RR-Compound: RDMA-Fused gRPC for Low Latency, High Throughput, and Easy Interface

Liang Geng, Hao Wang, *Member, IEEE*, Jingsong Meng, Dayi Fan, Sami Ben-Romdhane, Hari Kadayam Pichumani, Vinay Phegade, and Xiaodong Zhang, *Fellow, IEEE*

***Abstract*—Advanced data centers strive for high performance and throughput, which can be achieved through the desirable merits of Remote Procedure Call (RPC) programming model and the low latency of Remote Direct Memory Access (RDMA). However, despite the widespread availability of these software and hardware utilities, they have been utilized separately for their own applications in existing production systems for many years. Although researchers have attempted to develop RDMA-enabled RPC prototypes, they often face challenges such as API discrepancies and a lack of specific features for effective integration with major production software, rendering them incompatible. This industry R&D project aims to enhance the performance of gRPC, a widely utilized RPC framework in major companies, by integrating RDMA as an internal component. Our system solution, called RR-Compound, combines the simple user interface and other merits of gRPC with low latency for remote data accesses. RR-Compound is fully compatible with gRPC and can serve as a seamless replacement without altering existing applications. However, to achieve low latency, high throughput, and scalability for RR-Compound, several technical challenges in managing network connections and memory space utilization must be effectively addressed. To overcome the limitations of existing connection methods, we have developed a new method called BPEV that is independent of gRPC and applicable to all RDMA systems. We have also retained the asynchronous framework of gRPC, albeit with limited buffer space in RDMA memory management. In micro-benchmarks, RR-Compound outperforms mRPC - the state-of-the-art RPC framework for a large number of connections, achieving a 14.77% increase in throughput and a 42.55% reduction in latency. Subsequently, we compare RR-Compound with gRPC over IPoIB using two real-world applications: KV-Store and TensorFlow. RR-Compound achieves up to a 2.35x increase in throughput and reduces the average latency by 46.92%.**

***Index Terms*—RDMA, RPC, networking.**

I. Introduction

As THE advancement of networking technologies, such as Terabit Ethernet (TbE) [1], the bandwidth of moving data remotely in a large-scale data center can be as high as the internal bandwidth inside a single node. Consequently, modern data centers serving various applications have been evolved to a powerful platform of in-memory computing to harvest a huge capacity of remote memory resources [2], [3], [4], and being able to scale-up by upgrading each node and scale-out by increasing the number of nodes. Specifically, the following four requirements are expected for an advanced data center. *(1) An easy programming interface*. To keep the service scope as large as possible, the application program development in different areas should be independent of the rapid technical advancement of data center architecture and systems. In this way, application users do not need to modify their programs to respond the infrastructure changes in both software and hardware of data centers. *(2) Low-latency memory accesses*. This requires that frequent data accesses to remote memory modules be fast. *(3) High scalability*. A data center must be prepared to serve increasingly more users by physically connecting more computing nodes, and by timely upgrading each computing node, achieving sustainable performance improvement. *(4) High-throughput for data processing tasks*. This is a major performance factor for data-intensive applications in data centers, which counts the number of operations per time unit, such as the number of Key-Value stores per second. Continued improvement of throughput for various applications is a critical factor for a high productivity and high efficiency of data centers. To meet all the above four requirements is challenging due to several conflicting design goals of system components under the existing framework of data centers. So far, none of running systems meet all the requirements. We will explain the rationale behind this fact by linking the two major protocols of RPC [5] and RDMA to each of the four requirements.

Modern data centers consist of many micro-services [6], [7], [8], [9], which are supported by RPC [5] (remote procedure calls). The RPC abstraction hides communication details and makes remote procedure calls just like local ones. Thus, its original design well meets the requirement of "an easy programming interface". Open source software gRPC initially developed by Google [10], has been widely used in many production systems. Its simple service definition using Protocol Buffers [11] with a variety of programming languages provides a general-purpose interface to users for their application development being independent of the data center infrastructure. According to a Google data center report, there are over 10 billion RPCs per second [12]. However, gRPC relies on a TCP-based transport layer; thus,

Manuscript received 28 September 2023; revised 8 May 2024; accepted 18 May 2024. Date of publication 23 May 2024; date of current version 1 July 2024. The work was supported in part by the U.S. National Science Foundation under Grant MRI-2018627, Grant CCF-2005884, Grant CCF-2210753, Grant CCF-2312507, and Grant OAC-2310510. Recommended for acceptance by K. Gopalan. *(Corresponding author: Hao Wang.)*

Liang Geng, Jingsong Meng, Dayi Fan, and Xiaodong Zhang are with the The Ohio State University, Columbus, OH 43210 USA (e-mail: geng.161@osu.edu; meng.479@osu.edu; fan.1090@osu.edu; zhang@cse.ohio-state.edu).

Hao Wang is with the International Digital Economy Academy (IDEA), Shenzhen 100191, China (e-mail: wanghao2020@idea.edu.cn).

Sami Ben-Romdhane, Hari Kadayam Pichumani, and Vinay Phegade are with eBay, Inc., San Jose, CA 95125 USA (e-mail: sbenromdhane@ebay.com; hkadayam@ebay.com; vphegade@ebay.com).

Digital Object Identifier 10.1109/TPDS.2024.3404394

it has a well-known issue of high-latency memory accesses to remote nodes over TCP due to memory copies between user space and kernel space, and due to high overhead caused by frequent OS kernel interrupts.

In contrast, the widely available Remote Direct Memory Access (RDMA) protocol and its Network Interface Cards (NICs) allow remote memory accesses without involving OS kernels either in senders or receivers. The low latency in RDMA is achieved by managing data operations in user space, which eliminates the data copies between user and kernel space, and the overhead of interrupting CPU. Thus, it well meets the requirement of "low-latency memory accesses". We have compared the latency of data accesses between TCP and RDMA, by changing data sizes from 16 Bytes to 512 KBytes in the standard network benchmarks of *sockperf* for TCP and *perftest* for RDMA on a 100 Gbps Ethernet cluster. Our experiments show that RDMA is 8.6x - 15x faster than TCP for memory-to-memory data communication. However, the RDMA abstraction to users is narrowly focused on data exchange, which is not "an easy programming interface", seriously limiting the scope of users. In addition, the latency comparisons in our experiments are done in an isolated environment, which only reflects a single system factor. The comprehensive performance improvement of a system comes from a deliberate balancing consideration among multiple factors, such as the four requirements, and should be evaluated by representative application workloads.

Regarding the requirement of scalability, we focus on network connection management in this paper. The remote connection in gRPC and other advanced RPC systems, such as mRPC [13], is either handled by OS with a polling engine (typically implemented by a system call of epoll in Linux) or busy-polling in the user space to detect incoming messages, which can efficiently handle a large volume of network connections. This partially meets the requirement of "high scalability" because gRPC is efficient for many network connections with small data volumes. In contrast, the connection management in RDMA is supported by two options: (1) a simple user mechanism of "busy-polling", and (2) an event-based method by soft interrupts. However, our experiments show that neither option is scalable as we increase the number of connections under different connection frequencies. It is desirable to have a single connection management mechanism that shares the merits of both busy-polling and event-based method.

Regarding the requirement of throughput, high throughput for remote data-intensive applications is not guaranteed over TCP in gRPC. While, RDMA is designed for applications requiring high throughput. It meets the requirement of "high throughput of data processing tasks" under the condition of having an effective network connection management method. In practice, the low latency of RDMA is also beneficial to the improvement of throughput.

Table I summarizes the positions of gRPC and RDMA under the four requirements, showing that gRPC and RDMA can be complementary to each other if they are well integrated into one system. Unfortunately, from our industry perspective, they have been run separately for their own applications on different platforms for many years. We present such an integrated system in this paper by fusing RDMA into gRPC. The system is called RR-Compound, aiming to provide two unique advantages. First, the user scope is as large as that of gRPC, serving as a general-purpose system and enlarging its scope to data-intensive applications. Second, RR-Compound retains the merits of both gRPC and RDMA, achieving the goals for low latency, high throughput, and high scalability.

TABLE I
INDEPENDENT POSITIONS OF GRPC AND RDMA UNDER THE FOUR EXPECTED REQUIREMENTS OF ADVANCED DATA CENTERS

	gRPC	**RDMA**
General-purpose programming interface	Yes	No
Latency	High	Low
Connection Management	Epoll	Busy-polling or RDMA-Event
Throughput	High for infrequent data communication applications	High for frequent data communication applications

In order to effectively develop a "compound" based on both gRPC and RDMA (represented by an acronym of RR) for the above-mentioned two advantages, we have addressed the following technical challenges. First, gRPC uses two-sided socket APIs to transfer data and RDMA provides two-sided semantic APIs too. However, the one-sided RDMA APIs are more efficient [14], [15]. How to fuse the one-sided RDMA into gRPC is non-trivial, but it is a critical task in this project. Second, the mechanisms of detecting incoming messages are fundamentally different between gRPC and RDMA: gRPC relies on system calls, e.g., "epoll", to detect I/O events while RDMA only provides limited APIs for event-based connection management, e.g., *ibv_get_cq_event*. RDMA also supports "busy-polling" [15]. How to develop and embed a single RDMA message-detecting mechanism in the gRPC system without losing functionality and generality is challenging. The merits of event-based method and busy-polling must be considered for system scalability. Finally, the memory management of RDMA does not fit in the asynchronous framework of gRPC. gRPC uses two sets of buffers, i.e., double buffering, to achieve asynchronous processing, while RDMA only has one set of buffers. The challenge is to maintain the asynchronous framework after fusing RDMA into gRPC. We have made the following contributions in this paper.

- We have integrated one-sided RDMA into gRPC, keeping the general-purpose user interface with a transport layer of RDMA.
- Aiming for high scalability, we have designed and implemented a new and effective network connection management scheme to detect incoming messages, which is gRPC independent and general-purpose for any RDMA systems. We also show that network connection management plays a critical role for the scalabilty of RDMA and RPC related systems.
- We have maintained the asynchronous processing framework of gRPC by a limited RDMA buffers in user space, aiming to keep the merits of both.

- We have developed a separate module to support RDMA-fused gRPC, so the TCP based gRPC is still available. This design allows users to switch flexibly based on different workloads if necessary.
- We have conducted extensive and comparative experiments on three platforms, namely RR-Compound, gRPC and mRPC, verifying the effectiveness of our systems design and implementation.

Besides micro-benchmarks, we evaluate RR-Compound on two representative workloads: YCSB [16] and TensorFlow [17]. We ran YCSB on Jungle database [18] (an embedded key-value storage library developed by eBay) to understand performance insights. A high percentage of execution time in YCSB workloads is spent on data communications, where RDMA can gain high performance. Our experiments show that RR-Compound achieves up to 2.35x improvement in throughput for YCSB over that on gRPC. On the other hand, the TensorFlow workloads are local computing-intensive, and only a small percentage of time is spent on data communications. Even under this condition, RR-Compound still achieves up to 30% throughput improvement over that on gRPC.

II. Background

In this section, we will briefly introduce the merits and limits of RDMA and gRPC. We then present the technical challenges for fusing RDMA into gRPC.

A. RDMA

Remote Direct Memory Access (RDMA) is a communication protocol for applications to directly read/write data across a network from a specified memory region in one node to another specified memory region in another node. RDMA users will register the memory region and set the desired memory access attributes of the memory. The attributes tell the RDMA device how the memory region can be accessed from local or remote nodes. After the registration, the RDMA client and server will exchange the data stored in the memory regions to each other, so that the client and server can access the memory regions of each other via specialized RDMA Network Interface Cards (NICs). As shown by the dotted line in Fig. 1(a), a client node can directly move data from its send buffer in a specified memory region to the receive buffer in a specified memory region in the server node without the involvement of the kernel in both sides. The specified memory region comes from a "pin" by users, which tells the kernel that this memory region is owned by an application in the user space. In contrast, the solid line in Fig. 1(a) shows that the same data transmission by TCP requires data movement of copying buffer content from user space to kernel space along with context switches of OS. The shortcut in the data transmission path in RDMA is the reason for the latency gap between the two protocols.

