PETRA: Persistent Transactional Non-blocking Linked Data Structures

RAMIN IZADPANAH, CHRISTINA PETERSON, YAN SOLIHIN, and DAMIAN DECHEV,
University of Central Florida, USA

Emerging byte-addressable Non-Volatile Memories (NVMs) enable persistent memory where process state can be recovered after crashes. To enable applications to rely on persistent data, durable data structures with failure-atomic operations have been proposed. However, they lack the ability to allow users to execute a sequence of operations as transactions. Meanwhile, persistent transactional memory (PTM) has been proposed by adding durability to Software Transactional Memory (STM). However, PTM suffers from high performance overheads and low scalability due to false aborts, logging, and ordering constraints on persistence.

In this article, we propose PETRA, a new approach for constructing persistent transactional linked data structures. PETRA natively supports transactions, but unlike PTM, relies on the high-level information from the data structure semantics. This gives PETRA unique advantages in the form of high performance and high scalability. Our experimental results using various benchmarks demonstrate the scalability of PETRA in all workloads and transaction sizes. PETRA outperforms the state-of-the-art PTMs by an order of magnitude in transactions of size greater than one, and demonstrates superior performance in transactions of size one.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**; *Shared memory algorithms*; • **Hardware** → *Memory and dense storage*;

Additional Key Words and Phrases: Persistent memory, concurrency, transactional data structure, non-blocking data structure, non-volatile memory, durability

ACM Reference format:

Ramin Izadpanah, Christina Peterson, Yan Solihin, and Damian Dechev. 2021. PETRA: Persistent Transactional Non-blocking Linked Data Structures. *ACM Trans. Archit. Code Optim.* 18, 2, Article 23 (March 2021), 26 pages.

<https://doi.org/10.1145/3446391>

1 INTRODUCTION

With persistent or non-volatile memory (NVM) recently becoming available commercially, there has been a surge of interest in utilizing it not only as a high-capacity main memory (e.g., Optane DC PM with 3TB per socket [42]), but also for hosting persistent/durable data. To enable

This work was supported by NSF grant Nos. 1914717 and 1740095 and by Sandia National Laboratories, which is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Authors' addresses: R. Izadpanah, C. Peterson, Y. Solihin, and D. Dechev, University of Central Florida, Department of Computer Science, 211 Harris Center (Building 116), 4000 Central Florida Boulevard, Orlando, FL, 32816, USA; emails: {r.izadpanah, clp8199}@knights.ucf.edu, yan.solihin@ucf.edu, dechev@cs.ucf.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.

1544-3566/2021/03-ART23

<https://doi.org/10.1145/3446391>

Table 1. Taxonomy of Persistent Data System

	Transactional	Non-transactional
High-Level (Data structure semantics)	PETRA (This paper)	Persistent data structures
Low-Level (Reads/writes)	Persistent Transactional Memory (PTM)	N/A

applications to rely on persistent data, approaches to construct persistent data systems are proposed. In general, such systems can be categorized based on two aspects, shown in Table 1. One aspect is whether they support transactions as primitives or not. A transaction supports ACID: *atomicity* (all operations must all succeed or none does), *consistency* (the data structure state is consistent before and after the transaction), *isolation* (concurrent executions of transactions appear to take effect in some sequential order), and *durability* (effects of a transaction are not lost upon a power failure). Another aspect is if the persistent data system relies on low-level information, such as reads and writes, or relies on high-level information such as data structure semantics.

In one approach, specific *persistent data structures*, such as list, set, tree, queue, and hash map have been proposed [15, 29, 64, 65, 94]. These data structures allow the application to execute individual operations in a crash atomic manner, but transactions are not supported. A problem arises when an application may need to execute not just single operations atomically, but a sequence of operations atomically, i.e., as transactions. Consider a transaction that moves a node from one persistent set to another, i.e., $\{\text{set1.delete}(x); \text{set2.insert}(x)\}$. If a crash were to occur after $\text{set1.delete}(x)$ and before $\text{set2.insert}(x)$, then $\text{node}(x)$ could be permanently lost. While individual crash atomic operations are useful, a transaction allows both operations to make durable changes to both sets atomically. Executing transactions on data structures is an essential functionality [72], especially in applications such as databases [62], data analytics tools [31, 61, 86], and solving complex graph problems [22, 47]. The ability to provide durable transactions is a vital aspect of these applications, because it guarantees prevention of data loss, which is critical in high performance computing due to the risk of losing computationally expensive operations that have completed or are in progress. Furthermore, to support a broad spectrum of applications, a more general framework is needed beyond individual data structure designs.

In another approach, such as *Persistent Transactional Memory* (PTM) [80], researchers provide transactional support by adding durability to general-purpose transactional programming models. PTMs typically use an underlying Software Transactional Memory (STM) to allow for the execution of transactions atomically. Since STM already supports “ACI,” PTM only needs to add durability (“D”) to STM. Unfortunately, the reliance on STM results in inheriting its limitations. One issue is that STM relies on low-level information of memory accesses (reads/writes) for conflict detection, which is attributed to the problem of frequent false aborts [6, 38], incurred by false conflicts stemming from high contention on the data structure’s points of accesses, e.g., the top pointer of a stack. Transactions that have read the top of the stack are aborted if another transaction writes to the top of the stack, even though they might not conflict based on data structure semantics, leading to wasted computational resources due to restarting the transactions [38]. Furthermore, supporting durable transactions requires maintaining (undo or redo) logs, which incurs performance overheads.

In this article, we propose a new approach for constructing a persistent transactional data system for linked data structures (PETRA). Our approach, PETRA, is the first to rely on a transactional

approach combined with the high-level information from the data structure semantics. Beyond having a unique approach, the goals of our design are:

- **High performance:** low overheads added to achieve durability.
- **High scalability:** performance scaling well with increasing thread counts.
- **Non-blocking:** there is guaranteed system-wide progress.

PETRA achieves high performance by keeping the number of cache line flushes and memory fences low. Consider descriptor objects [34], commonly used in the design of lock-free data structures [16, 20, 26, 35, 91, 92]. Our descriptor object (*transaction descriptor*) contains the information needed to execute a transaction. All nodes that are accessed by a transaction hold a reference to a shared transaction descriptor. We observe that transaction descriptors can be utilized to verify the consistency of the underlying data structures after a crash and correct possible inconsistencies. These observations lead us to leverage transaction descriptors as redo logs instead of introducing additional logging constructs. Removal of explicit logging not only leads to fewer instructions to execute, but also relaxes the ordering constraints between persistent memory operations, leading to the removal of many persist barriers (cache line flushes and store fences). In contrast, PTMs typically need to enforce orderings between writes to the log and writes to the actual data structure [54, 57]. In PETRA, enforcing the persistence of the transaction descriptors at the end of the transaction is sufficient. Furthermore, since the transaction descriptor is already needed to manage concurrency, leveraging it to manage crash consistency adds only minor additional overheads. A limitation of PETRA is that it is optimized for failure-free execution at the expense of a recovery time that is proportional to an unbounded number of transaction descriptors. Periodic checkpointing, an approach that is orthogonal to PETRA, can be used to place a bound on the number of transaction descriptors visited during recovery by ensuring consistency up to the checkpoint [12, 21, 64, 83].

PETRA achieves high scalability by adopting a technique that works at the data structure semantics level [38, 92], hence removing false aborts. Aborts only occur when transactions conflict on the data structure nodes. Again, this is achieved, thanks to transaction descriptors. Descriptors enable a node-based conflict detection scheme that does not rely on transactional memory nor require the use of an additional data structure. Furthermore, when conflicts are detected, PETRA uses the transaction descriptors to implement a recovery strategy based on the interpretation of the logical status of nodes [92] instead of explicitly revoking executed operations on an abort.

Finally, PETRA is an obstruction-free transactional persistent data system, where system-wide progress is guaranteed. *Obstruction freedom* guarantees that a single thread executed in isolation is guaranteed to complete its operation in a finite number of steps. PETRA does not rely on using locks, hence deadlocks are not possible. With locks, if a thread holding a lock is pre-empted or if it crashes, no system progress can be made. With a persistent data system, if the lock is persistent, post-crash recovery is not simple, as we have to recover the thread that held the lock at the crash time [15]. With PETRA, no locks are used. PETRA utilizes transaction descriptors that ensure global progress through a helping mechanism. If a thread is pre-empted while executing a transaction, another thread can help complete the transaction.

PETRA is also applicable to non-linked object-based containers such that container elements can hold a reference to a descriptor. Examples include vector, ring buffer, and hash map comprising objects that are contiguous in memory.

To summarize, this article makes the following contributions:

- We introduce a methodology to build persistent transactional linked data structures that achieves high performance, high scalability, and non-blocking progress.

- We propose to use the data structure semantic knowledge to implement an efficient transaction logging mechanism that ensures consistency and durability, replacing traditional logging that relies on memory accesses and incurs a large number of flushes and fences.
- We demonstrate PETRA's advantages using micro-benchmarks, transaction and database benchmarks. It outperforms the state-of-the-art PTMs by more than one order of magnitude in transactions with more than one data structure operation and exhibits superior performance in the baseline scenario with transactions of size one.

2 PRELIMINARIES AND RELATED WORK

In this section, we provide an overview of the concepts, techniques, and tools used by PETRA and discuss the related work that proposed transactional executions of data structures.

2.1 Related Work

Non-blocking Progress Assurance. Concurrent data structures can either be *blocking* or *non-blocking* based on the progress guarantee they provide. Blocking data structures do not provide system-wide progress guarantees, such as the completion of one or more operations. Non-blocking data structures allow scalable and thread-safe access to shared data while providing progress guarantees that are not possible with the use of locks. The correctness of such algorithms is typically established by relying on a key correctness condition, linearizability [37], and its more relaxed derivatives.

