NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories

David Schwalb Markus Dreseler Matthias Uflacker Hasso Plattner
Hasso Plattner Institute, Potsdam, Germany
firstname.lastname@hpi.de

ABSTRACT

Non-volatile RAM (NVRAM) will fundamentally change in-memory databases as data structures do not have to be explicitly backed up to hard drives or SSDs, but can be inherently persistent in main memory. To guarantee consistency even in the case of power failures, programmers need to ensure that data is flushed from volatile CPU caches where it would be susceptible to power outages to NVRAM.

In this paper, we present the NVC-Hashmap, a lock-free hashmap that is used for unordered dictionaries and delta indices in in-memory databases. The NVC-Hashmap is then evaluated in both stand-alone and integrated database benchmarks and compared to a B+-Tree based persistent data structure.

1. INTRODUCTION

Non-Volatile Main Memory (NVRAM) has lately received a great deal of attention in the database community [1, 10, 12, 16]. Because the periodic refreshes known from DRAM are no longer required, NVRAM guarantees data persistence even across power outages. Also, without periodic refreshes, NVRAM will only consume power when being accessed. This in turn will reduce the power consumption of data centers. Unfortunately, the advantages of NVRAM are not for free. Early versions of NVRAM are predicted to have higher access latencies, especially for writes [12]. Also, to make use of its non-volatility, data structures have to be adapted so that modifications reach the NVRAM and are not held in volatile CPU caches.

In this paper, we show how programmers can adapt existing data structures to ensure their persistence on NVRAM. We discuss how building on existing lock-free data structures can help in the process and what changes need to be made. The paper is organized as follows: In Section 2, we outline the different challenges programmers face when using NVRAM, such as dealing with volatile CPU caches and reordering. Also, we compare these challenges with those faced when developing lock-free data structures. After that, Section 3 shows how we adapted an existing hashmap to make use of NVRAM. The integration of the Hashmap into our NVRAM research database HYRISE-NV is described in Section 4. We evaluate and compare the hashmap both in stand-alone benchmarks (Section 5) and in a database context (Section 6). We will show how the necessary adaptations affect the performance of the data structures and how the changed performance characteristics affect the choice of data structures in a DBMS. Finally, we discuss related as well as future work and give concluding remarks in Section 8.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IMDM '15, August 31 2015, Kohala Coast, HI, USA
© 2015 ACM. ISBN 978-1-4503-3713-7/15/08...$15.00
DOI: http://dx.doi.org/10.1145/2803140.2803144

2. CHALLENGES FOR NON-VOLATILE DATA STRUCTURES

NVRAM comes with new challenges as the consistency and persistence of in-memory data structures has to be guaranteed even across system crashes. We will look at the complications and show solutions to these.

There are different solutions proposed for this problem that would allow the hardware or a low-level software layer handle all of these issues [15, 17]. None of these are yet ready to be used. In the absence of such a holistic approach, the task of guaranteeing correct cache flushes and execution orders remains with the programmer.

Traditionally, the caches are designed to be transparent to the programmer. With NVRAM, however, the programmer has to be aware where the data is currently stored. This crash has to be assumed to be a complete power loss in which both SRAM and DRAM lose their contents. Data stored in the volatile components will be lost. Thus, data that has to be preserved must be moved into what is called the *persistence domain*, i.e., one of the non-volatile stores.

To avoid such loss of data, the CPU has to be instructed to flush the contents of its volatile components into the persistence domain. One solution to this challenge is the `CLFLUSH` instruction, writes a cache line's contents to memory. A big disadvantage, however, is that it also invalidates the cache line meaning that this cache line has to be re-retrieved when accessed the next time, causing significant performance costs. This will be alleviated with the arrival of the `CLWB` instruction in future processors, which writes a cache line to memory without invalidating it [11].

There is another aspect to this: While `CLFLUSH` (or `CLWB`) guarantee that the data has been written to NVRAM, it might have also previously been written as the result of a cache eviction. Programmers have no control over this. If the system crashes in the middle of an operation that modified multiple addresses, some of these might have been evicted to NVRAM (where they endure the crash) while

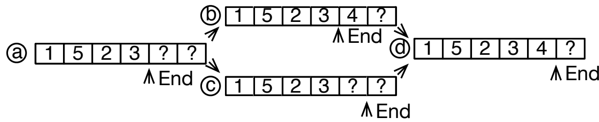

Figure 1: Inserts into a vector can be inconsistent if writes are reordered

other modifications get lost. This almost inevitably leads to data corruption. Thus, the programmer must ensure that premature evictions do not affect the consistency of the data.

Recent work [5] suggests that, in addition to the CPU caches, additional memory buffers have to be taken into account when flushing to the persistence domain. These buffers, as well, are volatile and are said to require explicit flushes. Intel introduced the `PCOMMIT` instruction for this. As neither `CLWB` nor `PCOMMIT` are available in current CPUs, we will use `CLFLUSH` in the following.

2.1 Reordering

As discussed above, the order in which data is written to NVRAM can affect its consistency. But not only premature cache evictions can affect the order in which modifications reach the persistence domain. Both the compiler and the CPU may also reorder operations in a way that puts the consistency at risk. An example of this is shown in Figure 1: For adding a value (here 4) to a vector, two steps are necessary: The size of the vector has to be increased by moving the end pointer (assuming sufficient capacity) and the value has to be written. If the value is written first (A→B→D), a crash in between would not compromise the consistency of the vector. The new value is beyond the end pointer and is thus ignored. If, however, the end pointer is moved first (A→C→D), the vector contains an uninitialized value in step C. If a crash were to occur, a wrong value would be included in the vector.

