SoftWrAP: A Lightweight Framework for Transactional Support of Storage Class Memory*

Ellis R. Giles
Rice University
erg@rice.edu

Kshitij Doshi
Intel Corporation
kshitij.a.doshi@intel.com

Peter Varman
Rice University
pjv@rice.edu

***Abstract*—In-memory computing is gaining popularity as a means of sidestepping the performance bottlenecks of block storage operations. However, the volatile nature of DRAM makes these systems vulnerable to system crashes, while the need to continuously refresh massive amounts of passive memory-resident data increases power consumption. Emerging storage-class memory (SCM) technologies combine fast DRAM-like cache-line access granularity with the persistence of storage devices like disks or SSDs, resulting in potential 10x - 100x performance gains, and low passive power consumption.**

This unification of storage and memory into a single directly-accessible persistent tier raises significant reliability and programmability challenges. In this paper, we present SoftWrAP, an open-source framework for Software based Write-Aside Persistence. SoftWrAP provides lightweight atomicity and durability for SCM storage transactions, while ensuring fast paths to data in processor caches, DRAM, and persistent memory tiers. We use our framework to evaluate both handcrafted SCM-based micro-benchmarks as well as existing applications, specifically the STX B+Tree library and SQLite database, backed by emulated SCM.

Our results show significant benefits of SoftWrAP over existing methods such as undo logging and shadow copying, and can match non-atomic durable writes to SCM, thereby gaining atomic consistency almost for free.

I. Introduction

This paper examines the use of byte-addressable persistent memory (also called Storage Class Memory [13] or SCM) as a replacement for traditional non-volatile storage (hard disks or SSDs) in data intensive applications. A growing number of these applications employ in-memory database technology that operate almost entirely from DRAM (*e.g.* [12], [3], [4], [1]). DRAM-based data access has two main advantages: one is the fast speed of DRAM that enables the high throughputs and real-time response times required of new classes of web applications [21]; second is the fine-grained memory word (or cache-line) accessibility of DRAM, which facilitates applications involving traversing sparse data structures or graph processing. However, the volatile nature of DRAM makes these systems vulnerable to system crashes, often requiring ad-hoc checkpointing techniques to maintain a persistent copy of the data on non-volatile storage. This incurs run-time overheads, and long recovery times following a crash or scheduled maintenance to rebuild in-memory structures.

SCM technologies [13] like Memristors or Phase Change Memory (PCM) raise the possibility of having the best of both worlds: fast, cache-line granularity access of DRAM with the persistence of disk or SSD. This makes it possible to use algorithms and data structures designed for byte-addressable volatile memory without either the programming and performance overheads of blocking and unblocking these structures for disk access, or having to deal with the potential loss of data due to a system crash.

Since SCM is nonvolatile it raises the issues of consistency faced by all storage systems. However, the ground rules change significantly when using SCM. First, the techniques developed by database and operating systems use complex software intervention for buffering and logging to exploit the slow block storage interface and access latencies [18] of traditional storage devices. Secondly, SCM is attached to the memory subsystem and interacts directly with the processor cache hierarchy, leading to new problems caused by uncontrolled cache evictions. In the ideal situation, a program simply reads or writes locations in SCM just as it reads or writes locations in volatile memory, with the assurance that its writes into SCM locations are durable and consistent. Durability refers to the property that a store to an SCM location is not left buffered in a volatile tier (such as a processor cache or a memory controller buffer), while consistency refers to the property that a compound update which spans multiple locations in SCM is not incompletely reflected in SCM after a machine restart. Ensuring durability and consistency is not straightforward for a number of reasons recounted briefly in section II, and in recent years a number of approaches to durable and consistent updates have been proposed. These are summarized and compared with our approach in Section VI.

In this paper, we examine the problem of providing durability and atomicity in a system using SCM for persistence, and describe a novel solution. We present a software-based approach for SCM atomic persistence based on a framework which we call **SoftWrAP** (Software-based Write Aside Persistence). The approach requires no hardware changes, beyond building upon the expectation that store-fencing capabilities of current machines will be extended naturally to fencing writes to SCM. In fact, Intel recently introduced the PCOMMIT instruction [16] that does just this – the instruction retires when pending stores to SCM locations have drained out of

*Supported by NSF Grant CCF 1439075 and Intel SSG.

intermediate buffers. The SoftWrAP framework is designed to be portable, and its performance is amenable to tuning by compiler optimizations. The approach specifically allows an application to benefit from high speed processor caches while simultaneously achieving consistent updates into SCM. We show that SoftWrAP outperforms other methods that perform atomic updates to SCM and approaches the speed of direct writes to SCM that do not guarantee atomicity.

In Section II, we elaborate on the problem and describe our approach in Section III. Performance results of our implementation are presented in Section V.

II. Problem Overview

This work addresses the problem of ensuring that the updates made by a sequence of stores to scattered addresses of SCM are performed atomically, even in the presence of machine failure. A good solution must exploit the processor cache hierarchy to communicate updates within and across transactions. Algorithm 1 illustrates the problems with a simple example. Routine **moveNode** transfers a node between singly-linked lists *freeList* and *workList*. If the store of step 3 has reached persistent memory while 2 has not when the machine fails, then the entire *workList* is unreachable and lost; if only the store of step 2 makes it to persistent memory then the nodes of *freeList* beyond the first become unreachable. Implementation of simple data structures becomes a complicated programming challenge, and redundancy needs to be built in to account for different store ordering scenarios.

Algorithm 1: Atomic Region

```
moveNode(node * freeList, workList);
begin
    Atomic_Begin
    1. temp = freeList -> next;
    2. freeList -> next = workList;
    3. workList = freeList;
    4. freeList = temp;
    Atomic_End
```

There are two complementary problems that arise in trying to achieve atomic writes in the presence of failures: **durability** and **atomicity**. This gives rise to complementary demands on the cache subsystem. Durability requires a guarantee that designated cache lines evicted from the cache *are* actually reflected in the back-end persistent memory. Atomicity requires a guarantee that designated cache lines *are not* evicted and written to persistent memory at arbitrary times.

Durability requires that writes made by a program have actually been committed to the non-volatile medium. However, the default semantics of processor store or cache flush instructions makes no guarantees of when the update will actually be reflected in the backend device. This is not an issue in normal programming models since the coherence mechanisms and fencing instructions ensure value propagation of the updates without the need to consult the backend volatile DRAM memory. In this situation a machine restart can result in the loss of an update even if the backend memory were non-volatile. To address this gap, processor manufacturers are providing *persistent fence* instructions (*e.g.* Intel's recent PCOMMIT instruction [16]) that guarantee pending stores have been successfully committed in a power-safe domain. A new flushing-write instruction CLWB would allow a designated cache line to be written to memory without evicting it from the cache. Together it is thus possible to perform a durable write of a single cache line by a sequence of CLWB, SFENCE, and PCOMMIT instructions following a store.

The problem of atomicity is complicated by the complementary problem of eager or premature cache evictions. The cache subsystem evicts cache lines to the back-end memory autonomously based on its specific cache management policies. There are no guarantees regarding the order in which these evictions may occur. At the time of a machine crash, an arbitrary subset of an atomic sequence of stores may have been written to the memory while the remainder are held back in volatile cache and memory buffers. On restart it is impossible to know which of the persistent memory locations correspond to new and which correspond to old data. Hardware changes to the front-end cache hierarchy to control cache evictions [9], [29], [32] have been proposed for this problem. The solutions in [9], [29] control the order of cache evictions by tagging the cache blocks to prevent spurious updates. In Kiln [32], the cache controller tracks the progress of atomic sequences and reflects the state (in-flight or completed) in the updated cache lines; the state is also communicated to a back-end non-volatile cache that buffers evicted cache lines until it is safe to update the persistent memory locations. Alternatively, software may employ additional metadata (*e.g.*, a log or journal) also in non-volatile memory, to track which sequences completed and which did not, and use that metadata for effecting recovery. This approach is used in [10], [14], which extends the backend cache hierarchy by interposing a volatile victim cache between the processor LLC (last-level cache) and persistent memory to block cache evictions and uses the log to move updates safely to the persistent memory locations. The work presented here does not require any changes to the processor cache hierarchies. Since the hardware changes proposed are significant disruptions of the existing well-understood cache subsystems, their incorporation into real systems is speculative and will certainly not be available in any form for the near future. Hence we seek a solution that does not rely on changes to existing cache mechanisms. The only hardware support assumed is the PCOMMIT instruction that as mentioned had already been announced by Intel [16].

III. SoftWrAP Approach

In this Section, we describe key implementation issues of Software-based Write-Aside Persistence, or *SoftWrAP*, a library to provide atomicity for a sequence of persistent-memory writes that may be interrupted by a machine restart or failure. Algorithm 2 shows an example program fragment. It comprises a single atomic region using two static persistent variables x and p and a dynamically allocated region of persistent memory obtained by a call to *pmalloc*, a persistent memory version of the usual *malloc* function. The programmer identifies the atomic region within *wrapOpen* and *wrapClose* tags. The programmer (or a preprocessor) translates accesses to persistent memory within the atomic region to calls into the SoftWrAP library as shown in the comments. The wrapOpen call marks the beginning of an atomic region that ends with a wrapClose. The atomic region will also be referred to as a *wrap* and a call to *wrapStore* will be referred to as *wrapping a variable*. In our current implementation the variables in the atomic section must be manually wrapped by the programmer. Compiler assisted wrapping is part of our future work.

