基于硬件 CAS 原语的高效多字无锁同步算法

吴 昊,季振洲,朱素霞
(哈尔滨工业大学计算机科学与技术学院,黑龙江哈尔滨 150001)

摘 要: 共享内存体系结构下,为解决锁同步导致的并发性能瓶颈,本文提出了一种基于硬件 CAS(比较交换)原语的无锁同步算法.该算法利用底层处理器提供的比较交换指令,实现了在多核多线程环境下对共享变量的非阻塞同步操作,通过采用全局标记值的方式,避免了传统设计中由于使用内存字标记导致的性能开销,同时确保数据在并发访问中的一致性.实验结果表明,本文算法可以高效地支持任意多字的 CAS 同步,提高了对共享数据的并发访问性能,具有较好的可扩展性.

关键词: 无锁同步; 多线程; 并发算法
中图分类号: TP314 **文献标识码:** A **文章编号:** 0372-2112 (2013) 11-2127-08
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2013.11.003

Efficient Multi-Word Lock-Free Synchronization Algorithm Based on Hardware CAS Primitive

WU Hao, JI Zhen-zhou, ZHU Su-xia
(*School of Computer Science and Technology, Harbin Institute of Technology, Harbin, Heilongjiang* 150001, *China*)

Abstract: Lock-based synchronization may become a performance bottleneck which limits the concurrency in shared-memory machines. In order to solve this problem, a lock-free synchronization algorithm based on hardware CAS(Compare And Swap) primitive is proposed in this paper. In the proposed method, compare and swap instruction provided by underlying processor is used to implement non-blocking synchronization for shared variables in multi-core or multi-thread environment. Global mark value is introduced to avoid performance overhead caused by bits reservation of memory word in the traditional design, and guarantee the consistency. Theoretical analysis and experimental results show that the proposed method can efficiently support arbitary multi-word CAS synchronization, improve the concurrent access performance and provide good scalability.

Key words: lock-free synchronization; multithread; concurrent algorithm

1 引言

随着半导体工艺的进步,采用共享存储结构的多核处理器逐渐取代单核处理器,从硬件上提供了更强大的并行处理能力.同时越来越多的算法如事务内存,并发垃圾回收等[1,2]对共享数据的并发访问提出了更高的要求.传统的并行编程模型中,使用锁同步保护共享资源,将一部分并发的操作串行化以保证并行程序的正确执行.然而由 Amdahl 定律可知,在并行系统中加速比的极限是串行代码所占比例的倒数,因此锁同步严重限制了程序的并行性[3].此外,使用锁同步还要考虑可能会导致死锁、线程饥饿以及优先级倒置等问题.

新的同步模型一直是学术界研究的热点问题[4],在现有的硬件平台下,软件事务内存(STM[5]: Software Transaction Memory)可以作为替代锁同步的一种技术.其设计思想是基于乐观的并发控制策略,当一个线程执行对共享内存的修改时忽略其它线程,在完成一个事务后再根据日志决定是否提交或回滚,其设计的初衷是采用这种乐观策略以增加系统的并发性能.然而 Dragojevic[6]等人指出由于 STM 的同步方式和冲突检测机制导致运行时的开销过大,限制了系统加速比.

近年来,对无锁算法无锁数据结构的研究得到了越来越多的关注[7],研究人员希望设计一种无锁的同步方式从根本上克服锁同步的诸多缺陷,从而构建更加高效和可扩展的系统.无锁同步需要借助于底层系统提供的原子操作[8],比如比较交换原语(CAS: Compare And Swap),

收稿日期:2012-10-26;修回日期:2013-04-08;责任编辑:马兰英
基金项目:国家自然科学基金(No.61173024)

它可以以原子方式实现对一个内存字的条件更新(Conditional Update).然而,由于受硬件的限制,现有的处理器仅能实现单字或双字的 CAS 原子操作.多字 CAS(MCAS:Multi – word CAS)同步是对单字 CAS 的一种扩展,可以克服单字 CAS 的不足.然而由于现有的计算机系统还不能从硬件上支持 MCAS,这就为 MCAS 的设计造成了困难,已有的设计往往需要过多的限制条件,如在内存字中预留标记位、仅支持连续内存字、或者依赖特定处理器架构下特殊指令等.

针对多核多线程环境下对共享数据的高并发访问要求和已有研究的不足,本文提出了一种新的无锁方式支持任意多字的 CAS 同步的方法 LMCAS(Lock – free Multi – word Compare And Swap),该设计以全局标记值的方式解决了传统 MCAS 设计中内存字标记占位导致的开销过大的问题,支持非连续的多字 CAS 同步,满足数据在并发访问中的一致性要求.理论分析和实验结果表明,算法在多核多线程环境下能够有效地实现任意多字的无锁同步操作,相比锁同步在并发访问性能和可伸缩性上具有较大优势.

