RF-RPC: Remote Fetching RPC Paradigm for RDMA-Enabled Network

Yongwei Wu, *Senior Member, IEEE*, Teng Ma, Maomeng Su, Mingxing Zhang, Kang Chen, and Zhenyu Guo

Abstract—Remote Direct Memory Access (RDMA) devices are widely deployed in modern data centers. However, existing RDMA usages lead to a dilemma between performance and redesign cost. *Server-reply* mode directly replaces socket-based send/receive primitives with corresponding RDMA counterparts. It can not fully harness the power of RDMA devices. *Server-bypass* is distinct mode provided by RDMA for reaching the hardware limits. This mode can totally bypass the server by using one-sided RDMA operations but at the cost of redesigning software.

This paper provides a different approach, called *RF-RPC* (Remote Fetching RPC Paradigm). Different from *server-reply*, *RF-RPC* makes client fetch the results from server using one-sided RDMA instead of waiting the results pushed by server. Different from *server-bypass*, *RF-RPC* demands server to process client's requests for supporting programming paradigm like *RPC*. Thus, *RF-RPC* can achieve high performance without abandoning traditional programming models. An *RF-RPC* supported in-memory key-value store shows that the performance can be improved by 1.6× comparing to *server-reply* paradigm and 4 × comparing to *server-bypass* paradigm.

Index Terms—RDMA, RPC, key-value database, distributed systems.

1 Introduction

RDMA offers features like low-latency, high-bandwidth, and server-bypassing. *RDMA* has been widely deployed in modern data centers [2], [3], [4], [5], [6], [7], [8], [9], [10]. Infiniband, RoCE/RoCEv2 (RDMA over Converged Ethernet), and iWARP (Internet Wide-area RDMA Protocol) are different network protocols which implement RDMA technique. Infiniband is the most commonly used Infiniband-based RDMA protocol which needs special Infiniband NIC and switches supporting. RoCE [11]/RoCEv2 [12] and iWRAP [13] base on Ethernet and can run over RDMA-support Ethernet NIC, but also lost some critical pieces of the Infiniband and can only be treated as an alternative to InfiniBand. Compared with iWARP, RoCE/RoCEv2 is the only industry-standard Ethernet-based RDMA solution with a multi-vendor ecosystem delivering [13].

A common usage of RDMA, denoted as *server-reply*, is to replace original TCP/IP socket send/receive primitives with the corresponding RDMA counterparts. Through this way, the same RPC interfaces can be implemented [4], [7], [8], [14], [15], [16], [17]. It can boost the performance without much efforts of re-programming, e.g., RDMA-Memcached [7] has applied this approach and boosted the performance Memcached [18] by 4× comparing with using TCP/IP.

However, *server-reply* does not really unleash all the power of RDMA. The iconic feature of RDMA is **one-sided** operations that can totally bypass CPU and OS on remote machines. This gives clients the ability of accessing server memory directly. We call this as *server-bypass* [2]. Previous works such as Pilaf [2] and FaRM [3] all have embraced *server-bypass*. They show that that *server-bypass* can be faster than *server-reply* by as much as 100% [2], [3], [5], [19].

Despite the performance gain, *server-bypass* needs developers to redesign the software such as using RDMA friendly data structures and algorithms because the remote CPU is bypassed and not processes the requests. For instance, Pilaf [2] (a key-value store) uses CRC64 for data race detection among GET from clients (using *server-bypass*) and PUT on server (using *server-reply*), while it also designs a specific hash-table to reduce the number of RDMA operations for completing GET requests. This becomes a dilemma between redesign cost and performance. More importantly, these special data structures are usually application-specific. For example, a data structure designed for serving GET/PUT operations on a key-value store cannot be used for other applications, such as those with simple statistic operations [5].

To solve this dilemma, we propose a new RDMA-based RPC paradigm called Remote Fetching RPC Paradigm (*RF-RPC*). In *RF-RPC*, the server processes the requests sent from clients, so that its CPU usage is similar to that with traditional RPC interfaces. As a result, applications that use traditional RPC can remain largely unchanged. Meanwhile, *RF-RPC* achieves higher performance than both *server-reply* and *server-bypass*, which comes from two design choices. The first is releasing the server CPU from processing network operations, similar to *server-bypass*. The second is avoiding bypass access amplification that limits the usage of *server-bypass* supporting traditional RPC based applications.

The performance boosting of releasing server CPU from processing network operations comes from the differences

- *An earlier version of this work [1] appeared in Eurosys 2017.*
- *Yongwei Wu, Teng Ma, Kang Chen, Mingxing Zhang are with the Department of Computer Science and Technology, Graduate School at Shenzhen, Tsinghua University, Beijing 100084, China. Email: {wuyw@, mt16@mails., chenkang@}tsinghua.edu.cn; Mingxing Zhang is also now with Sangfor Inc., Maomeng Su is now with Huawei Inc., Zhenyu Guo is now with Ant Financial Inc.*
- *Corresponding Author: Yongwei Wu (wuyw@tsinghua.edu.cn) and Kang Chen (chenkang@tsinghua.edu.cn)*

between in-bound and out-bound RDMA operations. **Serving** a one-sided RDMA operation, e.g., server gets a request from client, is called as *in-bound* operation. **Issuing** a one-sided RDMA operation, e.g., server sends a response back to client, is called as *out-bound* operation. It is obvious that in-bound operation requires no CPU involvement while out-bound operation does. As mentioned in Cell [20], out-bound RDMA operations in server side are more expensive than in-bound RDMA operations especially in a typical scene like one server serving many clients. For the out-bound RDMA operations, server CPU needs to involve every network operation for response sent back to clients. Since the server CPU will often become a bottleneck, we need to change the out-bound operations to in-bound operations on the server side. In fact, as we will see in the experimental results, a significant improvement can be gotten using in-bound RDMA operations instead of using out-bound RDMA operations on the server side. This also explains why *server-reply* mode is sub-optimal since it always uses out-bound RDMA operations on the RPC server.

Bypassing access amplification is a phenomenon related to the programming paradigm of *server-bypass*. Such phenomenon often leads to a significant gap between expected performance and measured performance of *server-bypass*. The expected performance corresponds to the ideal case where only one RDMA operation is required to complete a request but usually not true in reality. Since 'NO' CPU processing is on server, multiple clients need to **coordinate** their access to avoid data access conflict. It will lead to more RDMA operations i.e., more network rounds. Moreover, additional RDMA operations might be needed for meta-data probing to find where the data are on server memory. Thus, the measured performance of *server-bypass* is typically much lower than the expected. For example, Pilaf uses 3.2 RDMA operations for each GET request on average even with read-intensive workloads. The performance is even worse when conflicts are heavy (e.g., with write-intensive workloads) [21], [22], [23].

Based on the above observations and analysis, *RF-RPC* makes two important design decisions. First, to alleviate the constraint of out-bound operations, server buffers the results in its local memory instead of sending results back to clients through out-bound RDMA operations. And then, clients use *RDMA_Read* to fetch these results remotely, so that the server only handles in-bound RDMA i.e. server CPU is released from network operations. This way leverages the in-bound RDMA performance of the server's RNIC (RDMA NIC) by offloading the result-transferring responsibility from server to client. Second, *RF-RPC* requires the server to be responsible for processing the incoming requests. This way not only avoids performance degradation due to bypass access amplification, but also avoids the need of redesigning application architecture for fitting RDMA.

In addition to the above design decisions, to improve the performance further, *RF-RPC* uses batch to boost the performance of applications requiring large amount of RPC operations in parallel. Usually, each RPC invoke needs to get its response before sending the next one. This can mimic the local procedure call. However, due to the network overhead, current RPC systems often support invoking of multiple simultaneous RPC calls. Similar things happen in the multiple threading environment. Batching multiple RPC calls can benefit such applications.

TABLE 1
Design paradigms based on all possible design choices for applying RDMA (from the server's perspective).

	Request Send	Request Process	Result Return
Server-reply	In-bound RDMA	Server involved	Out-bound RDMA
Server-bypass	In-bound RDMA	Server bypassed	In-bound RDMA
RF-RPC	In-bound RDMA	Server involved	In-bound RDMA
Meaningless	In-bound RDMA	Server bypassed	Out-bound RDMA

To demonstrate the effectiveness of *RF-RPC*, we designed and implemented an in-memory key-value store using such paradigm. Experimental results show that *RF-RPC* improves the throughput[1] by 1.6× compared with *server-reply* and 4× compared with *server-bypass* under different workloads. Moreover, batching in *RF-RPC* can greatly improve the throughput by 11.4×. However, the improvement of batching is not a free lunch as it is well-known that the latency of a request will also increase when more messages are required to be batched. To mitigate this problem, a tuning mechanism is provided that can automatically choose a proper batch size. It can guarantee that the latency of most requests ($> 95\%$) be kept lower than a **user-defined** threshold while still achieve the best throughput.

The remainder of this paper is organized as follows. Section 2 discusses the design choices of *RF-RPC* and give the detailed analysis. Section 3 presents Remote Fetching RPC Paradigm (*RF-RPC*) based on design choices and gives solutions to challenges over *RF-RPC*. Section 4 describes how to apply batching technical over *RF-RPC* and the challenges to tune *RF-RPC* through the batch size. Section 5 depicts the evaluations. Section 6 shows the related works, and finally Section 7 draws concludes.

2 Design Choices and Observations

In this section, we will discuss the design choices for supporting RPC using RDMA. And also we give the detail analysis leading to the design of *RF-RPC*.

2.1 Design Choices for RDMA-Based RPC

As one of the most prevalently used communication mechanisms in distributed applications programming, RPC is well known for hiding the complexity of message-based communication for upper layer applications [8], [22], [23], [24]. There are many variations and subtleties in the implementation of RPC, which results in a variety of different (incompatible) RPC mechanisms. However, a typical RPC call consists of three steps: (1) *Request Send* step: client sends the function call identity as well as parameters to server; (2) *Request Process* step: the requests are processed and results are generated on server; (3) *Result Return* step: where the results are transferred to client [25], [26].

