学习手册

THUWC2020 命题组

2019 年 12 月

目录

1 基本概念

1.1 Cache 简介

现代计算机通常采取多层存储的设计，Cache（缓存）是 CPU 与主存之间不可或缺的一级。这是由于计算机程序通常存在很强的局部性（locality），通常包括以下两类：

时间局部性 最近被访问的内容很可能在近期再次被访问

空间局部性 最近被访问的内容附近的内容很可能在近期被访问

利用以上的局部性原理，一个设计良好的 Cache 可以有效地弥补主存在速度上的不足。一般来说，处理器都使用多级的 Cache，每一级的容量逐渐增大，而访问速度相应降低。而无论其所处层级，Cache 的工作原理都是相同的。

1.2 基本假设

在下列的所有叙述以及题目中，除非特别声明，我们都遵循下列假设：

- 涉及的所有地址都是物理地址。
- 计算机的主存以字节编址，大小为 2^N 字节（即物理地址长度为 N 比特）。
- Cache 的总大小为 2^M 字节，且有 $M < N$。
- Cache 中连续存储数据的最小单位（一个**块**，通常也称为一个 Cacheline）大小为 2^K 字节，且 $K < M$。这意味着 Cache 总共有 $P = 2^{M-K}$ 个块，我们将其编号为 $0, \ldots, P-1$；主存总共有 $Q = 2^{N-K}$ 个块，类似编号为 $0, \ldots, Q-1$。Cache 访问主存的最小单位为一个块，即 Cache 每次从主存读写时，数据长度均为连续的一个块，并且地址需要对齐到块的边界。
- 为了简单起见，CPU 产生的访存序列均以块为单位，并且地址保证对齐到块的边界。
- Cache 在初始情况下均为 Invalid 状态。

2 Cache 映射方式

由于 Cache 的容量远小于主存，Cache 中的一个块一定会对应主存中的多个块。这种对应关系称为 Cache 到主存的映射关系。通常来说，Cache 具有下列三种映射方式：

2.1 直接映射（Direct Mapped）

直接映射意味着每一个主存中的块对应的 Cache 中的块是确定的，主存中编号为 q 的块在 Cache 中将被存储在编号为 $q \% P$ 的块中。在这种情况下，无需额外信息即可从主存地址定位到 Cache 中的块。

假设我们有 16 KB 大小的主存和 256 字节大小的 Cache，块大小为 4 字节，即 $N = 14, M = 8, K = 2$。采用直接映射的 Cache 构成如图 1 所示。主存地址此时分为三部分：块内偏移（offset）、对应的 Cache 块编号（index）、标签（tag）。由于一个 Cache 块对应多个主存块，所以需要标签来确定对应 Cache 块内存储的是否为当前地址。

每次访问 Cache 时，只需要根据地址的 index 找到相应的 Cache 块，而后比较 tag，即可确定是否命中。

2.2 全相联（Fully Associative）

全相联意味着每一个主存中的块都可以对应 Cache 中的任意块。图 2 为全相联 Cache 的示例（其余参数与上述一致），可见此时内存地址中不再存在 index 字段，而 tag 字段将与 Cache 中所有块保存的 tag 进行比对，以确定是否命中了某一块。

