Future Generation Computer Systems 92 (2019) 162–177

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

RHKV: An RDMA and HTM friendly key–value store for data-intensive computing

Renke Wu [a], Linpeng Huang [a,*], Haojie Zhou [b]

[a] Department of Computer Science and Engineering, Shanghai Jiao Tong University, 800 Dong Chuan Road, Minhang, Shanghai, 200240, China
[b] The State Key Laboratory of Mathematic Engineering and Advance Computing, 699 Shanshui East Road, Wuxi, 214000, China

HIGHLIGHTS

- We introduce the architecture of RHKV key–value store to provision data access.
- We establish RDMA communication model to construct RDMA network engine.
- We design an improved hashing scheme named G-Cuckoo for managing and storing data.
- The scheme constructs Cuckoo graph to alleviate the endless loop problem.
- We synthesize HTM technique to guarantee the atomicity of data operations.

ARTICLE INFO

Article history:
Received 17 January 2018
Received in revised form 15 July 2018
Accepted 1 October 2018
Available online xxxx

Keywords:
Key–value store
RDMA
HTM
G-Cuckoo
Performance

ABSTRACT

Serving DRAM as the storage through key–value abstraction has proved as an attractive option, which provides fast data access for data-intensive computing. However, due to the drawbacks of network round trips and requesting conflicts, remote data access over traditional commodity networking technology might incur high latency for the key–value data store. The major performance bottleneck lies in client-side request waiting and server-side I/O overhead. Accordingly, this paper proposes ***RHKV***: a novel RDMA and HTM friendly key–value store to provide fast and scalable data management by using the designed *G-Cuckoo* hashing scheme. Our work expands the idea as follows: (*i*) An RHKV client transmits data requests to our improved Cuckoo hashing scheme — *G-Cuckoo*, which constructs a Cuckoo graph as directed pseudoforests in RHKV server. The RHKV server computes the reply for each data request. The server maintains a bucket-to-vertex mapping and pre-determines the possibility of a loop prior to data insertion. Through the use of this Cuckoo graph, the endless kick-out loop of data insertions that can potentially be experienced in the case of generic Cuckoo hashing can be detected. (*ii*) Despite messaging primitives are slower than RDMA READs for data requests, RHKV adopts RDMA messaging verbs unconventionally. It leverages rich network semantics and makes full use of RDMA's high bandwidth and low latency for data access over high-performance RDMA interconnects. (*iii*) Moreover, in order to ensure the data operation's atomicity, RHKV strives to utilize the advanced HTM technique. Experimental performance evaluation with YCSB workloads shows that, when basic data operations are conducted, RHKV outperforms several state-of-the-art key–value stores with lower latency and higher throughput.

© 2018 Elsevier B.V. All rights reserved.

1. Introduction

In the last several years, the demand of in-memory data-intensive computing increases fastly. In modern high-performance clusters (HPC), key–value data store is being deployed as a storage solution to keep up with the incoming data access. How to achieve both high throughput and low latency for key–value data storage is a critical issue [1].

In the 21 century, a large-scale of data has led to a new concept of "Big Data", which includes the characters of 4V – "Volume", "Variety", "Velocity" and "Veracity". Specifically, *volume* refers to the amount of data, which is stored in physical storage systems; *variety* refers to the various formats of data which data storage deals with (texts, raw data, etc.); *velocity* refers to the speed at which data is created, stored and analyzed; *veracity* refers to the trustworthiness of the value which is obtained from the data of a data-intensive computing application.

The above requirements make big data processing manner beyond the use of relational databases. The new generation of databases, named NoSQL ("No SQL" or "Not only SQL") key–value stores, have emerged as the core technology for processing big

* Corresponding author.
E-mail addresses: sjtuwrk@sjtu.edu.cn (R. Wu), lphuang@sjtu.edu.cn (L. Huang), zhou.haojie@meac-skl.cn (H. Zhou).

https://doi.org/10.1016/j.future.2018.10.001
0167-739X/© 2018 Elsevier B.V. All rights reserved.

data. The conventional operations of processing data often refer to CRUD operations – *Create*, *Read*, *Update* and *Delete*. These data operations usually possess ACID (Atomicity, Consistency, Isolation, Durability) properties.

When conducting data access upon key–value data storage, general client/server mode can be leveraged over interconnect or networking, especially in modern clustering environment. However, remote data access based on client/server mode over traditional networking, e.g., Ethernet, might exist the performance bottleneck [2]. Due to the drawback of network round trips and the disadvantage of supporting fast data transmission, the high performance of data access for key–value data stores would be not obtained easily [3–5]. Fortunately, Mellanox and Intel company manufacture InfiniBand interconnect solutions for servers, storage or hyper-converged infrastructure. The high-performance Infiniband Remote Direct Memory Access (RDMA) networking technology emerges subsequently [6]. Infiniband RDMA is able to overcome the drawback of network round trips effectively. Thus, in today's modern clustering environment, a common way to improve data operation's performance is to leverage Infiniband RDMA, so as to relieve the existing data request's overhead and reply traffic [7].

Note that, when remote data access is conducted normally, RDMA READs can be utilized to traverse remote memory, similar to operating data in local memory [7]. RDMA READs bypass the host CPU and initiate control transfers. For instance, Pilaf [7] has been designed to leverage RDMA READs to conduct GET operations for distributed key–value store based on caching solutions, e.g., Memcached [8]. Moreover, the boost in throughput of FaRM [9], etc., comes from bypassing the network stack and avoiding CPU interrupts by using RDMA READs.

Even RDMA READs can be used to accelerate data operations, we argue that directly leveraging RDMA READs has limitations for realizing the existing key–value service schemes. It is difficult to handle sophisticated operations, e.g., the remote memory's following and dereferencing a pointer [10]. Thus, we attempt to design a key–value data storage without directly using RDMA READs at all, but harnessing high-performance RDMA messaging with designing an efficient hashing scheme.

It is also challenging to design a key–value data store on basis of *an efficient hashing scheme*, instead of operating sophisticated pointers on memory directly. Adopting Cuckoo hashing scheme is one of convenient ways to manage basic data elements [7]. However, the conventional Cuckoo hashing scheme would actually exist *the endless kick-out loop problem* between hash tables during the data item's insertion. Hence, the performance would be degraded obviously. Our actual purpose is to leverage fast RDMA protocol for data communication, and leverage RDMA READs to obtain the optional object persistence, thus leading to the similar effect of direct memory access.

Additionally, in order to avoid concurrent requesting conflicts, data operation's atomicity onto key–value data stores should be ensured. When there is any concurrent conflicting operation, the remote machine would abort. Currently, Hardware Transactional Memory (HTM) has become popular and appeared in the forms of Hardware Lock Elision (HLE) and Intel's Restricted Transactional Memory (RTM) [11]. HTM can be utilized for doing concurrency control [12]. Hence, there exists an opportunity to combine HTM and RDMA to design an efficient key–value data store with using a hashing scheme. To be concrete, researchers and engineers leverage the fallback handler of HTM to provide the contention management, so as to prevent possible locks effectively.

Actually, several key features should be considered for designing the "RDMA+HTM"-based key–value data store: (*i*) possessing high efficiency and high throughput for data access, (*ii*) consuming low CPU resource for RDMA-based communication, (*iii*) having an efficient hashing scheme with returning the result for any data request, and (*iv*) existing no data request conflicts, and so on.

Motivated by these observations, we propose **RHKV**, a versatile and general-purpose RDMA and HTM friendly key–value data store, which realizes high throughput and low latency from two dimensions of *communication and I/O*. Specifically, RHKV client transmits data requests to RHKV server's *G-Cuckoo* hashing scheme by using RDMA messaging verbs, due to its high performance feature in data transmission. An improved Cuckoo hashing scheme – *G-Cuckoo* is designed in RHKV server to alleviate the endless kick-out loop problem between two basic hash tables by the means of *bucket-vertex mapping*. The scheme ensures each data request to be accomplished with low latency and low cost. While receiving a data request, the server executes the data request's task in local hashing scheme. After the data operation is done, the response would be sent back to RHKV client. The response is sent as a message over a datagram connection. Moreover, RHKV strives to utilize HTM technique to ensure the atomicity of each data operation.

Contributions. The following main contributions are made:

- We introduce the holistic architecture of key–value data store RHKV to provision fast data access.
- We establish RDMA communication model, and leverage RDMA messaging verbs to construct RDMA network engine for accelerating data access.
- We design an improved hashing scheme named G-Cuckoo for managing data and storing data. The scheme constructs Cuckoo graph by bucket-vertex mapping. It alleviates the endless kick-out loop problem between two hash tables by using predetermination process, during the key–value data item's insertion.
- We synthesize HTM technique and two-phase locking (2PL) -like protocol to guarantee the atomicity of each data operation and prevent possible locks.
- We conduct the experimental evaluation of the prototype implementation of RHKV over an InfiniBand environment (FDR interconnects). This evaluation leverages YCSB [13] workloads with Zipfian distribution and Uniform distribution, as compared to the relevant state-of-the-art key–value stores.

The remainder of the paper is organized as follows: Section 2 follows related work. Section 3 starts by presenting background and prior knowledge. Section 4 introduces the detailed design of RHKV. Section 5 shows some implementations within RHKV. Section 6 reports the experimental campaign of the proposed key–value data store, including some illustrated application scenarios. Section 7 concludes the paper.

2. Related work

NoSQL databases can play a vital role in data-intensive computing systems, because they provide the functionality of data store. To reduce time cost and improve throughput, we raise the need to construct our data store's architecture. Therefore, in this section, we will introduce the relative knowledge of *NoSQL databases*, *RDMA-based key–value data stores*, *transaction services for NoSQL databases* and *data-intensive computing for NoSQL databases*, respectively.

2.1. NoSQL databases

In relational databases, strong ACID properties should be basically ensured for data operations. As so far, SQL (Structured Query Language) has been widely used for operating relational databases. Nowadays, new technologies have emerged and data-intensive computing applications have been significantly increasing in the amount of data, e.g., social media, weather data, etc. They have been developed in the new trend of databases, named as NoSQL databases.

NoSQL databases deal with large-scale data in real time. Such data-intensive computing applications should require basic data operations with high response time, availability and scalability. The conventional key–value data store often consists of a set of key–value data pairs. To be concrete, the values within the key–value store can be text string or complex lists. It is common knowledge that, data searches are conducted on basis of keys, not values. Famous key–value data stores, such as Redis [14], BerkleyDB [15], MongoDB [16], etc., have appeared in the last years. Simple key–value data stores can be the standard data exchange files, e.g., XML and JSON. These files are useful for data services, keys and values are searchable. Alternatively, tables of databases can be stored in column families. These examples can be Cassandra [17], SimpleDB [18], DynamoDB [19], etc.

Moreover, a convergence of distributed key–value data storage systems are presented continuously [20]. Specifically, ZHT, a zero-hop distributed key–value data store system, which has been tuned for the requirements of high-end computing systems, is also given in the literature [20]. As same as ZHT, our proposed RHKV is a pure key–value data store. RHKV also aims to be a building block for distributed systems, e.g., distributed job management systems, parallel and distributed file systems, or parallel programming systems [20]. In RHKV key–value data store, key–value pairs can be grouped into a namespace named "bucket". The established key–value store should be believed to provide satisfactory requirements of performance, consistency and availability.

2.2. RDMA-based key–value stores

This subsection illustrates different types of RDMA-based key–value databases. Distributed in-memory key–value store systems have witnessed a burgeoning development [15,21–23]. The popular key–value data store system, Memcached [8], has been developed by leading organizations at scale to cache database queries objects from the servers using "keys". Nevertheless, Memcached cannot efficiently support RDMA communication protocol. Redis [14] is another RAM-based key–value store. It offers data object persistence and master–slave data replication. But, there exist the limitations of sophisticated data partitioning and load rebalancing for Redis. Also, RAMCloud [24] demonstrates the capacity of InfiniBand to reduce the read latency of small data objects. In the above data store systems, few leverages RDMA communication protocol for data transfers natively.

