HotRing: A Hotspot-Aware In-Memory Key-Value Store

Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu,
Yuanyuan Sun, Huan Liu, and Feifei Li, *Alibaba Group*

https://www.usenix.org/conference/fast20/presentation/chen-jiqiang

This paper is included in the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-12-0

HotRing: A Hotspot-Aware In-Memory Key-Value Store

Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu,
Yuanyuan Sun, Huan Liu, Feifei Li
Alibaba Group

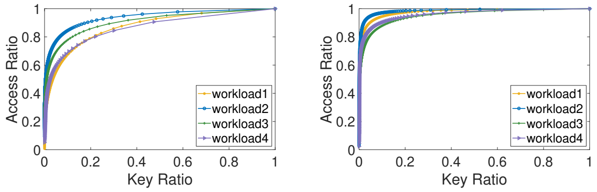
Abstract

In-memory key-value stores (KVSes) are widely used to cache hot data, in order to solve the *hotspot issue* in disk-based storage or distributed systems. The hotspot issue inside in-memory KVSes is however being overlooked. Due to the recent trend that hotspot issue becomes more serious, the lack of hotspot-awareness in existing KVSes make them poorly performed and unreliable on highly skewed workloads.

In this paper, we explore hotspot-aware designs for in-memory index structures in KVSes. We first analyze the potential benefits from ideal hotspot-aware indexes, and discuss challenges (i.e., *hotspot shift* and *concurrent access* issues) in effectively leveraging hotspot-awareness. Based on these insights, we propose a novel hotspot-aware KVS, named HotRing[1], that is optimized for massively concurrent accesses to a small portion of items. HotRing is based on an ordered-ring hash index structure, which provides fast access to hot items by moving head pointers closer to them. It also applies a lightweight strategy to detect hotspot shifts at run-time. HotRing comprehensively adopts lock-free structures in its design, for both common operations (i.e., read, update) and HotRing-specific operations (i.e., hotspot shift detection, head pointer movement and ordered-ring rehash), so that massively concurrent requests can better leverage multi-core architectures. The extensive experiments show that our approach is able to achieve 2.58× improvement compared to other in-memory KVSes on highly skewed workloads.

1 Introduction

The in-memory key-value store (KVS) is an essential component in storage infrastructures (e.g. databases, file systems) that caches frequently accessed data in memory for faster access. KVSes help to improve the performance and scalability of these systems, where billions of requests need be processed in each single second. Many state-of-the-art KVSes, e.g., Memcached [44], Redis [31] and their variants [8, 15, 17, 33], are widely developed and deployed in production environments of enterprises, such as Facebook [42], Amazon [3], Twitter [49] and LinkedIn [12].



(a) Daily distribution (b) Extreme distribution

Figure 1: Access ratio of different keys.

The *hotspot issue* (i.e., a small portion of items that are frequently accessed in a highly-skewed workload) is a common problem in real-world scenarios, and has been studied extensively in literature [4, 10, 20, 23, 27]. There are many solutions to address cluster-wide hotspots, such as consistent hash [29], data migration [9, 11, 46] and front-end data caching [16, 26, 32, 36]. Besides, the single-node hotspot issue is also well addressed. For example, computer architecture leverages hierarchical storage layout (e.g., disk, RAM, CPU cache) to cache frequently accessed data blocks in low-latency storage medium. Many storage systems, e.g., LevelDB [18] and RocksDB [14], use in-memory KVSes to manage hot items.

However, the hotspot issue inside an in-memory KVS is usually being overlooked. We have collected access distributions in in-memory KVSes from Alibaba's production environments, as illustrated in Figure 1. We observe that 50% (daily cases) to 90% (extreme cases) of accesses only touch 1% of total items, which shows that the hotspot issue becomes unprecedentedly serious in the Internet era. There are several reasons behind this phenomenon. First, the population of active users in online applications keeps growing. A real-time event (e.g., online sales promotions, breaking news) is able to attract billions of accesses to a few items in a short period of time, where fast access to these hotspots is critical. It has been reported that every 0.1s of loading delay would

[1] HotRing is a subcomponent of *Tair* — a NoSQL product extensively used in Alibaba Group and publicly available on Alibaba Cloud.

cost Amazon 1% in sales, and every 0.5s of additional load delay for Google search results would lead to a 20% drop in traffic [35]. Second, infrastructures beneath such applications become complex. It is common that a minor error (e.g., due to software bugs or configuration mistakes) somewhere in the pipeline may lead to (unpredictably) repeated accesses to an item (e.g., read and return an error message endlessly). It is desired that such unpredicted hotspots shall not crash or block the entire system. Hence, keeping a KVS performant and reliable in the existence of hotspots is of great importance.

Many index structures can be used to implement a KVS, such as skip list [14, 18], balanced/trie trees (e.g., Masstree [37]), and hashes (e.g., Memcached [44], MemC3 [15], MICA [33], FASTER [8]), where hashes are the most popular due to faster lookups. However, we observe that most approaches are not aware of hotspots, i.e., they indistinguishably manage all items via a same policy. In such case, reading a hot item involves the same number of memory accesses compared to other items. From a theoretical analysis (detailed in § 2.2), we find that looking up hotspots in current hash indexes requires much more effort than from an ideal strategy. Though there exist mechanisms to reduce memory accesses, they only render limited efficacy. For example, CPU cache helps to speedup hotspot accesses, but has only 32MB of capacity. Rehash operation helps to reduce the length of each collision chain, but significantly increases the memory footprint. This situation provides us opportunities to further optimize hotspot accesses in highly-skewed workloads.

In this paper, we propose HotRing, a hotspot-aware in-memory KVS that leverages a hash index optimized for massively concurrent accesses to a small portion of items, i.e., hotspots. The initial idea is to make memory accesses required for looking up an item (negatively) correlated to its hotness, i.e., the hotter items shall be read faster. To achieve this goal, two challenges have to be addressed: *hotspot shift* - the set of hot items keeps shifting, and we need to detect and adapt to such shifts in a timely manner; *concurrent access* - hotspots are inherently accessed by massively concurrent requests, and we need to sustain high concurrency for them. For the hotspot shift issue, we replace the collision chain in the hash index with an *ordered-ring* structure, such that bucket headers can directly re-point to hot items as hotspots shift, without compromising correctness. In addition, we use a lightweight mechanism to detect hotspot shift at run-time. For the concurrent access issue, we adopt a lock-free design inspired by existing lock-free structures [19, 50], and extend it to support all operations required by HotRing, including hotspot shift detection, head pointer movement and ordered-ring rehash.

We have conducted extensive experimental evaluations on benchmarks that simulate real workload, and have compared HotRing with lock-free chain-based hashes and other baselines. The results show that, in extremely skewed workloads, HotRing processes up to 565M read requests per second, providing 2.58× improvement over other systems. It also achieves 2.17× and 1.32× improvement for in-place-updates and read-copy-updates respectively. This verifies that HotRing is an effective structure to improve the capability of hotspot processing on each single node, making it a performant and reliable in-memory KVS.

Our main contributions are summarized as follows:

- We identify the hotspot issue in existing in-memory indexes, and demonstrate that hotspot-aware designs have great potential to improve performance for hot items.
- We propose HotRing, an ordered-ring hash structure, as the first effort to leverage hotspot-aware designs. It provides fast access to hot items by moving head pointers closer to them. It also adopts a lightweight strategy to detect hotspot shifts at run-time.
- We make HotRing lock-free to support massively concurrent accesses. In particular, we design from scratch HotRing-specific operations, including hotspot shift detection, head pointer movement and ordered-ring rehash.
- We evaluate our approach on real-workload-based benchmarks. The results show that HotRing significantly outperforms other KVSes when the accesses are highly-skewed.

