内存障碍:软件黑客的硬件视角

保罗·麦肯尼 Linux 技术中心

paulmck@linux.vnet.ibm.com

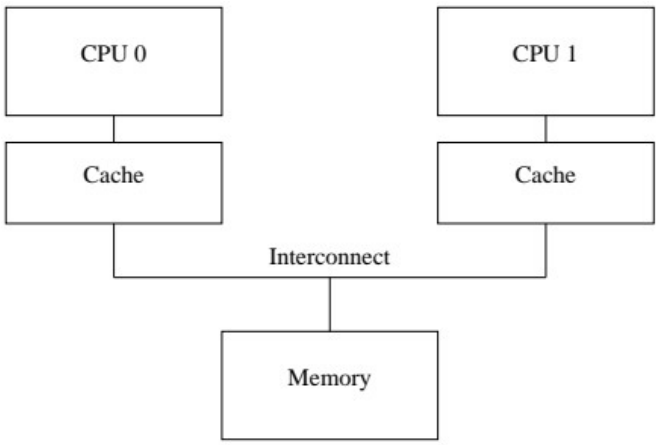
2010 年 7 月 23 日

那么，是什么让中央处理器设计者对毫无戒心的软件设计者设置内存障碍呢？

简而言之，因为对内存引用进行重新排序会带来更好的性能，所以需要内存屏障来强制对同步原语进行排序，这些原语的正确操作依赖于有序的内存引用。

要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，尤其是要让缓存真正正常工作需要什么。以下部分:

每纳秒 10 条指令，但是需要几十纳秒才能从主存储器中取出数据项。这种速度上的差异——超过两个数量级——导致了在现代处理器上发现的数兆字节的高速缓存。如图 1 所示，这些高速缓存与中央处理器相关联，通常可以在几个周期内访问。



1.呈现缓存的结构，

2.描述高速缓存一致性协议如何确保处理器同意内存中每个位置的值，最后，

3.概述存储缓冲区和无效队列如何帮助缓存和缓存一致性协议实现高性能。

我们将会看到，内存屏障是实现良好性能和可伸缩性所必需的一种邪恶，这种邪恶源于这样一个事实，即处理器比它们之间的接口和它们试图访问的内存快几个数量级。

图 1:现代计算机系统缓存结构

数据在中央处理器的高速缓存和内存之间以固定长度的块的形式流动，称为“高速缓存线”，其大小通常是 2 的幂，范围从 16 到 256 字节。当给定数据项第一次被

标准做法是使用多级高速缓存，一级高速缓存靠近中央处理器，只有一个周期的访问时间，二级高速缓存较大，访问时间较长，大约有 10 个时钟周期。更高性能的处理器通常有三级甚至四级缓存。

1 缓存结构

现代中央处理器比现代存储系统快得多。一个 2006 年的中央处理器可能能够执行-

1

给定一个中央处理器，它将从该中央处理器的高速缓存中消失，这意味着发生了“高速缓存未命中”(或者更具体地说，发生了“启动”或“预热”高速缓存未命中)。高速缓存未命中意味着当从内存中取出项目时，中央处理器将不得不等待(或“停止”)数百个周期。然而，该项目将被加载到该 CPU 的高速缓存中，以便后续的访问将在高速缓存中找到它，并因此全速运行。

一段时间后，CPU 的缓存将会填满，后续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造成的。但是，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表桶(或“集合”，如 CPU 设计者所称)，并且没有链接，如图 2 所示。

该高速缓存有 16 个“组”和两个“路”，总共 32 个“行”，每个条目包含一个 256 字节的“高速缓存行”，这是一个 256 字节对齐的内存块。这个高速缓存行的大小有点大，但是使得十六进制算法简单得多。用硬件术语来说，这是一个双向集合关联缓存，类似于一个有 16 个存储桶的软件哈希表，其中每个存储桶的哈希链最多只限于两个元素。大小(本例中为 32 条高速缓存线)和关联性(本例中为两条)统称为高速缓存的“几何图形”。由于这个缓存是在硬件中实现的，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尝试，它可以包含一个 256 字节的缓存行。然而，缓存条目可以是空的，如图中的空框所示。其余的框用它们包含的高速缓存行的内存地址来标记。由于高速缓存行必须是 256 字节对齐的，每个地址的低 8 位为零，选择硬件散列函数意味着下一个高 4 位匹配散列行号。

如果程序代码位于地址，可能会出现图中所示的情况

0xF

0xE

0xD

0xC

0xB

0xA

0x9

0x8

0x7

0x6

0x5

0x4

0x3

0x2

0x1

0x0

方式 0

0x12345E00

0x12345D00

0x12345C00

0x12345B00

0x12345A00

0x12345900

0x12345800

0x12345700

0x12345600

0x12345500

0x12345400

0x12345300

0x12345200

0x12345100

0x12345000

方式 1

0x43210E00

图 2:中央处理器高速缓存结构

0x43210E00 至 0x43210EFF，该程序从 0x12345000 至 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 0xF 行，并且该行的两个方向都是空的，因此可以容纳相应的 256 字节行。如果程序要访问 0x1233000 位置，该位置散列到 0x0 行，则相应的 256 字节高速缓存行可以容纳在方式 1 中。然而，如果程序要访问位置 0x1233E00，该位置散列到行 0xE，则必须从高速缓存中弹出一个现有行，以便为新的高速缓存行腾出空间。如果稍后访问该弹出行，将导致缓存未命中。这种高速缓存未命中被称为“关联性未命中”。

到目前为止，我们只考虑了 CPU 读取数据项的情况。当它写的时候会发生什么？因为重要的是所有的中央处理器都同意给定数据项的值，所以在给定的中央处理器写入该数据项之前，它必须首先使其从其他中央处理器的高速缓存中被移除或“无效”。一旦失效完成，中央处理器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，则该过程被称为“写未命中”。一旦给定的中央处理器完成了从其它中央处理器的高速缓存中使给定的数据项无效，该中央处理器可以重复-

2

无意识地写(和读)数据项。

稍后，如果另一个中央处理器试图访问该数据项，它将导致高速缓存未命中，这一次是因为第一个中央处理器为了向其写入而使该数据项无效。这种类型的高速缓存未命中被称为“通信未命中”，因为它通常是由几个使用数据项进行通信的处理器造成的(例如，锁是一个使用互斥算法在处理器之间进行通信的数据项)。

显然，必须非常小心地确保所有中央处理器保持数据的一致视图。通过所有这些获取、失效和写入，很容易想象数据丢失，或者(可能更糟)不同的处理器在其各自的高速缓存中对相同的数据项具有冲突的值。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 高速缓存一致性协议

高速缓存一致性协议管理高速缓存行状态，以防止不一致或丢失数据。这些协议可能非常复杂，有几十个状态，但是出于我们的目的，我们只需要关注四状态 MESI 缓存一致性协议。

2.1 MESI 国家

MESI 代表“修改的”、“排他的”、“共享的”和“无效的”，给定高速缓存行使用该协议可以采取的四种状态。因此，除了高速缓存行的物理地址和数据之外，使用该协议的高速缓存还在每个高速缓存行上维护一个两位状态“标签”。

处于“已修改”状态的行已经受到来自相应的中央处理器的最近存储器存储的影响，并且保证相应的存储器不会出现在任何其他中央处理器的高速缓存中。因此，处于“修改”状态的高速缓存行可以被称为“拥有”

分别参见第 670 页和第 671 页的九态图和 26 态图，分别用于 SGI Origin2000 和 Sequent(现在的 IBM) NUMA-Q。这两幅图都比现实生活简单得多。

中央处理器。因为这个缓存保存了数据的唯一最新副本，所以这个缓存最终有责任将它写回到内存中，或者将它移交给其他缓存，并且必须在重用这个行来保存其他数据之前这样做。

“独占”状态与“修改”状态非常相似，唯一的例外是高速缓存行尚未被相应的中央处理器修改，这又意味着高速缓存行驻留在存储器中的数据副本是最新的。然而，由于中央处理器可以在任何时候存储到该行，而无需咨询其他中央处理器，所以处于“独占”状态的行仍然可以说是由相应的中央处理器拥有的。也就是说，因为内存中的相应值是最新的，所以该缓存可以丢弃这些数据，而无需将其写回内存或交给其他某个 CPU。

处于“共享”状态的行至少可以复制到另一个 CPU 的缓存中，这样，在没有与其他 CPU 协商的情况下，就不允许该 CPU 存储到该行中。与“独占”状态一样，因为内存中的相应值是最新的，所以该缓存可以丢弃这些数据，而无需将其写回内存或交给其他某个 CPU。

处于“无效”状态的行是空的，换句话说，它不保存数据。当新数据进入高速缓存时，如果可能的话，它被放入处于“无效”状态的高速缓存行中。这种方法是首选的，因为如果将来引用被替换的行，替换任何其他状态的行都可能导致代价高昂的缓存未命中。

由于所有的处理器必须保持高速缓存行中携带的数据的一致视图，高速缓存一致性协议提供了协调高速缓存行在系统中的移动的消息。

2.2 MESI 协议消息

上一节中描述的许多转换都需要中央处理器之间的通信。如果中央处理器在一条共享总线上，以下信息就足够了:

读取:“读取”消息包含要读取的缓存行的物理地址。

3

阅读回复:“阅读回复”信息

包含早期“读取”消息所请求的数据。该“读取响应”消息可以由存储器或其他缓存提供。例如，如果其中一个缓存具有处于“修改”状态的所需数据，则该缓存必须提供“读取响应”消息。

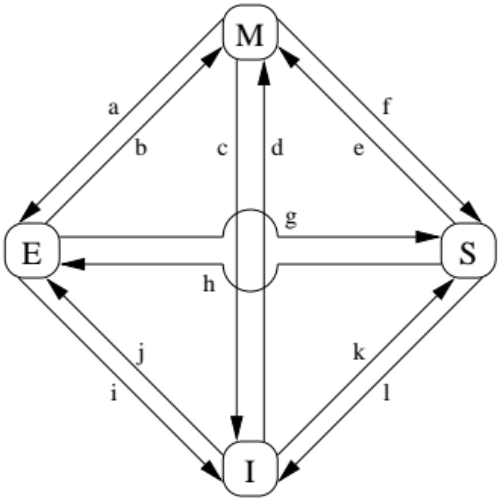
快速测试 3:如果 SMP 机器真的在使用消息传递，为什么还要麻烦 SMP 呢?

