Persisting In-Memory Databases Using SCM*

Ellis Giles
Rice University
Houston, TX 77005
erg@rice.edu

Kshitij Doshi
Intel Corporation
Portland, OR
kshitij.a.doshi@intel.com

Peter Varman
Rice University
Houston, TX 77005
pjv@rice.edu

***Abstract*—Big Data applications need to be able to access large amounts of variable data as fast as possible. Emerging Storage Class Memory (SCM) fit this need by making memory available in large capacity while making changes endure as a seamless continuation of load-store accesses through processor caches. However, when writing values into a persistent memory tier, programmers are faced with the dual problems of controlling untimely cache evictions that might commit changes prematurely, and of grouping changes and making them durable as a unit so that consistency can be guaranteed in the event of sudden failure.**

In this paper, we present various methods to achieve high-performance byte-addressable persistence for an in-memory data store. We chose Redis, a popular high-performance memory oriented key value database. We modified its source code to use SCM such that updates to data and structures are performed in a failure resilient manner. We evaluated the changes using both internal benchmarks and the Yahoo! Cloud Servicing Benchmark (YCSB). We found that even though Redis uses many SCM read operations, it can benefit from highly optimized persistent SCM write based approaches, especially when SCM write times are longer than DRAM write times. The paper presents an innovative Local Alias Table Batched (LATB) method, and shows that it outperforms the alternatives.

I. Introduction

Big Data has recently become a popular term that is used to describe the size and variety of the growing data sets used by organizations worldwide. Initially characterized by three components: Volume, Velocity, and Variety [1] [2], Big Data now often encompasses seven V's like Value, Veracity, Variability, and Visualization [3] as well.

A new type of emerging computer memory technology, Storage Class Memory (SCM), dovetails perfectly with the requirements of managing big data. SCM is both byte-addressable (like DRAM main memory) and persistent (like traditional storage devices) and operates on the main memory bus; SCM has the potential of supplanting the traditional block I/O interface to hard drives and SSDs with direct processor access using normal memory operations. This combination of direct and cacheable access from CPUs, persistence, and high capacity means that SCM can bridge a long-standing gap between slow block-based persistent storage and fast, byte-addressable volatile memory (DRAM) as shown in Figure 1. Several technologies with different performance, durability, and capacity characteristics are under research and development or on the verge of broad commercial deployment.

*Supported by NSF Grant CCF 1439075 and Intel SSG.

The emergence of new large capacity and durable memory technologies is thus timely for Big Data. SCM (see Figure 1), helps solve some of the Big Data challenges:

- **Large Capacity** - Peta-byte sized SCM volumes using Phase Change Memory or PCM, allow for more data to be processed per node and closer to the processor.
- **Persistence** - Non-volatility of SCM permits interrupt-free and streamlined computation as threads do not need to schedule or serialize memory objects for IO.
- **Byte Addressability** - Data structures and algorithms can be optimized for cache-friendly, fine-grained memory access to both structured and unstructured data.
- **Speed** - SCM will be offered in DIMMs that fit alongside DRAM on the high-speed main-memory bus.

Most recently, In Memory Databases (IMDBs) such as Redis [4], have become a popular solution to meet the throughput and response time requirements of applications dealing with large amounts of unstructured data or NoSQL access. Redis is a high-performance in-memory data store that offers both in-memory and persistent data operations. Redis can be accessed over traditional networking channels or can be linked into applications as an embedded high-performance API.

The many potential benefits of SCM are tempered with the new challenges that arise in using the technology for big data applications, specially with regard to reliability and consistency. When writing values into a persistent memory tier using direct loads and stores, application programmers are faced with the responsibility of ordering their writes correctly and ensuring that the SCM remains consistent in the presence

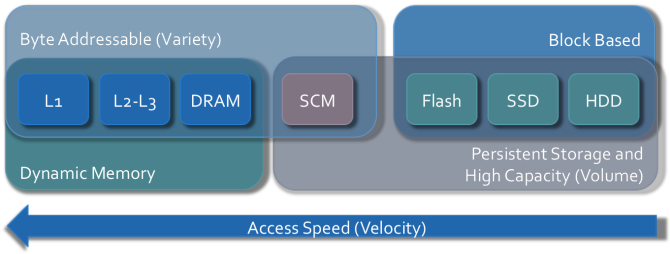

Fig. 1: Storage Class Memory provides byte addressability for Variety, operates on the main-memory bus for Velocity, and some types, like PCM, have a large persistence data Volume.

978-1-4673-9005-7/16/$31.00 ©2016 IEEE

of arbitrary failures. Besides having to create metadata to facilitate recovery in case of failure, SCM-based applications must further ensure that their efforts are not undermined by the actions of an independent cache hierarchy that can make autonomous and unpredictable cache evictions to persistent memory in the midst of atomic store sequences.

In this paper, we present a light-weight approach for writing ordered and atomic store sequences to SCM that is both fast and read-friendly. Other approaches such as a synchronous Copy-on-Write approach and a Redo Log based Local Alias Table approach [5] provide either fast reads or fast writes, as seen with writes to an in-memory B-Tree in figure 4, but not both. Our approach benefits from write-combining log data on the foreground path and minimizes read set reconstruction by limiting pending data changes. We wanted to analyze the performance of how our approach performed in a real Big Data application to see if the persistence mechanism has an overall impact. We modified the Redis source code to achieve in-memory persistence for many of the Redis data structure operations from Keys and Values to Lists, Sets, Hash Maps, and Zip Lists. This paper evaluates the different persistence options for Redis with built in benchmarks and the Yahoo! Cloud Servicing Benchmark [6].

