计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.20210565
59(3):499-517,2022

一种基于时间戳的高扩展性的持久性软件事务内存

刘超杰 王 芳 邹晓敏 冯 丹
(华中科技大学武汉光电国家研究中心 武汉 430074)
(zkliuchaojie@163.com)

A Scalable Timestamp-Based Durable Software Transactional Memory

Liu Chaojie, Wang Fang, Zou Xiaomin, and Feng Dan
(*Wuhan National Laboratory for Optoelectronics*, *Huazhong University of Science and Technology*, *Wuhan* 430074)

Abstract The emerging non-volatile memory (NVM) provides a lot of advantages, including byte-addressability, durability, large capacity and low energy consumption. However, it is difficult to perform concurrent programming on NVM, because users have to ensure not only the crash consistency but also the correctness of concurrency. In order to reduce the development difficulty, persistent transactional memory has been proposed, but most of the existing persistent transactional memory has poor scalability. Through testing, we find that the limiting factors of scalability are global logical clock and redundant NVM write operation. In order to eliminate the impact of these two factors on scalability: A thread logical clock method is proposed, which eliminates the problem of global logical clock centralization by allowing each thread to have an independent clock; a dual version method of cache line awareness is proposed, which maintains two versions of the data, and updates the two versions cyclically to ensure the crash consistency of the data, thereby eliminating redundant NVM write operations. And based on these two methods, a scalable durable transactional memory (SDTM) is implemented and fully tested. The results show that under YCSB workload, compared with DudeTM and PMDK, its performance is up to 2.8 times and 29 times higher, respectively.

Key words durable transactional memory (DTM); concurrent control; non-volatile memory (NVM); data consistency; properties of atomicity, consistency, isolation, and durability (ACID properties)

摘 要 新兴的非易失性内存(non-volatile memory, NVM)具有字节寻址、持久性、大容量和低功耗等优点,然而,在NVM上进行并发编程往往比较困难,用户既要保证数据的崩溃一致性又要保证并发的正确性.为了降低用户开发难度,研究人员提出持久性事务内存方案,但是现有持久性事务内存普遍存在扩展性较差问题.测试发现限制扩展性的关键因素在于全局逻辑时钟和冗余NVM写操作.针对这2个方面,提出了线程逻辑时钟方法,通过允许每个线程拥有一个独立时钟,消除全局逻辑时钟中心化问题;提出了缓存行感知的双版本方法,为数据维护2个版本,通过循环更新这2个版本来保证数据的

收稿日期:2021-06-07;**修回日期**:2021-10-20
基金项目:国家重点研发计划项目(2018YFB1003305);国家自然科学基金重点项目(61832020);国家自然科学基金创新研究群体项目(61821003)
This work was supported by the National Key Research and Development Program of China (2018YFB1003305), the Key Program of the National Natural Science Foundation of China (61832020), and the Foundation for Innovative Research Groups of the National Natural Science Foundation of China (61821003).
通信作者:王芳(wangfang@mail.hust.edu.cn)

崩溃一致性,从而消除冗余的 NVM 写操作.基于所提出的这 2 个方法,实现了一个基于时间戳的高扩展的持久性软件事务内存(scalable durable transactional memory, SDTM),对比测试显示,在 YCSB 负载下,与 DudeTM 和 PMDK 相比,SDTM 的性能最多分别提高了 2.8 倍和 29 倍.

关键词 持久性事务内存;并发控制;非易失性内存;数据一致性;ACID 特性

中图法分类号 TP311

传统的单核处理器受限于功耗和散热问题,难以进一步提升运行频率[1-2].因而多核处理器成为了处理器设计的主流方向[3].为了充分发挥多核处理器的性能,必须编写并发的程序.然而,编写正确并且高效的并发程序往往是一件比较困难的事.这是因为在多线程同步方面,目前往往是基于传统的锁来实现,比如互斥锁、自旋锁和读写锁.粗粒度锁编程简单、易实现,但是扩展性不好,无法充分利用多核的性能;细粒度锁扩展性好,但是往往更难实现,编程时容易出错且难以调试,并且可能会由于锁定顺序的不同而导致死锁、优先级反转以及锁护送等问题.

为了降低在多核处理器上并发编程的难度,Herlihy 和 Moss[4] 提出了事务内存(transactional memory, TM).事务内存是一种轻量级的多线程同步机制,类似于数据库中的事务,可以提供原子性、一致性和隔离性的特性.事务内存按照实现方法分为 2 种:一种是软件事务内存(software transactional memory, STM),其完全由软件来实现[5-7];另一种是硬件事务内存(hardware transactional memory, HTM),其在 CPU 缓存层实现[8-9].当然,把这 2 种方法混合在一起便是混合事务内存[10-11](hybrid transactional memory, HyTM).鉴于软件事务内存具有的良好兼容性和可移植性,本文主要研究软件事务内存.

与此同时,内存技术也在快速发展,非易失性内存(non volatile memory, NVM)成为了当今工业界以及学术界研究的热点,比如 3D Xpoint[12]、相变存储器[13](phase-change memory, PCM)、自旋转移力矩随机存储器[14](spin-transfer torque random access memory, STT-RAM)和可变电阻式存储器[15](resistive random access memory, ReRAM)等.NVM 具有许多优点,比如容量大、非易失、能耗低,并且支持 CPU 的 load/store 指令.但 NVM 也有一些缺陷,比如其读写延迟仍然比 DRAM 要高,并且读写不对称,以及有限的写次数等[16-18].总体来说,NVM 的出现将可能会改变计算机体系结构,为程序性能的提升带来了机遇.

基于 NVM 的应用程序需要自己保证数据的崩溃一致性.这是因为在机器掉电或者系统崩溃后,可能存在部分更新的数据,从而导致了数据不一致的问题.为了保障崩溃一致性,数据需要按照特定的顺序持久化到 NVM.英特尔 x86 架构提供了内存屏障(MFENCE)以及缓存刷回指令(CLFLUSH)来保证数据持久化的顺序性[19-20].除此之外,还需要用户编写特定的崩溃恢复程序,从而在系统崩溃后将数据恢复到一个一致性的状态,这增加了在 NVM 上编程的难度.另外,如果想要充分发挥多核处理器的性能,用户既要保证并发正确性又要保证崩溃一致性,这进一步增加了用户的编程负担,降低了用户的开发效率.

为了降低在 NVM 上编写并发程序的难度,研究人员提出了持久性事务内存(durable transactional memory, DTM)[21-22],通过扩展传统事务内存,即增加持久性的特性,在保证并发正确性的同时又能保证数据的崩溃一致性.目前,针对持久性事务内存已经有很多相关的工作,但是这些工作普遍存在扩展性较差的问题,即随着线程数量的增加,DTM 的性能无法继续提升甚至会快速下降[22-23].测试发现,导致扩展性差的原因主要有 2 个:1)中心化的全局逻辑时钟;2)大量冗余的 NVM 写操作.

大多数事务内存的实现都是基于时间戳的(或者称为基于时间的)[24-25].通过全局逻辑时钟,事务内存以较低的代价来检验在一个事务中操作的数据是否是一致的,从而保证并发访问的正确性.然而,全局逻辑时钟一般是通过一个全局的整型变量来实现的,并通过原子指令或者锁来保证多线程同时修改的正确性.在持久性事务内存中,每个更新的事务都需要向全局逻辑时钟申请一个时间戳.当运行线程数量增加时,多个线程同时更新全局变量会引起大量的缓存争用,特别是跨 NUMA(non-uniform memory access)节点时,这种缓存争用的开销更大.因此,全局逻辑时钟在多线程环境下有较差的扩展性,从而限制了持久性事务内存的扩展性.

另外,为了保障数据的崩溃一致性,在更新数据时,往往不能直接原地修改,这是因为如果此时发生系统崩溃或者机器掉电,NVM 中将会存在部分更新的数据,进而导致数据不一致.通常的做法是先将数据在异地持久化之后再进行修改,由此便引入了大量冗余的 NVM 写操作[19,22].由于 NVM 的写延迟较高,冗余的写操作将导致事务的执行时间变长,在事务内存中,更新事务在完成前会一直持有所要修改的数据的锁,其他访问该数据的事务因无法获得锁而处于等待的状态,从而造成持久性事务内存较差的扩展性.

本文主要针对持久性事务内存中的全局逻辑时钟和大量冗余的 NVM 写操作问题,设计了高扩展性的线程逻辑时钟以及缓存行感知的双版本方法,并实现了一个高性能且高扩展性的持久性软件事务内存.

本文工作的主要贡献有 3 个方面:

1) 针对全局逻辑时钟的扩展性较差,本文提出线程逻辑时钟,允许每个线程都拥有一个独立时钟,并采用了基于数据驱动的时钟获取方法,完全消除了中心化问题,为用户提供了一个高扩展性的时钟原语;

2) 现有崩溃一致性保障方法引入了大量 NVM 写操作,本文提出了缓存行感知的双版本方法,利用 NVM 缓存行特性,通过在 NVM 上连续存储 2 个版本以及轮流更新来保证崩溃一致性,从而消除了冗余的 NVM 写操作;

3) 基于线程逻辑时钟和缓存行感知的双版本实现了一个高扩展性的持久性软件事务内存(scalable durable transactional memory, SDTM),并在真实 NVM 器件上进行了充分的测试.测试结果显示,在真实负载下,SDTM 相比于 DudeTM 和 PMDK,其性能最高分别提升了 2.8 倍和 29 倍.

1 相关工作

目前,针对持久性事务内存已经有很多相关的研究,为了提升其性能以及扩展性,各种各样的优化被应用到持久性事务内存上.表 1 展示了最新的研究中所使用的优化方法,这些优化可以分为 2 类:1)基于硬件辅助的优化;2)基于多版本或者多副本的优化.基于硬件辅助的方法主要使用硬件时钟以及硬件事务内存来提升性能,这种基于硬件辅助的方法虽然能够在一定程度上提升持久性事务内存的扩展性,但是存在兼容性以及可移植性的问题.多版本和多副本的方法分别主要用来提升只读事务性能和降低崩溃一致性保障开销,不过,这些方法浪费了一定的存储空间,是对性能和存储的折衷.

由表 1 可以发现,目前大部分持久性事务内存都是基于软件事务内存的,和硬件事务内存相比,软件事务内存不依赖于特定的硬件,具有更好的实用性.DudeTM[25]既支持软件事务内存又支持硬件事务内存,后面将进行详细的说明.需要指出的是,PMDK[26]仅仅支持单线程,为了使其支持多线程,按照通用的方法,对其增加了读写锁.KaminoTX[27]和 Romulus[28]采用的也是基于锁的并发控制,和基于时间戳的并发控制相比,其对每个数据的访问都要获得锁,由此带来了大量的加锁开销.本文将主要研究基于时间戳的并发控制[29],基于锁的并发控制不再做过多的讨论.

Table 1 Optimizations of Durable Transactional Memory

表 1 不同的持久性事务内存采用的优化方法

事务内存	逻辑时钟	硬件时钟	STM	HTM	单版本	多版本	多副本
PMDK[23]			√		√		
KaminoTX[27]			√				√
Romulus[28]			√				√
Mnemosyne[19]	√		√		√		
DudeTM[22]	√		√	√			√
Pisces[20]	√		√			√	
TimeStone[26]		√	√			√	

注:"√"表示支持.

