


# Lecture12 Synchronization in Multicores

## 1. 多核与同步原语

### 互斥锁微基准测试

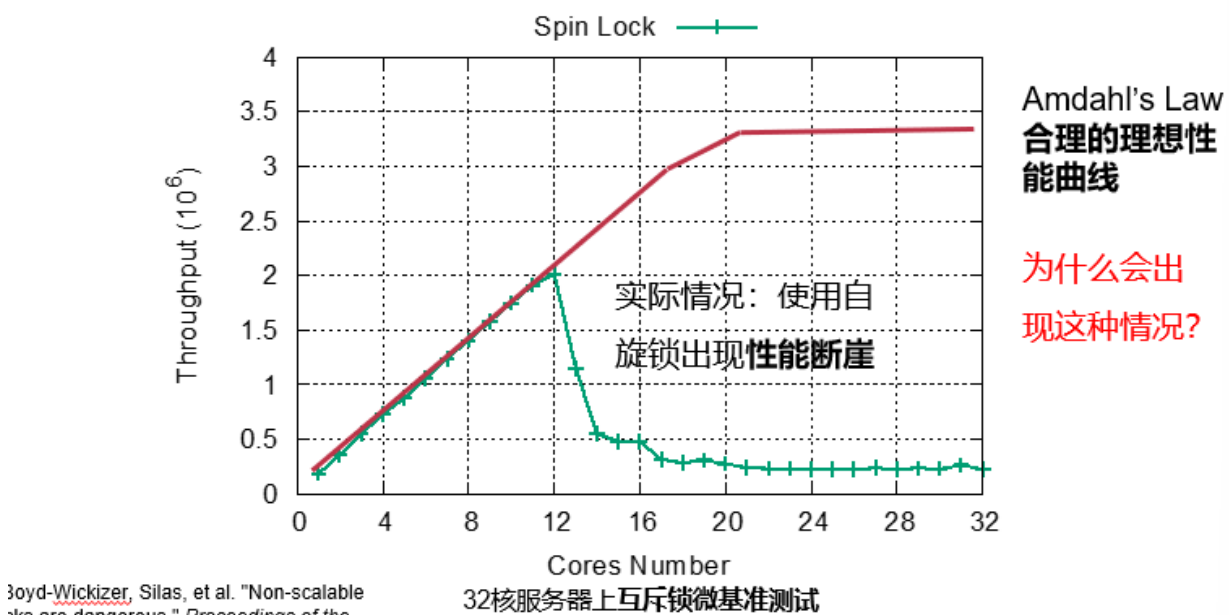
```
void *thread_routine(void *arg) {
    while(1) {
        lock(glock);
        gcnt = gcnt + 1;
        visit_shared_data(shared_data, 1);
        unlock(glock);
        interval();
    }
}
```

 **临界区**

使用**微基准测试**来复现这个现象

记录**固定时间内执行的临界区数量** (吞吐率)

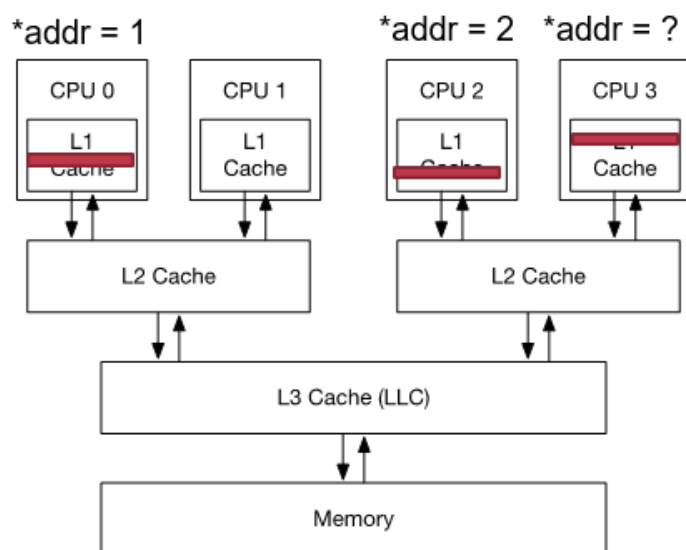
### 但是：可扩展性断崖



## 2. 问题分析：多核环境下的缓存

### 多核环境中的缓存结构

- 多级缓存：
  - 每个核心有自己的**私有**高速缓存 (L1 Cache)
  - 多个核心共享一个**二级**高速缓存 (L2 Cache)
  - 所有核心共享一个**最末级**高速缓存 (LLC)
- 非一致缓存访问 (NUCA)
- 数据一致性问题

