High Performance Design for Redis with Fast Event-Driven RDMA RPCs

Xuecheng Qi, Huiqi Hu(✉), Xing Wei, Chengcheng Huang, Xuan Zhou, and Aoying Zhou

School of Data Science and Engineering, East China Normal University, Shanghai, China
{xcqi,simba_wei,cchuang}@stu.ecnu.edu.cn,
{hqhu,xzhou,ayzhou}@dase.ecnu.edu.cn

Abstract. Redis is a popular key-value store built upon socket interface that remains heavy memory copy overhead within the kernel and considerable CPU overhead to maintain socket connections. The adoption of Remote Direct Memory Access (RDMA) that incorporates outstanding features such as low-latency, high-throughput, and CPU-bypass make it practical to solve the issues. However, RDMA is not readily suitable for integrating into existing key-value stores. It has a low-level programming abstraction and the original design of existing systems is a hurdle to exploit RDMA's performance benefits. RPCs can provide simple abstract programming interfaces that make it easy to be integrated into existing systems. This paper proposes a fast event-driven RDMA RPC framework named FeRR to promote the performance of Redis. First, we describe our design of FeRR that is based on one-sided RDMA verbs. Second, we propose an efficient request notification mechanism using event-driven model that can decrease the CPU consumption of polling requests. Finally, we introduce a parallel task engine to eschew the bottleneck of the single-threaded execution framework in Redis. Comprehensive experiment shows that our design achieves orders-of-magnitude better throughput than Redis - up to 2.78 million operations per second and ultra-low latency - down to 10 µs per operator on a single machine.

Keywords: RDMA · FeRR · Redis

1 Introduction

Existing key-value stores like Redis [1] use the conventional socket I/O interface, which is easy to develop and compatible with commodity NIC. Because of large CPU copy overhead that network data package should be copied between user space to kernel space, about 70% time is spent on receiving queries and sending responses over TCP [2]. Furthermore, low-speed NIC confines the performance of key-value stores. Remote Direct Memory Access (RDMA) can provide high-throughput and low-latency network communication. There has been increasing interest in recent years to build high performance key-value stores using RDMA-capable network [3–8].

In the above works, many design considerations in RDMA suit are confined. Pilaf [4] and FaRM [5] use one-sided RDMA read to traverse remote data structures but suffer from multiple round trips across the network. As we demonstrate later, RDMA read cannot saturate the peak performance of the RDMA hardware. Herd [7] transmits clients' requests to server memory using RDMA write and polling server memory for incoming requests using its CPU. However, this request notification approach is inefficient and bings considerable CPU overhead.

Although RDMA-capable network is a treasure, integrating it into existing commercial systems is disruptive. These prototype systems type the code to a specific interconnect API to support RDMA communication. But for existing systems, this can lead to significant code changes. In terms of simplicity, RPCs reduce the software complexity required to design key-value stores compared to one-sided RDMA-based systems.

To solve these issues, we propose FeRR, a fast event-driven RDMA RPC framework that delivers low latency and high throughput. FeRR has fully considered network primitives of RDMA hardware and possesses an efficient event-driven request notification mechanism. FeRR provides simple abstract programming interfaces that make it easy to be integrated into existing systems. Furthermore, we have designed and implemented a new novel branch of Redis over FeRR, FeRR-driven Redis.

Since FeRR can improve the performance of Redis, several new issues have emerged. Redis Serialization Protocol (RESP) is not suitable for RDMA verbs but is necessary for multiple data type support. Recent works [4,7] need no serialization protocol, they set only one data type of $<key, value>$ as $<string, string>$. As a remedy, we take a less disruptive approach to optimize the existing RESP to relieve the memory copy overhead, meanwhile maintain the support of multiple data types.

When network is not the bottleneck, the single thread framework of Redis is the new issue. To address this problem, we design a parallel task engine for FeRR-driven Redis. The major part of this engine is a cuckoo hashing [9] table with optimistic locking. Cuckoo hashing is friendly for read-intensive workloads while many workloads of key-value stores are predominately reads, with few writes [10]. Taking constant time in the worst case, cuckoo hashing has great lookup efficiency. Meanwhile, optimistic locking outperforms pessimistic locking for read-intensive workloads.

Overall, this paper makes three key contributions:

(1) We discuss our design considerations based on a sufficient analysis of the performance of RDMA and disadvantages of previous works.
(2) We propose a fast event-driven RDMA framework name FeRR that delivers extremely high throughput and low latency.
(3) We design and implement a high-performance version of Redis named FeRR-driven Redis, including an optimized serialization protocol for low-latency transmission, and a parallel task engine for high-throughput execution.

We implemented these designs on Redis v3.0, branching a new branch of Redis. We conducted experiments on ConnectX-3 RNIC to evaluate the performance of our design. Experiments show that FeRR-driven Redis achieves throughput up to 2.78 million operations per second and latency down to 10 µs per operation, which are over 37.6× higher throughput and only 15% latency of Redis on IPoIB.