1.1 基于硬件辅助的持久性事务内存

为了提升持久性事务内存的性能,可以采用一些硬件辅助的方法,这主要体现在 2 个方面:1)硬件时钟;2)硬件事务内存.基于时间戳的持久性事务内存

需要一个统一的时钟来产生独一无二的时间戳,该时间戳被用作数据的版本号,当执行事务时,可以用该版本号来校验数据的一致性[25].因此,需要为每个更新事务分配一个时间戳,当更新事务较多时,时间戳的产生速度很可能会成为性能的瓶颈.按照实现方法的不同,时钟可以分为2种:1)逻辑时钟[25,30];2)硬件时钟[31-32].

逻辑时钟通常使用一个全局的整型变量来实现,每次分配时将其值加1.为了保证多线程更新的正确性,可以使用锁或者原子指令来对其进行保护,但是,当线程数量增加时,特别是超线程或者跨NUMA[33]节点时,这种方法会引入大量的缓存争用,导致时间戳的产生速度急剧下降,进而成为了整个系统扩展性的瓶颈[34-35].为了解决这个问题,一些研究直接使用处理器自带的时钟,即硬件时钟.处理器时钟的值可以使用特定的指令来获得,由于时钟的值是单调递增的,因此其产生的时间戳也是独一无二的.在NUMA架构下,每个处理器都拥有一个自己的时钟,因此多个硬件时钟需要进行同步.然而,几乎所有类型的处理器都无法提供一个全局同步且高分辨率的时钟,这是因为提供这种时钟的开销较大,在硬件上很难实现[31].

目前,一些主流的处理器开始提供一种不变硬件时钟(invariant hardware clocks),该时钟拥有一个重要的特性,即其频率不随处理器的频率改变,始终以固定的频率运行[36-37].根据这个特性可以发现,虽然多个处理器无法提供全局同步的时钟,但是它们之间的时钟仅仅存在一个固定的偏差.因此,Kashyap等人[31]利用不变时钟提供了一个全局同步的时钟,即ORDO,并提供了一套相应的接口,用户可以使用这些接口直接将逻辑时钟替换成ORDO.TimeStone[26]直接使用了ORDO,从而消除由于逻辑时钟而导致的性能瓶颈.但是,ORDO也存在一些问题,比如并不是所有的处理器都支持硬件时钟,因此,和逻辑时钟相比,其兼容性和可移植性较差.

除了使用硬件时钟来消除逻辑时钟扩展性的瓶颈外,还可以使用硬件事务内存来降低事务内存的开销[38].相比于软件事务内存,硬件事务内存直接在缓存一致性协议中检查数据的一致性,因此具有更低的开销.目前,英特尔公司的Haswell架构处理器已经支持了硬件事务内存,即受限事务内存(restricted transactional memory, RTM)[39].除此之外,ARM和IBM的Power架构也提供了对硬件事务内存的支持.另外,硬件事务内存提供了和软件事务内存类似的接口,因此在持久性事务内存中,将软件事务内存替换成硬件事务内存是十分简单的.

如表1所示,DudeTM的实现既可以使用基于时间戳的软件事务内存,又可以使用硬件事务内存.当使用硬件事务内存时,所有的事务内存功能都在硬件层实现,可避免时钟的使用.现有的持久性事务内存大多采用日志保障崩溃一致性,日志和数据之间严格的持久化顺序约束引起了很大的开销.为了降低此开销,DudeTM解耦了持久性内存在关键路径上的操作.其在DRAM中维护了一个非持久性的数据副本,所有的事务操作均在此副本上执行.对于更新操作,当事务完成后,再按照事务执行的顺序异步(或同步)写回到NVM上的持久性副本中.

尽管硬件事务内存可大大提升持久性事务内存的性能,但是其也存在3个问题:1)硬件事务内存受限于CPU缓存的容量限制,只能有效支持小事务;2)有些架构的处理器不支持硬件事务内存,如MIPS和alpha架构;3)对于同一架构,不同的版本对硬件事务内存的支持也有区别,比如IBM的POWER9支持硬件事务内存,但在POWER10中又删除了硬件事务内存.因此,基于硬件辅助的持久性事务内存虽然可以消除时钟的瓶颈以及提升事务内存的性能,但是有限的容量和较差的可移植性导致其实用性较低.

1.2 基于单版本/多版本/多副本的持久性事务内存

1) 基于单版本的持久性事务内存

为了保证数据的崩溃一致性,往往不能对数据进行就地更新,这是因为在更新时如果发生系统崩溃或者掉电,将会导致部分更新,并且在系统重启后也无法恢复到一个一致性的状态[40-41].为了解决这个问题,之前的工作通常会采用写前日志的方法来保证数据的崩溃一致性.日志主要分为undo日志和redo日志.

PMDK[23]采用了undo日志,即在修改数据之前,将原始数据持久化到日志中,如果发生崩溃,可以借助日志将数据恢复到修改前的状态.该方法存在一个很大的问题,即针对每一个更新,都必须等到日志持久化之后才可以进行修改,即将事务分割成多个交替的日志和数据,这增加了顺序化的约束且位于关键路径上,影响了持久性事务内存的性能.和PMDK不同,Mnemosyne[19]采用了redo日志的方法,其将更新的数据存储在日志中,因此允许事务内存将一个事务涉及的所有日志一起持久化到NVM,

再进行数据更新.相比于 undo 日志减少了顺序化约束的开销,但是对更新数据的读要重定向到日志中,增加了地址重映射的开销,并且也存在至少 1 倍的写放大问题.另外,在并发控制上,Mnemosyne 采用的是全局逻辑时钟,其扩展性被严重制约.

2) 基于多版本的持久性事务内存

在事务内存中,多版本的方法经常用来降低更新事务对只读事务的阻塞,提升其性能.当更新事务正在更新某个数据时,只读事务是无法对该数据进行读取的,只能等待或者重试.这是因为更新事务还没有提交,数据处于一种中间状态,此时读取数据将破坏事务的隔离性与原子性.因此,更新事务会阻塞只读事务的执行,造成只读事务产生较大的延迟.特别是对于执行时间较长的只读事务,比如遍历所有数据的操作,在最坏的情况下,该操作可能一直在重试,从而永远无法完成.

Pisces[20]使用了双版本来提升读操作的性能,其将 redo 日志作为一个新的版本来为读操作服务.具体来说,在更新数据时,Pisces[20]先把更新写入日志,此时日志中便存储了该数据的第 2 个版本中.当有其他事务读取该数据时,便可以利用这个较新的版本为读操作服务.然而,Pisces[20]放松了对一致性的约束,仅仅提供了快照隔离级[42],将会导致用户编程复杂.另外,日志仍然需要写回到本地,依旧存在写放大的问题.TimeStone[26]使用多版本来实现了更高的扩展性和支持不同的隔离级,其通过操作日志来保证持久性,并在 DRAM 中维护了多个版本来提升持久性事务内存的性能.然而,TimeStone 依赖于硬件时钟,并且操作日志和 redo 日志类似,仍然存在写放大的问题.

3) 基于多副本的持久性事务内存

经过分析可以发现,无论是 undo 日志还是 redo 日志,都存在一些问题.首先,它们都会导致冗余的 NVM 写操作.其次,undo 日志会引入较多的持久化顺序约束;而 redo 日志由于将数据写到了异地,在事务中访问这些未提交的数据时,将会引入数据重定向的问题[22].这些问题都会影响持久性事务内存的性能以及扩展性,为了解决这些问题,一些研究开始使用多副本的方法对其进行优化.

DudeTM[22]通过非持久性副本和持久性副本来解耦更新事务在关键路径上的操作,并通过重做日志的方法将事务持久化.但是为了保证数据的一致性,DudeTM 仅使用一个线程来重做日志,因此,其扩展性较差.KaminoTX[27]和 Romulus[28]为数据维护了 2 个持久性副本,称为主副本和从副本,所有的更新操作直接在主副本中进行,等到主副本持久化之后,再异步写回到从副本中.如果发生崩溃,可以使用从副本(或主副本)来恢复,因此消除了关键路径上的阻塞.同样,这 2 种方法都存在写放大的问题.和 DudeTM 类似,Romulus 使用单线程来将数据写回到从副本,因此扩展性较差.而 KaminoTX 在将数据持久化到从副本的过程中都需要持有锁,阻碍了其他事务的执行,影响了扩展性.另外,基于多副本的方法由于需要额外的存储数据,因此对 NVM 的空间造成了浪费.

2 持久性事务内存扩展性测试

本节通过测试来分析制约基于时间戳的持久性事务内存扩展性的因素.其中,2.1 节测试了全局逻辑时钟的扩展性;2.2 节分析了在崩溃一致性保障机制中,冗余的 NVM 写操作对持久性事务内存扩展性的制约.

2.1 全局逻辑时钟

在基于时间戳的事务内存中,有 3 个操作涉及时钟:1)事务开始时,读取时钟的值,称为起始时钟,并记录在局部变量 *sc*(start clock)中;2)读取数据时,通过 *sc* 和数据的时间戳来检验数据的一致性;3)提交事务时,将时钟的值加 1 并返回,该返回值即是提交时间戳 *ct*(commit timestamp).这 3 个操作封装成 3 个 API,即 *get_time*,*cmp_time*,*new_time*.

目前,使用最广泛的是全局逻辑时钟,其实现方式简单并且不依赖于特定硬件.该时钟可以是一个全局整型变量,其初始值为 0,并使用锁或者原子指令来保证并发的正确性.图 1 展示了在不同的并发保护方法下,全局逻辑时钟的扩展性.基于 C++ mutex 的方法在修改全局逻辑时钟前,需先获取互斥锁,然后再进行修改;基于 *add_and_fetch* 或 *compare_and_swap*(CAS)指令的方法可以对全局逻辑时钟直接进行更新.测试所用的机器有 2 个 NUMA 节点,每个 NUMA 节点有 36 个核,这里使用了 1~64 个线程进行测试.从图 1 中可以发现,无论是基于哪种并发控制的时钟,其扩展性都很差.当线程数量为 1 时,其性能是最好的,这是因为在只有一个线程的情况下,不会产生任何缓存竞争的开销.随着线程数量的增加,便有更多的缓存需要同步,维护缓存一致性的开销也就越大.

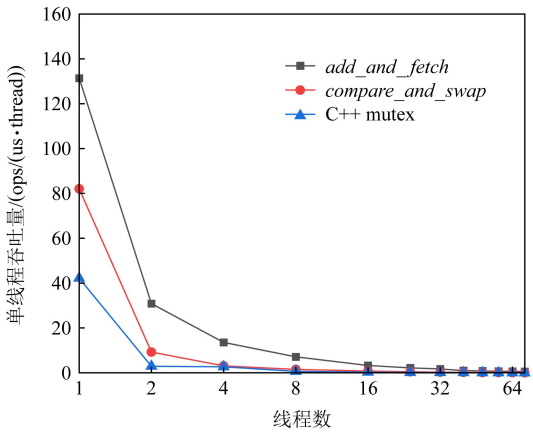


Fig. 1 The scalability of global logical clock

图 1 全局逻辑时钟的扩展性

TL2 是经典的基于全局时间戳的软件事务内存,借助于 TL2 算法,本文实现了一个并发的链式散列表,并用链表来解决冲突.图 2 展示了在不同数量的线程下执行插入操作时的吞吐量以及时钟操作所占的开销.从中可以发现,随着线程数量的增加,散列表的吞吐量在达到 8 个线程便开始缓慢增加(32～40 线程出现下降,是因为跨域了 NUMA 节点),而全局逻辑时钟所占的开销也是越来越大,最高达到了 79%.