2.3 MESI 状态图

给定高速缓存行的状态随着协议消息的发送和接收而改变，如图 3 所示。

无效:无效消息包含

要作废的缓存行的物理地址。所有其他缓存必须从其缓存中移除相应的数据并做出响应。



无效确认:中央处理器接收到一个

从缓存中删除指定数据后,“invalidate”消息必须以“in-validate acknowledge”消息响应。

读取无效:“读取无效”消息

包含要读取的高速缓存行的物理地址，同时指示其他高速缓存删除数据。因此，它是一个“读”和“无效”的组合，用它的名字来表示。“读取无效”消息需要一个“读取响应”和一组“验证确认”消息作为回复。

图 3: MESI 缓存一致性状态图

写回:“写回”消息包含两者

地址和数据将被写回到内存中(也许还会“窥探”到其他处理器的缓存中)。该消息限制缓存弹出处于“修改”状态的行，以便为其他数据腾出空间。

该图中的过渡圆弧如下:

转换(A):缓存行被写回

内存，但 CPU 将它保留在其缓存中，并进一步保留修改它的权利。这种转换需要一个“写回”消息。

非常有趣的是，共享内存多处理器系统实际上是一个隐藏的信息传递计算机。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实现共享内存。

快速测试 1:如果两个处理器同时试图使同一个缓存行无效，会发生什么?

转换(b):中央处理器写入高速缓存行

它已经独占了。这种转换不需要发送或接收任何消息。

转换(c):中央处理器接收到一个“读无效”

已修改的缓存行的“日期”消息。中央处理器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发出请求的中央处理器，并指示它不再有本地副本。

快速测试 2:当大型多处理器中出现“无效”消息时，每个处理器必须给出“无效确认”响应。由此产生的“无效确认”响应的“风暴”难道不会完全浸透系统总线吗?

4

转换(d):中央处理器进行原子读取-

对缓存中不存在的数据项的修改-写入操作。它发送“读取无效”，通过“读取响应”接收数据。一旦收到一整套“无效确认”响应，中央处理器就可以完成转换。

“读取响应”和“无效交流知识”消息。

转换(j):这个中央处理器存储数据

不在其高速缓存中的高速缓存行中的项目，并因此传输“读取无效”消息。在收到“读取响应”和一整套“验证确认”消息之前，中央处理器无法完成转换。一旦实际存储完成，缓存行大概将通过转换(b)转换到“修改”状态。

转换(e):中央处理器进行原子读取-

对以前在其缓存中为只读的数据项的修改写操作。它必须传输“无效”消息，并且在完成转换之前必须等待一整套“无效确认”响应。

转换(k):这个中央处理器装入一个数据项

不在其缓存中的缓存行。中央处理器发送“读取”消息，并在接收到相应的“读取响应”时完成转换。

转换(f):其他一些中央处理器读取高速缓存

行，它由该 CPU 的缓存提供，该缓存保留一个只读副本，也可能将其写回内存。这种转换是通过接收“读取”消息来启动的，并且该中央处理器用包含所请求数据的“读取响应”消息来响应。

转换(1):其他一些中央处理器存储到

此高速缓存行中的数据项，但是由于它被保存在其他 CPU 的高速缓存中(例如当前 CPU 的高速缓存)，所以将此高速缓存行保持在只读状态。这种转换是由接收到“无效”消息开始的，这个中央处理器用“无效确认”消息作出响应。

转换(g):其他一些中央处理器读取一个数据项

在这个高速缓存行中，它或者由这个中央处理器的高速缓存提供，或者由存储器提供。在任一种情况下，该中央处理器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，该中央处理器用包含所请求数据的“读取响应”消息来响应。

快速测验 4:硬件如何处理上述延迟转换?

过渡(h):这个中央处理器意识到它将很快

需要写入该高速缓存行中的某个数据项，并因此传输“无效”消息。在收到一整套“无效确认”响应之前，中央处理器无法完成转换。或者，所有其它的中央处理器通过“写回”消息从它们的高速缓存中弹出这个高速缓存行(大概是为了给其它高速缓存行腾出空间)，因此这个中央处理器是最后一个高速缓存它的中央处理器。

过渡(一):其他一些中央处理器做原子

对高速缓存行中数据项的读-修改-写操作，该高速缓存行仅保存在该 CPU 的高速缓存中，因此该 CPU 使其从其高速缓存中失效。这种转换是通过接收“读入验证”消息来启动的，并且该中央处理器以以下方式响应

2.4 MESI 协议示例

现在让我们从高速缓存行的数据的角度来看这个问题，当它在四个 CPU 系统中的各种单行直接映射高速缓存中行进时，数据最初驻留在地址 0 处的存储器中。表 1 显示了这一数据流，第一列显示操作顺序，第二列显示执行操作的中央处理器，第三列显示正在执行的操作，接下来的四列显示每个中央处理器的高速缓存行的状态(存储器地址后跟 MESI 状态)，最后两列显示相应的存储器内容是否是最新的(“V”)或不是(“I”)。

最初，数据所在的 CPU 缓存行处于“无效”状态，并且数据在内存中有效。当中央处理器 0 在

5

地址 0，它在 CPU 0 的缓存中进入“共享”状态，并且在内存中仍然有效。中央处理器 3 也在地址 0 加载数据，因此在两个中央处理器的高速缓存中都处于“共享”状态，并且在存储器中仍然有效。接下来，CPU 0 加载一些其他的高速缓存行(在地址 8 处)，这通过无效将地址 0 处的数据从其高速缓存中强制出来，用地址 8 处的数据替换它。现在，CPU 2 从地址 0 进行加载，但是该 CPU 意识到它很快需要存储到它，因此它使用“读取无效”消息来获得独占拷贝，使它从 CPU 3 的缓存中无效(尽管内存中的拷贝保持最新)。接下来，中央处理器 2 进行预期的存储，将状态更改为“已修改”。内存中的数据副本现已过期。CPU 1 执行原子增量，使用“读取无效”来从 CPU 2 的高速缓存中窥探数据并使其无效，从而使 CPU 1 的高速缓存中的副本处于“修改”状态(并且内存中的副本保持过期)。最后，CPU 1 读取地址 8 处的高速缓存行，该行使用“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将中央处理器的缓存全部恢复到“无效”状态?

3 商店导致不必要的停顿

尽管图 1 所示的高速缓存结构为从给定的中央处理器到给定的数据项的重复读取和写入提供了良好的性能，但是它对给定的高速缓存行的第一次写入的性能非常差。要看到这一点，请考虑图 4，它显示了 CPU 0 向 CPU 1 的缓存中保存的缓存行写入的时间线。由于 CPU 0 必须等待高速缓存行到达后才能向其写入，因此 CPU 0 必须暂停一段时间。

将高速缓存行从一个中央处理器的高速缓存转移到另一个中央处理器的高速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。

中央处理器 0 中央处理器 1

写

确认

使无效

货摊

图 4:写入查看不必要的停顿

但是没有真正的理由强迫 CPU 0 停止这么长时间——毕竟，不管 CPU 1 发送的缓存行中有什么数据，CPU 0 都会无条件地覆盖它。

3.1 存储缓冲区

防止这种不必要的写入停顿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其写入并继续执行。当缓存行最终从 CPU 1 移动到 CPU 0 时，数据将从存储缓冲区移动到缓存行。

然而，有一些复杂的东西必须进行广告修饰，这将在接下来的两节中讨论。

3.2 商店转发

要查看第一个复杂性，即违反自我一致性，请考虑下面的代码，其中变量“a”和“b”最初都为零，并且包含变量“a”的缓存行最初由 CPU 1 拥有，包含“b”的缓存行最初由 CPU 0 拥有:

6

中央处理器高速缓存

顺序#中央处理器#操作 0 1 2 3 0 8

0 初始状态-/I-/I-/I-V

1 0 负载 0/S-/I-/I-V

2 3 负载 0/S -/I -/I 0/S 电压

3 0 失效 8/S -/I -/I 0/S V

4 2 RMW 8/S -/I 0/E -/I V

5 2 商店 8/S-/I 0/M-/I V

6 1 原子公司 8/s0/M-/I-/I-V

7 1 写回 8/S 8/S -/I -/I V

表 1:缓存一致性示例

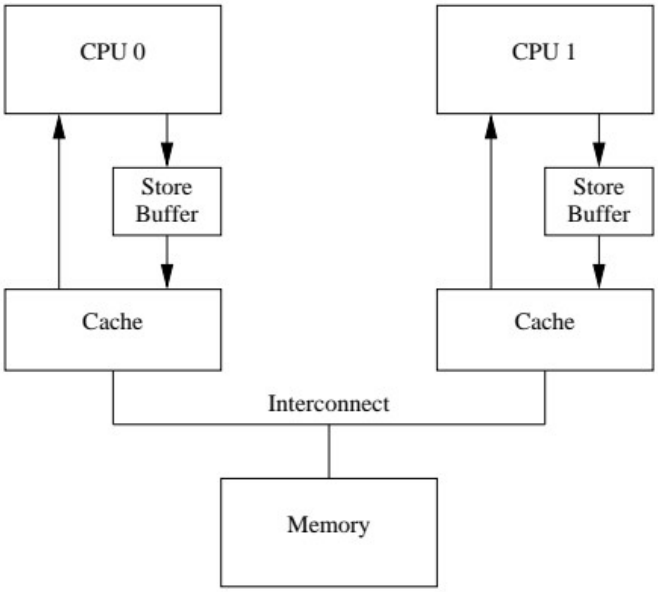


图 5:带存储缓冲区的缓存

它不见了。

3.因此，为了获得包含“a”的高速缓存行的独占所有权，CPU 0 发送“读取无效”消息。

4.CPU 0 在其存储缓冲区中将存储记录为“a”。

5.中央处理器 1 接收“读取无效”消息，并通过发送高速缓存行和从其高速缓存中移除该高速缓存行来进行响应。

6.CPU 0 开始执行 b=a+1。

7.中央处理器 0 从中央处理器 1 接收高速缓存行，该高速缓存行的“a”值仍然为零。

8.CPU 0 从其高速缓存中加载“a”，发现值为零。

9.CPU 0 将其存储队列中的条目应用于新到达的缓存行，将缓存中的值“a”设置为 1。

1a = 1;