The rest of this paper is organized as follows. §2 introduces hash indexes and hotspot issues, and discusses opportunities and challenges for hotspot-aware hashes. §3 elaborates the detailed design of HotRing, and §4 evaluates its performance. Lastly, §5 reviews related work and §6 concludes the paper.

2 Background & Motivation

In this section, we first introduce the hash index and hotspot issues in existing KVSes. We then show potential benefits from ideal hotspot-aware hashes theoretically. At last, we discuss challenges to effectively leverage hotspot-awareness in practical indexes, as well as our design principles.

2.1 Hash Indexes and Hotspot Issues

Hash index is the most popular in-memory structure used in KVSes, especially when range queries are not needed by upper applications. Figure 2 illustrates the typical structure of a hash index, which contains a global *hash table* and one *collision chain* for each entry in the table. To access a key, we first calculate its hash value h to locate the corresponding entry head, and then check items in the collision chain until that key is found or the end of chain is reached (i.e., key not exists). A n-bit hash value can be further divided into a hash table part (e.g., k-bit) and a *tag* part (e.g., $(n-k)$-bit). The tag can be included in each item to avoid comparing long keys [8, 33]. As can be seen in Figure 2, hash indexes are not aware of hotspots, i.e., hot items might be distributed evenly

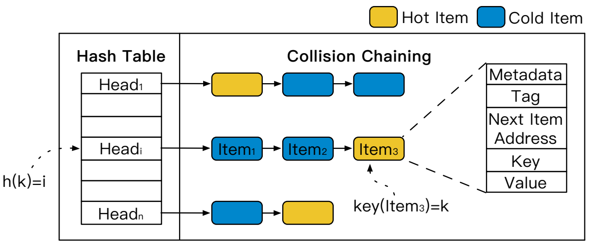


Figure 2: The conventional hash index structure.

in collision chains, For a hot item placed close to the tail of the collision chain (e.g., Item3 in the figure), it requires more memory accesses than other items in front. However, in highly skewed workloads, slight increase of hot-item access cost may result in severe decline of overall performance.

There are several ways to reduce hot-item access cost, however, with only limited effects. First, CPU caches can speedup accesses to hot data blocks (i.e., in the unit of a 64-byte cacheline). However, for most commodity servers, the capacity of CPU cache is around 32 MB, while the entire memory volume exceeds 256 GB. Only 0.012% of memory can be cached, far less than observed hotspot ratios in Figure 1. To better utilize CPU cache, many cache-friendly index structures [8, 33] are proposed. Second, the hash table can be enlarged (i.e., via *rehash*) to reduce lengths of collision chains, so that locating a hot item needs fewer memory accesses. However, rehash is no longer advised when the hash table is already huge in size. For example, for two successive rehash operations, the second one requires two times the memory space, but only brings in half of the efficacy (in terms of the chain length reduction). In summary, all existing approaches only mitigate the hotspot issue to a small extent.

2.2 Potential Benefits of Hotspot-Awareness

As hotspot issue is getting serious (shown in Figure 1), it renders a rising opportunity to the design of hotspot-aware hash indexes. First of all, it is interesting to have a rough estimation and analysis on how much potential benefits we can obtain from leveraging hotspot-aware designs.

In conventional chain-based hash indexes, hot items are randomly placed in the collision chain, so that hot items and cold items are equivalent in terms of access cost. Suppose that we have N items (i.e., key-value pairs) stored in a hash table with B buckets, the average length of each bucket chain is $L = N/B$. The number of expected memory accesses to retrieve an item in the chain E_{chain} is

$$E_{chain} = 1 + \frac{L}{2} = 1 + \frac{N}{2 \cdot B} \tag{1}$$

where the leading 1 represents the lookup in the hash table.

In an ideal hotspot-aware hash index, memory accesses required to retrieve an item should be (negatively) correlated to this hotness, e.g., the hottest item needs the fewest memory accesses to retrieve. We model item hotness in a Zipfian distribution, where the access frequency f of the x-th hottest item is expressed as:

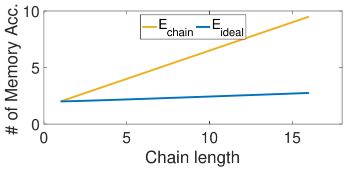


Figure 3: Expected memory accesses for an index lookup (total items $N = 2.5 \cdot 10^8$).

$$f(x) = \frac{\frac{1}{x^\theta}}{\sum_{n=1}^{N} \frac{1}{n^\theta}} \tag{2}$$

where θ is the skewness factor. To simplify the analysis, we assume that hotspots are evenly distributed in B buckets, i.e., each bucket contains exactly one from the top B hottest items, one from the top $B+1$ to $2B$ hottest items, and so on. In this case, if we can sort all items in a chain by their access frequencies (in descending order), the number of expected memory accesses to retrieve an item E_{ideal} is

$$\begin{aligned} E_{ideal} &= 1 + \sum_{k=1}^{L} F(k) \cdot k \\ &= 1 + \sum_{k=1}^{\frac{N}{B}} [\sum_{i=(k-1)\cdot B+1}^{k \cdot B} f(i)] \cdot k \end{aligned} \tag{3}$$

where $F(k)$ represents the accumulated access frequencies of the k-th item on each chain.

To estimate the potential benefits from hotspot-aware designs, we calculate the expected number of memory accesses for both traditional hash and ideal hotspot-aware hash, as shown in Figure 3. We can observe that, as collision chain length keeps growing, hotspot-aware hash significantly improves the access efficiency. This result confirms that the consideration of hotspot-awareness in a hash index is a promising direction for performance improvement.

2.3 Challenges and Design Principles

We have shown that making an index hotspot-aware is beneficial. However, there remains several challenges before we can leverage this insight in practical designs:

- **Hotspot Shift.** In real applications, access patterns keep changing over time. It is prohibitive to order all items ideally by their latest hotness. Hence, we need a lightweight approach to track the shift of hotness.

- **Concurrent Access.** Each hotspot is being inherently accessed by massively concurrent requests. Therefore, it is critical to support high concurrency for both read/write operations, in order to sustain satisfactory performance.

For the hotspot shift problem, our design principle is to avoid re-order items in the chain, but to move head pointers

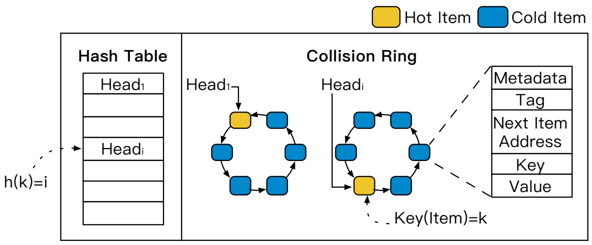


Figure 4: The index structure of HotRing.

instead, e.g., point to the hottest item or a globally better position. To ensure all items in a bucket are always accessible regardless of head pointer movement, we replace the collision chain with a *ordered-ring* structure, called HotRing (§3.1). Though this design cannot achieve optimal hotspot-awareness discussed in §2.2, we observe in experiments that it is sufficiently effective and fast. Besides, we apply two lightweight strategies to detect hotspot shift at run-time (§3.2).

