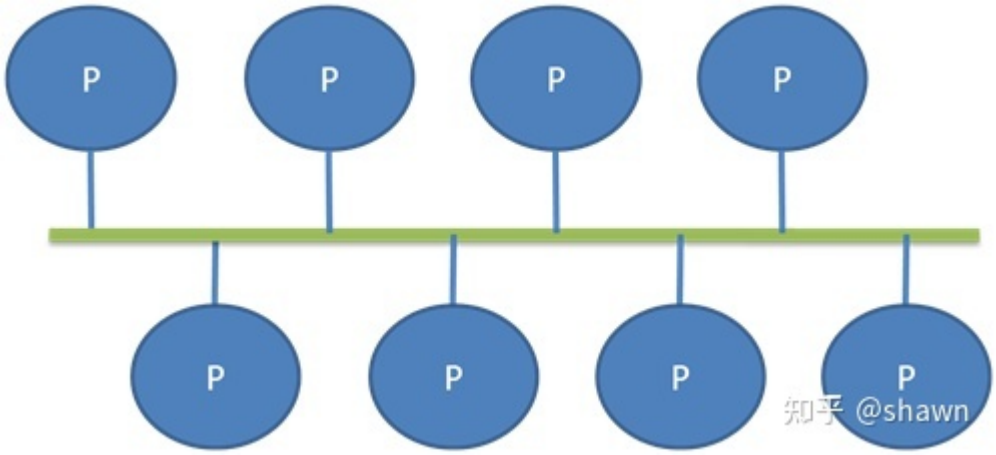
zhuanlan.zhihu.com /p/419722803

Cache一致性的那些事儿 (3)--Directory方案

3. Directory-based 解决方案

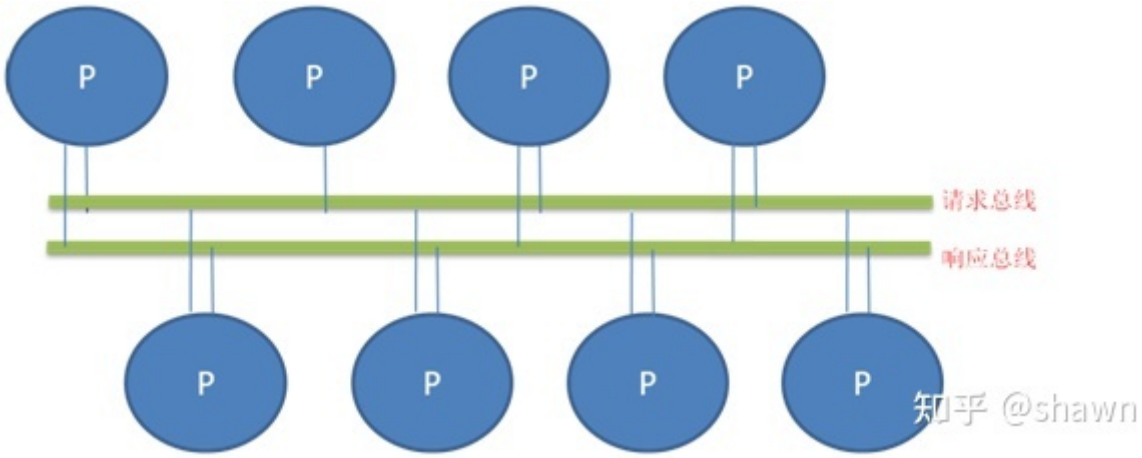
在一个大型的计算系统中，当其处理器的数目比较多时，



采用snooping方案存在两个方面的弊端：

