International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2022, 12(1): 31–53, doi: 10.21655/ijsi.1673-7288.00274

International Journal of Software and Informatics

Research Article

Reducing Transaction Processing Latency in Hardware Transactional Memory-based Database with Non-volatile Memory

Xingda Wei (魏星达), Fangming Lu (陆放明), Rong Chen (陈榕),
Haibo Chen (陈海波), Binyu Zang (藏斌宇)

(Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai 200240, China)
Corresponding author: Rong Chen, rongchen@sjtu.edu.cn

Abstract The emergency of Hardware Transactional Memory (HTM) has greatly boosted the transaction processing performance in in-memory databases. However, the group commit protocol, aiming at reducing the impact from slow storage devices, 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. In this paper, we propose 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. We have integrated this technique into DBX, a state-of-the-art HTM-based database, and propose DBXN: a low-latency and high-throughput in-memory transaction processing system. Evaluations using typical OLTP workloads including TPC-C show that it has 99% lower latency and 2.1 times higher throughput than DBX.

Keywords non-volatile memory; in-memory transaction; hardware transactional memory; OLTP

Citation Wei XD, Lu FM, Chen R, Chen HB, Zang BY. Reducing transaction processing latency in hardware transactional memory-based database with non-volatile memory. *International Journal of Software and Informatics*, 2022, 12(1): 31–53. http://www.ijsi.org/1673-7288/274.htm

Modern database applications including the official 12306 Chinese train ticket booking system require transactions to process the data correctly and reliably. With the popularity of information technology, database applications are carrying more and more traffic, so people increasingly need high-performance transaction processing systems. In recent years, the rise of multi-core processors and the expansion of memory capacity gave birth to some high-throughput multi-core memory databases[1–6]. Meanwhile, the new processor feature of Hardware Transactional Memory (HTM) further pushes the transaction throughput of multi-core in-memory databases to a new peak[1, 7, 8].

Although existing HTM-based multi-core in-memory databases can provide extremely high throughput, they still have high transaction commit latency. They commit transactions in a batch

This is the English version of the Chinese article ‘‘基于非易失性内存和硬件事务内存的低时延事务处理. 软件学报, 2022, 33(3): 849–866. doi: 10.13328/j.cnki.jos.006444”.
Funding items: National Key Research and Development Program of China (2020YFB2104100); National Science Fund for Distinguished Young Scholars of China (61925206)
Received 2021-07-01; Revised 2021-07-31; Accepted 2021-09-13; IJSI published online 2022-03-28

fashion so that transactions can avoid the influence of slow storage devices without compromising durability[1, 3, 4]. However, batch commit introduces an order of magnitude higher latency to in-memory transactions. For example, in DBX[1], a typical HTM-based multi-core in-memory database, the difference in commit latency between durable and non-durable transactions is nearly 18,000 times (9 μs vs.160 ms; see Section 2.2 for details). Low latency and high throughput are equally important for transaction processing. For example, Amazon has revealed that each 100 ms increase in request latency results in a 1% loss in economic benefits[9].

Recently, Intel has finally released the first commercially available Non-Volatile Memory (NVM) device, 3D-XPoint[10, 11]. With the help of NVM, this paper answers a natural question: can we use it to significantly reduce the transaction commit latency of HTM-based multi-core in-memory databases while retaining the benefits of HTM? Besides high performance, NVM provides the same interface as DRAM, which means that systems can read from and write to NVM directly in HTM. Therefore, it is possible to combine both hardware features to accelerate in-memory databases.

In exploring the use of NVM to reduce the latency of HTM-based transaction commits, we encountered two key challenges (Section 2.3). First, NVM does not collaborate with HTM: writing to NVM can always abort HTM. Second, the performance characteristics of the real NVM are completely different from those of in-memory database and difficult to reproduce by simulators[12, 13]. Therefore, fully exploiting the high performance of NVM requires a co-design with transaction's implementation.

Since real NVM hardware only emerged in the last two years, existing databases designed for HTM and NVM do not fully consider their hardware characteristics. For example, PHyTM[14] assumes that NVM can be written in HTM, while real hardware cannot. On the other hand, existing NVM-based persistent transactional memory systems are not designed specifically for the transactions, which mainly focus on providing a universal transactional programming interface[15–17]. As a result, these systems do not achieve the best performance in the database scenario (Section 4.3). To the best of our knowledge, this paper is the first work to reduce transaction processing latency of in-memory databases by systematically analyzing the combination of real NVM and HTM.

To address the problem that HTM and NVM cannot collaborate, we propose a mechanism called parity version (Section 3.1.2), which separates the operations of NVM and HTM in transaction processing from the software level so that transactions can be accelerated by both hardware technologies. Thanks to the fact that the parity version can be implemented by reusing the concurrency control mechanism of existing HTM-based transaction processing (Section 3.1.3), it has only a minor overhead. Finally, we also perform a series of implementation optimizations for real HTM and NVM hardware characteristics (Section 3.4).

We apply the above techniques to DBX[1], a state-of-the-art HTM-based in-memory database, and propose DBXN, an in-memory transaction system taking advantage of both HTM and NVM features. DBXN achieves both high-throughput and low-latency transaction processing. On the one hand, DBXN reduces the DBX latency by 99% in the typical database transaction scenario TPC-C[18] and also has 2.1 times higher throughput. On the other hand, DBXN can provide 65% better throughput than existing transactional durable memory systems based on real NVM hardware (Pisces[15]) in the database scenario.

In summary, we make the following contributions:

(1) a systematically analysis on how to use NVM with HTM (Section 2);
(2) an NVM and HTM friendly transaction protocol (Section 3);
(3) DBXN, a high-throughput low-latency multi-core in-memory database that utilizes the features of HTM and NVM (Section 3);

(4) DBXN can significantly reduce the transaction commit latency and improve the transaction throughput of existing HTM-based multi-core in-memory databases in typical transaction processing benchmarks, e.g., TPC-C (Section 4).

1 Background Knowledge

1.1 HTM

It is challenging for developers to manually ensure the Atomicity, Consistency, and Isolation (ACI) of memory reads and writes for concurrent programs, such as by using fine-grained locking, which is error-prone and introduces additional software overheads. HTM[19] greatly simplifies the writing of concurrent programs and avoids the software concurrency control overheads by ensuring the ACI properties at the CPU level. Since Intel released the new processor feature called Restricted Transactional Memory (RTM)[1], HTM has become a reality. Specifically, RTM provides new instructions: `xbegin`, `xend`, and `xabort`. The program can use `xbegin` and `xend` to indicate the beginning and the end of HTM, respectively, and `xabort` to abort HTM execution. RTM is internally implemented by Optimistic Concurrency Control (OCC)[20]: the processor uses its cache to record the set of reads and writes of program memory and detects conflicting memory accesses with the help of a cache coherence protocol.

Though RTM simplifies application development and opens up optimization opportunities, it has the following limitations due to the limited hardware resources[1]: first, a limited set of instructions is allowed to execute within HTM. For example, HTM programs cannot make system calls. Second, its read-write sets capacities are limited by the cache size: the current implementation uses the L1 cache to record the write-set and uses the L2 cache to record the read-set. When the size of the program's read and write memory exceeds the size of the cache, the HTM will be aborted.

1.2 HTM-based multi-core memory database

By observing that the high-performance ACI properties offered by HTM match the ACI properties in ACID transaction processing of multi-core in-memory databases, people began to explore how to improve the performance of transaction processing with HTM[1, 7, 8, 21]. Figure 1 shows an example of using HTM to accelerate transactions. In this example, the transaction updates all data greater than 10 in column `x` of Table `A` in the database to 10. As displayed in the left part of the figure, the system can leverage HTM to ensure the ACI properties of the transaction by including the transaction logic in `xbegin` and `xend`.

