RFP: When RPC is Faster than Server-Bypass with RDMA

Maomeng Su[1] Mingxing Zhang[1] Kang Chen[1] Zhenyu Guo[2] Yongwei Wu[1]
[1]*Tsinghua University* [*] [†] [2]*Microsoft Research* [‡]

Abstract

Remote Direct Memory Access (RDMA) has been widely deployed in modern data centers. However, existing usages of RDMA lead to a dilemma between performance and redesign cost. They either directly replace socket-based send/receive primitives with the corresponding RDMA counterpart (*server-reply*), which only achieves moderate performance improvement; or push performance further by using one-sided RDMA operations to totally bypass the server (*server-bypass*), at the cost of redesigning the software.

In this paper, we introduce two interesting observations about RDMA. First, RDMA has asymmetric performance characteristics, which can be used to improve *server-reply*'s performance. Second, the performance of *server-bypass* is not as good as expected in many cases, because more rounds of RDMA may be needed if the server is totally bypassed. We therefore introduce a new RDMA paradigm called Remote Fetching Paradigm (*RFP*). Although *RFP* requires users to set several parameters to achieve the best performance, it supports the legacy RPC interfaces and hence avoids the need of redesigning application-specific data structures. Moreover, with proper parameters, it can achieve even higher IOPS than that of the previous paradigms.

We have designed and implemented an in-memory key-value store based on *RFP* to evaluate its effectiveness. Experimental results show that *RFP* improves performance by 1.6×∼4× compared with both *server-reply* and *server-bypass* paradigms.

[*] Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China; Research Institute of Tsinghua University in Shenzhen, Guangdong 518057, China.

[†] M. Su is now with Huawei Inc.

[‡] Z. Guo is now with Ant Financial Inc.

EuroSys '17, April 23-26, 2017, Belgrade, Serbia
© 2017 ACM. ISBN 978-1-4503-4938-3/17/04…15.00
DOI: http://dx.doi.org/10.1145/3064176.3064189

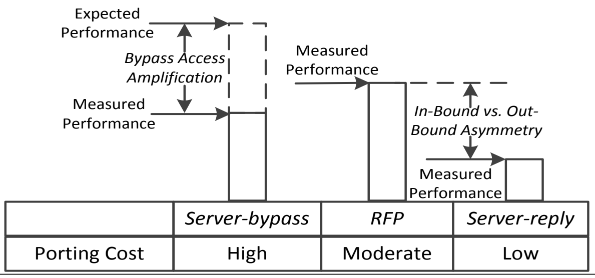

Figure 1. Improvements of *RFP* over *server-reply* and *server-bypass*. A higher bar means better performance.

1. Introduction

RDMA is a novel network technology that offers low-latency, high-bandwidth, and server-bypassing features, which has been widely deployed in modern data centers [4, 8, 10, 15, 19, 23, 29, 30, 34]. A common usage of RDMA is to replace original TCP/IP socket send/receive primitives with the corresponding RDMA counterpart, so that the same RPC interfaces can be implemented [7, 9, 10, 19, 31, 33, 34]. This way boosts applications' performance with minimal programming cost. For example, RDMA-Memcached [10] has applied this approach to Memcached [21], and it gains a performance improvement of 4×∼6× compared with that of using TCP/IP. We denote such design paradigm as *server-reply*, because it requires the server to reply results to clients.

However, the above approach does not really unleash all the power of RDMA. The iconic feature of RDMA is that it also provides **one-sided** operations, which totally bypasses CPU and OS kernel on remote machines. Another way of using RDMA is therefore to make clients access server's memory directly. Recent works such as Pilaf [23] and FaRM [4] have done great work towards this direction, and validated that server-bypassing can be faster than *server-reply*-based applications by as much as 100% [4, 23, 30, 32]. We denote this design paradigm as *server-bypass* [23].

While relieving server CPU from handling network packets can achieve better performance, it relies on developers to design specific data structures and algorithms, so that one-sided RDMA operation is feasible for the given problem. For instance, Pilaf [23] (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 kinds of applications, such as those with simple statistic operations [30].

To solve this dilemma, we propose a new RDMA-based RPC paradigm called Remote Fetching Paradigm (*RFP*). In *RFP*, 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, *RFP* achieves higher performance than both *server-reply* and *server-bypass*, which comes from the two observations illustrated in Figure 1.

The first is **in-bound vs. out-bound asymmetry**. Specifically, issuing a one-sided RDMA operation (i.e. out-bound RDMA) has much higher overhead than that of serving one (i.e. in-bound RDMA). This is because, taking *RDMA_Read* as an example, the issuing side needs to maintain certain context and involve both software as well as hardware to ensure the operation is sent and completed, while the serving side is purely handled by hardware. As a result, taking the RDMA Network Interface Card (RNIC) from InfiniBand we have in hand as an example, the peak IOPS (Input/output Operation Per Second) of in-bound RDMA (about 11.26 MOPS[1]) is about 5$\times$ higher than that of out-bound RDMA (about 2.11 MOPS). This explains why *server-reply* is suboptimal: besides not bypassing server's CPU, it also requires server to issue RDMA operations, which is bounded by the limit of out-bound RDMA. The latter is quickly saturated while the in-bound IOPS are far from being saturated by client requests. In the above example, there is only one *RDMA_Read* operation issued by the client and handled by the server. It is called to be out-bound RDMA on the issuing side and called to be in-bound RDMA on the other side.

The second observation is **bypass access amplification**, which leads to a significant gap between expected performance and measured performance of *server-bypass*. The former corresponds to the ideal case where only one RDMA operation is required to complete a request, which is usually not true in reality. The root cause is that CPU processing on server is bypassed and multiple clients need to coordinate their access to avoid data access conflict with more RDMA operation rounds! Moreover, they may also need additional RDMA operations for meta-data probing to find where the data is stored on server. Thus, the measured performance of *server-bypass* is typically much lower than its expected one. 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) [14, 17, 25].

