内存屏障：软件黑客的硬件视角

保罗·E·麦肯尼

Linux技术中心

IBM比弗顿paulmck@linux.vnet.ibm.com

2010年6月7日

*--翻译：百度翻译，--整理：yankefei <yankefei1114@163.com>*

那么，是什么让CPU设计人员对可怜的毫无戒心的SMP软件设计人员设置了内存屏障呢？

简而言之，因为对内存引用进行重新排序可以获得更好的性能，所以需要内存屏障来强制同步原语之类的东西进行排序，它们的正确操作取决于有序的内存引用。

要想得到这个问题更详细的答案，需要很好地理解CPU缓存是如何工作的，尤其是如何使缓存真正工作良好。以下各节：

1.    呈现缓存的结构，

2.    描述缓存一致性协议如何确保CPU对内存中每个位置的值达成一致，最后，

3.    概述存储缓冲区和失效队列帮助缓存和缓存一致性协议如何实现高性能。

我们将看到，内存屏障是实现良好性能和可伸缩性所必需的一个evil，这一evil源于这样一个事实：CPU比它们之间的互连和它们试图访问的内存都快几个数量级。

# 1缓存结构

现代CPU比现代存储系统快得多。2006年的CPU可能每纳秒能执行10条指令，但从主存获取数据项需要数十纳秒。这种速度上的差异——超过两个数量级——导致了现代CPU上的多兆缓存。这些缓存与CPU关联，如图1所示，通常可以在几个周期内访问。[1]

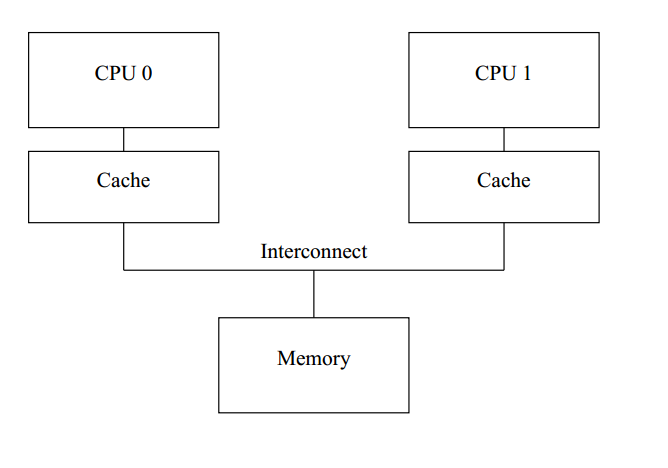


图1：现代计算机系统缓存结构

[[1]](#_ftnref1)标准做法是使用多个级别的缓存，一个小的一级缓存靠近CPU，具有单周期访问时间，一个较大的二级缓存具有更长的访问时间，可能大约10个时钟周期。高性能CPU通常有三级甚至四级缓存。

数据在CPU的缓存和内存之间以固定长度的块（称为“cache lines”）流动，通常大小为2的幂，从16字节到256字节不等。当给定的CPU首次访问给定的数据项时，该CPU的缓存中将不存在该数据项，这意味着发生了“缓存未命中”（或者更具体地说，**“启动”或“预热”缓存未命中**）。缓存未命中意味着在从内存中提取项目时，CPU将不得不等待（或“暂停”）数百个周期。但是，该项将加载到该CPU的缓存中，以便后续访问将在缓存中找到它，从而全速运行。

一段时间后，CPU的缓存将被填满，随后的未命中可能需要从缓存中弹出一个项目，以便为新获取的项目腾出空间。这种缓存未命中被称为“**容量未命中**”，因为它是由缓存的有限容量引起的。然而，大多数缓存可能会被迫弹出旧项目，以便为新项目腾出空间，即使它们尚未满。这是因为大型缓存被实现为硬件哈希表，具有固定大小的哈希桶（或CPU设计者称之为“sets”），没有链接，如图2所示。

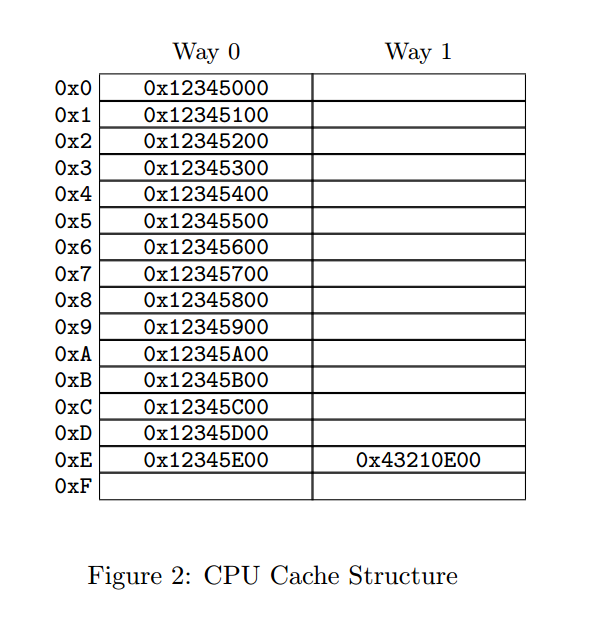


图2：CPU缓存结构

这个缓存有16个“sets”和两个“Way”，总共32行，每个条目包含一个256字节的“cache lines”，这是一个256字节对齐的内存块。这种cache lines的大小在很大的尺寸上有点小，但使十六进制算法更简单。用硬件术语来说，这是一个双向关联缓存，类似于一个包含16个bucket的软件哈希表，其中每个bucket的哈希链最多只能包含两个元素。大小（本例中为32条cache lines）和关联性（本例中为两条）统称为缓存的“几何体”。因为这个缓存是在硬件中实现的，所以哈希函数非常简单：从内存地址中提取四位。

在图2中，每个框对应一个缓存条目，它可以包含一个256字节的cache lines。但是，缓存项可以是空的，如图中的空框所示。其余的框用它们包含的cache lines的内存地址标记。由于cache lines必须是256字节对齐的，每个地址的低位8位为零，而硬件哈希函数的选择意味着下一个高位4位与哈希行号匹配。

如果程序代码位于地址，则可能出现图中所示的情况:0x43210E00到0x43210EFF，该程序从0x12345000到0x12345EFF顺序访问数据。假设程序现在要访问位置0x12345F00。该位置散列到第0xF行，该行的两条路径均为空，因此可以容纳相应的256字节行。如果程序要访问位置0x1233000（散列到第0x0行），则相应的256字节cache lines可以通过方式1容纳。但是，如果程序要访问位置0x1233E00（散列到第0xE行），则必须从缓存中弹出现有的一行，以便为新cache lines腾出空间。如果稍后访问此弹出的行，将导致缓存未命中。这种缓存未命中被称为“**关联性未命中**”。

到目前为止，我们只考虑CPU读取数据项的情况。当它书写时会发生什么？因为所有CPU都必须对给定数据项的值达成一致，所以在给定CPU写入该数据项之前，必须首先使其从其他CPU的缓存中删除或“失效”。一旦失效完成，CPU可以安全地修改数据项。如果该数据项存在于该CPU的缓存中，但为只读(表示其他的CPU也有使用)，则该过程称为“**写未命中**”。一旦一个给定的CPU从其他CPU的缓存中失效了一个给定的数据项，该CPU就可以重复地写入（和读取）该数据项。

稍后，如果另一个CPU试图访问该数据项，它将导致缓存未命中，这一次是因为第一个CPU为了写入该数据项而使该数据项无效。这种类型的缓存未命中被称为“**通信未命中**”，因为它通常是由于多个CPU使用数据项进行通信（例如，锁是用于使用互斥算法在CPU之间进行通信的数据项）。

显然，必须非常小心地确保所有CPU对数据保持一致的视图。通过所有这些获取、失效和写入操作，很容易想象数据丢失，或者（可能更糟的是）不同的CPU在各自的缓存中对同一数据项有冲突的值。这些问题可以通过“缓存一致性协议”来避免，这将在下一节中介绍。

# 2缓存一致性协议

缓存一致性协议管理cache lines状态，以防止数据不一致或丢失。这些协议可能非常复杂，有几十个状态[2]，但出于我们的目的，我们只需要关注四态的MESI缓存一致性协议。

[[2]](#_ftnref2)参见Culler等人[CSG99]第670页和第671页，分别了解SGI Origin2000和Sequent（现在的IBM）NUMA-Q的ninestate和26个状态图。这两张图都比现实生活简单得多。

## 2.1MESI状态

MESI代表“修改的（modified）”、“独占的(exclusive)”、“共享的(shared)”和“无效的(invaild)”，这四种状态是给定cache lines可以使用该协议的状态。因此，使用该协议的缓存在每条cache lines上除了保留该cache lines的物理地址和数据外，还保留一个两位状态“**标记**”。

处于“修改”状态的一行已经受到来自相应CPU的最近内存存储的影响，并且相应的内存保证不会出现在任何其他CPU的缓存中。因此，处于“修改”状态的cache lines可以说是由CPU“拥有”。因为此缓存保存数据的唯一最新副本，所以此缓存最终负责将其写回内存或将其交给其他缓存，并且必须在重用此line保存其他数据之前执行此操作。

“独占”状态与“修改”状态非常相似，唯一的例外是cache lines尚未被相应的CPU修改，这反过来意味着驻留在内存中的cache lines数据的副本是最新的。然而，由于CPU可以随时存储到该行，而无需咨询其他CPU，因此处于“独占”状态的行仍然可以说是由相应的CPU所有。这就是说，因为内存中相应的值是最新的，所以这个缓存可以丢弃这些数据，而无需将其写回内存或将其交给其他CPU。

处于“共享”状态的线路可能会被复制到至少一个其他CPU的缓存中，因此，在未事先咨询其他CPU的情况下，不允许该CPU存储到line中。与“独占”状态一样，由于内存中的相应值是最新的，因此该缓存可以丢弃该数据，而无需将其写回内存或将其交给其他CPU。

处于“无效”状态的行是空的，换句话说，它不包含任何数据。当新数据进入缓存时，如果可能的话，它会被放入处于“无效”状态的cache lines中。这种方法是首选的，因为在任何其他状态下替换行可能会导致昂贵的缓存未命中（如果将来引用替换的行的话）。

由于所有CPU都必须对cache lines中携带的数据保持一致的视图，因此缓存一致性协议提供了协调cache lines在系统中移动的消息。

## 2.2 MESI协议消息

上一节中描述的许多转换都需要CPU之间的通信。如果CPU位于单个共享总线上，则以下消息就足够了：

**Read：**“读取”消息包含要读取的cache lines的物理地址。

**Read Response：**“读取响应”消息包含先前“读取”消息请求的数据。此“读取响应”消息可能由内存或其他缓存之一提供。例如，如果其中一个缓存的所需数据处于“修改”状态，则该（表示其他）缓存必须提供“读取响应”消息。

**Invalidate：**“使无效”消息包含要失效的cache lines的物理地址。所有其他缓存必须从其缓存中删除相应的数据并做出响应。

**Invalidate Acknowledge：**从缓存中删除指定数据后，接收到“无效”消息的CPU必须以“无效确认”消息进行响应。（由其他的CPU来响应）

**Read Invalidate：**“读无效”消息包含要读取的cache lines的物理地址，同时指示**其他缓存**删除数据。因此，它是“read”和“invalidate”的组合，如其名称所示。“读取无效”消息需要“读取响应”和一组“无效确认”消息作为回复。

**WriteBack：**“写回”消息包含要写回内存的地址和数据（可能会在过程中“窥探”到其他CPU的缓存）。此消息允许缓存根据需要弹出处于“修改”状态的行，以便为其他数据腾出空间。

有趣的是，一个共享着内存的多处理器系统实际上是一个隐蔽的信息传递计算机。这意味着使用分布式共享内存的SMP机器集群正在使用消息传递在系统架构的两个不同级别上实现内存共享。

