目录

[1 L1权限解析 3](#_Toc120525747)

[1.1 四种状态权限 3](#_Toc120525748)

[1.2 常用的状态转化函数 4](#_Toc120525749)

[2 ICache消息 6](#_Toc120525750)

[2.1 概述 6](#_Toc120525751)

[2.2 AcquireBlock(NtoB) 6](#_Toc120525752)

[2.3 Release（\_toN） 7](#_Toc120525753)

[2.4 Probe 7](#_Toc120525754)

[2.5 权限转换 8](#_Toc120525755)

[2.5.1 重填流程（GrantData） 8](#_Toc120525756)

[2.5.2 替换流程（Probe） 8](#_Toc120525757)

[3 DCache 9](#_Toc120525758)

[3.1 概述 9](#_Toc120525759)

[3.2 Acquire 10](#_Toc120525760)

[3.2.1 处理load请求miss时会发送Acquire 10](#_Toc120525761)

[3.2.2 MissQueue 10](#_Toc120525762)

[3.3 Release 11](#_Toc120525763)

[3.3.1 load替换块写回 11](#_Toc120525764)

[3.3.2 Store替换块写回 11](#_Toc120525765)

[3.4 Probe 12](#_Toc120525766)

[3.5 权限转换 12](#_Toc120525767)

[3.5.1 Grant[Data] 13](#_Toc120525768)

[3.5.2 Probe 13](#_Toc120525769)

[4 L2Cache 14](#_Toc120525770)

[4.1 概述 14](#_Toc120525771)

[4.2 缓存中与一致性相关的特性 16](#_Toc120525772)

[4.2.1 接收和发送TileLink总线请求及响应 16](#_Toc120525773)

[4.2.2 支持TileLink一致性协议 16](#_Toc120525774)

[4.2.3 为请求分配MSHR 16](#_Toc120525775)

[4.2.4 采用目录记录缓存行信息 17](#_Toc120525776)

[4.2.5 MSHR根据请求安排不同的任务 18](#_Toc120525777)

[4.2.6 高优先级对低优先级打断（请求嵌套） 18](#_Toc120525778)

[4.2.7 采用ProbeHelper处理Client Directory容量冲突 18](#_Toc120525779)

[4.3 MSHR 19](#_Toc120525780)

[4.3.1 概述 19](#_Toc120525781)

[4.3.2 常用函数 20](#_Toc120525782)

[4.3.3 权限控制：如何更新Self Directory / Client Directory中的权限。 21](#_Toc120525783)

[4.3.4 请求控制：是否需要向上下层Cache发送子请求，及等待响应 26](#_Toc120525784)

[4.3.5 子请求与响应参数设置 31](#_Toc120525785)

[4.3.6 优先级与复位 32](#_Toc120525786)

[4.3.7 请求嵌套 33](#_Toc120525787)

[5 L3Cache 35](#_Toc120525788)

[5.1 概述 35](#_Toc120525789)

[5.2 L3与L2结构上的主要区别 35](#_Toc120525790)

[6 香山南湖架构缓存一致性结构 36](#_Toc120525791)

[7 消息流与权限变化实例 37](#_Toc120525792)

[7.1 Acquire 37](#_Toc120525793)

[7.2 Release 40](#_Toc120525794)

[7.3 Put & Get 40](#_Toc120525795)

1. L1权限解析

TileLink->Metadata代码解析：

* 1. 四种状态权限

|  |  |
| --- | --- |
| 状态 | 权限 |
| Nothing | Invalid |
| Branch | 可读 |
| Trunk | 可读可写 |
| Dirty | 可读可写 |

可以注意到L1没有Tip状态但是多是多出了一个Dirty状态，这是因为L1支持的并不是完整的TileLink一致性状态，因为L1上面没有权限实体，所以其实它不需要完整的Tilelink权限表示（这里的Trunck其实是Tip,，Dirty只是在Trunk的基础上表示这个数据被修改过，这种方式是RocketChip里继承过来的），而L2则是完整的Tilelink权限状态（Tip, Trunck, Branch, Nothing）。

为支持一致性，Cache的meta信息中记录了对应数据块的一致性信息。具体地，数据块共有Nothing、Branch、Trunk和Dirty四种编号依次升高的状态，其中，所有高于Nothing的状态具有读权限，所有高于Branch的状态具有写权限。具体解释如下：

* Dirty：数据被写过
* Trunck: 表示有读写权限，但是数据时干净的
* Branch: 只有读权限
* Nothing: 没有权限

L1到L2Cache 的tilelink通道的user bits里里会有一个dirty位， 专门在传输时表明数据是否为脏（CD通道）。Dirty状态的存在意义是为了复用Dcache的一些处理函数。严格上一个cacheline 的dirty状态只是在ICache里转一圈，并不会对它做修改。

若一个块在DCache被修改过（Dirty），在ICache仍有可能记录这个块状态是Branch，因为在读取ICache数据时并不关心这个块是否为脏，只要它是最新的就可以了。

|  |  |  |  |
| --- | --- | --- | --- |
| 存储操作类别 | Write | WriteIntent | 备注 |
| wr | True | True | 要进行写或者可能会被写 |
| wi | false | True | 可能会被写 |
| rd | false | false | 读 |

* 1. 常用的状态转化函数

growStarter：确定这个cmd是否miss，以及新的状态（命中时）或要发送的参数（miss时）

|  |  |  |  |
| --- | --- | --- | --- |
| 要对存储进行的操作 | 当前状态 | 是否命中 | 新的状态/Acquire参数 |
| 命中：rd，wi不变，wr修改TRUNK未Dirty | | | |
| rd | Dirty | True | Dirty |
| rd | Trunk | True | Trunk |
| rd | Branch | True | Branch |
| wi | Dirty | True | Dirty |
| wi | Trunk | True | Trunk |
| wr | Dirty | True | Dirty |
| wr | Trunk | True | Dirty |
| 未命中 | | | |
| Rd | Nothing | False | NtoB |
| Wi | Branch | False | BtoT |
| Wi | Nothing | false | NtoT |
| Wr | Branch | False | BtoT |
| Wr | Nothing | False | NtoT |

growFinisher：根据Grant参数决定miss后进入什么状态。

|  |  |  |
| --- | --- | --- |
| 要对存储进行的操作 | 参数 | 新的状态 |
| Rd | toB | Branch |
| Rd | toT | Trunk |
| wi | toT | Trunk |
| wr | toT | Dirty |

onAccess：调用了growStarter查询这个缓存在这个块上是否有足够的权限来执行操作。以及接下来要做什么（获取消息参数或更新元数据）。

onSecondaryAccess：block上的一个第二次miss是否需要另一个 Acuqire 消息

onGrant：调用了growFinisher实现在Grant后的Metadata改变

shrinkHelper：根据Probe参数确定要变化的状态（可以发现处理probe的原则是probe对权限不能Grow只能Report或者Shrink）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 参数 | 状态 | 是否脏 | Ack返回 | 新权限 |
| toT | Dirty | True | TtoT | Trunk |
| toT | Trunk | False | TtoT | Trunk |
| toT | Branch | False | BtoB | Branch |
| toT | Nothing | False | NtoN | Nothing |
| toB | Dirty | True | TtoB | Branch |
| toB | Trunk | False | TtoB | Branch |
| toB | Branch | False | BtoB | Branch |
| toB | Nothing | False | NtoN | Nothing |
| toN | Dirty | True | TtoN | Nothing |
| toN | Trunk | False | TtoN | Nothing |
| toN | Branch | False | BtoN | Nothing |
| toN | Nothing | False | NtoN | Nothing |

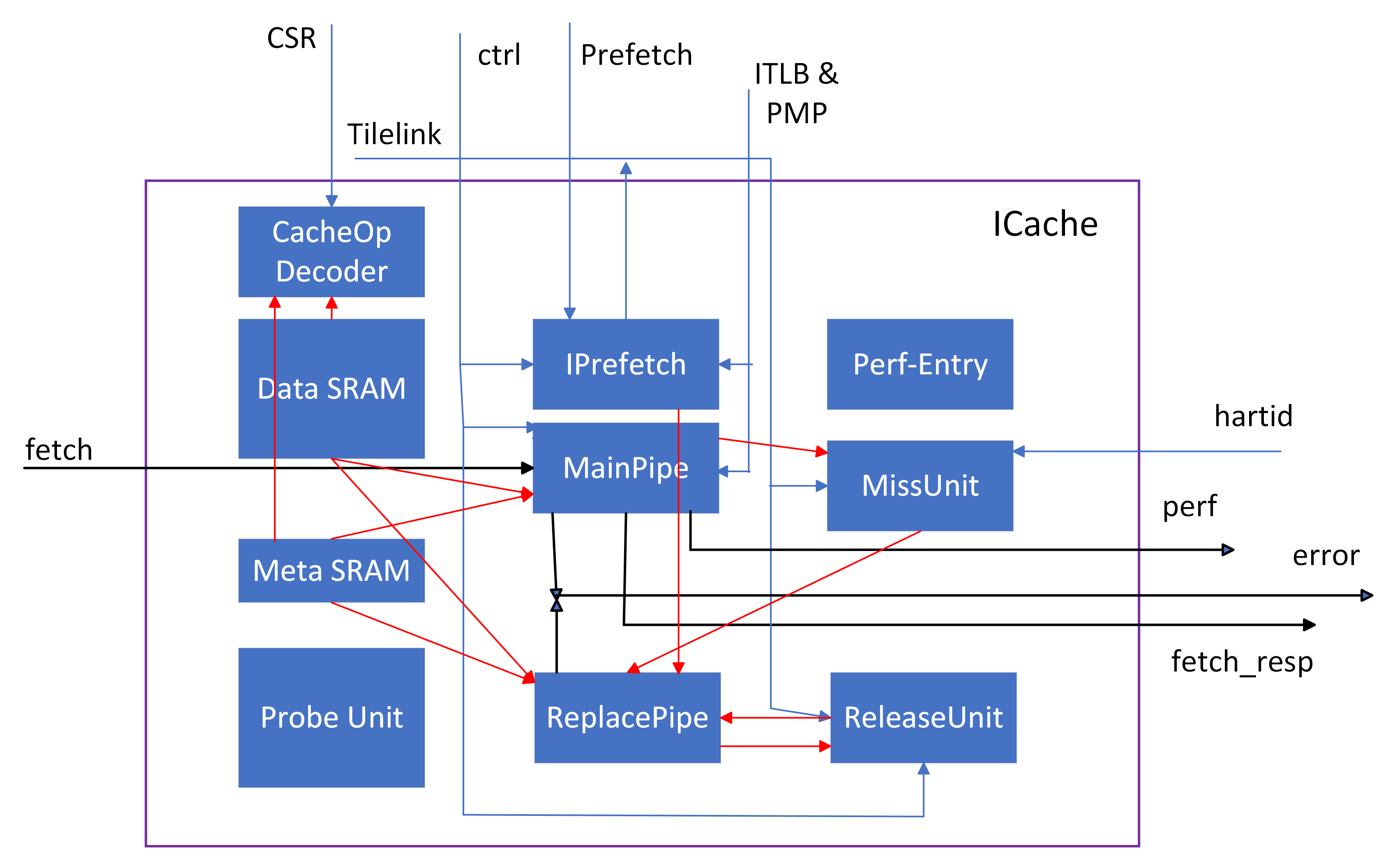
cmdToPermCap：将缓存控制cmds翻译成Probe参数

|  |  |  |
| --- | --- | --- |
| Cmd | 注释 | Param |
| M\_FLUSH | write back dirty data and cede割让 R/W permissions | toN |
| M\_PRODUCE | write back dirty data and cede割让 W permissions | toB |
| M\_CLEAN | write back dirty data and retain保留 R/W permissions | toT |

onCacheControl：shrinkHelper(cmdToPermCap(cmd))对Cache进行控制

onProbe：shrinkHelper(param)处理Probe请求

1. ICache消息
   1. 概述

**

Icache结构图

在ICache中，一致性信息仅在miss重填和替换两流程中会有改变。probe流程

默认情况下，倾向于在L2中存储data，即preferCache默认为True。

IPrefetch:

io.mem\_hint.bits.user.lift(PreferCacheKey).foreach(\_ := True.B)

* 1. AcquireBlock(NtoB)

当Cache出现miss时，将会把miss请求记录下来，利用Tilelink总线发送Acquire请求并等待总线回复的grant信号，将grant信号返回的数据暂存并发送grant\_ack请求表明收到了grant请求。

Miss Unit接收到来自Cache主流水线的请求，将miss请求存储到本模块内寄存器中，而后准备利用Tilelink总线在下一拍发送acquire请求。Acquire请求的id为本模块的id，地址为请求的物理地址，宽度为一个Cache行宽度，user信号为虚地址12-13比特，并指定向下发送的权限转换为NtoB(表示该Cache行在指令缓存中的权限提升，权限转换详见下表)

Tilelink Acquire信号fire后，本模块将等待tilelink总线的grant回复，mem\_grant将按beat返回数据及可能的数据损坏信号，本模块将返回数据全部暂存在respDataReg内，当refill结束后进入发送grant\_ack阶段。

Grant\_ack阶段将grant fire时的回复原样发回，表明收到了grant请求。本模块进入到替换Cache行的阶段。

* 1. Release（\_toN）

接下来，Cache会首先读出待替换的Cache行，而后利用Tilelink总线发送ReleaseData请求，其中携带刚刚读出的Cache行数据，传输完毕后等待D通道的回复ReleaseAck表明Cache行的替换过程结束。

总的来说，Acquire和Release的流程如下：MissUnit收到请求后，将向下游L2 Cache发送Tilelink的Acquire请求，等待MissUnit收到对应数据的Grant请求后如需替换Cache行则向ReleasePipe发送Release请求，Release Pipe会按照替换算法选出待替换Cache行，重新读出SRAM得到数据，然后向L2 Cache发送Release请求，最后MissUnit重填写SRAM，重填结束后返回数据给MainPipe，MainPipe将数据返回给IFU。

* 1. Probe

Probe主要用于数据同步(用于自修改代码中), 更具体地，程序写了代码段的数据后，fence.i指令将这个脏数据从DCache里写回到Memory来让修改可见，这个过程中L2会查询目录得知ICache中有旧数据，就向ICache发送Probe请求把旧数据无效掉（权限转换为toN），然后流水线重新执行再向L2发送AcquireBlock获取新数据

