A Scalable and Persistent Key-Value Store Using Non-Volatile Memory

Doyoung Kim
Department of Computer Science
Yonsei University
50, Yonsei-ro, Seodaemun-gu, Seoul,
Republic of Korea
kem2182@yonsei.ac.kr

Won Gi Choi
Department of Computer Science
Yonsei University
50, Yonsei-ro, Seodaemun-gu, Seoul,
Republic of Korea
cwk1412@yonsei.ac.kr

Hanseung Sung
Department of Computer Science
Yonsei University
50, Yonsei-ro, Seodaemun-gu, Seoul,
Republic of Korea
hssung@yonsei.ac.kr

Sanghyun Park †
Department of Computer Science
Yonsei University
50, Yonsei-ro, Seodaemun-gu, Seoul,
Republic of Korea
sanghyun@yonsei.ac.kr

ABSTRACT

Non-volatile random-access memory[1] has gained recent attention because of its guaranteed data persistence and low data access latency. In-memory key-value stores generally operate by storing log files, which generate disk I/O to prevent data loss from unexpected system failure. As the performance of in-memory key-value stores is bound by disk speed, the advent of NVRAM can be a viable solution to alleviate performance degradation. However, leveraging NVRAM to store entire data is nascent in terms of the cost per capacity. We propose a novel hybrid key-value scheme that consists of NVRAM and dynamic random-access memory, which supports a higher level of data persistence while maintaining high performance. Results from our proposal scheme show outstanding results against NoSQL benchmarks in terms of performance per data persistency. In addition, our scheme provides scalability allowing NVRAM and DRAM to be used without possibility of data loss.

CCS CONCEPTS

• **Information Systems** → **Key-value stores**; Hardware → Non-volatile memory

KEYWORDS

In-Memory Key-value stores; Non-volatile memory; Data Persistence; Database Logging; Hybrid Database system

† Corresponding author

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04.
https://doi.org/10.1145/3297280.3298991

1 INTRODUCTION

In-memory key-value store [6] is a database that holds all the metadata in dynamic random-access memory (DRAM), including the key-value data. However, since DRAM is volatile, data is lost when a system is unexpectedly crashed. To prevent data loss, Redis [15], a well-known open source In-memory key-value store widely used in various enterprise solutions, provides two types of logging schemes known as Redis-database file (RDB) and append-only file (AOF). As both schemes generate disk I/O to access persistent storage, Redis suffers from critical performance issues.

Non-volatile random-access memory(NVRAM, NVM) has been considered a promising device for replacing the existing block devices, such as hard disk drives or solid-state drives. Several NVRAM devices [3, 12] have been actively researched. As NVRAM provides high performance, that is comparable to DRAM, it can be used to alleviate the performance gap between DRAM and slower block devices. Although In-memory key-value stores leverage DRAM to achieve high throughput for client requests, the stores maintain information in a persistent device to facilitate recovery from an unexpected system crash, which bounds the entire performance of the system. As NVRAM preserves data even if failure occurs, while maintaining higher performance than block devices, studies that utilize NVRAM in memory stores have emerged in recent years [5, 9, 18, 22].

Intel implemented an NVRAM-aware Redis to mitigate the performance degradation using persistent development kit (PMDK) APIs to develop NVRAM-related architecture. However, designating NVRAM as the main Redis storage is premature because NVRAM devices are not commercialized and do not have the capacity required to be cost effective [21].

We propose a novel data management scheme to utilize a hybrid environment consisting of NVRAM and DRAM. To achieve this, we designed a hybrid architecture with a relatively limited cost

per capacity of NVRAM. In addition, boosting the utilization of DRAM is our primary contribution while also increasing data persistence over that of the conventional persistence model of Redis.

2 BACKGROUND

2.1 Redis Persistence Methods

Redis provides two persistence methods for preserving data from unexpected system crashes: RDB and AOF [16]. RDB is a snapshot file in binary format consisting of the Redis contents. Redis periodically saves this file at a user-defined time. While RDB provides fast recovery times, Redis cannot guarantee data persistence between successive generations of RDB.

On the contrary, AOF is a file tracing the commands that modify the contents of Redis. The commands are kept in the AOF buffer in memory, and the contents of the AOF buffer are flushed to a persistent disk through an fsync call. AOF provides a higher level of data persistence than RDB. However, depending on the fsync invocation policy, Redis cannot guarantee persistence across the entire set of data, or its performance can be poor. For example, the "always" policy, which invokes fsync whenever a command is requested, heavily degrades the throughput of Redis. The "everysec" policy uses a background thread to invoke an fsync call every second and cannot guarantee all commands between fsync invocation.