2 Preliminary

2.1 RDMA

RDMA enables zero-transfer, low round-trip latency, and low CPU overhead. RDMA capable networks can provide 56 Gbps of bandwidth and 2 µs round-trip latency with Mellanox ConnectX-3 RNIC. In this section, we provide an overview of RDMA features: verb types, transport modes that are used in the rest of the paper.

Message Verbs: SEND and RECV, provide user-level two-sided message passing that involves remote machine's CPU. Before a SEND operation, a pre-posted RECV work request should be specified by the remote machine where the request is written to. Then, responder could get the request by polling the completion queue. Compared to RDMA verbs, message verbs have higher latency and lower throughput.

RDMA Verbs: WRITE and READ are one-sided operations, namely, allow full responder's CPU bypass that client can write or read directly the memory of responder without its CPU involved. The new type of message passing technique can relieve the overhead of responder's CPU since responder is unaware of client's operations. Furthermore, one-sided operation has the lowest latency and highest throughput.

Transports Mode: RDMA transports are divided into reliable or unreliable, and connected or unconnected. Reliable transports guarantee that messages are delivered in order and without corruption, while unreliable has no such guarantee. InfiniBand uses a credit-based flow control in the link layer to prevent loss of data, and CRC to ensure data integrity [11]. Thus, the packet losses of unreliable transports are extremely rare. The difference between connected and unconnected transports is the number of connected queue pairs (QP). Connected transports need that one QP sends/receives with exactly one QP, and unconnected transports allow one QP sends/receives with any QP. RDMA verbs support two types of connected transports: Reliable Connection (RC) and Unreliable Connection (UC). Unreliable Datagram (UD) supports only SEND operations.

2.2 Redis

The existing open-source Redis implementation is designed using traditional Unix socket interface. While having an only single thread to handle socket connections, Redis has built an I/O multiplexing event model, such as epoll/select,

Table 1. Redis serialization protocol

Data type	Label byte	Example
Simple String	+	“$+ok\backslash r\backslash n$”
Errors	−	“$-Error\ message\backslash r\backslash n$”
Integers	:	“$: 1\backslash r\backslash n$”
Bulk String	\$	“$\$6\backslash r\backslash nfoobar\backslash r\backslash n$”
Arrays	*	“$*3\backslash r\backslash n : 1\backslash r\backslash n : 2\backslash r\backslash n : 3\backslash r\backslash n$”
Example:	$*3\backslash r\backslash n\$3\backslash r\backslash nSET\backslash r\backslash n\$3\backslash r\backslash nkey\backslash r\backslash n\$5\backslash r\backslash nHello\backslash r\backslash n$	

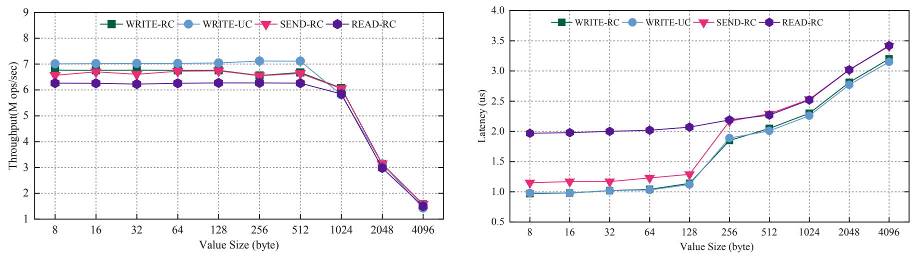
that has the ability to handle hundreds of connections and achieve good performance. In Redis, I/O multiplexing event model considers operations as file events(except time event). A connection, a command or a reply is constructed as a file event and inserted into an event set which is monitored by epoll() in Linux. Then Linux kernel epolls a fired file event for a corresponding function to process when its socket is ready to read or write.

Different from Memcached [12], multiple data types are supported in data transport of Redis, such as string, integer, array. As Table 1 shows, RESP attaches one byte to different types of data to distinguish them. Taking a set command **“SET key Hello”** as an example, it should be encoded in client-side as “Example” in Table 1 that means a size of 3 arrays including 3 bulk strings with respective length 3, 3, 5. And server decodes the encoded buffer to a set command to process. RESP remains several times of memory copy for encoding and decoding.

