DudeTx: Durable Transactions Made Decoupled

MENGXING LIU, MINGXING ZHANG, and KANG CHEN, Tsinghua University
XUEHAI QIAN, University of Southern California
YONGWEI WU and WEIMIN ZHENG, Tsinghua University
JINGLEI REN, Microsoft Research

Emerging non-volatile memory (NVM) offers non-volatility, byte-addressability, and fast access at the same time. It is suggested that programs should access NVM directly through CPU load and store instructions. To guarantee crash consistency, durable transactions are regarded as a common choice of applications for accessing persistent memory data. However, existing durable transaction systems employ either *undo logging*, which requires a fence for every memory write, or *redo logging*, which requires intercepting all memory reads within transactions. Both approaches incur significant overhead.

This article presents DudeTx, a crash-consistent durable transaction system that avoids the drawbacks of both undo and redo logging. DudeTx uses shadow DRAM to *decouple* the execution of a durable transaction into three fully asynchronous steps. The advantage is that only minimal fences and no memory read instrumentation are required. This design enables an out-of-the-box concurrency control mechanism, transactional memory or fine-grained locks, to be used as an independent component. The evaluation results show that DudeTx adds durability to a software transactional memory system with only 7.4%–24.6% throughput degradation. Compared to typical existing durable transaction systems, DudeTx provides 1.7× –4.4× higher throughput. Moreover, DudeTx can be implemented with hardware transactional memory or lock-based concurrency control, leading to a further 1.7× and 3.3× speedup, respectively.

CCS Concepts: • **Hardware → Memory and dense storage**;

Additional Key Words and Phrases: Non-volatile memory, durable transaction, decoupled

ACM Reference format:
Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2018. DudeTx: Durable Transactions Made Decoupled. *ACM Trans. Storage* 14, 1, Article 7 (March 2018), 28 pages.
https://doi.org/10.1145/3177920

M. Liu and M. Zhang equally contributed to this work.
An earlier version of this work [38] appeared in Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17).

This work is supported by National Key Research and Development Program of China (2016YFB1000504), Natural Science Foundation of China (61433008, 61373145, 61572280, U1435216, 61402198), and National Basic Research (973) Program of China (2014CB340402). This work is also supported by NSF CRII-1657333, Spanish Government and European ERDF under TIN2010-21291-C02-01 and Consolider CSD2007-00050.

Authors' addresses: M. Liu, M. Zhang, K. Chen, Y. Wu, and W. Zheng are with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China; emails: liu-mx15@mails.tsinghua.edu.cn, {zhang-mingxing, chenkang, wuyw, zwm-dcs}@tsinghua.edu.cn; M. Zhang is also with the Graduate School at Shenzhen, Tsinghua University and Sangfor Technologies, Inc; X. Qian is with the University of Southern California; email: xuehai.qian@usc.edu; J. Ren is with the Microsoft Research Asia; email: jinren@microsoft.com.
Corresponding authors: K. Chen (chenkang@tsinghua.edu.cn) and Y. Wu (wuyw@tsinghua.edu.cn).

https://doi.org/10.1145/3177920

1 INTRODUCTION

Emerging non-volatile memory (NVM) technologies [5, 31, 71] are promising because they offer non-volatility, byte-addressability, and fast access at the same time. Phase change memory (PCM) [36, 48], spin-transfer torque RAM (STT-RAM) [2, 35], and ReRAM [1] are representative examples of NVM. Notably, Intel and Micron recently announced 3D XPoint, a commercial NVM product on the way to the market [30]. As NVM enjoys DRAM-level access latency, empirical studies [14, 24, 34, 60, 72, 73] suggest that NVM should be directly accessed through the processor-memory bus by load/store instructions, making a *persistent memory*. It avoids the overhead of legacy block-oriented file systems or databases. Persistent memory also allows programmers to update persistent data structures directly at byte level and without the need for serialization to storage.

While persistent memory provides non-volatility, it is challenging for an application to ensure correct recovery from the persistent data on a system crash, namely, *crash consistency* [45, 60]. A solution to this problem is using the crash-consistent *durable transaction*, which makes a group of memory updates appear as one *atomic* unit with respect to a system crash. Durable transactions offer both strong semantics and an easy-to-use interface. Many prior works [24, 28, 34, 60] have provided durable transactions as the software interface for accessing persistent memory data.

Most implementations of durable transactions enforce crash consistency through logging [14, 28, 34, 67]. However, there is a *dilemma* in choosing between undo logging and redo logging [61], the two basic logging paradigms. In *undo logging*, the original (old) values are stored to a log before they are modified by a transaction. Having its old value preserved in the log, the data can be modified *in-place*. Therefore, a memory load can directly read the latest value without being intercepted and remapped to a different address. This avoids noticeable overheads, thus many systems [14, 28, 34, 67] use undo logging. However, undo logging requires each in-place update to perform *after* the logged old value is persistent, so that a transaction is able to roll back to the original value if a crash occurs. To enforce this order, the system uses *persist*, a costly operation to flush the logged old value from CPU caches to NVM for *every* update. This incurs significant overhead [24, 60, 61]. To mitigate the overhead, one approach [28, 34] is to log *all* the old values that are about to be updated in a transaction at once, reducing the frequency of persist ordering from once per update to once per transaction. However, this method requires prior knowledge of the write set of a transaction and hence supports only *static* transactions (accordingly, we refer to transactions without predefined write sets as *dynamic* ones).

To the contrary, *redo logging* [24, 60, 67] imposes only one persist order for each transaction, no matter whether the transaction is static or dynamic. In particular, all memory updates of a transaction are first buffered in a log in persistent memory, and then applied to the data in-place. The process is referred to as *update redirection*. The system only needs to guarantee that updates of old values happen *after* the whole log is persisted, incurring only one persist per transaction. However, since the data updates are buffered in the log, the reads of them in a transaction must be remapped to the log to obtain the latest values. The overhead of such address remapping entailed by update redirection is expensive [51, 61]. Recently, Kiln [73] avoids the overhead by using non-volatile caches. However, it requires non-trivial hardware changes, including modification of the cache coherent protocol.

The dilemma between undo and redo logging is essentially a tradeoff between update redirection cost and persist ordering cost. For any design, *either* in-place update with per-update persist ordering overhead (as in undo logging) *or* transaction-level persist ordering with update redirection overhead (as in redo logging) have to be employed. Nevertheless, our investigation demonstrates that it is possible to *make the best of both worlds* while supporting both dynamic and static transactions. The key insight of our solution is *decoupling* a durable transaction into three *fully asynchronous* steps. (1) `Perform`: execute the transaction on a shadow memory,[1] and produce a redo log for the transaction. (2) `Persist`: flush redo logs of transactions to persistent memory in an atomic manner. (3) `Reproduce`: modify original data in persistent memory according to the persisted redo logs. It is essential that we never directly write back dirty data from shadow memory to persistent memory—all the updates are realized via the logs. That means we never have to worry about uncontrollable CPU cache eviction violating the crash consistency guarantee, because the evicted data from CPU caches is only written to the shadow memory. By decoupling, our solution can perform in-place update in `Perform`, and requires only transaction-level persist ordering in `Persist` and `Reproduce`, achieving both minimal update redirection overhead and low persist ordering overhead.

A core component of our decoupled framework is a *shared*, *cross-transaction* shadow memory. It acts as a volatile mirror/cache of the persistent memory. More importantly, it only requires *page-level* mapping, which is coarse-grained and can be efficiently supported by the operating system and CPU hardware. In contrast, traditional redo logging incurs costly object-level or even byte-level address mapping implemented by software.

This article presents DudeTx, a **du**rable **de**coupled transaction system on persistent memory that implements the above insight and design rationale. Besides resolving the dilemma between undo and redo logging, DudeTx offers three other advantages as follows.

First, `Perform` in DudeTx is a decoupled step that simply executes concurrent transactions on shadow memory. As a result, users of DudeTx can optionally adopt an existing concurrency control mechanism to guarantee the isolation property of those transactions. To demonstrate this capability, we show the methods to integrate DudeTx with three main types of concurrency control mechanisms: *(1)* an *out-of-the-box* software transactional memory system (STM, Section 4.1); *(2)* a hardware transactional memory system that is slightly customized (HTM, Section 4.2); and *(3)* direct use of locks (Section 5). Although the implementation of these concurrency control mechanisms vastly differ from each other, the integration to DudeTx is almost the same and simple. In other words, we essentially provide a framework of designing and implementing durable transactions that can cooperate with existing concurrency control mechanisms.

Second, in DudeTx, each thread conducts `Perform` of consecutive transactions back-to-back, without stalling due to persistence. By design, DudeTx reduces persistence-induced stalls with less complexity than recently proposed relaxed persistence models [32, 34].

Third, the fully decoupled framework introduces many unique optimization opportunities, such as applying cross-transaction write combination and data compression to the redo logs before flushing them in `Persist`. These optimizations can improve the throughput and, in certain cases, significantly reduce the amount of writes to persistent memory (whose endurance is much lower than DRAM [11, 23]).

In summary, we make the following contributions.

- We propose a *decoupled* framework for implementing durable transactions on persistent memory. It incorporates a *shared*, *cross-transaction* shadow memory, and achieves advantages of both traditional undo and redo logging. One of our implementations, based on an existing STM named TinySTM [21], achieves 1.7×–4.4× higher throughput than prior systems, NVML [28] and Mnemosyne [60], while running various benchmarks including TPC-C [58] and TATP [56].

[1]The shadow memory can be either volatile or non-volatile. Considering performance advantage, we assume use of DRAM.

—We demonstrate the benefits of several optimizations that are uniquely enabled by our decoupled framework. Specifically, we apply cross-transaction write combination and data compression to the redo logs. Our experiments show that these techniques can reduce the amount of writes to persistent memory by up to 93%.

—For the first time, an *out-of-the-box* concurrency control mechanism, STM or HTM (with minor modification) or locking, can be used as a stand-alone component for implementing durable transactions on persistent memory. Most prior durable transaction systems are coupled with and bound to a specific transactional memory implementation. Besides, our evaluation shows that HTM and fine-grained locks give an additional 1.7× and 3.3× speedup over the STM-based DudeTx, respectively.

2 BACKGROUND AND MOTIVATION

2.1 Requirements of Durable Transactions

Requirements of database transactions have been well defined. They are applicable to persistent memory. We briefly review the four properties of a full transaction on persistent memory, i.e., atomicity, consistency, isolation, and durability (ACID) [25].

