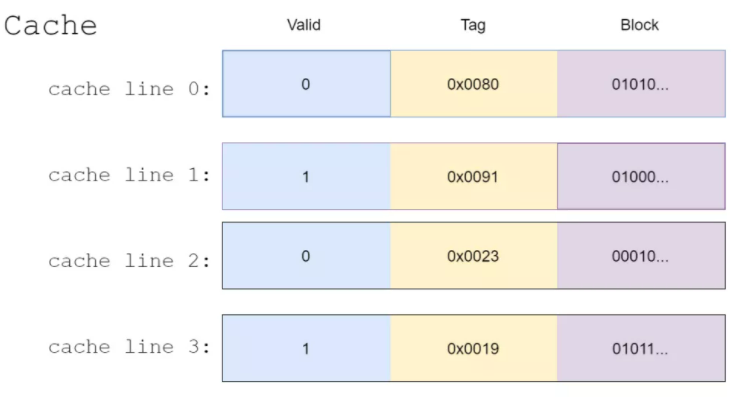
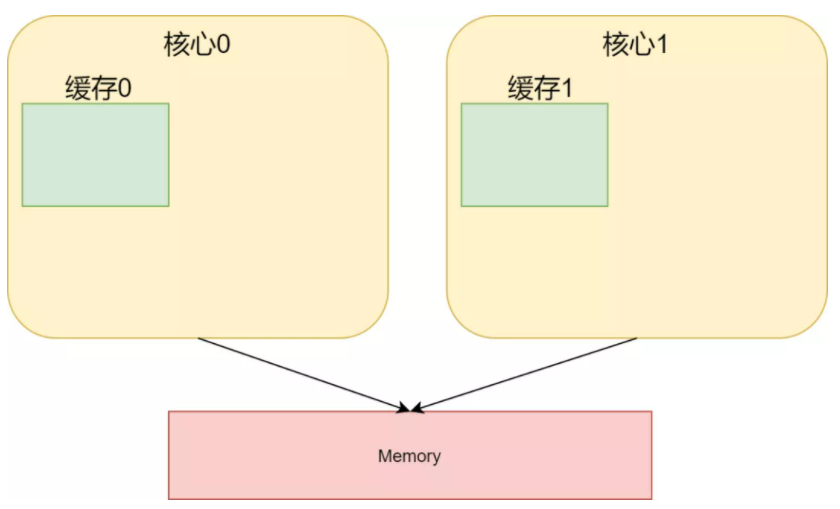
## 缓存一致性协议

现代CPU都是多核处理器，由于CPU核心读取内存数据较慢，因此设计了缓存来提升变量访问速度。缓存的结构可以如下设计：



一个缓存区可以分为多个缓存行（cache line），缓存行是和内存进行数据交换的最小单位。每个缓存行包含三个部分，其中valid用于标识数据的有效性，tag用于指示数据对应的内存地址，block用以存储数据。



