NVMcached: An NVM-based Key-Value Cache

Xingbo Wu, Fan Ni, Li Zhang[†], Yandong Wang[†], Yufei Ren[†], Michel Hack[†], Zili Shao[‡], and Song Jiang

Wayne State University, [†]IBM T. J. Watson Research Center, [‡]The Hong Kong Polytechnic University

Abstract

As byte-addressable, high-density, and non-volatile memory (NVM) is around the corner to be equipped alongside the DRAM memory, issues on enabling the important key-value cache services, such as `memcached`, on the new storage medium must be addressed. While NVM allows data in a KV cache to survive power outage and system crash, in practice their integrity and accessibility depend on data consistency enforced during writes to NVM. Though techniques for enforcing the consistency, such as journaling, COW, or checkpointing, are available , they are often too expensive by frequently using CPU cache flushes to ensure crash consistency, leading to (much) reduced performance and excessively compromised NVM's lifetime.

In this paper we design and evaluate `NVMcached`, a KV cache for non-volatile byte-addressable memory that can significantly reduce use of flushes and minimize data loss by leveraging consistency-friendly data structures and batched space allocation and reclamation. Experiments show that `NVMcached` can improve its system throughput by up to 2.8× for write-intensive real-world workloads, compared to a non-volatile `memcached`.

1. Introduction

With recent announcement of *3D XPoint* technology [13], non-volatile memory (NVM) becomes the reality and will change how major system components are designed and built. The emerging byte-addressable, high-density NVMs also include PCM [30], STT-RAM [23], and RRAM [28]. They enable alternatives to DRAM as main memory of much higher energy-efficiency and larger capacity. When computer servers configured with NVM become commonly available, porting popular key-value (KV) caches, whose designs assume DRAM memory, onto NVM-equipped servers would allow their data to survive power outage and system crash. This transition brings significant benefit to the critical service in today's data centers. KV caches leverage large memory in a cluster of servers to temporarily store data and provide quick access to them in the form of key-value item. The data can be expensive to re-generate (e.g., after a long execution of a query in a backend database system) with a miss in the cache [19]. The cache is accessed via a simple interface, such as `GET` for reading data, `SET` for writing data,[1] and `DELETE` for removing data. KV caches have been playing a critical role in maintaining high service quality and improving user experience in many large-scale websites, such as Facebook and Amazon [1, 2]. By enabling data persistency, an NVM-based KV cache can retain its cached data and continue supplying them after a system restart without dramatically degrading service quality [35].

However, to ensure KV items cached on NVM are still usable after a system crash, writing of the KV items must be crash consistent [22]. That is, the cache must maintain data integrity in the presence of a system crash that potentially leads to incomplete writes and compromised data consistency. There have been many studies on providing crash consistency. Proposed solutions can be categorized into logging [4, 27], copy-on-write (COW) [5, 26], and checkpointing [9, 24]. However, existing solutions are too expensive for high-performance KV caches. They can incur substantial overhead on writes, which not only compromises NVM's limited write endurance but also degrades the KV cache's throughput.

Recent research on KV cache has pushed its peak throughput to many millions and theoretically 1 billion requests per second by using efficient data structures to reduce processor cache misses and enable efficient concurrency control [8, 20, 33]. The aforementioned solutions, such as logging and COW, mostly rely on CPU cache flush[2] to promptly persist data and enforce write order. However, it has been reported that frequently flushing cache lines can slow memory writes by more than 4× [21].

In this paper, we leverage a unique property of KV cache to remove all flushes except those associated with `DELETE`

APSys '16 August 4–5, 2016, Hong Kong, China

ACM 978-1-4503-4265-0/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2967360.2967374

[1] The `SET` command can perform either `INSERT` or `UPDATE` depending on presence of its corresponding key.