Algorithm 2: Programmer annotated atomic region

```
// x, p and pmalloc array are persistent.
wrapOpen();
begin
    x = 1;              |wrapStore(&x, 1);
    ........
    p = pmalloc(100);
                        |temp = pmalloc(100);
                        |wrapStore(&p, temp);
    ........
    for (i = 0; i < 25; i++)
    begin
        p[i] = i;       |wrapStore(p+i, i);
    .........
wrapClose();
```

The basic idea in SoftWrAP is to simultaneously propagate updates made within an atomic region along two paths: a foreground path through the cache hierarchy that is used for value communication within and between wraps, and an asynchronous background path to persistent memory to log the updates. By creating these two paths, SoftWrAP effectively decouples value communication for transaction execution from persistent memory logging for recovery.

The SoftWrAP approach has three logical components: *logging*, *alias handling*, and *retirement*. The *logger* maintains a sequential log in persistent memory that is updated using efficient, cache-line-combined streaming writes. The log is only used to recover from a crash. During normal operation the log is updated efficiently in an append-only fashion using cache-line combined writes, and periodically pruned by deleting entries of retired transactions (*i.e.*, those whose updates have been retired to their home persistent memory locations). To handle the problem of spurious cache evictions described in Section II, SoftWrAP employs a software *aliasing mechanism*. This redirects updates to persistent memory locations by wrapStore to stores to locations in a managed area of DRAM referred to as the *shadow DRAM*. The stores to shadow DRAM allow the updates to be freely communicated via the cache hierarchy (as is done for normal variables) but uncontrolled cache evictions can do no harm. The *retirement* component copies the values of aliased variables from the shadow DRAM to the persistent memory home locations when it is safe to do so. This step is performed in the background, asynchronously and concurrently with foreground transactions. When a portion of shadow DRAM has been retired it can be reused, and all logs records from retired transactions can also be deleted.

Figure 1 shows a contiguous region P of persistent memory that is mapped into the virtual address space V of the application. The shadow DRAM is mapped to a different range of the address space V' that is a small fraction of the size of V. V' only maintains the recently updated wrapped variables in persistent memory and is regularly emptied by retiring its contents. An access to persistent address a is redirected to address a' by the aliasing mechanism. Following the first access to a until it is retired, reads and writes to a are done from location a'. If evicted from the cache, the updated value of a updates only the shadow DRAM location $\phi(a')$ rather than its persistent home location $\phi(a)$. Thus, value communication takes place via the cache hierarchy using the aliased location (primed variables). The record of updates is streamed to a log area in persistent memory asynchronously and concurrently in the form of log records. The figure shows that following the wrapped write a = 5, the cached value 5 is backed by a DRAM address $\phi(a')$ corresponding to the aliased address a' and the redoLog stores a copy of the new value as the record $(\phi(a), 5)$.

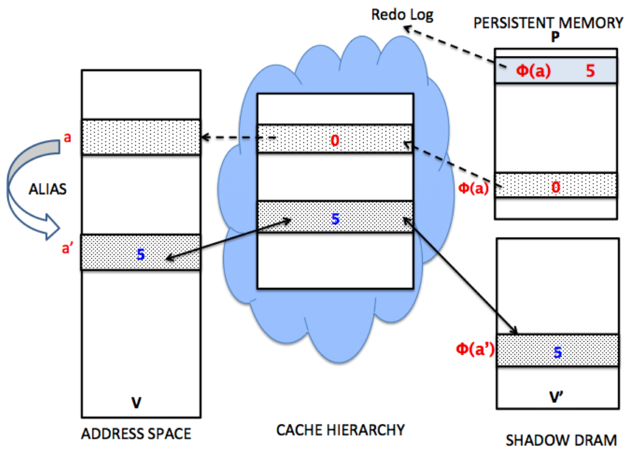


Fig. 1: Updates in atomic region using SoftWrAP

IV. SoftWrAP API and Implementation

In this Section, we describe the basic SoftWrAP functions used to operate on groups of variables and objects atomically as in Algorithm 2.

Algorithm 3 presents a basic implementation of the SoftWrAP library. The data structure to implement the alias table is a hash-table based Key-Value store. A non-blocking implementation is used to avoid excessive locking and unlocking when multiple threads access the table, as discussed later. For scalar variables, the alias table itself stores the latest value of the variable, while for large objects only the pointer to the copy of the object is stored in the alias table. Figure 2 (see

Hash Table A) shows scalar persistent variables M, Z, N with values 1, 3 and 2 respectively in the alias table along with a size qualifier. In Hash Table B, object *A* (a 1K page) is stored in the alias table as a pointer to the shadow DRAM area where the copy of A is stored.

Wraps may be nested. For instance, an application may require that the insertion of an element into a B-tree data structure and the deletion of some other element to be performed transactionally by enclosing the operations in a wrap. The persistent B-tree operations may themselves need to be wrapped to maintain data structure integrity while multiple internal tree pointers are updated. In our implementation nested wraps are subsumed by the calling wrap. Hence calls to wrapOpen and wrapClose increase and decrease the nesting level by 1 respectively. A wrapClose with nesting level 0 indicates the closing of the top-level wrap and requires the log to be made persistent in SCM.

The SoftWrap API is shown in Algorithm 3. The outermost *wrapOpen* call registers a thread-specific handle to the alias table and to a RedoLog area allocated for the wrap. Subsequent nested calls by this thread are simply rolled up to its outermost call, since atomicity needs to be preserved at the outermost level. The *wrapLoad* and *wrapStore* calls are used for reading or writing scalar persistent variables within the wrap. The wrapStore inserts or updates an existing entry for the variable in the Alias Table with the new value being stored. It also appends a log record with the persistent memory address and value of the variable to the end of the redo log bucket for this thread.

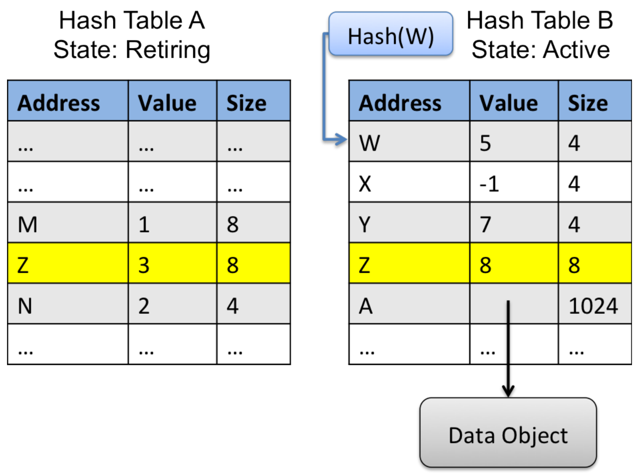
Additionally, *wrapRead* and *wrapWrite* are used to read and write objects. This is useful for applications that read and write data in large extents. These objects are allocated space in the shadow DRAM and accessed indirectly via pointers in the alias table. To simplify space management and support legacy database applications, objects are broken up into units of fixed-size pages (we use 1KB pages but this is a tunable parameter), and one entry is maintained per page in the alias table. A *wrapWrite* whose destination spans multiple pages that have all been already inserted in the alias table simply updates the data pages in shadow DRAM. Otherwise, if the data size being written spans an entire page, it is written to a newly allocated DRAM page. The worst-case occurs if the new data spans only part of a newly allocated page. In this case the updated data needs to be merged with missing bytes from persistent memory. A record containing the new data being written is also appended to the redo log in persistent memory.

Algorithm 3: SoftWrAP API

```
wrapOpen (options $o = default$)
begin
    if WrapNestingDepth == 0 then
        Acquire handle to AliasTable;
        Open RedoLog;
    WrapNestingDepth += 1;

wrapLoad (address $a$)
begin
    if Entry for $a$ in AliasTable then
        Return value from AliasTable;
    Return value from SCM address $a$;

wrapStore (address $a$, value $newVal$)
begin
    if No entry for $a$ in AliasTable then
        Add $(a, newVal)$ to AliasTable;
    else
        Update $(a, newVal)$ in AliasTable;
    Append $(\phi(a), newVal)$ to RedoLog;

wrapRead (address $src$, size $n$)
begin
    Break up $n$-byte address range of $src$ into
                aligned page sequences $S_i$;
    for (each page $S_i$) {
    if Entry for $D_i$ in AliasTable then
        Return $D'_i$;
    Return $D_i$;
    }

wrapWrite (address $dest$, address $src$, size $n$)
begin
    Break up $n$-byte address range of $src$ and $dest$ into
        aligned page sequences $S_i$ and $D_i$ respectively;
    for (each page $D_i$ in the destination) {
    if (No entry for $D_i$ in AliasTable) then
        Allocate Shadow DRAM page $D'_i$ for $D_i$;
        Add $(D_i, D'_i)$ to AliasTable;
    Update page addressed by $D'_i$ from $S_i$ // Write DRAM
    Append $(\phi(D_i), D_i)$ to RedoLog;
    }

wrapClose ()
begin
    WrapNestingDepth -= 1;
    if WrapNestingDepth == 0 then
        // Commit all log records to persistent memory
    PCOMMIT;    // Stall until pending persistent memory
                   writes are committed to SCM
```

A. Alias Table Design

The alias table is the key data structure in SoftWrap and needs to be managed carefully for performance. We implemented the alias table as a double-buffered hash table based key-value store that supports *update* and *lookup* operations. Entries are never deleted from the table so we do not need to support a delete operation. Instead, the entire hash table is recycled after the home SCM locations of the variables are updated from the alias table. This permits a scan-based, thread-safe non-locking implementation [27] that simplifies the design and improves the performance significantly. To lookup an address p, the table is scanned starting from the index computed by $Hash(p)$ until either p is found or the scan encounters a blank entry in the table. In the first case an update operation can simply rewrite the value field of the



Address	Value	Size
...	...	...
...	...	...
M	1	8
Z	3	8
N	2	4
...	...	...

