Pisces: A Scalable and Efficient Persistent Transactional Memory

Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang,
Binyu Zang, Haibing Guan, Haibo Chen
Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University
Institute of Parallel and Distributed Systems (IPADS), Shanghai Jiao Tong University

Abstract

Persistent transactional memory (PTM) programming model has recently been exploited to provide crash-consistent transactional interfaces to ease programming atop NVM. However, existing PTM designs either incur high reader-side overhead due to blocking or long delay in the writer side (efficiency), or place excessive constraints on persistent ordering (scalability).

This paper presents Pisces, a read-friendly PTM that exploits snapshot isolation (SI) on NVM. The key design of Pisces is based on two observations: the redo logs of transactions can be reused as newer versions for the data, and an intuitive MVCC-based design has read deficiency. Based on the observations, we propose a dual-version concurrency control (DVCC) protocol that maintains up to two versions in NVM-backed storage hierarchy. Together with a three-stage commit protocol, Pisces ensures SI and allows more transactions to commit and persist simultaneously. Most importantly, it promises a desired feature: hiding NVM persistence overhead from reads and allowing nearly non-blocking reads.

Experimental evaluation on an Intel 40-thread (20-core) machine with real NVM equipped shows that Pisces outperforms the state-of-the-art design (i.e., DUDETM) by up to 6.3× for micro-benchmarks and 4.6× for TPC-C new order transaction, and also scales much better. The persistency cost is from 19% to 50% for 40 threads.

1 Introduction

Non-volatile memory (NVM) such as phase-change memory (PCM) [46, 67, 78], resistive random-access memory (ReRAM) [9, 45], and Intel/Micron's 3D-XPoint [2, 4], is revolutionizing the storage hierarchy thanks to the promising features like byte-addressability and non-volatility with a close-to-DRAM speed. By supporting persistent data access via CPU load/store instructions, these technologies bring ample opportunities for applications to achieve optimal performance as well as efficient crash consistency [20, 62].

To efficiently program on NVM with a balance among good programmability, high performance, and low software overhead, persistent transactional memory (PTM), also known as durable (memory) transactions, has been exploited by prior work [21, 37, 44, 49, 56, 74, 76]. Through combining transactional memory [29, 34, 66, 68, 70] and NVM, PTM offers the properties of atomicity, consistency, isolation, durability (ACID) to applications on NVM.

To ensure the durability for transactions, some prior designs [21, 44, 56, 74] need to persist a transaction's log while holding the locks of the data being modified or explicitly track the dependencies among transactions through locks. This, however, may block concurrent read operations on the same data. The long blocking duration may become a severe performance bottleneck due to the amplified persistence overhead incurred by high write latency of NVM (usually 10× compared to DRAM) (*low read efficiency*). This is especially true when read operations dominate in many workloads [15, 19, 52, 63]. In contrast, another design [49] eliminates the persistence latency from a transaction's critical path through relaxing the durability semantics, i.e., making a transaction's modifications visible before its log reaches NVM. However, such a design sacrifices the durability guarantee and requires to apply logs back to the durable data according to a total order. Unfortunately, such a strict persistence ordering may be the bottleneck of scalability since it is hard to parallelize the persistence operations. Overall, it is challenging to design a PTM system that insulates readers from being affected by high NVM persistence overhead while enforcing strong durability as well as avoids overly-constrained persistence ordering simultaneously.

We notice that *snapshot isolation* (SI) [8, 12] can avoid read-write conflicts and suffices for many real-world applications [6, 11, 26, 28, 33, 47, 48, 61, 68, 77], which makes it possible to design a PTM that allows a transaction to persist its log in its critical path (no sacrifice the durability), while hiding the high persistence overhead from concurrent read operations. Multi-version concurrency control (MVCC) [13, 28] is a common choice to achieve SI. We *observe* that the (redo) logs which will be finally applied to the durable data (old) can be regarded as a new data version, which enables us to efficiently introduce MVCC to PTM.

However, after a deep analysis, we *find* that a straightforward MVCC-based PTM design not only brings high reader-side overhead due to read-indirection problems (challenge-1), i.e., locating the consistent objects in the version lists, but also still leaves the readers affected by the NVM persistence overhead (challenge-2). So, we further present *Pisces*, a read-friendly PTM design that also embraces SI while solving the above two problems and achieves both high read efficiency and good scalability.

Specifically, Pisces proposes *dual-version concurrency control* (*DVCC*) inspired by MVCC and scalable synchronization primitives [18, 51, 54, 55], to solve challenge-1. DVCC still avoids read-write conflicts and thus allows high parallelism for transaction execution, but only keeps one or two versions (using the log as the newer version) for each data object, which minimizes the high cost for maintaining multiple versions in NVM as well as searching in the version lists. To solve challenge-2, Pisces hides the NVM persistence overhead from readers through *three-stage commit* that separates the durable point and the visible point of a (read-write) transaction and minimizes the possible read-blocking time to the duration of two DRAM stores. This blocking *rarely* happens since the possible blocking time is extremely short. A transaction persists its logs (new versions for objects) into NVM in a persist stage (durable) and makes its logs readable to other transactions in a following concurrency commit stage (visible). Note that the potential blocking period resides in the latter stage. Hence, the NVM persistence overhead can be hidden from readers. Besides, a transaction eagerly reclaims the old versions of objects and overrides them with new versions in the last write-back stage, which is for avoiding indirect reads. Furthermore, Pisces also enables *flush-diff* (persist modifications only) to prevent excessive NVM persistence operations and leverages *group-commit* to reduce the overhead for write transactions.

In all, Pisces hides the NVM persistence overhead from readers and promises *almost non-blocking* reads. Pisces guarantees *snapshot isolation* (a formal proof is also provided), and promises *crash consistency* that can restore the system to a consistent snapshot after crashes. In essence, Pisces explores a trade-off between isolation and performance by loosening the isolation level for better performance.

We have implemented and evaluated Pisces on a 40-thread machine. Evaluation results show that Pisces has a notably higher throughput and better scalability compared with the state-of-the-art design (i.e., DUDETM [49]). Specifically, Pisces achieves up to 6.3× throughput improvement in micro-benchmarks and can improve the throughput of TPC-C new order transaction [24] and TATP benchmark [72] by 460% and 64%, respectively.

In summary, this paper makes the following contributions:

- An observation that redo logs can be used as newer data versions and an intuitive MVCC-based PTM design with the observation. A careful analysis of the read-inefficiency of the MVCC-based design.
- A first PTM with snapshot isolation (Pisces), which leverages DVCC and three-stage commit to benefit readers most.
- An implementation and evaluation on a real machine with NVM that demonstrate Pisces's efficiency and scalability.

2 Background & Overview

Comparing with database transaction, Transactional memory (TM) [38] ensures atomicity, isolation and consistency (ACI), but lacks the important property of durability. However, the emergence of non-volatile memory provides an opportunity to equip TM with durability [44, 49, 56, 74, 76]. This section first introduces the backgrounds of NVM and PTM, then provides an overview of our system.

2.1 Background

NVM. The recent release of Intel Optane DC Persistent Memory [2] marks the transition of non-volatile memory (NVM) technology from research prototypes to mainstream products. NVM promises to provide fast data persistency. According to current studies [47, 79, 81], NVM has the following three features. First, most NVM designs are byte-addressable. This is one major reason why we can directly replace DRAM with NVM. Second, NVM has close-to-DRAM read latency, but about 10× write latency comparing with DRAM. For example, PCM's write latency is 150∼1000ns and ReRAM's is 500ns, while DRAM has only 60ns write latency [47, 79]. Third, special instructions [3] are provided to help persist the data: 1). *pflush* (e.g., *clwb*) will flush a cache line from CPU cache to NVM. 2). *pfence* (e.g., *mfence*) ensures all previous *pflush* instructions finish.

PTM. There are already various researches which build Persistent Transactional Memory (PTM) systems by leveraging NVM [21, 23, 44, 49, 56, 74]. However, most of them focus on optimizing the persistence overhead [21, 44, 56, 74]. For example, Kamino-tx [56] removes the overhead of data copy in a transaction's critical path by maintaining a backup of all the data. However, in these systems, an on-going write operations usually block conflicting read operations to ensure the consistency. As a result, these read operations will also suffer from NVM's high write latency.