ICache提供了一致性支持，DCache写入的数据借助L2同步给ICache主要借助于Probe操作。外部的Probe操作在Probe队列有空闲时进入其中空闲Probe Entry模块中，Probe Entry内使用状态机将Probe请求转换为Replace Unit请求发送给Replace Unit。

* 1. 权限转换

|  |  |  |
| --- | --- | --- |
| 类型 | 参数 | 对应使用到的消息 |
| Permissions | None，Branch，Trunk | - |
| Cap | toT(0)，toB(1)，toN(2) | Probe，Grant[data] |
| Grow | NtoB(0)，NtoT(1)，BtoT(2) | AcquireBlock，AcquirePerm |
| Shrink | TtoB(0)，TtoN(1)，BtoN(2) | ProbeAck[data]，Realse[data] |
| Report | TtoT(0)，BtoB(1)，NtoN(2) | ProbeAck[data]，Realse[data] |

表 基于TileLink的一致性协议所需的权限转换

|  |  |
| --- | --- |
| 消息 | 参数 |
| AcquireBlock | NtoB |
| ReleaseData | TtoN，BtoN（取决于替换块的当前状态） |

表 ICache可发送的消息及其参数

ICache的权限变化由本身的权限和接收到的消息参数决定，详见下面的两个流程。

* + 1. 重填流程（GrantData）

接收到GrantData后的权限变化规则：

具体地，在miss重填处，根据grant（Acquire的回复）返回的参数及dirty信号不同，ICache数据块有不同的新权限：

|  |  |  |
| --- | --- | --- |
| 参数 | Dirty | 新权限 |
| toB | false | Branch |
| toB | True | Branch |
| toT | false | Trunk |
| toT | True | Dirty |

表 接收消息Grant时ICache的状态变化

* + 1. 替换流程（Probe）

接收到Probe后的权限变化规则：

在替换流程中，根据请求发来的参数及当前状态信号，ICache数据块有不同的新权限：

|  |  |  |
| --- | --- | --- |
| 参数 | 状态 | 新权限 |
| toT | Dirty | Trunk |
| toT | Trunk | Trunk |
| toT | Branch | Branch |
| toT | Nothing | Nothing |
| toB | Dirty | Branch |
| toB | Trunk | Branch |
| toB | Branch | Branch |
| toB | Nothing | Nothing |
| toN | Dirty | Nothing |
| toN | Trunk | Nothing |
| toN | Branch | Nothing |
| toN | Nothing | Nothing |

表 接收消息Probe时ICache的状态变化

在nothing的情况下L2任然会probe让L1toT是因为可能有多个操作对权限修改，假设有一个对地址A的Release请求先进入ReplacePipe，将状态从置为Nothing,之后有一个Probe toT的请求进入ReplacePipe，那状态已经改为Nothing了，自然就只能返回NtoN了，一个原则是：Probe只能降低权限，不能升高权限。

* 1. 请求嵌套

ICache只有Master，所以Acuqire和Release消息都是ICache主动发出的，只有Probe消息是被动接收的。

为了避免死锁，ICache收到了 Probe 就必须能够接收并且能够返回 ProbeAck，不能被 Grant 阻塞。在ICache里我们这样设计的：如果MissUnit里有正在处理的请求，但是这个请求的GrantAck已经发出去了，也就是说如果这个miss请求需要发送Release到ReplacePipe，那么直到它发送到ReleaseUnit之前，**相同set的Probe都要被阻塞（不进入ReplacePipe），而在等待Grant的时候这个Probe是不会被阻塞的**

1. DCache
   1. 概述

图 DCache结构图1图示

描述已自动生成

图 DCache结构图2

DCache模块接收来自核内访存流水线MemBlock的数据访存请求，根据请求类型与命中情况用不同流水线完成对缓存数据的读写和替换，并通过TileLink总线协议和L2 Cache交互完成数据块的写回、重填操作以及Probe请求。

对于Dcache默认情况下，倾向于不在L2存储data，即PreferCache默认为False。

MissQueue:

  // prefer not to cache data in L2 by default

  io.mem\_acquire.bits.user.lift(PreferCacheKey).foreach(\_ := false.B)

* 1. Acquire
     1. 请求miss时会发送Acquire

对于普通的Load请求，DCache从Load Queue 接收一条load指令后（实现的Load流水线有两条，可以并行处理两个load请求），根据计算得到的地址查询tagArray和metaArray，比较判断是否命中：若命中缓存行则返回数据响应；若缺失则选出替换路并分配MSHR (miss queue) 项，将请求交给Miss Queue处理，Miss Queue负责向L2 Cache发送 Acquire 请求取回重填的数据，然后通过Refill Pipe将重填数据块写入存储单元，同时把取回的重填数据转发给Load Queue完成响应；若被替换的块需要写回，则在Writeback Queue中向L2发送Release请求将其写回。

如果缺失的请求分配MSHR项失败，DCache会反馈一个MSHR分配失败的信号，由load保留站随后重新调度该load请求。

* + 1. MissQueue
       1. load请求

对于miss的load请求，Miss Queue为它分配一项空的Miss Entry，并且可以在一定条件下合并请求或拒绝请求，分配后在Miss Entry中记录相关信息。根据 way\_en 所在的块是否有效, 判断是否需要替换，如果要替换则向 Main Pipe 发送 replace 请求，并向L2发送AcquireBlock消息，等待 L2 返回权限 (Grant) 或者数据加权限 (GrantData)；在收到 GrantData 每一个 beat 后要将数据转发给 Load Queue；在收到 Grant / GrantData 第一个 beat 后向 L2 返回 GrantAck；在收到 Grant / GrantData 最后一个 beat，并且 replace 请求已经完成后, 向 Refill Pipe 发送 refill 请求, 并等待应答, 完成数据回填；最后释放Miss Entry。

* + - 1. store请求

对于miss 的store请求，和load的流程基本一致, 区别在于不需要把回填的数据转发给 Load Queue，而是在最终完成回填后由Refill Pipe向 Store Buffer 返回应答, 表示 store 已完成。

对于miss的原子指令，在 Miss Queue 中分配一项空的 Miss Entry, 并在 Miss Entry 中记录相关信息；向 L2 发送 AcquireBlock 请求；等待 L2 返回 GrantData；在收到 GrantData 第一个 beat 后向 L2 返回 GrantAck；在收到 GrantData 最后一个 beat 后向 Main Pipe 发送请求, 在 Main Pipe 中同时完成替换和回填, 完成后向 Miss Entry 返回应答；最后释放 Miss Entry。此外还会向 L2 发送 Acquire 请求，如果是对整个 block 的覆盖写则发送 AcquirePerm (L2 将会省去一次 sram 读操作)，否则发送 AcquireBlock；

* 1. Release
     1. load替换块写回

对于普通的Load请求，DCache从Load Queue 接收一条load指令后（实现的Load流水线有两条，可以并行处理两个load请求），根据计算得到的地址查询tagArray和metaArray，比较判断是否命中：若命中缓存行则返回数据响应；若缺失则选出替换路并分配MSHR (miss queue) 项，将请求交给Miss Queue处理，Miss Queue负责向L2 Cache发送 Acquire 请求取回重填的数据，然后通过Refill Pipe将重填数据块写入存储单元，同时把取回的重填数据转发给Load Queue完成响应；若被替换的块需要写回，则在Writeback Queue中向L2发送Release请求将其写回。

如果缺失的请求分配MSHR项失败，DCache会反馈一个MSHR分配失败的信号，由load保留站随后重新调度该load请求

* + 1. Store替换块写回

对于普通的Store请求，DCache从StoreBuffer接收一条store指令后，使用Main Pipe流水线计算地址查询tag和meta，判断是否命中，若命中缓存行则直接更新DCache数据并返回应答；若缺失则选出替换路并分配MSHR将请求交给Miss Queue，向L2请求要回填到Dcache的原目标数据行，然后通过Refill Pipe将回填数据写入DCache存储单元，在完成对该数据的store操作后向StoreBuffer返回应答；若被替换的块需要写回，则在Writeback Queue中向L2发送Release请求将其写回。

如果缺失的请求分配MSHR项失败，DCache会反馈一个MSHR分配失败的信号，由store保留站随后重新调度该store请求。

* 1. Probe

对于Probe请求，DCache从L2 Cache接收Probe请求后，使用Main Pipe流水线修改被Probe的数据块的权限，命中后下一拍返回应答。

ProbeQueue：负责接收并处理来自L2的一致性请求，包含 16 项 Probe Entry，每一项负责一个Probe 请求, 将 Probe 请求转成内部信号后发送到 Main Pipe, 由 Main Pipe 修改被 Probe 块的权限，等Main Pipe返回应答后释放Probe Entry。

Writeback Queue：包含18项WritebackEntry，负责通过TL-C的C通道向L2 Cache写回替换块(Release)，以及对Probe请求做出应答 (ProbeAck)，且支持Release和ProbeAck之间相互合并以减少请求数目并优化时序。

* 1. 权限转换

|  |  |  |
| --- | --- | --- |
| 类型 | 参数 | 对应使用到的消息 |
| Permissions | None，Branch，Trunk | - |
| Cap | toT(0)，toB(1)，toN(2) | Probe，Grant[data] |
| Grow | NtoB(0)，NtoT(1)，BtoT(2) | AcquireBlock，AcquirePerm |
| Shrink | TtoB(0)，TtoN(1)，BtoN(2) | ProbeAck[data]，Realse[data] |
| Report | TtoT(0)，BtoB(1)，NtoN(2) | ProbeAck[data]，Realse[data] |

表4.9基于TileLink的一致性协议所需的权限转换

|  |  |
| --- | --- |
| 消息 | 参数 |
| AcquirePerm | NtoB(0)，NtoT(1)，BtoT(2) |
| AcquireBlock | NtoB(0)，NtoT(1)，BtoT(2) |
| ReleaseData | TtoB(0)，TtoN(1)，BtoN(2) |
| Release | TtoB(0)，TtoN(1)，BtoN(2) |

表 DCache可发送的消息及其参数

Acquire：（MissQueue 408行）

 val grow\_param = req.req\_coh.onAccess(req.cmd).\_2

growPermissions = grow\_param

Release: (MainPipe 646行)

\* val (\_, miss\_shrink\_param, \_) = s3\_coh\_dup(2).onCacheControl(M\_FLUSH) //\_toN

//算 meta 的更新结果。这一行是运算的一部分

\* val writeback\_param = Mux(probe\_wb, probe\_shrink\_param, miss\_shrink\_param)

* + 1. Grant[Data]

|  |  |  |  |
| --- | --- | --- | --- |
| 存储操作类别 | Grant参数 | 数据是否dirty | 新权限 |
| Rd | toB | Flase | Branch |
| Rd | toB | True | Branch |
| Rd | toT | Flase | Trunk |
| Rd | toT | True | Dirty |
| Wi | toT | Flase | Trunk |
| Wi | toT | True | Dirty |
| Wr | toT | Flase | Dirty |
| Wr | toT | True | Dirty |

表 接收消息Grant时DCache的状态变化

Grant[Data]：（MissQueue 479行）

 refill.meta.coh := ClientMetadata(missCohGen(req.cmd, grant\_param, isDirty))

* + 1. Probe

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 参数 | 状态 | 是否脏 | Ack返回 | 新权限 |
| toT | Dirty | True | TtoT | Trunk |
| toT | Trunk | False | TtoT | Trunk |
| toT | Branch | False | BtoB | Branch |
| toT | Nothing | False | NtoN | Nothing |
| toB | Dirty | True | TtoB | Branch |
| toB | Trunk | False | TtoB | Branch |
| toB | Branch | False | BtoB | Branch |
| toB | Nothing | False | NtoN | Nothing |
| toN | Dirty | True | TtoN | Nothing |
| toN | Trunk | False | TtoN | Nothing |
| toN | Branch | False | BtoN | Nothing |
| toN | Nothing | False | NtoN | Nothing |

表 接收消息Probe时DCache的状态变化及回复

Probe:（MainPipe 498行）

  val (\_, probe\_shrink\_param, \_) = s3\_coh.onProbe(s3\_req\_probe\_param\_dup)

