(10110_2 \bmod 8) = 110_2$
26	11010_2	缺失（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	缺失（图 5-9d）	$(10000_2 \bmod 8) = 000_2$
3	00011_2	缺失（图 5-9e）	$(00011_2 \bmod 8) = 011_2$
16	10000_2	命中	$(10000_2 \bmod 8) = 000_2$
18	10010_2	缺失（图 5-9f）	$(10010_2 \bmod 8) = 010_2$
16	10000_2	命中	$(10000_2 \bmod 8) = 000_2$

索引	有效位 (V)	标记	数据
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a) 上电后 cache 的初始状态

索引	有效位 (V)	标记	数据
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10₂	主存 (10110 ₂)
111	N		

b) 处理地址 (10110₂) 缺失后的 cache 状态

索引	有效位 (V)	标记	数据
000	N		
001	N		
010	Y	11₂	主存 (11010 ₂)
011	N		
100	N		
101	N		
110	Y	10₂	主存 (10110 ₂)
111	N		

c) 处理地址 (11010₂) 缺失后的 cache 状态

索引	有效位 (V)	标记	数据
000	Y	10₂	主存 (10000 ₂)
001	N		
010	Y	11₂	主存 (11010 ₂)
011	N		
100	N		
101	N		
110	Y	10₂	主存 (10110 ₂)
111	N		

d) 处理地址 (10000₂) 缺失后的 cache 状态

索引	有效位 (V)	标记	数据
000	Y	10₂	主存 (10000 ₂)
001	N		
010	Y	11₂	主存 (11010 ₂)
011	Y	00₂	主存 (00011 ₂)
100	N		
101	N		
110	Y	10₂	主存 (10110 ₂)
111	N		

e) 处理地址 (00011₂) 缺失后的 cache 状态

索引	有效位 (V)	标记	数据
000	Y	10₂	主存 (10000 ₂)
001	N		
010	Y	10₂	主存 (10010 ₂)
011	Y	00₂	主存 (00011 ₂)
100	N		
101	N		
110	Y	10₂	主存 (10110 ₂)
111	N		

f) 处理地址 (10010₂) 缺失后的 cache 状态

图 5-9 对相应的地址序列给出每次请求缺失后 cache 中的内容、索引和标记域（二进制表示）。cache 初始为空，所有的有效位关闭（N）。处理器请求以下地址：10110₂（缺失）、11010₂（缺失）、10110₂（命中）、11010₂（命中）、10000₂（缺失）、00011₂（缺失）、10000₂（命中）、10010₂（缺失）以及 10000₂（命中）。这些图指出了依次出现的每一次缺失处理后 cache 中的内容。当地址 10010₂（18）被访问时，地址为 11010₂（26）中的项就要被替换掉，随后再访问 11010₂ 会引起缺失。标记域只包含地址的高位部分。cache 块 i 、标记域为 j 的字的完整地址是 $j \times 8 + i$ ，或者等效为地址域 j 和索引 i 的连接。例如，上面的 f 图中，索引 010₂、标记为 10₂ 的块，对应地址 10010₂。

385
l
387

由于 cache 初始为空，第一次访问的一些数据都会发生缺失。图 5-9 对每一次访问行为进行了描述。第 8 次访问将会对 cache 中的一个块产生冲突的请求。地址 18 (10010₂) 的字将被取到 cache 的第 2 块 (010₂) 中。因此，它将替换掉原先存在于 cache 第 2 块 (010₂) 中的地址为 26 (11010₂) 中的字。这种行为令 cache 具有时间局部性：最近访问过的字替换掉较早访问的字。

这种情况就好比要从书架上取一本书，而书桌上已经没有任何地方可以放这本书了，因此原先摆在书桌的某本书必须被放回书架。在直接映射 cache 中，只有一个位置可以存放最新请求的数据项，因此对于哪个数据项被替换也只有一种选择。

对每个可能的地址，在 cache 中进行如下查找：地址的低位用来找到 cache 中与该地址匹配的唯一项。图 5-10 说明一个地址可以划分为：

- 标记域：用来与 cache 中标记域的值进行比较。
- cache 索引：用来选择块。

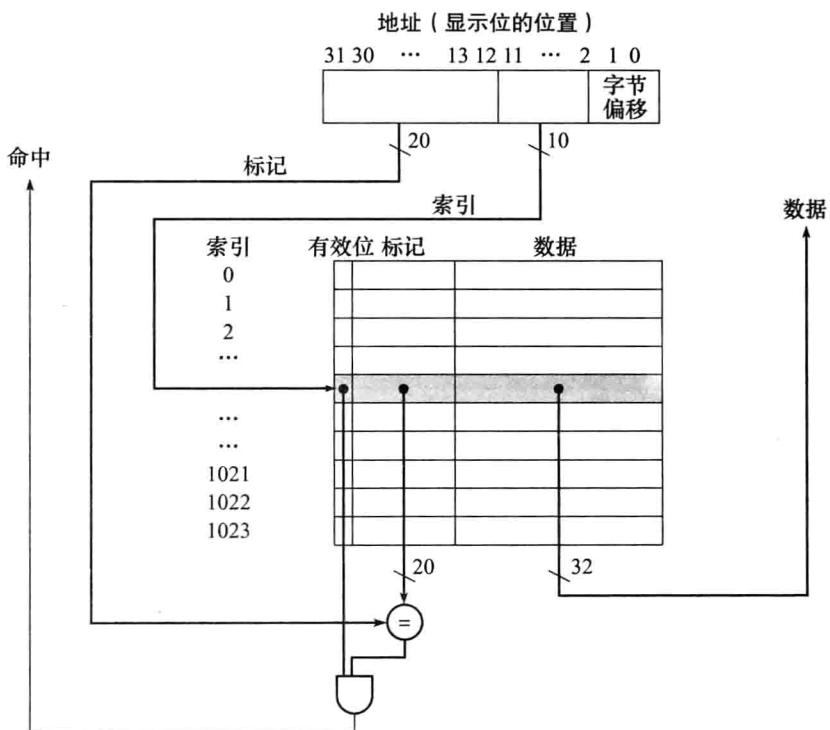


图 5-10 对这个 cache，地址的低位用来选择由数据字和标记组成的一个 cache 项。这个 cache 中有 1 024 个字，即 4KiB。在这一章中，我们假设使用 32 位的地址。cache 中的标记与地址高位相比较，判断 cache 中的项是否符合请求的地址。由于 cache 有 2^{10} (1 024) 个字，块大小为 1 个字，因此，索引 cache 需要 10 位，剩下的 $32 - 10 - 2 = 20$ 位用来和标记相比较。如果标记和地址的高 20 位相等，并且有效位开启，那么请求在 cache 中命中，相应的字被提供给处理器。否则，发生缺失。

cache 块的索引以及标记唯一确定了 cache 块中存放内容的主存地址。由于索引域用来寻址，而且一个 n 位的域有 2^n 种值，直接映射 cache 中项的总数必须为 2^n 的幂。在 MIPS 体系结构中，由于字是以 4 字节的倍数对齐的，每个地址至少有两位用来指定字中的一个字节。因此当选择块中的一个字时至少两位被忽略。

由于 cache 不仅存储数据也存储标记位，cache 所需的总位数是 cache 大小和地址位数的函数。在前文中提及的块大小为 1 个字，但通常块大小为多字。就像下面的情况：

- 32 位地址。
- 直接映射 cache。
- cache 大小为 2^n 个块，因此 n 位被用来索引。
- 块大小为 2^m 个字 (2^{m+2} 字节)，因此 m 位用来查找块中的字，两位是字节偏移信息。

标记域的大小为

$$32 - (n + m + 2)$$

直接映射的 cache 总位数为

$$2^n \times (\text{块大小} + \text{标记域大小} + \text{有效位域大小})$$

由于块大小为 2^m 个字 (2^{m+5} 位)，同时我们需要 1 位有效位，因此这样一个 cache 的位数是

$$2^n \times (2^m \times 32 + (32 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 31 - n - m)$$

尽管以上计算是实际的大小，但是通常对 cache 命名只考虑数据的大小而不考虑标记域和有效位域的大小。因此，图 5-10 中是一个 4KiB 的 cache。

01 例题 · cache 中的位数

假设一个直接映射的 cache，有 16KiB 的数据，块大小为 4 个字，地址为 32 位，那么该 cache 总共需要多少位？

01 答案

我们知道 16KiB 是 $4\,096(2^{12})$ 个字，块大小是 4 个字 (2^2)，那么就有 $1\,024(2^{10})$ 个块。每个块有 4×32 即 128 位的数据，加上 $32 - 10 - 2 - 2$ 位的标记域，再加上一个有效位，因此，总的 cache 大小是

$$2^{10} \times (4 \times 32 + (32 - 10 - 2 - 2) + 1) = 2^{10} \times 147 = 147\text{KiB}$$

即能装 16KiB 数据的 cache 总共需要 18.4KiB 的容量。这个 cache 的总位数是数据存储量的 1.15 倍。□

01 例题 · 将一个地址映射到多字大小的 cache 块中

考虑一个 cache 中有 64 个块，每块大小为 16 字节，那么字节地址 1200 将被映射到 cache 中的哪一块？

01 答案

块由下面公式给出：

$$\text{(块地址)} \bmod \text{(cache 中的块数)}$$

其中块地址为

$$\text{字节地址} / \text{每块字节数}$$

注意，这个块地址包含了所有在

$$\text{字节地址} / \text{每块字节数} \times \text{每块字节数}$$

和

$$\text{字节地址} / \text{每块字节数} \times \text{每块字节数} + (\text{每块字节数} - 1)$$

之间的地址。

因此，由于每个块有 16 字节，字节地址 1200 对应的块地址为

$$1\,200 / 16 = 75$$

对应于 cache 中的块号 $(75 \bmod 64) = 11$ 。事实上，地址 1200 和 1215 之间的所有地址都映射在这一块。□

较大的 cache 块能更好地利用空间局部性以降低缺失率。如图 5-11 所示，增加块大小通常会引起缺失率下降。而当块大小在 cache 容量中所占比例增加到一定程度时，缺失率也随之增加。这是因为此时 cache 中块的数量变得很少，对于这些块将会有大量的竞争发生。结果，就造成一个块中的数据在被多次访问之前就被替换出 cache。另一方面，对于一个太大的块，块中各个字之间的空间局部性也会降低，缺失率降低所带来的益处也会相应减少。

391 仅仅增加块大小所带来的一个更加严重的后果是缺失成本的增加。由较低存储器层次取出块并存放至 cache 中所花费的时间决定了缺失代价。取出块的时间可以分为两部分：第一个字的延迟时间和剩余部分块的传输时间。很显然，除非改变存储系统，否则，传输时间，也就是缺失代价将随着块大小的增大而增加。此外，当块越来越大时，缺失率的改善也开始降低。而当块过于大时，缺失代价的增长超过了缺失率的降低，因此 cache 的性能也随之降低。当然，如果把存储器设计得能更有效地传输较大的块时，我们就能增加块的大小并且进一步改善 cache 性能。这一点我们将在下一节讨论。

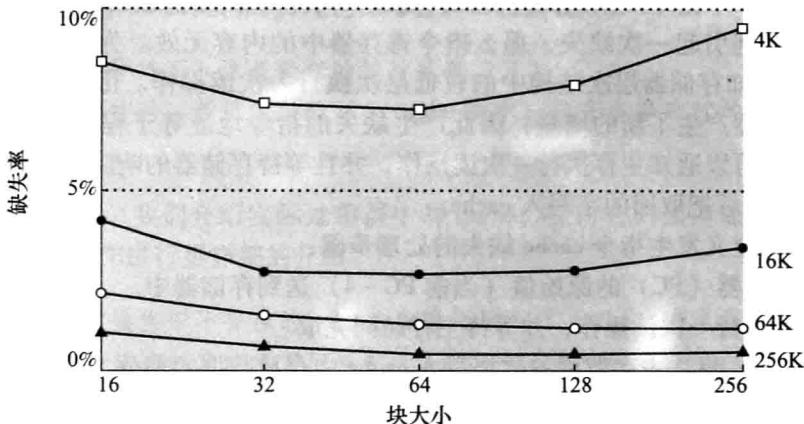


图 5-11 缺失率与块大小。注意到相对于 cache 容量来说，块大小太大，缺失率实际上是上升的。每条曲线代表不同容量的 cache（图中没有考虑相联度，稍后讨论）。不幸的是，如果包括块大小，那么 SPEC CPU 2000 追踪信息将花费太长的时间，因此这些数据都基于 SPEC92

01 精解 缺失时，较大的块会带来长延迟从而增加了缺失代价。要减少这一部分延迟尽管比较困难，但我们可以隐藏一些传输时间来有效地降低缺失代价。最简单的方法是提前重启（early restart），即当块中所需字一旦返回就马上继续执行，而不需要等到整个块都传过来之后再执行。许多处理器利用这种技术进行指令访问，效果甚佳。大部分指令访问都具有连续性，因此存储系统每个时钟周期都能传送一个字，只要存储系统能保证及时传递新的指令字，那么当所请求的字返回时，处理器就可以重新开始操作。将这种技术应用于数据 cache 时效率要低一些，这是因为所请求的字可能以一种无法预知的方式分布，而在传输结束前处理器请求另一块中的字的可能性也很高。如果数据传输正在进行，处理器就无法访问数据 cache，因而它必然阻塞。

另一种更复杂的机制是重新组织存储器，使得被请求的字先从存储器传到 cache 中，然后再传送该块的剩余部分，从所请求的字的下一个地址开始传送，再回到块的开始。这种技术被称为请求字优先（requested word first）或者关键字优先（critical word first），它比提前重启要快一些，但与提前重启一样，会因为同样的问题而受到限制。

5.3.2 cache 缺失处理

在研究一个真实系统中的 cache 之前，让我们先来看一下控制单元是如何处理 cache 缺失（cache miss）的（在 5.9 节将详细介绍 cache 控制器）。控制单元必须能检测到缺失的发生，然后从主存（或者较低一级 cache）中取回所需的数据来处理缺失。如果在 cache 中命中，计算机继续使用该数据，就好像什么都没有发生过。

● cache 缺失：由于数据不在 cache 中而导致被请求的数据不能满足。

命中时，对处理器控制的修改不太重要；缺失时则需要增加一些额外的工作。cache 缺失处理由两部分共同完成：处理器控制单元，以及一个进行初始化主存访问和重新填充 cache 的独立控制器。cache 缺失引起流水线阻塞（见第 4 章），这与中断不同，中断发生时需要保存所有寄存器的状态。当 cache 缺失，我们等待主存操作完成时，整个处理器阻塞，临时寄存器和程序员可见的寄存器中的内容基本被冻结。与之相比，更为复杂的乱序执行处理器在等待 cache 缺失处理的同时，依然能执行其他一些指令。但是，在本节中，我们均假定为顺序执行处理器，当 cache 缺失时其被阻塞。

我们再来讨论一下指令发生缺失时将如何处理，同样的方法略加修改便可以用来处理数据缺失。如果指令访问引起一次缺失，那么指令寄存器中的内容无效。为了将正确的指令取回 cache，我们必须通知存储器层次结构中的较低层次执行一次读操作。由于在执行的第一个时钟周期，程序计数器产生了新的增量，因此产生缺失的指令地址等于程序计数器中的值减 4。当地址产生时，就可以通知主存执行一次读操作，并且等待存储器的响应（访问主存可能需要多个时钟周期），随后把取回的字写入 cache。

现在我们可以定义发生指令 cache 缺失的处理步骤：

- 1) 把程序计数器 (PC) 的原始值 (当前 PC - 4) 送到存储器中。
- 2) 通知主存执行一次读操作，并等待主存访问完成。
- 3) 写 cache 项，将从主存取回的数据写入 cache 中存放数据的部分，并将地址的高位 (从 ALU 中得到) 写入标记域，设置有效位。
- 4) 重启指令执行第一步，重新取指，这次该指令在 cache 中。

数据访问时对 cache 的控制基本相同：发生缺失时，处理器发生阻塞，直到从存储器中取回数据后才响应。

5.3.3 写操作处理

写操作略微不同。如果有一个 store 指令，我们只将该数据写入数据 cache (而不改变主存的内容)；那么，在写入 cache 之后，主存与 cache 相应位置中的值将不同。在这种情况下，cache 和主存被认为不一致 (inconsistent)。保持主存和 cache 一致性最简单的方法就是将这个数据同时写入主存和 cache 中，这种方法称为写直达 (write-through) 法。

- ② 写直达：也译为写通过或写穿。写操作总是同时更新 cache 和下一存储器层次，以保持二者一致性。

写操作要考虑的另一个主要方面是发生写缺失的情况。我们首先从主存中取出块中的字。数据块被取回并存入 cache 中后，我们就可以将引起缺失的字重新写入 cache 中。同时，我们使用全地址将该字写入主存。

尽管这种设计方案能简单地处理写操作，但却无法提供良好的性能。使用写直达的机制，每次写操作都要把数据写入主存之中。这些写操作将花费大量的时间，可能至少要花费 100 个处理器时钟周期，并且大大降低了机器速度。例如，假设 10% 的指令是 store 指令，没有 cache 缺失的情况下 CPI 为 1.0，每次写操作要额外花费 100 个周期，就使得 CPI 为 $1.0 + 100 \times 10\% = 11$ ，性能降低 10 倍多。

这个问题的一种解决方法是采用写缓冲 (write buffer)。当一个数据在等待被写入主存时，先将它放入写缓冲中。当把数据写入 cache 和写缓冲后，处理器可以继续执行。当写主存操作完成后，写缓冲里的数据项也得到释放。如果写缓冲已经满了，那么当处理器执行到一个写操作时就必须停下来直到写缓冲中有一个空位置。当然，如果存储器完成写操作的速度比处理器产生写操作的速度慢，那么再多的缓冲器也没有用，因为产生写操作比存储系统接收它们要快。

产生写操作的速度也可能比存储器接收它们的速度慢，尽管这样，仍有可能发生阻塞。当写操作突发产生时，这种情况就会发生。为了减少这种阻塞的发生，通常需要增加处理器写缓冲的深度。

除了写直达，另一种可供选择的方法为写回 (write-back)。在写回机制中，当发生写操作时，新值仅仅被写入 cache 块中。只有当修改过的块被替换时才需要写到较低层存储结构中。写回机制可以提高系统的性能，尤其是当处理器产生写操作的速度和主存处理写操作的速度一

样快甚至更快时；但是，写回机制的实现也比写直达要复杂得多。

- ② 写缓冲：一个保存等待写入主存数据的缓冲队列。
- ③ 写回：当发生写操作时，新值仅仅被写入 cache 块中，只有当修改过的块被替换时才写到较低层存储结构中。

在本节的剩余部分，我们介绍实际处理器中的 cache，探讨它们如何处理读和写操作。在 5.8 节，我们会对写操作进行更详细的介绍。

01 精解 写操作将读操作中不存在的一些复杂情况引入了 cache。这里我们讨论其中的两种情况：写缺失时的策略以及使用写回机制的 cache 中写操作的有效执行。

考虑在写直达机制下的 cache 缺失，最常使用的策略是分配 cache 中的一块，称为写分配（write allocate）。数据块从主存中取回，并且在该块中的恰当区域重写数据。另一种策略则是只更新主存中块的一部分，而不写入 cache 中，这种方法称为写不分配（no write allocate）。这种机制产生的原因是，有时程序会写整个块，就像有时操作系统会将存储器中的一页全部填零一样。在这种情况下，由初始的写缺失引起的取数据就不必要了。一些计算机允许基于每一页来更改写分配策略。

使用写回策略的 cache 比使用写直达策略的 cache 实现有效存储要复杂得多。在写直达的 cache 中，可以将数据写入 cache 并且读标记，如果标记不匹配，就发生缺失。由于 cache 采用写直达策略，在 cache 中重写数据块并不会有危险，因为主存中存储了正确的值。在写回 cache 中，如果 cache 中的数据被重写过并且此时发生缺失，就必须把整块写回主存中。如果在不知道 cache 是否命中（在写直达的 cache 中可以知道）的情况下就简单地根据存储指令重写块，我们就破坏了块的内容，而块本身也没有在存储层的较低层进行备份。

在写回 cache 中，由于无法重写块，存储操作需要两个周期（一个周期用来检查命中情况，下一个周期才真正执行写操作），或者需要一个写缓冲来保存数据——通过流水线有效地使存储操作只花费一个周期。如果使用存储缓冲区，处理器在正常的 cache 访问周期内查找 cache 并把数据放入存储缓冲区中。如果 cache 命中，在下一个还没有用到的 cache 访问周期，新数据被从存储缓冲区写入 cache 中。

相比较而言，在写直达 cache 中，写操作总是在一个周期内完成。我们读标记位，并且写被选择块的部分区域。如果标记与被写块的地址相同，处理器通常可以继续执行，因为正确的块已经被更新过了。如果标记与被写块的地址不同，处理器产生写缺失并去取对应于该地址块的剩余部分。

很多写回机制的 cache 也使用写缓冲，当在发生缺失替换一个被修改的块时，写缓冲可以起到降低缺失代价的作用。在这种情况下，被修改的数据块移入与 cache 相联的写回缓冲器，同时从主存中读出所需要的数据块。随后，写回缓冲器再将数据写入主存。如果下一次缺失没有立刻发生，当脏数据块必须被替换时，这种方法可以减少一半的缺失代价。

5.3.4 一个 cache 的例子：内置 FastMATH 处理器

内置 FastMATH 处理器是一个快速的嵌入式微处理器，它采用 MIPS 架构，cache 实现很简单。在本章的最后，我们将介绍 ARM 和 Intel 微处理器中更为复杂的 cache 设计，但是出于教学的目

的，我们首先分析这个简单的实例。图 5-12 给出了内置 FastMATH 处理器数据 cache 的结构。

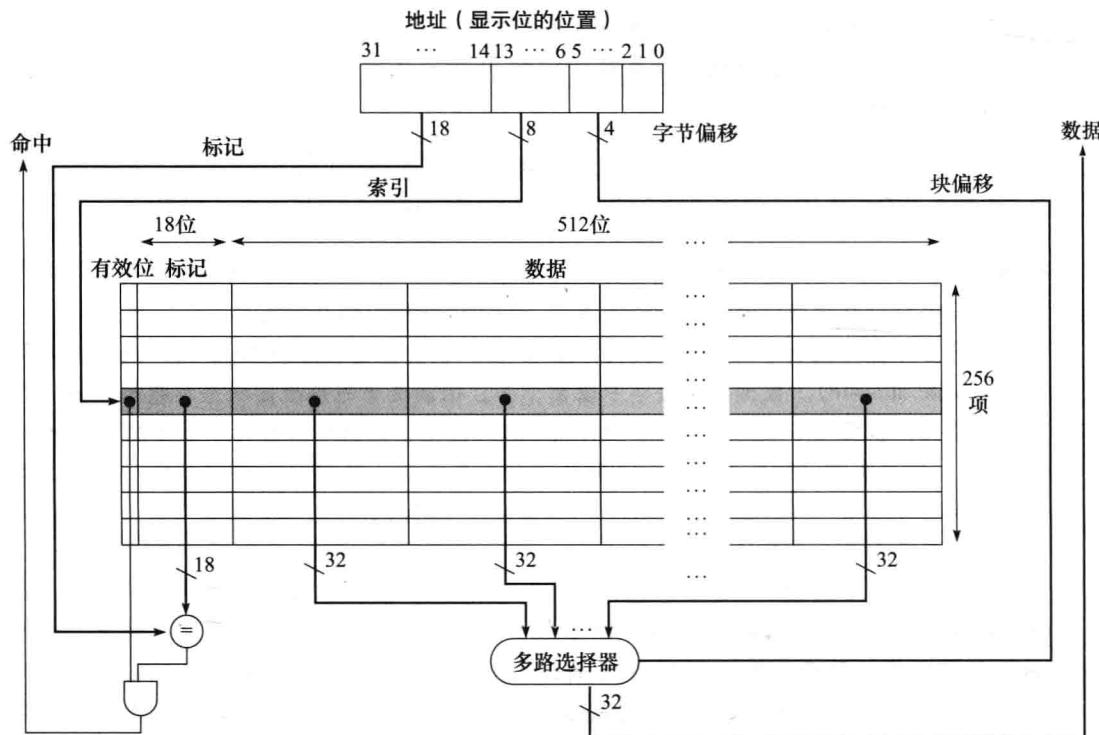


图 5-12 内置 FastMATH 处理器的 16KiB 的 cache，cache 中有 256 块，每块 16 个字。标记域是 18 位，索引域是 8 位，另有一个 4 位（2~5 位）的域用来索引块，并使用一个 16 选 1 的多路选择器从块中选择所需的字。实际上，为了消除多路选择器，cache 使用一个大容量的 RAM 单独存放数据，一个更小的 RAM 则用来存放标记，大容量数据 RAM 所需的额外地址位由块偏移提供。这样，大容量 RAM 中字长为 32 位，字数必须为 cache 中块数的 16 倍

该处理器采用 12 级流水线结构。当以峰值速度执行时，处理器每个时钟周期可以请求一个指令字和一个数据字。为了满足不阻塞流水线的需求，使用了分离的指令 cache 和数据 cache。每个 cache 容量为 16KiB，即 4096 个字，每块有 16 个字。

对 cache 的读请求很简单，由于使用了分离的指令 cache 和数据 cache，读写每个 cache 都需要各自独立的控制信号（记住当发生缺失时，需要更新指令 cache）。因此，对任何一个 cache 执行读请求的步骤如下：

- 1) 将地址送到适当的 cache 中去，该地址来自程序计数器（对于指令访问），或者来自于 ALU（对于数据访问）。
- 2) 如果 cache 发出命中信号，请求的字就出现在数据线上。由于在请求的数据块中有 16 个字，因此需要选择那个正确的字。块索引域用来控制多路选择器（如图 5-12 底部所示），从检索到的块中选择 16 个字中的某个字。
- 3) 如果 cache 发出缺失信号，我们把地址送到主存。当主存返回数据时，把它写入 cache 后再读出以满足请求。

对于写操作，内置 FastMATH 处理器同时提供写直达和写回机制，由操作系统来决定某种应用该使用哪个机制。它有一个只包含一项的写缓冲。

内置 FastMATH 处理器采用的 cache 结构的缺失率是怎样的呢？图 5-13 给出了指令 cache 和数据 cache 的缺失率。综合缺失率是在考虑了指令和数据的不同访问频率后每个程序每次访

问的实际缺失率。

指令缺失率	数据缺失率	综合缺失率
0.4%	11.4%	3.2%

图 5-13 内置 FastMATH 处理器执行 SPEC CPU2000 测试程序时指令和数据的近似缺失率。综合缺失率是将 16KiB 的指令 cache 和 16KiB 的数据 cache 结合起来考虑的实际缺失率。它是以指令和数据访问频率为权重，分别考虑指令和数据缺失率后得到的

尽管缺失率是 cache 设计的一个重要标准，但最终的衡量标准是存储系统对程序执行时间的影响。我们将简要介绍缺失率与执行时间之间的关系。

01 精解 混合 cache 容量等于两个分离 cache (split cache) 容量的总和。通常来说，混合 cache 具有较高的命中率，其原因是混合 cache 没有将指令用的 cache 块数与数据用的 cache 块数严格区分出来。不过，很多处理器使用分离的指令和数据 cache 以提高 cache 的带宽（同时也可减少冲突引起的缺失，见 5.8 节）。

下面是与内置 FastMATH 处理器中 cache 容量相同的 cache 的缺失率，混合 cache 的容量等于两个分离 cache 容量之和。

- 总的 cache 容量：32KiB。
- 分离 cache 的实际缺失率：3.24%。
- 混合 cache 的缺失率：3.18%。

分离 cache 的缺失率只是稍差一点。

通过支持指令和数据同时访问来使 cache 带宽加倍，这一优点很容易就克服了缺失率稍微增加的缺点。这一事实也提醒我们缺失率不是衡量 cache 性能的唯一标准，正如 5.4 节所示。

02 分离 cache：一级 cache 由两个独立的 cache 组成，两者可以并行工作，一个处理指令，另一个处理数据。

5.3.5 小结

前面我们从介绍最简单的 cache 开始：每块只有一个字的直接映射 cache。在这样的 cache 结构中，命中和缺失都很简单，因为每个字都明确地被写入到一个位置，同时每个字都有单独的标记。为了保持 cache 和主存的一致性，可以使用写直达机制，这样，每次对 cache 进行写操作都会引起主存的更新。不同于写直达机制，写回机制仅在 cache 中有需要被替换的块时才将相应的块复制到主存中去。在后面的章节中我们将进一步讨论这一机制。

为了利用空间局部性，cache 中的块大小必须大于一个字。使用较大的块可以降低缺失率，减少 cache 中与数据存储量相关的标记存储量，从而提高 cache 的效率。尽管块容量的增大可以降低缺失率，但同时也会带来缺失代价的增加。如果缺失代价与块容量成线性关系增长，那么较大的数据块很轻易就能导致性能变差。

为了避免性能损失，可以通过增加主存的带宽来更高效地传输数据块。增加 DRAM 外部带宽最常用的方法包括：增加存储器位宽和交叉存取。DRAM 设计者还改进了处理器和存储器之间的接口以增加突发模式下传输的带宽，以减小使用更大 cache 块带来的开销。

