NUMA 和 AMP 架构下的同步原语技术的调研分析

上海交通大学并行与分布式系统研究所（IPADS）

1 现有同步原语技术分析

1.1 SMP 架构下的同步原语

SMP(Symmetric Multi-Processor) 对称多处理器结构，是指服务器中多个 CPU 对称工作，无主次或从属关系。各 CPU 共享相同的物理内存，每个 CPU 访问内存中的任何地址所需时间是相同的。在过去大部分的机器都采用 SMP 架构。在该架构下的诸多锁策略仅需考虑在多个核心之间的竞争与公平性问题。如不保证公平性的 tas 锁在核心数较少的情况下有着较好的性能，但在核心数增多的情况下，多个核心对一个变量的增强导致了巨大的开销并且公平性差。而 ticket, mcs, clh 策略均保证了不错的公平性排队获取锁。

1.2 NUMA 架构下的同步问题

NUMA 全称 Non-Uniform Memory Access，译为 “非一致性内存访问”。这种构架下，不同的内存器件和 CPU 核心从属不同的 node，每个 node 都有自己的集成内存控制器（IMC，Integrated Memory Controller）。一般来说，一个内存插槽对应一个 node。需要注意的一个特点是，QPI 的延迟要高于 IMC Bus，也就是说 CPU 访问内存有了远近（remote/local）之别，访问当前的 node 延迟与性能高于其它 node 。NUMA 架构下锁频繁的在不同 node 之间迁移会导致很高的性能开销，需要尽可能的将锁在 NUMA node 内部传递。现有的 cohort,hmcs,hclh 锁已经对 NUMA 架构下的同步问题提出了解决方案。

1.3 AMP 架构下的同步问题

在 SMP 和 NUMA 中有可扩展性的锁无法在 AMP 中可扩展，并导致吞吐量或延迟崩溃。原因：1. 性能方面这些锁假设核是对称的因此，在 AMP 中，它们给较慢的小核提供了与大核相同的机会来持有锁，这将在小核中引入更长的临界段的执行时间到关键路径，并导致吞吐量崩溃。2. 公平性方面不保护获取公平性的锁依赖于原子操作来决定锁的持有者 (例如，test-and-set 自旋锁)。当同时执行时，它们假定原子操作的成功率是对称的，这在 AMP 中也是不对称的。因此，这些锁很可能只在一种类型的核 (即，要么大核，要么小核) 之间传递。此外，当速度较慢的小内核有更高的锁定机会时，吞吐量也会由于它们上的临界段的较长执行时间而崩溃。

1.4 异构众核下的同步问题

处理器架构持续演进，为提升计算密度和能效比，异构众核（超普核，数百上千核）正在逐步进入到服务器领域，Intel、AMD 下一代服务器芯片规划向大小核架构发展。现有的技术都只针对 AMP 或 NUMA 单一

架构提升互斥锁的可扩展性，这些可扩展互斥锁无法在 NUMA+AMP 异构众核中提供良好的可扩展性。其本质原因是这些互斥锁采用了简单的静态单一倾向性策略以在 AMP 或 NUMA 架构中提供良好的可扩展性，而异构众核（NUMA+AMP）的锁调度中，不再存在静态最优解：单一的静态倾向策略无法提供可扩展的性能表现。因此需要针对异构众核场景研究新型可扩展同步原语。

2 现有锁策略介绍

表2列出了现有针对不同场景、不同硬件架构上的互斥锁可扩展性的研究工作。本文将下面小节依据针对的硬件架构进行划分详细介绍各个工作的核心思路与局限性。

简称	引用	描述	小节
SMP 本地锁			
ttas	[1]	在尝试使用 tas 指令原子地获取锁内存地址之前，对其执行非原子加载，减少原子指令执行次数	3.2
ticket	[29]	通过取票的方式排队获取锁试	3.2
backoff	[24]	在 tas 锁的基础上如果锁被持有进行指令级回退	3.2
clh	[7]	基于链表的公平队列锁，线程在其前驱节点上自旋	3.2
mcs	[25]	基于链表的公平队列锁，线程在其自身节点上自旋	3.1
qspinlock	[6]	基于 mcs 队列的自旋锁，在 linux 内核中使用	3.2
k42mcs	[2]	通过巧妙的手段，使线程结点保存在函数栈中，避免了不必要的内存占用，并在调用时无需除了锁的其他参数	3.2
mutexee	[13]	从能源效率出发的 spin-then-park 锁	3.3
malthusian	[9]	阻塞锁，分离执行队列和等待队列，让部分线程执行部分线程睡眠	3.4
lock-free lock	[3]	未拿到互斥锁的线程帮助已经拿到锁的线程完成临界区	
prwlock	[21]	在 tso 架构下，最大化读取器之间的并行性	
SMP 代理锁			
falt-combine	[16]	粗粒度锁，维护一个 ReqList 保存每个核的 CS 区域，竞争一个锁，获得锁的成为服务器并遍历一遍 ReqList 服务，没有获得锁的等待服务器处理自己的 CS 区域。	3.5
rcl	[22]	代理锁，将所有的锁保护的 CS 区域交给一个当前没有用到的核统一处理	3.6
NUMA 本地锁			
hbo	[28]	允许动态调优回退参数，以便当一个线程注意到来自自己集群的另一个线程拥有锁时，它可以减少两次尝试获取锁之间的延迟，从而增加获得锁的机会	

hclh	[23]	基础 clh 的分层锁，本地 NUMA 节点收集 clh 队列后插入到全局队列	4.1
hmcs	[4]	多层次 NUMA 感知锁，创建了一个 MCS 锁树	4.5
hmcs-wmm	[26]	验证 hmcs 在 arm 架构下的正确性	
ahmcs	[5]	在 hmcs 的基础上提出了一种更精细的快速路径，允许线程绕过树的部分部分，直接从更高级别获得锁	
cohort	[12]	允许用任意两种锁构造一个新 NUMA 感知锁	4.3
cst	[18]	NUMA 感知阻塞锁，既维持本地队列和全局队列，同时在本地队列中使一部分线程超时睡眠进入到睡眠队列	4.4
cna	[10]	维护两个队列，一个 active 队列一个 secondary 队列，线程在放锁时将与其不同的 socket 上的线程放入 second 队列中。	4.6
shfllock	[17]	将锁获取/释放阶段从锁顺序策略中分离出来，并使用锁等待者来找到同一 socket 上的线程（在关键路径之外）	4.7
clof	[8]	通过构建 NUMA 感知锁，可以生成数百个正确的锁	4.8
NUMA 代理锁			
fc-mcs	[11]	在 flat-combing 的基础上构建等待线程的本地队列，并使用指定的线程将它们加入到全局 MCS 队列中	4.2
d-sync	[14]	cc-sync 支持内存一致的方法，h-sync 是 cc-sync 的分层方法，d-sync 则在 cacheless 的 NUMA 机器上表现良好	4.9
sanl	[31]	结合了本地锁和代理锁的优点，在不同的竞争层级下选择不同的策略	4.10
ffwd	[30]	在不考虑支持现有应用的情况下，追求效率展示了代理锁的高速	4.11
amp			
libasl	[19]	针对 AMP 架构，允许大核与小核重新排序，在保留应用程序的延迟需求的条件下获得更高的吞吐量	5.1
general			
pilot	[20]	arm 架构下的屏障优化	6.1
sync-cord	[27]	提供一系列 API 和框架，允许在不重新编译或重新启动内核的情况下修改内核锁	6.2