但是当多个核心读取同一个变量值的时候，由于每个核心有自己单独的缓存，且读取数据的时候只读取自己的缓存，每个核心的缓存数据不一致将会导致一系列问题。

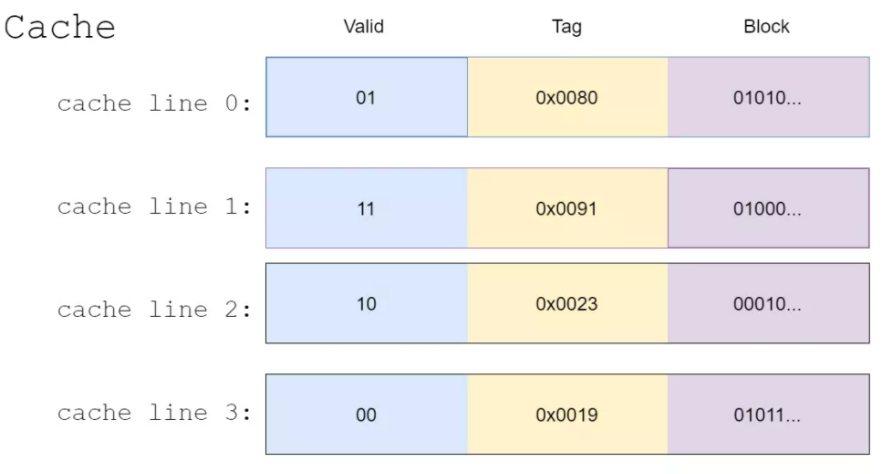
为了保证缓存一致性，业界有两种思路：

1. **写失效（write invalidate）**：当一个核心修改了一份数据，其它核心如果有这份数据，就把valid标记为无效。
2. **写更新（write update）**：当一个核心修改了一份数据，其它核心如果有这份数据，就都更新为新值，并且valid还是标记为有效。

业界有多种实现缓存一致性的协议，诸如MSI、MESI、MOSI、Synapse、Firefly Dragon Protocol等，其中最为流行的是MESI协议。

## MESI协议

MESI协议是根据写失效的思路设计的一种缓存一致性协议，为了实现这个协议，原先的缓存行修改如下：



在MESI协议中，valid改成两位，有四个状态，分别为：

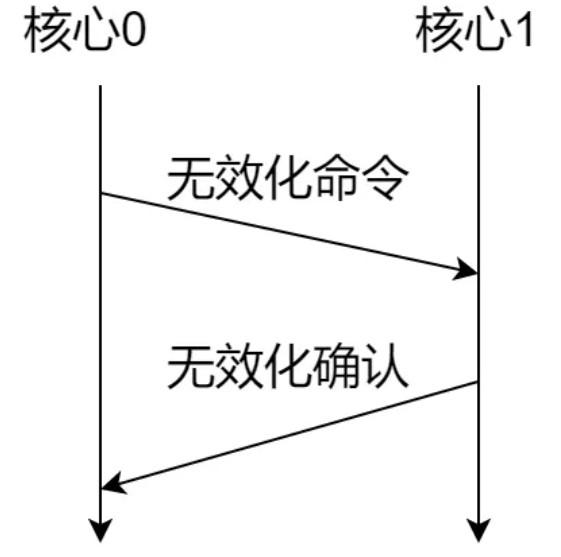
1. **M（Modified）**：表示核心的数据被修改了，缓存数据属于有效状态，但是数据只处于本核心对应的缓存，还没有将这个新数据写入内存。此时数据在各个核心缓存区只有唯一一份，不涉及缓存一致性问题。
2. **E（Exclusive）**：表示数据只存在本核心对应的缓存中，别的核心缓存没有这个数据，缓存数据属于有效状态，并且该缓存中的最新数据已经写到内存中了。此时数据在各个核心缓存区只有一份，不涉及缓存一致性问题。
3. **S（Shared）**：表示数据存于多个核心对应的缓存中，缓存数据属于有效状态，和内存一致。这种状态的值设计缓存一致性问题。
4. **I（Invalid）**：表示该核心对应的缓存数据无效。

为了保证缓存一致性，每个核心要写新数据之前，需要确保其它核心已经置同一变量数据的缓存行状态位为Invalid后，再把新数据写到自己的缓存行，并之后写到内存中。

MESI协议包含以下几个行为：

* **读（read）**：当某个核心需要某个变量的值，并且该核心对应的缓存没有这个变量时，就会发出读命令，希望别的核心缓存或者内存能给该核心最新的数据。
* **读命令反馈（read response）**：读命令反馈是对读命令的回应，包含了之前读命令请求的数据。
* **无效化（invalidate）**：无效化命令是一条广播指令，它告诉其它所有核心，缓存中某个变量已经失效了。如果变量是独占的，只存在某一个核心对应的缓存区中，那就不存在缓存一致性问题，也不用发送无效化指令。
* **无效化确认（invalidate acknowledge）**：该指令是对无效化指令的回复，收到无效化指令的核心需要将自己缓存区对应的变量状态改为Invalid，并回复无效化确认，以此保证发送无效化确认的缓存已经无效了。
* **读无效（read invalidate）**：这个命令是读命令和无效化命令的综合体，它需要接受读命令反馈和无效化确认。
* **写回（writeback）**：这个命令将核心中某个缓存行对应的变量值写回到内存中。