For the concurrent access problem, lock-free structures are canonical solutions, with which expensive lock and synchronization operations are eliminated. Many works have demonstrated that lock-free designs can significantly improve system throughput [2, 5, 21]. Examples include read-copy-update (RCU) [13] and Hazard Pointers [40], based on atomic Compare-And-Swap (CAS). In our work, the lock-free design adopts the existing work [19, 50], which ingeniously solves the concurrency problem of deletions and insertions (§3.3). We extend this design to support all basic operations required by HotRing, including hotspot shift detection, head pointer movement and rehash (§3.4).

3 Design of HotRing

In this section, we elaborate detailed designs in HotRing that adopt hotspot-awareness, including the index structure, hotspot-shift detection strategies, and lock-free operations (i.e., read/write, insert/delete, head pointer movement, and rehash).

3.1 Ordered-Ring Hash Index

Figure 4 depicts the index structure of HotRing, which refines the structure of collision chains in conventional hash indexes. In our design, the last item in the chain is linked to the first item, forming a *collision ring*. In this manner, a head pointer in the hash table can point to any items in the corresponding ring, rather than being fixed to the front item in the chain. The design of collision ring makes it possible for HotRing to move the head pointer according to the data hotness, and scan the entire ring from any starting position. Note that if there is a single item in the ring, its next-item pointer just points to itself.

However, due to the ring-based design, there exists a serious problem: if the target item is not found, it may lead to infinite traverses in the ring. It is important to figure out when we can safely terminate the lookup process. Note that it is insufficient to mark the first item pointed by the head pointer as the stop signal, because it can be modified by concurrent requests (e.g., the marked item is deleted). Hence, we propose an ordered-ring structure to help determine the termination of lookup processes. Intuitively, we can sort items in the ring by their keys. In this case, the occurrence of item-not-found can be determined if we have already encountered two successive items that are respectively smaller and larger than the target item. Furthermore, since comparing two long keys might be costly, we utilize the tag field (introduced in §2.1) first. That is to say, an item k is ranked by the pair of its tag and key fields, i.e., $order_k = (tag_k, key_k)$.

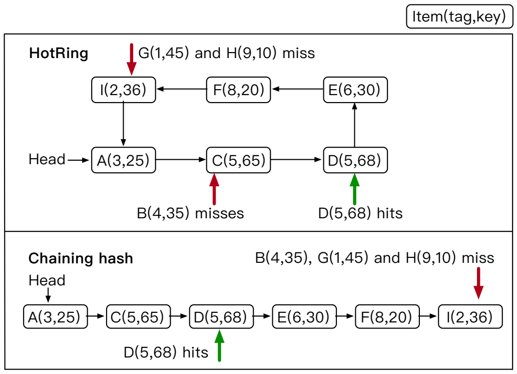


Figure 5: Find operation in HotRing.

During a lookup process for item k, suppose that the item i is being accessed, we can immediately terminate if one of the following conditions satisfies.

$Condition\ for\ Item\ Found\ (Hit):$

$$order_i = order_k \tag{4}$$

$Conditions\ for\ Item\ Not\ Found\ (Miss):$

$$\begin{cases} \quad order_{i-1} < order_k < order_i \\ or\ order_k < order_i < order_{i-1} \\ or\ order_i < order_{i-1} < order_k \end{cases} \tag{5}$$

Figure 5 illustrates all possible situations of looking up an item in HotRing. We show the dictionary order (tag, key) of each item in the figure. For example, the item C is behind item A due to $tag_A < tag_C$; and the item D is behind item C (with the same tag), because of $key_C < key_D$. The item B is confirmed to be a miss when compared to the item C, because of $tag_A < tag_B < tag_C$; the items G and H are misses when compared to the item I, because of $tag_G < tag_I < tag_F$ and $tag_I < tag_F < tag_H$ respectively. Unlike the traditional chain-based hashes, not all items in the ring have to be accessed before a miss is concluded. Assume that a ring contains n items, we only need to compare with $(n/2) + 1$ items in average for a lookup.

3.2 Hotspot Shift Identification

In ordered-ring hash index, the lookup process can easily determine whether there is a hit or miss. The remaining problem is how to identify hotspots and adjust head pointer when hotspot shift occurs.

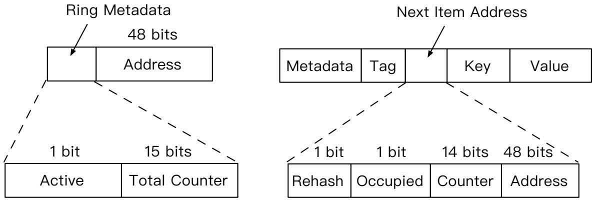
Hotspot items are evenly distributed in all buckets, due to strongly uniformed distribution of hash values. Here, we focus on hotspot identification in each bucket independently. In practice, the number of collision items in each bucket is relatively small (e.g., 5 to 10 items), so that there is usually one hotspot in each collision ring (under 10% - 20% hotspot ratio). We can improve the hotspot access by pointing the head pointers to the only hotspot, which avoids re-organizing data and reduces memory overhead. To obtain good performance, two metrics have to be concerned, i.e., identification accuracy and reaction delay. The accuracy of hotspot identification is measured by the proportion of identified hotspots. The reaction delay is the time span between the time a new hotspot occurs and the time we successfully detect it. Considering both metrics, we first introduce a *random movement* strategy that identifies hotspots with extremely low reaction delay. We then propose a *statistical sampling* strategy that provides much higher identification accuracy with relatively high reaction delay.

First of all, we define several terms used throughout this section. The first item pointed by the head pointer is called the *hot item*, and the rest items are *cold items*. Their accesses to them are defined as *hot access* and *cold access*, respectively.

3.2.1 Random Movement Strategy

Here we introduce a straightforward random movement strategy, which retains less reaction delay but achieves relatively low accuracy. The basic idea is that the head pointer is periodically moved to a potential hotspot from an instant decision, without recording any historical metadata. In particular, each thread is assigned a thread-local parameter to record the number of requests it executes. After every R requests, the thread determines whether to perform a head pointer movement operation. If the R-th access is a *hot access*, the position of head pointer remains unaffected. Otherwise, the pointer is moved to the item accessed by this *cold access*, which becomes the new *hot item*. The parameter R affects the reaction delay and identification accuracy. If a small R is used, the reaction delay to achieve stable performance will be low. However, this may also adversely cause frequent and ineffective head pointer movement. In our scenarios, data accesses are highly skewed and hence the head pointer movement tends to be infrequent. The parameter R is empirically set to 5 by default, which has been demonstrated to provide low reaction delay and negligible performance impact (as shown in Figure 15(b)).

Note that if the workload skewness is not that obvious, the random movement strategy will become inefficient. More importantly, this strategy is unable to handle multiple hotspots in a collision ring. In this case, the head pointer tends to move frequently, which does not help to speedup hotspot accesses but adversely affects normal operations.



(a) Head Pointer Format (b) Item Format

Figure 6: HotRing Index Format.

3.2.2 Statistical Sampling Strategy

In order to achieve higher performance, we design a statistical sampling strategy that aims to provide more accurate hotspot identification with slightly higher reaction delay. We first introduce the detailed formats of items and pointer in HotRing and show how to take advantage of existing formats to maintain statistics without additional space overhead. Then, we elaborate the sampling strategy to estimate access frequencies. Finally, we propose a way to derive the optimal head pointer movement when hotspot shifts, considering that multiple hotspots may exist in a ring.

