第 5 章　共享存储的多处理器

并行体系结构最常见的形式是中小规模的多处理器，它提供全局物理地址空间，可以从任何处理器对等地访问全部主存。这种体系结构通常称为对称多处理器（SMP）。每个处理器均有自己的高速缓存，所有的处理器和存储模块均连接在同一个互连设备上，此互连设备通常是一共享总线。SMP 机器在服务器市场上占据着主导地位，在桌面计算机市场上也日益流行，同时它们还是构成大规模系统的重要组件。由于对存储器和处理器等资源的有效共享，这些 SMP 机器有很强的吞吐能力，在同时处理多个对存储器与 CPU 需求各异的串行作业时表现不凡。这种机器的各个处理器可以使用普通的装入和存储命令高效地访问共享数据，还可以自动地在各自的高速缓存中移动和复制共享数据。这些特性对并行程序设计颇具吸引力。另外，这些特性对操作系统也相当有用，使操作系统的不同进程可以共享数据结构，但在不同的处理器上运行。

从图 5-1 所示的通信体系结构的层次观点来看，这种共享地址空间的程序设计模型可以直接由硬件支持。用户进程可以用共享虚拟地址进行读写操作。这些操作通过对共享物理地址的装入和存储指令来实现。事实上，这里的程序设计模型和硬件操作之间的关系是如此密切，以至于它们通常都被简单地称为“共享存储”。一个消息传递的程序设计模型可由一个软件层来实现——典型的例子是某个运行库。这一软件层将共享地址空间的大部分划分给每个进程，当作它们的私有地址空间来处理，而将另外一些共享地址空间按照进程间的消息缓冲区进行管理。通过在这些缓冲区中做数据拷贝的方式来实现发送和接收操作。由于共享缓冲区地址的翻译和保护由硬件提供，这部分工作无须涉及操作系统。出于可移植性的考虑，大部分消息传递程序设计接口均在常见的 SMP 上得到实现。事实上，只要对共享总线和存储的争用不成为瓶颈，这种实现对消息传递程序带来的性能往往优于传统的分布式存储消息传递系统。这在很大程度上是由于没有涉及到操作系统。操作系统仍然用于输入/输出和对多道程序设计的支持。

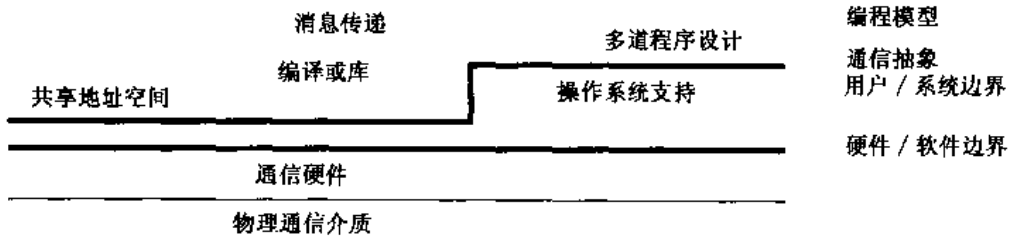


图 5-1　基于总线的 SMP 系统的通信体系结构抽象层次。共享地址空间由硬件直接支持，而消息传递由软件支持

由于所有的通信操作和本地计算均产生对共享地址空间中的存储访问，从系统设计者的观点来看，高层设计的关键问题在于对这种扩展存储层次的组织。通常多处理器的存储层次分为 4 类，如图 5-2 所示，这 4 类层次与可考虑到的多处理器规模大致对应。前 3 个对应的都是对称式多处理器（所有处理器到所有主存空间的访问时间都一样），而第 4 个不同。

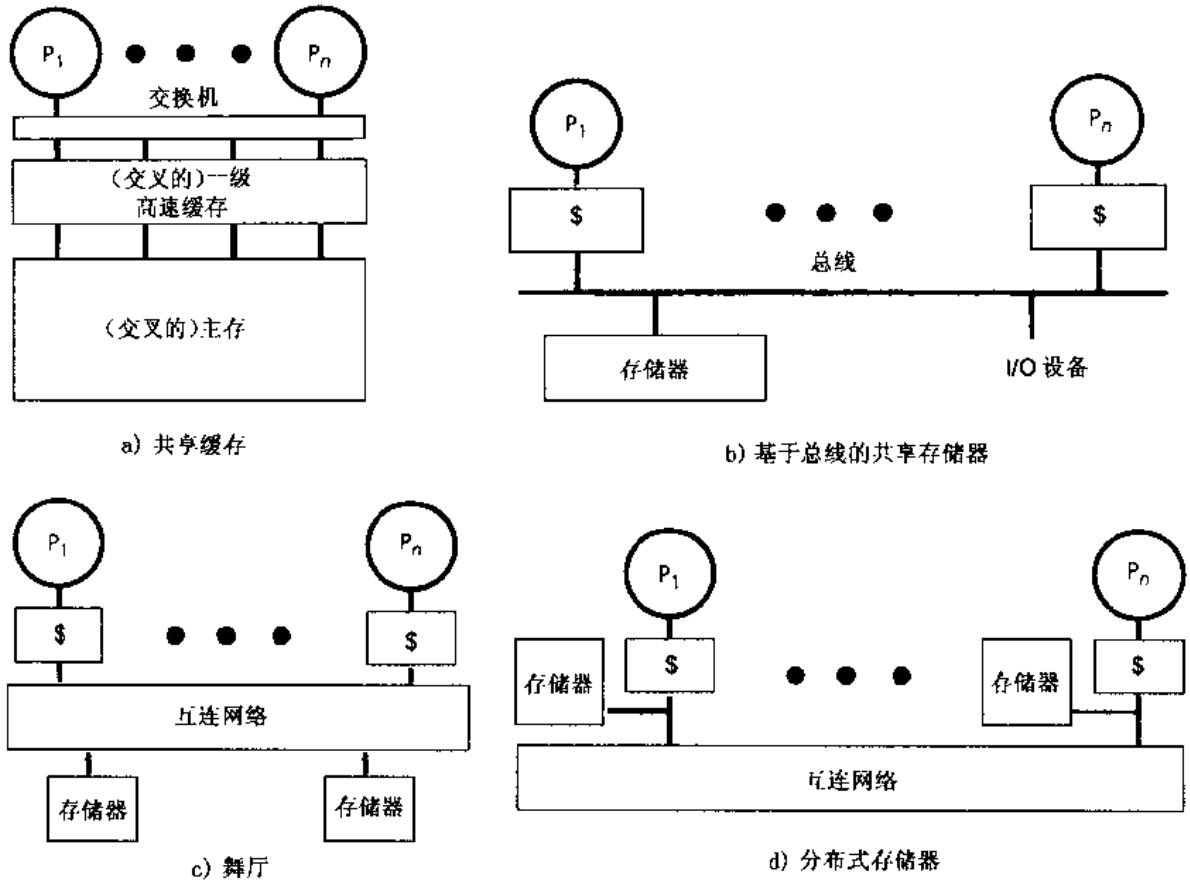


图 5-2　多处理器中常见的扩展存储层次结构

在共享高速缓存方式下（如图 5-2a 所示），互连设备位于处理器和共享的一级高速缓存之间，而此高速缓存连接到共享主存子系统上。高速缓存和主存系统都可以是按交叉方式组织的，从而可以增加有效带宽。这种方法适用于处理器数目较少的情况（2～8 个）。在 20 世纪 80 年代中期，在一块电路板上安放一对处理器是常见的。而现在，这可能在单片多处理器上实现，即几个处理器在同一块芯片上，共享芯片上的一级高速缓存。然而，这种策略仅适于很小的规模，因为处理器和它们共享的一级高速缓存之间的互连机构是决定高速缓存访问延迟的关键因素，还由于该高速缓存必须为多个处理器的同时访问提供很高的带宽。

在基于总线的共享存储方式下（如图 5-2b 所示），互连设备是共享总线，它位于处理器的私有高速缓存（或高速缓存的某种层次结构）和共享主存子系统之间。这种方式广泛用于中小规模的多处理器之上，典型的中小规模多处理器通常包括 20 或 30 个处理器。目前出售的并行计算机主要是这种形式。在现代的多处理器设计中，投入的大量设计工作都是为了支持“高速缓存一致性”的共享存储系统。例如，Intel 的 Pentium Pro 处理器可以直接挂在共享总线上，无须任何“粘接逻辑”。低成本的基于总线的机器常常使用这些处理器，极大地推广了基于总线的共享存储方式的使用。这种机器的扩展限制主要来自于共享总线和存储系统的带宽限制。

最后的两种方式可扩展到许多处理节点。“舞厅”（dancehall）方式也将互连设备放在高速缓存与主存之间，但这时的互连设备不一定只是总线，它可以是可扩展的端到端互连网络，同时存储器划分为许多逻辑模块，这些模块连接到互连设备中逻辑上不同的位置（如图

5-2c 所示），这种方式是对称的，即所有主存模块到所有处理器的距离相同。但这种方式也有局限性，即存储器到所有处理器的距离太远。特别是在规模较大的系统中，为了使得所有的处理器能访问所有的存储模块，必须经过互连网络中的若干“跳步”。第 4 种方式是分布存储方式，它不是对称性的。可扩展的互连设备位于处理节点之间，但每个节点均在全局存储中拥有自己的一部分本地存储，对这部分主存的访问速度更快（如图 5-2d 所示）。通过在分布的数据中发掘局部性，大部分高速缓存失效可以在本地存储中满足，无须跨越网络。这种设计对可扩展的多处理器最具吸引力。本书中后面的若干章节会集中于这一主题。当然，也可以将上述方式组合在一种机器设计中——例如，设计一台分布式存储的机器，其节点是基于总线的 SMP；或设计一台机器，其处理器共享的高速缓存不是一级高速缓存。

在种种情况下，高速缓存总是扮演着重要角色，对处理器而言，高速缓存可以减少平均数据访问时间，对共享的互连设备和存储系统而言，它可以减少每个处理器需要的通信带宽。带宽需求的减少是因为某个处理器提出的数据访问要求可在高速缓存中得到满足，无须使用互连设备。除了共享高速缓存方式之外，其他方式下每个处理器至少有一级私有的高速缓存，这带来了一个非常关键的问题——高速缓存一致性问题。当同一个存储块的副本出现在一个或多个处理器的高速缓存中时，就会产生这样一个问题：如果某个处理器写入并因而修改了此存储块，那么除非采用特殊的操作，否则其他处理器在访问此存储块时，实际上访问的还是自己高速缓存中保存的旧内容。

269 ~ 271

当前，大多数小规模的多处理器使用共享总线互连设备，每个处理器有自己的高速缓存，同时这些处理器共享集中式主存；而舞厅和共享高速缓存方式只是用在特定的条件下。随着技术的演化，具体的系统组织方式可能会发生变化。然而，基于总线的组织方式和分布式存储组织方式是当前最流行的方式，它们体现了解决高速缓存一致性问题的两种基本方法。根据互连设备本质特性的不同，一种方法适用的情况是：互连设备上的操作对所有的处理器均是可见的（类似总线）；在另一种情况下，互连设备不是集中式的，端到端的操作仅对于其参与者可见。本章主要讨论协议的逻辑设计，这种协议应能充分利用总线的基本属性解决高速缓存一致性的问题。与这些高速缓存一致性技术的硬件实现相关的设计问题在下一章介绍。可扩展的分布式存储多处理器的基本设计在第 7 章介绍，随后的第 8、9 章涵盖了特定于可扩展高速缓存一致性的问题。

5.1 节详细地描述了共享存储体系结构的高速缓存一致性问题，并给出一个最简单的例子，称为侦听式（snooping）高速缓存一致性协议。一致性不仅是一个关键性的硬件设计概念，而且是存储抽象的直观概念中不可或缺的部分。然而在存储行为方面，并行软件常常需要依赖比这种一致性更强的假设。5.2 节对第 1 章提出的序的概念作进一步讨论，引入存储同一性（consistency）的概念，这一概念定义了共享地址空间的语义。在计算机体系结构和编译器设计中，这一问题越来越重要；近来大多数指令集系统结构的参考手册中，有很大篇幅在介绍存储同一性模型。在有了存储抽象和相关的概念之后，5.3 节详细介绍了一些更实用的侦听协议，同时向读者展示这些协议如何满足高速缓存一致性条件和有用的存储同一性模型。协议的操作通过在逻辑状态变迁层次得以描述。对这一层次上的几个设计方面的权衡考虑有一个量化的评估，使用的评估技术在 5.4 节中介绍，这一技术涉及到第 4 章中谈到的工作负载驱动评估方法的若干要素。

本章后面的部分探讨具有高速缓存一致性的共享存储系统对于在其上面运行的软件的影

272 响。5.5 节讨论低层的同步操作如何利用具有高速缓存一致性的多处理器上可用的硬件原语以及如何改造锁和栅障算法，使机器能更好地得到利用。5.6 节讨论并行程序设计的影响，特别讨论了如何利用数据的时间和空间局部性，来减少高速缓存扑空和共享总线上的通信量。

5.1 高速缓存的一致性

直觉上，存储行为的模型应该是这样的：它提供一组保存数值的单元，当读取某个单元时，应返回写入此单元的最新值。这是存储系统抽象的基本属性。在串行程序中我们依赖于这一属性，利用存储器将从程序中某个位置产生的值传递到其他使用此值的地方。当使用共享地址空间，在运行于一个处理器上的线程或进程之间进行数据通信时，我们同样依赖于存储系统的这一属性。读操作返回的是最新写入所读取单元的值，与此值由哪个进程写入无关。由于所有的进程都通过相同的高速缓存层次访问存储器，不会有诸如高速缓存一致性问题的发生。当两个进程运行于共享同一存储系统的不同处理器上时，我们也希望能依赖于这样的存储属性。也就是说，含有多个进程的程序在物理上不同的处理器上运行，或者在同一个处理器上（以交叉方式或以多道程序方式）运行时，结果不应该有区别。然而，当两个进程通过不同的高速缓存访问存储器时会存在这样的危险：一个进程在其高速缓存中看到的是新值，而另一个看到的还是旧值。

5.1.1 高速缓存一致性问题

多处理器中的高速缓存一致性问题是普遍存在的，而且对机器性能有重大影响。例 5.1 描述是这方面的一个情况。

例 5.1　图 5-3 显示了 3 个带有高速缓存的处理器，经总线连接到共享主存。处理器有一个对单元 u 的访问序列。首先，处理器 P_1 从主存中读取 u，将 u 值的副本放在其高速缓存中。然后处理器 P_3 从主存中读取 u，将 u 值的副本放在自己的高速缓存中；然后 P_3 对 u

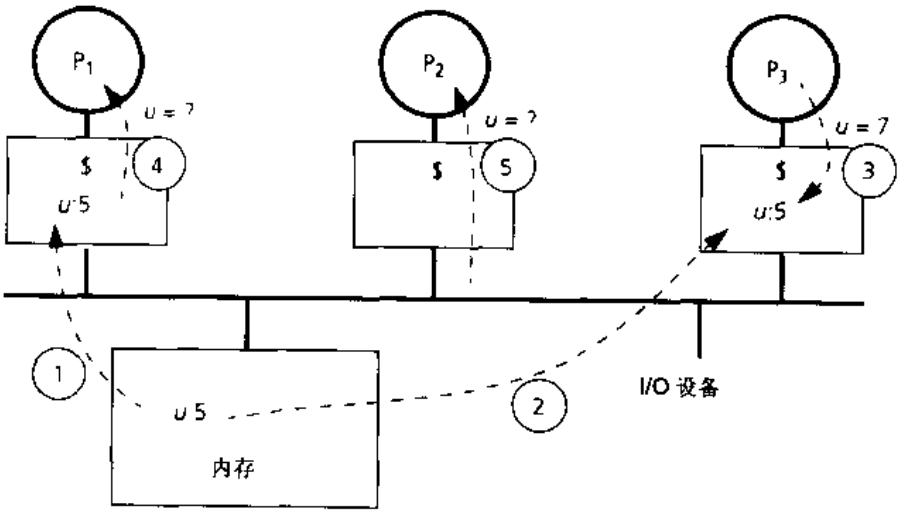


图 5-3　高速缓存一致性例子。此图显示 3 个带有高速缓存的处理器，经总线连接到共享主存。u 是存储器中的一个单元，这些处理器对其进行读写操作。读写操作的顺序由表现操作的弧上所标的数字给出。容易看到，当 P_3 将 u 的值由 5 改为 7 时，如果不采用什么特别的措施，P_1 从自己的高速缓存中读的 u 值是过时的，而 P_2 从主存中读的值也是旧的

写入，将其中的值由5改为7。在直写式高速缓存中，这会导致主存中相应位置的更新；然而，当 P_1 再次读取 u 时（第4个操作），不幸的是，它从自己高速缓存中读得旧值5，而不是从主存中读得新值7。这就是高速缓存一致性的问题。在回写式高速缓存中情况又如何呢？

解答：在回写式高速缓存中情况更糟糕。P_3 的写操作仅仅在保存单元 u 的高速缓存块的相关标志位作了标记，而不会立即修改主存。仅在此高速缓存块从 P_3 的高速缓存中替换出来时，其内容才会写回主存中。因此，不仅 P_1 读的是旧值，而且当 P_2 读 u 时（第5个操作），由于在高速缓存扑空它会从主存中读取，但这时读取的也是旧值5，而不是新值7。最后，如果多个处理器在其回写式高速缓存中将不同的值写入单元 u 时，最终到达主存的值由包含 u 的高速缓存块的替换顺序决定，而与各处理器对 u 进行写操作的顺序无关。■

显然，例5.1描述的行为与我们对存储系统应有行为的认识相抵触。事实上，即使在单处理器上，当执行I/O操作时也会产生高速缓存一致性问题。大部分I/O传输由直接存储访问（DMA）设备完成，DMA设备在存储器与外设之间移动数据不会涉及处理器。当DMA设备写入主存的某个单元时，除非采取特殊的操作，否则，如果此单元先前在某个处理器的高速缓存中，则此处理器看到的仍然是旧值。在回写式高速缓存中，DMA设备可能会从主存中读取某个单元的旧值，因为最新的值可能还在处理器的高速缓存中。由于I/O操作远不如存储操作频繁，在单处理器系统中采用了某些粗粒度的解决方案。例如，存储空间中用于I/O的段可以标记为“不可用于高速缓存”（即这段存储空间的内容不能进入处理器的高速缓存），或者在访问用于与I/O设备通信的存储段时，处理器总使用不通过高速缓存的载入和存储操作。对于每次传输较大数据块的I/O设备，例如硬盘，操作系统会提供支持以确保数据的一致性。在许多系统中，在I/O操作进行之前，数据传输涉及到的存储页会由操作系统根据处理器的高速缓存内容进行刷新。在另一些系统中，所有的I/O操作均会经过处理器的高速缓存层次，由此来维护一致性。当然，这种策略有其消极影响，因为处理器不会马上用到的数据也可能被记录在高速缓存中了。幸运的是，用于解决多处理器高速缓存一致性的技术和支持同样可以解决I/O一致性问题。现在，几乎所有的多处理器均提供对高速缓存一致性支持。

273
274

在多处理器中，不同进程读取和写入共享变量是经常性事件，因为这是并行应用程序的多个进程之间通信的基本方法。因此，我们不希望禁止在高速缓存中存有共享数据，也不希望在引用所有共享变量时必须调用操作系统的相应操作。这时，高速缓存一致性需要作为一项基本的硬件设计任务来完成；例如，当某个共享单元被修改时，此位置在高速缓存中的旧副本（如例5.1中 P_1 高速缓存里 u 的副本）必须淘汰，要么使此值无效，要么用新值更新。事实上，操作系统自身也会从这种透明的、硬件支持的数据结构一致性中获益匪浅。

在探索可提供一致性的技术之前，有必要更精确地定义一致性的概念。我们的直观概念“每个读操作必须返回最后写入此位置的值”在并行体系结构中有些问题，因为“最后”一词没有很好地定义。两个不同的处理器可能在同一时刻对同一存储单元进行写操作；或者对于同一个存储单元，一个处理器可能在另一个处理器刚刚发出写操作后马上发出读操作，但由于光速或其他什么因素的影响，写者的更新根本来不及传播到读者。[○]即使在顺序程序的

○ 这里强调从处理器看到的操作的发出时间，而不是完成时间，见后面关于操作发出的定义。——译者注

情况下，“最后”一词也不是时序或物理概念，它指的是程序定义的操作次序中的“最后”。现在，我们可以暂时将进程所对应的程序操作次序看作是在机器语言程序中存储操作发生的次序。在5.2节中，关于程序操作序这一概念有进一步的详尽阐述。在并行情况下，问题在于程序操作序是由每个独立进程中的操作定义的，要定义一致性存储系统的语义，我们需要先形成关于一组程序操作序的概念。

我们首先回顾一下单处理器存储系统环境中某些术语的定义。对于存储操作，我们指的是一次单独的读取（载入）、写入（存储），或对存储单元的“读-改-写”访问。对于完成多个读取和写入操作的指令，例如那些出现在许多复杂指令集中的复杂指令，可以将其分解为多个存储操作。指令中的这些存储操作的执行次序由此指令确定。指令中的这些存储操作可看作以彼此相关的指定次序自动执行；也就是说，一个操作的各种影响应发生在下一操作的任何影响之前。称一个存储操作被发出，指的是它离开了处理器的内部环境并呈现给存储系统之时，这里提到的存储系统包括高速缓存、写缓冲区、总线和存储模块。关于序的一个十分重要的问题是，处理器观察存储系统状态的惟一方法是通过发出存储操作（例如，读操
275 作）；这样，我们称一个存储操作相对于这个处理器执行了，只要它能够感受到所发出的存储操作的效果。特别地，我们若称“一个写操作相对于这个处理器执行了”，则意味着该处理器后续的读操作能够返回由此写操作产生的值，或者返回由此写操作之后的另一个写操作产生的值。称“一个读操作相对于这个处理器执行了”，意味着该处理器后续的写入操作不会影响此读操作的返回值。注意，在这两种情况下我们都没有指明要访问存储芯片上的具体物理单元，也没有具体要求硬件上的哪些部分改变它们的状态。此外我们注意到，在串行情况下“后续”一词的含义是明确的，因为读和写操作的次序是由程序中语句的次序决定的。

关于存储操作“发出”和“相对于一个处理器执行了”的定义也可应用于并行情况；我们只需将定义中的“这个处理器”替换为“某个处理器”。问题在于，“后续”和“最后”这两个术语的含义出了一些问题，因为我们此时没有一个程序操作序：每个进程有自己的程序操作序，在访问存储系统时这些程序操作序会交织在一起。要澄清我们对一致性存储系统的概念，一种方法是设想在没有高速缓存的单一共享存储环境下会发生什么情况。对存储单元的每个写入和读取操作均会访问主存中的物理位置。这时的操作“相对于所有的处理器都执行了”，于是被称为完成了。因此，存储器就可以在所有处理器对一个单元发出的读写操作上强加一个序。进一步，在这个整体序中，来自某个处理器的读写操作应该遵从自己的程序操作序。那么，在这种情况下，主存单元就是一个硬件上的自然参照点，可确定跨处理器访问此单元的操作顺序。没有理由认为不同处理器必须按照特定的方式交替访问存储系统，因此任何交替访问均是合理的，只要此访问序列能够体现每个处理器的程序操作序即可。不过我们要假定一些基本的公平性原则，即最终每个处理器的操作都应该完成。我们的直观概念“最后的”可看作是在保持这些性质的一个假想串行序中最近的元素，“后续的”可以类似地定义。由于这一串行序必须保持一致性，所有处理器看到对某个单元的写操作应该是相同序的（如果它们愿意看的话，即读这一单元的值），这一点很重要。

对每个单元的操作都有一个总体的、串行的操作序是我们希望在任何一致性存储系统中见到的。当然，总体的序无须在执行程序时真正建立。特别是在具有高速缓存的系统中，我们不希望主存看到所有的存储操作，希望尽量避免串行化。我们仅希望程序行为就好像某种序存在一样。

使用更加形式化的方法，我们说多处理器存储系统是一致的，如果某个程序的任何执行结果都满足下列条件：对于任何单元，有可能建立一个假想的操作序列（也就是说，将所有进程发出的读写操作排成一个全序），此序列与执行结果一致，并且在此序列中：

1）任何特定进程发出的操作所表现出来的序和该进程向存储系统发出它们的序相同，并且 276

2）每个读操作返回的值是对相应单元按串行顺序写入的最后一个值。

关于一致性的这一定义中隐含了两个性质：写传播意味着写操作的效果对其他进程可见；写串行化意味着对于某个单元的所有写操作（来自相同的或不同的进程）而言，所有进程都是以相同顺序看到这些操作。例如，写串行化意味着，如果 P_1 进程对某个单元的读操作先看到由写操作 $w1$（由 P_2 完成）产生的值，再看到由 $w2$（P_3 中的操作）产生的值，那么 P_4（也可以是 P_2 或 P_3）的读操作看到的应该是同样的顺序。没有必要将写串行化的概念推广到读操作上，因为读操作的结果只有发出此操作的进程可见。

程序的结果可以看作是程序中读操作的返回值，也许可能增广到程序结束时对所有单元的一组隐含的读操作。从程序结果中，我们无法确定机器执行这些操作的真正次序，也无法精确获知数据位是何时更改的，只能知道程序外在的执行序。幸运的是，这也就够了，因为这就是处理器能够得到的所有信息。当讨论存储同一性模型时，这一概念会显得更加重要。

5.1.2　通过总线侦听的高速缓存一致性

定义过存储一致性的性质后，让我们来考察解决高速缓存一致性问题的技术。例如在图 5-3 中，我们如何保证 P_1 和 P_2 能够看到 P_3 写的值？实际上，高速缓存一致性的一种非常简单而漂亮的解决方法可以从总线的根本性质中得到。总线是连接几台设备的单独一组导线，其中每一设备都可以观察到每一个总线事务，诸如在共享总线上的每一次读或写操作。当一个处理器向它的高速缓存发出请求时，高速缓存控制器检查其高速缓存的当前状态并采取适当的措施，这可以包括产生总线事务以访问存储器。通过让所有的高速缓存控制器“侦听”总线并监管其上发生的事务，一致性得到维持，如图 5-4 所示（Goodman 1983）。如果一个总线事务和某个高速缓存控制器相关的话，例如在该高速缓存中含有与该总线事务相关的一个

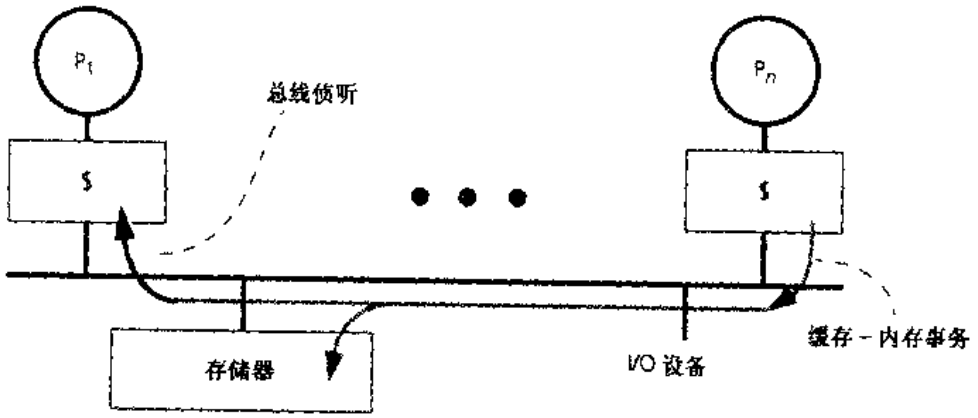


图 5-4　一个侦听高速缓存一致性的多处理器。带有各自私有高速缓存的多个处理器连接到一个共享总线上。每个处理器的高速缓存控制器时刻在总线上“侦听”，观察和它相关的总线事务并相应地更新其状态以保持高速缓存的一致性。灰色箭头表示总线上出现的一个事务并被主存接受，这和单处理器系统一样。黑色箭头指示的是“侦听”

存储块的拷贝，该控制器可能采取行动。这样的话，如果 P_1 看到了 P_3 的写操作，它可能采取行动，比如使这个拷贝无效或使它更新。实际上，由于高速缓存中数据的分放和替换是以高速缓存块（通常为几个字长）的粒度进行管理的，并且访问高速缓存扑空时要从存储器获取一整个高速缓存块的数据，因此通常高速缓存一致性也是以高速缓存块的粒度来维持的。换句话说，在高速缓存中，要么整个高速缓存块处于有效态，或者整个高速缓存块都处于无效态。因此，高速缓存块是高速缓存中数据分配的粒度，是高速缓存间数据传输的粒度，也是缓存一致性管理的粒度。

下面我们看支持一致性的总线的最关键性质。首先，总线上出现的所有事务对高速缓存控制器来说是可见的。其次，它们对所有控制器以同样的顺序（也就是它们出现在总线上的顺序）可见。一个一致性协议必须保证为响应存储操作的所有“必要的”事务都真正出现在总线上，并且当控制器发现一个相关的事务时将做出适当的动作。

保持一致性最简单的例子是一个含有单级直写高速缓存的系统。这基本上就是在 20 世纪 80 年代中期那些最初的基于总线的商用 SMP 所遵从的方法。在那样的系统中，每一个写操作引发总线上的一个写事务，于是每一个高速缓存控制器将观察到每一次写操作（于是实现了写传播）。如果某个侦听高速缓存有这个块的一个拷贝，它或者使这个拷贝无效，或者更新该拷贝。前者被称为基于作废的协议，后者被称为基于更新的协议。不管何种情况，任何处理器随后访问这一块数据时它将看到最新的值，或者通过一次缓存扑空转而从主存得到，或者因为被更新的值在它自己的高速缓存中。主存中含有的总是有效的数据，于是当发现总线上的一个读操作时，高速缓存不需要采取任何行动。例 5.2 显示了图 5-3 中的一致性问题是如何被直写缓存解决的。

例 5.2　考虑图 5-3 中的情形。假设为直写缓存，试说明使用基于作废的协议时，总线怎样可以被用来提供一致性。

解答： 当处理器 P_3 把 7 写到 u 单元时，P_3 的高速缓存控制器生成一个总线过程以更新存储器的内容。P_1 的高速缓存控制器发现此总线事务是和它相关的并且是一个写事务，于是就作废它其中含有 u 的数据块。主存控制器将 u 的值更新为 7。以后从处理器 P_1 和 P_2 发出的读 u 的请求（动作 4 和 5）均将在它们私有的高速缓存中扑空，并将从主存获得正确的值 7。■

277
↓
278

确定一个总线事务是否与某个高速缓存相关从本质上说和对从处理器发出的一个请求进行标识匹配是完全一样的。采取的动作可以是作废或更新那块高速缓存块的内容或状态并且（或者）将那个高速缓存块的最新值从高速缓存中传送到总线上。

侦听式高速缓存一致性协议将在单处理器中也有体现的计算机体系结构的两个基本方面结合到了一起：总线事务和与高速缓存块相关的状态转换图。前面提过，一个总线事务由三个阶段组成：仲裁、命令/地址、数据。在仲裁阶段，想发起一次事务的设备发出它们的总线请求，总线仲裁器从中选择一个并且给出相应的准许信号作为响应。收到准许信号后，被选中的设备将命令，比如读或写以及相关的信号放到总线的命令和地址线上，所有的设备都会看到这个地址。在一个单处理器中，其中一个设备知道由它对这个特别的地址负责。对一个读事务，地址阶段接下来就是数据传输。根据数据是在地址阶段期间传输的还是在地址阶段之后传输的，写事务对不同的总线将有所不同。对大多数总线来说，一个响应设备能够发出并维持一个等待信号直到数据就绪为止。这个等待信号不同于其他的总线信号，因为它是

在各个处理器之间线“或”（wired-OR）的，即任何设备发出该信号都得到逻辑1。发起者不需要知道哪一个响应设备正在参与数据传输，而只需知道有这样一个设备并且它是否就绪即可。

计算机体系结构的第二个基本方面是，每一个单处理器高速缓存中的数据块，除了标记和数据外，还有一个状态和它联系，用来表明该数据块的特征（例如，无效、有效、脏）。高速缓存的策略由*高速缓存块的状态转换图*来定义，它是一个指示数据块的特征如何改变的有限状态自动机。高速缓存块的转变发生在访问该数据块或者访问一个映射到和此数据块相同的高速缓存线的一个地址上。（我们用高速缓存块来表示实际数据，用缓存线来表示在高速缓存硬件中固定的存储位置，这和主存中的页和页帧区别是类似的。）虽然只有实际存在于高速缓存线中的数据块才具有硬件状态信息，逻辑上，所有不在缓存中的数据块均可以被看作或者处于一个特殊的“不存在”状态，或者处于“无效态”。在一个单处理器系统中，对一个直写、写不分配的高速缓存（Hennessy and Patterson 1996）仅仅需要两个状态：有效态和无效态。起初，所有的数据块都是无效的。当一个处理器对高速缓存的读操作扑空时，系统要生成一个总线事务来从存储器中调入该块，该块被标记为有效。写操作生成一个总线事务来更新存储器，如果高速缓存块处于有效态，它们也更新该块。写操作不改变数据块的状态。如果一个数据块被替换了，它可以被标记为无效直到存储器提供新的数据块使它成为有效的。一个回写缓存的每一条缓存线需要一个额外的状态，用来标记一个“脏”的或者说已被修改的数据块。

在一个多处理器系统中，一个数据块在每个高速缓存中都有一个状态，这些高速缓存状态根据状态转换图而改变。这样，我们可以把一个数据块的缓存状态看作一个由 p 个状态而不是单独一个状态组成的向量，这里 p 是高速缓存的个数。高速缓存状态由 p 个分离的 279
自动状态机来控制，由高速缓存控制器来实施。控制状态转换的状态机或状态转换图对所有的数据块和所有的高速缓存是完全相同的，但不同高速缓存中数据块的当前状态是不同的。像以前一样，如果一个数据块在高速缓存中不存在，我们可以假设它处于一个特殊的“不存在”状态或无效态。

在一个侦听式高速缓存一致性模式中，每个高速缓存控制器接收两套输入：由处理器发出的存储器请求以及由总线侦听器通知关于从其他高速缓存发起的总线事务。不管对应哪一种，根据当前状态和状态转换图，控制器都可以更新高速缓存中相关块的状态。控制器也可能采取一个行动。例如，它以被请求的数据来响应处理器的要求，潜在地生成新的总线事务以获得数据。它通过更新自己的状态来响应总线事务，有时候也干预事务的完成。这样来说，侦听式协议是一种分布式算法，由一组相互作用的有限状态自动机来表示。它由下面的部分确定：

- 和存在于局部高速缓存的存储块相联系的一组状态。
- 状态转换图，它以当前状态和处理器请求或观察到的总线事务为输入，并产生缓存块的下一状态作为输出。
- 和每一次状态转换相关的动作，它们部分是由总线、高速缓存和处理器的设计所定义的一组动作决定的。

一个数据块的不同状态自动机由总线事务来协调。

在图5-5中，状态转换图描述了一个简单的基于作废的协议，它针对是一致的直写、写

不分配的高速缓存。像在单处理器的例子中一样，每一个缓存块仅有两个状态：无效态（I）和有效态（V）（假设“不存在”态和无效态是一样的）。转换由引起转换的输入和由转换产生的输出来标记。例如，当一个控制器看到处理器传来的一个读操作在缓存中扑空时，就生成一个 BusRd 事务，在此事务完成时，该数据块转换为有效态。每当控制器看到处理器对某单元的一个写操作时，就生成一个总线事务来更新主存中的那个单元，但并不改变状态。对单处理器状态图的关键扩充在于，当一个总线侦听器看到总线上有一个对自己局部高速缓存的存储块的写过程时，控制器将那个数据块的状态置为无效，从而有效的废除了它的拷贝。（图 5-5 中用虚线显示此总线引起的事务）。以此扩展，如果任何处理器生成一个对缓存在其他所有处理器的数据块的写操作，那些处理器将使它们的拷贝无效。这样来说，一个数据块的多个同时出现的读操作可以共存而不会生成总线事务或无效，但写操作将废除所有其他被缓存的拷贝。

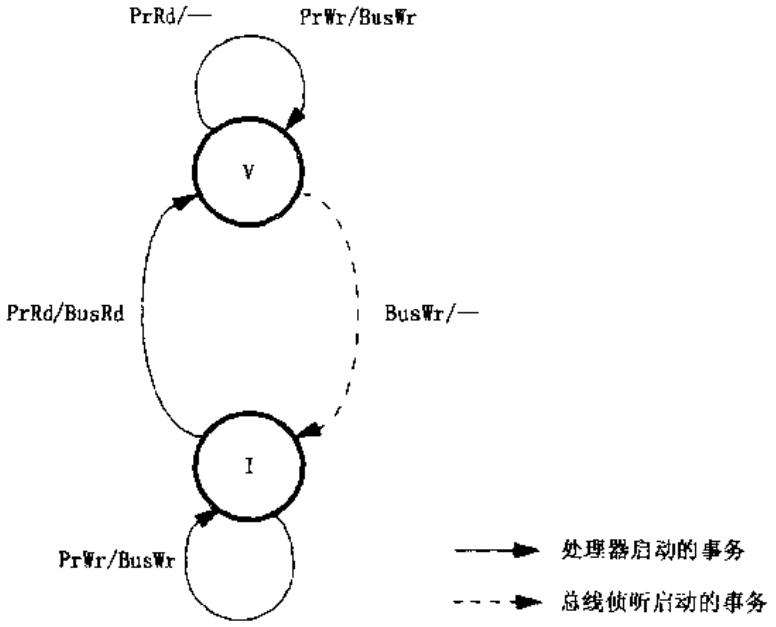


图 5-5　带有直写、写不分配高速缓存的多处理器的侦听一致性。直观上看，有两种状态，有效态（V）和无效态（I）。*A*/*B*（例如，PrRd/BusRd）表示如果 *A* 被观察到，那么将生成事务 *B*。从处理器的角度看，请求可以是读（PrRd）或写（PrWr）。从总线的角度来看，高速缓存控制器可以观察/生成总线读（BusRd）或总线写（BusWr）的事务

