计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2018.20170863
55(2):305-318,2018

面向数据库的持久化事务内存

Hillel Avni　　王　鹏
(华为技术有限公司　广东深圳　518129)
(hillel.avni@huawei.com)

Persistent Transactional Memory for Databases

Hillel Avni and Wang Peng
(*Huawei Technologies Co. Ltd.*, *Shenzhen*, *Guangdong* 518129)

Abstract　Hardware transactional memory (HTM) and byte-addressable nonvolatile memory (NVM) are already available in new computer equipment. It is tempting, but not trivial, to combine them to implement transactions having the capabilities of ACID (atomicity, consistency, isolation and durability), by using HTM for consistency and isolation, and NVM for durability. ACID transactions are especially useful in databases but, because of the size of database transactions, the challenge is to cope with the inherent HTM limitations of size and contention level. In this paper, we first present persistent HTM (PHTM), a software-hardware solution for ACID transactions with HTM. We continue with two methods to mitigate PHTM limitations. One is a persistent hybrid TM algorithm called PHyTM, which allows PHTM transactions to execute concurrently with pure software, unbounded transactions. The other is for workloads where most transactions are too large for PHTM. For the purpose we propose a new algorithm called split transactions execution (STE), which is tailored for relational database transactions. In a nutshell, this paper discusses the extension of HTM to ACID database transactions on NVM.

Key words　hardware transactional memory (HTM); nonvolatile memory (NVM); database transaction; multicore; shared memory; ACID; concurrency

摘　要　硬件事务内存(hardware transactional memory, HTM)和可字节寻址的非易失性内存(nonvolatile memory, NVM)已经可以在新的计算机设备中使用.使用 HTM 确保一致性和隔离性,使用 NVM 确保持久性,组合使用两者可以实现满足原子性、一致性、隔离性和持久性(atomicity, consistency, isolation and durability, ACID)特性的事务.ACID 事务在数据库中非常有价值,但由于数据库事务通常较大,其面临的挑战是 HTM 固有的容量限制和争用水平.首先提出了一种通过 HTM 进行 ACID 事务处理的软硬件解决方案——持久化 HTM(persistent HTM, PHTM).使用 2 种方法来消除 PHTM 的局限性:1)持久化混合事务内存(persistent hybrid TM, PHyTM),允许 PHTM 事务与支持任意大小的纯软件事务(software transactional memory, STM)并发执行;2)分离事务执行(split transaction execution, STE)算法,该算法为关系数据库事务量身定制,解决了大多数事务超过 PHTM 的容量限制的问题.简而言之,讨论了利用 NVM 将 HTM 扩展到 ACID 数据库事务的问题.

关键词　硬件事务内存;非易失性内存;数据库事务;多核;共享内存;ACID 特性;一致性

中图法分类号　TP391

收稿日期:2017-11-15;**修回日期**:2017-12-25

通信作者:王鹏(wangpeng.mail@huawei.com)

非易失性内存(nonvolatile memory, NVM)是一种新兴的技术,有望在计算机内存领域掀起一场变革.

研究人员刚刚开始了解如何在配置了非易失性内存的机器上进行编程.非易失性内存的编程模型仍然在不断变化中,但当前下列模型处于领先地位.系统可以仅使用 NVM,也可以将 DRAM 和 NVM 组合使用.在任何时候数据都会异步刷新到 NVM,该过程对程序员透明.程序员也可以通过显式调用刷新原语(Flush)将数据刷新到 NVM.而持久化屏障(persistence barrier)原语允许阻塞进程,直到已将数据刷新到 NVM.

处理器的缓存和寄存器将满足易失性还是非易失性仍存在重大争议.一些研究人员正在研究如何在意外掉电时提供足够的电能将这些数据刷新到非易失性内存中[1].这种方法使应用程序无需为持久化增加任何运行开销,因为整个处理器的状态是持久的.但是,硬件设计人员对其可行性持怀疑态度,他们担心将处理器缓存刷新到 NVM 需要消耗很多电能,因为缓存容量非常大,处理器非常复杂而耗电.他们表示未来的硬件只能使用余电保存 NVM 控制器的易失性缓冲区中的数据.英特尔目前正在基于这个假设并考虑了 NVM 而设计新的高效的 Flush 指令 CLWB 和 CLFLUSHOPT[2].这些新指令假设缓存为易失性,并允许刷新的数据保存在缓存中,以避免缓存缺页.这些指令也将加速刷新到 NVM.

我们考虑处理器缓存和寄存器不具备非易失性的系统.在这样的系统中,关键挑战是发生掉电并且缓存和寄存器被清除时仍能确保 NVM 总是处于一致的状态.

另一个被称为硬件事务内存(hardware transactional memory, HTM)的新技术最近在英特尔处理器中得以实现,该技术将数据库中的事务概念引入共享内存(HTM 也在 IBM 的生产系统和各种研究系统中得以实现,本文仅考虑英特尔的实现).HTM 允许程序员在事务中执行任意代码块,这些代码块将以原子方式完成提交,或中止而不影响内存中原有的内容.英特尔的 HTM 实现采用尽力提交的方式,这意味着没有事务可以保证提交.因此,当一个事务中止足够多次后,必须由程序员指定并执行其非事务的回退路径.最简单的回退路径是全局锁定(即阻止其他进程执行事务)并重新执行事务.但是,这种方法不适用于 NVM,因为任何修改后的数据都会异步地刷新到 NVM(可能会违反原子性).

HTM 的实现和 NVM 方案之间的相互作用特别复杂,因为事务必须是原子的,但是在回退路径上执行的写入可以在任何时间异步刷新到 NVM.因此,必须仔细设计回退路径,以避免在发生掉电故障时将正在进行的事务的部分结果暴露给其他进程.另一个复杂之处在于 HTM 不能直接修改主存,这样事务对共享内存的任何修改都是在执行该事务的处理器核的本地缓存中的数据副本上执行的.这样,在事务提交与从缓存刷新到 NVM 之间存在一个时间窗.在此时间窗内发生电源故障可能会导致部分或全部已提交事务的结果丢失.

Avni 等人在最近的工作[3]中引入了 PHTM 算法,该算法允许硬件事务在包含 NVM 的系统中运行.必须修改现有的 HTM 实现以支持 NVM,因为至少需要增加一个用于标记事务是否已提交的比特位作为 HTM 提交的一部分,与 HTM 一起原子性地刷新到 NVM,否则,掉电故障可能导致已提交的事务丢失.Avni 等人建议修改英特尔的 HTM 实现,允许将单个比特位作为事务提交的一部分原子性地刷新到 NVM.PHTM 使用此功能来维护重演日志(redo log),以确保已提交的事务不会丢失.具体来说,允许 PHTM 同时提交一个事务,并将 NVM 中的重演日志标记为已完成(completed),以便在掉电后当且仅当事务已提交时才进行重演.PHTM 中的回退路径是面向 NVM 设计的持久化软件事务内存(persistent software transactional memory, PSTM),这是一种软件事务内存.不幸的是,PSTM 顺序执行所有事务,并且该算法不支持硬件事务和软件事务之间的并发.这消除了在回退路径上执行时的所有并发性,并且使得该算法无法随着 HTM 系统中处理器核数的增加而扩展.

事务内存(transactional memory, TM)的相关文献引入了混合事务内存(hybrid transactional memory, Hybrid TM)以解决此类性能问题.混合事务内存通过使用在回退路径上允许并发的软件事务内存算法来提高性能,并设计快速路径算法,以便硬件和软件事务可以同时运行.然而,现有的混合事务内存算法不适用于 NVM,所以需要新的算法.

本文介绍了第一种面向 NVM 的混合事务内存系统——PHyTM.与 PHTM 一样,PHyTM 通过重演日志进行掉电后的恢复.PHyTM 提供了无死锁、无活锁的原子事务.它有 3 种执行路径:1)快速

HTM(fast HTM),具有不插桩的硬件读取速度;2)慢速 HTM(slow HTM),可以进行插桩的读、写;3)软件事务内存,锁定读集合和写集合,并缓冲所有的写操作,直到事务提交时的写回(write-back)阶段.为了避免在极少数情况下发生活锁,STM 路径上的事务可能会使用一个粗粒度的锁排除 STM 路径上的其他事务,但允许硬件事务继续.

