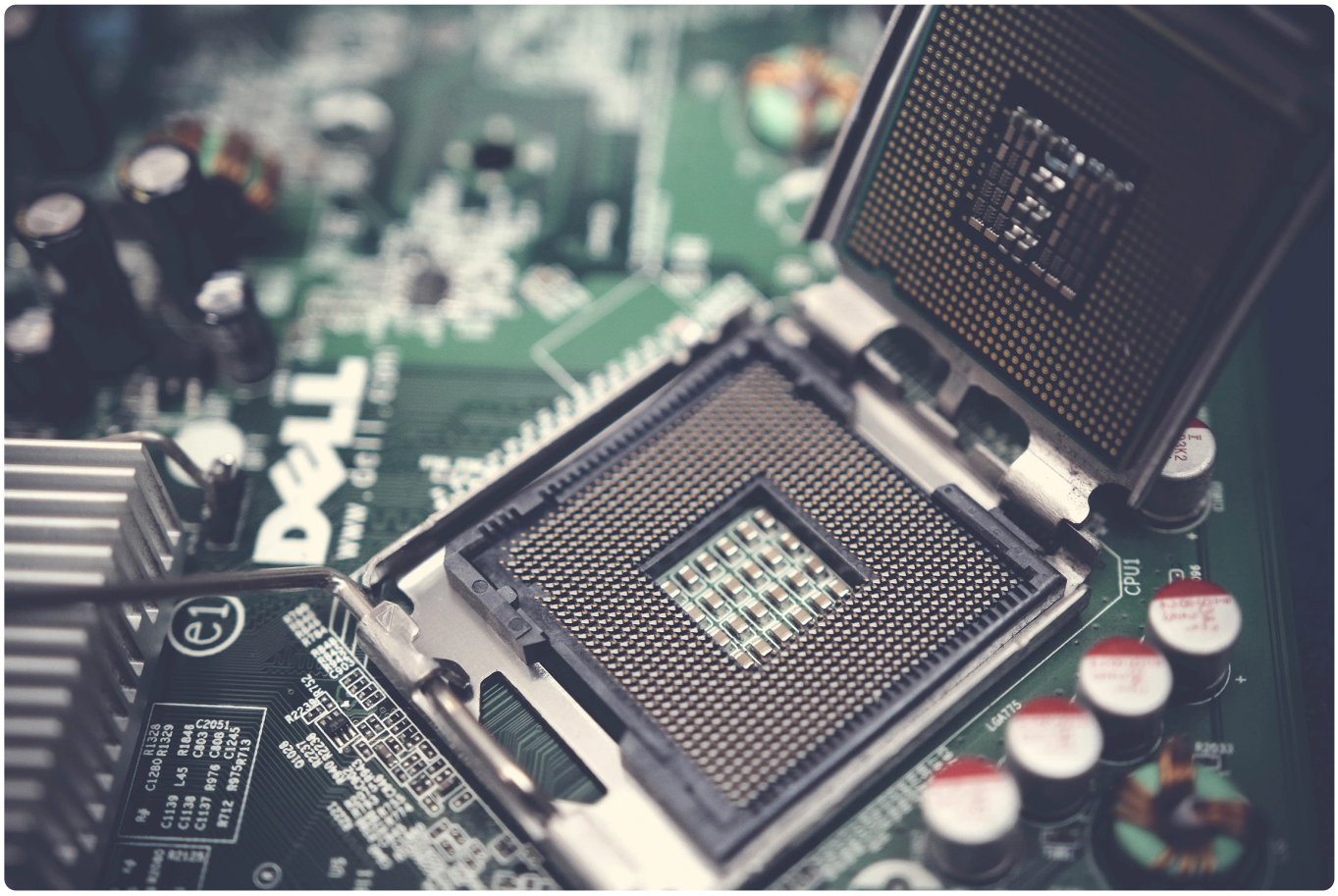
下载APP

16 | 内存模型：有了MESI为什么还需要内存屏障？

2021-12-03 海纳

《编程高手必学的内存知识》 课程介绍 >



讲述：海纳

时长 19:25 大小 17.80M

你好，我是海纳。

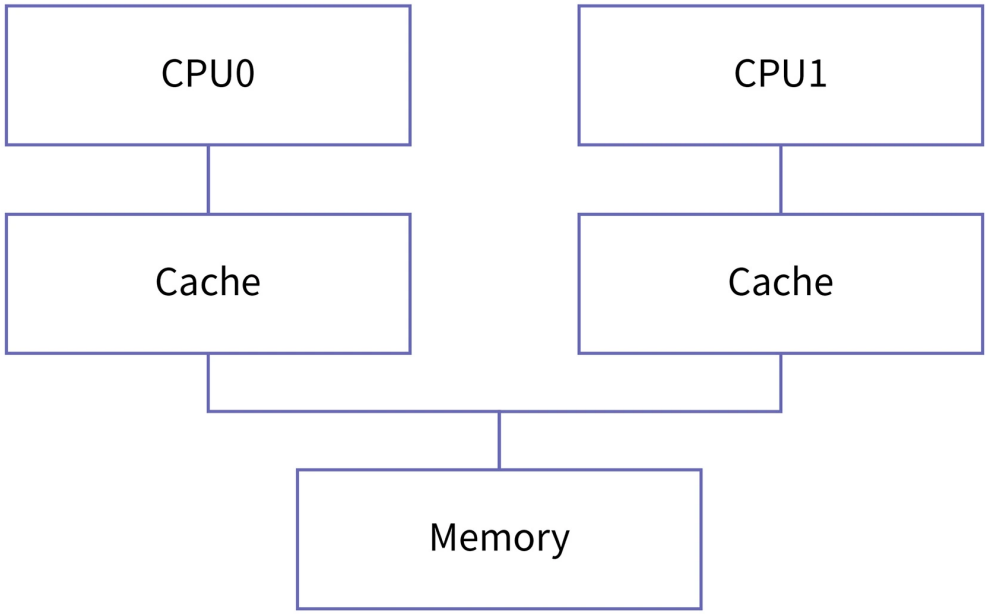
上一节课，我们学习了 MESI 协议，我们了解到，MESI 协议能够解决多核 CPU 体系中，多个 CPU 之间缓存数据不一致的问题。但是，如果 CPU 严格按照 MESI 协议进行核间通讯和同步，核间同步就会给 CPU 带来性能问题。既要遵守协议，又要提升性能，这就对 CPU 的设计人员提出了巨大的挑战。

那严格遵守 MESI 协议的 CPU 会有什么样的性能问题呢？我们又可以怎么来解决这些问题呢？今天我们就来仔细分析一下。搞清楚了这些问题，你会对 C++ 内存模型和 Java 内存

模型有更加深入的理解，在分析并发问题时能够做到有的放矢。

严守 MESI 协议的 CPU 会有啥问题？

我们上节课说过，MESI 代表的是 Modified、Exclusive、Shared、Invalid 这四种缓存状态，遵守 MESI 协议的 CPU 缓存会在这四种状态之间相互切换。这种 CPU 缓存之间的关系是这样的：



极客时间

从上面这张图你可以看到，Cache 和主内存 (Memory) 是直接相连的。一个 CPU 的所有写操作都会按照真实的执行顺序同步到主存和其他 CPU 的 cache 中。

严格遵守 MESI 协议的 CPU 设计，在它的某一个核在写一块缓存时，它需要通知所有的同伴：我要写这块缓存了，如果你们谁有这块缓存的副本，请把它置成 Invalid 状态。Invalid 状态意味着该缓存失效，如果其他 CPU 再访问这一缓存区时，就会从主存中加载正确的值。

发起写请求的 CPU 中的缓存状态可能是 Exclusive、Modified 和 Share，每个状态下的处理是不一样的。

如果缓存状态是 Exclusive 和 Modified，那么 CPU 一个核修改缓存时不需要通知其他核，这是比较容易的。

但是在 Share 状态下，如果一个核想独占缓存进行修改，就需要先给所有 Share 状态的同伴发出 Invalid 消息，等所有同伴确认并回复它“Invalid acknowledgement”以后，它才能把这块缓存的状态更改为 Modified，这是保持多核信息同步的必然要求。

