EKRM: Efficient Key-Value Retrieval Method to Reduce Data Lookup Overhead for Redis

Yiming Yao[1(✉)], Xiaolin Wang[1], Diyu Zhou[1,2], Liujia Li[1], Jianyu Wu[1], Liren Zhu[1], Zhenlin Wang[3], and Yingwei Luo[1]

[1] National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University, Beijing 100871, China
`{yim,liujia_li}@stu.pku.edu.cn`, `{wxl,jywu2017,zhuliren,lyw}@pku.edu.cn`, `diyu.zhou@epfl.ch`

[2] The Swiss Federal Institute of Technology in Lausanne, Lausanne, Switzerland

[3] Michigan Technological University, Houghton, USA
`zlwang@mtu.edu`

Abstract. As an open-source key-value system, Redis has been widely used in internet service stations. A key-value lookup in Redis usually involves several chained memory accesses, and the address translation overhead can significantly increase the lookup latency. This paper introduces a new software-based approach that can reduce chained memory accesses and total address translation overhead of lookup requests by placing key-value entries in a specially managed memory space organized as huge pages with a fast hash table and enabling a fast lookup approach with simple hash functions, while keeping the integrity of Redis data structure. The new approach brings up to 1.38× average speedup for the key-value retrieval process, and significantly reduces misses in TLB and last-level cache. It outperforms SLB, an address caching software approach and has match the performance to STLT, a software-hardware co-designed address-centric design.

Keywords: Key-Value store · Redis · Hash function · Translation lookaside buffer · Address translation · Chained memory access · Huge page

1 Introduction

The volume of data has experienced explosive growth in the past decade, and various applications have increasing demand for accessing large amounts of data with low latency. To address this challenge, in-memory key-value systems such as Redis [28], Memcached [29] have become an important layer in the client/server architecture.

Cloud services often rely on key-value store as a key infrastructure, because it enables a simple and flexible data model with high performance and scalability

[12–15]. Cloud service providers use Redis [30], Memcached [29], and their variants for various purposes in the production environments. For key-value store systems, the data are not only diverse in types and characteristics but also mainly stored in memory, which allows for faster access and lower latency than disk-based storage structures. These features, combined with high access volume, pose many different challenges for the design of key-value systems. As the most critical performance factor for key-value storage, maximizing the data retrieval speed is still the most important goal. The hash table is a critical part of typical key-value systems. To ensure the security and integrity of the data, the hash function must have a certain complexity to resist malicious attacks. Hash tables often use linked lists to solve collisions and have the problem of poor locality. Under a large number of key-value accesses, the hash table accesses can become an important performance bottleneck.

In typical key-value store applications, accessing a record in memory involves certain kinds of index and data structure traversal, which is a process composed of multiple address translations. Before actually accessing the record associated with a key, the hash function needs to convert the key to an integer, then the index data structure needs to convert the integer to the virtual address of the target record, and then *Memory Management Unit*(MMU) tries to convert the virtual address to the physical address of the record through *Translation Lookaside Buffer* (TLB). A TLB miss will lead to expensive *Page Table Walk* (PTW). This process is shown in Fig. 1.