Because of RDMA's high-bandwidth (e.g., 56 Gbps), low-latency (e.g., 1-3 μs) and one-sided zero CPU consumption capability, utilizing RDMA communication protocol for message communication has obtained great attention. The boost to throughput of in-memory key–value data store, e.g., Pilaf [7], FaRM [9] and HERD [10], comes from bypassing the network stack and avoiding CPU interrupts. HERD analyzes the *pros* and *cons* of different RDMA primitives [10]. However, it makes full use of one-sided zero CPU consumption capability. HERD does not guarantee reliable data transfers. Pilaf has been introduced to optimize Memcached by leveraging RDMA READs to improve the performance of GET operations, while continuing to rely on message passing for other performance-critical operations. Jose, et al. have tried to optimize the speed of GET operations in Memcached by utilizing RDMA [25]. But, their solution requires a control message to notify the completion of data access on memory, thereby needing an additional time cost of round trip.

Another in-memory key–value data store, MICA, has been proposed in the literature [26]. MICA is a scalable in-memory key–value data store that handles 65.6 to 76.9 million key–value data operations per second by using a single general-purpose multi-core system. MICA maps data requests directly from the client to specific CPU cores at the server NIC level. It uses client-supplied information and adopts a light-weight network stack that bypasses the kernel. Additionally, Wu et al. propose a key–value cache with both higher performance and fewer misses called zExpander [27]. They also then propose another novel NVM-based key–value cache called NVMcached [28].

It is worth noting that, RHKV does not use RDMA READs directly. It leverages RDMA messaging to establish communication model. Thus, the functionality of RDMA communication model can be easily invoked. This process does not involve too much complex data operations with pointers on memory. Additionally, the implementation of RDMA communication model is encapsulated in an independent module, so as to obtain later maintenance for programmers conveniently.

2.3. Transaction services for NoSQL databases

It has been known that an excellent NoSQL key–value database (e.g., Cassandra [17]) should have basic characters of efficiency, high availability and scalability. NoSQL databases follow different design principles than those of relational databases [7,9,10]. Compared with the usage of SQL language in relational database, the usage of NoSQL query language has been simplified to Get/Put operations in most cases, and seldom Update/Delete operations. Importantly, NoSQL databases prioritize availability over consistency.

Transactional data services of data-intensive computing applications are extremely vital to NoSQL databases [15,21–23]. It is common that data-intensive computing applications should implement data transactions in NoSQL databases. Such data-intensive computing services should be implemented in three different layers, i.e., client layer, middleware layer and data store layer. In client layer, programmable APIs are developed for receiving data requests of the client's data-intensive computing applications. Data management is conducted in data store layer. Concurrent control and ACID properties are managed in the data store simultaneously. In the middleware layer, such approaches are implemented with transactional services at middleware level, which is an interface between clients and database. At the same time, network communication is conducted in this layer.

Inspired from the above related work, our proposed key–value data store RHKV aims to exploit advanced techniques to ensure serialization and concurrency of data operations on the key–value data store. Moreover, locking technique is leveraged for ensuring data consistency [11]. In summary, existing key–value data stores use different techniques to ensure concurrency, consistency and availability.

2.4. Data-intensive computing for NoSQL databases

Recently, distributed computing represents a paradigm for the process of provision computing resources and infrastructures. Users are able to access suitable application services from anywhere in the world pervasively and remotely, thus improving

the inter-operability. It shifts the location of needed computing resources and infrastructure to network [29]. In the past years, MapReduce has represented the de facto standard of big data technologies. Hadoop gains the reputation for being the most essential big data technology. Most notably, in-memory computing systems, e.g., S4 [30], Hadoop [31], Spark [32], Storm [33] and Piccolo [34], emerge as a response to the ever-increasing amount of data. There exist some challenges of improving or re-designing the bottom data store.

Data-intensive computing applications have complex context of data processing business, and provide the expected outcome [1]. Upon concurrent requests for the bottom NoSQL databases, data-intensive computing programs should be naturally integrated with databases. Currently, there exist some literatures on data-intensive computing systems based on NoSQL key–value databases.

Data-intensive computing systems which are built based on NoSQL key–value databases have been used in many fields. For instance, in smart cities, data-intensive computing applications can be used to deal with traffic data and weather data, etc [35]. Meanwhile, in the context of a data-intensive computing application, the concrete application can be related with concrete entities, e.g., computing entities and application entities. In critical infrastructures, the social context can be related with people and/or personality business, etc [29]. In mobile commerce's data-intensive applications, a context could refer to different aspects about the system's information and the physical device's information [36].

Additionally, as storage technology is quickly developing, the relative research of data processing is conducted on basis of in-memory computing systems. The main goal is to accelerate the speed of data operations. For instance, energy-aware data allocation (EADA) and balancing data allocation are proposed to energy and write operations (BDAEW), to perform data allocation to different memories and task mapping to different cores, thus reducing energy consumption and latency effectively [37]. Moreover, a method of task scheduling and data assignment on heterogeneous hybrid memory multi-processor systems for real-time applications is proposed in the literature [38].

3. Prior knowledge and preliminaries

3.1. Conventional Cuckoo hashing

Cuckoo hashing.[1] If existing d ($d > 0$) hash tables, we let $\mathcal{K}$ be the set of keys in terms of key–value data items. When $d = 2$ (by default), two hash tables are utilized in Cuckoo hashing. In this way, hash table T_1 and hash table T_2 has n data items, respectively. We give out two independent functions h_1, h_2: $\mathcal{K} \rightarrow \{0, \ldots, \text{n-1}\}$. As shown in Fig. 1, *ka*, *kb*, ...and *kf* represent keys, while *TEST*1, *TEST*2, ...and *TEST*6 represent values, correspondingly. A key $k \in \mathcal{K}$ can be inserted in either slot (bucket position) $h_1(k)$ of T_1 or slot $h_2(k)$ of T_2.

The default Cuckoo hashing scheme ($d = 2$) uses two hash functions. Thus, a key–value data item has two candidate positions for data insertion. By substituting the key into hash function, two hash values are generated. If one of two positions is empty, the key is inserted directly. It is worth noticing that, candidate positions of a vertex cannot belong to the same hash table. Actually, if two empty candidate positions in hash tables are all empty, one data item to be inserted can be used for the data item's insertion at last. But, sometimes, one or two of the candidate positions would be possessed by other data items. As a result, one or zero position can be used for the data item's insertion.

Cuckoo hashing scheme has primitive data operations, e.g., SET, GET and UPDATE, etc. During conducting the key–value data item pair's SET operation, the server would calculate out the candidate locations for the data item to be inserted. It ensures that the locations of a key which stay in at one or two hash tables. Once positions are occupied, several step-by-step kick-out operations would be done subsequently. It leads to moving out a key–value pair to one of that key's alternative locations.

[1] Cuckoo hashing. https://en.wikipedia.org/wiki/Cuckoo_hashing.

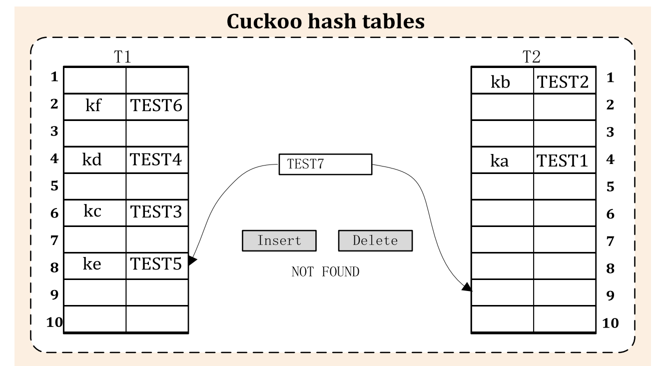


Fig. 1. The default Cuckoo hashing data management pattern.

The reason why we choose to use Cuckoo hashing scheme is that, Cuckoo hashing scheme is a form of open addressing and each cell of a hash table has a key or key–value pair. A hash function is used to determine the location for each key. The presence in the table is found by examining that cell in the table. Deletions and lookups may be performed by blanking the basic cell containing a key, in constant worst case time, more simply than other schemes such as linear probing.

3.2. Advanced hardware features

(1) Infiniband RDMA. InfiniBand and RoCE (RDMA over Converged Ethernet) are the most popular manners in RDMA-providing interconnects [11,12]. They provide low latency for data operations or data transmissions by implementing several layers of network stack. RDMA NICs can provide reliable delivery by employing hardware-based retransmission of lost packets [11]. The verbs form a semantic definition of the interface which is provided by RDMA NICs, thus realizing message exchanges in user-space. It is the main difference from Ethernet network NICs. RDMA-capable NICs provide commutation patterns, including *two-sided* channel semantics and *one-sided* memory semantics bypassing target CPU [10]. The features of them are depicted in Table 1.

(2) HTM. HTM has become an effective way to ensure the atomicity of data operations [11,12]. Isolation and atomicity, consistency are three basic features for HTM [11]. Intel's RTM provides atomicity within a machine, and detects conflicts. RTM provides interfaces including `XBEGIN`, `XEND` and `XABORT` [12]. They indicate *begin*, *end* and *abort*, respectively. Nevertheless, HLE is easier to be integrated with the current x86 software. Meanwhile, it is able to be compatible with x86 microprocessor that does not have TSX (**T**ransactional **S**ynchronization E**x**tensions) in Intel Haswell. It can avoid the conflicts of requesting the shared data, and improve the synchronization performance.

Transactional memory tries to simplify concurrent programming by allowing to execute *load* instructions and *store* instructions. Transactional memory systems provide high-level abstraction as an alternative to low-level thread synchronization. This abstraction allows for coordination between concurrent reads and writes of shared data in parallel systems.

Table 1
The features of channel semantics and memory semantics.

Semantics	Features
Channel semantics	RDMA SEND and RECV are two main manners and they are *two-sided*. A SEND's payload can be written remotely, which is specified by the responder in pre-posted RECV. CPU at the responder should post a RECV.
Memory semantics	RDMA READs and WRITEs are two main manners and they are *one-sided*. The responder's CPU can be unaware of data operations.

Note. "two-sided" indicates that each SEND operation requires a matching RECV operation at remote process. The responder's CPU is involved.

3.3. Bottleneck analysis

In RHKV, the time latency of each data operation is contributed by two parts:

(1) Forward the data request from client-side to server-side and return the result via Infiniband RDMA communication with inter-node messages.

(2) I/O overhead in server-side: position searching such that the position of key–value pair data item can be updated, read or written.

Given a request R of data operation, we denote the latency of data operation R by $\Gamma(R)$. The goal is to let the latency of requesting data operation R with minimized $\Gamma(R)$.

4. Design of RHKV

This section provides the detailed design of key–value data store RHKV with HTM/RDMA. Due to the complexity of RHKV involves the design and development of several components or modules which include: the design of a data management scheme, the design of RDMA-based network communication engine and the design of data operation's atomicity guarantee.

Although HTM/RDMA design has been proposed as in-memory transaction processing system [11]. In this paper, such design is leveraged for constructing the pure key–value data store. The logic of transaction processing system is actually stronger than that of key–value data store. Compared with the conventional Cuckoo hashing, we add a new design mechanism of Cuckoo hashing into our designed key–value system, so as to obtain the complete key–value data storage system. In this system, the consistency mechanism of data management is embedded successfully. Also, in RHKV, we add RDMA communication model to conduct data access. We hope that such statement can help understand the contributions of this paper better.