为看到这种简单的直写作废协议是如何提供一致性的，我们需要说明在此协议下程序的任何一次执行所含对一个单元的存储操作，都可以构造一个满足程序语句序和写串行化条件的完全序列。就目前的讨论，让我们假设总线事务和存储器操作都是原子的。也就是说，总线上一次只有一个事务在进行：一旦一个请求被放到总线上，此总线过程的所有阶段（包括数据响应），都在任何从其他处理器发出的请求被允许放到总线上之前完成（这种过程具有原子性的总线被称为*原子总线*）。同样，处理器要等待前面的存储操作完成才能发出下一个存储操作。对单级高速缓存来说，我们也很自然的假设作废被应用于高速缓存，从而写操作本身在总线事务期间完成。（本章将保持这些假设，但在第 6 章，当我们更细致地来看协议的实现和用更高的并行性来研究高性能设计时，这些假设将被放宽。）最后，我们可以假设存储器处理写和读操作的顺序和它们被提交到总线上的顺序一致。

在直写协议中，所有的写操作都在总线上出现。因为一次只进行一个总线事务；不管在哪次执行中，所有的对某一单元的写操作都按它们出现在共享总线上的顺序串行化（一致地），该顺序被称为总线顺序。因为每个侦听高速缓存控制器是在总线事务期间执行作废操作，于是作废操作是按总线顺序由高速缓存控制器来执行的。

280 ~ 281

处理器通过读操作“看到”写操作，于是对写串行化我们必须保证从所有处理器传来的读操作以串行化的顺序看到这些写操作。然而，对一个单元的读并不是完全串行化的，因为命中的读操作可以在它们的高速缓存中被独立地、并行地执行而不会生成总线事务。为了显示读操作如何可以被插入写操作的串行序列，考虑下面的情况。一个到达总线上的读操作（读扑空）由总线和写操作一起来进行串行化；因此它将获得按总线顺序最近写到相应单元的值。不出现在总线上的仅有的存储器操作是读命中操作。在这个例子中，所读的值是由同一处理器上对此单元最近的一次写操作或者通过它的最近的读扑空（以程序顺序）放到高速缓存中的。因为这两种值的来源都出现在总线上，读命中操作同样也看到在一致的总线顺序中产生的值。这样看来，在此协议下，总线顺序和程序顺序一起提供了足够的约束以满足一致性要求。

更一般地，我们可以通过观察下面的由协议强加的偏序来构造一个（假想的）满足一致性的全序：

- 如果操作是由同一个处理器发出的并且在程序顺序中存储器操作 M_2 在存储器操作 M_1 之后，则 M_2 是 M_1 的后继。
- 如果读操作生成一个过程在写操作 W 生成的事务之后，则该读操作是 W 的后继。
- 如果一个读或写操作 M 生成一个总线事务并且一个写操作的总线事务在 M 的总线事务之后，则该写操作是 M 的后继。
- 如果一个读操作并不生成总线事务（是一次命中）且并没有被另一个总线事务将其与写操作分离，该写操作是该读操作的后继。

任何保持这种偏序的串行顺序都是一致的。“后继”序关系是可传递的。在图 5-6 中勾画了这种偏序的一个例子，其中和写操作相关的总线事务分隔各自的程序顺序。虽然总线可能建立一个特别的顺序，但偏序并不限制从不同处理器发出的、发生在两个写过程之间的读操作总线事务的次序。实际上，只要遵循程序的顺序，此分隔中两个写操作之间的任何交错的读操作都是一个有效的串行顺序。

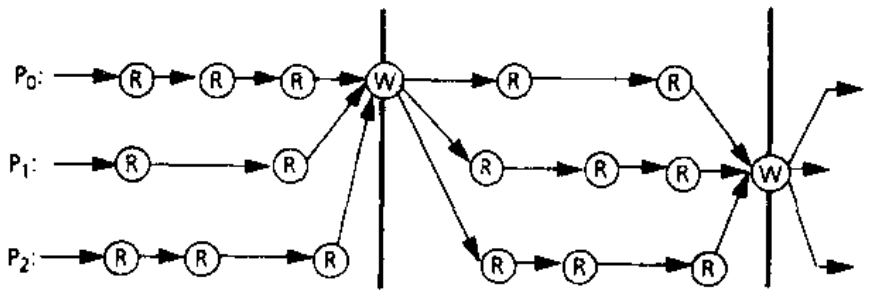


图 5-6　按照直写作废协议的一次执行的存储操作偏序图。写总线事务定义了一个全局事件序列，其间各个处理器按照程序操作序执行读操作。在保证各自的序的前提下，处理器之间操作的交织得到的任何全序是和该执行结果一致的

当然，这种简单的直写方法的问题在于每一个存放指令都要发到存储器，这就是大多数

微处理器使用回写高速缓存的原因（至少在接近总线的这一层）。此问题在多处理器的环境下被加重了，因为从每一个处理器发出的每一个存储命令消耗了珍贵的共享总线的带宽，从而造成可扩展性不高，正如例 5.3 所示。

例 5.3　考虑一个以 200 MHz 运行的超标量 RISC 处理器每一个时钟周期发出两条指令。假设此处理器的平均 CPI（每条指令的时钟数）为 1，15% 的指令是存储指令，并且每一次存储写 8 字节的数据。在没有达到饱和状态时，一个 1 GBps 的总线可以支持多少个处理器？

解答：单个处理器每秒钟将产生 3 000 万次存储指令（每条指令 0.15 个存储 × 每周期 1 条指令 × 每秒钟 1 000 000/200 个周期），因此总的直写带宽是每秒每处理器 240 MB 数据。即使忽略地址和其他信息并忽略读扑空，一条每秒 1 *GB* 的总线只能支持 4 个左右处理器。■

对大多数应用来说，一个回写高速缓存将吸收绝大多数写操作。不过，如果写操作不达到存储器，它们就不会产生总线事务，从而就不清楚其他高速缓存怎么能够观察到这些修改并且保证写的传播。同时，如果允许对不同高速缓存的写操作并发，对那些写操作的一种定序机制也不是一目了然的。我们需要某种更复杂的高速缓存一致性协议来使得“关键的”事件为其他高速缓存可见并且保证写的串行化。

针对回写高速缓存协议的设计空间是相当大的。在研究它之前，让我们先回到在本章引言部分提到的更一般的定序问题，考察由存储同一性模型确定的共享地址空间的语义。

5.2　存储同一性

如果信息是由一个处理器对某个单元写入，而另一个处理器从中读出这样的方式来得以传递的话，那么我们前面所关注的一致性将是非常重要的。最终，写在一个单元的数据将对所有的读取者都会是可见的；但这种一致性并没有指明所写入的数据何时成为可见的。通常，在编写一个并行程序时，我们希望确保一个读操作能够返回一个特定的写操作的值；也就是说，我们希望在写和读之间建立一种序。典型地，我们使用某种形式的事件同步来表达
282 ? 283 这种依赖关系，并且为此用到的存储地址不止一个。

比如，考虑图 5-7 中的代码段被处理器 P_1 和处理器 P_2 执行，我们在第 2 章讨论在一个共享地址空间中的点对点事件同步时见过。很明显，在这个程序中，当共享变量 flag 值为 1 时，一直处于空闲状态的处理器 P_2 才能跳出循环，并接着输出变量 A 的值。因为变量 A 的值先于 flag 已经被处理器 P_1 更新了，所以输出的值为 1。在这种情形下，我们通过对另一个地址空间（如 flag）的访问来维持不同处理器访问同一地址空间（如 A）的某种期望顺序。特别是，假定了 P_1 对变量 A 的写操作在它对 flag 的写操作之前就可见了，并且 P_2 对变量 flag 的读操作使它跳出 while 循环，在它对 A 的读操作之前就完成了（一个输出操作就是一个读操作）。前述一致性概念在此并没有隐含着处理器 P_1 和 P_2 对不同地址单元的

P_1	P_2
/*Assume initial value of A and flag is 0*/	
A = 1;	while (flag == 0); /*spin idly*/
flag = 1;	print A;

图 5-7　通过标记变量做事件同步的需求。此图显示两个处理器并发执行两个不同的代码段。从程序员的直觉来看，输出的 A 值必须是 1，因为根据程序语句序，如果进程 P_2 看到 flag = 1，那么它必须也能看到 A = 1

访问顺序，一致性在此仅仅要求 A 的当前值最终对处理器 P_2 是可见的，而不一定是在 flag 的新值被观察到之前。

程序员可能试图通过使用栅障或者其他显式的事件同步方式（如图 5-8 所示）来避免这个问题。我们期望上例中的变量 A 的值在栅障前被设定为 1，其后的输出也为 1。但是这个解决方案存在两个潜在的问题。首先，我们给栅障的语义增加了一个假设，即不仅进程在运行到栅障时必须等待所有其他进程的到来，而且进程在运行到栅障前发出的所有写操作在运行到栅障时要对其他所有进程可见。第二个问题是：一个栅障常常是通过对普通的共享变量（比如，图 5-8 中的 b1）的读写操作来实现的，而并不需要专门的硬件支持。既然这样，那么从机器的角度看，上述方法仅仅是对经过编译的代码中不同的共享变量的访问，而并不是专门的栅障操作。但是，存储同一性却根本没有涉及到这些访问的顺序问题。

P_1	P_2
/*Assume initial value of A is 0*/	
A = 1;	. . .
- - - BARRIER(b1)- - - - - -	BARRIER(b1)- - - - - - -
	print A;

图 5-8　用栅障来显式同步，保证对同一个单元不同访问的一定顺序。类似于图 5-7，程序员期望输出 A 的值为 1，由于通过了栅障意味着 P_1 对 A 的写操作已经完成，从而为 P_2 可见

显然，我们希望一个存储系统能提供更深刻的东西，而不仅仅只是对每个单元都能“返回最后一次写操作的值”。为了建立不同的进程对相同单元（例如 A）的访问顺序，我们有时希望一个存储系统遵从同一进程对不同的存储单元（A 和 flag 或者 A 和 b1）执行读写操作的顺序。一致性也没有涉及到对不同存储单元的写操作成为可见的顺序问题。同样地，一致性也没有涉及 P_2 对不同存储单元的读操作相对于 P_1 所见到的顺序。由此，存储同一性本身并没有防止在以上每个例子中输出结果为 0，当然这不是程序员所期望的。

在其他情况下，程序员的主观意图可能并不是很明确。考虑图 5-9 中的例子，进程 P_1 执行的存储访问是通常的写操作，而且 A 和 B 并没有作为标记变量或者是同步变量来使用。如果变量 B 输出值为 2，那么，从直觉上我们会认为变量 A 的输出值为 1 吗？不管答案如何，这两条输出语句都对不同的存储地址执行了读操作，并且一致性概念也没有讲 P_1 的不同写操作对 P_2 是可见的顺序。这个例子实际上是 Dekker 算法（Tanenbaum and Woodhull 1997）中的一个程序段，用来决定两个进程谁先到达一个临界点，成为互斥得到保证的一个步骤。这个算法依赖于这样的假设：一个进程对不同存储地址的写操作为其他进程可见的顺序和那些写操作在程序中出现的顺序相同。很明显，我们需要某些规则，而不仅仅是前面讨论过的一致性来给共享存储地址空间一个明确的语义，也就是定义一个序模型；依照该模型，程序员能推断他们程序的执行结果及其正确性。

P_1	P_2
/*Assume initial values of A and B are 0*/	
(1a) A = 1;	(2a) print B;
(1b) B = 2;	(2b) print A;

图 5-9　没有同步的访问序。这里，由于既没有用标记，也没有用显式事件同步，程序员的意图不太清楚

在共享存储空间上，多个进程对存储器的不同单元做并发的读写操作，每个进程都会看到一个这些操作被完成的序。一个存储同一性模型规定了对这种序的若干约束。值得一提的是，这里涉及的并发存储操作既包括对相同单元的，也包括对不同单元的；既可以来自同一进程，也可以来自不同进程。在这个意义上，存储同一性包含了一致性。

284
~
285

5.2.1　顺序同一性

在第1章关于通信体系结构基本设计的讨论中，1.4节描述了关于共享地址空间的一个序模型：对于一个多线程程序在单处理器环境中任意交织执行的性质所做的任何推断，在多处理器环境下，线程分到不同处理器上并行运行时应该继续有效。所以，在一个进程中的数据访问顺序也就是这个程序的执行顺序，并且在不同进程间的数据访问顺序也就是某种程序执行顺序的交织。也就是说，在多处理器的情形不应该使进程在共享地址空间中看到任何交织执行都不可能产生的值。这种直观模型被 Lamport 形式化为顺序同一性模型（Sequential Consistency，SC），其精确定义如下（Lamport 1979）[⊖]：

称一个多处理器是顺序同一的，如果程序在它上面的任何一次执行的结果都和其中所有操作按某种顺序执行的结果一致，并且在这种顺序执行中每个处理器所完成的操作的顺序和它的程序执行顺序一致。

图5-10描述了一种由顺序同一性系统呈现给程序员的存储抽象（Adve and Charachorloo 1996）。除了现在是针对多存储单元外，它同我们曾用来介绍存储同一性的机器模型是相似的。尽管在真实机器里，主存可能被分布在多个处理器中，并且每个处理器有它自己的高速缓存和缓冲区，多个进程在这里看起来是共享一个单一的逻辑存储空间。每个进程依照程序原序执行并完成一次次存储操作；即一个存储操作要待相同进程内的前一操作完成后才开始执行。另外，公共存储区按照某种任意（但希望是公平）的调度方式，交叉响应来自不同进程的操作请求。存储操作是这个交叉顺序中的原子操作，即每个存储操作应该是全局性的

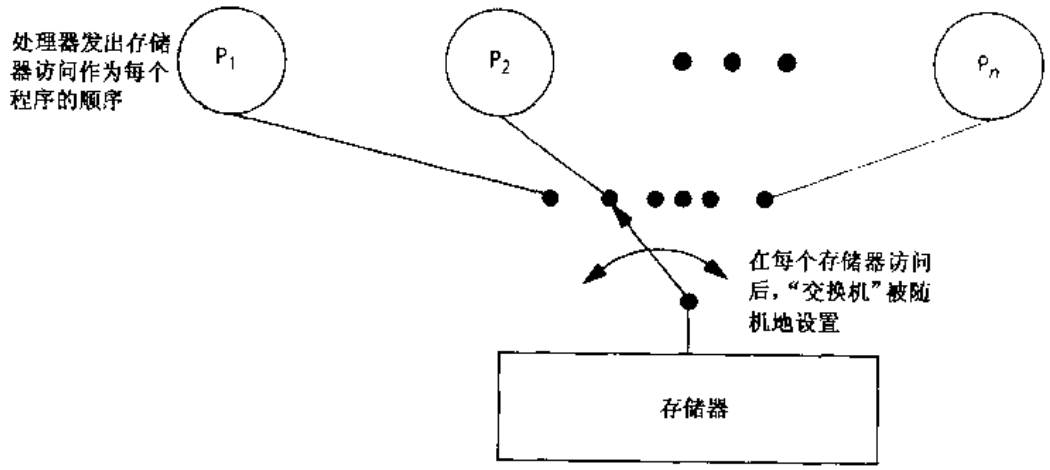


图5-10　在顺序同一性模型下程序员所看到的存储子系统的抽象。这个模型对程序员完全隐藏了背后存储系统硬件的并发性（例如，可能的分布式主存储器、高速缓存和写缓冲区等）

⊖ 在软件系统中有两个相关的概念，一是对于数据库并发更新的可串行性（Papadimitriou 1979）；二是关于并发对象的可线性（Herlihy and Wing 1987）。

(对于所有的进程)，一个操作完成后下一个操作才开始执行。

如同一致性，在这里存储操作具体以何种顺序来执行甚至完成是不重要的。对于顺序同一性来说，真正重要的是存储操作的完成满足上述约束条件。在图5-9的例子中，按照SC模型，(A，B)结果值为(0，2)是不允许的，这和我们直观上的认识一致，否则处理器P_1对A和B执行的写操作就不符合程序执行序了。然而，这些存储操作可能确实以1b、1a、2b、2a这样的顺序来执行并完成。它们实际完成的顺序不同于程序执行序是没有关系的，因为执行结果(1，2)与存储操作按照程序执行序来执行并完成的结果是相同的。另一方面，因为产生了在SC模型下不允许的结果(0，2)，所以实际的执行顺序1b、2a、2b、1a不是顺序同一性的。在习题5.6中，还有一些说明顺序同一性的例子。值得注意的是SC模型并不意味着可以不同步了。其原因是SC模型只是允许任意交错的不同进程的操作在单个指令的粒度上执行。如果我们想要在一个进程中的多个存储操作之间维护原子性(互斥)，或者如果我们想要在交错进程间强加某种约束，那么同步是必需的。 286

术语“程序执行序”(也称程序原序)也具有某些需要斟酌的细节。从直观上看，某个进程的程序执行序仅仅是源程序中语句执行的顺序；更明确地说，它是编译器按照直截了当的方式所产生的汇编代码中存储操作的发生顺序。但这不一定是优化编译器产生的硬件存储操作顺序，因为这个编译器可能对存储操作重新排序(遵循特定的约束条件，比如对相同存储地址的存储依赖性)。程序员的头脑中有的是源程序中语句执行的顺序，但是处理器只关注机器指令的执行顺序。事实上，在并行计算机体系结构的每一个接口上都有一个“程序执行序”，特别是在程序员可见的程序设计模型接口上和硬件/软件接口上，并且这个序模型是可以分别定义的。因为程序员总以源程序来推断程序执行的行为，所以在讨论存储同一性模型时，使用源程序来定义程序执行序是有意义的；也就是说，我们所关心的同一性模型由程序设计语言和执行系统呈现给程序员。

实现SC模型，要求系统(包括软件和硬件)体现先前定义的直观上约束条件。这里实际上是两个条件。一个是程序执行序的要求：一个进程的存储操作为自己以及别的进程成为可见的顺序必须符合程序执行序。另一个约束是基于操作的原子性假设来确保所有操作的全序，或者说在进程间交叉执行的情况，对所有进程都是一样的。也就是说，一个所有进程都能看到的操作的完成应该在总体顺序中的下一个操作开始执行前(不管是由哪个进程来执行)。对第二个约束值得注意的是要使写操作表现出原子性，尤其是在一个多副本的存储系统中，一个写操作需要影响到相关的所有副本。在先前定义的顺序同一性中，写操作原子性的要求，意味着一个写操作在所有操作全序中的执行位置应该对所有处理器都是一样的。这就保证了当一个处理器看到自己执行的一个写操作的效果后，在别的处理器见到这个写操作产生的新值前，这个处理器所做的任何事情(例如另一个写操作)对其他处理器都是不可见的。从效果上看，SC模型要求的写操作原子性扩展了一致性要求的写操作的串行化：即写操作的串行化要求同一个存储单元的一系列写操作在所有处理器看来都以相同的顺序发生，而写操作的原子性要求所有的写操作(对于任何存储地址)在所有处理器看来是以相同的顺序发生的。例5.4表明了写操作原子性的重要性。 287

例5.4 考虑图5-11中的三个进程。指出如果写操作的原子性得不到保证，顺序同一性就做不到。

解答： 因为处理器P_2直到变量A值为1时才开始运行，并接着置B的值为1，而处理

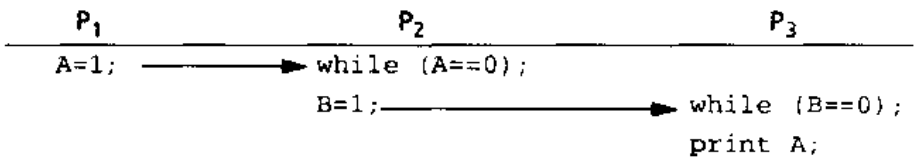


图 5-11　写操作原子性对顺序同一性的重要性的示例

器 P_3 直到变量 B 值为 1 时才开始运行，并接着读出变量 A 的值，由传递性我们可以推断出处理器 P_3 应该发现变量 A 的值为 1。如果处理器 P_2 被允许越过对变量 A 的读操作，在确保处理器 P_3 已经可见变量 A 的新值前对变量 B 执行写操作，那么处理器 P_3 就可能读出变量 B 的新值和变量 A 的旧值（比如，从它的高速缓存中），这样也就和我们对顺序同一性的认识不相符。■

更形式化地说，每个进程的程序执行序在所有操作集上强加了一个偏序；也就是说，它在自己所执行的操作（所有操作的一个子集）上强加了一个序。不同进程操作的交织执行就定义了一个全序。因为 SC 模型并没有定义具体的交织方式，满足每个进程操作偏序的总体程序执行序可能会很多。因此我们有下列定义：

- *顺序同一性的执行*。如果程序的一次执行产生的结果与前面定义的任意一种可能的总体顺序（交织产生的）产生的结果一样，那么程序的这次执行就称为是顺序同一的。也就是说，存在这么一个总体顺序，或者说是程序的一种交叉执行，能产生与实际执行相同的结果。
- *顺序同一性的系统*。如果在一个系统上的任何可能的执行都是顺序同一的，那么这个系统就是顺序同一的。

288

5.2.2　保证顺序同一性的充分条件

讨论了定义和高层需求，让我们来看看多处理器的实现怎么能够满足 SC。我们可能定义一组充分条件，通过它们使得顺序同一性在多处理器中得到保证，无论是基于总线还是分布存储的，也不管是否有高速缓存的一致性。下面这些条件，最初源于（Dubois, Scheurich, and Briggs 1986; Scheurich and Dubois 1987）是相对简单的：

1）每个进程按照程序执行序发出存储操作。

2）发出写操作后，进程要等待写的完成，才能发出它的下一个操作。

3）发出读操作后，进程不仅要等待读的完成，还要等待产生所读的数据的那个写操作的完成，才能发出它的下一个操作。也就是说，如果该写操作对这个处理器来说是完成了（即返回了所写的值），那么这个处理器应该等待该写操作对于所有处理器都完成了。

第三个条件保证了写操作的原子性，要求是相当高的。由于读操作必须等待逻辑上先前的写操作变得全局可见，它不是一个简单的局部限制。我们注意到这些是充分的、但不是必要的条件。在许多情形，顺序同一性可以在较低的串行化要求下也能得到保证，后面将会看到。

源程序定义了一种程序的执行顺序，重要的是编译器不应该改变程序呈现给硬件（处理器）存储器操作的次序。否则，从程序员所看到的顺序同一性就可能被破坏，即使还不一定涉及到硬件层次。但是，在编译和处理器中广为采用的许多优化技术违反这个充分条件。例

如，在一个进程内对不同存储单元的访问重新排序是编译器一贯做的事情，因此处理器就可能给出和程序员所看到的不同的访问顺序。显式并行程序用的是单处理器的编译器，它只关心保持对相同单元的相关性。那些极大改善性能的高级编译优化技术，诸如公共子表达式的 289
删除、常数传播、寄存器分配和循环变换，如循环拆分、循环反序和循环封锁（Wolfe 1989）等等，能改变对不同单元访问的次序，甚至消除一些存储操作[○]。在实践上，为了限制这些编译优化，多线程和并行程序对那些用来维持某种序要求的变量或存储引用进行注释。一种特别严格的例子是在变量声明时赋予其 volatile 属性，它告诉编译器不要对该变量进行寄存器分配，也不要改变在该变量上的存储操作相对于它前后操作的次序。例 5.5 是关于这些要点的一个解释。

例 5.5　在图 5-7 中，不同的存储操作的序会如何影响串行程序的语义（只一个进程运行）、在多处理器上并行程序的语义以及在一个多线程程序上的语义（其中两个线程在同一个处理器上交叉执行）？你怎样解决这个问题？

解答：编译可以改变对 A 和 flag 写操作次序，并对串行程序没有影响。然而，这可能违反我们对于并行程序和并发（或多线程的）单处理器程序行为的直观认识。对后者，上下文切换可能在两个重排序的写之间发生，于是切换进来的进程可能看见对 flag 的更新但看不到对 A 的更新。如果编译器改变了读 flag 和 A 的次序，类似违背直觉的情况也会出现。对许多编译来说，通过声明变量 flag 为 volatile integer，而不是简单的声明为 integer，就可以避免变序的情形。还有一些其他的解决方案，将在第 9 章讨论。■

即使编译器保持了程序指令的执行顺序，现代处理器所用的一些复杂的机制，诸如写缓冲、交叉存储、流水线和乱序执行技术（Hennessy and Patterson 1996）等，也会使得从进程看到的存储操作的发出、执行和/或完成不同于程序中所描述的次序。和编译优化类似，这些系统结构方面的优化对于串行程序是能正常工作的，这是因为在那些程序中序的表现只要求对相同存储单元的访问的相关性得到保证，如图 5-12 所示。在并行程序中的问题是，一个进程对不同共享变量操作的乱序处理能被其他进程检测到。

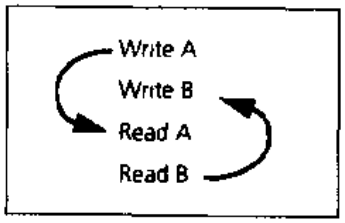


图 5-12　维持一个串行程序在单处理器上执行的操作序。只是对应于两个相关弧的序必须被维持。前面两个、后面两个和中间两个操作可以换序

执行在多处理器中，上述保持 SC 的充分条件是一个相当强的要求，它使编译器变序和乱序执行技术的使用受到了限制。若干弱化一些的同一性模型和技术提了出来，满足 SC 的要求，但放松了充分条件。在第 9 章讨论可扩展共享存储系统时将会考察这些方法。对于本章来说，假定编译器不改变存储操作的顺序，从而处理器所看到的程序的顺序和程序员所看到的是相同的。在硬件方面，假设必须满足充分条件。为此，我们需要有一个处理器能检测

○ 我们注意到，现代编译器用来消除一些存储操作的寄存器分配，不仅会影响存储同一性，还会影响一致性。对于图 5-7 中的标记同步例子来说，如果编译器在 P_2 中将变量 flag 分到了寄存器中，可能会导致进程永远踏步等待下去；高速缓存一致性硬件只是更新或者作废存储器和高速缓存中的内容，而不涉及到寄存器，于是违反了一致性所要求的写传播性质。

它的写操作完成，从而可以继续向下执行的机制（读操作完成的检测是容易的；当数据返回到处理器时，读就完成了）。还需要机制来保证写操作的原子性。对于本章所考虑的所有协议和系统来说，我们将看到它们如何满足一致性（包括写操作的串行化），如何满足顺序同一性（特别地，如何检测写操作的完成、写原子性的保证）以及在仍然满足这些充分条件下，可以进行什么样的简化处理。

对基于总线的机器来说，共享总线所带来的必然的操作串行化对存储器操作的定序是很有用的。容易验证，前面讨论过的两状态直写作废协议实际上相当容易地提供了顺序同一性——不仅是一致性。将关于一致性的论点扩充的关键是注意到对所有存储单元的读和写扑空（不仅是对个别单元），都在总线上串行化了。当读操作得到了一个写操作的值，那个写操作肯定已经就完成了（由于它引起了一个前面的总线事务），这样就保证了写操作的原子性。当对任何处理器进行一个写操作的时候，所有先前的写操作已经以它们访问总线的次序完成了。

5.3　总线侦听协议的设计空间

基于侦听的高速缓存一致性的精彩之处在于，解决一个困难问题的整个机制可以归结到对在系统中自然发生的若干事件进行少量的额外解释。处理器完全不需改变。在程序中不需要插入显式的一致性操作。通过扩充高速缓存控制器的功能，利用总线的性质，程序中固有的读和写隐含地保持了高速缓存的一致性，由总线提供的串行化维护了同一性。每个高速缓存控制器观察并解释由其他高速缓存产生的总线事务，并相应维护自己的内部状态。我们最初的针对直写缓存的设计不是很高效的，但我们现在可以来研究侦听式协议的设计空间，高效利用有限的共享总线带宽。所有这些都用回写高速缓存，允许多个处理器并发地在它们的局部缓存中写不同的存储块，不用任何总线事务。这样，为保证足够的信息在总线上传送，需要引入额外的措施来维护一致性。

290
|
291

回顾单处理器的回写高速缓存，处理器写扑空引起缓存从存储器读进一个存储块，更新其中的一个字，将这一块置成已修改（或者脏）状态，以便在替换时写回存储器。在多处理器的情形，这个已修改状态也由协议用来指出一个缓存对某一存储块的单独拥有权。一般来说，我们称某个高速缓存是一存储块的拥有者，如果当对该块有请求时必须由它提供数据（Sweazey and Smith 1986）。称一个高速缓存独立地拥有某一存储块的一个拷贝，如果它是含有该块有效拷贝的惟一高速缓存（主存可能有也可能没有一个有效的拷贝）。这种排它性隐含着该高速缓存可以修改这一存储块而不需要通知其他高速缓存。如果一个高速缓存对一存储块的拥有没有排他性，那么它就应该首先产生一个通知其他高速缓存的总线事务，才能向该块写入新值。试图做写操作的这个高速缓存可能有一份有效的块，但由于要产生一个总线事务，它也称为写扑空，就好像要写入一个不存在或者无效的块一样。如果高速缓存里的这一块处于修改状态，那么显然它是拥有者，并且有排他性。（区别拥有权和排他性的必要性很快就会很清楚了。）

在作废协议中发生一次写扑空时，一种特殊形式的总线事务，称为排他读，用来告诉其他高速缓存将发生一次写操作，并且获得对于该块的单独拥有权。这就是将该块以修改状态放入高速缓存，然后就可以写了。多个处理器不可能并发写同一个存储块，不然会引起非一致的值。由这样的写操作产生的排他读总线事务将由总线来串行化，因此一次只有一个能获

得对该块的单独拥有权。高速缓存一致性的动作由这两种类型的过程驱动：读和排他读。最终，当一个已修改块从高速缓存中被替换时，数据被写回存储器，但这个事件不是由一个对该块的存储操作引起的，对协议几乎是随机性的。不处于修改状态的块在替换时不需要回写，简单地丢掉即可，这是因为存储器有最新的拷贝。许多协议都是针对回写高速缓存设计的，我们来考察一些基本的可选方案。

我们也考虑基于更新的协议。回忆在基于更新的协议中，只要一个共享单元被一个处理器写，它的值就在所有含有该存储块的处理器的高速缓存中更新[⊖]。这样，当这些处理器接着访问这一存储块时，它们就可直接从它们的低时延高速缓存中得到。所有其他处理器的高速缓存的更新由一次总线过程完成，这样当有多个共享者时就节省了带宽。相比之下，对于基于作废的协议，一个处理器在执行写操作时，所有含有相应存储块的处理器高速缓存的状态被置成无效的，于是这些处理器在下一次读的时候必须通过一次扑空，从而也是一次总线事务来获得那一块。不过对执行该写操作的处理器来说，在其他处理器访问之前，对那一存 292
储块后继的写操作不会引起总线事务（在更新协议时是会引起的）。在一个处理器连续向一个存储块做写操作，中间没有其他处理器访问该块的情形，这是有吸引力的。具体的折中要复杂许多，它们取决于机器的工作负载情况。对此，在 5.4 节有量化的讨论。一般来说，基于作废的策略要更健强，因此许多厂家都提供作为缺省协议。有些厂家提供更新协议作为备选，用于某些对应于特定数据结构或页的存储块。

对于协议（更新或作废）和高速缓存策略所作的选择直接影响状态的选择、状态转换图和相关动作的选择。对于计算机体系结构设计师来说，在这一层次的设计任务有很大的灵活性。我们这里不一一列举所有的可能性，下面通过考虑三种常见的一致性协议来体会有关的设计思路。

5.3.1　一种三态（MSI）回写作废式协议

我们这里考虑的第一个协议是针对回写缓存、基于作废的协议。它和在 Silicon Graphics 4D 系列多处理器中所用的很相似（Baskett，Jermoluk，and Solomon 1988）。这个协议利用任何回写协议都要有的三个状态，来区别有效存储块的未修改（干净）和已修改（脏）。具体来说，三个状态分别为*已修改的*（M），*共享的*（S）和*无效的*（I）。无效的意义是明显的。共享意味着该存储块在这个高速缓存中存在但未修改，主存中有最新的值，其他高速缓存有可能有最新的拷贝（共享）。已修改的，也称脏的，意味着只有这个缓存有一个该存储块的有效拷贝，主存中的拷贝是过时的。在对一个共享或者无效块做写操作，并将其置成已修改状态之前，所有其他拷贝必须通过一个排他读总线事务作废。除了引起作废外，这个总线事务还用来为写操作定序，从而保证写操作对其他高速缓存是可见的（写传播）。

处理器发出两种类型的请求：读（PrRd）和写（PrWr）。所读和所写的，可能是存在于高速缓存中的一个存储块，也可能是高速缓存中不存在的。对于后者的情形，高速缓存中的某一块就要被新请求的块所替换，如果所存在的块处于已修改状态，它的内容就要被写回主存。

⊖ 这是一种写广播的情形。人们也研究过读广播的设计，其中一个高速缓存在总线上看到一个读操作后将自己含有的一个已修改存储块送上总线，所有其他高速缓存中的拷贝也得到更新。

我们假定总线允许下面的过程：

- 总线读（BusRd）：这个过程由引起高速缓存扑空的 PrRd 产生，处理器指望一个数据返回作为结果。高速缓存控制器将地址放在总线上，请求一个它不想修改的拷贝。存储系统（可能是另外的高速缓存）提供数据。
- 总线排他读（BusRdX）：这个过程由 PrWr 产生，这个 PrWr 要读的一个存储块，要么
293 不在高速缓存或者在高速缓存但不处于已修改状态。高速缓存控制器将地址放到总线上，请求一个它要修改的排他拷贝。存储系统（可能是从另一个高速缓存）提供数据。所有其他高速缓存被作废。一旦高速缓存得到了这个排他拷贝，写就可以在高速缓存里完成。处理器可能要求一个认可来作为这个过程的结果。
- 总线回写（BusWB）：这个过程由高速缓存控制器在回写时产生；处理器不知道它的发生也不指望有一个响应。高速缓存控制器将地址和存储块的内容放到总线上。主存被更新。

总线排他读（有时称为读后拥有）是仅有的特别和高速缓存一致性有关的过程。为支持回写协议所需要的新动作是，除了改变高速缓存块的状态外，一个高速缓存控制器能够根据一个观察到的总线事务，从它的高速缓存中将一个被引用的存储块提出来，放在总线上，而不是让主存来提供数据。当然，高速缓存控制器也能启动如上所述的总线事务，提供回写数据，或者拾取出存储系统提供的数据。

1. 状态转换

图 5-13 中的状态转换图表达了在这种侦听式协议中一个存储块的各种情况。其中的状态是这样组织的，越是接近顶部的状态，表示对应的存储块和处理器绑定得越紧。处理器读一个无效的（或者不存在的）存储块，引起一个 BusRd 事务来解决这次扑空。新装入的存储块在发出请求的缓存中被提升，在这个状态图中，即从无效到共享状态，无论其他高速缓存是否也有一个拷贝。任何有着这一存储块（处于共享状态）的高速缓存观察到这一 BusRd，但是不采取任何行动，而让主存储器提供数据响应。然而，如果一个高速缓存有着处于已修

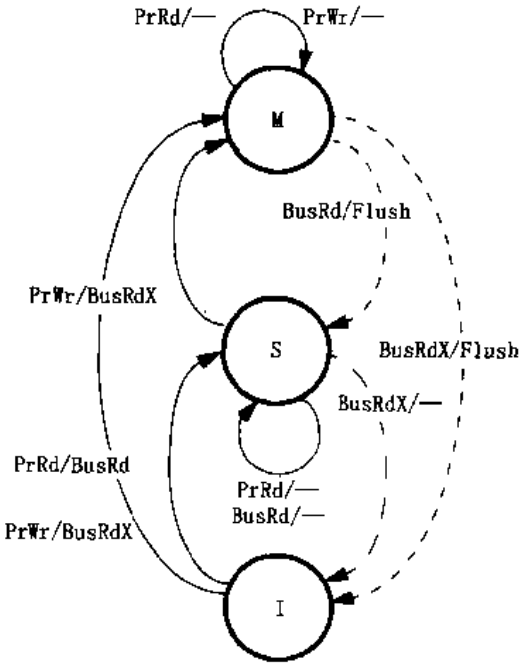


图 5-13　基本三态作废式协议。M、S 和 I 分别表示已修改、共享和无效状态。记号 *A*/*B* 表示如果控制器从处理器或总线方观察到了事件 *A*，那么除了状态改变外，它要产生一个总线事务或者动作 *B*。“—”表示空动作。由所观察到的总线事务所引起的转移用虚线弧表示，由于本地处理器动作所引起的转移用实线弧表示。如果有多个 *A*/*B* 和一个弧相联，指的是多种输入能引起相同的状态转换。为完整性起见，我们应该对每个状态和每个可观察到的事件说明相应的动作。如果有些转移没有显示出来，这里表示要么不关心，要么没有什么动作需要实施。为简单起见，替换和它们可能引起的回写在图中没有示出