A state-of-the-art design, called DUDETM [49], tries to address this issue with a decoupled PTM design: it temporarily buffers the running transaction's updates and its redo log in DRAM, then a number of threads will flush the log to NVM asynchronously. Furthermore, a dedicated thread (named reproduce thread) will replay the log to apply the updates to the persistent objects in NVM. However, once the log buffer becomes full because the reproduce thread cannot timely replay and clean the logs, the system needs to stall to wait for the reproduce thread to catch up. To ensure the consistency of the persistent state, the reproduce thread needs to replay the operations in the log sequentially. As a result, the reproduce thread harms the system scalability. Figure 1-(a) shows the scalability issue of DUDETM: its performance can only scale up to 8 cores, after which the performance will be bottlenecked by the reproduce thread.

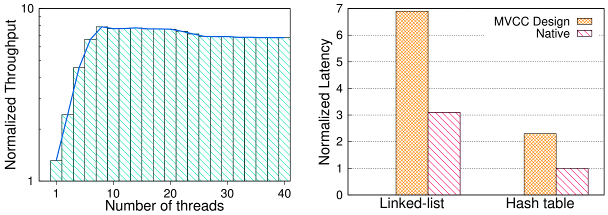

Fig. 1: (a) A hash table benchmark with 40% update rate. (b) A comparison on read-only transaction latency.

2.2 Overview

Goal. Comparing with existing works, Pisces is a PTM system with a read-friendly design, since lots of workloads using transactions are read-dominated [15, 19, 22, 30, 52, 63]: for example, the read-write ratio in the update operation of an 8 layer (8-15 keys per node) B+-tree is about 80:1; In the TPC-E and TATP [25, 72] benchmarks, about 80% of the transactions are read-only.

Strawman. Pisces is based on the intuition that *snapshot isolation* (SI) [8, 12] is able to avoid blocking reads by conflicting writes. At the same time, SI is applicable not only to database workloads [6, 11, 26, 28, 33, 47, 77], but also to TM workloads [48, 61, 68]. For example, Lu et al. [53, 68] prove that SI is enough to support a concurrent skip list. Thus, both database and STM systems have begun to use SI to improve concurrency [48, 61, 68].

However, is an intuitive SI implementation good enough to achieve our goal, a read oriented PTM? To answer this question, we implement a prototype system to provide SI based on multi-version concurrency control. Each object maintains a list of multiple versions and is identified by an ID. Each version has a timestamp to indicate its committed point-in-time. When a transaction starts, it sets a start timestamp based on the global timestamp kept by the system. During execution, to read an object, a transaction finds the most recent version which has a smaller timestamp than the transaction's start timestamp by traversing the object's list. For write, a transaction buffers updates in the write set and records the operation in a redo log. To commit its updates, a transaction first acquires the locks of all objects it tries to update. Then it detects write-write conflicts by checking the latest timestamp of these objects. If any object's latest timestamp is larger than the transaction's start timestamp, then the transaction is aborted. After passing the validation, the transaction retrieves its commit timestamp and updates the global timestamp. Then, the transaction flushes its log and the commit timestamp from CPU cache into the NVM, then flushes the updates in the write set to the persistent objects in NVM. At last, it releases all locks.

Issues. To analyze the efficiency of this design, we use it to implement concurrent data structures and compare with their native (single-threaded) implementations. Figure 1-(b) shows the evaluation results: the intuitive design has considerable overhead on read requests' latency because of the following problems:

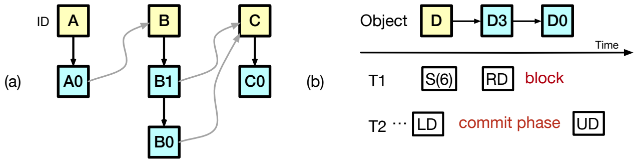

Fig. 2: (a) An ordered linked list structure in our MVCC-based PTM. Black arrows represent pointers in different version lists, while the gray arrows indicate the pointers in the linked list. (b) Read operations get blocked in MVCC.

First, traversing the multi-version list of each object increases the latency. This is not only because a transaction with small start timestamp may need to perform multiple indirect memory accesses, but also because the random accesses may harm the cache locality and get blocked due to the cache line being evicted to NVM. Figure 2 (b) gives a simple example: if a transaction traverses this 3-object list, it actually needs to traverse a much longer list which at least contains three objects and three object IDs.

Second, read operations may still be blocked by the NVM persist operations. Specifically, when a reader accesses an object which is locked by a writer, the reader may be blocked until the writer commits. The reason is the reader is not sure if the writer will have a smaller commit timestamp than its start timestamp or not. Unfortunately, the writer cannot commit until it flushes all its logs into NVM and applies the updates in its write set. Figure 2 (b) gives a simple example: T_1 starts with timestamp 6. When it reads object D, it finds D is locked by T_2. Then, it has to be blocked, as T_2 may update D with a timestamp smaller than 6.

Basic idea. We develop *Pisces* to solve above issues based on the following basic designs:

Dual-version concurrency control (DVCC). To reduce the cost of traversing an object's list, Pisces keeps up to two versions for each object: original object and object copy. When a transaction tries to write an object, it creates and links a new copy to the original object. When the transaction commits, it writes the object copy back to the original object. Concurrent transactions update the same object exclusively by acquiring a lock. Read transactions are able to directly access either version based on their timestamps. Furthermore, to reduce unnecessary NVM writes, we reuse the updates in the redo log as object copy. However, the challenge to implement DVCC is how to ensure an original object won't be overwritten when it may still be needed by some outstanding transactions with smaller start timestamps.

Three-stage commit protocol. To reduce the blocking overhead in the MVCC design, Pisces proposes a three-stage commit protocol: the commit phase is divided into concurrency commit stage, and write-back stage. In the persist stage, a transaction flushes its log into NVM. In the concurrency commit stage, the transaction updates its end timestamp (commit timestamp) and the timestamps of all the object copies in the redo log atomically. In the write-back stage, the transaction writes all object copies back to their original objects. By decoupling different functionalities of the commit phase, Pisces allows nearly non-blocking reads. But the challenge lies in how to atomically update both the end timestamp and the timestamps of object copies efficiently.

Limitation. The main limitation of Pisces is it only provides SI which does not work for all applications, and SI suffers from the well-known write skew anomaly[1] under certain conditions. However, there is a long line of research [16, 33, 48, 53] on how to detect or eliminate write skew anomalies for SI. Moreover, making SI serializable is also well studied [59, 64, 69, 75]. Leveraging these techniques to provide a stronger isolation level is future work. Currently, careful programming on Pisces is required.

3 Design

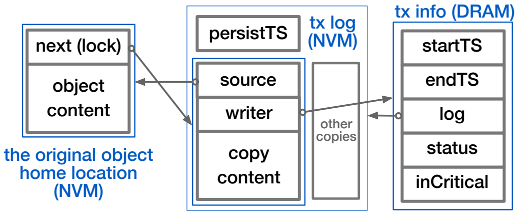

Fig. 3: The memory layout of an object, per-transaction log and per-thread metadata. Arrows represent pointers.

Layout. Figure 3 shows the memory layout of three critical components (data object, per-transaction log and per-thread metadata) in Pisces. Pisces attaches each object with a pointer (named *next* and initialized as 0) which may point to a next version of this object and is also used as a write lock that needs to be exclusively acquired by a writer. Pisces pre-allocates log area for each thread and each transaction gets its log from the log area of the execution thread when it begins. Pisces also keeps per-thread metadata to record the metadata of the running transaction in each thread (a thread executes at most one running transaction at a time). Object copies as the next (newer) versions of objects reside in the transactions' log and each object copy contains two pointers. One is named as *source* and points to the original object. The other is named as *writer* and points to the running transaction that owns this log (creates this copy).

For the challenges mentioned in the above section, we provide simple but efficient solutions accordingly: First, to prevent an original version from being falsely overwritten, we leverage an RCU-similar design (grace period detection) to block the writer in the write-back phase until the original version in home location is safe to be overwritten. Note that the blocking time will not be exposed to readers. Second, to atomically update a transaction's end timestamp and each copy's version timestamp in an efficient way, we do not explicitly maintain the version timestamp for each copy. Instead, each copy contains the *writer* pointer and reuses the transaction's end timestamp as its version timestamp. As a result, atomicity is guaranteed by simply updating the end timestamp of the write transaction. Next, we discuss the details about the algorithm whose pseudo code is provided in Algorithm 1 and the correctness argument.