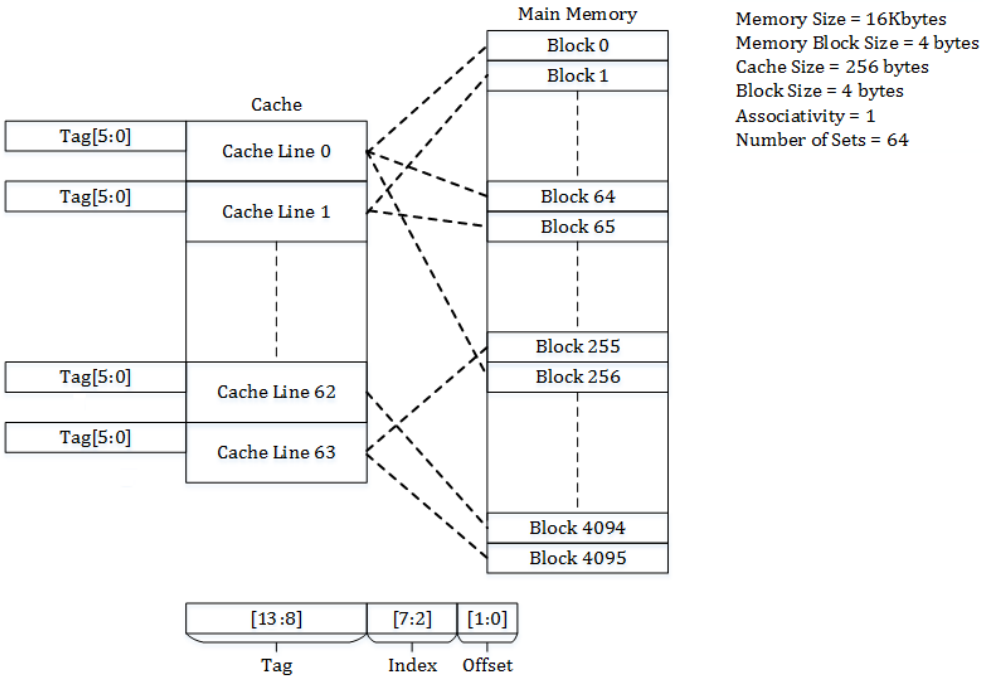

图 1: 直接映射 Cache

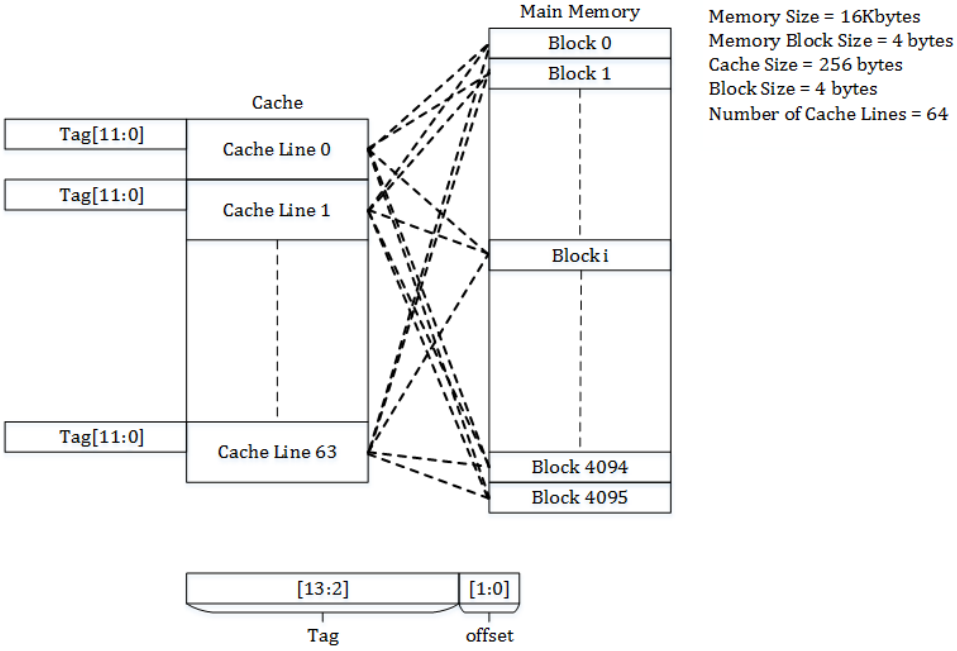

图 2: 全相联 Cache

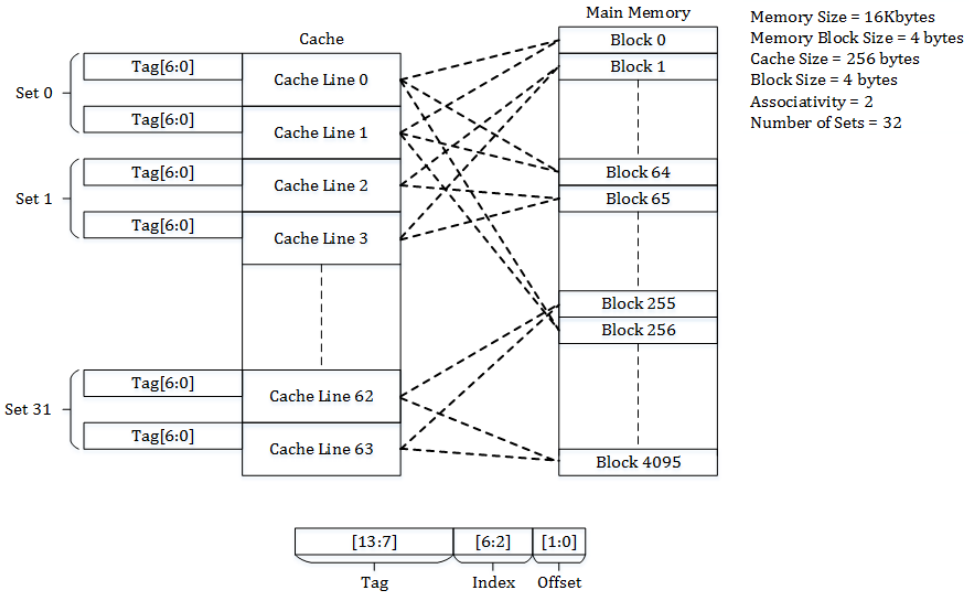

图 3: 2 路组相联 Cache

2.3 组相联（Set Associative）

组相联是介于直接映射与全相联之间的一种映射方式。具体地，Cache 根据块编号被顺序地分为 $S = 2^L$ 组，则每组内有 $W = 2^{M-L}$ 块。Cache 中第 p 块的组号为 p/W（此处表示整除），其在组内的编号为 $p \% W$。主存中的块对于这 S 个组使用直接映射，而在组内的 W 个块则为全相联。为了区分不同的 W，我们通常将其称为 W-路组相联（W-way set associative），W 一般也被称为相连度（associativity）。

图 3 为一个 2 路组相联的 Cache 图示，可见 Cache 的每两块成为一组。访问 Cache 时，首先通过地址的 index 字段找到对应的组，再在组内通过 tag 查询是否命中了某一块。

根据上面的叙述可以知道，直接映射和全相联其实是组相联的两种极端形式：前者的 $S = P, W = 1$，而后者的 $S = 1, W = P$。因此，我们在题目中不再对它们进行特别的区分，而是通过给出 S 的方式来代表映射关系。

3 Cache 读策略

对于来自上级的读请求，Cache 的处理方式很简单，可以直观地描述如下：

1. 根据请求地址，找到对应的组（可能有多个块）；
2. 从地址中提取标签，与组中各块比对（直接映射的情况下可跳过），确定是否命中了组中的某一块；
3. 如果命中（hit），则直接向上级返回结果；