2 相关工作

由于无法从硬件直接执行 MCAS,已有的研究工作[9~12]是以软件的方式在现有的硬件条件下实现该功能.Israeli 和 Rappaport 等人设计了第一个基于 LL/SC(Load-link/Store-Conditional)原语的 MCAS 算法[13].当线程数为 N 时,他们的算法需要在每个待更新的内存字(Memory Word)中预留出 N 个比特位(N bits reserved per word)作为标记,然后依次在每个待更新的位置上执行 LL/SC 操作,并根据标记的状态保证 MCAS 同步,算法支持不相交的访问并行(Disjoint access parallel).Anderson 和 Moir[14]等人改进了 Israeli 算法的存储开销,将每个待更新内存字的预留占位降到 logN,可以支持更多的线程,但他们的改进却牺牲了一定的访问并行性.

Harris[15]等人为了将 MCAS 操作中的预留占位限制为常量,同时又要保留访问并行的特性,他们引入描述符(Descriptor)结构.在描述符结构中存储了每个待更新字的原值、更新值以及用于指示 MCAS 的状态标记,此外还包括一个指向待更新字的指针.他们的设计将 MCAS 中每个字的占位降到 2 比特,然而 Harris 的 MCAS 设计过于复杂,在实际应用中不如使用锁同步方便而且极易出错.在此基础上,Fraser[16]封装了 Harris 的 MCAS,设计了基于字(word-based)和基于对象(object-base)两种粒度的软件事务内存,以简化的接口提供给程序开发人员,但仍未解决预留标记占位的问题.

此外,Sundell[17]也对 Harris 的 MCAS 算法进行了改进,在处理多线程竞争方面,采用抑制并发线程的策略.在发生竞争时选出可能会导致 MCAS 操作失败次数最多的线程,迫使被选出的线程放弃对当前 MCAS 的操作,直到消除所有竞争为止.其改进仅关注了执行的效率却忽略了并发性能,而且在处理冲突时没有考虑多线程执行 MCAS 的先后顺序,牺牲了一定的公平性.

表 1 总结了已有的 MCAS 设计并与本文的方案进行对比,其中 N 为并发执行的线程数.可以看出,已有的相关研究都需要在每个内存字预留标记位,本文 LMCAS 的优势在于不仅支持并发操作特性,而且不需要内存字预留标记位.

表 1 不同版本的 MCAS 算法比较

相关研究	并发操作	标记位/字	硬件原语
Israeli[13]	支持	N	LL/SC
Anderson[14]	不支持	LogN	LL/SC
Harris[15]	支持	2	CAS
Fraser[16]	支持	2	CAS
Sundell[17]	不支持	2	CAS
LMCAS	支持	0	CAS/LL/SC

3 CAS 硬件同步原语

在多核多线程系统中,无锁(Lock-Free)同步的基础是系统支持原子的 RMW(Read-Modify-Write)操作序列,许多现代多核架构都从指令上支持某种形式的 RMW 指令,如 CAS 或 LL/SC.CAS 最早在 IBM System/370[18]系统中实现,该操作可以定义成程序 1 的伪码形式,其操作包含三个操作数:内存地址 a,预期值 e,以及拟向该地址写入的新值 n.当且仅当从地址 a 读到的原值与预期的值相等时,CAS 才会以原子的方式用新值 n 更新 a.如程序 1 所示.

程序 1 CAS 原子操作

```
word_t CAS(word_t * a, word_t e, word_t n)
{
    word_t old;
    atomic{
     old = * a;
     if (old == e) * a = n;
    }
    return old
}
```

程序 2 是本文在 32 位 x86 平台 Linux 环境下使用 Intel 的 cmpxchg 指令[19]实现的 CAS 同步操作,它是实现本文无锁同步的基础.cmpxchg 指令将累加寄存器中的值与目的操作数进行比较,如果相等则将源操作数的

值装载到目的操作数,同时将标志寄存器的 *ZF* 置 1.如果不等,目的操作数的值装载到累加寄存器中,同时 ZF 清 0.