改状态的存储块（只可能有一个），当它观察到总线上一个 BusRd 事务，它就必须参加这个事务（因为主存的块是过时的）。这个高速缓存将数据送到总线上替代存储器，并且将它的拷贝降级为共享状态（见图 5-13）。存储器和高速缓存都接受这个存储块。这个操作的完成可以通过在 BusRd 期间直接跨总线的高速缓存——高速缓存传送或者通过在 BusRd 事务中给出一个出错信号，产生一个写过程来更新存储器。在后者的情形，发起的高速缓存将最终重新产生它的请求，从存储器中得到这一块。（也可能让总线上的数据只被发请求的高速缓存，而不被存储器接收，使存储器内容仍然是陈旧的，但这要求有更多的状态［Sweazey and Smith 1986］。）

对一个无效的块做写操作导致一个写扑空，处理的过程首先是将整个块装入，然后修改它其中的有关字节。这个写扑空产生一个排他总线读事务，引起所有其他高速缓存拷贝被作废，从而保证发请求的高速缓存对于该存储块具有单独拥有权。由这个排他读所返回的数据块被提升到已修改状态，其中相关的字节然后得到更新。如果另一个高速缓存后来请求排他的访问，那么作为对它的 BusRdX 事务的响应，在将它的排他拷贝放到总线上后，这个块就要被作废（降级到无效状态）。

最有意思的状态转换发生在对一个共享块进行写操作的时候。如先前所讨论的，这基本上可以当作写扑空来处理，用一个排他读总线事务来获得排他的所有权；在本书中我们称它为写扑空。由于数据已经在缓存中了，在排他读里返回的数据在这种情况下可以忽略，这一点不像对无效或者不存在块进行写操作的情形。事实上，在总线协议中一种常见的减少数据流量的优化是引入一种新的事务，称为总线升级或者 BusUpgr。一个 BusUpgr 得到排他的所有权，就像 BusRdX 那样让其他拷贝作废，但它不引起主存或者其他设备进行数据响应。不管是 BusUpgr 还是 BusRdX（让我们还是假设 BusRdX），在发请求的高速缓存中的块都转换到已修改状态。在这个已修改状态下，对于那个块的其他的写操作都不产生额外的总线事务。

294
~
295

从高速缓存里替换一个存储块，逻辑上就是使那个块降级为无效的（不存在的）。一次替换因此就引起两个块的状态机在高速缓存中改变状态：被替换的块从它的当前状态到无效，新带入的块从无效（不存在）到它的新状态。后者状态的改变不能在前者之前发生，这要求在实现中采取一些措施。如果被替换的块处于已修改状态，这个从 M 到 I 的替换转移产生一个回写事务。其他的高速缓存在这个过程上不采取什么特殊行动。如果这个被替换的块处于共享或者无效状态，那么它就不会引起任何总线事务。为简单起见，替换在状态图中没有表示出来。

注意，为了能完整地说明这个协议，对于每一个状态我们必须有外向的弧，带有对应于所有可观察事件的标记（从处理器和总线一边的输入），还必须表示相对于它们的动作。当然，这些动作和状态转移有时候可能是空的，在那种情况下，可以要么显式说明空动作（见图 5-13 中的状态 S 和 M，或者可以简单地从图中略去这些弧（见状态 I）。进而，由于我们将不存在状态当作无效处理，当扑空时一个新存储块被带进高速缓存的时候，状态转移的处理就好像该高速缓存块的先前状态是无效的一样。例 5.6 说明状态转换图的解释方式。

例 5.6　利用 MSI 协议，说明图 5-3 所示情形的状态转移和总线事务。

解答：结果如图 5-14 所示。■

对于回写协议来说，在存储器被实际更新以前，一个块能被多次写。一个读操作所得的

处理器动作	在 P_1 状态	在 P_2 状态	在 P_3 状态	总线动作	提供数据的实体
1. P_1 reads u	S	–	–	BusRd	Memory
2. P_3 reads u	S	–	S	BusRd	Memory
3 P_3 writes u	I	–	M	BusRdX	Memory
4. P_1 reads u	S	–	S	BusRd	P_3 cache
5. P_2 reads u	S	S	S	BusRd	Memory

图 5-14　和图 5-3 中的处理器事务对应的三态作废协议执行情况。此图表现了在每次处理器动作结束时相关存储块的状态，所产生的总线事务以及提供数据的实体

数据可能不是来自存储器，而是来自一个高速缓存。事实上，可能是这个读而不是一个替换引起存储器被更新。除此以外，写命中不出现在总线上，因此相对于其他处理器来说，写的概念有些不同。事实上，说正在进行一个写操作，意味着这个写被“弄得可见”。对一个共享的或者无效块的写的可见是通过它所触发的总线排他读事务。写者在这个事务后将“观察到”在它缓存里的数据。这个写为其他处理器可见，是通过排他读所产生的作废，那些处理器在实际看到所写的值之前将经历一次缓存扑空。对一个已修改块的写命中对其他处理器是可见的，但同样只是在通过总线事务的一次扑空之后才能观察到。这样，在 MSI 协议中，当 BusRdX 事务发生时，对一个未修改块的写就被完成或者是可见了；对一个已修改块来说，当这个块在写者的高速缓存中被更新时，写的效果就可见了。

2．对一致性的保证

在回写协议中，由于读和写的发生都可以不产生总线事务，它是否能满足一致性条件并不是显而易见的，尽管一致性要比顺序同一性要求弱得多。让我们首先考察一致性。从前面的讨论来看，写操作效果的传播是没有问题的，因此让我们集中考虑写的串行化问题。排他读事务保证了当一个块实际写入缓存的时候，该缓存有着惟一的拷贝，就像在直写协议中的一个写事务。紧跟着的，是在缓存中的一个写，发生在缓存控制器处理任何其他总线事务之前，因此它的序对所有处理器来说（包括写者），相对于其他总线过程都是相同的。和直写协议的惟一区别是，针对一个存储单元的操作序列，不是所有写都会产生总线事务。然而，这里的关键点是，如果对某个块的两次操作的确出现在总线上了，那么只有一个处理器能完成这样的写命中；即那个最近完成对那个块排他读总线事务 w 的处理器 P。在串行化中，这个写命中的顺序因此出现在（以程序执行序）w 和下一次对于该块的总线事务之间。处理器 P 的读操作将清楚地看见它们（相对于其他的写）以这种序出现。至于另一个处理器的一个读操作，对那个块至少有一个总线事务，将读的完成和这些写命中的完成分开。这样一个总线事务保证了那个读也以同样的顺序看到这些写。这样，所有处理器的读看到所有写的顺序是相同的。

3．对顺序同一性的保证

为了分析 SC 是如何得到满足的，让我们首先看看定义本身，看看所有存储操作的一种全局的交叉执行怎么能够构造出来。如同直写缓存，关于总线的串行仲裁结果事实上定义了一个针对所有存储块的总线事务的全序，不只是关于某个块的操作序。所有的缓存控制器观察到的读和排他读总线事务的序都是相同的，并且以这个序完成作废。在相继的总线事务之间，每个处理器以程序中的次序完成一系列存储操作（读和写命中）。这样，一个程序的任何执行定义了一个自然的偏序：

存储操作 M_j 是操作 M_i 的后继，如果 1）这些操作由相同的处理器发出并且 M_j 在程序中出现在 M_i 之后，或者 2）M_j 产生一个跟在 M_i 存储操作后面的总线事务。

这个偏序从图形上看来类似于图 5-6，不同之处在于在一个段中的本地序列除了有读外还有写，并且排他读和读总线事务在建立这个序中都起了重要作用。在总线事务之间，任何来自于不同处理器的局部操作（命中）序列的交叉导致一个统一的全序。对于出现在总线事务之间同一段中的写，一个处理器将观察到其他处理器的写操作，其次序依从它所产生的总线事务，而它自己的写的序是程序中的序。

我们还可以从充分条件的角度看到 SC 是如何被满足的。写完成的检测是当排他读总线事务出现在总线上且这个写在缓存中进行时做出的。读完成的条件，提供了写操作的原子性，它被满足是因为一个读要么 1）引起一个总线事务，跟在其值正在被返回的读的后面，在这种情况下，这个写必须在这个读之前全局完成；2）由同一个处理器以程序执行序跟着这个读；或者 3）随着在完成写操作的同一个处理器上以程序执行序发生，在这种情况下，该处理器已经等着该写操作的全局完成（在可见之前）。这样，所有的充分条件都很容易得到了保证。在第 6 章讨论协议的实现时我们还要回到这个问题上来。

4. 低层设计的选择要素

为了了解在协议中某些隐含的设计决策，让我们更仔细地考察当针对一个存储块的 BusRd 被观察到时从 M 状态发生的转移。在图 5-13 中，我们转移到状态 S 并且将存储块的内容放到总线上。尽管将内容放到总线上是必须的，我们仍可以考虑转移到状态 I，这样完全放弃这个块。转移到 S 而不是 I 的选择反映了设计者认识：和另外的处理器对该存储块做写操作相比，最初的这个处理器更可能继续读这个块。直觉上看，这个认识对于大多数读数据是成立的，而且这正是许多程序中常见的。然而，一种不成立的常见情形是对于在进程之间来回传送信息的一个标记或者缓冲区：一个处理器写入，另一个处理器读出且修改，然后第一个处理器读并修改，如此等等。对一个共享计数器的累加表现了类似的跨处理器的迁移行为。这种将赌注压在读共享的问题是每一个写都要首先产生一个作废，因此增加了它的时延。的确，用在早期 Synapse 多处理器中的一致性协议采取的是不同的选择，在 BusRd 上从 M 直接转移到 I 状态，这种做法的基本出发点是认为上述那种迁移模式会经常发生。某些机器（Sequent Symmetry model B 和 MIT Alewife）试图在这种迁移型访问模式被观察到时对这种协议做些适应性改造（Cox，Fowler 1993；Dahlgren，Dubois，and Stenstrom 1994）。在本章的后面将会看到，这些选择可能影响存储系统的性能。 298

5.3.2　一种四态（MESI）回写作废式协议

如果我们考虑一个串行程序运行在多处理器上的情形，我们就可能产生对 MSI 协议的一种顾虑。事实上，这种多道程序的用法是在小规模多处理器系统上最常见的工作负载情况。当进程读进并且修改一个数据项，尽管并没有其他处理器来共享这个数据，但在 MSI 协议中还是要产生两个总线事务。第一个是 BusRd，将存储块置成 S 状态，第二个是一个 BusRdX（或者 BusUpgr），将 S 转换成 M 状态。通过增加一个状态，指出这个块是惟一的（排他的）拷贝，但没被修改，并且装入这个块时使它处于这个状态，我们能够省去后面一次总线事务，因为这个状态指出没有其他处理器的缓存也含有这一块。这个新的状态，称为干净的独

占或者非拥有的独占（甚至就简单称“独占”），指出一种在共享和已修改之间的情况　它是独占的，因此不同于共享状态，缓存能够执行写操作并且转移到已修改状态，而不需要进一步的总线事务；但它又不隐含拥有权（存储器里也有一个有效拷贝），因此不同于已修改状态，缓存在观察到关于那个块的请求时不需要答复。这种 MESI 协议的各种变形用于许多现代微处理器中，包括 Intel Pentium、PowerPC 601 以及用于 SGI Challenge 多处理器的 MIPS R4400。这个协议首先是由 UIUC 的研究人员发表的（Papamacros and Patel 1984），因而常称为 Illinois 协议（Archibald and Baer 1986）。

MESI 协议于是由四个状态构成：已修改（M）或者脏的、干净的独占（E）、共享（S）和无效（I）。M 和 I 的语义和先前的一样。E，干净的独占或者独占状态，意味着只有一个缓存有此块的一个拷贝，并且是没被修改的（即主存相应内容是最新的）。S 意味着在它们的缓存中潜在有两个或者多个处理器有这个存储块，处于未被修改的状态。所需的总线事务和动作非常类似于 MSI 协议的情形。

1．状态转换

当一个存储块首次被某个处理器读进来时，如果一个有效的拷贝存在于另一个缓存中，和通常一样，它就要以 S 状态进入这个处理器的缓存。不过，如果这时没有其他的缓存有拷贝（例如，在顺序应用程序执行的情形），它就要以 E 状态进入缓存。当这一块被同一个处理器写的时候，由于没有其他的缓存有拷贝，它就可以直接从 E 状态进入 M 状态，而不产生总线事务。如果另一个缓存在同时获得了一个拷贝，这一块的状态就要被侦听协议从 E
299 降级到 S。

这个协议对于总线的物理互连加进了一个新的要求。一个附加的共享信号（S）要被缓存控制器用来在 BusRd 上确定是否有其他缓存当前持有数据。在总线事务的地址阶段，所有缓存确定它们是否含有所要求的存储块，若如此，就给出这个共享信号。这个信号是一个线或连接，因此提出请求的控制器能够观察到是不是有其他处理器缓存含有所请求的存储块，从而能够决定是将装入的存储块置成 E 还是 S 状态。

图 5-15 表示了 MESI 协议的一个状态转换图，我们还是假设不用 BusUpgr 事务。记号 BusRd（S）表示由共享信号 S 引起的总线读事务；BusRd（$\overline{S}$）意味着 S 没有作用。BusRd 则意味着我们不关心 S 在那个过程中的值。无论处于什么状态，对于一个块的写将它提升到 M 状态，但如果它是 E 状态的话，就不会有总线事务。观察到一个 BusRd，处理器会将相关存储块的状态从 E 降级到 S，因为现在有另一个拷贝存在了。和通常一样，观察到 BusRd 将从 M 到 S 状态，还要将存储块的内容送到总线上；同样，这个块只可能被请求缓存拾取，而不是被主存，但这可能要求超出 MESI 以外附加的状态。（第五个称为被拥有的状态可以被加进来，它指出即使存在其他共享的拷贝，这个缓存（而不是主存）要在观察到相关的总线事务时负责提供数据。这就导致一种 5 状态 MOESI 协议［Sweazey and Smith 1986］）。注意，即使没有其他拷贝存在，一个块也可能处于 S 状态，这是由于拷贝可能被替换（S→I）而不需要通知其他拷贝。满足一致性和顺序同一性的论点和在 MSI 协议中的相同。

2．低层设计的选择要素

关于基于总线的协议的一个有趣的问题是，当发生一个 BusRd 事务时，如果存储器和另一个缓存都有拷贝，谁应该提供数据块。在 MESI 协议的原始（Illinois）版本中，提供数据的是缓存，这称为是一种缓存到缓存共享的技术。采用这种做法的出发点是：缓存是 SRAM，

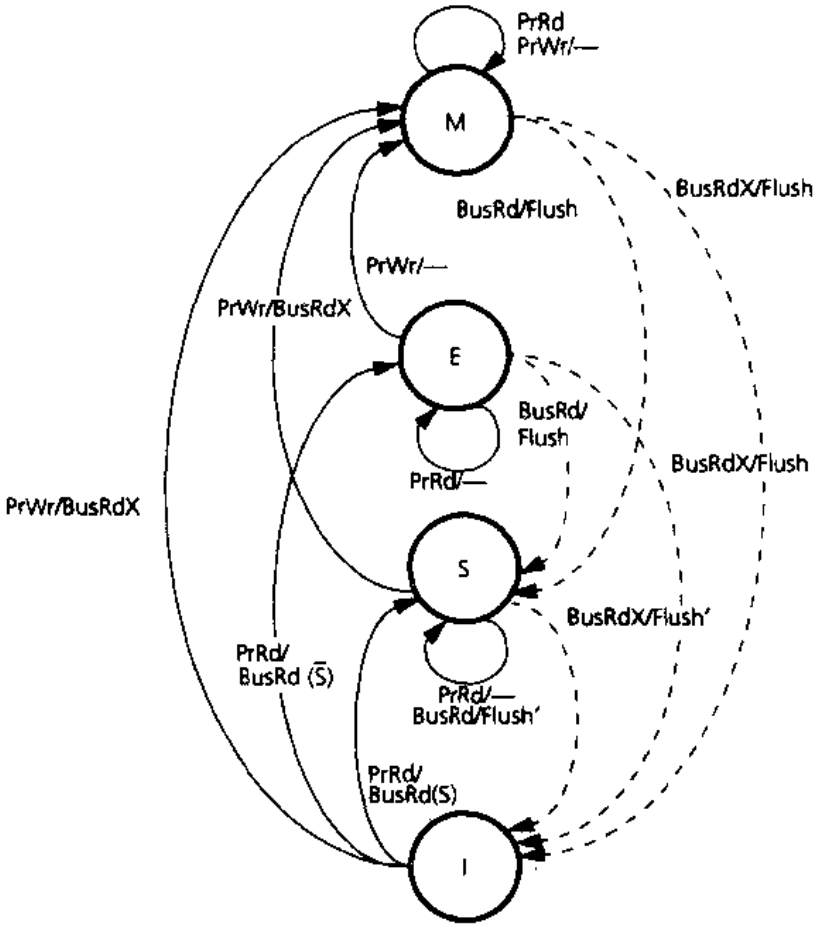


图 5-15　Illinois MESI 协议的状态转换图。MESI 是已修改（M）或脏的、独占（E）、共享（S）和无效（I）状态的英文缩写。这里用的记号和图 5-13 相同。E 状态有助于在串行程序数据不共享时降低总线上的流量。只要可能，Illinois 版本的 MESI 协议都让高速缓存，而不是存储器为 BusRd 和 BusRdX 事务提供数据。由于多个处理器可能在它们的缓存中有相同的存储块，我们需要选出一个来向总线提供数据。图中 Flush′只是针对相关的那个处理器而言的；其他处理器做它们常规的动作（作废或没动作）。一般来说，状态转换图中的 Flush′指出相关的存储块被清除，其条件是用到了缓存到缓存的共享，并且清除是由负责提供数据的缓存来实行的

存储器是 DRAM，前者能更快地提供数据。然而，这个优势在现代基于总线的机器中不一定存在，让另一个缓存来提供数据可能比直接从存储器提供的代价要更高。缓存到缓存共享也增加了基于总线协议的复杂性：主存必须等待，直到它能肯定没有缓存将提供数据后才能驱动总线，并且如果多个缓存都有这个数据，那么就要有一个选择算法来确定哪一个应该提供数据。另一方面，这个技术对于物理上分布存储的多处理器系统是有用的（如我们在第 8 章会看到的），因为从近处的缓存获取数据要比从远处的存储器快得多。特别是，在由 SMP 节点构成的网络型机器中，数据请求者的 SMP 节点中的缓存可能提供数据，这种缓存到缓存共享的方式可能就特别重要了。基于这种考虑，斯坦福 DASH 多处理器（Lenoski et al. 1993）用的就是这种缓存到缓存的传送机制。

5.3.3　一种四态（Dragon）回写更新式协议

现在让我们考察一种回写缓存上基于更新的协议。这个协议最初是由 Xerox PARC 的研

300 301 究人员提出来的，背景是他们的 Dragon 多处理器系统（McCreight 1984; Thacker, Stewart, and Satterthwaite 1988）。该协议的一个增强版本用在 Sun SparcServer 多处理器系统中（Catanzaro 1997）。

Dragon 协议由四个状态构成：干净的独占（E）、干净的共享（Sc）、共享已修改（Sm）和已修改（M）。干净的独占（简称独占）的意图和含义与前面相同：只有一个缓存（这个缓存）有这一存储块的一个拷贝，并且还没有被修改（即，主存的内容也是有效的）。干净的共享意味着，潜在地有两个或者多个缓存（包括这个）有这一存储块，主存可能有但不一定是最新的。共享已修改意味着潜在地有两个或者多个缓存有这一存储块，主存不是最新的，并且当这个块要被从缓存中替换出去时，这个缓存有责任对主存进行更新（即这个缓存是拥有者）。一个存储块在一个时间只能在一个缓存里是 Sm 状态。不过，一个存储块在一个缓存是 Sm 状态，同时在其他缓存是 Sc 状态的情况是相当可能的，或者没有缓存是 Sm 状态，但有一些是 Sc 状态。这就是为什么当在一个缓存里发现某个存储块是 Sc 状态时，不能断定存储器里的内容是否是最新的道理；还取决于这个存储块是否在其他缓存里处于 Sm 状态。和先前一样，M 表示独占的拥有权：这个存储块已经修改了（脏）并且只是出现在这个缓存里，主存的内容是陈旧的，一旦发生替换，这个缓存有责任提供数据来更新存储器的内容。注意，这里没有在前面的协议中具有的显式的无效（I）状态。这是因为 Dragon 是一个基于更新的协议；它总是保持缓存中块的内容是最新的，因此如果标记匹配成功，则总是可以用缓存中的数据。然而，如果一个存储块根本就不在缓存中，它就可以被想像为一种特别的无效或者不存在状态。○

处理器请求，总线事务以及 Dragon 协议的动作都类似于 Illinois MESI 协议。处理器仍然假设为只发出读（PrRd）和写（PrWr）请求。不过，由于没有一个无效状态，为了说明当标记匹配不成功时的动作，加上两种请求类型：处理器读扑空（PrRdMiss）和写扑空（PrWrMiss）。至于总线事务，有总线读（BusRd）、总线回写（BusWB）和一个新的称为总线更新（BusUpd）的事务。BusRd 和 BusWB 事务有着通常的语义。BusUpd 事务将这个处理器写的特定的字（或者字节）广播到总线上，从而所有其他处理器的缓存能更新它们自己。这里广播的只是被修改的内容，而不是整个存储块。这种处理方式是指望总线带宽能得到更高效的利用（见习题 5.4，其中谈到这一种愿望不一定总是奏效的）。如同 MESI 协议，为了支持 E 状态缓存控制器需要用到一个共享信号（S）。最后，协议提出的惟一新要求是，当一个相关的 BusUpd 事务将数据广播到总线上时，缓存控制器要能够用总线上的内容来更新一个本地缓存的存储块（标记为 Update 动作）。

302

1. 状态转换

图 5-16 所示的是 Dragon 更新协议的状态转换图。按照处理器为中心的观点，能够用在一个缓存发生读扑空、写（命中或者扑空）或者替换所采取的动作（读命中没有动作）来解释这个图。

- 读扑空。产生一个 BusRd 事务。取决于共享信号（S）的状态，读到的这个存储块装入缓存后，被置成 E 或者 Sc 状态。如果这个块在某个其他缓存里是 M 或 Sm 状态，

○ 逻辑上讲，还有另一个状态，但它只是用来自启这个协议。一个“扑空模式”位用在每条缓存线上，强制导致一次扑空。初始化软件在开始将数据读入缓存中来时，这一“扑空模式”位是激活的，以保证第一次读产生扑空。在这第一次扑空后，“扑空模式”位关闭，缓存正常运行。

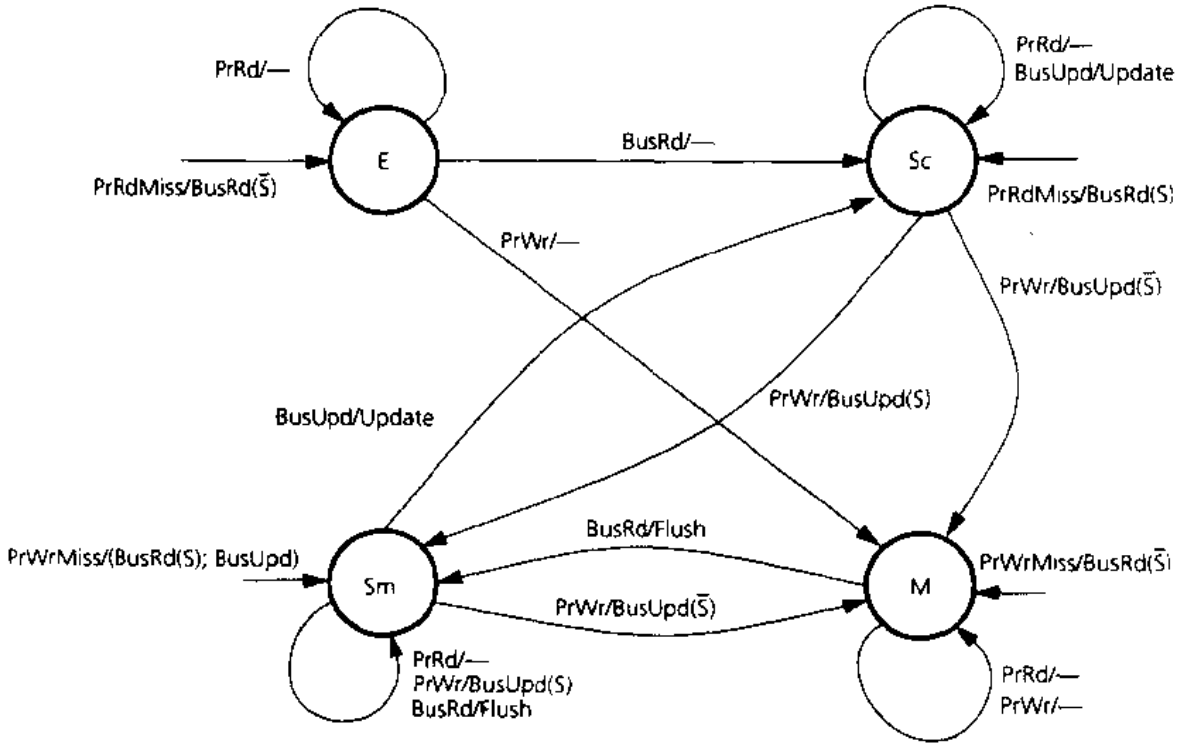


图5-16 Dragon更新协议的状态转换图。四个状态是独占（E）、干净的共享（Sc）、已修改共享（Sm）、和已修改（M）。因为更新协议总是保持使缓存中的数据为最新，这里没有无效状态（I）

那么缓存就要给出这个共享信号并且将最新的数据提供到总线上。该存储块以 Sc 状态装入本地缓存。如果它在其他缓存里是 M 状态，它就要改变为 Sm。如果它在其他缓存是 Sc 状态，则存储器提供数据，它以 Sc 状态装入。如果所有其他缓存都没有相应拷贝，则共享信号线保持为无效的，数据由主存提供，这个块以 E 状态装入本地缓存。

- 写。如果这个块在本地缓存中的状态是 M，那么不需要有什么动作。如果是 E，那么就变到 M 状态，也没有什么动作。然而，如果是在 Sc 或者 Sm 状态，就要产生一个 BusUpd 事务。如果其他某个缓存有一个拷贝，它们就要置共享信号，更新它们拷贝中的相关字节，并且如果必要的话要将状态改为 Sc。本地缓存也更新它的拷贝，如果必要的话要将状态改为 Sm。主存不被更新。如果没有其他缓存有该数据的拷贝，则共享信号保持无效，本地缓存更新状态变为 M。最后，如果在写操作的时候这个存储块在缓存中不存在，这个写就简单地处理为一次读扑空过程，跟着是一个写过程。这样，就产生了第一个 BusRd。如果这个块也在其他缓存中，就要产生一个 BusUpd，存储块以 Sm 状态装入本地缓存；否则，它就以 M 状态装入。 303
- 替换。在一次替换时（图中没有显示相应的弧），只是当相关的存储块处于 M 或者 Sm 状态上，我们才通过一个总线过程将存储块写回存储器。如果它是 Sc 状态，那么或者某个其他缓存有它在 Sm 状态，或者谁也没有。在这种情况下它在存储器中就是有效的。

针对一个我们已经熟悉情形，例 5.7 解释了这些状态转换过程。

例 5.7 用 Dragon 更新协议，针对图 5-3 所示的情形，指出状态转换和总线事务。

解答：结果如图5-17所示。我们可以看到在更新协议中，关于处理器的动作3和4只有一个字被送到总线上，整个存储块在基于作废的协议中被传送两次。当然，我们也可以构造一种情况，其中作废协议的表现要好于更新协议。在5.4节，我们将讨论详细的权衡。■

	处理器动作	在 P_1 状态	在 P_2 状态	在 P_3 状态	总线动作	提供数据的实体
1	P_1 reads u	E	–	–	BusRd	Memory
2.	P_3 reads u	Sc	–	Sc	BusRd	Memory
3	P_3 writes u	Sc	–	Sm	BusUpd	P_3 cache
4	P_1 reads u	Sc	–	Sm	null	–
5.	P_2 reads u	Sc	Sc	Sm	BusRd	P_3 cache

图5-17　Dragon更新协议对于图5-3中处理器的操作所产生的动作。此图表明在每次处理器动作结束时相关存储块的状态所产生的总线事务以及提供数据的实体

2. 低层设计的选择要素

同样，在这个协议中也有许多隐含的设计抉择。例如，共享已修改状态是有可能取消的。事实上，用在DEC Firefly多处理器系统中的更新协议就是这么做的。这里的道理是每当BusUpd事务发生时，和其他持有该存储块的缓存一道，主存也可以更新它的内容；因此，干净的共享就足够了，已修改共享就不再需要。而Dragon协议是基于这样的假设：更新SRAM缓存要比更新DRAM主存快得多，因此在所有BusUpd事务上等待主存的更新就不合适了。另一个微妙的因素和缓存替换时所取的动作有关。当一个干净的共享存储块被替换时，应该通过一个总线事务让其他缓存得知这个替换吗？这样做的理由可能是，如果只有一个缓存持有该存储块的一个拷贝，它就可以将其状态改为独享的或者已修改的。这样做的好处是，这个由替换引起的总线事务可能不在一次存储器操作的关键路径上，而它所省下来的后来的总线事务可能就在关键路径上。

由于在更新协议中所有写操作都会出现在总线上，对于一条原子型总线来说，写串行化、写完成的检测以及写操作的原子性都是相当明确的，就像在直写缓存的情形那样。然而，对于含有作废和更新两种功能的协议，我们必须考虑许多精细的实现问题和竞争条件，即便是原子型总线和单级缓存也是如此。将在第6章中讨论这一层次的协议和硬件设计以及带有流水线总线、多级缓存层次的更实际的情形、还有能够变序完成存储操作的硬件技术。尽管如此，基于到目前为止所考虑的状态图的层次，也能够量化许多协议的权衡考虑。

5.4　关于协议设计中若干折中的评估

如同任何其他复杂系统，一个多处理系统的设计要做出许多相互关联的决定。即使处理器已经被选定了，也必须决定系统所要支持的最大处理器的个数、缓存层次结构的各种参数（例如，层次数，每级缓存的大小、关联度、块的大小以及是采用直写还是回写策略等等）、总线的设计（例如，数据和地址总线的宽度、总线协议）、存储系统的设计（例如，是否采用多模块交叉存储技术、存储模块的宽度、内部缓存的大小）、还有I/O子系统的设计。这其中许多因素和单处理器系统所要考虑的类似（Smith 1982），但矛盾可能进一步突出了。例如，直写缓存对于多处理器来说可能是个不好的选择，因为总线带宽是由多个处理器共享的，存储器可能需要在更大程度上交叉，因为它要服务于多个处理器的缓存扑空。较大的缓存关联度在减少冲突扑空（要产生总线传输）方面也可能是有用的。

缓存一致性协议对于多处理器来说是一个关键的新设计考虑。它包括协议类型（作废还是更新）、协议状态和动作以及低层实现的权衡。协议的决定和所有其他设计因素都有关系。另一方面，协议影响系统部件的时延和带宽特征被强调的程度；另外，除存储组织和通信体系结构外，性能特征也影响协议的选择。如第 4 章所讨论的，这些设计的决定需要针对实际程序的行为来评估。这样的评估在 20 世纪 80 年代后期是很时兴的，尽管用的是一些不成熟的并行程序作为负载（Archibald and Baer 1986；Agarwal and Gupta 1988；Eggers and Katz 1988，1989a，1989b）。 304 305

在真实的系统设计中要做出决定，部分是艺术，部分是科学。艺术源于过去的经验、直觉和设计者的审美观，科学是基于工作负载驱动的评估。设计目标通常就是满足代价和性能指标，达到一个平衡的系统，从而没有个别资源成为性能瓶颈，同时每个资源只有很小的富裕能力。这一节通过第 4 章介绍的负载驱动评估方法，讨论一些关键的协议权衡。

5.4.1　方法论

我们的基本策略如下。如同第 4 章所描述的，让工作负载在一个多处理器体系结构的模拟器上执行。通过观察在模拟器中遇到的状态转换，能够确定各种事件的频率，诸如缓存扑空和总线事务。然后可以从其他设计参数，例如时延和带宽需求的角度来评估协议选择的效果。

按照第 4 章的方法选择参数，通过一组程序在四状态 Illinois MESI 协议上的执行，本节首先建立起基本的状态转换特性。然后解释如何用这些测得的结果，针对上述协议例子来获得一个关于设计权衡初步的量化分析，诸如 MESI 协议中独占状态的使用和 S→M 转换中 BusUpgr 事务（而不是 BusRdX 事务）的使用。本节还解释更传统的设计问题，诸如缓存块的大小（既是一致性也是通信的粒度）如何影响应用对于时延和带宽需要的。为了理解这个效果，我们将缓存扑空分为几种情况，诸如冷启动扑空、容量扑空和共享扑空，考察缓存块的大小对于不同情况的效果，并从应用特征的角度来解释结果。最后，也是从时延和带宽影响的角度，通过这种对应用的理解，来解释在基于作废和基于更新协议之间的权衡。

本节的分析是基于各种重要事件的频率，而不是基于它们所花的绝对时间（即性能）。这种做法在缓存体系结构的研究中是通行的，因为这样的结果不依赖于具体的系统实现和工艺的假设。然而，它应该只被看作是一种初步的分析，这是由于许多可能在真实系统中影响性能权衡的细节因素被抽象掉了。例如，记录状态转换的情况为计算扑空率和总线事务提供了一个方式，但如果没有真实的时延、开销和占用度的值，我们就不能将这些量转换成作用在系统上实际的带宽需求。为了得到带宽需求的一个估计，我们可能人为地假设每一次引用都花同样多的周期数完成。然而，带宽需求本身并不直接对应性能，只是通过增加竞争扑空 306
的代价发生间接的影响。竞争是很难估计的，因为它不仅取决于所用的时序参数，还取决于流量的突发性，而这些在频率测量中是不可能捕获的。竞争、时序，以及性能也受到和低层硬件结构（例如队列和缓冲）相互作用和策略的影响。

用在本节的模拟不涉及竞争。它们只用一个简单的 PRAM 代价模型：所有存储操作的完成时间都假定是相同的（这里是单周期），无论它们是命中还是不命中缓存。对于这一点有三个主要原因。首先，这里关注的主要是理解协议固有的行为和与频率有关的权衡，不在于性能。第二，由于对不同的缓存块的大小和组织进行实验，不管具体的参数选择如何，希望从模拟器运行应用程序产生的引用交织是相同的；也就是说，所有协议和块的大小应该看到

相同的引用轨迹。由于这里用的是执行驱动，而不是踪迹驱动，只有使得模拟中所有的存储操作有同样代价，这才是有可能。否则，如果一次引用对于一个小缓存扑空，但对于一个大缓存命中的话，那么它在两种情形下表现出两种不同的延迟。于是，要确定哪些效果是协议固有的，哪些是由于所选择参数所导致的，就不是件容易的事情。第三，包含竞争效果的真实模拟要花更多的时间。用这种简单模型来采集频率的缺点是这种时序模型可能影响某些我们观察到的值；不过，这个影响对于我们研究的应用来说不大。

这里用来解释问题的工作负载是6个并行程序（来自SPLASH-2）和在第3、4章描述的一个多道程序。这些并行程序以批处理模式运行，对机器是独占的方式，并且在模拟中不包括操作系统的活动。所用的应用程序的数量相对较小，但这些应用主要是用来做解释用的，如第4章所描述的那样；这里所强调的是所选择的程序要能代表重要的计算类型，有着广泛变化的特性。这些应用中基本操作的频率如表5-1所示。现在更进一步来研究它们，以评价在缓存一致性协议中的设计权衡。

表5-1　由应用程序发生的每1 000次数据存储访问所导致的状态转换

应用			至				
			NP	I	E	S	M
Barnes-Hut	从	NP	0	0	0.0011	0.0362	0.0035
		I	0.0201	0	0.0001	0.1856	0.0010
		E	0.0000	0.0000	0.0153	0.0002	0.0010
		S	0.0029	0.2130	0	97.1712	0.1253
		M	0.0013	0.0010	0	0.1277	902.782
LU	从	NP	0	0	0.0000	0.6593	0.0011
		I	0.0000	0	0	0.0002	0.0003
		E	0.0000	0	0.4454	0 0004	0.2164
		S	0.0339	0.0001	0	302.702	0.0000
		M	0.0001	0.0007	0	0.2164	697.129
Ocean	从	NP	0	0	1.2484	0.9565	1.6787
		I	0.6362	0	0	1.8676	0.0015
		E	0.2040	0	14.0040	0.0240	0.9955
		S	0.4175	2.4994	0	134.716	2.2392
		M	2.6259	0.0015	0	2.2996	843.565
Radiosity	从	NP	0	0	0.0068	0.2581	0.0354
		I	0.0262	0	0	0.5766	0.0324
		E	0	0.0003	0.0241	0.0001	0.0060
		S	0.0092	0.7264	0	162.569	0.2768
		M	0.0219	0.0305	0	0.3125	839.507
Radix	从	NP	0	0	0.004746	3.524705	11.41111
		I	0.130988	0	0	1.108079	4.57868
		E	0.000759	0.002848	0.080301	0	0.00019
		S	0.029804	1.120988	0	178.1932	0.817818
		M	0.044232	11.55127	0	4.03157	802.282
Raytrace	从	NP	0	0	1.3358	0.15486	0.0026
		I	0.0242	0	0.0000	0.3403	0.0000
		E	0.8663	0	29.0187	0.3639	0.0175
		S	1.1181	0.3740	0	310.949	0.2898
		M	0.0559	0.0001	0	0.2970	661.011

