HaCache: A Hybrid Adaptive Cache for Persistent Memory Based Key-Value Systems

Lixiao Cui, Gang Wang, Yusen Li, and Xiaoguang Liu(✉)

TKLNDST, College of Computer Science, Nankai University, Tianjin, China
{cuilx,wgzwp,liyusen,liuxg}@nbjl.nankai.edu.cn

Abstract. Many previous studies have proposed high-performance key-value systems based on Persistent memory (PM). However, these work ignores the fact that the read performance of PM is also lower than DRAM. In this paper, we propose HaCache, a well-designed hybrid cache for PM-based KV systems to improve read performance. HaCache combines key-value (KV) cache, key-pointer (KP) cache, and Block cache to retain the advantages of the three types of cache schemes. It can adaptively adjust the partition of cache space among the three cache schemes to adapt to workload changing. The evaluation results show that HaCache outperforms pure KV cache, pure KP cache, and pure Block cache by 2.7x, 2x, and 18% respectively.

Keywords: Cache · Key-value system · Persistent memory

1 Introduction

The emerging byte-addressable persistent memory (PM) [4] provides opportunities to design high-performance persistent key-value (KV) systems. There are many previous works that aim to design persistent KV systems [6,7,9,10,12] dedicated for PM. These works mainly focus on optimizing write operations. However, according to recent analyses of Intel Optane PM [4], the read performance of PM is only 1/3 of DRAM [13], which implies that optimizing the read performance of PM-based KV systems is also important. Cache technology is suitable for read optimization of PM-based KV systems.

There are usually two types of read operations of PM-based KV systems, point lookup (`GET`) and range query (`SCAN`). To accelerate point lookup, we can cache *Key-Value* pairs (KV cache) or cache *Key* and *Pointers* to associated values (KP cache). Compared with caching key-pointer pairs, caching key-value pairs can improve overall read performance more significantly. However, KP cache is space-efficient because it can save more DRAM space than KV cache.

This research is supported in part by the NSF of China (No. 62141412, 61872201), the Science and Technology Development Plan of Tianjin (20JCZDJC00610, 19YFZCSF00900), the Fundamental Research Funds for the Central Universities, the Graduate Student Scientific Research Innovation Project of Tianjin (2021YJSB082).

B. Li et al. (Eds.): APWeb-WAIM 2022, LNCS 13421, pp. 180–189, 2023.
https://doi.org/10.1007/978-3-031-25158-0_14

For ordered KV systems, range query is also an important operation, we could cache *Blocks* (Block cache) to accelerate it. These three cache schemes have their advantages and disadvantages. We aim to design a cache solution able to combine the virtues of performance-efficient, space-efficient, and support for range query. In this paper, we propose **HaCache**, **H**ybrid **A**daptive **Cache**, for PM-based KV systems. HaCache is a general cache solution to optimize the read performance of PM-based KV systems (with hash index, B+tree index, and so on), which contains point cache (include KV cache and KP cache) and block cache to serve point lookups and range queries respectively.

With limited space, it is difficult to combine three caches to achieve all their advantages. So the cache partition is important. HaCache follows a top-down cache partition strategy. The high-level space partition is between point cache and block cache. It determines which kind of read operation (point lookup and range query) HaCache is more advantageous. In different scenarios, the performance requirements for these two operations may be different. Therefore, we allow users to set preferences for different read operations through certain metrics (e.g. latency) and partition cache to meet the preference. The low-level space partition is between KV cache and KP cache. In this part, we partition cache to achieve higher performance with smaller space. Since the workloads usually change with time [2], the main challenge of HaCache is how to dynamically adjust cache space to achieve the purposes under changing workloads. For the high-level partition, HaCache leverages Proportional-Integral-Derivative (PID) control algorithm to dynamically adjust the cache space partition, so HaCache can be stable around the metrics set by users. For the low-level partition, we proposed KV adaptive caching (KV-AC) algorithm motivated by Adaptive Replacement Cache (ARC) [8]. To achieve the best partition, when adjusting the capacity, we take the size, access frequency, and benefit of cache items into consideration.