The RDMA programming is provided by the APIs called "verbs" [19]. The communication primitives of RDMA are categorized into two classes: one-sided verbs and two-sided verbs. As shown in Fig. 1(b) top side, one-sided verbs send

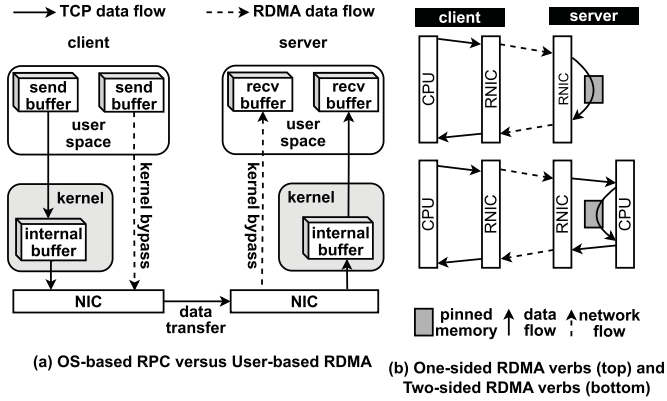


Fig. 1. Data flow of TCP and RDMA.

messages to remote memory without getting the remote CPU involved; whereas, two-sided verbs require the CPU involvement in both sides, as shown in Fig. 1(b) bottom side. We mainly focus on two RDMA primitives: *RDMA_WRITE* (one-sided verb) and *RDMA_WRITE_WITH_IMM* (two-sided verb). In *RDMA_WRITE*, the receiver keeps polling its completion queue until it gets the result. We call this method as RDMA busy-polling method (RDMA-BP). In *RDMA_WRITE_WITH_IMM*, the receiver uses a completion event channel, which is supported by the OS kernel, to detect incoming messages. The completion event channel is a file descriptor (FD) that is used to deliver completion notifications to the user space process. When a completion event is generated in a completion queue, the event is delivered via the completion event channel. The RDMA library provides a function call, *ibv_get_cq_event()*, to wait for the next completion event. During the waiting period, CPU cycles can be used for other tasks. We call this method RDMA-Event.

B. gRPC

Remote Procedure Call (RPC) [5] is a request-response protocol. It allows programmers to invoke a remote function just like to call it locally without knowing the communication details, the underlying operating system, and even the programming language used to implement the remote function. The RPC system typically consists of a client and its server. A client initiates an RPC request with arguments of a function to its server over network. The server processes this request and sends a response back to the client. The RPC protocol hides the internal message-passing mechanism from users. Its high-level abstraction makes programmers productively develop many applications in distributed systems.

gRPC is a modern and open-source library and framework, which is one of the most widely used RPCs in production systems. Two major components of gRPC are network connection management and memory management. In connection management, gRPC relies on its polling engine for connection management, where events like new client connections and incoming data are monitored on a set of file descriptors. gRPC does not spontaneously generate threads; instead, it depends on

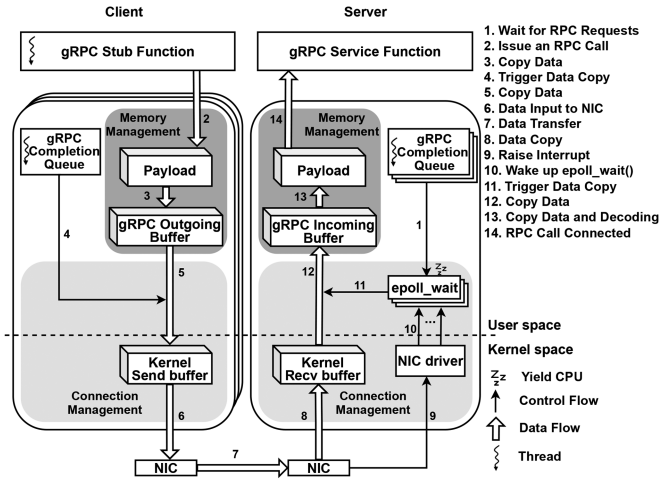


Fig. 2. Asynchronous RPC call in gRPC.

user-created threads to drive the polling engine, enabling the detection of I/O events and facilitating data exchange with peers.

The gRPC memory management and *CompletionQueue* design enables synchronous and asynchronous RPC calls. Fig. 2 shows the data flow of an asynchronous RPC call in gRPC, where the connection and memory management are presented. Before a client starts an RPC call, its server prepares to process the RPC by calling the I/O multiplexing function *epoll_wait()*[1] to wait for incoming data (step 1). If there is no incoming data, the thread will yield its CPU time and wait to be woken up. Since the server is ready to accept requests, the client can issue an RPC call at any time by placing related parameters and data in the Payload buffer to be encoded according to the binary wire format of Protocol Buffer language (gRPC uses Protocol Buffer to serialize and de-serialize RPCs) (step 2). The encoded data is copied from the Payload buffer to the gRPC Outgoing Buffer (step 3). At this moment, the data has not been sent out, since it is an asynchronous RPC call. Only if the client calls the corresponding function in *CompletionQueue* (step 4), gRPC will be notified to copy the data to the Kernel Send Buffer by TCP, which is initiated by OS (step 5). The kernel buffer data will be sent to its NIC (step 6), and the RPC data will be transferred from the client NIC to the server NIC (step 7). On the server side, when the message reaches its NIC, it will be moved to the kernel receive buffer first by OS (step 8), and then the NIC driver will issue a top-half interrupt to the CPU, followed by a bottom-half interrupt (step 9). After that, kernel threads will wake up *epoll_wait()* function in gRPC (step 10). As a part of the asynchronous framework, only after epoll_wait notifies gRPC (step 11), the data will be copied from a kernel receive-buffer to gRPC incoming buffer by TCP function *recvmsg* (step 12). After that, the server will start data processing in user space, and the encoded data in the gRPC Incoming Buffer will be moved to the Payload buffer to be decoded (step 13). After decoding, the raw RPC data will be submitted to the server from the Payload buffer for its function processing (step 14) and the service function will process the data.

[1]Although the I/O multiplexing functions used in gRPC is platform dependent, *epoll_wait()* is the most widely used on Linux. Thus, we mainly focus on *epoll_wait()* function.

C. Challenges of Fusing RDMA Into gRPC

Since gRPC is a commonly used system platform, we use it as the basic platform in our system development. In order to fuse RDMA into gRPC for its low latency merits subject to retaining the merits of gRPC, we must address multiple technical challenges in both network connection management and memory management.

1) Network Connection Management: RDMA-BP uses a polling thread to keep checking the memory buffer to detect incoming data. In RDMA-Event method, users create a completion event channel, which is a FD to deliver the completion event from kernel space to user space. Then, users call blocking function *ibv_get_cq_event()* to check incoming data. In comparison, gRPC detects incoming connections by *epoll_wait()* that monitors I/O events happening on FDs. A technical issue is how to effectively incorporate the two RDMA methods into the existing gRPC facility for high performance.

The other issue is research-related. As we know, RDMA-busy-polling and RDMA-event can effectively handle certain connection workloads but do not work well on others. Can we develop a new method that minimizes the limits of the two methods and retains their merits? We will answer this question in Section IV-D.

2) Memory Management: gRPC is designed and implemented as an asynchronous framework that is beneficial to both execution efficiency and application programming productivity. gRPC introduces concepts of *closure* and *closure scheduler*. A *closure* combines together a function call with its arguments, and a *closure scheduler* collects execution context for every *closures*. With the help of *closure* and *closure scheduler*, functions can be executed asynchronously by scheduling *closures*, which can be executed in the future by waiting for their I/O events if necessary. Since the execution time of closure is unknown in advance, gRPC dynamically allocates memory to hold RPC data during *closure* execution. As a TCP-based system, all gRPC internal buffers are managed by OS that facilitates dynamic memory allocation and data movement from user space to kernel space. Accordingly, the memory management in gRPC is supported by multi-level memory buffers and intermediate memory copies to support its asynchronous framework.

OS plays an important role in the memory management for its asynchronous execution structure in gRPC. The low latency of RDMA comes from its pinned memory buffers in user space, which is out of the control of OS but is managed by an RDMA-enabled NIC. The challenge to fuse RDMA to gRPC without support from OS on dynamic memory allocations is on how to effectively emulate dynamic data allocations in user space. The objective is to gain the benefits of low latency from RDMA, subject to retaining the asynchronous execution structure of gRPC. To work around this challenge and to simplify implementations for RDMA in RPC, several prototypes only provide synchronous execution, e.g., [20], [21], [22], which cannot be applied to

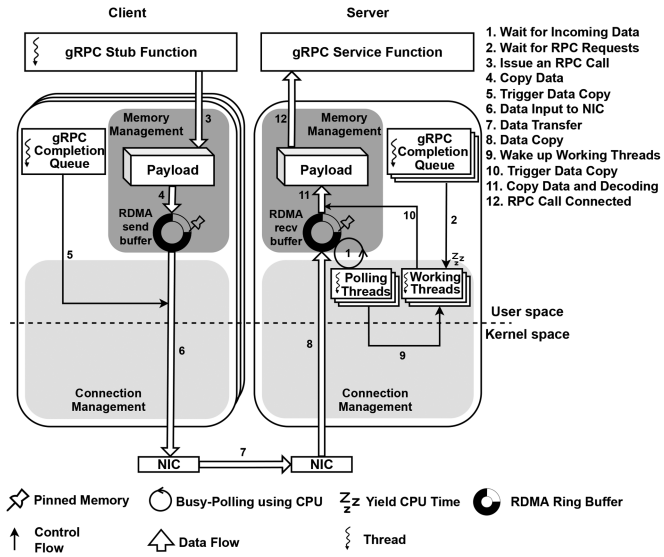


Fig. 3. Overview of RR-compound.

TABLE II
NODE SPECIFICATIONS

Items	Specifications
CPU	Intel(R) Xeon(R) Gold 6148 CPU
MEMORY	188GB
NIC	Mellanox ConnectX-5 100Gbps
NIC Driver	4.9-4.1.7
NIC Firmware	16.24.1000
OS	Red Hat Enterprise Linux Server release 7.9
Kernel	Linux 3.10.0-1160.83.1.el7.x86_64

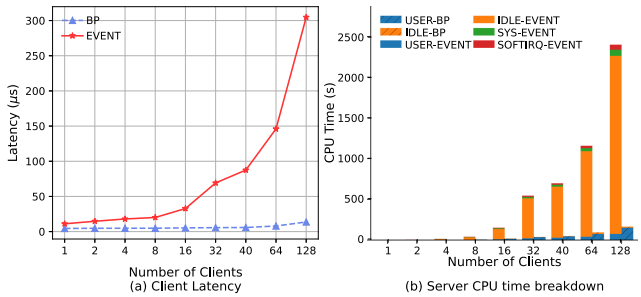


Fig. 4. Busy-polling versus event latency.

gRPC, because the memory management, thread management and gRPC *CompletionQueue* are tightly coupled with gRPC's asynchronous framework. We will explain how we accomplish our goal in Section IV-F.

III. An Overview of RR-Compound

Fig. 3 presents the basic architecture of RR-Compound, where the network connection management and the memory management will be discussed in detail in Section IV. The interface to users is the same as that of gRPC for general-purpose programming. We have designed a new connection method that combines the merits of both RDMA busy-polling method and RDMA event-based method. The new method is called BPEV, which consists of two groups of light-weight threads: working threads and polling threads shown in Fig. 3. Kernel send/receive buffers are replaced by RDMA send/receive buffers. Fig. 3 also gives the data path and execution path of RR-Compound. On the server side, one or a small number of polling threads are running to check incoming data in a busy-polling manner (step 1), and working threads will call *epoll_wait()* function to wait for events (step 2). On the client side, the client starts to issue an RPC call by placing related parameters and data in the Payload buffer to be encoded in a gRPC format (step 3). The encoded data is copied from the Payload buffer to the RDMA Send Buffer (step 4). After the client calls gRPC function *cq.next()* to notify the availability of the data (step 5), data in the RDMA send buffer will be sent directly to NIC (step 6). After the data is transferred from client NIC to server NIC (step 7), it will be directly moved to RDMA receive buffer (step 8). Immediately, polling threads in the server will detect this incoming data and wake up working threads (step 9). Then a working thread will notify the availability of the data in the RDMA receive buffer (step 10). After that, the data is moved to the Payload buffer to be decoded (step 11). Finally the raw RPC data is submitted to the service function for its execution (step 12) in the server.