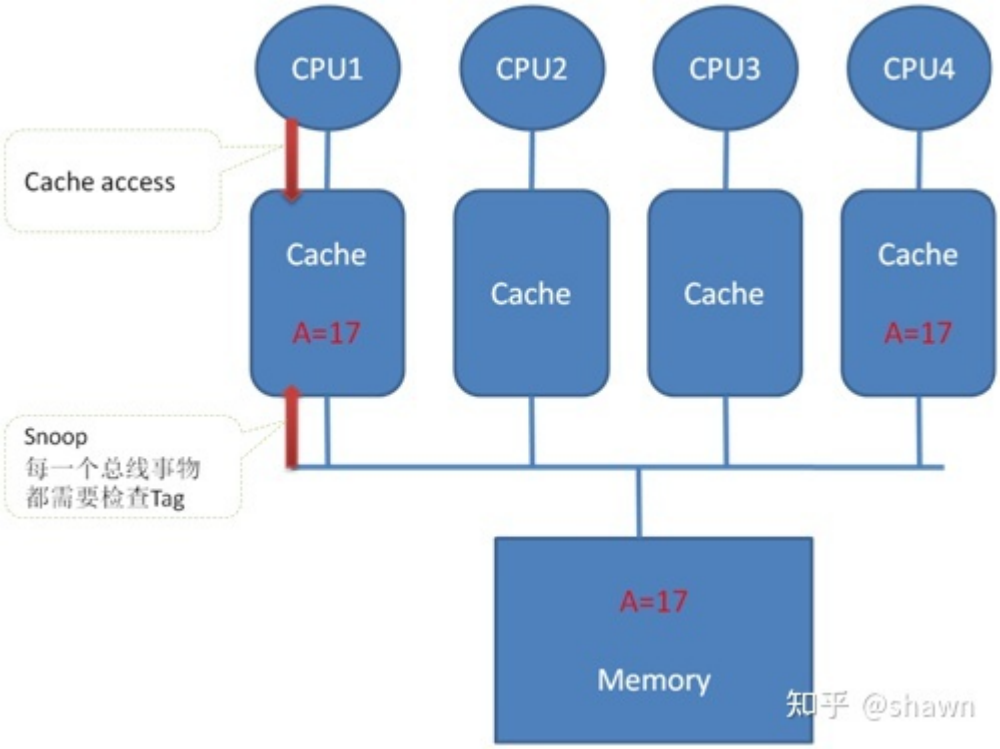
- 由于所有cache的一致性操作全部需要通过shared bus来完成，shared bus的带宽就会成为整个系统的瓶颈
- 由于处理器的增多，连接这些处理器的共享总线物理上肯定也会变长，这也会导致时延的增加。

我们当然也可以采用多总线、crossbar或者采用一种小型的点对点网络来取代上面的单一共享总线(shared bus), 但这些替换都会让cache的一致性管理变得非常复杂。