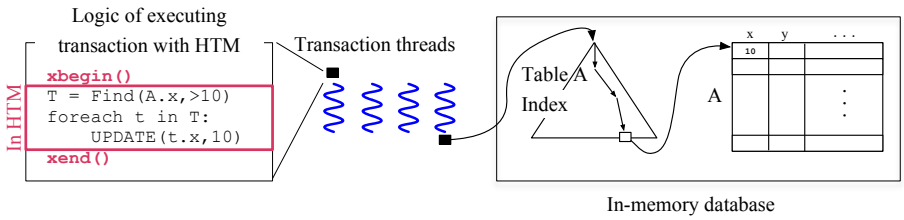


Figure 1 An overview of using HTM to accelerate transaction execution in a multi-core in-memory database

Although HTM can efficiently ensure the ACI properties, since the database transaction logic is relatively complex, the direct execution of the transaction with a complete HTM (Figure 1) suffers from HTM hardware limitations (Section 1.1) and thus cannot fully exploit the

[1]Since this paper focuses only on RTM, which is the mainstream HTM implementation, HTM herein refers to RTM unless directly stated.

high performance of HTM[21]. As shown in Figure 1, the transaction first needs to query the index of Table `A` in order to find data greater than 10 in column `x` of Table `A`. Usually, there are many memory accesses for index operations (e.g., B+Tree traversal). Thus, the transaction is likely to be aborted due to the memory size limit of HTM access. Since it is inefficient to directly use HTM to accelerate transactions in general cases, works have been made to mitigate HTM hardware limitations by imposing a lightweight software concurrency control layer atop of HTM[1, 7, 8]. For example, DBX[1] uses OCC[20] to divide the transaction that requires HTM protection so that the split part of the program can be effectively accelerated by HTM. Similarly, DrTM[8] adopts transaction chopping[22] for the segmentation, and the segments can be accelerated using HTM. Note that executing transactions using HTM cannot ensure durability. Thus, all the above systems further employ logging to provide durability. We describe logging in Section 2.1.

1.3 NVM

With the release of Optane DC persistent memory[11][2] by Intel, NVM has been commercially available to the pubic market. Compared to traditional durable devices (such as HDD and SSD), NVM offers lower latency and higher bandwidth. Moreover, NVM is attached to the processor's memory bus, so it is byte-addressable, that is the processor can read from and write to it directly using `load`, `store`, and `nt-store`[3] instructions, just like DRAM. There are durability issues with reading from and writing to NVM using traditional memory instructions. For example, memory write instructions (e.g., `store`) will first write to the processor's volatile cache. To support durability, Intel further provides an extended instruction set: the processor can use `clwb` and `clflush` to flush data from the cache to NVM and can use `sfence` to wait for the completion of the flush operation.

Although NVM provides the same interface as memory, its performance characteristics are completely different. To make better use of NVM, researchers have investigated its performance characteristics in depth[12, 13, 23]. In summary, when a processor reads and writes NVM, it needs to consider the following four performance characteristics. First, NVM has asymmetric read and write performance. Its read bandwidth is much higher than its write bandwidth[12, 13]. Second, NVM uses a different read and write granularity than that of the processor. NVM reads and writes at a granularity of 256 bytes, while the processor uses 64 bytes. Therefore, writing NVM with big payloads (e.g., larger than 64 bytes) should be performed at the granularity of 256 bytes to fully utilize NVM's bandwidth. Third, processors' read requests for NVM can affect the throughput of write requests[13]. Therefore, applications should avoid asking processors to send unnecessary read requests to NVM. For example, when a processor's write request size is less than 64 bytes (i.e., it does not satisfy the processor's read/write granularity), the processor would send an additional read request to the NVM before writing. To avoid this, the processor should transmit the small messages (less than 256 bytes) at a granularity of 64 bytes[12]. Finally, the processor's cache is not NVM friendly[13, 23]. Therefore, using `nt-store` to write to NVM can achieve better performance than `store`.

2 Analysis on Existing Drability Mechanisms and Challenges of Using HTM with NVM

In this section, we first introduce the existing durability mechanism of HTM-based multi-core in-memory databases (Section 2.1). We then experimentally analyze the problems of this mechanism in the new NVM scenario (Section 2.2).

[2]Since Intel's Optane DC persistent memory is the only available NVM, we use NVM as a synonym for the Optane memory in this paper.

[3]The nt-store is a special form of store, which additionally bypasses the processor's cache.

2.1 Existing durability mechanism for HTM-based multi-core in-memory databases: Redo log-based group commit

To support durable transactions in HTM-based multi-core in-memory databases, existing systems use redo logs to store the results of transactions in durable devices[3]. Writing redo logs to a traditional durable device (e.g., SSD) introduces an overhead much larger than the transaction execution time[23]. For example, in a DBX[1] system, adding a write operation to the disk after a transaction has been executed causes a 98% performance loss (compared with a transaction that does not support durability). Therefore, HTM-based multi-core in-memory databases or multi-core in-memory databases both use group commit to amortize the durability overhead[1–4, 7, 12]. Group commit groups transactions in epochs (time slices) and commits all transactions within an epoch asynchronously in a batched fashion, where epoch is the duration between each group commit operation. This approach effectively reduces the overhead of logging to non-volatile devices. However, the asynchronous commit process significantly increases the transaction latency.

Figure 2 shows a specific example of executing a transaction in HTM using group commit. When execution threads start transactions, they first obtain the current database epoch number (①). Thus, when the transaction is committed later (i.e., when `xend()` is executed), the execution threads can record the log of the transaction in the log area corresponding to the epoch number (②). Noteworthily, since the log has not been synchronized to the durable device (e.g., NVM) at this point, the transaction has not yet completed its commit. The transaction log is synchronized to the durable device asynchronously by a background thread (the log thread). This thread periodically obtains the current epoch number and synchronizes the logs of all transactions in this epoch (③). The specific operations are as follows: first, the log thread updates the epoch number of the current system (④) to prevent future transactions from writing logs to that epoch. Second, it starts collecting the logs of the relevant transactions (⑤). Before collecting the logs, the thread needs to wait for all transactions in the current epoch to be finished (or aborted). Finally, the log thread writes the collected transaction logs to the durable device via a single IO operation (⑥). From this flow, we can see that a single persistent operation can write the logs of all transactions in an epoch. Therefore, group commit can effectively amortize the overhead of transaction persistent operations.

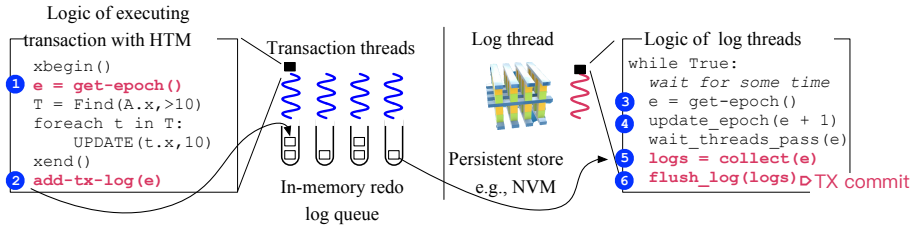
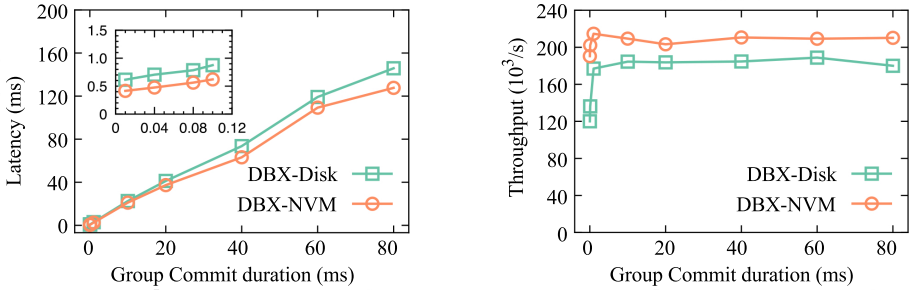


Figure 2 An overview of using redo log-based group commit to support durable transactions

Since a transaction is considered committed only when its logs are synchronized to the durable device (i.e., the execution satisfies ACID), the commit delay of group commit is directly related to the synchronization duration (i.e., the epoch) of the log thread. In general, systems should select a large epoch to commit more transactions in a batch fashion as far as possible. For example, Silo[4] and DBX[1], two typical in-memory databases, both set the epoch to 40 ms by default. However, large epochs can introduce high latency to transactions. Reducing the epoch improves the latency, but it affects the throughput: the log thread will block the transaction log in the current epoch because it must wait the logs in the previous epoch to flush to the storage device. In the next section (Section 2.2), we analyze this phenomenon in depth via experiments.

2.2 Asynchronous nature of group commit significantly increases transaction processing latency

To analyze the latency impact of group commit on HTM-based multi-core in-memory databases, we conduct experiments on DBX[1], a representative system. We use TPC-C[18], the standard OLTP benchmark for the evaluation. For each test, we configure the DBX to use its peak performance configurations[1][4]: the database uses 10 transaction threads and 1 log thread. To avoid the log thread being affected by slow storage devices, we evaluate DBX performance with both disk[5] (DBX-Disk) and NVM (DBX-NVM). In DBX-NVM, the log thread writes the transaction logs to the file system supported by NVM (which uses Ext4-DAX mode). Figure 3 shows the results, from which we can draw the following two observations.



