软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(3):849−866 [doi: 10.13328/j.cnki.jos.006444]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于 NVM 和 HTM 的低时延事务处理*

魏星达, 陆放明, 陈 榕, 陈海波, 臧斌宇

(上海交通大学 并行与分布式系统研究所, 上海 200240)

通信作者: 陈榕, E-mail: rongchen@sjtu.edu.cn

摘 要: 硬件事务内存(hardware transactional memory, HTM)能够极大地提升多核内存事务处理的吞吐. 然而, 为了避免慢速持久化设备对事务吞吐的影响, 现有系统以批量的方式提交事务, 这使得事务提交有极高的延迟. 低时延非易失性内存(non-volatile memory, NVM)的出现, 给降低基于 HTM 的内存事务处理时延带来了机遇; 然而, 利用 NVM 需要解决 HTM 无法和 NVM 硬件协同的挑战: 持久化写入 NVM 会直接中断 HTM 的执行. 为了解决这一问题, 提出了名为 Parity Version 的机制, 将事务中的 NVM 操作和 HTM 操作进行区分. 这样, 事务可以正确且高效地利用 NVM 降低基于 HTM 事务处理的时延. 将该机制集成到现有基于 HTM 的内存数据库、DBX 中, 并提出了 DBXN: 一个低时延高吞吐的内存数据库. DBXN还针对真实 NVM硬件的特性对事务实现进行了相应的优化. 在典型事务处理测试基准 TPC-C 中, DBXN 能够将 DBX 的事务提交时延降低 99%, 同时还有 2.1 倍更高的吞吐.

关键词: 非易失性内存; 内存事务处理; 硬件事务内存

中图法分类号: TP316

中文引用格式: 魏星达, 陆放明, 陈榕, 陈海波, 臧斌宇. 基于 NVM 和 HTM 的低时延事务处理. 软件学报, 2022, 33(3): 849–866. http://www.jos.org.cn/1000-9825/6444.htm

英文引用格式: Wei XD, Lu FM, Chen R, Chen HB, Zang BY. Low-latency Transaction Processing Using NVM and HTM. Ruan Jian Xue Bao/Journal of Software, 2022, 33(3): 849–866 (in Chinese). http://www.jos.org.cn/1000-9825/6444.htm

Low-latency Transaction Processing Using NVM and HTM

WEI Xing-Da, LU Fang-Ming, CHEN Rong, CHEN Hai-Bo, ZANG Bin-Yu

(Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai 200240, China)

Abstract: The emergency of hardware transactional memory (HTM) has greatly boosted the transaction processing throughput in in-memory databases. However, the group commit used in in-memory databases, which aims at reducing the impact from slow persistent storage, leads to high transaction commit latency. Non-volatile memory (NVM) opens opportunities for reducing transaction commit latency. However, HTM cannot cooperate with NVM together: flushing data to NVM will always cause HTM to abort. This study proposes a technique called Parity Version to decouple the process of HTM execution and NVM write. Thus, the transactions can correctly and efficiently use NVM to reduce their commit latency with HTM. This technique has been integrated to DBX, a state-of-the-art HTM-based database, and DBXN: A low-latency and high-throughput in-memory transaction processing system, is proposed. Evaluations using typical OLTP workloads, including TPC-C, show that it has 99% lower latency and 2.1 times higher throughput than DBX.

Key words: non-volatile memory; in-memory transaction; hardware transactional memory

以 12306 订票系统为代表的现代数据库应用, 都需要数据库事务来正确并可靠地处理数据. 随着信息化技术的普及, 数据库应用承载的流量越来越大, 人们愈发需要高性能的事务处理系统. 近年来, 多核处理器的

* 基金项目: 国家重点研发计划(2020YFB2104100); 国家杰出青年科学基金(61925206)
本文由“数据库系统新型技术”专题特约编辑李国良教授、于戈教授、杨俊教授和范举教授推荐.
收稿时间: 2021-06-30; 修改时间: 2021-07-31; 采用时间: 2021-09-13; jos 在线出版时间: 2021-10-21

兴起和内存容量的扩大，催生了一批高吞吐的多核内存数据库[1−6]. 同时，硬件事务性内存(hardware transactional memory, HTM)这一新处理器的特性，又进一步地将多核内存数据库的事务吞吐能力推向了一个新的高峰[1,7,8].

尽管现有基于HTM的多核内存数据库能够提供极高的吞吐，这些系统仍有较高的事务提交时延. 为了避免事务受到较慢的存储设备的影响，它们均采用批量提交的方式确保事务的持久性[1,3,4]. 然而，批量提交的方式会给内存事务处理带来一个数量级别的时延增高. 例如，在DBX[1]这一典型基于HTM的多核内存数据库中，其持久化和非持久化事务的提交时延相差近 18 000 倍(9 μs 对比 160 ms，详见第 2.2 节). 低时延和高吞吐对事务处理同样重要. 例如，电子商务巨头亚马逊(Amazon)曾总结过，每 100 ms 请求时延的增加，会带来 1%的经济效益损失[9].

Intel在近期终于推出了第一款商用的NVM设备，3D-XPoint[10,11]. 当有了NVM这种低时延的持久化设备后，本文期望回答一个问题：我们能否利用它显著地降低基于HTM的多核内存数据库的事务提交时延，同时保留HTM所带来的性能优势？除了高性能以外，NVM还提供了和DRAM一样的接口，这意味着系统可以在HTM中直接读写NVM. 因此，同时结合这两款硬件特性加速内存数据库就成为了可能.

在探索利用NVM降低基于HTM的事务提交时延的过程中，我们遇到了两个关键挑战(第 2.3 节).

- 首先，写入NVM无法和HTM相协作：该操作会百分百中断HTM的执行.
- 其次，真实NVM性能特征完全不同于内存，并很难被现有模拟器复现[12,13].

因此，完全利用NVM的高性能需要在事务实现中进行针对性设计.

由于真实NVM硬件近两年才出现，现有针对HTM和NVM设计的数据库并没有完全考虑其硬件特性. 例如，PHyTM[14]假设HTM中可以写入NVM，而真实硬件不能. 另一方面，现有基于NVM的事务性持久内存系统未专门针对数据库场景进行设计，他们主要关注于如何提供通用的事务编程接口[15−17]. 因此，这些系统在数据库场景中无法获得最好的性能(见第 4.3 节).

为了解决HTM和NVM无法协作的问题，我们提出了一个名为parity version的机制(见第 3.1.2 节). 该机制从软件层面将事务处理中NVM的操作和HTM的操作进行了分离，事务从而可以同时利用这两个硬件的特性进行加速. 同时，由于parity version可以复用现有基于HTM的事务处理的并发控制机制去实现(见第 3.1.3 节)，它只会给事务执行带来微小的性能影响. 最后，我们还针对真实HTM和NVM硬件的特性进行了一系列实现上的优化(见第 3.4 节).

我们将上述机制应用在了DBX[1]，一个现有典型基于HTM的内存数据库中，并提出了DBXN，一个同时利用HTM和NVM特性的内存事务处理系统. DBXN能同时满足高吞吐和低时延的事务处理需求：一方面，在典型数据库事务场景TPC-C[18]中，DBXN将DBX的时延降低了99%，同时还有2.1倍更高的吞吐；另一方面，DBXN在数据库场景中能比现有基于真实NVM硬件的事务性持久内存系统(Pisces[15])具有65%更好的吞吐.

综上所述，本文做出如下贡献：

(1) 系统化分析了如何在HTM中使用NVM(见第 2 节);
(2) 提出了一个对HTM和NVM友好的事务处理方法(见第 3 节);
(3) 提出了DBXN，一个结合HTM和NVM特性的高吞吐低时延多核内存数据库(见第 3 节);
(4) 在典型事务处理基准TPC-C中，DBXN能够大幅度降低现有基于HTM的多核内存数据库的事务提交时延，并提升事务吞吐(见第 4 节).

1 背景知识

1.1 硬件事务性内存(hardware transactional memory, HTM)

确保并发程序内存读写的原子性(atomicity)、一致性(consistency)和隔离性(isolation)，需要开发人员手动进行并发控制，例如采用细粒度锁等机制. 手动进行并发控制不仅容易出错，各种并发控制方法还会带来额外的软件开销. HTM[19]通过在硬件层确保程序并发内存读写的ACI特性，极大地简化了开发人员并发程序的

编写, 并避免了软件并发控制的开销. 随着 Intel 提出了名为 RTM (restricted transactional memory) (由于本文只关注 RTM 这一主流的 HTM 实现, 因此除非直接说明, 文中的 HTM 指的就是 RTM)的处理器新特性, 使用 HTM 加速并发程序变成了现实. 具体来说, RTM 提供了全新的指令集: xbegin, xend 和 xabort. 其中, 程序使用 xbegin 和 xend 表明 HTM 的开始和结束, 使用 xabort 中断 HTM 的执行. RTM 采用了基于乐观并发控制方法[20]的实现: 处理器利用其缓存(cache)记录程序内存的读写集, 并借助缓存一致机制(cache coherance)来检测冲突的内存访问.

由于硬件资源有限, HTM 有如下限制[1].

- 首先, 其执行的指令集有限. 例如, HTM 中程序无法进行系统调用.
- 其次, 其保护的程序的内存读写集有限: 当前实现使用 L1 缓存记录程序的读写集合、使用 L2 缓存记录读集合, 当程序读写内存的大小超过缓存大小时, HTM 就会中断.

1.2 基于HTM的多核内存数据库

观察到 HTM 所提供的高性能 ACI 性质和多核内存数据库 ACID 事务处理中的 ACI 性质违和, 人们开始广泛探索如何利用它来提升事务处理的性能[1,7,8,21]. 图 1 展示了一个使用 HTM 加速事务的例子, 在该例子中, 事务将数据库表 A 里所有 x 列值大于 10 的数据更新为 10. 如左半部分所示, 系统通过将该事务逻辑包含在 xbegin 和 xend 内, 即可利用 HTM 确保事务的 ACI 性质.