1. L2Cache
   1. 概述

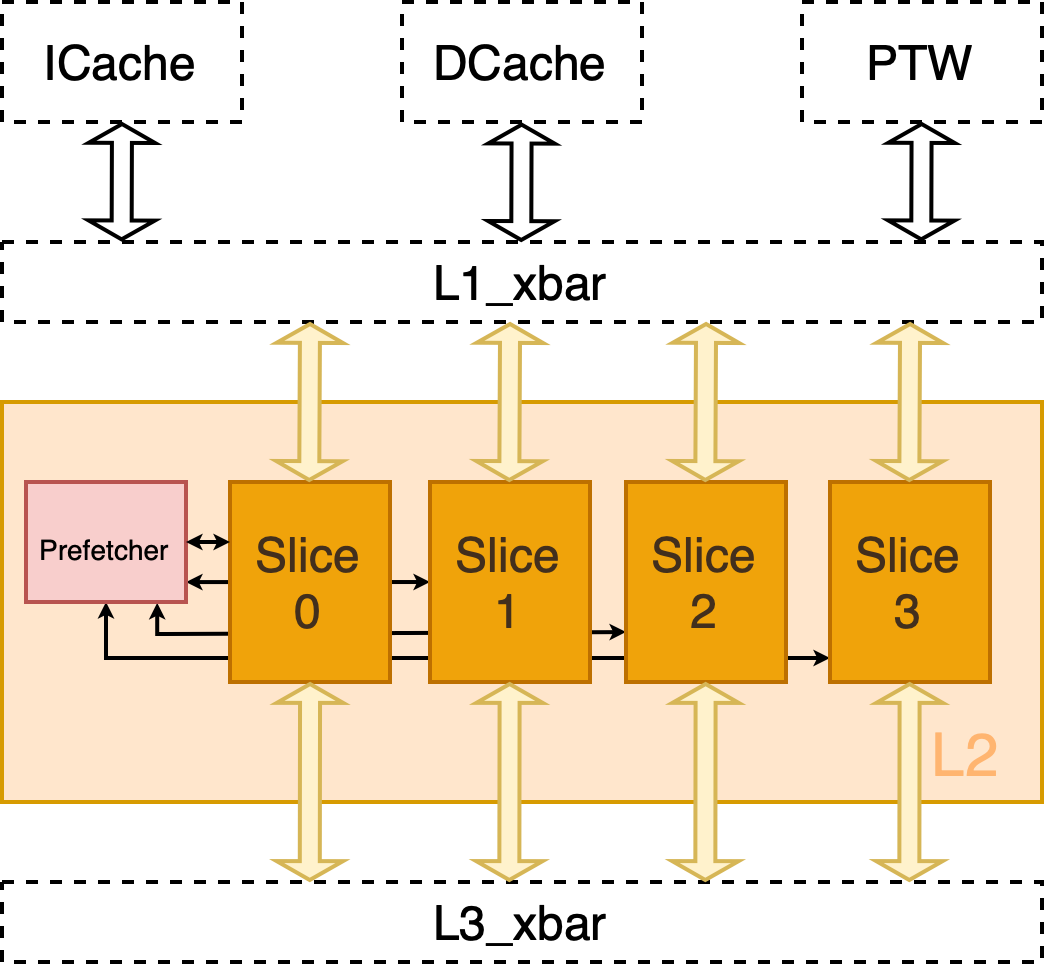


图 L2结构

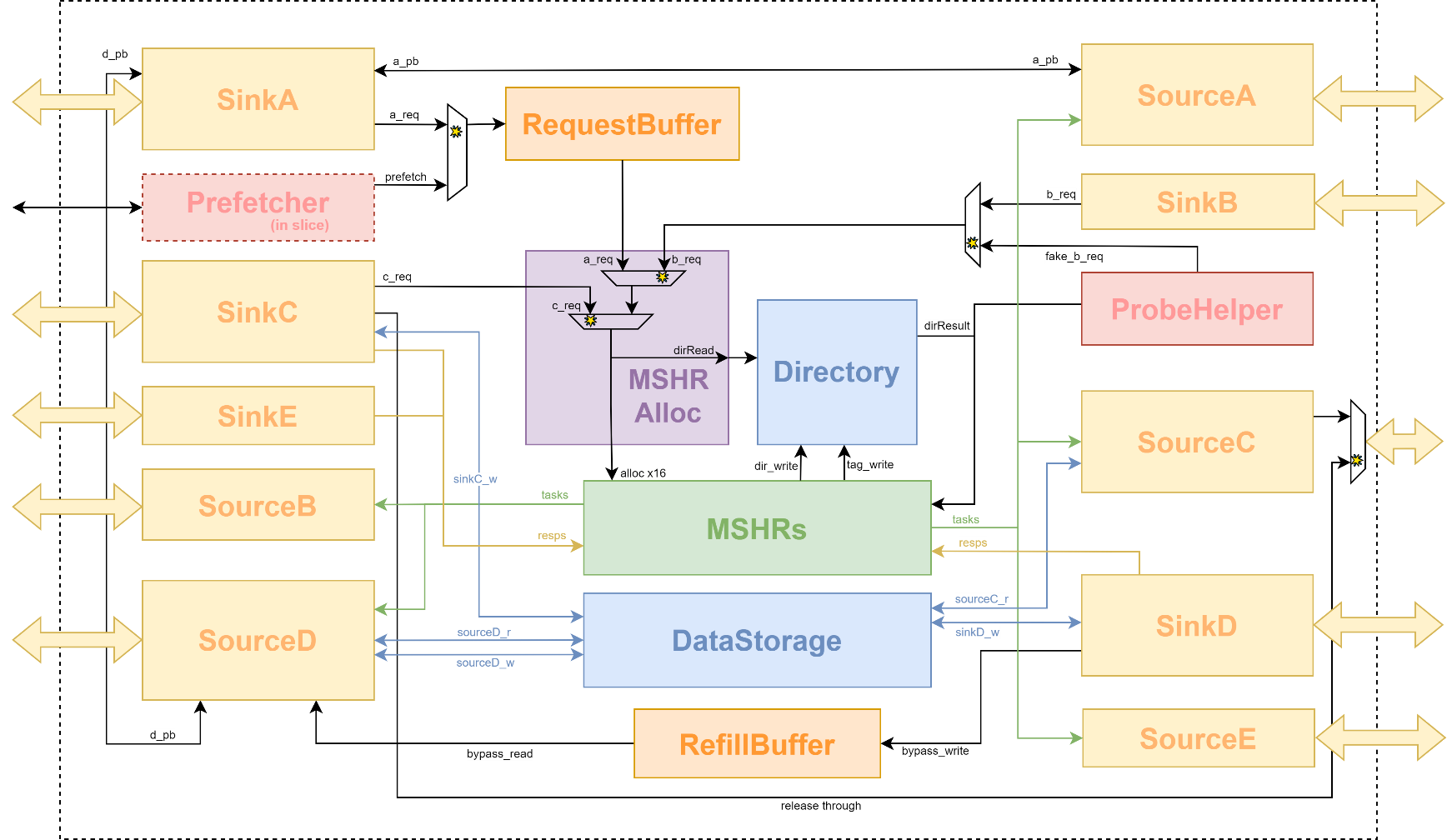


图 HuanCun Slice整体框图

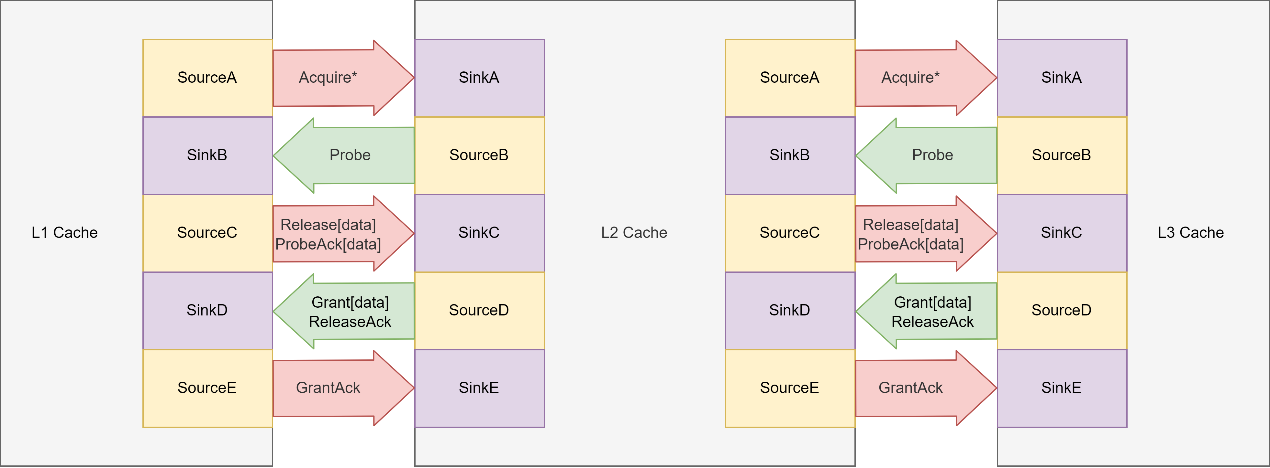


图 L2可收发信息（只包含和缓存一致性相关）

L2 HuanCun根据缓存块索引的低位分为4个Slice（也就是Bank，将address[5:0]作为块内偏移量，address[7:6]作为slice索引）。每个Slice分别处理其地址空间范围内的访存请求，即各个Slice之间是并行的。

HuanCun顶层模块只负责将4组TileLink通道的接口连到对应的Slice上，具体缓存事务的处理流程都由Slice完成。

关于PreferCache：

1. 因为PTW没有设置PreferCache的值，所以L2接收到PTW的消息时PreferCache应该使用的是默认值False。
2. 在SourceA中设置了PreferCache为False，也就是说L2向下传递A通道消息时PreferCache信号为False。
   1. 缓存中与一致性相关的特性
      1. 接收和发送TileLink总线请求及响应

根据TileLink总线协议，HuanCun包含Sink A~E和Source A~E十个通道控制模块。

在功能上，Sink 模块接收 TileLink 总线上的主动请求和被动响应，转化为MSHR内部请求。具体地，对于主动请求（SinkA，SinkB，SinkC），将该请求转换为 HuanCun内部请求发送给 MSHR Alloc 模块或者 Request Buffer。对于被动响应，则将响应回馈给正在处理对应请求的 MSHR（SinkD，SinkE）。而Source 模块接收 MSHR 的内部请求，经过处理和包装发送到 TileLink 总线上。

从方向上，SinkA，SinkC，SinkE，SourceB，SourceD面向上层，SourceA，SourceC，SourceE，SinkB，SinkD面向下层。

* + 1. 支持TileLink一致性协议

XiangShan的缓存系统中每个地址都遵循TileLink一致性树的规则。每个块在缓存系统的各层都有N(None)、B(Branch)、T(Trunk)、TT(Trunk TIP)四个状态，分别对应着没有权限、只读、不可读不可写、可读可写。

一致性树按照内存、L3、L2、L1的顺序自下而上生长，内存作为根节点拥有可读可写的权限，子节点的权限都不能超过父节点的权限。其中TT代表拥有T权限的最上层子节点（也是T权限树的叶子节点），说明该节点上层只有N或B权限，相反T权限而不是TT权限的节点代表上层一定还有T/TT权限节点。详细规则请参考TileLink手册9.1章。

* + 1. 为请求分配MSHR

MSHR Alloc模块接收SinkA，SinkB，SinkC收到的TileLink总线请求，为任务分配MSHR，并同时向目录发起读请求。

为了避免多个并发请求同时在 Cache 中被处理时产生资源竞争和相互干扰，我们的MSHR以 Set 为粒度对请求进行阻塞。根据 TileLink手册规范，为了避免系统中出现死锁，高优先级的请求要能够打断低优先级的请求先执行（优先级：通道C > B > A）， 因此 HuanCun设计了 N (默认配置N=14) 个abc MSHR，1 个 b MSHR，1 个 c MSHR。

在没有 Set 冲突的情况下，所有请求都在 abc MSHR 中挑选一个空项进入；当新到达的高优先级请求与 MSHR 中已有的低优先级请求发生Set冲突，且满足特定的嵌套条件时，高优先级请求会进入专用 MSHR。

（注：后面的接口信号中可能会出现abc/bc/c\_XX，其中的bc对应的都是b MSHR）

* + 1. 采用目录记录缓存行信息

HuanCun 是基于目录结构的 Non-inclusive Cache（此处所指的“目录”是广义的，包含元数据和Tag）。多级 Cache 之间，数据是 Non-inclusive 的，而目录则是 Inclusive 的。

在结构组织上，HuanCun包含上层数据目录Client Directory（包含权限state和别名位alias，详见特性9）和本地数据目录Self Directory（包含权限state，脏位dirty，上层权限clientStates，是否是被预取的prefetch）。两者结构类似。以 Set 为索引，每一个 Set 中有 Way 路数据。

当 MSHR Alloc 模块将一个请求分配进入 MSHR 时，它也会同时向目录发起读请求。并行读取上层与本地的元数据与 Tag，根据 Tag 比对判断是否命中，依命中情况将相应路的元数据传递到对应的 MSHR 中。

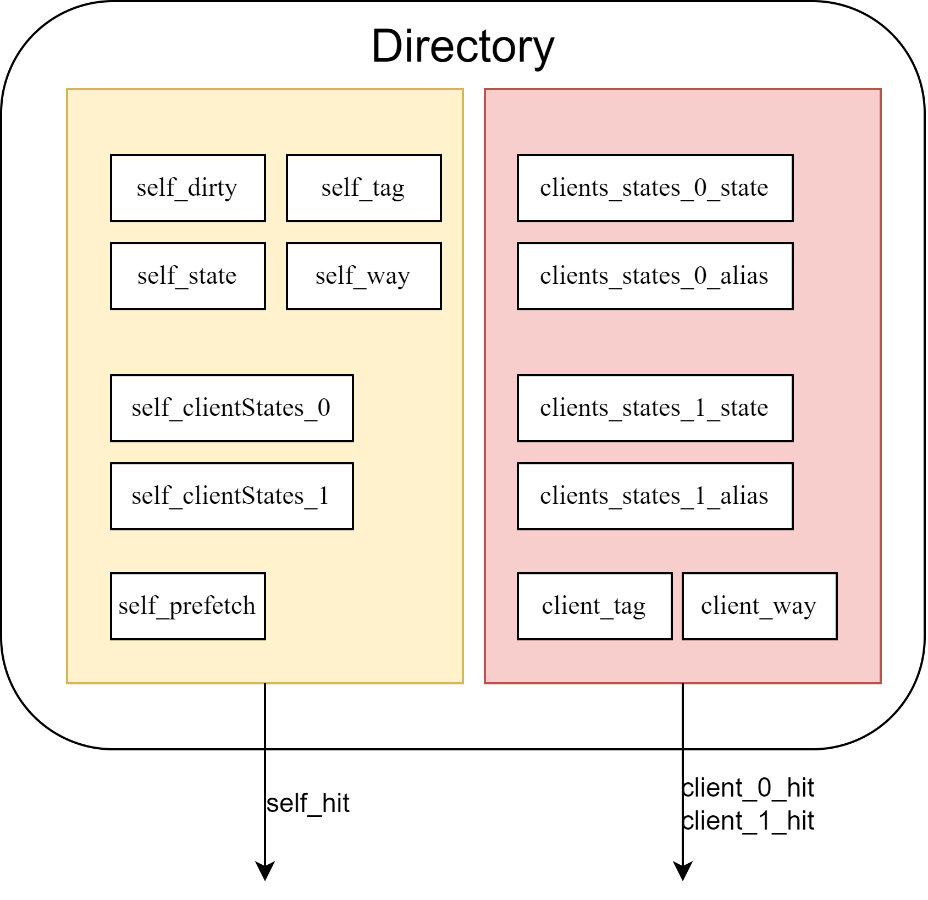
****

图 缓存结构

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 假设请求的tag | 假设替换的tag（若需要） | 对比tag是否命中 | self dir传递的路 | client dir传递的路 |
| 0x1 | 0x2 | hit | 0x1 | client hit：0x1  client miss：0x2 |
| 0x1 | 0x2 | miss | 0x2 | client hit：0x1  client miss：0x2 |