II. Overview

Traditional applications requiring durability typically serialize their data into consecutive segments before persisting it to disks or SSDs. Reliable saving to non-volatile media typically requires creating a log to guard against failure and, after the log is committed, writing the updates into the the persistent block store before finally removing the log.

A fast data structure store such as Redis can persist data to disk or SSD while keeping as much data as possible in memory for speed as allowed by its configuration. Frequent storing of data to a persistent tier greatly affects performance, while infrequent saving can result in losing a considerable amount of updated data. This is particularly a concern for high-speed data stores, as the less frequently they save, the more data that can be lost; however, saving frequently or appending every update to a continuous log can negate the desired performance goals.

Storage Class Memory will allow an in-memory data structure store such as Redis to operate directly in persistent memory, without having to check-point operations and serialize the disk data. However, applications are now faced with managing the dual problems of spurious cache evictions and atomic grouping stores to guarantee consistency in case of failure.

Consider a system with a non-volatile SCM DIMM attached to a memory bus alongside a volatile DRAM DIMM. A write to a variable x might be held in a number of places ranging from the front end cache buffer, the cache hierarchy, write buffers, or memory. Upon completion of an atomic or locked section, the new value of x should be visible and accessible to other threads (or later transactions of the same thread), but the actual location of the variable x is not guaranteed to be in SCM and will likely reside in the volatile cache hierarchy.

Power failure when updating DRAM variables is not a problem since the variable value is cleared on restart. However, an update to a persistent variable that has not become durable when power fails is problematic. Compounding the problem, a later write may have bypassed it in the cache hierarchy and actually updated SCM. Using cache-line flush, *clflush*, to force the value out of the cache into SCM is insufficient, as the value may still be pending in a write buffer. Clflush also has the unfortunate side effect of invalidating the entire cache line.

Fortunately, the new Intel architecture specification [7] specifies a few new instructions. A new instruction *clwb*, or cache-line write-back, writes a cache line out to the write buffers but doesn't invalidate the cache line. A store-fence should be used before issuing a *clwb* or *clflush* as it might otherwise flush the value of the cache line before the update has been applied to it in the cache. However, following a *clwb*, the updated value might still be in the write buffer and not persisted on SCM. To solve this problem, Intel is adding capabilities for fencing stores to SCM locations, with hardware guaranteed safety against loss of data in the event of power failure. Such synchronous operations are effectively global as they are preceded by the flush of the affected cachelines.

Some of the recently established approaches for achieving atomic persistence in SCM are discussed below.

- **Copy-on-Write or Undo Log:** A Copy-on-Write approach is a synchronous operation which first makes a persistent copy of a variable's value before updating it. On block storage this is done through a disk flush. In SCM this can be accomplished by writing the address of x and value of x to a log, followed by a persistence commit of the log entry, and then updating x. The writes to the variables are written to cache, fenced and a cache write back to home locations initiated. The writes must be persistence-commited before the log can be removed.
- **Redo Logging:** A Redo Log writes data to shadow locations before writing to home locations. This may be faster than an Undo Log in write heavy workloads, but for reads, the log must be searched. Expensive persistence commit operations don't need to be performed until the end of a transaction as it is safe to lose the log if a failure occurs. After the updated values have been persisted in the log, they can be lazily written back to their home locations.
- **Checkpoints:** Checkpoint based mechanisms may use a variety of logging or shadow techniques that might not guarantee that every completed transaction is durable to a storage medium, but still guarantees the ordering of the transactions. In this method, the most recent completed transactions may be lost, but consistency is guaranteed up until some prior completed transaction.

A. Aliasing

In this section we describe the differences between shared aliasing with SoftWrAP [8] and a Local Alias Table [5]. The basic idea in SoftWrAP [8] is to simultaneously propagate updates made within an atomic code section along two paths:

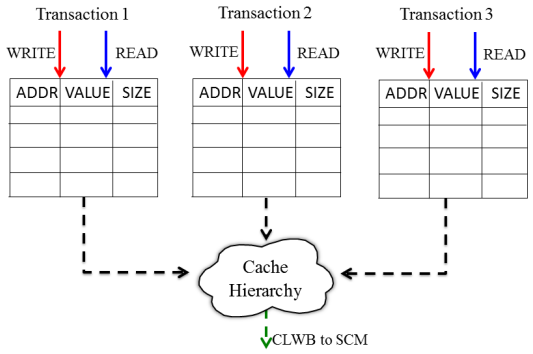

Fig. 2: Local Aliasing in SoftWrAP

a foreground path through the cache hierarchy that is used for value communication, and an asynchronous background path to SCM for recovery logging. In [8], a shared, volatile alias table containing the most recent values is used for value communication between writers and readers. New values are entered into the alias table while being streamed to a SCM based log. Post-transactional values are transferred to SCM from the alias table and space in the alias table and log are released. By creating these two paths, SoftWrAP decouples transaction value communication from recovery logging.

The decoupling of concurrency control (or transaction isolation) from failure atomicity allows persistence to be added flexibly to code that is already multi-threading safe. For the common case of multi-threading safe or strictly isolated transactions, this paper further decouples the reclamation of the aliasing structure from the retirement of transaction logs, and makes post-transactional values available immediately to subsequent transactions without aliasing.

We first describe how [8] achieves failure safe updates in SCM, and then how a Local Alias Table or LAT [5] simplifies the transactional persistence operation. Our approach, described in Section III, is a variant on LAT [5] that makes it both read friendly and write fast.