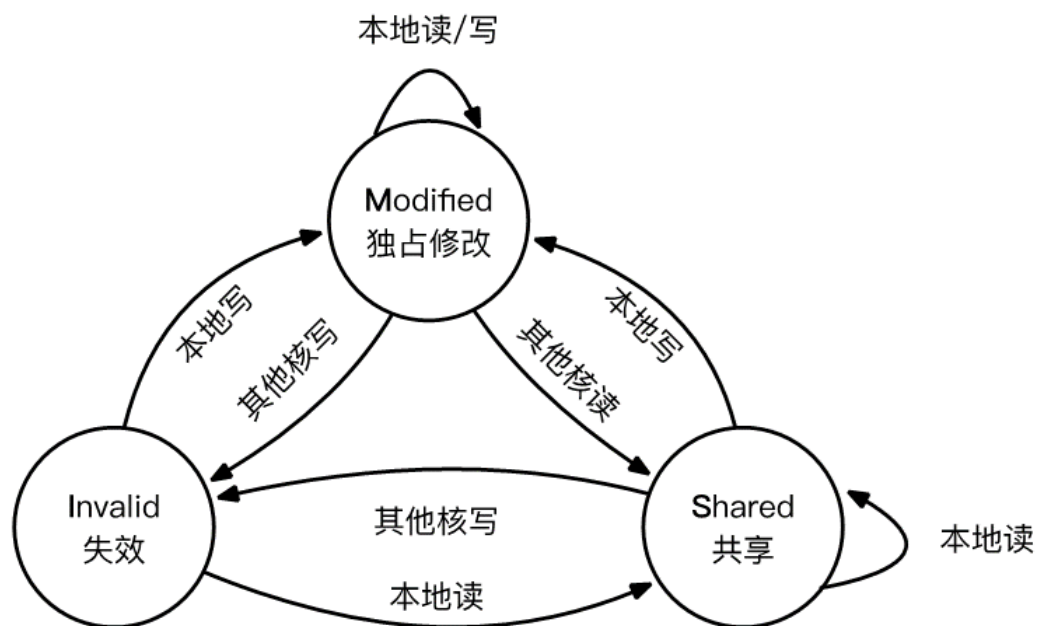


一个典型多核系统高速缓存架构\*

## 缓存一致性

缓存一致性：保证不同核心对同一地址的值达成共识

- 多种缓存一致性协议：窥探式/目录式缓存一致性（今天介绍的）协议
  1. 缓存行处于不状态(MSI状态)
  2. 不同状态之间迁移
  3. 所有读/写缓存行操作遵循协议流程
- MSI状态迁移

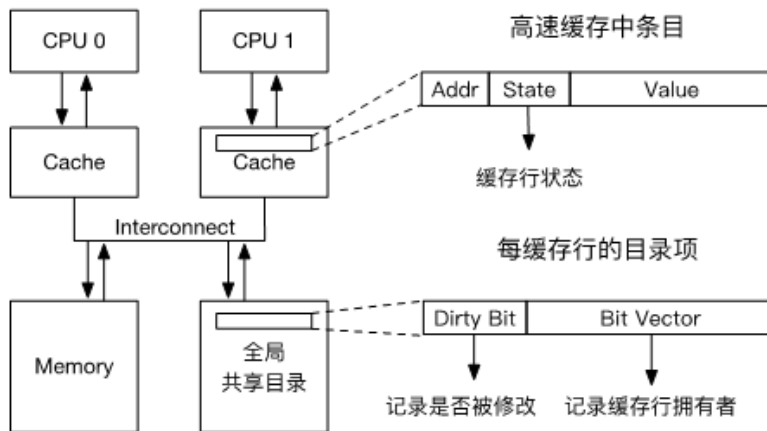


1. 独占修改(Modified)
  1. 该核心独占拥有缓存行
  2. 本地可**读**可**写**

3. 其它核**读**需要迁移到**共享**
  4. 其它核**写**需要迁移到**失效**
2. 共享(Shared)
    1. 可能有多个核同时拥有缓存行的拷贝
    2. 本地可**读**
    3. 本地**写**时需要迁移到**独占修改**，并使其它核该缓存行**失效**
    4. 其它核**写**需要迁移到**失效**
  3. 失效(Invalid)
    1. 本地缓存行失效
    2. **本地不能读/写**缓存行
    3. 本地**读**需要迁移到共享，并使其它核该缓存行**失效**（原文：啊很难过迁移到共享）
    4. 本地**写**需要迁移到独占修改，并使其它核该缓存行**失效**
- **全局目录项**

### 如何通知其他核心需要迁移缓存行状态？

全局目录项：记录缓存行在不同核上的状态，通过总线通讯



## 可扩展性断崖的原因

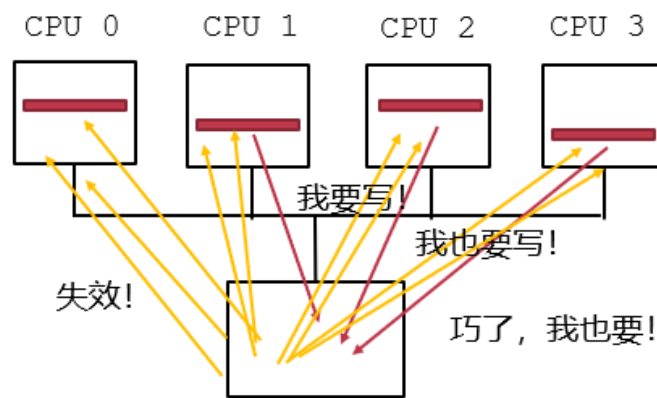
```

void lock(int *lock) {
    while(atomic_CAS(lock, 0, 1)
        != 0)
        /* Busy-looping */;
}

void unlock(int *lock) {
    *lock = 0;
}