AOF continuously writes the command requested to Redis, so its file size becomes infinitely large. To prevent this problem, Redis performs an AOF-rewrite operation. If the size of an AOF file becomes larger than a certain size, Redis temporarily delays incoming commands and writes stored data to a new AOF file as a SET command log. After writing a new AOF file, Redis replaces the old AOF file with the new file. Since the commands are delayed while the AOF rewrite is performed, AOF rewrite is a major factor that temporarily degrades the performance of Redis.

2.2 PMDK/Redis

Intel implemented NVRAM-aware Redis with the PMDK API [14], which supports transactional updates to NVRAM and preserves data persistence between the CPU and NVRAM. NVRAM-aware Redis stores Redis keys and values in a memory pool, which is allocated in NVRAM as a mapped file. As the PMDK API requires additional latency to ensure data persistence, NVRAM-aware Redis stores internal structures like hashes in DRAM. The internal structure can be reconstructed by referencing a list that keeps the addresses of the keys and values in NVRAM when the system restarts. NVRAM-aware Redis guarantees data persistence generated from all modifications compared with Redis, the performance against persistence is remarkable. However, as NVRAM has not been commercialized, designating NVRAM as primary storage is immature in terms of cost per capacity.

3 METHODS

We propose a new database system to maintain performance against persistency and to overcome the limited cost per capacity of NVRAM. The proposed model uses NVRAM to lower the AOF log dependency so that frequent rewrites do not occur. We describe our proposed method for building an NVRAM-DRAM eviction system. Because the cost per capacity of NVRAM is limited, we extend the total storage capacity using DRAM by evicting some victims from NVRAM to DRAM. An outline of the proposed structure is shown in Fig. 1.

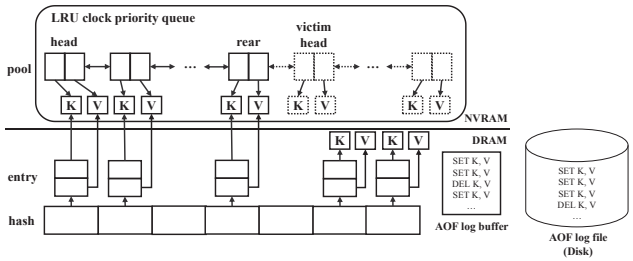


Figure 1: Outline of the NVRAM-DRAM hybrid database system.

First, NVRAM was set to tier 1 storage and data were stored in NVRAM. When the capacity of NVRAM exceeded the set point, the system selects victims by the least-recently-used (LRU) policy. These victims are evicted to tier 2 storage, DRAM with AOF log writing. To improve utilization of DRAM space, a hash table built in the DRAM manages both data in NVRAM and DRAM. If the system experiences failure, a recovery handler will reconstruct data from the AOF log file first, and data is subsequently reconstructed from NVRAM. When the system reconstructs data from NVRAM, if data are already present in the database, that was already been reconstructed from AOF, then it is rewritten to data from NVRAM because the data in NVRAM are up-to-date.

Fig. 2 shows an example of a situation where the AOF log buffer is flushed. Prior to flushing the AOF log buffer, the victims are kept in the NVRAM list to prevent data loss during flushing. After the AOF log buffer is flushed, victims are removed from the NVRAM list.

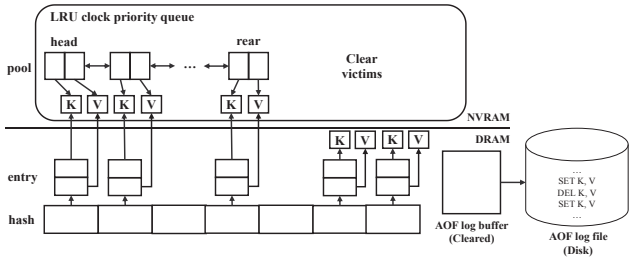


Figure 2: AOF log buffer flushing.

3.2 Eviction

Victims are chosen to be sent to DRAM from NVRAM by an eviction policy. We used the LRU policy for our eviction policy. This is because, we can greatly reduce the total number of evictions when updates occur frequently if we perform eviction with LRU. If a victim that was evicted by DRAM is updated, it will be returned to NVRAM because NVRAM is the higher-level storage. Then, if we choose the LRU policy that rarely updates victims, victims will not return to NVRAM because selected victims have low update probability. Therefore, in the case that victim selection is updated frequently, the LRU policy could be used as a proper eviction policy.

3.3 Reconstruction

Our recovery system reconstructs a hash table and recovers data that was stored in DRAM. In addition, the system remaps the data address stored in NVRAM to the hash table. During step 1, evicted entries are restored in the DRAM while the AOF log file is recovered. During step 2, victims that were not erased from the NVRAM list are restored to the DRAM. Because AOF logs of victims are not written yet, the system again attempts to write the AOF logs of victims to the disk. After the AOF log buffer is flushed to the file, victims in the NVRAM victim list are cleared. Finally, recovery completes during step 3 while reconstructing K-V data from the NVRAM list. Database persistence is fully maintained because victims must be stored in either the AOF file or the NVRAM list.