3.1 Algorithm

TM_Start begins a transaction. A transaction marks its status as ACTIVE first, executes a *fence* instruction and reads the global timestamp (globalTS) as its start timestamp (startTS). The *fence* instruction ensures line 2 is executed before line 4.

TM_Read returns a pointer for reading an object. It first reads the value of next pointer in the original object and returns the original object directly if next is zero. This is the *fast path*: accessing the pointer located just before the object introduces nearly-zero overhead because the CPU will prefetch adjacent cache lines. If next is non-zero, which means there exists an object copy, *TM_Read* returns the object copy when it is created by the current transaction (line 12) or its version is no greater than the current transaction's start timestamp (line 15). There is only one rare case in which *TM_Read* needs to wait (line 14). We discuss this later when introducing *TM_Commit*.

TM_Write returns a pointer for writing an object. A transaction can directly write a copy created by itself (line 21 to 23). When writing an object for the first time, a transaction reserves an area for the object copy in its log and tries to acquire the object's write lock (i.e., next pointer) with a compare-and-swap instruction (line 25). If fails to lock, the transaction aborts and restarts after a random delay, which also avoids deadlocks. Otherwise, it copies the original object's content to the object copy and can directly read or write the copy now. Pisces makes copies (redo log) at object granularity, which mitigates the *read-indirection problem* of redo logging at byte granularity. Pisces chooses encounter-time locking to detect conflicts early and thus can avoid unnecessary NVM writes. Also, an object's write lock ensures that it can only be updated sequentially.

TM_Commit always successfully commits a transaction. A transaction marks its status as INACTIVE, indicating it no longer reads any object. It commits directly if it is a read-only transaction. A read-write transaction needs to go through

[1] A typical write skew example is: One transaction reads A and writes B while another concurrent transaction reads B and writes A.

Algorithm 1: Pseudo code of Pisces

```
 1: Function TM_START(tx)
 2:     tx.status = ACTIVE
 3:     fence
 4:     tx.startTS = globalTS
 5:     tx.endTS = INF
 6:
 7: Function TM_READ(tx, p_obj)
 8:     next = p_obj.next
 9:     if next is EMPTY then
10:         return p_obj          // fast path
11:     wtx = next.writer
12:     if wtx is tx then
13:         return next
14:     wait until wtx.inCritical is FALSE
15:     if wtx.endTS $\leq$ tx.startTS then
16:         return next
17:     else
18:         return p_obj
19:
20: Function TM_WRITE(tx, p_obj)
21:     if p_obj.next is NON-EMPTY
22:     and p_obj.next.writer is tx then
23:         return p_obj.next
24:     copy = tx.log.alloc(p_obj)
25:     copy.writer = tx
26:     if CAS(p_obj.next, EMPTY, copy) fails then
27:         abort()
28:     copy.source = p_obj
29:     memcpy_content(copy, p_obj)  // p_obj -> copy
30:     return copy
31:
32: Function TM_COMMIT(tx)
33:     tx.status = INACTIVE
34:     // stage 1: persist stage
35:     if tx.log is EMPTY then
36:         return
37:     pflush(tx.log)
38:     pfence
39:     tx.log.persistTS = globalTS
40:     pflush(tx.log.persistTS)
41:     pfence
42:     // stage 2: concurrency commit stage
43:     tx.inCritical = TRUE
44:     fence
45:     tx.endTS = globalTS + 1
46:     tx.inCritical = FALSE
47:     AtomicInc(globalTS)
48:     // stage 3: write-back stage
49:     WRITEBACK(tx)
50:
51: Function WRITEBACK(tx)
52:     while exists an ACTIVE transaction t do
53:         if t.startTS $<$ tx.endTS then
54:             wait
55:     for each copy in tx.log do
56:         memcpy_content(copy.source, copy)
57:         pflush(copy.source.content)
58:         copy.source.next = EMPTY
59:     pfence          // not necessary for correctness
```

three stages. In the *persist stage*, a transaction persists[2] all the object copies in its log into NVM (line 37-38). After that, it retrieves the value of the global timestamp as its log's persist timestamp (line 39) and makes the persistTS persistent (line 40-41). The *pfence* instruction in line 38 guarantees the log's content reaches NVM before its persistTS, and the *pfence* instruction in line 41 ensures both the log and its persistTS reaches NVM. A checksum can be appended to reduce the two fences to one [65]. A transaction's updates become durable once its persistTS reaches NVM (*durable point*). After a crash, a recovery procedure will replay transactions according to the redo logs and in persistTS order.

In the *concurrency commit stage*, the transaction updates its timestamp atomically by updating the 64 bit endTS with the globalTS (line 45). A boolean flag *inCritical* is used to protect this update to make sure the updated endTS is eventually visible to concurrent reads. For example, T_1 may read the globalTS and update its endTS. However, the updated endTS may be kept in the CPU store buffer and waits to be flushed to CPU cache. As a result, a concurrent transaction T_2 whose startTS is not less then T_1's endTS may fail to observe T_1's update. With the *inCritical* flag, T_1 will be blocked until *inCritical* is disabled before it tries to read T_2's endTS (line 14, 15). This ensures T_2's updates on endTS is eventually visible to T_1. As TSO architecture may reorder read-/write instructions, one fence[3] (line 44) is needed to ensure the execution order of line 43 and 45.

In the *write-back stage*, the transaction first waits for all active transactions whose startTS is less than its endTS to finish (line 52-54). This period actually is the *grace period*. It avoids falsely overwriting an original object which may be needed by transactions with small startTS. Because after this period, all threads are either in an inactive state (not executing transactions) or executing transactions with a startTS larger than the original object's timestamp. Therefore, the original object is *dead* which means it is no longer needed by any transactions. At the same time, this period also helps to detect write-write conflicts. Considering another conflicting transaction with smaller startTS, but access the same object after this transaction. This transaction will be blocked at the write-back stage and cannot release the lock. So, the conflicting transaction will abort since it fails to acquire the lock when accessing the object (line 26). At the end of the write-back stage, it writes each next object to the original object (line 56-57) and releases locks by clearing *next* fields (line 58).

Programming: Each transaction should be surrounded by *TM_Start* and *TM_Commit*. For reading/writing an object, it first uses *TM_Read*/*TM_Write* to achieve the ob-

[2] In the current implementation on Intel CPU, Pisces uses *clwb* to flush cacheline and *MFENCE* to ensure previous flushed cachelines reach NVM.

[3] Currently, Pisces uses *MFENCE* [3] instructions which ensures the CPU store buffer is always drained besides serializing load and store operations.

ject pointer and then directly accesses that object with the pointer. Pisces also offers helper functions (*TM_Read_Field* and *TM_Write_Field*) to ease programming.

3.2 Log Recycle

Pisces stores logs in per-thread ring buffers and lets each thread recycle its own logs. Generally speaking, there are two principles for log recycle in Pisces for snapshot isolation and crash consistency, separately.

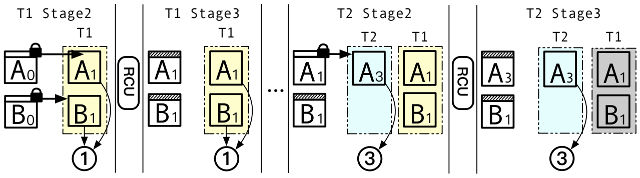

Fig. 4: T1 and T2 are two read-write transactions in one thread. T2 happens after T1. Colored rectangles represent transactions' logs.

P-1: A transaction log can only be recycled after all the copies in it are dead. A transaction creates new versions of objects in its log and exposes them to other transactions. As shown in Figure 4, after a transaction T1 writes the new versions back to the original objects, it is possible that the transaction's log is still required. For example, another transaction can read T1's log if it starts before T1 unlocks the objects and its startTS is no smaller than A's endTS. So, a transaction will not reclaim its own log. Instead, when a following read-write transaction (T2) finishes, the execution thread marks the log of the previous read-write transaction (T1) as reclaimable. Similar to how RCU grace periods can help safely overwriting original objects, the end of the grace period in T2's write-back stage can ensure other transactions no longer access T1's log. Specifically, the end of this grace period ensures (recall line 52-54 in Algorithm 1): previous transactions that may access A1 and B1 in T1's log due to smaller startTS are finished. Therefore, Pisces guarantees all the copies in a log are dead before recycling the log, which achieves *P-1*.