**快速测试1：**如果两个CPU试图同时使同一cache lines失效，会发生什么情况？

**快速测试2：**当大型多处理器中出现“invalidate”消息时，每个CPU都必须给出“invalidate acknowledge”响应。由此产生的“无效确认”响应的“风暴”不会使系统总线完全饱吗？

**快速测试3：**如果SMP机器真的在使用消息传递，为什么还要使用SMP呢？

## 2.3 MESI状态图

给定cache lines的状态随着协议消息的发送和接收而改变，如图3所示。

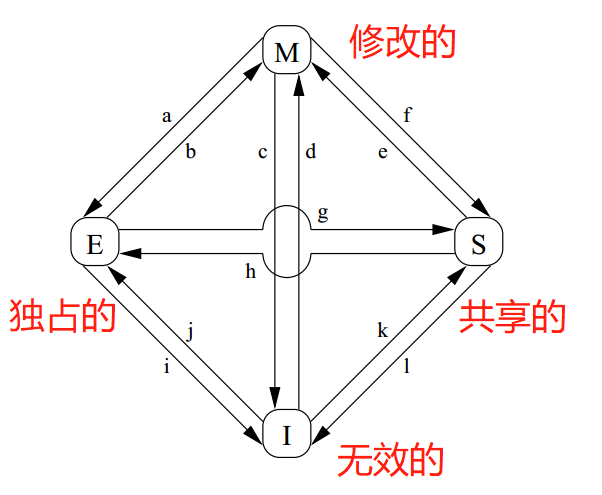


图3:MESI缓存一致性状态图

图中的过渡弧如下所示：

**过程（a）：**cache lines被写回内存，但CPU将其保留在缓存中，并进一步保留修改它的权利。此转换需要“写回”消息。

**过程（b）：**CPU向cache lines写入它已经拥有独占访问权限的数据。这种转换不需要发送或接收任何消息。

**过程（c）：**CPU收到一条“读取无效”消息，用于它已修改的cache lines。CPU必须使其本地副本无效，然后用“读取响应”和“无效确认”消息进行响应，这两个消息都将数据发送到请求的CPU，并指示其不再具有本地副本。（指示其他缓存删除消息）

**过程（d）：**CPU对缓存中不存在的数据项执行原子的[读-修改-写]操作。它发送“读取无效”，通过“读取响应”接收数据。一旦CPU也收到一整套“无效确认”响应，它就可以完成转换。（表示对一个数据进行了修改）

**过程（e）：**CPU对缓存中以前为只读的数据项执行原子的[读-修改-写]操作。它必须传输“无效”消息，并且必须等待一整套“无效确认”响应，然后才能完成转换。

**过程（f）：**另一个CPU读取cache lines，它由这个CPU的缓存提供，该缓存保留一个只读副本，可能还会将其写回内存。这种转换是通过接收“读取”消息来启动的，CPU用包含请求数据的“读取响应”消息进行响应。（有其他CPU来获取刚修改过的数据）

**过程（g）：**其他一些CPU读取该cache lines中的数据项，该数据项由该CPU的缓存或内存提供。在这两种情况下，该CPU都保留一个只读副本。这种转换是通过接收“读取”消息来启动的，CPU用包含请求数据的“读取响应”消息进行响应。

**过程（h）：**这个CPU意识到它很快就需要写入这个cache lines中的一些数据项，从而传输一条“invalidate”消息。CPU在收到一整套“无效确认”响应之前无法完成转换（让其他CPU的数据失效）。或者，所有其他CPU通过“写回”消息从缓存中弹出该cache lines（可能是为了给其他cache lines腾出空间），因此该CPU是缓存它的最后一个CPU。

**过程（i）：**其他一些CPU对cache lines中的数据项执行原子的[读-修改-写]操作，该cache lines仅保存在该CPU的缓存中，因此该CPU使其从缓存中失效。此转换是通过接收“读取无效”消息而启动的，该CPU同时响应“读取响应”和“无效确认”消息。

**过程（j）：**该CPU对cache lines中不在其缓存中的数据项进行store，从而传输“读取无效”消息。CPU在收到“读取响应”和一整套“无效确认”消息之前无法完成转换（让其他CPU的数据失效）。一旦实际存储完成，cache lines可能会通过转换（b）转换到“修改”状态。

**过程（k）：**此CPU将数据项加载到cache lines中，而cache lines不在其缓存中。CPU发送“读取”消息，并在收到相应的“读取响应”后完成转换。

**过程（l）：**其他一些CPU会store此cache lines中的数据项，但会将此cache lines保持为只读状态，因为它被保存在其他CPU的缓存中（例如当前CPU的缓存）。这种转换是通过接收“无效”消息来启动的，CPU会以“无效确认”消息进行响应。

**快速测试4：**硬件如何处理上述延迟转换？

## 2.4 MESI协议示例

现在，让我们从cache lines数据的角度来看这一点，当它在四CPU系统中通过各种单行直接映射缓存时，数据最初位于地址0的内存中。表1显示了这种数据流，第一列显示操作顺序，第二列显示CPU执行操作，第三列显示正在执行的操作，接下来的四列显示每个CPU cache lines的状态（内存地址后接MESI状态），最后两列显示相应的内存内容是否为最新（“V”）或非最新（“I”）。

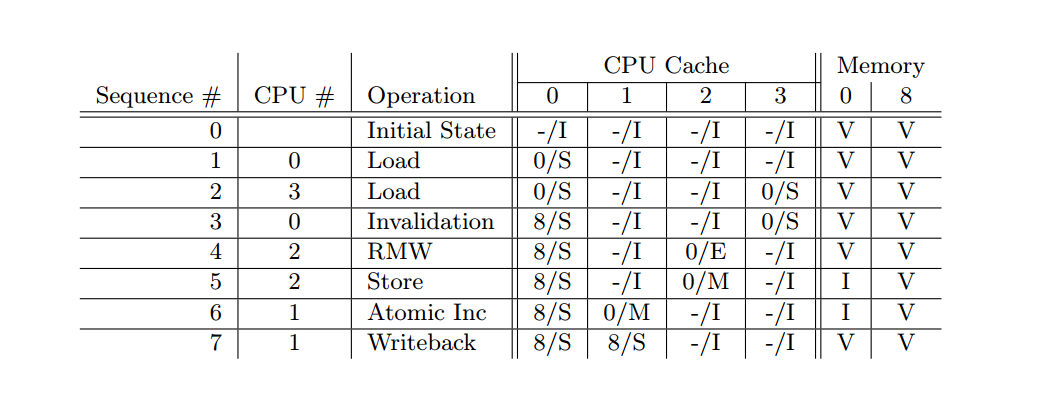


表1：缓存一致性示例

最初，数据所在的CPU cache lines处于“无效”状态，并且数据在内存中有效。当CPU 0加载地址0处的数据时，它在CPU 0的缓存中进入“共享”状态，并且在内存中仍然有效。CPU 3还加载地址0处的数据，使其在两个CPU的缓存中都处于“共享”状态，并且在内存中仍然有效。

接下来，CPU 0加载一些其他cache lines（地址8处），通过失效将地址0处的数据强制移出缓存，并用地址8处的数据替换。

CPU 2现在从地址0进行加载，但该CPU意识到它很快需要store到地址0，因此它使用“read invalidate”消息来获取独占副本，使其从CPU 3的缓存中失效（尽管内存中的副本保持最新）(RMW,读-修改-写)。下一个CPU 2执行其预期存储，将状态更改为“已修改”。内存中的数据副本现在已过期。

CPU 1执行一个原子增量，使用“读取失效”来窥探CPU 2缓存中的数据并使其失效，这样CPU 1缓存中的副本就处于“修改”状态（内存中的副本仍然过期）。

最后，CPU 1读取地址8处的cache lines，该cache lines使用“写回”消息将地址0的数据推回内存。

请注意，我们以一些CPU缓存中的数据结束。

**快速测试5：**什么样的操作顺序会使CPU的缓存全部恢复到“无效”状态？

# 3 在非必要的暂停中写入结果

尽管图1所示的缓存结构为从给定CPU到给定数据项的重复读写提供了良好的性能，但它对给定cache lines的第一次写操作的性能相当差。要了解这一点，请考虑图4，它显示了CPU 0写入CPU 1高速缓存中的cache lines的时间线。由于CPU 0必须等待cache lines到达才能写入，因此CPU 0必须暂停较长时间。[3]

[[3]](#_ftnref3)将cache lines从一个CPU的缓存转移到另一个CPU的缓存所需的时间通常比执行简单的寄存器到寄存器指令所需的时间多几个数量级。

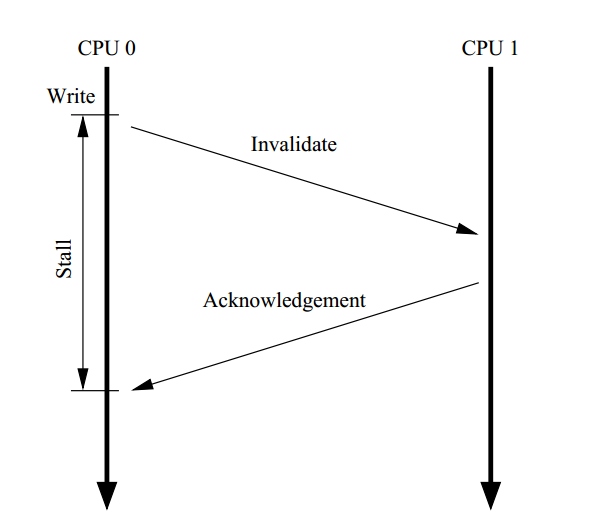
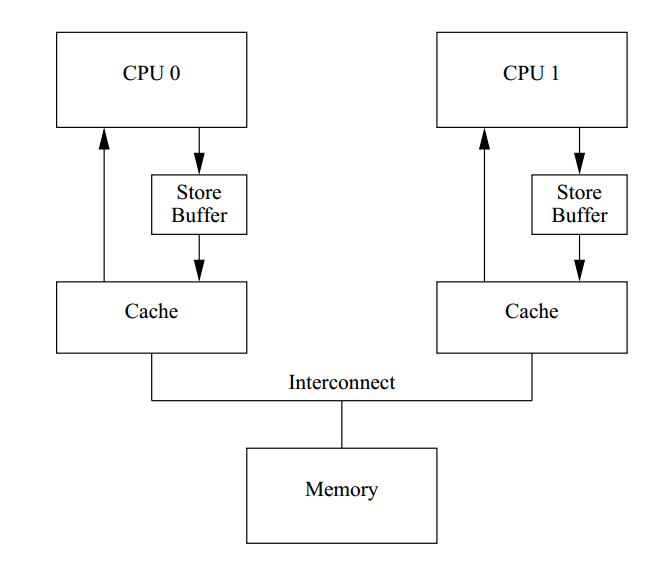


图4：writes see 不必要的暂停

但没有真正的理由强迫CPU 0暂停这么长时间——毕竟，不管CPU 1发送的cache lines中碰巧有什么数据，CPU 0都会无条件地覆盖它。(注：过程H就是这样，见上文)

## 3.1存储缓冲区

防止这种不必要的写暂停的一种方法是在每个CPU及其缓存之间添加“存储缓冲区”，如图5所示。通过添加这些存储缓冲区，CPU 0可以简单地在其存储缓冲区中记录写操作并继续执行。当cache lines最终从CPU 1移动到CPU 0时，数据将从存储缓冲区移动到cache lines。



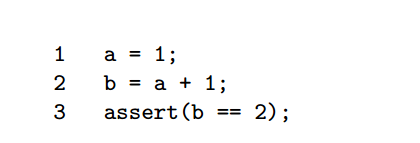
|  |
| --- |
|  |

图5：带有存储缓冲区的缓存

然而，还有一些复杂的问题必须解决，这将在接下来的两部分中介绍。

## 3.2 Store转发

若要查看第一个复杂度，即**违反自洽性**，请考虑以下变量：“A”和“B”两个代码，它们最初都为零，并且CPU 1中最初拥有的变量“A”，CPU 0中最初拥有的“B”的高速缓存行：