2.3 Related Work

RDMA-Optimized Communication: The HPC community engages in taking advantage of RDMA with Infiniband to improve the MPI communication performance, such as MVAPICH2 [13] OpenMPI [14]. They provide RDMA-based user-level libraries that support the MPI interface. These works [3,15,16] utilize RDMA to improve the throughput and reduce the CPU overhead of communication of the existing systems Hadoop, HBase, Memcached. However, Most of them only use SEND/RECV verbs as a fast alternative to socket-based communication despite leveraging one-sided RDMA primitives. Memcached-RDMA [3] fixes message verbs and RDMA verbs to build a communication library. For put operations, the client sends the server a local memory location using SEND verb and the server reads the key-value pair via RDMA read. Get operations involve SEND verb and RDMA write. The server writes the data using RDMA write into the allocated memory address pre-sent by the client.

User-Level Communication: Other than the usage of RDMA capable network, MICA [17] utilizes the Intel DPDK library [18] to build the key-value store on classical Ethernet. The Intel DPDK library supports zero-copy technology that eliminates CPU interrupt and memory copy overhead. Specifically,



(a) Throughput of 56Gbps infiniband

(b) Latency of 56Gbps infiniband

Fig. 1. Throughput and latency under different verbs type and transport modes

MICA combines exclusive partitions with CPU cores to avoid synchronization but introduces core imbalance for skewed workloads. KV-Direct [8] introduces a new key-value store built on a programmable NIC. It offloads key-value processing on CPU to the FPGA-based programmable NIC to extend RDMA primitives from memory operations to key-value operations and enable remote direct access to the host memory. As a result, systems on user-level communication can achieve matched throughput to RDMA based solutions, nonetheless with higher latency.

RDMA-Based Key-Value Store: RAMCloud [19] takes advantage of message verbs of RDMA to build an in-memory, persistent key-value store. Pilaf [4] lets clients directly execute get operations at client side using multiple RDMA read operations and introduces a self-verification data structure to detect read-write races between client-side get operations and server-side put operations. Meanwhile, HydraDB [6] uses RDMA read to operate get as well and implement the client cache to promote system performance. HERD [7] points out that RDMA write has better throughput and lower latency than RDMA read. For this, HERD building a high-performance key-value store using one-sided RDMA write for sending requests and use SEND over UC for replying responses because datagram transport can scale better for applications with a one-to-many topology. FaRM [5] exploits RDMA verbs to implement global shared address among distributed nodes.

3 Design Consideration

3.1 Network Primitives Choice

Since we have introduced FeRR, the first challenge is that how to choose networking primitives of RDMA to saturate the peak performance of RNIC. We conduct an experiment to measure the throughput and latency of ConnectX-3 56 Gbps InfiniBand under different verbs and transport modes. As Fig. 1 shows, RDMA write over UC have the best performance in both throughput and latency. As to verbs, message verbs require involvement of the CPU at both the sender and receiver, but RDMA write bypass the remote CPU to operate directly on remote memory. Meanwhile, RDMA write outperforms RDMA read as well, because the responder does not need to send packets back, RNIC performs less processing. Performance of RDMA write over UC and RC are nearly identical, using UC is still beneficial that it requires less processing at RNIC to save capacity. Therefore, we select RDMA write over UC to as network primitives of FeRR.

3.2 Request Notification

Another issue is how to build an efficient request notification mechanism. As well-known, RDMA verbs allow CPU bypass that can greatly relieve the overhead of the server's CPU since the server is unaware of the client's operations. Therefore, the server needs to scan the message pool continuously to get new requests, which leads to high CPU overhead. Also, with increasing number of clients, latency is increased much. Furthermore, Redis server's CPU is easier to become a bottleneck due to complex code path. Event-driven way is very attractive that it relieves CPU consumption and is effective for request notification. Therefore, we design an event-driven request notification mechanism based on RDMA-write-with-imm. It is the same as RDMA write, but a 32-bit immediate data will be sent to remote peer's receive (CQ) to notify the remote CPU of the completion of write. Meanwhile, this RDMA primitive retains the property of high throughput and low latency of RDMA write.

4 FeRR

In this section, we explain the design of FeRR, and how we reap the benefits of RDMA. We first motivate a well-tuned RPC architecture, which incorporates RDMA-write-with-imm as network primitive. Then we describe an event-driven mechanism that can efficiently poll requests while saving CPU capacity.

4.1 Architecture

In the architecture of FeRR, each client connects the server with two QPs where one takes charge of the outbound verbs and the other for the inbound verbs. As Fig. 2 shows, a client maintains two buffers that one for sending requests and the other for receiving responses. Server has a message pool for temporary storage when exchanging messages. After establishing the connection with server, a client is given a clientID and allocated a pair of correspondingly buffers in the pool for communication. By doing so, the server can easily manage the allocation and displacement of buffers in the message pool.