Table 1 illustrates the design choices for each step in an RPC call with RDMA. For Step 1, as server does not know when client may invoke an RPC call, the only choice is that client uses out-bound RDMA operations to send

1. $Throughput$: the number of requests completed per second.

the request to server. In this case, server always uses in-bound RDMA operations. Step 2 has two choices according to whether server is involved in processing the request. Porting cost is lower if server is involved. *Server-reply* follows this paradigm. *Server-bypass* does not require server to process the requests and hence reduces CPU utilization on the server. The cost is that special data structures are needed for each different application to coordinate the concurrent accesses from multiple clients. Step 3 also has two choices for *transferring data from server to client*. Server can directly send the result to client by issuing out-bound RDMA operations from server, or client can fetch the result from server's memory through RDMA-read (i.e., in-bound RDMA to server), which are adopted by *server-reply* and *server-bypass*, respectively. For completeness, Table 1 lists all possible design choices, one of which is meaningless, i.e., server does not process requests but sends results using out-bound RDMA.

2.2 Reduce Network Overhead of Server Side

Previous sections have already mentioned and made some analysis of using in-bound RDMA to replace out-bound RDMA on the server side. We also have written some micro benchmarks to quantify such improvement. Eight machines are used and the details are in Section 5. One machine act as server and others are clients. This is the typical settings of one server serving multiple clients. We measure the IOPS of issuing out-bound RDMA operations (i.e., issuing *RDMA_Write* to clients) and serving in-bound RDMA operations (i.e., receiving *RDMA_Read* from clients) on the server machine. The out-bound IOPS is tested by letting server continuously issue *RDMA_Write* operations to other clients. Each server thread randomly chooses a client machine, and issues an *RDMA_Write* operation to its memory, and repeats this operation after the current one is completed. Similarly, the in-bound IOPS is tested by letting clients issue *RDMA_Read* operations to server. Memory buffers for client threads and server threads are independent and do not interference with each other. For both tests, we launch four threads on each client machine to saturate the server's RNIC.

In order to simulate the typical work-flow of issuing RPC requests, rather than issuing asynchronous RPC requests, we always wait for an RDMA operation's completion before starting the next operation. In other words, different threads may issue RDMA operations concurrently, but at most one operation is processed by each thread. This is the traditional way of doing RPC request and response. This can be improved by using better implementation as well as batching. We will defer these discussions later in the sections discussing batching and experiments.

We measure the peak IOPS of in-bound operation (11.26MOPS) is about 5× higher than that of out-bound (2.11 MOPS) as shown in Fig. 1. This study also verifies that although *server-reply* provides good programmability, it suffers from low performance (at most 2.1 MOPS) as its IOPS is limited by the out-bound RDMA IOPS of the server. The extra CPU involvement during out-bound network operation does influence the server performance. Such a performance gap between in-bound and out-bound operation will become more severe when using relatively higher performance RDMA RNICs connected to relative lower performance CPUs.

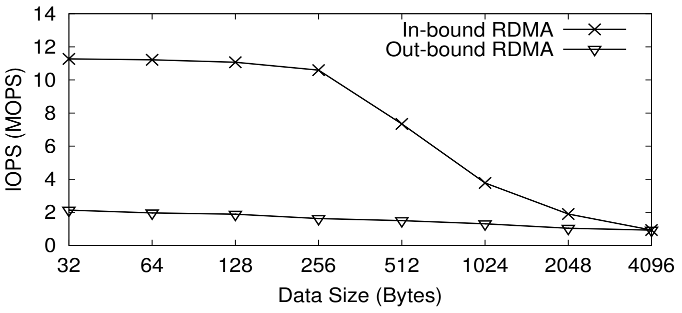


Fig. 1. The IOPS of out-bound RDMA and in-bound RDMA under different size.

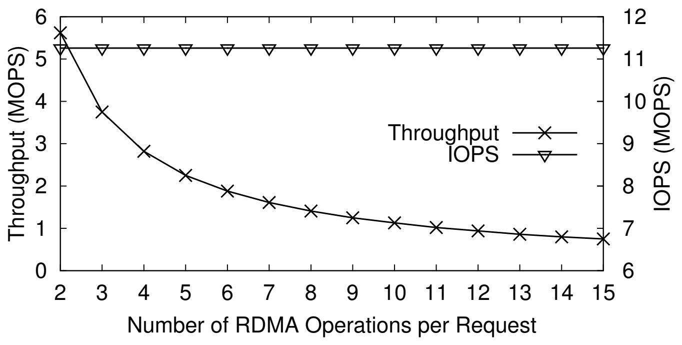


Fig. 2. The throughput declines in *server-bypass* when more RDMA operations are needed to achieve a single client request.

One can notice that when the data size is larger than 2 KB, in-bound RDMA and out-bound RDMA perform as same in IOPS. This is because the bandwidth becomes the bottleneck in this case. In contrast, when data size is less than 2 KB, in-bound RDMA significantly outperforms out-bound RDMA in IOPS.

2.3 Bypass Access Amplification

Server-bypass allows clients to directly read/write server's memory through one-sided RDMA operations without involving CPU processing on server. This has been considered a promising approach for building high performance applications with RDMA [2], [3], [5], [19]. However, as CPU processing is bypassed, the application has to rely on specific design of data structures and algorithms to do the coordination among multiple clients. This is because they may access the same memory region which leads to data race. Such programming paradigm is quite different from the traditional way of using RPC. The support for legacy RPC applications is therefore poor. The programming is also not easy.

Moreover, *server-bypass* in many cases cannot achieve good performance as expected. This is because usually multiple RDMA operations are required to complete a single request. Take Pilaf as an example, even with a 75%-filled 3-way Cuckoo hash table, a client in Pilaf has to spend 3.2 RDMA-read operations on average including metadata probing (find where the key-value pair is stored in server) and data transferring for completing a key-value GET [2].

TABLE 2
The basic APIs provided by *RF-RPC* for implementing RPC. All *local_buf*s are allocated with *malloc_buf*. Messages are directly put into these buffers for transferring through RDMA.

APIs	Description
client_send(server_id,local_buf,size)	client sends message (kept in *local_buf*) to server's memory through RDMA-write
client_recv(server_id,local_buf)	client remotely fetches message from server's memory into *local_buf* through RDMA-read
server_send(client_id,local_buf,size)	server puts message for client into *local_buf*
server_recv(client_id,local_buf)	server receives message from *local_buf*
malloc_buf(size)	allocate local buffers that are registered in the RNIC for message transferring through RDMA
free_buf(local_buf)	free *local_buf* that is allocated with *malloc_buf*

It slows down the performance boost by RDMA. Even worse, without server involved in request processing, clients have to use more RDMA operations to resolve conflicts themselves in a request. As illustrated in Fig. 2[2], when conflicts are heavy with write-intensive workloads [21], [22], [23], the throughput even decreases to below 1 MOPS due to an increasing number of RDMA operations involved. This significantly curbs the usage of *server-bypass* for a wide range of applications.

RF-RPC tries to avoid *server-bypass* model but still stick to the direct memory access capabilities provided by RDMA network operations. By using server CPU for doing request processing, the conflicts among clients can be eliminated.

2.4 Batching over RDMA

Batching is a popular technique that works by coalescing multiple messages into one bundle and sending this bundle at once. It is well-known that, if the messages sent by a machine is dominated by small messages, *batch* technology can greatly improve the overall throughput. According to recent investigations [27], [28], [29], a larger batch size will usually lead to higher throughput, until reaching the bandwidth limit of the underlying network devices. Besides, the latency[3] of RDMA operation will slowly increase at the beginning, which is also observed by other researchers [30], [31], [32]. RDMA/InfiniBand environment is more insensitive than the traditional Ethernet network. The latency of *RDMA_Write* increases from 1.12 μs to 2.04 μs when size is increased from 8 bytes to 256 bytes.

Therefore, batch over RDMA will benefit the overall throughput without increasing latency too much. This mechanism is particularly useful for applications requiring a large amount of small messages transfer over the high-speed network such as InfiniBand. For example, Koop M J, et al. [28] have tried to apply batch in RDMA-based MPI applications. Through merging multiple MPI operations upon RNIC into one operation, they achieved nearly 3.1× higher throughput in the best case.

But, as we have mentioned in Section 4, the improvement from batch is not a free lunch. Typically, the higher throughput obtained by raising the batch size is achieved with the cost of higher latency. This may not be acceptable for certain kinds of systems. As the result, the designer of transmission systems must make trade-off between latency and throughput.

2. Tested with 21 client threads connecting to a single server.
3. The latency means access the corresponding amount of data. The write latency is defined as the time for writing the data totally to the destination. This is different from the definition of *network latency*.

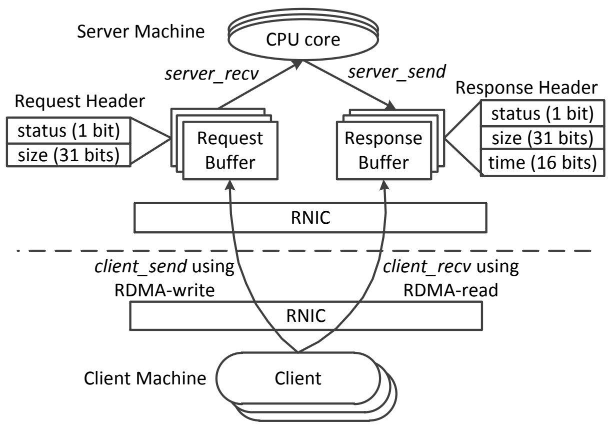


Fig. 3. The overview of *RF-RPC*.

Fortunately, according to our evaluation (details in Section 4), *RF-RPC* has a lower latency than previous paradigms. This gives more rooms for adding the batch mechanism in such a paradigm without hurting latency too much. Moreover, as we will discuss in Section 4.2, although *RF-RPC* cannot get the higher throughput and lower latency at the same time, we have developed an automatic tuning mechanism to achieve a better balance. This mechanism can automatically choose the proper batch size that *1)* lead to a near-best throughput. And at the same time, it *2)* makes sure that the latencies of most messages are less than an user-defined upper bound.

3 *RF-RPC*: Remote Fetching RPC Paradigm

Based on the design choices we discussed above, this section presents the design of Remote Fetching RPC Paradigm (*RF-RPC*), a new RDMA-based RPC paradigm that provides traditional RPC interfaces (therefore be friendly to legacy applications) as well as higher performance than both *server-reply* and *server-bypass*. First, server should process the request rather than totally bypassed, so that no application-specific data structure or redesign is needed. Second, results should be remotely fetched by the client through *RDMA_Read* instead of being sent by server, hence the server only handles in-bound RDMA operations.