下面是一个应用MESI读写数据的例子。假设CPU有两个核心，核心0想要写新数据到自己的缓存。



1. 首先核心0完成新数据的创建。
2. 核心0向全体其它核心发送无效化指令，告诉其它核心其所对应的缓存区中的这条数据已经过期无效。
3. 其它核心收到广播消息后，将自己对应缓存的数据的标志位改为无效，然后给核心0发送无效化确认。
4. 收到所有其它核心的无效化确认后，核心0才能将新数据写回到它所对应的缓存结构中去。

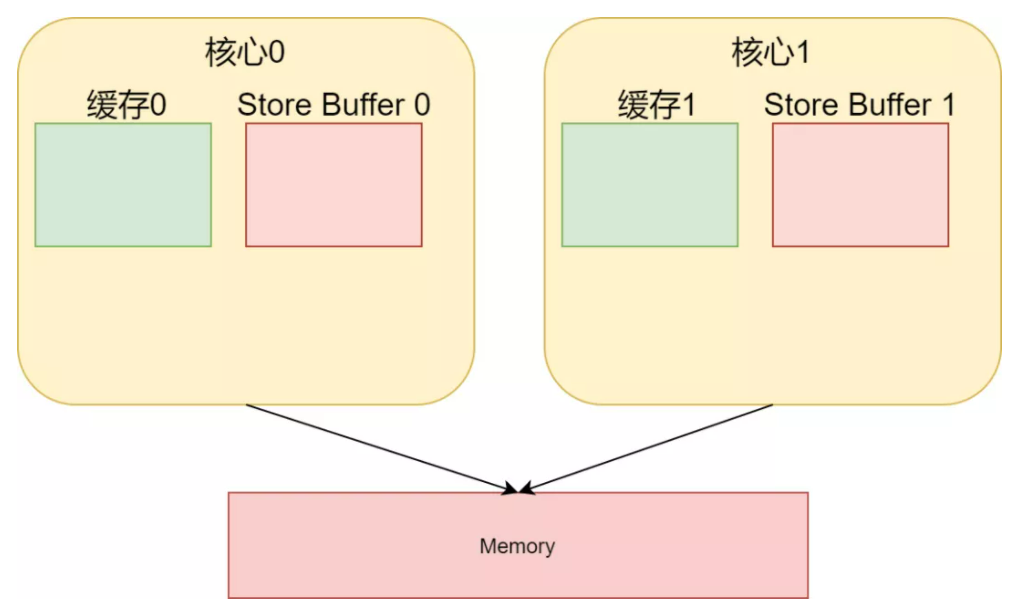
根据上图我们可以知道，影响MESI协议的时间瓶颈主要有两个：

1. 无效化指令：核心0需要通知所有其它核心。
2. 无效化确认：核心0需要收到所有其它核心的无效化确认，在收到全部确认消息之前，该核心不能做任何关于这个变量的操作。

