Cache一致性的那些事儿 (2)--Snoop方案

2.Snooping-based解决方案

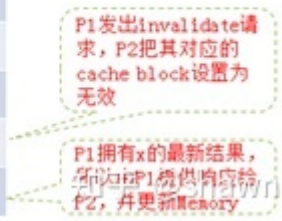
Cache需要对共享总线进行侦测，如果侦测到总线上的操作与自己cache中的某个cache block相符合(tag一致)，则采取某种动作（具体动作由cache一致性协议定义，比如MSI），这种系统需要支持广播功能的总线，此外这种方案比较适用于小规模的系统；

2.1 Write Invalid和Write update

在Snooping方案中，当一个处理器需要更新某个数据时，有两种处理方式，我们称之为Write-invalidate和Write-update。

n Write-invalidate: 一个处理器需要更新某个数据时，其先往总线上发送一条Invalidate请求，系统中其它处理器上的cache控制器监控到这条消息，把自己的状态设置为invalid。如果这些invalidated cache block在下一次需要访问该数据时，则会重新进行一个总线的read miss操作。

时间	处理器操作	总线行为	P1中cache	P2中cache	内存状态
t0					X=0
t1	P1 read x	Read miss x	X=0		X=0
t2	P2 read x	Read miss x	X=0	X=0	X=0
t3	P1 write x=1	Invalidate x	X=1		X=0
t4	P2 read x	Read miss x	X=1	X=1	X=1



n Write-update（或者叫write broadcast）：当一个处理器更新某个数据时，其不往总线上发送invalidate消息，而是往总线上发送一条update A消息，直接告知该变量的最新值，其它处理器上的cache侦听到这个操作后更新本地的变量为这个广播的最新值。其它处理器在下一次使用该变量时，直接cache 命中，可以直接使用最新数据，而不产生read miss操作。

时间	处理器操作	总线行为	P1中cache	P2中cache	内存状态
t0					X=0
t1	P1 read x	Read miss x	X=0		X=0
t2	P2 read x	Read miss x	X=0	X=0	X=0
t3	P1 write x=1	Update x	X=1	X=1	X=1
t4	P2 read x		X=1	X=1	X=1

乍一看，好像write update的方式性能更好，但真的是这样吗？让我们对这两种更新方式的优缺点做一个简单的对比：

使用场景	Write invalidate	Write update
对一个地址进行多次写操作： Write A, Write A, Write A	总线上只需要发送一条 invalidate 消息	需要发送多条 write update 消息
对同一个 cache block 中的不同的地址进行写操作： Write A[0], write A[1], write A[2]	总线上只需要发送一条 invalidate 消息	需要发送多条 write update 消息
一个 core 上的写于另外一个 core 的读操作的时延	时延比较大	时延小

从上表中我们可以看出，write update方式需要消耗比write invalidate更大的带宽，所以通常实现中我们都采用write invalidate方式，我们后面章节谈论的一致性协议也都是基于write invalidate方式的。

2.1 MSI协议介绍

实现内存一致性的协议可以细分为很多种，但最基本的还是MSI协议，MSI代表了 cache block的三种不同状态 I,S,M，分别是invalid, Shared和Modified。任何时刻Cache block必处于这三种状态之中的一种状态。

