False-Sharing

破解“大力出奇迹”的迷信，CPU 越多性能不一定越好。

> https://netflixtechblog.com/seeing-through-hardware-counters-a-journey-to-threefold-performance-increase-2721924a2822

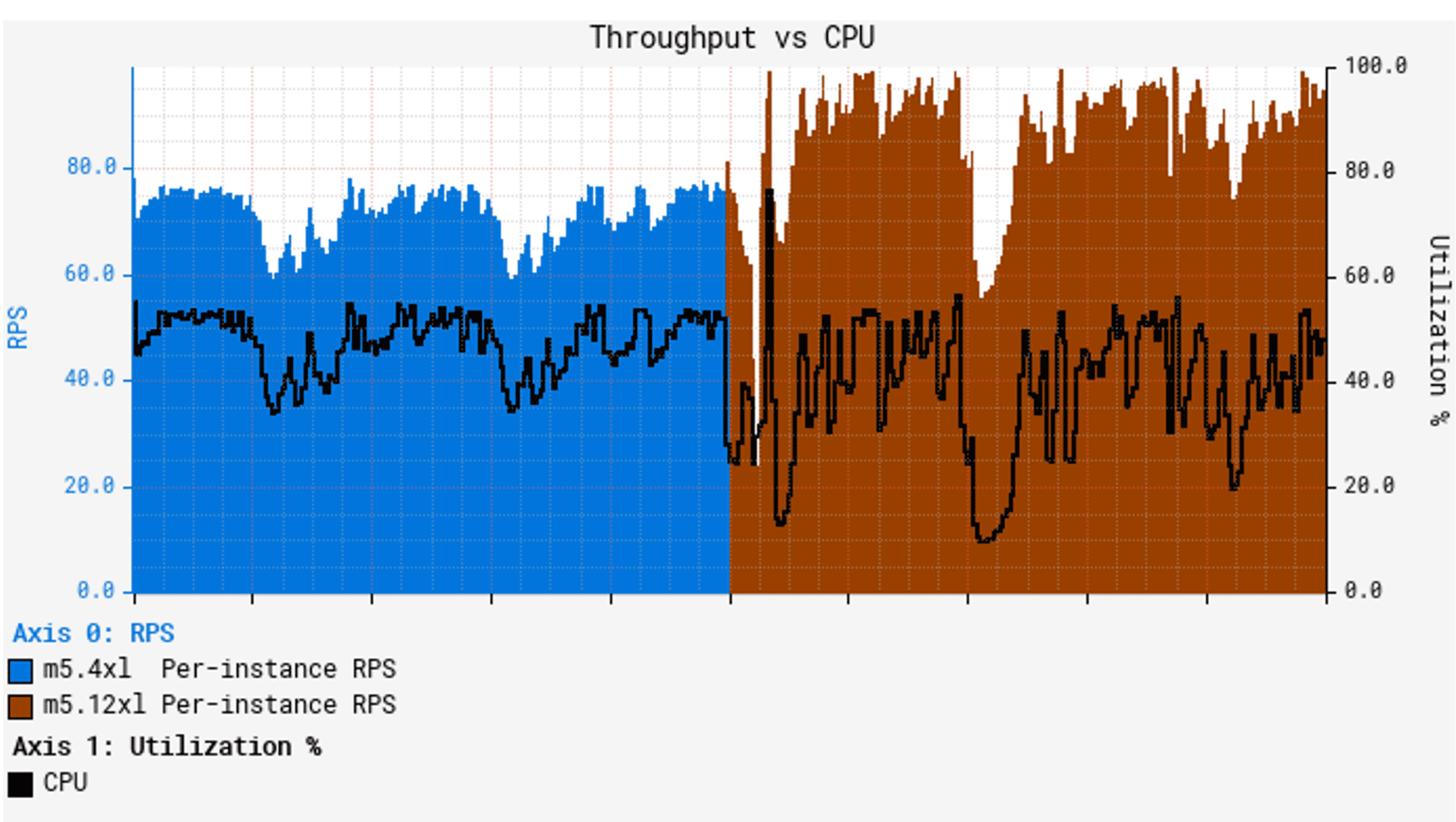
一个真实案例

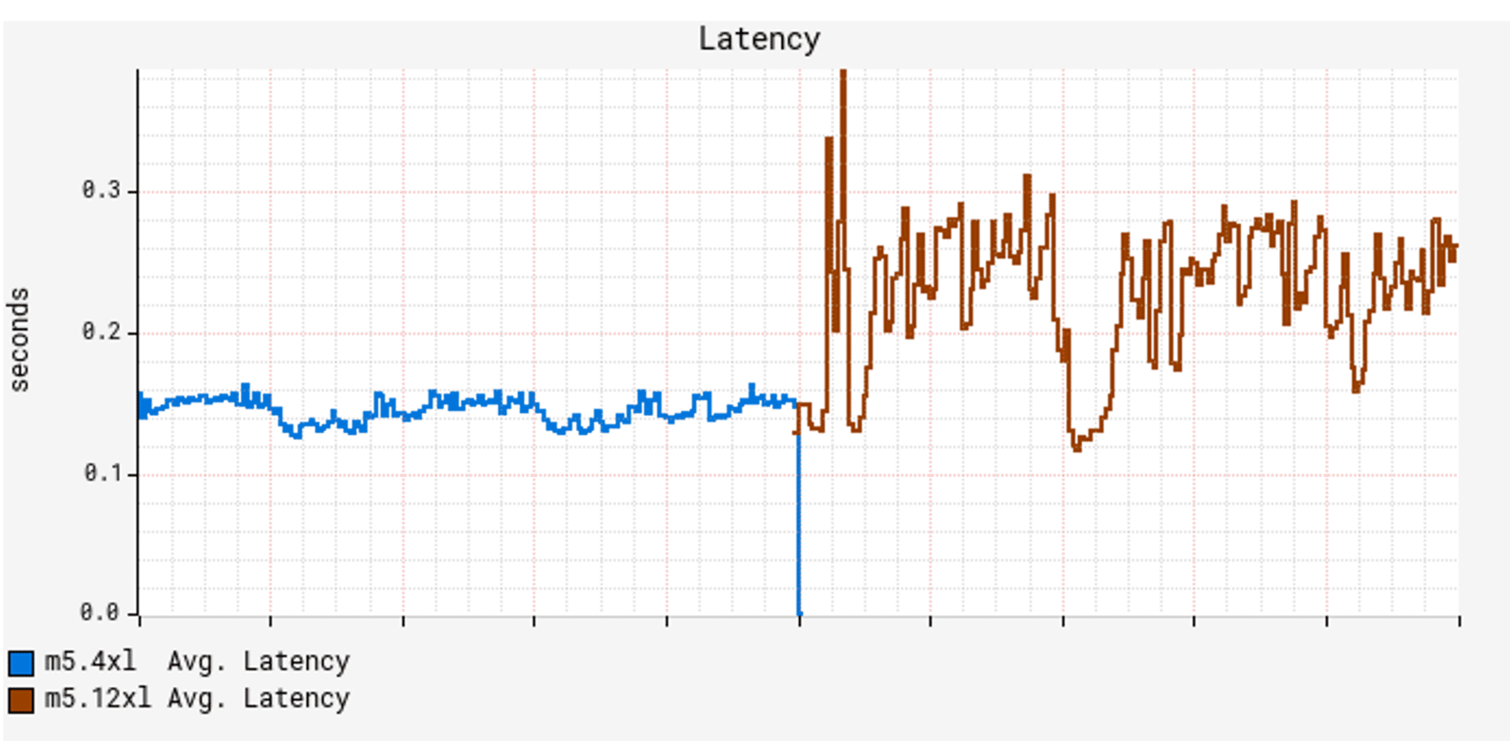
有一个跑在 16 核 CPU 上的应用，没有上游瓶颈。