快速 HTM 和慢速 HTM 可以同时运行(因为两者都使用 HTM,因此硬件可以解决数据冲突),慢速 HTM 和 STM 也可以同时运行(因为慢速 HTM 像 STM 一样获取锁).但是,由于快速 HTM 支持不插桩读,因此不能在没有额外机制能够确定 STM 是否使存储器处于不一致状态的情况下与 STM 同时运行.因此,快速 HTM 上的每个事务 T 都需要订阅一个计数器,该计数器记录了 STM 路径上处于写回阶段的事务数.如果在 T 执行期间计数器非零,那么 T 可能看到不满足一致性要求的状态,所以需要中止 T 的执行;如果 T 的中止次数过多,T 将切换到 STM 路径以保证得到处理.

诸如 HANA(SAP),Hekaton(微软),TimesTen(Oracle),MemSQL 和 VoltDB 等内存数据库(in-memory databases, IMDB)经常应用在单节点配置的数据分析系统中,以及响应时间至关重要的应用程序中,如电信网络和实时金融服务.在此类实现中,内存数据库使用同步机制的一部分来协调进程对内部数据结构的访问.常见的同步机制包括两阶段锁(2-phase locking, 2PL):乐观并发控制(optimistic concurrency control, OCC)和多版本并发控制(multi-version concurrency control, MVCC).最近的研究表明,SAP HANA 可以从当前的 HTM 实现[4]中获得显著的性能提升,并且声称通过 NVM 的实现[5]提升了容错能力.因此,本文的一个有价值的应用方向就是将 PHyTM 作为内存数据库新的同步机制,利用 HTM 的能力减少传统方法的同步开销,同时增加对 NVM 的支持.

为了证明这种方法的潜力,我们在支持 HTM 的英特尔系统上通过模拟 NVM 进行了实验.本文使用 Yahoo!的云服务基准测试(Yahoo! cloud serving benchmark,YCSB)和 TPC-C 进行基准测试,比较了使用 4 种不同的同步机制实现的简单内存数据库的性能:2PL,OCC,PHTM 和 PHyTM.结果表明,PHyTM 效率高并且可扩展,往往明显优于其他机制.

我们预测非易失性内存将成为标准,并可以适应通用应用程序.这些从未使用 ACID 事务的程序将必须在 NVM 中保持一致性,以保持系统的可恢复性.那时,PHTM 和 PHyTM 可以用来维护数据库、数据结构和数据库上下文.

作为正在进行的和将来的工作,我们讨论了分离事务执行(split transactions execution, STE)算法,本文在数据库特定的同步协议内使用 PHTM;本文也展示了在多数事务对 PHTM 而言过大的情况下,如何通过 STE 帮助 PHTM 获得更简单和更快速的解决方案.

1 模　　型

本文考虑使用了 HTM 和 NVM 的多进程异步共享内存系统.

1.1 内　存

本文所考虑的内存体系结构在最底层由全 NVM 内存作为主存(如果系统中也包含了 DRAM,逻辑内存空间一般被分割成持久化和非持久化地址空间,为了简化问题,本文仅考虑无 DRAM 的情况).不失一般性,将缓存行定义为数据的最小粒度.最底层之上是缓存,包含了主存中缓存行的副本.缓存一致性协议确保了无论有多少个缓存副本各处理器都能获得一致的主存数据.最高层为寄存器,即每个处理器内部用于暂存数据的特殊存储.

一般而言,在内存系统低层中进行的操作比高层慢几个数量级.NVM 的写操作比 DRAM 慢,读操作速度则至少持平.

数据从缓存异步刷新到 NVM,这一过程程序员无法感知.数据也可以通过硬件原语 FLUSH,参数是内存地址,显式刷新到 NVM.FLUSH($addr$)操作利用缓存一致性协议将最新的缓存行副本刷新到主存地址 $addr$.

1.2 故　障

本文将考虑系统掉电后所有易失性内存的内容丢失的情况,而不考虑其他类型的故障,如进程崩溃或拜占庭式故障.所有的缓存和寄存器都是易失性存储,掉电后仅 NVM 仍保存着数据.

掉电后系统通过单个恢复进程执行一系列特殊的恢复流程.这个恢复流程在其他进程恢复运行前修复数据.由于恢复进程单独运行,因此具有相当大的自由度,可以进行危险操作,例如强制释放其他进程在掉电前已持有的锁.

1.3 硬件事务内存

考虑英特尔 HTM 的实现,进程 p 通过调用 *xbegin* 启动事务,若处理器进入事务执行模式则 *xbegin* 返回 OK,否则返回退出原因.在事务模式下,每一次 p 对某一地址进行读或写操作时,将该地址加入到该事务的读地址集合(read-set)或写地址集合(write-set).该事务的读地址集合和写地址集合的并集称为数据地址集合(data-set).若一个事务的数据地址集合与其他并发事务的写地址集合相交则存在数据冲突.HTM 系统将中止相互冲突的至少一个事务.进程 p 也可以通过调用 *xabort* 主动退出当前事务的执行.

事务也会因其他原因而退出执行,例如系统调用、中断,缺页中断或内部缓冲区溢出.特别地,由于事务存在可访问内存地址数的限制,超过这一限制将因容量超限而退出.容量超限无法预测,而且硬件事务由于数据地址集合持续增长而更容易出现容量超限问题.为避免容量超限而退出事务执行,事务的数据地址集合必须最少满足如下条件:适合 L1 缓存的大小、避免缓存的关联冲突、避免将事务载入的缓存行换出.另外,当 2 个超线程在同一个核上运行时,该核的本地缓存由这 2 个超线程共享,这使得每一个线程的 L1 缓存容量减半,容量超限退出问题也更严重.

无论何时进程 p 的事务 T 退出时,p 停止该事务,控制流返回到调用 *xbegin* 之前(事务启动前).因此 p 下一步是调用 *xbegin*,这一次调用将返回退出的原因(如冲突、容量超限).这样 *xbegin* 的典型调用模式为"if *xbegin*() = OK then {transaction body} else {abort handler}".

1.4 增加对 NVM 的支持

需要指出硬件事务并不直接修改主存,相反,所有的事务写是在缓存中实施的,所有相关的缓存行在事务提交后一个接一个地刷入主存.缓存一致性协议确保了受影响的缓存行将在事务提交后原子性地刷入主存.但是若在刷入主存过程中发生了掉电故障,缓存的信息可能会丧失原子性.

避免这种问题的方法之一就是让每个事务在提交前以日志记录写操作并刷新到 NVM 中,以便在恢复过程中有足够的信息完成掉电前的任何事务.当然,这种方法要求已实现事务处理过程中的 FLUSH(不中止事务).在接下来的 Avni 等人的研究中[3],我们假定透明刷新(TFLUSH)不会中止事务.此外,既然日志在事务提交前已刷新到 NVM,恢复进程一定能够检查掉电前事务是否已提交.同样地,类似文献[3],我们假定 *xend* 通过扩展指令在 NVM 中设置一个标记位,可以在事务提交到缓存时原子性地设置该比特位并刷新到 NVM 中.在事务提交时原子性地将标记位刷新到 NVM 是对当前 HTM 协议的最小化修改.

2 持久化 HTM

Herlihy 和 Moss 定义了硬件事务内存(HTM)[6],用于充分利用硬件缓存一致性机制在多核芯片的可缓存共享内存中执行原子事务.其基本思想是每一个事务都在当前核的本地 L1 缓存中隔离."原子性"是从数据库系统中借用的概念[7],但是,数据库的事务与 HTM 事务不同,数据库的事务是持久化的,例如当一个事务成功提交后也将备份到持久化存储中.

由于 HTM 发展缓慢,大量的相关研究集中在 STM 以获得低开销和无需硬件辅助的可扩展同步事务内存.英特尔公司最早的 HTM 实现在 2013 年上市,同时 STM 也已整合进 GCC 编译器中[8-9].

内存技术的最新进展(如 PCM,STT-RAM 和忆阻器)展示了 NVM 设备可以如 DRAM 一样高速进行字节寻址,比 DRAM 节能,还具有非易失性,并且和 HDD 一样便宜.本节将提出一种通过利用 NVM 进行持久化存储而非(补充)DRAM 的 HTM 事务的方法,同时还能够保持易失性缓存结构.

2.1 相关研究

Coburn 等人[10]提出了一种可与 NVM 正确工作的软件事务内存——NV-Heaps.其基本思想遵循 DSTM[11],即事务对象存储在 NVM 中,可以在写入时打开,然后 STM 事务 T 将它复制到撤销日志(undo log)中并锁定.T 为每个事务维护一个易失性读日志和一个非易失性撤消日志.如果发生系统故障,则中止 T 并使用持久化的撤消日志来复原 T 所做的改变.