双总线结构

Snoop方案的另外一个缺点是CPU对cache访问与cache 控制器对总线侦测的相互干扰问题。



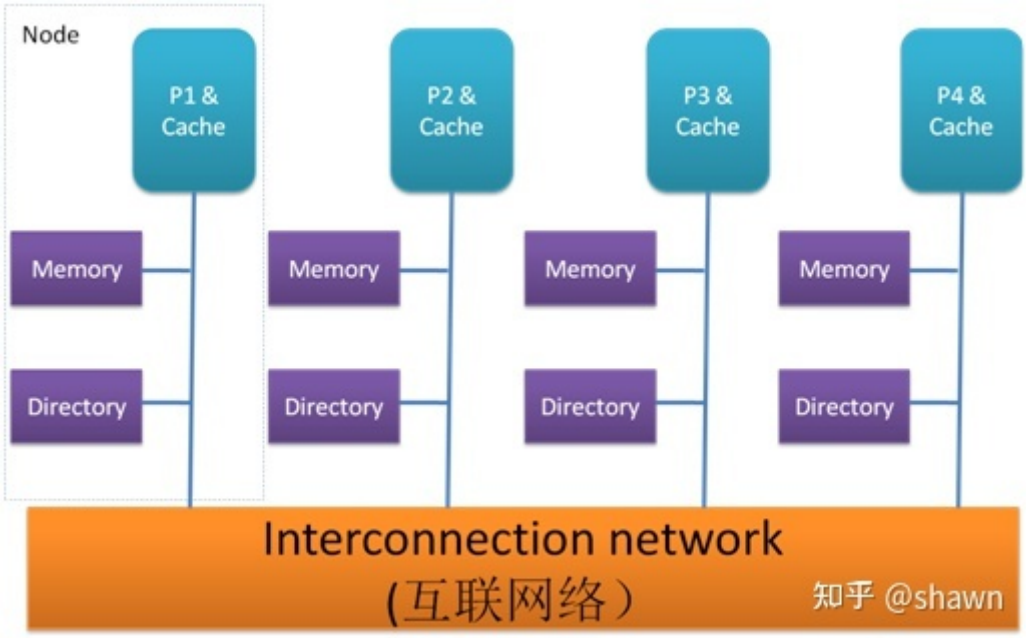
由于cpu访问cache过程中需要对cache block的tag进行比对，此外cache控制器自身在侦听到总线消息时候，也需要比对Cache block中的tag值，这两者就会产生干扰。我们可以采用如下的一些方法来降低这种风险（增加了硬件实现的复杂性，增加了成本，也无法消除风险）：

- 比如采用双tag机制，一个tag用于CPU访问cache，另外一个tag用于总线snooping本对tag。
- 采用L1和L2两级cache也能缓解这种干扰，这两级cache采用inclusion的方式组织，cpu访问L1 cache, 总线侦听采用L2 cache。【注：当然L1和L2两级cache的设计不是为了解决这个问题的，只是说它对缓解这个风险有帮助】

所以上述的Snoop方案一般对于4-8 核的系统比较可行，方案的可扩展性比较差，我们需要一种分布式系统来解决对更多核的支持问题。

3.1 分布式内存系统

分布式的内存方案，如下图所示：



在这个分布式的系统中，每个处理器都拥有一个L1/L2私有caches, 此外每个处理器还拥有一个本地memory （我们把这个小系统称为一个节点Node）。处理器通过内存地址来决定要访问的资源是在本节点memory还是在其它节点；如果要访问的资源在其它节点，则通过一个互联网络、以点对点的方式来向远程节点（remote node）访问该信息。互联网络(interconnection network)可以看作一个全相连的mesh结构（不再是一个shared bus）。

在分布式内存系统中，我们一般采用Directory-based的cache一致性方案。也就是是把cache block的共享信息存放在一个固定的地方，这个地方我们称之为directory。Directory记录系统中每一个cache block的共享信息,记录的内容可以如下（假设系统中N个节点）：

一个cache控制器想要发起一个一致性访问请求，则它把该请求通过点到点的方式直接发送到该资源的归属节点(Home Node), Home node的directory根据其目前的状态来采取动作（前面介绍的MSI,MESI, MOESI等一致性协议中定义的状态变化和使用原则继续适用，只是底层消息传递的方式变化了）。

