C-Hint: An Effective and Reliable Cache Management for RDMA-Accelerated Key-Value Stores

Yandong Wang, Xiaoqiao Meng, Li Zhang, Jian Tan

IBM T.J Watson Research, New York, USA

yandong, xmeng, zhangli, tanji@us.ibm.com

Abstract

Recently, many in-memory key-value stores have started using a High-Performance network protocol, *Remote Direct Memory Access* (RDMA), to provision ultra-low latency access services. Among various solutions, previous studies have recognized that leveraging RDMA Read to optimize GET operations and continuing using message passing for other requests can offer tremendous performance improvement while avoiding read-write races. However, although such a design can utilize the power of RDMA when there is sufficient memory space, it has also raised new challenges on the cache management that do not exist in traditional key-value stores. First, RDMA Read deprives servers of the awareness of the read operations. Therefore, how to track popular items and make replacement decisions at the server side becomes a critical issue. Second, without the access knowledge from the clients, new approaches are needed for servers to efficiently and reliably reclaim the resources. Lastly, the remote pointers hold by clients to conduct RDMA are highly susceptible to the evictions made by remote servers. Thus, any replacement algorithm that solely considers the server-side hit ratio is insufficient and can cause severe underutilization of RDMA.

In this work, we present C-Hint, an efficient and reliable cache management system that is designed to address the above three challenges. Its basic mechanism relies on a holistic design of the servers and the clients to orchestrate the access history information. Specifically, it consists of several techniques, including lease-based key-value management, popularity differentiated lease assignment, as well as several optimizations to achieve efficient and reliable cache management. We have implemented and integrated C-Hint into an in-house memory-resident key-value store, named HydraDB, and systematically evaluated its performance on an InfiniBand cluster with different workloads. Our experiment results demonstrate that C-Hint can efficiently outperform several other alternate solutions and deliver both high hit rate and low latency performance.

SOCC '14, November 03 - 05 2014, Seattle, WA, USA.
Copyright © 2014 ACM 978-1-4503-3252-1/14/11. . . $15.00.
http://dx.doi.org/10.1145/2670979.2671002

Keywords RDMA, Key-Value Stores, Cache Management

1. Introduction

Distributed memory-resident key-value store has been a critical cornerstone for Internet services to deliver low-latency access to large-scale data sets. In recent years, using high performance interconnects, such as InfiniBand [15], found in High-Performance Computing (HPC) clusters to further push down the latency bound has attracted a lot of attention from both industry and academia [10, 20–22, 25, 27, 28]. While, in this paper, we focus on a different angle to explore how to effectively and reliably tackle a cache management challenge that arises recently when a high-performance network protocol, named Remote Direct Memory Access (RDMA), is leveraged to optimize in-memory key-value stores.

Notwithstanding the introduction of many network protocols, such as IP over InfiniBand (IPoIB) [16] and Send/Recv Verbs [15], to facilitate the incorporation of fast networks, RDMA (*a.k.a*, One-sided operation) is still the ultimate design choice to fully exploit the performance from underlying networks due to its capability to deliver the lowest latency, highest bandwidth, and zero CPU overhead. The power of RDMA lies in the fact that it allows a client to directly read from or write to a memory region inside another machine without consuming remote-side CPU power and involving intermediate memory copies, *i.e.*, kernel-bypass.

Given that production key-value workloads are commonly dominated by read operations, many previous studies [25, 28] have attempted to leverage RDMA Read to highly optimize *GET* requests, while resorting to message passing (*e.g.*, Send/Recv Verbs) for the rest types of operations to avoid read-write races. However, although such a design is effective in capturing the benefit of RDMA when there is sufficient memory space, it has raised many challenges on the cache management that do not exist in traditional memory-resident key-value systems.

Due to the unawareness of the RDMA operations, servers are no longer able to keep track of the popularity of cached items; thereby severe amount of false evictions can occur. For instance, a highly popular item tends to be accessed more frequently by RDMA Read. However, without those access knowledge on the servers, traditional cache replacement algorithms, such as LRU or CLOCK used by Memcached [2, 12] and Redis [3], are likely to rank it as one of the least recently used items and evict it when it remains hot.

In addition, to allow clients to access a key-value pair with RDMA Read, the server needs to dedicate a memory area for the item and inform the clients about the corresponding memory address. However, when many clients are referencing the same memory region on the server side, what is an efficient and reliable approach for servers to carry out the resource reclamation? Any premature reclamation can cause clients to fetch incorrect data and relying solely on race inhibition technique is unable to address such issue.

More importantly, the status of above memory address is highly susceptible to the server-side eviction decision that also has to invalidate the address cached by the clients; otherwise using stale information can also lead to unpredictable behavior. However, frequent address invalidation is highly undesirable since it reduces the opportunities to use RDMA Read. Therefore, an immediate design implication is that any replacement algorithm that only considers hit ratio on the server side is insufficient and can impair the utilization of RDMA Read.

To tackle above three major challenges raised by using RDMA, in this paper, we present C-Hint, an efficient and reliable cache management system for in-memory key-value stores that leverage RDMA Read. C-Hint synthesizes a collection of ideas in order to achieve high hit ratios of cached items without sacrificing the performance advantage offered by RDMA Read. Meanwhile, it aims to inhibit potential incorrect behaviors that can be incurred by access-unawareness. Overall, the key ideas of C-Hint design are threefold. First, it takes advantage of the opportunity to co-design the server and client to orchestrate the access history information among different roles. Second, it has extended the concept of lease to reliably manage key-value pairs and corresponding memory regions. Third, by introducing a popularity differentiated lease assignment, C-Hint is able to retain the high-performance offered by RDMA Read.

We have implemented and integrated C-Hint into an in-house memory-resident key-value store, called HydraDB, and systematically evaluated its performance on an InfiniBand cluster with multiple workloads. Because to the best of our knowledge, C-Hint is the first cache management system targeting RDMA-Accelerated key-value stores, there is a lack of existing solutions we can compare to. Therefore, we have also introduced several alternate approaches to gain better understanding of the performance of C-Hint. Our experiment results demonstrate that C-Hint can efficiently outperform other alternatives, delivering both higher hit ratios and lower average latency.

The rest of the paper is organized as follows. In section 2, we provide a brief background of HydraDB, then we describe the challenges associated with leveraging RDMA Read in details. Section 3 presents the design details of C-Hint. Section 4 further provides some implementation details. Section 5 describes our experimental methodology and shows the evaluation results. Section 6 discusses the related work. Eventually we conclude the paper in section 7.

2. Background and Challenges

This work is based on HydraDB, a memory-resident and RDMA-powered high-performance key-value store introduced by IBM. HydraDB has encompassed many optimizations, including NUMA-awareness, lock-free data structures, load balancing enhancement *etc*, for achieving the ultimate performance. A full description of HydraDB deserves another paper. Our goal is to shed a light on the issues and challenges pertaining to the cache management raised by leveraging one-sided RDMA operations. So in this section, we briefly describe the high-level design choices made by HydraDB, while highlighting its RDMA capability. Then we present the challenges and the characteristics of the production workloads that inform our work.