[2] Unless otherwise stated, 'flush' or 'cache flush' in the paper refer to 'CPU cache flush'.

and UPDATE requests. Unlike KV stores, it is acceptable for a KV cache to lose some of its cached KV items, as anyway KV items can be replaced out of the cache without notifying the users. Note that KV cache is a look-aside cache, and any lost KV items can be re-computed and re-inserted into the cache by users. Take memcached as an example, it is described as a cache for which "*clients must treat memcached as a transitory cache; they cannot assume that data stored in memcached is still there when they need it.*" [29]. For correctness we only need to make sure that no wrong values are returned for subsequent GET requests. To this end, we do not use any flushes for INSERT requests to immediately persist metadata and data or to enforce their write order. Instead, we store checksum of data along with its corresponding metadata so that the data integrity can be verified when necessary. In this way users will receive only correct data. On the other hand, for DELETE or UPDATE operations, cache flushes have to be used to remove or invalidate the corresponding KV items persistently. In this way, these items will never re-surface even after an unexpected server crash and restart.

In summary, we make three contributions in the paper:

- We identify opportunities and challenges of building an efficient byte-addressable-NVM-based key-value cache.
- We design NVMcached, an NVM-based KV cache that uses checksums to weed out corrupted data so that most cache flushes can be avoided. It also adopts a batched approach to improve efficiency of item replacement and space reclamation.
- We prototype NVMcached and extensively evaluate it with Facebook's memcached traces and Zipfian workloads generated by YCSB. NVMcached can improve the system throughput by up to $2.8\times$ for real-world workloads and up to $4.5\times$ for synthetic Zipfian workloads.

2. Design Challenges

While tolerance of data loss in a cache provides opportunity of mostly removing needs of immediately persisting data, a KV cache has another unique property that poses bigger design challenges. Once a KV cache is full, it runs its replacement algorithm to constantly identify and evict victim items to reclaim space. To minimize use of flushes, we aim not to immediately persist changes reflecting the space reclamation. However, there are three issues to address to achieve this goal.

First, a KV cache often chains its items in linked lists of a hash table. If an item in a list is being evicted, one may need to use a flush to maintain the list integrity. All of the items following the deleted one on the list can be lost after a crash if immediate flushes are not applied. Though a cache tolerates data loss, the loss should be minimized. Second, when an item's space is reclaimed and becomes available, the metadata recording free spaces for allocation, such as a free space list or bitmaps, needs to be updated. If the update is not promptly persisted with a cache flush, losing the information due to a system crash can compromise the entire allocation metadata after a restart. Such a possibility demands a complete walk over all cached items to recover the correct memory usage, making the KV caching service unavailable for an extended period of time. Third, when the cache decides to replace a KV item, it updates certain metadata, such as pointer or status bit, to reflect this replacement. If flush is performed for each replacement, the number of flushes can be as large as number of items inserted into the cache (after the cache is full). However, if these updates are not immediately persisted, there is a possibility that correctness of the system is violated. Let's consider this scenario: when the status about a replaced KV item is still cached in the processor cache, a deletion request about this item is processed according to in-cache information indicating that the item is no longer present in the KV cache. Consequently it is deemed that further actions for the deletion request, such as persistently invalidating the in-memory key, is not necessary for the request to be acknowledged. If the system crashes at this moment, the item that supposedly has been deleted is still in the non-volatile memory and will re-surface after a restart. A user deletes an item from the cache usually because it becomes invalid and should not be used. Making the deleted item available again without user's consent is an unacceptable mistake.

3. Design of NVMcached

The primary performance goal of the design is to eliminate all flushes except ones necessitated by DELETE/UPDATE requests. The secondary goal is to maximize number of KV items surviving a system crash.

To achieve the goals, there are three design principles to follow. First, NVMcached should not use (long) linked lists, because without using immediate flushes a system crash may break multiple lists and make many KV items on the lists unreachable (lost). Second, space must be reclaimed in a temporally and spatially batched fashion. That is, multiple physically contiguous KV items should be replaced in batch so that cost of necessary flushes for recording the replacement can be amortized. Third, memory allocation must also be in a batched fashion for a quick recovery of memory allocation metadata even without immediately using flushes for their persistency.