Address	Value	Size
W	5	4
X	-1	4
Y	7	4
Z	8	8
A		1024
...	...	...

Fig. 2: The Alias Table implementation for global aliasing is a double-buffered lock-free hash table implementation. It handles reads and writes to both primitive data types and object data and can retire directly from the table.

existing entry. In the second case it must fill in the address (key) and value fields of the blank entry. A simple compare-and-swap test of the single entry just prior to the update is sufficient to prevent races. The table does not need to be locked nor does one need to lock extended code sequences.

The home locations in SCM of written variables need to be updated to their updated values and the alias table memory freed for new entries. This retirement process could be performed either from the values in the logs or from the alias table. Retiring updates from the log requires reading the log and then writing to SCM, while the latter approach can stream DRAM-resident data in the alias table to SCM using efficient memory scatter instructions. We use alias table based retirement in our design. For atomicity, it is necessary that only alias table entries from closed wraps be retired.

To permit retirement of the alias table entries in the background along with foreground activities (new wrap openings and closings, and wrap reads and writes) we use a two-table double-buffered approach. At any time one table is the active table and the other is being retired to SCM. However, we need to be careful to avoid races or unnecessary locking in implementing such an approach.

Figure 2 shows the two-table design. In the figure there are two hash tables A and B; B is the currently *active* table while A is *being retired* to persistent memory. Additions of new alias table entries are only made to the active table. However lookups must consult both tables until we are sure that the latest value of a variable has been written to SCM, at which time the lookup can be made from its home location. In the figure, a store to variable W will lookup the active hash table B and either update the existing entry or add a new entry to B. A load of W will look up table B first, and if found there (as in the figure) return its value. On the other hand, a load of M will fail in table B, and must be followed by a lookup of table A. If a variable is not in either hash table, then it then it must be retrieved from SCM, which is guaranteed to have its last updated value.

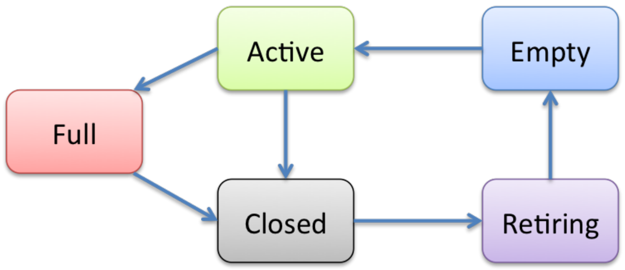


Fig. 3: The lifecycle of Hash Table states within the Alias Table structure used for global aliasing.

The complete design of the two-table Alias Table design requires additional states to be maintained for the tables as shown in Figure 3. Each table can be in one of five states: Empty (E), Active (A), Full (F), Closed (C) , and Retired (R) with the following semantics. In the E state the table has no valid entries and is available for use. A table in the A state will be used for making updates and will be given priority in lookups. In the F state the table will not accept new update operations. When all wraps that were opened while this table was active complete, the table transitions to the C state. In the C state, it is safe to begin retiring the table entries to the SCM. During this time lookups of the table from concurrently executing wraps can continue safely and without conflict. When the retirement of the entries to SCM is complete, the table transitions to the R state. In this state no further lookups will be permitted; instead all lookups that are not found in the active table will go to SCM. The table remains in the R state until the last of the wraps that may have started looking up the table (before it entered the R state) completes. At that point the table can be safely deleted or recycled by entering the E state.

The state transitions are maintained by $wrapOpen$ and $wrapClose$ functions. Wraps are tagged with one of four colors indicating the global state of the two hash tables when the wrap was opened. Suppose X and Y denote the two tables. We need to keep track of the active table at the time of opening a wrap since this can only be retired once these wraps have closed. Similarly, an arriving wrap needs to distinguish whether the non-active table has entered the R state or not, since the table cannot be deleted until all the wraps that could read it have closed.

On an OpenWrap, the wrap is paced in one of four sets $A_X R_Y$, $A_X \bar{R}_Y$, $A_Y R_X$, and $A_Y \bar{R}_X$. The subscript X or Y denotes a table, A indicates the active state, and $\bar{R}$ means *not* in the R state. A counter of the number of wraps in each of these sets in maintained. On a $wrapOpen$ at nesting level 0 the state of the tables is used to tag the thread, and the appropriate counter incremented. When a wrap at nesting level 0 closes, the counter corresponding to its tag is decremented. For active table X to transition to state C it requires that all wraps that were opened while X was active should close. That is the number of wraps in the set $\{A_X R_Y \cup A_X \bar{R}_Y\}$ is 0. Similarly, to transition from the R state to the E state, the number of wraps in the set $\{A_Y \bar{R}_X\}$ is 0.

B. Long running transactions

Even though it may be an extremely rare case, a very long running transaction can exhaust all shadow DRAM space. Anticipating the possible corner case, a wrap manager thread may detect a long running transaction by periodically comparing the elapsed time of each open wrap to a user defined threshold. If the elapsed time exceeds this threshold, then the thread first attempts to speed the long running wrap along by preventing new wraps from opening. If unsuccessful, on a transaction abort or timeout, the alias table is cleared and all logs of successfully completed wraps are replayed - reading the log from SCM and copying the variable in the log to its home SCM location. Once this process is complete, the system is released back into normal operation.

C. Restart and Recovery

Recovery after a sudden failure and restart is simple, and it proceeds as follows. The manager thread replays the redo log from a previous consistency point. Stale wraps are discovered (ones that never closed), and their log records are bypassed as SCM locations get updated by a recovery thread.

D. On Aliasing Alternatives and Relation to Isolation Models

Different implementation alternatives of the basic aliasing scheme described in subsection A are possible. First, as an alternative to creating aliases in DRAM, one could instead simply alias a variable to its copy in the redo log record, which requires trading away the cache efficiency (achieved by treating the log records as non-temporal and not caching them). That is, redo log records would now need to go through the cache hierarchy to allow fast path communication, and cache misses resulting from evictions of these records would have to access the slower SCM rather than DRAM. That has the potential to degrade performance when the cache pressure is high and variable reuse is frequent. Also, the aliased location will change as different transactions access the variable and redirect it to their private log locations. Frequent updates will cause increased coherency traffic as hash table entries are repeatedly invalidated.

The SoftWrAP aliasing approach is isolation agnostic: if the programmer chooses, she may choose to permit dirty reads or phantom reads by allowing one thread to read modified values from another thread's wrap operations even if the second thread has yet to reach a wrap close point. One optimization that is possible under the common model of strict isolation is to buffer up all updates in a local alias table and then flush them using efficient streaming or AVX VSCATTER operations from the local alias table to home locations in SCM. Thus, value propagation within the wrap proceeds through local aliasing, but once the wrap closes with a log commit, the local alias table can be immediately reclaimed and values propagate normally through caches without aliasing.

The framework allows programmer to choose– global aliasing (default) for more algorithmic flexibility and local aliasing for the common strict isolation case. A final consideration concerns the mapping of a shared persistent object in the address space of multiple threads. As is commonly done for shared libraries, in this implementation we assume a fixed mapping based on common agreement, in preference to more costly dynamic alias conflict handling mechanisms.

Ordering between concurrent transactions is achieved at the transaction level by ensuring the same isolation that a developer employs for controlling concurrent execution independent of persistence of memory. Ordering among updates within a transaction needs to be reflected as all-or-nothing in its effect across a machine failure. This is achieved by ensuring that data updates made by the transaction are kept from appearing at their home addresses until the write-aside log has been committed and flushed to NVM. Since the write-aside log either commits or does not commit at the point of a machine/software crash, either all updates in the same transaction are committed (independent of their intra-transaction order) or none are committed. Data writes can thus be held up in caches and flow to backing NVM medium in arbitrary order, but they are visible to software threads (via cache coherence) in transaction order.

V. Evaluation

In this section we describe the evaluation of SoftWrAP using micro benchmarks and two applications: STX B+Tree library, and a SQLite database running a scaled TPC-C benchmark. We will compare our SoftWrAP approach with two other methods referred to as *Non-Atomic* and *Undo Log*.

