从volatile到分布式CPU架构优化

从一道常见面试题开始

volatile如何解决可见性问题?

查看了不少文章和书籍，发现其中绝大多数只是从JMM的层面（工作内存和主内存）解释可见性问题的处理，对于可见性问题在硬件方面的体现提及较少, 或最多到MESI层面。

好奇心驱使下，继续深入查阅了一些偏硬件的文章，结合CPU架构演化过程，梳理了volatile关键字是如何调用CPU指令集，以及CPU架构解决可见性问题的方案

前言

后续内容会简单讲解volatile在JVM层面如何调用CPU指令，重点是讲解对应的CPU指令如何适配CPU架构演进。

文章也只会讲解volatile在CPU架构层面的表现和原理

一段危险的代码

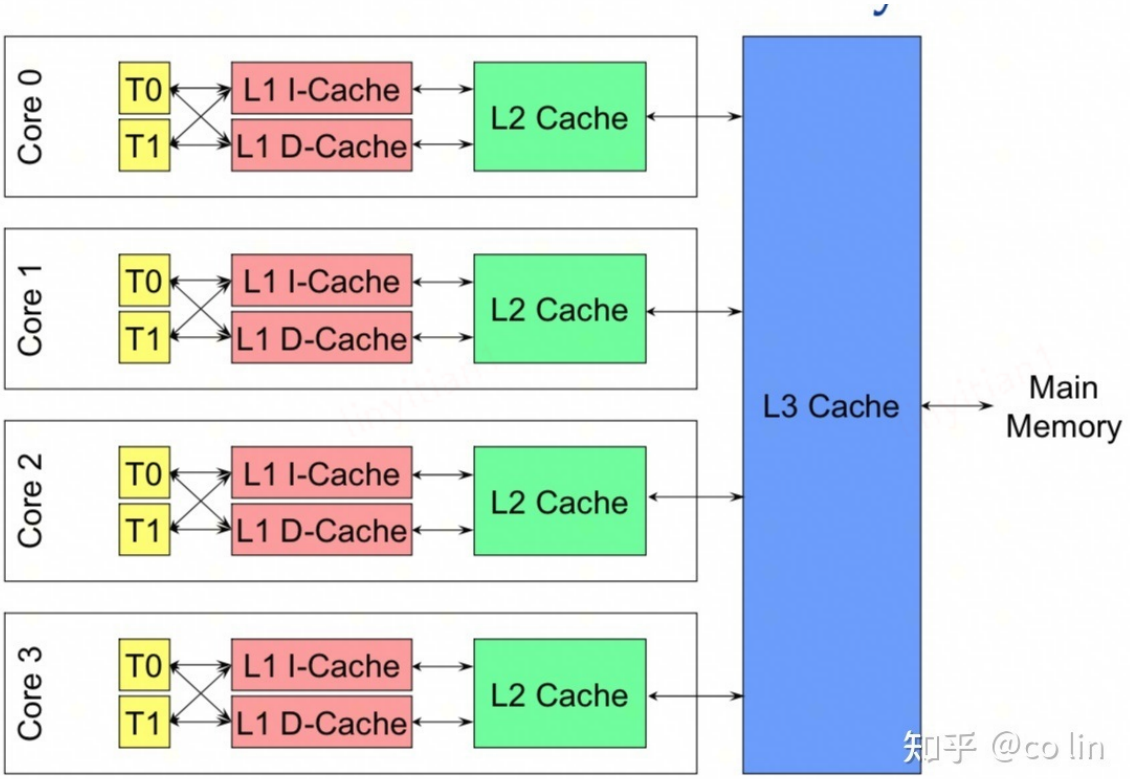
先来一个简单的代码

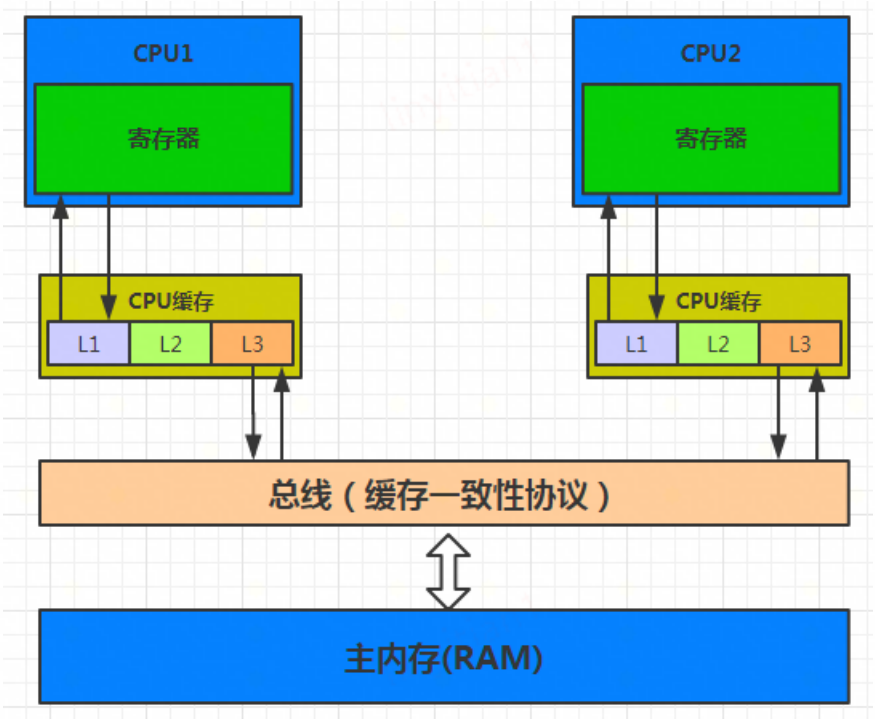
Java

```java
package com.jd.bluedragon;

public class VolatileTest {
    static int x = 0, y = 0;
    static int a = 0, b = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread one = new Thread(new Runnable() {
            public void run() {
                a = 1;
                x = b;
            }
        });

        Thread other = new Thread(new Runnable() {