(a) Latency (b) Throughput

Figure 3 Effect of duration of group commit (epoch) on latency and throughput in DBX

Group commit brings a significant latency increase to HTM-based transaction processing. Since a transaction must wait until all other transactions in an epoch are finished before its logs are written to the durable device, the latency of a transaction is strongly correlated with the duration of group commit (epoch). As shown in Figure 3(a), when the epoch is 1 ms, the latency of DBX transaction commit is 2.92 ms; when the epoch is 80 ms, the latency can be as high as 145 ms. As we mentioned earlier, the duration of the group commit is generally much longer than the execution time of the transaction so that the effect of the group commit can be maximized. For example, the epoch is 40 ms in DBX[1], and the same epoch is adopted by other typical multi-core in-memory databases[3, 4]. This epoch is much longer than the transaction execution time in mainstream transaction processing scenarios, as listed in Table 1, where transactions take only 0.26 μs and 9.3 μs to complete execution in Smallbank[25] and TPC-C[18] scenarios, respectively. Even for the TPC-E[26] scenario, which has a long execution time, it takes only 310 μs to complete the execution in the in-memory database. Therefore, transactions in common transaction scenarios have significantly longer latency when group commit is adopted to commit transactions. Finally, NVM helps little for group commit. In Figure 3(a), the latency of DBX-NVM is only 32% less than that of DBX-Disk at most. This is because the asynchronous mechanism of the software is a major factor in the latency increase.

Reducing epoch can shorten the latency of transactions. For example, in Figure 3(a), the latency of a transaction is only 0.62 ms when the epoch is set to 0.01 ms. However, a too small epoch would significantly affects the transaction throughput. For small epochs, the throughput of TPC-C reduces at most 44%. Based on these results, we can make another observation.

[4]The configuration has been confirmed by the original authors. The specific experimental configuration is described in Section 4.1.

[5]The disk is a Seagate ST1200 MM0088 1.5 TB mechanical hard disk.

Table 1 Execution time of the transaction test benchmarks

Test benchmark	Execution latency (μs)
TPC-C[18]	9.3
TPC-E[26]	310
Smallbank[25]	0.26

Shortening the duration of the group commit to reduce the transaction latency lowers the transaction throughput. As shown in Figure 3(b), the transaction throughput of DBX-NVM is affected by 23% when the epoch is 0.1 ms. This is because when the log thread is affected by the slow persistent operation, it blocks the concurrently executed transactions, thus triggering a drop in the system throughput. Besides, even with a rather small duration of group commit in DBX (e.g., 0.01 ms), the transaction durability latency is still much higher than the transaction execution time (0.62 ms vs. 9 μs for TPC-C). This is because the latency of asynchronous/synchronous operations cannot be reduced by the duration of the group commit.

In summary, existing HTM-based multi-core in-memory databases have high latency when supporting durable transactions, and it needs to sacrifice latency to support high-throughput durable transactions. Meanwhile, their durability mechanisms are not suitable for acceleration with new media such as NVM.

2.3 Reducing transaction durability latency with NVM and the challenges of combining HTM with NVM

Observation 1: NVM's low latency and high throughput can efficiently support synchronous log operations.

First, NVM provides extremely low-latency persistent operations. As shown in Table 2, when performing 64-byte durable writes, NVM takes only 171 ns for sequential writes and 510 ns for random writes. Such latency is comparable to that of DRAM and much lower than the access latency (in the μs level) of a traditional durable device and the execution time of a typical transaction (see Table 1, where the execution latency of a TPC-C transaction is 13 μs). Therefore, transactions can synchronously persist transaction logs, i.e., by writing logs directly to NVM after execution. We conducted preliminary experiments to verify the feasibility of synchronous log writing on NVM. In the experiments, we add an additional NVM write operation after the DBX transaction is executed. The test results show that adding an additional NVM write only brings an 11% reduction to the DBX transaction throughput. In contrast, writing to disk causes a 98% reduction in transaction throughput.

Table 2 Statistics of 64-byte access latency in NVM and DRAM

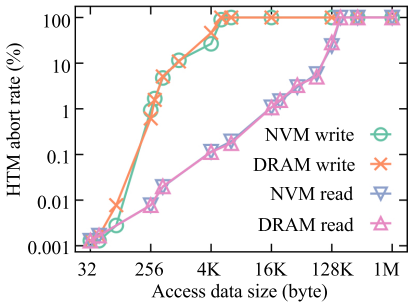
Device access latency	Sequential write (ns)	Random write (ns)	Sequential read (ns)	Random read (ns)
NVM	171	510	248.06	276.6
DRAM	153	234	82.0	82.2

Note: We use Intel's memory latency checker[27] to obtain the read latency of each device and use an LAT tester similar to the one in the work of Yang *et al.*[13] to measure the write latency.

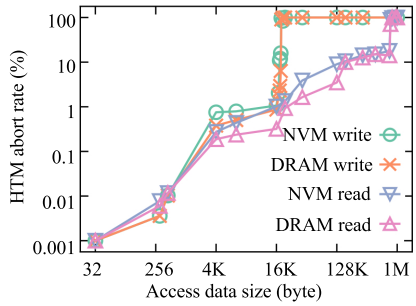
Observation 2: HTM can be accessed efficiently in NVM.

Further, we find that accessing NVM in HTM has a similar performance to accessing DRAM. Although the access latency of NVM is larger than that of DRAM (see Table 2), the main operations of HTM work in the processor cache. Thus, HTM does not interact with the NVM frequently, except for reading data into the cache. To illustrate this, we replicate the original evaluations of DBX[1] on HTM access performance except that we use NVM. As shown in Figure 4(a) and (b), HTM's access to NVM and access to DRAM have close abort rates in

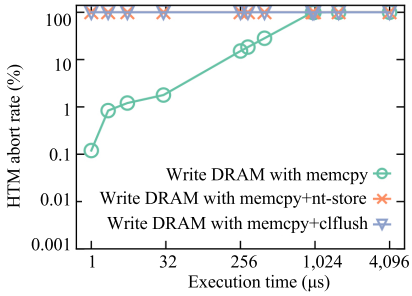
random read/write scenarios. This rate is limited by the processor cache size, which is the main factor affecting the HTM performance. In sequential read/write scenarios, NVM has a 3%–20% higher abort rate than DRAM at 16 B–256 KB, while they are close in performance at other access sizes. This difference is mainly due to the higher read access latency of NVM (250 ns vs 82 ns), but it is not significant.



(a) Comparisons of HTM abort rate when increasing the number of random DRAM/NVM reads



(b) Comparisons of HTM abort rate when increasing the number of sequential DRAM/NVM reads



(c) Comparisons of HTM abort rate when increasing the program execution time with different instructions

Figure 4 HTM performance feature studies in consideration of NVM

Based on the above two observations, an naive idea is to combine HTM and NVM organically to execute transactions, leveraging the high-performance transactional ACI properties of HTM and the high-performance transactional D property (durability) of NVM. Figure 5(a) shows an examplee. After a transaction executes the transaction logic (`do-tx-logic`) in HTM (the program area marked with `xbegin` and `xend`), it uses `add-tx-log-to-nvm` to write the transaction logs persistently to the NVM. However, this approach faces two main challenges.

Challenge 1: The persistent operation of NVM cannot be performed in HTM.

To ensure that data is written persistently to NVM, the processor needs to execute the `nt-store` and `clflush` instructions (Section 1.3). However, `nt-store` and `clflush` always abort the HTM. As shown in Figure 4(c), when the processor executes these two instructions in the HTM, HTM is bound to abort regardless of how long the HTM is executed or whether there are memory accesses. The reason is that these two instructions involve the processor cache mechanism, on which the current HTM implementation relies (Section 1.1). Specifically, HTM detects memory access conflicts between different cores with the help of the processor

cache coherence mechanism. Since `nt-store` bypasses the cache by design, HTM cannot ensure the correctness of the program executed under this instruction and therefore has to be aborted. Similarly, `clflush` flushes uncommitted data from the cache back into memory, so this operation aborts HTM.