3 smp 架构下的同步锁策略

3.1 mcs: 将锁竞争移出关键路径的队列锁

概要 在共享内存并行执行的场景下，为保证互斥与同步，常常使用大量的忙等，这会带来大量的内存争用与线程之间共享变量的同步，导致性能瓶颈。本文章提出一种新的队列锁 mcs[25] 锁，通过排队的方式将争用在关键队列上移除。[25]

关键设计 在数据结构方面，mcs 锁仅需要一个指针 tail 指向队列的末尾，每个线程都有一个数据结构 node，node 中仅有两个变量，第一个变量 next 指向在队列中的后继节点，第二个变量 spin 表示自身是否获取锁。在拿锁时，假设线程 B 拿锁时，线程 B 尝试加入队列的队尾，使用 cas 原子操作，得到原本队尾指针指向的线程 A，并将队尾指针 tail 指向自身，然后将线程 A 的 next 指向 B。在放锁时，线程将锁释放给队列中的后继节点。

效果分析 mcs 保证先出先出顺序获得锁，公平性非常强，每个线程获取锁的机会均等。且每个锁仅需要一个变量的空间，每个线程只需要维持一个有两个变量的数据结构，并且 mcs 在关键路径上移除了争用，队尾的原子操作在大部分时间下不影响性能。

局限性 解决了自旋锁大量内存争用已经共享变量同步带来的性能瓶颈问题，以队列的方式保证公平性，且代码可移植性强，是 linux 内核 qspinlock 的基础，后续有多篇工作均在 mcs 策略的基础上进行。

3.2 smp 架构下传统的锁策略分析

概要 本小节主要收集归纳了 smp 架构下传统的一些锁的分析

spinlock spinlock（自旋锁，tas 锁）[1] 不会引起调用者睡眠，如果锁已经被别的线程持有，调用者就在 while 循环中一直使用 cas 原子操作直到操作成功。

pthread pthread（互斥锁）[15] 是 C 语言开发中最常见的锁，他的底层实现逻辑是 spin-then-park，它在自旋等待一段时间后如若没能拿到锁，则调用 futex API 进入到睡眠状态。在放锁时如果没有其它线程自旋等待，则调用 futex API 去唤醒睡眠的线程（如果有的话）。

backoff [24] 在 tas 锁的基础上，如果锁被占用，则进行指数级回退，解决了高竞争情况下的过度争用问题。

ttas [1] 是在 tas 锁的基础上，在执行 test-and-set 操作前加一层 test 操作来减少原子操作的次数（因为原子操作开销大）。

ticket [29] 让试图获得锁的线程原子地获得一个“票”(作为递增计数器实现)，并在它的票不等于“下一个票”数字时旋转。在解锁时，锁持有人增加“下一票”号码。

qspinlock [6] 是 linux 内核锁，它集成了 mcs 算法到自旋锁中，继承了 mcs 算法的所有优点，有效解决了 CPU 高速缓存行颠簸问题，并且没有增加 spinlock 数据结构的大小，把锁变量 var 细分成多个域，较好的实现了 mcs 算法。

k42mcs [2] 在 mcs 锁的基础上通过巧妙的手段，使其结点保存在函数栈中，避免了不必要的内存占用，并在调用时无需除了锁的其他参数。但是从外部调整队列时无法 work。

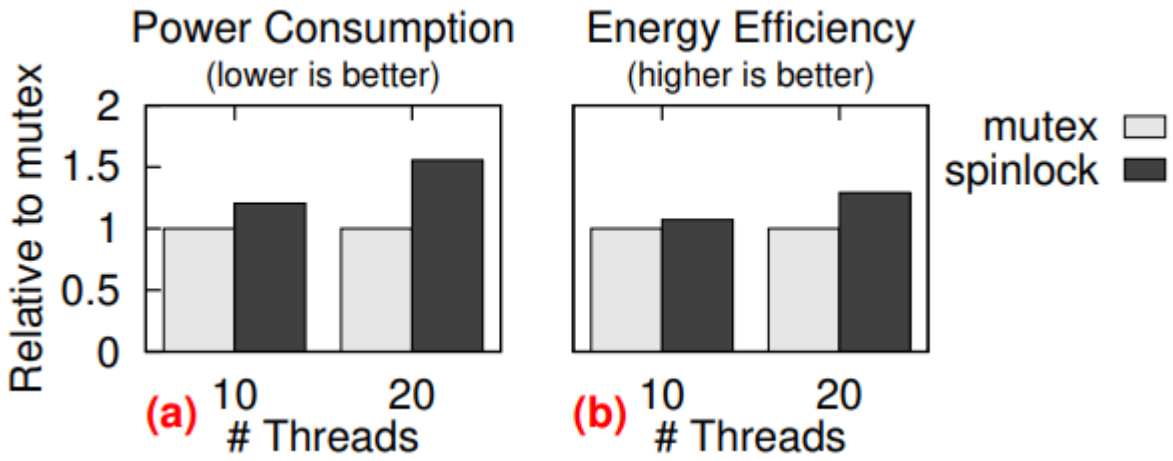
clh [7] 等待线程被组织为一个反向链表，线程在其前趋节点的上下文（即链表节点）上自旋等待，如果前趋节点放锁则结束自旋。该方法在 SMP 结构下性能良好，但在 NUMA 系统结构下，前趋结点在不同的 socket，导致吞吐量断崖下降。

3.3 mutexee: 从能源效率出发的 mutex 锁

概要 在锁算法的环境下，能源效率和吞吐量是正相关的，mutexee[13] 从设计更节能的算法出发角度出发，改善了锁的吞吐率。

核心观察 本篇文章从比较互斥锁 (mutex) 和自旋锁 (spinlock) 锁的区别出发，观察到以下结果：1. mutex 在获取锁失败时睡眠，而 spinlock 则忙等。2. 选择睡眠的 mutex 会有 50% 的电力节省 3. spinlock 版本比 mutex 多消耗 50% 的电力，但它的能源效率 (单位吞吐量下所消耗的能量) 比互斥锁高 25%，因为它的吞吐量是互斥锁的两倍。

图 1: 自旋锁与互斥锁的能源效率



得出阶段性结论：1. 忙等本质上增大了电力消耗，在忙等待的情况下，底层硬件上下文保持活动状态。例如，在英特尔 (Intel) 的机器上，减少繁忙等待的功耗实际上是不可实现的。2. 睡眠会节省能量但会损害能量效率，当一个线程处于活跃状态，不管它执行的是什么类型的工作都会消耗能量，但如果线程在等待一个忙锁后时处于休眠状态，我们可以节省这部分能量。pthread 互斥锁调用 futex 系统调用来实现睡眠，然而在大多数现实的场景中，频繁的 futex 调用开销 (唤醒一个正在沉睡的线程至少需要 7000 个周期) 抵消了睡眠所带来的能源收益，从而导致更糟糕的能源效率。

关键设计 mutexee: 一个优化后的 mutex 锁, 相比 mutex 锁，mutexee 自旋 8000 个 cycle, 因为执行越不公平，吞吐量越高，能源消耗越小。用 mfence 替换 pause, 加入 mfence 避免投机性执行, 来提高自旋等待时的 CPI，从而降低功耗。在放锁后调用 futex_wake 前等待一会，这是因为在 mutex 中，放锁后会如果没有其它线程拿锁会立即唤醒睡眠的线程，但如果有其它线程在被唤醒的线程能够执行前拿锁，被唤醒的线程可能迟迟拿不到锁又进入到睡眠状态。在高争用的情况下可能会产生高尾时延，但论文声明在实际系统中未观察到尾时延的大幅上升。mutexee 提供降低尾时延的方法，在 futex_wait 时加入 timeout，长时间睡眠后唤醒，唤醒后一直自旋不会再次睡眠，该方法会大大降低吞吐量。