            public void run() {
                b = 1;
                y = a;
            }
        });
// 两个线程并发执行, 可能产生 x=0;y=0的情况
        one.start();
        other.start();
        one.join();
        other.join();
        System.out.println("(" + x + "," + y + ")");
    }
}
```

上面的代码启动了两个线程, join是为了main线程不会在两个子线程执行完毕之前退出

可见性在CPU层面的体现





先看一下CPU的运作原理

如上图所示, CPU cache是通过总线和主内存交互的.

在早期CPU架构中, 程序指令执行时, CPU会按 [寄存器- CPU cache- 主内存] 的优先级查找所需的数据(数据+算法=程序), 如果缓存中已经存在相关的数据, CPU就会直接用这个缓存进行计算, 而不顾及其他core 或者同core的不同线程的L1 cache可能存在这个数据的并发副本.

回到代码

main方法执行, class被解析到主内存, 然后在线程执行时,在不同core(或core的多个线程)中形成变量 a 和变量 b 的多个副本, 由于系统总线只会把最近的cache中的副本加载到寄存器并计数, 所以可能出现线程one 执行到 x=b 时, b仍为0 , 线程other执行到 y=a时, a仍为0 的情况, 也就是 可见性问题的出现是CPU架构追求性能加入了分布式缓存的结果