IV. The Core Design and Implementation

A. Experimental Environment

Comprehensive performance evaluation by intensive experiments is a foundation for the design and implementation of RR-Compound. The configurations of our experiment platform presented in this sub-section will be referenced in the rest of paper when experiments and their results are reported. Unless specified, we use the following configurations for our performance evaluation. The cluster used for the experiments consists of 9 nodes with the following two setups: (1) *single client and single server*, where the client issues requests on one node and the server responds the requests from another node, and (2) *multiple clients and single server*, where the clients are distributed evenly and run across 8 nodes, and the server execution is conducted on a dedicated node. All nodes are configured with the specifications listed in Table II.

We have developed RR-Compound based on gRPC v1.38.0. All performance results of gRPC are produced from this version and our extension to ensure fair comparisons.

B. Event-Based versus Busy-Polling Methods

We have conducted intensive experiments to compare the latency changes between busy-polling and an event-based method for network connection management of RDMA under two different request frequencies. We use "ibverbs" to implement a microbenchmark consisting of a client and a server. Each client issues 200 K requests in a ping-pong pattern to the server. The number of clients ranges from 1 to 128 in our experiments. One parameter in our experiments is the request frequency, which reaches the peak when requests are continuously sent to the server without any delay. For such a case, Fig. 4(a) presents the average latency curves per request as the number

of clients increases from 1 to 128. It shows that the latency curve of busy-polling almost stays constant when the number of connections is lower than the number of CPU cores (40) on the server. After that, the latency curve slowly increases. In contrast, the latency of the event-based method starts to increase when the number of clients is more than 8, and sharply increases when the number of clients is more than 40.

To investigate the reason behind this behavior, we measure the total server CPU execution time breakdowns in Fig. 4(b), where the right bar in a pair is for busy-polling and the left is for the event-based method. Since the server processes client requests in a batch, we use the total server time breakdowns to understand where the time goes for each group of requests. The process of busy-polling (labeled as "USER-BP") consists of two parts: (1) busy detection in a loop for requests, and (2) processing requests after each successful detection. We also measured the idle CPU time during busy polling, which is a very small percentage of CPU time. The increase in busy-polling time observed on the server in Fig. 4(b) is proportional to the rise in client latency in Fig. 4(a) as the number of clients grows.

The event-based method is handled in both OS space and user space in the server, and the server running time is spent in four parts: (1) upon a request, a bottom-half interrupt[2] is scheduled to generate the event in kernel space, labeled as "SOFTIRQ-EVENT"; (2) OS further wakes up waiting threads for the request, labeled as "SYS-EVENT"; (3) the waiting threads process the request in the user space, labeled as "USER-EVENT"; and (4) the server is idle when it does not get requests to process, labeled as "IDEL-EVENT". Fig. 4(b) shows that the processing times in the user space of busy-polling ("USER-BP") and in the event-based method ("USER-EVENT") are comparable. However, when the number of clients is more than 8, the server idle time starts to increase proportionally as the number of clients increases for the event-based method. Compared with the total server time, the idle time is up to more than 80%. We have further looked into the reason for the increasingly high idle times. Here are our findings. Under the current Mellanox driver implementation, there is only one CPU core used for processing the interrupts because the event handling procedure is implemented by *tasklets* APIs in Linux kernel. A tasklet procedure in Linux generally cannot be executed by more than one CPU core concurrently. With this restriction, the interrupts are handled slowly by a small portion of execution time (see "SOFTIRQ-EVENT" in Fig. 4(b)). Thus, the server spends increasingly more time waiting to process requests. This explains why the client latency per request increases dramatically as the number of clients increases in Fig. 4(a). In addition, the kernel is responsible for waking up threads waiting for events, with frequent context-switching overhead that is reflected by the execution time portion of "SYS-EVENT".

However, the event-based method proves highly effective in scenarios with low communication frequency. To simulate such a scenario, we set the sending interval at 300 μs.[3] Fig. 5(a) shows that the event-based method is scalable, but the busy-polling gets into trouble. Notably, busy-polling exhibits increased latency after 40 clients due to the significant CPU cycles wasted by dedicated polling threads. Fig. 5(b) shows the server CPU time breakdown. The user-space CPU time consumed by busy-polling significantly increases, while a small amount of time is dedicated to user space in the event-based method. This inefficiency in busy-polling escalates with the addition of more clients, stemming from continuous message detection even when no requests are present. Conversely, the event-based method keeps the CPU available for other tasks when request frequency is low, showcasing its effectiveness in managing low-frequency connections without excessive consumption of CPU cycles.

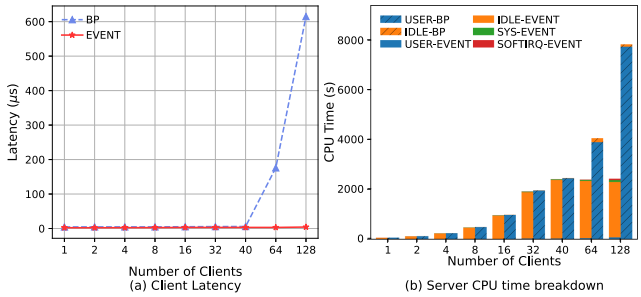


Fig. 5. Busy-polling versus event latency with 300 μs sending delay. BP exhibits a sharp increase in latency as the number of clients grows.

C. Ineffectiveness of a Hybrid Method

The cases presented in Section IV-B indicate that we may have an opportunity to develop a hybrid method by switching between busy-polling and event-based methods to adaptively respond to the workload changes. However, the hybrid approach is infeasible for three reasons. First, in practice, the request frequency is highly workloads dependent. We need a monitoring mechanism to dynamically collect the connection frequency data in order to timely make a switching decision. The overhead of this mechanism can be nontrivial. Second, switching between the two methods can also cause high overhead, considering the jumping between the user and kernel spaces.

Most importantly, our experiments demonstrate that both methods can encounter scalability issues when processing workloads with identical connection frequencies. We observed that a sending interval of 100 μs represents a critical "pain point" at which both BP and RDMA-Event struggle to manage medium communication frequencies. As depicted in Fig. 6, neither busy-polling nor event-based methods scale beyond 40 clients. In this case, BP is still suffering from CPU contention. However, RDMA-Event is constrained by the tasklets mechanism in the Linux kernel that the RNIC driver implementation of event mode relies on, which employs only one kernel thread to handle interrupts, resulting in high latency. This experiment underscores that the bottlenecks discussed for each method can coexist within the

[2]A kernel interrupt handler splits its execution into two halves: the top-halve and the bottom-halve. The top-halve quickly acknowledges the interrupt and schedules the bottom-halve at a proper time. Since the time spent in top-halve is so short, we do not present it in the figure. The bottom-half does the rest of processing that needs much longer time.

[3]This value exposes the scalability issue of busy polling. We identify the value by trails, which may be different on other platforms.

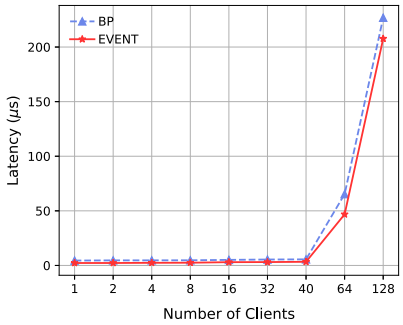


Fig. 6. Busy-polling versus event latency with 100 μs sending delay. Both BP and RDMA-Event experience an increasingly high latency.

same workload and system environment, thus confirming that a hybrid approach combining these methods is impractical.

D. BPEV: A New Management Method

We have identified the following two critical bottlenecks that seriously increase the latency and limit the throughput based on our experimental studies in Section IV-B: (1) The use of a busy-polling mechanism leads to a substantial waste of CPU cycles, particularly under conditions of low request frequency. This inefficiency results in escalating latency as client numbers grow. (2) The interrupt function in the RDMA-Event is executed serially, which causes increasingly long idle time on the server as the number of clients increases. This is the main reason why the event-based method is unable to handle frequent requests.

To mitigate the issue of CPU cycle wastage, we have developed a method termed Busy-Polling Time-Sharing (BPTS). Unlike the traditional approach of dedicating a single thread to each connection, BPTS employs a single thread to manage multiple connections through a round-robin, time-sharing manner. To integrate BPTS with gRPC, it is necessary to embed the RDMA busy-polling logic within gRPC's polling engine. This integration allows for the simultaneous monitoring of RDMA connections and gRPC's internal FDs, such as those listening FDs designed to accept incoming TCP connections.[4]

Algorithm 1 illustrates the BPTS in pseudocode, implemented within the $WaitEvents$ function of gRPC's polling engine, responsible for awaiting I/O events. The algorithm relies on three key parameters: $epfd$, the epoll file descriptor monitoring a collection of FDs; C, a set of RDMA connections; and $timeout$, denoting the maximum period to await events. The output, $events$, encompasses events such as $EPOLLIN$ (indicating readiness to read) and $EPOLLOUT$ (indicating readiness to write).

In Line 4, the $epoll\_wait$ function is iteratively invoked to gather events from monitored FDs. To prevent blocking during RDMA polling, we configure $epoll\_wait$ with a zero timeout (the third parameter). Subsequently, from Lines 6 to 14, the algorithm iterates over each RDMA connection to detect incoming messages or pending write operations, generating $EPOLLIN$ or $EPOLLOUT$ events, respectively. The polling engine processes these events, facilitating RPC reception and responses dispatch.

[4]RR-Compound still relies on the TCP to exchange queue pair, memory regions, and other metadata to make RDMA connections.

Algorithm 1: Wait for I/O Events With BPTS.

Input: epfd - the epoll file descriptor to handle new connections
Input: C - RDMA connections
Input: timeout - user-specified polling timeout
Output: events - an array of readable/writable events

1: ▷ Running on user-created working threads
2: **procedure** WaitEvents
3: **while** $events = \emptyset$ and elapsed time $< timeout$ **do**
4: $epoll\_wait(epfd, events, 0)$ ▷ Monitoring gRPC's internal FDs
5: ▷ Traverse RDMA connections handled by the thread
6: **for** c in C **do**
7: **if** c has incoming messages **then**
8: $events = events \cup EPOLLIN$ ▷ Produce a readable event
9: **end if**
10:
11: **if** c has pending writes **then**
12: $events = events \cup EPOLLOUT$ ▷ Produce a writable event
13: **end if**
14: **end for**
15: **end while**
16: **end procedure**

It's crucial to highlight that gRPC, does not automatically initiate new threads (Section II-B); while, users instantiate threads to engage the polling engine through gRPC's API calls. Algorithm 1 is designed to enable a single thread to efficiently manage multiple clients. Therefore, users must ensure that the number of threads created aligns with the available CPU cores. Adhering to this guideline minimizes CPU contention, enhancing system efficiency and overall performance.

This enhanced approach presents two significant advantages. First, the efficiency of polling is high, as it merely involves a quick check of the ring buffer's head to ascertain the presence of new messages–a process that typically consumes only several nanoseconds [15]. Consequently, utilizing a single thread to manage hundreds of connections does not noticeably impact latency. Second, by aligning the number of polling threads with the count of CPU cores, the overhead associated with inter-thread scheduling is substantially reduced. The BPTS design is ideally suited for scenarios requiring high-frequency communication, such as Key-Value stores.

