REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures

Andreas Chatzistergiou
University of Edinburgh, UK
a.chatzistergiou@sms.ed.ac.uk

Marcelo Cintra
Intel, Germany
marcelo.cintra@intel.com

Stratis D. Viglas
University of Edinburgh, UK
sviglas@inf.ed.ac.uk

ABSTRACT

Recent non-volatile memory (NVM) technologies, such as PCM, STT-MRAM and ReRAM, can act as both main memory and storage. This has led to research into NVM programming models, where persistent data structures remain in memory and are accessed directly through CPU loads and stores. Existing mechanisms for transactional updates are not appropriate in such a setting as they are optimized for block-based storage. We present REWIND, a user-mode library approach to managing transactional updates directly from user code written in an imperative general-purpose language. REWIND relies on a custom persistent in-memory data structure for the log that supports recoverable operations on itself. The scheme also employs a combination of non-temporal updates, persistent memory fences, and lightweight logging. Experimental results on synthetic transactional workloads and TPC-C show the overhead of REWIND compared to its non-recoverable equivalent to be within a factor of only 1.5 and 1.39 respectively. Moreover, REWIND outperforms state-of-the-art approaches for data structure recoverability as well as general purpose and NVM-aware DBMS-based recovery schemes by up to two orders of magnitude.

1. INTRODUCTION

Non-volatile memory (NVM) technologies, such as PCM, STT-MRAM and ReRAM, raise the prospect of persistent byte-addressable random access memory with large enough capacity to double as storage. By itself this would allow applications to store their persistent data in main memory by mounting a portion of the file system to it. This introduces NVM into the data management programming stack, but in a far from ideal manner. Consider a typical multi-tier application: the programmer decides on the application-level control and data structures, and then decides on the storage-level persistent representation of the data structures. Specialized APIs (*e.g.*, embedded SQL, JDBC, *etc.*) translate data between the two runtimes using SQL as the intermediate language, in a cumbersome and sometimes error-prone process. Moreover, data may be replicated in both DRAM and NVM, while the byte-addressability of NVM is not leveraged. Clearly, this is suboptimal. An alternative is to port an in-memory database system to persistent memory. That would make use of byte addressability, but it would still require data be replicated and represented in two data models. We argue that we need a solution that is not intrusive to the programmer and seamlessly integrates the application's data structures with their persistent representation.

We target use-cases where the data owner has full control of the data, foresees little change to the schema, and would like to tightly co-design the schema with the operations for performance. These use-cases capture a large group of contemporary applications. Indeed, persistence APIs are being used across a variety of operating systems [1, 18]. These platforms support either a persistent storage manager [22] or an embedded database [16]. Our stance is that in such scenarios we are better off integrating the storage manager and the application memory spaces. By doing so we enable the use of arbitrary persistent data structures in a lightweight software stack that significantly reduces the cost of managing data [15, 27]. To address these issues we introduce REWIND: a user-mode library that enables transactional recoverability of an arbitrary set of persistent updates to main memory data. The runtime system of REWIND transparently manages a log of program updates to the critical data and handles both commit and recovery of transactions. By tracking the operations of the transactions that commit the runtime can identify the point of failure and can resume operation relying on the consistency of critical data.

Our work stems from an alternative persistent memory access model that has gained interest recently: *directly* programming with persistent memory through mechanisms such as persistent regions [14] or persistent heaps [5, 31]. Persistent data is accessed directly by the processor with a load-store interface and with (mostly) automatic persistence without interacting with the I/O software stack. This model is highly disruptive as it enables a new class of data management systems in which both user data and database metadata are managed entirely in memory as one would manage volatile data [29]. Thus, some fundamental assumptions on programming interfaces and software architecture need to be revised as persistent data needs to be directly managed by the programmer's code using imperative languages.

The programmer-visible API of REWIND offers two main functionalities: one to demarcate the beginning and end of

Proceedings of the VLDB Endowment, Vol. 8, No. 5

transactions and one to log updates to critical data. Our intention is to completely do away with the second functionality by relying on compiler support so that the programmer only needs to identify the critical data. The key challenge of using persistent in-memory data structures is guaranteeing consistent data updates in the presence of system failures (*e.g.*, power failures and system crashes). This differs from transactional memory where the main intention is to deal optimistically only with atomicity and isolation. It also differs from non-recoverable device failure as this is an orthogonal issue and is hardware-related; we only target failures due to system and software malfunctions. REWIND provides full atomicity and durability for persistent memory through write-ahead logging. While the principles of the mechanism are still applicable to persistent memory, its implementation and tradeoffs must be revisited given the significant differences in access latencies and synchronization control (see, *e.g.*, [10, 12]). Thus, REWIND overcomes a number of challenges that arise in this new context.

First, the processing model is that persistent data is on byte-addressable NVM, accessible directly from user code through CPU loads and stores.[1] Traditionally, data updates are first performed in volatile memory. It is thus possible to delay making log entries persistent until the transaction commits or the data updates are purged from main memory. In REWIND, updates are done directly on NVM data: the log entries must be made persistent immediately, and ahead of the data updates. We achieve this through enhanced versions of memory fences (*i.e.*, barriers that enforce ordering and persistence to preceding instructions), cacheline flushes and non-temporal stores (*i.e.*, direct to NVM stores that bypass the cache) with persistence guarantees.

REWIND uses physical logging as it fits better with imperative languages and allows easier compiler support. However, it might result in more log records than logical/physiological logging when memory blocks are shifted in memory. Then, the log itself must be manipulated atomically in a recoverable way. Traditionally, the log is maintained in volatile memory and pushed to persistent storage through system calls. In REWIND, the log itself resides in persistent main memory and updates are made in-place. Transactional handling of failure of log updates is attained with *carefully crafted data structures and code sequences*. Furthermore, performance is relative to a baseline with the low cost of individual memory operations. Thus, logging must be optimized to incur only a small increase in the cost of a memory operation. In REWIND, we guarantee this with *minimalist data structures and code sequences*. Moreover, while contemporary systems offer record level locking from a data-centric perspective, they use coarse-grained page-level latching internally. REWIND employs *fine-grained latching at a log record granularity*: this enables more efficient and flexible locking mechanisms. Finally, the majority of recovery managers based on ARIES [20] are implemented within DBMSs. Thus, they hide data management behind some data model (*e.g.*, relational) and allow data manipulation through a well-defined query language (*e.g.*, SQL). REWIND is implemented as a *user-mode library* that can be linked to any native application, giving the programmer full access to the data using an arbitrary sequence of imperative commands. Moreover, the design of REWIND itself is such that it can be straightforwardly embedded into the compiler so that the disruption to user code is further minimized.

[1]NAND-based battery-backed NV-DIMMs already support this (see also `http://www.smartstoragesys.com/pdfs/ULLtraDIMM_overview.pdf`). Newer technologies will bring more practical implementations.

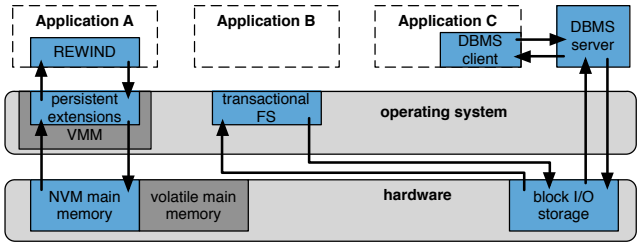

Figure 1: Transactional access to application data.

