DudeTM: Building Durable Transactions with Decoupling for Persistent Memory

Mengxing Liu* Mingxing Zhang* Kang Chen* Xuehai Qian∓
Yongwei Wu* Jinglei Ren†
*Tsinghua University ∓University of Southern California †Microsoft Research

Abstract

Emerging non-volatile memory (NVM) offers non-volatility, byte-addressability and fast access at the same time. To make the best use of these properties, it has been shown by empirical evidence that programs should access NVM directly through CPU load and store instructions, so that the overhead of a traditional file system or database can be avoided. Thus, durable transactions become a common choice of applications for accessing persistent memory data in a crash consistent manner. 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.

This paper presents DudeTM, a crash-consistent durable transaction system that avoids the drawbacks of both undo logging and redo logging. DudeTM 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 also enables an out-of-the-box transactional memory (TM) to be used as an independent component in our system. The evaluation results show that DudeTM adds durability to a TM system with only $7.4\% \sim 24.6\%$ throughput degradation. Compared to the existing durable transaction systems, DudeTM provides $1.7\times$ to $4.4\times$ higher throughput. Moreover, DudeTM can be implemented with existing hardware TMs with minor hardware modifications, leading to a further $1.7\times$ speedup.

* Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China; Research Institute of Tsinghua University in Shenzhen, Guangdong 518057, China.

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

ASPLOS '17, April 08-12, 2017, Xi'an, China
© 2017 ACM. ISBN 978-1-4503-4465-4/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3037697.3037714

1. Introduction

Emerging non-volatile memory (NVM) technologies [5, 25, 56] are promising because they offer non-volatility, byte-addressability and fast access at the same time. Phase change memory (PCM) [30, 36], spin-transfer torque RAM (STT-RAM) [2, 29] 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 [24]. As NVM enjoys DRAM-level access latency, empirical studies [10, 19, 28, 47, 57, 58] suggest that NVM should be directly accessed through the processor-memory bus by load/store instructions, making a *persistent memory*, to avoid the overhead of the legacy block-oriented file systems or databases. Persistent memory also allows programmers to update persistent data structures at byte level without the need for serialization.

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* [33, 47]. A solution to this problem is using crash-consistent *durable transaction*, which makes a group of persistent 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 [19, 22, 28, 47] have provided durable transactions as the software interface for accessing persistent memory.

Most implementations of durable transactions enforce crash consistency through logging [10, 22, 28, 53]. However, there is a *dilemma* in choosing between undo logging and redo logging [48], 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 [10, 22, 28, 53] 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 the *persist* operation, which

is implemented by a sequence of instructions [41], for *every* update. This incurs significant overhead [19, 47, 48]. To mitigate the overhead, one approach [22, 28] is to log *all* the old values that are about to be updated in a transaction at once, to reduce the frequency of persist ordering from once per update to once per transaction. However, it 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* [19, 47, 53] 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 real data in-place. The process is referred to as *update redirection*. The system only need to guarantee that updates of old values happen *after* the whole log is persisted. 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. Update redirection incurs the overhead of address mapping, which can be expensive [38, 48]. Recently, Kiln [58] avoids such overhead by using non-volatile caches. However, it requires non-trivial hardware changes, including modifications of the cache coherent protocol.

The dilemma between undo and redo logging is essentially a trade-off 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 used. 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 in a shadow memory[1], and produce a redo log for the transaction. (2) `Persist`: flush the redo log of each transaction to persistent memory in an atomic manner. (3) `Reproduce`: modify original data in persistent memory according to the persisted redo log. It is essential that we never directly write back dirty data from shadow memory to persistent memory – all the updates are realized via the redo log. One implication is that we never need to worry about whether uncontrollable CPU cache eviction may affect 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 without redirection in `Perform`, and requires only transaction-level persist ordering in `Persist` and `Reproduce`, achieving both low 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. In contrast, traditional redo logging incurs costly object-level or even byte-level mapping.

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

This paper presents DUDETM, a **du**rable **de**coupled transaction system for persistent memory that implements the above insight and design. Besides resolving the dilemma between undo and redo logging, DUDETM offers three advantages. First, `Perform` in DUDETM can be easily built with any *out-of-the-box* transactional memory (TM) implementation. It can be a software TM (STM) or even a hardware TM (HTM). Second, each thread conducts `Perform` of consecutive transactions back-to-back, without stalling due to persistence. By design, DUDETM reduces persistence-induced stalls with less complexity than recently proposed relaxed persistence models [26, 28]. Third, the fully decoupled steps bring many unique optimization opportunities, such as *1)* overlapping the three steps, and *2)* 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 [8, 18].