2.1 HydraDB: A RDMA-Accelerated Key-Value Store

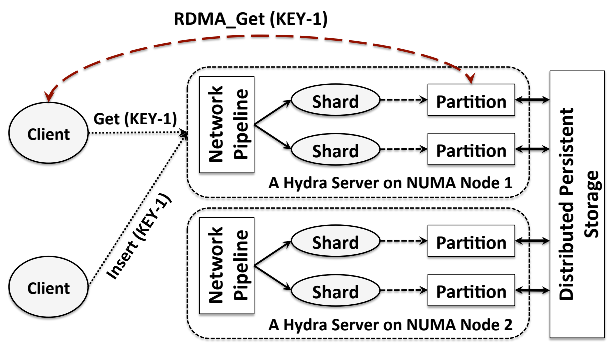


Figure 1: Architecture of HydraDB on a multi-core system.

HydraDB (hydra) is an in-memory key-value store that provides strong consistency guarantee with configurable persistence support. It exposes a rich set of APIs, including conventional operations, like GET, SET, REMOVE *etc* and hash-based operations, such as HSet, HGet *etc*.

When deployed on a machine with multiple cores, HydraDB instantiates several hydra server instances in accordance with the number of NUMA nodes as shown in Figure 1. Within each instance, hydra partitions keys among multiple Shard processes, one Shard per CPU core. Each

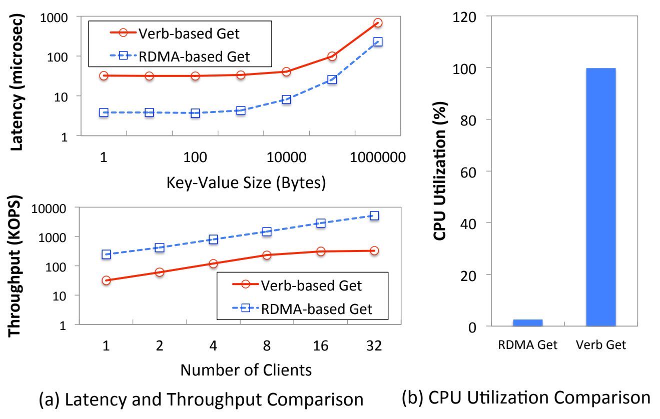


Figure 2: Performance comparison between RDMA-based GET with Verb-based approach.

Shard manages one or more partitions. A network pipeline, pinned to other cores on the same NUMA node, is introduced inside each server to handle network functionalities, such as connection establishment, data transfer *etc*. Meanwhile, to provision persistence, all the servers can be configured to be attached to a shared distributed persistent storage.

Given the superior performance benefits of RDMA, such as ultra-low latency, high throughput, and zero CPU overhead *etc*, as demonstrated by previous studies [21, 25, 28]. HydraDB strives to exploit its performance across the design. In particular, HydraDB leverages one-sided RDMA Read to allow clients to directly access the remote memory within the servers to highly optimize GET operations that dominate many production workloads [5, 7], while using two-sided Verb-based Send/Recv [25] for other key-value APIs, including SET, REMOVE *etc*.

When a client accesses an item from the store at the first time, the operation, either SET or GET, is routed to the corresponding Shard according to the hash of the key. Among the returning information, a *remote pointer* (<address, length, rdma-key>) that indicates the location of the key-value pair inside server is included and cached at the client side. With such pointer, RDMA Read can be applied for future GET operations of the same key. While, due to the unawareness of RDMA Read conducted by clients, race between RDMA reads and server-side updates can lead to incorrect data fetching. Many previous studies have recognized this issue and proposed several solutions to inhibit such race via resorting to either checksum [25] or cache line versioning [10]. Therefore, we omit further discussion on this topic in this paper. The current HydraDB leverages key-value versioning to alleviate the read-write races.

Figure 2 (a) exhibits the latency comparison between RDMA-based GET with Verb-based approach. Comparing to Verbs, RDMA Read substantially reduces the fetch latency by up to one order of magnitude for key-value pairs below 1KB. In addition, when the workloads exhibit highly skewed GET/SET ratios, by leveraging RDMA Read, a single Shard process can sufficiently process all the requests without being saturated, and deliver significantly higher throughput than Verb-based approach. Figure 2 (a) also shows the throughput comparison via using a 99% GET+1% SET workload (24Byte as the key-value size). As shown in the figure, RDMA-based approach displays an ideal scalability and substantially outperforms Verb-based approach, yielding an improvement of up to 16.2× for 32 clients case. Moreover, RDMA Read incurs zero CPU overhead on the server sides, making more rooms for servers to conduct processing and achieve energy efficiency. Figure 2 (b) demonstrates such advantage by illustrating the average CPU utilization comparison between the two approaches when the workload consists of 100% GET (100Byte as the key-value size). In sharp contrast to the Verb case, in which CPU sustains about 100% utilization, RDMA Read leads to close to 0% utilization on the servers.

Table 1: Overhead of calculating checksum.

Size	CRC64	RDMA Get	Verb Insert
1 KB	5.2 μs	4.3 μs	33.8 μs
10 KB	40.3 μs	8.1 μs	45.4 μs
100 KB	641.1 μs	26 μs	259.2 μs
1000 KB	5122.7 μs	230.4 μs	1539.5 μs

2.2 Challenges associated with Access-Unawareness

Despite the great performance benefit, leveraging one-sided RDMA Read also introduces new challenges that do not exist in conventional key-value stores. In particular, it substantially complicates the design of cache management, which is a critical component of memory-resident key-value stores.

Challenges in popularity tracking: Due to CPU bypass, hydra server is unaware of the data accesses conducted by RDMA Reads, thereby disabling the servers to keep track of the access frequency and recency of key-value pairs. In addition, such access-unawareness renders traditional cache replacement algorithms, such as LRU or CLOCK, and their variants *etc*, impossible to be accurately implemented on the server sides, leaving eviction mechanism in dilemma when the system is under heavy memory pressure.

A naive approach that aims to faithfully memorize access history via informing the servers about every key access after RDMA Read can significantly debilitate or even eliminate the benefit of RDMA. An alternate solution is to leverage an RDMA variant, called RDMA Read with immediate data (RDMA with IMM) [23] that interrupts CPU with a message after each RDMA operation. However, this approach also suffers from two major issues. First, its performance is significantly worse than that of raw RDMA as shown in [23]. Second, the message size must be less than 32 bits, half of the key hash, thereby limiting its practicality. Using Random Replacement (RR) is another tentative to address such issue. Though simple, RR cannot efficiently handle commonly observed skewed workloads in which small percent of items are significantly hotter than the rest.

Challenges in resource reclamation: When remote pointers are cached on the client sides, they carry implicit promises that the memory areas referenced by those pointers remain valid. However, a challenge is raised by such promise. When many clients are holding remote pointers for the server-side memory regions, when can servers reliably reclaim promised memory areas to guarantee data integrity? Since any premature resource reclamation can corrupt the data being accessed by RDMA Read, causing clients to retrieve garbage data that unfortunately passes the verification procedure. And we have observed that simply using race protection data structure is unable to address this issue. Though leveraging checksum as proposed by [25] for every access can reduce the probability of incorrect data fetching, it can incur substantial CPU overhead during item write and read for large key-value sizes as shown in Table 1.