In *Non-Atomic*, all the stores in a wrap are made durable in SCM before the next wrap can begin. This ensures the atomicity of completed wraps, but a machine failure during the execution of a wrap can leave SCM in an inconsistent state. This approach provides a lower bound on the performance of methods which provide both durability and atomicity. In *Undo Log*, the idea is to directly update SCM just as in *Non-Atomic*. However, to preserve atomicity of all wraps, the current values of the variables before the update are made persistent in SCM using an undo log. In the case of failure the state can be rolled back to the state at the start of an incomplete wrap by replaying the undo log. At the end of the wrap all updated values are retired to their home location by a persistent commit operation.

In *SoftWrAP*, updated values are written to DRAM shadow memory and to a redo log. However, there are two main differences from Undo Log which improve its performance remarkably. First, the log records in SoftWrAP can be write-combined into compact cache lines and streamed to the log asynchronously. In contrast, *Undo Log* needs to synchronously write the current value of each variable to the undo log before it can update its value. Secondly, Undo Log persists all its updates at the end of the wrap to guarantee the durability of the completed wrap. SoftWrAP merely writes its updates to DRAM shadow memory (the alias table), and only needs to persist the much more compact sequential log structure when the wrap closes. The updates of the persistent memory locations are performed asynchronously in the background directly from the alias table.

As our experiments will show this has two performance consequences. First, both the throughput and response time (time from OpenWrap to the end of CloseWrap) in SoftWrAP will be significantly better than Undo Log due to the asynchronous write of a smaller number of cache-line write-combined log writes and the asynchronous writing back to the home locations. Secondly, the throughput of SoftWrAP will be slightly less than Non-Atomic because the latter does not need to write any metadata log records. However, the response time will be slightly better than Non-Atomic because with SoftWrAP, retirements are performed asynchronously in the background while Non-Atomic needs to compete the retirement synchronously before closing.

Table I shows the number of SCM writes and pcommit operations for a wrap of n word-sized stores using the three methods. Non-Atomic requires the n words (to scattered SCM locations) to be written to SCM a single pcommit instruction to ensure their persistence. Undo Log requires a pcommit after writing each log record and another at wrap close to persist the updates. Each of the n log records (a record is 3 words long) generally require 1 cache line write, and each of the n updates require another write, for a total of $2n$ writes. The additional terms in the expression are 1 write for an end-of-log marker and a correction term to account for splitting of a log record at a cache line boundary. Cache lines are 64 bytes. Finally, SoftWrAP can close the wrap after writing the $3n$ consecutive words (12 bytes) that make up the log, followed by a single pcommit. Due to write combining, this results in $12n/64$ cache line writes.

	SCM Writes	*pcommits*	Estimated Time
Non-Atomic	n	1	$nT_w + T_s$
Undo Log	$2n + 1 +$ $\lceil 12n/64 \rceil$	$n+1$	$n(2T_w + T_s) +$ $\lceil 12n/64 + 1 \rceil T_w + T_s$
SoftWrAP	$1+$ $\lceil 12n/64 \rceil$	1	$T_w + max(n * T_{alias},$ $\lceil 12n/64 \rceil T_w) + T_s$

TABLE I: Time to perform a wrap of n 4-byte words .

A. Experimental Results

In this Section, we describe our experimental results with SoftWrAP. The absence of readily available systems with persistent memory raises challenges in evaluating the performance of SCM-based software. We opted for an approach similar to that used for Mnemosyne [30] based on measuring the execution time of a running application that has been instrumented to add a delay to specific types of persistent memory stores. A description of the emulated system model and comparison of its predicted and measured behavior is presented in Section V-B. This is followed in Section V-C by microbenchmark experiments to demonstrate the behavior and advantages of **SoftWrAP** as predicted in Table I. In Section V-D, we discuss the evaluation of the STX B+Tree library, and in Section V-E evaluate a SQLite database running a scaled TPC-C benchmark.

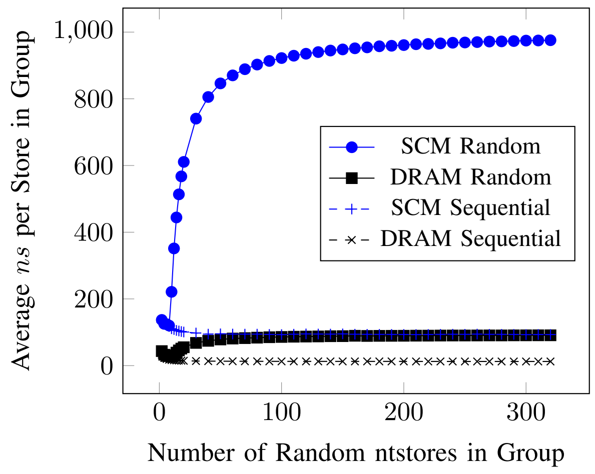


Fig. 4: Average time for an *ntstore* for different group sizes. Write Buffer size: 8 cache-line entries, Cache Line size: 16 words, SCM write delay: T_w=1μs.

B. Model Validation

The streaming non-temporal store instruction (**ntstore**) allows a consecutive sequence of bytes to be combined into cache-lines and streamed to memory without disturbing the cache. The emulated memory system to handle *ntstore* operations has a write-combining buffer implemented by an 8-entry, cache-line-wide queue. Stores to the same cache line are combined into a single entry and written to memory in a single cache-line write operation. The time for a cache-line write from the head of the buffer is denoted by T_w. In our experiments this is set at $T_w = 1\mu s$, a conservative write time for SCM [17].

On an *ntstore* operation, a run-time routine is invoked to emulate the memory system. The routine waits until there is space in the write buffer and then adds the request to the queue while recording the insertion time, if possible combining it with a pending store to the same cache line. Consequently, when a group of sequential writes are made in quick succession, they are write-combined into a single cache-line effectively increasing the data that can be written in a single memory operation by a factor of 16 over scattered single-word writes (cache line size is 64 bytes and word size is 4 bytes). Space in the write queue is created by removing entries that would have completed writing to SCM based on the position in the queue, insertion time, and T_w. A pseudo-instruction *pcommit* is introduced to emulate the forcing of queued requests to the memory, which will behave similar to the PCOMMIT instruction recently introduced by Intel. A *pcommit* call is implemented by inserting a software delay loop until the completion time of the last queued request. The delay loop is bookended by CPUID instructions to force the serialization of the delay with respect to the rest of the program.

B.1 *Validating model of* ntstore *and* pcommit

Our tests are performed on an Intel(R) Xeon(R) CPU x5660 at 2.80GHz with 64GB of DDR3 system memory running Red Hat Enterprise Linux Server 6.5. The SoftWrAP implementation and benchmarks were built with GCC 4.4.7. First we tested the implementations of the streaming non-temporal store (*ntstore*) and persistent memory sync (*pcommit*) operations, which are used to emulate the writes to SCM. A group of n 4-byte writes using *ntstore* instructions in a tight program loop was run and its execution time recorded. The average time for a store in the group is computed and reported in the plots in Figure 4. The writes in a group are either to random addresses within a large array or to sequential array locations that can exploit write-combining. Two backing devices were tested: DRAM and emulated persistent memory. For the DRAM-based experiments, the *ntstore* was streamed directly to memory bypassing the emulated write buffer, while for the persistent memory experiments, the time for an SCM write T_w was set to $1\mu s$.

The average time for a store in a group is plotted for different group sizes in 4. Once the write buffer is full, the writes are written at the average rate of T_w. The measured average write time becomes steady at about $100ns$ for DRAM (a reasonable measured value for T_w for DRAM writes) and, as predicted, to the delay $T_w = 1000ns$ for emulated SCM. When the group is made up of sequential writes, they are write-combined into a single cache-line, effectively increasing the data that can be written in a single memory operation by a factor of 16. The reduced time for sequential writes is also shown empirically in Figure 4. Note that the drain rate of DRAM for sequential writes is so fast that the measured delay reflects the instrumentation overheads. Nonetheless, the experiments show that our delay model provides a good emulation of the SCM write performance for use as a basis of comparison.

B.2 Comparison of analytical and experimental performance

We executed $100,000$ consecutive wraps, each consisting of a group of n $4-byte$ writes. The arrival rate of the wraps was fixed at $10,000$ wraps per second. Writes were made to random word addresses in a large 1GB array to avoid effects of data reuse and caching. The size of each hash table is 8K entries, and an entry is made up of an 8-byte address field, an 8-byte data field and a 4-byte size field. Flushing of the hash table to SCM begins when the table has 500 entries. A sparse table allows rapid insertion and lookup operation, but is large enough to amortize the overheads of creating and retiring the table. The performance as a function of cache size is discussed in Section V-C below.

We recorded the number of writes to SCM, total number of *pcommit* operations, and the average execution time for each method, for *n*=10, *n*=15, and *n*=20. A comparison of the predicted and experimental times are shown in Table II. There is close agreement between them for Non-Atomic and Undo Log. For SoftWrAP there is an additional unknown variable T_{alias}. Computing this value from the experimental data for $n = 10$ we use it to predict the execution time for the other values of n and get a good correspondence.