效果分析 mutexee 的吞吐率能够达到 mutex 的 2.5 倍，有着更高的 TPP，但是由于部分线程长时间睡眠，尾时延非常高，此外可以通过 timeout 唤醒长时间睡眠的线程能够保证尾时延，但吞吐率大幅下降，仅为 mutex 的 1.5 倍左右。

局限性 从能源的角度出发，让线程长时间睡眠来提高能源效率，并带来吞吐量的提升，但牺牲了尾时延。

图 2: mutexee vs mutex

	MUTEX	MUTEXEE
lock	for up to ~ 1000 cycles spin with pause	for up to ~ 8000 cycles spin with mfence
	if still busy, sleep with futex	
unlock	release in user space (lock->locked = 0)	
		wait in user space
	wake up a thread with futex	

3.4 malthusian：让部分线程睡眠的阻塞锁

概要 在 over_subscription 的情况下，重新排序一个类似 mcs 的队列，在保证锁被充分竞争的前提下，让部分线程睡眠，让部分锁能够重复快速拿锁，并保证长期公平性。[9]

关键设计 malthusian 让多余的线程睡眠并移出 active 队列，假如当前有 20 个线程，而非临界区执行时间 5s，临界区执行时间 1s，那么尽量保证参与竞争进入 active 队列的线程数数量为 6，剩下的 14 个线程进入 passive 队列中，他的实现逻辑如下：1. 在拿锁的时候就是简单的 mcs 队列。2. 在放锁的时候，如果当前线程和队尾之间还有线程，则放入一个后继线程进入 passive 队列中。3. 如果发现没有后继线程，从 passive 队列中取出一个线程进入 active 线程。4. 保证长期公平性，线程长时间进入 passive 队列后会进入 active 队列。

效果分析 有意地将多余的线程（不需要维持争用的多余线程）剔除到 passive 队列中，减缓和减少了活动线程集合的大小，提高了相对于公平的 fifo 锁的吞吐量。malthusian 通过定期地重排锁队列，在执行队列和等待队列之间转移线程，提供长期的公平性，节省了共享资源，还可以减少抖动效应和性能下降。在 over_subscription 的场景下，相比于扩展了 spin-then-park 策略的 mcs 锁有着 16 倍吞吐量的优势。

局限性 malthusian 分离了执行队列与等待队列，让部分线程执行部分线程睡眠，这种维持多个队列的方法需要额外的内存空间。此外，通过线程重复拿取锁次数来替换线程的方法在关键路径引入额外开销。

3.5 flat-combining: 将临界区访问访问请求收集起来集中由 combiner 执行

概要 用粗粒度锁，收集对需要保护的共享数据结构的访问信息，集中在获取锁的线程（combiner）调用执行 [16]

关键设计 flat-combining 维护一个 ReqList 保存每个核的 CS 区域，竞争一个锁，获得锁的成为 combiner 并遍历一遍 ReqList 服务，没有获得锁的线程等待 combiner 处理自己的 CS 区域。这么做的优点是一个线程去处理所有的临界区，缓存局部性更强更不容易出现 Cache miss。但 flat-combining 的缺点也很明显，如下：1. 成为服务器的 thread 的 CS 区域很长，需要遍历全部的 list，完成全部的请求后才可以跑。2. 服务器的迁移开销，成为服务器比较随机，有可能导致没有意义的服务器迁移。3. 没有对 NUMA 架构进行优化，是非 NUMA 感知的。4. 没有限制服务器的存活时间，很可能被阻塞。

flat-combine 的额外开销：对 ReqList 的 head 的 cas。如果没服务器的话，还要多一个对 lock 的 cas 以及遍历所有请求。

效果分析 使用 flat-combining 技术设计的线性化堆栈、队列和优先级队列算法，性能大大优于所有先前

的算法。

局限性 flat-combining 使用的内存较多，相对复杂，且需要特定的负载、平台和工作负载的调优，来确保性能。

3.6 rcl: 将临界区交给专门核心处理的代理锁

概要 代理锁，rcl[22] 通过在一台或多台专用服务器上远程执行访问高竞争锁的关键部分，来提高多核硬件多线程应用程序的性能。

关键设计 rcl 将所有的锁保护的 cs 区域丢给一个当前没有用到的核统一处理，相较于 flat combining 解决改进以下几个问题：1. 利用没有用到了核心来增加程序的性能。2. 避免单个服务器所在的线程 cs 区域太长。3. 避免服务器切换带来的开销。4. 解决服务器可能出现的锁嵌套以及被调度走等限制性能的因素。

效果分析 rcl 的局限性：服务器核心的处理效率。rcl 中只有固定的核心来处理 cs 区域。这个核心上有其他的应用程序，也会影响到服务器的效率，以至于影响整个 cs 区域运行的效率。rcl 是非 NUMA 感知的，无法在 NUMA 架构下可扩展。rcl 的额外逻辑也比较少，大部分逻辑在服务器核心上。对于普通的客户端请求锁，只需要填下每线程的 request 即可（无竞争，按页对齐）。

局限性 相较于 flat-combining 有所改进，但是局限性仍然较强，且易受服务器所在核心效率的影响。

4 NUMA 架构下的同步锁策略

本小节锁策略，均可在 NUMA 架构下可扩展。

4.1 hclh: 基于 clh 的分层 NUMA 感知锁

概要 基于 clh 锁扩展的 NUMA 感知锁（分层 clh 队列锁），线程构建等待线程的隐式本地队列，只需要一个 cas 操作将它们拼接到一个全局队列中 [23]

关键设计 每个 NUMA 节点内的线程维持本地 clh 队列，同时有一个全局队列。当线程想要获得锁时，它将自己添加到所在 NUMA 节点的本地队列中，线程在它们的本地队列的前任上自旋。位于本地队列头部的线程试图将整个本地队列拼接到全局队列上，以便来自同一 NUMA 节点的多个线程连续出现在全局队列中，改善内存访问局部性，这样的操作仅仅需要一个 cas 操作。hclh 牺牲全局的先进先出的顺序，保证了本地锁的先进先出顺序，锁优先在本地队列传递来，并提供长期公平性，同时允许重用锁，N 个线程访问的 L 个锁只需要 O(N + L) 内存。

效果分析 hclh 克服了基于回退的 NUMA 感知锁的公平性问题，在 NUMA 架构下可扩展。并且相比回退锁频繁的 ttas 操作，本地队列并入全局队列仅需要一个 cas 操作，开销更小，提高了吞吐量，有着更好的局部性和公平性。

局限性 为确保线程是否执行成功合并本地 clh 队列与全局队列的操作，hclh 在其关键执行路径上有复杂的条件检查，拖慢了执行速度。此外在共享变量上的 swap 操作会引入开销造成瓶颈，并且如果本地队列长，本地队列合并到全局队列的线程必须等待很长时间，如果本地队列少，执行锁的局部性就得不到保证，因此 hclh 能提供的本地性提升其实是十分有限的。其有较短的 batch 时间区间（指一段时间内的请求能 batch 到一起）上限，固定的数目上限（最多只能 batch 结点上请求的数目，每个请求只能有一次）。但是由于加入到全局队列是一个 fifo 的，其能保证一段时间范围内的公平性。