The challenge lies in the fact that it is complex, if not impossible, for servers to maintain a counter for every key-value pair under access-unaware context to record active memory references in a distributed environment. A variety of factors, such as client failures, can lead to inaccurate counting. Another alternative is to force servers to broadcast every reclamation to all the clients. However, it is at the significant cost of performance. In addition, simply allowing clients to retain all the accessed remote pointers all the time can prevent servers from reallocating resources for capturing emerging popular items, leading to precipitous degradation of caching efficiency when the capacity is saturated.

Challenges in reconciling hierarchical caching: In HydraDB, the total hit ratio is determined not only by the verb-based Gets that servers are aware of, but also the RDMA Gets that contributes more than the former to the performance. Therefore, efficient utilization of client-side remote pointers is crucial given its critical role in allowing the RDMA Read. However, they are highly susceptible to the eviction decisions made by server-side replacement algorithm. Any eviction of the key-value pair also needs to invalidate the corresponding remote pointers, otherwise using stale cached information can cause unpredictable behavior as mentioned above. An immediate implication to our design is that any cache replacement algorithm that solely considers the hit ratios on the server side is insufficient and can decrease the chance of conducting RDMA.

However, many challenges exist around such issue. How shall servers invalidate remote pointers for evicted items? How to design a replacement algorithm to achieve both high hit ratios for items cached on the servers meanwhile avoiding frequently invalidating remote pointers so that HydraDB can maintain high-performance offered by RDMA?

2.3 Characteristics of Production Workloads

Before presenting our solutions to address above challenges, we briefly discuss the workloads that inform our design.

The characteristics of several production key-value workloads have been comprehensively analyzed in [5]. Previous studies [12, 14] have focused on the distribution of key-value sizes to enhance the hashing mechanism and request handling. While in this work, we concentrate more on the access patterns to design an efficient cache management system.

Skewed workloads are considerably common. Production workloads exhibit highly skewed access pattern, with small percentage of keys are accessed much more frequently than the rest. In the SYS workloads reported by [5], 25% of keys appear in above 90% of requests. In other cases, 50% of keys are only touched by less than 1% requests. In another workload analysis study [7], similar skewed access pattern occurs showing 10% videos receive 80% views with the rest 90% videos browsed by only a few requests. Therefore, it is critical for the cache management to capture those hot items, and retain them in memory.

However, in access-unaware HydraDB, a highly popular items are more likely to be accessed by RDMA Reads, leading servers to deem them as cold contradictorily.

Data preference is time correlated. Furthermore, for many workloads, certain items can be highly popular in a time range but not during the rest of the time as revealed by [5, 7]. For example, the newly added photos receive large amount of clicks, while the interest on them decays over time. Therefore, the caching system must dynamically adapt to evolving workloads and seize the aging of popularity.

3. Design of C-Hint

To tackle the challenges illustrated in section 2.2, we have introduced *C-Hint* that synthesizes a collection of ideas to manage caching systems for RDMA-Accelerated in-memory key-value stores, such as HydraDB. Overall, the goals C-Hint aims to accomplish are threefold: (1) To deliver sustainable high hit rates even when the cache capacity is fully saturated. (2) To ensure that the performance advantages offered by RDMA are not sacrificed. (3) To provision reliable resource management to eliminate potential errors caused by conflicting RDMA access and memory reclamation. Figure 3 depicts a high-level overview of the design of C-Hint, which spans over both hydra server and client sides. In the following sections, we firstly present the key idea of C-Hint, and then elaborate our fulfilling solutions.

3.1 Client-Assisted Cache Management

Without the knowledge of the *Get* operations issued by RDMA Reads, hydra servers alone are unable to keep track of the popularity of cached items. In this sense, C-Hint leverages the opportunity of co-designing the client and server, and seeks the assistance from hydra clients, designing them to periodically propagate partial access history of key-value pairs over the past time range to the server sides, so that each server can establish a global view of access patterns of local cached items to facilitate cache management decisions.

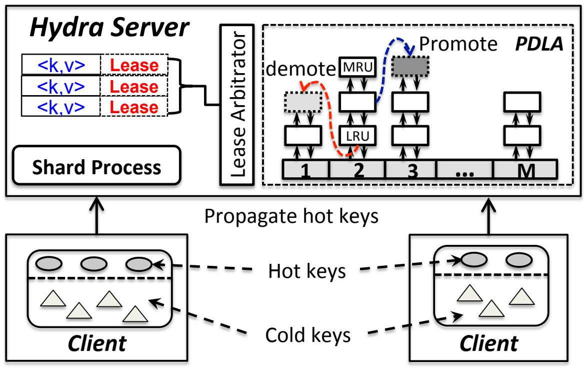


Figure 3: C-Hint synthesizes a set of ideas, including (1) client-assisted cache management, (2) lease-based key-value management, (3) popularity-differentiated lease assignment, to collectively manage the caching system.

However, forcing every client to faithfully propagate a large amount of access history can precipitously increase the queuing impact on the server sides delaying the process of incoming requests. Therefore, a critical issue to address involves which access history and how much information clients should propagate to avoid overloading the servers.

In addition, in order to maintain a global view of changing access patterns, what is an efficient mechanism to aggregate and merge the access records covering a past time range from many clients? Since any careless design of records aggregation can substantially consume CPU powers. Furthermore, assuming that server has obtained the access history from clients. Which cache replacement algorithms should server use to achieve high hit ratios for evolving workloads without excessive remote pointers invalidation?

In the following, we further unfold our design details via answering above questions.

3.2 Managing Key-Value Pairs via Leases

To maintain a global view of the access patterns of cached key-values on the server sides, we have extended the idea of *lease* to support cache management. In contrast to the lease concepts used in prior arts [11, 13] that mainly leverage leases to ensure consistency between cached data with that in backend storage. Our lease serves for three distinct purposes. When a client caches a remote pointer for future data access, the lease stands as a *pact* between server and the pointer holder. It grants the holder the privilege to conduct *Get* with RDMA read (but not update) and guarantees the availability of the data on the server side for a limited time range. Meanwhile, lease manifests itself as an efficient criteria for C-Hint to track the popularity of stored items. For example, an item with an old expired lease is much less likely to be reused in the near future than the one with active leases. Lastly, by dynamically tuning the lease terms, C-Hint can balance the trade-off between cache utilization with various performance requirements.

The first time a key-value pair is created, a lease (64 bits) is maintained into a descriptor that indexes this item. Later on, for every server-aware operation (e.g., SET or Verb GET) that accesses the item, C-Hint extends the corresponding lease by a time period, calculated by $lease = now + term$ to prevent lease from being extended unlimitedly when a large number of clients access the data simultaneously. Afterward, the server embeds extended lease term in the returning values to inform the client the valid time range during which RDMA read can be applied to fetch the data. Note that, an RDMA-based Get does not extend the lease itself unless a lease renewing message is explicitly sent to the server asking for the extension. We will further discuss this in section 3.4.

In addition, when servers are notified of the access history from clients through lease-renewing messages, they extend the leases of corresponding key-values to reflect their likelihood to be reused in the near future by the clients. Such lightweight lease update avoids time-consuming history aggregation, thereby mitigating the computation burden on the server sides.

3.3 Popularity Differentiated Lease Assignment

Though lease provides recency hint, it lacks the capability to facilitate the tracking of access frequency. Moreover, how to determine the lease term is a critical factor that can substantially impact the utilization of remote pointers. Rigidly assigning fixed lease term to all the key-value pairs irrespective of their popularity can result in many undesirable issues. For instance, resorting to long leases can cause cache to be fraught with unpopular items. On the contrary, short leases can lead clients to frequently invalidate cached remote pointers that reference to the hot key-value pairs, thereby decreasing the utilization of RDMA.