The number of *pcommit* and *ntstore* operations are also measured during the experiment. and are shown (per wrap) in Figure 5. For Non-Atomic and SoftWrAP there is 1 *pcommit* operation at wrapClose, while Undo Log performs (n+1) *pcommit* operations. The retirement of the Alias Table only requires 1 pcommit every time a hash table is written to SCM, which occurs at a fixed fraction (determined by the retirement threshold) of the incoming transaction rate. Likewise, the number of SCM stores for Non-Atomic and Alias Table processing is n as shown. SoftWrAP reduces the number of stores by a factor of 16 while Undo Log has roughly twice the number of ntstores compared to Non-Atomic. The experimental results match the predictions from Table I.

	n	$Est(ns)$	$Exp(ns)$
Non-Atomic	10	10,220	10,226
	15	15,220	15,223
	20	20,220	20,251
Undo Log	10	25,430	25,653
	15	37,530	37,198
	20	48,630	48,701
SoftWrAP	10	5,720	6,043
	15	7,970	7,956
	20	10,220	11,049

TABLE II: Performance of Array Update Benchmark. $T_w = 1\mu s$ and $T_s = 220ns$. T_{alias} calculated is $450ns$.

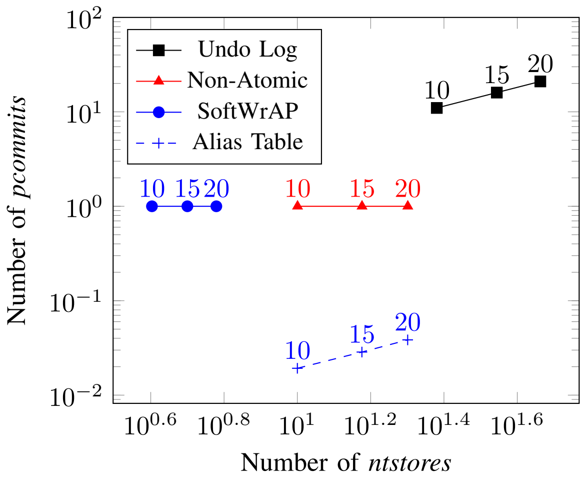


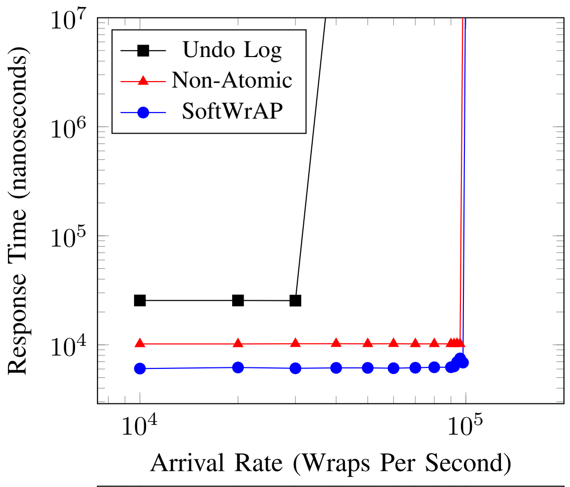
Fig. 5: Average number of pcommits and ntstores per transaction of sizes n =10, 15, and 20 after performing 100,000 transactions.

C. Micro-Benchmarks

In the following experiments, we create a large 1GB array and perform a number of experiments using the same machine setup as described above. We test the wrap response time and throughput and their dependence on the size of the alias table. Additionally, we determine the sensitivity to SCM write time and number of elements in a wrap. Finally, we examine reuse of data across wraps, and show additional benefits from the caching done by SoftWrAP.

C.1 Response Time and Throughput

First, we consider wraps made up of 10 random updates to the array. We insert a tunable delay in-between transaction



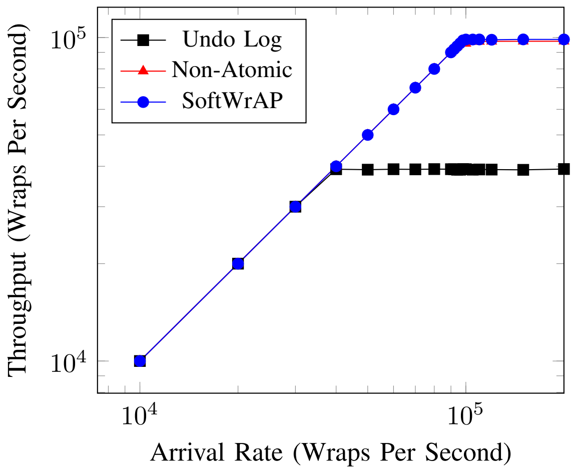


Fig. 6: Response time and throughput for varying arrival rates of transaction size n=10, alias table size of max 8k entries, and SCM T_w=1μs.

arrivals to simulate different arrival rates. The transaction arrival time and completion times are recorded along with the throughput. The arrival rate is varied from $10,000$ to $200,000$ wraps of $n = 10$ elements each per second. The results are shown in Figure 6 on log-log plots to include the performance of Undo Log. The response time at low arrival rates for Non-Atomic is slightly more than $10\mu s$. By comparison, Undo Log has more than double the response time, and SoftWrAP is much faster. The times follow the service times of the wrap as in Table I at low request rates. When the arrival rate exceeds the service rate, all methods show a sharp increase in response time, eventually increasing to infinity, and a leveling off in throughput. SoftWrAP can absorb a much higher arrival rate before breakdown. Before the knee, the alias table can be retired in the shadow of the foreground operations, without slowing the latter. However, ultimately the rate at which the Alias Table is filled exceeds the rate at which it can be retired, and the wraps stall waiting for alias table to become free.

The maximum throughput of both Undo Log and SoftWrAP is slightly less than $100,000$ transactions per second. The maximum throughput of Undo Log is only about $39,000$ transactions per second. As discussed previously, each wrap in Undo Log is slowed down since it does significantly more writes than the other methods. Non-Atomic must perform a *pcommit* operation after each wrap (group of 10 writes), therefore performing slightly less than the maximum possible throughput of $100,000$ transactions per second based on memory bandwidth alone. SoftWrAP isn't limited in the writing to the table, but its throughput is limited by the time to retire the alias table. It is an interesting property that at reasonable arrival rates in a write-intensive workload, SoftWrAP can have a service and response time faster than that of the Non-Atomic method that does not guarantee atomicity, and only a little less throughput due to the small logging overhead.

C.2 Alias Table Size

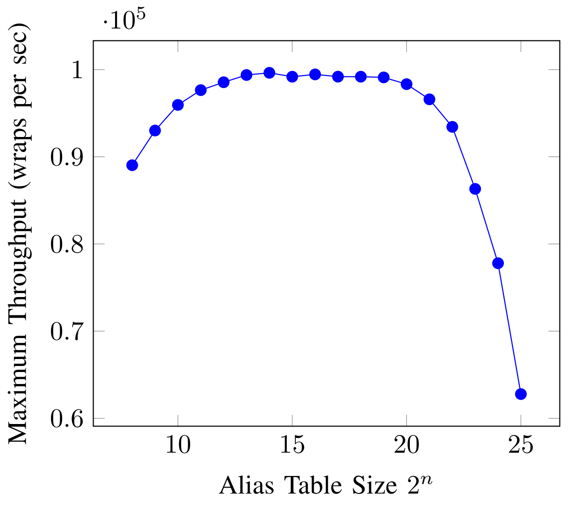


Fig. 7: Maximum throughput for various Alias Table sizes. Arrival rate = 50k wraps per sec, n=10, T_w=1μs.

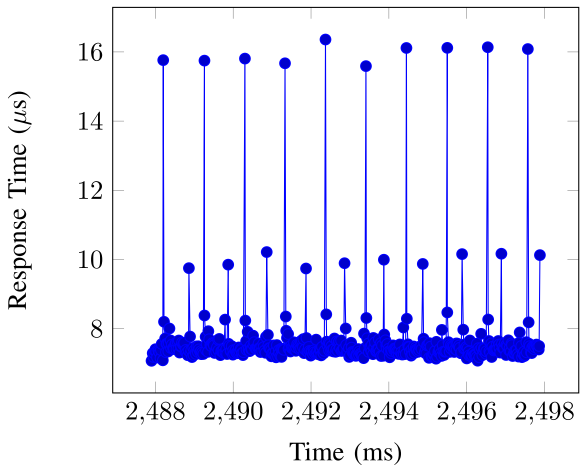


Fig. 8: Response time over time. Arrival rate = 50k wraps per sec, n=10, Alias Table: 8k entries, T_w=1μs.

In the following sets of experiments, the sensitivity of SoftWrAP to the size of the alias table is analyzed. Our implementation of the alias table uses a double buffering technique with two hash tables. Each hash table is array based and walks the hash index looking for a match or free entry. To avoid walking a dense table, we start retiring the table using a retirement thread after reaching a configurable size threshold, currently set at $1/8$ the capacity, to reduce the average probe time. On a wrapClose operation, if the table has reached the threshold size, the retirement thread is notified. The retirement thread might be busy processing the other table, therefore a new wrapOpen may have to wait until that table has been completely retired. Each element in the table requires a key, value, and size; so an entry requires 24 bytes. We test the size of the alias table for $n = 10$ elements per wrap and an incoming rate of $50,000$ wraps per second. As shown in Figure 7, when the table is relatively small (2^{12} entries or less) or very large (greater than 2^{20} entries) the throughput falls. The reason for the drop in throughput at large table sizes is that the alias table thrashes in cache and requires a long retirement time. At small sizes, the drop in throughput is caused by the bookkeeping operations in creating, freeing, and updating state of the hash tables as discussed below.