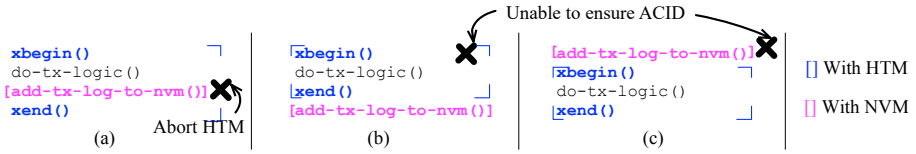


Figure 5 Examples of naively using HTM and NVM to boost transaction processing. In (a), the persistent write to NVM will cause 100% HTM abort. In (b) and (c), the executions cannot guarantee ACID properties

To ensure the atomicity and durability of data modification by transactions, we must ensure the durability of NVM operations in HTM. If we do not put the persistent operation in the HTM, other transaction threads will observe a piece of uncommitted data. For example, in Figure 5(b), the log is not written to the NVM after being committed by HTM. At this point, other transactions will read a piece of uncommitted data. In Figure 5(c), if the transaction executed in HTM is aborted, the system writes a redo log of the uncommitted transaction.

In DBXN, we use the parity version mechanism (Section 3.1.2) to properly distinguish between transaction execution in HTM and write operations to NVM.

Challenge 2: Exploiting the high performance of the hardware requires co-optimization for NVM.

NVM has completely different performance characteristics from those of DRAM[12, 13, 23]. If it is directly accessed as DRAM without special optimization, the system can only obtain 33% of the original NVM performance[12]. Therefore, how to apply NVM-related optimization (Section 1.3) is crucial for HTM-based and NVM-based multi-core in-memory databases.

3 Design and Implementation of DBXN

(1) Objective

The goal of DBXN is to reduce the transaction commit latency of the existing HTM-based multi-core in-memory databases with NVM. To achieve this objective, we need to find (i) an efficient approach to combine NVM and HTM and (ii) a transaction commit implementation designed with real NVM features.

(2) Overview

DBXN adopts two key techniques: (i) HTM-NVM-friendly log mechanism (Section 3.1.2) based on parity version and (ii) real NVM-friendly log writing method (Section 3.4). (a) The parity version efficiently and correctly distinguishes HTM operations from NVM write operations, thus breaking the limitation of being unable to durably write to NVM in HTM (Section 2.3). (b) Log writing is optimized according to the nature of the real NVM hardware, which affects the performance of main interaction operations between transaction commit and NVM.

We build DBXN atop of DBX[1], an existing state-of-the-art HTM-based multi-core in-memory database. Its system architecture is shown in Figure 6. Similarly to the traditional memory-based multi-core in-memory databases[1, 2, 4, 7], data (including data tables and indexes corresponding to tables) are stored in memory. When the host's memory is insufficient, DBXN stores the data in NVM. A storage layer abstracts all the storage-related operations and assumes that it provides a key-value store interface. Upon receiving a transaction request, the transaction execution thread begins executing the transaction. The execution process includes reading and

writing the data required by the transaction through the storage layer and executing the DBXN protocol to ensure the ACID properties of the transaction execution. DBXN applies the parity version to DBX's HTM-friendly OCC protocol to execute transactions, which consists of the following phases. The execution and validation phases (① and ②) ensure the ACI properties of the transaction, while the logging phase (③) and commit phase (④) ensure the D property of the transaction. Finally, unlike DBX, DBXN does not require a background log thread to persist the transaction log to NVM[1, 3, 4]. In the logging phase (③), DBXN writes the log directly to NVM in a synchronous manner.

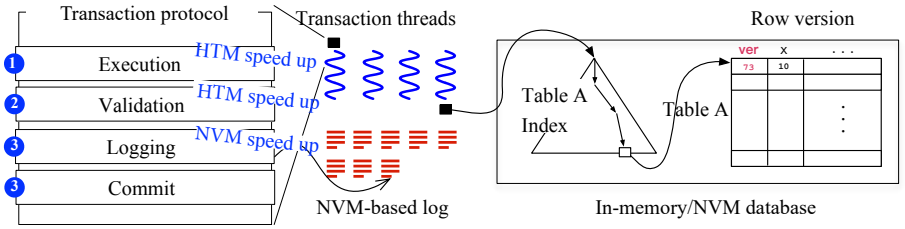


Figure 6 Architecture of DBXN. The detailed transaction execution protocol is shown in Figure 7 and Figure 8

3.1 Parity version and DBXN commit protocol

In this section, we describe the parity version technique and how it is applied in DBXN to execute the protocol for transactions. This section is organized as follows. First, we describe the data structure of the database records in DBXN (Section 3.1.1). According to this data structure, we describe the parity version mechanism (Section 3.1.2). Second, we describe DBXN's transaction protocol based on DBX's OCC and parity version in Section 3.1.3. Third, we give an informal argument of DBXN correctness (Section 3.2) and how it supports common database operations (Section 3.3). Finally, we describe DBXN optimization for real hardware (Section 3.4).

3.1.1 *Data structure of database records in DBXN*

To support efficient transaction processing, DBXN stores database records in the same way as DBX with row storage.

- **Data version:** Represented by 8B unsigned integer, DBXN's protocol synchronizes information about concurrent transaction access and transaction durability between different threads through this version.
- **Data content pointer:** 8B virtual address, which points to the data stored in memory or NVM.

When a transaction modifies data, DBXN updates the data by swapping pointers. The transaction allocates a new memory area, copies the modified data into the area, and finally replaces the data content pointer with the newly allocated address. This design can effectively reduce the number of memory writes, especially for transactions that modify data in HTM; about the size of the modified data, a transaction to modify a piece of data requires only 8 bytes (the size of a pointer) of memory or NVM to be written. Since the HTM abort rate is strongly correlated with the size of memory accessed by the program (Figure 4), this design is effective in reducing the HTM abort rate.

3.1.2 *Parity version mechanism*

To utilize both HTM and NVM, the system can only be executed in the way shown in Figure 5(b) or Figure 5(c). Since the database writes the log of an uncommitted transaction

according to Figure 5(c), DBXN takes the approach in Figure 5(b) which can avoid writing NVM in HTM. To ensure ACID properties, we need to prevent transactions from reading data from transactions whose logs have not been written yet based on Figure 5(b)'s approach.

Parity version determines whether the transaction's modified data has completed log durability by version checking. The data version is divided into two states: odd version and even version. An odd version indicates that the data has passed the ACI inspection of HTM, but has not yet completed the persistent operation. As in Figure 5(b), the versions of the modified data in `xbegin` and `xend` are odd. An even version indicates that the persistent operation of data has been further made durable. When the transaction finishes logging, DBXN changes the data modified by the transaction to an even number. When other transactions read data in an odd version, they need to wait for it to become even before committing. These waits prevent transactions from reading data of log that has not been written and thus ensuring the ACID property of the transction's execution.

Specifically, assume that all data is initialized with even versions. When a transaction modifies data in the HTM, it increments the corresponding version of the data by one, i.e., making it an odd version of data that is to be committed but has not yet completed the persistent operation. Once the increment is completed, the transaction can exit the HTM and start writing a log to the NVM. If another transaction reads the data in an odd version, it needs to use `xabort` to be aborted and retried. Once the transaction writes the log to the NVM, it needs to add one again to the version of the data it modified, i.e., raising the version to an even version. At this point, other transactions can correctly read the transaction's modifications. Finally, the system needs a specific mechanism to synchronize the operations between concurrent transactions when they read and write data and perform version checking. Since we can perform all data reads/modifications and version checking in HTM, there is no need for locking and the ACI properties of HTM can be used to synchronize concurrent version checking.

Version checking in HTM slightly increases the read-set of the transaction. However, the actual impact is negligible. First, HTM can support a larger set of reads (Figure 4(a) and Figure 4(b)). Second, the operation can reuse the checking mechanism of OCC in DBX. We will describe this in detail in the next section.

3.1.3 *Basic transaction protocol of DBXN*

In this section, we describe how to apply the parity version to the DBX transaction execution protocol. We assume that the transaction has only read and write operations and that the data it accesses must exist in the database. In Section 3.3, we will further discuss how it supports common database operations such as insert-delete and read-not-existing. Figure 7 and Figure 8 show the pseudo-code of the DBXN transaction execution. The red parts are DBXN extensions to the original DBX protocol.