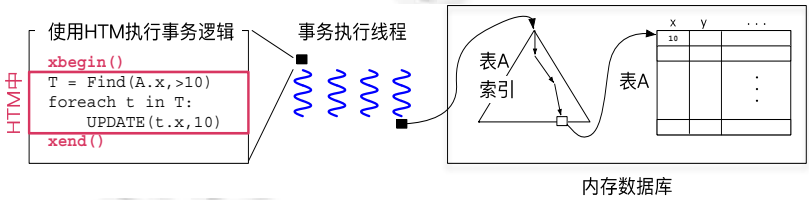


图 1 在多核内存数据库中使用 HTM 加速事务执行

尽管 HTM 能够高效确保 ACI 性质, 由于数据库事务逻辑相对复杂, 直接将事务用一个完整的 HTM 执行(如图 1 所示)会受到 HTM 硬件限制(见第 1.1 节)的影响, 从而无法完全利用 HTM 的高性能[21]. 以图 1 的事务作为例子来说, 为了找到表 A 中 x 列大于 10 的数据, 该事务首先需要查询表 A 的索引. 通常, 索引操作会存在大量内存访问, 这样, 该事务会受 HTM 访问内存大小限制而中断. 由于一般情况下我们无法直接利用 HTM 加速事务, 一系列工作通过软硬件结合的方式避免事务性能受 HTM 硬件限制的影响[1,7,8], 其中包括如下典型系统: DBX[1]利用乐观并发控制[20]拆分了需要 HTM 保护的事务的程序, 这样, 程序拆分后的部分能够有效地借助 HTM 进行加速; 和 DBX 类似, DrTM[8]借助 TransactionChopping[22]的技巧对事务进行了切分, 切分后的事务可以使用 HTM 进行加速. 最后, 使用 HTM 执行事务无法确保持久性, 因此上述系统均采用了一套额外的日志机制来提供持久性. 本文第 2.1 节将对该机制进行介绍.

1.3 非易失性内存技术(non-volatile memory, NVM)

随着 Intel 发布了 Optane DC persistent memory[11] (由于 IntelOptane DC persistent memory 是目前唯一能够使用的 NVM, 因此在本文中, NVM 即指代 Intel 的 OptaneDC NVM), 非易失性内存正式从模拟器变为现实. 与传统的持久化设备(如 HDD 和 SSD)相比, NVM 提供了更低的时延和更高的带宽. 与此同时, NVM 能够直接接入处理器的内存总线, 因此它具有字节寻址的特点, 即处理器可以像内存一样直接使用 load, store 和 nt-store (nt- store 是一种特殊形式的 store, 它在语义上和 store 一致. 不同的是, nt-store 会绕过处理器的缓存)指令对其进行读写. 使用传统内存指令读写 NVM 会存在持久化问题, 例如内存写指令(如 store)会优先将数据写入处理器的非持久化缓存, 而不是 NVM. 为了支持持久化, Intel 提供了一个针对 NVM 的扩展指令集: 处理器可以使用 clwb 和 clflush 将缓存的数据刷入 NVM, 并能使用 sfence 等待数据刷入的完成.

虽然 NVM 提供了和内存一致的接口, 其性能特点和内存完全不同. 为了更好地利用 NVM, 现有工作对

其性能特点进行了深入的研究[12,13,23]. 总结来说, 当处理器读写 NVM 时, 它需要考虑以下 4 条特性.

- 首先, NVM 具有不对称的读写性能: 它的读带宽远比写带宽高[12,13].
- 第二, NVM 使用和处理器不一致的读写粒度: NVM 以 256 字节为粒度进行读写, 而处理器使用 64 字节. 因此, 为了更好地利用非易失性内存的带宽, 处理器对 NVM 的大数据(大于 64 字节)写应以 256 字节为粒度进行.
- 第三, 处理器对 NVM 的读请求会影响到写请求的吞吐[13], 所以上层应用应该避免使用处理器往 NVM 发送不必要的读请求. 例如, 当处理器写请求大小小于 64 字节(即不满足处理器读写粒度)时, 处理器会先往 NVM 发送一个额外的读请求再进行写. 为了避免这种情况发生, 处理器对 NVM 的小消息(小于 256 字节)写, 应以 64 字节为粒度进行传输[12].
- 最后, 处理器的缓存对 NVM 并不友好[13,23], 因此, 使用 nt-store 去写 NVM 能获得更好的性能.

2 现有工作的分析和在 HTM 中使用 NVM 的挑战

在本节中, 我们首先介绍现有基于 HTM 的多核内存数据库的持久化机制(见第 2.1 节). 随后, 我们通过实验分析该机制在新型 NVM 场景中存在的问题(见第 2.2 节).

2.1 现有基于HTM多核内存数据库的持久化机制: 基于redo日志的群组提交(group commit)

为了在基于 HTM 的多核内存数据库中支持持久化(durability)事务, 现有系统使用 redo 日志将事务的结果存储在持久化设备中[3]. 然而, 将 redo 日志写入传统持久化设备(如 SSD)会带来远大于事务执行时间的开销[24]. 例如, 在 DBX[1]系统中, 在事务执行完后加入写入磁盘操作会给它带来 98%的性能损失(与不支持持久化的事务相比). 因此, 基于 HTM 的多核内存数据库或多核内存数据库均采用 group commit 这一机制均摊事务持久化的开销[1−4,7,12]. Group commit 通过 epoch(时间片)的形式对事务进行分组, 并将一个 epoch 内所有的事务以批量(batching)的方式异步提交. 其中, epoch 是 group commit 的间隔. 这种方式有效地降低了 redo 日志持久化的开销, 但其异步的提交方式会显著增加事务的时延.

图 2 展示了使用 group commit 在 HTM 中执行事务的具体示例. 当执行线程开始执行事务时, 它们首先从系统中获取当前数据库的 epoch 号(①). 这样, 在之后提交事务(即执行过 *xend*(·))时, 执行线程可以将该事务的日志记录到 epoch 号所对应的日志区域中(②). 注意: 由于此时日志还没有被同步到持久化设备(如 NVM)中, 因此该事务仍未完成提交. 事务日志由一个后台线程(日志线程)异步地同步到持久化设备. 该线程定期获取当前系统的 epoch 号, 并同步这些 epoch 中所有事务的日志(③). 具体操作如下: 首先, 日志线程更新当前系统的 epoch 号(④), 以避免未来执行的事务再往该 epoch 中写入日志; 随后, 日志线程开始收集相关事务的日志(⑤), 在收集日志之前, 线程需要等待当前 epoch 中所有事务执行完(或中断); 最后, 日志线程使用一次 IO 操作将收集到的事务日志写入持久化设备(⑥). 从该流程中我们可以看到, 一次持久化操作即可写入一个 epoch 中所有事务的日志. 因此, group commit 可以有效地均摊事务的持久化操作开销.

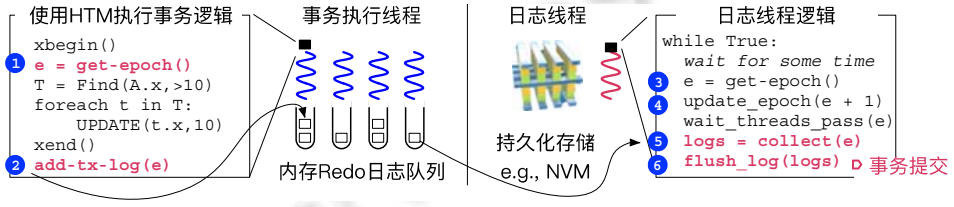


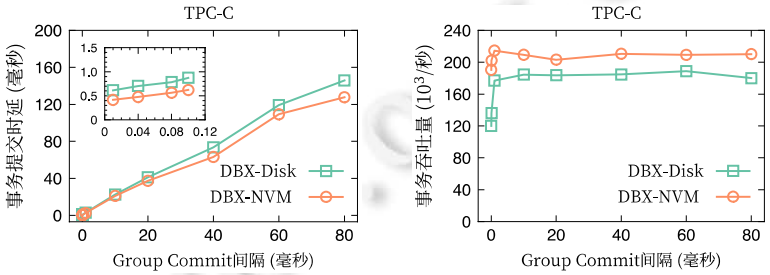
图 2 使用基于 redo 日志的群组提交支持持久化事务的示意图

由于只有事务的日志同步到持久化设备时, 事务才可被视为提交(即执行符合 ACID 特性), group commit 机制下, 事务的提交时延和日志线程的同步间隔(即 epoch 的大小)直接相关. 通常, 为了尽可能地以批量的方式去提交更多的事务, 系统会选择一个较大的 epoch. 例如, Silo[4]和 DBX[1]这两个典型的内存数据库默认均将

epoch 设置为 40 ms. 然而, 较大的 epoch 会给事务带来了高时延. 缩短 epoch 可以降低事务的时延, 但该配置会影响到事务的吞吐: 日志线程会因为来不及写入上一个 epoch 中的事务日志而阻塞当前 epoch 中的事务. 在下一节(第 2.2 节)中, 我们使用实验对该现象进行深入分析.

2.2 Group commit的异步提交机制会显著增高事务处理的时延

为了具体分析 group commit 给基于 HTM 的多核内存数据库带来的时延影响, 本文对 DBX[1], 一个代表性的系统进行了实验分析. 我们使用 TPC-C[18]这一典型的 OLTP 数据库事务处理测试基准进行实验. 在进行测试时, 我们均采用了 DBX 默认最高性能的配置[1](配置经原作者确认. 具体实验配置见第 4.1 节): 数据库使用了 10 个事务处理线程和 1 个日志线程. 同时, 为了避免日志线程受到较慢的存储设备的影响, 我们同时测试了使用磁盘(实验磁盘的配置为希捷 ST1200MM0088 1.5 TB 机械硬盘) (DBX-Disk)和使用 NVM 作为持久化设备的 DBX 性能(DBX-NVM). 在 DBX-NVM 中, 日志线程将事务的日志写入由 NVM 支撑的文件系统(该文件系统使用 Ext4-DAX 模式). 图 3 展示了我们具体的实验结果.



(a) DBX 事务时延 (b) 吞吐

图 3 分析 group commit 间隔对 DBX 事务时延和吞吐的影响

从这些结果中, 我们可以得到以下两个结论.

- Group commit 给基于 HTM 的事务处理带来了显著的时延增高.

