第6章　基于侦听的多处理器的设计

我们看到，市场上对称多处理器在性能、价格和规模上有很大的差异，这些差异主要不是由于高速缓存一致性协议的不同选择上，而是在于支持协议逻辑操作的组织结构的设计和实现上。人们对协议诸方面的权衡有了很好的理解；大多机器使用的都是上一章所描述协议的某种变形。然而，协议所导致的时延和带宽依赖于总线设计、高速缓存的设计以及和存储器系统的整合，还取决于系统工程的开销。本章考察基于侦听的一致性高速缓存的对称多处理器的详细物理设计问题。

虽然从第5章我们看到一致性协议对应的抽象状态转换图是相当的简单，但是在实现层会有一些很微妙的问题要解决。实现都必须争取达到至少三个相关的目标：正确性、高性能和最少的额外硬件。正确性问题产生的原因是抽象层认为原子性的活动到硬件层并不一定是原子性的。性能问题产生的主要原因是使存取操作流水化，允许同一时间有多个操作待完成（使用存储器的不同部件），而不是必须等待上一操作完成才能开始下一操作。不幸的是，正是由于这些事件关系错综复杂，使得正确性往往难以得到保证。因为一致性硬件中难以察觉的差错，一些商用系统，包括含有片上一致性控制器的微处理器的发布时间都大大推迟了。总体来说，缓存一致性多处理器中通信辅助部件（控制器）的设计提出了一系列挑战，其形式和复杂性和现代处理器的设计不相上下，表现在大量的待完成指令和乱序执行。我们需要深入一层来考察基于侦听多处理器的设计，理解状态转换图中所表达的实际需求。

本章首先列举缓存一致性存储系统主要的正确性要求。在6.2节，我们给出一个含有单级缓存和单事务原子总线的基础设计，简述在处理单个总线事务时的关键事件。在正确性讨论中这一节假定的是作废协议，但有关要点也可以直接应用于更新协议中。6.3节将这个设计扩展到多级高速缓存，展示了协议事件是如何在层次之间进行传播的。6.4节将这个基础设计扩展到使用事务拆分型总线的情形。在这样的总线中，一个总线事务被分为请求和响应两个阶段，这两个阶段都要涉及总线仲裁，因此多个事务在总线上可以同时处于待完成状态，能够以流水方式处理。然后，我们讨论多级高速缓存和事务拆分型结合的情形。从这一设计点来看，支持源于多个处理器的多个待完成扑空只是迈出了一小步，这是因为所有事务已经能够流水化处理，一些事务能够并发发生。在此当中潜在的本质挑战是要能保证由一致性和存储器同一性模型所要求的操作序的体现。随着设计复杂性的增加，如何做到这一点，也是在这些章节中讨论的。

一旦理解了一般情况下的关键设计要点，就能着手研究具体设计的细节。6.5节给出了两个案例：SGI Challenge 和 Sun Enterprise，用微基准测试程序和我们自己的例子应用程序来阐述它们的性能。最后，6.6节考察了若干将上述技术在功能和规模上扩展所涉及的进一步问题。

6.1　正确性需求

高速缓存一致性的存储系统理所应当要满足一致性的要求，并且要保持由存储器同一性

模型表达的语义。尤其是对一致性来说，必须要能够找到过时的副本，并且在写操作发生时使其作废或更新，还要提供写操作的串行化。如果要保持顺序同一性，那么应当提供写原子性和检测写操作完成的能力。另外，设计应该满足任何协议实现都要具有的性质，即要避免死锁和活锁，消除挨饿或使挨饿发生的可能性极小。最后，设计还要应付控制之外的错误情况（如奇偶校验错），并且尽量从错误中恢复。

死锁发生在操作仍然待完成但所有系统活动均已停止时。多个并发的实体争相获得共享资源，并以不可剥夺方式占有，产生资源依赖环，这样就导致潜在的死锁产生。图 6-1 所示交叉路口的交通是一简单类比。在此交通例子中，实体是车辆，资源是道路。每辆车需要两个道资源才能通过交叉路口，缺一不可，但每辆车都拥有一个道的资源，彼此互不相让。

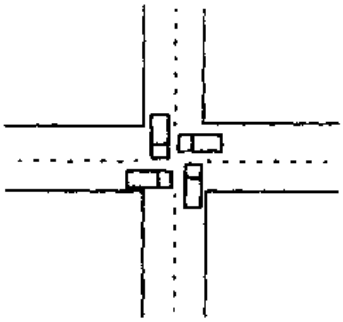


图 6-1　在十字路口的死锁。四辆车到达一个十字路口，每辆车占一条道。由于每辆车都占有着另一辆车得以前进所需的资源，它们相互阻塞。即使每辆车都让其右边的那一辆，该十字路口还是死锁。为消除这个死锁，某些车必须后退，让其他车前进，从而它自己才能前进

在计算机系统中，典型的实体是控制器，资源是缓冲区。例如，图 6-2a 所示，有两个控制器 A 和 B 通过缓冲区通信。A 的输入缓冲区已满，而且 A 拒绝接受任何来的请求信号，除非 B 接受了来自 A 的一个请求（从而释放 A 的缓冲区，A 才能接受来自其他控制器的请求信号）。但是 B 的输入缓冲区也满了，除非 A 能接受来自 B 的一个请求，否则拒绝任何来的请求信号。两个控制器都不能接受到请求，因此死锁形成。图 6-2b 所示的三个处理器例子可用于说明一般的多于两个控制器的情形。为防止死锁，根本上是要避免这种依赖环或者当其产生时将它消除。

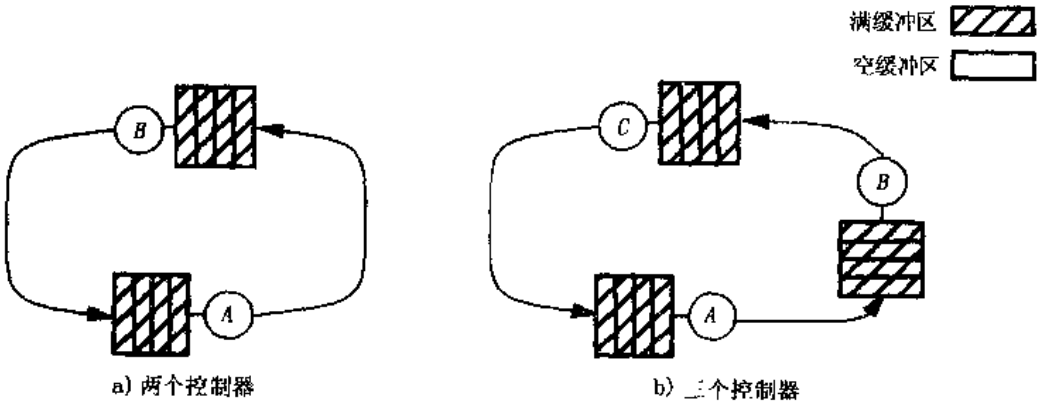


图 6-2　在计算机系统中的死锁。如果相互独立的控制器通过有限缓冲区相互通信的话，死锁很容易发生。如果在通信图上可能形成环路，那么每个控制器可能都会停滞，等待它前面的控制器释放资源

若尽管计算过程仍在系统中进行，但任何处理器的计算却不能得到进一步的推进，则称该系统处于活锁中。继续考虑上面的交通例子，任一车辆可能原路返回，以清空交叉路口，然后再试图前进。然而，如果车辆都同时重复前进后退，那将有大量的活动但最终不断重复

停止在同一位置，而无任何实质性的进展。在计算机系统中，活锁典型发生于多个独立控制器竞争某一共享资源时，当任何一方未完成当前操作对该资源的使用时，另一方就把资源剥夺过去。

378 ~ 379

挨饿现象不会停止全部的进展，但这是一种极度不公的现象，即一个或多个处理器毫无进展而其他处理器不断获得机会。例如，在交通例子中，活锁问题可以通过简易的优先级安排来解决。如果向北去的车辆比向东去的车辆优先级高，后者必须后退让前者在后者重试之前通过交叉路口。同样，南向的车辆可以比西向的车辆优先级高。[○] 不幸的是，这样却不能解决挨饿现象：在繁忙拥挤的交通中，东向车辆可能因为总有新的北向车辆准备通过而总不能通过路口。北向的车辆不断前进而东向的车辆处于挨饿状态。可能的解决办法是设置一仲裁者（警察或交通灯）公平的安排资源使用。这个比喻很容易扩展到计算机系统中。

一般而言，挨饿的可能性被认为比死锁和活锁危害要小。挨饿不会导致整个系统停止前进而且也不是永久状态。就是说过去一段时间内处于挨饿的处理器并不意味着将来所有时间内都会挨饿（在某些时候，北向的车将会清空了，东向的车辆就可以通过路口）。事实上，和这种无监控的交通例子相比，挨饿现象更不可能产生于计算机系统中，因为挨饿是和时间相关的，而必要的时序条件通常不会持久。在基于总线的系统中，挨饿现象易被消除，只要使用合理的总线仲裁设备和使用先进先出队列方式访问硬件资源就行了。然而，在后面的章节讨论可扩展系统的时候，会看到完全消除挨饿现象会大大增加协议的复杂度，降低常见情况下事务的推进速度。因此，虽然几乎所有的系统都在尽力减少挨饿现象发生的可能，但是许多系统并没有完全消除挨饿。

6.2 基础设计：采用原子总线的单级高速缓存

在第5章中，我们讨论了高速缓存一致性协议如何保证写串行化、如何满足顺序同一性的充分条件。我们假设总线是原子性的，给定进程的操作相对于其他进程也是原子的，而且产生总线事务的存储器操作，从发出到完成，也是原子的，即使来自不同的处理器。本节中，我们的假设会稍微实际些。每个处理器仍只有单级高速缓存，总线事务是原子不可分的；高速缓存在执行一存储操作所包含的一系列步骤时，它可以使处理器停滞；从而一个进程内的操作相对都是原子不可分的。除上述外，再没有其他假设了。本节讨论在这样的系统中实现侦听和状态转换所引起的基本问题和种种权衡，以及在提供写串行化、检测写完成和保证写原子性所引起的新问题。随后，我们将讨论更大胆的系统设计，包括先前讨论过的更复杂的高速缓存层次结构和更复杂的总线。不过，我们的讨论都是基于回写高速缓存的，至少最接近总线的缓存是如此，这样可以减少总线的流量。

380

即使是对简单的单级高速缓存和原子总线的例子，也必须做出一些设计上的选择。首先，给定处理器和总线端的侦听部件都需要访问高速缓存中的标记，我们应该如何设计标记和控制器呢？其次，从高速缓存控制器得到的侦听的结果需要作为总线事务的一部分体现出来，这又如何和何时办到？第三，即使总线有原子不可分性，满足处理器的存储器操作的若干动作还要使用其他资源（如高速缓存控制器），而这些活动并非原子不可分，从而引入可能的竞争条件。在这种非原子性条件下，我们如何来设计高速缓存控制器的协议状态机

○ 这里在方向上的描述和图6-1不符。如果以图6-1为准，应该是东向比北向的优先级高、西向比南向的优先级高。——译者注

呢？这样对写串行化、检测写完成和写原子性会产生什么新的问题呢？关于死锁、活锁和挨饿又会产生什么新问题呢？最后，从高速缓存中回写也能引入有意思的竞争条件，我们必须有支持原子不可分的读-改-写操作的机制。下面逐个来考虑这些问题。

6.2.1　高速缓存控制器和标记的设计

首先考虑传统单处理器的高速缓存，它由包含数据块、标记和状态位的存储阵列、比较器、控制器和总线接口部件组成。当处理器执行对高速缓存的操作时，地址的一部分用来访问可能含有相应存储块的一高速缓存组。标记和其他的地址位进行比较，确定寻址的存储块是否确实存在。然后进行适当的数据操作和修改状态位。例如，在一个干净的高速缓存块上的写命中，会导致一个字被更新并且将状态设置为已修改状态。高速缓存控制器按顺序进行它的存储阵列的读和写。如果操作需要将存储块从高速缓存传送到存储器（或者反之），则高速缓存控制器要发起一个总线操作。该总线操作要求总线接口部件执行一系列步骤，典型步骤如下：1）发出总线请求信号；2）等待总线的认可；3）驱动地址和命令；4）等待命令被相关设备接收；5）传送数据。高速缓存控制器所采取的一系列动作用有限状态机实现，一个总线事务中的若干步骤的实现也是如此。注意，不要将这里的状态机和缓存存储块所遵循的协议中的状态转移图相混淆。

为支持侦听一致性协议，必须对基本的单处理器高速缓存控制器的设计有所扩充。首先，高速缓存控制器除了响应处理器的操作外，还必须能监控总线上的操作。最简单方法是把高速缓存看作有两个控制器：总线端控制器和处理器端控制器，分别控制来自相应端的外部事件。任何情况下，当操作发生时，控制器都要访问高速缓存的标记。每个总线事务中，总线端控制器必定要捕获来自总线端的地址，并使用该地址进行标记对比。如果对比失败（一次侦听扑空），则不采取任何行动：即该总线操作和本高速缓存无关。如果侦听“命中”，
381 则控制器可能会因为高速缓存一致性的要求卷入到该总线事务中。这可能涉及到对状态位的读-改-写操作或者是将一个存储块放到总线上（或者两者兼有）。

如果高速缓存只有一组标记，则很难允许这两个控制器对标记的同时访问。于是，在总线过程中，处理器将被锁在外面，不能访问高速缓存，这将降低处理器的性能。如果给处理器较高的优先级，于是侦听控制器必须在获得了标记访问权后才能进行总线过程，从而导致有效总线带宽的降低。为缓解这一问题，一致性高速缓存的设计对标记和状态可以使用双端口的随机存取存储器或者对每个块的标记和状态进行拷贝。高速缓存的数据部分不被复制，因为对它的访问不那么经常。如果采用双标记的方式，两套标记的内容是完全相同的，一套用于处理器端控制器查询，另一套用于总线端控制器侦听（如图 6-3 所示）。两个控制器可以同时读取标记和进行检验。当然，一旦某一存储块的标记或状态更改（如当状态改变或写入新的存储块时），最终两个版本都要修改，因此控制器之一可能被锁住一段时间。机器设计师们可以考虑用一些技巧来缩短控制器被锁住的时间。例如，上述例子中处理器端的标记不是在总线端标记更新时立即更新的，而是仅在后来高速缓存中数据被修改时更新。标记更新的频率也远远小于标记查询的频率，从而可以认为总线端标记更新和处理器高速缓存访问冲突较小。

另一个对单处理器中高速缓存的主要扩充是现在控制器不仅可以是总线事务的发起者，还可以作为总线事务的应答者：传统应答设备，如存储器模块控制器，它要监控总线，看一

个过程是否和某个特定的地址子集相关，且可能在若干“等待”周期后对相关读写操作做出应答。传统应答设备甚至可以把数据放到总线上。高速缓存控制器的做法与之类似，只是高速缓存控制器不对固定地址子集应答，而是对任何事务都要监控总线进行标记对比来决定是否相关。对基于更新的协议来说，控制器也可能侦听总线传出的新数据。多数现代微处理器已经实现此种增强型高速缓存控制器，因而用它们能很方便地构成多处理器。

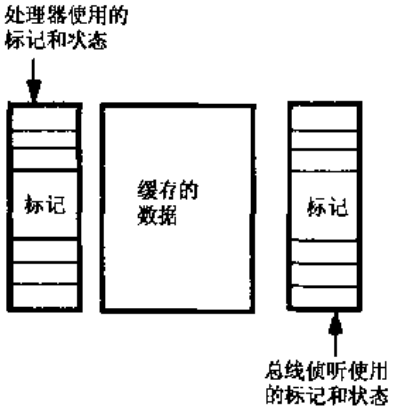


图 6-3 单级侦听缓存的组织。对于单级高速缓存，用双拷贝标记和状态组来减少冲突。处理器用一组，总线侦听器用另一组。不过，对缓存内容或状态的任何改变都要同时更新两组标记

6.2.2 侦听结果的报告

侦听也给总线事务引入了新的内容。在传统单处理器系统的总线事务中，某一设备（发起者）将地址放到总线上，所有其他的设备监控地址，其中有一设备（应答者）发现地址和它相关。然后数据在两设备之间传输。应答者发出“线或”信号来确认自己的作用；如果在一定时间后没有设备应答，则产生总线错误。对侦听高速缓存来说，每个高速缓存要用它的标记和地址对比，从各高速缓存侦听的总结果必须在总线事务继续之前让总线得知。特别地，这种侦听结果的一个功能就是通知主存是否要响应请求信号，或者某个高速缓存是否持有该存储块的已修改版本，从而必须采用另一行动。这里的问题是，什么时候将侦听结果报告到总线上，且以何种形式？

首先让我们着眼于“何时”这个问题。显而易见，令人满意的就是使延迟时间尽可能的少，使得主存能迅速决定如何行动[⊖]。下面是三个主要选项：

1）保证在一定的时间内产生侦听结果，通常指当地址出现在总线上后某个固定的时钟周期内。一般而言，这需要双套标记，因为处理器（一般具有优先级）在总线事务发生时会多次访问标记。即使有了双套标记，由于当处理器更新标记时两套标记都不能被访问，我们在完成侦听所需的时延上还需要采取一种保守的态度；例如在 MESI 协议由执行（E）向修改(M)状态转移时就是如此[⊜]。采取这种做法的优点在于主存设计不受影响，而且高速缓存

⊖ 注意，在原子型总线上，我们有办法让系统对侦听时延不那么敏感。由于在任何时间只能有一个总线事务（原著中误为“存储事务”。——译者注）待完成，主存能够开始访问存储块，不管最后是它还是缓存要提供数据；否则主存子系统就可能空闲。然而，我们后面要讨论到，减小这个延迟对事务拆分型总线十分重要。在那种情况下，多个事务可能并发地在总线上待完成，于是存储子系统可能正在服务于另一个请求，为它（而不是高速缓存）提供数据。

⊜ 有趣的是，在我们所描述的基本三态作废协议中，如果没有相应的总线事务的卷入，缓存块的状态是不会更新的。这通常会给标记的更新留有足够的时间。

到高速缓存之间的联系非常简单。而缺点是需要额外的硬件和可能较长的侦听时延。一种由4个 Pentium Pro 构成的多处理器就是用的这种方案，它能在必要时延长或推迟侦听阶段（见第8章），HP公司的商业服务器（Chan et al. 1993）和 Sun Enterprise 也是如此。

382 ~ 383

2）设计能够支持可变延迟的侦听方案。主存首先假设的是某一个高速缓存将提供数据，但提供的时机是在所有其他高速缓存控制器都完成侦听并且指出不提供数据以后。一定的握手协议是需要的，但高速缓存控制器不用担心标记访问冲突会影响及时的查询，且设计者也不用在侦听结果延迟时间的估计上采用保守的态度。SGI Challenge 多处理器使用的方法是这种做法的一种变形，即存储子系统先是针对请求取出数据，然后停滞，直到侦听完成后再根据情况相应动作（Galles and Williams 1993）。

3）第三种可能的做法是让主存子系统对每一存储块维护一个标记位，来指示该块是否被某一高速缓存修改。用这个方法，主存子系统不会被迫依靠侦听结果来决定下一步行动。这种方法的缺点是对主存子系统增加了额外的复杂性。

侦听的结果以何种格式报告到总线上呢？对 MESI 设计来说，请求的高速缓存控制器需要知道被请求的存储块是否在其他处理器的高速缓存中，这样才能决定以独享（E）还是共享（S）状态来装入该存储块。另外，存储系统也要知道是否有一高速缓存将该块置为修改状态；如果有，存储器则不用应答。一种合理的办法是使用三组线或信号，两个用来报告侦听结果的情况，另一个用作指示侦听结果是否有效。第一个信号指示某个处理器的高速缓存中（除请求处理器外）含有该存储块的副本。第二个信号指出该存储块在某个处理器的高速缓存中处于修改状态。我们不需要知道是哪个高速缓存，因为高速缓存本身知道该采取什么行动。第三个信号是禁止信号，它的出现表示还有高速缓存没有完成侦听。当第三个信号未出现时，请求者和存储器能够安全地检验其他两个信号。MESI 协议的完整的 Illinois 版本要更加复杂些，这是因为即使在共享状态下，存储块更优先地从其他高速缓存中取出而不是取自存储器。如果多个高速缓存均有副本，优先级机制决定从哪个高速缓存中提取数据。这也是为何大多数使用 MESI 协议的商业机型限制了高速缓存到高速缓存的传输。Silicon Graphics Challenge 和 Sun Enterprise 在数据在某一高速缓存中处于修改状态时，才使用高速缓存到高速缓存的传输，此时只有一个提供数据者。Challenge 在高速缓存到高速缓存传输中更新存储器，而 Enterprise 不更新存储器，并且使用第5章中讨论过的 MOESI 协议所含的第五个状态，即拥有状态。

6.2.3 对回写的处理

回写的实现比较复杂，因为回写过程涉及到一个要进入缓存的块和一个要被替换出的缓存块（已修改），因此包括两个总线事务。一般来说，为了使处理器在引起回写的缓存扑空发生后能尽可能快的继续工作，我们愿意推迟回写的处理，而先来处理引起回写的扑空。这一做法给我们提出了两个要求。其一，需要机器提供额外的存储空间一个（回写缓冲区）当新存储块写入高速缓存而总线还未能响应第二个事务时，被覆盖的存储块能暂时保存在回写缓冲区中。其二，在完成回写前，也许能发生总线事务，包含正在回写的存储块的地址。在这种情况下，控制器一定要从回写缓冲区中提取数据，且取消刚才未完成的回写总线请求。这就需要有地址比较器来监视该回写缓冲区。由第8章可知，在物理上分布存储的机器中，回写的正确实现有更进一步的问题。

384

6.2.4 基础系统组织

图 6-4 所示的是我们得到的基础侦听结构的框图。每个处理器有单级回写式高速缓存。高速缓存是双标记的，从而总线端控制器和处理器端控制器可并行的进行标记对比。处理器端控制器通过将地址和命令放到总线上来启动一个事务。在回写事务中，数据从回写缓冲区传输。在读事务中，数据被捕获在数据缓冲区内。总线端控制器侦听回写标记和高速缓存标记。总线仲裁器以一种全序安排总线上运行的请求信号。对于每个总线事务来说，请求阶段中的地址和命令以这种全序来驱动侦听查找。线或侦听结果用于向发起者确认所有的高速缓存都已看到请求并采取了相关行动。

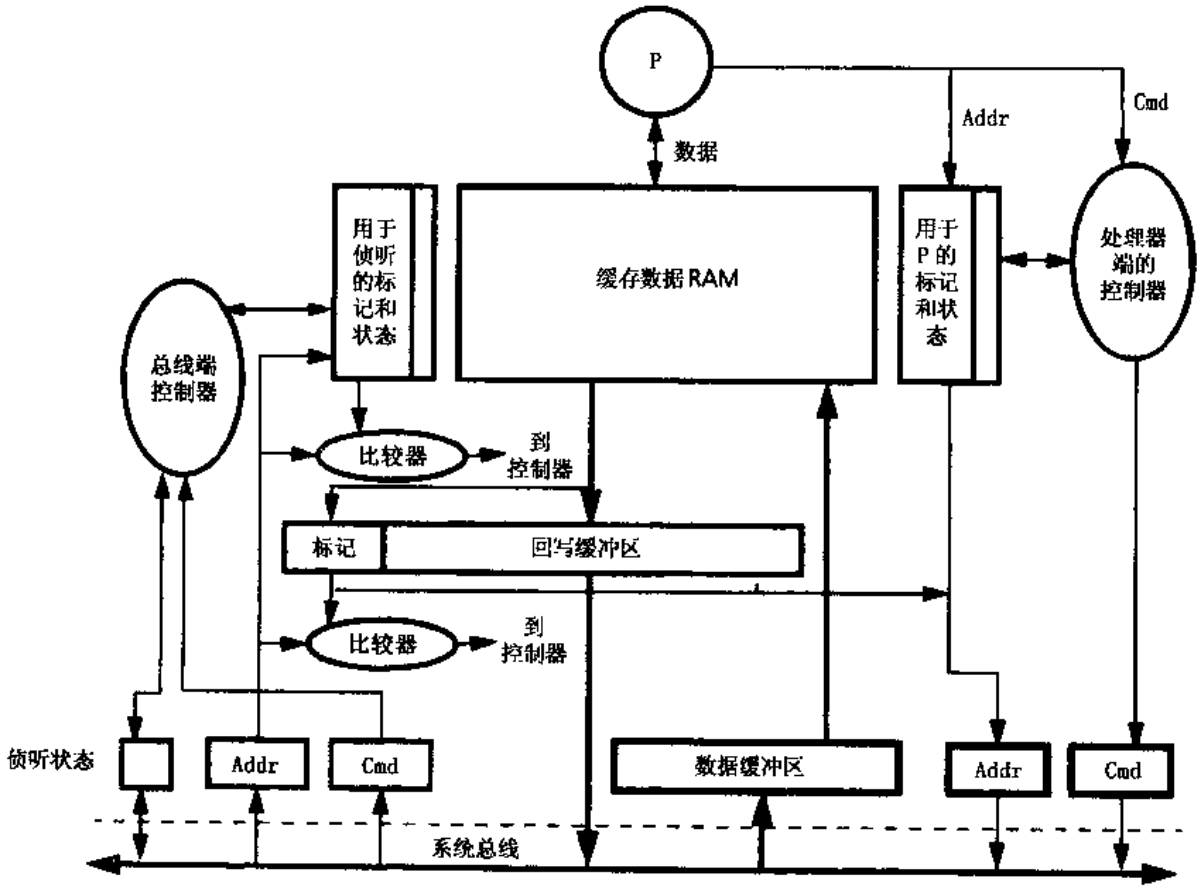


图 6-4 基础机器的侦听高速缓存的设计。假设每个处理器有一个单级回写高速缓存，用一种作废协议，处理器只能有一个待完成的存储请求，系统总线是原子的。为保持简单，我们没有画出总线仲裁逻辑和所需的某些低级信号和缓冲区。也没有画出在总线方控制器和处理器方控制器之间所需的协调信号

用这种简单设计，让我们来考虑进一步的正确性问题，这些问题需要对状态机和协议进行扩充，或者需要在实现中有些特别的措施。它们包括非原子性的状态转移、为达到一致性和同一性的序列化、死锁、活锁和挨饿现象。

6.2.5 非原子性的状态转移

在第 5 章的状态转移图中，状态转移和相关活动都被假设为同时发生或至少是原子不可分的。事实上，由处理器产生的请求要花一段时间才能完成，常常包括一个总线事务。虽然

在我们的简单系统中，一个总线事务是原子不可分的，但这仅仅是要满足处理器请求一系列活动的其中之一。这一系列活动包括查询高速缓存的标记、总线仲裁、由其他控制器对相应高速缓存采取行动以及由请求处理器的控制器在总线事务结束时采取的行动（可能包括实际地将数据写入存储块中）。总的来说，这一系列活动不是原子不可分的，即使是使用原子总线，来自不同处理器的多个请求可能同时在系统的各部分活动，或许当处理器（控制器）P有一请求等待完成（如等待获得总线访问权）而来自另一处理器的请求出现在总线上，并且需要由P提供服务，甚至可能是和P请求待完成一样的存储块。这类复杂问题将在例6.1中加以说明。

例6.1　假定处理器P_1和P_2在它们的缓存中共享存储块A，并且同时对A发出一个写操作。说明当P_2的事务出现在总线上时，P_1的待完成请求是如何等待总线的，解决这种复杂情况可能有些什么办法。