Index Format. We plan to record access frequencies for all items in each collision ring. Since the physical address of modern machines occupies only 48 bits (but can be updated with 64-bit atomic compare-and-swap operations), we can utilize the remaining 16 bits to record metadata. In HotRing, each head pointer consists of three parts (as shown in Figure 6(a)): an *Active* bit, a *Total Counter* (15 bits), and the address (48 bits). The *Active* bit is a flag used to control statistical sampling for hotspot identification. The *Total Counter* records the number of accesses to the corresponding ring. Besides, the structure of an item is shown in Figure 6(b). *Rehash* is a flag to control rehash process (discussed in §3.4). *Occupied* is used to ensure concurrency correctness (discussed later in this section). HotRing uses the remaining 14 bits in the *Next Item Address* to record access counts for each item. Based on statistics maintained at both ring level and item level, the calculation of access frequencies is straightforward.

Statistical Sampling. How to dynamically identify hotspots with low overhead is a challenging problem. The hash table is usually large, e.g., contains $2^{27} \sim 2^{30}$ buckets. The simultaneous and continuous updates of statistics on massive rings will cause severe performance degradation. It is critical to minimize the overhead while retain the accuracy, which is achieved by periodical sampling in HotRing. In particular, each thread maintains a thread-local counter for processed requests. After every R requests are finished, we determine whether to launch a new round of sampling (by turning on the *Active* flag in Figure 6(a)). If the R-th access is a *hot access*, it means that the current hotspot identification is still accurate, and sampling needs not be triggered. Otherwise, it means the hotspot has shifted and we start the sampling. The parameter R is set to 5, following similar considerations as in §3.2.1. When the *Active* bit is set, the subsequent accesses to the ring are to be recorded in both *Total Counter* and corresponding items' counters. This sampling process requires additional CAS operations, and results in temporary access deficiency. To shorten this period, we set the number of samples the same as the number of items in each ring, which we believe already provides enough information to derive new hotspots.

Hotspot Adjustment. Based on collected statistics, we are able to determine new *hot item* and move the head pointer according to the access frequencies of items. After sampling process is done, the last accessing thread is responsible for frequency calculation and hotspot adjustment. First, the thread atomically resets the *Active* bit using a CAS primitive, which ensures that only one thread will perform subsequent tasks. Then, this thread calculates the access frequency of each item in the ring. The access frequency of item k is n_k/N, where N is *Total Counter* of the ring and n_k is the counter of the k-th item. Next, we calculate the income of the head pointer to each item. When the item t $(0 < t < k)$ is pointed by the head pointer, the corresponding income W_t is calculated by the following formula:

$$W_t = \sum\nolimits_{i=1}^{k} \frac{n_i}{N} * [(i-t) mod\ k] \qquad (6)$$

The income W_t measures the average number of memory accesses for the ring when item t is selected to be pointed by the head pointer. Therefore, selecting the item with the $min(W_t)$ as the *hot item* ensures that hotspots can be accessed faster. If the calculated position is different from the previous head, the head pointer should be moved using a CAS primitive. Note that the strategy not only deals with single hotspot, but also works for multiple hotspots. It helps to figure out the optimal position (e.g., may not necessarily be the hottest item) that avoids frequent movement between hotspots. After the hotspot adjustment is done, the responsible thread resets all counters to prepare for the next round of sampling in future.

Write-Intensive Hotspot with RCU. For update operations, HotRing provides an in-place update method for those values less than 8 bytes (i.e., modern machines support atomic operations for up to 8 bytes). In this case, reading and updating an item is treated the same in terms of hotness. However, the situation is completely different for larger values, as shown in Figure 7. The read-copy-update (RCU) protocol has to be applied for high performance. In this case, the preceding item's pointer needs to be modified to point to the new item during an update. If the write-intensive hotspot in the head is modified, the entire collision ring has to be traversed to reach its preceding item. That is to say, a write-intensive hot item also makes its preceding item hot. Taking this insight, we modify the statistical sampling strategy slightly. For a RCU update, the counter of its preceding item is incremented instead. This helps to point the head to the precedent of a write-intensive hotspot, making the entire RCU update operation fast.

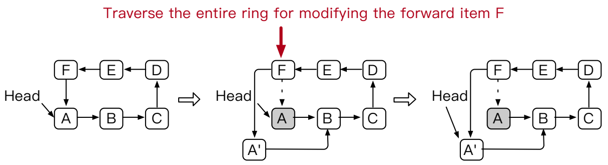


Figure 7: Updating *hot item* A with RCU makes item F hot.

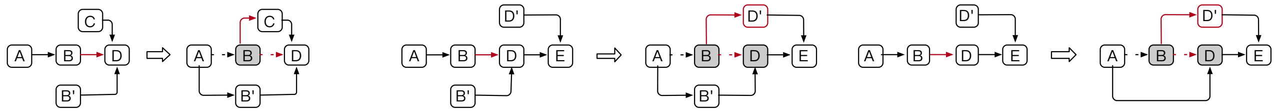
3.2.3 Hotspot Inheritance

When performing RCU update or deletion on the head item, we need to move the head pointer to another item. However, if the head pointer is moved randomly, it may point to a *cold item* with a high probability, which will cause the hotspot identification strategies to be triggered frequently. Furthermore, the performance of the system will be seriously degrades due to frequent triggering of identification strategies.

First of all, if the collision ring has only one item (i.e., the *Next Item Address* has the same position as the head pointer), the head pointer is modified by CAS to complete the update or deletion. If there are multiple items, HotRing uses existing hotspot information (i.e., head pointer position) to inherit the hotness. We design different head pointer movement strategies for both RCU update and delete operations to ensure the validity of hotspot adjustment: For the RCU update of the head item, the most recently updated item has a high probability of being accessed immediately due to the temporal locality of accesses. Hence, the head pointer is moved to the new version of the head item. For the deletion of the head item, the head pointer is simply moved to the next item, which is a straightforward and effective solution.

3.3 Concurrent Operations

The head pointer movement makes the lock-free design more complicated. This is mainly reflected in the following aspects: On one hand, the head pointer movement may be concurrent with other threads. Hence, we need to consider the concurrency of head pointer movement and other modification operations, preventing the pointer from pointing to invalid items. On the other hand, when we delete or update an item, we need to check if the head pointer is pointing to the item. If so, we need to move the head pointer correctly and smartly. In this section, we mainly introduce the control method of concurrent access to solve the concurrency problem in HotRing. In order to achieve high access concurrency and ensure high throughput, we have implemented a complete set of lock-free designs, which has been rigorously introduced by previous work [19, 50]. The atomic CAS operation is used to ensure that two threads will not modify the same *Next Item Address* simultaneously. If multiple threads are trying to update the same *Next Item Address*, only one thread succeeds and others fail. Failed threads have to re-execute their operations.



(a) RCU update & Insert

(b) RCU update & RCU update

(c) RCU update & Delete

Figure 8: Different concurrency issues that involve RCU operations.

Read. HotRing scans the collision ring to search for an item with the target key as described in Sec. 3.1. No additional operations are required to ensure correctness of read operations. Therefore, the read operations are completely lock-free.

Insertion. The insertion create a new item (e.g., item C in Figure 8(a)), and modify the preceding item's *Next Item Address* . Two concurrent insertions may compete for the same *Next Item Address*. The CAS ensures that only one succeeds and the other has to retry.