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. Our implementation, based on an existing STM named TinySTM [16], achieves $1.7\times$ to $4.4\times$ higher throughput than prior systems, NVML [22] and Mnemosyne [47], with various benchmarks including TPC-C [45] and TATP [43].
- 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* TM, either STM or HTM (with minor modification), can be used as a standalone component for implementing durable transactions on persistent memory. Most prior implementations are coupled with and bound to a specific TM. Besides, our evaluation shows that HTM gives an additional $1.7\times$ speedup compared to a STM based implementation.

2. Background and Motivation

2.1 Requirements of Durable Transactions

Requirements of durable transactions have been well investigated in databases and file systems. They are applicable to persistent memory. We briefly review their four basic

properties, atomicity, consistency, isolation, and durability (ACID) [20].

First, a persistent memory system may contain volatile storage components, such as CPU caches and possibly DRAM for performance purpose [34]. Still, it has to retain each data update of acknowledged transactions despite power loss or system crashes. This is referred to as the *durability* property. Second, a logical update of data records performed by a transaction typically constitutes a sequence of 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 local 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*. 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 is typical that the application rather than the storage system is responsible for defining consistent data updates. Prior research [16, 17, 20, 21] has shown that the transaction with atomicity and isolation guarantees is a powerful and convenient interface to realize consistency.

2.2 Drawbacks of Undo and Redo Logging

In order to develop ACID transactions for persistent memory, many logging techniques have been proposed. For example, NVML [22], NV-Heap [10], and DCT [28] 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`) [41], incurring significant overhead [8, 15, 44, 47]. Hence, it is prohibitively expensive to enforce persist ordering for *every* memory write in a transaction. To mitigate this issue, DCT [28] and NVML [22] 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.

Another approach of supporting ACID 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 [47] 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 trade-off is that one can reduce the number of persist ordering (as in undo logging), at the cost of intercepting and redirecting writes and reads. Although object-based [3, 31, 42] or block-based [40, 54] 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 [17, 49]. Indeed, a *page-level* mapping mechanism (e.g., as used in eNVy [53]), incurs lower overhead than a finer-grained mapping mechanism (e.g., as used in Mnemosyne [47] or SoftWrAP [19]) because page-level mappings occupy relatively small memory space and can be accelerated by CPU 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 [38, 48].

Consider 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. DUDETM Design

Based on the principle of decoupling, this section describes the design of DUDETM, a C library that provides *1)* an easy-to-use interface that is almost identical to traditional TM; and *2)* the guarantees of complete ACID in spite of a system crash. DUDETM requires almost no hardware modifications and supports both static and dynamic transactions.

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 [19, 47, 53]. 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 DUDETM, 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 DUDETM.

Second, we use an *out-of-the-box* TM to execute transactions on the shadow memory for isolation and consistency. A TM implementation, either HTM or STM, 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 DUDETM 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 a 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. Each committed transaction[2] generates one redo log, which is temporarily stored into one of *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.

[2] Here, commit is in terms of the TM, which guarantees isolation, consistency and atomicity *with regard to the shadow memory*, but not durability.

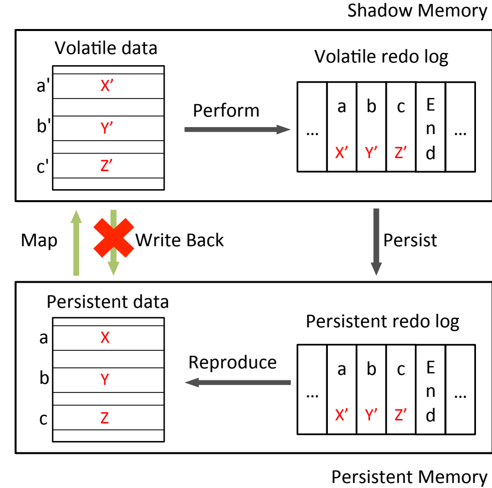

Figure 1. DUDETM memory architecture and data movement in the three steps. Data X, Y and Z locate at addresses a, b and c in the persistent memory, respectively. These addresses are mapped to a', b' and c' in the shadow memory.

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 background.

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

Figure 1 depicts DUDETM'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 regular memory mapping, DUDETM 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 4.3. Next, we introduce more details of the three steps.