4. 如果缺失（miss），则从主存读取相应块，装入 Cache 中（此时可能发生替换），并向上级返回结果。

4 Cache 写策略

除了数据的读取，Cache 需要处理来自上级的写请求。Cache 的查找过程与读取时一致，而无论是否命中，Cache 都可以使用不同的策略处理写操作。

4.1 写命中策略

当一个写请求命中 Cache 中的块，可以有以下的两种处理策略：

Write Through 在写入 Cache 块时，同时向主存写入该块数据；

Write Back 只写入 Cache 块（并将其标记为脏，即使写入的新数据和原数据相同），在该块被替换出时再写入主存，如果一个块没有被标记为脏，则在被替换出时不会写入主存。

4.2 写缺失策略

当一个写请求未能命中 Cache 中的块（缺失）时，也有两种处理策略：

Write Allocate 先将对应的块载入 Cache 中（相当于一次读缺失），再对 Cache 进行写（此时按照写命中策略进行，不算一次访问）；

No-write Allocate 不将对应的块载入 Cache 中，直接对主存的对应块进行写操作。

上述两个方面的策略可以任意组合，形成四种不同的策略组合。而在实践中，我们通常会使用下面的两种组合：

- Write Through + No-write Allocate
- Write Back + Write Allocate

5 Cache 替换算法

由于 Cache 的容量受限，我们需要经常将块替换出 Cache，并换入新的块，以期取得最好的缓存效果。被替换的块（一般称为 victim）的选择，事实上是决定 Cache 效率的关键因素。在理论上，只有选择在最长的时间间隔后将被使用的块进行替换，才能取得最好的效果。但我们不可能预知程序将来的行为，因此只能尽量逼近这一最优解。在历史上，出现了一批有代表性的替换算法。