In view of these issues, we have introduced a new server-side cache replacement algorithm based on Multi-queue technique [32] in C-Hint to help determine the leases for different key-value pairs adaptively according to their recent access frequencies and memory utilization on the server. We name the policy as Popularity-Differentiated Lease Assignment policy (PDLA). It strives to attain high hit ratios for cached items on the server sides while increasing the chance for clients to leverage RDMA Get.

For each in-memory key-value pair, PDLA memorizes an approximate access count c in the descriptor that is only updated by server-aware operations and lease-renew requests. Meanwhile, it organizes all cached items according to their approximate frequency counts into M (M = 8 in our case) LRU queues. Within each queue, the least recently accessed descriptor is linked to the head. Queues with higher indices contain key-value pairs that are more frequently used and expected to receive more accesses than those in queues with lower indices. PDLA differentiates the popularity of different key-value pairs and assigns longer leases to items contained in higher queues by formula $term = qid \times \alpha$ with the intention that the remote pointers for hot items can stay valid on clients for longer period. In addition, α is further determined based on the memory utilization p following a linear model $\alpha = f(p)$. The purpose of doing so is that when cache is under heavy utilization, server cannot guarantee a long-term monopolization of one memory chunk for the same key-value pair, thus it reflects such information by decreasing the α, and such information will be conveyed to clients to limit the time range RDMA can be issued to retrieve the item. In contrast, PDLA increases α to allow more RDMA accesses when there is sufficient space on the server sides.

When hit occurs, PDLA updates the recent popularity of the item via $qid = \log_2(c)$, and determines whether the item shall be promoted into a higher queue (when qid exceeds the range, it is put into the queue with the highest index). As shown in Figure 3, a promotion inserts the descriptor of the item into the tail of next higher queue. Upon the completion of descriptor update, PDLA also examines the LRU item of queues with index$\geq$1. If the lease of the item has expired, PDLA demotes it into the tail of the lower queue in order to capture the aging of popularity. For a cache miss, PDLA iterates the LRU item of every queue attempting to find a key-value pairs with expired lease to yield space. However, in certain cases, the LRU items can still hold an active lease. Therefore, PDLA adopts a simple heuristic approach to probe the leases of r items behind the LRU key-value pair of each queue (r is set as 3 in our case) trying to find an evictable item. While, in worst case, when there is no eligible items to be evicted, we force write- or read- through.

In summary, PDLA allows C-Hint to conduct adaptive lease determination based on the recent access frequency and memory utilization. Meanwhile, by leveraging multiple LRU queues for different popularity ranks, PDLA seizes both access frequency and recency of cached items on the server with the consideration of popularity aging as well. Furthermore, with $O(1)$ complexity, PDLA imposes negligible overhead on the performance-critical operations. Our evaluation results show that compared to static lease assignment, C-Hint can efficiently optimize the utilization of remote pointers, which directly translates to the latency reduction. Meanwhile, it also delivers high hit ratios for the caching system.

3.4 Classifying Hot-Cold Items for Access History Report

To keep the server cache warm and boost the chance of reusing cached remote pointers, C-Hint resorts to the assistance from clients, asking them to periodically issue lease-renew messages (a.k.a access history report that is a vector of $< hash, version >$) so that servers can approximately track the evolving popularity of cached items. However, processing a complete access history report covering a wide time range (several seconds) can rapidly consume a significant amount of CPU power on the servers, hampering the quality-of-services to performance-critical operations.

In this regard, C-Hint prioritizes the quality-of-service over accurate popularity tracking via setting up a time-bound (e.g., 500μs), indicating the time limit Shard is willing to commit to processing a report from a single client, and relies on clients to selectively choose key-values that are deemed as worth renewing the leases. Within the response to each report message, the server embeds remaining available time that can be spared for renewing the lease to provide clients with the opportunity to renew the leases of more items.

In order to efficiently utilize such time-bound, C-Hint classifies local hot-cold items on the client sides and strives to report as much access history as server allows. Many factors affect the selection of classification mechanisms. First, although a client uses remote pointers to access data, the number of cached pointers can be significant, thus any algorithm with computation complexity above $O(1)$ cannot satisfy latency requirement. Second, the mechanism shall be adaptive to evolving workloads and resist to certain access patterns, such as *scan*. Lastly, it must be an online-algorithm with attainable simplicity. Under above constraints, we have selected self-tuning Adaptive Replacement Algorithm [24] (ARC) to facilitate the classification due to constant-time complexity, adaptivity, and simplicity.

On client side, C-Hint uses ARC to keep track of the records that have been accessed by using RDMA Get. ARC maintains 2 LRU queues with L_1 recording the items that have only been accessed once while L_2 memorizing those touched more than once. Two ghost lists are coupled with L_1 and L_2 respectively to track recently evicted items from both queues. When determining the items to renew the leases, C-Hint equally selects m MRU items from both queues under the intuition to capture both the items with growing popularity and items with established access frequency. In addition, C-Hint tags a count on each item to filter those that have been consecutively reported for several times (8 in our case). It is worth mentioning that our choice of ARC over Multiqueue to classify hot-cold items on the clients is due to its adaptivity to various client cache sizes that are allowed on client sides for caching remote pointers.

One concern about such design is that the sampling percentage from a single client may be too small (e.g., $\leq$5% access history) if a client has touched a large key spaces. However, C-Hint leverages collective efforts from many clients, thereby enabling the servers to sufficiently obtain the global access pattern and capture popular items. For workloads that exhibiting skewed access patterns, our evaluation results demonstrate that such approach can deliver both high hit ratios on server sides and allow clients to increase the chance of using RDMA Read.

3.4.1 Optimizing Access Reporting via RDMA Write

Given that the size of each history report message can be fairly large (e.g., 8KB), excessively involving the servers in the transportation of report messages can substantially burden the CPU power of hydra servers, degrading the quality-

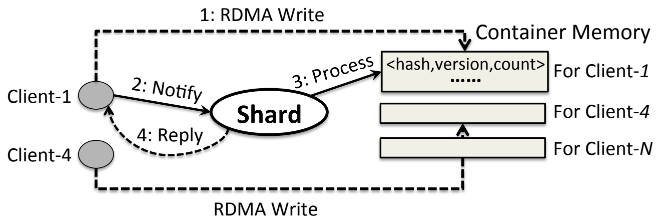


Figure 4: C-Hint uses a small-size dedicated container memory for each client connection so that RDMA Write can be leveraged to accelerate the access history report and minimize the CPU involvement on the server sides.

of-service. Therefore, we have leveraged RDMA Write to minimize the CPU overhead on servers across this procedure and accelerate the movement of history report.

Figure 4 illustrates the main idea of the optimization via RDMA write. For each client, the hydra server allocates a small memory container (*e.g.*, 8KB) to allow the client to directly put a history report via RDMA write without involving the CPU power of the server (Step 1). Upon the completion of the write, client notifies the server about the container id (Step 2), by which Shard locates the container to further process the access history (Step 3). Eventually, RDMA write, along with a notification are also used to send back the reply (Step 4). Although such design increases the memory space overhead, it is a tradeoff that we deem worth making to reduce the CPU overhead and accelerate the movement of report messages. Our evaluation studies show that by leveraging RDMA write, we can efficiently reduce the cost of reporting access history by as much as 115%.