在大量的执行样本下, 最后输出的结果可能是 x=0;y=0; 这就是可见性问题的一种表现

给代码中的abxy变量都加上volatile关键字后, 可见性问题消失, 通过javap查看代码的字节码如下:

```
0x0000000011777b7ec: movabs $0x76b159d90,%rsi  ;   {oop(a 'java/lang/Class' = 'com/jd/bluedragon/VolatileTest')}
0x0000000011777b7f6: mov    $0x1,%edi
0x0000000011777b7fb: mov    %edi,0x6c(%rsi)
0x0000000011777b7fe: lock addl $0x0,(%rsp)     ;*putstatic y
                                               ; - com.jd.bluedragon.VolatileTest::main@1 (line 6)

0x0000000011777b803: mov    $0x1,%edi
0x0000000011777b808: mov    %edi,0x68(%rsi)
0x0000000011777b80b: lock addl $0x0,(%rsp)     ;*putstatic x
                                               ; - com.jd.bluedragon.VolatileTest::main@5 (line 7)

0x0000000011777b810: add    $0x30,%rsp
0x0000000011777b814: pop    %rbp
0x0000000011777b815: test   %eax,-0xf53271b(%rip)        # 0x0000000108249100
                                               ;   {poll_return}
0x0000000011777b81b: retq
```

可以看到, 每次变量被修改后, 都会执行一次 lock addl 的指令

lock指令前缀是jmm层面保证可见性的基本工具, 事实上在jvm层面, 会通过宏定义确定当前的CPU类型, 把lock指令适配成对应的CPU指令

那lock指令被执行时CPU架构都做了哪些事情呢?

CPU的努力

为什么加上lock指令前缀就可以解决缓存不一致的问题呢?

在《深入理解Java虚拟机- JVM高级特性与最佳实践》中是这样解释lock指令的作用

> 这句指令中的“addl$0x0，(%esp)”（把ESP寄存器的值加0）显然是一个空操作，之所以用这个空 操作而不是空操作专用指令nop，是因为IA32手册规定lock前缀不允许配合nop指令使用。这里的关键 在于lock前缀，查询IA32手册可知，它的作用是将本处理器的缓存写入了内存，该写入动作也会引起 别的处理器或者别的内核无效化（Invalidate）其缓存，这种操作相当于对缓存中的变量做了一次前面 介绍Java内存模式中所说的“store和write”操作[4]。所以通过这样一个空操作，可让前面volatile变量的 修改对其他处理器立即可见。

语焉不详, 继续查阅IA32的操作手册, 终于发现lock指令的真正作用

> 8.1 LOCKED ATOMIC OPERATIONS
>
> The 32-bit IA-32 processors support locked atomic operations on locations in system memory. These operations are typically used to manage shared data structures (such as semaphores, segment descriptors, system segments, or page tables) in which two o r more processors may try simultaneously to modify the same field or flag. The proces sor uses three interdependent mechanisms for carrying out locked atomic operations:

> - Guaranteed atomic operations
> - Bus locking, using the LOCK# signal and the LOCK instruction prefix
> - Cache coherency protocols that ensure that atomic operations can be carried out on c ached data structures (cache lock); this mechanism is present in the Pentium 4, Intel X eon, and P6 family processors

其中说明了三个原子操作的场景:

- 基本的原子操作, 如32位运算和64位读(64-bit CPU)等
- LOCK# 信号 和 lock指令前缀提供总线锁
- 通过缓存锁实现的缓存一致性协议

也就是说, jre通过把字节码解析为 lock 指令前缀, 使得在CPU执行时用总线锁解决了可见性问题

洪荒时期的解决方案--总线锁

既然CPU都是通过总线读写的缓存, 那直接让当前CPU独占总线是最简单直接的方法, 这就是总线锁

> 总线锁就是使用处理器提供的一个LOCK#信号，当一个处理器在总线上输出此信号，其他处理器的请求将被阻塞，那么该处理器就可以独占共享锁。

总线锁的缺点非常明显, 在锁定期间, CPU和内存的通信都给锁住了，其他处理器不能操作其他内存地址的数据，从而开销较大, 而我们只需要保证同级缓存的数据版本是一致的