将其垂直扩容到了 48 核 CPU 上，期望能显著的改善性能。

CPU 扩大三倍，然后实测接口的吞吐量并没有翻倍，实际上只增长了一点点。

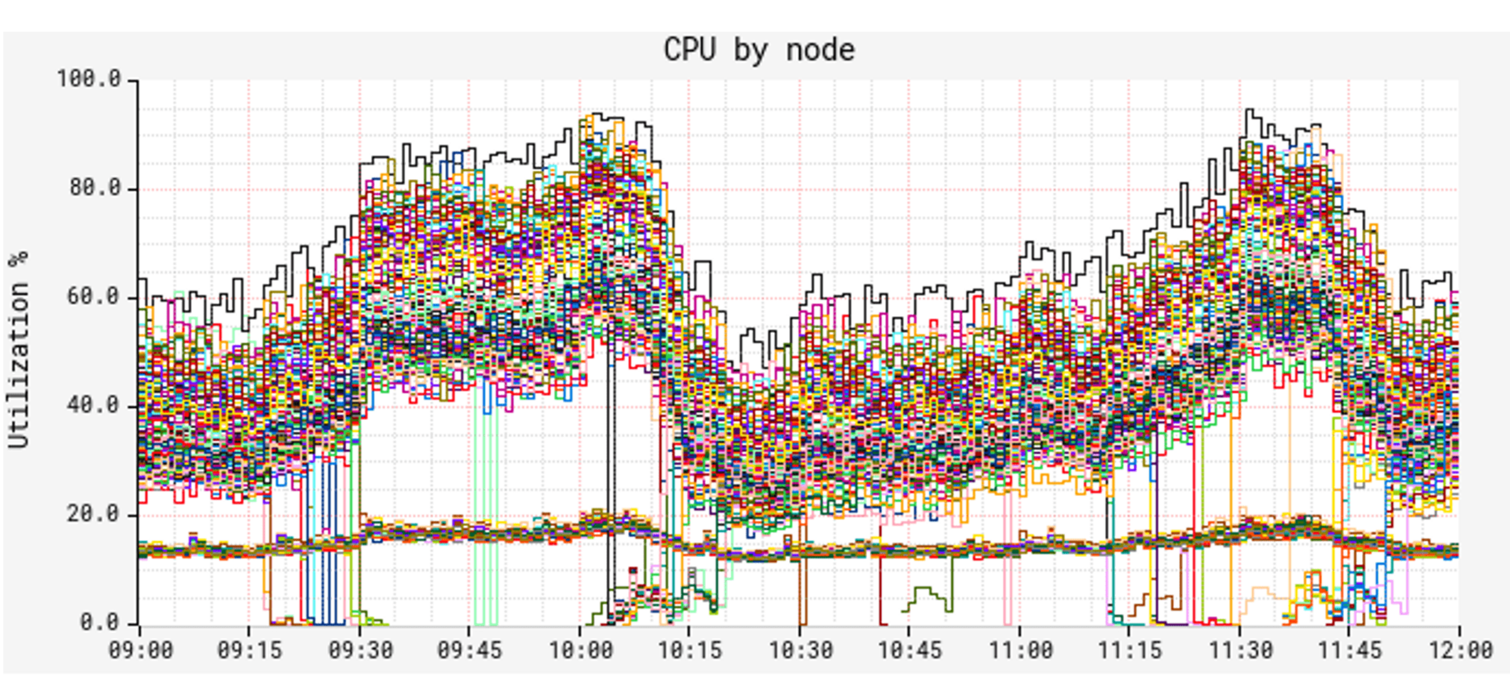
然后更糟糕的是，延迟不降反增，在请求量不变的情况下，48 核的延迟显著高于 16 核。