To make sure that reordering does not change the write order in a way that compromises the consistency, programmers have to use memory barriers (i.e., `SFENCE`/`MFENCE`) that enforce ordering both with regards to compiler optimizations and the CPU's out-of-order execution.

2.2 Recoverability

The system can crash at every point during the program's execution. Recoverability has to be ensured by making sure that state transitions do not leave the data structure in a state in which its consistency is jeopardized. Ensuring this can happen with single atomic writes (such as shown in Figure 1), logging, copy-on-write, or versioning [5, 19].

A common theme is to use transitions from a consistent state via a recoverable state to another consistent state. That recoverable state, while not consistent itself, can be repaired after a crash so that the data structure returns to its previous state. This is similar to undo logs in databases. An example of this would be a (single-threaded) double-linked list. Inserting a new element into the middle could be implemented as (1) allocating and writing the element, (2) updating the *prev* pointer of the following element, and (3) updating the *next* pointer of the previous element. A crash after the first step would not impact the consistency of the data structure. However, the NVRAM allocator would have to clean up the element to avoid memory leaks. The transition described by the second step is one that moves the data structure from a consistent state to a recoverable state. If the system crashes after writing the *prev* pointer, but before writing the *next* pointer, the list is corrupted. By traversing the list during the recovery process, this state can be identified and repaired by resetting the *prev* pointer and undoing the partial insertion. If no crash occurs, the last step - writing the *next* pointer - moves the data structure to a consistent state.

2.3 Similarities with Lock-free Programming

Lock-free data structures, as well, require atomic state transitions from one consistent state to another. One way to guarantee this is Linearizability [9]. It is used to show that other threads that access the data structure in between do not see a transient, inconsistent version of the data. Linearizability is a good way to identify positions in the code of a lock-free data structure at which flushes to NVRAM have to be inserted. This is because every linearizable operation has a Linearization Point [8, p. 55] at which "[f]or implementations that do not use locking [...] the effects of the method call become visible". Before the Linearization Point, the data structure appears to be unchanged to other callers and after the Linearization Point, all changes are visible atomically. As such, adding NVRAM flushes to the Linearization Point will add atomic state transitions even across system crashes. We will show how a flush was added to such a Linearization Point in the following section.

3. THE NVC-HASHMAP

In this section, we present the NVC-Hashmap[1] as an alternative data structure for the use in NV-persisted in-memory databases. The implementation is based on Split-Ordered Lists, an extensible, lock-free hash table [18]. We first discuss the layout of the hashmap and how its lock-free properties help in converting it into an NV-persisted data structure. After this, we describe the implementation changes in depth.

3.1 Data Layout

Most hashmaps store their entries in a number of buckets, each of which holds the entries that share the same hashkey (or a part thereof). In Split-Ordered Lists, on the other hand, all entries are part of a single, forward-linked list. Instead of being a container for a number of entries, buckets are pointers into this list, identifying parts of the list as members of the bucket. "Metaphorically speaking, [the] algorithm differs from prior known algorithms in that extensibility is derived by 'moving the buckets among the items' rather than 'the items among the buckets' " [18]. Instead of inserting entries into a fixed-size bucket, entries are inserted into the list at their appropriate position.

When buckets are pointers to specific entries in the linked list, this becomes a problem when those entries are deleted. To avoid referencing a deleted entry, dummy entries are introduced. These are elements of the linked list that hold a key that belongs into the same bucket as following entries. Dummy entries, however, are not returned as part of a search and are never deleted. As a result, they can be safely referenced by the bucket list.

[1]Non-Volatile and Concurrent Hashmap

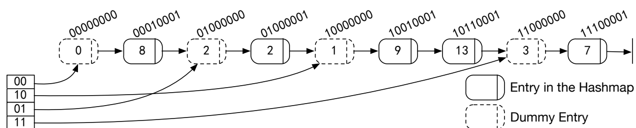

Figure 2: Example of a Split-Ordered List, based on [18]

Figure 2 gives a visualization of such a Split-Ordered List. Dashed entries are dummy entries that are referenced by the bucket list on the left.

3.2 Use of Split-Ordered Lists

The choice for Split-Ordered Lists was made because their layout helps in adapting them for NVRAM. As discussed in Section 1, one of the major challenges for non-volatile (NV) data structures is to allow for atomic updates. While single changes, such as updating a value in a vector, can be done and flushed atomically, more complex changes require some sort of atomicity control. When deciding what data layout to use for the hashmap, this plays an important role.

Take linear hashing [13] as an example. If an insert results in an overflow, a number of changes will be done to the hashmap: A new primary page has to be created, entries need to be moved, and meta data has to be updated. When taking NV atomicity into account, this can only be achieved using versioning or shadow copies.

Split-Ordered Lists, on the other side, are already designed to use compare-and-swap operations to insert new entries into the linked list. When an entry B is inserted between A and C, the previous entry A will either have the old state (pointing to B), or it will have been atomically changed to C. Inserts into the list can be performed without locks by comparing and swapping (i.e., using the CAS operation) the next pointer of the previous element. If a concurrent modification is detected, the CAS would fail and would be retried. The moment in which the compare-and-swap succeeds is the Linearization Point of the insert operation.