01 小测验

存储系统的速度影响了设计人员如何选择 cache 块的大小。下面哪些 cache 设计者的指导思想是正确的？

1. 存储器延迟越短, cache 块越小。
2. 存储器延迟越短, cache 块越大。
3. 存储器带宽越高, cache 块越小。
4. 存储器带宽越高, cache 块越大。

5.4 cache 性能的评估和改进

在这一节中, 我们首先探讨评估和分析 cache 性能的方法。随后我们对两种改进 cache 性能的不同技术进行研究。第一种技术是通过减少存储器中不同数据块争用 cache 中同一位置的概率来降低缺失率。第二种技术通过在存储器层次结构中额外增加一层来减少缺失代价。这种技术被称为多级高速缓存 (multilevel caching), 最初出现在 1990 年售价超过 100 000 美元的高端计算机中, 此后该技术被广泛应用于个人移动设备中, 而售价只有几百美元。

398

CPU 时间可以划分为 CPU 执行程序花费的时钟周期和 CPU 等待存储系统花费的时钟周期。通常来说, 我们假定 cache 访问命中的开销是 CPU 正常执行周期的一部分。因此,

$$\text{CPU 时间} = (\text{CPU 执行时钟周期数} + \text{存储器阻塞的时钟周期数}) \times \text{时钟周期}$$

我们假设存储器阻塞的时钟周期数主要来自于 cache 缺失, 同时我们将讨论限制在存储系统的简化模型上。在实际的处理器中, 由读、写操作引起的阻塞可能十分复杂, 并且对性能的准确预测通常需要对处理器和存储系统进行细致的模拟。

存储器阻塞的时钟周期数可以被定义为读操作与写操作引起阻塞的时钟周期数之和:

$$\text{存储器阻塞时钟周期数} = \text{读操作引起阻塞的时钟周期数} + \text{写操作引起阻塞的时钟周期数}$$

读操作阻塞的时钟周期数可以根据每个程序中读的次数、读操作发生缺失时的代价 (缺失处理需要的时钟周期) 以及读缺失率来定义:

$$\text{读操作阻塞的时钟周期数} = (\text{读的次数} / \text{程序数}) \times \text{读缺失率} \times \text{读缺失代价}$$

写操作的情况就要复杂一些。对于写直达机制, 有两种情况引起阻塞: 一种是写缺失, 它通常要求在继续执行写操作之前取回数据块 (详情参考 5.3.3 节精解中关于写处理的详细介绍); 另一种是写缓冲区阻塞, 当写操作发生时写缓冲已满则可能发生这种情况。因此, 写操作阻塞的时钟周期数为这两种情况阻塞的时钟周期数之和:

$$\text{写操作阻塞的时钟周期数} = [(\text{写的次数} / \text{程序数}) \times \text{写缺失率} \times \text{写缺失代价}] + \text{写缓冲区阻塞}$$

由于写缓冲区阻塞不仅仅取决于频率, 还取决于写操作的执行时机, 因此这样的阻塞不能由一个简单公式来计算。幸运的是, 如果系统中写缓冲区的深度合适 (例如, 4 个或多个字), 并且存储器接收写操作的速率要明显超过程序中平均写频率 (例如, 是它的两倍), 写缓冲区的阻塞将变得很少, 可以将其忽略。如果系统不能达到这些标准, 则说明它设计得不够好; 设计人员应该使用更深的写缓冲区或者使用写回机制。

399

写回机制同样可能产生额外的阻塞。阻塞的产生原因是, 当数据块被替换时需要将其写回到主存中。我们将在 5.8 节中对此进行更详细的讨论。

在大部分写直达 cache 结构中, 读和写的缺失代价是一样的 (都是从主存中取回数据块的时间)。如果假设写缓冲区阻塞可以被忽略, 那么我们可以合并读写操作并共用一个缺失率和缺失代价:

$$\text{存储器阻塞时钟周期数} = (\text{存储器访问次数} / \text{程序数}) \times \text{缺失率} \times \text{缺失代价}$$

也可以表示如下:

$$\text{存储器阻塞时钟周期数} = (\text{指令数} / \text{程序数}) \times (\text{缺失数} / \text{指令}) \times \text{缺失代价}$$

让我们通过一个简单的例子来帮助理解 cache 的性能对处理器性能的影响。

01 例题·计算 cache 性能

假设指令 cache 的缺失率为 2%，数据 cache 的缺失率为 4%，处理器的 CPI 为 2，没有存储器阻塞，且每次缺失的代价为 100 个时钟周期，那么配置一个从不发生缺失的理想的 cache，处理器的速度快多少？这里假定全部 load 和 store 的频率为 36%。

01 答案

根据指令计数器 (I)，由指令缺失引起的时钟周期损失数为

$$\text{指令缺失时钟周期数} = I \times 2\% \times 100 = 2.00 \times I$$

由于所有 load 和 store 指令出现的频率为 36%，我们可以计算出数据缺失引起的时钟周期损失数：

$$\text{数据缺失时钟周期数} = I \times 36\% \times 4\% \times 100 = 1.44 \times I$$

400

总的存储器阻塞时钟周期数为 $2.00 \times I + 1.44 \times I = 3.44 \times I$ ，每条指令的存储器阻塞超过 3 个时钟周期。因此，包括存储器阻塞在内的总的 CPI 是 $2 + 3.44 = 5.44$ 。由于指令计数器或时钟频率都没有改变，CPU 执行时间的比率为

$$\begin{aligned} & \text{有阻塞的 CPU 执行时间 / 配置理想 cache 的 CPU 执行时间} \\ &= (I \times \text{CPI}_{\text{阻塞}} \times \text{时钟周期}) / (I \times \text{CPI}_{\text{理想}} \times \text{时钟周期}) \\ &= \text{CPI}_{\text{阻塞}} / \text{CPI}_{\text{理想}} = 5.44 / 2 \end{aligned}$$

因此，配置了理想的 cache 的 CPU 的性能是原来的 $5.44 / 2 = 2.72$ 倍。 □

如果处理器速度很快，而存储系统却不快，又会发生什么？在第 1 章介绍的 Amdahl 定律提醒我们这样一个事实：存储器阻塞花费的时间占据执行时间的比例会上升。一些简单的例子会说明这个问题有多严重。假设我们加速上面例子中的计算机，通过改进流水线，在不改变时钟频率的情况下，将 CPI 从 2 降到 1。那么具有 cache 缺失的系统的 CPI 为 $1 + 3.44 = 4.44$ ，而配置理想的 cache 的系统性能是它的

$$4.44 / 1 = 4.44$$

倍。存储器阻塞所花费的时间占据整个执行时间的比例则从

$$3.44 / 4.44 = 63\%$$

上升到

$$3.44 / 4.44 = 77\%$$

同样，仅仅提高时钟频率而不改进存储系统也会因 cache 缺失的增加而加剧性能的流失。

前面的例子和等式是建立在命中时间不计入计算 cache 性能的假设之上。很明显，如果命中时间增加，那么从存储系统中存取一个字的总时间也会增加，继而导致处理器时钟周期的增加。我们还将看到其他一些实例以了解导致命中时间略微增加的原因，一个例子是 cache 容量的增加。显然，一个大容量的 cache 访问时间也较长，就像图书馆的书桌很大（有 $3m^2$ ），要找到桌上的一本书必然要花费很长的时间。命中时间的增加相当于又增加了一级流水线，因为 cache 命中操作需要多个时钟周期完成。尽管计算深度流水对性能的影响会更复杂，但在某种程度上，大容量 cache 命中时间的增加反而会影响命中率的改进使其不起作用，从而导致处理器性能的下降。

401

为了分别找到在命中和缺失情况下数据访问时间对性能影响的证据，设计人员有时会使用平均存储器访问时间（AMAT）作为检测 cache 设计的方法。平均存储器访问时间是综合考虑了命中、缺失以及不同访问的频率后得出的访存平均时间，它等于下面的公式：

$$\text{AMAT} = \text{命中时间} + \text{缺失率} \times \text{缺失代价}$$

01 例题·计算平均存储器访问时间

处理器时钟周期的时间为 1ns，缺失代价是 20 个时钟周期，缺失率为每条指令 0.05 次缺失，cache 访问时间（包括命中判断）为 1 个时钟周期。假设读操作和写操作的缺失代价相同并且忽略其他写阻塞。请计算 AMAT。

01 答案

每条指令的平均存储器访问时间为

$$\text{AMAT} = \text{命中时间} + \text{缺失率} \times \text{缺失代价} = 1 + 0.05 \times 20 = 2 \text{ 个时钟周期}$$

即 2ns。 □

下一节我们将讨论另一种 cache 组织结构，这种结构减少了缺失率，但是有时可能会增加命中时间。在 5.15 节中我们将给出其他的例子。

5.4.1 通过更灵活地放置块来减少 cache 缺失

到目前为止，我们将一个块放入 cache 中，采用的是最简单的定位机制：一个块只能放到 cache 中一个明确的位置。正如前面所述，这种方法称为直接映射，因为存储器中任何一块都被直接映射到存储器层次结构中较高层的唯一位置。实际上，有一整套放置块的方法。**直接映射**，是一种极端的情况，此时一个块被精确地放到一个位置。

402 另一种极端方式是：一个块可以被放置在 cache 中的任何位置。这种机制称为**全相联** (fully associative)，因为存储器中的块可以与 cache 中任何一项相关。在全相联 cache 中要找一个指定的块，由于该块可能被存放在 cache 中的任何位置，因此需要检索 cache 中所有的项。为了使检索更加有效，它是由一个与 cache 中每个项都相关的比较器并行完成的。这些比较器加大了硬件开销，因而，全相联只适合块数较少的 cache。

介于直接映射和全相联之间的设计是**组相联** (set associative)。在组相联 cache 中，每个块可被放置的位置数是固定的。每个块有 n 个位置可放的 cache 被称作 n 路组相联 cache。一个 n 路组相联 cache 由很多个组构成，每个组中有 n 块。根据索引域，存储器中的每个块对应到 cache 中唯一的组，并且可以放在这个组中的任何一个位置上。因此，组相联映射将直接映射和全相联映射结合起来：一个块首先被直接映射到一个组，然后检索该组中所有的块判断是否匹配。例如，图 5-14 是根据这三种策略，块 12 被放置在一个容量为 8 块的 cache 中的情况。

- 全相联 cache：cache 的一种组织方式，块可以放置到 cache 中的任何位置。
- 组相联 cache：cache 的另一种组织方式，块可以放置到 cache 中的部分位置（至少两个）。

回想直接映射的 cache，一个存储块的位置是这样给出的：

$$(\text{块号}) \bmod (\text{cache 中的块数})$$

而在组相联 cache 中，包含存储块的组是这样给出的：

$$(\text{块号}) \bmod (\text{cache 中的组数})$$

由于该块可能被放在组中的任何一个位置，因此组中所有块的标记都要被检索。而在全相联 cache 中，块可以被放在任何位置，因此 cache 中全部块的标记都要被检索。

我们同样可以把所有的块定位策略看成是组相联的一个特例。图 5-15 显示了一个 8 块的 cache 可能的相联结构。直接映射 cache 是一个简单的一路组相联 cache：cache 的每项有一个

块，并且每组只有一个元素。有 m 项的全相联 cache 可以看成是一个简单的 m 路组相联 cache，它只有一个组，组里有 m 块，每一项可以放在该组的任何一块中。

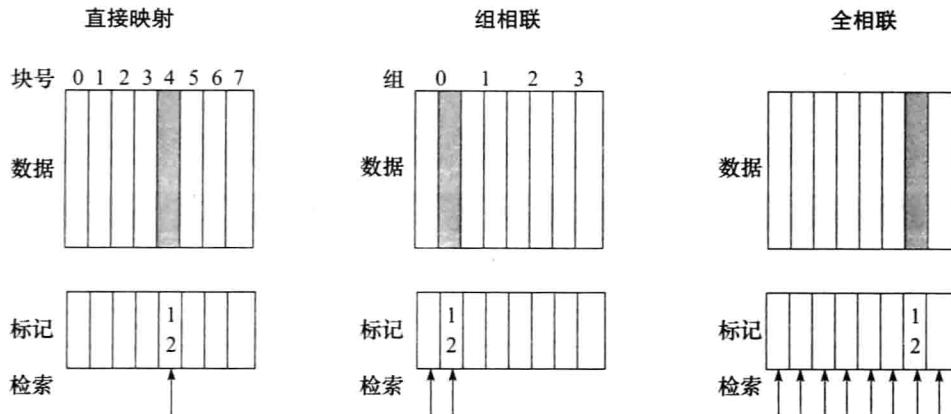


图 5-14 地址为 12 的主存块在 cache 中的位置，cache 容量为 8 块，采用直接映射、组相联以及全相联机制。在直接映射方式下，主存块 12 只能放置在 cache 中唯一的块中，该块为 $(12 \bmod 8) = 4$ 。在两路组相联 cache 中，有 4 个组，主存块 12 必须放在第 $(12 \bmod 4) = 0$ 组中；主存块可以放在该组的任意位置。在全相联方式下，块地址为 12 的主存块可以放在 cache 中 8 个块的任意一块

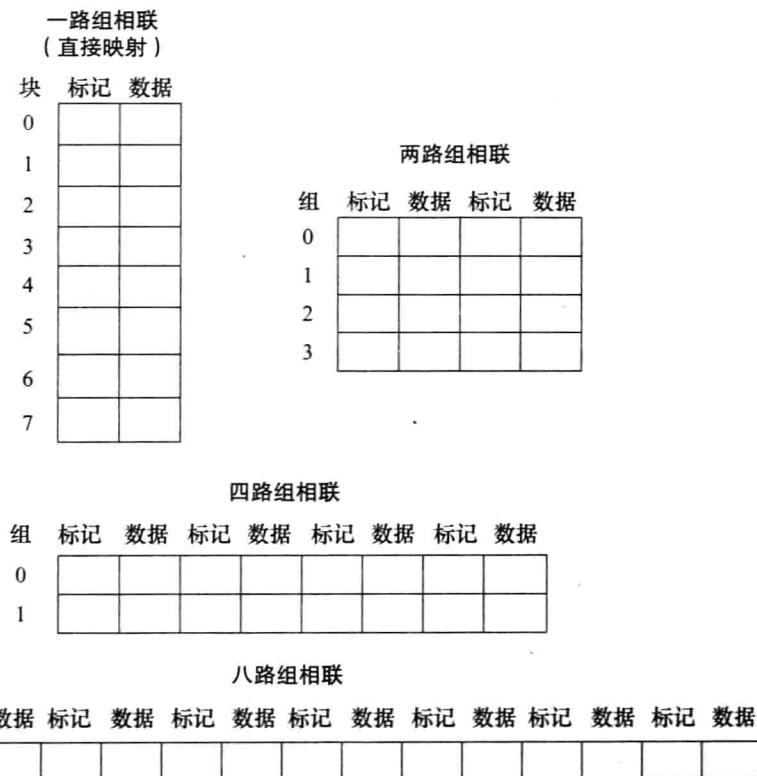


图 5-15 一个拥有 8 个块的 cache 被配置成直接映射、两路组相联、四路组相联以及全相联结构。cache 中块的总数等于组数乘以相联度。因此，对于一个固定大小的 cache，增加相联度的同时减少了组数，同时也增加了每组的块数。对于容量为 8 个块的 cache，一个八路组相联的 cache 也就等同于一个全相联 cache

提高相联度的好处在于它通常能够降低缺失率，如下例所示。而主要的缺点则是增加了命中时间，稍后我们将详细讨论。

01 例题 · cache 的缺失与相联度

假设有三个小的 cache，每个 cache 都有 4 个块，块大小为 1 个字。第一个 cache 是全相联方式，第二个是两路组相联，第三个是直接映射。若按以下地址 0、8、0、6、8 依次访问时，求每个 cache 的缺失次数。

01 答案

直接映射 cache 最简单，首先让我们判断每个地址对应的 cache 块：

块地址	cache 块
0	$(0 \bmod 4) = 0$
6	$(6 \bmod 4) = 2$
8	$(8 \bmod 4) = 0$

现在，在每次引用后我们填入 cache 的内容，空白项表示无效的块。加粗的项表示在相关引用中，有一个新的项被加入 cache 中，未加粗的项则表示 cache 中旧的项。

被访问的存储器的块地址	命中/缺失	引用后 cache 中的内容			
		0	1	2	3
0	缺失	主存 [0]			
8	缺失	主存 [8]			
0	缺失	主存 [0]			
6	缺失	主存 [0]		主存 [6]	
8	缺失	主存 [8]		主存 [6]	

直接映射 cache 的 5 次访问产生 5 次缺失。

组相联 cache 有两组（组 0 和 1），每组有两个块，我们首先来确定每个块地址映射到哪一组：

块地址	cache 组
0	$(0 \bmod 2) = 0$
6	$(6 \bmod 2) = 0$
8	$(8 \bmod 2) = 0$

由于缺失时，我们需要选择替换组中的某一项，因此需要一个替换规则。组相联 cache 通常会选择替换一组中最近最少使用的块；也就是说，在过去最久的时间用到的那一块将被替换（稍后我们将详细讨论其他替换规则）。使用这个替换策略，每次引用后组相联 cache 中的内容如下所示：

被访问的存储器的块地址	命中/缺失	引用后 cache 中的内容			
		组 0	组 0	组 1	组 1
0	缺失	主存 [0]			
8	缺失	主存 [0]	主存 [8]		
0	命中	主存 [0]	主存 [8]		
6	缺失	主存 [0]	主存 [6]		
8	缺失	主存 [8]	主存 [6]		

注意到当块 6 被访问时，它将块 8 替换掉了，因为比起块 0，块 8 是最近最少被使用的那一块。两路组相联 cache 总共有 4 次缺失，比直接映射的 cache 少一次。

全相联 cache 有 4 个块（在一组中），存储器中任意一块可放到 cache 的任何位置。全相联 cache 性能最好，仅有 3 次缺失。

被访问的存储器的块地址	命中/缺失	引用后 cache 中的内容			
		块 0	块 1	块 2	块 3
0	缺失	主存 [0]			
8	缺失	主存 [0]	主存 [8]		
0	命中	主存 [0]	主存 [8]		
6	缺失	主存 [0]	主存 [8]	主存 [6]	
8	命中	主存 [0]	主存 [8]	主存 [6]	

对于这一系列的访问，三次缺失是我们可以得到的最好结果，因为有三个不同地址的块被访问。注意，如果 cache 中有 8 个块，两路组相联 cache 将不会发生替换（请读者自己验证），并且缺失次数与全相联 cache 的一样多。同样，如果有 16 块，这 3 种 cache 会有相同的缺失次数。上面的例子已经说明了在判断 cache 性能时，cache 容量和相联度不能分开考虑。□

相联度能使缺失率下降多少呢？图 5-16 显示了一个容量为 64KiB，块大小为 16 字的数据 cache，当相联度从直接映射到八路组相联变化时性能的改进情况。从一路组相联到两路组相联，缺失率下降了大约 15%，但是更高的相联度对缺失率的改善就很小了。

406

5.4.2 在 cache 中查找一个块

现在，我们考虑在组相联的 cache 中如何查找一个 cache 块。正如在直接映射 cache 中一样，组相联 cache 中每一块都包含一个地址标记用来给出块地址。在被选中的组中每一块的标记都要进行检测，从而判断是否和来自处理器的块地址相匹配。图 5-17 解析了地址。索引值用来选择包含所需地址的组，该组中所有块的标记都将被检索。由于速度是最根本的，所以被选中的组中所有块的标记并行检索。就像在全相联 cache 中一样，组相联 cache 使用顺序检索将使得命中时间太长。

相联度	数据缺失率
1	10.3%
2	8.6%
4	8.3%
8	8.1%

图 5-16 使用与内置 FastMATH 处理器相似的 cache 结构，相联度从一路到八路，采用 SPEC CPU2000 基准测试程序测出的数据 cache 缺失率。10 个 SPEC CPU2000 测试程序的结果来自 Hennessy 和 Patterson (2003)

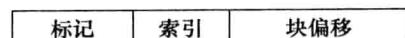


图 5-17 组相联或者直接映射 cache 中地址的三个组成部分。索引位用来选择一个组，标记位用来和选中组中的块进行比较来选择块，块偏移是块中被请求数据的地址

如果 cache 总容量保持不变，提高相联度就增加了每组中的块数，也就是并行查找时同时比较的次数：相联度每增加到两倍就会使每组中的块数加倍而使组数减半。相应地，相联度每增加到两倍，检索位就会减少 1 位，标记位增加 1 位。在全相联 cache 中，只有一组有效，所有块必须并行检测，因此没有索引，除了块偏移地址，整个地址都需要和每个 cache 块的标记进行比较。换句话说，我们不使用索引位就可以查找整个 cache。

在直接映射 cache 中，只需要一个比较器，这是因为每一项只能对应 cache 中唯一的块，并且，我们通过索引就能很简单地访问 cache。图 5-18 是一个四路组相联 cache，需要 4 个比较器以及一个 4 选 1 的多路选择器，用来在选定组中的 4 个成员之间进行选择。cache 访问包括检索相应的组，然后在组中检测标记。一个组相联 cache 的开销包括额外的比较器以及由于对组里数据块进行比较和选择而产生的延迟。

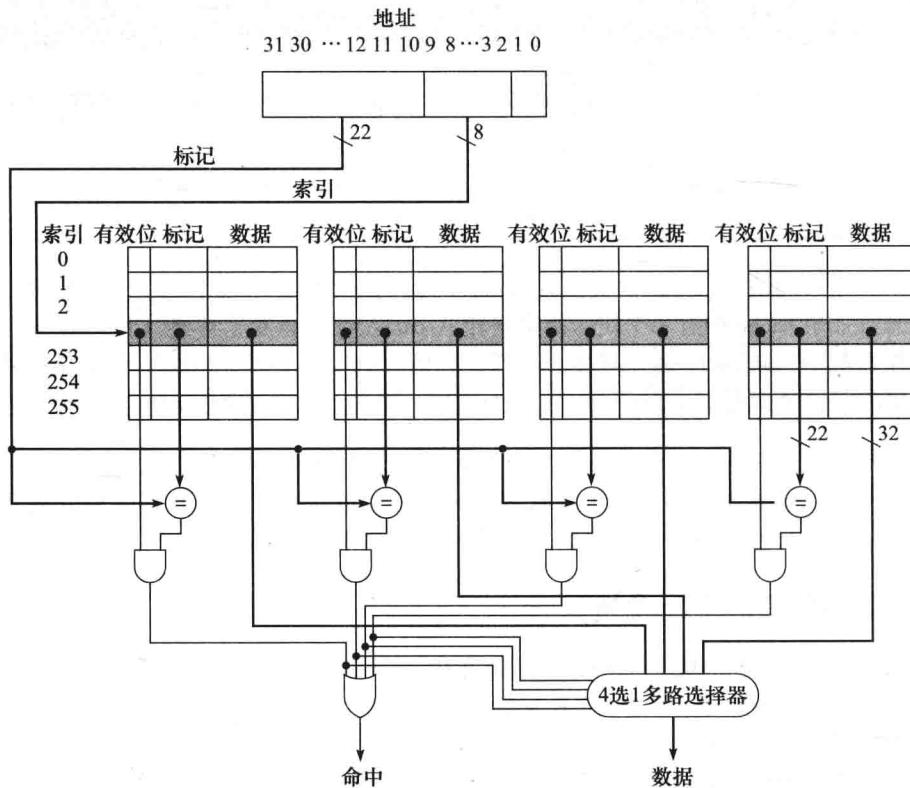


图 5-18 实现一个四路组相联的 cache 需要 4 个比较器和一个 4 选 1 的多路选择器来判断被选中的组中哪一个单元（如果有）与标记匹配。比较器的输出通过使用带有译码选择信号的多路选择器在选中组里的 4 个块之中选择一个数据。在一些具体实现中，cache RAM 数据部分的输出使能信号可以用来选择驱动输出的组中的数据项。输出使能信号来自比较器，使得匹配的单元驱动数据的输出。这种结构不需要使用多路选择器

在任何存储器层次结构中选择直接映射、组相联，还是全相联映射，需要在缺失代价和相联度实现的代价之间进行权衡，既要考虑时间，也要考虑额外的硬件。

01 精解 内容可寻址存储器 (Content Addressable Memory, CAM) 是一种将比较器和存储单元结合在一个部件上的电路结构。它不像 RAM 那样根据地址读数据，而是由用户提供数据，然后 CAM 查看它是否有副本并且返回匹配行的索引。CAM 的出现意味着设计者能提供更高的相联度，这比在 SRAM 和比较器之外还需要构建硬件才能实现的相联度还要高。在 2013 年，CAM 更大的容量和功耗使得两路和四路组相联结构一般采用标准的 SRAM 和比较器构建，八路以及更多路组相联的结构则由 CAM 构建。

407
408

5.4.3 替换块的选择

当直接映射的 cache 发生缺失时，被请求的块只能放置于 cache 中唯一位置，而原先占据

那个位置的块就必须被替换掉。在相联的 cache 中，被请求的块放置在什么位置需要进行选择，因此替换哪一块也要进行选择。在全相联 cache 中，所有的块都将可能被替换。在组相联 cache 中，我们将在选中的组中挑选被替换的块。

最常用的方法是最近最少使用（Least Recently Used, LRU）法，也是我们在前面例子中使用的方法。在 LRU 算法中，被替换的块是最久没有使用的那一块。前面组相联的例子中就使用了 LRU 算法，这也是为什么我们替换主存（0）那块而不是主存（6）。

② 最近最少使用：一种替换策略，总是替换很长时间没有使用的块。

LRU 替换算法的实现是通过跟踪每一块的相对使用情况。对于一个两路组相联 cache，跟踪组中两个数据项的使用情况可以这样实现：在每组中单独保留一位，通过设置该位指出哪一项被访问过。当相联度提高时，LRU 的执行就变得困难些；在 5.8 节中，我们将会讨论另一种替换机制。

01 例题·标记位大小与组相联

提高相联度需要更多比较器，同时 cache 块中的标记位数也需要增加。假设一个 cache 有 4 096 个块，块大小为 4 个字，地址为 32 位，请分别计算在直接映射、两路组相联、四路组相联和全相联映射中，cache 的总组数以及总的标记位数。

01 答案

由于块大小为 $16 (= 2^4)$ 字节，32 位地址域中的 $32 - 4 = 28$ 位用来提供索引和标记位。直接映射中组数和块数一样，由于 $\log_2(4096) = 12$ ，因此有 12 位是索引位；因此总的标记位数是 $(28 - 12) \times 4096 = 16 \times 4096 = 66\text{Kb}$ 。

相联度每增加 1 倍，组数就会减少 1/2，因此用来索引 cache 的位数也要相应减 1，而标记位则是增 1。因此，对于一个两路组相联 cache，有 2 048 个组，总的标记位数为 $(28 - 11) \times 2 \times 2048 = 34 \times 2048 = 70\text{Kb}$ 。而四路组相联中组数为 1 024，那么总的标记位数为 $(28 - 10) \times 4 \times 1024 = 72 \times 1024 = 74\text{Kb}$ 。

对于全相联 cache，只有一个有 4 096 个块的组，标记位是 28 位，因此总的标记位数是 $28 \times 4096 \times 1 = 115\text{Kb}$ 。

□

409

5.4.4 使用多级 cache 结构减少缺失代价

所有现代计算机都使用 cache。为了进一步减小现代处理器高时钟频率与日益增长的 DRAM 访问时间之间的差距，大多数微处理器都会增加额外一级 cache。这种二级 cache 通常位于芯片内，当一级 cache 缺失时就会访问它。如果二级 cache 中包含所需的数据，那么一级 cache 的缺失代价就是二级 cache 的访问时间，这要比访问主存快得多。如果一级和二级 cache 中均不包含所需的数据，就需要访存，这样就会产生更大的缺失代价。

使用二级 cache 后，性能能改进多少？下面这个例子将会告诉我们。

01 例题·多级 cache 的性能

假定我们的处理器基本的 CPI 为 1.0，所有访问在一级 cache 中均命中，时钟频率为 4GHz。假设主存访问时间为 100ns，其中包括缺失处理时间。设一级 cache 中每条指令缺失率为 2%。如果增加一个二级 cache，命中或缺失访问的时间都是 5ns，而且容量大到必须使访问主存的缺

失率减少到 0.5%，这时的处理器速率能提高多少？

01 答案

主存的缺失代价为

$$100\text{ns}/(0.25\text{ns}/\text{时钟周期}) = 400 \text{ 个时钟周期}$$

只有一级 cache 的处理器的有效 CPI 由下列公式给出：

$$\text{总的 CPI} = 1.0 + \text{每条指令的存储器阻塞时钟周期} = 1.0 + 2\% \times 400 = 9$$

对于两级 cache，一级 cache 缺失时可以由二级 cache 或者主存来处理。访问二级 cache 时的缺失代价为

410

$$5\text{ns}/(0.25\text{ns}/\text{时钟周期}) = 20 \text{ 个时钟周期}$$

如果缺失能由二级 cache 处理，那么这就是整个缺失代价。如果缺失处理需要访存，总的缺失代价就是二级 cache 和主存的访问时间之和。

因此，对一个两级的 cache，总的 CPI 是两级 cache 的阻塞时钟周期和基本 CPI 的总和：

$$\begin{aligned}\text{总的 CPI} &= 1 + \text{一级 cache 中每条指令的阻塞} + \text{二级 cache 中每条指令的阻塞} \\ &= 1 + 2\% \times 20 + 0.5\% \times 400 = 1 + 0.4 + 2.0 = 3.4\end{aligned}$$