NV-Heaps 是基于对象的,而同时发表的源自 TinySTM[12]的 Mnemosyne STM[13]是基于字的,但是这些算法背后的思想都是非易失性撤消日志和易失性读日志,这是相同的.

虽然正确并且可行,但由于维护记录的开销和锁的序列化,基于软件的方法性能较弱.因此,虽然有些数据库实现使用了 HTM 进行同步[14-15],但是,这些数据库仍然使用硬盘进行持久化.

PMFS[16]使用 NVM 作为文件系统存储,文献[17]使用 NVM 在 OLTP 数据库中进行持久化. PMFS 确实使用了 HTM,但仅用于管理文件系统元数据这一特定目的.

2.2 术 语

非易失性内存技术的已日趋成熟,这必将改变事务系统的构建方式. NVM 设备如 DRAM 一样快,同样可以进行字节寻址,比 DRAM 更加节能,还具有非易失性,并且和 HDD 一样便宜. 因此,非易失性内存将消除传统的多层内存体系的分层,这种分层是 ACID 事务持久性的保证. 在主存中维持失效状态可导致系统故障后无法恢复. 因此,设计故障恢复(重启)时的持久化方案需要一种全新设计的、并且考虑周全的方法,这将从 NVM 中受益. 后续我们将使用如下术语.

1) HTM. 原子性地提交事务的同步机制,并保持隔离性. 一旦 HTM 事务 T 提交,所有新修改的数据都在易失性缓存中.

2) HTM 事务 T_K. 由处理器的核心 P_k 执行的事务.

3) NVM. 非易失性,可字节寻址、可写的内存.

4) 重启. 重新启动的任务是使数据处于一致性状态,消除未提交事务的影响,并恢复丢失的已提交事务.

这里考察的硬件模型包括无限的 NVM,多核处理器并且无硬盘. 所有的 NVM 都是可缓存的,缓存是易失的并且是一致的. 该系统包括有限容量的 DRAM. 最后,我们使用术语持久化硬件事务内存(persistent HTM, PHTM)来指代现有 HTM 实现的概念,并纳入 NVM 所需的最少硬件和软件调整. PHTM 系统包括软件和硬件.

2.3 问题定义

随着现代硬件中处理器核心数越来越多,NVM 就会越来越难以满足持久化要求. 一方面,由于数据在 NVM 中,因此不需要为其另外分配一个持久化存储器,这减少了持久化的开销. 但另一方面,向某一地址写入时,新值必须以新的、一致的并且持久化的状态原子性地对外暴露. 保证这种原子性的一个方案是通过锁. 随着核心数量的不断增加,锁必将成为瓶颈. 实现免锁原子性的一种方法是 HTM,但 HTM 无法访问物理内存. 这样,当前面临的主要问题就是填补 HTM 与 NVM 之间的差距,并允许应用程序在 NVM 中保持一致和持久的状态,而且不加锁,不复制数据.

2.4 数据存储流程

由 HTM 事务 T 写入的数据项 x(可寻址的字)的状态如图 1 所示. 请注意,x 可能缓存在易失性缓存中,或者只驻留在 NVM 中(就像任何其他可寻址的字一样):

1) Private/Shared. Private 表示 x 只在一个线程的 L1 缓存中,对其他线程不可见. 当 x 是 Shared 时,缓存一致性使其新值对其他线程可见.

2) Persistent/Volatile. Persistent 意味着 x 最后写入的值在 NVM 中,Volatile 表示 x 的新值在缓存中,并且掉电时会消失.

3) Logged/Clear. 当 x 是 Logged 时,重启将从非易失性日志中恢复 x. 如果 x 是 Clear,重启将不会对 x 执行任何操作,因为不存在相关的记录. 这可能在事务提交或中止之后发生.

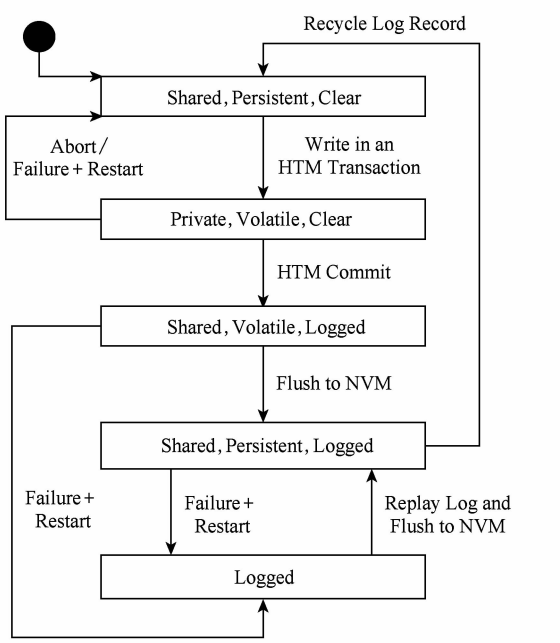

Fig. 1 State machine for a persistent transactional variable

图 1 持久化事务的状态机

尽管图 1 说明了 PHTM 事务中单次写入的状态机,Logged 状态应该是针对整个事务的. 也就是说,将单个事务的所有写入从 Clear 转变为 Logged 需要唯一的持久化写. 在 PHTM 提交中,所有写入都由 HTM 暴露,并同时被记录下来. 每次写入都必须生成一个持久化的日志记录,但是直到成功提交之前,写入操作不会被重启进程重演.

当 HTM 事务 T_k 写入变量 x 时,x 在 P_k 的 L1 缓存中被标记为事务性的,并且是私有的(Private),即仅存储在 P_k 的缓存中. 它是易失(Volatile)的,

因为它只在缓存中,而且是未记录(Clear)的,即未记录到日志中,因为事务还没有被提交.在中止或掉电故障时,x 的易失(Volatile)私有(Private)值将被丢弃,并将恢复到先前共享(Shared)和持久(Persistent)值.

在 PHTM 提交中,x 的状态改变了 2 次.首先变成共享的(Shared),即可见的,同时也被记录(Logged)下来.这 2 次状态改变都必须原子性地提交成功.成功提交后,PHTM 将 x 的新值透明刷新到 NVM,并清除 x.如果系统发生故障并在 x 记录时重新启动,则恢复过程使用 x 的日志记录来写入提交的 x 值,然后清除 x.

NVM 中的日志不是典型的顺序日志,相反,它仅保留无序的日志记录,这些日志记录仅用于正在运行的或已提交的事务,而且尚未回收其日志记录.

有关 PHTM 实现的详细信息,包括衍生硬件以及正确性证明,请参阅 Avni 等人的文献[3].

2.5 评 估

本节使用红黑树的数据结构和复合基准程序测试 PHTM 的性能.该复合基准程序测试 PHTM 的开销,以及 PHTM 和 HTM 大小限制的相关性.

测试在英特尔 core i7-4770 3.4 GHz Haswell CPU 上进行测试,共有 4 个核心,每核 2 个超线程,每个核心拥有本地 L1 和 L2 缓存,大小分别为 32 KB 和 256 KB,同时所有核心共享 8 MB L3 缓存.

2.5.1 硬件仿真

英特尔的 Haswell 处理器具备用于实验的 HTM 特性.但是,NVM 和 PHTM 仍未能在硬件上实现,所以需要仿真以进行评估.我们通过将电源开启并仅将易失性区域(如日志标记)置零来模拟 PHTM 中掉电故障的影响.为了模拟 tx_end_log,提交记录在事务中写入.由于这是事务的一部分,所以 HTM 本身使提交记录在成功提交的同时可见.在 TF 仿真中,根据预期的 NVM 性能,通过加入 100 ns 的延迟来模拟 NVM 访问时间.本文并未仿真互连流量,因为这个流量与解决方案中的互连流量(即持久化 STM)相同.

2.5.2 对比算法

PHTM 与标准 HTM、模拟持久化的 STM(persistent software transactional memory, PSTM)、无持久化的标准 STM 进行了比较.STM 来自 GCC[8-9],并使用最低优化级别.这是为了使所有 4 种算法能够进行公平的比较,避免了用以减少访问次数的编译器优化.

与 PHTM 一样,PSTM 在持久化存储中实现重演日志.在 PSTM 中写入的开销是提交之前的日志条目的一次刷新和提交之后的一次刷新,因此它与 PHTM 相当.PSTM 基于 Mnemosyne[13],但有少量修改.本文选择实现重演日志和本地数据更新,以避免写惩罚后巨大的读操作.PHTM 相对于 STM 的优势在于以处理器的速度加载.为了公平比较,PSTM 应该尽可能快地加载以挑战 PHTM.

