采用向量时钟的软件事务存储算法

彭 林 谢伦国 张小强
(国防科技大学计算机学院 长沙 410073)

摘 要 在多核处理器上,事务存储是一种有望取代锁的同步手段。软件事务存储不需要增加额外硬件支持,就可以充分利用当前商业多核处理器的多线程能力。提出一种软件事务存储实现算法 VectorSTM,该算法不需要使用原子操作。VectorSTM 采用分布的向量时钟来跟踪各线程事务执行情况,能够提供更高的并发度。对事务存储基准程序 STAMP 的测试表明,VectorSTM 在性能或者语义上比软件事务存储算法 TL2 和 RingSTM 有优势。

关键词 多核处理器,软件事务存储,向量时钟
中图法分类号 TP31 **文献标识码** A

Vector Timestamp Based Software Transactional Memory Algorithm

PENG Lin XIE Lun-guo ZHANG Xiao-qiang
(School of Computer Science, National University of Defense Technology, Changsha 410073, China)

Abstract Transactional Memory(TM) is perceived as an alternative to locks for multi-core processors. Software transactional memory can run on commercial multi-core processors without additional hardware support, and make full use of them. We proposed VectorSTM a software transactional memory algorithm without employing any atomic instructions. VectorSTM employs distributed vector timestamps to track the progress of transactions and provides more concurrency. We evaluated VectorSTM with STAMP benchmarks and the results show that the design offers superior performance or stronger semantics than TL2 and RingSTM.

Keywords Multi-core processors, Software transactional memory, Vector timestamp

1 引言

多核处理器上的多个线程并发访问共享变量存在数据竞争时需要同步,以保持线程间不相互干扰。锁机制是一种常用的同步手段,但是粗粒度锁性能差,而细粒度锁性能较好,但使用过于复杂,容易出错,编写简单高效的代码非常困难,容易导致死锁、优先级倒置和锁护送(convoy)。

人们借鉴数据库中事务的思想,提出了事务存储(Transactional Memory, TM)的概念。事务存储中的事务为程序员提供一种同步抽象,在程序员看来,各事务线程以原子的方式执行。程序员能够较容易地对程序正确性进行推理,从而减轻编程的负担。事务存储把维持事务特性的工作交给了编译器、运行时库和硬件,进一步的性能调优可以在系统层完成。程序员在事务的抽象层次上,不用关心事务存储的实现细节,程序具有较好的可组合性和跨平台的可移植性。

目前的事务存储系统有基于硬件实现的 Hardware Transactional Memory(HTM),如 Stanford 大学的 TCC[1],基于运行时库实现的 Software Transactional Memory(STM),如 Rochester 大学的 RSTM[2] 和 Dice[3] 等提出的 TL2,以及结合软硬件实现的 Hybrid Transactional Memory(HybridTM),如 Rajwar 等提出的 Virtualizing Transactional Memory(VTM)[4]。HTM 性能优于 STM,但是要修改原有体系结构,它会受硬件资源的制约,对大事务支持差,灵活性也差,当使用资源超出硬件的限制时,需要借助软件的方法来解决。STM 不需要额外的硬件支持,具有良好的灵活性,能支持大事务,但是运行时开销较大,需要优化性能。目前基于硬件实现的事务存储离实用还比较远,还没有支持事务存储的商用多核处理器可以购买。STM 能有效利用现已大量投入使用的多核处理器,因此值得深入研究。

STM 的实现无需额外的硬件支持,但是其实现仍需借助于锁和原子操作,而原子操作的开销比普通读写操作要大,对性能影响明显。TL2 就是基于锁的 STM,在对每个变量最后写入结果时需要通过原子操作获取该变量对应的锁,保证其在写入结果的过程中对共享变量的独占,以防止出现数据竞争的情况,有多少个变量就需要多少个原子操作来获取锁,同时依赖于一个全局时钟来判断事务之间的次序,每次事务提交时都要通过原子操作对全局时钟增加来获得自己的时钟。RingSTM[5] 采用基于 Bloom Filter 原理[6] 的冲突检测方法,

