内存屏障的来历

王大龙
一切从专注开始 :)

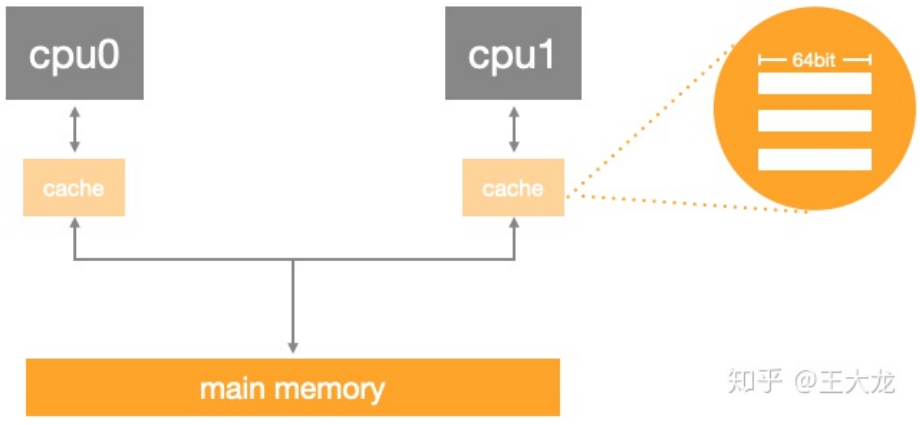
已关注

109 人赞同了该文章

前置内容：

缓存一致性协议的工作方式
34 赞同 · 7 评论 文章

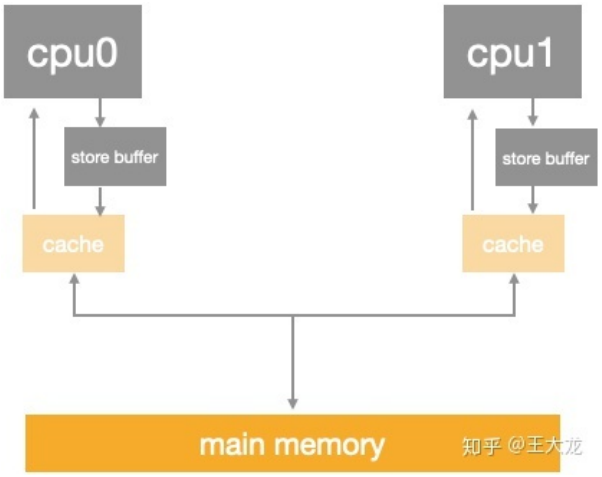
在「前置内容」中，我们了解到「最原始」的cpu是如何在缓存一致性协议MESI的指导下工作的，同时我们也发现此时cpu的性能因为同步请求远程（其他cpu）数据而大打折扣。本文章将介绍cpu的优化过程，你将了解到硬件层面的优化——Store Buffer, Invalid Queue，以及软件层面的优化——cpu内存屏障指令。



最原始的架构

性能优化之旅——Store Buffer

当cpu需要的数据在其他cpu的cache内时，需要请求，并且等待响应，这显然是一个同步行为，优化的方案也很明显，采用异步。思路大概是在cpu和cache之间加一个store buffer，cpu可以先将数据写到store buffer，同时给其他cpu发送消息，然后继续做其它事情，等到收到其它cpu发过来的响应消息，再将数据从store buffer移到cache line。