We design a novel data management scheme for our RDMA-based key–value database. In particular, it is constructed based on Cuckoo hashing scheme. Unlike the existing conventional hashing scheme, we improve Cuckoo hashing scheme and then propose the G-Cuckoo hashing scheme based on Cuckoo graph by bucket-vertex mapping.

The RDMA-based network communication engine is implemented as a separated communication layer over RDMA-based key–value store RHKV between server-side and client-side. The atomicity of data operation is ensured by HTM technique in RHKV.

The evaluation of RHKV's performance has been carried out by using YCSB benchmark. We set different types of workloads that consist of GET/UPDATE and GET/INSERT ratios and with the request distribution of Zipfian and Uniform.

The following subsections explain the detailed design and the development of different components/modules of RHKV.

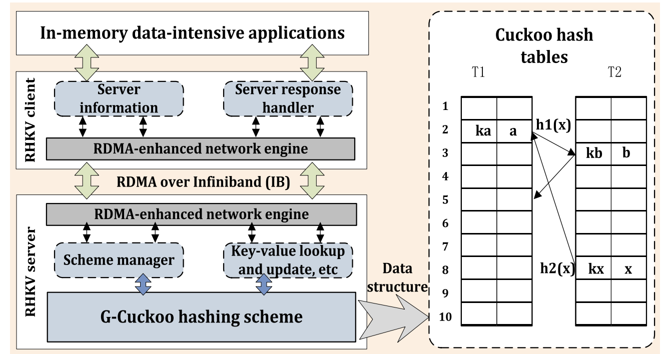


Fig. 2. RHKV is utilized as a data store layer to mitigate I/O bottleneck and exploit the benefits of RDMA-capable networks; $T1$ and $T2$ refer to Cuckoo hash tables. RHKV serves I/O requests by providing software interfaces with basic CRUD operations. These exposed interfaces allow multi-threaded data-intensive computing applications' requests concurrently.

4.1. Architecture design

RHKV is deployed on clustering environment that is equipped with high-performance network of InfiniBand, the performance benefits in infrastructures. Such data store can take responsibility to prefetch data input. RHKV serves I/O requests that are from upper-layer's data analysis applications (see Fig. 2).

Server-side layer: RHKV manages keys and values in hash tables, and leverages two-sided RDMA SEND/RECV verbs for data messaging. All the data is maintained in the aggregated DRAM of server and served to clients across RDMA-capable networks. Moreover, RHKV exploits the computing power from server through adopting HTM-awareness along with RDMA-friendly compact hash tables. In Fig. 2, one edge exists between the actual position and backup position. The start point of an edge represents the actual storage of a data item and the end point is the backup position. The hashing scheme design adopts two static hash tables and thus two hash functions $h_1(x)$ and $h_2(x)$ correspondingly. When the data request is sent to the server, the control information is transmitted in advanced. Upon receiving the message, the server performs a read operation or a write operation to fetch the payload for the client.

Client-side layer: Client sides locate key–value pairs according to our improved Cuckoo hashing scheme – G-Cuckoo. It receives every data request from the client (user) that needs to be executed. There can be numerous of clients that perform data access on the server's data store.

4.2. Endless loop problem analysis

Before introducing our improved G-Cuckoo hashing scheme, we first analyze the cases of inserting key–value pair between two hash tables on basis of the conventional Cuckoo hashing scheme.

We next try to give out a proof and a complexity analysis of the possible kick-out infinite-loop problem, when data insertions are conducted in the conventional Cuckoo hashing scheme. So, the motivation of our improved work about hashing scheme is presented.

In Fig. 3, there are two hash tables, i.e., hash table T_1 and hash table T_2. Given a key–value data item (kx, x), this data item can be plugged into two independent functions, h_1 and h_2. Thus, two candidate positions are calculated out at last. If these two positions are all empty, one of them is chosen for the data item's insertion. Actually, the actual storage and backup position of an item are ascertained randomly. Nevertheless, if only one position is empty,

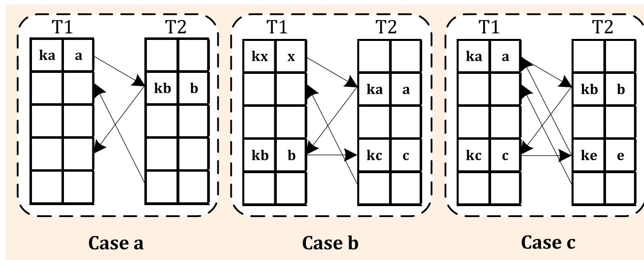


Fig. 3. Insertion cases (*Case a*, *Case b*, *Case c*) on basis of the conventional G-Cuckoo hashing scheme.

the empty position is chosen for the data item's insertion. Moreover, a bad situation is that all of them are occupied by data items. In this situation, the action of kicking out data items between two basic hash tables is triggered.

There exist three cases of inserting data item (kx, x) in total, as shown in Fig. 3. We label key–value pair (ki, i) as $T_j[i]$ $(j = 1, 2.$ $i = 0, 1, 2 \ldots)$.

Normal insertion. When finding vacant positions among hash tables for one data item's insertion, there exist two situations: The one is that the vacant bucket can be found directly, and the other is that the vacant bucket can be found after conducting several kick-out operations.

(1) As shown in *Case a* of Fig. 3, there are two data items, key–value pair (ka, a) and (kb, b), which have been inserted into hash tables $(d = 2)$. They are located in $T_1[0]$ and $T_2[1]$, respectively. When inserting data item (kx, x), two candidate positions of this data item that are calculated by hash functions are all empty (*existing vacant buckets*), they do not locate in $T_1[0]$ and $T_2[1]$. Thus, data item (kx, x) is directly inserted into $T_2[4]$. An edge is added pointing from $T_2[4]$ to the backup position $T_1[1]$.

(2) On basis of *Case a*, when inserting a new key–value pair (kx, x), after calculating the locations for inserting data, two candidate positions are occupied by data item (ka, a) and (kb, b). In this case, the action of kicking out between two hash tables shall be taken to find a vacant bucket. Specifically, after conducting step-by-step kick-out operations, we see from *Case b* that, the original key–value pair item (ka, a) in table $T1$ is kicked into table $T2$, and the original item (kb, b) in table $T2$ is kicked into table $T1$. The original item in $T_1[3]$ is kicked into $T_2[3]$. The key–value item (kx, x) is inserted into the original key–value item in $T_1[0]$.

Endless loop. There is a bad situation that, even though the action of kicking out data items has been taken, there are no available vacant bucket positions to store data item (kx, x). For example, there exists an endless loop in *Case c* of Fig. 3, $T_1[0]$, $T_1[3]$, $T_2[1]$ and $T_2[3]$ construct an endless loop between $T1$ and $T2$. Once the situation that inserting key–value data item pair in these four candidate positions, the endless loop occurs.

To conclude, the conventional Cuckoo hashing scheme adopts the random walk to find a vacant bucket when inserting a data item. It might lead to the result of **endless loops** and **performance degradation**. Note that, the classic mitigation usually stops after conducting a number of kick-out operations with respect to the table size. Switching to an in-place re-hashing, it might waste much time for kick-out actions, thus dealing with endless loops inefficiently.

4.3. Core G-Cuckoo hashing scheme

In order to get a better Cuckoo hashing scheme, we detect endless loops successfully. Thus, we inherit the good virtue of the conventional Cuckoo hashing, and in further take the kick-out action with predetermination into careful consideration. Cuckoo hashing scheme has two hash tables when $d = 2$. Each position for data insertion has a backup position, so there exist edges between hash tables. Therefore, we propose an improved Cuckoo hashing scheme named ***G-Cuckoo*** through bucket-vertex mapping and *Cuckoo graph*'s construction. In the following, we give out some definitions.

Definition 1 (*Cuckoo Graph*)**.** The positions between two hash tables constitute a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ are the sets of vertices and edges, respectively. Every vertex in the graph has a unique vertex identifier. The set of vertices can be divided into subgraphs.

Definition 2 (*Cuckoo Subgraph*)**.** A subgraph of $\mathcal{G}$ is formally denoted by $\mathcal{P}_i = (V_i, E_i)$, where $V_i \subseteq \mathcal{V}$ and $E_i = \{(v_i, v_j) \in \mathcal{E} \mid v_i \in V_i\}$. Note that, E_i includes all outgoing edges from vertices in V_i, which may cross subgraphs. All subgraphs are distributed among hash tables. For ease of illustration, we denote by ϕ_p a mapping: it records the information of which vertex belongs to which subgraph.

Bucket-vertex mapping. When designing RHKV by using Cuckoo hashing, we construct Cuckoo graph by bucket-vertex mapping between two hash tables, which is one of the innovations in the context of work. Inserted key–value data items in Cuckoo hash tables form a total Cuckoo graph. Next, we statement how the graph is created.

The graph consists of directed edges, and kick-out actions are conducted for data item's insertion. We represent Cuckoo graph as a directed pseudoforest. Each vertex of the pseudoforest corresponds to a bucket of hash tables, and each edge locates between two candidate positions of data item.

It is easily known that, in Cuckoo graph, because Cuckoo graph is constructed by mapping from buckets (data items) that are located in two Cuckoo hash tables. Each position for data insertion has a backup position, so there exist edges between two hash tables. The data item that is located in Cuckoo hash tables has at most one edge which points out to a backup position. Therefore, after conducting bucket-vertex mapping, the vertex has at most one outdegree. Vertices of outdegree of zero represent vacant positions (buckets) in the directed pseudoforest. Meanwhile, there are no vertices who have an out degree of more than one. So next, we give the basic definition of maximal and non-maximal directed pseudoforest in Cuckoo graph.

In overall, because of the bucket-to-vertex mapping, the possibility of a loop prior for data insertion can be detected. Through using the mapping, the complex data operations based on G-hashing scheme can be more intuitively.

Definition 3 (*Maximal and Non-maximal Directed Pseudoforest*)**.** In Cuckoo graph, a maximal directed pseudoforest is a directed graph in which each vertex has an outdegree of exactly one. The non-maximal directed pseudoforest can be transformed to a maximal directed pseudoforest by connecting any vertex whose outdegree is zero with any other vertex in the graph by adding a new edge.

A maximal subgraph has no room to admit a new edge (see Fig. 4). It causes an endless loop when the directed edges are transferred. There are no endless loops in a non-maximal subgraph. This subgraph does not contain a cycle. Thus, a new edge can be added into a non-maximal subgraph, because there are no cycles.

The action of just adding a new edge is actually a logic operation based on the constructed Cuckoo graph. As so far, we could not identify that such data operation can impact the final result consequentially. In this paper, after conducting the experiment, we only explore the final result of the speed of data operations (e.g., putting

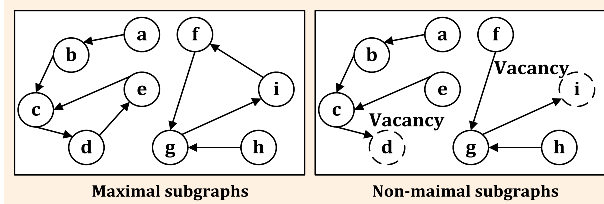


Fig. 4. Maximal and non-maximal directed pseudoforest.

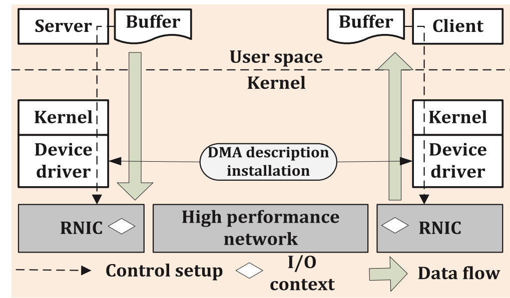


Fig. 5. RDMA communication model in RHKV.

and getting data items on the data store). Even if such operation might bring a little impact on the final result, but we would conduct the experiment to verify that, the speedup of operating data on the data store is effective by leveraging the G-Cuckoo hashing scheme.