解答：这里是一种可能的情形。P_1的写操作会检查它的高速缓存，确定在实际向块中写数据之前，它需要将存储块的状态从共享提升到已修改，并且发出一个升级总线请求。同时，P_2已经发出了一个类似的升级或者排他读A的过程，它可能首先通过仲裁得到总线。

385
~
386

P_1的控制器会看到总线事务，必须将A的状态从共享降级到无效。否则，当P_2的事务完成后，A在P_2的高速缓存中为已修改状态，在P_1的高速缓存中为共享状态，这是违反协议规定的。但现在使P_1待完成的升级总线请求不再是合适的，必须用排他读请求替换。这样，控制器也必须要能够将它自己待完成的请求的地址和从总线上侦听得到的地址对比，并且在必要的时候修改前者。（如果在协议中没有升级过程，并且即使在对共享状态的存储块做写操作时也用排他读，那么在这种情况下即便存储块的状态改变了，请求也没必要改变。因此，当评价协议优化的复杂性时，我们应该考虑这些实现方面的需求。）■

要处理状态转移中非原子性现象，而且有时需要根据所观察到的事件来修改请求和行动，一种方便的方法就是使用中间或过渡状态扩展协议状态图（原协议状态我们已深入讨论过，如MESI，其状态都认为是稳态）。例如，可用单独的一个状态来指示升级请求正待完成。图6-5给出了MESI协议的一种状态扩展图。为应答处理器的写操作，高速缓存控制器通过确认总线请求和转移到中间的S→M状态来开始总线仲裁。当总线仲裁器确认了给该设备的总线授予（BusGrant）信号，则从中间状态转移出去。此时，总线升级事务（BusUpgr）放到总线上，高速缓存中相应块的状态得到更新。然而，当处于S→M状态下，如果总线上观察到对该存储块的总线读请求（BusRdx）或者总线升级请求（BusUpgr），那么控制器将视该存储块在此过程和转移I→M状态之前已被设置为无效。（我们可以撤回总线请求，转移到I状态，让挂起的PrWr再度得到处理。）在处理器从无效状态读数据时，控制器转到中间状态（I→S，E）；而下一个要转移到的稳态由读操作得到总线后共享信号线的值决定。这些中间状态通常在高速缓存块的状态位中未加以编码，所编码的仍然是稳定的MESI状态。这是因为要表示可能出现处于暂时状态的高速缓存块需在每个高速缓存块中扩充，实在过于浪费。这些状态可以通过状态位及控制器的状态共同反映出。可是当我们考虑高速缓存允许多个待完成过程时，就必须对可能处于过渡状态的高速缓存（多个）块加以明确表示。

协议中的状态数的扩充加大了证明实现过程正确性和测试设计的难度，因此设计者也寻求避免过渡状态的机制。例如Sun Enterprise没有使用MESI协议中的BusUpgr事务，而是使用BusRdx事务中侦听结果来消除冗余的数据传输。回顾在总线写操作中，包含该存储块的

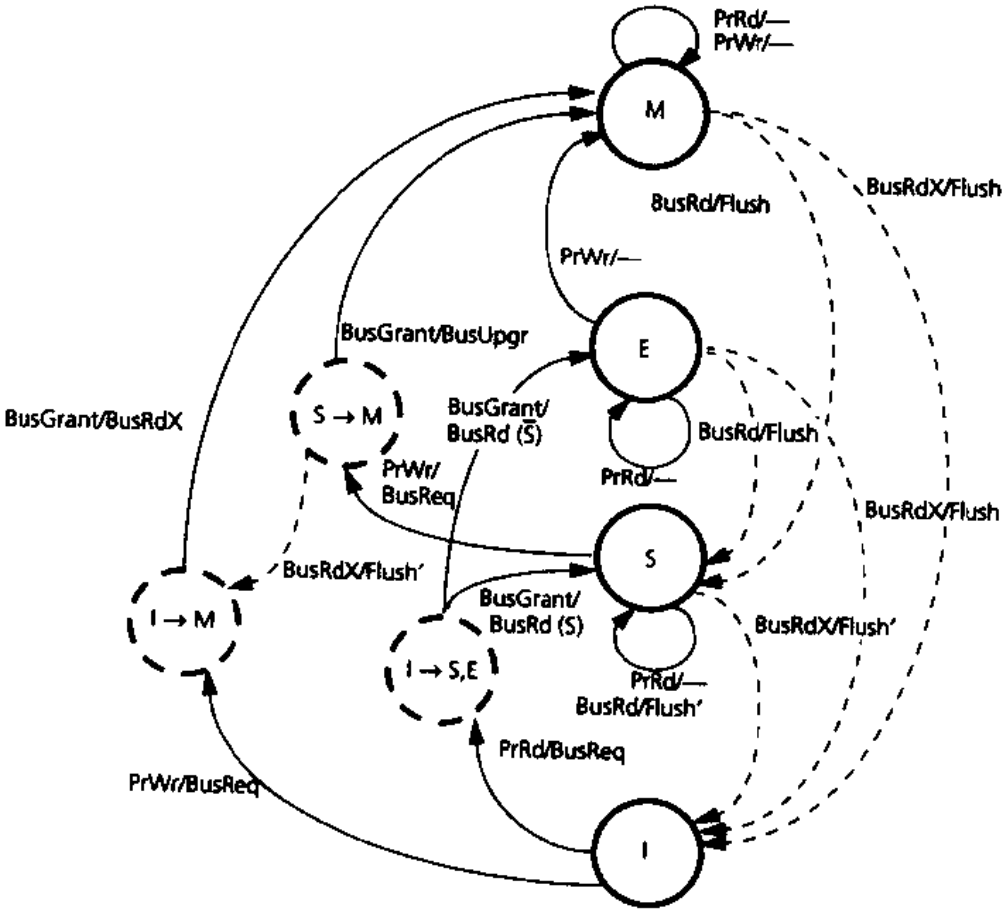


图 6-5　扩充的 MESI 协议状态图，指出了总线获取中的过渡状态。缓存控制器在其请求进行仲裁时监测总线。一个有冲突的事务可能改变稳态之间的转换

高速缓存要将副本置为无效。如果某一高速缓存包含修改状态下的存储块，它就会给出脏信号线，从而阻止存储器提供数据，它自己将数据送到总线上。这里没有用到共享信号线。当总线事务在总线实际进行时，此方法将使发出总线读请求的处理器侦听自身的标记。如果高速缓存中该存储块还处在有效状态下，它就给出共享信号来禁止主存。因为它已经含有有效存储块，该块在任何其他高速缓存中都不可能处于已修改状态，且事务中的数据阶段被忽略。高速缓存控制器不需要过渡状态，因为不管发生什么，它只要做一件事——将 BusRdX 事务放到总线上。

6.2.6　串行化

鉴于不同处理器发出的存储操作的非原子性，在处理器和高速缓存的握手机制当中一定要注意保持由总线事务的串行化所决定的顺序。对于读操作，处理器需要操作结果。为获得写操作上更高的性能，一种有吸引力的做法是当高速缓存控制器获取存储块的独享权时，让处理器在缓存块更新的时候继续执行一些有用的指令，还可能通过一个总线事务装入存储块的剩余部分。这里的问题是，在处理器向高速缓存发出写操作和高速缓存控制器获得排他读（或者升级）事务总线所有权之间有一个空当。正如我们所知，其他总线事务（包括写操作）可能出现在这一空档中，这就可能改变高速缓存中该存储块或其他块的状态。这将使一致性

387
388

写串行化（如果这个事务针对同一存储块）和 SC（如果事务针对不同块）复杂起来。为提供写操作串行化或 SC，这些过程都一定要让处理器看起来是出现在写操作之前，因为这就是它们通过总线得到串行化且告知其他处理器的方式。为保守起见，高速缓存控制器应该不允许发出写操作的处理器，在排他读事务出现在总线上且使得写操作对其他处理器可见之前，认为写已经完成了从而可以去执行程序中在写操作之后的其他操作。

事实上，高速缓存不用等到排他读事务完成之后，即等到其他高速缓存中的其他副本实际被作废后，才允许处理器继续工作；事务一旦出现在总线上，只要传输中存储块的访问能处理合理，处理器就可以为读、写命中服务。5.2 节给出的一致性和顺序同一性的关键论点，在于所有的高速缓存控制器以同样的顺序观察由写操作产生的独享所有权事务（BusRdx 或 BusUpgr），以及独享所有权事务完成后立即将数据写入高速缓存中。一旦总线事务开始，在我们的基本设计中写者认为所有其他的高速缓存将在其他事务发生之前将其副本作废。我们称写操作被提交了，指的是在总线顺序中该写操作的位置已完全确定，与进一步的活动无关。写者不可能确切知道在其他处理器的局部程序顺序中何处会被插入作废操作；它只知道不论操作产生下一步的总线事务如何，插入作废都在此之前且所有处理器以相同的顺序插入作废。同样，写者在其后的本地高速缓存命中的顺序也仅仅在下一个总线事务中才可见。这些为保证一致性和 SC 必须维护的序关系是很重要的，它允许写者能在达到 SC 的充分条件下用提交代替实际完成。事实上，这个基本的观察结论就是有可能用流水的方式实现高速缓存的一致性和顺序同一性的关键；这里的流水包括总线、多层存储器和每处理器的多个待完成操作。写原子性和先前 5.3 节阐述相同。

操作串行化的讨论产生一个重要的但有点难以说清的观点。写串行化和写原子性对于何时把数据写回到存储器，或者何时存储器单元被更新没有任何影响。任何读或写操作如果产生要被覆盖的脏块，就会导致一次回写。回写也是总线事务，但它们不需要按照顺序进行另一方面，一次写操作并不一定导致新的值出现在总线上，即使是扑空了；它会产生一次排他读。对程序来说重要的是新值绑定到地址上的时间。写操作完成，就是说一旦总线读事务
389 （BusRdx）或者总线升级事务（BusUpgr）发生，就会返回刚才写进去或者在其后写进去的值。通过作废旧的高速缓存块，保证了所有返回旧值的读操作在这个事务之前发生。发起这个事务的控制器能保证新的值在总线事务后写入高速缓存且无任何其他的存取操作干扰。

6.2.7　死锁

一种如同存储操作中请求-应答那样的两阶段协议表现出了协议层死锁的一种形式，有时称作“取死锁”（Leiserson et al. 1996），它不仅仅是缓冲区使用的问题。当一实体试图发出请求信号，它需要为接踵而来的事务服务。在带原子总线的 SMP 中，当高速缓存控制器等待总线时将产生下列情况：高速缓存控制器既要继续侦听，又要处理请求信号，而请求信号可能会要求控制器把存储块传输到总线上。如果两个控制器各含有一待完成事务，而这两个事务都要求对方应答却同时拒绝处理请求信号，这样系统就可能死锁。例如，假设总线上出现对 B 块的 BusRd，同时某个处理器 P 对另外一块 A 的排他读的请求信号被总线接受。若 P_i 有 B 块的已修改副本，等待它控制器就应当在等待获得总线时给当前总线事务提供数据（并不需要原子总线的仲裁），且改变修改状态为共享状态。否则，当前总线事务就会等待 P_i 控制器而 P_i 控制器又在等待总线事务释放总线。

6.2.8 活锁和挨饿

在基于作废协议的高速缓存一致性存储系统中，典型潜在的活锁问题是由所有处理器试图同时写入同一存储单元而产生的。设想一下，开始任何处理器在其高速缓存中均无该单元的副本。某一处理器的写操作有下列非原子不可分的系列活动：它的高速缓存获得对应存储块的独享所有权（即它将其他副本作废，获得修改状态的存储块）；处理器中状态机确认高速缓存中该块处于合理状态下；状态机再进行写操作。除非对处理器和高速缓存握手的设计非常严密，不然很有可能出现存储块以修改状态进入了高速缓存，但处理器还没有完成写，该存储块又被其他的处理器的 BusRdX 请求作废了。处理器写操作再次失败，如此可能总重复下去。为避免活锁，一个已获得独享所有权的写操作一定要在所有权被拿走之前能够完成。

当处理器争用总线时，有可能一些处理器反复获得总线而另一些却由于不能获得总线而挨饿。挨饿现象可通过在总线仲裁和其他方面采用先来先服务的原则避免。然而，这往往要另加缓冲设备，因此有时就采用启发式技术来减少可能发生的挨饿现象。例如，记录下该请求被拒绝的次数，一旦超过某个阈值，则拒绝其他请求；直到该请求服务完成，才服务其他新请求，或者可以提高该请求的优先级。 390

6.2.9 原子操作的实现

在讨论更实际的体系结构之前，我们对基本体系结构还应了解的最后一项是原子不可分的读－改－写指令的实现，诸如 test&set 和 fetch&op 以及能合成原子操作的 LL-SC[⊖] 原语(参看第 5.5 节)。

考虑简单的 test&set 指令，该指令包含读成分（test）和写成分（set）。第一个问题就是 test&set（加锁）指令的变量是否可放入高速缓存中从而可以在处理器的高速缓存中执行；如果不能放入，则原子操作在主存里执行。5.5 节中有关同步的讨论假设了锁变量可以缓存。其优点是允许对局部性的利用，从而当同一个处理器反复需要该锁时降低时延和流量；锁变量以已修改状态保留在高速缓存中，不产生作废和扑空。在锁的状态不满足要求时，处理器还可以在其高速缓存中踏步等待，从而减少无用的总线流量。然而，在存储器上执行锁操作能加快锁从一处理器传到另一处理器。对于可高速缓存的锁，忙等待下的处理器首先被置为无效，然后试着从其他处理器的高速缓存或者主存来访问锁。对于不能高速缓存的锁，锁的释放直接反映到存储器（不需要作废什么），且在作废操作到达存储器之前，正在等待的处理器的下一个读操作可能也正在抵达存储器的路上，因此可以以很低的时延从存储器获得锁。总的看来，流量和局部性的因素占支配地位，因此锁变量多为可高速缓存的，从而处理器在忙等待时可不必访问总线。

如果可缓存的 test&set 命令的实现不能由高速缓存本身完成，一种自然的方法是使用两个总线事务：读总线事务处理 test 部分和写总线事务处理 set 部分。保证这个序列原子不可分的办法之一是在读事务时锁住总线直到写过程完成，使其他处理器不能在两个事务之间访问总线。对于原子总线来说，实现这一点非常容易，但对于事务拆分型总线则有较大难度：锁住总线不仅会有损效率，而且若某一事务不放弃总线就不能立即满足的话，则可能导致死锁。

幸运的是我们有更佳的方法。研究一个回写式高速缓存的基于作废的协议。处理器真正

⊖ 即装入链接－条件存储。——译者注

要做的是获得高速缓存块的所有权（如通过发出一次排他读总线事务），然后执行高速缓存
391 中读成分和写成分，只要是两者之间不放弃块所有权就可以；即使算是非原子的总线，新来的从总线到那一存储块的访问也被禁止且一直等到数据写入高速缓存为止。更加复杂的原子操作，如 fetch&op 也都要保留独占所有权直至操作结束。

Compare&swap 是一更难实现的原子指令。它要求在一存取指令中指定三个操作数：存储单元、用于比较的寄存器以及要与存储单元交换的数值/寄存器。RISC 指令系统一般不包括该指令。

实现 LL-SC 需要些特殊的支持。典型的实现方法是在每个处理器中使用硬件锁标记和锁存地址寄存器。LL 操作读存储块的同时也设置锁标记和将块地址放入锁存地址寄存器。新的来自总线的作废（或更新）请求信号和锁存地址匹配，匹配成功（称冲突写）则复位锁标记。条件存储检查锁标记，判断是否发生冲突写；如果标记已经复位，则检查失败，否则则检查成功。如果锁变量从高速缓存被替换，锁标记同样被复位（条件存储失败）。这是因为处理器无法看到对该变量的作废或更新操作。最后，锁标记还可能在上下文切换的时候复位，因为在 LL 和它的条件存储之间的上下文切换可能错误地使得老进程的 LL 导致切换进来的新进程的条件存储执行的成功。

在实现 LL-SC 中为避免活锁问题又产生了新的细节问题：第一，实际上我们不应允许占有锁变量的高速缓存块的替换发生在 LL 和 SC 之间。替换会清除锁标记且形成处理器不断试图执行 SC 但总不成功的情况。由于在 LL 和 SC 操作之间可能有不断的替换发生。为了禁止和取指相冲突的替换，我们可以使用拆分型指令和数据高速缓存或者组相联的统一高速缓存。对于和其他数据引用的冲突，常见的解决办法是简单在 LL 和条件存储之间禁止涉及存储器的指令。隐藏时延（如乱序问题）的技术可能将问题复杂化，因为程序代码不在 LL 和 SC 之间的存储操作到执行时，则可能在两者之间。简单的解决办法就是不允许重排存储操作跨 LL 或 SC 操作发生。

第二种活锁的潜在情况发生在两个进程连续在条件存储上失败，而且每个进程失败的条件存储使其他进程的存储块作废或者更新，从而清除了锁标记。如果这种错误情况继续存在，那两个进程都不能成功。这也就是为什么不把条件存储视为简单的写操作和失败时也不能发出作废或更新命令的重要原因。

和实现原子的读-改-写指令相比，LL-SC 在效率上有问题，这是因为 LL 和 SC 即便成功，但都可能产生缓存扑空。当 LL 装入处于共享状态下的存储块时，就会发生这种情况，导致
392 两个扑空，而不是一个。为提高性能，我们可能希望在 LL 执行时以独占或修改状态获得（或预取）存储块，从而条件存储除非失败否则会扑空。然而，这又产生了第二种活锁情况：将其他副本作废来获得独享所有权，因此如果不能保证这处理器条件存储成功，那其他处理器的条件存储也将失败。如果引入这种优化，则应该在失败的操作之间引入某种形式的回退，以减少（尽管不能完全消除）活锁的可能性。

6.3　多级高速缓存层次结构

上一节给出的简单设计是可以说明一定问题的，但它有两个简化的假设并不适用于大多数现代系统：即单级高速缓存和原子总线。这一节将放弃第一个假设，来研究所导致的设计问题。

自 20 世纪 90 年代早期开始，微处理器的设计趋势一直是两级高速缓存，第一级在片

内，第二级高级缓存容量要大些，可能在片内也可能在片外[⊖]。不少系统也使用片内第二级高速缓存和片外第三级高速缓存。多级高速缓存的层次结构看来使一致性问题更加复杂，因为由处理器对第一级高速缓存做的修改，可能对负责总线操作的更低级的高速缓存不可见，且第一级高速缓存也不能直接看见总线事务。然而，为达到高速缓存一致性的基本机制可以自然地扩展到多层高速缓存上。让我们看看图 6-6 所示的两级高速缓存具体的层次结构；到多级情形的扩展就显而易见了。

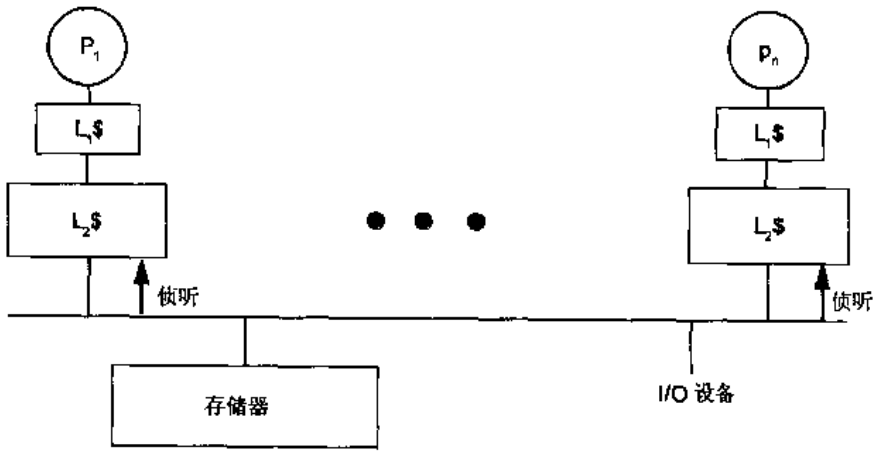


图 6-6　一种基于总线、包含两级高速缓存的处理器的机器

处理多级高速缓存的一种显然的方法是高速缓存层次结构中每级都有相应独立的总线侦听硬件。由于种种原因，此方法不太理想。首先，L_1 高速缓存总是在处理器芯片内，片内侦听器要占用珍贵的芯片管脚来监听共享总线上的地址。第二，做双份标记以允许处理器和侦听器同时访问，也会消耗过多的芯片资源。第三，由于大多时候出现在 L_1 高速缓存的存储块也出现在 L_2 高速缓存上，因此 L_1 和 L_2 侦听有重复操作；因此，L_1 高速缓存的侦听是不必要的。

实用的解决办法就是基于对上面最后一条的认识。当使用多级高速缓存时，设计者要确保所谓包含特性，要求如下：

1）如果一存储块在 L_1 高速缓存内，那它也必在 L_2 高速缓存内。换句话说，L_1 高速缓存的内容是 L_2 高速缓存的子集。

2）如果该存储块在 L_1 高速缓存中是拥有状态（如 MESI 或 MOESI 中的修改状态，Dragon 中的共享修改状态和 MOESI 中的拥有状态），那么它在 L_2 高速缓存一定是已修改状态。 393

第一条要求保证任何与 L_1 高速缓存相关的总线事务必与 L_2 高速缓存相关，因此让 L_2 高速缓存控制器侦听总线就足够了。第二条保证如果一个总线事务向 L_1 高速缓存或 L_2 高速缓存请求某个已修改状态的存储块，L_2 高速缓存能知道这个存储块就在它那里。

6.3.1　包含性的维护

包含性的维护并非小事。有三个方面要考虑到。第一，与 L_2 高速缓存有关的处理器引用

⊖ HP 的 PA-RISC 微处理器是一个特例，在许多其他厂家追求较小的片上一级缓存后，它在多年里一直用一个大容量的片外一级缓存。

能使它改变状态和执行替换，这就需要以保证包含性的方式来处理。第二，总线过程引起 L_2 高速缓存改变状态和输送存储区块，它们要反映到第一级。最后，已修改状态必须要传播到 L_2 高速缓存。

乍一看，由于所有 L_1 高速缓存扑空都会到 L_2 高速缓存中去找，似乎包含性能自动满足。然而问题是在扑空时，两个高速缓存可以选择不同的存储块和数据来替换。只是在某种高速缓存的组合配置下，包含性才能自动维持。研究一下如果不采取特殊措施，在何种情况下典型的高速缓存层次结构会导致包含性被破坏，是一个很有趣的练习（Baer and Wang 1988）。在知道如何保持包含性之前，先研究上面这个问题。为表达方便，假设 L_i 高速缓存的相联度是 a_i，组数是 n_i，存储块大小是 b_i，总容量 $S_i = a_i \times b_i \times n_i$。相应 L_2 高速缓存的参数是 a_2，n_2，b_2 及 s_2。我们还假设参数均是 2 的幂次。

- 基于历史的替换策略，组相联 L_1 高速缓存。基于存储块访问的历史纪录的替换算法，如 LRU（最近最少使用），其问题在于 L_1 高速缓存看到的访问历史和 L_2 高速缓存及其他级看到的不同，因为处理器的所有存储访问都要察看 L_1，但并不一定都到更低层的高速缓存。假设 L_1 是两路组相联，采用 LRU 替换策略，L_1 和 L_2 高速缓存大小相同（$b_1 = b_2$），L_2 是第一级的 k 倍（$n_2 = k \times n_1$）。容易看出在此简单例子中不能保持包含性。假设三个不同的存储块 m_1，m_2 和 m_3，映像在第一级的同一组内；若 m_1 和 m_2 都处在 L_1 内该组的两个有效位置上，且出现在 L_2 高速缓存中。现在看看当处理器引用 m_3，在与 m_1 和 m_2 冲突时，会发生什么事情。它要导致替换 m_1 和 m_2 之一，这不仅涉及 L_1 高速缓存，而且也涉及 L_2 高速缓存。因为 L_2 高速缓存对 L_1 高速缓存的访问史一无所知，而这个访问史决定了 L_1 高速缓存是替换 m_1 还是 m_2；因此容易发现有可能 L_2 高速缓存替换了 m_1 和 m_2 之一，而 L_1 高速缓存替换的却是另外一块。如果 L_2 是直接映像方式，甚至是双路组相联，且 m_1 和 m_2 在同一组，不同的替换也会发生。事实上，将该例子的情形推广可见，如果 L_1 不是直接映像方式，但采用 LRU 替换策略，不管 L_2 的相联度、存储块的大小还是整个 L_2 的大小如何，包含性都可能被破坏。

394

- 在一个层次有多个高速缓存。当第一级高速缓存将数据和指令分开，在发生替换时可能出现类似的问题，即便是采用直接映像方式，且被一体化的第二级缓存支持。首先假设 L_2 高速缓存也是直接映像，在 L_2 高速缓存发生冲突的指令块 m_1 和数据块 m_2 因进入不同的高速缓存中而在 L_1 不发生冲突。如果在引用 m_1 时 m_2 在 L_2 高速缓存中，m_2 将在 L_2 高速缓存中被替换，但在 L_1 数据高速缓存中保留，这就违反了包含性。这就说明，如果在一级有多个独立的高速缓存，即使下面的高速缓存是一体化的，且有很高的相联度，包含性仍无法得到保证（见习题 6.72）。
- 不同的存储块大小。最后，不同的存储块大小也会破坏包含性。考虑某种使用直接映像方式的系统，一体化的 L_1 和 L_2 高速缓存（$a_1 = a_2 = 1$），存储块大小分别为一个字和两个字（$b_1 = 1$，$b_2 = 2$），组数分别为 4 和 8（$n_1 = 4$，$n_2 = 8$）。因此，L_1 的大小是 4 个字，存储字单元 0，4，8，……映射到组 0。单元 1，5，9，……映射到组 1，依次类推。L_2 的大小是 16 个字，字单元 0 和 1，16 和 17，32 和 33，……映射到

组 0；字单元 2 和 3，18 和 19，34 和 35，……映射到组 1，依次类推。现在易看到 L_1 高速缓存可以同时包含单元 0 和单元 17 的字（它们分别映射到组 0 和组 1），而 L_2 高速缓存因为两字映射到同一组（第 0 组）却不是相继的字（于是存储块大小为两个字无济于事），所以不能同时包含它们。正如所示的那样，即使 L_2 高速缓存有更大容量或者更高的相联数，只要存储块大小不同，包含性就可能被破坏。而且我们也已看到了当 L_1 高速缓存有更高相联度时出现的问题。

幸运的是，在最常见的一种情形包含性是自动保持的。那是这样一种情况：L_1 高速缓存采用直接映像方式（$a_1 = 1$）；L_2 高速缓存可以是直接映像，也可是组相联方式（$a_2 \geqslant 1$），采用任何一种替换算法（如 LRU、FIFO、随机），只要求新的存储块同时被放到 L_1 高速缓存和 L_2 高速缓存，块的大小相同（$b_1 = b_2$），并且 L_1 高速缓存的组数小于或等于 L_2 高速缓存的组数（$n_1 \leqslant n_2$）。使用这种配置就是解决包含性问题的一种流行方法。 395

然而，许多实际使用的高速缓存配置并不能自动在替换时保持包含性。解决的办法是通过在高速缓存层次结构中扩充用于传播一致性事件的机制，使包含性得以保持。一旦 L_2 高速缓存内的存储块被替换，该块地址则传到 L_1 高速缓存，将该块作废或者送出（如果已被修改）。若 $b_2 > b_1$，则可能有多个存储块受到影响。