3.2 Perform

DUDETM 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 DUDETM to perform a transaction that transfers $1 from one account to another in a bank.

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();
```

DUDETM 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 DUDETM'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, DUDETM 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 transactions in `Perform`. DUDETM 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 [16, 17]. Meanwhile, according to our experiments, the current maximum transaction throughput has not reached such a level that the global clock is the bottleneck.

In DUDETM, 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 the thread in `Perform` step. In our evaluation, we find that the log flushing of `Persist` is generally faster than the log entry generation of `Perform`. Therefore, a thread rarely blocks for this reason in practice.

Algorithm 2: DUDETM APIs 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);
```

3.3 Persist

DUDETM 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, DUDETM 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 access 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 another word, 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, DUDETM 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`, DUDETM have the freedom to persist redo logs in a batched manner. This brings new optimization opportunities that is 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 [37]. 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 insert 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. DUDETM uses the lz4 [11] 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.

3.4 Reproduce

In `Reproduce`, DUDETM 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. DUDETM 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, DUDETM scans the whole persistent log region and replays logs that have not been processed by `Reproduce`. Similar to `Reproduce`, the recovery procedure 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 notice that and may re-execute the transaction after recovery according to its own application-specific policy.

Another step in recovery is to restore persistent memory allocation information. Similar to other durable transaction systems [19, 22, 47], DUDETM 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 DUDETM can determine which regions of persistent memory are allocated.

4. Implementation

In this section, we describe two implementations of DUDETM based on STM and HTM. We also discuss the details of memory management.

4.1 STM-based Implementation

We developed an implementation of DUDETM using TinySTM [16], a lightweight STM that represents the category of *time-based* software TM [16, 17, 39]. 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[3].

By design, the three steps in DUDETM'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, DUDETM 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, DUDETM can expose the application individual transaction IDs and the

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

global durable ID to the users. Then the application based on DUDETM 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) [23]. As a representative HTM implementation, RTM offers `XBEGIN` and `XEND` to specify an HTM transaction, a code 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 aborted transaction or execute a fallback routine.

The interface of HTM is largely identical to a STM, so using HTM for DUDETM 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 DUDETM 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 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 5.7).

4.3 Memory Management

In practical system configuration, the shadow memory, typically DRAM, is typically smaller than the persistent memory. A paging mechanism is required to swap shadow and persistent pages. The implementation of paging mechanism is critical for the performance of the DUDETM system.

Our paging mechanism is similar to the one used in the operating system. 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, DUDETM has to evict an existing shadow page. Consequently, the persistent-shadow page address mappings have to be updated accordingly. Different from the traditional OS paging, the evicted page is simply discarded. It is correct because all updates to that pages 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 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, DUDETM 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, therefore, the page loading needs to wait until the touching ID become smaller.

In DUDETM, we implement both hardware-based and software-based paging mechanisms.

Hardware-based Paging. We use Dune [6] to trigger page faults and manage page mappings. Dune utilizes Intel's VT-x virtualization technique to enable user-space programs to manipulate their own page tables and use other privileged CPU features. One advantage of this approach is that the address translation between virtual memory and shadow memory is efficiently done by TLB. But accordingly, we have to flush TLBs of all processors, i.e., do TLB shootdown, when a thread modifies a page mapping. We add a TLB shootdown feature to Dune by reusing the inter-processor interrupt facility in the Linux kernel. DUDETM has to stall all threads and issue the `INVVPID` instruction to perform TLB shootdown.

Software-based Paging. We maintain a simple one-level page table in DRAM. Every `dtmRead`/`dtmWrite` operation has to look up the page table to translate a virtual address to an address[4] in the shadow memory. The address translation and page swapping are all controlled by our software library. To ensure that a page to swap out is not being used by other transactions, 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.