Originally designed to allow for lock-free updates, this property becomes an important advantage when storing the hashmap on NVRAM. If every compare-and-swap insert into the hashmap is followed by a flush of the cache line, it is guaranteed that both the in-use version (which might be partially in the CPU cache) and the on-NVRAM version are consistent. We will show this in the next subsection.

3.3 Implementation

Our implementation is based on the open-source version of Intel's Threading Building Blocks library, more specifically the `tbb::concurrent_unordered_map`.

When adapting an existing data structure for use with NVRAM, four design decisions have to be made:

1. How is non-volatile memory managed? List nodes have to be dynamically allocated and freed. At the same time, memory leaks from incomplete object creations have to be identified and fixed.
2. How are object references kept valid if the address layout changes? All known methods to access NVRAM from the user-space are based on file systems from which files are `mmap`ped into the address space of the process. After a crash, the layout of this user address space may have changed; for example because of Address Space Layout Randomization (ASLR).
3. How to find an entry point after recovery? When the file has been mapped into the address space, it only contains a number of objects that are referencing each other, but no pointer into the file exists.
4. How to deal with incomplete updates? As both the compiler and the CPU's cache eviction policies might cause updates to be executed out-of-order, the programmer has to use fences and cache-line flushes in order to avoid erroneous states from which recovery might be impossible.

For the NVC-Hashmap, we made the following decisions: With regards to (1), the Persistent Memory File System (PMFS) [5] and an NVRAM allocator, `pmemalloc`[2] were used. For dealing with pointers to objects (2), the hashmap stores pointers (e.g., to the following list node) relative to the start position of the memory-mapped file. Before being followed as part of operations on the data structures, these relative pointers are converted into absolute addresses (`abs_x`) by adding the base address of the `mmap`ed file. Correspondingly, absolute pointers are converted to relative pointers (`rel_x`) by subtracting the base address before storing them on NVRAM. `pmemalloc` also allows to recover the entry point of the data structure (in our case, a pointer to the bucket list) by storing it in a static area at the beginning of the PMFS file. Additionally, information such as the maximum bucket size are stored in that area as well.

When dealing with incomplete updates (4), we analyzed the Linearization Points of the operations to find correct positions to flush changes to NVRAM. We demonstrate this by going through the insertion of a new entry into the hashmap as shown in Figure 3. After creating the node to be inserted, called `new_node` (1), its relative pointer `rel_next` is set to the following element (`current_node`). By assigning a value to `rel_next` (2), that value is stored in the CPU cache, but not necessarily on NVRAM. This is denoted by the dashed and solid lines. Next, the `rel_next` pointer is persisted by using a method of the pmem library that performs the `CLFLUSH` and adds a memory fence (3). Up to now, the list (and on a higher level, the hashmap) has not been modified. This happens in step 4, when the `rel_next` pointer of the previous element is changed to point to the new element using compare-and-swap. This atomically inserts the element into the list and into the hashmap. The current version of the hashmap now includes the new element. However, if the system were to crash here, the updated pointer might not have been written to the persistence domain. This happens in the final step (5) where a second call to `pmem_persist` flushes the updated pointer to NVRAM.

Similar adaptations have been made to other modifying steps, such as updating the bucket list. These, as well, follow the principle of first persisting newly created objects and then finalizing the modification on NVRAM right after the Linearization Point.

4. NVRAM IN IN-MEMORY DATABASES

We will now show how the NVC-Hashmap can be integrated into a DBMS. Hashmaps are used in databases both for index structures and as dictionaries for dictionary-compressed columns. To benchmark our NVC-Hashmap, we added it into HYRISE-NV, a modified version of the

[2] https://github.com/pmem/linux-examples

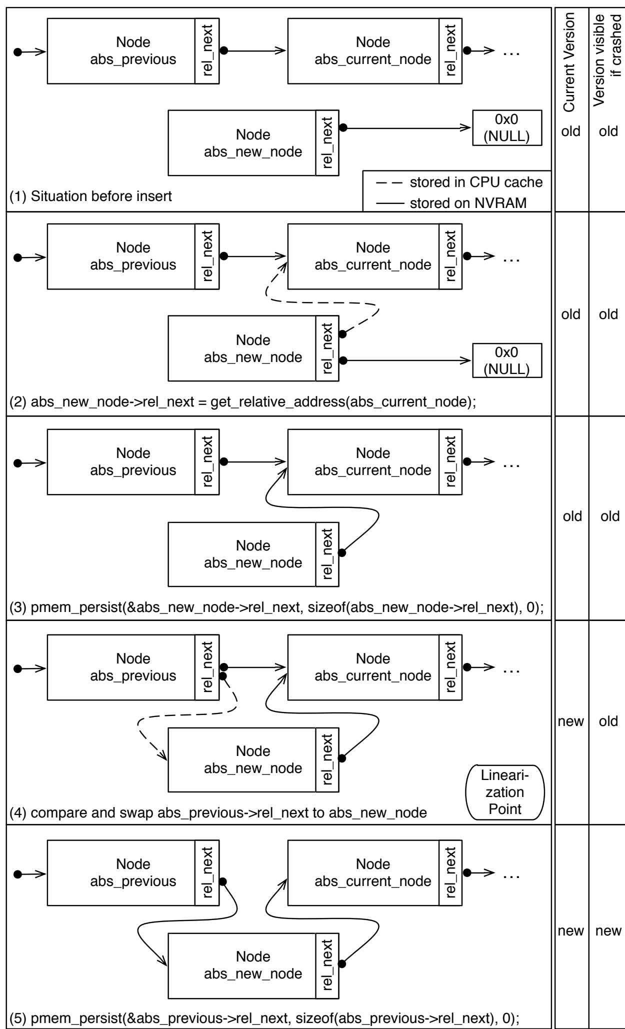

Figure 3: Five steps are needed to guarantee persistent inserts into an NVC-Hashmap