基于 cmpxchg 指令实现 CAS 操作

程序 2

```
char CAS(word_t * address, word_t oldValue, word_t newValue)
{
  char result;
  __asm__ __volatile__(
    "lock; cmpxchg %3, %2 \n"
    "sete %b \n"
    :"=r"(result), "+a"(oldValue), "+m"(* address)
    :"r"(newValue)
    :"memory", "cc");
  return result;
}
```

对于 64 或 128 位的 CAS 操作可以使用 cmpxchg8b 或 cmpxchg16b 指令来实现,在不支持 cmpxchg 指令的平台上,如 MIPS,可以使用 LL/SC 指令实现对共享资源的保护.使用 LL/SC 原语也可以实现 LMCAS 算法,因此本文的算法并不局限在 Intel 的 x86 架构,在不同架构下的 LMCAS 的差别只在底层实现所使用的硬件原语.表 2 给出了本文基于 cmpxchg 和 cmpxchg8b 实现 32 和 64 位 CAS 操作的执行性能情况,测试的平台分别为 Intel 的 Pentinum4, Core2, Xeon.可以看出成功执行的 CAS 操作的时间约为失败时的 1.1 – 1.3 倍,而 64 位 CAS 执行时间为 32 位的 1.3 – 1.5 倍.

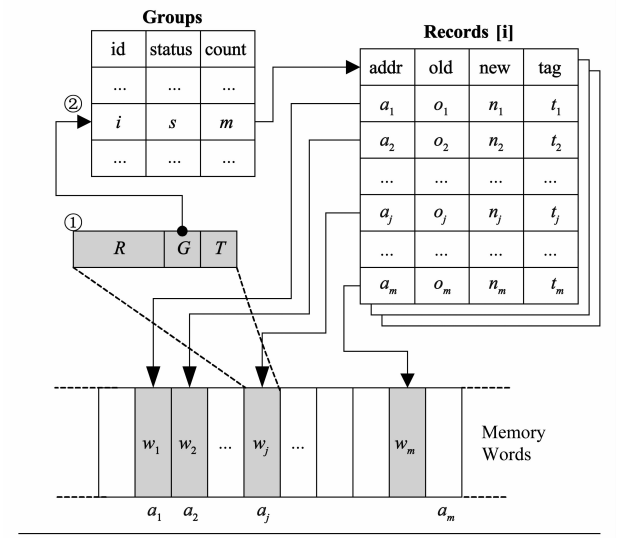
表 2　本文的 CAS 在不同处理器下的性能表现(百万次执行时间,单位 10^{-6}s)

CPU	cmpxchg		cmpxchg8b	
	success	failure	success	failure
Pentium4/2.4GHz	121040	94808	175824	189085
Core2/1.5GHz	87845	74890	123729	107582
Xeon/2.4GHz	106905	107030	207280	197020

4　LMCAS 设计

4.1　算法实现

为了支持多线程并发执行并保证更新过程中的一致性,本文引入了记录(Records)和组(Groups)这两个结构,算法执行过程中根据记录和组的状态实现正确的更新操作.如图 1 所示,每个被更新的内存字 $w_1, w_2, \cdots, w_m$ 都关联一个记录,每个记录由内存字的地址(addr),原值(old),更新值(new)和标记(tag)组成.同一组的所有记录关联一个组结构用于指示该组的记录和更新状态,组结构包括组标识(id),更新状态(status)和该组的内存字数量(count).在预处理阶段,内存字的更新信息以记录的形式提交给系统,所有的记录组成一个全局记录表,使用组标识作为该表第一维的索引,如图 1 中所示,可以通过 Records[*i*]访问组标识为 *i* 的所有 *m* 个记录.



注①: *R*: Rvalue field; *G*: Group field; *T*: Thread field.
②: Group _Field (w_j) = *i*

图1　CAS组结构示意图

当建立了全局的记录表和组结构之后,接下来是标记和更新的过程.标记的目的是为了将控制信息写入到内存字,从而保证接下来的更新过程是原子操作,即操作完成后 $w_1, w_2, \cdots, w_m$ 全部更新到新值或者都保持原值,不出现部分更新的情况.如图 1 中所示,对一个字 *w* 标记的控制信息由低位到高位划分为三个功能域:线程标识域(Thread field),组标识域(Group field)和标记值域(Rvalue field),表 3 定义了对各功能域的相关操作.线程标识域用于指示标记该字的线程 ID,组标识域指示该字所属的组,标记值域用于区分该字是否被标记.当从一个内存字读出标记值域与全局标记值不等时,可判断该字未被标记.而当标记值域与全局标记值相等时,还需通过组标识查询记录 Record[*i*]中是否包含此内存字地址,具体的标记判断过程见算法 1.