First, a persistent memory system may contain volatile storage components, such as CPU caches or DRAM, for performance purposes [46]. Still, it has to retain data updates of any acknowledged transaction despite power loss or system crashes. This property is referred to as *durability*. Second, a logical update of data records performed by a transaction typically constitutes multiple writes to various addresses in persistent memory. To ensure correctness of application semantics, these writes have to be executed "all or nothing." That means, either all writes of a transaction are successfully performed, or none of them are performed (i.e., data in persistent memory is intact). This property is called *atomicity*. Third, when multiple transactions execute concurrently in the system, each transaction should see an isolated view of the memory data. Specifically, data updates made by one transaction should be invisible to other concurrent transactions until the transaction is committed. This property is *isolation*. There are three paradigms to realize isolation: STM, HTM, and lock acquiring/releasing. Transactional memory (TM) has an easy-to-use interface, but sacrifices performance because it has to trace all memory read and write operations of a transaction as a whole. On the contrary, fine-grained locks may enable more concurrency. But it is hard to develop correct lock-based programs. Finally, the *consistency* property means that each update to the memory data only brings the data from one consistent state to another. The definition of a consistent state is application-specific. It most cases, the application rather than the storage system is responsible for defining consistent data updates. Prior research [21, 22, 25, 26] has shown that the transaction with atomicity and isolation guarantees is a powerful and convenient interface to realize consistency.

The durable transaction in this article provides full ACID properties. Thus, it is different from the well-known storage transaction described in [39, 41], which only requires one transaction to be atomically persisted to storage devices.

2.2 Drawbacks of Undo and Redo Logging

In order to develop durable transactions for persistent memory, logging techniques have been proposed. For example, NVML [28], NV-Heap [14], and DCT [34] are based on *undo logging*, which records old values to a separate log before actually modifying the data. Undo logging is prevalent because it enables transactions to perform *in-place* updates, avoiding the overhead of redirecting updates. However, undo logging has to ensure the following persist order: for *each* single update, the undo log is flushed to persistent memory before the corresponding in-place update is applied to the real data structure. Typically, the persist order is enforced by a persist operation which includes cache line flushing (e.g., `CLFLUSHOPT`, `CLWB`) and store ordering (e.g., `SFENCE`, or deprecated `PCOMMIT`) [54], incurring significant overhead [11, 20, 57, 60]. Hence, it is prohibitively expensive to enforce persist ordering for *every* memory write in a transaction.

To mitigate this issue, DCT [34] and NVML [28] perform *all* undo logging of old data at the beginning of a transaction, so that only one persist ordering is needed for the transaction. However, this solution requires prior knowledge of the write set of a transaction and hence only supports static transactions. If such knowledge is not available, their authors claimed that a program must execute a read phase to identify all regions it might touch and acquire all locks. Clearly, it would introduce considerable performance penalty.

Another approach of supporting durable transactions is based on *redo logging*, which requires only one persist ordering for each transaction, no matter if it is a static or dynamic transaction. Take Mnemosyne [60] as an example. Every memory write in a transaction is intercepted and transformed to an append operation to the redo log, which stores the new values (uncommitted data); at the same time, all memory reads of the transaction to the uncommitted data are redirected to the redo log to obtain the latest values. After the transaction is committed, all the redo log records of the transaction are flushed to persistent memory at once. The tradeoff is that one can reduce the number of persist ordering (as in undo logging), at the cost of intercepting and redirecting writes and reads.

LOC [41] removes persistence ordering of redo logging by modifying hardware. It provides eager commit and speculative persistence to remove intra-transaction ordering and inter-transaction ordering, respectively. However, these existing techniques still suffer redirecting overhead. Although object-based [3, 37, 55] or block-based [53, 69] storage systems can alleviate the cost by increasing the redirection or mapping granularity, TM systems for persistent memory have to support fine-grained byte-level redirection or mapping [22, 62]. Indeed, a *page-level* mapping mechanism (e.g., as used in eNVy [67]), incurs lower overhead than a finer-grained mapping mechanism (e.g., as used in Mnemosyne [60] or SoftWrAP [24]) because page-level mappings occupy relatively small memory space and can be accelerated by CPU Translation Lookaside Buffer (TLB) hardware. However, it introduces the write amplification issue in which an entire page may have to be read or written in order to update just a few bytes of the page. In contrast, a fine-grained mapping can minimize the write amplification. Overall, the redirection-induced cost in the traditional redo logging mechanisms is significant [51, 61].

Considering both undo and redo logging, we realized that the overhead of persist ordering and address mapping are introduced due to the lack of efficient control of *when* a memory store actually modifies the data in persistent memory. With undo logging, the system does not know when an in-place update will be evicted from the CPU caches to the persistent memory. To guarantee the ability to rollback a transaction, we have to ensure that the undo log becomes persistent before issuing the in-place data update to persistent memory. With redo logging, the new data is stored in a redo log, so that the evictions do not matter. Unfortunately, as a result, the address mapping mechanism and its overhead become inevitable.

In essence, current systems couple the memory stores in volatile memory with their persistence in NVM. Without significant hardware modifications, we believe that *decoupling* the persistence is the best (and possibly the only) way to avoid the drawbacks of both undo and redo logging and reduce the performance penalty.

3 THE DECOUPLED FRAMEWORK

Based on the principle of decoupling, this section describes the design of DudeTx, a C library that provides the guarantees of complete ACID in spite of a system crash. DudeTx provides two kinds of interface: (1) TM-based interface and (2) lock-based interface, which share a same inner framework. In this section, we take TM-based interface as an example to show this inner framework. Implementation-dependent issues will be described in Section 4 (for TM-based DudeTx) and Section 5 (for directly using locks), respectively. In other words, in this section, we focus on describing the method of building durable transactions with an out-of-the-box isolation control component.

3.1 The Decoupled Framework

To realize the efficient decoupled execution, we make the following design choices. First, we maintain a single *shared*, *cross-transaction* shadow memory, which is logically a volatile mirror or cache of the whole persistent memory space. The key feature is that the shadow memory is shared among transactions rather than transaction-local [24, 60, 67]. It enables cost-effective *page-level* management of the shadow memory, which requires less metadata than finer-grained management and enables use of hardware support such as TLB. A transaction-local shadow memory hardly uses page-level management because of excessive memory I/O. When a small object is to be updated, a full shadow page may have to be read from the persistent memory. In DudeTx, the page is *not* discarded after the transaction ends. That means subsequent transactions can still access that page. Therefore, the cost of extra memory I/O is amortized among multiple transactions in DudeTx.

Second, we use an *out-of-the-box* TM[2] to execute transactions on the shadow memory for isolation and consistency. A TM implementation can by definition execute transactions in isolation and ensure application-defined consistency, by detecting and resolving conflicts. Decoupled with TM, the durability and atomicity of the updates of each transaction are ensured by DudeTx library during flushing the updates to persistent memory after TM commits the transaction.

Third, we use a *redo log* as the *only means* to transfer updates on the shadow memory to the persistent memory. The reason why we do not choose undo logging is that its persist ordering requirement would inevitably introduce persistence latency into the critical path of transaction execution. The redo log ensures that no dirty data is ever directly written back to the persistent memory. This design guarantees that a TM can safely execute on the shadow memory without affecting persistent memory. Also, all updates to the persistent memory are crash consistent.

Following the above design choices, we build a framework to realize the whole execution of an ACID transaction into three *decoupled*, *asynchronous* steps as follows.

Perform: Transactions execute with an out-of-the-box TM on top of the shadow memory and access volatile data by load/store instructions. The TM guarantee the isolation and atomicity properties in the volatile memory. Each committed transaction generates one redo log, which is temporarily stored into one of the *thread-local* log buffers. This avoids contention among threads.

Persist: The committed transactions are persisted by flushing their redo logs to a log region in persistent memory. It can be done by the background threads (typically one is enough). This step ensures the atomicity and durability properties of the transactions.

Reproduce: The system finally reproduces the updates of a transaction and modifies the data in persistent memory according to the redo log. This is the only step that operates on the actual persistent data structures of the application. `Reproduce` can be performed in the background.

Although `Reproduce` is necessary for completing a transaction, a transaction is considered to be persistent in DudeTx once its `Persist` step is finished, because we can always perform `Reproduce` as long as the redo log is persisted.

Figure 1 depicts DudeTx's memory architecture and demonstrates the three decoupled steps. The NVM and (part of) DRAM are mapped to an application's address space. Different from the

[2]As we have mentioned before, in this section, we will use TM-based DudeTx as an example to show the shared inner framework. Issues related to lock-based DudeTx will be discussed in Section 5.

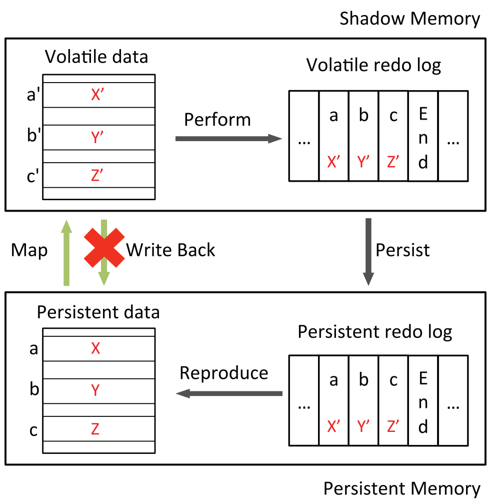


Fig. 1. DudeTx memory architecture and data movement in the three steps. Data X, Y, and Z locate at addresses a, b, and c in the persistent memory, respectively. These addresses are mapped to a′, b′, and c′ in the shadow memory.

regular memory mapping, DudeTx maintains two physical pages for a logical page: *(1)* one volatile *shadow page* as the shadow memory in DRAM, which is visible to the application and can be directly accessed by load/store instructions of transactions (in the `Perform` step); and *(2)* one *persistent page* stored in the persistent memory or NVM, which is only modified by background threads in the `Reproduce` step. The two pages communicate only via redo logs, which also occupy two regions in DRAM and NVM—the volatile log region and persistent log region, respectively. The arrows show data movement in the three steps. We can see that each of the two log regions acts as a channel connecting two steps. If the size of the shadow memory is equal to that of the persistent memory, the address mapping between them is simply a constant offset; otherwise, we implement dynamic address mappings and page-in/out as will be described later in Section 3.6. Next, we introduce more details of the three steps.

DudeTx offers five APIs for applications to use: `dtmBegin`, `dtmEnd`, `dtmAbort`, `dtmRead`, and `dtmWrite`. A transaction should be wrapped with a pair of `dtmBegin` and `dtmEnd`. Users need to replace all memory reads/writes in a transaction with `dtmRead` and `dtmWrite`, respectively. Algorithm 1 gives an example of using DudeTx to perform a transaction that transfers $1 from one account to another in a bank.

3.2 Perform

DudeTx integrates an out-of-the-box TM system to perform the functions of the five APIs in the shadow memory. Algorithm 2 presents the pseudo-code to implement DudeTx's APIs with an existing TM, whose corresponding functions are referred to as `tmBegin`, `tmEnd`, `tmAbort`, `tmRead`, and `tmWrite`. `dtmBegin` and `dtmRead` simply call `tmBegin` and `tmRead`, respectively. `dtmWrite` has to append an entry to the local volatile redo log of the transaction, in addition to calling `tmWrite`. The entry consists of the address and the value of the write. `dtmEnd` appends an end mark to the volatile redo log, and `dtmAbort` clears redo log entries generated by the aborted transaction.