因此，有二级 cache 的处理器性能是没有二级 cache 处理器性能的

$$9.0/3.4 = 2.6$$

倍。

我们还可以使用另一种方法来计算阻塞时间。在二级 cache 命中的阻塞周期为 $(2\% - 0.5\%) \times 20 = 0.3$ ；而访问主存的阻塞周期必须同时包括访问二级 cache 和访问主存的时间，为 $0.5\% \times (20 + 400) = 2.1$ 。对它们求和为 $1.0 + 0.3 + 2.1$ ，同样等于 3.4。□

一级 cache 和二级 cache 的设计思想明显不同，这是因为对于单级 cache，另一级 cache 的存在改变了最佳选择。特别是两级 cache 的结构使得一级 cache 致力于减少命中时间获得较短的时钟周期或者较少的流水级，二级 cache 则主要针对改善缺失率以减少长时间的访存代价。

通过将每一级 cache 与最优化单级 cache 的设计进行比较，我们可以看出这些变化对两级 cache 的影响。与单级 cache 相比，**多级 cache** (multilevel cache) 中的一级 cache 通常很小。另外，一级 cache 的块容量通常也很小，再伴随小容量的 cache 使得缺失代价降低。相比之下，由于二级 cache 的访问时间不是关键，因此二级 cache 的容量比一般的单级 cache 要大得多，块容量也比单级 cache 中的要大。它还经常使用比一级 cache 更高的相联度以减少缺失率。

② 多级 cache：存储系统由多级 cache 组成，而不仅仅只有主存和一个 cache。

01 理解程序性能 我们用尽一切方法对冒泡排序 (Bubble Sort)、快速排序 (Quicksort) 和基数排序 (Radix Sort) 等进行分析，希望找到最好的排序算法。图 5-19a 说明了使用基数排序和快速排序时，指令执行的情况。果然，对于大的数组，在操作次数上，基数排序比快速排序要有优势。图 5-19b 是每个排序项平均所需的时间，而不是执行的指令数。我们可以看到开始两条曲线的轨迹与图 5-19a 中相似，但是随着排序数据的增加，基数排序的曲线开始偏离，这是为何？图 5-19c 用每项排序平均 cache 缺失数解答了这个问题：快速排序一直有比基数排序少得多的每项缺失数。

411

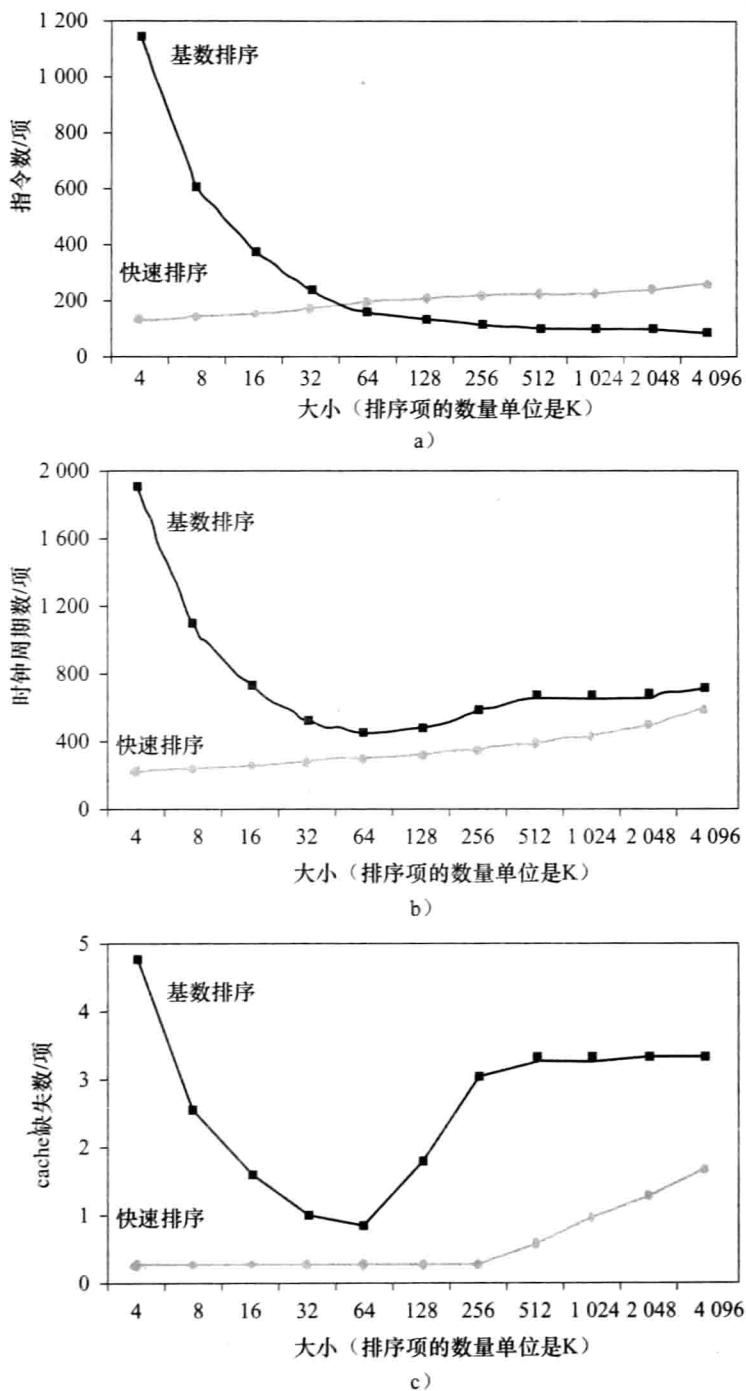


图 5-19 比较快速排序和基数排序。a) 每个排序项平均执行指令数; b) 每个排序项平均时间; c) 每个排序项平均 cache 缺失数。数据来自于 LaMarca 和 Ladner 在 1996 年的一篇文章。由于这些结果，人们又发明了新版本的基数排序，将存储器层次结构考虑进来，以重新获得算法的优势（见 5.15 节）。cache 优化的基本思想是在某个块被替换前，重复使用该块中所有的数据

标准算法分析通常会忽视存储器层次结构的影响，正如更快的时钟频率和摩尔定律让体系结构设计者从指令流中获取所有的性能，合理地使用存储器层次结构是获得高性能的关键。如我们在概述中所说的，理解存储器层次结构的行为对于理解当今计算机的程序性能是十分关键的。

5.4.5 通过分块进行软件优化

由于存储器层次对程序性能具有非常重要的影响，因此许多软件优化技术通过对 cache 中的数据进行重用，提升了数据的时间局部性并因此降低缺失率，这些优化技术可以大大提高 cache 的性能。

在处理数组时，如果能够将数组元素按照访问顺序存放在存储器中，则能够获得很好性能。假定要处理多个数组，这些数组有些按行访问，而另外一些按列访问。因为在每次循环迭代中，既有按行访问的数组，又有按列访问的数组，因此无论采用按行存储或按列存储的方式都不能解决问题。

与对一个数组进行整行或整列操作不同，分块算法对子矩阵（或称为块）进行操作。其目标是在数据被替换出去之前，最大限度地对已装入 cache 的数据进行访问，即通过提升时间局部性的方法来降低 cache 缺失率。

例如，DGEMM 的内循环（第 3 章中图 3-21 中的 4~9 行）如下：

```
for (int j = 0; j < n; ++j)
{
    double cij = C[i+j*n]; /* cij = C[i][j] */
    for( int k = 0; k < n; k++)
        cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
    C[i+j*n] = cij; /* C[i][j] = cij */
}
}
```

该程序段读取了数组 B 中的所有 $N \times N$ 个元素，另外反复读取了数组 A 中对应行中的 N 个元素，并且对数组 C 中对应行的 N 个元素进行了写操作。（注释使得矩阵的行列更容易识别。）图 5-20 给出了访问三个数组的大致情况。一个深色阴影表示一次最近访问的元素，浅色阴影表示早期访问的元素，而白色表示还没有被访问的元素。

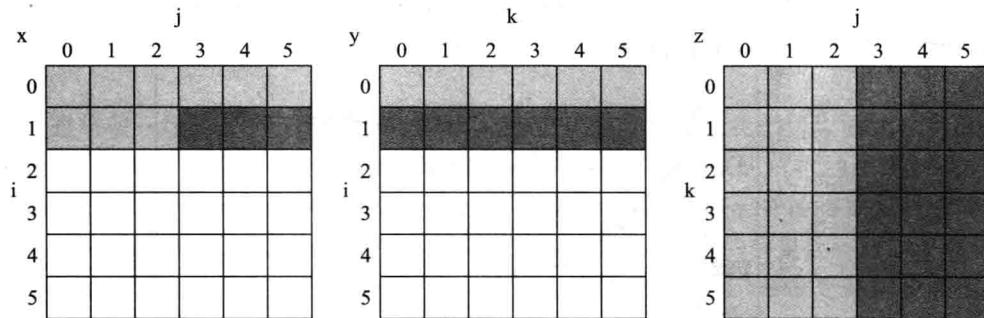


图 5-20 三个数组 C、A、B 的访问情况， $N = 6$, $i = 1$ 。数组元素的访问时间情况用阴影表示：白色表示该元素未被访问，浅色阴影表示早期访问，深色阴影表示新的访问。与图 5-21 相比，反复读取 A 和 B 的元素以计算新的元素 x。行列边上的变量 i、j 和 k 用于表示对数组的访问

很明显，缺失的次数依赖于 N 和 cache 的容量。如果在不发生冲突的前提下三个 $N \times N$ 矩阵的所有元素都在 cache 中，则没有任何问题。与第 3 章和第 4 章相同，假定 DGEMM 中的矩阵大小为 32×32 。每个矩阵有 $32 \times 32 = 1024$ 个元素，且每个元素为 8 字节，因此这三个矩阵将占据 24KiB 的空间，这意味着 Intel Core i7 (Sandy Bridge) 中 32KiB 的数据 cache 足够使用。

如果 cache 中能够保持一个 $N \times N$ 的矩阵和一个长度为 N 的行，则 A 的第 i 行和数组 B 可驻留在 cache 中。如果 cache 容量再小，将可能导致 B 和 C 访问都缺失。在最坏情况下， N^3 次操作需要 $2N^3 + N^2$ 次存储器字的访问。

为了确保正在访问的元素能够在 cache 中命中，可把原先的程序改为每次循环迭代只计算一个子矩阵。因此，可通过参数 BLOCKSIZE 使得第 4 章图 4-80 中的 DGEMM 程序循环处理大小为 BLOCKSIZE 的数组，其中 BLOCKSIZE 称为分块因子（blocking factor）。

图 5-21 给出了 DGEMM 的分块版本。do_block 函数使用三个新的参数 si、sj 和 sk 表示每个子数组的起始位置。do_block 的内层循环以 BLOCKSIZE 为步进长度进行计算，而不是 B 和 C 的长度。gcc 优化器通过“inling”功能去除任何调用开销；也就是说，它将直接插入代码以避免传统的参数传递和返回地址的保存与恢复。

```

1 #define BLOCKSIZE 32
2 void do_block (int n, int si, int sj, int sk, double *A, double
3 *B, double *C)
4 {
5     for (int i = si; i < si+BLOCKSIZE; ++i)
6         for (int j = sj; j < sj+BLOCKSIZE; ++j)
7             {
8                 double cij = C[i+j*n];/* cij = C[i][j] */
9                 for( int k = sk; k < sk+BLOCKSIZE; k++ )
10                     cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11                     C[i+j*n] = cij; /* C[i][j] = cij */
12             }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16     for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17         for ( int si = 0; si < n; si += BLOCKSIZE )
18             for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19                 do_block(n, si, sj, sk, A, B, C);
20 }
```

图 5-21 图 3-21 中 DGEMM 的 cache 分块版本。假定 C 初始化为 0。do_block 函数来源于第 3 章中的基本 DGEMM，使用了新参数来指明大小为 BLOCKSIZE 的子矩阵的起始位置。gcc 优化器通过内联的 do_block 函数消除函数开销

图 5-22 显示了使用分块之后对三个数组的访问情况。只考虑容量失效，访问存储器的总字数为 $2N^3/BLOCKSIZE + N^2$ 次，大约降低到原来的 $1/BLOCKSIZE$ 。分块技术同时利用了空间局部性和时间局部性，其中访问 A 时利用了空间局部性，访问 B 时则利用了时间局部性。

414

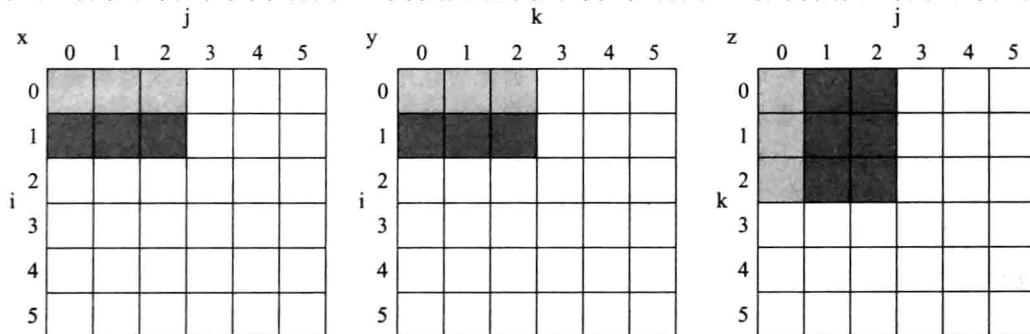


图 5-22 当 $BLOCKSIZE = 3$ 时数组 C、A 和 B 的访问时间。注意，与图 5-20 相比，访问的元素减少了

虽然分块技术的目标是降低 cache 缺失率，但是分块技术也可用来帮助寄存器的分配。通过采用较小的分块，使得一个块可以驻留在寄存器中，程序中可以将 load 和 store 操作的数量大大减少，从而提高性能。

图 5-23 给出了采用分块技术的 DGEMM 的性能产生的影响，其中矩阵尺寸逐渐增加。可以看出，当矩阵尺寸增大到不能在 cache 中完全容纳这三个矩阵时，性能下降为最优情况的一半。即

使矩阵尺寸为 960×960 时（是第 3 章和第 4 章矩阵尺寸的 900 倍），性能仅下降了不到 10%。

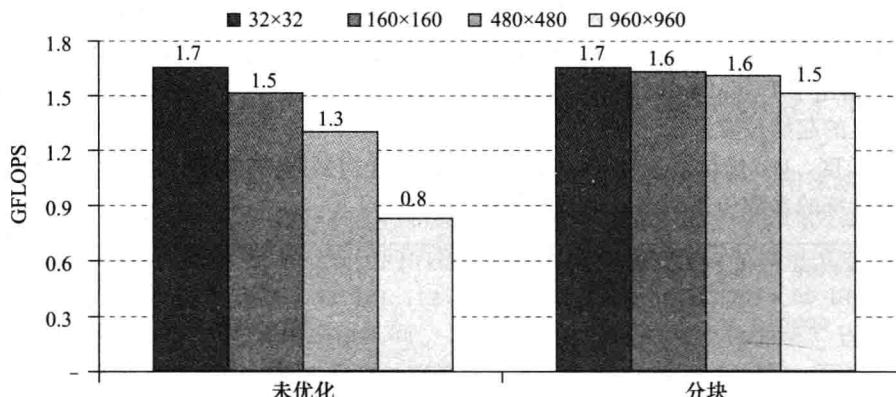


图 5-23 未优化的 DGE MM (图 3-21) 和采用 cache 分块技术的 DGE MM (图 5-21) 的性能比较。在采用 cache 分块技术时，矩阵尺寸由 32×32 (三个矩阵均留在 cache 中) 到 960×960

01 精解 使用多级 cache 会产生一些复杂情况。首先，存在多种不同类型的缺失以及相应的缺失率。在“多级 cache 的性能”的例子中，我们看见了一级 cache 缺失率以及全局缺失率 (global miss rate)，即在所有级 cache 中都缺失的那部分访问。同时还有二级 cache 缺失率，是二级 cache 所有缺失次数和访问次数的比率。这个缺失率称为二级 cache 的局部缺失率 (local miss rate)。由于一级 cache 过滤了一些访问，特别是那些具有较好的空间局部性和时间局部性的访问，这就使得二级 cache 的局部缺失率要大大高于全局缺失率。在“多级 cache 的性能”的例子中，可以计算出二级 cache 的局部缺失率为 $0.5\%/2\% = 25\%$ ！幸运的是，全局缺失率决定了访问主存的次数。

- 全局缺失率：在多级 cache 的所有级中都缺失的那部分访问。
- 局部缺失率：在多级 cache 中，某一级 cache 的缺失率。

415
416

01 精解 乱序处理器（见第 4 章）在缺失时仍能执行指令，因而性能更加复杂。我们使用每条指令缺失数来代替指令缺失率和数据缺失率，公式如下：

$$\text{存储器阻塞周期数 / 指令数} = \text{缺失数 / 指令数} \times (\text{总的缺失延迟} - \text{重叠的缺失延迟})$$

计算重叠的缺失延迟没有通用的方法，因此对乱序处理器的存储器层次结构进行评估需要模拟处理器和存储器层次结构。只有观测到每次缺失时处理器的执行情况，我们才能知道缺失时处理器是阻塞下来等待数据还是在执行其他工作。一个指导方针是处理器通常会隐藏在一級 cache 缺失而在二级 cache 命中时的那部分缺失代价中，但是却很少隐藏二级 cache 的缺失代价。

01 精解 对算法性能的挑战在于：对相同的结构采用不同的实现方法，包括 cache 容量、相联度、块大小以及 cache 的数量，都会使得存储器层次结构变得多样化。为了应对这些变化，近来一些数值库将它们的算法变得参数化，通过实时搜索参数空间来找到特定计算机上的最佳组合。这种方法称为自动调节 (autotuning)。

01 小测验

有关多级 cache 的设计，下面哪些是正确的？

1. 一级 cache 更关注命中时间，二级 cache 则更关注缺失率。
2. 一级 cache 更关注缺失率，二级 cache 则更关注命中时间。

5.4.6 小结

这一节集中讨论了 4 个问题：cache 性能、利用相联度来降低缺失率、利用多级 cache 结构来降低缺失代价和采用软件优化技术提高 cache 的有效性。

存储系统对程序执行时间有着重要影响。存储器阻塞时钟周期数取决于缺失率和缺失代价。在 5.8 节中将会看到我们面临的挑战，就是如何降低这些因素中的一个而不会影响到存储器层次结构中的其他关键因素。

为了降低缺失率，我们对相联定位方法进行研究。这种方法通过将数据块更灵活地放置在 cache 中以降低缺失率。全相联机制允许将块放在 cache 中的任何位置，但是仍然需要查找 cache 中的每一块以找到所需的数据块。较高的成本使得大容量的全相联 cache 的实现不切实际。而组相联 cache 则更加可行，我们只需要在索引唯一选中的组中进行查找。组相联 cache 缺失率更高，但是访问速度更快。使用何种相联度能达到最佳性能不仅取决于技术本身，还取决于实现的细节。

我们探讨了多级 cache 技术，它通过使用一个大的二级 cache 来处理一级 cache 的缺失，从而降低了缺失代价。二级 cache 已经逐渐普遍，这是因为设计者发现由于硅的局限以及高时钟频率的要求，一级 cache 的容量已经无法更大了。而二级 cache 的容量通常是一级 cache 的 10 倍甚至更多，因而能处理很多一级 cache 缺失引起的访问。在这些情况下，缺失代价就是二级 cache 的访问时间（通常小于 10 个处理器周期）而不是主存访问时间（通常大于 100 个处理器周期）。和相联度考虑相似，在二级 cache 容量和访问时间之间的权衡取决于实现过程中的很多方面。

最后，针对存储器层次对性能影响的重要性，我们讨论了如何对算法进行变换来提高 cache 性能，主要讨论了针对大数组进行分块的技术。

5.5 可信存储器层次

本章前面所有的讨论集中在如何提高存储器层次的性能上，但是不能忘记如果可信性不够，即使速度再快也不具有吸引力。正如在第 1 章所述，最好的可信性方法是冗余技术。本节将首先重温与失效相关的术语定义和度量，然后将讲述如何采用冗余技术构造可靠的存储器。

5.5.1 失效的定义

假定有某类服务的需求，针对该需求，用户可以看到一个系统在两种服务状态之间做出选择：

- 1) 服务实现：交付的服务与需求相符。
- 2) 服务中断：交付的服务与需求不符。

失效导致状态 1 到状态 2 的转换，而由状态 2 转换到状态 1 的过程称为恢复。失效可以是永久性的，也可以是间歇性的。间歇性失效更加复杂一些，因为当一个系统因间歇性失效在两个状态间摇摆时，诊断将会非常困难。而永久性失效的诊断要容易许多。

这种定义将引出两个术语：可靠性和可用性。

可靠性是一个系统或模块能够持续提供用户需求的服务的度量，即从开始使用到失效的时间间隔。因此，平均无故障时间（mean time to failure, MTTF）是一个可靠性度量方法。与之相关的一个术语是年失效率（annual failure rate, AFR），它是指在给定 MTTF 情况下，在一年内预期的器件失效比例。由于从 MTTF 中可能会得到误导性的结果，因此 AFR 会获得更加直观的结果。

418

例题 · 磁盘的 MTTF 和 AFR

当今的一些磁盘号称其 MTTF 为 1 000 000 小时，大约是 $1\,000\,000 / (365 \times 24) = 114$ 年，这意味着这些磁盘永远不会失效。运行搜索引擎等网络服务的仓储式计算机可能有 50 000 台服务器，假定每台服务器有两块磁盘，使用 AFR 计算每年将会有多少块磁盘失效。

答案

一年有 $365 \times 24 = 8\,760$ 小时。1 000 000 小时的 MTTF 意味着 AFR 为 $8\,760 / 1\,000\,000 = 0.876\%$ 。由于系统中有 100 000 块磁盘，因此每年将有 876 块磁盘失效，即平均每天有超过两块磁盘失效！□

服务中断使用维修平均时间（mean time to repair, MTTR）来度量。失效间隔平均时间（mean time between failure, MTBF）= MTTF + MTTR。虽然 MTBF 广泛应用，MTTF 却更加确切。可用性是指系统正常工作时间在连续两次服务中断间隔时间中所占的比例：

$$\text{可用性} = \text{MTTF} / (\text{MTTF} + \text{MTTR})$$

需要注意的是，可靠性和可用性是可以量化的，而可信性是不可量化的。与增加 MTTF 类似，减少 MTTR 同样可以提高可用性。例如，可以采用故障检测、诊断和修复的工具来减少故障维修花费的时间，从而提高可用性。

我们希望系统具有很高的可用性。一种简化的表示方法是“每年中可用性的 9 的数量”。例如，一个很好的网络服务器可提供 4 个或 5 个 9 的可用性。一年有 $365 \times 24 \times 60 = 526\,000$ 分钟，简化表示如下：

- 1 个 9: 90% => 36.5 天的维修时间/年
- 2 个 9: 99% => 3.65 天的维修时间/年
- 3 个 9: 99.9% => 526 分钟的维修时间/年
- 4 个 9: 99.99% => 52.6 分钟的维修时间/年
- 5 个 9: 99.999% => 5.26 分钟的维修时间/年

以此类推。

为了提高 MTTF，可以提高器件的质量，也可以设计能够在器件失效的情况下继续工作的系统。由于一个器件的失效可能不会导致系统的失效，因此需要根据具体情况定义失效。为了明晰差别，使用术语故障来表示一个器件的失效。有如下三种方式可以提高系统的 MTTF：

- 1) 故障避免技术 (fault avoidance)：通过合理构建系统来避免故障的出现。
- 2) 故障容忍技术 (fault tolerance)：采用冗余措施，当发生故障时，通过冗余措施保证系统仍然正常工作。
- 3) 故障预报技术 (fault forecasting)：对故障进行预测，从而允许在器件失效前进行替换。

419

5.5.2 纠正一位错、检测两位错的汉明编码 (SEC/DED)

理查德·汉明 (Richard Hamming) 发明了一种广泛应用于存储器的冗余技术，并因此获得 1968 年的图灵奖。二进制数间的距离对于理解冗余码很有帮助。两个等长二进制数的汉明距离是两个数对应位置不同的位的数量。例如，011011 和 001111 的距离为 2。在一种编码中，如果码字之间的最小距离为 2，且其中有 1 位错误，将会发生什么？这将会将一个有效的码字转化为无效码字。因此，如果能够检测出一个码字是否有效，则可检测出 1 位的错误，称为 1 位错误检测编码。

- 错误检测编码：这种编码方式能够检测出数据中有 1 位错误，但是不能对错误位置进行精确定位，因此不能纠正错误。

汉明使用奇偶校验码进行错误检测。在奇偶校验码中，计算码字中 1 的数量是奇数个还是偶数个。当一个字写入存储器时，奇偶校验位也被写入（1 代表奇数，0 代表偶数）。这就是说， $N+1$ 位码字中 1 的个数永远为偶数。因此，当读出数据时，校验码也被读出并进行检测。如果计算出的校验码与保存的不符，则发生错误。

01 例题

计算十进制数 31 对应的 8 位二进制数的奇偶性，并写出存储器中的表示形式。假设奇偶校验位在最右边，并且假定存储器中最高位发生了翻转，然后将其读回。请问能否检测到错误？如果最高两位同时翻转呢？

01 答案

十进制数 31 的二进制形式为 00011111，有 5 个 1。为了使编码后的码字为偶性，需要向校验位写入 1，也就是 00011111。如果最高位发生翻转，读回的将是 10011111，具有 7 个 1。由于期望码字为偶性，但是计算结果却是奇性，因此报告出错。如果最高两位同时发生翻转，则得到 11011111，具有 8 个 1 或者说具有偶性，因此不能报告出错。□

如果有两位同时出错，该情形下码字的奇偶性不变，因此一位奇偶校验无法检测到该错误。（实际上，一位奇偶校验可以检测到任意奇数个错误，但实际情况是，出现三位错的概率远小于出现两位错的概率，所以一位奇偶校验码主要用于检测 1 位出错。）

当然，奇偶校验码不能纠正错误，汉明想要做到检错的同时又能纠错。如果我们采用最小距离为 3 的码组，那么任意一个发生 1 位错的码字与其对应的合法码之间的距离要小于该非法码与其他合法码字的距离。汉明提出了一种易于理解的映射方法，该方法将数据映射到距离为 3 的码字，为了表达对他的敬意，我们称这种编码方法为汉明纠错码（Hamming Error Correction Code, ECC）。我们采用额外的校验位确定单个错误的位置。下面是计算汉明纠错码的步骤：

- 对数据部分从左到右由 1 开始依次编号，这跟通常采用的从最右边开始由 0 开始编号的做法相反。
- 将所有编号为 2 的整数次幂的位标记为奇偶校验位（1, 2, 4, 8, 16, …）。
- 其他剩余位置用作数据位（位置 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, …）。
- 奇偶校验位的位置决定了其对应的数据位（图 5-24 用图形的方式进行了说明）如下所示：

位置	1	2	3	4	5	6	7	8	9	10	11	12
编码之后的数据位	p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
奇偶校验位覆盖范围	p1	X		X		X		X		X		
	p2		X	X			X	X		X	X	
	p4				X	X	X	X				X
	p8							X	X	X	X	X

图 5-24 用于 8 位数据的汉明纠错码，其中奇偶校验位、数据位及覆盖范围如图所示

420

- 校验位 1 (0001_2) 检查第 1, 3, 5, 7, 9, 11, … 位，这些数位编号的二进制形式最右边一位均为 1 ($0001_2, 0011_2, 0101_2, 0111_2, 1001_2, 1011_2, \dots$)。

- 校验位 2 (0010_2) 检查第 2, 3, 6, 7, 10, 11, 14, 15, …位，这些数位编号的二进制形式右边起第二位均为 1。
- 校验位 4 (0100_2) 检查第 4 ~ 7, 12 ~ 15, 20 ~ 23, …位，这些数位编号的二进制形式右边起第三位均为 1。
- 校验位 8 (1000_2) 检查第 8 ~ 15, 24 ~ 31, 40 ~ 47, …位，这些数位编号的二进制形式右边起第四位均为 1。

注意到每个数据位都被至少两个奇偶校验位覆盖。

5) 设置奇偶校验位，对各组进行偶校验。

如同变魔术一样，你可以通过查看校验位来确定数据位是否出错。采用图 5-24 当中的 12 位码字，如果 4 个校验位组成的二进制数 (p_8, p_4, p_2, p_1) 是 0000_2 ，这说明没有发生错误。但是，如果校验位组成的二进制数为 1010_2 ，也就是十进制数 10 时，汉明纠错码告诉我们第 421 位 (d_6) 出错了。由于是二进制数，所以只需将第十位的数进行取反，就完成了纠错。