到稿日期:2009-06-08 返修日期:2009-08-01 本文受核高基科技重大专项(2009ZX01036-001-003-001),863 国家重点基金项目(2008AA01Z110)资助。

彭 林(1979—),男,博士生,CCF 会员,主要研究方向为多核处理器编译环境与编译优化技术,E-mail:penglin@nudt.edu.cn;谢伦国(1947—),男,研究员,博士生导师,CCF 高级会员,主要研究方向为计算机体系结构、存储系统体系结构、高性能微处理器设计;张小强(1973—),男,博士,主要研究方向为编译优化和并行计算。

大大减少了原子操作的数目，事务提交时只需要一个原子操作来获得自己提交的时钟。随着线程数目的增加和需要提交的事务增多，每个线程都需要以原子操作修改全局时钟，对全局时钟变量原子的访问成为软件事务存储的性能瓶颈。如果能够完全消除原子操作，并提高事务提交时的并发度，就能够进一步提高软件事务存储的性能。

为了消除原子操作带来的开销，提出了一种向量全局时钟算法 VectorSTM，该算法不需要任何原子操作，能够有效降低由于全局时钟竞争带来的开销。同时 VectorSTM 能够保证使用事务对共享变量进行私有化时的安全[7]。对事务存储基准测试程序 STAMP 的测试结果表明，VectorSTM 能够在性能或者语义上优于 TL2 和 RingSTM。

2 TM 实现简介

事务存储的实现要在事务代码并发执行的情况下获得事务之间串行次序的效果，即当前事务所读的数据不会被其他事务写，写的数据也不会被其他事务读写，只有在事务结束时，对共享变量的修改才为其他事务所见。为此，事务存储的实现必须具备冲突检测(conflict detection)、数据版本管理(data version management)和冲突消解(conflict resolution)三大基本功能。

事务读和写的共享变量分别构成该事务的读集合(read set)和写集合(write set)。当两个并发的事务同时访问同一个数据，且至少有一个是写操作时，我们称这两个事务间存在冲突(conflict)。一个事务如果在其执行期间没有发生冲突，那么它可以提交(commit)，其对内存中共享变量的修改生效并为其他事务所见；如果发生冲突，根据冲突消解策略，裁决某个事务作废(abort)。作废事务必须撤销已完成的操作，恢复事务开始执行时的共享变量状态。事务没有提交或作废时称为活跃(live)状态。

冲突检测就是要发现和判定系统中并发执行的事务之间是否发生冲突，通过对事务读集合和写集合中变量的检查来实现。冲突检测策略决定何时进行冲突检测，急切(eager)策略在对数据进行读写的时候立即进行检测；懒惰(lazy)策略则在提交阶段才进行检测，通常采用这种策略来提高性能。而冲突消解就是在事务间发生冲突后决定哪些事务能够继续执行或者提交，哪些事务必须作废，撤销已经完成的操作，重新执行。冲突消解策略对事务的吞吐量影响较大。

为了保证事务在发生冲突时可以作废回滚，必须同时保存更新的数据和原始的数据，并且要控制数据在哪些情况下对其他事务可见，这就是数据版本管理。数据版本管理能够让事务并发修改共享变量而不互相干扰，支持前瞻执行，提高程序的内在并行性。数据版本管理包括急切方式和懒惰方式，急切方式直接修改存储器中的共享变量，同时备份共享变量原来的值；懒惰方式则将共享变量修改值缓冲于本地，直到提交时才将修改写入存储器中。为了保证各事务具有一致的存储视图，急切方式要防止其他事务在当前事务没有提交时读取其对共享变量的修改，通常采取加锁的方式，而懒惰方式则只需要保证在其提交写入修改值的过程中，其他事务不能访问其修改的共享变量。