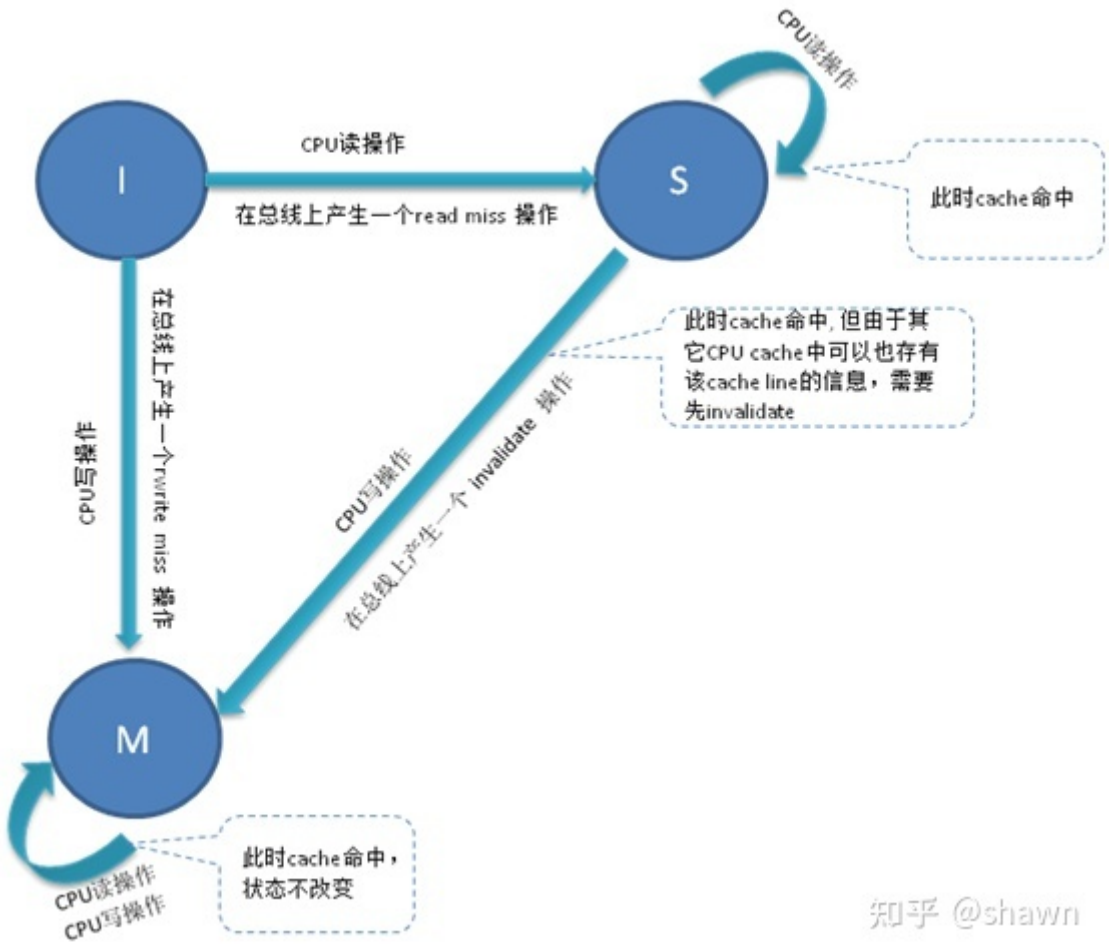
n 状态 I(Invalid)：该cache block在当前cache中不存在或者被总线上的invalidate操作设置为无效，处于该状态的cache block需要从Memory或者其它cache中获取，在访问该cache block时，cache控制器需要往总线上产生一个read miss或者write miss操作。

n 状态S(Shared): 该cache block的内容没有被修改并且处于只读状态，该cache block存在于至少一处cache中和memory中，该状态下CPU能够直接读取该cache block数据而无需与其它cache进行通信，总线上没有操作。

n 状态M(Modified):该状态只能存在一处cache中，该状态下cache block，CPU能够直接读写而不需要知会其它CPU上的Cache.该状态的cache block负责为其它cache节点提供最新的数据，同时也负责把最新数据写回到memory中。

2.1.1 CPU视角的MSI状态机

从CPU的视角（由于本地CPU的操作导致的本地cache block状态变化）来看，MSI协议的状态机可以表示如下：



我们对该状态机做一些简单的说明：

n 处于I状态的cache block:

CPU中cache block的状态总是从I状态开始的，因为一开始的时候cache中并没有缓存某个内存地址的数据。

² 当CPU需要读取某个内存数据时，此时发生cache miss，于是cache控制器对总线发送一个read miss操作。Memory或者其它远端cache控制器(处于M状态)对该请求进行响应，本地cache拿到数据，进入shared状态。