For simplicity, DudeTx assumes that the TM provides *transaction ID* that is a globally unique and monotonically increasing number. The transaction ID indicates the global order of committed

ALGORITHM 1: Transactional Transfer

```
Func transfer(src, dst)
    /* transcation begin                                   */
    dtmBegin();
    /* using dtmRead to read                               */
    srcBalance = dtmRead(accounts[src]);
    /* checking balance                                    */
    if srcBalance ≤ 0 then
        /* abort in transaction                            */
        dtmAbort();
    end
    /* using dtmWrite to write                             */
    dtmWrite(accounts[src], srcBalance - 1);
    dstBalance = dtmRead(accounts[dst]);
    dtmWrite(accounts[dst], dstBalance + 1);
    /* transaction end                                     */
    dtmEnd();
```

ALGORITHM 2: DudeTxAPIs Implementation

```
Func dtmBegin()
    return tmBegin();
Func dtmRead(addr)
    return tmRead(addr);
Func dtmWrite(addr, val)
    /* the thread-local redo log recording the address and value */
    vlog.AppendEntry(addr, val);
    /* transactional memory write                          */
    return tmWrite(addr,val);
Func dtmAbort()
    /* clear log entries generated in this transaction     */
    vlog.PopToLastTx();
    tmAbort();
Func dtmEnd()
    /* get transaction id                                  */
    tid = tmEnd();
    /* append an end mark                                  */
    vlog.AppendTxEnd(tid);
```

transactions in `Perform`. DudeTx follows this order to `Reproduce` transactions based on redo logs from different threads. Most existing TM techniques can produce this ID as they already maintain a global clock in their implementations [21, 22]. Meanwhile, according to our experiments, the current maximum transaction throughput has not reached such a level that the global clock is the bottleneck.

In DudeTx, each volatile redo log is stored in a fixed-length, circular buffer, which maintains a pair of cursors pointing to the log head and end, respectively. Because the size of a volatile log buffer is limited, if the buffer is full, the `Perform` thread will be blocked and wait for a thread doing `Persist` to flush the log and unblock it. In our evaluation, we find that the log flushing of `Persist` is generally faster than the log entry generation of `Perform`.[3] Therefore, a thread rarely blocks for this reason in practice.

3.3 Persist

DudeTx maintains one or more background threads (typically one is enough) to continuously flush redo logs from the volatile log region to the persistent log region. These `Persist` threads work together to determine a global *latest durable transaction ID* ("durable ID" for short) so that *all* transactions with smaller durable IDs are persistent. Consequently, a `Perform` thread can query the global durable ID and response to callers of earlier transactions with the status of durability.

When a `Perform` thread is created, DudeTx assigns a `Persist` thread that is responsible for flushing redo logs of that `Perform` thread. The `Persist` thread maintains, in the persistent log region, a thread-local persistent log buffer, similar to the volatile log buffer accessed by the `Perform` thread. The redo logs do not have to be flushed according to the commit order of transactions, because only the `Reproduce` step updates actual data in persistent memory. In other words, only `Reproduce` needs to follow the transaction commit order. Due to the potential out-of-order redo log flushing, `Persist` of a transaction is only considered to be finished when the global durable ID is larger than its transaction ID. In any case, DudeTx does not need to wait for the finish of `Reproduce` to acknowledge durability of a transaction, because the transactions with `Persist` finished can be always replayed with the redo log to reproduce all memory updates.

Moreover, although the redo log of a transaction can be flushed to persistent memory immediately after the transaction is committed in `Perform`, DudeTx have the freedom to persist redo logs in a batched manner. This brings new optimization opportunities that are only possible in our decoupled framework. We use cross-transaction log combination and log compression to reduce the amount of writes to persistent memory which has limited endurance compared with DRAM.

Log Combination. If two writes modify the same memory address, they can be coalesced when flushed to the persistent log. The earlier write of them can be saved as long as they are flushed atomically. Such log combination has been applied to the cross-transaction case in other fields, e.g., MobiFS [50]. However, for durable transaction systems, this technique is only applicable with our decoupled framework. The `Persist` thread splits successive transactions into groups. In each group, it reads log entries by the order of their transaction ID and inserts them into a hash table. Later log entries can overwrite earlier entries if they write the same data addresses. Finally, a group of log entries is flushed in an atomic manner.

Log Compression. The write size of `Persist` can be further reduced by compressing the combined logs before flushing them to persistent memory. DudeTx uses the lz4 [15] algorithm and achieves a compression ratio over 69% in our experiments. It is important to note that the decompression is *not* always needed. We can keep a redo log in the volatile log region even after the log has been flushed to persistent memory, if extra memory space is available. `Reproduce` can directly read redo logs from the volatile log region. Therefore, without a crash, the overhead to read out and decompress the redo logs from persistent memory can be avoided.

[3]Pure log generation (in the cache) is faster than log flushing (in the NVM). But the log generation is only one part of the transaction execution. Accessing data, acquiring/releasing locks, and so forth consumes most of the CPU times. Thus, in the real world, log flushing is faster.

3.4 Reproduce

In Reproduce, DudeTx replays redo logs according to the order determined by transaction IDs and recycles the replayed logs. The only necessary persistence ordering in this step is to ensure that recycling happens after the logs have reproduced their updates to the persistent memory. DudeTx can only start reproducing a transaction after it is durable, i.e., its transaction ID is smaller than the current global durable ID. When cross-transaction log combination is used, the recycle granularity is enlarged to a group of transactions.

3.5 Recovery

During recovery, DudeTx scans the whole persistent log region and replays logs that have not been processed by Reproduce. Similar to Reproduce, the recovery replays the redo logs in an increasing order of their transaction IDs until it finds a transaction whose log is omitted or incomplete. The incomplete log, along with its corresponding transaction, is abandoned. Since the transaction must have not been acknowledged with durability, the application should re-execute the transaction if it promises ACID transactions to users.

Another step in recovery is to restore persistent memory allocation information. Similar to other durable transaction systems [24, 28, 60], DudeTx provides pmalloc and pfree for applications to allocate and free persistent memory. The specific allocation algorithm is orthogonal to our design, but the system needs a separate log for each thread to record all the pmalloc/pfree operations of each transaction. On recovery, these logs are also scanned so that DudeTx can determine which regions of persistent memory are allocated.

3.6 Memory Management

In a practical hybrid system configuration, the size of shadow memory is typically smaller than the size of persistent memory. This is because one may intend to use DRAM only as the shadow to achieve the best performance and, currently, the density of NVM is higher than DRAM. In such a case, a paging mechanism is required, and its implementation is critical to the performance of the whole DudeTx system.

Our paging mechanism is similar to the one used in operating systems. When a transaction accesses a memory address whose page is not present in the shadow memory, a page fault is triggered and the page in persistent memory is swapped in as a shadow page. If no shadow memory space is available, DudeTx has to evict an existing shadow page. Consequently, the persistent-shadow page address mappings have to be updated accordingly. Meanwhile, different from the traditional OS paging, the evicted page is simply discarded without any writing back. It is correct because all updates to that page by different threads have already been recorded in redo logs, which will be eventually reproduced in persistent memory asynchronously (if the transaction is successfully committed in Perform step).

Since a page could be evicted without writing back, we are required to address a subtle issue: we need to ensure that all updates to a persistent page have been reproduced when the page needs to be swapped into the shadow memory. To ensure this requirement, we maintain a *touching ID* for each page, which is the ID of the last transaction that writes on the page. Before loading a persistent page into the shadow memory, DudeTx compares its touching ID and the ID of the most recent transaction that has finished Reproduce. If touching ID is bigger, that is, modified data in this page has not been updated in persistent memory yet, the page loading needs to wait until the touching ID becomes smaller.

As for the address translation mechanism, in DudeTx, we provide both hardware-based and software-based paging methods.

ALGORITHM 3: Software-Based Address Paging

```
Func transfer(vaddr)
    page = addr_to_page(vaddr);
    while true do
        if page.ref < 0 then
            swapIn(page);
        else
            ref = page.ref;
            if compare_and_swap(page.ref, ref, ref+1) then
                return translate(addr)
            end
        end
    end
Func swapOut(page)
    if compare_and_swap(page.ref, 0, -1) then
        return true;
    else
        return false;
    end
```

3.6.1 Hardware-Based Paging. As we mentioned above, there are two physical memory regions: the shadow memory and the persistent memory. But they are both invisible to users. We provide a virtual address space for users, which is supposed to be persistent and has the same size of the persistent memory region. The virtual address is logically equal to the persistent memory address. That is, if a user wants to access virtual memory, which he regards as persistent memory, the OS would translate this address to shadow memory transparently. This mapping is done by page table in OS, so the program can directly access value on the shadow memory through the assistance from TLB. No overhead exists in this translation. When a page fault happens, a free shadow page is selected to map this virtual page and page table is updated.

We use Dune [6] to manage page faults and page mappings. Dune utilizes Intel's VT-x virtualization technique to enable user-space programs to handle page fault exceptions. When there are not enough free shadow pages, we pick up some pages and evict them directly. Before that, we have to delete page table entries and flush TLB entries in all processors, i.e., do TLB shootdown. We add a TLB shootdown feature to Dune by reusing the inter-processor interrupt facility in the Linux kernel. DudeTx has to stall all processors and issue the `INVVPID` instruction to flush all TLB entries.

3.6.2 Software-Based Paging. Although the hardware-based paging mechanism is fast in translating, its cost on page eviction is high (i.e., it has to stop all processors for doing TLB shootdown). As a result, it is suitable for workloads that have a low swapping frequency. As a complement, we also implement a software-based paging mechanism that maintains a simple one-level page table in DRAM. In this case, every `dtmRead`/`dtmWrite` operation has to first look up the page table for translating a virtual address to an address[4] in the shadow memory. The address translation and page swapping are all controlled by our software library.

[4]In the software-based approach, this address is also a virtual address managed by OS. But this address is not exposed by DudeTx to the application.

Software-based paging does not have to stop all processors when swapping out pages. Page swapping in/out can be done with other transactions' execution simultaneously. Thus, it is more efficient when there are too many page evictions than hardware-based paging. But we have to ensure that a page to swap out is not being used by any other transactions. To that end, we record a reference in each page that is the number of transactions accessing the page. A page with reference more than zero is not allowed to swap out. If reference of a page is smaller than zero, this page is swapped out.

Algorithm 3 shows how we use this reference to transfer a virtual address and swap out pages. *(1)* In *transfer*, we check if the reference of the page is smaller than zero first. If yes, a *swap in* procedure has to be invoked. Otherwise, we try to use compare and swap based atomic instruction to update the reference value. Then, it is safe to read the global page table and translate the address. *(2)* In *swapOut*, we simply use a compare and swap instruction to mark that the page is evicted. The result is *false* means that the page is used by other transactions or has already been swapped out.

4 TM-BASED DUDETX

In this section, we describe the implementation of DudeTx on STM and HTM.