One of the techniques employed in non-blocking data structures is the use of descriptor objects [16, 20, 26, 35, 91, 92], which are shared objects that keep the required information for executing data structure operations and allow several updates to take effect atomically. This shared object allows for cooperation between threads. When a thread stalls, another thread can read the descriptor object for it and execute its operations according to the information provided by its descriptor. Such a thread is referred to as a *helper thread* and the act as *helping*.

Persistent Data Structures. We consider a concurrent data structure “persistent transactional” if it provides the full ACID guarantee [32]. Specific persistent data structures such as list and set [15, 94], tree [9, 15, 52, 53, 65, 78, 81, 88], queue [29], hash map [15, 64, 71] have been proposed, with each operation designed to keep data in the containers persistent. The idea of building the structure of the containers upon recovery has been proposed recently [59, 94]. However, to the best of our knowledge, they do not provide native support for ACID transactions. To execute transactions on these data structures, they can be integrated with a PTM-based approach. PETRA introduces a methodology for adding transactional persistency to concurrent linked data structures, instead of specific non-transactional data structures.

(Persistent) Transactional Memory. The typical transactional programming model allows for the transactional execution of atomic blocks specified by the programmer. Transactional memory was first proposed as a hardware primitive (HTM) [39] that relies on the cache to buffer a thread's speculative state. Implementations on real processors (e.g., Intel TSX) do not guarantee progress, as a transaction with size exceeding the cache size aborts the transaction. STM [73] is its software counterpart that is not restricted with transaction size but incurs higher performance overheads. It performs conflict detection using memory-level read and write sets. STM's lack of knowledge about the high-level data structure semantics introduces false aborts, which restricts concurrency and scalability.

Recently, researchers have added durability to HTM [46, 74], or STM to form PTM [2, 3, 80]. To support durability, software PTMs rely on a (redo or undo) logging mechanism. The log structure introduces substantial performance overhead. For example, an undo log must persist before

Table 2. Comparison of Non-durable Transactional Data Structure Methodologies

Transactional Methodology	Semantic Conflict Detection	TX Logging	Non-blocking Progress
LFTT [92]	✓	✓	✓
TDSL [76]	✓		
Transactional Boosting [38]	✓		
STO [41]	✓	✓	
STM [28]		✓	✓

They all support Atomicity, Isolation, and Consistency, but not Durability. Check-marks indicate existing features.

the data structure can persist and in the case of using redo logs, a traversal of the write set for concurrent read operations is needed [57]. Examples include PMDK [67], Atlas [7], JUSTDO [43], iDO [55], NV-Heaps [11], Mnemosyne [80], Romulus [14], Kamino-Tx [58], and generic STM transformation methods [89]. Some PTMs such as JUSTDO [43], PHTM [3], and PHyTM [2] require special hardware support not available in commercial processors. Mnemosyne [80] is built on top of TinySTM [25] and uses a redo log. Kamino-Tx removes the logging overhead from the critical path by keeping two copies of persistent data. In case of dependency between transactions in Kamino-Tx, the dependent transaction holds locks for the objects in its working set until the transaction is committed. Romulus [14] relies on data redundancy instead of persistent logs. One-File [70] is a variant of Romulus based on a universal construction [36], which tends to be expensive because of the overhead that is mainly incurred by its maintenance and instrumentation [10, 24, 37, 56]. There are multiple problems with PTMs. First, PTMs are built on top of STMs or HTMs, hence relying on low-level information of memory accesses (reads/writes) for conflict detection, which leads to the problem of voluminous false aborts that are not caused by actual conflicts at the data structure semantic level. Second, PTMs rely on logging; log updates add performance-robbing cache line flushes and memory fences, and additional ordering constraints. Finally, many STMs rely on locks, hence most software PTMs are blocking. QSTM [5] is a non-blocking PTM, but is word-based and therefore synchronizes on read/write conflicts. In contrast, PETRA removes false aborts by utilizing high-level data structure semantics, adapts transactional descriptors instead of traditional logging, and provides non-blocking transactional behavior.

Unlike other PTMs, TimeStone [49] enables multi-version concurrency control at the semantic level, but relies on object locks to ensure thread-safety when making transient copies in DRAM to be manipulated by transactions. On a commit, the transient copy is added to the version chain. The advantage of TimeStone is that it reduces write amplification by writing only the latest transient copy on a writeback. The disadvantage of TimeStone is that it incurs additional overhead due to the management of a linked list of transient copies. PETRA's lock-free design avoids the overhead of managing copies, because updates are made directly to the object using atomic hardware primitives.

2.2 PETRA Baseline Selection

Choice of the Non-durable Baseline. Recall that the goals of PETRA design are high performance, high scalability, and non-blocking progress. Given that there are many choices of non-durable transactional data structure methodologies (Table 2), we must select one that is most relevant for PETRA design goals. While all the listed transactional data structure methodologies provide atomicity, isolation, and consistency, they differ in their conflict detection, transaction logging,

and progress guarantee. STM does not use data structure semantic conflict detection, hence is not suitable for PETRA. Transaction logging is an appealing feature for enabling durability, since persisting the log is sufficient for recovery. Lock-free Transactional Transformation (LFTT) [92] and STMs such as Word-based Software Transactional Memory (WSTM) [28] and Object-based Software Transactional Memory (OSTM) [28] provide transaction logging. Software Transactional Objects (STO) [41] logs a transaction's actions such as validation, installation, and rollback in a tracking set. Validation is required to ensure that all objects accessed during transaction execution have not been modified by other transactions prior to commit. Transactional Data Structure Libraries (TDSL) [76] does not explicitly log transactions, but their methodology could be extended to log the read/write set per transaction for recovery. Transactional Boosting [38] is a lock-based approach that maintains a log of inverse operations for rollbacks that is updated as the transaction executes. If the transaction does not finish executing, then only a subset of the transaction's operations are included in the log. LFTT and lock-free variants of STM are the only methodologies that provide non-blocking progress. From the table, we decided to choose LFTT to build PETRA on, due to its non-blocking progress, transactional logging, and data structure semantic conflict detection.

Overview of LFTT. LFTT enables developers to build non-blocking transactional data structures using existing non-blocking containers. Unlike generic STM-based approaches, LFTT leverages the semantic knowledge of the data structure to allow *commutative operations*, i.e., operations that have no dependencies on each other, to proceed concurrently in a non-blocking manner. This eliminates most false aborts due to access conflicts. LFTT also uses this knowledge to find conflicts in non-commutative operations through a node-based conflict detection mechanism. To synchronize transactions, LFTT uses a cooperative technique [4] allowing threads that share a node in their transactions to help complete each other's operations by including the required information in a descriptor. The helping scheme reduces not only false aborts, but also many true aborts, by allowing the thread that detects the conflict to execute the delayed transaction associated with the conflicting node. The progress guarantee in transactions based on the LFTT approach could degrade to obstruction-freedom if concurrent transactions access the same keys in reverse order and each of the concurrent transactions abort because they detected a cyclic dependency in the helping scheme at the same time, preventing progress. To guarantee lock-freedom in such cases a pre-processing technique can be used to ensure that transactions access the keys in the same order. Also, in case of an abort, LFTT uses a logical rollback technique that cancels the effects of an aborted transaction by inversely interpreting the logical status of the nodes. This method eliminates the overhead of physical rollback and wasting CPU cycles, while guaranteeing system-wide progress. Several variants of LFTT extend this methodology to support more linked data structures such as dictionary, and binary search trees with features such as dynamic transactions, wait-freedom, transactions among multiple data structures, and transactions on non-linked data structures such as dynamic arrays [50, 51, 93].

Applicability to Other Baselines. Our methodology can be generalized and applied to other transactional data structure baselines. PETRA provides durability using existing logging in the transactional data structure methodology as redo logs and persists these logs for recovery after a failure. If the transactional data structure baseline does not use sufficient logging for recovery (e.g., TDSL [76] and Transactional Boosting [38]), it can be extended to log the transaction operations in a redo log. The general approach for recovery in PETRA can then be applied to any transactional data structure baseline through a recovery procedure that uses the redo logs to fix inconsistencies following a crash event.

2.3 Usability of PETRA Compared to Other PTMs

PMDK [67], Mnemosyne [80], Atlas [7], and Romulus [14] enable a sequential implementation to be transformed into a persistent implementation by allocating and deallocating memory using custom methods for memory management and persistent variable declaration, and wrapping a transaction in a call to `write_transaction`. OneFile [70] transforms a sequential implementation to a persistent implementation with wait-free progress guarantees by wrapping all methods in calls to `updateTx` or `readTx` and replacing allocation and deallocation methods with OneFile's methods for memory management. JUSTDO [43], iDO [55], and NV-Heaps [11] provides C/C++ library calls for failure-atomic sections to enable concurrency and durability. PETRA requires more programmer involvement than these PTMs, because it is a methodology for transforming a base non-blocking data structure into a persistent non-blocking transactional data structure through maintenance of transaction descriptors. However, the additional programmer effort to maintain transaction descriptors leads to higher scalability, because the transaction descriptors contain sufficient information for full recovery and can be utilized as redo logs by the recovery procedure. Since only the transaction descriptors need to be persisted for recovery, the number cache line flushes and memory fences is kept low. Additionally, PETRA's semantic conflict detection and logical rollback enable PETRA to inherit the high performance capabilities of the base non-blocking data structure for thread-level synchronization.

3 PETRA METHODOLOGY

In this section, we present our PETRA methodology to design non-blocking durable transactional data structures. Building on top of LFTT, we discuss how we add durability to transactions while avoiding additional logging overheads.