An atomic persistence region starts with atomic_begin which translates into a call to the *OpenWrap* library function. OpenWrap starts a log record for the transaction in SCM. Each store, such as x = 1, translates into a library call to *wrapStore*, which updates (or allocates) an entry for x, designated by x_a in a shared alias table in DRAM with the SCM address of x, its size and new value. Similarly, it appends an entry to the log with x's SCM address, its new value, and size. A load of a wrapped variable, such as $z = x$ translates into a library call to *wrapLoad*, which uses the alias of x, x_a, to return the value of x from the shared alias table for use in the assignment. When the transaction closes with atomic end, the CloseWrap library function performs a streaming flush of the log record with an end-of-record marker, and persistence-commits the log record safely in SCM. Subsequent transactions that need to read values of variables x, y, z modified in this atomic persistent region use wrapLoads to get them from the shared alias table. In the background, post transactional values of persistent variables are copied into SCM home locations and alias table entries are removed in lock step with trimming of the log records. A double buffered, multi-state shared alias table provides for lock-free reclamation of space [8].

For persistence transactions which are strictly isolated from one another, as with serialization schemes such as locks, a shared alias table is not necessary. Instead, a local aliasing scope suffices to forward values from writers to readers. In the LAT scheme its difference from a shared aliasing scheme of [8], is discussed. As shown in Figure 2, each persistence transaction maintains a private (local) alias table in DRAM. The library calls *wrapStore* and *wrapLoad* operate on aliased entries x_a, y_a, and z_a as in [8], except that the aliasing is performed through $Transaction\ Local\ Aliasing$, since strict isolation removes any overlap among variables written in one transaction and concurrently read in another. On CloseWrap, each persistence transaction streams the updated values from its local alias table to the cache hierarchy. The updated values may asynchronously move to SCM and are available in place (at their SCM addresses) without aliasing, for subsequent transactions. To expedite trimming of log records, updates are flushed (using clwb) and durably fenced for persistence commit; however, it is possible to batch the fencing for data value updates for multiple transactions as an optimization.

The LAT approach [5] can be also used when applications can tolerate unprotected reads, i.e. eager reads of variables (not yet durably committed) that produce stale values. Also, LAT can make streaming of log and data values to SCM efficient with AVX VSCATTER operations. Further, a compiler can be used to maximize static aliasing at compile time since LAT aliased values do not exist beyond each wrap.

However, when reconstructing disjoint structures for read access, the table must be consulted for primitive values that may be contained at various previously updated positions within a data structure. This makes for a higher read cost. Several performance enhancements have been evaluated including Bloom Filters and faster hash tables, but with overlapping blocks in an Alias Table, many reads have to be reconstructed from several table entries.

III. Approach

Seemingly the local aliasing based Softwrap-LAT approach of Figure 2 is well matched with a single threaded application like Redis in which the aliased values, as they are only accessed from one thread, can be retired after the transactional log has been committed into persistence. However, many operations in Redis are characterized by a high read fraction. As an overview of Redis in this section explains, at times, Redis may generate stores that happen to overlap at a sub-word granularity from different scopes; which complicates the aliasing of the affected locations and makes it challenging to optimize. We develop a variant of LAT for better handling of read-heavy SET operations while minimizing the number of synchronous commits under update operations, as described in subsection B.

A. Redis

Redis has several persistence options. It can be executed with no persistence, with data maintained and updated in memory, which may suffice for some usages. Common durability options consist of two: a Redis database ('.rdb') file and an Append Only File (AOF). An rdb file can be configured to be saved either periodically or whenever the number of updates crosses a threshold; but since even one second of data loss can result in thousands of updates being lost, the AOF is an attractive option: as it can be configured to append updates to a file and flush that file frequently (on each update or after each batch of updates).

A naïve approach for persisting changes to Redis data when hosted in SCM is to synchronously flush every store from the cache hierarchy, while simultaneously logging those updates so that stores associated with incomplete operations can be discovered and discarded in the event of a crash. However, Redis and other key value databases only need to be kept consistent at a coarser grain: that is, instead of logging and committing every store, the aggregate result of creating a key-value association, updating the value for a given key, and deleting a key need to be atomic in SCM. This is the approach taken in this paper, and compared with alternatives. The options are:

1) Minimize alterations to Redis, but write the checkpoints (RDB files) to SCM – allowing frequent checkpointing and thus reduced data loss.
2) As above, but write the Redis AOF file to SCM on every change, such as an insert or update of an item.
3) Keep Redis data in SCM along with an Undo Log also in SCM that is used during a system restart to reverse incomplete updates.
4) As above, keep Redis data and log in SCM, but alias data and precede SCM writes with write-aside log commits [8].
5) Only use SCM as volatile memory (i.e., benefit only from the large capacity).

Options 1, 2, and 5 require little to no effort but also do not derive the full benefit of SCM. We fully evaluate options 3 and 4 and describe in more detail along with optimizations below. We also explore a light-weight approach for writing SCM logs without holding up Redis transactions by employing a group-commit like approach that is commonly used with disk-based databases. Implementing options 3, 4 needed changes to less than 5% of Redis core source. We removed the block based persistence methods and modified Redis to run safely in emulated SCM.

Redis contains a memory allocator zmalloc with familiar operations to allocate, re-allocate, clear-allocate, and free memory chunks. In addition to using dynamically allocated memory for the core Redis data and structures - such as keys, values, strings, sets, lists, hashes, etc., it uses the same allocator for obtaining transient memory used in its client-server networking.

Listing 1: Before In-Memory Persistence

```
static void _dictReset(dictht *ht)
{
    ht->table = NULL;
    ht->size = 0;
    ht->sizemask = 0;
    ht->used = 0;
}
...
int dictRehash(dict *d, int n) {
...
while(d->ht[0].table[d->rehashidx]==NULL)
       {
            d->rehashidx++;
            if (--empty_visits == 0)
                return 1;
        }
...
}
```

Among its core structures are: 1) sds, or simple dynamic strings (comprising length, free byte count, and a buffer for the string itself) 2) robj, or Redis Object, a compound structure for storing and linking values in memory while encoding type and linkage information into a header, and 3) collections such as sets, hashes, lists, etc., in compressed or non-compressed variants.