open-source research database HYRISE[3]. This way, we can compare it to the other persistent data structure used, the NVB+-Tree which we will describe later in this section.

4.1 HYRISE-NV

HYRISE is an in-memory database that supports hybrid row and column layouts [7]. With fine-grained control over which columns are stored together, it balances OLTP and OLAP needs. Data is compressed using dictionary compression which replaces values by value ids stored in a dictionary. This reduces the memory footprint and improves the performance as most operators can operate on small integer value ids instead of more expensive data types. Value ids are stored in a contiguous in-memory attribute vector. Additional performance improvements are achieved by keeping the dictionaries sorted so that an order on the value ids reflects an order on their associated values. Because adding a value into the middle of these dictionaries would require a rewrite of the table, values are separated into a main partition with sorted dictionaries and a delta partition with unsorted dictionaries. These are periodically merged. For storing these dictionaries, both trees and hashmaps can be used. We will show the performance implications in Section 6. Both the main and the delta partition can be indexed. While the main partition uses an optimized vector-based index [6], the delta index uses either a tree or a hashmap. Again, these will be evaluated later on. In addition to the attribute vectors, dictionaries, and indices, tables also store visibility information for each row. Employing Multi-Version Concurrency Control (MVCC), Hyrise uses this information to isolate transactions and to detect transaction conflicts.

To ensure persistence on NVRAM, a number of data structures have to be kept on NVRAM. First of all, the attribute vectors and the associated dictionaries are needed to restore the information in the table. Furthermore, the MVCC information is preserved so that deletes and incomplete inserts can be identified after recovery from NVRAM. Additionally, the state of the Transaction Manager is partially preserved. Finally, to improve recovery speed, indices are persisted as well to avoid expensive rebuilds after the system restarts.

We identified two places in HYRISE-NV which can profit from the presented NVC-Hashmap: the uncompressed dictionaries and the delta indices. Both of these currently use a different, B+-Tree based data structure, the NVB+-Tree.

4.2 NVB+-Tree

HYRISE-NV currently uses the "NVB+-Tree", an internal data structure based on the STX B+-Tree[4] which was adapted to be persisted on NVRAM. This was done by introducing versioning similar to Venkataraman's CDDS Tree [19]. As in MVCC, begin and end versions of tree nodes are stored and a global "current version" determines which nodes are visible. Modifications of the tree have to be flushed to the persistence domain starting with the updated leaf node and up to the highest changed node. After this, the global "current version" information can be updated and flushed. While this guarantees atomic updates even if the system crashes, it requires a high number of NVRAM flushes, especially in the case of node splits. Here, both the left and the right node, as well as the parent node and the "current version" information have to be flushed. If inserting the split node into its parent node causes that node to overflow, even more flushes are required.

5. STAND-ALONE EVALUATION

We evaluated the NVC-Hashmap in stand-alone benchmarks and when used as a persistent data structure within Hyrise-NV. The purpose of these evaluations is to measure the overhead introduced by persisting the data in the hashmap to NVRAM and to compare it to that of an alternative data structure, the NVB+-Tree.

All benchmarks were executed on a platform with two blades, each having four Intel E7-8870 processors clocked at 2.40 GHz with power management disabled. We used a single NUMA node to remove NUMA effects from the measurements. The system has 1.5 TB of DRAM, running at 1067 MHz. As actual NVRAM hardware was not available, this memory was used for the NVRAM mount as well. While this leaves out the latency effects of NVRAM, it gives accurate results when looking at the system overhead caused by the necessary flushes. Once this latency comes into play,

[3]https://github.com/hyrise/hyrise

[4]https://github.com/bingmann/stx-btree

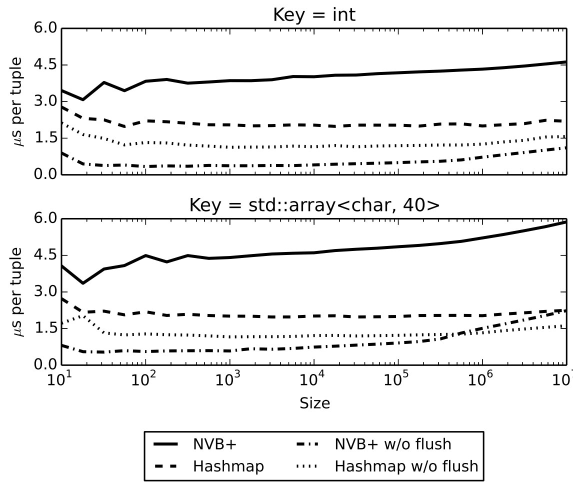

Figure 4: Cost of inserting n values into a map (insert costs per tuple)