分别用 *T*、*G* 和 *R* 表示各功能域宽度,对于同一个 LMCAS 操作,算法支持的最大线程并发数为 2^T,由于设置了组 ID 域,在系统中支持并发执行的 LMCAS 操作组数为 2^G,非标记字与全局标记值相等的概率为 2^{-R}.标记过程支持多线程并发执行,通过 CAS 保证有且只有

一个线程标记成功,避免了因重复标记造成的冲突.在对内存字标记成功后,算法将与该字关联的记录中的tag更新为执行标记的线程ID.对于整个组的标记过程使用组结构中的status指示标记完成的状态.内存字标记判断算法1

表3 对各功能域的相关操作

函数名称	功能描述
Thread_Field(w)	读取字 w 的线程标识域
Group_Field(w)	读取字 w 的组标识域
Rvalue_Field(w)	读取字 w 的标记值域
CreateRWord(t,g,r)	由输入的参数构造一个线程标识为 t,组标识为 g,标记值为 r 的内存字

```
算法1 IsCASWord
Input:address/*内存字地址*/
Output:result/*判断结果:已标记true,未标记false*/
Begin
  word_t w:=*address;
  if RValue_Field(w)=Global_R then//使用全局标记值判断
    word_t group_id:=Group_Field(w);
    for i:=0 to Group_Count[group_id]-1 do
      if MCAS_Records[group_id][i].addr=address then
        return true;          //已标记
  return false;
End
```

标记完成后,执行对同组内已标记的内存字进行并发更新.为保证一致性,更新时需要判断已标记内存字的线程标识域与该字所关联的记录的tag是否一致,防止该字被意外修改,但对于执行更新操作的线程并不要求与tag一致,即对一个内存字的标记和更新可以由两个不同的线程来执行,尽可能满足并发执行的需求.当一个组内的全部内存字更新后,算法结束并更新该组的完成状态.本文LMCAS算法实现过程,见算法2.

```
算法2 LMCAS
Input:group_id/*组ID*/,thread_id/*当前线程ID*/
Output:status/*完成状态:成功true,失败false*/
Begin
C1: word_t R:=GreateRWord(Clobal_R,thread_id,group_id)
  //第一阶段:Mark
C2:foreach erc in MCAS_Records[group_id]do
C3:   if CAS(rec.addr,rec.old,R)then
C4:     mark_id:=thread_id;//rec被当前线程标记
C5:   else  //当前线程对rec的标记失败,需处理以下三种情况
C6:     word_t curr:=*rec.addr;
C7:     if RValue_Field(cuee)=Global_R&&
C8:       Group_Field(curr)=group_id then
C9:       mark_id=Thread_Field(curr);//[1]rec已被其他线程标记
C10:    elseif CAS(&Group_Status[group_id],
C11:      MARK_INPROGRESS.MARK_FAILURE)then
C12:      return false;//[2]rec没有被标记过,不满足比较条件
C13:    else
C14:      break;//[3]rec所在组的标记已完成
C15:  CAS(&rec.tag,null,mark_id);
C16:CAS(Group_Status[group_id],MARK_INPROGRESS,
      MARK_OK);  //第二阶段:Update
C17:foreach rec in MCAS_Records[group_id]do
C18:  word_t curr:=*rec.addr;
C19:  if Thread_Field(curr)=rec.tag then
C20:    CAS(rec.addr,curr,rec.new);
C21:    RemoveFromGroup(rec);
C22:CAS(&Group_Status[group_id]),MARK_OK;UPDATE_OK
C23:RemoveGroup(group_id);//该组更新完成
C24:return true;
End
```

4.2 多线程并发执行

本文的LMCAS算法支持多线程并发执行,提高了算法的性能,当系统中某个线程正在执行LMCAS时,其他的线程对内存字的访问都需要通过CAS_READ接口进行,如算法3所示.CAS_READ首先检查访问的目标地址是否被标记,如果已标记则参与到该组的LMCAS执行过程,执行完毕后再将更新过的字返回给调用者.此机制保证了算法执行过程中内存字不被意外修改,理想情况下参与的线程越多,LMCAS就会在越短的时间内完成,在此过程中没有发生因竞争共享资源导致的线程阻塞,这是本文的无锁同步与锁同步的最大区别.CAS_READ实现过程见算法3.

```
算法3 CAS_READ

Input:address/*内存字地址*/
```

```
Output: w/* 读取的内存字值 */
Begin
  while IsCASWord(address)do
    LMCAS(Group_Field(word_t) * address), ThreadId
    ());
  return w;
End
```

当多个线程竞争同一内存地址执行 CAS 操作时,有且只有一个线程能够成功,其他线程在比较条件不满足时,可以根据状态决定是否重试或采取其他操作,LMCAS 的两阶段操作过程中保证了组记录中的字不会被重复标记或更新.以两个线程 A 和 B 竞争同一地址的内存字 w 为例,说明 LMCAS 是如何处理线程竞争的情况:用 M_A, M_B, U_A, U_B 分别表示线程 A 和 B 的 Mark 和 Update 操作,那么 M_A, M_B 与 U_A, U_B 按执行 CAS 原子操作先后有以下六种顺序:(1) M_A, U_A, M_B, U_B;(2) M_A, M_B, U_A, U_B;(3) M_AM_B, U_B, U_A;(4) M_B, M_A, U_A, U_B;(5) M_B, M_A, U_B, U_A;(6) M_B, U_B, M_A, U_A.假设 A 线程先执行标记操作,即线程 A 在代码 $C3$ 处的 CAS 原子操作成功,下面分析执行顺序 1、2 和 3 这三种情况.