需要注意的是，并不存在所谓的“银弹”，没有替换算法是万能的。大部分算法的目的是为了良好地处理通常的情况，而也有部分算法专为特定情境而设计。

5.1 伪随机（Pseudo Random）算法

伪随机（Pseudo Random）算法不使用任何过去的信息，首先选择 Invalid 块中编号最小的，如果不存在 Invalid 块则随机地选择一个。在本题中，我们使用下列的线性同余伪随机数发生器来产生你所需要的伪随机数。

```c
#include <stdint.h>
static uint64_t next = 1;
uint32_t my_rand(void) {
    next = next * 1103515245 + 12345;
    return (uint32_t)(next/65536) % 32768;
}
```

该伪随机函数生成的前五个数是 $16838, 5758, 10113, 17515, 31051$。你应该将得到的伪随机数对于每组中的块数 W 取模，以确定需要替换出的块。请注意，当且仅当需要进行替换时，你才应该调用这一发生器。否则，你获得的伪随机数序列将与标准答案中使用的存在偏差，从而导致错误。

事实上，尽管看似随意，在大多数情况下，伪随机替换算法都能够取得较好的效果。这是因为其他所有的算法都或多或少地使用了历史访存序列的信息，从而不可避免地在某些特定序列上产生糟糕的表现；而伪随机算法不使用过去信息的特征让它避开了这一问题。

5.2 先入先出（FIFO）算法

先入先出（First In First Out）算法的思路很简单，它首先选择 Invalid 块中编号最小的，如果不存在则选择当前组中最早进入 Cache 的块进行替换。

5.3 最不经常使用（LFU）算法

最不经常使用（Least Frequently Used）算法的思路利用了局部性原理的否命题：如果某个块最近一直没有被访问，则它在将来很可能也不会被访问。因此，它记录 Cache 一组中每一个块的访问次数（读或写均记为一次访问，由装入带来的也计算在内，下面均相同），并在替换时首先选择 Invalid 块中编号最小的，如果不存在则选择访问次数最少的；如果有两块访问次数相同，则优先选择在组内编号较小的。

5.4 最近最少使用（LRU）算法

最近最少使用 (Least Recently Used) 算法的思路与 LFU 算法类似，它首先选择 Invalid 块中编号最小的，如果不存在则选择组中上次被访问时间最早的一块进行替换，此时选择总是唯一的。

LRU 思路直观，效果一般也较好，因此是目前在软件中使用最广泛的缓存算法。但是在硬件上，LRU 的实现存在一定的困难。并且对于一些循环的访存序列，LRU 将会遭遇严重的污染问题（考虑 $W = 4$，初始时 Cache 为空，访存序列为第 $0, 1, 2, 3, 4$ 块的循环，此时缓存命中率几乎为 0）。

5.5 最近最多使用（MRU）算法

最近最多使用（Most Recently Used）正是为了解决 LRU 算法上述的污染问题而产生的，它的选择方式与 LRU 完全相反，即在没有 Invalid 块时选择组中上次被访问时间最晚的一块进行替换。

5.6 伪最近最少使用（PLRU）算法

为了解决 LRU 算法在硬件上实现困难的问题，一系列的伪 LRU（Psuedo LRU）算法应运而生。其中较为简明的一种被称为 Tree-PLRU。

这一算法的原理如下：考虑一颗以组中的 W 个块为叶子结点的二叉搜索树，树中每个非叶子节点有一比特的标记，记录从该节点出发，找到待替换元素需要前进的方向（我们总是记 0 为左，1 为右，并且初始时所有节点的值均为 0）。为了找到 Tree-PLRU 算法需要替换的节点，只要从根节点出发，根据标记确定方向，不断遍历直到叶子结点即可。而在每次访问一个块时，先找到对应的叶子结点，遍历从根结点到叶子结点的路径，并将途径结点的标记更改为与遍历路径相反的方向即可（如果本来就与路径相反，标记不会发生变化）。直观地理解，每次访问一个块时，都会将树上的标记修改为更利于它自身留在 Cache 中的方向。

图 4 是 Tree -PLRU 算法的一个示例。右图为 ABCD 被依次访问后的状态；再进行了对 E 的访问后，原本的 A 被替换，路径上的节点也被相应更新；下一个 PLRU 节点即为 B，这与 LRU 的结果是相同的。