4.1 STM-Based Implementation

We developed an implementation of DudeTx using Tiny-STM [21], a lightweight STM that represents the category of *time-based* software TM [21, 22, 52]. TinySTM maintains a timestamp for every object and tries to linearize concurrent transactions. If a transaction fails to access a "snapshot" of all the objects it accesses during execution, it is aborted via a `longjump` to roll back to the begin of execution. One of the following two methods can be used to support rollback of memory updates in TinySTM: *Write-back access* does not modify the shadow memory during a transaction, but records the write set, which is simply discarded if the transaction is aborted. *Write-through access* directly modifies the shadow memory but records old values before modification. If the transaction is aborted, the old values are restored. We choose the write-through access in our implementation as it permits in-place update.[5]

By design, the three steps in DudeTx's framework are asynchronous, ensuring different properties of ACID. In the implementation, applications can use different options to ensure these properties. One option is to let `dtmEnd` wait for `Persist` until the transaction is durable. In that case, DudeTx directly provides full ACID transactions through its APIs as specified in Section 3.2. The other option is more flexible and may have performance advantage. In this case, `dtmEnd` immediately returns without waiting for its log to be durable, so that the `Perform` thread can execute transactions back-to-back and the execution does not suffer from persistence-induced stalls. At this point, the transaction is not guaranteed to be durable. An application may require that a transaction should be durable before responding to external users. To fulfill such requirement, DudeTx can expose the application individual transaction IDs and the global durable ID to the users. Then the application based on DudeTx can periodically check the global durable ID and notify the users that all transactions with smaller IDs than durable ID can be acknowledged with full ACID.

4.2 HTM-Based Implementation

Intel's Haswell processor supports Restricted Transactional Memory (RTM) [29]. As a representative HTM implementation, RTM offers `XBEGIN` and `XEND` to specify an HTM transaction, a code

[5]Essentially, this is a form of undo logging, but as the shadow memory is volatile, there is no costly persistence ordering issue.

region to be executed atomically and in isolation. Conflicts between HTM transactions on different cores are detected by the cache coherence protocol. Programs can choose to re-execute an aborted transaction or execute a fallback routine.

The interface of HTM is largely identical to a STM, so using HTM for DudeTx is straightforward. We replace `tmBegin`, `tmAbort`, `tmRead`, and `tmWrite` in Algorithm 2 with `XBEGIN`, `XABORT`, regular memory read, and regular memory write, respectively.

The only issue is that `XEND` does not return a transaction ID for DudeTx to determine the transaction order. It is easy to maintain a global transaction ID generator in software, such as simply a long integer incremented in every HTM transaction. Unfortunately, it will lead to a prohibitive abort rate in the current HTM system, because the global integer itself will introduce conflicts. We cannot manipulate the ID generator outside a HTM transaction, otherwise the ID value is not guaranteed to reflect the real order of transactions. To resolve this issue, a minor hardware change is proposed: we simply require the HTM to ignore conflicts on certain memory addresses. In this way, the software maintained transaction ID could be allocated in this region and does not cause aborts due to its increment. In our evaluation, as the proposed hardware support is not available, we report a good estimate of the HTM-based performance (see details in Section 6.7).

5 LOCK-BASED DUDETX

Previous sections demonstrate that an out-of-the-box transactional memory can be integrated into our framework. This enables us to easily utilize all the advances of STM and HTM techniques. Meanwhile, using fine-grained locks often supports more concurrency and hence higher performance than transactional memory. In this section, we further explore the approach to fitting lock-based concurrency control into our decoupled framework. The challenge is that locks lead to a more complex dependency relationship among durable transactions. Nevertheless, since our framework elegantly decouples a durable transaction into three steps, such integration is well supported.

Next, we first introduce the background and a thorough example to explain the difficulties caused by lock-based concurrency control. Then, we describe the methods to overcome them in DudeTx, and show that using locks is as simple as replacing them with a wrapped lock implementation provided by us. Finally, we elaborate on the core design: how to detect dependencies among transactions.

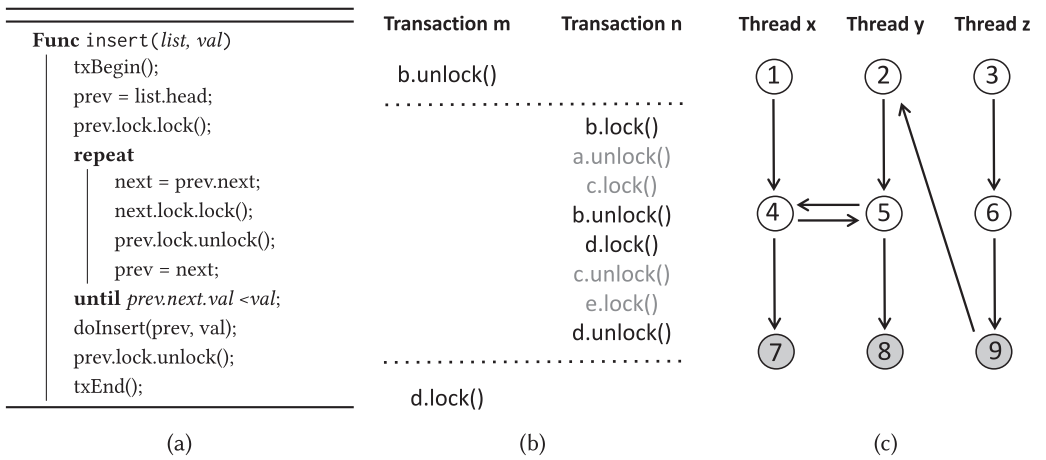
5.1 Background and Example

For a transactional system, the order to commit transactions is essential. In a TM system, the order is determined by a monotonic transaction ID assigned to each transaction. However, a lock-based concurrency control mechanism largely complicates the ordering problem. Such a problem and lock-based programming model are well discussed in Atlas [8, 9]. We borrow two important definitions.

Failure-Atomic SEction (FASE): In a single thread execution, a section is failure-atomic if the following conditions hold:

(1) The point just before the section holds no lock.
(2) The point just after the section holds no lock.
(3) Any point in the section holds at least one lock.

We have to maintain such a property: If any update within a FASE is durable, i.e., visible in NVM after a failure, then all updates within that FASE must be. The FASE in Atlas has the full ACID properties (Section 2.1). However, Atlas automatically builds FASE according to lock acquire and release operations, which limits programming flexibility. For example, users cannot specify two



```
Func insert(list, val)
    txBegin();
    prev = list.head;
    prev.lock.lock();
    repeat
        next = prev.next;
        next.lock.lock();
        prev.lock.unlock();
        prev = next;
    until prev.next.val <val;
    doInsert(prev, val);
    prev.lock.unlock();
    txEnd();
```

Fig. 2. An example of complex dependency relationships: (a) lock-based code to insert an item into an ordered link list; (b) a possible interleaving that results in circular dependency; (c) a possible dependency graph.

adjacent automatically identified FASEs to be atomically persisted in Atlas. Therefore, we offer an interface for programmers to specify the scope of a FASE by themselves. A section wrapped by `txBegin()` and `txEnd()` in DudeTx is a FASE. They are not enforced to align with lock operations. In our context, we continue to use the term transaction to refer to such a FASE.

Durability-Ordered Relationship: If the release operation of a FASE f_1 happens before an acquire operation of another FASE f_2, we say f_1 is durability-ordered before f_2 (donated by $f_1 <_d f_2$). Because operations after the acquire operation in f_2 depend on data modified by operations before the release operation in f_1, we can easily conclude that *if* $f_1 <_d f_2$, f_2 *should be durable only if* f_1 *is.* In this article, we briefly refer to this relationship as a *dependency*.

Based on the above concepts, we discuss an example to illustrate a typical case of complex dependency relationships in Figure 2. The example DudeTx code (a) is to insert an item to an ordered link list. The code contains a pair of `txBegin()` and `txEnd()` to wrap a transaction, and locks to protect access to a single node. The programmer can freely use lock and unlock functions at any point within the transaction (note that it is programmers' responsibility to avoid deadlock or double-entry bugs). Although this convenience enables users to design fine-grained locks to achieve best performance, it may introduce subtle interleavings that lead to complicated dependency relationships. For example, Figure 2(b) demonstrates a possible interleaving of two transactions m (T_m) and n (T_n) running code in (a). We assume a, b, . . . , e in the figure are locks of a sequence of nodes in the list. The two transactions make circular dependency: $T_m <_d T_n$ because of the lock b, while $T_n <_d T_m$ because of the lock d. As a result, it is *impossible* to decide a global order between these two transactions.

Atlas solves this problem by merging multiple FASEs into one. Specifically, if a cycle exists among FASEs, i.e., if $f_1 <_d f_2 <_d \cdots <_d f_1$, then all of the constituent FASEs must be durable if one of them is. For example, Figure 2(c) depicts a dependency graph, where each node represents a transaction and each arrow represents a dependency. In this figure, Transaction 4 and Transaction 5 circularly depend on each other. They should be considered as a single logical transaction to be committed in an atomic manner. Thus, the two transactions should be merged. Furthermore, the complexity grows as the graph expands with other dependency relationships. For example, Transaction 5 depends on Transaction 2. So, Transaction 4 and Transaction 5 can be committed only if Transaction 2 can be. In other words, Transaction 4 and Transaction 5 should both be rolled back if Transaction 2 is aborted (i.e., *cascading roll-back*).

5.2 Building Dependency Graph

In DudeTx, we provide wrapped lock/unlock functions that automatically generate the dependency graph on-the-fly. As Figure 2(a) already shows, programmers still need to use `txBegin` and `txEnd` to annotate a transaction, so that all updates within the transaction scope are persisted on NVM in an atomic manner. But using lock-based concurrency control with DudeTx is as simple as replacing the regular locks with the wrapped ones provided by DudeTx. Particularly, programmers only need to use the DudeTx wrappers `txLock` and `txUnlock` to replace the regular `lock` and `unlock` in the code, respectively. The wrappers add the functionality to record necessary information for constructing a dependency graph.

`txUnlock` attaches a metadata to the lock it releases. This metadata describes the transaction that executes this `txUnlock`. It includes the transaction's ID and the executing thread's ID. Based on the metadata, a following transaction that acquires the same lock can notice a dependency on the precedent transaction. This is done by `txLock`. Then, we can establish a dependency relationship between the two transactions.

The dependency graph shown in Figure 2(c), for example, is generated in the following steps: (1) When `txBegin` is called, a new node is added to the graph. (2) An execution of `txLock` will add an arrow to the current transaction from the last transaction that releases the lock. In addition to dependency relationships among transactions from different threads, we also add a dependency from a transaction x to a transaction y if y is executed immediately after x in the same thread.

5.3 Reproduce According to the Dependency Graph

As the dependency graph is generated on the fly, we can group the transactions/nodes into two categorizes: *(1)* volatile transactions, whose logs are not yet persisted; and *(2)* persistent transactions, whose logs are all persisted on NVM. We use gray nodes and white nodes to represent these two kinds, respectively. For example, in Figure 2(c), transactions 1, 2, 3, 4, 5, 6 are persistent transactions while transactions 7, 8, 9 are volatile transactions. Note that a volatile transaction can either already finish its execution in the shadow memory or be in the process.