(1) Execution context of transaction

When a transaction starts, DBXN assigns a context to it to record the metadata of its execution. The details are shown in Figure 7. ReadSet is a set of data read by the transaction; each item includes the key to the data read by the transaction, the version of the data at the time of reading, and a reference to that data (Section 3.1.1). WriteSet is a set of data written by the transaction; each item adds a pointer to the data modified by the transaction on the basis of ReadSet. Typically, all data in the WriteSet is also in the ReadSet. LogQueue is the transaction's log queue in NVM. As in the existing work, DBXN organizes the log content according to redo log[3, 4], which contains the keys of all data modified by the transaction, the version of the data committed by the transaction, and the newly modified content.

DBXN executes a transaction sequentially in execution, validation, logging, and commit phases. The logic of the transaction is executed in the execution phase; whether the result of

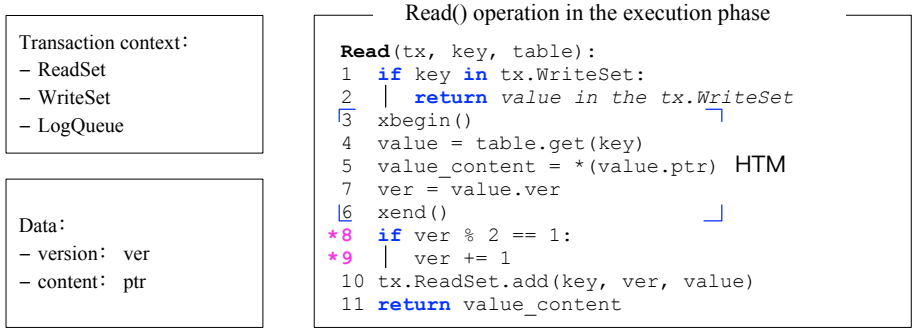


Figure 7 Data structure of transaction context used in DBXN and the pseudo-code of Read in the execution phase. Lines marked as red (*) are extensions to the original DBX protocol.

the transaction execution in the execution phase can be committed, i.e., whether it satisfies the ACI properties, is determined in the validation phase; the logging phase ensures that the result of the transaction is stored in NVM; the result of the transaction is committed at the end of the commit phase, namely that it can be seen by other transactions.

(2) Execution phase

In this phase, the transaction executes the logic of the user transaction by reading and writing the data it needs. For the read/write of each data, the transaction must call the specific `Read/Write` interface of DBXN to interact with the storage layer. The pseudo-code of `Read` is shown in Figure 7. The main operation of `Read` is to call the storage layer interface to read the value corresponding to the data key into the transaction (line 4). To support the inspection in the validation phase, the transaction also reads the current version of the data (value.ver). Since reading the data version and the data value must be atomic, DBXN uses HTM to protect the reading of the data value and version as DBX (lines 3–6). Finally, when the read is completed, the transaction adds the metadata of the data to its ReadSet (line 10). For write operations, DBXN caches the writes in the transaction's write set instead of updating the storage layer in place. First, DBXN reads the data by `Read` into the ReadSet, allocates a new block of memory to store the transaction's modifications, and finally caches the modifications in the write set. Since the logic of `Write` is similar to the existing OCC protocol, we omit its pseudo-code. Finally, since a transaction needs to read its write operation, `Read` will inspect whether the data to be read is in the transaction's write set (lines 1–2) when it will read. If yes, DBXN will directly return the data cached in the write set.

Unlike the DBX OCC, the DBXN transaction may read an odd version of the data, i.e., data whose persistent operation is not completed, in the execution phase (Section 3.1.2). Theoretically, the transaction needs to abort the OCC execution and be retried in this situation. Here, DBXN adopts an optimization that delays the checks to the validation phase. This is based on the observation that since NVM has low latency, the persistent operation of data is likely to be finished when the transaction reaches the validation phase. This avoids the need to abort the transaction early in the execution phase. To ensure that whether the persistent operation of read data has been completed can be correctly detected in the validation phase, DBXN further inspects whether the version of the data is odd in line 8 of `Read`. If it is odd, the transaction needs to additionally inspect whether the version of the data becomes even in the validation phase later. For this reason, DBXN adds one to the version (line 9) and puts it into the ReadSet. Thus, if the transaction still has an odd version of the data in the validation phase, the transaction will be aborted and retried because the OCC version is inconsistent. Details are shown in the description of the validation phase below.

(3) Validation phase

To determine whether a transaction can be committed, DBXN validates the transaction by means of the validation phase of OCC. Like DBX, DBXN determines whether a transaction meets the ACI properties by validating whether the transaction's read set has changed or not since the execution phase.

As shown in Figure 8, the transaction first inspects whether the latest version of the data in its read set matches the log version in the read set (line 2–3). If it does not, this indicates that either a concurrent transaction has modified the data read by the transaction (violation of the ACI properties) or the data has an odd version (violation of the D property). When either happens, DBXN will abort the transaction and retry it (line 4). If the data in the transaction read set has not been modified, the transaction will be executed in accordance with the ACID properties. In such a case, DBXN starts trying to commit this transaction. Before entering the logging and commit phases, DBXN first updates the database with the modifications of the data in the write set. For all data in the write set, DBXN points the content pointer of the data to the address assigned in the execution phase (line 7). Also, DBXN updates the version of the write data to an odd number (line 6) so that other transactions do not read its modifications and commit until the logging phase is completed.

```
Validate(tx):
1  xbegin()                                          ┐
2  for _, ver, value in ReadSet:                     │ HTM
3  |  if value.ver != ver:                          │
4  |  |  abort()                                     │
5  for _, _, value,n_ptr in WriteSet:                │
*6 |  value.ver += 1                                 │
7  |  value.ptr = n_ptr # omit GC for simplicity     │
8  xend()                                            ┘
```

```
Log(tx):
*1  for key, _, _, n_ptr in WriteSet:
*2  |  tx.LogQueue.add(key,n_ptr)
*3  tx.LogQueue.write_to_nvm()                        NVM
*4  tx.LogQueue.flush()
```

```
Commit(tx):
*1  for _, _, value, _ in WriteSet:
*2  |  value.ver += 1
```

Figure 8 Pseudo-code of validation, logging, and commit phase in DBXN. Lines marked as red (*) are extensions to the original DBX protocol

In OCC, the validation of the read set and the writing to the write set (i.e., lines 2–7) need to be performed in line with atomicity to ensure correctness. Usually, the implementation of this operation requires global or fine-grained locks for protection[1, 4]. However, locking operations introduce additional overhead. Like DBX, DBXN includes these operations in a single HTM (lines 1 and 8) to ensure atomicity efficiently with hardware.

(4) Logging phase

This phase ensures that the modifications made by the transactions are stored in a durable (i.e., NVM) log. In this way, the results of the transaction can be recovered from the durable log after downtimes. DBXN uses the mechanism of SiloR[3] for recovery, so this paper will not go into details. Since NVM has very low write memory latency, DBXN chooses to synchronize the log writing operation directly after the validation phase is completed. As shown in Figure 8, DBXN first organizes the modifications made to the database by the transaction into a redo log in Log (lines 1–2). After that, it writes the log to NVM (line 3). As writing directly to NVM cannot ensure that the log content is durable (e.g., data is written to the processor cache), DBXN further utilizes the extended instruction set of NVM after writing the log to ensure that the written content is flushed to NVM. On current hardware platforms, DBXN can use `clwb` and `sfence` to ensure that the data is written to NVM.

(5) Commit phase

When a transaction has completed the logging phase, its modifications are guaranteed to be durable. At this point, the transaction can be considered committed, and the transaction commit status is returned to the user. Meanwhile, to allow other transactions to read the modifications

made by this transaction, DBXN changes the version of the data written by this transaction to an even number in accordance with the parity version mechanism (lines 1–2 of Commit in Figure 8). It should be noted that the commit phase does not require HTM protection: this is because other concurrent transactions cannot commit when the data version of the transaction is odd. Thus, there are no concurrent writes that conflict with the transaction.

3.2 Correctness

Strict serializability is a desirable ACI property for ACID transactions. This section shows informally that DBXN ensures both strict serializability and durability of transactions.

3.2.1 *Strict serializability*

We illustrate that DBXN ensures strict serializability by reducing the protocol of DBXN to Strict Two-Phase Locking (S2PL)[28], a concurrency control protocol that ensures strict serializability. In S2PL, the transaction execution is divided into two phases. In the first phase, the transaction locks all read and write data (when that data is read). After the transaction is committed, it executes the second phase of S2PL, which releases all the locks applied in the first phase.