Furthermore, after the above definitions are given, we introduce the main principle of core G-Cuckoo hashing scheme. The bucket-to-vertex mapping is prepared for the data item's insertion. In RHKV, there exists no bucket-to-vertex mapping's maintenance for read-only or update-only workloads. The server maintains a bucket-to-vertex mapping and pre-determines the possibility of a loop prior for data insertion. When inserting a key–value pair, two candidate positions are occupied by two data items, so there are no vacant positions in a subgraph. In this way, the key–value pair should not be inserted. Successful data item's insertion depends on looking up a vacant bucket for data storage. One of candidate buckets of a key–value data item to be inserted should belong to a subgraph containing one vertex whose outdegree is zero, corresponding to a vacant slot in hash table. It is critical to detect vacancies at the data item's insertion.

In the core G-Cuckoo hashing scheme, it is easy to read or update a key–value pair in hash tables. During this process, it just copes with the two-dimensional key–value data table in the inner design. Note that, there is no certain relationship between mapping and hit ratio. Actually, they do not matter so much.

In fact, knowing the path of kicking out action (finding empty positions) for a data item's insertion would help to avoid an endless kick-out loop. To find a vacant vertex from the directed pseudo-forest for data insertion, at least one candidate bucket for the data item's insertion contain a vacant position. During this process, each subgraph should be identified that it is a non-maximal subgraph or a maximal subgraph. Finding a vacant position and predetermination of an endless loop will be introduced, so as to bring in the server's I/O enhancement. We design a strategy on the selection of a path, which leads to finding a vacant position for the data item's insertion. For more details, we introduce them in Section 4.7.

4.4. RDMA network engine of RHKV

Once receiving a data request, RHKV will transmit the data request by using the constructed RDMA network engine, and interact with G-Cuckoo hashing scheme. At last, RHKV finds a vacant position for data insertions or other data operations. Hence, in this subsection, we introduce RDMA network engine of RHKV for data messaging.

Fig. 5 reports RDMA communication model in RHKV. In this model, user space leverages kernel's functions and device driver to invoker NICs. The device driver is established based on DMA description installation. The server and client leverage buffer to manage data, and then communicate data with kernel. In the bottom, RNIC and high performance network provide basic work for this model.

A Queue Pair is utilized for data communication. QP consists of a Send Queue and a Receive Queue. Completion Queues (CQs) are used for notification of completion of send and receive operation. User level communication can be utilized by mapping queues to user memory space. In a message transferring operation, a program creates QPs, then it transfers a message to the other computing node by requesting a Work Queue Request to a Send Work Queue. Receive buffers are provided by using WQRs to a Receive Work Queue.

Program in user space accesses RDMA NICs by using functions called verbs. RDMA verbs are posted by *Online* and *Offline* data-intensive computing applications to queues. These queues are maintained inside RDMA NICs. Queue exists in pairs of two important forms: *a send queue* and *a receive queue*. [10]. The RNICs fill in *a completion queue* (CQ) of each queue pair to complete verb execution. One queue pair can communicate with other queue pairs in an unconnected transport, and RDMA can provide message verbs in technique. The RDMA NICs maintain the state for each active queue in the queue pair context cache. RDMA hosts communicate by using queue pairs (QPs) [10]; and hosts creates QPs. We call the host initiating a verb as the requester, and the destination host as the responder. In clustering communication environment, reliable communication is required, therefore, Reliable Connection and Reliable Datagram can be utilized in transport service types of InfiniBand.

There are two main *messaging communication operations*:

(1) Messaging requests. Clients post GET and PUT requests to G-Cuckoo hashing scheme within RHKV server. Specifically, a GET request consists a key, while a PUT request contains a key and a value. A network configuration uses all-to-all, bidirectional communication with connected transports. It would require queue pairs at the server. An initialized process creates the request region, registers it with RDMA NICs, establishes an Unreliable Connection with each client and goes to sleep.

(2) Messaging responses. In RHKV, responses are sent as SENDs over Unreliable Datagram. Each client creates Unreliable Datagram queue pairs (QPs), whereas each server process uses only one Unreliable Datagram QP. Before writing a new data request to server process, a client posts a RECV to Unreliable Datagram QP. This RECV gets messages on the client where the server's response will be sent back. After writing out data requests, the client starts to check for responses by polling for RECV completions.

Using callback handlers via active messages has also been explored. While RHKV uses it two-sided UD verbs differently. To be concrete, when constructing IB connection, IB-context is established. When IB connection is freed, IB-context is destroyed. RHKV saves IB_Connection in session, and sets if application uses dynamic- or static-sized buffer. If connection is fixed, local and remote memory can already be allocated. RHKV allocates local and remote RDMA memory. RHKV sets *sendMessage* according to just registered MR, and sends message with RDMA details. RHKV changes Send-State to SS_MSG_SENT, and resets send and receive state of ib_connection. When the message communication process is completed, RHKV stops listening to incoming messages, and lists connections contained in IB Connection Array.

4.5. Interaction with hash tables in G-Cuckoo hashing scheme

It is vital to describe the interaction with the improved G-Cuckoo hashing scheme. We have designed basic functions of *Infiniband Server* and *Infiniband Client* in RHKV for RDMA communication. *InfiniBand Server* registers to leverage the network card. *InfiniBand Client* obtains the corresponding registration keys when it first establishes a connection to *InfiniBand Server*. For instance, when a read operation is required, *Infiniband Client* looks up a key in hash table array. Then, the client fetches one hash table's entry corresponding to the key.

The process of data communication. To allow reading data items in RDMA friendly hash tables, the server exposes the invoking interfaces for operating hash tables. The operations for data communication between server-side and client-side are shown:

1. *Server* registers *callback handler*, and defines how the incoming messages would be handled. It binds and starts to listen the incoming client connection. The connection would be stopped and then unbinded to listen the incoming client's connection.
2. *Client* sets the host node's information (i.e., host name or IP address, port-number where *Infiniband Server* listens) that *Client* should connect to.

Moreover, in RHKV, a *constructor* and a *destructor* are given out in the Infiniband RDMA communication component, initializing and freeing Infiniband connection, respectively. We define function *connect* to add connection array. *Infiniband Server* communicates with *Infiniband Client* by exchanging QP status. *InfiniBand::send* is defined to send messages in buffer form. Message is sent with RDMA details and send-state is hung to SS_MSG_SENT. *InfiniBand::recv* is defined to receive messages in buffer form. In Infiniband RDMA communication component, we use function *init*, *insert* and *free* to operate Infiniband connection array. The payload of the request is of type *std::vector char*. The content to be sent by the client has to be taken into *a char vector*. It is encapsulated in messages for communication. Function *bind* requires a valid callback handler which is called when a message is received.

4.6. HTM-awareness strong atomicity

Data-intensive computing applications can access key–value pairs in hash tables within RHKV server. Nevertheless, when RHKV is confronting with concurrent data requests, there might exist lock contention and requesting conflict problem. It would cause synchronization problem. Fierce races are caused by the concurrent RDMA accessed to hashing scheme. To ensure the atomicity of data operations, each data operation on hash tables needs to lock the key–value record beforehead.

The operations of shared data in hashing scheme can be handled by two-phrased locking-like technique or snapshot isolation. Our RDMA-based key–value data store uses the former technique which can deal with the concurrency. Once write operations or read operations can manipulate the shared data at the same time, it may lead to write–write conflicts and read–write conflicts. We attempt to control concurrent data access, so as to prevent data constraint violations, e.g., read skew, write skew.

Once data operation's atomicity is not guaranteed, it would bring about data inconsistency problem while conducting concurrent data requests. The operation is on basis of Cuckoo graph's complex operations in inner design. Thus, one basic data operation, e.g., UPDATE, should be accomplished in one time interval, but one new request would come.

The new data request operates each key–value data item simultaneously. With the help of HTM-aware strong atomicity, such situation will not occur. During the execution of program, each basic data request will not be disturbed by other data requests. Before HTM guarded section is entered, it is vital to acquire lock for each data operation.

Table 2
The results of inserting a data item without increasing vertex count in G-Cuckoo hashing scheme (Case C).

Case C	The positions of two candidate buckets of the item	Results after inserting the data item
C(1)	All are in *two different non-maximal directed* subgraphs, or *the same non-maximal directed subgraph.*	Each bucket can be the candidate position for insertion, because kick-out operation can be conducted. The original non-maximal graph could be transformed into a maximal graph.
C(2)	One is in *the non-maximal directed subgraph,* the other is in *the maximal directed subgraph.*	The item is inserted into the non-maximal subgraph. Once it is inserted into the maximal subgraph, it leads to endless kick-out loops. After insertion, subgraphs might be merged.
C(3)	All are in *the same maximal directed subgraph*, or separated into *two maximal directed subgraphs.*	The data insertion would turn out a failure. The kick-out action in an endless loop is triggered.

Once the data is accessed by an *Infiniband client*, and it is intended to be modified to another value, the code snippet about data operation should be added a lock, before returning the result to the client. There have been approaches to implement concurrency control, e.g., two phase locking, commit ordering and timestamp [11]. Leis et al. combine time-stamp ordering with HTM to provide scalable transactions [39].

Strong atomicity. RHKV leverages Intel's HLE as an useful mechanism to implement RHKV's atomicity for data operations. Actually, HLE imports two new instructions, `XACQUIRE` and `XRELEASE`. Among them, `XACQUIRE` instruction is used to acquire the lock, representing the lock region's beginning. `XRELEASE` is utilized to release the lock, representing the lock region's releasing. To ensure data operation's atomicity, lock operation and unlock operation are defined as `ACQUIRE_LOCK()` and `RELEASE_LOCK()`, respectively. Such these functions contain basic instructions, i.e., `XACQUIRE` and `XRELEASE`.

```
1 #define RELEASE_LOCK()  __atomic_store_n (&lock_var, 0, __ATOMIC_RELEASE |
        __ATOMIC_HLE_RELEASE)
2 #define ACQUIRE_LOCK() {
3  do {
4   while(lock_var == 1) {
5    _mm_pause;
6   }
7  } while(__atomic_exchange_n(&lock_var, 1, __ATOMIC_ACQUIRE |
        __ATOMIC_HLE_ACQUIRE) == 1);
8 }
```

RHKV has serializable transaction [11,12], which can be organized into three steps with two-phase locking (2PL)-like protocol, which contains *Start* phase, *LocalTX* phase and *Commit* phase. An HTM transaction firstly locks and prefetches required remote records. An HTM transaction provides transactional read and write for all local records. An HTM transaction for data operations is committed by using `XRELEASE`, and then all remote records are updated and then unlocked.

Before committing an HTM region, a transaction can log all updates of both local and remote records [11]. These records can

be utilized for recovery by writing on the target machines. If the machine crashes before the HTM commits (e.g., XEND), it means that the transaction is not committed, due to the all-or-nothing property of HTM. The lock-ahead log can be utilized to unlock remote records of recovery process when necessary.

4.7. Finding a vacant position and predetermination of an endless loop

The outcome of the data item's insertion into G-Cuckoo hashing scheme based on subgraphs, which incurs endless kick-out loops, must be pre-determined. Searching a vacant position is a vital step for predetermining the endless kick-out loop between hash tables.

4.7.1. Finding a vacant position

Key–value pairs would be inserted into G-Cuckoo hashing scheme. Once key–value data items should be inserted into hash tables successfully, the vacant positions are found. At least one of candidate positions are located in one non-maximal subgraph. In a directed pseudoforest, each vertex has an outdegree. But, the vertex with the outdegree of zero represents the vacant bucket located at the end of the directed paths in non-maximal subgraphs. When the data item is inserted into a non-maximal subgraph, it stores in one of vacant buckets. The vacant bucket corresponds to the last vertex of a directed Cuckoo path.