```

自旋锁实现



全局目录项

Lock所在缓存行状态

对**单一缓存行**的竞争导致**严重的性能开销**

### 3. 解决可扩展性问题: MCS锁

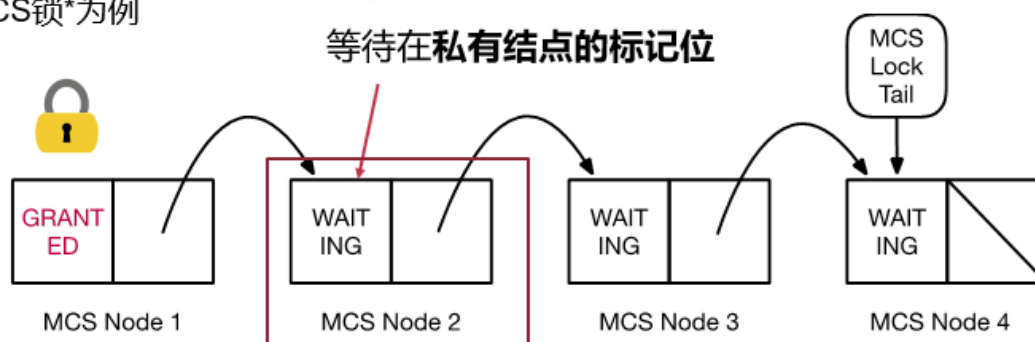
核心思路: 在关键路径上避免对单一缓存行的高度竞争

核心思路: 在**关键路径上**避免对单一缓存行的高度竞争

以MCS锁\*为例

需要获取锁的竞争者拥有一个**私有节点**

等待在**私有节点**的标记位

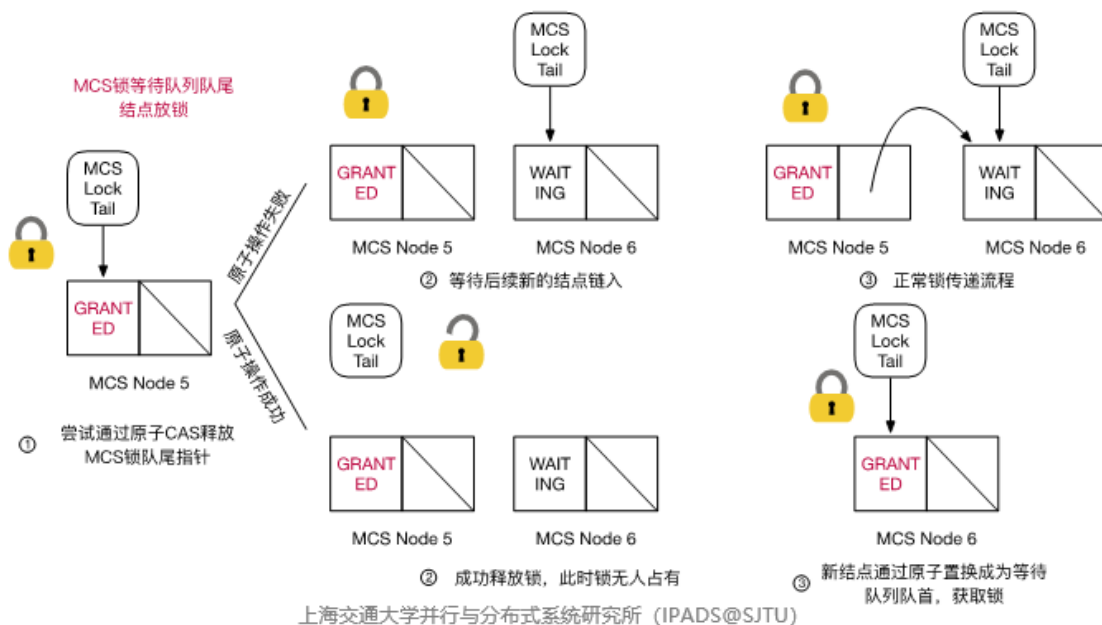


每节点单独缓存行

通过 **等待队列** 串起所有需要持锁竞争者节点

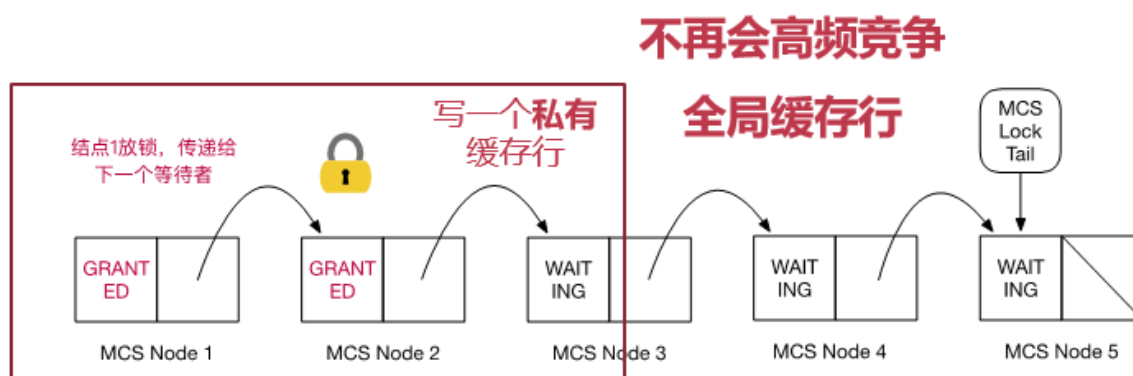
Wellor-Crummey J M, Scott M L. Algorithms for