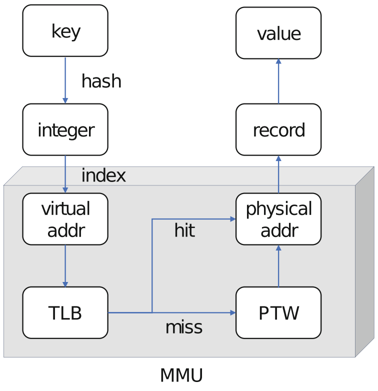

Fig. 1. The process of data retrieval of a key-value system

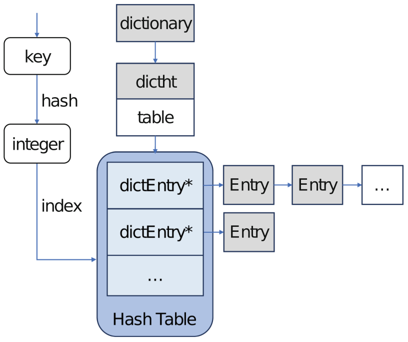

Fig. 2. The process of data retrieval in Redis dictionary

Redis uses ziplist or dictionary for key-value store. Ziplist is for small amounts of data storage. In the Redis dictionary, the value is retrieved from the key by the following steps shown in Fig. 2. First, the key is mapped to the hash table of the dictionary by a hash function, and the hash table stores pointers to key-value entries that are the virtual addresses. The virtual address is converted to the physical address by the TLB, and then the value entry is obtained from memory by the physical address. These steps are repeated for each memory access, causing large address translastion overhead. Due to the high data volume demand for key-value stores, a key-value system actually requires a TLB that is much larger than the hardware can offer to reach a sufficient TLB hit ratio, and if a TLB miss occurs in address translation, the system needs to perform PTW to obtain the page table entry, which means additional overhead. Research work [2] has shown that the overhead from address translation and key finding could account for more than 50% in data retrieval.

Our optimization focuses on the process of key-value retrieval. This paper develops a software solution, EKRM that can reduce chained memory accesses and TLB misses, while being simple to be implemented in Redis and also compatible with many other methods. The main contribution of the paper is that, EKRM significantly accelerate data retrieval in Redis by reducing total addressing translation and memory access overhead. It surpasses the address caching method SLB [4] and achieves performance comparable to the software-hardware co-designed address-centric approach STLT [2].

2 Related Work

Key-value store efficiency has been studied in depth in the past decades. It is widely used to accelerate lookup in hash table by software and hardware in previous work. The idea of hardware-based caching addresses has been widely used in key-value stores to get the physical address of the data in a more efficient way. SDC [8] is a high-performance software-defined cache which uses a part of the space in last level cache to assist the lookup, enabling programs to use cache as a look-aside key-value buffer. HTA [10] is a software-hardware co-designed method that creates a cache for hash table, which is cache-friendly and can serve most of the hash lookup work by hardware. The limitations of these methods are that the record size are limited and the main reliance on hardware cache creates the tension between the capacity and cost.

SLB [4] uses a software cache that stores the addresses of recently accessed records. It has demonstrated promising performance but requires much more storage and needs a trade-off between the capacity and cost. STLT [2] was proposed based on the concept of address-centric approach. It reduces address translation overhead and enhances TLB's performance as address translation accounts for a large part of the total time spent on a key-value lookup. STLT is a kind of look-aside buffer that bypasses the index, while it places the index, virtual address, and page table entries in the cache instead of the value. It can bring up to 1.4 times acceleration on Redis in a simulation experiment, but needs hardware support to be fully implemented in a real environment. This paper also proposes a scheme that can reduce total address translation overhead, but uses a relatively low-cost approach by software.

Besides, there are some typical relevant solutions that accelerate data retrival in key-value store. pRedis [6] is an improved scheme for cache replacement algorithm in key-value store, which mainly solves the problem that traditional cache replacement algorithm cannot balance locality and miss penalty by a new approach to evaluate the miss penalty of each access. Cavast [7] uses the characteristic of pages in modern operating systems to place data in appropriate positions of cache, thereby reducing the eviction of frequently accessed keys. WIDX [9] is a unit specially designed for hash tables, consisting of a hash calculation unit and a walker, which can reduce the overhead from the main hash table.

3 Motivation

In Sect. 1, we introduced the main optimization difficulties and bottlenecks of the current key-value system and some existing optimization schemes. We summarize the most important issues as follows. First, hash tables with linked lists have the characteristics of poor locality. Under a large number of key-value accesses, hash table access has become an important performance bottleneck. Second, the virtual address to physical address conversion overhead resulting from Redis multiple chained memory accesses accounts for a large part of the data retrieval process. We use the following experiment to illustrates this issue. We divided the cost of data retrieval into the three following parts: the cost of address translation and memory access related to hash tables and entries, the cost of key hash and compare to find the corresponding entry, and other cost like data maintenance to finish a single command. We record the ratio of the cumulative total time from key to value to the total execution time of these commands. The results are collected using 0.1 to 1000 million of distinct keys and 10× *GET* operations generated by YCSB. The overhead of the previous two parts increases from 49.6% to 63.1% and among which the overhead of address translation and memory access related to hash tables and entries increases from 34.5% to 52.3% as shown in Fig. 3. As the number of stored keys grows, address translation and memory access overhead related to the hash table and entries can become the bottleneck. This cost is impacted by the design of Redis as we mentioned above.