P-2: A transaction log can only be recycled no earlier than all the logs with smaller persist timestamps are recycled. Suppose a transaction A updates an object before another transaction B. If B's log is recycled before A and a crash happens, B's updates will lose after recovery because A will be redone according to its log. To enforce *P-2*, Pisces uses an epoch-based mechanism for recycling logs. It logically distributes logs to epochs according to their persist timestamps. First, an execution thread marks a transaction's log as reclaimable through recording the transaction's persist timestamp as the thread's *reclaim timestamp* (a per-thread variable). Second, an execution thread will atomically advance the global *epoch* when it finds that all the threads' *reclaim timestamp* exceeds the current global *epoch*. Once the global *epoch* increases, the logs belong to the previous epoch are no longer required, and the corresponding log area can be reused.

3.3 Proof Sketch of Snapshot Isolation

A formal proof can be found in [1]. According to the specification of snapshot isolation [8], we prove Pisces is correct by proving the following two theorems are correct.

THEOREM 1 (SNAPSHOT WRITE).

If two transactions update the same object, then one transaction's start TS (short for timestamp) should be greater than another's end TS.

PROOF. Based on the fact that, because of locking (line 26, 58), the conflicting transactions update the same object sequentially, we only need to prove the latter's start TS is always larger than the former's end TS. Pisces ensures this invariant by aborting the latter one when it gets a smaller (illegal) start TS: let's assume both of T_i and T_j access object x and T_i is before T_j. If T_j's start TS is smaller than T_i's end TS, T_i will be blocked by the active T_j at the write back phase (line 52-54). T_j must be active because, by the assumption, it will access x after T_i. Thus, when T_j accesses x, it will find the lock is held by T_i and abort itself. □

Before giving Theorem 2, we first define *the TS of an object* as the end TS of its last writer.

THEOREM 2 (SNAPSHOT READ).

If a transaction T_r reads an object x with timestamp TS_x, then: 1) T_r's start TS is not less than TS_x; and 2) There does not exist a transaction T_w that updates x and its end TS is larger than TS_x, but not greater than T_r's start TS.

PROOF. To prove Pisces holds the first invariant, we show that the x's copy returned by TM_READ must be committed by a transaction whose end TS is not greater than T_r's start TS. First, considering the case TM_READ returns x's original version (line 10, 18). On the one hand, Pisces ensures that, when T_r starts, all objects' original versions have timestamp which is not greater than T_i's start TS. On the other hand, Pisces also forbids any transaction whose end TS is greater than T_i's start TS to overwrite the original version (line 52-54). Next, we consider the case that TM_READ returns x's the next version (line 13, 16). Pisces ensures the invariant by adding an extra constraint that the writer's end TS must be not greater than T_r's start TS (line 15). [4]

Instead of directly showing Pisces holds the second invariant, we prove a variant: if T_w's end TS is not larger than T_r's start TS (assumption), then it is also not larger than TS_x. By the assumption above, we can have T_r starts (line 4) after T_w reads the global TS in commit phase (line 45). Now, let's consider two cases: 1). T_r reads x before T_w unlinks the next version from the object (line 58). Thus T_r is able to get the next version updated by T_w. Because the TSO architecture does not reorder the updates/reads on *endTS* and *inCritical* in TM_COMMIT/TM_READ, thus if T_r finds T_w's *inCritical* is false then it must be able to observe T_w's *endTS*. As a

[4] The detail proof of the consistency between the *next* copy and its *writer* pointer can be found in [1].

result, T_w's end TS should be equal to TS_x 2). T_r reads x after T_w writes back and unlinks the next copy. For this case, T_r will read the version committed by T_w or the transaction accessing x after TS_w. Then, we can have TS_x that must be less than T_w's end TS by simply deriving from Theorem 1. □

3.4 Crash Consistency

A recovery procedure will start after a crash and redo the durable transactions in a non-decreasing order of their logs' persistTS. The recovery time is affected by the length of the log which is decided by the log recycling frequency. By default, each execution thread in Pisces tries to recycle the logs after executing 3 (the default threshold) read-write transactions, so Pisces does not suffer from a high recovery cost. Also, Pisces provides a persistent allocator based on SSMalloc [50], which can recover the allocation information.

The key to ensuring crash consistency is that *dependent* transactions are (i) persisted and (ii) redone (after a crash) in the correct order that corresponds to their commit order. Two transactions are *dependent* if the read set or write set of the subsequent one overlaps with the write set of the previous one. In this subsection, we explain how Pisces achieves both (i) and (ii).

Achieving (i): Pisces guarantees the persistence ordering of dependent transactions by deferring the visibility of the updates of a transaction. Specifically, a transaction reaches its durable point (the end of *persist stage*) before it is visible (the end of *concurrency commit stage*). If transaction B observes (depends on) transaction A' updates (visible), A must already be persisted (durable). So, the persistence ordering of two dependent transactions must correspond to their commit order, as shown in Figure 5.

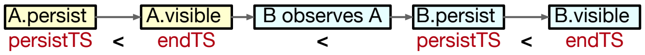

Fig. 5: Transaction B depends on A. Arrows mean happen-before.

Achieving (ii): On one hand, Pisces guarantees that if a transaction B depends on another transaction A, B's persist timestamp must be greater than A's (see Figure 5). On the other hand, Pisces's recovery procedure redoes the transactions' logs in a non-decreasing order of their persist timestamps after a crash. Therefore, transaction B that depends on A will be redone after A, which also corresponds to their commit order (B's endTS is greater than A's.) Besides, independent transactions that have the same persist timestamp can be redone in any order.

4 Optimizations

Flush-diff. Creating redo logs in the granularity of whole objects can avoid searching for new values in the address-value pairs. But it is quite expensive when the modification to an object is much smaller than the object size because the whole log (entire copy) has to be persisted into NVM. Therefore, we give an optimization named *flush-diff* that only persists the modifications. A transaction creates the object copies in DRAM instead of NVM and only records the updates (address-value pairs) in NVM. In the first commit stage, a transaction only needs to flush those updates into NVM; In the second commit stage, it lets the in-DRAM object copies (new versions) become readable; In the third commit stage, it only needs to apply the logged updates to the original objects in the NVM. Therefore, *flush-diff* can still embrace the advantage of logging a whole object (i.e., directly read/write the copy without the overhead of indirection) by keeping the volatile object copy in DRAM, and significantly reduce the amount of NVM persistence operations.

It is worth mentioning that DRAM footprint in Pisces is limited and not related to the amount of NVM in use. This is because Pisces only temporarily buffers the new object versions in DRAM and timely recycles the transactions' logs. Therefore, the DRAM needed for *flush-diff* is only related to the working set size of the transactions whose newly created copies may still be referenced.

Group Commit. Pisces chooses to eagerly write a new version back to an object's home location, which is for benefiting readers but makes the last commit stage for write transactions heavier. As mentioned in Section 3.1, a read-write transaction uses the RCU-like waiting mechanism to overwrite the objects in the home location. Besides, it is also costly to update *globalTS* with atomic instruction like *fetch_and_add* in a high-contention workload.

To amortize the overhead of both RCU reclamation and updating *globalTS*, Pisces batches and commits several write transactions together. An execution thread delays the commitment of a write transaction till the number of pending transactions (wait for commit) reaches a threshold or another transaction needs to update the same object with T. Then, the thread commits the pending transactions together, i.e., execute RCU mechanism and update the globalTS for one time. Thus, the overhead is amortized by these transactions.

5 Evaluation

5.1 Experimental Setup

Basic Setup. We conduct the experiments on a server provided by Intel. The server has two sockets, each containing a 10-core Intel Xeon Gold 5215M CPU, 128GB DRAM and 128GB Intel Optane DC Persistent Memory (NVM). We enable hyper-threading and bind each software thread to each hyper-thread (40 hyper-threads in all) that runs at 2.5GHz. The Linux kernel version is 4.19.32, and the GCC version is 8.3.1. Without an explicit statement, we use *clwb* instructions to persist data into NVM.