3.1 Design Overview

As listed in Table 2, *RF-RPC* provides an interface that contains four basic APIs, i.e., *client_send*, *client_recv*, *server_send*, and *server_recv*, which is similar to the interface provided by TCP/IP socket. Therefore, RPC mechanisms can be built on top of *RF-RPC* by simply replacing the original TCP/IP socket interfaces with ours, which is straightforward [8], [25], [26]. As we will discuss later in Section 3.2, there are several parameters should be manually set before using *RF-RPC*. But, since *RF-RPC* makes the server to handle requests sent from clients, it does not rely on application-specific data structures and hence imposes only moderate porting cost.

Fig. 3 illustrates how to use *RF-RPC*. At the bottom of the figure, clients use *client_send* to send their requests to server's memory through *RDMA_Write* and server uses *server_recv* to get requests from local memory buffers and then process these requests, which is the same as *server-reply*. However, unlike the case with *server-reply*, server does not send results back to clients directly after it processes the requests. Instead, *server_send* called by server only writes results into **local** memory buffers, and it is clients' responsibility to use *client_recv* to remotely fetch results from server's memory through in-bound *RDMA_Read*.

In summary, *RF-RPC* combines the strength of the other two paradigms, while it also avoids their weakness. Firstly, *RF-RPC* relies on server to process the requests, which *1)* avoids the need of designing application-specific data structures, which means that it can be used to adapt many legacy applications with only moderate programming cost; *2)* also solves the bypass access amplification problem that we have described in Section 2.3.

Fig. 4 gives an example of using *RF-RPC* and *server-bypass* to implement key-value store's GET operation. From Fig. 4(b) we see that *server-bypass* involves more steps: it requires clients to probe meta-data (line 3), fetch data from server (line 5), and check data correctness and integrity through checksum (line 6). Clients have to retry if they find the data is being modified by server, or if there is a key conflict (line 10). In contrast, as shown in Fig. 4(a), *RF-RPC* just needs clients to send requests and receive results (lines 3∼4), which is compatible with *server-reply*. More importantly, the complexity of using *server-bypass* is not only embodied by the number of steps required. The above special GET procedure is specifically designed for this certain purpose and hence cannot be used in other kinds of application.

Despite the compatibility with traditional RPC, *RF-RPC* also does not waste server CPU cycles on network operations for sending results, which is different from the traditional wisdom. Instead, server only writes the results into local response buffers, but asks clients to remotely fetch the results. This eliminates the bottleneck of using out-bound RDMA operations at server side, which brings higher performance than the other two paradigms.

Since RDMA requires memory blocks to be registered to RNIC before using them, *RF-RPC* provides two APIs, *malloc_buf* and *free_buf* (see Table 2), to allocate and free buffers registered to RNIC automatically. Clients and server put messages directly into request/response memory buffers allocated from *malloc_buf*. The corresponding location information for request/response buffers are recorded by both the server and the client when client registers itself to the server. Thus, both the server and clients can directly read-/write their exclusive buffers without the need of further

```
1  int GET(int server_id, void *key, int key_size, void *value_buf){
2    r_buf=prepare_request(key, key_size, GET_MODE);
3    client_send(s_id, r_buf, sizeof(r_buf));
4    size=client_recv(s_id, value_buf);
5    return size;
6  }
```

(a) Using RF-RPC

```
1  int GET(int server_id, void *key, int key_size, void *data_buf){
2     while(true){
3        md=probe_metadata(server_id);
4        while(true){
5           data=get_data(s_id, md, data_buf);
6            if checksum of data_buf is ok:
7               break;
8         }
9         get key_size' and value_size;
10        if equal(key, key_size, data_buf, key_size')
11           break;
12    }
13    return value_size;
14 }
```

(b) Using Server-Bypass

Fig. 4. How *RF-RPC* and *server-bypass* implement GET for in-memory key-value stores at client side.

synchronizations. As shown in Fig. 3, each buffer has a header to denote the status (whether the request/response has arrived) and its size. Moreover, in each response buffer, the header also contains a two-byte variable *time* to keep the response time of server for the corresponding request. This field is used by clients to better setup the parameters for the *RF-RPC* primitives, which will be discussed in the next section.

3.2 Challenges and Solutions

To maximize the throughput of applications, *RF-RPC* faces two challenges: *1)* when clients should fetch the results from server to reduce unnecessary RDMA operations; and *2)* what default size clients should use to fetch the results so that in most cases only one *RDMA_Read* operation is required.

In our implementation, each of these two challenges is equalized to a parameter selection problem. Thus, in the rest of this section, we first show how these two challenges are transferred into parameter selection problems, and then we present our mechanism to select the optimum parameters.

For the first challenge, a straw-man design is repeatedly fetching results from server's response buffers in *client_recv*, i.e., without any interval between two retries. A similar method is used in the server side, i.e., the server will repeatedly check its request buffer for fetching new requests. It is obvious that this method can be used to achieve the best latency. However, different from the server that we assume it should spend all its CPU cycles in request processing, clients may have other responsibilities such as interacting with the user. As a result, this simple straw-man design may not be optimal as it leads to higher CPU consumption at client side and waste server's in-bound IOPS, especially

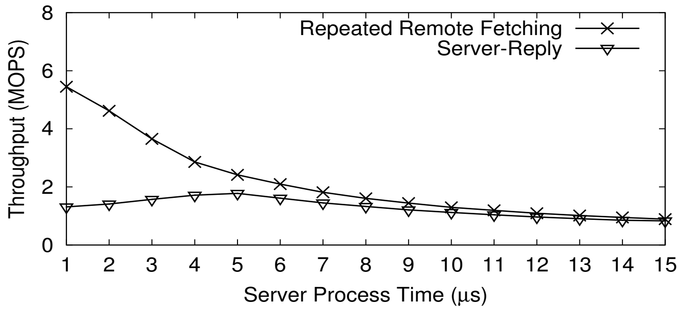


Fig. 5. The throughput difference between repeated remote fetching and *server-reply* under different server process time (P). F and S are all 1 byte for this test.

when the average number of retries is large. *RF-RPC* therefore uses a hybrid mechanism to achieve a good trade-off among latency, throughput, and clients' CPU consumption. The mechanism starts from using repeated remote fetching and then automatically switches to *server-reply* if it detects the number of retires is larger than a certain threshold R. When the number of retries is less than (or equal to) R, *RF-RPC* uses repeated remote fetching to provide higher throughput and lower latency. Otherwise, repeated remote fetching brings little throughput improvement compared with *server-reply*, and *RF-RPC* switches to *server-reply* to save CPU consumption of clients. R is the parameter for the first challenge and its selection will be discussed later.

For the second challenge, different RPC calls generate different size of results and it is impossible to generally predict the size beforehand. *RF-RPC* asks server to fill a result size of every RPC call in its response memory buffers for clients to fetch, as shown in Fig. 3. However, it is expensive if a separated *RDMA_Read* is always needed to fetch the result size before getting the data, which wastes half of the RNIC's IOPS resource. To mitigate this problem, we store the result continuously after the header in the response buffer, and we set a default *fetching size* (denoted as F) for each client. A client will fetch both the response header and the payload data from server's memory with one RDMA operation when F is not less than the total response size.Only if the real result size is larger than the fetching size, does the client need to issue another *RDMA_Read* to fetch the remaining data. This mechanism greatly reduces the average number of RDMA operations used for an RPC call, especially when the size of result is usually small. F is the parameter for the second challenge.

Parameter Selection. With R and F, we model the two challenges together as a parameter selection problem: The selection of R and F plays an important role on the throughput for upper-layer applications. In *RF-RPC*, the throughput T is determined by the following form:

$$T = \underset{R,F}{\operatorname{argmax}} f(R, F, P, S) \tag{1}$$

As we can see from the equation, the throughput (T) of an *RF-RPC* application is related to four factors:

- R - the retrying number of *RDMA_Read* from clients before it switches to *server-reply* mode;
- F - the fetching size used by the clients to read remote results from server;
- P - the process time for requests on server;
- S - the RPC call result sizes.

Among these factors, P and S are related to applications only, while R and F are related to both applications and the RDMA hardware. We therefore start by understanding how R and F are related to the given hardware capabilities. Later on, we discuss how to determine the best R and F so as to achieve great application throughput by connecting them with P and S, i.e., application characteristics.

For designing a mechanism that can automatically choose the best R and F for an application, we investigate the impact of modulating these two parameters separately and observe that it is complicated to use an equation to describe their relations and hence, it is hard to directly calculate the optimum results. However, an enumeration-based method is enough to solve the optimization problem, as, surprisingly, we find that the possible range of optimum R and F is limited.

First, Fig. 5 shows the throughput of repeated remote fetching and *server-reply* when the server process time of the requests varies, with both F and S setting to 1 byte so that only one *RDMA_Read* operation is required for fetching the result. The throughput (MOPS) is therefore the upper-bound of T for every P, no matter how F and S change. This is because: (i). making F and S not equal to each other leads to either additional RDMA operations required (when $F < S$), or no benefit at all but only bandwidth waste (when $F > S$); (ii). when F (and S) increases, throughput will only drop.

Given this upper-bound curve over P for all possible F and S, we can have an upper bound of R, i.e., R should be within [1,N], where N is the upper-bound number of *RDMA_Read* retries. If $R > N$, the throughput improvement of repeated remote fetching is limited while it consumes more clients' CPU resources than *server-reply*. The setting of N depends on the hardware configurations (similar to Fig. 5) as well as developers' inputs about their expectations on trade-off between throughput improvement and CPU consumption of clients. In this case, we choose N to be 5, which is mapped to the point whose P is 7, according to the curve above. This is because the throughput of repeated remote fetching is not significantly larger than *server-reply* when $P \geq 7\mu s$ (within 10%), while the client may spend more than twice the CPU consumption.

Second, Fig. 1 presents the IOPS of RNIC under different data size. The curve in the figure, presenting the relationship between IOPS and data size, can be divided into three ranges: [1,L), [L,H], and (H,∞). Data size smaller than L (in the first range) does not increase the throughput, due to the startup overhead of data transmission in the RNIC. Data size larger than H also does not increase the throughput, as bandwidth becomes bottleneck at this time and throughout decreases linearly with the size increasing. L and H rely on hardware configuration, and can be gotten by running benchmark once (similar to Fig. 1). For example, in our RNIC (InfiniBand) configurations, L is 256 bytes and H is 1024 bytes.