3.1 Storing Data Checksums to Avoid Ordered Writes

One potential use of long lists in a KV cache is to chain KV items on collision at a hash table entry. One example is in memcached, where each KV item contains a pointer to the next one on a list. To avoid use of lists, we separate storage and indexing of items into two data structures. As shown in Figure 1, a list of chained KV items from a hash table entry are replaced with an array of KV-item descriptors.

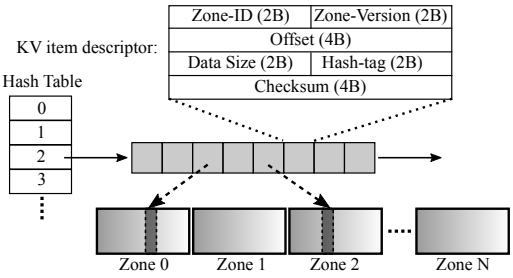


Figure 1: On-NVM data structure of NVMcached.

Each descriptor includes a pointer[3] to actual storage of the corresponding KV item. The array is pre-allocated with a limited number of slots (eight by default). A slot can be empty when the array has not yet been filled or the KV item associated with it has been deleted. In rare cases where a new slot is demanded when all slots in an array are filled, a second array is allocated and linked to the first one. As slots have been pre-allocated according to the KV cache's capacity, the expansion is rare and its overhead is minimal.

To ensure crash consistency, most designs require a persistency order between data and its corresponding metadata be enforced. Specifically, each KV item has to be persisted on the NVM before updating its corresponding pointer to prevent wrong data from being reached. While an intuitive method of enforcing the desired write order is to use flush, it can be too expensive. Instead, we compute 32 bit-checksum of each KV item's content using the CRC32 hash function. Our experiment shows that calculating the checksum of a piece of data can be about 5× to 20× faster than performing cache flushes on the data. We store the checksum and the item's size as part of a descriptor along with the pointer (see Figure 1). When accessing KV items, those with mismatched checksums are detected so that corrupted items will not be returned. Note that it is not necessary to compute checksum and perform the comparison upon every KV-item read. This is necessary only after a crash and restart. Accordingly, the use of checksums is of very low cost.

3.2 Integrating KV item replacement and space reclamation in a Log of Zones

In a memory cache management system, data replacement and space reclamation are two fundamental operations and must be efficiently supported. The former is to identify and evict cold data, and the latter is to locate and remove garbage data, including deleted ones as well as overwritten ones if log-style writes are assumed. Both operations pose challenges to the design of NVMCached.

3.2.1 Support of KV Item Replacement

A replacement algorithm for a KV cache usually maintains a long linked list to record KV-items' access history for identifying items of weak locality for eviction. One example is the LRU list of the LRU algorithm adopted by memcached. Without using flushes, the list(s) can be broken and become unusable after a crash. To make matters even worse, the list(s) need to be updated even with GET requests. While using the CLOCK replacement algorithm can eliminate the list(s) [8], one still needs to frequently update the CLOCK data structure for each GET request. To address the issue, we move the LRU list to the DRAM to remove necessity of maintaining its crash consistency, as we assume a hybrid memory where a fraction of its space is composed of DRAM. However, by doing so all access history would be lost after a crash, and cache efficacy can be compromised accordingly. To address this issue, NVMcached lays out KV-items in the NVM to roughly reflect their access locality, which provides clues for the replacement algorithm after a crash. Section 3.3 provides detailed description.

3.2.2 Support of Space Allocation and Reclamation

A KV cache requires a memory allocation mechanism for space allocation and de-allocation. One readily available choice is off-the-shelf memory allocators such as the Glibc's malloc() and its alternatives [7, 10, 17]. However, their implementations usually involve long lists for free and allocated spaces, and demand extensive use of cache flushes and even hardware supports in NVM [21], which is inconsistent with NVMcached's design principle.