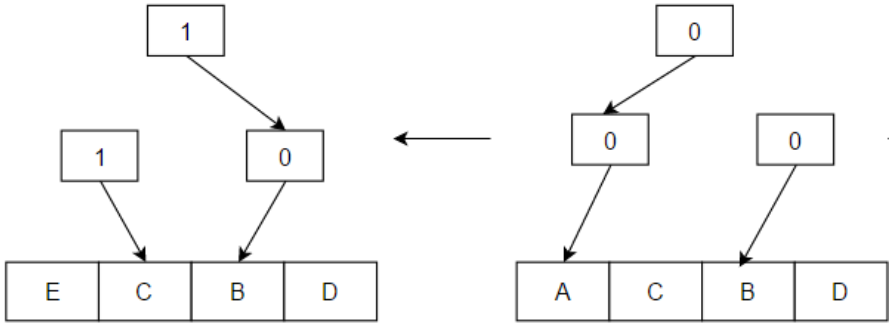

图 4: PLRU 替换算法示例

5.7 低访问相近集合（LIRS）算法

上述大多数算法的共同点为，它们并没有利用同一个块临近两次的访问信息，或者说不能应对突发“冷数据”访问将“热数据”挤占出 Cache 的问题。直观上，如果一个块两次被访问的间隔很短，那么它应该是“热”的；即使在一段时间内它没有被访问，也不应该被立刻从 Cache 中移除。低访问相近集合（Low Inter-reference Recency Set）替换算法正是引入了这一思想。它使用两个指标，分别称为 IRR(Inter-Reference Recency) 和 R(Recency)，来对每一个块的访问进行追踪，并使用一个栈和一个队列来管理 Cache 中的块。我们不会提供 LIRS 算法的具体实现，你需要阅读我们提供的论文 *LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance* 中的第三节 `THE LIRS ALGORITHM` 的内容，来完成此算法的实现。

在实现时，我们遵循以下规定：

- 参数 $L_{hirs} = \text{floor}(\log_2 \text{L})$；
- 当存在 Invalid 块时先选取编号最小的 Invalid 块被替换。

实验表明，LIRS 算法在很多情况下都有优于 LRU 算法的表现；但由于其复杂性，这一算法并不适合直接在硬件上实现。MySQL 数据库目前正是采用了 LIRS 替换算法。

6 Cache 一致性协议

上述讨论的所有内容，都只适用于单 CPU 的情况；而目前，多核 CPU 已经成为市面上的主流。通常，每个 CPU 核心都具有自己独立的 Cache，它们缓存的内容也各自不同。如何在这些 Cache 中维护内容的一致性（coherence），是计算机系统正确工作需要保证的。其中最为重要的一点是，如何保证某个 CPU 核心写入主存的数据能够正确地被其他 CPU 读取到（而非获得 Cache 中被修改前的数据）。许多缓存一致性协议应运而生，其中最著名、最为广泛应用的是伊利诺伊大学提出的 **MESI 协议**，我们将对其进行简单的介绍。

由于 MESI 协议只关注多个 Cache 中对应到主存同一个块的 cacheline 的状态，因此在下面我们不妨可以假设每个 Cache 都只有一个块，当前都对应到主存中的同一个块。事实上，一致性协议与 Cache 的其他部分都是独立工作的：当一个块被载入 Cache 时，其它 Cache 中与它对应地址相同的块共同构成了执行 MESI 协议的一个**组**；而当一个块被从 Cache 中换出时，它就从这一组中脱离了。在一组中，每个 Cacheline 在每一时刻都处于以下四种状态之一（这正是 MESI 协议得名的原因）；

Invalid Cacheline 中的内容是无效的（如已经在别的 Cache 中被修改，或者没有载入）。

Exclusive 此 Cacheline 的内容只在当前 Cache 中并且是干净的（clean，与主存相同）。

Shared Cacheline 的内容也存在于其它 Cache 中且是干净的。

Modified 此 Cacheline 的内容只在当前 Cache 中并且是脏的（dirty，与主存不同）

6.1 Cache 事件

这些状态的转换由不同的事件进行驱动，而事件分为两类，分别是来自 CPU 和来自总线的事件。如图 5 所示，每个 Cache 只能接收到来自于其上级 CPU 的事件，能够接收到所有组件向总线发出的事件。总线是连接各个 Cache 和 Memory 的公共通讯通路，每一个 Cache 发出的事件都可以被其他 Cache 和 Memory 收到。