Based on the above observations, the selection of R and F is limited in $[1, N]$ and $[L, H]$ respectively, which means that only $(H - L) * N$ pairs of candidates are needed to be considered. More importantly, both N and $H - L$ are small enough for an simple enumeration. As a result, *RF-RPC* uses an enumeration-based method to decide best R and F, in which the following equation is used for comparison:

$$T = \sum_{i=1}^{M} T_i, \ where \ T_i = \begin{cases} I_{R,F} & F \geq S_i \\ I_{R,F}/2 & F < S_i \end{cases} \quad (2)$$

Specifically, for each result of an application, *RF-RPC* calculates the throughput for it (T_i). The calculation of T_i depends on the fetching size (F), the result size (S_i), and the IOPS of the RNIC under R and F ($I_{R,F}$): if $F \geq S_i$, T_i is $I_{R,F}$; if $F < S_i$, T_i is half of $I_{R,F}$ as two RDMA operations are used to fetch the whole result. $I_{R,F}$ is tested by running benchmarks only once. *RF-RPC* enumerates all possible candidates, and chooses the F and R that maximize the throughput (T) for all M results as the optimum parameters for the application. The M results of the application can be collected by pre-running it for a certain time or sampling periodically during its run. The selection complexity is $O((H-L)NM)$.

Optimization. In order to further improve the performance of *RF-RPC*, we also apply to more optimizations. First, we use inlining to reduce latency. When the payload of an *RDMA_Send* or *RDMA_Write* is smaller than 92 bytes, this payload can be inlined into the request that notifies the RNIC about this RDMA operation. Otherwise, if inline is not enabled, RNIC needs to use one more DMA (Direct Memory Access) read to fetch the payload. Usually, this technique is most effective when the payload and the corresponding header can fit into one or two cache lines, and this is why we use 92 bytes as the threshold.

Moreover, we also take advantage from the "unsignal" operations provided by RNIC. Typically, when a "signaled" RDMA operation is completed, a corresponding completion event will be pushed into the completion queue (CQ). CPU needs to poll this CQ for being noticed of this completion, which will bring extra overhead. In contrast, "unsignal" operations allow users to issue RDMA operations without waiting for its completion event, and hence can largely improve the performance. But, without signal, CPU cannot synchronize with RNIC exactly, thus we need to issue one signal RDMA operation after several unsignal RDMA operations to make sure that all these operations are completed.

Discussion. There are two more details for implementing the hybrid mechanism to switch between repeated remote fetching and *server-reply*. First, both client and server maintain a *mode_flag* for each pair of $\langle$*client_id, RPC_id*$\rangle$, which designates the current paradigm in usage. This flag can only be modified by the corresponding client (by a local write to the local flag and an *RDMA_Write* to server's flag), and server gets to know the current paradigm by checking its local mode_flag. Initially, the flag is set to repeated remote fetching and hence client will continuously fetch results from server. If the number of failed retries becomes larger than R, client will update the mode_flag (both local and remote) to *server-reply* and switch itself to *server-reply*, i.e., waiting until the result is sent from server. In contrast, if client is currently in *server-reply*, it will record the last response time and switch back to repeated remote fetching if it finds the response time becomes shorter. *RF-RPC* records the response time for completing the request in the header of the response buffer (see Fig. 3), which will be gotten by client through *RDMA_Read*.

Second, some requests with unexpectedly long server process time may cause unnecessary switch between repeated remote fetching and *server-reply*. To avoid this phenomenon, *RF-RPC* only switches to *server-reply* if it finds a pre-defined number (e.g., two) of continuous RPC calls suffer from 5 failed retries of remote fetching. Otherwise, *RF-RPC* remains in repeated remote fetching mode. According to the evaluation in Section 5.3.2, only 0.2% of the requests have unexpectedly long process time for applications. Thus, it is quite rare that two (or more) continuous RPC calls suffer from unexpectedly long process time.

4 Batching over *RF-RPC*

According to recent investigations [33], about more than 99% of messages transmitted in real-world data centers are small packages whose sizes are less than 1KB. Thus, as we have discussed in Section 2.2, in such an environment the upper bound of maximum IOPS of an RNIC will be reached earlier than the upper bound of its bandwidth. As a result, although our novel *RF-RPC* model can achieve a higher IOPS than the traditional *server-reply* and *server-bypass* paradigms, it is still not able to fully utilize the bandwidth provided by underlying InfiniBand devices.

To mitigate this problem, in this paper, we propose the usages of batching. Specifically, in Section 4.1, we will present the method of using buffering to integrate batching with *RF-RPC*. Different from *server-bypass* that totally bypass the server, *RF-RPC* involves the server in message handling, which make this integration quite straightforward. However, although *RF-RPC* can also achieve a better latency than *server-reply* and *server-bypass*, it is not free from the trade-off between higher throughput and lower latency. To ease the users' overhead of manually modulating the batch size, in Section 4.2, we will discuss the design of an automatic parameter tuning mechanism, which can calculate the proper batch size by giving only a pre-defined upper bound of latency.

4.1 Design Details of Batching

Batch Format Fig. 6 presents the format of the request batch and the result batch, respectively. As we can see from the figure, every single request/result is considered as an entry of the batch and is stored immediately after a 32-bit integer that describes its size. Besides the payload, both the request and result batches have a header that consists of *1)* a single status bit; *2)* a 31-bit integer that is the size of the whole payload; *3)* an auxiliary field. The usage of the auxiliary field in response batch's header is already discussed in Section 3.2. In contrast, the auxiliary field in request batch's header is unique in the batch mode, it is used for automatic batch-size modulating that we will describe in the next subsection. Moreover, there is one more status bit in the tail of a request batch. These meta-data are used for making sure that the server/client is able to decide whether the request/result batch it received/fetched is complete or not, whose usages will be discussed in the rest of this section.

Sending a Request Batch For using the batching mechanism, users can define a size limit L (typically set to

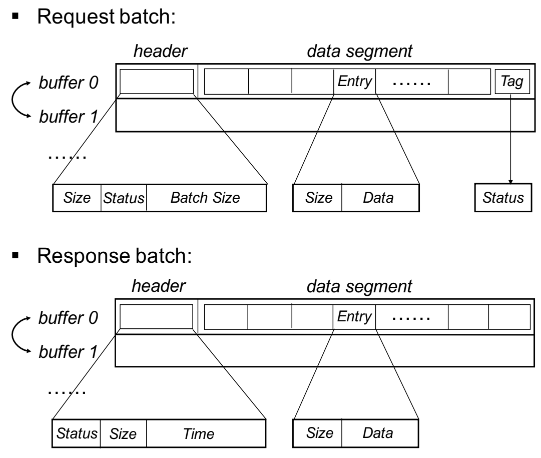


Fig. 6. Design of batching packet which includes request batch and response batch.

2048 bytes), a time-out threshold T, and a number N that indicates the maximum number of entries in a batch. With these parameters, the clients will first buffer the requests and only send them in a batch when *1)* there are already N requests buffered; *2)* the payload of the batch exceeds L; or *3)* there is no more request after passing T seconds. We set the timeout threshold T as 5 ms in our evaluations.

The above procedures are similar to existing ones used in other paradigms. But, in *RF-RPC*, requests are sent to the server via one-sided *RDMA_Write* operations by the clients, which bypasses the server's CPU. As a result, we need to design a mechanism that enables the server to check whether a batch of request is sent and, more importantly, whether the received batch is completed. To provide this capability, we reserve two status bits in both the front and the tail of the request batch. Initially, the server will clear its local buffer so that both these two bits are zero in the server side. After the client prepared a full batch, it set both of these two bits to one and send them to the server. Since the *RDMA_Write* operation is guaranteed to write the data in a sequential order [3], [34], the server can be aware the batch sending is started after the first status bit of its local buffer is changed to one, and the completion of this sending after the tail bit is also changed to one.

Fetching Result Batches After the server receives a complete batch of requests, it can process them with arbitrary functions as required. The results are first buffered in a result buffer and fetched back by the clients through one-sided *RDMA_Read* operations. Similar to the sending procedure, we use the status bit to enable clients to check whether it has successfully fetched a complete result batch or not. The head bit is first set to zero and only changed to one after the server has finished writing all the payloads and the other meta-data fields. As a result, the client should repeat its remote fetching until this bit is set.

The tail status bit is not used in result batch because the client can check whether all the sent requests are processed by counting the number of results returned. This mechanism enables us to return an arbitrary length result that is not necessarily equal-size to the request. In other words, for a single request batch, its results may be split into multiple result batches if the result size is larger than the request. We assure that the payload of each result batch is not larger than L so that a fixed size buffer is enough.

Large Entry Former discussions focus on small entries whose size are less than L. In real-world cases, there may be extreme cases that the size of a single request/response is larger than L. However, the difference between *RF-RPC* and *server-reply*'s IOPS becomes narrow after the size of each packet becomes larger than 8192 bytes. Thus, we simply fall back to *server-reply* for such extreme cases.

Pipeline To further reduce the latency cost of using batching, we also enable users to send multiple request batches without waiting for all the results of the former request batch is fetched. This pipeline mechanism overlaps the sending and fetching procedure and hence can reduce the average latency. Specifically, the client can continuously send multiple request batches if the number of requests whose results have not been returned is not larger than a certain threshold. The implementation of this mechanism just requires that the server should allocate multiple buffers in the local for reserving the requests and buffering the results.

4.2 Batch Size Selection

As we have discussed in Section 2.4, the increasing of batch size will not only increase the throughput but also enlarge the latency, which is not desirable by users. This phenomenon deserves more serious considerations when batch size is larger.

