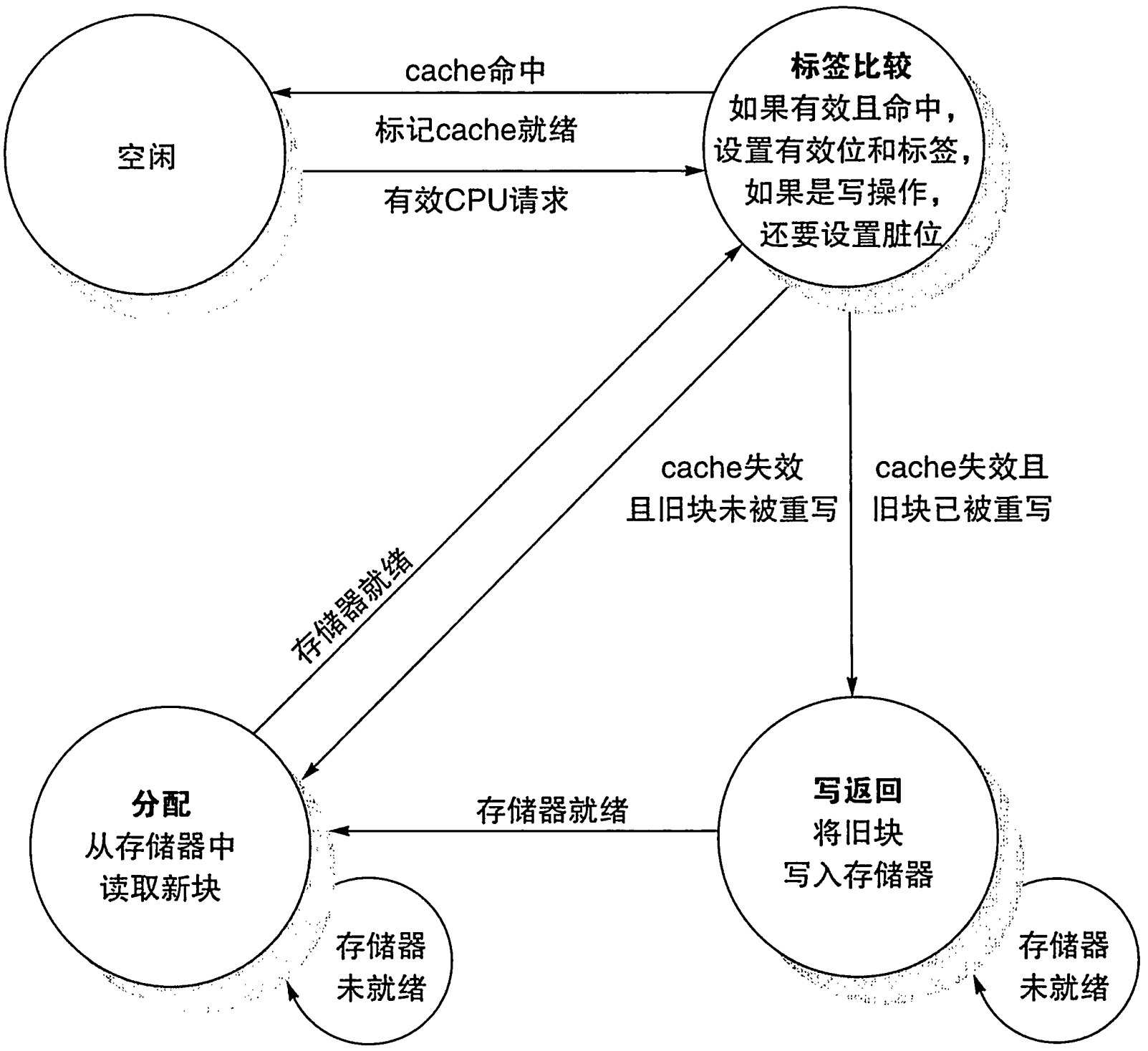


图 5-39　简单控制器的四个状态

- 标签比较：正如名称所示，该状态检测读或写请求是命中还是失效。地址的索引部分选择用于比较的标签。如果地址中的索引部分引用的 cache 块中的数据是有效的，并且地址中的标签部分与标签相匹配，则命中。如果是加载指令就从选择的字中读取数据，如果是存储指令就将数据写入选择的字中。之后设置 cache 就绪信号。如果这是一个写操作，脏位还要设置为 1。需要注意的是，写命中也需要设置有效位和标签字段，即使这看上去并不需要。这是因为标签使用单独的存储器，因此在改变脏位时也需要同时改变有效位和标签字段。如果发生命中并且当前块是有效的，有限状态自动机会返回空闲状态。失效时先更新 cache 标签，之后如果当前块的脏位为 1，则跳转到写回状态，如果该位为 0，则跳转到分配状态。
- 写返回：该状态使用由标签和 cache 索引组成的地址将 128 位的块写回存储器。之后继续停留在该状态等待存储器发出就绪信号。等待存储器写操作完成后，有限状态自动机跳转到分配状态。
- 分配：从存储器中取出一个新块。之后继续停留在该状态等待存储器发出就绪信号。等待存储器读操作完成后，有限状态自动机跳转到标签比较状态。尽管我们可以不重新使用标签比较状态而跳转到一个新的状态完成操作，但是分配状态之后的操作与标签比较状态的操作有大量的重叠，包括当访问为写操作时更新块中相应的字。

这个简单模型很容易扩展至更多的状态以提升性能。例如，标签比较状态在一个时钟周期内同时做了比较操作和 cache 数据的读和写操作。通常会将比较操作和 cache 访问操作分成两个状态以改进每个时钟周期所需的时间。另一个优化是添加一个写缓存，这样可以保存脏块，之后就可以提前读取新的块，使得处理器在脏块缺失时无须等待两次存储器访问。之

后 cache 会从写缓存中将脏块写回，同时处理器处理被请求的数据。

5.12 节中对有限状态自动机进行了更详细的描述，用硬件描述语言描述了整个控制器，并展示了这个简单 cache 的框图。

5.10　并行和存储层次结构：cache 一致性

假定一个多核多处理器意味着在一个单芯片上有多个处理器，这些处理器很可能共享一个共同的物理地址空间。cache 共享数据引入了一个新的问题，由于两个不同的处理器保存的存储器视图是通过它们各自的 cache 得到的，如果没有任何额外的保护措施，那么处理器可能看到两个不同的值。图 5-40 说明了这个问题，并且展示了对于同一个地址两个不同的处理器是如何有两个不同的值的。这个难点通常被称为 cache 一致性问题。

时间	事件	CPU A cache内容	CPU B cache内容	内存位置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 5-40　两个处理器 A 和 B 对同一个内存位置 X 的读写造成的 cache 一致性问题。我们假设最初两个处理器的 cache 中都不包含该变量，且 X 的值为 0。我们还假设这里的 cache 是写直达 cache，如果是写返回 cache 则会增加一些额外但是相似的复杂情况。在 X 的值被 A 写入后，A 的 cache 和内存中均保存新值，但是 B 的 cache 中的值没有改变，如果此时 B 读取 X 的值，会得到原值 0 而不是新值 1

简单来说，如果对任何一个数据项的读取都能返回该数据项最近被写入的值，则称这样的存储系统是一致的。虽然在直观上这个含糊而简单的定义很易懂，但实际的情况会比这个定义更加复杂。这个简单的定义涵盖了存储器系统行为的两个不同方面，这两个方面对于编写正确的共享存储程序都非常重要。第一个方面称为一致性（cache coherence），定义了读取操作会返回什么值。第二个方面称为连续性（memory consistency），定义了写入的值什么时候会被读取操作返回。

首先看一致性。如果满足以下属性，则称一个存储系统是一致的：

1. 在处理器 P 对位置 X 的值写入后，P 读 X 的值，如果在 P 对 X 的写和读操作之间没有其他处理器对 X 的写操作，那么本次读一定可以返回 P 写入的值。因此，在图 5-40 中，如果 CPU A 在第 3 步后读 X 的值，返回值应为 1。