FeRR is fundamentally built on RDMA-write-with-imm that can notify remote machine immediately. RDMA-write-with-imm consumes a receive request form the QP of remote side and the request is processing immediately after being polled from the CQ. Besides, it is able to carry a 32-bit immediate data in the message. We attach the client's identifier in the immediate data filed including a clientID and an offset of the client's receive buffer. The server can directly locate

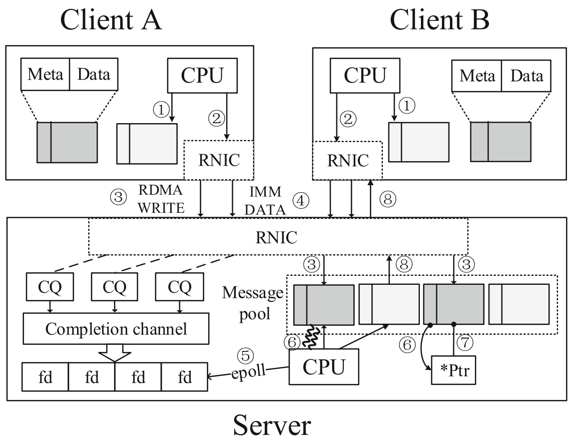


Fig. 2. Architecture of FeRR

the request in the message pool with the identifier instead of scanning the whole pool.

Figure 2 illustrates the process of a FeRR call. Before the call, the server and clients exchange their pre-allocated memory space addresses and keys via socket connection. A client creates and serializes a new request into the request buffer ①, then calls the write API of FeRR ②. FeRR uses RDMA-write-with-imm to write the memory-object of request into the pre-exchanged memory space in the message pool of server ③. FeRR uses the IMM value to include the clientID and the offset that indicates which buffer in the message pool ④. In the server, a polling thread epolls entries from CQs via completion channel ⑤. The server read CQE to get offset and clientID to know the request is in which buffer and belongs to which client. The user thread decodes the request and constructs the corresponding pointer to reference the memory object ⑥. FeRR returns the pointer to a worker thread to handle the request ⑦. Finally, response is written back to the client ⑧.

4.2 Event-Driven Request Notification

In FeRR, we design an event-driven mechanism for efficient request notification. For the conventional way, the server needs polling the CQ ceaselessly to learn whether there is a work completion. Reading the work completions using events decreases the CPU consumption because it eliminates the need to perform constant polling the CQ. A completion channel can monitor several CQs and notify the thread once a work completion yields. FeRR registers fixed number (can be configured) of CQs to one completion channel and epolls the coming completion event. After the request notification, the worker thread polls the work completion from the CQ. Finally, the request is got and processed by the thread. This is the fundamental procedure of event-driven request notification in FeRR.

Algorithm 1: Event-driven request notification

```
1  Function Binding():
2      $channel \leftarrow$ initialize a Completion Channel;
3      for $i = 0$ to $CQ\_NUM\_PER\_CHANNEL$ do
4          $cq[i] \leftarrow$ initialize the $i_{th}$ CQs;
5          $cq[i].channel \leftarrow channel$ ;
6          for $j = 0$ to $QP\_NUM\_PER\_CQ$ do
7              $qp[j] \leftarrow$ initialize the $j_{th}$ QPs;
8              $qp[j].cq \leftarrow cq[i]$ ;
9  Function f_epoll():
10     while true do
11         $ret \leftarrow epoll(channel\_fd)$;
12         if $ret = true$ then
13             $cq \leftarrow ibv\_get\_cq\_event(channel)$;
14             while cq has any Work Completion do
15                 $wc \leftarrow ibv\_poll\_cq(cq)$;
16                 notify corresponding worker thread to process the request;
```

The native approach has a major bottleneck that is inefficient. Upon polling a work completion, the worker thread needs to switch for polling to process the request. As one CQ is always shared by multiple QPs, there are probably more than one work completions in the CQ. To solve the bottleneck, we exploit specific polling thread to take charge of request notification. As Algorithm 1 shows, FeRR first initializes a completion channel "channel" which is used for monitoring CQs (line 2). Then, FeRR initializes CQs and registers a fixed number of them to the "channel" (line 3–5). Next, QPs are created and a fixed number of them are bound to one CQ (line 6–8). In *f_epoll*() function, FeRR calls *epoll*() to wait for completion event and gets the CQ context (line 10–13). After that, Polling threads call *ibv_poll_cq*() to poll all of the completions from the CQ and notify corresponding worker threads to process the requests (14–16).

4.3 APIs of FeRR

FeRR supports three kinds of interface: RPC constructor(f_init())), memory management(f_reg(), f_free()) and messaging interface(f_send(), f_recv()) for user-level applications. We select these semantics because they are simple to RDMA-driven application programmers. Table 2 lists major APIs of FeRR.