Inspired by the two observations, *RFP* makes two important design decisions. First, to alleviate the constraint of in-bound vs. out-bound asymmetry, server buffers the results in its local memory instead of sending results back to clients through out-bound RDMA operations. Instead, clients use *RDMA_Read* to fetch these results remotely, so that the server only handles in-bound RDMA. This way leverages the in-bound RDMA performance of the server's RNIC by offloading the result-transferring responsibility from server to clients. Second, *RFP* 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-specific data structures.

There are two major challenges to implement this new paradigm. The first is when clients should fetch the results from server. A straw-man design is clients repeatedly polling the possible results on server, which ensures low latency, but at the cost of wasted remote RDMA-reads and high CPU consumption at clients' side, as well as wasted in-bound RDMA operations at server side. The second challenge is with what size to fetch the results from server as clients do not know the response size for RPC calls. Using an RDMA operation to get the size separately requires at least two remote fetches for each RPC call, lowering the performance unnecessarily [4, 23].

To address these challenges, *RFP* uses a hybrid mechanism that adaptively switches between repeated remote fetching and *server-reply*. A client uses *RFP* will at first continuously try to fetch the result from the server after it sends a request. But if it fails to fetch the result after a certain threshold of retries, it will switch to the traditional *server-reply* paradigm and wait for the server to send the result back. The threshold should be decided according the hardware configuration, so as to achieve a good trade-off between performance and client CPU consumption. Our mechanism automatically switches between continuous remote fetching and server-reply based on server load. Thus, RFP at least has the same performance with the *server-reply* paradigm when the server load becomes extremely high.

Moreover, we design an inline-based mechanism that requires clients to fetch a region of data and the size at once. By choosing an appropriate fetching size, we are able to maximize the chance of getting the whole result with one single RDMA operation. This mechanism can significantly reduce the average number of RDMA operations used for an RPC call, especially when the size of result is usually small. In *RFP*, the fetching size is adaptively set according to application characteristics and RNIC configurations, so that it is able to guarantee optimal performance for applications with different result size.

[1] MOPS: million operations per second.

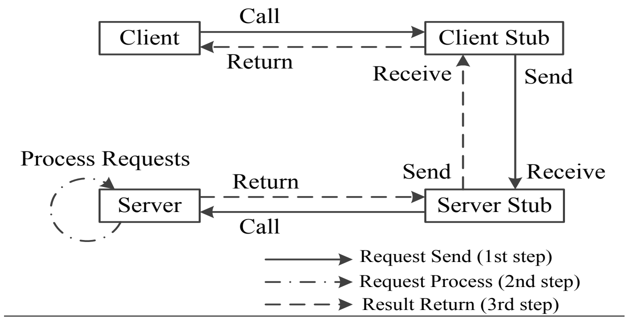

Figure 2. Overview of an RPC mechanism. The three steps can be optimized by RDMA with different design choices.

To calculate the optimum RDMA-read retrying number and fetching size, we model the problem into a parameter selection problem. Our solution leverages the hardware characteristics to get the lower bound and upper bound of these parameters, and connects them to application characteristics through pre-run and on-line sampling to approach the best performance results. Since using inappropriate parameters may offset the performance advantage of using *RFP*, these parameters should be carefully selected by the users, which is not required in previous works. However, as we will discuss in Section 3.2, the range of useful parameters is very limited so that the overhead of this selection procedure is acceptable.

The main contributions of this paper are as follows: (1) we report two key observations about RDMA and its usage paradigms; (2) we propose a new RDMA-based RPC paradigm called *RFP* that provides higher performance and incurs only moderate porting cost; (3) we design and implement an in-memory key-value store named *Jakiro* to validate the effectiveness of *RFP*. Experiment results show that *RFP* improves the throughput[2] by $1.6\times\sim4\times$ under different workloads, compared with *server-bypass* and *server-reply*.

The rest of this paper is organized as follows. Section 2 describes the two observations. Section 3 presents *RFP* in detail, as well as the solutions to address the challenges. We evaluate *RFP* in Section 4. Section 5 discusses the related works and we conclude in Section 6.

2. Design Choices and Observations

In this section, we discuss the design choices of implementing RPC using RDMA, followed by our two observations that lead to the key design decisions in *RFP*.

2.1 Design Choices for RDMA-Based RPC

As one of the most prevalently used communication mechanisms in distributed applications, RPC is well known for hiding the complexity of message-based communication for upper applications [14, 25, 27, 34]. Currently, 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 always consists of three steps, shown in Figure 2: (1) the *Request Send* step that client sends the function call identity as well as parameters to server; (2) the *Request Process* step within which the requests are processed to generate results on server; (3) the *Result Return* step where the results are transferred to client [2, 6].

[2] $Throughput$ in this paper means the number of requests completed per second.

	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
RFP	In-bound RDMA	Server involved	In-bound RDMA
Meaningless	In-bound RDMA	Server bypassed	Out-bound RDMA

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

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 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 In-Bound vs. Out-Bound Asymmetry

To quantify the asymmetry and study other properties such as scalability, we wrote some micro benchmarks and run them against a cluster of eight machines. Each machine is equipped with a Mellanox ConnectX-3 InfiniBand NIC (MT27500, 40 Gbps) [20] and dual 8-core CPUs (Intel Xeon E5-2640 v2, 2.0 GHz). Details of these machines are presented in Section 4. We choose one machine as server and others as clients. This is a typical client-server architecture. 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 7 clients. Each server thread randomly chooses a

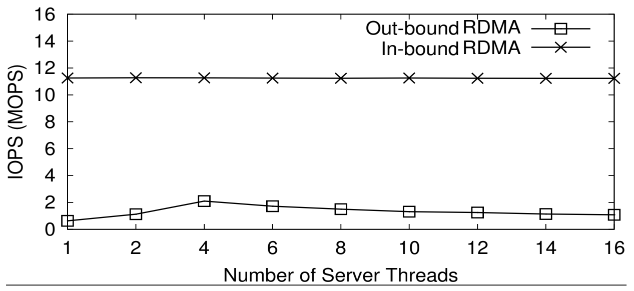

Figure 3. The IOPS of out-bound RDMA operations and in-bound RDMA operations with 32-byte data.