还需要加强处理总线事务和处理写操作的能力。考虑 L_2 高速缓存能见到的总线事务。有些和 L_2 高速缓存有关的过程也和 L_1 高速缓存相关，因此需要广播到 L_1 高速缓存去。例如，如果 L_2 高速缓存内存储块因某一总线事务（如 BusRdx）被作废，如果数据也在 L_1 高速缓存，那作废也要广播到 L_1 高速缓存。有几种方法可以做到此点。其一是所有与 L_2 高速缓存相关的过程均通知 L_1 高速缓存，由 L_1 高速缓存来忽略地址与其任何标记都不相匹配的事务。这种方法会发送给 L_1 高速缓存大量无用的干扰，而且由于高速缓存的标记不能被处理器访问而降低效率。更好点的解决办法是在 L_2 高速缓存内存储块都设置一新状态（称为“包含位”），记录该块是否也在 L_1 高速缓存内。这种方法用一点额外的硬件资源和复杂性代价，能适当地过滤对 L_1 高速缓存的干扰。

最后，当 L_1 高速缓存写命中时，修改操作要传播给 L_2 高速缓存，从而必要时 L_2 高速缓存能够提供给总线最新数据。一种方法是使得 L_1 高速缓存直写。这样做的优点是单周期的写操作易于实现（Hennessy and Patterson 1996）。可是写操作会消耗 L_2 高速缓存的部分带宽。为避免处理器停转，L_1 高速缓存和 L_2 高速缓存之间需要一写缓冲区。另一种满足需求的方法是 L_1 高速缓存采用回写式，因为 L_2 高速缓存的数据并不需要立即更新而是 L_2 高速缓存要知道 L_1 高速缓存是何时有最近的数据，这样增强了 L_2 高速缓存内存储块的状态信息，使得存储块可以置为“修改了但仍是陈旧的（modified-but-stale）”状态。L_2 高速缓存内存储块可以作为一致性协议中已修改的块，但如果需要传到总线上，数据则从 L_1 高速缓存中提取。（设置 modified-but-stale 状态的简单方法是既置修改位也置作废位）。直写式和回写式的 L_1 高速缓存都被许多基于总线的多处理器使用。保持包含性的详细资料可见（Baer and Wang 1988）。

6.3.2　在高速缓存层次结构中传播一致性的事务

假定我们有包含关系，并且可以根据需要传播作废信号以及发出请求到 L_1 高速缓存， 396

现在看有关的过程如何能够在缓存层次结构上下渗透。层次内的协议通过向下渗透（离开处理器），直到碰到一个持有被请求的处于适当状态的存储块或者到达总线，来处理响应器的请求。对这些处理器请求的回应沿着缓存层次向上传送，随着向处理器方向的推进，依次对缓存进行更新。对读操作的应答，将数据装载到层次中的每个处于共享或者独享状态的缓存中，而对于排他读的响应，要装入除了最内层（L_1 高速缓存）外的所有处于已修改但陈旧状态的层次中。在最内层缓存，排他读数据以已修改状态装入，就好像是写入了新数据，这就是最新的拷贝。

从总线来的请求从外部界面（总线）向上渗透，修改一路上缓存块的状态。有些请求要使得一个存储块送回到总线上，这样的请求可以分成两种，一种是“送回请求”，它也使得该块被作废；另一种是“反拷贝请求”，不要求作废。这些请求向上渗透，直到遇到已修改的拷贝，在这一点，为其外部界面的请求产生一个响应。对于简单的作废，没有必要让总线事务停滞直到所有的拷贝被作废。最底层的缓存控制器（最接近总线的）能够看见这个事务在总线上的出现，对请求者来说，这可以保证作废将会以适当的次序完成。一旦作废请求出现在总线上，对这种作废的响应可能通过它自己的总线界面被送到请求处理器，这样就没有响应在目的缓存层次中产生。所要求的只是要在进来的作废信号和其他通过缓存层次的事务之间保持某种次序，我们将在介绍事务拆分型总线时进一步讨论，这种类型的总线允许多个事务在同一时间处于待完成状态。

有趣的是，在多层缓存的情况下，双标记不是那么关键。L_2 高速缓存的作用相当于是 L_1 高速缓存的一个过滤器，筛取从总线来的不相关的过程，从而 L_1 高速缓存的标记几乎全被处理器可用。类似地，由于 L_1 高速缓存可以看作是 L_2 高速缓存和处理器之间的一个过滤器（期望能满足大多数处理器的请求），L_2 标记几乎全部为总线侦听器的查询所用（图6-7）。尽管如此，许多机器在多层缓存设计中依然保持双标记。

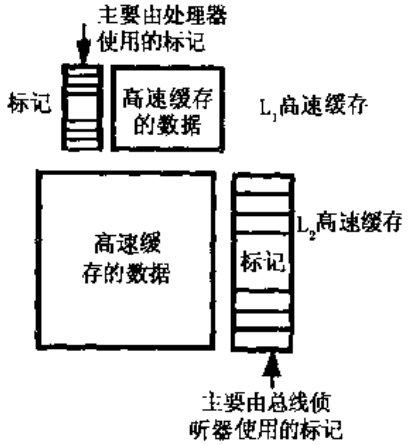


图 6-7　两级侦听式高速缓存的组织。对每个缓存来说只需要一组标记

对于在一个时刻总线上只能有一个事务进行的设计，只要包含关系得以保持，主要的正确性问题并不由于多层次的应用改变多少。必要的事务在层次的上下传播，总线事务可能停滞，直到必要的传播发生。当然，保持总线直到得到一个响应所导致性能的惩罚是更加重的，因此我们试图将这些操作分开。在沿着这条路线进一步下去之前，让我们去掉第二个简化假设，即总线是原子性的假设，考察更大胆些的事务拆分型总线。为简单起见，首先回到单级缓存的情形，然后再考虑多级缓存的层次结构。

397

6.4　事务拆分型总线

对于原子型总线来说，在地址从总线上取走后，直到存储系统或者另一个缓存提供数据响应之前，总线的线路是闲置的，因此原子型总线大大限制了可以发挥出来的总线带宽。在事务拆分型总线的情形，要求得到响应的总线事务被分成两个独立的子事务——一个请求事务和一个响应事务。其他的总线事务（或者子事务）允许在这两个子事务之间发生，这样在产生对一个请求的响应的时候，总线就可能被其他活动利用。在总线和缓存控制器之间设置缓冲，使得总线上有多个活动同时存在，等待来自控制器的侦听和/或数据响应。当然，这里的优点是可以通过总线操作的流水，能更有效地利用总线，从而有更多的处理器能共享相同的总线。缺点是增加了复杂性。

作为请求-响应关系的例子，一个 BusRd 事务现在就是需要一个数据响应的请求。BusUpgr 不需要数据响应，但它要求一个认可信号来指出它已经被确认了从而已经被安排在待完成的序列中。为保证这个认可信号不作为一个单独的事务出现在总线上，它通常是在得到相对于 BusUpgr 请求的总线控制权后被直接送给请求处理器。BusRdX 需要一个数据响应以及一个提交的确认；典型地，它们作为数据响应的部分出现。最后，回写通常没有响应。

拆分型总线所带来的主要新问题有：

1）在侦听和服务一个前面的请求完成之前，一个新的请求可能出现在总线上。特别地，有冲突的请求（两个要求同一存储块，至少一个是写操作）可能在总线上同时处于待完成状 398
态，这是一种必须非常仔细处理的情形。注意这是不同于先前用原子型总线的活动，但宏观上可能出现的非原子性情形。在那里，相互冲突的请求，在它获得总线之前，可以被一个缓存控制器看到，于是就可能在放上总线之前做适当地修改。而这里，两个请求的子事务都已经出现在总线上了。例 6.2 给出了其中的差别。

2）在总线和缓存控制器之间，请求和数据响应缓冲区的大小通常是固定的，并且也很小，于是我们必须面对缓冲器充满的问题，要么想办法不让其发生，要么有一个方法来处理它的发生。由于它影响了通过系统的总线事务流，称为是*流控*问题。

3）由于来自总线的请求被缓冲了，我们需要重新考虑何时、如何在总线上产生侦听响应和数据响应。例如，它们的顺序是否按请求出现在总线上，侦听和数据是不是同一个响应过程的不同部分？

例 6.2　考虑前面两个处理器 P_1 和 P_2 的例子，存储块的缓存状态是共享，并且同时决定对它进行写操作（例 6.1）。说明拆分型总线可能会引入哪些在原子型总线不会出现的复杂性。

解答：对于事务拆分型总线来说，P_1 和 P_2 可能产生 BusUpgr 请求，这些请求在后续周期中得到总线。例如，P_2 可能在缓存中查询到 P_1 的请求，并且在测得会发生冲突之前得到总线。如果它们两者都假设它们已经获得了排他的所有权，这个协议就出问题了，因为此时两个处理器都认为它们拥有处于已修改状态的存储块。在原子总线上，这种事情是不会发生的，因为第一个 BusUpgr 事务会在第二个处理器得到总线以前完成（包括侦听、响应等阶段），于是这第二个处理器就必须将它的请求从 BusUpgr 变到 BusRdX。（注意，即使在例 6.1 中讨论的原子总线出问题的情况下，也是只有一个处理器中的存储块处于已修改状态，其他处理器中存储块为共享状态。）■

事务拆分、缓存一致性总线的设计有很大的空间，并且许多改进正在工业界进行。也许，从一致性协议的角度看，最关键的问题是如何建立一个序以及报告侦听结果的时机。它们是请求阶段的一部分还是响应阶段的一部分？所采取的立场事实上会影响到对冲突操作的处理方式，即，前面描述的第一个主要问题。关于流控的决定（以及冲突操作）是受同时在总线上允许的未完成请求数影响的。通常，较大的未完成请求数使总线的利用率较高，但要求更多的缓冲和设计复杂性。剩下的高层设计决策是数据响应是否需要以和请求发出相同的顺序返回。Intel Pentium Pro 和 DEC Turbo Laser 总线是“保序”的例子，而 SGI Challenge 和 Sun
[399] Enterprise 总线允许变序的响应。后者对于存储访问时间差异的容忍更强（由于存储模块的冲突或者页外的 DRAM 访问，存储器可能会先满足后提交的请求），但更加复杂。让我们先完整地考察一个具体的例子，看这些问题是如何解决的，然后讨论可能的其他方案。

6.4.1 事务拆分型总线设计的一个例子

这个例子主要基于 SGI Challenge 的总线体系结构，即 Powerpath-2。它在三个设计问题上采取如下做法。对有冲突的请求的处理非常简单或者说保守：这种设计不允许对一个存储块同时有多个请求在总线上处于待完成状态。事实上，它只允许在总线上最多同时有 8 个待完成请求，这样就使得必要的冲突检测可行。在总线和缓存控制器之间提供少量缓冲，对于缓冲的流量控制通过总线上的否认回答，即 NACK 线来实现。即，如果在看到一个请求或者响应事务时缓冲区是满的（它一旦出现在总线上就能被检测出来），这个事务就要被拒绝，给出 NACK 信号；这就显示该事务无效，要发起者重新再试。最后，允许响应的次序和最初请求出现在总线上的次序不同。在这种设计中，一致性过程中全（总线）序的建立发生在请求阶段；然而，从缓存控制器来的侦听结果作为响应阶段的一部分和数据一起（如果有的话）出现在总线上。

现在进一步考察这个总线体系结构设计的例子。首先考虑高层总线设计以及响应如何同请求匹配起来，然后深入看看流控和侦听结果的问题。最后，考察一个请求通过系统的通路，包括如何避免相互竞争的请求在总线上同时呈现待完成状态的。

6.4.2 总线设计和请求－响应的匹配

事务拆分型总线的设计在本质上是用两条分离的总线，一条是请求总线用于命令和地址，一条是用于数据的响应总线。请求总线提供请求的类型（例如，BusRd、BusWB）和目的地址。由于响应可以以和请求不同的次序到达，应该有一个方式来让返回的响应和它们的待完成的请求匹配。当仲裁器将总线控制权给予一个请求（命令-地址对）时，这个请求还得到一个惟一的标记（由于这个设计允许 8 个待完成请求，这个标记为 3 位）。响应包含数据总线上的数据和 3 位宽的标记总线上的原始请求标记。标记的使用意味着响应不需要用地址总线，这就使得它们可为其他请求所用。因此地址和数据总线就能分别仲裁。仲裁以及流控和侦听结果也都有分别的总线线路。

在这个设计中，缓存块是 128 字节（1 024 位），数据总线是 256 位宽，于是 4 个总线周
[400] 期和一个 1 周期的往返时间要用在响应阶段。下面会讨论一种统一的流水策略，因此请求阶段也是 5 个总线周期：仲裁、冲突解决、地址、译码和确认。总的来说，一个完整的请求-响应过程要用到这其中至少 3 个周期的活动——至少是地址请求阶段（用到地址总线），数

据请求阶段（用到数据总线仲裁逻辑，为响应子事务获得数据总线的访问），以及一个数据传送或响应（用到数据总线）。三种不同的存储器操作可能同时出现在三个不同的周期中。这种基本流水策略是若干高层设计决策的基础。

为理解这种策略，让我们考虑一个读操作的全过程，如图 6-8 所示。我们从地址请求阶段开始。在请求仲裁周期，缓存控制器向总线提交请求。在请求解决周期，要考虑所有请求，但只有一个会被受理，并分配一个标记。胜者在下面的地址周期驱动地址线，然后所有控制器用一个周期来对它译码并察看缓存标记，查看是不是有侦听命中（侦听结果将在后面提交到总线上）。这时，缓存控制器可以做相应的动作，使这些操作对处理器可见。对于 BusRd，一个独享的存储块降级到共享；对于 BusRdX 或 BusUpgr，存储块就要被作废。在所有情况下，如果一个缓存拥有的存储块处于脏状态，就需要它在响应阶段将那个块送上总线。如果某个缓存控制器在地址阶段不能完成侦听，并且采取必要的行动（比如，如果它没能得到缓存标记的访问），它可以在确认周期冻结这一阶段的继续，直到完成它的侦听。（在确认周期，前一次存储操作的第一次数据传送周期可以进行，占用数据总线的 4 个周期；见图 6-8）。

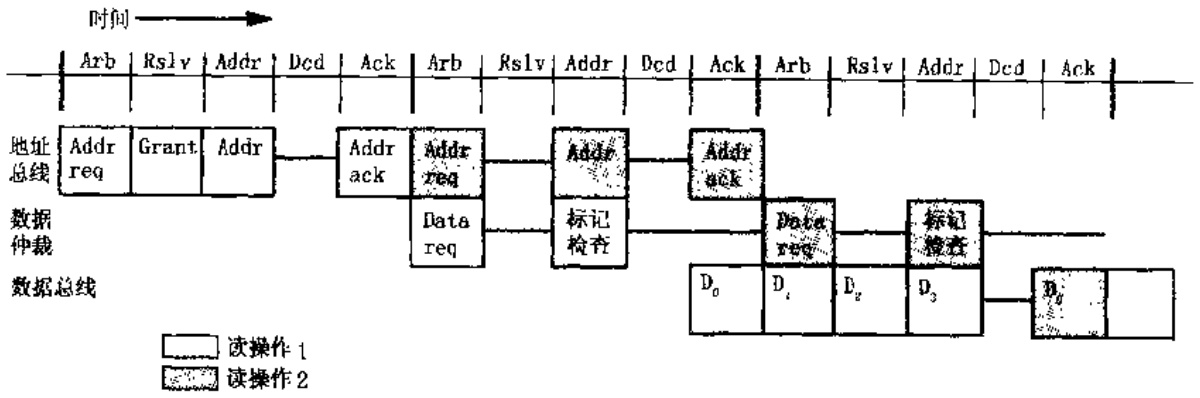


图 6-8　事务拆分型总线的完整的读事务。一对相继的读操作在相继的阶段完成，由阴影框区分。每一个阶段由五个特定的周期组成：仲裁、冲突解决、地址、译码、确认。事务被分为三个阶段：地址请求（用到地址总线），数据请求（用到数据总线仲裁和相应逻辑），数据响应（用到数据总线）

在整个总线事务的地址周期之后，我们就知道了是该存储器还是该缓存提出数据响应。 401
响应者可能在下 5 个周期阶段的仲裁周期请求数据总线。（注意在这个周期请求者也在地址总线上发出一个新的请求。）数据总线仲裁在下一个周期解决，并且在这个地址周期可以检查标记。如果目标准备好了，数据传送就从确认周期开始，并在下 3 个周期继续（即进入数据传送或响应阶段）。在一个来回后，就可以开始下一次数据传送（其仲裁并行进行）。缓存块的共享状态（侦听结果）被带到响应阶段，当数据在缓存中被更新后，设置状态位。

如前面所讨论的，回写（BusWB）只有请求阶段。它们要一起用到地址和数据线，这样就必须仲裁对这两种资源的同时使用。最后，更新（BusUpgr）用来获取对一个块的排他所有权，由于不需要在总线上有数据响应，也只有一个请求部分。对于通过一个写操作产生了 BusUpgr 的处理器，在这个 BusUpgr 出现在总线后，将得到一个由它自己的总线控制器发出的响应，指出这个写操作已被受理并且已安排在总线操作的序列中。

为了跟踪总线上 8 个已受理的请求，每个缓存控制器维护一个有 8 个条目的表，称为请求表（见图 6-9）。只要一个新的请求发到总线上，它就以同样的索引被加到所有的请求表

中，作为仲裁过程的一部分。索引是在仲裁期间分给那个请求的3位标记。（请求也被单独缓冲在缓存层次中）请求表项包含和请求相联的存储块的地址、请求类型、存储块在本地缓存中的状态（如果它已经被确定了的话），以及其他几位。由于请求表是全相联的，因此要考察所有请求表项和请求的匹配，包括本地处理器发出的请求和在总线上看到的其他请求（用地址域）和响应（用标记）。当对于某个请求的响应出现在总线上时，相应的请求表项就被释放。只是在此时，和请求相联的3位标记值才由总线仲裁器重新分配，因此在请求表里没有冲突。

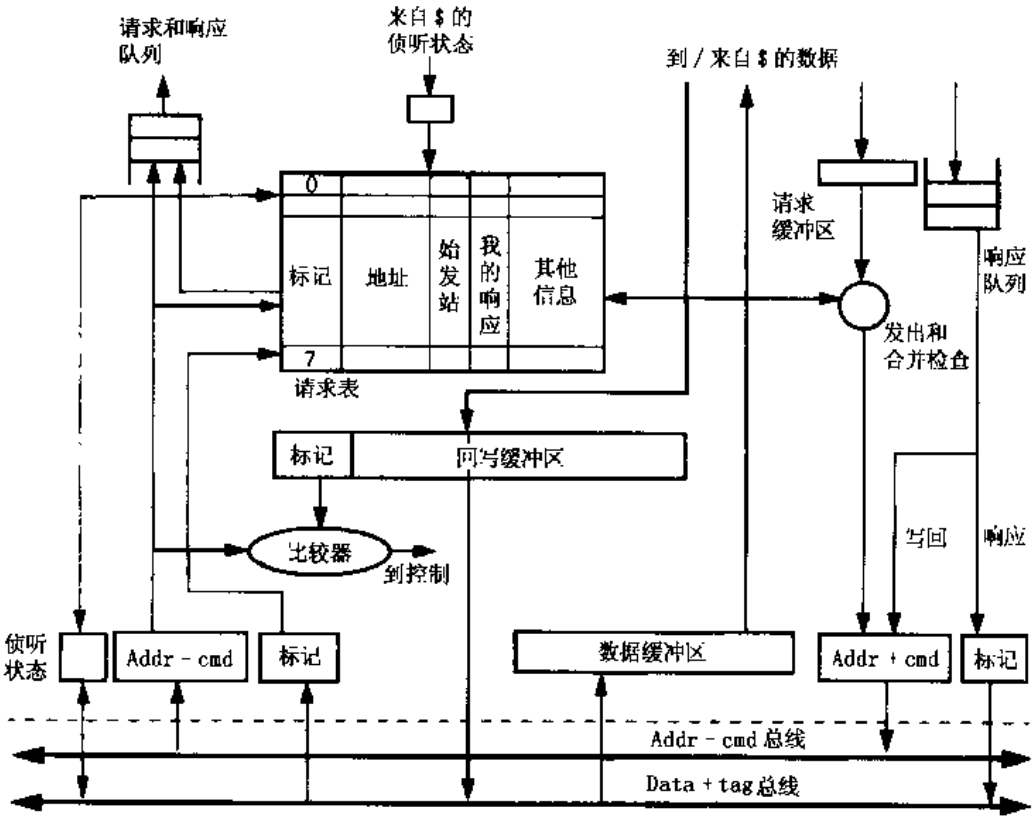


图6-9　图6-4所示的总线接口逻辑的扩充，以完成事务拆分型总线的功能。关键的增加是一个有8个条目的请求表，跟踪所有在总线上待完成的请求。当一个新请求发到总线上，它就被加到所有处理器请求表中的相同的索引处。请求表服务于多个目的，包括请求归并和保证对任何给定的存储块来说只能有一个请求待完成

6.4.3　侦听结果和冲突的请求

如同SGI Challenge，这个设计例子用的是可变延迟侦听法。前边讨论过，总线的侦听部分由三根线或线构成：共享、脏和禁止（它延长当前响应阶段的期间）。当在地址请求阶段末尾确定出哪个模块应该给出数据响应，在数据准备好和响应者获得总线访问之前可能还有许多周期。在这个期间，侦听响应保持在请求表中，其他的请求和响应有可能发生。为了简化请求和侦听结果的匹配，在这个设计中，当所有的控制器看到对一个请求实际的响应被放到总线上时，即在响应阶段，它们就将侦听结果呈现到总线上。回写和升级请求没有数据响应，它们也不要求侦听响应。

避免请求冲突是容易的：由于每个控制器都在它的请求表中有一个记录，表示那些发送

到总线上但未完成的事务，它就不会发出对某个还有待完成事务存储块的请求。这样，即使总线是流水操作，对各个存储单元的操作也是串行化的，同原子情形一样。写操作的确认在请求阶段期间，它影响着串行化。

402
~
403

6.4.4 流控制

除了对来源于总线的入向请求外，在系统的其他部分也可能需要流控。除了前面讨论过的回写缓冲区外，缓存子系统有一个缓冲区，其中可以存放对它请求的响应。如果处理器或者缓存每次只允许一个待完成请求，如我们隐含假设的那样，这个响应缓冲区的深度只有一项。由于响应缓冲区项不仅含有一个地址，还包含一个缓存块数据，因此容量就比较大，于是缓冲区项数通常都是很少的。缓存控制器通过限制它所有的待完成请求数来进行流控，使得对每个响应都有缓冲空间。

主存也是需要流控的。除请求本身外，8个中的每个挂起请求会产生一个主存必须接受的回写。由于回写事务不需要响应，它们可能在总线上迅速连续发生，可能使主存子系统的缓冲区溢出。

由于总线允许对于不同部分进行独立仲裁，SGI Challenge为总线的地址和数据提供分别的NACK线。在一个请求或者一个响应子事务到达它的确认周期和完成之前，主存或任何其他处理器能给出一个NACK信号，例如，如果它发现其缓冲区满。然后这个子事务在每个地方被取消，而且必须重新开始。在Challenge中，一种常用的选择方案是让那个子事务的请求者做周期性的尝试，直到成功。回退和优先级技术能用来减少失败的重试对带宽的消耗，避免“挨饿”进程。对于遇到缓冲区满的数据传送，Sun Enterprise用另一种有趣的方案。在这种情形下，接受者（它不能在第一次尝试时接纳数据）当有足够缓冲时启动重试过程。最初的提供者只是简单地看着总线上重试事务，把数据放到总线上。Enterprise总线的操作保证了当数据到达时，在目的缓冲区的空间是可用的。这就保证了数据传送的成功，只需一次重试总线事务。

6.4.5 一次缓存扑空的路线

给定这个例子设计，我们可以来考察如何处理各种请求，以及会出现什么竞争条件。首先看看这种情况，一个处理器在缓存中有一个读扑空，于是应该产生一个BusRd事务的请求部分。这个请求首先检查当前在请求表中挂起的项。如果发现一个地址匹配，取决于挂起请求的性质，它可能做两种不同的动作。

1）如果当前的请求是针对相同块的，这对处理器来说是一个好消息：这个请求不需要送上总线，而可以在对先前请求的响应出现在总线上时直接获得数据。为做到这一点，我们在请求表中的每一项加上两位，这表示：我希望获得这个请求的数据响应吗？我是这个请求的最初产生者吗？在我们的情形中，这些位被分别置1和0。第一位的目的是明显的；第二位的目的是要帮助确定在哪个状态下（独享还是共享）这个数据响应被装入。如果一个处理器不是最初的请求者，那么它必须在它从总线获得相应数据时，在侦听总线上声明共享线；从而所有缓存都将这一存储块以共享态装入，而不是独享。如果某个处理器是最初的请求者，它从总线得到响应后不会给出共享信号；如果共享信号根本没有出现，那它就以独享态装入该存储块。

404

2）如果前面的请求和 BusRd 冲突（例如 BusRdX），控制器就必须保持请求，直到它看到在总线上有一个对先前请求的响应，而且在那之后才尝试这个请求。处理器方的控制器通常对此负责。

如果控制器在请求表中找不到匹配项，它就可以直接向总线发请求。不过，它必须关注前面提过的那种竞争条件。当控制器首先检查请求表时，它可能找不到冲突请求，因此它可能请求总线仲裁。然而，在它被赋予总线控制权之前，一个冲突请求可能出现在总线上，然后它可能被赋予紧接着的下一次总线使用权。由于这个设计不允许总线上有冲突请求，当控制器看到在它前面的时隙中正好有一个冲突请求的时候，它应该 1）向总线发出一个空请求（没有动作的请求），以占据它已被赋予的时隙；2）从后面的仲裁中退出，直到对这个冲突请求的响应产生。

假设处理器来管理在总线上发出 BusRd 请求，其他缓存控制器和主存控制器应该干什么？请求一旦出现在总线上，就进入所有缓存控制器的请求表，包括发出请求的那一个。控制器开始检查它的缓存，看有没有被请求的存储块。主存子系统根本不知道这个块是否在某个处理器缓存中是脏的，因此独立地开始取出这一存储块。现在要考虑三种不同的情况。

1）某个缓存可能发现它的相应存储块处于修改状态，于是可能在主存响应之前得到总线，从而产生一个响应。在总线上一看到这个响应，主存就放弃它所启动的取数据动作，等待着这一存储块的缓存控制器将数据装载进来，其状态取决于侦听线上的值。如果一个缓存控制器在响应出现在总线上之前没有完成侦听，它就会保持禁止线的作用状态，响应周期就被延长（即将呆在总线上）。由于这个块在缓存中是脏的，主存也接收这个响应。如果主存没有所需的缓冲空间，它就给出用于流控的 NACK 信号，控制器有责任保持存储块的脏态并在以后重试这个响应事务。

405

2）在持有脏块的缓存控制器完成它的侦听和/或获取总线之前，主存可能要取数据并获得总线。持有脏块的控制器将首先置禁止线，直到它完成它的侦听；然后置脏线并且释放禁止线，这就告诉存储器，它有最新的拷贝，而且存储器不应该将数据放到总线上。观察到这脏线后，存储器取消它的响应事务，从而不将数据放到总线上。带有脏块的缓存稍后将获得总线并将数据响应放到总线上。