人们不会指望这一主张会失败。然而，如果一个人愚蠢到使用图5所示的非常简单的体系结构，他会感到惊讶。这样的系统可能会看到以下事件序列：

1.       CPU 0开始执行a=1。

2.       CPU 0在缓存中查找“a”，并发现它不见了。

3.       因此，CPU 0发送“读取无效”消息，以获取包含“a”的cache lines的独占所有权。

4.       CPU 0将存储记录到其存储缓冲区中的“a”。

5.       CPU 1接收到“读取无效”消息，并通过传输cache lines并从其缓存中移除该cache lines来响应。

6.       CPU 0开始执行b=a+1。

7.       CPU 0从CPU 1接收cache lines，该cache lines的“a”值为零。

8.       CPU 0从缓存中加载“a”，找到值0。(这时b == 1)

9.       CPU 0将其存储队列中的条目应用于新到达的cache lines，将其缓存中的“a”值设置为1。

10.   CPU 0在上面为“a”加载的值0上加1，并将其存储到包含“b”的cache lines中（我们假设它已经归CPU 0所有）。

11.   CPU 0执行断言（b==2），但失败。

问题是我们有两个“a”的副本，一个在缓存中，另一个在存储缓冲区中。

这个例子打破了一个非常重要的保证，即每个CPU总是能看到自己的操作，就好像它们是按程序顺序发生的一样。

这种保证严重违反了软件类型的直观性，以至于硬件人员同情并实现了“存储转发”，每个CPU在执行加载时都会引用（或“嗅探”）其存储缓冲区以及缓存，如图6所示。换句话说，给定CPU的存储直接转发到其后续加载，而无需通过缓存。

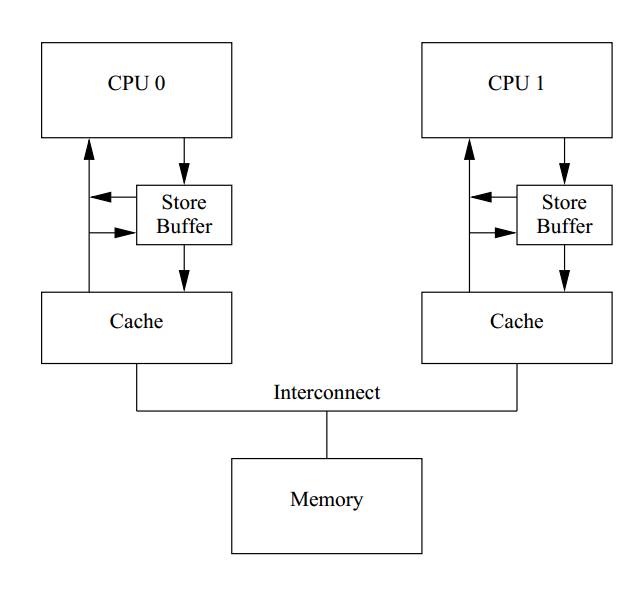
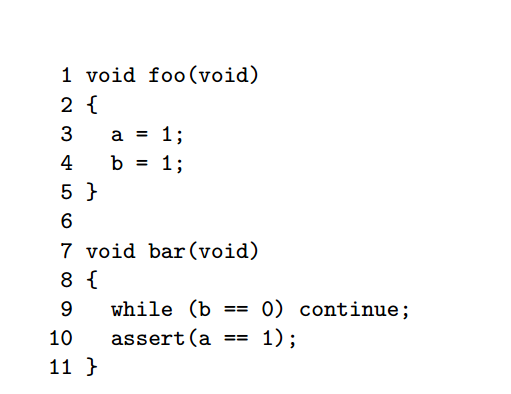


图6：Caches with store forwarding

在存储转发到位的情况下，上述序列中的第8项将在存储缓冲区中找到“a”的正确值1，因此“b”的最终值将是2，正如人们所希望的那样。(注：也就是说，cpu执行加载的时候，会把缓冲区和缓存结合起来，而不是忽略了已经在缓冲区中的信息，就可以保证上面的b == 2了)

## 3.3存储缓冲区和存储屏障

要查看第二个并发症，违反**全局内存排序**，考虑下面的代码序列，变量“A”和“B”最初为零：



假设CPU 0执行foo（），CPU 1执行bar（）。进一步假设包含“a”的cache lines只驻留在CPU 1的缓存中，包含“b”的cache lines归CPU 0所有。那么操作顺序可能如下所示：

1.    CPU 0执行a=1。cache lines不在CPU 0的缓存中，因此CPU 0将新值“a”放入其存储缓冲区，并传输“读取无效”消息。

2.    CPU 1在（b==0）继续时执行，但包含“b”的cache lines不在其缓存中。因此，它会传输一条“读取”信息。

3.    CPU 0执行b=1。它已经拥有这个cache lines（换句话说，cache lines已经处于“已修改”或“独占”状态），因此它在cache lines中存储新的“b”值。

4.    CPU 0接收到“读取”消息，并将包含当前更新值“b”的cache lines传输到CPU 1，同时在其自身的缓存中将该cache lines标记为“共享”。

5.    CPU 1接收包含“b”的cache lines，并将其安装到缓存中。

6.    CPU 1现在可以在（b==0）继续时完成执行，因为它发现“b”的值是1，所以它继续执行下一条语句。

7.    CPU 1执行断言（a==1），由于CPU 1使用的是旧值“a”，因此该断言失败。

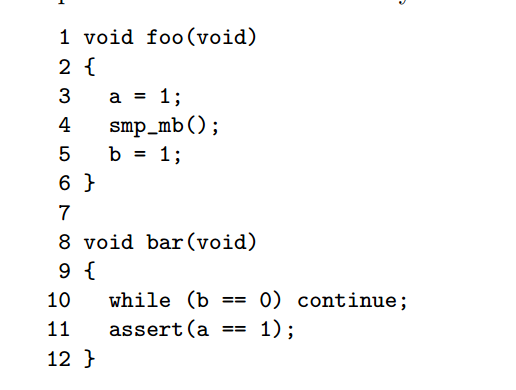
8.    CPU 1接收“read invalidate”消息，并将包含“a”的cache lines传输到CPU 0，并从其自身的缓存中使该cache lines失效。但为时已晚。

9.    CPU 0接收包含“a”的cache lines，并及时应用缓冲存储，以使其成为CPU 1失败断言的牺牲品。

**快速测试6：**在上面的步骤1中，为什么CPU 0需要发出“读取无效”而不是简单的“无效”？

**（注：因为可以让CPU1发出读取响应的消息，因为这时的a还被cpu1共享，要让cpu1也收到该消息）**

硬件设计师在这里无法直接提供帮助，因为CPU不知道哪些变量相关，更不用说它们可能如何相关。因此，硬件设计者提供内存屏障指令，允许软件告诉CPU这种关系。必须更新程序片段以包含内存屏障：



内存屏障smp\_mb（）将导致CPU在将后续存储应用到cache lines之前**刷新其存储缓冲区**。CPU可以简单地暂停，直到存储缓冲区清空后再继续，也可以使用存储缓冲区保存后续存储，直到应用存储缓冲区中的所有先前条目。

对于后一种方法，操作顺序可能如下所示：

1.       CPU 0执行a=1。cache lines不在CPU 0的缓存中，因此CPU 0将新值“a”放入其存储缓冲区，并传输“读取无效”消息。

2.       CPU 1在（b==0）继续时执行，但包含“b”的cache lines不在其缓存中。因此，它会传输一条“读取”信息。

3.       CPU 0执行smp\_mb（），并**标记所有当前存储缓冲区条目**（即a=1）。

4.       CPU 0执行b=1。它已经拥有这个cache lines（换句话说，cache lines已经处于“修改”或“独占”状态），但存储缓冲区中有一个标记的条目。因此，它不是将新值“b”存储在cache lines中，而是将其放在存储缓冲区中（但放在未标记的条目中）。（注：**这里开始和之前的操作有了区别，因为a有mb标记，而且没有被修改， 所以b无法执行存储1的操作，只能先把自己设置为共享，**见步骤5）

5.       CPU 0接收“读取”消息，并将包含原始值“b”的cache lines传输给CPU 1。它还将自己的cache lines副本标记为“共享”。(注：需要把b重新设置为共享状态)

6.       CPU 1接收包含“b”的cache lines，并将其安装到缓存中。

7.       CPU 1现在可以在（b==0）继续时完成执行，但由于它发现“b”的值仍然是0，所以它会重复while语句。新值“b”安全地隐藏在CPU 0的存储缓冲区中。

8.       CPU 1接收“read invalidate”消息，并将包含“a”的cache lines传输到CPU 0，并从其自身的缓存中使该cache lines失效。（注：这个时候才把a的缓存交给CPU0）

9.       CPU 0接收包含“a”的cache lines，并应用缓冲存储，将该cache lines置于“修改”状态。（a已经修改）

10.   由于存储到“a”是存储缓冲区中唯一由smp\_mb（）标记的条目，CPU 0还可以存储新的“b”值，但除了包含“b”的cache lines现在处于“共享”状态的事实。（注：终于可以修改b了，所以先把b的状态移除“共享”状态）

11.   因此，CPU 0向CPU 1发送一条“失效”消息。

12.   CPU 1接收“失效”消息，使其缓存中包含“b”的cache lines失效，并向CPU 0发送“确认”消息。

13.   CPU 1在（b==0）继续时执行，但包含“b”的cache lines不在其缓存中。因此，它向CPU 0发送“读取”消息。（注：很详细地流程，先发确认消息，让自己缓存失效，再发读取消息来获取新缓存）

14.   CPU 0接收“确认”消息，并将包含“b”的cache lines置于“独占”状态。CPU 0现在将新值“b”存储到cache lines中。（注：b的状态设置独占，才可以开始修改，终于修改了b的值，b = 1）

15.   CPU 0接收“读取”消息，并将包含新值“b”的cache lines传输给CPU 1。它还将自己的cache lines副本标记为“共享”。（注：传输后，b的值的状态重新设置为共享）

16.   CPU 1接收包含“b”的cache lines，并将其安装到缓存中。

17.   CPU 1现在可以在（b==0）继续时完成执行，因为它发现“b”的值是1，所以它继续执行下一条语句。

18.   CPU 1执行断言（a==1），但包含“a”的cache lines不再在其缓存中。一旦它从CPU 0获得这个缓存，它将使用最新的值“a”，因此断言将通过。

正如你所见，这个过程涉及到不小的簿记工作量。即使是直观上简单的事情，比如“加载a的值”，也可能涉及cpu计算单元中的许多复杂步骤。

# 4. 在非必要的暂停中存储有序的结果

不幸的是，每个存储缓冲区必须相对较小，这意味着执行适度存储序列的CPU可以填满其存储缓冲区（例如，如果所有存储都导致缓存未命中）。此时，CPU必须再次等待失效完成，以便在继续执行之前耗尽的存储缓冲区。当所有后续存储指令都必须等待失效完成时，无论这些存储是否导致缓存未命中，在内存屏障之后，也会立即出现同样的情况。

通过使无效确认消息更快到达，可以改善这种情况。实现这一点的一种方法是使用每个CPU的无效消息队列，或“无效队列”。

## 4.1使队列无效

使确认消息失效可能需要很长时间的一个原因是，它们必须确保相应的cache lines实际上已失效，并且如果缓存繁忙（例如，如果CPU正在集中加载和存储数据，所有数据都驻留在缓存中），则此失效可能会延迟。此外，如果大量无效消息在短时间内到达，则给定的CPU可能会在处理这些消息时落后，从而可能导致所有其他CPU暂停。

然而，在发送确认之前，CPU实际上不需要使cache lines失效。相反，它可以将invalidate消息排队，并理解在CPU发送关于该cache lines的任何进一步消息之前，将对该消息进行处理。

## 4.2使队列无效和使确认无效

图7显示了一个带有无效队列的系统。具有失效队列的CPU可以在消息放入队列后立即确认失效消息，而不必等到相应的line实际失效。当然，在准备传输失效消息时，CPU必须参考其失效队列——如果相应cache lines的条目在失效队列中，CPU无法立即传输失效消息；相反，它必须等到无效队列条目被处理完毕。

将条目放入失效队列本质上是CPU在传输任何有关该cache lines的MESI协议消息之前处理该条目的承诺。只要相应的数据结构不是很有竞争力，CPU就很少会因为这样的承诺而感到不便。