Contributions and organization Our contributions and the structure of the rest of this paper are as follows:

- We introduce REWIND, a user-mode library for logging and transaction management in NVM.
- We explore the design space of REWIND (Section 2) via four configurations that result from choosing between: (*a*) two different log implementations optimized for minimizing either logging overhead or search speed; and (*b*) forcing or not user data to non-temporal stores.
- We describe two ways to implement the log completely in persistent memory, so it is recoverable and atomic as well. We then present two optimized log versions that further reduce the write overhead (Section 3).
- By leveraging the recoverable log we show how to enable in-memory persistent data structures. We present how these new mechanisms can be incorporated into imperative general-purpose languages through the REWIND library and runtime (Section 4).
- We analyze the sensitivity of REWIND to its parameters and show how it can be configured to deliver low-overhead transactional processing and recoverability of data structures in NVM. We compare REWIND to state-of-the-art recovery managers as well as to traditional and NVM-aware DBMS-based techniques. REWIND's overhead is within a factor of 1.5× from its non-recoverable counterpart, while it outperforms the competition by up to two orders of magnitude (Sections 5.1 and 5.2).
- We use a modified version of TPC-C to show how REWIND enables the co-design of algorithms and data structures. Workload- and program-specific optimizations result in a REWIND performance within a factor of 1.3× from its non-recoverable version (Section 5.3).

Finally, we present related work in Section 6 and conclude and identify future work directions in Section 7.

2. SYSTEM OVERVIEW

REWIND is a user-mode recovery runtime system that can be used by programmers and compilers to provide atomic recoverability to arbitrary code that operates on persistent data structures in NVM. We envision the REWIND library being statically linked with executables, but other variations such as dynamic linking or a shared library could also be developed. Thus, REWIND can be used then as a standalone recovery manager for individual applications, or

Scheme	Pros	Cons
Transactional FS	• portability • scalability	• programmability • flexibility
DBMS	• robustness • scalability	• initial perf. cost • flexibility
REWIND	• programmability • initial perf. cost	• disruptive model

Table 1: Pros and cons of the options of Figure 1.

could be used as the building block of a larger, multi-user data management system. We view REWIND as a fundamental building block towards introducing persistence at the system level, especially when data outlives applications.

In Figure 1 we show the options available for transactional data management. The application can interface with a DBMS (Application C), or with a file system that offers transactional access to user data (Application B) [7, 28]. The library approach of REWIND (Application A) operates directly on data stored in NVM. This requires an NVM-aware memory manager in the OS [5, 31]. Table 1 summarizes the high-level pros and cons of each option. The use of a file system leads to portable data formats, but suffers in terms of programmability and flexibility by expecting the programmer to manage both an in-memory and a serialized on-disk version of the data. The DBMS is a tried approach, but limits flexibility by imposing a data model and query API, and suffers from the overheads of a client-server architecture and a highly complex server. REWIND offers increased programmability by enabling in-memory persistent data structures and APIs as well as the low overhead of a user-mode library. REWIND currently targets individual applications and may require additional functionality for operation in multi-user environments. It provides, however, a critical mechanism to enable this new class of data management in NVM. Others have also argued for supporting a variety of storage models and infrastructures to meet the demands of different workloads (*e.g.*, [27]).

The core of REWIND is a transactional recovery protocol based on WAL (Write-Ahead Logging). Unlike the ARIES implementations of DBMSs, REWIND provides programmers with direct control of what updates should be transactional through a simple API with a construct to mark transactions and a function call to the log operation. Logging calls are currently inserted manually by the programmer, but we expect them to be inserted transparently by the compiler, similarly to how Software Transactional Memory (STM) compilers work [11, 32]. Listing 1 is a function to remove an element from a doubly-linked list with the state of the list updated through CPU writes (lines 3 to 6). To make the operation recoverable, we enclose the critical updates in a `persistent atomic` block.

```
1 void remove (node* n) {
2   persistent atomic {
3     if (n == tail) tail = n->prv;
4     if (n == head) head = n->nxt;
5     if (n->prv) n->prv->nxt = n->nxt;
6     if (n->nxt)n->nxt->prv = n->prv;
7     delete (n); } } // end of atomic block
```

Listing 1: Removal from a doubly-linked list.

To provide programmers with the familiar access interface of current main memory data, we are restricted to physical logging and in-place updates. This differs from traditional disk-based systems where we are free to use logical logs or delay forcing user updates to improve performance. As REWIND is based on write-ahead logging, all updates to persistent data must be preceded by a call to the log function. This separates data from the log as the data are handled by the user program while the log is handled by REWIND. The resulting code, as expanded by the programmer or compiler, is shown in Listing 2. The runtime's transaction manager is called at the start of the block (line 2) to create a new transaction identifier: transaction management is transparent to the programmer and compiler. Logging calls (*e.g.*, line 4) precede every critical update (*e.g.*, line 5). Logging call parameters include the transaction identifier, the address of the memory location being updated, and the previous and new values[2]. At the end of the expanded code the commit call marks the end of the persistent atomic block. The de-allocation of the memory occupied by the removed node must be placed after transaction commit (line 16): without additional OS support, de-allocating memory is an operation that cannot be undone by REWIND.

```
1 void remove (node* n) {
2   int tID = tm->getNextID();
3   if (n == tail) {
4     tm->log(tID, &tail, tail, n->prv);
5     tail = n->prv; }
6   if (n == head) {
7     tm->log(tID, &head, head, n->nxt);
8     head = n->nxt; }
9   if (n->prv) {
10    tm->log(tID, &n->prv->nxt, n->prv->nxt, n->nxt);
11    n->prv->nxt = n->nxt; }
12  if (n->nxt) {
13    tm->log(tID, &n->nxt->prv, n->nxt->prv, n->prv);
14    n->nxt->prv = n->prv; }
15  tm->commit(tID);
16  delete (n); }
```

Listing 2: Expanded code for Listing 1.

We propose and evaluate four configurations of logging and transaction management in REWIND through deciding: (*a*) whether or not to force user updates to NVM as they happen; and (*b*) the number of logging layers to employ (one or two). Each configuration comes with its own tradeoffs.

Forcing/not forcing user updates A force policy slows down logging due to the extra time needed to guarantee the persistence of the update. However, it only requires a two-phase recovery (*analysis* and *undo*) instead of the three-phase recovery (*analysis*, *redo*, and *undo*) of the no-force policy. Thus, the tradeoff is faster recovery over a slight slowdown during logging. The force policy also allows (without dictating, however) an alternative log clearing method instead of checkpoints. Each transaction can clear its own records immediately after commit, resulting in slower commits but eliminating checkpoints. Log clearing becomes more expensive as the number of concurrent transactions attempting to commit grows through increased locking congestion: clearing requires coarser-grained locks than adding records as it invalidates the iterators of concurrent threads. However, it utilizes memory better: memory is de-allocated right after commit and not after the checkpoint. It also minimizes the size of the log, which improves the time to find records of a transaction. In our implementation we combine the force policy with log clearing at commit-time.

[2]By address of a location we mean a persistent virtual address, *e.g.*, that offered by [31], a relative address, or some other form of persistent reference to the memory location.

Number of logging layers The logging infrastructure and the recovery manager offer two configurations of the log data structure. In its simplest form the log is a specially crafted recoverable persistent doubly-linked list. Alternatively, the log data structure is organized in two layers: an auxiliary data structure (an AVL tree) at the top layer, over the recoverable persistent doubly-linked list. Thus, the recovery manager caters for the recovery of up to three elements: *programmer transaction*, an optional *complex log structure*, and a *fundamental and simple data structure*. Recovery starts by recovering the simple data structure to a consistent state, whose contents are then used to recover the auxiliary log structure, if there is one. The contents of both primary and auxiliary log structures are used to recover the updates of the programmer transaction. The main tradeoff between these two variants is that logging is faster in the one-layer case but the two-layer log structure offers faster search which benefits rolling transactions back.

3. THE RECOVERABLE LOG

3.1 Design overview

In NVM, the log is itself an in-memory non-volatile data structure and log updates require a series of CPU writes. Thus, updates for logging and recovering user updates must themselves be logged and recoverable. The challenge is to ensure atomicity and durability for log updates using a recovery mechanism. Note that user updates are now much cheaper, which makes traditional logging infrastructure heavyweight and calls for a low overhead design.