在本章下面的描述中，我们都假设上图中每个节点的本地memory大小为1G, 其内存地址范围如下：

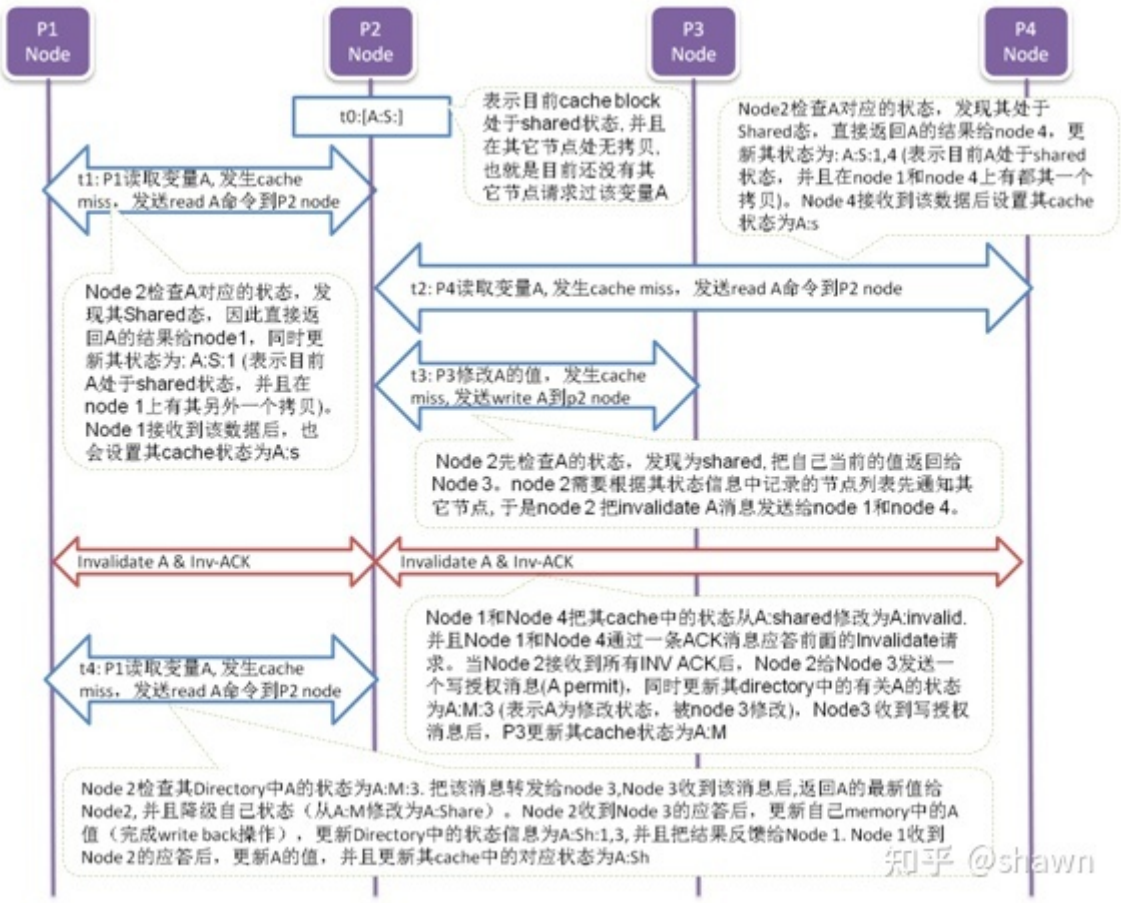
- node1的memory地址区间为0G-1G,
- node2的memory地址区间为1G-2G

- node3的memory地址区间为2G-3G
- node4的memory地址区间为3G-4G

3.2 Directory操作流程

1. 假设P1处理器访问一个变量A, 其地址在1.5G的地方。P1查询本地的directory发现该变量不在本地的memory，而是在node2的地址范围，于是node1通过network interface发送一个点对点的消息到node2。
2. Node2通过其network interface接收到该消息，并从其memory中获取相关数据以一个cache block返回给node1. 我们从这个流程可以看出，该处理过程只有Node1和node2介入，因为没有广播消息，所以不相关的node3和node4对这个操作毫无感知。在以前的snooping方案中，由于在总线上采用了广播消息，每个处理器都需要对总线上发生的事物(transaction)进行处理。
3. 此时如果P3也需要访问该变量A, 则需要另外一种机制，比如通过Node2作为中间代理，在节点node1与node3之间进行cache的一致性维护，这就是directory-based cache一致性的基本原理。我们把该流程中的node 2称为A的home node, node 3称为请求节点（又称为local node），如果node 1和Node 4中也有A的一份拷贝，则node 1和Node 4称为remote nodes。