事务存储实际上是通过前瞻执行，动态检测线程间的数据相关来发掘程序中的并行性。可以想像，只有在任何两个并发的事务之间都存在冲突的最坏情况下，每次只有一个事务能够提交，其他事务需要重新执行时，才会出现事务真正串行执行的情况，否则事务总是可以并行执行的。

3 VectorSTM 的设计与实现

3.1 VectorSTM 的基本算法

VectorSTM 的数据版本管理采用写缓冲的方式，对共享变量的写操作先缓冲到本地，提交时仍然未发生冲突，则将修改写入主存。采用写缓冲的方式，保证对共享变量的更新只有在事务提交时才为其他线程所见。

VectorSTM 采用 Bloom Filter[6] 记录读写集合，并且通过比较事务间的 Bloom Filter 位向量值来检测冲突。如图 1 所示，每个线程对应一个队列，队列中的每一项都包含记录读集合和写结合的 Bloom Filter 位向量，还有其他一些相关数据。各线程的队列构成 Bloom Filter 队列元组(Bloom Filter Queue Array，BFQA)。每当一个事务提交时，执行线程先将该事务的读写集合的 Bloom Filter 位向量更新到自己的队列中，然后依次检查其他线程队列中的写集合，看其他已经提交的事务是否写了自己读的数据，自己要写的数据是否与其他正在提交的事务要写的数据存在冲突。每次读共享变量时，依次检查其他线程是否更新了队列，对本事务所有已读数据进行验证，保证读到的数据没有被别的事务修改。这两个时刻的冲突检测能够保证当前事务所读的数据不会被其他事务写，写的数据则不会被其他事务写。

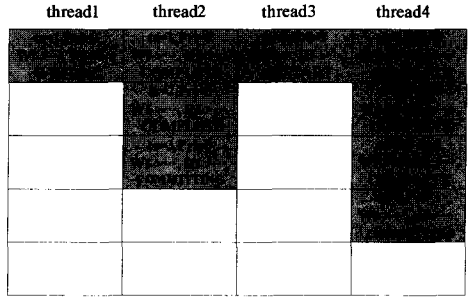

图 1 线程执行事务过程中 BFQA 中的信息

事务执行中未遇到冲突时，事务继续执行后续操作。如果其他事务已经提交并修改了自己读取的数据，则作废前面完成的操作，如清空写缓冲，事务重新执行。如果需要提交的事务发现其他事务正在写自己要写的数据，可以选择等待该事务完成后再写入，或者作废重新执行。如果两个事务发生冲突时，发现冲突的事务作废自己，并且随机后退(back off)，重新执行。

VectorSTM 通过一个全局向量时钟 global vector timestamp (GVT)索引一个 Bloom Filter 队列元组 BFQA，来检测冲突，保证事务执行的原子性、一致性和隔离性。每个线程拥有 BFQA 中一个 Bloom Filter 位串(bit vector)队列(Bloom Filter Queue，BFQ)，BFQ 中每一项包含读写集合 Bloom Filter 位串和其他一些数据，其他数据的作用在下面介绍。VectorSTM 通过 GVT 中每个时钟指示每个线程当前执行的事务在 BFQ 中的位置。每个执行的事务有一个局部向量时钟 local vector timestamp (LVT)，记录该事务看到的 GVT 快照值，另外还有两个局部的 Bloom Filter 位串，分别记录该事务的读写集合。VectorSTM 中事务的写操作都缓冲起来，当事

务成功提交时写入内存。

3.2 向量时钟的发生序(happen before)关系

在 VectorSTM 中,每个线程可能以不同的速度并发执行事务。每个事务在 LVT 中记录 GVT 的一个快照,不妨设该事务为 A 事务。一个线程 i 提交一个事务时,该线程对应的时钟 GVT[i]增加 1,记录读写集合的两个 Bloom Filter 位串存放到 BFQA 中该线程对应的 BFQ 顶部,位置由时钟 GVT[i]指示。A 事务验证时检查 GVT,由于线程 i 提交的是新的事务,LVT 中 LVT[i]就会与 GVT[i]有所不同。在上次快照与这次检查之间线程 i 发生了一些事件,这种事件先后关系,即发生序关系要维持,A 事务要看到线程 i 完成事件的效果。进而 A 事务会检查 BFQA 中线程 i 的 BFQ,从 BFQ 的位置 LVT[i]到 GVT[i]。这种方法保证了 A 事务看到线程 i 中所发生的事件,即线程 i 提交的事务写的变量以及其他的一些情况。比较整个 GVT 与 LVT,可以查看其他所有线程是否有动作。