3.5 Delayed Memory Reclamation

To ensure reliable and correct data fetching, C-Hint applies delayed memory reclamation to comply with the *pact* signed by the lease. When a REMOVE or Time-To-Live (TTL) operation deletes a key-value pair, C-Hint marks the key-value pair as deleted without immediately reclaiming the memory chunk. Only after lease expires, C-Hint frees memory area that then can be reallocated to accommodate other key-value items. Such mechanism avoids unreliable reference counts on each item and eliminates the expensive broadcast to invalidate cached remote pointers.

An interesting issue that can violate the pact is the unsynchronized clock between servers and clients. The violation happens when server-side clock proceeds faster than that on clients, then an inconsistent view of the lease will occur and it becomes possible that clients will attempt to RDMA Get an item that has been reclaimed by the server. In this work, we assume that the clock difference ε is in a much smaller scale (microseconds) than that of lease terms (several seconds), and previous work [31] that provides reliable clock synchronization can be employed to address this issue.

4. Implementation

To this end, we have implemented and integrated C-Hint into HydraDB by using C++. During the implementation, we have observed several technique details that are unique to RDMA Read Accelerated key-value stores.

Disallowing automatic memory compaction: Once lease is determined, a specific memory chunk cannot be moved arbitrarily even when system is experiencing high memory fragmentation. Such constraint is unique to the key-value stores that allow RDMA Read, which requires pages to be pinned. At this point, we simply disable automatic memory compaction within our memory allocator. When memory fragment becomes severe, RDMA access is disabled while the system can continue serving Verb-base operations.

Remote pointer sharing among co-located clients: when many clients are co-located on the same physical machine, we allow a remote pointer cached by one client to be visible to all co-located clients via storing all the remote pointers in a shared hash-table residing in a shared memory. The purpose is to reduce the first time access latency if the same items have been visited by other clients on the same node.

Client-side report batching and consolidation: To further reduce the amount of access reports, we have batched the access history from co-located clients on the same machine and removed redundant items in the batched reports. However, simple consolidation can lose a critical access frequency information, therefore, we have appended optional *count* fields in the report message (as shown in Figure 4) to take account of total access frequency of each item from all local clients over the past time range.

5. Performance Evaluation

This section aims to investigate the behavior of C-Hint and its design choices from two major aspects, including the *Hit Ratio* and *Latency* impact. In order to gain a better understanding of the performance of C-Hint, we have also introduced several other cache management alternatives and systematically compared C-Hint to them. The following four alternatives have been included:

- First of all, to validate the necessity of introducing the C-Hint, we have compared to the original HydraDB (Original) that simply allows all the clients to retain all accessed remote pointers all the time without any effort to differentiate the popularity of key-value pairs.
- A faithful LRU implementation (LRU), in which we disable the RDMA capability and force all the requests to be processed by the hydra servers. The purpose of doing so is to establish a baseline hit ratio to compare with.
- We have also implemented an approximate LRU (Approx-LRU) based on the framework offered by C-Hint. But instead of using PDLA, We use a single LRU queue for each partition on the server sides to approximately track the recency of accessed items via relying on the history

report from clients. However, due to the lack of capability to capture the frequency, such design is unable to differentiate the lease terms for key-value pairs with distinct access patterns.

- Furthermore, an faithful Multi-Queue implementation (MQ) [32] has been carried out without using the RDMA so we can compare to the hit ratios that can be achieved by the algorithm that is specifically designed for second level buffer caching.

5.1 Experimental Environment

Cluster Setup: All of our experiments are conducted on a cluster of 5 x86_64 machines. Each machine is equipped with 2 2.60GHz Intel Xeon E5-4650L Octa-core processors, featuring 256KB for L1 instruction and data caches, 2MB L2 and 20MB L3 cache. 256GB memory is installed on each node. We have turned off the Hyper-threading during the evaluations. All nodes are connected through InfiniBand FDR using Mellanox MT27500 ConnectX-3 HCA. On each machine, we run Red Hat Enterprise Linux Server 6.4.

To set up the key-value cache, we have allocated 4 hydra server instances on 2 different machines while spread the clients among the rest nodes. Within each server instance, we run 1 shard process with 8 communication threads. All the memory used as cache are registered to the NIC during the runtime to allow RDMA Read. Note that such cache only contains the key-value data without including the hash table that is used for primary indexing. In addition, PDLA is tuned to determine the lease terms between 4∼256 seconds based on the popularity of key-value pairs.

Furthermore, due to the incompletion of the integration with distributed persistent storage systems, we have set up another HydraDB cluster with 4 servers and 320GB total memory on the same cluster to emulate the backend storage. When cache miss occurs, the cache-end HydraDB accesses the backend hydra to retrieve the data. Given such system configuration, we focus on evaluating the hit ratios and latency distribution instead of throughput to investigate the cache management performance of hydra across our experiments.

It is also worth mentioning that the reliance on the access history reports from clients to track the popularity of key-value pairs increases the vulnerability to the manipulation from malicious clients in a wild environment. However, in this work, we assume that HydraDB runs in a controlled cluster environment with all the reports from clients are trustworthy. We defer the work of how to detect and cope with malicious activities without degrading the performance as a future research topic.

Benchmark: We have employed Yahoo! Cloud Serving Benchmark (YCSB 0.14) [8] to examine the performance of C-Hint. A C++ Client has been implemented to bridge the HydraDB with YCSB. Given that YCSB workloads generator can be highly CPU-intensive, we have designed our YCSB clients to fully buffer the workloads in memory before starting the tests and scattered all the clients among 3 machines to prevent them from being the bottleneck. Note that our YCSB client does not batch local requests and only issues all the requests in a sequence manner.

Based on the production workload analysis from many previous work [5, 12] that reveals highly skewed GET/SET ratios (can be as much as 500:1) and Zipf distribution, we have generated three workloads with different GET/SET ratios, which are 100% GET + 0% SET, 95% GET + 5% SET, 90% GET + 10% SET, respectively, all following the Zipf distribution. Each workload consists of 300 million operations querying 80 million key-value pairs (23B Key with 1KB value), featuring about 80GB of data in total.

Metrics: Different from conventional key-value stores, in which the server-side hit ratio is the major criterion to measure the efficiency of cache management. While for RDMA Read accelerated key-value stores, both client-side RDMA GET and regular GET contribute to the total hit ratio, and the RDMA GET is more critical than the latter to the performance. Therefore, it is imperative to distinguish the hit ratios contributed by these two different operations. Thus, in this paper we define following two hit ratio metrics.

- $Total\ Hit\ Ratio = \dfrac{RDMA\_Get\ Hit + Verb\_Get\ Hit}{Total\ Get\ Operations}$
- $Remote\ Pointer\ Hit\ Ratio = \dfrac{RDMA\_Get\ Hit}{Total\ Get\ Operations}$

Note that a higher *remote pointer hit ratio* implies a more efficient utilization of RDMA.