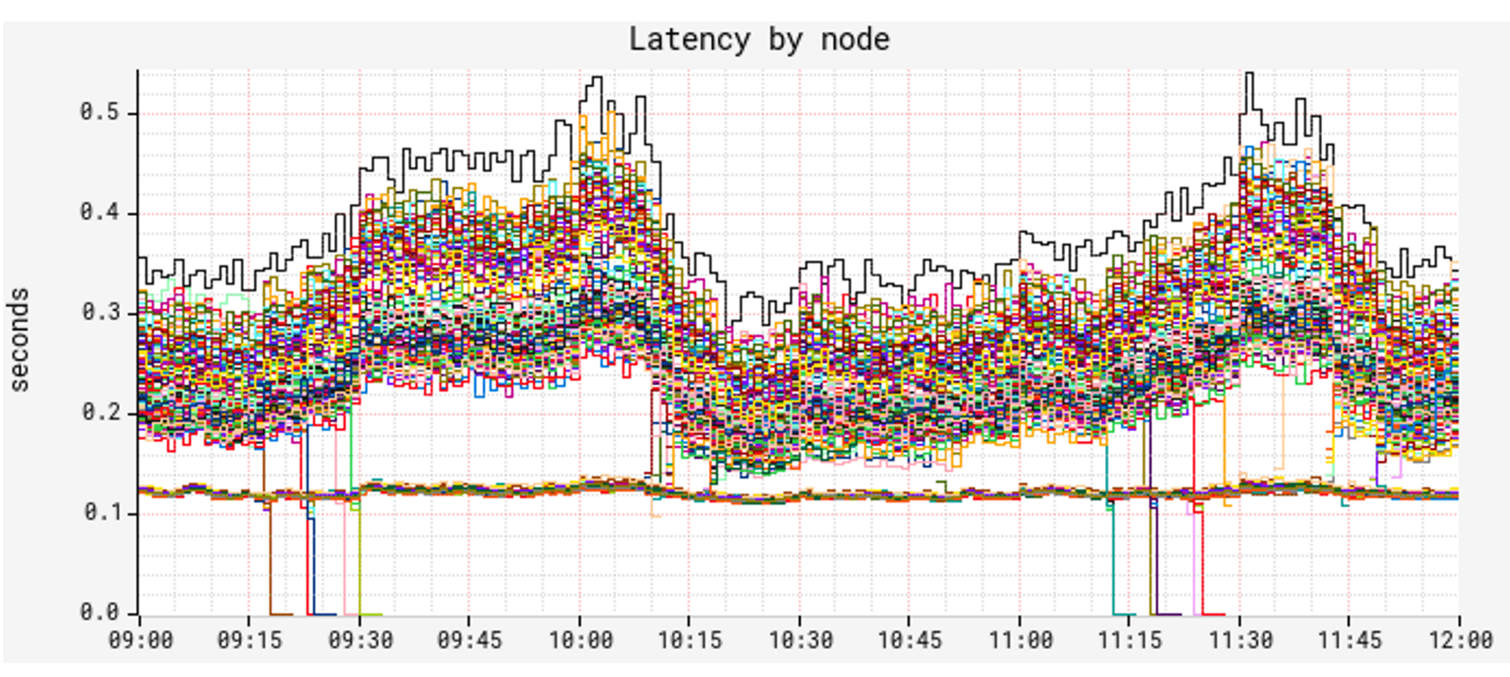




这是一个分布式服务，有很多相同的机器节点。然后仔细观察这些服务的延迟，发现一个非常有趣的现象。

同样的 48 核机器，同样的软件，容易的负载，但是有 12% 的机器 CPU 和延迟显著降低。