Update. We design two update strategies for different value sizes. The in-place-update operation (for 8-byte values) does not affect other operations, which is guaranteed through CAS. However, the RCU operation (for longer values) needs to create a new item, which challenges the concurrency of other operations. Taking the RCU update & Insert as an example in Figure 8(a): one thread is trying to insert the item C by modifying the *Next Item Address* of the item B, and another thread is trying to update the B with B′ concurrently. Operations of both threads will succeed since they modify different pointers with CAS. However, since item B is not visible to the ring, even though the insert operation for item C has succeeded, it cannot be accessed subsequently and lead to incorrectness. The same problem exists in Figure 8(b). In order to solve the problem, HotRing uses the *Occupied* bit (as shown in Figure 6(b)) to ensure correctness. We perform the update operation in two steps. For example, in the case of Update & Insert: Firstly, the *Next Item Address* of item B that to be updated is atomically set as occupied. Once the *Occupied* bit is set, the insertion of Item C will fail and have to retry. Secondly, the *Next Item Address* of item A is atomically changed to item B′ and the *Occupied* bit for B′ is reset.

Deletion. The deletion is achieved by modifying the pointer to the deleted item to its next item. Therefore, it must be ensured that *Next Item Address* of the deleted item is not changed during the operation. Similarly, we utilize the *Occupied* bit to ensure correctness of concurrent operations. For the case of RCU update & Delete as shown in Figure 8(c), the update for item D is processed by updating the forward item B's pointer, while item B is currently being deleted. The updated item D′ cannot be traversed correctly, resulting in data miss. If the *Occupied* bit of item B is set for deletion, the update for item D will fail to modify item B's *Next Item Address* and have to retry. Once the deletion of item B is completed, the update operation can be successfully executed.

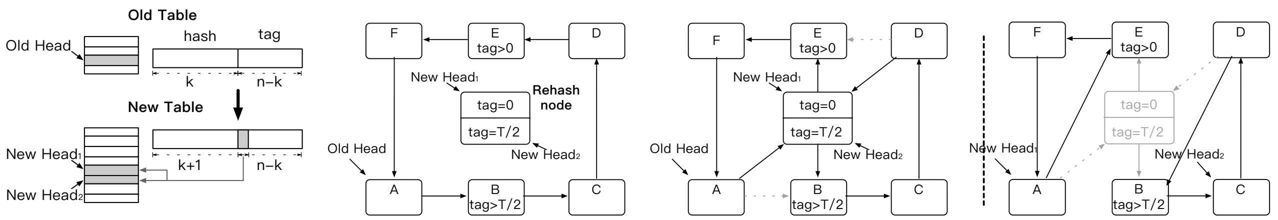
Head Pointer Movement. The movement of the head pointer is a special action in HotRing. In order to ensure the correctness of the head pointer movement with other operations (especially for update and deletion), we need additional management. There are two major problems that need to be addressed: (1) how to handle the concurrency of normal operations and the head pointer movement caused by identification strategies? (2) how to handle the head pointer movement, caused by updating or deleting of the head item?

For the head pointer movement caused by the identification strategies, we also use the *Occupied* bit to ensure correctness. When moving the head pointer to a new item, we set its *Occupied* bit to ensure that the item will not be updated or deleted during the movement. For head item update, HotRing moves the head pointer to new version of this item. Before moving the head pointer, we need to ensure that the new version item will not be changed (i.e., updated or deleted) by other threads. Therefore, when updating the item, HotRing sets the *Occupied* bit of the new version item first, until the movement is completed. For head item deletion, HotRing needs to occupy not only the item that is ready to be deleted, but also its next item. Because if the next item is not occupied during the deletion operation, the next node may have been changed, which makes the head pointer points to an invalid item.

3.4 Lock-free Rehash

As new data arrives from insertions, the number of collision items in a ring continues to increase, resulting in traversing more items per access. In this case, the performance of KVSes will be seriously degraded. We propose in HotRing a lock-free rehash strategy that allows for flexible rehash as data volumes increase. The conventional rehash strategy is triggered by the load factor (i.e., average length of chain) of the hash table. However, this fails to consider the effect of hotspots, and hence is unsuitable for HotRing. In order to make the index adapt to the growth of hotspot items, HotRing uses the access overhead (i.e., average number of memory accesses to retrieve an item) to trigger the rehash. Our lock-free rehash strategy includes three steps:

Initialization. First of all, HotRing creates a backend rehash thread. The thread initializes the new hash table that is twice the size of the old one, by sharing the highest bit of the tag. As shown in Figure 9(a), There is an old head pointer in an *Old Table*'s bucket, and there are two new head pointers in the *New Table*' correspondingly. The number of bits required for hash is expanded from k to $k+1$. HotRing divides data based on the tag range. Assuming that the hash value has n bits, and the tag range is $[0,T)$ $(T = 2^{(n-k)})$, two new head pointers manage items from $[0,T/2)$ and $[T/2,T)$



(a) Hash table (b) Initialization (c) Split (d) Deletion

Figure 9: The lock-free rehash strategy (The dotted line between (c) and (d) represents a transition period before deletion).

respectively. Meanwhile, the rehash thread creates a *rehash node* consisting of two child rehash items, which correspond to two new head pointers respectively. Each rehash item has the same format as data item, except that no valid KV pair is stored. HotRing identifies rehash items by the *Rehash* bit in each item. In the initialization phase, the tags of two child rehash items are set differently. As shown in Figure 9(b), the corresponding rehash items set the tags to 0 and T/2, respectively.

Split. In the split phase, The rehash thread splits the ring by inserting two rehash items into it. As shown in Figure 9(c), rehash items are inserted before item B and item E respectively, becoming the boundaries of the tag range to divide the ring. When two insertion operations complete, the *New Table* is made active. After that, subsequent accesses (from *New Table*) need to select the corresponding head pointer by comparing the tags, while previous accesses (from *Old Table*) proceed by identifying the *rehash node*. All data can be accessed correctly without effecting concurrent reads and writes. Until now, accesses to items are logically divided into two paths. When we look up a target item, at most half of the ring needs to be scanned. For example, the traversal path for accessing item F is Head$_1$ → E → F.

Deletion. In this phase, the rehash thread delete the *rehash nodes* (as shown in Figure 9(d)). Before that, the rehash thread have to maintain a transition period to ensure that all accesses initiated from the *Old Table* have finished, such as the grace period for read-copy-update synchronization primitive [13]. When all accesses end, the rehash thread can safely delete the *Old Table* and then *rehash nodes*. Note that the transition period only blocks the rehash thread, but not access threads.

4 Evaluation

In this section, we evaluate the performance of HotRing using real-workload-based benchmarks. In particular, we compare the throughput and scalability of HotRing with lock-free chain-based hash and other baseline systems. We also provide detailed evaluations to demonstrate the effectiveness of major designs adopted by HotRing.

4.1 Experimental Setup

Environment. We run experiments on a machine consisting of two Intel(R) Xeon(R) CPU E5-2682 v4 with 2.50GHz processors. They have 2 sockets, each with 16 cores (64 hyperthreads in total). The machine has 256GB RAM capacity, and runs CentOS 7.4 OS with Linux 3.10 kernel. To achieve better performance, we bind each thread to the corresponding core. Table 1 summarizes the detailed hardware configuration of the machine.