Nonetheless, BPTS necessitates that polling threads remain active continuously, leading to full utilization of all CPU cores even in the absence of requests. Furthermore, the implementation of busy-polling in user space diverges fundamentally from gRPC's event-based polling mechanism. This discrepancy requires constant invocation of $epoll\_wait$ within the busy-polling

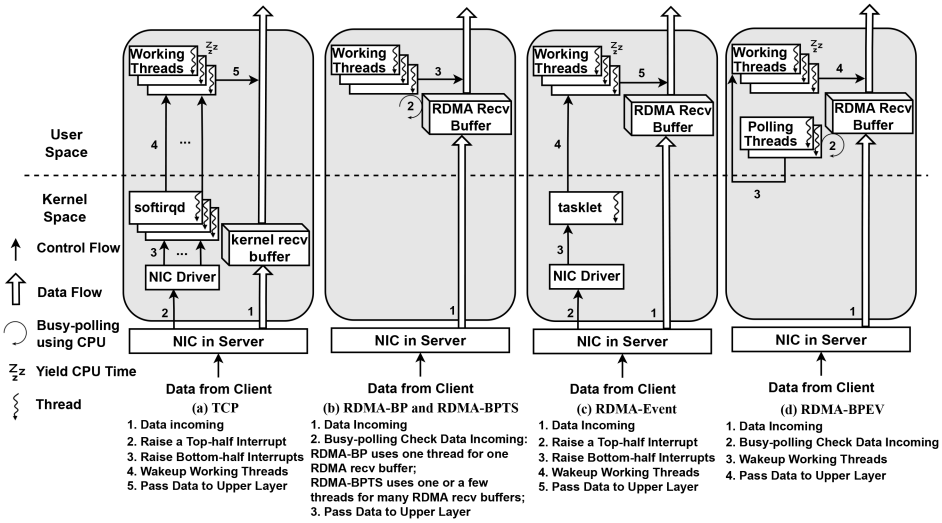


Fig. 7. Overview of 5 connection managements.

loop to monitor socket FDs for new connections. Failure to do so results in the server's inability to accommodate incoming clients, as the threads are preoccupied with polling and cannot process TCP connections. The interleaving of busy-polling and $epoll\_wait$ calls incurs system call overhead.

Having intensively studied the insights into busy-polling and event-based methods, we are ready to develop an effective network connection management for RDMA to retain the merits and eliminate the limits in both methods.

We have designed an event-based method named "Busy-Polling driven with EVents" (BPEV), implemented in user space with support from the OS kernel. The core concept involves deploying a limited number of dedicated threads, referred to as "polling-threads", which actively engage in busy-polling to promptly check messages in the receiving buffer across all connections. Once detecting new messages, a polling thread signals a working thread through *eventfd*, an event wait/notify mechanism supported by the OS. Each RDMA connection is assigned a unique *eventfd*, registered with gRPC's polling engine (added to "$epfd$") during connection establishment. Within the polling engine, "$epoll\_wait$" is used to manage events from both RDMA and TCP connections. This seamless integration with the gRPC's event-based architecture involves the straightforward addition of dedicated polling threads.

BPEV comprises two procedures, as illustrated in Algorithm 2. The *RDMAPoll* procedure operates on dedicated polling threads, monitoring all RDMA connections to identify new messages or pending writes. If a polling thread detects new messages from any connection (Line 6), it wakes up working threads using "$eventfd\_write$". In *WaitEvents*, an initial attempt at busy-polling occurs within a predefined timeout period, "$bp\_timeout$" (Lines 16-26). Recognizing that waking up threads from sleep incurs a kernel-driven context-switch cost, frequent context-switch operations can potentially bottleneck overall performance. To minimize this cost, we have optimized BPEV by enabling working threads to engage in busy polling within the timeout period before resorting to "$epoll\_wait$". In scenarios with high request frequency, this optimization allows continuous request processing without frequent kernel intervention, significantly reducing context switch overhead. In situations of exceptionally high request volume, BPEV operates nearly identically to busy-polling of RDMA, leveraging its optimal performance. Conversely, with moderate request frequency, BPEV benefits from both low-latency busy-polling and efficient kernel scheduling triggered by less frequent events.

E. Analysis of the Five Connection Methods

We have integrated four connection management methods (BP, BPTS, EVENT, and BPEV) into gRPC to demonstrate the effectiveness of BPEV by comparing their performance results with that of the conventional TCP connection method in gRPC. Fig. 7 presents the control flows and data flows of the five connection methods in our experiments. Fig. 7(a) is the basic TCP connection for gRPC, where interrupts in two phases are conducted to inform a working thread that incoming data is available in the kernel buffer. In the last step of the connection, the working thread moves the data from the kernel buffer to a user buffer. Fig. 7(b) shows that BP uses a dedicated thread to check on one buffer and that BPTS uses one or a few threads to check on many buffers. Both methods are in user space without the involvement of OS. Fig. 7(c) shows the RDMA-Event method, where interrupts in two phases are conducted to inform a working thread that incoming data is available in the RDMA buffer. Fig. 7(d) shows our newly designed RDMA-BPEV that has been well discussed in Section Section IV-D. Fig. 7 also illustrates the reason for the low latency of RDMA by comparing the last step of the connection.

As we intend to focus on analyzing the raw connection performance of each method, the remote procedure in the server should be simple, such that it directly replies to requests without any other data processing. We have evaluated the five methods on a cluster described in Table II. Each client sends a request with a 4 KB payload, and the server quickly replies with 4 KB

Algorithm 2: Wait for I/O Events With BPEV.

Input: epfd - the epoll file descriptor to handle new connections
Input: C - RDMA connections
Input: timeout - user-specified polling timeout
Input: bp_timeout - timeout for switching to epoll
Output: events - an array of events

▷ Running on the dedicated polling threads
procedure RDAMPoll
  Register each RMDA connection's $eventfd$ to $epfd$
  while running **do**
    for c in C **do**
      if c has incoming messages or pending writes **then**
        $eventfd\_write(c.eventfd, 1)$ ▷ Wake up working threads
      end if
    end for
  end while
end procedure
▷ Running on user-created working threads
procedure WaitEvents
  ▷ Busy-polling before resorting to epoll
  while elapsed time $<$ $bp\_timeout$ **do**
    for c in C **do**
      if c has incoming messages **then**
        $events = events \cup EPOLLIN$ ▷ Produce a readable event
      end if

      if c has pending writes **then**
        $events = events \cup EPOLLOUT$ ▷ Produce a writable event
      end if
    end for
  end while

  ▷ busy-polling is timeout, turn to kernel-supported $epoll$
  if $events \neq \emptyset$ **then**
    $epoll\_wait(epfd, events, timeout - bp\_timeout)$
  end if
end procedure

payload to the client. This simple communication pattern allows us to analyze performance characteristics. For BP, we fixed the number of working threads to the number of clients. For the rest of the methods, we fixed the number of threads to 40 when the number of clients is less than 64; to 64 threads for 64 and 128 clients. In our experiments, we tended to oversubscribe threads because gRPC has internal locks that hinder the full utilization of CPU cores.

Fig. 8(a) shows when there is four client, the throughput of gRPC over IPoIB and EVENT are below 50 k RPCs/s due to the overhead of complex network stack and system calls. In contrast, the throughput of BP, BPTS, and BPEV reached to 132 K, 135 K, and 120 K RPCs/s, respectively. The high throughputs of BP, BPTS, and BPEV were contributed by the low latency of busy-polling. When the number of clients reached to 32, the throughput of BP, BPTS, and BPEV demonstrated a respective capacity of 812 K, 785 K, and 620 K RPCs/s. Here, The highest throughput was achieved by BP due to the lower number of clients compared to the number of CPU cores, allowing busy polling to optimally utilize CPUs for detecting incoming messages. In contrast, IPoIB and EVENT merely obtained 367 K and 413 K RPCs/s, respectively, due to the heavy socket APIs and limited tasklet interrupt processing ability. Fig. 8(b) gives the evidence, which shows 71% and 73% of CPU time are idle for IBoIP and EVENT, indicating that threads yield CPU cores waiting for events to work on the next tasks.

When the number of clients reaches 64 which is more than the number of CPU cores of 40, the inefficiency of BP is exposed. As shown in Fig. 8(a), the throughputs of BPTS and BPEV both are more than 790 K RPCs/s, compared with 598 K RPCs/s for BP, which is an improvement of 32%. The poor performance of BP comes from the CPU preemption by using a dedicated thread to check on one client, which deteriorates performance as the number of clients increases. As shown in Fig. 8(b), for 128 clients, BP spent 148 s in user space, which is 1.8 times longer than that of BPTS and 2.3 times longer than that BPEV. The throughput of BPEV is up to 1417 K RPCs/s, which is 49% higher than that of BPTS.

To verify the effectiveness of BPEV under a limited connection rate, it is necessary to set a lower sending rate than we used in Fig. 5. Under this setting, we consider the RDMA performance under the gRPC framework. Therefore, we need to set a higher sending interval (400 μs) than we used in Section IV-B (300 μs) to offset the overhead from introducing gRPC. As shown in Fig. 8(c), for 128 clients, BP had the lowest throughput of 94 K RPCs/s; and IPoIB was the next worst one with a throughput of 228 K RPCs/s. As we illustrated in Section IV-B, a lot of CPU cycles are wasted by creating an unnecessary amount of polling threads. Indeed, Fig. 8(d) shows that BP spent 2.8 times and 15.2 times longer on busy time than BPTS and BPEV, respectively. In contrast, EVENT and BPEV saved a lot of CPU time since both methods prevent unnecessary busy-polling operations from happening. From the idle time portion in the execution time breakdowns in Fig. 8(d), we can see that BPEV method is as efficient as RDMA-Event and keeps the merits of high throughput.

F. The Memory Management

The RPC data communication between a client and its server is done by two major steps in each node: (1) to prepare a data package, and (2) to send the package to the destination node via network, where the memory management in the system design plays an important role. We will discuss how gRPC makes this happen with the support of OS, and how our memory management is done in order to fuse RDMA into gRPC without a support of OS.

Fig. 9(a) depicts the execution and data flows for an RPC request from a client or for its server response. In the data

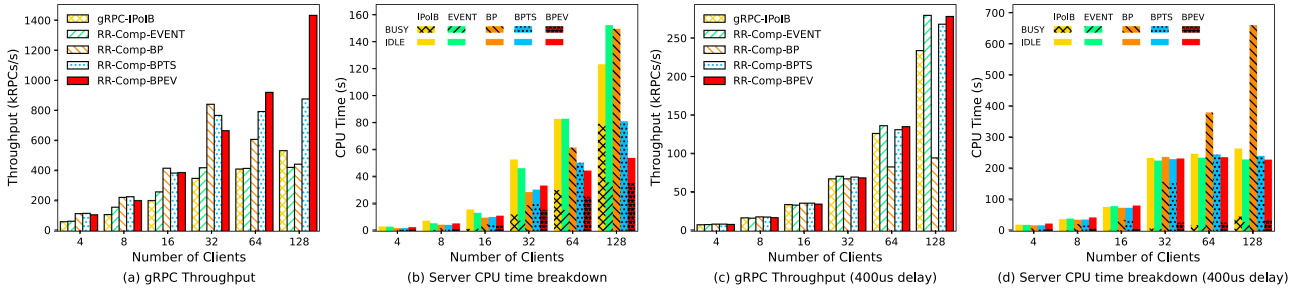


Fig. 8. RR-Compound throughput.