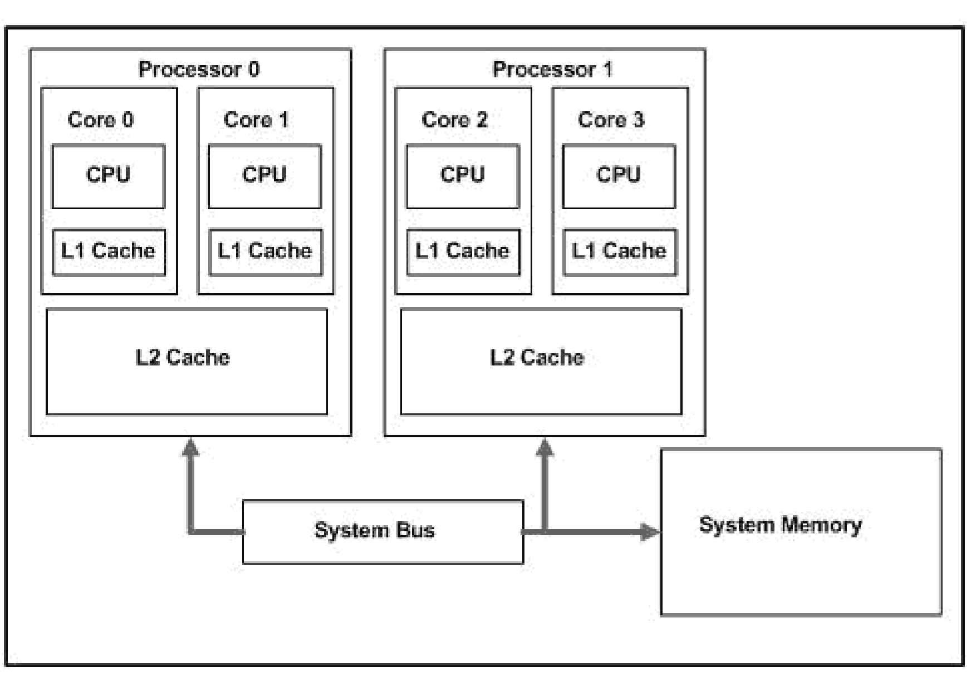




这个案子先放一放，我们先回顾一些背景知识。

Cache Coherency

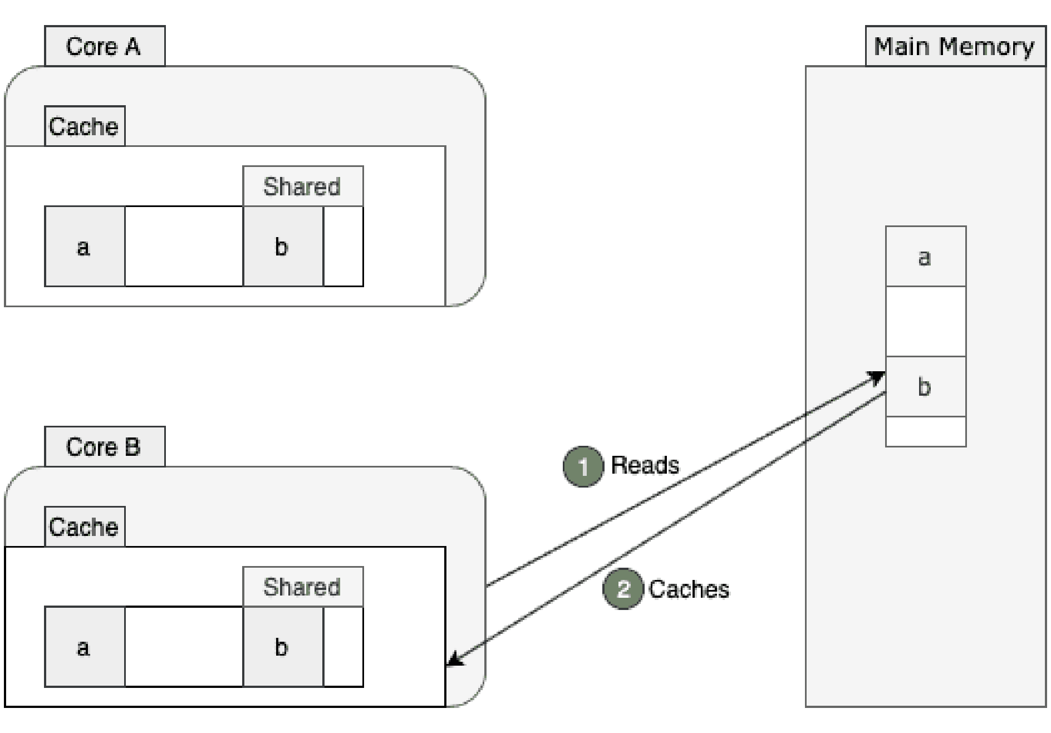
多核机器上，每个核心都有自己的 cache，这些 cache 的数据都是来自同一个共享的主存