01 例题

假定存在某个单字节数据 10011010_2 。首先写出对应的汉明纠错码，然后把第 10 位取反，说明纠错码如何找到并纠正该错误。

01 答案

将校验位的位置空出来，12 位的码字 1 001 1010。

位置 1 检查第 1, 3, 5, 7, 9, 11 位，为使该组为偶校验，我们应当把第 1 位填 0。

位置 2 检查第 2, 3, 6, 7, 10, 11 位，为使该组为偶校验，我们在第 2 位填入 1。

位置 4 检查第 4, 5, 6, 7, 12 位，所以我们在第 4 位填入 1。

位置 8 检查第 8, 9, 10, 11, 12，所以我们在第 8 位填入 0。

最终得到的码字为 011100101010_2 。把数据位第 10 位取反之后变成 011100101110_2 。

校验位 1 为 0 (011100101110_2 有 4 个 1，为偶性，故该组无错误)。

校验位 2 为 1 (011100101110_2 有 5 个 1，为奇性，故该组某个位置上有错误)。

校验位 4 为 1 (011100101110_2 有两个 1，为偶性，故该组无错误)。

校验位 8 为 1 (011100101110_2 有 3 个 1，为奇性，故该组某个位置上有错误)。

校验位 2 和 10 不正确。因为 $2 + 8 = 10$ ，第 10 位肯定是错的。因此，我们将其翻转为 011100101010_2 ，即完成了纠错。□

汉明并没有止步于 1 位纠错码。通过付出增加一位的代价，可以让码组中的最小汉明距离变到 4。这就意味着我们可以做到纠正 1 位错并检测 2 位错。该方法增加了 1 位奇偶校验码，对整个字进行计算校验。这里我们以 4 位的数据字为例，这只需要 7 位就能完成单位错检测。计算出汉明奇偶校验位 H (p_1, p_2, p_3)，这里仍然采用偶校验，最后计算出整个字的偶校验位 p_4 ：

1	2	3	4	5	6	7	<u>8</u>
p_1	p_2	d_1	p_3	d_2	d_3	d_4	p_4

上述用于纠正 1 位错同时检测 2 位错的算法仍像之前那样先计算出纠错码组的奇偶性 H ，最后再计算一下全组的奇偶校验位 p_4 就可以了。以下是可能出现的 4 种情况：

- 1) H 为偶并且 p_4 为偶，这表示没有错误发生。
- 2) H 为奇并且 p_4 为奇，这表明出现了一位可纠正错误。（当出现一位错时， p_4 应当为奇。）
- 3) H 为偶并且 p_4 为奇，这说明出现的仅仅是 p_4 ，因此将 p_4 取反即可。
- 4) H 为奇并且 p_4 为偶，这表示出现了两位错。（当出现两位错时， p_4 应当为偶。）

纠 1 位错检 2 位错 (SEC/DED) 的技术现在广泛应用于服务器的内存。方便的是，8 字节的数据块做 SEC/DED 时只需要恰好一个字节的额外开销。这也是为什么许多双列直插式存储模块宽度为 72 位。

01 精解 为了计算出 SEC 需要的位数，假定 p 表示校验位的位数， d 表示数据位的位数，则整个码字为 $p + d$ 位。如果采用 p 个纠错位指示错误（码字长度为 $p + d$ 的情况下），再加上没有出现错误的情况，不难得到下面的不等式：

$$2^p \geq p + d + 1 \text{ 位, 因此 } p \geq \log(p + d + 1)$$

例如，对 8 位的数据而言， $d = 8$ ，并且 $2^p \geq p + 8 + 1$ ，所以 $p = 4$ 。类似的，数据长度为 16 位时 $p = 5$ ，32 位时 $p = 6$ ，64 位时 $p = 7$ ，以此类推。

01 精解 在大型系统中，出现多位错的概率和整个内存芯片出错的概率变得显著起来。为解决这一问题，IBM 引进了一种叫作 chipkill 的技术，之后很多大型系统都应用了该项技术 (Intel 称他们所用的为 SDDC)。chipkill 本质上类似于磁盘阵列中采用的 RAID 技术 (见 5.11 节)，将数据和校验码分散开来，因此当某一内存芯片全部出错时，可以通过其他内存芯片中的内容对出错的内容进行重建。假定现有一个由 10 000 个处理器构成的集群，其中每个处理器配备 4GiB 内存，IBM 针对为期三年的运行时间计算出了以下不可恢复内存错误出现的比率：

- 仅采用奇偶校验——出现大约 90 000 次不可恢复（或者不可检测）错误，换句话说，每 17 分钟就出现一次。
- 仅采用 SEC/DED——出错大约 3 500 次，换句话说，每 7.5 小时出现一次不可检测或者不可恢复的错误。
- 采用 chipkill——出错 6 次，换句话说，每两个月出现一次不可检测或不可恢复的错误。

因此，用于数据仓库级别的计算机需要采用 chipkill 技术。

01 精解 虽然存储器系统出现 1 位错或者 2 位错的情况比较典型，但是网络系统中可能会出现突发型错误。解决该问题的一个方法是采用循环冗余校验。对于一个具有 k 位的字块来说，发送端生成一个 $n - k$ 位长度的帧校验序列。这样最终发送出的是一个长度为 n 位的序列，并且该序列构成的数字可以被某个数整除。接收端用那个数去除接收到的帧。如果余数为 0，就认为没有发生错误。如果余数不为 0，接收端将收到的消息丢弃，并通知发送端重新发送。从第 3 章你不难猜到，对于某些二进制数，可以利用移位寄存器方便地完成除法运算，这使得即便在硬件价格十分昂贵的时代，CRC 校验码也可以被广为采用。更进一步说，里德 - 索罗蒙 (Reed-Solomon) 编码使用伽罗瓦 (Galois) 域来纠正多位传输错误，数据被看作是多项式系数，校验码被看作是多项式的值。里德 - 索罗蒙计算复杂度远远高于二进制除法运算！

423

5.6 虚拟机

虚拟机 (Virtual Machine, VM) 最早出现于 20 世纪 60 年代中期，这些年来一直是大型机中的重要组成部分。尽管在 20 世纪 80 年代和 90 年代期间，它们大多被单用户计算机领域所忽略，但是最近才受到人们的关注，这是因为

- 在现代计算机系统中，隔离和安全的重要性在增长。
- 标准操作系统在安全性和可靠性方面存在缺陷。
- 在多个不相关的用户间共享一台计算机，尤其是在云计算中。

- 在过去 10 年里，处理器速度大幅增长，使得虚拟机引起的开销降至可接受的范围内。

最广泛的虚拟机的定义包括所有基本的仿真方法，这些方法提供一个标准的软件接口，如 Java 虚拟机。在这一节中，我们对虚拟机感兴趣的地方在于，在二进制指令集系统结构 (ISA) 的层次上提供一个完整的系统级环境。尽管一些虚拟机在本地硬件上运行不同的指令集系统结构，但我们假设它们都能与硬件匹配。这样的虚拟机被称为（操作）系统虚拟机 (system virtual machine)，如 IBM VM/370、VirtualBox、VMware ESX Server 以及 Xen。

系统虚拟机让用户觉得自己在使用包括操作系统的副本在内的整个计算机。一台运行多个虚拟机的计算机可以支持多个不同的操作系统。在一个传统的平台上，一个单独的操作系统拥有所有的硬件资源，但是通过使用虚拟机，多个操作系统共享硬件资源。

支持虚拟机的软件被称为虚拟机监视器 (virtual machine monitor, VMM) 或者管理程序 (hypervisor)；VMM 是虚拟机技术的核心。底层的硬件平台称为主机 (host)，它的资源被客户端 (guest) 虚拟机共享。VMM 决定如何将虚拟资源映射到物理资源：物理资源可能是分时共享、划分，甚至通过软件模拟的。VMM 比传统的操作系统小很多；一个 VMM 的隔离区可能只需要 10 000 行代码。

尽管我们所感兴趣的是虚拟机可以提供保护功能，但是在商业意义上，虚拟机还有其他两个优势：

1) 软件管理：虚拟机提供一个可以运行完整软件堆的抽象，甚至包含像 DOS 这样的旧操作系统。虚拟机典型的调度包括：一些虚拟机运行旧的操作系统，多数虚拟机运行当前的操作系统，少数虚拟机用来测试下一代操作系统版本。

2) 硬件管理：需要多个服务器的一个原因是为了让每个应用程序运行在一台单独的计算机上，并拥有与之兼容的操作系统，这样的分隔能改善可靠性。虚拟机允许这些独立的软件堆能在共享硬件的同时独立运行，因此合并了服务器的数量。另一个例子是，一些 VMM 支持将正在运行的虚拟机移植到另一台计算机上，这样可以平衡负载或在硬件故障时实施迁移。

424 **01 硬件/软件接口** 亚马逊 Web 服务 (AWS) 在其云计算平台中使用虚拟机提供 EC2 主要有 5 个原因：

- 1) 在用户共享同一个服务器时，AWS 允许提供用户间的保护。
- 2) 它简化了仓储式计算机上软件的分布。用户只需要安装一个虚拟机映象，并配置合适的软件，AWS 为用户分配其所需的所有服务。
- 3) 当用户完成工作时，用户 (和 AWS) 可以“杀死”一个 VM 来控制资源的使用。
- 4) 虚拟机隐藏了运行用户应用软件的硬件特性，这意味着 AWS 可以在继续使用老的服务器时引入新的更有效的服务器。用户希望所获得的机器性能与“EC2 计算单元”能够匹配，AWS 将其定义为：提供与 1.0–1.2 GHz 2007 AMD Opteron 或 2007 Intel Xeon 处理器相等的 CPU 能力。依据摩尔定律，很明显，新的服务器能够提供更多的 EC2 计算单元，但是出于经济型的考虑，AWS 仍然能够出租旧服务器。
- 5) 虚拟机监控器可以控制一个 VM 使用处理器、网络和磁盘的比率，这就使得 AWS 可以在相同的底层硬件上提供许多价格不同的节点类型。例如，2012 年时 AWS 提供 14 种不同类型的节点：从 0.08 美元/小时的小标准节点到 3.10 美元/小时的高 I/O 性能节点。

通常来说，处理器虚拟化的开销取决于工作量。用户级处理器限制型程序没有虚拟化开销，这是因为操作系统很少被调用，因此所有的程序都能以本来速度运行。I/O 密集型负载通常也是操作系统密集型的，它们会执行许多系统调用和特权指令，从而导致很高的虚拟化开

销。另一方面，如果 I/O 密集型负载同样也是 I/O 限制型的，由于在等待 I/O 时，处理器通常处于空闲状态，因此处理器的虚拟化开销就完全能被掩藏。

开销取决于需要由 VMM 进行模拟的指令数目以及模拟速度的快慢。因此，假设客户端虚拟机和主机运行同样的 ISA 时，系统结构和 VMM 的目标是尽可能在本地硬件上运行所有指令。425

5.6.1 虚拟机监视器的必备条件

虚拟机监视器需要做什么？它给客户软件提供了一个软件接口，分开每个客户端的状态，并且需要将自己从客户端软件中（包括客户操作系统）隔离。定性的需求是：

- 除了性能相关的行为或因多虚拟机共享而造成的固定资源限制以外，客户软件在虚拟机上的运行应该和它在本地硬件上的运行完全相同。
- 客户软件不能直接改变实际系统中的资源分配。

为了对处理器进行“虚拟化”，VMM 必须能控制一切——访问特权状态、I/O、异常和中断——尽管客户虚拟机和当前运行的操作系统临时使用它们也不受影响。

例如，在定时器中断的情况下，VMM 需要挂起当前正在运行的客户虚拟机，保存其状态，处理中断，然后决定下面该运行哪个客户虚拟机，并读取其状态。依赖定时器中断的客户虚拟机会有由 VMM 提供的一个虚拟定时器和模拟的定时器中断。

为了方便管理，VMM 必须运行在一个比用户虚拟机更高的特权级别下，其中，用户虚拟机通常运行在用户模式下，这也确保了任何特权指令的执行都需要由 VMM 来处理。系统级虚拟机的基本必备条件如下：

- 至少两个处理器模式，系统级和用户级。
- 特权级指令集合只能在系统模式下使用，如果在用户模式下执行将会产生 trap 中断；所有系统资源只能由这些指令控制。

5.6.2 指令集系统结构（缺乏）对虚拟机的支持

如果在 ISA 设计过程中考虑到了虚拟机的使用，那么由 VMM 执行的指令数目和模拟这些指令所花费的时间会相对减少些。允许虚拟机直接在硬件上执行的系统结构被冠以可虚拟化（virtualizable）的名称，IBM 370 就是如此。

由于虚拟机只是近期才考虑应用于桌面系统和基于 PC 的服务器，所以大部分指令集在创建时都没有考虑虚拟化的思想。426

x86 和大部分 RISC 系统结构，包括 ARM v7 和 MIPS 都是如此。VMM 必须保证客户系统只能和虚拟资源交互，因此常规的客户操作系统在 VMM 的顶层运行用户模式程序。如果客户操作系统试图通过特权指令访问或者修改相关硬件资源的信息——例如，读/写一个状态位来启动中断——它会向 VMM 发出 trap 中断。VMM 会进行适当的调整来对应实际资源。

因此，如果任何指令试图在用户模式下读/写这样敏感的信息 trap，VMM 将截获它并且如客户操作系统所需的那样，支持敏感信息的虚拟版本。

如果上述条件不具备，那么需要其他的方法。VMM 必须使用特殊的预防措施来定位所有存在问题的指令，并且确保它们能被客户操作系统正确执行，这就增加了 VMM 的复杂度，同时也降低了虚拟机的运行性能。

5.6.3 保护和指令集系统结构

保护需要同时依赖于体系结构和操作系统，但是随着虚拟存储器的广泛使用，体系结构设

计者需要对指令集体体系结构中一些不方便使用的细节进行修改。

例如，x86 的指令 POPF 从存储器堆栈的顶部加载标记寄存器。其中有一个标记是中断使能标记位（IE）。如果在用户模式下运行 POPF，它只是简单地改变除了 IE 位以外的所有标记位，而不是发生 trap 中断。如果在系统模式下，它确实会改变 IE 位。但是有一个问题，运行在虚拟机用户模式下的客户操作系统希望看见 IE 位的改变。

在过去，IBM 的大型机硬件和 VMM 采用以下三个步骤来改善虚拟机的性能：

- 1) 降低处理器虚拟化的开销。
- 2) 降低由虚拟化引起的中断开销。
- 3) 中断发生时交给相应的虚拟机，而不用调用 VMM，从而降低中断开销。

在 2006 年，AMD 和 Intel 提出新的计划尽力满足第一个要点，即降低处理器虚拟化的开销。体系结构和 VMM 需要经过多少代的改进才能完全满足上面三点？21 世纪的虚拟机需要经过多长时间才能像 20 世纪 70 年代的 IBM 大型机和 VMM 一样有效呢？这些都是令人感兴趣的研究。

5.7 虚拟存储器

已经设计出这样的系统：对程序员来说，复合的存储结构看起来像单层的存储器，所需的数据传输也会自动完成。

—— Kilburn 等，《One-level storage system》，1962

在前面的章节中，我们知道 cache 是如何对程序中最近访问的代码和数据提供快速访问的。427 同样，主存也可以为通常由磁盘实现的辅助存储器充当“cache”。这项技术被称作**虚拟存储器**（virtual memory）。从历史观点来说，构造虚拟存储器有两个主要动机：允许云计算在多个虚拟机之间有效而安全地共享存储器；消除一个小而受限的主存容量对程序设计造成的影响。50 年后，第一条变成主要设计动机。

② 虚拟存储器：一种将主存用作辅助存储器高速缓存的技术。

当然，为了允许多个虚拟机共享同一个存储器，我们必须在虚拟机之间进行保护，确保每个程序只能对划分给它的那部分主存进行读写操作。主存只需存放众多程序中活跃的那部分，就像 cache 中只存放一个程序的活跃部分一样。因此，局部性原理也同样适用于虚拟存储器，虚拟存储器使得我们能更有效地共享处理器和主存。

在编译的时候，我们不知道哪些虚拟机将和其他虚拟机共享存储器。事实上，当虚拟机在执行的时候，虚拟机共享存储器的情况是动态变化的。由于这种动态的相互影响，我们希望将每个程序都编译到它自己的地址空间（address space）——存储器中只能有该程序访问的独立的一连串地址。虚拟存储器实现程序地址空间到物理地址（physical address）的转换。这种地址转换处理加强了各个程序地址空间之间的保护（protection）。

② 物理地址：主存储器的地址。

② 保护：一组确保共享处理器、主存、I/O 设备的多个进程之间没有故意地、无意地读写其他进程的数据机制，这些保护机制可以将操作系统和用户的进程隔离开来。

使用虚拟存储器的第二个动机就是允许单用户程序使用超过主要存储器的容量。以前，如果一个程序对存储器来说太大，将它变成合适的大小就是程序员的责任。程序员将程序划分成许多段，并且将这些段标记成为互斥的。这些程序段（overlay）在执行过程中由用户程序控制装入或换出，由程序员保证程序不会访问没有装载的程序段，并且装载的程序段不会超过存储

器的总容量。传统的程序段被构造成模块，每一个都包含了代码和数据。不同模块之间的过程调用将导致一个模块覆盖掉另一个模块。

可以想象，这种责任对程序员来说是很大的负担。虚拟存储器的发明就是为了将程序员从这些困境中解脱出来，它自动管理由主存（为了区别虚拟存储器，有时也称为物理存储器）和辅助存储器组成的两级存储器层次结构。

尽管虚拟存储器和 cache 的工作原理是一样的，但是不同的历史根源决定它们要使用不同的术语。虚拟存储器中，块被称为页（page），访问缺失则被称为缺页（page fault）。在虚拟存储器中，处理器产生一个虚拟地址（virtual address），再结合软硬件转换成一个物理地址（physical address），然后就可以被用来访问主存了。图 5-25 显示了一个分页的虚拟寻址的存储器被映射到主存中。这个过程被称作地址映射或者地址转换（address translation）。如今，个人移动设备中由虚拟存储器控制的两级存储器层次结构是 DRAM 和闪存，而在服务器中则是 DRAM 和磁盘（见 5.2 节）。如果还拿图书馆作类比，我们可以认为一本书的书名就是虚拟地址，物理地址则是这本书在图书馆中的位置，它可能是图书馆的索书号。

428

- ① 缺页：访问的页不在主存储器中。
- ② 虚拟地址：虚拟空间的地址，当需要访问主存时需要通过地址映射转换为物理地址。
- ③ 地址转换：也称为地址映射。在访问内存时将虚拟地址映射为物理地址的过程。

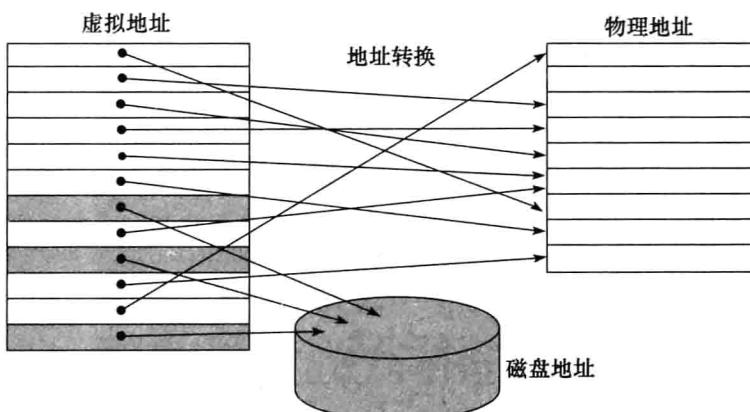


图 5-25 在存储器中，主存中的块（称为页）从一组地址（称为虚拟地址）映射到另一组地址（称为物理地址）。访问主存使用物理地址，而处理器产生虚拟地址。虚拟存储器和物理存储器都被划分成页，因此一个虚页被映射到一个物理页。当然，一个虚页也可能不在主存中，因此无法映射到物理地址；在这种情况下，页就映射到磁盘上。物理页可以被两个指向相同物理地址的虚拟地址共享。这种方法用来使两个不同的程序共享数据或代码。

虚拟存储器还提供重定位（relocation）来简化执行时的程序加载过程。在用地址访存之前，重定位将程序所用的虚拟地址映射到不同的物理地址。重定位的方法允许我们将程序加载到主存中的任何位置。另外，现今所有的虚拟存储器系统将程序重定位成一组固定大小的块（页），因此减少了寻找主存中连续的块来放置程序的必要；取而代之的是，操作系统只需要在主存中找到足够数量的页。

在虚拟存储器中，地址被划分为虚页号（virtual page number）和页偏移（page offset）。图 5-26 所示是虚页号到物理页号（physical page number）的转换。物理页号构成物理地址的高位部分，而页偏移是不变的，构成物理地址的低位部分。页偏移域的位数决定了页的大小。虚拟地址可寻址的页数与物理地址可寻址的页数可以不同。拥有比物理页数多得多的虚页数是描述一个没有容量限制的虚拟存储器的假象的基础。

429

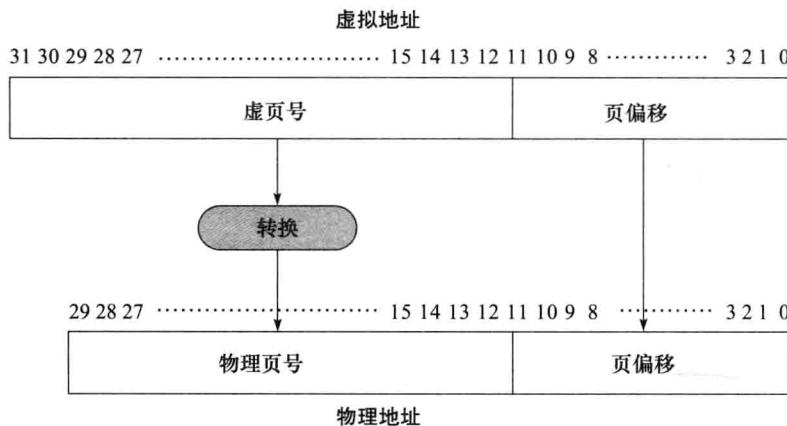


图 5-26 虚拟地址到物理地址的映射。页大小为 $2^{12} = 4\text{KiB}$ 。由于物理页号有 18 位，存储器中物理页数为 2^{18} 。因此，最多可以支持 1GiB 的主存，而虚拟地址空间为 4GiB

页缺失引发的高代价是许多设计选择虚拟存储系统的原因，缺失在虚拟存储器中通常称为缺页。一次缺页处理将花费数百万个时钟周期（5.2 节的表指出了主存储器大概比磁盘快 100 000 倍）。这一巨大的缺失代价，主要由取得标准大小的页中第一个字所需的时间来确定，因此在设计虚拟存储系统时需要考虑一些关键性的因素：

- 为了弥补较长的访问时间，页应该足够大。目前典型的页大小从 $4 \sim 16\text{KiB}$ 。能支持 $32 \sim 64\text{KiB}$ 页的新型台式计算机和服务器正进行研发，但是新的嵌入式系统走的是相反的方向，页大小为 1KiB 。
- 能降低缺页率的组织结构具有吸引力。这里用到的主要技术是允许存储器中的页以全相联方式放置。
- 缺页可以用软件处理，这是因为与访问磁盘的时间相比，这样的开销不算大。此外，软件可以使用一些更先进的算法来选择替换页，只要缺失率减少很小一部分就足以弥补算法的开销。
- 由于写时间太长，因此在虚拟存储器中，写直达机制不能很好地管理写操作。因此虚拟存储系统中都采用写回机制。

下面几节将把这些因素融入虚拟存储器的设计中去。

01 精解 我们引入虚拟存储器是由于许多虚拟机器共享存储器，但是，虚拟存储器发明的最初原因是在分时系统中，许多程序可以共享一台计算机。由于当今的许多读者没有使用分时系统的经验，本节使用虚拟机作为引入共享存储器的动机。

01 精解 对台式机和服务器来说，32 位地址的处理器已经很有问题了。通常我们认为虚拟地址要远大于物理地址，但是如果相对于存储技术，处理器地址字较小的时候，相反的情况也会出现。单个程序或虚拟机不会受益，但是一组程序或虚拟机同时执行就可能因无需交换到主存，或者在并行处理器上执行而受益。

01 精解 本书对虚拟存储器的讨论主要集中于使用固定大小的块的页式虚拟存储。还有一种可变长度块的机制称为段式管理（segmentation）。在段式存储中，地址由两部分组成：段号和段内偏移。段号被映射到物理地址，然后与段内偏移量相加来找到实际物理地址。因为段大小是可变的，所以还需要进行边界检查以确定偏移量是否在段内。分段最主要的应用就是支持更多有效的保护方法，以及共享地址空间。与分页相比，大多数操作系统的教科书都会更多地讨论分段，以及如何利用分段来逻辑共享地址空间。分段的主要缺点在于

它将地址空间划分为许多逻辑上独立的块，因而这些块就由两部分地址控制：段号和段内偏移。相反，分页使得页号和偏移量的界限对于程序员和编译器都是不可见的。

分段也曾被用作不改变计算机字的大小而扩展地址空间的方法。然而这些尝试都没有获得成功，这是由于程序员和编译器必须意识到使用两部分地址本身的不便和性能代价。

许多体系结构将地址空间划分成固定大小的大块以简化操作系统和用户程序之间的保护，同时提高分页实现的效率。尽管这些划分通常称为“段”，但是这种结构比块大小可变的分段要简单得多，并且对用户程序不可见。稍后我们对此进行详细讨论。

- ② 段式管理：一种可变长度的地址映射策略，其中每个地址由两部分组成：映射到物理地址的段号和段内偏移。

5.7.1 页的存放和查找

由于缺页的代价高得惊人，设计人员通过对页的放置进行优化从而降低缺页频率。如果允许一个虚拟页映射到任何一个物理页，那么当缺页发生时，操作系统可以选择任意一个页进行替换。例如，操作系统可以使用复杂的算法和复杂的数据结构来跟踪页的使用情况以选择在较长一段时间内不会被用到的页。使用更先进更灵活的替换策略降低了缺页率，也使全相联方式下页的放置变得更简单。

正如在 5.4 节中提到的，全相联映射的困难在于项的定位，这是由于它可能在较高的存储器层次中的任何位置。全部进行检索是不切实际的。在虚拟存储系统中，我们使用一个索引存储器的表来定位页；这种结构称为页表（page table），它被存放在存储器中。页表使用虚拟地址中的页号作索引，以找到相应的物理页号。每个程序都有它自己的页表，用来将程序的虚拟地址空间映射到主存中。让我们再用图书馆进行类比，页表对应于书名和藏书位置之间的映射。就像卡片目录可能会包含学校中另一个图书馆中书的条目，而不仅仅是本地的分馆，我们将看见页表也可能含有不在存储器中的页的条目。为了指出页表在存储器中的位置，硬件包含一个指向页表首地址的寄存器；我们称之为页表寄存器（page table register）。现在假定页表存在存储器中一个固定的连续区域内。

- ② 页表：保存着虚拟地址和物理地址之间转换关系的表。页表保存在主存中，通常使用虚页号来索引，如果这个虚页当前在主存中，页表中的对应项将包含虚页对应的物理页号。

01 硬件/软件接口 页表、程序计数器以及寄存器，确定了一个虚拟机的状态（state）。如果我们想让另一个虚拟机使用处理器，我们必须保存该状态。随后，在恢复了该状态之后，虚拟机就可以继续执行。我们通常称该状态为一个进程（process）。如果一个进程占据了处理器，那么这个进程就是活跃的（active），否则就认为它是非活跃的（inactive）。操作系统可以通过加载进程的状态令一个进程活跃起来，同时激活程序计数器，进程将会在程序计数器中保存的值处开始执行。

进程的地址空间，以及它在主存中可以访问的所有数据，都由驻在主存中的页表所定义。操作系统只是简单地加载页表寄存器用来指向它想激活的进程的页表，而不是保存整个页表。由于不同进程使用相同的虚拟地址，因此每个进程有各自的页表。操作系统负责分配物理主存和更新页表，因此不同进程的虚拟地址空间不会发生冲突。我们很快会看到，使用分离的页表同样能分别保护进程。

431

432

图 5-27 使用页表寄存器、虚拟地址以及被指向的页表来说明硬件是如何形成物理地址的。每个页表项使用 1 位有效位，就像在 cache 中设计的一样。如果该位为无效，该页就不在主存中，就发生一次缺页。如果该位为有效，表明该页在主存中，并且该项包含有物理页号。