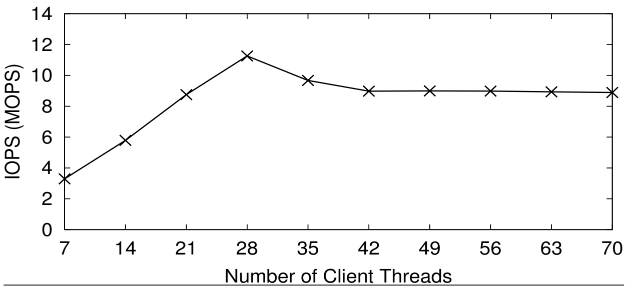

Figure 4. The IOPS of server's in-bound RDMA under different number of client threads.

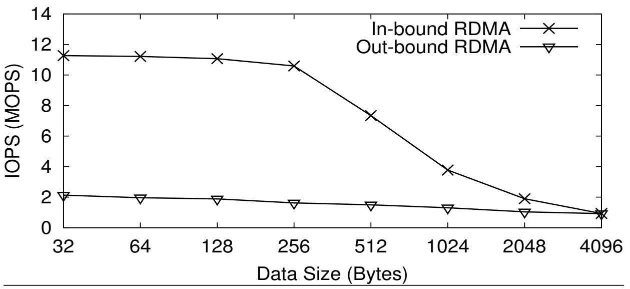

Figure 5. The IOPS of out-bound RDMA and in-bound RDMA under different size.

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 7 clients issue *RDMA_Read* operations to server. Memory buffers for client threads and server threads are independent and do not impact each other. For both tests, we launch four threads on each client machine to saturate the server's RNIC.

In order to simulate the common manner 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. As discussed by many existing works [11, 13], batching the requests or issuing several RDMA operations without waiting for the notifications of their completion can improve the performance. However, these optimizations are not always applicable and are out of this paper's topic.

Figure 3 shows the IOPS difference between out-bound RDMA operations and in-bound RDMA operations with 32-byte data. We see that the peak IOPS of in-bound (11.26 MOPS) is about 5× higher than that of out-bound (2.11 MOPS). We repeat this experiment with all the three kinds of RNICs we have (i.e., ConnectX-2, ConnectX-3, and ConnectX-4), and the results show that this asymmetry appears on all these different versions of hardware.

Our further discussion with developers in Mellanox explains why. For a one-sided RDMA operation, works on the side that receives/responds this operation (i.e., in-bound) are all handled by RNIC's hardware; while the other side that issues this operation (i.e., out-bound) requires interactions between hardware and software to ensure the operation is sent and completed. As a result, the workload on the receiving side (in-bound) is lighter and processed faster for one single data transfer through RDMA. As a circumstantial evidence of our speculation, the other two-sided RDMA operations, such as *RDMA_Send/Recv*, do not show asymmetry.

We also evaluate the IOPS under different client threads and different data size to study its scalability. The results show that server's in-bound RDMA IOPS decreases when the number of threads on each client has passed a certain threshold (see Figure 4). This is because clients experience some software contentions (caused by mutex) and hardware contentions (caused by using multiple queue-pairs (QP) and completion queues (CQ)) when issuing the RDMA operations, therefore the out-bound IOPS does not scale on clients, which decreases server's in-bound IOPS. Figure 3 shows that four server threads (each bounded to a dedicated CPU core) are enough to achieve peak out-bound performance at 2.11 MOPS. These phenomena further justify our argument that there are certain contentions between the competing client threads issuing RDMA operations, which limit the peak IOPS of out-bound RDMA. As shown in Figure 5, one can note that when the data size is larger than 2 KB, in-bound RDMA and out-bound RDMA perform the same in IOPS, 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.

The study shows 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 server.

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 [4, 23, 30, 32]. 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 as they may access the same

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*

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

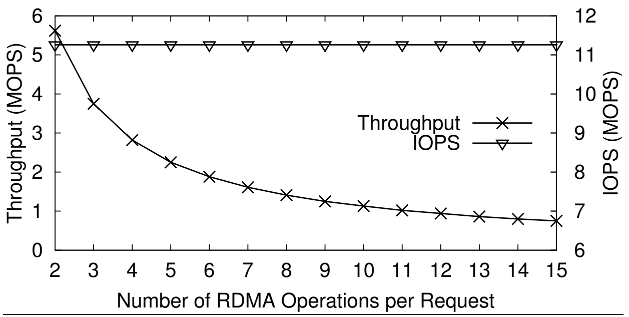

Figure 6. The throughput declines in *server-bypass* when more RDMA operations are needed to achieve a single client request.

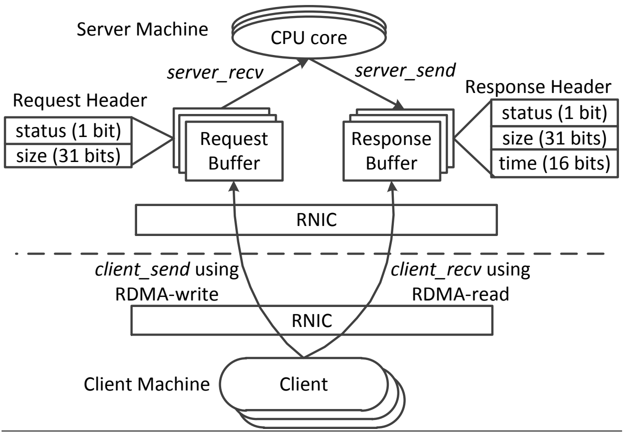

Figure 7. The overview of Remote Fetching Paradigm.

memory region which leads to data race! The support for legacy RPC applications is therefore bad, and the programming is also not easy (like all the other lock-free data structures and algorithms).