3）最简单的情形是，没有缓存有脏块。主存将获得总线并产生这个响应。没有完成它们侦听的缓存控制器在看到来自存储器的响应后将给出禁止信号，但一旦它们取消，存储器就可以提供数据。（在这个系统中，缓存到缓存之间的共享技术没有用于共享态的数据）

处理器写操作的处理类似于读。如果进行写操作的处理器在它自己的缓存中没有发现处于有效状态的数据，就会产生 BusRdX。和前面一样，它检查请求表，然后到总线上。除了主存不会接受来自另一个缓存的数据响应（由于它将被写者再次修改）以及没有其他处理器能抓住这个数据外，各个方面都和总线读相同。如果被写的存储块是有效的但处于共享状态，BusUpgr 就会被发出。这不要求有响应事务（当前有效的存储块在主存，也在写者的缓存）；然而，如果任何其他处理器正准备对同一存储块发 BusRdX，它现在就要将请求转换为 BusRdX，这和原子总线是一样的。

6.4.6　串行化和顺序同一性

考虑对单个存储单元操作的串行化问题。如果出现在总线上的一个请求子事务是读，在

这个读后面出现在总线上的写应该不能改变由这个读返回的值。尽管在总线上有多个待完成的事务，由于对相同单元的冲突请求不允许同时出现在总线上，做到这一点也是不难的；因此读响应子事务将优先于这个写请求，它将在写操作能影响缓存的值之前完成。如果出现在总线上的事务是由一个写操作产生的 BusRdX 或者 BusUpgr，请求缓存将对缓存阵列完成这个写，在响应阶段后和发出任何其他存储操作之前发生；从任何其他处理器来的对该存储块其后（冲突的）的读只允许在该写的响应阶段之后的总线上发生，因此它们保证得到的是新数据。（回顾前面讨论过的，写操作的响应阶段可能是总线上的一个单独的操作，如同 BusRdX，或者是在请求赢得仲裁后隐式产生的，如同 BusUpgr。）

现在考虑对不同单元操作的串行化所涉及的顺序同一性问题。总线过程逻辑全序的建立是通过对地址总线请求所赋予的序来确立的。一旦 BusRdX 或 BusUpgr 得到了总线，相联的 406
写就被认可了。然而，对于总线上有多个待完成请求的情况，作废也被缓冲起来，可能在它们被实际应用到缓存上之前还有一段时间（不同于原子总线，那里作废是立即发生）写操作的认可不担保由这个写所产生的值已为所有其他处理器可见；只有实际的完成才能保证（针对一个处理器的动作对该处理器有保证）需要有进一步的机制来保证所需的序在总线和处理器之间能够维持。例 6.3 将这一点具体化了。

例 6.3　考虑如下所示的两个代码段。在 SC 要求下，(A, B) 的什么结果是不允许的？假设每个处理器只有一级缓存，总线上有多个待完成事务，没有特别的机制来保持总线和缓存或者处理器之间的次序，表明如何能得到不允许的结果。假定一种基于作废的协议，在两个缓存中 A 和 B 的初值都是 0。

P_1	P_2
A = 1	rd B
B = 1	rd A

P_1	P_2
A = 1	B = 1
rd B	rd A

解答： 在左边的第一个例子中，SC 条件下不允许的结果是 (A, B) = (0, 1)。然而，考虑下面的情形。P_1 对 A 的写认可，于是它继续写 B（在修订后的 SC 充分条件之下）。在 A 之前，B 的作废应用到 P_2 的缓存，这是由于它们在缓冲区中重新定序。P_2 在 B 上引起一个读扑空，得到新的值 1。然而，A 的作废仍然是在缓冲区中，即使在 P_2 发出读 A 之前也应用不到 P_2 的缓存。读 A 是一个命中，它的完成使 A 从缓存中得到旧值 0。

右边的例子对于破坏 SC 并不需要作废的重新排序。不允许的结果是 (0, 0)。然而，考虑下面的情形。P_1 发出并认可它对 A 的写，然后完成对 B 的读，读进旧值 0。P_2 然后写 B 并认可，因此 P_2 继续读 A。对 B 的写出现在总线上（认可）是在 A 的写之后，因此它们应该以这种次序串行化，P_2 应该读到 A 的新值。然而，对应于 P_1 写 A 的作废仍在 P_2 的输入缓冲区，还没有被应用到 P_2 的缓存中。P_2 看到 A 上的一个读命中，并完成返回 A 的旧值 0。■

如果我们用认可代替完成，并且允许在总线和处理器之间有多个待完成操作缓冲起来，此时为保持顺序同一性必须的关键性质是：不应该允许一个处理器在先前的写（总线序）为它所见之前，实际看到的是由于当前写所产生的新值。有两种保持这种性质的方式：不让某些从总线到缓存的输入过程在输入队列中重定序；允许重定序，但其后要保证重要的序在机器中的必要点得到维持。下面简单考察这两种做法。 407

一种遵循第一种策略简单的方法是保证所有从总线来的事务（作废、读扑空答复、写提

交确认等）以 FIFO 序传播到处理器。然而，这样一种严格的定序是不必要的。针对一种基于作废的协议，考虑维持刚才描述过的所期望的性质。这里，有两种方式让新的值读到缓存中，并且使它为处理器可读而不引起另外的总线操作。一种是通过一个读扑空，另一个是通过那个处理器的一个写。另一方面，一个处理器可以通过其他处理器的作废操作，来看见那些处理器的写（即使这个值在本地还不可用）。为了使写操作定义成前面那样提供新值的操作，它们必须在操作前出现在总线上（在写命中的情形，则是一个先前发自那个处理器的总线事务）。因此，在相关事务出现在总线上的时候，也就是当它的响应回来时，这些写所产生的作废就已经在输入队列中了，或者兑现到缓存中了。因此，我们需要保证的是一个答复（读扑空或者写提交确认）在总线和缓存之间不越过一个作废，即所有先前的作废都兑现，先于缓存接收到这个答复。

注意，进入队列中的作废可能会被重新定序。这是因为对应于一个作废的新值只有通过相应的读扑空时才被看见，而对读扑空的答复不允许相对先前的作废重新定序。另一方面，在一种基于更新的协议下，一旦进来的更新被兑现，就能够看见相应写的新值。这意味着，不仅答复不应该超越更新，更新之间也不应该相互超越。

一个备选方案是允许从总线来的入向事务在它们到达缓存的路上随机重新定序，但要保证的是：如果一个来自本地处理器的操作使它能看见一个新值，则在这个操作被完成之前，所有事先认可的写被兑现到缓存中（通过从入向队列提供服务）。毕竟，重要的不是作废或者更新被兑现的次序，而是相应的新值能被处理器看到的顺序。有两种自然的方法能达到这一点。其一是每当处理器试图完成一个向缓存写入新值的操作时，处理从队列中来的入向作废和更新。在基于作废的协议，这意味着要首先处理队列，先于允许处理器来完成一个读扑空或者一个要产生总线过程的写；在基于更新的协议中，它意味着对每次读命中都要服务。另一个方式是当一个处理器真正快要访问一个值的时候（完成一个读命中或扑空）来处理队列，如果一个新值（即一个答复或一个自从上次队列被服务后的一次更新）的确兑现到了缓存。从总线到缓存的操作重新定序，以及一个新值被兑现到缓存的事实意味着，作废或者更新可能在队列中，对应于先前和那个新值有关的写；这些写现在应该在读能够完成之前兑
408 现。这些技术不会引起如例6.3所讲的不希望的结果，见本章的习题。这些习题可能有助于使这些技术更具体化。正如我们即将能看到的，将这些技术扩充到多层缓存结构是相当自然的。

不管用哪种方法，写的原子性由总线的广播特性自然地得到保证。这个总线隐含着写操作被认可的顺序对所有处理器都是相同的，并且在写对于所有处理器都认可之前，一个读没法看到由它产生的值。（注意到写处理器也在本地保证这一点）。用前面这些技术，我们就能用完成来替代提交，从而保证原子性。对于事务拆分型总线的另一些主要的正确性问题——死锁、活锁、挨饿——在我们介绍了多层缓存之后会有讨论。首先，让我们看看在事务拆分型总线上协议设计的一些方法。

6.4.7　其他设计选择

针对请求－响应的定序、请求冲突的处理和流控，人们还研究了其他一些方法，不同于我们用作例子的事务拆分型总线设计。例如，保证响应在总线上产生的次序和请求的次序一致——缓存控制器倾向于此——将简化设计。为了请求－响应的匹配，全相联的请求表可以被一个简单的 FIFO 缓冲区代替（如果不允许冲突的请求，全相联查询可能依然是需要的）。

如前所述，只有当一个请求实际出现在总线上时，它才被放到 FIFO 中，保证所有实体（处理器和存储系统）对未决请求都有完全相同的看法。缓存控制器和存储系统按 FIFO 序处理请求。当响应出现的时候（如前面的设计），如果其他的处理器还没有完成它们的侦听，它们就要置禁止线，延长其事务的周期，即侦听依然和响应一起报告出来。区别在于，尽管一个处理器在它的缓存中有一脏块，但存储器先产生了一个响应。在先前无序设计的情况下，有着脏块的缓存控制器释放禁止线，置脏线，当它从缓存获得了数据后再次参与总线的仲裁。但现在为了保持 FIFO 序，这个响应必须放到总线上，先于针对任何后来请求的响应。于是带有脏块的控制器不释放禁止线，而是延长当前的总线事务，直到它从其缓存中得到那一块数据提供给总线。实现这一点和任何其他实体访问总线无关，因此没有死锁问题。

尽管 FIFO 请求-响应序是比较简单，它可能有性能问题。考虑带有交叉存储系统的多处理器。假设有三个请求 A，B，C 依次发到总线上，A 和 B 到同一个存储模块，C 到另一个存储模块。迫使系统按序产生响应意味着 C 要等待 A 和 B 的处理完成，尽管由于 A 和 B 发 409
生模块冲突，C 的数据先于 B 可用。主存的行为是允许变序响应主要的动机，因为缓存不管怎样都可能是按序对请求做响应的。

让响应保持其顺序，也使得处理总线上同时存在的对同一存储块的冲突请求更容易些，这样就消除了对全相联请求表查询的需要，也增加了带宽。假设两个 BusRdX 请求连续发给一个存储块。如前所述，发出后面一个请求的控制器在看到前面的请求后将要作废它的存储块。对于事务拆分型总线来说，问题在于发出前面请求的控制器，在它所等待的数据响应之前看到了后面的请求。控制器不能由于这个后面的请求简单地作废它的块，因为那一存储块正在去往发出前面请求的控制器的路上，它自己的写应该先于一个请除或作废之前完成。对于乱序响应来说，允许这种冲突请求可能是困难的。对于保序响应，前面的请求者知道它的响应会首先出现在总线上，因此这实际上是一种性能优化的机会。前面请求的控制器对后面请求的反应是简单地注意到后者是待完成的。当它的响应块到达时，它就更新要写入的字，并且“顺手”将修改后的存储块送回到总线上，作为对后面请求的响应，使它自己的块无效。这种优化降低了在写-写伪共享下往复转换一个存储块的时延。

如果从请求到侦听结果的延迟是固定的，就可以允许冲突请求的存在，甚至不需要数据响应的保序。然而，由于对一个存储块的冲突请求也要到存储器中的同样一个队列中，对这些请求的数据响应本身通常就是按序出现的，于是它们可以由上述快捷的方法来处理（在 Sun Enterprise 系统中就是这么做的）。

事实上，只要一个明确定义的序能在请求过程之间识别出来，它们甚至不需要顺序地发到同一个的总线上。例如，Sun SparcCenter 2000 用两个不同的事务拆分型总线，CRAY 6400 用四个，以改善大配置系统的带宽。于是就可以在一个周期中发出多个请求。同时，在总线之间建立起一个简单的优先级，从而在并发的请求中间定义了一个逻辑顺序。

6.4.8 带有多级高速缓存的事务拆分型总线

现在可以将基本协议上的两种主要的增强技术结合起来：多级缓存和事务拆分型总线。我们考察的设计是一个（类似于 Challenge）事务拆分型总线和两级缓存层次结构。其中的问题和解决方案可以推广到更深的层次。我们已经看到了请求、响应和作废沿这个层次结构传播的基本问题。我们需要对付的新的关键问题是，一个请求传播通过控制器要占用相当多的

410 总线周期。在这个期间，我们必须允许其他的过程也在层次中上下传播。为了允许单个单元（例如，控制器和缓存）以它们自己的速度运行的同时维持高带宽，也在层次的级别之间布置了队列。然而，这就引起了关于死锁和串行化的一系列问题。

一个简单的多级缓存组织如图 6-10 所示，假设一个处理器同时只能有一个待完成请求，于是在处理器和第一级缓存之间就没有队列。这种队列结构要关心的一个问题是死锁。为了避免前面所讨论过的取数死锁问题，L_2 缓存在有请求未完成时需要能缓冲入向请求或响应，从而使总线得以自由。对于每个处理器有一个待完成请求的情形，在总线和 L_2 缓存之间的入向队列要足够大，能够容纳从其他处理器来的待完成请求数加上对它自己请求的一个响应。这覆盖了所有请求都到达同一个缓存且这个缓存有一个请求待完成的情况。如果队列比这种情况的要小，在队列满了不足以接收新的请求时，总线请求就应该被给予 NACK 信号。这种说法适用于带有事务拆分型总线的单级或多级缓存的系统。在总线到 L_2 之间和在 L_2 到 L_1 中的一个时隙预留给对处理器待完成请求的响应，以便每个处理器总能排干它的待完成响应。如果用 NACK，总线仲裁需要含有一种机制，诸如一种简单的优先级方案，来保证在严重的竞争条件下使系统仍能向前推进。

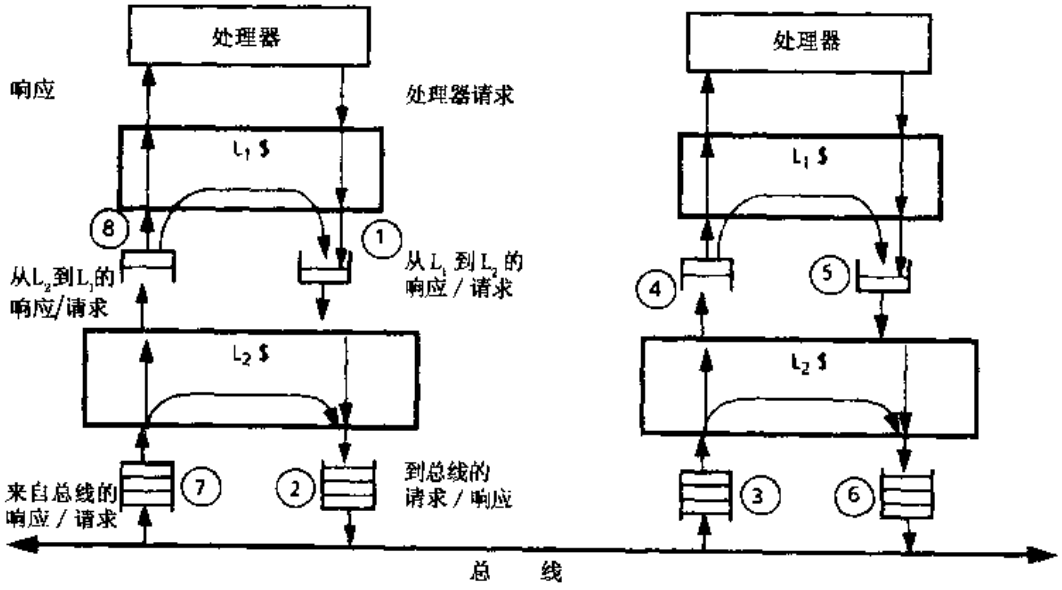


图 6-10 可能存在于多级缓存层次结构内部的队列。每个层次有一个自上的输入队列和一个自下的输入队列。一个操作可能产生对相邻层的请求或者响应。例如，一个在 L_1 缓存扑空的读请求被送到 L_2 缓存①。如果也扑空，总线上就会出现一个请求②。在输入队列中该读请求被所有其他缓存控制器捕获③。假设存储块当前以修改状态出现在另一个处理器的 L_1 缓存中，这个请求就在它的 L_1 服务中排队①。L_1 将该存储块降到共享状态，并送到 L_2 缓存中⑤，由它再送上总线⑥。这个响应被请求者捕获⑦并送给它的 $L_1$⑧，在那里向处理器提供所要求的存储字

除了取数死锁外，经典的缓冲区死锁也可能在多层缓存层次之内发生。例如，假设在 L_1 和 L_2 缓存之间每个方向都有一个队列，两者都是回写缓存的，每个队列能持有一个数据项。有可能 $L_1 \rightarrow L_2$ 队列持有一个出向读请求，它能够在 L_2 中被满足，但要对 L_1 产生一个回答；$L_2 \rightarrow L_1$ 队列持有一个入向读请求，它能够在 L_1 缓存中被满足，但要对 L_2 产生一个回答。现在有了一个典型的循环缓冲依赖关系，因此死锁。注意，这个问题只出现在这样的层

次结构中：上层的缓存（靠近处理器），而不是最靠近总线的，是一个回写缓存。否则，入向请求不产生来自高层缓存的答复，于是就没有循环，也就没有缓冲区死锁问题（注意，作废的确认被总线自己隐含完成，不需要来自缓存的确认）。

在多层回写缓存层次系统中，一种对付这种缓冲区死锁问题的硬件方法是限制从处理器来的待完成请求数，在每个缓存对入向请求和响应提供足够的带宽。然而，这要求用到大量的器件并且不是可扩展的。每个请求可能需要两个出向缓冲区项——一个是为请求，另一个是为它可能产生的回写。对于允许大量待完成总线事务的情况，入向缓冲区也可能需要有许多项。另外一种方法是用一种通用的死锁回避技术，针对有限缓冲的情形，我们将在第7章和物理上分布存储的系统一并讨论，在那种情况下，这个问题更加明确。基本思想是将流过缓冲区和通信介质的操作分离成请求和响应。一个操作被分为响应，如果它不产生进一步的操作，只是简单地被其目的地吸收。一个请求可能产生一个响应，但没有操作能产生请求（尽管在这种情况下，一个请求如果不在最初层次产生一个响应，则可能传送到层次的下一个级别，初始化一个新的请求-响应对，将自己结束）。针对这种分类，我们可以避免死锁，如果我们在每个方向为请求和响应提供单独的队列，保证响应总被从队列上抽取出来，这样也允许请求向前推进。在第7章讨论了这个技术后，我们在练习中将它应用到这个特别的带有多层回写缓存的情形。

还有其他一些潜在的死锁情况。例如，如果总线上的待完成过程数小于缓存允许的待完成请求数，处理器缓存来的一个响应可能需要先于从它发出的新的外向请求到达总线。否则，存在的请求可能总不会得到满足，于是就不会有进展。出向队列或者队列组必须支持在必要时响应能够绕过请求。

除死锁外，对于这些队列结构的一个考虑是维护顺序同一性。对于多级缓存来说，重要的是总线不要等待一个作废信号一直上朔到第一级缓存，返回一个答复；反过来它应该考虑放上总线后所提交的写，这样的写位于最低级缓存的输入队列中。提交和完成的分离在这种情形更加明显。然而，我们所讨论过的对于单级缓存的技术在此自然地得到扩充：我们只要简单地将它们应用到缓存层次的每一级。这样，在一种基于作废的协议中，第一个技术扩充到保证在层次的每一级答复不要针对那一级的入向队列中的作废重定序（为这个目的，从低级缓存到高一级缓存的答复也被当成答复处理）。第二个技术的扩充体现在，在对某一层次的入向作废兑现之前，不让出向存储操作推进超过该存储层次；或者是排空对一个层次的入向作废，如果一个答复自从上一次排空已经在那一级兑现的话。

6.4.9 对一个处理器有多个待完成扑空的支持

尽管我们已经考察了事务拆分型总线，到目前为止我们还是隐含地假设了一个给定的处理器在一个时刻只能有一个待完成的存储请求。这个假设对现代处理器来说是简化的，即使是单处理器系统，现在的处理器都允许多个待完成请求以包容缓存扑空的时延。一方面，允许处理器有多个待完成引用会改善性能，但它也将语义复杂化了，使得来自同一处理器的存储访问可能在存储系统中完成的序不同于它们发出的序。

多个待完成引用的一个例子是用写缓冲区。由于希望让处理器在发出了第一个写操作后继续进行其他的计算甚至存储操作，我们将这个写操作放到写缓冲区。直到写被串行化，它不应该可见；否则，它可能违反写串行化和一致性。一个可能性是将它写到本地缓存，但在

独占所有权得到之前不使它可用（即在此之前不让缓存响应关于它的请求）。更通常的一个办法是将它保持在写缓冲区中，只是当得到了独占所有权才将其放入缓存。

多数处理器用更大胆的方式使用写缓冲区，处理器不停地快速发出一个写序列到写缓冲区。在单处理器，只要读操作检测写缓冲区以满足相关性，这个方法就非常有效。在多处理器中的问题是，通常在关于一个存储块的独占所有权过程被放到总线上，因此被串行化之前，不能允许处理器推进让存储操作超越这个写。然而，有些特别的情形，处理器能够发出一个写序列，并认为它们是完成了，不需停滞。一个例子是是否能确定写操作所针对的存储
413 块处于已修改状态。然后它们可以在处理器和缓存之间被缓冲，只要缓存在服务来自总线方的请求（对一致性来说是相同的块，对 SC 来说是任何块）之前处理这些写操作。存在一个重要的特殊情况，写序列能被缓冲而不管缓存的状态：写都是针对同一个存储块，没有来自那个处理器的其他存储操作插在这些写之间。当控制器正在得到读排他事务的总线时，这些写可能被归并。当那个事务出现的时候，它立刻使整个写序列可见。这个行为如同在这个总线事务后但在下一个总线事务之前写在本地命中并完成。注意，回写序列没有问题，协议不要求它们有什么顺序。

更一般地讲，为了满足顺序同一性的充分条件，一个有能力跨越待完成的写，甚至读操作的处理器提出了这样的问题：哪个实体应该等待来“发出”一个操作，直到程序原序前面的那个完成。强迫处理器自己等待，可能会丧失所有由复杂的处理器机制带来的好处（诸如写缓冲和乱序执行）。相反，由于这里的问题是可见度，持有待完成操作的缓冲区（诸如写缓冲或者是在动态调度乱序执行处理器中的重定序缓冲）可以为此目的服务。处理器能够紧接着前一个操作发出下一个操作，缓冲区承担的责任有两方面：一是担保在适当的时间之前写操作不被存储器和互连系统可见（即不将它们发到外部可见的存储系统）；二是读操作不允许乱序完成（相对于待完成写提交的次序），即使处理器可能乱序发出并执行它们。单处理器中提供精确中断的机制通常可用于这里的缓冲区，在后面的章节中将讨论它们。当然，某些较简单的处理器，不支持操作推进跨越读或写，能使顺序同一性的维护比较容易些。在第 9 章考察同一性模型的时候，还会进一步讨论多个待完成引用对存储同一性模型的语义影响。

从设计的角度来看，最有效地开拓多重待完成引用要求缓存允许在同一时间有多个缓存扑空待完成，从而这些扑空的时延能被重叠。这反过来要求缓存或者某些辅助数据结构来跟踪待完成的扑空；由于响应可能乱序返回，这可能是相当复杂的。允许多重待完成扑空的缓存称为免锁定缓存（Kroft 1981），相对于阻塞缓存，其只允许一个待完成扑空。在第 11 章中，讨论包容时延措施的时候，也将讨论免锁定缓存的设计。

最后，考虑事务拆分型总线和多级缓存层次之间的相互作用，以及避免死锁的需求。给定一个支持事务拆分型总线和多级缓存的设计，为支持每个处理器有多个待完成操作所需的
414 扩充并不多，且多数只是出于性能的考虑。我们只是需要在处理器到总线之间提供较深的请求队列（在图 6-10 中，请求队列向下指），因此多个待完成操作能够缓冲起来，于是处理器或者缓存能够不停滞。更深的响应队列和较多的回写缓冲也可能是有用的，这时系统能承受更多的并发性。只要死锁的处理是通过不同于响应的请求来完成的，并且为它们提供逻辑上分开的缓冲，这些队列的准确长度对正确性的影响都不大。我们看到这里做的改变是相当小的，其原因是免锁定缓存本身对同一块响应的任务做了复杂的融合请求和管理，因此对于缓存以及它下面的总线子系统来说，只看到对于不同块的多个请求来自该处理器。某些潜在的

取死锁情形可能会暴露出来，它们在每处理器只有一个待完成请求情形不会出现；例如，我们现在可能看到这种情形，从所有处理器来的待完成请求的总数多于总线能够接受的数量；因此，需要保证响应能顺便旁路掉一些请求。尽管如此，我们讨论过的支持多个待完成过程的技术（针对事务拆分型总线）使系统的其余部分能够对付源于一个处理器的多个请求，而不产生死锁。

6.5　实例分析：SGI Challenge 和 Sun Enterprise 6000

前面的部分论述了一般性的设计和实现，接下来将讨论两个具体的基于总线的多处理器系统——SGI Challenge 和 Sun Enterprise 6000。我们将更多地集中于讨论这些实际系统的组织和工程方面的问题，而较少地注重它们的逻辑，考察它们在若干问题上采取的不同做法。

SGI Challenge 被设计成为能支持到 36 个 MIPS R4400 处理器（最高能达到 2.7 GFLOPS）或 18 个 MIPS R8000 处理器（最高能达到 5.4 GFLOPS）。两者都使用 Powerpath-2 总线，最高带宽 1.2 GBps。此系统能支持高达 16 GB 的 8-通路交叉存取主存储器，多达 4 个 PowerChannel-2 I/O 总线（每个总线支持最高带宽 320 MBps 和多以太网连接、VME/SCSI 总线、图形卡以及其他外围设备）。全部磁盘存储容量能达到几万亿字节。操作系统使用 IRIX（它是 SVR4 UNIX 的一个变型。具有对称多处理器内核，能把操作系统的任务分配到任一个处理器上进行）。图 6-11 给出了 SGI Challenge 系统的高层组织结构。