3.1 Overview of the Methodology

The foundation of LFTT's transaction management is the maintenance of a transaction descriptor for each transaction. LFTT provides transactional synchronization for conflicts on the same node (non-commutative operations) by requiring a thread to check if the node to be operated on has an active status in the transaction descriptor prior to starting an operation for its own transaction. If the node has an active transaction, then LFTT requires the thread to help complete the active transaction prior to writing its transaction descriptor to the node and performing its own operation. This strategy enables transactional synchronization for conflicts on the same node. Read/write conflicts for operations on different nodes are handled using thread-level synchronization, because the order of operations on different nodes (commutative operations) does not matter. Since PETRA leverages transaction descriptors as redo logs, the only additional step required by PETRA is to explicitly persist the transaction descriptor. The challenges that PETRA faces when building on top of LFTT include (1) how to prevent a transaction from being visible until the transaction descriptor has been persisted, and (2) how to recover the data structure to a consistent state. To address the first challenge, PETRA adds a persistency status to the transaction descriptor that is not set to persisted until all fields of the transaction descriptor have been flushed, followed by an SFENCE to ensure that the flushed fields are persisted. PETRA uses the transaction status, persistency status, and operation type to logically interpret the existence of a node. To address the second challenge, PETRA introduces a novel recovery procedure that builds a key-descriptor map from the persisted transaction descriptors that is used to determine the presence/absence of a node in the recovered consistent state.

Figure 1 presents high-level steps to execute transactions using PETRA. Every thread starts the execution of a transaction by creating a descriptor object that includes information about the

Fig. 1. Methodology overview. Each transaction executes data structure operations specified by its descriptor. The EXECUTEDATASTRUCTUREOPERATION step is detailed in Figure 3. Based on the operations status, the transaction commits or aborts, then persists.

transaction status, data structure and transaction's operations (Section 3.2). A transaction begins after calling the EXECUTETRANSACTION function that performs initialization, such as preparing the helping scheme for executing the transaction's operations. Next, the EXECUTEOPS function executes the sequence of operations specified by the transaction descriptor (Section 3.4). Each operation interprets the logical status of a node in the data structure based on the transaction status and type of operation, which determines if the operation attempts to update the node or fails (Section 3.3). A transaction commits if all operations succeed, otherwise it aborts (Section 3.4). After setting the transaction status atomically, we perform the required actions to durably commit (or abort) the transaction. Finally, the transaction finishes with post-execution activities such as marking the removed nodes for deletion. We use the pointer marking approach to indicate logically deleted nodes [33]. If a node is bit-marked, the key associated with it does not logically exist in the data structure.

We apply PETRA's methodology on different linked data structures such as linked list and skiplist based sets, a multi-dimensional (md) list, and a hash map and present the evaluation results in Section 5. Without losing generality, we illustrate PETRA using a set abstract data type with three standard operations (INSERT, DELETE, and FIND). We list the constants and data type definitions in Algorithm 1. Each node of the data structure has an arbitrary key (*key*) that is assigned during node allocation. Additionally, each node of the data structure has a pointer, named *info*, to an object of type NODEINFO that keeps track of the latest executed transaction on this node. This information is provided by a reference to the transaction descriptor (*desc*) and the index of the most recent operation in the transaction that accessed the node (*opid*). A transaction descriptor is created at the beginning of a transaction, and persisted when a transaction ends. The *ops* array in the transaction descriptor is a static list of operations to be invoked by the transaction. During transaction execution, the operations in the *ops* array are executed in sequence, where a NODEINFO object is created for every operation in a transaction descriptor with a reference its own transaction descriptor *desc* and its operation ID *opid*. An operation succeeds once it replaces the old *info* stored in the node with its own NODEINFO object using CAS, described in detail in Section 3.3. The *desc* and *opid* fields of a node's *info* are updated on a successful operation.

Other important information in the transaction descriptor determines the status of the transaction and its durability status. The fields corresponding to durability are highlighted. The transaction status may be Active (being executed), Aborted (a data structure conflict has been detected necessitating an abort), and Committed (transaction execution is successful). The durability status may be Persisted (transaction already persisted), InProgress (transaction being persisted), and Maybe (persistence status unclear), which is the default value. A transaction becomes visible to other threads when the transaction status is Committed and persistence status is Persisted.

ALGORITHM 1: Type Definitions

1: enum TxStatus	15: int key
2: Active	16: struct Desc
3: Committed	17: int size
4: Aborted	18: int txid
5: enum PersStatus	19: TxStatus status
6: Maybe \\Default value	20: PersStatus pstatus
7: InProgress	21: Operation ops[ ]
8: Persisted	22: struct NodeInfo
9: enum OpType	23: Desc* desc
10: Insert	24: int opid
11: Delete	25: struct Node
12: Find	26: NodeInfo* info
13: struct Operation	27: int key
14: OpType type	28: ...

3.2 Durability via Transaction Descriptors

In LFTT, transaction descriptors are used for managing consistency and ensuring progress. When considering adding durability in PETRA, we deviate from the typical PTM approach of explicit logging. A PTM may rely on undo logging to keep track of old values of memory locations that will be written by the transaction. For correct recovery, the log itself must be made durable before the data structure is written. Alternatively, a PTM may rely on redo logging to record all memory writes of a transaction that need to be made durable at transaction commit. The logs are used after a crash to recover to a consistent state. Thus, traditional logging incurs two types of overheads: the additional instructions that manage and persist the log, and the additional ordering that requires the log to persist before data structure modification. Note that if crashes are infrequent, traditional logging is very expensive: Each transaction is slowed down even when the log is needed only when a crash occurs.

A key point of PETRA is our observation that the transaction descriptor object contains sufficient information of all data structure operations that a transaction must execute. Hence, we can re-purpose the transaction descriptor as a redo log to support durability. As such, our redo log has high-level information of data structure operations, rather than low-level information of memory accesses. Due to this high-level information, in PETRA, persisting a transaction is achieved by persisting its transaction descriptor, but the data structure itself does not need to be persisted, i.e., flushed out of the cache. We let the memory system naturally handle the durability of the data structure and resolve any inconsistencies during the recovery using the transaction descriptors. In contrast, PTMs must persist both the log and then the data structure in that order. Note that as computation progresses, changes made by a completed transaction will become durable, as cache blocks modified in the transaction will be gradually evicted from the cache. Thus, in contrast to PTMs, data structure changes are persisted *lazily*, as opposed to *eagerly* in PTMs [1].

Since only transaction descriptors are persisted, if a crash occurs, recovery needs to visit each past transaction to validate whether all operations specified in the persisted transaction descriptor have been reflected durably in the data structure. If they have, nothing else is needed. This is likely the case for most transactions because modified data blocks in the cache will get evicted over time. Otherwise, the transaction must be repeated, and here the descriptor serves as a redo log that specifies which operations need to be performed. Recovery procedure details are discussed in Section 3.5.

Fig. 2. Using transaction descriptors for helping, conflict detection, and ensuring durability.

In addition to the benefits discussed above, a transaction descriptor serves the following additional purposes: First, by keeping all the necessary information to complete a transaction, descriptors enable threads to help each other when a transaction is delayed. Delays in transactions can happen for reasons such as contention on shared resources and the operating system interrupts [92]. Second, it reflects the latest status of the transaction and makes it accessible to all threads that are executing transactions on common nodes. Finally, it enables *detectable execution* [29], the ability to determine after recovery whether a specific operation was executed.

Figure 2 illustrates how PETRA uses transaction descriptors for helping, detecting conflicts, and ensuring durability. Each thread executes the operations in the transaction descriptor by calling `EXECUTETRANSACTION`, as described in Section 3.1 and described in further detail in Section 3.4. The linkage between nodes is only updated if an insert is performed on a node that does not physically exist in the list, or if the node descriptor is marked for deletion. In this example, the set data structure consists of keys 1 and 3, which were inserted by Thread 1 through transaction $t1$. $t1$ was Committed and Persisted, as indicated by its *Status* and *PStatus*. Next, threads 2 and 3 execute their transactions $t2$ and $t3$ concurrently. Transaction $t2$ specifies two insert operations with keys 4 and 2, while $t3$ attempts to delete keys 3 and 4. Thread 3 performs its first operation by accessing the transaction descriptor stored in node 3. It reads the transaction descriptor *Status* as Committed and *PStatus* as Persisted and determines that no help is needed to complete the transaction. Thread 3 then performs the logical interpretation, described in detail in Section 3.3, by reading the operation *Type* as *Insert* and determining that node 3 exists in the set, which satisfies the precondition of the *Delete* operation. Thread 3 then applies CAS to update the *info* pointer on node 3 with a new `NODEINFO` prior to proceeding with its operation. If the CAS fails, thread 3 must re-access node 3's transaction descriptor, check for conflicts, help if needed, and do logical interpretation prior to re-trying the CAS.

Thread 3 then performs its second operation by accessing the transaction descriptor stored in node 4. Since $t2$'s transaction descriptor is stored in node 4 with an Active status, a conflict is detected with $t2$, because it has not finished its operations. Thread 3 must help transaction $t2$ to complete its remaining operations prior to continuing with its own operation. This helping

mechanism is possible because PETRA has the semantic knowledge of the data structure and divides the transaction into multiple steps, i.e., data structure operations. This division allows for keeping track of the transaction progress. Note that this helping mechanism can be prone to a live-lock problem when circular dependencies between helper threads exist. This problem is avoided through the use of a per-thread helping stack that contains the descriptors of the transactions that are being helped and checking for duplicates [92]. Once t_2 is Committed and Persisted, thread 3 performs logical interpretation by reading the operation *Type* as *Delete* and determining that node 4 does not exist in the set, which satisfies the precondition of the *Insert* operation. Thread 3 then applies CAS to update the *info* pointer on node 4 with a new *NodeInfo* prior to proceeding with its operation.