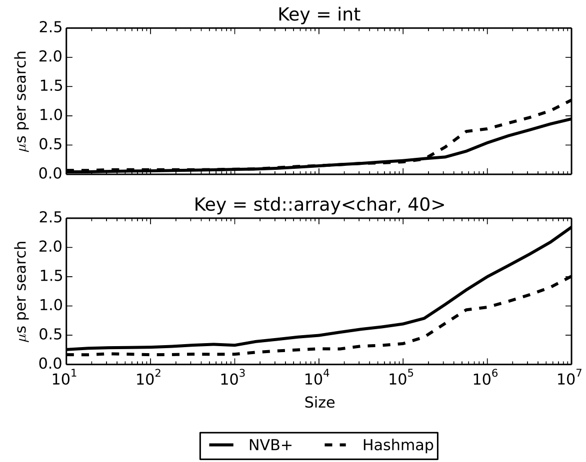

Figure 5: Cost of searching in a map with size=n preloaded entries

it will amplify the difference between the shown data structures. This is because the NVB+-Tree requires significantly more flushes than the NVC-Hashmap. The cost of these flushes increases with increased NVRAM latencies.

In order to simulate the requirements of an in-memory database, two different configurations of the maps were used, both of which support multiple entries for the same key. The first is a map from `int`s to `size_t`, which would be used for example in single-column indices. Second is a map from std::array<char, 40> to `size_t` as an example for the use in a multi-column index.

5.1 Inserts

In the insert benchmark, a varying number of uniformly distributed pairs were inserted into the different maps by a single thread (multithreaded benchmarks will be discussed soon). The distinctivity of was chosen as 10% after preliminary experiments showed that its influence was only minor. The results of the insert benchmark are shown in Figure 4.

Two graphs are shown, both with four lines. The upper graph shows the results for a map that uses an integer as its key, while the lower graph uses the aforementioned array of 40 characters to simulate a multi-column index.

Within each graph, one line shows the performance of the NVB+-Tree and another that of the NVC-Hashmap. In addition, two other lines show the performance of those data structures with `CLFLUSH`es and memory fences disabled.

Looking at the `int` variant, inserts into the NVC-Hashmap are faster than into the NVB+-Tree by a factor of 2x to 2.5x. As expected for hashmaps, the insert performance remains constant while it is increasing for the NVB+-Tree. When comparing the variants without flushes, the NVB+-Tree performs faster inserts than the NVC-Hashmap but still has increasing costs towards the end.

Comparing the map implementations with their no-flush variants, the NVC-Hashmap has an almost constant overhead of 1.4x to 1.8x, while the overhead of the NVB+-Tree decreases from 10x to 4x. This is because the cost of navigating the tree (which does not require flushes) gets higher with increasing tree height while the number of modified tree nodes per operation (which require flushes) stays mostly constant.

When the maps are used to store larger keys, as shown in the second graph, these tendencies remain the same. For the NVB+-Tree, the increased size of the key leads to more expensive inserts both with and without flushes, while the impact on the NVC-Hashmap's performance is significantly less. The reason for this is that the NVB+-Tree needs to write the inserted values on multiple levels while the hashmap only writes them once. This could be alleviated by using compression for inner nodes or by using a trie.

5.2 Reads

In a database context, not only the insert costs, but also read costs are of importance. If a database index is used, this usually means that more reads than inserts are performed. Thus, the read performance of the two implementations is evaluated as well. As the structures are not modified in a read-only benchmark, there are no flushes occurring. Accordingly, the flush and no-flush variants perform equally and only the variant with flushes is plotted. In the benchmark, the tree was first filled with a varying number of entries as described for the insert. Then, one million searches (i.e., `equal_range(key)` calls) were performed.

The results of this benchmark are shown in Figure 5. Surprisingly, the hashmap is not searched in (close to) linear time but in what appears to be logarithmic time. An explanation for this lies in an increasing number of cache misses. All entries of a bucket have to be traversed for a search. As the buckets in a Split-Ordered List are implemented as a forward-list, these entries may be stored all over the NVRAM file, causing one cache miss each. The B+-tree, on the other side, has lower costs because entries with the same key are stored in a lower number of leaf nodes. This means that more entries are stored on a cache line and that fewer cache misses are caused, especially because the inner nodes are likely to be kept in the caches. This effect does not apply to the insert benchmark because inserts into a Split-Ordered List do not traverse the bucket but insert the new list element in the front of the bucket.

Again, the array variant is more expensive than the `int` variant because key comparisons become more expensive. Especially the tree suffers from this. For the future, we are

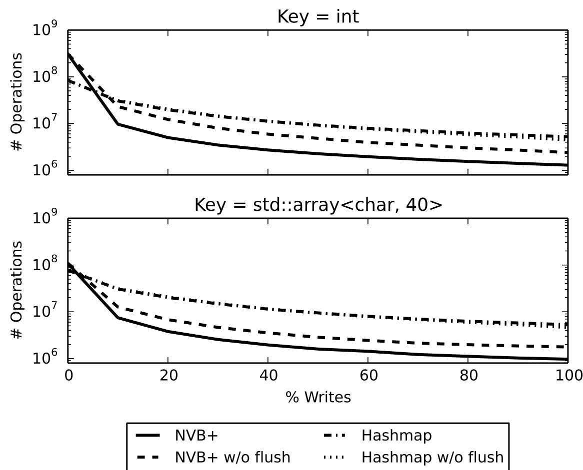

Figure 6: Number of operations completed by 20 threads with varying number of writes