In Sect. 2, we introduced some typical methods to accelerate data retrieval in a key-value store. Based on the experiment above, we focus on optimizing the hash table and related parts in Redis dictionary to reduce the overhead of address translation and memory access, as optimizing this part is of great significance to the acceleration of the data retrieval in Redis. Our idea is to better organize the hash table entries indexed by hash functions, so that Redis can find the corresponding entry of a key faster. The general idea is to create a specially managed memory space to hold key-value entries. This space is divided into two parts and the first part is used as a hash table which can be directly indexed by some simple hash functions. The second part is for handling hash collisions. If we can find the entry corresponding to the key indexed by a hash function, there will be no need for executing the original lookup method. The original data structure of Redis is still unchanged, therefore the lookup methods in Redis dictionary still works, keeping the integrity of the Redis data structure. The advantage of this design is that, first, it can reduce the access frequency of the original hash table because a large part of the entries can be directly found

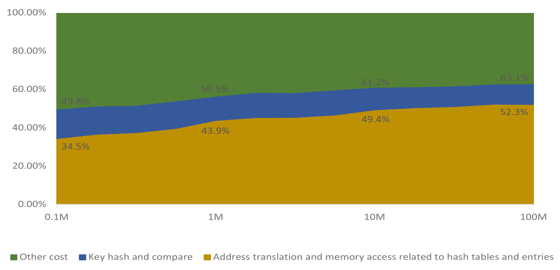

Fig. 3. Breakdown of the execution time of Redis. The results were collected using different numbers from 0.1 million to 100 million of distinct keys and 10× GET operations generated by YCSB. We use Unix domain socket to transmit data and do not count network transmission overhead.

in the first part of the space, second, these new hash functions map the keys to a continuous address space for holding entries, and third, our design can be regard as a bypass approach for a regular value request, retaining the ability to resist potential hash attacks.

When Redis manages a huge number of key-value pairs, finding entries within regular-page memory could result in large address translation overhead. To further reduce TLB misses and total translation overhead, we consider using huge pages specially for the memory space mentioned above.

4 EKRM Design

This section presents the detail design of EKRM to accelerate key-value lookup operations.

4.1 Design Overview

To reduce the occurrences of hash function collisions and resist hash attacks, Redis usually uses relatively complex functions like *SipHash* or *MurmurHash*, and uses a chaining method to solve hash collisions. Therefore, it often takes multiple address accesses to find the required entry from the key, resulting in a large latency.

To be compatible with the original design of Redis, EKRM's scheme is as follows. EKRM stores the entries of the dictionary within a continuous address space of huge pages, and these entries are indexed by fast hash functions with lower computational complexity, which mainly perform simple bitwise operations on the key. When inserting key-value pairs, EKRM prefers to allocate the entry in the position indexed by the fast hash functions. If a hash collision happens and can not be resolved, the entry will be placed in the second part of the huge page space. The second part is managed by a free list in which one available position is linked to the next, and EKRM can allocate an available position dynamically. It is feasible to allocate memory for the second part when the space is not enough to hold entries. When the hash collision ratio is not high, most of the entries can be quickly indexed by a fast function in the first part, which reduces the number of memory accesses in the key-value query process. To increase the hit ratio of the fast lookup approach, EKRM uses two fast hash functions $H1$ and $H2$ that work like Cuckoo hashing [11]. Besides, there is extra space to store metadata in the design of the Redis entry data structure, so several databases in Redis can share the fast hash table by marking their serial numbers in entries.

4.2 EKRM Operations

EKRM supports INSERT, GET, and DELETE operations.

INSERT. The operation attempts to insert a new key-value pair into Redis and the *key* should not be in any existing entries. If the position in fast hash table indexed by any fast hash function $H1$ or $H2$ is available, a new entry of the key-value pair will be placed into the corresponding position, otherwise EKRM will work like Cuckoo hashing and attempts to move the entry of $key2$ that occupies index $H1(key)$ to position $H2(key2)$ or $H1(key2)$. In order to control the overhead, at most one entry in fast hash table is allowed to migrate in an INSERT operation. When all the above attempts fail, the entry will be placed into the second part of the space. After the position of the entry is determined, EKRM still needs to execute the original insertion process to maintain the original hash table and linked lists in Redis. If an entry in the fast hash table is moved, its relevant information in Redis dictionary should be updated.

GET. The operation attempts to look up the corresponding value to the given key. If any of the two hash functions in the fast hash table indexes the corresponding entry to the key, the operation is successful, otherwise, the original lookup approach of Redis will be used. In this operation, the fast lookup method does not conflict with the original one and can be regarded as a bypass approach.