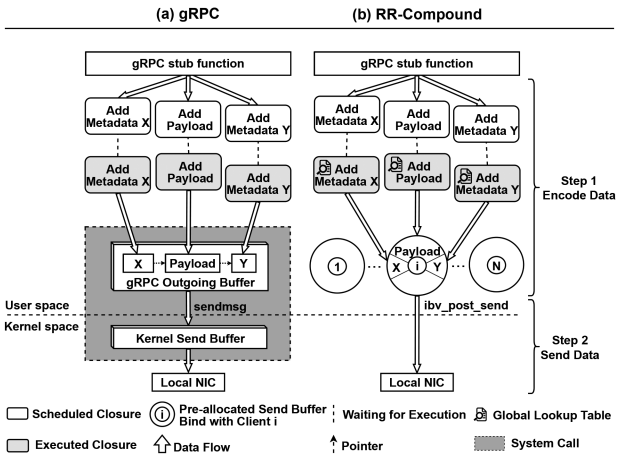


Fig. 9. Memory management of gRPC and RR-compound.

preparation step, an RPC request/response object is created from the client/server, which carries the required data. When a stub/service function is invoked/returned, the payload of the RPC object will be encoded with metadata to describe its attributes (Step 1 in Fig. 9(a)). The process of encoding is implemented with many transient functions called closures to allow gRPC to work asynchronously. The closure mechanism allows a function to be executed at a proper time, such that a caller can immediately return for another task after scheduling a closure without being blocked. When executing a closure that is for encoding RPC, an OS-managed temporary memory space (called a slice in gRPC) is allocated to carry the payload or metadata. Multiple slices are organized into a linked list as the data package in the outgoing buffer. For example, in Fig. 9(a), metadata X, payload data, and metadata Y are linked together and stored in the outgoing buffer.

After an outgoing buffer is fully constructed, gRPC uses *sendmsg* to copy the data by iterating each slice in the outgoing buffer to the kernel buffer (see Step 2 in Fig. 9(a)), where OS connects the delivery destination address. After then, each memory slice can be safely destructed and recycled in the outgoing buffer by the OS. The RPC data object is now ready to be sent from the kernel buffer to its destination node via the NIC that is in the bottom of Fig. 9(a). Since gRPC uses Linux-provided socket API, *sendmsg*, to send the data, it uses the fd (file descriptor) of the connected socket to locate the request/response destination of the RPC. In gRPC design, the socket fd is maintained in the transport layer, so that the destination of an RPC can be found by looking up attributes at the time of data transmission.

Supported by OS, dynamic memory allocations are effectively used for the asynchronous framework in gRPC. However, fusing RDMA into gRPC subject to retaining the asynchronous framework of gRPC, we do not have a privilege of OS support for dynamic memory allocations and for handling network connections. We cannot dynamically allocate a temporary buffer and register it to NIC, considering the significantly high overhead in memory-to-NIC registration. Thus, we use a pre-allocated memory region for each data communication, which is pinned and registered to NIC. Since RDMA bypasses the kernel space, we maintain a pre-allocated send-/recv-buffer for each connection in user space to carry the RPC data. Fig. 9(b) shows that N send-buffers are created for N connections. Each send-buffer is organized in the format of a ring data structure, which is called ring buffer. In this way, asynchronous sending operations can be achieved. The address of a send-buffer can be fetched with a Global Lookup Table (to be discussed next) by data encoding closures so that the encoded data can be directly written to the buffer without another data copy (Fig. 9(b), Step 1). Compared with gRPC, in the data encoding step, we eliminate an extra memory copy. For example, in the data sending step, we directly move the data from the send-buffer to NIC without an OS support (Fig. 9(b), Step 2).

(1) A pre-allocated ring buffer: This buffer is in a pinned memory region, with which we emulate dynamic memory management at a very low cost. We maintain a fix-sized circular memory space for each RDMA connection. Although one-buffer per-connection may not perfectly utilize the memory space, maintaining a global send-buffer for all connections can cause much bigger performance concerns, such as a high lock contention when many threads try to write the shared buffer. When executing a closure to write payload or metadata, we instruct the closure to append the data to the tail of a selected ring buffer (Step 1 in Fig. 9(b)). The starting address of the written data is returned and recorded, which will be used as a pointer for us to manage the ring buffer in the physical memory space. After then, an RDMA write operation, *ibv_post_send*, uses the buffer address to send data to the remote node directly. After this is done, we recycle the corresponding memory space from the head of the ring buffer. The ring buffer on the receiver side is also maintained in the same structure to minimize the memory management overhead.

(2) A global lookup table: This table contains a mapping from $grpc\_channel$ to the address of the send-buffer. An gRPC channel provides the connection information of the peer. We build a mapping from an RPC $grpc\_channel$ to its send-buffer at the time when the gRPC client/server accesses the buffer address. We implement the global table existing in singleton - only one instance exists in the system. This design allows us to access the mapping information anywhere in the codebase. As an RPC call that contains $grpc\_channel$ in its attributes is created, the table will be looked up to locate the address of the send-buffer, and then the encoded data can be written to the buffer. Since the table is read-only, there will be no lock contention concerns, so the lookup overhead is negligible.

With the support of these two data structures, we retain the asynchronous framework since process of data encoding is still implemented with closures. If a closure requires a memory space to carry the payload or metadata, we look up the global table for data communication to match its send-buffer. Then, we allocate memory space from the send-buffer to store the RPC data. After data encoding, the RPC data is directly delivered from the local send-buffer to the destination receive-buffer via NICs, achieving a "zero-copy" task, which is in Step 2 in Fig. 9(b).

V. Performance Evaluation

First, we use microbenchmarks to compare RR-Compound with mRPC [13], a recently developed and customized RPC system leveraging RDMA aiming for performance and advanced features, such as live upgrade, Quality-of-Service (QoS) at low cost. Then, we investigate the optimizations of RR-Compound and discuss the tuning of its parameters. Finally, we conduct extensive experiments to assess RR-Compound in comparison with gRPC across two representative workloads: the KV-store and TensorFlow.

A. mRPC versus RR-Compound on Scalability and Latency

One of our objectives in designing and implementing RR-Compound is to strike a balance between low latency and high throughput of RDMA, while also retaining the ease of programming interface from gRPC. We acknowledge that achieving the highest performance within the constraints of the gRPC framework is impractical. System scalability is our major concern for production systems. In this section, we use a microbenchmark to evaluate the scalability and latency of RR-Compound, compared with mRPC. This system abstracts RPC as a service instead of linking the RPC library into applications to allow live upgrade and advanced features such as QoS without the overhead of introducing a proxy.

We follow the settings provided by mRPC's benchmark, with 32-byte requests and 8-byte responses, to evaluate the aggregated throughput among all the clients. Each mRPC client keeps 32 concurrent RPCs. To avoid the HTTP2 overhead of RR-Compound, we use the gRPC's streaming APIs. This practice follows gRPC's official benchmark [23]. The clients are evenly distributed to eight nodes. For mRPC, we use the same number of threads for the server as the client. This setting is suggested in paper [13]. For RR-Compound, we adopt the BPEV connection management method that uses 38 working threads to serve all the clients and two polling threads with 100 μs busy-polling timeout.

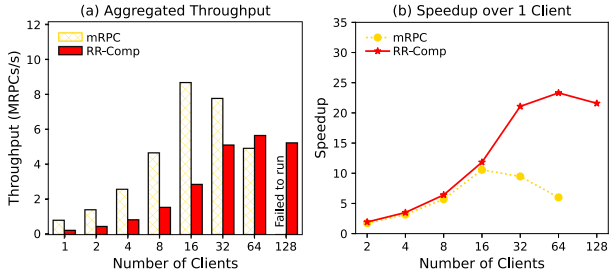


Fig. 10. Scalability comparisons between mRPC and RR-Compound by increasing the number of clients.

Fig. 10(a) plots the RPC rates for both mRPC and RR-Compound as the number of clients increases from 1 to 128. For a solitary client, mRPC demonstrates a very high throughput with an aggregated RPC rate of 820 K RPCs/s, surpassing RR-Compound, which achieves 242 K RPCs/s. This discrepancy is understandable considering RR-Compound relies on gRPC, characterized by an extended execution path leading to significant delays. As the number of clients surpasses eight, mRPC accelerates to 4.66 M RPCs/s while RR-Compound achieves 1.55 M RPCs/s. Evaluating the speedup of eight clients over one client, mRPC records a 5.68x improvement, while RR-Compound achieves a 6.41x improvement (refer to Fig. 10(b)). The peak throughput for mRPC occurs at 16 clients, reaching 8.67 M RPCs/s. Subsequently, scalability issues become apparent in mRPC, causing a decline to 7.76 M RPCs/s when increasing to 32 clients. In contrast, the throughput of RR-Compound rises to 2.86 M RPCs/s. With 64 clients, mRPC's throughput further drops to 4.92 M RPCs/s, marking approximately half of its peak throughput. In the meantime, RR-Compound excels at 5.65 M RPCs/s, outperforming mRPC by 14.77%. As the number of clients reaches 128, RR-Compound experiences only 7% drop in RPC rate, while mRPC struggles to cope with the increased demand from busy connections.

The significant divergence in scalability between mRPC and RR-Compound can be attributed to their distinct approaches to RDMA connection support. mRPC adopts a two-sided RDMA implementation coupled with a basic busy-polling mechanism, exerting demands on CPU resources for both clients and the server. While mPRC exhibits impressive performance with a limited number of clients, it faces a substantial decline in performance, leading to service interruptions in scenarios such as serving 128 clients, where ibverbs failure becomes a critical issue. In contrast, the demonstrated scalability of RR-Compound, as shown in Fig. 10, stems from specific design choices. RR-Compound's BPEV design takes the benefits of event-based and polling-based methods. By allocating threads equal to the available CPU cores, RR-Compound mitigates the risk of preempting CPU cycles, ensuring consistent and efficient utilization. Furthermore, RR-Compound leverages the one-sided RDMA write implementation, offloading CPU resources on the receiver

TABLE III
ROUND-TRIP LATENCIES IN MICRO-SECOND

Solution	Median Latency	P99 Latency
mRPC	10.16	12.76
RR-Comp	7.22	7.33

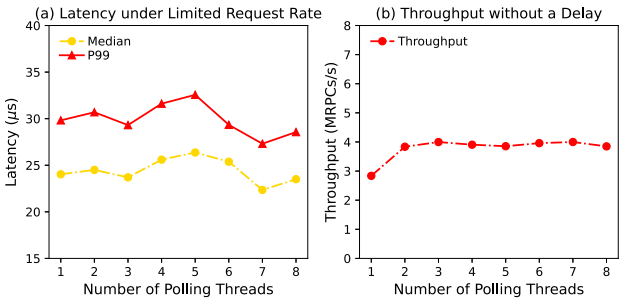


Fig. 11. (a) 320 clients are sending RPCs in low communication frequency; (b) 64 clients are continuously sending RPCs.

side. This further enhances scalability and overall system performance.

To further highlight the efficacy of the connection mechanisms in RR-Compound, a comparative analysis of latency with mRPC and RR-Compound is conducted, which results are shown in Table III. In this experiment, we follow the mRPC's configurations to evaluate RR-Compound, encompassing small RPCs with 64-byte requests and 8-byte responses in a ping-pong mode. Since RR-Compound adopts one-sided RDMA write, it exhibits lower latency compared to mRPC. The median latency comparison reveals that RR-Compound outpaces mRPC by 1.41x. When evaluating the 99th (P99) percentile tail latency, RR-Compound demonstrates a 1.74x improvement over mRPC. Note that the latency numbers for mRPC are higher than those reported in their original paper. We attribute this variance to the lower base frequency of the CPUs in our experimental settings.

B. Parameters Tuning and Disscussions

Varying Polling Threads Number: We conduct investigations into the optimal number of polling threads required by BPEV, examining both low and high load conditions. In a low load scenario, we initiate a substantial number of clients (320) with low communication frequencies, and assign 32 working threads dedicated to handling RPCs on the server, each responsible for ten clients. The RPC request frequency adheres to an exponential distribution with a rate parameter $\lambda = 1$. To evaluate the impact of polling threads on performance, we increase their count from 1 to 8, monitoring changes in latency. A reduction in latency with the addition of more polling threads would suggest that the limited polling threads serve as the bottleneck.