这个过程相对于直接在核内缓存里修改缓存内容，非常漫长。这也就会导致，某个核请求独占时间比较长。

那怎么来解决这个问题呢？

写缓冲与写屏障

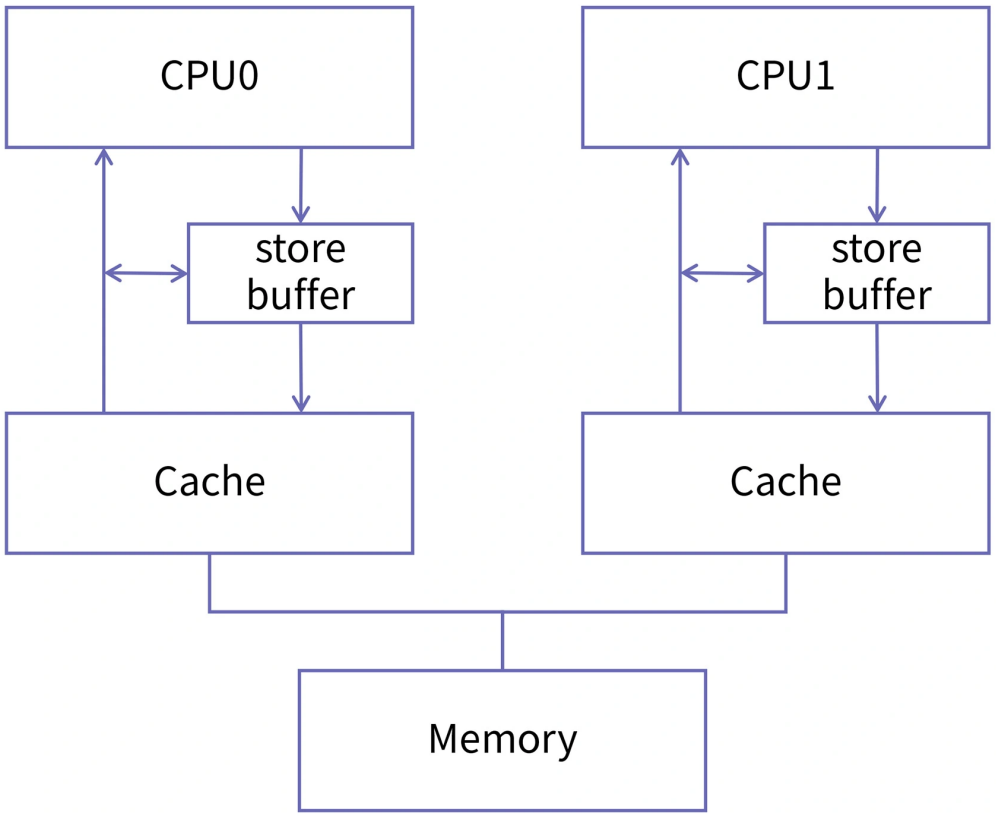
CPU 的设计者为每个核都添加了一个名为 **store buffer** 的结构，store buffer 是硬件实现的缓冲区，它的读写速度比缓存的速度更快，所有面向缓存的写操作都会先经过 store buffer。

不过，由于中文材料中经常将 cache 和 buffer 都翻译成缓冲，或者缓存，很容易混淆概念，所以在这里，我想强调一下 cache 和 buffer 的区别。

cache 这个词，往往意味着它所存储的信息是副本。cache 中的数据即使丢失了，也可以从内存中找到原始数据（不考虑脏数据的情况），**cache 存在的意义是加速查找**。

但是 buffer 更像是蓄水池，你可以理解成它是一个收作业的课代表，课代表会把所有同学的作业都收集齐以后再一次性地交到老师那里。buffer 中的数据没有副本，一旦丢失就彻底丢失了。store buffer 也是同样的道理，它会收集多次写操作，然后在合适的时机进行提交。

增加了 store buffer 以后的 CPU 缓存结构是这样的：



在这样的结构里，如果 CPU 的某个核再要对一个变量进行赋值，它就不必等到所有的同伴都确认完，而是直接把新的值放入 store buffer，然后再由 store buffer 慢慢地去做核间同步，并且将新的值刷入到 cache 中去就好了。而且，每个核的 store buffer 都是私有的，其他核不可见。