² 当CPU发起一个写操作时，Cache控制器往总线上发送一条write miss命令，然后进入Modified状态。（问题：CPU为什么可以对处于I状态的cache block 发起写操作？）

n 处于S状态的cache block:

² 如果CPU发起读操作，由于直接cache命中，状态不发生改变；

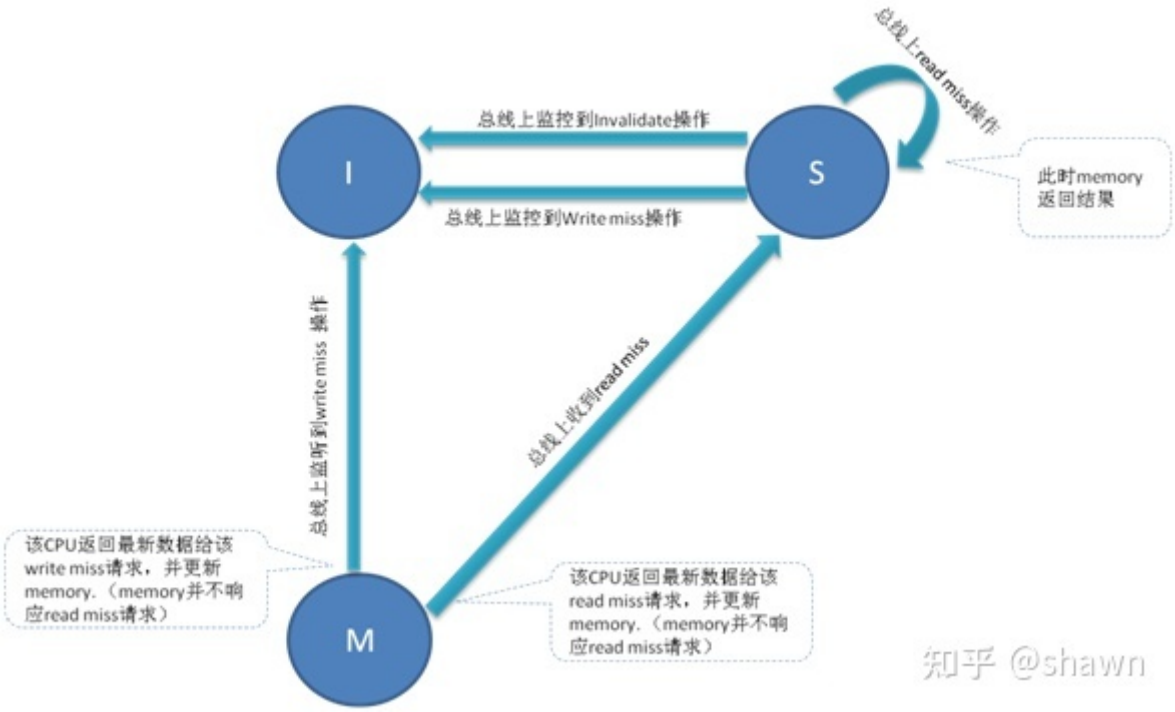
² 如果CPU发起一个写操作，则cache控制器往共享总线上放置一个invalidate命令，该cache block切换到Modified状态。

n 处于M状态的cache block:

无论CPU发起读还是写操作，由于直接cache命中，状态都不发生改变；

2.1.2总线视角的MSI状态机

从总线的视角（由于远端CPU在总线上的操作导致的本地cache block的状态变化）来看，MSI的状态状态机可以表示如下：



我们对该状态机做一些简单的说明：

n 处于I状态的cache block:

表明本CPU对于该cache block数据不关心，所以也就不关心总线上的任何消息，也不会对总线上的任何消息产生动作。

n 处于S状态的cache block:

Ø 如果侦测到总线上发生了invalidate命令或者write miss命令，则表明其它CPU需要对该cache block进行修改，所以把本地cache block设置为无效，进入I状态。