It has been shown that a log-structured approach for memory management can achieve a high memory utilization [25]. This approach manages memory space much like a log-structured file system, where space is allocated in the log order and free space is reclaimed during garbage collection. NVMcached adopts the approach. However, there are two issues to address to enable efficient persistent memory allocation. First, garbage collection is a major source of inefficiency in a log-structured system, as it needs to migrate live data while collecting garbage data. When invalidated data is of only a small fraction, the migration can be very inefficient. Second, in a log-structured allocation free spaces released by evicting unpopular KV items can be scattered across the log. They cannot be immediately reused until the garbage collection process reaches them, making re-collection of the spaces very inefficient. To address the issues, we integrate garbage collection and replacement operations in a zone, and exploit strong access locality exhibited in KV cache workloads.

3.3 Batched KV Item Replacement and Space Reclamation in Zones

In NVMcached, we organize the log for space allocation and reclamation in zones of fixed size. In a zone to be cleaned, there are two types of KV items for garbage collection. One includes those that have been deleted or invalidated by DELETE or UPDATE requests. The other includes those that have been evicted by the replacement algorithm. The replacement of KV items in a zone is performed when the zone

[3] Because KV items are allocated in segments of memory named *zone*, the pointer of an item is actually composed of its zone id and offset in the zone.

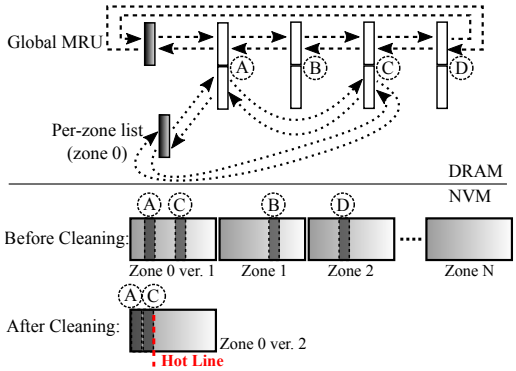


Figure 2: Data structures for zone cleaning. Each zone maintains a list tracking hot items. In the shown scenario where Zone 0 is being cleaned, hot items A and C are identified by walking through the per-zone list for Zone 0. On the NVM, A and C are moved to the front of Zone 0 demarcated by a hot line.

is being cleaned to increase efficiency of space reclamation and reduce garbage collection cost. In addition, this enables batched replacement of KV items, which amortizes the cost of a cache flush demanded by each replacement operation.

Each zone has a unique identifier (Zone-ID) and a version number (Zone-Version), as shown in Figure 1. Each KV-item descriptor contains a Zone-ID, a corresponding version number of the zone, the item's offset in the zone, and the item size. The purpose of associating a version to each zone is to quickly remove all items in a zone from being accessed. To access an item in a zone, in addition to the checksum verification, the zone version number in its descriptor must match its counterpart recorded in the zone. A mismatched zone version suggests that the descriptor is invalid. Once a zone's version number is changed, all the KV items in it become inaccessible. Therefore, we only need to immediately persist the updated version number of the zone with one cache flush to keep all items in the zone from being accessed. This design significantly reduces cost of flushes usually required for each KV item's replacement. After incrementing a zone's version number, items in the zone that have been deleted/invalidated or should be replaced are abandoned. Other live items (of strong locality) in the zone are revitalized by incrementing the version number in their descriptor.

To determine the locality strength of an item, we maintain a global LRU list in the DRAM to record access history for the `GET` requests. To provide more replacement candidates to reduce live migrations in the zone cleaning, the list is actually a most-recently-used (MRU) list, which tracks only hot items to prevent them from being discarded.

Figure 2 illustrates essential data structures related to the zone cleaning operation. When a zone is to be cleaned, only the items recorded in the MRU list are immune to replacement. All other live items in a zone will be discarded along with deleted and invalidated items in the zone. To reduce NVM writes, we maintain the list in the DRAM. However, after a crash access history recorded in the list will be lost. To address the issue, we lay out items in a zone in such a way that access history can be approximately reflected even without the MRU list. To this end, during a cleaning process we move hot items to the beginning of the zone, and revitalize them by populating their descriptors with the zone's new version number. The offset at the end of the area for hot items is named *hot line*. With a hot line recorded in each zone, `NVMcached` knows which items should not be replaced right after a recovery from a crash, and the space after the hot line can be reclaimed and ready for new allocations.