5 FeRR-Dirven Redis

In this section, we design and implement FeRR into Redis that proposes a new branch of Redis, FeRR-Redis. Then we propose a subtle serialization protocol

Table 2. Major APIs of FeRR

Interface	Description
status → f_init()	Initialize resources for FeRR
addrID → f_reg()	Register memory space into RNIC
status → f_free()	Free the memory space registered
status → f_send()	FeRR sends data to remote machine
status → f_recv()	Receives next FeRR send
status → f_epoll()	Epoll WQEs from the completion channel

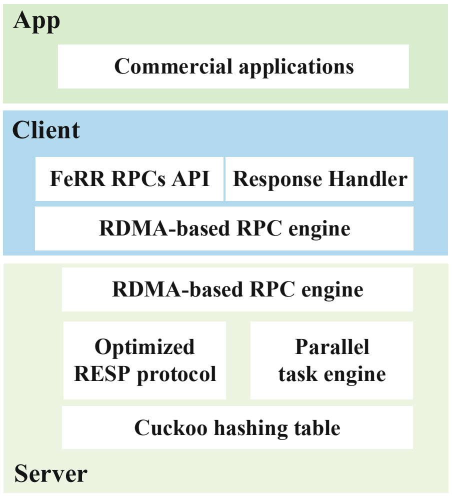


Fig. 3. Overview framework of FeRR-Redis

to replace the origin RESP, which not only reserves the supporting of multiple value types but also solves the memory copy issue. Finally, we point out that single-threaded task engine is a new bottleneck instead of socket-based network and propose a parallel task engine for Redis.

5.1 Overview

Figure 3 illustrates the overview framework of FeRR-Redis that contains two parts: (1) Client-side layer: Client side provides APIs of FeRR to the application layer and is in charge of handling responses. (2) Server-side layer: The main part of this layer is a parallel task engine. To embrace Redis with multi-threaded execution, we implement a concurrent cuckoo hashing table with optimistic locking. Besides, we propose an optimized serialization protocol of Redis to substitute the original one.

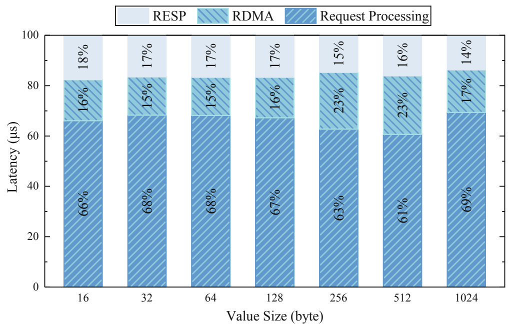


Fig. 4. Respective proportions of latency

5.2 Optimized Serialization Protocol

RDMA write can directly write the payload of request to responder's memory region, neither needs an extra serialization protocol. Meanwhile, serialization protocol leads to considerable memory copy overhead. To measure the overhead of RESP, we conduct an experiment to evaluate its latency on different value sizes from 16 bytes to 1024 bytes. We simulate set commands and use RESP to serialize and deserialize it for 1000000 times, and calculate the average latency. As Fig. 4 shows, the average latency of RESP increases by value size and occupies about 17% of the total latency of a request-reply. The experiment shows that the overhead of RESP is considerable and could significantly affect the performance of FeRR-Redis, while it is crucial and worthy to optimize RESP.

Supposed that the memory address and type of an object are known, it's easy to new a corresponding pointer to directly reference the object that can be manipulated by this pointer. Likewise, serialization protocol is a facile thing if we can use a request pointer to reference the request object. We consider that RDMA supports memory-object access that an object in client can directly be copied to the memory of the server, like a memory copy operation. The remaining issue is how to exactly know the struct type of this request to construct the pointer.

As Fig. 2 shows, we divide the request buffer into two parts: metadata part and data part, the former is used to store encoding bytes to distinguish the struct type of the command, the latter is used to store the command. Taking a request as an example, the data part stores the command *SET key Hello*, while the encoding bytes are stored in the metadata part. The entire request should be "$*3\$3\$3\$5SETkeyHello$". Then the client writes the request buffer into server memory region. Upon getting the request, the server decodes the metadata part of the request to get the encoding bytes of the struct. It is easy to realize a 3-element string struct and constructs a corresponding struct pointer to reference the command. Then the server processes the command by manipulating this pointer. For multiple data types, encoding bytes distinguish them using different byte in the metadata part. The optimized serialization protocol is a simple and effective approach that relieves the default memory copy overhead.

5.3 Parallel Task Engine

In order to support multi-threaded execution, we replace the chained hash table used in Redis that is not friendly to concurrent access. A chained hash table uses a linked list to deal with hash conflict where the new hash table entry is linked to the last entry of the conflicting bucket. Chaining is efficient for inserting or deleting single keys. However, a lookup may require scanning the entire chain that is time-consuming.

Cuckoo hashing table is an attractive choice where each lookup requires only k memory reads on average with k hash functions for get operations. In many application scenarios, 95% of the operations are get operations [10]. It makes cuckoo hashing more attractive that is designed for read-intensive workloads of Redis. Second, we should cope with concurrency control challenge. There are serveral ways like mutexes, optimistic locking [20], or lock-free data structures to ensure consistency among multiple threads. For read-intensive workloads, optimistic locking outperforms pessimistic locking because of no overhead of context switch. Furthermore, lock-free data structures are too hard to code correctly and integrate into Redis, so we minimize the synchronization cost with efficient optimistic locking scheme.