Although users can modulate the batch size manually, it is usually a tedious work as the achieved latency depends on many factors (e.g., the current network situation, the load of server/client) and hence is unpredictable. More importantly, a fixed batch size may not always achieve the best result as the environment and workload are continuously changing. As a result, we propose an automatic tuning mechanism for facilitating the usages. With this mechanism, users only need to specify a threshold of latency l_{limit} and a certain degree of tolerance α, then the system will automatically choose a batch size that achieves the best-possible throughput while, at the same time, assures that at least $1-\alpha\%$ of the requests' latency is lower than l_{limit}.

$$s_{next} = \begin{cases} s_{now}-1, & k > \alpha\% \\ s_{now}+1, & k < 0.1\times\alpha\% \\ s_{now}, & other\ conditions \end{cases} \tag{3}$$

- s_{next} - the recommend batch size for server side;
- s_{now} - the batch size at present;
- k - the proportion of latency which exceeds the l_{limit};
- l_{limit} - the limit upper bound latency;

Specifically, the tuning mechanism is implemented by the following procedures. First, the client should record the latency of each request it sends, which will be used for the later calculation. With such information, the client can track the current status by calculating what proportion of requests' latency are higher than the user given threshold l_{limit}. If the calculation result k is higher than α, the batch size should be reduced for lowering the average latency, which is achieved by changing the batch size from s_{now} to

$s_{next} := s_{now} - 1$. This new parameter will be stored in the 16-bit auxiliary field that we have demonstrated in Fig. 6, so that the server can synchronize with client to use the same batch size. Similarly, if the calculation result k is lower than $0.1 \times \alpha$, the batch size is enlarged ($s_{next} := s_{now} + 1$) to obtain a better throughput. The reason why we use a fixed step 1 is because that, according to our evaluation (more details in Section 5.6), the range of best batch size is very limited ($1 \sim 128$). As a result, a simple method that uses a small delta step is already enough to modulate the parameter to the best one very quickly.

5 Evaluation

5.1 Experiment Setup

An *RF-RPC* based application. To evaluate how applications can easily and effectively use *RF-RPC*, we implement *Jakiro*, an *RF-RPC* based in-memory key-value store similar as Memcached [18] that exports RPC interfaces (i.e., PUT and GET) for clients to operate key-value pairs. Similar to many existing works [35], [36], [37], we hash the key to distributing data to the different partition. The in-memory data structure of *Jakiro* is partitioned across different server threads in an Exclusive Read Exclusive Write (EREW) [35] fashion, i.e., each server thread only accesses its own data partition. In our implementation, the in-memory structure contains a number of buckets, each of which contains eight slots 4. A slot is used to keep the information of a key-value pair (such as the memory address that is keeping the pair). When a bucket is full, we use a strict LRU (Least Recently Used) policy for slot eviction in this bucket.

Cluster. For the evaluation, we use a cluster based on InfiniBand for the evaluation. The cluster contains eight machines, each of which is equipped with dual 8-core CPUs (Intel Xeon E5-2640 v2, 2.0 GHz), 96 GB memory space, and a Mellanox ConnectX-3 InfiniBand NIC (MT27500, 40 Gbps). All of these machines are connected by an 18-port Mellanox InfiniScale-IV switch. Though our RNIC supports dual-port, we just use one of the two ports to simplify our experiments. The machines run MLNX-OFED-LINUX-2.3-2.0.0 driver provided by Mellanox for Ubuntu 14.04 [38].

Workloads. Unless explicitly specified, we choose key-value pairs with 16-byte key and 32-byte value. This is aligned with the real-world workloads for in-memory key-value store [33], [36], [39], [40]. We use YCSB [41] to uniformly generate 128 million key-value pairs off-line for the experiment. The skewed workload is generated according to Zipf distribution with parameter .99. For workloads with 32-byte value-size, we pre-run such workloads and select R (RDMA-read retrying number) as 5 and F (fetching size) as 256 bytes.

Comparison. We firstly compare *Jakiro* with Pilaf [2] that adopts *server-bypass* in Section 5.2. In Section 5.3, we compare *Jakiro* with two in-memory key-value systems that use *server-reply*. The first system is ServerReply, which is extended from *Jakiro* and differs from *Jakiro* in that the server thread directly sends the result back to the client thread through *RDMA_Write*. The other system is RDMA-based Memcached (denoted as RDMA-Memcached) [7]. In RDMA-Memcached, the server thread sends status or notification information to the client thread after it processes the requests, and the client thread relies on the information to do further RDMA operations. We run RDMA-Memcached in memory mode without interacting with the underlying persistent storage. We use one machine as the server machine and other 7 machines as the client machines to run *Jakiro*, ServerReply, and RDMA-Memcached. Five threads are launched in each client machine (35 client threads in total), which are enough to saturate the server's RNIC.

5.2 Comparison with *Server-Bypass*

As mentioned before, *server-bypass* sometimes cannot achieve good performance as expected, due to bypass access amplification. Pilaf [2] is a state-of-the-art in-memory key-value store using *server-bypass*. In Pilaf, even with a specific-design memory-efficient 3-way Cuckoo hash table, client still needs 3.2 RDMA-read operations in average to complete a GET request. Specifically, these 3.2 lookups include the round trips used for *1)* finding where the key-value pair is stored in server; and *2)* transforming the real data to client. It does not include the process for addressing collisions, so that the number can be even bigger if the chance of collision is high (a write-intense workload rather than the read-intense one tested by Pilaf). Thus, in theory, the peak throughput of Pilaf in the experimental cluster we use is only 3.5 MOPS for uniform and read-intensive workload (95% GET). According to our evaluation, the peak throughput of Jakiro that adopts *RF-RPC* reaches 5.5 MOPS, which is 57.14% higher than Pilaf's[4].

In contrast, the peak throughput of Jakiro (achieved with 35 client threads) is half of the peak in-bound RDMA IOPS of this server's RNIC (11.2 MOPS, as illustrated in Fig. 1). A closer look reveals that thanks to the fetch-size decision mechanism in *RF-RPC*, only 2.005 round-trips on average are needed in *Jakiro* to successfully complete a key-value RPC call: one is to send the request through RDMA-write and the other 1.005 is to fetch the result through RDMA-read, making almost no waste of the fetch operations for RPC results.

Moreover, a client in *server-bypass* needs more RDMA operations to complete a request when the conflicts are heavy. According to Mitchell et al. [2], the peak throughput of Pilaf under uniform workload with 50% GET is only 1.3 MOPS. The experimental RNICs for Pilaf are 20 Gbps Mellanox InfiniBand NICs [2]. We also run *Jakiro* in a cluster of six machines equipped with 20 Gbps Mellanox Infiniband NICs, under uniform and 50% GET workload. The peak throughput of *Jakiro* on values from 32 bytes to 256 bytes is about 5.4 MOPS, which is 4× as high as that of Pilaf.

5.3 Comparison with *Server-Reply*

In this section, we compare *RF-RPC* with *server-reply* and show the performance improvement brought by *RF-RPC*. Besides, we also bench the performance of HERD [36] in the same testing environment. HERD can reach 8.8MOPS comparing with 5.5MOPS of Jakiro with 32-byte value size. But HERD uses UD model which can not provide reliable connection. Since the fundamental physical devices are

4. Since the code of Pilaf is not available, we directly compare with the number reported in its paper. The platform we used to test Jakiro is the same as the environment reported by Pilaf.

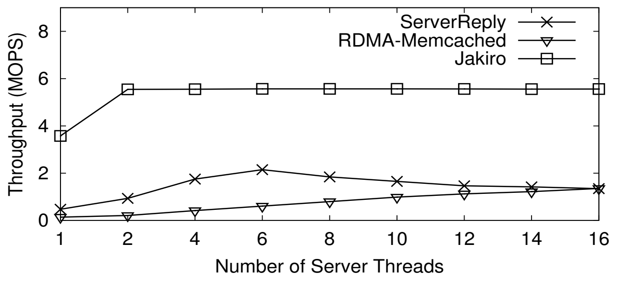


Fig. 7. Throughput of *Jakiro*, ServerReply, and RDMA-Memcached on 32-byte value size. *Jakiro* outperforms ServerReply and RDMA-Memcached by about 160%∼310% in throughput.

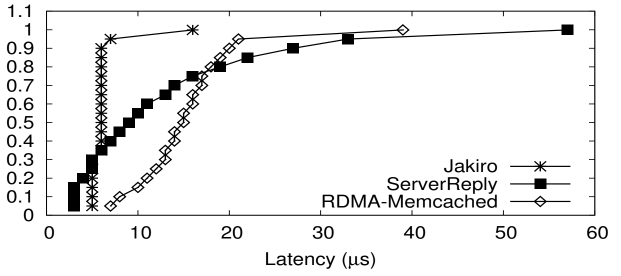


Fig. 8. CDF of latency of *Jakiro*, ServerReply, and RDMA-Memcached on value size of 32 bytes.

working in different modes, it is inappropriate to compare HERD with Jakiro. Thus we do not make the comparison.

5.3.1 Comparison on Throughput

As shown in Fig. 7, the peak throughput of *Jakiro* on 32-byte value is 5.5 MOPS. It is about 158% higher than that of ServerReply (2.1 MOPS) and about 310% higher than that of RDMA-Memcached (1.3 MOPS) respectively. Although ServerReply also requires only two round-trips to complete an RPC call (one is to send the request by the client thread and the other is to reply by the server), its peak throughput on key-value pairs is limited by the RNIC's out-bound RDMA IOPS (2.1 MOPS). Moreover, when the number of server threads increases (larger than 6), the throughput of ServerReply decreases, due to the poor scalability of the RNIC's out-bound RDMA operations. In contrast, the server thread in *Jakiro* does not have to spend cycles on network communication. Therefore, launching more than 2 server threads is enough to serve requests when the server's RNIC is saturated by the clients, and the peak throughput of *Jakiro* remains 5.5 MOPS in this case (see Fig. 7).

5.3.2 Comparison on Latency

The average latency of *Jakiro* on key-value pairs with 32-byte value is 5.78 μs. It beats ServerReply's average latency (12.06 μs) by 108% and RDMA-Memcached's average latency (14.76 μs) by 155%. Fig. 8 illustrates the cumulative probability distribution of latency of the three systems when all of them achieve peak throughput with the uniform and read-intensive workload. We see that ServerReply has lower 15-percentile latency than *Jakiro*. This is caused by the following: (1) a single RDMA-write has lower latency than a single RDMA-read, as RDMA-write needs less state and operations than RDMA-read in the RNIC. Such phenomenon

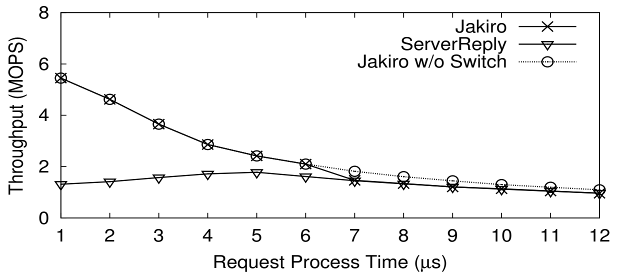


Fig. 9. Throughput of *Jakiro* and ServerReply under different request process time at server side. *Jakiro* and ServerReply have comparable performance when the request process time becomes larger, as *RF-RPC* automatically switches to *server-reply*.

also has been observed in HERD [36] and RDMA-PVFS [14]; (2) server sends back the result to client immediately in ServerReply, while in *Jakiro* it has to (possibly) pay an extra delay before clients come to fetch the result.