Ø 如果侦测到总线上发生了read miss操作，则memory会提供有效响应，而本cache block不做动作，维持在S状态。

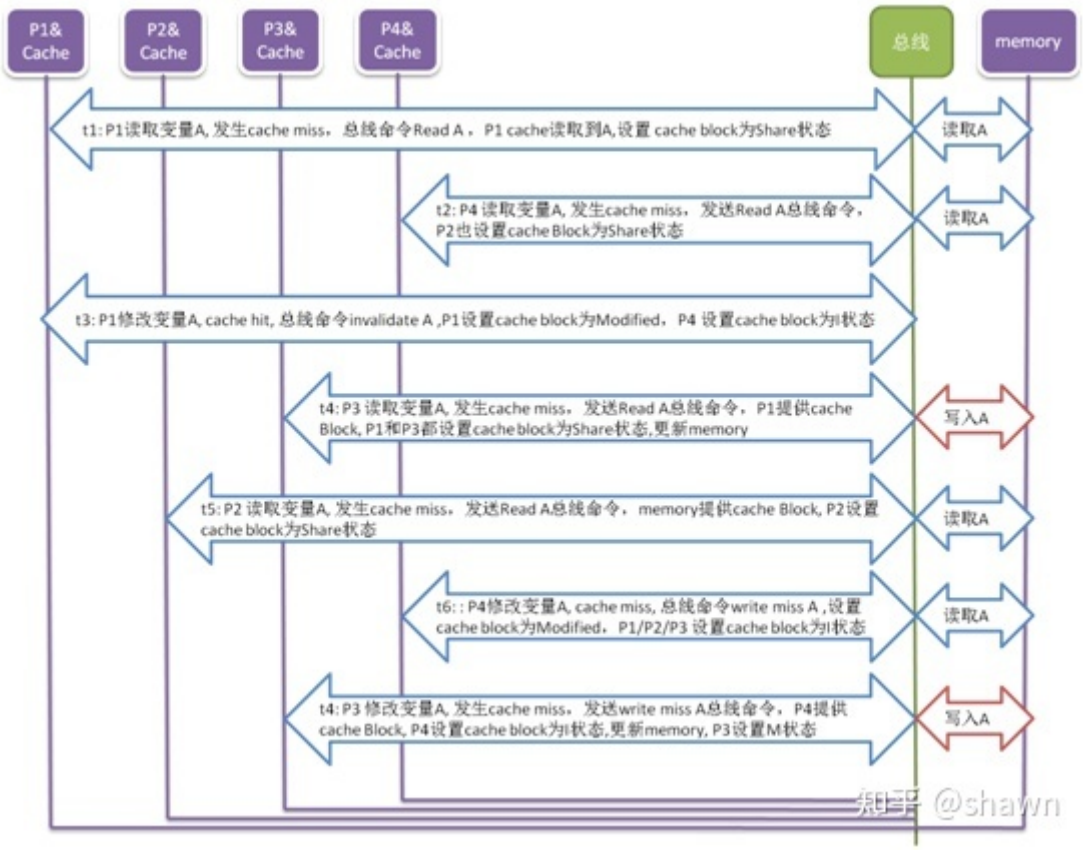
n 处于M状态的cache block:

Ø 如果侦测到总线上发生read miss操作，则本cache需要为请求方提供该cache block,同时会利用这个机会更新memory。此时该cache block进入共享状态S。

Ø 如果侦测到总线上发生write miss操作，则本cache需要先更新自己的最新数据给请求方提方,同时会利用这个机会更新memory。但此时该cache block进入无效状态I。

1.2 MSI协议怎么解决Cache一致性

MSI协议是如何解决Cache的一致性问题的呢？如何理解上面的两种不同视角的MSI状态转换图，我们来看一些实际的cache操作过程。



² t1: P1读取变量A, 发生cache miss，Cache控制器发送Read A命令到总线，P1读取A，并把该cache block设置为Share状态（表示只是读取，没有修改）。

² t2: P4 读取变量A, 也发生cache miss，Cache控制器发送Read A命令到总线，P4读取A，并把该cache Block设置为Share状态。