5.2 Hit Ratio Analysis

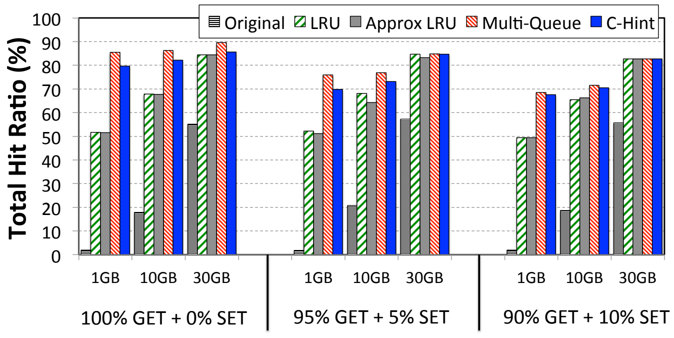


Figure 5: Overall hit ratio comparison between C-Hint and the other four solutions.

In this section, we firstly validate the necessity of introducing C-Hint by examining the total hit ratios delivered by original HydraDB. Then we compare the total hit ratios C-Hint can achieve with that of the other three solutions. Following that we demonstrate how C-Hint is able to efficiently improve the remote pointer hit ratio. Throughout the experiments, we have deployed 36 hydra clients over 3 physical machines and demanded them to propagate the history report in every 10 seconds for the C-Hint and Approx-LRU tests. Meanwhile, we vary the total aggregated cache sizes from 1GB to 30GB. Figure 5 shows the hit ratios comparison.

As shown in Figure 5, original HydraDB delivers poor hit ratios for all the workloads due to its incapability to keep track of the popularity of all the key-value pairs. For 1GB cache size, the hit ratios are consistently below 2% for all three workloads. Though increasing the cache size improves the hit ratio since more items are retained in memory, it still performs 1.5∼3.8× worse than the baseline LRU, rendering it as an inefficient solution to be used in practice.

In contrast, for all the three workloads, C-Hint effectively outperforms the LRU and Approx-LRU for small caches ($<$ 30GB) by considering both the recency and frequency of accessed items. For 100% GET + 0% SET workload and 1GB cache case, C-Hint improves the hit ratios by up to 54.1% and 54.6%, compared to LRU and Approx-LRU, respectively. However, since C-Hint only seizes approximate access frequencies, C-Hint performs marginally worse than the faithful Multi-Queue implementation for small caches. The maximum hit ratio degradation is 8% for the 95%+5% 1GB case. When cache size is increased, C-Hint achieves comparable total hit ratio as MQ.

In addition, with the increase of the SET percentage, the hit ratios of LRU and Approx-LRU remain fairly stable. While, there is non-negligible hit ratio reduction in MQ and C-Hint (20% and 15.3% for 1GB cache). The causes are mainly two-fold. First, the number of eviction rises in response to the increase of SET percentage. Second, the SET are uniformly distributed within the workload, thereby increasing the chance of evicting popular items before MQ and C-Hint accurately capture them.

5.2.1 Benefits of Adaptive Lease Assignment

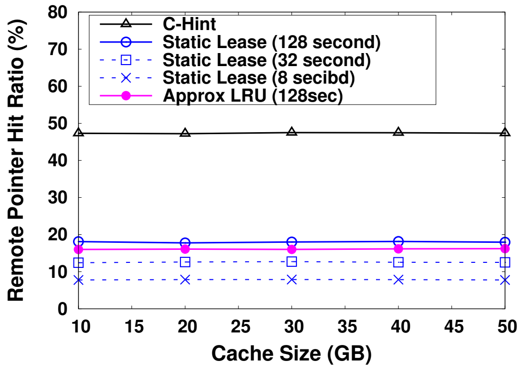


Figure 6: Hit Ratio of remote pointers cached on the client side with 95%GET + 5%SET workload.

Then we further examine how much hit ratios are contributed by the RDMA Get and the advantage of *PDLA* over static lease assignment for improving the remote pointer hit ratios. Due to similar performance, we have used the results from 95% GET + 5% SET workload as an illustrative case. Figure 6 shows the performance results. Because all the faithful cache replacement algorithms (MQ and LRU) require all the operations to be processed by the server-sides without being able to use RDMA, their remote pointer hit ratios are always 0%, thus we omit them in the figure for clear presentation.

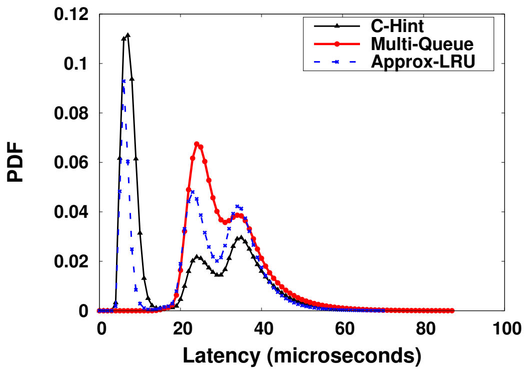
As shown in the figure, C-Hint can effectively leverage the remote pointers to conduct RDMA operations. On average, C-Hint enables RDMA Get to contribute up to 47.4% hit ratios for different cache sizes, ranging from 10GB to 50GB. In addition, C-Hint substantially outperforms Approx-LRU, since the latter is unable to differentiate the lease terms for key-value pairs with distinct popularity. Furthermore, to demonstrate the efficacy of using adaptive lease assignment, we have modified the C-Hint to use static lease terms, which are 128, 32, 8 seconds, respectively. As revealed by the results, C-Hint can efficiently outperform static lease assignment by up to 2.6×, 3.8×, 6.1×, respectively for above three different static lease terms. And smaller lease terms can cause more frequent remote pointer invalidation, leading to even lower hit ratios.

One interesting observation is that the remote pointer hit ratio remains stable across different cache sizes. Such phenomenon is because it is correlated to the number of accesses to the hot items that is determined by the workload. When cache size is sufficiently large enough to include those popular key-value pairs, further increasing the cache size is only beneficial to cover small percentage of hit rates for unpopular key-value items.

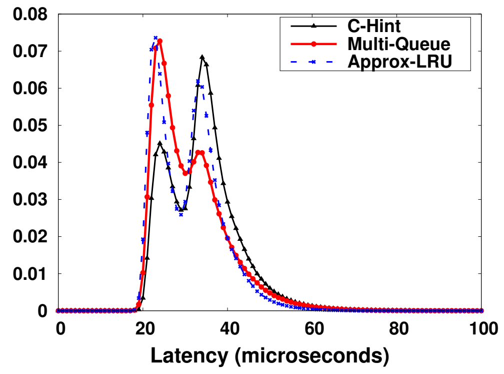
5.3 Latency Analysis

Figure 7 compares the Probability Distribution Function (PDF) of read and write latencies yielded by C-Hint, Approx-LRU, and Multi-Queue, respectively. As shown in Figure 7 (a), the read latency distribution of C-Hint and Approx-LRU exhibit clear multimodal distribution with three discrete modes, illustrating the percentages of latencies contributed by RDMA-GET, Fast Verb-GET, and relatively Slow accesses that are delayed by factors, such as congested server *etc*, respectively. While, the PDF of MQ shows a bimodal distribution since it disables the RDMA-GET. On the other side, the PDF of write latency, shown in Figure 7 (b), always displays two discrete modes because all the writes have to be processed by the server and there is only a distinction between Fast Verb-Set, and delayed writes.