Moreover, *server-bypass* in many cases cannot achieve good performance as expected, when multiple RDMA operations are required to complete a 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 on metadata probing (find where the key-value pair is stored in server) and data transferring for completing a key-value GET [23]. 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 Figure 6[3], when conflicts are heavy with write-intensive workloads [14, 17, 25], 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.

3. *RFP*: Remote Fetching Paradigm

Based on the two observations above, this section present the design of Remote Fetching Paradigm (*RFP*), 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*. Two design decisions are made for *RFP* based on the above two observations. 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] Figure 6 is tested under 21 client threads connecting to the server.

3.1 Design Overview

As listed in Table 2, *RFP* 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 *RFP* by simply replacing the original TCP/IP socket interface with ours, which is straightforward [2, 6, 34]. As we will discuss later in Section 3.2, there are several parameters should be manually set before using *RFP*. But, since *RFP* 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.

Figure 7 illustrates how to use *RFP*. 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

```
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 RFP

```
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

Figure 8. How *RFP* and *server-bypass* implement GET for in-memory key-value stores at client side.

clients' responsibility to use *client_recv* to remotely fetch results from server's memory through in-bound *RDMA_Read*.

In summary, *RFP* combines the strength of the other two paradigms, while it also avoids their weakness. Firstly, *RFP* 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.

Figure 8 gives an example of using *RFP* and *server-bypass* to implement key-value store's GET operation. From Figure 8(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 Figure 8(a), *RFP* 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.

Second, *RFP* 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 memories used being registered into RNIC, *RFP* provides two APIs, *malloc_buf* and *free_buf* (see Table 2), to allocate and free buffers registered into RNIC automatically by *RFP*. 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 synchronizations. As shown in Figure 7, 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 *RFP* primitives, which will be discussed in the next section.

3.2 Challenges and Solutions

To maximize the throughput of applications, *RFP* 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 when the average number of retries is large. *RFP* 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, *RFP* uses repeated remote fetching to provide higher throughput and lower latency. Otherwise, repeated remote fetching brings little throughput improvement compared with *server-reply*, and *RFP* switches to *server-reply* to save CPU consumption

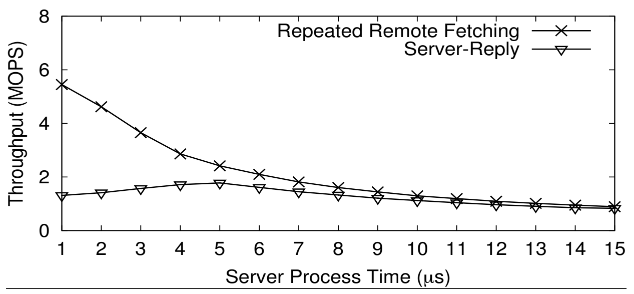

Figure 9. The throughput difference between repeated remote fetching and *server-reply* under different server process time (P). F and S are all 1 byte for this test.

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. *RFP* asks server to fill a result size of every RPC call in its response memory buffers for clients to fetch, as shown in Figure 7. 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 *RFP*, 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 *RFP* 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, Figure 9 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 Figure 9) 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, Figure 5 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. Thus, F must be in the second range [L,H]. L and H rely on hardware configuration, and can be gotten by running benchmark once (similar to Figure 5). 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, *RFP* uses an enumeration-based method to decide best R and F, in which the following equation is used for comparison:

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

Specifically, for each result of an application, *RFP* 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. RFP 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 $O((H - L)NM)$.

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 ⟨*client_id, RPC_id*⟩, 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. *RFP* records the response time for completing the request in the header of the response buffer (see Figure 7), 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, *RFP* only switches to *server-reply* if it finds a predefined number (e.g., two) of continuous RPC calls suffer from 5 failed retries of remote fetching. Otherwise, *RFP* remains in repeated remote fetching mode. According to the evaluation in Section 4.4.2, 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. Evaluation

Our evaluation answers the following questions:

- How much is the porting cost of using *RFP*?
- How much does *RFP* outperform *server-reply* and *server-bypass*?
- How does *RFP* perform under different workloads?

4.1 An *RFP*-Based Application

To evaluate how applications can easily and effectively use *RFP*, we implement *Jakiro*, an *RFP*-based in-memory key-value store. *Jakiro* contains two main modules: the communication protocol that uses *RFP* for RPC calls, and the in-memory key-value structure that keeps key-value pairs.

Communication Protocol. In *Jakiro*, the server exports RPC interfaces (i.e., PUT and GET) for clients to operate key-value pairs. The process follows strictly the traditional RPC practice: clients invoke RPC call stubs, which further uses *RFP* primitives to send the data to server. The server invokes RPC reply stubs, which stores the response in memory, waits for clients to fetch it, or transmits it to clients when *RFP* is switched to *server-reply* mode automatically. While the internal implementation is quite different, *Jakiro* uses *RFP* as it was a common RPC library.

In-Memory Key-Value Structure. We currently build *Jakiro* by caching key-value pairs in memory for the other applications (similar to Memcached [21]). 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. The whole structure is partitioned across different server threads in Exclusive Read Exclusive Write (EREW) [18]. Each server thread only accesses its own data partition. As proved in previous work of [11, 18, 22], such design is able to provide high performance for processing key-value pairs.

We implement *Jakiro* in about 3,000 lines of C++ code. The underlying libraries used for RDMA transferring in *RFP* are *rdmacm* and *ibverbs* provided by Mellanox OpenFabrics Enterprise Distribution [20]. In *Jakiro*, both the server and the client threads directly poll the memory buffers and the RNICs for message sending/receiving events. Meanwhile, each server thread does all the work of message packing/unpacking, sending/receiving, as well as request processing.

4.2 Experiment Setup

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

[4] Each slot is 8-byte so that a bucket fills in a cacheline.

connected by an 18-port Mellanox InfiniScale-IV switch. The machines run MLNX-OFED-LINUX-2.3-2.0.0 driver provided by Mellanox for Ubuntu 14.04 [20].