² t3：P1决定要修改变量A的值，为了向总线上的其它CPU通知该修改，P1上的cache控制器向总线发送invalidate A命令，其它CPU监控到共享总线上的invalidate命令, P4的cache控制器通过对比自己的Cache状态，从而把A变量对应的状态由share修改为invalid状态, P1将状态由share修改为Modified. （拥有M状态的CPU可以对变量做read, write操作而不修改状态， 所有对A变量的load操作和write操作都可以在本地cache中完成）

² t4: 此时如果P3需要读取A, 也发生cache miss, P3往总线上发送read A命令，P1监控到P3发出的read A操作，发现自己拥有A的Modify状态，也就表面自己是A的有效值的唯一拥有者。于是P1为P3提供A的cache block，同时更新memory中A值到最新状态。并且更改自己的状态从modify到shared。P3收到最新的A值，也把其状态修改为shared. (此时P1, P3同时拥有相同的A)

² t5: P2此时也读取A的值，发生cache miss, 于是P2往shared bus上发送read A命令，此时P1和P3发现自己的状态为shared，对该请求不做响应，而由main memory为该请求提供cache block。P2也进入shared状态。

² t6: P4此时想修改A的值，由于其cache中的A的状态为invalid, 发生cache miss. 于是P1往shared bus上发送一个write miss A命令。P1, P2, P3检测到该write miss A命令，对比自己的cache状态，并把自己A的状

态修改为Invalid. 此时Memory响应P4的write miss A命令提供最新的cache block；P4的状态从invalid修改为modified。

² t7: 假设此时P3也想修改A的值，但由于其cache状态是invalid, 所以P3需要往总线上写一个write miss A命令。其它cache监测到该命令后需要做一个自我检查, P4由于当前拥有M状态，所以响应该请求，为P3提供A的最新cache block，并且更新主memory，同时修改自己的状态为invalid. (留一个思考题：此处P3只是修改A的值，并不需要读取A,为什么P4还需要提供其当前的最新cache block？)。

我把上述操作过程整理为一张表：

时刻	CPU 请求	Cache 命中	总线命令	命令响应者	P1 状态	P2 状态	P3 状态	P4 状态
t0					I	I	I	I
t1	P1: read A	Miss	Read A	Memory	S	I	I	I
t2	P4: read A	Miss	Read A	Memory	S	I	I	S
t3	P1: write A	hit	Invalidate A		M	I	I	I
t4	P3: read A	miss	Read A	P1 Cache	S	I	S	I
t5	P2: read A	miss	Read A	Memory	S	S	S	I
t6	P4: write A	miss	Write miss A	Memory	I	I	I	M
t7	P3: write A	miss	Write miss A	P4 Cache	I	I	M	I

（注：MSI一致性协议不仅可以用在snoop的总线结构上，还可以应用在后面的directory-base方案中）

1.3 MSI协议的增强版本

本节对MSI协议的几个增加版本（或者叫变种）做一个简单的介绍，增强版本中的完整状态机处理过程，可以参考专门的资料，就不做太多的细节描述了。

2.3.1 MESI协议

让我们考虑一种内存访问场景：

时刻	CPU1指令	CPU2指令	Bus操作	P1中A的状态	P2中B的状态
t0				I	I
t1	Read A		Read miss A	S	I
t2	Write A		Invalidate A	M	M
t3		Read B	Read miss B	M	S
t4		Write B	Invalidate B	M	M

系统中A和B只存在于一处cache block中，在其它CPU上并不存在A和B的另外一份拷贝, 但t2时刻与t4时刻的invalidate消息真的有必要吗？所以我们可以对原来的MSI协议进行一个增强，引入一个新状态E(Exclusive), 从而MSI协议演进为MESI协议。

