
Parallel Processing

Lecture: Synchronization

王展

授课提纲

- 共享存储多处理器**Cache**一致性回顾
- 共享存储多处理器架构下的同步问题
- 事务内存（**Transactional Memory**）

共享存储多处理器**CACHE**一致性回顾

回顾上节课关于SMP cache一致性的讨论

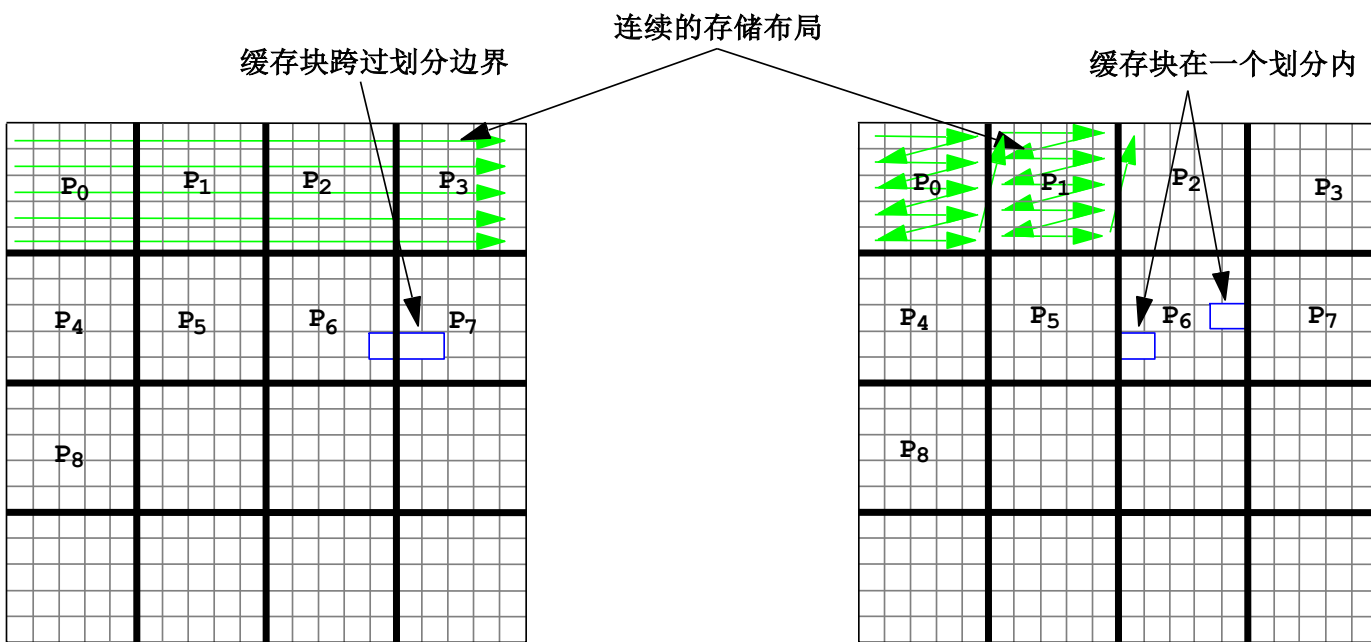
- 共享存储多处理器架构的四种实现形式
- 基于总线（Bus）的SMP架构
- 关于一致性的定义
 - 写传播和写串行化
- 关于存储同一性的定义
 - 串程序的序是什么
 - 顺序同一性的充分条件
- 基于侦听的SMP架构Cache一致性协议
 - Write-Through一致性协议
 - Write-back 基于作废的一致性协议（MSI、MESI）
 - Write-back 基于更新的一致性协议（MSI、MESI）
 - 不同协议的实现选择
- SMP上的编程框架：OpenMP的基础知识

探究高速缓存一致多处理器架构对软件的影响

- 之前我们一致关注的是工作负载对体系结构和协议的影响
- 一些负载均衡、通信优化等方法在不同体系结构上是通用的，这里我们主要关注**cache**一致性多处理架构的一些软件设计原则
 - 例如在对计算任务分配时，力图只有一个处理器对一组数据做写操作（至少在单个计算阶段中）
 - 例如在图像处理中通过计算任务和数据结构划分尽可能避免共享写
- 通信和映射的结构不是主要问题
- 关键问题是利用时间局部性和空间局部性来减少高速缓存扑空
 - 减少时延、通信量和共享总线争用
 - 时间局部性: 让工作数据集尽可能放到高速缓存中
 - 空间局部性: 减少存储碎片（将不必要的数据取到了**cache**中）和伪共享

利用好空间局部性的一些方法

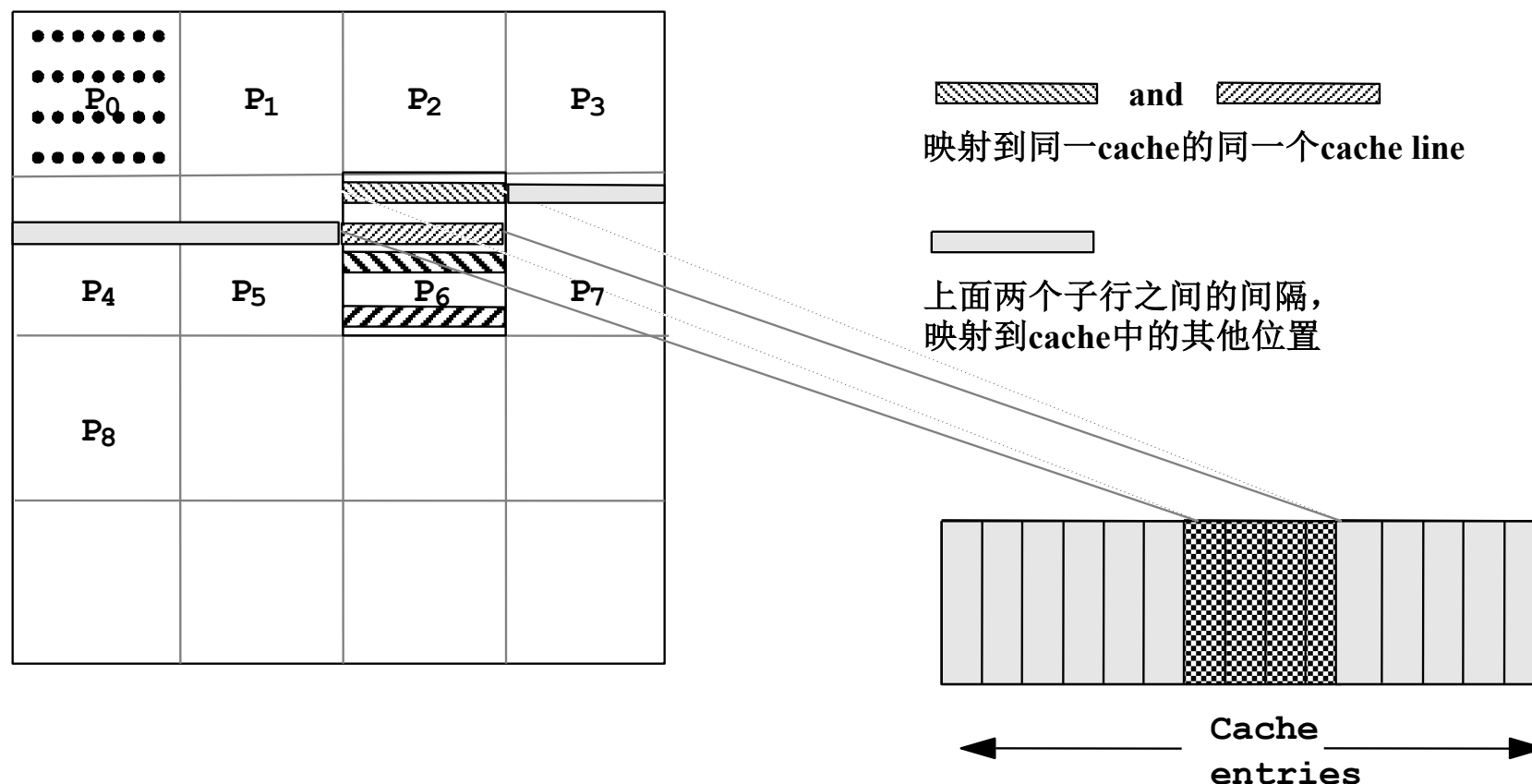
- 任务的分配应减少访问模式的空间交叉
 - 让处理器尽可能去访问一片相邻的区域，例如 n 个数组给 p 个处理器，每个处理器访问相邻的 n/p 个元素
- 组织数据以减少访问模式的空间交叉
 - 用高维数组使一个数组在地址空间上是连续的（下图的4维数组）
 - 减少伪共享和存储碎片以减少冲突产生的扑空（conflict misses）



(a) Two-dimensional array

(b) Four-dimensional array

二位数组表示下的高速缓存冲突问题



- 相继的两个子行之间的间隔刚好等于cache大小
- 当数组和cache容量大小都是2的幂次时问题会更加严重

还有一些方法(contd.)

- 防止更一般的冲突扑空 (**Conflict Miss**)
 - 高速缓存的大小通常是2的幂次, 当数组的存储分配规模也是2的幂次时冲突会更严重
 - 当映射冲突发生在不同的数据结构之间时: 利用随意填充和对准来缓解
 - 要特别注意映射冲突发生在一些看似无害的共享变量和数据结构上, 非常隐蔽
- 每个处理器都使用单独的堆 (尽量不要让不同处理器使用的数据在同一块缓存)
- 通过数据拷贝来提高空间局部性
 - 如果需要反复用到一组本来不是连续分配的数据, e.g. blocks in 2D-array LU
 - 需要在性能和开销之间权衡
- 填充数组: 缓存的一个块可能容纳数组 (比如进程标识号数组) 的大部分元素, 造成伪共享, 通过填充消除伪共享, 但又会引入存储碎片问题
- 如何组织记录数组突出空间局部性
 - E.g. 粒子与场: 按粒子特性组织或按场特性组织
- 局部性和附加通信问题比固有数据通信对性能更重要
 - 可能使我们为了某个应用而重新考虑算法的划分决策 (e.g. strip v. block in grid)

SMP总结

- **SMP是单处理器的自然延伸**
 - 是实现并行化的一条自然路径
 - 细粒度的多道程序设计和操作系统共享
- **关键的技术挑战是扩展内存架构的设计**
 - 即使在逻辑层，协议和总线也有诸多设计抉择（具体硬件实现会有更多抉择）
- **还会继续重要下去，因为它有：**
 - 极具吸引力的性价比
 - 当前的处理器已经是该架构
 - 相关软件技术成熟
 - 大规模并行计算系统的基本组成单元 (成本摊分)
- **考虑一下协议的具体实现？**

共享存储多处理器架构下的同步问题

同步 Synchronization

- “一台并行计算机就是一组可以相互通信[协作](#)快速解决大型问题的单元的集合。”

(A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast)

- 同步的类型
 - 互斥锁 *Mutual Exclusion*
 - 事件同步 Event synchronization
 - 点对点事件 *point-to-point*
 - 组事件 group
 - 全局事件 *global (barriers)*

历史与展望

- 关于要多少硬件支持以及提供什么样的硬件原语讨论了多年
- 结论随技术工艺和机器的设计风格变化而变化
 - 速度（硬件） vs 灵活性（软件）
- 大多数的现代实现方式都使用某种原子性读-改-写（**atomic read-modify-write**）的硬件实现
 - IBM 370: 在多道程序设计中
使用原子的 `compare & swap`
 - x86: 任何指令都可以用
锁定修饰符作为前缀
 - 高级语言支持希望硬件支持
`locks/barriers`
 - 但是它违背了“RISC”简单指令理念，
并
有其他问题
 - SPARC: 原子的寄存器-存储器操作(`swap, compare & swap`)
 - MIPS, IBM Power: 没有原子操作，但是有指令对可以组合实现
 - `load-locked, store-conditional`
 - later used by PowerPC and DEC Alpha too
- 在设计中需要非常多的权衡