This process is illustrated in Fig. 4 and an example is shown. The entries in the fast hash table in Fig. 4 can be directly indexed by the fast hash functions $H1$ and $H2$, and the entries in the second part can only be accessed by the linked lists from the original Redis hash table. In the hash table directed by fast hash functions, a hash line contains one 32-byte entry. For example, if the key-value pair is *Bob:002*, there will be two lookup approaches. By fast hash function $H2$ in Fig. 4, we get index 1 and find the corresponding entry in the table line, or by calculating Redis hash function, we get index 1 and find the entry through a linked list from *Alice* to *Bob*. The fast lookup approach is prioritized.

DELETE. The operation attempts to delete a key-value pair in Redis. EKRM finds the entry and marks its position as available, and updates the free list if necessary. The Redis structure will also be updated according to the original method.

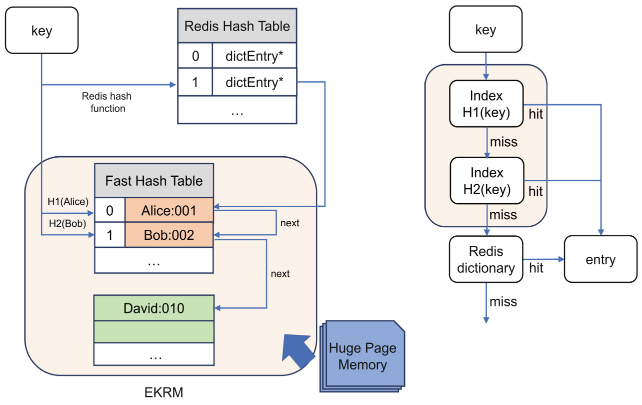

Fig. 4. The process of data retrieval in EKRM

4.3 Other Supports

Linux operating system supports allocating 2MB and 1GB huge pages, and we set */proc/sys/vm/nr_ hugepages* to control the usage of huge pages. Note that we do not use transparent huge pages. In the code implementation, we use the *mmap()* function to map a memory space to the allocated huge pages. Before starting Redis, we need to estimate the working set size, and allocate suitable memory of huge pages and divide it into two reasonably sized regions. The probability of success lookup by fast hash functions can be increased at the cost of additional memory overhead.

Redis generally does not recommend using huge pages to take advantage of the Copy-on-Write (CoW) mechanism of forked processes to achieve persistent storage and data security in the *Redis Database* (RDB) and *Append Only File* (AOF) functions [28]. In addition to the time-consuming page table copying, some study shows that there will also be additional delays caused by the usage of huge pages [16,17]. Note that we only allocate huge page memory specifically for storing key-value entries. To address these problems, we set the address space that organizes as huge pages to be Copy-on-Write free, to avoid the extra overhead caused by a large number of huge pages after Redis forks a sub-process. We record a backup of the huge page space, and start recording the latest updates that occur in the memory space after the RDB sub-process is created, and delete the records after the sub-process finishes. Generally, these changes do not have much impact on performance in our tests.

5 Evaluation

In this section, we evaluate the performance of the proposed EKRM on different workload distributions of Redis and provide speedup analysis, sensitivity studies on the design, and discuss some special cases.

5.1 System Configurations

We perform our experiment in a 64-bit X86-64 system shown in Table 1. The L1d TLB for general 2M pages is 4-way, 32 entries and for 4K pages is 4-way, 64 entries. The L2 shared TLB for 4K and 2M pages is 6-way, 1536 entries.

Table 1. System Configurations

Component	Parameter
ISA	64-bit X86-64
CPU	16 core, Intel Xeon Silver 4216 CPU @ 2.10GHz
L1d TLB	2M/4M pages: 4-way, 32 entries 1G pages: 4-way, 4 entries 4K pages: 4-way, 64 entries
L2 TLB	4K/2M pages: 6-way, 1536 entries
L1d Cache	512KB
L2 Cache	16MB
L3 Cache	22MB

5.2 Benchmarks and Datasets

Zipfian distribution was observed in most database workloads by Cooper et al. [3], who proposed the *Yahoo! Cloud Serving Benchmark* (YCSB) based on it. The same distribution was confirmed for key-value stores in production environments by other studies [18–20]. Spatial locality in key-value store was reported by recent studies [21,22].