2. 如果一个处理器对位置 X 的读操作是跟随在另一个处理器写入 X 值之后，并且读操作和写操作之间有足够的时间间隔、在两次对 X 的访问之间没有其他写入 X 的操作，那么本次读操作应该返回上次写入的值。因此，在图 5-40 中，需要一个机制使得在第 3 步 CPU A 将值 1 写入存储器位置 X 后，将 CPU B cache 中的值 0 替换为 1。

3. 对同一位置的写入操作是串行的。也就是说，两个处理器对同一位置的两次写入操作在其他处理器看来都具有相同的顺序。例如，如果 CPU B 在第 3 步后在存储器位置 X 上存入值 2，那么处理器永远不可能在读 X 值得到 2 之后再次读 X 值得到 1。

第一个属性仅仅保留了程序的顺序——我们当然希望这个属性在单处理机中是正确的。第二个属性定义拥有存储器的一致性意味着什么的概念：如果一个处理器总是读取到旧的数据值，我们就可以很清楚地说这个存储器是不一致的。

对写操作串行化的需求更加精细，但是也同样重要。假设没有将写操作串行化，并且处理器 P1 写入位置 X，之后 P2 也写入位置 X。串行化写操作确保每个处理器都可以在某个时刻看到 P2 写入的结果。如果没有串行化写操作，就可能有一些处理器先看到 P2 写入的结果，随后又看到 P1 写入的结果，最终可能保留 P1 写入的值。避免这种情况的最简单的方法就是确保对同一位置的所有写操作都以相同的顺序被观察到，这就是我们说的写操作串行化。

5.10.1 实现一致性的基本方案

在一个支持 cache 一致性的多处理器中，cache 为每个共享数据项提供迁移和复制：

- 迁移：数据项可以移动到本地 cache 并以透明的方式被使用。迁移减少了访问远程分配的共享数据项的延迟，也减少了共享存储器的带宽需求。
- 复制：当同时读取共享数据时，cache 会在本地 cache 中创建数据项的副本。复制减少了读取共享数据项的访问延迟和争用。

支持迁移和复制对于访问共享数据的性能至关重要，因此许多处理器都采用硬件协议来维护 cache 的一致性。维护多个处理器之间的一致性的协议被称为 cache 一致性协议。实现 cache 一致性协议的关键是追踪每一个共享数据块的状态。

最常用的 cache 一致性协议是监听。cache 中既包含物理存储器数据块副本，也含有该数据块的共享状态，但并不集中保留状态。这些 cache 都可以通过一些广播媒介（总线或网络）访问，而且所有的 cache 控制器都可以监视或监听媒介，以确定它们是否有总线或交换机的访问所需的数据块的副本。

在下面的章节中，我们将介绍通过共享总线来实现的基于监听的 cache 一致性，任何可以向所有处理器广播 cache 失效的通信介质都可以用于实现基于监听的一致性策略。这种向所有 cache 广播的方法使得监听协议易于实现，但同时限制了它的可扩展性。

5.10.2 监听协议

一种实现一致性的方法是确保处理器在写入一个数据项前可以独占访问该项。这种协议被称为*写无效协议*，因为它在写入时使得其他 cache 中的副本无效。独占访问可确保在写入时没有该项的其他可读写副本存在：所有该项的其他 cache 中的副本都将失效。

图 5-41 是一个使用写回 cache 的监听总线失效协议的示例。为说明这个协议如何确保一致性，考虑一个写操作后跟随着另一个处理器的读操作的情景：因为这个写操作需要独占性访问，所以发出读操作的处理器 cache 中保存的任意副本均会失效。因此，当发出读操作时，会造成一次 cache 失效，cache 被迫去获取该数据的最新副本。对写操作而言，我们要求写入处理器有独占访问权限，以防止任何其他处理器能够同时写入。如果两个处理器确实同时试图写入相同的数据，那么只有其中一个处理器能够赢得写权限，同时也会导致其他处理器的副本失效。其他处理器想要完成写入必须获得该数据的最新副本，这个最新副本必须包含已经被更新的值。因此，该协议还强制实现了写操作串行化。

处理器活动	总线活动	CPU A cache内容	CPU B cache内容	存储器位置X的内容
				0
CPU A读X	X在cache中失效	0		0
CPU B读X	X在cache中失效	0	0	0
CPU A向X写入1	令X无效	1		0
CPU B读X	X在cache中失效	1	1	1

图 5-41　一个使用写回 cache 的在单个 cache 块 X 上工作的监听总线的失效协议示例。我们假设最初两个处理器的 cache 中都不包含该变量，且 X 的值为 0。CPU 和存储器中的内容显示在处理器和总线活动都完成之后的值。空白表示没有活动或者没有 cache 副本。当 CPU B 发生第二次缓存失效时，CPU A 回应，同时取消来自存储器的响应。随后，B 中 cache 的内容和存储器中 X 中的内容同时被更新。这种当块共享时更新存储器的设定简化了协议，但只有在块被替换时才可以追踪所有权并强制写回。这需要引入一个被称为“所有者”的附加状态，该状态表示块可以被共享，但是所有者处理器负责在更改块或替换块时更新其他处理器和存储器

硬件/软件接口　一种观点认为块大小在 cache 一致性中起重要作用。以在一个块大小为 8 字的 cache 上监听为例，两个处理器可以对一个字交替进行读写。大多数协议在处理器之间交换完整块，从而增加实现一致性的带宽需求。

错误共享：当两个不相关的共享变量位于同一个缓存块中时，即使处理器正在访问不同的变量，也会在处理器之间交换完整的块。

大的块还会导致所谓的错误共享：当两个不相关的共享变量位于同一个 cache 块中时，即使处理器正在访问的是不同的变量，处理器之间交换的也会是一整个块。程序员和编译器应该仔细放置数据以避免错误共享。

详细阐述　尽管 5.10 节开始处讲的三个属性已经足以确保一致性，但是被写入的值何时可以被看到，这个问题同样很重要。来看看这是为什么。注意在图 5-40 中我们不能要求对 X 的读取能够立刻看到其他处理器写入 X 的值。例如，如果一个处理器对 X 的写入仅仅比另一个处理器对 X 的读取早了很短的时间，那很可能无法确保读取操作可以返回这次写入的值，因为写入数据在被读取的那一刻甚至可能还没有离开处理器。这个写入值何时可以被其他读取操作看到的问题由存储器一致性模型详细定义。

我们做出以下两个假设。第一，直到所有处理器都可以看到写入的结果时写操作才算完成。(在没有完成时不允许下一次写操作发生。) 第二，相对于其他存储器访问操作处理器不会改变任何写操作的顺序。这两个条件意味着如果处理器在写入位置 X 后又写入了位置 Y，则任何看到 Y 的新值的处理器都必须能看到 X 的新值。这些假设允许处理器对读操作进行重排序，但是强制处理器按照程序顺序完成写入。

详细阐述　因为输入都是先改变 cache 后才更新存储器，并且在写回 cache 中更新操作需要最新的值。所以犹如多处理器之间的 cache 一样，单处理器的 cache 与 I/O 之间也存在一致性问题。多处理器和 I/O 的 cache 一致性问题（见第 6 章）虽然起源相似，但它们具有不同的特性，也影响了相应的解决方案。与很少拥有多个数据副本的 I/O 不同——应该尽可能地避免这种情况发生——在多个处理器上运行的程序通常会在多个 cache 中拥有相同数据的副本。

详细阐述　除了共享块状态呈分布式的缓存一致性监听协议之外，基于目录的 cache 一

致性协议还将物理存储块的共享状态保存在一个称为目录的位置。基于目录的一致性协议实现开销比监听略高，但它可以减少 cache 之间的交互，并因此可以扩展到更多的处理器数量。

5.11 并行与存储层次结构：廉价磁盘冗余阵列

本章节为网络章节，描述了如何使用多个磁盘来提供更高的吞吐量，这是廉价磁盘冗余阵列（RAID）的最初灵感。然而，RAID 真正普及的原因是采用适当数量的冗余磁盘所带来的更大可靠性。本节介绍不同 RAID 级别之间的性能、成本和可靠性等方面的差异。