Figure 8 shows the response time and alias table sizes for each table in the double buffer implementation for a subset of time in the same experiment. Once one hash table fills up, the retirement thread must be signaled and the hash table switched. The spike in response time as measured in the experiment is caused by the filling of a hash table, since the incoming rate of $50,000$ wraps per second of 10 writes each, fills a table of threshold 500 every $1ms$. The extra time includes signaling the retirement thread, zeroing out the table, and performing a memory barrier. The retirement thread can also zero out the table, but first must copy entries to home locations and perform a barrier to prevent threads from reading stale values. Chaining additional hash tables could reduce spikes in response time, but increase overall latency due to reads potentially having to consult each table in the chain.

Currently our design has two options to handle high churn rates of the alias table: object granular updates in which SoftWrAP is used with simplified pointer flipping for coarse-grained objects or by using local alias tables which are recycled immediately after a wrap closes.

C.3 Multi-Threaded Operation

Next, we tested writing with multiple threads to show that the performance of SoftWrAP and the alias table is not adversely affected with scaling the number of writers. In this experiment, a thread is created that updates its own data object independently. Each thread continuously executes transactions to update 10 randomly positioned integer fields on the object. A varying number of threads were created, and the transaction throughput was recorded and plotted in Figure 9. To support data from multiple threads, the size of the alias table was increased to 2^{19}, as from Figure 7 before response time decreases. We compared SoftWrAP with both Undo Log and Non-Atomic on a 6 core CPU. When the number of threads exceed the number of cores, performance starts to decrease. SoftWrAP is much faster than Undo Log and close to the performance of a Non-Atomic method.

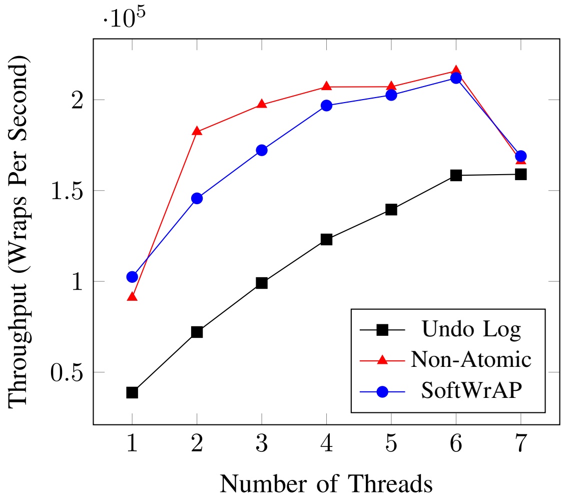


Fig. 9: Throughput of wraps with 10 elements each. SoftWrAP has an alias table size of max 2^{19} entries and SCM T_w=1μs.

C.4 Varying SCM and Transaction Size

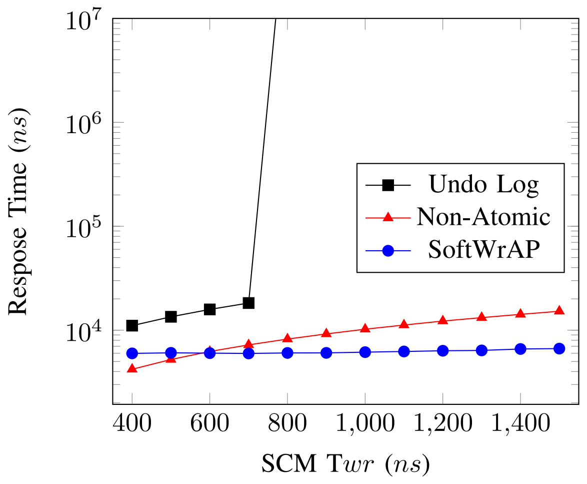


Fig. 10: Response time for various SCM T_w times with arrival rate of 50k txps of size n=10 and alias table size of max 8k entries.

Next, we vary the SCM write time T_w in an array update at 50,000 txps in Figure 10. On fast T_w times approaching the speed of DRAM, the SoftWrAP overhead is greater than the Non-Atomic time. As the time to write a cache-line to SCM is increased, the benefit of SoftWrAP increases as the aliasing time T_{alias} remains constant. At SCM T_w of $600ns$, both implementations have equal performance. This is as expected from the values in Table I, setting n=10, and $T_w = 600ns$, yields a Non-Atomic time of $6,220ns$, close to the same processing time for SoftWrAP. Additionally, for SoftWrAP, the time required for aliasing, T_{alias}, can overlap the time required to stream SCM writes to the log, aside from the final T_w log write. With SCM T_w of $1,500ns$, SoftWrAP has a response time measured at $6,640ns$. SoftWrAP, even with fast SCM, provides atomicity and consistency to a group of writes.

Figure 11 shows the relationship between the size of a

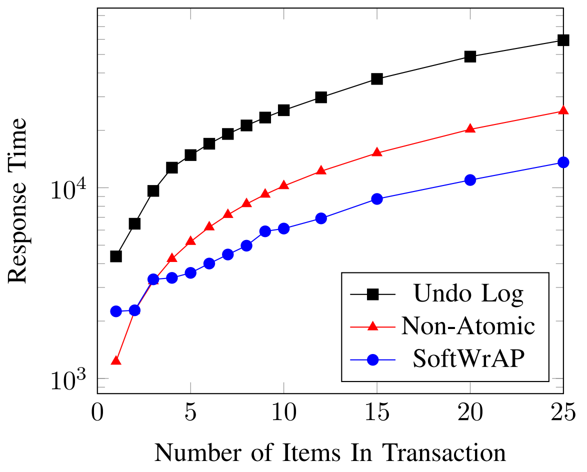


Fig. 11: Response time for various transaction sizes with arrival rate of 1,000 txps, alias table size of max 8k entries, and SCM T_w=1μs.

transaction and the effect on the overall performance. As the size of the transaction grows, SCM writes in SoftWrAP only grow at 12/64 the rate. When there is just one element in a transaction, obviously any overhead of logging by SoftWrAP, Undo Log, or any transactional method, will perform worse when compared to Non-Atomic. However, with just two elements in a transaction, SoftWrAP can perform as well as Non-Atomic. Certainly, anyone wishing to update just one element would not go to the trouble to create a transaction mechanism. However, even if one does, or the size of an update is not known prior to the start of a transaction, then SoftWrAP still performs exceptionally well.

C.5 Data Reuse

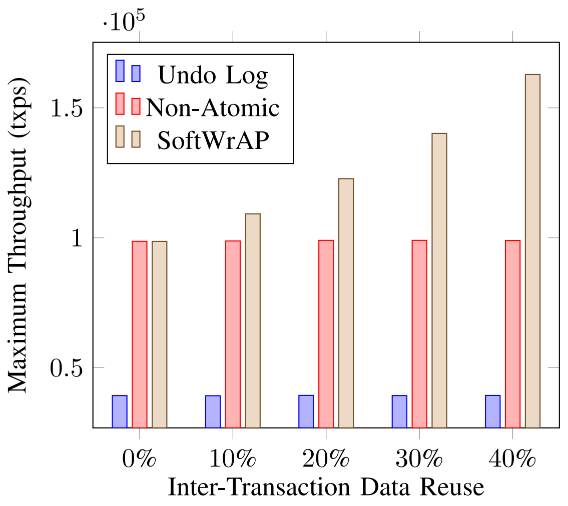


Fig. 12: Maximum throughput for various percentage of data reuse across transactions with size n=10, alias table size of max 8k entries, and SCM T_w=1μs.

The previous experiments only examine the cases where data was not reused across transactions. However, as in the case of many applications, data is reused both across and within transactions for both reads and writes. We show in Figure 12 that with data being reused between transactions, that the overall throughput is increased. With no reuse, SoftWrAP performs equally as well as Non-Atomic. When 10% of the data is reused across the transactions, then the number of writes to SCM that must be performed by the retirement thread is decreased by 10%. In the maximum throughput case, the bottleneck is the processing of a full hash table. Reducing this burden by a percentage allows the throughput to increase proportionately. Undo Log and Non-Atomic do not benefit from data reuse and remain unchanged.

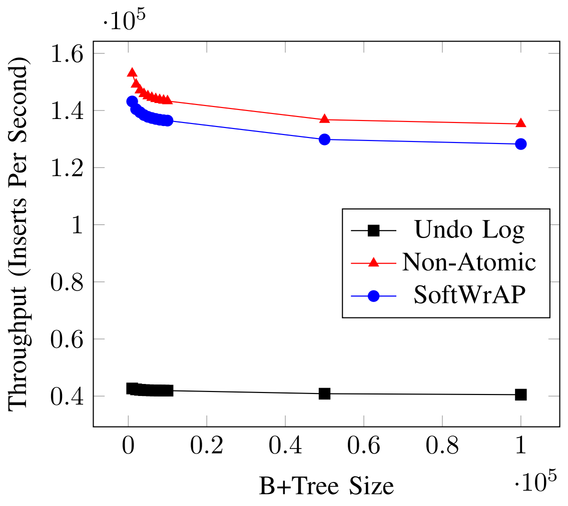


Fig. 13: Average insert throughput of random elements into a B+Tree with alias table size of max 4k entries and SCM T_w=1μs.