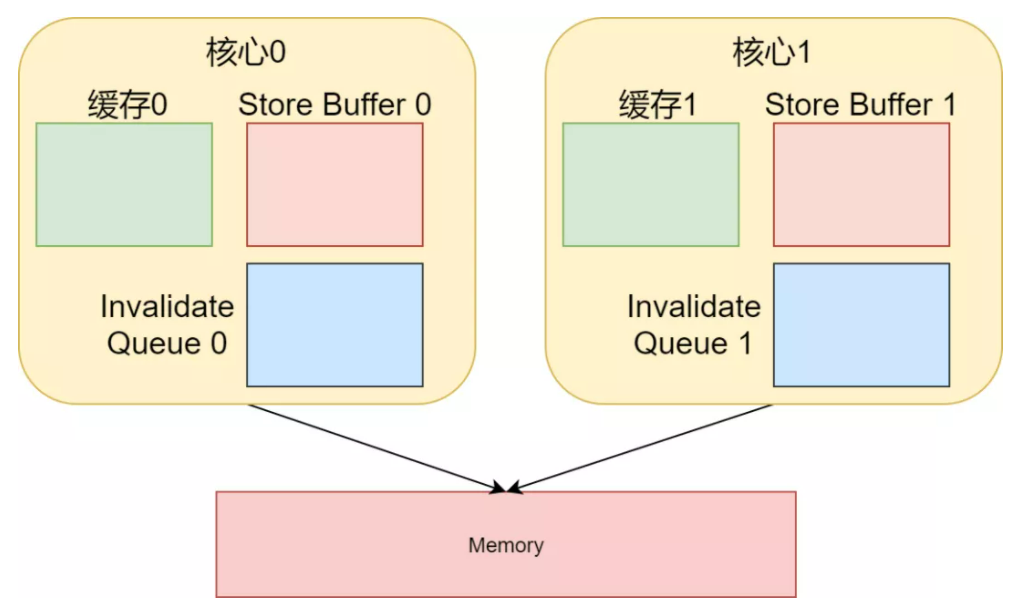
## MESI协议的优化

针对这两部分，我们可以进一步优化：

1. 针对无效化指令的加速：在缓存的基础上，引入store buffer结构。这是一个特殊的硬件存储结构。核心可以先将变量写入store buffer，然后再处理其它事情。如果后面的操作需要用到这个变量，就可以从store buffer中读取变量的值，核心读数据的顺序变成store buffer→缓存→内存。这样在任何时候核心都不用卡住，做不了关于这个变量的操作了。



1. 针对确认响应的加速：在缓存的基础上，引入invalidate queue这个结构。其它核心收到核心0的invalidate命令后，立刻给核心0发送acknowledge，并把invalidate这个操作记录到invalidate queue中，当其它操作结束时，再从invalidate queue中取命令，进行invalidate操作。所以当核心0收到确认响应时，其它核心对应的缓存行可能还没完全置为invalid状态。

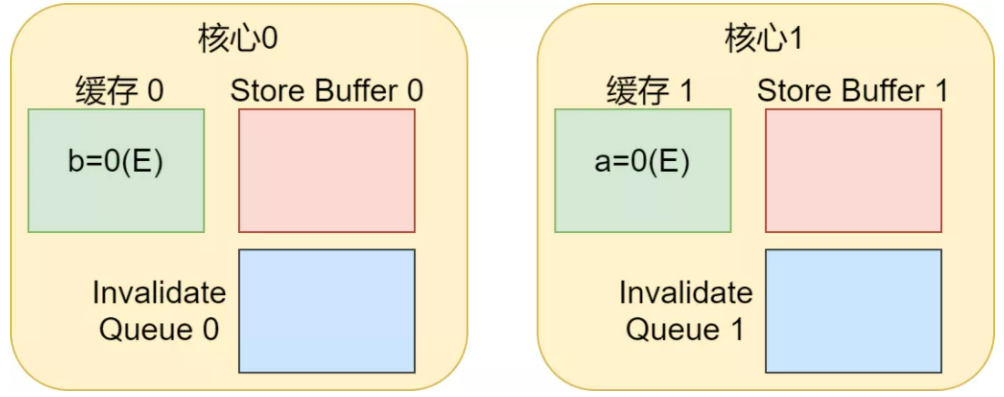


## 缓存一致性协议优化存在的问题

考虑下面的例子：

1. **public** **void** foo() {
2. a=1;
3. b=1;
4. }
5. **public** **void** bar() {
6. **while**(b==0) **continue**;
7. **assert**(a==1): "a has a wrong value!";
8. }