对于顺序 1,当线程 A 执行 U_A 时,由 $C16$ 处代码可知,此时 w 所在组的状态必定为 MARK_OK,因此线程 B 的 M_B 操作在 $C3$ 处 CAS 失败并在 $C14$ 处直接跳过标记阶段,而由于线程 A 先执行 U_A,即在 $C20$ 处的 CAS 成功更新 w,因此 U_B 失败,避免了对 w 的重复更新.

对于顺序 2,在线程 B 执行 M_B 时,w 的组状态还不确定,以图 2 示顺序 2 执行过程中各变量及操作的完成状态.由算法可知,M_B 在 $C3$ 处 CAS 失败,失败后线程 B 可能会执行 $C9$, $C12$ 或者 $C14$ 三处的代码,不考虑 w 被其他线程修改的情况下,线程 B 的执行取决于当前 w 组状态,但并不会造成对 w 的重复标记,且都能保证在 $C15$ 处对记录的 tag 执行正确的赋值.在更新阶段 $C20$ 处的 CAS 保证了 w 只被更新一次,而不会造成重复更新或者错误更新.

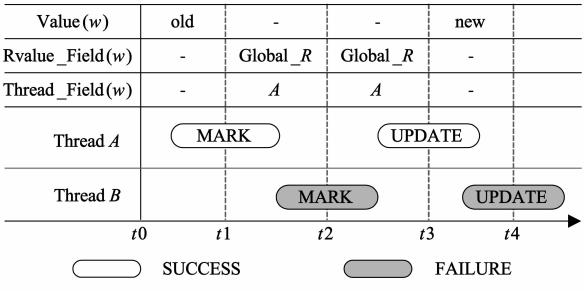


图2　执行顺序2的状态及相关变量变化情况

对于顺序 3,虽然由线程 A 对 w 进行成功标记,但最终的更新操作由线程 B 来完成,代码 $C19$ 处的判断保证了 Thread_Field(w)与记录中的 tag 一致,具体的执行及状态时序见图 3.