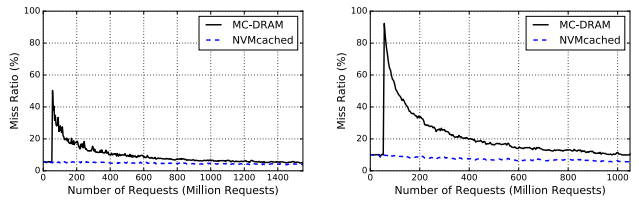
A major concern on the log-structured space allocation is about its potentially high cleaning cost, or specifically the cost of moving live data out of the area being cleaned. `NVMcached` has effectively addressed the issue by taking advantage of a property of KV cache workloads, which is that access of KV items is highly skewed. That is, a small percentage of items receives most of the accesses. This is reported in the study of `memcached`'s workload [2] and widely assumed in studies of KV caches [8, 20, 31, 32]. This property supports `NVMcached`'s choice of using MRU items, instead of pure LRU items, for replacement. More importantly, it implies that set of the hot items before the hot line, or named *hot set*, is highly likely to be stable. When `NVMcached` starts to clean a zone, it first identifies current hot set comprising this zone's items in the MRU list, and then it is compared with the last hot set comprising items before the hot line. If most of the items in the last hot set (80% by default) still remain in the new hot set, then items in the last hot set will not be relocated and the rest of the new hot items will be relocated right after the old hot line. In this way, most data movement for zone cleaning can be avoided.

3.4 Caching at DRAM

In the design of `NVMcached`, we assume a hybrid memory with a fraction of it being DRAM. Write latency and energy consumption of NVM can be much worse than those of DRAM [16, 18, 30]. Accordingly, `NVMcached` uses DRAM as a write cache to keep short-lived items from entering the NVM, including ones that can quickly be deleted or updated. A key to effective use of the cache is to identify frequently updated keys, which are placed in the cache to reduce writes to NVM. To this end, we maintain another MRU list to track locality of recently written items. We use a small MRU list of 1024 entries. Instead of caching every write, we only cache an item if it shows a high hit count (at least three) in the MRU list. In this way, only the hot items are kept in the write cache and data movement between write cache and NVM can be reduced.

4. Evaluation

We evaluate efficacy of `NVMcached` by answering three questions. How does a persistent KV cache retain its hit ratio



(a) APP (b) USR

(c) ETC (d) SYS

Figure 3: Miss ratios before and after a system's crash and restart.

after a crash-restart? Is it affordable to apply conventional approaches, such as using flushes, to provide crash consistency? Does NVMcached retain high performance while keeping most of its data persistent after a crash?

The experiments run on a server with two Xeon E5-2680 v3 processors of 30 MB L3 cache, and 256 GB (16×16 GB) DDR4 DRAM. The operating system runs 64-bit Linux 4.4.1. The hyper-threading feature is turned off from BIOS. We currently do not have large-capacity NVM devices. Instead, DRAM is used to emulate NVM. Because NVM has (much) higher write cost than DRAM, the performance advantage of NVMcached, which mainly attributes to its reduction of flushes and writes, is actually under-reported with this emulation. We use CLFLUSH and MFENCE instructions to flush cache data to the NVM and maintain a correct write order.

In the evaluation, we replay the traces collected at Facebook's production memcached system [2] and Zipfian workloads generated from Yahoo's YCSB benchmark suite [6]. We compare NVMcached with three memcached-based KV caches. **MC-DRAM** is the stock memcached running on DRAM. **MC-NVM** is the memcached running on an assumed NVM. After every SET/DELETE operation, new KV item and its metadata are promptly persisted with carefully ordered cache flushes. However, the LRU operations are not persisted, and the LRU list will be discarded after a crash. **MC-NVLRU** is similar to MC-NVM except that crash consistency is also maintained for the LRU list.