DBMSs often use auxiliary data structures for indexing the log. Updating such structures may require a variable number of updates due to the need to re-organize the data structure. This makes maintaining transactional atomicity difficult in NVM. Our solution is to create a specialized data structure that embeds transactional logic and is able to recover itself in the case of a system failure. The data structure requires a constant number of operations to insert or remove an entry, so that its state can be tracked with only a few variables that can be updated and made persistent in a single, atomic, CPU write. This relies on only the last operation in the basic structure being pending, so we only need to log one operation. We also require the append/remove operations to be thread-safe. We force all updates on the basic data structure to be performed directly on NVM through: memory fences to force pending writes; non-temporal, synchronous, writes that bypass the cache and do not complete before reaching NVM; and cacheline flushes. All primitives are present in most instruction sets today, but they guarantee only write visibility within the memory consistency model of the machine; they do not guarantee persistence. We assume that when NVM systems become widely available they will be capable to also guarantee persistence to NVM. Most work on persistent data structures for NVM makes similar assumptions (*e.g.*, [5, 10, 12, 31]).

Based on these requirements we use a doubly-linked list as the basic log data structure. List nodes contain the log records, which contain information also found in ARIES, *e.g.*, a transaction identifier and the old and new values of the affected memory location, *etc.* With carefully crafted code it is possible to make atomic node insertions and deletions over the linked list. This *Atomic Doubly-Linked List* (ADLL) is the key data structure for logging user updates. However, it requires linear search to locate an entry. An alternative is to index log records by transaction identifier and use the ADLL to log the pending updates to the index. The result is a two-layer configuration where the index (an AVL tree in our case) logs pending user updates and the basic data structure (*i.e.*, the ADLL) logs pending REWIND updates to the complex data structure. The one-layer configuration offers faster logging but may lead to slower rollback; and vice-versa for the two-layer configuration.

3.2 One-layer logging: the Atomic Doubly-Linked List

Assuming that recovery and rollback are rare events, we can achieve faster logging at the cost of a slower retrieval of log entries. The only logging structure is the ADLL, so inserting a record costs a small constant number of writes. We optimize logging at the expense of more work during recovery. We do so by not keeping any transaction-specific state. At the price of a higher rollback/recovery cost we eliminate the transaction table during logging and reduce the number of variables we update; we only construct the transaction table during recovery. This departure from back-chaining is acceptable as we expect rollback/recovery to be rare events.

Instead of rolling back one transaction at a time, we perform a single backward scan of the log and recover all transactions, at the expense of higher memory utilization (see Section 4.5). Rolling back a single transaction is not typical in system failures, but selective rollback is necessary to allow users to abort specific transactions. To achieve this we need to scan the entire log just for the rolled back transaction. Long-running transactions exacerbate this, as do the number of concurrent transactions: they increase the number of records between records of the transaction being rolled back that we need to skip. To rectify, we clear the log at checkpoints (Section 4.6): by tuning the checkpointing frequency we balance the insertion overhead against the rollback speed.

The ADLL is a keystone of REWIND as it enables the atomic insertion and removal of log records into/from the log *in NVM*. The ADLL is recoverable itself through: (*a*) use of single variables to log the internal state, which can be updated atomically in hardware; (*b*) recovery by redoing only the last operation: repeated redos, either partial or in full (due to further system failures in the middle of an ADLL recovery), are safe and leave the list in a correct state; (*c*) simple operations that make it easier to produce code with the redo recoverability property; and (*d*) performing all writes via non-temporal stores. The ADLL uses four logging variables: `lastTail`, the tail of the list before insertion; `toAppend`, the node to be appended; and `toRemove`, the node to be removed. Each list node points to the next and previous nodes, and to the actual log record. The latter is so we can create new records "off-line" and atomically insert/append them to the list.

Append This operation involves creating the new node, updating the tail/head of the list (if needed) and the `next` pointer of the last tail. The operation for the ADLL is shown in Algorithm 1. Lines 5 and 10 mark the beginning and end, respectively, of the persistent operation. Line 5 corresponds to the critical update: it saves the node to be appended so the operation can be redone during recovery. If the system fails at any point before line 5, the state of the list is not altered, and thus consistent. If the system fails after line 5, the recovery operation (described next) will re-apply the

Algorithm 1: Append operation on the ADLL, invoked as part of the transaction manager's logging operation.

```
   input: element E to insert
 1 // set up new node
 2 n = new Node();     n.element = E;     n.prior = tail;
 3 // undo information
 4 lastTail = tail;     // Keep tail before logging last insertion
 5 toAppend = n;
 6 if head = NULL then head = n;                    // update head
 7 if tail ≠ NULL then tail.next = n;               // update tail
 8 tail = n;
 9 // append finished, clear undo
10 toAppend = NULL ;
```

append. Line 4 is not critical as it only sets `lastTail` and does not alter the state of the list. If the system crashes between lines 4 and 5 this value will be overwritten by the next append attempt. The order of the updates of lines 4 and 5 is critical for correct recovery. In line 6 the head of the list is updated, if necessary. This is not critical as it is designed so that it can be repeated multiple times during recovery. In lines 7 and 8 the `next` pointer of the tail and the tail itself are updated. If the system fails after line 10, the state of the list includes the new node and is consistent.

Recovery during append We use the `toAppend` variable to identify the interrupted action: a non-`NULL` value implies an unfinished append operation. Thus, we need to repeat the critical section of the append. To allow the recovery code to be recoverable itself we use the `lastTail` variable, instead of `tail` used originally (line 7 of Algorithm 1). This resolves the problem of a crash between lines 8 and 10 of Algorithm 1 that would cause the second recovery to reinsert the node.

Removal To guarantee atomicity and recoverability of removals we follow the same principles. We store the node to remove in the `toRemove` variable at the beginning of the critical section, similarly to `toAppend`. To recover, we repeat the removal code which is designed to be safely re-executed. We first check the `toRemove` variable to identify if the system crashed during removal and repeat the process.

ADLL recovery We recover by first identifying the interrupted operation (append or removal) by checking the `toAppend` and `toRemove` variables. Then, we repeat the appropriate operation as discussed.

3.3 Optimizing the log structure

Appending a record to the ADLL through Algorithm 1 requires multiple non-temporal stores and bears overhead due to the write latency and the use of fences. Moreover, writes refer to non-consecutive locations (the list's nodes), which forbids packing them to fewer cachelines. We can significantly reduce the write overhead by changing the memory layout of the data structure by blocking multiple records into fixed-size buckets represented as arrays, as shown in Figure 2. After creating a log record we place it into a bucket with one write, rendering insertion both atomic and cheap. The log is resized by atomically appending new buckets to the ADLL. This layout uses cheap array appends and amortizes the cost of atomic expansion: instead of single nodes, we insert buckets. The recovery algorithms are unaffected by the new structure. The only exception is that we need to keep the next position in the last bucket to insert a new record. Doing so through a non-temporal store would increase the insertion cost. Instead, we reconstruct the information during the analysis phase in the event of a crash. We initialize the cells of each bucket to zero, and, during analysis, we identify the last occupied cell after skipping all empty cells cleared by the log clearing process.

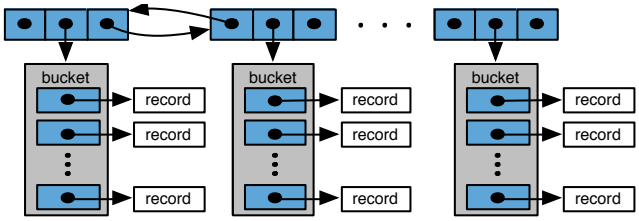

Figure 2: Minimizing the write overhead.

Clearing the log Removing log records from the hybrid structure is more involved due to the need to shift records to fill removed record gaps. Doing this atomically is expensive and adds unnecessary complexity. We avoid it by allowing marked gaps in a bucket, keeping count of occupied cells, and removing a bucket when it becomes empty. We do not explicitly store bucket counts, but, in the event of a crash, we reconstruct them through the marked gaps. We thus simplify record removal but may waste memory in long-running transactions. Under both force policies, the records of long-running transactions can span multiple buckets, thus preventing bucket removal. We can tune bucket size to balance the impact of long-running transactions. Alternatively, we can compact the log if its occupancy drops below some threshold by creating a new log, copying records over, and atomically changing the pointer to the head bucket.