To ready Redis for SCM, first we created temporary versions of zmalloc functions for memory allocation and release, e.g. tzmalloc, tzfree, tzrealloc, tzcalloc that used the configurable zmalloc memory allocation. We then changed zmalloc functions to use the SCM allocator described in [8]. As zmalloc is used in Redis for both its core data and for transient purposes like networking, this had the side effect of placing such transient elements in SCM as well. Since sds is used for keys, we created a new tsds type, similar to sds, that consists of the transient keys associated with such elements. To make these temporary sds objects become non-persistent, we created a temporary malloc arena (handled by tzmalloc, a wrapper around zmalloc) and used that to cleave the non-persistent elements aside from SCM range used for holding durable objects.

This approach to migrating Redis to an in-persistent-memory database is conservative. In effect, we treat an object as durable unless we find it to be a temporary object used for communication (by inspecting, or by instrumenting code). If a transient object is left in SCM, the approach would still work (as the object will get freed at some point) but incur needless overhead in logging and synchronous flushing to SCM for transient elements.

After marking the persistent/temporary data used by Redis, we ran a Pin [9] tool to identify source locations in the Redis codebase from where durable updates are made to data in SCM. We then wrapped [8] those lines of source code: using wrap-load/wrap-store calls for primitive data types and wrap-read and wrap-write calls for multi-location accesses such as those produced by memcpy calls. An example source code change for resetting a dictionary entry is shown below in Listings 1 and 2. In resetting a dictionary entry, the update

Listing 2: After Safely Wrapping Operations for SCM

```
static void _dictReset(dictht *ht)
{
    WrapOpen();
    WrapStore64(ht->table, NULL);
    WrapStore64(ht->size, 0);
    WrapStore64(ht->sizemask, 0);
    WrapStore64(ht->used, 0);
    WrapClose();
}
...
#define GetDictEntryTbl(table, offset)\
        WrapLoad64(((dictEntry**)\
              WrapLoad64(table))[offset])
...
int dictRehash(dict *d, int n) {
...
  while (GetDictEntryTbl(d->ht[0].table,
            WrapLoad64(d->rehashidx))
                == NULL) {
       WrapStore64PP(d->rehashidx);
            if (--empty_visits == 0)
                return 1;
        }
...
}
```

Listing 3: After Safely Wrapping Operations for SCM

```
int processCommand(redisClient *c) {
...
        WrapOpen();
        call(c,REDIS_CALL_FULL);
        WrapClose();
...
    return REDIS_OK;
}
```

must be performed atomically so as to not corrupt the in-memory data structure in case of failure. In a function, a wrap is opened before a group of transactional writes is performed and closed after the writes have completed; allowing for the function to be used by background transaction operations.

Some Redis operations, such as removing an item from a list, are performed fast on the foreground path by removing the object from the list and placing it on a cleanup list to be freed in the background. Several functions run at periodic intervals on an event loop, so these functions must also be transactional.

We found that Redis transactions could be wrapped at a high level around the main Redis call operations to support multi-set and pipelined operations. When a request comes in to update a Redis in-memory data operation, the object can be in a temporary data structure from the original network request. Redis then decodes the request and calls the corresponding operation through a function lookup table. Requests from clients to Redis all go through the processCommand and call functions such as that shown in Listing 3.

B. Local Alias Table Batched

Redis manages various compressed data types (compressed lists, sets, etc) with a data structure comprising an encoded dually linked list, called a zip list. This data type complicates

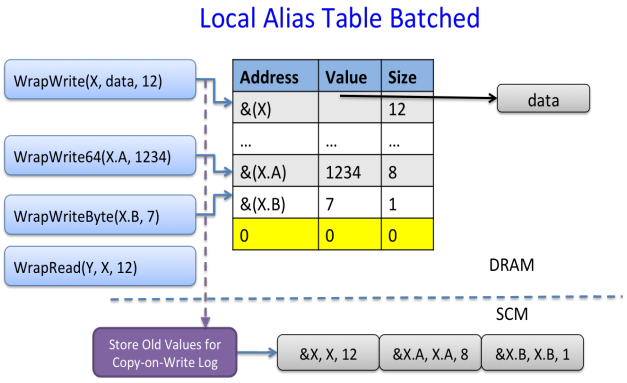

Fig. 3: Example with Local Alias Table Batched

wrapping memory operations for SCM atomicity. For instance, the zip list might have the first byte contain a few bits that specify a size and type. When traversed, these structures give rise to overlapped accesses to wrap-aliased locations spanning the different base and bounds in the encoded data structure, due to loads or stores in different function scopes within an overall Redis transaction such as a GET or SET. As a result, Alias Table design an important area for optimization. A common step in Redis is to initialize a structure, such as a dictionary hash table, and then to write the entire structure; this creates a block of entries in an alias table. Now if fields of the structure are touched from elsewhere in the transaction (which is common), care is needed as explained next. A given variable may or may not be aliased at a point in time; thus for reads, the wrap implementation has to check whether it is in the alias table (so that if it is not, then an access can be satisfied by reading from its unaliased home location in SCM); and for writes if the destination alias partially overlaps the source alias (due to changes arising from compression or decompression) then a new entry has to be added to the alias table. In general, creating, maintaining and accessing alias table entries for these mutable structures requires checking boundary conditions to distinguish between when such an overlap exists or does not, and whether when it exists, it is partial or complete; and, which fields in the overlapped area actually change and which fields do not. This generates a high price for mixing a chain of read/write operations as a block, and primitive reads and writes.