PSTM 的缺点源自 STM,即与指令插桩、锁和版本相关的开销.PHTM 的缺点源自 HTM,即有限的事务大小.

2.5.3 性能基准

本文将在复合数组基准测试中对无争用条件下的 PHTM 进行性能测试,然后在红黑树上测试争用条件下的性能.检查的算法包括 HTM,PHTM,STM 和 PSTM.在图 2 中,增加了 HTM-CAP 和 PHTM-CAP 行来统计 HTM 容量超限中止次数,并增加了 HTM-CON 以及 HTM-CON 和 PHTM-CON 行用以表示一些图中因冲突而终止的次数.中止次数通过每秒中止的操作数进行统计.因冲突而中止并在加全局锁之前需重新尝试 20 次,但容量超限中止不会重试并立即加锁,因为重试成功的可能性非常小.

2.5.3.1 数组负载

在这个负载中,所有的事务都具有相同的访问次数以使其执行时间相当.该测试访问一组连续的内存地址,所以缓存中无碎片.每个待访问地址都在单独的缓存行中.

1) 只读(read-only)

首先观察 PHTM 在只读工作负载上的性能.这对于 PHTM 而言是最好的情况,因为 PHTM 以处理器的速度加载.在图 2(a)中,每个事务循环地执行 512 次加载指令到不同数量的连续缓存行.可以看到,只要避免 HTM 容量限制,PHTM 和 HTM 的性能相同,并且比 STM 和 PSTM 高一个数量级.图 2(a)中的所有测试都在所有 8 个硬件线程上执行.由于存在超线程,每个线程的缓存大小是其核心缓存大小的一半,即 16 KB 或 256 个缓存行(每个缓存行 64 B),所以当访问集大小为 512 时,所有的 HTM 和 PHTM 事务都将触及容量上限并中止退出.在这一点上,HTM 和 PHTM 的性能等同于 STM,因为它使用了撤回.如预期的那样,STM 和 PSTM 表现相当.

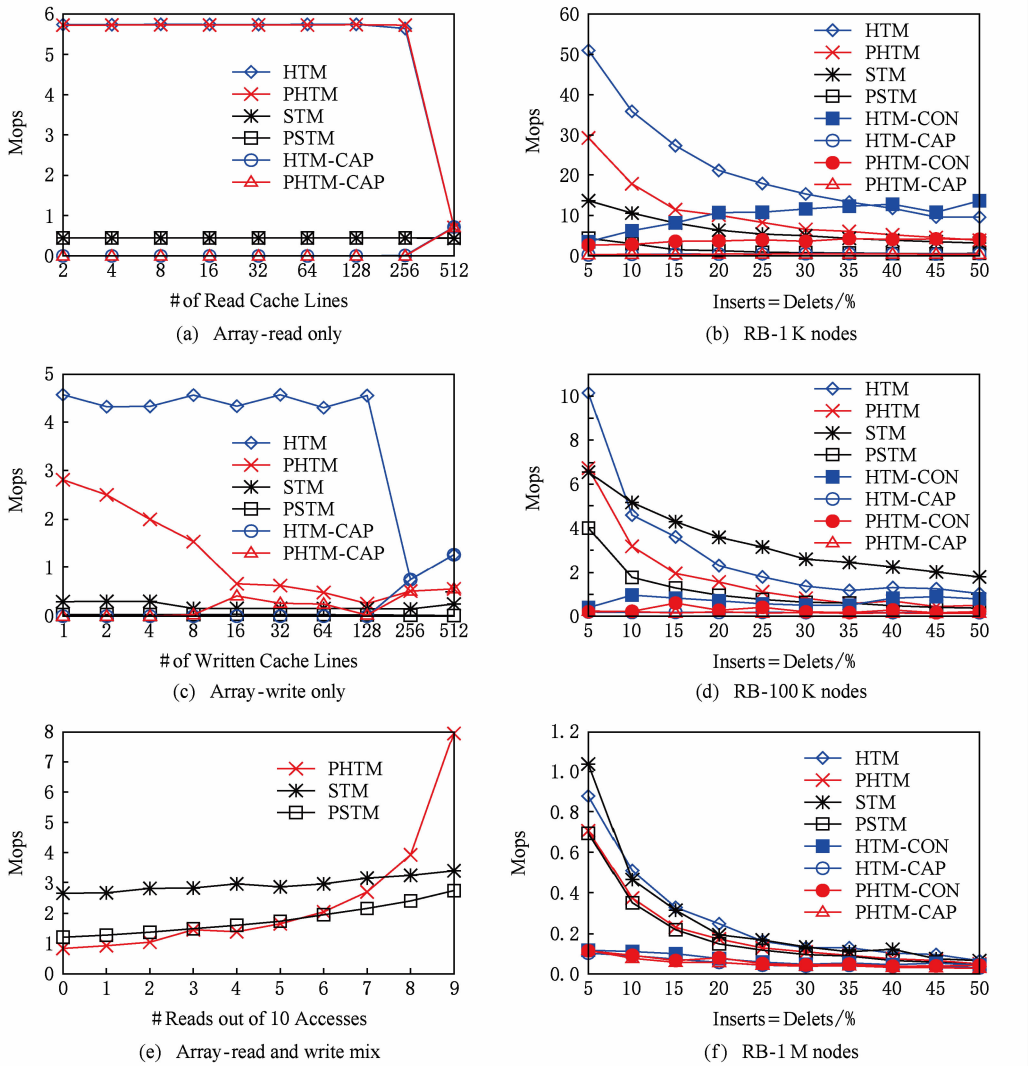

Fig. 2 Synthetic array and a red-black tree benchmarks

图 2 数组和红黑树复合性能结果

2) 只写(write-only)

这个测试检验 PHTM 的写入性能. 图 2(c)中的性能结果与图 2(a)中的性能结果非常接近. 每个事务循环地执行 512 次存储指令到不同数量的连续缓存行.

可以看到,HTM 性能未受到待访问缓存行数量增加的影响,但 PHTM 接近 PSTM 的性能,因为刷新到 NVM 的速度决定了性能. 必须强调的是,即使多次写入缓存行,PHTM 和 PSTM 也只能在事务中刷新一次缓存行. 因此,如果事务只访问一个缓存行,则它将 512 次写入相同的缓存行,但仅刷新 2 次,一次在提交之后刷新数据,另一次在提交之前刷新日志. 如果将 TF 置为非阻塞可以减小开销.

当 HTM 和 PHTM 事务开始接近容量上限时,它们会加全局锁,虽大大降低性能但可防止事务继续扩大. PHTM 在标准 HTM 之前达到容量限制. 这是因为 PHTM 的每次写入都要写入一个日志条目,所以它可以访问更多的内存. 此外,日志条目不在连续内存中,所以即使在缓存已满之前,它们也可能违反缓存关联性. 请注意,在一个真正的实现中,非事务性的存储指令可以被用于日志,并且避免增加事务占用的空间.

3) 读写混合(read-write mix)

在不存在容量超限中止问题的情况下,即在小事务中,该测试检查事务性加载指令与事务性存储指令的数量如何影响 PHTM,以及与 STM 和 PSTM 相比的性能.

在图 2(e)中,每个事务对 10 个相互独立的缓存行访问 10 次,并执行不同数量的写操作. HTM 并未在图 2 中展示,因为它以大致相同的速度进行读写. 图 2(e)表明直到只读部分达到 70%,STM 都比 PHTM 快. 但是当只读部分达到 90%时,PHTM

已经是 STM 性能的 2 倍.如图 2(a)所示,当只读部分达到 100%时,PHTM 比 STM 快 12 倍.

2.5.3.2 红黑树(RB-Tree)负载

为了评估争用条件下的 PHTM 性能,本文使用了一种红黑树进行基准测试,所有事务都使用随机键值访问树进行插入、删除或查找操作.所执行的工作负载运行在 8 个核上,具有固定的键值范围大小,更新的比例范围从 10%到 100%.每棵树从半满开始,插入的数量等于删除的数量以维持树的大小.

第 1 组测试是在一个小的 1K 个节点的树上执行的.如图 2(b)所示,HTM 和 PHTM 没有因容量超限中止,出现了冲突退出,但可扩展性与 STM 相当.在这个测试中,PHTM 比 PSTM 快大约 6 倍.图 2(d)中的第 2 组测试是在 1M 个节点的树上进行的,可以看到出现了容量超限中止,但仍较少.因此,PHTM 只比 PSTM 快 40%.尽管冲突中止率很高,但可扩展性仍然相同,这表明中止率与 STM 接近.为了进一步增加容量带来的挑战,我们原子性地在 2 个 1M 个节点的树上执行相同操作的事务.如图 2(f)所示,容量超限中止数上升,PHTM 的性能下降到和 PSTM 相当.