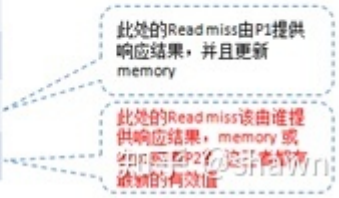
n 处于独占状态（E）的cache block只出现在一个处理器节点上，并且是干净的（表示cache中的数据与memory的数据一致）。处于E状态的cache block, 修改时候不需要进行总线上的invalidate操作。

n 处于E状态的Cache block, 如果侦听到总线上有read miss操作，则状态从E改变为S，也即从独占状态降级为共享状态，此时由memory或者本cache控制器提供对read miss的响应。

2.3.2 MESIF协议

Intel的i7处理器使用一个MESI的变种协议MESIF来处理cache一致性问题。通过引入一个新状态F(Forward)来指定哪个CPU core应该对read miss和write miss做出响应。当系统中有多处cache block都处于S状态时，此时如果总线上发生了read miss操作，该由哪个cache做出响应呢？让我们来看一个具体的例子：

时间系列	处理器P1	处理器P2	处理器P3	总线请求	C1状态	C2状态	C3状态
t0					I	I	I
t1	Read A			Read miss	S	I	I
t2	Write A			Invalidate	M	I	I
t3		Read A		Read miss	S	S	I
t4			Read A	Read miss	S	S	S



该例子中的memory, P1处理器和P2处理器都拥有资源A，如果此时总线上出现了read miss A操作，哪该由谁来响应该read请求并给出响应的结果呢？由于cache的读取速度比memory要快很多，所以由P1或者P2来提供响应性能会更好一些。但P1和P2都处于同等位置，无法确定该由二者之一的谁来提供响应。

MESIF协议通过把多个处于S状态的cache block中的一个设置为F状态，并且规定由F状态的cache对read miss做出响应。这就解决了上述场景的应答的冗余性（多个处于S状态的cache都可以响应）和低效性（memory也可以响应）问题。

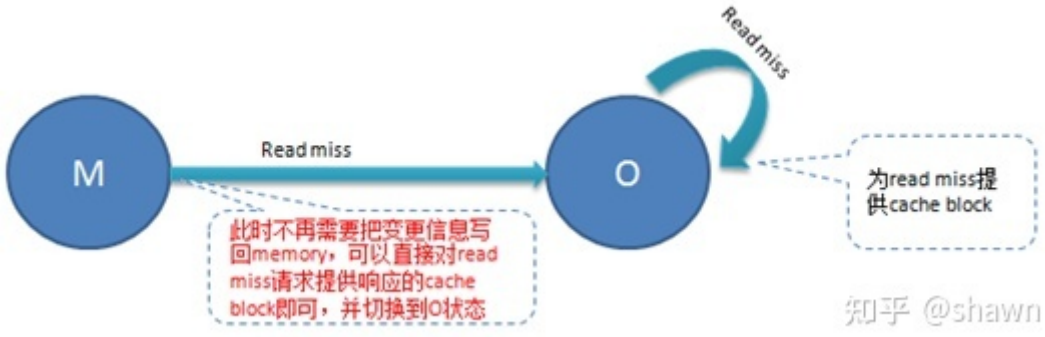
注：处于F状态的cache block必须满足两个条件：

（1）cache block是干净（clean）的，也就是与memory中的数据时一致的

（2）该cache block处于shared 状态；

2.3.2 MOESI协议

在MESIF协议中，F状态的cache block必须是shared和clean的，那么对于处于modified和shared两种混合状态下的cache block还有没有更进一步的总线操作优化空间呢？答案是有的，那就是MOESI协议。 通过对MESI协议进行扩展，引入一个新状态O(owned), MESI协议就演进为MOESI，由于O状态的引入，对于原来MSI和MESI协议中的M状态下的动作和状态变化需要进行如下调整：



O状态的引入解决两个问题：

（1） M状态的cache block在侦听到总线上的read miss后，不需要立即将最新的数据写回到主memory, 而是直接提供cache block给请求方，同时自己进入到O状态；

（2） 处于O状态的cache block负责响应总线上的read miss操作，取到与MESIF协议中F状态的功能。

注：AMD Opteron采用MOESI协议来实现cache 一致性。