When reads occur more numerously in comparison with writes, it is advantageous to shift to a 'copy-on-write (C-o-W)' approach, since writes are made to a parallel structure, and reads do not therefore need to be aliased at all. We developed a hybrid scheme that has the write benefits of LAT [5], as shown in Figure 2 alongside the read benefits of Copy-on-Write. The scheme we use, LATB (for Local Alias Table, Batched) works as follows. We create an Undo Log in SCM but do not guarantee persistence on a store-by-store basis. We allow the LAT to grow to some threshold size before committing it to SCM. We switch from a REDO based consistency to a CoW based consistency when we detect block (i.e., multivalue) writes.

Figure 3 shows an example of the LATB and several writes to a data structure X. The first write is to the entire structure X of size 12. Subsequent writes to fields in the structure X, X.A and X.B, grow the alias table in memory. With the regular LAT, when a read of the structure X is executed, the data structure X must be rebuilt from scattered entries in the alias table. However, with LATB, the entries in the log are copies of the original values, so the table may be flushed at any time. With LATB, we flush the table when it reaches a configurable size or when a block wrapped read is executed that contains values inside the table. Therefore, with LATB, we don't need to scan the entire table to create X, but rather can read the value directly from main SCM memory.

IV. Evaluation

For evaluation we used an Intel Xeon(R) CPU E5-2697 v2 12 core processor at 2.70 GHz, with 32 GB DDR3 (4 x 8 GB) clocked at 1.867 GHz on a Cent OS 7.2 x64 distribution and a Linux 4.5.3 kernel compiled (per [10]) with NVDIMM, PMEMFS, and Direct Access (DAX) supports. We built Redis and supporting libraries using gcc 4.8.5. SCM emulation is done using 6 GB of DRAM held back from kernel page management. We used techniques described in [8] to emulate different SCM speeds. We added new commands to Redis to start, reset, and print such statistics as numbers of transactions opened/closed, streaming operations not requiring atomicity, SCM loads and stores, nesting levels for transactions, non-temporal or streaming stores, number of synchronous (fenced) SCM stores.

We measure in-memory operations without RDB or AOF options, and compare them to file based persistence, including lossy persistence to block media. Below are the two groups of configurations:

Redis In-Memory Only Configurations:

- **C-o-W:** In-memory structures are updated atomically using Copy-on-Write (synchronous, Undo-Logging).
- **LATB:** Local Alias Table Batched method, which updates SCM backed by an asynchronous Undo Log.
- **LAT:** Local Alias Table method with Delayed Write-Back and Redo Log.

File-Based Configurations:

- **SSD:** Updates are logged by non-lossy appends.
- **SSD Lossy:** Appends are grouped and reflected into the log every second.
- **Pmem FS:** Non-lossy appending from every update into a Pmem (SCM) based file.

First, we wanted to compare our LATB method in a heavy-write scenario. We created an in-memory B-Tree to update internal elements atomically. The B-Tree is initialized with 200k random elements, and then we vary the incoming transaction insert request rate and measure the total insert time. The results for C-o-W, LAT, LATB, and a write-through store method are shown in Figure 4. The write-through store method provides the fastest time, however it is not atomic; any failure during a group of writes may corrupt the data structure. This provides a lower bound on the update time required to insert an element in the SCM based, persistent B-Tree. C-o-W performs the slowest, since it must first copy old values in the data structure and persist them to SCM before writing the new value. LAT and LATB both perform similarly well in the heavy write scenario.

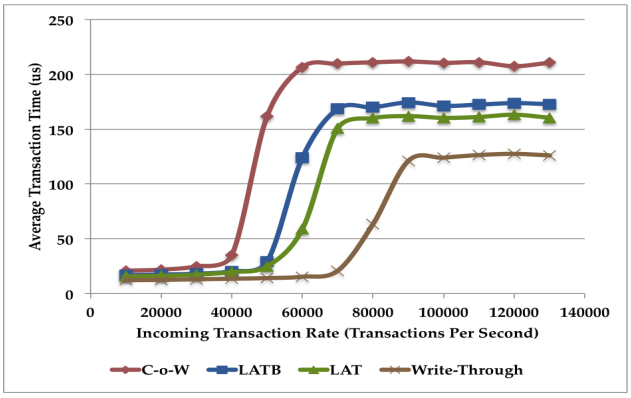

Fig. 4: Average Transaction B-Tree Element Insert Time for Various Byte-Addressable Persistence Methods

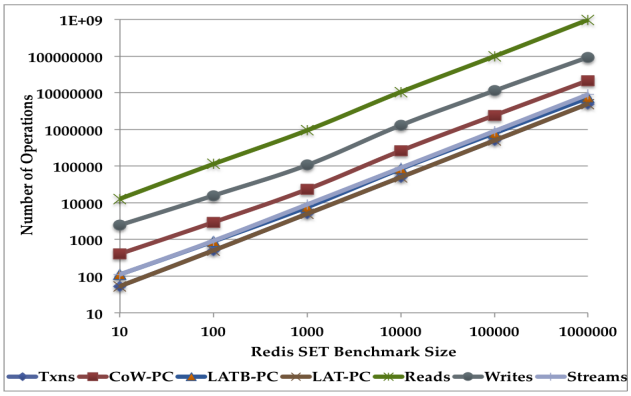

Fig. 5: Number of Operations in the Redis Benchmark Set for Various Byte-Addressable Persistence Methods