Concurrent Hash Index: We use the 3-way cuckoo hashing table to replace the original chaining hash table. The 3-way cuckoo hashing means the hash schema has 3 orthogonal hash functions. When a new key comes, it chooses one of the 3 locations that are empty to insert. If all possible locations for the new key to insert are full, resident key-value pair is kicked to an alternative location to make space for the new key. There may happen cascaded kicking until each key-value pair is in the proper location. The hash table is resizing when the number of kicks exceeds the limit or when a cycle is detected.

This data structure is space-efficient that pointers are eliminated from each key-value object of the chained hashing table. Cuckoo hashing increases space efficiency by around 40% over the default [20]. Meanwhile, cuckoo hashing is friendly to lookup operations that needs few hash entry traversals per read. At a fill ratio of 95%, the average probes in 3-way Cuckoo hashing are 1.5. In the worst case, the max lookup time of 3-way cuckoo hashing is 3 while chained hashing needs to search the whole hash chain that takes much more time.

Supporting Concurrent Access: To solve the consistency issue, we adopt the optimistic locking schema that performs reads optimistically without satisfying get operations when generating put. We add a 32-bit version field at the start of each hash table entry. The initial version number is zero, and the maximum is 2ˆ32-1. When nearly reaching the maximum version number, it is about to be made zero by a successful put operation.

For a get operation, it first read the key version: if this version is odd, there must be a concurrent put operator on the same key, and it should wait and retry; otherwise, the version is even, it reads the key-value pair immediately. Upon completion of entry fetched from the hash table, it reads the version again and checks if the current version number is equal to the initial version number. If the check fails, the get operation retries.

We use a compare-swap (CAS) operation instruction to allows multiple put operations to access the same entry. Before a put operation displaces the original key-value, it first reads the relevant version and waits until the initial version number is even. Then the writer increases the relevant version by one using a CAS operation. If succeeds, the odd version number indicates other operations to wait for an on-going update for the entry. Upon completion, it increases the version number by one again. As a result, the key version increases 2 and keeps even after each displacement.

6 Evaluation

In this section, we analyze the overall performances of our high-performance design of FeRR-Redis, then the benefits from each mechanism design.

6.1 Experimental Setup

Hardware and Configuration: Our experiments run on a cluster of five machines, one for server and the other four for clients. Each machine is equipped with an Intel(R) Xeon(R) Silver 4110 CPU @ 2.10 GHz processor. 192 GB memory is installed on each node. All nodes are connected through InfiniBand FDR using Mellanox ConnectX-3 56 Gb/sec. On each machine, we run Centos7.5.

Comparison Target: We compare FeRR-Redis against Redis on IPoIB, FeRR-Redis with single-threaded execution. We run a FeRR-Redis server on one physical machine, while clients are distributed among four remaining machines. All of the Redis implementations disable logging function. We intend to find out the importance of RDMA kernel bypass and parallel task engine of FeRR-Redis. Furthermore, the performance gap between them demonstrates this idea.

YCSB Workloads: In addition to get workload and put workload of redis-benchmark, we use two types of YCSB workloads: read-intensive (90% GET,10% PUT) and write-intensive (50% GET, 50% PUT). Our workloads are uniform workloads, the keys are chosen uniformly at random. This workload is generated off-line using YCSB [21].

6.2 Microbenchmark

We integrate FeRR into redis-benchmark tool in Redis project to supprt RDMA-capable network. We run it to measure the throughput and latency of different workloads.

Latency: Figure 5 shows the latency of get and put operations of FeRR-Redis and Redis (IPoIB) with 16 clients. With few clients, server can process operations

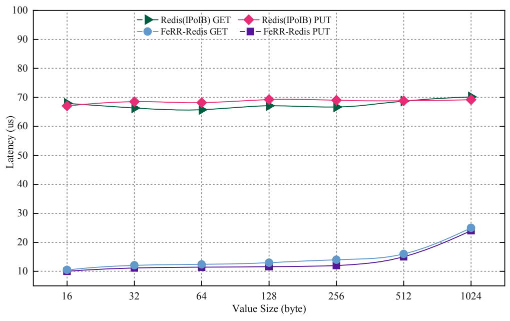


Fig. 5. Latency of FeRR-Redis compared to Redis (IPoIB)