同步事件的组件

- **获取方法 Acquire method**
 - 进程用来获取同步权的方式 (进入临界区或者向前推进)
- **等待算法 Waiting algorithm**
 - 进程用来等待同步可用的方法
- **释放方法 Release method**
 - 进程用来使其他进程推进通过一个同步事件的方法
- 等待算法的选择基本上是独立于同步类型的

等待算法 Waiting Algorithms

■ 阻塞 Blocking

- 等待的进程/线程简单的挂起自己
- 开销较大（挂起和重启一个进程涉及操作系统）
- 但是使处理器能为其他进程或者线程所用

■ 忙等待 Busy-waiting

- 等待进程/线程在一个循环中不断的测试某个变量是否改变了值
- 释放资源的进程改变这个变量的值
- 避免了挂起开销，但是消耗了处理器和缓存带宽资源

■ 在下面几种情况下，选择忙等待更好

- 调度开销（时间）远大于等待时间
- 处理器资源不需要为其他任务所用
- Scheduler-based blocking is inappropriate (e.g. in OS kernel)

■ 混合方法：忙等待一会，如果超出了某个阈值，就阻塞

在同步中，用户和系统的抉择

- 用户（程序员）要用锁、事件或者更高层次的操作
 - 不关心它们的内部实现
- 系统设计者：提供多少硬件支持，哪些功能用软件来实现？
 - 速度 vs. 开销和灵活性
 - 在硬件层面实现等待算法非常困难, 可以考虑对其他组件进行支持
- 较为流行的做法是：
 - 系统提供简单的硬件原语 (atomic operations)
 - 软件库使用这些简单原语实现 lock, barrier 等算法
 - 也有方案提供全硬件的同步实现

好的同步设计所面临的挑战

- 同一种同步，可能在不同的时间用于非常不同的运行条件
 - 对一把锁的访问可能是低冲突的（较少的处理器），也可能是高冲突的（较多的处理器）
 - 不同的性能要求: low latency（低争用） or high throughput（高争用）
 - 不同的算法可以更好的满足不同的需求，不同的算法也可能依赖不同的基本硬件原语
- 多道程序对同步要求更为复杂
 - 涉及进程调度和其他资源交互
 - 和在专用条件下表现出低延迟高带宽的简单算法相比，可能需要更加复杂算法
- 硬件/软件相互作用是设计的焦点
 - 哪些原语可用→可以使用哪些算法
 - 哪些算法是有效的→要提供哪些原语
- 需要基于负载特征来评估

互斥锁：硬件锁的实现方式

- 在总线上用一组**锁线路（lock lines）**：其中每一条线代表一把锁
 - 优先级机制决定下一次该哪个处理器得到这把锁
- **锁寄存器 lock register(Cray XMP)**
 - 一组在不同处理器之间共享的寄存器
- 不灵活，因此不适用于通用场景
 - 在同一时间内可同时使用的锁的数量有限 (one per lock line)
 - 使用固定的硬件实现的等待算法
- 主要为较高级别的软件锁提供原子性支持

简单的软件锁算法

- 问题:锁在它自己的实现中需要原子性
 - 进程对一个锁变量的Read (test) 和 write (set) 不是原子的
- 解决方法: 使用***atomic read-modify-write*** 或原子交换指令
 - 原子的测试某个位置的值并将其设置为另一个值, 返回成功或失败

```
lock:  ld register, location    /* copy location to register */
      cmp location, #0         /* compare with 0 */
      bnz lock                 /* if not 0, try again */
      st location, #1          /* store 1 to mark it locked */
      ret                      /* return control to caller */
unlock: st location, #0        /* write 0 to location */
      ret                      /* return control to caller */
```

两个进程P0和P1同时执行上面的锁操作，都有可能读到变量为0，而进入到临界区（需要提供原子性的区域），两个进程都可以自由对变量写1，锁的效果没有实现

基于原子交换指令的实现方式

- 指定存储单元位置和寄存器:
 - 将存储单元的值（0）读入寄存器
 - 将另一个值（1）写入到该单元中
 - 期间不允许有其他对该单元的访问
- 这种操作可以有很多变形
 - 通过所存储的值的的特点提供不同的灵活性
- 一个简单的例子：给定这样一条指令 **test&set (t&s)**
 - 将存储单元的值读入特定寄存器
 - 将常量1写入该单元
 - 考察寄存器中的值如果是 0，则成功获得了锁
 - 也可以设置其他值来代替典型值1和0

```
lock:   t&s    register, location
        bnz    lock      /* if not 0, try again */
        ret                        /* return control to caller */
unlock: st     location, #0 /* write 0 to location */
        ret                        /* return control to caller */
```

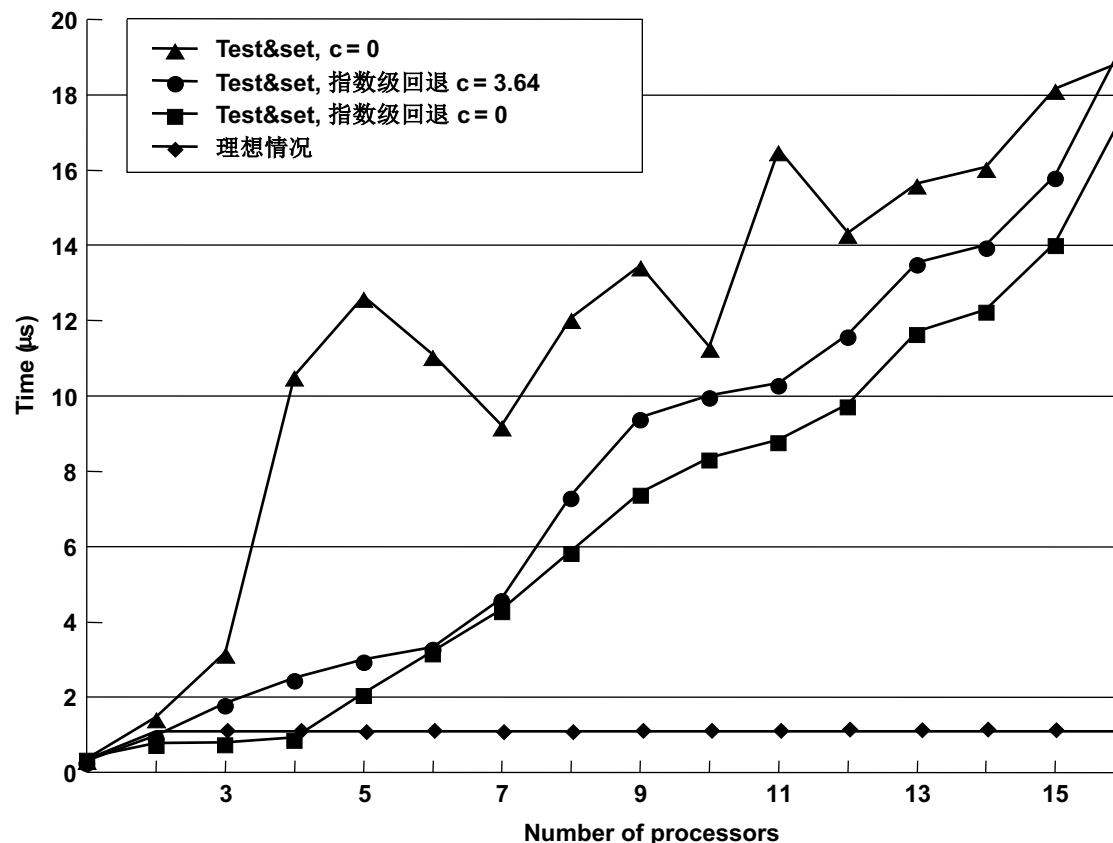
基于原子交换指令的实现方式

- **T&S指令还有一些更加复杂的变形**
 - **Swap**
 - 将特定存储单元的值读入特定寄存器，但将该寄存器开始的任意值写入该存储单元
 - 一般用0和1，保证寄存器的值在交换指令执行前为1
 - **Fetch&op**
 - 将特定存储单元的值读入寄存器，并将op后的值写入该存储单元
 - **Compare&swap**
 - 3操作数指令: location, register to compare with, register to swap with
 - RISC体系结构通常不支持
- 可以是**cacheable**，也可以是**uncacheable** (我们假定**cacheable**)
- 利用该硬件指令来支持软件的互斥结构

T&S Lock 微基准性能测试

On SGI Challenge. Code:

```
lock;  
delay(c);  
unlock;
```



- 每个处理器测试锁是否释放，都产生一个对含有该锁变量的缓存块的写操作，每个写操作产生一个总线事务来作废该块先前的拥有者，所导致的竞争大大减慢了锁的转移，最上面的曲线处理器间争用最激烈

我们对上述简单的锁算法进行增强

- 减少进程在等待时发出**test&set**指令的频率
 - 带有回退 (backoff) 的 *Test&set lock*
 - 不要回退太多, 否则当锁自由了, 处理器还处于空闲状态
 - 实验结果显示, 让回退时间按指数规律变化较好: $i^{\text{th}} \text{ time} = k * c^i$
- 让进程只是在读操作时忙等待, 而不是 **test&set** 指令
 - *Test-and-test* (多次test) & set lock
 - 重复执行一个标准装入指令
 - 在缓存一致性机器中, 所有处理器都可以在缓存中进行这样的读操作, 而不产生总线流量
 - 当锁释放时(value to 0), 在执行 test&set 锁指令来实际试图获取该锁
 - 其中之一将成功, 其他的将失败, 并返回到基于读的等待方法

性能目标(T&S Lock)

■ 低延迟

- 如果一把锁是自由的并且没有其他处理器同时试图获取它，一个处理器应该能以低延迟获得它

■ 低流量

- 如果许多或者所有处理器同时试图获取一把锁，它们应该能够一个接一个得到该锁，尽量不产生流量或总线事务

■ 低存储

- 一把锁所需的信息不应该随处理器数量增加而增加很多

■ 公平性

- 理想上，应该以处理器发出请求的次序获得锁，避免饥饿或明显的不公平

■ **Test&set with backoff** 与基本实现方式性能差别不大, 但流量更小

■ **Test-and-test&set**: 延迟稍高, 但流量减少很多

■ 但都会在锁释放时同时去竞争该锁

- p 个处理器同时竞争产生的流量是: $O(p^2)$

■ 后续又设计了更好的硬件原语和算法实现

改进的硬件原语: Load-Locked, Store-Conditional (LL-SC)

■ 目标:

- 用读操作来测试
- 让读-改-写 (read-modify-write) 失败的进程不产生作废
- 并且能够实现多种r-m-w操作

■ ***Load-Locked (or -linked), Store-Conditional***

- 一对指令来实现对一个变量 (同步变量) 的原子性访问

■ LL 将同步变量装入寄存器

■ 后面可以跟任意指令完成对寄存器中值的操作

■ SC 试图将寄存器的值写回去, 条件是当且仅当没有其他处理器在本处理器LL后, 对该单元 (或者缓存块) 进行写

- 如果SC 成功, 就意味着上述三步的执行是原子的
- 如果失败, 就不会试图将值回写 (或者产生任何作废), 从LL再开始尝试
- 后面我们再讨论LL-SC是如何具体实现的, 现在主要关心语义和性能

LL-SC简单示例

```
lock:      ll      reg1, location    /* LL location to reg1 */
           bnz     reg1, lock        /* if location was locked (nonzero), try again */
           sc      location, reg2    /* SC reg2 into location */
           beqz    lock              /* if failed, start again */
           ret
unlock:    st      location, #0      /* write 0 to location */
           ret
```