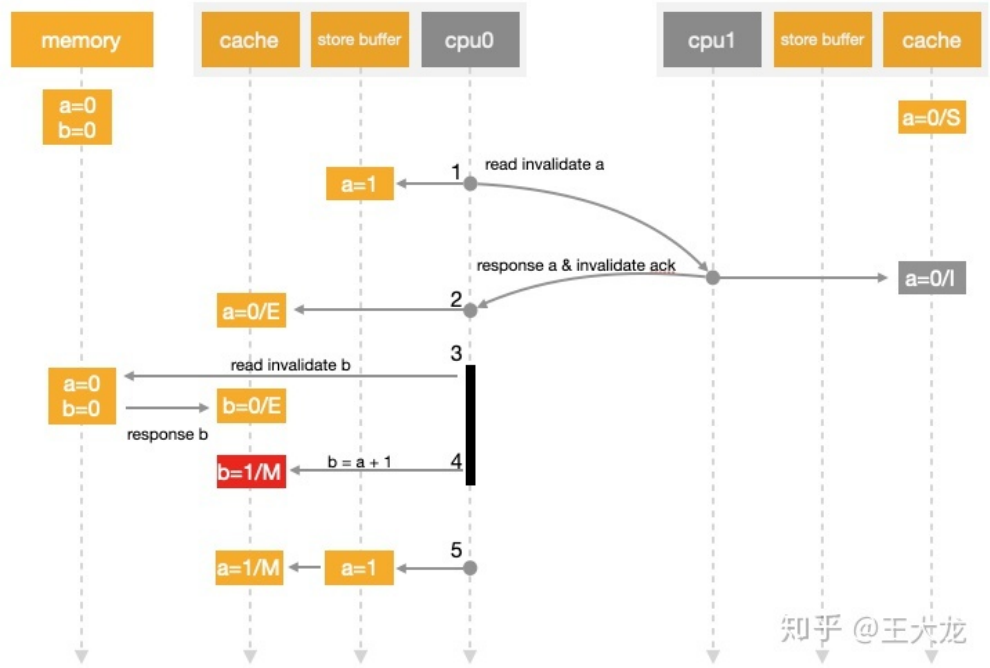


增加store buffer

该方案听起来不错！但逻辑上有漏洞，需要细化，我们来看几个漏洞。比如有如下代码:

```
a = 1;
b = a + 1;
assert(b == 2);
```

初始状态下，假设a，b值都为0，并且a存在cpu1的cache line中(Shared状态)，可能出现如下操作序列:



代码执行时序图，表明cache line的变化

1. cpu0 要写入a，将a=1写入store buffer，并发出Read Invalidate消息，继续其他指令。
2. cpu1 收到Read Invalidate，返回Read Response(包含a=0的cache line)和Invalidate ACK，cpu0 收到Read Response，更新cache line(a=0)。
3. cpu0 开始执行b=a+1，此时cache line中还没有加载b，于是发出Read Invalidate消息，从内存加载b=0，同时cache line中已有a=0，于是得到b=1，状态为Modified状态。
4. cpu0 得到 b=1，断言失败。
5. cpu0 将store buffer中的a=1推送到cache line，然而为时已晚。

造成这个问题的根源在于对同一个cpu存在对a的两份拷贝，一份在cache，一份在store buffer，而cpu计算b=a+1时，a和b的值都来自cache。仿佛代码的执行顺序变成了这个样子：

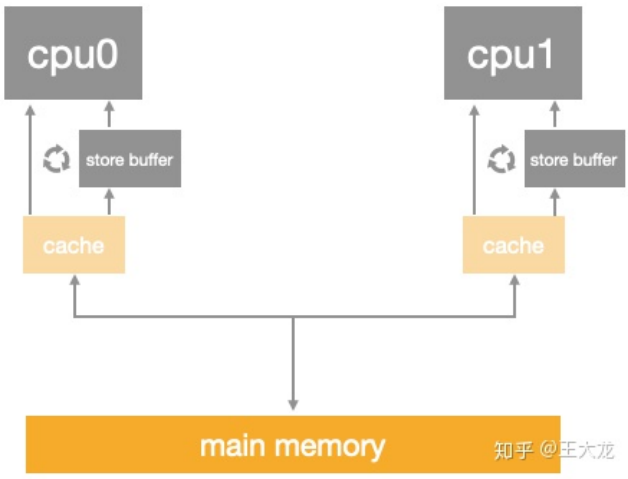
```
b = a + 1;
a = 1;
assert(b == 2);
```

这个问题需要优化。

> 思考题：以上操作序列只是其中一种可能，具体响应（也就是操作2）什么时候到来是不确定的，那么假如响应在cpu0计算“b=a+1”之后到来会发生什么？

性能优化之旅——Store Forwarding

store buffer可能导致破坏程序顺序的问题，硬件工程师在store buffer的基础上，又实现了”store forwarding”技术: cpu可以直接从store buffer中加载数据，即支持将cpu存入store buffer的数据传递(forwarding)给后续的加载操作，而不经由cache。