为了解决这个问题, CPU架构设计师选择了保证缓存的数据一致, 这就是缓存一致性协议MESI的由来

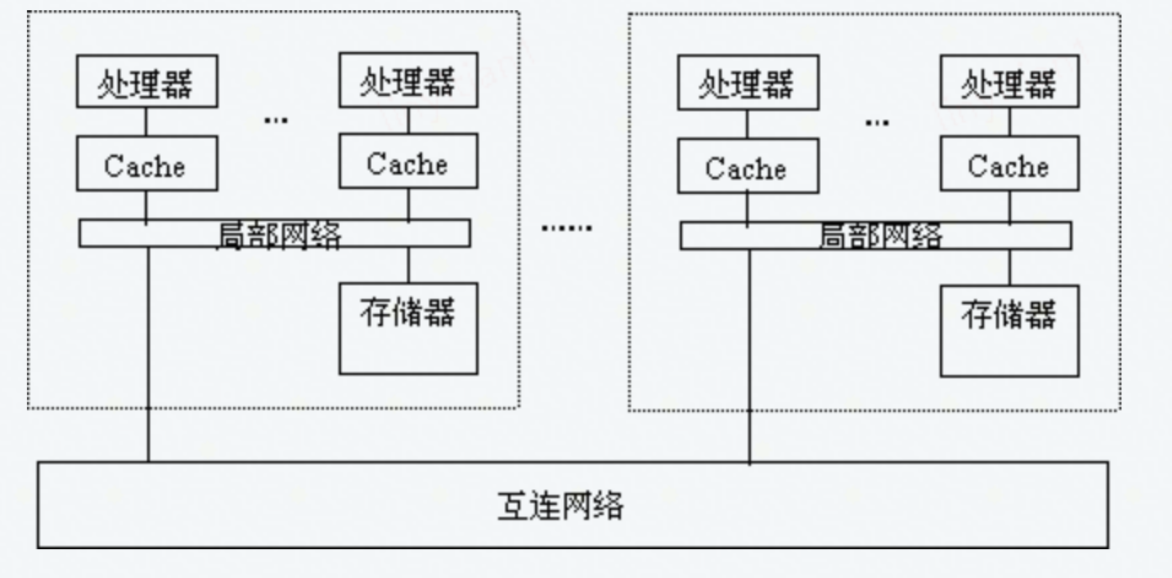
MESI机制

如何避免缓存中副本版本不一致的情况? 答案是总线

总线连接所有缓存和主内存, 通过总线可以实现CPU事件的广播, 同时也可以实现缓存间的状态传递

先来看一下目前支持的两种MESI的实现

> 1. 目录协议：内存地址为key，挂载链表，如果在私有缓存A和私有缓存B上都有key的缓存数据, 则链表内容为 (A+offset)->(B+offset)
> 2. 监听协议：总线接收到CPU的写指令时, 会同时把指令广播给其它私有缓存



目录协议实现如下:

> 当某一处理器修改了私有Cache中的数据后，并非每个处理器的Cache中都有该数据的副本。因此使用广播的方式并不是高效的。而且，并非所有的并行系统都支持广播。 能否只通知那些含有被修改数据的副本的处理器？这就是基于目录的协议的思想。
>
> 在基于目录的协议中，共享存储器维护一个目录，称为高速缓存目录，该目录中记载了申请了某一数据的所有处理器。 这样，当数据被更新时，就根据目录的记载，向所有其Cache中包含该数据的处理器"点对点"地发送无效信息或更新 后数据。目录可以是集中的，也可以分布于各个存储模块上。特别在某些系统内，存储器在逻辑上是共享的，但物理 上是分布的。此时目录可以分布于各存储器内。目录也有多种形式，比如全映射目录，有限目录和链式目录。

> 在全映 射目录方案中，对于每个数据块，为每个处理机都设置一个标志位，这种目录比较庞大。在有限目录限方案中，限制 同时含有同一数据块拷贝的Cache的数量。这样，当含有某一数据块的Cache数目已达限制数量，但仍有Cache申请 该数据块时，就需要从已经含有该数据的Cache中"驱逐" 出一个。因此有限目录需要相应的"驱逐策略"。 在链式目录方案中，在处理器申请数据块的时将含有同一数据块的处理器用指针形成一个链表。通过该链表就可以找 到所有含有该数据块的处理器。有限目录和链式目录更能适应处理器较多的场合，亦即有更好的扩展性.