We evaluate HaCache using real PM devices. Under pure `GET` workload, compare with `KV cache` and `KP cache`, HaCache improves the throughput by up to 2.7x and 2x respectively. Under mixed `GET-SCAN` workload, HaCache can satisfy user's preference precisely and outperforms `Block cache` by 18%.

2 Background and Motivation

2.1 PM-Based KV Systems

Similar to DRAM, persistent memory device is attached to the memory bus and can be accessed in byte granularity. Meanwhile, it can guarantee data persistence after the power is off. The emergence of PM provides an opportunity to make traditional in-memory KV systems [2] persistent while maintaining high performance. In the past few years, dozens of works are proposed to optimize PM oriented KV systems [6,7,9,10,12]. However, these works mainly optimize the write performance but ignore the fact that the read performance of PM is slow than DRAM. We measure the read latency of two PM-based KV systems (CCEH [9] and FAST&FAIR [7]). For `GET` operation, the latency of KV systems in PM is about 3x of DRAM alternatives. For `SCAN` operation, the performance gap is widened to 5 times. These observations imply the necessity of optimizing the read performance of PM-based KV systems.

2.2 KV, KP and Block Cache

To figure out the impact on KV systems of different cache schemes, we evaluated KV, KP and, block cache in a B+tree (i.e. FAST&FAIR) based KV system. The results of `GET` are shown in Table 1. With a large cache, the performance of KV cache is better than KP cache because no extra PM accesses are needed. With a small cache, KP cache has a higher hit ratio than KV cache because of the smaller cache item. A very low hit ratio leads to the poor performance of KV cache, even though every KV cache hit has no further overhead. **Motivation 1: Combining KV cache and KP cache to maximum `GET` performance.**

Table 1. `GET` throughput and hit ratio under different cache sizes. The key size is 8B and the value size is 100B. We use YCSB [3] workload with 50M key-value pairs.

	Throughput (M ops/s)			Hit ratio		
Cache size	KV	KP	Block	KV	KP	Block
200MB	1.04	2.48	0.6	58.2%	92.7%	84.8%
500MB	1.33	3.75	0.63	69.1%	100%	88.5%
1GB	2.03	3.75	0.65	82.4%	100%	90.6%
2GB	7.48	3.75	0.7	100%	100%	93.1%

Table 2. The performance under mixed `GET` -`SCAN` workload. The cache size is 1GB and we allocate the space to KV cache and Block cache in different proportions.

KV/Block cache ratio	0/1	1/7	1/1	7/1	1/0
Throughput (K ops/s)	201	218	213	217	189
`GET` latency (ns)	1798	1241	889	641	541
`SCAN` latency (ns)	12210	12270	13448	13811	16640
Latency ratio (`SCAN`/`GET`)	6.79	9.88	15.12	21.54	30.75

`SCAN` is another important read operation. We measured the performance of a simple hybrid cache that contains KV cache and block cache under mixed `GET`-`SCAN` workload. The number of `SCAN` is 30% of total operations. Table 2 shows the results. With block cache, the `SCAN` latency is significantly reduced compared to using KV cache only. Another observation is the performance gap (latency ratio) between `GET` and `SCAN` changes drastically varying different KV/Block cache ratios. Different application scenarios may favor different read operations. For example, users could require the `GET` to be processed as soon as possible, but relax the requirements for `SCAN`. Therefore, the respective performance of different read operations should be considered. **Motivation 2: Balancing the performance of `GET` and `SCAN` when handling mixed `GET`-`SCAN` workloads.**

3 Related Work

Optimizing Read for PM-Based KV Systems. The representative work of using cache is Bullet [6]. It maintains a KV cache in DRAM to accelerate `GET`. Some studies store partial of indexes into DRAM to reduce the overhead of lookup. FPtree [10] stores the leaf nodes of B+tree into PM while placing the internals into DRAM. HiKV [12] uses a hybrid index composed of hash table and B+tree. The hash table is placed at PM and the B+tree is placed at DRAM.

Hybrid Cache. SwapKV [5] is a hotness aware KV caches that places the metadata and hot data to DRAM and stores cold data to PM. Cassandra [1] is an LSM-tree-based system. It uses KP cache and KV cache to replace the block cache. The users can configure the size of different cache statically.