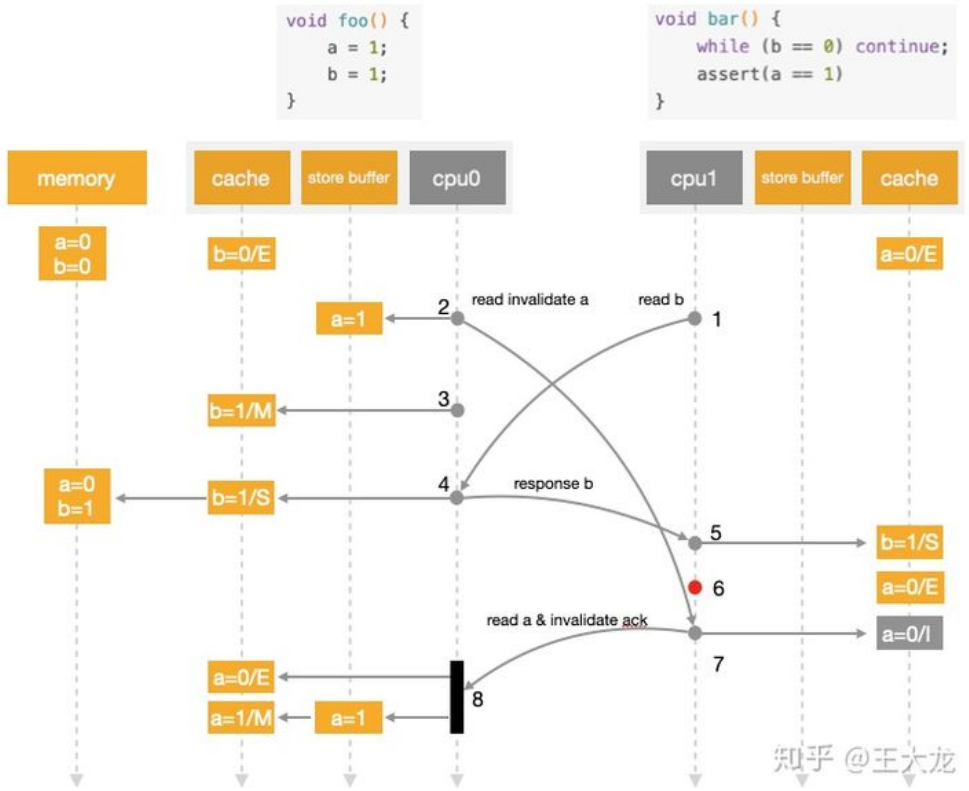


虽然现在解决了同一个cpu读写数据的问题，但还是有漏洞，来看看并发程序:

```
void foo() {
    a = 1;
    b = 1;
}
void bar() {
    while (b == 0) continue;
```

```
    assert(a == 1)
}
```

初始状态下，假设a，b值都为0，a存在于cpu1的cache中，b存在于cpu0的cache中，均为Exclusive状态，cpu0执行foo函数，cpu1执行bar函数，上面代码的预期断言为真。那么来看下执行序列:



代码执行时序图

1. cpu1执行while(b == 0)，由于cpu1的Cache中没有b，发出Read b消息
2. cpu0执行a=1，由于cpu0的cache中没有a，因此它将a(当前值1)写入到store buffer并发出Read Invalidate a消息
3. cpu0执行b=1，由于b已经存在在cache中，且为Exclusive状态，因此可直接执行写入
4. cpu0收到Read b消息，将cache中的b(当前值1)返回给cpu1，将b写回到内存，并将cache Line状态改为Shared
5. cpu1收到包含b的cache line，结束while (b == 0)循环
6. cpu1执行assert(a == 1)，由于此时cpu1 cache line中的a仍然为0并且有效(Exclusive)，断言失败
7. cpu1收到Read Invalidate a消息，返回包含a的cache line，并将本地包含a的cache line置为Invalid，然而已经为时已晚。
8. cpu0收到cpu1传过来的cache line，然后将store buffer中的a(当前值1)刷新到cache line

出现这个问题的原因在于cpu不知道a, b之间的数据依赖，cpu0对a的写入需要和其他cpu通信，因此有延迟，而对b的写入直接修改本地cache就行，因此b比a先在cache中生效，导致cpu1读到b=1时，a还存在于store buffer中。从代码的角度来看，foo函数似乎变成了这个样子:

```
void foo() {
    b = 1;
```