Suppose that a crash happens in the middle of the execution of t_2 and t_3 . Neither of these transactions completed before the crash and their effects are not visible to other threads. During recovery, the members of the set must reflect only the outcome of transactions that were completed and persisted before the crash. In the example, we only accept keys 1 and 3 as the members of the set. This state of the data structure is verified by the completed and persisted transaction descriptors, i.e., t_1 in this example. If any effect from t_2 and t_3 remained in the data structure, they would be canceled during the recovery, because those impacts were not visible before the crash. If any of the keys 1 and 3 do not exist in the data structure, then they will be inserted using the information provided by the transaction descriptor of t_1 .

3.3 Determining the Logical Status

We adapt the logical transaction management capabilities of LFTT [92] for building durable transactional data structures with ACID properties. We assign a logical status to the nodes to ensure atomicity and isolation. The status of each node is inferred based on the status of the latest transaction that accessed that node. This logical status allows us to hide the intermediate state of the shared data from concurrent transactions. Modifications are visible to other threads when the transaction is complete and can guarantee durability. Upon abort, a transaction can revoke the modifications made by the completed operations to guarantee atomicity. One approach to cancel the effects of the completed operations in transactional data structures is to invoke their inverse operations [38]. This method increases contention among threads without contributing to the overall throughput. Instead, in our logical mechanism, a transaction inverts its interpretation of the logical status of a node that was last accessed by an aborted transaction.

Algorithm 2 provides the details of our method to determine the node's status. The *IsNodePresent* function is called after traversing from the head to node n with the key of interest *key*, where the traversal stops at the predecessor if the key of interest does not exist in the data structure. On line 2.2, the physical presence of a node with the specified key is verified. Determining the logical status of a key is done by the function *IsKeyPresent*. This function returns a Boolean value that indicates the logical presence of the key in the abstract state of the data structure. This function uses the information from the last transaction that accessed the node and the descriptor object of the current transaction.

We know that the state of a node is not altered if the last transaction that accessed it was a *FIND* operation. We report this node as present in this case (line 2.6). On line 2.7, we read the status of the last transaction that accessed the node. If the last transaction is still active (case on line 2.10) and the last transaction is equivalent to the current transaction (checked on line 2.11), then the node is logically present if the operation is an *INSERT* (line 2.12). Otherwise, if the last transaction that accessed the node is not equivalent to the current transaction, then the node is logically present if the operation is a *DELETE* (line 2.14).

ALGORITHM 2: Logical Status

```

1: function ISNODEPRESENT(Node* n, int key)
2:   return n.key == key
3: function ISKEYPRESENT(NodeInfo* info, Desc*desc)
4:   OpType op  $\leftarrow$  info.desc.ops[info.opid]
5:   if op == Find then
6:     return True
7:   TxStatus status  $\leftarrow$  info.desc.status
8:   PersStatus pstatus  $\leftarrow$  info.desc.pstatus
9:   switch (status)
10:    case: Active
11:      if info.desc == desc then
12:        return op == Insert
13:      else
14:        return op == Delete
15:    case: Committed
16:      return op == Insert and pstatus == Persisted
17:    case: Aborted
18:      return op == Delete and pstatus == Persisted

19: function UPDATEINFO(Node* n, NodeInfo* info, bool, wantkey)
20:   NodeInfo* oldinfo  $\leftarrow$  n.info
21:   if ISMARKED(oldinfo) then
22:     Do_DELETE(n)
23:     return retry
24:   if oldinfo.desc  $\neq$  info.desc then
25:     EXECUTEOPS(oldinfo.desc, oldinfo.opid + 1)
26:   else if oldinfo.opid  $\geq$  info.opid then
27:     return success
28:   bool haskey  $\leftarrow$  ISKEYPRESENT(oldinfo)
29:   if (!haskey and wantkey) or (haskey and !wantkey) then
30:     return fail
31:   if info.desc.status  $\neq$  Active then
32:     return fail
33:   if CAS(&n.info, oldinfo, info) then
34:     return success
35:   else
36:     return retry

```

If the last transaction has finished its execution on the node, then we determine whether its effect is observable by the current transaction by examining two cases. In the first case (case for Committed on line 2.15), the key logically exists only if the last transaction has executed an INSERT operation, committed successfully, and made its descriptor durable (evaluated on line 2.16). In the second case (case for Aborted on line 2.17), we consider a DELETE operation. By definition, a successful DELETE operation must remove the key from the set. To report a key as present, if the last transaction has executed a DELETE operation, then it must have aborted and persisted its descriptor. If any of the above conditions are not met (evaluated on line 2.18), then the key does not logically exist from the point of view of the current transaction. It is possible for an aborted transaction to be persisted and correctly rolled back. For example, suppose that transaction t_1

Fig. 3. The EXECUTE DATA STRUCTURE OPERATION step of the methodology displayed in Figure 1.

in Figure 2 aborts and is persisted. When t_3 accesses node 3, it reads transaction t_1 's *status* as Aborted, its *pstatus* as Persisted, and operation type as INSERT. Based on line 2.18, t_3 interprets that node 3 does not exist in the data structure.

Function `IsKeyPresent` is called by function `UPDATEINFO`, which starts at line 2.19. An operation in the underlying data structure needs to update the *info* pointer of its active node (read on line 2.20) before making changes. This update is necessary, as the *info* pointer of the node is used to determine its logical status. The active operation calls `UPDATEINFO` to perform this modification (Figure 3). If the target node is logically marked for deletion (checked on line 2.21), then we complete the operation by invoking the base data structure delete method (line 2.22) and inform the caller to retry the current operation line 2.23. Before updating the node info, the current thread first checks if the transaction descriptor for the *info* pointer of the active node is for a different transaction (line 2.24) and helps complete this other transaction if needed (line 2.25 (`EXECUTEOPS` is illustrated in Figure 1)). Otherwise, if a helper thread already executed the current operation (checked on line 2.26), we can ignore this operation by returning success (line 2.27) and continue the rest of the transaction. Next, we interpret the logical presence of a key (line 2.28) and check if the logical presence/absence is matched with the need of the operation. For example, a `DELETE` operation expects that the key to be present in the list (*wantkey* is true) and an `INSERT` operation requires that the key to not be a part of the list (*wantkey* is false). `UPDATEINFO` evaluates these conditions on line 2.29 and returns *fail* if the conditions are not met. After verifying that the current transaction is still active (line 2.31), *n.info* is updated by using a CAS (line 2.33) and the data structure operation can proceed.

3.4 Executing Durable Transactions

As illustrated by Figure 1, each transaction executes the data structure operations specified by the descriptor object. Figure 3 illustrates how this step is executed by a linked list that is transformed using PETRA. Each data structure operation features a CAS-based while loop, which is the typical approach for implementing non-blocking data structures. Each thread attempts to apply updates on the shared object atomically, and if it fails, it retries the operation execution if needed. Functions that start with a prefix `Do_` represent the methods typically implemented by a linked list-based set, which is the underlying data structure in our example. The data structure should be refactored to support these methods, if it does not offer those already. For example, `DO_LOCATEPRED` returns the required nodes and variables for handling the linkage in the structure, e.g., the predecessor node. `DO_OPERATION` could be any of `DO_INSERT` and `DO_DELETE` functions that add and remove the necessary links to perform their operations, respectively.

ALGORITHM 3: Persistence of Transactions

```

1: function PERSISTTRANSACTION(Desc* desc)
2:   if (CAS(&desc.pstatus, Maybe, InProgress)) or (desc.pstatus == InProgress) then
3:     for op ∈ desc.ops do
4:       FLUSH(&op)
5:     desc.txid ← GETNEXTTXID()
6:     FLUSH(desc)
7:     SFENCE()
8:     desc.pstatus ← Persisted

```

Another modification to the base data structure is the invocation of the UPDATEINFO function. If the node exists in the structure of the linked list, then we call the UPDATEINFO function (line 2.19), which makes a recursive call to EXECUTEOPS to finish a pending transaction before making changes. This step is necessary to interpret the logical status of a node and update it to prevent unsafe access by concurrent transactions. Based on the results of the call to this function, we determine whether another attempt is needed to perform the operation or return the result. If the node does not exist, then no call to UPDATEINFO is needed, and the operation can proceed.

Algorithm 3 presents the PERSISTTRANSACTION function that is used to ensure the durability of transactions. Since we only need to ensure the durability of the transaction descriptor object, the descriptor is all that PERSISTTRANSACTION needs as the input. A thread in this function first declares its intent to persist the transaction descriptor by performing a CAS on line 3.2. This declaration prevents possible helper threads from re-persisting the descriptor by executing expensive flush and fence operations. If a thread commits or aborts its transaction, but gets delayed in the middle of the persistence, then another thread can help persist the transaction. The need for help can be inferred based on the delayed transaction's persistence status as being *InProgress*.

If the current thread successfully declares its intent to persist the descriptor, then it traverses over all the operations and flushes the information related to each operation on line 3.4. On line 3.5, the transaction ID is assigned to the descriptor. The recovery procedure (Section 3.5) uses the transaction ID to determine the order of transactions executed on each key in a data structure. Next, the descriptor object is flushed to store the remaining information about the transaction on line 3.6. To guarantee that the persisted transaction is visible, we use the SFENCE instruction on line 3.7. Finally, we set the persistency status of the transaction to PERSISTED to notify other transactions. After the execution of line 3.8, the effect of the current transaction is globally visible. Note that we do not need to ensure the persistence of the PSTATUS itself, as its value is implied by a transaction that is persisted.