Multiple log records per cacheline A key challenge in keeping user data in NVM is the lack of control over when their updates become persistent, preventing DBMS-like optimizations [10, 27] where the log tail is flushed from memory to persistent storage in batches. This guarantees the persistence of log records and allows the packing of writes in cachelines in NVM, but it is not possible when user data is also in NVM: delaying log writes may cause data writes to overtake their log records, violating the WAL protocol.

In REWIND, we can perform similar optimizations over our hybrid log. Multiple records are packed into a single cacheline since the record pointers are stored in consecutive memory locations. This does not require the log records themselves to be stored together in memory. The compiler needs to reorder the log calls and place them in batches above the corresponding user writes. This guarantees the log writes are not overtaken by user writes and records are placed in one cacheline. With 64-byte cachelines and 8-byte pointers we need just a single fence and a single non-temporal store for every eight log records. This also mitigates the cost of the fence and the group size serves as a tuning knob for adjusting to different fence latencies.

Commit log records can be reordered safely before the user writes as the log records that precede them guarantee recoverability. Even if the commit records cannot be moved, we can move all preceding records and proceed as before. This requires the cacheline be written atomically since we only assume the hardware can guarantee single-word atomic writes. We do this by keeping the position in the bucket up to which log records are guaranteed to be persistent. This is set to zero when the bucket is created. Then, it is updated after we issue a memory fence (using a non-temporal store) with the position of the last record. This guarantees that all log records up to that point are persistent. If a cacheline

is not intentionally flushed, this index is not updated. This is vital for ensuring correctness, as during recovery we only consider log records up to the last persistent index. We issue a memory fence/index update for every $\lfloor |\text{cacheline}|/|\text{pointer}| \rfloor$ records; or when the bucket is full; or when we find an END record. The latter is important since END records mark the completion of commit/rollback. Delaying the update of the last persistent position after a commit may lead to having to abort a completed transaction after a crash.

3.4 Two-layer logging: the Atomic AVL Tree

To improve search in the ADLL we use an auxiliary structure: an *Atomic AVL Tree* (AAVLT), which indexes log records by their identifier and is recoverable by maintaining a log of its internal operations in the ADLL. The most intensive logging activity is during rebalancing on insertion/removal. We use the optimized version of the ADLL. Every update to the AAVLT is only executed by a single thread and forwarded directly to NVM. Doing so allows us to log only the last operation on the AAVLT and clear the log entries after completion, thus reducing the length of the ADLL.

In terms of AAVLT insertion and removal: (*a*) we log all the writes that affect the state of the structure, and (*b*) we delay the de-allocation of the removed nodes until after the successful completion of the operation. The logging and recovery implementation is a simplified version of the recovery scheme of Section 4. We also skip the analysis phase as there is only one transaction to undo. Rollback also largely remains the same with the only issue being locating the next log record to undo. This is straightforward for a normal rollback (the previous entry), but after a crash during the rollback operation itself we need to skip all records that were previously undone and continue from that point. We do this in the same way as in the recovery of one-layer logging (see Section 4.5). Finally, we clear log entries after each AAVLT operation as we describe in Section 4.6 for the force policy.

4. THE RECOVERY RUNTIME

4.1 Transaction recovery management

The transaction recovery manager maintains two structures: the log and the transaction table. The log tracks the program's writes. The format of these records is standard and includes the record ID, the transaction ID, the record type, the old and new values, the address of the memory location modified, and pointers to other records. The transaction table stores information about the active transactions. Transaction table entries include the transaction ID, its status, the ID of the last record of the transaction, and the ID of the record to undo next. The transaction table is constructed during recovery in all configurations but is maintained during logging in the two-layer configuration. There is no need for a dirty page table as NVMs are byte-addressable. The transaction recovery manager constructs the transaction table at application start and determines whether a system or application crash occurred, in which case recovery is performed, or whether this is a clean start.

4.2 Logging

Under the WAL protocol a log record must be persisted before the corresponding persistent write. We use this approach for the ADLL, the AAVLT in the two-layer configuration, and the programmer data (Section 2). We use *physical logging* instead of *logical logging* as it fits better with imperative languages. DBMSs enforce WAL with system calls and a synchronous I/O interface. In REWIND, we must enforce WAL for CPU writes that pass through a complex memory hierarchy and may be re-ordered before reaching NVM. We use a simplified version of the original ARIES log function with the key difference being that the dirty page table is absent, as we do not have pages. For one-layer logging the transaction table is also absent during logging and only reconstructed during recovery. Log records are created given appropriate parameters and then a memory fence is issued to ensure the record fields have reached the memory. After that, the record is inserted atomically to the log. If two-layer logging is used, the record is inserted into the AAVLT and the AAVLT maintenance operations are logged instead. As we discussed in Section 3.3 we can reduce the number of fences required by moving groups of log records before the writes and then issuing a single fence.

4.3 Commit

The log function guarantees that the relevant log records are in NVM upon commit. Under a force policy, all updates of a transaction must be in NVM by the time a transaction commits. We do this by directly updating NVM using non-temporal stores and follow it, at commit-time, with a memory fence and an END log record. We may also then remove the log entries of this transaction. In no-force configurations, all we need is to insert the END log record at commit-time. The log entries of committed transactions are cleared in the background by checkpointing (as we will see in Section 4.6). ARIES follows a no-force policy to improve I/O when writing log pages and dirty pages to disk. In NVM, persisting the log entries is as expensive as making the updates themselves persistent. ARIES uses a steal policy, which in our case is inapplicable as there is no buffer-pool. Committing in ARIES explicitly forces any in-memory log entries to persistent storage. This is not required in REWIND as log entries are immediately made persistent (through non-temporal stores). This is a novel requirement in NVM-based systems to prevent reordering of writes in the memory hierarchy from breaking the WAL protocol.

Memory de-allocation (*e.g.*, line 7 of Listing 1) requires special handling for recoverability. In no-force configurations, we delay memory de-allocation until the corresponding log entry is processed at the next checkpoint (see Section 4.6). The de-allocation details are stored in a special DELETE record. In force configurations, we postpone memory de-allocation until after committing (as in line 16 of Listing 2). We also rely on a DELETE record to handle a system failure between commit and the actual de-allocation.

4.4 Rollback

Transaction rollback in REWIND (either explicitly or as a result of a system failure) proceeds as follows. In one-layer logging, rollback is a trivial backward scan. The situation is more complicated in two-layer logging, where we selectively scan the log for the transaction being rolled back through the AAVLT. The rollback can be repeated an unlimited number of times through the use of CLRs that log undo operations. As we use physical logging, undo sets a variable to its old value. Note that under the force policy the undos should be made persistent as well. This is required to be able to clear the logs after the rollback. One complication is that we need

to redo the last CLR when we recover from a crashed rollback. This protects from the corner case of a crash after the creation of the last CLR but before the corresponding update was made persistent. Finally, we mark the successful rollback completion by writing an END log record.

4.5 Recovery

To recover, we must first recover the log itself. This is followed by either three phases (*analysis*, *redo*, and *undo*) or two phases (*analysis* and *undo*) depending on the force/no-force configuration. ARIES and DBMSs exploit the I/O substrate to present a consistent and persistent log structure in case of system failures during log writes. In our case the log is in NVM. Thus, we require custom mechanisms to recover from interrupted log updates (see also Section 3). When recovery finishes, we also clear the transaction table as all transactions are henceforth considered completed.

After recovering the log, the analysis phase reconstructs the transaction table by scanning the log forward to the point of failure. Then, in the no-force/three-phase configuration only, we scan the log forward again and redo all writes. The redo phase handles a crash during a previous rollback, as it ensures that all undos are redone and consequently not lost during the second crash. In the third phase we consult the transaction table to undo all uncommitted transactions. The undo implementation depends on whether we use one- or two-layer logging as we will discuss shortly. After completing recovery, and under a force policy, we know that all transactions are completed—either committed or aborted. Thus, we clear the log in three steps: (*a*) keep the pointer to the log in a temporary variable; (*b*) create a new log; and (*c*) de-allocate the old log. De-allocating the entire log is faster compared to individually removing its records.