Considering that the keys to be looked up in the real world often have a certain distribution pattern, we set the search or update frequency of different keys to have a Zipfian, Uniform, or Latest distribution, among which Zipfian distribution has an alpha value of 0.99, and Latest distribution data satisfies that the data inserted recently is more likely to be accessed. We use YCSB for generating data and set 5% of requests as UPDATE operations and the others as GET operations. ETC is the closest workload to a general-purpose one, with the highest miss ratio of more than 19% in all Facebook's Memcached pools [25]. We export the data stream generated by mutilate [31] and run it on our Redis testing program. These workloads have 10 million keys and 100 million key accesses. The size of the keys is set to 24 bytes.

Data retrieval efficiency is an essential part that influences performance of Redis. As we have discussed in Sect. 3, our optimization work accounts for a large portion of the total execution time excluding network overhead, we exclude other components of Redis in our measurement, including data fetching, data validation, conversions between input/output commands, and their internal representations and other possible overhead. Those components are not the target of EKRM, but the focus of other complementary techniques. We intend not to consider the overhead of network communication and other data maintenance work within Redis, in order to focus on the overhead of data retrieval in Redis. In our testing program, we repeatedly execute the functions of data retrieval or modification for each requested key in the Redis program in a loop and record the total time usage.

We use *Siphash* as the default hash function for Redis. In three YCSB workloads, the tests are divided into two phases. The first phase contains *SET* operations that insert data into Redis, and the second phase performs search or update operations including *GET* and *SET*. We first run YCSB on the general redis-server, and then export the testing traces from the database.

5.3 Speedup on Redis

In the following experiment, we set the memory usage of 2 MB huge pages as 1 to 8 times of the total size of the entries and set the size of the second part of as 1/4 of the first part. A comparative method to EKRM is SLB, the software-implemented address caching method introduced in Sect. 2. We use the original Redis program as the baseline and measure the speedup effect of both methods.

The speedups are shown in Fig. 5. The memory usage of SLB refers to the extra memory for address caching besides the origin memory to store entries, while EKRM places these entries in a special memory space. On average, EKRM brings up to 1.38$\times$ average speedups while SLB only brings 1.16$\times$ average speedups with 8$\times$ memory usage of the total size of the entries on these workloads. EKRM also shows a significant advantage against SLB with different sizes of memory usage. Choosing the memory usage as about 2$\times$ of all entries' size is a relatively suitable option for EKRM, as the speedup has obvious improvement compared to the size of 1$\times$, while further increasing it will reduce the sensitivity. The results show that EKRM consistently outperforms SLB in our workloads substantially and can run without extra memory demands.

We also compare the performance of EKRM with STLT using SniperSim [26] simulating the 64-bit X86, Gainestown architecture. The datasets and testing program are the same as above. As is shown in Fig. 6, STLT can brings 1.38$\times$ speedups on average to Redis and EKRM brings 1.31$\times$ speedups on average when using 2$\times$ memory of the total entries size. Our software-implemented method can match the performance of hardware-supported STLT.

Speedup Analysis. We record a hit when the entry is successfully indexed by either of the fast hash functions and record a miss when both of them fail. We measure dTLB and last-level cache (LLC) misses by Linux perf [27], which is a useful tool for performance analysis. The results are compared to the original Redis version.

The speedup of EKRM for address translation and memory access is the result of the combination of dTLB miss reduction and cache miss reduction.

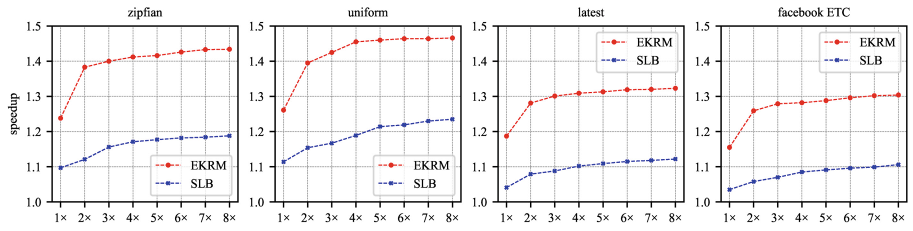

Fig. 5. The speedup of EKRM and SLB in workloads of different memory usage from 1× to 8× of the total entries size