3.3 分布的并发控制

采用一个全局时钟能够降低并发控制的难度,但是缺点是可能成为性能瓶颈。对于并行程序而言,每个线程都要竞争读写这个全局时钟,而且需要以原子操作的方式进行,当竞争很激烈的时候,这个全局时钟将成为瓶颈。尤其是随着线程规模的增大,延迟也越来越大,性能将大大下降。分布的向量时钟能够缓解这种情况。

假设一个线程一次只执行一个事务,事务执行过程中需要完成的操作实际上是由执行该事务的线程完成的。每个线程在 BFQA 中拥有对应的 BFQ,如图 1 所示。BFQ 的一项包含了表示该事务情况的信息:包括当前项的读写集合 Bloom Filter 位串 Wf 和 Rf、事务状态 State、时钟 ts、优先级 Pri。事务执行中有 4 种可能状态:COMPLETE,COMMITTING,WRITING 和 RESTART。COMPLETE 状态表示当前事务已经提交完成;COMMITTING 状态表示当前事务想进行提交操作;WRITING 状态表示当前事务已经获得提交的许可,正在往内存写入缓冲的写操作;RESTART 状态表示当前事务由于某种原因被作废,重新执行。优先级 Pri 可以根据竞争管理策略来处理,如每次事务被作废掉,其优先级就增加 1。

图 1 中显示,线程 2 和线程 4 提交事务时,它们的状态均为 COMMITTING。假设线程 2 最近一次检测后的 LVT 值为[0,3,0,2],而当前的 GVT 值为[1,3,1,4]。通过比较 LVT 与 GVT,线程 2 需要检测其他每个线程的 BFQ。所有线程检测其他线程 BFQ 的顺序必须是相同的,否则会导致不一致的情况出现。

首先线程 2 检测线程 1 的 BFQ,发现线程 1 处于 WRITING 状态。这表示线程 1 已经获得提交的许可,正在往内存写入缓冲的写操作。线程 2 要确保线程 1 所写的变量中不包含线程 2 以及读过的变量。如果包含,线程 2 需要作废自己的事务,重新执行。线程 2 处于 COMMITTING 状态,要检测线程 1 当前写的变量中是否包含自己要写的变量。如果包含,线程 2 可以采取两种方法解决:作废自己执行的事务,或者等待线程 1 完成写操作。如果在线程 1 完成操作后,线程 2 可以将自己的 LVT 更新为[1,3,0,2],并继续对线程 3 进行检测。更新 LVT 的值是为了避免下次对已经完成的操作再次检测。值得指出的是,如果线程 2 不是处于 COMMITTING 状态,而只是对一个读操作进行验证,那么线程 2 不能更新 LVT 中 LVT[1]的值。线程 2 因为不能确定下一次读操作访问的变量是否在线程 1 所写的变量中,需要在下次再次检测。

线程 3 的 BFQ 顶端的项中内容显示线程 3 的状态是 RESTART,即在执行一个作废的事务后重启事务。线程 3 不会对线程 2 的提交产生威胁,因此线程 2 可以忽略对线程 3 的 BFQ 中读写集合的检测,并更新 LVT 到[1,3,1,2]。同样,如果线程 2 只是对一个读操作进行验证,也不能更新 LVT[3]的值,因为线程 3 可能在下次读之前会发生状态转换,需要线程 2 再次检测。