But the difficulty here is that an individual persistent transaction is not always a *reproducible* transaction. A transaction is reproducible if it is safe to apply logs of the transaction to the original data. It is possible that a persistent transaction has to be rolled back because a transaction that it depends on is aborted or rolled back. Such a persistent transaction is not reproducible.

As described in Section 4.1, it is relatively easy to identify reproducible transactions in a TM setting. A persistent transaction is reproducible if its ID is smaller than the global durable ID. But, in a lock-based setting, there is no monotonic transaction ID that defines this global order. Instead, we can only determine a reproducible transaction from its relationships in the dependency graph: a transaction is reproducible if *(1)* it is a persistent transaction, and *(2)* all the transactions it depends on are reproducible.

In our implementation, a background thread continuously seeks for reproducible transactions according to the above definition. Once a reproducible transaction is identified, it is removed from the dependency graph and passed to the `Reproduce` thread.

Atlas provides a method to detect reproducible transactions in the dependency graph. It marks all persistent transactions as reproducible at first, and then gradually invalidates false marks. The invalidation procedure works by iteratively transferring a transaction's state from reproducible to non-reproducible if it relies on a volatile or non-reproducible transaction. For each transaction that is transferred from reproducible to non-reproducible, a depth first search (DFS) is performed to mark all transactions that depend on it as non-reproducible. After the procedure terminates, transactions that are still reproducible can go through `Reproduce`. Take Figure 2(c) as an example. Initially, all transactions are labeled as reproducible. Then, because Transaction 2 relies on a volatile transaction, its state is transferred to non-reproducible, which leads to a DFS that marks Transaction 4 as non-reproducible as well. In the end, only Transactions 1, 3, and 6 are still marked as reproducible. We name this method *two-round detecting*, because some transactions may be accessed twice. For example, after we have determined Transaction 4 as reproducible, a state update of Transaction 5 incurs another access to Transaction 4.

It is easy to understand the correctness of two-round detecting. If a persistent transaction x is not reproducible, there must be another transaction y that it directly or indirectly depends on that is not reproducible. Then there must be a dependency path from transaction y to transaction x. Meanwhile, if a transaction x is finally labeled as reproducible, all its direct and indirect depended transactions should also be reproducible, otherwise a DFS must have transferred transaction x to non-producible.

However, the performance of two-round detecting is low, because it has to scan all transactions in some cases. As we will see in Section 6.8, two-round detecting could be a bottleneck when the system is heavily loaded. Thus, we provide a more efficient method, *one-round detecting*.

One-round detecting works as long as there is no cycle in the dependency graph. Then we can scan all the nodes of a snapshot of the dependency graph in its **topological order**. If there is no cycle in a dependency graph, there must be at least one node without any precedent node. In a single thread, the first node must be the ancestor of other nodes (i.e., Transactions 1, 2, and 3 in Figure 2(c)). Thus, we can scan a single thread from its first transaction in the dependency graph; if it has no precedent node, we can mark it as reproducible and remove it from the graph. For example, in Figure 2(c), the detecting thread first checks Transactions 1 and 3, which do not depend on any volatile transactions, so they are marked as reproducible. Then the thread checks Transaction 6 which is also reproducible. However, Transaction 2 depends on Transaction 9 which is not persistent. Thus, only Transactions 1, 3, and 6 are reproducible. We call this method one-round detecting, as all transactions will be accessed at most once.

The problem of one-round detecting is that sometimes topological ordering is impossible. For example, in Figure 2(c), there is a cycle between Transactions 4 and 5. According to [7], circular dependency can be forbidden theoretically by restricting developers following a principle like two phase lock (2PL) to build strict transactions. In other words, if we restrict developers using techniques like 2PL, we can adopt one-round detecting to achieve better performance.

Thus, there is a tradeoff between two-round detecting and one-round detecting. The two-round detecting algorithm, as adopted by Atlas, allows programmers to introduce circular dependency, but incurs high detecting overhead. In contrast, we propose one-round detecting which relies on programmers to avoid circular dependency but offers fast detecting.

6 EVALUATION

6.1 Setup

Environment. We perform all our experiments on a 12-core Intel(R) Xeon(R) CPU E5-2643 v4 machine (3.4GHz, supporting RTM) with 64GB physical DRAM, running x86-64 Linux version 3.16.0 kernel. The results are the average of 10 runs. We use 1GB DRAM to emulate NVM and up to 1GB DRAM as shadow memory.

Persistent Memory Emulation. As real NVM is not yet available, we emulate persistent memory using DRAM. Similar to prior works [10, 24, 60], our method models the slow writes of persistent memory, and ignores the small additional latency of reads. Specifically, *(1)* if a single write to persistent memory is required to become persistent right away, we model the persist ordering overhead by adding a fixed extra delay. The write latency of PCM can be as large as 1μs [11, 20], hence we set the extra delay to 3,500 cycles, the same as prior works [24, 64]. In addition, we believe the write latency would drop in the future, so we also run experiments with the extra delay set to 1,000 cycles (about 300ns). The delay is realized by looping on the processor's timestamp counter via `RDTSC`. *(2)* If a sequence of writes are persisted together, we consider both the latency and bandwidth limit of persistent memory. Only one persist order is required for all these writes, so the extra delay is calculated by $\max\{latency, (total\ write\ size)/(NVM\ bandwidth)\}$. We test various bandwidth values in our experiments.

Benchmarks. We evaluate DudeTx with both micro benchmarks (HashTable and B+-Tree), and realistic workloads in TPC-C [58] and TATP [56].

HashTable benchmark is a simple fixed-size hash table that maps 64-bit integer keys to 64-bit integer values (randomly generated inputs, 100% insertion). In our implementation, hash collisions are resolved by circularly testing the next bucket. We guarantee the ACID of every operation by wrapping it in a transaction.

B+-Tree benchmark inserts randomly generated inputs to a B+-tree that maps 64-bit integer keys to 64-bit integer values. Similar to HashTable, we build a concurrent B+-tree by transferring a single-thread B+-tree with transaction APIs.

TPC-C [58] is a well-known online transaction processing (OLTP) benchmark. We implement the New Order transaction of it, which simulates a customer buying different items from a local warehouse. The transaction is write-intensive and requires atomic updates to different tables. We implement the TPC-C with both B+-tree and hash table as its table storage. An array of B+-trees or hash tables are used to represent a table with a compound key. As our implementation is identical to [10], we omit the details.

TATP [56] is a benchmark that models a mobile carrier database. We implement the Update Location transaction of it, which records the handoff of a user from one cell tower to another. The transaction is much shorter than New Order in TPC-C, because there is only one search and one update in it. Similar to TPC-C, we also implement both a B+-tree version and a hash table version of TATP.

Evaluated Systems. Our evaluation involves the following four systems.

(1) Volatile-STM, the regular TinySTM running on DRAM without the durability guarantee and no delay is added. This gives the theoretical performance upper bound of DudeTx based on TinySTM.

(2) DudeTx, the standard asynchronous implementation based our decoupled framework. The capacity of its log buffers is 1 million log entries per thread. If the log buffer is full, the `Perform` step has to block and wait for `Persist` to flush more logs to persistent memory and release space.

(3) DudeTx-*Inf*, an asynchronous implementation of DudeTx that has infinite log buffers. In this case, `Perform` never has to block.

(4) DudeTx-*Sync*, a synchronous implementation of DudeTx that immediately flushes logs to persistent memory after *Perform* step. The transaction returns after it becomes persistent. In other words, the first and second steps of DudeTx are merged and transactions cannot be executed back-to-back.

6.2 Throughput

6.2.1 Performance Analysis. Theoretically, if ordered by throughput from high to low, the above four systems should be Volatile-STM, DudeTx-Inf, DudeTx, and DudeTx-Sync. The throughput differences between them represent the overhead of log generation, log buffer saturation, and log persistence, respectively. To measure these overheads, we evaluated the four systems using the

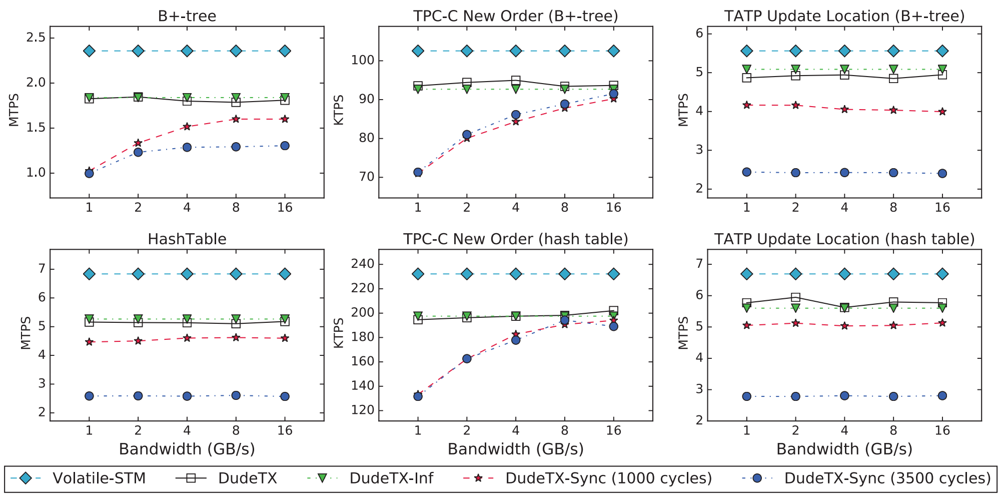


Fig. 3. Throughput of evaluated systems with different NVM bandwidth.

benchmarks described in Section 6, with various persistent memory bandwidth, from 1GB/s to 16GB/s. Because the latency only affects DudeTx-Sync, we only use 1,000 cycles latency for other systems. Figure 3 presents the results, from which we can derive the following three findings.

Finding (1): DudeTx incurs minor overhead (7.4%–24.6% less throughput) over the original volatile STM.

As expected, Volatile-STM provides the highest throughput among all systems under all benchmarks. Based on TinySTM, DudeTx adds a small performance penalty to the original STM. The throughput of DudeTx is only 7.4% (B+-tree based TPC-C) to 24.6% (HashTable) less than that of Volatile-STM. The different performance penalties among benchmarks are due to the different write intensity. Since the main overhead introduced in DudeTx is log generation, a benchmark that has higher write intensity typically suffers from a higher overhead. For example, as the major operations of a hash table insertion are memory writes and the HashTable benchmark consists of 100% insertions, HashTable is the most write-intensive benchmark and hence sees the largest performance penalty.

Finding (2): Log flushing is not the bottleneck of the decoupled framework in DudeTx.

We see that DudeTx-Inf produces almost the same throughput as DudeTx, which means that the `Perform` step of a transaction is rarely blocked by a full log buffer. More importantly, the observation applies to not only 16GB/s NVM but also 1GB/s NVM. For more direct evidence, Table 1 shows the number of memory writes executed by each benchmark when the NVM bandwidth is set to 1GB/s and latency is 1,000 cycles. We can calculate that around 78MB (TATP based on B+-tree) to 488MB (TPC-C based on hash table) redo logs are generated per second, which are less then the bandwidth of persistent memory.