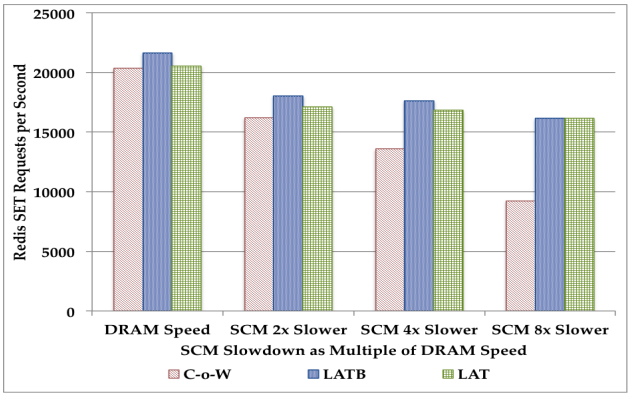

Fig. 6: Redis Benchmark Set Throughput for Byte-Addressable Persistence Methods and Increasing SCM Write Times

Next, we investigate how LATB performs in a Big Data

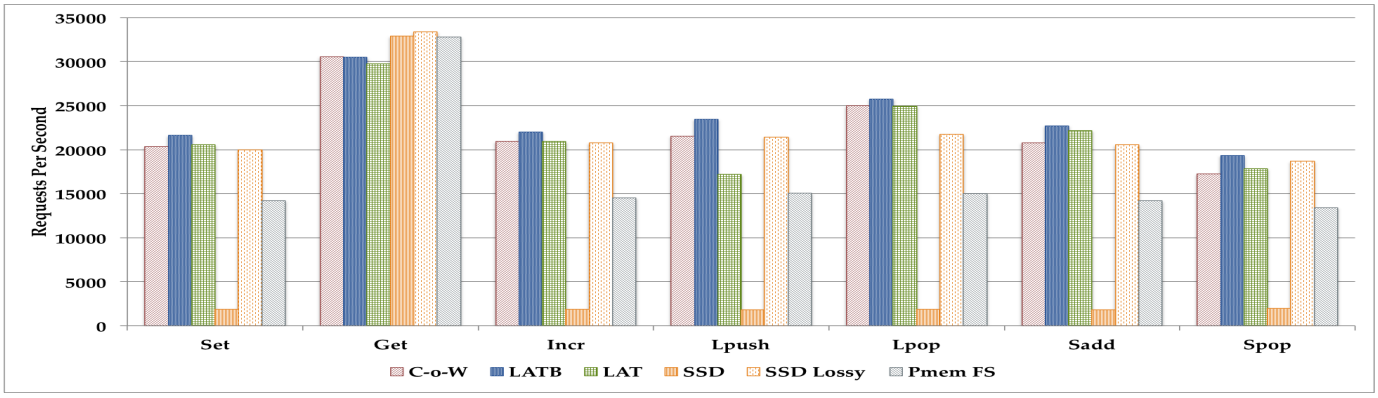

Fig. 7: Redis Benchmark Throughput for Various Byte-Addressable Persistence Methods

TABLE I: Normalized number of operations for internal Redis transactions

Set Size	Normalized Number of Operations in Redis						
	Txns	CoW-PC	LATB-PC	LAT-PC	Reads	Writes	Streams
10	5.30	40.50	10.90	5.30	1281.60	244.80	10.90
100	5.04	29.50	8.87	5.03	1145.67	157.84	9.22
1000	5.00	23.30	7.61	5.00	948.88	106.71	9.02
10000	5.00	26.28	8.41	5.00	1057.72	130.31	9.00
100000	5.00	24.25	7.90	5.00	992.14	113.99	9.00
1000000	5.00	21.64	7.30	5.00	983.33	93.10	9.00

TABLE II: Redis Command Distribution for Redis Benchmark

Command	SET	GET	INCR	LPUSH	LPOP	SADD	SPOP
Percentage	12.5%	12.5%	12.5%	25.0%	12.5%	12.5%	12.5%

benchmark that has a higher percentage of reads. The Redis Benchmark Suite consists of many groups of operations, such as key-value Set/Get, Auto-incremented elements, list and set operations, and others. We increased the random key values used by the benchmark to 5 million in our experiments so as to stress the in-memory data structure store. While the benchmark has options like pipelining requests and using multiple instances in order to calibrate peak operation counts, we do not use them so that we can evaluate individual operations in detail. Table II shows the distribution of commands exercised by the benchmark suite.

Figure 5 depicts data from the Redis set key-value test for varying numbers of set operations. We record the number of transactions, SCM streams, reads, and writes, and the number of persistent commits (i.e., synchronous/fenced stores), for the CoW, LATB, and LAT methods.

The data is also normalized to the number of set operations and shown in Table I. Each set operation generates about five transactional updates to Redis data (col. 1), and this is also the number of PC operations with LAT since that method minimizes persistent commits to 1 per transactional update. Note however the 10-to-1 ratio between reads and writes even though the Set test exercises the least complicated data structure organization. Of these writes, several are not transactional and occur as multi-value writes (captured in the last column, 'Streams') primarily resulting from initialization of newly allocated memory chunks. CoW requires four fold PC operations compared to LAT, and thrice that for LATB – LATB trades a few extra PC operations for reducing number of aliased reads. The number of PCs per transaction is small because only a few synchronous writes are needed to record pointer updates transactionally after data is initialized and streamed into SCM.

TABLE III: Redis Command Distribution for the YCSB-A

Command	HGETALL	HMSET	ZADD
Percentage	16.6%	50.1%	33.3%

Figure 6 charts the throughput for the experiment of Figure 5 across different SCM speeds (in multiples of DRAM speeds). As SCM writes get slower, the LATB method outpaces CoW significantly (by 60% at 8x DRAM speed). LAT, which performs delayed write-backs remains comparable to the LATB method. This underscores the point that curbing the number of persistent commit operations is critical for performance under the wider gap between DRAM and SCM speeds.