Adaptive Cache Algorithms. Some previous studies use multiple caching schemes and adaptively adjust the capacity of different caches. The Adaptive Replacement Cache (ARC) [8] is the most representative. ARC is designed for page cache. It partitions the cache into a frequency cache and a recency cache. A page that is first accessed will be placed into the recency cache. If the pages in the recency cache are accessed again, they are promoted into the frequency cache. ARC uses two ghost cache regions to help space partition of the recency cache and the frequency cache adaptively. The ghost recency cache and the ghost frequency cache keep the metadata of recently evicted pages from the two real caches respectively. A ghost cache hit implies that the hit item should not be evicted, therefore, the corresponding real cache should be expanded to accommodate more cache items.

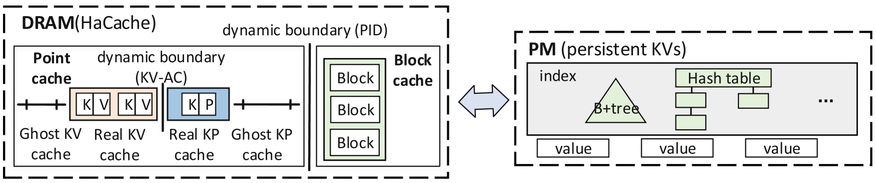


Fig. 1. Overview of HaCache.

4 Design and Implementation

Figure 1 shows the structure of HaCache. Following motivation 1, we design a KV-AC algorithm inspired by ARC, to adjust the cache capacity between KV and KP cache. The two ghost cache is used to guide the movement of the dynamic boundary. Following motivation 2, a dynamic boundary adjusted by PID algorithm is responsible for capacity partition between point and block cache.

4.1 Basic Operations of HaCache

Cache Replacement. HaCache uses CLOCK rather than LRU as the cache replacement algorithm, which uses less space (1-bit metadata for each cache item). To make the hit ratio of CLOCK close to LRU, we develop Frequency-based CLOCK for real cache. It selects several cache items (default 2 items) as the candidate evicted items at a time. The candidate with minimal access frequency is evicted.

Conversion Between KV and KP Caches. The items fetched from PM will be inserted into KP cache first. When an item in the KP cache is accessed and the corresponding CLOCK bit is active (implying that the item has been accessed recently), the KP item is converted into a KV item. Note that for the same key, only one cache item is stored in the KP cache or the KV cache. An item evicted from the KV cache is considered cold and converted to a KP item. To guide the adjustment of cache capacity, HaCache inserts the metadata of evicted items to ghost cache. For an evicted item that has been converted to a KV item, the corresponding metadata will be inserted into the ghost KV cache. Otherwise, the metadata of it will be inserted into the ghost KP cache.

Processing of Commands. When processing `GET` operation, The real KV cache and the real KP cache are searched first. If KEY (denote the requested key) does not exist in both of them, HaCache will examine the ghost KV cache and the ghost KP cache. If KEY exists in the ghost region, the corresponding real region will be expanded according to the KV-AC algorithm (Sect. 4.2). Then the persistent KV system will search KEY and return the associated value directly. Meanwhile, KEY and the associated pointer to value will be inserted into the real KP cache and the metadata of KEY will be deleted from the ghost cache. If KEY is found in real region, HaCache will check whether a KP-KV conversion should be performed. When processing `SET` operation, HaCache first writes KV pair to the PM device. If the corresponding item is found in the real cache, HaCache updates it to ensure the correctness of the subsequent `GET`. HaCache does not lookup KEY in ghost region when processing `SET`. When the `SET` operation is applied to an existing KV pair (i.e. update operation), the value is modified but the key is not. The items in the ghost region do not contain values or pointers, so there is no need to update the items in the ghost region when processing `SET`.