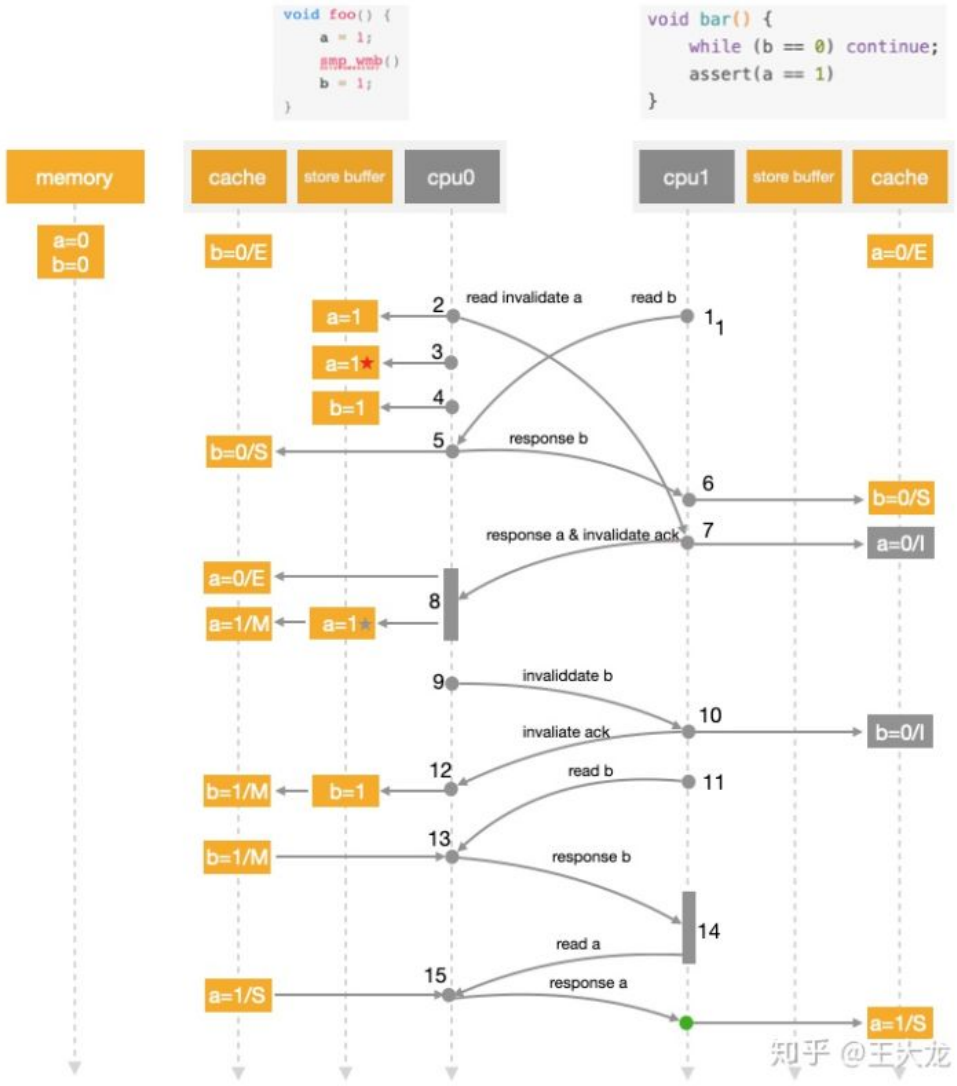
```
    a = 1;
}
```

foo函数的代码，即使是store forwarding也阻止不了它被cpu“重排”，虽然这并没有影响foo函数的正确性，但会影响到所有依赖foo函数赋值顺序的线程。看来还要继续优化！

性能优化之旅——写屏障指令

到目前为止，我们发现了”指令重排“的其中一个本质[1]，cpu为了优化指令的执行效率，引入了store buffer（forwarding），而又因此导致了指令执行顺序的变化。要保证这种顺序一致性，靠硬件是优化不了了，需要在软件层面支持，没错，cpu提供了写屏障（write memory barrier）指令，Linux操作系统将写屏障指令封装成了smp_wmb()函数，cpu执行smp_mb()的思路是，会先把当前store buffer中的数据刷到cache之后，再执行屏障后的“写入操作”，该思路有两种实现方式: 一是简单地刷store buffer，但如果此时远程cache line没有返回，则需要等待，二是将当前store buffer中的条目打标，然后将屏障后的“写入操作”也写到store buffer中，cpu继续干其他的事，当被打标的条目全部刷到cache line，之后再刷后面的条目（具体实现非本文目标），以第二种实现逻辑为例，我们看看以下代码执行过程：