- 可以通过改变LL & SC之间的指令来实现更多的原子操作
 - 但是需要让这段指令尽可能小来保证SC的更容易成功
 - 不要包括那些需要撤销的指令(e.g. stores)
- SC在下述情况下会失败(**without putting transaction on bus**) :
 - 在尝试获取总线之前就检测到正在执行的写操作
 - 试图获得总线，但另一个处理器的SC先获得了总线
- LL本身不是加锁，SC本身也不是解锁，可以理解为把t&s拆分成了两段
 - 多个处理器同时执行LL，只有第一个将SC放到总线上的才可获得成功
 - 只保证LL-SC之间没有对同步变量的写冲突，不一定是原子的
 - 但是可以直接用来实现在共享数据结构上的某些原子操作

更加有效的软件加锁算法

■ 简单的LL-SC lock的问题

- 虽然在获得锁的尝试失败后不会产生作废，但当锁被释放时，等待的处理器很可能在该单元上扑空，并向总线产生读事务
- 可以在LL和SC之间使用回退减少突发性流量
- 不是一种公平的锁并且所引起的流量不是最小的

■ 更好的锁算法，更适合基于总线的机器(for r-m-w instructions or LL-SC)

- 我们希望当锁被释放时只有一个进程实际上试图去获取锁
 - 对我们使用 **test&set** 类型指令很有用; LL-SC 已经是这样了
- 我们还希望当一把锁释放时，只有一个进程发生读扑空
 - LL-SC同样也希望这样
- 票号锁（***Ticket lock***）达到了第一个目的
- 基于数组的锁（***Array-based queueing lock***）两个目的都达到了
- 而且两种锁都是公平的，基于FIFO序让处理器得到锁

票号锁 (Ticket Lock)

■ 就像在餐馆排队买餐或者银行的排队系统

- 需要获得锁的进程取一个票号 *next_ticket*，然后等待全局号 *now_serving* 和自己的相等

- 获取: `fetch&inc next_ticket`;

- 等待: *now_serving* 和自己的 *next_ticket* 是否相等

- 释放: `increment now-serving`

- 进程排队等待锁的到达, 减少了锁释放时的征用

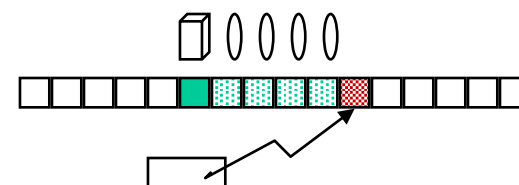
- 按照FIFO顺序, `fetch&inc` (原子操作) 如果是cacheable的, 在低竞争时延迟很低

- 票号锁释放时, 所有进程都在相同的变量 *now_serving* 上等待, 要引发 $O(p)$ 次读扑空 read miss

- 和LL-SC lock (票号锁可以使用LL-SC实现) 一样, 避免了多处理器在锁被释放后试图获取它时发出作废信号

- 锁释放时, 多处理器读扑空, 会产生突发流量, 如何使用回退方法避免

- 指数型回退是不合理的, 因为进程已经有了FIFO order
 - 按照 (*now-serving - next-ticket*) 的比例回退应该更好



■ 另一个自然的想法是让不同的进程在不同的单元上循环...

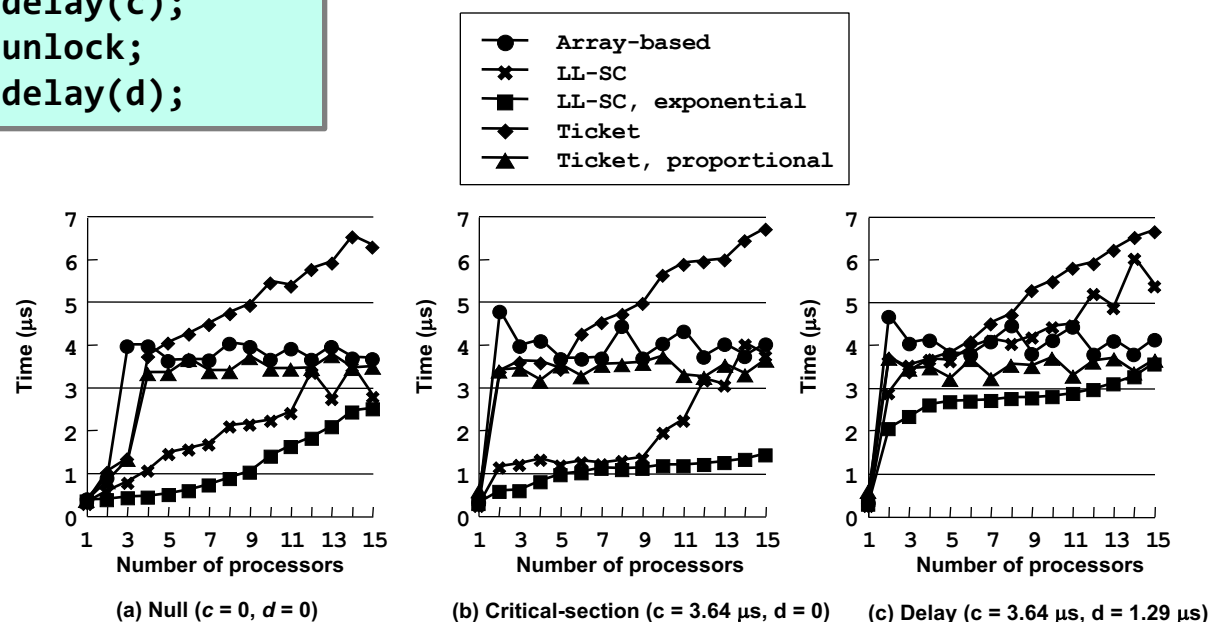
基于数组的锁

- 等待进程在大小为 p 的数组的不同位置上进行轮询
 - 获取
 - 使用fetch&inc 来获取一个唯一的单元而不是值(next array element), 然后在这个单元上忙等待
 - 理想情况是不同的单元分布在不同的存储块上, 以避免伪共享
 - 释放
 - 向下一个单元写入一个表示“解锁”的值, 这种释放使得在下一个单元上等待的处理器发生缓存块的作废, 它后面读扑空告诉它, 它已经得到了这把锁。
- 由于当释放时只有一个进程被通知, coherent caches总线上的流量就是 $O(1)$ 的
- 和票号锁一样, 也是FIFO公平的
- 但是这种所使用了 $O(p)$ 的空间
- 在基于总线的机器上性能很好
- 在没有cache一致性的分布式存储机器上会有问题
 - 某进程所使用的所并不在其本地内存里 (*Parallel Computer Architecture*)

我们来看一下SGI Challenge上不同锁的性能

```

Loop: lock;
      delay(c);
      unlock;
      delay(d);
  
```



- 简单的LL-SC 锁在处理器数量P比较小时性能最好，利用了不公平性
 - 释放锁的处理器能比其他处理器更快的重新获得锁，但是总线上的作废和读扑空会增加
- 使用按比例回退算法的票号锁和基于数组的锁扩展性较好
- 具体应用时还需要详细考察真正的应用负载特征

点对点的事件同步

■ 软件算法:

- 中断 Interrupts
- 忙等待 Busy-waiting: 使用通用变量做标记，在上面忙等待
- 阻塞 Blocking: 像操作系统中一样，使用信号量（semaphores）

■ 硬件支持: 让存储器的每个字都和“满-空位”的状态（*full-empty bit*）相联

- 该位置1表示满，即该字装有新数据（写操作上）
- 该位置0表示空，即该字被消费该数据的处理器“腾空”（读操作后）
- 和字级别的生产者-消费者同步天然匹配
 - 生产者: write if 0, set to 1;
 - 消费者: read if 1; set to 0;
- 硬件保证对满-空位读写操作的原子性
- 问题: 灵活性
 - 难以对付单生产者-多消费者同步，或者生产者在消费行为之前多次更新值的情况?
 - 需要语言和编译器中支持来区分何时使用该特性
 - 复杂数据结构如何支持?
 - 大多数商用机器中该技术都没有受到青睐

全局栅障事件同步

- 栅障软件算法的实现通常都用锁、共享计数器和标记单元
- 栅障的硬件实现
 - 使用独立于地址/数据总线的 线与 线（Wired-AND line）
 - 当到达barrier时，对应的线拉高，线与输出拉高标志所有进程都到达
 - 在实际使用中，多条连线允许重用
 - 当栅障是全局的、并且非常频繁的时候有用
 - 难以支持处理器的任意子集
 - 更难支持每个处理器有多个进程的情况
 - 难以动态更改参与者的数量和角色
 - e.g. 进程迁移会使参与处理器的角色发生变化
 - 在基于总线的机器上已经不常见了
- 我们下来主要考察基于简单硬件支持的软件栅障算法实现

集中式软件Barrier

■ 用一个共享的计数器来记录到达栅障的进程数

- 每一个到达的进程使它加1，这些加1是互斥的(lock)，检查计数是否等于P
- 会有什么问题？

```
struct bar_type {
    int counter;
    struct lock_type lock;
    int flag = 0;} bar_name;

BARRIER (bar_name, p) {
    LOCK(bar_name.lock);
    if (bar_name.counter == 0)
        bar_name.flag = 0;           /* reset flag if first to reach*/
    mycount = bar_name.counter++;     /* mycount is private */
    UNLOCK(bar_name.lock);
    if (mycount == p) {               /* last to arrive */
        bar_name.counter = 0;         /* reset for next barrier */
        bar_name.flag = 1;           /* release waiters */
    }
    else while (bar_name.flag == 0) {}; /* busy wait for release */
}
```


集中式软件Barrier

- 连续地进入同一**barrier**会出现问题
 - 防止一个进程在所有其他进程都离开先前的栅障之前再次进入这个栅障
 - 一个办法是用另一个计数器来记录离开栅障的过程，但是会引起进一步的时延和争用
- 带有感应（**sense reversal**）逆转的集中式Barrier: 在连续的栅障之间使用不同的释放值
 - 当所有进程都到达时转换释放值

```
BARRIER (bar_name, p) {  
    local_sense = !(local_sense);           /* toggle private sense variable */  
    LOCK(bar_name.lock);  
    mycount = bar_name.counter++;           /* mycount is private */  
    if (bar_name.counter == p)  
        UNLOCK(bar_name.lock);  
        bar_name.flag = local_sense;       /* release waiters*/  
    else{  
        UNLOCK(bar_name.lock);  
        while (bar_name.flag != local_sense) {};  
    }  
}
```

我们追求的集中式Barrier的性能

■ 低延迟 **Latency**

- 关键路径长度要小
- 集中式算法的关键路径长度和 p 成比例

■ 低流量 **Traffic**

- Barriers时全局操作，会产生高度竞争
- 集中式Barrier会产生 $3p$ bus transactions

■ 低存储代价 **Storage Cost**

- Very low: centralized counter and flag

■ 公平性 **Fairness**

- 避免同一个处理器总是最后一个离开barrier
- 在集中式算法中没有这个问题

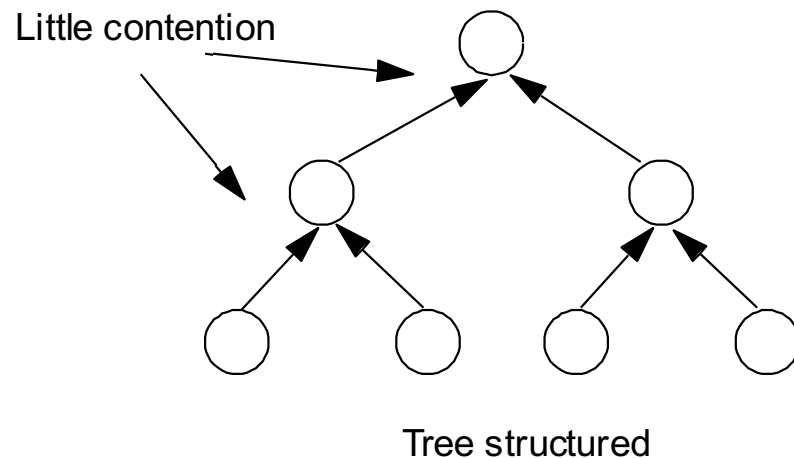
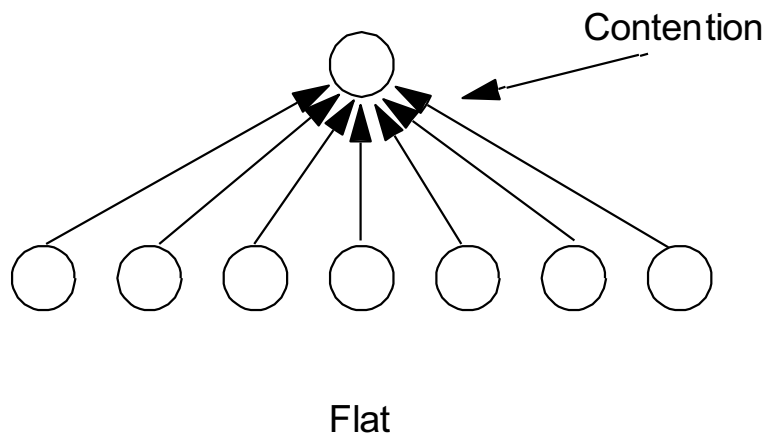
■ 集中式barrier的关键问题是latency和traffic