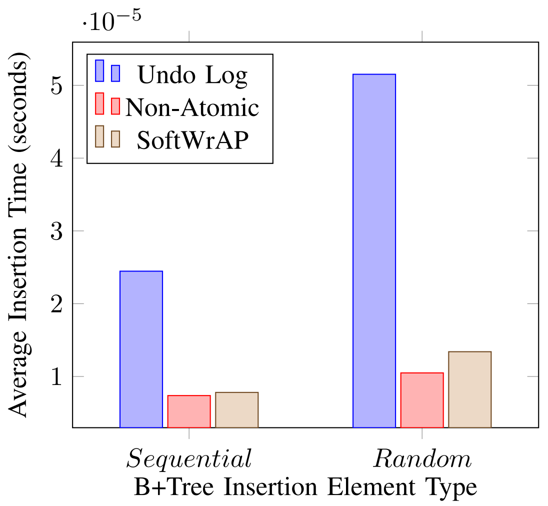


Fig. 14: Average insertion time of a single integer element (sequential or random) into a B+Tree with alias table size of max 4k entries and SCM T_w=1μs.

D. B+Tree Benchmark

The previous experiments consisted of all write workloads. In this experiment, we also perform a series of writes into a data structure. However, inserting just one element into a B+Tree requires a number of reads and writes that need to be wrapped with atomic semantics to preserve the integrity of the tree. We setup this experiment by slight modifications to the STX B+Tree [5] extension to the C++ STL, wrapping persistent data accesses in the B+Tree elements. We also created a simple STL based memory allocator that allocates memory segments for the B+Tree in persistent memory. Nested transactions get rolled up into the top level transaction. For Non-Atomic, just a single *pcommit* is executed once all nested writes are finished in the insertion of an element, but if a failure occurs, the data structure can become corrupted.

To test the performance of the B+Tree, we perform 1M transactions of one element inserts into the B+Tree, recording the number of transactional reads and writes to SCM needed by the internals of the B+Tree data structure along with the average time per transaction. The results for both $Sequential$ and $Random$ series of insertions for each method are shown in Figure 14. Throughput for $Random$ insertions is shown in Figure 13. The fastest way to build a B+Tree with a given set of elements is sequentially, as it requires less transversing and modification of internal nodes. Due to the contiguous nature of data elements allocated in the B+Tree, many of the writes, even in the Non-Atomic method, get write combined into a single cache-line write to SCM.

In our $Sequential$ experiment, 1M inserts requires 61.6M reads and 8.1M writes, an average over 61 reads and 8 writes per insert. Non-Atomic averages $7.4\mu s$ per insertion, which is less than the expected average insert time that would require 8 writes to SCM, $8{*}T_w$ or $8\mu s$. However, we also measure a write combining effect of the B+Tree of 56% which reduces the Non-Atomic SCM writes to 5, and the expected time from Table I is therefore $5,220ns$. The remaining execution time of $2.2\mu s$ is needed for reads and traversing the tree. Undo Log requires all 8.1M writes to be synchronous, and referring back to Table I, Undo Log for n=8 writes takes 19 T_w + (n+1)T_s or 21.3 $1\mu s$. Adding the same additional read and transversal time as Non-Atomic of $2.2\mu s$ yields $23.5\mu s$ for Undo Log which is close to the measured $24.4\mu s$. For a SoftWrAP of 8 writes and 62 reads, the time required from Table I is $4.8\mu s$ for the writes plus the cost of 62 reads into the alias table structure, for the measured $7.8\mu s$. The cost of the read is then calculated to be approximately $44ns$ which is higher than a direct read to memory might cost, but reasonable given the complexity and benefit of SoftWrAP. For the $Random$ insertion experiment of 1M elements, 74.8M reads and 20M writes are measured, which produces similar expected times that match the measured experiments for each method as well. SoftWrAP outperforms Undo Log and remains close to Non-Atomic even with heavy read-intensive workloads.

E. TPC-C Database Benchmark

Finally, we tested the performance of the SoftWrAP framework with the TPC-C Benchmark. The TPC-C benchmark is a complex, online transaction processing benchmark [28].

We chose to test the SoftWrAP Framework with SQLite due to its widespread use and ease of extendibility. SQLite is implemented as a compact library and embedded database engine with a pluggable interface for extended media [2]. SQLite performs writes to the main database file atomically by first creating a journal file. The journal file is like a re-do log in that if a crash happens before the main database update has been completed, the journal can be replayed to capture any outstanding changes.

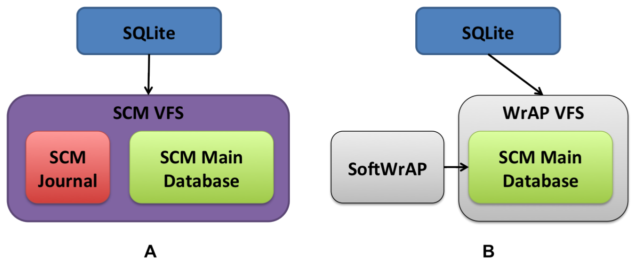


Fig. 15: SQLite Pluggable Virtual File System Implementations using SCM for block writes and using SoftWrAP.

Figure 15 shows how SQLite is extended to use Storage Class Memory using the Virtual File System Interface. This interface requires implementation of Open, Close, Read, Write, Sync, and other methods that closely align with the SoftWrAP API. In part A, a SQLite VFS is created that performs all writes to SCM. When the VFS requests a Sync or Close operation, a persistent memory $pcommit$ is performed to ensure that all writes to SCM have been committed. The journal is also updated in SCM just like the main database, and the journal can be discarded once the main database has been updated.

In B, the SQLite VFS utilizes the SoftWrAP framework. In the SoftWrAP version, the journal need not be created directly as direct writes to the database are contained within the SoftWrAP based logs. Only the main database needs to have updates when using SoftWrAP. Therefore, the updates to the main database can be streamed to the SoftWrAP log location and aliased in DRAM. Database reads need not query the SCM based journal and SCM database, but rather just utilize the SoftWrAP API, which can direct a read to either the DRAM alias table or home SCM location. This reduces the overall number of SCM reads, writes and persistent memory syncs. In such an application, where an application already delineates its updates clearly, the effort to introduce wrapping of those updates is low and straight-forward.

To generate the SQL statements that represent a portion of the TPC-C Benchmark, we utilize the PY-TPCC engine, a Python based implementation of the TPC-C Benchmark [23]. We intercepted the SQL statements from the benchmark to a supported database and directed them to our SQLite database. The test with SQLite is then performed with the SCM VFS from part A, and the SoftWrAP based implementation in part B. Undo Log and Non-Atomic are also tested with the SoftWrAP Framework.

Figure 16 shows the throughput of SQLite for the TPC-C benchmark operations for various SCM write times. Note that SoftWrAP performs almost as fast as a Non-Atomic implementation that doesn't observe consistency. Execution on

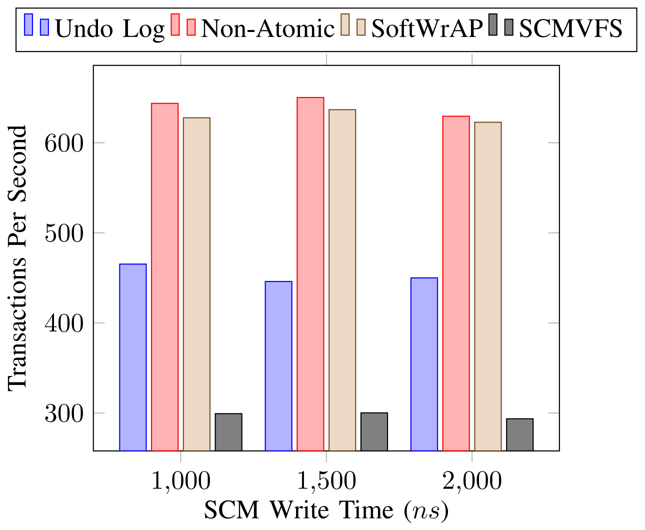


Fig. 16: Throughput in Transactions Per Second for the TPC-C Benchmark with SQLite.

disk based media only resulted in less than 20 transactions per second, and SoftWrAP is over 30 times that performance at 1 and 2 μs SCM write times.

VI. Related Work

Analysis of consistency models for persistent memory was considered in [24]. To enforce atomicity in NVM, software approaches in the literature rely on simple hardware support: Atomic 8-byte writes, memory fences, and instructions with PSYNC semantics. We similarly rely on these hardware capabilities. Memory controller designs for persistent memory haven been proposed in [25], [34], [14], [33]. Adding a small DRAM buffer in front of SCM to improve latency and coalesce writes was proposed in [25].