We have the constructed Cuckoo graph corresponding to the vertices, data insertions would lead to different results of the increasing count of vertices, as represented by $count(v) + i$, $(i = 2, 1, 0)$. Specifically, we illustrate these three cases in Fig. 6.

• **Case A:** $count(v) + 2$. Suppose that two candidate positions of a key–value pair have not yet been represented by any vertex in a directed pseudoforest, a new non-maximal directed subgraph is created during the process of data insertion. It is easily known that, the new data item can be inserted successfully. The increasing vertex count of the subgraph is 2, and the increasing edge count is 1.

• **Case B:** $count(v)+1$. There is another scenario that one of candidate positions corresponds to an existing vertex in a subgraph, and the other has not yet been represented by any vertex. The data insertion will be a success at last. In such a subgraph, the count of vertices is increased by 1 and the count of edges is also increased by 1. A new edge connecting the new vertex with the existing vertex is added into the subgraph of the directed pseudoforest.

• **Case C:** $count(v) + 0$. When inserting a data item without increasing vertex count, there exist three results. We summarize them in Table 2 with three results.

4.7.2. Pre-determination

It is critical to know which the above mentioned case that a data item insertion would result in. It is easily known that, when the data item's insertion confronts with the case of $count(v) + 0$, the data item's insertion will not directly incur the endless kick-out loops. In the scenario of $count(v) + 0$, both candidate positions are vertices in subgraphs. The edge count of the corresponding subgraph is increased by 1, and the count of vertices does not increase at all. Only if at least one of subgraphs is non-maximal, the data item's insertion is conducted. Otherwise, the data item's insertion would fail after walking into an endless loop.

The insertion process of key–value pair is listed in Algorithm 1, which includes predetermination steps for the endless loops.

Avoiding endless kick-out loops. Importantly, the status of each subgraph should be detected and monitored to identify whether any of the key–value data item's candidate positions is on an endless loop. To predetermine the outcome of the key–value data item's insertion, we need to know the status of subgraphs timely. That is to say, the subgraph should be identified of checking whether it is maximal or non-maximal.

Algorithm 1 Inserting key–value pair into G-Cuckoo.

Input: Key–value pair to be inserted $x = (kx, x)$.
Output: The execution state of insertion, *Boolean*.

```
1: /* the candidate position p1, p2 of data item x = (kx, x) */
2: p1 ← Hash_func_1 (kx);
3: p2 ← Hash_func_2 (kx);
4: /* pre-determine the result of the increasing count of vertices */
5: c ← Predetermine (p1, p2);
6: if (c == count (v) + 2) then
7:    Do the new subgraph operation;
8:    /* add the edge between p1 and p2 */
9:    Union (p1, p2);
10:   /* insert item x into the empty bucket */
11:   Insert (x, p1, p2);
12:   return true;
13: else if (c == count (v) + 1) then
14:   /* subgraph unioning operation and data inserting operation */
15:   Union (p1, p2);
16:   Insert (x, p1, p2);
17:   return true;
18: else
19:   /* do kick-out operations and then add the insert item */
20:   KickAndInsert (x, p1, p2);
21: end if
```

4.8. Optimization: Consistency mechanism

Inconsistency scenarios. When operating data on compact hash tables, the halfway completed operation might cause the key–value data store to stay in an inconsistency status. There are several potential inconsistency scenarios as below: (1) Initializing hash tables; (2) Rehashing; (3) Inserting or updating a key–value data item; (4) Conducting InfiniBand RDMA communication; (5) Deleting an existing key.

Traditionally, a simple way is to apply the commercial Infiniband network to pull the newest data in server periodically until the status changes. However, in RHKV, a callback operation is used to notify a client when a specified value changes. Within a given period of time, if *Infiniband Server* finds the value being changed to the expected value, it returns successful signal to *Infiniband Client*; otherwise, it returns failed signal to *Infiniband Client*.

Key–value data storages prioritize efficiency/availability over data consistency. An efficient offline/online checkpointing strategy has been proposed to apply in the hardware/software co-design nonvolatile FPGA [40]. As for data recovery, RHKV uses *append operation* to ensure the versioned values for one key. Only one version of data is marked as current and active, while in append primitives, all data fields are in current version. When the original data should be obtained, the original value could be recovered soon. The append operation is required to support lock-free concurrent modification.

In order to ensure the accuracy of message communication between server-side and client-side, the concrete supersteps of self-verifying data structure *Checksum* [41] in RHKV are presented as below:

(1) Calculate the actual hash value of the key;

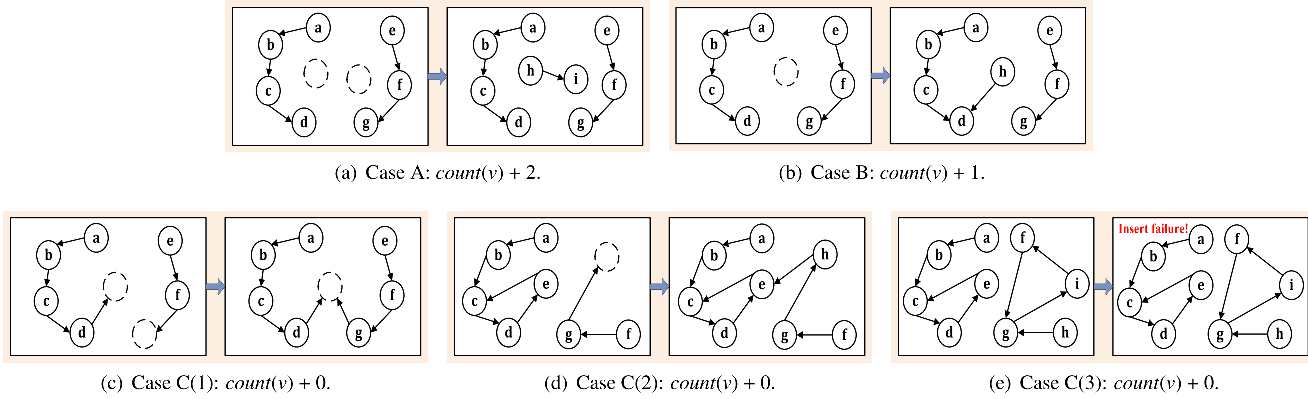
(2) Query it from the hash table to look up for the value;

(3) Get key–value pair content to calculate the Checksum value of the actual content;

(4) Compare the content with the stored Checksum in hash table's buckets.

4.9. The execution protocol

In this subsection, we describe the execution protocol with comprising several supersteps. The execution protocol between server-side and client-side is summarized in Algorithm 2.



(a) Case A: *count*(v) + 2. (b) Case B: *count*(v) + 1.

(c) Case C(1): *count*(v) + 0. (d) Case C(2): *count*(v) + 0. (e) Case C(3): *count*(v) + 0.

Fig. 6. Different results of the increasing count of vertices.

The execution protocol contains the following three supersteps:

Superstep A. RHKV client submits CRUD operation(s) from data-intensive computing applications and forwards them to RHKV's server-side.

Superstep B. Check CRUD operation(s) and forward the data request to server-side: The types of data operations (read, write, update and delete) are identified by actual business requirements of data-intensive computing applications. The protocol creates a data request for the client to the server, and identifies the request's type, such as: the type of CRUD operation ("C", "R", "U", "D"). Several APIs are designed for RDMA-based communication between server-side and client-side. The data request is forwarded to the server-side's hashing scheme. So, the data request would be sent to server-side through using RDMA-based network engine.

Superstep C. Execute CRUD operation(s): If a data operation is completed successfully, the result should be returned to the client with satisfying its expectation. It would be considered that data operations could be conducted as a result of "successful" or committed. Otherwise, data operations would be aborted. From the perspective of data-intensive computing applications, data operations would be done with meeting functional requirements. The server's hashing scheme would compute the actual result, which could be returned to the server-side by using RDMA-based network engine. It starts, executes and ends the data request for one CRUD operation.

Algorithm 2 The execution protocol between server-side and client-side.

Input: Data-intensive computing application, *App*.

Output: The execution result, *clientGet*.

1: /*initializing variables*/
2: Initialize *context*, *responseResult*, *serverGet*, *clientGet*;
3: /*capturing data-intensive computing application's context*/
4: Application *app* = new Application (*App*);
5: *context* ← *app.contextCapture* ();
6: /*creating a request set*/
7: ClientRequestSet *clientRequestSet* = new ClientRequestSet (*context*);
8: /*transport the client's request set*/
9: *serverGet* ← RDMAHelper.*transport* (*clientRequestSet*);
10: *t* ← *identifyTypeCRUDOperation* (*serverGet*);
11: Client *client* = new Client ();
12: /*executing a CRUD operation*/
13: *client.startCRUDOperation* (*t*);
14: *client.executeCRUDOperation* (*t*);
15: *responseResult* ← client.*endCRUDOperation* (*t*);
16: /*return the response result*/
17: *clientGet* ← RDMAHelper.*returnResult* (*responseResult*);
18: **return** *clientGet*;

5. Some key implementations

The prototype key–value store system named RHKV is implemented in C++ language in more than 6000 LOC. It includes some Linux shell compiling scripts. We hope our research can help provide a guiding role for researchers to achieve autonomous key–value data stores easier. Here, we introduce some key implementations.

• **Software interface abstraction.** RHKV provides software interfaces (client APIs) and consists of basic data operations. Basic data operations, i.e., function *rhkv_get*, *rhkv_set* and *rhkv_update*, are implemented. The interfaces allow multi-threaded applications to share the key–value data store concurrently.

The runtime in the low of software interface abstraction is given out as follows: In *rhkv_get*, the parameter key m is transmitted to function *Hash* through Infiniband communication network. The function calculates two position indices via using mask operation to find the actual value in two hash tables. At last, the final result is returned back. The implementation of function *rhkv_set* is more complex. Specifically, it first finds the key–value data item in stash (a data structure). If the data item exists, the function updates the key–value data item; otherwise, it tries to find the position in hash tables. It calculates the hashing positions in tables by using function *Hash*, and updates the actual value. Also, it updates the constructed Cuckoo graph synchronously. In function *rhkv_update*, similar to the principle of set operation, it gets the actual position in hash tables or stash, and updates the old value to a new one.

```
1 void rhkv_get(char* key, size_t key_length, char** value, size_t* key_length);
2 bool rhkv_set(char* key, char* value, size_t key_length, size_t value_length);
3 bool rhkv_update(char* key, size_t key_length, char** value);
```

• **Hashing scheme module.** We define a data structure of *Node* which contains variables of *appVar*, *occuVar* and *subVar* in the inner design of RHKV. Where, *appVar* is used to identify whether the subgraph is non-maximal or maximal; *occuVar* represents that this vertex in graph is occupied or not; and *subVar* stands for the subgraph identifier that the vertex locates.

It is worth noticing that, a two-dimensional key–value data table [TABLE_COUNT][MAX_SIZE] and a two-dimensional state table node [TABLE_COUNT][MAX_SIZE] are specially designed in RHKV. Specifically, the two-dimensional KV data table contains much KV data, while the two-dimensional state table contains much state of KV data in Cuckoo graph. Each state element refers to the *Node* data structure. This data structure contains three variances. The

Table 3
Several function API excerpts for G-Cuckoo hashing scheme.

RHKV APIs	Descriptions
init()	This function initializes the improved hashing scheme - G-Cuckoo, initializes graph vertices, and initializes hash seed, etc.
judgeNum()	This function explores the situation that two candidate graph vertices are located in the non-maximal or the maximal subgraph.
subGraphIsFull()	This function returns Boolean value that the subgraph is full or not.
kickOutOperation()	This function kicks out *KVPair* kv in hash table i to the other hash table j on basis of G-Cuckoo.
findSubNum()	This function returns the identifier of subgraph. The pointer of *sub_queue_front* increases.
changeSubGraph()	This function changes the number of the subgraphs, i.e., increasing or decreasing number.

first variance represents that the node connects with the non-maximal sub-graph or the maximal subgraph; the second variance represents that the node is occupied or not; the third variance represents the sub-graph's identifier which the node locates.