表 self目录与client目录

* + 1. MSHR根据请求安排不同的任务

当HuanCun接收到上层Cache的Acquire/Release请求，或者接收到下层Cache的Probe请求时，会为该请求分配一项MSHR，同时通过读目录获取该地址在本层及以上Cache中的权限信息，MSHR根据这些信息进行权限控制和请求控制。

权限控制：如何更新Self Directory / Client Directory中的权限。

由于目录是严格采用包含策略的，所以MSHR可以拿到请求地址在ICache、DCache和L2 Cache中的所有权限中最高的信息，从而控制该地址权限的变化。MSHR根据请求的类型和读目录的结果更新Self Directory的dirty位、权限域、clientStates域，其中clientStates表示如果该地址在当前Cache中有权限(B及以上权限)，那么这个地址在上层的L1 Cache中的权限是什么。此外，MSHR还会更新ICache和DCache对应的client Directory，包括权限域和别名域。

子请求控制：是否需要向上下层Cache发送子请求，及等待响应。

MSHR需要根据请求的内容和读目录的结果判断需要完成哪些子请求（比如是否需要向下Acquire或Release，是否需要向上Probe，是否应该触发一条预取，是否需要修改目录和tag等等），以及需要等待哪些子请求的应答。MSHR将这些【要调度的请求】和【要等待的应答】具体成一个个事件，并用一系列状态寄存器记录这些事件是否完成。当所有事件都完成后，该项MSHR就会被释放。详细见MSHR模块部分。

* + 1. 高优先级对低优先级打断（请求嵌套）

根据TileLink手册，高优先级请求要能够打断低优先级请求的执行（优先级C>B>A）。

MSHR根据自己当前内部请求的执行状态，告诉MSHR Alloc是否能够接收高优先级请求的打断。如果可以被打断，且有一个同Set的高优先级请求H进入，那么MSHR就会为高优先级请求H分配B/C MSHR，并且阻塞低优先级MSHR任务的执行。

高优先级请求H会将自己的信息传给低优先级请求L，L会根据状态给H一些指示信息（比如ReleaseDataThrough）。高优先级任务完成后，如果对目录进行了更改，则需要同步更新低优先级MSHR中保存的目录状态。

* + 1. 采用ProbeHelper处理Client Directory容量冲突

由于 HuanCun采用了 Inclusive-directory & Non-inclusive data 的设计，当 Client Directory 由于容量有限而无法存放新的 Cache Block目录时，需要本级 Cache 向上级发送 Probe 请求，将上级的一个旧 Cache Block Probe下来，再将新的 Cache Block 对应状态存入 Client Directory中。

为了简化单个 MSHR 的处理流程，我们设计了 ProbeHelper 来监听 Client Directory 的读取结果，如果发现了容量冲突，则由 ProbeHelper 生成一个伪造的 B 请求，将目标块从上层 Probe 下来。 这样单个 MSHR 中便不需要考虑 Client Directory 的容量冲突问题，简化了设计。

* 1. MSHR
     1. 概述

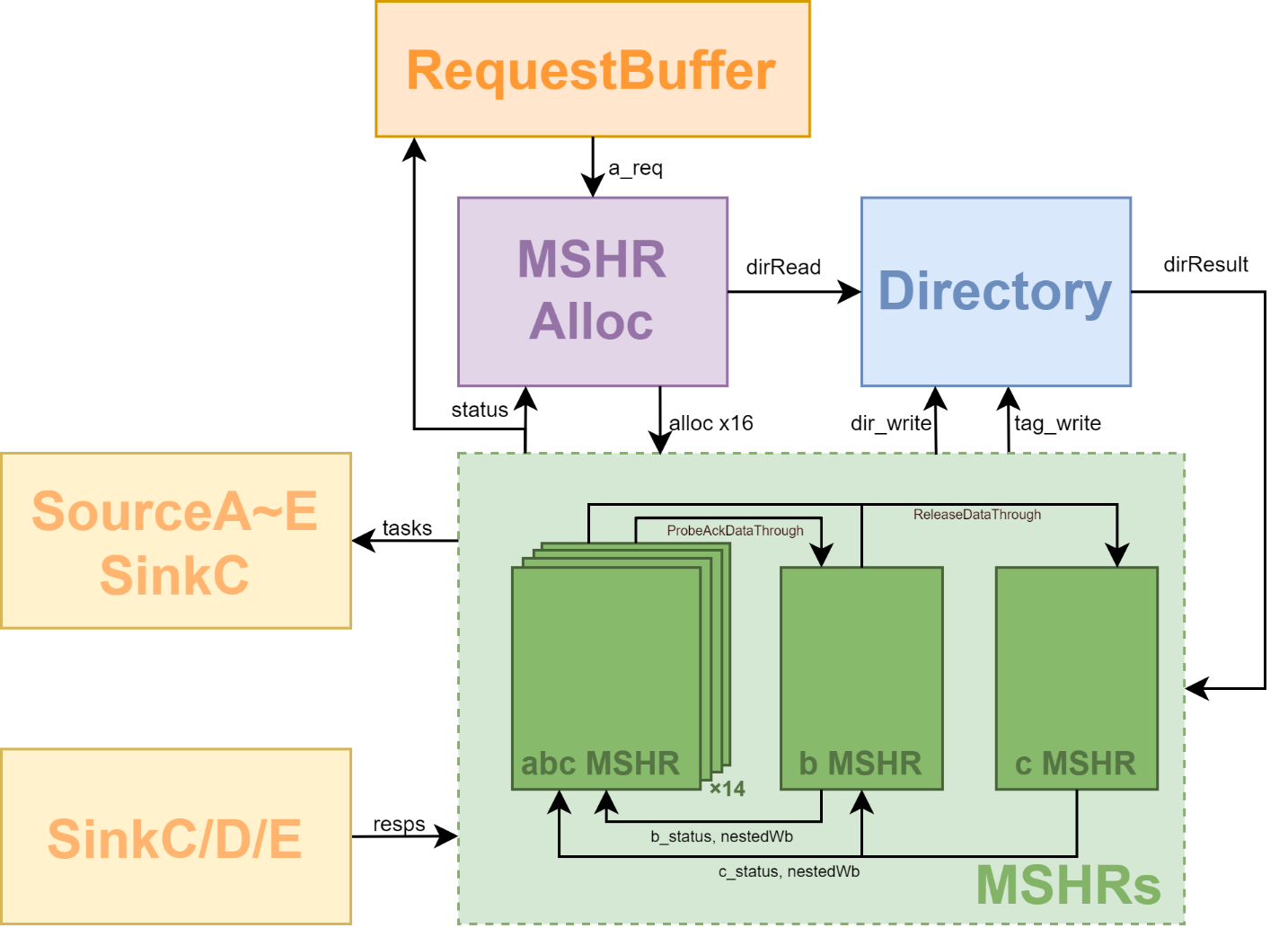
**

图 MSHR整体框图

当HuanCun接收到上层Cache的Acquire/Release请求，或者接收到下层Cache的Probe请求时，会为该请求分配一项MSHR，同时通过读目录获取该地址在本层及以上Cache中的权限信息，MSHR根据这些信息进行权限控制和请求控制。

\*因为本文档主要考察的是缓存的一致性，所以关于alias，预取以及probeHelper的部分没有进行描述，留给相关的文档进行补充。

* + 1. 常用函数

needT(opcode: UInt, param: UInt): Bool =（需要权限T才能处理该请求）

|  |  |  |
| --- | --- | --- |
| operation | param | 是否需要T权限 |
| PutFullData | - | True |
| PutPartialData | - | True |
| ArithmeicData | - | True |
| LogicalData | - | True |
| Get | - | False |
| Hint | PREFETCH\_WRITE | True |
| AcquireBlock | \_toT | True |
| AcquirePerm | \_toT | True |
| else | | False |

\*只适合处理A通道的消息

probe\_next\_state(state: UInt, param: UInt):

|  |  |  |
| --- | --- | --- |
| state | param | Next\_state |
| TIP/TRUNK | toT | state |
| TIP/TRUNK | toB | BRANCH |
| BRANCH | toT/toB | BRANCH |
| INVALID | - | INVALID |
| - | toN | INVALID |

shrink\_next\_state(param: UInt):

|  |  |
| --- | --- |
| param | NextState |
| TtoB | BRANCH |
| BtoB | BRANCH |
| else | INVALID |

\*在当前设计中Probe toT是不被允许的。

probe\_shrink\_perm(state: UInt, param: UInt):

|  |  |  |
| --- | --- | --- |
| State | Param |  |
| - | toN | True |
| TIP/TRUNK | toB | True |
| else | | False |

probe\_is\_report(state: UInt, param: UInt):

|  |  |  |
| --- | --- | --- |
| State | Param |  |
| BRANCH | toB | True |
| else | | False |

* + 1. 权限控制：如何更新Self Directory / Client Directory中的权限。

由于目录是严格采用包含策略的，所以MSHR可以拿到请求地址在ICache、DCache和L2 Cache中的所有权限信息，从而控制该地址权限的变化。MSHR根据请求的类型和读目录的结果更新Self Directory的dirty位、权限域、clientStates域，其中clientStates表示如果该地址在当前Cache中有权限(B及以上权限)，那么这个地址在上层的L1 Cache中的权限是什么。此外，MSHR还会更新ICache和DCache对应的client Directory，包括权限域和别名域。

处理优先级：C > B > A

* + - 1. 处理完 c 请求(Release)后应当如何更新目录

new\_self\_meta.dirty : Self Directory的dirty位

self\_meta.hit && self\_meta.dirty || req.dirty && isParamFromT(req.param)

在hit以及是Release Tto\_ 的前提下，若自身或传下来的数据为dirty则更新Directory的dirty为True

new\_self\_meta.state：Self Directory的权限域

|  |  |  |
| --- | --- | --- |
| Req.param | new\_self\_meta.state | 条件 |
| TtoT（Report） | Trunk | - |
| TtoB | TIP | - |
| TtoN | TIP | - |
| BtoN | TIP | self\_meta.hit && self\_meta.state === TIP |
| BtoN | BRANCH | !self\_meta.hit || self\_meta.state =/= TIP |
| -(默认) | self\_meta.state | - |

new\_self\_meta.clientStates：Self Directory的clientStates域

对源client：

|  |  |
| --- | --- |
| Req.param | clientStates |
| \_toN | INVALID |
| \_toB | BRANCH |
| -(默认) | self\_meta.clientStates |

new\_clients\_meta：client Directory的权限域

对源client：

|  |  |
| --- | --- |
| Req.param | clientStates |
| \_toN | INVALID |
| \_toB | BRANCH |
| -(默认) | self\_meta.clientStates |

* + - 1. 处理完b请求（Probe）后如何更新目录

new\_self\_meta.dirty : Self Directory的dirty位

 new\_self\_meta.dirty := req.fromProbeHelper && !probeAckDataThrough && (self\_meta.hit && self\_meta.dirty || probe\_dirty)

dirty条件：

1. 有该数据且它本来就是dirty或是probe下来的数据是dirty的
2. probe来自probeHelper
3. ProbeAck不直接传递到下层

new\_self\_meta.state：Self Directory的权限域

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Self.meta.hit | 来自ProbeHelper且ProbeAck不直接传递到下层  （probeAckDataThrough） | state | param | Next\_state |
| True | False | TIP/TRUNK | toT | state |
| TIP/TRUNK | toB | BRANCH |
| BRANCH | toT/toB | BRANCH |
| - | toN | INVALID |
| True | TIP | - | TIP |
| TRUNK | - | TIP |
| BRANCH | - | BRANCH |
| False | - | - | - | 不变 |

\*如果这个请求是来自ProbeHelper，req参数必须是toN，但是self state不应该变成INVALID。

new\_self\_meta.clientStates：Self Directory的clientStates域

probe\_next\_state(state: UInt, param: UInt):

|  |  |  |  |
| --- | --- | --- | --- |
| Self.meta.hit | state | param | Next\_state |
| True | TIP/TRUNK | toT | state |
| TIP/TRUNK | toB | BRANCH |
| BRANCH | toT/toB | BRANCH |
| INVALID | - | INVALID |
| - | toN | INVALID |
| False | - | - | 不变 |

new\_clients\_meta：client Directory的权限域

m.state := Mux(m.hit, shrink\_next\_state(client\_probeack\_param\_vec(i)), clients\_meta\_reg(i).state)

|  |  |  |
| --- | --- | --- |
| Clients\_meta.hit | (client probeAck)param | NextState |
| True | TtoB | BRANCH |
| BtoB | BRANCH |
| else | INVALID |
| False | - | 不变 |

* + - 1. 处理完a请求后如何更新目录

A通道处理的请求:Acquire / Intent / Put / Get / Atomics

Self Directory的dirty位

请求为Acquire：

|  |  |  |  |
| --- | --- | --- | --- |
| 处理该操作需要T权限(eedT) | Self\_meta.hit | 需要将Client提升到T | New\_dirty |
| True | - | - | False |
| False | True | True | False |
| True | False | self\_meta.dirty or probe\_dirty |
| False | - | gotDirty || probe\_dirty |

请求不为Acuqire:

|  |  |  |
| --- | --- | --- |
| 请求 | 条件 | New\_dirty |
| Put\* | - | True |
| Else（Hint&Get） | gotDirty | True |
| probe\_dirty | True |
| self\_meta.hit && self\_meta.dirty | True |

\*self\_meta.dirty：自身数据是脏的

\*probe\_dirty：从别client Probe下来的块是脏的

\*gotDirety：从下层获得的GrantData是脏的

对于Acquire修改self目录原则：

1. 读self目录命中要改
2. preferCache要改
3. 需要向上发送probe时读目录miss要改

new\_self\_meta.state：Self Directory的权限域

1. 请求需要自身为T才能处理（needT）:

|  |  |
| --- | --- |
| 请求 | Next\_state |
| Acquire\* | TRUNK |
| Else | TIP |

1. 请求不需要自身为T就能处理（no needT）:
2. 读self目录miss：

|  |  |  |  |
| --- | --- | --- | --- |
| 自身无数据但别的client有该数据 | gotT：sinkD通道得到的Grant消息参数为toT | 请求为acquire且promoteT\_safe | Next\_state |
| True | True | True | TRUNK |
| False | TIP |
| False | - | 整个层面的最高perm(权限)，包括自己和client |
| False | True | Ture | TRUNK |
| False | TIP |
| False | - | BRANCH |