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: according to the analysis of previous works [1, 11, 26, 35], the value size of more than half of key-value pairs in Facebook's data center is around 20 bytes. Firstly, we present results on uniformly distributed and read-intensive workloads (95% GET) in Sections 4.4.1 and 4.4.2. Then we provide the experimental results on skewed workload, write-intensive workload, and workload with different value size respectively in Section 4.4.3. We use YCSB [3] 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 [23] that adopts *server-bypass* in Section 4.3. In Section 4.4, 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), which is developed by OSU [10]. 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.

4.3 Comparison with *Server-Bypass*

As mentioned before, *server-bypass* sometimes cannot achieve good performance as expected, due to bypass access amplification. Pilaf [23] 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

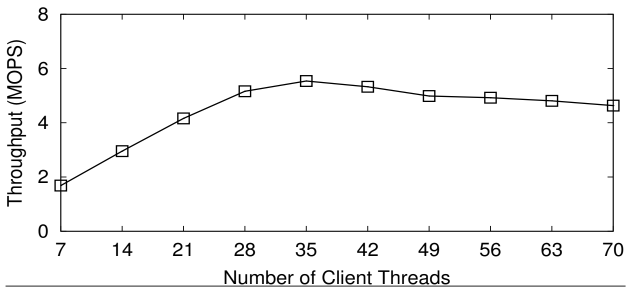

Figure 10. Throughput of *Jakiro* under different number of client threads. The server thread number is 6 and the value size is 32 bytes. The workload is uniform and read-intensive (95% GET).

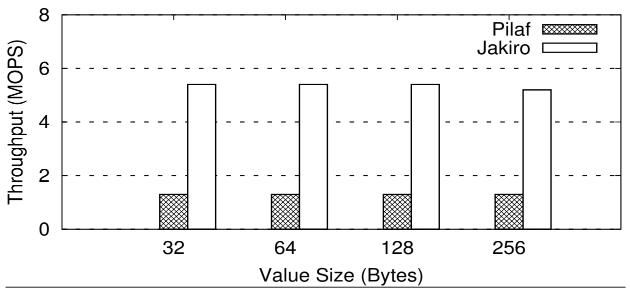

Figure 11. Peak throughput of *Jakiro* and Pilaf under uniform workload with 50% GET. The throughput of *Jakiro* is 4× as high as Pilaf's on value size from 32 to 256 bytes.

3.5 MOPS for uniform and read-intensive workload (95% GET). As displayed in Figure 10, the peak throughput of Jakiro that adopts *RFP* reaches 5.5 MOPS, which is 57.14% higher than Pilaf's[5].

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 Figure 3). A closer look reveals that thanks to the fetch-size decision mechanism in *RFP*, 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. When the number of client threads further increases, throughput of *Jakiro* decreases slightly (see Figure 10). This is because when each client machine launches more client threads, the out-bound IOPS of its RNIC reaches its upper limit. In this experiment, we tested two kinds of settings: *1)* one is fixing the number of client threads and hence each data point is measured separately; *2)* the other is dynamically adding and reducing the number of client threads. Results show that the measured throughput is not affected by

[5] 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.

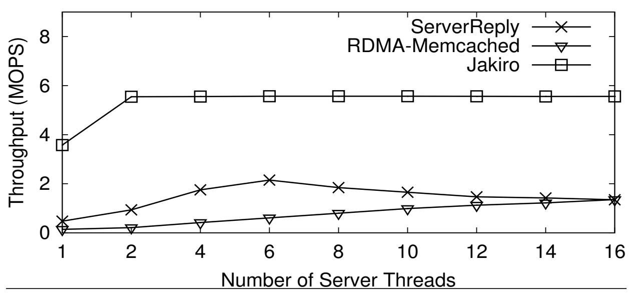

Figure 12. Throughput of *Jakiro*, ServerReply, and RDMA-Memcached on 32-byte value size. *Jakiro* outperforms ServerReply and RDMA-Memcached by about 160%∼310% in throughput.

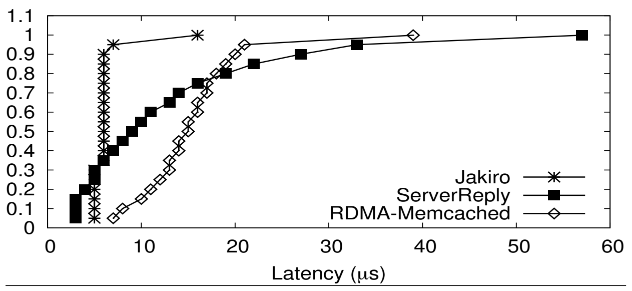

Figure 13. CDF of latency of *Jakiro*, ServerReply, and RDMA-Memcached on value size of 32 bytes.

the setting used, which show that the overhead of adding/reducing clients in Jakiro is minimal.

Moreover, a client in *server-bypass* needs more RDMA operations to complete a request when the conflicts are heavy. As shown in Figure 11, 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 [23]. 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.

4.4 Comparison with *Server-Reply*

In this section, we compare *RFP* with *server-reply* and show the performance improvement brought by *RFP*.

4.4.1 Comparison on Throughput

As shown in Figure 12, 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 Figure 12).

From Figure 12 we see RDMA-Memcached is bounded by the utilization of CPU rather than RNIC, and increasing the number of server threads improves its throughput. However, even 16 server threads of RDMA-Memcached still cannot saturate the RNIC's out-bound capacity, which brings lower peak throughput than ServerReply. This is because a server thread of RDMA-Memcached has to coordinate with other threads for sharing data structures (e.g, LRU lists) as well as to perform network operations, which does not exhibit good scalability [5, 16, 23]. By using data partition, a server thread in ServerReply does not need to interact with other threads, so ServerReply is only limited by out-bound RDMA operations. Moreover, as the server thread in RDMA-Memcached fully uses a CPU core, launching more than 16 server threads will only reduce the throughput as all cores are already saturated.