然而，可以在失效队列中缓冲失效消息这一事实提供了额外的好处

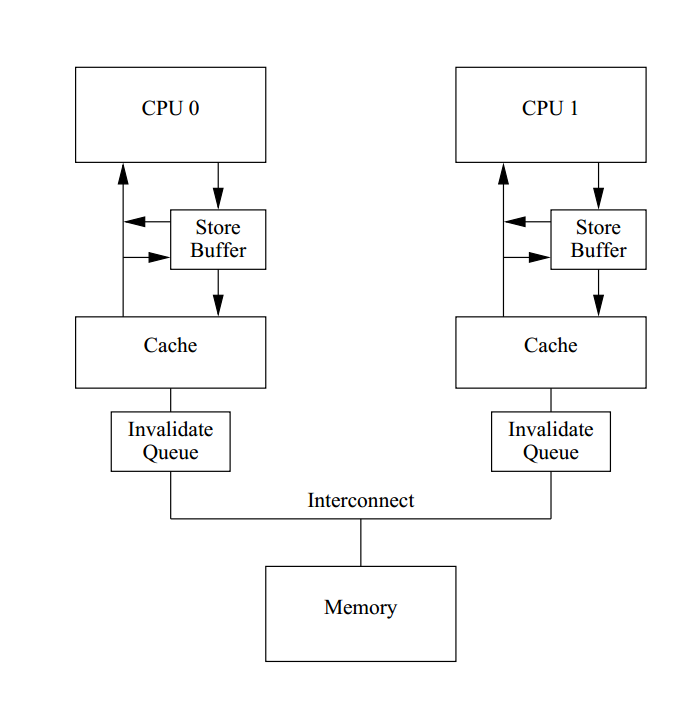


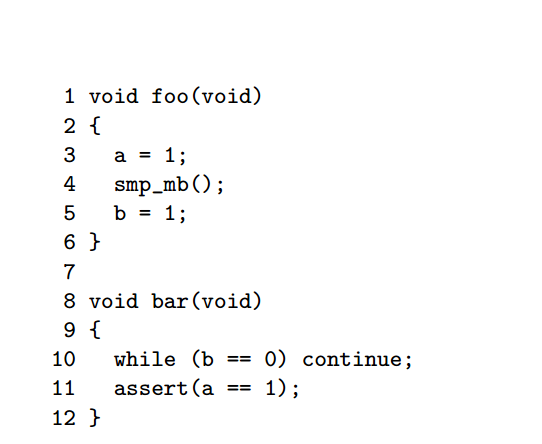
图7：带有失效队列的缓存

内存错误排序的机会，将在下一节中讨论。

## 4.3使队列和内存屏障失效

假设CPU将失效请求排队，但立即响应它们。这种方法将CPU进行存储时看到的缓存失效延迟降至最低，但会使内存屏障失效，如下示例所示。

假设“a”和“b”的值最初为零，“a”为只读复制（MESI“共享”状态）（注：**与之前的例子不同，a并不是在cpu1的缓存中**），而“b”为CPU 0所有（MESI“独占”或“修改”状态）。然后假设CPU 0执行foo（），而CPU 1执行以下代码片段中的函数bar（）：



那么操作顺序可能如下所示：

1.       CPU 0执行a=1。相应的cache lines在CPU 0的缓存中是只读的，因此CPU 0将新的“a”值放入其存储缓冲区，并传输“invalidate”消息，以便从CPU 1的缓存中刷新相应的cache lines。

2.       CPU 1在（b==0）继续时执行，但包含“b”的cache lines不在其缓存中。因此，它会传输一条“读取”信息。

3.       CPU 1接收CPU 0的“失效”消息，将其排队，并立即响应。（注： 应该是存储a失效的消息）

4.       CPU 0从CPU 1接收响应，因此可以自由地通过上面第4行的smp\_mb（），将“a”的值从其存储缓冲区移动到其cache lines。(注：**为什么会通过？**，因为CPU0已经告知了CPU1关于a失效的消息，而且CPU1因为使用了失效队列，所以立即予以响应，那么CPU0就可以名正言顺地通过了)

5.       CPU 0执行b=1。它已经拥有这个cache lines（换句话说，cache lines已经处于“已修改”或“独占”状态），因此它在cache lines中存储新的“b”值。

6.       CPU 0接收到“读取”消息，并将包含当前更新值“b”的cache lines传输到CPU 1，同时在其自身的缓存中将该cache lines标记为“共享”。

7.       CPU 1接收包含“b”的cache lines，并将其安装到缓存中。

8.       CPU 1现在可以在（b==0）继续时完成执行，因为它发现“b”的值是1，所以它继续执行下一条语句。

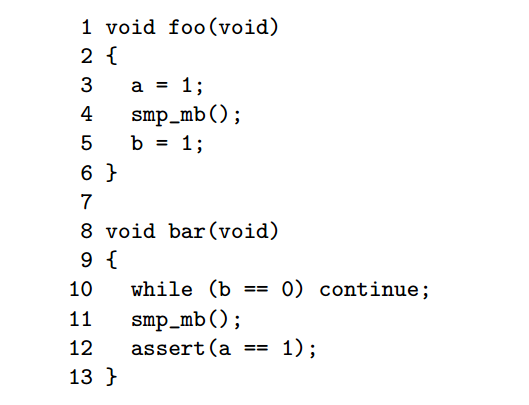
9.       CPU 1执行断言（a==1），由于“a”的旧值仍在CPU 1的缓存中，因此该断言失败。

10.   尽管断言失败，但CPU 1处理排队的“invalidate”消息，并（缓慢地）使其自身缓存中包含“a”的cache lines无效。

**快速测试7：**在第4.3节第一个场景的步骤1中，为什么发送“无效”消息而不是“读取无效”消息？CPU 0不需要与“a”共享此cache lines的其他变量的值吗？

**（注：好问题，因为这个时候a所在的缓存位置不是CPU1, 而是在CPU0中）**

如果这样做会导致内存屏障被有效忽略，那么加速失效反应显然没有多大意义。然而，memory-barrier指令可以与无效队列交互，因此当给定的CPU执行内存屏障时，它会标记其无效队列中当前的所有条目，并强制任何后续加载等待，直到所有标记的条目都应用到CPU的缓存中。因此，我们可以在函数bar上添加一个memory barrier，如下所示：



**快速测试8：**说什么？？？既然CPU在while循环完成之前不可能执行assert（），为什么我们需要一个内存屏障呢？？？

**（注： 用于清理失效队列中的数据）**

进行此更改后，操作顺序可能如下所示：

1.       CPU 0执行a=1。相应的cache lines在CPU 0的缓存中是只读的，因此CPU 0将新的“a”值放入其存储缓冲区，并传输“invalidate”消息，以便从CPU 1的缓存中刷新相应的cache lines。

2.       CPU 1在（b==0）继续时执行，但包含“b”的cache lines不在其缓存中。因此，它会传输一条“读取”信息。

3.       CPU 1接收CPU 0的“失效”消息，将其排队，并立即响应。（注： 应该是存储a失效的消息）

4.       CPU 0从CPU 1接收响应，因此可以自由地通过上面第4行的smp\_mb（），将“a”的值从其存储缓冲区移动到其cache lines。（注：为什么会通过？同上个例子）

5.       CPU 0执行b=1。它已经拥有这个cache lines（换句话说，cache lines已经处于“已修改”或“独占”状态），因此它在cache lines中存储新的“b”值。

6.       CPU 0接收到“读取”消息，并将包含当前更新值“b”的cache lines传输到CPU 1，同时在其自身的缓存中将该cache lines标记为“共享”。

7.       CPU 1接收包含“b”的cache lines，并将其安装到缓存中。

8.       CPU 1现在可以在（b==0）继续时完成执行，因为它发现“b”的值是1，所以它继续执行下一个语句，这现在是一个内存屏障。

9.       CPU 1现在必须暂停，直到它处理其失效队列中所有预先存在的消息。（预存的是a失效的消息）

10.   CPU 1现在处理排队的“invalidate”消息，并从其自身的缓存中使包含“a”的cache lines无效。

11.   CPU 1执行断言（a==1），由于包含“a”的cache lines不再在CPU 1的缓存中，因此它传输一条“读取”消息。

12.   CPU 0用包含新值“a”的缓存行响应此“读取”消息。

13.   CPU 1接收这个cache lines，其中包含“a”的avalue为1，因此断言不会触发。

通过大量传递MESI消息，CPU得到了正确的答案。本节说明了为什么CPU设计人员必须非常小心地优化缓存一致性。

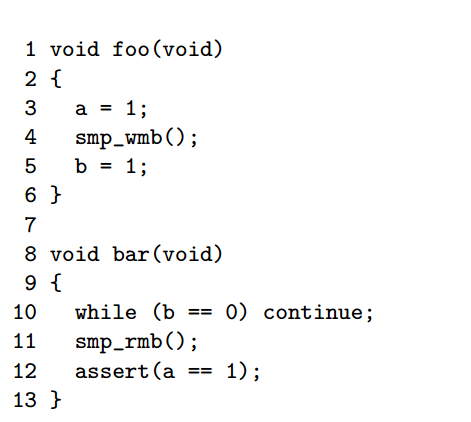
# 5. Read & Write Memory Barriers

在上一节中，内存屏障用于标记存储缓冲区和失效队列中的条目。但在我们的代码片段中，foo（）没有理由对invalidate队列做任何事情，bar（）也没有理由对store队列做任何事情。

因此，许多CPU架构提供的内存屏障指令较弱，只能执行这两种指令中的一种或另一种。粗略地说，**“读内存屏障”只标记无效队列，“写内存屏障”只标记存储缓冲区。而一个完整的内存屏障可以同时起到这两个作用。**

这样做的效果是，读内存屏障命令只在执行它的CPU上加载，因此读内存屏障之前的所有加载似乎都在读内存屏障之后的任何加载之前完成。类似地，写入内存屏障命令仅在执行它的CPU上存储，并且再次使写入内存屏障之前的所有存储看起来在写入内存屏障之后的任何存储之前完成。一个完整的内存屏障命令加载和存储，但同样只在执行内存屏障的CPU上。

如果我们将foo和bar更新为使用读写内存屏障，它们将显示如下：



有些计算机甚至有更多的内存屏障，但了解这三种变体将很好地介绍一般的内存屏障。

# 6 内存屏障序列示例

本节介绍了一些对内存屏障的诱人但微妙的突破。尽管它们中的许多将在大部分时间内工作，有些将在某些特定的CPU上一直工作，但如果目标是生成在所有CPU上可靠工作的代码，则必须避免这些使用。为了帮助我们更好地看到细微的破损，我们首先需要关注一个有序的体系结构。

## 6.1非有序的架构

保罗曾遇到过许多非有序的计算机系统的人，但这种逆性质一直非常微妙，理解它需要对特定硬件有详细的了解。与其选择某个特定的硬件供应商，不如让我们设计一个虚构的、但最大限度地利用内存的计算机体系结构，这可能是一个有吸引力的替代方案，而不是让读者浏览详细的技术规范。[4]

[[4]](#_ftnref4)希望详细了解真实硬件架构的读者，请参考CPU供应商手册[SW95、Adv02、Int02b、IBM94、LSH02、SPA94、Int04b、，

此硬件必须遵守以下订购约束[McK05a，McK05b]：

1.    每个CPU都将始终认为自己的内存访问是按程序顺序进行的。

2.    仅当两个操作引用不同的位置时，CPU才会对具有Store的给定操作重新排序。

3.    在读内存屏障（smp\_rmb（））之前的所有给定CPU负载将被所有CPU感知为在读内存屏障之后的任何负载之前。

4.    所有CPU都会认为写内存屏障（smp\_wmb（））之前的所有给定CPU存储位于该写内存屏障之后的任何存储之前。

5.    在一个完整的内存屏障（smp\_mb（））之前，一个给定CPU的所有访问（加载和存储）将被所有CPU感知为在该内存屏障之后的任何访问之前。

**快速测试9：**保证每个CPU按顺序看到自己的内存访问是否也保证每个用户级线程按顺序看到自己的内存访问？为什么？

（注：不能保证，因为有时序问题，必须加以干预）

设想一个大型非统一缓存体系结构（NUCA，non-uniform cache architecture）系统，为了向给定节点中的CPU公平分配互连带宽，在每个节点的互连接口中提供每个CPU队列，如图8所示。尽管给定CPU的访问顺序是由该CPU执行的内存屏障指定的，但是，正如我们将看到的，给定CPU对访问的相对顺序可能会被严重重新排序。[5].