放锁流程



47

## 性能分析

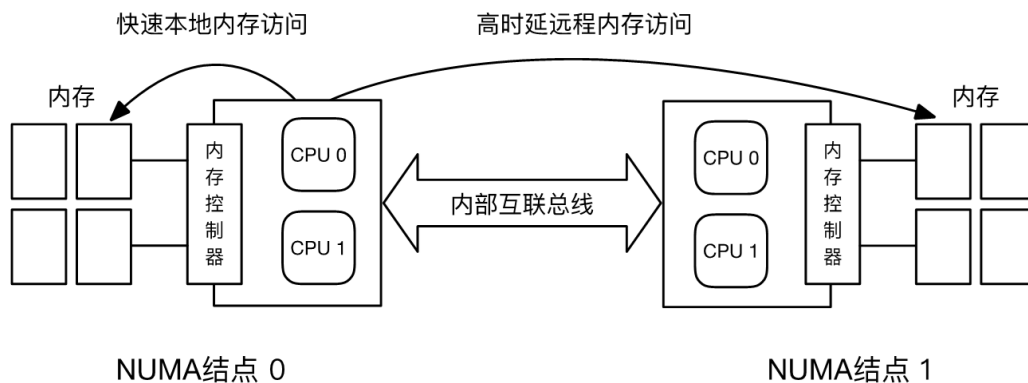


### 高竞争程度时的关键路径

这时修改的是**私有缓存行**的内容, 只会invalidate掉**下个节点**的上层Cache, 然后下个节点所在的CPU就会向上一个节点所在CPU发送最新的值

## 4. 非一致内存访问 (NUMA)

访问不同位置的内存速率不一样 (受物理距离/光速的限制)



避免单内存控制器成为瓶颈，减少内存访问距离

常见于多处理器（多插槽）机器 单处理器众核系统也有可能使用，如Intel Xeon Phi

## NUMA环境中新的挑战

临界区中访问的共享数据可能需要经过多跳才能访问到

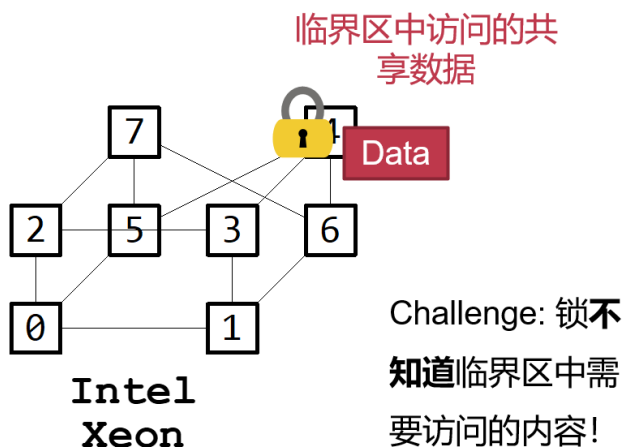
```
while(TRUE) {
```

申请进入临界区

临界区部分

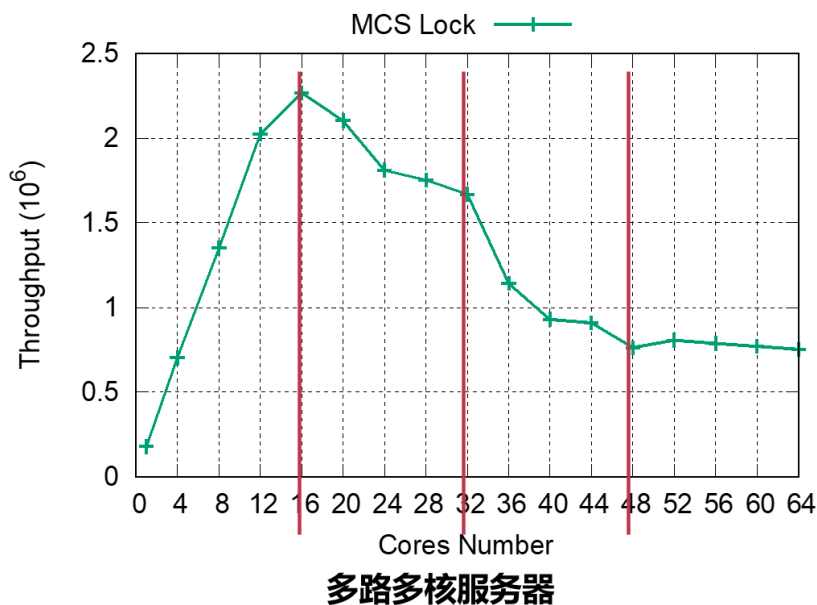
通知退出临界区

其他代码



**跨结点的缓存一致性协议开销巨大**

## MCS锁可扩展性



available: 4 nodes (0-3)  
 node 0 cpus: 0 1 2 3 4 5  
 6 7 8 9 10 11 12 13 14  
 15  
 node 1 cpus: 16 17 18 19  
 20 21 22 23 24 25 26 27  
 28 29 30 31  
 node 2 cpus: 32 33 34 35  
 36 37 38 39 40 41 42 43  
 44 45 46 47  
 node 3 cpus: 48 49 50 51  
 52 53 54 55 56 57 58 59  
 60 61 62 63  
 node distances:  
 node 0 1 2 3  
 0: 10 15 20 20  
 1: 15 10 20 20  
 2: 20 20 10 15  
 3: 20 20 15 10