Finding (3): Decoupling enables high performance of DudeTx and avoids the bottleneck in log flushing.

Although flushing is not a bottleneck with decoupling, it does impact the performance if a transaction immediately flushes its log to persistent memory after it is performed and blocks until the log is persisted, especially when the bandwidth of persistent memory is low (e.g., 1GB/s). We can see that the throughput is gradually improved with the increase of persistent memory

Table 1. Statistics of Memory Writes in Different Benchmarks (1GB/s NVM Bandwidth, 1,000 Cycles Latency, Four 4 Threads)

Benchmark	# writes	Throughput	# writes per tx
B+-tree	28.9M/s	1.83 MTPS	15.8
TPC-C (B+-tree)	17.2M/s	93.5 KTPS	183.5
TATP (B+-tree)	4.87M/s	4.87 MTPS	1.0
HashTable	15.5M/s	5.16 MTPS	3.0
TPC-C (hash)	30.5M/s	195 KTPS	156.5
TATP (hash)	5.77M/s	5.77 MTPS	1.0

"M/s" stands for millions per second.

bandwidth, until the latency of persistent memory becomes the new bottleneck. When the bandwidth is beyond 8GB/s, the throughput of most benchmarks is hardly improved by higher bandwidth. However, for transactions that have a lot of instructions (e.g., a TPC-C transaction costs about 110k cycles), the latency of persistent memory (i.e., 3,500 cycles) is negligible compared to the whole execution time. As a result, for those benchmarks, the performance gap between DudeTx and DudeTx-Sync is small. On the contrary, for transactions that have fewer instructions (e.g., a TATP transaction costs about 3,000 cycles), they show a clear performance decline from DudeTx to DudeTx-Sync, when latency is increased from 1,000 cycles to 3,500 cycles.

6.2.2 Comparison to Current Systems. Mnemosyne [60] is a durable transaction library for persistent memory, with the same guarantees as DudeTx. It uses Intel's STM compiler to instrument memory reads/writes and TinySTM to manage transactions. Mnemosyne uses TinySTM's write-back access scheme (redo logging) rather than the write-through scheme (undo logging) as in DudeTx. Since it does not follow our decoupling approach, Mnemosyne faces the tradeoff as described in Section 2.2, i.e., a redo logging scheme requires less fences but costly address mapping.

We also evaluate NVML [28], an undo logging based persistent transactional memory library developed by Intel. NVML requires that users have prior knowledge of the memory write set of a transaction, which means that it supports only static transactions. As NVML transactions do not guarantee the isolation property, users of NVML need to use separate concurrency control mechanisms (e.g., locking). Accordingly, we implement a hash table using fine-grained locks with NVML, but the complex changes leading to a high-performance lock-based concurrent B+-tree would make the comparison with other systems unfair. Therefore, we only run the hash table based benchmarks over NVML. In our evaluation, we move all the memory allocation operations to the beginning of the program, so that our comparison is focused on transaction execution excluding the slow NVML allocation.

Table 2 shows the result. As we can see, DudeTx and DudeTx-Sync are about 1.7×–4.4× and 1.3×–3.0× faster, respectively, than Mnemosyne and NVML running various benchmarks. Mnemosyne is slow for the following reasons: (1) Intel STM Compiler instruments all the memory accesses of every transaction, resulting in a noticeable performance degradation [51]; (2) Mnemosyne needs to use `CLFLUSH`, which invalidates the cache line and hence increases cache misses. This problem is avoided in DudeTx because the shadow memory is resident in DRAM; (3) Each transaction requires a costly synchronous persist, which is similar to DudeTx-Sync. Finally, (4) the address mapping overhead of redo logging. In contrast, although NVML also suffers from the cache invalidation and persist issues, it avoids excessive instrumentation and address mapping by asking users to manually annotate NVM writes and by using undo logging, respectively. NVML's specific implementation details also affect its performance. For example, NVML

Table 2. Throughput of DudeTx, DudeTx-Sync, Mnemosyne, and NVML (1GB/s NVM Bandwidth, 1,000 Cycles Latency, Four Threads)

Benchmark	DudeTx	DudeTx-Sync	Mnemosyne	NVML
B+-tree/MTPS	1.83	1.02	0.77	-
TPC-C(B+-tree)/KTPS	93.5	71.0	42.1	-
TATP(B+-tree)/MTPS	4.87	4.16	2.81	-
HashTable/MTPS	5.16	4.47	1.95	2.04
TPC-C(hash)/KTPS	195	133	76.6	44.7
TATP(hash)/MTPS	5.77	5.05	2.56	2.70

Table 3. Durable Transaction Latency of the Hash Table Based TPC-C Benchmark in DudeTx, DudeTx-Sync, Mnemosyne, and NVML

Percentage	DudeTx	DudeTx-Sync	Mnemosyne	NVML
50%	45μs	18μs	62μs	112μs
90%	73μs	40μs	87μs	161μs
99%	124μs	90μs	126μs	254μs

dynamically allocates transaction metadata and undo logs for each transaction, which is very expensive. In DudeTx and Mnemosyne, both the metadata and log buffers are allocated collectively on thread creation. As a result, NVML can only run at most 1.14 million empty transactions per second per thread. In contrast, the maximum throughput of running empty transaction on DudeTx/Mnemosyne is 30+ millions per second.

6.3 Latency

The transaction implemented in a decoupled manner can return to users immediately after the `Perform` step but, at that point, the transaction is not durable yet. For applications that require an explicit acknowledgement of the durability of a transaction, users of DudeTx can periodically inquire the global latest durable transaction ID as mentioned in Section 3.3. Particularly, an application thread can work this way: executing a transaction of ID t_i; getting the durable ID d_i and acknowledging transactions whose ID $\leq d_i$; executing a transaction of ID t_{i+1}; getting the durable ID d_{i+1} and acknowledging transactions whose ID $\leq d_{i+1}, \ldots$. In certain cases, the *latency* of a transaction (measured by the time between the beginning of the transaction and its durability acknowledgment) may increase in DudeTx because `Persist` is done asynchronously. In this section, we present our evaluation results of this latency by comparing DudeTx, DudeTx-Sync, Mnemosyne, and NVML.

Table 3 presents the distribution of latency (e.g., in DudeTx, 50% of the transactions can be durable within 45μs). We see that decoupling only introduces a moderate extra latency compared to DudeTx-Sync. The extra latency is mainly due to the fact that we do not check the latest durable transaction ID in the middle of a transaction. According to our evaluation, about 99% of the transactions can be persisted before the end of their next transaction's `Perform`. It means that in most cases, the background `Persist` thread can finish flushing the log of a transaction during the `Perform` step of the next transaction. Therefore, the latency of DudeTx is about 2× the ideal latency that is 1/(throughput of DudeTx). Because DudeTx's throughput is more than 2× that of Mnemosyne and NVML, DudeTx has an even better latency performance than those existing synchronous durable transaction solutions. Compared to them, DudeTx achieves *both* higher throughput *and* lower latency.

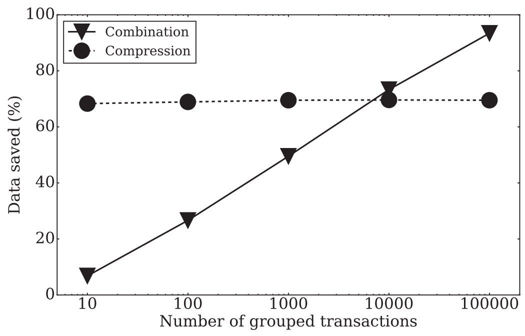


Fig. 4. Log optimization.

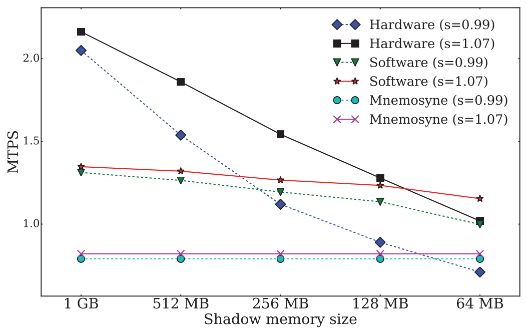


Fig. 5. Swap overhead.

6.4 Log Optimization

The decoupled framework enables the log optimization techniques before the log is persisted. The effect of log combination is determined by the skewness of workload. In one extreme, if all the writes access the same address, all but the last one can be omitted. In reality, according to the "power law" [4, 17], many real-world workloads are indeed skewed. In this experiment, we use the Session Store workload of YCSB [16] and run it on a B+-tree based key-value store. The store is loaded with 10K records, and the ratio of read/update transactions is 50:50. Transactions follow the Zipfian distribution with a constant of 0.99.

Figure 4 presents our evaluation results. We see that a higher optimization ratio can be obtained if the transactions are grouped to persist. Around 7% NVM writes can be saved if the log combination is performed for every 10 transactions, and the ratio increased to 93% when each group contains 100,000 transactions. Figure 4 also shows the result of using lz4 [15] to compress logs. It can stably achieve a compression ratio as high as 69% even when applying to only 10 transaction groups. However, log compression per se can only reduce NVM writes in `Persist`. `Reproduce` still needs to execute the same number of writes as `Perform`.

DudeTx allows users to explore the tradeoff in optimization—a larger group of transactions means a higher latency and more memory usage, but leads to less NVM write traffic. Besides, we find that log optimization typically has no influence on the throughput of applications (unless the size of each group is too large). This is because that log flushing is not the bottleneck of the system, i.e., Finding (2). However, the latency of our system is proportional to the number of grouped transactions, as a transaction has to wait for other transactions in the same group to persist.

6.5 Swapping Overhead

We evaluate the overhead of swapping in/out when the size of shadow memory is smaller than NVM. Overall overhead of swapping largely depends on its frequency, which is determined by (1) the ratio of shadow memory to NVM and (2) how skewed the transactional access is. Figure 5 depicts the throughput of updating a B+-tree based key-value store. The workloads are generated by Zipfian distribution with two constants, 0.99 and 1.07. We use 1GB NVM and a shadow DRAM that varies from 64MB to 1GB. The total working set of each workload is around 650MB. The figure indicates that the throughput decreases as the size of shadow memory shrinks or when the workload becomes less skewed.

Our software-based and hardware-based implementations have different sensitivity over the shadow memory size. Figure 5 shows that hardware-based paging has better performance than software-based paging when the shadow memory is relatively large, but its performance drops

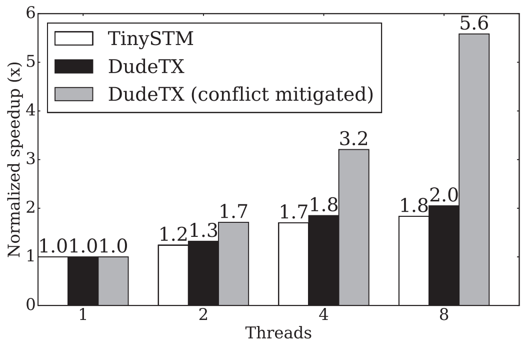


Fig. 6. Scalability.