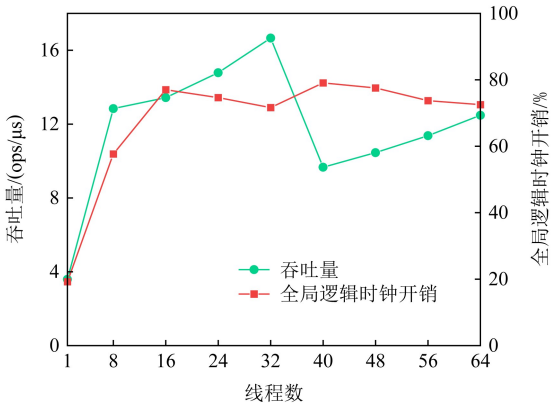


Fig. 2 The throughput of Hash table and the overhead of clock

图 2 散列表的吞吐量以及时钟开销

基于图 1 与图 2 的分析可以发现,全局逻辑时钟的扩展性较差,严重制约了事务内存的扩展性,设计并实现一个低开销且高扩展性时钟成为提升持久性事务内存扩展性的重要途径.虽然目前已经有一些方法对逻辑时钟进行了研究,但是这些方法仅仅缓解了时钟扩展性问题,而没有真正的解决.Silo[43]使用了批量的原子修改,根据 DBX1000[44]测试,当存在较多冲突时,该方法的扩展性较差;TicToc[45]消除了全局时钟,但是这种方法并不保证事务的 Opacity[46]特性;Zhang 等人[47]试图通过优化事务的提交阶段来减少对时钟不必要的更新,仅仅缓解了时钟的问题.

2.2 冗余的 NVM 写操作

为了保证数据一致性,持久性事务内存通常采用 undo 日志或 redo 日志.显然,两者都存在大量冗余的 NVM 写操作,这主要体现在:为了使数据在崩溃后可以恢复一致性的状态,需要将数据在异地进行持久化,这种异地的持久化至少增加了 1 倍的写操作(对于 redo 是 2 倍,因为还要将数据写回到本地).图 3 展示了在使用 undo 或者 redo 日志的情况下,基于时间戳的持久性事务内存的吞吐量.这里使用持久性事务内存实现了一个并发且持久性的二叉搜索树,并用读写混合负载(50%读,50%更新)对其进行测试.由图 3 可以发现,不管使用哪种日志,持久性事务内存的扩展性都不是很好,在超过 16 个线程后,吞吐量便不再有明显的提升.并且,在达到 40 个线程时,吞吐量有明显的下降,这是因为跨越了 NUMA 节点,导致了远程内存访问.

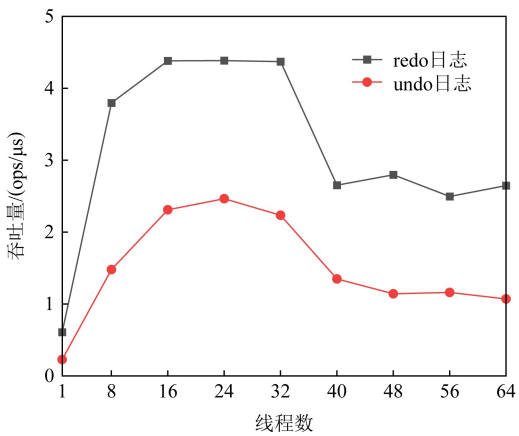


Fig. 3 The throughputs of transactional memory with different logs

图 3 不同的日志方法下事务内存的吞吐量

为了进一步探究 NVM 的写操作对持久性事务内存扩展性的影响,本文测试了在不同的访问粒度下随机写操作的延迟,如图 4 所示.从图 4 中可以发现,针对 DRAM 的随机写延迟最低,而针对 NVM 的随机写,其延迟大概是 DRAM 的 3 倍.另外,为了将数据进行持久化,需要调用缓存写回指令(CLFLUSH 或 CLWB)将数据从 CPU 缓存写回 NVM 中,从图 4 可以看出,缓存写回指令也会带来

较多延迟.另外,为了保证持久化的顺序,需要调用内存屏障指令(MFENCE)确保前面指令完成再执行后面指令.基于 undo 日志的持久性事务内存需要使用大量的内存屏障,这是因为当更新数据时,必须等待针对该数据的日志持久化到 NVM 后才可以更新,这引入了大量的等待,且都位于关键路径.而 redo 日志只需要在数据写回时调用一次内存屏障指令,因此 redo 日志的扩展性以及性能要明显优于 undo 日志.

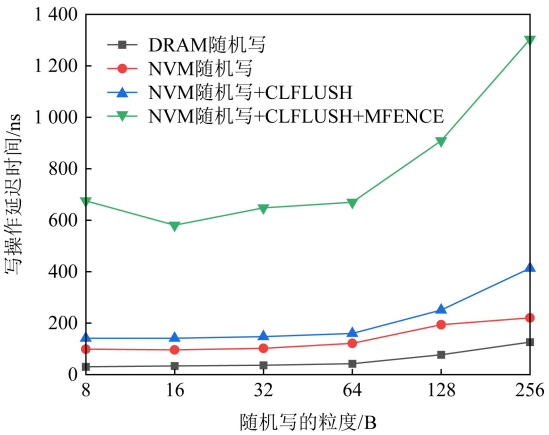


Fig. 4 The latency of NVM and DRAM under random write

图 4 NVM 和 DRAM 随机写操作的延迟

尽管基于 redo 日志不需要使用大量的内存屏障指令,但是带来了更多的 NVM 写操作,并且引入了数据重定向的开销,即在访问数据时必须重定向到日志中,这增加了一次缓存缺失的开销.大量冗余的 NVM 写操作是这样影响持久性事务内存的扩展性的:由图 4 可知,NVM 上的写操作延迟较高,会导致事务执行时间更长;而在持久性事务内存中,更新事务在完成前会一直持有所要修改的数据的锁,这会导致其他需要访问该数据的事务由于无法获取锁而处于等待的状态,从而严重影响了事务内存的扩展性.因此,提升扩展性的核心在于减少单个事务的执行时间,从而降低不同事务之间由于锁的争用而导致的阻塞,undo 日志和 redo 日志显然都没有很好地解决这个问题.

3 高扩展性的线程逻辑时钟

本文提出了一种线程逻辑时钟方案:通过允许每个线程都拥有自己的时钟,消除了全局逻辑时钟中心化问题;采用数据驱动的时钟获取方法,完全消除了缓存争用的开销;通过动态扩展起始时钟,大大降低了事务的终止率.基于这些优化,线程逻辑时钟拥有了近似于硬件时钟的性能以及扩展性.

3.1 时间戳的构成

线程逻辑时钟和全局逻辑时钟最大的区别在于其允许每个线程都拥有一个自己的逻辑时钟,并且每个线程只能修改自己的逻辑时钟,而不能修改其他线程的.为了生成独一无二的时间戳,使用线程 ID 和此线程的时钟值共同构成时间戳.由于所有线程的 ID 是不同的且每个线程的时钟都是单调递增的,因此,使用这种方法构成的时间戳是一定不会重复的.图 5 展示了时间戳的结构,大小是 64 b,其中前 n 位存储线程 ID,后 $64-n$ 位用来存储逻辑时钟的值.n 的大小由线程数量决定,须满足 2^n 大于或等于线程数.

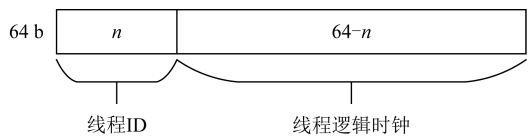


Fig. 5 The composition of timestamp under thread logical clock

图 5 线程逻辑时钟下时间戳的构成

和全局逻辑时钟一样,线程逻辑时钟使用整型变量实现,只是不再是全局变量,而是线程局部变量(thread-local variables),从 C++11 开始已经支持该类型.对于线程 ID,操作系统通常会为每个线程分配单独的值.但是系统提供的线程 ID 值一般较大,这会导致存储线程逻辑时钟值的空间变小,进而容易引起时钟的溢出.并且,当线程运行结束后,其 ID 不会立即被回收,而是等达到最大值的时候再回收.那么随着线程的创建与销毁,在运行时系统中便需要维护大量的线程 ID,这会带来一定的存储开销.因此,线程逻辑时钟不使用操作系统提供的线程 ID,而是自己为线程分配.

为了对线程 ID 进行管理,设计了一个线程 ID 分配器.由于线程 ID 的分配与回收仅仅在线程创建和销毁时被调用,其性能不会成为系统的瓶颈,因此设计了一个全局的 ID 分配器,并用 C++中的互斥锁来保证并发的正确性.线程 ID 是从 0 开始分配的,和线程 ID 绑定的还有其对应的逻辑时钟,其初始值也是 0.当新的线程被创建时,便会从分配器中搜索一个空闲的 ID 分配给它,相应的也将该 ID 对应的逻辑时钟交给该线程使用.当线程被销毁时,该

线程 ID 和该线程的逻辑时钟被一起回收,但时钟的值不会被重置,这样当再次分配后,便不会出现重复的时间戳.

3.2 基于数据驱动的时钟获取方法

虽然每个线程的逻辑时钟只有该线程自己可以修改,但是多个线程的读操作也会带来大量的缓存争用开销.这是因为根据缓存一致性协议,当一个线程读取一个变量后,便在自己的缓存里增加了该变量的一个副本,如果有其他线程对该变量进行了修改,那么这个线程里的副本便需要被同步,通常的做法是置为无效状态.当线程数量增加,这种同步的操作便会带来大量的开销.特别是在跨 NUMA 节点时,这种开销就会更大,由此严重影响了线程逻辑时钟的扩展性.

为了解决这个问题,需要避免使用全局变量.本文提出的基于数据驱动的时钟获取方法源自于一种简单的观察:数据的时间戳存储着线程逻辑时钟的历史值.根据 3.1 节的介绍,时间戳由 2 部分组成:线程 ID 和该线程的逻辑时钟值.当更新事务在提交时,会通过函数 *new_time* 获得独一无二的时间戳,即提交时间戳,并且该提交时间戳会被写入到被更新的数据的时间戳属性中.因此,在数据的时间戳属性中存储着所有线程的逻辑时钟的历史值.

这里使用了历史值,是因为并不是所有数据的时间戳都存储着逻辑时钟的最新值,根据线程逻辑时钟的执行流程,只有少量数据存储着最新值,大部分数据存储的是时钟的旧值,即比真正的逻辑时钟的值要小.使用这些较旧的时钟值来检验数据的一致性是完全可行的,这是因为时钟值总是单调递增的,如果事务开始后,其访问的数据被其他更新事务修改,那么被修改后的数据,其时间戳一定大于任何一个之前产生的旧的时间戳.和 TicToc[45] 相比,尽管都使用了基于数据驱动的方法,但是本文是为了获取起始时钟,而 TicToc 是通过数据的时间戳信息来计算得到提交时间戳.

图 6 是一个基于线程逻辑时钟的更新事务执行过程的例子,该事务所作的操作是先读取对象 A 再修改对象 B.该更新事务是由线程 0 执行的,因此线程 0 的起始时钟可以直接获得,方法 *get_time* 会把线程 0 的起始时钟设置为 1,线程 1 的起始时钟设置为 0.接着,事务开始读写数据.对象 A 校验通过,当使用 *cmp_time* 检验对象 B 时,发现其时间戳大于线程 1 的起始时钟,此时先将线程 1 的起始时钟修改为 4,然后终止该事务并重试.重试的事务其起始时钟分别是 1 和 4,此时数据校验都可以通过.最终该事务完成.