5.12 高级专题：实现缓存控制器

本章节为网络章节，展示了如何实现对缓存的控制，就像在第 4 章中实现对单周期、流水线数据通路的控制一样。本节首先介绍了有限状态自动机以及在简单数据缓存中实现缓存控制器，包括用硬件描述语言中描述 cache 控制器。之后详细介绍了一个缓存一致性协议的示例以及实现此类协议的难点。

5.13 实例：ARM Cortex-A53 和 Intel Core i7 的存储层次结构

本节介绍第 4 章中描述的两种微处理器 ARM Cortex-A53 和 Intel Core i7 的存储器层次结构。本节内容基于《计算机体系结构：量化研究方法》(第 5 版）中的 2.6 节。

图 5-42 总结了这两个处理器的地址尺寸和 TLB。需要注意的是 Cortex-A53 具有两个 10 个表项全相联的微 TLB，它们由共享的 512 个表项四路组相联的主 TLB 以及一个 48 位虚拟地址空间和一个 40 位物理地址空间支持。Core i7 有三个具有 48 位虚拟地址和 44 位物理地址的 TLB。虽然这些处理器的 64 位寄存器可以支持更大的虚拟地址空间，但是没有软件需要这么大的空间，而 48 位虚拟地址可以缩小页表内存占用，还可以简化 TLB 硬件。

特性	ARM Cortex-A53	Intel Core i7
虚拟地址	48位	48位
物理地址	40位	44位
页表大小	可变：4、16、64KiB，1、2MiB，1GiB	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，10个表项，轮转替换（RR）策略 64个表项，四路组相联的主TLB 硬件处理TLB缺失	1个指令TLB和1个数据TLB 两个TLB均为四路组相联，LRU替换策略 L1 I-TLB对于小尺寸页面有128个表项，每个线程对大页面有7个表项 L1 D-TLB对于小尺寸页面有64个表项，对于大页面有32个表项 L2 TLB四路组相联，LRU替换策略 L2 TLB有512个表项 硬件处理TLB缺失

图 5-42 ARM Cortex-A53 和 Intel Core i7 920 的地址转换和 TLB 硬件。两个处理器均支持大页，这些大页用于操作系统或映射帧缓冲区等内容。大页方案避免使用大量条目来映射始终存在的单个对象

图 5-43 展示了它们的缓存。Cortex-A53 拥有 1 ～ 4 个处理器（或核），而 Core i7 则固定有 4 个。Cortex-A53 每个核都有一个 16 ～ 64KiB 二路组相联的 L1 指令缓存，而 Core i7 有

一个 32KiB 四路组相联的 L1 指令缓存。cache 块大小为 64 字节。Cortex-A53 将数据缓存的关联度增加至四路，其他变量保持不变。类似的，Core i7 除了将相联度增加至八路外保持其他变量一致。Core i7 为每个核提供一个 256KiB 八路组相联的统一 L2 缓存，块大小为 64 字节。相比之下，Cortex-A53 提供了一个在 1 ～ 4 个核之间共享的 L2 缓存。该缓存为十六路组相联，块大小为 64 字节，大小在 128KiB ～ 2MiB 之间。由于 Core i7 用于服务器，它还提供了由片上所有核共享的 L3 缓存。其大小取决于核的数量。在有四个核的情况下，它的大小为 8MiB。

特性	ARM Cortex-A53	Intel Core i7
L1 cache组织	指令数据分离cache	指令数据分离cache
L1 cache容量	指令/数据cache可配置为16～64KiB	每核的指令/数据cache均为32KiB
L1 cache相联度	两路（I）、四路（D）组相联	四路（I）、八路（D）组相联
L1替换策略	随机	类似于LRU
L1块大小	64字节	64字节
L1写策略	写返回，可变分配策略 （默认为写分配）	写返回，写不分配
L1命中时间（load）	2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～2MiB	256KiB（0.25MiB）
L2 cache相联度	十六路组相联	八路组相联
L2替换策略	类似于LRU	类似于LRU
L2块大小	64字节	64字节
L2写策略	写返回，写分配	写返回，写分配
L2命中时间	12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	—	十六路组相联
L3替换策略	—	类似于LRU
L3块大小	—	64字节
L3写策略	—	写返回，写分配
L3命中时间	—	35个时钟周期

图 5-43　ARM Cortex-A53 和 Intel Core i7 920 的 cache

缓存设计人员面临的一个重大挑战是支持像 Cortex-A53 和 Core i7 这样的可以在每个时钟周期执行一条以上的访存指令的处理器。一种通常的做法是将缓存分成多个 bank 并在访问不同 bank 的前提下允许多个独立的并行访问。这种技术类似于 DRAM bank 交错式访问（参见 5.2 节）。

非阻塞 cache：允许处理器在处理前面的 cache 缺失时仍可以访问 cache。

Cortex-A53 和 Core i7 采用了额外的优化技术，从而可以减少缺失代价。其中第一项是在缺失时先返回关键字。处理器还会在缓存缺失期间继续执行访问数据缓存的指令。设计人员通常使用一种被称为**非阻塞 cache** 的技术来试图隐藏缓存缺失延迟。缺失下命中（hit under miss）允许在缺失期间有其他的缓存命中，缺失下缺失（miss under miss）允许发

生多个未完成的缓存缺失。第一种技术的目的是通过其他工作来隐藏缺失延迟，第二种技术的目的是重叠两个不同缺失的延迟。

重叠多个未完成缺失的大部分缺失时间需要一个能并行处理多个缺失的高带宽存储系统。个人移动设备的存储系统通常可以流水化、合并、重排或优先化请求。大型服务器和多处理器通常具有能够并行处理多个未完成缺失的存储器系统。

Cortex-A53 和 Core i7 采用了数据访问的预取机制。它们会查看数据缺失的模式，使用这些信息来尝试预测下一个数据访问的地址，在缺失发生之前就从预测的地址开始获取数据。这种技术在循环访问数组时通常具有很好的效果。

这些芯片的复杂存储系统和大部分专用于缓存和 TLB 的模块体现出为了缩小处理器时钟周期和存储器延迟之间的差距所花费的大量努力。

Cortex-A53 和 Core i7 存储器层次结构的性能

测试 Cortex-A53 的存储器层次结构时使用了一个 32KiB 二路组相联的 L1 指令缓存，一个 32KiB 四路组相联的 L1 数据缓存，还有一个 1MiB 十六路组相联的 L2 缓存，运行在 SPEC2006 整数基准测试上。

Cortex-A53 在这些基准测试上的指令缓存缺失率非常小。图 5-44 是 Cortex-A53 的数据缓存结果，其中标注了明显的 L1 和 L2 的缺失率。L1 数据缓存缺失率在 0.5% 至 37.3% 之间，平均值为 6.4%，中位数为 2.4%。全局 L2 缓存的缺失率在 0.1% 至 9.0% 之间，平均值为 1.3%，中位数为 0.3%。1GHz 的 Cortex-A53 上的 L1 缺失代价为 12 个时钟周期，而 L2 的缺失代价为 124 个时钟周期。图 5-45 显示了在这种缺失代价下每次数据访问的平均缺失代价。当这些低缺失率乘以高缺失代价时，可以看到它们在 12 个 SPEC2006 程序的 5 个中都占据了很大一部分。