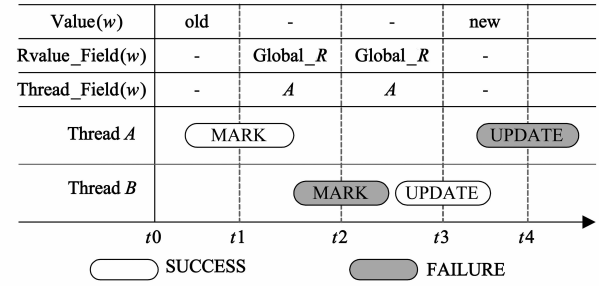


图3　执行顺序3的状态及相关变量变化情况

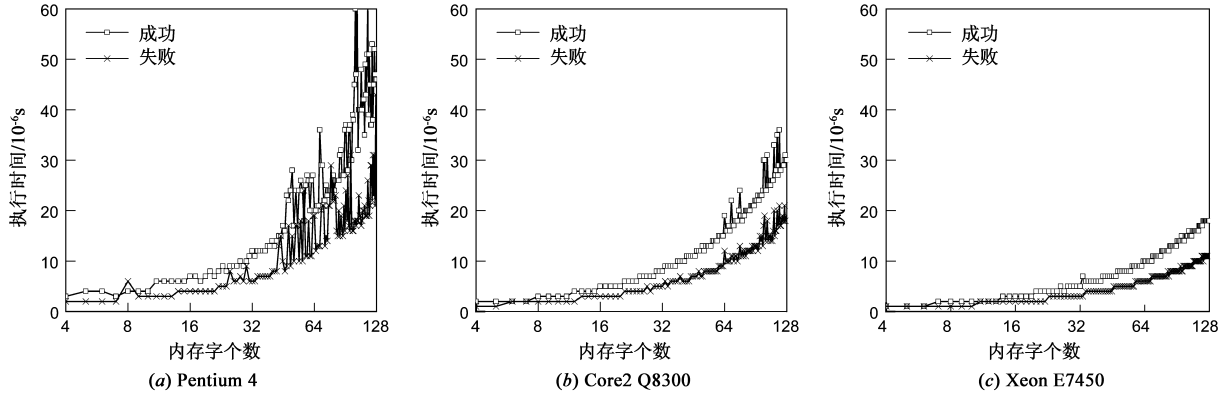
5　性能评价

为验证本文算法的性能,测试实验分别在三个不同的计算平台进行性能测试,具体配置分别为:(1)Pentium4 处理器,主频 2.40GHz,512KB 缓存,256MB 内存;(2)Core2 Q8300 四核处理器,主频 2.50GHz,2048KB 缓存,2GB 内存;(3)Xeon E7450 24 核心处理器,主频 2.40GHz,2MB 缓存,8GB 内存.操作系统内核版本为 Linux 3.0,编译环境为 GCC 4.5.

5.1　算法执行时间

本节实验首先在三个计算平台上测试了算法的执行时间,然后将本文的算法与 Fraser[16] 的方案进行了对比.在实验中设置的内存字个数从 4 递增到 128,记录每次运行时间,并根据随机产生错误期望值,测试了当比较条件失败的情况下算法的执行时间.实验结果如图 4 所示,在三个计算平台下当比较条件失败时平均执行时间约为成功的 50%,执行时间与并发同步的字个数呈线性增长.

图 5 是本文与 Fraser 方案的执行时间对比实验.实验对两种算法各重复运行 10 次,并将输入的内存字个数由 4 增加到 128,实验中记录两种方案的首次执行时间和其他 9 次的平均执行时间.从实验结果可以看出,两种方案的平均执行时间差别不大.而在首次执行时间方面,LMCAS 仅为 Fraser 方案的 10%.这是由于 Fraser 的方案是基于 2 比特/字的标记方式造成了额外的开销,而且其方案采取的提前绑定线程的方式也增加了额外开销.本文基于全局标记值方式有效的避免了这方面的开销,本文的算法在首次运行时间和平均运行时间上明显差别,因此可以看出本文算法具有较好的稳定性.



(*a*) Pentium 4 (*b*) Core2 Q8300 (*c*) Xeon E7450

图4 本文算法在不同平台下的执行性能

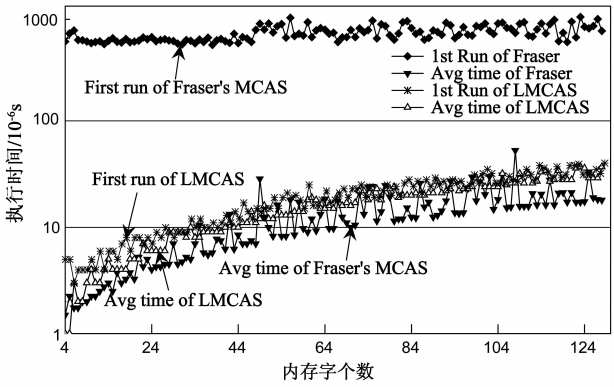
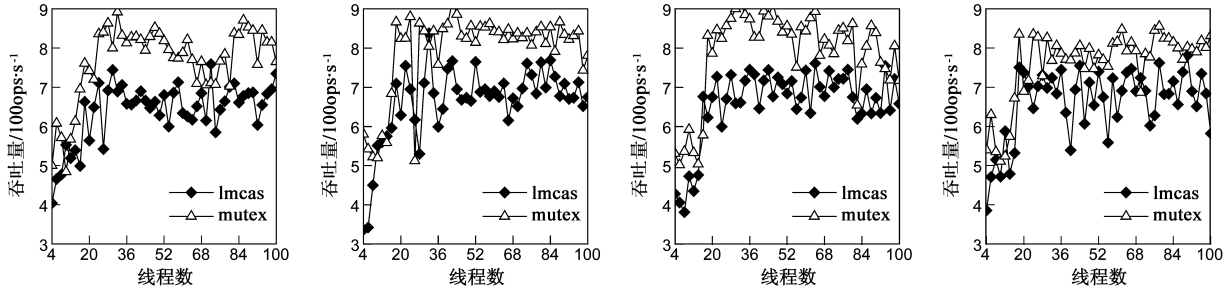