4.1 Results on Miss Ratio

To examine impact of a crash-restart on cache miss ratio, we replay four of Facebook's memcached traces (APP, USR, ETC, and SYS). One of them (VAR) is not selected as it is write-dominant with only a few distinct keys touched. Each experiment has two phases. In the first phase, the cache is warmed up by replaying the trace until it produces a stable hit ratio. In the second phase, we emulate a crash and observe the variation of the miss ratio until it becomes stable.

In this experiment we compare two systems (MC-DRAM and NVMcached). While all data will be lost in volatile memcached, NVMcached can retain most of its data after a crash on an assumed NVM. As the NVM is emulated, the crash of NVMcached has to be simulated by destroying data that are likely still in the processor at the time of the crash. To be conservative, we remove a total amount of 60 MB of the most recently inserted KV items, which is two times larger than the last-level cache of the processor (30 MB). We set the capacity of the KV caches to 64 GB, the same configuration of the servers where the traces were collected.

Figure 3 shows the miss-ratios before and after a crash for MC-DRAM and NVMcached. For three of the workloads, APP, USR, and ETC, MC-DRAM shows a significant increase of miss ratio after the crash. However we do not observe 100% miss ratio, because during the time period when the hits/misses are being collected, popular items have been cached and contribute to the hit count for subsequent GET requests. After a restart, the miss ratio slowly reduces but it does not receive a complete recovery even after billions of requests have been served. In contrast, NVMcached does not show any disruption on miss ratio with the crash. The reason for its strong resilience is that the lost data is just of a tiny fraction of the total amount of data – only about 0.1% data in the cache is lost.

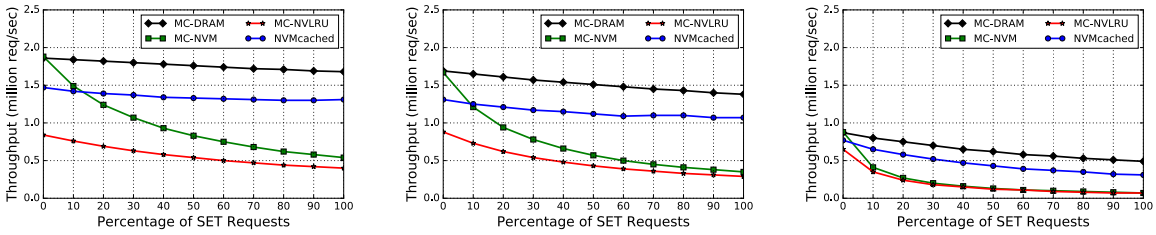
For the SYS workload, the miss ratio shows a small spike and is recovered quickly. This workload has a very small working set and over 50% of its requests are SETs, which makes it easier to recover from a crash. Again NVMcached keeps a consistently low miss ratio without showing any visible disruption.

4.2 Results on Throughput with Synthetic Workloads

In this section we compare impact of operations for crash consistency on system throughput between NVMcached, MC-DRAM, MC-NVM, and MC-NVLRU.

We use Zipfian workloads with mixed GET/SET requests of different compositions. Figure 4 shows the results with three different value sizes. MC-DRAM's throughput is the highest among all the systems as it is by nature designed for DRAM with no cost for crash consistency. MC-NVM performs well for GET-dominant workloads but the throughput deteriorates quickly as the percentage of SET requests increases. With additional cost of persisting LRU and ensuring its consistency, MC-NVLRU performs even worse and its throughput for GET-only workloads is also reduced. However, with larger value size, the cost of persisting the KV data dominates the execution time and the cost of persisting LRU history becomes less significant.

NVMcached maintains a mostly high throughput. However, the design of NVMcached also adds extra overheads. For example, the write cache adds extra lookup cost for both GET and SET requests. This is why with SET-dominant workloads NVMcached can have a lower throughout than MC-NVM. While removing the write cache can improve per-



(a) Value Size = 100 B (b) Value Size = 1000 B (c) Value Size = 10000 B

Figure 4: Throughput of four KV caches. A single thread is used for each test. The key size of each item is 10 bytes.