Each element of two data structures should be corresponding, respectively. When the state of KV data is changed, one of elements in table node [TABLE_COUNT][MAX_SIZE] should be changed correspondingly. When conducting the get or update operation, the cost of operating data structure in table node [TABLE_COUNT] [MAX_SIZE] can be ignored.

When conducting the update or get operation, we only need scanning KVPair table [TABLE_COUNT][MAX_SIZE]. In this case, there is no need to modify the element in node [TABLE_COUNT] [MAX_SIZE].

It is worth noticing that, the bucket-vertex mapping is the basis of the infinite loop prediction, which reduces the data operation's latency and the overall data operation's latency. There is no certain relationship between mapping and hit ratio. Actually, they do not matter so much.

To manage the improved hash tables based on the constructed Cuckoo graph with detecting endless kick-out loops, major functions have been defined. Some of them are shown in Table 3. The functions of *search* and *update* are realized of the concrete data operations on hash tables. Nevertheless, in the implementation of function *insert*, function *judgeNum* is invoked to identify whether each subgraph is non-maximal or maximal. Alternatively, *UnionSet* data structure is designed to conduct the merging subgraphs operation, and relevant functions to find parent of the union set. A variety of techniques have been studied to hide memory latency from intermediate fast memories (caches) to various prefetching and memory management techniques [42]. Therefore, in RHKV's inner design, we have designed `cache data structure` to store a little part of recent accessed data items. The process of the whole data operation is interpreted in the following stages:

(1) Check, load and update cache data structure;
(2) Server responses;
(3) Client waits;
(4) Result returns.

• **Infiniband communication module.** The setter and getter operations for the defined objects *network layer*, *transport layer*, *crypto layer* and *application layer* should be defined in Infiniband configuration component. The functions of *setHostName*, *setPort*, *getHostname* and *getPort* are defined in the module. Object *App termination listener* listens the input messages from client, and also sets payload for replying to the client. Specifically, Infiniband message which includes serialized metadata for communication is defined by leveraging with *std::map<std::string, std::string>::iterator metadata_iter*.

Moreover, function *server_on_completion* changes receive-state to MSG_RECEIVED and sets remote-connection accordingly to the received information. This function uses a dynamic buffer, when the server gets RDMA-read request. It passes parameters to SERVER-user, therefore, the user can smooth with the received data. After the message is dealt, this function resets *ib_connection* send and receives state. In function *client_on_completion*, if send-state is the state of MSG_SENT and receive-state is INIT, it implies that the client has received, a message from the server contains RDMA details for the client to read. When the client has read data, it changes send-state to REDMA_READ_DONE. Function *create_ib_connection* explicitly sets the pointer to application context, and allocates memory for message buffers.

6. Performance analysis on HPC cluster

6.1. Experimental setup

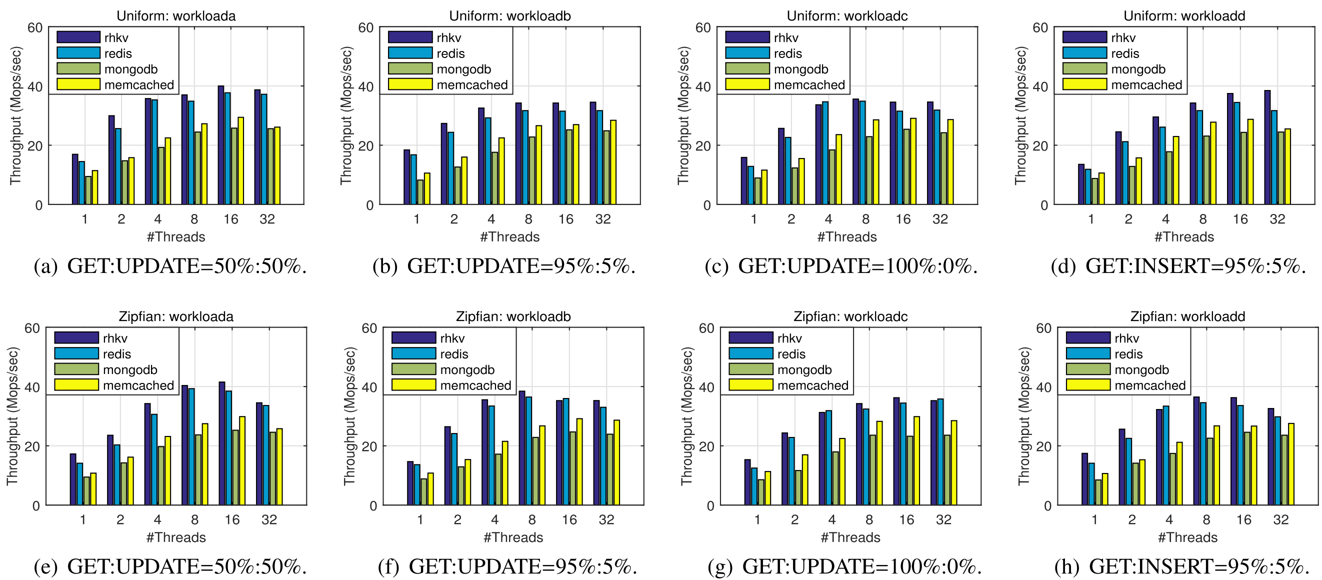
All experiments are conducted on a small-scale clustering environment with server machines in the testable environment of **Data Driven Software Technology Laboratory**, **Shanghai Jiao Tong University**. Machines are equipped with a Mellanox FDR 56Gb/s and QDR 120Gbps InfiniBand Switch (MT25408 ConnectX Mellanox Technologies). Each machine runs CentOS 6.6 with Mellanox OFED stack. The clustering environment is one server acts as server node and the other servers can act as client nodes. The machine (Power Edge Dell-R730 Server, *Intel Haswel*, 2133 MHz DDR4, *Intel TSX interface*) for acting as RHKV server has a 64-core Intel Xeon E5-2650 CPU V3 clocked at 2.30 GHz, 128 GB of memory and 16TB SATA hard disks. The ability of RHKV is systematically explored to handle concurrent workloads in comparison with key–value data stores, including *Redis*, *Memcached*, *MongoDB* and *HERD*.

Micro-benchmark. YCSB (Yahoo! Cloud Serving Benchmark) [13], which is a "real-world" NoSQL benchmark suite, is used to investigate RHKV's performance. YCSB constructs key–value data pairs with variable key and value lengths, modeled on the statistical properties of real-world workloads. When the experiments are conducted, all clients load data requests into hashing scheme before starting the performance measurement. We focus on representative types of workloads that consist of different GET/UPDATE and GET/INSERT ratios. We measure the performance when the data request's distribution follows both ***Zipfian*** distribution and ***Uniform*** distribution.

6.2. Overall throughput improvement

We make full use of two different types of access patterns: *Uniform* distribution and *Zipfian* distribution, so as to measure the overall throughput's improvement. (1) Uniform distribution refers to choosing an item uniformly at random. When choosing a record, all records in database are equally like to be chosen. Under Uniform distribution workload, there is no hotness and coldness difference among all of keys. Each data request is handled randomly, the query efficiency and system's concurrency affect the final performance greatly. (2) Zipfian distribution refers to that some records will be extremely popular (the head of the distribution), while most of these records will be unpopular. Under Zipfian distribution workload, hot keys can be buffered into memory part of each system. And they can be returned to the client with a frequent request directly.

We change different ratio workloads and multiple thread response, and use YCSB benchmark to produce different types of



(a) GET:UPDATE=50%:50%. (b) GET:UPDATE=95%:5%. (c) GET:UPDATE=100%:0%. (d) GET:INSERT=95%:5%.

(e) GET:UPDATE=50%:50%. (f) GET:UPDATE=95%:5%. (g) GET:UPDATE=100%:0%. (h) GET:INSERT=95%:5%.

Fig. 7. Throughput performance of basic data operations with different ratios (Compared with Redis, MongoDB, Memcached).

GET/UPDATE or GET/INSERT ratios. In our experiment, we set them as GET:UPDATE = **50%:50%**, **95%:5%**, **100%:0%** and GET:INSERT = **95%:5%** (i.e., *workloada, workloadb, workloadc, workloadd*), respectively. Considering that YCSB workload generation [13] can be highly CPU-intensive and time-consuming, all the workloads are pre-generated. In the experiment, one client requests the server at the same time, and each workload consists of $100 * 10^6$ operations.

We briefly introduce several commercial key–value data stores – Redis, MongoDB and Memcached for comparison in next:

(1) Redis is an open-source in-memory database implementing a distributed, in-memory key–value data store with optional durability. Redis supports different kinds of abstract data structures, e.g., strings, lists, maps, sets, sorted sets, hyperloglogs, bitmaps and spatial indexes.

(2) MongoDB is an open-source cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schemas.

(3) Memcached is a general-purpose distributed memory caching system. The system uses a client–server architecture. The servers maintain a key–value associative array; the clients populate this array and query it by key.

It is worth noticing that, data communication for Redis, MongoDB and Memcached [8] does not leverage Infiniband RDMA at all between server-side and client-side. The purpose is to evaluate the overall performance of RHKV. The experimental results of testing different key–value data stores are shown in Fig. 7. Specifically, Fig. 7(a)–(d) shows the throughput results with different workloads with values for Uniform distribution; while Fig. 7(e)–(h) reveals throughput results with different workloads with values for Zipfian distribution. Actually, we have run the experiment in the comparatively stable environment, but there might exist random errors in the actual experimental records. However, we can find that, RHKV outperforms the other three baseline key–value data stores in most cases.

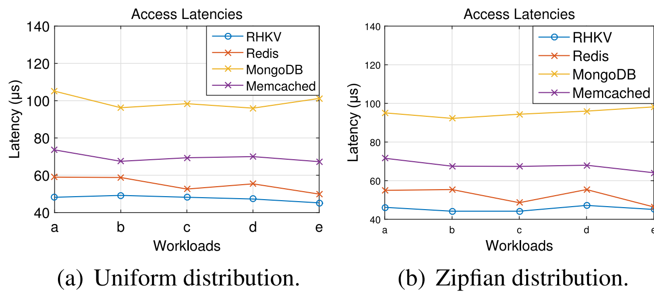
The experimental results also show the good scalability of RHKV under different kinds of workloads. The reason is that the designed G-Cuckoo hashing scheme can receive the I/O request concurrently better than other data structure for data organization, e.g., list. Also, it verifies that the mentioned network communication – Infiniband RDMA messaging verbs have the not bad performance effect which contributes to the final throughput results. We can see results from Fig. 7 that, MongoDB [16] depicts the worst performance among these mentioned above baseline key–value data stores. In absolute terms, the difference between RHKV and Redis seems minimal for most cases. Both RHKV and Redis perform excellently in data access, as compared with other two databases. Moreover, it is significant that RDMA technique brings the improvement, RHKV presents a better performance.

Moreover, we evaluate the scalability of data storage by varying different numbers of data operations. With the increasing number of operations, i.e., $100 * 10^6$, $400 * 10^6$, $700 * 10^6$, $1000 * 10^6$, we find out that RHKV's throughput outperforms than other systems as well to some extent.

