Non-volatile memory through customized key-value stores

Leonardo Mármol
VMware

Jorge Guerra
VMware

Marcos K. Aguilera
VMware

1 Introduction

Non-volatile memory, or NVM, is coming. Several technologies are maturing (FeRAM, ReRAM, PCM, DWM, FJG RAM), and soon we expect products from Intel, Micron, HP, SanDisk, and/or Samsung. Some of these products promise memory density close to flash and performance within a reasonable factor of DRAM. This technology could substantially improve the performance of software systems, especially storage systems.

Unfortunately, using NVM is hard: each technology has its quirks, and the details of products are not yet available. We need a way to integrate NVM into our software systems, without full knowledge of all the NVM product details and without having to redesign every software system for each forthcoming NVM technology.

We advocate the use of *customized* key-value stores. Rather than programming directly on NVM, developers (1) design a key-value store customized for the application, (2) implement the key-value store for the target NVM technology, and (3) program the application using the key-value store. When new NVM products emerge, with similar performance characteristics but different access mechanisms, developers need only modify the key-value store implementation, which is simpler, faster, and cheaper than redesigning the application. Thus, the key-value store serves as a middle layer that hides the details of the NVM technology, while providing a simple and familiar interface to the application. Customization ensures that the design is performant and simple.

We illustrate this idea with an example, METRADB, a key-value store that we customize for VSAN, a distributed storage system offered by VMWARE. Key-value stores vary in functionality. Some provide rich data types; some have variable-length keys; and some support transactions with various assurances. The ideal store for an application depends on its needs. We explain the options, so that developers can decide what they need. In METRADB, we chose these options to satisfy the needs of VSAN.

We report on an early performance evaluation, which compares METRADB against a general solution provided by Intel, namely, the data structures in NVML [13]. We find that METRADB's performs 2.2x to 50x better in terms of latency, and its throughput scales well with the number of threads. This performance advantage comes from the customized functionality of METRADB. The trade-off of customization is that developers must build a store for each application. This was not a significant issue for METRADB: it has only 2.3K lines of C code.

We are not the first to propose a key-value store over NVM. Prior work includes Echo [1], CDDS [14], wB+Trees [3], and the data structures provided with Intel's NVML [13]. These are general solutions with broad scope, while we advocate a custom solution for each application. Similarly, other ideas to facilitate the use of NVM include persistent regions [15], persistent objects [4, 9], file systems [5, 6], persistent transactions [8, 15], memory management [11], and durability of lock-based code [2]. Again, these ideas target generality. Other works propose relaxed ordering of writes to NVM [12, 10], nonvolatile caches [16], and other software-hardware architectures [7]. These idea are orthogonal to and motivate our work, since they facilitate access to NVM by proposing alternative hardware.

2 NVM: benefits and challenges

By NVM we mean memory that preserves its contents when power is lost and that can be accessed at a granularity of bytes or words rather than blocks. While there are many NVM technologies, the most mature and promising one is 3D XPoint from Intel and Micron. This technology bring two key benefits relative to DRAM:

- *Non-volatility.* The memory survives power cycles, so software need not resort to slow disks or flash.
- *Density.* DRAM is limited to a few terabytes per machine, but NVM can grow to tens of terabytes.

Relative to disks and SSDs, NVM has two advantages:

- *Performance.* Disks and SSDs can incur large write latencies. NVM has performance closer to DRAM.
- *Fine granularity.* Disks and SSDs operate on 512- or 4096-byte blocks, making them inefficient for small operations; NVM operates on individual words.

However, NVM brings many challenges to developers:

- *Non-persistent caching:* While memory is nonvolatile, memory caches are not. When the CPU stores data, it remains in a cache until it is flushed.
- *Out-of-order flushes:* Caches may be flushed without asking and out of order.
- *Torn writes:* Applications may explicitly flush data, but if power is lost during the flush, only some parts will be persistent.
- *Complex interface:* To persist data, applications must follow a ritual of instructions: flushing dirty cache lines, issuing a barrier, committing data, and issuing

another barrier (§6.3). This ritual is expensive, so developers must worry about how to minimize its use.
- *Non-uniform wear:* As the hardware provides no wear leveling, if a word in NVM changes more often than another, it will wear out more quickly. This can lead to reliability issues if the developer is not careful.
- *Lack of details:* We lack details about cost and performance of NVM.

The first four issues above relate to crash recovery, while the last two issues relate to normal operation. We next describe a middle layer for accessing NVM that can mitigate these issues.