当不同的核都从内存中加载了相同的数据时，同一个数据的不同副本就存在于不同的 cache 里。

为了防止出现数据冲突，毕竟每一个 CPU 都希望自己操作的是最新的数据，以满足线性一致性。

就需要涉及复杂的缓存同步协议，称为 cache coherency。



Cache line

现实情况其实更糟糕。

为了提高性能，cpu cache 的加载存在预读，每次以 64 字节为单位加载数据，称为一个 cache line。

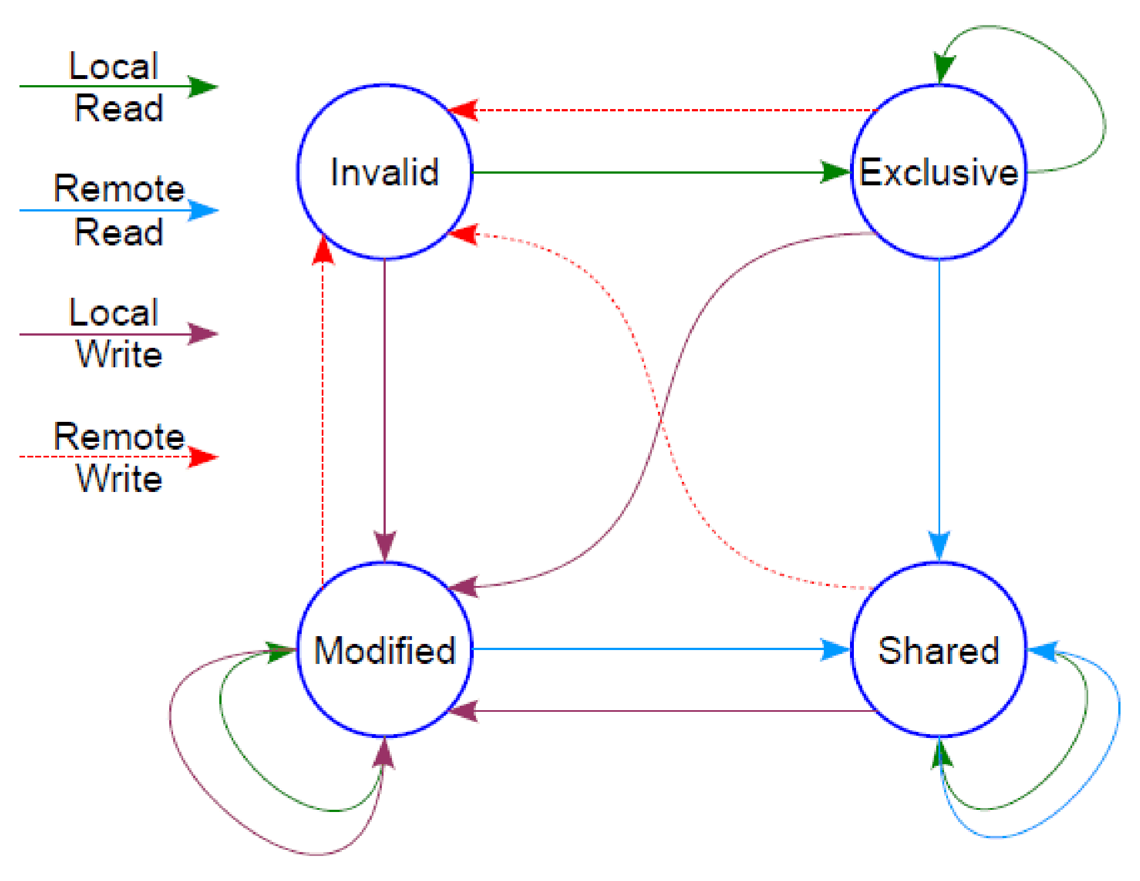
也就是说，即使两个核操作的是不同的数据，但如果这两个数据恰好存在于同一个 cache line，那么这两个核也会出现竞争。 这种情况，就称为 `false-sharing`

MESI

一个最常见的让 CPU 负责管理 cache 一致性的协议，就称为 MESI。

MESI 的原理非常简单，就是将 cache 中的数据标记为四种状态：

- `M: Modified`，修改状态，表示该数据已经被修改，且只存在于当前 CPU 的 cache 中
- `E: Exclusive`，独占状态，表示该数据只存在于当前 CPU 的 cache 中
- `S: Shared`，共享状态，表示该数据存在于多个 CPU 的 cache 中
- `I: Invalid`，无效状态，表示该数据无效，不可用



CPU 要操作 cache 时，只需要判断一下当前数据处于什么状态，就知道是可以直接读写，还是需要重新从主存中重载或者写回。

而 false-sharing 这一现象，会导致缓存中本来可以直接操作的数据，触发大量的重载或写回，导致程序性能下降。

题外话：MESI 的底层

前面说 MESI 简单，因为 MESI 只是告诉你怎么根据状态操作数据，却只字未提这个状态是从而何来？

而了解分布式的人都知道，状态同步正是分布式里最难的地方！

继续再深入计算机体系结构了解一下 MESI 的状态同步机制，就会发现殊途同归，其实靠得还是锁。 而且也有两种常见实现方式：全局锁，或者分段锁。

核心不多的时候就把 bus 当成全局锁，每个核心都通过 bus snooping 订阅数据状态。

核心多的时候就靠分布式 directory，每个内存块的 cache status 都存放在各自的 directory 中，每次更新都去这个 directory 里查询一下共享情况， 然后向所有相关核发送通知并等待回复，等于是一个分段锁 + 2-phase commit 的流程。