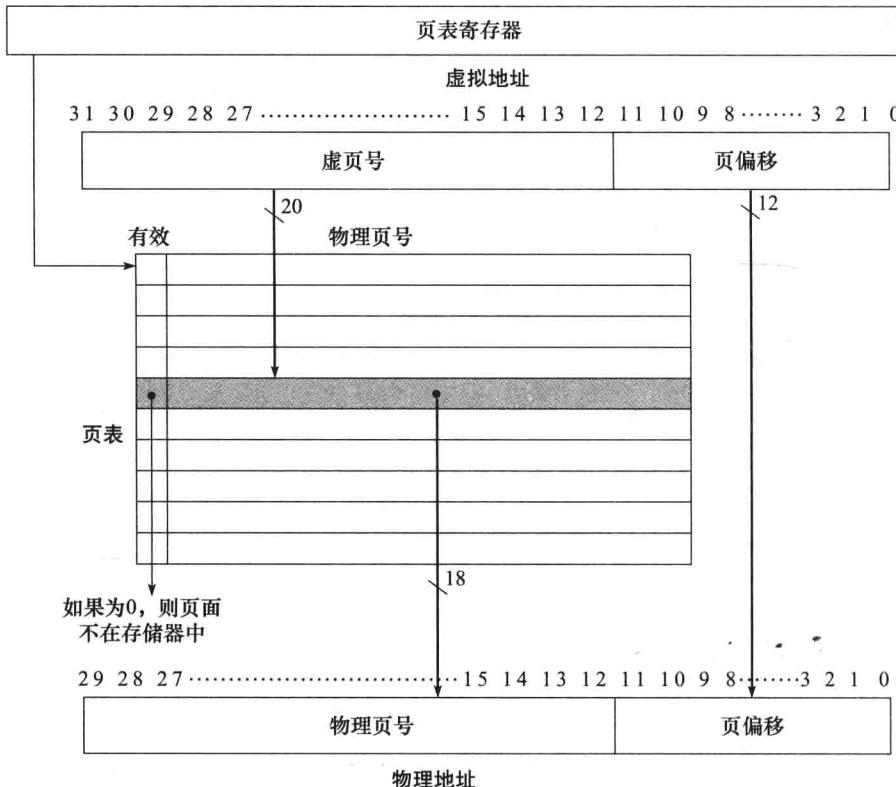


图 5-27 用虚页号来索引页表以获得对应的物理地址部分。假定地址为 32 位。页表的首地址由页表指针给出。在本图中，页大小为 2^{12} 字节，即 4KiB，虚拟地址空间为 2^{32} 字节，即 4GiB，物理地址空间为 2^{30} 字节，可以支持高达 1GiB 的主存。页表项数为 2^{20} ，即 100 万项。每一项的有效位指出了映射是否合法。如果该位为 0，那么该页就不在主存中。尽管图中所示的页表项宽度只需 19 位，但为了寻址方便，通常让它有 32 位。其他位则用来存放每页都要保留的基本的附加信息，如保护信息

433

由于页表包含了每个可能的虚拟页的映射，因此不需要标记位。在 cache 术语中，索引是用来访问页表的，由整个块地址即虚页号组成。

5.7.2 缺页故障

如果虚拟页的有效位关闭，就会发生缺页故障。操作系统获得控制权。控制的转移由异常机制完成，这点在第 4 章已经看到并在本节稍后进行讨论。一旦操作系统获得控制权，它必须在下一级存储器层次（通常是闪存或磁盘）中找到该页，然后决定将其放到主存中。

虚拟地址本身并不会马上告诉我们页在磁盘中的位置。还拿图书馆作类比，我们不能仅仅依靠书名就找到图书的具体位置。而是按目录查找，获得书在书架上的位置信息，比如说图书馆的索引书号。同样，在虚拟存储系统中，我们必须保持跟踪记录虚拟地址空间的每一页在磁盘上的位置。

由于我们无法提前获知存储器中的某一页什么时候将被替换出去，因此操作系统在创建进程的时候通常会在闪存或磁盘上为进程中所有的页创建空间。这一磁盘空间称为交换区（swap

space)。同时，它也会创建一个数据结构来记录每个虚拟页在磁盘上的存放位置。这个数据结构可能是页表中的一部分，也可能是辅助数据结构，寻址方式和页表一样。图 5-28 是一个包含物理页号或磁盘地址的单个表的结构。

◎ 交换区：为进程的全部虚拟地址空间所预留的磁盘空间。

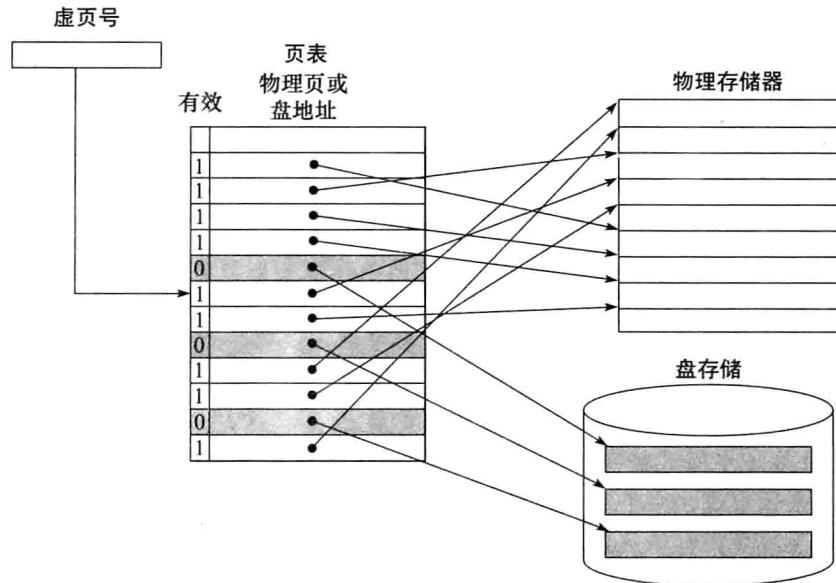


图 5-28 页表将虚拟存储器中的每一页映射到主存中的一页或者存储结构的下一层（磁盘上的一页）。虚页号用来检索页表。如果有效位开启，页表提供虚页对应的物理页号（如存储器中该页的首地址）。如果有效位关闭，那么该页就只存在磁盘上的某个指定的磁盘地址。在许多系统中，物理页地址和磁盘页地址的表在逻辑上是一个表，但是保存在两个独立的数据结构中。因为即使有些页当前不在主存中，我们也必须保存所有页的磁盘地址，所以使用双表在某种程度上是合理的。请记住主存中的页和磁盘上的页大小相等

操作系统同样会创建一个数据结构来跟踪记录使用每个物理页的是哪些进程和哪些虚拟地址。当一次缺页发生时，如果主存中所有的页都在使用，操作系统仍必须选择一页进行替换。我们希望最小化缺页的次数，因而大多数操作系统都会选择它们认为近期内不会被使用的页进行替换。使用过去的信息来预测未来的使用情况，操作系统遵循我们在 5.4 节中提到的最近最少使用 (LRU) 替换策略。操作系统查找最近最少使用的页，假定某一页在很长一段时间都没有被访问，那么该页再被访问的可能性比最近经常访问的页的可能性要小。被替换的页写入磁盘的交换区。如果还不是很明白，可以把操作系统看成是另一个进程，而那些控制主存的表也在主存中；这看起来似乎有些矛盾，稍后将具体解释。

434

01 硬件/软件接口 要完全准确地执行 LRU 算法的代价太高了，因为每次存储器访问时都需要更新数据结构。作为替代，大多数操作系统通过跟踪哪些页最近被使用，哪些页最近没有用到来近似地实现 LRU 算法。为了帮助操作系统估算最近最少使用的页，一些计算机提供了一个引用位 (reference bit) 或者称为使用位 (use bit)，当一页被访问时该位被置位。操作系统定期将引用位清零，然后再重新记录，这样就可以判定在这段特定时间内哪些页被访问过。有了这些使用信息，操作系统就可以从那些最近最少访问的页中选择一页（通过检查其引用位是否关闭）。如果硬件没有提供这一位，操作系统就要通过其他的方法来估计哪些页被访问过。

引用位：也称为使用位。每当访问一个页面时该位被置位，通常用来实现 LRU 或其他替换策略。

01 精解 虚拟地址为 32 位，页大小为 4KiB，页表每一项为 4 字节，我们可以计算总的页表容量为：

$$\text{页表项数} = 2^{32}/2^{12} = 2^{20}$$

$$\text{页表容量} = 2^{20} \text{ 个页表项} \times 2^2 \text{ 字节/页表项} = 4\text{MiB}$$

也就是说，每个程序在执行的任何时候都需要 4MiB 的存储器空间。对单个进程来说，这个大小并不差。但是如果计算机中同时有成百上千的进程同时运行时，每一个程序有各自的页表，这将会怎样？我们又如何处理 64 位地址，通过这个计算需要 2^{52} 个字？

一系列的技术已经被用于减少页表所需的存储量。下面 5 种技术都是针对减少所需的最大存储量以及减少用于页表的主存：

1) 最简单的技术是使用一个界限寄存器，对给定的进程限制其页表的大小。如果虚拟页号大于界限寄存器中的值，就必须在页表中加入该项。这种技术允许页表随着进程消耗空间的增多而增长。因此，只有当进程使用了虚拟地址空间的许多页时，页表才会变得很大。这种技术要求地址空间只朝一个方向扩展。

2) 允许地址空间只朝一个方向增长并不够，因为多数语言需要两种大小可扩展的区域：一个用来保留栈，一个用来保留堆。由于这种二元性，如果将页表划分，使其既能从最高地址向下扩展，也能从最低地址向上扩展，就方便多了。这也意味着有两个独立的页表和两个独立的界限。两个页表的使用将地址空间分成两段。地址的高位用来判断该地址使用了哪个段和哪个页表。由于段由地址的高位部分决定，每一段可以有地址空间的一半大。每段的界限寄存器指定了当前段的大小，该大小以页为单位增长。这种类型的段被应用于很多体系结构，包括 MIPS 结构。不同于 5.7 节的第三个精解中讨论的段，这种形式的段对应用程序是不可见的，尽管它对操作系统可见。这种机制主要的缺陷在于当以一种稀疏方式使用地址空间而不是一组连续的虚拟地址时，它的执行效果就不太好。

3) 另外一种减小页表容量的方法是对虚拟地址使用哈希函数，这样，页表需要的容量仅仅是主存中的物理页数。这种结构称为反置页表 (inverted page table)。当然，反置页表的查找过程略微有些复杂，因为我们不能仅仅依靠索引来访问页表。

4) 多级页表同样可以用来减少页表存储量。第一级映射到虚拟地址空间中较大的固定大小的块，一共有 64 ~ 256 页。这些大的块有时候称为段，而第一级的映射表有时称为段表，对用户来说段表是不可见的。段表中的每一项指出了该段中是否有页被分配，如果有，就指向该段的页表。地址转换发生在第一次段表查找时，使用地址的高位部分。如果段地址有效，下一组高位地址则用来索引由段表项指向的页表。这种机制允许以一种稀疏的方式（多个不相连的段同时处于活跃状态）来使用地址空间而不用分配整个页表。对很大的地址空间和在需要非连续地址分配的软件系统中，这种机制尤为有效。但是这种两级映射方式的主要缺陷在于地址转换过程更为复杂。

5) 为了减少页表占用的实际主存空间，现在，多数系统也允许将页表再分页。尽管听起来这很复杂，但是它的工作原理和虚拟存储器一样，并且允许页表驻留在虚拟地址空间中。另外，还有一些很小却很关键的问题，例如，要避免不断出现的缺页。如何克服这些问题需要描述地很细节，并且一般对机器的依赖性很高。简而言之，要避免这些问题，可以将全部页表置于操作系统地址空间中，并且至少要把操作系统中一部分页表放在主存中的可物理寻址的一块区域中，这部分页表总是存在于主存而非磁盘中。

5.7.3 关于写

访问 cache 和主存的时间相差上百个时钟周期，写直达机制也可以使用，但是我们需要一个写缓冲区来隐藏写延迟。在虚拟存储器系统中，对存储器层次结构中下一层（磁盘）的写操作需要数百万个处理器时钟周期；因此，创建一个缓冲区用来允许系统用写直达的方式对磁盘进行写的方法是完全不可行的。相反，虚拟存储器系统必须使用写回机制，对存储器中的页进行单独的写操作，并且在该页被替换出存储器时再被复制到磁盘中去。

01 硬件/软件接口 在虚拟存储系统中，写回机制有另一个主要的优点。因为相对于磁盘访问时间，其传输时间要少得多，因此，把整页复制回磁盘比把单个字写回要高效得多。尽管写回操作比传输单个字更高效，但是开销却很大。因此，当某一页被替换时，我们希望知道该页是否需要被复制写回。为了追踪读入主存中的页是否被写过，可以在页表中增加一个脏位（dirty bit）。当页中任何字被写时就将这一位置位。如果操作系统选择替换某一页，脏位指明了在把该页所占用的主存让给另一页之前，是否需要将该页写回磁盘。因此，一个修改过的页也通常被称为脏页（dirty page）。

437

5.7.4 加快地址转换：TLB

由于页表存放在主存中，因此程序每次访存至少需要两次：第一次访存先获得物理地址，第二次访存才获得数据。提高访问性能的关键在于依靠页表的访问局部性。当一个转换的虚页号被使用时，它可能在不久的将来再次被用到，因为对该页中字的引用同时具有时间局部性和空间局部性。

因此，现代处理器都包含一个特殊的 cache 以跟踪最近使用过的地址变换。这个特殊的地址转换 cache 通常称为快表（Translation-Lookaside Buffer，TLB）（将其称为地址变换高速缓存更精确）。TLB 就相当于记录目录中的一些书的位置的小纸片；我们在纸片上记录一些书的位置，并且将小纸片当成图书馆索书号的 cache，这样就不用一直在整个目录中搜索了。

快表：用于记录最近使用地址的映射信息的高速缓存，从而可以避免每次都要访问页表。

如图 5-29 所示，TLB 的每个标记项存放虚页号的一部分，每个数据项中存放了物理页号。由于我们每次访问的是 TLB 而不是页表，TLB 需要包括其他状态位，如脏位和引用位。

每次访问，我们都要在 TLB 中查找虚页号。如果命中，物理页号就用来形成地址，相应的引用位被置位。如果处理器执行的是写操作，脏位同样要被置位。如果 TLB 发生缺失，我们必须判断是发生缺页还是仅仅是一次 TLB 缺失。如果该页在主存中，那么 TLB 缺失只是一次转换缺失。在这种情况下，处理器可以通过将页表中的变换装载到 TLB 中并且重新访问来进行缺失处理。如果该页不在主存中，TLB 缺失就是一次真的缺页。在这种情况下，处理器调用操作系统的异常处理。由于 TLB 中的项比主存中的页数少得多，发生 TLB 缺失会比缺页频繁得多。

TLB 缺失既可以通过硬件处理，也可以通过软件处理。实际上，两种方法的性能差别很小，这是因为无论哪种方法，需要执行的基本操作都是一样的。

在发生了 TLB 缺失，并且已经在页表中找到了缺失的变化时，我们就需要从 TLB 中选择一项进行替换。由于 TLB 表项中包含了引用位和脏位，当替换某一项时，需要把这些位复制回页表项中。这些位是 TLB 表项中唯一可以修改的部分。利用写回策略——只是在缺失的时候将这些表项写回而不是任何写操作都写回——是非常有效的，因为我们期望 TLB

缺失率更低。一些系统使用其他技术来近似引用位和脏位，以消除除了缺失后装入新表项之外写 TLB 的必要。

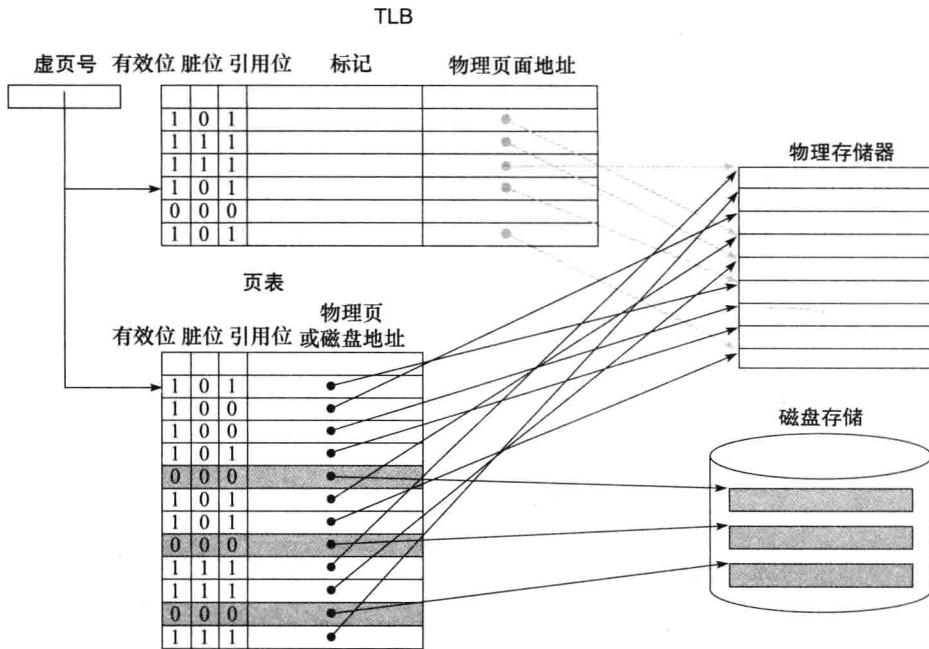


图 5-29 TLB 作为页表的 cache，用于存放在映射到物理页中的那些项。TLB 包含了页表中虚页到物理页映射的一个子集。TLB 映射以灰线显示。因为 TLB 是一个 cache，它必须有标记域。如果一个页在 TLB 中没有匹配的项，就必须检查页表。页表或者提供该页的物理页号（可用来创建一个 TLB 项），或者指出该页在磁盘上，这时就会发生缺页。由于页表对于每个虚页都有一个相应的项，并不需要标记；换句话说，不同于 TLB，页表并不是 cache

438

TLB 的一些典型的值为：

- TLB 大小：16 ~ 512 个项。
- 块大小：1 ~ 2 个页表项（通常每个为 4 ~ 8 字节）。
- 命中时间：0.5 ~ 1 个时钟周期。
- 缺失代价：10 ~ 100 个时钟周期。
- 缺失率：0.01% ~ 1%。

439

设计者在 TLB 设计中相联度的设置非常多样化。有些系统使用小的全相联的 TLB，这是由于全相联有较低的缺失率；此外，由于 TLB 很小，全相联映射的成本也不会太高。其他一些系统通常使用相联度低且容量大的 TLB。在全相联映射的方式下，由于用硬件实现 LRU 策略的代价很大，因此替换项的选择就很复杂。另外，由于 TLB 的缺失比缺页要频繁得多，因此需要用较低的代价来处理缺失，而不能像缺页处理那样选择一个开销大的软件算法。所以很多系统都支持随机地选择替换表项的方法。在 5.8 节中我们将会详细讨论替换策略。

内置 FastMATH TLB

为了弄清楚这些想法是如何实际应用到处理器中的，我们来进一步研究内置 FastMATH 的 TLB。存储系统页大小为 4KiB，地址空间为 32 位，因此，虚页号长为 20 位，如图 5-30 顶部所示。物理地址和虚拟地址长度相等。TLB 包含了 16 个项，采用全相联映射，由指令和数据访问共享。每个表项宽为 64 位，包含了 20 位的标记位（作为该 TLB 表项的虚页号）、相应的物

理页号（也是 20 位）、一个有效位、一个脏位以及一些其他管理操作位。与大多数 MIPS 系统类似，它采用软件来处理 TLB 缺失。

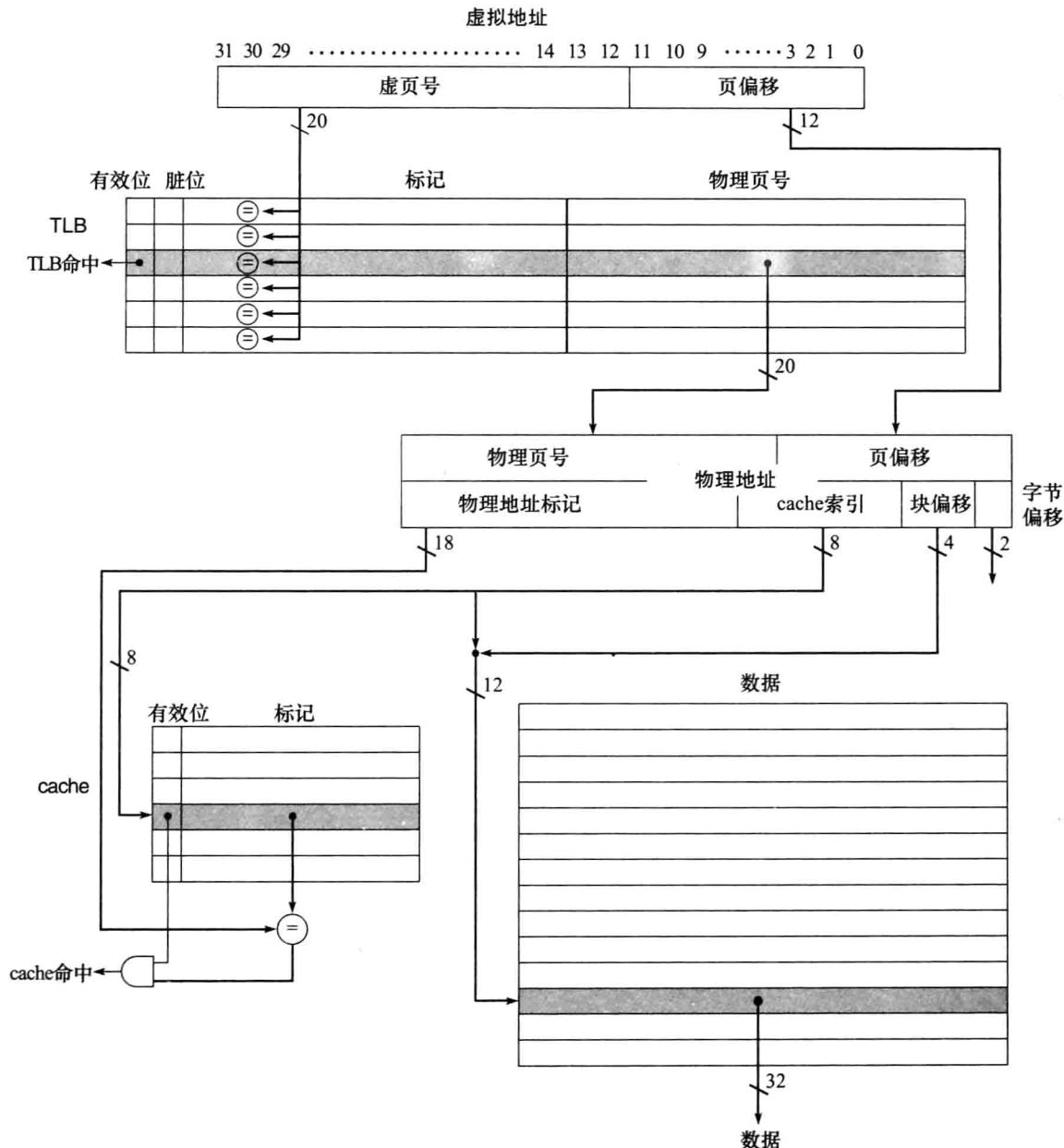


图 5-30 内置 FastMATH 中 TLB 和 cache 实现了从虚拟地址到数据项的转换过程。本图描述了 TLB 和数据 cache 的结构，这里假设页大小是 4KiB。本图主要研究读操作，图 5-31 则描述了如何处理写操作。注意到不同于图 5-12，标记和数据 RAM 是分开的。用 cache 索引和块偏移来寻址长而窄的数据 RAM，无需使用 16:1 的多路选择器我们也能选出块中所需的字。当 cache 采用直接映射方式时，TLB 是全相联的。由于需要的项可能在 TLB 中的任何位置，因此要实现全相联 TLB 需要将每个 TLB 标记都与虚页号进行比较（参考 5.4.2 节精解的内容）。如果匹配表项的有效位有效，那么 TLB 访问命中，物理页号与页偏移中的位共同形成访问 cache 的索引

图 5-30 是 TLB 和一个 cache，图 5-31 则说明了处理一次读或写请求的步骤。当一次 TLB 缺失发生时，MIPS 硬件把被访问的页号保存在一个特殊寄存器中，并产生一次异常。异常请求操作系统通过软件处理缺失。为了找到缺失的页的物理地址，TLB 缺失程序用虚拟地址的页

号，以及能指出活跃进程页表起始地址的页表寄存器来检索页表。通过使用一系列更新 TLB 的特殊指令，操作系统将页表中的物理地址放入 TLB 中。假设代码和页表项都在指令 cache 和数据 cache 中，那么一次 TLB 缺失大概需要花费 13 个时钟周期（在 5.7.7 节，我们将讨论 MIPS TLB 代码）。如果页表项中没有有效的物理地址，就会发生一次真的缺页。硬件保存着被建议替换项的索引，而这一项则是随机选取的。

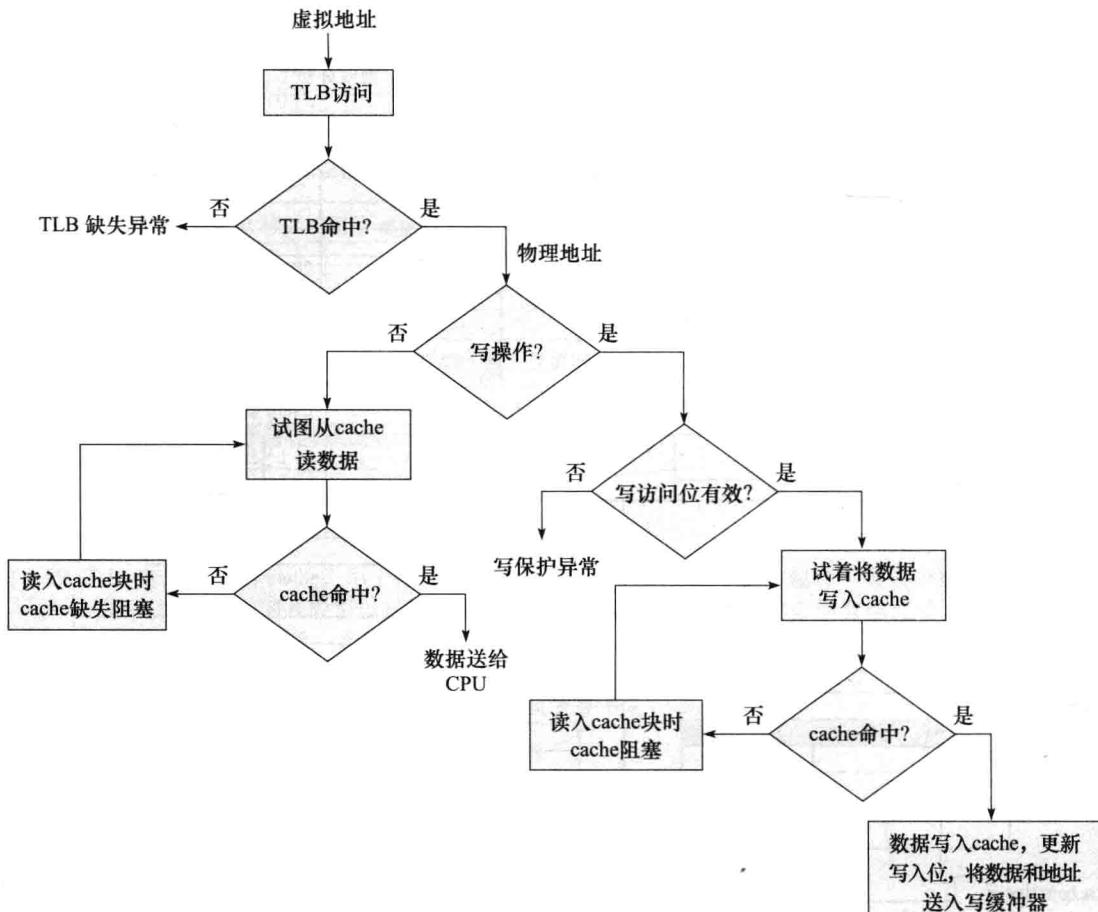


图 5-31 在内置 FastMATH 的 TLB 和 cache 中处理读或者写直达操作。如果 TLB 命中，最终的物理地址就可以用来访问 cache。对于读操作，当从存储器中取数据时，cache 产生命中或缺失，提供数据或者引起阻塞。对于写操作，若命中，cache 某数据项中的一部分内容将被重写，如果采用写直达策略还要将数据送到写缓冲区中。写缺失和读缺失相同，只是在数据块从存储器中读出后会被修改。写回策略需要将 cache 的写入位置位，并且只有当读或写缺失时，如果被替换的块处于修改状态，才将整块写入写缓冲。注意，TLB 命中和 cache 命中是相互独立的事件，但是 cache 命中只可能发生在 TLB 命中之后，这就意味着数据必须在主存中。TLB 缺失和 cache 缺失之间的联系将在接下来的例子和本章最后的习题中进一步研究

对于写请求来说，有一个额外的复杂情况：必须检查 TLB 中的写访问位。该位可以阻止程序向它仅具有读权限的页中进行写操作。如果程序试图写，且写访问位是关闭的，则会产生异常。写访问位构成了保护机制的一部分，我们将在稍后讨论。