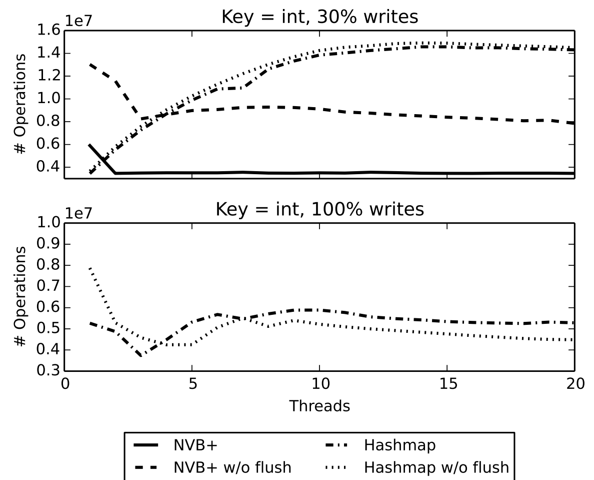

Figure 7: Number of operations completed by varying number of threads

looking into measuring tree structures optimized for longer keys, such as tries.

5.3 Multi-Threaded Benchmark

An important difference between the two map implementations is that while the NVB+-Tree supports a single writer (protected by a mutex) and multiple readers, the hashmap is lock-free and can be written to by multiple threads. To show the influence of this, the performance with 20 threads performing reads and writes on a map with one million pre-inserted values is shown in Figure 6.

Both implementations experience significant slow-downs as the write ratio is increased. The sharp decrease in the performance of the NVB+-Tree, even with only few writers, is explained by the overhead of and the contention on the mutex preventing multiple writes. Not only does the flush increase the latency of a single operation. With multiple threads, it now also increases the time the mutex is held and that other threads have to wait.

In the graph, the line of the two hashmap variants, with and without flushes, overlap. This might be surprising as at least some overhead from the flushes was expected, similar to what was experienced in the insert experiment. It can, however, be explained when looking at Figure 7, a graph plotting the scalability of the data structures.

When single-threaded, the NVB+-Tree (with flushes) performs better than the hashmap. Otherwise, the advantage is on the side of the hashmap which can utilize its lock-free characteristics. Looking at 100% writes, we find that the NVB+-Tree shows performance characteristics similar to those measured with 30% writes. For the hashmap, the results for the single-threaded execution with 100% writes are in line with those in Figure 4. As soon as multiple threads are used, the overhead vanishes. This is an interesting result as it shows that the flush overhead can be hidden in a multithreaded environment.

At some point the flushing variant even gets faster than the non-flushing variant. At first this is surprising because the flush is an additional overhead and should decrease the overall performance. This behavior is not unheard of for lock-free data structures [2]. Looking at the implementation of the hashmap, it can be explained by the contention around the compare-and-swap operation (CAS). If a CAS fails, the work done so far has to be voided. Furthermore, failed swaps also influence other threads. We measured the number of failed CAS operations when updating a nodeâĂŹs next pointer (see Figure 3) and found that the non-flushing variant has 3-4x more failed swaps.

5.4 Interpretation

We interpret the benchmarks shown above as follows:

1. In any case, writes are faster with our NVC-Hashmap than with the NVB+-Tree.
2. The NVB+-Tree has a better read performance.
3. For single-threaded use, the NVB+-Tree shows a better performance in the case of mixed workloads.
4. The overhead of flushing is bigger for the NVB+-Tree than for the hashmap.
5. Flushing drastically decreases the speed of the operations.
6. This effect increases when other threads have to wait for the flush to finish.
7. With multiple threads, the hashmap is performing better.
8. The influence of flushes can be hidden in multi-threaded, lock-free data structures. This is interesting for the development of future NV-aware data structures.

6. DBMS EVALUATION

After showing the stand-alone performance of the data structures, this section focuses on their use in an In-Memory Database, HYRISE-NV. We executed the TPC-C benchmark using `py-tpcc`[5]. To focus on the flush overhead, only write-heavy NewOrder transactions were used. Both unsorted (delta) dictionaries and delta indices used the NVC-Hashmap (or the NVB+-Tree respectively).

Figure 8 shows a number of differences between the two map implementations. “None” is a build without any persistence, “NVRAM” a build with our proposed changes for Non-Volatile RAM. First looking at the unpersisted NVB+-Tree, we notice that during the 30 seconds of execution, the overall throughput increases from 11500 to 13000 transactions per second. This is because in the beginning, the