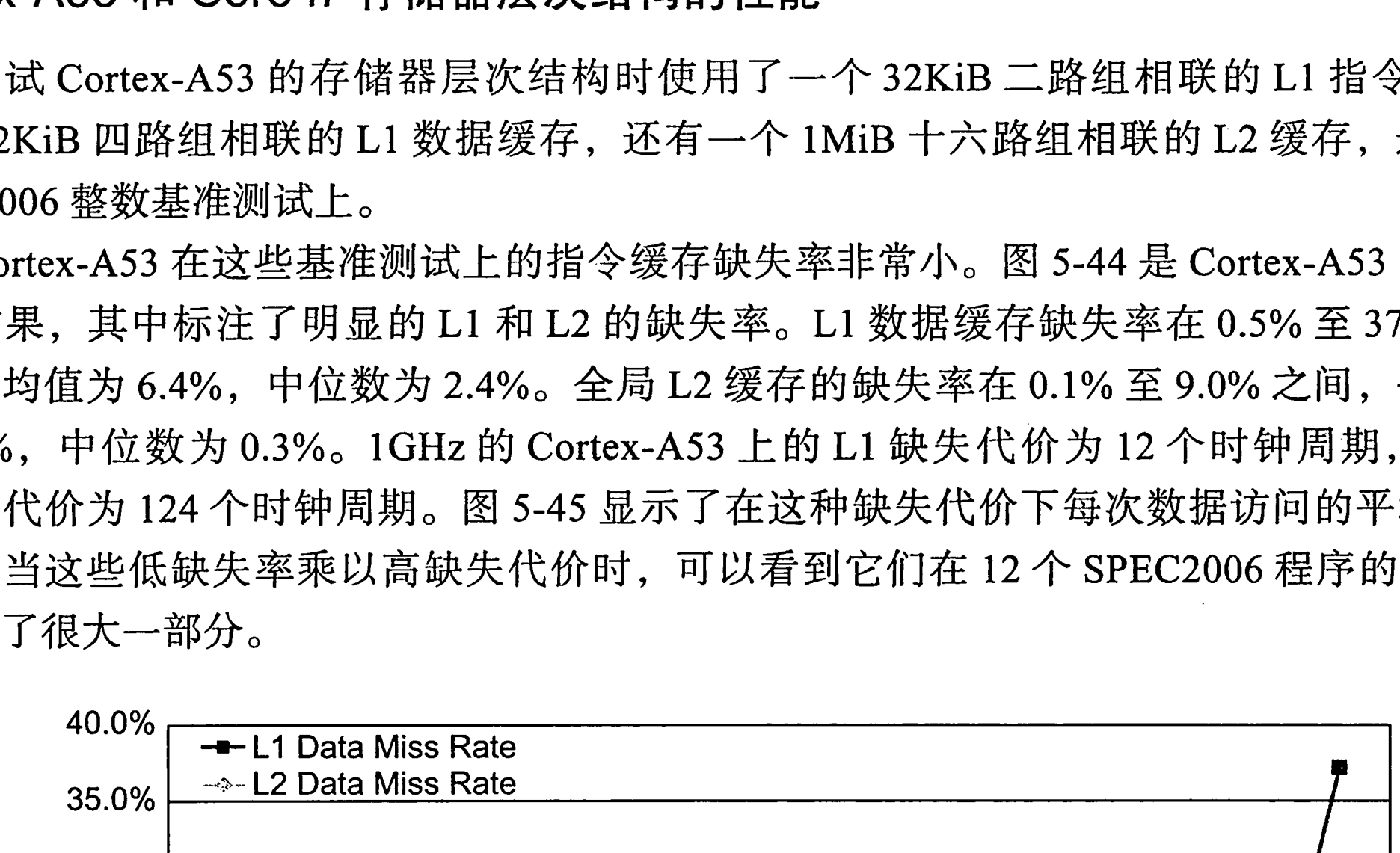


图 5-44 运行 SPEC2006 整数基准测试时 ARM Cortex-A53 的数据缓存缺失率。内存占用率较大的应用程序在 L1 和 L2 中的缺失率往往较高。需要注意的是 L2 中的缺失率为全局缺失率，也就是包含那些在 L1 中命中的访问。mcf 是缓存不友好的程序。注意，该图与图 4-74 使用相同的系统和基准测试程序

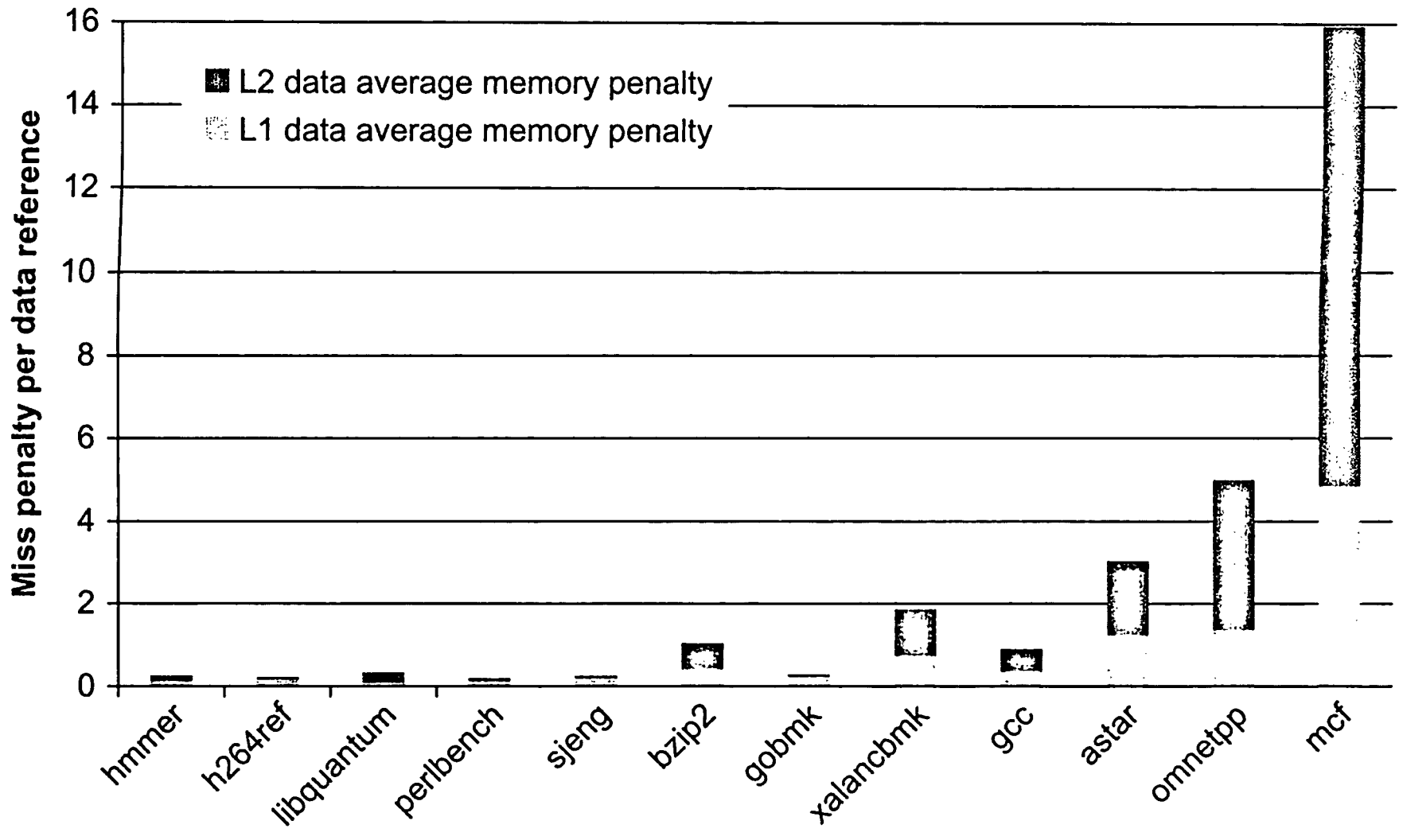


图 5-45 ARM 处理器运行 SPEC2006 整数基准测试时 L1 和 L2 的平均存储器访问代价（以时钟周期计）。尽管 L1 的缺失率比较高，但是 L2 的缺失代价高了 5 倍以上，这意味着 L2 缺失对性能的影响非常大

图 5-46 显示了在 SPEC2006 基准测试下 Core i7 的缓存缺失率。L1 指令缓存的缺失率在 0.1% 至 1.8% 之间，平均值仅比 0.4% 多一点。该缺失率与 SPEC2006 基准测试在其他研究中的缺失率一致。L1 数据缓存的缺失率在 5% 至 10% 之间，有时会变得更高。L2 和 L3 缓存的重要性是显而易见的。L2 的缺失代价在 100 个时钟周期以上，并且缺失率为 4%，因此 L3 缓存非常关键。假设一半的指令是 load 或 store 指令，如果没有 L3 缓存，那么 L2 缓存的缺失会导致 CPI 增加 2！相比之下，虽然 L3 缓存 1% 的缺失率依然很高，但是比 L2 的缺失率低了 4 倍，比 L1 的缺失率低了 6 倍。