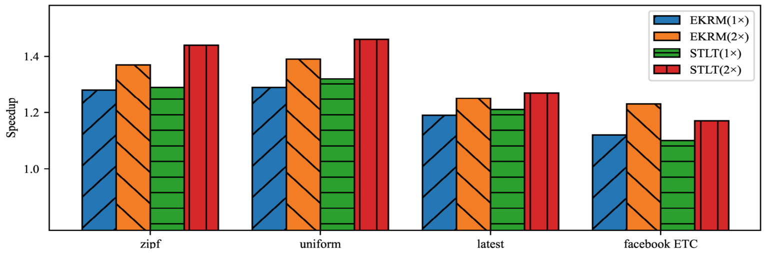

Fig. 6. The speedup of EKRM and STLT of 1× and 2× memory usage in workloads

Figure 7 shows the average hit ratio by fast hash functions and the average dTLB miss reduction on workload in Sect. 5.2. EKRM has a significant effect on reducing dTLB misses in the four workloads. The average number of dTLB misses declines by 52%. For data randomly generated in the key space of three YCSB distributions, the average hit ratio of the fast hash functions can reach about 90% with huge page memory usage of 2× size of the total inserted entries. EKRM has a relatively lower hit ratio on ETC workload due to compulsory misses.

EKRM has the highest speedup on the workload of Uniform distribution, while has the lowest speedup on the workload of Latest distribution in 3 YCSB workloads. Among the three distributions, Latest has the best data locality, Zipf modest, and Uniform the worst, so the original Redis program performs the best on the workload of Latest distribution and performs the worst on the workload of Uniform distribution. The dTLB reduction by EKRM also shows that the original Redis program suffers from the most severe dTLB misses on the workload of Uniform distribution and the least in Latest distribution. Additionally, the query requests under the Latest distribution are more often related to the data that have been inserted recently, and these data have a lower probability of being hit by the fast hash functions compared to the earlier data due to possible hash collision, while retrieving these data by the original Redis approach is less likely to suffer from dTLB miss and cache miss. The decline of the hit ratio by the fast hash functions on the workload of Latest distribution in Fig. 7 shows this problem. The worse the locality of the workload in these distributions, the

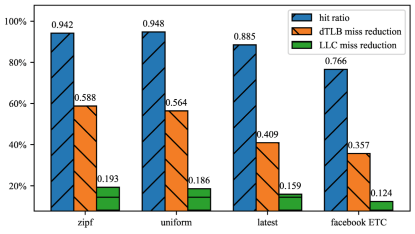

Fig. 7. The hit ratio, dTLB miss reduction and LLC miss reduction of EKRM with huge page memory usage of 2× size of the total inserted entries

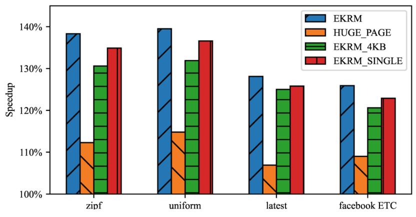

Fig. 8. Speedup for different EKRM configurations

more advantage EKRM has over the original Redis. Speedup on facebook ETC workload is the worst, as it has the lowest hit ratio caused by compulsory misses [25], and this means EKRM has to perform more original lookups, leading to more dTLB misses and LLC misses.

Breakdown of Speedup. We conduct an ablation study for EKRM by configuring it in different ways. Figure 8 shows their speedups over original Redis.

HUGE_PAGE is the configuration that only uses huge page memory to store entries and does not enable fast lookup approach. EKRM_4KB allocates 4KB size pages for the fast hash table and otherwise works the same as EKRM. EKRM_SINGLE uses only a single fast hash function. The main contribution of speedup comes from the fast lookup. EKRM_4KB without usage of huge pages still shows obvious speedup over original Redis.

5.4 Case Studies

Parallel Optimization in Situation of High Miss Ratio. In Sect. 5.3, our experiment shows that although our fast lookup method can speed up key-value lookup in most cases, it may increase the overhead in the situation of a high miss ratio. One solution is that when EKRM detects that the miss ratio of keys is significantly high, it prioritizes the original lookup method of Redis to some extent. The effectiveness of this feature depends on how frequently we monitor and how aggressively we adjust.

Since our fast lookup approach and the original approach are independent and our design can be regarded as a bypass solution, using parallel optimization can be an effective way. Two threads are dispatched to execute the fast lookup and Redis hash, respectively, and once one thread successfully finds the entry, the other task is terminated. If the overhead by synchronization is low enough, parallel optimization will work effectively in situations of high miss ratio. To reduce the overhead of synchronization between threads, establishing a low-cost communication mechanism based on hardware is a possible solution.