5.7.5 集成虚拟存储器、TLB 和 cache

虚拟存储器和 cache 系统就像一个层次结构一样共同工作，因此除非数据在主存中，否则

它不可能在 cache 中出现。操作系统帮助管理该层次结构，当它决定将某一页移到磁盘上去时，就在 cache 中将该页中的内容刷新。同时，操作系统修改页表和 TLB，而后尝试访问该页上的数据都将发生缺页。

在最好的情况下，虚拟地址由 TLB 进行转换，然后被送到 cache，找到相应的数据，取回并送入处理器。在最坏的情况下，访问在存储器层次结构的 TLB、页表和 cache 这三个部件中都发生缺失。下面的例子将详细介绍这些交互作用。

440

l

442

01 例题·存储器层次结构的全部操作

存储器层次结构类似于图 5-30，由一个 TLB 和一个 cache 组成。一次存储器访问可能遇到三种不同类型的缺失：TLB 缺失、缺页以及 cache 缺失。考虑这三种缺失发生一个或多个时所有可能的组合（7 种可能性）。对每种可能性，说明这种情况是否会真的发生，在什么条件下发生。

01 答案

图 5-32 说明了所有可能发生的组合以及事实上它们是否真的可能发生。

TLB	页表	cache	这种情况可能发生么？如果可能，在什么情况下发生？
命中	命中	缺失	可能，但若 TLB 命中就不可能检查页表
缺失	命中	命中	TLB 缺失，但在页表中找到表项；重试后在 cache 中找到数据
缺失	命中	缺失	TLB 缺失，但在页表中找到表项；重试后在 cache 中未找到数据
缺失	缺失	缺失	TLB 缺失，随后发生缺页；重试后在 cache 中必找不到数据
命中	缺失	缺失	不可能：如果页不在主存中，TLB 中没有此转换
命中	缺失	命中	不可能：如果页不在主存中，TLB 中没有此转换
缺失	缺失	命中	不可能：如果页不在主存中，数据不允许在 cache 中存在

图 5-32 在 TLB、虚拟存储器系统以及 cache 中可能发生的事件组合。在这些组合中，有三种是不可能的，有一种是可能的但是永远不可能检测到（TLB 命中，虚拟存储器命中，cache 缺失） □

01 精解 图 5-32 假定在访问 cache 之前，所有存储器地址都被转换成物理地址。在这种结构中，cache 是按物理地址索引（physically indexed）并且是物理标记（physically tagged）的（所有 cache 的索引和标记都用物理地址，而不是虚拟地址）。在这个系统中，假定 cache 命中，那么访问主存的时间要包括对 TLB 访问和 cache 访问的时间，当然，这些访问可以流水地执行。

另外，处理器可以用一个完整的或者部分虚拟的地址来索引 cache。这称为虚拟寻址 cache（virtually addressed cache），它使用虚拟地址作为标记；因此这种 cache 是按虚拟地址索引（virtually indexed）并且是虚拟标记（virtually tagged）的。在这种 cache 中，地址转换硬件（TLB）在正常的 cache 访问过程中没有被用到，这是因为使用的是没有被转换成物理地址的虚拟地址来访问 cache 的。这样就把 TLB 排除在关键路径之外，减少了 cache 延时。当 cache 访问缺失时，处理器需要将该地址转换成物理地址以便从主存中取出 cache 块。

当使用虚拟地址访问 cache，并且进程之间共享页（进程可能使用不同的虚拟地址访问页）时，就可能有别名（aliasing）存在。当同一个对象有两个名字时就会产生别名——在这种情况下，两个虚拟地址对应于同一个页。这种多义性就会产生一个问题，由于页上的一个字可能存在与 cache 中的两个不同位置，每个位置对应不同的虚拟地址。这就会出现一个程序写数据，而另一个程序并不知道数据已经改变的情况。完全虚拟寻址的 cache 或者对 cache 和 TLB 的设计进行限制以减少别名，或者需要操作系统也可以是让用户来采取措施以保证别名不会发生。

这两种设计观点常用的方法是采用虚拟地址索引的 cache——有时仅仅使用地

址的页偏移部分，由于没有被转换，因此实际上是物理地址——但使用物理标记。这些采用虚拟索引和物理标记的设计，试图同时拥有虚拟地址索引 cache 的优越性能以及物理寻址 cache (physical addressed cache) 的简单结构。例如，在这种情况下就没有别名的问题。图 5-30 假定的页大小为 4KiB，但实际上有 16KiB，因此内置 FastMATH 就使用了这种方法。要实现这种方法，必须在最小页大小、cache 大小以及相联度之间进行谨慎的权衡。

- 443
- ② 虚拟寻址 cache：一种使用虚拟地址而不是物理地址访问的 cache。
 - ② 别名：使用两个地址访问同一个目标的情形，一般发生在虚拟存储器中两个虚拟地址对应到同一个物理页时。
 - ② 物理寻址 cache：使用物理地址寻址的 cache。

5.7.6 虚拟存储器中的保护

虚拟存储器最重要的功能就是允许多个进程共享一个主存，同时为这些进程和操作系统提供存储保护。保护机制必须确保：尽管多个进程在共享同一个主存，但是无论有意或是无意，一个恶意进程不能写另一个用户进程或者操作系统的地址空间。TLB 中的写访问位可以防止一个页被改写。如果没有这一级保护，计算机病毒将更加泛滥。

- 01 硬件/软件接口 为了使操作系统能保护虚拟存储系统，硬件至少提供下面总结的三种基本能力。注意，由于前两者都需要虚拟机，因此其需求相同。

1) 支持至少两种模式，并指出当前运行的进程是用户进程还是操作系统进程，操作系统进程也称为超级用户管理 (supervisor) 进程、核心进程或者主管进程。

2) 提供一部分处理器的状态，这部分内容是用户进程可读而不可写的。这包括指示处理器是处于用户态还是管理态的用户/管理模式位、页表指针以及 TLB。操作系统可以通过使用只能在管理态下可用的特殊指令对它们进行写操作。

3) 提供能让处理器在用户态和管理态下相互切换的机制。从用户态到管理态的转换通常是由系统调用 (system call) 异常处理完成的，它用特殊指令（如 MIPS 指令集中的 syscall）将控制权传到管理代码空间的指定位置。和其他异常处理一样，系统调用处的程序计数器中的值被保存在异常程序计数器中 (EPC)，处理器被置于管理态。从异常中返回至用户模式，使用异常返回 (return from exception) 指令，将重置用户模式，并且跳转到 EPC 中的地址处。

通过使用这些机制并且把页表保存在操作系统的地址空间中，操作系统可以更改页表，并且阻止用户进程改变它们，确保用户进程只能访问由操作系统提供给它的存储部分。

- ② 超级用户管理模式：也称作核心模式 (kernel mode)。运行操作系统进程的模式。
- ② 系统调用：将控制权从用户模式转换到管理模式的特殊指令，触发进程中的一个异常机制。

我们同样要防止一个进程读取另一个进程的数据。例如，当成绩放在处理器的主存中，我们不希望学生程序读到它们。一旦我们开始共享主存，必须赋予进程保护数据防止被其他进程读或写的能力；否则，共享主存将变得乱七八糟。

每个进程有它自己的虚拟地址空间。因此，如果操作系统管理页表的组织，使独立的虚拟

页映射到不相交（disjoint）的物理页上，就能使得一个进程无法访问另一个进程的数据了。当然，这也要求一个用户进程不能改变页表的映射。如果操作系统能防止用户进程更改自己的页表，那么安全性也就有了保证。然而，这样一来，操作系统必须负责修改页表。将页表放在操作系统的保护地址空间就能满足所有要求。

当进程希望以受限的方式共享信息时，操作系统必须协助它们，这是因为访问另一个进程的信息需要改变访问进程的页表。写访问位可以用来把共享限制为只读，并且和页表中其他位一样，该位只能被操作系统修改。为了允许另一个进程，设为 P1，去读属于进程 P2 的一页，P2 就要请求操作系统在 P1 地址空间中为一个虚拟页生成页表项，指向 P2 想要共享的物理页。如果 P2 要求，操作系统可以使用写保护位以防止 P1 对数据进行改写。由于只有 TLB 缺失才会访问页表，任何决定页的访问权限的位不仅要包含在页表中，还要包含在 TLB 中。

445

01 精解 当操作系统决定从执行进程 P1 切换到执行进程 P2（称为上下文切换（context switch），或者进程切换）时，它必须保证 P2 不能访问 P1 的页表，否则不利于数据保护。

如果没有 TLB，只要把页表寄存器转而指向 P2 的页表（而不是 P1 的）就足够了；如果有 TLB，我们必须在其中清除属于 P1 的表项——不仅是为了保护 P1 的数据，而且是为了迫使 TLB 装入 P2 的表项。如果进程切换的频率很高，这一举措的效率就很低。例如，在操作系统切换回 P1 之前，P2 可能只装入了很少的 TLB 表项。不幸的是，P1 随后发现它所有的表项都不见了，因此不得不通过 TLB 缺失来重新加载这些表项。产生这个问题是因为 P1 和 P2 使用同一个虚拟地址空间，并且我们必须清除 TLB 以防止地址混淆。

另一种常用的方法则是通过增加进程标识符（process identifier）和任务标识符（task identifier）来扩展虚拟地址空间。为此内置 FastMATH 有 8 位地址空间标识域（ASID）。就是这个域标识了当前正在运行的进程；当进程切换时，它保存在由操作系统装入的寄存器中。进程标识符与 TLB 的标记部分相连接，因此只有在页号和进程标识符同时匹配时，TLB 才会发命中。这样的话，除非特殊情况，否则我们就不需要清除 TLB。

同样的问题可能在 cache 中发生，这是由于在进程切换的时候，cache 包含正在执行的进程的数据。对物理寻址和虚拟寻址的 cache 来说，这些问题以不同方式产生，并且有不同的解决方法，比如使用进程标识符来确保一个进程只能获得它自己的数据。

⑤ 上下文切换：为允许另一个不同的进程使用处理器，改变处理器内部的状态，并保存当前进程返回时需要的状态。

5.7.7 处理 TLB 缺失和缺页

尽管当 TLB 命中时，利用 TLB 将虚拟地址转换成物理地址是很简单的，但是处理 TLB 缺失和缺页要复杂得多。当 TLB 中没有一个表项能匹配虚拟地址时，TLB 缺失就会发生。TLB 缺失有下面两种可能性之一：

- 1) 页在主存中，只需要创建缺失的 TLB 表项。
- 2) 页不在主存中，需要将控制权交给操作系统来解决缺页。

MIPS 通常采用软件来处理 TLB 缺失。它从主存中取出页表项装入 TLB，然后重新执行引起 TLB 缺失的那条指令，这时就会得到 TLB 命中。如果页表项指出该页不在主存中，此时就会发生缺页异常。

处理 TLB 缺失或者缺页需要使用异常机制来中断活跃的进程，将控制权传给操作系统，然

446

后恢复执行被中断的进程。缺页将在主存访问时钟周期的某一时刻被发现。为了在缺页处理完毕后重新启动引起缺页的指令，必须保存该指令的程序计数器中的值。正如第 4 章所述，**异常程序计数器**（Exception Program Counter，EPC）用来保存这个值。

另外，TLB 缺失或者缺页异常必须在访存发生的同一个时钟周期的末尾被判定，因此下一个时钟周期就开始进行异常处理而不是继续正常的指令执行。如果在这个时钟周期没有断定缺页发生，一条 load 指令可能改写寄存器，而当我们试图重新启动指令时，这可能是灾难性的错误。例如，考虑指令 `lw $1,0 ($1)`：计算机必须防止写流水级发生，否则，就不能重新启动指令，因为 `$1` 的内容将被破坏。store 指令也会发生类似复杂情况。当发生缺页而没有完成处理时，我们必须阻止写主存的操作；这通常是通过令到主存写控制线为无效来完成。

01 硬件/软件接口 在操作系统开始进行异常处理和保存处理器所有状态位的时候，操作系统特别脆弱。例如，如果在操作系统中正在处理第一个异常时，另一个异常又发生了，控制单元将重写异常程序计数器，就不能返回引起缺页的那条指令。我们可以通过提供禁止异常（disable exception）和使能异常（enable exception）来避免这种错误的发生。当异常第一次发生时，处理器设置一个管理态模式位，禁止其他异常的发生；这可以与处理器设置管理态模式位同时进行。随后操作系统保存足够的状态，如果有另一个异常发生——异常程序计数器（EPC）和异常引发寄存器也能保存这些状态。异常程序计数器和异常引发寄存器是协助处理异常、TLB 缺失以及缺页的两个特殊控制寄存器；图 5-33 列出了其他的寄存器。而后操作系统可以重新允许异常发生。这些步骤保证了异常不会使处理器丢失任何状态，因此也就不会出现无法重新执行中断指令的情况。

使能异常：也称为中断使能（interrupt enable），用于控制处理器是否响应异常的信号或动作；在处理器安全地保存重启所需信息之前，必须阻止异常的发生。

寄存器	CP0 寄存器号	说明
EPC	14	异常之后重启的位置
Cause	13	异常的原因
BadVAddr	8	引发异常的地址
Index	0	TLB 中读/写的位置
Random	1	TLB 中伪随机位置
EntryLo	2	物理页地址和标记位
EntryHi	10	虚页地址
Context	4	页表地址和页号

图 5-33 MIPS 控制寄存器。这些寄存器被视为位于协处理器 0 中，因此读时使用 `mfc0`，写时使用 `mtc0`

一旦操作系统知道了引起缺页的虚拟地址，它必须完成以下三个步骤：

- 1) 使用虚拟地址查找页表项，并在磁盘上找到被访问的页的位置。
- 2) 选择替换一个物理页；如果被选中的页被修改过，需要在把新的虚拟页装入之前将这个物理页写回到磁盘上。
- 3) 启动读操作，将被访问的页从磁盘上取回到所选择的物理页的位置上。

当然，最后一个步骤将花费数百万个时钟周期（如果被替换的页被重写过，那么第二步也需要花费这么多时间）；因此，操作系统通常都会选择另一个进程在处理器上执行直到磁盘访问结

447

束。由于操作系统已经保存了当前进程的状态，因此它可以很方便地将控制权交给另一个进程。

当从磁盘读页的操作完成后，操作系统可以恢复原先引起缺页的进程状态，并且执行从异常返回的指令。该指令将处理器从核心态恢复到用户态，同时也恢复程序计数器的值。用户进程接着重新执行引发缺页的那条指令，成功地访问请求的页，然后继续执行。

数据访问引起的缺页异常很难处理，这是由于以下三个特性：

- 1) 它们发生于指令中间，不同于指令缺页。
- 2) 在异常处理前指令没有结束。
- 3) 异常处理之后，指令必须重新执行，就好像什么都没发生过一样。

要保持指令可重新启动（restartable），这样异常被处理之后，指令也能继续执行，这在类似于 MIPS 的结构中的实现相对简单。因为每条指令只能写一个数据项并且只能在指令周期的最后进行写操作，我们就可以阻止指令的完成（不执行写操作）并且在开始处重新启动指令。

可重启指令：一种在异常被处理之后能从异常中恢复而不会影响指令的执行结果的指令。

我们再来看 MIPS 的一些细节。当 TLB 发生缺失时，MIPS 的硬件将被引用的页号保存在一个叫 BadVAddr 的特殊寄存器里，然后产生异常。

这个异常请求操作系统通过软件来处理缺失。控制权被传到地址 $8000\ 0000_{16}$ ——TLB 缺失处理程序（handler）的位置。为了找到缺失页的物理地址，TLB 缺失处理程序使用虚拟地址的页号，以及指向活动进程页表起始地址的页表寄存器来检索页表。为了能快速地检索，MIPS 将所需的一切信息都放在特殊的现场寄存器（Context）中：高 12 位是页表的基准地址，接下来的 18 位是缺失页的虚拟地址。每个页表项是 1 个字，因此最后两位为 0。因此，头两条指令将现场寄存器中的内容复制到内核临时寄存器 \$k1 中，然后根据其中的地址将页表项装入 \$k1。回想 \$k0 和 \$k1 是为操作系统保留的不做保存的寄存器；这样做的主要原因是使得 TLB 缺失处理程序执行得更快。下面是典型的 TLB 缺失处理程序的 MIPS 代码：

```
TLBmiss:
    mfc0  $k1,Context      # copy address of PTE into temp $k1
    lw     $k1,0($k1)        # put PTE into temp $k1
    mtc0  $k1,EntryLo       # put PTE into special register EntryLo
    tlbwr                         # put EntryLo into TLB entry at Random
    eret                           # return from TLB miss exception
```

处理程序：用于“处理”异常或中断的软件程序的名字。

正如上面所示，MIPS 有一组特殊的系统指令用来更新 TLB。指令 tlbwr 把控制寄存器 EntryLo 中的内容复制到由控制寄存器 Random 所选择的 TLB 表项中。Random 实现随机替换，所以它基本上是一个独立运行的计数器。TLB 缺失大概要花费 12 个时钟周期。

注意到 TLB 缺失处理程序并不检查页表项是否有效。因为发生 TLB 表项缺失异常比缺页异常要频繁得多，所以操作系统对页表中的表项并不做检查就直接装入 TLB 并重新执行指令。如果表项无效，另一个不同的异常就发生，操作系统认为缺页。这种方法让频繁发生的 TLB 缺失处理得快一些，但是对不频繁发生的缺页处理就会有一些性能损失。

一旦产生缺页的进程被中断，控制权就被转到 $8000\ 0180_{16}$ ，这是一个与 TLB 缺失处理程序不相同的地址。它是处理异常的通用地址；TLB 缺失有一个专门的入口点是为了减少 TLB 缺失代价的。操作系统使用异常引发寄存器来判断产生异常的原因。由于是缺页异常，操作系统知道需要进一步处理。因此，不同于 TLB 缺失，它保存了活动进程的全部状态，包括所有的通用寄存器和浮点寄存器、页表地址寄存器、EPC 和异常引发寄存器的状态。由于浮点寄存器在异常处理程序不常使用，因此通用入口点并没有保存它们，而是留给少数需要它们的处理器。

图 5-34 描述了异常处理程序的 MIPS 代码。我们使用 MIPS 代码来保存和恢复状态，注意何时允许和禁止异常，但是我们调用 C 代码来处理特殊的异常。

保存状态			
Save GPR	addi \$k1, \$sp, -XCPSIZE sw \$sp, XCT_SP(\$k1) sw \$v0, XCT_VO(\$k1) ... sw \$ra, XCT_RA(\$k1)	# save space on stack for state # save \$sp on stack # save \$v0 on stack # save \$v1, \$ai, \$si, \$ti,... on stack # save \$ra on stack	
保存 hi, lo	mfhi \$v0 mflo \$v1 sw \$v0, XCT_HI(\$k1) sw \$v1, XCT_LO(\$k1)	# copy Hi # copy Lo # save Hi value on stack # save Lo value on stack	
保存异常寄存器	mfc0 \$a0, \$cr sw \$a0, XCT_CR(\$k1) ... mfc0 \$a3, \$sr sw \$a3, XCT_SR(\$k1)	# copy cause register # save \$cr value on stack # save \$v1,... # copy status register # save \$sr on stack	
设置sp	move \$sp, \$k1	# sp = sp - XCPSIZE	
寄存器允许嵌套异常			
	andi \$v0, \$a3, MASK1 mtco \$v0, \$sr	# \$v0 = \$sr & MASK1, enable exceptions # \$sr = value that enables exceptions	
调用C异常处理程序			
Set \$gp	move \$gp, GPINIT	# set \$gp to point to heap area	
Call C code	move \$a0, \$sp jal xcpt_deliver	# arg1 = pointer to exception stack # call C code to handle exception	
恢复状态			
Restore most GPR, hi, lo	move \$at, \$sp lw \$ra, XCT_RA(\$at) ... lw \$a0, XCT_A0(\$k1)	# temporary value of \$sp # restore \$ra from stack # restore \$t0,..., \$a1 # restore \$a0 from stack	
恢复状态寄存器	lw \$v0, XCT_SR(\$at) li \$v1, MASK2 and \$v0, \$v0, \$v1 mtco \$v0, \$sr	# load old \$sr from stack # mask to disable exceptions # \$v0 = \$sr & MASK2, disable exceptions # set status register	
异常返回			
Restore \$sp and rest of GPR used as temporary registers	lw \$sp, XCT_SP(\$at) lw \$v0, XCT_VO(\$at) lw \$v1, XCT_V1(\$at) lw \$k1, XCT_EPC(\$at) lw \$at, XCT_AT(\$at)	# restore \$sp from stack # restore \$v0 from stack # restore \$v1 from stack # copy old \$epc from stack # restore \$at from stack	
Restore ERC and return	mtco \$k1, \$epc eret \$ra	# restore \$epc # return to interrupted instruction	

图 5-34 异常时保存状态和恢复状态的 MIPS 码

引发缺失的虚拟地址取决于当前缺失是指令缺失还是数据缺失。产生缺失的指令地址在 EPC 中。如果是指令缺页，EPC 中包含了缺失页的虚拟地址；否则，缺失页的虚拟地址可以通过查看指令（指令地址在 EPC 中），找到基址寄存器和偏移量来计算得到。

❶ 精解 这个简化版本假设了堆栈指针（sp）有效。为了避免执行低层异常代码时发生缺页的问题，MIPS 预留了一部分不会产生缺页的地址空间，称为非映射（unmapped）。操作系统将异常入口点代码和异常堆栈存放在非映射的内存中。MIPS 硬件将虚拟地址

$8000\ 0000_{16} \sim BFFF\ FFFF_{16}$ 转换成物理地址时，虚拟地址的高位忽略不计，即把这些地址放在物理内存的低位。因此，操作系统就将异常入口点和异常堆栈放置于非映射的主存中。

- 非映射：地址空间中的一个部分，在这个区域不会导致缺页异常。

01 精解 图 5-34 中的代码显示了 MIPS - 32 的异常返回序列。早先的 MIPS-I 架构采用 rfe 和 jr 来代替 eret。

01 精解 对于有着更为复杂指令的处理器来说，可能会访问主存中的很多位置并且写很多数据项，这就使指令重新启动变得更加困难。处理一条指令可能在指令中间产生多次缺页。例如，x86 处理器有能访问成百上千数据字的块移动指令。在这样的处理器中，指令通常无法像在 MIPS 中那样从起始位置重新启动。相反，指令必须被中断，稍后从执行中断处继续执行。在执行的中间恢复一条指令通常需要保存一些特殊状态，处理异常，然后恢复那些特殊状态。要正确地执行这项工作需要在操作系统的异常处理代码和硬件中进行细致而详细的协调。

01 精解 与每次存储器访问都需要一次间接寻址不同，虚拟机支持一个影子页表（shadow page table）用于进行用户虚拟地址到硬件物理地址的转换。通过检测对用户页表的所有修改，虚拟机可以确保硬件正在用于转换的影子页表表项与用户操作系统中的页表表项一致，不同的是在用户页表中使用正确的物理地址替代了实地址。因此，虚拟机必须在用户操作系统试图改变其页表或访问页表指针时产生自陷。这通常由用户操作系统通过对用户页表进行写保护和对页表指针的任何访问产生自陷来实现。如前所述，如果是特权操作访问页表指针后会发生后面一种情况。

01 精解 体系结构中需要虚拟化的最后一部分是 I/O。由于计算机中 I/O 设备数量和类型不断增加，I/O 虚拟化就变成了系统虚拟化中最困难的一部分。另外一个难点是多个虚拟机之间共享实际的设备。还有一个问题是支持大量的设备驱动程序，这在一个支持多个用户操作系统的虚拟系统上更加严重。它为每种虚拟机中各种类型的 I/O 设备提供一个通用的驱动，并且将其留给 VMM 以管理实际的 I/O。

01 精解 除了要对指令集进行虚拟之外，另一个挑战是虚拟存储器的虚拟化，这主要是因为每种虚拟机上的操作系统要维护自己的页表。为了使其工作，虚拟机中实存储器和物理存储器是两个不同的概念（这两个概念通常被认为是相同的），实存储器是位于虚拟存储器和物理存储器之间的一个独立的层次。（有人使用虚拟存储器、物理存储器和机器存储器来表示相同的三个层次。）用户操作系统通过其页表将虚拟存储器映射到实存储器，虚拟机页表将用户实存储器映射到物理存储器，虚拟存储器体系结构要么如 IBM VM/370 和 x86 一样通过页表实现，要么如 MIPS 一样通过 TLB 实现。

5.7.8 小结

虚拟存储器是管理主存和磁盘之间数据缓存的一级存储器层次。虚拟存储器允许单个程序在主存有限的范围内扩展地址空间。更重要的是，虚拟存储器以一种保护的方式，同时支持多个活跃的进程共享主存。

管理主存和磁盘之间的存储器层次结构很具有挑战性，这是因为缺页的代价很高。通常采用下面一些技术来降低缺失率：

- 1) 增大页的容量以便利用空间局部性并降低缺失率。
- 2) 使用页表实现的虚拟地址和物理地址之间的映射采用全相联的方式，这样虚拟页就可以被放置到主存中的任何位置。

452

- 3) 操作系统使用类似 LRU 和访问位之类的技术来选择替换哪一页。

写磁盘的代价是很高的，因此虚拟存储器使用写回机制并且跟踪是否有一页更改过（采用脏位）以避免把没有变化的页写回到磁盘。

虚拟存储器机制提供了从被程序使用的虚拟地址到用来访问主存的物理地址空间之间的转换。这个地址转换允许对主存进行受保护的共享，同时还提供了很多额外的好处，如简化了存储器分配。为了保证进程间受到保护，要求只有操作系统才能改变地址变换，这是通过防止用户程序更改页表来实现的。可以在操作系统的帮助下实现在进程之间受控制地共享页，页表中的访问位被用来指出用户程序对页进行读访问还是写访问。

如果对于每一次访问，处理器不得不访问主存中的页表来进行转换，这样虚拟存储器的开销将很大，cache 也将失去意义。相反，对于页表，TLB 扮演了地址转换 cache 的角色，利用 TLB 中的变换，将虚拟地址转换为物理地址。

cache、虚拟存储器以及 TLB 都建立在一组共同的原理和策略基础上。下一节讨论这个共同的架构。

01

理解程序性能 尽管虚拟存储器能使一个小容量的存储器看起来像大容量的存储器，但二级存储器和主存之间的性能差异意味着，如果一个程序经常访问比它拥有的物理存储器多的虚拟存储器，程序运行速度就会很慢。这样的程序会不断地在存储器和磁盘之间交换页面，称为抖动 (thrashing)。抖动的发生将会是灾难，但很少见。如果你的程序产生抖动，那么最简单的解决方式就是让你的程序在一个有着更大存储器的计算机上运行，或者为你的计算机增加存储器。一个复杂的方法是重新检查所使用的算法和数据结构，看看能否改变它的局部性，从而减少程序同时使用的页数。这一组页通常被称为工作集 (working set)。

一个更常见的性能问题是 TLB 缺失。由于 TLB 同时只能处理 32~64 个页表项，一个程序很容易会有较高的 TLB 缺失率，因为处理器只能直接访问不到 $64 \times 4\text{KiB} = 0.25\text{MiB}$ 。例如，对于基数排序，TLB 缺失通常是一个挑战。为了缓解这个问题，现在很多计算机体系结构都支持可变的页大小。例如，除了 4KiB 的标准页面，MIPS 硬件还支持 16KiB、64KiB、256KiB、1MiB、4MiB、16MiB、64MiB 和 256MiB 大小的页面。因此，如果一个程序使用大容量的页面，就能直接访问更多主存而不会有 TLB 缺失。

453

令操作系统允许程序选择这些大容量的页面也是一个实际的难题。同样，减少 TLB 缺失更为复杂的方法是重新检查算法和数据结构以减少页面工作集；另外，由于存储器访问对于性能以及 TLB 缺失频率至关重要，所以一些工作集较大的程序已经在这方面做了重新设计。

01

小测验

将左边的存储器层次结构组成部分与右边最匹配的说明连线：

- | | |
|-------------|------------------|
| 1. 一级 cache | a. cache 的 cache |
| 2. 二级 cache | b. 磁盘的 cache |
| 3. 主存 | c. 主存的 cache |
| 4. TLB | d. 页表项的 cache |

5.8 存储器层次结构的一般框架

到目前为止，我们已经知道了不同类型的存储器层次结构共用许多原理。尽管存储器层次结构中很多方面都有量的区别，但是决定层次结构如何运作的许多策略和特征在本质上是相同的。图 5-35 给出了存储器层次结构的一些量的特征区别。在本节的剩余部分，我们将讨论存储器层次结构的共同运作方面以及这些方面将如何决定它们的行为。我们通过一系列适用于存储器层次结构两层之间的 4 个问题来研究这些策略，为了简单起见，我们主要使用 cache 中的术语。