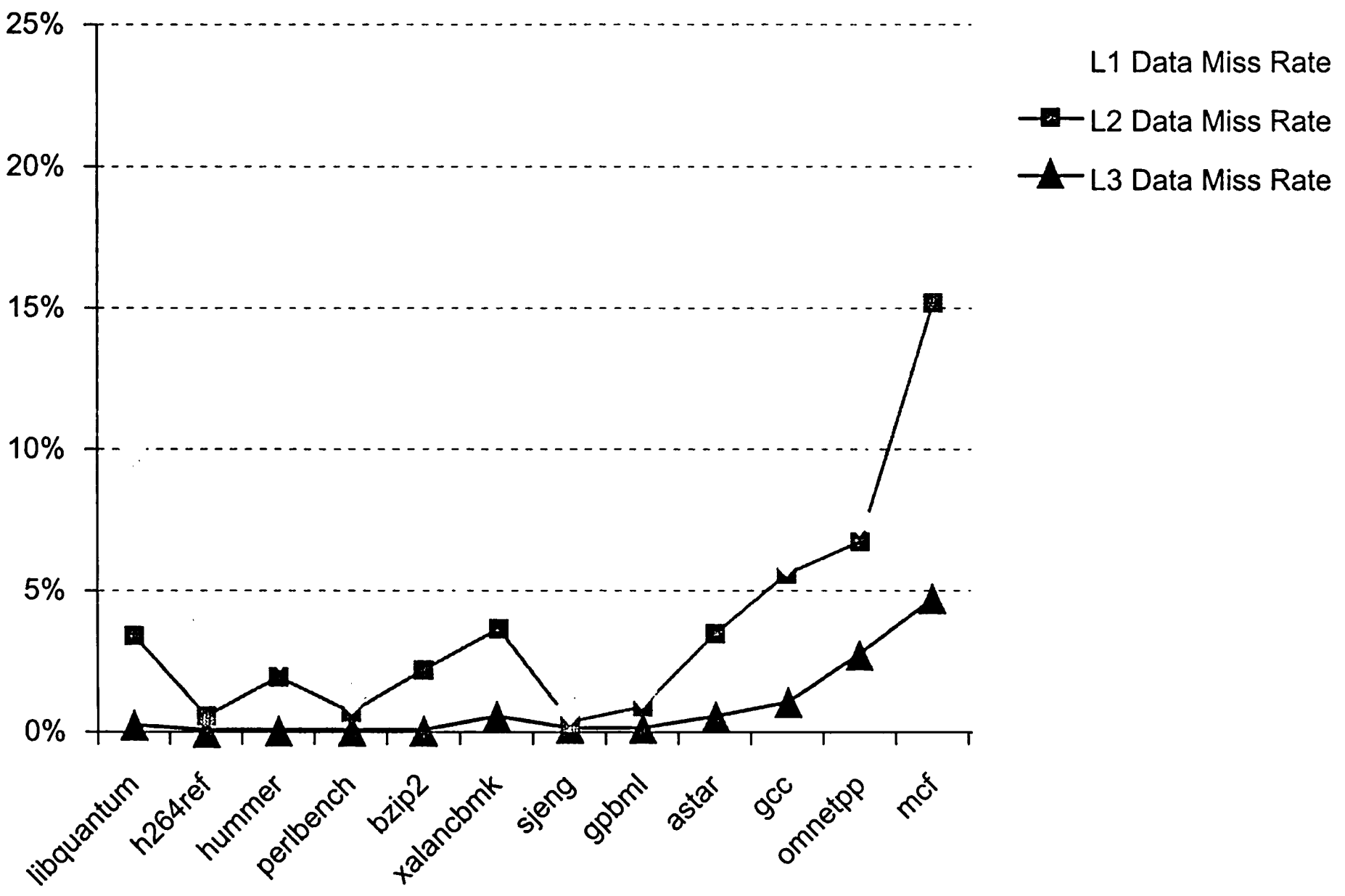


图 5-46 Intel Core i7 920 运行 SPEC2006 整数基准测试时的 L1、L2 和 L3 数据缓存缺失率

详细阐述 由于推测执行有可能是错误的（见第 4 章），所以对 L1 数据缓存进行的访问不对应于最终完成的 load 和 store 指令，有一些对 L1 数据缓存进行访问的 load 和 store 指令最终没有执行。图 5-44 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比测量的缺失率要高 1.6 倍（L1 缓存的平均缺失率为 9.5% 而不是 5.9%）。

5.14 实例：RISC-V 系统的其他部分和特殊指令

图 5-47 列出了剩余的 13 条 RISC-V 指令，它们用于特殊用途和系统类。

栅栏指令为指令（fence.i）、数据（fence）和地址转换（sfence.vma）提供同步栅栏。第一条指令 fence.i 通知处理器软件已修改指令存储器，以保证获取指令时可以得到更新后的指令。第二条指令 fence 影响多处理器和 I/O 的数据存储器访问顺序。第三条指令 sfence.vma 通知处理器软件已修改页表，以保证地址转换时可以得到更新后的数据。

六个控制和状态寄存器（CSR）访问指令在通用寄存器和 CSR 之间移动数据。csrrwi（CSR 读 / 写立即数）指令将 CSR 复制到整数寄存器，然后用立即数覆盖 CSR。csrrsi（CSR 读取 / 设置立即数）指令将 CSR 复制到整数寄存器，然后用 CSR 的按位 OR 和立即数覆盖 CSR。csrrci（CSR 读取 / 清除）指令与 csrrsi 指令类似，但是清除位而不是设置它们。csrrw、csrrs 和 csrrc 指令与上述三条指令的功能类似，区别是使用寄存器操作数而非立即数。

类型	助记符	名称
存储器顺序	fence.i	指令栅栏
	fence	栅栏
	sfence.vma	地址转换栅栏
CSR访问	scrrwi	CSR读/写立即数
	scrrsi	CSR读/设置立即数
	scrrci	CSR读/清除立即数
	scrrw	CSR读/写
	scrrs	CSR读/设置
	scrrc	CSR读/清除
系统	ecall	环境调用例外
	ebreak	环境断点例外
	sret	管理态例外返回
	wfi	等待中断

图 5-47 完整 RISC-V 指令集中系统和特殊操作的汇编语言指令列表

有两个指令的唯一目的是生成例外：ecall 生成环境调用例外以调用操作系统，ebreak 生成断点例外以调用调试器。管理态例外返回指令（sret）允许程序从例外处理程序返回。

最后，等待中断指令（wfi）通知处理器它将进入空闲状态，一直持续到中断发生。

5.15 加速：cache 分块和矩阵乘法

除了第 3 章和第 4 章中的子字并行和指令级并行技术外，通过定制底层硬件来提高 DGEMM 性能的一系列操作的下一步是为优化添加 cache 分块技术。图 5-48 显示了图 4-78 中 DGEMM 的添加 cache 分块后的版本。这些变化与之前从图 3-22 中未经优化的 DGEMM 到图 5-21 中分块 DGEMM 的变化相同。这次我们从第 4 章中取出 DGEMM 的展开版本，并在 A、B 和 C 的子矩阵上多次调用它。实际上，除了第 7 行中的循环增量增大了之外，图 5-48 中的第 28 ～ 34 行和第 7 ～ 8 行与图 5-21 中的第 14 ～ 20 行和第 5 ～ 6 行一一对应。

与前面的章节不同，本章没有给出生成的 x86 代码，因为分块并不会影响计算，只会影响访问内存中数据的顺序，所有内部循环代码与图 4-79 几乎完全相同，不同点在于循环语