线程 2 当前的 LVT 值为[1,3,1,2],发现 LVT[4]与 GVT[4]相差 2,有多个 BFQ 项需要检测。在 BFQ 顶端的事务才是活跃的事务,其他都是已经提交完成的事务,因此对于完成的事务只需要检测自上次检测以来,这些完成的事务是否对当前事务读的数据进行了写操作。线程 2 对线程 4 的 BFQ 中第 3 项只需要检测该事务是否写了线程 2 当前事务读的数据。如果有冲突,线程 2 要作废当前事务。如果没有冲突,线程 2 继续检测线程 4 的下一个 BFQ 项,线程 4 的 BFQ 的顶端的项。

注意到线程 4 的顶端的项也处于 COMMITTING 状态,线程 2 需要检测自己的读写集合是否与对方的读写集合存在冲突。如果没有冲突,线程 2 获得提交的许可,进入 WRITING 状态,在写操作结束后,进入 COMPLETE 状态。如果有冲突,简单的处理方法是作废线程 2,但有可能产生不必要的作废。如线程 4 已经看到线程 2 的读写集合,已经作废了线程 4 执行的事务,那么线程 2 就没有必要作废自己执行的事务了,而且两个线程如果不停地互相作废自己执行的事务,可能导致活锁,较为简单的解决办法是采取随机后退(back-off)的办法。在 VectorSTM 这种分布并发控制情况下,需要小心处理,以避免出现错误或者活锁的情况。

3.4 冲突检测与竞争管理

维持一个读写操作的地址集合,然后或者逐个比较读写集合的地址来发现冲突,或者数据记录版本号,逐个比较数据版本号,这些方法随着读写集合的增大,计算量不断增加。而且在提交时逐个获取写集合变量锁的方式,可能会因为获取锁的顺序问题引发死锁,这可以通过作废当前事务,以及随机后退或指数后退的方式消除,但获取锁的开销仍然不小。

在 VectorSTM 中,事务之间的冲突是通过比较事务读写集合的 Bloom Filter 位串来实现的。Bloom Filter 采用一定长度的位串记录读写集合,存储空间开销小,而且对两个集合进行比较时只需要比较两个位串,使得对读写集合的记录和比较为常数时间,性能优于记录地址集合的方法。但 Bloom Filter 的问题是存在误判率,会将两个不冲突的事务判为冲突,错误地作废了有效工作,从而降低性能,这是我们需要进一步优化的,可以采用硬件实现的方法来解决,对硬件系统的改动相对较小。

竞争管理所采用策略的效果往往与具体的应用有关,研究表明没有哪种策略是普适的[8],因此这里采用较为简单的随机后退的办法,也有利于消除活锁。

3.5 确保私有化过程的安全

Spear 等[7]深入探讨了事务存储中共享变量的私有化问题以及解决方法，线程可以在共享变量私有化后单独处理，不需要事务，能够有效提高性能，同时消除 STM 语义上的弱点。文章指出导致共享变量私有化过程出问题的原因有两个：1)私有化代码不能导致"僵尸"事务(肯定会作废但当前还在执行的事务)执行错误的外部可见的操作；2)私有化代码没有看到之前提交的事务对私有化共享变量的修改。

要消除第一个因素，需要及时验证当前事务读取的数据是否被其他事务写。如果能够保证当前事务使用的数据都是一致的，那么可以消除僵尸事务的存在，从而私有化代码也不会与僵尸事务发生冲突。VectorSTM 在每次读取数据和最后提交时，都对已经读取的数据进行验证，保证事务使用数据的一致性，消除了僵尸事务的存在。

对第二个因素，VectorSTM 中，事务 A 在提交时，如果发现存在事务 B 进入 WRITING 状态比自己早，而且 B 读取的数据与 A 写的数据存在冲突(通过比较读写集合位串)，那么事务 A 会等待事务 B 进入 COMPLETE 状态，以保证事务 A 后的代码能够看到事务 B 对共享变量的修改。这种方法与 Spear[7]提出的采用加锁的策略类似，事务 A 对事务 B 读集合的检查就好像事务 B 对读取变量加了锁。