特征	一级 cache 的典型值	二级 cache 的典型值	页式存储器的 典型值	TLB 的 典型值
块的总容量	250 ~ 2 000	2 500 ~ 25 000	16 000 ~ 250 000	40 ~ 1 024
以 KB 计量的总容量	16 ~ 64	125 ~ 2 000	1 000 000 ~ 1 000 000 000	0.25 ~ 16
块的字节数	16 ~ 64	64 ~ 128	4 000 ~ 64 000	4 ~ 32
缺失代价的时钟周期数	10 ~ 25	100 ~ 1 000	10 000 000 ~ 100 000 000	10 ~ 1 000
缺失率（二级 cache 是全局缺失）	2% ~ 5%	0.1% ~ 2%	0.000 01% ~ 0.000 1%	0.01% ~ 2%

图 5-35 计算机中存储器层次结构主要组成部分的关键定量设计参数。本图是这些层次在 2012 年的典型值。值的范围很大，一部分原因是许多值是随着时间的变化而变化的；例如，当 cache 容量变大以克服较高的缺失代价时，块容量也随之增长。图中没有显示的是，服务器处理器中还有三级 cache，容量通常为 2 ~ 8 MiB，块数比二级 cache 多很多。三级 cache 使二级 cache 的缺失代价降低到 30 ~ 40 个时钟周期。

5.8.1 问题 1：一个块可以被放在何处

我们已经看到，可以根据很多机制将块放置到存储器层次的较高层结构中，从直接映射到组相联，再到全相联。就像前面所提到的，这些机制都可以看成是组数和每组块数各不相同的组相联方案的特例：

机制	名称组数	每组块数
直接映射	cache 中的块数	1
组相联	cache 中的块数/相联度	相联度（一般为 2 ~ 16）
全相联	1	cache 中的块数

增加相联度的好处在于它通常能降低缺失率。缺失率的改进来自于减少竞争同一位置而产生的缺失。我们稍后将详细讨论。首先来看能获得多少性能改进。图 5-36 显示了不同的 cache 容量，在相联度从直接映射到八路组相联之间变化的缺失率。最大的改进出现在直接映射变化到两路组相联，缺失率下降了 20% ~ 30%。当 cache 容量增加时，相联度的提高对性能改进作用很小；这是因为大容量 cache 的总的缺失率很低，从而改进缺失率的机会减少，并且由相联度引起的缺失率的绝对改进明显减少。如前所述，相联度增加的潜在缺点是增加了代价以及访问时间。

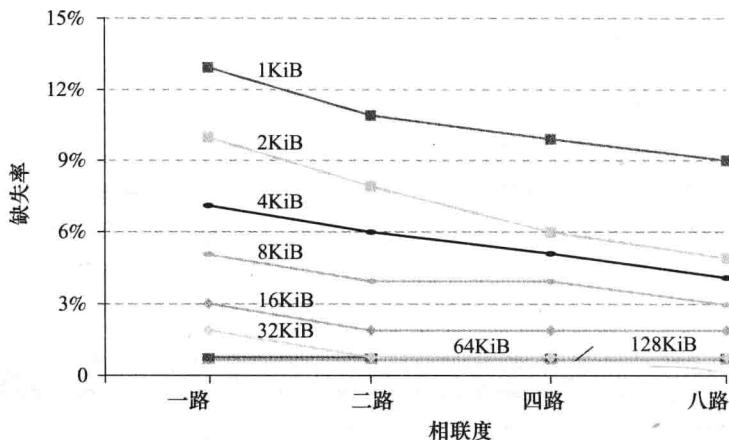


图 5-36 当相联度增加时，8 种不同容量数据 cache 各自的缺失率。从一路（直接映射）到两路组相联变化时获益明显，进一步增加相联度所获得的好处就小一些了（例如，从两路到四路提高了 1% ~ 10%，而从一路到两路提高了 20% ~ 30%）。从四路到八路组相联，缺失率的改进更小，它们反而接近于全相联 cache 的缺失率。容量小的 cache 由于其本身缺失率较高，因此从相联度所获得的好处就很明显。图 5-16 解释了这些数据是如何收集的

5.8.2 问题 2：如何找到一个块

我们如何选择一个块的存放位置取决于块放置机制，因为它指明了可能存放位置的数量。我们可以把这些机制总结如下：

相联度	定位方法	需要比较的次数
直接映射	索引	1
组相联	索引组，查找组中元素	相联的度
全相联	查找所有 cache 项	cache 的容量
	独立的查找表	0

在存储器层次结构中选择直接映射、组相联还是全相联映射取决于缺失代价和相联度实现代价的权衡情况，包括了时间和额外硬件开销。在片内拥有二级 cache 允许实现更高的相联度，这是因为命中时间不再关键，设计者也不用依靠标准 SRAM 芯片来构建块。除非容量很小，否则 cache 不使用全相联映射方式，在小容量 cache 中，比较器的开销并不是压倒性的因素，其绝对缺失率的改进才是最明显的。

在虚拟存储器系统中，页表是一个独立的映射表，它用来索引存储器。除了表本身需要占用存储资源外，使用索引表还会引起额外的存储器访问。选择全相联映射和额外的页表有以下几个原因：

- 1) 全相联有其优越性，因为缺失的代价非常高。
- 2) 全相联允许软件使用复杂的替换策略以降低缺失率。
- 3) 全映射很容易被索引，而不需要额外的硬件，也不需要进行查找。

因此，虚拟存储系统通常使用全相联映射。

组相联映射通常用于 cache 和 TLB，访问时包括索引和在小组内查找。一些系统使用直接映射的 cache，这是因为访问时间短并且实现简单。访问时间短是因为不需要比较就能找到被请求的块。这样的设计选择取决于许多细节的实现，如 cache 是否集成在片上，实现 cache 的

技术以及 cache 访问时间对处理器时钟周期的重要性。

5.8.3 问题3：当cache缺失时替换哪一块

在相联的 cache 中发生缺失时，我们必须决定替换哪一块。如果是全相联 cache，所有的块都是被替换的候选者。如果 cache 是组相联的，我们必须在某一组的块中进行选择。当然，直接映射的 cache 的替换很简单，因为只有一个可以替换的候选者。

在组相联或者全相联 cache 中，有两种主要的替换策略：

- 随机法：随机选择候选块，可能使用一些硬件协助实现。例如，对于 TLB 缺失，MIPS 支持随机替换。
- 最近最少使用算法：被替换的块是最久没有被使用过的块。

实际应用中，在相联度不低（典型的是两路到四路）的层次结构中实现 LRU 的代价太高了，这是因为跟踪使用信息的代价很高。尽管对于四路组相联，LRU 通常也是近似实现的——例如，跟踪记录哪一对块是最近最少使用的（需要使用 1 位），然后跟踪记录每对块中哪一块又是最近最少使用的（要求每对使用 1 位）。

对于更高的相联度的层次结构，可以用近似的 LRU 算法，也可以采用随机替换策略。在 cache 中，替换算法是由硬件实现的，这意味着算法应该容易实现。随机替换算法用硬件很容易实现，而对于两路组相联的 cache，使用随机替换算法的缺失率要比 LRU 替换算法的缺失率高 1.1 倍。随着 cache 变得更大，所有替换策略的缺失率都下降了，绝对差别也变小了。事实上，有时候，随机替换算法的性能比用硬件简单实现的近似 LRU 算法的性能还要好。

在虚拟存储器中，LRU 的一些形式都是近似的，因为当缺失代价很大时，缺失率即使只有微小的降低都是很重要的。通常提供引用位或者其他等价的功能使操作系统更方便地追踪一组最近最少使用的项。由于缺失的代价特别高，并且相对来说不经常发生，主要用软件来近似这项信息的做法是可行的。

5.8.4 问题4：写操作如何处理

对任何存储器层次结构来说，一个关键的问题是如何处理写操作。我们已经看到了两种基本选项：

- 写直达：信息被同时写到 cache 的块和存储器层次结构较低层的块中（对 cache 来说是指向主存）。5.3 节中的 cache 使用这个机制。
- 写回：信息仅仅写到 cache 中的块。被改写的块只有在它被替换时才写回到存储器层次结构的较低层中。虚拟存储器系统通常采用写回策略，原因在 5.7 节中讨论过。

写回和写直达策略有其各自的优点，写回的主要优点如下：

- 处理器可以以 cache 而不是存储器能接收的速度写单个的字。
- 多次写同一块中的字只需对存储器层次结构较低层进行一次写操作。
- 当块被写回时，由于写一整块，系统可以充分利用高带宽传输。

写直达的优点如下：

- 缺失比较简单，缺失代价也较小，这是因为不需要把整个块写回到较低层存储系统中。
- 尽管为了可行性，写直达的 cache 需要一个写缓冲区，然而写直达还是比写回更易于实现。

在虚拟存储器系统中，由于写到存储器层次结构的较低层（磁盘）的延迟很大，因此只有写回策略是可行的。尽管允许存储器的物理、逻辑宽度更宽，并对 DRAM 采用突发模式，然

而处理器产生写操作的速度通常还是超过存储系统可以处理它们的速度。因此，现在最低一级的 cache 通常采用写回策略。

01 重点 cache、TLB 和虚拟存储器可能一开始看起来非常不同，但是它们都基于相同的两个定位原理，并且可以通过对 4 个问题的各自解答来理解。

问题 1：一个块可以被放在何处？

答：一个位置（直接映射），一些位置（组相联），或者是任何位置（全相联）。

问题 2：如何找到一个块？

答：有 4 种方法：索引（在直接映射的 cache 中），有限的检索（在组相联的 cache 中），全部检索（在全相联的 cache 中）和专用查找表（在页表中）。

问题 3：当 cache 缺失时替换哪一块？

答：通常是最近最少使用的块或者是随机选取的一块。

问题 4：写操作如何处理？

答：层次结构中的每一层都可以使用写直达或者写回策略。

458

5.8.5 3C：一种理解存储器层次结构行为的直观模型

在这一节中，我们来看一个模型，通过它能够很好地洞察存储器层次结构中引起缺失的原因以及层次结构的变化对缺失的影响。我们从 cache 方面来解释这个观点，尽管这个观点对其他层次也都直接适用。在这个模型中，所有的缺失被分成下面三类（3C 模型（three Cs model））：

- **强制缺失**（compulsory miss）：对从没有在 cache 中出现的块第一次进行访问引起的缺失。也称为**冷启动缺失**（cold-start miss）。
 - **容量缺失**（capacity miss）：由于 cache 容纳不了一个程序执行所需的所有块而引起的 cache 缺失，当某些块被替换出去，随后再被调入时，将发生容量缺失。
 - **冲突缺失**（conflict miss）：在组相联或者直接映射的 cache 中，多个块竞争同一个组时而引起的 cache 缺失。冲突缺失在直接映射或组相联 cache 中存在，而在同样大小的全相联 cache 中不存在。这种 cache 缺失也称为**碰撞缺失**（collision miss）。
- ② 3C 模型：将所有的 cache 缺失都归为三种类型的 cache 模型，三类分别为：强制缺失、容量缺失和冲突缺失。因其三类名称的英文单词首字母均为 c 而得名。
- ③ 强制缺失：也称为冷启动缺失。对没有在 cache 中出现过的块第一次访问时产生的缺失。
- ④ 容量缺失：由于 cache 在全相联时都不可能容纳所有请求的块而导致的缺失。
- ⑤ 冲突缺失：也称为碰撞缺失。在组相联或者直接映射 cache 中，很多块为了竞争同一个组导致的缺失。这种缺失在使用相同大小的全相联 cache 中是不存在的。

图 5-37 显示了缺失率是如何按照引起的原因被分为三种的。改变 cache 设计中的某一方面就能直接影响这些缺失的原因。冲突缺失是因为争用同一个 cache 块而引起的，因此提高相联度就可以减少冲突缺失。然而，提高相联度会延长访问时间，导致整个性能的降低。

容量缺失可以简单地通过增大 cache 容量来减少；的确，多年来二级 cache 的容量总是在不断地增加。当然，在增大 cache 的同时，我们也必须注意访问时间的增长，这将导致整体性能的降低。因此，尽管一级 cache 也在增大，但是增大得非常缓慢。

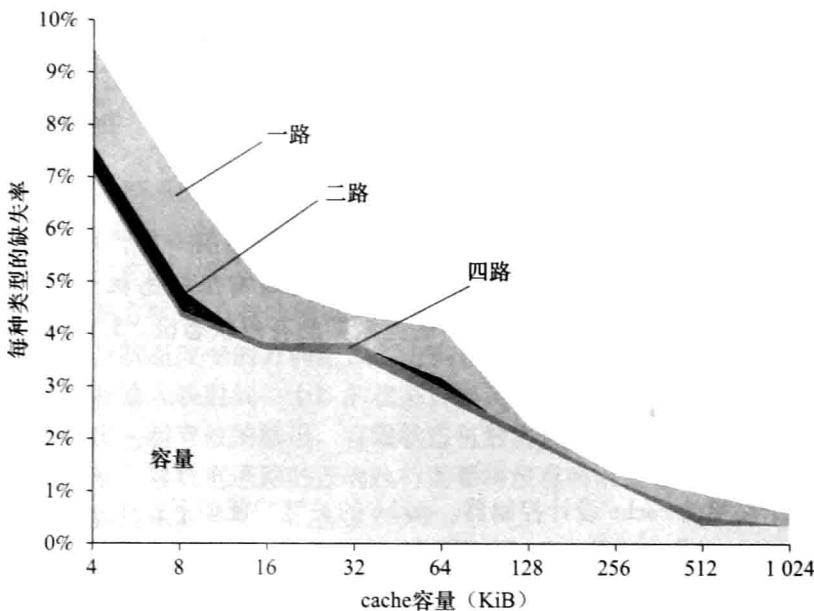


图 5-37 根据缺失原因缺失率被分成三种。这幅图显示了不同容量 cache 的总缺失率及其组成部分。数据与图 5-36 出自同一来源，都是由 SPEC CPU 2000 整型和浮点基准程序测试得到的。强制缺失部分只占 0.006%，在图中看不出来。下一部分是容量缺失，取决于 cache 的容量。冲突缺失部分既取决于相联度，又取决于 cache 的容量，图中给出了相联度从一路到八路的冲突缺失率。在每种情况下，当相联度从下一个更高度变化到标记地方的相联度时，标记地方对应缺失率的增加。例如，标有两路的部分说明当 cache 相联度从四路变化到两路时缺失增加。因此，同样大小的直接映射 cache 和全相联 cache 的缺失率的差别由标记着八路、四路、两路和一路的各部分之和给出。八路和四路之间变化太小，以至于在图中很难看出。

由于强制缺失是对块的第一次访问产生的，因此，对 cache 系统来说，减少强制缺失次数最主要的方法是增加块的大小。由于程序将由较少的 cache 块组成，因此这就减少了对程序每一块都要访问一次的情况下的总的访问次数。如前所述，块容量增加太多可能对性能产生负面影响，因为缺失代价会增长。

将缺失分成 3C 是个有用的定性模型。在实际 cache 设计中，许多设计的选择是相互影响的，改变 cache 的一个特征通常会影响另一些缺失率的组成部分。尽管存在这些缺点，3C 模型对于观察 cache 设计的性能来说仍是一种有效的方法。

459

01 重点 存储器层次结构设计所面临的挑战在于：任何一个改进缺失率的设计同时也可能对整体性能产生负面的影响，如图 5-38 所示。正面与负面作用的结合就使得存储器层次结构的设计令人关注。

设计变化	对缺失率的影响	可能对性能产生的负面影响
增加 cache 容量	减少了容量缺失	可能增加访问时间
提高相联度	由于减少了冲突缺失，因此降低了缺失率	可能增加访问时间
增加块的容量	由于空间局部性，因此对很宽范围内变化的块大小，都能降低缺失率	增加缺失代价，块太大还会增加缺失率

图 5-38 存储器层次结构设计面临的挑战

01 小测验

下面哪些表述（如果有）是正确的？

1. 没有减少强制缺失的方法。
2. 全相联 cache 中没有冲突缺失。
3. 在减少缺失方面，相联度比容量更为重要。

5.9 使用有限状态机来控制简单的 cache

就像我们在第 4 章中对单周期、流水线数据通路实现控制一样，现在我们可以实现对 cache 的控制。本节从定义一个简单的 cache 开始，随后对有限状态机（finite-state machine, FSM）进行介绍。最后介绍了这个简单 cache 的控制器的有限状态机。5.12 节用一种新的硬件描述语言更深入地介绍了 cache 和控制器。

5.9.1 一个简单的 cache

460
461

我们将为一个简单的 cache 设计控制器。cache 的关键特征如下：

- 直接映射的 cache。
- 写回机制，采用写分配策略。
- 块大小为 4 个字（16 字节或者 128 位）。
- cache 大小为 16KiB，因此它能容纳 1 024 个块。
- 32 字节地址。
- cache 中每个块包含一个有效位和写入位。

根据 5.3 节，我们可以计算出 cache 的地址域：

- cache 索引位为 10 位。
- 块偏移为 4 位。
- 标记位为 $32 - (10 + 4) = 18$ 位。

处理器和 cache 之间的信号为：

- 1 位读/写信号。
- 1 位有效信号，指示是否有一个 cache 操作。
- 32 位地址。
- 32 位数据（从处理器到 cache）。
- 32 位数据（从 cache 到处理器）。
- 1 位准备信号，指示 cache 操作完成。

存储器和 cache 之间的接口与处理器和 cache 之间一样有相同的域，除了数据域这里是 128 位宽。如今，一般的微处理器都有额外的存储器位宽，在处理器中可以处理 32 位或 64 位的字，而 DRAM 控制器通常是 128 位。为了简化设计，可以使 cache 块匹配 DRAM 的位宽。下面是一些信号：

- 1 位读/写信号。
- 1 位有效信号，指示是否有一个存储器操作。
- 32 位地址。
- 128 位数据（从 cache 到存储器）。
- 128 位数据（从存储器到 cache）。
- 1 位准备信号，指示存储器操作完成。

请注意，到存储器的接口并没有固定的周期数。我们假设当存储器读或写完成后，存储器控制器通过准备信号来通知 cache。

在介绍 cache 控制器之前，我们需要回顾一下有限状态机，它支持控制一个花费多个时钟周期的操作。

462

5.9.2 有限状态机

为了给单周期的数据通路设计控制单元，我们使用一组真值表，根据指令的分类来指定控制信号的设置。对于 cache，由于操作可以是一系列的步骤，因此控制变得更加复杂。对 cache 的控制既要指定在任何步骤中信号的设置，又要依次指出下一步的步骤。

最常见的多步控制方法基于有限状态机（finite-state machine），通常以图形化表示。有限状态机由一组状态以及状态改变的方向组成。方向由下一状态函数（next-state function）来定义，它将当前的状态和输入映射到一个新的状态。当我们使用有限状态机控制时，每个状态还要在当时的状态下指出一组有效的输出。有限状态机的实现通常假定那些没有明确置为有效的输出是无效的。类似地，对数据通路的正确执行需要将没有明确设置为有效的信号设置成无效状态，而不是对信号置位采取不关心的态度。

- ② 有限状态机：由一组输入和输出，以及下一状态函数和输出函数组成的时序逻辑函数。下一状态函数将当前状态和当前输入映射为一个新的状态，输出函数将当前状态和当前输入映射为一组确定的输出。
- ③ 下一状态函数：根据当前状态及当前输入来确定有限状态机下一状态的组合函数。

多路选择控制略微有一些不同，它们从输入（0 或 1）中选择一个。因此，在有限状态机中，我们总是指定我们关心的所有多路选择控制的设置。当我们使用逻辑实现有限状态机时，设置为 0 的控制可能就是默认值，因此不需要任何门电路。一个简单的有限状态机的例子在附录 B 中给出，如果不熟悉有限状态机的概念，在继续学习之前，读者可能需要花一些时间来研究附录 B。

一个有限状态机的实现包括：一个保持当前状态的临时寄存器和一个组合逻辑，组合逻辑用来决定有效的数据通路信号和下一状态。图 5-39 显示了可能的实现效果图。附录 D 详细介绍了使用这个结构如何实现有限状态机。在 B.3 节中，一个有限状态机的组合逻辑由 ROM（read-only memory，只读存储器）或 PLA（programmable logic array，可编程逻辑阵列）来实现。（附录 B 中对这些逻辑单元也进行了描述。）

01 精解 注意，这是一个阻塞式 cache，因此处理器必须等到 cache 处理完请求之后才能继续执行。5.12 节中将会讲述另外一种称为非阻塞式 cache 的结构。

01 精解 本书中的有限状态机的类型被称作 Moore 型有限状态机，以 Edward Moore 来命名。它的标识特征是输出仅仅取决于当前的状态。对于 Moore 型有限状态机，标记着组合控制逻辑的逻辑单元可以被分成两部分：一部分包括控制输出，并且仅有状态输入；另一部分仅包含下一状态输出。

另一种状态机是 Mealy 型有限状态机，以 George Mealy 命名。Mealy 型有限状态机的输出取决于输入和当前的状态。Moore 型有限状态机潜在的实现优势在于速度和控制单元的规模。由于在时钟周期开始就需要控制输出，而该输出与输入无关，仅仅取决于当前的状态，因此有助于速度的提升。在附录 B 中，用逻辑门就可以实现这种有限状态机，因而可以很明显地看出它在规模上的优势。Moore 型有限状态机潜在的缺点是它可能需要额外的状态。例如，在两个状态序列中仅有一个状态不同的情况下，Mealy 状态机会通过使用输出依赖输入的方法将状态统一。

463

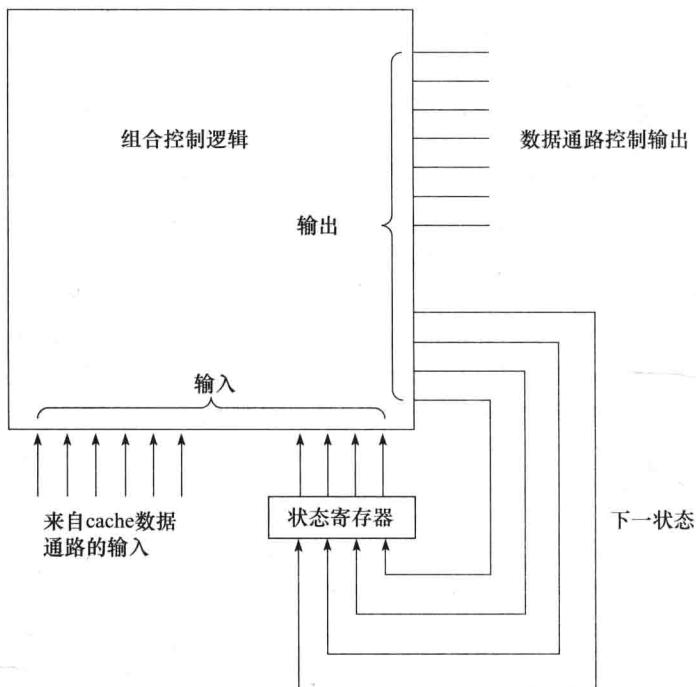


图 5-39 典型的有限状态机控制器由一个组合逻辑和一个保存当前状态的寄存器来实现。组合逻辑的输出是下一个状态号以及当前状态的有效控制信号。组合逻辑的输入是当前的状态以及用来决定下一状态的一些输入。注意到，在本章所使用的有限状态机中，输出仅由当前状态来决定，而与输入无关。对此，精解更详细地进行了解释

5.9.3 一个简单的 cache 控制器的有限状态机

图 5-40 是简单 cache 控制器的 4 个状态：

- 空闲：这个状态等待从处理器发出有效的读/写请求，使得有限状态机转移到标记比较的状态。
- 标志比较：如名称所示，这个状态主要检测该读/写请求是命中还是缺失。地址的索引部分用来选择比较用的标记。如果它的有效位和地址的标记部分与标记位相匹配，则命中。这时，或者从选中的字中读出数据，或者将数据写入选中的字，随后 cache 准备信号被置位。如果是写操作，还要将脏位设置为 1。注意，如果是写命中，还要设置有效位和标记域；这些设置看起来并不需要，却还是要设置，因为标记使用单独的存储器，因此，改变写入位时，我们也要改变有效位和标记域。如果请求命中并且 cache 块有效，有限状态机返回到空闲状态。发生一次缺失时首先要更新 cache 标记，随后，如果这个位置的块的写入位为 1，则转入写回状态；如果写入位为 0，则进入分配状态。
- 写回：这个状态根据标记和 cache 索引组合的地址，将 128 位的块写回存储器。我们继续停留在该状态等待存储器返回准备信号。当存储器写回完成时，有限状态机进入分配状态。
- 分配：新的块从存储器中取回。我们继续停留在该状态等待从存储器返回准备信号。当存储器读操作完成时，有限状态机转入标记比较状态。尽管我们可以转移到一个新的状态来完成操作，而不再使用标记比较状态，但是这个操作中有很多重复，包括当访问是写操作时更新块中恰当的字。

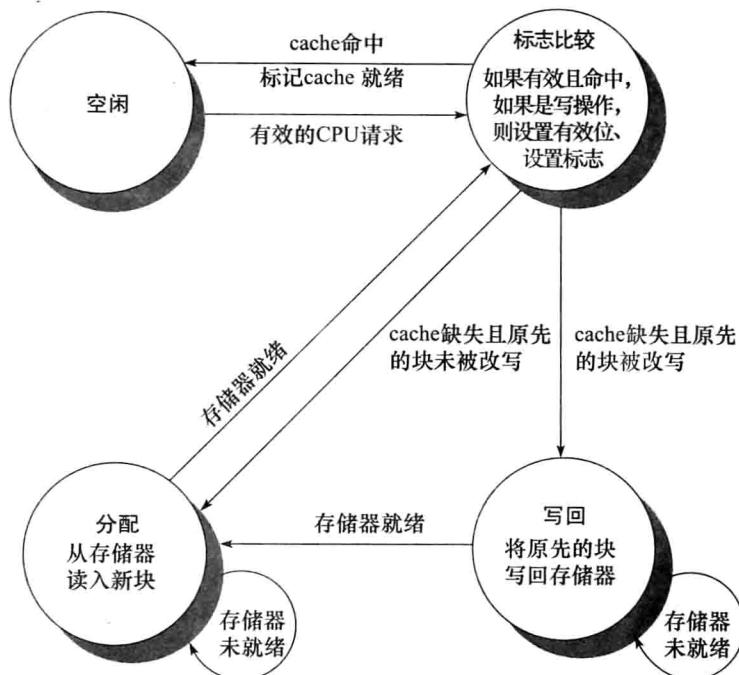


图 5-40 简单控制器的 4 个状态

这个简单的模型可以很方便地扩展到多个状态以改进性能。例如，标记比较状态在一个单独的时钟周期里既要比较，又要读/写 cache 数据。通常，比较和 cache 访问被放在分离的状态中，以改进时钟周期。另一个优化是增加一个写缓冲，这样我们就可以先保存脏块，然后再读出新的块。这样，当一个脏块缺失时，处理器就不用等待两次存储器访问。随后，cache 将从写缓冲器中将脏块写回，同时处理器正在处理被请求的数据。

在 5.12 节将对有限状态机进行更深入的研究，用硬件描述语言描述了整个控制器，并显示了这个简单 cache 的方框图。

5.10 并行与存储器层次结构：cache 一致性

多核多处理器意味着在单芯片上有多个处理器，这些处理器可能会共享一个公共的物理地址空间。cache 共享数据带来了一个新的问题，由于两个不同的处理器所保存的存储器视图是通过各自的 cache 得到的，如果没有其他的防范措施，两个处理器可能分别得到两个不同的值。图 5-41 解释了这个问题，并且说明了为什么两个不同的处理器对存储器相同位置进行操作会得到不同的值。这个问题通常称为 cache 一致性问题。

时间	事件	CPU A 的 cache 内容	CPU B 的 cache 内容	存储器位置 X 的内容
0				0
1	CPU A 读 X	0		0
2	CPU B 读 X	0	0	0
3	CPU A 向 X 写入 1	1	0	1

图 5-41 cache 一致性问题：两个处理器（A 和 B）对同一个存储器位置 X 进行读写操作。我们假设最初两个 cache 中都不包含该变量并且 X 的值为 0。假设是写直达 cache；如果是写回 cache 则会带来额外的更加复杂的情况。当 X 的值被 A 改写后，A 的 cache 和存储器中的副本都做了更新，但是 B 的 cache 没有，如果 B 读 X，得到的值为 0

一般情况下，如果在一个存储器系统中读取任何一个数据项的返回结果总是最近写入的值，那么可以认为该存储器具有一致性。这个定义尽管看起来是正确的，但仍很模糊而且过于简单；实际情况复杂得多。这个简单的定义包括了存储器系统行为的两个不同方面，它们对于编写正确的共享存储程序是至关重要的。第一个方面称为一致性（coherence），它定义了读操作可以返回什么样的数值。第二个方面称为连贯性（consistency），它定义了写入的数据什么时候才能被读操作返回。

首先来看一致性。如果一个存储系统满足如下条件，那么认为该存储系统是一致的：

1) 处理器 P 对位置 X 的写操作后面紧跟着处理器 P 对 X 的读操作，并且在这次读操作和写操作之间没有其他处理器对 X 进行写操作，这时读操作总是返回 P 写入的数值。因此，在图 5-41 中，如果 CPU A 在时间 3 之后读 X，它将得到数值 1。