4.2 fcmcs: 基于 flat-combine 的 NUMA 感知锁

概要 本研究结合 flat combining 技术和 mcs 队列，设计实现了在 NUMA-aware 的 fc-mcs 锁。[11]

关键设计 1. 线程在它们的本地节点上旋转，同时选择一个线程作为指定的 combiner。2. 该 combiner 通过收集所有旋转线程的请求来构造一个本地队列，然后将这个队列拼接到全局队列中。

效果分析 与 hclh 锁相似，均是将本地队列收集起来，通过 cas 操作将本地队列插入全局队列，然后依据全局队列顺序来决定获取锁的顺序，但 fcmcs 克服了 hclh 的缺点，相比采用共享位置上进行 swap 操作的 hclh 锁，fcmcs 允许线程仅使用简单的写操作 (甚至没有写-读内存屏障) 将请求并发到非共享位置，因此线程能够被快速地收集到本地队列中, 能够形成相对较长的本地队列，并且几乎没有延迟，且关键路径短，开销低。

局限性 非 memory efficient, 一个线程必须为它反复访问的任何锁保留一个节点，并且只有在线程停止访问给定的锁之后，这些节点才会被回收。

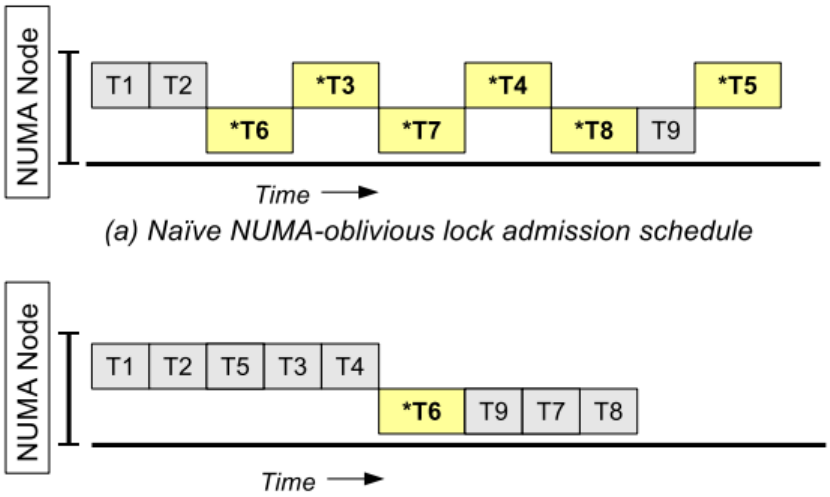
4.3 cohort: 将任意锁算法组合成新的 NUMA 感知锁

概要 本研究针对 NUMA 环境提出的一种队列锁，允许将任何自旋锁算法转换为可扩展 NUMA 感知的自旋锁，只需要进行很小的非侵入性更改。新的队列技术允许我们轻松创建 NUMA 感知版本的 tatas-backoff、CLH、MCS 和 ticket 锁。[12]

关键设计 cohort 采用分层的方法，两层锁，一层全局锁，一层本地锁。线程先获取本地锁，再获取全局锁。全局锁和本地锁的策略可以灵活变换不同的组合，在不同的场景下不同的策略组合有着不一样的性能。cohort 下的公平性仅需要通过维持多久全局锁后放锁来维持，过长的持有全局锁的时间可能会导致更高的吞吐量，但也会带来更不公平的时延。

效果分析 如图所示，cohort 锁大大降低了锁在不同 NUMA 节点之间转换的次数，大大提高了吞吐量。当负载较低时，队列锁的性能与已知的 NUMA 感知锁一样好，甚至更好，而当负载增加时，队列锁的性能明

图 3: cohort 下的锁调度



显优于已知锁。

局限性 cohort 这一套组合的方法不容易扩展到支持多级 NUMA 系统，且在单线程下，由于较多的原子指令操作有次优的性能。

4.4 cst:NUMA 感知的阻塞锁

概要 提出了几种可扩展的阻塞同步原语的设计，来处理 under-over_subscription 的场景，解决了 NUMA 场景下的可扩展性问题。[18]

核心观察 实现可扩展性在 NUMA 场景下十分重要，目前 NUMA 机器可以将多达 4096 个硬件线程组织到 socket 中，但是现有方案的不足以满足 NUMA 架构下的可扩展：1. 操作系统的开发人员倾向使用 tas 锁及其变体，因为其实现简单并且在核心数少的情况下缓存行争用不明显，但在大型的多核机器上可扩展性很差。2. 现有的阻塞同步原语是 NUMA 无知的。3. NUMA 感知锁（如 cohort locks) 静态的为所有的 NUMA 节点分配内存空间，导致了内存膨胀问题，并且这些锁均为非阻塞锁。4. 当前的阻塞原语受低效的调度策略、系统负载估计、全局 parking_list 的使用，缓存行竞争的影响。

关键设计 cst 提出了两种可扩展的同步阻塞原语：cst-mutex 和 cst-rwsem，它们有以下优点 1. 高效的数据结构，每个 socket 动态分配内存，解决内存膨胀问题。2. 将锁传递给自旋等待的线程，减轻了与调度器的交互。3. 通过查看 CPU 上正在运行的任务的数量 (即 CPU 调度运行队列的长度) 来评估系统负载，扩展了调度程序来预测系统负载，有效地处理订阅过多和订阅不足的情况 cst-mutex 的实现逻辑如下：1. 每个线程获取锁分为两步，第一步获取本地锁 (即 NUMA 节点内部的锁)，第二部获取全局锁。2. 每一个 NUMA 节点

拥有一个 snode(在创建该 NUMA 节点下第一个线程时动态分配), snode 第一个成员 qtail 指向 waiting_list 的队尾, parking_list 组织被阻塞的线程, 每一个线程有一个 qnode 节点, 在尝试获取锁时加入到 waiting_list 队列中。3. 每一个线程都有一个时间片，在拿锁时进入到 waiting_list 的末尾自旋等待，当自旋等待一段时间后时间片消耗完就会进入到 parking_list。以此来减少 waiting_list 上的线程数（类似 malthusian），达到减少 cpu 争用的目的。

效果分析 cst 锁在高度竞争但 under_subscription 的系统中具有与队列锁相同的好处。与 cohort 锁不同的是，cst 锁有意识地通过增加套接字计数来分配内存，parking_list 组织被阻塞的线程, 每一个线程有一个 qnode 节点, 在尝试获取锁时加入到 waiting_list 队列中。得益于高效的 parking 设计, 在 over_subscription 的场景, cst 的吞吐率是 cohort 锁和 linux 本地锁的 2 倍左右。

局限性 低争用时 cst 的原子操作执行频繁导致吞吐率受影响，此外 cst 追求吞吐量，不考虑时延，只保证了长期公平性。

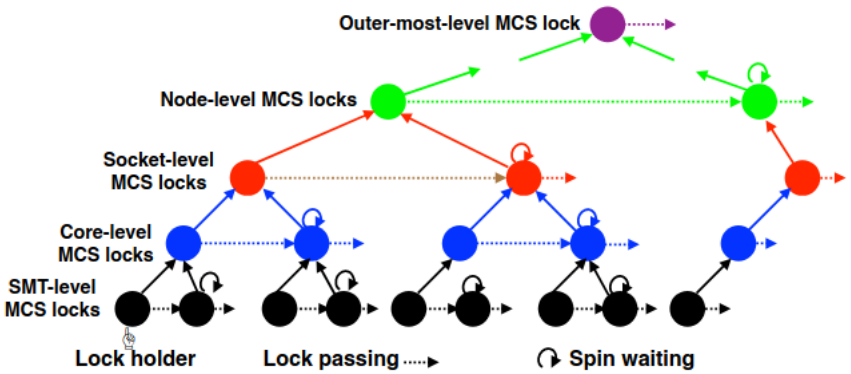
4.5 hmcs: 多级 NUMA 架构下的可扩展锁

概要 提出了分层 mcs 锁 (hmcs)———一种充分利用每一级 NUMA 层次的锁队列，在高争用的环境下有更好的性能。[4]