We use instruction-level synchronization based on register renaming dependency to implement the optimization in the Gem5 simulator [32], and our parallel version shows 10% performance improvement compared to the sequential version when the miss ratio is higher than 30%. Therefore, EKRM can be potentially accelerated by parallel lookups.

Numbers of Entries of an Index. We study the speedup of using different numbers of entries of an index in EKRM. The speedup is almost the same when using 1 or 2 entries by an index, and decreases when using more. Using 2 entries can increase the hit ratio in the fast hash table while increasing dTLB misses and LLC misses, which is mainly caused by extra key comparison.

Memory Usage. EKRM requires additional memory for a larger index table to increase the hit ratio of fast hash functions. In our implementation, it is required to allocate the huge page memory space in advance with an estimate of the size of the working set. By adding a rehash method which has been implemented in Redis, the size of our index table can be also adjustable but it brings up more overhead.

In our design, EKRM is suitable for accelerating data retrieval when memory is relatively abundant and the size of the working set is relatively stable. When allocated with an appropriate size of memory and with a relatively stable size of the working set, EKRM has little chance of suffering from fragmentation. When the memory is insufficient, Redis will start the elimination strategy and EKRM can consider replacing entries in the fast hash table that are seldom accessed.

6 Conclusion and Future Work

This paper presents a new method to accelerate key-value lookup in Redis. It specially focuses on reducing chained memory access and data addressing overhead, utilizing an efficient lookup method using fast hash functions. The new solution places the entries of key-value records into a continuous address space organized as huge pages and enables most of them to be quickly indexed. The new method reduces TLB miss in the lookup process and preserves the integrity of Redis, and addresses the CoW and fragmentation problems caused by the usage of huge pages in some ways. Experiments on various request distributions show that it brings up to 1.38× speedups on the key-value retrieval process for a set of Redis workloads.

For future work, EKRM can be explored to work in parallel by a mechanism with less synchronization overhead. Accelerating EKRM with a mechanism like direct segment [5] instead of huge page is possible to further lower the address translation overhead. Additionally, bypassing infrequently accessed entries in last-level-cache [23,24] is a possible approach.

Acknowledgments. The research is supported in part by the National Key R&D Program of China under Grant No. 2022YFB4500701, and by the National Science Foundation of China (Nos. 62032001, 62032008, 62372011).