4 EVALUATION

We implemented the NVRAM-DRAM hybrid system in Redis (NDHedis) and evaluated it with the following device specifications. The system consists of quad-core 4.0 GHz processors with 64 GB of DRAM. The AOF log file is stored on a 3 TB of HDD. 8 GB of NVRAM in DRAM was built using the NVRAM emulator [4] provided by PMDK.

We set eviction to trigger when the NVRAM capacity exceeds 10 MB. This is because it is difficult to measure the performance of NDHedis unless we attempt to force eviction.

4.1 NVRAM usage evaluation

In this test, we measured the usage of NVRAM for PMDK/Redis and NDHedis. We used the Redis-benchmark[17] and stored non-redundant key-value data. We set the size of the data to 32 Bytes.

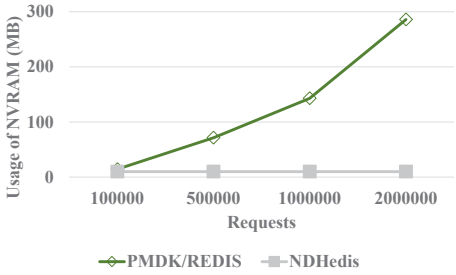


Figure 3: NVRAM usage evaluation results.

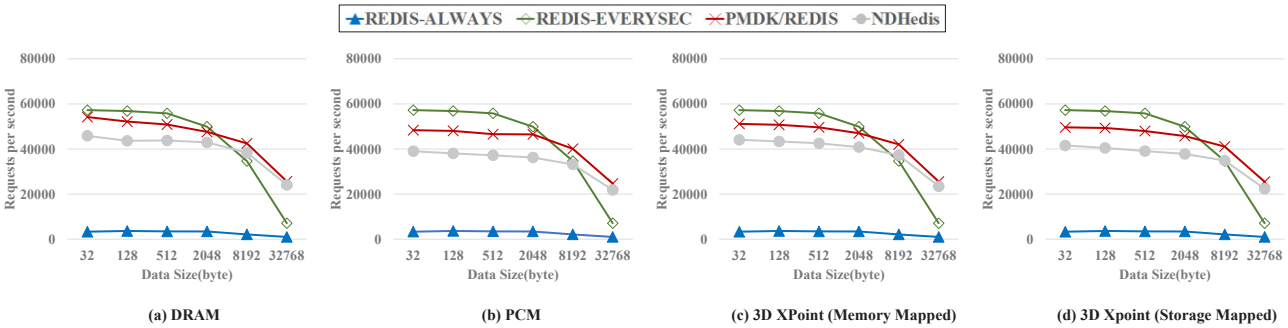
Fig. 3 shows the NVRAM utilization results used by PMDK/Redis and NDHedis. In NDHedis, when the capacity of the NVRAM increases by more than a certain amount, the data is evicted to the DRAM. Therefore, the capacity of provided by NVRAM and used by NDHedis is much smaller and constant than PMDK/Redis. NDHedis showed the same NVRAM capacity as PMDK-Redis when storing 100,000 data entries, but it is a factor 0.06 smaller than the NVRAM capacity of PMDK/Redis when storing 2,000,000 data entries.

4.2 NVRAM latency evaluation

	Read (ns)	Write (ns)
DRAM	50	50
PCM	50	500
3D XPoint (Memory Mapped)	100	100
3D XPoint (Storage Mapped)	200	200

Table 1: Latency of NVRAM devices.

NVRAM has a different latency for each type[3, 12]. Table 1 shows the approximate latency of various types of DRAM and



(a) DRAM (b) PCM (c) 3D XPoint (Memory Mapped) (d) 3D Xpoint (Storage Mapped)

Figure 4: SET performance graph measured for each emulated NVRAM device.

NVRAM. 3D XPoint (memory mapped) has great performance among the various types of NVRAM. However, the memory mapped method in 3D XPoint does not completely guarantee non-volatility of stored data.

In this experiment, we compared performance changes of NDHedis and PMDK/Redis according to the type of NVRAM. We used the Memtier-benchmark[11] and stored key-value data ranged from 0 to 1,000,000, leading to infrequent updates. Since the NVRAM emulator provided by PMDK does not provide a latency option, we implement a wrapper function on the PMDK interface to set the latency virtually. A virtually implemented latency option is not exactly the same as the latency of an actual NVRAM device. However, the performance of NDHedis and PMDK/Redis can be approximated by a virtualized latency option. Fig. 4 shows the SET performance of several emulated NVRAM devices. Redis-everysec and Redis-always are not affected by the type of NVRAM emulation. However, PMDK/Redis and NDHedis exhibit different performance depending on the type of NVRAM emulation. Among the NVRAMs used in the experiment, 3D XPoint (memory mapped) showed the best SET processing performance, but data persistence cannot be guaranteed. 3D XPoint (storage mapped) is slower than 3D XPoint (memory mapped), but it has better performance than PCM. Among all the types of NVRAM, NDHedis exhibits better performance than Redis-everysec when the data size exceeds 8192 bytes. This is because AOF-rewrite operations occur frequently in Redis-everysec as the data size increases, whereas NDHedis performs AOF-rewrite only once when the NVRAM capacity exceeds a certain eviction point.