核心观察 之前的 NUMA-aware 锁都是两层锁 (如 hclh，fc-mcs)，在多级 NUMA 系统下没有三层及以上的锁。

关键设计 1. 在多个级别的层次结构上使用 mcs 锁，以利用每个级别的局部性，是一个 n 元的 mcs 锁树，如下图, 只有当从叶子到树根的路径上的所有锁都被获取线程持有时，才能进入临界区。持有锁的线程在完成其临界区后，将锁传递给其 NUMA 域中的后继者。在 NUMA 域内的锁传输达到阈值数量后，线程在最低最近的的 NUMA 域传递锁。

图 4: Hierarchical MCS lock



2. 同 hclh 和 cohort-mcs 一样，它不确保全局的请求线程的 fifo 顺序，而确保在 NUMA 层次结构的同一级别的请求线程中的 fifo 排序

效果分析 1. hmcs 在多级 numa 结构上性能很好。在 128 线程的 IBM Power 755 机器上的高度竞争情况下，三级的 hmcs 锁会比二级的 cohort-mcs 锁吞吐量高 7.6 倍。而一个五级 hmcs 锁在 4096 线程的 SGI UV 1000 上提供了高达 72 倍的锁吞吐量 2. 以牺牲延迟为代价提高吞吐量，最适合于高竞争的锁。3. hmcs

锁在初始化后不需要显式的内存管理。

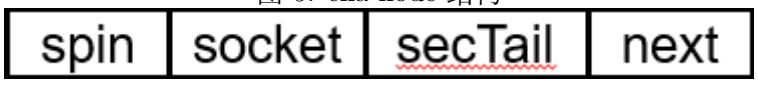
局限性 hmcs 无法在不同级别的竞争下调优，不适合低竞争环境。

4.6 cna: 精简的 NUMA 感知锁

概要 这项工作提出了一种精简的 NUMA 感知锁，称为 cna。之前的 NUMA 感知锁都采用两层锁，由一组 NUMA 节点本地队列（通常为 socket 数量个队列）和一个维持全局的队列组成。而 cna 锁不管有多少个 NUMA 节点都仅需要一个 int 类型的变量的内存空间，cna 锁是 mcs 锁的一种变体，其维护两个队列，一个由与当前运行的线程同一 NUMA 节点上运行的线程组成，另一个由在不同的套接字上运行的线程组成。cna 尝试将锁传递给运行在同一套接字上的后续线程，即不断在第一个队列上传递。[10]

关键设计 每一个线程的加入队列的 node 结构，第一个变量 spin，非 0 代表获取锁，第二个变量 socket 代表线程所在的 NUMA 节点，第三个变量 secTail 指向第二队列的尾部，第四个变量指向队列的下一个线程。

图 5: cna-node 结构



cna 维护两个队列，一个 active 队列一个 secondary 队列，插入队列的方式均采用 mcs 队列的方式，线程首先加入到 active 队列，拿到锁的线程在放锁时逐一检查后续线程是否与其在同一 socket，当找到同一个 socket 的线程传递锁并将两个线程之间的线程放入 secondary 队列，若没有同一 socket 的队列或在同一 socket 中传递了达到阈值(防止饥饿)的次数，放锁给 secondary 队列的队首并将 active 队列插入到 secondary 队列队尾组成新的 active 队列。

效果分析 在 NUMA 环境下有着与分层的 NUMA 感知锁相似的吞吐率，但有着更低的内存占用。与分层的 NUMA-aware 锁不同，cna 可以很方便的进入内核，只需要基于 qspinlock 做一些小的修改，而由于需要的地址空间远远超过 4 字节，之前的分层 NUMA-aware 锁无法集成到内核。

局限性 cna 的锁状态与队列尾是耦合的，在低竞争的情况下性能差，并且仅仅支持两层 NUMA 结构，目标平台是 x86，并没有考虑 WMM。

4.7 shfllock：将锁执行顺序和锁释放阶段解耦的 NUMA 感知锁

概要 本研究提出了一种内存占用低的 NUMA 感知锁，将锁获取/释放阶段从锁顺序策略中分离出来，并使用锁等待者 (即等待获取锁的线程) 来执行 shuffle 策略，且在关键路径之外 [17]

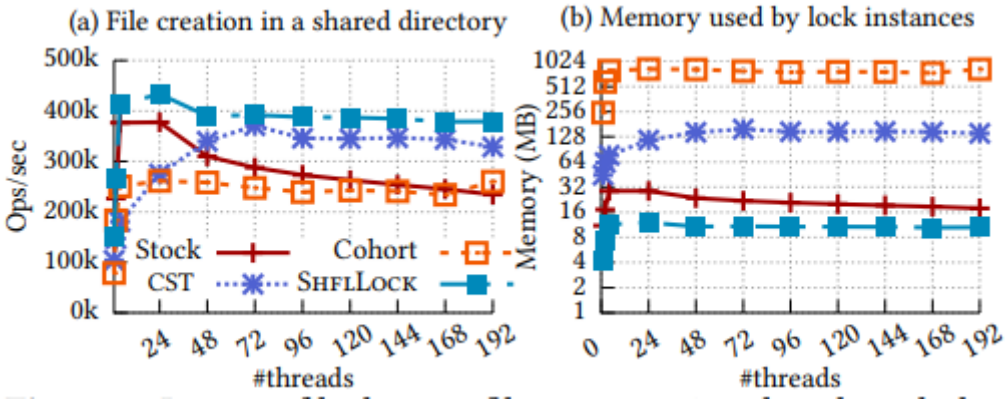
核心观察 提出四种影响锁算法因素（在此之前没有锁完美考虑了四个因素）：1. 避免跨 NUMA 节点访问 (cache-line movement between different caches)。2. 是否适用线程竞争程度。3. 在 over-under oversubcription 下都性能良好。4. 低内存占用。

关键设计 拿锁阶段，要拿锁的线程先执行一次 tas 操作，成功则执行返回（保证了低争用下的性能），否则加入到 mcs 队列中。mcs 队列的队首成为 shuffler，将队列中同一 numa 节点的线程重排序到队列前列，从而最大限度地减少 NUMA 节点之间移动，这一操作在成功拿锁前完成，不在关键路径上。同一个 numa 节点的线程拿锁次数达到阈值后会找到非同一节点下的线程成为新的 shuffler。此外，针对 over_subscription 的

情况，shfllock 加入了 park/wait 策略，因此 shfllock 也是高效的阻塞锁。在内存占用方面，shfllock 仅需要一个恒定的、最小的数据结构，并且不需要在临界区内进行额外的分配，内存占用非常小。

效果分析

图 6: 不同争用下 shfllock 与其他锁的对比



如图所示，在有 8 个插槽共 192 个核的 x86 架构的机器上，比较了 shfllock 与 linux 默认版本锁 (stock)，队列锁（cohort) 和 cst，shfllock 在低争用和高争用下都有着很好的性能，并且内存占用率最低。