老法师把脉

根据前面的背景知识，我们知道 CPU 读取 cache line 一般是以 64B 为单位的。

然后我们知道一个 64 位指针的大小是 8B，而 8/64 正好等于 0.125。

这个数字是不是有点熟悉？我们前面正好提到，升级后的集群里，有 12% 的机器延迟顺利降低了。

再捋一捋。

false-sharing 是两个无关的数据恰好分配到了同一个 cache-line 里。

拿指针来说，就是两个 8B 的数据正好处于同一个 64B 的连续内存里，而这个概率正好是 87.5%。

也就是前面所观测到的现象，12.5% 的机器性能提升了，87.5% 的机器受到了 false-sharing 的影响。

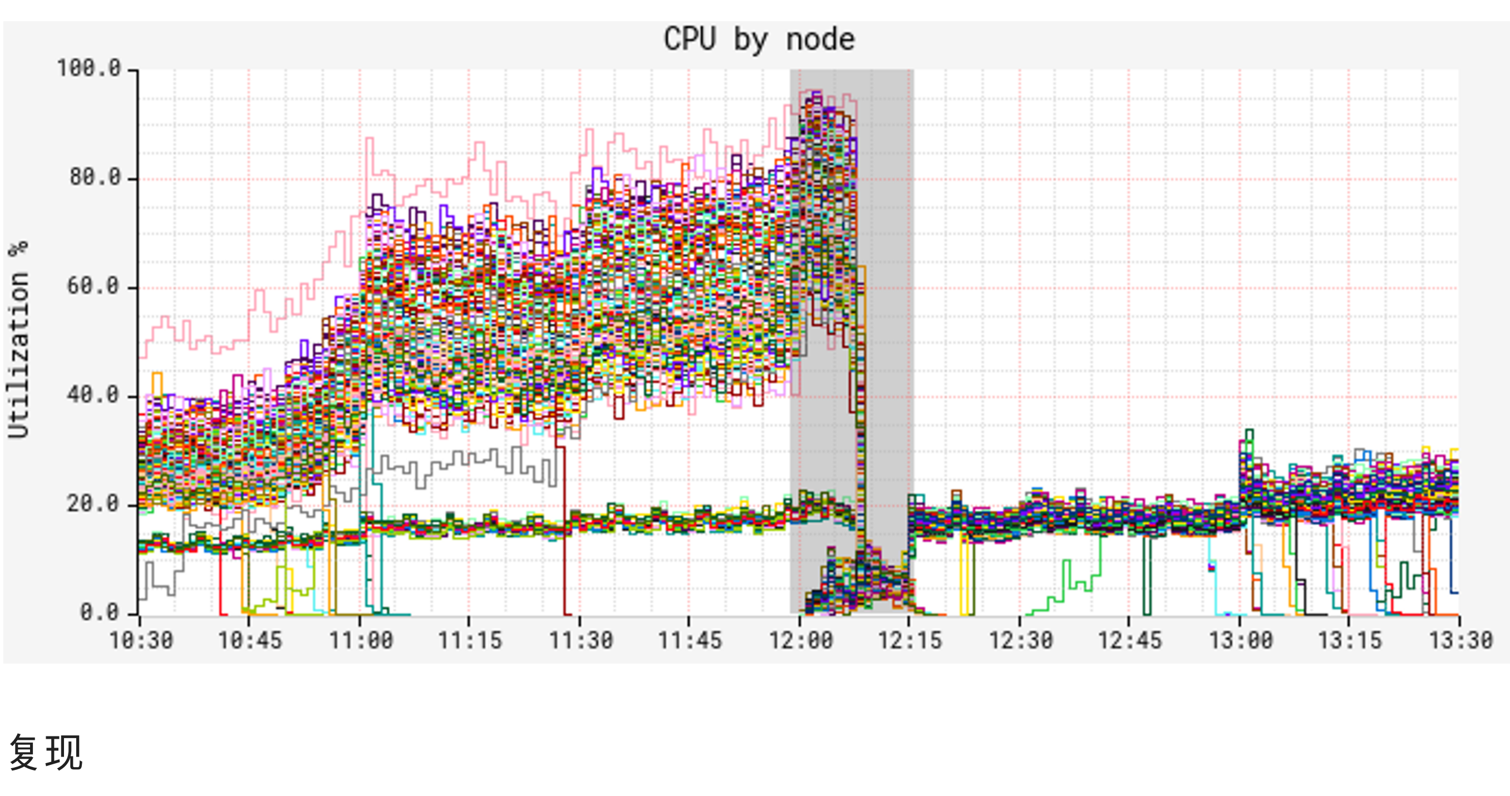
解决

既然猜到了原因，那么就可以按照这个思路解决一下试试。

我们的目的是避免不同的数据共享同一个 cache line，那么最简单的办法就是让每一个热点数据都独占一整个 cache line。

最简单的技术就是内存对齐，我们将热点数据的长度 padding 到 64B，让它不可能和其他数据共享 cache line。

药到病除



复现

在 Go 中也可以很容易的复现这一现象

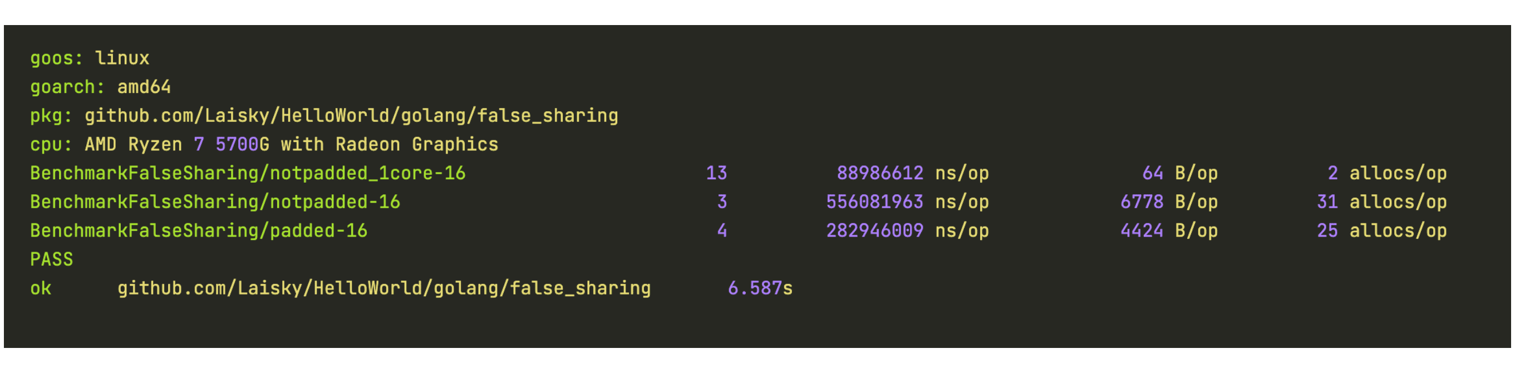
https://github.com/Laisky/HelloWorld/blob/master/golang/false_sharing/sample.go

定义两个相同成员的结构体，一个有 padding，另一个没有，

然后通过 goroutine 进行并发读写。

```
type NotPaddedCounter struct {
    v1 uint64
    v2 uint64
    v3 uint64
}

type PaddedCounter struct {
    v1 uint64
    p1 [8]uint64
    v2 uint64
    p2 [8]uint64
    v3 uint64
    p3 [8]uint64
}
```



```
goos: linux
goarch: amd64
pkg: github.com/Laisky/HelloWorld/golang/false_sharing