Scalability较差，每增加16个核性能都会下降

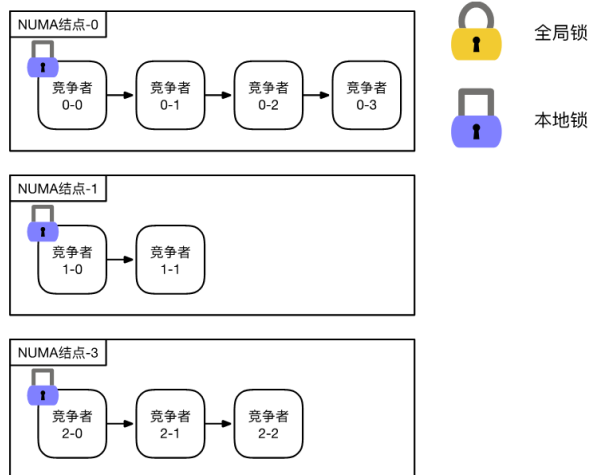
## 5. NUMA-AWARE: COHORT锁

核心思路：在一段时间内将访存限制在本地

### COHORT锁

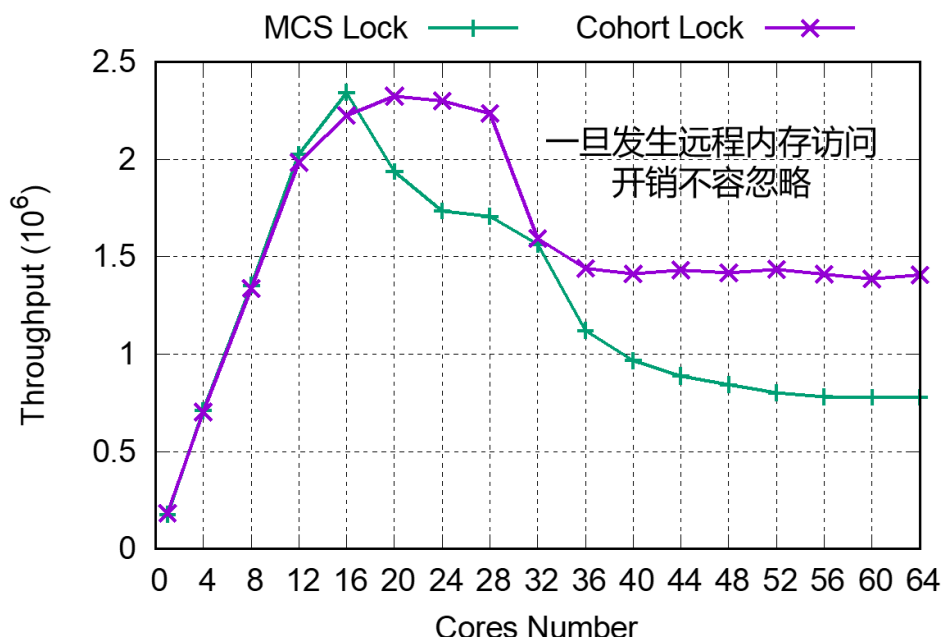
核心思路：在一段时间内将访存限制在本地

先获取**每结点本地锁**  
 再获取全局锁  
 成功获取全局锁  
 释放时将其传递给  
**本地等待队列的下一位**  
 全局锁在一段时间内  
**只在一个结点内部传递**



1. 在一个NUMA节点中使用MCS锁
2. 在各个NUMA节点间采用全局锁来协调全局变量的访问权限（全局变量的cache在持锁的NUMA节点之下）

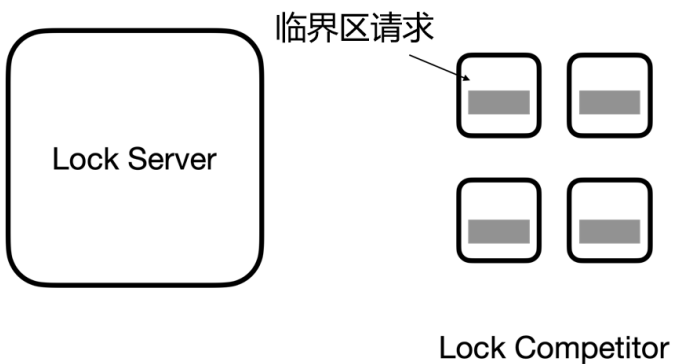
## COHORT锁的性能表现



32核之后性能还是会急剧下降，主要原因还是会涉及跨NUMA的内存访问

## 6. NUMA-AWARE: 代理锁

通过代理执行避免跨节点访问 -> 一直将缓存行限定在一个核心上



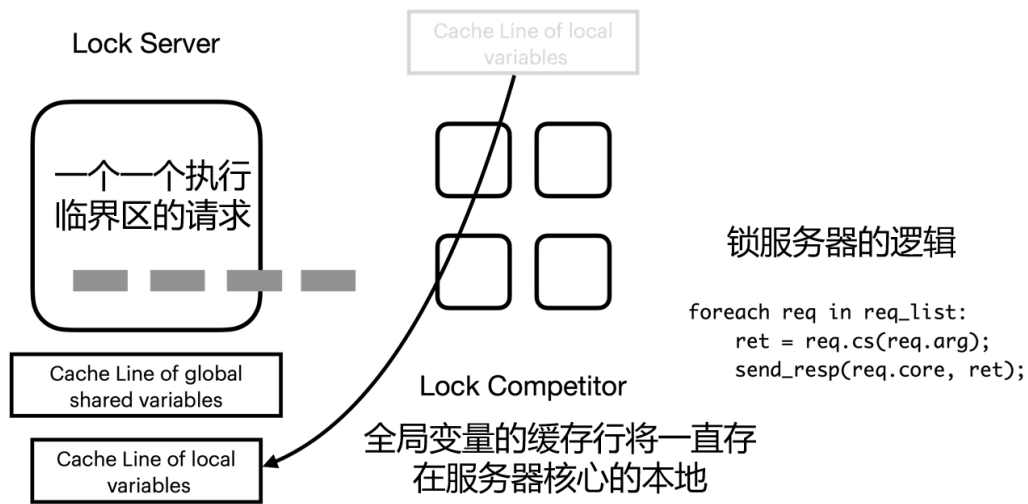
```
lock(&glock);  
global_variable += 1;  
local_variable = global_variable;  
unlock(&glock);
```