（续）

应用			至				
			NP	I	E	S	M
Multiprog User Data References	从	NP	0	0	0.1675	0.5253	0.1843
		I	0.2619	0	0.0007	0.0072	0.0013
		E	0.0729	0.0008	11.6629	0.0221	0.0680
		S	0.3062	0.2787	0	214.6523	0.2570
		M	0.2134	0.1196	0	0.3732	772.7819
Multiprog User Instruction References	从	NP	0	0	3.2709	15.7722	0
		I	0	0	0	0	0
		E	1.3029	0	46.7898	1.8961	0
		S	16.9032	0	0	981.2618	0
		M	0	0	0	0	0
Multiprog Kernel Data References	从	NP	0	0	1.0241	1.7209	4.0793
		I	1.3950	0	0.0079	1.1495	0.1153
		E	0.5511	0.0063	55.7680	0.0999	0.3352
		S	1.2740	2.0514	0	393.5066	1.7800
		M	3.1827	0.3551	0	2.0732	542.4318
Multiprog Kernel Instruction References	从	NP	0	0	2.1799	26.5124	0
		I	0	0	0	0	0
		E	0.8829	0	5.2156	1.2223	0
		S	24.6963	0	0	1 075.2158	0
		M	0	0	0	0	0

注：除 Multiprog 用 8 个处理器外，这些数据都假设 16 个处理器、1 MB 四路组相联高速缓存、64 字节高速缓存块，Illinois MESI 一致性协议。

5.4.2　在 MESI 协议下的带宽需求

如第 4 章所述，我们从容量为 1 MB、每个处理器只有一级缓存的情况开始。对于问题的缺省规模来说，这种缓存能够容纳其重要的工作集，对于所有应用这是现实的情形。我们用四路组相联（LRU 替换）以减少冲突扑空并取高速缓存块的大小为 64 字节，这是实用的。让这些工作负载通过模拟 Illinois MESI 协议的高速缓存模拟器，我们得到如表 5-1 所示的状态转换频率。其中数据指的是由处理器发出的每 1000 次访问所导致的状态转换数。注意，在这个表中，我们引入了一个新的状态 NP（未现态）。它帮助表现当缓存扑空时的状态转换，即一个缓存块被替换（创建一个从 I、E、S、M 到 NP 的转换），一个新的块被带进来（创建一个从 NP 到 I、E、S、M 的转换）。尽管我们表现的是每 1000 次访问的情形，但状态转换的总和可以大于 1000，这是因为有些访问会引起多个状态转换。例如，一次写扑空，除了在其他缓存中引起的作废转换（I/E/S/M→I）外，还可以在本地缓存引起两个转换（例如，对于旧缓存块是 S→NP，对于新进来的缓存块是 NP→M）。这个状态转换频率数据对于回答“如果……怎样”类型的问题是很有用的。例 5.8 告诉我们如何来确定这些负载带给存储系统的带宽需求。

307 ~ 309

例 5.8　假设整数密集型应用以每处理器 200 MIPS 持续运行，浮点密集应用为 200 MFLOPS。假设缓存块的传送涉及 64 字节的数据（在数据总线）和每个总线事务涉及 6 字节的命令和地址（在地址总线），问每个处理器产生的流量是多少？

解答：第一步是计算每条指令的流量。对于每个可能的状态转换，我们确定所进行的总

线动作，从而得知和每一事务相关联的流量。例如，M→NP 转换指示的是，由于扑空一个已修改的高速缓存块要被回写。类似地，S→M 转换表示一个更新请求要被放到总线上。作为对一种总线事务（例如，M→S 或 M→I 事务）的响应，将一个已修改的缓存块刷新也导致一个 BusWB 事务。所有可能转换的总线事务如表 5-2 所示。除 BusUpgr 只是产生地址流量外，所有事务都产生 6 字节地址总线流量和 64 字节数据流量。[○]■

表 5-2　Illinois MESI 协议中响应状态转移的总线活动

		至				
		NP	I	E	S	M
从	NP	—	—	BusRd	BusRd	BusRdX
	I	—	—	BusRd	BusRd	BusRdX
	E	—	—	—	—	—
	S	—	—	Not possible	—	BusUpgr
	M	BusWB	BusWB	Not possible	BusWB	—

我们现在能计算所产生的流量。用表 5-2，我们可以将表 5-1 中的每 1000 次存储访问的状态转移变换到每 1000 次存储访问的总线事务，并通过乘以每个事务的流量，将这些转换为地址和数据流量。然后，用表 4-1 中存储访问的频率，我们可以将它转换为每条指令的流量。最后，乘以假设的处理速率，我们就得到关于每个应用的地址和数据的带宽需求。这个计算方法对于每个应用的结果对应于图 5-18 直方图中每一组的最左项。

先前例子中的计算给出了平均带宽需求，但假设了总线带宽足以使处理器以全速来执行它们的任务（实际上，带宽局限可能使处理器和事件都慢下来，从而反过来导致单位时间的流量降低）。这个计算，对于得到一个系统在不饱和的前提下所能支持的处理器数提供了一个有用的基础。例如，像 SGI Challenge 这样的机器，有 1.2 GBps 数据带宽，对于我们的那些应用来说（除 Radix 外），总线提供了足以支持 16 个处理器的平均带宽。一种典型的经验规则是留出 50%的“余量”，来应付数据传送的突发性。如果 Ocean 和 Multiprog 工作负载被排除在外，这个总线则可能支持到 32 个处理器。如果带宽不足以支持某个应用，这个应用的执行就要慢下来。这样，我们可以预料到 Radix 加速比的曲线随处理器数的增加会很快扁平下来。通常，一个多处理器系统用于各种不同的工作负载，其中许多对带宽的要求不高，因此设计者要选择支持一种处理器规模，能在要求最高的应用上使总线处于充分利用的状态。

5.4.3　协议优化的影响

给定这种基本的设计要点，我们就能在常用机器参数假设下来评估协议的权衡了。下面就是一个例子。

例 5.9　在本章里，我们已经描述了两种作废式协议——基本三态 MSI 协议和 Illinois MESI 协议。关键的区别在于 MESI 协议包含了独占状态。这个 E 状态对带宽的节省到底有多

○ 对于 Multiprog 工作负载，为了提高模拟的速度，在把指令传送到 1 MB 的统一的指令和数据高速缓存之前用了一个 32 KB 的指令高速缓存，作为一个过滤器。对于指令访问的状态转换频率的计算只是基于在 L_1 指令缓存中扑空的访问。这种过滤不影响我们计算数据流量的方式，但它意味着指令流量的计算是不同的。除此以外，对 Multiprog，我们分别给出内核指令、内核数据访问、用户指令和用户数据访问。一次访问可能产生多种用户数据和内核数据类型的状态转移。例如，如果一条内核指令扑空引起一个已修改的用户数据块的回写，那么我们将有针对内核指令的转移 NP→E/S 以及针对用户数据访问的转移 M→NP。

大呢？

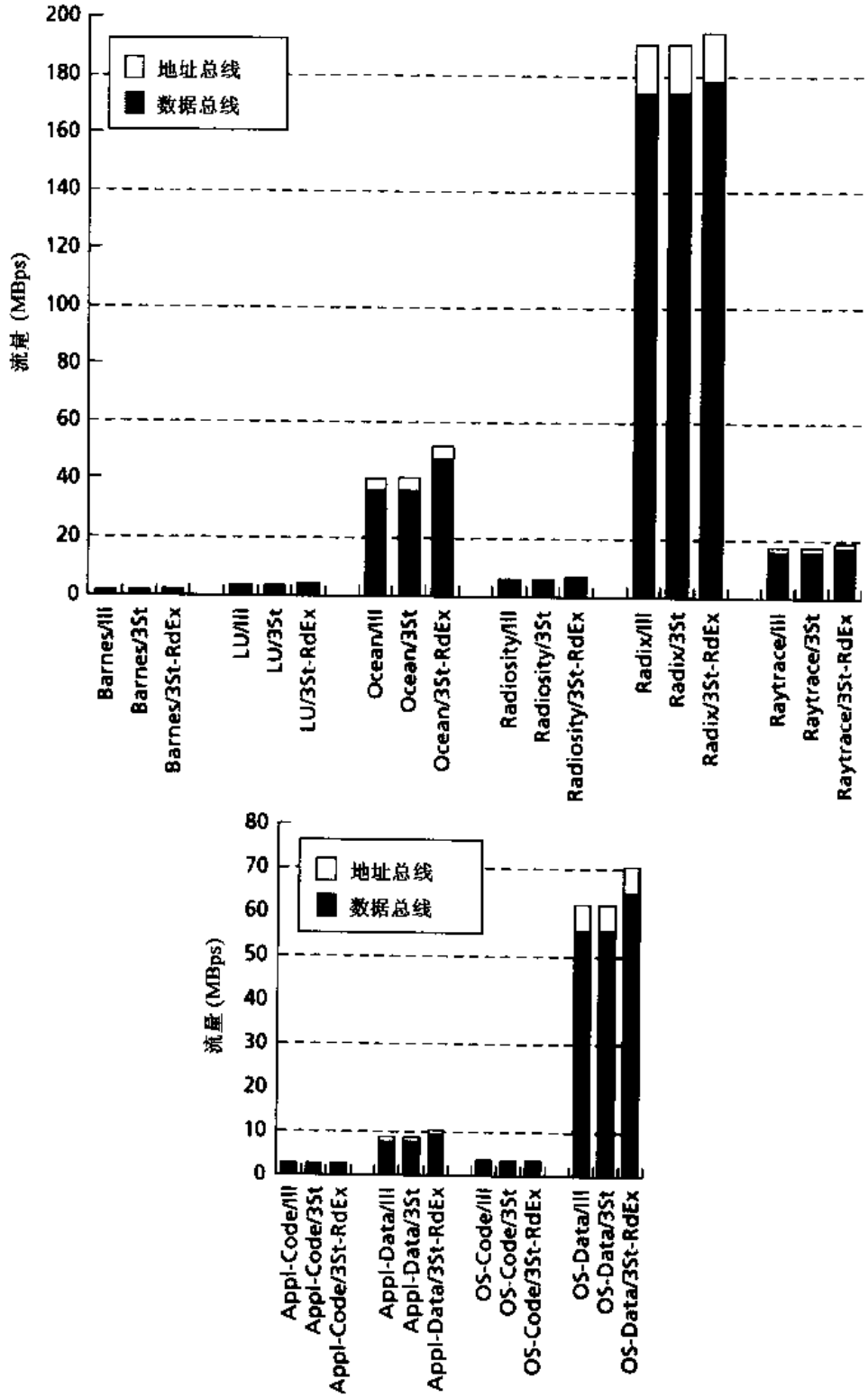


图 5-18　假定处理器计算性能为 200 MIPS/MFLOPS，且每处理器有 1 MB 高速缓存，这里表现的是各种应用从每个处理器产生的带宽需求。上边的直方图表示并行程序的数据，下边的图表示从 Multiprog 的模拟得到的数据。数据流量和地址（包括命令）流量分别表示。每一组中最左边的项表示 Illinois MESI 协议（Ill）；中间项表示的是基本三态作废协议（3St），即没有 E 状态；最右边的项表示在 S→M 转换时用 BusRdX 而不是 BusUpgr 的三态协议（3St-RdEx）

解答：E状态的主要优点是当从E→M转换时没有流量产生。而一个三态协议会产生一个BusUpgr事务，以获得对于存储块的独占拥有权。计算带宽的节省，我们只要在表5-2中对于E→M转换放一个BusUpgr，然后和以前一样重新计算流量，图5-18各组的中项就是所得到的带宽需求。■

310
312

例5.9说明了对于一个比较复杂的设计，直觉的推理可能经不起工作负载的量化测试。这对于Multiprog也是对的，尽管它基本上是由顺序程序构成，看起来应该是受益最大的。这种收益微不足道的主要原因是表5-1中E→M转换的份额相当小（即由于读扑空以独占状态装载的存储块并不经常在同一状态被写）。另外，在三态协议中对于S→M转换所需要的BusUpgr事务，只是用到6个字节的地址流量，而没有数据流量。例5.10考察了BusUpgr事务的优点。

例5.10 回顾即使在三态MSI协议中，一个对于共享状态存储块的写操作在总线上会产生一个BusUpgr请求，而不是BusRdX。这是节省带宽的，因为对BusUpgr没有数据传送的需要。但如我们将要看到的，它使实现复杂了。这里的问题是，这种额外的复杂性能使我们节省多少带宽？

解答：要计算不太复杂的实现和一个三态协议的带宽，我们只要在表5-2中对E→M和S→M转换放上BusRdX（在三态MSI协议，这都是S→M转移），然后重新计算带宽数　结果由图5-18直方图各组最右项所示。尽管对大多数应用来说，带宽的差别是不大的，但Ocean和Multiprog内核数据访问表明，这个差别可能大到10%～20%。■

在带宽需求方面的差别对性能的影响取决于总线事务是如何实现的。然而，这种高层分析指出了在何处需要有更细致的评估。

最后，就像在第4章所讨论的，对于所用的输入数据集的大小，在较小的缓存上也运行Ocean、Raytrace和Radix应用是重要的，因为那可以模拟重要的工作集在缓存层次中放不下的情形。我们在此用64KB的高速缓存，对于这些问题的规模，除了最大的工作集外，它能适应所有其他的工作集。对于这种情形粗略的状态转换数据如表5-3所示，图5-19示出按处理器的带宽需求。就像所看到的，如果由于容量性扑空导致一个关键的工作集不能放到处理器的缓存中，对总线带宽的需求可能大大增加。1.2 GBps的总线现在对于Ocean和Radix只能勉强支持4个处理器，而对Raytrace可支持到16个处理器。

表5-3 在高速缓存容量较小的情况下，应用程序每发出1000次存储访问所导致的状态转换

应用			至				
			NP	I	E	S	M
Ocean	从	NP	0	0	26.2491	2.6030	15.1459
		I	1.3305	0	0	0.3012	0.0008
		E	21.1804	0.2976	452.580	0 4489	4.3216
		S	2.4632	1.3333	0	113.257	1.1112
		M	19.0240	0.0015	0	1.5543	387.780
Radix	从	NP	0	0	9.440787	2.557865	27.36084
		I	4.354862	0	0.00057	0.157565	1.499903
		E	8.148377	0.001329	140.9295	0.012339	0.126621
		S	3.825407	0.481427	0	102.4144	0.484464
		M	23.03084	5.629429	0	2.069604	717.1426

（续）

应用			至				
			NP	I	E	S	M
Raytrace	从	NP	0	0	7.2642	3.9742	0.1305
		I	0.0526	0	0.0003	0.2799	0.0000
		E	6.4119	0	131.944	0.7973	0.0496
		S	4.6768	0.3329	0	205.994	0.2835
		M	0.1812	0.0001	0	0.2837	660.753

注：这里的数据假设 16 个处理器、64 KB 四路组相联高速缓存、64 字节的高速缓存块，Illinois MESI 一致性协议。

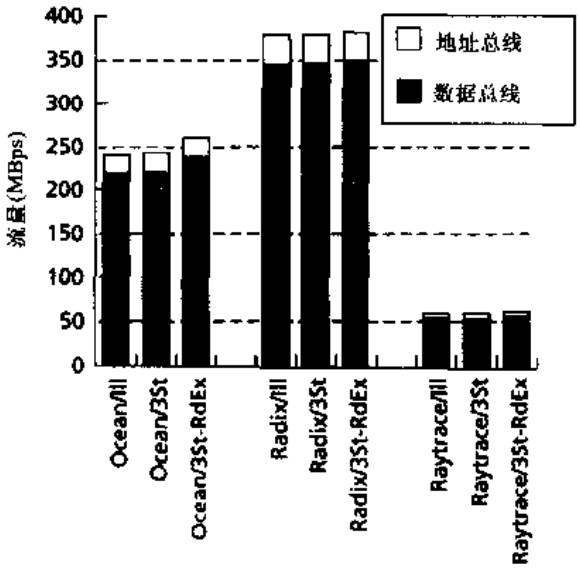


图 5-19　假定处理器计算性能为 200 MIPS/MFLOPS，且每处理器有 64 KB 高速缓存，这里表现的是各种应用从每个处理器产生的带宽需求。数据流量和地址（包括命令）流量分别表示。每一组中最左边的项表示 Illinois MESI 协议；中间项表示的是基本三态作废协议，即没有 E 状态（如 5.3.1 节所述）；最右边的项表示在 S→M 转换时用 BusRdX，而不是 BusUpgr 的三态协议

5.4.4　高速缓存中存储块大小的权衡

在所有现代计算机中，缓存的组织是一个关键的性能因素，对于多处理器尤其如此。在单处理器场合，缓存扑空典型地被分为“3C”，即强制（compulsory）扑空，容量（capacity）扑空和冲突（conflict）扑空（Hill and Smith 1989；Hennessy and Patterson 1996）。强制性扑空，也称为冷启动扑空，发生在处理器对一个存储块的首次引用时。容量扑空是由于在程序执行过程中处理器所要引用的所有块没法放到高速缓存中（即使是全相联），因此某些块要被替换，后来还要再被访问。冲突或碰撞扑空发生在非全相联的缓存中，当一个程序引用的应映射到同一缓存块的集合在一组中装不下的时候；它们在全相联缓存中是不会发生的。许多研究成果都是针对缓存大小、相联度和缓存块大小是如何影响这几种扑空的。

从系统结构来说，容量扑空可以通过加大缓存来减少，冲突扑空可以通过增加缓存的相联度或者增加可以映射的线数来减少（增加缓存规模，减少块大小）。冷启动扑空只能通过增加块的大小来减少，因为那样一次冷启动扑空将带进来较多的数据，那些数据可能紧接着被访问到。在单处理器中缓存设计的挑战性在于这些因素相互影响。例如，对于固定的缓存容量，增加块的大小将减少块的数量，因此减少冷启动扑空可能的代价是增加冲突扑空。同样，在缓存组织上的变化可能影响扑空所带来的惩罚或者命中时间，从而可能影响到处理器的周期。

313
↓
314

缓存一致性多处理器引入第四种扑空：一致性扑空。它们发生在数据块在多个缓存中的共享。这类扑空又分两种：真共享扑空和伪共享扑空。真共享扑空发生在由一个处理器写的数据字要被另一个处理器读或者写的时候。伪共享扑空指的是，不同的数据字被不同的处理器访问，但它们处于同一个存储（缓存）块中，并且至少有一个访问是写。缓存块的大小不仅是对主存进行数据访问的单位（粒度），它也是相容性的粒度。即一个处理器的写，另外处理器缓存中的整个块都被作废，而不仅是写操作涉及的那个存储字。

更精确地讲，真共享扑空发生在一个处理器向缓存中的某些字做写操作时，作废另一处理器中缓存的相应块，在那之后后者要从修改的字中读。之所以称为是“真”共享扑空，是因为这个扑空真正涉及到了共享的数据；无论和机器组织或者粒度有什么相互作用，对这种扑空的处理对于程序的正确性是必须的。另一方面，当一个处理器向一个缓存块中写入某个字，然后另一个处理器读（或者写）同一缓存块的另一个字，即使没有有用的数据在处理器之间交流，块的作废和随后的缓存扑空也会发生。这样的扑空就称为伪共享扑空（Dubois et
315
al. 1993）。随着缓存块的增大，处于同一存储块中不同的变量被不同的处理器访问的概率增加。如果对这些变量的访问是写操作，则伪共享扑空的可能性也就增加。如果存储块的大小只是一个字，仍然可能有真共享扑空，但不会有伪共享扑空。工艺的进步趋向于较大的缓存块（例如，DRAM 的组织和访问模式以及需要通过分摊开销以获得高的数据传送带宽），因此理解伪共享扑空的潜在影响以及如何能避免它们是重要的。

真共享扑空是给定的并行分解和分配所固有的，因此如同冷启动扑空，减少它们的惟一办法是增加存储块的大小和增加通信数据的空间局部性。另一方面，由于是体系结构的相互作用所引起的，伪共享扑空是第 3 章所讨论的附加通信的例子。同真共享和冷启动扑空相比，伪共享扑空能够通过减小块的大小来减少，还可以通过软件（调整）和硬件方面的一些其他优化措施来减少，对此，在后面将会有讨论。这样，决定最好的缓存块的大小就具有根本性的意义了，而这件事只能通过针对真实程序的评测才能解决。

1．缓存扑空的一种分类

图 5-20 中的流程图给出了一个具体的算法来对于缓存一致性多处理器中的缓存扑空进行分类[⊖]。理解其中的细节现在还不是很重要，对于本章后面的内容来说，只要理解前面的定义就够了，但对细节的理解能够加强我们的观念，并且是一个有用的练习。在这个算法中，一个存储块在缓存中的生存时间定义为它在缓存中处于有效状态的时间区间，也就是说从引起它装载进缓存的扑空开始到它被作废，替换或者程序结束。当一次扑空出现的时候，还说不清楚它应该归于哪一类；只是当相应的存储块被替换或者作废时，我们才能明确，因为只有在那个时刻我们才知道在该存储块生存期是否发生了真共享或者仅仅是伪共享。让我们首先考虑简单的情形。情形 1 和 2 是直接的冷启动扑空，发生在事先没被写过的块上。情形 7 和 8 反映了在一个块上的伪共享和真共享，被共享的那个块在缓存中先被作废，然后被另一块替换。共享类型的确定是通过考察在生存期作废后是否有特定的字被修改了。情形 9 是简单的容量性（或者冲突）扑空，这是由于存储块先被替换并且自从上次访问后其中的字还没有被修改。所有其他的情形反映的是一些组合因素的扑空。例如，由于这个处理器以前

⊖ 在这种分类中，我们不区别容量扑空和冲突扑空，主要考虑到它们都是可用资源（缓存组或整个缓存）用尽后表现出来的现象，而强调它们之间的差别并不给多处理器问题的研究增加什么有意义的结论。

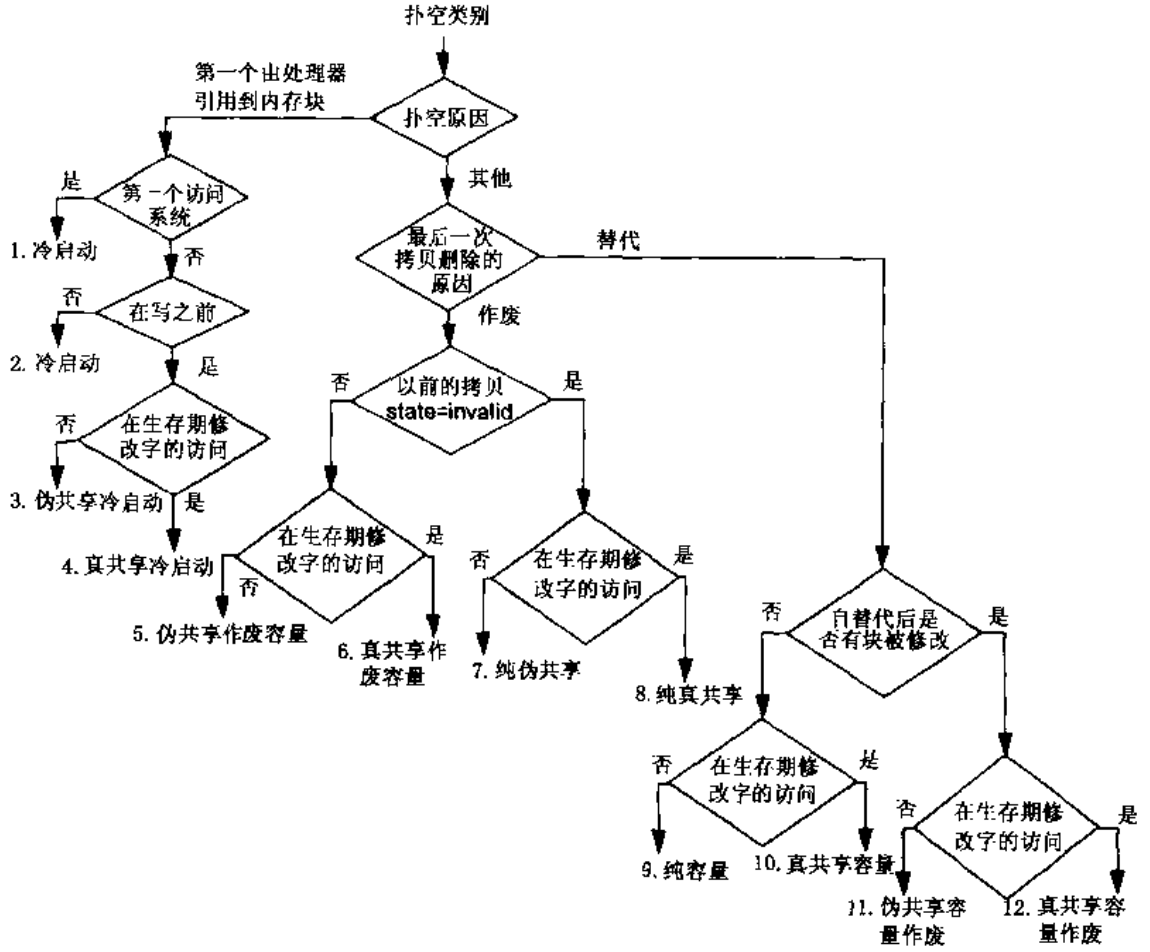


图 5-20　共享存储多处理器高速缓存扑空的一种分类。按照这种分类法，四种基本的缓存扑空是冷启动扑空、容量扑空、真共享扑空和伪共享扑空（冲突扑空在这里被看成是容量扑空）。由于一次扑空可能有多种原因，人们还提出了一些混合类型。例如，一个存储块可能先在处理器 *A* 的缓存中被替换，然后由处理器 *B* 写，然后又由 *A* 读回去，于是造成了一次容量-cum-作废伪/真共享扑空。这在分类中可能被称为“伪/真共享容量-作废”，由于共享优先级高，还由于替换在作废之前发生（图中的情形 11 和 12）。如果存储块先在 *A* 的缓存中作废，然后这个无效块被替换，然后再被 *A* 读出，它就要被称为“伪/真共享作废-容量”（情形 6 和 7）。按照我们给出的四种类型来说，上面这些都属于真或伪共享扑空。注意：图中的问题“已修改的字在生命周期被访问过?”问的是自从上一次“根本的一致性”扑空后已修改的存储块是否在当前生命周期里被访问过了，其中“根本的一致性”对应于第 4、6、8、10 和 12 类。这只有在当前生命期结束时才能决定

从来没有访问过这个块，情形 4 和 5 是冷启动扑空；然而，一些其他的处理器写过这个块，因此也有共享（假或真）。类似地，我们可能在那些曾经由于容量或者冲突被替换的存储块上发生假的或真共享。例如，如果一个扑空的发生是由于伪共享和容量问题，那么通过减小块的大小可能消除伪共享，但可能并不能消除扑空。另一方面，共享扑空在某种意义上要比容量扑空更基本，这是因为共享扑空的存在是与缓存容量无关的，因此我们在多原因扑空中给予它们优先级。在我们的分类结果中，所有名称带有真共享的扑空被称为根本性一致性扑空。即使缓存有无限大、单字块、所有数据预装在缓存中了（即没有了冷启动扑空）它们还是会出现的。例 5.11 解释了扑空分类中的这些定义。

316
~
317

例 5.11　假设三个处理器 P_1、P_2、P_3，发出存储器操作，如表 5-4 中的头几列所示（第

一列指出虚拟时间或步子)。用扑空分类算法对最后一列的扑空进行分类。假定每个处理器的缓存只有一个4字的缓存块，所有缓存最初都是空的。

解答： 结果如表5-4所示。■

表5-4　对源于三个处理器的访问流产生的扑空分类

次数	P_1	P_2	P_3	扑空类别
1	ld w0		ld w2	P_1 和 P_3 扑空；类别将在后面说明
2			st w2	$P_{1,1}$：纯冷扑空，$P_{3,2}$：提升
3		ld w1		P_2 扑空；类别将在后面说明
4		ld w2	ld w7	P_2 命中；P_3 扑空；$P_{3,4}$：冷扑空
5	ld w5			P_1 扑空
6		ld w6		P_2 扑空；$P_{2,6}$：冷真共享扑空（w2被访问）
7		st w6		$P_{1,5}$：冷扑空；$P_{2,7}$：提升；$P_{3,4}$：纯冷扑空
8	ld w5			P_1 扑空
9	ld w6		ld w2	P_1 命中；P_3 扑空
10	ld w2	ld w1		P_1，P_2 扑空；$P_{1,8}$：纯真共享扑空；$P_{2,6}$：冷扑空
11	st w5			P_1 扑空；$P_{1,10}$：纯真共享扑空
12			st w2	$P_{2,10}$：容量型扑空；$P_{3,11}$：提升
13			ld w7	P_3 扑空；$P_{3,9}$：容量型扑空
14			ld w2	P_3 扑空；$P_{3,13}$：作废容量型假共享扑空
15	ld w0			P_1 扑空；$P_{1,11}$：容量型扑空

注：如果在同一行列有多次引用，我们假定 P_1 在 P_2 之前，P_2 在 P_3 之前　记号 ld/st wi 表示对字 i 的装入/存放，w1到w4在相同的缓存块，等等。记号 $P_{i,j}$ 指向由处理器 i 在第 j 行发出的存储访问

2．缓存块的大小对于扑空率的影响

将图5-20的算法应用于一种负载的模拟运行结果，我们可以确定各种扑空发生在程序中的频率，以及这些频率如何随着缓存组织的变化而改变，例如块大小的变化。图5-21表示各种应用运行在16个处理器上扑空的分解，这些处理器都有一个1 MB的4路组相联缓存，缓存块的大小在8～256字节之间变化。直方图表现4种基本的扑空：冷启动扑空（情形1和2），容量（包括冲突）扑空（情形9），真共享扑空（情形4，6，8，10，12），以及伪共享扑空（情形3，5，7和11）。除此以外，它们还表现了升级的频率——也就是那些发现块在缓存中处于共享状态的写操作。更新不同于其他类型的扑空，由于缓存已经有了有效的数据，只是需要独享的所有权。虽然没有包含在图5-20的分类方案中，它们仍然被看作是扑空，这是由于它们在处理器互连机构上产生流量，从而可能阻滞处理器的执行。

对于每个独立的应用，扑空特征随块大小变化，其情况和我们对程序和扑空类别的理解一致。冷启动、容量和真共享扑空倾向于随块的变大而减少，这是因为随着扑空所带进来的附加数据在该块被替换前被访问。在所有情形下，真共享在扑空中占有明显比例，于是即便是理想的、无限大的缓存，扑空率和总线带宽将都不会为零。然而，整个特征对于不同的程序大相径庭。例如，真共享部分的大小变化范围可能是很大的。某些应用在伪共享中表现出随块大小实质性的增加，而另一些则几乎没有。进一步看，这个图只是表现了针对缺省数据集的情况。在实际中，得到关于一个应用程序的伪共享或空间局部性之前，考察结果随输入数据规模和处理器数的扩展变化是重要的（见第4章）。下面让我们来研究应用的性质，它们反映了在机器层次看到的扑空特征方面的差别，这使我们能够定性地理解扩展的效果。

318

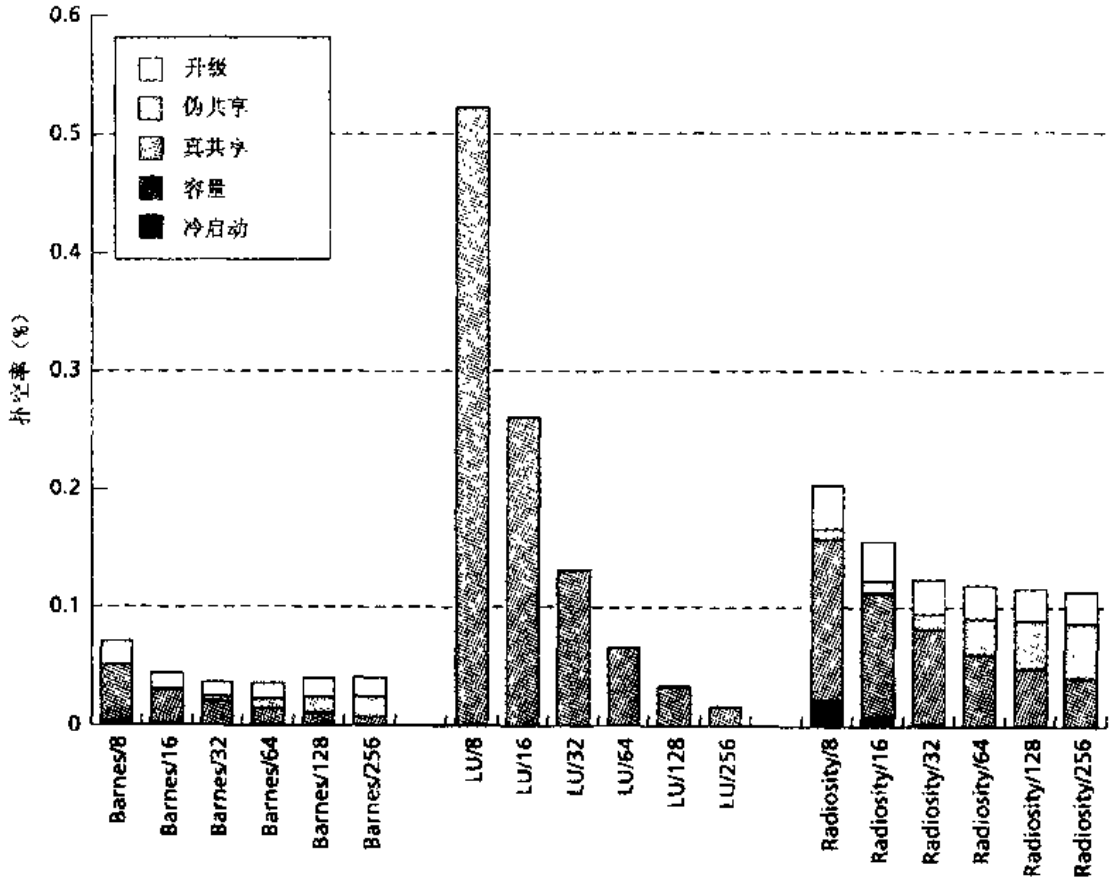


图 5-21a　在 1 MB 缓存的处理器上，Barnes-Hut、LU、Radiosity 应用在不同缓存块大小下的扑空情况。冲突扑空含在容量扑空中。应用不同，扑空的情形变化很大，但我们还是能够看到一些共同的特点。冷启动扑空和容量扑空随缓存块大小的增加很快下降，这是由于空间局部性得到了利用。真共享扑空也随之减少，但伪共享扑空增加。对小缓存块来说，伪共享扑空的成分通常较小，但有时会很快增加。更新所导致的情形是直方图顶部的非阴影部分，可以忽略不计

3．同应用程序结构的关系

含有多个字的缓存块通过预取所访问地址附近的数据，使程序的空间局部性得到利用。当然，超出某个点，较大的缓存块反而会对性能有不好的影响，原因在于：1）预取了不需要的数据；2）引起冲突扑空增加，这是由于对固定的缓存规模来说，这样就减少了缓存块的个数；3）引起伪共享扑空增加。并行程序中的空间局部性一般低于串行程序，这是因为当一个存储块带到缓存时，其中有些数据可能属于另一个处理器，并且将不会被完成扑空的处理器用到。作为一个极端的例子，某些并行程序将数组中相邻元素赋给不同的处理器，虽然可能取得好的负载平衡，但在进程中就大大地降低了程序的空间局部性。

图 5-21 中的数据表示，即使在并行情况下，LU 和 Ocean 有好的空间局部性并且没有伪共享。扑空率的许多成分随缓存块大小的增加按比例下降，伪共享扑空基本上是不存在的。这在很大程度上是由于这些基于数组的代码用到了和体系结构相匹配的数据结构，如同第 3、4 章所讨论的那样。例如，Ocean 中的一个网格不是由一个二维数组（二维数组在面向列划分的边界引起严重的伪共享），而是由一个四维数组来表示的：一个二维块的数组，每一个块自己也是一个二维数组。这样通过程序或者编译的结构化，保证了多数访问是单位跨距的，覆盖大量连接数据，于是表现出很好的结果。