3.6 VectorSTM 的实现

VectorSTM 实现算法的基本数据结构包括：枚举类型 VectorEntryState 表示事务的 4 中状态：RESTART，WRITING，COMMITTING 和 COMPLETE；VectorEntry 是 BFQ 中的一项，如图 1 所示，其中包括两个 Bloom Filter 位串 wf 和 rf，分别表示写集合与读集合，优先权 pri 记录当前事务的优先级，st 是表示当前事务状态的 VectorEntryState 枚举类型。宏 LENGTH 定义了 BFQ 的长度。根据线程数目和 BFQ 长度定义 BFQA：VectorEntry 数组 BFQ。无符号整形数组 GVT 和 LVT 分别表示全局向量时钟和本地向量时钟，其中 GVT 是 volatile 易失型数据，确保不会由于编译优化而使得线程不能看到 GVT 的最新值。结构体 Transaction 则表示一个事务的基本数据结构，包括缓冲写操作的 wset，读写集合 Bloom Filter 位串 wf，rf，以及用于表示事务嵌套深度的变量 nesting_depth，每当提交时，事务会把自己的读写集合 wf，rf 复制到 BFQ 中。

由于每次验证需要轮询每个线程的状态是否变化，开销较大，我们采用 volatile 的无符号整形变量 global_change 来表示当前线程是否有新动作。每次线程改变状态时，首先将 global_change 读到本地临时变量 tmp，然后对 tmp 加 1，通过循环，将 tmp 赋值给 global_change，保证 global_change 不小于 tmp 即可，不需要原子操作，因为其他线程验证时只需要看 global_change 是否发生变化，就能决定是否轮询 GVT，并不需要知道多少事务发生了状态改变。

在 Intel 多核处理器平台实现时，由于 Intel 采用的存储一致性模型放松了"写-读"的次序，我们指出在实现中需要在事务 i 提交时，在修改 GVT[i]和 BFQ[i]中状态后，在读取验证其他线程 GVT 前，需要加一个同步操作，保持这两部分操作之间的"写-读"顺序，否则会因为存储一致性模型的放松而导致错误。

VectorSTM 中提供了 6 个基本函数，包括 TM_begin()，TM_write(addr，val)，TM_read(addr)，TM_end()，TM_abort()和 check()。TM_begin()用于开始一个事务，初始化相关数据，TM_end()用于提交事务，TM_write(addr，val)和 TM_read(addr)分别对共享变量进行写和读操作，check()用于检查读取数据一致性，TM_abort()用于发生冲突时作废事务。

4 实验与分析

在本节比较 VectorSTM，RingSTM 和 TL2 的性能。其实现测试平台是 Xeon SMP Linux 服务器，Linux 版本 2.6.28，2.6GHz双 Xeon x5355 处理器，每个处理器 4 核，4GB 内存。编译器 GCC 4.3.2 使用-O3 优化开关。

STAMP[9]是斯坦福大学开发的事务存储性能测试基准程序，STAMP 中自带 x86 平台上可执行的 TL2。在 RSTM 的 RingSW 是 RingSTM 的一个简化实现，性能与 RingSTM[5]非常相近。选取了 labyrinth，bayes 和 ssca2 进行测试，分别给出了 1，2，4 和 8 个线程的执行情况。

图 2 中对于 labyrinth，VectorSTM 比 TL2 和 RingSW 具有更好的可扩展性。其中在 4 线程时比 TL2 性能高 14.8%，在 8 线程时比 TL2 性能高 27%。这是由于采用了开销小的 Bloom Filter 进行冲突检测，另一方面是采用了向量时钟，不需要原子操作。比 RingSW 性能好，是因为不需要原子操作。