由于事务必须等到一个 epoch 内所有其他事务全做完才能将日志写入持久化设备, 事务的时延和 group commit 的间隔(epoch)强相关: 如图 3(a)所示, 当 epoch 的间隔为 1 ms 时, DBX 事务提交的时延为 2.92 ms; 而当其间隔为 80 ms 时, 其时延可以高达 145 ms. 正如我们先前提到过的那样, 为了最大化 group commit 的效果, group commit 的间隔一般远大于事务的执行时长. 例如, 在 DBX[1]中, epoch 的间隔为 40 ms, 其他典型多核内存数据库也采取一样的间隔[3,4]. 这一间隔远大于主流事务处理场景中事务执行的时间, 见表 1, 在 Smallbank[25]和 TPC-C[18]场景中, 事务只需要 0.26 μs 和 9.3 μs 即可完成执行. 即使对 TPC-E[26]这种执行时间较长的事务场景, 其在内存数据库中也只需要 310 μs 即可完成执行. 因此, 使用 group commit 提交事务后, 在常见事务场景中的事务都会有显著的时延增高. 最后, 使用新型 NVM 硬件并不能减少 group commit 事务提交的时延: 在图 3(a)中, DBX-NVM 的时延最多仅比 DBX-Disk 降低 32%. 这是由于软件的异步机制是时延增加的主要因素.

表 1 事务测试基准的执行时间

测试基准	执行时延(μs)
TPC-C[18]	9.3
TPC-E[26]	310
Smallbank[25]	0.26

- 降低 epoch 大小能够降低事务的时延.

例如在图 3(a)中, 当 epoch 设置为 0.01 ms 时, 事务的时延只有 0.62 ms. 然而, 将 epoch 降低到非常小会显著影响事务的吞吐: 当 DBX 的 epoch 减少时, TPC-C 场景中事务吞吐最多会降低至 44%. 从这一观察中, 我们可以得到如下结论:

- 缩短 group commit 间隔以降低事务时延会减少事务吞吐.

如图 3(b)所示，当 epoch 为 0.1 ms 时，DBX-NVM 的事务吞吐受到了 23%的影响. 由于日志线程受较慢的持久化操作影响，其会阻塞住并发执行的事务，因此引发了系统吞吐的下降. 同时，即使在 DBX 中设置一个非常小的 group commit 间隔(如 0.01 ms)，事务的持久化时延仍远高于事务的执行时间(0.62 ms 对比 TPC-C 的 9 μs). 这是由于异步同步操作的时延无法通过 group commit 的间隔进行缩短.

综上所述，现有基于 HTM 的多核内存数据库支持持久化事务时会有较高的时延；同时，它需要牺牲时延来支持高吞吐的持久化事务；同时，它们的持久化机制并不适合用 NVM 这种新型介质进行加速.

2.3 观察：利用NVM减少事务持久化时延，以及其结合HTM使用的挑战

- 观察 1: NVM 的低时延高吞吐可以高效地支持同步日志操作.

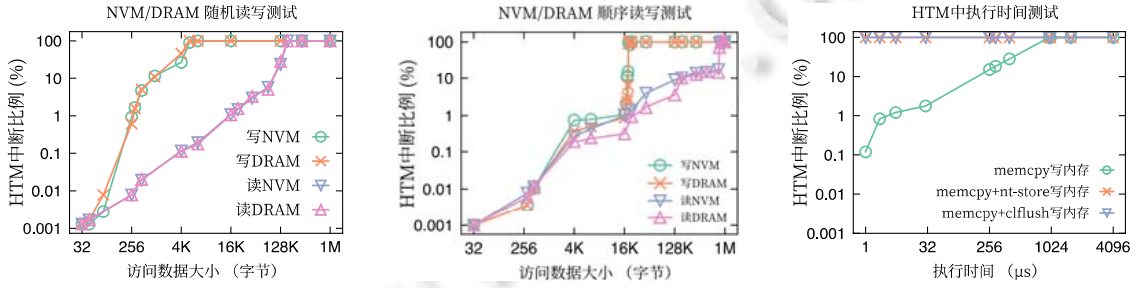
首先，NVM 提供了极低时延的可持久化操作. 如表 2 所示(我们使用 Intel 的内存时延统计器(memory latency checker[27])统计各设备的读时延，并使用类似 Yang 等人工作中的 LATTester[13]统计写时延)，当进行 64 字节的持久化写入时，NVM 的顺序写和随机写分别只需要 171 ns 和 510 ns. 这一时延和 DRAM 相当，且远低于传统持久化设备的访问时延(微秒级)和典型事务的执行时间(见表 1，TPC-C 事务的执行时延为 13 μs).因此，事务可以用同步的方式，即执行完直接将日志写入 NVM 来持久化事务日志. 我们进行了初步的实验来验证同步写入日志在 NVM 上的可行性：在实验中，当 DBX 事务执行完后，我们加入了一次额外的 NVM 写入操作. 测试结果表明，加入一次额外的 NVM 写入只会给 DBX 的事务吞吐带来 11%的降低. 与之相对，写入磁盘会给事务带来 98%的吞吐降低.

表 2 NVM 和 DRAM 中 64 字节访问时延的统计

设备	访问时延(ns)			
	顺序写	随机写	顺序读	随机读
NVM	171	510	248.06	276.6
DRAM	153	234	82.0	82.2

- 观察 2: NVM 中可以高效访问 HTM.

另一方面，我们发现，HTM 中访问 NVM 有和访问 DRAM 相似的性能. 尽管 NVM 的访问时延大于 DRAM (见表 2)，HTM 的主要操作落在处理器缓存中. 因此，除了将数据读取到缓存外，HTM 不会频繁和 NVM 进行交互. 为了验证这一观点，本文在 NVM 中复现了 DBX[1]对 HTM 访问性能的测试(如图 4 所示)，该测试主要考察了 HTM 中 NVM 访问模式对 HTM 中断概率的影响. 如图 4(a)和图 4(b)所示，在随机读写场景中，HTM 访问 NVM 和访问 DRAM 有接近的 HTM 中断概率，该概率受处理器缓存大小所限制，是影响 HTM 性能的主要因素；在顺序读写场景中，NVM 在 16 B–256 KB 时有比 DRAM 高 3%–20%的中断概率，而在其他访问大小时，它们性能接近. 这一差别主要来自 NVM 更高的读访问时延(250 ns 对比 82 ns)，但是并不明显.



(a) (b) (c)

(a) 在随机访问场景中 HTM 中断概率随程序访问 DRAM/NVM 大小的变化
(b) 在顺序访问场景中 HTM 中断概率随程序访问 DRAM/NVM 大小的变化
(c) HTM 中断概率随着程序执行时间和读写内存指令的变化

图 4 在 HTM 中访问 NVM 的性能分析

基于以上两个观察, 一个直观思路是: 将 HTM 和 NVM 有机地结合在一起执行事务, 借助 HTM 提供高性能的事务 ACI 性质, 同时利用 NVM 提供高性能的事务 D (durability)性质. 图 5(a)展示了一个直观地将这两个硬件结合在一起的方式: 当事务在 HTM (xbegin 和 xend 所标记的程序区域)中执行完事务逻辑(do-tx-logic)后, 其使用 add-tx-log-to-nvm 将事务的日志持久化的写入 NVM 中. 然而, 由于以下两个挑战, 我们发现实际中并不能以这种方式将这两个硬件结合在一起:

- 挑战 1: HTM 中无法执行 NVM 的持久化操作.

为了确保数据持久化地写入 NVM, 处理器需要执行 nt-store 和 clflush 指令(见第 1.3 节). 然而, nt-store 和 clflush 会中断 HTM. 如图 4(c)所示, 当处理器在 HTM 中执行这两个指令时, 无论 HTM 执行多久或有无内存访问, HTM 都必然会发生中断. 其背后的原因在于: 这两条指令涉及了处理器缓存机制, 而当前 HTM 的实现及其依赖该机制(见第 1.1 节). 具体来说, HTM 借助处理器缓存一致性来检测不同核之间的内存访问冲突. 由于 nt-store 从设计上绕过了缓存, 因此 HTM 无法确保带该指令执行的程序的正确性, 故只能中断. 与之类似, clflush 会将缓存中未提交的数据刷回内存中, 因此该操作会中断 HTM.

为了确保事务对数据的修改的原子性和持久性, 我们必确保 HTM 中 NVM 操作的持久性. 假设我们不把持久化操作放入 HTM 中, 则其他事务执行线程会观测到一个未提交的数据. 例如, 在图 5(b)中, HTM 提交后日志并未写入到 NVM 中. 此时, 其他事务会读到一个 uncommitted 的数据. 在图 5(c)中, 若 HTM 中执行的事务中断, 则系统会写入一个未提交事务的 redo 日志.

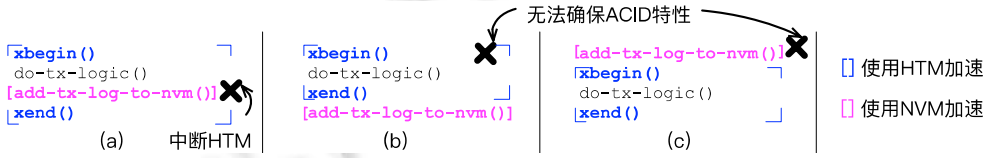


图 5 将 HTM 和 NVM 简单结合在一起降低事务时延的例子, 在图 5(a)中, 持久化写入 NVM 会造成 HTM 100%中断, 而图 5(b)和图 5(c)中都无法确保 ACID

DBXN 中, 我们用 parity version 机制(第 3.1.2 节)来正确区分 HTM 中的事务执行和对 NVM 的写入操作.

- 挑战 2: 利用好硬件的高性能需要针对 NVM 进行协同优化.

NVM 具有和 DRAM 完全不同的性能特性[12,13,23]. 如未经特殊优化、直接将 NVM 当作 DRAM 访问, 则系统只能获得原始 NVM 性能的 33%[12]. 因此, 如何应用 NVM 相关的优化(见第 1.3 节), 对基于 HTM 和 NVM 的多核内存数据库至关重要.

3 DBXN 的设计和实现

- 目标

DBXN 的目标是利用 NVM 降低现有基于 HTM 的多核内存数据库的事务提交时延. 为了达成这一目标, 其需要找到: (1) 高效地将 NVM 和 HTM 结合起来的方法; (2) 结合真实 NVM 特性设计的事务提交方法.