Int04a，Int04c]，加拉霍洛的论文[Gha95]，或彼得·苏厄尔的作品[Sew]。

5毫无疑问，任何真正的硬件架构师或设计师都会在陶瓷对讲机上大声呼叫拉尔夫，因为他们可能只是对确定哪个队列应该处理涉及两个CPU都访问的cache lines的消息的前景感到有点不安，更不用说这个例子所带来的许多竞争了。我只能说“给我一个更好的例子”。

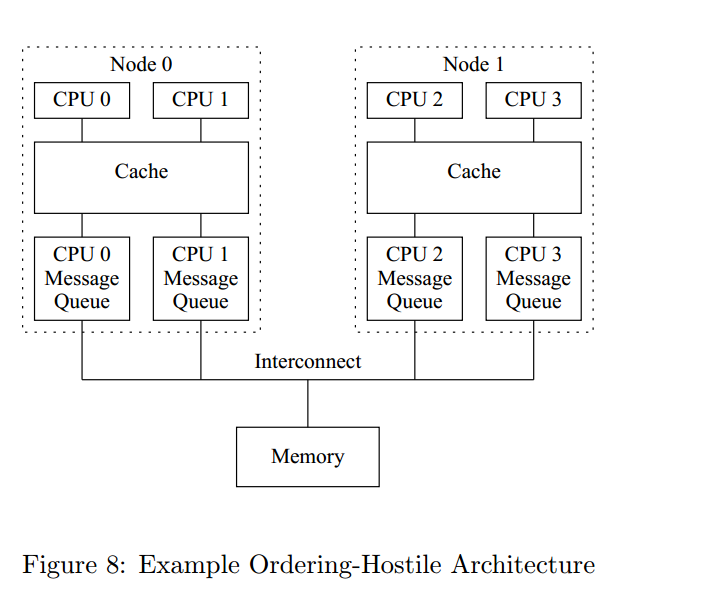


图8：非有序的架构示例

## 6.2示例1

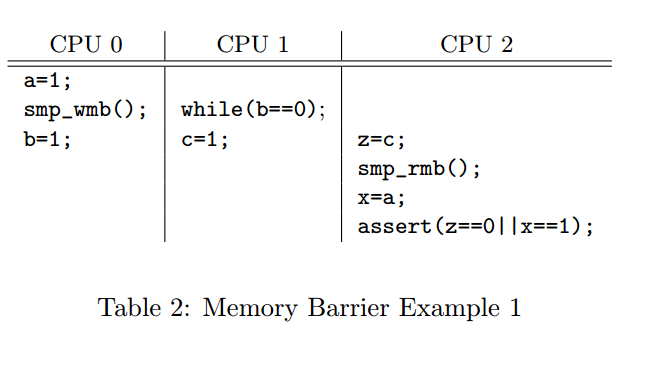


表2：内存屏障示例1

表2显示了由CPU 0、1和2并发执行的三段代码。“a”、“b”和“c”最初都是零。

假设CPU 0最近经历了多次缓存未命中，因此其消息队列已满，但CPU 1一直在缓存中独占运行，因此其消息队列为空。然后，CPU 0对“a”和“b”的分配将立即出现在节点0的缓存中（因此对CPU 1可见），但将被阻止在CPU 0之前的通信量之后。相比之下，CPU 1分配给“c”的任务将通过CPU 1以前的空队列。因此，在CPU 0分配给“a”之前，CPU 2可能会先看到CPU 1分配给“c”，从而触发断言，尽管存在内存屏障。

理论上，可移植代码不能依赖于这个示例代码序列，但实际上，它在所有主流计算机系统上都能工作。

**快速测试10：**通过在CPU 1的“while”和分配给“c”之间插入一个内存屏障，可以修复这个代码吗？为什么？

**(注：不行，CPU0的消息队列满了，所以cpu0中的写屏障失效。Cpu1中的内存屏障将自动通过)**

## 6.3示例2

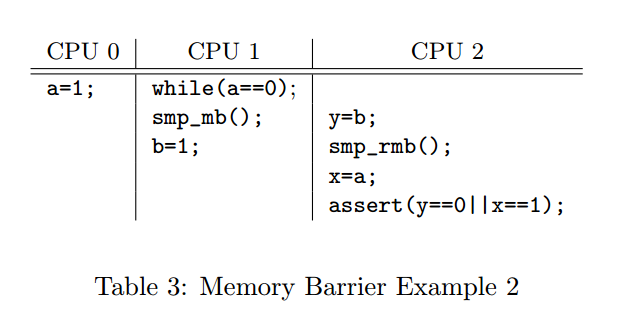


表3：内存屏障示例2

表3显示了由CPU 0、1和2并发执行的三段代码。“a”和“b”最初都是零。

同样，假设CPU 0最近经历了多次缓存未命中，因此其消息队列已满，但CPU 1一直在缓存中独占运行，因此其消息队列为空。然后，CPU 0对“a”的分配将立即出现在节点0的缓存中（因此对CPU 1可见），但将被阻止在CPU 0之前的流量后面。相比之下，CPU 1分配给“b”的任务将通过CPU 1以前的空队列。因此，在CPU 0分配给“a”之前，CPU 2很可能会先看到CPU 1分配给“b”，从而触发断言，尽管存在内存屏障。

从理论上讲，可移植代码不应该依赖于这个示例代码片段，然而，像以前一样，在实践中，它确实在大多数主流计算机系统上工作。

## 6.4示例3

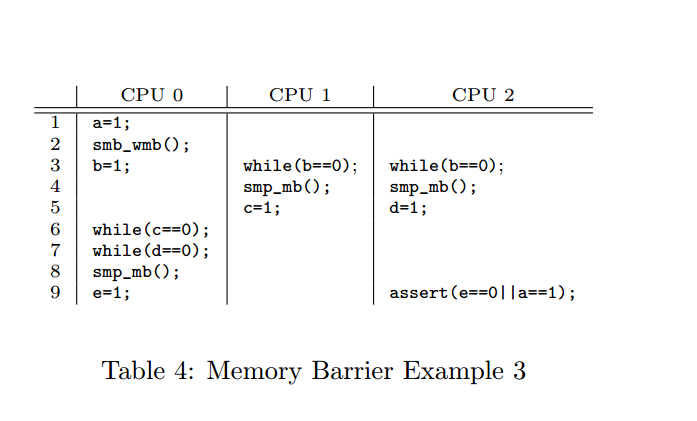


表4显示了由CPU 0、1和2并发执行的三段代码。所有变量最初都为零。

请注意，CPU 1和CPU 2都不能继续到第4行，直到它们看到CPU 0在第三行分配到“b”上。一旦CPU 1和2在第4行上执行了它们的内存屏障，它们都保证看到CPU 0在第2行上的内存屏障之前的所有分配。类似地，第8行的CPU 0的内存屏障与第4行的CPU 1和CPU 2的内存屏障成对，因此CPU 0不会执行第9行的“e”赋值，直到它对“a”的赋值对其他两个CPU都可见。因此，CPU 2第9行的断言保证不会触发。

**快速测试11：**假设CPU 1和2的第3-5行在中断处理程序中，CPU 2的第9行在进程级运行。为了使代码能够正确工作，换句话说，为了防止触发断言，需要进行哪些更改（如果有）？

Linux内核的synchronize\_rcu（）原语使用与本例中所示类似的算法。

# 7特定CPU的内存屏障说明

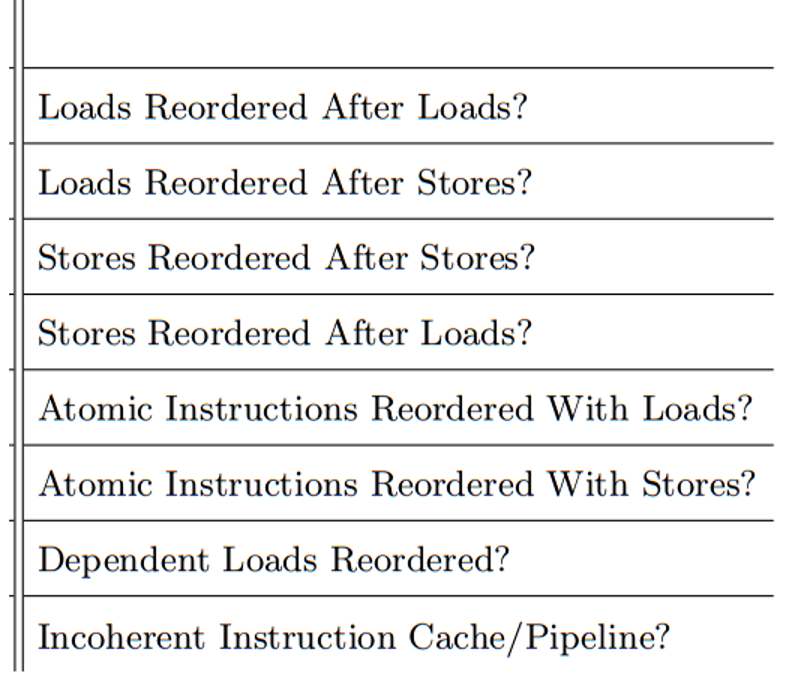
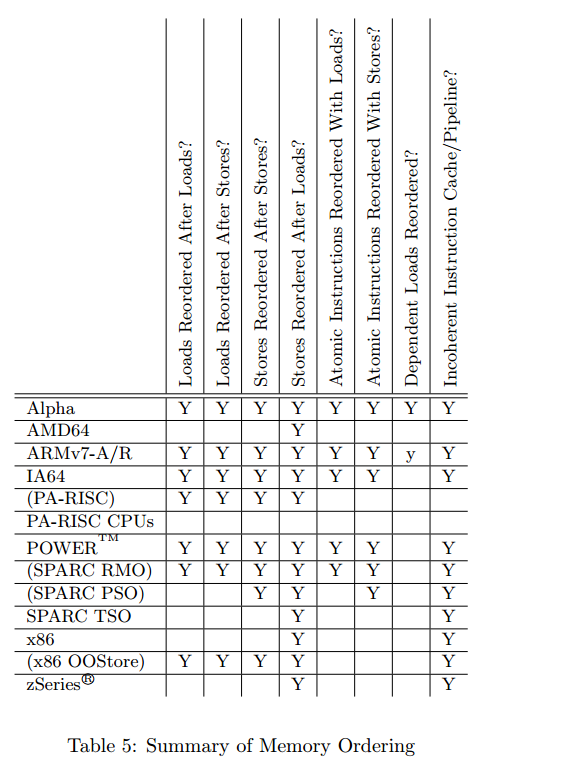


表5：内存排序摘要

如表5所示，每个CPU都有自己独特的内存屏障指令，这会使可移植性成为一个挑战。事实上，许多软件环境，包括pthreads和Java，只是禁止直接使用内存屏障，将程序员限制为互斥原语，这些原语在需要的程度上结合了它们。在表中，前四列表示给定CPU是否允许对四种可能的加载和存储组合进行重新排序。接下来的两列指示给定的CPU是否允许使用原子指令对加载和存储进行重新排序。

第七列“**相关读取重新排序**”需要一些解释，这将在下面介绍Alpha CPU的部分中进行。简短的版本是，Alpha需要为读卡器以及链接数据结构的更新程序设置内存屏障。是的，这确实意味着Alpha实际上可以在获取指针本身之前获取指向的数据，这很奇怪，但却是真的。请看：<http://www.openvms.compaq.com/wizard/wiz_2637.html>.