为了让你更好理解核间同步的问题，我们现在来举个例子。我们使用两个 CPU，分别叫做 CPU0 和 CPU1，其中 CPU0 负责写数据，而 CPU1 负责读数据，我们看看在增加了 store buffer 这个结构以后，它们在进行核间同步时会遇到什么问题。

假如 CPU0 刚刚更新了变量 a 的值，并且将它放到了 store buffer 中，CPU0 自己接着又要读取 a 的值，此时，它会在自己的 store buffer 中读到正确的值。

那如果在这一次修改的 a 值被写入 cache 之前，CPU0 又一次对 a 值进行了修改呢？那也没问题，这次更新就可以直接写入 store buffer。因为 store buffer 是 CPU0 私有的，修改它不涉及任何核间同步和缓存一致性问题，所以效率也得到了比较大的提升。

但用 store buffer 也会有一个问题，那就是**它并不能保证变量写入缓存和主存的顺序**，你先来看看下面这个代码：

复制代码

```
// CPU0
void foo() {
    a = 1;
    b = 1;
}

// CPU1
void bar() {
    while (b == 0) continue;
    assert(a == 1);
}
```

你可以看到，在这个代码块中，CPU0 执行 foo 函数，CPU1 执行 bar 函数。但在对变量 a 和 b 进行赋值的时候，有两种情况会导致它们的赋值顺序被打乱。

第一种情况是 CPU 的乱序执行。在 Cache 的基本原理一课中，我们已经讲过 CPU 为了提升运行效率和提高缓存命中率，采用了乱序执行。

第二种情况是 store buffer 在写入时，有可能 b 所对应的缓存行会先于 a 所对应的缓存行进入独占状态，也就是说 b 会先写入缓存。

这种情况完全是有可能的。你想，如果 a 是 Share 状态，b 是 Exclusive 状态，那么尽管 CPU0 在执行时没有乱序，这两个变量由 store buffer 写入缓存时也是不能保证顺序的。

那这个时候，我们假设 CPU1 开始执行时，a 和 b 所对应的缓存行都是 Invalid 状态。当 CPU1 开始执行第 9 行的时候，由于 b 所对应的缓存区域是 Invalid 状态，它就会向总线发出 BusRd 请求，那么 CPU1 就会先把 b 的最新值读到本地，完成变量 b 的值的更新，从而跳出第 9 行的循环，继续执行第 10 行。

这时，CPU1 的 a 缓存区域也处于 Invalid 状态，它也会产生 BusRd 请求，但我们前面分析过，CPU0 中对 a 的赋值可能会晚于 b，所以此时 CPU1 在读取变量 a 的值时，加载的就可能是老的值，也就是 0，那这个时候第 10 行的 assert 就会执行失败。

我们再举一个更极端的例子分析一下：

复制代码

```
// CPU0
void foo() {
    a = 1;
    b = a;
}
```

这个例子中，b 和 a 之间因为有数据依赖，是不可能乱序执行的，这就意味着上面我们分析的情况一是不会发生的。但由于 store buffer 的存在，情况二仍然可能发生，其结果就像我们上面分析的那样，CPU 执行第 10 行时会失败。这会让人感到更加匪夷所思。

为了解决这个问题，CPU 设计者就引入了**内存屏障，屏障的作用是前边的读写操作未完成的情况下，后面的读写操作不能发生**。这就是 Arm 上 dmb 指令的由来，它是数据内存屏障（Data Memory Barrier）的缩写。

我们还是继续沿用前面 CPU0 和 CPU1 的例子，不过这一次我加入了内存屏障：

复制代码

```
// CPU0
void foo() {
    a = 1;
    smp_mb();
    b = 1;
}

// CPU1
void bar() {
    while (b == 0) continue;
    assert(a == 1);
}
```

在这里，smp_mb 就代表了多核体系结构上的内存屏障。由于在不同的体系结构上，指令各不相同，我们使用一个函数对它进行封装。加上这一道屏障以后，CPU 会保证 a 和 b 的赋值指令不会乱序执行，同时写入 cache 的顺序也与程序代码保持一致。