The discussion of cache data structure's effect. We have designed cache data structure to store a little part of recent accessed key–value data items in RHKV's inner design. We also record the experimental results to explore the cache data structure's effect. In the experiment, we disable the cache data structure in RHKV. After conducting the experiment under a skewed workload for 20 million key–value pairs, we find that G-Cuckoo hashing scheme with cache data structure eliminates about 9.0% look up operations in hash tables. The overall performance is also subjected to the additional cache data structure's hit rate. It can be explained as below: The cost of looking up operations that are conducted on hash tables is more time-consuming and complex than the cost of that with using cache data structure. The most frequent data items are often located in such cache. The recent data has put into the cache data structure, which boosts data access. Yet, it makes the data item's insertion a little complicated. In this way, the cost of lookup operations on basis of the improved G-Cuckoo hashing scheme reduces considerably.

6.3. Access latencies reduction

To explore the access latencies reduction of RHKV, we try to use YCSB benchmark to report performance in access latencies of RHKV and baseline key–value stores with default configuration. We add Redis and Memcached for performance comparison. One client requests the server, and each workload consists of $100 * 10^6$ operations. With configuring READ/INSERT ratio as 95%:5% and following the latest distribution, Fig. 8 shows the normalized latency with leveraging RDMA communication of RHKV, as compared with the other baseline key–value data stores. Looking into the results of workloads, each key–value data store obtains the similar latency except for Memcached, when conducting the



(a) Uniform distribution. (b) Zipfian distribution.

Fig. 8. The experimental situation of access latencies reduction with varying different workloads (Unit: µs).

Table 4
The experimental situation with RDMA and TCP/IP (1 thread).

Zipfian	50%:50%		95%:5%		100%:0%	
	R	*T*	*R*	*T*	*R*	*T*
TCP/IP	15.23	43.2	12.34	41.3	14.41	39.5
RDMA	17.25	36.2	14.66	31.4	15.32	26.5

Note. R: Throughput (Mops/sec); T: Normalized latency (µs).

execution of *workloadd*. As a result, RHKV achieves about 40.0% access latency of MongoDB as the best case. In different workloads, we have observed that Redis performs nearly to RHKV. RHKV keeps the lowest latency among these key–value data stores. Note that, we have logged the running points into the code, after conducting the experiment, due to the protection of HLE technique, data operations do not abort. Meanwhile, the endless kick-out loops never happen.

Moreover, we launch multiple client processes and one key–value server on a single node, and stress key–value stores, concurrently interleaved writes and reads. Memcached's performance is worse than RHKV and Redis, because Memcached is designed for reliable cache service. It periodically checkpoints in-memory data for persistence, and more data is checkpointed than actually received. Overall, RHKV and Redis deliver both faster and higher PUT/GET performance, as compared with other two key–value data stores, i.e., Redis and Memcached. It is critical that data persistence must be implemented without heavy impact on performance in terms of PUT operations.

We next discuss RDMA communication's effect, the infinite-loop kick-out problem and the HTM's effect in detail:

The discussion of RDMA communication's effect. We change the communication manner from Infiniband RDMA to TCP/IP for RHKV by modifying the inner code of RHKV, i.e., invoking different communication functionalities. There are $100 * 10^6$ data operations, and we use 1 thread to conduct this experiment. Experiment with setting the same configuration is conducted to verify the actual effect of using Infiniband RDMA message communication, as compared with TCP/IP based network. The detailed results are depicted in Table 4.

We can see from Table 4, RHKV with *Infiniband RDMA-based communication* outperforms than that with using *TCP/IP-based manner* by varying GET/UPDATE ratios. This result of more excellent performance in RDMA-based communication environment (compared with that in TCP/IP-based communication environment) is straightforward to be understood, because communication media (Infiniband RDMA) itself boosts the communication process effectively. This is attributed to RHKV decoupled the I/O request handling process by leveraging Infiniband RDMA communication.

The discussion of infinite-loop kick-out problem. We compare G-Cuckoo with other existing solution to the infinite-loop kick-out problem, so as to verify what is the advantage of G-Cuckoo. One client requests the server, and each workload consists of $100 * 10^6$ data operations. We modify the code and compare the performance of our system with the inner designed G-Cuckoo hashing scheme and the conventional Cuckoo hashing scheme. With the setting of GET:INSERT = 80%:20%, 85%:15%, 90%:10% and 95%:5%, on average, the overall performance ratios of these access latencies reduction is 1.151 : 1.101 : 1.049: 1.000. Because the *get* operation is easier, and it is not related to what hashing scheme manner is adopted, but the *insert* operation does not.

The discussion of classic locking vs. HTM. In order to show that HTM plays an important role, we also conduct the experiment with adding classic locking and HTM. Here, the classic locking refers to the mutex variation's judgment. One client requests the server, and each workload consists of $100 * 10^6$ data operations. With the setting of GET:UPDATE = 50%:50%, 95%:5%, 100%:0% and GET:INSERT = 95%:5%, access latency reduction performs better with 10.1%, 8.7%, 8.2% and 8.6%, respectively, in terms of two different locking manners, i.e., HTM and classic locking. The performance advantage of using HTM over locks is about 10%, since the overall performance shows up to 2x improvements over Memcached, Redis, the obvious improvement is mainly contributed from using RDMA vs. TCP/IP-over-IB.

6.4. Compared with RDMA-based key–value store

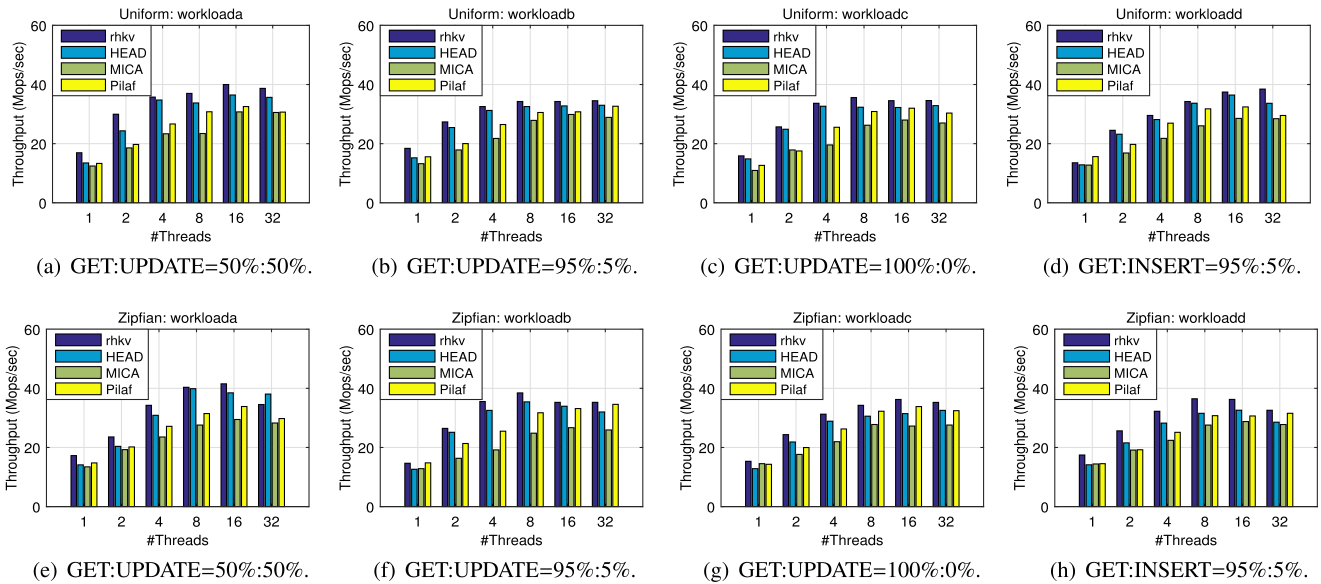
To further explore the performance of RHKV and let the experiment (the closest point of comparison) be well justified, we compare its throughput and latency with RDMA-based key–value data storages, HEAD [10], Pilaf [7] and MICA [26] with its InfiniBand support, with using the same benchmark and the similar fundamental configuration.

We test several workloads, i.e., GET:UPDATE = 50%:50%, 95%:5%, 100%:0% and GET:INSERT = 95%:5%. HERD provides average latency (3.6 µs) and throughput (5.8 Mops/sec), due to its design choice that saves one round trip but amplifies the read size. We look into the experimental results and find out that, RHKV outperforms the above baseline RDMA-based key–value store, HERD, by up to 1.09 × on average. HERD uses RDMA-write for data requests and UD for server responses to read the key–value pair after the lookup operation. Specifically, partly due to the RDMA-friendly design of hashing scheme, on average, RHKV achieves both lower latency and higher throughput by 8% and 7%, respectively. HERD has an excellent throughput for a small value, due to avoiding additional RDMA READs, but the performance significantly degrades with the increase in value size. We also test the performance of HEAD, Pilaf and MICA in the experiment. The experimental result is shown in Fig. 9.

RHKV does not directly utilize RDMA READs, data operations in server do not require too much CPU resource. Thus, on average, more CPU resource should be required for RHKV's data operations in message exchange for saving RTT (Round-Trip Time), as compared with other RDMA-based key–value storages. When RHKV handles PUT requests, it requires CPU involvement in the server. Theoretically, it might seem that RHKV should be performed better than the baseline key–value data store.

6.5. Resistance to workload skew

To understand how RHKV's behavior is impacted by workload skew, we conduct the experiment with YCSB workload where keys come from a Zipfian distribution. We tune Zipfian distribution parameters from 0.1 to 0.9 by increasing 0.1 gradually. Even we set the high parameter value, which stands for the extent of workload



(a) GET:UPDATE=50%:50%. (b) GET:UPDATE=95%:5%. (c) GET:UPDATE=100%:0%. (d) GET:INSERT=95%:5%.

(e) GET:UPDATE=50%:50%. (f) GET:UPDATE=95%:5%. (g) GET:UPDATE=100%:0%. (h) GET:INSERT=95%:5%.

Fig. 9. Throughput performance of basic data operations with different ratios (Compared with RDMA-based key–value data storages HEAD, MICA, Pilaf).

skew, it turns out that the excellent performance as well. Because CPU cores share RDMA NICs for communication, the highly loaded cores are able to benefit from the idle time. Under Zipfian distributed workload, the most loaded CPU core is a little more than the least loaded core, even though the most popular key is 91 times popular than the average. This is the factor that contributes to the resistance to skew. Briefly, it shows that RHKV plays a good effect in resistance to skew.

6.6. Space amplification

Space amplification refers to the ratio of the actual storage size to the logical storage size. To explore space amplification's situation, we also look into the space situation of RHKV with the baseline target key–value data stores.

We configure Redis to use the append-only file persistent mechanism and set the append sync parameter to every second. Thus, we use YCSB benchmark *workloadc* to explore the situation of space amplification. Theoretically, suppose that each key–value pair has a key size of 10 bytes and a value size of 100 bytes, the logical size of database is totally 1.02 GB. We explore the synchronized database size of the mentioned store every second. Consequently, we confirm that RHKV achieves a faster persist process than Redis and MongoDB, and finally consumes less space to store the same size of logical database. By using 1 thread to running *workloadc*, RHKV obtains a speed up of the persist process and a storage space savage. The actual database size for RHKV, Redis and MongoDB is 1204.25 MB, 1256.32 MB, 1230.32 MB, respectively. Therefore, space amplification of RHKV is the smallest. On average, we find out that the persist time of each data operation for them is 52.23 ms, 62.21 ms, 65.32 ms, respectively.

The reason for the result is that, fewer levels and unsorting of keys make RHKV achieve a faster persistent speed than Redis and MongoDB. Redis executes a persist process every second. It does not need to log the key to calculate data firstly, but synchronizes the key–value pair periodically. Thus, it performs a faster persist process than MongoDB.