（没有考虑CacheLine冲突）

\*promoteT\_safe：将权限提升到T是安全的

何时不安全：（和通道C的回复有关）

when(req\_valid && sink\_c\_resp\_valid) {

when (

!acquire\_flag && req.fromA &&//当需要发送Acquire标记flag为Ture

probeack\_last && resp.last && !resp.hasData && !nested\_c\_hit && !self\_meta.hit

) {

promoteT\_safe := false.B

}

}

1. 读self目录hit：

|  |  |  |
| --- | --- | --- |
| State | 条件 | Next\_state |
| INVALID | - | BRANCH |
| BRANCH | - | BRANCH |
| TRUNK | - | TIP |
| TIP | 该数据在所有client中都是INVALID且请求为Acquire | TRUNK |
| Else | TIP |

1. 需要INVALID自己的块时self的新状态为INVALID

（如果我们一开始release了自己的块，这个时候需要把状态设置为INVALID）

when(inv\_self\_dir){

new\_self\_meta.state := INVALID

}

new\_self\_meta.clientStates：Self Directory的clientStates域

（待补充）

1. 请求源对应的client

|  |  |  |
| --- | --- | --- |
| 请求 | 条件 | Next\_State |
| Acuqire\* | needT或promoteT | TIP |
| else | BRANCH |
| Get | Client查询命中 | BRANCH |
| 未命中 | INVALID |
| Else | Client查询命中 | 不变 |
| 未命中 | INVALID |

1. 非请求源

|  |  |  |  |
| --- | --- | --- | --- |
| 请求 | 一级条件 | 二级条件 | Next\_State |
| Acuqire\*或Put\* | 1. Acquire.param不为NtoB 2. 请求promoteT 3. 请求是put | - | INVALID |
| else | Client查询命中且MSHR正在调用source b执行probe | cond |
| else | INVALID |
| Get | Client查询命中且MSHR正在调用source b执行probe | - | perm\_after\_probe |
| - | - | INVALID |
| Else（Hint） | 预测失败需要Probe | PREFETCH\_READ | perm\_after\_probe |
| Else | INVALID |
| Else | - | 不变 |

\*cond在L2中为：Mux(a\_probe\_clients(i), perm\_after\_probe, clients\_meta\_reg(i).state)

\*a\_probe\_clients：这个client应该被probe吗

\* perm\_after\_probe：shrink\_next\_state(client\_probeack\_param\_vec(i))

\*cond在L3中等价于perm\_after\_probe

new\_clients\_meta：client Directory的权限域

1. 请求源对应的client

|  |  |  |  |
| --- | --- | --- | --- |
| 请求 | 一级条件 | 二级条件 | Next\_State |
| Acquire\* | needT或promoteT | - | TIP |
| Else | - | BRANCH |
| else | - | - | 不变 |

\*req\_promoteT = req\_acquire && Mux(//将client提升到T

    self\_meta.hit,//自身命中

    meta\_no\_clients && self\_meta.state === TIP,//client无权限且自身的权限是TIP

    gotT && promoteT\_safe //Grant toT && 提升到T是安全的

  )

1. 非请求源

|  |  |  |  |
| --- | --- | --- | --- |
| 请求 | 一级条件 | 二级条件 | Next\_State |
| Acuqire\*或Put\* | 1. Acquire.param不为NtoB 2. 请求promoteT 3. 请求是put | Client查询命中且MSHR正在调用source b执行probe | INVALID |
| else | 不变 |
| Else（NtoB） | Client查询命中且MSHR正在调用source b执行probe | Cond |
| Else | 不变 |
| Else（Hint） | 预测失败需要Probe | PREFETCH\_READ | perm\_after\_probe |
| Else | INVALID |
| Else | - | 不变 |

1. 若请求时Get

|  |  |
| --- | --- |
| 条件 | Next\_State |
| Client查询命中且MSHR正在调用source b执行probe | perm\_after\_probe |
| Else | 不变 |

* + 1. 请求控制：是否需要向上下层Cache发送子请求，及等待响应

MSHR需要根据请求的内容和读目录的结果判断需要完成哪些子请求，包括是否需要向下Acquire或Release，是否需要向上Probe，是否应该触发一条预取，是否需要修改目录和tag等等；除了子请求，MSHR还要记录需要等待哪些子请求的应答。

MSHR将这些要调度的请求和要等待的应答具体成一个个事件，并用一系列状态寄存器记录这些事件是否完成。s\_\*寄存器表示要调度的请求，w\_\*寄存器表示要等待的应答，MSHR在拿到读目录的结果后会把需要完成的事件(s\_\*和w\_\*寄存器)置为false，表示请求还未发送或应答还没有收到，在事件完成后再将寄存器置为True。当所有事件都完成后，该项MSHR就会被释放。

可以调度的请求汇总：

|  |  |
| --- | --- |
| 要调度的请求 | 注释 |
| s\_acquire | 在source\_a上发送acquire请求 |
| s\_probe | 在source\_b上发送probe请求 |
| s\_release | 在source\_c上发送release请求 |
| s\_probeack | 在source\_c上发送probeack响应 |
| s\_execute | 在source\_d上发送Grant[data]或ReleaseAck响应 |
| s\_grantack | 在source\_e上发送GrantAck响应 |
| s\_wbselfdir | 写self目录 |
| s\_wbselftag | 写self tag |
| s\_wbclientsdir | 写clients 目录 |
| s\_wbclientstag | 写clients tag |
| s\_transferput | 将sink a（Put）中数据写入 |
| s\_writerelease | 将sink c（ReleaseAckData）中数据写入储存或者外部cache（通过release through） |
| s\_writeprobe | 将sink c（probeAckData）中数据写入 |
| s\_triggerprefetch | - |
| s\_prefetchack | - |

可以等待的响应汇总：

|  |  |
| --- | --- |
| 要等待的响应 | 注释 |
| w\_probeackfirst | 等待probeack的第一拍 |
| w\_probeacklast | 等待probeack的最后一拍 |
| w\_probeack | 等待probeack |
| w\_grantfirst | 等待grant的第一拍 |
| w\_grantlast | 等待grant的最后一拍 |
| w\_grant | 等待grant |
| w\_releaseack | 等待releaseack |
| w\_grantack | 等待grantack |
| w\_putwritten |  |

* + - 1. c\_schedule（Release[data]）

|  |  |  |  |
| --- | --- | --- | --- |
| 要调度的请求 | 注释 | 一级触发条件 | 二级触发条件 |
| s\_execute | 发送ReleaseAck | - | - |
| s\_wbclientsdir | param不是Report的情况写client目录更新state | client\_shrink\_perm  (release TtoB,TtoN,BtoN) | - |
| s\_wbselftag | 当需要保存Release的Data需要调度的任务 | will\_save\_release  保存release数据  （默认为True，详情请看4.3.7.1） | !self\_meta.hit且请求为releaseData  （读self目录miss并且release有数据时，在自己的dir中分配一个新的块） |
| s\_wbselfdir | self\_meta.hit或为releaseData |
| s\_release（替换） | 1. 读self目录miss 2. 请求为releaseData 3. self\_meta不为INVALID 4. 替换时需要向下发送release |
| w\_releaseack（替换） |
| w\_releaseack | 根据Slice结构Release直通的情况只需要等待回复即可，不需要发送release | Release直通且请求为releaseData | - |
| s\_writerelease | 数据写回存储中或传到下层Cache（release直通的情况） | req为releaseData | - |

\* replace时需要release的两种情况：1.自己的perm大于所有client的权限 2.这是一个脏块且自己是T

对于ReleaseDataThrough，（需要替换时）不需要向下发送release，不需要写self目录和tag，只需要等待releaseAck即可。

* + - 1. b\_schedule（Probe）

|  |  |  |  |
| --- | --- | --- | --- |
| 要调度的请求 | 备注 | 一级触发条件 | 二级触发条件 |
| s\_probeack | 在Source\_c进行回复 | 非ProbeAck直通 | 不来自ProbeHelper |
| s\_wbselfdir | 写self目录 | 读self目录命中 | Probe要对权限进行Shrink |
| s\_probe | 向上层发送probe | 不需要向上层probe的情况：   1. self命中 2. self的状态是B/Tip 3. probe参数是toB | clients至少有一个命中 |
| s\_wbclientsdir |
| w\_probeackfirst |
| w\_probeacklast |
| w\_probeack |

\*如果自己的Meta是B/Tip，参数是toB，则不需要Probe client。

对于ProbeAckDataThrough，只是不用发送ProbeAck，仍然需要需改自身目录。

* + - 1. a\_schedule(Acuquire\*,Put\*,Get,atomics,Hint)

|  |  |  |  |
| --- | --- | --- | --- |
| 要调度的请求 | 备注 | 一级触发条件 | 二级触发条件 |
| s\_execute | 在source\_d发送回复消息 | Req不为Hint | - |
| S\_release | set 容量不够了，需要进行替换并向下release | 1. 未命中 2. 需替换块的state不为INVALID 3. 替换时需要Release | 两种条件满足一种  一：   1. preferCache 2. 请求为AcquireBlock或Get   二：   1. 预取需要数据 2. transmit from other client 3. cache alias   (或的关系) |
| W\_releaseack |
| a\_do\_release  （不属于调度，但其与a和c的嵌套相关） |
| S\_acquire（在s\_transferPut的情况下表示发送Put） | Acquire需要向下传递 | 请求是Acquire\*：   1. needT且本层及上层的权限最高不为T（除源client外） 2. !needT且本层及上层权限都为INVALID（除源client外）   请求不是Acquire\*：   1. needT且本层及上层权限最高不为T 2. !needT且本层及上层权限都为INVALID   （或的关系） | - |
| W\_grantfirst |
| W\_Grantlast |
| W\_Grant（在非acquire中表示w\_accessack） |
| S\_Grantack |
| s\_grantack | !bypassGet 且!bypassPut |
| s\_wbselfdir | 1. preferCache或读目录命中且req不为Get 2. !acquirePermMiss |
| SetProbe  （self未命中时会调度s\_wbselfdir,s\_wbselftag） | 需要向上发送probe | req不为Get或Put：  对于非源client：   1. client命中 2. req为Acquire\*   （预取和alias部分待补充） | 1. 需要本地为T才能处理 2. self未命中 3. 非源client的上层状态是T   （或的关系） |
| S\_wbclientdir |
| req为Get或Put：   1. client命中 2. 三个条件满足其一： 3. client的状态为T 4. self未命中 5. 请求为Put | - |
| s\_wbselfdir | 读self目录命中（该条件包含在req为Get或Put中） |
| W\_Grant | 接收到Acquire请求时需要处理回复Grantack | Req为Acquire | - |
| S\_wbselfdir | 1. !acquirePermMiss 2. 读self目录命中或preferCache |
| s\_wbclientsdir | - |
| s\_wbclientstag | 源client未命中 |
| s\_wbselftag | 需要写self tag | 1. 读self目录miss 2. preferCache | 请求为Get或AcqurireBlock |
| s\_transferput | 当Put miss时，需要准确地将请求转移到源A  （Put的数据在Cache中miss） | 1. 请求为Put 2. 读self目录miss 3. 读client目录miss   （bypassPut） | - |
| w\_putwritten | Put的数据在Cache中hit | 1. 请求为Put 2. !bypassPut | - |
| s\_wbselfdir | Put请求需要修改目录 | 1. 请求为Put 2. 读self目录命中 3. 数据状态不是dirty | - |

条件解释：

\*setProbe：调度a\_do\_probe,s\_probe,w\_probeackfirst, w\_probeacklast,w\_probeack,未命中时（s\_wbselfdir,s\_wbselftag）

\*acquirePermMiss = req.opcode === AcquirePerm && !self\_meta.hit

\*bypassPut = req\_put && !self\_meta.hit && !Cat(clients\_meta.map(\_.hit)).orR()

1. 请求为Put\*

2. 读self目录Miss

3. 读clients目录client 都为INVALID

\*preferCache = (req.preferCache && !bypassPut) || cache\_alias

bypassGet = req.opcode === Get && !preferCache

transmit\_from\_other\_client：自己的缓存中没有该块，但其他client拥有该块

// 当替换数据阵列中的一个块时，并不总是需要发送Release。

// 但只有当状态perm>clientStates'perm或替换一个脏块时才需要发送Release。

 val replace\_need\_release = (self\_meta.state > replace\_clients\_perm) || self\_meta.dirty && isT(self\_meta.state)

* + 1. 子请求与响应参数设置

* + - 1. SourceD(Grant[data])

ReleaseAck与AccessAck[data]的param保留，固定为0

|  |  |  |  |
| --- | --- | --- | --- |
| 请求是Acquire | 请求的参数 | | od.param |
| False | - | | 请求的参数 |
| True | NtoB | promoteT | toT |
| !promoteT | toB |
| BtoT | | toT |
| NtoT | | toT |

\*req\_promoteT = req\_acquire && Mux(//将client提升到T

    self\_meta.hit,//自身命中

    meta\_no\_clients && self\_meta.state === TIP,//client无权限且自身的权限是TIP

    gotT && promoteT\_safe //Grant toT && 提升到T是安全的

  )

* + - 1. SourceC(Release,ProbeAck)

Replace\_Param：（请求不来自B通道时SourceC参数）

|  |  |  |
| --- | --- | --- |
| self状态 | clients中最高的状态 | Replace\_Param |
| BRANCH | INVALID | BtoN |
| BRANCH | BRANCH | BtoB |
| BRANCH | TRUNK | TtoT |
| BRANCH | TIP | TtoT |
| TIP | INVALID | TtoN |
| TIP | BRANCH | TtoB |
| TRUNK | TIP | TtoT |