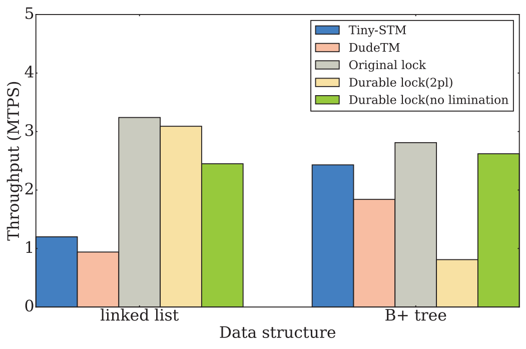


Fig. 7. Lock-based DudeTx (four threads).

more quickly as the size of shadow memory decreases. The reason is that hardware-based paging has less overhead in address translation due to use of TLB, but more overhead in page fault and swapping mainly because it has to do a VM exit to shoot down invalid TLB entries when it evicts a shadow page. (This is due to our implementation with Dune, and it can be improved in future work.) On the contrary, software-based paging does not involve TLB shootdown but incurs more overhead in address translation (at least two memory accesses per address translation). It turns out that modification of page references, though by a simple compare-and-swap instruction, is costly too, as multiple cores visit the same page frequently.

NVM is considered to have higher density than DRAM. But in works [19, 42] which gave the exact data, density of NVM is not more than 4× of that of DRAM. In this case, hardware-based implementation is still better than others. Even in worse case, e.g., when DRAM size is only $\frac{1}{16}$ of NVM size, we can choose software-based implementations.

6.6 Scalability

Figure 6 presents our results on the scalability of DudeTx. It shows the throughput of running B+-tree based TPC-C with different numbers of threads. The NVM bandwidth is set to 1GB/s and the latency is 1,000 cycles (actually they have negligible impact on results). The results are normalized to the throughput of one thread. We see that our TinySTM-based DudeTx implementation achieves a similar speedup as TinySTM itself. DudeTx has even a little better speedup, because its throughput of one thread is slower than TinySTM.

The scalability bottleneck of DudeTx lies in the concurrency control mechanism of TinySTM, instead of other parts of DudeTx. To manifest the bottleneck, we implement another version of TPC-C with less conflicts, where each thread serves customer requests for a fixed district. As each B+-tree in Order tables is responsible for a certain district, this eliminates most conflicts in concurrency control. Therefore, the bottleneck in TinySTM is avoided. We can see from Figure 6 that scalability of DudeTx shows almost linear scalability in this case. In other words, DudeTx-specific overhead has little influence on scalability.

6.7 HTM-Based DudeTx

As we discussed in Section 4.2, we can only **estimate** the possible speedup of using HTM by generating the transaction ID with atomic operations that are *not* wrapped in transactions. Such methodology is reasonable because, although the order of transactions may not be accurate, it does not affect the performance evaluation. Moreover, in our implementation, if a HTM transaction fails more than five times, a fallback routine is called to execute the transaction using a global lock.

Table 4. Throughput of DudeTx Based on STM and HTM (1GB/s NVM Bandwidth, 1,000 Cycles Latency, Four Threads)

	B+-Tree	HashTable	TATP (B+-tree)
Volatile-STM/MTPS	2.36	6.84	6.69
DudeTx-STM/MTPS	1.83	5.16	5.77
Slowdown	22%	26%	14%
Volatile-HTM/MTPS	3.59	11.8	6.96
DudeTx-HTM/MTPS	3.21	7.47	6.50
Slowdown	11%	28%	7%

Table 4 shows the throughput of HTM-based DudeTx. TPC-C is not shown because its transaction issues such a large write set that Intel Haswell's HTM cannot handle it. It achieves up to 1.7× higher throughput than the STM-based implementation, especially when applications are more complicated. Among all benchmarks, B+-Tree shows the largest speedup. This is because a transaction in this benchmark is bigger than other benchmarks and conflict management in HTM is more effective than STM. In contrast, TATP has less speedup (about 1.33×) because there is only one concurrent write in a transaction, which means that most of the execution time is spent in local reads that cannot be improved by replacing STM with HTM. The same reason applies to the HashTable benchmark in which aborted transactions are less than 0.3%. In addition, HashTable has a very high write ratio (72%) and inserting logs for each write has a great influence on performance. That also results in higher overhead on pure HTM. However, the overhead of DudeTx is still less than 28% for both STM and HTM. That means our decoupled framework is compatible and effective to many kinds of transactional memory technologies.

6.8 Lock-Based DudeTx

In this section, we discuss the performance of lock-based DudeTx and two dependence graph detecting algorithms. To evaluate their performance, we implement two popular data structures: linked list and B+-tree. For linked list, we use optimistic concurrent control to decrease the use of lock. Specifically, it does not acquire any lock when searching from the head of the list, until it finds the correct nodes. Then it would check if the node is modified during acquiring the lock. It would abort this transaction and retry if the node has been modified by other transactions. We use the 2PL scheduler when acquiring locks so that *one-round detecting* and *two-round detecting* are both suitable here. For B+-tree, we used techniques described in [63], which is against the 2PL rule, i.e., it acquires and releases locks with no limitation. Thus, we implement another B+-tree with 2PL, which holds all locks of nodes on the path from the root to the leaf, and releases them only after finishing all operations.

We test the durable lock-based program in DudeTx, using *one-round detecting* and *two-round detecting*, respectively. Volatile TM-based, durable TM-based, and volatile lock-based programs are also evaluated as comparison. Figure 7 presents our evaluation results. As we can see from the figure, first, lock-based programs are faster than the TM-based programs, especially for linked list. It is because TM has to wrap each read and write operation in a transaction, which has a much higher overhead than accessing the data directly. On the contrary, a lock-based program only needs to lock the special data.

Secondly, *two-round detecting* may be the bottleneck of the system. If the buffer for storing dependency graph is full, the worker threads have to stop and wait for detect thread. In linked list benchmark, detecting dependency graph is slower than transaction execution. Thus, *two-round* *detecting* blocks the system. On the contrary, *one-round detecting* has nearly no overhead and has almost the same performance as the original volatile lock-based program.

Thirdly, 2PL has many limitations. In the B+-tree benchmark, a 2PL program has to hold all locks from the root to the leaf. Threads are all blocked on the lock of the root node. A multi-thread program is not better than a single thread program (maybe even worse because of context switch). As Figure 7 shows, in B+tree experiment, the 2PL program is much worse than others, even TM-based programs. On the contrary, total throughput is still in the control of *two-round detecting*, thus the *two-round detecting* program has the near performance of the volatile lock-based program. Thus, it is necessary to provide two kinds of detecting algorithms for users so that they can choose a better one according to their own situations.

7 RELATED WORK

We first distinguish DudeTx from traditional transactional memory work, and compare it with other persistent memory systems. They can be classified by a taxonomy of crash consistency protection methods [49]: persistent locks work with memory-access-level protection; durable transactions protect code blocks; NVM-oriented data structures offer object-based interfaces; hardware solutions typically achieve software-transparent program-level protection. Finally, we also discuss related work on file systems and databases.

Transactional Memory. The traditional concept of transactional memory is about concurrency control, having no durability guarantee [21, 22, 26, 43]. DudeTx relies on transactional memory to perform concurrent transactions and detect conflicts. However, our focus is not on optimizing software transactional memory [68] or hardware transactional memory [65, 66]. Instead, we solve the problem of how to efficiently make transactions durable on persistent memory.

Persistent Lock Programming Model. Atlas [8, 9] offers a lock-based programming model for NVM. However, it is based on undo logging, and limited by the persist ordering overhead. In contrast, making best use of both undo and redo logging is the fundamental contribution of this article. Moreover, Atlas automatically builds FASEs according to lock acquire and release operations. It is not as flexible as DudeTx which does not necessarily align FASEs with lock operations. Finally, Atlas uses two-round detecting (Section 5.3), while DudeTx incorporates a faster one-round detecting algorithm.

Durable Transaction Systems. Previous NVM-based durable transaction systems [14, 24, 34, 60] usually suffer from a dilemma between per-update persist ordering and update redirection overhead. DCT [34] and NVML [28] bypass this issue by allowing only static transactions. They also require that all locks should be acquired at transaction start and released after transaction commit. Those limitations result in less concurrency and inferior performance [47]. Mnemosyne [60] designs an alternative asynchronous reproducing method that replays redo logs on background, similar to the `Reproduce` step in DudeTx. However, it does not decouple `Perform` and `Persist` steps as DudeTx does. It still does not solve the dilemma, and does not perform well when the write latency of persistent memory is large.

SoftWrAP and DudeTx both use the concept of *shadow memory*, but they are fundamentally different. First, SoftWrAP still has to do object-level address mapping. Due to its "double-buffered" alias table, SoftWrAP needs up to three times of indirection for reading a value. Second, changing SoftWrAP's mapping granularity to reduce its indirection overhead is not easy. In SoftWrAP's implementation, a page-level mapping would bring excessive write amplification overhead. Meanwhile, a straightforward one-to-one mapping would result in both unacceptable memory space consumption (>3× due to the use of two alias tables) and increased execution time (due to tracking modified memory to avoid scanning the whole alias table for dumping). Third, SoftWrAP argues that concurrency control can be decoupled from durable transactions, but it requires non-trivial modifications. It claims that isolation can be implemented by using local alias table, but it is not clear how to automatically merge it into global alias table. Moreover, dedicated mechanisms are required to find a quiescent point that SoftWrAP can safely switch alias table from active to closed, which would incur further delay and complexity. According to our investigation, the reason why SoftWrAP suffers from the above disadvantages is that SoftWrAP tries to directly copy data from shadow memory to NVM. In DudeTx, copying data from the per-thread redo log can fundamentally avoid these problems.

LOC [41] employs eager commit and speculative persistence to reduce persist ordering overhead in redo logging. BPPM [40] uses execution-in-log (EIL) to save one copy of log, and volatile checkpoint with bulk persistence (VCBP) to reduce persistence overhead due to checkpointing. Log-structured NVMM [27] also reduces persist ordering and storage cost of redo logging. Different from these works, DudeTx focuses on eliminating address mapping overhead of redo logging.

NVM-Oriented Data Structures. Prior works [11, 12, 18, 66, 70] optimize implementation of certain data structures on NVM. For example, Yang et al. [70] design a high-performance persistent B+-tree, which reduces the number of NVM writes by disordering keys in leaf nodes. Moreover, PCM-DB [11] proposes two optimizations on B+-tree and hash joins for database on PCM. Different from those works that deal with specific data structures, DudeTx aims to provide a general transactional library for wider use scenarios (e.g., mingling of different data structures).