System for Comparison. *DUDETM* [49] maps persistent data in NVM to DRAM and uses TinySTM [35] which is a word-based STM to execute transactions in DRAM. A transaction only needs to write logs in a per-thread volatile log buffer, and the background persist threads flush these volatile logs into persistent log buffer. DUDETM also requires another reproduce thread to write logs back to the persistent data. When evaluating DUDETM, we set the size of pre-allocated DRAM area the same as pre-allocated NVM area to make it able to cache all the NVM data in DRAM, which avoids the potential high overhead of page swapping. The volatile log buffer for each thread can hold 1 million log entries (default configuration), and we double the default size of persistent log buffer to make it able to hold 32 million log entries. In all the experiments for DUDETM, we wait for only foreground threads and not for background threads to finish. Thus, the committed transaction may be not durable yet. Besides, the background threads are not counted into the thread number. Currently, DUDETM does not implement multiple background persistent threads, but the support can be added without changing its design. However, the background reproduce thread needs to replay transactions' logs according to the unique transaction ID, i.e., a total order (hard to utilize multi-threading), which will become the bottleneck in some scenarios. To avoid the single persist thread becoming the bottleneck in the experiments, we allocate the persistent log buffer (should be in NVM actually) in DRAM.

Benchmarks. Generally speaking, to develop SI-safe applications, programmers need to analyze whether write skew anomalies may happen in some cases and then avoid the potential anomalies through making write-write conflicts in such cases. We manually ensure the presented benchmarks are SI-safe. referring to [48, 53, 54]. For examples, a transaction on a linked list (used in the following hash tables) adds all the modified nodes *including the nodes to be removed* into its write set in which each node object will be locked before updated; a transaction on the following tree structures traverses from the root node to some node in one way and also adds all the to-be-modified nodes into its write set.

5.2 Micro-benchmarks

Hash Table. Each hash table contains 10K buckets (implemented as linked lists) and initially contains 100K key-value pairs. We create different numbers of threads to execute search and insert/remove transactions. Figure 6 presents the evaluation results with various update rates. Note that the y-axis uses *log-scale*. Every test runs 30 seconds.

When the update rate is low, such as 0% and 2%, both Pisces and DUDETM scale well. Although Pisces's read-only transactions may load data from NVM when the data does not reside in CPU cache, they are still faster than that in DUDETM. It is because DUDETM requires a software page table mechanism for translating NVM addresses to DRAM addresses and validation for read operations. Different from DUDETM which incurs software overhead for read operations, Pisces embraces a read-friendly design, and thus its throughput is much higher and grows faster.

When the update rate is 20%, Pisces still scales well within 20 threads, i.e., a single NUMA (non-uniform memory access) node. The throughput of Pisces increases from ∼1.9 Mops/s to ∼26 Mops/s when the thread number increases from 1 to 20. However, there is an obvious performance drop when the thread number changes from 20 to 22. The main cause is cross-NUMA memory accesses. First, a global timestamp is updated in read-write transactions in Pisces, and a large number of read-write transactions incur high contention on that timestamp. Second, the grace period detection is another overhead for the read-write transactions, and its cost can be enlarged by frequent remote memory accesses. Recall that Pisces uses the RCU/RLU grace period detection mechanism (line 52-54 in Algorithm 1) to avoid overwriting in-use objects' versions. Therefore, while the total throughput of Pisces still grows as the thread number increases from 22 to 40, the growth speed is much slower than that within a single NUMA node.

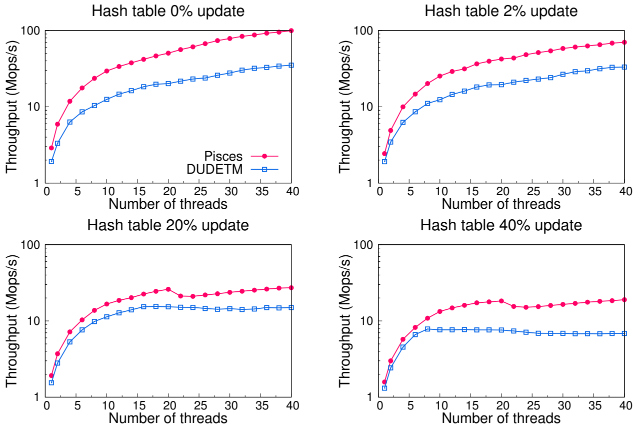

Fig. 6: The throughput of hash table (8-byte key and 64-byte value) at various update rates (legends in the first figure).

The throughput of DUDETM grows from ∼1.5 Mops/s to ∼15 Mops/s as the thread number increases from 1 to 16. However, DUDETM's throughput cannot keep growing or even decreases a little when the thread number becomes larger. The reason is that the background threads fails to timely clean up the logs of transactions. Note that the reproduce thread has to modify the persistent objects according to the logs in the order of transaction execution, and make the modifications persistent with cacheline flush instructions. In contrast, Pisces lets each execution thread to persist and write-back the transactions' logs, which can better utilize the NVM write bandwidth.

Table 1: The average cost of a read-write transaction and one grace period detection in Pisces's hash table. The update rate is 40%.

#Thread	10	20	30	40
RW TX Latency (cycles)	2902	3986	7180	8496
Grace Period (cycles)	357	790	1689	2140

The performance of DUDETM and Pisces at 40% update rate shows similar trends with those at 20% update rate. Nev-

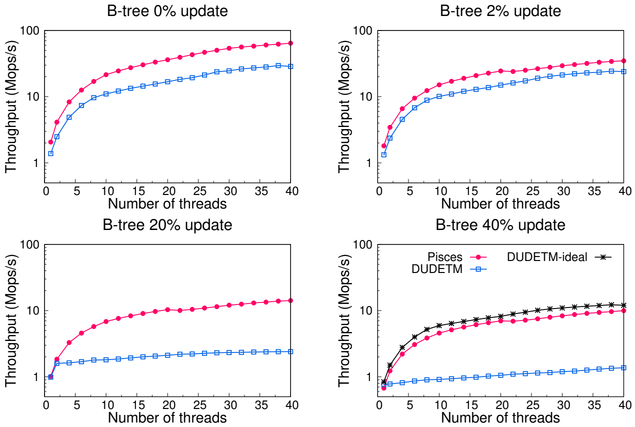

Fig. 7: The throughput of B+-tree whose node size is 256 bytes with various update rates (legends in the last figure).

ertheless, the throughput of DUDETM only grows within 8 threads because a higher update rate means more read-write transactions (generate more logs and fill the log buffer earlier). For Pisces, the growth speed of throughput becomes lower when the thread number exceeds 20. As presented in Table 1, the cost of grace period increases as the thread number increases. The reason is one thread has to check other threads' status for detecting the grace period.

In the case of 40 threads, Pisces's throughput is about 1.8× and 2.7× of DUDETM's when the update rate is 20% and 40%, respectively. And, its persistency cost is 19% at 40% update rate. Besides, the abort rates of both Pisces and DUDETM (if no blocking) are nearly zero since the hashing mitigates the contention among different threads. However, DUDETM's abort rate increases (up to 9%) if blocking happens because a thread may get blocked with holding locks.

We also evaluate the hash table with an occupancy of 0.75, i.e., 10K buckets and 7.5K key-value pairs. The evaluation results show similar trends. Specifically, at 20% update ratio, the throughput of DUDETM grows from ∼3.3 Mops/s to ∼15.9 Mops/s as the thread number increases from 1 to 16. As before, its throughput cannot grow when the thread number is larger than 16. Pisces's throughput grows from ∼2.7 Mops/s to ∼35.9 Mops/s as the thread number increases from 1 to 20. Nevertheless, when the thread number is 1 or 2, DUDETM has a higher throughput than Pisces for two reasons: first, DUDETM leverages extra CPUs (background threads) for persisting data; second, each transaction reads fewer data due to lower occupancy, which mitigates the benefits of read-friendly design in Pisces.