From the perspective of latency performance, C-Hint substantially reduces the average latency by shifting 49.4% of GET operations from server-sides to clients and allows 44.1% GET to achieve latency below 10μs. While the Approx-LRU only accelerates 24.2% of GET via RDMA Read and only 23.7% GET obtains latency of $\leq$ 10μs. Such latency differentiation between C-Hint and Approx-LRU is



(a) Read Latency.



(b) Write Latency.

Figure 7: Latency comparison between C-Hint, Multi-Queue, and Approx-LRU as well by using 95%GET + 5%SET workload.

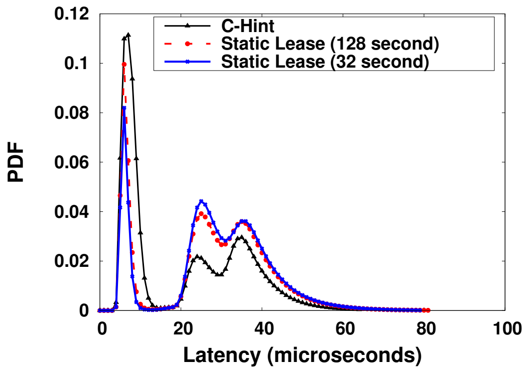


Figure 8: C-Hint can efficiently improve the GET latency by avoiding statically assigning the lease terms without differentiating distinct access patterns.

corroborated by the left-most steep "bell curves" in Figure 7 (a). For MQ, all the requests have to reach server-sides, therefore imposing a lower bound of latency as 16μs and exhibiting a bimodal latency distribution differentiating Fast Verb-GET from Slow item retrieval. Beside latency improvement, an immediate impact of such workload shifting provided by C-Hint is fewer amounts of GET operations on the server sides verified by the lower curve of C-Hint between 20 and 40 μs, thereby efficiently decreasing the computation burden on the servers.

However, as shown in Figure 7 (b), compared to MQ that can rapidly locate an evictable item during the eviction, C-Hint imposes overhead on the SET due to multiple heuristic probes for looking up the key-value pairs with expired lease during the eviction. Compared to MQ, 18.5% of SET are penalized by 9μs. while the overhead compared to Approx-LRU is negligible. Nevertheless, given that the percentage of SET is usually small in the real workloads, such slight overhead is a worthy tradeoff to make for achieving highly optimized GET latency.

Recall that section 5.2.1 illustrates the benefits of popularity-differentiated lease assignment offered by PDLA on remote pointer hit ratios. The performance enhancement of improved hit ratios is directly reflected on the latency reduction. As shown in Figure 8, C-Hint achieves a much higher percentage of ultimate low latency access (between 4 and 10 μs) than using static lease assignment, reaching up to 24.2% and 26.7% higher percentage of low latency access than 128 and 32 second static assignments, thereby furthering substantiating the necessity of differentiating the lease terms for key-value pairs of distinct access patterns.

5.4 Impact of Access History Report

A major concern about C-Hint is its communication overhead incurred by access history reports issued by clients. To evaluate its impact, we vary the report frequency from 64 seconds to 1 second to increase the amount of reports. Meanwhile, we disable the batching and consolidation during this experiment. For the following experiments, we run one hydra server instance on a single machine to remove unnecessary noise that can impact the results. The workload we use is 95%GET + 5%SET.

Figure 9 illustrates the latency distributions under different report frequencies. Overall, the results show an ideal overlap of four test cases, indicating that increasing the amount of reports does not cause observable performance degradation. A key factor that contributes to such lightweight reporting is the adoption of RDMA Write, which efficiently reduces the CPU involvement on the server side and accelerates the movement of report messages as shown in Figure 10, which further compares the cost of using different approaches to conduct history reports. Since the computation cost of renewing the leases is bounded as elaborated in section 3.4, the results in the figure only include the com-

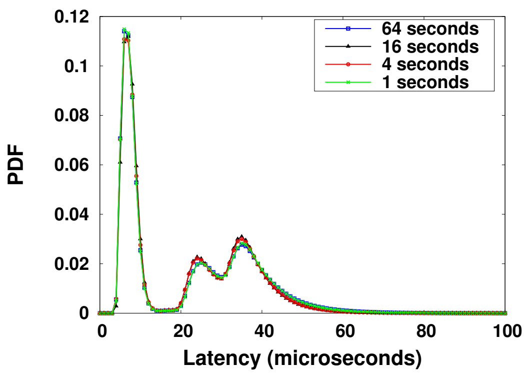


Figure 9: Frequently reporting the access history imposes negligible performance overhead.

munication cost. As shown in the figure, leveraging RDMA Write can efficiently reduce the cost by up to 115% from the perspective of latency, compared to Verb-based approach.

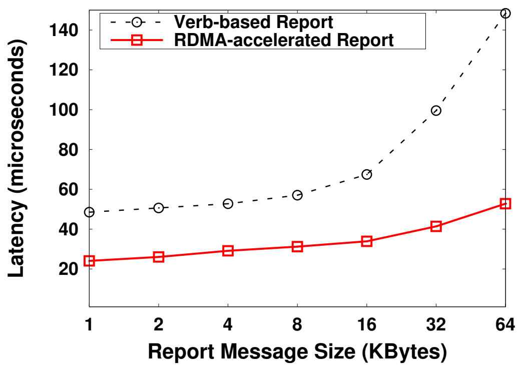


Figure 10: Using RDMA Write to conduct history report can efficiently save communication cost.

6. Related Work

High-Performance Networks: Leveraging fast network hardware and protocols, particularly RDMA, to accelerate key-value stores, such as Memcached *etc*, and cluster computing frameworks has attracted many research efforts [10, 21, 25, 28–30] over the past few years. And recently, many cloud providers are actively marketing RDMA-capable cloud services, such as Softlayer [4].

Mitchell, *et al* [25] have investigated the Get operations of Memcached using RDMA Read and introduced a self-verification data structure to inhibit the race between RDMA Get and server-side writes. Jose, *et al* [21] have attempted to improve the Memcached by integrating *Unified Communication Runtime* into the store to allow Memcached transparently benefit from high-performance networks. However, such design still requires each operation to send a verb message for obtaining the metadata of a key-value from the targeting server before conducting RDMA operation, thus unable to fully exploit the performance of RDMA semantics. Stuedi, *et al* [28] have studied the benefit of using Soft-RDMA [26] on Memcached. Its Get operation bears a resemblance to that of HydraDB. Client firstly probes if a stag entry exists locally before deciding if a RDMA Read can be issued to retrieve data. Dragojevic, *et al* [10] have designed a new key-value store from ground up to demonstrate the efficiency of a memory-based distributed computing platform, named FaRM, that leverages RDMA as communication channel to establish a shared address spaces. However, in contrast to above studies that solely examine the performance benefit of RDMA on key-value stores, our work aims to address the cache management challenges raised by leveraging the RDMA Read that deprives the servers of the capability to keep track of the popularities of key-value pairs.

More recently, Kalia, *et al* [22] have explored the feasibility of combining Unreliable-Connection (UC)-based RDMA Write with Unreliable-Datagram (UD)-based Send for in-memory key-value stores, and demonstrated via a system, named HERD, that they can outperform RDMA-Read oriented designs for certain workloads. However, their design is at the cost of reliable data transmission, which is a highly desirable feature required by many datacenter applications. It shifts all the responsibility relating to the message loss detection and retransmission to the key-value store level, thus complicating the design when reliability is necessary. In contrast, HydraDB provisions reliable data transfer via following RDMA-Read based strategy. Nevertheless, there is indeed a great potential of leveraging RDMA Write to further optimize the UPDATE/SET operations, and we are actually in the process of exploiting the advantages of both approaches to enhance HydraDB.