5. Evaluation

5.1 Setup

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

Persistent Memory Emulation. As real NVM is not yet available, we emulate persistent memory using DRAM. Similar to prior works [7, 19, 47], our method models the slow writes of persistent memory, and ignores the small additional latency of reads. Specifically, *1)* if a single write

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

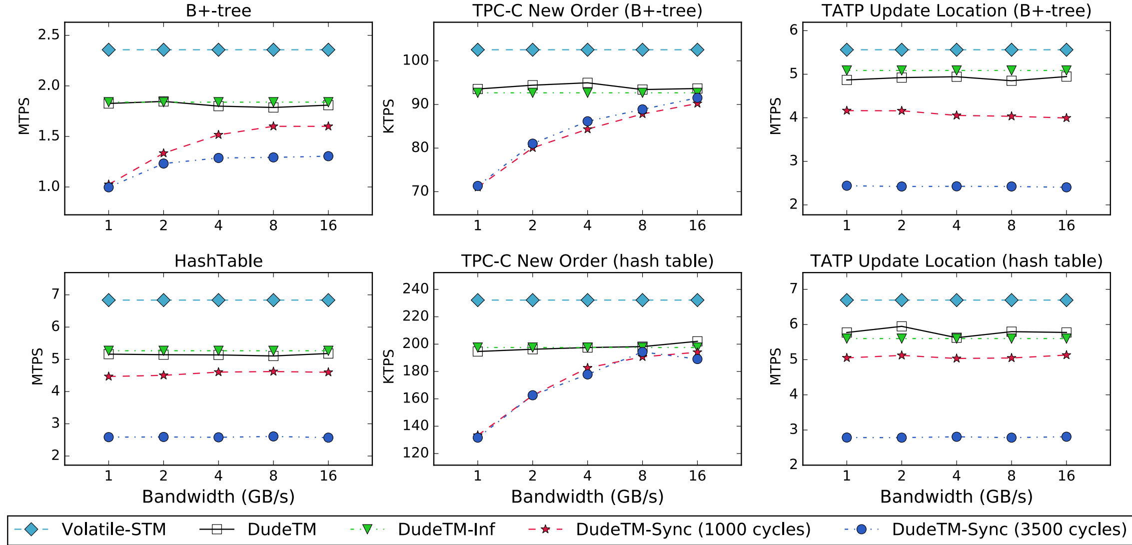

Figure 2. Throughput of evaluated systems with different NVM bandwidth.

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 1 us [8, 15], hence we set the extra delay to 3500 cycles, the same as prior works [19, 50]. In addition, we believe the write latency would drop in future, so we also run experiments with the extra delay set to 1000 cycles (about 300 ns). 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 DUDETM with both micro benchmarks (HashTable and B+-Tree), and realistic workloads in TPC-C [45] and TATP [43]. (1) The *HashTable* benchmark inserts randomly generated inputs to a simple fixed-size hash table that maps 64-bit integer keys to 64-bit integer values. Hash collisions are resolved by circularly testing the next bucket. We guarantee the ACID of every operation by wrapping it in a transaction. (2) The *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. (3) *TPC-C* [45] is a well-known on-line transaction processing (OLTP) benchmark. We implement its New Order transaction, 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. As our implementation is identical to [7], we omit the details. (4) *TATP* [43] 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. This gives the theoretical performance upper bound of DUDETM based on TinySTM. (2) DUDETM, the standard asynchronous implementation based our decoupled framework. The capacity of its log buffers is one 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) DUDETM-*Inf*, an asynchronous implementation of DUDETM that has infinite log buffers. In this case, `Perform` never has to block. (4) DUDETM-*Sync*, a synchronous implementation of DUDETM that immediately flushes logs to persistent memory after `Perform` step. The transaction returns after it becomes durable. In other words, the first and second steps of DUDETM are merged and transactions cannot be executed back-to-back.

5.2 Throughput

5.2.1 Performance Analysis

Theoretically, if ordered by throughput from high to low, the above four systems should be Volatile-STM, DUDETM-Inf, DUDETM, and DUDETM-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 benchmarks described in Section 5.1, with various per-

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

Table 1. Statistics of memory writes in different benchmarks (1 GB/s NVM bandwidth, 1000 cycles latency, 4 threads). "M/s" stands for millions per second.

sistent memory bandwidth, from 1 GB/s to 16 GB/s. Because the latency only affects DUDETM-Sync, we only use 1000 cycles latency for other systems. Figure 2 presents the results, from which we can derive the following three findings.

Finding (1): DUDETM incurs minor overhead ($7.4\% \sim 24.6\%$ less throughput) over the original volatile STM.

As expected, Volatile-STM provides the highest throughput among all systems under all benchmarks. Based on TinySTM, DUDETM adds a small performance penalty to the original STM. The throughput of DUDETM 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 DUDETM 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 DUDETM.