B+-tree. We construct B+-trees in which each node contains at most 16 children and randomly insert about 1 million key-value pairs at the beginning of each test. Figure 7 shows the evaluation results of executing search and insert transactions (an insert transaction will modify the target key-value pair if the pair already exists) on B+-trees. Each transaction goes down from the root node to some leaf node. For an insert transaction, before going down to some node, it first checks if the node is full. If the node is full, it splits the node for creating space. Since we only implement delete operations as marking the target node as deleted, we run every test for 10 seconds in case the trees get too large.

Similar to the results of hash table benchmark, both Pisces and DUDETM can scale well to 40 threads when the update rate is low such as 0% and 2%, because the number of NVM writes is small. Owing to the read-friendly designs, Pisces shows a better performance than DUDETM. Nevertheless, the performance gap between Pisces and DUDETM decreases when the update rate changes from 0% to 2% because Pisces synchronously persists data into NVM while DUDETM hides the persistence overhead through asynchronously persisting the data in the background.

At 20% update rate, DUDETM's throughput is almost the same as Pisces's when there is a single execution thread. However, DUDETM can only scale to two threads while Pisces has much better scalability. The scalability issue of DUDETM arises earlier in B+-tree benchmark than in hash table benchmark because the read-write transactions in B+-tree generates more log and thus burden the reproduce thread more. Enlarging the size of log buffer can mitigate/hide the problem to some extent but cannot eliminate/solve this problem. At 40% update rate, DUDETM actually outperforms Pisces when the thread number is 1. However, its scalability issue gets worse because of the higher update rate.

At 40% update rate, DUDTEM's throughput grows from 763 Kops to 1370 Kops as the thread number increases from 1 to 40. The reason is each execution thread has one volatile log buffer. Therefore, the total throughput grows a little when adding more threads (more buffer). Nevertheless, longer running time will further flatten the throughput.

To clearly show that the reproduce thread blocks the execution threads and restricts the overall performance, we also evaluate the performance of *DUDETM-ideal* in which the background reproduce thread directly marks the persistent log area as free without writing back the logs in it to persistent objects in NVM. In fact, *DUDETM-ideal* emulates the performance of DUDETM with an infinite persistent log buffer. As shown in the last sub-figure in Figure 7, *DUDETM-ideal* scales very well to 38 threads. The reason for the performance drop in 40 threads is the total thread number (40 execution threads together with 2 background threads) exceeds the hardware thread number (40 hyper-threads). Since the only difference between *DUDETM-ideal* with DUDETM is whether the background reproduce thread really flushes data to NVM, we can conclude the centralized reproduce thread severely harms the system's scalability.

When the update rate is 20% or 40%, the throughput of Pisces almost keeps growing as the thread number increases to 40. And, the persistency cost is 36% at 40% update rate. The NUMA problem in B+-tree is less severe than that in the hash table since the total throughput is lower in B+-tree

Table 2: The average cost of a read-write transaction and one grace period detection in Pisces's B+-tree. The update rate is 40%.

#Thread	10	20	30	40
RW TX Latency (cycles)	9,374	12,024	15,210	16847
Grace Period (cycles)	1,162	2,021	2,915	3528

(less read-write transactions). Nevertheless, the performance growth speed still becomes slower when the thread number exceeds 20. Table 2 presents the cost of grace period detection in Pisces's B+-tree.

5.3 Real-world Benchmarks

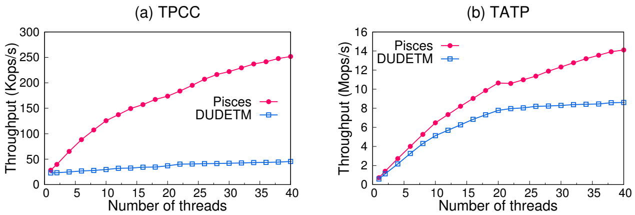

Fig. 8: (a) TPC-C new-order transactions and (b) TATP.

We also evaluate macro-benchmarks (i.e., TPC-C and TATP) which are tested in DUDETM. Besides, we further evaluate Pisces and DUDETM on *kmeans*, *ssca2*, and *vacation* which are popular transactional memory benchmarks [57, 58].

TPC-C. TPC-C is an online transaction processing (OLTP) benchmark. We implement its *new-order transaction* [24] with B+trees whose nodes contain at most 32 children as the tables. In this experiment, each execution thread works on its corresponding warehouse and executes new-order transactions (the update rate is 100%). On average, each transaction involves inserting over 10 new objects into different tables as well as modifying over 10 existing objects, which generates much more logs than the transactions in the previous micro-benchmarks.

Figure 8-(a) gives the evaluation results. Since there is no conflict among transactions from different execution threads, the throughput of Pisces continues to grow with the increase of thread number. There are no transaction aborts in this experiment for Pisces. For fairness, we also modify the TinySTM used by DUDETM (enlarge the *LOCK_ARRAY_LOG_SIZE*) to avoid false sharing of address locks and reduce the abort rate in DUDETM to zero. However, the throughput of DUDETM at 40 threads is only twice of that at 1 thread.

The evaluation results clearly demonstrate that Pisces utilizes the NVM bandwidth in a much better way than DUDETM. A centralized log-reproducing thread can hardly catch up with the progress of multiple execution threads. So the execution threads fill the log buffers in DUDETM. Once the log buffers are full, all the execution threads are blocked, and then the whole system's progress relies on the background log-reproducing thread. Instead of flushing data to NVM in a centralized way, Pisces lets each thread make the transactions' persistent and thus allows more parallelism in the NVM persistence operations. Since NVM device can serve the memory operation requests from different CPUs at the same time, the throughput of Pisces grows from ∼28 Kops/s to ∼252 Kops/s as the thread number increases from 1 to 40. The NVM hardware bandwidth limit is still not reached, inferred from the growth in throughput.

Compared with DUDETM, Pisces can achieve about 4.6× speedup when the thread number is 40. And, the persistency cost is about 50%.

TATP. TATP benchmark [72] is another OLTP application. We implement three read-only transactions and three read-write transactions of it. We use the same B+Tree in the TPC-C experiments as the data structure of tables, set the update ratio to 18% and initialize the population size to 200,000.

Different from the new-order transactions in TPC-C, the read-write transactions in TATP are much smaller. For example, the *update-location transaction* that occupies 14% of the total transactions only update one single existing object. So, each TATP read-write transaction involves less NVM writes than TPC-C transactions and even less than the B+-tree micro-benchmark in which most read-write transactions insert a new object. Thus, DUDETM can scale to 22 threads. However, its scalability issue still comes up when the thread number gets larger.

For Pisces, the throughput grows from ∼0.7 Mops/s to ∼14 Mops/s as the thread number increases from 1 to 40. Pisces's peak performance is about 64% higher than DUDETM's. And, the persistency cost is 26%. The NUMA issue also appears in this benchmark. The RCU grace period detection cost is higher if the total throughput of read-write transactions is higher because of the higher (cross NUMA nodes) cacheline contention (i.e., checking other thread's status). This is also why the NUMA issue is not obvious in the TPC-C benchmark. The NUMA issue also leads to a slower growth speed of the throughput after the thread number exceeds 20.

TM Applications. Referring to [7], we implement and evaluate *kmeans*, *ssca2*, and *vacation* in both Pisces and DUDETM. Currently, we only persist data that are surrounded/protected by TM interfaces.

Table 3: The execution time of *kmeans* (shorter is better).

#Threads	1	2	4	8	16	32
DUDETM (s)	5.5	5.0	3.8	2.6	0.25	0.54
Pisces (s)	3.2	2.4	1.3	0.8	0.7	1.2

Kmeans is a machine learning application and this experiment test it with low contention and medium data set. Table 3 shows the execution time. In this benchmark, both DUDETM and Pisces scale to 16 threads. However, there is a performance drop for both systems when the thread number increases from 16 threads to 32 threads, because they both suffer from high abort rates (84% for DUDETM and 77% for Pisces). The performance of Pisces is also bottlenecked by grace periods' cost which is enlarged by NUMA. It is also worth mentioning that the foreground threads in DUDETM get blocked by full volatile logs (slow background threads) when the thread number is less than 8. As a result, DUDETM has an obvious performance improve when the thread number increases from 8 to 16. For the same reason, DUDETM performs worse than Pisces with no more than 8 threads. When there are more than 8 threads, DUDETM has better performance than Pisces because the foreground threads in DUDETM neither need to persist write transactions nor get blocked by background threads.