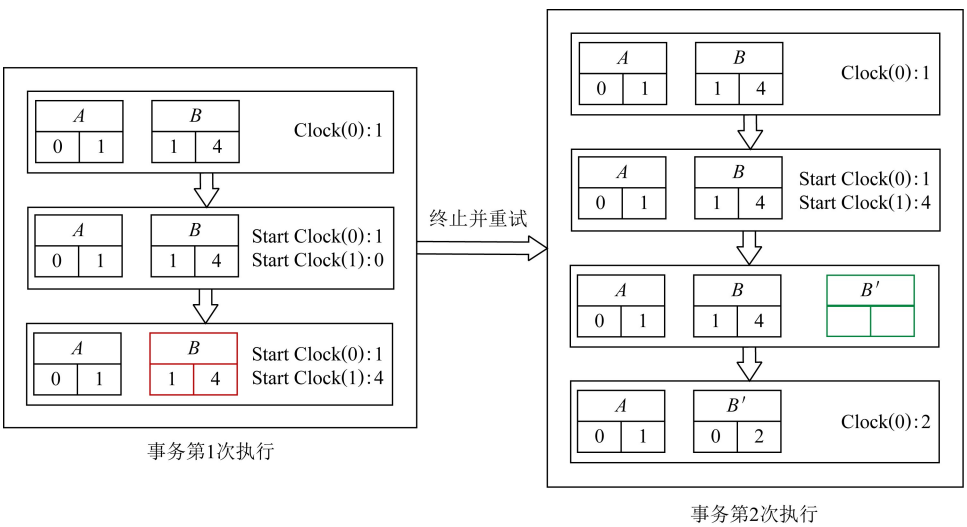


Fig. 6 An example under data-driven method

图 6 数据驱动方法下事务执行的过程

由更新事务的执行流程来看,基于数据驱动的方法直接从数据的时间戳中获取起始时钟,从而避免了访问全局变量,完全消除了缓存一致性的开销.但是,从图 6 的例子中也可以发现,该方法增加了事务的终止率.可以优化的一点是,当事务完成后,该事务记录的起始时钟可以存储下来,这样下个事务便可以直接使用上个事务的起始时钟,从而降低事务的终止率.即使通过这种方法优化后,事务的终止率仍然很高,这是因为基于数据驱动的方法获取的时钟值比当前的时钟值更旧,这意味着在进行数据

校验时更容易失败.尽管较高的终止率会影响事务的性能,但是和全局逻辑时钟相比,基于数据驱动的线程逻辑时钟仍然拥有更好的扩展性.

3.3 动态扩展起始时钟

基于数据驱动的时钟获取方法虽然能够消除缓存同步的开销,但所获取的时钟值往往偏旧.尽管使用这些偏旧的时钟值来校验数据的一致性是完全可行的,但是往往会带来较高的事务终止率,进而影响事务性能.这是因为事务终止的处理是使用函数 *sigsetjmp* 来实现的,在事务开始前,首先调用函数 *sigsetjmp* 来保存目前的堆栈环境,如果事务终止,便调用函数 *siglongjmp* 跳转到由前面保存的位置,继续该事务的执行.这种指令的跳转不利于 CPU 流水线的执行,并且对 CPU 的缓存也不友好,进而影响事务的性能以及扩展性.

为了解决这个问题,本文提出了一种动态扩展起始时钟的方法.传统的基于时间戳的事务内存,在事务开始时首先获得起始时钟,如果事务不发生终止,那么该时钟的值在该事务的执行过程中便不会再发生改变.然而,本文发现起始时钟在一定条件下是可以改变的,该条件是该事务所访问的数据在事务开始后没有被其他线程修改.这是因为如果满足该条件,那么和事务终止后再重新执行所达到的状态时完全相同的.因此当该条件成立时,将对应的起始时钟修改为一个更大的值仍然能够保证数据的一致性,同时避免了事务终止的发生.由于被更新的数据在访问时都会获取锁,所以一定不会被其他事务修改,因此不需要校验那些更新的数据,只需要对只读的数据(只读的数据会保存在集合中,称为 read-set)进行校验.

尽管采用动态扩展起始时钟的方法需要遍历 read-set 集合以检验数据的一致性,但是该方法总是能够提升事务的性能的.这是因为,如果不采用该方法,在事务终止并重试时,需要重新开始执行,这种重新执行也会将终止之前的数据再访问一遍.因此,动态扩展起始时钟的方法总是有效的,它在一定程度上避免了长指令跳转,有利于 CPU 流水线的执行以及利用 CPU 的缓存.

3.4 线程逻辑时钟的实现

线程逻辑时钟是使用 C++来实现的,其为用户提供了简单易用的 API,即 *get_time*,*cmp_time*,*new_time*,本节将主要描述这 3 个函数的实现.在介绍这些函数的实现之前,这里先来描述事务结构体 TX,该结构体的主要成员如表 2 所示.在 TX 结构体中,因为起始时钟不止一个,这里使用了数组来存储起始时钟,该数组的大小即为支持的最大线程数量.另外,分别使用集合 *read_set* 和 *wrtie_set* 记录数据读写过的数据,*read_set* 仅仅记录数据的地址,*write_set* 同时还保存着数据修改后的值.

Table 2 Data Members of Transaction Structure

表 2 事务结构体的主要数据成员

类型	数据成员	描述
uint32_t	*tid*	线程 ID 号
uint64_t	*tc*	线程独占的逻辑时钟
uint64_t[]	*scs*	所有线程的起始时钟
set	*read_set*	记录事务读取过的数据
set	*write_set*	记录事务修改过的数据

每个线程单独拥有一个 TX 结构体实例,在线程刚被创建时,需要对该实例进行初始化.首先从线程 ID 管理器中获得一个空闲的线程 ID,并对 *tid* 和 *tc* 进行初始化,然后将数组 *scs* 中的元素全部初始化为 0.为了能够复用 TX 结构体实例,特别是 *scs* 成员,避免针对每个事务都创建一个新的结构体,在事务开始前,需要对集合 *read_set* 和 *write_set* 清 0,其他的数据成员不需要改动.

线程逻辑时钟为用户提供了简单的 API,这些 API 的形式和全局逻辑时钟相似,以方便用户对传统的时钟进行替换.使用者可以完全不用了解线程逻辑时钟的具体实现方法,只需要调用这些 API 即可,下面将分别介绍这 3 个 API 的实现,如算法 1 所示:

算法 1. 线程逻辑时钟伪代码.

```
① Function get_time(tx)
②   tx.scs[tid]=tx.tc;
③ Function cmp_time(tx)
④   if ts.tc≤tx.scs[ts.tid] then
⑤     return true;
⑥   else
⑦     tx.scs[ts.tid]=ts.tc;
⑧     if tx.read_set 是一致的 then
⑨       return true;
⑩     end if
⑪     return false;
⑫   end if
⑬ Function new_time(tx)
⑭   tx.tc=tx.tc+1;
⑮   return(tx.tid≪(64−n))|tx.tc.
```

在基于数据驱动的时钟获取方法下,*get_time* 方法只需要获取自己线程的逻辑时钟值,并赋值给对应的起始时钟.*cmp_time* 首先检验数据的时间戳 *ts*,如果通过,直接返回 true,否则在事务的数组 *scs* 中更新该时间戳对应的起始时钟.下一步,即算法 1 的行⑧,检查集合 *read_set* 的一致性,如果检查通过,返回 true,否则返回 false.方法 *new_time* 为用户返回一个独一无二的时间戳,首先将该线程的时钟值加 1,然后将线程 ID 和该时钟值组合在一起并返回,该返回值即是更新事务的提交时间戳.

4 缓存行感知的双版本方法

为了消除数据崩溃一致性保障过程中的冗余 NVM 写操作,本文还提出了一种缓存行感知的双版本方法.该方法利用真实 NVM 器件的特性,将 2 个版本连续存储,并利用混合内存的架构来提升性能;另外,类似于传统的多版本,双版本可以用来提升只读事务的性能;给出了 2 种崩溃一致性恢复机制.

4.1 数据存储格式

缓存行感知的双版本结合了多副本和多版本的优点,既能够实现数据的直接更新和避免异地持久化操作,又能够利用双版本提升只读事务的性能.传统的多版本方案通常使用链表存储多个版本,动态地为新的版本分配空间,并且保证最新的版本被最先访问到.但是由于链表指针访问的缓存局部性不好,NVM 的读写延迟比 DRAM 高,其缓存缺失的开销也更大,因此缓存行感知的双版本采用了连续存储的数据布局方案,如图 7 所示:

lock	*new*	*V*0	*ts*0	*V*1	*ts*1

Fig. 7 Data layout of cache line conscious dual versions
图 7 缓存行感知的双版本的数据布局

其中 *lock* 是该数据的锁,这里用 CAS 指令实现了该锁.标记位 *new* 用来指定哪个版本是最新的,*new*=0 表示 *V*0 是最新的,*new*=1 表示 *V*1 是最新的.之所以使用 *new* 而不是直接比较 2 个版本的时间戳,是为了和线程逻辑时钟进行集成,因为线程逻辑时钟的时间戳是无法直接进行比较的,除非拥有相同的线程 ID.紧接着存储的是数据的 2 个版本 *V*0 和 *V*1,并附带它们的时间戳 *ts*0 和 *ts*1.数据是与 NVM 的缓存行对齐的,由于现有的 NVM 器件(Intel Optane DCPMM)的缓存行是 256 B[48],因此对于大部分数据结构来说 2 个版本可以存储在同一个缓存行中,从而避免了在访问第 2 个版本时所引起的缓存行缺失的开销.当然,这种提前为多版本分配存储空间的做法对存储空间造成了浪费,一种可行的方法是针对那些不经常更新的数据,仍采用单版本方法进行存储,因为那些只读的数据是不会从缓存行感知的双版本中得到性能提升的.

在基于缓存行感知的双版本方法中,数据的 2 个版本是循环被修改的,这样能保证在任何时候 NVM 中都存在一个一致性的版本,从而在系统崩溃或者掉电后将数据恢复到一致性的状态.例如,假设一个更新事务要修改数据 *A*,当前 *A* 的 *V*0 版本是最新的,在事务提交时,便将对 *A* 的修改直接持久化到 *A* 的 *V*1 版本.如果此时系统发生崩溃,在机器重启后可以使用 *A* 的 *V*0 版本将数据恢复到一致的状态.如果更新事务成功,那么 *A* 的 *V*1 成为了最新的版本,下次的修改将直接写到 *V*0 中.因此,缓存行感知的双版本方法完全消除了日志所引起的冗余 NVM 写操作.相比于多副本方法在主副本更新成功后还需将更新同步到备用副本而引起 NVM 写放大,缓存行感知的双版本方法是直接覆盖较旧的版本,不需要任何同步操作.

为了降低 NVM 较高的写延迟对事务内存扩展性的影响,缓存行感知的双版本还充分利用了混合内存的优势.目前,NVM 的读写延迟和 DRAM 相比仍然有比较大的差距,很难在短时间内取代 DRAM,在未来较长时间内,NVM 和 DRAM 共存于计算机系统将成为一种常态.因此,可以利用 DRAM 来提升持久性事务内存的性能,本文设计了一种基于混合内存的存储架构,数据完全存储在 NVM 中以保证持久性,将事务在执行时临时产生的数据存储在 DRAM 中,因为这些临时的数据不需要进行持久化,如果发生崩溃,不需要对它们进行恢复.