However, as the RNIC has limitation on out-bound RDMA, ServerReply imposes higher latency (e.g., 50-percentile latency or 99-percentile latency) than *Jakiro* when more operations are observed. In *Jakiro*, about 99% RPC calls are below 7 μs, which is significantly better than ServerReply and RDMA-Memcached. Moreover, all the three RDMA-based systems suffer from the long-tail latency issue, while the one from *Jakiro* is shortest.

Additionally, our auto-switch mechanism balances latency and throughput. For *Jakiro*, some RPC calls suffer higher latency (15∼17 μs) because they have to go through more round-trips (4−8) for request sending and result fetching. However, the RPC calls that need more than 2 round-trips to complete only account for a small proportion (0.2%) and no two continuous calls suffer from 5 RDMA-read retries to fetch the result. Developers using *RF-RPC* can avoid unnecessary switch between repeated remote fetching and *server-reply* to balance throughput and latency, by configuring how many contiguous RPCs that exceed the switch point should happen before a real switch happens. In this case, *Jakiro* achieves 5.5 MOPS peak throughput as well as having a low latency.

Fig. 9 displays the performance of *Jakiro* and ServerReply under different request process time (using 16 server threads and 35 client threads). We mimic the processing procedure with a for loop, so that we can exactly control the process time by using the RDTSC instruction. As we see from the figure, when the request process time is less than 7 μs, repeated remote fetching with 5 times is enough to successfully fetch the result back in *RF-RPC*. Thus, the throughput of *Jakiro* is about 30%∼320% higher than that of ServerReply. ServerReply is still bounded by out-bound RDMA of the server's RNIC in this case. When the request process time is larger than or equal to 7 μs, the performance of *Jakiro* mainly depends on the server load instead of the performance asymmetry from the in-bound and out-bound operations. Under this circumstance, repeated remote fetching does not increase the performance and *RF-RPC* switches to *server-reply* after 5 failed retries in two continuous RPC calls with our current hardware configuration. *Jakiro* and ServerReply have comparable performance when the request process time (i.e., server load) becomes larger, as both

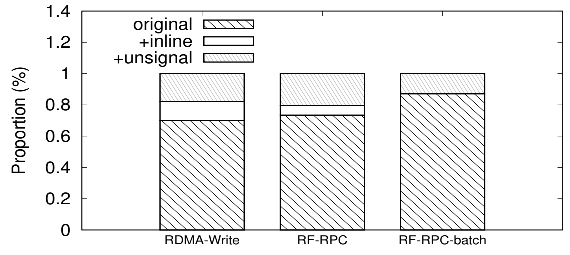


Fig. 10. The performance gain of two optimizations. In RDMA-Write, the client write data to the server.

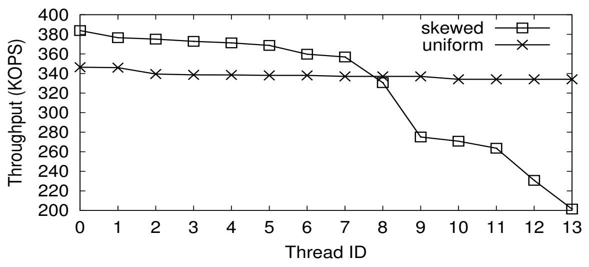


Fig. 11. Throughput of different thread under uniform and skewed workloads, when every thread has 3 QPs.

of them require more server resources at this time. Therefore, *RF-RPC* achieves good performance for applications under different server loads.

5.4 Breakdown of RDMA Optimizations

To evaluate the optimizations mentioned in Section 3.2, we measure the performance gain of two optimizations ("unsignal" and "inline") as shown in Fig. 10. We don't apply the inline optimization to *RF-PRC-batch* due to the payload size of *RF-RPC-batch* exceeds one or two cache-line size (as we described before, inline optimization doesn't suit for larger size payload).

As we can see from this figure, the"inline" and "unsignal" optimization will bring with nearly 30% throughput improvements in both RDMA-Write and *RF-RPC*. Furthermore, unsignal optimization will bring more improvements compared with inline optimization. The inline optimization is only available for *RDMA_Write* and *RDMA_Send/Recv* operations. So the effects of inline optimization are not significant in *RF-RPC* and *RF-RPC-batch* due to that we can only benefit from inlining data when sending requests via *RDMA_Write*.

5.5 Performance of Multi-QPs/Single-QP per-thread

As we have mentioned in Section 3.1, although the number is less than *server-reply*, *RF-RPC* still needs to use N×M queue pairs (QPs) if the number of server processes is N and the number of clients is M. In order to manage these QPs, there are mainly two kinds of mapping mechanisms. Specifically, the most straightforward method is creating one thread for every QP and let the OS kernel to schedule these threads. In contrast, one can also create only one thread for every CPU core and assign QPs to these threads equally. If a thread is assigned with multiple QPs, it can poll them in a round robin fashion. This second method is adopted by our

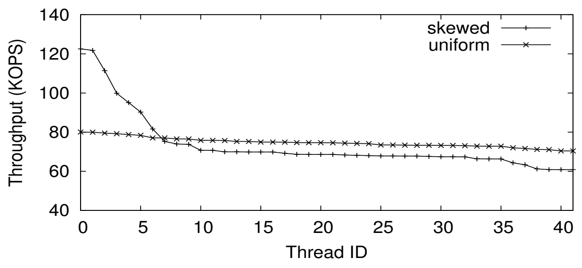


Fig. 12. Throughput of different thread under uniform and skewed workloads, when every thread has one QP.

implementation because it avoids the cost of thread context switch and hence leads to a better performance.

In order to justify our choice, we evaluate both kinds of mechanisms with 42 clients that continuously issuing key-value operations (32 bytes value, read-intensive (95% GET) workload). The results are presented in Fig. 12 and Fig. 11, in which the server creates 42 threads (one per client) and 14 threads (one per core and each thread is assigned with 3 QPs), respectively. As we can see from the figure, when the distribution of keys follows a uniform distribution, the total throughput is 4.73 MOPS if the thread is only created for every CPU core. This number is about 1.51 × larger than the simpler single-QP per-thread setting.

Moreover, we've also evaluated these two kinds of mechanisms under skewed workload (Zipf distribution with parameter .99). The results are two-fold. First, compared with the uniform workload, there exists an obvious difference between the throughput of different threads. However, the sum of these per-core throughput seems to be equal to the result of uniform workload. As a result, multi-QPs per-thread is still better than single-QP per-thread in the skewed workload. This phenomenon is also observed by A. Kalia, et al. [36] in their evaluation of HERD.

5.6 Comparison under Different Batch Size

In this section, we evaluate the effect of using batching and our automatic batch size choosing mechanism. Specifically, we integrate batching into both *RF-RPC* and *server-reply*, which result in *RF-RPC-batch* and *server-reply-batch* respectively. The comparison between these two systems demonstrates that *RF-RPC-batch* can achieve both higher throughput and lower latency than *server-reply-batch* when they are set with the same batch size. Moreover, we also present the evaluation results of *RF-RPC-auto-batch*, whose batch size is automatically selected to ensure that at most 5% of the messages latency is higher than the user-given upper bound (i.e., $k = 5\%$). All the experiments are executed on a setting of 16 server processes connected with 35 client processes. The size of value and key are 32 bytes and 16 bytes, respectively.

Throughput Fig. 13 presents a comparison between the above three systems. For *RF-RPC-batch* and *server-reply-batch*, we manually modulate the batch size from 2 to 96 and report the corresponding throughput and latency in the figure.

First, We make performance comparisons to show the improvement of batching. The throughput of *RF-RPC-batch* (62.7 MOPS) can reach 11.3× in peak compared to *RF-RPC*.

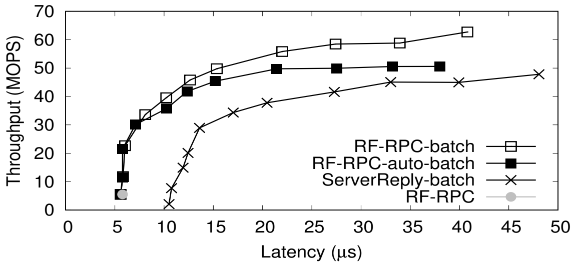


Fig. 13. Throughput of *RF-RPC-batch*, *RF-RPC-auto-batch*, *ServerReply-batch* and *RF-RPC* under different latency (variable is batch size).

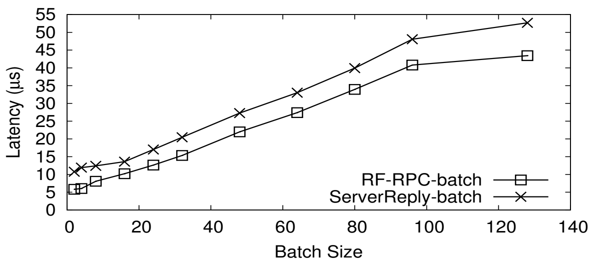


Fig. 14. Latency of *RF-RPC-batch* and *server reply-batch* under different batch size.

Besides, *RF-RPC-batch* can perform 3.9× throughput than *RF-RPC* with only 8% increase of latency (6.17μs) when batch size is 4.

As we can see from the figure, with the increasing of batch size, both the throughput of *RF-RPC-batch* and *server-reply-batch* are increased, which also leads to an increasing in latency. However, although *RF-RPC-batch* does not change the fact that one cannot achieve the best of throughput and latency simultaneously, the results show that *RF-RPC-batch* can achieve a better throughput at any specific latency. As a illustration, when batch size is set to 24, *RF-RPC-batch* can execute 45.80 MOPS at a latency of only 12.63 μs, while *server-reply-batch*'s throughput is only 21.67 MOPS if an average of 12.63 μs latency is required.