Caching Optimization: Many research efforts have been invested on optimizing the performance of in-memory key-value stores, however, little work has been carried out to examine the efficiency of their cache management layer.

As elaborated by [12], maintaining strict LRU order can incur severe synchronization overhead for key-value stores that use single partition on each server. Accordingly, Fan, *et al* have proposed using CLOCK replacement algorithm [9] to eliminate the need of locking the LRU queue along the critical path of Get operations. For workload that follows the Zipf distribution, their performance results show that their CLOCK implementation can achieve comparable hit ratios as LRU while delivering much higher throughput.

Different from Memcached type of key-value stores, each partition is exclusively managed by a single CPU core in HydraDB, thus the needs of locking shared data structures for tracking the frequency or recency of key-values within the partition does not exist. However, CLOCK is still potentially beneficial given its less memory utilization and higher cache efficiency. Therefore, we aim to further investigate its performance impact on HydraDB in our future study.

Cache Replacement Algorithms: Although there is a large body of research on cache replacement algorithms, LRU is still one of the dominant approaches widely used by many memory-resident key-value stores, including Memcached, Redis [3] and Hazelcast [1] *etc*, due to its simplicity and efficacy. However, simply considering the recency is insufficient for large-scale caching systems. Yuanyuan, *et al* [32] have observed that access frequency reflects the popularity more accurately than recency on second level buffer caches but LFU does not perform well, because LFU cannot adapt to evolving workloads due to its incapability of capturing the aging of frequency. Accordingly, the authors have introduced Multi-queue (MQ) algorithm to seize both frequency and recency of accessed items. Though exhibiting higher hit ratios than other alternatives on the server sides, including 2Q [19], LRU-2, MRU *etc*. MQ cannot satisfy the demands of HydraDB that needs to consider a hierarchical hit ratios of both remote pointers cached on the client sides and items on the server-sides. Bairavasundaram, *et al* have introduced X-Ray [6] for RAID array caches to reduce the chance of caching same blocks within array caches and file system buffer caches redundantly. However, its goal to achieve high exclusive caching is different from that of C-Hint, which aims to provision high RDMA utilization for items cached on server sides. Many other cache algorithms [17, 18, 24] have also been studied for the processor caches, virtual memory management and disk controllers. But only a few of them have been investigated as a replacement policy for distributed caching systems.

7. Conclusion

As emerging in-memory key-value stores are leveraging high-performance networks to push forward new boundary of performance. It is equally critical to have efficient cache management systems to enhance the practicality of new techniques. Accordingly, in this paper, we have introduced C-Hint, an effective and reliable cache management system that has conquered three major challenges raised by using RDMA Read to accelerate memory-resident key-value stores. Those three challenges include (1) How to keep track of the popularity of cached items when servers are unaware of the RDMA operations. (2) How to efficiently and reliably conduct resource reclamation on the servers. (3) How to efficiently improve the hit ratio without sacrificing the performance advantages offered by RDMA. C-Hint has synthesized a collection of ideas, including Client-Assisted cache management, Lease-based key-value management, and Popularity-Differentiated Lease Assignment, along with several optimizations, such as classification of hot-cold items and RDMA accelerated history report, to collectively achieve the goal of efficient cache management. We have demonstrated the effectiveness of such design by implementing C-Hint on a high-performance key-value store, called HydraDB. Our experiment results substantiate that C-Hint can efficiently outperform other alternate solutions, and deliver both high hit ratio and low latency performance.

Acknowledgments

We are very thankful to our shepherd Dr. Dushyanth Narayanan and anonymous reviewers for their invaluable feedback that helped improve this paper.

References

[1] hazelcast. http://hazelcast.com/.

[2] Memcached. http://memcached.org/.

[3] Redis. http://redis.io/.

[4] Softlayer, an IBM Company. http://www.softlayer.com/.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[6] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-ray: A non-invasive exclusive caching mechanism for raids. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 176–, Washington, DC, USA, 2004. IEEE Computer Society.

[7] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 1–14, New York, NY, USA, 2007. ACM.

[8] 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 '10, pages 143–154, New York, NY, USA, 2010. ACM.

[9] F. J. Corbato. A paging experiment with the multics system.

[10] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association.

[11] Venkata Duvvuri, Prashant Shenoy, and Renu Tewari. Adaptive leases: A strong consistency mechanism for the world wide web. *IEEE Trans. on Knowl. and Data Eng.*, 15(5):1266–1276, September 2003.

[12] 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 Conference on Networked Systems Design and Implementation*, nsdi'13, pages 371–384, Berkeley, CA, USA, 2013. USENIX Association.

[13] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP*, pages 202–210, 1989.

[14] Lim Hyeontaek, Han Dongsu, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, nsdi'14, Berkeley, CA, USA, 2014. USENIX Association.

[15] InfiniBand Trade Association. The InfiniBand Architecture. http://www.infinibandta.org/specs.

[16] J. Chu and V. Kashyap. Transmission of IP over InfiniBand(IPoIB). http://tools.ietf.org/html/rfc4391, 2006.

[17] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. Dulo: An effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.

[18] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, pages 31–42, New York, NY, USA, 2002. ACM.

[19] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[20] Jithin Jose, Hari Subramoni, Krishna Kandalla, Md. Wasi-ur Rahman, Hao Wang, Sundeep Narravula, and Dhabaleswar K. Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 236–243, Washington, DC, USA, 2012. IEEE Computer Society.

[21] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. Memcached design on high performance rdma capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 743–752, Washington, DC, USA, 2011. IEEE Computer Society.

[22] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[23] Patrick MacArthur and Robert D. Russell. A performance study to guide rdma programming decisions. In *HPCC-ICESS*, pages 778–785. IEEE Computer Society, 2012.

[24] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[25] 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 Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, Berkeley, CA, USA, 2013. USENIX Association.

[26] F. D. Neeser, B. Metzler, and P. W. Frey. Softrdma: Implementing iwarp over tcp kernel sockets. *IBM J. Res. Dev.*, 54(1):50–65, January 2010.

[27] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, January 2010.

[28] 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 Conference on Annual Technical Conference*, USENIX ATC'12, pages 31–31, Berkeley, CA, USA, 2012. USENIX Association.

[29] Yandong Wang, Robin Goldstone, Weikuan Yu, and Teng Wang. Characterization and optimization of memory-resident mapreduce on HPC systems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 799–808, 2014.

[30] Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, and Dhiraj Sehgal. Hadoop acceleration through network levitated merge. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 57:1–57:10, New York, NY, USA, 2011. ACM.

[31] Li Zhang, Zhen Liu, and C. Honghui Xia. Clock Synchronization Algorithms for Network Measurements. In David Lee and Ariel Orda, editors, *INFOCOM 2002. 21st Annual Joint Conference of the IEEE Computer and Communications Societies on Computer Communications*, volume 1, pages 160–169, Psicataway, NJ, USA, June 2002. IEEE Computer Society, IEEE Communications Society, IEEE.

[32] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 91–104, Berkeley, CA, USA, 2001. USENIX Association.