Two-layer logging For each unfinished transaction, we update its status as being aborted and scan its log records backwards by following the `undoNextLogID` pointers: the `ID` of the next record to undo; we retrieve each record through the AAVLT and call the rollback function. Then, we write END records for all aborted transactions. In the force policy, to address the corner case of a crash between the last CLR and the corresponding user write, we redo the last CLR.

One-layer logging This is similar to undo in two-layer logging with two main differences: First, selectively scanning the log is too inefficient so we implement a custom undo process (shown in Algorithm 2) by undoing all uncommitted transactions in a single backward scan. Second, during the scan we track the last CLR (undo) record of all transactions at an unfinished rollback state with the aid of an auxiliary data structure. We use this to skip the UPDATE records that have already been aborted so we can find the next record to undo without using the `undoNextLogID` pointer.

4.6 Log checkpointing

Reducing the size of the log is an important requirement of REWIND as: (*a*) despite their good scalability, NVM capacities will likely lag behind those of disk, and (*b*) the fine-grained logging of REWIND leads to larger metadata sizes. Keeping the log small is critical in one-layer logging to reduce the cost of scanning. The removal of log records depends on the configuration. When forcing, we clear the log records right after a transaction commits/rollbacks. In a no-force policy the records are removed at checkpoints. At a checkpoint, the cache is flushed to make all pending writes persistent. Regardless of the method used, we have to update the log in a recoverable way. We thus atomically remove each transaction's END log record as the last operation to guarantee that, after a crash during clearing, the next attempt will be performed in exactly the same way.

Algorithm 2: Undo operation (one-layer logging) invoked during recovery.

```
1  while ADLLLog.hasPrior() do
2      rec = ADLLLog.prior();
3      xact = transactionTable[rec.xactID];
4      if xact.status = RUNNING or xact.status = ABORTED
       then
5          if xact.status = RUNNING then
6              ADLLLog.insert(xact.xactID, ROLLBACK);
7          if rec.type = CLR then
8              if undoMap.[rec.xactID] ≠ NULL then
9                  undoMap[rec.xactID] = rec.undoLogID;
10                 if force policy then rec.redo();
11         else if rec.type = UPDATE and rec.isUndoable then
12             if undoMap[rec.xactID] ≠ NULL
13                 and undoMap[rec.xactID] ≤ rec.logID then
14                 // extra arguments for CLR record omitted
15                 ADLLLog.insert(xact.ID, CLR, ...);
16                 rec.undo();
17 // Add END records
18 while transactionTable.hasNext() do
19     xact = transactionTable.next();
20     if xact.type ≠ FINISHED then ADLLLog.insert(xact.ID, END) ;
```

To clear the log when forcing, we scan the log backwards and remove the records of committed transactions. A checkpoint under a no-force policy is more complex. It is designed as a "cache-consistent" checkpoint to allow fine-grained locking. This forces a scan of the log, but allows concurrent transactions to keep using the log, which is possible as transactions only append to the log while checkpointing removes records from the middle. We insert a CHECKPOINT record before the cache flush to mark the point in the log that is persistent; all records before that point can be safely removed. We do this by removing END records last. Issuing first the cache flush and then the CHECKPOINT record could lead to newly inserted records appearing to be persistent.

4.7 Concurrency

REWIND allows low-overhead, fine-grained concurrency. We use simple locks to serialize log access and ensure traversals (during a checkpoint) are thread-safe. The one-layer/no-force configuration offers the finest-grained concurrency due to the simple log structure, which allows us to lock the log only briefly during insertion or removal. REWIND could further benefit from a lock-free ADLL but this is left for future work. Thread-safe access to user data by multiple transactions in REWIND is up to the programmer. This is due to REWIND's imperative language nature, which allows the programmer to arbitrarily update data.

5. PERFORMANCE EVALUATION

We implemented REWIND in C++ (using g++ 4.7.3) to evaluate its performance. We used a quad-core Intel® Xeon E5420 clocked at 2.5GHz per core with 12GB of fully buffered DDR2 memory running the GNU/Linux 3.9 kernel. We emulated NVM by adding latency through a busy loop (see also [31]) preceded by a cacheline flush and followed by a memory fence. The latency emulation is inlined before accessing NVM. We consider every non-temporal store as an individual NVM write, but group consecutive writes to the same cacheline into a single NVM write. We set the NVM write latency to 510 processor cycles (150ns). We do

not model a higher NVM read latency than DRAM because: (*a*) the two are already comparable in current NVM technology [25]; and (*b*) transaction processing is update-heavy so writes affect performance the most.

We compare REWIND to Stasis [27], a state-of-the-art storage manager for persistent data structures. Stasis employs data-structure-specific persistence and recovery optimizations as opposed to general DBMS-based recovery mechanisms. We also compare to the popular BerkeleyDB. Finally, we include the version of Shore-MT from [33], which is heavily modified for persistent memory. All approaches work over block devices, so an easy way to port them to NVM would have been to run them on a memory-mounted file system (*e.g.*, RAMFS). We followed a different approach and used PMFS [9]: a kernel-level file system that is memory-mounted and byte-addressable. PMFS guarantees persistence through standard file system calls, but its implementation is optimized for byte-addressability, thus minimizing the overhead over NVM. Thus, it does not adversely impact the performance of Stasis, BerkeleyDB or Shore-MT. We further favor the two former systems by only charging latencies for user-data writes to PMFS and not for PMFS's internal bookkeeping writes. We also favor Shore-MT by disabling any latencies used in [33]. We do not compare to approaches like Mnemosyne [31] or NV-Heaps [5] since they do not provide full logging and recovery functionality and are thus complementary (see also Section 6).

We used BerkeleyDB version 6.0.20 deployed as in [27]. The cache and log buffer sizes matched those of Stasis. The lock manager was disabled to further improve performance. For Shore-MT we used the transaction-level partitioning variant with durable-cache enabled and similar configuration with the other two systems. We refer to the three versions of REWIND as Simple, Optimized and Batch and these correspond to the doubly-linked-list (Section 3.2), hybrid doubly-linked-list (Section 3.3) and hybrid doubly-linked-list with batched log records (Section 3.3) implementations. We configured the Optimized version with a bucket size of 1,000 records and the Batch version with a 64-byte cacheline size and 8-byte pointer size to match our hardware. In Section 5.1 we use Optimized REWIND for all one-layer configurations as this is the configuration used as the bottom layer of the two-layer approach. All results are the average of three runs with standard deviation average of 1.4%.

5.1 Sensitivity analysis

Logging overhead We measure the overhead of logging as a function of the number of memory stores. We implemented a microbenchmark with a single transaction that alternates between updating an in-memory table and performing some computation between updates. The transaction successfully commits at the end. We calibrated the computation cost to be a multiple of the cost of a non-logged store to NVM. In the left plot of Figure 3 we show the logging overhead as a function of the fraction of time spent on updates; the overhead is reported as the ratio between the performance of REWIND and the non-recoverable implementation over NVM *e.g.*, a ratio of 5 means REWIND is 5x slower. We tested all four configurations: two-layer or one-layer logging (2L or 1L); and force or no-force policies (FP or NFP).

The rightmost point of the plots represents the worst case: the user program only updates critical data. Then, the overheads of the two-layer configurations are higher compared

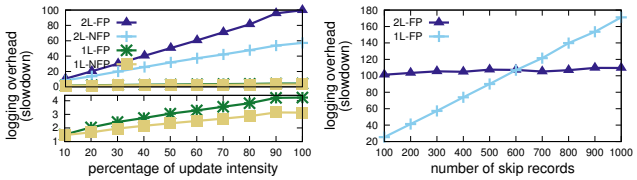

Figure 3: Logging overhead as a function of update intensity (left) and number of skip records (right).

to the one-layer configurations. The low overheads of the one-layer configurations show the effectiveness of the Optimized implementation of REWIND. The difference between the overheads of the two-layer and one-layer configurations stem from the cost of using the AVL tree and maintaining the transaction table. The total overhead decreases steeply as the intensity of updates decreases. For a 10% update intensity, the overall overhead drops to only 1.5× for the one-layer no-force configuration and 8.5× for the two-layer no-force configuration. The difference in logging overhead between the force and no-force policies is not as dramatic, especially for one-layer logging, but it is still significant. To better convey the information we have magnified the plot for the one-layer runs at the bottom of the graph. For one-layer logging the overhead varies between 2% to 35% and for two-layer logging between 24% to 74%. The increased logging overhead of the force policy is due to the more expensive non-temporal writes to NVM for the user updates and from the extra work to clear the log at commit (Section 4.3).