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



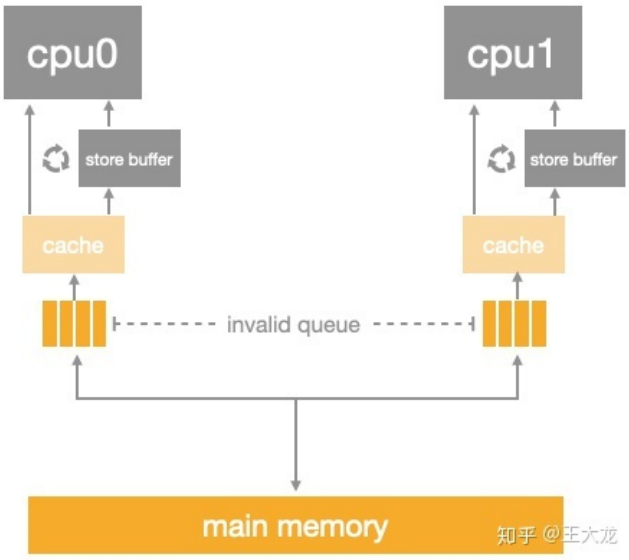
1. cpu1执行while(b == 0)，由于cpu1的cache中没有b，发出Read b消息。
2. cpu0执行a=1，由于cpu0的cache中没有a，因此它将a(当前值1)写入到store buffer并发出Read Invalidate a消息。
3. cpu0看到smp_wmb()内存屏障，它会标记当前store buffer中的所有条目(即a=1被标记)。
4. cpu0执行b=1，尽管b已经存在在cache中(Exclusive)，但是由于store buffer中还存在被标记的条目，因此b不能直接写入，只能先写入store buffer中。
5. cpu0收到Read b消息，将cache中的b(当前值0)返回给cpu1，将b写回到内存，并将cache line状态改为Shared。
6. cpu1收到包含b的cache line，继续while (b == 0)循环。
7. cpu1收到Read Invalidate a消息，返回包含a的cache line，并将本地的cache line置为Invalid。
8. cpu0收到cpu1传过来的包含a的cache line，然后将store buffer中的a(当前值1)刷新到cache line，并且将cache line状态置为Modified。
9. 由于cpu0的store buffer中被标记的条目已经全部刷新到cache，此时cpu0可以尝试将store buffer中的b=1刷新到cache，但是由于包含B的cache line已经不是Exclusive而是Shared，因此需要先发Invalidate b消息。
10. cpu1收到Invalidate b消息，将包含b的cache line置为Invalid，返回Invalidate ACK。
11. cpu1继续执行while(b == 0)，此时b已经不在cache中，因此发出Read消息。
12. cpu0收到Invalidate ACK，将store buffer中的b=1写入Cache。
13. cpu0收到Read消息，返回包含b新值的cache line。
14. cpu1收到包含b的cache line，可以继续执行while(b == 0)，终止循环，然后执行