In this benchmark, with the increase of thread number, foreground threads in DUDETM are less likely to get blocked by the reproduce thread. The reason is that *kmeans* evenly distribute a specific amount of work to the foreground execution threads and thus each thread executes less transactions when there are more threads. Specifically, to finish this benchmark, all the threads need to commit 1M transactions in total and generate 40M log entries. When the thread number is more than 16, the foreground threads in DUDETM do not get blocked since each of them has a volatile buffer with 1M log entries. Nevertheless, when testing with large data set (10M transactions and 400M log entries), the foreground threads in DUDETM still get blocked when there are 32 threads.

Table 4: The execution time of *ssca2* (shorter is better).

#Threads	1	2	4	8	16	32
DUDETM (s)	17.2	13.1	12.0	11.6	9.3	9.3
Pisces (s)	18.8	13.8	8.4	5.4	3.6	3.7

Scalable Synthetic Compact Applications (*ssca2*) simulates the computation on graphs. Table 4 gives the evaluation result of ssca2 with medium data set (2^{18} nodes). Different from kmeans, ssca2 involves a larger number of transactions and DUDETM cannot scale well since the background reproduce thread cannot timely consume the logs. So for DUDETM, 32 threads cannot finish this benchmark faster than 16 threads since the execution threads get blocked. Nevertheless, if evaluating ssca2 benchmark with the small data set (2^{13} nodes), DUDETM can scale well but the performance of Pisces also gets much better.

Pisces scales better than DUDETM in this benchmark. However, the execution time of Pisces is longer than that of DUDETM when the thread number is 1 and 2. This is because foreground threads in DUDETM do not make the transactions' updates persistent, and we only calculate the runtime of the foreground threads. Similar to previous experiments, Pisces suffers from NUMA problem again. Since the throughput in this benchmark is high (executes ∼11M transactions in total) and the update rate is 100%, the NUMA problem is more severe and thus causes the performance to drop when threads number changes from 16 to 32.

Table 5: The execution time of *vacation* (shorter is better).

#Threads	1	2	4	8	16	32
DUDETM (s)	10.0	5.2	2.5	1.2	0.8	0.4
Pisces (s)	8.2	4.7	2.7	1.6	0.9	0.6

Vacation is an OLTP system which emulates a travel reservation system. The *vacation* benchmark has 100% update ratio, and each transaction has bigger read/write sets. We use hash tables to implement tables in the benchmark. Table 5 shows the evaluation result of this benchmark with medium data set and low contention.

As shown in Table 5, the execution time of both DUDETM and Pisces decreases as the thread number increases in this benchmark. In the case of 1 and 2 threads, Pisces performs better than DUDETM. The reason is that DUDETM introduces software overhead for the read operations in the transactions. Since the read sets are large, the software overhead such as read validation is non-negligible. But DUDETM scales well since this workload (executes 400K transactions in total) does not cause the full log problem.

Overall, Pisces does not scale as well as DUDETM. As the benchmark is update-only and the average transaction latency is long (big transaction), the grace period detection mechanism in Pisces becomes more time-consuming and brings high overhead. Hence, with the increase of thread number, transactions spend more time in the grace period detection mechanism, which leads to dissatisfactory performance. On average, the mechanism costs each transaction 30,433 cycles and 46,070 cycles when the thread number is 16 and 32, respectively. And the average transaction latency is about 168,000 cycles when there are 32 threads.

5.4 Other Performance Analysis

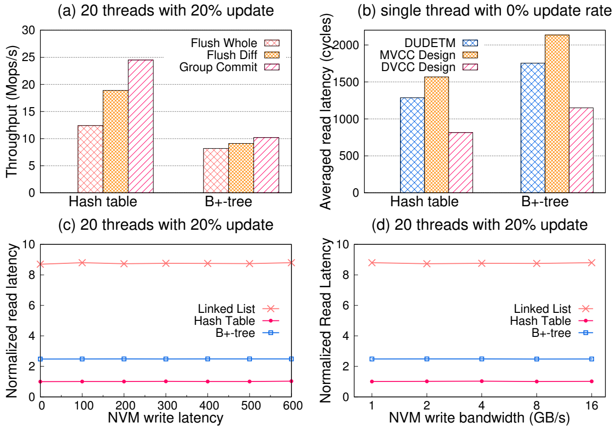

Fig. 9: (a) The performance gain break down. (b) The performance of read-only transactions. Read latency in Pisces with (c) increased NVM write latency and (d) increased NVM write bandwidth.

Figure 9-(a) shows step-wise performance gain from the optimizations in Pisces. *Flush Diff* improves the throughput of hash table and B+-tree by 52% and by 11%, respectively, because it effectively reduces the number of NVM writes. Since a write transaction on a hash table changes a small portion of the nodes, there is more data that does not need to be logged and written back. However, a transaction on a B+-tree usually involves more data modification, especially, when node splitting is required. So *Flush Diff* benefits hash table more in the presented settings. *Group Commit* (setting group number as 3) brings a performance gain of 30% for hash table and 12% for B+-tree, through reducing the number of grace period detection and *fetch_and_add* instructions on the global timestamp. The performance improvement in hash table is larger for two reasons. First, Pisces has a higher throughput in the hash table benchmark which means higher contention on the global timestamp. Second, the grace period detection in hash table takes a higher percentage of cost in a write transaction than that in B+-tree. Note that the current *Group Commit* implementation will increase the latency of write transactions. Nevertheless, Pisces can only batch the write-back stages instead of the whole commit phases to mitigate this issue.

Figure 9-(b) compares the average latency of read-only transactions in micro-benchmarks. Compared with our MVCC-based design, the DVCC design in Pisces significantly reduces the read latency (about 2× faster). The reason is that each read operation in the MVCC-based design involves locating the version list of an object and traversing the list, leading to at least one more random memory access (i.e., read indirection). Compared with DUDETM (needs address translation and read validation), Pisces's read operations are also faster owing to the read-friendly design. For the hash table benchmark, the read-only transaction in Pisces is faster than DUDETM's by 472 cycles. While for the B+-tree benchmark, the read-only transaction in Pisces is faster than DUDETM's by 605 cycles. This is because a read transaction in B+-tree contains more read operations.

Different persistent memory technologies may have different persistent cost. We further use an NVM emulator which explicitly add delays to the NVM flush operations according to the NVM write latency and bandwidth (similar to prior work [17, 37, 49, 49]). As shown in Figure 9-(c) and Figure 9-(d), the average latency[5] of read-only transactions, each of which searches for an element in the corresponding data structure, is stable with various NVM write latency and bandwidth. The reason is that Pisces avoids writers blocking readers. Although the NVM write latency and bandwidth affect the latency of write operations in Pisces, the latency of read-only transactions is insensitive to that of write operations. Therefore, Pisces produces a stable average latency of read-only transactions with increased NVM write latency and bandwidth. Other experiments with different thread numbers or different update rates give similar results.

6 Related Work

Compared with most PTM designs (Mnemosyne [74], NV-Heaps [21], Kamino-Tx [56], and DCT [44]) which reduce the persistence latency of write transactions through various novel techniques but may expose NVM persist overhead to readers, Pisces focuses on benefiting read operations and can always hide NVM persist latency from readers. A most recent PTM named Romulus [23] promises never blocking read-only transactions through maintaining twin copies of the durable data. Pisces shares a similar idea to avoid blocking read-only transactions and further exploits SI to avoid blocking any read operation. Romulus instruments loads and stores to NVM through programming language feature, which is elegant and can be borrowed to Pisces. Besides, Romulus chooses a single writer design which can reduce the average number of fences for write transactions, however, limits the concurrency of NVM persistence. Some recent studies [42, 60, 71] leverage hardware modifications to implement efficient PTM systems. NVM is also exploited by in-memory database systems [27, 43] and new file systems [31, 32, 80]. Others [5, 37, 40, 41, 47] provide libraries for applications to utilize NVM.