We next focus on the comparison of one- and two- layer logging under a force policy. Recall that commits in one-layer logging require linear scans of the log, which become more expensive with more interleaving log entries (a measure of the number of concurrently running transactions). We term such entries *skip* records, as they will need to be skipped if this transaction is to be selectively processed. Two-layer logging rectifies this through the AVL index. We changed our microbenchmark to generate a variable number of records from other transactions between records of a specific transaction. All transactions update the same in-memory table, so they correspond to the worst-case 100% update-intensive workload of the previous experiment. The number of skip records varied from 100 to 1,000. This might seem like a small number, but recall that REWIND runs in user-mode and in a single application context. Skip records correspond to the number of intervening concurrent updates of a shared resource in a single context (performed, perhaps, by multiple threads), so a smaller number of such records is enough to measure the overhead and sufficient to indicate the performance trends of each REWIND configuration.

In the right plot of Figure 3 we report the overhead of the one- and two-layer configurations as a function of the number of skip records. The overhead is again expressed as the ratio over the performance of the non-recoverable version of the same microbenchmark. In one-layer logging the overhead grows sharply with the number of skip records. In two-layer logging, on the other hand, the overhead is relatively fixed. In reality, it also grows with the number of skip records, but at such a slow rate that it is untraceable in the plot. Even though one-layer logging starts off performing better than two-layer logging, its degradation as the number of skip records grows is so severe that the two-layer

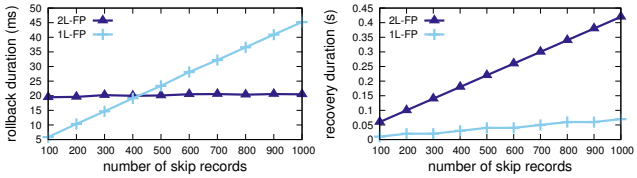

Figure 4: Single-transaction rollback (left) and recovery (right) for a varying number of skip records.

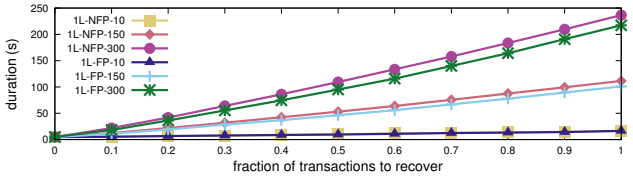

Figure 5: Logging and recovery cost as a function of the fraction of recovered transactions.

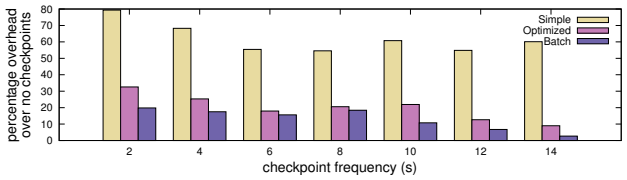

Figure 6: Impact of checkpointing frequency.

configuration outperforms it after around 600 skip records. This suggests that in a user application the decision of which REWIND configuration to employ is not a clear one as there will be a crossover point beyond which the two-layer configuration starts showing its merits. It is up to the user to decide if the concurrency needs of the application are high enough for two-layer logging to be the best choice.

Rollback and recovery costs Our purpose is to assess the impact of the number of logging layers on the performance of single transaction rollback. We use the same microbenchmark as before, but instead of committing the targeted transaction we roll it back. In the left plot of Figure 4 we show the rollback duration (in milliseconds) as a function of the number of skip records for the one- and two-layer configurations and for a force policy. The rollback time of the one-layer configuration grows faster than that of the two-layer configuration as we increase the number of skip records. The two-layer configuration catches up with the one-layer one at around 400 skip records. As was the case for commit, this suggests that the two-layer configuration exhibits its merits after a sufficient number of skip records. Again, the programmer should customize the REWIND configuration for the expected application workload. REWIND itself can be tuned to adapt to various workloads.

Next we report the cost of aborting a single uncommitted transaction during recovery, instead of rolling it back during normal operation, again as a function of the number of skip records. This case appears when a transaction starts its commit protocol, but does not finish committing (*i.e.*, it does not log an END record); it must then be aborted during recovery. This continues the analysis of the choice between one- or two-layer configurations, but in a more contrived scenario. We extended the microbenchmark to commit all other transactions but the target one, but without clearing them from the log so their entries have to be skipped when recovering the target. That could happen if the system crashed after these transactions logged their END logged (so the system will not try to abort them) but before clearing the log. In the right plot of Figure 4 we report the recovery time as a function of the number of skip records for the one- and two-layer configurations and with a force policy. One-layer logging now significantly outperforms the two-layer configuration. Although two-layer logging performs better during the undo phase, and for selective transaction rollback, it is swamped by the slower iteration over the log contents during the analysis/redo phases thus greatly exacerbating the recovery time. This contrasts the earlier results where one-layer logging was outperformed by two-layer logging and reinforces the intricacies of choosing a configuration.

We now report the total processing cost (logging plus commit or recovery) as a function of the likelihood that transactions are recovered. We extended the microbenchmark to select a varying number of transactions to be recovered and timed both the logging and the commit or recovery process of *all* transactions. In Figure 5 we show the total time as a function of the fraction of transactions that need to be recovered, for the one-layer configuration with both force and no-force policies and with three values of skip records: 10, 150 and 300. For both policies we factor out the duration of log clearing to compare the methods irrespectively of whether clearing is immediate or through checkpoints. We do not consider the two-layer configuration as we have already seen it perform worse than the one-layer one in terms of both recovery and logging. The execution time is sensitive to the number of skip records given the dependency between rollback/recovery cost and the value of this parameter, as was shown earlier. Recall that the no-force policy requires two phases during recovery, whereas the force policy requires three. Observe then that the no-force policy has a slight advantage for the same number of skip records and a very low crash probability. It is eventually outperformed by the force policy because of the extra recovery phase. This is more evident as the number of skip records increases because the duration of the extra phase increases as well.

Checkpoint overhead To measure the checkpoint overhead we inserted ten million log records in the three REWIND versions, configured with one-layer logging and a no-force policy. We ran the insertions for each configuration with and without checkpoints and we report the overhead of the checkpointed run as the percentage of non-checkpointed execution for a varying checkpoint frequency. Overall, the overhead declines with decreasing checkpoint frequency. However, the overhead in the Simple version is more severe compared to the other two versions. This is due to the coarser degree of concurrency: the Simple approach needs to lock and serialize the insertion of a new record to the ADLL while the other methods only apply a single update to a bucket. As shown in Figure 6, the overheads of the Simple, Optimized, and Batch REWIND versions vary from 79% to 60%, 32% to 9%, and 20% to 3% respectively.

5.2 Complex transactional workloads

Logging We evaluate the overhead of REWIND when recovering data stored in a B$^+$-tree. We tested eight con-

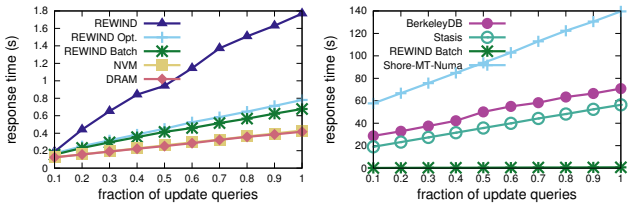

Figure 7: B$^+$-tree logging performance for REWIND *vs.* no recoverability (left); REWIND *vs.* Stasis, BerkeleyDB and Shore-MT (right).