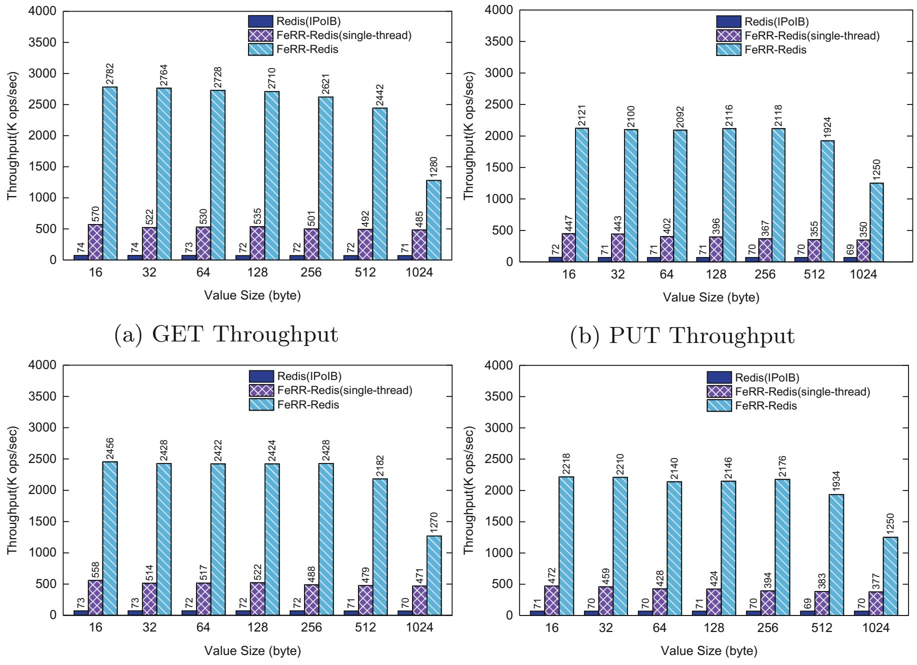
as fast as possible. For small key-values, FeRR-Redis achieves 10.0 µs latency for get operations and 10.4 µs latency for operations. For large key-values (1024bytes values), the latencies of get and put are 24.2 µs and 25.1 µs.

With value size increasing, the average latencies of Redis on IBoIP have a little change. This is because of much time spent to transmit network packets over socket interface and generate serialization and deserialization. No matter for small or large values, the latencies of get and put are more than 60 µs. Generally, the average latencies of FeRR-Redis beat that of Redis on IPoIB by 3×–7×.

Throughput: Figure 6 shows the throughput of FeRR-Redis, single-threaded FeRR-Redis and Redis on IPoIB under different read/write ratio workloads. The throughput of FeRR-Redis is achieved with 16 concurrent clients, compared to the single-threaded FeRR-Redis and Redis on IPoIB running 16 concurrent clients as well. For small key-values (16bytes values), FeRR-Redis can achieve 2.78 million operations per second, compared to 570 Kops/sec for the single-threaded FeRR-Redis and 74 Kops/sec for Redis on IPoIB. For get operations, FeRR-Redis can achieve throughput improvement 4.7× that of FeRR-Redis with single-threaded execution and 37.6× that of Redis (IPoIB), while 4.7× that of FeRR-Redis with single-threaded execution and 29.5× that of Redis (IPoIB) for put operations.

For larger key-values, FeRR-Redis keeps great performance. The throughputs of get and put operations slightly go down while their usage of Infiniband card's performance increases. However, FeRR-Redis cannot saturate the Infiniband card's performance because Redis has an extremely long code path and much exception handling that incurs much CPU overhead. Specifically, for 1024-byte value size, FeRR-Redis achieves 1.28 million get per second and 1.25 million put per second.

Because IBoIP remains CPU copy and kernel involved, Redis on IPoIB is bottlenecked by the poor performance of IPoIB and its performance is the same when running on 10 Gbps Ethernet. When the network is not the bottleneck, the single CPU is the new bottleneck. The single-threaded FeRR-Redis's throughput is restricted and unable to saturate the Infiniband card's performance. As a



(a) GET Throughput

(b) PUT Throughput

(c) Throughput (90% GET, 10% PUT)

(d) Throughput (50% GET, 50% PUT)

Fig. 6. Throughput achieved for FeRR-Redis, FeRR-Redis (single-thread), and Redis (IPoIB)

conclusion, we precisely found out the main bottlenecks, and have crafted high-performance design with corresponding approaches to boost the performance of Redis.

As YCSB workloads, FeRR-Redis is outstanding for the read-intensive workload (95% GET, 10% PUT) that achieves peak throughput of 2.45 million operations per second, which is slightly lower than that of get workload. For write-intensive workload (50% GET, 50% PUT), FeRR-Redis achieves almost the same throughput as that of put workload, because read- write contention incurs much overhead. As a result, the data structure of multi-threaded framework is suitable for read-intensive workload.