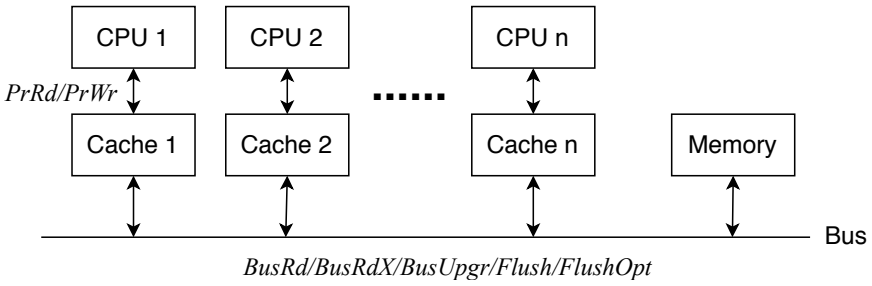

图 5: 典型的多核 CPU 结构

来自 CPU 的事件共两类：

PrRd 处理器请求读此 Cacheline

PrWr 处理器请求写此 Cacheline

来自总线的请求共五类：

BusRd 其他处理器请求读一个 Cache 块

BusRdX 其他处理器请求写一个该处理器不拥有的 Cache 块

BusUpgr 其他处理器请求写一个该处理器拥有的 Cache 块

Flush 回写 Cache 块到主存（可能由处理器或外部产生）

FlushOpt 某个 Cache 块被发到总线以发送给另外一个处理器（Cache 到 Cache 的复制）

6.2 Cache 状态转移

由这些事件驱动，组中的 Cache 块以 6 所示的方式进行状态转换（图中并没有完整标出）。图中每一条边的源节点为当前状态，边上有形如 A/B 的标记，左侧（A）为接收到的事件，右侧（B）为将会发送的事件，目标节点为转移到的状态。例如 M 点的黑边表示如果接受到 PrRd 或者 PrWr ，那么状态不会变化，也不会发送事件。

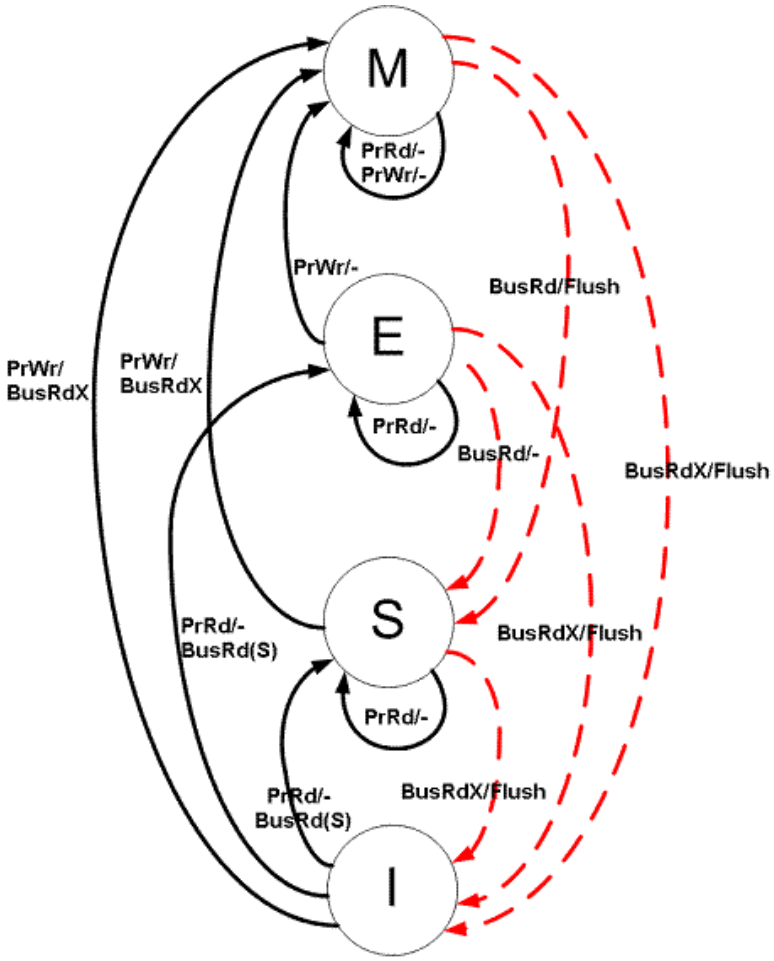

图 6: MESI 协议状态转换图（图中的 Flush 表示的是 FlushOpt，黑线表示对来自 CPU 请求的处理，红线表示对来自总线的请求的处理，如果与下表出现不一致以下表为准）