- 综述

DBXN 采取以下两个关键技术点: (1) 基于 parity version 的 NVM 日志机制(第 3.1.2 节); (2) 对真实 NVM 友好的日志写入方法(第 3.4 节). 其中, (1) parity version 能够高效且正确地将 HTM 的操作和 NVM 写入操作进行区分, 从而避免 HTM 中无法持久化写入 NVM 的限制(见第 2.3 节); (2) 根据真实 NVM 硬件的性质优化日志写入, 这一事务提交和 NVM 主要交互操作的性能.

DBXN 基于 DBX[1], 一个现有最优基于 HTM 的多核内存数据库进行设计, 其系统架构如图 6 所示(其具体算法见图 7 和图 8). 和传统基于内存的多核内存数据库一样[1,2,4,7], 数据(包括数据表和表对应的索引)存储在内存中; 当内存不够时, DBXN 使用 NVM 来扩展系统可使用的内存大小. DBXN 使用一个存储层对所有存储相关的操作进行了抽象, 并假设其提供键值存储(key-valuestore)接口. 在收到用户事务请求后, 事务执行线

程开始执行事务; 其执行过程包括通过存储层读写事务所需要的数据以及执行 DBXN 协议以确保事务执行的 ACID 特性. DBXN 将 parity version 应用在 DBX 对 HTM 友好的乐观并发控制协议来执行事务, 其具体包含以下阶段: 执行和验证阶段(①,②)确保了事务的 ACI 性质, 而日志(③)和提交(④)阶段确保了事务的 D 性质. 最后, 和 DBX 不同, DBXN 无需一个后台日志线程把事务日志持久化到 NVM 中[1,3,4]; 在日志阶段(③), DBXN 直接以同步方式将日志写入 NVM.

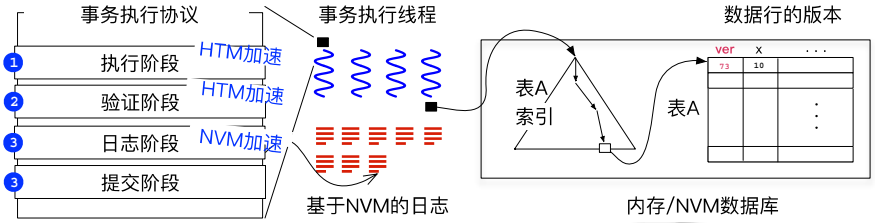


图 6 DBXN 的系统架构图

3.1 Parityversion和DBXN事务执行的协议

本节我们具体介绍 parity version 和其如何应用在 DBXN 中以执行事务的协议. 本节按如下方式进行组织: 首先, 我们介绍 DBXN 中数据库记录的数据结构(见第 3.1.1 节); 基于该数据结构, 我们描述 partiyversion 机制(见第 3.1.2 节); 随后, 我们在第 3.1.3 节描述 DBXN 基于 DBX 的乐观并发控制(optimistic concurrency control, OCC)和 partiyversion 的事务协议; 紧接着, 我们给出一个非正式的 DBXN 正确性说明(见第 3.2 节)以及其如何支持常见数据库操作(见第 3.3 节); 最后, 我们介绍 DBXN 针对真实硬件所做的优化(见第 3.4 节).

3.1.1 DBXN 中数据库记录的数据结构

为了支持高效的事务处理, DBXN 和 DBX 一样, 以行(row)存的方式存储数据库记录, 具体内容如下.

- 数据版本: 由 8 B 整形表示, DBXN 的协议通过该版本同步不同线程之间的并发事务访问和事务持久化的信息.
- 数据内容指针: 8 B 的虚拟地址, 其指向存储在内存或 NVM 中的数据.

当事务修改数据时, DBXN 以交换指针的方式对数据进行修改. 事务会分配一块新的内存区域, 将修改的数据复制到分配的内存区域中, 最后将数据内容指针替换为新分配的地址. 此设计能有效地减少内存写的数量, 尤其适合在 HTM 中修改数据的事务: 无论修改的数据有多大, 事务对一个数据的修改只需要写 8 字节(指针的大小)的内存或 NVM. 由于 HTM 的中断概率和程序访问的内存大小强相关(如图 4 所示), 所以该设计能有效地减少 HTM 的中断概率.

3.1.2 Parityversion 机制

为了能同时利用 HTM 和 NVM, 系统只能采取如图 5(b)或者图 5(c)的方式去执行. 由于图 5(c)中数据库会写入一个未提交事务的日志, DBXN 采取图 5(b)的方式同时避免在 HTM 中写入 NVM. 为了确保 ACID 特性, 我们需要在图 5(b)的基础上防止事务读取到日志未写入的事务的数据.

Parity version 通过版本校验来判断修改数据的事务是否已完成日志持久化. 其中, 数据版本分为两个状态: 奇数版本和偶数版本. 奇数版本的数据表明该数据通过了 HTM 的 ACI 检查, 但是仍未完成持久化.如在图 5(b)中, xbegin 和 xend 中修改的数据的版本则为奇数. 偶数版本则表明数据进一步地完成了持久化. 当事务完成日志写入后, DBXN 将事务所修改的数据变为偶数. 当其他事务读取到奇数版本的数据时, 它们需要等待它变为偶数才可进行提交. 这一等待避免了事务读取到未写入日志的数据.

具体来说, 假设初始数据库中所有数据的版本都是偶数. 当事务在 HTM 中对数据进行修改时, 其需要对数据的相应版本加 1, 也就是将其变成一个待提交但未完成持久化的奇数版本. 递增完成后, 事务可以退出 HTM 并开始往 NVM 中写入日志. 若另一个事务以奇数版本读取到该数据, 则它需要使用 xabort 进行中断并且重试. 直到事务将日志写入 NVM 中后, 它需要再次将其所修改数据的版本加一, 即将它们的版本提升为一

个偶数版本. 此时, 其他事务就可以正确读取到该事务的修改. 最后, 在事务读写数据和进行版本校验时, 系统需要特定机制来同步并发事务之间的操作. 由于我们可以在 HTM 中执行所有的数据读取/修改和版本检查, 因此其无需上锁(locking), 使用 HTM 的 ACI 特性即可完成并发版本校验的同步.

在 HTM 中进行版本校验会略微增大其读写的内存数量, 然而其实际影响可以忽略不计: 首先, HTM 能够支持更大的读集合(如图 4(a)和图 4(b)所示); 其次, 该操作可以复用 DBX 中 OCC 的校验机制. 我们在下一节中进行详细描述.

3.1.3 DBXN 的基本事务协议

在本节中, 我们描述如何将 parity version 应用在 DBX 的事务执行协议中. 我们假设事务只有读写操作, 同时其访问的数据一定存在于数据库中. 在随后的第 3.3 节, 我们会进一步讨论其如何支持插入、删除和读取未存的数据等常见数据库操作. 图 7 和图 8 展示了 DBXN 事务执行的伪代码, 其中, 粉色的部分为 DBXN 在 DBX 协议基础上的扩展.

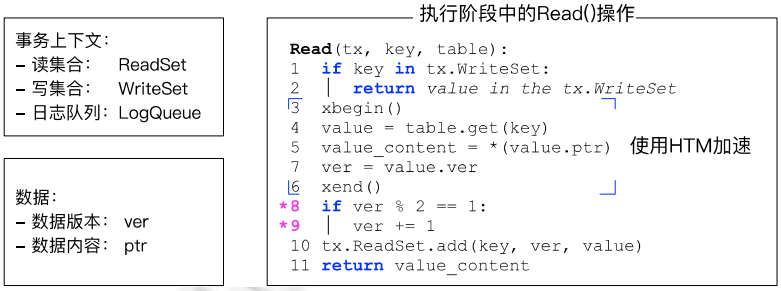


图 7 DBXN 中事务上下文数据结构和执行阶段读写数据(Read)的伪代码

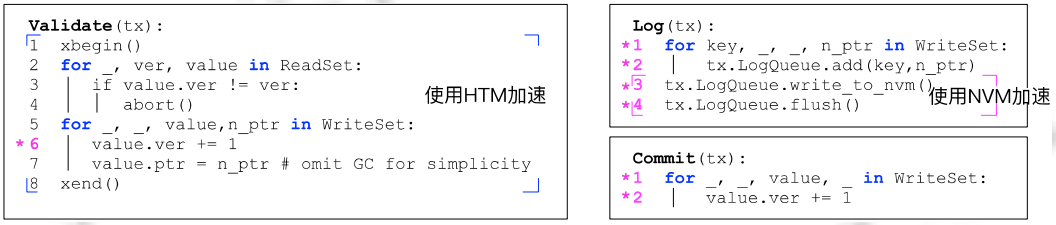


图 8 DBXN 中事务验证、日志和提交阶段的伪代码

- 事务的执行上下文

当一个事务开始时, DBXN 为其分配一个上下文来记录事务执行的元数据. 具体内容如图 7 所示. ReadSet 是事务读的数据的集合, 其每项内容包括事务读取数据的键、读取时数据的版本和指向该数据的引用(见第 3.1.1 节). WriteSet 记录了事务写数据的集合, 其每项内容在 ReadSet 的基础上增加了事务修改后数据的指针. 通常, 所有写集合中的数据也在读集合中. LogQueue 为事务在 NVM 中的日志队列. 和现有工作一样, DBXN 按照 redo 日志[3,4]组织日志内容, 其中包含事务所有修改数据的键、事务提交的数据版本和新修改的内容.

DBXN 按执行、验证、日志和提交阶段依次执行一个事务, 其中, 执行阶段执行事务的逻辑; 验证阶段判断执行阶段中事务执行的结果是否可以提交, 即是否满足 ACI 特性; 日志阶段确保事务的结果存储在 NVM 中; 提交阶段在最后将事务的结果提交, 即可见于其他事务.