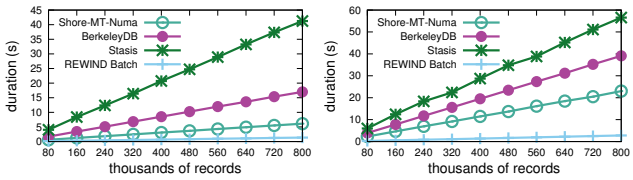

Figure 8: B$^+$-tree recovery for single (left); and multiple transactions (right).

figurations: DRAM, without persistence or recoverability; NVM with persistence but without recoverability; the three REWIND versions running on NVM; and Stasis, BerkeleyDB and Shore-MT running on NVM. The last six configurations guarantee persistence and are recoverable. All REWIND versions were configured with a no-force policy and without checkpoints. We implemented one in-memory B$^+$-tree version for each different persistence layer: REWIND, Stasis, BerkeleyDB, Shore-MT. We loaded the B$^+$-tree with 100k 32-byte-long records and performed a mix of 200k lookups and updates as we varied the read/update ratio. The updates were equally divided between insertions and deletions for a constant tree size per read/update ratio.

In the left plot of Figure 7 we show the total execution time for the workload as a function of the fraction of update queries. The overhead of the DRAM and NVM implementations grows with the fraction of updates, albeit gently, as updates are more expensive than lookups. This is exaggerated in the NVM implementation because of the overhead of NVM writes. All REWIND configurations perform well and close to the DRAM and NVM implementations. The Optimized version improves the Simple version by 27% and the Batch version improves it by 37%. We therefore focus on the REWIND Batch variant from now on. In the right plot of Figure 7 we compare the overhead of REWIND to Stasis, BerkeleyDB, and Shore-MT. REWIND outperforms Stasis by 85×, BerkeleyDB by 105× and Shore-MT by 205× at 100% update queries. This is due to REWIND's minimalistic design, leaner software stack, and NVM-specific optimizations. Shore-MT is outperformed as it is optimized for multi-threaded performance while the workload is single-threaded. In Section 5.2 we show how Shore-MT scales better than BerkeleyDB and Stasis in multi-threaded mode.

Rollback and recovery We report the cost of transaction rollback as a function of the number of operations. We started with a 100k-record B$^+$-tree and then invoked a mixed workload of an equal number of randomly distributed insertions and deletions. This keeps the size of the B$^+$-tree small, but generates a large number of log records. We report the results in the left plot of Figure 8. REWIND Batch outperforms Stasis by 30×, BerkeleyDB by 12× and Shore-MT by 4×. This is due to the REWIND algorithms and its minimal physical fine-grained logging, as opposed to the logical logging of Stasis, or the coarse-grained, page-level logging of BerkeleyDB and Shore-MT. Shore-MT's excellent performance is due to undo buffers keeping the undo log records in memory. In the right plot of Figure 8 we report the cost of full recovery for multiple transactions. We used the same setup but now we created a new transaction every 200 operations. Thus, the number of transactions varies from 400 to 4,000. REWIND outperforms Stasis by 20×, BerkeleyDB by 14× and Shore-MT by 8×. This is due to the lower per-transaction overhead of REWIND and one-layer logging doing away with the transaction table. Coupled with the efficient NVM-specific implementation, the result is a large performance margin over the competition.

Concurrency To test REWIND's fine-grained concurrency, we started multiple threads with each thread performing 100k operations on a B$^+$-tree. Each operation is either an insert/delete pair or a lookup. The lookup-to-insert/delete ratio ranges from 20% to 80% (*e.g.*, 30% lookups, 70% insert/delete). Each thread is assigned a ratio at the beginning and picks up operations from a pool of available tasks. We measured the total duration of the run, *i.e.*, until all threads finished, as a function of the number of threads. REWIND uses its own library-level concurrency mechanisms. For Stasis and BerkeleyDB we let readers progress without locks but use locks for insert/delete pairs. This improves performance for BerkeleyDB as it eliminates deadlocks. For Shore-MT we use its own concurrency mechanisms for up to four threads as it creates one log partition for each core. Beyond that we found it better to use the same locking as Stasis and BerkeleyDB. As shown in Figure 9, the processing times of Stasis and BerkeleyDB grow linearly with the number of threads. Shore-MT as expected scales better than Stasis and BerkeleyDB until the first four threads and then yields similar performance with BerkeleyDB. REWIND scales significantly better after three threads. The processing time for REWIND does not increase monotonically. This is due to the OS scheduling threads to the same core. Although we set the affinity of each task to a different core, the lightweight locking of REWIND results in threads finishing so fast that the OS seems to ignore that hint and schedules threads with different affinities to the same core.

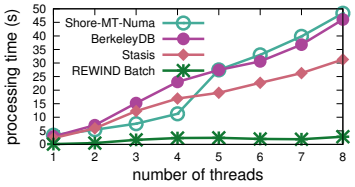

Figure 9: Multithreaded B$^+$-tree logging performance.

Memory Fence sensitivity Memory fence latency varies depending on the storage architecture. We show how we can mitigate its impact by grouping log records. We repeat the benchmark of Figure 7 with the fraction of update queries set to 1 (the worst case scenario). We compare REWIND Optimized, which supports in-place updates solution but no grouping, with REWIND Batch for varying group sizes, *e.g.*, REWIND Batch 8 uses 8 records per memory fence; we also include variations of 16 and 32 records per group.

Our results are shown in Figure 10. REWIND Optimized is affected and is slowed down by 5× while REWIND Batch has a slowdown of 1.63×, 1.32×, 1.18× for group sizes of 8, 16, and 32 respectively. We can therefore mitigate the fence cost of different storage architectures by tuning the group size. We also tested Stasis, a disk-replacement solution, which remained unaffected as expected. These results are in line with previous work [24]. In REWIND the optimizations of Section 3.3 are twofold as they mitigate the cost of the fence and also reduce the write overhead. Due to the lack of pages REWIND does not need to restrict the transactions of the group *i.e.*, force all transactions in the group to commit or abort.

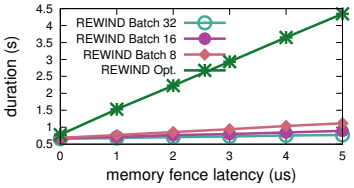

Figure 10: Memory Fence sensitivity.

5.3 TPC-C

We use a variant of the TPC-C benchmark to: (*a*) stress-test REWIND; and (*b*) show that by collapsing the boundaries between the in-memory and the persistent representations we can improve performance by co-designing the algorithms and the physical data layout. We implement the TPC-C schema with B$^+$-trees for table storage and focus on the new order transaction. We use a scaling factor of one and use ten threads on our test machine to simulate the ten terminals issuing new order transactions, which is a slight deviation from TPC-C where a terminal can choose among five types of transactions. However, our goal is to measure the overhead in write-intensive operations and not test the features of a full-blown DBMS. Thus, the new order transaction is the best choice as it is the most write-intensive TPC-C transaction and the backbone of the entire workload. We use four data layouts: standard persistent but not recoverable B$^+$-trees in NVM; naive B$^+$-trees over REWIND; an optimized layout of B$^+$-trees over REWIND to represent compound keys; and the latter with a distributed log [24].

For REWIND with optimized B$^+$-trees, we use an array of B$^+$-trees to represent a table with a compound key. For the order tables (`orders`, `order_line`, and `new_order`), instead of having a B$^+$-tree with a compound key on (`warehouse_id`, `district_id`, `order_id`) per table, we noted that the domains of `warehouse` and `district` consisted of one and ten values respectively as there are ten districts in a single warehouse. Thus, we build an array of ten B$^+$-trees, each on `order_id`. In REWIND, the use of distributed logging is up to the user. Using a single transaction manager for all transactions dictates a shared log; while a per-transaction manager implies a distributed log. This flexibility further enables co-design: through the persistence and recovery guarantees of REWIND, programmers can optimize the data structures and the implementation of transactions.