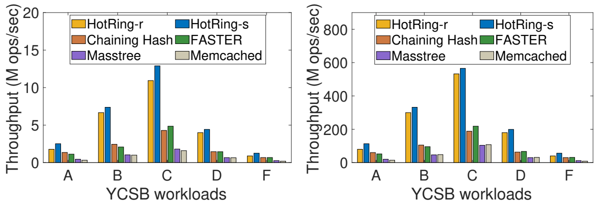
Table 1: Experimental environment.

CPU	2.50GHz Intel Xeon(R) E5-2682 v4 * 2
L2 cache	256KB (512 * 8 way)
L3 cache	40MB (32768 * 20 way)
Cache Alignment	64B
Main Memory	32GB 2133MHz DDR4 DRAM * 8

Table 2: Hotspot access ratio with different hotspot definition (i.e., top α of hottest items) and zipfian parameter (θ).

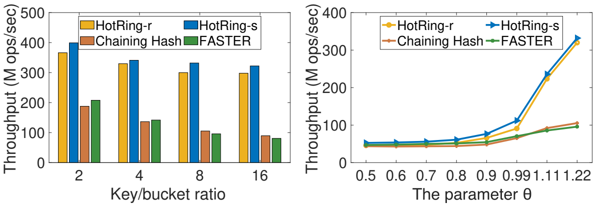
α θ	1%	10%	20%	30%	40%	50%
0.5	9.9%	31.6%	44.7%	57.7%	63.2%	70.7%
0.7	24.9%	50.0%	61.6%	71.9%	75.9%	81.2%
0.9	57.3%	76.2%	82.2%	86.9%	89.9%	92.2%
0.99	75.1%	87.4%	91.2%	93.4%	94.9%	96.2%
1.11	91.7%	96.4%	97.6%	98.2%	98.7%	99.0%
1.22	97.8%	99.2%	99.5%	99.6%	99.7%	99.8%

Workloads. We conduct experiments using the YCSB core workloads [10], except workload E that involves scan operations. For each item (i.e., key-value pair), we set the key size to be 8 bytes, and the value size to be 8 and 100 bytes for in-place-update and read-copy-update (RCU) respectively. In each test, the number of loaded keys is fixed to 250 millions, and the key-bucket ratio (i.e., the number of keys divided by the number of buckets) varies to control average length of collision chains. In addition, we tune the zipfian distribution parameter θ in YCSB to generate workloads that simulate daily and extreme hotspot scenarios. Table 2 shows the ratio of hotspot accesses with different hotspot definitions and skewness. Recall that Figure 1 shows the workload distributions in our production environments. We observe that θ falls in [0.9, 0.99] for daily scenarios, and in [1, 1.22] for extreme cases. Hence, we choose 0.99 and 1.22 for θ as representa-



(a) Single thread (b) 64 threads

Figure 10: Throughput of HotRing and other systems.



(a) Impact of chain length (b) Impact of θ

Figure 11: Performance impact of collision chain length and access skewness.

tives.

Baselines. In order to better demonstrate the advantage of hotspot-aware designs in HotRing, we implement a lock-free chain-based hash index as a baseline (Chaining Hash). It is modified from the hash structure in Memcached, and uses the CAS primitive to insert new items to the head of collision chains. We also compare to other KVS systems: the C++ version of FASTER [7], in which we ensure all data resides in memory; Masstree [30], a high-performance in-memory range index that is a representative KVS with non-hash indexes. In addition, the lock-based Memcached [44] is also included as a reference.

Note that the memory footprint of an index structure greatly affects system performance. In order to have fair comparisons, we strictly make the memory consumption of indexes the same for all approaches. In each test, if not otherwise specified, we use following default settings: 64 threads, 8-byte value payloads, workload B of YCSB, θ set to 1.22, and key-bucket ratio set to 8 (for HotRing).

4.2 Comparison to Existing Systems

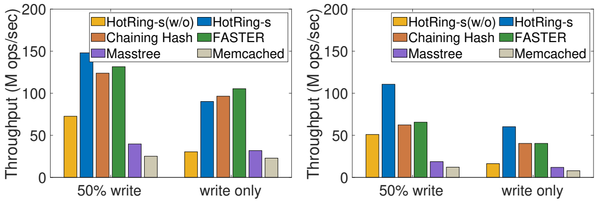
We evaluate HotRing against four baselines introduced above, i.e., Chaining Hash, FASTER, Masstree and Memcached, which are all high-performance KVS implementations.

Overall performance. Figure 10 shows the overall system throughput of all approaches on different YCSB workloads. We run two HotRing variants with distinct hotspot identification strategies: HotRing-r adopts random movement strategy, and HotRing-s adopts sampling statistics strategy. Compared to other systems, HotRing achieves better performance in throughput under all workloads, especially for workloads B and C. HotRing-s outperforms other approaches by 2.10× - 7.75×. It achieves 12.90M ops/sec with a single thread and 565.36M ops/sec with 64 threads, which implies promising scalability. Besides, HotRing also keeps advantages for insertion operations. For workloads D and F of YCSB (with massive insertions), HotRing-s outperforms other approaches by 1.81× - 6.46×. This is because the ordered-ring structure speeds up item location by early termination, while the tag field reduces the cost of sorting. Though the hotspot identification of HotRing-r is less accurate than HotRing-s (about 7% worse), its overall performance is still significantly improved compared to other systems.

Collision chaining length. Figure 11(a) shows the throughput of different approaches when we vary the length of collision chains. We tune the key-bucket ratio from 2 to 16, which means that the conflicts in the hash table become more intense. We can see that Chaining Hash and FASTER have good performance when key-bucket ratio is 2. This is because when the collision chain is short, the memory access overhead for hotspot items is relatively minor, making the effect of cache more significant (especially for FASTER). However, as the length of the collision chain increases, frequent accesses to hotspot items drop the performance of Chaining Hash and FASTER seriously. In contrast, HotRing retains satisfactory performance even for long chains. In particular, when the key-bucket ratio is 2, its read throughput is 1.93× and 1.31× compared to that of Chaining Hash and FASTER. When the ratio becomes 16, the performance gap increases to 4.02× and 3.91× respectively. This is because HotRing puts hot items close to the head, so that less memory accesses are required. This design is more cache-friendly, where only head pointers and hotspot items need be cached, rendering higher performance. Therefore, we conclude that HotRing has better performance and scalability due to its hotspot-aware designs.

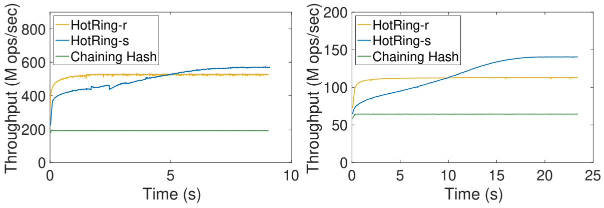
Access skewness. Figure 11(b) shows the throughput of different approaches when the zipfian parameter θ varies. We tune θ from 0.5 to 1.22, which means that the hotspot issue in workloads become more severe. As can be seen, the performance improvement in both Chaining Hash and FASTER is not obvious as θ increases, since they lack hot-aware considerations. In contrast, the performance of HotRing significantly improves as θ increases, especially when θ is greater than 0.8. Even when θ is in $[0.5, 0.8]$ range, where hotspot issue is minor, HotRing-s still achieves better performance than others. This is because HotRing-s is able to handle the case of multiple hotspots in the collision ring. When there are multiple items with similar access frequencies, HotRing-s can find the best head pointer position to achieve optimal performance (§3.2.2). However in this case, HotRing-r fails to choose the optimal head pointer position, leading to frequently-triggered pointer movements.