4.2 Adaptive Adjustment Within Point Cache

The purpose of point cache is to combine the performance efficiency of KV cache with the space efficiency of KP cache. HaCache comprehensively consider the space and performance of different cache items. HaCache uses the *adjustment factor* F to weigh the performance and space. Every cache item has its factor. The variable k denotes the access count of the cache item, b presents the benefit if it is cached and s is its size.

$$F = (\log(k) + 1)(b/s) \tag{1}$$

Among these parameters, the benefit of caching is difficult to determine. We define the benefit to be reduced latency (i.e. latency difference between DRAM cache and PM for the same item). However, for different sizes of cache items, the saved latency is different. To address these issues, we develop a *sampling method* to estimate benefits for specific KV systems. For KV cache items, we sample the average latency of acquiring KV pairs from PM and HaCache under different value sizes. Then we use piecewise linear regression (PLR) to estimate the relation between value size and latency reduction. In this way, the saved latency can be estimated for each size. We measure the latency reduction in a B+tree-based KV system is roughly linearly related to the value size. For KP cache items, the benefit is independent of value size. We measure the latency of accessing pointers from PM and HaCache respectively.

HaCache uses KV adaptive caching (KV-AC) algorithm to manage the capacity partition of point cache. It uses *adjustment factor* as the granularity of capacity adjustment. C_{point} presents the overall point cache capacity. $[KV_{real}]$ and $[KP_{real}]$ represent The maximum size of the real KV cache and KP cache. $[KV_{ghost}]$ and $[KP_{ghost}]$ represent the maximum size of the ghost KV cache and KP cache, assumed that all of the items were converted to real KV/KP items. KV-AC always maintains Eq. 2. Same with ARC, the ghost cache size in KV-AC is the difference between overall size and corresponding real cache size.

$$[KV_{real}] + [KP_{real}] = [KV_{real}] + [KV_{ghost}] = [KP_{real}] + [KP_{ghost}] = C_{point} \quad (2)$$

If the requests miss in the real cache but hit in the corresponding ghost cache, the cache capacity will be adjusted. For example, if a key is found in the ghost KV cache, the real KV cache will be expanded. Then the new size of the real KV cache should be $[KV_{real}] + nF$, where F is the *adjustment factor* and n is an adjustable coefficient. Correspondingly, the new size of the real KP cache should be $[KP_{real}] - nF$. The value of n determines the magnitude of capacity adjustment. After the new real KV cache size is determined, the subsequent requested items will be brought to KV cache without evicting until $[KV_{real}]$ reaches the new size. Meanwhile, some items will be evicted from the real KP cache until $[KP_{real}]$ is smaller than or equal to its new size. After the real cache size is updated, $[KV_{ghost}]$ and $[KP_{ghost}]$ are also adjusted according to Eq. 2. The processing is similar when a key is found in the ghost KP cache.

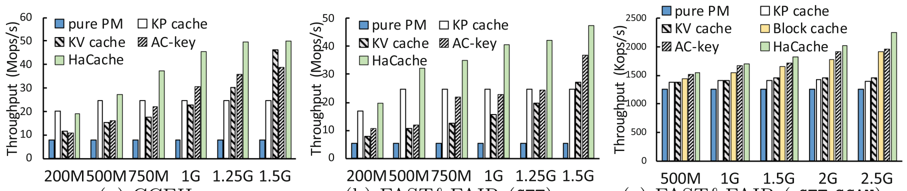
4.3 Adaptive Adjustment Between Point Cache and Block Cache

In HaCache, the capacity partition between block cache and point cache will affect the performance of `GET` and `SCAN` directly. To balance the optimization of `GET` and `SCAN`, HaCache allows users to set preferences for different read operations to adapt to different application scenarios. In our implementation, the metric of preference is the average latency ratio of `SCAN` and `GET`, that is, $Ratio = Latency_{\texttt{SCAN}}/Latency_{\texttt{GET}}$. The larger the *Ratio*, the more conducive to `GET`, otherwise it is conducive to `SCAN`. To meet an expected ratio from users, HaCache uses PID to adjust the cache partition. Eq. 3 shows the PID algorithm,

$E(T_k)$ presents the error at time T_k. PID algorithm contains proportional, integral and derivative terms. K_p, K_i, and K_d are tuning factors for the three terms respectively. When using PID, we need to set a target stable state. Then we measure the error $(E(T_k))$ between the current state and the target state. By bringing $E(T_k)$ into the equation, we get the output value $U(T_k)$ to adjust the controlled system to approach and stabilize at the target state.

$$U(T_k) = K_p E(T_k) + K_i \sum_{n=0}^{T_k} E(n) + K_d(E(T_k) - E(T_{k-1})) \tag{3}$$