We see that DUDETM-Inf produces almost the same throughput as DUDETM, 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 16 GB/s NVM but also 1 GB/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 1 GB/s and latency is 1000 cycles. We can calculate that around 78 MB (TATP based on B+-tree) to 488 MB (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 DUDETM 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 (DUDETM-Sync), especially when the bandwidth of persistent memory is low (e.g., 1GB/s). We can see that the throughput of DUDETM-Sync is gradually improved with the increase of persistent memory bandwidth (unless the latency of persistent memory is the bottleneck). When the bandwidth is beyond 8 GB/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. 3500 cycles) is negligible compared to the whole execution time. As a result, for those benchmarks, the performance gap between DUDETM and DUDETM-Sync is small. In contrast, for transactions that have fewer instructions (e.g. a TATP transaction costs about 3000 cycles), they show a clear performance decline from DUDETM to DUDETM-Sync, when latency is increased from 1000 cycles to 3500 cycles.

5.2.2 Comparison to Current Systems

Mnemosyne [47] is a transactional memory library for persistent memory, with the same guarantees as DUDETM. 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 DUDETM. Since it does not follow our decoupling approach, Mnemosyne faces the trade-off as described in Section 2.2, i.e., a redo logging scheme requires less fences but costly address mapping.

We also evaluate NVML [22], an undo logging based durable transaction library for persistent memory 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 mechanism (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, all the memory allocation operations are moved to the beginning of the program, so that our comparison is focused on transaction execution excluding the slow NVML allocation.

Benchmark	DUDE	DUDE-Sync	Mnem.	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 2. Throughput of DUDETM ("DUDE"), DUDETM-Sync ("DUDE-Sync"), Mnemosyne ("Mnem.") and NVML (1 GB/s NVM bandwidth, 1000 cycles latency, 4 threads).

Table 2 shows the result. As we can see, DUDETM and DUDETM-Sync are about $1.7\times$–$4.4\times$ and $1.3\times$–$3.0\times$ 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 [38]; (2) Mnemosyne needs to use `CLFLUSH`, which invalidates the cache line and hence increases cache misses. This problem is avoided in DUDETM because the shadow memory resides in DRAM; (3) Each transaction requires a costly synchronous persist, which is similar to DUDETM-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. However, NVML's specific implementation details affect its performance. For example, NVML dynamically allocates transaction meta data and undo logs for each transaction, which is very expensive[5]. 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 DUDETM/Mnemosyne is 30+ millions per second.

5.3 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 DUDETM 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 $\leq d_{i+1}$; ... In certain cases, the *latency* of a transaction (measured by the time between the beginning of the transaction and its durability acknowledgement) may increase in DUDETM because `Persist` is done asynchronously. In this section, we discuss our evaluation results of this latency by comparing DUDETM, DUDETM-Sync, Mnemosyne and NVML.

Percentage	DUDE	DUDE-Sync	Mnem.	NVML
50%	45 us	18 us	62 us	112 us
90%	73 us	40 us	87 us	161 us
99%	124 us	90 us	126 us	254 us

Table 3. Durable transaction latency of the hash table based TPC-C benchmark in DUDETM ("DUDE"), DUDETM-Sync ("DUDE-Sync"), Mnemosyne ("Mnem.") and NVML

Table 3 presents the distribution of latency (e.g., in DUDETM, 50% of the transactions can be durable within 45 us). We see that decoupling only introduces a moderate extra latency compared to DUDETM-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 DUDETM is about 2× the ideal latency that is 1/(throughput of DUDETM). Because DUDETM's throughput is more than 2× that of Mnemosyne and NVML, DUDETM even has a better latency performance than those existing synchronous durable transaction systems. Compared to them, DUDETM achieves *both* higher throughput *and* lower latency.

5.4 Log Optimization

The decoupled framework enables log optimization techniques to apply before the log is flushed. 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, 13], many real-world workloads are indeed skewed. In this experiment, we use the Session Store workload of YCSB [12] and runs 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 to 50. Transactions follow the Zipfian distribution with a constant of 0.99.

Figure 3 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 increase to 93% when each group contains 100,000 transactions. Figure 3 also shows the result of using lz4 [11] 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`.

DUDETM allows users to explore the trade-off in log 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 other transactions in the same group to persist.