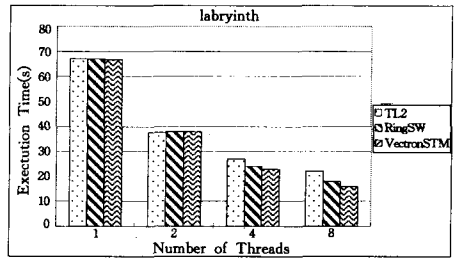

图 2 labyrinth 使用 TL2，RingSW 和 VectorSTM 的执行结果

图 3 中对于 bayes，1 个线程时，VectorSTM 和 RingSW 比 TL2 在 bayes 上高 18.9%，因为 VectorSTM 和 RingSW 采用 Bloom Filter 进行冲突检测，比 TL2 使用地址集合的方式开销更低，但随着线程数目增加，误判率对性能的影响显示出来。

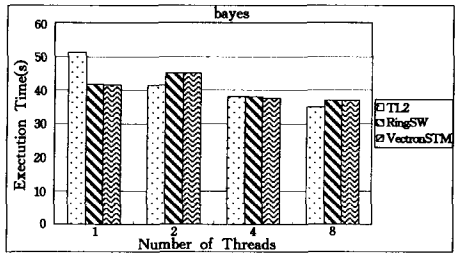

图 3 bayes 使用 TL2，RingSW 和 VectorSTM 的执行结果

图 4 中对于 ssca2，1 个线程时，VectorSTM 和 RingSW 比 TL2 在 bayes 上高 23.5%，在两个线程时仍有一定优势，但 4 和 8 个线程时，误判率导致二者性能明显低于 TL2。RingSW 在 ssca2 中 8 线程时，性能比 VectorSTM 差 41.1%，原因来自原子操作的开销。

VectorSTM 与 RingSW 比有明显的性能优势。在 bayes 和 ssca2 上性能比 TL2 差，但我们强调 TL2 不能保证共享变量私有化过程的安全，而私有化过程的安全对事务存储非常重要[7]，因此 VectorSTM 有更强的语义。

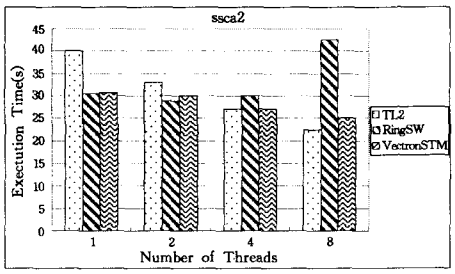

图 4　ssca2 使用 TL2，RingSW 和 VectorSTM 的执行结果

结束语　本文提出了一种采用向量时钟的不需要原子操作的 STM 实现算法 VectorSTM。VectorSTM 能够保证共享变量私有化的安全，而且实验数据表明 VectorSTM 有较好的可扩展性，在一些问题上还具有性能优势。但是 VectorSTM 性能受到误判率影响很严重，下一步我们将探索如何有效降低误判率，进一步提高 VectorSTM 的性能。

参 考 文 献

[1] Hammond L, Carlstrom B D, Wong V, et al. Programming with *transactional coherence and consistency*(TCC)[J]. ACM SIGPLAN Notices, 2004, 39(11): 1-13

[2] University of Rochester Department of Computer Science. Rochester Software Transactional Memory[EB/OL]. http://www.cs.rochester.edu/research/synchronization/rstm/

[3] Dice D, ShalevN O, Shavit. Transactional Locking II[C]// Proc. of the 20th Intl. Symp. on Distributed Computing. Berlin: Springer, 2006: 194-208

[4] Rajwar R, Lai M H K. Virtualizing Transactional Memory[C]// Proc. of the 32nd Annual Intl. Symp. on Computer Architecture (ISCA). Los Alamitos, CA: IEEE, 2005: 494-505

[5] Spear M F, MichaelC M M, von Praun. RingSTM: scalable transactions with a single atomic instruction[C]// 20th ACM Symposium on Parallelism in Algorithms and Architectures. New York: ACM, 2008: 275-284

[6] Blustein J, El-Maazawi A. Bloom Filters-A Tutorial, Analysis, and Survey[R]. Halifax, NS: Dalhousie University, 2002: 1-31