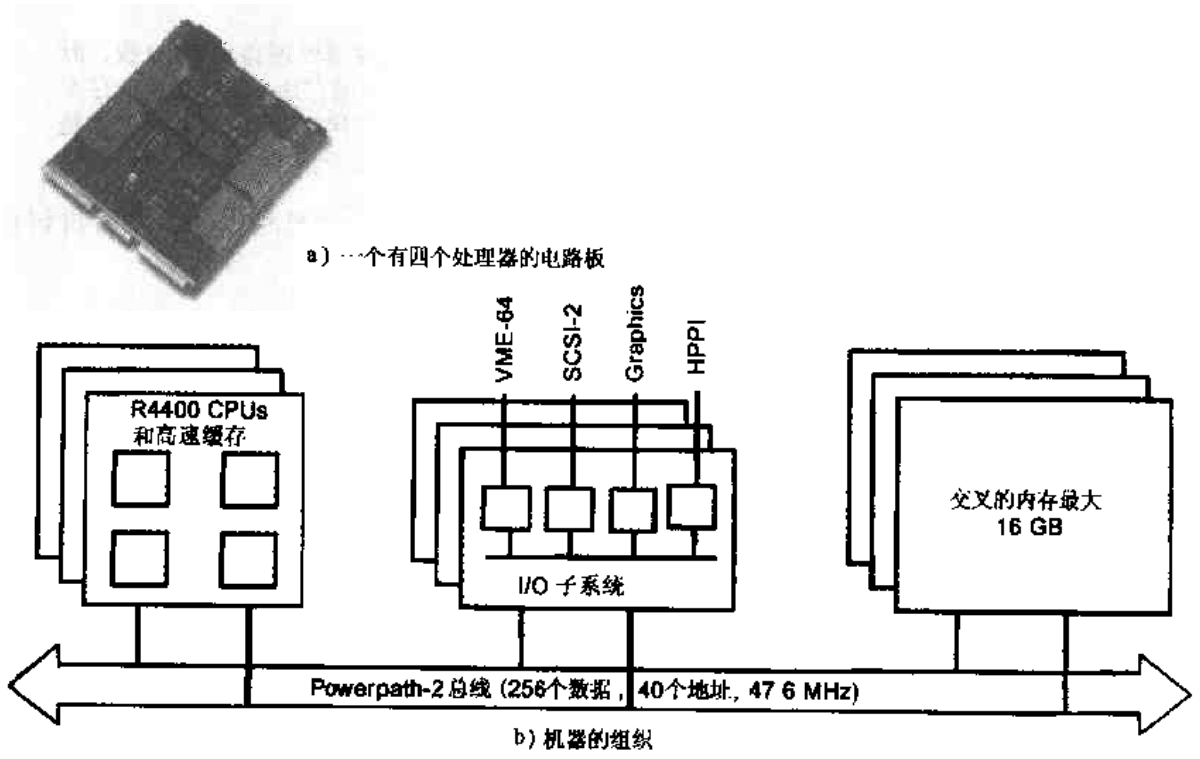


图 6-11　SCI Challenge 多处理器　每个板有 4 个处理器，9 个总线槽共可容纳 36 个处理器，此系统能支持高达 16 GB 的 8 通路交错主存储器。I/O 板提供一个独立的 320 MBps 的 I/O 总线，通过它与其他标准总线和设备相接。系统总线有一个独立的 40 位地址通路、一个独立的 256 位数据通路以及命令通路与其他信号的通路，支持最高带宽 1.2 GBps。总线是事务拆分型的，在给定的时间内可以有多达 8 个请求在总线上等待处理

来源：CHALLENGE 是 Silicon Graphics Inc. 公司的注册商标。

Sun Enterprise 6000 被设计成能支持多达 30 个 UltraSparc 处理器（最高能达到 9 GFLOPS），支持 Gigaplane 系统总线（总线支持最高带宽 2.67 GBps），支持高达 30 GB 的 16 通路交叉存取存储器。机器有 16 条槽，能混合装备处理板和 I/O 板，且每种板至少需有一个。每个处理板有两个 CPU 模块，两个最高达 1 GB 的 512 位宽的内存模块（因此，系统的内存容量和带宽受限于处理器数目）。虽然特定的内存在物理位置上与一特定的处理器对相连，但由于对内存的存取都是通过系统总线进行的，故各内存的存取时间都是一致的。每个内存都有特定的地址，它所处的电路板就叫做此特定地址的主板（home board）。每个 I/O 板都提供两个独立的 64 位 x 25 MHz 的 SBUS I/O 总线，因此，系统的 I/O 带宽受限于 I/O 板的数量。所有磁盘存储容量能达到数十万亿字节。操作系统使用 Solaris UNIX。图 6-12 给出了 Sun Enterprise 系统的高层结构。

415
~
416

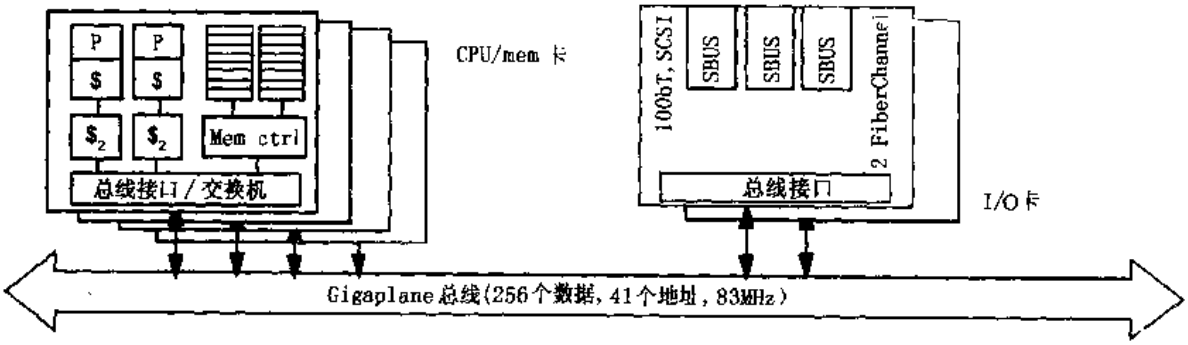


图 6-12　Sun Enterprise 6000 多处理器。系统提供了 16 条总线槽，能容纳处理器板或 I/O 板，但每种都至少得存在一个。处理器板能容纳两个处理器和两个内存库，对所有板内存库都能被一致地存取。I/O 板为多种独立的外部设备总线建立了接口，因此看起来就像是系统总线的高速缓存控制器。事务拆分型总线允许有多达 112 个请求在总线上等待处理

下面几小节将描述 SCI Challenge 的体系结构，并给出它的一些性能特征，然后再讨论 Sun Enterprise 6000。

6.5.1　SGI Powerpath-2 系统总线

系统总线是系统内各部件间相互连接的核心。因此，它的设计要考虑所有其他部件的请求，相应地，一旦采用了某种设计方案，也会影响其他部件的设计。总线设计方案的选择要考虑如下因素：多选还是非多选的地址和数据总线、较宽的数据总线（如 256 位或 128 位）还是较窄的数据总线（如 64 位）、总线的时钟频率（它受所用的信号发送技术、总线长度、槽的数目等的影响）、事务拆分型还是原子型设计、流控策略等等。Powerpath-2 系统总线是非多选的，它有 256 位宽的数据部分、独立的 40 位宽的地址部分、以及命令部分和其他信号部分。它采用的时钟频率是 47.6 MHz，它是事务拆分型设计，并支持 8 个等待的读请求。然而宽数据通路意味着其他设备连接到总线的硬件开销是较大的（连接要求多个位片芯片做接口），好处是一个适中的时钟频率也能得到 1.2 GBps 的高带宽。总线支持 16 个槽，其中的 9 个能装备四处理器板，从而达到 36 个处理器的配置。总线的宽度也影响和受影响于其他的设计。例如，最接近总线的高速缓存（在这指二级高速缓存）的块大小是 128 字节，隐含着整个块能在 4 个总线时钟周期内被总线传输；由于两次传输之间死循环的影响，块越小，总线流程效率越低或设计更复杂。另外，由于总线连接大，也要求板相当的大。总线接口分

布在板边缘，占了大约 20% 的面积，这很自然地要在每个板上多放几个处理器。

让我们更仔细地看一下 Powerpath-2 总线的设计。总线共包括 329 个（位）信号：256 位数据，8 位数据奇偶校验，40 位地址，8 位命令，2 位地址 + 命令奇偶校验，8 位数据资源 ID，和 7 个其他信号。总线上事务的类型少，所有的事务处理都花费 5 个周期，这和前面讨论过的例子一致。所有的总线控制器专用芯片（ASIC）都同步地执行如下的五态自动机：仲裁、解决、写址、译码、确认。但没有事务发生时，每个总线控制器都进入两态空转自动机。这种更小的两态空转自动机使得每个新请求都能立刻得到仲裁，而不像等待五态自动机的仲裁要经过更长的时间（至少需要两个状态，如果只有一个，将会阻止不同的请求者驱动仲裁进入下面的周期）。图 6-13 给出了基本总线协议的状态机。 417

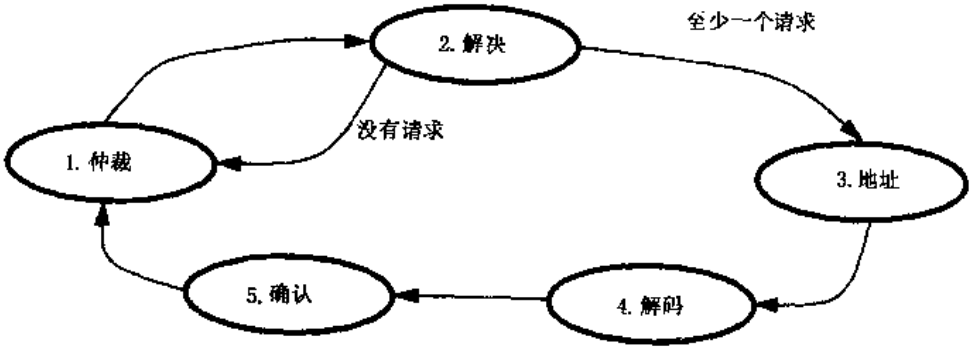


图 6-13　Powerpath-2 总线状态转换图。连到系统总线上的所有板的总线接口都按图所示同步地进行五态循环，这也是所有总线上的地址和数据事务的周期。但当总线进入空转态时，它只在状态 1 和 2 间循环

由于总线是事务拆分型的设计，地址总线和数据总线必须被独立地仲裁。在仲裁周期，使用 48 位地址线 + 命令线。这些线中的低 16 位被 16 块板使用（1 位/板）用于请求数据线，中间的 16 位用于地址线仲裁（对那些既要求地址线也要求数据线的事务，相关的位都被置高），高 16 位用于紧迫的或高优先级的请求。紧迫的请求要防止挨饿；例如，当一个处理器等待存取总线时超时。通过设置请求的优先程度，设计者可以灵活地考虑问题，使得对一些请求的服务因为性能因素可以比其他请求优先（例如读比写优先），但又肯定不会使任何请求者挨饿。

图 6-14 是图 6-8 的扩展，给出了过程的各周期（包括被驱动的各种总线）和它们的语义。在仲裁周期的末尾，所有的总线接口控制器捕捉到地址线 + 命令线的 48 位的状态，从而获悉了所有的总线请求。使用分布式体系结构方案，每个控制器能看到所有的总线请求，并在“消解”周期独立地处理同一个获胜者。虽然分布式体系结构消耗了更多的 ASIC 门资源，但它节省了集中控制器把获胜者经总线授权线传达给每一位的等待时间。 418

在“写址”周期，地址总线的获胜者用相关信息来驱动地址总线和命令总线。同时，数据总线的获胜者根据响应来驱动数据资源 ID 线。（数据资源 ID 是一个 3 位的全局标签，在读请求刚生成时，被赋予此标签。标签的使用请见 6.4.2 节）。

在“译码”周期，没有信号在地址总线上被驱动。每个总线接口槽各自决定如何对事务做出响应。例如，如果事务是回写，而当前内存系统没有足够的缓冲资源来接纳数据，那么在下个周期这个过程就会被否定回答（或拒绝），以待将来重试。另外，所有的接口槽都有准备提供适当的高速缓存一致性信息。

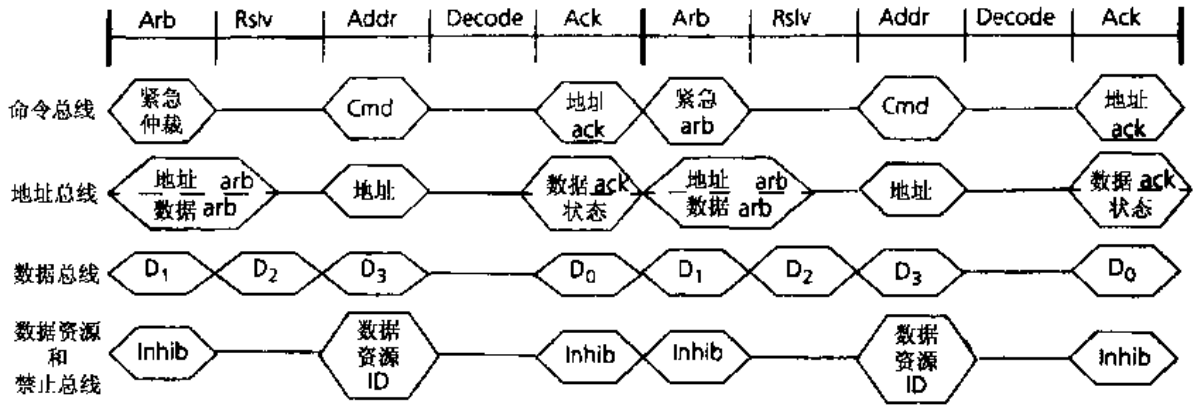


图 6-14　Powerpath-2 总线时序图。在仲裁周期，48 位地址总线 + 命令总线传达了来自 16 条总线槽的关于数据事务、地址事务和紧迫事务的请求。每个总线接口用一个公共的算法独立地决定仲裁的结果。就地址请求而言，地址 + 命令在“写址”周期被传送，请求可能会在“确认”周期被“否定回答”。类似地，就数据请求而言，相关标签（即数据资源 ID）会在“写址”周期被传送，请求可能会在“确认”周期被“否定回答”或数据在接着的 D0 ~ D3 周期（D0 即“确认”周期）被传送

在“确认”周期，每个总线接口槽都对数据事务或地址事务做出响应。48 位地址总线 + 命令总线如下使用：高 16 线表明响应槽上的设备是否因为缓存空间不足而拒绝地址总线事务，类似地，中间的 16 线针对数据总线事务。最低 16 线表明在数据总线上传输的块在高速缓存中的状态（在或不在）。这些线有助于决定将被载到请求的处理器中的数据块的状态
[419] （如排他或共享）。最后，假如某一个处理器通过这一周期没有完成侦听，它就会申明使用相关的禁止线（数据资源 ID 线在“接受”周期和“仲裁”周期加倍为抑制线），它会持续地声明直到它完成侦听。如果侦听表明了一个“干净”的高速缓存块，侦听节点就会释放禁止线，并允许请求节点接受内存响应。如果侦听节点表明了一个“脏”的高速缓存块，则节点重新仲裁数据总线，提供数据的最新版本，然后才释放禁止总线。

对于数据总线事务，一旦一条槽成为主槽，128 字节的高速缓存块数据将会在连续的 4 个周期内通过 256 位宽的数据通路。这 4 个周期序列起始于“确认”周期，终止于下一个循环的“地址”周期。由于 256 位宽的数据通路只用了 5 个周期中的 4 个，因此数据线的最高使用率为 80%。尽管如此，从某种意义上说，这是一种最好的办法，因为 Powerpath-2 总线使用的信号发送技术要求在不同的控制器驱动总线之间，要给与一个周期的回转时间。

6.5.2　SGI 处理器和内存子系统

在这个体系结构中，每个板有多个处理器。为了减少总线接口的开销，许多总线接口芯片被处理器间共享。图 6-15 给出了处理器板的高层结构。

处理器板使用 3 个不同类型的芯片来与总线接口，并支持高速缓存一致性。有一个单一的 A-芯片作为处理器与地址线的接口。它包含了用于分布式体系结构的逻辑仲裁，8 个条目的请求表用于存储当前总线上待处理的事务（见 6.4 节）以及其他的控制逻辑用来决定事务何时在总线上发布，怎样响应它们。它把总线上观察到的请求传到 CC-芯片（每个处理器一个），CC-芯片使用一个复制的标签集来决定内存块在本地高速缓存器中的存在性，并把它传回给 A-芯片。所有来自处理器中的请求流经 CC-芯片到达 A-芯片，然后由 A-芯片把它放到总线上。有 4 个位片 D-芯片与 256 位宽的数据总线接口，它们非常简单，并在处理器间共享；

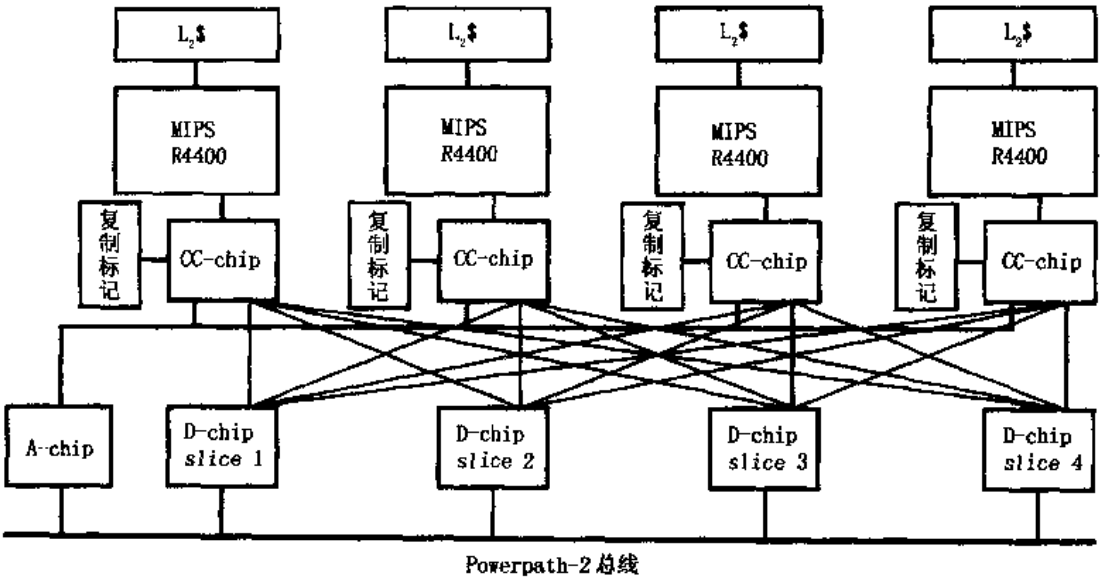


图 6-15　SGI Challenge 处理板的组织和芯片分割。为了支持 36 个处理器而使用较少的总线槽，每块板安排了 4 个处理器。为了保持一致性和与总线接口，每个处理器都有一个高速缓存一致性（CC）芯片，一个共享的 A-芯片来跟踪来自/去往所有芯片的请求以及和地址总线的接口，4 个共享的位片 D-芯片与 256 位宽数据线接口

它们提供有限的缓冲能力，并简单地把数据在和每个处理器（高速缓存）相联系的总线和处理器的 CC-芯片间传送。

Challenge 的主存子系统使用高速缓存把地址扇出到 576 位宽的内部 DRAM 总线，576 位包括 512 位的数据，64 位的错误校验码（ECC），实现一位纠错和两位检错。快速页模式的存取使得 128 字节的高速缓存块在两个内存周期被读入，数据缓冲器把响应传入 256 位宽的系统数据总线。当地址在总线上出现并过了 12 个总线周期（大约 250 ns）后，响应数据出现在数据总线上。一个单一的内存板能容纳 2 GB 的内存，支持一个两通路的交叉存取存储器系统，它的饱和系统总线带宽能达到 1.2 GBps。

假定主存系统占用的原始等待时间粗略计为 250 ns，我们不妨来看看二级缓存扑空时处理器的整个等待时间。在 Challenge 系统中，这个数字接近 1μs。使请求第一次出现在总线上花了大约 300 ns，其间包括处理器认识到一级缓存扑空、二级缓存扑空，并把请求经过 CC-芯片传到 A-芯片。完整的高速缓存数据块穿过总线到达 D-芯片另外花了大约 400 ns，其间包括 3 个总线周期等待请求过程进入“写址”阶段，12 个总线周期（大约 250 ns）存取主存，以及 5 个总线周期等待数据事务在总线上发送数据。最后，再经过 300 ns，数据流经 D-芯片、CC-芯片、64 位接口进入处理器芯片（对 128 个字节的高速缓存块 16 个周期）；在这儿，数据被载入主高速缓存，然后重启处理器流水线[⊖]。

为了保持高速缓存一致性，SGI Challenge 缺省使用 Illinois MESI 协议，它也支持更新事务。高速缓存一致性协议和事务拆分型总线的相互作用见 6.4 节中的描述。

420
|
421

⊖ 注意，在往返路径上，内存请求都穿过非同步的边界，从而增加了双倍的同步器延迟，在每个方向上大约为 30 ns。非耦合的优点就是处理器和系统总线能运行不同的时钟频率，从而允许引入更高时钟频率的处理器而保持总线时钟频率。当然，这种开销就属于额外开销。

⊖ 更新的处理器，如 MIPS R10000，允许处理器在收到所需的关键字后，就能重起流程，而不用等待完整的高速缓存块全部到达。这种关键字重起机制降低了扑空延迟。

6.5.3 SGI I/O子系统

为了支持多处理器提供的强大计算能力，在提供匹配的I/O能力时必须格外的小心。SGI Challenge提供了可扩放的I/O性能，它允许多个I/O卡插在系统总线上，每个卡提供一个320 MBps的本地HIO I/O总线。多种不同的ASIC作为I/O总线和标准一致设备（如Ethernet、VME、SCSI、HPPI），非标准一致设备（如SGI Graphics）的接口。

图6-16给出了SGI Challenge的PowerChannel-2 I/O子系统的高层结构。总线是一种64位宽的多选地址/数据总线，与系统总线运行相同的时钟频率。它支持读事务的分离，每个设备可以有4个待完成事务。和系统总线不同，它使用集中式仲裁，节省等待时间。然而，仲裁是流水进行的，使总线带宽不致被浪费。由于HIO总线支持多个不同的事务长度（它不要求每个事务都处理一个完整的高速缓存块的数据），因此事务在请求时，要表明它们的长度，仲裁器使用此信息来保证总线得到更高效的利用。接到较窄HIO总线的ASIC比直接接口到较宽的系统总线上的ASIC相对便宜。另外需要接口到系统总线的公共机制通过这种方式被若干ASIC电路共享。

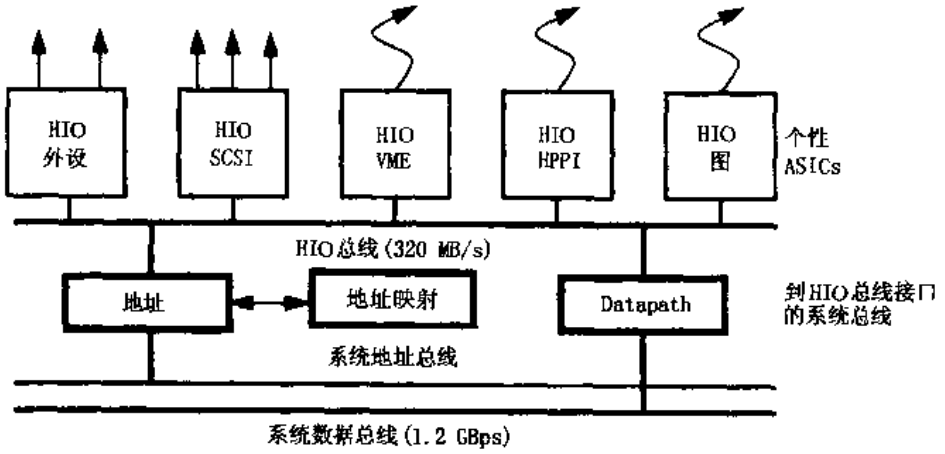


图6-16　SGI Challenge PowerChannel-2 I/O子系统的高层组织结构。每个I/O板提供了Powerpath-2系统总线和内部的具有最高带宽为320 MBps的64位宽的HIO I/O总线的接口。较窄的HIO总线降低了与它接口的开销．支持大量的专门ASIC，它相应的支持标准总线和外部设备

HIO接口芯片可以请求对DMA的读/写，使用全40位系统总线传输自/至系统内存的任何位置，使用系统接口中的映射资源提出对地址翻译的请求，要求中断处理器或响应处理器的I/O（PIO）读。系统总线提供DMA读响应，对I/O设备提供地址翻译的结果，并传送到要读它们的PIO。

在系统的其余部分（处理器板和主内存板）看来，在I/O板上的系统总线接口是一个清晰的接口；它工作起来就像一个处理器板。因而，当一个DMA读请求通过系统总线接口到
422 达系统总线时，它就变成了一个Powerpath-2读，就像系统总线可能做的一样。类似地，当一个完全的高速缓存块DMA写发生时，它在总线上变成了一个特定的写块事务，使所有处理器高速缓存中的拷贝都变得无效（不仅仅更新内存）。我们需要一个特别的事务来进行处理，因为即使此块在一个处理器高速缓存中是脏块，我们也不想在这种情况下把它写回。

为了支持部分缓存块DMA写，必须小心地把数据一致地合并到内存中。为了支持这些

部分块 DMA 写，系统总线接口包含一个全相联的、含有 4 个存储块的高速缓存，来侦听 Powerpath-2 系统总线。高速缓存块的状态只有两种：作废或被修改。刚开始发出部分块 DMA 写时，此块被放进这个特殊的缓存，置以被修改状态，并使所有处理器高速缓存中的块拷贝都变得无效。以后的部分块 DMA 写如果命中此缓存，就不必再到系统总线，因而增强了系统总线的工作效率。在以下情况，修改的块状态将置为作废，并进入系统总线：1）有任何系统总线事务存取此块；2）有其他部分块 DMA 写过程将使此块替换出这个缓存；3）有任何 HIO 总线读事务存取此块。虽然块 DMA 读也可能会使用这 4 块高速缓存，但设计者感到部分块 DMA 读比较稀少，从优化中得到的收益很小。

在系统总线接口中的地址映射 RAM 为 I/O 设备提供了通常的地址翻译来访问主存。例如，经常要把小地址空间（如 VME-24 或 VME-32）映射到 Powerpath-2 总线的 40 位物理地址空间。有两种映射：一级映射和二级映射。一级映射仅仅返回地址映射 RAM 的 8 K 条目中的一条，每一条目对应物理内存中的 2 MB。在二级映射方案中，映射条目指向主存的页表。每一 4 K 页在二级表中有自己的表目，于是虚页能被正确地映射到物理页。注意 PIO 请求（来自处理器）到 I/O 设备有一个相似的翻译问题，它不使用地址映射 RAM 而直接通过专门的 ASIC 接口芯片来处理。

最后我们来看流控。所有从 I/O 接口到 Powerpath-2 系统总线的请求都是隐示地流控；即 423
HIO 接口如果没有给响应保留缓冲空间，就不会向系统总线发出读请求。类似地，HIO 仲裁器不会把 HIO 总线交给请求者，除非系统接口有空间来接纳事务。例外的是，PIO 能不经请求地从处理器到达 I/O 设备，它们需要进行显示地流控。

在 Challenge 系统中使用显示流控方案目的是使 PIO 看起来经过了请求才到达 HIO 接口 ASIC。在复位后，HIO 接口芯片（如 HIO-VME，HIO-HPPI）使用特定的称作 IncPIO 的请求将它们可得到的 PIO 缓冲空间传给系统总线接口。系统总线接口为每一个 HIO 设备维护一个独立的计数器。每次当一个 PIO 被传到一个特定的设备时，相关的计数减少。每次设备重试一个 PIO，它就发另一个 IncPIO 请求来增加计数器。如果系统总线接口收到一个关于设备的 PIO 请求却没有足够的可利用缓冲空间，它就会在系统总线上拒绝（NACK）这个请求，于是这个请求就必须在以后重新提出。

6.5.4 SGI Challenge 内存系统性能

各种级别的 SGI Challenge 内存系统的存取时间能用第 4 章介绍的简易的读微基准测试程序来得到。微基准测试程序方案测试出在一特定的跨距内，读给定大小的一组元素花费的平均存取时间。图 6-17 给出了在一定范围的大小和跨距内读操作的时间。每条曲线给出了对一定的大小受跨距影响的平均时间。小于 32 KB 的数组完全适用于一级缓存。二级缓存的访问大约用 75 ns，在 16 字节跨距处的拐点表明二级缓存和一级缓存之间的转变大小是 16 字节。第二个凸起部分给出了 TLB 扑空的额外代价大约是 140 ns，并揭示了页的大小大约是 8 KB。（你能想出为什么随着跨距的进一步增大每次扑空的时间反而回落了呢?）从 2 MB 数组开始，在 1 MB 的二级缓存中发生访问扑空，我们看到在二级缓存控制器、Powerpath 总线控制器和 DRAM 存取的共同作用下，一次访问时间大约为 1 150 ns。与前面的讨论相符，实现从请求到回应共 13 个周期的最小的总线协议使得这个时间略低于 300 ns。TLB 扑空在 1 150 ns 中增加了大约 200 ns。一个简易的乒乓微基准测试程序（在此方案中，有一对节点