临时数据是在事务访问数据时产生的,当事务第 1 次访问某个数据时,会将该数据直接拷贝到 DRAM 中,再次访问该数据时,便可直接从快速 DRAM 中获取.在事务提交时,对于只读数据,可以直接释放其在 DRAM 上分配的空间;对于更新的数据,在释放 DRAM 空间前,需先将修改写回到 NVM 中.这种方法使得持久性事务内存的大部分时间都是在 DRAM 上运行的,从而降低了单个事务的执行时间,极大地提升了持久性事务内存的性能和扩展性.

4.2 基于双版本的只读事务优化

缓存行感知的双版本方法的优势不仅在于能够消除冗余的 NVM 写操作,同时还能用来提升只读事务的性能.和传统的多版本的组织方法不同,缓存行感知的双版本采用的是循环更新的方法,版本的新旧在每一次更新后都会发生变化.因此,对于一个只读事务来说,在访问数据时,需要先使用 *new* 来定位最新的版本,并优先对其进行访问.如果最新的版本不满足条件,再去访问较旧的版本.

在 2 种场景下,双版本是可以提升只读事务的性能的.1)被访问的数据没有被其他事务修改,且对最新版本的时间戳检查没有通过,这时访问较旧的版本能够降低只读事务的终止率.2)被访问的数据正在被其他事务修改,且对最新版本的时间戳检查没有通过,此时只读事务会终止并重试,防止因等待更新事务的完成而产生阻塞.但是有一点需要注意的是,在函数 *cmp_time* 中,被检查的时间戳必须是一个已经提交了事务的时间戳.这是因为,如果该事务还未完成,那么基于数据驱动的时钟获取方法会得到一个超前的时钟,使用该时钟去访问数据将会读取到事务的中间状态.因此,在检查数据的时间戳前,首先获取该版本的时间戳并记录,然后通过检查该版本是否被加锁来验证时间戳的有效性.

4.3 崩溃恢复机制

系统崩溃可能会导致系统在重启后处于不一致的状态,比如一个更新事务完成了对部分数据的修改,如果这时系统发生崩溃,在系统重启后便残留着这个没有完成的事务.缓存行感知的双版本崩溃恢复机制和基于日志的方法有很大不同.在基于日志的方法下,不管是 undo 日志还是 redo 日志,它们都将被修改的数据的地址在日志中进行了持久化.在崩溃后进行恢复时,便可以按照日志将数据恢复到更新事务开始前的状态或者重做日志以完成更新事务.缓存行感知的双版本方法没有使用日志,而是采用循环更新的方法直接覆盖较旧的版本,并且保留较新的版本以便在崩溃时将数据恢复到一致的状态.这里存在的问题是,在崩溃恢复时,将无法知道哪些数据在崩溃前被修改,即无法知道哪些数据是不一致的.

本文提出了 2 种缓存行感知的双版本崩溃恢复机制:1)基于全局扫描的方法;2)基于更新地址写回的方法.在基于全局扫描的方法下,在更新事务执行时不需要持久化被更新的数据的地址.在系统崩溃恢复时,由于不知道哪些数据是不一致的,因此需要全局扫描所有的数据.为了能够检测出不一致的数据,即那些被中断的更新事务修改过的数据,在修改数据前,需要将提交时间戳持久化到日志中;在事务完成后,再将该提交时间戳从日志中删除.这样,在进行全局扫面时,如果数据的时间戳等于日志中的时间戳,即表示该数据是不一致的,便可以使用另一个版本将其恢复到更新事务执行前的状态.该方法的优点是不需要持久化更新数据的地址,减少了运行时开销,但是由于需要全局扫描,其恢复时间较长.

基于更新地址写回的方法,在更新事务修改数据前,需要把被修改的数据的地址全部持久化到日志中.这样,在崩溃恢复时,通过日志中记录的地址便可以准确知道哪些数据在崩溃前被修改,从而避免了全局扫描的开销,做到快速恢复.显然,这种方法引入了一定的运行时开销.但是,由于仅仅持久化数据的地址,而不是数据本身,因此这种开销和传统的基于日志的方法相比是比较小的.在具体的应用场景下,可以根据需求的不同选择不同的崩溃恢复机制,默认情况下,本文选用了基于更新地址写回的方法.

4.4 缓存行感知的双版本的实现

缓存行感知的双版本也是使用 C++ 实现的,借助于 C++ 中的抽象类以及类模板,为用户提供了简单的编程接口.首先,缓存行感知的双版本为用户提供了 *AbstractPtmObject* 抽象类,该类主要定义了 2 个接口:*CopyToDram* 和 *WriteToNvm*.其中 *CopyToDram* 是将本地的数据拷贝到 DRAM 中;*WriteToNvm* 是把 DRAM 中的数据写回到 NVM 中.用户所有的类都应该继承该抽象类,并实现这 2 个函数.其次,在事务内存中,所有的读写操作都应该被截获.缓存行感知的双版本对用户的类进行了封装,提供了一个类模板 *PtmObjectWrapper*,如表 3 所示:

Table 3 The Main Members of *PtmObjectWrapper*

表 3 *PtmObjectWrapper* 类模板主要成员

类型	成员	描述
公有函数	*OpenWithRead*	读取数据
	OpenWithWrite	修改数据
uint8_t	*lock*|*new*	*lock* 和 *new* 占 1 B
T	*V*0	第 1 个版本,类型 T 由用户指定
	*V*1	第 2 个版本
uint64_t	*ts*0	*V*0 对应的时间戳
	*ts*1	*V*1 对应的时间戳

用户通过 *OpenWithRead* 和 *OpenWithWrite* 来对数据进行读写[5],本节将重点介绍这 2 个函数的实现,如算法 2 所示.

算法 2. 缓存行感知的双版本.

① Function *OpenWithRead*(*tx*)
② *saved_new*=*new*;
③ *saved_ts*=*saved_new*==0?*ts*0:*ts*1;
④ if *lock*≠*saved_new* && *time_cmp*(*tx*, *saved_ts*) then
⑤ *ret*=*saved_new*==0?*V*0.*CopyToDram*():*V*1.*CopyToDram*();
⑥ if *new*==*saved_new* && *lock*≠*saved_new* && (*saved_new*==0?*ts*0:*ts*1)==*saved_ts* then
⑦ return *ret*;
⑧ end if
⑨ end if
⑩ if *tx* is not READ_ONLY then
⑪ *sdtm_abort*(*tx*); /* 事务终止并重试 */
⑫ *saved_new*=*saved_new*==0?1:0;
⑬ *saved_ts*=*saved_new*==0?*ts*0:*ts*1;
⑭ if *lock*≠*saved_new* && *time_cmp*(*tx*, *saved_ts*) then
⑮ *ret*=*saved_new*==0?*V*0.*CopyToDram*():*V*1.*CopyToDram*();
⑯ if *lock*≠*saved_new* && (*saved_new*==0?*ts*0:*ts*1)==*saved_ts* then
⑰ return ret;
⑱ end if
⑲ end if
⑳ end if
㉑ Function *OpenWithWrite*(*tx*)
㉒ *saved_new*=*new*;
㉓ *saved_ts*=*saved_new*==0?*ts*0:*ts*1;
㉔ if *CAS*(*lock*,0,*saved_new*==0?2:1) && *time_cmp*(*tx*,*saved_ts*) then
㉕ *ret*=*saved_new*==0?*V*0.*CopyToDram*():*V*1.*CopyToDram*();
㉖ if *new*==*saved_new* && (*saved_new*==0?*ts*0:*ts*1)==*saved_ts* then
㉗ return ret;
㉘ end if
㉙ end if
㉚ *sdtm_abort*(*tx*). /* 事务终止并重试 */

OpenWithRead 在读取数据时会首先尝试较新的版本,当较新的版本不满足条件时再尝试较旧的,如果仍然不满足,便会终止并重试.在函数 *OpenWithRead* 中,当把数据拷贝到 DRAM 后,需要对数据的有效性再次进行检查,如算法 2 中的⑥⑯所示.这是因为在拷贝数据的过程中,数据可能已经被其他事务修改.和 *OpenWithRead* 不同的是,*OpenWithWrite* 需要获取锁,这里使用了 CAS 指令来执行加锁操作.需要注意的是,在获取锁后 *lock* 的值应该指向较旧的版本,这是因为根据缓存行感知的双版本的更新逻辑,较旧的版本会被直接覆盖.在获取锁成功并且时间戳检查通过后,调用 *CopyToDram* 来将较新的版本拷贝到 DRAM 中,以后在该事务中对于该数据的访问将直接重定向到 DRAM 中.

5 可扩展的持久性软件事务内存 SDTM

基于线程逻辑时钟和缓存行感知的双版本方法,我们实现了一个可扩展性的持久性软件事务内存 SDTM,其架构如图 8 所示.SDTM 的架构分为 2 层:在 NVM 中持久化数据;在 DRAM 中加速数据的访问,吸收对数据的读写操作.图 8 也展示了一个事务执行的过程,该事务读取对象 *A*,并修改对象 *B*.在读取数据时,依据 *new* 来判断哪个版本是最新的,然后将对应的版本拷贝到 DRAM 中.由于对 *B* 进行了修改,因此,在事务提交时需要将其写回到 NVM 中.可以发现,这里采用了就地更新的方法,直接覆盖了较旧的版本,消除了冗余的 NVM 写操作.

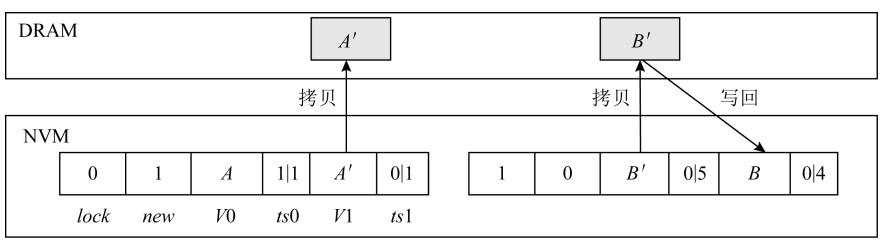


Fig. 8 The architecture of SDTM

图 8 SDTM 架构图

SDTM 集成了线程逻辑时钟和缓存行感知的双版本方法,并为用户提供了 4 个 API: *sdtm_start*, *sdtm_commit*, *sdtm_read*, *sdtm_write*,如算法 3 所示.用户使用 *sdtm_start* 开始一个事务,使用 *sdtm_commit* 来提交事务,需要注意的是,在函数 *sdtm_commit* 中,只有更新事务才需要提交,对于只读事务,由于没有进行任何修改,直接返回即可. *sdtm_abort* 用来终止并重试事务,它释放 *write_set* 的锁,并跳转到事务开始的地方进行重试,用户不需要自己调用该函数.

算法 3. SDTM 算法.