2b = a+1;

3 断言(b = = 2);

10.CPU 0 在为上面的“a”加载的值 0 上加上 1，并将其存储到包含“b”的缓存行中(我们假设它已经为 CPU 0 所拥有)。

人们不会料到这一断言会失败。然而，如果一个人愚蠢到使用图 5 所示的非常简单的架构，他会感到惊讶。这样的系统可能会看到以下事件的顺序:

11.CPU 0 执行断言(b==2)，但失败。

问题是我们有“a”的两个副本，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，即每个中央处理器总是看到自己的操作，就好像它们是按程序顺序发生的一样。破坏

1.中央处理器 0 开始执行 a=1。

2.CPU 0 在缓存中查找“a”，并发现

7

这种保证与软件类型的直觉背道而驰，以至于硬件人员很同情并实现了“存储转发”，其中每个 CPU 在执行加载时都会引用(或“窥探”)其存储缓冲区和缓存，如图 6 所示。换句话说，一个给定的中央处理器的存储直接转发给它的后续加载，而不必通过高速缓存。

1 void foo(void) 2 {

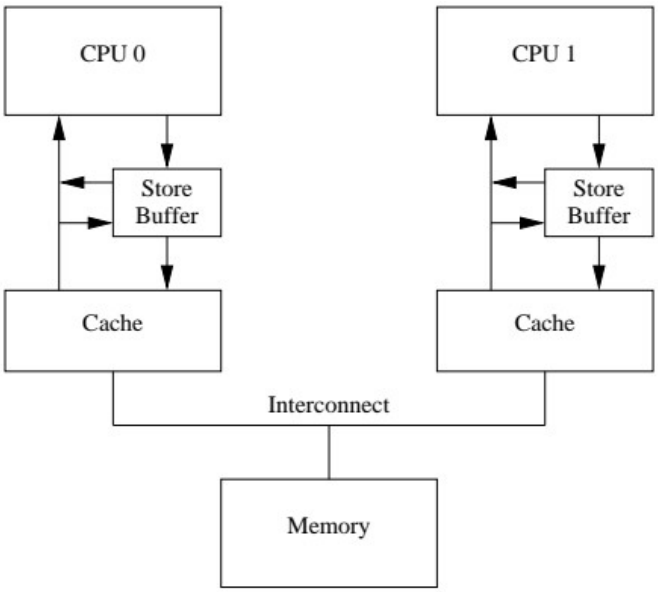
3a = 1;

4b = 1;

5 } 6

7 空心杆(空心)8 {

9，同时(b == 0)继续；10 断言(a = = 1)；11 }



假设 CPU 0 执行 foo()，CPU 1 执行 bar()。进一步假设包含“a”的高速缓存行仅驻留在 CPU 1 的高速缓存中，并且包含“b”的高速缓存行由 CPU 0 拥有。那么操作的顺序可能如下:

1.中央处理器 0 执行 a=1。高速缓存行不在 CPU 0 的高速缓存中，因此 CPU 0 在其存储缓冲区中放置新的值“a”，并发送“读取无效”消息。

2.当(b==0)继续时，CPU 1 执行，但是包含“b”的高速缓存行不在其高速缓存中。因此，它会发送一条“已读”消息。

3.CPU 0 执行 b=1。它已经拥有这个高速缓存行(换句话说，高速缓存行已经处于“修改”或“独占”状态)，所以它在其高速缓存行中存储新的值“b”。

图 6:带有存储转发的缓存

有了存储转发，上述序列中的第 8 项将在存储缓冲区中为“a”找到正确的值 1，因此“b”的最终值将是 2，正如人们所希望的那样。

3.3 存储缓冲区和内存屏障-

ers

要查看第二个复杂情况，违反全局内存排序，请考虑变量“a”和“b”最初为零的以下代码序列:

4.CPU 0 接收“读取”消息，并将包含现在更新的值“b”的高速缓存行传输到 CPU 1，还将该行标记为在其自己的高速缓存中“共享”。

5.CPU 1 接收包含“b”的高速缓存行，并将其安装在其高速缓存中。

6.当(b==0)继续时，CPU 1 现在可以完成执行，并且由于它发现“b”的值是 1，所以它进行到下一个语句。

7.CPU 1 执行断言(a==1)，由于 CPU 1 使用旧值“a”，该断言失败。

8.中央处理器 1 接收“读取无效”消息，并将包含“a”的高速缓存行发送到

8

CPU 0，并从其自己的缓存中使该缓存行无效。但为时已晚。

9.CPU 0 接收包含“a”的高速缓存行，并及时应用缓冲存储，以避免 CPU 1 的断言失败。

快速测验 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”？

硬件设计者在这方面不能直接提供帮助，因为中央处理器不知道哪些变量是相关的，更不用说它们是如何相关的了。因此，硬件设计者提供内存屏障指令，允许软件告诉中央处理器这种关系。必须更新程序片段以包含内存屏障:

1 void foo(void) 2 {

3a = 1;

4 SMP _ MB()；5b = 1；6 } 7

8 空心杆(空心)9 {

10，同时(b == 0)继续；11 断言(a = = 1)；12 }

内存屏障 smp_mb()将导致 CPU 在将每个后续存储应用到其变量的缓存行之前刷新其存储缓冲区。在继续之前，中央处理器可以简单地暂停，直到存储缓冲区为空，或者它可以使用存储缓冲区来保存后续存储，直到存储缓冲区中的所有先前条目都被应用。

采用后一种方法，操作顺序如下:

1.中央处理器 0 执行 a=1。高速缓存行不在 CPU 0 的高速缓存中，因此 CPU 0 在其存储缓冲区中放置新的值“a”，并发送“读取无效”消息。

2.当(b==0)继续时，CPU 1 执行，但是包含“b”的高速缓存行不在其高速缓存中。因此，它会发送一条“已读”消息。

3.CPU 0 执行 smp_mb()，并标记所有当前的存储缓冲区条目(即 a=1)。

4.CPU 0 执行 b=1。它已经拥有这个缓存行(换句话说，缓存行已经处于“修改”或“独占”状态)，但是在存储缓冲区中有一个标记的条目。因此，它不是将新值“b”存储在缓存行中，而是将其放在存储缓冲区中(但在未标记的条目中)。

5.CPU 0 接收“读取”消息，并将包含初始值“b”的缓存行传输到 CPU 1。它还将自己的缓存行副本标记为“共享”。

6.CPU 1 接收包含“b”的高速缓存行，并将其安装在其高速缓存中。

7.CPU 1 现在可以加载“b”的值，但是因为它发现“b”的值仍然是 0，所以它重复 while 语句。新值“b”被安全地隐藏在 CPU 0 的存储缓冲区中。

8.中央处理器 1 接收“读取无效”消息，并将包含“a”的高速缓存行传输到中央处理器 0，并使该高速缓存行从其自己的高速缓存中无效。

9.CPU 0 接收包含“a”的高速缓存行，并应用缓冲存储，将该行置于“修改”状态。

10.因为存储到“a”是存储缓冲区中唯一由 smp_mb()标记的条目，所以 CPU 0 也可以存储新的值“b”——除了包含“b”的高速缓存行现在处于“共享”状态的事实。

11.因此，中央处理器 0 向中央处理器 1 发送“无效”消息。

12.CPU 1 接收“无效”消息，从其高速缓存中验证包含“b”的高速缓存行，并向 CPU 0 发送“确认”消息。

13.当(b==0)继续时，CPU 1 执行，但是包含“b”的高速缓存行不在其高速缓存中。因此，它会向 CPU 0 发送一条“已读”消息。

14.CPU 0 接收“确认”消息，并将包含“b”的高速缓存行置于“独占”状态。CPU 0 现在将新的值“b”存储到缓存行中。

4.1 无效队列

无效确认消息可能花费如此长时间的一个原因是，它们必须确保相应的高速缓存行实际上是无效的，并且如果高速缓存繁忙，例如，如果中央处理器正在集中加载和存储数据，所有这些数据都驻留在高速缓存中，则该无效可能被延迟。此外，如果大量无效消息在短时间内到达，给定的中央处理器可能会在处理它们时落后，从而可能使所有其他中央处理器停滞不前。

15.CPU 0 接收“读取”消息，并将包含新值“b”的高速缓存行传输到 CPU 1。它还将自己的缓存行副本标记为“共享”。

16.CPU 1 接收包含“b”的高速缓存行，并将其安装在其高速缓存中。

17.CPU 1 现在可以加载“b”的值，因为它发现“b”的值是 1，所以它退出 while 循环并进入下一个语句。

然而，在发送确认之前，CPU 实际上不需要使高速缓存行无效。相反，它可以将无效消息放入队列，并理解在中央处理器发送任何关于该高速缓存行的进一步消息之前，将对该消息进行处理。

18.CPU 1 执行断言(a==1)，但是包含“a”的高速缓存行不再在其高速缓存中。一旦它从 CPU 0 获得这个缓存，它将使用最新的值“a”，因此断言通过。

4.2 无效队列和无效确认

正如你所看到的，这个过程涉及到大量的簿记。即使是一些直观上简单的东西，比如“加载 a 的值”，也可能涉及很多复杂的硅步骤。

图 7 显示了一个带有无效队列的系统。具有无效队列的中央处理器可以在无效消息被放入队列后立即确认该消息，而不必等到相应的行被实际无效。当然，当准备传送验证中消息时，中央处理器必须参考其无效队列——如果对应高速缓存行的条目在无效队列中，则中央处理器不能立即传送无效消息；它必须等待直到无效队列条目被处理。