句的循环边界助记符发生了改变（由 n 变为 si+blocksize）。图 4-78 中内循环前有 14 条指令，内循环后有 8 条指令，在图 5-48 中，因为循环边界助记符发生了改变，这些指令的数量被扩展到内循环前有 40 条指令，内循环后有 28 条指令。虽然指令条数增加了，但是相比于 cache 失效率减少对性能提升做出的贡献，执行这些额外指令的开销就显得微不足道了。图 5-49 对比了使用未优化技术的性能和采用子字并行、指令级并行和 cache 分块优化技术后的性能。对于较大的矩阵来说，cache 分块循环展开的 AVX 代码提升了 2 ～ 2.5 倍的性能。当我们将未优化的代码与使用所有这三种技术优化后的代码进行比较时，性能的改进是 8 ～ 15 倍，其中最大的矩阵得到的性能提升也是最大的。

详细阐述 正如 3.9 节中的详细阐述所述，这些结果是在 Turbo 模式关闭的情况下得到的。如第 3 章和第 4 章所述，如果打开 Turbo 模式，就会通过临时增加 3.3/2.6=1.27 倍时钟速率的方式来改善所有结果。这种情况下 Turbo 模式的运行效果非常好，因为它仅使用了八核芯片中的一个核。但是如果想要运行得更快，就应该使用所有的内核，在第 6 章中将会介绍这种情况。

```
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
23      for ( int x = 0; x < UNROLL; x++ )
24        _mm256_store_pd(C+i+x*4+j*n, c[x]);
          /* 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　图 4-78 中的 DGEMM 使用 cache 分块技术优化后的 C 语言版本。这些变化与图 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 4-79 中的几乎完全相同。再次强调，调用 do_block 函数没有任何开销，因为编译器会采用内联函数调用

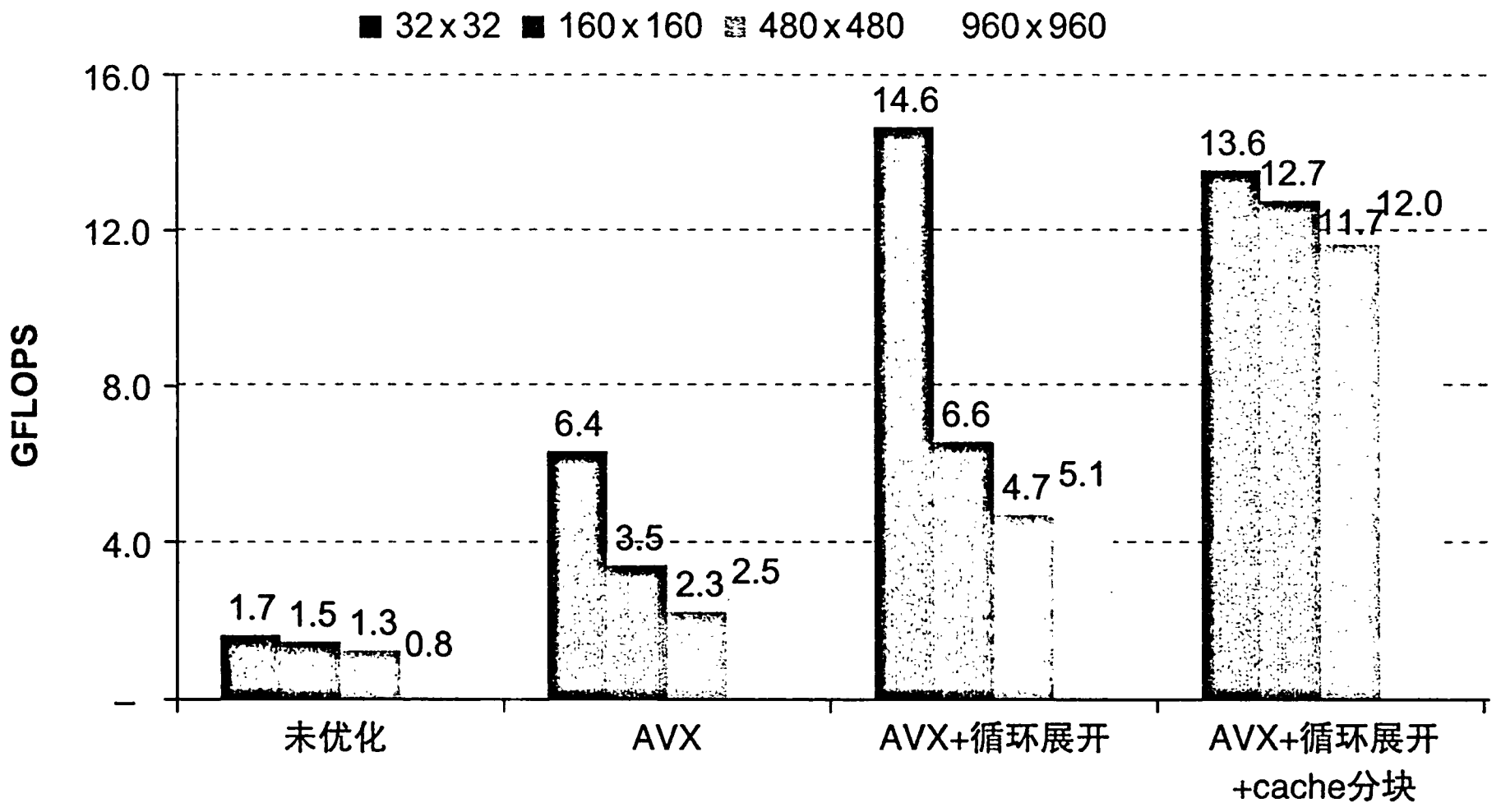


图 5-49 矩阵规模从 32 × 32 增加到 960 × 960 时四个版本的 DGEMM 的性能。最大矩阵完全优化代码的性能几乎是图 3-22 中未优化版本的 15 倍

5.16 谬误与陷阱

作为计算机体系结构中的定量原则，存储器层次结构似乎不容易受到谬误和陷阱的影响。但实际上却大相径庭，很多人不仅已经有了很多的谬误，还遭遇了陷阱，而且其中的一些还导致了很多负面的结果。下面从学生在练习和考试中经常遇到的陷阱开始讲解。

陷阱：在写程序或编译器生成代码时忽略存储系统的行为。

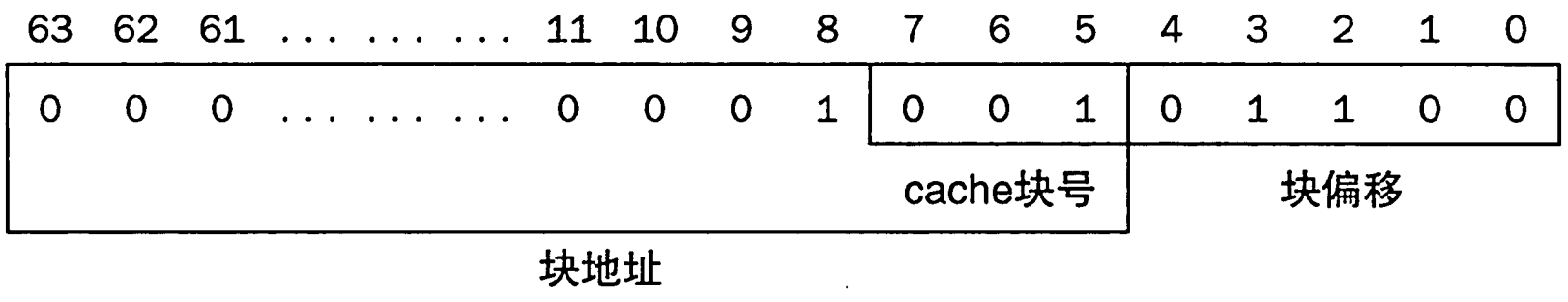
这可以很容易地写成一个谬误："在写代码时，程序员可以忽略存储器层次。"图 5-19 中的排序和 5.13 节的 cache 分块技术证明了如果程序员在设计算法时考虑存储系统的行为，则很容易将性能翻倍。

陷阱：在模拟 cache 的时候，忘记说明字节编址或者 cache 块大小。

（手动或者通过计算机）模拟 cache 的时候，我们必须保证，在确定一个给定的地址被映射到哪个 cahce 块中时，一定要说明字节编址和多字块的影响。例如，如果我们有一个容量为 32 字节的直接映射 cache，块大小为 4 字节，则字节地址 36 映射到 cache 的块 1，因为字节地址 36 是块地址 9，而 9 mod 8 = 1。另一方面，如果地址 36 是字地址，那么它就会映射到块 36 mod 8 = 4。因此要保证清楚地说明基准地址。

同样，我们必须说明块的大小。假设我们有一个 256 字节大小的 cache，块大小为 32 字节，那么字节地址 300 将落入哪个块中？如果我们将地址 300 划分成字段，就可以看到

答案：



字节地址 300 是块地址

$$\left\lfloor \frac{300}{32} \right\rfloor = 9$$

cache 中的块数是

$$\left[\frac{256}{32}\right]=8$$

块号 9 对应于 cache 块号 9 mod 8 = 1。

许多人，包括作者（在早期的书稿中）和那些忘记自己预期的地址是字、字节或块号的教师，都犯过这个错误。当你做练习时一定要注意这个易犯的错误。

陷阱：对于共享 cache，组相联度少于核的数量或者共享该 cache 的线程数。

如果不特别注意，一个运行在 2^n 个处理器或者线程上的并行程序为数据结构分配的地址可能映射到共享二级 cache 的同一个组中。如果 cache 至少是 2^n 路组相联，那么通过硬件可以隐藏这些程序偶尔发生的冲突。如果不是，程序员可能要面对明显不可思议的性能缺陷——事实上是由于二级 cache 冲突缺失引起的——在程序迁移时发生，假定从一个 16 核的机器迁移到一个 32 核的机器上，并且如果它们都使用十六路组相联的二级 cache。

陷阱：用存储器平均访问时间来评估乱序处理器的存储器层次结构。

如果处理器在 cache 缺失时阻塞，那么你可以分别计算存储器阻塞时间和处理器执行时间，因此可以使用存储器平均访问时间来独立地评估存储器层次结构（见 5.4 节）。

如果处理器在 cache 缺失时继续执行指令，而且甚至可能维持更多的 cache 缺失，那么唯一可以用来准确评估存储器层次结构的办法是模拟乱序处理器和存储器结构。

陷阱：通过在未分段地址空间的顶部增加段来扩展地址空间。

从 20 世纪 70 年代开始，许多程序都变得很大，以至于不是所有的代码和数据都能仅用 16 位地址寻址。于是，计算机修改为 32 位地址，一种方法是直接使用未分段的 32 位地址空间（也称为平面地址空间），另一种方法是给已经存在的 16 位地址再增加 16 位长度的段。从市场观点来看，增加程序员可见的段，并且迫使程序员和编译器将程序划分成段，这样可以解决寻址问题。但遗憾的是，任何时候，一种程序设计语言要求的地址大于一个段的范围就会有麻烦，比如说大数组的索引、无限制的指针或者是引用参数。此外，增加段可以将每个地址变成两个字——一个是段号，另一个是段内偏移——这些在使用寄存器中的地址时就会出现问题。

谬误：实际的磁盘故障率和规格书中声明的一致。

最近的两项研究评估了大量磁盘，目的是检查实际结果和规格之间的关系。其中一项研究了将近 100 000 个磁盘，它们标称其 MTTF 为 1 000 000 ～ 1 500 000 小时或者说具有 0.6%~0.8% 的 AFR。他们发现 2%~4% 的 AFR 是常见的，通常比设定的故障率高 3~5 倍 [Schroeder and Gibson, 2007]。另一项研究了 100 000 个磁盘，这些磁盘标称具有 1.5% 的 AFR，但是在第一年中，磁盘故障率为 1.7%，到第三年，磁盘的故障率上升到 8.6%，也就是说，大约是规格书中指定的故障率的 6 倍之多 [Pinheiro, Weber, and Barroso, 2007]。