RCU operation. In order to show the performance of RCU more prominently, we use YCSB to generate write-intensive workloads with 100-byte value payloads (both 50% write and write only). Figure 12 shows the throughput of different approaches when RCU operations are involved. In this test, we



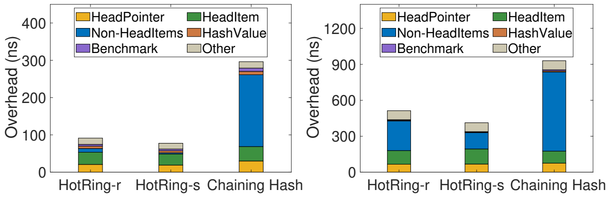
(a) key/bucket = 2 (b) key/bucket = 8

Figure 12: Performance of RCU operations.



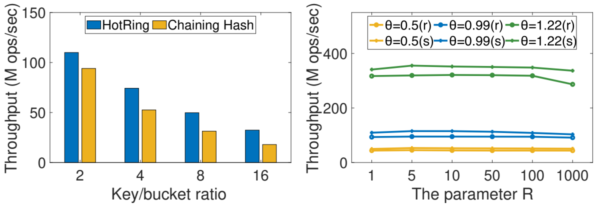
(a) θ = 1.22 (b) θ = 0.99

Figure 14: Reaction delay.



(a) θ = 1.22 (b) θ = 0.99

Figure 13: Break-down cost.



(a) 100% read miss (b) Impact of *R*

Figure 15: Performance impact of read miss and *R*.

demonstrate the need for special processing of RCU operation in HotRing (§3.2.2). In particular, HotRing-s indicates that the sampling statistics strategy will increase the forward item counter instead when an item is updated by RCU. HotRing-s(w/o) represents the strategy without distinguishing RCU operations. Firstly, HotRing-s(w/o) has poor performance in all cases, even worse than Chaining Hash and FASTER. This is because HotRing-s(w/o) needs to traverse the entire collision ring for completing an RCU operation on the hot item. However in HotRing-s, the optimized hotspot counting strategy significantly improves RCU performance. Note that when key-bucket ratio equals 2, the performance of HotRing-s is slightly slower than that of FASTER. This is because the hotspot item requires RCU operation at the second slot pointed by the header point, where one additional memory access is needed. Furthermore, it involves one extra CAS operation (on *Occupied* bit) to complete the RCU operation. As the number of collision items keeps increasing, above issues will be greatly mitigated. For example, when key-bucket ratio reaches 8, the throughput of HotRing-s is 1.32× better than Chaining Hash and FASTER.

4.3 Investigation of Detailed Designs

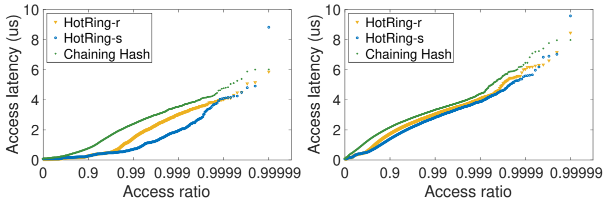
In this section, we compare HotRing with the conventional chaining hash, in order to investigate the advantages of hotspot-aware designs.

Break-down cost. We collects the break-down cost of different functions involved during workload execution. Figure 13 shows the average break-down cost for a single read access in HotRing and Chaining Hash (where key-bucket ratio is 8). In this figure, *HeadPointer* is the cost to locate the head pointer of the corresponding collision ring or chain; *HeadItem* is the cost to access the head item; *Non-HeadItem* is the cost to access other items; *HashValue* is the cost due to the hash calculation; *Benchmark* is the cost to read and interpret the workload command; and *Other* is the cost from the system kernel. As can be observed, the cost in Chaining Hash is mainly dominated by *Non-HeadItem* accesses, which is about 193ns/660ns when θ = 1.22/0.99. This indicates that hotspot items in chain-based hash tend to be evenly distributed in the chain, increasing the access cost significantly. In contrast, HotRing-r and HotRing-s significantly reduce the *Non-HeadItem* cost. Especially for HotRing-s, the *Non-HeadItem* cost is about 10ns/136ns when θ = 1.22/0.99. Since the proportion of *Non-HeadItem* is negatively related to the hotspot identification accuracy, this implies that HotRing-s has higher hotspot identification accuracy where more hotspot items are detected and placed at the head.

Reaction delay. Reaction delay is one of important metrics for measuring hotspot identification strategies. Figure 14 shows the throughput trends over time after hotspots have shifted (workload C). We can observe that HotRing-r has faster reaction than HotRing-s, which only takes less than 2 seconds to reach stable state. However, its peak throughput is much lower due to its inaccurate hotspot detection. The throughput of Chaining hash is not affected since it lacks hotspot-awareness.

Read miss. We evaluate the throughput of both approaches for handling read misses as shown in Figure 15(a). As can be seen, the performance gap between HotRing and Chaining Hash expands with the increase of chain length. In particular, HotRing achieves 1.17× improvement when the key-bucket ratio is 2, and 1.80× when the ratio reaches 16. This is because that HotRing only needs to compare with half of items in average for a lookup (as shown in Figure 5), while Chaining Hash accesses all items in the chain.

Parameter *R*. Recall that the choice of parameter *R* affects the frequency of head pointer movement (§3.2). When *R* is small, the hotspot identification has less reaction delay, but results in more frequent (and invalid) head pointer movements.



(a) θ = 1.22 (b) θ = 0.99

Figure 16: Tail latency.

Figure 15(b) shows the impact of *R* on overall throughput in different scenarios. It can be observed that the performance get slightly worse when *R* is either too small (due to overheads from hotspot identification) or too large (due to delayed handles of hotspot shifts). In practice, we set *R* to 5 for balanced consideration and better throughput.

Tail latency. HotRing-s requires statistical sampling and the last thread during a sampling process needs to calculate access frequencies to find the best position for the head pointer. Hence, there might exist long-tail accesses due to such additional computations. Figure 16 shows the latency distribution of 100 thousands accesses. When θ = 1.22, the 99-percentile response time is about 2μs, but there are long-tail accesses requiring 8.8μs. It is similar when θ = 0.99, where 99-percentile response time is 3μs and long-tail access time is 9.6μs. Note that the long-tail access is partially related to the simplification of our implementation choices, and can be further mitigated by moving additional computations to dedicated backend threads.

Lock-free rehash. Rehash is an important mechanism to ensure stable performance of growing hash tables. We construct following scenario to evaluate our lock-free rehash operation: in the initial state, the number of loaded keys is 250 millions and the key-bucket ratio is 8; then, we use a YCSB workload with 50% read (θ =1.22) and 50% insertion to simulate the continuous growth of hash tables. Figure 17 shows HotRing' performance over time when rehashes are conducted. In particular, I,T, and S represent the initialization, transition, and splitting phases of rehash, respectively. It can be observed that two consecutive rehash operations help to retain the throughput as data volume continuously grows. The short-term drops during rehash are attributed to the temporary lack of hotspot awareness when the new hash table starts to work.

5 Related Work

Many existing works focus on the design of index structures for key-value stores. Memcached [17] is a widely-used distributed key-value store, which is used by a large number of companies. However, its multi-threading performance is unsatisfactory, due to frequent competition of locks. Based on Memcached, there is plenty of work with outstanding contributions in the literature. By implementing lock-free designs and cache-friendly optimizations, they achieve higher concurrency and throughput [8, 15, 17, 24, 33, 43]. In particular, FASTER [8] is one of the state-of-the-art implementations with lock-free designs. For hotspot awareness, Splay trees [47] is an inspiring work that adapts its structure to optimize for recently accessed items. However, its lock-based design makes it unsuitable for highly concurrent scenarios.