3 Main idea and rationale

Instead of programming on NVM directly, we believe developers should introduce a middle layer that hides the complexity of the NVM. When a new NVM technology emerges, developers need not modify the application, just the middle layer. This layer consists of a key-value store library with transactions. A key-value store offers operations to write and read key-value pairs, and possibly more functionality, depending on the store. A key-value store is a good abstraction: it is simple and familiar, it can be implemented easily, and it can perform well.

But a plain key-value store does not suffice to address the above challenges (§2); it must also provide *transactions* to avoid torn writes and facilitate crash recovery.

We propose that the key-value store be customized for an application. Many applications do not need all features possible in a key-value store (discussed next), and those features may hinder performance and simplicity.

4 Features of key-value stores

Broadly, key-value stores permit users to put, get, and delete key-value pairs, but key-value stores vary in their exact functionality and data models:

- *Key length.* Keys can be fixed- or variable-length, and fixed-length keys can be sparse (e.g., 8-byte integers) or dense (e.g., 0..32K).
- *Value length.* Values can be fixed- or variable-length.
- *Value types.* Values can be typed or opaque. Typed-values can offer richer operations, such as increment for numerical types or set union for set types.
- *Operations.* Besides gets and puts of key-value pairs, the store may support: (a) gets and puts of partial values given by an offset and length, (b) multigets and multiputs of many pairs at once, (c) key enumeration, ordered or not, and (d) read-modify-write.
- *Containers.* If present, containers provide different namespaces to avoid key clashes across users.
- *Transactions.* If present, transactions provide atomicity, isolation, or both. Atomicity protects against crashes and torn writes. Isolation protects against concurrent access.

5 Case study: VSAN

We now explain how NVM can be used for a specific application, a VMWARE product called VSAN. VSAN is a distributed storage system that provides logical volumes, where each volume is partitioned and replicated across storage servers. A server stores the volumes on disks or SSDs; in the former case, it can use SSDs as a server-side cache. In addition, clients cache data in memory for efficiency. We envision three uses of NVM in VSAN.

SSD cache metadata. Servers have an SSD block cache; this cache needs a map from SSD to disk LBAs. The map can exceed the DRAM allocated to VSAN. Moreover, reconstructing the map upon recovery can be costly. NVM can avoid these problems. This use case leverages NVM's non-volatility and larger capacity than DRAM.

Client cache. VSAN clients can benefit from a larger cache than fits in DRAM. This use case leverage's NVM's larger capacity.

Checksum storage. VSAN supports 32-byte checksums of 4 KB blocks for reliability. Unfortunately, disk storage is block aligned, making it inefficient to store the checksum. Various schemes are possible, but they cause fragmentation or increase the number of disk operations. NVM solves the problem because it operates on words. This use case leverages NVM's fine access granularity and non-volatility.

In all use cases, NVM accesses require low latency and high throughput since they affect VSAN's performance.

6 Key-value store design for VSAN

We now explain how we customize the design of METRADB for VSAN with 3D XPoint as the NVM technology. Our design employs existing techniques or variations; our goal is not to propose new mechanisms but rather to combine known mechanisms in a custom way for a specific application.

6.1 Features of METRADB

Key-value stores vary in functionality (§4). VSAN needs the following features, which we take as the requirements of METRADB:

Feature	Choice for VSAN
Keys	Fixed length
Values	Variable length, untyped
Operations	Get, Put, and Delete only
Containers	Yes
Transactions	Atomicity only, within one container

More precisely, in the VSAN use cases (§5), keys are always small binary identifiers, and values can be small values or buffers with generic lengths—hence the choice of fixed-length keys and variable-length untyped values. For the supported operations, it suffices to be able to

read, write, and delete keys. Containers are needed because different components of VSAN will use the key value store. Finally, transactions with atomicity facilitate crash recovery, but transactions need not support isolation and need not span multiple containers because each container will be accessed by one thread and each thread will access one container at a time.

6.2 Application interface

The interface to the key-value store is below.

Operation	Description
open(*name*,*flags*)	open/create container, get handle
remove(*name*)	remove container
close(*h*)	close a handle
put(*h*,*k*,*buf*,*len*)	put key-value pair
get(*h*,*k*,*buf*,*len*)	get key-value pair
delete(*h*,*k*)	delete key-value pair
commit(*h*)	commit transaction
abort(*h*)	abort transaction

Broadly, there are container operations (open, remove, close), key-value operations (put, get, delete), and transactional operations (commit, rollback). Opening a container returns a handle to that container, which is later used to put and get key-value pairs. Puts, gets, and deletes are executed in the context of a transaction, which can be later committed or aborted. A transaction is limited to operate on a single container.