Figure 7 compares all the persistence options across all of the tests in the Redis benchmark suite for the case when SCM speed equals DRAM speed. For the Set operation, LATB has the best non-lossy performance, and the lossy SSD configuration does much better than the (non-lossy). For the Get operation, all methods operate largely from memory (and processor caches, actually); here, the in-memory CoW, LATB, and LAT methods underperform slightly as they are instrumented. In all other benchmarks, the LATB method does well due to its ability to keep loads comparable to baseline Redis while performing only a mildly higher number of synchronous stores over the LAT method.

The open source Yahoo! Cloud Servicing Benchmark, or YCSB [6] can be used to test a wide variety of NoSQL stores (including Redis). Table III shows the distribution of the only three Redis commands exercised under YCSB-A workload - HGETALL, HMSET, and ZADD. YCSB-A performs reads and writes in equal measure. Figure 8 compares transactional

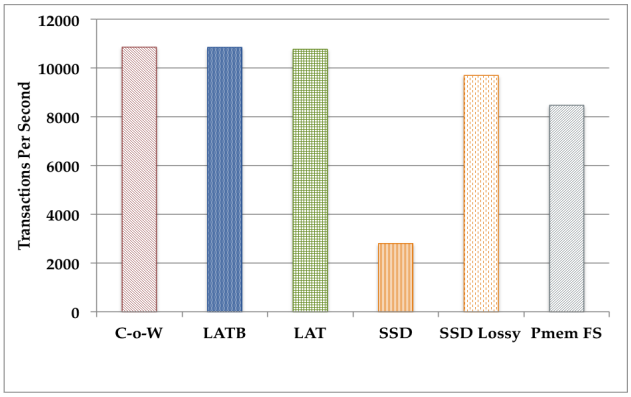

Fig. 8: Throughput for YCSB Workload A with Redis and Various Byte-Addressable Persistence Methods

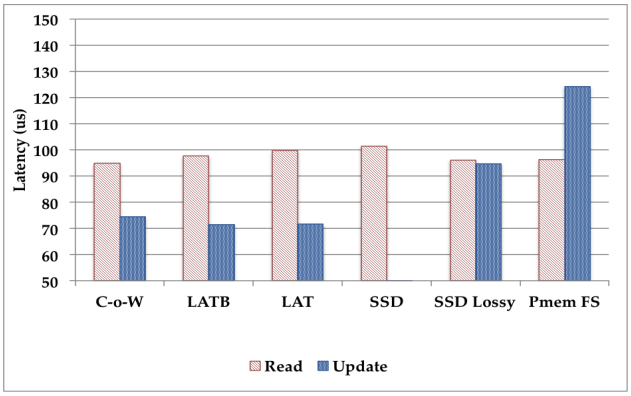

Fig. 9: Read and Update Latency for YCSB Workload A with Redis and Various Byte-Addressable Persistence Methods

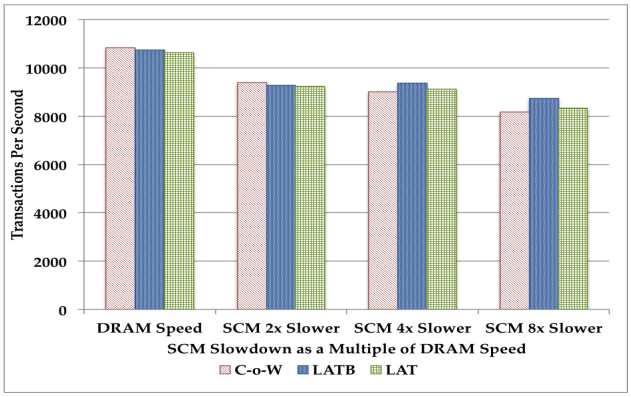

Fig. 10: Throughput for Workload A of the YCSB with Redis and Increasing SCM Write Times

throughputs among the six options, a discussion of which follows. The three in-memory methods, CoW, LAT and LATB all perform comparably. SSD lossy outperforms PmemFS by a small margin, and all methods do significantly better than SSD (non-lossy). Figure 9 depicts operation latencies (in microseconds) for the same test among the six options. While read operations take about equal time among the options,

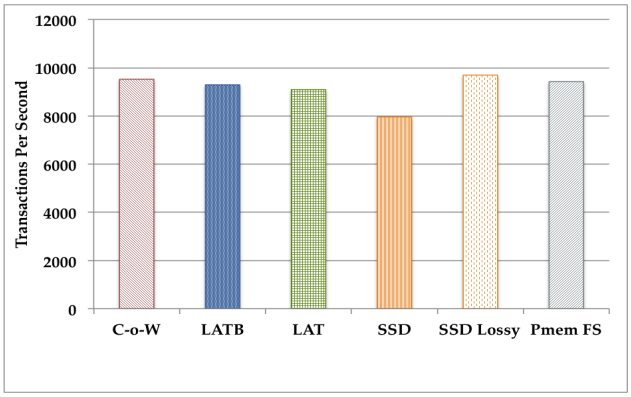

Fig. 11: Throughput for Workload B of the YCSB with Redis and Various Byte-Addressable Persistence Methods

the update latency for SSD is 5 times that of other methods due to the latency of committing to disk on each transaction. Figure 10 shows how an increased SCM write time affects the performance of in-memory options, and it also shows that LATB does modestly better with slower SCM write speeds.

Next, Figure 11 shows the results from YCSB Workload B which is read heavy with 95% lookups and only 5% updates. Predictably, CoW has the best performance although performance is roughly comparable across the board since all execute primary from memory (and caches). The five percent updates cause SSD to lag but only by about 15%.

A. *Continuing Work*

Going forward, we plan on further optimizing the Local Alias Table Batched (LATB) approach for higher read performance and to adapt to the workload. We are exploring enhancing it to more quickly adapt to when to best write alias table entries to persistent SCM so that 1) it is not performed too often as to affect write performance, while 2) the write of the LAT entries is not performed too infrequently so reads suffer from having to consult a larger table. We are also analyzing additional workloads for performance comparisons including several micro-benchmarks and popular workloads.