- 事务执行阶段

在该阶段中, 事务通过读写其所需要的数据来执行用户事务的逻辑. 对于每一个数据的读写, 事务必须调用 DBXN 的特定 Read/Write 接口与存储层进行交互, 其中, Read 的伪代码如图 7 所示. Read 的主要的操作为调用存储层接口将数据 key 对应的值读取到事务中(第 4 行). 为了支持验证阶段的检查, 事务同时会读取该数据当前的版本(value.ver). 由于读取数据版本和数据值必须为一个原子的操作, DBXN 和 DBX 一样使用

HTM 保护数据值和版本的读取(第 3 行–第 6 行). 最后, 当读取完成后, 事务将该数据的元数据加入其读集合(ReadSet)中(第 10 行). 对于写操作, DBXN 将写的内容缓存在事务的写集合中而不直接更新存储层. 首先, DBXN 将数据按 Read 读取到读集合中, 分配一块新的内存来存储事务的修改, 最后将修改的内容缓存到写集合. 由于 Write 的逻辑和现有 OCC 协议类似, 所以本文略过其伪代码. 最后, 由于一个事务需要读到自己的写操作, 在进行读取的时候, Read 会优先检查要读的数据是否在事务的写集合中(第 1 行、第 2 行): 如果是, 则 DBXN 直接返回写集合中缓存的数据.

和 DBX 的 OCC 不同, 在执行阶段, DBXN 的事务会读取到一个奇数版本, 即未完成持久化的数据(见第 3.1.2 节). 理论上, 遇到该情况, 事务需要中断 OCC 执行并进行重试. 在这里, DBXN 采取了一个优化, 即将中断推后到验证阶段而非执行阶段. 这是基于一个观察: 由于 NVM 具有低时延, 因此当事务执行到验证阶段后, 数据大概率完成了持久化. 这样, 可以避免在执行阶段提前中断事务. 为了确保验证阶段能够正确检测出读的数据是否完成持久化, DBXN 在 Read 的第 8 行额外检查数据的版本是否是奇数: 如果是奇数, 则表明事务在之后的验证阶段需要额外检查该数据版本是否变为偶数. 为此, DBXN 将其版本加 1(第 9 行), 再将其放入读集合中. 这样, 若事务在验证阶段该数据版本仍为奇数, 则事务会因为 OCC 的版本不一致而中断并重试. 详细见下文验证阶段的描述.

- 事务验证阶段

为了确定事务是否能够提交, DBXN 使用 OCC 的验证阶段对事务进行验证. 和 DBX 一样, DBXN 通过验证事务的读集合自执行阶段以来是否发生改变来判断事务是否满足 ACI 特性.

如图 8 中 Validate 的伪代码所示, 事务首先检查其读集合中的数据的最新版本是否和读集合中记录的版本一致(第 2 行、第 3 行): 如果不一致, 则要么有一个并发事务修改了该事务读到的数据(违反 ACI 性质), 或者该数据的版本为奇数(违反了 D 性质). 当任意一种情况发生时, DBXN 中断该事务并进行重试(第 4 行). 如果事务读集合中的数据并未发生修改, 则事务的执行符合 ACID 性质. 在此情况下, DBXN 开始尝试提交该事务. 在进入日志和和提交阶段之前, DBXN 首先将写集合中数据的修改更新到数据库中. 对于写集合中的所有数据, DBXN 把数据的内容指针指向事务在执行阶段分配的地址(第 7 行). 同时, 为了确保在事务日志阶段完成前, 其他事务并不会读到它的修改并提交, DBXN 会将写的数据的版本更新为奇数(第 6 行).

在 OCC 中, 读集合的验证和写集合的写入(即第 2 行–第 7 行)需要原子执行以确保正确性. 通常, 实现该操作需要全局或者细粒度的锁进行保护[1,4], 而上锁操作会带来额外的开销. 和 DBX 一样, DBXN 将这些操作包含在一个 HTM 中执行(第 1 行和第 8 行), 以借助硬件高效确保原子性.

- 事务日志阶段

该阶段确保事务的修改会存储在持久化(即 NVM)的日志中, 这样, 当机器发生宕机时, 事务的结果可以从持久化的日志中进行恢复. DBXN 采用 SiloR[3]的机制进行恢复, 因此本文不再赘述. 由于 NVM 具有极低的写内存时延, DBXN 选择在事务验证阶段执行完之后直接同步进行日志写入操作. 如图 8 所示, 在 Log 中, DBXN 首先将事务对数据库的修改组织成 redo 日志(第 1 行、第 3 行); 随后, 其将日志写入到 NVM 中(第 3 行). 由于直接写入 NVM 无法确保日志内容持久化(例如数据写入到了处理器缓存), 在写完日志后, DBXN 会进一步利用 NVM 的扩展指令集, 以确保写入的内容刷入到 NVM 中. 在当前硬件平台, DBXN 可以使用 clwb 和 sfence 确保数据写入到 NVM.

- 事务提交阶段

当事务执行完日志阶段后, 它的修改已经能确保持久化. 此时, 事务已经可以视为提交, 即将事务提交状态返回给用户. 同时, 为了让其他事务可以读到该事务的修改, DBXN 在图 8 中 Commit 的第 1 行、第 2 行根据 parity version 的机制将事务写的数据的版本变为偶数. 需要注意的是, 提交阶段无需使用 HTM 保护. 这是由于其他并发事务无法基于该事务的奇数版本数据提交, 因此不会有和该事务冲突的并发写操作.

3.2 正确性说明

强可线性化(strict serializability)是理想的 ACI 性质. 本节以非正式的形式说明 DBXN 能够同时确保事务

的强可线性化和持久性.

- 强可线性化

我们通过将 DBXN 的协议规约成强二阶段锁(strict two-phase locking, S2PL)[28], 一个能够确保强可线性化的并发控制协议, 来说明 DBXN 能够确保强可线性化. 在 S2PL 中, 事务执行分为两个阶段: 第 1 个阶段, 事务对所有读写的数据进行拿锁(在读取到该数据时); 在事务提交后, 其执行 S2PL 的第 2 个阶段, 该阶段把所有第 1 阶段拿的锁释放.

在 DBXN 中, 当一个事务提交时, 其读或写的数据可以被归约成在执行阶段拿锁(图 7 第 10 行), 在提交阶段(图 8 中 Commit 的第 2 行)放锁. 这是由于没有其他事务可以对其读写的数据进行修改, 和拿锁的效果等价. 这是因为在 DBXN 中, 根据 parity version(第 3.1.2 节)的机制, 只有数据版本从偶数变为奇数或者偶数变为另一个偶数时, 其内容才会发生修改. 当一个事务读取一个奇数版本(v)的数据时, Read 的第 9 行和 Validate 的第 3 行、第 4 行确保了只有提交时该数据版本为(v+1)事务才可正常提交. 根据 parity version, 此时读取的数据未发生改变. 当事务读取一个偶数版本的数据时, 只要数据版本发生变化(即内容发生变化), 事务就会中断.因此, 若一个事务完成了提交, 其读写的数据在执行期间均不会被其他事务修改.

在 DBXN 中, 若一个处理器核完成了事务提交, 后续的处理器一定能够读取到该事务对数据库的修改. 这里, 我们假设的提交是完成了 commit 操作. 首先, DBXN 在事务提交时, 数据的新内容会在 HTM 中更新到数据库中(Validate 第 7 行). 由于 HTM 特性可以保证后续(该处理器或其他处理器)的内存/NVM 读操作一定会读取到它的修改. 第二, DBXN 会直接从内存/NVM 中读取数据(Read 的第 4 行). 这样, 根据之前 HTM 的特性, 当一个处理器完成事务提交后, 后续处理器一定能够读取到该事务的修改.

- 持久化

为了说明 DBXN 支持持久化, 我们只需说明其中的事务不会读到一个日志未持久化的事务的修改. 根据图 8 日志阶段的伪代码, 事务只有在日志刷到 NVM 后(Log 的第 4 行)才会变为偶数. 根据 parity version 的机制, 只有读取到偶数的数据事务才能提交.

3.3 支持常见数据库操作

本节描述如何基于第 3.1.3 节的基本事务协议进一步支持常见数据库操作, 如插入、删除和范围查找. 在 DBXN 中, 这些操作和原本 DBX 的处理方案类似, 即加入基于 NVM 的低时延事务日志不会影响 DBX 中这些操作的执行.

- 读取未存在于数据库中的数据

当一个事务读取到未存在于数据库中的数据时, DBXN 需要检测出他与并发插入该数据的事务的冲突情况. DBXN 利用 DBX 存储层的 get-with-insert 的接口, 方便对该情况进行检测. 如名字所示, get-with-insert 在遇到一个不存在的数据时, 会插入一个该数据键对应的空的数据. 这样, 并发插入该数据的事务会转化为一个冲突的写操作. 因此, DBXN 可以在验证阶段检通过读写冲突检测出并发插入的冲突情况. 注意, 此插入不会加入 DBXN 的日志中, 因为其并未插入真实数据.

- 插入和删除

当 DBXN 在执行阶段遇到插入操作时, 其通过 get-with-insert 在存储层插入一个空的数据值, 并将插入的数据缓存在事务的写集合中. 这样, 该插入操作可视为一个普通的写操作在 DBXN 的协议中正确执行. 对于删除操作, DBXN 不能直接在执行阶段把数据删除, 因为事务可能会中断. 因此, DBXN 和 DBX 一样, 将删除操作当作一个写入空指针的写操作. 这一方案会造成索引中的内存浪费, DBX 采用基于 group commit 的 epoch 机制进行垃圾回收. 由于 DBXN 中没有 group commit, 所以其只是复用了 DBX 中的 epoch 机制进行垃圾回收.

- 范围查找

和 DBX 一样, DBXN 依赖存储层中的 B+树索引支持范围查找. 在第 3.1.3 节协议的基础上, 支持范围查找需要额外检测幻读(phantom)[29]现象. 和 DBX 一样, DBXN 通过在协议中检测事务范围查找中所涉及的叶子节点是否发生变化来检测该现象. 为了支持检测, 首先, B+树所有的叶子节点都需要记录一个额外的版本号,

该版本和数据版本(见第 3.1.1 节)一样. 当叶子节点发生插入或者分裂时, 其版本号就会进行增加. 同时, 当事务进行范围查找时, DBXN 会将其所访过程中所有 B+树叶子节点记录到读集合中. 为了确保原子性, DBXN 使用 HTM 保护访问叶子节点和读取版本号的操作. 通过以上两个机制, 在验证阶段, DBXN 可以通过检测事务读集合中叶子节点的版本是否发生变化来检测是否发生幻读现象. 需要注意的是, 由于索引操作可以在恢复时重建, 因此 DBXN 不会将对树的修改记录到事务的 redo 日志中. 所以, 对叶子节点的版本号更新为加 2 操作, 即从偶数变为偶数, 而不是像 DBXN 一样进行加 1.