6.3 Performance considerations

To achieve good performance with 3D XPoint NVM, we must be concerned about three things: cache flushes, memory fences, and NVM commits. Specifically, the CPU caches are volatile; if an application wants its writes to persist, it must flush the dirty cache lines using CLFLUSH, CLFLUSHOPT, or CLWB instructions, then issue a memory fence using SFENCE to ensure the flushes are visible, then commit data to NVM using PCOMMIT.[1] To ensure the writes are ordered before subsequent writes, the application then needs to issue an additional fence using SFENCE. The challenge is that the flush, fence, and commit instructions are expensive and must be avoided to obtain good performance.

6.4 Architecture

Figure 1 shows the architecture of METRADB. METRADB is a library that links to the application. Internally, the library is organized as low-level back-end, which is specific to NVM technology (3D XPoint), and a high-level front-end, which is independent. The back-end includes modules for logging transactions, managing segments for memory allocation, and implementing the data index as a hash table. The front-end components

[1] Asynchronous DRAM Refresh could simplify matters, but it requires a special power supply and its details are not yet available.

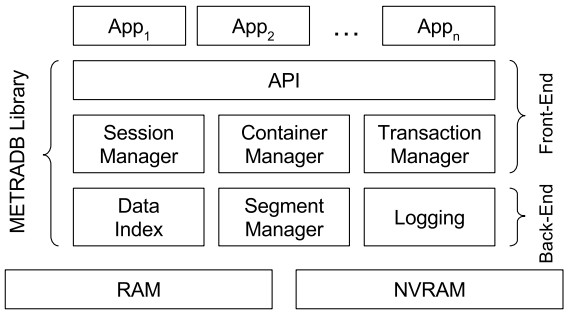


Figure 1: METRADB architecture

manage the session with the application (open handles, in-memory indexes for performance, etc), keep track of existing containers, and execute transactions.

Below, we give more details about the data structure (data index), the transaction mechanisms (transaction manager), and memory allocation (segment manager).

6.5 Data structure

We store each container in a hash table where each bucket has a head pointer to a doubly linked list. This is a simple and efficient data structure. Inserting a key-value pair requires just two writes: one write to a data buffer containing the key-value pair, one to the previous tail of the linked list. Minimizing writes is important in order to avoid expensive cache flushes. We considered using balanced trees or skip lists, but they were more complex and incurred additional writes.

Each hash table has a fixed size chosen at creation time. This is acceptable because we know the number of keys, within an order of magnitude, of the containers needed for each VSAN use case.

6.6 Transactions

We use logging to implement transactions, a well-known technique. There are two types of logs: redo or undo. Redo logs store information to reproduce the writes of a transaction, while undo logs store information to revert a transaction. Undo logs are simpler because transactions can update the underlying data as they execute. However, with NVM, it turns out that undo logs are much less efficient: they require multiple NVM commits per transaction because undo information must be committed to NVM prior to each transactional write (otherwise a power failure may leave the data modified without undo information). In contrast, a redo log need only be committed once to NVM, when the transaction commits. This is much more efficient, so we choose a redo log.

Specifically, as the transaction executes, we append its changes to the log. The log does not store the actual buffers with newly written data, but rather references to the buffers. This avoids subsequent copying overheads. When the transaction commits, we append a commit

record, commit the log to NVM, and update the hash table to point to the written buffers.

METRADB uses transactions for two purposes: to provide user-level transactions within a container, and to execute metadata operations on containers and segments (e.g., create container, create segment). There are separate logs for these: one log per container for user-level transactions, and one global log for the metadata operations. We assume there is at most one thread executing a user-level transaction per container; for metadata transactions, we support multithreading using a global lock, which suffices as these transactions need not be efficient.

6.7 Memory allocation

We keep track of memory allocations using *slabs* stored in NVM. Each slab has buffers of a given length and a bitmap indicating buffer availability. There are many slabs, to provide buffers of many different sizes. When a transaction executes and allocates a buffer, it creates a DRAM copy of the bitmap of the appropriate slab. This copy is called a *shadow bitmap*. The shadow bitmaps track the tentative allocations done by the transaction. Allocations within a container are done as part of the transaction that allocates the buffer. When the transaction commits, we copy the modified shadow bitmaps into the real slab bitmaps in NVM. Note that a transaction commit need not copy any data buffers.