① Function *sdtm_start*(*tx*)
② *get_time*(*tx*);
③ clear *read_set*; /* 清空 *read_set* */
④ clear *write_set*; /* 清空 *write_set* */
⑤ Function *sdtm_commit*(*tx*)
⑥ if *tx* is not READ_ONLY then
⑦ if *tx.read_set* is not valid then
⑧ *sdtm_abort*();
⑨ end if
⑩ *ct*=*new_time*(*tx*);
⑪ for each item in *tx.write_set* do
⑫ *item.copy.WriteToNvm*(*item.wrapperAddr*, *ct*);
⑬ end for
⑭ release all lock in *tx.write_set*;
⑮ end if
⑯ Function *sdtm_read*(*tx*, *wrapperAddr*)
⑰ *ret*=NULL;
⑱ if (*ret*=*tx.write_set.Find*(*wrapperAddr*)) ‖ (*ret*=*tx.read_set.Find*(*wrapperAddr*)) then
⑲ return *ret*;
⑳ end if
㉑ *ret*=*wrapperAddr*→*OpenWithRead*();
㉒ *tx.read_set.Insert*(*wrapperAddr*, *ret*);
㉓ return *ret*;
㉔ Function *sdtm_write*(*tx*, *wrapperAddr*)
㉕ *ret*=NULL;
㉖ if (*ret*=*tx.write_set.Find*(*wrapperAddr*)) then
㉗ return *ret*;
㉘ end if
㉙ *ret*=*wrapperAddr*→*OpenWithWrite*();
㉚ *tx.write_set.Insert*(*wrapperAddr*, *ret*);
㉛ return *ret*;
㉜ Function *sdtm_abort*(*tx*)
/* 用户不可直接调用 */
㉝ 释放 *tx.write_set* 所持有的锁;
㉞ 跳转到 *sdtm_start*.

sdtm_read 用来读取数据,在真正地访问数据前,首先检查在 *write_set* 和 *read_set* 中是否存在该数据的拷贝.如果存在,表明在之前访问过该数据,且在 DRAM 中存在一个拷贝,这样可以直接返回该拷贝的地址,避免访问低速的 NVM.这里,首先检查的是 *write_set*,这是因为如果在这个事务中该数据被修改,那么被修改后的数据(也即是最新的数据)一定在 *write_set* 中.如果在 DRAM 中找不到该数据,则通过函数 *OpenWithRead* 在 DRAM 中生成一个拷贝,并插入到 *read_set* 中,然后返回该拷贝的地址. *sdtm_write* 和 *sdtm_read* 相似,只是 *sdtm_write* 仅仅在 *write_set* 中查找是否有该数据的拷贝,并且调用的是函数 *OpenWithWrite*.

6 实验与结果

6.1 实验环境与内容

本文所做的测试都是在真实的 NVM 硬件环境下进行的,实验设备的配置参数如表 4 所示.其中,服务器有 2 个 NUMA 节点,每个节点有 36 个核,并使用了超线程,并装备了 192 GB 的 DRAM 和 1.5 TB 的 Optane DCPMM. Optane DCPMM 共有 3 种配置模式: Memory Mode, App Direct Mode, Mixed Mode.由于 Memory Mode 不保证持久性,因此这里使用的是 App Direct Mode 配置,并使用 PMDK 来对其空间的分配与回收进行管理.本文使用了 TINYSTM[24] 中的测试框架,每个测试的运行时间为 10 s,并连续运行 5 次然后取平均值.在测试中发现,libc 自带的内存分配器 ptmalloc[49] 扩展性较差,为了消除内存分配器对测试的影响,本文使用了在多线程下具有更高性能的 jemalloc[50].

Table 4 The Configuration of Server
表 4 服务器配置信息

硬件	型号
CPU	Intel® Xeon® Gold 6240 CPU@2.60 GHz
内存	192 GB, 24 MB LLC
NVM	Intel Optane DC Persistent Memory Module(DCPMM)
操作系统	Ubuntu 16.04(kernel version 4.15.0)
内存分配器	libvmem, jemalloc

测试共分为 2 个部分,分别是并发数据结构测试和真实负载测试.在并发数据结构的测试中,本文基于 SDTM 实现了 3 种并发且持久性的数据结构,包括散列表、二叉搜索树、有序链表,并使用了不同比例的读写负载对其扩展性进行了测试.之所以选择 3 种不同的数据结构,是因为这些数据结构的竞争情况不同.散列表的冲突较少,因此竞争较小;而有序链表的冲突较多,因此竞争最大.这样,可以测试出 SDTM 在不同竞争情况下的性能与扩展性.在真实负载的测试中,本文基于 SDTM 实现了一个 B^+ 树,并使用 YSCB[51] 提供的典型负载对其进行了详细的测试,以此来验证 SDTM 在真实负载下的效果.

另外,为了验证 SDTM 的先进性,本文将 SDTM 和最新的研究进行了对比,包括 DudeTM 和 PMDK.在 DudeTM 的具体实现中,其既可以基于软件事务内存,也可以基于硬件事务内存,由于 SDTM 是完全基于软件的,这里为了公平性,进行对比的也是基于软件事务内存的 DuteTM.另外,按照对持久性的要求,DudeTM 又分为同步的 DudeTM 和异步的 DudeTM,异步的 DudeTM 在事务完成后不会立即将数据写回到 NVM,而是在累积一定数据量的日志后再一起写回,这可以带来性能的提升,但是放松了对持久性的要求,本文进行对比测试的是异步的 DudeTM.由于 PMDK 没有对多线程进行同步,按照通用的做法,本文使用了读写锁来保证并发的正确性.

6.2 并发持久性数据结构测试

在本节中,基于 SDTM,DudeTM,PMDK 分别实现了 3 种并发且持久性的数据结构:散列表、二叉搜索树和有序链表,并使用读为主(2%更新)、读密集(20%更新)和写密集(80%更新)3 种负载对这 3 种数据结构进行测试.另外,为了进一步对测试结果进行分析,本文还统计了对应的事务终止率,其中事务终止率指的是被终止的事务占提交事务的比例,由于一个事务可能会被多次终止并重试,因此终止率是可以大于 100%的.

1) 散列表

本实验将散列表的初始大小设为 1 万个桶,并使用链表解决冲突.对于所有测试,预先往散列表中插入 1 万个键值对,再执行工作负载.由于散列表被设置的较大,并且现有的散列函数能很好地将数据均匀分布,因此发生冲突的概率很小,具有很好的扩展性.

图 9 是测试结果,图 9(a)~9(c)分别对应读为主、读密集和写密集 3 种负载,图 9(d)~9(f)是对应

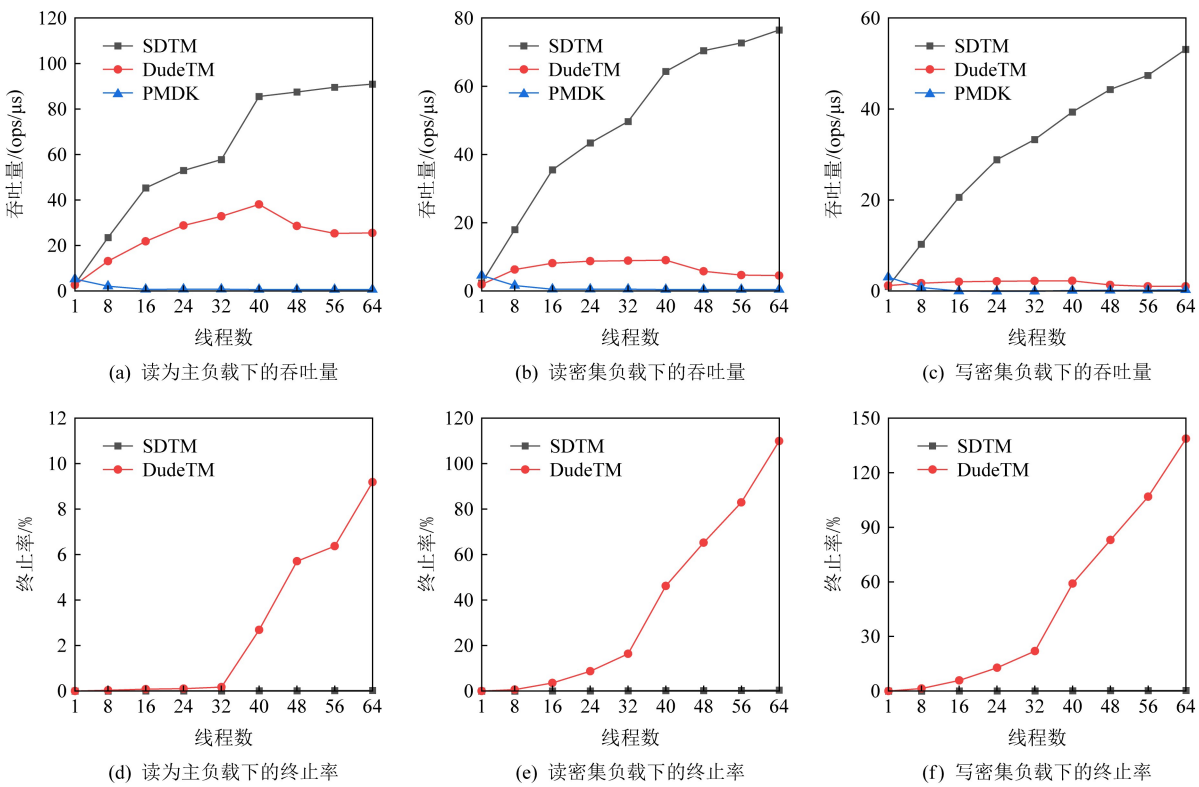


Fig. 9 The throughput and abort rate of Hash table

图 9 散列表的吞吐量以及事务终止

的事务终止率.可以看到,不管是基于哪种负载,SDTM 都拥有最好的扩展性,且随着更新比例的增加,其他事务内存和 SDTM 的性能差距越来越大.另外,从事务的终止率中可以看到,SDTM 的终止率一直维持在一个很低的水平.这主要有 2 个原因:①线程逻辑时钟和缓存行感知的双版本方法加速了单个事务的执行时间,降低了事务冲突的概率;②双版本减少了更新事务对只读事务的阻塞,降低了只读事务的失败重试次数.由于 PMDK 使用读写锁来进行并发控制,不会存在事务终止的情况,因此图 9 中没有显示它的终止率.