正如预期的那样,容量限制是 HTM 最大的问题,因为它会迫使事务序列化,而 PHTM 性能的最大障碍是它从 HTM 继承的容量超限中止.在每个执行的测试中,PHTM 在所有争用水平上与 HTM 和来自 STM 的 PSTM 保持恒定的差异.不同之处在于工作负载中的写入部分.树越小,遍历它的时间就越少,所以写入部分相对增加,并且持久化算法的开销相应增加.

3 持久化混合 HTM

在 PHyTM 中,事务可以以如下 3 种路径执行:快速 HTM、慢速 HTM 和 STM.每种路径都提供了一个操作集用于启动和提交事务,以及读/写内存位置.这些操作并非直接由用户代码调用,相反,用户简单调用编译器提供的操作以启动和提交事务,编译器将通过我们在操作集中提供的操作编译用户代码,自动选择合适的路径执行事务.

3.1 STM 路径

本文所述 STM 算法通过遭遇时序(encounter-time order, ETO)实现了两阶段锁.每一个事务首先锁住所有它可能访问的地址(以首次遭遇的顺序),随后可以在这些地址上进行所需操作,最后释放所有的锁.为避免死锁而使用了 try-lock,当无法获取到锁时立刻返回 false,而非一般的阻塞.当一个进程获取 try-lock 锁失败时,将释放其持有的所有锁并再次尝试.存在一种罕见的情况:若一个进程获取锁时持续失败,该进程将锁住整个 STM 路径.作为 try-lock 的备选方案,可以使用标准的死锁检查算法,但这样会降低效率,尤其是在非常高竞争的场景下.

2PL 和 ETO 的应用使得正确性的条件——不透明性[18]非常容易得以证明,不透明性可直观表述为进程无法观察到事务的部分结果.考虑一个要写入一些地址的事务 T,2PL 和 ETO 要求 T 在访问地址前锁住所有这些地址,并持有锁直到完成写入.另外,其他任何尝试读取这些地址的事务 T' 均无法在 T 解锁前操作.因此 T' 无法看到 T 的任何写入,直到 T 的所有操作均已经完成.

STM 将锁住事务欲访问的所有地址,完成读操作并将写操作记录到日志中,随后进入写回阶段.在写回阶段,事务将日志刷入 NVM,完成所有的写入操作,随后将写入结果刷入 NVM.这种实现机制确保了日志刷入 NVM 的原子性,这使得恢复进程可以在提交时访问日志(否则,已提交的事务可能丢失,未提交的事务也可能由恢复进程重演).

STM 路径提供了 4 种操作:用于启动事务的 *STMBegin*、用于替换标准内存读的 *STMRead*、用于替换标准写的 *STMWrite*、用于提交事务的 *STMFinalize*.

STMBegin 操作作为读者获取 *stmLock*(若该事务尝试次数未超限)或者作为写者获取(如果超限).作为写者获取 *stmLock* 将阻止其他事务在 STM 路径上运行.

STMRead 操作首先调用 *GetLockAddr*(*addr*)来确定 locks 中的哪一个保护 *addr*;然后调用 *TryReadLock*(*lock*)以获取读锁,读取该地址,并将该地址(*addr*)保存在它的读地址集合中.*STMWrite* 操作也需要首先通过调用 *GetLockAddr*(*addr*)来确定合适的 *lock*,随后调用 *TryWriteLock* 以获取写锁.这样做有 2 个目的:这个锁授予待写入地址的排他访问权限,并且独占地在写日志(write-log)中存储该地址.如果执行 *TryWriteLock* 的进程当前作为读者持有锁,并且没有其他读者,则 *TryWriteLock* 将读锁升级为写锁.接下来,*STMWrite* 将 *addr* 和 *val* 添加到其写日志条目中.请注意,*STMWrite* 不会显式刷新这些修改,但可能异步地写入 NVM.如果

STMRead 或 *STMWrite* 无法获取锁,则事务中止,释放所有锁并重试事务.

进程需要调用 *STMFinalize* 以提交 STM 事务. *STMFinalize* 首先将写日志条目刷新到 NVM,然后通过调用 *numSTMWriteback* 上的 *FetchAndIncrement* 表示事务已进入写回阶段(将会在 3.3 节中看到如何运用 *numSTMWriteback*). 然后设置并刷新日志条目中的 logged 位,这表示日志条目已准备好,可以在发生掉电故障时由恢复进程重演. 当 logged 位刷入 NVM 时,事务为 committed 状态. 接下来,*STMFinalize* 调用 *ReplayLogEntry* 函数来重演事务的日志条目,执行所有的写操作并将其刷新到 NVM. *ReplayLogEntry* 也会清除并刷新 logged 位以表明该日志条目不再需要重演. 在事务的所有写操作完成并刷新后,*STMFinalize* 将取 *numSTMWriteback* 并进行减操作,以表明该事务不再处于写回阶段. 最后,*STMFinalize* 解锁所有的锁,并准备其日志条目以供进程的下一个事务重用.

3.2 慢速 HTM 路径

与 STM 路径一样,慢速 HTM 需要在所有待写入地址上获取锁,随后在日志中记录所有的写,这将保证无法对同一地址重复记录写. 但是慢速 HTM 和 STM 有 2 处不同:1)慢速 HTM 实际上在记录到日志之后立刻执行写操作(不需要等待日志重演). 这在 HTM 中可以实现,因为所有写操作都保留在处理器的私有缓存中直到事务提交. 2)慢速 HTM 在读取地址时不会获取任何锁,相反,它只是简单地读取每个地址的锁的状态. 如果当前已由写者锁定(写锁),则事务将中止. 读取锁状态意味着 HTM 事务将订阅这把锁. 这样,如果在事务首次检查其状态时为未锁定状态,但在事务提交前的某个时间点被其他进程锁定,则事务将中止.

慢速 HTM 订阅它所读取的地址的锁,并锁定它写入的地址,记录和执行其写入,因为它锁定了要写入的每个地址. 当所有的写入操作完成后,将其日志刷新到 NVM 并通过 *xend* 原子性执行如下操作:提交事务并将其日志标记为已完成,以便恢复进程在发生掉电故障时进行重演. 最后,慢速 HTM 重演其日志条目,将其所有写入刷新到 NVM,然后清除其日志条目. 如果一个事务在慢速 HTM 上失败了很多次,它将切换到 STM 路径.

SlowHTMBegin 操作调用 *xbegin* 来启动硬件事务,然后判断是否以事务模式执行(即 *xbegin* 返回 OK). 如果不是,那么跳转到先前的慢速 HTM 事务中止时的代码行. 所以,*SlowHTMBegin* 检查是否已经到达尝试次数上限. 如果已到达尝试次数上限,该事务就切换到 STM 路径;否则,*SlowHTMBegin* 再次尝试在硬件中执行.

SlowHTMRead 操作读取待读取地址的锁的状态,若有其他进程作为写入者获得锁中止;否则,*SlowHTMRead* 只是简单地读取地址并返回结果. *SlowHTMWrite* 操作尝试给一个地址加写锁,如果失败则退出. 该锁授予对正在写入的地址的独占访问权,并且独占许可将该地址存储在写日志中. 如果 *SlowHTMWrite* 获取该锁,则它将要写入的地址和值添加到事务的写日志条目中. 最后执行实际的写入.

为了在慢速 HTM 中提交一个事务,进程还需要调用 *SlowHTMFinalize*. 这会将写日志条目刷新到 NVM 中,接下来还需要调用 *xend*. 调用 *xend* 将同时提交事务,并设置和刷新日志条目中的记录位(表示如果发生电源故障,日志条目已准备好由恢复进程重演). 在此之后,*SlowHTMFinalize* 调用 *ReplayLogEntry* 重演自己的日志条目,将写操作刷新到 NVM 中. *ReplayLogEntry* 也清除并刷新记录下的比特位,表明该日志条目不再需要重演(与在 *STMFinalize* 中调用 *ReplayLogEntry* 不同,这个 *ReplayLogEntry* 的调用不需要执行事务的写操作,因为它们已经作为硬件事务的一部分执行了). 最后,*SlowHTMFinalize* 解锁所有的锁,并准备其日志条目以供进程的下一个事务重用.