Function PERSISTTRANSACTION provides durability at low cost by reducing the number of flushes and fences. In total, the number of flushes corresponds to the size of the transaction plus one more flush to store the transaction descriptor. Finally, for each transaction, we explicitly execute one fence instruction regardless of its size.

3.5 Recovery Management

3.5.1 System Support and Memory Addressing. When a system crashes, objects in persistent memory need to be found and mapped back into the process address space. This requires system support, such as memory-mapped files [11, 13, 19, 42, 79, 80, 84], persistent memory-aware file systems [13, 19, 84, 85], or system-managed objects in memory [87]. Once found, the region may be remapped to the process address space at a different virtual memory location, hence relocatability needs to be supported [48, 60], such as using new relocatable pointer formats [8, 82] and persistent

Fig. 4. Recovery steps.

page table [87]. Addressing these issues is orthogonal to PETRA and beyond the scope of this article. Note that there is nothing that fundamentally prevents these ideas from being applied to PETRA.

3.5.2 Recovery Procedure. Recall that PETRA explicitly persists transaction descriptors at the end of each transaction. The recovery procedure rebuilds the underlying data structure, verifies its consistency using the transaction descriptors, and fixes possible inconsistencies that might have occurred as a result of a crash. Figure 4 presents the steps to recover a data structure (linked list-based set) after a crash. Upon recovery, the initial set is built by loading the head node. Any node reachable from the head node is a part of this initial list. Next, the transaction descriptors that were persisted by each thread are read to figure out transaction execution records ①. Based on the transaction descriptors, we build the key-descriptor map (*KDMap*) ②. This involves visiting each Committed/Persisted transaction to find the transactions that accessed each key in the data structure. If we have more than one transaction that is executed on a key, then we use *TXID* of the descriptor to identify the transaction that happened last. Note that *TXID* is assigned by a global monotonically increasing generator before persisting the transaction descriptor (line 3.5). No transaction is visible to other threads unless *TXID* of its descriptor is assigned and persisted. The ordering mechanism does not need to enforce a global ordering on all transactions. It is sufficient to know the order of transactions executed on each key, which can be achieved using *TXID* generated by tools such as *FETCH-AND-ADD* operations, time-stamps, or other similar techniques.

Next, we traverse the loaded set ③ and determine the logical status of a key based on the last valid transaction that is executed on the node with that key. *KDMap* provides the descriptor for this transaction. If the descriptor pointer of the node is not persisted before crash, it does not match the descriptor found by *KDMap*. In this case, we remove the node and execute the corresponding operation based on the data provided by the valid descriptor and we end up in a valid state for the node. If the descriptor found by *KDMap* matches the node's descriptor, then there are two cases to consider: (1) node contents are valid, i.e., the value is correct, and (2) node contents are invalid. In the second case, to restore the consistent state, we remove the node and execute the descriptor's operation. To fix other possible inconsistencies, we insert the items that, according to the *KDMap*, should be present in the data structure but are not ④. After this step, the data structure is restored to a consistent state and the recovery procedure is complete ⑤. If a crash happens during recovery, then this procedure should start from the last known valid point in the transaction records.

To guarantee consistency, we do not need to use any of the transaction descriptors that are not persisted, even for those transactions that are completed. As we describe in Section 4, we use *durable linearizability* [44], which is the strictest correctness property to ensure a consistent state of the data structures. Durable linearizability requires that the state of the data structure after a crash includes a consistent subhistory of the operations that actually occurred and were globally visible before the crash. As we discuss in Section 3.3, the effect of a completed transaction is visible to other threads only when its transaction descriptor becomes persistent.

4 CORRECTNESS

In this section, we show that PETRA satisfies durable linearizability.

4.1 Correctness Definitions

Definitions are provided to facilitate reasoning about durable linearizability. An execution of a concurrent system is modeled by a *history*, a finite sequence of method *invocation* and *response* events [40]. A response *matches* an invocation if they are called by the same thread on the same object. A *method call* in a history H is a pair consisting of an invocation and next matching response in H , also referred to as an *operation*. An invocation is *pending* in H if no matching response follows the invocation. An *extension* of H is a history constructed by appending responses to zero or more pending invocations of H . The notation $complete(H)$ denotes the subsequence of H consisting of all matching invocations and responses. A *sequential specification* for an object is a set of sequential histories for the object. A sequential history H is *legal* if each object subhistory is legal for that object.

Definition 1. A history H is *linearizable* if it has an extension H' and there is a legal sequential history S such that (1) $complete(H)$ is equivalent to S , and (2) if m_0 precedes method call m_1 in H , then the same is true in S [40].

Legal sequential history S in Definition 1 is referred to as a *linearization* of H .

Definition 2. Given an execution E , an operation O is *durable* at step t of the (extended) execution E if the following holds: For any legal execution E' , which equals E in the first t steps, if the execution of the recovery of O completes in E' , then for any linearization of E' , O is linearized.

An operation is considered durable if there is sufficient information in NVM such that the recovery procedure causes this operation to be linearized.

Definition 3. Given an extended execution E , the *durability point* of operation O is the first point t in the execution when the operation O becomes durable.

Definition 4. Given an execution E , the durability points of the operations in the execution E imply an order on the operations, called *durability order*.

Definition 5. A linearizable object is *durably linearizable* if for all executions E of the object, (1) the durability point of each operation is between its invocation and response, and (2) there exists a linearization of E whose order of operations is the same as the durability order of operations in E [29].

Definition 6. A history H is *strictly serializable* if the subsequence of H consisting of all events of committed transactions is equivalent to a legal sequential history S in which these transactions execute sequentially in the order they commit [66].

Legal sequential history S in Definition 6 is referred to as a *strict serialization* of H .

We extend the notion of durable linearizability to transactions by considering an “operation” in Definition 5 to be a transaction and a “linearization” in Definition 5 to be a strict serialization.

4.2 Durable Linearizability

To prove that PETRA is durably linearizable, it must be shown that for all multithreaded executions E , (1) the durability point of each transaction is between its invocation and response, and (2) there exists a strict serialization of E whose order of transactions is the same as the durability order of transactions in E .

THEOREM 1. *PETRA is durably linearizable.*

PROOF. First, it is shown that the durability point of a transaction occurs between its invocation and response. When `EXECUTETRANSACTION` is invoked for transaction T_1 , the operations listed in T_1 's transaction descriptor are executed according to operation order. If T_1 detects a conflict with transaction T_2 , then T_1 helps complete T_2 prior to proceeding with its own operations. If transaction T_3 detects a conflict with T_1 , then T_3 helps complete T_1 . Once T_1 's operations have been completed, a CAS is attempted to either commit or abort T_1 . If the CAS fails, then some other thread must have either committed or aborted T_1 . After T_1 has either committed or aborted, it is persisted by invoking `PERSISTTRANSACTION`. Since T_1 is guaranteed to be durable once it returns, the durability point for T_1 occurs between its invocation and response.

Next, it is shown that there exists a strict serialization of E whose order of transactions is the same as the durability order of transactions in E . The `ISKEYPRESENT` function prevents transaction T_1 's operations from being visible to other transactions until T_1 is persisted due to the return value on line 2.16 and line 2.18. Since the effects of T_1 's operations are visible to other transactions at the instant it is persisted and PETRA is strictly serializable by the LFTT methodology [92], there exists a strict serialization of E whose order of transactions is the same as the durability order of transactions in E .

If a crash occurs, the recovery procedure is invoked by the main thread to restore the state of the PETRA-based data structure. It now must be shown that the restored state reflects a strict serialization of E whose order of transactions is the same as the durability order of the operations in E . As described in Section 3.5, *KDMap* is a map where the key is the node key and the value is the most recent committed/persisted transaction that accesses the node key in the data structure. We now show that a valid state of the data structure can be recovered from *KDMap*. Since set operations that access different nodes are commutative, the order of the set operations relative to different keys does not affect the outcome of the node state. Let $T_1, T_2, \dots, T_j, \dots, T_{n-1}, T_n$ be the history of committed/persisted transactions in persist order as described in Section 3.5. The `ISKEYPRESENT` function only enables committed transactions that have persisted to be visible to other transactions, so the commit order is equivalent to the persist order. Let T_j be the last committed/persisted transaction to access some node k . Let op_j be the last operation in T_j to access node k . Since T_j commits, this implies that op_j succeeds. Let S be the set of nodes that exist in the list. If op_j is `FIND` or `INSERT`, then $node\ k \in S$. If op_j is `DELETE`, then $node\ k \notin S$. The same reasoning applies for all other nodes in the data structure. Therefore, the state of the data structure consistent with a strict serialization of E whose order of transactions is the same as the durability order of the operations in E can be recovered from *KDMap*. $\square$

5 EXPERIMENTAL EVALUATION

We evaluate our approach and compare it against the state-of-the-art PTM platforms using various benchmarks.

5.1 Experimental Setup

Machine Testbed. We conduct our tests on a machine equipped with Intel Optane DC Persistent Memory (DCPM). The machine has Intel's most recent second-generation Xeon Scalable

processors (code-named Cascade Lake) with 48 cores (2 sockets), supporting 96 threads. The main memory consists of Optane DCPM with 6 TB total capacity, plus 768 GB DRAM. In all experiments, we place persistent data structures in the DCPM; DRAM is used to store everything else (e.g., code). The machine is configured to run in 100% App Direct Mode [45], which allows applications byte-addressable access to the persistent memory. The OS is Ubuntu 18.04 LTS. The application and micro-benchmarks were compiled using gcc 7.4 with the -O3 optimization flag and C++14 standard flags.