Fig. 11(a) visualizes the impact of increasing polling threads on latency. With a single polling thread, the median latency observed is 24.03 μs, and the 99th percentile (P99) latency is 29.82 μs. However, the addition of more polling threads does not significantly alter latency, with variations remaining under

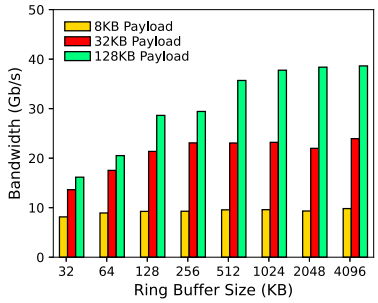


Fig. 12. Bandwidth by varying ring buffer size.

5 μs. This marginal change in latency can be attributed to the low rate of RPC requests, where polling threads primarily spend time inspecting the ring buffer head for new messages, and the costly "$eventfd\_write$" operations are infrequently required. Consequently, under low load conditions with up to 320 clients, the polling thread count does not emerge as a bottleneck, indicating that BPEV's design efficiently supports managing a large number of clients with a minimal number of polling threads under low load conditions.

Additionally, we simulate a high-load scenario with a small number of clients (64) continuously issuing RPCs, where throughput becomes a more appropriate performance metric. As depicted in Fig. 11(b), a single polling thread allows RR-Compound to reach a throughput of 2.8 M RPCs/s. With two polling threads, the throughput increases to 3.8 M RPCs/s, and with three, it further reaches 4.0 M RPCs/s. Beyond this point, the throughput exhibits negligible sensitivity to further increases in polling thread count. This outcome underscores that deploying additional polling threads fails to continuously improve performance, even under high load conditions. Therefore, allocating a limited number of polling threads, coupled with dedicating more CPU cores to worker threads, proves to be a pragmatic choice.

Varying Ring Buffer Size: The ring buffer size plays a crucial role in influencing the performance of large RPCs. A small ring buffer may frequently block the peer from writing, resulting in an increased number of RDMA writes. Subsequently, this causes reduced bandwidth and increased latency. In contrast, a large ring buffer may introduce memory space overhead, particularly when handling a large volume of connections. Hence, it is imperative to explore an appropriate ring buffer size according to different RPC sizes.

As shown in Fig. 12, for RPCs sized at 8 KB, RR-Compound achieves a bandwidth of 8.1 Gb/s with a 32 KB ring buffer, capable of accommodating up to four RPCs. Beyond this point, enlarging the ring buffer size does not yield bandwidth improvements for small RPCs, as the ring buffer on the peer side seldom reaches full capacity. However, the ring buffer size significantly impacts the large RPCs. With 128 KB RPCs, bandwidth significantly increases with a growing ring buffer size. For example, with a 32 KB ring buffer, RR-Compound can achieve a bandwidth of 16.2 Gb/s, and the bandwidth more than doubles when the ring buffer size is augmented from 32 KB to 256 KB.

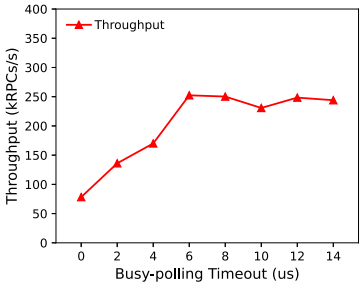


Fig. 13. Throughput by varying the busy-polling timeout.

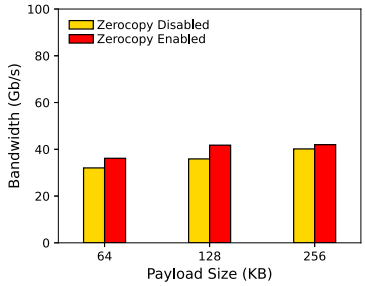


Fig. 14. Bandwidth when disabling and enabling zero-copy.

In practice, we opt to set the ring buffer size at 4 MB, aligning with gRPC's default maximum RPC size. This configuration provides a reasonable upper limit for most use cases. Furthermore, users have the flexibility to adjust the buffer size to better align with their specific workloads. Our intuitive approach could involve users specifying the maximum expected RPC size. Upon connection establishment, clients could progressively send RPCs of increasing sizes, gathering performance data to iteratively determine the most efficient ring buffer size. While an auto-tuning method to determine an optimal ring buffer size is desirable, we acknowledge its significance and complexity, and differ it to future work.

Varying Busy Polling Time: A small timeout value for BPEV ($bp\_timeout$ in Algorithm 2) does not improve performance as it doesn't allow sufficient time for new messages to arrive. Conversely, a large timeout value will degenerate BPEV into BPTS, negating its intended advantages. This underscores the timeout setting is essential to effectively utilize both busy-polling and event-based waiting without performance compromise.

Fig. 13 shows the throughput of RR-Compound for one client. When the timeout is zero, BPEV behaves as an event-based method, completely depending on the kernel to wake up working threads. As a result, the throughput is only about 78 K RPCs/s, and the median latency is about 17.07 μs. With the timeout increase, the throughput climbs up and stays stable at about 252 K RPCs/s when the timeout is greater than 6 μs. Notably, the median latency in this scenario is approximately 6.80 μs, aligning closely with the given timeout. This observation suggests a practical guideline for selecting an appropriate timeout: it should be marginally longer than the typical round-trip time (RTT) of RPCs. By following this principle, BPEV can effectively leverage busy-polling to bypass the costly $epoll\_wait$ process while still preserving CPU resources.

C. Effectiveness of Zero-Copy

To evaluate the effectiveness of the zero-copy optimization, we employ varying payload sizes. To ensure a clear understanding of the specific benefits introduced by the zero-copy optimization, we opt to measure bandwidth with a single client, as multiple clients could easily saturate the bandwidth capacity of hardware (RNIC), obscuring the benefits by the zero-copy technique.

Fig. 14 shows the bandwidth of RR-Compound under different payload sizes. With a 64 KB payload, RR-Compound achieves a bandwidth of approximately 32 Gb/s without the zero-copy optimization. When activating zero-copy, the bandwidth increases to 36 Gb/s, marking an improvement of 12.5%. For a larger payload of 128 KB, the zero-copy optimization yields a bandwidth improvement of 16.4%. However, as shown in the figure, the performance gains become less significant with a 256 KB payload. As the payload size increases, a significant portion of the time is spent on serialization/deserialization and memory allocation, amortizing the memory copy cost. As a result, with the 256 KB payload, the zero-copy optimization can only improve the bandwidth by 4.5%. The evaluation across different payload sizes provides insights into the varying impact of zero-copy optimization, indicating its effectiveness in certain scenarios and diminishing returns in others.

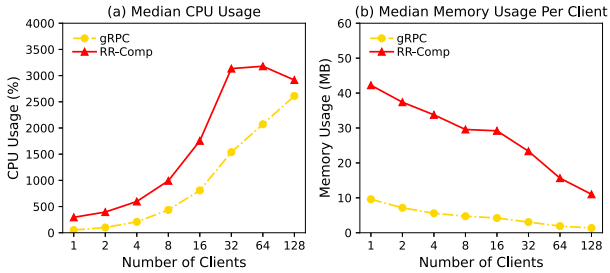


Fig. 15. CPU and memory utilization of RR-Compound compared to gRPC under heavy load.

D. Resource Utilization

Fusing RDMA into gRPC significantly enhances performance but comes with increased resource utilization. Regarding CPU usage, the BPEV design requires a group of polling threads running in the background to detect incoming messages. Regarding memory overhead, RR-Compound introduces a fixed-sized ring buffer as a receive buffer and a send buffer with half the receiver buffer size [15]. For n connections with ring buffers in size c bytes, RR-Compound consumes $n \times 1.5 \times c$ bytes on the RPC server, where $c = 4$ MB by default.

Fig. 15 shows the resource utilization of RR-Compound compared to gRPC under heavy load, as per the configurations listed in Section V-F. RR-Compound exhibits higher CPU utilization than gRPC, ranging from 1.12x to 5.80x. This is reasonable given higher throughput, demanding a proportionally higher utilization of CPU resources. Fig. 15(a) shows that the CPU

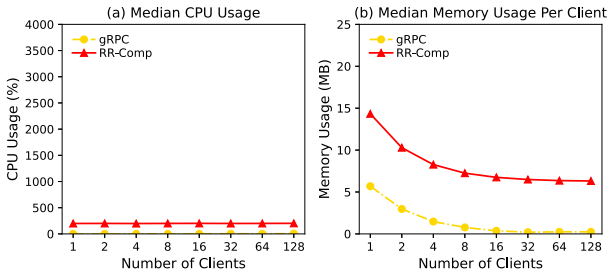


Fig. 16. CPU and memory utilization of RR-Compound compared to gRPC under low load.

utilization of RR-Compound drops slightly for 128 clients. This potentially attributes to gRPC's internal locks contention under high loads, causing certain threads to enter sleep mode. Fig. 15(b) compares the median memory usage per client. Owing to higher throughput in RR-Compound, it requires more temporary memory for in-flight RPCs, resulting in 4.4x to 8.2x higher memory consumption than gRPC. As the client count increases, the per-client memory consumption decreases. This trend can be attributed to two main factors: static memory allocation necessary for gRPC's operation, remaining constant regardless of client numbers, and linear increase in send/receive buffer size with the number of clients. Consequently, adding more clients amortizes the constant memory, resulting in a decreasing memory usage per client in RR-Compound.

Fig. 16 shows the resource utilization of RR-Compound under low load, by setting an RPC rate limiter, as configured in Section V-C. In this settting, RR-Compound consistently utilizes 200% CPU due to the allocation of two polling threads. In contrast, gRPC, being purely event-based, leads to threads entering sleep mode while waiting for messages, resulting in a lower CPU utilization (4%). Fig. 16(b) shows the memory utilization of RR-Compound. For one client, gRPC takes 5.6 MB of memory, and RR-Compound takes 14.3 MB. As the number of clients increases, the amortized memory of RR-Compound decreases. With 16 clients, the memory consumption per client is about 6.7 MB, comprising 4 MB for the receive buffer and 2 MB for the send buffer. This also aligns with our analysis above.

Modern data center servers are typically equipped with TBs of memory and tens to hundreds of CPU cores. While RR-Compound exhibits higher resource utilization, it can be justified by achieving higher throughput and lower latency. A possible improvement involves putting polling threads into sleep mode during extended periods of inactivity. We mark it as a potential part for future research.

E. gRPC versus RR-Compound by Key-Value Store

Key-value store is an RPC favorable workload for its small size of payload and high communication frequency. In this section, we use Yahoo! Cloud Serving Benchmark (YCSB) [16] to evaluate the performance of RR-Compound against gRPC over IPoIB. Six core workloads of YCSB are shown in Table IV.

Since YCSB is written in Java, we implemented a gRPC-binding that invokes the gRPC client with Java Native Interface (JNI). For 4 to 32 clients, we fixed the number of threads to the CPU cores (40). For 64 to 128 clients, we fixed the number of threads to 64. We tend to oversubscribe CPUs to gain a higher throughput because the CPU will be yielded when accessing the database. The completion queues are set to the number of threads. We used Jungle [18] to store the key-value-pairs. It is based on a combined index of LSM-Tree [24] and copy-on-write (append-only) B+ tree [25]. Jungle is used as a storage engine for eBay production systems. We use Yahoo! Cloud Serving Benchmark (YCSB) [16] to evaluate the performance of RR-Compound. During the execution of the workloads, 20 K records are inserted into the database and 20 K operations are executed for each type of workload. Other YCSB settings remain as the default except for the record count and operation count.

TABLE IV
SIX CORE WORKLOADS OF YCSB

Name	Description	Ratio
Workload A	Update heavy	50% Read, 50% Update
Workload B	Read mostly	95% Read, 5% Update
Workload C	Read only	100% Read
Workload D	Read latest inserts	95% Read, 5% Insert
Workload E	Short ranges scan	95% Scan, 5% Insert
Workload F	Read-modify-write	50% Read, 50% Read-modify-write

TABLE V
YCSB SPEEDUPS OF RR-COMPOUND OVER GRPC

Name	Speedups over gRPC
Workload A	1.63
Workload B	2.24
Workload C	2.04
Workload D	2.33
Workload E	1.97
Workload F	2.35

Fig. 17 shows the aggregated throughput results by increasing the number of clients up to 128. The experimental results consistently show that RR-Compound outperforms gRPC running over IPoIB for all workloads, which confirms our R&D efforts discussed in the previous sections. Table V summarizes the throughput speedups of RR-Compound, which range from 1.63x to 2.35x.