5 CONCLUSIONS

We designed and implemented a hybrid system that compensates the capacity limit of NVRAM and the volatile limit of DRAM. Compared with Redis, which has fast performance but is partially persistent, we guarantee data persistence using NVRAM. We also store data in the NVRAM first, thereby reducing the occurrence of AOF-rewrite so that no performance degradation occurs. In addition, compared with PMDK/Redis, which is persistent and fast but has limited capacity, we overcome the limitation of cost per capacity by using DRAM to extend the entire capacity. Our proposal provides the same persistence level as PMDK/Redis by supporting persistence during an eviction.

In a further study, we will apply other eviction algorithms to the NVRAM eviction policy. Since maintaining a priority queue is costly, we will design the structure of the list built into NVRAM with a different structure, such as a tree or heap. AOF-rewrite can still occur if the amount of stored data is very large. We will improve the performance by re-designing the AOF-rewrite operation using NVRAM.

6 ACKNOWLEDGEMENTS

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the SW Starlab support program (IITP-2017-0-00477) supervised by the IITP (Institute for Information & communications Technology Promotion).

REFERENCES

[1] J. Arulraj, and A. Paylo. 2017. How to Build a Non-Volatile Memory Database Management System. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA 2017), ACM, (2017), 1753–1758.

[2] J. Arulraj, M. Perron, and A. Pavlo. 2016. Write-behind Logging. *Proc. VLDB Endow.*, 10 (Nov. 2016), 337–348.

[3] P. Cappelletti. 2015. Non volatile memory evolution and revolution. In *2015 IEEE International Electron Devices Meeting (IEDM)* (Dec. 2015), 10.1.1–10.1.4.

[4] S.R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA 2014), ACM, 15:1–15:15.

[5] R. Fang, H. Hsiao, B. He, C. Mohan, and Y. Wang. 2011. High performance database logging using storage class memory. In *2011 IEEE 27th International Conference on Data Engineering* (Apr. 2011), 1221–1231.

[6] J. Han, H. E, G. Le, and J. Du. 2011. Survey on NoSQL database. In *2011 6th International Conference on Pervasive Computing and Applications* (Oct. 2011), 363–366.

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

[8] W.H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. *SIGOPS Oper. Syst. Rev.*, 50 (Mar. 2016), 385–398.

[9] S.K. Lee, K.H. Lim, H. Song, B. Nam, and S.H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (CA, Santa 2017), USENIX Association, 257–270.

[10] L. Lersch, I. Oukid, W. Lehner, and I. Schreter. 2017. An Analysis of LSM Caching in NVRAM. In *Proceedings of the 13th International Workshop on Data Management on New Hardware* (New York, NY, USA 2017), ACM, 9:1–9:5.

[11] Memtier-benchmark, https://github.com/Redis-Labs/memtier_benchmark

[12] S. Mittal, and J. S. Vetter. 2016. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Transactions on Parallel and Distributed Systems*, 27 (May 2016), 1537–1550.

[13] Persistent Memory Development Kit, https://github.com/pmem/pmdk

[14] PMDK Implementation Redis, https://github.com/pmem/redis

[15] Redis, https://redis.io/

[16] Redis Persistence, https://redis.io/topics/persistence

[17] Redis-benchmark, https://redis.io/topics/benchmarks

[18] Y. Son, H. Kang, H.Y. Yeom, and H. Han. 2017. A Log-structured Buffer for Database Systems Using Non-volatile Memory. In *Proceedings of the Symposium on Applied Computing* (New York, NY, USA 2017), ACM, 880–886.

[19] T. Wang, and R. Johnson. 2014. Scalable Logging Through Emerging Non-volatile Memory. *Proc. VLDB Endow.*, 7 (June 2014), 865–876.

[20] F. Xia, D. Jiang, J. Xiong, and N. Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (CA, Santa 2017), USENIX Association, 349–362.

[21] H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 27 (July 2015), 1920-1948.

[22] P. Zuo, and Y. Hua. 2018. A Write-Friendly and Cache-Optimized Hashing Scheme for Non-Volatile Memory Systems. *IEEE Transactions on Parallel and Distributed Systems*, 29 (May 2018), 985–998.