Micro-benchmarks. We conduct our evaluations on four transactional non-blocking data structures: three different sets based on linked list, skiplist and multi-dimensional list (mdlist), and hash map. In the linked list-based set experiments, each thread performs 100,000 transactions and the key range is set to 10,000. In the experiments for other data structures, each thread performs 1,000,000 transactions and the key range is set to 1,000,000.

In micro-benchmarks, we compare the overhead and scalability of PETRA against three state-of-the-art PTMs: OneFile (lock-free version) [70], Romulus (LR version) [14], and PMDK (libpmem-obj++ protected using read-write locks) [67]. We also ran experiments using Mnemosyne [80], but we did not include the results, because it exhibits the lowest throughput and does not support more than 31 threads [14]. To evaluate the overhead of transactional synchronization, we compare against non-transactional durable sets [94] and LFTT [92]. Romulus was reported to outperform PMDK and Mnemosyne and OneFile shows a slightly better throughput compared to Romulus in some cases in the literature [14, 70]. We run our micro-benchmark experiments to evaluate the overall performance using various workloads based on the ratio of read and write operations. This method of evaluation, commonly used in the literature [14, 29, 33, 70, 92], consists of a loop that randomly chooses a transaction to execute with a mixture of read and write operations according to a uniform distribution, and operation ratio and workload type.

5.2 Micro-benchmark Evaluation Results

Figure 5 displays the throughput for the transactional linked list (a,b), map (c,d), skiplist (e,f), and mdlist (g,h) implementations using different workloads (note the logarithmic scales). Throughput (y-axes) reflects the number of completed operations per second. In all plots, our scheme is denoted by PETRA, OneFile by OFLF, Romulus by ROM, PMDK by PMDK, LFTT by LFTT, and non-transactional durable sets by NTDS_{ET}. The transaction size (number of operations in a transaction) varies from 1 to 4, shown in Figure 5. The transaction size appears as a suffix to each set (e.g., PETRA-4 means transaction size 4 for PETRA). Each thread allocates memory from a pre-allocated pool. The number of threads varies from 1 to 96.

Figure 5(a) displays results for a write-dominated workload for the linked list-based set. For a single thread, all approaches perform close to each other. As the thread count increases, PETRA's throughput increases substantially, while the throughput of other PTMs stagnates or declines. The only other approaches that scale well as the thread count increases are LFTT (PETRA's baseline) and NTDS_{ET}. The structure of the SET abstract data type makes it a suitable choice to exploit the parallelism of a multi-threaded system by distributing contention across nodes. PETRA exhibits high throughput and scalability that can be attributed to its non-blocking approach that keeps abort rates low. The high abort rates due to false aborts in the alternative approaches keep them from increasing their throughput. At 48 and 96 threads, PETRA outperforms the next performing technique, OneFile, by more than one order of magnitude.

We show the results from read-dominated workloads in Figure 5(b). The results for these workloads follow a similar trend as the write-dominated intensive workload, but OneFile and Romulus exhibit better performance compared to the read-dominated workloads. Romulus uses lighter

Fig. 5. Throughput for transactional data structures for transactions of size 1 and 4. Operation ratio for write-dominated workload in lists: 40% Insert, 40% Delete, 20% Find and maps: 40% Insert, 30% Delete, 10% Update, 20% Find. Operation ratio for read-dominated workload in lists: 10% Insert, 10% Delete, 80% Find and maps: 10% Insert, 10% Delete, 5% Update, 75% Find. Key range for linked list: 10K, other data structures: 1M.

synchronization mechanisms to optimize read-only operations that enable reader scalability, with throughput slightly increasing with thread counts. PETRA uses transaction descriptors for all operations and updates the references even for read operations such as FIND, hence its scalability remains the same as in write-dominated workload. As a result, PETRA's throughput advantage over OneFile and Romulus decreases, but it is still larger than one order of magnitude with 96 threads.

For hash map experiments, in part (c) with the write-dominated case, PETRA outperforms all PTM alternative approaches, again, thanks to not suffering from many transaction aborts due to helping and not having false aborts. In part (d) with mostly read operations, similar to the linked list experiments, the throughput of other approaches is improved. PETRA performs not as well for lower thread counts, but it scales better at higher thread counts and outperforms alternative transactional implementations. NTDSet outperforms PETRA because it does not have the overhead of transaction management, and LFTT outperforms PETRA because it does not have the persistence overhead. The instruction breakdown of number of flush, fence, and CAS per transaction update is shown in Table 3. The number of instructions expressed as a function of number of modified words N_w for PMDK, Romulus, and OneFile are provided from Ramalhetete et al. [70]. PETRA's instruction breakdown is provided as a function of the number of operations Op and number of modified words per operation $N_{w/Op}$, where $\lceil \frac{N_{w/Op}}{8} \rceil$ determines flushes per operation based on the cache line size. One of PETRA's performance benefits over other PTMs is that it requires the fewest flushes, since only the operations in the transaction descriptor and the transaction descriptor itself must be explicitly persisted.

Transactional skiplist and mdlist display a similar trend to the transactional hash map. The base data structures in both cases [27, 90] have logarithmic search times and execute transactions more efficiently compared to the linked list-based set. The overhead of PETRA when compared

Fig. 6. Performance comparison of PETRA with general-purpose PTMs in TATP benchmark.

Table 3. Instruction Breakdown on Update Transactions, as a Function of Number of Modified Words N_w (PMDK, Romulus, OneFile from Ramalhete et al. [70]) or Number of Operations Op

Transaction Methodology	flush	fence	CAS
PETRA	$1 + Op \cdot \lceil \frac{N_w/Op}{8} \rceil$	1	$2 + Op$
PMDK	$2.25 \cdot N_w$	$2 + 2 \cdot N_w$	1
RomulusLog	$3 + 2 \cdot N_w$	4 or less	1
OneFile (lock-free)	$1 + 1.25 \cdot N_w$	0	$2 + N_w$
OneFile (wait-free)	$2 + 1.25 \cdot N_w$	0	$3 + N_w$

to NTDSets and LFTT is larger for maps and skiplists than linked lists, because the logarithmic search time enables more operations to be performed. Although for these types of data structures, in read-dominated cases with lower thread counts all approaches exhibit close throughput, overall PETRA performs three times better than the next best PTM.

5.3 TATP Benchmark

We evaluate our transactional map in the TATP benchmark [77] by testing *UpdateLocation* transactions and compare its performance with generic PTMs proposed in the recent literature [89]. Figure 6 presents these results. Throughput reflects the number of millions of transactions executed per second. While other approaches exhibit poor scalability, TLRW [18] and Orec [17, 25, 75] perform as good as PETRA for low thread counts but fail to scale as we increase the number of threads (TLRW crashed when running with 96 threads). Orec uses ownership records with variants of undo/redo logging, the locking mechanisms, and lazy/eager approaches. TLRW is an eager algorithm with readers/writer locks that does not require quiescence to ensure safety during commit. This feature and other optimizations, such as fence pipelining, contribute to the better scalability. Similar to write-dominated workloads in Figure 5, PETRA demonstrates its scalability and shows over nine times higher throughput compared to the best PTM at 96 threads. This advantage happens as a result of leveraging the data structure semantic knowledge to manage both concurrency and durability efficiently, which also reduces the number of required flushes and fences.

5.4 Database Benchmark

We demonstrate the application of our methodology in a persistent key-value store by using PETRA's transactional map. We integrated our transactional hash map with *pmemkv* [68], a key/value datastore for persistent memory. We evaluate and compare it against an implementation based on Intel TBB concurrent hash map. To add transactional capabilities to the implementation based on the TBB map, we use *abstract locking* with undo logs, analogous to transactional boosting [38]. We use a benchmark named *pmemkv_bench* from *pmemkv-tools* [69], which provides a collection of standard read and write benchmarks. The benchmarks are based on the *db_bench* utility, which is integrated with popular databases such as LevelDB [30] and RocksDB [23].

In all benchmarks, we utilize integer keys and values and each thread executes 1M transactions and each transaction performs four operations. In the *fillseq* benchmark, each thread executes insert-only transactions using sequential keys. The *fillrandom* benchmark performs the

Fig. 7. Database benchmark. Number of threads in all plots (1,2,4,8,16,48,96).

same but with random keys per thread. The *overwrite* benchmark performs the insertions similar to *fillrandom*, but works on a database that is filled with the key-value pairs. The *readseq*, *readrandom*, *deleteseq*, and *deleterandom* benchmarks are similar to their *fill* versions, but perform read and delete transactions. The *readmissing* benchmark reads N missing values in random order. In the *readrandomwriterandom* benchmark, all threads carry out transactions with both types of operations randomly. In this benchmark, 90% of operations are read and 10% of them are write operations.

Figure 7 presents the results, with the y-axis showing the time (in microseconds) to execute an operation, while the x-axis shows two sets of bars: *cmap* represents Intel TBB's concurrent hash map, and our approach is denoted with *PETRA*. Each set contains 7 bars corresponding to the following thread counts: 1, 2, 4, 8, 16, 48, 96.

For write-only workloads, (a–e), *PETRA* allows faster database transaction execution in all cases. For read-only workloads, (f–h), *PETRA* outperforms *cmap* in low thread counts except when the system uses threads on both CPU sockets. *PETRA*'s engine outperforms Intel TBB's concurrent map engine in the mixed workload, (i), that each transaction can execute both read and write operations.