As per the TPC-C specifications, we abort 1% of transactions. In REWIND these transactions are rolled back while in the standard NVM version they are considered non-recoverable and ignored: this adds a significant overhead to the REWIND B$^+$-tree. We do not compare to other systems as REWIND significantly outperformed them earlier.

In Figure 11, the non-recoverable implementation with naive data structures has a throughput of 273k transactions per minute (tpm). The optimized implementation over REWIND yields a throughput of 197k tpm for a 1.39× overhead. This highlights the potential of co-design: REWIND enables program-level workload-specific optimizations as persistence and recoverability need not be offloaded to a different runtime. Distributed logging improves the throughout even more to 262k tpm and a 1.05× overhead. REWIND with naive implementation of the data structures gives a throughput of 37k tpm for a slowdown of 7.37× over the non-recoverable NVM version. This performance is in line with the microbenchmark results of Section 5.1 and the results of [24] for distributed logging.

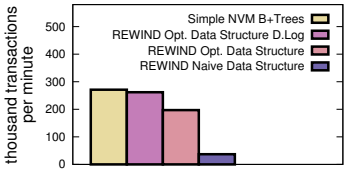

Figure 11: TPC-C throughput.

6. RELATED WORK

Persistent virtual memory [26, 34] has received renewed interest through persistent regions [14]. Such attempts employ block-level I/O devices and file abstractions. Recoverability relies on staging persistence and logging through combining volatile main memory and persistent disk storage. Closer to our work, [19] uses battery-backed DRAM for persisting the file cache [3], but ultimately relies on I/O and uses a coarse-grained region approach to undo logging.

Two recent proposals [5, 31] provide NVM heaps to applications. We leverage this to support in-memory persistent data structures. Both [5, 31] only provide primitives for programmers to create and manage their own recovery protocols. Fang *et al.* [10] propose an NVM-based log manager for DBMSs, which, unlike our approach, relies on a client-server design and uses epoch barriers to guarantee persistence. Giles *et al.* [12] address embedded transaction management in user code, but unlike our work they require custom hardware to force the redo log to NVM before committing, while keeping user updates in a dedicated buffer before persisting the log; they do not elaborate on recovery mechanisms. More recent work [13] has similar goals to REWIND, but does not go as far in addressing recovery and concurrency: it only performs redo logging without in-place updates. Similarly, [35] embeds transaction management in user code but it assumes the existence of a non-volatile cache that it uses instead of logging. Finally, [2] studies the update semantics of NVM data in lock-based code (as opposed to transactional code) and touches only superficially on the mechanisms used for logging and recovery in NVM.

Prior to NVM, researchers proposed battery-backed DRAM and buffer manager extensions to support recoverability [21]. For instance, [6] uses battery-backed DRAM with an ARIES-like protocol, but, unlike us, it still assumes page-level I/O for data and log updates. DBMSs optimized for volatile memory [8, 17] are also relevant. These significantly improve disk-based alternatives but are still suboptimal for NVM as they are subject to the inefficiencies of a block-based design towards durability.

Pelley *et al.* [24], propose distributed logging and group commits for mitigating the memory fence latency in NVM.

These are orthogonal to REWIND and we examine their effects in Sections 5.2 and 5.3, respectively. Similarly, [33] examines how NVM allows practical distributed logging. Unlike our approach, [33] targets page-level data and log updates. We compared this to REWIND in Section 5.2.

Recent work has considered more lightweight data management than full-blown DBMSs. For instance, [27] compares both DBMSs and file systems to custom alternatives; while [15] quantifies the overhead of several DBMS functionalities. There has also been recent interest in similarly extending file systems. For example, [23] presents an extended transactional and recoverable I/O interface for multiple, non-consecutive blocks. These still present a block interface to programmers unlike our byte-addressable approach. Finally, there has also been work on query processing algorithms *e.g.*, [4, 30] for NVM. These differ from our approach as they assume a complete DBMS instead of our programmer-managed data structures.

7. CONCLUSIONS AND FUTURE WORK

New NVM technologies allow programmers to maintain a single copy of their persistent data structures in main memory and access them directly with CPU loads and stores. This renders transactional recovery mechanisms based on block I/O and the separation of volatile and non-volatile data inappropriate. We presented REWIND, a user-mode library that directly manages persistent data structures in NVM in a recoverable way. The library provides a simple API and transparently handles recovery of critical data. Our results show that REWIND outperforms I/O-based solutions at a minimal overhead, thereby providing a promising path toward enabling persistent in-memory data structures.

As this is a fresh research area, there is more work to be done. Our overarching goal is to embed REWIND into a compiler framework *à la* software transactional memory. Further performance benefits will likely come if we implement the basic log structure using lock-free techniques. Another goal is to introduce autotuning so that the system adapts to the workload through monitoring.

Acknowledgments The authors would like to thank the anonymous reviewers for their comments and the authors of [33] for their implementation of Shore-MT. This work was supported by the Intel University Research Office.

8. REFERENCES

[1] Apple Developer Library. Core Data Programming Guide, 2014.

[2] D. Chakrabarti and H.-J. Boehm. Durability semantics for lock-based multithreaded programs. In *HOTPAR*, 2013.

[3] P. M. Chen *et al.* The Rio file cache: Surviving operating system crashes. In *ASPLOS*, 1996.

[4] S. Chen *et al.* Rethinking database algorithms for phase change memory. In *CIDR*, 2011.

[5] J. Coburn *et al.* NV-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.

[6] G. Copeland *et al.* The case for safe RAM. In *VLDB*, 1989.

[7] B. Cornell *et al.* Wayback: A user-level versioning file system for Linux. In *ATC*, 2004.

[8] C. Diaconu *et al.* Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*, 2013.

[9] S. R. Dulloor *et al.* System software for persistent memory. In *EuroSys*, 2014.

[10] R. Fang *et al.* High performance database logging using storage class memory. In *ICDE*, 2011.

[11] P. Felber *et al.* Transactifying applications using an open compiler framework. In *TRANSACT*, 2007.

[12] E. Giles *et al.* Bridging the programming gap between persistent and volatile memory using WrAP. In *CF*, 2013.

[13] E. Giles *et al.* Software support for atomicity and persistence in non-volatile memory. In *MEAOW*, 2013.

[14] J. Guerra *et al.* Software persistent memory. In *ATC*, 2012.

[15] S. Harizopoulos *et al.* OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.

[16] D. R. Hipp *et al.* SQLite Database, 2014.

[17] R. Kallman *et al.* H-Store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2), 2008.

[18] Linux Kernel. Linux Programmer's Manual, 2014.

[19] D. E. Lowell and P. M. Chen. Free transactions with Rio vista. In *SOSP*, 1997.

[20] C. Mohan *et al.* ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1), 1992.

[21] W. T. Ng and P. M. Chen. Integrating reliable memory in databases. In *VLDB*, 1997.

[22] Oracle Corporation. Oracle Berkeley DB 11g, 2014.

[23] X. Ouyang *et al.* Beyond block I/O: Rethinking traditional storage primitives. In *HPCA*, 2011.

[24] S. Pelley *et al.* Storage management in the NVRAM era. *PVLDB*, 7(2), 2014.

[25] M. K. Qureshi *et al. Phase Change Memory: from devices to systems.* Morgan & Claypool, 2012.

[26] M. Satyanarayanan *et al.* Lightweight recoverable virtual memory. In *SOSP*, 1993.

[27] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *OSDI*, 2006.

[28] R. P. Spillane *et al.* Enabling transactional file access via lightweight kernel extensions. In *FAST*, 2009.

[29] S. Venkataraman *et al.* Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, 2011.

[30] S. D. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5), 2014.

[31] H. Volos *et al.* Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.

[32] C. Wang *et al.* Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO*, 2007.

[33] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10), 2014.

[34] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *ASPLOS*, 1994.

[35] J. Zhao *et al.* Kiln: Closing the performance gap between systems with and without persistence support. MICRO, 2013.