都受某一标记控制而不断运转，当轮到自己时，就设置标记把信号发给对方）给出了6.2 μs的往返时间。

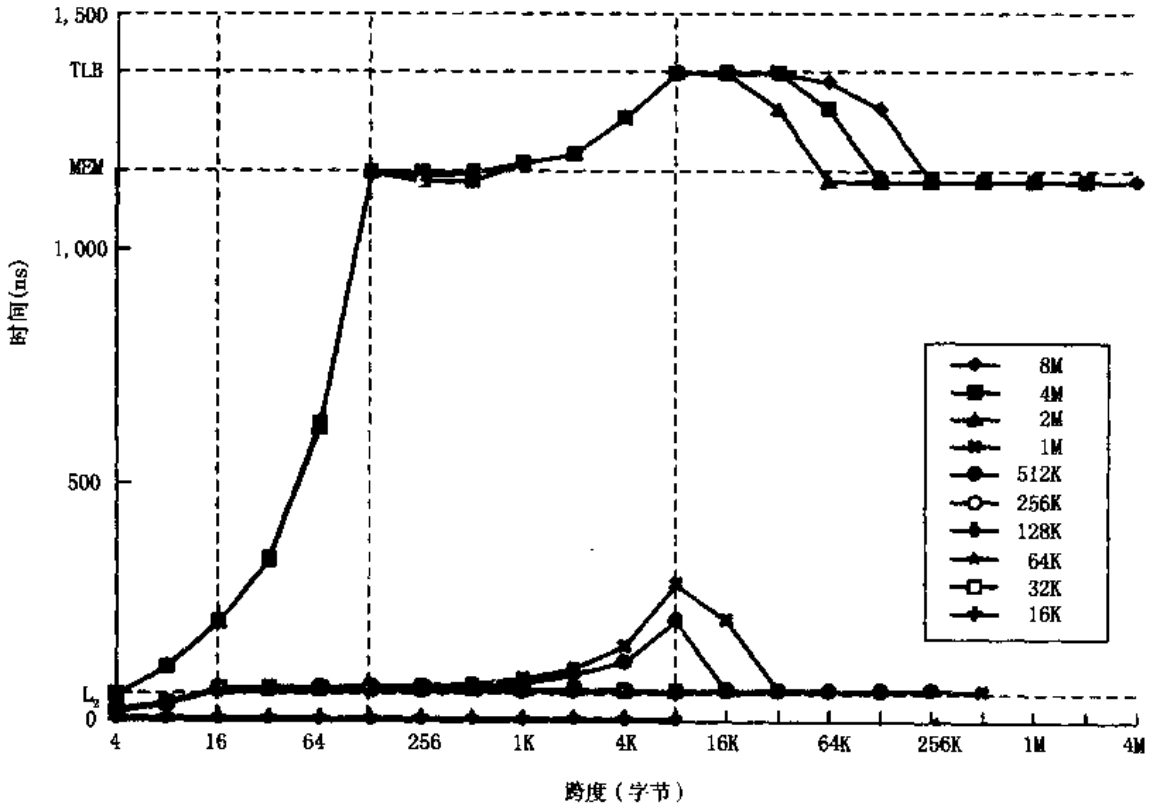


图 6-17　SGI Challenge 读微基准测试程序的结果。每条曲线都针对图表符号中所示的一组有相同尺寸的元素。数组尺寸从 32 K 到 256 K 的数据点非常接近，很不容易区分

6.5.5　Sun Gigaplane 系统总线

Sun Gigaplane 也是非多选的事务拆分型总线，有 256 位数据线，41 位物理地址线，但采用 83.5 MHz 的时钟频率。它是一种中心平面设计，一条总线贯穿和接口汇集，允许双面插板，而不是单面的底板。总线全长 18 英寸，故每一边能插入 8 个板，且在每两个板间留有两英寸的散热空间，连接口间有 1 英寸的空间。它与 Powerpath-2 总线最大的不同在于，它支持多达 112 个等待处理的事务。每个板高达 7 个，因此它是为能支持多个等待事务的设备而设计的，例如免锁定的高速缓存。这种电气上和机制上的设计允许处理模块和 I/O 模块的热插拔。

总线共包括 388 个信号：256 个数据、32 个校验码、43 个地址（带有奇偶校验）、7 个 ID 标记、18 个仲裁以及许多的配置信号。电气上的设计实现了数据传送间没有死循环的回转。它把重点放在了实现尽可能少的操作延迟上（如图 6-18 所示），它的协议与 SGI Challenge 有很大的不同。一个著名的基于冲突的推测仲裁技术被用于减少总线仲裁的开销。当请求者对地址总线进行仲裁时，如果总线自从前一个周期后尚未被调度使用，那么在仲裁周期，它就试探性地把自己的请求驱动到地址总线上。如果在那个周期没有其他的请求者，它就赢得了仲裁并将地址送出去，于是它继续余下的事务。如果发生了请求冲突，赢得仲裁的请求者只简单地把地址在下个周期重新驱动一遍，和常规的仲裁一样。

424
~
425

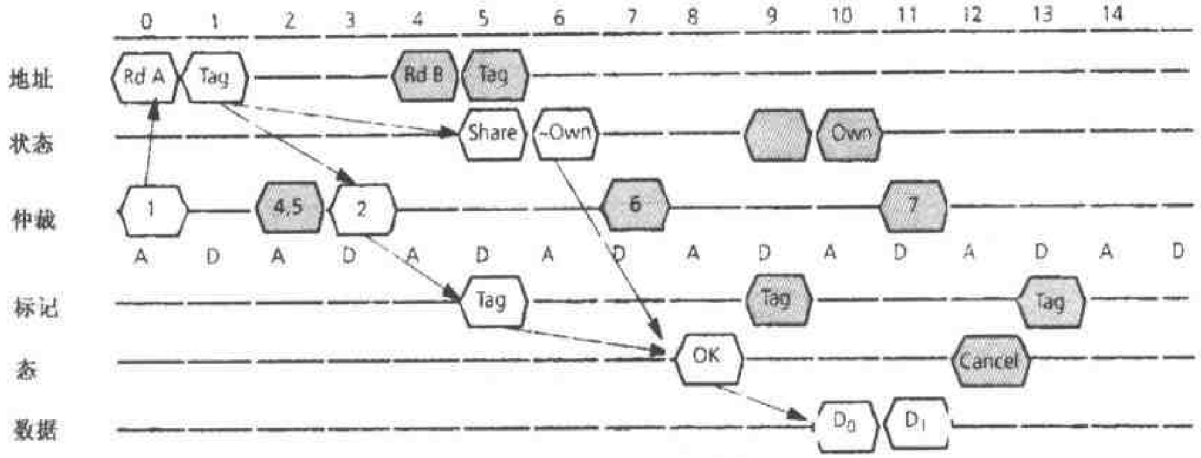


图 6-18　Sun Gigaplane 对采用快速地址仲裁的 BusRd 的信号时序。正如所示，在流水线时隙上，有两个 BusRd 事务，一个不带阴影，另一个有阴影。水平粗线显示了总线的不同组件，垂直点线划分了周期。总线仲裁组件下的“A”和“D”表明了地址仲裁周期和数据仲裁周期。箭头显示了第一个 BusRd 事务的路径。板 1 初始化了一个采用快速仲裁的读事务（地址在同一个周期被成功地驱动），然后被主板 2 响应。侦听结果表明没有高速缓存容纳此块，于是主板把结果驱动到数据线上。对于第二个 BusRd 事务，板 4 和板 5 在地址总线仲裁期间发生冲突，板 4 获胜并初始化了一个读事务。主板 6 对数据总线进行仲裁，由于侦听结果表明有高速缓存含有此块，故取消了响应。最后板 7 上的此高速缓存以数据做出响应。板 5 的重试事务没有显示

426

有关请求的 7 位标记在紧跟着“写址”周期后的周期内被放到地址总线上（如图 6-18 所示）。侦听的状态与地址阶段相联，与数据阶段无关。在“写址”周期后，过了 5 个周期，所有的板都在状态总线上声明自己的侦听信号（共享、被拥有、被映射、忽略）。在此期间，响应内存地址（主板）的板会在“写址”周期 3 个周期后，侦听结果出来之前，请求数据总线。DRAM 访问也能被试探性地启动。当主板得到仲裁后，它必须在两个周期后给出标记总线的电平，告知所有设备处理数据正在到来。在驱动标记 3 个周期后，即驱动数据的前两个周期，主板驱动了一个状态信号，用于表明如果某个高速缓存拥有这个块（在侦听状态时得到），数据传送就会被取消。拥有者通过仲裁数据总线把数据放到总线上，驱动标记，启动数据。图 6-18 给出了另一个读过程（灰色），它在仲裁时遇到冲突，于是地址由常规的槽提供。对过程的侦听表明拥有者是一个高速缓存，故主板取消了数据传送。然后，这个变速缓存仲裁数据总线，驱动相应的数据。

像 SGI Challenge 一样，出现在地址总线上的 BusRdX 请求发出作废命令，此命令被高速缓存子系统以先进先出的方式进行处理。这样，也不必要求对完成“作废”工作的显示的认可。为了保持连续的一致性，仍有必要在允许写处理器在写之后继续从事内存操作之前获得对地址总线的仲裁。[⊖]

6.5.6　Sun 处理器和内存系统

如图 6-19 所示，在 Sun Enterprise 中，每个处理板有两个处理器（每个处理器有一个外部的二级缓存）和两个内存模块（通过交叉开关相连）。UltraSparc 内的数据线被缓冲来驱动称为 UPA（通用端口体系结构）的内部总线（具有 1.3 GBps 的内部带宽）。到内存的通路很

⊖ Sparc V9 规范弱化了同一性模型，以使处理器能使用写缓存，我们将在第 9 章更深入地讨论这个问题。

宽，读一个完全的64字节的高速缓存块只要一个内存周期或两个总线周期。地址控制器使UPA协议和Gigaplane协议相匹配，并实行高速缓存一致性协议，提供缓冲，跟踪潜在的大量的待完成的事务。它为二级缓存保有一组备份的标记，称为D-标记。为了保证高速缓存一致性，即使是从处理器到本地内存模块的访问也要经过地址控制器。

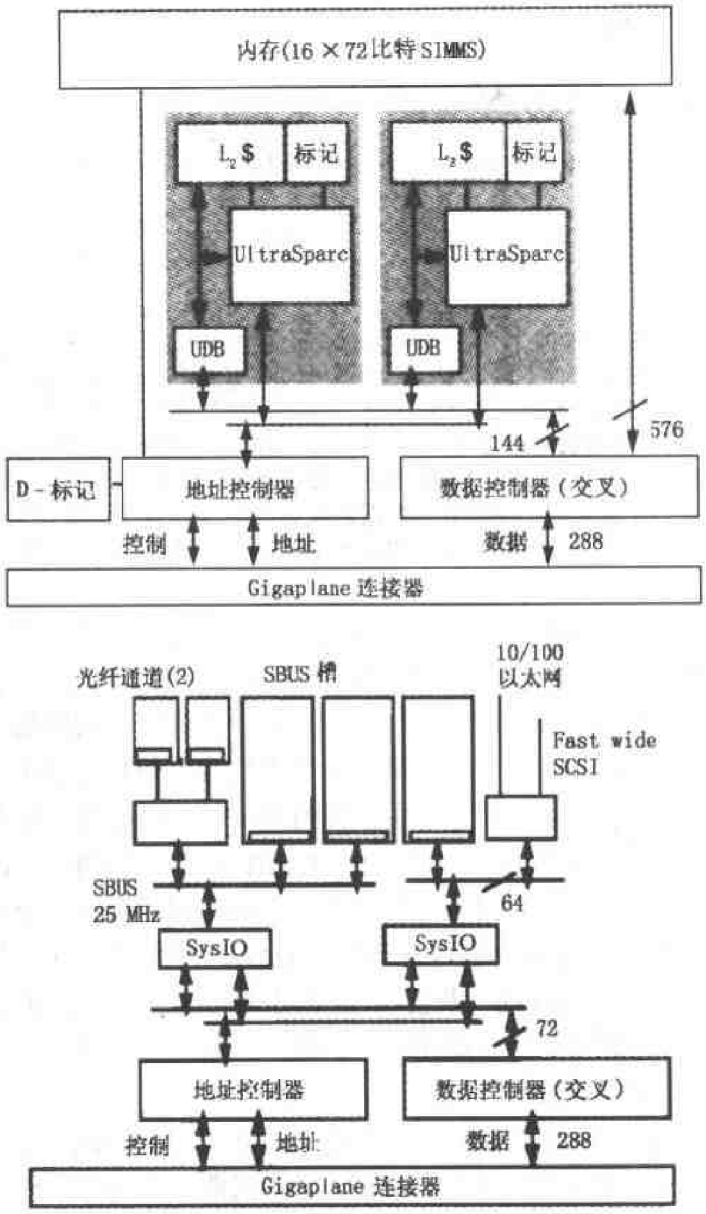


图6-19　Sun Enterprise处理板和I/O板的组织结构。处理板（上）包括两个UltraSparc模块（带有在内部总线上的二级缓存）和两个宽内存模块（通过两个ASIC与系统总线接口）。地址控制器使两个总线协议相匹配，并实现了高速缓存一致性协议。数据控制器本质上就是一个交叉开关。I/O板（下）使用两个相同的ASIC与两个I/O控制器接口。SysIO ASIC看起来就像一个遵从一致性协议的单块高速缓存。另一方面，它们支持独立的I/O总线，并与FiberChannel、Ethernet、SCSI接口

虽然UltraSparc在二级缓存里实现了5态的MOESI协议，D-标记只使用了3个状态：被拥有、共享、作废。它们基本上综合了在Gigaplane级上一致处理的状态。特别地，地址控制器需要知道二级缓存是否有一个块的拷贝以及此块是否是一个排他的拷贝。它不必知道这个块是否是干净的或脏的。例如，在进行BusRd时，如果存储块在二级缓存中的状态是如下

427
~
428

三个之一：已修改、被拥有（最近一次修改后已被发送过）、排他的（不能共享读和不能修改），块需要被送到总线上；因而，D-标记代表的仅仅是被拥有状态。这能使当 UltraSparc 把块由排他的改为已修改时，不用告知地址控制器。当发生由作废、被拥有、共享到已修改的状态转变时，需要告知 UltraSparc，以便它初始化一个总线事务。

6.5.7 Sun I/ O 子系统

一个 Enterprise I/O 板与处理板使用一样的总线接口 ASIC，但内部总线只有一半宽，并且没有内存通路。在外看来，I/O 板就像处理板一样只做高速缓存块级的事务，为了简化主系统总线的设计。SysIO ASIC 实现了一个单块高速缓存代表 I/O 设备，并遵循一致性协议。在内支持两个独立的 64 位 25 MHz 的 SBUS。其中一个支持两个专用的 FiberChannel 模块，它提供大磁盘存储阵列的冗余的、高带宽的互连。另一个支持专用以太网和快速的宽 SCSI 连接。另外，3 个 SBUS 接口卡能被插入这两条总线以支持仲裁外设，包括 622 MBps ATM 接口。I/O 带宽、与外设的连接、I/O 子系统的开销决定了 I/O 卡的数目。

6.5.8 Sun Enterprise 内存系统性能

图6-20给出了用读微基准测试得到的Sun Enterprise的各种等级的存取时间。小于或等

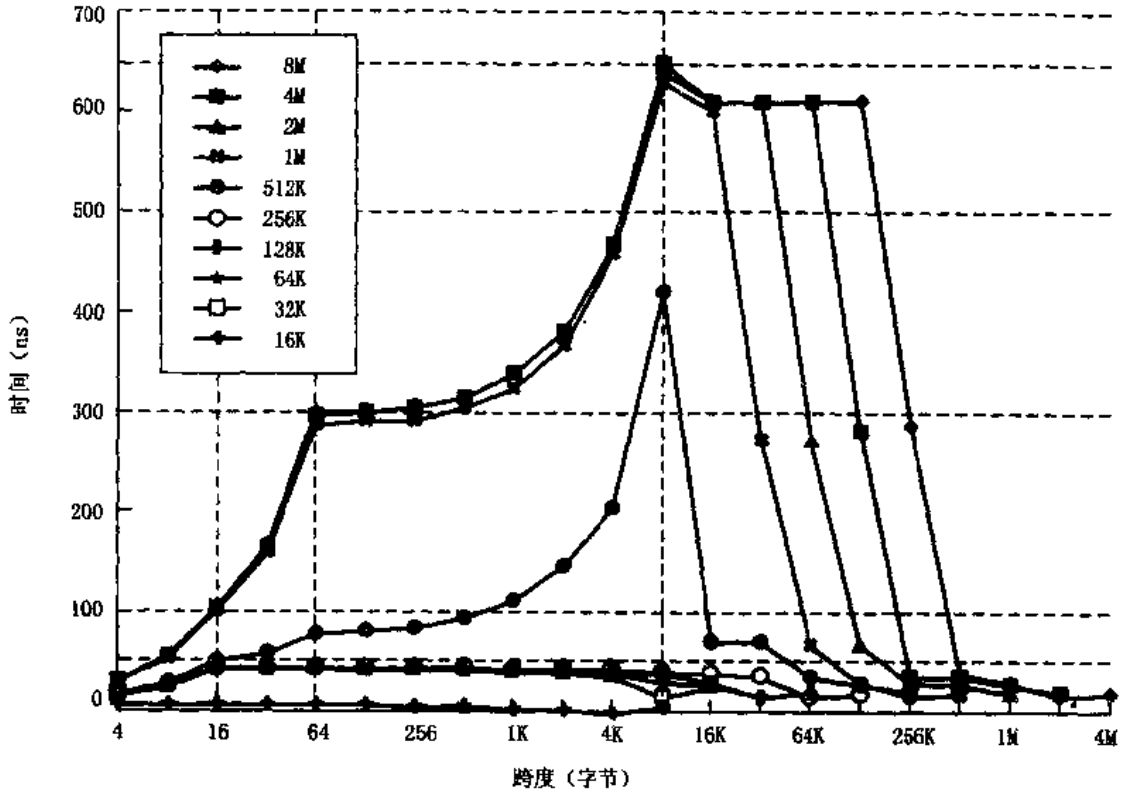


图 6-20 Sun Enterprise 在读操作微基准程序上的测试结果。每条曲线对应不同的规模，具体见图例所示

于 16 KB 的数组完全适用于一级缓存。二级缓存的存取时间大致是 40 ns，拐点表明二级缓存和一级缓存之间的转变大小是 16 字节。对于存取二级缓存未中的 1 MB 的数组，我们看到由于二级缓存控制器，总线协议和 DRAM 存取共同作用的结果，存取时间大约为 300 ns。

83.5 MHz 下共 11 个周期的最小的总线协议占用了这段时间内的 130 ns。由于机器有软件 TLB 管理器，所以 TLB 未中增加了大约 340 ns 未中的代价。一个简易的乒乓微基准测试（在此方案中，有一对节点，都受某一标记控制而不断运转，当轮到自己时，就设置标记发给对方信号）给出了 1.7 μs 的往返时间，相当于访问内存 5 次。

6.5.9 应用程序性能

现在我们已对机器和它们的微基准测试性能有了一定的理解，接下来检验一下对于并行应用程序性能的提高。本书不说明商用机器绝对的应用性能，而着重在并行性带来的性能提高。以 SGI Challenge 为例解，让我们先看看应用程序的加速比，接着是性能随规模的扩放情况。

1. 应用程序的加速比

图 6-21 给出了 6 个并行程序（在两组不同数据规模上）得到的加速比。我们可以看到，除了基数排序内核外，多数程序的加速比都不错。通过分析排序执行时间的细节，能够发现绝大部分时间都花在等待数据访问上。在排序算法的交换阶段，数据流量和一致性流量都很大，使共享总线不堪重负，所导致的访问冲突极大地破坏了性能。冲突在数据访问时间内导致了严重的负载不平衡，从而时间花在了等待全局栅障上，即使繁忙时间被很好地平衡。不幸的是，增加问题的规模不能对冲突有很大地减轻，由于通讯和计算的比以及所产生的带宽要求，使得交换问题独立于数据规模的大小（见第 4 章 4.4.1 节）。图示的基数为 256，它要得到最好的性能与两个问题大小因素有关，不仅仅是处理器的数目。Barnes-Hut、Raytrace 和 Radiosity 即使在相对小的输入规模情况下也加速得非常好。LU 也是如此，它在 16 个处理器下对较小规格问题的瓶颈主要是由于矩阵因子分解过程造成的负载不平衡。最后，Ocean 的瓶颈既是由高的通讯和计算比，也由一些分区具有较少的相邻者造成的不平衡产生的，两个问题可以由运行大的数据集来减轻。

429
|
430

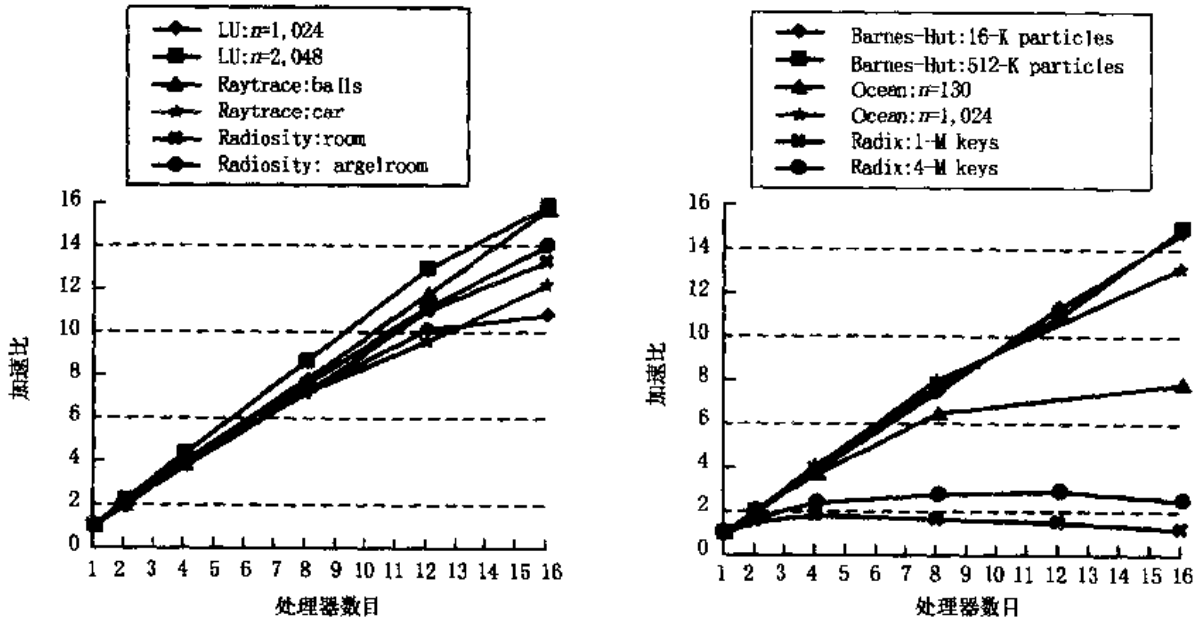


图 6-21　SGI Challenge 上 6 个并行应用程序的加速比。成组的 LU 因子分解的块尺寸是 32 x 32

2．性能随规模的扩放

现在让我们考察一下规模扩放对一些应用程序的影响。根据第 4 章的讨论，我们看看不同的规模扩放模型下得到的加速比，以及完成的工作和使用的数据集大小的变化。图 6-22 给出了对于 Barnes-Hut 和 Ocean 应用程序的结果。其中 Naive TC（时间受限）或 Naive MC（内存受限）扩放指的是仅仅改变决定数据集大小的参数（星体或网格点的数目 n），而不改

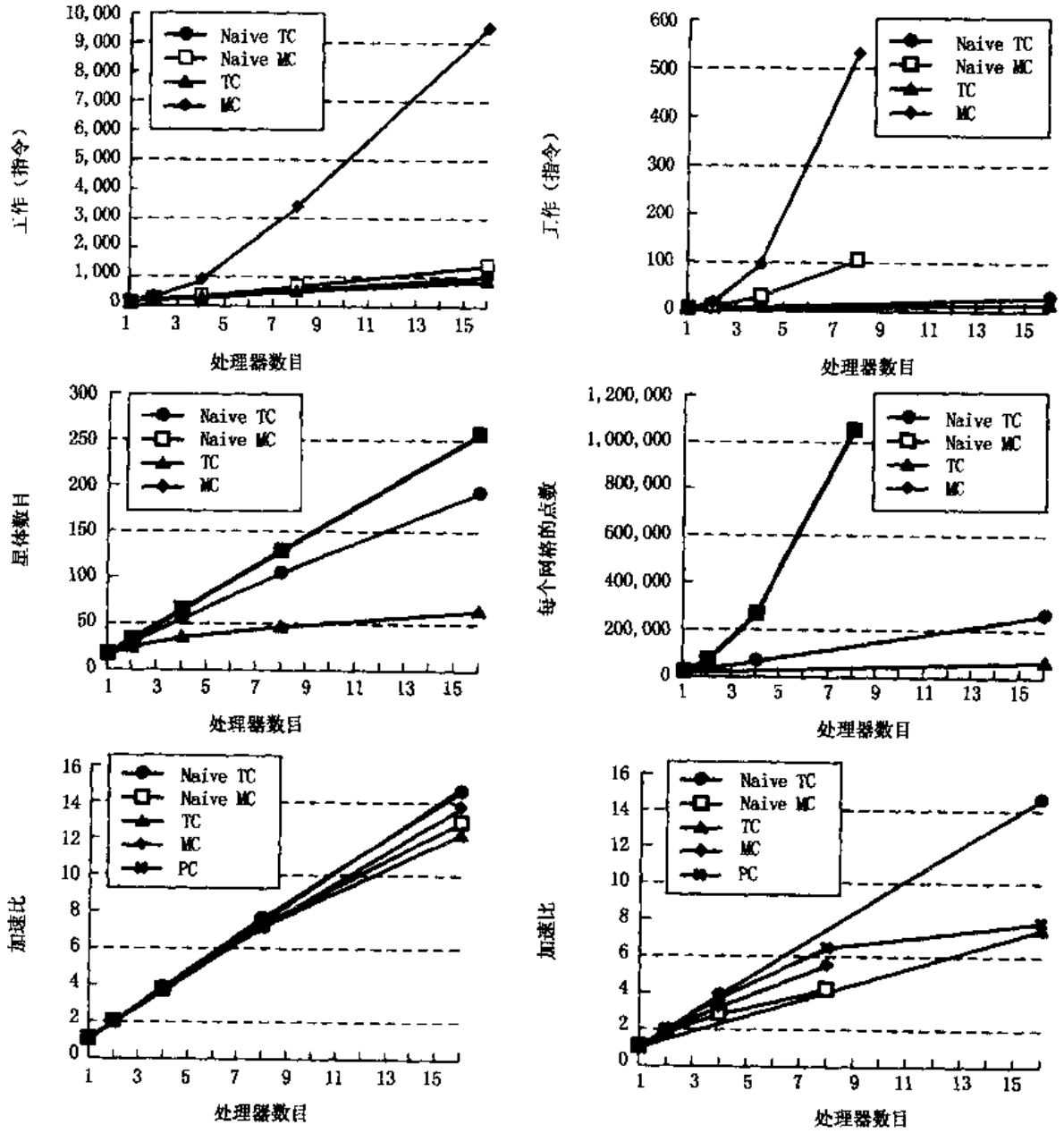


图 6-22　SGI Challenge 上 Barnes-Hut（左）和 Ocean（右）应用程序的缩放结果。图给出了在不同的扩放模型下完成的工作、数据集的大小（测量主体或网格点的数目）、加速比。PC、TC 和 MC 分别指的是问题约束、时间约束、内存约束的缩放。对 Barnes-Hut 来说，问题尺寸的基准是 16 K 星体；对 Ocean 来说是 130 × 130 网格。最上的一组图显示出对于这两个应用程序在 Realistic MC 缩放下需要解决的工作都增长的非常快。中间的一组图显示出在 MC 下或 Naive TC 下，数据集大小的增长比在 realistic TC 下增长的更快。缩放模型在加速比方面对于 Ocean 的影响比 Barnes-Hut 大得多，主要是因为在 Ocean 里通信和计算比更强烈地依靠数据集大小和处理器数目