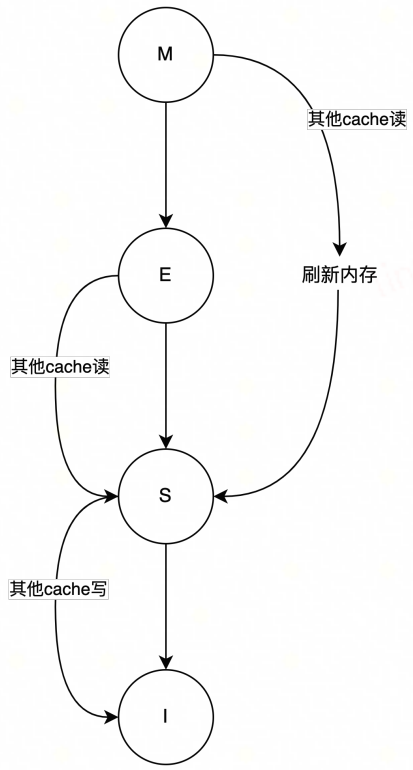
简单来说, 就是在缓存层级之外, 通过共享存储器维护一份加载了各个内存地址的目录, 一般在各个大节点(也就是跨core集群)使用广播协议

> 整个系统分为几个"大结点"，每个"大结点"中含有多个处理器和存储器。每个处理器有私有Cache。 在每个"大结点"内使用监听总线协议，而"大结点"之间用基于目录的协议维护Cache一致性。

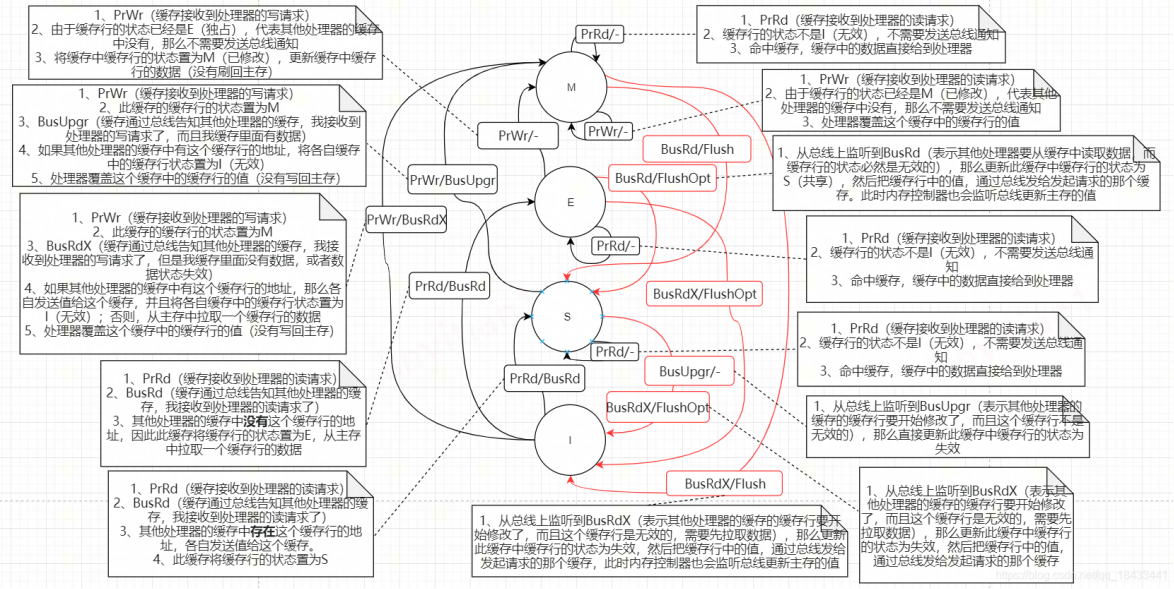
之所以把该协议称为MESI, 是因为设计师通过modified, Exclusive, Shared, Invalid四种状态标识缓存是否可用以及指导缓存如何同步数据版本