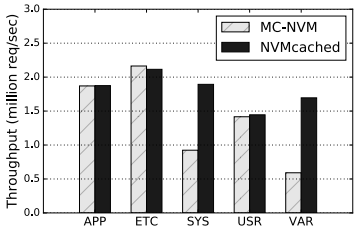


Figure 5: Throughput with real-world workloads.

	APP	ETC	SYS	USR	VAR
GET	84.05%	74.14%	46.88%	99.72%	1.78%
SET	4.64%	2.03%	53.09%	0.17%	98.22%
DEL	11.31%	23.84%	0.03%	0.11%	0.00%

Table 1: Statistics of request type for five real-world traces.

formance of GETs, frequent SET operations can add even more cost to NVM devices. As DRAM is used here to emulate the slower NVM, the write cache should be more cost-effective in a real hybrid DRAM+NVM system.

4.3 Results on Throughput of Real-world Workloads

To know how NVMcached helps with real-world KV cache systems, we replay the five Facebook KV traces on NVMcached and MC-NVM. Figure 5 shows the experiment results. While three of them–APP, ETC and USR–show almost identical throughput for NVMcached and MC-DRAM, on the other two traces they show more than 2× difference. Table 1, which lists proportion of each type of requests in the traces, helps to explain the throughput difference. Two of the traces, SYS and VAR, contain a significant percentage of SET requests. This suggests that NVMcached has effectively removed the flushes for SETs, improving the overall throughput by 2× for SYS and by 2.85× for VAR. However, for the other three GET-dominant workloads, each with a large portion of DELETE requests, NVMcached and MC-NVM show similar throughput, as in both systems cache flushes are unavoidable in processing DELETEs.

5. Related Work

In this section we discuss previous studies on using NVM and its integration in KV systems.

Persistent Memory Allocator. Mnemosyne, NV-heaps, and the NVM Library provide general-purpose programming interfaces for accessing NVM [4, 12, 27]. They need to employ expensive undo/redo logging to maintain consistency for persistent transactional updates. NVMalloc tries to optimize the persistent memory allocation by minimizing metadata writes [21]. However, expensive flushes are still required to persist each allocation and it shows a roughly 75% performance degradation compared to non-persistent malloc. In addition to the cost of allocation, user data still requires extra cost for correct ordering and persistency. Instead of using expensive allocation interface to provide consistency, NVMcached mostly removes necessity of promptly persisting data/metadata by using KV item descriptor to verify integrity of KV items after a crash-restart.

Persistent Data Structures. Write-atomic B$^+$-Tree enables single-write metadata update, which uses a single flush instead of using expensive logging/journaling for each update in the tree [3]. CDDS is a similar design that employs versioning to enable atomic metadata update in B$^+$-tree [26]. NV-Tree allows the keys in each B$^+$-Tree node to stay unsorted to enable atomic metadata update [34]. Among these works, the atomic operations (using only one cache flush) only maintain integrity of the B$^+$-tree itself, the actual data–the KV items–are not protected by the B$^+$-tree. As a result, the system has to pay additional cost for maintaining consistency and persistency. NV-Logging [11] and NVWAL [15] propose to use NVM as a temporary logging device for fast transaction processing in OLTP systems. In this scenario data will be finally committed to the database which is hosted on a slower storage such as SSD. NVMcached is a complete design of KV cache system that eliminates flushes except for the unavoidable ones for deletions and invalidations.

Hardware Supports for NVM. The commodity x86 CPUs provide essential but limited support for NVM. CLFLUSH instruction forces writing a cache line to memory and invalidates that cache line [14]. In addition, memory fences (such as MFENCE instruction) should be used to enforce order between writes. The use of CLFLUSH has significant impact on workloads with strong locality because cache-line write-backs always lead to their invalidation. A recently proposed instruction (CLWB) enables write-back of a cache line without invalidation [14]. However, even with weak locality, explicit cache write-back still introduces significant overhead.

NVMcached leverages the unique property of KV cache to remove need of flushes except for deletion and invalidation.

