内存屏障（Memory Barriers）

原文出处： Martin Thompson　译文出处：coderbee

在这篇文章里，我将讨论并发编程里最基础的技术——内存屏障（Memory Barriers），就是它让一个处理器内的内存状态对其他处理器可见。

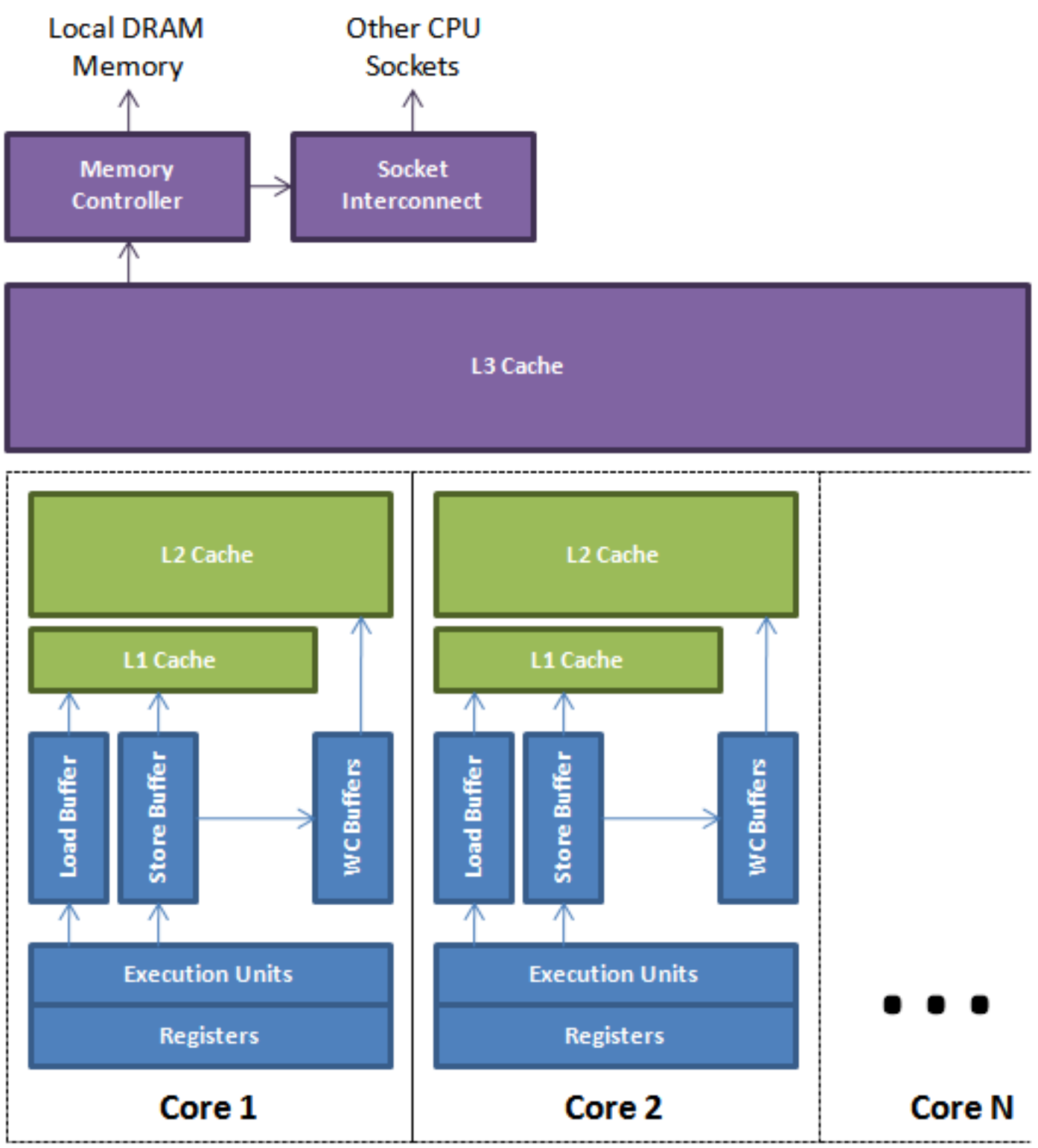
CPU 使用了很多技术去尝试和适应这样的事实：CPU 执行单元的性能已远远超出主内存性能。在我的《Writing Combining》文章，我只是谈及其中一种技术。CPU 使用的用来隐藏内存延迟的最普通技术是管线化指令，然后付出巨大努力和资源去尝试重排序这些管线来最小化缓存不命中的有关拖延。

当一个程序执行的时候，它不在乎，如果重排序后的指令提供了一样的最终结果。例如，在一个循环内，如果循环内没有操作使用循环计算器，循环计数器什么时候更新是不在乎的。编译器和 CPU 自由地重排序指令来最大化地利用 CPU，直到下一次迭代即将开始时才更新它（循环计数器）。也可能，在一个循环的执行过程中，这个变量可能存储在一个寄存器里，永远不会推到缓存或主内存，因此，它对其它 CPU 永远不可见。