Software-Transparent Solutions. Other work increases performance of persistent-memory applications or simplifies their programming by hardware support. WSP [44] achieves so by using residual power to flush data in CPU caches to NVM upon caches. ThyNVM [51] relies on an efficient checkpointing design in the memory controller to realize software-transparent crash consistency protection. DudeTx assumes no essential hardware modifications.

File Systems. The file system community has studied crash consistency for a long time. OptFS [13] decouples consistency and durability, and LightTx [39] decouples concurrency control and crash consistency. Such decoupling has a different purpose or method with DudeTx, as file systems and durable transactions face different challenges. For example, LightTx does not have to deal with address mapping overhead, because the flush translation layer (FTL) in solid state disk (SSD) helps translate address in coarse granularity. For SSDs, the address translation overhead is not a critical issue as it is far less than that of disk accesses. On the contrary, accessing data in NVM is much faster so that the address mapping latency becomes noticeable. Actually, "shadow memory" used in certain file systems corresponds to the persistent log in DudeTx, while the shadow memory in DudeTx has a totally different meaning.

Database Systems. In-memory databases [33, 59, 65] have to flush logs to disks to provide ACID guarantees. These systems may achieve a better throughput than TM systems, but their data can only be accessed through specific database operations. In contrast, the TM interface is more flexible. Moreover, certain databases have extra constraints. For example, FOEDUS [33] uses a dual-page technique so that data must fit into fixed-size pages, and hence is not efficient or directly usable for durable transactions on persistent memory.

8 CONCLUSION

This article presents DudeTx, a decoupled framework to implement atomic, durable transactions on persistent memory. DudeTx avoids the inefficiencies of traditional undo logging and redo logging based techniques. Its key design is to decouple an ACID transaction into three asynchronous steps, which enable us to run an out-of-the-box concurrency control mechanism on the shadow memory as a stand-alone component in our system, including STM, HTM, and fine-grained lock. Our evaluation results show that DudeTx adds guarantees of crash consistency and durability to TinySTM by adding only 7.4%–24.6% overhead, and is 1.7× to 4.4× faster than existing works

Mnemosyne and NVML. Using HTM and fine-grained lock leads to another 1.7× and 3.3× speedup, respectively. Through decoupling, we have also enabled the possibility of reducing the write traffic to NVM by log optimization up to 93% reduction.

REFERENCES

[1] H. Akinaga, and H. Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010).

[2] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi. 2013. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM J. Emerg. Technol. Comput. Syst.* 9, 2 (May 2013), 13:1–13:35.

[3] J. Arulraj, A. Pavlo, and S. R. Dulloor. 2015. Let's talk about storage and recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*, pp. 707–722.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*, pp. 53–64.

[5] G. Atwood. 2011. Current and emerging memory technology landscape. *Flash Memory Summit*, 9–11.

[6] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, pp. 335–348. https://github.com/ix-project/dune.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. 1986.*Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

[8] H.-J. Boehm, and D. R. Chakrabarti, Persistence programming models for non-volatile memory. 2016. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ACM, pp. 55–67.

[9] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (*OOPSLA'14*). ACM, New York, pp. 433–452.

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

[11] S. Chen, P. B. Gibbons, and S. Nath. 2011. Rethinking database algorithms for phase change memory. In *Online Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (ICIDR'11)*, pp. 21–31.

[12] P. Chi, W.-C. Lee, and Y. Xie. 2014. Making B+-tree efficient in PCM-based main memory. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*. ACM, pp. 69–74.

[13] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2013. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, pp. 228–243.

[14] 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. *ACM Sigplan Notices*, 46 3 (2011), 105–118.

[15] Y. Collet. 2013. Lz4: Extremely fast compression algorithm. *code. google. com.*

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, pp. 143–154.

[17] C. Cunha, A. Bestavros, and M. Crovella. 1995.*Characteristics of WWW Client-Based Traces*. Computer Science Department Technical Report BU-CS-95-010, Boston University.

[18] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. 2015. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, pp. 54–70.

[19] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. 2014. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*. ACM, New York, pp. 15:1–15:15.

[20] S. Eilert, M. Leinwander, and G. Crisenza. 2009. Phase change memory: A new memory enables new memory usage models. In *Proceedings of the 2009 IEEE International Memory Workshop*. IEEE, pp. 1–2.

[21] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. 2010. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems* 21, 12 (2010), 1793–1807.

[22] P. Felber, C. Fetzer, and T. Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, pp. 237–246.

[23] R. F. Freitas, and W. W. Wilcke. 2008. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development* 52, 4/5 (2008), 439.

[24] E. R. Giles, K. Doshi, and P. Varman. 2015. Softwrap: A lightweight framework for transactional support of storage class memory. In *Proceedings of the 2015 31st Symposium on Mass Storage Systems and Technologies (MSST'15)*. IEEE, pp. 1–14.

[25] T. Haerder, and A. Reuter. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)* 15, 4 (1983), 287–317.

[26] M. Herlihy, and J. E. B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, pp. 289–300.

[27] Q. Hu, J. Ren, A. Badam, and T. Moscibroda. 2017. Log-structured non-volatile main memory. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'17)*, pp. 703–717.

[28] Intel. NVM Library. 2014. Retrieved July 2016 from https://github.com/pmem/nvml.

[29] Intel. 2012. Architecture instruction set extensions programming reference. Intel Corporation (Feb. 2012).

[30] Intel Corp. 2015. Intel and Micron Produce Breakthrough Memory Technology.

[31] International Technology Roadmap for Semiconductors (ITRS). 2011. Process, integration, devices and structures. Retrieved July 2016 from http://www.itrs.net/Links/2011ITRS/2011Chapters/2011PIDS.pdf.

[32] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. 2010. Aether: A scalable approach to logging. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 681–692.

[33] H. Kimura. 2015. FOEDUS: OLTP engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, pp. 691–706.

[34] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. 2016. High-performance transactions for persistent memories. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, pp. 399–411.

[35] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceeding of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*, pp. 256–267.

[36] B. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. 2010. Phase-change technology and the future of main memory. *IEEE Micro* 30 (Jan. 2010), 131–141.

[37] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pp. 1–13.

[38] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, and J. Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. ACM, pp. 329–343.

[39] Y. Lu, J. Shu, J. Guo, S. Li, and O. Mutlu. 2013. LightTx: A lightweight transactional design in flash-based SSDS to support flexible transactions. In *Proceedings of the 2013 IEEE 31st International Conference on Computer Design (ICCD'13)*. IEEE, pp. 115–122.

[40] Y. Lu, J. Shu, and L. Sun. 2015. Blurred persistence in transactional persistent memory. In *Proceedings of the 2015 31st Symposium on Mass Storage Systems and Technologies (MSST'15)*. IEEE, pp. 1–13.

[41] Y. Lu, J. Shu, L. Sun, and O. Mutlu. 2014. Loose-ordering consistency for persistent memory. In *Proceedings of the 2014 32nd IEEE International Conference on Computer Design (ICCD'14)* . IEEE, pp. 216–223.

[42] S. Mittal, and J. S. Vetter. 2016. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27, 5 (2016), 1537–1550.

[43] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, D. A. Wood, et al. 2006. LogTM: Log-based transactional memory. In *Proceedings of HPCA*, Vol. 6, pp. 254–265.

[44] D. Narayanan, and O. Hodson. 2012. Whole-system persistence. *ACM SIGARCH Computer Architecture News* 40, 1 (2012), 401–410.

[45] S. Pelley, P. M. Chen, and T. F. Wenisch. 2014. Memory persistency. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 265–276.

[46] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*, pp. 24–33.

[47] H. E. Ramadan, C. J. Rossbach, and E. Witchel. 2008. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, pp. 246–257.

[48] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4 (July 2008), 465–479.

[49] J. Ren, Q. Hu, S. Khan, and T. Moscibroda. 2017. Programming for non-volatile main memory is hard. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys'17)*. ACM, pp. 13:1–13:8.
[50] J. Ren, C.-J. M. Liang, Y. Wu, and T. Moscibroda. 2015. Memory-centric data storage for mobile systems. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*, pp. 599–611.
[51] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu. 2015. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, pp. 672–685. http://persper.com/thynvm/.
[52] T. Riegel, C. Fetzer, and P. Felber. 2007. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*. ACM, pp. 221–228.
[53] M. Rosenblum, and J. K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems* 10, 1 (Feb. 1992), 26–52.
[54] A. Rudoff. 2016. Deprecating the PCOMMIT instruction. Retrieved September 2016 from https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.
[55] S. M. Rumble, A. Kejriwal, and J. Ousterhout. 2014. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, pp. 1–16.
[56] N. Simo, W. Antoni, M. Markk, and R. Vilho. 2011. Telecom application transaction processing benchmark. Retrieved July 2016 from http://tatpbenchmark.sourceforge.net/.
[57] K. Suzuki, and S. Swanson. 2015. A survey of trends in non-volatile memory technologies: 2000-2014. In *Proceedings of the 2015 IEEE International Memory Workshop (IMW)*. IEEE, pp. 1–4.
[58] THE TRANSACTION PROCESSING COUNCIL. 2010. TPC-C Benchmark V5. Retrieved July 2016 from http://www.tpc.org/tpcc/.
[59] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, pp. 18–32.
[60] H. Volos, A. J. Tack, and M. M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*, pp. 91–104.
[61] H. Wan, Y. Lu, Y. Xu, and J. Shu. 2016. Empirical study of redo and undo logging in persistent memory. In *Proceedings of the 2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 1–6.
[62] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and Adl-A.-R. Tabatabai. 2007. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*, pp. 34–48.
[63] P. Wang. 1991. *An In-Depth Analysis of Concurrent B-tree Algorithms*. MIT Cambridge Lab for Computer Science Technical Report, Massachusetts Institute of Technology, Cambridge, MA.
[64] T. Wang, and R. Johnson. 2014. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment* 7, 10 (2014), 865–876.
[65] Z. Wang, H. Qian, J. Li, and H. Chen. 2014. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the 9th European Conference on Computer Systems*. ACM, p. 26.
[66] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, pp. 87–104.
[67] M. Wu, and W. Zwaenepoel, eNVy: A non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pp. 86–97.
[68] P. Wu, M. M. Michael, C. von Praun, T. Nakaike, R. Bordawekar, H. W. Cain, C. Cascaval, S. Chatterjee, S. Chiras, R. Hou, et al. 2009. Compiler and runtime techniques for software transactional memory optimization. *Concurrency and Computation: Practice and Experience* 21, 1 (2009), 7–23.
[69] J. Xu, and S. Swanson, NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16)*, pp. 323–338.
[70] 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 *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, pp. 167–181.
[71] J. H. Yoon, H. C. Hunter, and G. A. Tressler. 2013. Flash & DRAM Si scaling challenges, emerging non-volatile memory technology enablement—Implications to enterprise storage and server compute systems. *Flash Memory Summit*.
[72] Y. Zhang, and S. Swanson. 2015. A study of application performance with non-volatile main memory. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST'15)*, pp. 1–10.
[73] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, pp. 421–432.

Received February 2017; revised September 2017; accepted January 2018