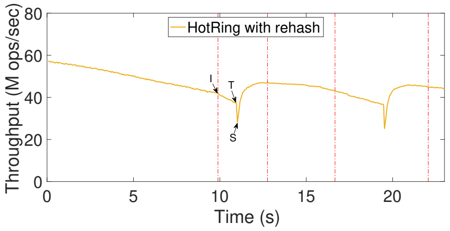


Figure 17: Rehash performance.

Besides, there are many works on the better integration of system designs with emerging hardwares, such as FPGAs [6,34], RDMA-enabled NICs [28,41,45], GPU [22], low-overhead user-level implementations for TCP [25], and InfiniBand with hardware-level reliable datagrams [41]. Meanwhile, in order to provide fast memory allocation (for insertion and deletion), many protocols also leverage lock-free memory management methods [38, 39], which can be used to prevent the ABA problem [48]. Note that these optimization for hardware and memory management are orthogonal to the design of index structures, and we can also adopt these ideas to further improve HotRing's performance.

6 Conclusion and Future Work

In real-world deployment of KVSes, the hotspot issue is common and becomes more serious recently. For example, in order to provide a highly-available service, Alibaba's NoSQL product *Tair* [1] has to allocate more machines than necessary to handle sudden occurrences of hotspots. Hence, we explore opportunities and challenges for designing hotspot-aware in-memory KVS. Based on discovered insights, we propose a hash index called HotRing that is optimized for massively concurrent accesses to a small portion of items. It dynamically adapts to the shift of hotspots by pointing bucket heads to frequently accessed items. In most cases, hot items can be retrieved within two memory accesses. HotRing comprehensively adopts lock-free structures in its design, for both common hash operations and HotRing-specific operations. The extensive experiments show that our approach is able to achieve 2.58× throughput improvement compared to other in-memory KVSes on highly skewed workloads. Now HotRing has become a subcomponent of *Tair*, extensively used in Alibaba Group.

At present, HotRing-r is designed for single hotspot on each chain, while HotRing-s also handles multiple hotspots. In most cases, we can mitigate the multiple hotspot issue by reducing chaining length via rehash. For some extreme cases where it fails to handle, we leave the exploration of a suitable solution as future work.

References

[1] Tair - NoSQL product of Alibaba Group. https://help.aliyun.com/document_detail/145957.html, 2019.

[2] Juan Alemany and Edward W Felten. Performance Issues in Non-blocking Synchronization on Shared-memory Multiprocessors. In *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing (PODC 1992)*, pages 125–134. ACM, 1992.

[3] Amazon. Amazon ElastiCache. https://aws.amazon.com/cn/elasticache, 2014.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. 40(1):53–64, 2012.

[5] Brian N Bershad. Practical Considerations for Lock-free Concurrent Objects. 1991.

[6] Michaela Blott, Kimon Karras, Ling Liu, Kees A Vissers, Jeremia Bär, and Zsolt István. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *HotCloud*, 2013.

[7] Badrish Chandramouli. microsoft FASTER. https://github.com/microsoft/FASTER, 2018.

[8] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD 2018)*, pages 275–290. ACM, 2018.

[9] Yue Cheng, Aayush Gupta, and Ali R Butt. An In-memory Object Caching Framework with Adaptive Load Balancing. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys 2015)*, page 4. ACM, 2015.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC 2010)*, pages 143–154. ACM, 2010.

[11] Frank Dabek, M Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP 2001)*, volume 35, pages 202–215. ACM, 2001.

[12] Dhananjay Ragade. David Raccah. Linkedin Memcached. https://www.oracle.com/technetwork/server-storage/ts-4696-159286.pdf, 2014.

[13] Mathieu Desnoyers, Paul E McKenney, Alan S Stern, Michel R Dagenais, and Jonathan Walpole. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012.

[14] Facebook. RocksDB. https://rocksdb.org.

[15] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2013)*, volume 13, pages 371–384, 2013.

[16] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Small cache, big effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC 2011)*, page 23. ACM, 2011.

[17] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux journal*, 2004(124):5, 2004.

[18] Sanjay Ghemawat and Jeff Dean. LevelDB. https://github.com/google/leveldb, 2011.

[19] Timothy L Harris. A Pragmatic Implementation of Non-blocking Linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC 2001)*, pages 300–314. Springer, 2001.

[20] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of HDFS Under HBase: A Book Messages Case Study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 199–212, 2014.

[21] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture (ISCA 1993)*, volume 21, pages 289–300. ACM, 1993.

[22] Tayler H Hetherington, Mike O'Connor, and Tor M Aamodt. MemcachedGPU: Scaling-up Scale-out Key-value Stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC 2015)*, pages 43–57. ACM, 2015.

[23] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert van Renesse. Characterizing Load Imbalance in Real-world Networked Caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets 2014)*, page 8. ACM, 2014.

[24] Intel. Intel Threading Building Blocks. https://www.threadingbuildingblocks.org, 2017.

[25] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)*, volume 14, pages 489–502, 2014.

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing Key-Value Stores with Fast In-Network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017)*, pages 121–136. ACM, 2017.

[27] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *Proceedings of the 11th international conference on World Wide Web (WWW 2002)*, pages 293–304. ACM, 2002.

[28] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA Efficiently for Key-Value Services. *ACM SIGCOMM 2014 Conference (SIGCOMM 2014)*, 44(4):295–306, 2015.

[29] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on The World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing (STOC 1997)*, volume 97, pages 654–663, 1997.

[30] Eddie Kohler. Masstree. https://github.com/kohler/masstree-beta, 2013.

[31] Redis lab. Redis. https://https://redis.io/, 2017.

[32] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. Be fast, Cheap and in Control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, 2016.

[33] Hyeontaek Lim, Donsu Han, David G Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)*, pages 429–444, 2014.

[34] Kevin Lim, David Meisner, Ali G Saidi, Parthasarathy Ranganathan, and Thomas F Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA 2013)*, volume 41, pages 36–47. ACM, 2013.

[35] Greg Linden. Akamai Online Retail Performance Report: Milliseconds are Critical. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html, 2006.

[36] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, 2019.

[37] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the Seventh EuroSys Conference on Computer Systems, (EuroSys 2012)*, pages 183–196. ACM, 2012.

[38] Maged M Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2002)*, pages 73–82. ACM, 2002.

[39] Maged M Michael. Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing (PODC 2002)*, pages 21–30. ACM, 2002.

[40] Maged M Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[41] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC 2013)*, pages 103–114, 2013.

[42] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2013)*, volume 13, pages 385–398, 2013.

[43] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, pages 29–41. ACM, 2011.

[44] Matthew Shafer. Memcached. https://github.com/memcached/memcached, 2012.

[45] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC 2012)*, pages 347–353, 2012.

[46] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[47] Robert Endre Tarjan. Sequential Access in Splay Trees Takes Linear Time. *Combinatorica*, 5(4):367–378, 1985.

[48] R Kent Treiber. *Systems Programming: Coping With Parallelism*. New York: International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[49] Twitter. Twitter Memcached. https://github.com/twitter/twemcache, 2014.

[50] John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC 1995)*, volume 95, pages 214–222. Citeseer, 1995.