4 存储序列导致

不必要的摊位

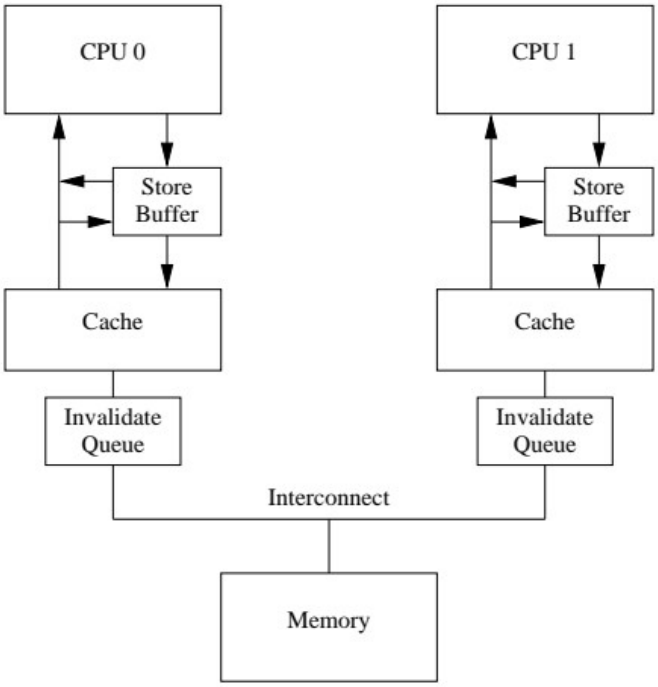
不幸的是，每个存储缓冲区必须相对较小，这意味着执行适度存储序列的中央处理器可以填充其存储缓冲区(例如，如果它们都导致高速缓存未命中)。此时，在继续执行之前，中央处理器必须再次等待无效指令完成，以便耗尽其存储缓冲区。当所有后续的存储指令都必须等待失效完成，而不管这些存储是否会导致高速缓存未命中时，这种相同的情况会在内存屏障之后立即出现。

这种情况可以通过使无效日期确认消息更快到达来改善。实现这一点的一种方法是使用每个 CPU 的无效消息队列，即“无效队列”。

将一个条目放入无效队列本质上是一个承诺，由中央处理器在传输任何 MESI 协议消息到该高速缓存行之前处理该条目。只要相应的数据结构不是高度竞争的，CPU 就很少会因为这样的承诺而感到不便。

然而，无效消息可以缓冲在无效队列中的事实为内存混乱提供了额外的机会，这将在下一节中讨论。

10



1 void foo(void) 2 {

3a = 1；

4 SMP _ MB()；5b = 1；6 } 7

8 空心杆(空心)9 {

10，同时(b == 0)继续；11 断言(a = = 1)；12 }

那么操作的顺序可能如下

低:

1.中央处理器 0 执行 a=1。相应的高速缓存行在 CPU 0 的高速缓存中是只读的，因此 CPU 0 在其存储缓冲区中放置新的值“a”，并发送“无效”消息，以便从 CPU 1 的高速缓存中刷新相应的高速缓存行。

图 7:带有无效队列的缓存

4.3 无效队列和内存障碍

让我们假设中央处理器对无效请求进行排队，但是要立即响应它们。这种方法最大限度地减少了执行存储的中央处理器所经历的缓存失效延迟，但也可以消除内存障碍，如下例所示。

假设“a”和“b”的值最初为零，“a”被复制为只读(MESI“共享”状态)，而“b”被 CPU 0 拥有(MESI“排除”或“修改”状态)。然后假设 CPU 0 执行 foo()，而 CPU 1 执行下面代码片段中的函数条()。

2.当(b==0)继续时，CPU 1 执行，但是包含“b”的高速缓存行不在其高速缓存中。因此，它会发送一条“已读”消息。

3.CPU 1 接收 CPU 0 的“无效”消息，将其排队，并立即对其做出响应。

4.CPU 0 接收来自 CPU 1 的响应，因此可以自由地越过上面第 4 行的 smp_mb()，将值“a”从其存储缓冲区移到其缓存行。

5.CPU 0 执行 b=1。它已经拥有这个高速缓存行(换句话说，高速缓存行已经处于“修改”或“独占”状态)，所以它在其高速缓存行中存储新的值“b”。

6.CPU 0 接收“读取”消息，并将包含现在更新的值“b”的高速缓存行传输到 CPU 1，还将该行标记为在其自己的高速缓存中“共享”。

7.CPU 1 接收包含“b”的高速缓存行，并将其安装在其高速缓存中。

8.当(b==0)继续时，CPU 1 现在可以完成执行，并且由于它发现“b”的值是 1，所以它进行到下一个语句。

11

9.CPU 1 执行断言(a==1)，由于旧的值“a”仍在 CPU 1 的缓存中，该断言失败。

10.尽管断言失败，但 CPU 1 处理排队的“无效”消息，并(缓慢地)使其自己的高速缓存中包含“a”的高速缓存行无效。

快速测试 7:在第 4.3 节第一个场景的第 1 步中，为什么发送“无效”消息而不是“读取无效”消息？难道 CPU 0 不需要与“a”共享该缓存行的其他变量的值吗？

如果加速验证中的响应会导致内存障碍被有效忽略，那么这显然没有多大意义。但是，内存屏障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，它会标记当前在其无效队列中的所有条目，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，我们可以在功能栏中添加一个内存屏障，如下所示:

1 void foo(void) 2 {

3a = 1；

4 SMP _ MB()；5b = 1；6 } 7

8 空心杆(空心)9 {

10，同时(b == 0)继续；11 SMP _ MB()；

12 断言(a = = 1)；13 }

快速测验 8:说什么？？？既然在 while 循环完成之前，CPU 不可能执行 assert()，为什么我们在这里需要内存屏障呢？？？

随着这一变化，操作顺序可能如下:

1.中央处理器 0 执行 a=1。对应的缓存行在 CPU 0 的缓存中是只读的，因此 CPU 0 在其存储缓冲区中放置新的值“a”

发送“无效”消息，以便从 CPU 1 的高速缓存中刷新相应的高速缓存行。

2.当(b==0)继续时，CPU 1 执行，但是包含“b”的高速缓存行不在其高速缓存中。因此，它会发送一条“已读”消息。

3.CPU 1 接收 CPU 0 的“无效”消息，将其排队，并立即对其做出响应。

4.CPU 0 接收来自 CPU 1 的响应，因此可以自由地越过上面第 4 行的 smp_mb()，将值“a”从其存储缓冲区移到其缓存行。

5.CPU 0 执行 b=1。它已经拥有这个高速缓存行(换句话说，高速缓存行已经处于“修改”或“独占”状态)，所以它在其高速缓存行中存储新的值“b”。

6.CPU 0 接收“读取”消息，并将包含现在更新的值“b”的高速缓存行传输到 CPU 1，还将该行标记为在其自己的高速缓存中“共享”。

7.CPU 1 接收包含“b”的高速缓存行，并将其安装在其高速缓存中。

8.当(b==0)继续时，CPU 1 现在可以完成执行，并且由于它发现“b”的值是 1，所以它前进到下一个语句，这现在是一个存储器屏障。

9.现在，CPU 1 必须暂停，直到它处理其无效队列中所有预先存在的消息。

10.现在，CPU 1 处理排队的“无效”消息，并使包含“a”的高速缓存行从其自己的高速缓存中无效。

11.CPU 1 执行断言(a==1)，并且由于包含“a”的高速缓存行不再在 CPU 1 的高速缓存中，所以它发送“读取”消息。

12.CPU 0 用包含新值“a”的高速缓存行来响应这个“读取”消息。

13.CPU 1 接收这个缓存行，它包含一个“a”的值 1，所以断言不会触发。

12

随着 MESI 信息的大量传递，中央处理器得出了正确的答案。本节说明了为什么 CPU 设计人员必须极其小心地进行缓存一致性优化。

1 void foo(void) 2 {

3a = 1；

4 SMP _ wmb()；5b = 1；6 } 7

5 读写存储器

障碍

8 空心杆(空心)9 {

10，同时(b == 0)继续；11 SMP _ RMB()；12 断言(a = = 1)；13 }

有些电脑有更多种类的记忆障碍，但是理解这三种变体将会对记忆障碍提供一个很好的介绍。

在前一节中，内存屏障用于标记存储缓冲区和无效队列中的条目。但是在我们的代码片段中，foo()没有理由对无效队列做任何事情，bar()也同样没有理由对存储队列做任何事情。

6 记忆屏障示例

顺序

这一部分展示了一些诱人但微妙的记忆障碍的使用。虽然它们中的许多将在大多数时间工作，并且一些将在某些特定的处理器上一直工作，但是如果目标是产生在所有处理器上可重复工作的代码，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反对订单的架构。

因此，许多中央处理器架构提供了较弱的内存屏障指令，只能执行这两种指令中的一种。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，而完全成熟的内存屏障两者都标记。

这样做的效果是，读内存屏障只对执行它的 CPU 上的负载进行排序，因此读内存屏障之前的所有负载看起来都在读内存屏障之后的任何负载之前完成。类似地，写存储器屏障命令仅存储，再次存储在执行它的中央处理器上，并且再次使得写存储器屏障之前的所有存储将看起来在写存储器屏障之后的任何存储之前已经完成。一个成熟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。

6.1 订购-敌对架构

保罗曾遇到过许多对秩序有敌意的计算机系统，但敌意的本质总是极其微妙，要理解它需要对具体硬件有详细的了解。与其挑一个特定的硬件供应商，不如让我们设计一个虚构的但最大程度上反对内存排序的计算机体系结构，这可能是一个吸引读者阅读详细技术规范的替代方案。

希望详细了解真实硬件结构的读者，请参考中央处理器厂商的产品
[SW95、Adv02、Int02b、IBM94、LSH02、SPA94、Int04b。

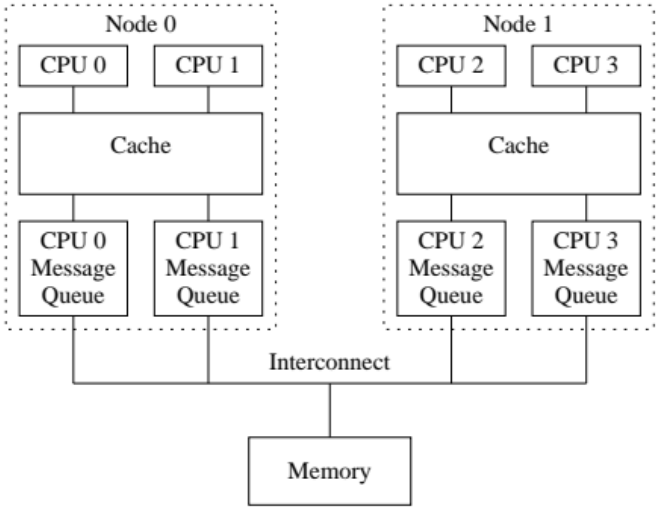
如果我们更新 foo 和 bar 来使用读写内存屏障，它们会显示如下:

该硬件必须遵守以下订购约束条件[McK05a、McK05b]:

1.每个中央处理器总是感觉到自己的内存访问是按程序顺序进行的。

2.只有当给定操作引用不同的位置时，CPU 才会用存储对这两个操作进行重新排序。

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



4.在写入内存屏障(smp_wmb())之前的给定 CPU 的所有存储将被所有 CPU 感知为在该写入内存屏障之后的任何存储之前。

图 8:示例排序-敌对架构

CPU 0 CPU 1 CPU 2

a = 1；

SMP _ wmb()；而(b = = 0)；b = 1；c = 1；z=c。

SMP _ RMB()；x=a。

断言(z = = 0 | | x = = 1)；

5.一个给定的 CPU 在一个完整的内存屏障(smp_mb())之前的所有访问(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:保证每个中央处理器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？为什么或为什么不？

表 2:记忆障碍示例 1

想象一下一个大型的非统一缓存体系结构(NUCA)系统，为了给给定节点中的处理器提供公平的互连带宽分配，在每个节点的互连接口中提供了每个处理器的队列，如图 8 所示。尽管一个给定的中央处理器

的访问是由该中央处理器执行的内存屏障指定的顺序，但是，给定的一对中央处理器的访问的相对顺序可能会被严重地重新排序，正如我们将要看到的。

Int04a，Int04c]，加拉乔罗的论文《[·加 95》，或彼得·苏厄尔的作品《[·苏》。

任何真正的硬件架构师或设计人员无疑都会在瓷制对讲机上大声呼唤拉尔夫，因为他们可能只是有点不高兴，不知道哪个队列应该处理涉及两个处理器都访问的高速缓存行的消息，更不要说这个例子中的许多竞争了。我只能说“给我一个更好的例子”。

6.2 示例 1

表 2 显示了三个代码片段，由中央处理器 0、1 和 2 同时执行。 “a”、 “b”和“c”最初都为零。

假设 CPU 0 最近经历了多次高速缓存未命中，因此其消息队列已满，但 CPU 1 一直在高速缓存内独占运行，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(因此对于 CPU 1 是可见的)，但是将在 CPU 0 的先前流量之后被阻止。相比之下，CPU 1 对“c”的分配将通过 CPU 1 以前的空队列。因此，在看到 CPU 0 分配给“a”之前，CPU 2 很可能会看到 CPU 1 分配给“c”，从而导致断言触发，尽管存在内存障碍。

14

CPU 0 CPU 1 CPU 2

a = 1；而(a = = 0)；

SMP _ MB()；y=b。b = 1；SMP _ RMB()；

x=a。

断言(y = = 0 | | x = = 1)；

表 3:记忆障碍示例 2

CPU 0 CPU 1 CPU 2

1a = 1；

2 SMB _ wmb()；

3b = 1；而(b = = 0)；而(b = = 0)；

4 SMP _ MB()；SMP _ MB()；

5c = 1；d = 1；

6，而(c = = 0)；7，而(d = = 0)；8 SMP _ MB()；

9 e = 1；断言(e = = 0 | | a = = 1)；

表 4:记忆障碍示例 3

理论上，可移植代码不能依赖于这种简单的代码序列，然而，实际上它确实在所有主流计算机系统上工作。

快速测试 10:这个代码可以通过在 CPU 1 的“while”和“c”之间插入一个内存屏障来修复吗？为什么或为什么不？

6.3 示例 2

表 3 显示了三个代码片段，由中央处理器 0、1 和 2 同时执行。“a”和“b”最初都为零。

同样，假设 CPU 0 最近经历了多次高速缓存未命中，因此其消息队列已满，但 CPU 1 一直在高速缓存内独占运行，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(因此对于 CPU 1 是可见的)，但是将在 CPU 0 的先前流量之后被阻止。相比之下，CPU 1 对“b”的分配将通过 CPU 1 以前的空队列。因此，在看到 CPU 0 分配给“a”之前，CPU 2 很可能会看到 CPU 1 分配给“b”，从而导致断言触发，尽管存在内存障碍。

理论上，可移植代码不应该依赖于这个示例代码片段，但是，和以前一样，实际上它确实在大多数主流计算机系统上工作。

3 号线。一旦中央处理器 1 和 2 在第 4 行执行了它们的内存屏障，它们都保证在第 2 行的内存屏障之前看到中央处理器 0 的所有分配。类似地，第 8 行上的 CPU 0 的存储器屏障与第 4 行上的 CPU 1 和 2 的存储器屏障成对，因此 CPU 0 将不会执行第 9 行上对“e”的分配，直到它对“a”的分配对其他两个 CPU 都是可见的。因此，第 9 行的 CPU 2 断言保证不会触发。

快速测验 11:假设表 4 中处理器 1 和 2 的第 3-5 行在中断处理程序中，而处理器 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，为了防止断言被触发，需要做什么改变(如果有的话)？

快速测试 12:如果在表 4 的例子中，中央处理器 2 执行了一个断言(e= =0||c==1)，这个断言会触发吗？

Linux 内核的 synchronize_rcu()原语使用了类似于本例所示的算法。

针对特定处理器的 7 条内存屏障指令

6.4 示例 3

表 4 显示了三个代码片段，由中央处理器 0、1 和 2 同时执行。所有变量最初都是零。

请注意，无论是 CPU 1 还是 CPU 2 都不能继续到第 5 行，直到它们看到 CPU 0 的“b”分配打开

每个中央处理器都有自己独特的内存屏障指令，这使得可移植性成为一个挑战，如表 5 所示。事实上，包括 pthreads 和 Java 在内的许多软件环境只是简单地禁止直接使用内存屏障，将程序员限制在互斥原语中，这些原语将他们限制在所需的范围内。在表中，前四列指示是否

给定的中央处理器允许重新排序加载和存储的四种可能组合。接下来的两列表示给定的 CPU 是否允许用原子指令对加载和存储进行重新排序。

第七列，数据相关的读取重新排序，需要一些解释，这在下面介绍阿尔法处理器的章节中没有解释。简而言之，Alpha 要求读者和链接数据结构的更新者有内存障碍。是的，这确实意味着 Alpha 可以在获取指针之前获取指向的数据，这很奇怪，但却是真的。如果你认为我只是瞎编的，请参阅。这种非常弱的内存模型的好处是 Alpha 可以使用更简单的缓存硬件，这反过来又在 Alpha 的全盛时期允许更高的时钟频率。

最后一列指出给定的中央处理器是否有不一致的指令高速缓存和流水线。这种中央处理器需要为自修改代码执行特殊指令。

带括号的 CPU 名称表示体系结构上允许的模式，但在实践中很少使用。

常见的“拒绝”内存屏障的方法在它适用的地方可能非常合理，但是在某些环境中，比如 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，如下所示:smp mb():“内存屏障”将两者排序

装载和存储。这意味着在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给内存。

载入后重新排序载入？存储后重新排序加载？商店在商店后重新排序？载入后重新排序的商店？原子指令随加载重新排序？原子指令随存储重新排序？从属装货已重新排序？不一致的指令高速缓存/流水线？

阿尔法 Y Y Y Y Y Y Y Y Y Y Y Y Y Y

AMD64 Y

arm V7-A/R Y Y Y Y Y Y Y Y Y

IA64 Y Y Y Y Y Y Y Y Y Y Y Y

(PA-RISC)耶耶耶

PA-RISC 消费物价指数

POWERTM Y Y Y Y Y Y Y Y Y Y Y

(SPARC·RMO)

(SPARC 粒子群算法)

SPARC 左友友

x86 钇

(x86 OOStore)Y Y Y Y Y Y Y Y

zSeries Y Y

smp rmb():“读取内存屏障”，仅命令加载。

smp wmb():只订购商店的“写内存屏障”。

smp 读屏障 depends()强制对依赖于先前操作的后续操作进行排序。除了阿尔法，这个原语在所有平台上都是不可操作的。

16

表 5:内存排序摘要

mmiowb()强制对由全局自旋锁保护的 MMIO 写进行排序。在自旋锁中的内存障碍已经迫使 MMIO 排序的所有平台上，这种初级是不可操作的。具有非不可操作 mmiowb()定义的平台包括一些(但不是全部)IA64、FRV、MIPS 和 SH 系统。这个原语相对来说比较新，所以相对来说很少有司机利用它。

运用大量的知识！对于那些希望更多了解单个中央处理器的记忆一致性模型的人来说，接下来的章节将描述那些最流行和最突出的中央处理器。虽然没有什么可以代替实际阅读给定的 CPU 文档，但这些部分给出了一个很好的概述。

7.1 阿尔法

smp mb()、smp rmb()和 SMP wmb()primitive 也迫使编译器避免任何可能影响跨障碍的内存优化的优化。smp 读屏障依赖于()原语具有类似的效果，但只对阿尔法处理器有影响。

这些原语只在 SMP 通道中生成代码，但是，每个原语都有一个 UP 版本(mb()、rmb()、wmb()和读屏障依赖项()，它们甚至在 UP 内核中也会生成内存屏障。大多数情况下应该使用 smp 版本。然而，后一种原语在编写驱动程序时很有用，因为即使在 UP 内核中，MMIO 访问也必须保持有序。在没有内存屏障指令的情况下，中央处理器和编译器都会很高兴地重新安排这些访问，这最多只会让设备行为异常，并可能导致内核崩溃，或者在某些情况下，甚至损坏硬件。

所以大多数内核程序员不需要担心每个 CPU 的内存障碍特性，只要他们坚持使用这些接口。当然，如果您在给定的特定于 CPU 架构的代码中深入工作，所有的赌注都是输了。

此外，所有的 Linux 锁定原语(自旋锁、读写锁、信号量、RCU，...)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架构的代码或同步原语了。