5.5 Recovery Evaluation

The performance of the recovery procedure to restore the linked list to a consistent state for *PETRA*, *Romulus* (ROM), and *OneFile* (OFLF) is shown in Figure 8. The transaction size is fixed at 4 and the workload information is described in Figure 5. *OneFile* has the fastest recovery time, because *OneFile* transactions guarantee that all stores are made durable on a commit. The only transaction that may not be fully persisted is the most recent committed transaction, so the recovery procedure only requires the threads to identify the write-set of the last committed transaction and persist the remaining stores if the request is still open. *Romulus* recovery time takes longer

Fig. 8. Recovery time for linked list.

than OneFile, because the recovery procedure copies the touched cache lines of back (the previous consistent state of the main region) to main if the thread was mutating main at the time of the crash to revert incomplete modifications, or copies the touched cache lines of main to back if the thread crashes while copying main to back. PETRA recovery time is the longest, because it is not optimized for a system with frequent crashes. Petra must iterate through the threads and build the key-descriptor map based on the thread's transaction descriptors, then traverse the linked list starting from the head and use the key-descriptor map to either remove invalid keys or insert missing keys. Although PETRA has the longest recovery times, it has the best performance and scalability, since it reduces the number of explicit cache line flushes and memory fences by only persisting the descriptors.

6 LIMITATIONS AND FUTURE WORK

PETRA brings the benefit of high performance at the cost of *write amplification* [63], i.e., the number of additional bytes written to persistent memory for every byte of user data ($2 - 3\times$). Most of the amplification is due to LFTT. To achieve persistence, PETRA itself only adds 12%–35% space overheads on top of LFTT, depending on the transaction size. This is a reasonable tradeoff, since persistent memory capacity is much higher than DRAM. In this work, we assumed that small objects are used in the transactions and operation data fit in the cache-line. To guarantee failure-atomicity for transactions with larger objects, we need to ensure the durability of the large object before persisting the transaction, and to persist data that does not fit in a single cache-line, more flushes are needed. We also assumed that a crash is rare and our methodology optimizes failure-free execution at the expense of possibly slower recovery. In future work, we plan to employ a *periodic checkpointing* mechanism to put an upper bound on the number of past persisted transactions to validate. This mechanism can also improve the persistent memory space overhead. The periodic checkpointing would reduce the time for the recovery procedure, because the number of transaction descriptors to consider when building the key-descriptor map would be bounded. However, the performance and scalability of PETRA would also be reduced, because the persisted data structure would need to guarantee consistency up to the checkpoint. We also plan to apply our approach to the extended versions of LFTT to support features such as wait-freedom, dynamic transactions, and more data structures, and apply our techniques in implementing an in-memory database.

7 CONCLUSION

In this article, we presented PETRA, a new technique to create persistent non-blocking transactional data structures with ACID properties. We leveraged descriptor objects to implement an efficient scheme that manages concurrency and durability. PETRA achieves high performance by keeping the number of cache line flushes and memory fences low, persisting a transaction by only

persisting its descriptor, and by persisting data structures lazily without using flushes and fences. It uses the transaction descriptors as redo logs. It achieves high scalability by eliminating false aborts (by utilizing high-level knowledge of data structure semantics) and reducing true aborts (through helping). PETRA also preserves LFTT's non-blocking progress guarantee. Our performance evaluation demonstrates that our approach, on average, exhibits 17× and 3× higher throughput compared to the state-of-the-art PTM, for mixed workloads that utilize set and other data structures, respectively.

REFERENCES

- [1] M. Alshboul, J. Tuck, and Y. Solihin. 2018. Lazy persistency: A high-performing and write-efficient software persistency technique. In *ISCA*. IEEE, Washington, DC, 439–451. DOI: <https://doi.org/10.1109/ISCA.2018.00044>
- [2] Hillel Avni and Trevor Brown. 2016. Persistent hybrid transactional memory for databases. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 409–420. DOI: <https://doi.org/10.14778/3025111.3025122>
- [3] Hillel Avni, Eliezer Levy, and Avi Mendelson. 2015. Hardware transactions in nonvolatile memory. In *Distributed Computing*, Yoram Moses (Ed.). Springer Berlin, 617–630.
- [4] Greg Barnes. 1993. A method for implementing lock-free shared-data structures. In *SPAA*. ACM, New York, NY, 261–270. DOI: <https://doi.org/10.1145/165231.165265>
- [5] H. A. Beadle, W. Cai, H. Wen, and M. L. Scott. 2020. Nonblocking persistent software transactional memory. In *PPoPP*. ACM, New York, NY. DOI: <https://doi.org/10.1145/3332466.3374506>
- [6] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. 2008. Software transactional memory: Why is it only a research toy? *Queue* 6, 5 (2008). DOI: <https://doi.org/10.1145/1454456.1454466>
- [7] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. In *OOPSLA*. ACM. DOI: <https://doi.org/10.1145/2660193.2660224>
- [8] G. Chen, L. Zhang, R. Budhiraja, X. Shen, and Y. Wu. 2017. Efficient support of position independence on non-volatile memory. In *MICRO*. ACM, 191–203. DOI: <https://doi.org/10.1145/3123939.3124543>
- [9] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797. DOI: <https://doi.org/10.14778/2752939.2752947>
- [10] P. Chuong, F. Ellen, and V. Ramachandran. 2010. A universal construction for wait-free transaction friendly data structures. In *SPAA*. ACM, New York, NY. DOI: <https://doi.org/10.1145/1810479.1810538>
- [11] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*. ACM.
- [12] N. Cohen, D. T. Aksun, H. Avni, and J. R. Larus. 2019. Fine-grain checkpointing with in-cache-line logging. In *ASPLOS*. ACM. DOI: <https://doi.org/10.1145/3297858.3304046>
- [13] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *SOSP*.
- [14] A. Correia, P. Felber, and P. Ramalhete. 2018. Romulus: Efficient algorithms for persistent transactional memory. In *SPAA*. ACM. DOI: <https://doi.org/10.1145/3210377.3210392>
- [15] T. David, A. Dragojević, R. Guerraoui, and I. Zabolotchi. 2018. Log-free concurrent data structures. In *USENIX ATC*. USENIX. Retrieved from <https://www.usenix.org/conference/atc18/presentation/david>.
- [16] D. Dechev, P. Pirkelbauer, and B. Stroustrup. 2006. Lock-free dynamically resizable arrays. In *Principles of Distributed Systems*. Springer Berlin.
- [17] D. Dice, O. Shalev, and N. Shavit. 2006. Transactional locking II. In *Distributed Computing*. Springer, Berlin.
- [18] D. Dice and N. Shavit. 2010. TLRW: Return of the read-write lock. In *SPAA*. ACM. DOI: <https://doi.org/10.1145/1810479.1810531>
- [19] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. 2014. System software for persistent memory. In *CS*.
- [20] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert. 2014. The amortized complexity of non-blocking binary search trees. In *PODC*. ACM. DOI: <https://doi.org/10.1145/2611462.2611486>
- [21] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin. 2017. Efficient checkpointing of loop-based codes for non-volatile main memory. *IEEE*. DOI: <https://doi.org/10.1109/PACT.2017.58>
- [22] David Eppstein and Elham Havvaei. 2019. Parameterized leaf power recognition via embedding into graph products. In *IPEC*, Vol. 115. LIPIcs, Dagstuhl, Germany, 16:1–16:14. DOI: <https://doi.org/10.4230/LIPIcs.IPEC.2018.16>
- [23] Facebook. 2020. A Persistent Key-Value Store for Flash and RAM Storage. Retrieved from <https://github.com/facebook/rocksdb>.