所以说，内存屏障保证了，其他 CPU 能观察到 CPU0 按照我们期望的顺序更新变量。

总的来说，store buffer 的存在是为提升写性能，放弃了缓存的顺序一致性，我们把这种现象称为**弱缓存一致性**。在正常的程序中，多个 CPU 一起操作同一个变量的情况是比较少的，所以 store buffer 可以大大提升程序的运行性能。但在需要核间同步的情况下，我们还是需要这种一致性的，这就需要软件工程师自己在合适的地方添加内存屏障了。

好了，到这里你可能也发现了，我们前面说的都是 CPU 核间同步的“写”的问题，但是核间同步还有另外一个瓶颈，也就是“读”的问题。那这个又要怎么解决呢？我们现在就来看看。

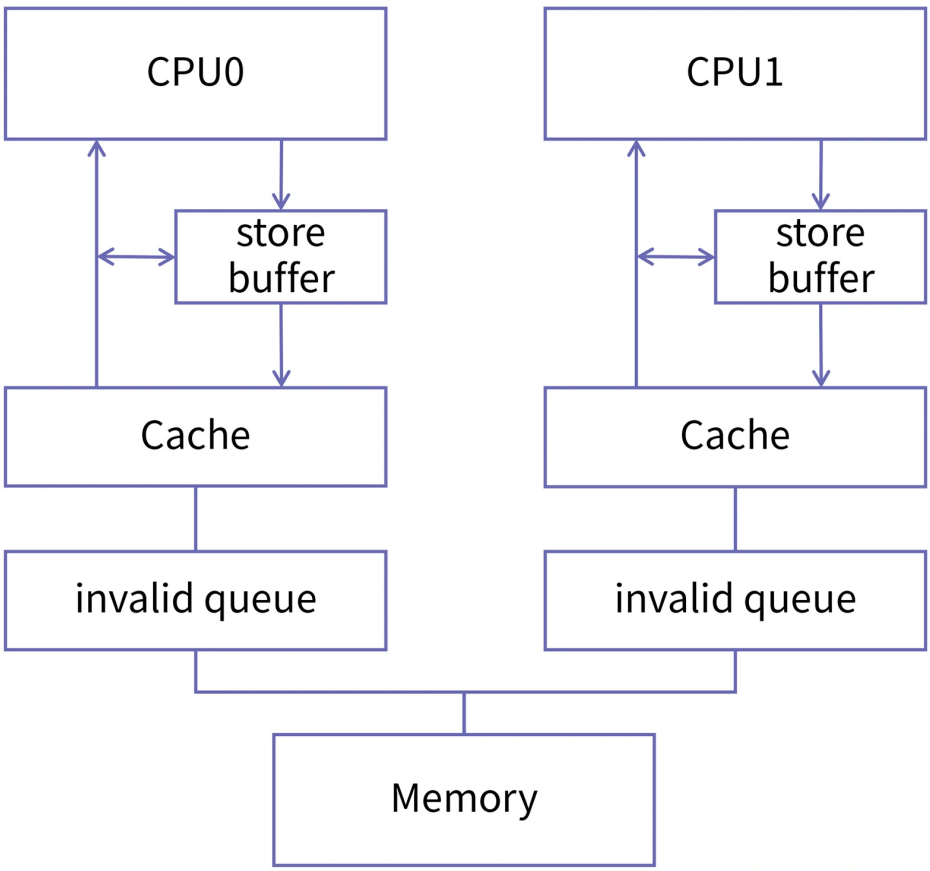
失效队列与读屏障

我们前面说过，当一个 CPU 向同伴发出 Invalid 消息的时候，它的同伴要先把自己的缓存置为 Invalid，然后再发出 acknowledgement。这个从“把缓存置为 Invalid”到“发出 acknowledgement”的过程所需要的时间也是比较长的。

而且，由于 store buffer 的存在提升了写入速度，那么 invalid 消息确认速度相比起来就慢了，这就带来了速度的不匹配，很容易导致 store buffer 的内容还没有及时更新到 cache 里，自己的容量就被撑爆了，从而失去了加速的作用。