In DBXN, when a transaction is committed, the data it reads or writes can be reduced as applying locks in the execution phase (line 10 of Figure 7) and releasing locks in the commit phase (line 2 of `Commit` in Figure 8). This is because no other transaction can modify the data it reads or writes, and the effect is equivalent to that of applying locks. In DBXN, only when the data version changes from an even number to an odd number or from an even number to another even number will its contents be changed according to the parity version mechanism (Section 3.1.2). When a transaction reads data of an odd version (v), line 9 of `Read` and lines 3–4 of `Validate` ensure that the transaction will be only committed if the data version is $(v + 1)$ at the commit moment. The data read at this point has not been modified, as indicated by the parity version. When a transaction reads data of an even version, the transaction will be aborted whenever the data version changes (i.e., the contents change). Therefore, if a transaction is committed, the data it reads and writes will not be modified by other transactions during execution.

In DBXN, if a processor core commits a transaction, subsequent processors must be able to read the database modifications made by the transaction. Here, we assume that the `Commit` operation is completed. First, DBXN updates the database with the new contents of the data in HTM when the transaction is committed (line 7 of `Validate`). The HTM feature ensures that read operations of subsequent memory/NVM (of this processor or other processors) will definitely read its modifications. Second, DBXN will read the data directly from memory/NVM (line 4 of `Read`). Thus, according to the features of the previous HTM, when a processor commits a transaction, subsequent processors must read the modifications by this transaction.

3.2.2 *Durability*

To illustrate that DBXN supports durability, we just need to show that the transactions in it do not read the modifications by a transaction whose log is not durable. According to the pseudo-code of the logging phase shown in Figure 8, the transaction only becomes even after the log is flushed to the NVM (line 4 of `Log`). According to the parity version mechanism, only transactions that read an even version of data can be committed.

3.3 Support for common database operations

This section describes how to further support common database operations such as insert, delete, and range lookup based on the basic transaction protocol presented in Section 3.1.3. In DBXN, these operations are performed similarly to the original DBX

scheme. In other words, adding NVM-based low-latency transaction logs does not affect the execution of these operations in DBX.

3.3.1 *Read data not existing in the database*

When a transaction reads data that does not exist in the database, DBXN needs to detect its conflict with the transaction that inserts the data concurrently. DBXN makes use of the DBX storage layer's `get-with-insert` interface to facilitate the detection of this condition. As the name suggests, `get-with-insert` inserts null data corresponding to this data key when it encounters the data that does not exist. Thus, a concurrent transaction inserting this data is transformed into a conflicting write operation. Therefore, DBXN can detect concurrent insertion conflicts according to read/write conflicts in the validation phase. It should be noted that this insertion is not added to the DBXN log because it does not insert real data.

3.3.2 *Insert and delete*

When DBXN encounters an insert operation in the execution phase, it inserts null data at the storage level via `get-with-insert` and caches the inserted data in the transaction's write set. In this way, the insert operation can be considered as the proper execution of a normal write operation in the DBXN protocol. For a delete operation, DBXN cannot delete the data directly in the execution phase because the transaction may be aborted. Therefore, DBXN, like DBX, treats the delete operation as a write operation that writes a null pointer. This scheme causes memory waste in the index, and DBX uses the group commit based epoch mechanism for garbage collection. Since there is no group commit in DBXN, it just reuses the epoch mechanism of DBX for garbage collection.

3.3.3 *Range lookup*

Like DBX, DBXN relies on the B+ tree index in the storage layer to support range lookup. On the basis of the protocol given in Section 3.1.3, the support for range lookup requires additional detection of phantom[29]. Like DBX, DBXN recognizes this phenomenon by detecting in the protocol whether the leaf nodes involved in a transaction range lookup have changed. For purpose of supporting the detection, all leaf nodes of the B+ tree need to record an additional version number, which is the same as the data version (Section 3.1.1). When a leaf node is inserted or split, its version number is incremented. Also, when a transaction performs a range lookup, DBXN records all the leaf nodes of the B+ tree it visits in the read set. To ensure atomicity, DBXN uses HTM to protect the operations of accessing leaf nodes and reading version numbers. With these two mechanisms, DBXN can detect phantoms in the validation phase by detecting whether the version of the leaf nodes in the transaction read set has changed. It should be noted that DBXN does not record the modifications of the tree in the transaction redo log because index operations can be rebuilt during recovery. Therefore, the version number update of leaf nodes is a plus-2 operation. In other words, it changes from even to even, instead of adding 1 as DBXN does.

3.4 Implementation and optimization

We implemented DBXN based on the code of DBX[1]. There are two advantages of DBX-based implementation. First, DBXN can reuse DBX's high-performance storage layer, i.e., DBX-store. DBX-store uses an HTM-based B+ tree to support concurrent ordered key-value storage efficiently. In addition, DBXN can also reuse a series of implementation optimizations of DBX for HTM, such as fall back handler. On the basis of DBX, DBXN further makes a series of optimizations for NVM.

NVM-friendly log writing method: In Section 2.3, we have mentioned that designing in combination with NVM requires optimizing for real hardware characteristics. For this purpose,

we optimize the log writing operation in DBXN by referring to the existing research work for real NVM features[12, 13]. First, when DBXN writes logs to the NVM (line 3), DBXN uses `nt-store` instead of the traditional `memcpy` for writing. On the one hand, the characteristic of `nt-store` that it by passes the processor cache is friendlier to the real NVM[13]. Further, after the logs are ensured to be written to the NVM in line 4, the use of `nt-store` eliminates the need to perform `clflush/clwb`, which saves instruction overhead. In addition to `nt-store`, DBXN performs a padding operation on the logs to select the most suitable access granularity for writing to NVM. For transaction logs smaller than 256 B, DBXN writes them to NVM with a granularity of 64 B[12]. For logs larger than 256 B, DBXN writes them with a granularity of 256 B[13]. We will analyze the performance impact of these three optimizations on DBXN through experiments in Section 4.4.

4 System Evaluation

To test the performance of DBXN using HTM and NVM to accelerate the transaction processing of multi-core in-memory databases, we conducted tests on a server equipped with real NVM and HTM (Section 4.1). Our tests are expected to answer the following questions.

- Can the combination of HTM and NVM help DBXN significantly reduce the latency of DBX durable transaction processing (Section 4.2)?
- How is the transaction processing performance of DBXN compared with the existing NVM-based transaction systems (Section 4.3)?
- How does each design of DBXN affect its performance (Section 4.4)?

4.1 Test platform configuration

We conduct all tests on a server equipped with a 10-core Intel Xeon Gold 5215 M processor that supports the HTM feature. The server has 192 GB of RAM and 750 GB of Intel Optane PM NVM. The NVM consists of 6 NVM DIMMs. This configuration is the one that can realize the highest NVM performance: 320 Gb/s read bandwidth and 100 Gb/s write bandwidth.

4.1.1 *Comparison objects and configuration of each system*

We compare DBXN with the following systems. DBX-DRAM[1] is a typical current HTM-accelerated multi-core in-memory database that uses an OCC protocol similar to that of DBXN to reduce the impact of HTM hardware limitations on transactions. By default, DBX-DRAM does not enable durable transactions (so we mark it as DRAM). Therefore, it represents the upper limit of system performance that can be achieved in the experiments. DBX-NVM is the version of DBX with durability that supports durable transactions on NVM-based file systems using the group commit mechanism (Section 2.2). DBX-naïve is the simplest way to accelerate DBX durable transactions using NVM (Figure 5(a)), which cannot combine NVM and HTM effectively (Section 2.3).

In addition to comparing with the multi-core in-memory databases represented by DBX, we also compare DBXN with Pisces[15], a programming framework of NVM-based persistent transactional memory. Since persistent transactional memory supports the ACID properties, it can also be used to perform database transactions[15]. By default, Pisces puts all data (including indexes) into NVM. To ensure a fair comparison, we also compare Pisces with DBXN-NVM: DBXN-NVM puts all data (including indexes) in NVM on the basis of DBXN.

We use the configurations to realize the highest performance that each system can achieve unless otherwise stated. For example, in DBXN and DBX-DRAM, the maximum performance of the system is configured as using 10 threads (the maximum number of processor cores on the server) for transaction processing.