Optimized Latency: In our previous analysis, RESP consumes several μs to execute, which is almost 15% of the whole latency. Our optimization aims to eliminate memory copy during serialization and deserialization process. Figure 7 shows that FeRR-Redis get latency with optimized serialization protocol decrease by 2–3 µs compared to the default RESP while a native round-trip of RDMA write for 16 byte is only 2 µ. The result demonstrates our optimization for RESP is effective.

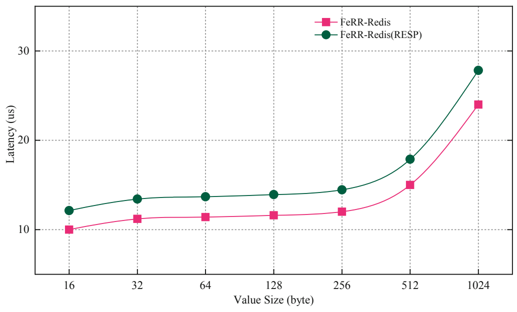


Fig. 7. FeRR-Redis latency with RESP and optimized serialization protocol

7 Conclusion

In this paper, we propose a high-performance design of Redis leveraging RDMA, FeRR-Redis. We solve three key issues of integrating RDMA into Redis: First, we design a fast event-driven RPC framework that is closely coupled with RDMA primitives with high-level programming interface. Second, we optimize RESP to relieve memory copy overhead and support multiple data types via transmission. Finally, we exploit an efficient parallel task engine that embraces Redis with multi-core processing. Evaluations show that FeRR-Redis effectively explores hardware benefits, and achieves orders-of-magnitude better throughput and ultra-low latency than Redis.

Acknowledgement. This work was supported by National Key R&D Program of China (2018YFB1003303), National Science Foundation of China under grant number 61772202, Youth Program of National Science Foundation of China under grant number 61702189, and Youth Science and Technology-Yang Fan Program of Shanghai under Grant Number 17YF1427800.

References

1. Redis homepage. https://redis.io/
2. Metreveli, Z., Zeldovich, N., Kaashoek, M.F.: Cphash: a cache-partitioned hash table, vol. 47. ACM (2012)
3. Jose, J., et al.: Memcached design on high performance RDMA capable interconnects. In: 2011 International Conference on Parallel Processing, pp. 743–752. IEEE (2011)
4. Mitchell, C., Geng, Y., Li, J.: Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In: USENIX Annual Technical Conference, pp. 103–114 (2013)
5. Dragojević, A., Narayanan, D., Castro, M., Hodson, O.: FaRM: fast remote memory. In: 11th USENIX Symposium on Networked Systems Design and Implementation, pp. 401–414 (2014)

6. Wang, Y., et al.: HydraDB: a resilient RDMA-driven key-value middleware for in-memory cluster computing. In: SC 2015: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11. IEEE (2015)
7. Kalia, A., Kaminsky, M., Andersen, D.G.: Using RDMA efficiently for key-value services. In: ACM SIGCOMM Computer Communication Review, vol. 44, pp. 295–306. ACM (2014)
8. Li, B., et al.: KV-direct: high-performance in-memory key-value store with programmable NIC. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 137–152. ACM (2017)
9. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004)
10. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: Workload analysis of a large-scale key-value store. In: ACM SIGMETRICS Performance Evaluation Review, vol. 40, pp. 53–64. ACM (2012)
11. Connect-IB: Architecture for scalable high performance computing whitepaper. https://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf/
12. Memcached homepage. https://memcached.org/
13. Liu, J., et al.: Design and implementation of MPICH2 over InfiniBand with RDMA support. In: 18th International Parallel and Distributed Processing Symposium, Proceedings, p. 16. IEEE (2004)
14. Shipman, G.M., Woodall, T.S., Graham, R.L., Maccabe, A.B., Bridges, P.G.: Infiniband scalability in open MPI. In: Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, pp. 10-pp. IEEE (2006)
15. Lu, X., et al.: High-performance design of Hadoop RPC with RDMA over InfiniBand. In: 2013 42nd International Conference on Parallel Processing, pp. 641–650. IEEE (2013)
16. Huang, J., et al.: High-performance design of HBase with RDMA over InfiniBand. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, pp. 774–785. IEEE (2012)
17. Lim, H., Han, D., Andersen, D.G., Kaminsky, M.: MICA: a holistic approach to fast in-memory key-value storage. In: 11th USENIX Symposium on Networked Systems Design and Implementation, pp. 429–444 (2014)
18. Intel DPDK homepage. https://dpdk.org/
19. Ongaro, D., Rumble, S.M., Stutsman, R., Ousterhout, J., Rosenblum, M.: Fast crash recovery in RAMCloud. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 29–41. ACM (2011)
20. Fan, B., Andersen, D.G., Kaminsky, M.: MemC3: compact and concurrent memcache with dumber caching and smarter hashing. Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation, pp. 371–384 (2013)
21. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154. ACM (2010)