ProbeAck\_Param：（请求来自B通道时SourceC参数）

|  |  |  |
| --- | --- | --- |
| highest\_perm | probe\_next\_state(highest\_perm, req.param) | ProbeAck\_Param |
| TRUNK | TRUNK | TtoT |
| TIP | TIP | TtoT |
| TRUNK | BRANCH | TtoB |
| TIP | BRANCH | TtoB |
| TRUNK | INVALID | TtoN |
| TIP | INVALID | TtoN |
| BRANCH | BRANCH | BtoB |
| BRANCH | INVALID | BtoN |
| INVALID | INVALID | NtoN |

* + - 1. SourceB(Probe)

1. 若请求来自B通道，Probe\_Param设置为请求的Param
2. 若请求不来自B通道，Probe\_Param按以下逻辑设置：
3. 请求需要T权限才能处理->toN
4. 请求不需要T权限就能处理->toB
   * + 1. SourceA(Acquire)
5. 若不需要执行子任务s\_transferput，Acquire\_Param设置为请求的Param
6. 若需要执行子任务s\_transferput，Acquire\_Param按以下逻辑设置：
7. 请求为Get->NtoB
8. 请求不为Get->acquire\_param

acquire\_param:

L2->请求的param(没有考虑Hint的情况)

L3：

1. 请求为AcquireBlock且请求参数为BtoT->NtoT
2. 请求为put->NtoT
3. else->请求的param
   * 1. 优先级与复位

在读目录结果返回的时候开始置标志位。

子任务调度中优先级c\_schedule > b\_schedule > a\_schedule，并且当请求为Hint且有client命中时不调度a\_schedule。

* + 1. 请求嵌套
       1. abc/bcMSHR可以被嵌套条件

MSHR可以接收嵌套的B请求的条件：B nest A

1. 读目录有效
2. 不需要或已经收到ReleaseAck
3. 不需要或已经收到ProbeAck的最后一拍（避免多次Probe）
4. 不需要将ProbeAck的内容写入储存或进行直通
5. a. 正在等待Grant的第一拍 （暗示请求来自A通道）

b. client目录冲突且probeHelper未处理完成（目录冲突只发生在请求为Acquire时）

MSHR可以接收嵌套的C请求的条件：C nest A 或C nest B

1. 读目录有效
2. 不需要或已经收到ReleaseAck
3. 如下：
4. 正在等待ProbeAck的第一拍
5. 正在等待Grant的第一拍
6. client目录冲突且probeHelper未处理完成（目录冲突只发生在请求为Acquire时）
   * + 1. abc/bcMSHR 让cMSHR ReleaseThrough条件

abcMSHR/bcMSHR让cMSHR Through前提条件：

1. set相同
2. cMSHR中的请求是ReleaseData

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A请求让C请求 Through的条件 | | | | |
| set匹配 | A请求 self是否命中 | tag是否匹配 | way是否匹配 | 可以触发Through 的条件 |
| 是 | 否 | 否 | 否 | 不能触发 |
| 是 | 请求将会在本 层存储block |
| 是 | 否 | 可以触发 |
| 是 | A请求此时容量 冲突向下发送 了release |
| 是 | 否 | 否 | 不能触发 |
| 是 | 请求将会在本 层存储block |
| 是 | - | - |
| 是 | 不能触发 |

\*self命中的前提下若两个请求tag相同选择的一定是同一个way

A请求让C请求 Through的条件：

1. 请求来自于A通道
2. 情况一：A与C请求同一个block但选择了不同的way
3. tag匹配
4. way不匹配
5. self未命中（本层无请求需要的块）

这种情况不Through将导致一个block在set中存了两份

情况二：A与C请求不同的block但选择了相同的way

1. tag不匹配
2. way匹配
3. \*请求将会在本层存储block（包含多个判断，详情请阅读代码）

这种情况不Through将导致C请求要存储的block被A请求的block覆盖，且两个block不同

情况三：A与C请求相同的block且选择了相同的way，但A请求向下发送了release

1. tag匹配
2. way匹配
3. self未命中（本层无请求需要的块）
4. A请求此时容量冲突向下发送了release

这种情况不Through将导致？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| B请求让C请求 Through的条件 | | | | |
| set匹配 | A请求 self是否命中 | tag是否匹配 | way是否匹配 | 可以触发Through 的条件 |
| 是 | 否 | 否 | 否 | 不能触发 |
| 是 | 不能触发 |
| 是 | 否 | 可以触发 |
| 是 | 可以触发 |
| 是 | 否 | 否 | 不能触发 |
| 是 | 可以触发 |
| 是 | - | - |
| 是 | 不能触发 |

B请求让C请求 Through的条件：

1. 请求来自于B
2. 如下：
3. tag匹配且self未命中
4. way匹配且self和tag不同时命中
   * + 1. cMSHR处理Release策略

在cMSHR中处理Release请求的Task 分为三种类型，分别是：

Save：将 Buffer 中的数据项存进 DataStorage；（默认为True）

Through：将 Buffer 中的数据项经过包装直接通过release信号组发送到下层 Cache 或内存；

Drop：丢弃 Buffer 中的数据项。

当接收到Task时，SinkC进入Busy状态，不接收后续任务直到该 Task 处理完成。

执行相应task的逻辑：

1. io\_releaseThrough True:（在c MSHR中使用，被嵌套的abc / b MSHR决定让它进行Release Through）
2. 读目录有效
3. 对MSHR来说该请求来自于C
4. 决定Through还是Drop：
5. 请求的param为TtoN，TtoB或BtoN且其它的client没有这个块：执行release\_through
6. 请求的param为report或TtoB或其它的client有这个块（!release\_through）：执行release\_drop

同时把release\_save设置为False

io\_releaseThrough True例子：

A MSHR 给嵌套C MSHR 的releaseThrough：nest\_c\_tag\_match && !self\_meta.hit && !nest\_c\_way\_match

首先是 tag\_match（再加上嵌套请求 set 相同），那就说明确实是同一个地址，是同一个数据块

然后 A 的 self\_meta 不 hit，说明本层缓存里实际上是没有这个数据块的

最后 !nest\_c\_way\_match。在不命中的情况下，会挑选 set 中的某一路数据进行替换。!nest\_c\_way\_match 的意思是 A MSHR 和 C MSHR 选择了不同的路，如果都让它们把数据保存了，那缓存里就有同一个数据块的两份了，这肯定不行，所以我们就让 C MSHR 直接向下 ReleaseThrough 了

A MSHR 允许被嵌套的条件，就是还没收到下层 Grant 上来的数据。A MSHR 被嵌套卡住，然后 C 直接将这个数据 Release 到下层完成后，A 再恢复执行，取到的就是最新的数据了

* + - 1. abcMSHR让bMSHR probeAckDataThrough条件

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A请求让B请求的带数据回复 Through的条件 | | | | |
| set匹配 | A请求 self是否命中 | tag是否匹配 | way是否匹配 | 可以触发Through 的条件 |
| 是 | 否 | 否 | 否 | 不能触发 |
| 是 | PreferCache 请求不是AcquirePerm |
| 是 | 否 | 不能触发 |
| 是 | 不能触发 |
| 是 | 否 | 否 | 不能触发 |
| 是 | 可以触发 |
| 是 | - | - |
| 是 | 不能触发 |

A请求让B请求的带数据回复 Through的条件：

1. 请求有效
2. set匹配
3. tag不匹配
4. way匹配
5. bcMSHR中的回复是ProbeAckData
6. 请求来自于A
7. PreferCache或self命中
8. AcquirePerm没有miss
   * + 1. bMSHR处理ProbeAckData策略

与releaseThrough类似，在SinkC上也有三个任务处理ProbeAckData，默认都为False分别是

1. probeack\_through
2. drop\_probeack
3. save\_probeack（若请求不来自于B通道且读目录有效则设置为Ture）

要注意的是，这三个任务影响的是数据是否存储在本层中，例如Drop：数据被丢弃，不在本层中存储，但是下层若需要响应，仍需要发送ProbeAckData到下层。

执行条件：

1. 读目录有效
2. 请求来自于B通道
3. 决定执行哪一个任务：
4. probeack\_through:
5. client中有T权限
6. 请求需要ProbeAckData且本层及上层最高权限不是INVALID且self未命中
7. drop\_ probeack：proback\_through为False
8. save\_probeack：
9. probeack\_through为True
10. io\_probeAckDataThrough为False
11. self未命中
12. 请求的参数为toB

对于来自ProbeHelper的B请求处理方式不同。

* + - 1. cMSHR嵌套bcMSHR嵌套abcMSHR

mshr.io\_releaseThrough := Cat(

(abc\_mshr :+ bc\_mshr).map(non\_inclusive).map(\_.io\_c\_status.releaseThrough) ).orR()

此时cMSHR的io\_ReleaseThrough信号由bcMSHR和abcMSHR决定，只有一者的releaseThrough信号为True则cMSHR的信号则为True。

1. L3Cache
   1. 概述

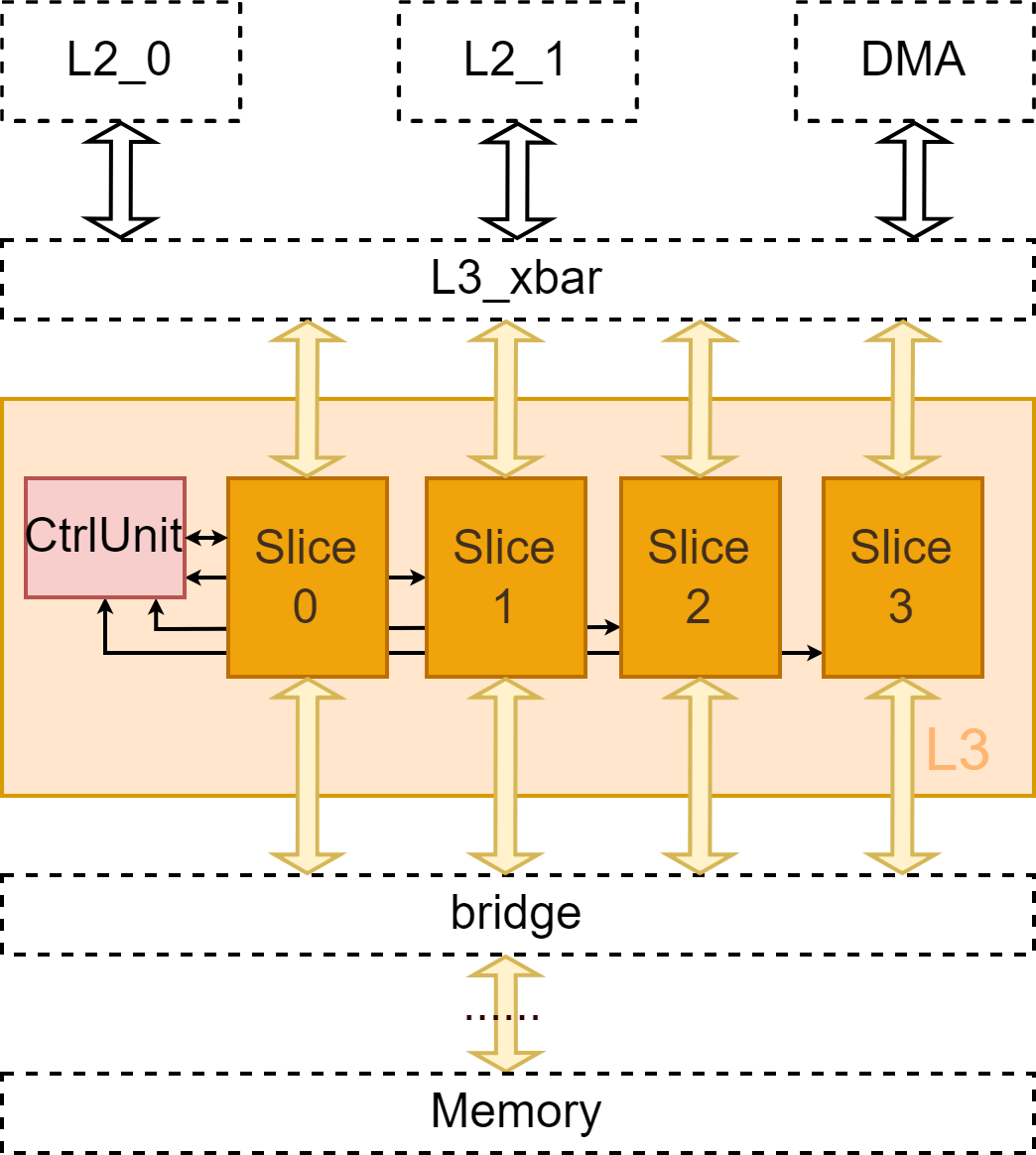


图 HuanCun L3整体框图

L3 HuanCun根据缓存块索引的低位分为4个Slice，在结构上与L2类似，但是L3的连接对象与L2不一致，特别地：L3的上层模块多出了一个不支持TL-C的DMA，而下层模块也是脱离了一致性系统的Memory，所以L3到Memory会有一个TL-C的转换桥来完成信息交互。

* 1. L3与L2结构上的主要区别

L3 HuanCun和L2 HuanCun框架基本一致，区别的地方仅在于：

* 有4个Bank，采用六路组相联
* 没有硬件预取器，及相应的预取逻辑
* L3对PreferCache有特殊处理：若请求为Get或Put，PreferCache直接设置为True
* 有控制单元，用来处理Cache指令