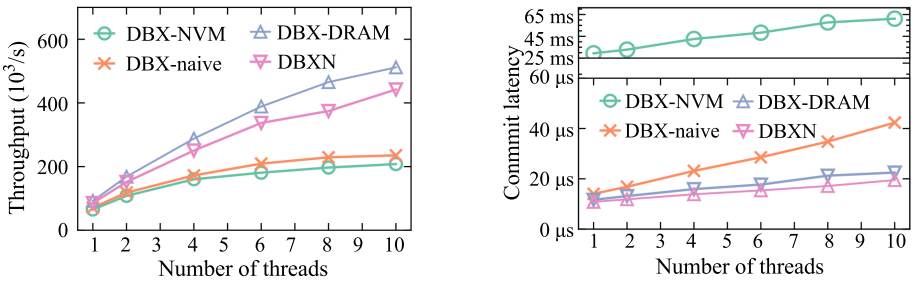
4.1.2 *Test benchmarks*

We analyze DBXN and its counterparts according to two typical transaction processing test benchmarks: TPC-C[18] and Smallbank[25]. TPC-C simulates a stock trading scenario, and its transactions include relatively complex processor operations. For example, the logic of the transaction with the highest percentage in TPC-C, the new-order transaction, is to purchase a dozen stocks. This transaction needs to read and write stocks to the database and insert the corresponding orders. In TPC-C, we deploy a database size of 10 warehouses. Smallbank simulates a simple bank transaction system with a transaction logic of read/write operations of one or two databases. For example, the logic of the deposit checking transaction in Smallbank is to transfer the amount from one bank account to another bank account. In Smallbank, we have deployed a database with 250,000 accounts.

4.2 End-to-end performance comparison

4.2.1 *Throughput analysis on TPC-C*

Figure 9(a) shows an end-to-end comparison of DBXN's throughput with that of the comparison objects on the TPC-C test benchmark. DBXN achieves the highest throughput in each configuration ensuring durable transactions (DBX-NVM, DBX-naïve, and DBXN): with 10 threads, it processes 432,000 TPC-C transactions per second, achieving 86% of the DBX-DRAM throughput without durable transactions (432,000 vs.505,000). This experimental result shows that DBXN's use of the durability mechanism provided by NVM has a little overhead compared with DBX's original HTM-based transaction concurrency control method. Meanwhile, DBXN has 2.1 and 1.9 times higher throughput than DBX-NVM and DBX-naïve, respectively. DBX-NVM still uses DBX's default group commit mechanism to commit transactions, and the system performance is affected by the transaction and log thread interactions underlying this design, which cannot achieve the full high performance of NVM. Since DBX-naïve does not consider the problem that NVM writes would abort the execution of HTM, it cannot use HTM for transaction commit. Therefore, it cannot fully utilize HTM. DBXN effectively circumvents both of these design problems and can utilize both HTM and NVM to achieve high throughput.



(a) Throughput (b) Latency

Figure 9 Comparisons of DBXN's throughput and latency with other systems on TPC-C workloads

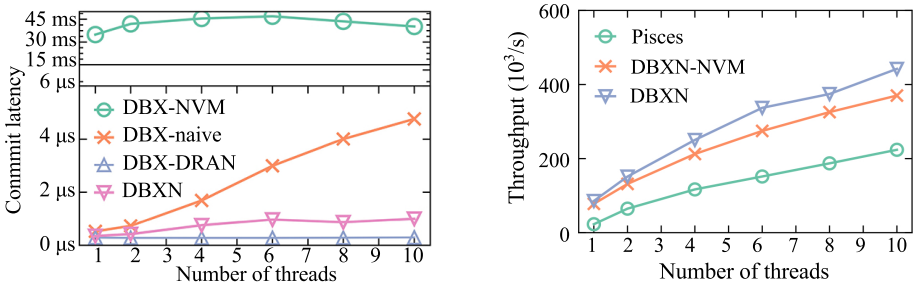
4.2.2 *Latency analysis on TPC-C*

Figure 9(b) further illustrates the latency comparison between DBXN and its counterparts on the TPC-C test benchmark. DBXN achieves the lowest latency under the configurations ensuring durable transactions: with ten threads, DBXN has a latency of 22 µs for durable transactions, while with one thread, it has a latency of 13 µs. These two latencies are only

1.15 times and 1.1 times higher than the non-durable transaction latency, respectively. With ten threads, the transaction will have a higher latency due to the synchronous operation of concurrency control. The latency of DBX-NVM is 2,720 times that of DBXN. In DBX-NVM, the asynchronous transaction commit mechanism is the main reason for the latency increase (Section 2.2). Thus, even a low-latency NVM cannot reduce its latency. Finally, the latency of DBXN is reduced by 53%–83% compared with that of DBX-naïve. Since DBX-naïve cannot utilize HTM for concurrency control, it can only degrade to a software concurrency control mechanism with higher overhead.

4.2.3 *Latency analysis on Smallbank*

Figure 10(a) analyzes the latency comparison between DBXN and its counterparts on the Smallbank test benchmark. Since the comparison of the throughput of systems in Smallbank is similar to that of TPC-C, we omit the analysis for the sake of brevity. From the test charts, we can see that the trend of latency comparison among systems is similar to that in TPC-C (Figure 9(b)): DBX-DRAM has the lowest latency because it does not support durable transactions. DBXN uses NVM on the basis of DBX-DRAM to provide low-latency transactions, whose latency is 1.7–2.4 times higher than that of DBX-DRAM. DBX-naïve cannot take advantage of the hardware features of HTM and NVM together. Thus its latency is 1.2–3 times higher than that of DBXN. Finally, the asynchronous commit mechanism of DBX-NVM introduces a four-order-of-magnitude difference in latency (40,000–50,000 times that of DBXN).



(a) Comparisons of DBXN with other systems in terms of latency on Smallbank workloads

(b) Comparison of DBXN-NVM with Pisces on TPC-C workloads

Figure 10 Comparisons of DBXN with other systems in terms of latency on Smallbank workloads, and comparison of DBXN-NVM with Pisces on TPC-C workloads

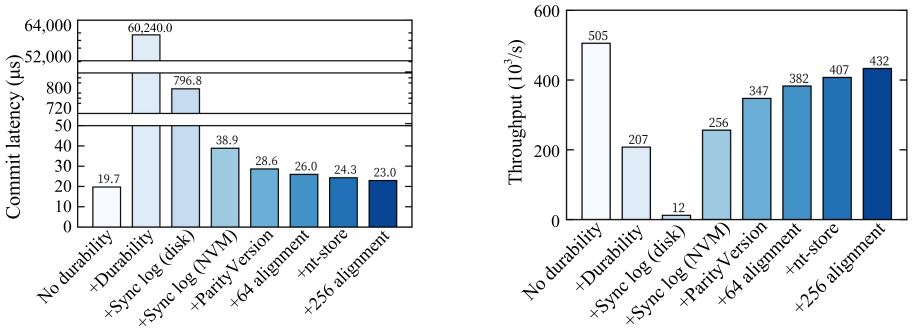
4.3 Performance comparison with existing real NVM-based persistent-transactional memory systems

We compare the performance of DBXN, DBXN-NVM, and Pisces on TPC-C (Figure 10(b)). DBXN-NVM and Pisces can be compared directly because they both put all database data (including indexes) into NVM. DBXN-NVM is 1.65–3.41 times faster than Pisces. This performance improvement comes from three main sources. First, Pisces is a general-purpose persistent transactional memory mechanism. Thus, it is not optimized for the specific scenario of database transactions. Second, although Pisces uses real NVM to accelerate the persistent operation, it is not optimized depending on the hardware characteristics of real NVM. In contrast, DBXN is designed and implemented according to the characteristics of the real NVM (Section 3.4). Pisces' protocol is not accelerated according to HTM.

Compared with DBXN, DBXN-NVM has an additional performance loss of 9%–16%. This is due to the additional data structures (e.g., indexes) it puts in the NVM. The access operations to the NVM are slower than that to DRAM (Table 2).

4.4 Impact of design factors on performance

We analyze the impact of design factors on DBXN performance (Figure 11). First, we can see from +durability that DBX-NVM improves the latency of DBX-DRAM by an order of magnitude from 19.7 μs to more than 60 ms. This latency improvement is mainly due to the software mechanism of asynchronous transaction commit. The synchronous write to disk, i.e., +synchronous log (Disk), reduces latency to 796.8 ms, but brings a 94% drop in transaction throughput. This is because the slow disk becomes a performance bottleneck for high-throughput in-memory databases. NVM [+synchronous log (NVM)] effectively reduces the durability overhead of synchronous log: it further reduces latency to 38.9 μs and increases throughput to 256,000 transactions per second. However, this configuration is still far from optimal performance (432,000 per second). This is because (1) it does not take advantage of HTM and (2) it does not perform optimization for real NVM. When the parity version (+Parity Version) mechanism is adopted, DBXN is able to effectively utilize HTM for transaction commit, which further improves the throughput by 1.35 times and reduces the latency by 26%. Finally, +64 alignment, +nt-store, and +256 alignment (optimizations according to real NVM characteristics) improve DBXN throughput (on the basis of +Parity Version) by 1.1, 1.18, and 1.24 times and reduce latency by 9%, 15%, and 20%, respectively.