3.4 实现和优化

我们基于 DBX[1]的代码实现了 DBXN. 基于 DBX 实现有两个好处.

- 首先, DBXN 可以复用 DBX 的高性能存储层: DBX-store. DBX-store 使用了一个基于 HTM 的 B+树来支持高效的并发有序键值存储操作.
- 除此之外, DBXN 还可以复用 DBX 的一系列针对 HTM 的实现优化, 包括退回处理器(fallback handler)等优化操作. 在 DBX 的基础上, DBXN 进一步针对 NVM 做了一系列的优化.
- 对真实 NVM 友好的日志写入方法.

在前文(第 2.3 节)中我们提到, 结合 NVM 进行设计需要针对真实硬件的特性进行相应优化. 为了达到这一目标, 我们基于现有针对真实 NVM 特性的研究工作[12,13]优化了 DBXN 中的日志写入操作. 首先, 在 DBXN 往 NVM 写入日志时(第 3 行), DBXN 使用 nt-store 而不是传统的 memcpy 进行写入. 一方面, nt-store 绕过处理器缓存的特性对真实 NVM 更为友好[13]; 另一方面, 在第 4 行确保日志写入到 NVM 中后, 使用 nt-store 无需执行 clflush/clwb, 节省了指令开销. 除了采用 nt-store 以外, DBXN 对日志进行留白操作(padding), 以达到选取最适合 NVM 的访问粒度进行写入的目的. 对于小于 256 字节的事务日志, DBXN 以 64 字节的粒度将其写入 NVM[12].对大于 256 字节的日志, DBXN 以 256 字节的粒度进行写入[13]. 我们将在第 4.4 节通过实验分析这 3 个优化对 DBXN 的性能影响.

4 系统评测

为了测试 DBXN 利用 HTM 和 NVM 加速多核内存数据库的事务处理性能, 我们在一台配有真实 NVM 和 HTM 的服务器(第 4.1 节)进行测试. 我们的测试期望回答以下几个问题.

- DBXN 的 HTM 和 NVM 结合方法能否大幅降低 DBX 持久化事务处理的时延(见第 4.2 节)?
- 和现有基于 NVM 的事务系统相比, DBXN 的事务处理性能如何(见第 4.3 节)?
- DBXN 的各个设计对其性能有何影响(见第 4.4 节)?

4.1 测试平台配置

我们在一台装备有 10 核 Intel Xeon Gold 5 215 M 处理器的服务器上开展所有的测试, 此处理器支持 HTM 特性. 服务器有 192 GB 内存和 750 GB IntelOptanePM 非易失性内存. 其中, 非易失性内存(NVM)由 6 根 NVMDIMM 组成. 该配置为服务器能够获得的最高 NVM 性能的配置: 320 Gb/s 的读和 100 Gb/s 的写带宽.

- 对比对象和各系统配置.

我们将 DBXN 和以下系统进行对比.

- DBX-DRAM[1]是当前典型的基于 HTM 加速的多核内存数据库, 其采用了和 DBXN 类似的 OCC 协议以减少事务受 HTM 硬件限制的影响. 默认情况下, DBX-DRAM 不开启持久化事务(所以我们将其标记为 DRAM). 因此, 其在实验中代表系统性能所能达到的上限.
- DBX-NVM 是 DBX 带持久化的版本, 其利用 group commit 机制(见第 2.2 节)在基于 NVM 的文件系统上支持持久化事务.
- DBX-naïve 是最简单利用 NVM 加速 DBX 持久化事务的方法(如图 5(a)所示), 其无法将 NVM 和 HTM 有效结合在一起(见第 2.3 节).

除了和以 DBX 为代表的多核内存数据库进行对比外, 我们还和 Pisces[15], 一个基于 NVM 的持久化事务性内存编程框架进行了对比. 由于持久化事务性内存支持 ACID 特性, 因此它也可以用来执行数据库事务[15]. 在默认情况下, Pisces 将所有的数据(包括索引)放入 NVM 中. 为了确保公平的比较, 我们还将 Pisces 和 DBXN-NVM 对比: DBXN-NVM 在 DBXN 的基础上, 将所有的数据(包括索引)均放在了 NVM 中.

当没有显示地说明时, 我们均采用各系统所能达到的最高性能的配置. 例如, 在 DBXN 和 DBX-DRAM 中, 系统的最高性能配置为使用 10 个线程(服务器的处理器核上限)进行事务处理.

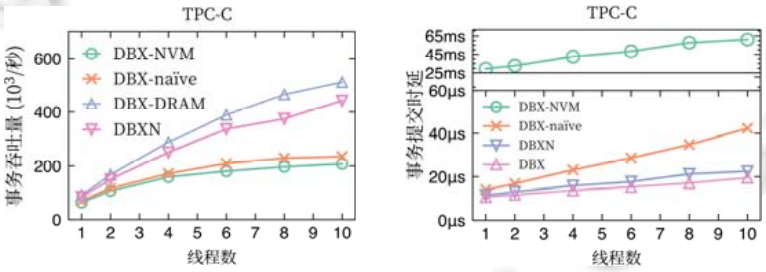
- 测试基准.

我们使用两个典型的事务处理测试基准对 DBXN 和其对比对象进行分析: TPC-C[18]和 Smallbank[25]. TPC-C 模拟了一个股票交易场景, 其事务包括了相对复杂的处理器操作. 例如, TPC-C 中占比最高的事务, new-order 事务的逻辑为采购十几个股票, 该事务需要对数据库进行股票的读写并插入相应的订单. 在 TPC-C 中, 我们部署的数据库规模为 10 个仓库. Smallbank 模拟了一个简单的银行交易系统, 其事务的逻辑只有一到两个数据库读写操作. 例如, Smallbank 中, deposit checking(模拟转账)事务的逻辑为将一个银行账户的金额转移至另一个银行账户. 在 Smallbank 中, 我们部署的数据库中总共包括 250 000 个账号.

4.2 端对端性能比较

- TPC-C 的吞吐分析

图 9(a)展示了 DBXN 和对比对象在 TPC-C 测试基准下吞吐端到端(end-to-end)的对比. 在确保持久化事务的各配置(DBX-NVM, DBX-naïve 和 DBXN)中, DBXN 能达到最高的吞吐: 使用 10 个线程时, 每秒能够处理 432 K 个 TPC-C 事务, 达到了 86%未带持久化事务的 DBX-DRAM 吞吐(432 K 对比 505 K). 这一实验结果说明: DBXN 利用 NVM 提供的持久化机制, 对 DBX 原本基于 HTM 的事务并发控制方法开销非常小; 同时, DBXN 比 DBX-NVM 和 DBX-naïve 吞吐分别提高了 2.1 倍和 1.9 倍. DBX-NVM 仍采用 DBX 默认的 group commit 机制去提交事务, 在该设计下, 系统性能受事务和日志线程交互的影响, 无法发挥 NVM 的全部高性能. DBX-naïve 由于没有考虑 NVM 写入会中断 HTM 执行的问题, 无法利用 HTM 进行事务提交, 因此无法发挥 HTM 的全部性能. DBXN 有效地规避了以上两种设计的问题, 能同时利用 HTM 和 NVM 获得高吞吐.



(a) 吞吐 (b) 时延

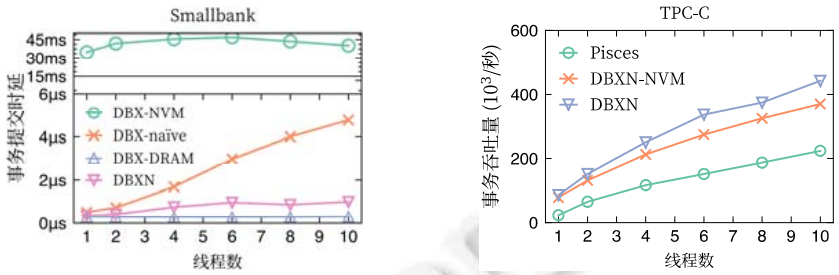
图 9 DBXN 和对比对象在 TPC-C 中的吞吐和时延比较

- TPC-C 的时延分析

图 9(b)进一步地展示了 DBXN 和对比对象在 TPC-C 测试基准下的时延对比. 在确保持久化事务的各配置中, DBXN 能够获得最低的时延: 使用 10 个线程时, DBXN 的持久化事务时延为 22 μs; 而使用一个线程时, 其时延为 13 μs. 这两个时延分别只比非持久化事务时延增加了 1.15 倍和 1.1 倍. 在 10 个线程时, 事务会因为并发控制的同步操作造成时延上升. 与 DBXN 相比, DBX-NVM 的时延是其 2 720 倍. 在 DBX-NVM 中, 异步的事务提交机制是时延上升的主要原因(见第 2.2 节), 所以即使是低时延的 NVM 也无法降低其时延. 最后, 与 DBX-naïve 相比, DBXN 的时延降低了 53%–83%. 由于 DBX-naïve 无法利用 HTM 进行并发控制, 其只能退化到开销更大的软件并发控制机制.

- Smallbank 的时延分析

图 10(a)分析了在 Smallbank 测试基准中 DBXN 和各对比对象的时延对比. 由于 Smallbank 中各系统的吞吐对比和 TPC-C 类似, 因此为了简略, 我们略过对其分析. 从测试图中我们可以看到, 各系统时延的对比趋势与 TPC-C(如图 9(b)所示)中类似: 由于不支持持久化事务, DBX-DRAM 具有最低的时延; DBXN 在 DBX-DRAM 的基础上使用 NVM 提供低时延事务, 其时延比 DBX-DRAM 高了 1.7–2.4 倍; DBX-naïve 无法同时利用 HTM 和 NVM 的硬件特性, 因此其时延比 DBXN 高了 1.2–3 倍; 最后, DBX-NVM 的异步提交机制带来了 4 个数量级的时延差别(时延为 DBXN 的 4 万–5 万倍).