如果你认为我只是在编造。这种极为脆弱的内存模型的好处是，Alpha可以使用更简单的缓存硬件，从而在Alpha的全盛时期允许更高的时钟频率。

最后一列指示给定CPU是否具有不一致的指令缓存和管道。这样的CPU需要为自修正的代码执行特殊指令。

括号中的CPU名称表示架构上允许的模式，但在实践中很少使用。

对于内存屏障，常用的“**只说不（just say no）**”方法在适用的地方非常合理，但在某些环境中，例如Linux内核，需要直接使用内存屏障。因此，Linux提供了一组精心选择的最小公分母的内存屏障原语，如下所示：

•smpmb（）：“内存屏障”，它同时命令加载和存储。这意味着内存屏障之前的加载和存储将在内存屏障之后的任何加载和存储之前提交到内存。

•smprmb（）：“读取内存屏障”，仅使加载有序。

•smpwmb（）：“写入内存屏障”，仅使存储有序。

•smpread\_barrierdepends（）强制对依赖于先前操作的后续操作进行排序。这个原语在除Alpha之外的所有平台上都是不可操作的。

•mmiowb（），强制对受全局自旋锁保护的MMIO写入进行排序。这个原语在所有平台上都是不可操作的自旋锁中的内存屏障已经强制MMIO排序。具有non-no-op mmiowb（）定义的平台包括一些（但不是全部）IA64、FRV、MIPS和SH系统。这个原语相对较新，因此利用它的使用者相对较少。

smp mb（）、smp rmb（）和smp wmb（）原语还迫使编译器避免进行任何优化，这些优化会产生跨屏障重新排序内存优化的效果。smp read barrier dependens（）原语具有类似的效果，但仅在Alpha CPU上。

这些原语只在SMP内核中生成代码，然而，每种原语都有一个UP版本：

（分别是mb（）、rmb（）、wmb（）和read\_barrier\_depends（））

即使在UP内核中也会生成内存屏障。smp大多数情况下都应该使用版本。然而，这些后一种原语在编写驱动程序时很有用，因为MMIO访问必须保持有序，即使是在Up版内核中。在没有内存屏障指令的情况下，CPU和编译器都会很乐意重新安排这些访问，这最多只能使设备行为异常，并可能导致内核崩溃，有时甚至损坏硬件。

因此，大多数内核程序员不必担心每个CPU的内存屏障特性，只要他们坚持使用这些接口。当然，如果您正在深入研究给定CPU的特定于体系结构的代码，那么一切都是徒劳的。

此外，所有Linux的锁定原语（自旋锁、读写器锁、信号量、RCU等）包括任何需要的屏障原语。因此，如果您正在使用使用这些原语的代码，您甚至不需要担心Linux的内存排序原语。

也就是说，深入了解每个CPU的内存一致性模型在调试时非常有用，更不用说编写特定于体系结构的代码或同步原语了。

此外，他们说一点知识是非常危险的。想象一下，如果你有很多知识，你会造成多大的伤害！对于那些希望进一步了解单个CPU的内存一致性模型的人，下面几节将介绍最受欢迎和最著名的CPU。虽然没有什么可以取代实际阅读给定CPU的文档，但这些部分给出了一个很好的概述。

## 7.1 Alpha

对于一个已经宣布生命终结的CPU来说，说太多似乎有些奇怪，但Alpha很有趣，因为它的内存排序模型最弱，它对内存操作的排序最为激进。因此，它定义了Linux kernel内存排序原语，它必须在所有CPU上工作，包括Alpha。因此，了解Alpha对Linux内核黑客来说非常重要。

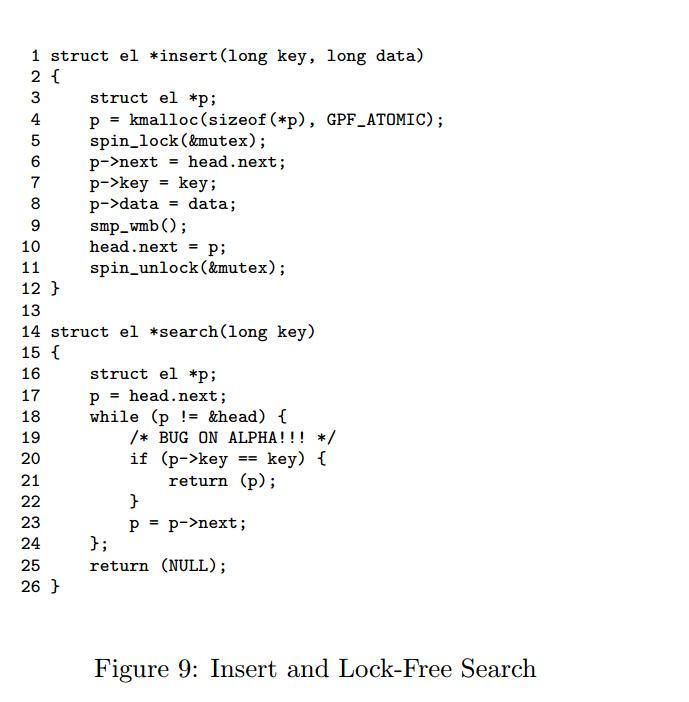


图9：插入和无锁搜索

图9所示的代码说明了Alpha和其他CPU之间的差异。此图第9行的smp wmb（）保证在将元素添加到第10行的列表之前执行第6-8行中的元素初始化，以便无锁搜索正常工作。也就是说，它对除Alpha之外的所有CPU都做出了保证。

Alpha的内存顺序非常弱，因此图9第20行的代码可以看到在第6-8行初始化之前存在的旧的垃圾值。

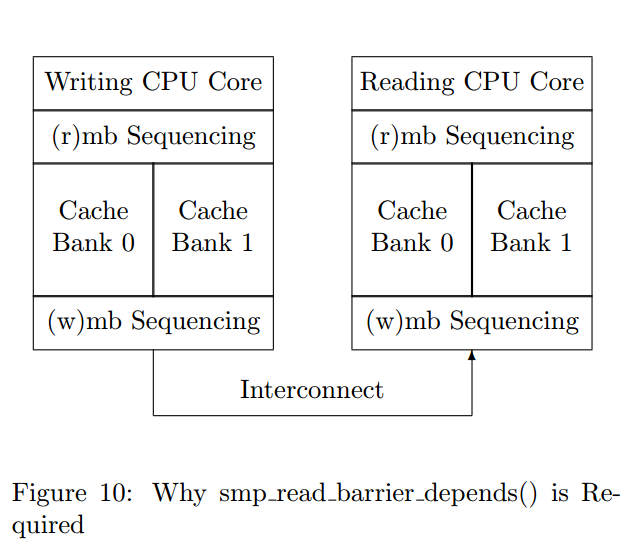


图10显示了如何在具有分区缓存的并行机器上实现这一点，以便交替的cache lines由缓存的不同分区处理。假设list header head将由缓存组0处理，新元素将由缓存组1处理。在Alpha上，smp wmb（）将保证图9第6-8行执行的缓存失效将在第10行到达互连之前到达互连，但绝对不能保证新值到达CPU Reading core的顺序。例如，reading CPU’s的缓存组1可能非常繁忙，但缓存组0空闲。这可能会导致新元素的缓存失效被延迟，因此reading CPU会获取指针的新值，但会看到新元素的旧缓存值。如果你认为这一切都是我编造出来的，你也可以去看看前面提到的网站，了解更多信息。[5]

[[5]](#_ftnref5)当然，精明的读者已经认识到Alpha远没有它可能的那么卑鄙和肮脏，（谢天谢地）第6.1节中的神奇架构就是一个很好的例子。

可以在指针fetch和dereference之间放置一个smp rmb（）原语。

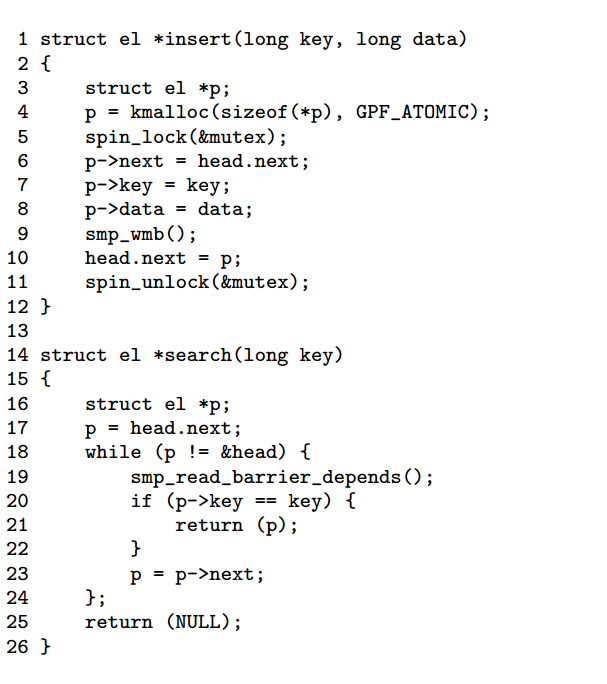


图11：安全插入和无锁搜索

然而，这给在读端且尊重数据依赖性的系统（如i386、IA64、PPC和SPARC）带来了不必要的开销。Linux 2.6内核中添加了smp read barrier dependes（）原语，以消除这些系统上的开销。该原语可如图11第19行所示使用。

还可以实现一个软件屏障来代替smp wmb（），它将强制所有reading CPU按顺序查看writing CPU的写入。然而，Linux社区认为这种方法会给极弱有序的CPU（如Alpha）带来过多的开销。这个软件屏障可以通过向所有其他CPU发送处理器间中断（IPI）来实现。在收到这样的IPI后，CPU将执行内存屏障指令，实现内存屏障拆除。需要额外的逻辑来避免死锁。当然，尊重数据依赖性的CPU会将这种屏障定义为smpwmb（）。也许这个决定应该在未来随着阿尔法逐渐消失在夕阳中重新考虑。

Linux内存屏障原语的名称取自Alpha指令.

因此smp mb（）是mb，smp rmb（）是rmb，smp wmb（）是wmb。Alpha是唯一一个CPU中，smp\_read\_barrierdepends（）是smp mb（）而不是no-op的。

有关Alpha的更多详细信息，请参阅参考手册[SW95]。

## 7.2 AMD64

AMD64与x86兼容，并且最近更新了其内存模型[Adv07]，以加强实际实现一段时间以来提供的更严格的顺序。Linux smp mb（）原语的AMD64实现是mfence，smp rmb（）是lfence，smp wmb（）是sfence。从理论上讲，这些可能会relaxed，但任何此类relaxed都必须考虑SSE和3DNOW指令。

## 7.3 ARMv7-A/R

ARM系列CPU在嵌入式应用程序中非常流行，尤其是在手机等功率受限的应用程序中。然而，ARM的多处理器实现已经有五年多了。其内存模型与Power类似（参见第7.6节，但ARM使用了一组不同的内存屏障指令[ARM10]：

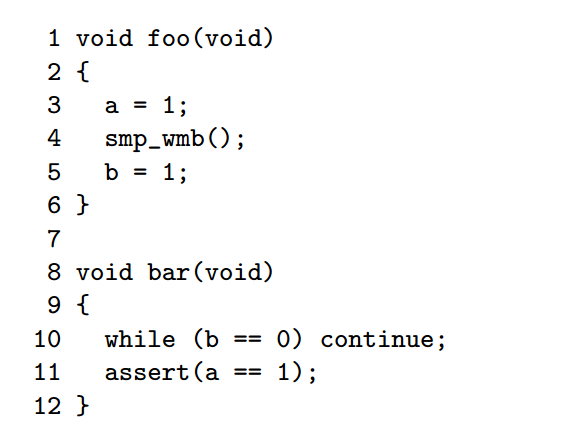
1.    DMB（数据存储屏障）使指定类型的操作在同一类型的任何后续操作之前完成。操作的“类型”可以是所有操作，也可以仅限于写操作（类似于Alpha wmb和POWER EIO指令）。此外，ARM允许缓存一致性具有三个作用域之一：单处理器、处理器子集（“内部”）和全局（“外部”）。

2.    DSB（数据同步屏障）使指定类型的操作在执行任何后续操作（任何类型）之前实际完成。操作的“类型”与DMB相同。在早期版本的ARM体系结构中，DSB指令被称为DWB（漏写缓冲区或数据写屏障，由您选择）。

3.    ISB（指令同步屏障）刷新CPU管道，以便只有在ISB完成后才能获取ISB后面的所有指令。例如，如果您正在编写一个self-modiying程序（例如JIT），那么您应该在生成代码和执行代码之间执行ISB。

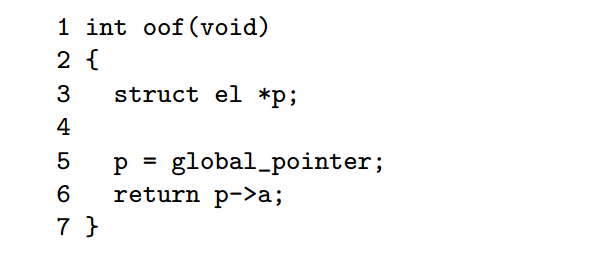
这些指令都与Linux的rmb（）原语的语义不完全匹配，因此必须将其实现为完整的DMB。DMB和DSB指令对屏障前后的访问顺序有递归定义，其效果类似于功率的累积性。