- 尤其是在分布式存储系统中, 流量往往会集中到同一个节点上

Barrier算法针对总线的改进

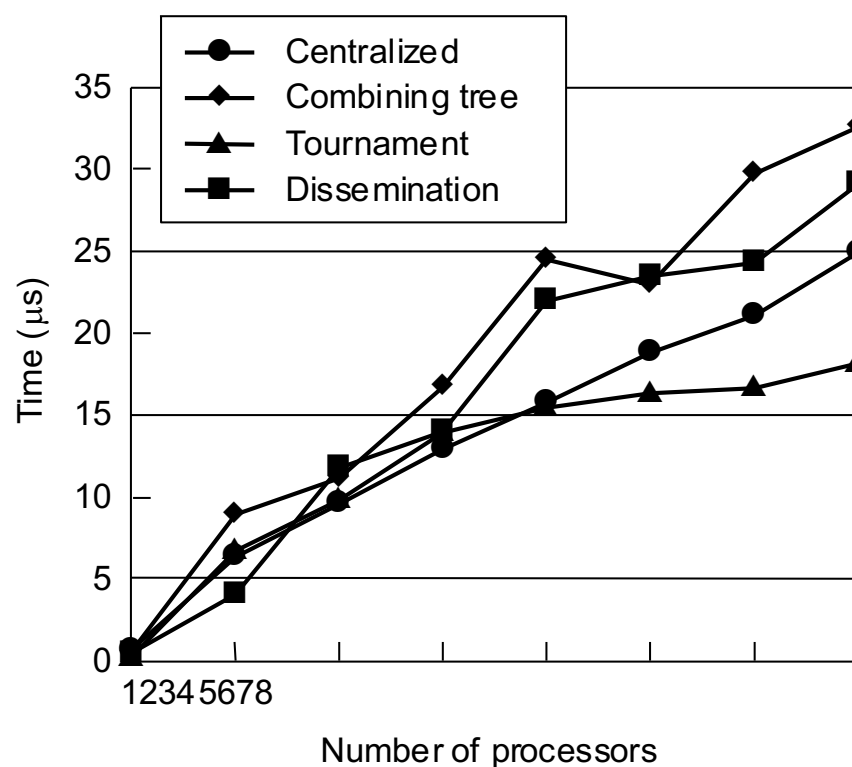
软件结合树 (Software combining tree)

- 只有 k 处理器访问同一变量, k 是结合树的度



- 不同分组的处理器争用不同的变量, 减少争用
- 在分布式网络架构中非常有用: 通信在不同的网络路径上进行
- 在总线架构中, 所有流量仍然在同一总线上, 并没有减少总体流量
- 而且会产生更高的延迟($\log p$ 个工作步骤, 每个步骤 $O(p)$ 个串行总线事务)
- 但是可以使用普通的 **reads/writes** 操作替代锁

SGI Challenge 上一些Barrier的性能



- 度量的是一个循环中连续执行许多Barrier，每个Barrier消耗的平均时间
- 集中式Barrier算法的性能很好
 - 分布式存储机器上如何设计更好的算法
- 结合树在总线架构上没有体现出优势，其他两种算法有特殊硬件支持

同步问题小结

- 该问题包含丰富的软硬件交互层面的权衡
- 需要将硬件原语和软件算法结合评估
 - 原语觉得哪个算法的性能更好
- 如何评估是一个挑战
 - 需要同时使用微基准测试程序和真实负载
- 基于通用硬件原语支持的软件算法在总线架构上表现不错
 - 如果我们考察分布式存储架构，情况会更加复杂
 - 硬件如何支持仍然是争论的主题

无锁算法实验

- C++11 提供了 `<thread>` and `<atomic>` 标准库
 - using `std::thread` 来生成threads
 - using `std::atomic` 来实现无锁算法
- 实验
 - 实现一个队列（queue）
 - 这个队列主要用于Producer-Consumer 问题
 - 有多个producer 和多个consumer
 - producers 产生随机数
 - consumers 计算所有他们获得的数的和
 - 比较基于锁的算法和无锁算法的性能

事务内存（TRANSACTIONAL MEMORY）

提高同步的抽象级别

- 机器硬件级别的同步原语：
 - fetch-and-op, test-and-set, compare-and-swap
- 我们使用上述原语构造了更高层次的，但仍非常基础的软件原语：
 - lock, unlock, barrier
- 我们进一步提高抽象层次：
 - 事务内存transactional memory

后续关于事务内存将介绍

- 事务（**transaction**）是什么
- 事务性内存有什么优势 **vs.** 锁
- 事务内存的设计空间
 - 数据版本控制策略 data versioning policy
 - 冲突检测策略 conflict detection policy
 - 检测的粒度 granularity of detection
- 理解事务内存的硬件实现(考虑一下它与我们上节课讲的一致性协议如何关联)

先举一个例子

```
void deposit(account, amount)
{
    lock(account);
    int t = bank.get(account);
    t = t + amount;
    bank.put(account, t);
    unlock(account);
}
```

- **Deposit**（存款函数）是一个**read-modify-write** 操作：我们希望存款这个操作相对于银行在这个账户上的其他操作是原子的
- **Lock/unlock pair** 是一种保证原子性的机制

如果我们将上述例子用事务内存实现

```
void deposit(account, amount){  
    lock(account);  
    int t = bank.get(account);  
    t = t + amount;  
    bank.put(account, t);  
    unlock(account);  
}
```



```
void deposit(account, amount){  
    atomic {  
        int t = bank.get(account);  
        t = t + amount;  
        bank.put(account, t);  
    }  
}
```

■ 陈述性的同步表达 **Declarative synchronization**

- 程序员通常喜欢表达是什么，而不喜欢表达如何做到什么
- 不需要显式的声明或者管理锁

■ 系统实现同步

- 通常使用乐观并发控制
- 只有当真正的冲突出现时才会减速 (R-W or W-W)

陈述式 Declarative vs. 命令式 imperative

- 陈述式: 程序员定义什么应该被实现
 - Process all these 1000 tasks
 - Perform this set of operations atomically
- 命令式: 程序员陈述具体应该如何做
 - Spawn N worker threads. Pull work from shared task queue
 - Acquire a lock, perform operations, release the lock

事务内存 Transactional Memory (TM)

■ 内存事务 Memory transaction

- 包含n个原子的相互独立的（atomic & isolated）内存访问的序列
- 受数据库中的事务概念启发

■ 原子性 Atomicity (all or nothing)

- 一旦提交（**commit**），序列中所有的内存写操作即刻生效
- 一旦取消（**abort**），序列中所有的内存写都不能生效

■ 隔离性 Isolation

- 在提交之前，其他代码看不到序列中写所产生的效果

■ 串行性 Serializability

- 事务内的所有操作按照某个序列提交
- 不过确切的序列并不能得到保证

事务内存的好处

我们再来看另一个例子: Java 1.4 HashMap

■ Map: Key → Value

```
public Object get(Object key) {
    int idx = hash(key);           // Compute hash
    HashEntry e = buckets[idx];    // to find bucket
    while (e != null) {            // Find element in bucket
        if (equals(key, e.key))
            return e.value;
        e = e.next;
    }
    return null;
}
```

- 不是线程安全的
- 但是在没有必要时也消除了锁带来的开销

同步的 HashMap

■ Java 1.4 solution: synchronized layer

- 对map函数进行线程安全改造
- 程序员可以使用显式的粗粒度的锁实现这个功能

```
public Object get(Object key) {  
    synchronized (mutex) { // mutex guards all accesses to hashMap  
        return myHashMap.get(key);  
    }  
}
```

■ 粗粒度同步的HashMap

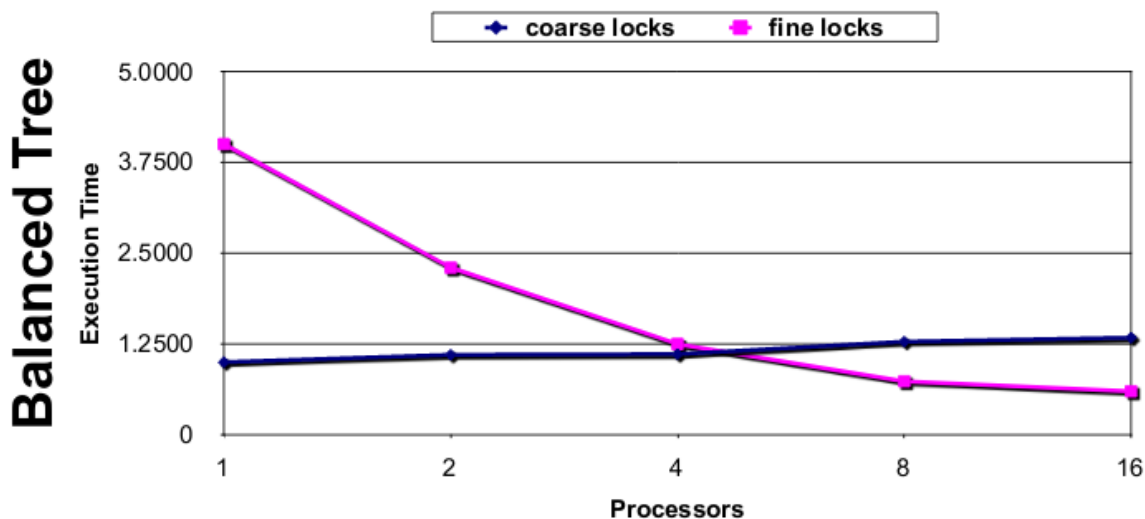
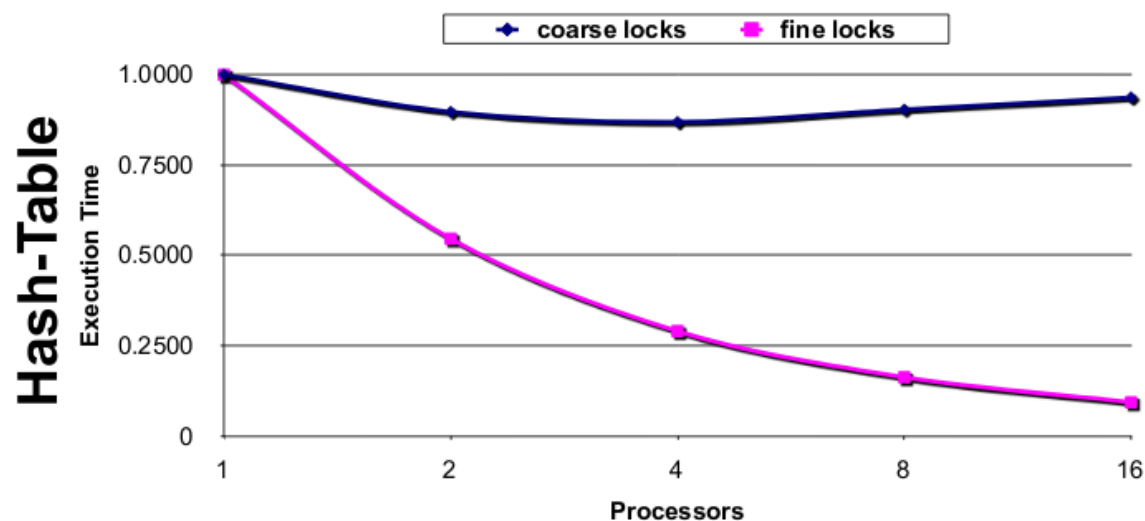
- 好处: 线程安全, 编程容易
- 缺点: 限制了并发性, 扩展性较差
 - 同一时刻只有一个线程可以在map上操作