我们来看一个具体的访问流程来加强对directory-based cache一致性的理解；我们继续采用前面的假设：A的地址为1.5G，位于node2的memory中。



- t0（初始状态）: 在Node2的directory中会创建了一个对变量A的cache block的状态信息，记作：A:S: 【表示目前cache block处于shared状态, 并且在其它节点处无拷贝（share list为空）,也就是目前还没有其它节点请求过该变量A】

- t1: P1对变量A做了一个read操作（read A），P1查找其cache, 发生一次cache miss, 于是访问其本地的Directory，进而通过互联网络以点对点的把read A消息请求发送到Node2。Node 2检查A对应的状态，发现其Shared态，因此直接返回A的结果给node1，同时更新其状态为: A:S:1 (表示目前A处于shared状态，并且在node 1上有其另外一个拷贝)。Node 1接收到该数据后，也会设置其cache状态为A:s （该信息直接在Node 1的cache中，不在Directory中）
- t2: P4也发起对变量A的一个读操作(read A)，同样地先检查自己的cache，发生cache miss, 进而也通过互联网络以点对点的把read A消息请求发送到Node2。Node2再一次检查A对应的状态，发现其还是处于Shared态，因此直接返回A的结果给node 4，同时更新其状态为: A:S:1,4 (表示目前A处于shared状态，并且在node 1和node 4上各有一个拷贝)。Node 4接收到该数据后，也会设置其cache状态为A:s （该信息在Node 4的cache中，不在Directory中）
- t3: Node 3想修改A的值，该请求消息(write A)通过互联网络发送到节点node 2。Node 2先检查A的状态，发现为shared, 于是先把自己当前的值返回给Node 3。但是在Node 2给予Node 3写权限之前，node 2需要根据其状态信息中的节点记录先通知其它节点（此处是node 1和node 4）把该cache block标记为invalid。于是node 2 通过点对点的方式把invalidate A消息发送给node 1和node 4。Node 1和Node 4把其cache中的状态从A:shared修改为A:invalid. 并且Node 1和Node 4通过一条ACK消息应答前面的Invalidate请求。当Node 2接收到所有INV-ACK后，Node 2给Node 3发送一个写授权消息(A permit)，同时更新其directory中的有关A的状态为A:M:3 (表示A为修改状态，被node 3修改)，Node3 收到写授权消息后，P3更新其cache状态为A:M。
- t4: 此时P1想读取A的值，由于其cache中的A为invalid状态，于是通过互联网络把读请求（read A）发送给A的home node (Node 2). Node 2检查其Directory中A的状态为A:M:3. 所以node 2把该消息转发给node 3,Node 3收到该消息后，意识到其它节点需要使用A, 于是返回A的cache block给Node2, 并且降级自己状态（从A:M修改为A:Share）。Node 2收到Node 3的应答后，更新自己memory中的A值（完成write back操作），更新Directory中的状态信息为A:Sh:1,3, 并且把结果转发给Node 1，Node 1收到Node 2的应答后，更新A的值，并且更新其cache中的对应状态为A:Sh

我们也以图表的方式来总结上面过程中的状态变化和消息交互：

时刻	CPU 请求	Cache 命中	消息内容	Directory 状态	C1 状态	C2 状态	C3 状态	C4 状态
t0					I	I	I	I
t1	P1: read A	Miss	Read-req to Dir & Dir responds	A:S:1	S	I	I	I
t2	P4: read A	Miss	Read-req to Dir & Dir responds	A:S:1,4	S	I	I	S
t3	P3: write A	Miss	write-req to Dir Dir sent Invalidate to P1 and P4 P1 and P4 send INV-ACK to Dir Dir grants permission to P3	A:M:3	I	I	M	I
t4	P1 read A	miss	Read-req to Dir Dir forware read-req to P3 P3 send data to Dir Dir send data to P1	A:sh:1:3	S	I	S	I