319
320

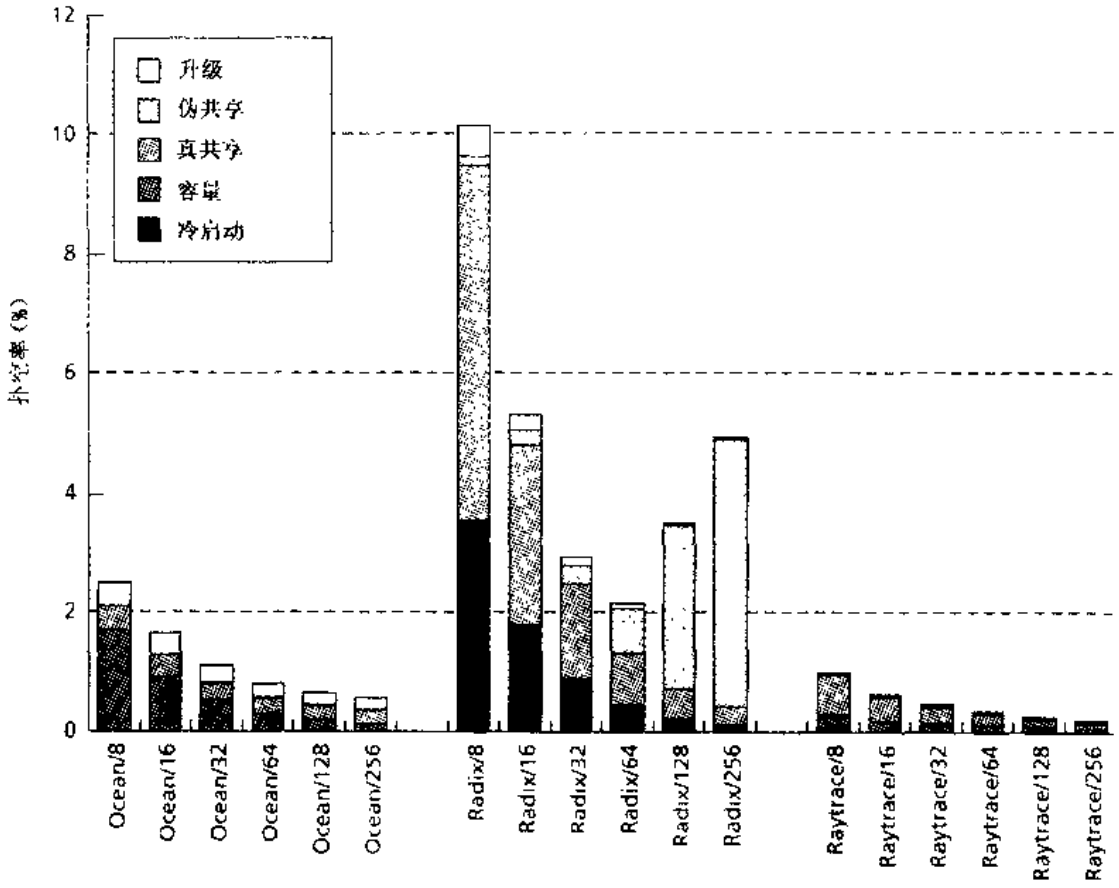


图 5-21b　在 1 MB 缓存的处理器上，Ocean、Radix、Raytrace 应用在不同缓存块大小下的扑空情况

在 Ocean 中，容量性扑空是值得注意的，但它们只是反映在一个进程划分的内部元素上，因此它们有非常好的空间局部性。和 LU 的一个区别是，Ocean 中的真共享扑空展示不出这样好的空间局部性。大多数真共享扑空反映在相邻划分的边界元素上。在面向行的边界安排下，由于被访问的数据在地址空间中是连续的，于是能表现出好的空间局部性。然而，当一个处理器访问面向列边界的一个元素时，要涉及到它的相邻划分的内部元素的一整个缓
321 存块，其中大部分数据将不被使用，因此就浪费了。由于容量性扑空对于这个问题和这种机器配置来说不是很大的，整个空间局部性的限制在于真通信的要求。在 LU 中，即使真通信也是每次 $B \times B$ 连续块，因此空间局部性即使在真共享扑空情况下也是很好的。

至于扩展性，当问题的规模和处理器个数增加时，这两个应用的空间局部性都保持得相当好，没有伪共享（至少在划分变得过分小之前）。即使对于大于 256 字节的缓存块，至少对于 LU 应该是这样的。在 Ocean 中，容量相对于真通信扑空（因此也就是空间局部性）的变化在很大程度上取决于数据规模和处理器数的相对变化关系。

图形应用程序 Raytrace 所表现出来的伪共享基本可以忽略，但它在空间局部性方面表现得明显的不好。伪共享小是因为主要的数据结构（构成场景的多边形的集合）是只读的。读写共享只是发生在图像平面数据结构和任务队列上，但那是在控制范围内的并且对大多数问题来说相比很少。这种真共享扑空率可通过增加缓存块的大小来减少。至于较差的容量扑空的空间局部性问题（尽管在这种配置中总的量是小的），其原因是对于多边形集合的访问模式是相当任意的，主要由于一束光线在其中反射的对象集合不可预测。在扩展性方面，随着

问题规模的扩大（大多数情形下意味着更多的多边形），基本的效果可能是更大的容量性扑空率；在单个部分内的空间局部性应该没什么变化。除能在图像平面和任务队列中的数据结构中看到更多的共享外，从许多方面来看，处理器数增大的效果类似于问题规模的变小。

Barnes-Hut 和 Radiosity 应用表现出中等的空间局部性和伪共享。这些应用用到了复杂的数据结构，包括树型编码的空间信息和数据，其中分配给每个处理器的记录在存储器中不是连续的。例如，Barnes-Hut 的操作作用在存放在数组中的粒子记录上，随着应用程序的执行，粒子在物理空间中运动，粒子记录可能要重新分配给不同的处理器，数组中相邻的粒子在一段时间后很可能属于不同的处理器。空间局部性在一个粒子记录内体现得很好，但在跨记录之间体现得就不好。在缓存块较大的情况下，伪共享成为一个问题，其原因是多方面的。首先，不同的处理器可能对共享同一个缓存块的不同记录进行写操作。其次，一个粒子的数据结构（记录）包含有某些数据域，要被在本阶段拥有它的处理器修改（例如，在计算阶段作用在粒子上的力），还包含有数据域要被其他处理器读，但在本阶段不被修改（例如，当前粒子的位置）。由于这两个域可能落入同一个缓存块，就要导致伪共享。通过按照数据域的访问模式拆分粒子数据结构，有可能消除这样的伪共享；但由于扑空率总体上很小，人们并没有这样做。随着问题的规模和处理器个数的变化，Barnes-Hut 的扑空率情况也不会有太大变化。这是由于工作集改变得很慢（和粒子数的对数成比例，这一点和 Ocean、Raytrace 不 322
同），空间局部性是由一个粒子记录的大小确定的，因此保持不变。同时，伪共享的来源对于处理器数来说并不敏感。Radiosity 要复杂得多，对于较大的数据集合和较多的处理器，它的行为难以推测；惟一的办法是收集表现趋势的试验数据。

Radix 显示的共享情形最差，即使对于 1 MB 的缓存来说，它不仅有很高的扑空率（由于冷启动和真共享扑空），而且由于在 128 字节或更大缓存块情形的伪共享扑空变得更差。Radix 中的这种伪共享效果如第 4 章所示。现在来考察它的具体情况。考虑要对 256 K 个键排序，使用 1 024 为基和 16 个处理器。平均来说，这导致每个基每个处理器有 16 个键（64 字节数据），然后它们写到一个全局数组的一个连续的部分，起始点不可预测。在这个数组中相邻的 64 字节区域被不同的处理器写。如果缓存块大于 64 字节，伪共享的可能性显然就很大。随着问题规模的增加，我们将清楚地看到伪共享变小。增加处理器数的效果恰好是相反的。Radix 突出地表明，仅仅通过看问题的规模和处理器数来得到关于伪共享和空间局部性的结论是不够的。重要的是要理解这些结果是如何依赖于实验中所选择的关键参数，并且这些参数在实际中可能如何变化。

Multiprog 工作负载在 1 MB 缓存上运行的数据如图 5-22 所示。其中分别表示了用户代码、用户数据、内核代码和内核数据。对于代码来说，只有冷启动和容量性扑空。进而看到操作系统中数据引用的空间局部性并不怎么好。在某种程度上，这对于应用程序的数据也如此， 323
这是因为 gcc（在 Multiprog 中引起扑空的主要应用）用到了大量的链表，而这种数据结构的空间局部性不好。有意思的是，尽管我们只是运行串行程序，我们仍然看到了真共享扑空。这是由于进程的迁移所导致的（操作系统为资源管理所作的工作），进程从一个处理器迁移到另一个处理器，引用它曾经在另一个处理器写入的存储块。冷启动和容量扑空中的空间局部性是合理的，对于内核数据来说真共享扑空一点也不减少。这种情形的一个原因可能是操作系统本身还不是一个好的并行程序。

最后，让我们考察 Ocean、Radix 和 Raytrace 在 64 KB 缓存上的行为。所导致的扑空率如

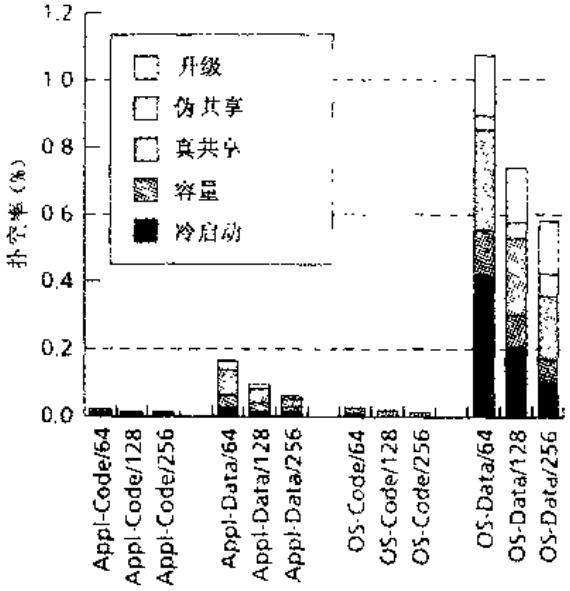


图 5-22　在 1 MB 缓存的处理器上，Multiprog 应用在不同缓存块大小下的扑空情况。这里的结果基于 1 MB 缓存，对真共享扑空来说，空间局部性对这个应用要比对操作系统好得多

图 5-23 所示，正如所预想的，整体扑空率较高并且容量性扑空增加较明显。缓存块大小对于真共享和伪共享扑空的效果和 1 MB 缓存相比没有实质性的不同，这是由于这些性质是直接针对程序的并行分配和性能调试策略的，和缓存的大小关系不大。不过，容量性扑空的行

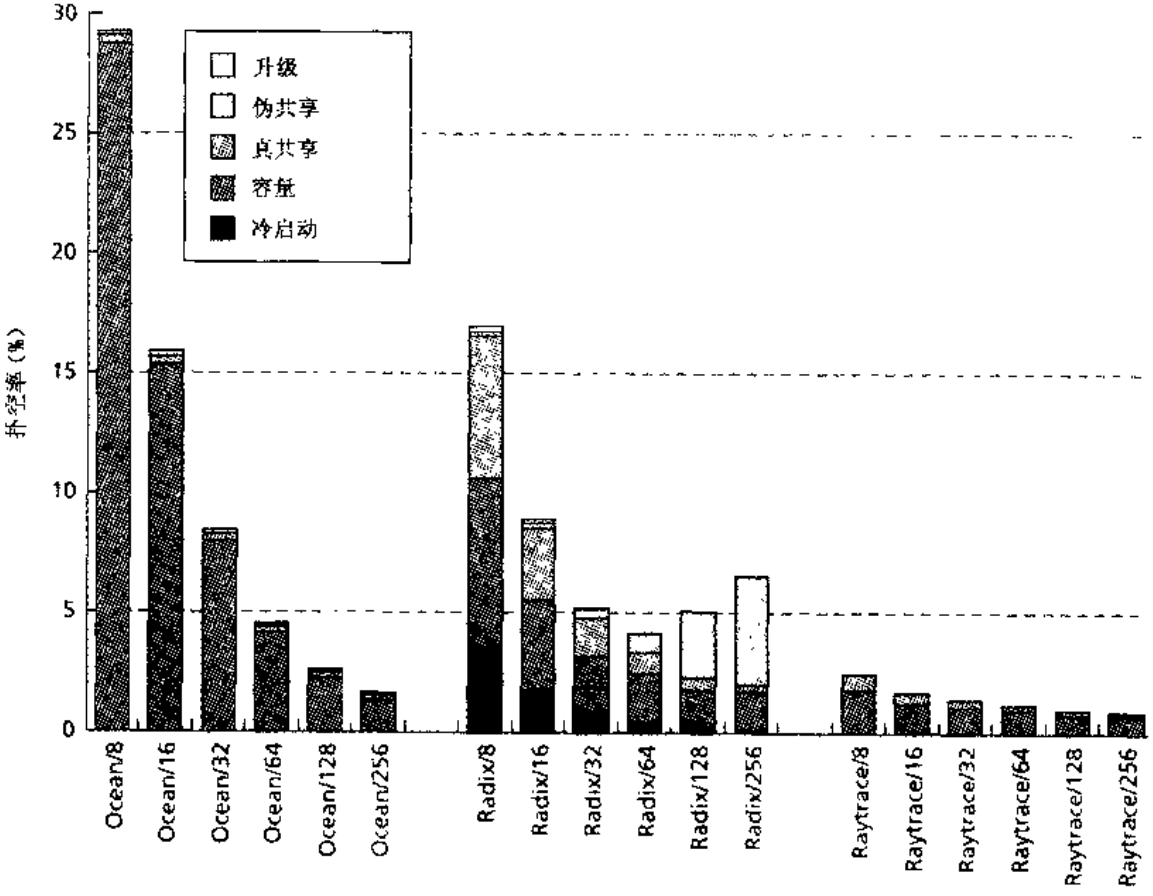


图 5-23　在 64 KB 缓存上，应用程序在不同缓存块大小下的扑空情况。容量扑空现在在整个扑空率中占有较大的比例。随缓存块大小的增加，不同的应用表现出的容量性扑空率下降的情况也不同

为对于整个扑空率的影响要大得多。例如，在 Ocean 中，容量性扑空对于共享扑空具有支配地位；由于它们有好得多的空间局部性，和 1 MB 缓存相比整个扑空率随块大小的增加下降很快。（在小缓存中用很大的块可能有问题，存储块可能由于冲突在处理器还没有机会访问其中所有的字时就被替换出去了。）在 Raytrace 中，容量扑空和真共享扑空相比有较差的空间局部性，因此对于较小缓存用较大块的总体效果看来要差些。文献中有对于其他应用程序的伪共享和空间局部性的研究结果（Torrellas, Lam, and Hennessy 1994; Jeremiassen and Eggers 1991; Woo et al. 1995）。

对于多数应用来说，较大的缓存块会减少扑空率，但在我们考虑的大小范围内，大缓存块有两个重要的缺点。首先，它们会增加每次扑空的开销，这是由于有更多的数据要在总线上传送（尽管有些技术能缓解这种情形，例如关键字重启技术，只要所需的存储字到达就允许处理器继续向下执行）。其次，如果不是整个块都有用的话，它们增加流量，从而也增加了竞争。

4. *缓存块大小对于总线流量的影响*

让我们简略地考察缓存块大小对于总线流量（而不是扑空率）的影响。尽管扑空数和所产生的总流量显然是相关的，它们对于所观察到的性能的影响可能是相当不同的。尽管现代处理器常通过用其他活动来覆盖，努力隐藏扑空所带来的延迟，扑空所带来的代价还是可能直接影响到性能。另一方面，流量影响性能是间接的，主要是由于所产生的竞争增加了其他扑空的代价。例如，一个应用程序的扑空率可能通过增加缓存块的大小得到了明显的减少，但总线的流量增加了 50%；如果该应用程序最初只用到总线能力和存储带宽的 10%，这可能就是一个合理的折中。增加总线和存储的利用率到 15% 不可能明显增加扑空延迟。然而，如果应用程序最初要用 75% 的总线和存储带宽，那么增大块的大小可能就是一个不好的主意。

图 5-24 表示我们应用程序随缓存块大小变化的总的总线流量，单位是每条指令的字节数或者每 FLOP 的字节数。从这个图中可以看到三个要点。首先，流量的行为和扑空率表现得相当不同。只有 LU 表现出随块大小单调递减的总流量。多数其他应用随块大小的变化看到的是两倍或者三倍的流量。其次，除 Radix 外，其他应用的总流量需求仍然是小的，即使块大小为 256 字节。Radix 对于带宽较大的需求（对于 128 字节的缓存块，假设一个持续 200 MIPS 的处理器，大约每个处理器 650 MBps）反映了它在大缓存块上的伪共享问题。第三，对于每一次总线事务或者扑空，地址和命令流量的固定开销是小缓存块情形总流量的一个重要组成部分。因此，尽管实际应用数据流量通常随块大小的增加而增加，由于空间局部性比较差，总流量经常在 16 ~ 32 字节，而不是 8 字节时得到极小化，这是由于改善的扑空率分摊了开销。

324 ~ 325

图 5-25 表示 Multiprog 的流量数据。缓存块从 64 ~ 128 字节变化所引起的流量增加是小的，跳到 256 字节时就很显著了（主要是由于内核数据的引用）。最后，图 5-26 表示 64 KB 缓存对于这三个相关应用的流量结果。对 Ocean 来说，即使 64 字节和 128 字节的缓存块也不是那么差，这是由于好的空间局部性使得起支配作用的容量扑空较小。

5. *缓解大缓存块的缺点*

现在的处理器倾向于用较大的缓存块。这个趋势是由处理器性能和存储器访问时间之间的差距变大带来的。较大的缓存块，在较大量数据时能够分摊总线事务和存储访问的代价。

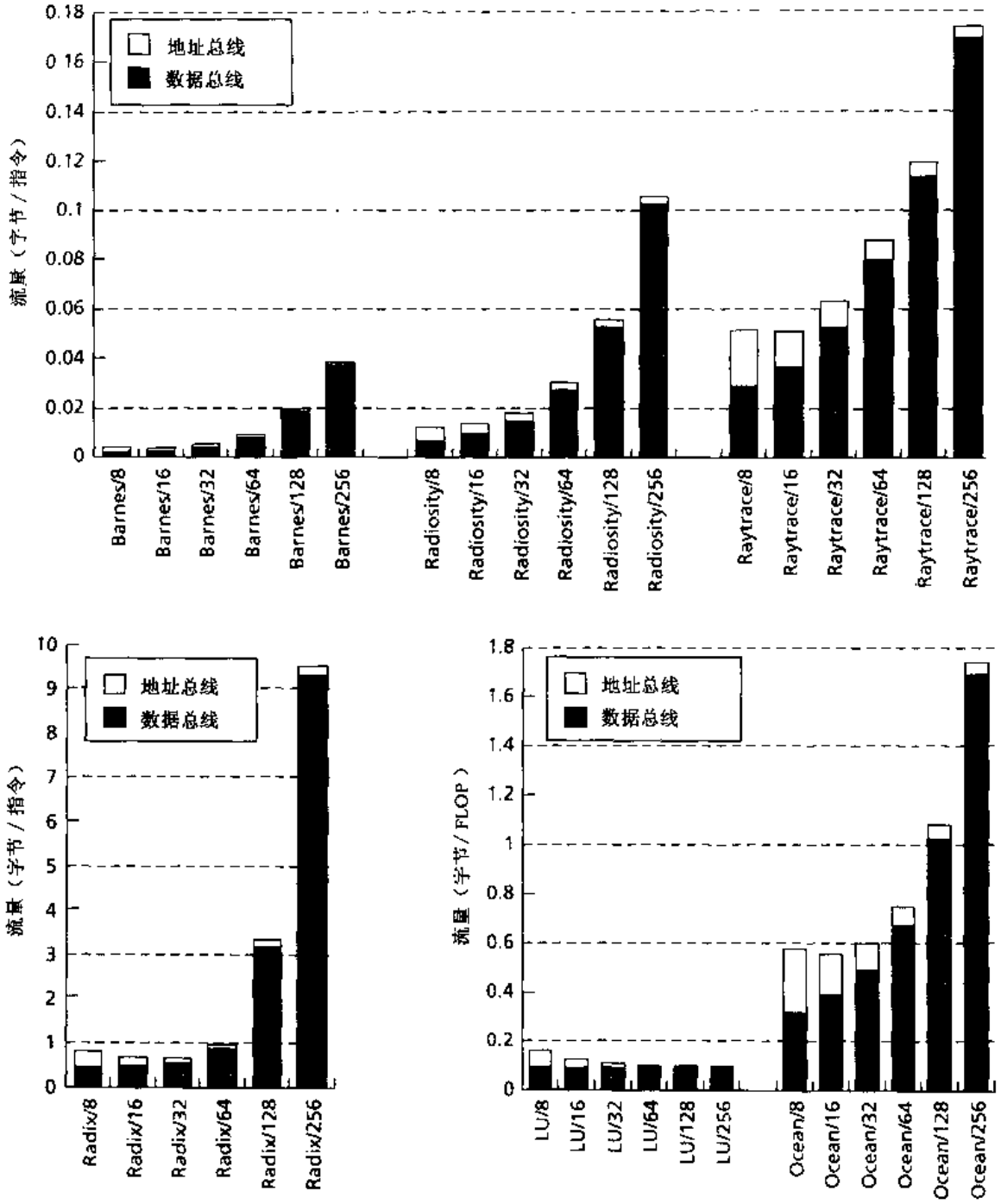


图 5-24 每处理器 1 MB 缓存情形，流量（每指令字节数或每 FLOP 字节数）随缓存块大小变化的情况。当通信扑空起支配作用时，数据流量随缓存块变化增加得相当快；LU 是个例外，它在所有类型的扑空上空间局部性都很好。地址（包括命令）总线流量倾向于随缓存块大小递减，这是由于扑空率和传送的缓存块数都减少了

处理器和存储器芯片的密度的增加使得用很大的一级和二级缓存成为可能，较大缓存块所预取的数据带来的好处可能掩盖由此带来的冲突扑空增加的问题。然而，这个趋势对多处理器设计来说可能预示着不妙，因为伪共享变成了一个较大的问题。幸运的是，我们有一些硬件和软件的机制能用来对付大缓存块的不良影响。

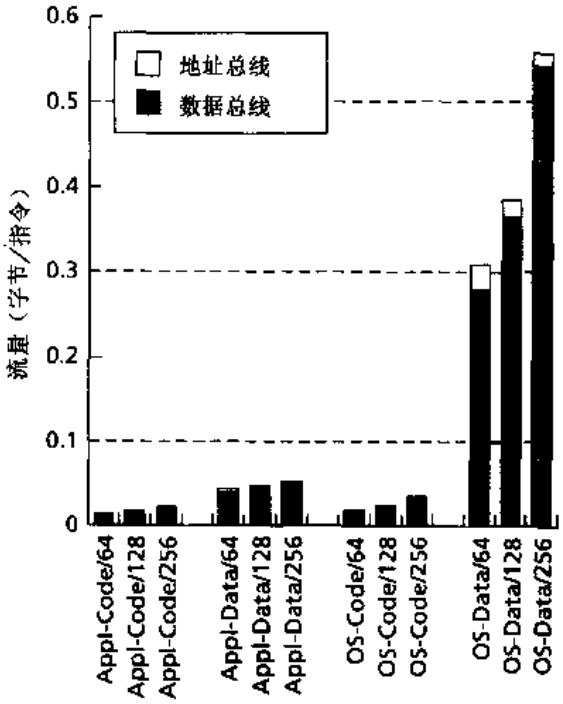


图 5-25　1 MB 缓存情形，Multiprog 应用中流量（每指令字节数）随缓存块大小变化的情况。从操作系统内核来的数据流量随缓存块变化增加得相当快

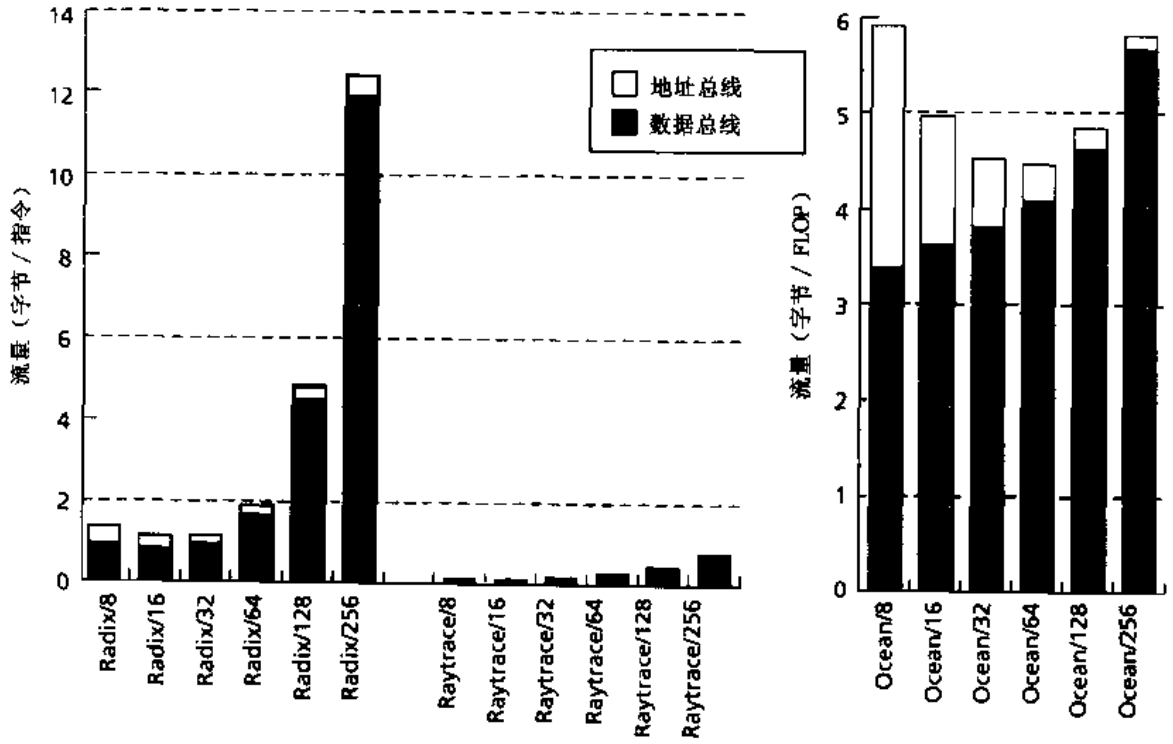


图 5-26　每个处理器 64 KB 缓存情形，流量（每指令字节数或每 FLOP 字节数）随缓存块大小变化的情况。对 Ocean 来说，流量的增加要比 1 MB 缓存情形慢，这是由于现在起支配作用的容量扑空表现了很好的空间局部性（进程在子网格上的遍历）。然而，一旦引起伪共享的缓存块阈值被超过，Radix 的流量就增加得很快

在本章的后面，我们对减少伪共享，针对一致性扑空改进局部性的软件技术有进一步的详细讨论。它们本质上涉及数据结构的组织或者任务的分配，使得由不同进程访问的数据不在共享地址空间细粒度交织存放。一个例子是用高维数组，使得存储块或划分整个是连续

326 ~ 328 的。人们也开发了若干编译技术来自动地进行数据布局，从而减少伪共享（Jeremiassen and Eggers 1991）

由于伪共享是由一致性的粒度过大引起的，减少它但仍然利用空间局部性的方法是在数据传送时用大的存储块，但在一致性管理中用较小的单位。一种自然的硬件机制是用存储子块。每个缓存块有一个地址标记，同时对每一个子块还有不同的状态位。一个子块可能是有效的，而其他的子块同时可能是无效的或者脏的。这种技术用在许多单处理器系统中以在替换时减少写回存储器的数据量，或者在读扑空时减少存储器访问时间（一旦有关子块到达就重启处理器，称为关键字重启）。为了避免伪共享，一个处理器所做的写操作可能会作废另一个处理器的子块，但保持其他的子块有效。从另一个角度来实现这种思路，也可以用较小的缓存，但扑空发生时系统可以多预取一些存储块。还有人提出过可调整块大小缓存的建议（Dubnicki and LeBlanc 1992）。这些做法的缺点是增加状态和复杂性，不太适用商用缓存的设计。

一种更精细的硬件技术是延迟从一个处理器发出的作废操作的传播或者应用，等着有多个写操作后一起来作废。延迟作废操作，将一组操作集中起来一次完成，以减少对那些块的干涉读扑空的出现。然而，这种技术可能以一种微妙的方式改变存储器同一性模型。在第9章，当我们考虑可扩展机器的弱同一性模型的时候，会有进一步的讨论。另一个减少伪共享的硬件技术是用基于更新的协议。

5.4.5 基于更新和基于作废协议的对比

对一个缓存中某个块的写操作应该引起其他缓存中的拷贝更新还是作废？这一直是争论不休的话题。不同的厂家有不同的态度并且实际上对不同的设计用了不同的做法。这种矛盾的起因是由于基于更新和基于作废协议的相对性能在很大程度上取决于应用程序负载所表现出来的共享的模式，以及各种背后操作的代价。从直觉上看，如果处理器在更新之前使用数据，并可能希望要在将来看到新的值，更新就会比作废表现出更好的性能。然而，如果持有老数据的处理器再也不会用它了，更新就是不必要的，所引起的流量只是消耗了互连和控制器资源。作废则会清除旧的拷贝，消除了明显的共享[○]。这种更新协议的“收集鼠”现象在多道程序下特别令人难受，串行进程在操作系统的控制下在处理器之间迁移，于是无用的更新在不再运行该进程的处理器的缓存中进行。如例5.12所示，我们容易构造出一种方案明显优于另一种方案的情形。 329

例5.12 考虑下述两个程序访问模式：

- 模式1：重复 k 次；处理器1向变量 V 写入一个新的值，处理器2到处理器从 P 都读 V 的值。这表现了一种单生产者多消费者机制可能出现的情形，例如，在一到多事件同步中处理器访问一个高度争用的标记变量。
- 模式2：重复 k 次；处理器1向变量 V 写 M 次，然后处理器2读 V 的值。这表示了一种在处理器对之间可能出现的一种共享模式，其中第一个处理器连续计算并累加一个变量的值，当累加完成后，另一个处理器读这个值。

用缓存扑空数和总线流量来评价，基于更新和基于作废协议的相对代价是什么？假设一

○ 这种共享是不必要的，——译者注

个作废/升级过程要消耗 6 个字节（5 个字节地址，1 个字节命令），更新需 14 个字节（6 个字节地址和命令，8 个字节为更新的数据），且一个常规的缓存扑空要 70 个字节（6 个字节地址和命令，加上 64 个字节的数据，对应一个缓存块）。还假设 $P = 16$、$M = 10$、$k = 10$，所有缓存最初都是空的。

解答：对于模式 1 用更新方案，在所有 P 个处理器上的第一次迭代将引起一个常规的缓存扑空（包括处理器 1 在写操作的时候）加上由于写的一次更新。在后续的 $k-1$ 次迭代中不会产生更多的扑空，并且每次迭代只有一次更新产生。这样总体上将看到扑空 = P = 16；流量 = $P \times$ RdMiss + $(k-1) \times$ Update = $16 \times 70 + 10 \times 14 = 1\,260$ 字节。

对于作废方案，所有 P 个处理器在第一次迭代将引起一个常规缓存扑空。在后续 $k-1$ 次迭代中，处理器 1 将产生一次升级，但所有其他的处理器将经历一次读扑空。这样，将升级算作扑空，整个我们将看到扑空 = $p + (k-1) \times p = 16 + 9 \times 16 = 160$，其中 151 个是读扑空，9 个是升级；流量 = 读扑空 × RdMiss + $(k-1) \times$ Upgrade = $151 \times 70 + 9 \times 6 = 10\,624$ 字节。

对于模式 2 用更新方案，首次迭代将发生两次常规缓存扑空，处理器 1 一次，处理器 2 一次。在后续 $k-1$ 次迭代中，将没有更多的扑空产生，但在每次迭代会产生 M 次更新。这样，总体上我们将看到扑空 = 2；流量 = $2 \times$ RdMiss + M × $(k-1) \times$ Update = $2 \times 70 + 10 \times 9 \times 14 = 1\,400$ 字节。

对于作废方案，首次迭代将发生两次常规缓存扑空。在后续 $k-1$ 次迭代中，每次迭代会产生一次升级（对于第一个写）加上一次常规读扑空。这样，将升级算作扑空，总体上我们将看到扑空 = $2 + (k-1) \times 2 = 2 + 9 = 11$；流量 = 扑空 × RdMiss + $(k-1) \times$ Upgrade = $11 \times 70 + 9 \times 6 = 824$ 字节。■

这些例子说明，有可能设计出方案来，使更新和作废协议双方的优点都体现出来。这样方案的成功将取决于它们的代价和对于真实并行程序和负载的共享模式。我们下面先简略地探讨一下设计选项，然后用负载驱动方法来进行评估。

1. 基于更新和基于作废协议的结合

一种能够体现两种类型协议优点的方式是在硬件上对两者都支持，在页面粒度动态决定对给定的一个页一致性是通过更新还是通过作废来维护。关于协议选择的决定可以通过系统 [330] 调用来指出。这种方案的主要优点是它们相对比较容易支持；它们利用 TLB 来向一致性子系统的其他部分指明用哪一种协议。这种方案的主要缺点是它们给程序员增加了为页或者数据结构选择协议的负担。由于控制的粒度较粗，也使得这种决定不容易作出，因为适应不同协议的数据结构可能落在相同的页面上。

一种替代的方法是通过在运行时观察共享的行为，在缓存块的粒度来选择协议。理想情况下，对每一个写，希望能够看到未来所有处理器要对当前缓存块的引用，然后再决定是否作废其他的拷贝或者更新。由于这个信息显然是得不到的，还由于有缓存替换和伪共享带来的严重波动，需要一种更实际的方案。

所谓竞争性方案，是基于运行时观察到的模式，从硬件上在作废和更新之间改变对于一个存储块的协议。这种方案的关键属性是，如果对一个缓存块作了错误的决定，由于那个错误决定所带来的损失应该保持有界并且很小（Karlin et al 1986）。例如，如果一个块当前用更新模式，一旦一个处理器连续向它写，但没有其他处理器从中读，它就不应该保留在那个模式中。

有一类方案，旨在限定更新协议的损失，原理如下（Grahn、Stenstrom and Dubois 1995）。从 5.3.3 节介绍的基本 Dragon 更新协议开始，让每一缓存块和一个向下计数器联系起来。只要一个缓存块被本地处理器访问，那一块的计数值就被置一个阈值 k。每当一个块收到一次更新，计数器就递减。如果计数器到了 0，这个块就在本地被作废。本地作废的后果是下一次一个更新在总线上产生时，它可能找不到持有一个有效拷贝的缓存；在这种情形，这个块将切换到已修改状态（如 Dragon 协议那样）并且将停止产生更新。如果现在某个其他处理器访问这一块，它就要再次被切换到共享状态，这个混合协议会又开始产生更新。

一个在 Sun SparcCenter 2000 中实现的相关做法是以某种概率有选择地作废，而不是更新，这个概率是在配置机器时建立的（Catanzaro 1997）。还可以有一些其他混合的做法。例如，在一级缓存上用一种基于作废的协议，在二级缓存上缺省地用基于更新的协议。然而，如果对于一级缓存中的一个有效块二级缓存收到了第二次更新，那么这个块也要在二级缓存
331 中作废。这样当一个块在所有二级缓存中都作废时，对该块的写就不再引起更新。

2．利用工作负载驱动的评估

为了评价我们前面介绍的作废，更新和混合协议的折中，图 5-27 按类表示四种应用的扑空率，用 1 MB 四路组相联缓存和 64 字节的块。所用的混合协议是刚才描述的基于阈值的方案。我们看到，对于明显容量性扑空的应用，扑空有时在使用更新协议时增加。这是合乎情理的，因为这个协议（用 LRU 替换算法）将数据保持在处理器缓存中，那些数据要被作废协议去掉。对于那些具有明显真共享或者伪共享扑空率的应用程序，这些类别随着更新协议减少：在一个写更新后，持有该块的其他的缓存能够访问它们而不会扑空。从总体上看，对于这三种情形来说更新协议看起来是优越的，而混合协议的优越性居中。然而，那种没有显示在这个图中的类型是对于这些协议的升级或者更新操作。这个数据表示在图 5-28 中。注意，