Pipelined Redis operations combine multiple updates into one request and boost performance drastically. For all of the methods the performance boost is large. We are continuing to increase our performance even higher by keeping larger write sets in the alias tables when under heavy write operations, and when the requests are more read based, push contents to main SCM more frequently. With higher SCM write delays, this also has an additional benefit in that writes to the same location in SCM are only performed once.

Our process of adapting Redis to use SCM safely required using a Pin tool and manual editing of source files and updating memory allocation schemes. We are exploring a process to allow for easy porting of Big Data applications to use SCM safely and effectively with a decrease in development effort.

V. Related Work

Analysis of consistency models for persistent memory was considered in [11]. Software approaches to SCM atomicity rely on simple hardware support such as atomic 8-byte writes and memory fences with persistence semantics. We similarly rely on these hardware capabilities. Memory controller designs for persistent memory have been proposed in [12]–[16]. Unlike other approaches, the controller in [14], [15] does not require changes to the existing cache hierarchy. Whole-system persistence [17] snapshots all micro architectural state on impending failure and requires extensive hardware support.

Application-level control of consistency is discussed in RVM [18]. Specialized reliable SCM software structures like B-Trees [19], multi-versioned indexes [20], [21], and NV heaps [22] have been proposed recently; these often require changes to the underlying system architecture.

Mnemosyne [23] uses software transactional memory (STM) to intercept transactional writes and reads and integrate concurrency control and atomicity. The approach has the advantage of exploiting a single framework, but forces a single concurrency control model (TM), that may not fit legacy applications. ATLAS [24] uses a compiler pass to automatically generate transactional regions for atomic writes utilizing a synchronous undo log, but faces the same disadvantage of coupling distinct concerns in a single framework.

An early version of the SoftWrAP technique [8] was introduced in [25], and a similar approach is described in REWIND [26], which offers a more elaborate alternative for in-place updates with multilayered recovery aided by an Atomic Doubly Linked List. A technique for ensuring atomicity of **sync** was discussed in [27]. Using Local Alias Tables was compared to SoftWrAP in [5]. File system support for NVM is presented in PMFS [28], NOVA [29], Aerie [30], and SCMFS [31], along with a recently released persistent memory library by Intel [10].

VI. Summary

Redis is a popular high-performance in-memory data store that offers in-memory and persistence operations for data. Storage Class Memory is an exciting new memory technology offering byte-addressable persistence alongside DRAM on the main memory bus. Combining Redis with persistent byte-addressable memory is a natural fit for Big Data applications.

In this paper, we present various methods to achieve high-performance byte-addressable persistence for Redis. We modified the Redis source code to use SCM safely, so that in-memory updates to data structures are performed safely in light of failure. We evaluated Redis using both internal benchmarks and the Yahoo! Cloud Servicing Benchmark. We found that even though Redis uses many SCM read operations, it can benefit from highly optimized persistent SCM write based approaches, especially when SCM write times are longer than DRAM write times. Our LAT Batched method combines a fast atomic write approach using aliasing with the high read performance of Copy-on-Write based approaches. It performs best for most byte-addressable SCM update methods in the Redis benchmarks and YCSB workloads examined.

References

[1] S. Singh and N. Singh, "Big data analytics," in *Communication, Information Computing Technology (ICCICT), 2012 International Conference on*, Oct 2012, pp. 1–4.

[2] P. Russom *et al.*, "Big data analytics," *TDWI Best Practices Report, Fourth Quarter*, 2011.

[3] M. F. Uddin, N. Gupta *et al.*, "Seven v's of big data understanding big data to extract value," in *American Society for Engineering Education (ASEE Zone 1), 2014 Zone 1 Conference of the*. IEEE, 2014, pp. 1–5.

[4] J. Zawodny, "Redis: Lightweight key/value store that goes the extra mile," *Linux Magazine*, vol. 79, 2009.

[5] E. Giles, K. Doshi, and P. Varman, "Transaction local aliasing in storage class memory," in *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 349–350.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

[7] Intel, "Intel architecture instruction set extensions programming reference," http://software.intel.com/.

[8] E. Giles, K. Doshi, and P. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *31st International Conference on Massive Storage Systems and Technology*, 2015.

[9] C.-K. Luk and et al, "Pin: Building customized program analysis tools with dynamic instrumentation," in *ACM PLDI*, 2005.

[10] Pmem.io. (2016) Pereistent Memory Programming, pmem.io. [Online]. Available: http://pmem.io/

[11] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ISCA'14*, 2014, pp. 265–276.

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

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

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

[15] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for scm with a non-intrusive backend controller," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 77–89.

[16] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 421–432. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540744

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

[18] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler, "Lightweight recoverable virtual memory," *ACM Trans. Comput. Syst.*, vol. 12, no. 1, pp. 33–57, Feb. 1994. [Online]. Available: http://doi.acm.org/10.1145/174613.174615

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

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

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

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

[23] H. Volos, A. J. Tack, and M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM ASPLOS'11*, 2011, pp. 91–104.

[24] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *ACM OOPSLA '14*, 2014, pp. 433–452.

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

[26] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "Rewind: Recovery write-ahead system for in-memory non-volatile data-structures," *VLDB'15*, vol. 8, no. 5, pp. 497–508, 2015.

[27] S. Park, T. Kelly, and K. Shen, "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.

[28] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 15:1–15:15. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592814

[29] J. Xu and S. Swanson, "NOVA: a log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 323–338.

[30] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 14.

[31] X. Wu and A. L. N. Reddy, "Scmfs: a file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 39:1–39:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063436