ARMv7和POWER内存模型之间的一个区别是，虽然POWER同时考虑数据和控制依赖性，但ARMv7只考虑数据依赖性。这两个CPU系列之间的差异可以在下面的代码片段中看到，这在前面的第4.3节中进行了讨论：



在上述示例中，第10行和第11行之间存在控件依赖关系。这种控制依赖性会导致POWER在这两行之间插入隐式内存屏障，但ARM会允许在第10行完成之前推测性地执行第11行。[6] 另一方面，这两个CPU都尊重以下代码中的数据依赖性：

[[6]](" \l "_ftnref6" \o ")当然，如上所述，编译器有权对第10行和第11行重新排序。所以请小心点！



两个CPU都会在第5行和第6行之间设置一个隐式内存屏障，避免第6行看到p->a的预初始化值。当然，假设编译器被阻止对这两行重新排序！

## 7.4 IA64

IA64提供了一个弱一致性模型，因此在没有显式内存屏障指令的情况下，IA64有权对内存引用进行任意重新排序[Int02b]。IA64有一个名为mf的内存围栏指令，但也有“半内存围栏”修饰符来加载、存储和它的一些原子指令[Int02a]。acq修饰符防止后续内存引用指令在acq之前重新排序，但允许之前的内存引用指令在acq之后重新排序，如图12所示。类似地，rel修饰符防止之前的内存引用指令在rel之后重新排序，但允许后续内存引用指令在rel之前重新排序。

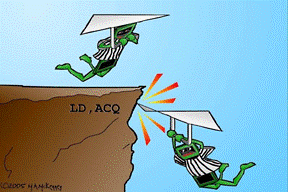


图12：半内存屏障

这些半内存围栏对于关键部分很有用，因为将操作推入关键部分是安全的，但如果让它们流出，则可能是致命的。然而，作为唯一具有此属性的CPU之一，IA64定义了与锁获取和释放相关的Linux内存顺序语义。

IA64 mf指令用于Linux内核中的smp rmb（）、smp mb（）和smp wmb（）原语。哦，尽管有相反的传言，“mf”mneumonic确实代表“内存栅栏”。

最后，IA64为“release”操作（包括“mf”指令）提供了全局总订单。这提供了及物性的概念，如果给定的代码片段会将给定的访问视为已发生，任何后续的代码片段也会将先前的访问视为已发生。假设所有涉及的代码片段都正确地使用了内存屏障。

## 7.5 PA-RISC

尽管PA-RISC体系结构允许对加载和存储进行完全重新排序，但实际的CPU运行完全有序[Kan96]。这意味着Linux内核的内存排序原语不会生成任何代码，但是，它们确实使用gcc内存属性来禁用编译器优化，这些优化会跨内存屏障重新排序代码。

## 7.6 POWER/Power PC

POWER和POWER PC CPU系列有多种内存屏障指令[IBM94，LSH02]：R

1.    sync 会导致之前的所有操作在任何后续操作开始之前都已完成。因此，这个指令相当昂贵。

2.    lwsync（轻型同步）针对后续loads和stores，以及orders stores。但是，它不会针对后续load 来order stores。有趣的是，lwsync指令执行与zSeries相同的顺序，巧合的是，SPARC TSO。

3.    EIO（如果您不知道的话，请按顺序执行I/O）会导致所有之前的可缓存存储在所有后续存储之前完成。但是，可缓存内存的存储与不可缓存内存的存储是分开排序的，这意味着EIO不会在释放自旋锁之前强制MMIO存储。

4.    isync强制所有前面的指令在任何后续指令开始执行之前看起来都已完成。这意味着前面的指令必须进展得足够快，它们可能产生的任何陷阱要么已经发生，要么保证不会发生，并且这些指令的任何副作用（例如，页表更改）都可以在后续指令中看到。

不幸的是，这些指令中没有一条与Linux的wmb（）原语完全一致，后者要求对所有存储进行排序，但不需要sync指令的其他高开销操作。但别无选择：wmb（）和mb（）的ppc64版本被定义为重量级同步指令。然而，Linux的smp wmb（）指令从未用于MMIO（毕竟，驱动程序必须小心地将MMIO与smp内核一起order），因此它被定义为重量较轻的eioio指令。这一指令在五元音助记符方面很可能是独一无二的。smp mb（）指令也被定义为sync指令，但smp rmb（）和rmb（）都被定义为轻量级lwsync指令。

幂的特征是“累积性”，可以用来获得及物性。如果使用得当，任何看到早期代码片段结果的代码也会看到该早期代码片段本身看到的访问。更多细节可从McKenney和Silvera[MS09]获得。

POWER体系结构的许多成员都有不一致的指令缓存，因此存储到内存不一定会反映在指令缓存中。谢天谢地，现在很少有人编写self-modifying代码，但JIT和编译器一直都在这样做。此外，从CPU的角度来看，重新编译最近运行的程序就像是self-modifying代码。icbi指令（指令缓存块失效）使指令缓存中指定的cache lines失效，并可在这些情况下使用。

## 7.7 SPARC RMO、PSO和TSO

SPARC上的Solaris使用TSO（total-store order），Linux在为“SPARC”32位体系结构构建时也使用TSO。然而，64位Linux内核（“sparc64”体系结构）在RMO（放松内存顺序）模式下运行SPARC[SPA94]。SPARC体系结构还提供了中间PSO（部分store order）。在RMO中运行的任何程序也将在PSO或TSO中运行，同样，在PSO中运行的程序也将在TSO中运行。向另一个方向移动共享内存并行程序可能需要小心地插入内存屏障，尽管如前所述，标准使用同步原语的程序不需要担心内存屏障，。

SPARC有一个非常灵活的内存屏障指令[SPA94]，允许对排序进行细粒度控制：

StoreStore：先order前一个store，再订购后一个store。（此选项由Linux smp wmb（）原语使用。）

LoadStore：在后续store之前order之前的load。

StoreLoad：在后续load之前order之前的store。

LoadLoad：在后续load之前对之前的load进行排序。（Linux smp rmb（）原语使用此选项。）

Sync：在开始任何后续操作之前，完全完成之前的所有操作。

MemIssue：在后续内存操作之前完成之前的内存操作，这对于内存映射I/O的某些实例很重要。

Lookaside：与MemIssue相同，但仅适用于之前的store和后续load，甚至仅适用于访问同一内存位置的store和load。

Linux smp mb（）原语同时使用前四个选项，如membar#LoadLoad|#LoadStore |#StoreStore |#StoreLoad，从而对内存操作进行完全排序。

那么，为什么需要membar#MemIssue呢？因为membar#StoreLoad可能会允许后续加载从写入缓冲区获取其值，如果写入MMIO寄存器会对要读取的值产生副作用，这将是灾难性的。相比之下，membar#MemIssue会等到写入缓冲区被刷新后才允许加载执行，从而确保加载实际上从MMIO寄存器获取其值。驱动程序可以使用membar#Sync，但如果不需要更昂贵的membar#Sync的附加功能，则更轻的membar#Memission是首选。

membar#Lookaside是membar#MemIssue的一个较轻版本，在写入给定的MMIO寄存器时非常有用，它会影响下一次从该寄存器读取的值。但是，当对给定MMIO寄存器的写入影响到下一个将从其他MMIO寄存器读取的值时，必须使用更重的membar#MemIssue。

目前尚不清楚SPARC为什么不将wmb（）定义为membar#MemIssue，而将smb wmb（）定义为membar#StoreStore，因为当前的定义似乎容易受到某些驱动程序中错误的影响。Linux运行的所有SPARC CPU很可能都实现了比体系结构所允许的更保守的内存排序模型。

SPARC要求在存储和执行指令之间使用flush指令[SPA94]。这是从SPARC的指令缓存中刷新该位置的任何先前值所必需的。请注意，flush接受一个地址，并且在SMP系统上只从指令缓存中刷新该地址。所有CPU的缓存都会被刷新，但是没有方便的方法来确定offCPU何时完成刷新，尽管有一个实现说明的参考。

## 7.8 x86

**由于x86 CPU提供“进程顺序”，因此所有CPU都同意给定CPU写入内存的顺序，因此smp wmb（）原语对于CPU[Int04b]来说是不可操作的。(注：自身就保持order)**但是，需要一个编译器指令来防止编译器执行可能导致smp wmb（）原语重新排序的优化。

另一方面，x86 CPU传统上不保证加载的顺序，因此smp mb（）和smp rmb（）原语扩展为lock；addl。这个原子指令对加载和存储起到了屏障的作用。

最近，英特尔发布了x86[Int07]的内存模型。事实证明，英特尔的实际CPU比之前的规范中要求的更严格，因此这种模式实际上只是强制执行早期的事实行为。甚至在最近，英特尔发布了x86[Int09]的更新内存模型，该模型要求a total global order for stores，尽管单个CPU仍被允许看到自己的stores比这个total global order显示的更早。为了实现涉及存储缓冲区的重要硬件优化，需要对总排序进行此例外。软件可能会使用原子操作来覆盖这些硬件优化，这是原子操作往往比非原子操作更昂贵的原因之一。在较旧的处理器上不能保证这一total store order。

但是，请注意，一些SSE指令是weakly ordered的（clflush和非时态移动指令[Int04a]）。具有SSE的CPU可以对smp mb（）使用mfence，对smp rmb（）使用lfence，对smp wmb（）使用sfence。

一些版本的x86 CPU有一个模式位，可以启用无序存储，对于这些CPU，smp wmb（）也必须定义为lock;addl.

尽管许多较旧的x86实现支持self-modifying 代码，而不需要任何特殊指令，但x86体系结构的较新版本不再要求x86 CPU具有如此大的适应性。有趣的是，这种放松正好及时给JIT实现者带来不便。

## 7.9 zSeries

zSeries机器构成了IBM大型机系列，以前称为360、370和390[Int04c]。并行性在zSeries中出现得较晚，但考虑到这些大型机是在20世纪60年代中期首次发布的，这并不能说明什么。bcr 15,0指令用于Linux smp mb（）、smp rmb（）和smp wmb（）原语。它还具有较强的内存排序语义，如表5所示，这应该允许smp wmb（）原语成为nop（在阅读本文时，这种变化很可能已经发生）。该表实际上低估了这种情况，因为zSeries内存模型在其他方面是顺序一致的，这意味着所有CPU将同意来自不同CPU的无关存储的顺序。

与大多数CPU一样，zSeries体系结构不能保证缓存一致的指令流，因此，self-modifying代码必须在更新指令和执行指令之间执行序列化指令。也就是说，许多实际的zSeries机器实际上可以在不序列化指令的情况下容纳self-modifying代码。zSeries指令集提供大量串行化指令，包括比较和交换、某些类型的分支（例如，前面提到的bcr 15,0指令）以及测试和设置等。

# 8.内存屏障永远存在吗？

最近出现了一些系统，它们对无序执行，尤其是对内存引用进行重新排序的攻击性要小得多。这种趋势是否会持续到内存屏障成为过去式的程度？

赞成的理由会引用提议的大规模多线程硬件架构，这样每个线程都会等到内存准备就绪，同时还有数十个、数百个甚至数千个线程在进行开发。在这样的体系结构中，不需要内存屏障，因为给定的线程只需等待所有未完成的操作完成，然后再继续执行下一条指令。因为可能会有数千个其他线程，CPU将被完全利用，因此不会浪费CPU时间。反对的理由是，能够扩展到1000个线程的应用程序数量极为有限，而且实时性要求也越来越严格，有些应用程序的实时性要求仅为几十微秒。Realtime-response需求很难满足现状，而且考虑到大规模多线程场景所隐含的极低单线程吞吐量，这一需求甚至更难满足。

另一个支持的论点是，越来越复杂的延迟隐藏硬件实现技术，这很可能允许CPU提供完全顺序一致执行的假象，同时仍然提供无序执行的几乎所有性能优势。另一种反驳是，电池驱动设备和环境责任都提出了越来越严格的能效要求。