4.4.2 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%. Figure 13 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 also has been observed in HERD [11] and RDMA-PVFS [33]; (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%)

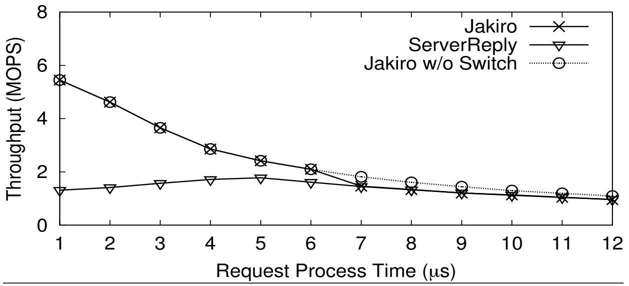

Figure 14. Throughput of *Jakiro* and ServerReply under different request process time at server side. *Jakiro* and ServerReply have comparable performance when the request process time becomes larger, as *RFP* automatically switches to *server-reply*.

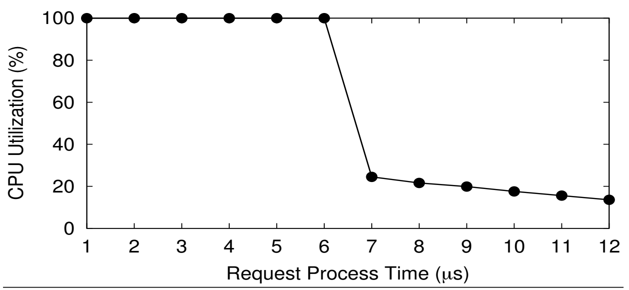

Figure 15. The client CPU utilization in *Jakiro* under different request process time at server side.

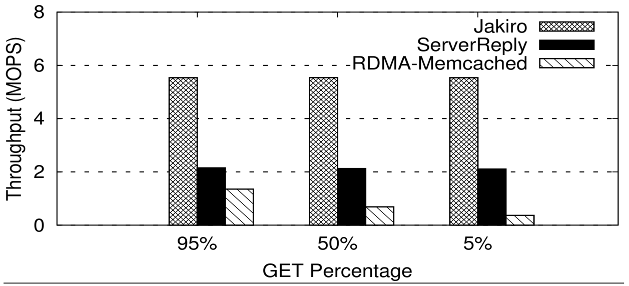

Figure 16. Comparison under varying GET percentages (32-byte value size). *Jakiro* achieves peak throughput at 5.5 MOPS under 5%, 50%, and 95% GET percentages, and significantly outperforms ServerReply and RDMA-Memcached.

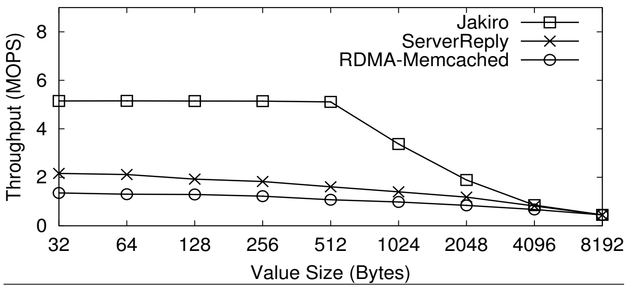

Figure 17. Comparison under varying value size. The workload is uniform with 95% GET.

and no two continuous calls suffer from 5 RDMA-read retries to fetch the result. Developers using *RFP* 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.

Figure 14 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 *RFP*. 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 in the RNIC. Under this circumstance, repeated remote fetching does not increase the performance and *RFP* 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 of them require more server resources at this time. Therefore, *RFP* achieves good performance for applications under different server loads.

In Figure 15, we show the corresponding client CPU utilization in *Jakiro* on different server loads. As we can see from the figure, the client CPU is always 100% used under the *RFP* mode, which is necessary for achieving the best latency. In contrast, if the request process time is long, *Jakiro* automatically switches to the *server-reply* mode for reducing clients' CPU utilization. Then the client CPU utilization drops from 100% to below 30%.

4.4.3 Comparison under Different Workloads

We also compare the three systems under different types of workloads (varying GET percentage, varying value size, and skewed workload). The experiment for the three systems is run at the configurations in which each of them achieves peak throughput on value size of 32 bytes in uniform and read-intensive (95% GET) workload, as presented in Section 4.4.1. For both *Jakiro* and ServerReply, the configuration is 6 server threads connecting to 35 client threads. For RDMA-Memcached, the configuration is 16 server threads connecting to 35 client threads.

Varying GET Percentage. Figure 16 illustrates the throughput on 32-byte value size of *Jakiro*, ServerReply, and RDMA-Memcached under different GET percentages with uniform workload. *Jakiro* still fully uses the RNIC's in-bound IOPS to obtain peak throughput at 5.5 MOPS under varying GET percentages. As the server threads in *Jakiro* are (mostly) not performing networking operations, their computing capacity is enough to process RPC calls whether they are GET or PUT. That is why the peak throughput of *Jakiro* reaches 5.5 MOPS even when the workload is write-intensive (95% PUT). Such throughput is still about 160% higher than that of ServerReply, which saturates server RNIC's out-bound RDMA-write performance (2.1 MOPS) under GET percentages from 95% to 5%. However, as shown in Figure 16, RDMA-Memcached is limited by the CPU utilization[6], and has decreased throughput when the workloads become write-intensive. In workload with 95% PUT, *Jakiro* improves the throughput by 14× compared with that of RDMA-Memcached.

Varying Value Size. Figure 17 presents the throughput of *Jakiro*, ServerReply, and RDMA-Memcached on value size from 32 bytes to 8,192 bytes. Through pre-running such workload, *RFP* sets the fetching size as 640 bytes for *Jakiro*, keeping R as 5. As seen in this figure, *Jakiro* significantly outperforms ServerReply and RDMA-Memcached by 60%∼280% on value size from 32 bytes to 2048 bytes. For ServerReply, when the value size becomes larger, the server thread has to spend more time in sending the result through out-bound RDMA-write for a GET RPC call. Therefore, its throughput decreases due to wasting more CPU cycles on networking operations. Increasing the number of server threads helps mitigate the issue for ServerReply on larger value size. For example, when the server thread number is increased to 12, ServerReply achieves about 2.1 MOPS for key-value pairs with value size of 512 bytes. However, as shown in Figure 12, it is at the expense of reducing throughput on smaller value size (e.g., 32 bytes or 64 bytes). When the value size grows to 4096 bytes, the throughput of *Jakiro*, ServerReply, and RDMA-Memcached are comparable. This is because both ServerReply and RDMA-Memcached saturate the network's bandwidth at this time.