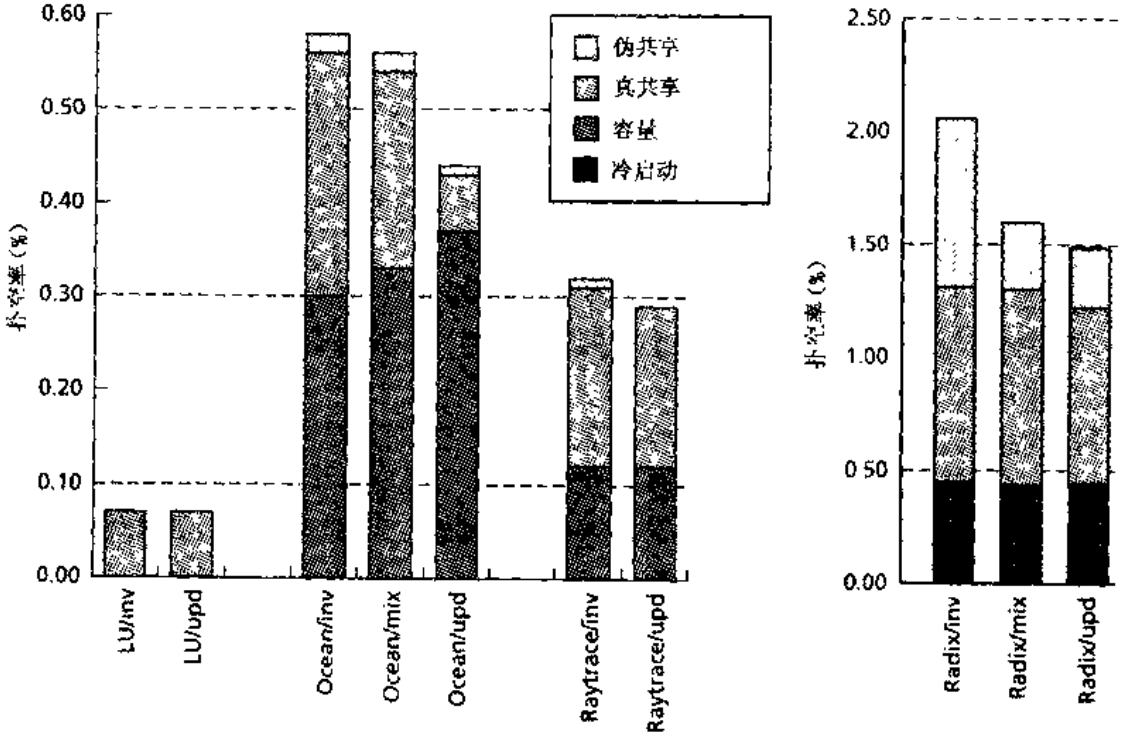


图 5-27　扑空率及其在作废、更新和混合协议上的分解。这里的数据假设 1 MB缓存、64 字节缓存块、4 路组相联、对混合协议阈值 $k=4$

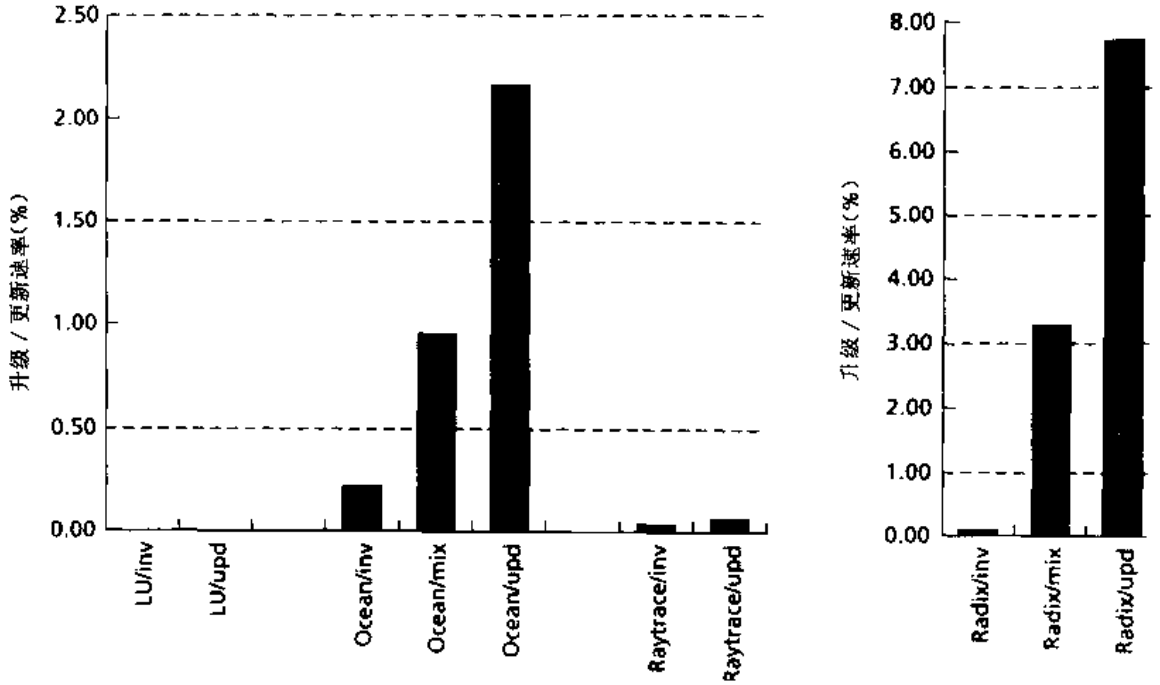


图 5-28　作废、更新协议和混合协议上的升级和更新率。这里的数据假设 1 MB 缓存，64 字节缓存块、4 路组相联、对混合协议阈值 $k = 4$。数据率是相对于总的存储访问数

由于更新操作大概 4 倍于扑空，这些图的比例已经改变了。由于在机器中处理它们的方式可能不同，将这些操作和其他扑空分别开来是有用的。更新是一个单字的写操作，而不是整个缓存块的传送。因为数据是从它所产生的地方被推出来的，它可能在被需要之前到达消费者。即使对于生产者来说，更新和升级操作的时延和扑空的时延相比，可能不那么关键，这是因为它很容易被安排在处理器的关键路径之外（见第 11 章）。

不幸的是，和更新相联的流量是相当大的。这主要是由于同一个处理器对相同的块在一个读之前发出的多个写操作都要产生更新。而对作废协议来说，第一个写可能引起作废，但其他的可以简单地累积在本地块中，在回写或者向别的缓存提供数据时以一次总线事务传送完成（见例 5.12）。这种增加的流量引起冲突，能够大大增加扑空的代价。复杂的更新方案可能试图延迟更新的时机，以取得一种类似的效果（通过合并写缓冲中的写操作）或者用其他的技术来减少流量和改进性能（Dahlgren 1995）。然而，所增加的带宽要求支持更新所带来的复杂性、缓存块变大的趋势，以及多道顺序程序负载的收集鼠现象，使得基于更新的协议在业界用得越来越少。我们在第 8 章会看到，更新协议对于可扩展缓存一致性系统结构还有一些其他的问题，这进一步使得它对多处理器系统设计者的吸引力越来越低。

讨论了如何保持数据的一致性，我们现在来考虑在基于总线的多处理器中同步是如何管理的。

332 ~ 333

5.5　同步

在多处理器中，硬件和软件一种关键的相互作用表现在对同步操作的支持上。同步操作包含互斥、点对点事件和全局事件。在过去的若干年里，关于要多少硬件支持以及到底应该提供什么样的硬件原语来支持这些同步操作，有相当多的争论。而且结论不仅随时间变化，

还随技术工艺和设计风格的变化而变化。硬件支持有速度的优越性，但将功能由软件实现在代价、灵活性和对不同情况的适应性方面有优越性。Dijkstra（1965）和 Knuth（1966）的经典工作表明，只通过原子性的读和写操作就能够提供互斥（假设一个顺序同一的存储器）。然而，所有实际应用中的同步方法都依赖于某种原子性读－改－写操作的硬件支持，其中一个存储单元的值的读、修改和写回被原子性地完成，中间没有其他处理器对该单元的访问。从简单到复杂的各种同步算法都可以通过软件用这样的原语来实现。

指令集的历史也给我们提供了硬件对同步支持的演化过程。在 IBM370 指令集中的一个关键的增强就是包含了一个复杂的原子指令（比较并交换指令）来在单处理器或者多处理器系统支持并发程序设计中的同步。这种比较并交换指令将一个存储单元的值和一个寄存器的值对比，如果相等，则将该存储单元的值和另一个寄存器的值交换。Intel x86 允许给任何指令设置一个前缀，相当于加锁，使它成为原子性的；由于源和目的操作数是存储单元，许多这样的指令能用来实现各种原子操作，甚至涉及到不只一个存储单元。高级语言系统结构的支持者们还提出了用户层次的同步操作（例如锁和栅障）也应该直接由硬件来支持，而不仅是原子性的读－改－写原语；这就是说，同步“算法”本身应该由硬件实现。对于这个问题，在精简指令集的讨论中是很活跃的，主要由于 RISC 中存储访问的操作只涉及一个存储操作数的简单的装入和存放。Sparc 的方法是提供涉及寄存器和一个存储单元的原子操作，用一个简单的交换（将寄存器内容和存储单元内容原子性交换）和比较并交换。MIPS 在早期的指令集中没有安排原子性原语，IBM 的 Power 最初在 RS6000 也是如此。最终包含到 MIPS 中的原语是一个新颖的组合，涉及一个特别的装入和一个条件存放，将在本节后面讨论，它使得各种高层次的读-改-写操作都可以形成，不需要硬件一个个实现它们。从本质上讲，这一对指令，而不是单个指令能用来实现原子性的交换或者更复杂些的原子操作。这种做法后来也被 PowerPC 和 DEC Alpha 系统结构采用，现在是相当流行的。如同我们将要看到的那样，同步引出了一系列跨通信体系结构各个层次的折中。不仅许多高层的操作和低层的原语能由硬件来实现，应用所提出的同步要求也是变化很大的。

334

本节的要点是讨论同步操作如何能通过软件算法和硬件原语的组合，在一种基于总线的缓存一致的多处理器上实现。特别地，讨论互斥的实现，通过加锁-解锁对，通过标记实现点对点事件同步，通过栅障实现全局事件同步。让我们从考虑同步事件的各个成分开始。这将清楚地表明在硬件直接支持高层互斥和事件操作是困难的，而且可能使得实现很不灵活。然后，给定硬件只对最基本的原子操作提供支持，我们能够考察用户软件和系统软件在同步操作中的作用，最后较详细地考虑硬件和软件的设计权衡。

5.5.1　同步事件的组成部分

一个同步事件主要由三个成分组成：

1）*获取方法*：指的是进程用来获得同步权的方式（这里“同步”包括进入临界区或者向前推进，以通过事件同步机构）。

2）*等待算法*：这是进程用来等待同步可用的方法；例如，如果一个进程试图获取一把锁但该锁被别的进程占有着，或者想推进通过一个事件但事件还没有发生。

3）*释放方法*：这是进程用来使其他进程推进通过一个同步事件的方法；例如，Unlock 操作的一种实现，最后到达栅障的进程释放等待进程的方法，或者是通知一个等待在点对点

事件上的进程事件已经到达的方法。

等待算法的选择基本上是独立于同步类型的。有两种主要的选择：忙等待和阻塞。忙等待意味着进程在一个循环中不断测试某个变量是否改变了值。由另一个进程做的同步事件的释放改变这个变量的值，让等待进程可以向前推进。在阻塞情形，进程不执行循环，而是简单地阻塞（挂起）它自己，如果需要等待的话，还要释放处理器。当它所等待的释放信号出现时，它将被唤醒并且进入就绪状态。忙等待和阻塞相比的利弊是显而易见的。阻塞的开销较大，这是由于挂起和重启一个进程涉及到操作系统（挂起和重启一个线程涉及一个线程包的运行支持），但这种做法使得处理器能为其他进程或线程所用。忙等待避免了挂起的开销， 335
但在等待时消耗了处理器和缓存带宽资源。阻塞机制的功能要比忙等待强，这是因为如果不允许正在等待的进程或者线程运行，忙等待就永远不会结束[○]。在等待周期比较短的场合，忙等待可能会好一些；而如果等待时间比较长或者还有其他进程要运行，阻塞可能是一种更好的选择。也可以用一些混合的方法，例如让进程忙等待一会儿，如果超出了某个时间阈值，进程就阻塞，让其他进程投入运行（一种两阶段等待算法）。

用硬件实现高层同步操作的困难不在于获取或释放部分，而在于等待算法。这样，对于获取和释放方法的关键之处在于提供硬件支持，而用软件将三者结合起来是有道理的。不过，如何从硬件/软件相互作用方面实现忙等待中的循环操作仍然留有微妙但十分重要的问题。

5.5.2 用户和系统的角色

谁应该负责实现诸如锁和栅障之类高层同步操作的具体内容？典型地，程序员要用锁、事件或者更高层的操作，而不需要关心它们的内部实现。实现留给系统，系统来决定提供多少硬件支持，哪些功能用软件来实现。人们已经开发出了利用简单原子交换原语的软件同步算法，其性能可以和全硬件实现相媲美，而且它们所带来的灵活性和对硬件的简化是很有吸引力的。和系统设计的其他方面一样，由硬件提供更快操作的实用性取决于这些操作在应用中出现的频度。这样，我们又一次体会到最好的答案的作出在于对应用行为的最好的理解。

同步构造的软件实现通常包含在系统库中。好的同步库设计是相当具有挑战性的。一种潜在的复杂因素在于同一种同步（锁，栅障），甚至相同的同步变量，可能在不同的时间用于非常不同的运行条件。例如，对一把锁的访问可能是低冲突的（较少的处理器，其中可能只有一个在同一时间试图获取这把锁），也可能是高冲突的（许多处理器同时试图获取这把锁）。不同的情形提出了不同的性能要求。在高争用情况下，大多数进程将等待一段时间， 336
对锁算法一个关键的需求是要提供高的加锁－开锁传送带宽；在低争用的情况，关键目标是对锁的获取提供低延迟。不同的算法可能更好地满足不同的需求，于是我们必须或者找到一个好的折中算法，或者对不同的同步提供不同的算法供用户选择。如果我们运气好，一种灵活的库可能在运行时根据具体情况，选择最好的实现。不同的同步算法也可能依赖于不同的基本硬件原语，这样，某些算法可能特别适合一个特定的机器，而对其他机器并不适合。在多道程序情况下，进程调度和其他资源交互可能改变并行程序中进程的同步行为。和在专用条

○ 这种进程或线程得不到资源的问题在多处理器情形实际上要简单些。当进程在一个处理器上分时的情况下，没有强占的忙等待肯定是个问题。如果每个进程或线程有它自己的处理器，就肯定没有问题。在多处理器上的多道程序环境可以看作是在上述两种情况之间。

件下表现出低时延高带宽的简单算法相比，考虑到多道程序效果的复杂算法可能在实践中给出更好的性能。所有这些因素使得同步成为硬件/软件相互作用的一个焦点。

5.5.3 互斥

有许多算法可用于实现互斥（加锁-开锁）操作。简单的算法通常在对锁的争用很小的情况下是快的，但在高争用情况下则效率不高，而复杂的算法对于争用处理得较好，但在低争用时的代价较高。在关于硬件锁的一个简略讨论后，本节描述基于存储器的一个最简单的软件锁算法，该算法用了一种原子交换指令。之后，我们讨论这些简单的算法如何可以不用原子性交换指令，而用专门的装载-加锁和条件存储指令对来实现，以及有关的利弊。然后，我们会看看更复杂一些的算法，它们都可以用上述任何实现原子操作的方法来构成。

1. 硬件锁

尽管目前锁操作在基于总线的机器上不那么流行，但锁操作能够完全用硬件支持。在某些较老的机器上，一种做法是在总线上用一组锁线路，其中每一条线代表一把锁。持有锁的处理器负责设置该线的相应状态，等待该锁的处理器要等待这条线的释放。当有多个请求者时，一种优先级线路决定下一次该哪个处理器得到这把锁。然而，这种做法相当不灵活，因为在同一时间只有一个有限量的锁可用，并且等待算法是固定的（典型的是某种形式的忙等待，超时放弃控制）。通常，只是操作系统为一些特定的目的用这些硬件锁，这些目的之一就是在存储器中实现大量的软件锁。CRAY Xmp 用的就是这种做法的一种有趣的变形　一组
337 寄存器在处理器之间共享，包括若干固定数量的锁寄存器。尽管体系结构使得将寄存器分给用户进程成为可能，但这样的寄存器太少，在一种通用计算环境下这样做很不方便，因此在实践中这些锁寄存器主要用来在存储器中实现高层的锁。

2. 简单的软件锁算法

考虑一种为一个临界区代码提供原子性的锁操作。对于其获取方法，一个试图获得一把锁的进程必须检测锁是否是自由的；如果是的话，还要声明对于该锁的占有权。锁的状态可以存在于一个二进制变量中，0 表示自由，1 表示忙。考虑锁的获取操作的一种简单方法是，试图获得该锁的进程应该检测这个变量是否为 0，若是则将它置 1，这样就将该锁标为忙；如果是 1（忙），则进程应该按照一定的等待算法等待该变量变成 0。解锁（开锁）操作应该简单地将该变量置 0（释放方法）。下面是试图实现上述想法的汇编层次的指令。（在我们的伪汇编记号中，如果有第一个操作数，则它总表示操作的结果。）

```
lock:   ld    register, location  /*copy location to register*/
        cmp   register, #0        /*compare with 0*/
        bnz   lock                /*if not 0, try again*/
        st    location, #1        /*store 1 into location to mark it locked*/
        ret                       /*return control to caller of lock*/
and
unlock: st location, #0           /*write 0 to location*/
        ret                       /*return control to caller*/
```

这个锁过程要为它后面的临界区提供原子性，但它自己却保证不了自己的原子性。为认识到这一点，假定锁变量最初为 0，两个进程 P_0 和 P_1 执行上面的锁操作。进程 P_0 读到了 0，

认为锁是自由的，于是它通过了转移指令。它的下一步是置该变量为1，将它标记为忙；但在它做这件事之前，进程 P_j 读到了变量为0，认为锁是自由的，也通过了转移指令。于是我们现在有了两个进程，同时通过了锁，进入到同一个临界区，但这正是我们希望用锁来避免的。让存储指令紧跟着装载指令也无济于事。由两个指令构成的序列——读（测试）锁变量的值以检测它的状态，若它是自由的则向它写（置）1——不是原子性的，并且没有任何措施来防止这些操作在不同的进程中交织执行。我们需要的是一种方法，它可以原子性地测试一个变量的值，并且如果测试成功就将它置成另一个值（即，原子性地读并且有条件地修改一个存储单元），并且然后无论这个原子序列执行成功与否都要返回。向用户进程提供这种 338
原子性的一种方式是将锁代码放在操作系统中，通过一个系统调用来访问它，但这种开销很大并且操作系统自己怎么支持锁也是个问题。另一种可能性是在锁代码序列前后用一把硬件锁，但这要求有硬件锁，并且在现代处理器上显得慢。

对于这种锁问题，一种高效的通用解决方案是在处理器的指令集合中支持一种原子性的读-修改-写指令。典型的做法有一种原子交换指令：由指令给出的一个存储单元的值被读到一个寄存器中，并且另一个值被写到该单元中。操作是原子性的，其间不允许有其他对该单元的访问。这种操作有许多变形，通过所存储的值的特点提供不同的灵活性。一种适用于互斥的简单例子是测试并设置指令。在这种情况下，存储单元的值被读到一个特殊的寄存器，同时常数1被写入该单元，操作原子性完成。这种测试并设置指令的成功通过考察寄存器中的值来确定。如果是0，则指令成功。如果是1，就是不成功的；由这测试并设置指令写入存储器的1和其中原有的值是相同的，因此也没什么害处（1和0是典型值，其他常数也是可用的）。给定这样一条指令，记为 t&s，可以写出如下加锁和解锁代码：

```
lock:   t&s register, location
                          /*copy location to reg, and set location to 1*/
        bnz register, lock /*compare old value returned with 0*/
                           /*if not 0, i.e., lock already busy, so try again*/
        ret                /*return control to caller of lock*/
```

和

```
unlock:st location, #0     /*write 0 to location*/
        ret                /*return control to caller*/
```

在这种锁的实现中，程序不断试着用测试并设置指令来获取该锁，直到寄存器中出现由测试并设置留下的0值，表示在测试的时候锁是自由的（当时测试并设置指令也将存储单元置1了，从而获得了它）。解锁操作简单地将和锁相连的存储单元置0，指出这把锁现在自由了，从而使得任何进程在其后的锁操作能成功。一种简单的互斥结构于是用软件实现了，依赖的硬件支持是一条原子性的测试并设置指令。

这种原子性指令还有一些更复杂的变形，如我们将会看到的，它们在不同的软件同步算法中可能用到。一个例子是交换指令。像测试并设置指令一样，这种指令将特定存储单元的值读入特定的寄存器，但将该寄存器开始的任意值写到该存储单元中。也就是说，指令将存 339
储单元和寄存器的值做了交换。很清楚，用交换指令可以像前面一样来实现一把锁，只要我们用0和1并且保证寄存器的值在交换指令执行前为1；如果由交换指令留在寄存器中的值为0，则加锁成功。

另一个例子是一类取并操作（fetch&op）指令。这样的指令也是指定一个存储单元和一

个寄存器。它原子性地将存储单元的值读入寄存器并且将一个值（由这个取并操作指令指定的一个操作，作用在读出来的值所产生的）写入该单元。取并操作指令的最简单形式是取并加 1（fetch & increment）和取并减 1（fetch & decrement），将当前值增减 1。取并加（fetch & add）指令则用另外一个操作数，可能来源于一个寄存器或者立即数，加到该单元先前的值上去。一种更复杂的原语是比较并交换（compare & swap）指令。它取两个寄存器和一个存储单元（即，它是一个 3 操作数指令，RISC 体系结构通常不支持）；它将存储单元的值和第一个寄存器比较，如果相等，就将存储单元内容和第二个寄存器内容交换。

3、简单锁的性能

图 5-29 示出 SGI Challenge[○] 中的一种简单的测试并设置锁的性能。性能是通过在一个循环中重复执行下述微测试程序来得到的：

```
lock(L);
critical-section(c);
unlock(L);
```

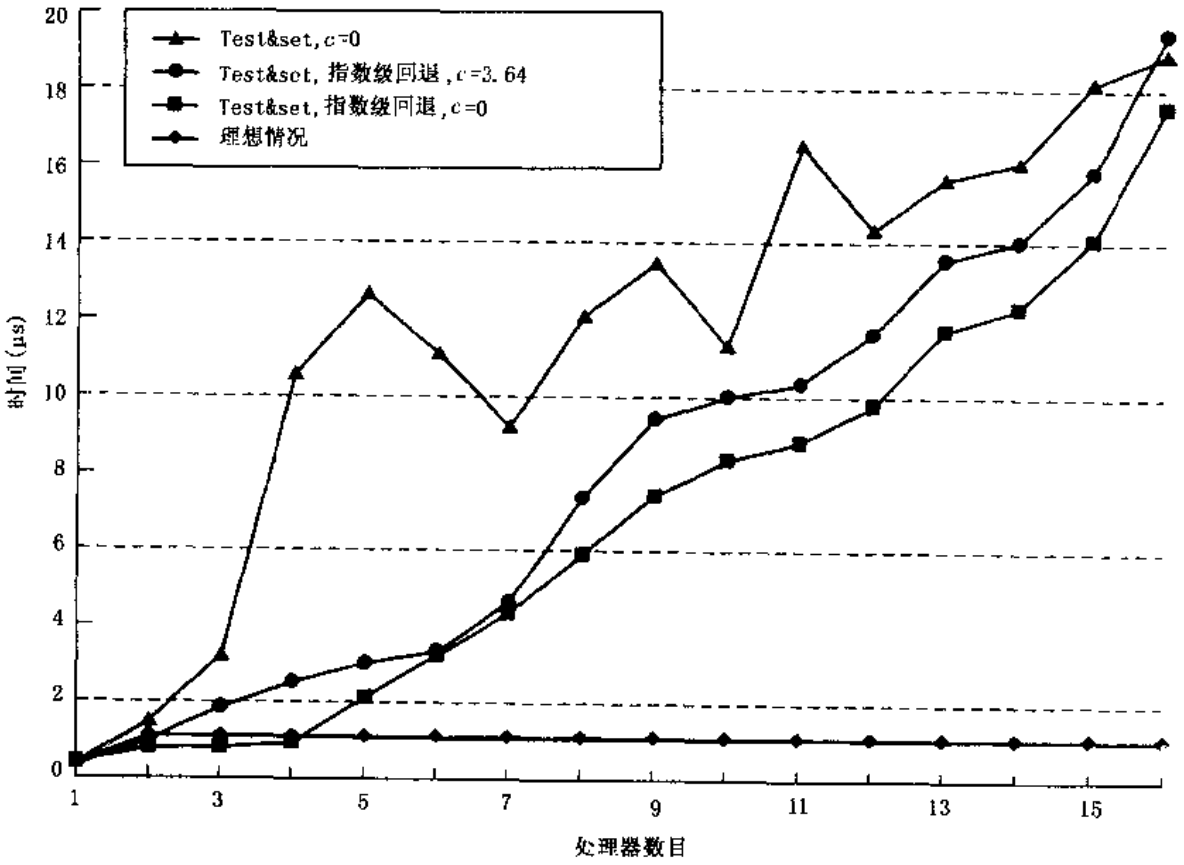


图 5-29　在 SGI Challenge 上，随处理器个数的增加，一种综合测试并设置（test & set）锁的性能。y 轴是每加锁－解锁操作对所花的时间，不包括临界区中的 c 微秒延迟。最上面一条曲线的非规则性是由于处理器的争用对时间的依赖性所引起的

○　事实上，用于本章来说明同步性能的 SGI Challenge 上的处理器并不提供测试并设置指令。它所用的是另外一种原语，本节后面将会谈到。对于这些实验来说，我们用其他一些原语综合出一种在行为上类似于测试并设置指令的实现。关于一些较早的机器，例如 Sequent Symmetry，人们得到过用基于真实测试并设置指令构成的锁的结果（Graunke and Thakkar 1990; Mellor-Crummey and Scott 1991）。

其中 c 是一个延迟参数，决定临界区的大小（这里它只是一个延迟，不做什么实际的事情）。这个测试程序的配置使得无论处理器的个数多少，锁调用的总数不变，这对应于一个固定的任务数在一个集中式的任务队列中要被处理，独立于参与工作的处理器的个数。性能评价的单位是每次锁的传递所花的时间，即所有处理器执行这个测试程序所花的总时间除以锁被获得的次数。在临界区中所花的总时间（即 c 乘以锁被获得的次数）要从总执行时间中减掉，从而得到锁传递所花的时间（也就是包括了锁操作引起的竞争）。所有测量单位为 μs。

图 5-29 中上面的曲线是用测试并设置锁、临界区很小的情况（暂时忽略带有回退的曲线）。理想情况下，我们希望看到每次获取锁的时间独立于参与竞争的处理器的个数，每次锁转移只有一次总线活动，如图中标有“理想情况”的曲线所示。然而，图中的情形显然表示了性能随处理器个数的增加而降低。

这里的问题在于等待方法期间所产生的流量：每次试图测试锁是否自由，无论成功与否，都产生一个对于含有该锁变量的缓存块的写操作（由于它用一个测试并设置指令，向该单元写 1）；由于这个块当前在某个别的处理器的缓存中（当它做测试并设置时向该块写过后），于是每一个写产生一个总线事务来作废该块先前的拥有者。这样，所有处理器不断发出总线事务，甚至在等待算法期间也消耗珍贵的总线带宽。随着处理器数的增加，从而也是测试并设置指令和总线过程的频度增加，所导致的竞争大大减慢了锁的转移。在实际中，它将影响在临界区中所做的工作。在总线上的大量竞争以及所导致的获得锁的时间依赖性引起测试时间随处理器数，甚至同样的处理器数但不同的执行实例，大幅度的变化。图 5-29 中所表现的是一组特殊的、有代表性的执行，针对不同的处理器数的结果。

340
|
341

4. 简单锁算法的增强

我们可以通过两个简单的措施来减缓这种流量。首先，我们可以减少进程在等待时发出测试并设置指令的频率；其次，我们可以让进程只是在读操作时忙等待，这样在锁被释放前它们就不产生作废和扑空。这两种做法称为带有回退（backoff）的测试并设置锁和测试-测试并设置（test- and -test & set）锁。

带有回退的测试并设置锁 回退的基本思想是让进程在一次试图获取锁的操作失败后插入一个延迟。在测试并设置指令之间的延迟不可以太长；否则当锁自由了，处理器还可能处于空闲状态。但这个延迟又应该足够长，以使得流量真正降低下来。一个自然的问题是这个延迟量应该是固定还是变化。实验结果显示，好的结果是让延迟按“指数规律”变化；即第一次尝试后的延迟是一个小常数 k，它随次数按几何级数增加，这样在第 i 次尝试后，它就是 $k \times c^i$，其中 c 是另一个常数。这样的锁称为带有指数回退的测试并设置锁。图 5-29 也示出了两种这样的情况，不同之处在于临界区的大小，所选的 k 对于性能测试来说是最好的。性能得到了改进，但可扩展性仍然不怎么好，这是由于在释放和获取时还是有大量的流量相干。在文献（Granuke and Thakkar 1990; Mellor-Crummey and Scott 1991）中有关于一些较早机器用真实测试并设置指令带有回退的性能结果。习题 5.14 还讨论了为什么用回退方法时非空临界区的性能要比空临界区的性能差。

测试-测试并设置锁 算法的一个更精细的变化是让它在忙等待时用一些不产生这么多总线流量的指令。进程忙等待时重复执行一个标准装入指令，而不是测试并设置，将锁变量读入，直到它从 1（加锁）变到 0（解锁）。在缓存一致性机器中，所有处理器都可以在缓存

中进行这样的读操作，而不产生总线流量，这是由于每个进程在第一次读的时候得到了该锁
342 变量的一个缓存拷贝。当这把锁释放时，所有等待的进程缓存中的拷贝都被作废，下一次读就要产生一次读扑空。等待进程之后会发现锁已经可用了，然后就执行测试并设置指令来实际试图获取该锁。其中之一将成功，其他的将失败，并返回到基于读的等待方法。这样的测试-测试并设置锁能大大减少总线流量。

5．锁的性能目标

在考察更复杂的锁算法和原语之前，有必要清楚地指出对于锁的性能目标，并且回顾一下我们关心锁的哪些表现。目标包括：

- 低时延。如果一把锁是自由的并且没有其他处理器同时试图获取它，一个处理器应该能够以低时延获得它。
- 低流量。如果许多或者所有处理器同时试图获取一把锁，它们应该能够一个接一个得到该锁，尽量不产生流量或总线事务。如前面所讨论过的，除了不相关的过程争用总线外（包括在临界区中的），由于高流量所导致的竞争能减慢锁的获取。
- 可扩展性。时延和流量都不应该随处理器数的变化而变化太大。然而，由于在基于总线的 SMP 中处理器的数目不会太大，重要的不是渐近可扩展性，而是在一定实际范围的可扩展性。
- 低存储代价。一把锁所需的信息应该很少并且不应该随处理器数增加很多。
- 公平性。理想上，处理器应该以它们发出请求的次序获得一把锁。至少，应该避免饥饿或者明显的不公平。由于饥饿的可能性通常不大，公平性的重要性要和它对性能的影响权衡考虑。

考虑简单的原子交换或者测试并设置锁。如果相同的处理器在没有竞争的情况下重复获取一把锁，它的时延是很小的，这是由于所执行的指令数很小并且锁变量将在处理器的缓存中。然而，我们已经看到在许多处理器竞争的情况下锁会产生大量的总线流量。这种锁的性能随着处理器数的增加扩充性很差。存储代价低（只是一个变量）并且不随处理器数变化。这种锁没有任何措施来保证公平性，运气不好的处理器可能挨饿。带有回退的测试并设置锁在非竞争情况下有同样的时延，产生较少的流量，可扩展性稍好一些，不取更多的存储并且公平性也没改善。和简单测试并设置锁相比，测试-测试并设置锁无冲突的时延稍大一些（即使在没有竞争的情况下，它也要多做一个读操作），但产生的总线流量要小得多，而且可扩展性也好些。它要求的存储也可以忽略，也不保证公平性。（习题 5.12 让你计算测试-测
343 试并设置类型的锁在不同场合下总线事务的次数，还有所花的时间。）

在这种测试-测试并设置锁中，由于测试并设置操作（也就是对应一个总线事务）只是在处理器注意到锁可用，并且它在忙等待一个缓存块失败的情况下才进行，所以没有回退的需要。然而这种锁也有问题，即当锁被释放时，所有等待的进程几乎会同时冲出来，进行它们的读扑空，执行它们的测试并设置指令。对于这些读扑空的总线事务可以组合在一种聪明的总线协议中；然而，每个测试并设置指令本身产生作废和后续扑空，对于 p 个处理器每个要求一次锁，会导致 $O(p^2)$ 总线流量。在发出测试并设置之前插入一个随机的延迟至少能够有助于拉开这些测试并设置指令，但它将增加在无竞争情况下获取锁的时延。测试-测试并设置方式曾经是当时一个比较大的进步，但后来人们又设计了更好的硬件原语和更好的算法来减缓它的流量问题。

6．改进的硬件原语：Load-Locked，Store-Conditional

除了在忙等待中用读代替读－改－写外，我们还想尝试让读－改－写失败的进程不产生作废。而且，希望有这样一个原语，能够实现多种原子性的读－改－写操作，诸如测试并设置，取并操作，比较并交换等，而不是要用单独的指令分别实现。在现代微处理器中，有一种用得越来越多的方式，能达到这两个目标。它的基本精神是用一对特殊指令，而不是单个读－改－写指令来实现对一个变量的原子性访问（让我们称这个变量为同步变量）。第一条指令，通常称为 Load-Locked（装入－加锁）或者 Load-Linked（装入－链接）指令（LL），将同步变量装入一个寄存器。它后面可以跟着任意的指令，这些指令完成对该寄存器中的值操作，即对应读－改－写的修改部分。这个序列的最后一条指令是第二个特殊指令，称为条件存放（store-conditional）。它试图将寄存器的值写回去（同步变量），条件是当且仅当没有其他处理器在本处理器完成了 LL 后，对该单元（或者缓存块）进行写操作。这样，如果条件存放成功，它意味着装入－加锁、条件存放（LL-SC）指令已经原子性地对该变量进行了读、可能的修改以及回写操作。如果条件存放检测到其间对该变量或者缓存块发生了一次写，它就不会试图将值回写（或者产生任何作废）。这意味着对于该变量的原子操作失败，必须从 LL 再开始尝试。条件存放的成功或者失败由条件码或者返回值来确定。LL 和条件存放是如何实现的将留在后面讨论；现在，我们关心它们的语义和性能。

利用 LL-SC 来实现原子操作，前面那种简单的加锁和解锁算法能如下实现。其中 reg 1 是存储单元当前值装入的寄存器，reg 2 持有要被存回的值（如同测试并设置，对于一次加锁操作，reg 2 可以就是 1）。

344

```
lock:   ll   reg1, location    /*load-locked the location to reg1*/
        bnz  reg1, lock        /*if location was locked (nonzero),
                                 try again*/
        sc   location, reg2    /*store reg2 conditionally into location*/
        beqz lock              /*if store-conditional failed, start again*/
        ret                    /*return control to caller of lock*/
```

和

```
unlock: st location, #0        /*write 0 to location*/
        ret                    /*return control to caller*/
```

多个处理器可能同时执行 LL，但只有第一个将它的条件存放指令放到总线上的才可获得成功。这个处理器将成功地获得这把锁，其他的将失败，不得不重新从 LL-SC 开始。注意，条件存放失败的途径有两个，一是它在试图访问总线之前就检测到其间发生了一次写；二是在它试图访问总线，但发现某个其他处理器的条件存储已经先到达那儿了。当然，如果当进程执行 LL 时该单元是 1（非 0），它将把 1 装入 reg 1，并且将从 LL 重新开始，根本不去尝试条件存放。

值得注意的是，LL 本身不是一种加锁操作，条件存放本身也不是解锁。首先，LL 的完成本身不意味着获得了排他的访问权；事实上，LL 和条件存放一起用来实现一种加锁的操作。其次，即使 LL-SC 对成功了，也不保证在它们之间的指令（如果有的话）的执行是原子性的，因此事实上这些指令不构成一个临界区。一个成功的 LL-SC 指令对惟一能保证的是在 LL 和条件存放之间没有发生对于同步变量的写。事实上，由于在 LL 和条件存放之间的指令是无条件执行的，但如果条件存放失败应该是不可见的，所以重要的是它们不修改任何其他

重要的状态。典型地，这些指令只是操作同步变量装入的寄存器（例如，完成取并操作中的操作部分）不修改程序中任何其他的变量（修改这个寄存器是没有问题的，因为这个寄存器总是要由下一次 LL 重新装入）。显式支持 LL-SC 的微处理器厂家鼓励软件编写人员遵循这种要求，并且还经常说明哪些指令在其间可用，能够保证它们所实现的 LL-SC 操作的正确性。在 LL 和条件存放之间的指令数也应该很小，以减少由于别的处理器插入其间的写导致的条件存放失败的可能性。尽管 LL 和条件存放不构成一个加锁－解锁对，它们可以直接用来实现在共享数据结构上的某些原子操作。例如，如果所希望的功能是在一个全局变量上的一个小操作（例如一个计数器或者全局和），和在变量更新周围做加锁和解锁相比，用一种自然
345 的指令序（LL、寄存器操作、条件存放、测试）实现起来会显得更有道理。