注：

(1) C1: P1 cache, C2: P2 cache, C3: P3 cache, C4:P4 cache

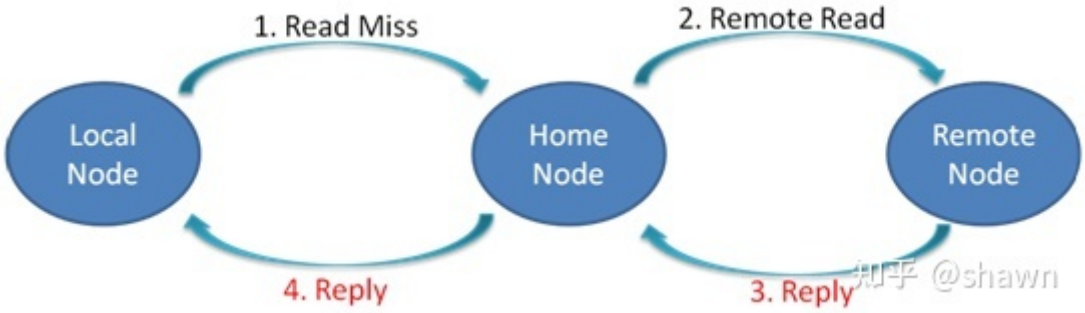
(2) State I: invalid, S:Shared, M:Modify

3.3 Directory优化

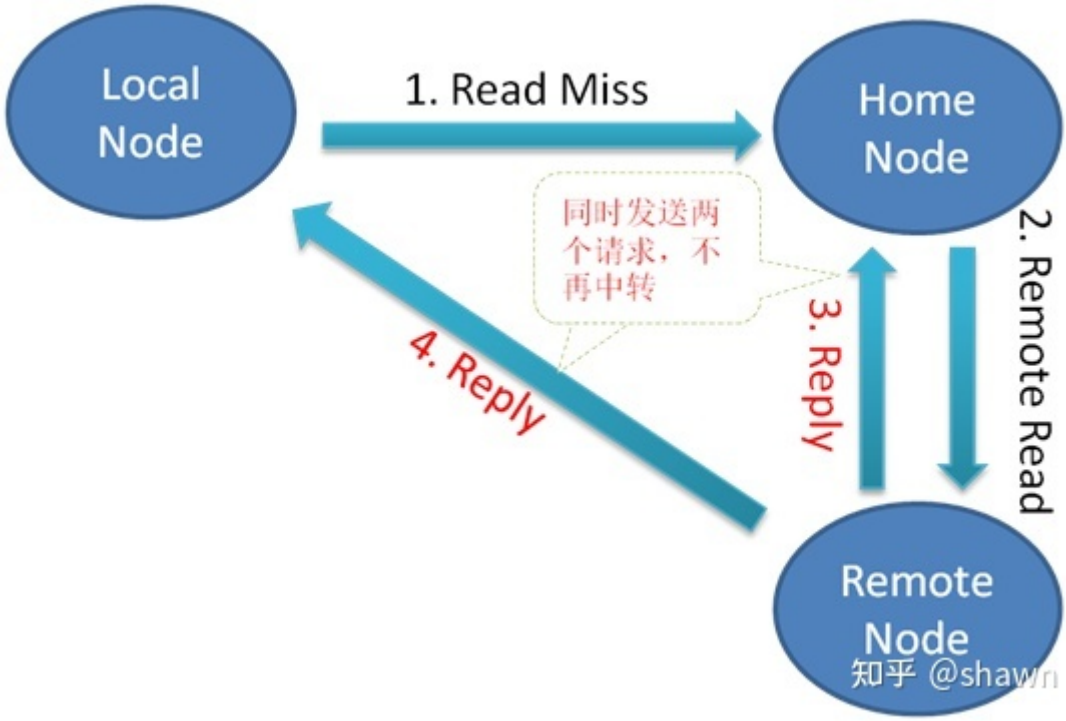
上面我们介绍了一种基于MSI三状态的directory解决方案的操作流程，为了改善性能，我们还可以同样引入E(exclusive)状态和O(owner)状态。其详细的状态转换流程请参考相关书籍，我个人推荐:《A primer on memory consistency and cache coherence》。此外上述的directory方案还可以进一步优化：（1）降低时延 （2）减少内存额外开销。