We use the user's expected latency ratio ($Ratio_{setting}$) as the target stable state. We measure the average latency of `GET` and `SCAN` over a period of time (default every 100k operations) and calculate the real *Ratio*. Then E_k is calculated ($E_k = Ratio_{setting} - Ratio_{real}$). We try different values and finally decide to set K_p to 100k, K_i to 5k, and K_d to 10k. Putting the above parameters into Eq. 3, we can get $U(T_k)$. If $U(T_k)$ is positive, the real *Ratio* is usually less than the user's setting. In this case, the latency of `GET` is too large. Therefore, HaCache allocate more space for point cache and the new size of it should be $C_{point} + U(T_k)$. On the contrary, if $U(T_k)$ is negative, HaCache allocates more space for block cache and shrinks KP and KP cache. By using PID, the *Ratio* will be stable on the user's setting latency ratio.



(a) CCEH. (b) FAST&FAIR (`GET`). (c) FAST&FAIR (`GET-SCAN`).

Fig. 2. Throughput under different cache sizes (Byte).

5 Evaluation

The evaluation was conducted on an x86-64 server that has two Intel Xeon Gold 5220 processors, 128 GB of DRAM and 512GB of Intel Optane DC PM. The size of key is 8B and value is 100B. The total number of KV pairs is 50M. We use YCSB [3] to generate the skewed workloads (0.99 skewness). We conduct pure `GET` workload and mixed `GET-SCAN` workload (30% `SCAN` and 70% `GET`, the range of `SCAN` is a random number less than 100.). We select CCEH (a persistent hashing) [9] and FAST&FAIR (a persistent B+tree) [7] as the underlying KV systems. The adjustable coefficient n is set to 500. The default latency ratio between `SCAN` and `GET` is set to 15:1. We compare HaCache with `pure PM`, `KV cache`, KP cache, and `Block cache`. We also compare `AC-key` [11], which uses a hierarchical ARC algorithm to adjust capacity among KV, KP, and block cache.

5.1 Impact of Cache Size

Figure 2 (a) and (b) show the `GET` throughput under 8 threads. We first compare HaCache and `KP cache`. When the cache size is small, HaCache tends to allocate most of the cache size to store KP items to guarantee a high hit ratio. When cache size is large, the effectiveness of caching KV items becomes obvious. In CCEH, HaCache outperforms `KP cache` by 1.1x to 2x. We then turn to the comparison between HaCache and `KV cache`. When the cache size is large, HaCache tends to store more KV items because it can bring more benefits. When cache size becomes small, HaCache can maintain a high hit ratio by adjusting the partition between KV items and KP items. In FAST&FAIR, the throughput of HaCache is 2.1x to 3x of `KV cache`. HaCache also outperforms `AC-key` significantly. This mainly because HaCache takes the access frequency into account to decide capacity partition and cache replacement, which can keep frequently accessed items in cache to improve the overall performance. Figure 2(c) shows the throughput of mixed `GET-SCAN` workload. Under different cache sizes, HaCache outperforms `KV cache` by 11% $\sim$ 54% and `KP cache` by 12% $\sim$ 62%. HaCache perform better than `Block cache` because it leverages point cache to serve `GET`. The throughput gap between HaCache and `Block cache` is 7% $\sim$ 18%. Meanwhile, HaCache can control the real latency ratio close to the setting ratio. The measured real latency ratio is 15.13:1.

5.2 Adaptive Adjustment

We conduct two changed workload to verify the effectiveness of HaCache's adaptive adjustment strategy. The first is a two-phase skewness-changed workload on CCEH-based HaCache. We first execute the workload with 0.99 skewness, then change the skewness to 0.88. The total cache size is 1GB. We plot the changes of the KV cache size and the KP cache size in Fig. 3(a). When the skewness is 0.99, most of the accesses are concentrated on a small amount of KV pairs. Therefore, HaCache keeps the frequently accessed KV items to get high performance. When the skewness decreases (100Mops in Fig. 3(a)), HaCache allocates more capacity to keep KP items (KP items are space-efficient) to maintain the high hit ratio. When the capacity partition becomes stable, the throughput of HaCache achieves 28.95 Mops/s, which is almost the same as the throughput of the workload that initially uses 0.88 skewness.