状态	描述	监听任务
M 修改 (Modified)	该Cache line有效，数据被修改了，和内存中的数据不一致，数据只存在于本Cache中。	缓存行必须时刻监听所有试图读该缓存行相对就主存的操作，这种操作必须在缓存将该缓存行写回主存并将状态变成S（共享）状态之前被延迟执行。
E 独享、互斥 (Exclusive)	该Cache line有效，数据和内存中的数据一致，数据只存在于本Cache中。	缓存行也必须监听其它缓存读主存中该缓存行的操作，一旦有这种操作，该缓存行需要变成S（共享）状态。
S 共享 (Shared)	该Cache line有效，数据和内存中的数据一致，数据存在于很多Cache中。	缓存行也必须监听其它缓存使该缓存行无效或者独享该缓存行的请求，并将该缓存行变成无效（Invalid）。
I 无效 (Invalid)	该Cache line无效。	无

简单描述如下:



也可以从CPU指令信号层面体会一下更详细的状态迁移机制

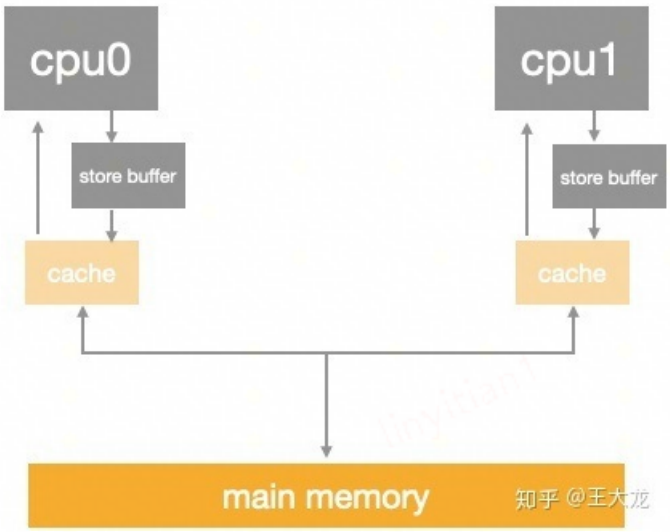


很明显, MESI必须是原子操作, 于是创造了 缓存锁 的机制

> 如果访问的内存区域已经缓存在处理器的缓存行中，处理器则不会声明LOCK#信号，它会对CPU的缓存中的缓存行进行锁定，在锁定期间，其它 CPU 不能同时缓存此数据，在修改之后，通过缓存一致性协议来保证修改的原子性，这个操作被称为“缓存锁”