(a) TPC-C latency (b) TPC-C throughput

Figure 11 Effect of design factors on DBXN in terms of TPC-C latency and throughput

4.5 Scalability test

Finally, we analyze the scalability of DBXN in the case of larger databases. As the data size increases, DBXN and its counterparts suffer from a decrease in the cache hit ratio, which results in a decrease in performance accordingly. The number of warehouses deployed in TPC-C is increased for testing (Figure 12). The DBXN performance is affected by the increase in data size in this scenario. When 200 warehouses are adopted, the performance of DBXN drops by 57% compared with that in the case of using 10 warehouses. This drop is mainly due to the drop in the system cache locality. For example, DBX-DRAM also has a 48% performance drop on a database of 200 warehouses. Noteworthily, DBXN still outperforms DBX-naïve by a factor of 1.63–2.07 even at larger data sizes.

5 Related Work

5.1 Study of real NVM nature

Yang *et al.*[13] systematize the study of real NVM nature for the first time. They summarize the characteristics of real NVM, including read/write asymmetric performance and access

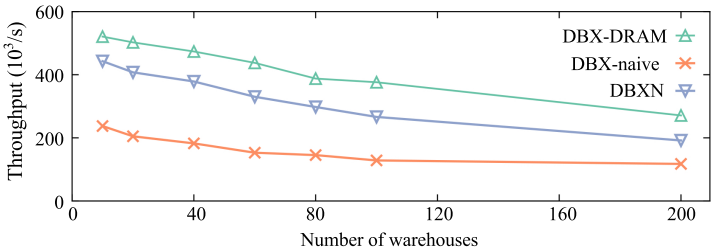


Figure 12 Performance when scaling with more warehouses in TPC-C benchmark

granularity. RDPMA[12] systematically summarizes how to efficiently co-design NVM together with new network hardware. Kalia *et al*.[23] find that processor caching affects the utilization of NVM bandwidth by upper layer systems. In implementing DBXN for real NVM, we draw on the optimizations adopted in these studies for NVM.

5.2 Systems utilizing NVM or HTM

Database and system researchers have done long research on how to accelerate systems with NVM and HTM. NVM has long been used to accelerate the performance of file systems[30]. In addition, NVRC[31] proposes a log update strategy to reduce the write operations in NVM. Xu *et al*.[32] studied the possible security issues of using NVM systems. Luo *et al*.[33] investigated how to optimize the disk connection operations of traditional databases with the architecture of DRAM and NVM. Meanwhile, some studies explore how to design NVM-friendly data structures[34–36] and NVM-friendly key-value store[37]. Finally, Shu *et al*.[38] summarize the research and opportunities related to NVM.

In terms of hardware transactional memory, Wu *et al*.[39] compare the performance between the design of concurrent chain tables accelerated by HTM and traditional lock-based concurrent chain tables. HybridTCache[40] is a hardware-software cooperative transactional memory system based on a dedicated transactional cache. Similarly, DBXN uses software serving DBX to break the limitation of hardware transactional memory. Zeng *et al*.[41] find that the dependency graph-based HTM has better performance than the current conflict detection-based HTM. SPINRTM[42] combines with HTM to design a new synchronization mechanism for virtual machines.

In addition to those using NVM or HTM exclusively, many studies explore how HTM and NVM could work in tandem[14, 43, 44]. Due to the lack of real NVM hardware at the time, these studies only consider designs based on simulated NVM. Similar to these studies, DBXN also focuses on how to better utilize HTM and NVM to accelerate database transaction processing. Meanwhile, DBXN further considers the impact of real NVM hardware features. Crafty[43] uses an undolog mechanism to synchronize transactions executed in HTM, while DBXN uses a redo log mechanism. NV-HTM[44] considers the problem that HTM cannot collaborate with NVM, and it proposes a hysteresis mechanism to properly commit transactions. DBXN uses the parity version mechanism to make HTM and NVM collaborate. PHyTM[14] is a hybrid transactional memory and HTM hybrid in-memory transaction system that focuses on how to solve the problem of limited progress and working sets in HTM. PHyTM does not consider the problem that NVM and HTM cannot be combined. Unlike it, DBXN's parity version mechanism effectively combines HTM and NVM.

6 Conclusion

The emerging of new low-latency NVM provides an opportunity to reduce the transaction commit latency of HTM-based multi-core in-memory databases. In this paper, we propose

DBXN, a multi-core in-memory database, to accelerate existing HTM-based in-memory database's commit latency with NVM. DBXN efficiently combines HTM and NVM to reduce transaction latency through a series of designs including the parity version. Implementations on existing HTM-based multi-core in-memory databases show that DBXN can reduce transaction commit latency by an order of magnitude while improving transaction processing throughput.

References

[1] Wang ZG, Qian H, Li JY, *et al*. Using restricted transactional memory to build a scalable in-memory database. 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. 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, *et al*. Fast databases with fast durability and recovery through multicore parallelism. 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, *et al*. Speedy transactions in multicore in-memory databases. 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 SC, *et al*. Calvin: Fast distributed transactions for partitioned database systems. Proc. of the 2012 ACM SIGMOD Int'l Conf. on Management of Data. 2012. 1–12.

[6] Yu XY, Bezerra G, Pavlo A, *et al*. 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. 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, *et al*. Fast in-memory transaction processing using RDMA and HTM. 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] https://www.anandtech.com/show/9541/intel-announces-optane-storage-brand-for-3d-xpoint-products，2015.

[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, *et al*. Characterizing and optimizing remote persistent memory with RDMA and NVM. Proc. of the 2021 USENIX Annual Technical Conf. (USENIX ATC 2021). USENIX Association, 2021.

[13] Yang J, Kim J, Hoseinzadeh M, *et al*. An empirical guide to the behavior and use of scalable persistent memory. 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, *et al*. Pisces: A scalable and efficient persistent transactional memory. Proc. of the 2019 USENIX Annual Technical Conf. (USENIX ATC 2019). 2019. 913–928.

[16] Liu MX, Zhang MX, Chen K, *et al*. DudeTM: Building durable transactions with decoupling for persistent memory. 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. Proc. of the 31st Symp. on Mass Storage Systems and Technologies (MSST). IEEE, 2015. 1–4.

[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. 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, *et al*. 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, *et al*. 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. 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, *et al*. The end of an architectural era: It's time for a complete rewrite. 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, 2019.

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

[29] Eswaran KP, Gray JN, Lorie RA, *et al*. 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. 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, *et al*. 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, *et al*. Consistent and durable data structures for non-volatile byte-addressable memory. 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. Proc. of the 13th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2018). 2018. 461–476.

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

[37] Kannan S, Bhat N, Gavrilovska A, *et al*. Redesigning LSMs for nonvolatile memory with NoveLSM. Proc. of the 2018 USENIX Annual Technical Conf. (USENIX ATC 2018). 2018. 993–1005.

[38] Shu JW, Lu YY, Zhang JC, *et al*. 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, *et al*. 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).

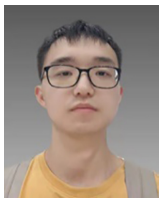
[42] Yu QQ, Dong MK, Chen HB. Hardware transactional memory assisted synchronization mechanism in virtualized environment. Journal of Frontiers of Computer Science and Technology, 2017, 11(9): 1429–1438 (in Chinese with English abstract).

[43] Genç K, Bond MD, Xu GQH. Crafty: Efficient, HTM-compatible persistent transactions. 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. Proc. of the 2018 IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS). 2018. 368–377. [doi: 10.1109/IPDPS.2018.00046]

Xingda Wei, Ph.D., assistant professor. His research interests include operating systems and distributed systems.

Haibo Chen, Ph.D., professor, doctoral supervisor, distinguished member of CCF. His research interests include operating systems and parallel and distributed systems.



Fangming Lu, bachelor. His research interests include operating systems and distributed systems.

Binyu Zang, Ph.D., professor, distinguished member of CCF. His research interest is operating systems.

Rong Chen, Ph.D., professor, senior member of CCF. His research interests include operating systems and distributed systems.