Finally, we evaluate the achieved throughput of *RF-RPC-auto-batch* when different upper bound of latency is given, which results are also plotted in Fig. 13. We can see from the figure that our automatic parameter tuning mechanism works well as it can approximate the best-possible throughput for users. The reason why *RF-RPC-auto-batch*'s throughput is lower than *RF-RPC-batch* is that: *1)* for *RF-RPC-auto-batch*, the reported latency is the user-given upper bound, so that the system should ensure that less than 5% of the requests will be returned later than that threshold; in contrast, *2)* the latency of the other two systems are only the average latency, which does not give any guarantee of how many percentages of requests' latency are higher than that (typically much larger than 5%).

Latency Fig. 14 presents a more detailed demonstration of the relationship between batch size and latency. As we can see, the latency is nearly linear correlation with the batch size. More importantly, as we have mentioned before, *RF-RPC-batch* can always achieve a better latency than *server-reply-batch* when they are using the same batch size. According to our results, the latency of *server-reply-batch* is about 18%∼100% higher than *RF-RPC-batch*.

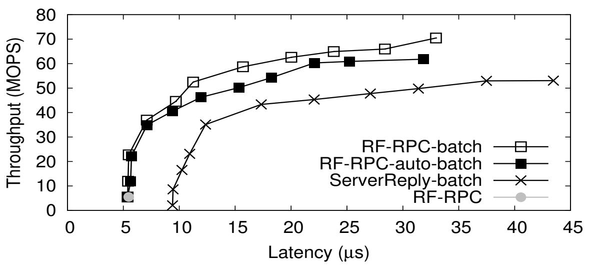


Fig. 15. With skewed workload, throughput of *RF-RPC-batch*, *RF-RPC-auto-batch*, *ServerReply-batch* and *RF-RPC* under different latency.

TABLE 3
Throughput of *RF-RPC* and *ServerReply* under different RNIC and with/without batch (using only 4 cores in server side).

	ConnectX-3		ConnectX-4	
	ServerReply	Jakiro	ServerReply	Jakiro
Without Batch	1.66 MOPS	3.05 MOPS	2.26 MOPS	4.48 MOPS
Batch Size is 16	10.1 MOPS	12.3 MOPS	16.7 MOPS	22.6 MOPS

Skewed Workload We also measure the three systems, *RF-RPC-auto-batch*, *RF-RPC-batch* and *server-reply-batch* with skewed workload. The keys in skewed workload are generated according to a Zipf distributed with parameter .99. The results are presented in Fig. 15, which shows that the skewness of workload does not impact the results much.

5.7 Faster RNIC

Fundamentally, *RF-RPC* should fit to any RDMA hardware configuration with the similar system settings i.e. one server serving multiple clients. We verified *RF-RPC* using *Jakiro* on a cluster with a different configuration i.e. equipped with faster RNIC. This cluster contains five machines and each machine is equipped with one 4-core CPU (Intel Xeon E5-2407 v2, 2.4GHz) and 32 GB memory space. They are interconnected with Mellanox ConnextX-4 InfiniBand NIC (MT27700, single-port) which can provide 100Gbps bandwidth. All of these machines run Ubuntu 16.04, with MLNX-OFED-LINUX-4.0-2.00 driver provided by Mellanox for Ubuntu 16.04. This cluster is much smaller than the one used in previous experiments. Four of the five machines act as client machines and each of them creates 4 clients (i.e., 4×4=16 clients), which are served by 4 server processes that hosted in the single server machine. All the experiment are using 32 bytes value size and uniform read-intensive (95% GET) workload.

Throughput Table 3 presents our evaluation results. The throughput of Jakiro under ConnectX-4 is about 47% higher than Jakiro under ConnectX-3. The increasing is limited because we have only 4 cores in the server side. As a result, the server's CPU becomes the bottleneck and hence cannot fully unleash the potential of 100Gbps InfiniBand NIC.

However, even in this circumstance, the throughput of Jakiro is still nearly 98% higher than ServerReply (with ConnectX-4 InfiniBand NIC). This result demonstrates that our novel paradigm is general enough to be used in different kinds of RNIC. Similarly, when the batch size is 16, Jakiro's throughput is 22.6 MOPS, which is about 84% higher than Jakiro with the batch size under ConnectX-3, and it is also higher than the corresponding ServerReply setting.

TABLE 4
Throughput of *RF-RPC* and *ServerReply* under different RDMA protocols (one-to-one configuration).

Type	RoCE	Infiniband
ServerReply	1.21 MOPS	1.30 MOPS
RF-RPC	1.72 MOPS	1.85 MOPS

Latency When running Jakiro with ConnectX-4, the average latency is 3.60 μs, which is *1)* about 47% faster than the average latency of running ServerReply under ConnectX-4 (6.77 μs); and *2)* about 29% faster than running Jakiro under ConnectX-3 (5.06 μs).

5.8 Ethernet-Based RDMA Protocol

We choose the RoCE to evaluate RF-RPC due to that RoCE/RoCEv2 is more commonly used than iWARP (Only a single vendor (Chelsio) supports iWARP on its own products). To evaluate the effects of using RoCE, we use a cluster based on Ethernet. Mellanox ConnectX VPI NIC series can support both Infiniband and Ethernet modes. According to this, we use ConnectX-3 VPI NIC and initiate it as a 10 GigE adapter card with Link Layer is Ethernet. So we can run RoCE protocol upon the Ethernet. This cluster contains three machines, each of which is equipped with an 8-core CPUs (Intel Xeon E5-2640 v2, 2.0 GHz), 96 GB memory space. We use one of three machines as server and others as clients.

To better demonstrate the difference between RoCE and Infiniband, we use one-to-one configurations (one server thread to one client thread). As shown in Table 4, our paradigm can also fit the RoCE protocol with little performance loss. Moreover, the gap between RF-RPC and ServerReply on RoCE will be less than on Infiniband environment. The reason is that the link layer of RoCE is based on Ethernet, and some special features which are friendly to out-bound verbs don't exist in Ethernet link layer.

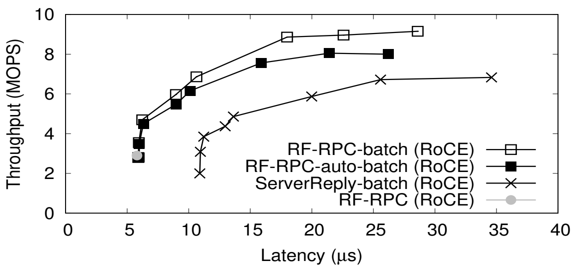


Fig. 16. Throughput of *RF-RPC-batch*, *RF-RPC-auto-batch*, *ServerReply-batch* and *RF-RPC* under different latency with RoCE.

With RoCE environment, We also measure *RF-RPC-batch* with different batch size in the Fig. 16. Two of the three machines act as client machines and each of them creates 8 clients (i.e., 2×8=16 clients), which are served by 8 server processes that hosted in the single server machine. All the experiments are using 32 bytes value size and uniform read-intensive (95% GET) workloads. For *RF-RPC-batch* and *server-reply-batch*, we manually modulate the batch size from 2 to 96 and report the corresponding throughput and latency. As shown in Fig. 16, The increasing trending is similar as batching over Infiniband: *1)* when batch size is smaller, throughput increase rapidly with raising little increment of latency, and *2)* with larger batch size, throughput stop to increase and latency will increase rapidly. Another observation is that the throughput in RoCE environment cannot reach very high compared with Infiniband environment. There are two reasons *1)* the bandwidth of RoCE is only 10Gbps (compared with 40Gbps of Inifiniband) and *2)* two clients cannot fully saturate the network (compared with 7 clients in our Infiniband cluster). The last observation is that the latency in RoCE environment will be little lower. Obviously, with fewer clients, the latency of server handling requests will reduce. So it will lead to lower total latency.

6 Related Work

Different Queue Pair Types. There are three kinds of queue pair types that can be used to support RDMA. *RF-RPC*, like all the *server-bypass* solutions, requires the use of Reliable Connection (RC), because it is the only queue pair type that supports both one-sided *RDMA-Read* and *RDMA-Write*. The other two kinds of queue pair types, i.e., Unreliable Connection (UC) and Unreliable Datagram (UD), provide only unreliable transport services, so that there is not any guarantee that the messages are received by the other side: corrupted and silently dropped are both possible. Moreover, they only support limited APIs (UC does not support *RDMA-Read*, while UD neither supports *RDMA-Read* nor *RDMA-Write*), which prohibits users from relieving server from handling packets.

There are some works that build key-value stores upon UD and UC, such as HERD [36] and FaSST [42], which may achieve higher performance than RC-based solutions. This is reasonable because the reliable-guaranteeing mechanism itself imposes certain overhead, but it is at a cost of requiring the applications to handle many subtle problems, such as message lost, reorder and duplication. Considering the fatal outcome, even if such subtle problems rarely happen in the real-world [42], they cannot be simply ignored. Moreover, as these UD and UC based methods require server to send results back to clients, the consumed server CPU cycles may become a bottleneck when the IOPS is extremely high. Figure 8 also demonstrate that *Jakiro* achieves better latency so that *RF-RPC* is more suitable for latency-sensitive applications. Anuj et al. [27] also present a guideline of using RDMA, which provides many useful optimization techniques. However, the main techniques the paper presents, such as Doorbell batching, can only be used for UD-based solutions. The other techniques that related to the hardware are orthogonal to our paradigm, so they can be used to further improve *RF-RPC*'s performance.

Different Paradigms. Other existing RDMA-based solutions usually apply *server-reply* [7], [8], [14], [15], [16], [17], [43], *server-bypass* [3], [5], [19], [30], or a combination of these two paradigms [2], [20]. As we have demonstrated in Section 5, *RF-RPC* can be faster than these two traditional paradigms, as it *1)* uses only in-bound RDMA in the server side; and *2)* avoids the "bypass access amplification" problem. In those server-bypass-based applications, multiple RDMA operations (3∼6) are usually needed to complete a request [2], which number can be even higher when conflicts from multiple clients are heavy.

RF-RPC also involves only moderate migration overhead for those legacy applications that use RPC, because it does not require any application-specific data structures to be feasible. In contrast, Pilaf and C-Hint have to propose solutions to reason about data consistency [2], [5]. DrTM relies on explicit locks and high-performance HTM for data race coordination [19].