assert(a == 1)，此时a不在其cache中，因此发出Read消息。

15. cpu0收到Read消息，返回包含a新值的cache line。
16. cpu1收到包含a的cache line，断言为真。

性能优化之旅——Invalid Queue

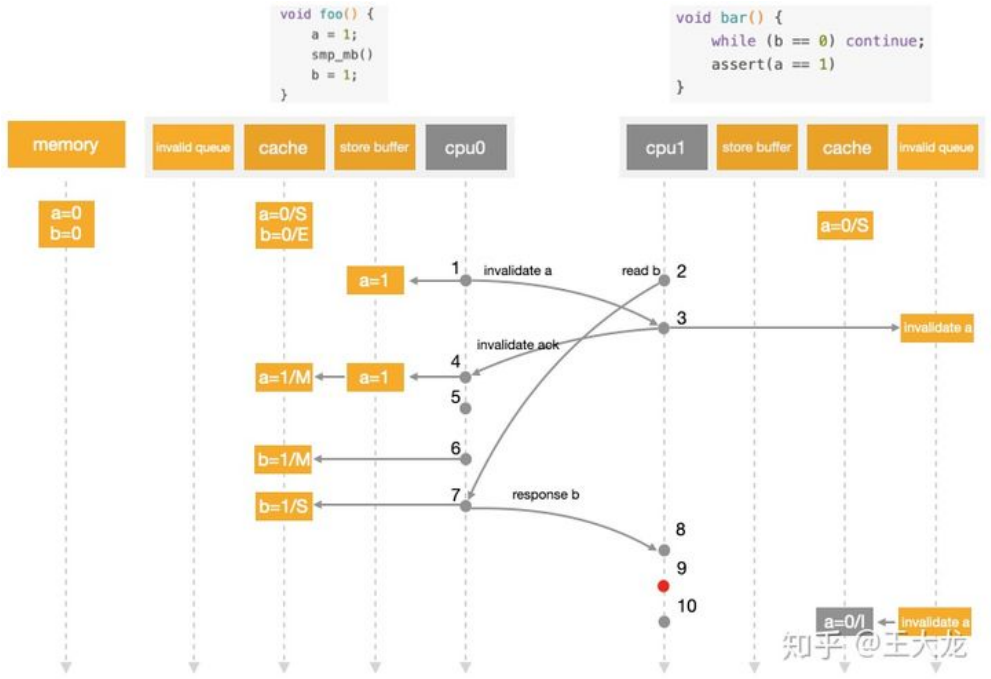
引入了store buffer，再辅以store forwarding，写屏障，看起来好像可以自洽了，然而还有一个问题没有考虑: store buffer的大小是有限的，所有的写入操作发生cache missing（数据不再本地）都会使用store buffer，特别是出现内存屏障时，后续的所有写入操作(不管是否cache missing)都会挤压在store buffer中(直到store buffer中屏障前的条目处理完)，因此store buffer很容易会满，当store buffer满了之后，cpu还是会卡在等对应的Invalidate ACK以处理store buffer中的条目。因此还是要回到Invalidate ACK中来，Invalidate ACK耗时的主要原因是cpu要先将对应的cache line置为Invalid后再返回Invalidate ACK，一个很忙的cpu可能会导致其它cpu都在等它回Invalidate ACK。解决思路还是化同步为异步: cpu不必要处理了cache line之后才回Invalidate ACK，而是可以先将Invalid消息放到某个请求队列Invalid Queue，然后就返回Invalidate ACK。CPU可以后续再处理Invalid Queue中的消息，大幅度降低Invalidate ACK响应时间。此时的CPU Cache结构图如下：



加入了invalid queue之后，cpu在处理任何cache line的MSEI状态前，都必须先看invalid queue中是否有该cache line的Invalid消息没有处理。另外，它也再一次破坏了内存的一致性。请看代码：

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

仍然假设a, b的初始值为0，a在cpu0，cpu1中均为Shared状态，b在cpu0独占(Exclusive状态)，cpu0执行foo，cpu1执行bar:



1. cpu0执行a=1，由于其有包含a的cache line，将a写入store buffer，并发出Invalidate a消息。
2. cpu1执行while(b == 0)，它没有b的cache，发出Read b消息。
3. cpu1收到cpu0的Invalidate a消息，将其放入Invalidate Queue，返回Invalidate ACK。
4. cpu0收到Invalidate ACK，将store buffer中的a=1刷新到cache line，标记为Modified。
5. cpu0看到smp_wmb()内存屏障，但是由于其store buffer为空，因此它可以直接跳过该语句。
6. cpu0执行b=1，由于其cache独占b，因此直接执行写入，cache line标记为Modified。
7. cpu0收到cpu1发的Read b消息，将包含b的cache line写回内存并返回该cache line，本地的cache line标记为Shared。
8. cpu1收到包含b(当前值1)的cache line，结束while循环。
9. cpu1执行assert(a == 1)，由于其本地有包含a旧值的cache line，读到a初始值0，断言失败。
10. cpu1这时才处理Invalid Queue中的消息，将包含a旧值的cache line置为Invalid。

问题在于第9步中cpu1在读取a的cache line时，没有先处理Invalid Queue中该cache line的Invalid操作，怎么办？其实cpu还提供了**读屏障指令**，Linux将其封装成smp_rmb()函数，将该函数插入到bar函数中，就像这样：

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

和smp_wmb()类似，cpu执行smp_rmb()的时，会先把当前invalidate queue中的数据处理掉之后，再执行屏障后的“读取操作”，具体操作序列不再赘述。

内存屏障

到目前为止，我们已经接触到了「读屏障」和「写屏障」，本节我们就专门聊聊内存屏障。首先内存屏障有几个不同的种类，比如「读屏障」、「写屏障」、「全屏障」等等，各个cpu厂商可以自行选择支持哪些语义以及如何支持，所以不同的cpu支持的语义不同，对应的指令也不同，我们来看看下图：