3.3 快速 HTM 路径

每个事务都以快速 HTM 开始. 快速 HTM 事务中的写入和提交与慢速 HTM 事务中的写入和提交相同. 快速 HTM 事务的读取效率更高,因为它们不需要订阅锁来保证事务处理看到一致的状态(即它读取的地址包含在事务处理过程中某些时刻看到的值),有 2 个原因:首先,HTM 系统保证慢速 HTM 上的事务不会导致快速 HTM 上的事务看到不一致的状态(反之亦然);其次,每个快速 HTM 事务首先验证 *numSTMWriteback* 是否为零,若非零则中止. 正如我们在 3.1 节看到的那样,每当一个 STM 事务开始写回,*numSTMWriteback* 就会增加,并且每当一个 STM 事务完成写回时递减. 因此,如果 T 在写回阶段与任何 STM 事务并行运行,它将中止. 因此,*FastHTMRead* 被实现为一个简单的读取,没有额外的同步.

最后,我们描述 *FastHTMBegin*. *FastHTMBegin* 首先调用 *xbegin*,然后验证是否以事务模式执行的. 若在事务模式下执行,则 *FastHTMBegin* 验证 *numSTMWriteback* 是否为零,若非零则中止,然后返回. 若非事务模式,则跳转到之前的快速 HTM 事务中止时的代码行. 所以,*FastHTMBegin* 检查尝试次数是否已经到了上限. 若已到达上限,则快速 HTM 切换到慢速 HTM;否则,*FastHTMBegin* 再次尝试在快速 HTM 上执行事务.

3.4 实验结果分析

在本节中,我们将研究如何使用 PHyTM 降低各种工作负载下简单内存数据库的同步成本.

1) 工作负载

我们实现了一个非常简单的内存数据库(very simple IMDB, VSDB),并用它来运行 Yahoo!云服务基准测试(Yahoo! cloud serving benchmark, YCSB)的一个子集. 本文还使用 DBx1000(来自文献[19]的内存数据库实现)研究了 TPC-C 基准. 结果及其分析可以在文献[20]中找到. 我们简单的 YCSB 基准测试证明了 PHyTM 的低同步成本,而 TPC-C 基准测试则以更复杂的事务来说明其性能.

2) 同步方法

我们使用不同的同步方法修改每个内存数据库. 在内存数据库中,我们增加了对 PHyTM 和 PHTM 的支持. 在 VSDB 中,我们实现了一个简单的 2PL 方案,按照遭遇顺序对高速缓存行进行细粒度的锁定. 由于我们使用的 YCSB 工作负载不会导致死锁,所以并未实现 2PL 的死锁检测. DBx1000 已经将 2PL 和乐观并发控制(OCC)作为同步方法. 2PL 实现在行上执行细粒度锁定,并结合死锁检测. 已证明该 2PL 和 OCC 实现是可扩展的,在模拟器中模拟了超过一千个处理器[19].

3) 系统

我们使用带有 4 个核心的 Intel i7-4770 3.4 GHz 处理器,每个处理器有 2 个超线程. 每个核心都有专用的 L1 和 L2 缓存,大小分别为 32 KB 和 256 KB. 还有一个由所有核心共享的 8 MB 三级缓存. 我们使用由硬件提供的 HTM 并模拟 NVM. 实验中线程是固定的,每个逻辑处理器上运行一个线程.

4) 仿真

我们使用文献[3]中采用的方法模拟对 NVM 的支持. 原子性分配和对 logged 位的刷新是通过避免在 HTM 提交和设置该位之间的任何模拟电源故障来模拟的. 由于预计 NVM 的写入速度比写入 DRAM 要慢,因此我们插入了延迟以模拟写入 NVM 的所有同步方法.

5) YCSB

在我们所有的 YCSB 实验中,使用了一个有 2 000 万条记录和 1 个单主键列的表. 执行 4 种类型的事务:短距离查询(short range queries, SRQ)、长距离查询(long range queries, LRQ)、短距离更新(short range updates, SRU)和长距离更新(long range updates, LRU). 短(或长)事务查找表中的 16(或 256)个随机的键. 查(或更新)事务选择一个列(或写入列). 这些事务只是简单地读取或写入列的内容,而不执行任何额外的计算. 例如,查询事务的 SQL 代码可能是“SELECT name FROM customers WHERE id IN(1350,2107,…,571)”. 更新事务的 SQL 代码可能是“UPDATE customers SET orders = 7 WHERE id = 1350; UPDATE customers SET orders=4 WHERE id=2107; …”. 所有事务都是相互独立的. 也就是说,事务的行为不依赖于前一个事务的结果. 请注意,由于长事务访问相当多的行,有可能导致容量超限而退出.

6) 结果

结果如图 3 所示. 一般而言,在终止退出较少时 PHyTM 的行为与 PHTM 相似. 这是因为只要 STM 路径上没有事务处理,2 种算法都没有读操作的开销,写入 NVM 的开销将导致写操作较多的工作负载存在性能差异. 在写操作较多的工作负载中,2PL 的性能同样由写入 NVM 的开销决定. 对于基准测试中的所有算法而言,NVM 的写入开销是相同的,因此它们在只写测试中表现出相似的性能,如图 3(c)所示. 然而,图 3(b)显示 PHTM 和 PHyTM 的读速度比 2PL 快一个数量级.

图 3(d)显示执行 50% SRQ 和 50% SRU 的 YCSB 工作负载. 在这种工作负载中,写入 NVM 的开销比任何算法相关的性能差异更显著,所以 2PL 的性能与 PHTM 和 PHyTM 的性能相当接近.

HTM 最大的缺点是事务规模受容量超限中止的限制. PHTM 包含有一个 STM 回滚路径以允许完成大型事务,但是该 STM 路径需要进行全局锁定,所以 STM 路径上的事务被序列化. 这是严重的性能瓶颈,特别是当 HTM 支持的系统中并发度越来越高时(支持数百个并发线程).