变其他应用程序参数（精确性或时段数目）。很明显，在 Realistic MC 扩放下，对于这两个应
431 用程序要完成的工作都比线性地改变处理器的数目快得多，于是并行执行时间增长得非常
~ 快。能在 TC 扩放下被模拟的星体或网格点的数目比在 MC 下增长慢得多，也比 Naive TC 扩
432 放下的增长慢得多，它仅仅只是一个应用程序参数的改变。扩放其他应用程序参数使得完成的工作和执行时间增长，使得增加 n 更加困难。

在不同的扩放模型下加速比的测量和第 4 章描述的一样。现考虑 Barnes-Hut 星系的模拟，它的加速比对于各种扩放模型下的机器尺寸都很好。通过检验主要的性能因素可以解释差别。在引力计算方面的通信和计算比主要依靠星体的数目。另外一个影响性能的重要因素为两个量的比，即在引力计算阶段的工作量和在建立计算树阶段的工作量的比，前者容易通过并行得到好的加速比，后者难以并行。这个比率倾向于在作用力计算方面有更高的精度，即更小的 θ。然而，更小的 θ（从更小地程度上说，是更大的 n）意味着增长每个处理器工作集的尺寸（Singh，Hennessy，and Gupta 1993）。重要的工作集即使在扩放下也可能继续适用大的二级缓存，但是在一个单处理器上，改变 θ 的扩放后的问题，相对于基准问题而言，也许会降低一级缓存的性能。这些因素就可以解释为什么 Naive TC 比 Realistic TC 加速得更好：工作集的行为越好，通信和计算比就越好（因为当 θ 和 Δt 不变时，n 增长得更快）。

Ocean 的加速比在不同模型下很不相同。在这儿，主要的控制因素还是通信和计算比、工作集大小、在不同情况下的时间花费等。然而，所有的效果更强烈地依靠网格的尺寸（与处理器数目有关）。在 MC 扩放下，通信和计算比不随使用的处理器数目而变化，于是我们希望得到更好的加速比。但随着处理器的变化，出现了两个效果。第一，随着处理器的网格划分在地址空间上的距离增大，跨网格的冲突扑空增加。第二，在求解器中，更多的时间花在多重网格的较高层次上，降低了并行性能。当精确性和时段间隔被细化后，后一个影响被减轻了（至少有益于并行加速比），于是实际的 MC 比简单 MC 稍好一点。在简单 TC 下，网格尺寸的增长速度不足以引起主要的冲突问题，但足以使通信和计算比不发生明显的增长，故加速比挺好。实际的 TC 使网格尺寸增长较慢，从而加大了通信和计算比，导致较低的加速比。很明显，许多效果在决定扩放下的并行性能时扮演了重要的角色，并且对某个应用程序最合适的扩放模型的选择影响对机器的评估。

6.6 高速缓存一致性的扩充

在这两章里描述的基于侦听获得缓存一致性的技术可以在许多方向上扩充。这一节考察
433 几个重要的情况：利用共享缓存的向下扩展、带有虚拟索引缓存和 TLB 的功能性扩放和利用非总线互连的向上扩放。

6.6.1 共享缓存的设计

将处理器组织在一起，让它们共享存储层次结构的一个层次（例如一级或二级缓存），对于共享存储多处理器来说是一个有潜在吸引力的考虑。尤其是考虑到封装时，若能将多个处理器放在同一个芯片上，这种想法会更有价值。和处理器在各自存储层次中有自己的缓存相比，将处理器集中起来有多种潜在的好处。这些好处，如同后面将要讨论的弊病一样，当共享任何一层存储时都会体现出来，但在处理器间共享的一级缓存的情形表现得最突出。在一层中的一组处理器间共享一个高速缓存的好处归纳为如下几点：

- 它消除了在这一级做缓存一致性协议的必要。特别是，如果一级缓存由所有处理器共享，那么就没有多份缓存块的存在，从而也就根本没有了一致性问题。
- 它降低了处理器组内部通信的时延。处理器之间通信的时延和它们通信时相遇在存储器层次的哪一层有关。当共享一级缓存时，通信时延可能只要 2 ~ 10 个处理器时钟周期，而在主存相遇的情形，时延要大许多倍（见关于 Challenge 和 Enterprise 的案例分析）。降低了时延就可能使不同处理器上执行的任务能在更细的粒度上共享数据。
- 一旦处理器在访问一个数据时发生扑空，而后将它带到共享缓存来，在同一组中的其他处理器当需要该数据时可能会发现它已经在缓存了，于是不会形成新的扑空。这称为跨处理器的数据预取。对于私有缓存来说，每个处理器会引发分别的扑空。扑空的数量减少了，也就降低了对下级存储和互连带宽的要求。
- 它允许更有效地利用较大的缓存块。即使一个组中的不同处理器访问一个缓存块的不同字，空间局部性也被开发出来。除此以外，由于在这一层次没有一致性协议，也就没有了伪共享问题。例如，考虑两个处理器 P_1，P_2 相继对一个大数组的各个元素依次交替进行写操作，考虑一级共享缓存和私有一级缓存的差别。
- 在一个组中的工作集（代码或数据）可能重叠很多，如果每个共享缓存都要保持其处理器的整个工作集，这就允许共享缓存的大小比私有缓存之和小。缓存容量的减小对单芯片的多处理器是特别有意义的，因为在这样的场合硅面积是一个重要的限制。 434
- 它提高了缓存硬件的利用率。共享缓存不会因为一个处理器停滞就闲起来，而是可以为组中的其他处理器服务。
- 成组的做法和层次封装技术适相应得很好（机柜，板，多芯片模块和芯片），且使我们能够有效地利用新出现的封装技术来获得更高的计算密度（每单元面积的计算能力）。

共享一级缓存时，处理器如图6-23那样通过一个交换机连接到一个共享缓存上。这个交

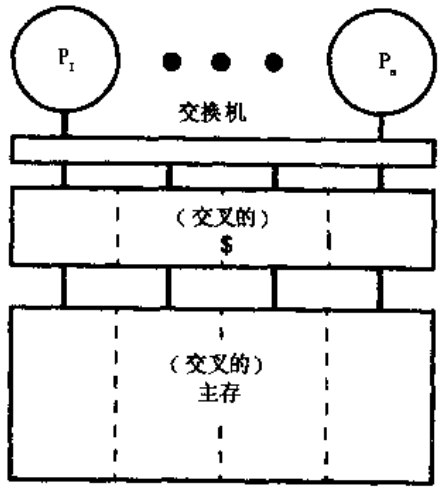


图 6-23 多处理器共享一级缓存的一般性结构。互连设置在处理器和一级缓存之间。在这种设计中，共享的缓存直接由主存支持，缓存和主存都可以分别交叉来提供高带宽。当然，由于共享缓存或交换机带宽的限制，这种设计只适合于较少的处理器数。另外较大的交换机会有较大的时延

换机可能是一个总线，但更可能是一个交叉开关，让从不同处理器来的缓存访问能够并行处理。类似地，为了支持多处理器带来的带宽要求，缓存和主存都按交叉方案组织。这种共享缓存系统的一个早期例子是设计于 20 世纪 80 年代初期的 Alliant FX-8 机。该系统可以最多含有 8 个特定的处理器，每个处理器是 68020 指令集的一个流水实现，并增强有向量指令，时钟周期 170 ns。所有 8 个处理器用一个交叉开关连接到一个 512 K 的 4 路交叉缓存。这个缓

存的块为 32 字节，回写型，直接映像，免锁定，允许每个处理器有两个待完成扑空。缓存到处理器带宽是每指令周期 8 个 64 位的字。

早期还有一种共享缓存的不同用法，出现在 Encore Multimax 中，和 FX-8 是同时代的产品。Multimax 是一种侦听缓存一致性多处理器，但每个缓存支持两个处理器（在一对处理器之间不需要一致性）。当时 Encore 的动机是要降低侦听硬件的费用，努力在非常慢的多 CPI 的处理器条件下提高缓存的利用率。

435 当前，人们在单片多处理器的背景下研究一级缓存的共享问题，4～8 个处理器共享一个片载一级缓存。这样的系统本身就是一个多处理器，也可以用作更大系统的基本模块，这样的大系统要在单片共享缓存组之间维持一致性。随着工艺的进步，一个芯片上的三极管数以数千万或亿计，这种方式越来越具有吸引力。由于在一个芯片内的处理器之间通信和同步开销是相当低的，用这样的芯片做成的工作站将能够对细粒度或粗粒度并行性都给出很高的性能。问题是和用更多的三极管做出更复杂的处理器相比，这种做法是否更加有效。

不尽人意的是，共享缓存也有若干弱点，带来若干挑战：

- 共享缓存需要满足来自多个处理器的带宽要求，这限制了组的大小。对一级缓存来说这个问题特别突出，因此只可能有很少的处理器。如何能提供所需的带宽是单片多处理器系统设计中最大的挑战。
- 由于互连的关系，对共享缓存的命中时延通常要高于私有缓存。对共享一级缓存来说，在处理器和缓存之间放一个交换机意味着机器周期拖长，或者在处理器流水线中装入指令中要增加额外的延迟时隙。前者造成的速度慢是显然的。虽然编译器有可能在装入延迟时隙中调度一些不相关的指令，成功与否取决于具体的应用程序。特别是对于那些没有许多指令级并行性的程序，减速是不可避免的。共享缓存引起的竞争使本已增加的命中时延更加严重，相应地，扑空时延也会由于共享增加。
- 鉴于前面的原因，设计一个有效的共享缓存要比设计私有缓存复杂许多。
- 虽然一个共享的缓存不需要如相应私有缓存之和那么大，但和单个私有缓存相比，它仍然要大得多，因此就会慢得多。对一级缓存来说，这就可能会加长机器时钟周期，或者要多个处理器周期才能完成缓存访问。
- 重叠工作集（或者是建设性干扰）的反面是共享缓存的性能受到伤害，由于跨越源于不同处理器的引用流的缓存冲突（破坏性干扰）。当一个共享缓存多处理器用来执行没有数据共享的负载（例如，并行编译或者数据库和事务处理工作负载），缓存中在不同处理器所需的数据集之间的干扰可能严重伤害性能。在科学计算中，性能是第一位的，许多程序试图管理它们对每个处理器缓存的使用，使得它们访问的数组在缓存中不发生干扰。所有这些程序员或编译的努力在共享缓存系统中很容易就白费了。同私有缓存相比，共享缓存可能要求更高的相联度，这也可能增加它们的访问时间。 436
- 最后，在目前情况来看，共享一级缓存不符合当前用商品微处理器技术建造成本效益合算的并行机器的趋势。

由于许多微处理器已经对第一级缓存提供侦听支持，一种有吸引力的做法可能是让处理器有私有的一级缓存，而在第二级缓存共享。这种做法“软化”了共享一级缓存的优缺点，可能总体上是一种不错的折中。这个共享缓存可能比较大，以减少破坏性干扰。在实践中，

封装方面的考虑对于共享缓存的决定也有很大影响。

6.6.2　虚拟标引缓存的一致性

回顾在单处理器系统结构中在物理的和虚拟的索引缓存之间的权衡，即，用物理地址还是虚拟地址对缓存索引。对于物理索引的一级缓存来说，允许缓存索引和地址转换并行进行，就要求缓存有很小或者很高的相联度。这保证了如果使用页染色方式，那些在转换下不改变的位——$\log_2$（Page_size）位或者更少些——足以索引缓存（Hennessy and Patterson 1996）。随着片载一级缓存变大，虚拟标引缓存变得越来越有吸引力。然而，这也有它们自己的问题。首先，不同的处理器可能用同样的虚拟地址来引用不同地址空间中不相关的数据。这可以通过在切换上下文时清除整个缓存来处理，或者通过使地址空间标识符（ASID）标记和缓存块相联，除虚拟地址标记外。对缓存一致性更严重的问题是同义：不同的虚拟页，来自相同或不同的进程，为了共享指向同一个物理页。对于虚拟地址缓存来说，相同的（共享的）物理存储块能够读到两个不同的缓存块、不同的索引。如我们所知，这对单处理器是一个问题，它也扩展到多处理器的缓存一致性：如果一个处理器写一个存储块，用一个虚拟地址别名表示，另一个读他，用不同别名的表示，那么如果简单地将虚拟地址放到总线上，侦听它们，对这个共享物理页的写就不会为后来的处理器可见。将虚拟地址放到总线上还有另外一个问题：它要求 I/O 设备和存储器做从虚拟地址到物理地址的变换，由于它们处理的是物理地址。然而，将物理地址放到总线上似乎要求相反的变换在一次侦听期间查找虚拟标引的缓存，不管怎样它自己解决不了由不同表示产生的别名一致性问题。

别名问题可以通过软件方法，限制其使用来避免。例如，可以让别名有同样的页颜色，即，如果这些多于 $\log_2$（Page_size）位，就让用于索引缓存的部分是相同的。另一个方法是，当引用相同页的时候，可以要求进程用相同的虚拟地址，如同在 SPUR 研究项目中那样（Hill et al. 1986）。

437

人们也提出过复杂的缓存设计来从硬件的角度解决不同表示的问题（Goodman 1987）。其思想是处理器用虚拟地址来进行缓存查询，但将物理地址放到总线上让其他缓存和设备来侦听。这要求提供能实现下述功能的机制：1）如果查找虚拟地址失败了，能用物理地址来查询缓存（届时物理地址可用）或者如果检测到那个块曾被由一个别名访问带到了缓存；2）保证相同的物理块在两个不同的虚拟地址下不会同时出现在相同的缓存中；3）将一个侦听的物理地址转换到一个有效的虚拟地址来查询侦听缓存。实现这些目标的一个办法是让缓存对它们的块维护虚拟和物理两种标签（和状态），分别由虚拟和物理地址来标引，让一个块的两个标签相互指向（即分别存放相应的物理地址和虚拟地址；如图 6-24 所示）。缓存数据阵列本身用虚拟标引（或者是物理标记项中的指针，在侦听的情形，它们是一样的）。让我们看看在一个高层中这种组织是如何提供所需机制的。

处理器用它的虚拟地址查询缓存，同时如果需要的话，虚拟地址到物理地址的转换由存储管理单元来完成。如果用虚拟地址的查询成功，一切都好。如果它失败了，转换得到的物理地址就用来查找物理标签；如果这命中了，就可以通过物理标签中的指针来找到这一块。如下所述，这就达到了第一个目标。一个虚拟的扑空但物理的命中检测了一个别名的可能性，因为物理块可能已被通过不同的虚拟地址带进来了。在一个直接映像缓存中，它一定是通过一个不同的虚拟地址带到了缓存，因此为简单起见让我们假设直接映像缓存。包含在物

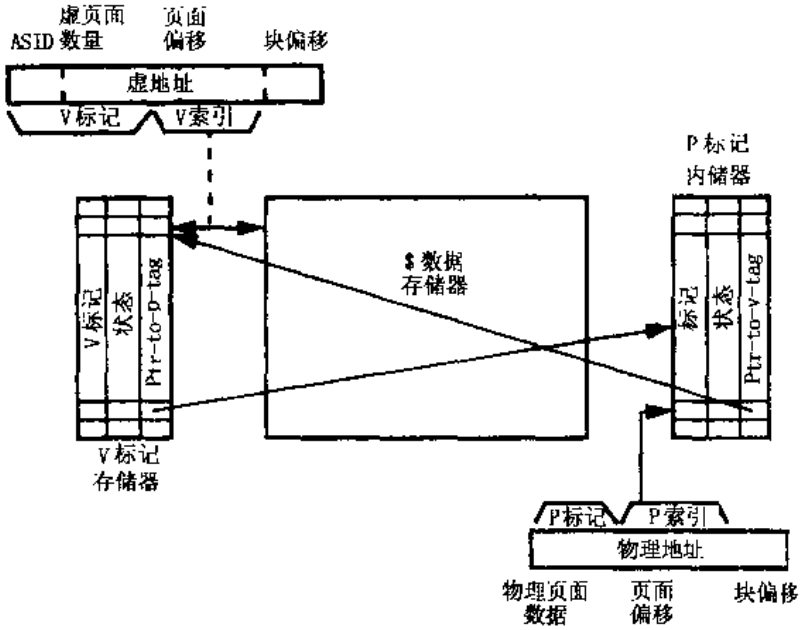


图 6-24　一种双标记的虚拟寻址的缓存的组织。左边的 V 标记存储器服务于 CPU，且由虚拟地址索引。右边的 P 标记存储器用于总线侦听，由物理地址索引。存储块的内容根据 V 标记的索引存放，相应的 P 标记和 V 标记条目相互指向，以处理对缓存的更新

理标签中的指针（别名虚拟标引）现在所指的缓存块不同于当前虚拟标引。我们需要使当前的虚拟标引指向这个物理数据，让虚拟地址和物理地址统一，以消除别名。当前被别名虚拟标引的物理块，被拷贝过来后，取代当前虚拟标引的块（如果必要，它会被回写），因此对当前虚拟标引的引用将从此立刻命中。在别名虚拟标引的块被置成作废或者不可访问，因此数据现在只能通过当前虚拟标引来访问（或者在侦听情形，借助于物理标记中的指针通过物理地址），但不能通过别名虚拟标引。对这个别名后续的访问将在它的虚拟地址查询上扑空，并且必须完成同样的过程。因此，在任何给定时刻，给定的物理块只是在缓存中一个（虚拟标引的）单元有效，于是实现了第二个目标。注意，如果虚拟地址和物理地址查询都失败了（真的缓存扑空），可能需要多到两次的回写。带入缓存的新块将被放在由当前虚拟（不是物理）地址确定的标引上，虚拟和物理的标记和状态将要适当地更新，相互指向。

不管是回写、读扑空、或者排他读或更新，放到总线上的地址总是物理地址。用从总线来的物理地址侦听是容易的。由于所需的信息已经在那里，不需要显式逆向转换。查询物理的标记以检查块的存在，从它所包含的指针找到数据。如果必须采取什么动作，由物理标记项所指的虚拟标记状态也要被更新。这样一种缓存系统如何操作的进一步的细节见（Goodman 1987）。这种做法也被扩展到多级缓存，此时它更具吸引力：L1 缓存用虚拟标记以加速缓存的访问，L2 缓存用物理标记以便利侦听和避免跨处理器的别名（Wang，Baer and Levy 1989）。

6.6.3　转换检测缓冲器的一致性

简单地讲，处理器的转换检测缓冲器（Translation lookaside Buffer，TLB）只不过是关于页表条目（PTE）的一个高速缓存，用于虚拟地址到物理地址的转换。由于实际的数据共享或者进程迁移，一个 PTE 可能出现在多个处理器的 TLB 中。PTE 可能被修改——例如，当存储

页被交换出去，或者它的保护被改变——导致和缓存一致性问题相似的情况。 438 ~ 439

有多种用于 TLB 一致性的方案。由于 TLB 一致性操作要比一般缓存一致性操作少得多，流行的做法是通过操作系统的软件方案。具体的方案取决于 PTE 是直接由硬件装入 TLB，还是由软件控制；还取决于 TLB 和操作系统如何实现的其他一些因素。硬件方案也用于某些系统中，特别是当 TLB 操作为软件不可见时。这一节对 4 种 TLB 一致性方法给出一个简略的概要：虚拟地址缓存、软件 TLB 击落、地址空间标识符（ASID）、硬件 TLB 一致性。进一步的细节可见（Thompson et al. 1988; Rosenburg 1989; Teller 1990）以及其中的参考文献。

TLB，以及由此而来的 TLB 一致性问题，能够通过用虚拟地址缓存完全避免。地址变换现在只在缓存扑空上需要，因此特别是如果缓存扑空率低，我们可以直接用页表。当页表项被访问的时候带入普通的数据缓存，因此由缓存一致性机制保持一致性。然而，当一个物理页被交换出存储器时，或者它的保护改变了，这些都是缓存一致性硬件看不见的，因此必须由操作系统将 PTE 从所有处理器的虚拟地址缓存中清除出去。除此以外，虚拟地址缓存的一致性问题必须要解决。这个方法在 SPUR 研究项目中得到了探讨（Hill et al. 1986; Wood et al. 1986）。

第二个方法称为 TLB 击落。有许多变形，取决于硬件支持的程度，通常包括对处理器之间中断的支持，个别 TLB 项的作废。这种 TLB 一致性过程由一个称为发起者的处理器在它改变那些可能被其他 TLB 缓存的 PTE 时启动。由于对 PTE 的变化必须由操作系统来做，操作系统知道哪些 PTE 正在被改变，它也可能知道哪些其他的处理器可能将它们缓存到它们的 TLB 中。（保守地，由于那些项可能已经被替换了）操作系统内核锁住正被改变的 PTE（或者相关的页表段，取决于锁的粒度），向其他被认为有拷贝的处理器发中断。一旦被中断，接受者屏蔽中断，察看正被修改的页表项（在共享存储中），在本地从它们的 TLB 中作废这些项。发起者等它们完成，也许通过轮询共享存储单元，然后打开页表段的锁。在 Mach 操作系统中，人们用了一种不同的、但要更复杂些的击落算法（Black 等 1989）。

某些处理器家族，尤其是 Silicon Graphics 的 MIPS 家族，用软件装入的 TLB 而不是硬件装入的 TLB，这意味着操作系统不仅涉及到 PTE 的修改，还涉及在扑空时将 PTE 装入 TLB。在这些情况下，由于进程迁移带来的处理器私有页面的一致性问题可以用第三种方法来解决，即 ASID，它避免了中断和 TLB 击落。每个 TLB 项都有一个 ASID 域与之相联，以避免在上下文交换时清除整个 TLB（就好像进程标识符用在虚拟地址缓存一样）。然而，在 TLB 的 440
情形，ASID 像是由操作系统以处理器为单位动态分配的标签，用一个自由池，当 TLB 项被替换时，让它们返回其中；它们不是在整个生命周期都和进程相联的。IRIX 5.2 操作系统用 ASID 的方法如下。操作系统为每个进程维护一个数组，跟踪分配给系统中每个处理器的那个进程的 ASID。当一个进程修改 PTE 时，对所有其他处理器那个进程的 ASID 被置零。这保证了当进程迁移到另外的处理器时，进程将发现它的 ASID 为零；因此内核就要分配给进程一个新的 ASID 值，这样就防止了用过时的 TLB 项。对于由进程真共享的页的 TLB 一致性用 TLB 击落来完成。

最后，某些处理器家族提供硬件指令来作废其他处理器的 TLB。在 PowerPC 家族中（Weiss and Smith 1994），“TLB 作废项”指令（tlbie）在总线上广播页地址，从而其他处理器上的侦听硬件能自动地作废相应的 TLB 项，而不中断处理器。处理 PTE 变化的算法是简单的：操作系统首先改变页表，然后发出一个针对这些变化的 PTE 的 tlbie 指令。如果 TLB 是

硬件装入的（如在 PowerPC 中那样），OS 不知道哪些其他的 TLB 可能缓存这个 PTE，因此这个作废必须广播到所有处理器。广播是很适合于总线的，但对于分布式网络的可扩放的系统来说却是不希望广播的，我们将在后边的章节讨论。

6.6.4　环上基于侦听的高速缓存一致性

由于基于总线的缓存一致性多处理器的规模受总线的限制，很自然我们就会问侦听式一致性怎么能够扩展到其他的、局限性小一些的互连结构。总线的一个直接扩展是环。和总线（所有模块都接在同一组线上）不同，环上每个模块都和相邻的两个模块相连。和总线相同的方面是，环固有地支持基于广播的通信，从一致性角度看它是一个有意思的互连网络。从一个节点到另一个节点的事务沿着环的链向下传递，由于目的节点的平均距离是环长的一半，应答的一种简单且自然的做法是让它传播通过环的剩余部分，返回发送者。事实上，用硬件组织通信结构的一种自然方法是，让发送者将事务放到环上，当事务通过时，让其他节点审查（侦听）此事务是否与自己有关。给定这种广播和侦听的基础结构，我们可以在环上提供侦听缓存一致性，即使存储器在物理上是分布在环的节点上。由于多个数据传送过程可能在环上同时进行，模块看到这些事务的时间不同，顺序可能潜在地不同，环上的串行化和顺序一致性要比总线上的复杂些。

除了用分布式存储器外，环相对于总线的优点还有较短的、点对点链接使它们能够以很高的时钟频率工作。例如，IEEE 可扩展一致性接口（SCI）传输标准（Gustavson 1992；IEEE
441 1993）就是基于 500 MHz，16 位宽的点对点连接。线性点对点特点也使得链路深度流水成为可能，即在前面的位没有到达目的节点之前新的位能够发送到线路上。这个特征允许链路做得长一些，同时不影响它们的吞吐量。环的一个缺点是通信时延高，高于总线，并且随着环上处理器个数增加而线性增长（平均来说，在单向环上，要达到目标，需要穿越 $p/2$ 跳；双向环上减半）。

由于环是一个广播介质，侦听缓存一致性协议可以很自然地在其上实现。早期的一种基于环的侦听缓存一致性机器是 Kendall Square Research 的 KSR1（Frank，Burkhardt，and Rothnie 1993）。最近一些的商用机器，诸如 Sequent NUMA-Q 和 Convex 的 Exemplar 家族（Convex 1993；Thekkath et al. 1997），用环作为二级互连，将多处理器节点连在一起。（这两种系统在环互连上都是用目录协议，而不是侦听，因此对它们的讨论将推迟到第 8 章，专门介绍那些协议。在 NUMA-Q 中，节点内的互连是总线；在 Exemplar 中，它是具有丰富连接关系，低时延的交叉开关。）多伦多大学的 Hector 系统（Vranesic et al. 1991；Faras，Vranesic and Stumm 1992）是一个基于环的研究原型系统。

图 6-25 示出一个用环连接的多处理器组织。典型地，环和物理上分布的存储一起使用，但存储器可能在逻辑上仍然是共享的。每个节点的构成包含有一个处理器、它的私有缓存、全局主存的一部分、还有一个环接口。这个环接口有一个从环上来的输入线，一组组织成 FIFO 缓冲区的锁存器，以及一个到环上的输出线。在每个环时钟周期，锁存器的内容向前移位，于是整个环就像一个环形流水线。锁存器的主要功能是将通过的信息符有足够长的时间，让接口来决定是否将它向前发送到下一个节点。一个信息可能被从环上取出，通过将锁存器的内容存入本地缓冲存储器，并向这个锁存器写入一个空时隙标记。如果一个节点要将事务放到环上，它要等待一个空时隙通过，并填充它。当然，我们希望在每个接口上的锁存