2) 二叉搜索树

二叉搜索树和散列表相比存在较多的竞争,这是因为对于树的访问,都是从根节点开始的,如果中间节点发生改变,那么路过该中间节点的事务都会受到影响.同样,在测试前,在二叉搜索树中插入 1 万个键值对来进行初始化,然后分别运行这 3 种不同类型的负载,结果如图 10 所示:

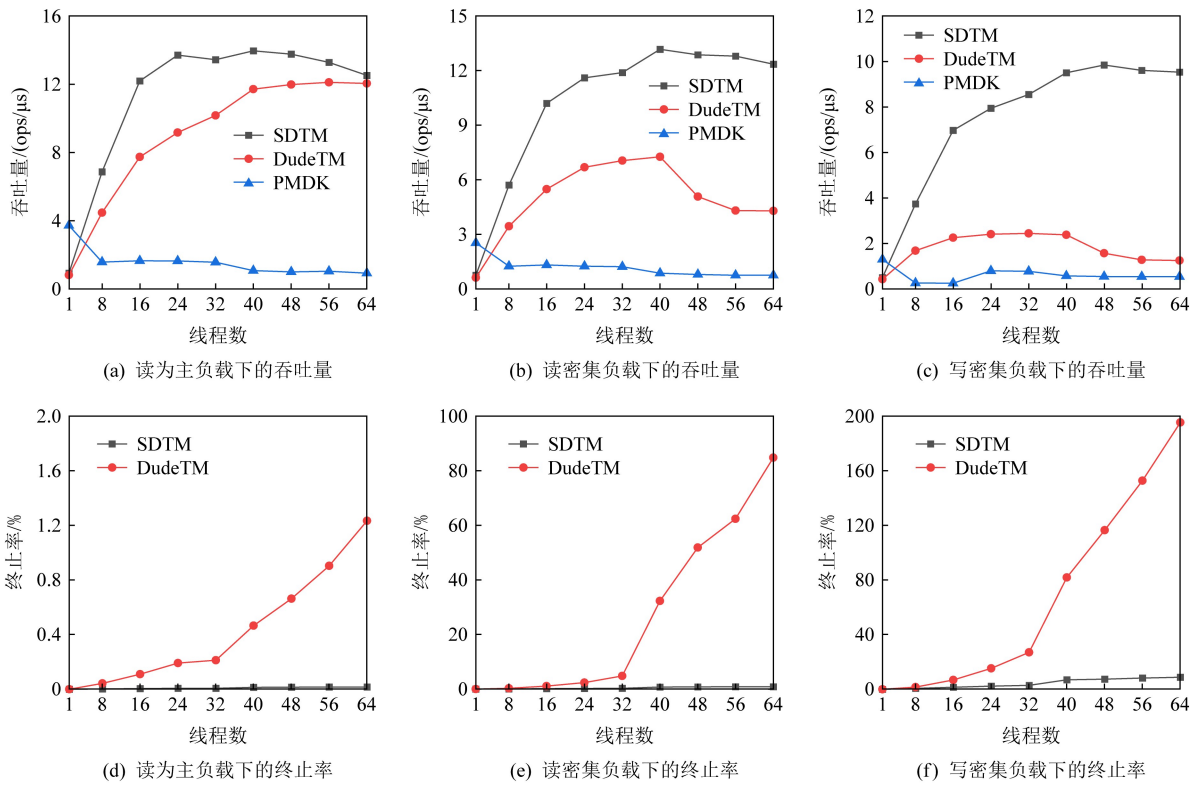


Fig. 10 The throughput and abort rate of binary search tree

图 10 二叉搜索树的吞吐量以及事务终止率

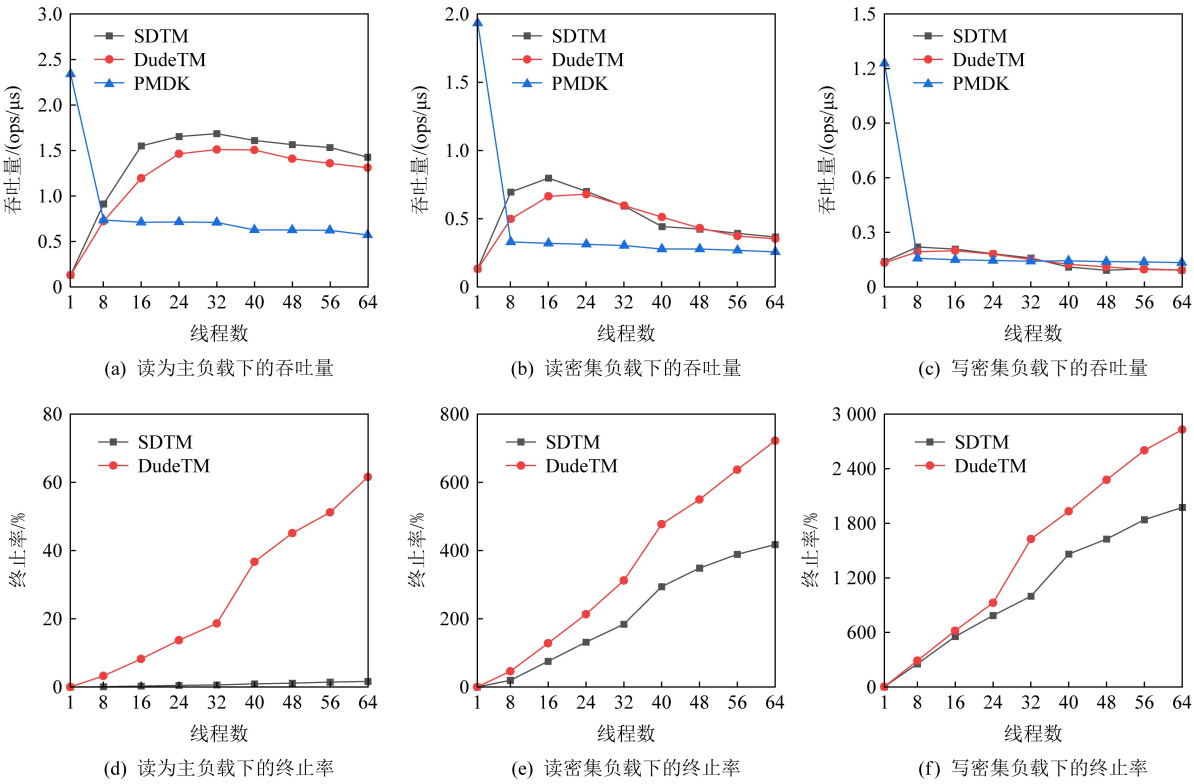
从图 10 中可以发现,不管是在哪种负载下,SDTM 都拥有最好的扩展性,DudeTM 和 PMDK 的性能随着线程数的增加迅速下降,这主要是受到了全局逻辑时钟以及 undo 日志的影响,而 SDTM 依然保持着很好的性能.在写密集型的负载下,SDTM 的吞吐量相对于 DudeTM 和 PMDK 最多可以分别达到 7.6 倍和 17.5 倍.从事务终止率来看,由于使用了双版本来提升只读事务的性能,即使是在写密集的负载下,SDTM 依然保持着较低的终止率,而 DudeTM 的终止率随着线程数量的增加会快速升高.

3) 有序链表

在有序链表中,对于数据的访问只能从链表的头部开始,依次向后遍历,即只有一条访问路径.根据事务的执行过程,这种访问数据的模式存在大量的竞争,将会导致大量的事务终止并重试.比如,更新事务在提交时,需要检查它所读取的数据是否发生修改,对于有序链表来说,要校验从链表头到这个被修改的数据之间的节点是否发生改变,当线程数量较多时,这种校验极大可能会失败,从而导致事务终止并重试.因此,有序链表本身是一个扩展性较差的数据结构.为避免因链表太长使得事务大部分执行时间花费在链表遍历上,难以对不同事务内存的扩展性以及性能进行比较,此处测试使用了 256 个键值对来初始化有序链表.

图 11 展示了在不同负载下不同事务内存的扩

展性以及终止率的测试结果.有意思的是,当只有一个线程时,PMDK 拥有最好的性能.这是因为 PMDK 采用的是 undo 日志,当以只读的方式访问有序链表中的某个节点时,可以直接对其进行读取,不需要像 SDTM 和 DudeTM 那样将数据拷贝到另一个地方(DRAM),这种内存拷贝会占据大量的开销.但是,随着线程数量的增加,PMDK 中的读写锁成为了扩展性的瓶颈,因为即使对于只读事务,也需要加锁.因此,在读为主和读密集的负载下,SDTM 总体上保持着最好的扩展性,但是在写密集的负载下,PMDK 展现了最好的性能.这是因为,从对应的终止率中可以看出,SDTM 和 DudeTM 在写密集负载下的终止率较高,这时采用读写锁来进行并发控制往往能够收获更好的性能.



(a) 读为主负载下的吞吐量 (b) 读密集负载下的吞吐量 (c) 写密集负载下的吞吐量

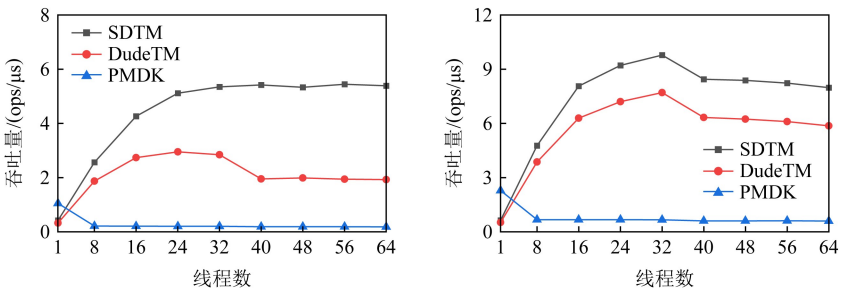
(d) 读为主负载下的终止率 (e) 读密集负载下的终止率 (f) 写密集负载下的终止率

Fig. 11 The throughput and abort rate of sorted list

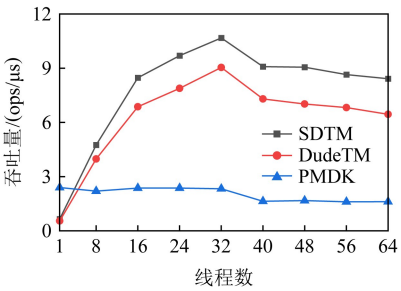
图 11 有序链表的吞吐量以及事务终止率

6.3 真实负载测试

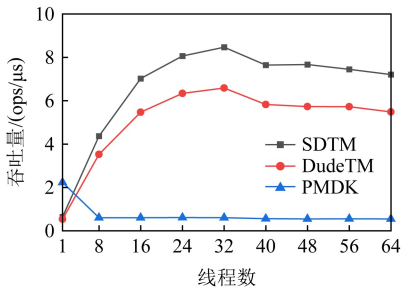
为了验证 SDTM 在真实负载下的效果,基于 SDTM,本文实现了一个并发且持久性的 B^+ 树,并使用 YCSB 性能测试工具集生成 $A\sim D$ 共 4 个工作负载,请求的分布模式为倾斜分布.其中,load 操作的数量为 100 万,run 操作的数量为 1 000 万.图 12 展示了 4 种 YCSB 负载下不同持久性事务内存实现的 B^+ 树的性能.从测试结果来看,不管是基于哪种负载,SDTM 都拥有最好的扩展性与性能,最高时 SDTM 的吞吐量是 DudeTM 的 2.8 倍,是 PMDK 的 29 倍.注意,在负载 B,C,D 中,当线程数由 32 增加到 40 时,可以看到 SDTM 和 DudeTM 的



(a) 负载A下的吞吐量(读:50%) (b) 负载B下的吞吐量(读:95%)



(c) 负载C下的吞吐量(读:100%)



(d) 负载D下的吞吐量(读:100%)

Fig. 12 The throughput of SDTM under YCSB workloads

图 12 YCSB 负载下 SDTM 的吞吐量

性能有明显的下降,这是因为存在跨越 NUMA 节点的访问,其开销影响了事务内存的性能.PMDK 的性能没有太大变化,是因为其性能主要受到读写锁开销的制约.

7 总 结

现有的持久性软件事务内存扩展性较差,本文测试并分析了制约扩展性的因素,发现中心化的全局逻辑时钟和冗余的 NVM 写操作严重影响了扩展性.针对这 2 个问题,分别设计并实现了线程逻辑时钟和缓存行感知的双版本方法,消除了制约扩展性的因素.基于这 2 种方法,设计并实现了一个高扩展性的持久性软件事务内存 SDTM,并在真实的 NVM 器件上使用 YCSB 工作负载进行测试.结果显示,相比于现有的持久性事务内存 DudeTM 和 PMDK,SDTM 的性能最高分别提升了 2.8 倍和 29 倍.