为了研究部分事务在硬件中执行失败,接下来必须在软件路径中执行时会发生什么,我们增加了一个工作负载,其中包含有不太可能在硬件中提交

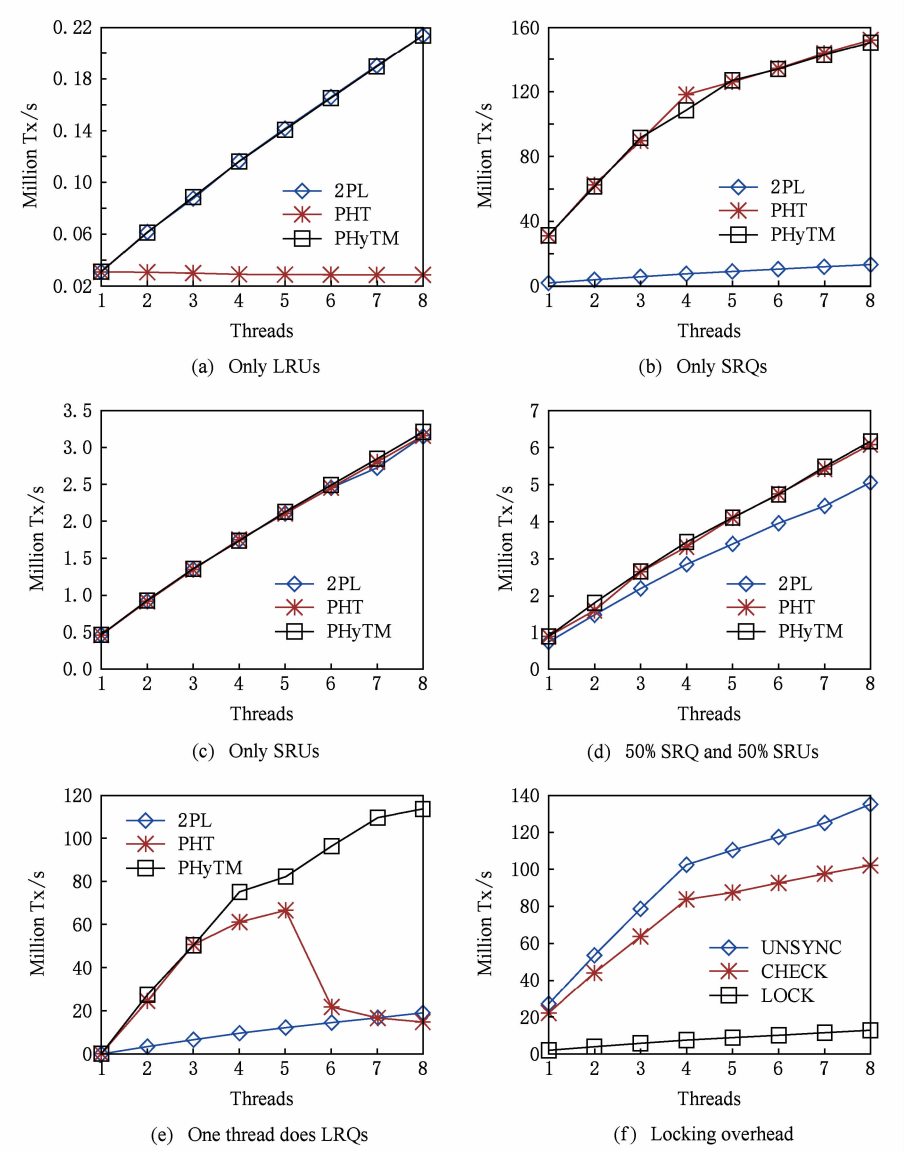

Fig. 3 YCSB workloads for different transaction mixes

图 3 YCSB 验证结果

的大事务. 当所有事务都是 LRU 时,如图 3(a),我们可以看到 PHTM 根本不可扩展,而 PHyTM 与 2PL 一样可扩展. PHyTM 可扩展是因为 STM 事务可以与其他硬件事务同时运行. PHyTM 在落入软件路径前乐观地尝试硬件事务所带来的开销并未使得其性能低于 2PL 的性能(从不中止退出),即使在这种包含众多异常中止的工作负载下也是如此. 我们认为这是因为 PHyTM 在提交之前避免了对 NVM 执行许多昂贵的写入/刷新操作. 这样,中止退出的事务避免了这种开销.

在完全由 LRU 组成的工作负载中,PHyTM 和 2PL 实现的吞吐量大致相同,并且比 PHTM 高一个数量级. 在完全由 SRQ 组成的工作负载中,PHTM 和 PHyTM 则实现了大致相同的吞吐量,并且比 2PL 高一个数量级.

当少量线程在 STM 路径上运行事务,而大多数线程在 HTM 中成功提交事务时,PHyTM 比 PHTM 和 2PL 的优势变得明显. 如图 3(e)所示,其中一个线程正在执行 LRQ(不太可能在硬件路径中成功执行,通常需落入软件路径运行),其他线程执行能够在硬件路径上成功提交 SRQ. 在这种情况下,PHyTM 比具有 8 个线程的竞争对手快一个数量级. PHTM 的 STM 路径执行读取操作时没有额外开销,因此其在处理次数较少时比 2PL(其必须获取到锁)要快得多. 但是,由于 PHTM 的 STM 路径需要获取到全局锁,因此无法扩展. 2PL 虽然可以通过将 PHTM 与 8 个并发线程联系起来而在此类工作负载中扩展,但是受制于获取锁的高昂成本. 只要

没有 STM 事务在其写回阶段,PHyTM 事务就可以在快速 HTM 上运行,这样它们的读取操作没有额外开销.此外,即使存在处于写回阶段的 STM 事务,PHyTM 事务也可以在慢速 HTM 上运行,在这种情况下,读操作可以简单地读取锁的状态而不需要获取到锁.

为了说明 PHyTM 减少了多少 HTM 的同步锁成本,我们用 PHyTM 的 3 种不同版本运行了图 3(b)所示的只读工作负载.在第 1 个版本(锁方式)中,读操作在所有路径上需要获取锁(类似 STM 路径);在第 2 个版本(检查方式),在所有路径上的读操作只需检查锁的状态(就像慢速 HTM 路径一样);在第 3 个版本(非同步方式),所有路径上的读操作都是简单的不插桩读(就像快速 HTM 路径一样).这些算法仅适用于只读工作负载,结果如图 3(f)所示,检查锁状态方式的速度比获取锁方式快 5 倍以上,而且不插桩读的速度比检查锁状态的方式快大约 20%.

4 分离事务执行

4.1 相关工作

PHTM 承诺更轻量的同步,但是 2 个固有的限制使得在数据库中应用 PHTM 面临巨大挑战.一个限制是 PHTM 事务处理受到 L1 缓存大小的限制,对于完整的数据库事务来说 L1 缓存大小通常太小.另一个是对冲突的过度反应,即使在低争用水平下有时也会导致序列化执行.

我们将一个数据库事务分离为多个满足 PHTM 容量限制的较小块,这样不太容易发生争用,随后将它们重组为单个原子事务,以降低额外开销.本文仅介绍利用 PHTM 的数据库事务提交.

我们所在的华为数据库实验室还为并行数据库的新型光纤解决方案的发展[21]以及实时分析型数据库的发展作出了贡献[22].

4.2 STE 算法

STE 由一组并发工作线程执行.每个工作线程都有一个唯一的 ID(tid)和一个单调递增的本地版本计数器(tv).另外还维护一个全局的最后提交版本数组(lca).数据库事务由 tid 和 tv 唯一标识.

每个线程在 lca 中有一个槽位,并且在数据库成功提交时在 lca 的槽位中写入 tv,即 $lca[tid]\leftarrow tv$,然后在本地增加 tv.一个数据库行同样有属性 rid 和 rv,其值为最后一个事务 T 写入的 tid 和 tv.如果 T 未提交,则该行指向到 $prev$ 的链接,即该行的最后提交版本,包括其 rid,rv 和数据.如前所述,$prev$ 链接仅在事务正在写入行时设置.总的来说,$prev$ 链接指向数据库事务 T 的回退集.

算法 1. 验证以及提交或终止.

```
① function ValidateCommit(T)
②   status←commit;
③   _xbegin(); /* Start HTM */
④   for e∈T.rs do
⑤     if e(rid, rv)=e.row(rid, rv) then
⑥       continue;
⑦     end if
⑧     if lca[e.row.rid]≥e.row.rv then
⑨       /* Newer writer is committed */
⑩       status←aborted;
⑪       break;
⑫     end if
⑬    /* Current writer (possibly self) is live */
⑭     if e.row.prev(rid, rv)≠e(id, rv) then
⑮    /* Already different committed data */
⑯       status←aborted;
⑰       break;
⑱     end if
⑲   end for
⑳   if status=commit then
㉑     lca[tid]=tv;
㉒   end if
㉓   tx_end_log(T.logged);
       /* Commit PHTM */
㉔   if status=commit then
㉕     increment(tv);
㉖   else
㉗     rollback(T);
㉘   end if
㉙   cleanup(T). /* Truncate read-set and
       undo-set */
㉚ end function
```

我们将数据库事务分离为访问行进行读取或写入的小型 PHTM 事务,并使用另一个 PHTM 事务来执行数据库事务的验证和提交.最后一个 HTM 事务可能会更大,但它是只读的,直到它最终写入 lca 并立即提交.英特尔的 HTM 正在使用大型布隆过滤器来检测冲突,同时允许从 L1 缓存中清除读取设置的条目,而不中止 HTM 事务处理.这样,

HTM 事务可以容纳非常大的读取集,并且可以支持大型只读前缀.

4.3 生效和提交

生效和提交函数的伪代码在算法 1 中描述. 行③~㉓执行一个 HTM 事务,验证读者乐观读的值并未被新提交的数据库事务所覆盖,然后在行㉑通过在 *lca* 中设置事务版本 *tv* 的方式提交写者悲观写的值. 在 HTM 事务之后,如果提交成功,那么在行㉕增加本地 *tv*,否则在行㉗中止并执行回滚操作.

为了验证读者看到满足一致性的结果,即最后提交的值,我们进行了 2 步验证. 如果读者记录的此行所记录的 *rv* 和 *rid* 不变,与在行⑤中所验证的相同,则读者看到最后提交的值. 第 2 种情况下当前的 *rv* 和 *rid* 不同于日志中记录的,但它们代表一个进行中的事务记录,而日志中记录的 *rv* 和 *rid* 是从 *prev* 链接的,所以记录的数据仍然是最后提交的版本,在这种情况下读仍然是有效的.

5 结 论

我们预计未来的系统将使用数千个处理器核心和 PB 级的 NVM. 基于锁的同步以及传统的基于日志的持久性将在这些系统中引入不可接受的开销. PHTM 是向 ACID 事务迈出的第一步,避免锁定并以面向 NVM 的方式提供持久性.

最近,数据库已经开始弃用磁盘,成为完全的内存数据库. 高效持久的混合事务内存将使得内存数据库能够从事务内存相关文献中积累的研究中受益. 另一种方法是基于 PHTM 定制的同步协议,如 STE,利用 PHTM 加速不受限数据库事务.