5.5 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

[5] In DUDETM and Mnemosyne, both the meta data and log buffers are allocated collectively on thread creation

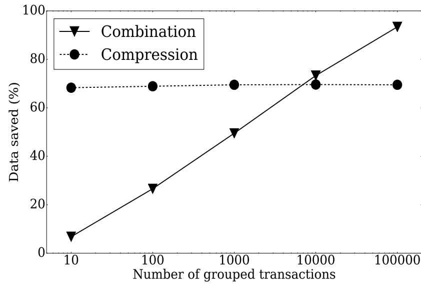

Figure 3. Log optimization

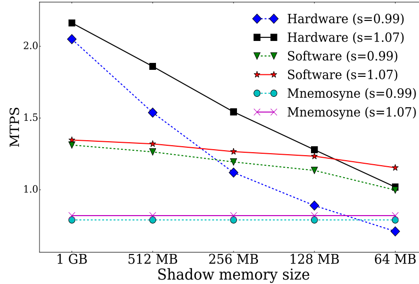

Figure 4. Swap overhead

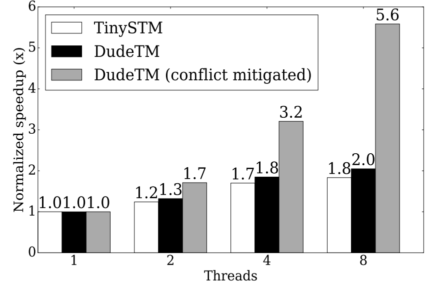

Figure 5. Scalability

(2) how skew the transactional access is. Figure 4 depicts the throughput of updating a B+-tree based key-value store. The workloads are generated following the Zipfian distribution with two constants, 0.99 and 1.07. We use 1 GB NVM and a shadow DRAM that varies from 64 MB to 1 GB. The total working set of each workload is around 650 MB. The figure indicates that the throughput decreases as the size of shadow memory shrinks or when the workload becomes less skew.

Our software-based and hardware-based implementations have different sensitivity over the shadow memory size. Figure 4 shows that hardware-based paging has better performance than software-based paging when the shadow memory is relatively large, but its performance drops 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[6]. In contrast, 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.

5.6 Scalability

Figure 5 presents our results on the scalability of DUDETM. It shows the throughput of running B+-tree based TPC-C with different numbers of threads. The NVM bandwidth is set to 1 GB/s and the latency is 1000 cycles (actually they have negligible impact on results). The results are normalized to the throughput of one thread. We see that our TinySTM-based DUDETM implementation achieves a similar speedup as TinySTM itself. DUDETM has even a little better speedup, because its throughput of one thread is slower than TinySTM.

The scalability bottleneck of DUDETM lies in the concurrency control mechanism of TinySTM, instead of other parts of DUDETM. 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 5 that scalability of DUDETM shows almost linear scalability in this case. In another word, DUDETM-specific overhead has little influence on scalability.

5.7 HTM-based DUDETM

As 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 HTM 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.

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

Table 4. Throughput of DUDETM based on STM and HTM (1 GB/s NVM bandwidth, 1000 cycles latency, 4 threads).

Table 4 shows the throughput of HTM-based DUDETM[7]. It achieves up to $1.7\times$ higher throughput than the STM-based implementation. Among all benchmarks, B+-Tree shows the largest speedup. That 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 $1.33\times$) because there is only one concurrent write in a transaction, which means that most of the execution time is spent on local reads that can not be improved by replacing STM with HTM. The same reason applies to the HashTable benchmark in which aborted

[6] This is due to our implementation with Dune, and it can be improved in future work.

[7] TPC-C is not shown because its transaction issues such a large write set that Intel Haswell's HTM cannot handle.

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 DUDETM is still within 28% for both STM and HTM. That means our decoupled framework is compatible to and effective with many kinds of transactional memory technologies.

6. Related Work

Durable Transaction Systems. Previous NVM-based durable transaction systems [10, 19, 28, 47] usually suffer from a dilemma between per-update persist ordering and update redirection overhead. DCT [28] and NVML [22] bypass this issue by supporting only static transactions. They also require that all the locks should be acquired at transaction start and released after the transaction is committed. It leads to less parallelism than TM implementations [35]. Mnemosyne [47] also provides an alternative asynchronous reproducing method which replays redo logs on background, similar to the asynchronous `Reproduce` step in DUDETM. However, it does not decouple the `Perform` and `Persist` steps of a transaction. In other words, it does not solve the dilemma and shows low performance especially when the write latency of persistent memory is large.