是否有更好的解决方案呢?

```
public Object get(Object key) {  
    int idx = hash(key);           // Compute hash  
    HashEntry e = buckets[idx];    // to find bucket  
    while (e != null) {           // Find element in bucket  
        if (equals(key, e.key))  
            return e.value;  
        e = e.next;  
    }  
    return null;  
}
```

- 使用更细粒度的同步: e.g., 对每一个**bucket**加锁
- 虽然线程同步, 但是有大量的锁资源开销 (有些甚至不是必须的)

例如我们比较粗粒度同步和细粒度锁下的性能



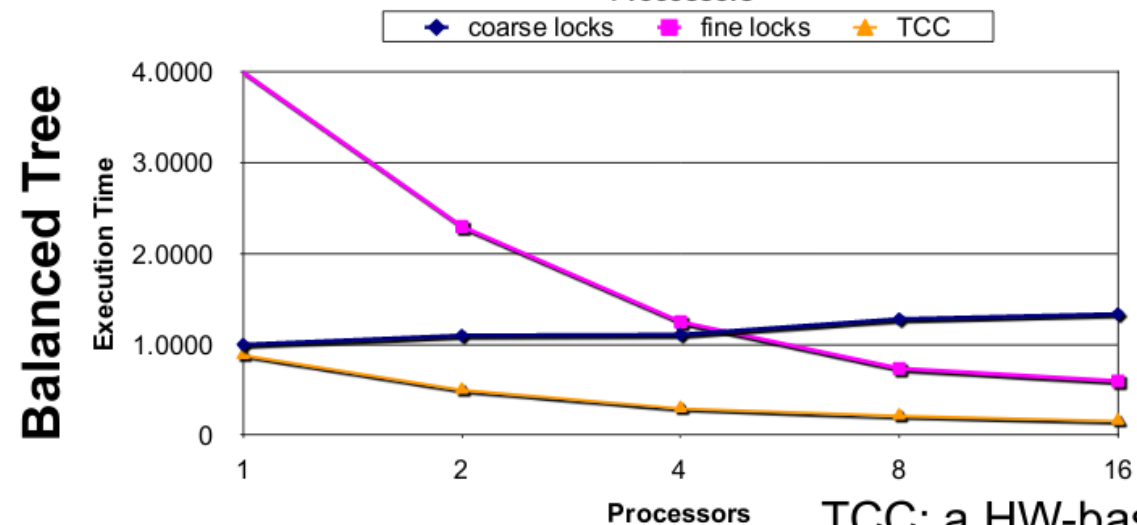
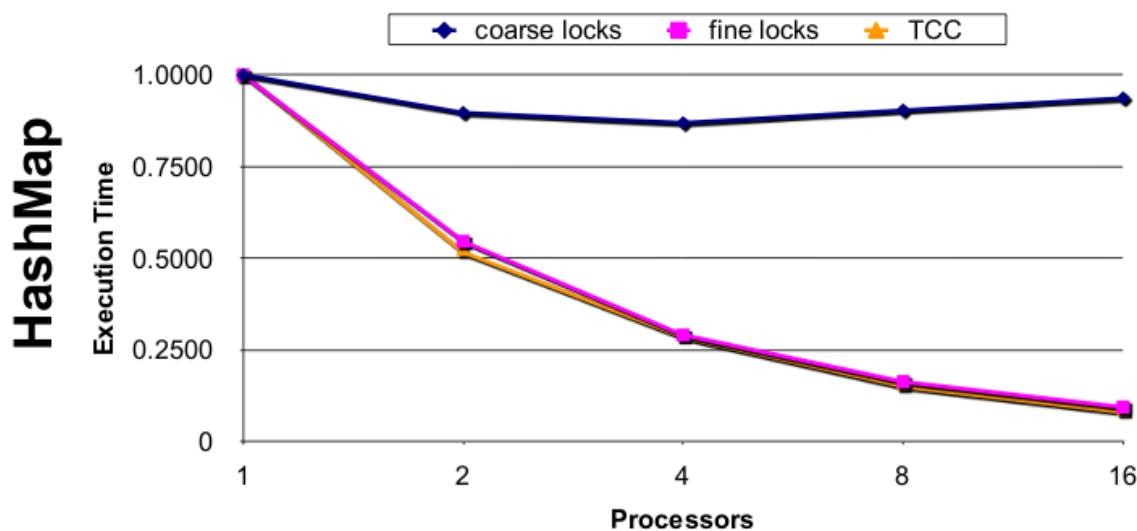
我们再来看事务性的HashMap

- 简单地将所有操作囊括在原子构造**atomic block**中
 - 由系统保证原子性的实现

```
public Object get(Object key) {  
    atomic {                // System guarantees atomicity  
        return m.get(key);  
    }  
}
```

- **Transactional HashMapn**
 - 优势: 线程安全, 编程简单
 - 问题: 是否能获得较好的性能和扩展性?
 - 取决于具体实现, 但一般情况下都能达到

性能比较: locks vs. transactions



TCC: a HW-based TM system

如果无法保证原子性: locks

```
void transfer(A, B, amount)
    synchronized(bank) {
        try {
            withdraw(A, amount);
            deposit(B, amount);
        }
        catch(exception1) { /* undo code 1*/ }
        catch(exception2) { /* undo code 2*/ }
        ...
    }
```

■ 程序员手动捕获异常

- 程序员case by case的提供撤销处理的方法
 - 复杂性在于: 都需要撤销什么? 如何撤销? ...

■ 其他线程可能会看到一些副作用

- E.g., 一个未捕获的异常可能会死锁系统...

如果无法保证原子性: transactions

```
void transfer(A, B, amount)
    atomic {
        withdraw(A, amount);
        deposit(B, amount);
    }
```

■ 系统处理异常

- 处理一些由程序员显式管理的，其他的都交由系统处理
- 取消事务，撤销事务产生的更新
- 其他线程不会看到部分更新
 - E.g., 没有失败的线程持有的锁...

可组合性（Composability）: locks

```
void transfer(A, B, amount)
synchronized(A) {
synchronized(B) {
    withdraw(A, amount);
    deposit(B, amount);
}
}

void transfer(B, A, amount)
synchronized(B) {
synchronized(A) {
    withdraw(B, amount);
    deposit(A, amount);
}
}
```

DEADLOCK!

- 编写基于锁的代码可能很棘手
 - 需要整个系统的策略以达到正确
 - 破坏了软件的模块化特性
 - 细粒度加锁: 性能很好, 但也很容易死锁

可组合性: transactions

```
void transfer(A, B, amount)
    atomic {
        withdraw(A, amount);
        deposit(B, amount);
    }
```

```
void transfer(B, A, amount)
    atomic {
        withdraw(B, amount);
        deposit(A, amount);
    }
```

■ 事务间的组合非常得当

- 程序员声明全局意图(原子化的转账)
 - 不需要指导全局的实现策略或细节
- 转账过程中的事务包括 取款 & 存款
 - 外层事务定义了原子性的边界

■ 系统尽可能的管理并发性

- transfer(A, B, \$100) & transfer(B, A, \$200) 相关账户转账串行化
- transfer(A, B, \$100) & transfer(C, D, \$200) 不相关账户转账并行化

总结一下事务性内存的优势

- 是一个易于使用的同步构造
 - 使用起来像粗粒度锁一样容易
 - 程序员只需要声明，系统负责具体实现
- 性能又和细粒度锁相当
 - 自动化的 read-read 并发 & 细粒度并发
- 原子化失败 & 恢复
 - 线程失败时不会丢失锁
 - 故障恢复= 取消事务 + 重启事务
- 可组合性
 - 安全可扩展的软件模块组合

TM还可与OpenMP组合使用

■ Example: OpenTM = OpenMP + TM

- OpenMP: master-slave parallel model
 - Easy to specify parallel loops & tasks
- TM: atomic & isolation execution
 - Easy to specify synchronization and speculation

■ OpenTM features

- Transactions, transactional loops & sections
- Data directives for TM (e.g., thread private data)
- Runtime system hints for TM

```
#pragma omp transfor schedule (static, chunk=50)
  for (int i=0; i<N; i++) {
    bin[A[i]] = bin[A[i]]+1;
  }
```

事务Atomic() \neq lock()+unlock()

■ 差别

- Atomic: 是一种原子性的高级声明
 - 不指定实现/阻塞的方式
 - 不参照特定的一致性模型
- Lock: 是一种较低级别的阻塞原语
 - 其本身并不提供隔离或原子性

■ 注意

- Locks 可以用来实现 atomic(), 但是...
- Locks 也可以用于原子性以外的目的
 - 因此我们不能把所有使用lock的地方替换成atomic
- Atomic 能够消除大量的数据竞争, 但是..
- 使用atomic blocks 编程仍然会遭遇原子性冲突. e.g., 原子序列错误地分裂成两个原子块

下面是一个使用lock但是无法使用atomic的例子

```
// Thread 1
synchronized(lock1) {
    ...
    flagB = true;
    while (flagA==0);
    ...
}
```

```
// Thread 2
synchronized(lock2) {
    ...
    flagA = true;
    while (flagB==0);
    ...
}
```

- 如果上述的synchronized换成atomic，会出现什么问题？
 - 两个线程都无法向下推进！

下面是一个atomic遇到原子性冲突的例子

```
// Thread 1
atomic() {
    ...
    ptr = A;
    ...
}

atomic() {
    B = ptr->field;
}
```

```
// Thread 2
atomic {
    ...
    ptr = NULL;
}
```

- 逻辑上原本应该串行化的原子事务序列，分裂成了两个冲突的原子事务块。

事务内存的实现方式

我们设计实现事务内存是为了：

- **TM = 陈述性的同步方式**
 - 用户只需要提出需求(atomicity & isolation)
 - 系统使用最佳方式实现
- **实现TM的动机**
 - 用户显式正确的实现各种同步是非常难的
 - Correctness vs. performance vs. complexity
 - 显式的同步扩展性也不好
 - Locking scheme for 4 CPUs is not the best for 64
 - 显式的同步难以和现成的软件组件进行组合
 - 需要制定一套全局的加锁策略
 - 需要更好的原子化失败处理，等等 ...
- **用户在支持事务的系统上可以获得细粒度锁90%的性能收益, 而只需要10%的开发时间**

考虑实现之前再回顾一下事务性内存实现的三要素

■ 原子性 **Atomicity (all or nothing)**

- 一旦提交 (**commit**), 序列中所有的内存写操作即刻生效
- 一旦取消 (**abort**), 序列中所有的内存写都不能生效

■ 隔离性 **Isolation**

- 在提交之前, 其他代码看不到序列中写所产生的效果

■ 串行性 **Serializability**

- 事务内的所有操作按照某个序列提交
- 不过确切的序列并不能得到保证

TM 实现的基础

- **TM 系统必须保证原子性和隔离性**
 - 同时不能牺牲并发性性能
- **基本的实现要求**
 - 数据版本管理 (用于支持事务取消)
 - 冲突检测 & 解决 (用于决定什么时候取消事务)
- **实现的选择**
 - Hardware transactional memory (HTM)
 - Software transactional memory (STM)
 - Hybrid transactional memory
 - e.g., Hardware accelerated STMs