Transactional Memory (TM) has been well studied [10, 14, 29, 34, 39, 66, 68, 70]. Some studies [36, 73] propose non-blocking designs and some others also investigate snapshot isolation to TM [48, 68], which significantly reduces the abort rates. But they do not consider crash consistency under NVM. Matveev et al. [54] propose a novel and lightweight synchronization mechanism (RLU) for concurrent programming. DVCC in Pisces is inspired by both MVCC and RLU. Thus, Pisces and RLU share a couple of similarities including maintaining two versions and allowing readers to read the write sets of writers. However, RLU neither provides a transactional programming semantic (no snapshot isolation) nor considers NVM (no durability and crash consistency).

7 Summary

This paper presents Pisces, a read-friendly PTM that provides transactional memory APIs for programming on NVM. With several techniques such as DVCC and three-stage commit, Pisces achieves both high throughput and good scalability while ensuring snapshot isolation and crash consistency.

8 Acknowledgement

We sincerely thank our shepherd Michael Swift and the anonymous reviewers for their insightful suggestions. Also, we sincerely thank the authors of DUDETM for giving guidance. This work is supported in part by China National Natural Science Foundation (No. 61672345). Zhaoguo Wang is the corresponding author.

[5] We set the read latency in hash table in the case of zero NVM write latency to 1. Other results are normalized against it.

References

[1] The formal proof on the snapshot isolation guarantee. https://ipads.se.sjtu.edu.cn/_media/publications/guatc19_proof.pdf.

[2] Intel 3dxpoint technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[3] Intel 64 and ia-32 architectures software developer's manual. https://software.intel.com.

[4] Micron 3dxpoint technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[5] Persistent memory development kit, nvml. http://pmem.io/pmdk/.

[6] Redis. https://redis.io/.

[7] The stanford transactional applications for multi-processing; a benchmark suite for transactional memory research. https://github.com/kozyraki/stamp.

[8] Atul Adya. Weak consistency: a generalized theory and optimistic implementations for distributed transactions. 1999.

[9] Hiroyuki Akinaga and Hisashi Shima. Resistive random access memory (reram) based on metal oxides. *Proceedings of the IEEE*, 98(12):2237–2251, 2010.

[10] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 4–18. Springer, 2009.

[11] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1327–1342. ACM, 2015.

[12] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[13] Philip A Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[14] Jayaram Bobba, Neelam Goyal, Mark D Hill, Michael M Swift, and David A Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 127–138. IEEE Computer Society, 2008.

[15] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[16] Michael J Cahill, Uwe Röhm, and Alan D Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)*, 34(4):20, 2009.

[17] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment*, 8(5):497–508, 2015.

[18] Haibo Chen, Heng Zhang, Ran Liu, Binyu Zang, and Haibing Guan. Fast consensus using bounded staleness for scalable read-mostly synchronization. *IEEE Transactions on Parallel & Distributed Systems*, (12):3485–3500, 2016.

[19] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. Tpc-e vs. tpc-c: characterizing the new tpc-e benchmark via an i/o comparison study. *ACM SIGMOD Record*, 39(3):5–10, 2011.

[20] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243. ACM, 2013.

[21] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 46(3):105–118, 2011.

[22] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[23] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 271–282. ACM, 2018.

[24] Transaction Processing Performance Council. http://www.tpc.org/tpcc/. *TPC Benchmark C*.

[25] Transaction Processing Performance Council. http://www.tpc.org/tpce. *TPC Benchmark E*.

[26] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd international conference on Very large data bases*, pages 715–726. VLDB Endowment, 2006.

[27] Justin DeBrabant, Joy Arulraj, Andrew Pavlo, Michael

Stonebraker, Stan Zdonik, and Subramanya Dulloor. A prolegomenon on oltp database systems for non-volatile memory. *ADMS@ VLDB*, 2014.

[28] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM, 2013.

[29] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *International Symposium on Distributed Computing*, pages 194–208. Springer, 2006.

[30] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[31] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (ATC 17)*, pages 719–731. USENIX Association, 2017.

[32] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 15. ACM, 2014.

[33] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005.

[34] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

[35] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM, 2008.

[36] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2):5, 2007.

[37] Ellis R Giles, Kshitij Doshi, and Peter Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–14. IEEE, 2015.

[38] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[39] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 31. ACM, 2016.

[40] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 468–482. ACM, 2017.

[41] Qingda Hu, Jinglei Ren, Anirudh Badam, and Thomas Moscibroda. Log-structured non-volatile main memory. In *Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC'17). Santa Clara, CA. http://jinglei. ren. systems/files/lsnvmm_slides_atc17. pptx*, 2017.

[42] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Dhtm: Durable hardware transactional memory. In *Proceedings of the International Symposium on Computer Architecture*, 2018.

[43] Hideaki Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 691–706. ACM, 2015.

[44] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. *ACM SIGPLAN Notices*, 51(4):399–411, 2016.

[45] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 256–267. IEEE, 2013.

[46] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE micro*, 30(1), 2010.

[47] Herwig Lejsek, Friðrik Heiðar Ásmundsson, Jónsson, and Laurent Amsaleg. Nv-tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):869–883, 2009.

[48] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P Stevenson. Si-tm: reducing transactional memory abort rates through snapshot isolation. *ACM SIGARCH Computer Architecture News*, 42(1):383–398, 2014.

[49] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–343. ACM, 2017.

[50] Ran Liu and Haibo Chen. Ssmalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the Asia-Pacific Workshop on Systems*, page 15. ACM, 2012.

[51] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *USENIX Annual Technical Conference*, pages 219–230, 2014.

[52] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The snow theorem and latency-optimal read-only transactions. In *OSDI*, pages 135–150, 2016.

[53] Shiyong Lu, Arthur Bernstein, and Philip Lewis. Correct execution of transactions at different isolation levels. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1070–1081, 2004.

[54] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 168–183. ACM, 2015.

[55] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[56] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *EuroSys*, pages 499–512, 2017.

[57] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46. IEEE, 2008.

[58] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 135–148. ACM, 2017.

[59] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 677–689. ACM, 2015.

[60] Matheus Almeida Ogleari, Ethan L Miller, and Jishen Zhao. Steal but no force: Efficient hardware undo+ redo logging for persistent memory systems. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 336–349. IEEE, 2018.

[61] Lois Orosa and Rodolfo Azevedo. Logsi-htm: Log based snapshot isolation in hardware transactional memory.

[62] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 265–276. IEEE Press, 2014.

[63] Hasso Plattner. The impact of columnar in-memory databases on enterprise systems: implications of eliminating transaction-maintained aggregates. *Proceedings of the VLDB Endowment*, 7(13):1722–1729, 2014.

[64] Dan RK Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. *Proceedings of the VLDB Endowment*, 5(12):1850–1861, 2012.

[65] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 206–220, New York, NY, USA, 2005. ACM.

[66] Hany E Ramadan, Christopher J Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 246–257. IEEE Computer Society, 2008.

[67] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[68] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, pages 1–10. Association for Computing Machinery (ACM), 2006.

[69] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. Accelerating analytical processing in mvcc using fine-granular high-frequency virtual snapshotting. In *Proceedings of the 2018 International Conference on Management of Data*, pages 245–258. ACM, 2018.

[70] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[71] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. Proteus: a flexible and fast software supported hardware logging approach for nvm. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 178–190. ACM, 2017.

[72] Neuvonen Simo, Wolski Antoni, manner Markku, and Raatikka Vilho. http://tatpbenchmark.sourceforge.net/. *Telecom Application Transaction Processing Benchmark*.

[73] Fuad Tabba, Mark Moir, James R Goodman, Andrew W Hay, and Cong Wang. Nztm: Nonblocking zero-indirection transactional memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 204–213. ACM, 2009.

[74] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 91–104. ACM, 2011.

[75] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. Efficiently making (almost) any concurrency control mechanism serializable. *The VLDB Journal*, 26(4):537–562, 2017.

[76] Zhaoguo Wang, Han Yi, Ran Liu, Mingkai Dong, and Haibo Chen. Persistent transactional memory. *IEEE Computer Architecture Letters*, 14(1):58–61, 2015.

[77] Wikipedia. https://en.wikipedia.org/wiki/Snapshot_isolation. *Snapshot isolation*, 2017.

[78] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[79] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (ATC 17)*, pages 349–362. USENIX Association, 2017.

[80] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, pages 323–338, 2016.

[81] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2015.