6.8 Data layout

Figure 2 shows the layout of data in NVM. Broadly, the NVM is organized as a super block, a global log, containers, and segments. The super block indexes all containers and their segments. The global log stores transactional updates on the super block, and the allocation of containers and segments. Every container consists of a metadata segment and several data segments. The metadata segment has a log for the transactions within its container, and the buckets of the container's hash table (pointers to the heads of doubly linked lists). The data segments store the slabs with the actual data of the hash table (elements in the doubly linked list), with each segment holding slabs of a fixed size; a bitmap indicates the allocated slab entries.

7 Preliminary evaluation

The evaluation of METRADB has three goals: (1) assess the latency of put, get, and delete operations, (2) assess throughput scalability of these operations, and (3) identify any bottlenecks in the design or implementation.

We compare METRADB against Intel's NVML, which is a library for facilitating the programming on NVM. The library provides atomic allocations of persistent objects, concurrent transactional updates, thread synchronization, and persistent pointers. It also contains several persistent data structures, which we use as key-value stores for comparison.

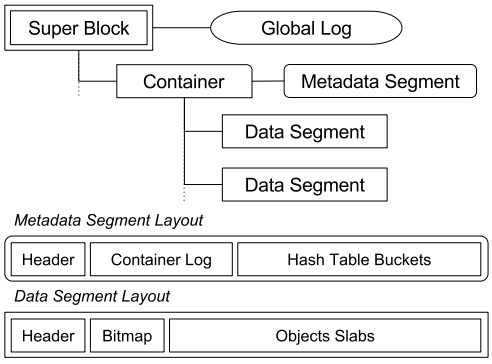


Figure 2: Data layout in METRADB

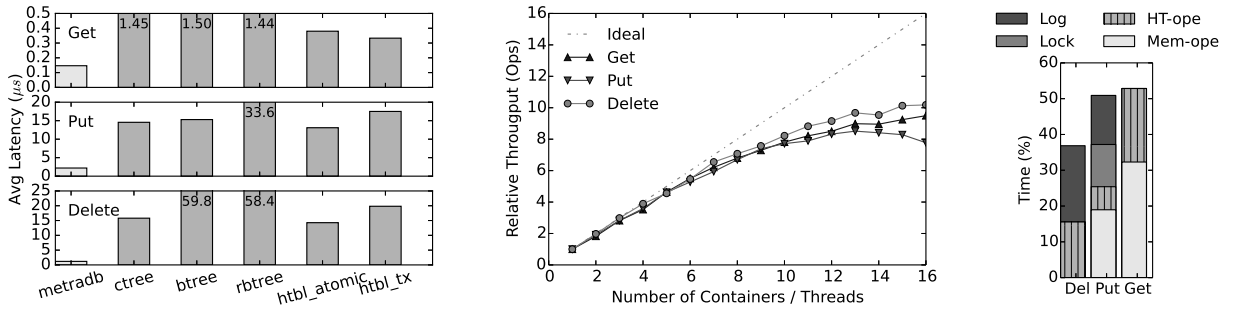
Methodology. We measure operation latency using the benchmark provided by NVML. We configure the benchmark to run lookup (get), insert (put), and delete operations on 250K key-value pairs with 1KB values. We compare METRADB against five data structures provided in NVML: radix-tree (ctree), B-tree (btree), red-black tree (rbtree), and two hash-tables (htbl_atomic and htbl_tx). These provide more features than METRADB (e.g., general transactions), since METRADB is customized to the needs of VSAN (§6); our goal is to understand the benefits of customization and the costs of generality.

We also measure throughput scalability of METRADB by running the benchmark with a variable number of threads, each accessing its own container.

To analyze METRADB's behavior, we run the benchmark under a profiler. We classify the execution time into five categories: Log (logging data), Lock (synchronization), HT-ope (hash table operations), Mem-ope (memcpy operation), and Other (remaining executing time).

Testbed. We run experiments on Linux with kernel v4.4, 24 GB of RAM, and an Intel XeonE5-2440 v2 1.90GHz CPU with 8 cores, each with 2 hyper-threads. NVM is not yet available, so instead we flush data (CLFLUSH) to a region of DRAM that is memory-mapped to a file.