核心0执行foo函数，核心1执行bar函数。假设初始情况是核心1包含变量a=0，不包含变量b，而核心0包含变量b=0，不包含变量a。

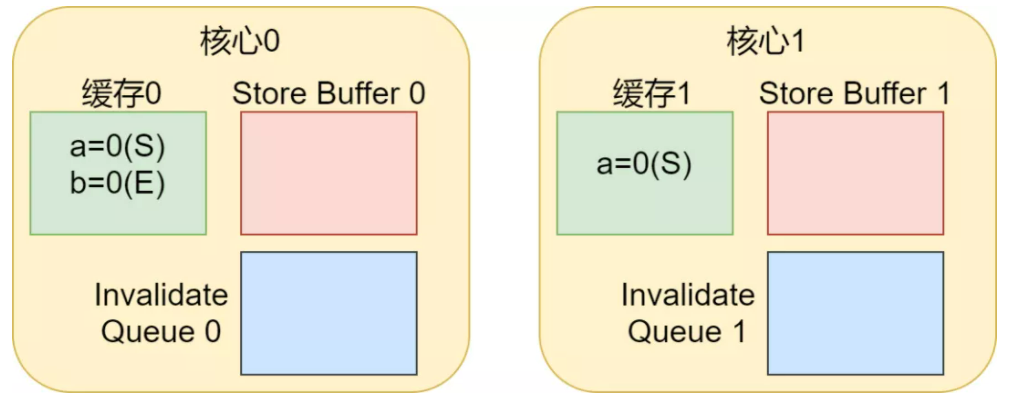


该程序的顺序可能如下：

1. 核心0执行a=1。由于核心0的缓存行不包含变量a，因此核心0会将变量a的值存在store buffer中，并且向其它核心进行读无效（read invalidate）操作，通知变量a缓存无效。
2. 核心1执行while(b==0)，由于核心1的缓存没有变量b，因此它需要发送一个读命令去找b的值。
3. 核心0执行b=1，由于核心0的缓存中已经有变量b且别的核心没有这个变量的缓存，所以它可以直接更改缓存b的值。
4. 核心0收到读命令后，将最新的b的值发送给核心1，并且将变量b的状态由E（独占）改为S（共享）。
5. 核心1收到b的值后，将其存到自己对应的缓存区中。
6. 核心1接着执行while(b==0)，因为此时b的值为1，所以跳出循环。
7. 核心1执行assert(a==1)，由于此时核心1缓存中a的值为0并且是有效的，所以断言出错。
8. 核心1终于收到了第一步核心0发送的读无效，赶紧将缓存区的变量a置为invalid，但是为时已晚。

这个例子出错的原因完全是由store buffer这个结构引起的。如果规定将store buffer中数据完全刷入到缓存，才能执行对应变量写操作的话，该错误就可以避免了。

同样的代码，我们假设初始情况是这样的：



1. 核心0执行a=1，此时由于变量a被更改了，需要给核心1发送无效化命令，并且将a的值存储在核心0的store buffer中。
2. 核心1执行while(b==0)，由于核心1对应的缓存不包含变量b，因此需要发出一个读命令。
3. 核心0执行b=1，由于是独占的，因此直接更改自己缓存的值。
4. 核心0收到读命令，将最新的b的值发送给核心1，并且将变量b的状态改为S（共享）。
5. 核心1收到核心0在第一步发的无效化命令，将这个命令存到invalidate queue中，打算之后再处理，并且给核心0发送无效化确认。
6. 核心1收到包含b值的读命令反馈，把该值存到自己缓存下。
7. 核心1收到b的值之后跳出while循环。
8. 核心1执行assert(a==1)，由于此时invalidate queue中的无效化还没执行，因此核心1会接着使用自己缓存中a=0这个缓存值，这就出现了问题。
9. 核心1开始执行invalidate queue中的命令，将a=0这个缓存值无效化，但为时已晚。

在这个例子中，出错的原因完全是由invalidate queue这个结构引发的。如果规定将invalidate queue中的命令完全处理完才能执行对应变量读操作的话，该错误也可以避免了。

## 内存屏障

1. 写屏障（strore barrier）：核心在后续变量的新值写入之前，把store buffer的所有值刷新到缓存。核心要么等到刷新完成后写入，要么就把后续变量的新值放到strore buffer中，直到store buffer的数据按顺序刷入缓存。
2. 读屏障（load barrier）：执行后需要等到invalidate queue完全应用到缓存后，后续的读操作才能继续执行，保证执行前后的读操作对其它CPU而言是顺序执行的。