作者贡献声明:刘超杰负责编写代码、做实验、撰写论文;王芳负责修改和指导论文;邹晓敏负责修改论文;冯丹负责指导论文.

参 考 文 献

[1] Roy A, Xu Jingye, Chowdhury M H. Multi-core processors: A new way forward and challenges [C] //Proc of the 26th Int Conf on Microelectronics. Piscataway, NJ: IEEE, 2008: 454-457

[2] Fielden J. Semiconductor inspection and metrology challenges [C/OL] //Proc of the 31st Int Vacuum Nanoelectronics Conf (IVNC). Piscataway, NJ: IEEE, 2018 [2021-09-11]. https://ieeexplore.ieee.org/document/8520121

[3] Ross P E. Why CPU frequency stalled [J]. IEEE Spectrum, 2008, 45(4): 72-72

[4] Herlihy M, Moss J E B. Transactional memory: Architectural support for lock-free data structures [C] //Proc of the 20th Annual Int Symp on Computer Architecture. New York: ACM, 1993: 289-300

[5] 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, 2003: 92-101

[6] Marathe V J, Scherer W N, Scott M L. Adaptive software transactional memory [C] //Proc of the 15th Int Symp on Distributed Computing. Berlin: Springer, 2005: 354-368

[7] Saha B, Adl-Tabatabai A R, Hudson R L, et al. McRT-STM: A high performance software transactional memory system for a multi-core runtime [C] //Proc of the 17th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2006: 187-197

[8] Rajwar R, Herlihy M, Lai K. Virtualizing transactional memory [C] //Proc of the 32nd Int Symp on Computer Architecture(ISCA'05). Piscataway, NJ: IEEE, 2005: 494-505

[9] Hammond L, Wong V, Chen M, et al. Transactional memory coherence and consistency [J]. ACM SIGARCH Computer Architecture News, 2004, 32(2): 102-113

[10] Damron P, Fedorova A, Lev Y, et al. Hybrid transactional memory [C] //Proc of the 12th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2006: 336-346

[11] Calciu I, Gottschlich J, Shpeisman T, et al. Invyswell: A hybrid transactional memory for Haswell's restricted transactional memory [C] //Proc of the 23rd Int Conf on Parallel Architecture and Compilation Techniques (PACT). Piscataway, NJ: IEEE, 2014: 187-199

[12] Hady F T, Foong A, Veal B, et al. Platform storage performance with 3D XPoint technology [J]. Proceedings of the IEEE, 2017, 105(9): 1822-1833

[13] Wong H S P, Raoux S, Kim S B, et al. Phase change memory [J]. Proceedings of the IEEE, 2010, 98(12): 2201-2227

[14] Kültürsay E, Kandemir M, Sivasubramaniam A, et al. Evaluating STT-RAM as an energy-efficient main memory alternative [C] //Proc of the 2013 IEEE Int Symp on Performance Analysis of Systems and Software (ISPASS). Piscataway, NJ: IEEE, 2013: 256-267

[15] Pan Feng, Gao Shuang, Chen Chao, et al. Recent progress in resistive random access memories: Materials, switching mechanisms, and performance [J]. Materials Science and Engineering: R: Reports, 2014, 83: 1-59

[16] Boukhobza J, Rubini S, Chen R, et al. Emerging NVM: A survey on architectural integration and research challenges [J]. ACM Transactions on Design Automation of Electronic Systems, 2017, 23(2): 1-32

[17] Bez R, Pirovano A. Non-volatile memory technologies: Emerging concepts and new materials [J]. Materials Science in Semiconductor Processing, 2004, 7(4/5/6): 349-355

[18] Chen An. A review of emerging non-volatile memory(NVM) technologies and applications [J]. Solid-State Electronics, 2016, 125: 25-38

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

[20] Gu Jinyu, Yu Qianqian, Wang Xiayang, et al. Pisces: A scalable and efficient persistent transactional memory [C] // Proc of the 30th Annual Technical Conf (ATC). Berkeley, CA: USENIX Association, 2019: 913-928

[21] Wang Zhaoguo, Yi Han, Liu Ran, et al. Persistent transactional memory [J]. IEEE Computer Architecture Letters, 2014, 14(1): 58-61

[22] Liu Mengxing, Zhang Mingxing, Chen Kang, et al. DudeTM: Building durable transactions with decoupling for persistent memory [J]. ACM SIGPLAN Notices, 2017, 52 (4): 329-343

[23] Scargall S. Programming Persistent Memory [M]. Berlin: Springer, 2020: 63-72

[24] Felber P, Fetzer C, Marlier P, et al. Time-based software transactional memory [J]. IEEE Transactions on Parallel and Distributed Systems, 2010, 21(12): 1793-1807

[25] Dice D, Shalev O, Shavit N. Transactional locking II [C] // Proc of the 20th Int Symp on Distributed Computing. Berlin: Springer, 2006: 194-208

[26] Krishnan R M, Kim J, Mathew A, et al. Durable transactional memory can scale with timestone [C] //Proc of the 25th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2020: 335-349

[27] Memaripour A, Badam A, Phanishayee A, et al. Atomic in-place updates for non-volatile main memories with kamino-tx [C] //Proc of the 12th European Conf on Computer Systems. New York: ACM, 2017: 499-512

[28] Correia A, Felber P, Ramalhete P. Romulus: Efficient algorithms for persistent transactional memory [C] //Proc of the 30th on Symp on Parallelism in Algorithms and Architectures. New York: ACM, 2018: 271-282

[29] Yu Wenqian. Optimization and implementation of distributed transaction based on multi-version concurrency control [D]. Shanghai: East China Normal University, 2020 (in Chinese) (俞文谦. 基于多版本并发控制的分布式事务优化与实现 [D]. 上海: 华东师范大学, 2020)

[30] Lee K S, Wang Han, Shrivastav V, et al. Globally synchronized time via datacenter networks [C] //Proc of the 22nd ACM SIGCOMM Conf. New York: ACM, 2016: 454-467

[31] Kashyap S, Min C, Kim K, et al. A scalable ordering primitive for multicore machines [C/OL] //Proc of the 13th EuroSys Conf. New York: ACM, 2018 [2021-10-09]. https://dl.acm.org/doi/10.1145/3190508.3190510

[32] Riegel T, Fetzer C, Felber P. Time-based transactional memory with scalable time bases [C] //Proc of the 19th Annual ACM Symp on Parallel Algorithms and Architectures. New York: ACM, 2007: 221-228

[33] Lameter C. NUMA (non-uniform memory access): An overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors [J]. Queue, 2013, 11(7): 40-51

[34] Boyd-Wickizer S, Clements A T, Mao Yandong, et al. An analysis of Linux scalability to many cores [C] //Proc of the 9th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2010: 86-93

[35] Schweizer H, Besta M, Hoefler T. Evaluating the cost of atomic operations on modern architectures [C] //Proc of the 24th Int Conf on Parallel Architecture and Compilation Techniques (PACT). Piscataway, NJ: IEEE, 2015: 445-456

[36] Intel. Intel 64 and IA-32 architectures software developer's manual, volume 3B: System programming guide [EB/OL]. 2016-09 [2021-08-15]. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html

[37] Fedotova I, Siemens E, Hu Hao. A high-precision time handling library [J]. International Journal of Computers Communications & Control, 2013, 10: 1076-1086

[38] Dice D, Lev Y, Moir M, et al. Early experience with a commercial hardware transactional memory implementation [C] //Proc of the 14th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2009: 157-168

[39] Ritson C G, Barnes F R M. An evaluation of Intel's restricted transactional memory for CPAs [J/OL]. Communicating Process Architectures 2013, 2013: 271-291 [2021-09-11]. https://kar.kent.ac.uk/36939/

[40] Zhao Jishen, Li Sheng, Yoon D H, et al. Kiln: Closing the performance gap between systems with and without persistence support [C] //Proc of the 46th Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2013: 421-432

[41] Pelley S, Chen P M, Wenisch T F. Memory persistency [C] //Proc of the 41st Int Symp on Computer Architecture (ISCA). Piscataway, NJ: IEEE, 2014: 265-276

[42] Riegel T, Fetzer C, Felber P. Snapshot isolation for software transactional memory [C/OL] //Proc of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing. New York: ACM, 2006 [2021-10-09]. http://citeseerx. ist. psu. edu/viewdoc/download?doi=10.1.1.113.7870&rep=rep1&type=pdf

[43] Tu S, Zheng Wenting, Kohler E, et al. Speedy transactions in multicore in-memory databases [C] //Proc of the 24th ACM Symp on Operating Systems Principles. New York: ACM, 2013: 18-32

[44] Yu Xiangyao, 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

[45] Yu X, Pavlo A, Sanchez D, et al. TicToc: Time traveling optimistic concurrency control [C] //Proc of the 43rd Int Conf on Management of Data. New York: ACM, 2016: 1629-1642

[46] 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

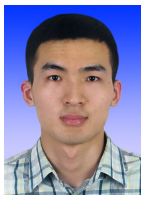
[47] Zhang Rui, Budimlić Z, Scherer III W N. Commit phase in timestamp-based STM [C] //Proc of the 12th Annual Symp on Parallelism in Algorithms and Architectures. New York: ACM, 2008: 326-335

[48] Yang Jian, Kim J, Hoseinzadeh M, et al. An empirical guide to the behavior and use of scalable persistent memory [C] //Proc of the 18th Conf on File and Storage Technologies (FAST). Berkeley, CA: USENIX Association, 2020: 169-182

[49] Liu Xiang, Tong Wei, Liu Jingning, et al. A review of dynamic memory allocator research [J]. Chinese Journal of Computers, 2018, 41(10): 2359-2378 (in Chinese)
(刘翔, 童薇, 刘景宁, 等. 动态内存分配器研究综述[J]. 计算机学报, 2018, 41(10): 2359-2378)

[50] Elias D, Matias R, Fernandes M, et al. Experimental and theoretical analyses of memory allocation algorithms [C] //Proc of the 29th Annual ACM Symp on Applied Computing. New York: ACM, 2014: 1545-1546

[51] Cooper B F, Silberstein A, Tam E, et al. Benchmarking cloud serving systems with YCSB [C] //Proc of the 1st ACM Symp on Cloud Computing. New York: ACM, 2010: 143-154



Liu Chaojie, born in 1994. Master. His main research interests include non-volatile memory, concurrency control, transactional memory.

刘超杰,1994 年生.硕士.主要研究方向为非易失性内存、并发控制、事务性内存.

Wang Fang, born in 1972. Professor at Huazhong University of Science and Technology. Her main research interests include network storage, parallel storage systems, parallel file systems, new storage systems based on non-volatile storage devices, storage and processing of massive graph data.

王　芳,1972 年生.教授.主要研究方向为网络存储、并行存储系统、并行文件系统、基于非易失存储器件的新型存储系统、海量图数据存储与处理.

Zou Xiaomin, born in 1994. PhD candidate at Huazhong University of Science and Technology. Her main research interests include non-volatile memory (NVM) and key-value store.

邹晓敏,1994 年生.博士研究生.主要研究方向为非易失性存储器(NVM)和键值存储.

Feng Dan, born in 1970. Professor at Huazhong University of Science and Technology. Her main research interests include computer system structure, big data storage system, non-volatile storage technology, storage and computing fusion technology.

冯　丹,1970 年生.教授.主要研究方向为计算机系统结构、大数据存储系统、非易失存储技术、存算融合技术.