	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache/Pipeline?
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64				Y				
IA64	Y	Y	Y	Y	Y	Y		Y
(PA-RISC)	Y	Y	Y	Y				
PA-RISC CPUs								
POWER™	Y	Y	Y	Y	Y	Y		Y
(SPARC RMO)	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86				Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries®				Y				Y

横轴表示8个内存屏障语义，纵轴表示不同的cpu，此图表示不同cpu对内存屏障语义的支持情况。事实上作为应用开发的程序员没必要再继续深入了，Linux操作系统面向cpu抽象出了自己的一套内存屏障函数，它们分别是：

- smp_rmb(): 在invalid queue的数据被刷完之后再执行屏障后的读操作。
- smp_wmb(): 在store buffer的数据被刷完之后再执行屏障后的写操作。
- smp_mb(): 同时具有读屏障和写屏障功能。
- smp_read_barrier_depends() that forces subsequent operations that depend on prior operations to be ordered. This primitive is a no-op on all platforms except Alpha.
- mmiowb() that forces ordering on MMIO writes that are guarded by global

spinlocks. This primitive is a no-op on all platforms on which the memory barriers in spinlocks already enforce MMIO ordering. The platforms with a nonno-op mmiowb() definition include some (but not all) IA64, FRV, MIPS, and SH systems. This primitive is relatively new, so relatively few drivers take advantage of it.

后面两个不想翻译了，等用到了再了解。

总结

本文通过cpu的优化过程，最终引出内存屏障的概念，可以看出，内存屏障是硬件之上，操作系统之下对一致性的最后一层保障。另外，我在看原论文中对执行序列的描述感觉头很大，所以想出用时序图的方式去表示代码执行过程中各cache line的状态变化，希望对你的理解有所帮助。

Memory Barriers: a Hardware View for Software Hackers
www.puppetmastertrading.com/images/hwViewForSw...

Cache一致性和内存模型
wudaijun.com/2019/04/cpu-cache-and-...

参考

1. ^ 「指令重排」分为两种类型，一种是「主动的」，编译器会主动重排代码使得特定的cpu执行更快。另外一个类型是「被动的」，为了异步化指令的执行，引入Store Buffer和Invalidate Queue，却导致了「指令顺序改变」的副作用。

编辑于 2020-04-11 11:07

多线程 并发 中央处理器 (CPU)

文章被以下专栏收录

编程

推荐阅读

MESI与内存屏障

betterfishXL
为什么需要内存屏障

现代的CPU比内存系统快很多，2006年的cpu可以在一纳秒之内执行10条指令，但是需要多个十纳秒去从内存读取一个数据，这里面产生了至少两个数量级的速度差距。

闻道

小知 发表于Java知...
多线程真的会使用CPU所有的

8 条评论 ⇌ 切换为时间排序

写下你的评论...

Finch 2021-09-11
大佬，有个疑问，【写屏障指令】这一节里，第5步里需要“将b写回到内存”。看图示里好像也没有写会到内存里，谢谢好文
赞

神探大厨 回复 Finch 02-07
此时不必做写回b操作，b还在store buffer里
赞

syng 2021-09-03
X86有invalidate queue?
赞

郑鹏 2021-07-18
大佬，假如两个cpu同时修改a，一个执行a=1，一个执行a=2，都将自己的结果写入store buffer，随后发送read Invalidate操作，那么两个cpu的操作结果是一个成功，一个失败吗?
赞

戴宗 回复 郑鹏 2021-09-03
1，同一个内存地址不能同时被两个u锁定（E/M状态），持有锁的那个u会先执行，没锁的等有锁的解锁回写后再执行
2，假设两个u都没持有锁，就都会向总线发起锁定请求，而总线同一时间只会响应1个请求，（以下部分为我大学课程的记忆，不保证准确）如果锁定请求存在些微的时间差，则谁先请求谁先锁定，然后回答情况1；如果非常凑巧刚好在同一个时钟周期同时发起锁定请求，则会由总线仲裁，默认是cpu编号小的优先拿到锁，仲裁失败的u等待锁释放
2

戴宗 回复 郑鹏 2021-09-03
补充：以上皆基于正确加载了读写屏障的前提，否则会后执行的覆盖先执行的🤣
赞

换宇 2020-12-14
👍
赞

暖暖 2020-08-09
很详细
赞