Hardware approaches that specialize processor and cache behaviors to achieve atomicity have been proposed [9], [8], [19], [29], [33], [20]. These approaches change the front-end architecture with additional cache hardware and policies. Copy-On-Write mechanisms (BPFS [9]) and timestamp-based multiversioning [29], [19] to maintain before and after versions of updated persistent memory have been proposed for ordering cache evictions. When combined with hardware-supported atomic writes to 8-bye words, this provides an effective mechanism for atomic updates of an important albeit restricted class of block-based tree-structured data structures, by using pointer flipping. Research into new data structures such as in NV-heaps [8], which uses logging and copying, provide support for ACID components in software applications using SCM. CDDS [29] provides a versioning method that copies data and uses sequences of fences and flushes to provide transaction support. BPFS [9] and NV-heaps [8] require changes to the system architecture to support the atomicity and consistency of data. A non-volatile victim cache to provide transactional buffering was proposed in [33], with the added property of not requiring logging at all; by comparison, our approach achieves efficiency through software based non-temporal write-combining streaming of log records. Unlike [33] which tracks pre- and post- transactional states for cache lines in both volatile and persistent caches and atomically moves them to durable state on transaction commits, our approach does not change the behavior of hardware and instructions and does not require synchronous cache-line write-backs out of processor caches on completions. Whole-system persistence [20] snapshots the entire micro architectural state at the point of a failure to allow for in memory databases, but relies on batteries to power non-persistent memories on system failure. The hardware based WrAP architecture allows an application to handle its own concurrency control such as in RVM [26], but requires hardware changes to the cache backend [14].

Some solutions [30], [6] combine concurrency control with persistence in an integrated framework. Mnemosyne [30] uses software transactional memory (STM) based interception of all writes and reads within a transaction, and uses internal copying and logging to achieve both concurrency control and atomicity. The approach has the advantage of using a single framework for meeting ACID requirements, and can potentially leverage advances in STM technology to improve performance. On the other hand it forces the application developer to a single concurrency control model (TM), and is difficult to fit to legacy software applications, which support different isolation models or employ lock-based concurrency control. ATLAS [6] uses a compiler pass to automatically generate transactional regions for atomic writes utilizing a synchronous undo log. Coupling the very distinct concerns of providing atomicity to a group of operations (like mutually dependent pointer switches) and controlling concurrent accesses to the data structure is not desirable, as applications demand more and more greater autonomy in their implementation (for instance the noSQL approaches to big data and databases like voltDB [4] that employ statically serialized transaction sequences to reduce concurrency control overheads). Our approach explicitly decouples concurrency control from durable atomicity requirements.

Research in persistent file systems built on SCM is also a promising area that might quickly enable software applications to take advantage of SCM. This will be advantageous for legacy file based programs to easily take advantage of SCM. However, new classes of applications can benefit from optimized byte-addressability, such as provided by SoftWrAP. PMFS [11] is a complete file-system implementation built for SCM. SCMFS uses sequences of **mfence** and **clflush** operations to perform ordering and flushing of load and store instructions and requires garbage collection [31].

A technique for ensuring atomicity of **sync** was discussed in [22]. REWIND [7] is a library that also supports transactional writes to main memory similar to early an early presentation of Software-based Write-Aside Persistence [15]. However, REWIND uses an in-persistent-memory version of a log structure. With SoftWrAP, reads proceed through DRAM, log processing can be performed from the DRAM based alias table, consecutive log entries can be streamed into SCM, and multiple writes to the same variable become one final merged update into its SCM home location at a later time.

VII. Summary

In this paper we presented SoftWrAP, a software framework for persistent memory providing lightweight atomicity and durability. The SoftWrAP framework is currently available for download from http://nvmwrap.com. SoftWrAP utilizes an alias table approach to benefit from processor caching while avoiding the problems caused by uncontrolled cache evictions. This is coupled with micro-logging of updates that allow for write-combining of streaming store operations (not possible with synchronous Undo Log methods) and asynchronous background retirement of the updates to SCM (not possible in the Non Atomic approach). SoftWrAP allows for a fast access path to data through the cache while making sure that persistent memory layers are not slowed down.

We compared the SoftWrAP to implementations that write directly to SCM and do not guarantee atomicity (Non Atomic approach) and that use well-known Undo Log approaches to guarantee atomicity. Our experiments show significant speedups over Undo Log based atomicity and comparable performance to Non Atomic implementations that provide no safety guarantees. Just as transactional memory liberates software developers from the mechanics of achieving execution atomicity, Software-based Write-Aside persistence in the SoftWrAP framework liberates software developers from the minutiae of delivering atomicity in storage operations while not affecting performance.

References

[1] Neo4J:. In *http://neo4J.com* (May 2015).

[2] Sqlite. In *http://www.sqlite.org* (May 2015).

[3] Unicom solidDB. In *http://unicomsi.com/products/soliddb/* (May 2015).

[4] VoltDB:. In *http://voltdb.com* (May 2015).

[5] Bingmann, T. Stx b+ tree c++. In *http://panthema.net/2007/stx-btree/*.

[6] Chakrabarti, D. R., Boehm, H.-J., and Bhandari, K. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications* (New York, NY, USA, 2014), OOPSLA '14, ACM, pp. 433–452.

[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).

[8] Coburn, J., Caulfield, A., Akel, A., Frupp, L., Gupta, R., Jhala, R., and Swanson, S. Nv-heaps: Making persistent objects fast and safe with next generation, non-volatile memories. In *Proceedings of 16th ASPLOS* (2011), ACM Press, pp. 105–118.

[9] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., and Coetzee, D. Better I/O through byte-addressable, persistent memory. In *Proceedings of 22nd ACM SOSP* (2009), ACM Press.

[10] Doshi, K., and Varman, P. WrAP: Managing byte-addressable persistent memory. In *Memory Archiecture and Organization Workshop.(MeAOW)* (2012).

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

[12] Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S., and Lehner, W. SAP HANA database: data management for modern business applications. *SIGMOD Rec. 40*, 4 (Jan. 2012), 45–51.

[13] Freitas, R., and Wilcke, W. Storage class memory, the next storage system technology. *IBM Journal of Research and Development 52*, 4/5 (2008).

[14] Giles, E., Doshi, K., and Varman, P. Bridging the programming gap between persistent and volatile memory using WrAP. In *ACM Computing Frontiers* (2013).

[15] Giles, E., Doshi, K., and Varman, P. Software support for atomicity and persistence in non-volatile memory. In *Memory Architecture and Organization Workshop (MeAOW'13)* (October 2013).

[16] Intel. Intel 64 and IA-32 architectures software developer manual, January 2015. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[17] Kim, H., Seshadri, S., Dickey, C. L., and Chiu, L. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. *ACM Transactions on Storage (TOS) 10*, 4 (2014), 15.

[18] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. 17*, 1 (Mar. 1992), 94–162.

[19] Moraru, J., Andersen, D., Kmainsky, M., Binkert, N., Tolia, N., Munz, R., and Ranganathan, P. Persistent, protected and cached: Building blocks for main memory data stores. In *CMU Parallel Data Lab Trechnical Report, CMU-PDL-11-114* (Dec. 2011).

[20] Narayanan, D., and Hodson, O. Whole-system persistence. In *Proceedings of 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ACM Press, pp. 401–410.

[21] Ousterhout, J. The case for RAMCloud. *Commun. ACM 54*, 7 (July 2011), 121–130.

[22] Park, S., Kelly, T., and Shen, K. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th Eurosys* (2013), pp. 225–238.

[23] Pavlo, A. Py-tpcc. In *https://github.com/apavlo/py-tpcc*.

[24] Pelley, S., Chen, P. M., and Wenisch, T. F. Memory persistency. In *ISCA'14* (2014), pp. 265–276.

[25] Qureshi, M. K., Srinivasa, V., and Rivers, J. A. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of 36th International Symposium on Computer Architecture* (2009), ACM Press, pp. 24–33.

[26] Satyanarayanan, M., Mashburn, H. H., Kumar, P., Steere, D. C., and Kistler, J. J. Lightweight recoverable virtual memory. *ACM Trans. Comput. Syst. 12*, 1 (Feb. 1994), 33–57.

[27] Shalev, O., and Shavit, N. Split-ordered lists: Lock-free extensible hash tables. *J. ACM 53*, 3 (2006), 379–405.

[28] Transaction Processing Performance Council. *TPC-C Benchmark Version 5.11*. http://www.tpc.org/tpcc/.

[29] Venkatraman, S., Tolia, N., Ranganathan, P., and Campbell, R. H. Consistent and durable data structures for non-volatile byte addressable memory. In *Proceedings of 9th Usenix Conference on File and Storage Technologies* (2011), ACM Press, pp. 61–76.

[30] Volos, H., Tack, A. J., and Swift, M. Mnemosyne: Lightweight persistent memory. In *Proceedings of 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ACM Press, pp. 91–104.

[31] Wu, X., and Reddy, A. L. N. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 39:1–39:11.

[32] 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* (New York, NY, USA, 2013), MICRO-46, ACM, pp. 421–432.

[33] 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* (New York, NY, USA, 2013), MICRO-46, ACM, pp. 421–432.

[34] Zhao, P., Zhao, B., Yang, J., and Zhang, Y. A durable and energy efficient main memory using phase change memory technology. In *SIGARCH Comput. Archit. News* (2009), ACM Press, pp. 14–23.