所有 Cache 块起初都为 Invalid 状态。表 1 和 2 中分别列举了 Cache 收到来自 CPU 和总线事件时的响应和状态转移，与图 6 的内容是一致的。表中没有列出一些在正常工作时不可能存在的请求处理（如在 Exclusive 状态下，不可能接收到 BusUpgr 事件）。需要注意，从主存读取数据只会在 Invalid 状态下发生，而写回主存只会在 Modified 状态下发生。事实上，按照 MESI 协议设计的 Cache 使用的是 Write Allocate 和 Write Back 的写策略。

表 3 为一个完整的示例，演示 MESI 协议的工作状态转换和产生的事件。假设有三个 CPU 和对应的 Cache，每个请求都对于同一个内存地址发出。表中每行列举了响应过程中总线上各个 Cache 发出的请求（按照发出的顺序列出，括号内为进行相应请求的 Cache 编号）、Cache 向内存发出的请求（同样给出编号）以及完成请求后各个 Cache 块的状态。

举例说明，对当前状态为 Invalid 并且收到来自 CPU 的 PrRd 事件的处理逻辑是：此时我对其他 Cache 的状态并不清楚，所以首先在总线上发送 BusRd 信号表示我要读数据，如果有其他 Cache 发出了 FlushOpt （携带 Cache 块内容），那自己就把数据记录下来，然后状态变为 S，因为此时有多个 Cache 共享了这一块内容；如果没有其他 Cache 响应，说明目前没有 Cache 有这一块的数据，则选择从主存获取数据，然后状态变为 E ，因为此时只有一个 Cache 持有这一块内容。

表 1: 来自 CPU 的请求处理

当前状态	事件	响应	下一状态
Invalid	PrRd	向总线发出 BusRd 信号，等待其他 Cache 的回复。 如果有 Cache 存在有效副本，转换为 S，从某个 Cache 获取数据； 如果没有 Cache 存在有效副本，转换为 E，从主存获取数据。	S/E
	PrWr	向总线发出 BusRdX 信号，等待其他 Cache 的回复。 如果有 Cache 存在有效副本，从某个 Cache 获取数据； 如果没有 Cache 存在有效副本，从主存获取数据。 向 Cache 中写入修改后的值。	M
Exclusive	PrRd	直接返回值。	E
	PrWr	向 Cache 中写入修改后的值。	M
Shared	PrRd	直接返回值。	S
	PrWr	向总线发出 BusUpgr 信号，向 Cache 中写入修改后的值。	M
Modified	PrRd	直接返回值。	M
	PrWr	向 Cache 中写入修改后的值。	

表 2: 来自总线的请求处理（其余没有列出的情况下状态不会转移）

当前状态	事件	响应	下一状态
Invalid	任意	无。	I
Exclusive/ Shared	BusRd	向总线发出 FlushOpt 事件（包含 Cache 内容）。	S
	BusRdX	若多个 Cache 同时收到，则只有编号最小的进行发送。	I
Shared	BusUpgr	无。	I
Modified	BusRd	向总线发出 FlushOpt 事件（包含 Cache 块内容）。	S
	BusRdX	向主存写回 Cache 块内容。	I
	Flush	向主存写回 Cache 块内容。	E

表 3: MESI 协议工作示例

CPU	请求	C1	C2	C3	产生的总线请求	内存请求
		I	I	I		
1	R	E	I	I	BusRd(C1)	Read(C1)
1	W	M	I	I		
3	R	S	I	S	BusRd(C3), FlushOpt(C1)	Write(C1)
3	W	I	I	M	BusUpgr(C3)	
1	R	S	I	S	BusRd(C1), FlushOpt(C3)	Write(C3)
3	R	S	I	S		
2	W	I	M	I	BusRdX(C2), FlushOpt(C1)	
1	W	M	I	I	BusRdX(C1), FlushOpt(C2)	Write(C2)