Results. Figure 3a shows the latency of each system. The y-axis indicates average latency per operation in μs. The first bar is METRADB; the other bars are the various NVML data structures. On each operation, METRADB outperforms the best NVML data structure, htbl_atomic. The other NVML data structures are more powerful, but this power is not needed for VSAN, reinforcing the benefit of customization. The advantage of METRADB is greater on put and delete operations: 6.6–15x for puts and 12–50x for deletes, depending on the NVML structure. NVML is worse as it logs and commits to NVM many times per operation/transaction (and more for deletes than puts) due to the use of an undo log, while METRADB can commit the log only once per operation/transaction due to



(a) Latency of each operation in each system.

(b) Throughput scalability of METRADB

(c) METRADB execution breakdown.

Figure 3: Evaluation results.

the use of a redo log. Using a redo log is feasible due to METRADB's limited functionality: redo logs in general cause complexity and overheads because the system needs special mechanisms for transactions to see their own updates; with METRADB, however, these mechanisms are straightforward and efficient since METRADB has only two simple update operations—a benefit of customization. For get operations, METRADB is better than NVML by 2.2–10.2x. This difference is due to the use of pointers with an extra level of indirection in NVML; these pointers permit the key-value store to be shared across different address spaces, but this is not needed for VSAN—another benefit of customization.

Figure 3b depicts how METRADB's throughput scales as the number of threads increases, each thread accessing its own container. The y-axis is normalized to the throughput of one thread, and the dashed line shows ideal scalability. We can see that METRADB scales almost linearly up to 8 threads, which is the number of CPU cores. Beyond 8, scalability suffers. Profiling information indicates that the main cause is kernel-level locks that synchronize parallel updates to the memory-mapped file.

Figure 3c shows a breakdown of the execution of gets, puts, and deletes. For gets, the cost is dominated by memcpy (Mem-ope) and the hash table (HT-ope). Puts spend comparable time on memcpy, locking, and logging; locks are acquired when the operation needs to create new data segments; logging is expensive because it writes to NVM. Deletes do not incur the overheads of memcpy or locking, and therefore are dominated by the hash table and logging. Overall, we see that the inherent cost of memcpy (for gets and puts) is comparable to the overheads imposed by the store for the data structure, locking, and logging. This indicates that there are no obvious bottlenecks in any of the operations.

Acknowledgements. This paper benefited from discussions with Radu Berinde, Christos Karamanolis, Eric Knauft, and Pavel Sokolov. We also thank our shepherd Amar Phanishayee for his comments.

References

[1] BAILEY, K. A., HORNYACK, P., CEZE, L., GRIBBLE, S. D., AND LEVY, H. M. Exploring storage class memory with key value stores. In *Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads* (2013).

[2] CHAKRABARTI, D. R., BOEHM, H.-J., AND BHANDARI, K. Atlas: leveraging locks for non-volatile memory consistency. In *OOPSLA* (2014).

[3] CHEN, S., AND JIN, Q. Persistent B+-trees in non-volatile main memory. *VLDB 8*, 7 (Feb. 2015), 786–797.

[4] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS* (2011).

[5] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., BURGER, D., LEE, B., AND COETZEE, D. Better I/O through byte-addressable, persistent memory. In *SOSP* (2009).

[6] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *EUROSYS* (2014).

[7] GILES, E., DOSHI, K., AND VARMAN, P. Bridging the programming gap between persistent and volatile memory using WrAP. In *ACM International Conference on Computing Frontiers* (Oct. 2013). Article 30.

[8] GILES, E., DOSHI, K., AND VARMAN, P. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *MSST* (2015).

[9] GUERRA, J., MÁRMOL, L., CAMPELLO, D., CRESPO, C., RANGASWAMI, R., AND WEI, J. Software persistent memory. In *ATC* (2012).

[10] LU, Y., SHU, J., SUN, L., AND MUTLU, O. Loose-ordering consistency for persistent memory. In *IEEE International Conference on Computer Design* (2014).

[11] MORARU, I., ANDERSEN, D. G., KAMINSKY, M., TOLIA, N., RANGANATHAN, P., AND BINKERT, N. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *TRIOS* (Nov. 2013).

[12] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *ISCA* (2014).

[13] RUDOFF, A. NVM library. http://pmem.io.

[14] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST* (2011).

[15] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *ASPLOS* (2011).

[16] ZHAO, J., LI, S., YOON, D. H., XIE, Y., AND JOUPPI, N. P. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO* (2013).