6.7. Consistency

The consistency mechanism of RHKV is constructed based on a Checksum approach, as presented before. Each key–value data pair has to be persisted to prevent wrong data. The intuitive method of enforcing the desired write order is to use primitives such as `ciflush` and `mfence`. These primitives require the key–value data store to execute strict write ordering between data and its corresponding metadata be enforced. In this way, huge overhead is brought to key–value data storage libraries with high response speed and high throughput demand. Instead, in RHKV, we compute 32-bit Checksum of each key–value pair item content by using CRC32 hash function for data communication. Experimentally, calculating Checksum of a piece of data is about 5 $\times$ to 20 $\times$ faster than performing cache flushed on the data. The experimental result shows that RHKV does not report any consistency vulnerability. Thus, we believe that the Checksum approach is safe and reliable, and it meets our data consistency requirements basically.

In order to further explore the effectiveness of RHKV's consistency, we conduct different experiments with varying different ratios of read or write operations. RHKV server of key–value database receives information periodically. We conduct the stimulation experiment, and find out that a thousand of CRUD operations can be executed with setting different ratios. The set could be treated as a whole write unit on its own, i.e., no concurrent read/write operations are allowed in the middle of the data operation. To be concrete, different pieces of information could be written in sequential order and independently one from another, i.e., concurrent read/write operations are allowed in the middle of data operations.

Each basic data operation is not disturbed by others through using HTM technique. The reason is that, the interval of time between write operations is not big enough. It there is no HTM technique's ensurance, each data operation would be disturbed by others. The data operation stays in a not stable state before the next write operation arrives. Moreover, it is easy to understood that read operations are concurrently executed with write operations then they may provide misleading data. However, it could introduce delays and would affect the system's response. With the help of the atomicity's ensurance, the old and new data would not co-exist while data is updated in RDMA-based key–value data store.

6.8. Some illustrated application scenarios

Data-intensive computing applications and services should be aware of the context logic. The goal is to seek reliable and feasible

outcome. Any software application should be designed with a context-aware architecture. With the continuous improvement in such data store, RHKV can be used in more applicable systems, e.g., *Radio Frequency IDentification systems*. These systems integrate complex data processing logic, which needs basic data operations in RHKV. In-memory computing systems emerge as a response to the ever-increasing amount of data, e.g., S4 [30], Spark [43], Storm [33]. The I/O communication with the underlying disk-oriented storage systems, e.g., HDFS [31], or Cloud-based NoSQL stores, is still a major bottleneck. RHKV might take responsibility to prefetch data input from HDFS into a cluster and serve the I/O requests from upper-layer data-intensive computing applications, bridging the I/O gap over fast networks of MapReduce stacks. Moreover, as the number of Internet users increases, Internet advertising industry develops rapidly and the mode of real-time advertisement bidding systems [44] emerges in recent years. RHKV can be leveraged to enhance such data access, so as to satisfy the quality of services better. Key–value data stores can be applied for data processing, as they can be used to deal with data which is distributed or stored across different computing nodes of NoSQL databases.

7. Conclusion

NoSQL key–value databases have emerged as data management systems, so as to manage and process big data in a scalable and efficient manner. In this paper, we analyze the bottleneck in key–value data storage's design, due to the drawbacks of network round trips and requesting conflicts. Accordingly, we have presented RHKV, a novel RDMA and HTM-friendly key–value data store which delivers fast and scalable data services for in-memory data-intensive computing. RHKV provides the data store through using the improved G-Cuckoo hashing scheme (alleviating the endless kick-out loops between hash tables during the data item's insertion) and requires SEND/RECV verbs for remote data access. It requires read or write sets for the proper locking to implement 2PL-like protocol. Our evaluation demonstrates that, as compared with the state-of-the-art key–value stores, RHKV improves the speed of basic data operations to some extent.

Acknowledgement

This work is supported by the National Natural Science Foundation of China (NSFC) under Grant No. 61472241 and the National Key Research and Development Program of China under Grant No. 2018YFB1003302.

References

[1] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, S. Meng, HydraDB: A resilient RDMA-driven key–value middleware for in-memory cluster computing, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2015, pp. 1–11.
[2] R. Gunasekaran, S. Oral, J. Hill, R. Miller, F. Wang, D. Leverman, Comparative I/O workload characterization of two leadership class storage clusters, in: Proceedings of the 10th Parallel Data Storage Workshop, 2015, pp. 31–36.
[3] D. Shankar, X. Lu, N.S. Islam, M. Wasi-ur-Rahman, D.K. Panda, High-performance hybrid key–value store on modern clusters with RDMA interconnects and SSDs: non-blocking extensions, designs, and benefits, in: Proceedings of 2016 IEEE International Parallel and Distributed Processing Symposium, 2016, pp. 393–402.
[4] X. Lu, D. Shankar, S. Gugnani, D.K. Panda, High-performance design of Apache Spark with RDMA and its benefits on various workloads, in: Proceedings of 2016 IEEE International Conference on Big Data, 2016, pp. 253–262.
[5] X. Lu, D. Shankar, N.S. Islam, M. Wasi-ur-Rahman, J. Jose, H. Subramoni, H. Wang, D.K. Panda, High-performance design of Hadoop RPC with RDMA over InfiniBand, in: Proceedings of High-Performance Design of Hadoop RPC with RDMA over InfiniBand, 2013, pp. 641–650.
[6] Mellanox technologies. http://www.mellanox.com/.
[7] C. Mitchell, Y. Geng, J. Li, Using one-sided RDMA READs to build a fast, CPU-efficient key–value store, in: Proceedings of 2013 USENIX Annual Technical Conference, 2013, pp. 103–114.
[8] Memcached. http://memcached.org/.
[9] A. Dragojević, D. Narayanan, O. Hodson, M. Castro, O. Hodson, FaRM: fast remote memory, in: Proceedings of Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, 2014, pp. 401-414.
[10] A. Kalia, M. Kaminsky, D.G. Andersen, Using RDMA efficiently for key–value services, in: Proceedings of ACM SIGCOMM 2014 Conference, SIGCOMM'14, 2014, pp. 295–306.
[11] Y. Chen, X. Wei, J. Shi, R. Chen, H. Chen, Fast and general distributed transactions using RDMA and HTM, in: Proceedings of the Eleventh European Conference on Computer Systems, 2016, pp. 26:1–26:17.
[12] X. Wei, J. Shi, Y. Chen, R. Chen, H. Chen, Fast in-memory transaction processing using RDMA and HTM, in: Proceedings of the 25th Symposium on Operating Systems Principles, 2015, pp. 87–104.
[13] YCSB. https://en.wikipedia.org/wiki/YCSB.
[14] Redis. https://redis.io/.
[15] M.A. Olson, K. Bostic, M.I. Seltzer, Berkeley DB, in: Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, 1999, pp. 183-191.
[16] MongoDB. https://www.mongodb.com/.
[17] Cassandra. https://cass{and}ra.apache.org/.
[18] SimpleDB. http://www.cs.bc.edu/sciore/simpledb/.
[19] Sivasubramanian Swaminathan, Amazon dynamoDB: A seamlessly scalable non-relational database service, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, 2012, pp. 729–730.
[20] T. Li, X. Zhou, K. Wang, D. Zhao, I. Sadooghi, Z. Zhang, I. Raicu, A convergence of key–value storage systems from clouds to supercomputers, Concurr. Comput.: Pract. Exp. 1 (28) (2016) 44–69.
[21] M. Seidemann, B. Seeger, ChronicleDB: A high-performance event store, in: Proceedings of the 20th International Conference on Extending Database Technology, 2017, pp. 144–155.
[22] S. Xu, S. Lee, S.W. Jun, M. Liu, J. Hicks, Arvind. Bluecache: a scalable distributed flash-based key–value store, in: Proceedings of PVLDB, 2016, pp. 301–312.
[23] H. Shim, PHash: A memory-efficient, high-performance key–value store for large-scale data-intensive applications, J. Syst. Softw. (123) (2017) 33–44.
[24] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, S. Rumble, The RAMCloud storage system, ACM Trans. Comput. Syst. 33 (3) (2015) 7.
[25] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur-Rahman, N.S. Islam, X. Ouyang, H. Wang, S. Sur, D.K. Panda, Memcached design on high performance RDMA capable interconnects, in: Proceedings of International Conference on Parallel Processing, 2011, pp. 743–752.
[26] MICA. https://github.com/efficient/mica.
[27] X. Wu, L. Zhang, Y. Wang, Y. Ren, M. Hack, S. Jiang, zExpander: a key–value cache with both high performance and fewer misses, in: Proceedings of the Eleventh European Conference on Computer Systems, 2016, pp. 14:1–14:15.
[28] X. Wu, F. Ni, L. Zhang, Y. Wang, Y. Ren, M. Hack, S. Shao, S. Jiang, NVMcached: An NVM-based key-value cache, in: Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems, 2016, pp. 18:1–18:7.
[29] R. Wu, L. Huang, P. Yu, H. Zhou, EDAWS: A distributed framework with efficient data analytics workspace towards discriminative services for critical infrastructures, Future Gener. Comput. Syst. (81) (2018) 78–93.
[30] Apache S4. http://incubator.apache.org/projects/s4.html.
[31] Apache Hadoop, http://hadoop.apache.org/.
[32] Apache Spark, http://spark.apache.org/.
[33] Apache Storm, http://storm.apache.org/.
[34] R. Power, J. Li, Piccolo: building fast, distributed programs with partitioned tables, in: OSDI. 2010, (10) 1-14.
[35] A.J. Jara, D. Genoud, Y. Bocchi, Big data for smart cities with KNIME a real experience in the SmartSantander testbed, Softw.: Pract. Exp. (14) (2010) 293–306.
[36] C. Dehury, P. Sahoo, Design and implementation of a novel service management framework for IoT devices in cloud, J. Syst. Softw. (119) (2016) 149–161.
[37] Y. Wang, K. Li, J. Zhang, K. Li, Energy optimization for data allocation with hybrid SRAM+NVM SPM, IEEE Trans Circuits Syst. I 65 (1) (2018) 307–318.
[38] J. Chen, K. Li, Z. Tang, C. Liu, Y. Wang, K. Li, Data-aware task scheduling on heterogeneous hybrid memory multiprocessor systems, Concurr. Comput.: Pract. Exp. (28) (2016) 17.
[39] V. Leis, A. Kemper, T. Neumann, Scaling HTM-supported database transactions to many cores, IEEE Trans. Knowl. Data Eng. 28 (2) (2016) 297–310.

[40] W. Kang, H. Zhang, P. Ouyang, Y. Zhang, W. Zhao, Programmable stateful in-memory computing paradigm via a single resistive device, in: Proceedings of Computer Design (ICCD) IEEE International Conference on, 2017, pp. 613–616.
[41] Checksum. https://en.wikipedia.org/wiki/checksum.
[42] Y. Wang, K. Li, K. Li, Partition scheduling on heterogeneous multicore processors for multi-dimensional loops applications, Int. J. Parallel Programm. 45 (4) (2017) 827–852.
[43] Apache Spark. http://spark.apache.org/.
[44] S. Yuan, J. Wang, X. Zhao, Real-time bidding for online advertising: measurement and analysis. CoRR, 2013.

Renke Wu is currently working toward the PhD degree in the department of computer science and engineering at the Shanghai Jiao Tong University (SJTU). His research interests lie in the area of parallel computing, software engineering, distributed systems, system of systems, high performance computing and big data analysis.

Linpeng Huang received his MS and PhD degrees in computer science from Shanghai Jiao Tong University in 1989 and 1992, respectively. He is a professor of computer science in the department of computer science and engineering, Shanghai Jiao Tong University. His research interests lie in the area of distributed systems, formal verification techniques, architecture-driven software development, system of systems, parallel computing, big data analysis and in-memory computing.

Haojie Zhou received his MS degree in computer science from Chinese Academy of Sciences. He works in the State Key Laboratory of Mathematic Engineering and Advance Computing, Jiangnan Institute of Computing Technology. His research interests lie in the area of distributed systems, architecture-driven software development and parallel computing.