The size of value is fixed in the above experiments. To evaluate the performance with variable value size, we also test the three systems under a read-intensive workload in which the value size of key-value pairs uniformly distribute between 32 bytes and 8,192 bytes. In this case, *Jakiro* achieves 3.58 MOPS, about 140% and 251% higher than that of ServerReply (1.49 MPOS) and RDMA-Memcached (1.02 MOPS), respectively.

Figure 18 shows the throughput of *Jakiro* under different fetching size. We observe that 640-byte can achieve a good throughput for a wide range of value size (from 32 bytes to 640 bytes), even though the throughput for smaller value size decreases slightly compared with smaller fetching size (e.g., 256 or 512 bytes). Further increasing the fetching size

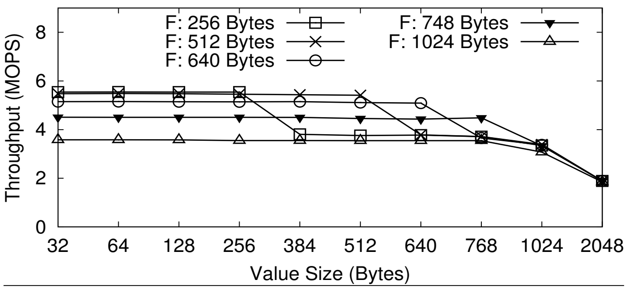

Figure 18. Throughput of *Jakiro* under different fetching size (F). The workload is uniform with 95% GET. *RFP* calculates 640 bytes as the optimum F (keeping R as 5), which helps Jakiro achieve the best throughput for the given workload.

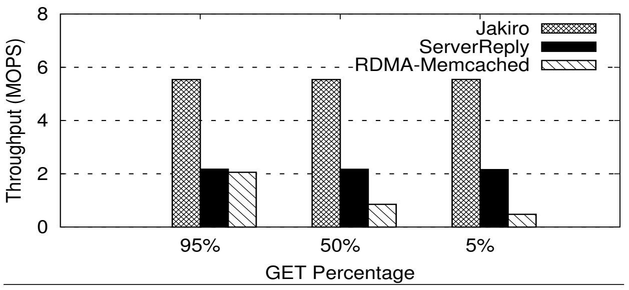

Figure 19. Comparison under skewed workload on 32-byte value size. *Jakiro* still outperforms ServerReply and RDMA-Memcached under skewed workload.

(e.g., 748 bytes) just lowers the performance of *Jakiro* in average due to network resource wasting. Using 1024-byte fetching size produces the lowest performance for *Jakiro*, as the bandwidth becomes the bottleneck.

Skewed Workload. We test how *Jakiro* perform in a skewed workload. The keys in the workload are generated according to a Zipf distribution with parameter .99. Figure 19 presents the throughput of *Jakiro*, ServerReply, and RDMA-Memcached on value size of 32 bytes under different GET percentages. Although the most popular key is about 10^5 times more often than the average key in the skewed workload, the load of the most loaded server thread is <25% more than that of the thread with the least load [11], in the case of launching six server threads. Even under skewed workload, the server threads in *Jakiro* are able to process key-value requests when the server's RNIC is saturated. Thus, the peak throughput of *Jakiro* is still 5.5 MOPS under 5%, 50%, and 95% GET percentages. According to Figure 20, *Jakiro* also performs best in average latency (we use the read-intensive workload as an example). ServerReply is still limited by the RNIC's out-bound RDMA-write, and obtains 2.1 MOPS for the skewed workload. As mentioned in Section 4.4.1, the RDMA-Memcached is bounded by the CPU at the server side. Under skewed workload, RDMA-Memcached benefits from serving the popular keys as this

[6] The CPU utilization consists of computing, accessing memory, and performing network operations.

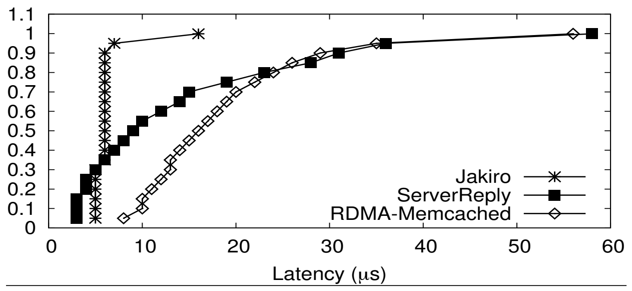

Figure 20. CDF of latency of *Jakiro*, ServerReply, and RDMA-Memcached on read-intensive skewed workload.

makes use of cache locality [30]. As a result, the throughput of RDMA-Memcached in this case is higher than that under uniform workload. For example, as shown in Figure 19, RDMA-Memcached achieves about 2.1 MOPS on value size of 32 bytes with 95% GET, which saturates the RNIC's capacity for out-bound RDMA. Overall, *Jakiro* beats both ServerReply and RDMA-Memcached under workload with skewed distribution.

Number of Retries in *Jakiro*. At last we show the number of retries in *Jakiro* under different workloads in Table 3. We see that the percentage of requests that need 2 retries to fetch results to clients is around 0.1% in all four workloads. In some extreme case, the number of retries can occasionally be as large as 9. But, according to our evaluation, this kind of occasional case never repeatedly appears. As a result, there will not be an unnecessary switch between *RFP* and *server-reply* (as discussed in Section 3.2).

4.5 Applicability and Limitations

Since *Jakiro* will automatically switch to *server-reply* when necessary, it can achieve good performance on various types of workloads. In contrast, as we have shown in prior evaluation results, *RFP* itself is only beneficial for a certain range of workloads.