（CtrlUnit和Slice中包含的SliceCtrl都为L3独有，主要用于实现外部对Slice的直接操作以及处理CMO指令）

L3对Memory发送的可能会是TL-C独有的请求，但Memory并不支持TL-C，所以在L3和memory间会有一个转换器（CacheCork）完成消息传递。

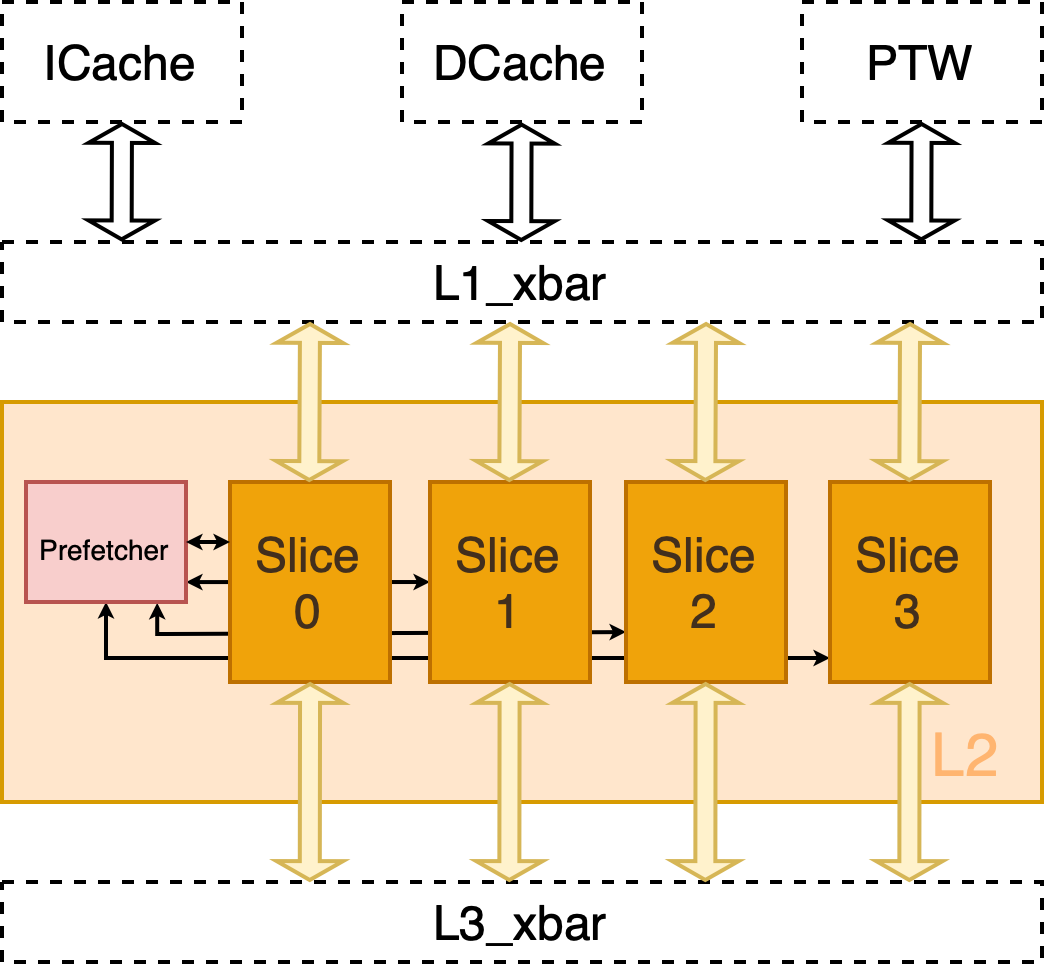
CacheCork

|  |  |  |
| --- | --- | --- |
| 转换前 | 转换后 | 参数 |
| Acquire | Get | - |
| AccessAckData | GrantData | toT |
| toB |
| ReleaseData | PutFullData | - |
| AccessAck | ReleaseAck | - |

\* 在南湖V2中GrantData的参数都是toT

1. Crossbar Arbiter
   1. 概述

当多个master连接到同一slave时需要使用Crossbar进行连接，此时会出现消息优先级问题，因为slave的一个通道在同一时刻只能接收一个消息。例如：ICache和DCache同时向L2发送A通道消息Acquire，此时需要进行仲裁。



关于仲裁部分的逻辑，南湖中TileLink的Crossbar使用的是RockerChip的TLXbar模块，而TLXbar模块里使用了Rocket->tilelink->Arbiter模块的roundRobin策略进行仲裁。

* 1. Arbiter的仲裁策略

Arbiter的仲裁策略一共分三种：

1. lowestIndexFirst（低地址消息优先）
2. highestIndexFirst（高地址消息优先）
3. roundRobin（轮询策略）

在南湖中使用的是roundRobin轮询策略。

1. 香山南湖架构缓存一致性结构

香山南湖微架构如下：

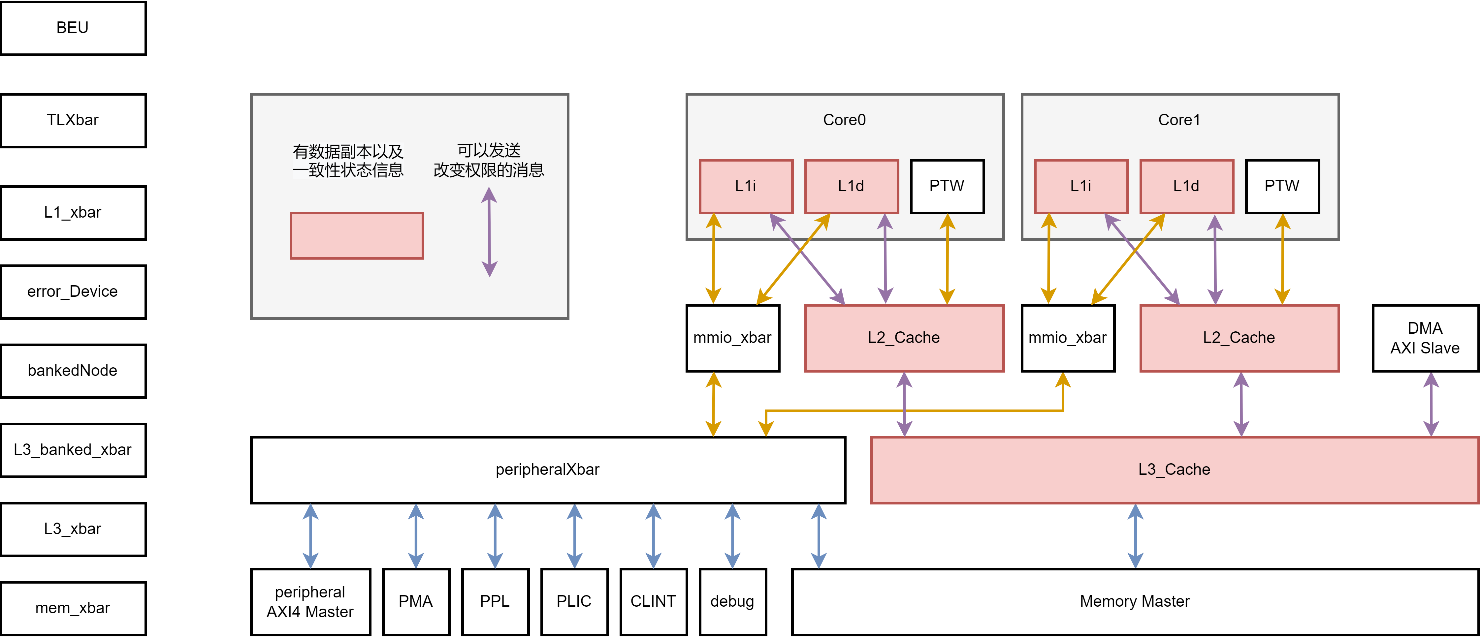


图 消息交互结构

其中标红的模块表示其会存储数据和设计缓存一致性的状态信息，紫色的箭头表示该通道的信息可以改变相应模块的状态信息。

两个L1cache下挂接一个共享L2cache，两个L2下挂接一个共享L3cache。其中L2 L3是 Non-inclusive Cache (inclusive directory, non-inclusive data)

需要注意的问题：

1. 该系统支持的不是标准的TL协议，在L1中没有TRUNK而有一种新的状态Dirty，在L2L3中TRUNK为不可读不可写，A通道额外传输dirty和preferCache两种信息。
2. 该系统是non-inclusive缓存（数据non-inclusive，目录inclusive），在A通道中传输的preferCache信号会影响该策略。
3. L1i只会读取数据，并不会修改数据。
4. DMA不支持TL-C（一致性请求），也不存储一致性信息，但是为了保证系统的缓存一致性，它与L3\_Cache交互时（Put或Get）时也会改变L3以及整个一致性树的状态。
5. L3对Memory发送的可能会是TL-C独有的请求，但Memory并不支持TL-C，所以在L3和memory间会有一个转换器（CacheCork）完成消息传递。
6. 消息流与权限变化实例