The second evaluation is a `GET-SCAN` ratio changed workload. We conduct it on FAST&FAIR-based HaCache. In the beginning, the number of `SCAN` is 30% of total operations. Then we turn to conduct a pure `GET` workload. As shown in Fig. 3(b), after workload changing, the block cache shrinks quickly. When the cache partition becomes stable (after 150Mops), the cache capacity distribution is almost the same as it that initially conducts pure `GET` workload. The above results confirm that HaCache could adapt to the workload's changes.

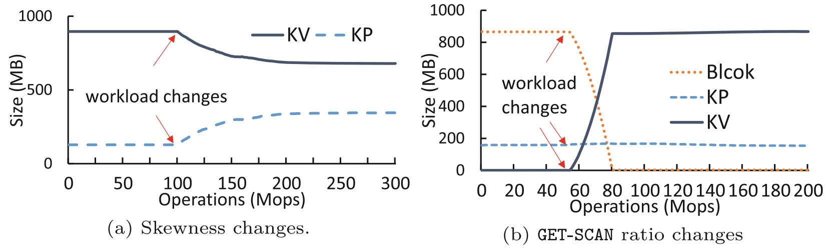


Fig. 3. Adaptive adjustment when workload changes.

6 Conclusion

In this paper, we propose HaCache, which incorporates key-value, key-pointer, and block cache schemes, to optimize the read performance for PM-based KV systems. HaCache employs PID algorithm and KV-AC algorithm to dynamically adjust the capacity allocation among KV, KP, and block parts. Our HaCache outperforms other single cache schemes in different read operations.

References

1. Apache: Cassandra. http://cassandra.apache.org/
2. Chen, J., et al.: HotRing: a hotspot-aware in-memory key-value store. In: USENIX Conference on File and Storage Technologies (FAST 2020), pp. 239–252 (2020)
3. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154. SoCC 2010 (2010)
4. Corporation, I.: Intel(R) Optane(TM) DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html
5. Cui, L., et al.: SwapKV: a hotness aware in-memory key-value store for hybrid memory systems. IEEE Trans. Knowl. Data Eng. **35**(1), 1–1 (2021)
6. Huang, Y., Pavlovic, M., Marathe, V.J., Seltzer, M., Harris, T., Byan, S.: Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In: USENIX Annual Technical Conference (USENIX ATC 2018), pp. 967–979 (2018)
7. Hwang, D., Kim, W.H., Won, Y., Nam, B.: Endurable transient inconsistency in byte-addressable persistent B+-tree. In: USENIX Conference on File and Storage Technologies (FAST 2018), p. 187 (2018)
8. Megiddo, N., Modha, D.S.: ARC: a self-tuning, low overhead replacement cache. In: USENIX Conference on File and Storage Technologies (FAST 2003), p. 9 (2003)
9. Nam, M., Cha, H., ri Choi, Y., Noh, S.H., Nam, B.: Write-optimized dynamic hashing for persistent memory. In: USENIX Conference on File and Storage Technologies (FAST 2019), pp. 31–44 (2019)
10. Oukid, I., Lasperas, J., Nica, A., Willhalm, T., Lehner, W.: FPTree: a hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In: Proceedings of the 2016 International Conference on Management of Data (SIGMOD 2016), p. 371–386 (2016)

11. Wu, F., Yang, M.H., Zhang, B., Du, D.H.: AC-Key: adaptive caching for LSM-based key-value stores. In: USENIX Annual Technical Conference (USENIX ATC 2020), pp. 603–615 (2020)
12. Xia, F., Jiang, D., Xiong, J., Sun, N.: HiKV: a hybrid index key-value store for DRAM-NVM memory systems. In: USENIX Annual Technical Conference (USENIX ATC 2017), pp. 349–362 (2017)
13. Yang, J., Kim, J., Hoseinzadeh, M., Izraelevitz, J., Swanson, S.: An empirical guide to the behavior and use of scalable persistent memory. In: USENIX Conference on File and Storage Technologies (FAST 2020), pp. 169–182 (2020)