↓ 临界区转换为可以远程执行的闭包

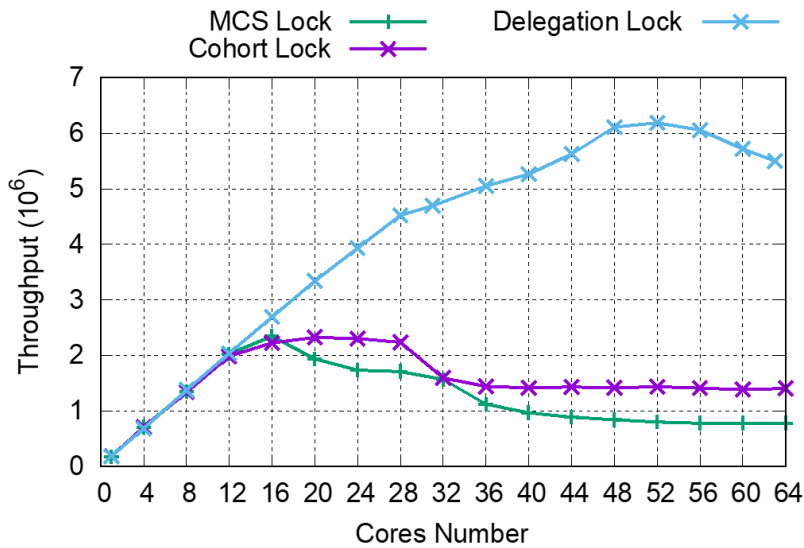
```
void *cs(void *arg) {  
    global_variable += 1;  
    *arg = global_variable;  
    return NULL;  
}
```

```
send_request(cs, &local_variable);
```





## 性能表现



ffwd的性能表现

在扩展到其他NUMA节点时，没有显著的性能的下坠

是否是最优的解？

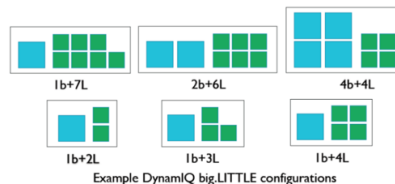
- 需要很多代码修改
- 额外逻辑复杂：锁竞争少时性能差

## 7.非对称多核的可扩展性

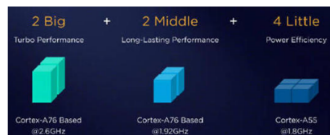
# 同构ISA异构多核系统

- 异构多核系统 (AMP)
- 高性能处理器核心 + 高能效处理器核心
- 适应更多场景的计算需求 (性能+能耗)
- 广泛地使用在移动处理器平台
- 已经运用到桌面平台  
(Intel Alder Lake, Apple M1)
- 调度器 (如EAS) 可以将线程调度到异构核心

## ARM DYNAMLQ 架构



## 麒麟990芯片

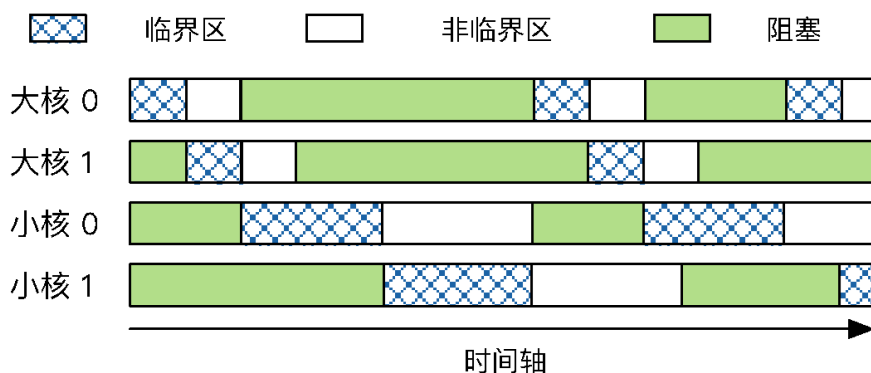


## Intel Alder Lake 架构



## 观测1：不同核处理性能各异，通过获取公平性不再使用

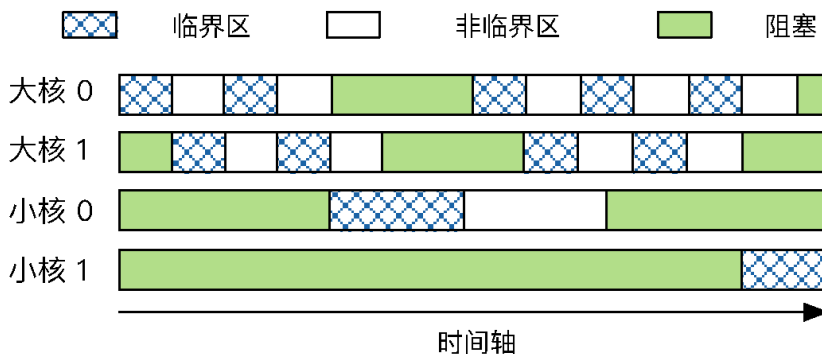
对于保证获取公平性的同步原语 (如Ticket Lock, MCS Lock等)：