Moreover, although SoftWrAP and DUDETM both use the concept of *shadow memory*, they are different in several fundamental aspects: *1)* SoftWrAP is still using object-level mapping. Due to its "double-buffered" alias table mechanism, SoftWrAP needs up to three times of indirection for reading a value. *2)* Changing SoftWrAP's mapping granularity to relieve its indirection overhead is not easy. In SoftWrAP's implementation, a page-level mapping will lead to excessive write amplification overhead. Meanwhile, a straightforward one-to-one mapping will result in both unacceptable memory consumption ($> 3\times$ 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). *3)* SoftWrAP argues that concurrency control can be decoupled, but it requires non-trivial modifications to work in a concurrent environment. It claims that isolation can be implemented by using local alias table, but it is not clear how to merge it into global alias table automatically. 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 because that SoftWrAP tries to directly copy data from shadow memory to NVM. As a result, copying data from the per-thread redo log can fundamentally avoid these issues.

Database Systems. Some in-memory databases [27, 46, 51] flush logs into disks to provide ACID guarantees. These systems usually achieve a better throughput than TM techniques, but their data can only be accessed by specific database operations. In contrast, the TM interface is more flexible. Moreover, certain databases may have extra constrains. For example, the main technique used in FOEDUS [27] is dual-page, which requires data to be fit in fixed-size pages and hence is not usable in transactions for persistent memory.

NVM-Oriented Data Structures. Some researchers [8, 9, 14, 52, 55] focus on the best method of implementing a certain kind of data structure on NVM. For example, Yang el al. [55] design a high-performance persistent B+-tree, which reduces the number of reuqired write operations by disordering the keys in leaf nodes. Different from these works that deal with a specific kind of data structure, DUDETM attempts to provide a general transactional library for wider use (e.g., mingling of different data structures).

Hardware Support. Kiln [58] adds a non-volatile cache to eliminate logging, and WSP [32] achieves similar improvements by assuming that residual power could flush data in caches and registers to NVM. DUDETM makes more conservative hardware assumptions.

7. Conclusion

This paper presents DUDETM, a decoupled framework to implement atomic, durable transactions on persistent memory. DUDETM 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 TM on the shadow memory as a stand-alone component in our system. Our evaluation results show that DUDETM adds guarantees of crash consistency and durability to TinySTM by adding only $7.4\% \sim 24.6\%$ overhead, and is $1.7\times$ to $4.4\times$ faster than existing works Mnemosyne and NVML. Through decoupling, we have also enabled the possibility of *1)* reducing the write traffic to NVM by log optimization (up to 93% reduction) and *2)* improving throughput by using HTM (a further $1.7\times$ speedup).

Acknowledgement

We thank anonymous reviewers for their valuable feedback. This work is supported by Natural Science Foundation of China (61433008, 61373145, 61572280, 61133004, 61502019, U1435216), National Key Research & Development Program of China (2016YFB1000504), National Basic Research (973) Program of China (2014CB340402), Intel Labs China (Funding No.20160520). This work is also supported by NSF CRII-1657333, Spanish Gov. & European ERDF under TIN2010-21291-C02-01 and Consolider CSD2007-00050.

References

[1] AKINAGA, H., AND SHIMA, H. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE 98*, 12 (2010).

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

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

[4] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. 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* (2012), SIGMETRICS '12, pp. 53–64.

[5] Atwood, G. Current and emerging memory technology landscape. *Flash memory summit* (2011), 9–11.

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

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

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

[9] Chi, P., Lee, W.-C., and Xie, Y. Making B+-tree efficient in PCM-based main memory. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design* (2014), ISLPED '14, pp. 69–74.

[10] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI, pp. 105–118.

[11] Collet, Y. Lz4: Extremely fast compression algorithm. https://github.com/lz4/lz4, 2013.

[12] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), SoCC '10, pp. 143–154.

[13] Cunha, C., Bestavros, A., and Crovella, M. Characteristics of WWW client-based traces. Tech. rep., BU-CS-95-010, Computer Science Department, Boston University, 1995.

[14] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A., and Castro, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 54–70.

[15] Eilert, S., Leinwander, M., and Crisenza, G. Phase change memory: A new memory enables new memory usage models. In *2009 IEEE International Memory Workshop* (May 2009), pp. 1–2.

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

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

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

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

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

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

[22] Intel. NVM Library . https://github.com/pmem/nvml.

[23] Intel. Architecture instruction set extensions programming reference, Feb. 2012.

[24] Intel, and Micron. Intel and Micron produce breakthrough memory technology, 2015. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory.