RR-Compound shows its high scalability over gRPC to handle a large volume of connections. For *Workload B and C*, read-intensive workloads, tend to exhibit the peak performance for both gRPC and RR-Compound due to reading operation in the database taking a small amount of time. For *Workload C*, when increasing the number of clients from 4 to 64, the throughput of gRPC increased from 28kOP/s to 303kOPs/s. For 128 clients, the throughput stays the same as 64 clients, but the number of clients doubled. In contrast, RR-Compound scales well. Its throughput increased from 49kOPs/s for four clients all the way up to 535kOPs/s for 128 clients. For *Workload A, B, E, and F*, the throughput of gRPC even dropped when increasing the number of clients from 64 to 128. This behavior could be blamed on gRPC's TCP communication channel, which relies on the kernel. Under high pressure, the performance deteriorates due to frequent system calls, interrupts, overloaded CPU cores, etc.

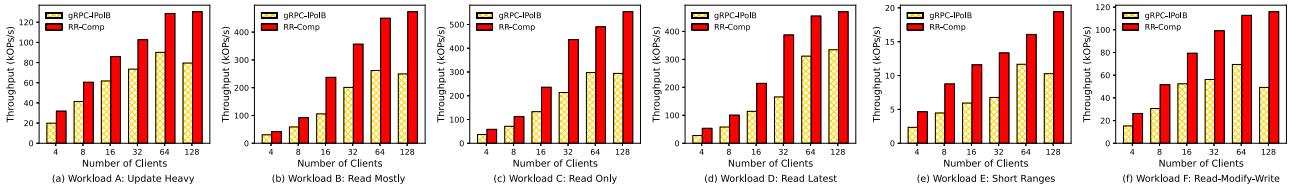


Fig. 17. YCSB throughput.

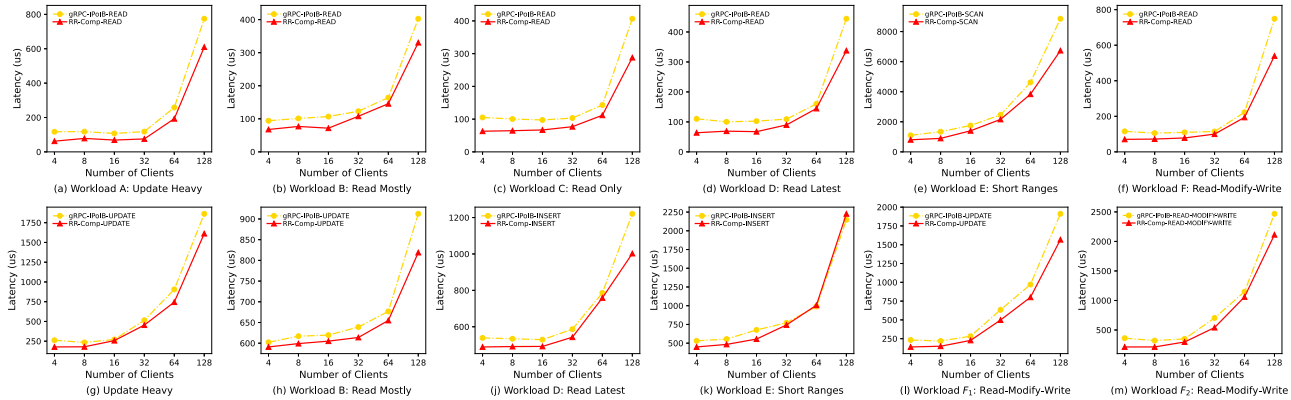


Fig. 18. YCSB latency.

We have also looked into the latency changes in Fig. 18. Again, gRPC has the highest latency almost for all operations no matter what the number of clients is. Fig. 18(a)–(f) shows that the read latency for all workloads stay in their flat curves from 4 to 32 clients. This is because the server still has enough CPU cores to handle the requests well. By increasing the number of clients from 32 to 64, the read latency of all workloads increases. It is interesting that for *Workload E* when the number of clients is more than 64, the latency of insert operation for RR-Compound is higher than gRPC (Fig. 18(k)) - the only case that RR-Compound is inferior to gRPC. We investigated that *Workload E* consists of 95% scan operation and only 5% insert operation. The scan operation fetches many values from the database for a single key, so it could easily saturate the CPU on the server side. Due to the high performance of RR-Compound, the high frequent scan operation could squeeze out CPU cores, leaving a small amount of CPU resources for insert operations. Thus, the latency of the insertion is pulled up. When the number of clients increases from 64 to 128, the read and write latency for all workloads significantly increase because the server resources are exhausted. Nevertheless, the latency of RR-Compound is still relatively low, achieving latency reduction from 10.25% to 46.92% compared to IPoIB.

F. gRPC versus RR-Compound by TensorFlow

TensorFlow [26] is an open-source Machine Learning framework designed by the Google Brain team. It provides an API for distributed training across multiple compute nodes. During the execution, each node communicates via gRPC to exchange training data updates with other nodes. In this section, we evaluate the performance of TensorFlow running on both gRPC and RR-Compound by testing several DNNs from TensorFlow CNN benchmark [27]. This benchmark generates synthetic image data and measures the throughput performance on the number of images processed per second. The TensorFlow version we used is 2.8.0. We set TensorFlow in Parameter Server mode and use up to twelve nodes for distributed training. Different from Table II, we use another group of machines, which equip with 2xIntel(R) Xeon(R) CPU E5-2680 v4, Mellanox ConnectX-5 NIC, and NVIDIA Tesla V100 GPUs. We run the benchmark on different DNN models. Like other parallel processing applications, the TensorFlow benchmark's execution interleaves between data communication stages and local computation stages. This set of benchmark programs are highly computing intensive. In other words, the data communication time spent in the total execution time for each benchmark program is relatively low. Since RR-Compound focuses on improving the data communication for gRPC, it may not significantly improve overall performance compared with that of gRPC. However, our RR-Compound achieved comparable performances with a customized and RDMA-based TensorFlow framework [20].

Inception4: Fig. 19(a), (b), and (c) compares the throughput performance between RR-Compound and gRPC on Inception4 [28]. We measured the data communication time in the total program execution time, which ranges from 14% to 49%. For Inception4 model, RR-Compound improves TensorFlow's performance over gRPC on 4 nodes, 8 nodes, and 12 nodes by 9.8%, 14.7% and 14.8%, respectively.

AlexNet: Fig. 19(d), (e), and (f) compares the throughput performance between RR-Compound and gRPC on AlexNet [29].

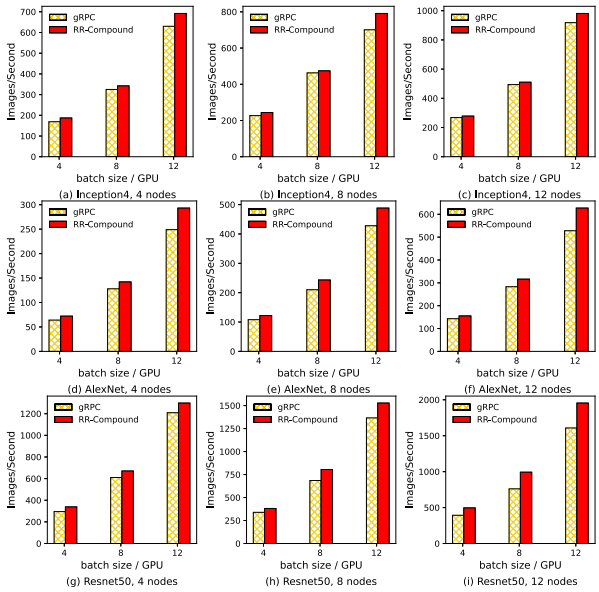


Fig. 19. TensorFlow performance.

We measured the data communication time in the total program execution time, which ranges from 8% to 40%. For Alexnet model, RR-Compound improves TensorFlow's performance over gRPC on 4 nodes, 8 nodes, and 12 nodes by 17.6%, 18.9% and 20.3%, respectively.

Resnet50: Fig. 19(g), (h), and (i) compares the throughput performance between RR-Compound and gRPC on Resnet50 [30]. The measured data communication time in the total program execution time ranges from 9% to 60%. For Resnet50 model, RR-Compound improves TensorFlow's performance over gRPC on 4 nodes, 8 nodes, and 12 nodes by 14.5%, 16.8% and 29.6%, respectively.

We achieved a speedup of TensorFlow up to 1.3x, while the speedup of YCSB experiments was up to 2.35x. The performance gap comes from different data communication ratios in the total execution times in TensorFlow and YCSB. For example, the data communication time ratio in YCSB is at least 77%, whereas the ratio in TensorFlow is at most 60%. In several cases, the ratio is under 10%. Being consistent with Amdahl's Law, we have shown that RR-Compound can make a significant performance improvement over gRPC if we have a sufficiently high ratio of data communication time in the total execution time for an application program because the RDMA with low latency and high throughput plays an effective role under this condition.

In RR-Compound, RDMA-fused gRPC is a separate module. TCP-based gRPC is still available, providing an option for highly computing-intensive applications with low data communication ratios.

VI. Related Work

Numerous research efforts have leveraged RDMA in distributed systems and applications, including distributed key-value stores [31], [32], [33], [34], [35], B-trees [36], [37], R-trees [38], [39], file systems, database and transactional systems [40], [41], [42], [43], [44], [45], and deep learning systems [46]. These approaches require developers to explicitly utilize RDMA interfaces to rewrite systems and applications, which significantly increases the human effort involved.

Concealing the complexities of RDMA behind a more general-purpose programming interface is highly desirable. Several customized RPC systems, including FaRM [15], Herd RPCs [47], FaSST [48], PRISM [49], StRoM [50], Octopus [51], ScaleRPC [22], RFP [52], and eRPC [21], utilize RDMA for RPC data transport. However, none of these systems offer the extensive service scope of gRPC, which supports a variety of programming languages and is compatible with different operating systems.

In practice, an RPC system often requires certain advanced management features, such as authentication, flow control, and live upgrades, demanding a proxy service support. mRPC [13] is a flexible architecture by providing RPC as a system service instead of a library integrated into each application. Moreover, mRPC supports various transport layers, such as TCP and RDMA, through the dynamical loading of libraries. It exhibits impressive performance, particularly with RDMA for a limited number of clients. Compared to mRPC, our experimental results demonstrate that RR-compound exhibits better scalability.

There are also several other studies that have incorporated RDMA into more conventional RPC systems like Hadoop RPC, gRPC, and Thrift [20], [53], [54]. However, these implementations have omitted some features and benefits of traditional RPC systems. For instance, AR-gRPC bypassed all connection and memory management modules of gRPC, constructing RPC stubs directly atop the RDMA transport layer, which made the TCP transport layer and asynchronous execution of gRPC unavailable.

Several studies focus on addressing the inherent bottlenecks of RDMA, such as significant performance degradation when the number of concurrent connections exceeds the capabilities of RDMA NICs, and when data bursts lead to congestion control issues [55], [56], [57], [58], [59], [60]. These challenges, along with the intricacies of RNIC microarchitecture and performance isolation issues, have been explored in depth [61]. Additionally, solutions to address bandwidth imbalances in heterogeneous RNIC environments have been proposed [62]. These studies necessitate new hardware and operating system features and are orthogonal to the focus of our work in this paper.

Literature [63] explores the utilization of RDMA and multicore to accelerate streaming data processing. A more recent work [9] advocates for customizing the network implementation to be entirely application-specific to minimize unnecessary overhead. Some auto-tuning frameworks, as discussed in [7], [64], adapt different threading models to optimize performance under various workloads. An example is μ Tune, which selects the most suitable combination of synchronous versus asynchronous communication, in-line versus dispatch-based RPC processing, and event versus poll-based message detection based on specific workloads [64]. Our perspective aligns with the belief that incorporating a flexible threading model and adaptive connection management will yield significant benefits for RR-Compound, and we recognize these aspects as part of our future work.