谁是对的？我们没有线索，所以我们准备接受这两种情况。

# 9对硬件设计师的建议

**硬件设计师可以做很多事情来让软件人的生活变得困难**。下面列出了我们过去遇到的一些此类问题，希望有助于防止未来出现此类问题：

1.    忽略缓存一致性的I/O设备。

这种迷人的错误特性可能会导致DMA内存丢失最近对输出缓冲区的更改，或者，同样糟糕的是，在DMA完成后，输入缓冲区会被CPU缓存的内容覆盖。为了让系统在这种错误行为面前正常工作，在向I/O设备显示任何DMA缓冲区之前，必须小心地刷新任何位置的CPU缓存。即便如此，您也需要非常小心地避免指针错误，因为即使是对输入缓冲区的错误读取也可能导致数据输入损坏！

2.    忽略缓存一致性的设备中断。

这听起来可能很天真——毕竟，中断不是内存引用，是吗？但是，想象一下，一个CPU有一个分割缓存，其中一组非常繁忙，因此保留了输入缓冲区的最后一个cache lines。如果相应的I/O完成中断到达该CPU，则该CPU对缓冲区最后一条cache lines的内存引用可能会返回旧数据，再次导致数据损坏，但其形式在以后的崩溃转储中将不可见。当系统开始转储有问题的输入缓冲区时，DMA很可能已经完成。

3.    忽略缓存一致性的处理器间中断（IPI）。

如果IPI在相应消息缓冲区中的所有cache lines都提交到内存之前到达其目的地，这可能会有问题。

4.    上下文切换可以领先于缓存一致性。

如果内存访问可以无序地完成，那么上下文切换可能会非常痛苦。如果在源CPU可见的所有内存访问到达目标CPU之前，任务从一个CPU飞到另一个CPU，那么任务很容易看到相应的变量恢复到先前的值，这可能会致命地混淆大多数算法。

5.    过于友好的模拟器和仿真器。

很难编写模拟器或仿真器来强制内存重新排序，因此，在这些环境中运行良好的软件在第一次在真正的硬件上运行时可能会令人大吃一惊。不幸的是，硬件比模拟器和仿真器更狡猾仍然是一条规则，但我们希望这种情况能够改变。

同样，我们鼓励硬件设计师避免这些做法！

# 致谢

我感谢许多CPU架构师耐心地解释了他们CPU的指令和内存重新排序功能，特别是韦恩·卡多萨、埃德·西尔哈、安东·布兰查德、布拉德·弗雷、凯西·梅、德里克·威廉姆斯、蒂姆·斯莱格尔、尤尔根·普罗布斯特、英戈·阿德隆和拉维·阿里米利。Wayne值得特别感谢，因为他耐心地解释了Alpha对依赖负载的重新排序，这是我极力抵制的一个教训！我们都应该感谢戴夫·凯克和阿尔特姆·比特尤茨基，因为他们帮助人们阅读了这些材料。

# 法律声明

这项工作代表了作者的观点，并不一定代表IBM的观点。

IBM、zSeries和Power PC是国际商用机器公司（International Business Machines Corporation）在美国和/或其他国家/地区的商标或注册商标。Linux是Linus Torvalds的注册商标。

i386是英特尔公司或其子公司在美国、其他国家或两者的商标。

其他公司、产品和服务名称可能是此类公司的商标或服务标志。

# 快速测验的10个答案

**快速测试1：**

如果两个CPU试图同时使同一cache lines失效，会发生什么情况？

**答复:**

其中一个CPU首先获得对共享总线的访问权，该CPU“获胜”。另一个CPU必须使其cache lines副本失效，并向另一个CPU发送“失效确认”消息。当然，丢失的CPU可能会立即发出“读取无效”事务，因此获胜的CPU的胜利将是短暂的。

**快速测试2：**

当大型多处理器中出现“invalidate”消息时，每个CPU都必须给出“invalidate acknowledge”响应。由此产生的“无效确认”响应的“风暴”不会完全饱和系统总线吗？

**答复:**

如果大规模多处理器真的以这种方式实现的话，这可能会发生。大型多处理器，尤其是NUMA机器，倾向于使用所谓的“基于目录的”缓存一致性协议来避免这一问题和其他问题。

**快速测试3：**

如果SMP机器真的在使用消息传递，为什么还要使用SMP呢？

**答复:**

在过去的几十年里，关于这个话题有很多争议。一个答案是缓存一致性协议非常简单，因此可以直接在硬件中实现，**从而获得软件消息传递无法达到的带宽和延迟**。另一个答案是，由于大型SMP机器和小型SMP机器集群的相对价格，在经济学中可以找到真正的真相。第三个答案是，SMP编程模型比分布式系统的编程模型更易于使用，但反驳可能会注意到HPC集群和MPI的出现。所以争论还在继续。

**快速测试4：**

硬件如何处理上述延迟转换？

**答复:**

通常通过添加额外的状态，尽管这些额外的状态实际上不需要与cache lines一起存储，因为一次只有几行会转换。延迟转换的需要只是导致实际缓存一致性协议比本附录中描述的过于简化的MESI协议复杂得多的一个问题。轩尼诗和帕特森（Hennessy and Patterson）的经典计算机体系结构介绍[HP95]涵盖了其中许多问题。

**快速测试5：**

什么样的操作顺序会使CPU的缓存全部恢复到“无效”状态？

**答复:**

没有这样的序列，至少在CPU指令集中没有特殊的“刷新我的缓存”指令的情况下。大多数CPU都有这样的指令。

**快速测试6：**

在上面的步骤1中，为什么CPU 0需要发出“读无效”而不是简单的“无效”？

**答复:**

因为所讨论的cache lines包含的不仅仅是变量a。

**快速测试7：**

在第4.3节第一个场景的步骤1中，为什么发送“无效”消息而不是“读取无效”消息？CPU 0不需要与“a”共享此cache lines的其他变量的值吗？

**答复:**

CPU 0已经有了这些变量的值，因为它有一个包含“a”的cache lines的只读副本。因此，CPU 0需要做的就是使其他CPU丢弃其cache lines的副本。因此，“失效”消息就足够了。

**快速测试8：**

说什么？？？既然CPU在while循环完成之前不可能执行assert（），为什么我们需要一个内存屏障呢？？？

**答复:**

CPU可以自由地推测性地执行，这可以在while循环完成之前执行断言。**也就是说，一些弱序CPU尊重“控制依赖性”这样的CPU会在每个条件分支之后执行隐式内存屏障，比如终止while循环的分支。**然而，这个例子使用了一个显式的内存屏障，这在DEC Alpha上是必需的。

**快速测试9：**

保证每个CPU按顺序看到自己的内存访问是否也保证每个用户级线程按顺序看到自己的内存访问？为什么？

**答复:**

不，考虑线程从一个CPU迁移到另一个CPU的情况，以及目标CPU感知源CPU最近的内存操作乱序的情况。为了保持用户模式的健全性，内核黑客必须在**上下文切换路径中使用内存屏障**。然而，安全地进行上下文切换所需的锁定应该自动提供使用户级任务按顺序查看其自身访问所需的内存屏障。也就是说，如果你正在设计一个超级优化的调度器，无论是在内核还是在用户级别，请记住这个场景！

**快速测试10：**

通过在CPU 1的“while”和分配给“c”之间插入一个内存屏障，可以修复这个代码吗？为什么？

**答复:**

不会。这样的内存屏障只会强制对CPU 1进行本地排序。它不会影响CPU 0和CPU 1访问的相对顺序，因此断言仍然可能失败。然而，所有主流计算机系统都提供一种或另一种机制来提供“及物性”，这提供了直观的因果顺序：如果B看到了A访问的影响，而C看到了B访问的影响，那么C也必须看到A访问的影响。

**快速测试11：**

假设CPU 1和2的第3-5行在中断处理程序中，CPU 2的第9行在进程级运行。为了使代码能够正确工作，换句话说，为了防止触发断言，需要进行哪些更改（如果有）？

**答复:**

断言需要进行编码，以确保“e”的负载先于“a”的负载。在Linux内核中，可以使用barrier（）原语来实现这一点，其方式与前面示例中的断言中使用内存屏障的方式大致相同。

# 工具书类

[Adv02]先进的微型设备。AMD x86-64体系结构程序员手册第1-5卷，2002年。

[Adv07]先进的微型设备。AMD x86-64体系结构程序员手册第2卷：系统编程，2007年。

[ARM10]ARM有限公司。ARM架构参考手册：ARMv7-A和ARMv7-R *版本*, 2010.

[CSG99]大卫·E·卡勒、贾斯温德·帕尔·辛格和阿诺普·古普塔。并行计算机体系结构：一种硬件/软件方法。摩根·考夫曼，1999年。

[Gha95]Kourosh Gharachorloo。共享内存多处理器的内存一致性模型。技术报告CSL-TR-95-685，加利福尼亚州斯坦福大学电气工程和计算机科学系计算机系统实验室，1995年12月。可供选择：<http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-9.pdf> [查看日期：2004年10月11日]。

[HP95]约翰·L·轩尼诗和大卫·A·帕特森。计算机体系结构：一种定量方法。摩根·考夫曼，1995年。

[IBM94]IBM微电子和摩托罗拉。PowerPC微处理器系列：编程环境，1994年。

[Int02a]英特尔公司。英特尔安腾体系结构软件开发人员手册第3卷：指令集参考，2002年。

[Int02b]英特尔公司。英特尔安腾体系结构软件开发人员手册第3卷：系统体系结构，2002年。

[Int04a]英特尔公司。IA-32英特尔体系结构软件开发人员手册第2B卷：指令集参考，N-Z，2004年。可供选择：<ftp://download.intel.com/design/Pentium4/manuals/25366714.pdf> [查看日期：2005年2月16日]。

[Int04b]英特尔公司。IA-32英特尔体系结构软件开发人员手册第3卷：系统编程指南，2004年。可供选择：<ftp://download.intel.com/design/Pentium4/manuals/25366814.pdf> [查看日期：2005年2月16日]。

[Int04c]国际商用机器公司. z/架构操作原则。可供选择：<http://publibz.boulder.ibm.com/epubs/pdf/dz9zr003.pdf> [查看日期：2005年2月16日]，2004年5月。

|  |  |
| --- | --- |
| [Int07] | 英特尔公司。英特尔64体系结构内存订购白皮书，2007年。可访问：<http://develop.intel.com/products/processor/manuals/318147.pdf> [浏览：2007年9月7日]。 |
| [Int09] | 英特尔公司。英特尔64和IA- |

*32体系结构软件开发人员手册，第3A卷：系统编程指南，第1部分*, 2009. 可供选择：<http://download.intel.com/design/processor/manuals/253668.pdf> [查看日期：2009年11月8日]。

[Kan96]格里·凯恩。PA-RISC 2.0体系结构。惠普专业书籍，1996年。

[LSH02]迈克尔·莱昂斯、埃德·西哈和比尔·海伊。PowerPC存储模型和AIX编程。可访问：

<http://www-106.ibm.com/developerworks/eserver/articles/powerpc.html> [查看日期：2005年1月31日]、8月 2002.

保罗·E·麦肯尼。《现代微处理器中的内存排序》，第一部分，《Linux期刊》，1（136）：52–57，2005年8月。可访问：<http://www.linuxjournal.com/article/8211> [http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.09. 19a.pdf](http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.09.%2019a.pdf) [查看2007年11月30日]。

保罗·E·麦肯尼。现代微处理器中的内存排序，第二部分。《Linux期刊》，1（137）：78-822005年9月。可访问：<http://www.linuxjournal.com/article/8212> [http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.09. 19a.pdf](http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.09.%2019a.pdf) [查看2007年11月30日]。

|  |  |
| --- | --- |
| [MS09] | 保罗·E·麦肯尼和劳尔·西尔维拉。c/c++内存模型的电源实现示例。可供选择：<http://www.rdrop.com/users/paulmck/scalability/paper/N2745r。2009.02.27a.html> [查看日期：2009年4月5日]，2009年2月。 |
| [Sew] | 彼得·苏厄尔。多处理器程序的语义。可供选择： |

<http://www.cl.cam.ac.uk/~pes20/weakmemory/> [查看时间：2010年6月7日]。

[SPA94]斯巴克国际。SPARC体系结构手册，1994年。

[SW95]Richard L.Sites和Richard T.Witek。Alpha AXP架构。数字出版社，第二版，1995年。