局限性 shfllock 与 cna 锁类似，不同之处在于将锁执行顺序和锁释放阶段解耦，与 cna 锁一样，shfllock 仅仅支持两层 NUMA 结构，仅在 x86 下实现良好，并没有考虑 WMM。

4.8 clof: 一个针对多级 NUMA 架构的组合锁框架

概要 x86 和 Arm 架构中 NUMA 层次结构的差异对性能的影响还没有得到充分的研究，当简单地将 NUMA 感知锁从 x86 移植到 Arm 时，性能会出现巨大差异。本篇论文提出了一个多层次 NUMA 系统的组合锁框架 (clof)。clof 在与目标平台匹配的层次结构中组合非 NUMA 感知锁，通过构建 NUMA 感知锁，可以生成数百个正确的锁，并验证了 clof 在 WMM 上的正确性 [8]

核心观察 为高性能 NUMA 感知锁的设计应该满足以下四个层次：A1:multi-level, 锁必须支持目标系统的整个内存层次结构，不仅是 NUMA 节点，还包括 package, cache levels, and cache coloring，通过利用不同 level 的局部性，锁实现可以降低内存压力并提高性能。A2:heterogeneity, 锁算法可以根据不同级别的竞争进行调优，简单的算法在低竞争下往往更快，但在高竞争下则会受到影响。A3:architecture-optimized, 不同的架构允许不同的优化，例如 x86 上的的 coherence-traffic 减少，而 arm 没有。A4:correctness，arm 实现了 WMM(weakly-ordered memory model)，为了确保 WMM 上的正确性，必须小心地使用内存屏障来强制执行必要的顺序。

现有锁无法同时满足以上四个特质。如图所示，cna 和 shfflock 仅支持两层 NUMA 架构（A1，A2)，目标平台仅是 x86 架构未在 arm 下优化且验证正确性（A3，A4)。hmcs 支持多级内存架构，但仅有一种锁实现（A2,A3)，且未在 arm 下验证正确信，而 hmcs-WMM 在 hmcs 的基础上加入内存屏障完成了在 arm 下的正确信。cohort 仅支持两层架构（A1)，但随机两种锁的组合满足了 A2，但未在 arm 上验证正确性。

关键设计 本篇文章分析了不同级别的内存层次结构，如图所示，同一个 cache group 出比 NUMA 节点有更好的性能，在 x86 架构中，同一个 core 上的超线程有更好的性能，在 Armv8 上，NUMA 节点和包之间的差异要比系统和包之间的差异大得多，在 x86 上，跨包时的差异是最不明显的。文章提出的 clof 通过语法

图 7: 目前最先进的 NUMA 感知锁的对四个关键方面覆盖情况

Algorithms	A1	A2	A3	A4
CNA lock [11]	✖	✖	✖	✖
ShflLock [24]	✖	✖	✖	✖
HMCS [6]	✔	✖	✖	✖
HMCS-WMM [33]	✔	✖	✖	✔
lock cohorting [15]	✖	✔	✔	✖
CLoF	✔	✔	✔	✔

✔: covered ✖: lacking

图 8: 不同内存结构下的吞吐量

cohort	x86	Armv8
system	1.00	1.00
package	1.54	1.76
NUMA node	1.54	2.98
cache group	9.07	7.04
core	12.18	–

递归生成器和上下文抽象生成上百种锁，并提出了选择最佳生成锁的自动化方法，详尽计算了所有生成锁的组合，通过一组 N 个基本锁和 M 个内存层次，总共有 N^M 个组合，实际上这个数字仍然较低，进行详尽的评估仍是可行的。在完成基本测试后将锁性能排序，通过不同竞争水平下的吞吐量正向加权和反向加权可以找到倾向于高竞争下和低竞争下性能最好的锁。此外文章通过 VSync 和归纳证明等方法对 shfllock, cna, clof 在 WMM 下做正确性验证。

效果分析 同时满足四种特性，clof 锁在大多数情况下都优于最先进的 NUMA 感知锁。在一个高争用的 levelDB 的测试中，有着 2 倍于 shfllock 和 cna 的吞吐量。

局限性 这项工作强调了根据底层硬件需要不同的锁设计，构造和寻找了性能最好的锁。但目前未能支持 AMP 架构下的大小核，文章声明正在探索将 AMP 上的策略整合进入 clof。

4.9 dsync: 基于组合技术的 cc-sync、h-sync、d-sync

概要 组合技术已经被证实比所有的细粒度锁性能更好，本文重新审视了组合技术，并提出了三种新实现：cc-sync（适用于缓存一致共享内存）、h-sync（cc-sync 的分层版本）、d-sync（适用于分布式共享内存）[14]

关键设计 RMRs（远程内存引用）是指如果访问一个共享内存，他在本地的缓存是无效的，则是一次 RMR。

cc-sync 和 dsm-sync 都在各自适应的内存结构下限制了 RMRs 的次数。

cc-sync: 维护一个列表，为发起活动请求的每个线程加入一个节点，同时维护一个虚拟节点永远指向列表末尾。每个线程发起请求将请求写入节点并插入链表末尾。在列表同步的线程成为 combiner，只有成为 combiner 的线程可以访问共享数据，combiner 首先处理自己的请求，然后顺着列表执行其它线程的请求，并让已经完成请求的线程停止自旋执行后续程序。当其处理完链表的所有请求或执行了 h 个请求后（h 是给定的阈值）。若是后一种情况，则新的链表表头成为新的 combiner 继续执行。

h-sync 是 cc-sync 的分层版本，他针对跨集群的现代多核系统，由于目前最先进的分层方法。

dsm-sync:cc-sync 会在 DSM 模型下有无限制的 RMRs 次数，而提出的 dsm-sync 只需要 O(hd) 个 RMR（其中 d 为服务但个请求需要的 RMRs 数量）。与 cc-sync 不同，当 combiner 发现当前链表仅有少于 2 个的节点便会停止执行，这确保执行一个有限次数的 RMRs。

效果分析 1. 对所执行的远程内存引用 (RMRs) 的数量提供了限制, 2. 支持更强的公平概念，3. 相比以前的方法，使用更简单更少的基本原语。4. 新实现的几种组合技术性能远远超过了之前所有最先进的基于组合和细粒度同步算法，且更容易编程。

局限性 通过使用 fifo 队列来改进 flat-combining 方法，实现了全局锁，并可以存储线程请求的工作。在高竞争中，组合方法优于细粒度锁，但会带来额外的内存管理和开销。

4.10 sanl：本地锁和代理锁自适应切换的 NUMA 感知锁

概要 sanl[31] 结合了本地锁和远程锁的优点，并针对 NUMA 环境进行了优化。

核心观察 在竞争层度较小时，代理锁所带来的通信开销覆盖了所带来的收益，且在 NUMA 环境下仍会出现频繁的跨节点开销。当在临界区中的时间占总程序时间比较少的时候：利用本地锁性能比较好，当在临界区中的时间占总程序时间比较多的时候：利用代理锁比较好，本地锁和代理锁仅在一定的配置下有着良好的性能。