为了解决这个问题，CPU 设计者又引入了**“invalid queue”**，也就是失效队列这个结构。加入了这个结构后，收到 Invalid 消息的 CPU，比如我们称它为 CPU1，在收到 Invalid 消息时立即向 CPU0 发回确认消息，但这个时候 CPU1 并没有把自己的 cache 由 Share 置为 Invalid，而是把这个失效的消息放到一个队列中，等到空闲的时候再去处理失效消息，这个队列就是 invalid queue。

经过这样的改进后，CPU1 响应失效消息的速度大大提升了，带有 invalid queue 的缓存结构是这样的：



我们还是以前面 CPU0 和 CPU1 中的例子来做说明。

假如，CPU0 和 CPU1 的缓存中都有变量 a 的副本，也就是说变量 a 所对应的缓存行在 CPU0 和 CPU1 中都是 Share 状态。CPU1 中没有变量 b 的副本，b 所对应缓存在 CPU0 中是 Exclusive 状态。

当 CPU0 在将变量 a 写入缓存的时候，会产生 Invalid 消息，这个消息到达 CPU1 以后，CPU1 不再立即处理它了，而是将这个消息放入 invalid queue，并且立即向 CPU0 回复了 invalid acknowledgement 消息。

CPU0 在得到这个确认消息以后，就可以独占该缓存了，直接将这块缓存变为 Modified 状态，然后把 a 写入。在 a 写入以后，foo 函数中的内存屏障就可以顺利通过了，接下来就可以写入变量 b 的新值。由于 b 是 Exclusive 的，它的更新比较简单，你可以自己思考一下。

接下来我们再看 CPU1 中的操作。

当 CPU1 发起对 b 的请求时，由于 b 不在缓存中，所以它会向总线发出 BusRd 请求，总线会把 CPU0 缓存中的 b 的新值 1 更新到 CPU1。同时，b 所在的缓存行在两个 CPU 中都变为 Share 状态。

CPU1 得到了 b 的新值以后，就可以退出第 10 行的 while 循环，然后对 a 的值进行判断。但是由于 a 的 Invalid 消息还在 invalid queue 里，没有被及时处理，CPU1 还是会使用自己的 Cache 中的 a 的原来的值，也就是 0，这就出错了。

你会发现，在这个过程中，虽然 CPU1 并没有乱序执行两条读指令，但是实际产生的效果却好像是先读到了 b 的值，后读到了 a 的值。如果是在严格遵守 MESI 协议的 CPU 中，CPU0 一定要确保 a 的值先更新到 CPU1，然后才能继续对 b 赋值。但是放宽了缓存一致性以后，这段代码就有问题了。

解决的方法和写屏障的思路是一样的，我们需要引入一个内存屏障，它会让 CPU 暂停执行，直到它处理完 invalid queue 中的失效消息之后，CPU 才会重新开始执行，例如：

复制代码

```
1  // CPU0
2  void foo() {
3      a = 1;
4      smp_mb();
5      b = 1;
6  }
7
8  // CPU1
9  void bar() {
10     while (b == 0) continue;
11     smp_mb();
12     assert(a == 1);
13 }
```

你看，这样在 bar 函数里增加了内存屏障以后，我们就可以保证 a 的新值是一定能读到的了。可见 smp_mb 可以同时对 store buffer 和 invalid queue 施加影响。

不过呢，你可能也会发现，在这个例子中，其实我们也不需要 smp_mb 有那么大的作用。我们只需要在第 4 行保证 store buffer 写入的顺序，在第 11 行保证 invalid queue 的顺序就好了。所以 smp_mb 相对于我们的需求来说，做的事情过多了，这也会导致不必要的性能下降。面对这种情况，CPU 的设计者也进一步提供了单独的写屏障和读屏障。

