Memory Barriers:a Hardware View for Software Hackers

MRAS Team

Distributed & Embedded System Lab
Lanzhou University

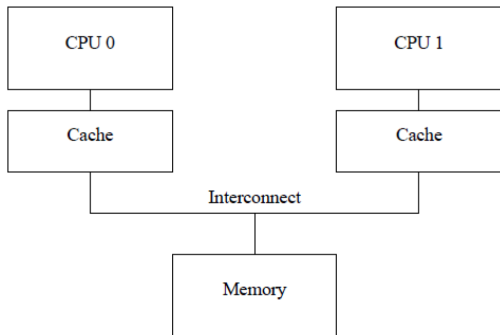
2014 年 8 月 29 日

要点

- Cache Structure
- Cache-Coherence Protocols
- Improvement in Cache Structure
- Read & Write Memory Barriers
- Invalidate Queues
- Summary of Memory Ordering(Specific CPU in x86)

Cache Structure

由于现代 CPU 指令执行速度十倍于从 Memory 中取数据。为了弥补 CPU 与 Memory 速度差距，在 CPU 与 Memory 中间增加了 Cache。



Cache Structure

现代 Cache 结构采用 N 路组相联（N-way set associative）的方式。一个内存地址对应一个 cache line。

	Way 0	Way 1
0x0	0x12345000	
0x1	0x12345100	
0x2	0x12345200	
0x3	0x12345300	
0x4	0x12345400	
0x5	0x12345500	
0x6	0x12345600	
0x7	0x12345700	
0x8	0x12345800	
0x9	0x12345900	
0xA	0x12345A00	
0xB	0x12345B00	
0xC	0x12345C00	
0xD	0x12345D00	
0xE	0x12345E00	0x43210E00
0xF		

Figure 2: CPU Cache Structure

Cache Structure

cache miss – 当给定 CPU 第一次访问某数据时，数据不在 CPU 的缓存中，这是一个"cache miss"

capacity miss – 当运行一段时间后，CPU 的缓存被填满。后续的 miss 都会需要从已满的缓存中剔除一项来为新获取的项腾出空间。

associativity miss – 如果一个已经被剔除的项又被访问了，又回引发一个 cache miss.

write miss – 一个 CPU 在写数据之前需要其他 CPU 的失效操作（validation）。当这个操作完成后，CPU 就可以安全的修改此项的数据。

communication miss – 稍后，如果另一个 CPU 想要试图访问数据项，它会引发一个 cache miss。

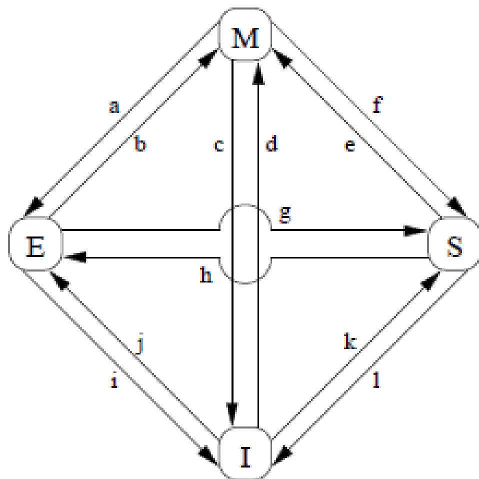
Cache-Coherence Protocols

为了使得 cache-line 在多处理器中的一致性，我们可以使用 Cache-Coherence Protocols。

虽然这个协议包含了数十种状态，但是我们只需要关注 4 个状态：

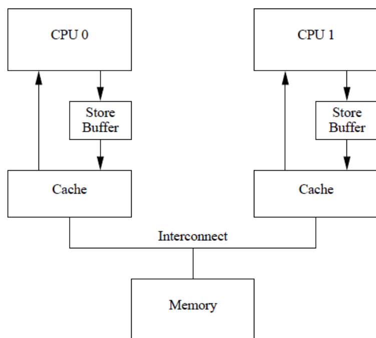
- M – 已修改状态"Modify"，此时只有该 CPU 的 cache 包含着最新的数据。
- E – "Exclusive" 状态与"Modified" 很接近，惟一的区别是 cache line 还未被某对应 CPU 修改。
- S – 一个在"Shared" 状态下的 cache line 可能在其他 CPU 中被复制最少一次。
- I – 在"Invalid" 状态下的 line 是空的，没有数据。

MESI State Diagram



Improvement in Cache Structure

cache 的引入一定程度上缓解了 cpus 与 memory 速度不匹配的问题，MESI 解决了 cache 数据一致性的问题。但是这又出现了新的问题，cpu 之间通信存在延迟，所以在 cpu 与 cache 中间加入了 store buffers。

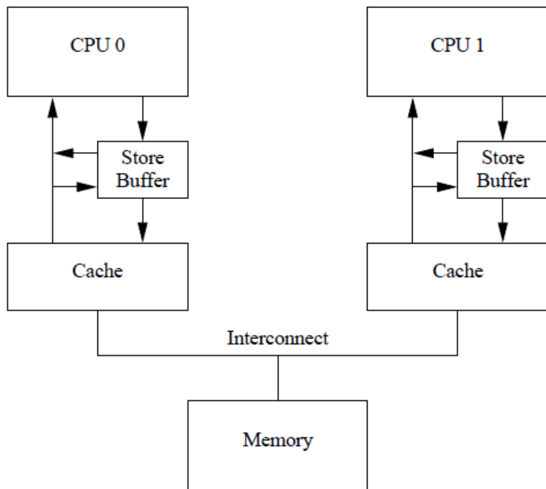


Improvement in Cache Structure

加入了 store buffers，解决了 Stalls, 但是又引入了新的问题：store buffers 与 cache 数据的不一致性。

从硬件角度来看，这种问题是无法解决的，因此引入了 Store Forwarding：也就是说一个特定 CPU 的 store 直接转给它随后的加载，而不通过 cache。这样避免了 cpu 取 cache 数据与 store buffer 数据的问题。