(a) DBXN 和对比对象在 Smallbank 上时延的对比　　(b) DBXN-NVM 和 Pisces 在 TPC-C 中的对比

图 10　DBXN 和对比对象在时延和吞吐上的对比

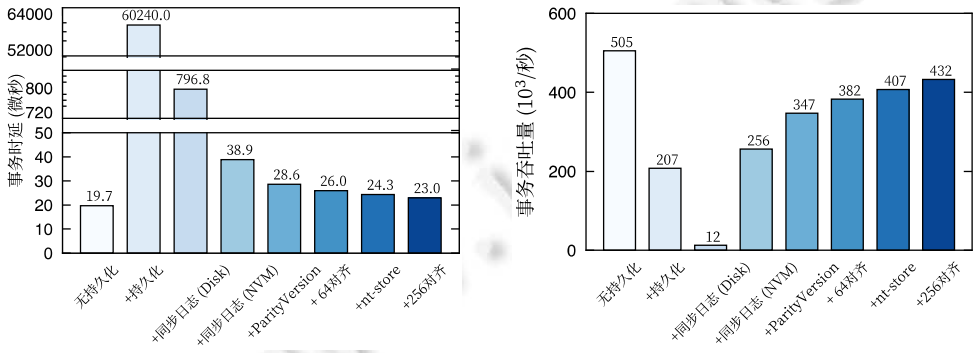
4.3 与现有基于真实NVM的持久化事务性内存系统的性能对比

在图 10(b)中, 我们对比了 DBXN、DBXN-NVM 和 Pisces 在 TPC-C 上的性能. 其中, DBXN-NVM 和 Pisces 能直接进行性能对比, 因为他们均将所有数据库数据(包括索引)放入 NVM 中. 我们可以看到, DBXN-NVM 比 Pisces 快了 1.65–3.41 倍. 这一性能提升主要来自 3 个方面: 首先, Pisces 是一个通用的事务性持久化内存机制, 因此它未针对数据库事务这一特定场景进行优化; 其次, Pisces 虽然利用了真实 NVM 去加速持久化操作, 它并未根据真实 NVM 的硬件特性进行相应的优化, 与其不同, DBXN 根据真实 NVM 的特性进行了针对性的设计和实现(见第 3.4 节); 最后, Pisces 的协议并没有根据 HTM 进行加速.

与 DBXN 相比, DBXN-NVM 有额外的 9%–16%的性能损失. 这是因其在 NVM 中放入了额外的数据结构(如索引), 而对 NVM 的访问操作比 DRAM 更慢(见表 2).

4.4 各设计因素对性能的影响

在图 11 中, 我们分析了各设计因素对 DBXN 性能的影响.



(a) TPC-C 时延　　(b) TPC-C 吞吐

图 11　各设计因素对 DBXN 在 TPC-C 时延和 TPC-C 吞吐的影响

首先可以看到, +持久化, 即 DBX-NVM 将 DBX-DRAM 的时延提升了一个数量级, 从 19.7 μs 达到了 60

多毫秒. 该时延的提升主要来自异步事务提交的软件机制. 采取同步写入磁盘的方式, 即, +同步日志(disk)将时延降低到了 796.8 ms, 却给事务吞吐带来了 94%的下降. 这是因为慢速的磁盘成为了高吞吐内存数据库的性能瓶颈. 采用 NVM(+同步日志(NVM))能够有效地降低同步日志的持久化开销: 其将时延进一步降低到了 38.9 ms, 同时将吞吐增加到了每秒 256 K 事务. 但是, 该配置离最优的性能(每秒 432 K)仍有较大的距离. 这是因为: (1) 它没有办法利用好 HTM. (2)它没有针对真实 NVM 进行优化. 当采用 ParityVersion(+Parity Version)机制后, DBXN 能够有效地利用 HTM 进行事务提交, 从而进一步地获得了 1.35 倍的吞吐提升和 26%的时延下降. 最后, 根据真实 NVM 特性采取相关优化(见第 3.4 节), 即+64 对齐、+nt-store 和+256 对齐将 DBXN 的吞吐(在+ParityVersion 的基础上)分别提升了 1.1 倍、1.18 倍和 1.24 倍, 将时延降低了 9%, 15%和 20%.

4.5 可扩展性测试

最后, 我们分析 DBXN 在更大数据库下的可扩展性. 当数据规模扩大时, DBXN 和其对比系统会受到缓存命中率下降的影响, 导致性能相应下降. 在图 12 中, 通过增加 TPC-C 中部署的 warehouse 数量来测试. 我们可以看出, 在该场景下, DBXN 性能受数据规模增大的影响. 当使用 200 个 warehouse 时, DBXN 的性能与使用 10 个 warehouse 相比下降了 57%. 这一下降主要来自系统缓存局部性(cache locality)的下降, 例如, 在 200 个 warehouse 的数据库上, DBX-DRAM 也有 48%的性能下降. 最后, 即使在更大的数据规模下, DBXN 的性能仍比 DBX-naïve 要好 1.63–2.07 倍.

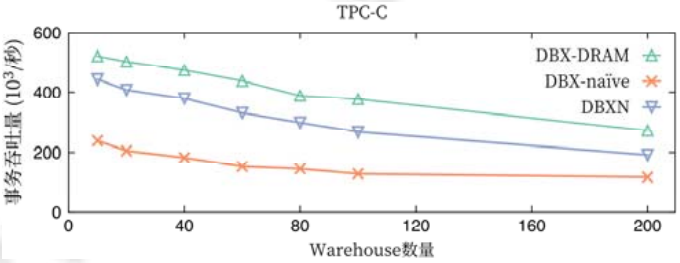


图 12 在 TPC-C 中使用更多 warehouse 带来的性能影响

5 相关工作

- 针对真实 NVM 特性研究的工作.

Yang 等人的工作[13]首次对真实 NVM 特性进行了系统化的研究. 他们总结真实 NVM 的特性, 包括读写非对称性能和访问粒度等. RDPMA[12]系统化地总结了如何高效地将 NVM 和新型网络硬件一起进行协同设计. Kalia 等人的工作[23]发现, 处理器缓存会影响上层系统对 NVM 带宽的利用. 在针对真实 NVM 实现 DBXN 的过程中, 我们分别借鉴了这些工作针对 NVM 采取的相应的优化.

- 利用 NVM 或 HTM 的系统.

数据库和系统研究者对于如何利用 NVM 和 HTM 加速系统做了长久的研究. NVM 被长期用来加速文件系统的性能[30]. 除此之外, NVRC[31]提出了一种日志更新策略来减少对 NVM 中的写操作. 徐远超等人[32]研究了使用 NVM 系统可能存在的安全问题. 罗永平等人的工作[33]研究了如何利用 DRAM 和 NVM 的架构优化传统数据库的磁盘连接操作. 同时, 一系列的工作探索了如何设计对 NVM 友好的数据结构[34–36]以及对 NVM 友好的键值存储[37]. 最后, 舒继武等人的工作[38]则对非易失性内存的相关研究和机遇进行了总结.

在硬件事务内存方面, 吴振伟等人的工作[39]对比了利用 HTM 加速并发链表的设计和传统基于锁的并发链表之间的性能. HybridTCache[40]是一种基于专用事务缓存的软硬件协同事务内存系统. 和它一样, DBXN 基于 DBX 使用软件解决硬件事务性内存的限制. 曾坤等人的工作[41]探索了基于依赖图的 HTM 比目前基于冲突检测的 HTM 性能更好. 最后, SPINRTM[42]结合 HTM 为虚拟机设计全新的同步机制.

除了专门使用 NVM 或 HTM 的工作外, 一系列的工作探索了如何将 HTM 和 NVM 进行协同工作[14,43,44].

由于当时缺乏真实 NVM 硬件, 这些工作只考虑了基于模拟 NVM 的设计. 和这些工作类似, DBXN 也关注于如何更好地利用 HTM 和 NVM 加速数据库事务处理. 同时, DBXN 进一步地考虑了真实 NVM 硬件特性带来的影响. Crafty[43]采用了一种 undolog 的机制来同步 HTM 中执行的事务, DBXN 采用了 redolog 的机制. NV-HTM[44]考虑了 HTM 无法和 NVM 协作的问题, 它提出了一种滞后机制来正确的提交事务. DBXN 使用 parity version 机制来协作 HTM 和 NVM. PHyTM[14]是一个混合软件事务内存和 HTM 的混合内存事务系统, 它关注于如何解决 HTM 中 progress 和工作集有限的问题. PHyTM 没有考虑 NVM 和 HTM 无法结合在一起的问题, 和它不同, DBXN 的 parity version 机制有效地结合了 HTM 和 NVM.

6 总 结

新型低时延非易失性内存的出现, 给减少基于 HTM 的多核内存数据库事务提交时延提供了机遇. 本文提出了 DBXN, 一个基于现有 HTM 的多核内存数据库, 并用 NVM 提供低时延的事务提交的内存数据库. DBXN 通过包括 parity version 等一系列设计, 高效地将 HTM 和 NVM 结合在一起, 降低了事务处理时延. 在现有基于 HTM 的多核内存数据库基础上, DBXN 能在提升事务处理吞吐的同时, 将事务提交时延降低了一个数量级.

References:

[1] Wang ZG, Qian H, Li JY, Chen HB. Using restricted transactional memory to build a scalable in-memory database. In: Proc. of the 9th European Conf. on Computer Systems (EuroSys 2014). New York: Association for Computing Machinery, 2014. Article 26. [doi: 10.1145/2592798.2592815]

[2] Wu YJ, Chan CY, Tan KL. Transaction healing: Scaling optimistic concurrency control on multicores. In: Proc. of the 2016 Int'l Conf. on Management of Data (SIGMOD 2016). New York: Association for Computing Machinery, 2016. 1689−1704. [doi: 10.1145/2882903.2915202]

[3] Zheng WT, Tu S, Kohler E, Liskov B. Fast databases with fast durability and recovery through multicore parallelism. In: Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation (OSDI 2014). USENIX Association, 2014. 465−477.

[4] Tu S, Zheng WT, Kohler E, Liskov B, Madden S. Speedy transactions in multicore in-memory databases. In: Proc. of the 24th ACM Symp. on Operating Systems Principles (SOSP 2013). New York: Association for Computing Machinery, 2013. 18−32. [doi: 10.1145/2517349.2522713]

[5] Thomson A, Diamond T, Weng S C, *et al*. Calvin: Fast distributed transactions for partitioned database systems. In: Proc. of the 2012 ACM SIGMOD Int'l Conf. on Management of Data. 2012. 1−12.