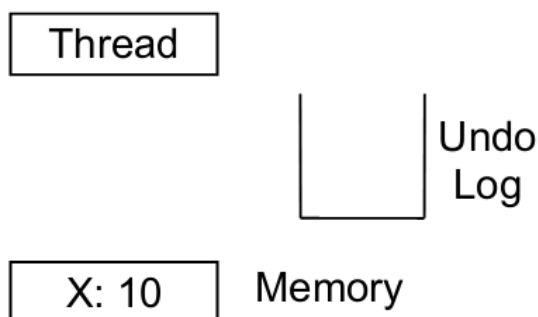
数据版本管理

- 在并发事务之间管理 **uncommitted (new)** 和 **committed (old)** 版本的数据
 - 急切的版本管理 Eager versioning (undo-log based)
 - 懒惰的版本管理 Lazy versioning (write-buffer based)

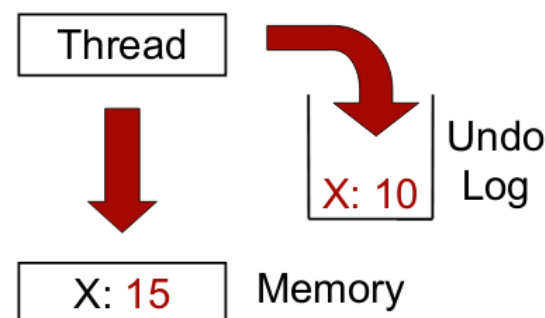
Eager versioning

- 立即更新内存, 但是维护一个 “undo log” 用作取消操作

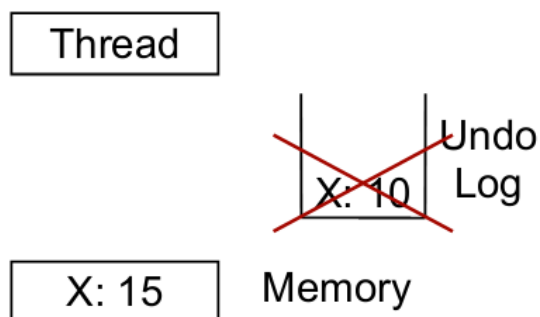
Begin Xaction



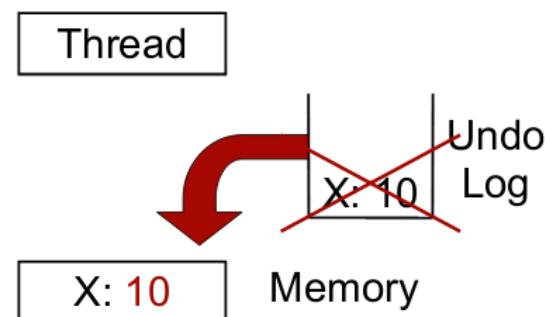
Write X ← 15



Commit Xaction



Abort Xaction

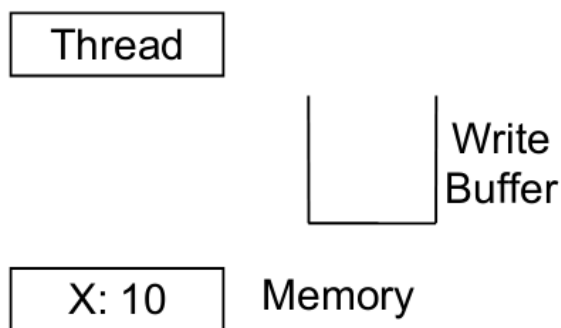


(CMU 15-418, Spring 2012)

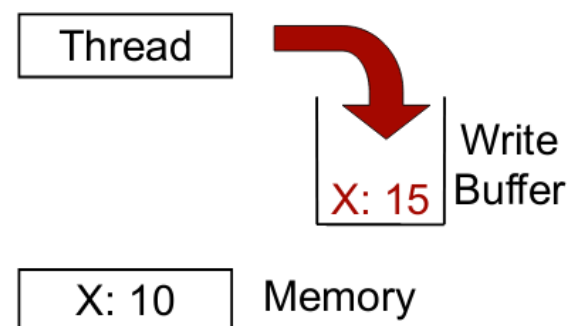
Lazy versioning

- 在transaction write buffer中记录所写的数据, 提交时将buffer中的数据写入内存

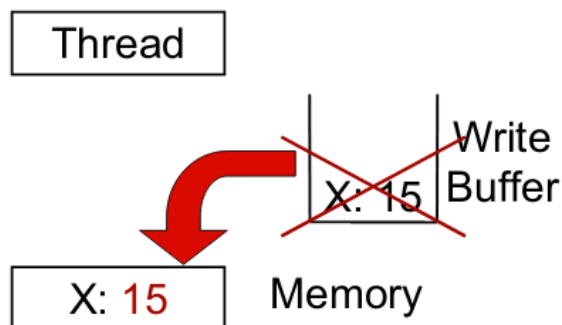
Begin Xaction



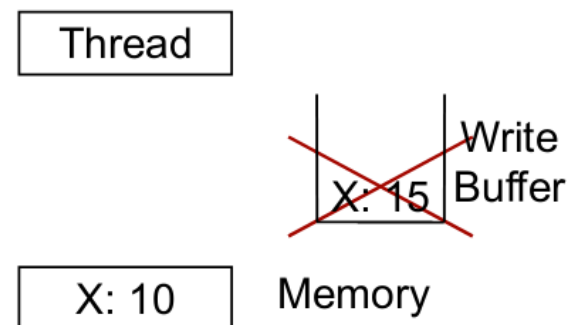
Write X ← 15



Commit Xaction



Abort Xaction



数据版本控制总结

- 在并发事务之间管理 **uncommitted (new)** 和 **committed (old)** 版本的数据
- **Eager versioning (undo-log based)**
 - 立即更新内存
 - 在log中维护需要撤销的信息 (per store penalty)
 - +Faster commit 快提交
 - Slower aborts, 慢撤销 会有容错问题 (crash in middle of trans)
- **Lazy versioning (write-buffer based)**
 - 直到提交前，数据都缓存在buffer中
 - 直到提交时才会更新内存中的数据
 - +Faster abort, 快撤销 no fault tolerance issues
 - Slower commits, 慢提交

冲突检测

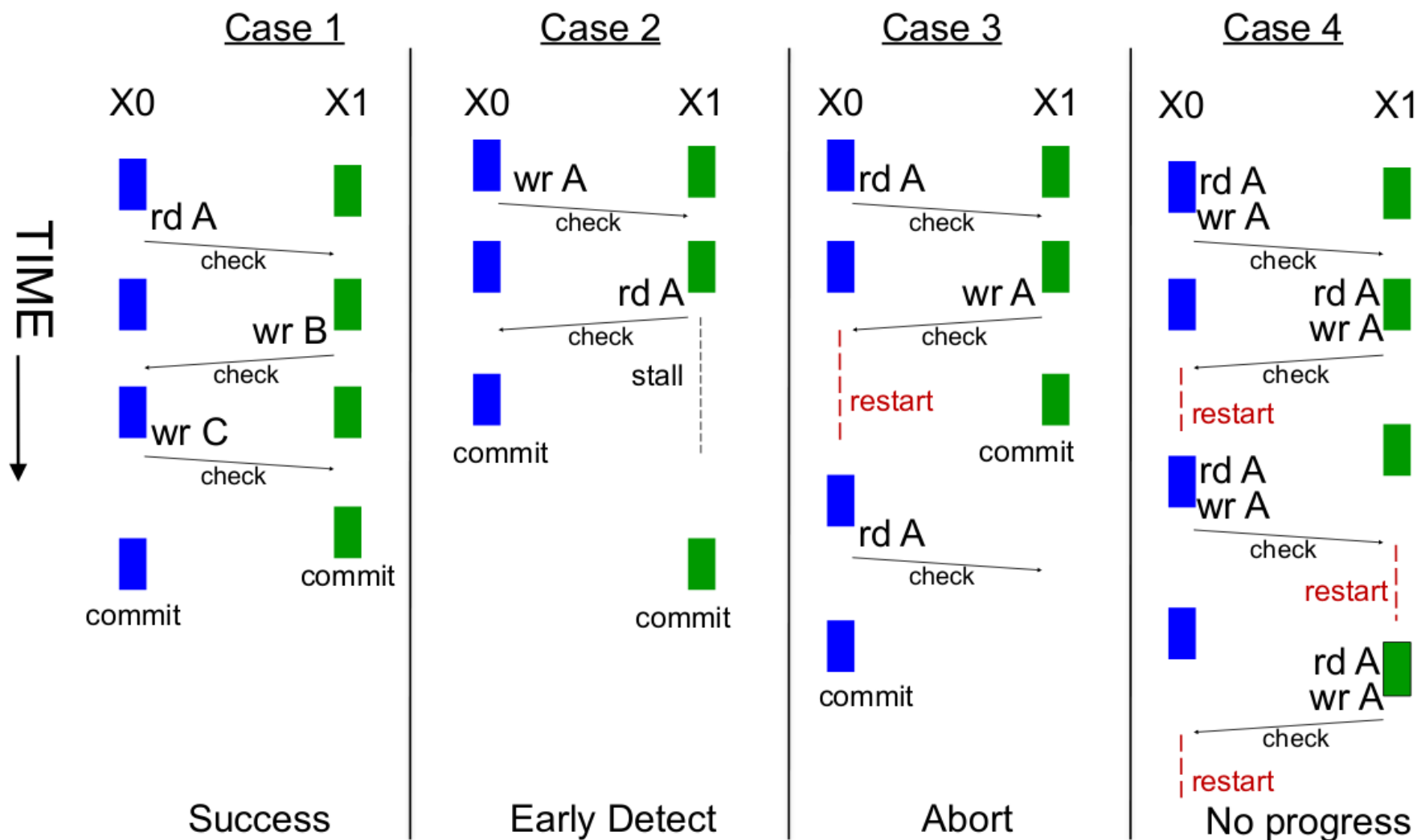
■ 检测并处理事务之间产生的访问冲突

- read-write conflict: 事务 A reads addr X, 但X正在被一个未完成的事务B写
- write-write conflict: 事务 A and B 都正在执行, 都在写地址 X.
- 需要跟踪 transaction的read-set 和write-set
 - Read-set: transaction中所有的地址读操作
 - Write-set: transaction中所有的地址写操作

悲观检测 Pessimistic detection

- 在loads or stores期间检测冲突
 - e.g.,某些硬件实现通过一致性操作进行检查（稍后讨论）
- “争用管理器 **Contention manager**” 决定停止或中止事务
 - 使用多种优先政策快速处理常见情况