缓存读写事件通过总线进行了同步广播, 可见性问题应该解决了吧? 其实还没有, 因为技术人员追求性能优化的脚步从不停止, 而每一次进步都是庞大的系统工程

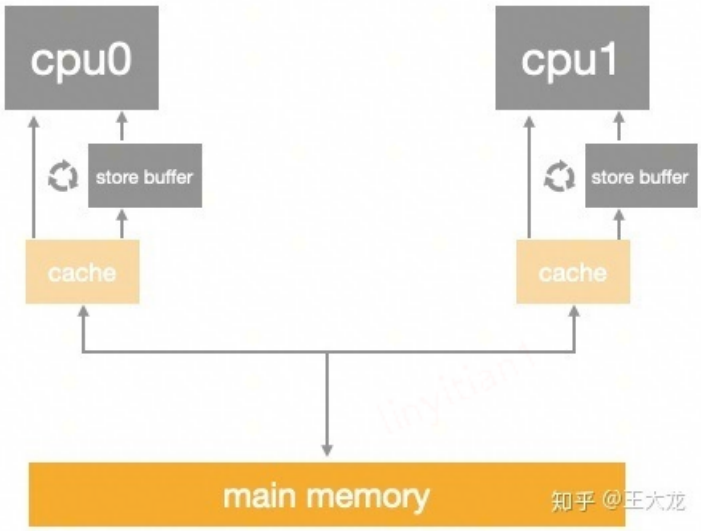
store buffer 和 store fowarding



store buffer是为了进一步提高CPU交互效率而产生的

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。

这个方案的问题是, store buffer只是作为一个数据的暂存工具, CPU core只能对 store buffer进行写操作, 而无法进行读操作, 这其实又造成了明显的不一致的问题- store buffer和L1缓存不一致. 为了解决这个问题，CPU架构师在store buffer的基础上做了进一步优化, 这就是 store fowarding 的来源



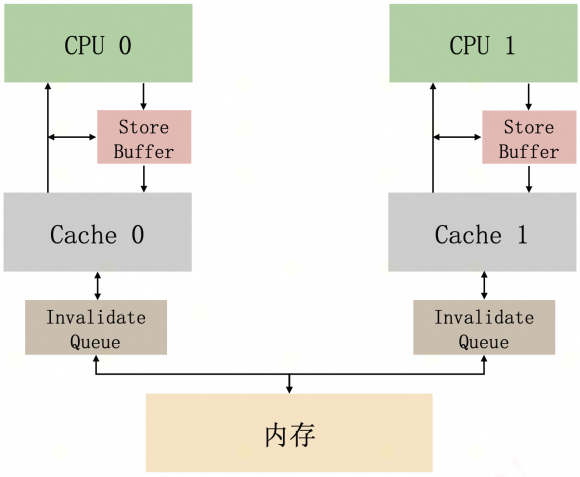
在store fowarding的机制下, store buffer成为了L1缓存的真子集, 同时维护了core内最新的数据副本.

但store buffer 存在一个问题:

> 一旦store buffer中的某个entry被标记了，那么随后的store都必须等待invalidation完成，因此不管是否cache miss，这些store都必须进入store buffer，这样就很容易塞满store buffer

问题归根结底是因为写store buffer时, 会向下传递 invalidate 信号, 通过MESI机制对其他缓存同步, 并需要等待 invalidate acknowledge 返回, 当前的CPU才能继续工作, 为了解决这个问题, 继续增加了异步机制:

invalidate queue



CPU cache接收到invalidate请求，可以先将请求入队缓冲队列，就可以回复acknowledgement消息了，后面在异步完成invalidate操作。

无论是store buffer , 还是 invalidate queue的加入, 都是在MESI机制中增添的彩蛋. 在总线锁或缓存锁的帮助下, 这些工具的加入并不会破坏MESI的原子性. 但这不代表MESI就是万能的，下面两种情况下，我们还是使用总线锁来完成相应保证一致性。

- 情况1：当操作的数据不能被缓存在处理器内部，或者操作的数据跨多个缓存行时，则处理器会字调用总线锁锁定。
- 情况2：有些处理器不支持缓存行锁定。

总结

无论是高级语言构建的业务系统, 还是电路搭建的CPU架构, 本质上都是在二维的逻辑上构筑各种功能点, 硬件范围的系统设计方案大多也可以被用于指导我们的业务系统实现

参考

https://www.cnblogs.com/flydean/p/jvm-volatile-assembly.html 从汇编角度分析Volatile

https://www.cnblogs.com/ningskyer/articles/14834303.html 指令重排序的原理

https://albk.tech/%E8%81%8A%E8%81%8ACPU%E7%9A%84LOCK%E6%8C%87%E4%BB%A4.html 聊聊CPU的LOCK指令

https://blog.csdn.net/reliveit/article/details/90038750 Intel LOCK前缀指令

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html Intel® 64 and IA-32 Architectures Software Developer Manuals

https://www.intel.cn/content/www/cn/zh/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html 英特尔® 64 位和 IA-32 架构开发人员手册：卷 3A

https://blog.csdn.net/qq_18433441/article/details/108585843 从硬件层面理解volatile