First, as illustrated in Figure 5, there is no asymmetry between in-bound and out-bound RDMA if the size of each packet is larger than 2 KB, because bandwidth rather than IOPS becomes the bottleneck. As a result, *RFP* can only improve IOPS for small packets. Second, if the process time of a request is longer than 7 μs, using *RFP* does not improve throughput but increases client CPU utilization (as shown in Figure 14). However, as reported by recent investigations [1, 11, 26, 35], most applications in modern data centers transfer small packets. In this case, *RFP* is better than existing methods for applications that *1)* have an extremely high throughput (Figure 11 and Figure 12); and *2)* are latency-sensitive (Figure 13 and Figure 20).

Moreover, since it is impossible to make both clients and servers handle only in-bound RDMA operations, our choice in *RFP* is "servers handle only in-bound RDMA while clients still use out-bound RDMA". This setting takes advantage from the asymmetry and hence can achieve a better aggregated throughput if the number of clients is higher than the number of servers.

	Uniform		Skewed	
	95% GET	5% GET	95% GET	5% GET
Percentage of $N > 1$	0.105%	0.13%	0.09%	0.09%
The largest N	6	5	9	4

Table 3. Number of retries (N) in *Jakiro* under different workloads (using 32-byte value).

5. Related Work

Different Queue Pair Types. There are three kinds of queue pair types that can be used to support RDMA. *RFP*, 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 [11] and FaSST [12], 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 [12], 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 13 and 20 also demonstrate that *Jakiro* achieves better latency so that *RFP* is more suitable for latency-sensitive applications. Anuj et al. [13] 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 *RFP*'s performance.

Different Paradigms. Other existing RDMA-based solutions usually apply *server-reply* [7, 9, 10, 28, 31, 33, 34], *server-bypass* [4, 30, 32], or a combination of these two paradigms [23, 24]. As we have demonstrated in Section 4, *RFP* 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$\sim$6) are usually needed to complete a request [23], which number can be even higher when conflicts from multiple clients are heavy.

RFP also involve 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 [23, 30]. DrTM relies on explicit locks and high-performance HTM for data race coordination [32].

FaRM [4] 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 Jakiro. Nevertheless, in order to achieve this performance, FaRM uses Hopscotch hashing that leads to something like "batching the requests". 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 Jakiro; 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.

Moreover, it has to be mentioned that to Pilaf, C-Hint, and FaRM, all of them use *server-reply* to serve PUT requests. In this case, these systems suffer from the limited performance of server's out-bound RDMA.

6. Conclusion

This paper proposes a new RDMA-based paradigm named *RFP*, which supports traditional RPC interfaces as well as providing high performance. The design of *RFP* is based on two observations: the first is performance asymmetry in RNIC and the second is performance degradation due to conflicts resolving in *server-bypass*. By making server involved in request processing, *RFP* is able to support traditional RPC interface based applications. By counter-intuitively making clients fetch results from server's memory remotely, *RFP* makes good usage of server's in-bound RDMA and thus achieves higher performance. Experiments show 1.6$\times\sim$4$\times$ improvement of *RFP* over *server-reply* and *server-bypass*. We believe *RFP* can be integrated into many RPC-based systems to improve their performance without much effort.

Acknowledgments

We thank our shepherd Prof. Marco Canini and the anonymous reviewers for their valuable feedback. We also want to take this opportunity to thank Changqing Li, Gaofeng Feng, and other developers from Mellanox Technologies that kindly answered our questions and discussed with us. This work is supported by National Key Research & Development Program of China (2016YFB1000504), National Basic Research (973) Program of China (2014CB340402), Natural Science Foundation of China (61433008, 61373145, 61572280, 61133004, 61502019, U1435216). Contact: Kang Chen (chenkang@tsinghua. edu.cn) and Yongwei Wu (wuyw@tsinghua.edu.cn).

References

[1] 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*, volume 40, pages 53–64. ACM, 2012.

[2] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[3] 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)*, pages 143–154. ACM, 2010.

[4] 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)*, pages 401–414, 2014.

[5] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent Memcache with dumber caching and smarter hashing. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 371–384, 2013.

[6] gRPC. https://github.com/grpc/grpc.

[7] 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)*, pages 774–785. IEEE, 2012.

[8] InfiniBand in Data-Centers. http://www.mellanox.com/pdf/whitepapers/InfiniBand_EDS.pdf.

[9] 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)*, pages 35:1–35:35. IEEE Computer Society Press, 2012.

[10] 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)*, pages 743–752. IEEE, 2011.

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

[12] 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)*, pages 185–201, 2016.

[13] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *Proceedings of*

the USENIX Annual Technical Conference (ATC), pages 437–450, 2016.

[14] 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)*, pages 71–86. ACM, 2015.

[15] 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)*, pages 3:1–3:16. ACM, 2015.

[16] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 476–488. ACM, 2015.

[17] 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)*, pages 27:1–27:14. ACM, 2014.

[18] 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)*, pages 429–444, 2014.

[19] 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)*, pages 16–27. IEEE, 2004.

[20] Mellanox. http://www.mellanox.com/.

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

[22] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. Cphash: A cache-partitioned hash table. In *ACM SIGPLAN Notices*, volume 47, pages 319–320. ACM, 2012.

[23] 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)*, pages 103–114, 2013.

[24] 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)*, pages 451–464, 2016.

[25] 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)*, pages 479–494, 2014.

[26] 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)*, pages 385–398, 2013.

[27] 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)*, pages 29–41. ACM, 2011.

[28] 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)*, pages 107–118, 2015.

[29] 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)*, pages 347–353, 2012.

[30] 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)*, pages 1–13. ACM, 2014.

[31] 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)*, pages 291–300. IEEE, 2015.

[32] 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)*, pages 87–104. ACM, 2015.

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

[34] 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)*, pages 408–421. ACM, 2015.

[35] 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)*, pages 71–82, 2015.