此外，他们说一点点知识是非常危险的事情。想象一下你可能造成的伤害

对于一个生命周期已经结束的中央处理器，说太多的话似乎有些奇怪，但是阿尔-法很有趣，因为在最弱的内存排序模型下，它对内存操作的重新排序最为激进。因此，它定义了 Linux 内核内存排序原语，这些原语必须适用于所有的处理器，包括 Alpha。因此，理解阿尔法对 Linux 黑客来说是非常重要的。

阿尔法和其他中央处理器之间的区别如图 9 所示的代码所示。该图第 9 行的 smp wmb()保证了第 6-8 行的元素初始化在元素被添加到第 10 行的列表之前被执行，因此无锁搜索将会正确工作。也就是说，它对除阿尔法以外的所有中央处理器作出这种保证。

Alpha 的内存排序非常弱，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垃圾值。

图 10 显示了这种情况如何发生在具有分区高速缓存的渐进并行机器上，从而交替的高速缓存行由高速缓存的不同分区处理。假设列表头标题将由高速缓存组 0 处理，并且新元素将由高速缓存组 1 处理。在 Alpha 上，smp wmb()将保证由图 9 的第 6-8 行执行的高速缓存失效将在第 10 行到达互连之前到达互连，但是绝对不能保证新值到达读处理器核心的顺序。例如，有可能读取的中央处理器的高速缓冲存储器组 1 非常繁忙，但是高速缓冲存储器组 0 空闲。这可能导致新元素的缓存失效被延迟，因此读取的 CPU 获得指针的新值，但看到新元素的旧缓存值。查看网络

17

1 结构 el *insert(长键，长数据)2 {

3 结构 El * p；

4 p = kmalloc(sizeof(*p)，GPF _原子)；5 自旋锁(& mutex)；6p-> next = head . next；7 p->键=键；8 p->数据=数据；9 SMP _ wmb()；10 头，下一个= p；11 旋转解锁(& mutex)；12 } 13

14 struct el *search(长键)15 {

16 结构 El * p；

17p = head . next；

18 同时(p！ = &head) {

19 /*阿尔法上的臭虫！！！ */

20 if (p->key == key) {

21 返回(p)；

22 }

23 p = p->下一个；

24 }；

25 返回(空)；26 }

甲基溴测序

缓存库 0

缓存库 1

甲基溴测序

写中央处理器核心

甲基溴测序

缓存库 0

缓存库 1

甲基溴测序

读取中央处理器核心

*

互相联系

图 10:为什么需要 smp 读屏障()

图 9:插入和无锁搜索

网站呼吁更早的信息，或者，再次，如果你认为我只是在编造这一切。

可以在指针获取和取消引用之间放置一个 smp rmb()原语。然而，这给系统(如 i386、IA64、PPC 和 SPARC)带来了不必要的开销，这些系统在读取端考虑了数据依赖性。在 Linux 2.6 内核中添加了一个 smp 读屏障 depends()原语，以消除这些系统的开销。这个原语可以如图 11 的第 19 行所示使用。

也有可能实现一个可以用来代替 smp wmb()的软件工具条，它将强制所有正在读取的 CPU 按顺序查看正在写入的 CPU 的写入。然而，这种方法被 Linux 社区认为是对极弱有序的 CPU(如 Alpha)施加过度的开销。这个软件障碍可以通过发送处理器间中断(IPIs)到所有其它的处理器来实现。一旦收到这样一个 IPI，

当然，精明的读者已经认识到阿尔法远没有它可能的卑鄙和肮脏，第 6.1 节中的(谢天谢地)神话建筑就是一个很好的例子。

一个中央处理器将执行一个内存屏障指令，实现一个内存屏障击落。需要额外的逻辑来避免死锁。当然，尊重数据依赖关系的处理器会将这种障碍简单地定义为 smp wmb()。也许这个决定应该在未来随着阿尔法逐渐消失在夕阳中而重新考虑。

Linux 内存屏障原语的名字取自 Alpha 指令，所以 smp mb()是 mb，smp rmb()是 rmb，smp wmb()是 wmb。阿尔法是唯一一个读写障碍依赖的处理器

快速测试 13:为什么阿尔法的 SMP _ read _ barrier _ dependences()是一个 smp_mb()而不是 smp_ rmb()?

关于阿尔法的更多细节，请参阅参考人[SW95]。

7.2 AMD64

AMD64 与 x86 兼容，并且最近更新了其内存模型[Adv07]，以实施实际实施一段时间以来提供的更严格的排序。Linux smp mb()原语的 AMD64 实现是 f sence，smp rmb()是 f sence，smp wmb()是 f sence。理论上，这些可能是放松的，但是任何这样的放松-

18

1 结构 el *insert(长键，长数据)2 {

3 结构 El * p；

4 p = kmalloc(sizeof(*p)，GPF _原子)；5 自旋锁(& mutex)；6p-> next = head . next；7 p->键=键；8 p->数据=数据；9 SMP _ wmb()；10 头，下一个= p；11 旋转解锁(& mutex)；12 } 13

14 struct el *search(长键)15 {

16 结构 El * p；

17p = head . next；

18 同时(p！= &head) {

19 SMP _ read _ barrier _ depends()；20 if(p-> key = = key){ 21 return(p)；22 }

23 p = p->下一个；

24 }；

25 返回(空)；26 }

图 11:安全插入和无锁搜索

2.DSB(数据同步屏障)使指定类型的操作在任何后续操作(任何类型)执行之前实际完成。业务的“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(你可以选择写缓冲区或数据写条)。

3.ISB(指令同步屏障)刷新中央处理器流水线，因此 ISB 之后的所有指令只有在 ISB 完成之后才被提取。例如，如果您正在编写一个自修改程序(如 JIT)，您应该在生成代码和执行代码之间执行一个 ISB。

这些指令没有一条与 Linux 的 rmb()原语完全匹配，因此必须作为完整的 DMB 实现。DMB 和 DSB 指令对交流有一个递归定义

操作必须考虑 SSE 和 3DNOW 指令。

7.3 ARMv7-A/R

ARM 系列处理器在嵌入式应用中非常受欢迎，尤其是手机等功率受限的应用。尽管如此，ARM 的多处理器实现已经有五年多了。它的记忆模型与 Power 的记忆模型相似(见第 7.6 节，但 ARM 使用了一套不同的记忆屏障工具[ARM10):

1.DMB(数据存储屏障)使指定类型的操作看起来在同一类型的任何后续操作之前已经完成。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 eieio 指令)。此外，ARM 允许缓存一致性具有三个范围之一:单处理器、处理器子集(“内部”)和全局(“外部”)。

在障碍之前和之后排序，其效果类似于 POWER 的累积性。

ARM 还实现了控制依赖，因此如果条件分支依赖于加载，那么在该条件分支之后执行的任何存储都将在加载之后排序。然而，除非在分支和加载之间存在 ISB 指令，否则不能保证对条件分支之后的加载进行排序。考虑以下示例:

1 R1 = x；

2 if(R1 = = 0)3 nop()；4y = 1；5 R2 = z；6 ISB()；7 R3 = z；

在本例中，加载-存储控制相关性或-ing 导致从第 1 行的 x 到第 4 行的 y 的加载在存储之前被排序。然而，ARM 不考虑负载-负载控制的依赖性，所以第 1 行的负载很可能发生在第 5 行的负载之后。另一方面，第 2 行的条件分支和第 6 行的 ISB 指令的组合确保了第 7 行的加载发生在第 1 行的加载之后。

19

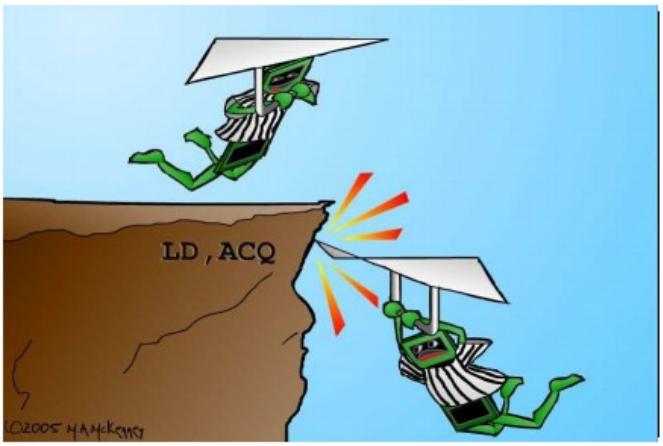


图 12:半个内存屏障

这提供了传递性的概念，如果一个给定的代码片段认为一个给定的访问已经发生，任何后面的代码片段也将认为前面的访问已经发生。假设，也就是说，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 架构允许完全重新订购货物和商店，但实际的中央处理器运行完全订购的[Kan96]。这意味着 Linux 内核的内存排序原语不生成任何代码，但是，它们使用 gcc 内存属性来进行编译器优化，这将跨内存屏障对代码进行重新排序。

7.4 IA64

7.6 电源/动力电脑

IA64 提供了一个弱一致性模型，因此就显式内存屏障指令而言，IA64 有权对内存引用进行任意重新排序([国际 02b】。IA64 有一个名为 mf 的内存栅栏指令，但也有“半内存栅栏”修改程序来加载、存储和修改它的一些原子指令[指令 02a]。acq 修饰符防止随后的存储器参考指令在 acq 之前被重新排序，但是允许先前的存储器参考指令在 acq 之后被重新排序，如图 12 所示。类似地，rel 修饰符防止先前的内存引用指令在 rel 之后被重新排序，但是允许后续的内存引用指令在 rel 之前被重新排序。

这些半内存栅栏对关键部分很有用，因为将操作推入关键部分是安全的，但让它们溢出可能是致命的。然而，作为唯一具有此属性的处理器之一，IA64 定义了与锁获取和释放相关联的 Linux 内存排序语义。

IA64 mf 指令用于 Linux 内核中的 smp rmb()、smp mb()和 smp wmb()原语。哦，尽管有与此相反的谣言，“MF”m neumonic 确实代表“记忆围栏”。

动力和动力电脑中央处理器系列有各种各样的内存屏障指令[IBM94，LSH02]:

1.同步使所有前面的操作看起来在任何后续操作开始之前已经完成。因此，这个指令相当昂贵。

2.lwsync(轻量同步)针对后续装载和存储订购装载，也订购存储。但是，它不会根据后续的装载来订购商店。有趣的是，lwsync 指令执行的顺序与 zSeries 相同，更确切地说，是 SPARC TSO。

3.eieio(强制按顺序执行输入/输出，如果您想知道的话)会导致所有前面的可缓存存储似乎都在所有后面的存储之前完成。然而，存储到可高速缓存的存储器与存储到不可高速缓存的存储器是分开排序的，这意味着 eieio 不会强制 MMIO 的存储预先释放自旋锁。

最后，IA64 为“再租赁”操作提供了全球总订单，包括“mf”指令。

4.isync 强制所有前面的指令在任何后续指令之前完成

指令开始执行。这意味着前面的指令必须已经进行到足够的程度，以至于它们可能产生的任何陷阱已经发生或者保证不会发生，并且这些指令的任何副作用(例如，页表变化)都可以被后面的指令看到。

不幸的是，这些指令没有一条与 Linux 的 wmb()原语完全一致，后者要求所有的存储都要排序，但不要求同步指令的其他高开销操作。但是没有选择:wmb()和 mb()的 ppc64 版本被定义为重量级同步指令。然而，MMIO 从来不使用 Linux 的 smp wmb()指令(毕竟，因为驱动程序必须小心地在 UP 和 smp 内核中订购 MMIOs)，所以它被定义为轻量级 eieio 指令。这个指令可能是独一无二的，因为它有五个元音。smp mb()指令也被定义为同步指令，但是 smp rmb()和 rmb()都被定义为轻量 lwsync 指令。

权力具有“累积性”，可以用来获得传递性。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早期代码片段本身看到的访问。更多细节可从麦肯尼和西尔维拉[MS09 获得。

除了用 Power isync 指令代替 ARM ISB 指令之外，Power 与 ARM 的控制依赖关系非常相似。

7.7 SPARC·RMO、粒子群算法和粒子群算法

SPARC 的 Solaris 使用 TSO(总存储顺序)，就像为“SPARC”32 位体系结构构建的 Linux 一样。然而，一个 64 位的 Linux 内核(“sparc64”架构)在 RMO 的 sparc(宽松内存顺序)模式下运行[SPA94]。SPARC 架构还提供了一个中间粒子群算法(部分存储或排序)。任何在 RMO 运行的程序都将运行在粒子群优化算

法或粒子群优化算法中，同样，运行在粒子群优化算法中的程序也将运行在粒子群优化算法中。将共享内存并行程序移到另一个方向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 有一个非常灵活的记忆屏障指令[SPA94]它允许对以下内容进行细粒度的控制:

商店商店:在子商店之前订购商店

昆特商店。(此选项由 Linux smp wmb()原语使用。)

LoadStore:在后续存储之前对前一次加载进行排序。

StoreLoad:在后续装载之前先订购商店。

LoadLoad:在子步骤之前对加载进行排序

大量货物。(该选项由 Linux smp rmb()原语使用。)

同步:在开始任何后续操作之前，完全完成所有前面的操作。

POWER 体系结构的许多成员都有不一致的指令高速缓存，因此存储到存储器中不一定会反映在指令高速缓存中。谢天谢地，现在很少有人编写自修改代码，但是 JITs 和编译器一直在这样做。此外，从中央处理器的角度来看，重新编译最近运行的程序看起来就像是自修改代码。icbi 指令(指令高速缓存块无效)使指令高速缓存中的指定高速缓存行无效，并可用于这些情况。

内存问题:完成之前的内存操作

在后续的内存操作之前，对内存映射的输入/输出的某些实例来说很重要

之前的存储和随后的加载，甚至仅针对访问相同存储位置的存储和加载。

Linux smp mb()原语将前四个选项一起使用，如在 membar #LoadLoad 中

| # LoadStore | # StoreStore | # StoreLoad，从而完全排序内存操作。

那么，为什么需要 membar # MemIssue？因为 membar #StoreLoad 可能允许后续加载从写缓冲区获取其值，如果写入到 MMIO 寄存器会对要读取的值产生副作用，这将是灾难性的。相反，membar #MemIssue 会等到写缓冲区被刷新后，才允许加载执行，从而确保加载实际上从 MMIO 寄存器获得其值。驱动程序可以改为使用 membar #同步，但是在不需要更昂贵的 membar #同步的附加功能的情况下，更轻的 membar # MemIssue 是首选。

membar #后备是 membar #MemIssue 的轻量版本，当写入给定的 MMIO 寄存器影响下一次从该寄存器中读取的值时，这是有用的。但是，当对给定 MMIO 寄存器的写操作影响到下一步将从其他 MMIO 寄存器读取的值时，必须使用较重的 membar #MemIssue。

尚不清楚为什么 SPARC 没有将 wmb()定义为 membar # MemIssue，将 smb wmb()定义为 membar #StoreStore，因为当前的定义对于某些驱动程序中的 bug 来说似乎更容易出错。很有可能所有运行 Linux 的 SPARC CPU 都实现了一个比架构允许的更保守的内存排序模型。

SPARC 要求在存储指令和执行[SPA94]之间使用刷新指令。这是从 SPARC 指令高速缓存中清除该位置的任何先前值所必需的。请注意，刷新获取一个地址，并将只从指令缓存中刷新该地址。在 SMP 系统上，所有 CPU 的高速缓存都被刷新，但是没有方便的方法来确定非 CPU 的刷新何时完成，尽管有对实现注释的引用。

7.8 x86

由于 x86 处理器提供了“进程排序”，因此所有处理器都同意给定处理器写入内存的顺序，所以 smp wmb()原语是不可操作的

用于中央处理器[04b]。但是，需要编译器指令来防止编译器执行可能导致 smp wmb()基元之间重新排序的优化。

另一方面，x86 处理器传统上没有为加载提供排序保证，因此 smp mb()和 smp rmb()原语扩展为锁定；addl。这个原子指令对装载和存储都是一个障碍。

最近，英特尔发布了 x86 英特尔 07 内存模型]。事实证明，英特尔的实际处理器执行的排序比先前的规范中宣称的更为严格，因此这一模型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 国际 09 的更新内存模型，第 8.2 节]，该模型规定了商店的全球总订单，尽管单个处理器仍被允许将自己的商店视为早于全球总订单的订单。总排序的这一例外是允许涉及存储缓冲区的重要硬件优化所必需的。此外，内存排序遵循因果关系，因此如果 CPU 0 看到 CPU 1 的存储，那么 CPU 0 保证会看到 CPU 1 在存储之前看到的所有存储。软件可能使用原子操作来覆盖这些硬件优化，这是原子操作比非原子操作更昂贵的一个原因。这一总的商店订单不能保证在旧的处理器上。

但是，请注意，一些 SSE 指令是弱有序的(clflush 和非时间移动指令[Int04a])。具有 SSE 的 CPU 可以使用 smp mb 的 f sence()，smp rmb 的 f sence()和 smp wmb 的 f sence()。

一些版本的 x86 处理器有一个模式位来支持无序存储，对于这些处理器，smp wmb()也必须定义为锁定；addl。

尽管许多较旧的 x86 实现都支持自修改代码，而不需要任何特殊指令，但是 x86 体系结构的较新版本不再需要 x86 处理器如此商品化。有趣的是，这种放松来得正是时候，给 JIT 实施者带来了不便。

22

7.9 系列

zSeries 机器组成了 IBM 主机系列，以前称为 360、370 和 390[国际 04c]。并行性出现在 zSeries 的后期，但是考虑到这些大型机在 20 世纪 60 年代中期首次发货，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。它还具有相对强大的内存排序语义，如表 5 所示，这应该允许 smp wmb()原语成为 nop(在您阅读本文时，这种变化可能已经发生了)。该表实际上描述了这种情况，因为 zSeries 内存模型在其他方面是连续一致的，这意味着所有的处理器将同意不同处理器的不相关存储的顺序。

与大多数处理器一样，zSeries 架构不能保证缓存一致的指令流，因此，自修改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 机器实际上在不序列化指令的情况下容纳自修改代码。zSeries 指令集提供了大量的序列化指令，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)和测试和设置等。

下一条指令。因为可能会有数以千计的其他线程，所以 CPU 将被完全利用，所以不会浪费 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其有限，以及越来越严格的实时要求，对于一些应用程序来说，这些要求在几十微秒内。实时响应要求很难满足，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，要满足这些要求会更加困难。

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，这种技术很可能允许中央处理器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。一个相反的论点是，电池供电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？我们不知道，所以我们准备接受这两种情况。

9 对硬件设计人员的建议

8 记忆障碍是永远的吗?

最近已经有许多系统对无序执行，特别是对内存引用的重新排序不太积极。这种趋势会延续到记忆障碍成为过去的那一步吗?

赞成的论点将引用被提议的大规模多线程硬件架构，这样每个线程将等待直到内存准备好，同时数十个、数百个、甚至数千个其他线程正在取得进展。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突出的操作完成，然后再继续

硬件设计师可以做很多事情来让软件人的生活变得困难。这里列出了我们过去遇到的一些类似的事情，希望能有助于防止将来出现类似的问题:

1.忽略缓存一致性的输入/输出设备。这一迷人的错误特性可能导致内存中的内存分配丢失最近对输出缓冲区的更改，或者同样糟糕的是，导致输入缓冲区在内存分配完成后被中央处理器缓存的内容覆盖。为了使您的系统在面对这种不良行为时能够正常工作，您必须在将缓冲区提供给输入/输出设备之前，仔细地刷新任何内存缓冲区中任何位置的中央处理器缓存。即便如此，你也需要非常小心地避免指针错误，

因为即使对输入缓冲区的错误读取也会导致数据输入损坏!

同样，我们鼓励硬件设计师避免这些做法!