Improvement in Cache Structure



Read & Write Memory Barriers

void foo(void)//CPU0 b

a = 1;

b = 1;

void bar(void)//CPU1 a

while (b == 0) continue;

assert(a == 1);

CPU1 执行 “assert(a == 1);”，因为 a 本来就在 CPU1-cache 中，而且值为 0，所断言为假。

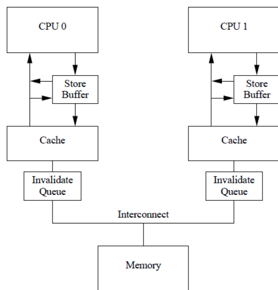
CPU1 收到 read invalidate 消息，将并将包含 a 的 cache-line 传递给 CPU0，然后标记 cache-line 为 invalid。但是已经太晚了。

Read & Write Memory Barriers

添加的 memory barrier 的 smp_mb() 指令可以使得 CPU 在执行 smp_mb() 后的 store 操作前刷新 store-buffer。以上面的程序为例，增加 memory barrier 之后，就可以保证在执行 b=1 的时候 CPU0-store-buffer 中的 a 已经刷新到 cache 中了，所以此时 CPU1-cache 中的 a 必然已经标记为 invalid。对于 CPU1 中执行的代码，则可以保证当 b==0 为假时，a 已经不在 CPU1-cache 中，从而必须从 CPU0-cache 读取，从而得到新值 “1。

Invalidate Queues

由于 store buffers 非常小，cpu 执行几个 store 操作就会把 buffer 填满, 这时候 CPU 必须等待 invalidation ACK 消息，来释放缓冲区空间——invalidation ACK 消息的记录会同步到 cache 中，并从 store buffer 中移除。所以我们在这里引入 Invalidate Queues。



Invalidate Queues 存在的问题

有“Invalidate Queues”的 CPU 会在第一时间将 Invalidate 消息加入队列，并回复 Invalidate Ack，而不用管此时 cache line 是否已被移除。CPU 在发送回复之前，CPU 需要参考当前的队列：如果对应的 cache line 已经在队列中，CPU 不能立即发送回复，而是需要等待这一项被处理后才能回复。

这就导致一个问题：当 CPU 排队某个 invalidate 消息后，在它还没有处理这个消息之前，就再次读取该消息对应的数据了，该数据此时本应该已经失效的。

Invalidate Queues & smp_mb()

解决此类问题的关键就是加入 Read & Write Memory Barriers。所以当一个 CPU 执行 memory barrier 时会将失效队列中的所有项都做标记，并强制 memory barrier 后续的所有加载都要等到标记项被处理后才执行。

Memory barrier 还可以细分为“write memorybarrier(wmb)”和“read memory barrier(rmb)”。

- rmb 只处理 Invalidate Queues，从而只限制有 rmb 的 CPU 的 load 操作。
- wmb 只处理 store buffer，从而只限制有 wmb 的 CPU 的 store 操作。
- 完整的 mb 对 Invalidate Queues 和 store buffer 都做标记，所以对于 CPU 的 load 和 store 操作都限制。

Invalidate Queues & smp_mb()

比如在 Ordering-Hostile Architecture 中，经常会遇到 Invalidate Queues 满，某个 cpu 会在 load 其他 node 数据前获取旧值，导致断言失败。

所以使用 smp_mb() 要成对使用，类似于临界区的概念。

但是总体来说：smp_mb() 屏障之后的某一个访存操作已经完成, 则屏障之前的所有访存操作必定都已经完成。

Summary of Memory Ordering(Specific CPU in x86)

我们在 SMP 环境下，我们不能假定：

- 一个 CPU 能够按顺序看到另一个 CPU 的访存效果。
- 一些与 CPU 相关的硬件会按顺序访存。

Read & Write Memory Barriers 会间接影响其他系统中的 CPU 或硬件设备。从而做到代码顺序局部乱序，总体有序。

Summary of Memory Ordering(Specific CPU in x86)

Memory ordering in some architectures[2][3]

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	zSerie
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	Y

Summary of Memory Ordering(Specific CPU in x86)

x86 CPUs 优化后允许局部 load 乱序，store 指令在 load 指令以后。

在 Intel 较新的 cpu 中，存在一个全局的顺序用来 store，但是 cpus 还是优先看到自己 store 的数据。这其中涉及到 store buffers 的硬件优化。cpus 之间存在 transitivity，也就是说如果 CPU0 可以看到 CPU1 的 store，那么就可以保证 CPU0 看到 CPU1 所有的 store。软件也许会使用原子操作来实现硬件优化，虽然时间开销可能是昂贵的。在一些稍老的 x86 cpu 中，会存在 store 乱序，这样的 cpu 需要使用 lock;addl 指令。

Summary of Memory Ordering(Specific CPU in x86)

在内核开发中，如果没有 memory barriers 指令。CPUS 与 Compilers 可以随意读取 memory，这会导致 kernel crash，甚至损坏硬件。

但是对于内核开发者并不用担心 memory barriers，只要在代码中严格使用 locking primitives（spinlocks，reader-writer lockers，semaphores，RCU....），其中这些原语的实现与平台相关。

THX !