[6] Yu XY, Bezerra G, Pavlo A, Devadas S, Stonebraker M. Staring into the ABYSS: An evaluation of concurrency control with one thousand cores. Proc. of the VLDB Endowment, 2014, 8(3): 209−220. [doi: 10.14778/2735508.2735511]

[7] Leis V, Kemper A, Neumann T. Exploiting hardware transactional memory in main-memory databases. In: Proc. of the 2014 IEEE 30th Int'l Conf. on Data Engineering. 2014. 580−591. [doi: 10.1109/ICDE.2014.6816683]

[8] Wei X, Shi J, Chen Y, Chen R, Chen H. Fast in-memory transaction processing using RDMA and HTM. In: Proc. of the 25th Symp. on Operating Systems Principles. 2015. 87−104.

[9] Amazon found every 100ms of latency cost them 1% in sales. 2019. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales

[10] 2015. https://www.anandtech.com/show/9541/intel-announces-optane-storage-brand-for-3d-xpoint-products

[11] Smith R. Intel announces optane storage brand for 3D XPoint products. 2015. https://www.anandtech.com/show/9541/intel-announces-optane-storage-brand-for-3d-xpoint-products

[12] Wei XD, Xie XT, Chen R, Chen HB, Zang BY. Characterizing and optimizing remote persistent memory with RDMA and NVM. In: Proc. of the 2021 USENIX Annual Technical Conf. (USENIX ATC 2021). USENIX Association, 2021.

[13] Yang J, Kim J, Hoseinzadeh M, Izraelevitz J, Swanson S. An empirical guide to the behavior and use of scalable persistent memory. In: Proc. of the 18th USENIX Conf. on File and Storage Technologies (FAST 2020). USENIX Association, 2020. 169−182.

[14] Avni H, Brown T. Persistent hybrid transactional memory for databases. Proc. of the VLDB Endowment, 2016, 10(4): 409−420. [doi: 10.14778/3025111.3025122]

[15] Gu JY, Yu QQ, Wang XY, Wang ZG, Zang BY, Guan HB, Chen HB. Pisces: A scalable and efficient persistent transactional memory. In: Proc. of the 2019 USENIX Annual Technical Conf. (USENIX ATC 2019). 2019. 913–928.

[16] Liu MX, Zhang MX, Chen K, Qian XH, Wu YW, Zheng WM, Ren JL. DudeTM: Building durable transactions with decoupling for persistent memory. In: Proc. of the 22nd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2017). New York: Association for Computing Machinery, 2017. 329–343. [doi: 10.1145/3037697. 3037714]

[17] Giles ER, Doshi K, Varman P. SoftWrAP: A lightweight framework for transactional support of storage class memory. In: Proc. of the 31st Symp. on Mass Storage Systems and Technologies (MSST). IEEE, 2015. 1–14.

[18] The Transaction Processing Council. TPC-C Bench-mark V5.11, 2022. http://www.tpc.org/tpcc/

[19] Herlihy M, Moss JEB. Transactional memory: Architectural support for lock-free data structures. In: Proc. of the 20th Annual Int'l Symp. on Computer Architecture (ISCA '93). New York: Association for Computing Machinery, 1993. 289–300. [doi: 10.1145/165123.165164]

[20] Kung HT, Robinson JT. On optimistic methods for concurrency control. ACM Trans. on Database Systems, 1981, 6(2): 213–226. [doi: 10.1145/319566.319567]

[21] Chen HB, Chen R, Wei XD, Shi JX, Chen YZ, Wang ZG, Zang BY, Guan HB. Fast in-memory transaction processing using RDMA and HTM. Article 3. ACM Trans. on Computer Systems, 2017, 35(1): Article No.37.

[22] Shasha D, Llirbat F, Simon E, Valduriez P. Transaction chopping: Algorithms and performance studies. ACM Trans. on Database Systems, 1995, 20(3): 325–363. [doi: 10.1145/211414.211427]

[23] Kalia A, Andersen D, Kaminsky M. Challenges and solutions for fast remote persistent memory access. In: Proc. of the 11th ACM Symp. on Cloud Computing (SoCC 2020). New York: Association for Computing Machinery, 2020. 105–119. [doi: 10.1145/3419111.3421294]

[24] Stonebraker M, Madden S, Abadi DJ, Harizopoulos S, Hachem N, Helland P. The end of an architectural era: It's time for a complete rewrite. In: Proc. of the 33rd Int'l Conf. on Very Large Data Bases (VLDB 2007). 2007. 1150–1160.

[25] The H-Store Team. SmallBank Benchmark. 2022. http://hstore.cs.brown.edu/documentation/deployment/ benchmarks/smallbank/

[26] The Transaction Processing Council. TPC-E Bench-mark V1.14, 2022. http://www.tpc.org/tpce/

[27] INTEL. Intel® memory latency checker v3.7. 2019. https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[28] Gray J, Reuter A. Transaction Processing: Concepts and Techniques. Elsevier, 1992.

[29] Eswaran KP, Gray JN, Lorie RA, Traiger IL. The notions of consistency and predicate locks in a database system. Communications of the ACM, 1976, 19(11): 624–633. [doi: 10.1145/360363.360369]

[30] Xu J, Swanson S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In: Proc. of the 14th USENIX Conf. on File and Storage Technologies (FAST 2016). 2016. 323–338.

[31] Fan PH, Huang GR, Jin PQ. NVRC: Write-limited logging for non-volatile memory. Compute Science, 2021, 48(3): 130–135 (in Chinese with English abstract).

[32] Xu CH, Yan JF, Wan H, Sun FY, Zhang WG, Li T. A survey on security and privacy of emerging non-volatile memory. Journal of Computer Research and Development, 2016, 53(9): 1930–1942 (in Chinese with English abstract).

[33] Luo YP, Jin PQ. Optimizing join algorithms for NVM+DRAM-based hybrid memory architecture. Chinese Journal of Computers, 2020, 43(6): 1069–1085 (in Chinese with English abstract).

[34] Venkataraman S, Tolia N, Ranganathan P, Campbell RH. Consistent and durable data structures for non-volatile byte-addressable memory. In: Proc. of the FAST, Vol.11. 2011. 61–75.

[35] Zuo P, Hua Y, Wu J. Write-optimized and high-performance hashing index scheme for persistent memory. In: Proc. of the 13th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2018). 2018. 461–476.

[36] Yang J, Wei Q, Chen C, Wang C, Yong KL, He B. NV-tree: Reducing consistency cost for NVM-based single level systems. In: Proc. of the 13th USENIX Conf. on File and Storage Technologies (FAST 2015). 2015. 167–181.

[37] Kannan S, Bhat N, Gavrilovska A, Arpaci-Dusseau A, Arpaci-Dusseau R. Redesigning LSMs for nonvolatile memory with NoveLSM. In: Proc. of the 2018 USENIX Annual Technical Conf. (USENIX ATC 2018). 2018. 993–1005.

[38] Shu JW, Lu YY, Zhang JC, Zheng WM. Research progress on non-volatile memory based storage system. Science & Technology Review, 2016, 34(14): 86–94 (in Chinese with English abstract).

[39] Wu ZW, Zhang W. A concurrent linked list based on hardware transactional memory. Computer Engineering & Science, 2018, 40(S1): 154−158 (in Chinese with English abstract).

[40] Wu SG, Wu D, Pang ZB, Yang XD. HybridTCache: Tightly coupled hybrid transactional memory system to support efficient unbounded transactions with strong isolation. Chinese Journal of Computers, 2008, 31(11): 1907−1917 (in Chinese with English abstract).

[41] Zeng K, Yang XJ. A best-efort hardware transactional memory based on dependency graph. Journal of Computer Research and Development, 2012, 49(1): 44−54 (in Chinese with English abstract).

[42] Yu QQ, Dong MK, Chen HB. Hardware transactional memory assisted synchronization mechanism in virtualized environment. Journal of Frontiers of Computer Science and Technology, 2017, 11(9): 1429−1438 (in Chinese with English abstract).

[43] Genç K, Bond MD, Xu GQH. Crafty: Efficient, HTM-compatible persistent transactions. In: Proc. of the 41st ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2020). New York: Association for Computing Machinery, 2020. 59−74. [doi: 10.1145/3385412.3385991]

[44] Castro D, Romano P, Barreto J. Hardware transactional memory meets memory persistency. In: Proc. of the 2018 IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS). 2018. 368−377. [doi: 10.1109/IPDPS.2018.00046]

附中文参考文献:

[31] 范鹏浩, 黄国锐, 金培权. NVRC: 一种面向 NVM 的写限制日志方案. 计算机科学, 2021, 48(3): 130−135.

[32] 徐远超, 闫俊峰, 万虎, 孙凤芸, 张伟功, 李涛. 新型非易失存储的安全与隐私问题研究综述. 计算机研究与发展, 2016, 53(9): 1930−1942.

[33] 罗永平, 金培权. NVM+DRAM 混合内存架构下的连接算法优化. 计算机学报, 2020, 43(6): 1069−1085.

[38] 舒继武, 陆游游, 张佳程, 郑纬民. 基于非易失性存储器的存储系统技术研究进展. 科技导报, 2016, 34(14): 86−94.

[39] 吴振伟, 张文喆. 基于硬件事务内存构建并发链表. 计算机工程与科学, 2018, 40(S1): 154−158.

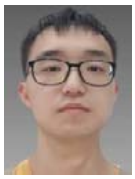
[40] 王绍刚, 吴丹, 庞征斌, 杨晓东. HybridTCache: 一种基于专用事务 Cache 的软硬件协同事务内存系统. 计算机学报, 2008, 31(11): 1907−1917.

[41] 曾坤, 杨学军. 基于依赖图的硬件事务存储技术研究. 计算机研究与发展, 2012, 49(1): 44−54.

[42] 余倩倩, 董明凯, 陈海波. 虚拟环境下硬件事务内存辅助的同步机制. 计算机科学与探索, 2017, 11(9): 1429−1438.

魏星达(1992－), 男, 博士, 助理教授, CCF 专业会员, 主要研究领域为操作系统, 分布式系统.

陈海波(1982－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为操作系统, 并行与分布式系统.



陆放明(1998－), 男, 学士, 主要研究领域为操作系统, 分布式系统.

臧斌宇(1965－), 男, 博士, 教授, CCF 杰出会员, 主要研究领域为操作系统.

陈榕(1981－), 男, 博士, 教授, CCF 杰出会员, 主要研究领域为操作系统, 分布式系统.