读写屏障分离

分离的写屏障和读屏障的出现，是为了**更加精细地控制 store buffer 和 invalid queue 的顺序。**

再具体一点，写屏障的作用是让屏障前后的写操作都不能翻过屏障。也就是说，写屏障之前的写操作一定会比之后的写操作先写到缓存中。

读屏障的作用也是类似的，就是保证屏障前后的读操作都不能翻过屏障。假如屏障的前后都有缓存失效的信息，那屏障之前的失效信息一定会优先处理，也就意味着变量的新值一定会被优先更新。

这里我们讨论的都是读写屏障对 store buffer 和 invalid queue 的影响。其实，这里还隐含了一个事实，那就是对 CPU 乱序执行的影响：**写屏障会禁止写操作的乱序。**

这个要求虽然是隐含的，但仔细想一下却是显然的，理由很简单。你想，如果某个 CPU 在进行写操作的时候，实际的执行顺序都是乱序的话，那我们根本就无法讨论新的值什么时候传递到其他 CPU。

而且，分离的读写屏障还有一个好处，就是它可以在需要使用写屏障的时候只使用写屏障，不会给读操作带来负面的影响，这种屏障也可以称为 StoreStore barrier。同理，只使用读屏障也不会对写操作造成影响，这种屏障也可以称为 LoadLoad barrier。例如我们前面 CPU0 和 CPU1 的例子，就可以进一步修改成这样：

复制代码

```
// CPU0
void foo() {
    a = 1;
    smp_wmb();
    b = 1;
}

// CPU1
void bar() {
    while (b == 0) continue;
    smp_rmb();
    assert(a == 1);
}
```

当然，这种修改只有在区分读写屏障的体系结构里才会有作用，比如 alpha 结构。而在 X86 和 Arm 中是没有作用的，这是因为 X86 采用的 TSO 模型不存在缓存一致性的问题，而 Arm 则是采用了另一种称为单向屏障的分类方式。这种单向屏障是怎样的呢？我们也来简单分析一下。

单向屏障

单向屏障 (half-way barrier) 也是一种内存屏障，但它并不是以读写来区分的，而是**像单行道一样，只允许单向通行**，例如 Arm 中的 stlr 和 ldar 指令就是这样。

stlr 的全称是 store release register，也就是以 release 语义将寄存器的值写入内存；ldar 的全称是 load acquire register，也就是以 acquire 语义从内存中将值加载入寄存器。我们重点就来看看 release 和 acquire 语义。

首先是 **release 语义**。如果我们采用了带有 release 语义的写内存指令，那么这个屏障之前的所有读写都不能发生在这次写操作之后，相当于在这次写操作之前施加了一个内存屏障。但它并不能保证屏障之后的读写操作不会前移。简单说，它的特点是**挡前不挡后**。

在支持乱序执行的 CPU（当前高性能多核 CPU 基本都支持乱序执行）中，使用 release 语义的写内存指令比使用全量的 dmb 要有更好的性能。

需要注意的是，stlr 指令除了具有 StoreStore 的功能，它同时还有 LoadStore 的功能。LoadStore barrier 可以解决的问题是真实场景中比较少见的，所以在这里我们就先不关心它了。对于最常用的 StoreStore 的问题，我们在 Arm 中经常使用 stlr 这条带有 release 语义的写指令来解决，尽管它的能力相比我们的诉求还是大了一些。

接着我们再来看一下与 release 语义相对应的 **acquire 语义**。它的作用是这个屏障之后的所有读写都不能发生在 barrier 之前，但它不管这个屏障之前的读写操作。简单说就是**挡后不挡前**。

与 stlr 相对称的是，它同时具备 LoadLoad barrier 的能力和 StoreLoad barrier 的能力。在实际场景中，我们使用最多的还是 LoadLoad barrier，此时我们会使用 ldar 来代替。

总结

好了，今天我们这节课的内容就讲完了，我们简单回顾一下。

在这节课，我们讲解了在 CPU 的具体实现中，通过放宽 MESI 协议的限制来获得性能提升。具体来说，我们引入了 store buffer 和 invalid queue，它们采用放宽 MESI 协议要求的办法，提升了写缓存核间同步的速度，从而提升了程序整体的运行速度。