CPU 核包含多个执行单元。例如，一个现代的Intel CPU 包含6个执行单元，可以做一组数学，条件逻辑和内存操作的组合。每个执行单元可以做这些任务的组合。这些执行单元并行地操作，允许指令并行地执行。如果从其它 CPU 来观察，这引入了程序顺序的另一层不确定性。

最终，但缓存不命中发生时，现代 CPU 可以根据内存加载的结果做一个假设，然后基于这个假设继续执行直至实际数据的加载完成。

提供“程序顺序”保留了 CPU 和编译器自由地做它们认为可以提升性能的事情



加载（load）和存储（store）到缓存和主内存是被缓冲和重排序的，使用加载（load），存储（store），和写组合（writing-combining）缓存。这些缓存是关联的队列，允许快速查找。这种查找是必须的，当一个稍后的加载需要读取一个之前存储的、还没有到达缓存的值时。上图描绘了现代多核CPU 的简化视图。它显示了执行单元如何使用本地寄存器和缓存来管理内存，与缓存子系统来回传送。

在多线程环境下，需要采用一些技术来让程序结果及时可见。我不会在这篇文章里涉及缓存一致性。仅仅假设一旦内存被推到缓存，然后有一个协议消息将发生，以确保所有共享数据的缓存是一致的。这种使内存对处理器核可见的技术被称为内存屏障或栅栏。

内存屏障提供了两种属性。首先，它们保留了外部可见的程序顺序，通过确保所有的、屏障两侧的指令表现出正确的程序顺序，如果从其他CPU观察。

第二，它们使内存可见，通过确保数据传播到缓存子系统。

内存屏障是一个复杂的主题。它们在不同的 CPU 架构上的实现是非常不同的。Intel CPU 有一个关联的强内存模型。本篇将以 x86 CPU 为基础讲解。

内存屏障（store barrier）

内存屏障，在x86 上是”sfence”指令，强迫所有的、在屏障指令之前的 存储指令在屏障以前发生，并且让 store buffers 刷新到发布这个指令的 CPU cache。这将使程序状态对其他 CPU 可见，这样，如果需要它们可以对它做出响应。一个实际的好例子是下面的、简化的、来自Disruptor的类BatchEventProcessor。当sequence被更新后，其他消费者和生产者知道这个消费者的进展，并进行适当的响应。所有在屏障之前对内存的更新现在都可见了

private volatile long sequence =

```
1
2
3  private volatile long sequence = RingBuffer.INITIAL_CURSOR_VALUE;
4  // from inside the run() method
5  T event = null;
6  long nextSequence = sequence.get() + 1L;
7  while (running)
8  {
9     try
10    {
11       // 译注：barrier 会读取其他sequence 的值，所以这里面有个 load barrier 指令。
12       final long availableSequence = barrier.waitFor(nextSequence);
13
```

```
14          while (nextSequence <= availableSequence)
15          {
16              event = ringBuffer.get(nextSequence);
17              boolean endOfBatch = nextSequence == availableSequence;
18              eventHandler.onEvent(event, nextSequence, endOfBatch);
19              nextSequence++;
20          }
21          sequence.set(nextSequence - 1L);
22          // store barrier 插入到这里 !!!
23      }
24      catch (final Exception ex)
25      {
26          exceptionHandler.handle(ex, nextSequence, event);
27          sequence.set(nextSequence);
28          // store barrier 插入到这里 !!!
29          nextSequence++;
30      }
31  }
32
```

加载屏障（load barrier）

加载屏障，在x86 上是”lfence”指令，强迫所有的、加载指令之后的指令在屏障之后发生，然后等待那个 CPU 的 load buffer 排空。这使其它 CPU 暴露出来的程序状态对这个 CPU 可见，在做出更多进展之前。这个的一个好例子

是前面引用的 BatchEventProcessor 的 sequence 被其它生产者或消费者读取时，Disruptor 里有等价的指令。

Full Barrier

Full Barrier，在x86 上是”mfence”指令，在 CPU 上是加载和内存屏障的组合。

Java 存储模型（Java Memory Model）

在Java 存储模型里，*volatile* 字段在写入后插入一个内存屏障，在读取前插入加载屏障。类里面修饰为 *final* 的字段在它们被初始化后插入一个存储指令，以确这些字段在构造函数完成、有可用引用到这个对象时是可见的。

原子指令和软件锁（Atomic Instructions and Software Locks）

原子指令，如x86里的 “lock …” 指令，是高效的 full barrier，它们锁住存储子系统来执行操作，有受保证的全序关系（total order），即使跨 CPU。软件锁通常使用内存屏障，或原子指令来达到可视性和保留程序顺序。

内存屏障对性能的影响（Performance Impact of Memory Barriers）

内存屏障阻止了 CPU 执行很多隐藏内存延迟的技术，因此有它们有显著的性能开销，必须考虑。为了达到最大性能，最好对问题建模，这样处理器可以做工作单元，然后让所有必须的内存屏障在工作单元的边界上发生。采用这种方法允许处理器不受限制地优化工作单元。把必须的内存屏障分组是有益的，那样，在第一个之后的 buffer 刷新的开销会小点，因为没有工作需要进行重新填充它。

赞 1 收藏 1 评论