参 考 文 献

[1] Dushyanth N, Orion H. Whole-system persistence [J]. ACM SIGARCH Computer Architecture News, 2012: 40 (1): 401-410

[2] Intel Corporation. Intel architecture instruction set extensions programming reference [EB/OL]. (2017-04-28) [2017-10-08]. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf

[3] Avni H, Levy E, Mendelson A. Hardware transactions in nonvolatile memory [C] //Proc of the 29th Int Symp on DISC 2015. Berlin. Springer, 2015: 617-630

[4] Karnagel T, Dementiev R, Rajwar R, et al. Improving in-memory database index performance with Intel® transactional synchronization extensions [C] //Proc of the 20th IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2014: 476-487

[5] Schwalb D, Faust M, Dreseler M, et al. Leveraging non-volatile memory for instant restarts of in-memory database systems [C] //Proc of the 32nd IEEE Int Conf on Data Engineering (ICDE). Piscataway, NJ: IEEE, 2016: 1386-1389

[6] Herlihy M, Moss J E B. Transactional memory: Architectural support for lock-free data structures [J]. ACM Sigarch Computer Architecture News, 1993, 21(2): 289-300

[7] Bernstein P A, Hadzilacos V, Goodman N. Concurrency Control and Recovery in Database Systems [M]. Boston: Addison-Wesley Longman Publishing Co, Inc, 1987

[8] Torvald Riegel. Tm support in the GNU compiler collection [EB/OL]. (2012-02-06)[2017-10-08]. http://gcc.gnu.org/wiki/TransactionalMemory

[9] Riegel T. Software Transactional Memory Building Blocks [D]. Dresden: Technische Universität Dresden, 2013

[10] Coburn J, Caulfield A M, Akel A, et al. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories [C] //Proc of the 16th Int Conf on ASPLOS. New York: ACM, 2011: 105-118

[11] Herlihy M, Luchangco V, Moir M, et al. Software transactional memory for dynamic-sized data structures [C] //Proc of the 22nd Annual Symp on Principles of Distributed Computing. New York: ACM, 1993: 92-101

[12] Felber P, Fetzer C, Riegel T. Dynamic performance tuning of word-based software transactional memory [C] //Proc of the 13th ACM SIGPLAN Sym on Principles and Practice of Parallel Programming. New York: ACM, 2008: 237-246

[13] Volos H, Tack A J, Swift M M. Mnemosyne: Lightweight persistent memory [J]. ACM SIGARCH Computer Architecture News, 2011, 39(1): 91-104

[14] Leis V, Kemper A, Neumann T. Exploiting hardware transactional memory in main-memory databases [C] //Proc of the 30th IEEE Int Conf on Data Engineering. Piscataway, NJ: IEEE 2014: 580-591

[15] Wang Zhaoguo, Qian Hao, Li Jinyang, et al. Using restricted transactional memory to build a scalable in-memory database [C] //Proc of the 9th European Conf on Computer Systems. New York: ACM, 2014: 26

[16] Dulloor S R, Kumar S, Keshavamurthy A, et al. System software for persistent memory [C] //Proc of the 9th European Conf on Computer Systems. New York: ACM, 2014: 15:1-15:15

[17] DeBrabant J, Arulraj J, Pavlo A, et al. A prolegomenon on OLTP database systems for non-volatile memory [EB/OL]. (2017-09-01) [2017-10-08]. http://www.adms-conf.org/adms_2014.html

[18] Guerraoui R, Kapalka M. On the correctness of transactional memory [C] //Proc of the 13th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2008: 175-184

[19] Yu X, Bezerra G, Pavlo A, et al. Staring into the abyss: An evaluation of concurrency control with one thousand cores [J]. Proceedings of the VLDB Endowment, 2014, 8(3): 209-220

[20] Brown T, Avni H. Phytm: Persistent hybrid transactional memory [J]. Proceedings of the VLDB Endowment, 2016, 10(4): 409-420

[21] Gasiunas V, Dominguez-Sal D, Acker R, et al. Fiber-based architecture for NFV cloud databases [J]. Proceedings of the VLDB Endowment, 2017, 10(12): 1682-1693

[22] Braun L, Etter T, Gasparis G, et al. Analytics in motion: High performance event-processing AND real-time analytics in the same database [C] //Proc of the 2015 ACM SIGMOD Int Conf. on Management of Data. New York: ACM: 251-264

Hillel Avni, born in 1971. PhD. His main research interests include concurrent programming and databases.

Wang Peng, born in 1983. PhD. His main research interests include database theory and storage medium.

2016 年《计算机研究与发展》高被引论文 TOP10

排名	论文信息
1	刘峤, 李杨, 段宏, 刘瑶, 秦志光. **知识图谱构建技术综述**[J]. 计算机研究与发展, 2016, 53(3): 582-600 Liu Qiao, Li Yang, Duan Hong, Liu Yao, Qin Zhiguang. Knowledge Graph Construction Techniques [J]. Journal of Computer Research and Development, 2016, 53(3): 582-600
2	刘知远, 孙茂松, 林衍凯, 谢若冰. **知识表示学习研究进展**[J]. 计算机研究与发展, 2016, 53(2): 247-261 Liu Zhiyuan, Sun Maosong, Lin Yankai, Xie Ruobing. Knowledge Representation Learning: A Review [J]. Journal of Computer Research and Development, 2016, 53(2): 247-261
3	张蕾, 章毅. **大数据分析的无限深度神经网络方法**[J]. 计算机研究与发展, 2016, 53(1): 68-79 Zhang Lei, Zhang Yi. Big Data Analysis by Infinite Deep Neural Networks [J]. Journal of Computer Research and Development, 2016, 53(1): 68-79
4	王兴伟, 李婕, 谭振华, 马连博, 李福亮, 黄敏. **面向"互联网+"的网络技术发展现状与未来趋势**[J]. 计算机研究与发展, 2016, 53(4): 729-741 Wang Xingwei, Li Jie, Tan Zhenhua, Ma Lianbo, Li Fuliang, Huang Min. The State of the Art and Future Tendency of "Internet+" Oriented Network Technology [J]. Journal of Computer Research and Development, 2016, 53(4): 729-741
5	孟小峰, 杜治娟. **大数据融合研究: 问题与挑战**[J]. 计算机研究与发展, 2016, 53(2): 231-246 Meng Xiaofeng and Du Zhijuan. Research on the Big Data Fusion: Issues and Challenges [J]. Journal of Computer Research and Development, 2016, 53(2): 231-246
6	甘丽新, 万常选, 刘德喜, 钟青, 江腾蛟. **基于句法语义特征的中文实体关系抽取**[J]. 计算机研究与发展, 2016, 53(2): 284-302 Gan Lixin, Wan Changxuan, Liu Dexi, Zhong Qing, Jiang Tengjiao. Chinese Named Entity Relation Extraction Based on Syntactic and Semantic Features [J]. Journal of Computer Research and Development, 2016, 53(2): 284-302
7	单言虎, 张彰, 黄凯奇. **人的视觉行为识别研究回顾、现状及展望**[J]. 计算机研究与发展, 2016, 53(1): 93-112 Shan Yanhu, Zhang Zhang, Huang Kaiqi. Visual Human Action Recognition: History, Status and Prospects [J]. Journal of Computer Research and Development, 2016, 53(1): 93-112
8	庄严, 李国良, 冯建华. **知识库实体对齐技术综述**[J]. 计算机研究与发展, 2016, 53(1): 165-192 Zhuang Yan, Li Guoliang, Feng Jianhua. A Survey on Entity Alignment of Knowledge Base [J]. Journal of Computer Research and Development, 2016, 53(1): 165-192
9	付志耀, 高岭, 孙骞, 李洋, 高妮. **基于粗糙集的漏洞属性约简及严重性评估**[J]. 计算机研究与发展, 2016, 53(5): 1009-1017 Fu Zhiyao, Gao Ling, Sun Qian, Li Yang, Gao Ni. Evaluation of Vulnerability Severity Based on Rough Sets and Attributes Reduction [J]. Journal of Computer Research and Development, 2016, 53(5): 1009-1017
10	曹珍富, 董晓蕾, 周俊, 沈佳辰, 宁建廷, 巩俊卿. **大数据安全与隐私保护研究进展**[J]. 计算机研究与发展, 2016, 53(10): 2137-2151 Cao Zhenfu, Dong Xiaolei, Zhou Jun, Shen Jiachen, Ning Jianting, Gong Junqing. Research Advances on Big Data Security and Privacy Preserving [J]. Journal of Computer Research and Development, 2016, 53(10): 2137-2151

数据来源:CSCD,中国知网;统计日期:2017 年 11 月 27 日