器数尽量小，以减少事务通过环的时延。

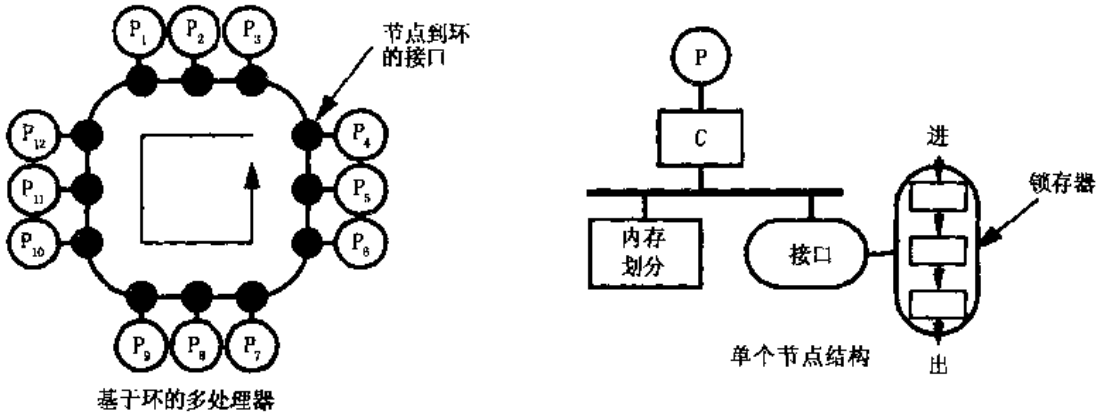


图 6-25　一种单环多处理器的组织结构

需要有一种机制，称为环访问控制机制，来确定一个节点何时能够将它的事务放到环上。它的复杂性在于：环上的数据通路要比在它上面传输的事务窄得多。结果，事务在环上需要多个相继的时隙。进一步，环上的事务（消息）本身大小也可能不一样。例如，请求消息短，只包含命令和地址，而数据应答消息存储块的内容要长得多。最后的复杂因素是，访问环的仲裁逻辑必须是分布式的，这是因为它不同于总线，没有全局可见的线路。

有三种主要的环访问控制（即仲裁）方法：令牌传递环、寄存器插入环和时隙环。在令牌传递环中，一种特别的位模式，称为令牌，在环上传递，只有当前占有这个令牌的节点才能在环上发送。这种情况仲裁是容易的，但缺点是同时只有一个节点能发起一个事务，尽管环上的其他节点可能传递的是空槽，这就导致带宽的浪费。寄存器插入环为 IEEE SCI 标准所选。这里，在环的接口输入和输出阶段之间，当本地节点在发送的时候，用一个旁路 FIFO 来缓冲输入消息（带有反向流控以避免溢出）。当本地节点完成时，旁路 FIFO 的内容就被转发到输出链路，本地节点不允许再发送，直到旁路 FIFO 为空。多个节点可以同时发送，环的各个部分当它们的旁路 FIFO 溢出时就要停滞。最后，在时隙化的环中，环被分为事务时隙，带有类型标签（针对不同大小的事物，如请求和数据应答），这些时隙在环上不断循环。一个准备好了发送消息的处理器等待所需类型的时隙的到来（由槽头的一位指出），然后插入它的消息。这里的“时隙”实际上是一串空时间片段，这个串的长度取决于消息的类型。时隙化的环可能会限制环带宽的利用率，用硬件混合不同类型的时隙，可能和给定负载的实际流量模式不匹配。然而，对于一个给定的一致性协议，人们对消息类型混合的情况了解得较好，因此在实践中几乎没有什么带宽浪费（Barroso and Dubois 1993，1995）。

也许我们会感到广播和侦听在像环这样的点到点互连上会浪费带宽，实际中不一定如此。在环上，和平均随机点对点消息相比，广播只须两倍的带宽，这是由于对两个随机选择的点来说，前者平均要穿越半个环。除此以外，只是请求消息需要广播（读扑空、写扑空、更新请求），而它们是短的；数据应答消息由数据源放到环上，运转到请求节点上停止。

442
~
443

考虑在基于环的侦听协议中的一次读扑空。如果我们关心的一个存储块所分配的存储单元（称为宿主存储器）不在本地节点，读请求就被放到总线上。如果宿主是本地的，我们还要确定这个块在其他节点是否脏，在这种情况，本地存储器不应该响应，一个请求应该放到总线上。一个简单的方案是将所有扑空都放到总线上，如 Sun Enterprise，它是一个基于总线

的设计，但存储在物理上是分布的。另一种方案是，对宿主存储器中的每一块都维护一个脏位。如果这一块在其他某个节点的缓存中为脏，这一位被置位。如果这一位是位，请求就送到环上　读请求现在沿着环循环。所有节点都检测它，宿主节点或者脏节点会作出响应（如果宿主不是本地，宿主节点就用脏位来决定它是否应该对请求做响应）　排他读和更新过程也出现在环上，其他节点侦听这些请求，必要时作废它们的存储块。当回到请求者时，请求和响应事务就从环上删除。请求返回到请求节点作为确认。当多个节点试图并发地写一个相同的块，赢者是首先到达该块的当前拥有者的节点（即如果那块在主存有效，则是宿主节点，否则是脏节点）；其他节点被隐式或显式送一个 NACK，然后它们必须重试。

从实现的角度看，环上侦听协议的一个主要难点在于所强加的实时限制：在环接口中的侦听器考察并对所有通过的消息作出反应，不能有过多的延迟或内部排队。这对于寄存器插入环是困难的，因为许多短请求消息可能在环上相邻。对于高速运行的环，请求可能相互挨得太近，侦听器难以在固定的时间里做出响应。这个问题在时隙环中得到简化，短请求消息和长数据响应消息（后者是点对点，不需缓存查找）的选择和布置能保证请求型消息相互有一定的距离（Barroso and Dubois 1995）。例如，时隙可以成组为帧，每个帧可以组织成请求时隙，紧跟着响应时隙。尽管如此，如同总线，环的带宽最终还是受控制器或缓存的侦听带宽限制，而不是受互连上的原始数据传送带宽限制。

环的串行化和顺序同一性要比总线复杂些，这是由于存在这样的可能性：环上不同点的处理器会看到环上一对消息有不同的顺序（取决于它们在环的什么地方，相对于哪些事务发送者）。用作废协议可以简化这个问题，由于写只是引起排他读事务放到环上，而不是数据本身，除宿主节点外的所有节点都只要简单作废它们的拷贝。宿主节点能确定环上出现冲突
444 的消息，并采取特定的行动，但这不会在协议中增加太多的暂态过程。

6.6.5　在基于总线的系统中的数据和侦听带宽的扩展

有一些方法可用来增加 SMP 设计中的带宽，同时保持基于总线做法的简单性。在事务拆分型总线中，我们看到仲裁、地址阶段和数据阶段的流水线使得它们能同时进行。事实上，扩展数据带宽事实上是较容易的；挑战在于拓展侦听的带宽。

首先考虑扩大数据带宽。缓存块的尺寸和描述它们的地址相比要大些[⊖]　增加数据带宽最直接的方法是使数据总线的宽度扩大。我们在 Sun Enterprise 和 SGI Challenge 的设计中看到了这一点，两者都用 256 位宽的数据总线。在 Enterprise 中，64 字节的缓存块只要两个周期就可以传送了。这种做法的不利之处是代价：随着总线的变宽，它要用较大的连接器，占据电路板上更多的空间，需要更大的功率。它显然将这种风格的设计推向极限，由于一个高效的、统一的流水线要求侦听操作必须在两个周期完成，侦听操作是需要被所有的缓存看见并确认的。一种更突出的做法是用点到点交叉开关代替总线的数据部分，直接将每个处理器－存储器模块相互连接。这种做法认识到了只是事务的地址部分需要广播到所有的节点，来确定一致性操作和数据源（即存储器或缓存）。这种做法在 IBM 基于 PowerPC 的 RS6000 G30 多处理器中被采纳。一个总线被用于地址和侦听结果，但交叉开关用来移动实际的数据。交叉开关中的单个通路不需要太宽，因为多个传送能同时发生。

⊖ 例如缓存块可能为 16 字节，而地址为 8 字节。——译者注

在基于总线系统中提高带宽的一个笨办法是简单地用多条总线，包括地址总线，如 6.4.7 节所提到的那样。事实上，这个做法提供了一种基础性贡献，它允许侦听带宽也能扩展。为了将侦听带宽扩展到在每地址周期一个一致性结果之外，必须有多个同时的侦听操作。一旦有了多个地址总线，数据总线问题可以通过数据总线，交叉开关或者任何其他机制来处理。同一性是容易的。地址空间的不同部分用不同的总线；典型地，每条总线将为特定的存储模块服务，因此一个给定的地址总是用相同的总线。多个地址总线似乎违反了用于保证顺序同一性关键的机制：对地址总线的序列化仲裁。然而，我们记得，顺序同一性要求的是一个逻辑的总序，而不是一个严格的地址事件的发生时间序。一种静态的序在逻辑上赋予参与的总线操作序列：一个地址操作 i 逻辑上在 j 之前，如果它在 j 之前出现（按照总线时钟）或者它们在相同的时钟周期发生，但 i 发生在较低编号的总线上。这种多总线的方法用在 Sun SparcCenter 2000 中，它提供两条事务拆分型总线，每条和 SparcStation 1000 中用的一样，可以扩展到 30 个处理器。CRAY CS6400 用 4 条这样的总线，可以扩展到 64 个处理器。每个缓存控制器侦听所有总线，按照缓存一致性协议作出响应。Sun Enterprise 10000，比本章讨论的 Sun Enterprise 6000 后发布的机器，将多个地址总线和数据交叉开关结合起来，可 445
以扩展到 64 个处理器。每个板上有 4 个 250 MHz 的处理器，4 个存储模块（每个最多 1 GB），以及两个独立的 SBUS I/O 总线。16 个这样的板通过 16×16 数据交叉开关连接起来，144 位宽的通路，以及 4 条和每个板上 4 个模块相联的地址总线。总体上来说，这就提供了 12.6 GBps 的数据带宽和一个高达 250 MHz 的侦听速率。

6.7　结论

本章探讨的设计问题是具有基础意义的；虽然所用的都是小规模并行的例子，但其中的原理在中等规模并行的情形也有指导作用。当然，最优的设计方案可能改变。例如，尽管当前不那么流行，但在一定条件下，在某个层次共享缓存可能变得相当有吸引力，这种条件就是多模块封装技术变得便宜起来，或者多个处理器出现在单个芯片上。

处理器通过共享总线互连，尽管是一个非常有效的机制，但随着处理器的数目或者速度的增加，受到明显的带宽限制。尽管如此，至少在小规模上，系统设计师们肯定还会不断去寻找新方法，从总线型设计中“榨取”更大的数据带宽和侦听带宽，来充分地利用基于广播方法的简单性。然而，建造可扩放缓存一致性机器的一般方法是让存储器物理上分布在节点上，用一种可缩放的互连，以及不依赖于侦听的一致性协议。这个方向是后面几章的主题。随着处理器变得比总线和侦听带宽快，这种思路在较小规模也可能有用。对于总线结构来说，肯定在一定的时间里还会有重要的作用，但现在难以预测它们的将来，以及在什么样的规模上还会用它们。虽然总会有各种变化和进步，但我们针对总线在这两章里讨论的问题，大多数都是独立于工艺的，对所有设计缓存一致性共享存储系统结构都至关重要，不管用何种互连方式。这些问题包括存储层次内的互连，缓存一致性问题和在状态转换层次的各种缓存一致性协议，以及当处理许多并发过程时所引起的关键正确性和实现问题。用于解决这些正确性和实现问题的具体做法可能改变，但这些问题本身、围绕它们所作的权衡以及基本的出发点是具有基础意义的，可以外推。进而，对于本书后面要讨论的大规模系统的设计来说，这些基于总线的设计提供了基本的元素。

习题

6.1　考虑两个机器 M_1，M_2。M_1 是一个 4 处理器共享 L_1 高速缓存机器，M_2 是一个 4 处理器基于总线的侦听缓存机器。M_1 有一个共享的 1 MB 两路组相联高速缓存，64 字节的块；M_2 中的每个处理器有一个 256 KB 的直接映像高速缓存，64 字节块。M_2 用 Illinois MESI 一致性协议。考虑如下代码：

446

```
double A[1024,1024]; /* row-major; 8-byte elems */
double C[4096];
double B[1024,1024];

for (i=0; i<1024; i+=1) /* loop-1 */
  for (j=myPID; j<1024; j+=numPEs)
    {
     B[i,j] = (A[i+1,j] + A[i-1,j] +
           A[i,j+1] + A[i,j-1]) / 4.0;
    }
for (i=myPID; i<1024; i+=numPEs) /* loop-2 */
  for (j=0; j<1024; j+=1)
    {
     A[i,j] = (B[i+1,j] + B[i-1,j] +
           B[i,j+1] + B[i,j-1]) / 4.0;
}
```

⊖

1）假设数组 A 从 16 进制地址（0x）0 开始，数组 C 从0x300 000开始，数组 B 从0x308 000开始。所有高速缓存都初始化为空。每个处理器执行上面的代码，和 4 个处理器对应，myPID 在 0 到 3 之间变化。针对两个循环嵌套，分别计算 M_1 的扑空。对 M_2 也做一遍，指明你所作的任何假设。

2）如果没有数组 C，简单地说明你的 1）中答案会如何改变。指明你所作的任何其他假设。

3）从这个练习中，我们了解到共享高速缓存结构的什么优缺点？

6.2　假设你从前面的章节和 5.4 节的数据中已经有了关于 Barnes-Hut、Ocean、Raytrace 和 Multiprog 负载的知识。考虑一个 4 处理器，共享 4 MB 高速缓存的机器和一个 4 处理器，基于总线侦听的机器，每处理器 1 MB 高速缓存，讨论这些应用在它们上执行的情况。用模拟来验证你的直觉可能是有用的。指明任何相关的假定。

6.3　和共享一级高速缓存相比，私有一级高速缓存但共享二级高速缓存有什么优缺点？评价现代处理器的设计中，例如 MIPS R10000 和 IBM/Motorola 的 PowerPC 620，是如何体现或回避这种趋势的。在这些设计中封装技术的影响是什么？

6.4　用 6.3 节关于高速缓存包含的术语，假设 L_1 和 L_2 是两路组相联，$n_2 > n_1$，$b_1 = b_2$，替换策略是 FIFO，不是 LRU。包含关系自然成立吗？如果替换是随机的或基于循环计数器又如何？

447

○ 原著中将 A [i+1, j] 和 B [i-1, j] 误为 A [i+i, j] 和 B [i+i, j]。——译者注

6.5　针对下述情形，给出一个表现高速缓存包含性质被违反的存储引用流：

1）L_1 高速缓存是 32 字节，两路组相联，8 字节缓存块，LRU 替换。L_2 缓存是 128 字节，4 路组相联，8 字节缓存块，LRU 替换。

2）L_1 高速缓存是 32 字节，两路组相联，8 字节缓存块，LRU 替换。L_2 缓存是 128 字节，两路组相联，16 字节缓存块，LRU 替换。

6.6　对于下面的系统，说明高速缓存是否自然地提供包含：如果不，说明问题在哪里或者给出一个违反包含的例子。

1）L_1：8 KB 直接映像基本指令高速缓存，32 字节块大小；8 KB 直接映像基本数据高速缓存，直写，32 字节块。

L_2：4 MB 4 路组相联统一二级高速缓存，32 字节块大小。

2）L_1：16 KB 直接映像统一基本高速缓存，直写，32 字节块大小。

L_2：4 MB 4 路组相联统一二级高速缓存，64 字节块大小。

6.7　回顾 6.3 节中关于高速缓存包含性质的讨论。

1）在通常情形，包含能相当自然地被满足。这情形是，L_1 高速缓存是直接映像（$a_1 = 1$），L_2 可以是直接映像或组相联（$a_2 \geq 1$），带有任何替换策略（例如，LRU、FIFO、随机），只要带进来的新块放到 L_1 和 L_2 高速缓存，块的大小是相同的（$b_1 = b_2$），L_1 高速缓存中的组数等于或小于 L_2 的（$n_1 \leq n_2$）。说明这为什么是对的。

2）我们的讨论称，用一个统一高速缓存支撑在某一级的多个高速缓存所产生的问题并没有通过使那个统一高速缓存相联而得到解决。用一个简单的例子，有直接映像指令和数据高速缓存并由一个统一的、两路组相联高速缓存支持，说明这是对的。

6.8　假设每个处理器有分开的指令和数据高速缓存，且没有指令扑空。还假定，当活跃时，每 3 个时钟周期处理器发出一个数据高速缓存请求，扑空率是 1%、扑空时延是 30 周期。假设读标记要 1 个时钟周期，但修改标记要两个时钟周期。

1）如果用带有一组高速缓存标记的单级数据高速缓存，试量化由于高速缓存标记争用的性能损失。假设总线过程要求侦听每 5 个时钟周期发生一次，它们的 10%将高速缓存的一个块作废。还假设侦听的优先级高于处理器访问标记的优先级。做粗略地计算。然后通过建立一个排队模型或写一个简单的模拟器，检查你的答案的精确度。

448

2）如果对处理器和侦听用分开的标记集，标记竞争的性能损失如何？

3）一般来讲，你会将访问标记的优先级给与处理器引用还是总线侦听？

6.9　SGI Challenge 多处理器的设计者们考虑了下面的总线控制器优化，以更好地利用交叉存储和总线带宽。如果控制器发现对一个给定的存储模块（它可以通过请求表来确定），一个请求已经是待完成的，它就不发出那个请求，直到那个模块的先前请求被满足。讨论这种优化潜在的问题，Challenge 设计中的什么特征允许这种优化？

6.10　1）尽管 Challenge 支持 MESI 协议的状态，它不支持最初 Illinois MESI 协议中的高速缓存到高速缓存的传送特征。

ⅰ）讨论这种选择的可能原因。

ⅱ）扩充 Challenge 实现，以支持高速缓存到高速缓存的传送。如果有的话，描述

总线上所需的额外信号，不要忘了公平性问题。

2）尽管 Challenge MESI 协议有四个状态，和高速缓存控制器芯片一起存放的标记只跟踪三个状态（I、S、E + M）。解释为什么这依然可以正确工作。你认为他们为什么做这种优化？

3）针对 SparcCenter 的多总线体系结构和 SGI Challenge 的单-快-宽总线体系结构，讨论它们之间代价、性能、实现和扩展性方面的权衡，以及任何程序语义和死锁的影响。

6.11　1）Challenge 的主存，甚至在它决定出一个数据是否在其他处理器的高速缓存中脏之前，可以推测式地启动该数据的读取过程。利用表 5-1 的数据，估计无用主存访问的比例。基于那个数据，你支持这种优化吗？这个数据从方法论上是充分的吗？请解释。

2）Challenge 的总线接口支持请求的融合。这样，如果多个处理器在等待同一个存储模块，那么当数据出现在总线上时，它们都可以从总线上抓住数据。这个特征在实现基于踏步等待的同步原语时特别有用。对于测试-测试和设置锁，计算有这种优化和没有这种优化时总线上最小的流量。假设有 4 个处理器，每个获得锁一次然后做一次开锁，最初没有处理器在其高速缓存中有包含锁变量的存储块。

6.12　SGI Challenge 总线允许 8 个待完成事务。设计人员是怎么得到这个数的？试建议一个通用公式，能够基于总线的参数给出应该支持的待完成过程数。用如下参数：

449

P——处理器数；

M——存储器模块数；

L——平均存储时延（周期）；

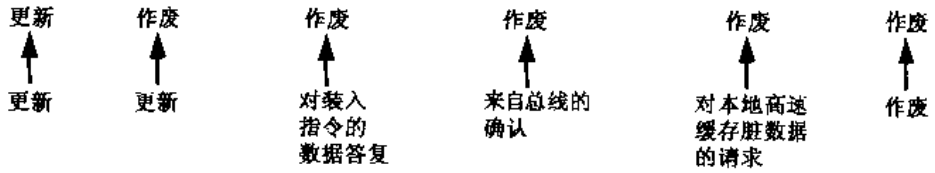
B——高速缓存块大小（字节）；

W——数据总线宽度（字节）。

定义其他你认为重要的参数。保持你的公式简单，讲清楚任何假设，说明你的决定的正确性。

6.13　在一个两级高速缓存系统中，考虑从总线到高速缓存层次的入向事务（请求和响应）和从高速缓存层次到总线的出向事务。为了保证当作废被提前确认（一旦它们出现在网络上）时的顺序同一性（SC），对于下图所描述的，必须在每一个方向上维护什么序的限制，哪些可以被重定序？对于一级高速缓存的入向事务回答同样的问题。假设每个处理器同时只有一个待完成请求，但总线是拆分型的。

向上的序（从总线到高速缓存和处理器）



向下的序（从高速缓存和处理器到总线）

6.14　在 6.4 节讨论的拆分型解决方案中，取决于处理器到高速缓存的界面，可能一个作废请求立刻在数据响应后到来，因此这个块，在处理器有机会实际访问高速缓存块满足它的请求之前就被作废。这为什么可能是个问题，你如何解决它？

6.15　当支持免锁高速缓存时，一个设计人员建议，我们也可以在事务拆分型总线接口中的请求表中加更多的项。这个想法好吗？是否能得到较大收益？

6.16　应用 6.4.6 节中描述的不同的技术在维持例 6.3 中有多个待完成总线事务的 SC，使你自己相信它们的确能解决问题。在什么条件下一种方案要好于其他的？ 450

6.17　假设一种类似于 Powerpath-2 那样的系统总线，如 6.5 节所讨论的。假设 200 MIPS/200 MFLOPS 处理器，1 MB 高速缓存和 64 字节高速缓存块，对于表 5-1 中的每个应用，计算总线带宽，用

1）Illinois MESI 协议

2）Dragon 协议

3）Illinois MESI 协议，假设 256 字节的高速缓存块

对每个部分 1），2），3）分别计算地址 + 命令总线的利用率和数据总线的利用率。阐明所有假设。

4）针对 SparcCenter XDBus，比较 1）和 2）两个部分，它有 64 位宽的多选地址和数据信号。假设总线在 100 MHz 上运行，发送地址信息要两个总线周期，64 字节的数据要 9 个总线周期。

6.18　在 6.4.8 节提出的关于多级高速缓存死锁的一个解决方案是使所有队列足够深，能够容纳所有的入向请求和响应。这些队列可以小一些吗？若如此，为什么？讨论为什么让队列的深度大于死锁考虑所需的深度可能是有益处的。

6.19　6.3 节给出的一致性协议假设两级高速缓存。如果有三级或更多的高速缓存层次又会如何？试将 Illinois MESI 协议扩充到一个三级高速缓存的中间级：列出任何附加的状态或所需的动作，给出状态转换图。

6.20　图 6-26 表示用在 Mach 操作系统中的 TLB 击落算法的细节（Black et al.1989）。对每个处理器来说，维护下面的基本数据结构：一个“活跃”标识，指出处理器是否正活跃地用着页表；一个 TLB 清理通知的队列，指出虚拟地址的范围，其映射要被改变；一个指出当前活跃页表的表（即那些其 PTE 当前可能被存到处理器的 TLB 中的进程）。对每一个页表，当一个处理器在改变页表时必须持有一把踏步等待锁和一组处理器，在其上这个页表当前是活跃的。虽然基本击落方法是简单的，实际的实现需要仔细安排有关步骤的顺序和数据结构的加锁。

1）为什么页表项的修改时间要在对 TLB 一致性的处理器发送中断或者作废消息之前？

2）为什么图 6-26 中击落的发起者要在获得页表锁之前屏蔽处理器之间的中断（IPI）并清除它自己的活跃标记？你能想到图中存在的任何死锁条件吗，若如此，怎样解决它们？

3）和 Mach 算法有关的一个问题是，当发起者改变页表时，它使所有的响应者忙等待。这里的原因是它的设计思想，在 TLB 替换时，只要脏位被设置了，就用微处理器自主回写整个 TLB 项到相应的 PTE。这样，如果其他处理器被允许用页表，当发起者正在改变它的时候，一个从这些处理器来的自主的回写可能覆盖新的变 451

化。你会怎样设计 TLB 硬件和/或算法，使得响应者不要忙等待？

4）在什么情况下，清除整个 TLB 要比有选择地使 TLB 的某些项无效来得更好？

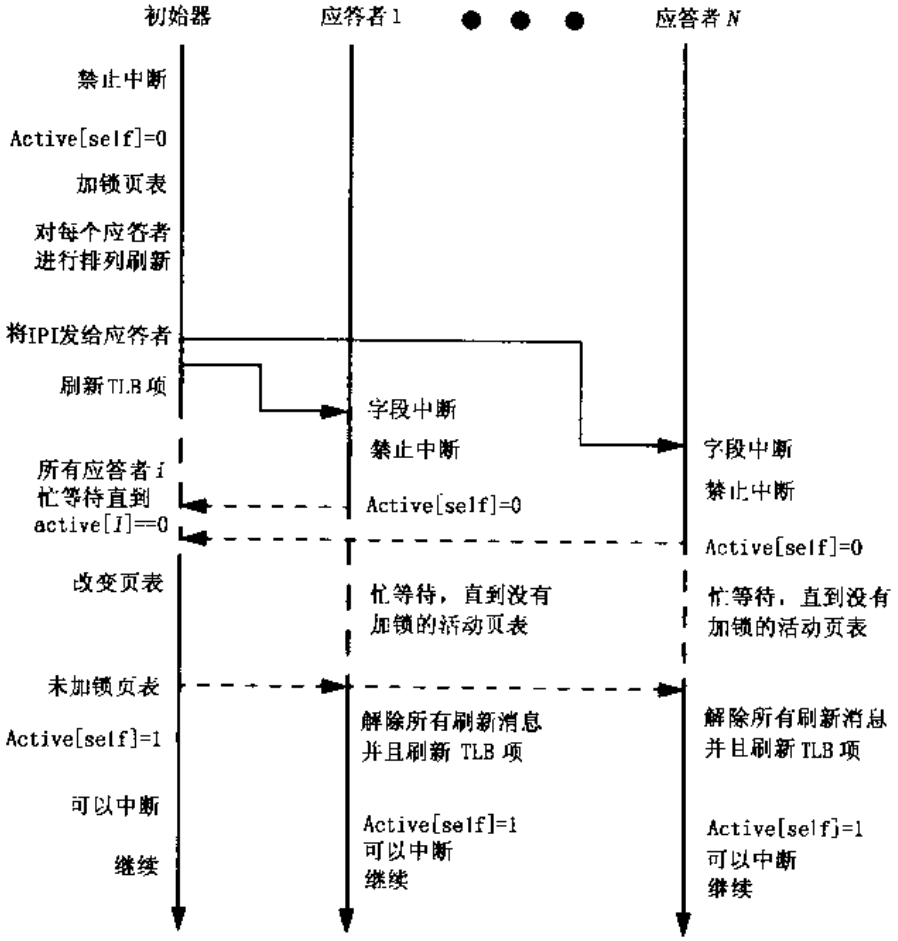


图 6-26　Mach TLB 的击落算法。发起者是改变某个页表的处理器，响应者是所有其他处理器，它们的 TLB 中可能高速缓存有发起者所改变页表的一些条目

452