cpu: AMD Ryzen 7 5700G with Radeon Graphics
BenchmarkFalseSharing/notpadded_1core-16          13          88986612 ns/op            64 B/op           2 allocs/op
BenchmarkFalseSharing/notpadded-16                 3         556081963 ns/op          6778 B/op          31 allocs/op
BenchmarkFalseSharing/padded-16                    4         282946009 ns/op          4424 B/op          25 allocs/op
PASS
ok      github.com/Laisky/HelloWorld/golang/false_sharing       6.587s
```

题外话：true sharing

false-sharing 是两个无关的数据恰好分配到了同一个 cache-line 里，导致多个 CPU 核心发生了虚假的竞争。

那么 true-sharing 就是同一个数据真的分配给了多个 CPU 核心，导致真实的竞争。

true-sharing 现象就像是一个性能的天花板，决定了程序的性能几乎不可能随着并行度的提高而线性增长。

这一现象也被称为 Amdahl's Law，即程序的性能提升和并行度成正比。

题外话：CPU 跑满不一定是真在干活

有时候我们做压测，或者给机器升级后再压测，看到 CPU Utilization 飙升甚至跑满，但是吞吐量等实际性能指标却没有明显提升。

这时候一定要注意，CPU 利用率实际上是一个很虚的指标，它只是表面当前 CPU 没空，不代表它真的在干活。

比如 CPU 可能在等非 DMA 的 I/O 响应，在等内存响应等等，这类事件的特点是 CPU 其实没在干活，但是它也没法接其他活。

所以在性能测试时还有个很重要的指标是 IPS（Instructions per second），要看看 CPU 究竟是在摸鱼还是在干活。

Thanks