3.3.1 时延优化

在前面的描述中，我们介绍了一个节点(local node)如果想获得某个资源的信息，先需要把请求发送到该资源的归属节点(home node), 归属节点根据其Directory中记录的状态情况，如果该资源目前被其它远端节点(remote node)拥有，处于修改状态，则归属节点转发该访问请求到远端节点。Remote node在收到请求后，返回应答给归属节点，归属节点更新资源cache block并把应答转发给请求节点。



从请求发出到应答，经过了多次中转，时延肯定受影响（4倍的网络传输时延），为了降低该时延，我们可以把应答的cache block直接发送给Local节点而不经由home节点中转，当然为了更新Home node处的cache block, 所以还需要给Home Node也发送一份应答。这两个应答消息可以利用网络传输能力同时分别发送。



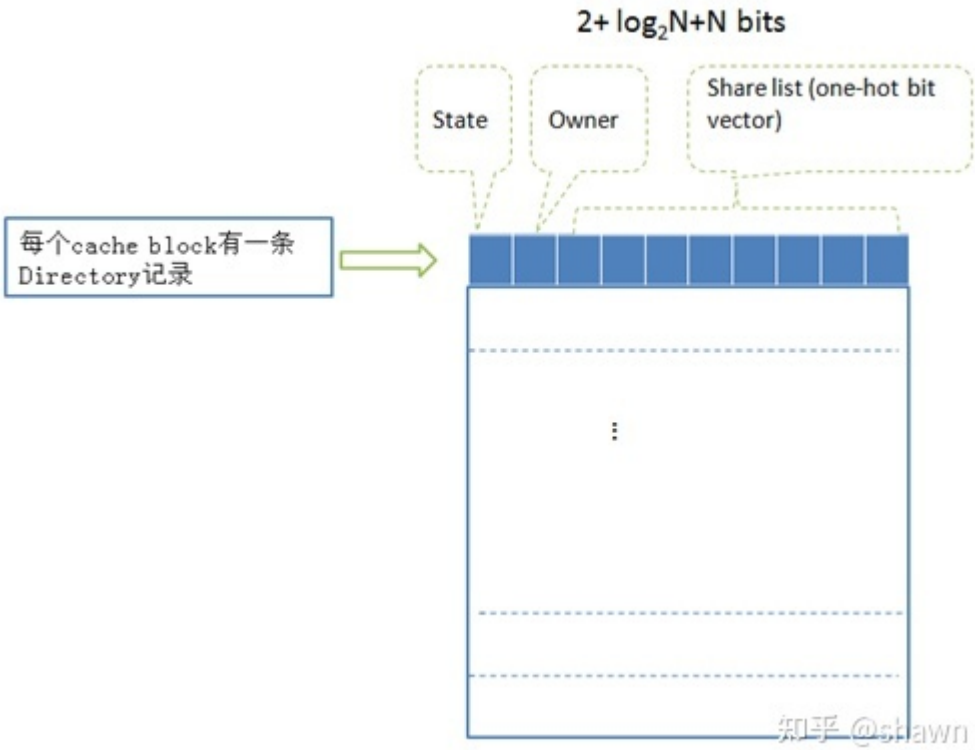
3.3.2 存储空间优化

为了实现cache的一致性，我们需要增加的额外的存储空间来存放Directory记录，这个空间是多大呢？

- 假设Directory记录总数为M， 由于节点中每个memory block都需要对应一个directory记录，所以M=memory_size/cache_block
- 假设系统中有N个节点，则单条Directory记录占据的空间大小为：2+ log2N+N bits 【注：状态2-bit，拥有者log2N bits和共享节点列表独热位矢量N bits】，简单起见，我们就取N bits。