[7] Spear M F, Marathe V J, Dalessandro L, et al. Privatization Techniques for Software Transactional Memory[R]. TR 915. Dept. of Computer Science, Univ. of Rochester, Feb. 2007

[8] Guerraoui R, Herlihy M, Pochon B. Polymorphic contention management[C]// 19th International Symposium on Distributed Computing. 2005

[9] Minh C C, Chung J, Kozyrakis C, et al. STAMP: Stanford transactional applications for multi-processing[C]// IEEE International Symposium on Workload Characterization. Los Alamitos, CA: IEEE, 2008: 35-46

(上接第 180 页)

示爬上法，SAKS 表示模拟退火法，TABUKS 表示禁忌搜索法。

由表 3 可知，本文提出的离散粒子群算法对于较大规模的背包问题其搜索效果要比传统的一些方法好得多。

表 3　DPSO 与传统方法所求结果的比较

物品数	100	250	500
方法	最大价值	最大价值	最大价值
DPSO	496.8714	1172.1	2241.9
HILLKS	412.74	1035.9	2032.3
SAKS	415.27	1045.8	2048.8
TABUKS	429.47	1111.8	2120.1

结束语　本文借鉴蚁群算法的信息素机制对粒子进行更新，应用蚂蚁的混沌行为来初始化粒子群，并且在一定的条件下重新初始化，得到一种基于蚂蚁混沌行为的离散粒子群算法；将其应用到背包问题中，并将结果与改进的遗传算法、改进的进化算法、改进的粒子群算法及传统一些求解背包问题算法相比较。结果显示，本方法能得到较好的实验结果。

参 考 文 献

[1] Kennedy J, Eberhart R C. Particle Swarm Optimization[C]// Proceedings of the IEEE International Conference on Neural Networks. Perth, Aust: IEEE Piscataway, 1995: 1942-1948

[2] Kennedy J, Eberhart R C. A Discrete Binary Version of Particle Swarm Algorithm[C]// Proceedings of the 1997 Conference on System, Man, and Cybernetics. Piscataway: IEEE Press, 1997: 4104-4108

[3] Shi Y, Eberhart R C. Fuzzy Adaptive Particle Swarm Optimization[C]// Proceedings of the IEEE Conference on Evolutionary Computation, Soul: IEEE, 2001: 101-106

[4] Noel M M, Jannett T C. Simulation of a new hybrid particle swarm optimization algorithm[C]// Proc. of the 36th Southeastern Symposium on System Theory. Atlanta: IEEE, 2004: 150-153

[5] 贾东立，张家树. 基于混沌变异的小生境粒子群算法[J]. 控制与决策，2007，22(1)：117-120

[6] 钟一文，杨建刚，宁正元. 求解 TSP 问题的离散粒子群算法[J]. 系统工程理论与实践，2006，6：88-94

[7] Kashan A H, Karimi B. A discrete particle swarm optimization algorithm for scheduling parallel machines [J]. Computers & Industrial Engineering (2008), doi: 10.1016/j.cie.2008.05.007

[8] Garey M R, Johnson D S. Computers and intractability: A guide to the theory of NP-completeness[M]. San Francisco: Freeman W. H and Co, 1979

[9] 张铃，张钹. 佳点集遗传算法[J]. 计算机学报，2001，24(9)：917-922

[10] Li K S, Jia Y Z, Zhang W S, et al. A new method for solving 0/1 knapsack problem based on evolutionary algorithm with schema replaced[C]// Proceedings of the IEEE International Conference on Automation and Logistics. Qingdao, 2008: 2569-2571

[11] Wang Y, Feng X Y, Huang Y X, et al. A novel quantum swarm evolutionary algorithm and its applications[J]. Neurocoputing, 2007, 70: 633-640

[12] Cole B J. Is animal behavior chaotic? Evidence from the activity of ants[J]. Proceedings of The Royal Society of London Series B-Biological Sciences, 1991, 244: 253-259