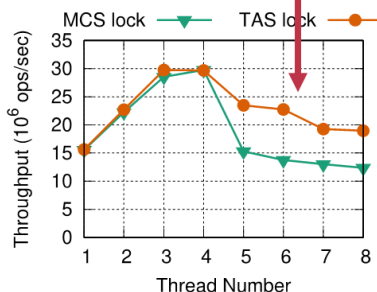
小核执行临界区时间更长，阻塞大核执行，导致**吞吐量**受到影响。

## 观测2：不同核原子操作成功率不同，带来性能问题

对于不保证获取公平性的同步原语 (如TAS spinlock等)：

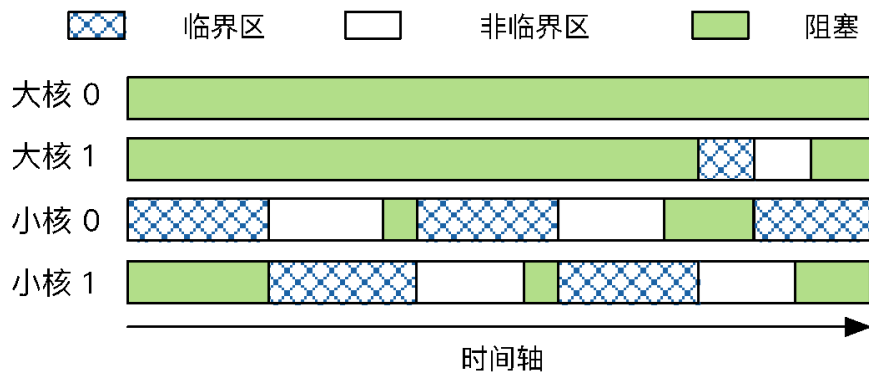


## 另一组测试 TAS吞吐量更优



当大核成功率高 (**大核倾向性**)，小核的**锁延迟**受到较大影响，甚至会出现**饥饿**

对于**不保证获取公平性**的同步原语（如TAS spinlock等）：

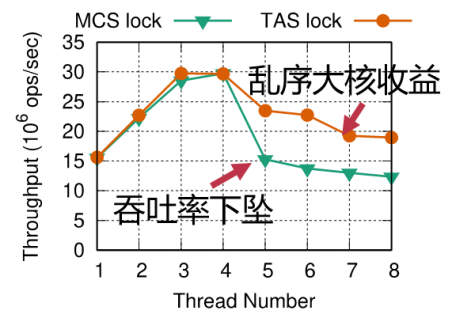


当小核成功率高（**小核倾向性**），**吞吐量**会大幅下降，且大核拥有较长**锁延迟**

## 如何在非对称多核中扩展

启示1：遵循**获取公平性**的**锁传递顺序**无法适用AMP，需要设计适合AMP的锁传递顺序。

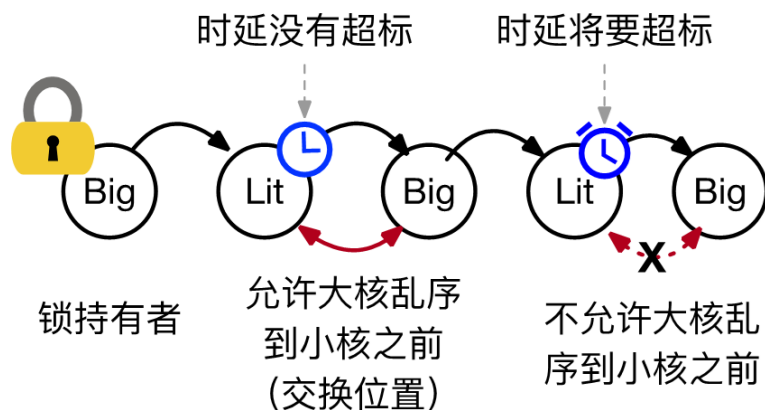
启示2：（针对获取锁的时间顺序）允许大核**乱序**到小核之前拿锁可以有效提升吞吐量，但乱序程度必须**可控**，避免违背应用时延需求。