但在这放宽的过程中，我们也看到会不断地出现新的问题，也就是说，一个 CPU 的读写操作在其他 CPU 看来出现了乱序。甚至，即使执行写操作的 CPU 并没有乱序执行，但是其他 CPU 观察到的变量更新顺序确实是乱序的。这个时候，我们就必须加入内存屏障来解决这个问题。

最后我们也学习了读写屏障分离和单向屏障。在不同的情况下，我们需要的内存屏障是不同的。使用功能强大的内存屏障会给系统带来不必要的性能下降，为了更精细地区分不同类型的屏障，CPU 的设计者们提供了分离的读写屏障 (alpha)，或者是单向屏障 (Arm)。

如何正确地使用内存屏障是一件很考验功底的事情，如果该加的地方没加，会带来非常严重的正确性问题。在操作系统，数据库，编译器等领域，会产生非常深远的影响，其代价甚至是完全无法接受的。而在不需要加的地方，如果你施加了比较重的屏障则可能带来性能下降，成为系统瓶颈。关于读写屏障更多的实际应用案例，你可以参考下我们华为 JDK 公众号发布的这篇文章。

思考题

假如以下代码是 Java 代码，你可以看到，代码中采用了 full fence 来保证缓存一致性。请阅读 Java 的相关 API 文档并思考，fullFence 是否合理？如果不合理，应该使用哪个 API 对它进行替代呢？欢迎在留言区分享你的想法，我在留言区等你。

复制代码

```
// CPU0
void foo() {
    a = 1;
    unsafe.fullFence();
    b = 1;
}

// CPU1
void bar() {
    while (b == 0) continue;
```

```
11      unsafe.fullFence();
12      assert(a == 1);
13  }
```

其实，除此之外，上面这段代码还有一种改法。我给你一个小提示，就是使用 volatile 关键字。我们会在第 17 课对 volatile 关键字进行讲解，如果你有兴趣也可以提前预习一下。

吊打面试官

- 了解Unsafe类里的API吗？解释一下fullFence和loadFence的区别？

Java中的Unsafe类主要提供一些低级别，不安全操作的方法。它提供了访问系统内存资源，调用系统接口，甚至是直接和硬件打交道的能力。由于Unsafe为Java提供了类似C直接访问指针的能力，这增加了程序发生问题的风险。

fullFence可以保证所有读写的顺序一致性，而loadFence只保证写后读和读后读的顺序一致性，不保证读后写和写后写的顺序一致性。

在Java程序员的岗位面试中 极客时间

欢迎你把这节课分享给更多对 MESI 协议和内存屏障感兴趣的人。我是海纳，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 15 | MESI协议：多核CPU是如何同步高速缓存的？

下一篇 不定期福利第一期 | 海纳：我是如何学习计算机知识的？

精选留言 (7)

写留言

八台上
2021-12-05

想问一下在程序中使用各种锁锁，和本节讲的内存屏障有什么关系吗

shenglin
2021-12-03

思考题实例代码使用fullFence()正确性是可以保证的，只是性能下降？

```
// CPU0
void foo() {
    a = 1;...
```

展开

shenglin
2021-12-03

请问一下， StoreStore barrier, LoadStore barrier, LoadLoad barrier, StoreLoad barrier具体含义是什么

.

2021-12-03

我总算等到这节课。。。。。。。。老师打算就java方面出类似的教程吗？在这块网上的比较零散。


csyangchsh

2021-12-03

这篇文章很棒，如果执行过程加上图示九更好了。

展开


费城的二鹏

2021-12-03

思考题第一瞬间想到的就是 volatile 结果它不是答案。

作者回复: 是答案呀。你看下文的提示里有讲的哦。


一塌糊涂

2021-12-03

必须赞，看过很多资料，唯一能讲明白的！大神推荐点资料吧。

作者回复: 多谢。这些内容是从很多文章里以及CPU源文件，还有向CPU设计者请教得到的心得。所以，我一下子也很难说得上来有什么资料。我觉得这个专栏本身可能就是目前的中文互联网上的一个比较好的资料吧。至于说这一节课的内容，核心思想是来自这篇文章:《Memory Barriers: a Hardware View for Software Hackers》，但我并不建议你去看这篇文章，因为这篇文章里在讲到MESI状态变化的时候过于复杂了。