谬误：操作系统是调度磁盘访问的最好地方。

正如 5.2 节中提到的那样，高层磁盘接口为宿主操作系统提供逻辑块地址。假设在这样的高层抽象中操作系统可以通过将逻辑块的地址按照递增的顺序排序以获得最好的性能。然而，由于磁盘知道逻辑地址被映射到扇区、磁道上以及磁面上的实际物理地址，这样通过调度就可以减少旋转以及寻道的时间。

例如，假设以下工作负载是 4 个读操作 [Anderson, 2003]：

操作	LBA起始地址	长度
读	724	8
读	100	16
读	9987	1
读	26	128

宿主操作系统可能对 4 个读操作重新进行调度，编排成逻辑块的读操作的顺序：

操作	LBA起始地址	长度
读	26	128
读	100	16
读	724	8
读	9987	1

依赖于数据在磁盘中的相对位置，如图 5-50 所示，重新编排 I/O 顺序可能会使情况变得更糟。磁盘调度的读操作在磁盘的 3/4 旋转周期就全部完成，而操作系统调度的读操作花费了 3 个旋转周期。

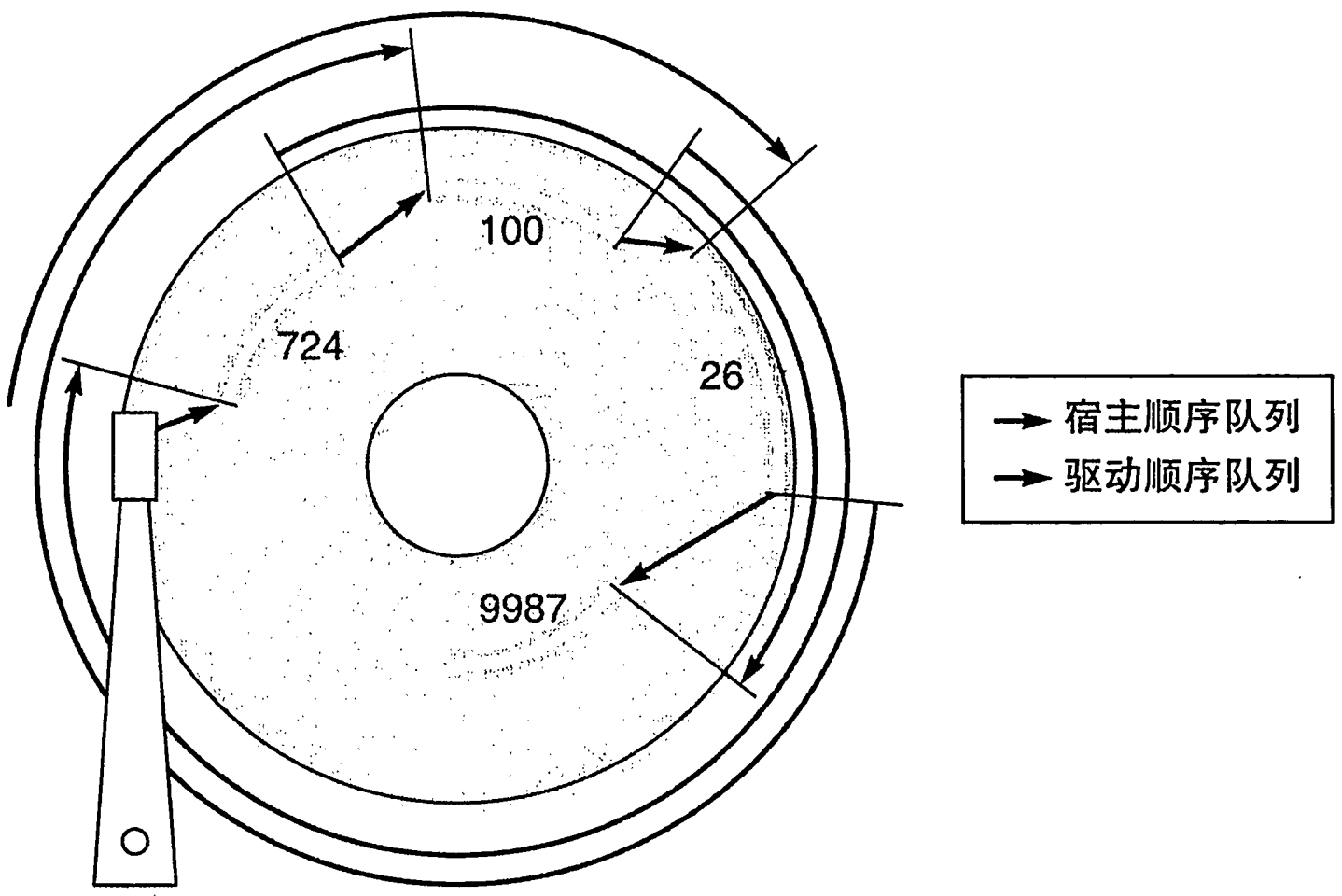


图 5-50　操作系统调度与磁盘调度访问的例子，标记为宿主顺序和驱动顺序。前者完成 4 个读操作需要 3 个旋转周期，而后者完成 4 个读操作仅仅在 3/4 旋转周期即可完成（资料来源：Anderson [2003]）

陷阱：在不为虚拟化设计的指令系统体系结构上实现虚拟机监视器。

在 20 世纪 70 年代和 80 年代，很多计算机体系结构设计者并没有刻意保证所有读写相关的硬件资源指令都是特权指令。这种放任的态度导致了 VMM 在这些体系结构上存在问题，包括 x86，这里我们就以它为例。

图 5-51 指出了虚拟化产生问题的 18 条指令 [Robin and Irvine, 2000]。其中两大类指令是：

- 在用户模式下读控制寄存器，暴露了在虚拟机上运行的客户操作系统（如前面提到的 POPF）。
- 检查分段的体系结构所需的保护，但却假设操作系统在最高的特权级运行。

为了简化在 x86 上实现 VMM，AMD 和 Intel 都提出通过新的模式扩展体系结构。Intel 的 VT-x 为虚拟机运行提供了一个新的执行模式，一个面向虚拟机状态的体系结构定义，快读虚拟机切换指令，以及一大组用来选择调入 VMM 环境的参数。总之，VT-x 在 x86 中加

了 11 条新指令。AMD 的 Pacifica 做了相似的改进。

问题种类	x86的问题指令
当运行在用户模式时，访问敏感寄存器无内陷中断	存储全局描述符表寄存器（SGDT） 存储局部描述符表寄存器（SLDT） 存储中断描述符表寄存器（SIDT） 存储机器状态字（SMSW） 标志入栈（PUSHF，PUSHFD） 标志出栈（POPF，POPFD）
在用户模式下访问虚拟存储机制时，x86保护检查指令失效	从段描述符读取访问权限（LAR） 从段描述符读取段的边界（LSL） 如果段描述符可读，进行读校验（VERR） 如果段描述符可写，进行写校验（VERW） 段寄存器出栈（POP CS，POP SS，…） 段寄存器入栈（PUSH CS，PUSH SS，…） 远调用不同的特权级（CALL） 远返回至不同的特权级（RET） 远跳转至不同的特权级（JMP） 软中断（INT） 存储段选择寄存器（STR） 移入/移出段寄存器（MOVE）

图 5-51 虚拟化产生问题的 18 条 x86 指令的概述 [Robin and Irvine, 2000]。上面一组的前 5 条指令允许程序在用户模式下读控制寄存器，而无须内陷中断，例如描述符表寄存器。标记出栈指令会修改包含敏感信息的控制寄存器，但在用户模式下将失效而无任何提示。x86 体系结构中段的保护检查在下面的一组指令中，当读取控制寄存器时，作为指令执行的一部分，都会隐式地检查特权级。进行检查时操作系统必须运行在最高特权级，但是对客户虚拟机并没有这样的要求。只有在移入段寄存器操作时会试图修改控制状态，但是，保护检查同样会组织它这么做

另一种方法通过修改硬件来对操作系统做细微的修改以简化虚拟化。这种技术称为泛虚拟化（paravirtualization），例如开源的虚拟机监视器 Xen 就是一个很好的例子。Xen 虚拟机监视器提供给客户操作系统一个抽象虚拟机，它仅仅使用了供虚拟机监视器运行的 x86 物理硬件中易于虚拟化的一部分。

5.17 本章小结

无论在最快的计算机还是最慢的计算机中，构成主存的原材料——DRAM 本质是相同的，并且是最便宜的，这使得构建一个和快速处理器保持同步的存储系统变得更加困难。

局部性原理可以用来克服存储器访问的长延迟，这个策略的正确性已经在存储器层次结构的各级都得到了证明。尽管层次结构中的各级从量的角度来看非常不同，但是在它们的执行过程中都遵循相似的策略，并且利用相同的局部性原理。

多级 cache 可以更方便地使用更多的优化，这有两个原因。第一，较低级 cache 的设计参数与一级 cahce 不同。例如，由于较低级 cache 的容量一般很大，因此可能使用更大容量的块。第二，较低级 cache 并不像一级 cache 那样经常被处理器用到。这让我们考虑当较低级 cache 空闲时让它做一些事情以预防将来的缺失。

另一个趋势是寻求软件的帮助。使用大量的程序转换和硬件设备有效地管理存储器层次结构是增强编译器作用的主要焦点。现在有两种不同的观点。一种是重新组织程序结构以增强它的空间和时间局部性。这种方法主要针对以大数组为主要数据结构的面向循环的程序，

大规模的线性代数问题就是一个典型的例子，例如 DGEMM。通过重新组织访问数组的循环增强了局部性，也因此改进了 cache 性能。

预取：使用特殊指令将未来可能用到的指定地址的 cache 块提前搬到 cache 中的一种技术。

还有一种方法是**预取**（prefetching）。在预取机制中，一个数据块在真正被访问之前就被预取入 cache 中了。许多微处理器使用硬件预取尝试预测访问，这对软件可能比较困难。

第三种方法是使用优化存储器传输的特殊 cache 感知（cache-aware）指令。例如，在 6.10 节中，微处理器使用了一种优化设计：当发生写缺失时，由于程序要写成整个块，因而并不从主存中取回一个块。对于一个内核来说，这种优化明显减少了存储器的传输。

我们将在第 6 章中看到，对并行处理器来说，存储系统也是一个重要的设计问题。存储器层次结构决定了系统性能的重要性在不断增长，这也意味着在未来的几年内，这一领域对设计者和研究者来说将成为焦点。

5.18 历史视角和拓展阅读

本节为网络章节，描述了存储器技术的概况，从汞延迟线到 DRAM，存储器层次结构的发明，保护机制以及虚拟机，最后以操作系统的简单发展历史作总结，包括 CTSS、MULTICS、UNIX、BSD UNIX、MS-DOS、Windows 和 Linux。

5.19 练习

在接下来的习题中，我们假设存储器为字节寻址，字的大小为 64 位，如有变化会特别说明。

5.1 本题研究矩阵计算的内存局部性特性。以下代码使用 C 编写，其中同一行中的元素是连续存储的。假设每个字是 64 位整数。