2) 在其他处理器对 X 的写操作后，处理器 P 对 X 执行读操作，这两个操作之间有足够的间隔并且没有其他处理器对 X 进行写操作，这时，读操作返回的是写入的数值。因此，在图 5-41 中，我们需要一个机制，以便在时间 3，CPU A 向存储器地址 X 写入数据 1 之后，CPU B 的 cache 中的数值 0 被数值 1 所替换。

3) 对同一个地址的写操作是串行执行的（serialized）；也就是说，任何两个处理器对同一个地址的两个写操作在所有处理器看来都有相同的顺序。例如，如果在时间 3 之后，CPU B 又向存储器地址 X 中写入 2，那么处理器绝不会从该地址中先读出 2 再读出 1。

第一个性质保证了程序的顺序——即使在单处理器中也要保证这个性质。第二个性质定义了存储器的一致性意味着什么：如果一个处理器总是读到旧的数值，我们就认为这个存储器是非一致性的。

写操作串行化的要求更加细致，但也同等重要。假如我们没有将写操作串行化，处理器 P1 写入地址 X 之后，紧跟着处理器 P2 也会写入地址 X。写操作串行化保证了每个处理器都能在某个时间看到 P2 写入的结果。如果没有将写操作串行化，就会出现一些处理器先看到 P2 写入的结果再看到 P1 写入的结果，从而可能保留了 P1 写入的数值。避免这种情况最简单的方法就是保证对同一个地址的写操作在所有处理器看来都具有相同的顺序，这个性质称为写串行化（write serialization）。

5.10.1 实现一致性的基本方案

在支持 cache 一致性的多处理器系统中，cache 提供共享数据的迁移（migration）和复制（replication）。

- **迁移：**数据项可以移入本地 cache 并以透明的方式使用。迁移不但减少了访问远程共享数据项的延迟，而且减少了对共享存储器带宽的需求。
- **复制：**当共享数据被同时读取时，cache 在本地对数据项做了备份。复制减少了访问延迟和读取共享数据时的竞争现象。

对这种迁移和复制的支持对于访问共享数据的性能来说是至关重要的，因此许多处理器引入硬件协议来维护 cache 一致性。这个用于维护多个处理器一致性的协议称为 cache 一致性协议（cache coherence protocol）。实现 cache 一致性协议的关键在于跟踪所有共享数据块的状态。

最常用的 cache 一致性协议是监听（snooping）协议。每个含有物理存储器中数据块副本的 cache 还要保留该数据块共享状态的副本，但是并不集中地保存状态。cache 可以通过一些广播媒介（总线或者网络）访问，所有的 cache 控制器对媒介进行监视或者监听，来确定它们是否含有总线或者交换机上请求的数据块副本。

在后面章节我们将介绍用共享总线实现基于监听的 cache 一致性方法，任何可以向所有处理器广播 cache 缺失的通信媒介都可以用来实现基于监听的一致性机制。这种向所有 cache 广播的方法使得监听协议的实现变得简单，但是也限制了其可扩展性。

5.10.2 监听协议

实现一致性的一种方法是：在处理器写数据之前，保证该处理器能独占地访问该数据项。这种协议称为写无效协议（write invalidate protocol），因为它在执行写操作的时候令其他副本无效。独占访问确保了写操作执行时不存在其他可读或可写的数据项副本：cache 中该数据项的其他所有副本都是无效的。

图 5-42 给出了一个基于监听总线的写无效协议的例子，其中 cache 使用写回机制。为了说明这个协议如何保证一致性，我们令写操作后面紧跟着其他处理器执行读操作的情况：由于写操作需要独占访问，执行读操作的处理器中保存的任何副本就要被置无效（协议因此得名）。因此，当执行读操作时，在 cache 中发生缺失，cache 需要取回新的数据副本。对于写操作，我们要求执行写操作的处理器可以独占访问，以防止其他处理器同时执行写操作。如果两个处理器试图同时对同一个数据项进行写操作，它们中的一个会在竞争中获胜，这就使得另一个处理器的副本被置为无效。竞争失败的处理器要完成写操作，就必须取得新的数据副本，这个副本中已经包含了更新后的数据。因此，这个协议也强制了写操作的串行化。

468

处理器动作	总线动作	CPU A 的 cache 内容	CPU B 的 cache 内容	存储器中位置 X 的内容
				0
CPU A 读 X	X 在 cache 中缺失	0		0
CPU B 读 X	X 在 cache 中缺失	0	0	0
CPU A 向 X 写 1	令 X 无效	1		0
CPU B 读 X	X 在 cache 中缺失	1	1	1

图 5-42 以对单个 cache 块 X 读写的过程为例（采用写回机制），说明监听总线上执行无效协议的过程。我们假设最初两个 cache 中都没有 X，而在存储器中 X 的值为 0。CPU 和 X 的存储器内容是处理器和总线动作都完成后的数值。空格表示没有动作或者没有存放副本。当 B 发生第二次缺失时，CPU A 回应，同时取消来自存储器的响应。随后，B 的 cache 和 X 的存储器内容都得到更新。这种当块共享时对存储器进行更新的方法简化了协议，但是可能只有当块被替换时才有跟踪所有权并强制写回。这就需要引入一个被称为“所有者”（owner）的额外状态，它表明块可以被共享，但是当块被改变或是替换时，由所有者处理器负责更新其他处理器和存储器。

01 硬件/软件接口 一种观点是：块大小对 cache 一致性起着重要作用。以对一个 cache 监听为例，cache 的块大小为 8 个字，两个处理器可以对块中的一个字进行读/写操作。多数协议会在两个处理器之间交换整个块，因此增加了所需要的一致性带宽。

大的块同样会引起所谓的假共享（false sharing）：当两个不相关的共享变量存在相同的 cache 块中时，尽管每个处理器访问的是不同的变量，但是在处理器之间还是将整个块进行交换。因此，程序员和编译器需要谨慎放置数据以避免发生假共享。

假共享：当两个不相关的共享变量放在相同的 cache 块中时，尽管每个处理器访问的是不同的变量，但是在处理器之间还是将整个块进行交换。

469

精解 尽管前面的三个属性已经能充分保证一致性，但是何时能看见写的值，这个问题同样很重要。让我们来看看为什么。注意到在图 5-41 中，我们不能要求对 X 的读操作立刻能看见其他处理器对 X 执行写操作的值。例如，假设一个处理器对 X 的写操作稍稍先于另一个处理器对 X 的读操作，这样就不能保证读操作返回的数值是被写的数据，因为在那一刻，被写的数据可能甚至还没有离开处理器。连贯性模型详细定义了写数据何时能被读操作看见。

我们做下面两个假设：第一，直到所有处理器看见写操作的结果，这个写操作才能完成（没有完成时可以允许下一个写操作发生）；第二，处理器不能改变与存储器访问相关的写操作的次序。这两个条件意味着：如果处理器在写位置 X 之后再写位置 Y，那么，任何处理器在看到 Y 的新值时也必须看见 X 的新值。这些限制条件允许处理器对读操作可以重新排序，但是强制处理器以程序执行的顺序完成写操作。

精解 由于输入操作可在不改变 cache 内容的情况下改变存储器内容，另外，在写回 cache 中，输出操作需要最新的存储器内容，因此在单处理器系统中也存在 I/O 和 cache 之间一致性问题，这与多处理器间 cache 一致性问题相同。cache 一致性问题对于多处理器和 I/O（见第 6 章）来说，尽管原因相同，但是却有不同的特性，从而影响了解决方法。与几乎很少拥有多个数据副本的 I/O 不同——只要可能有就应该避免——程序运行在多个处理器上时，cache 中通常都有相同数据的副本。

精解 除了分布地保存共享块状态的监听式 cache 一致性协议，基于目录的 cache 一致性协议将物理存储器的共享块的状态存放在一个地点，称之为目录（directory）。尽管基于目录的一致性比监听式一致性的实现开销略高一些，但是这种方法可以减少 cache 之间的通信，并且因此可以扩展更多的处理器。

5.11 并行与存储器层次结构：冗余廉价磁盘阵列

本节内容在网站上，讲述了如何采用许多块磁盘并行工作来提高吞吐率，该技术是冗余廉价磁盘阵列（Redundant Arrays of Inexpensive Disks, RAID）产生的灵感所在。然而，RAID 技术真正流行的原因在于其通过采用适当数量的冗余磁盘来提高可靠性。本节讲述了不同 RAID 级别在性能、开销和可靠性等方面的区别。

5.12 高级内容：实现 cache 控制器

本节内容在网站中，介绍了如何实现 cache 的控制，就像我们在第 4 章中实现对单周期、流水的数据通路的控制一样。这一节开始介绍了有限状态机以及在简单的数据 cache 中实现 cache 控制器，包括用硬件描述语言来描述 cache 控制器。随后详细介绍了一个 cache 一致性协议的实例以及实现的难点。

5.13 实例：ARM Cortex-A8 和 Intel Core i7 的存储器层次结构

本节将讲述第 4 章中提到的两种微处理器（ARM Cortex-A8 和 Intel Core i7）的存储器层次。本节内容基于《计算机体系结构：量化研究方法》第 5 版的 2.6 节。

图 5-43 总结了这两种处理器的地址尺寸和 TLB。注意，A8 中包含了 2 个具有 32 位虚拟地址空间和物理地址空间的 TLB。而 Core i7 中包含了 3 个具有 48 位虚拟地址空间和 44 位物理地址空间的 TLB。虽然 Core i7 中的 64 位寄存器能够支持更大的虚拟地址空间，但是没有软件需要如此大的空间，48 位的虚拟地址不但缩小了页表的踪迹（footprint），也简化了 TLB 的硬件。

470

特点	ARM Cortex-A8	Intel Core i7
虚拟地址	32 位	48 位
物理地址	32 位	44 位
页表大小	可变：4, 16, 64KiB, 1, 16MiB	可变：4KiB, 2/4MiB
TLB 组织	1 个指令 TLB 和 1 个数据 TLB 两个 TLB 均为全相联，32 个入口， 轮转替换策略 硬件处理 TLB 缺失	每核 1 个指令 TLB 和 1 个数据 TLB 两个 L1 TLB 均为 4 路组相联， LRU 替换策略 L1 I-TLB 对于小尺寸页面有 128 个入口，每线程对于大页面有 7 个入口 L1 D-TLB 对于小页面有 64 个入口，大页面有 32 个入口 L2 TLB 4 路组相联，LRU 替换策略 L2 TLB 有 512 个入口 硬件处理 TLB 缺失

图 5-43 ARM Cortex-A8 和 Intel Core i7 920 的地址转换和 TLB 硬件。两个处理器均支持用于操作系统或映射为帧缓冲器的大页面。大页面技术避免了将一个对象映射到多个入口的情况

图 5-44 给出了它们的 cache。需要注意的是 A8 只有 1 个核，而 Core i7 中有 4 个核。两个处理器的 L1 指令 cache 组织结构相同，容量均为 32KiB，都是 4 路组相联，cache 块均为 64 字节。A8 的数据 cache 和指令 cache 相同，而 Core i7 把相联度提高到了 8 路。虽然 A8 L2 cache 容量可以在 128KiB ~ 1MiB 间变化，而 Core i7 的 L2 cache 容量固定在 256KiB，但是两个处理器的 L2 cache 都采用 64 字节的 cache 块。由于 Core i7 用于服务器，因此它也提供了共享的片上 L3 cache，其容量与核的数目相关。对于 4 核而言，L3 cache 容量为 8MiB。

471

特点	ARM Cortex-A8	Intel Nehalem
L1 cache 组织	数据指令分离 cache	数据指令分离 cache
L1 cache 容量	数据/指令 cache 均为 32KiB	每核的数据/指令 cache 均为 32KiB
L1 cache 相联度	4 路 (I), 4 路 (D) 组相联	4 路 (I), 8 路 (D) 组相联
L1 替换策略	随机	近似 LRU
L1 块大小	64 字节	64 字节
L1 写策略	写回，按写分配 (?)	写回，不按写分配
L1 命中时间 (load)	1 个时钟周期	4 个时钟周期，流水执行
L2 cache 组织	统一（指令和数据）	每个核统一（指令和数据）
L2 cache 容量	128KiB ~ 1MiB	256KiB (0.25MiB)
L2 cache 相联度	8 路组相联	8 路组相联
L2 替换策略	随机 (?)	近似 LRU
L2 块大小	64 字节	64 字节
L2 写策略	写回，按写分配 (?)	写回，按写分配
L2 命中时间	11 个时钟周期	10 个时钟周期
L3 cache 组织	—	统一（指令和数据）
L3 cache 容量	—	8MiB，共享
L3 cache 相联度	—	16 路组相联
L3 替换策略	—	近似 LRU
L3 块大小	—	64 字节
L3 写策略	—	写回，按写分配
L3 命中时间	—	35 个时钟周期

图 5-44 ARM Cortex-A8 和 Intel Core i7 920 的 cache

在 A8 和 Core i7 中，cache 设计者面临的一个巨大挑战是要支持每个时钟周期执行一条以上的访存指令。通常的一种做法是将 cache 分成多个体 (bank)，从而在不发生体冲突时，能够对多个体并行进行访问。该技术与 DRAM 中的存储体间交叉类似 (见 5.2 节)。

Core i7 采用了另外一些优化技术来降低缺失开销。第一种是请求字优先策略。另外，在 cache 缺失时，继续执行访问数据 cache 的指令，这是设计者在设计乱序执行处理器时，为了隐藏 cache 缺失开销通常采用的技术，称为非阻塞 cache (nonblocking cache)。它们实现了无阻塞的两个特点，缺失命中 (hit under miss) 允许在缺失期间有其他的 cache 命中；缺失情况下的缺失 (miss under miss) 允许有多个未解决的 cache 缺失。这两者中第一个致力于用其他工作来隐藏一部分缺失延迟，而第二个的目标在于重叠两个不同缺失的延迟。

- ② 非阻塞 cache：在处理器处理前面的 cache 缺失时仍可正常访问的 cache。

要重叠多个未解决的缺失的大部分缺失时间，需要一个高带宽的存储系统来并行地处理多个缺失。在个人移动设备中，存储器只能获得这项功能的有限的益处，但是大型服务器和多处理器通常拥有的存储系统能并行处理不止一个缺失。
472

Core i7 采用了数据访问预取技术，在数据缺失前，根据缺失数据的特点来预测下次数据访问的地址，并使用该地址进行数据预取。该技术在访问循环中的数组时非常有效。

这些芯片的存储器层次非常复杂，且芯核上很大一部分用作 cache 和 TLB。这些都是为了解决处理器运行和存储访问速度之间的巨大差异的结果。

A8 和 Core i7 存储器层次的性能

对 Cortex-A8 的存储器层次进行仿真时，L2 cache 组织为 1MiB 八路组相联结构，使用整数的 Minnespec 基准测试程序。如第 4 章所述，Minnespec 基准测试程序由 SPEC2000 基准测试程序组成，但是使用了不同的输入集来将运行时间减少几个数量级。虽然使用小输入集没有改变指令的组合情况，但是影响了 cache 行为。例如，在 SPEC2000 中访存最为密集的整数程序 mcf 中，对于 32KiB 的 cache 来说，Minnespec 的 cache 缺失率只有完整版本 SPEC2000 的 65%。而对于 1MiB 的 cache，该数据只有 1/6。因此，不能将 Minnespec 和 SPEC2000 之间进行比较，更不能和用于 Core i7 的更大的 SPEC2006 比较。Core i7 在 SPEC2006 上的仿真结果如图 5-47 所示。然而，像在第 4 章中一样，在考察 L1 和 L2 cache 缺失率和整体 CPI 的相对影响时，这些数据非常有用。

对于这些基准测试程序而言，A8 的指令 cache 缺失率非常低，即使对于 L1 cache 也是如此：大部分程序的 L1 指令 cache 的缺失率接近于 0，所有程序的 L1 指令 cache 的缺失率在 1% 以下。其原因主要是 SPEC 程序属于计算密集型，且四路组相联的 cache 组织结构消除了大多数的块冲突。图 5-45 给出了 A8 的数据 cache 的结果，其 L1 和 L2 的缺失率都比较高。1GHz 的 Cortex-A8 的 L1 缺失率的开销为 11 个时钟周期，L2 缺失率的开销为 60 个时钟周期。图 5-46 给出了每次数据访问的平均缺失开销。

图 5-47 给出了基准测试程序 SPEC2006 在 Core i7 上运行时的 cache 缺失率。一级指令 cache 的缺失率在 0.1% ~ 1.8% 之间，平均值比 0.4% 高一点点。该缺失率与基准测试程序 SPEC2006 在其他研究中的缺失率一致。一级数据 cache 的缺失率在 5% ~ 10%，有时高一些，二级和三级 cache 的重要性就显而易见了。对于二级 cache 而言，缺失时的访存开销在 100 个时钟周期以上，且缺失率为 4%，因此三级 cache 非常关键。假定有一半的指令是取数或存数指令，如果没有三级 cache，则二级 cache 缺失将导致 CPI 增加 2 倍！1% 的三级数据 cache 缺失率仍然有点高，但是比二级 cache 缺失率低 4 倍，比一级 cache 缺失率低 6 倍。
473

01 精解 由于推测执行有时会推测错误（见第 4 章），有一些对一级数据 cache 进行访问的取数或存数指令最终没有执行，图 5-45 中的数据是对所有数据请求的统计，包括那些最

终被取消的访问。真正完成的数据访问的缺失率要高 1.6 倍（一级数据 cache 平均缺失率为 95% 比 5.9%）。

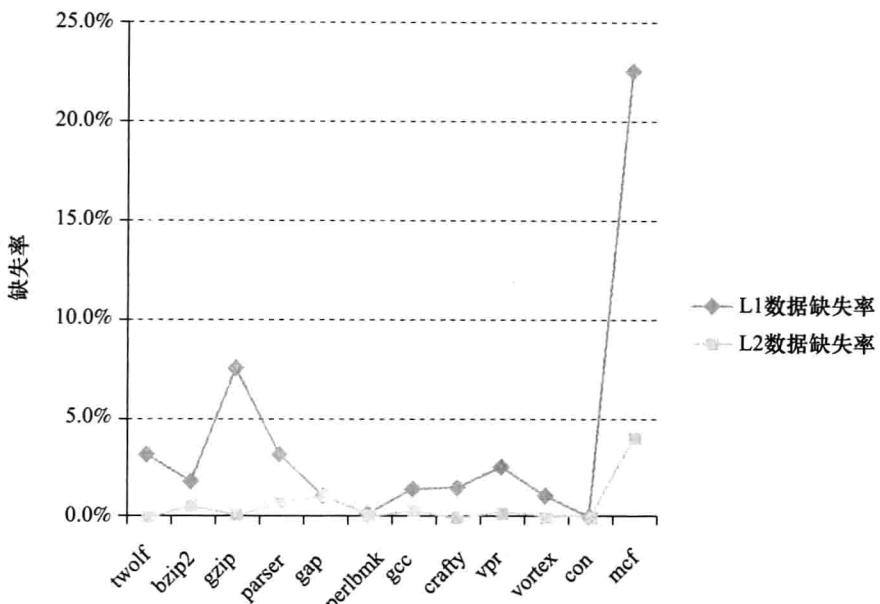


图 5-45 在 ARM Cortex-A8 上运行 SPEC2000 的简化版本 Minnespec 时的数据 cache 缺失率。对存储器需求大的应用的一级 cache 和二级 cache 的缺失率较高。需要注意的是二级 cache 缺失率是全局缺失率，也就是说，包括那些一级 cache 命中的所有访问（见 5.4 节的精解）。mcf 是 cache 不友好的程序。注意，该图与第 4 章图 4-76 使用相同的系统和基准测试程序

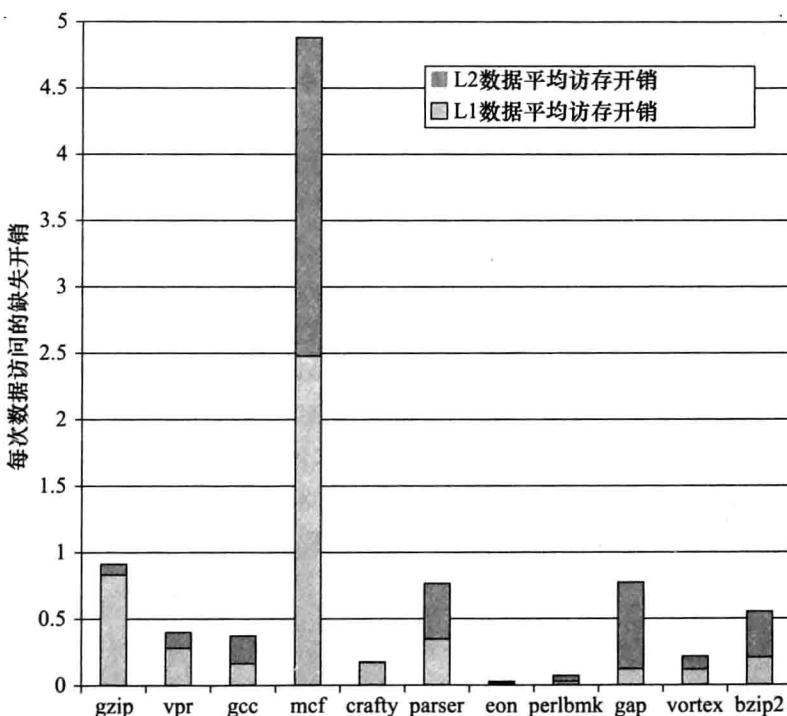


图 5-46 在 ARM 处理器上运行 Minnespec 时，每次数据存储器访问时一级和二级 cache 的平均访存开销（以时钟周期计）。虽然一级 cache 非常高，但是二级 cache 的缺失开销高了 5 倍以上，这意味着二级 cache 缺失对性能影响非常大

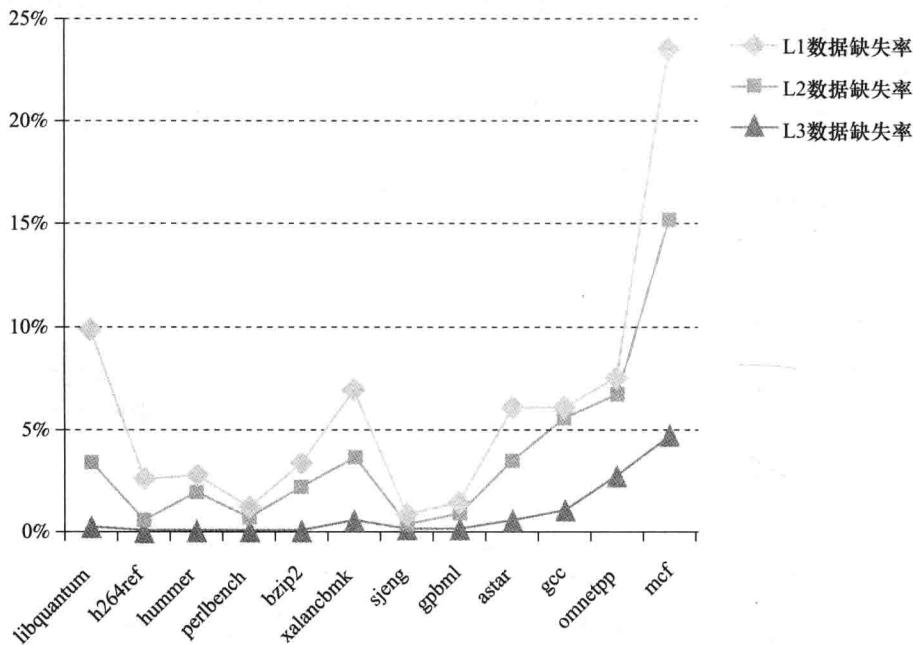


图 5-47 在 Intel Core i7 920 上运行完整的 SPEC2006 整数程序是一级、二级和三级数据 cache 的缺失率

5.14 运行更快：cache 分块和矩阵乘法

在第3章和第4章中已经通过子字并行和指令级并行来优化DGEMM的性能，可进一步通过在硬件上采用cache分块技术继续对其性能进行优化。图5-48给出了图4-80中DGEMM的分块版本。其变化与从图3-21中未做优化的DGEMM版本到图5-21的分块版本类似。此处使用第4章中循环展开后的DGEMM版本，并将其在A、B、C的子矩阵上调用多次。事实上，除了第7行中循环次数增量不同外，图5-48中第28~34行和第7~8行分别与图5-21中第14~20行和第5~6相同。

与前面章节不同，本节没有给出对应的x86代码，这主要是因为分块技术对于计算没有影响，只是访问存储器中的数据顺序发生了变换，从而内循环的代码与图4-81中的代码几乎相同。不同点在于使用了用于实现循环的bookkeeping整数指令。图4-80中内循环之前有14条指令，内循环之后有8条指令，而图5-48中产生的bookkeeping代码中，内循环之前有40条指令，内循环之后有28条指令。但是，相比cache缺失率减少对性能的贡献，这些额外的指令开销就显得微不足道了。图5-49对比了未采用优化技术的性能和采用子字并行、指令级并行和cache优化技术的性能对比。对于大矩阵运算，分块技术使未采用展开的AVX代码性能提升了2~2.5倍。如果同时采用这三种优化技术，则性能提高8~15倍，且矩阵越大，性能提升越大。

01 精解 如3.8节中精解所述，这些结果都是在将Turbo模式关闭时获得的。与第3章和第4章相同，如果将Turbo模式打开，则时钟频率将暂时提高到 $3.6/2.6 = 1.27$ 倍。在这种情况下，因为只使用了八个核中的一个核，所以Turbo模式运行得很好。然而，如果要运行得更快，则需要使用所有的核，这将在第6章中讲述。

```

1 #include <x86intrin.h>
2 #define UNROLL (4)
3 #define BLOCKSIZE 32
4 void do_block (int n, int si, int sj, int sk,
5                 double *A, double *B, double *C)
6 {
7     for ( int i = si; i < si+BLOCKSIZE; i+=UNROLL*4 )
8         for ( int j = sj; j < sj+BLOCKSIZE; j++ ) {
9             __m256d c[4];
10            for ( int x = 0; x < UNROLL; x++ )
11                c[x] = _mm256_load_pd(C+i+x*4+j*n);
12            /* c[x] = C[i][j] */
13            for( int k = sk; k < sk+BLOCKSIZE; k++ )
14            {
15                __m256d b = _mm256_broadcast_sd(B+k+j*n);
16            /* b = B[k][j] */
17                for (int x = 0; x < UNROLL; x++)
18                    c[x] = _mm256_add_pd(c[x], /* c[x]+=A[i][k]*b */
19                                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
20            }
21
22            for ( int x = 0; x < UNROLL; x++ )
23                _mm256_store_pd(C+i+x*4+j*n, c[x]);
24            /* C[i][j] = c[x] */
25        }
26    }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30     for ( int sj = 0; sj < n; sj += BLOCKSIZE )
31         for ( int si = 0; si < n; si += BLOCKSIZE )
32             for ( int sk = 0; sk < n; sk += BLOCKSIZE )
33                 do_block(n, si, sj, sk, A, B, C);
34 }

```

图 5-48 对图 4-80 中的 DGE MM 使用 cache 分块优化的 C 版本。与图 5-21 中的变化相同。编译器为 do_block 函数生成的汇编代码与图 4-81 中的代码几乎相同。需要再次强调的是，由于编译器才用内联函数调用，do_block 的调用没有开销

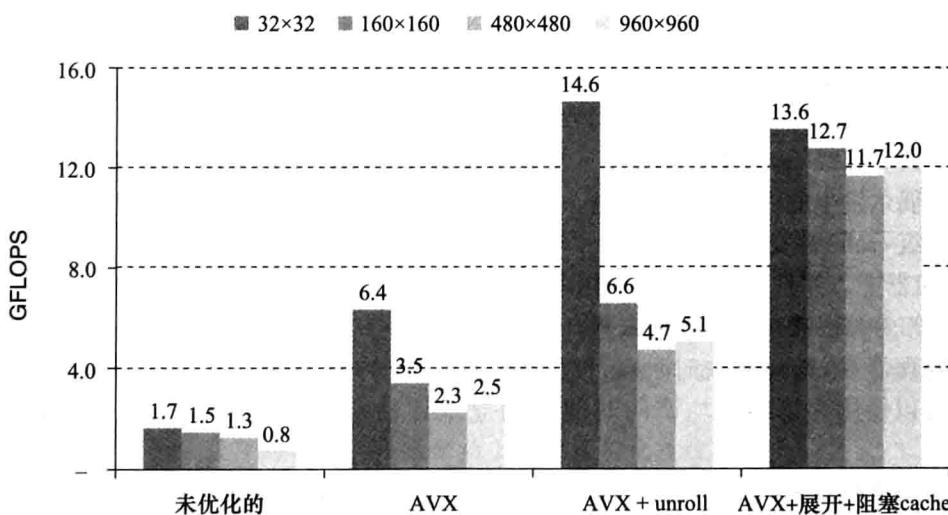


图 5-49 当矩阵规模从 32×32 增加到 960×960 时 4 种版本的 DGE MM 的性能。对于规模最大的矩阵，完全优化的代码的性能几乎是第 3 章图 3-21 中未优化代码的 15 倍