如同 test-and-test&set 指令，当 LL 指出当前锁被别人持有时，用 LL-SC 建立起来的锁在等待算法期间不会产生总线流量。比 test-and-test&set 优越之处在于它在一次获得锁的尝试失败后也不产生作废（即一次失败的条件存放）。然而，当锁被释放时，在一个装载－加锁操作循环上的踏步等待的处理器很可能会在该单元上扑空，并且向总线产生读事务。在这之后，对给定锁的一次获取只会由条件存放成功的处理器产生一个作废，但这会再次作废所有的缓存。即使和 test-and-test&set 相比，流量大大地减少，并且没有读－改－写总线事务，但流量依然随处理器数线性增加，即每次获取锁意味着 $O(p)$ 次总线事务。由于在一个上了锁的单元上踏步等待总是通过读来进行的（装载－加锁操作），不可能有什么类似的情形可以进一步改进 test-and-test&set 指令的性能。然而，在 LL 和条件装入之间可以用回退以减少突发性流量。

简单的 LL-SC 锁在时延和存储要求方面也是低的，但它也不是一种公平的锁并且所引起的流量也不是最小的。更高级的锁算法能够既提供公平性，也减少流量。它们可以用原子性的读－改－写指令来实现，也可以用由 LL-SC 综合得到的等价语义的原子操作实现。当然，这两种做法的优点是不同的。让我们考虑这样两个算法，它们适合于基总线的机器。

7．高级的锁算法

特别地，当用一种类似于测试并设置的原子交换指令（不是 LL-SC）来实现锁的时候，我们希望当锁被释放时只有一个进程实际上试图去获取该锁（而不是像所有前面的算法那样，让所有进程都冲出来做测试并设置，并且发出作废信号）。更希望的是当一把锁释放时只有一个进程发生读扑空（对 LL-SC 也希望如此）。加号锁达到了第一个目的；基于数组的锁两个目标都达到了，代价是需要稍多一点空间。和所有前面的锁不同，这样两种锁是公平的并且以 FIFO 序让处理器得到锁。

加号锁　加号锁的操作就像在餐馆排队买三明治，或者像银行的出纳队列的票号系统。需要获得锁的进程取一个票号，然后在一个全局 now-serving（现在服务）号上忙等待（就像我们在买三明治队列上刻意观察的 LED 显示屏上的数字）直到 now-serving 号等于自己所得的票号。要释放一把锁，进程只要将 now-serving 号加 1，于是下一个等待的进程能够获得该锁。所需的原子性原语是，当进程首次到达锁操作，要从一个共享计数器上获得它的票号时用到这个原语。由于只有票号等于 now-serving 的过程在看到锁被释放后试图进入临界区因此在实际获得锁时不需要做原子操作。这样，获取方法是 fetch&increment，等待算法是用忙等待
346 的方式检查 new-serving 是否等于自己的票号，释放方法是增加 now-serving 。这种锁的不竞争时延大约和 test-and-test& set 锁相等，但所产生的流量要小得多。尽管每一个进程在到达该锁时都做取并加 1 操作（假想所有进程不是同时到达），在锁的一次释放后测试并设置就不必要

了，而那通常更趋向于同时性，并且竞争性要严重得多。由于进程是以它们执行 fetch&increment 操作的次序得到该锁的，加号锁对存储的需求是固定的和小量的，并且它还是公平的。

加号锁所需的取并加 1 操作可以用 LL-SC 实现。然而，由于简单的 LL-SC 锁已经避免了多个处理器在锁被释放后试图获取它时发出作废信号，在流量上加号锁和简单的 LL-SC 锁没有大的区别。（简单的 LL-SC 锁稍微差一些，这是由于当一个处理器在它的条件存放上成功时，另外一个作废和一组读扑空会出现。）在这两种锁上面的关键区别在于公平性。

像简单的 LL-SC 锁，在释放中加号锁也有一个读流量的问题。原因是所有进程在相同的变量（now-serving）上踏步等待。当那个变量被释放写入后，所有处理器的缓存中的拷贝被作废，它们都要引发一次读扑空。这些读扑空在有些总线上可能结合起来，但如果没有结合功能或者结合失败，则会引起不必要的流量。一种降低这种突发性读扑空流量的方式是引入一种形式的回退。我们不想用指数性回退，因为在锁被释放时，我们不想让所有处理器都回退，从而过一会儿谁都不想尝试获取它。一种有前途的技术是让每个处理器在试图读 now-serving 计数器时回退一段正比于它期望轮到它所需的时间——即正比于它的票号和它上一次读到的 now-serving 值的差。作为另一种方案，基于数组的锁完全消除了这种在释放时额外的读流量，办法是让每个进程在不同的单元上循环。

基于数组的锁　这里的想法是用一种 fetch&increment，来获得一个惟一的单元而不是值，然后在这个单元上做忙等待。如果有 p 个进程可能争用一把锁，那么这把锁的数据结构包含一个含有 p 个单元的数组，供进程在上面循环等待，理想情况是不同的单元分布在不同的存储块上，以避免伪共享。这里，获取方法就是用一个 fetch&increment 操作来获得该向量中下一个可用的单元（带回卷），等待方法则是在这个单元上循环，释放方法是向下一个单元写入一个表示“解锁”的值。这种释放使得在下一个单元上等待的处理器发生缓存块的作废；它后面的读扑空告诉它，它已经得到了这把锁。如同加号锁的情形，在扑空后不需要做 test&set，这是由于当锁被释放时只有一个进程被通知。显然，这种锁也是 FIFO，因此是公平的。它的无竞争时延可能类似于 test-and-test&set 锁（一种 fetch&increment，跟着一个对所分配数组单元的读），并且它显然潜在地要比加号锁更具有可扩展性，这主要是因为只有一 347
个进程发生读扑空。基于同样的原因，不同于加号锁，它不需要任何形式的回退来减少流量。在基于总线的机器上，它的惟一缺点是它用 $O(p)$ 空间，而不是 $O(1)$，但对于较小的 p 和比例常数，这通常不是一个很重要的缺点。对于分布存储机器，它有一个潜在的缺点，我们将在第 7 章讨论它和克服它的锁算法。

8、性能

让我们简略地考察 SGI Challenge 上不同的锁的性能，如图 5-30 所示。所有的锁用 LL-SC 实现（由于 Challenge 只是提供这些，而没有提供原子性指令）。所得到的结果是基于前面提到过的微基准测试程序的一个更参数化的版本，其中一个进程不仅可以在临界区插入延迟，在锁的释放和它的下一次尝试之间也可以插入（如同真实程序会发生的）。即，代码是一个以下面为主要部分的循环：

```
lock(L);
critical_section(c);
unlock(L);
delay(d);
```

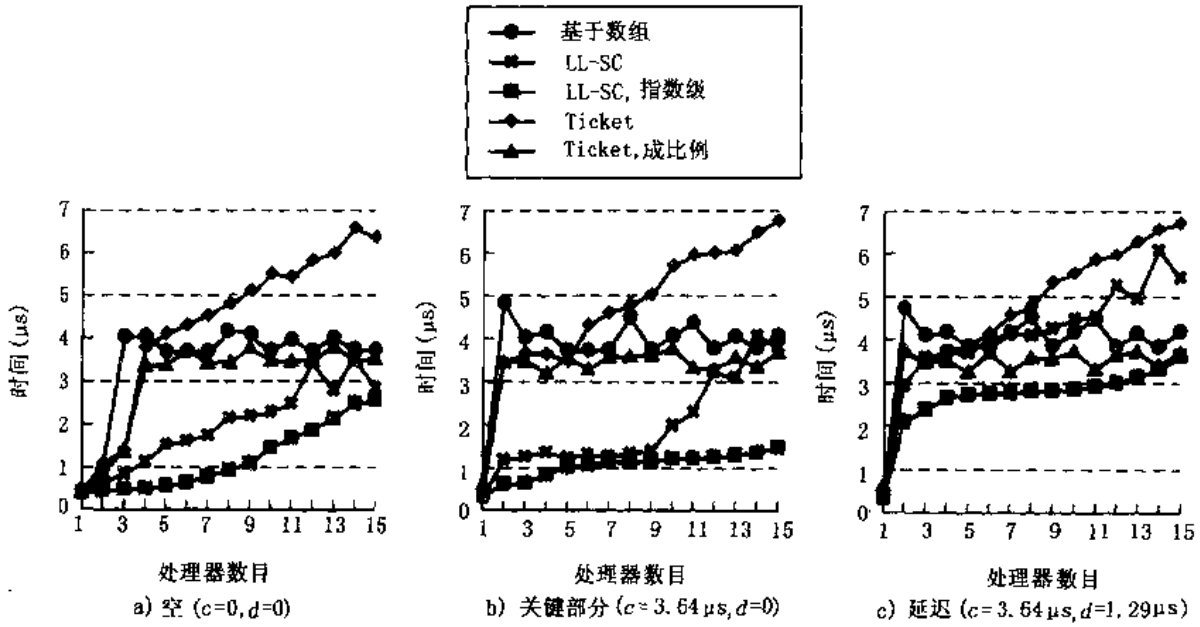


图 5-30　对于三种不同的情形，SGI Challenge 上锁的性能

考虑三种情形：1）$c=0$，$d=0$；2）$c=3.64\mu s$，$d=0$；3）$c=3.64\mu s$，$d=1.29\mu s$。它们分别称为空临界区情形、非空临界区情形和带有延迟的非空临界区情形。延迟 c 和 d 以处理器周期数为单位插入到代码中，转换为这里的微秒数。和先前讨论过的一样，延迟 c 和 d（乘以每个进程获取锁的次数）要从总的时间中减掉，它只是来测量一定数量的锁的获取和释放所花的时间（见习题 5.15）。

考虑空临界区的情形。比较图 5-30 和图 5-29，首先观察到的是所有其他锁的确都比测试并设置锁要好，这是已估计到的[○]。第二个观察是，简单的 LL-SC 锁实际上看来性能要好于比较复杂的加号锁和基于数组的锁。对这些锁来说，它们所遇到的竞争不像 test&set 锁遇到的那么多，性能主要由释放和成功获取之间的总线过程数来决定。尤其对于较少处理器的情况，LL-SC 锁表现特别好的原因是它们的不公平性，体系结构的相互作用利用了这种不公平性！特别是，当一个进程通过一个写操作释放了一把锁，紧跟着用一个读（LL）试图进行下一次获取，这个读和后续的条件存放可能在它的缓存中成功，先于其他处理器能越过总线来读这个存储块。（SGI Challenge 上的偏向实际上要更严重，这是由于释放的处理器能够从它的写缓冲区中来满足它的下一次读，甚至在对应于释放写的排他读达到总线之前。）锁的传递很快并且性能也很好，但同样的处理器可能重复获得自己释放的锁。随着处理器数和对总线竞争的增加，最后释放者的条件存储成功获得总线的机会减少，因此自传送的可能性减少。除此以外，总线流量会由于作废和读扑空增加，因此每次锁转移的时间增加。指数型回退有助于减少流量的突发性，从而也减缓了扩展的速率。非空的临界区（$c=3.64$，$d=0$）对此会有进一步的帮助。

对于在临界区内外都有延迟的情形（$c=3.64$，$d=1.29$），即使处理器数不多，LL-SC锁

○ 这里的测试并设置是通过 LL-SC 如下模拟的：每当一个条件存放失败，就向同一存储块的另一个变量做一个写操作，于是就好像 test&set 那样引起作废，这种模拟的方法可能导致的性能要比真正用 test&set 原语差，但它反映了趋势。

的性能也不怎么好。这是由于处理器在释放锁后，试图再次获取锁之前需要等待，很可能使得其他等待着的处理器先于它获得了该锁。自我传递不太可能，因此即使在两个处理器的情况，锁的传递也是比较慢的。有意思的是，在处理器数量少，解锁加锁之间有延迟 d 时，回退的使用表现的性能特别差。这是由于当一个处理器释放了锁，在等待 d 的期间，所有其他处理器都在回退周期，根本还没有试图来取锁。在 $d=0$ 情形下，释放锁的处理器马上又获取锁，这种情况在处理器数较少时特别明显。从上面可以看到，回退技术必须慎用才可能成功。

348
~
349

考虑另外一些锁。它们是公平的，因此每次都传递到不同的处理器，且在传递的关键路径上涉及到总线事务。因此，即使只有两个处理器，它们都以一个跳转开始。每次传递都在关键路径上大约跳三个总线事务。时间方面的实际区别取决于产生哪种总线事务以及它们有多少时延能被隐藏起来，不被处理器可见。不带回退的加号锁可扩展性相对较差：当所有处理器试图读 now-serving 计数器时，在释放和由正确处理器读得之间的总线事务的期望数是 $p/2$，导致在锁传递的关键路径上的线性性能损失。采用比例回退技术，正确的处理器可能是在释放后第一个发出读的处理器，这样每次传递的时间就是常数，不随 p 变化。由于只有正确的处理器发出一个读，基于数组的锁也有较好的可扩展性。

这些结果表现了在确定锁的性能方面体系结构诸要素之间相互作用的重要性。它们还表明，只要总线有足够带宽，简单的 LL-SC 锁在总线上的性能会相当好。在这个特定的机器上，因为总线带宽相当高，非公平 LL-SC 锁的性能对于多于 16 处理器的情形，不会比更复杂的锁差多少。当采用回退技术来降低流量后，简单 LL-SC 锁在所有情形下都有最好的平均锁传递时间。然而，这些结果也表明了在评估同步算法方面有效实验方法的难度和重要性。空临界区展示了某些有趣的效果，但有意义的比较取决于在实践（真实应用）中同步模式的具体情况。例如，临界区和延迟大小，对于自传递的作用，对比较公平和不公平锁来说，影响是实质性的。空情形不具有真实代表性，但有重要的方法论意义。人们还做过一个实验，用 LL-SC 同时通过附加变量来保证处理器（公平性）获得锁的循环，表明其性能类似于加号锁。这证实了非公平和自传递在处理器不多时的确是较好性能的原因。特别是，如果期望公平，带有比例回退的加号锁和基于数组的锁在基于总线的机器上表现得很好。

9．免锁的、非阻塞的以及免等待的同步

当机器用在多道程序设计的环境下时，涉及到同步的还有对其他一些性能的考虑。在这种环境下，其他进程会不时执行一段时间；即使我们独占这个机器，后台驻留程序也会周期运行，进程会发生缺页，会出现 I/O 中断以及进程调度器只能基于应用需求不完整的信息来作出调度决定。这些事件能引起进程推进速度在很大的范围内变化。一个重要的问题是，当其中一个进程被减慢，并行程序作为一个整体会如何减慢。对于传统的锁来说，这个问题可能很严重：如果一个进程持有一把锁，在它的临界区中停止或者减慢，所有其他进程可能都要等待。这个问题在操作系统调度器的研究方面受到了广泛的关注。在有些情况下，人们试图避免剥夺持有锁的进程的控制权。另外一些研究认为基于锁的操作不健全，应该避免；例如，如果一个进程在持有锁时死掉，其他进程就都没法推进。人们观察到，大多数加锁-解锁操作都用来支持在典型数据结构和对象上的操作，那些结构是由多个进程共享的，例如，更新一个共享的计数器或者操作一个共享的队列。这些在数据结构上的高层操作能够直接用原子原语实现，而不要用锁，如同先前讨论 LL-SC 所提到的那样。

350

一个共享数据结构被称为是*免锁的*，如果在它上面定义的操作不要求在多条指令上互斥。如果在数据结构上的操作，能够保证某个进程在有限时间里，即使其他进程停止也能完成它的操作，这种数据结构就是非阻塞的。如果操作能担保每一个进程都能在有限的时间里完成其操作，该数据结构就是*免等待的*（Herlihy 1993）。针对这种数据结构的理论和实践，人们做过一些研究，包括实现它们的基本原语（Herlihy 1988）、将顺序操作转换为非阻塞并发操作的通用技术、各种专门的免锁数据结构（Valois 1995; Michael and Scott 1996）、操作系统实现以及对系统结构支持的提议等等。这里基本的出发点是要实现对共享数据结构的更新，先读出其中一部分，做一拷贝，更新该拷贝，然后在没有已经发生冲突更新的情况下才执行一个操作，以确认这个改变（这会使我们想到 LL-SC）。作为一个简单的例子，考虑一个共享计数器。计数器读到一个寄存器中，给寄存器拷贝加上一个值，结果放到第二个寄存器。然后，只是当共享计数器的值和原来是相同时，才用一个比较/交换来更新共享的计数器。对于更复杂的链表数据结构，就要创建一个新的元素，如果插入仍然有效，才将它链到共享表中。这些技术用来限制共享数据结构处于非一致状态的窗口的大小，于是它们增强了健壮性；当然，要把它们做得高效可能是困难的。

根据用不同的原子交换操作来实现同步变量访问的时间复杂性，人们发现了这些原子交换操作的一些性质。特别是，人们发现像 test&set 和 fetch&op 那样的简单操作不足以保证由处理器访问一个同步变量的时间独立于处理器个数，而更复杂一些的操作，例如比较/交换和交换两个存储单元的值，就能够达到这种保证（Herlihy 1988）。

[351] 讨论了基于总线机器的互斥方式后，下面来考虑点对点事件同步，然后是栅障事件同步。

5.5.4　点对点事件同步

在一个并行程序中的点对点同步，通常的实现方式是用普通变量作为标记，在上面做忙等待。如果我们要用阻塞，不用忙等待，则我们能用信号灯，就像在并发程序设计和操作系统中所用的那样（Tanenbaum and Woodhull 1997）。

1．*软件算法*

标记是控制变量，专门用来通报同步事件的出现，而不是要传递值。如果两个进程在共享变量 a 上有一种生产者－消费者关系，那么就能使用如下一个标记来管理同步变量：

P_1	P_2
`a = f(x);  /*set a*/`	`while (flag is 0) do nothing;`
`flag = 1;`	`b = g(a); /*use a*/`

如果我们知道，变量 a 初始为某个值（比如，0），将被这个生产事件变到一个我们感兴趣的一个新值，那么我们能够用 a 本身作为同步变量，如下所示：

P_1	P_2
`a = f(x);  /*set a*/`	`while (a is 0) do nothing;`
	`b = g(a); /*use a*/`

这就消除了对一个单独标记变量的需要，节省了对那个变量的读和写，可能的代价是程序的可读性和可维护性。

2．*硬件支持：满－空位*

这种特别标记值的想法在有些研究型机器中得到进一步发展（尽管主要在物理分布存储

的机器里），以提供对细粒度生产者－消费者同步的硬件支持。让存储器的每个字都和称为满－空位的状态相联。这一位被置1，表示“满”，即该字装有新数据（即写操作上），置0，如果这个字被消费该数据的处理器“腾空”（即读操作后）。字级别的生产者－消费者同步然后可以如下完成。当生产者进程要向单元中写，如果看到满－空位是空，就将它置满。消费者要读，如果看到满，就置空。硬件保证带有对满－空位操作的读和写的原子性。给定满－空位，我们前面的例子就可以写成如下没有踏步循环的样子： 352

P_1	P_2
a = f(x); /*set a*/	b = g(a); /*use a*/

满－空位引起关于灵活性的一些问题。例如，它们难以对付单生产者－多消费者同步，或者生产者在消费行为之前多次更新值的情况。还有，所有读和写都应该用满－空位，还是只是编译到特殊指令的操作用？后者要求在语言和编译器中有支持，但前者将同步强加到所有单元访问中（例如，它不允许迭带方程求解器中的异步放松，见第2章），局限性太大。鉴于这些原因以及硬件代价，在大多数商用机器中满-空位都没有受到青睐。

3．中断

另一种重要的事件是中断，主要源于需要处理器关注的I/O设备。在单处理器机器中，中断该谁来处理不是一个问题，但在SMP中，任何处理器都可能承担中断任务。除此以外，还有一个处理器向另一个处理器发中断的情况。在早期的SMP设计中，提供特殊硬件来管理每个处理器上进程的优先级，将I/O中断送到低优先级的处理器上。这样的做法证明价值不大，多数现代机器用的是简单的仲裁策略。除此以外，通常存在一种存储映射的中断控制区，于是可以在内核层次，通过在相关的地址上写入中断信息，使得任何处理器能中断任何其他处理器。

5.5.5 全局（栅障）事件的同步

最后，我们考察一下在基于总线机器上的栅障同步问题。栅障软件算法的实现通常都用锁、共享计数器和标记单元。首先我们看 p 个进程之间的一种简单的栅障，称为集中式栅障，它只用一把锁、一个计数器和一个标记位。

1．集中式软件栅障

用一个共享的计数器来记录到达栅障的进程数，每一个到达的进程使它加1。这些加1操作必须是互斥的。在对该计数器作了加1操作后，进程检查计数值是否等于 p，即它是否是最后一个到达的进程。如果不是，它就进入忙等待状态，等在和栅障相关的标记位上；如果是，它就置标记位，释放 $p-1$ 个等待着的进程。于是，我们可能提出栅障算法的一个简单实现如下： 353

```
struct bar_type {
  int counter;
  struct lock_type lock;
  int flag = 0;
} bar_name;

BARRIER (bar_name, p)
{
```

```
    LOCK(bar_name.lock);
    if (bar_name.counter == 0)
      bar_name.flag = 0;                    /*reset flag if first to reach*/
    mycount = bar_name.counter++;           /*mycount is a private variable*/
    UNLOCK(bar_name.lock);
    if (mycount == p) {                     /*last to arrive*/
      bar_name.counter = 0;                 /*reset counter for next barrier*/
      bar_name.flag = 1;                    /*release waiting processes*/
    }
    else
      while (bar_name.flag == 0) {}; /*busy-wait for release*/
}
```

2. 带有感应逆转的集中式栅障

能看出上面的栅障有什么问题吗？我们会发现有一个问题。它发生在同样栅障变量上连续进行栅障操作的时候——例如，如果每个处理器执行下面的代码：

```
some computation...
BARRIER(bar1, p);
some more computation...
BARRIER(bar1, p);
```

第一个进入栅障的进程第二次进入时重新初始化栅障计数器，因此没问题。问题在于标记位。为了从第一个栅障出来，进程踏步在标记上直到它为 1。看到标记为 1 的进程将离开
354 栅障，进行后面的计算，再次进入栅障。然而，假设一个处理器 P_x 没有来得及看见标记变成了 1，其他进程又进入了第二个栅障；这种情况是可能的，例如由于等待时间太长而被操作系统交换出去了。当它再次被交换进来时，它就要继续等在标记位上。同时，其他进程可能已经第二次进入栅障，它们中的第一个将把标记置 0。现在，这个标记位只有当所有进程都第二次进入栅障后才能置 1，但这不可能发生，因为 P_x 没办法离开第一次栅障的反复循环。

如何解决这个问题？我们要做的是防止一个进程在所有其他进程都离开先前的栅障操作之前再次进入这个栅障。一个办法是用另一个计数器来记录离开了栅障的进程，在这个计数器未达到 p 之前不让进程在新的栅障操作中对标记复位。然而，对这个计数器的操作会引起进一步的时延和争用。另一方面，在目前的框架下面，我们没办法等所有进程都到达栅障后再将标记置 0，因为我们是将它置 1 来释放进程的。一个较好的解决方案是避免显示重置标记的值，而让进程等待标记在后面的栅障事例中获得一个不同的释放值。例如，进程可能在一次事例中等待标记变成 1，在下一次事例中等待变成 0。可以用一个私有变量来跟踪在本次事例中要等待的值。根据栅障的语义，一个进程不能超越其他进程多出一个栅障，于是我们只需要两个值（0，1）在其间交换。因此我们称这种方法为*感应逆转*（sense reversal）。现在，在前面的例子中，当第一个进程到达栅障时，标记不需要复位；相反，停在老栅障上的进程仍然等待标记达到老的释放值，而进入新事例的进程等待另外的释放值。当所有进程到达了新栅障事例，标记的值只改变一次，因此在老事例上的进程看到之前将不会改变。下面是一段对应的代码：

```
BARRIER (bar_name, p)
{
   local_sense = !(local_sense);        /*toggle private sense variable*/
```

```
    LOCK(bar_name.lock);
    mycount = bar_name.counter++;        /*mycount is a private variable*/
    if (bar_name.counter == p) {         /*last to arrive*/
      UNLOCK(bar_name.lock);
      bar_name.counter = 0;              /*reset counter for next barrier*/
      bar_name.flag = local_sense;       /*release waiting processes*/
    }
    else {
      UNLOCK(bar_name.lock);
      while (bar_name.flag != local_sense) {}; /*busy-wait for
                                                 release*/
    }
}
```

注意在计数器增量后锁不是立刻被释放的，锁的释放只是在条件被求值后发生；这里的原因在习题5.18中可以发现。我们现在有了一个正确的栅障了，可以连续重用任何次数。 355
剩下的问题就是我们下面要考虑的性能。(注意，保护计数器增量的 LOCK/UNLOCK 可以被简单的 LL-SC 或其他原子增量操作替代，效率更高。)

3. 性能

我们所追求的栅障性能指标类似于锁，包括：

- 低延迟。(关键路径的长度要小) p 个处理器通过栅障所需的相关操作的链和总线事务应该尽量小。
- 低流量。由于栅障是全局操作，很可能许多处理器会试图在同一时间执行一个栅障算法。这个算法应该减少总线事务的总数（无论是否在关键路径上），从而减少可能的争用。
- 可扩展性。时延和流量应该只随处理器数缓慢增加。
- 低存储代价。我们当然希望存储代价低。
- 公平性。我们应该避免同一个处理器总是最后一个离开栅障的情形（或者我们可能要保持 FIFO 序)。

在前面所描述的集中式栅障中，每个处理器访问一次锁，因此关键路径的长度至少和 p 成比例。下面考虑总线流量。为了完成它的操作，一个涉及 p 个处理器的集中式栅障要做 $2p$ 次总线事务来获得锁和对计数器加一，两次总线事务让最后一个处理器对计数器复位，写入释放标记，另外还需 $p-1$ 次总线事务来在标记作废后再读它。注意这要好于 p 个处理器要获取一把 test-and-test&set 锁的情况；因为在后面一种情况下，每次释放（共有 p 次）都引起作废，结果是 $O(p)$ 进程要在此又一次完成 test&set 操作，于是就导致 $O(p^2)$ 总线事务。然而，如果许多处理器同时到达栅障，源于这些竞争性总线事务的竞争可能是很大的，因此，栅障的代价可能很高。

4. 栅障算法针对总线的改进

集中式栅障的部分问题是所有处理器争用相同的锁和标记变量。为对付这样的问题，我们可以构造一种栅障，只引起少数处理器争用相同的变量。例如，处理器能通过一种软件结合树来指示它们到达了栅障（见3.3.2节)。例如在一个二元结合树中，只有两个处理器在树的节点上相互通报它们的到来。这样，只有两个处理器会访问一个给定变量。在有着多重并行通路的分布式网络中，诸如那些可扩展机器中所有的，一个结合树能比集中式栅障性能 356

好得多。这是由于两对不同的处理器能在网络的不同部分并行通信。然而，对于像总线那样的集中式互连来说，即使处理器对通过不同的变量通信，它们都要产生总线事务，于是就要串行化，且在这条总线上争用。由于含有 p 叶节点的二元树大约有 $2p$ 个节点，一个结合树所需要的总线事务数和集中式栅障类似。结合树方式还有较高的时延，这是因为除了它也要求 $O(p)$ 串行化总线事务外，即使没有总线串行化，每个处理器还至少等待 $\log p$ 步才能从叶节点到达树的根，每一步都有实质性的工作。结合树在总线上的优点是它不用锁，而只是简单的读写操作，如果总线上的处理器数目很大，这可能对它的较大的非竞争时延是个补偿。然而，如图 5-31 所示，简单集中式栅障在总线上表现很好。图 5-31 中示出的其他一些栅障将在第 7 章讨论可扩展机器时和树栅障一并讨论。

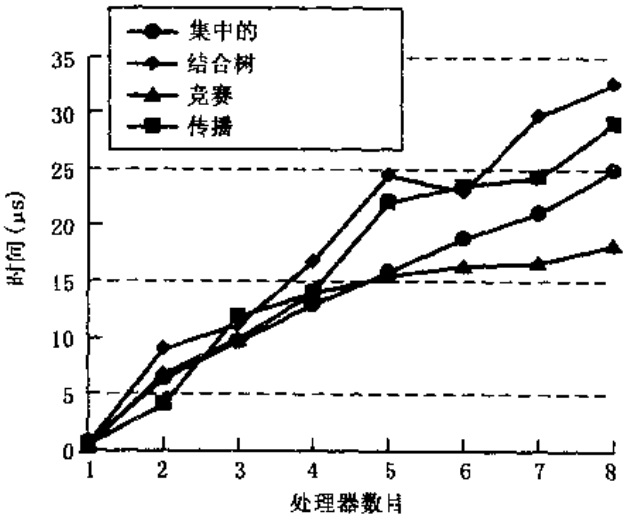


图 5-31　SGI Challenge 上一些栅障的性能。这里性能的度量是在一个循环中连续执行许多栅障，每次通过栅障的平均时间（在栅障之间没有工作或延迟）。结合树栅障在其关键路径上有较高的时延，这使得在总线上它的性能不好，因为在这种情况下它的流量和争用优势体现不出来

5．硬件原语

由于集中式栅障用锁和普通的读和写，所需的硬件原语取决于所用的锁算法。如果一个机器不能很好支持原语，结合树栅障也可以在基于总线的机器上用得很好。

357

一个特别的总线原语可以用来减少在集中式栅障中读扑空引起的总线事务的数量（也对处理器踏步等待同一变量的高度争用的锁有用）。这个优化利用了如下事实，当一个变量在释放作废后，所有处理器发出对它的读扑空。不同于所有处理器发出分别的读扑空总线事务，一个处理器能够管理总线，在看到对同一单元的读扑空（由另一处理器发出，首先到达总线）的响应后，将自己的读扑空在到达总线前取消，然后简单地从总线上取得返回数据。在最好的情况下，这种做法能将读扑空总线事务从 p 降到 1。

6．硬件栅障

如果提供一个单独的同步总线，如讨论锁时提到的，它也可以用来支持栅障。这会将流量和冲突从主要总线上分离出来，从而得到较高性能的栅障。从概念上讲，一条“线与”线路就足够了。当到达栅障时，处理器将它对这条线的输入置 1，然后等到输出为 1 后再向前推进（在实际中，重用栅障要求的线不只一根）。这样一种单独的栅障硬件机制在栅障频率很高的情况下特别有用，这种情况可能出现在由并行化编译生成的程序的内层循环里，在每次内层循环完成后都需要一次全局同步。不过，它的价值在实际中尚不清楚，而且在机器中的处理器只有部分参与栅障时难于管理。例如，不容易动态地改变参与栅障的处理器的数量，在操作系统使得进程在处理器之间迁移时也不容易改动参与处理器的配置。如果多个参

与进程运行在同一个处理器上也会造成麻烦。因此，当前总线型多处理器不倾向于提供特别的硬件支持，而是用锁和共享变量来构造软件栅障。

5.5.6　同步问题小结

有些基于总线的机器对诸如锁和栅障等同步操作提供直接的硬件支持。然而，关于灵活性的考虑导致大多数现代设计人员只在硬件上提供简单的原子操作，通过用它们来构成软件库，来得到高层同步操作。应用程序人员通常用这些库，可以不需要了解那些由机器支持的底层原子操作。这些原子操作可能是通过单条指令实现的，或者通过仔细推敲的读写指令对来实现的，如同装载-加锁和条件存储。后者有较大的灵活性，这使得它们日益流行。我们已经看到了一些同步原语，算法和系统结构细节之间的相互作用。在后面几章中，当我们讨论可扩展共享地址空间机器时，这种相互作用会进一步体现出来。 358

5.6　对软件的影响

迄今为止，我们一直关注着基于总线的、具有高速缓存一致性的多处理器的高层体系结构的问题，以及工作负载的特性对体系结构和协议折中的影响。现在，我们转回来探究这些小规模机器的体系结构是如何影响并行软件的。具体来说，我们不再是在固定工作负载的条件下研究提高机器性能或协议的方法，而是在给定机器配置情况下去探究怎样提高并行程序的性能。改进同步算法，以减少通信量和时延就是其中一个方面，下面先让我们更一般地看看并行程序设计的过程。

关于负载平衡和固有通信的一般性技术在第 3 章中已讨论过了，这些技术也适用于一致性高速缓存的机器中。除此以外，在这类机器有一种一般性划分原则，即在对计算任务的分配时，力图使得只有一个处理器对一组数据做写操作（至少在单个计算阶段中）。在很多计算问题中，处理器要读某一共享数据结构，但写另一个数据结构，因此这种原则可以应用于很多计算领域。例如，在 Raytrace 应用中，处理器读的是场景，写的是图像。常常可以有两种选择，一是划分计算任务以至于不同处理器写的是不相交的数据结构，但从共享数据结构中读；或是从不相交的数据结构中读，但写到同一块共享存储区。在所有其他因素相同的情况下（诸如负载均衡，程序设计的复杂度），我们认为通常应该避免写共享。写共享不仅能引起作废，从而导致缓存扑空和流量增加，而且如果不同的处理器写同一字，很可能这样的写操作必须被诸如锁一样的同步机制保护起来，所带来的开销于是更大。

通信的结构比较单纯：有一个集中式存储器，没必要用显式的存储器到存储器的数据传送，因而所有的通信都是通过 load 和 store 指令隐式完成的，这样的指令会导致缓存块的传送。映射不是一个问题（除了尽量使进程在不同处理器间少做迁移），它由操作系统全权处理。人们最关心的问题是在性能协调步骤中利用数据局部性和控制附加的通信，特别是利用时间和空间局部性来减少高速缓存扑空，从而减少时延、通信量和在共享总线上的争用。

由于主存是集中式的，时间局部性体现在处理器的高速缓存中。第 3 章介绍了基于总线机器的工作集曲线，其特性如图 5-32 所示。所有和容量有关的扑空都要出现在同一个总线上，从而要反映到存储器中，其代价和一致性扑空一样高。即使有无限大的高速缓存，另外三类扑空也会产生总线流量。开发时间局部性的主要目的就是让工作集能放到高速缓存的层次结构中，所用的技术和第 3 章中讨论过的相同。 359

对空间局部性来说，一个集中式的存储器使数据在主存中的分布和存储分配是无关紧要

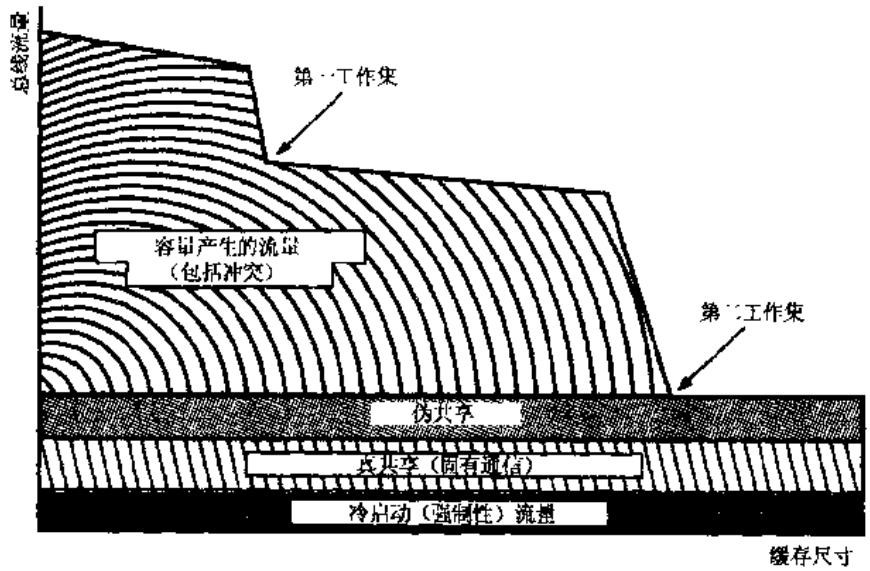


图 5-32　在共享总线上的数据流量及其各种成分随缓存大小的变化。拐点指出程序的工作集

的（仅仅在存储器不同模块之间安排数据的交叉存取以减少争用可能是应该考虑的，这一点和单处理器情形一样）。空间局部性不好的效果是存储碎片（也就是将不必要的数据取到了缓存块中）和伪共享。这里的原因是通信和一致性的粒度都是缓存块，要比一个字大。前者引起存储碎片，后者引起伪共享。（这里我们假定没有用到消除伪共享的一些技术，例如设置子块的脏位，因为它们在大多数实际机器中都没采用。）我们考察一些减轻这些问题的方法和有效地利用大缓存块预取的结果，以及通过更好的数据的空间组织去减轻缓存冲突的一些方法。在程序员的“技巧锦囊”中能发现很多这样的方法。下面所列出的是一些最通行的。

- *任务的分配应减少访问模式的空间交叉。*任务的分配应该是让每个处理器去访问一片相邻的数据区。例如，把一个 n 个元素数组的计算分配给 p 个处理器处理，最好的分配是让每个处理器访问 n/p 个相邻的元素，而不是对元素进行精细地交叉分配。这样提高了数据的空间局部性，减少了缓存块的伪共享。当然负载均衡或其他一些约束可能迫使我们不能这样做。
- *组织数据以减少访问模式的空间交叉。*在第 3 章中我们有一个在方程求解器内核中的例子，那里我们用高维的数组使一个数组在一个处理器中的分配在地址空间上是连续的，于是在物理上分布的存储器中，本地分配能按页的粒度来进行。这种方法也可以帮助减少伪共享，数据传送的存储碎片和冲突扑空，如图 5-33 和图 5-34 所示，从而减少扑空和总线上的流量。一个比网格元素大的缓存块可能跨越面向列的划分的边界，如图 5-33a 所示。如果一缓存块比两个网格元素还大，它就可能引起由伪共享导致的通信。如果我们暂且假定在算法中不存在固有的数据通信，这一点是很容易看到的；例如，假设在每一次遍历中一个进程只是简单地把一常量加到该进程所分配到的网格元素上，而不是执行一次相邻元素的计算。现在，即使缓存块有两个网格元素那么大（或更大），跨越一个划分的边界，由于不同的处理器向其中不同的字做写操作，也将会出现伪共享。这也将导致在通信中的存储碎片，因为一个读其边界元素但扑空了的进程会去取在同一缓存块、但在其他处理器划分中的其他元素，但那些数据是它不需要的。图 5-34 解释了冲突扑空的问题。在以上所有