[5]https://github.com/apavlo/py-tpcc

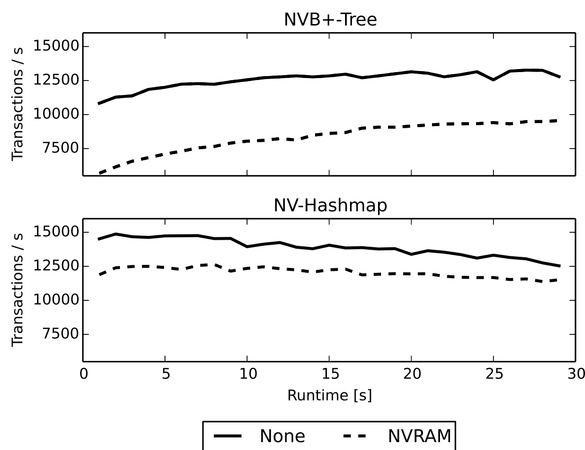

Figure 8: NewOrder transaction throughput with different map implementations

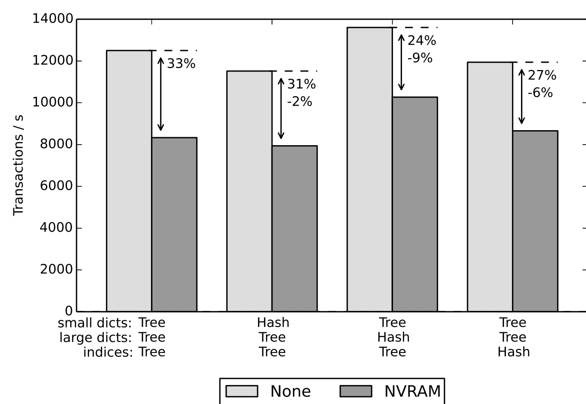

Figure 9: Performance implications of replacing NVB+-Trees with NVC-Hashmaps for small integer dictionaries, large string dictionaries, and indices

delta dictionaries are empty and need to be filled, causing a higher number of writes than are done towards the end. As seen in Figure 4, inserting into trees is expensive. This explains why the throughput gets higher when most values are already in the dictionary. For NVRAM, the throughput also increases from 6000 to 9000 transactions/s. The overhead of the NVRAM gets smaller over time (from 5500 Tx/s down to 4000). This again is explained by a lower number of dictionary inserts and a resulting lower number of flushes.

For the hashmap, three differences are observed. First, the overall performance of the hashmap is significantly higher (12% for the None build and 44% for the NVRAM build). This is in line with our expectations and our previous experiments. Secondly, the performance decreases over time. The reason for this can be found in our concurrency model: The more updates have been committed, the more invalidated rows are in the table. As these rows cannot be deleted from the index (running transactions may still work on an earlier snapshot), more and more rows have to be validated over time. This also happens for the NVB+-Tree. There, the effect is de-emphasized by the decreasing dictionary costs as explained before, explaining why it cannot be seen in the graph. Thirdly, the performance of the NVRAM build is now much closer to that of the unpersisted build. This is because hashmaps need fewer flushes and should therefore have a lower NV overhead. This is the case, as the experienced average overhead of 14% is lower than that of the NVB+-Tree (33%).

As seen in the stand-alone benchmarks (Section 5), the size of the keys and the workload both impact which map implementation performs better. Consequently, we will show measurements where only selected parts of the system were switched from the NVB+-Tree to the NVC-Hashmap. We measured the performance changes when replacing small dictionaries (with a key type of integers or floats), large (string) dictionaries, or indices. This is shown in Figure 9. The percentages given next to the bars are first the relative overhead of using NVRAM and second how that overhead compares to the variant using only NVB+-Trees (in percentage points).

When replacing the smaller dictionaries with hashmaps, the performance slightly decreases. This is in line with previous experiments showing that for small keys that are mostly read, the tree is a more suitable data structure. Even though the performance overhead of NVRAM decreases by 2 percentage points when using the NVC-Hashmap, the hashmap is still slower in absolute numbers.

This is different for the large dictionaries (here, string dictionaries). As explained in the stand-alone benchmarks, hashmaps are the map structure of choice here. Not only does the introduction of hashmaps increase the performance of both the None and the NVRAM build, it also decreases the overhead of NVRAM by 9 percentage points.

Replacing the trees in the indices with hashmaps gives an interesting result. For the non-NVRAM build, it decreases the performance by 5%, but for the NVRAM build it increases it by 4%, reducing the NVRAM overhead by 6 percentage points. This is interesting as it shows how moving to NVRAM affects the choice of data structures. While previously, trees were the structure of choice, their increased NVRAM overhead when compared to hashmaps shifts the advantage to the hashmap when using NVRAM.

6.1 Interpretation

From the benchmarks on the database, we draw the following conclusions:

1. As experienced in the stand-alone benchmarks, flushing the cache lines has a significant cost.
2. For the TPC-C benchmark as executed, the overhead of persisting on NVRAM is 33% with the NVB+-Tree and 14% with the NVC-Hashmap.
3. Different data structures within the database get varying performance gains when switched to NVC-Hashmaps. A careful choice of data structures is important.
4. A performance advantage of one data structure over another in a no-persistence database does not necessarily translate to the same advantage when NVRAM is used. In fact, data structures that were slower before may prevail once the flushes contribute to the costs.

7. RELATED WORK

In addition to the CDDS-Tree [19] on which the NVB+-Tree in HYRISE-NV is based, other researches have proposed approaches to persist trees on NVRAM. To our knowledge, this is the first work looking at hashmaps on NVRAM.