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

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

[27] Kimura, H. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), SIGMOD '15, pp. 691–706.

[28] Kolli, A., Pelley, S., Saidi, A., Chen, P. M., and Wenisch, T. F. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ASPLOS '16, pp. 399–411.

[29] Kltrsay, E., Kandemir, M., Sivasubramaniam, A., and Mutlu, O. 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* (Apr. 2013), ISPASS '13, pp. 256–267.

[30] LEE, B., ZHOU, P., YANG, J., ZHANG, Y., ZHAO, B., IPEK, E., MUTLU, O., AND BURGER, D. Phase-change technology and the future of main memory. *IEEE Micro 30* (Jan. 2010), 131–141.

[31] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), SOSP '11, pp. 1–13.

[32] NARAYANAN, D., AND HODSON, O. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS XVII, pp. 401–410.

[33] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (2014), ISCA '14, pp. 265–276.

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

[35] RAMADAN, H. E., ROSSBACH, C. J., AND WITCHEL, E. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture* (2008), MICRO-41, pp. 246–257.

[36] RAOUX, S., BURR, G. W., BREITWISCH, M. J., RETTNER, C. T., CHEN, Y.-C., SHELBY, R. M., SALINGA, M., KREBS, D., CHEN, S.-H., LUNG, H.-L., AND LAM, C. H. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev. 52*, 4 (July 2008), 465–479.

[37] REN, J., LIANG, C.-J. M., WU, Y., AND MOSCIBRODA, T. Memory-centric data storage for mobile systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (2015), USENIX ATC '15, pp. 599–611.

[38] REN, J., ZHAO, J., KHAN, S., CHOI, J., WU, Y., AND MUTLU, O. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture* (2015), MICRO-48, pp. 672–685. `http://persper.com/thynvm/`.

[39] RIEGEL, T., FETZER, C., AND FELBER, P. Time-based transactional memory with scalable time bases. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (2007), SPAA '07, pp. 221–228.

[40] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992), 26–52.

[41] RUDOFF, A. Deprecating the PCOMMIT instruction. `https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction`, Sept. 2016.

[42] RUMBLE, S. M., KEJRIWAL, A., AND OUSTERHOUT, J. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (2014), FAST '14, pp. 1–16.

[43] SIMO, N., ANTONI, W., MARKK, M., AND VILHO, R. Telecom application transaction processing benchmark. `http://tatpbenchmark.sourceforge.net/`.

[44] SUZUKI, K., AND SWANSON, S. A survey of trends in non-volatile memory technologies: 2000-2014. In *2015 IEEE International Memory Workshop (IMW)* (May 2015), pp. 1–4.

[45] THE TRANSACTION PROCESSING COUNCIL. TPC-C Benchmark V5. `http://www.tpc.org/tpcc/`.

[46] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP '13, pp. 18–32.

[47] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI, pp. 91–104.

[48] WAN, H., LU, Y., XU, Y., AND SHU, J. Empirical study of redo and undo logging in persistent memory. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)* (Aug. 2016), pp. 1–6.

[49] WANG, C., CHEN, W.-Y., WU, Y., SAHA, B., AND ADL-TABATABAI, A.-R. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization* (2007), CGO '07, pp. 34–48.

[50] WANG, T., AND JOHNSON, R. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment 7*, 10 (2014), 865–876.

[51] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys '14, pp. 26:1–26:15.

[52] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 87–104.

[53] WU, M., AND ZWAENEPOEL, W. eNVy: A Non-volatile, Main Memory Storage System. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (1994), ASPLOS VI, pp. 86–97.

[54] XU, J., AND SWANSON, S. 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* (2016), FAST '16, pp. 323–338.

[55] YANG, J., WEI, Q., CHEN, C., WANG, C., YONG, K. L., AND HE, B. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (2015), FAST '15, pp. 167–181.

[56] YOON, J. H., HUNTER, H. C., AND TRESSLER, G. A. Flash & DRAM Si scaling challenges, emerging non-volatile mem-

ory technology enablement – implications to enterprise storage and server compute systems. *Flash Memory Summit* (2013).

[57] ZHANG, Y., AND SWANSON, S. A study of application performance with non-volatile main memory. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies* (May 2015), MSST '15, pp. 1–10.

[58] ZHAO, J., LI, S., YOON, D. H., XIE, Y., AND JOUPPI, N. P. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013), MICRO-46, pp. 421–432.