图5 LMCAS与Fraser方案的性能对比

5.2 吞吐量性能比较

由前面的实验可知,本文与 Fraser 方案在首次执行性能上差别较大,二者平均性能相当,在吞吐量方面的表现也是如此.因此本节实验仅比较了锁同步和本文 LMCAS 无锁同步的可扩展性差异,因为这两种方式不存在首次执行上的性能缺陷.本节实验构造了一个多线程竞争多个共享变量的基准测试.在测试中模拟 N (N = 8,16,32,64)个计数器累加更新操作,每进行一次迭代,计数器都要访问上一次各计数器的更新结果,因此每迭代一次都需要一次线程同步.为了观察多线程下对共享变量独占访问时间对可伸缩性的影响,以计数器累加的次数来模拟线程对共享变量独占访问时间的长短,用 D(D = 1,5,10)表示累加次数,D 越大表示对变量独占的时间越长.并发执行的线程数从 4 增加到 100,每次增加两个线程,用每秒执行百次同步操作(100ops/s:100 operations per second)表示吞吐量.实验以两种方式实现了该基准测试:一种是使用互斥锁(Mutex)进行线程同步,另一种使用本文设计的 LMCAS 实现无锁同步,实验结果见图 6.

由实验结果可知,在吞吐量性能来方面,只有当对共享变量的独占访问时间较短时(图 6 中的(a ~ d)),使用互斥锁方案才具有较好的吞吐量性能.但应该意识到这种情况下,任何一个真实的程序都不会只竞争锁或原子变量,而不做其他工作.而当 D = 5、10 时,LMCAS 的吞吐量性能明显优于互斥锁方案,并且当共享变量数量较多时,这种优势更为明显.在扩展性方面,当共享变量数量增加时,互斥锁方案的吞吐量性能退化明显,每增加一倍共享变量时其吞吐量性能降低 20% – 30%,随着并发线程数量增加,互斥锁方案的吞吐量性能也呈线性递减.而本文的算法能保持良好的吞吐量性能,具有较好的扩展性.



(*a*) 8 words, D=1 (*b*) 16 words, D=1 (*c*) 32 words, D=1 (*d*) 64 words, D=1

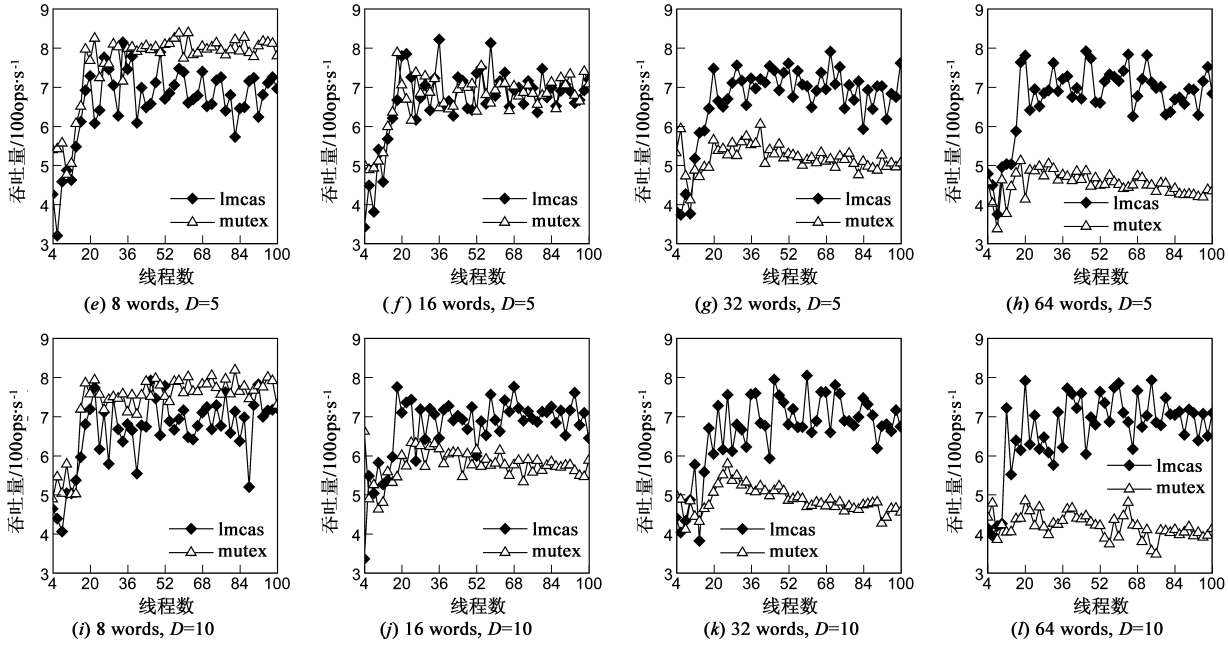


图6　LMCAS与互斥锁吞吐量比较

6　结论

本文提出了一种基于硬件 CAS 原语的无锁同步算法,该算法支持在多核多线程环境下对任意多个内存字的无锁同步.作为锁同步的一种替代方案,算法可以有效的避免由于锁竞争造成的程序串行化问题,提高了并发执行性能.本文将单字的 CAS 操作扩展成多字形式,克服了现有的计算机系统在硬件上不支持多字 CAS 同步的限制,可以更好的支持并发算法和并发数据结构的设计.算法采用全局标记值的方式,避免了传统设计中由于使用内存字标记占位导致的性能开销,同时确保数据在并发访问中的一致性.实验结果表明,算法在具有很好的并发访问性能和可扩展性.