360

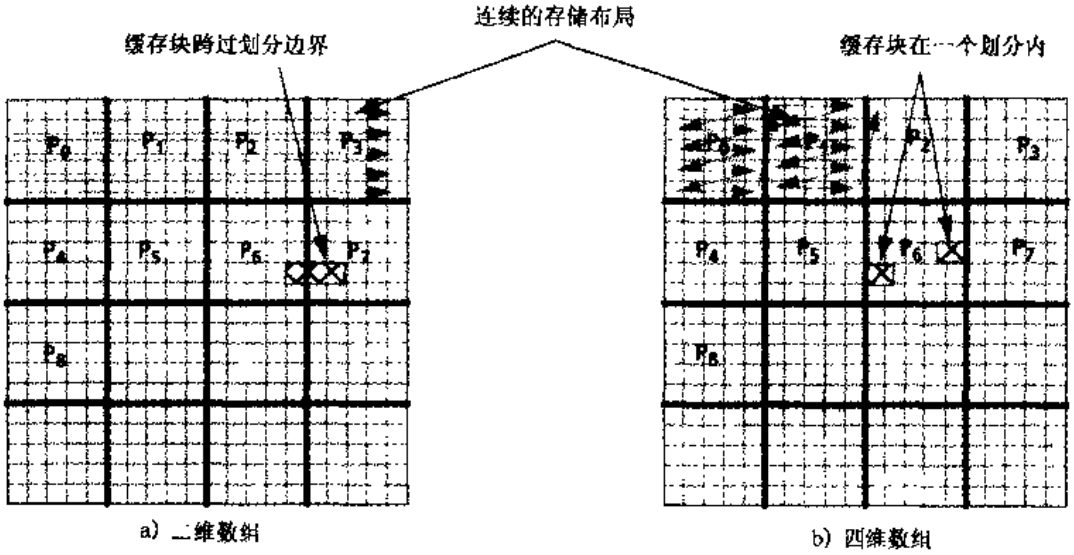


图 5-33 利用高维数组保持划分在地址空间中的连续，以减少伪共享和存储碎片。在二维数组的情形中，跨划分边界的缓存块存储碎片（扑空导致带入其他处理器划分中的无用的数据）和伪共享。四维数组表示使得划分连续，从而缓解了这些问题

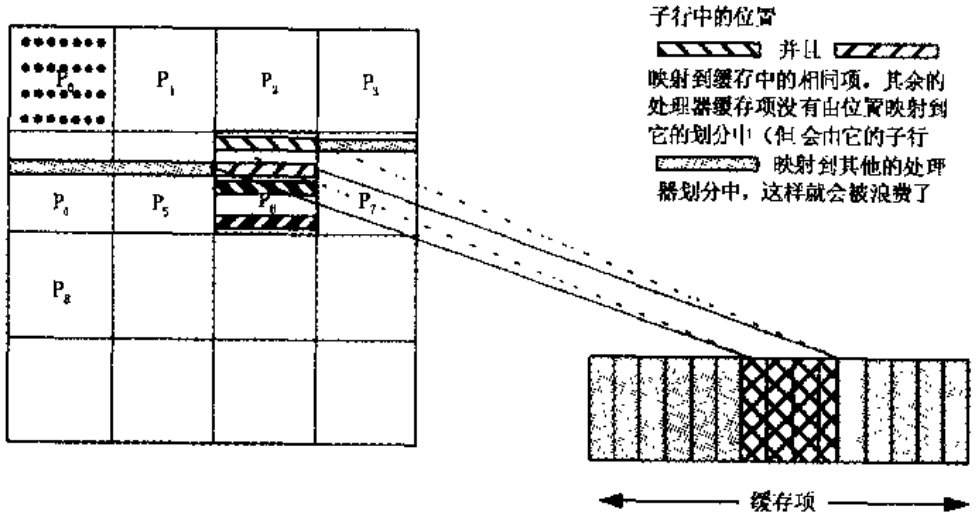


图 5-34 由二维数组表示在一个直接映射缓存中引起的高速缓存映射冲突。这个图表示最坏的情况，在处理器划分中相继两个子行的间隔（即二维数组中一行的大小）正好等于缓存的大小，于是相继的两个子行映射到缓存中的相同的位置。对每一子行的访问将替换出前面的子行。处理器在它的划分上做下一次遍历时，它会在它引用的每个缓存块上扑空，即便缓存大得能装下整个划分也是如此。受网格大小、处理器数和缓存大小的影响，我们可能遇到许多不如最坏情况差的中间状况。由于缓存大小是 2 的幂次，将待分配的数组的维展定为 2 的幂次是不利的

情况下的问题都是划分的不连续性。于是，一个简单的数据结构变换（如图 5-33b 所示）就帮助我们在方程求解器内核中解决了所有与空间局部性相关的问题。图 5-35 表示的是在 SGI Challenge 上，针对 Ocean 和 LU 应用用高维数组表示网格或分块矩阵对性能的影响。冲突和伪共享对于单处理器和多处理器性能影响的差别在此是很明显的。

- 防止冲突扑空。针对发生在网格求解器的冲突扑空，图 5-34 表明了由于高速缓存的

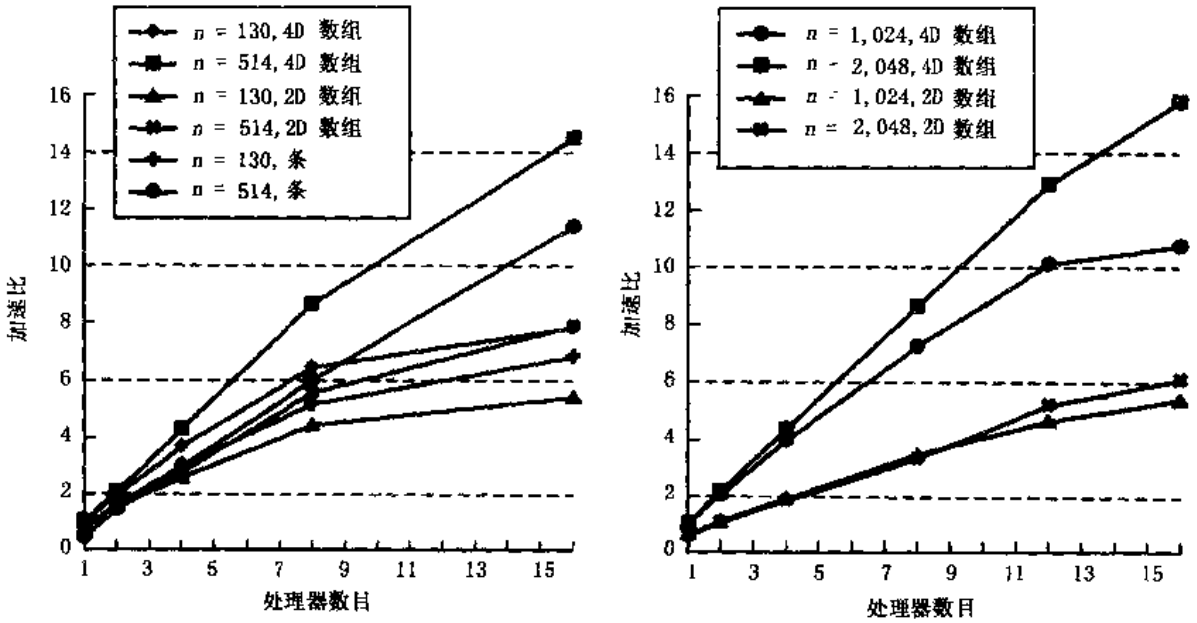


图 5-35　在 SGI Challenge 上用四维数组和二维数组表示矩阵数据对性能的影响。所显示的数据针对 Ocean 和 LU 的不同的问题规模。对于 Ocean，"条状" 表示按连续行的条状划分（其中二维或四维数组无关紧要），其他情形表示类方块状划分

大小通常是 2 的整次幂，当数组的存储分配规模也是 2 的幂时高速缓存冲突的严重情况。于是，在应用中即使数组的逻辑大小是 2 的整次幂，我们也常常分配一个更大的数组，但只访问其中有效的部分。不过这种策略会受到物理上分布存储分配页面粒度（也是 2 的整次幂）的影响，所以我们必须小心。在此例中的高速缓存映射冲突发生在一个以可预取模式访问的单一数据结构中，因而能在一定数据结构下得到缓解。当映射冲突发生在不同的主数据结构之间时，映射冲突是很难避免的（例如，通过由 Ocean 应用程序使用的不同网格），这种情况可能只利用随意填充和对准来缓解。然而，在一个共享地址空间上，当它们发生在一些看似无害的共享变量和数据结构上时（程序员通常不太注意这些变量和数据结构），常常是特别隐伏的。例如，在一个直接映射的高速缓存中，一个频繁被访问的指向重要数据结构的指针可能会与一个在同一计算中也被频繁访问的标量变量发生缓存冲突，导致很大的通信量。幸运的是，在现代两级高速缓存中（容量大并且是组相联）这些问题出现很少。总之，如果不把注意力放在减少冲突扑空上，那么在开发局部性上的努力就会白费了。

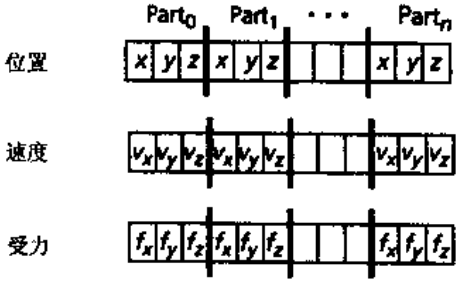
- 每个处理器都用单独的堆。这是值得考虑的一种方法，每个处理器（或进程）都有自己独立的堆用来动态地进行数据分配。否则，如果一个程序要访问很多非常小的存储区，被不同处理器使用的数据就可能落在同一缓存块中。
- 通过数据拷贝来提高空间局部性。如果一个处理器要反复用到一组本来不是连续分配的数据，我们可以考虑做适当的数据拷贝，使得它们在相应的计算周期中是连续分配的，从而提高数据的空间局部性，减少缓存冲突。这里要注意的是，拷贝数据需要访问主存因而有一定的开销；同时，如果那些数据可能都驻留在高速缓存中，这种方法就无效了。例如，在分块矩阵的因子分解或乘法的计算中，用二维数组表示矩阵，

361 ↓ 363

矩阵中的每一分块在地址空间中是不连续的（就像在方程求解器内核中的一个划分）。然而，二维表示能使程序设计更简单。因此一种常见的做法是用二维数组，但在相关的计算阶段将分给其他处理器的数据拷贝到自己的一片连续的临时数据区中，来减少冲突扑空。当然，拷贝的代价需要和冲突减少的利益权衡。在有关粒子的应用中，当一个粒子从一个处理器的划分移动到另一处理器时，将和粒子相关的数据移动到一个连续、紧凑的数据区，可以提高空间局部性。

- *填充数组。*并行程序设计者经常创建一些以进程标识号为索引的数组。例如，为了跟踪负载平衡的情况，可能要维持一个 p 元整型数组，数组的每一项记录了相应处理器已经完成的任务数。由于缓存的一个块可能容纳这个数组中的大部分元素，而这些元素经常要被不同的处理器更新，所以伪共享就是一个严重的问题。一个解决的办法是，用一些无用元素填充在数组的有用项之间，使得有用元素之间的数组段和缓存的块一样大（预见到程序可能在不同的机器上运行，也可以填充得更大一些），然后让数组和一个缓存块对齐。但是填充许多大数组将导致存储的大量浪费，而且导致数据传输中的存储碎片。一个更好的策略是把分给一个进程的这种性质的变量集中到一个记录中，将该记录填充至一缓存块，然后创建一个以这样的记录为元素、以进程标识符为索引的数组。
- *决定怎样组织记录数组。*假设我们要表示一些逻辑记录，如在 Barnes-Hut 引力模拟中的粒子。我们应该是用一个数组来表示粒子集，数组有 n 项，每一项为一个粒子的记录，包含粒子的位置、速度、受力、质量等域，如图 5-36a 所示的那样呢？还是把它们表示成若干 n 元数组，每个数组对应一个域，如图 5-36b 所示的那样？为 CRAY 那样传统的向量机写的程序常按一个对象的性质或域分类，每一类用一个数组（向量）表示——实际上，甚至对域中的每个物理维（x、y 或 z）都分别用一个数组（向量）表示。当按域访问数据时，如访问所有粒子的速度，由于访问跨距是存储单元，减少了存储体冲突，因而提高了向量操作的性能。但是，在缓存一致性的多处理器中，需要考虑一些新的折中，最好的数据组织方法取决于访问模式。

a) 按粒子组织



b) 按属性或域的粒子组织

图 5-36　基于记录数据的不同组织方式

在 Barnes-Hut 应用中，粒子状态的更新和受力计算阶段揭示了一个引人注目的冲突现象。先考虑更新阶段。在这一阶段处理器读写被分配的所有粒子的位置域和速度域，然而被分配的粒子在同一粒子数组中是不连续的。假定每个域或性质对应一个 n（粒子数）元数组。一个双精度三维的位置（或速度）数据占 24 个字节，故多个这样的数据能对应一个高速缓存块。由于在数组中邻近的粒子可能被不同的处理器读和写，因而将引起伪共享。在这个阶段，用一个记录数组，数组中的每一项记录了粒子的所有信息，这样组织数据比按域组织数据要好。

现在我们考虑受力计算阶段。假定我们是用一个记录数组来组织数据，每个粒子的所有数据在其中的一个记录中。为了计算一个粒子的受力，一个处理器要读其他许多粒子的位置数据，然后更新被处理粒子的受力值。然而，一个粒子的受力值和位置值可能存在于同一缓存块中。在更新受力域时，可能会同时作废该粒子在其他处理器的缓存中的位置值（这个粒子的记录在其他处理器缓存中的存在是由伪共享所导致的，其他处理器要用到该粒子的位置数据，而位置数据在这个计算阶段是不被修改的）。在这种情况下，如果我们把以粒子记录为单元的数组分解为两个大小都为 n 的数组，一个以位置域（或其他性质域）为单元，一个以受力域为单元。可以如前所述填充受力数组，减少跨粒子的伪共享。总之，把一个记录数组分解为两个，一个数组的每项记录是在一个计算阶段中只读的记录域，另一个数组的每项记录是在同一阶段中要更新的记录域。不同的情况或计算阶段可能规定不同的数据组织，但最终的结果要根据对性能起支配作用的模式和计算阶段。

- 数组的对准。结合前述的技术，将数组对准到缓存块边界能得到更多的好处。例如，一个缓存块的大小为 64 字节，一个记录域为 8 字节，我们用一个记录数组存放粒子数据，数组的每一项是一个记录，该记录有 x、y、z、fx、fy、fz 6 个域。为了避免跨粒子的伪共享，我们为每一个记录填充两个 8 字节的虚记录域，使每个记录正好用一缓存块存放。但如果数组在虚拟地址空间中页面偏移 32 字节的地方起始，即使利用填充法也会导致伪共享，因为每个粒子的数据将跨存在两个缓存块中。即使 malloc 调用不能返回和页或缓存块对齐的数据，数据的对准也是容易办到的，只要在调用 malloc 函数时多要求一点额外存储，然后调整数组的起始地址即可。

364
↓
365

通过以上的讨论，数据的组织、对准和填充等技术对于开发空间局部性，减少伪共享和冲突扑空都是很重要的。有经验的程序员，甚至一些编译器都使用这些方法。如第 3 章所讨论的，我们知道局部性和附加通信的问题比固有数据通信对性能更重要。这些问题可能使我们为了某个应用而重新考虑算法的划分决策（回顾在第 3 章 3.1.2 节中讨论的方程求解器问题，对比条状和块状两种划分，或参看图 5-35a）。

5.7　结论

对称共享存储的多处理器是工作站和个人电脑的自然扩展。一个串行应用程序可以照常在其上运行，同时享受到更大的处理器时间片、更大的共享主存和在这种机器中典型具备的 I/O 能力所带来的好处。运行并行应用也相对是容易的，因为对所有共享数据处理器可以通过普通的存取指令直接访问。在这样的机器上也容易从事渐进的并行化工作，即可以选择串行应用中的计算密集部分首先并行化，其效果当然要服从 Amdahl 定律。对多道程序的工作

负载，它优势的关键是对资源共享的细粒度，按这种粒度，系统的资源能在不同应用进程之间共享，并且由不同的操作系统共享，因此能很容易地为每个应用表现出一种熟悉的、单一系统映像。在时间上，处理器和/或主存页面时常被重新分配给不同的应用进程；在空间上，主存可以以页面为单位分配给不同的应用进程。因为这些吸引人的特点，所有大型的计算机系统厂商，从工作站供应商 Sun、Silicon Graphics、Hewlett-Packard、Digital、IBM 到个人电脑供应商 Intel，Compaq 都在生产和销售这样的机器。事实上，对于一些大的工作站供应商，这样的多处理器机器在它们的销售额和纯利中都占相当一部分，因为在这些高端机器上有更高的利润。

设计对称多处理器的关键技术难题是共享存储系统的组织和实现。这样的存储系统除了要满足常规的存储访问外，还要用来在处理器之间的通信。目前大多数小规模并行机用系统总线作为通信的互连，故问题转变为在处理器的私有缓存中保持共享数据的一致性。系统设 366
计者有很多有效的选择，其中包括与缓存块相关联的状态集，所用到的总线事务处理和动作，缓存块大小的选择，使用更新还是作废策略。但设计者关键的任务是依据预期的数据共享模式，在工作负载的高效执行和使任务实现更加容易之间做出选择。另一个难题是高效的同步技术的设计和实现，要求即有高性能也有灵活性。

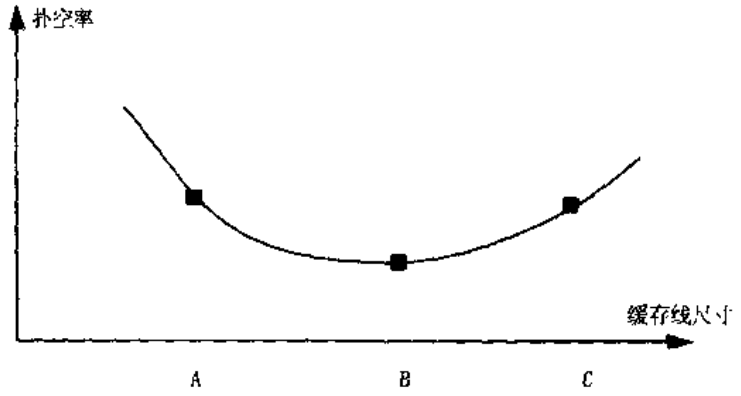
由于处理器、存储器、集成电路和封装技术的持续高速发展，人们关心小规模多处理器的未来和各种相关的设计问题。我们依据以下 3 个理由预料小规模多处理器将继续成为重要的发展方向。首先，它们提供了一个很吸引人的性能价格比。个人或小团体都能承受它们，把它们作为一个共享资源或一个计算或文件服务器。其次，当今微处理器的设计为多处理器作好了准备，在设计人员开始设计下一代多处理器的时候，他们是知道微处理器的未来趋势的，因此在新发布的微处理器和用其构成多处理器之间就不会有太大的延迟。如同在第 1 章所见，Intel Pentium Pro 微处理器就可以直接插在共享总线上。其三，一些必要软件支持正迅速成熟起来。例如，大多数计算机系统厂商都有支持它们自己基于总线的多处理器的并行操作系统版本。随着集成度的提高，在一块芯片上集成多个处理器将更加吸引人。尽管最佳的设计点会随技术和工艺的发展变化，但在本章中讨论的设计问题是基本的，无论技术如何发展它们将会保持其重要性。

本章在逻辑层研究了基于总线多处理器的设计的一些关键方面，包括缓存块状态的转换和完整的（原子的）总线事务。在这一层上，设计和实现表现在对传统缓存控制器的扩展。然而，很多在设计上的困难和很多优化与改进的机会出现在下一层协议设计和更具体的“物理”层中。下一章将讨论深入一层的设计，讨论基于总线缓存一致性多处理器的设计和体系结构，包括一些自然的推广。

习题

5.1 处理器寄存器是否也有和缓存一致性类似的问题？假定在硬件上不能保证寄存器的同一性，目前的系统怎样保证程序所期望的语义？

5.2 下图表示了在一个多处理器中一个应用程序的扑空率与缓存块大小的函数曲线。与预计的一样，曲线是 U 状。考虑在曲线上的 A、B、C 三点。指出在什么环境下，其中一 367
点可能是机器的敏感操作点（也就是说，在这一点比其他两点机器有更好的性能）。对于单处理器，曲线形状和位置有怎样的变化？



5.3　假设一个基于总线共享存储器的处理器的平均数据存储通信量：私有读——70%，私有写——20%，共享读——8%，共享写——2%。同时 50% 的指令（32 位）是存和取。有一个 32 KB 的数据/指令缓存，私有数据的命中率是 97%，共享数据是 95%，指令是 98.5%。缓存线为 16 字节。

我们希望在 64 条数据线和 32 条地址线的总线上放置尽可能多的处理器。一个处理器的时钟是总线时钟的两倍，不考虑存储器的影响，处理器的 CPI 是 2.0。如果我们采用 a）写分配策略的直写缓存，总线最大能支持多少处理器？b）如果是回写呢？忽略缓存一致性流量和总线争用。在回写缓存中因扑空取一新块来替换一脏块的概率是 0.3。对于读，存储器在收到地址后两个周期给出数据。对于写，地址和数据同时发给存储器。假定总线是原子的，处理器扑空损失正好等于每个扑空所需的总线周期数。

5.4　下面列出了三个存储器访问流，比较执行它们的开销，运行在基于总线机器上 a）Illinois MESI 协议，b）Dragon 协议。根据访问流的特点和相关协议解释执行的不同。

stream 1: r1 w1 r1 w1 r2 w2 r2 w2 r3 w3 r3 w3
stream 2: r1 r2 r3 w1 w2 w3 r1 r2 r3 w3 w1
stream 3: r1 r2 r3 r3 w1 w1 w1 w1 w2 w3

368　所有访问都是针对同一个单元的：*r*/*w* 指明读或写，后面的数字表示发出操作的处理器。假定所有缓存初始为空，用下面的代价模型：读/写缓存命中要花费 1 个周期，扑空带来在总线上的事务（BusUpgr，BusUpd）要花费 60 个周期，扑空带来的传送整个缓存块要花费 90 个周期。假定所有缓存都是写分配的。

5.5　1）随着扑空时延的增加，更新协议和作废协议哪一个更可取？为什么？

2）在一个多级缓存层次结构中，你是将更新传播到第一级缓存还是仅仅只到第二级？说明折中方案。

3）为什么在目前基于更新的一致性在多处理器计算服务上对典型的多道程序工作负载不是一个好方法？

4）为了提供一个更新协议作为选择，一些机器在页中为软件提供了协议类型控制；也就是说，给定的页能保持一致地使用更新方式或作废方式。另一种不是基于页的控制方法是为将导致更新而不是作废的写提供特殊的操作码。评价优劣。

5.6　下面给出了一些代码段，在顺序同一性下哪些结果是可能的（或不可能的）。假定在代码到达前，所有变量初始化为 0。

1）

P_1	P_2	P_3
A = 1	u = A	v = B
	B = 1	w = A

2）

P_1	P_2	P_3	P_4
A = 1	u = A	B = 1	w = B
	v = B		x = A

3）在下面的序列中，用虚线框起的操作是同一指令的一部分：fetch&increment。然后假定它们是独立的指令。针对这两种情况分别回答上面的问题。

P_1	P_2
u = A	v = A
A = u + 1	A = v + 1

369

5.7　1）在 5.2.2 节中提到过一种由于写缓冲区的使用所导致的重新定序问题。在单处理器中并发程序环境下它会不会是一个问题？如果是，你如何去预防它？如果不是，为什么？

2）按照程序执行序，对于同一个存储单元，同一个处理器发出的读能否在前面的写之前完成（例如，如果写被放置在写缓冲区中但对其他处理器尚不可见）但仍然提供一个一致的存储系统？如果能，读返回值是什么？如果不能，为什么？这样做能否仍然保证顺序同一性？

3）如果我们只关心一致性，不关心顺序同一性（SC），我们能否说处理器通过了写操作写就完成了？

5.8　顺序同一性（SC）的充分条件是必要的吗？将那些约束放松：1）尽可能少约束，2）采用一种适当的中间方式。评价在实现复杂度上的效果。

5.9　考虑下列 SC 的充分条件：

- 每个进程按程序执行序发出存储请求。
- 一个读或写操作发出后，发出操作的进程将在发出下一操作前等待当前操作的完成。
- 在一个处理器 P_j 能够返回一个被另外处理器 P_i 写的值前，所有关于 P_i 的在其发出写操作之前完成的操作也必须关于 P_j 完成。

这些条件能否真正保证 SC 执行？如果能，为什么；如果不能，构造一个反例，说明为什么在本章中列出的条件是充分的。[提示：想想这些条件与本章中的那些条件有何不同]

5.10　考虑一个四个处理器基于总线的多处理器使用 Illinois MESI 协议。每个处理器执行一个 test&set 锁去获得访问一个空临界区。假定 test&set 指令总是发到总线上并且同处理一般读过程的时间开销一样。初始时处理器 1 拥有锁，处理器 2、3、4 在自己的缓存中踏步等待锁被释放。每个处理器一旦得到锁就退出程序。只考虑总线对上锁和解锁操作的处理：

1）从初态到终态所执行的最少总线事务是多少？

2）最多总线事务数是多少？

3）假设用 Dragon 协议，1）和 2）又如何？

5.11 在锁中指数式回退的主要优缺点是什么？考虑 test&set 锁，test-and-test&set 锁，票号锁，基于数组的锁。如果 LL-SC 被用来代替原子指令，情况会如何变化？

370

5.12 假设所有 16 个处理器同时争用一个 test-and-test&set 锁（每个处理器只有一次）。假定在 0 时刻所有处理器在自己的缓存中的锁上踏步等待并且因为一个释放而被作废。

1）如果所有临界区为空（也就是说，每个处理器在 LOCK 和 UNLOCK 之间什么都不做），到所有处理器获得锁将有多少次总线事务？

2）假设总线是公平的（即总是在服务新请求之前先服务挂起的请求），每个总线事务的开销是 50 个周期，到第一个处理器能获得和释放锁要多长时间？到最后一个处理器获得和释放要多长时间？

3）在一个不公平的总线上，如果想让你希望的处理器获得优先权而不管请求的顺序，最多能做到什么程度？

4）能否有一个不同于公平总线的总线仲裁方案用来提高性能？

5）如果用来实现锁的变量没有被缓存起来，一个 test-and-test&set 锁仍比一个 test&set 锁产生的通信量少吗？为什么？

5.13 对于同习题 5.12 的 2）一样配置的机器，假设是公平总线，使用一个加号锁，第一个和最后一个处理器需要多少个总线事务和多长时间来获得和释放锁？如果使用基于数组的锁，情况又如何？

5.14 考虑图 5-29 中使用指数型回退的 test&set 设置锁的性能曲线，为什么针对非空临界区的曲线比针对空临界区的曲线差？

5.15 1）在我们锁的实验中，为什么我们安排解锁后的延迟 d 后要比临界区延迟 c 小？如果 d 比 c 大将产生怎样的问题？[提示：画出两个处理器的执行时序图。]

2）如果我们用大得多的数值 c 和 d，比较锁算法会有怎样的结果？

5.16 1）分别用 i）fetch&increment 和 ii）LL-SC 来写出实现票号锁和基于数组锁的伪代码（高层表示加上汇编）。

2）假定你不用 fetch&increment 原语而用 fetch&store（一种简单的原子交换）。用这个原语能实现基于数组的锁吗？描述你得到的锁算法。

5.17 用 LL-SC 实现一个 compare&swap 操作。

5.18 考虑在 5.5.5 节中描述的具有感应交替功能的栅障算法，如果把 UNLOCK 语句放在计数器增值语句后，而不是在 if 条件的每个分支后，有问题吗？问题是什么？

371

5.19 假设有一个机器在每个机器字中都提供了满-空位硬件支持[○]。这类机器允许下列 C 函数：

`ST_Special (loc, val)` 把 `val` 写到数据单元 `loc` 并且置满空位。如果满空位已经被设置，则发出自陷信号。

`int LD_Special (loc)` 等待直到 `loc` 的满空位被设置，读数据，清满空位，返回结果数据。

○ 可以想像满-空位是一个特别的二进制位置满空位意指让其表示“满”，清满-空位意指让其表示“空”。——译者注

利用上述原语写一个 C 函数 swap（i，j），它能够原子性地交换 A［i］和 A［j］的内容。你应允许很高的并发性（如果多个处理器要交换一对不相干的存储单元，它们应能并发进行）并且必须要避免死锁。

5.20 原子操作 fetch&increment 能被用于实现栅障、信号灯和其他同步机制。fetch-and-add 的语义是将第二个参数加到第一个参数所指的存储单元并且返回该存储单元在加之前的值。用 fetch-and-add 原语去实现栅障操作适合于共享存储的多处理器。为了使用栅障，一个处理器必须执行 BARRIER（BAR，N），这里 BAR 是栅障的名字，N 是需要到达栅障的进程数。假定在 BAR 的使用中 N 都是同一数值。栅障将能够支持下面的代码：

```
while (condition) {
  Compute for a while
  BARRIER(BAR, N);
    }
```

下面是实现这种栅障的一种方法：

```
BARRIER(Var B: BarVariable, N: integer)
{
  if (fetch-and-add(B, 1) = N-1) then
    B := 0;
  else
    while (B != 0) do {};
}
```

这段代码中的问题是什么？写出避免了这些问题的 BARRIER 函数。

5.21 考虑下面同步原语 BARRIER 的实现，它用在应用中每个计算阶段的尾部。假设 bar.releasing 和 bar.count 的初始值为 0，bar.lock 初始时是开锁状态。

```
struct bar_struct {
  LOCKDEC(lock);
  int count, releasing;
} bar;
...
BARRIER(N)
{
  LOCK(bar.lock);
  bar.count++;

  if (bar.count == N) {
    bar.releasing = 1;
    bar.count--;
  } else {
    UNLOCK(bar.lock);
     while (! bar.releasing)
       ;
    LOCK(bar.lock);
    bar.count--;
    if (bar.count == 0) {
    bar.releasing = 0;
    }
```

372

```
        }
        UNLOCK(bar.lock);
    }
```

1）本代码不能提供一个正确的栅障。描述这个实现中的问题。

2）尽量少改动本例中的代码，使其能提供一个正确的栅障实现。指明改动的地方或仔细描述改动。

5.22 考虑具有移动性的数据：在处理器之间来回窜的共享数据，每个处理器读，然后在其他处理器读之前写。在标准 MESI 协议下，读扑空和写都能导致总线事务。

1）根据表 5-1 中列出的数据，估计用升级（BusUpgr）代替 BusRdX 后能省下的最大带宽。

2）有可能增强缓存块和状态转换图的状态，使紧接在同一块写操作后的读操作能被承认，使具有移动性的块能在第一次读扑空时直接把独享状态引进缓存中（而不是共享状态）。试给出附加状态的建议和状态转换表的扩充。根据表 5-1、5-2、5-3 中的数据，计算能得到的带宽节省。除了带宽节省以外还有其他好处吗？指出在
373 程序中哪些地方具有移动性的协议可能有害于性能。

5.23 通过在更新时适当地更新主存，Firefly 更新协议删除了在 Dragon 协议中现有的 Sm 状态。我们能否通过合并状态 E 和 M，进一步减少 Dragon 和 Firefly 协议中的状态？有哪些折中的考虑？

5.24 人们发现处理器有时只写一个字到一缓存块中。为了优化这种情况，改变在所有情况下都用回写缓存的方式，就提出一个协议，它有下面的一些特点：1）在初始写一块时，处理器写直达到总线并且该块以一种新的状态——保留状态放置在缓存中；2）在写一个保留状态块时，缓存块转到已修改状态，用回写代替直写。

1）画出这种协议的状态变化，状态为 INVALID、SHARED、RESERVED、MODIFIED。图中要能显示对每一状态 BusRd、BusWr、ProcWr 和 ProcRd 的情况。指明处理器在斜杠（如 BusWr/WriteBlock）后的动作。由于采用整字和整块写，要指明刷新字（FlushWord）和刷新块（FlushBlock）。

2）如何区分这种协议与 4 状态的 Illinois 协议？

3）简明地描述你为什么认为这种协议没有用在像 SGI Challenge 一样的系统上。

5.25 考虑一个处理器写一个被很多处理器共享的块的情况（因而使它们的缓存作废）。如果这个缓存块在随后被其他处理器反复读，每次都将在该块上扑空。研究者提出了一个广播读的策略，一个处理器读该块时，所有其他的被作废的处理器将该块读到它们的二级高速缓存中。你认为这是一个好的协议扩展吗？至少给出两条理由说明你的选择，而且至少有一条是从反面说明。

5.26 在下面三个处理器的访问流中按图 5-20 中的类别把扑空分类（按照表 5-4 的格式）。假设每个处理器的缓存仅仅由一个 4 字的缓存块构成，字 w0 到 w3 在一个缓存块中，字 w4 到 w7 在另一缓存块中。

操作数	P_1	P_2	P_3
1	st w0		st w7
2	ld w6	ld w2	
3		ld w7	

（续）

操作数	P_1	P_2	P_3
4	ld w2	ld w0	
5		st w2	
6	ld w2		
7	st w2	ld w5	ld w5
8	st w5		
9		ld w3	ld w7
10		ld w6	ld w2
11		ld w2	st w7
12	ld w7		
13	ld w2		
14		ld w5	
15			ld w2

374

5.27 给你一个基于总线共享存储的机器。假设处理器有一个32字节的缓存块，A 是一个数组，其中的元素是4字节长的整数。考虑下面的循环：

```
for i ← 0 to 16
  for j ← 0 to 255 {
    A[j] ← do_something(A[j]);
```

1）在什么情况下最好使用动态调度的循环？

2）在什么情况下最好使用静态调度的循环？

3）对于动态调度的内循环，每一次一个处理器要迭代多少次？

5.28 如果你正在写一个图像处理程序，用一个二维像素数组表示图像。在计算中的基本迭代如下：

```
for i = 1 to 1024
  for j = 1 to 1024
    newA[i,j] = (A[i,j-1]+A[i-1,j]+A[i,j+1]+A[i+1,j])/4;
```

假设A是一个以4字节单精度浮点数为元素的数组，并且按行存储（即A[i, j]和A[i, j+1]在存储中的地址是相邻的），A的起始地址是0。你正写的这个代码是支持32个处理器的。每个处理器有一个32 KB的直接映射缓存，缓存块的大小为64字节。

1）首先试着按交错分配法给每个处理器分配数组中的32行。你期望的计算和总线通信量的比率是多少（固有的或附加的）？假设每一次循环是4个计算单位，忽略所有其他的控制和赋值操作，声明你用到的其他假设。

2）如果分配给每个处理器的数组行是相邻的，回答和1）同样的问题？ 375

3）如果分配连续的数组列给每个处理器，回答和1）同样的问题？

4）如果A的起始地址为32，而不是0，用3）中的分配法，计算和所产生的通信量的比率有变化吗？如果有，是变大还是变小，为什么？如果没有，为什么？

5.29 下面是用一个$O(N^2)$算法的简化的n体代码（即计算分子之间的相互作用），估计在稳定状态每一时间步的扑空数。再用本章中讨论的提高空间局部性和减少伪共享的方法改写代码。试着按你期望的处理器数和缓存块大小重构。假设16个处理器，1 MB直接映射缓存，缓存块为64个字节。估算重构后代码的扑空数。声明你做的所有

假设。

```
typedef struct moltype {
  double x_pos, y_pos, z_pos;   /*position components*/
  double x_vel, y_vel, z_vel;   /*velocity components*/
  double x_f, y_f, z_f;         /*force components*/
  } molecule;

#define numMols 4096
#define numProcs 16
molecule mol[numMols]

main()
{
... declarations ...
  for (time=0; time < endTime; time++)
    for (i=myPID; i < numMols; i+=numProcs)
    {
      for (j=0; j < numMols; j++)
      {
        x_f[i] += x_fn(position of mols i & j);
        y_f[i] += y_fn(position of mols i & j);
        z_f[i] += z_fn(position of mols i & j);
      }
      barrier(numProcs);
      for (i=myPID; i < numMols; i += numProcs)
      {
        write velocity and position components
        of mol[i] based on force on mol[i];
      }
      barrier(numProcs);
    }
}
```

376