Chi et al. [4] propose different modifications to the classic

B+-Tree that reduce the number of writes, and thus, the resulting wear on Phase-Change Memory (PCM). This is done by leaving nodes unsorted, using temporary overflow nodes instead of splitting nodes whenever necessary, and delaying the merge of underful nodes. The evaluation shows that, in most cases, the proposed changes result in fewer writes but a higher execution time. For the purpose of the work, this is acceptable as it is a trade-off for reducing the wear of the PCM. We believe that there is more in it which might turn the overhead into an advantage: The evaluation shown in the paper does not take into account the costs of flushing the modified nodes. By reducing the number of writes, one would also reduce the number of required flushes, which we have shown to be of a significant cost for B+-Trees. We are currently discussing to also remove the ordering of the nodes solely to decrease the number of flushes.

Moraru et al. [14] discuss a consistent and durable B+-Tree as an example of how their allocator, `nvmalloc` can be used. It requires a new type of hardware support from the CPU, called cache line counters. The programmer would mark a range of writes as one "logical update group" using a special `sgroup` instruction.

Chen et al. [3] propose the wB+-Tree, which allows for write atomic tree modifications, either by atomically updating the nodes or by employing redo-only logging. They compare their approach to other B+-Tree implementations and report a significant performance improvement over other persistence implementations.

8. FUTURE WORK AND SUMMARY

The experiments have shown the importance of avoiding flushes to NVRAM at (almost) any cost. Further work towards NVRAM-supporting databases will have to revisit previously made design decisions to see if the chosen data structures are still the data structures of choice for NVRAM. In this project, future work will benchmark the predictable performance gains that will come with the `CLWB` instruction.

A question undiscussed in this work is that of designing a good allocator suitable for NVRAM. The used `pmemalloc` library is a research example and allows for future improvements. For example, it is currently not multithreaded and hands out memory in a first-fit fashion. Future work could look at how existing allocator concepts can be brought to work consistently with NVRAM.

We presented the NVC-Hashmap, a concurrent, NVRAM-aware hashmap that can be used to persist unsorted dictionaries and delta indices in databases. Common challenges faced when programming for NVRAM were outlined and their solution for the NVC-Hashmap were shown. We discussed how programmers can profit from existing lock-free data structures by analyzing the Linearization Points and using these for NVRAM flushes. The hashmap was evaluated in both stand-alone benchmarks and in the context of HYRISE-NV, a columnar in-memory database, where it was compared against the current solution, the NVB+-Tree. Benchmarks showed a significantly better performance when compared to the tree solution, especially in write-heavy workloads. Finally, we report that the costs of RAM flushes becomes important when choosing data structures for databases and that the cost of these flushes may give previously slower data structures advantages over the currently chosen solution.

References

[1] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. "Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems". *SIGMOD*. 2015, pp. 707–722.

[2] Brian N Bershad. "Practical Considerations for Non-Blocking Concurrent Objects". *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2007, pp. 264–273.

[3] Shimin Chen and Qin Jin. "Persistent B+-Trees in Non-Volatile Main Memory". *Proceedings of the VLDB Endowment* 8.7 (2015), pp. 786–797.

[4] Ping Chi, Wang-Chien Lee, and Yuan Xie. "Making B+-tree Efficient in PCM-based Main Memory". *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 69–74.

[5] Subramanya R Dulloor et al. "System software for persistent memory". *European Conference on Computer Systems (EuroSys)*. 2014, 15:1–15:15.

[6] Martin Faust, David Schwalb, and Hasso Plattner. "Composite Group-Keys". *International Workshop on In-Memory Data Management and Analytics (IMDM)* (2014), pp. 42–54.

[7] Martin Grund et al. "HYRISE—A Main Memory Hybrid Storage Engine". *Proceedings of the VLDB* 4.2 (2010), pp. 105–116.

[8] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[9] Maurice Herlihy and Jeannette M Wing. "Linearizability: A Correctness Condition for Concurrent Objects". *ACM Transactions on Programming Languages and Systems* 12.3 (1990), pp. 463–492.

[10] Jian Huang, Karsten Schwan, and Moinuddin Qureshi. "NVRAM-aware Logging in Transaction Systems". *Proceedings of the VLDB* 8.4 (2014), pp. 389–400.

[11] Intel Corporation. *Intel® Architecture Instruction Set Extensions Programming Reference*. 2014.

[12] Hideaki Kimura. "FOEDUS: OLTP Engine for a Thousand Cores and NVRAM". *SIGMOD*. 2015, pp. 691–706.

[13] Witold Litwin. "Linear Hashing: A New Tool for File and Table Addressing". *VLDB*. 1980, pp. 212–223.

[14] Iulian Moraru et al. "Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory". *ACM SIGOPS*. 2013.

[15] Dushyanth Narayanan and Orion Hodson. "Whole-System Persistence". *International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS* (2012), pp. 401–410.

[16] Ismail Oukid et al. "Instant Recovery for Main Memory Databases". *Conference on Innovative Data Systems Research (CIDR)*. 2015.

[17] Vasily A Sartakov and Rüdiger Kapitza. "NV-Hypervisor: Hypervisor-based Persistence for Virtual Machines". *IEEE/IFIP International Conference on Dependable Systems and Networks* (2014), pp. 654–659.

[18] Ori Shalev and Nir Shavit. "Split-Ordered Lists: Lock-Free Extensible Hash Tables". *Journal of the ACM* 53.3 (2006), pp. 397–405.

[19] Shivaram Venkataraman et al. "Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory". *USENIX Conference on File and Storage Technologies FAST* (2010), pp. 1–15.