2.忽略缓存一致性的设备中断。这听起来可能很无辜——毕竟，中断不是内存引用，是吗？但是想象一下，一个 CPU 有一个分离的高速缓存，其中一个存储体非常繁忙，因此占用了输入缓冲区的最后一个高速缓存行。如果相应的输入/输出完成中断到达该中央处理器，则该中央处理器对缓冲区最后一个高速缓存行的内存引用可能返回旧数据，再次导致数据损坏，但其形式在以后的崩溃转储中是不可见的。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。

承认

我要感谢许多中央处理器架构师，他们耐心地解释了中央处理器的指令和内存重排序特性，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、于尔根·普罗斯特、英戈·阿德隆和拉维·阿里米利。韦恩值得特别感谢，因为他耐心地解释了阿尔法对相关负载的重新排序，这是我极力抵制的一课！我们都欠戴夫·凯克和阿尔特姆·比尤茨基一份情，感谢他们帮助我们将这些材料变得易于阅读。

3.忽略的处理器间中断

缓存一致性。法律声明

如果 IPI 在相应的消息缓冲区中的所有高速缓存行被提交到存储器之前到达其目的地，这可能是有问题的。

4.超越缓存一致性的上下文切换。

如果内存访问过于无序地完成，那么上下文切换会非常困难。如果任务在源 CPU 可见的所有内存访问到达目标 CPU 之前从一个 CPU 转移到另一个 CPU，那么任务可以很容易地看到相应的变量恢复到先前的值，这会致命地混淆大多数算法。

快速测验的 10 个答案

5.过度种类的模拟器和模拟器。很难编写迫使内存重新排序的模拟器或仿真器，所以在这些环境中运行良好的软件在真正的硬件上运行时会有一个令人讨厌的惊喜。不幸的是，硬件仍然比模拟器和仿真器更狡猾，但我们希望这种情况会改变。

快速测验 1:

如果两个处理器试图同时使同一个高速缓存行无效，会发生什么情况？回答:

其中一个 CPU 首先获得对共享总线的访问权，并且该 CPU“获胜”。另一个中央处理器必须使其高速缓存行的副本无效，并向另一个中央处理器发送“无效确认”消息。

当然，失败的 CPU 可能会立即发出一个“读无效”事务，所以获胜的 CPU 的胜利将是短暂的。

快速测验 2:

当大型多处理器中出现“无效”消息时，每个 CPU 都必须给出“无效确认”响应。由此产生的“无效确认”响应的“风暴”难道不会完全淹没系统总线吗？

额外的状态实际上不需要与高速缓存行一起存储，因为一次只有几行将被转换。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。轩尼诗和帕特森的经典电脑架构介绍[HP95]涵盖了许多这些问题。

回答:

如果大规模多处理器实际上是以这种方式实现的，那就有可能。更大的多处理器，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个和其他问题。

快速测验 5:

什么样的操作顺序会将中央处理器的缓存全部恢复到“无效”状态？回答:

没有这样的顺序，至少在中央处理器的指令集中没有特殊的“刷新我的缓存”指令的情况下。大多数中央处理器都有这样的指令。

快速测验 3:

如果 SMP 机器真的在使用消息传递，那为什么还要麻烦 SMP 呢？回答:

在过去的几十年里，关于这个话题有相当多的争议。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，获得软件消息传递无法获得的带宽和延迟。另一个答案是，真正的真相是在经济学中找到的，因为大型 SMP 机器和小型 SMP 机器集群的相对价格。第三个答案是 SMP 编程模型比分布式系统更容易使用，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。

快速测验 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”？

回答:

因为所讨论的缓存行不仅仅包含变量 a。

快速测验 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？难道 CPU 0 不需要与“a”共享该缓存行的其他变量的值吗？

快速测验 4:

硬件如何处理上述延迟传输？回答:

通常通过添加额外的状态，尽管这些

回答:

假设 CPU 0 具有包含“a”的高速缓存行的只读副本，则它已经具有这些变量的值。因此，所有 CPU 0 需要做的就是使其他 CPU 丢弃它们的这个高速缓存行的副本。因此，“无效”消息就足够了。

直观的因果排序:如果 B 看到了 A 访问的效果，C 看到了 B 访问的效果，那么 C 也必须看到 A 访问的效果。简而言之，硬件设计师至少对软件开发人员有点同情。

快速测验 8:

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

回答:

CPU 可以自由地推测执行，这可以在 while 循环完成之前执行断言。

快速测验 11:

假设表 4 中处理器 1 和 2 的第 3-5 行在中断处理程序中，处理器 2 的第 9 行在进程级运行。要使代码正常工作，换句话说，要防止断言被触发，需要做什么更改(如果有的话)?

快速测验 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？为什么或为什么不?

回答:

需要编写断言以确保“e”的负载先于“a”的负载。在 Linux 内核中，barrier()原语可以用来实现这一点，其方式与前面示例中断言中使用的内存屏障非常相似。

回答:

不。考虑这样的情况:一个线程从一个中央处理器迁移到另一个中央处理器，并且目标中央处理器察觉到源中央处理器最近的内存操作不正常。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，安全执行上下文切换所需的锁定应该自动提供所需的内存屏障，以使用户级任务按顺序看到自己的访问。也就是说，如果你正在设计一个超级优化的调度程序，无论是在内核还是在用户层面，请记住这个场景!

快速测验 12:

如果在表 4 的例子中，CPU 2 执行了一个断言(e==0||c==1)，这个断言会触发吗?

回答:

结果取决于中央处理器是否支持“传递性”换句话说，在看到 CPU 1 的存储为“c”后，CPU 0 存储为“e”，在 CPU 0 从“c”到存储为“e”的负载之间有一个内存屏障。如果其他某个 CPU 看到 CPU 0 的存储为“e”，是否也保证看到 CPU 1 的存储?

我所知的所有消费物价指数都声称提供过渡性的

快速测验 10:

这个代码可以通过在 CPU 1 的“while”和分配给“c”之间插入一个内存屏障来修复吗？为什么或为什么不？

城市。

快速测验 13:

为什么阿尔法的 SMP _ read _ barrier _ dependencies()是一个 smp_mb()而不是 smp_rmb()?

回答:

不。这样的内存屏障只会强制在中央处理器 1 本地订购。它对 CPU 0 和 CPU 1 的访问的相对顺序没有影响，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“传递性”，这提供了

回答:

首先，Alpha 只有 mb 和 wmb 指令，所以在这两种情况下，smp_rmb()都将由 Alpha mb 指令实现。

更重要的是，SMP _ read _ barrier _ dependencies()

必须订购后续商店。例如，考虑以下代码:

[英特尔公司。英特尔安腾架构软件开发人员手册第 3 卷:指令集参考，2002。

1 p =全局指针；2 SMP _ read _ barrier _ depends()；3 如果(do _ some _ with(p-> a，p-> b)= 0)4p-> hey _ look = 1；

[英特尔公司。英特尔安腾架构软件开发人员手册第 3 卷:系统架构，2002。

在这里，必须订购到 p->hey_look 的商店，而不仅仅是 p->a 和 p->b 的货物。

[英特尔公司。IA-32 英特尔架构软件开发人员手册卷 2B:指令集参考，N-Z，2004。可用:ftp://download.intel.com/设计/五月 4/手册/25366714.pdf

[观察:2005 年 2 月 16 日]。

参考

[Adv02]高级微型设备。AMD x86-64 架构程序员手册第 1-5 卷，2002。

[英特尔公司。IA-32 英特尔架构软件开发人员手册第 3 卷:系统编程指南，2004 年。可用:ftp://download.intel.com/设计/五月 4/手册/25366814.pdf

[观察:2005 年 2 月 16 日]。

[Adv07]高级微型设备。AMD x86-64 架构程序员手册第 2 卷:系统编程，2007。

[军火有限公司。ARM 架构参考手册:ARMv7-A 和 ARMv7-R 版，2010 年。

[国际商业机器公司。操作的架构原则。可得:[浏览时间:2005 年 2 月 16 日，2004 年 5 月。

大卫·库勒、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。莫尔-甘·考夫曼，1999 年。

[国际 07]英特尔公司。

英特尔 64 Ar-

架构内存订购白皮书，2007 年。可得:http://developer . Intel . com/products/processor/manuals/318147 . pdf[查看:2007 年 9 月 7 日]。

[·伽 95]库罗什·伽拉乔罗。共享内存多处理器的内存一致性模型。技术报告 CSL-TR-95-685，计算机系统实验室，电气工程和计算机科学系，斯坦福大学，斯坦福，加利福尼亚州，1995 年 12 月。可用:http://www.hpl.hp.com/techreports/康柏-DEC/WRL-95-9。pdf[查看时间:2004 年 10 月 11 日]。

[国际 09]英特尔公司。

英特尔 64 和英特尔架构-

32 架构软件开发人员手册，第 3A 卷:系统程序设计指南，第 1 部分，2009。可用:http://download.intel.com/design/处理器/手册/253668 . pdf[查看时间:2009 年 11 月 8 日]。

[·亨尼西和大卫·帕特森。

计算机体系结构:定量方法。摩根·考夫曼，1995 年。

[·凯恩格里·凯恩。PA-RISC 2.0 架构。惠普专业书籍，1996 年。

[IBM 微电子公司和摩托罗拉公司。功率-erPC 微处理器系列:编程环境，1994。

迈克尔·莱昂斯、埃德·西拉和比尔·海。PowerPC 存储模型和 AIX 编程。可用:http:

//www-106 . IBM . com/developer works/eserver/articles/powerpc . html

[观察:2005 年 1 月 31 日]，2002 年 8 月。

保罗·麦肯尼。《现代微处理器中的存储器》，第一部分，《Linux 杂志》，1(136):52- 57，2005 年 8 月。可得:http://www.rdrop.com/users/paulmck/可扩展性/论文/订购。2007.09。19a.pdf[查看 2007 年 11 月 30 日]。

保罗·麦肯尼。现代微处理器中的存储器，第二部分。《Linux 杂志》，1(137):78-82，2005 年 9 月。可得:http://www.rdrop.com/users/paulmck/可扩展性/论文/订购。2007.09。19a.pdf[查看 2007 年 11 月 30 日]。

保罗·麦肯尼和劳尔·西尔维拉。c/c++存储模型的示例电源实现。可查阅:2009 年 4 月 5 日，2009 年 2 月。

[·瑟尔]彼得·瑟尔。多道程序的语义-

分类程序。可查阅:2010 年 6 月 7 日]。

[SPA94] SPARC 国际。SPARC 建筑手册，1994。

[SW95]理查德·l·斯泰斯和理查德·t·威特克。阿尔法·AXP 建筑。数字出版社，第二版，1995 年。