```
for (I=0; I<8; I++)
  for (J=0; J<8000; J++)
    A[I][J]=B[I][0]+A[J][I];
```

5.1.1 [5] <5.1> 在一个 16 字节的 cache 块中可以存放多少个 64 位整数?

5.1.2 [5] <5.1> 对哪个变量的访问显示了时间局部性?

5.1.3 [5] <5.1> 对哪个变量的访问显示了空间局部性?

局部性受引用顺序和数据分布的影响。相同的计算也可以使用 Matlab 写出，与 C 的不同是在 Matlab 中相同列中的矩阵元素在存储器中的存储是连续的。

```
for I=1:8
  for J=1:8000
    A(I,J)=B(I,0)+A(J,I);
  end
end
```

5.1.4 [5] <5.1> 对哪个变量的访问显示了时间局部性?

5.1.5 [5] <5.1> 对哪个变量的访问显示了空间局部性?

5.1.6 [15] <5.1> 需要多少个 16 字节 cache 块来存储使用 Matlab 矩阵存储的所有将被访问的 64 位矩阵元素？如果是使用 C 的矩阵存储呢?（假设每行包含多个元素。）

5.2 cache 对于为处理器提供高性能存储器层次结构非常重要。下面是 64 位存储器地址访问顺序列表，以字地址的形式给出。

```
0x03, 0xb4, 0x2b, 0x02, 0xbf, 0x58, 0xbe, 0x0e, 0xb5,
0x2c, 0xba, 0xfd
```