关键设计 sanl 可以灵活地在本地锁和代理锁之间切换，每个线程计算本地的竞争层级，并根据本地竞争层级计算全局竞争层级，如果较小则选择本地锁，较大选择代理锁，结合了两种锁的优点。在代理锁下，针对 NUMA 架构做了以下优化：1. 所有的线程均 trylock，成功获取锁的线程成为第一代服务器，没有获取锁的成为客户端。2. 客户端对于当前锁的竞争层级进行评估（应该还包括了与当前服务器传输延迟等）。若小于阈值 x，则进入自由模式。若大于阈值 x，则进入限制模式。3. 当处于自由模式时，无论服务器节点是否处于本地，均会向服务器发送请求。4. 当处于限制模式时，有两种情况, 若服务器节点在本地则可以直接发送请求到服务器。若服务器节点不在本地则等待，直到竞争层级下降至某个阈值时，转换为自由模式，并向服务器节点发送请求，或者当服务器和本身在同一个节点上，就可以发送或者处理请求了。

代理锁同时具有针对饥饿的解决方法，可能面对的饥饿情况为以下两种：第一种是服务器节点的饥饿：需要不断处理别的节点的执行临界区请求，导致无法执行自己的代码，第二种是远处节点的饥饿：一直轮不到远处节点执行请求，导致饥饿，针对这两种饥饿，sanl 设置了最长服务器时间，以及最短服务间隔时间。

效果分析 在 40 核 Intel 机器和 64 核 AMD 机器上使用四个流行的多线程应用程序（Memcached、Berkeley DB、Phoenix2 和 SPLASH-2）对其进行了评估。与其他七种代表性锁定方案的比较结果表明，sanl 在大多数争用情况下都优于它们。例如，在一组测试中，sanl 比 rcl lock 快 3.7 倍，比 POSIX mutex 快 17 倍。

局限性 结合了本地锁和代理锁的优点，在绝大多数情况下都有着不错的性能，但在低争用的情况下，由于较高的管理成本，有着次优的性能。

4.11 ffwd: 追求效率的代理锁性能有多快

概要 本研究提出了快速、轻量级的代理锁实现，针对低延迟和高吞吐量进行了高度优化。[30]

核心观察 传统的代理锁对遗留应用的支持。细粒度锁需要大量对于现有的数据结构，环境的分析。但是传统代理锁只需要比较少的修改就可以实现支持。但是 rcl 在设计的时候，很大一部分就是考虑到对遗留应用的支持，使其必须支持 CS 区域中的 sleep，对锁的调用等操作。且由于考虑到移植代码（auto Rewrite tools）的便利性，rcl 的 request context 是比较低效的（读请求和读 CS 区域是分开的）。

代理锁实际能够更快，提出了 ffwd，其更加追求效率，而不是对于遗留应用的支持。

关键设计

只允许系统中的某一个核心访问共享资源，我们称其为该资源的管理者。而运行在其他核心上的线程需要访问或修改共享资源时，这些核心需要将访问或修改的请求发送到该资源对应的管理者，并由其代理执行需要进行的访问或修改操作。并通过将指令级并行和仔细管理内存访问相结合，有效地隐藏处在服务器和客户线程之间的互联的高延迟来达到这种性能。

单线程吞吐量限制了委托的吞吐量上限，同时互连带宽，互连延迟，存储缓冲区容量和消息解编组开销会限制委托系统的性能。ffwd 做了以下改进。1. 每个核都有一个 128byte 的缓存行，每核可写，Server 可读，用于发送请求, 解决了请求缓存行的竞争。2.Server 将 15 个 responce 放在一个 buffer 里面一次写。充分利用了缓存行，几乎没有缓存通信开销。3. 是 NUMA 感知的, 在 NUMA 环境下将请求行放在本地的 NUMA 节点，响应行放在服务器的 NUMA 节点，提供了巨大的性能优势。4. 避免在服务器中使用原子指令，允许重排序，根据客户端接近度将请求和响应打包到缓存行对中，以最小化缓存一致性流量等等。

效果分析 每个请求的服务器端开销为 40 个 cycle，非常接近单线程的性能。

局限性 ffwd 展示了代理锁到底能有多块，但是实际应用限制大，更多的是理想状态，如 ffwd 的 cs 函数只支持非阻塞，6 个 8byte 的参数。

5 amp 架构下的同步锁策略

本小节锁策略在 amp 架构下可扩展，目前的工作只有 libasl 在 amp 架构下展示了良好的可扩展性。

5.1 libasl：AMP 架构下的可扩展锁

概要 追求能源效率的非对称多核处理器（AMP）正在普及。然而现有的锁无法在 AMP 架构下可扩展，可能导致吞吐量和时延的崩溃。为解决这个问题本文提出了 AMP-aware 的 libasl，根据应用程序的延迟需求提供了一种新的锁排序方法，它允许大核与小核重新排序，从而在保留应用程序的延迟需求的条件下获得更高的吞吐量。[19]

关键设计 libasl 维持一个 mcs 队列，在大核上的线程拿锁时直接进入 mcs 队列排队拿锁，在小核上的线程拿锁时等待，直到等待了乱序窗口（指 reorder_window）长的时间或 mcs 队列中没有线程。同时应用程序需要在拿锁前和放锁后插入 epoch_start 和 epoch_end，其中 epoch_start 函数，记录从进入临界区到离开临界区的时长。epoch_end 接收给定的最高时延，若当前线程处于大核，直接返回，否则调节乱序窗口，若第二点中记录的时长小于最高时延，提高乱序窗口，否则降低乱序窗口大小。

效果分析 在各种基准中评估 libasl，包括 Apple M1 上的五个流行数据库。评估结果表明，libasl 可以将吞吐量提高 5 倍，同时精确地保留应用程序指定的尾延迟。

局限性 libasl 仅针对 AMP 架构，无法在 NUMA 下可扩展。此外 libasl 需要应用程序指定时延，可能是实际场景下无法接受的。

6 锁相关的技术

本节的内容并没有直接提出锁策略，但均对锁的设计与优化带来巨大影响。

6.1 pilot: ARM 架构弱内存模型中的内存屏障优化

概要： ARM 架构处理器平台使用弱内存模型，同步原语需要使用硬件内存屏障保证程序正确运行。本研究发现其中紧跟远程内存访问之后的硬件内存屏障会导致十分严重的性能问题。因此设计了新型同步原语改善该问题 [20]。

核心观察：在传统互斥锁实现中，临界区均在获取锁的核心本地执行，因此其在放锁时需要添加硬件内存屏障来保证临界区内对于全局变量的访存操作在互斥锁释放之前全局可见。该硬件内存屏障由于紧跟远程内存访问之后出现，因此造成巨大开销，导致性能可扩展性问题。代理锁将所有临界区都在同一个核心上运行，这些临界区内的全局变量的访存操作也均在一个核心中发生。此时硬件可以保证这些访存操作之间的顺序，因此无需额外的硬件屏障。然而，仅让所有临界区在一个核心上远端执行（与之前已有代理锁相似）也存在紧跟远程内存访问的硬件内存屏障：执行临界区的核心需要使用硬件内存屏障来保证对于临界区执行结果的写操作（远程内存访问）与通知临界区执行结束的标记位操作之间的顺序。而该内存屏障也会造成十分严重的性能开销。

关键设计：本研究提出了一种新的机制 Pilot 来避免该内存屏障。Pilot 将标记临界区执行结束的任务承载到写临界区的执行结果之上，因此核心只需要执行一次写操作，无需使用内存屏障。具体而言，当每次执行临界区返回值不相同时，执行临界区的核心通过设置新的返回值即可表示该临界区执行结束。而当临界区返回值相同时，Pilot 将回退到之前使用硬件内存屏障和额外标记位的实现上。因此，Pilot 可以消除互斥锁中十分影响性能的远程内存访问之后的硬件内存屏障，提升互斥锁传递关键路径的性能。