Wukong [30], [44] is an distributed graph-based RDF store. Wukong provides highly concurrent and low-latency queries by using DrTM-KV, which is an one-sided RDMA-friendly distributed key-value store derived from DrTM.

FaRM [3], [45] is also a memory distributed computing platform that adopts *server-bypass*. It is reported to be able to perform 167 million key-value lookups per second with 20 machines (i.e., about 8M requests per second per server), which is larger than *RF-RPC*. Nevertheless, FaRM uses Hopscotch hashing that leads to something like "batching the requests" for performance. With FaRM, a client needs to fetch $N * (S_k + S_v)$ data to get a single key-value pair, where N is usually larger than 6, S_k and S_v are the size of key and value respectively. As a result, *1)* the average latency for FaRM to get key-value pairs with 16-byte key and 32-byte value is 35μs, which is about 5$\times$ higher than that of *RF-RPC*; and *2)* a lot of the bandwidth and MOPS will be wasted if only a few data in the N fetched key-value pairs are used.

Some works aim to simplify RDMA programming paradigm (*RF-RPC* also focus on this). LITE [46] is a representative work, which is a middle-ware in Linux kernel to provide easy-to-use interfaces and share resources safety.

7 Conclusion

This paper proposes a new RDMA-based communication (programming) paradigm named *RF-RPC*, which supports traditional RPC interfaces as well as providing high performance. The design of *RF-RPC* is based on the analysis current design choices for RDMA enabled networking. *RF-RPC* totally avoids server from involving network operations and make server CPU process the requests. The former can increase the network capability on server to serving more requests. And the later can avoid the problem of access amplification. Thus, *RF-RPC* is able to support traditional RPC interface based applications. By counter-intuitively making clients fetch results from server's memory remotely, *RF-RPC* makes good usage of server's in-bound RDMA and thus achieves higher performance. Experiments show 1.6$\times$ improvement of *RF-RPC* over *server-reply* and 4$\times$ improvement of *RF-RPC* over *server-bypass* respectively. With batching and optimization methods, *RF-RPC-batch* performs at most 11.3$\times$ throughput than *RF-RPC*. We also propose an automatic parameter tuning mechanism to make the trade-offs between higher throughput and lower latency. We believe *RF-RPC* can be integrated into many RPC-based systems to improve their performance without much effort.

Acknowledgments

This Work is supported by National Key Research & Development Program of China (2016YFB1000504), Natural Science Foundation of China (61433008, 61373145, 61572280, U1435216), China Postdoctoral Science Foundation (2018M630162).

References

[1] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu, "Rfp: When rpc is faster than server-bypass with rdma." in *EuroSys*, 2017, pp. 1–15.

[2] C. Mitchell, Y. Geng, and J. Li, "Using one-sided RDMA reads to build a fast, CPU-efficient key-value store." in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2013, pp. 103–114.

[3] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "Farm: Fast remote memory," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 401–414.

[4] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen, "Design and implementation of MPICH2 over InfiniBand with RDMA support," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2004, pp. 16–27.

[5] Y. Wang, X. Meng, L. Zhang, and J. Tan, "C-Hint: An effective and reliable cache management for RDMA-accelerated key-value stores," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. ACM, 2014, pp. 1–13.

[6] P. Stuedi, A. Trivedi, and B. Metzler, "Wimpy nodes with 10GbE: Leveraging one-sided operations in soft-RDMA to boost Memcached." in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012, pp. 347–353.

[7] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur *et al.*, "Memcached design on high performance RDMA capable interconnects," in *Proceedings of the International Conference on Parallel Processing (ICPP)*. IEEE, 2011, pp. 743–752.

[8] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou, "GraM: Scaling graph computation to the trillions," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. ACM, 2015, pp. 408–421.

[9] H. Li, A. Kadav, E. Kruus, and C. Ungureanu, "MALT: Distributed data-parallelism for existing ML applications," in *Proceedings of the European Conference on Computer Systems (EuroSys)*. ACM, 2015, pp. 3:1–3:16.

[10] (2016, July) InfiniBand in Data-Centers. [Online]. Available: http://www.mellanox.com/pdf/whitepapers/InfiniBand_EDS.pdf

[11] Mellanox. (2014, Oct.) Roce in the data center. [Online]. Available: http://www.mellanox.com/related-docs/whitepapers/roce_in_the_data_center.pdf

[12] ——. (2017, Jun.) Roce v2 considerations. [Online]. Available: https://community.mellanox.com/docs/DOC-1451

[13] ——. (2017, Feb.) Roce vs. iwarp competitive analysis. [Online]. Available: http://www.mellanox.com/pdf/whitepapers/WP_RoCE_vs_iWARP.pdf

[14] J. Wu, P. Wyckoff, and D. Panda, "PVFS over InfiniBand: Design and performance evaluation," in *Proceedings of the International Conference on Parallel Processing (ICPP)*. IEEE, 2003, pp. 125–132.

[15] N. S. Islam, M. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High performance RDMA-based design of HDFS over InfiniBand," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society Press, 2012, pp. 35:1–35:35.

[16] J. Huang, X. Ouyang, J. Jose, M. Wasi-ur Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda, "High-performance design of HBase with RDMA over InfiniBand," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2012, pp. 774–785.

[17] M. Wasi-ur Rahman, X. Lu, N. S. Islam, R. Rajachandrasekar, and D. K. Panda, "High-performance design of YARN MapReduce on modern HPC clusters with Lustre and RDMA," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2015, pp. 291–300.

[18] (2016, July) The memcached website. [Online]. Available: http://memcached.org/

[19] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2015, pp. 87–104.

[20] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li, "Balancing CPU and network in the cell distributed B-Tree store," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2016, pp. 451–464.

[21] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in *Proceedings of the European Conference on Computer Systems (EuroSys)*. ACM, 2014, pp. 27:1–27:14.

[22] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li, "Extracting more concurrency from distributed transactions," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 479–494.

[23] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout, "Implementing linearizability at large scale and low latency," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2015, pp. 71–86.

[24] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in RAMCloud," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2011, pp. 29–41.

[25] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 1984.

[26] (2016, July) The gRPC package. [Online]. Available: https://github.com/grpc/grpc

[27] A. K. M. Kaminsky and D. G. Andersen, "Design guidelines for high performance rdma systems," in *Proceedings of USENIX ATC16 2016 USENIX Annual Technical Conference*, 2016, p. 437.

[28] M. J. Koop, T. Jones, and D. K. Panda, "Reducing connection memory requirements of mpi for infiniband clusters: A message coalescing approach," in *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*. IEEE, 2007, pp. 495–504.

[29] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V. Y. Ye, "Building linkedin's real-time activity data pipeline." *IEEE Data Eng. Bull.*, vol. 35, no. 2, pp. 33–45, 2012.

[30] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li, "Fast and concurrent rdf queries with rdma-based distributed graph exploration," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, pp. 317–332.

[31] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an rdma-enabled distributed persistent memory file system," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 773–785.

[32] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 202–215.

[33] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 53–64.

[34] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over InfiniBand," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[35] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 429–444.

[36] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proceedings of the ACM Conference on SIGCOMM*. ACM, 2014, pp. 295–306.

[37] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek, "Cphash: A cache-partitioned hash table," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 319–320.

[38] (2016, July) The Mellanox website. [Online]. Available: http://www.mellanox.com/

[39] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling Memcache at Facebook." in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 385–398.

[40] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "LSM-trie: An LSM-tree-based ultra-large key-value store for small data items," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2015, pp. 71–82.

[41] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. ACM, 2010, pp. 143–154.

[42] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 185–201.

[43] M. Poke and T. Hoefler, "DARE: High-performance state machine replication on RDMA networks," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2015, pp. 107–118.

[44] "Fast and concurrent RDF queries using rdma-assisted GPU graph exploration," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/wang-siyuan

[45] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: distributed transactions with consistency, availability, and performance," in *Proceedings of the 25th symposium on operating systems principles*. ACM, 2015, pp. 54–70.

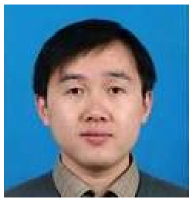
[46] S.-Y. Tsai and Y. Zhang, "Lite kernel rdma support for datacenter applications," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 306–324.

Yongwei Wu received the PhD degree in applied mathematics from the Chinese Academy of Sciences in 2002. He is currently a professor in computer science and technology at Tsinghua University of China. His research interests include parallel and distributed processing, and cloud storage. Dr. Wu has published over 80 research publications and has received two Best Paper Awards. He is an IEEE senior member.

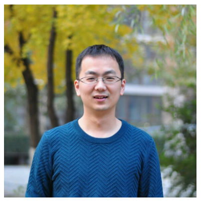
Teng Ma is a PhD student in Department of Computer Science and Technology, Tsinghua University, China. His research interests include distributed database, Remote Direct Memory Access (RDMA), and key-value store systems. He received his B.E. degree from University of Science and Technology Beijing, China, in 2016. He can be reached at: mt16@mails.tsinghua.edu.cn.

Maomeng Su received the BE degree from University of Science and Technology Beijing, China, in 2011, the PhD degree from Department of Computer Science and Technology at Tsinghua University, Beijing, China. He is currently working in Huawei Inc. His research interests include cloud storage systems, new datacenter networks, Remote Direct Memory Access (RDMA), and key-value store systems.

Mingxing Zhang received the PhD degree in computer science and technology from Tsinghua University, Beijing, China in 2017. His research interests include parallel and distributed systems. He received his B.E. degree from Beijing University of Posts and Telecommunications, China, in 2012. He can be reached at: zhangmx12@mails.tsinghua.edu.cn.



Kang Chen received the PhD degree in computer science and technology from Tsinghua University, Beijing, China in 2004. Currently, he is an Associate Professor of computer science and technology at Tsinghua University. His research interests include parallel computing, distributed processing, and cloud computing.



Zhenyu Guo received his master degree in Computer Science and Technology from Tsinghua Univeristy, Beijing, China. He worked in Microsoft Research Asia for ten years and is now in Ant Financial. His research interests include distributed systems, program analysis & verification, and machine learning recently. He published many papers in top conferences such as OSDI, PLDI, NSDI. He served in several programm commitees including OOPSLA and WWW.