参考文献

[1] M Herlihy, V Luchangco, P Martin, M Moir. Nonblocking memory management support for dynamic – sized data structures[J]. ACM Transactions on Computer Systems, 2005, 23(2):146 – 196.

[2] G Kliot, E Petrank, B Steensgaard. A lock – free, concurrent, and incremental stack scanning for garbage collectors[A]. Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments[C]. New York: ACM, 2009. 11 – 20.

[3] 杨际祥,谭国真,王荣生.多核软件的几个关键问题及其研究进展[J].电子学报,2010,38(9):2140 – 2146.
Yang Jixiang, Tan Guo zhen, Wang Rongsheng. Some key issues and their research advances for multi – core software[J]. Acta Electronica Sinica, 2010, 38(9):2140 – 2146. (in Chinese)

[4] 王睿伯,卢锡城,卢凯,王绍刚.面向 CC – NUMA 体系结构的事务内存冲突规避方法[J].计算机学报,2011,34(4):676 – 683.
Wang Ruibo, Lu Xicheng, Lu Kai, Wang Shaogang. CC – NUMA oriented conflict preventing method for transactional memory[J]. Chinese Journal of Computers, 2011, 34(4):676 – 683. (in Chinese)

[5] A Adl-Tabatabai, C Kozyrakis, B Saha. Unlocking concurrency [J]. ACM QUEUE, 2006, 4(10):24 – 33.

[6] A Dragojevic, P Felber, V Gramoll, R Guerraoul. Why STM can be more than a research toy[J]. Communications of the ACM, 2011, 54(4):70 – 77.

[7] D Dechev, B Stroustrup. Scalable nonblocking concurrent objects for mission critical code[A]. Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications[C]. New York: ACM, 2009. 597 – 612.

[8] M Herlihy. Wait-free synchronization[J]. ACM Transactions on Programming Languages and Systems, 1991, 13(1):124 – 149.

[9] J Giacomoni, T Moseley, M Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue[A]. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming[C]. New York: ACM, 2008. 43 – 52.

[10] H Sundell, P Tsigas. Lock-free deques and doubly linked lists [J]. Journal of Parallel and Dsitributed Computing, 2008, 68(7):1008 – 1020.

[11] D Cederman, P Tsigas. Supporting lock-free composition of concurrent data objects[A]. Proceedings of the 7th ACM International Conference on Computing Frontiers[C]. New York: ACM, 2010. 53 – 62.

[12] O Agesen, D L Detlefs, C H Flood, et. al. DCAS – based concurrent deques[A]. Proceedings of the 12th annual ACM Symposium on Parallel Algorithms and Architectures[C]. New York: ACM, 2000. 137 – 146.

[13] A Israeli, L Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives[A]. Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing[C]. New York: ACM, 1994. 151 – 160.

[14] J H Anderson, S Ramamurthy, R Jain. Implementing wait – free objects on priority – based systems[A]. Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing[C]. New York: ACM, 1997. 229 – 238.

[15] T L Harris, K Fraser, I A Pratt. A practical multi-word compare – and – swap operation[J]. Lecture Notes in Computer Science, 2002, 2508: 265 – 279.

[16] K Fraser, T Harris. Concurrent programming without locks [J]. ACM Transactions on Computer Systems, 2007, 25(2): 1 – 60.

[17] H. Sundell. Wait-free multi-word compare-and-swap using greedy helping and grabbing[J]. International Journal of Parallel Programming, 2011, 39(6): 694 – 716.

[18] K Fraser. Practical lock-freedom[D]. Cambridge, United Kingdom: Cambridge University Computer Laboratory, 2004.

[19] Intel. IA-32 Intel Architecture Software Developer′s Manual, Volume3: System Programming Guide[R]. Mt. Prospect IL: Intel Corporation, 2002.

作者简介

吴 昊 男, 1984 年生于辽宁沈阳, 哈尔滨工业大学计算机科学与技术学院博士研究生, 主要研究方向为并行计算、自动内存管理等.
E-mail: wuhaoster@hit.edu.cn

季振洲 男, 1965 年生于黑龙江哈尔滨, 哈尔滨工业大学计算机科学与技术学院教授、博士生导师, 主要研究方向为计算机系统结构、并行计算等.
E-mail: jizhenzhou@hit.edu.cn