References

1. Kwon, M., Lee, S., Choi, H., Hwang, J., Jung, M.: Realizing strong determinism contract on log-structured merge key-value stores. ACM Trans. Storage **19**, 1–29 (2023). https://doi.org/10.1145/3582695
2. Ye, C., Xu, Y., Shen, X., Liao, X., Jin, H., Solihin, Y.: Hardware-based address-centric acceleration of key-value store. In: 2021 IEEE International Symposium On High-Performance Computer Architecture (HPCA), pp. 736–748 (2021)
3. Cooper, B., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking Cloud Serving Systems with YCSB. Association for Computing Machinery (2010). https://doi.org/10.1145/1807128.1807152
4. Wu, X., Ni, F., Jiang, S.: Search lookaside buffer: efficient caching for index data structures. In: Proceedings of the 2017 Symposium on Cloud Computing, pp. 27–39 (2017). https://doi.org/10.1145/3127479.3127483
5. Basu, A., Gandhi, J., Chang, J., Hill, M., Swift, M.: Efficient virtual memory for big memory servers. In: Proceedings of the 40th Annual International Symposium on Computer Architecture, pp. 237–248 (2013). https://doi.org/10.1145/2485922.2485943
6. Pan, C., Luo, Y., Wang, X., Wang, Z.: PRedis: penalty and locality aware memory allocation in redis. In: Proceedings of the ACM Symposium on Cloud Computing, pp. 193–205 (2019). https://doi.org/10.1145/3357223.3362729
7. Wang, K., Liu, J., Chen, F.: Put an elephant into a fridge: optimizing cache efficiency for in-memory key-value stores. Proc. VLDB Endow. **13**, 1540–1554 (2020). https://doi.org/10.14778/3397230.3397247
8. Ni, F., Jiang, S., Jiang, H., Huang, J., Wu, X.: SDC: a software defined cache for efficient data indexing. In: Proceedings of the ACM International Conference on Supercomputing, pp. 82–93 (2019). https://doi.org/10.1145/3330345.3330353
9. Kocberber, O., Grot, B., Picorel, J., Falsafi, B., Lim, K., Ranganathan, P.: Meet the Walkers: Accelerating Index Traversals for in-Memory Databases. Association for Computing Machinery (2013). https://doi.org/10.1145/2540708.2540748
10. Zhang, G., Sanchez, D.: Leveraging caches to accelerate hash tables and memoization. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 440–452 (2019). https://doi.org/10.1145/3352460.3358272
11. Pagh, R., Rodler, F.: Cuckoo hashing. J. Algorithms **51**, 122–144 (2004). https://www.sciencedirect.com/science/article/pii/S0196677403001925
12. Lepers, B., Balmau, O., Gupta, K., Zwaenepoel, W.: KVell: the design and implementation of a fast persistent key-value store. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, pp. 447–461 (2019). https://doi.org/10.1145/3341301.3359628
13. Zhang, K., Hu, J., He, B., Hua, B.: DIDO: dynamic pipelines for in-memory key-value stores on coupled CPU-GPU architectures. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 671–682 (2017)
14. Kaiyrakhmet, O., Lee, S., Nam, B., Noh, S., Choi, Y.: {SLM-DB}: {Single-Level} {Key-Value} store with persistent memory. In: 17th USENIX Conference on File and Storage Technologies (FAST 2019), pp. 191–205 (2019)
15. Zhang, T., et al.: FPGA-accelerated compactions for {LSM-based}{Key-Value} store. In: 18th USENIX Conference on File and Storage Technologies (FAST 2020), pp. 225–237 (2020)
16. Liu, H., Liu, R., Liao, X., Jin, H., He, B., Zhang, Y.: Object-level memory allocation and migration in hybrid memory systems. IEEE Trans. Comput. **69**, 1401–1413 (2020)

17. Heo, T., Wang, Y., Cui, W., Huh, J., Zhang, L.: Adaptive page migration policy with huge pages in tiered memory systems. IEEE Trans. Comput. **71**, 53–68 (2020)
18. Chen, J., et al.: {HotRing}: a {Hotspot-Aware}{In-Memory}{Key-Value} store. In: 18th USENIX Conference on File and Storage Technologies (FAST 2020), pp. 239–252 (2020)
19. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: 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, pp. 53–64 (2012)
20. Wu, X., Zhang, L., Wang, Y., Ren, Y., Hack, M., Jiang, S.: Zexpander: a key-value cache with both high performance and fewer misses. In: Proceedings of the Eleventh European Conference on Computer Systems, pp. 1–15 (2016)
21. Cao, Z., Dong, S., Vemuri, S., Du, D.: Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at Facebook. In: 18th USENIX Conference on File and Storage Technologies (FAST 2020), pp. 209–223 (2020)
22. Gilad, E., et al.: EvenDB: optimizing key-value storage for spatial locality. In: Proceedings of the Fifteenth European Conference on Computer Systems, pp. 1–16 (2020)
23. Gaur, J., Chaudhuri, M., Subramoney, S.: Bypass and insertion algorithms for exclusive last-level caches. In: Proceedings of the 38th Annual International Symposium on Computer Architecture, pp. 81–92 (2011)
24. Park, J., Park, Y., Mahlke, S.: A bypass first policy for energy-efficient last level caches. In: 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), pp. 63–70 (2016)
25. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: Workload analysis of a large-scale key-value store. SIGMETRICS Perform. Eval. Rev. **40**, 53–64 (2012). https://doi.org/10.1145/2318857.2254766
26. Carlson, T., Heirman, W., Eyerman, S., Hur, I., Eeckhout, L.: An evaluation of high-level mechanistic core models. ACM Trans. Archit. Code Optim. **11** (2014). https://doi.org/10.1145/2629677
27. Linux kernel. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git. Accessed 15 Mar 2024
28. Redis. https://redis.io/. Accessed 15 Mar 2024
29. Memcached. https://memcached.org/. Accessed 15 Mar 2024
30. Redis documentation. https://redis.io/docs/management/optimization/latency/. Accessed 15 Mar 2024
31. Mutilate. https://github.com/leverich/mutilate. Accessed 15 Mar 2024
32. Gem5. https://github.com/gem5/gem5. Accessed 15 Mar 2024