下面我们把不涉及缓存一致性的模块省去，以该结构来演示TileLink对缓存的部分实际操作。

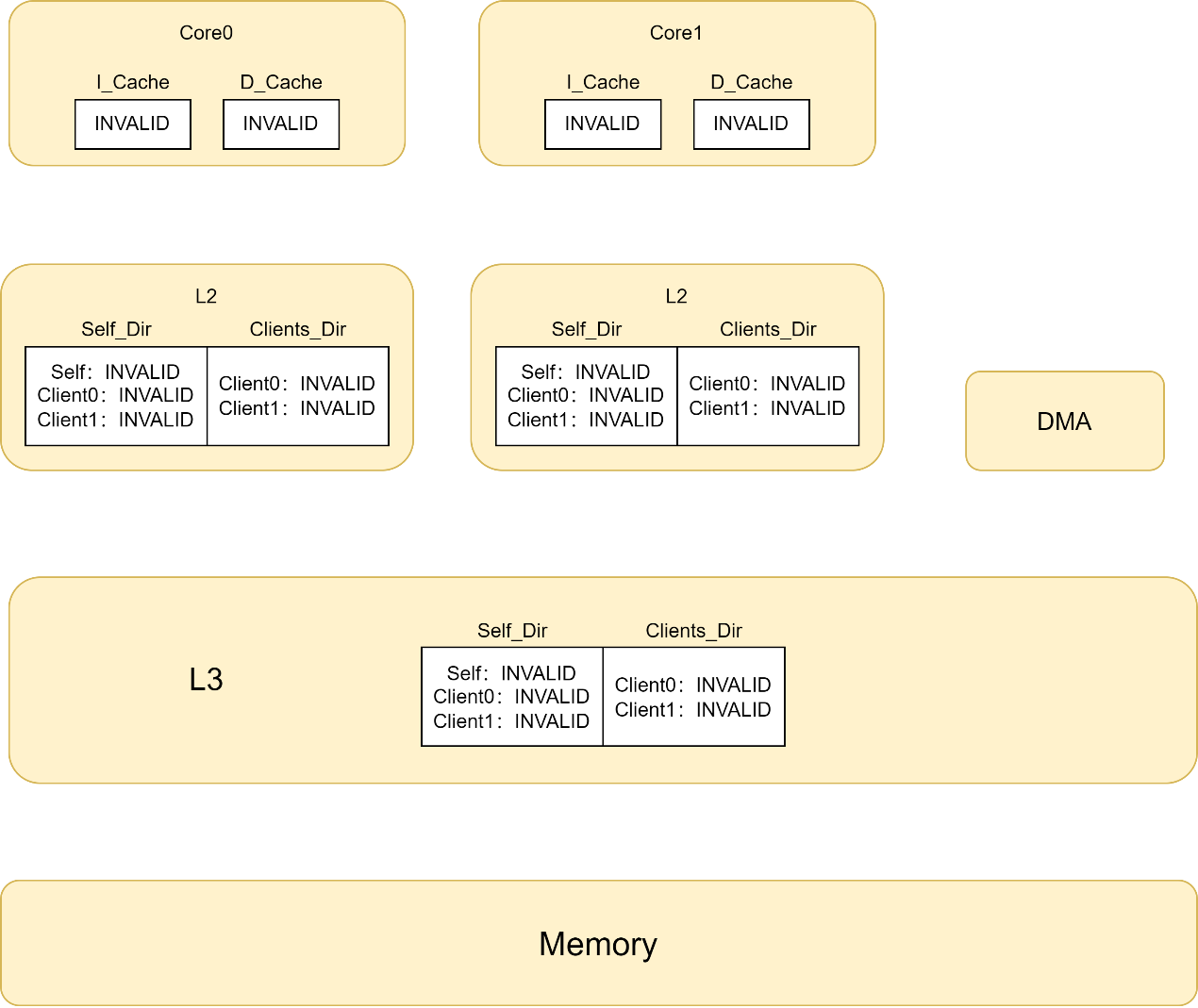


图 缓存一致性结构

\*省略了L3与Memory之间的转换桥

下面将分析以下三种情况：

1. 两个核心进行AcquireBlock

2. L1与L2进行Release

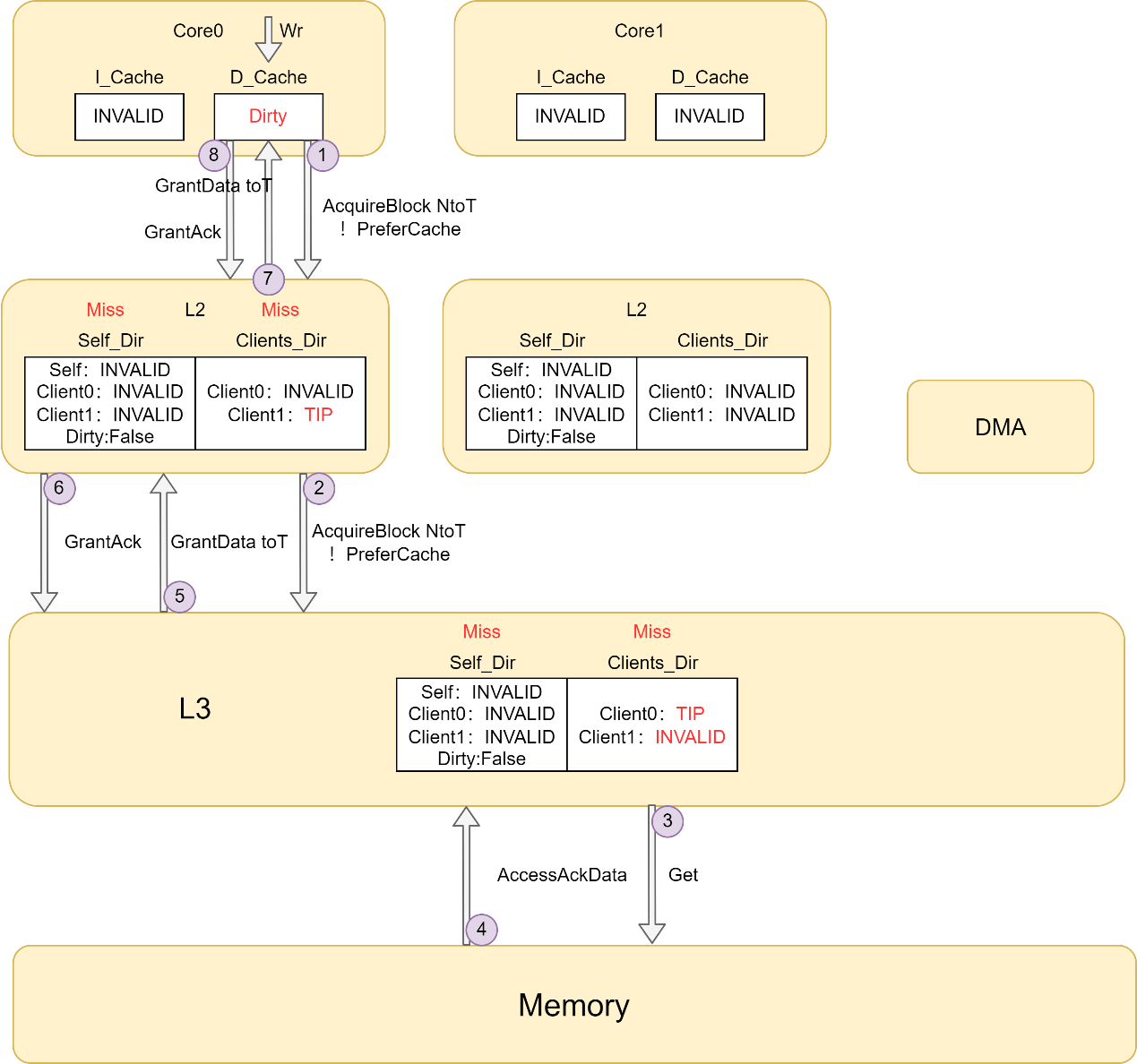
3. DMA Put&Get

* 1. Acquire

情况一：Core0对DCache进行写操作

初始条件：

1. 要写的block在L1L2L3中都没有存储
2. PreferCache为False

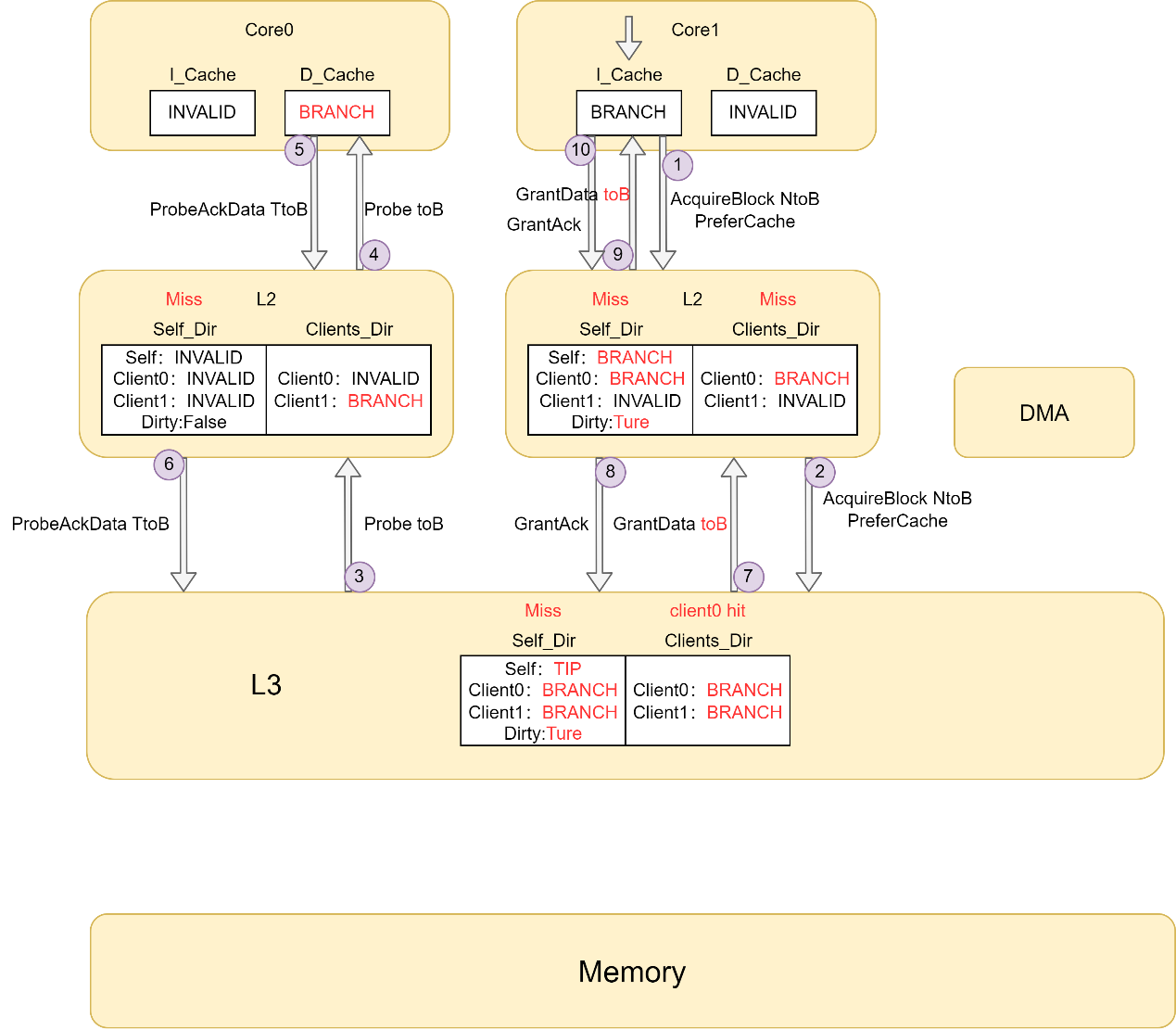


\*红色为更新后的状态以及在Cache这个block是否命中

情况二：Core1对ICache进行读操作

初始条件：

1. PerferCache为True
2. 要读的Block在存储系统中的状态为情况一完成时的状态

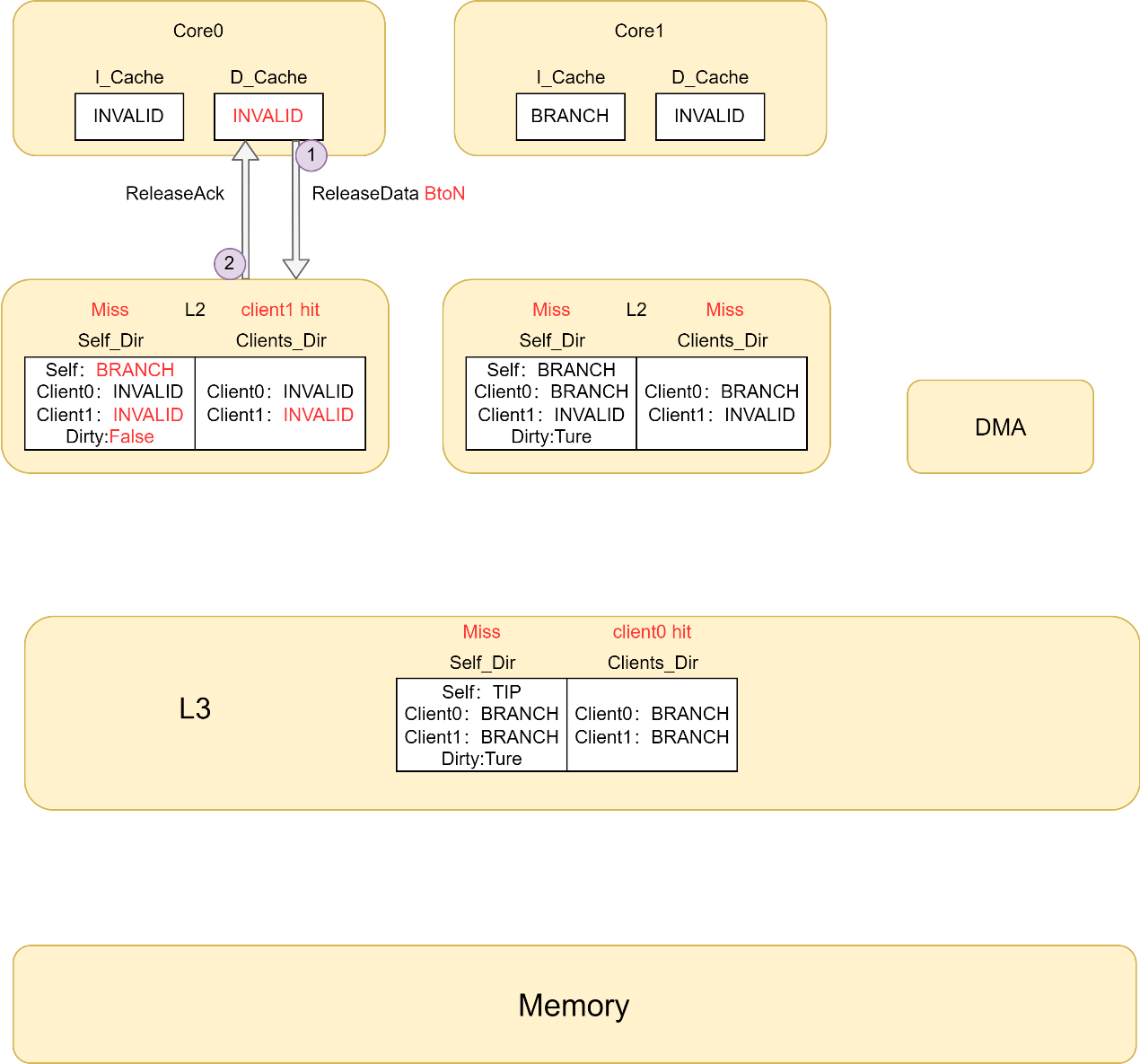


* 1. Release

core0在Writeback Queue中向L2发送Release请求将数据写回

初始条件：

1. 要release的Block在存储系统中的状态为7.1中情况二完成时的状态
2. 没有处理过程中没有嵌套

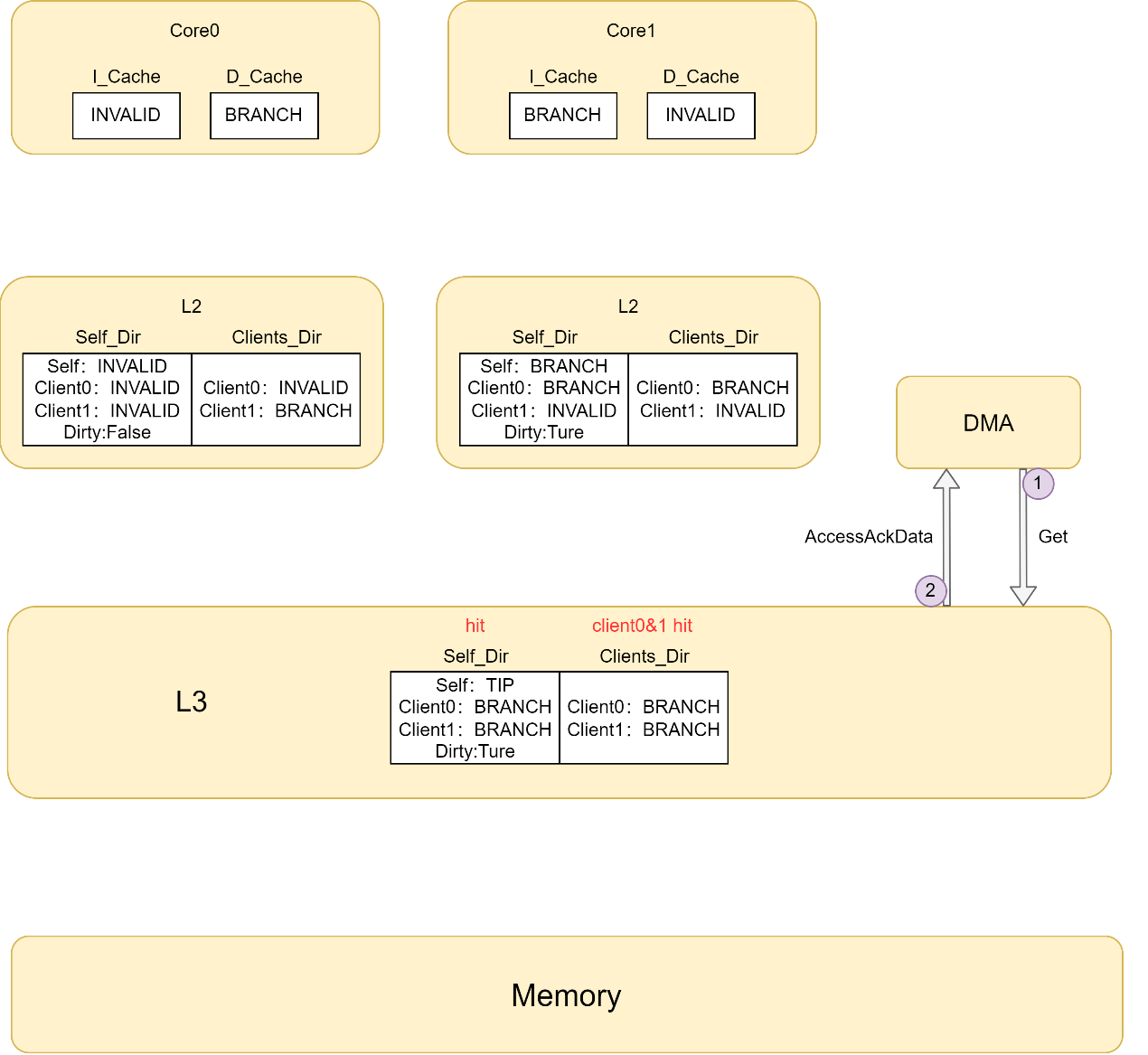


* 1. Put & Get

情况一：DMA向L3发起Get请求

初始条件：

要Get的Block在存储系统中的状态为7.1中情况二完成时的状态



情况二：DMA向L3发起Put请求

初始条件：

要Put的Block在存储系统中的状态为7.1中情况二完成时的状态