效果分析：本工作将 Pilot 应用在现有的代理锁中，带来显著的性能提升。相比于 FFWD 与 DSynch，Pilot 在队列测试集中能提升 20% 与 26%，在栈测试集中能提升 30% 与 16%。

局限性：Pilot 机制主要适用于采用弱内存机制的架构上，且需要该架构提供单拷贝原子性。此外，Pilot 机制优化的代理锁同样需要对现有应用进行大量修改。

6.2 SynCord 不重新编译或重新启动内核的情况下修改内核锁

概要 内核锁对于应用程序的性能和正确性都至关重要。然而内核锁离应用程序开发人员很远，他们无法影响内核行为，应用程序往往无法与快速开发内核锁相适应。为解决以上问题，本篇文章设计了 SynCord 框架，可以在不重新编译或重新启动内核的情况下修改内核锁。[27]

核心观察 文章发现不同的应用程序在不同的底层系统机制下有最好性能，随着多核、异构硬件的发展，针对不同场景特定优化的锁被不断提出，以 cna 和 shfllock 为代表的前序工作允许通过以策略的形式抽象出硬件和软件的要求来设计新的锁算法。然而通用的机制虽然能在不同的环境下提供可接受的性能，但与专用机制的性能往往差一个数量级甚至更多。同时在 OS 中，同步原语作为内核的一部分，很难动态改动，内核开发人员倾向采用更通用的锁。因此同步原语应该更容易更改，甚至是在运行中也可以修改。

关键设计 SynCordD 的三个主要目标如下：1. 正确性，保证用户定制后的内核锁正确性不会受到影响；2. 隔离性，防止用户定制代码导致内核崩溃；3. 易用性，保证框架能够提供丰富的语义以满足多种用户、平台的需求。

SynCord 的实现逻辑如下：1. 用户编写自定义锁代码并指定要定制的锁实例；2. SynCord 基于 eBPF 编译用户的锁代码；3. 验证编译后的字节码的基本属性以保障正确性；4. 如果验证失败，SynCord 会通知用户；5. 否则会将字节码加载到内核中并生成相应的补丁；6. 最后用户将补丁打入内核，这样就可以通过预定义的钩子节点调用用户定制的锁代码了。

效果分析 为了评测 SynCord 在不同锁应用场景中的效果，作者使用 SynCord 框架针对这些场景实现了相应优化并与前序工作的性能进行了对比。以非对称多核处理器（AMP）场景为例，作者基于前序工作 LibASL 的设计。在 SynCord 中实现了相应优化，使用 SynCord 框架的实现在多种场景下的性能达到了与 libasl 原有实现接近的水平。

局限性 该文章受 libasl 和 clof 启发，提出了一套框架，仅需要修改提供的 API，将锁策略根据 API 接口写入，就可以快速的修改内核锁并正确地执行不同的策略。但是目前仅局限于通过 C 语言编写自定义锁代码。

7 异构众核下的同步锁策略

7.1 sync-cord: 在异构众核下的简单策略

关键设计 sync-cord 中，提出了一个在异构众核下可扩展的策略。在 amp-aware 方面，采取类似 libasl 的方法，区别在小核在拿锁前等待一个固定的时间（10ms），同时每一个线程有一个 NUMA 节点的 id。线程在拿锁时先判断是否在大核上，如果不在则等待一段时间后进入等待队列，如果在则直接加入等待队列中。而在等待队列中执行简单的 NUMA-aware 的队列锁策略（如 cna 和 shflock)。

局限性 1. 需要遍历队列，无法高效获取当前竞争者信息。2. 在高竞争的情况下，重新排序方法无法正确执行大核偏向（无法在关键路径上避免传递到小核），从而导致性能下降。

参考文献

[1] T.E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[2] Marc Auslander, David Edelsohn, Orran Krieger, Bryan Rosenburg, and Robert Wisniewski. Enhancement to the mcs lock for increased functionality and improved programmability, October 23 2003. US Patent App. 10/128,745.

[3] Naama Ben-David, Guy E. Blelloch, and Yuanhao Wei. Lock-free locks revisited. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 278–293, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High performance locks for multi-level numa systems. *SIGPLAN Not.*, 50(8):215–226, jan 2015.

[5] Milind Chabbi and John Mellor-Crummey. Contention-conscious, locality-preserving locks. *SIGPLAN Not.*, 51(8), feb 2016.

[6] Jonathan Corbet. Mcs locks and qspinlocks, 2014.

[7] Travis Craig. Building fifo and priorityqueuing spin locks from atomic swap. Technical report, Technical Report TR 93-02-02, Department of Computer Science, University of ⋯, 1993.

[8] Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, and Haibo Chen. Clof: A compositional lock framework for multi-level numa systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 851–865, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Dave Dice. Malthusian locks. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 314–327, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Dave Dice and Alex Kogan. Compact numa-aware locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-combining numa locks. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 65–74, New York, NY, USA, 2011. Association for Computing Machinery.

[12] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 247–256, New York, NY, USA, 2012. Association for Computing Machinery.

[13] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. Unlocking energy. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 393–406, Denver, CO, June 2016. USENIX Association.

[14] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 257–266, New York, NY, USA, 2012. Association for Computing Machinery.

[15] Free Software Foundation. The gnu c library, 2018.

[16] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 355–364, New York, NY, USA, 2010. Association for Computing Machinery.

[17] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 586–599, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable NUMA-aware blocking synchronization primitives. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 603–615, Santa Clara, CA, July 2017. USENIX Association.

[19] Nian Liu, Jinyu Gu, Dahai Tang, Kenli Li, Binyu Zang, and Haibo Chen. Asymmetry-aware scalable locking. *CoRR*, abs/2108.03355, 2021.

[20] Nian Liu, Binyu Zang, and Haibo Chen. No barrier in the road: A comprehensive study and optimization of arm barriers. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 348–361, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive Reader-Writer locks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 219–230, Philadelphia, PA, June 2014. USENIX Association.

[22] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 6, USA, 2012. USENIX Association.

[23] Victor Luchangco, Dan Nussbaum, and Nir Shavit. A hierarchical clh queue lock. In *Proceedings of the 12th International Conference on Parallel Processing*, Euro-Par'06, page 801–810, Berlin, Heidelberg, 2006. Springer-Verlag.

[24] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, feb 1991.

[25] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, feb 1991.

[26] Jonas Oberhauser, Lilith Oberhauser, Antonio Paolillo, Diogo Behrens, Ming Fu, and Viktor Vafeiadis. Verifying and optimizing the hmcs lock for arm servers. In *Networked Systems: 9th International Conference, NETYS 2021, Virtual Event, May 19–21, 2021, Proceedings*, page 240–260, Berlin, Heidelberg, 2021. Springer-Verlag.

[27] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed kernel synchronization primitives. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 667–682, Carlsbad, CA, July 2022. USENIX Association.

[28] Zoran Radovic and Erik Hagersten. Hierarchical backoff locks for nonuniform communication architectures. *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 241–252, 2003.

[29] David P. Reed and Rajendra K. Kanodia. Synchronization with eventcounts and sequencers. *Commun. ACM*, 22(2):115–123, feb 1979.

[30] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 342–358, New York, NY, USA, 2017. Association for Computing Machinery.

[31] Mingzhe Zhang, Francis C. M. Lau, Cho-Li Wang, Luwei Cheng, and Haibo Chen. Scalable adaptive numa-aware lock: Combining local locking and remote locking for efficient concurrency. *SIGPLAN Not.*, 51(8), feb 2016.