## 非对称多核感知锁LIBASL

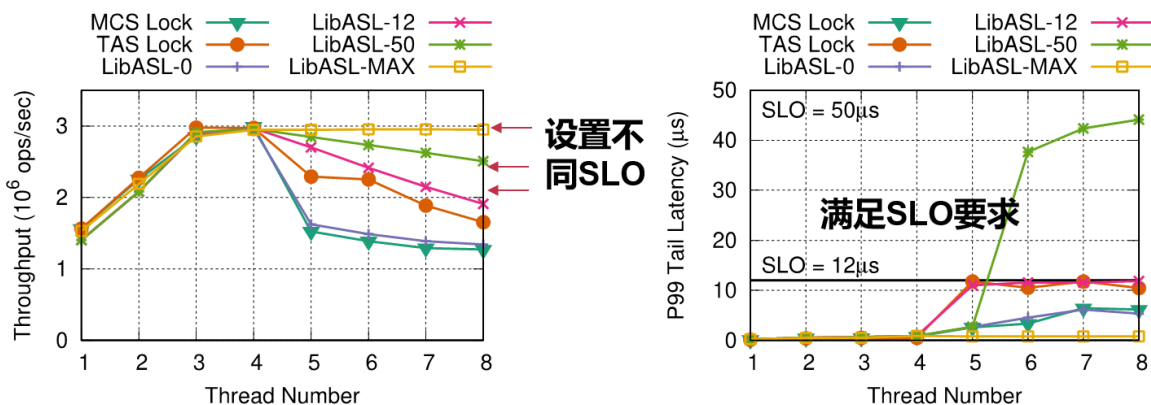
如何解决非对称多核的可扩展性问题：时延需求指导的锁传递顺序

### 具体设计



按照获取锁的时间顺序上 (FIFO)，在不违背小核时延需求的前提下，  
尽可能让大核乱序到小核之前，达到更高的吞吐率

## LibASL的可扩展性



## 吞吐率与时延

Epoch的窗口大小如何指定？

我们设计了一个反馈机制调节。

这是由于乱序窗口实际上是在每次加锁之前固定增加了一段等待时间。

因此在走相同code path的时候，增加乱序窗口的大小，与epoch的总的时延长度线性相关

因此，我们可以通过运行时动态调整来找到合适的窗口大小。

然而，实际上epoch的长度变化可能比较大，需要找到合适的算法来调整，快速响应

## 读写锁的可扩展性

最直观的性能问题：读者需要抢一把全局互斥锁，并对一个全局计数器自增

# 大读者锁

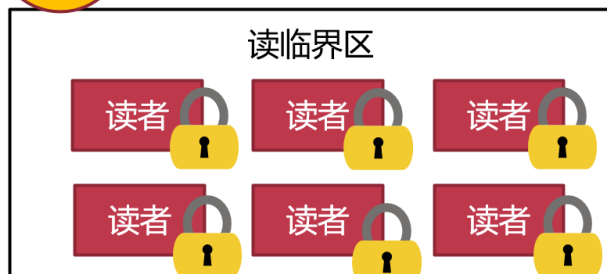
```
struct rwlock {
    unsigned int total_reader = 0;
    struct lock reader_lock[MAX_READER];
    struct lock writer_lock;
};
```

```
void lock_reader(struct rwlock *lock)
{
    lock(&lock->reader_lock[rid]);
}

void unlock_reader(struct rwlock *lock)
{
    unlock(&lock->reader_lock[rid]);
}
```

写者

每个读者将私有一把专属的读者锁：  
进读临界区：直接获取私有的互斥锁



```
__thread int rid; /* 每线程唯一读者编号 */
int rwlock_init_per_thread(struct rwlock *lock)
{
    lock(&lock->writer_lock);
    rid = lock->total_reader++;
    unlock(&lock->writer_lock);
    return rid >= MAX_READER? -1: 0;
}
```

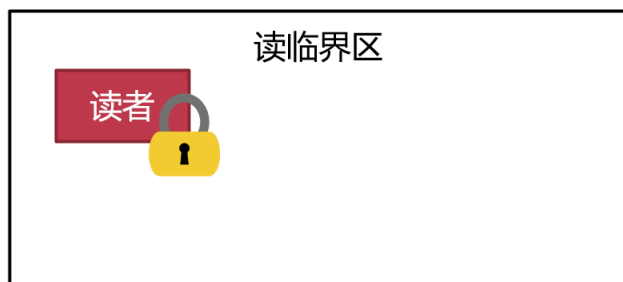
# 大读者锁

写者



```
void lock_writer(struct rwlock *lock)
{
    int i, j;
    lock(&lock->writer_lock);
again:
    for(i = 0; i < lock->total_reader; i++) {
        if (trylock(&lock->reader_lock[i]) == FAIL) {
            for (j = 0; j < i - 1; j++)
                unlock(&lock->reader_lock[j]);
            goto again;
        }
    }
}

void unlock_writer(struct rwlock *lock)
{
    for(int i = 0; i < lock->total_reader; i++)
        unlock(&lock->reader_lock[i]);
    unlock(&lock->writer_lock);
}
```



写者进入临界区：  
需要获取所有的私有  
读者锁  
一旦失败，立刻重试



上海交通大学分布式系统研究所 (IPADS@SJTU)

84

读者关键路径上仍然有上锁操作：涉及原子操作，性能影响仍然较大

与此同时，**写者开销巨大**

# PRWLock

## 进一步减少读者关键路径性能开销

核心思想：通过版本号协同**读者与写者**，读者关键路径上只有**三个本地访存操作**

当写者需要写时更新版本号

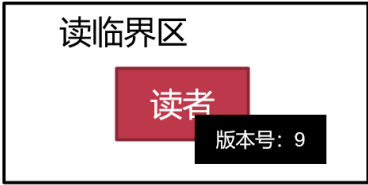


```
lock(writer_lock);
global_ver ++;
```

ps://wn.net/Articles/619355/

alable Read-mostly Synchronization Using Passive Reader-Writer  
cks: Ran Liu, Heng Zhang and Haiho Chen, Usenix ATC, 2014

读者进入/退出临界区时  
更新自己的版本号



```
while (is_locked(writer_lock))
    local_ver[rid] = global_ver
```



确保看到版本号的读者**知道写者的上锁意图，不会进入读临界区**