6. Conclusion

We introduce NVMcached, a KV cache for non-volatile memory that can significantly reduce expensive cache flushes. This is achieved by a systematical design that uses checksums to enable efficient data integrity checking and memory zones for bulk data removal while preserving access locality history after a crash. NVMcached improves system performance by up to 2.85× for real-world workloads, while preventing miss-ratio spikes after a system crash and restart.

Acknowledgments

We are grateful to the paper's shepherd, Xi Wang, and anonymous reviewers who helped to improve the paper's quality. We thank Facebook Inc. and Eitan Frachtenberg for sharing Memcached traces, which allowed one of the authors (Song Jiang) to conduct extensive experiments for the evaluation. This work was supported by US National Science Foundation under CNS 1217948 and CNS 1527076.

References

[1] Amazon. ElastiCache. http://aws.amazon.com/elasticache/.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[3] S. Chen and Q. Jin. Persistent B+-trees in Non-volatile Main Memory. *Proc. VLDB Endow.*, 8(7):786–797, Feb. 2015.

[4] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[5] 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 the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[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*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[7] J. Evans. jemalloc. http://www.canonware.com/jemalloc/.

[8] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 371–384, Berkeley, CA, USA, 2013. USENIX Association.

[9] S. Gao, B. He, and J. Xu. Real-Time In-Memory Checkpointing for Future Hybrid Memory Systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 263–272, New York, NY, USA, 2015. ACM.

[10] S. Ghemawat and P. Menage. tcmalloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[11] J. Huang, K. Schwan, and M. K. Qureshi. NVRAM-aware Logging in Transaction Systems. *Proc. VLDB Endow.*, 8(4):389–400, Dec. 2014.

[12] Intel Corporation. NVM Library. http://pmem.io/about/.

[13] Intel Corporation. 3D XPoint Unveiled - The Next Breakthrough in Memory Technology. http://goo.gl/JiilPt, 2015.

[14] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer Manuals*. Number 253665-056US. September 2015.

[15] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 385–398, New York, NY, USA, 2016. ACM.

[16] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 256–267. IEEE, 2013.

[17] D. Lea. A Memory Allocator. http://g.oswego.edu/dl/html/malloc.html.

[18] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory As a Scalable DRAM Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[19] C. Li and A. L. Cox. GD-Wheel: A Cost-aware Replacement Policy for Key-value Stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 5:1–5:15, New York, NY, USA, 2015. ACM.

[20] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 476–488, New York, NY, USA, 2015. ACM.

[21] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, pages 1:1–1:17, New York, NY, USA, 2013. ACM.

[22] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Crash Consistency. *Queue*, 13(7):20:20–20:28, July 2015.

[23] D. Ralph and M. D. Stiles. Spin transfer torques. *Journal of Magnetism and Magnetic Materials*, 320(7):1190–1216, 2008.

[24] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu. ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 672–685, New York, NY, USA, 2015. ACM.

[25] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[26] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[27] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[28] G. Wang, Y. Yang, J.-H. Lee, V. Abramova, H. Fei, G. Ruan, E. L. Thomas, and J. M. Tour. Nanoporous Silicon Oxide Memory. *Nano letters*, 14(8):4694–4699, 2014.

[29] Wikipedia. Memcached. https://en.wikipedia.org/wiki/Memcached.

[30] H. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[31] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: An LSM-tree-based Ultra-large Key-value Store for Small Data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 71–82, Berkeley, CA, USA, 2015. USENIX Association.

[32] X. Wu, L. Zhang, Y. Wang, Y. Ren, M. Hack, and S. Jiang. zExpander: a Key-Value Cache with both High Performance and Fewer Misses. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. ACM.

[33] Y. Xu, E. Frachtenberg, and S. Jiang. Building a high-performance key-value cache as an energy-efficient appliance. *Perform. Eval.*, 79:24–37, 2014.

[34] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181, Berkeley, CA, USA, 2015. USENIX Association.

[35] Y. Zhang, G. Soundararajan, M. W. Storer, L. N. Bairavasundaram, S. Subbiah, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Warming Up Storage-Level Caches with Bonfire. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 59–72, San Jose, CA, 2013. USENIX.