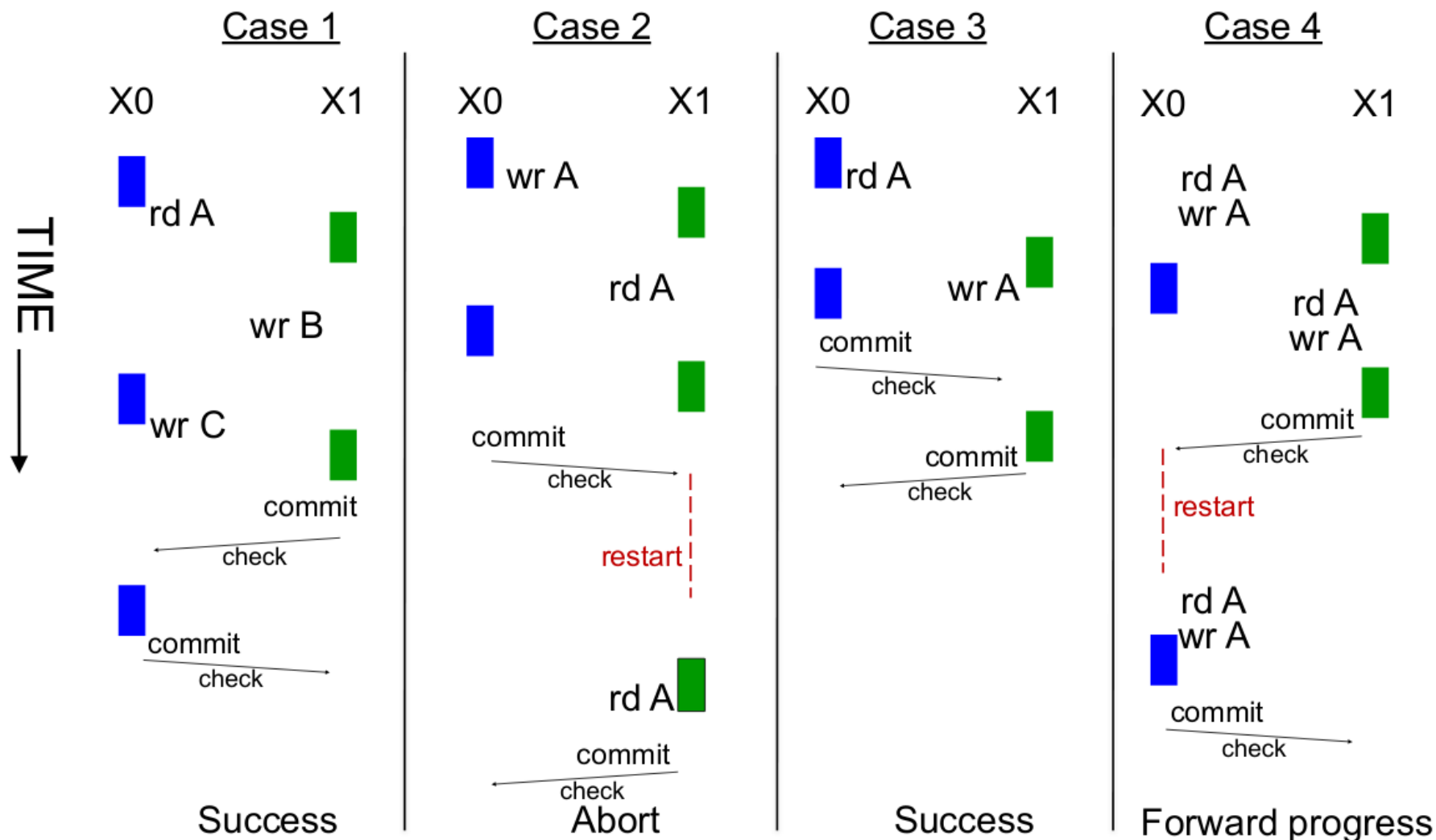
悲观检测的例子



乐观检测 Optimistic detection

- 在事务尝试提交时检测冲突
 - 硬件:使用一致性操作验证write-set
 - 获取write-set缓存行的独占访问权限
- 发生冲突时，优先提交事务
 - 其他事务稍后可能会中止
 - 在提交事务之间发生冲突时，使用争用管理器（contention manager）来决定优先级
- **Note: can use optimistic & pessimistic schemes together**
 - Several STM systems use optimistic for reads and pessimistic for writes

乐观检测Optimistic detection的例子



冲突检测的权衡

■ Pessimistic conflict detection (a.k.a. “encounter” or “eager”)

- +更早的发现冲突

 - 尽量少做无用功, 将abort转换为stall

- 但某些情况无法继续向前推进, 某些情况会产生更多的aborts

- 需要细粒度通信

- 在程序执行的关键路径上

■ Optimistic conflict detection (a.k.a. “commit” or “lazy”)

- +能够报证程序持续向前推进

- +可能减少冲突, bulk communication

- 检测冲突较晚有可能会产生公平性问题

冲突检测的粒度

■ Object granularity (SW-based techniques)

- +Reduced overhead (time/space)
- +Close to programmer's reasoning
- False sharing on large objects (e.g. arrays)

■ Word granularity

- +Minimize false sharing
- Increased overhead (time/space)

■ Cache line granularity

- +Compromise between object & word

■ Mix & match -> best of both words

- Word-level for arrays, object-level for other data, ...

TM implementation space (examples)

■ Hardware TM systems

- Lazy + optimistic: Stanford TCC
- Lazy + pessimistic: MIT LTM, Intel VTM
- Eager + pessimistic: Wisconsin LogTM
- Eager + optimistic: not practical

■ Software TM systems

- Lazy + optimistic (rd/wr): Sun TL2
- Lazy + optimistic (rd)/pessimistic (wr): MS OSTM
- Eager + optimistic (rd)/pessimistic (wr): Intel STM
- Eager + pessimistic (rd/wr): Intel STM

■ Optimal design remains an open question

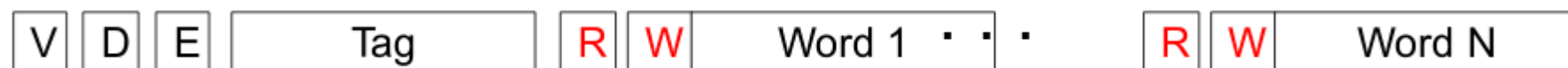
- May be different for HW, SW, and hybrid

Hardware transactional memory (HTM)

- 在缓存中做数据版本控制 **Data versioning in caches**
 - 缓存write-buffer 或 undo-log
 - 使用新的缓存元数据来跟踪 read-set 和 write-set
 - Can do with private, shared, and multi-level caches
- 通过**cache**一致性协议实现冲突检测
 - 一致性检查会检测到事务之间的冲突
 - Works with snooping & directory coherence
- **Notes**
 - 必须在事务开始之前先保留一个检查点

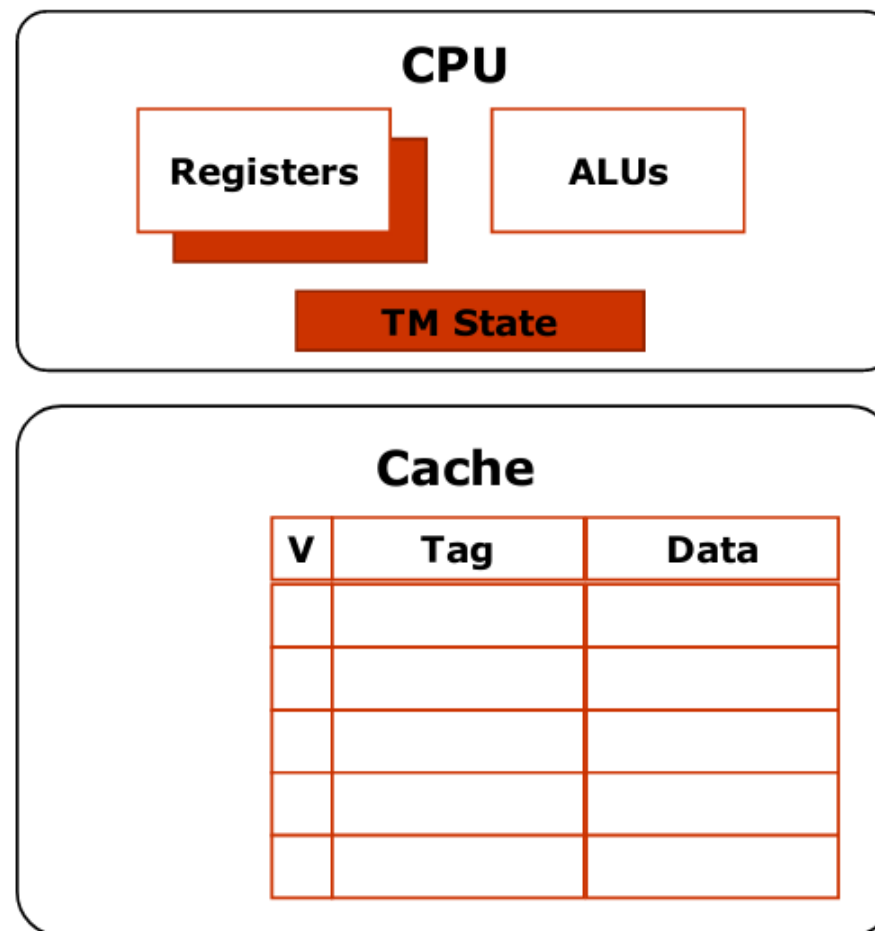
HTM design

- **Cache lines 增加注释元数据来追踪 read-set & write set**
 - R bit: indicates data read by transaction; set on loads
 - W bit: indicates data written by transaction; set on stores
 - R/W bits can be at word or cache-line granularity
 - R/W bits 当事务提交或者取消时清除
 - For eager versioning, need a 2nd cache write for undo log



- 收到一致性请求时，检查R/W位来发现冲突
 - Shared request to W-word is a read-write conflict
 - Exclusive request to R-word is a write-read conflict
 - Exclusive request to W-word is a write-write conflict

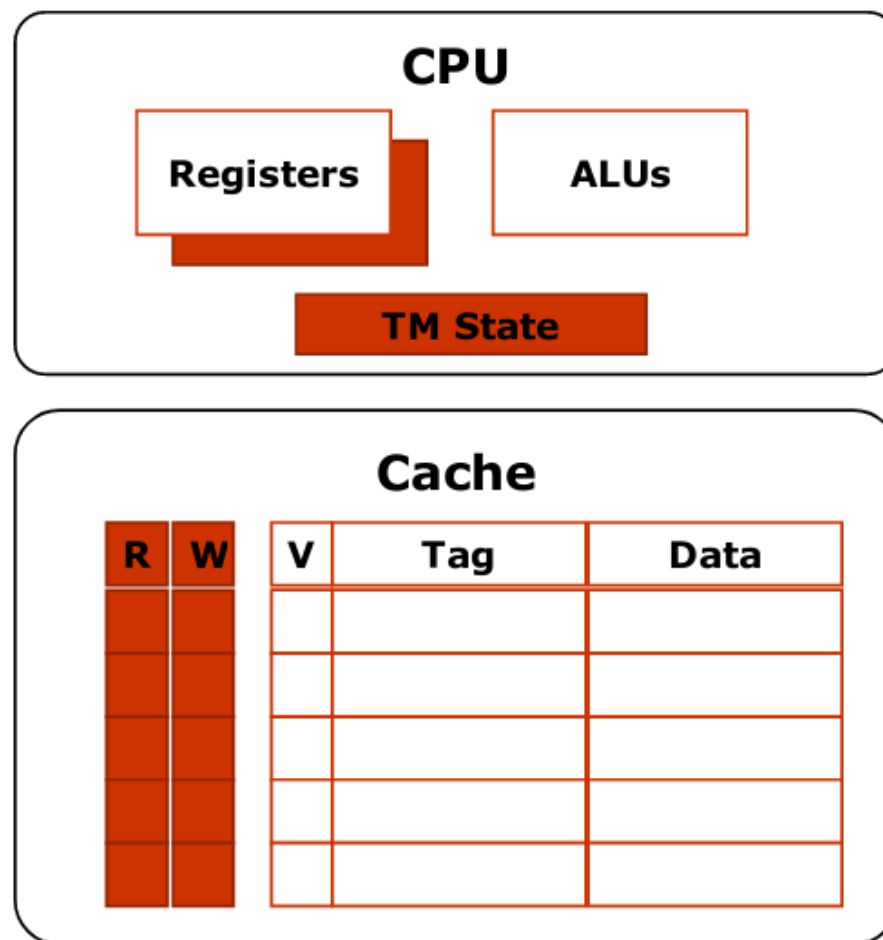
Example HTM: lazy optimistic



■ CPU changes

- Register checkpoint (available in many CPUs)
- TM state registers (status, pointers to handlers, ...)