- [24] P. Fatourou and N. D. Kallimanis. 2011. A highly efficient wait-free universal construction. In *SPAA*. ACM. DOI : <https://doi.org/10.1145/1989493.1989549>
- [25] P. Felber, C. Fetzer, and T. Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*. ACM. DOI : <https://doi.org/10.1145/1345206.1345241>
- [26] S. Feldman, C. Valera-Leon, and D. Dechev. 2016. An efficient wait-free vector. *IEEE Trans. Parallel Distrib. Syst.* 27, 3 (Mar. 2016), 654–667. DOI : <https://doi.org/10.1109/TPDS.2015.2417887>
- [27] Keir Fraser. 2004. *Practical Lock-freedom*. Technical Report UCAM-CL-TR-579. University of Cambridge, Computer Laboratory. Retrieved from <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf>.
- [28] K. Fraser and T. Harris. 2007. Concurrent programming without locks. *ACM Trans. Comput. Syst.* 25, 2 (2007), 5.
- [29] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A persistent lock-free queue for non-volatile memory. *SIGPLAN Not.* 53, 1 (Feb. 2018), 28–40. DOI : <https://doi.org/10.1145/3200691.3178490>
- [30] Google. 2020. LevelDB: A Fast Key-value Storage Library. Retrieved from <https://github.com/google/leveldb>.
- [31] Rajeev Gupta, Himanshu Gupta, and Mukesh Mohania. 2012. Cloud computing and big data analytics: What is new from databases perspective? In *BDA*. Springer, 42–61.
- [32] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM Comput. Surv.* 15, 4 (Dec. 1983), 287–317. DOI : <https://doi.org/10.1145/289.291>
- [33] Timothy L. Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing*, Jennifer Welch (Ed.). Springer Berlin, 300–314.
- [34] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A practical multi-word compare-and-swap operation. In *Distributed Computing*, Dahlia Malkhi (Ed.). Springer Berlin, 265–279.
- [35] D. Hendler, N. Shavit, and L. Yershulami. 2004. A scalable lock-free stack algorithm. In *SPAA*. ACM. DOI : <https://doi.org/10.1145/1007912.1007944>
- [36] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.* 13, 1 (Jan. 1991), 124–149. DOI : <https://doi.org/10.1145/114005.102808>
- [37] Maurice Herlihy. 1993. A methodology for implementing highly concurrent data objects. *ACM Trans. Prog. Lang. Syst.* 15, 5 (Nov. 1993), 745–770. DOI : <https://doi.org/10.1145/161468.161469>
- [38] M. Herlihy and E. Koskinen. 2008. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*. ACM. DOI : <https://doi.org/10.1145/1345206.1345237>
- [39] M. Herlihy and J. E. B. Moss. 1993. Transactional memory: architectural support for lock-free data structures. In *ISCA*. ACM. DOI : <https://doi.org/10.1145/165123.165164>
- [40] M. Herlihy and N. Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc.
- [41] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shriram. 2016. Type-aware transactions for faster concurrent code. In *ECCS*. ACM, 31.
- [42] Intel. 2020. Persistent Memory Programming. Retrieved from <http://pmem.io>.
- [43] J. Izraelevitz, T. Kelly, and A. Kolli. 2016. Failure-atomic persistent memory updates via JUSTDO logging. In *ASPLOS*. ACM. DOI : <https://doi.org/10.1145/2872362.2872410>
- [44] J. Izraelevitz, H. Mendes, and M. L. Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing*. Springer Berlin, 313–327.
- [45] J. Izraelevitz, J. Yang, L. u Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arxiv:cs.DC/1903.05714* (2019).
- [46] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. 2017. ATOM: Atomic durability in non-volatile memory through hardware logging. *IEEE*. DOI : <https://doi.org/10.1109/HPCA.2017.50>
- [47] N. D. Kallimanis and E. Kanellou. 2016. Wait-free concurrent graph objects with dynamic traversals. In *OPODIS*, Vol. 46. DOI : <https://doi.org/10.4230/LIPIcs.OPODIS.2015.27>
- [48] Terence Kelly. 2019. Persistent memory programming on conventional hardware. *Queue* 17, 4 (Aug. 2019). DOI : <https://doi.org/10.1145/3358955.3358957>
- [49] R. M. Krishnan, J. Kim, A. Mathew, X. Fu, A. Demeri, C. Min, and S. Kannan. 2020. Durable transactional memory can scale with timestamp. In *ASPLOS*. ACM. DOI : <https://doi.org/10.1145/3373376.3378483>
- [50] P. LaBorde, L. Lebanoff, C. Peterson, D. Zhang, and D. Dechev. 2019. Wait-free dynamic transactions for linked data structures. In *PMAM*. ACM. DOI : <https://doi.org/10.1145/3303084.3309491>
- [51] K. Lamar, C. Peterson, and D. Dechev. 2020. Lock-free transactional vector. In *PMAM*. ACM.
- [52] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh. 2017. WORT: Write optimal radix tree for persistent memory storage systems. In *FAST*. Retrieved from <https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon>.
- [53] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram. 2019. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *SOSP*. ACM. DOI : <https://doi.org/10.1145/3341301.3359635>

- [54] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. In *ASPLOS*. ACM. DOI : <https://doi.org/10.1145/3037697.3037714>
- [55] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. 2018. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *MICRO*. IEEE. DOI : <https://doi.org/10.1109/MICRO.2018.00029>
- [56] Yujie Liu. 2015. *Crafting Concurrent Data Structures*. Ph.D. Dissertation. Lehigh University.
- [57] Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. 2018. Persistent Memory Transactions. arxiv:cs.DC/1804.00701 (2018).
- [58] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson. 2017. Atomic in-place updates for non-volatile main memories with Kamino-Tx. In *EuroSys*. ACM. DOI : <https://doi.org/10.1145/3064176.3064215>
- [59] A. Memaripour, J. Izraelevitz, and S. Swanson. 2020. Pronto: Easy and fast persistence for volatile data structures. In *ASPLOS*. ACM. DOI : <https://doi.org/10.1145/3373376.3378456>
- [60] A. Memaripour and S. Swanson. 2018. Breeze: User-level access to non-volatile main memories for legacy software. In *ICCD*. 413–422.
- [61] Barzan Mozafari, Jags Ramnarayan, Sudhir Menon, Yogesh Mahajan, Soubhik Chakraborty, Hemant Bhanawat, and Kishor Bachhav. 2017. SnappyData: A unified cluster for streaming, transactions and interactive analytics. In *CIDR*.
- [62] MySQL. 2020. MySQL. Retrieved from <https://www.mysql.com/>.
- [63] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. 2017. An analysis of persistent memory use with WHISPER. In *ASPLOS*. ACM. DOI : <https://doi.org/10.1145/3037697.3037730>
- [64] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey III, D. R. Chakrabarti, and M. L. Scott. 2017. Dali: A periodically persistent hash map. In *DISC*, Vol. 91. 37:1–37:16. DOI : <https://doi.org/10.4230/LIPIcs.DISC.2017.37>
- [65] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-Tree for storage class memory. In *SIGMOD*. ACM. DOI : <https://doi.org/10.1145/2882903.2915251>
- [66] Christos H. Papadimitriou. 1979. Serializability of concurrent database updates. In *J. ACM* 26, 4 (1979).
- [67] pmem team. 2020. PMDK: Persistent Memory Development Kit. Retrieved from <https://github.com/pmem/pmdk/>.
- [68] pmem team. 2020. pmemkv: Key/Value Datastore for Persistent Memory. Retrieved from <https://github.com/pmem/pmemkv>.
- [69] pmem team. 2020. Pmemkv-tools: Optional Tools and Utilities for Pmemkv. Retrieved from <https://github.com/pmem/pmemkv-tools>.
- [70] P. Ramalhete, A. Correia, P. Felber, and N. Cohen. 2019. OneFile: A wait-free persistent transactional memory. In *DSN*. IEEE, 151–163. DOI : <https://doi.org/10.1109/DSN.2019.00028>
- [71] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner. 2015. NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *IMDM*. ACM. DOI : <https://doi.org/10.1145/2803140.2803144>
- [72] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. 2011. Testing atomicity of composed concurrent operations. In *OOPSLA*. ACM. DOI : <https://doi.org/10.1145/2048066.2048073>
- [73] N. Shavit and D. Touitou. 1997. Software transactional memory. *Distrib. Comput.* 10, 2 (1997), 99–116.
- [74] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin. 2017. Proteus: A flexible and fast software supported hardware logging approach for NVM. In *MICRO*. IEEE.
- [75] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. 2009. A comprehensive strategy for contention management in software transactional memory. *SIGPLAN Not.* 44, 4 (2009). DOI : <https://doi.org/10.1145/1594835.1504199>
- [76] A. Spiegelman, G. Golan-Gueta, and I. Keidar. 2016. Transactional data structure libraries. In *PLDI*. ACM. DOI : <https://doi.org/10.1145/2908080.2908112>
- [77] T. Strandell et al. 2010. *Open Source Database Systems: Systems Study, Performance and Scalability*. VDM Publishing.
- [78] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*. USENIX. Retrieved from <http://dl.acm.org/citation.cfm?id=1960475.1960480>.
- [79] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. 2014. Aerie: Flexible file-system interfaces to storage-class memory. In *EuroSys*.
- [80] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. *SIGPLAN Not.* 47, 4 (Mar. 2011), 91–104. DOI : <https://doi.org/10.1145/2248487.1950379>
- [81] T. Wang, J. Levandoski, and P. Larson. 2018. Easy lock-free indexing in non-volatile memory. In *ICDE*. IEEE. DOI : <https://doi.org/10.1109/ICDE.2018.00049>
- [82] T. Wang, S. Simbasivam, Y. Solihin, and J. Tuck. 2017. Hardware supported persistent object translation. In *MICRO*.
- [83] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, and Chandra Kintala. 1995. Checkpointing and its applications. In *FTCS*. IEEE, 22–31.
- [84] Jian X. and S. Swanson. 2016. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*. USENIX. Retrieved from <https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu>.

- [85] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiyah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff. 2017. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *SOSP'17*.
- [86] Quanqing Xu, Khin Mi Mi Aung, Yongqing Zhu, and Khai Leong Yong. 2018. A blockchain-based storage system for data analytics in the internet of things. In *New Advances in the Internet of Things*. Springer, 119–138.
- [87] Y. Xu, Y. Solihin, and X. Shen. 2020. MERR: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. In *ASPLOS*. ACM. DOI: <https://doi.org/10.1145/3373376.3378492>
- [88] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. 2015. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *FAST*.
- [89] P. Zardoshti, T. Zhou, Y. Liu, and M. Spear. 2019. Optimizing persistent memory transactions. In *PACT*. IEEE.
- [90] D. Zhang and D. Dechev. 2016. An efficient lock-free logarithmic search data structure based on multi-dimensional list. In *ICDCS*. IEEE. DOI: <https://doi.org/10.1109/ICDCS.2016.19>
- [91] D. Zhang and D. Dechev. 2016. A lock-free priority queue design based on multi-dimensional linked lists. *IEEE Trans. Parallel Distrib. Syst.* 27, 3 (Mar. 2016), 613–626.
- [92] D. Zhang and D. Dechev. 2016. Lock-free transactions without rollbacks for linked data structures. In *SPAA*. ACM. DOI: <https://doi.org/10.1145/2935764.2935780>
- [93] D. Zhang, P. Laborde, L. Lebanoff, and D. Dechev. 2018. Lock-free transactional transformation for linked data structures. *ACM Trans. Parallel Comput.* 5, 1 (June 2018). DOI: <https://doi.org/10.1145/3209690>
- [94] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. *Proc. ACM Prog. Lang.* 3, OOPSLA (Oct. 2019). DOI: <https://doi.org/10.1145/3360554>

Received July 2020; revised December 2020; accepted December 2020