于是一个节点中的directory占用的总空间大小为：N*M。

Directory单条记录的大小随着节点数目的增多而增多，我们可以计算一下单条记录的额外空间占比(假设cache block的大小为64字节)：



（1） 当节点数为8时：

Directory空间占比 = （2+ log28+8）/（64×8）= 13/512 = 0.025

占比为2.5%，基本可以忽略。

（2） 当节点数为64时：

Directory空间占比 = （2+ log264+64）/（64×8）=72/512 = 0.140

占比为14%， 这个额外开销的比例就比较高了。

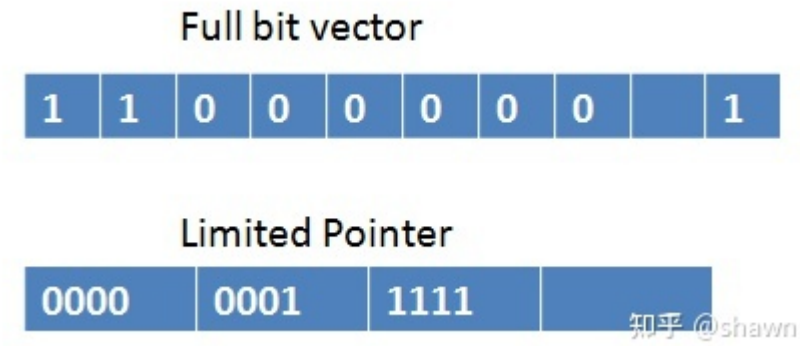
为了降低directory单条记录的额外开销，我们可以采用如下的方法：

3.3.2.1节点群组方案

在Shared-list中用一个bit表示一个节点群组而不是表示一个节点(Represent multiple processors in one bit)，然后在节点群组内部再通过一个snooping-based的方案来解决一致性问题。但是这样的话，就会产生很多额外的消息，复杂性也提高了。

3.3.2.2 有限指针方案

在实际的大型应用中，我们观察到在某个时间范围内很少有一个cache block会被所有的节点共享(data is expected to only be in a few caches at one time)，比如我们系统支持64个处理器，则一个cache block可能仅会在少数几个处理器上有拷贝，而绝大部分的处理器根本不会共享访问这些block, 所以我们可以把Directory中共享节点列表的one-hot bit vector信息改为有限个指针信息来表示（Limited Pointer Scheme）。



举个例子来阐述一下：假设一个系统中有1024个节点，而我们发现任何一个cache block的共享节点数不超过10个，而采用指针的方式表示1024个节点序号只需10bits,因此一个directory的共享列表长度从1024变为了10×10=100，大幅降低了额外的存储空间开销。我们可以把共享列表的长度设置为一个固定的值或者一个固定的节点数目比例（比如0.1）,但如果我们在应用中的某个偶然时刻，共享的节点数目超出了我们的预设值，那怎么办？我们还可以采用一些手段来处理，比如把director-based 方案回退为snooping-based 方案，也可以把某个最老的node给踢出去，其它的方法还有Coarse vector, Dynamic pointers等，大家可以参考相关的专门文章。

3.3.2.3 稀疏Directory方案

除了减少单条directory的占用空间外，我们还可以减少directory的总记录数目；因为我们在实际的应用中观察到一个现象：只有少部分内存会存在与cache中（only a few memory can be resident in cache）。比如一个系统拥有1MB的cache和1GB的主存储空间，如果我们把所有的内存空间全部记录在directory中，则浪费了99.9%的空间，因为只有那个1MB cache对应的directory条目才有价值，其它的条目都是空条目。稀疏directory方案（Sparse Directories Scheme）的核心思想就是减少空directory的内存占用, 实现方法可以是链表或者是一个查询表结构。