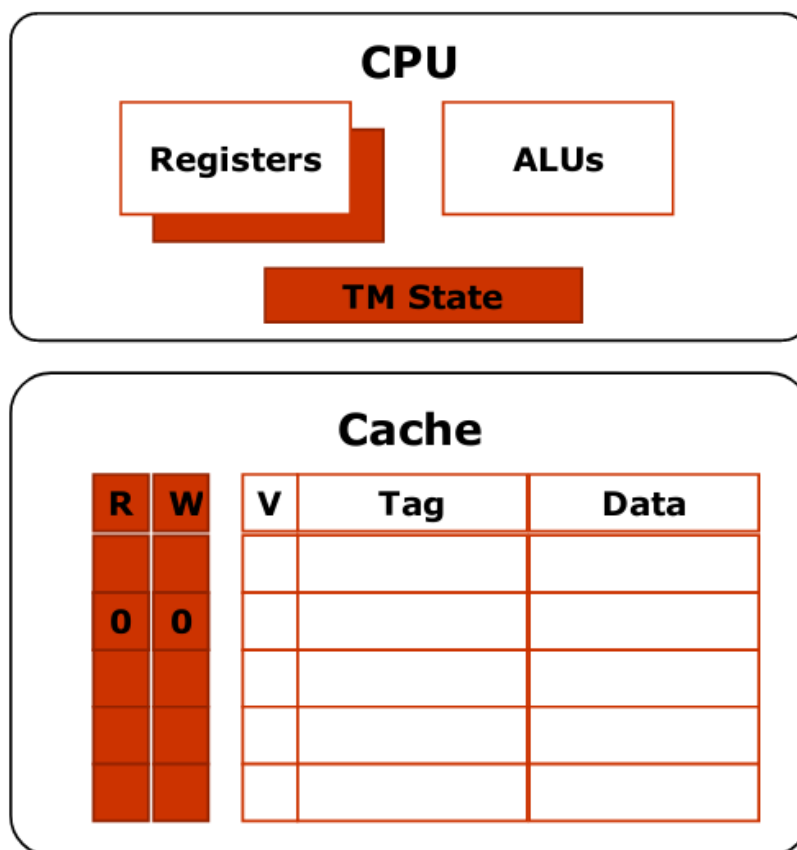
Example HTM: Lazy Optimistic



■ Cache changes

- R bit indicates membership to read-set
- W bit indicates membership to write-set

HTM transaction execution



Xbegin ←

Load A

Store B \Leftarrow 5

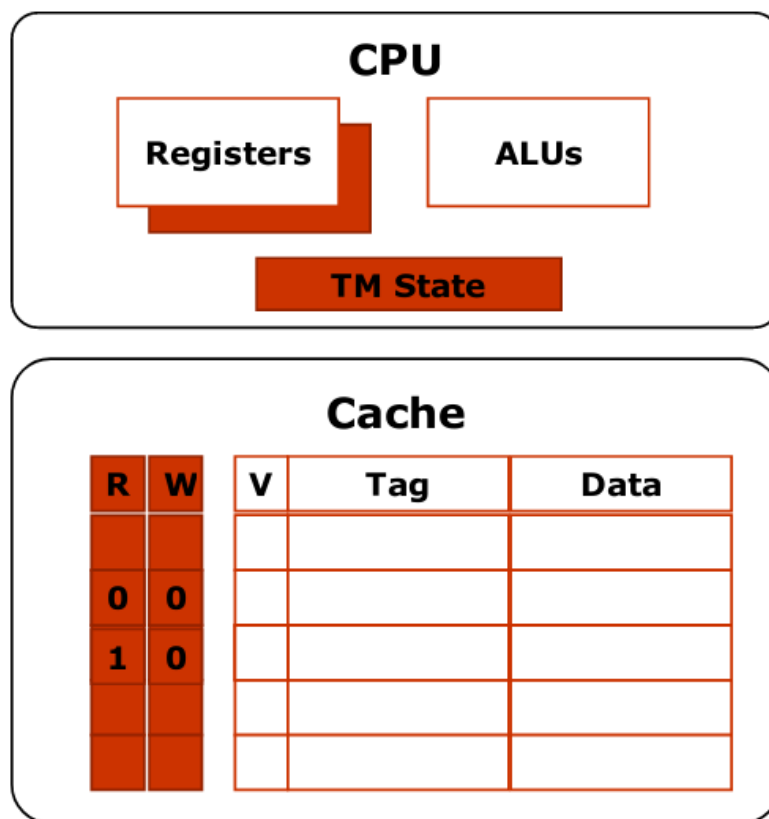
Load C

Xcommit

■ Transaction begin

- Initialize CPU & cache state
- Take register checkpoint

HTM transaction execution



Xbegin

Load A ←

Store B ⇐ 5

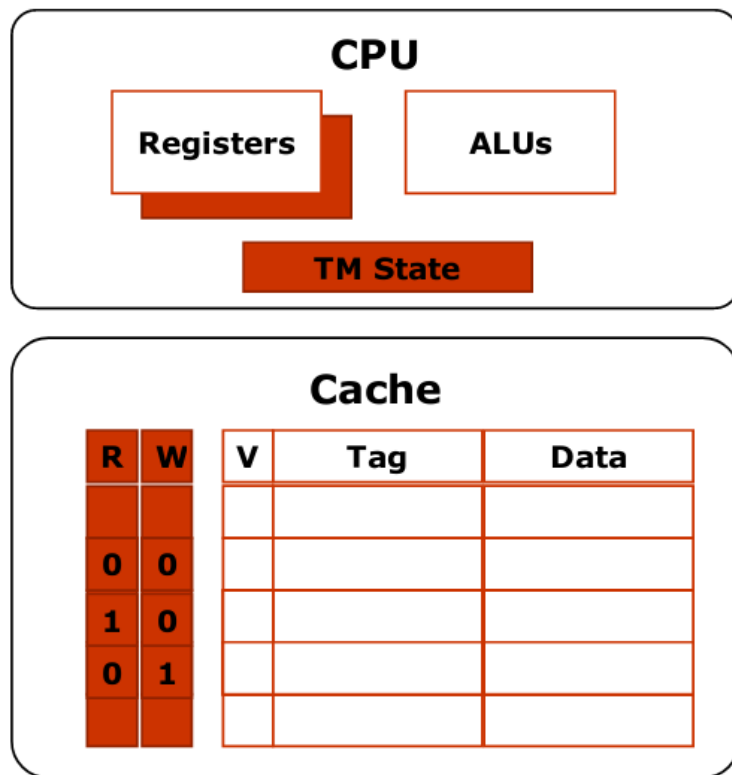
Load C

Xcommit

■ Load operation


- Serve cache miss if needed
- Mark data as part of read-set

HTM transaction execution



Xbegin

Load A

Store B \Leftarrow 5 

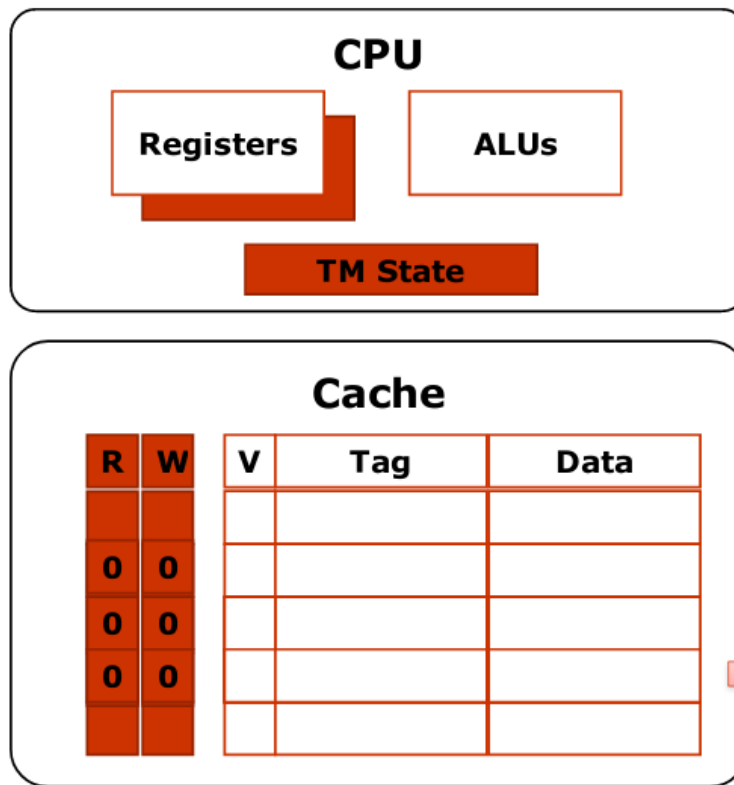
Load C

Xcommit

■ Store operation

- Serve cache miss if needed (**eXclusive** if not shared, **Shared** otherwise)
- Mark data as part of write-set

HTM transaction execution



Xbegin

Load A

Store B \leftarrow 5

Load C

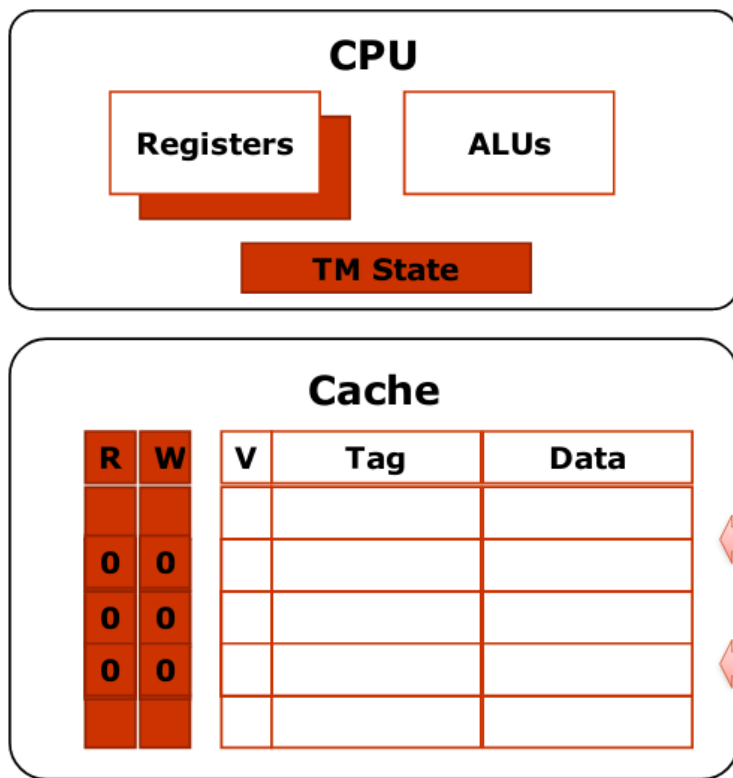
Xcommit 

 upgradeX B

■ Fast, 2-phase commit

- Validate: request exclusive access to write-set lines (if needed)
- Commit: gang-reset R & W bits, turns write-set data to valid (dirty) data

HTM conflict detection



Xbegin

Load A

Store B \Leftarrow 5

Load C

Xcommit

upgradeX D ☒

upgradeX A ☐

■ Fast conflict detection & abort

- Check: lookup exclusive requests in the read-set and write-set
- Abort: invalidate write-set, gang-reset R and W bits, restore checkpoint

Transactional memory summary

- 原子结构: 用于实现程序的某些原子行为
 - Motivating idea: increase simplicity of synchronization, without sacrificing performance
- 事务性内存如何实现
 - Many variants have been proposed: SW, HW, SW+HW
 - Differ in versioning policy (eager vs. lazy)
 - Conflict detection policy (pessimistic vs. optimistic)
 - Detection granularity
- 一种硬件实现事务性内存的架构
 - Versioned data kept in caches
 - Conflict detection built upon coherence protocol

Thanks Q&A