VII. Conclusion

Fusing RDMA into an RPC framework to create a unified system that leverages the benefits of both data communication protocols is a common wisdom. However, the integration process based on a well-established and widely used production system like gRPC presents several complex technical challenges. RR-Compound is a gRPC-based system that effectively incorporates the RDMA protocol, gaining high performance for low latency and high throughput. In order to fuse RDMA seamlessly with gRPC's asynchronous execution framework, we have developed adaptive network connection methods and space-efficient memory management mechanisms. We have made a case for RR-Compound's effectiveness by intensive experiments, particularly for workloads involving frequent data communications. Moreover, users working with highly computationally intensive tasks can choose to use gRPC directly. Our future work is to make efforts to merge RR-Compound to the gRPC ecosystems.

References

[1] J. D. Ambrosia and M. Nowell, "IEEE P802.3df 200 Gb/s, 400 Gb/s, 800 Gb/s, and 1.6 Tb/s ethernet task force," 2022. [Online]. Available: https://www.ieee802.org/3/df/index.html

[2] Y. Lee, H. Al Maruf, M. Chowdhury, A. Cidon, and K. G. Shin, "Hydra: Resilient and highly available remote memory," in *Proc. 20th USENIX Conf. File Storage Technol.*, 2022, pp. 181–198.

[3] H. Al Maruf and M. Chowdhury, "Effectively prefetching remote memory with leap," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 843–857.

[4] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A disseminated, distributed OS for hardware resource disaggregation," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 69–87.

[5] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, Feb. 1984.

[6] N. Dragoni et al., *Microservices: Yesterday, Today, and Tomorrow*. Berlin, Germany: Springer International Publishing, 2017, ch. 12, pp. 195–216.

[7] A. Sriraman and T. F. Wenisch, "μ suite: A benchmark suite for microservices," in *Proc. IEEE Int. Symp. Workload Characterization*, 2018, pp. 1–12.

[8] A. Sriraman, A. Dhanotia, and T. F. Wenisch, "SoftSKU: Optimizing server architectures for microservice diversity, scale," in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 513–526.

[9] X. Zhu et al., "Application defined networks," in *Proc. 22nd ACM Workshop Hot Topics Netw.*, 2023, pp. 87–94.

[10] Google, "gRPC: A high performance, open source universal RPC framework," 2022. [Online]. Available: https://grpc.io/

[11] Google, "Protocol buffers: A language-neutral, platform-neutral extensible mechanism for serializing structured data," 2022. [Online]. Available: https://developers.google.com/protocol-buffers

[12] Google, "Introducing gRPC, a new open source HTTP/2 RPC framework," 2015. [Online]. Available: https://developers.googleblog.com/2015/02/introducing-grpc-new-open-source-http2.html

[13] J. Chen et al., "Remote procedure call as a managed system service," in *Proc. 20th USENIX Symp. Netw. Syst. Des. Implementation*, 2023, pp. 141–159.

[14] P. MacArthur and R. D. Russell, "A performance study to guide RDMA programming decisions," in *Proc. IEEE 14th Int. Conf. High Perform. Comput. Commun. IEEE 9th Int. Conf. Embedded Softw. Syst.*, 2012, pp. 778–785.

[15] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation*, Seattle, WA: USENIX Association, 2014, pp. 401–414. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi'c

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

[17] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, and Z. Chen, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/

[18] J.-S. Ahn, M. A. Qader, W.-H. Kang, H. Nguyen, G. Zhang, and S. Ben-Romdhane, "Jungle: Towards dynamically adjustable key-value store by combining LSM-tree and copy-on-write B-tree," in *Proc. 11th USENIX Workshop Hot Topics Storage File Syst.*, 2019, Art. no. 9.

[19] RDMA core userspace libraries and daemons, 2022. [Online]. Available: https://github.com/linux-rdma/rdma-core

[20] R. Biswas, X. Lu, and D. K. Panda, "Accelerating TensorFlow with adaptive RDMA-based gRPC," in *Proc. IEEE 25th Int. Conf. High Perform. Comput.*, 2018, pp. 2–11.

[21] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter RPCs can be general and fast," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, Boston, MA: USENIX Association, 2019, pp. 1–16. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/kalia

[22] Y. Chen, Y. Lu, and J. Shu, "Scalable RDMA RPC on reliable connection with efficient resource sharing," in *Proc. 14th EuroSys Conf.*, 2019, Art. no. 19.

[23] Benchmarking | gRPC, 2024. [Online]. Available: https://grpc.io/docs/guides/benchmarking/

[24] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, Jun. 1996.

[25] O. Rodeh, "B-trees, shadowing, and clones," *ACM Trans. Storage*, vol. 3, no. 4, pp. 1–27, Feb. 2008.

[26] M. Abadi et al., "TensorFlow: A system for Large-Scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[27] Tensorflow benchmark, 2018. [Online]. Available: https://github.com/tensorflow/benchmarks/

[28] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-ResNet and the impact of residual connections on learning," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 4278–4284.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017.

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[31] J. Huang et al., "High-performance design of HBase with RDMA over InfiniBand," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, 2012, pp. 774–785.

[32] J. Appavoo et al., "Providing a cloud network infrastructure on a supercomputer," in *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput.*, 2010, pp. 385–394.

[33] J. Jose et al., "Scalable memcached design for infiniband clusters using hybrid transports," in *Proc. IEEE/ACM 12th Int. Symp. Cluster Cloud Grid Comput.*, 2012, pp. 236–243.

[34] J. Jose et al., "Memcached design on high performance RDMA capable interconnects," in *Proc. Int. Conf. Parallel Process.*, 2011, pp. 743–752.

[35] C. Mitchell, Y. Geng, and J. Li, "Using One-SidedRDMA reads to build a fast, CPU-EfficientKey-Value store," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 103–114.

[36] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li, "Balancing CPU and network in the cell distributed B-Tree store," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 451–464.

[37] X. Wei, R. Chen, and H. Chen, "Fast RDMA-based ordered Key-Value store using remote learned cache," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 117–135.

[38] M. Xiao, H. Wang, L. Geng, R. Lee, and X. Zhang, "An RDMA-enabled in-memory computing platform for r-tree on clusters," *ACM Trans. Spatial Algorithms Syst.*, vol. 8, no. 2, pp. 1–26, Jun. 2022.

[39] M. Xiao, H. Wang, L. Geng, R. Lee, and X. Zhang, "Catfish: Adaptive RDMA-enabled r-tree for low latency and high throughput," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 164–175.

[40] N. S. Islam et al., "High performance RDMA-based design of HDFS over infiniband," in *Proc. Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 1–12.

[41] J. Wu, P. Wyckoff, and D. Panda, "PVFS over InfiniBand: Design and performance evaluation," in *Proc. Int. Conf. Parallel Process.*, 2003, pp. 125–132.

[42] B. Callaghan, T. Lingutla-Raj, A. Chiu, P. Staubach, and O. Asad, "NFS over RDMA," in *Proc. ACM SIGCOMM Workshop Netw.-I/O Convergence: Exp. Lessons Implic.*, 2003, pp. 196–208.

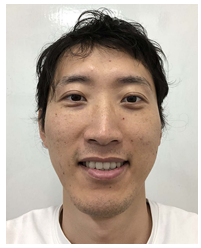
[43] M. DeBergalis et al., "The direct access file system," in *Proc. 2nd USENIX Conf. File Storage Technol.*, 2003, pp. 175–188.

[44] Y. Gao et al., "When cloud storage meets RDMA," in *Proc. 18th USENIX Symp. Netw. Syst. Des. Implementation*, 2021, pp. 519–533.

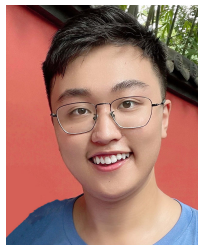
[45] D. Kim et al., "Hyperloop, group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 297–312.
[46] J. Xue, Y. Miao, C. Chen, M. Wu, L. Zhang, and L. Zhou, "Fast distributed deep learning over RDMA," in *Proc. 14th EuroSys Conf.*, 2019, pp. 1–14.
[47] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 437–450.
[48] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, Savannah, GA: USENIX Association, 2016, pp. 185–201. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia
[49] M. Burke et al., "PRISM: Rethinking the RDMA interface for distributed systems," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, New York, NY, USA, 2021, pp. 228–242, doi: 10.1145/3477132.3483587.
[50] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso, "StRoM: Smart remote memory," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, Art. no. 29.
[51] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: An RDMA-enabled distributed persistent memory file system," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 773–785.
[52] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu, "RFP: When RPC is faster than server-bypass with RDMA," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 1–15.
[53] X. Lu et al., "High-performance design of hadoop RPC with RDMA over infiniband," in *Proc. 42nd Int. Conf. Parallel Process.*, 2013, pp. 641–650.
[54] T. Li, H. Shi, and X. Lu, "HatRPC: Hint-accelerated thrift RPC over RDMA," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, New York, NY, USA, 2021, Art. no. 36.
[55] H. N. Schuh, W. Liang, M. G. Liu, J. Nelson, and A. Krishnamurthy, "Xenic: SmartNIC-accelerated distributed transactions," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, 2021, pp. 740–755.
[56] Y. Zhu et al., "Congestion control for large-scale RDMA deployments," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2015, pp. 523–536.
[57] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang, "Socksdirect: Datacenter sockets can be fast and compatible," in *Proc. ACM Special Int. Group Data Commun.*, 2019, pp. 90–103. [Online]. Available: https://www.microsoft.com/en-us/research/publication/socksdirect-datacenter-sockets-can-be-fast-and-compatible/
[58] E. Amaro et al., "Remote memory calls," in *Proc. 19th ACM Workshop Hot Topics Netw.*, 2020, pp. 38–44.
[59] A. Singhvi et al., "1RMA, re-envisioning remote memory access for multi-tenant datacenters," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2020, pp. 708–721.
[60] W. Reda, M. Canini, D. Kosti'c, and S. Peter, "RDMA is turing complete, we just did not know it yet!," in *Proc. 19th USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 71–85.
[61] X. Kong et al., "Understanding RDMA microarchitecture resources for performance isolation," in *Proc. 20th USENIX Symp. Netw. Syst. Des. Implementation*, 2023, pp. 31–48.
[62] Q. Li et al., "Flor: An open high performance RDMA framework over heterogeneous RNICs," in *Proc. 17th USENIX Symp. Operating Syst. Des. Implementation*, 2023, pp. 931–948.
[63] S. Zeuch et al., "Analyzing efficient stream processing on modern hardware," *Proc. VLDB Endowment*, vol. 12, no. 5, pp. 516–530, 2019.
[64] A. Sriraman and T. F. Wenisch, "μTune: Auto-Tuned threading for OLDI microservices," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 177–194.

Liang Geng received the BEng and MEng degrees from Liaoning Technical University and Northeastern University in China, respectively. He is currently working toward the PhD degree with The Ohio State University. His research interests include in computer systems, especially using advanced hardware technology to accelerate spatial data processing.

Hao Wang (Member, IEEE) received the PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences. He is a senior research scientist with the International Digital Economy Academy (IDEA), Shenzhen, China. His research interests include synergistic intersection of computer architecture, distributed and parallel computing, and algorithms for database and blockchain.



Jingsong Meng received the bachelor of engineering (BE) degree from Xi'an Jiaotong University, in China, and the master's degree from Syracuse University. He is currently working toward the PhD degree with The Ohio State University. He is interested in distributed systems.

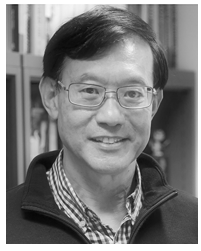


Dayi Fan received the BEng degree from the Southern University of Science and Technology in China. He is currently working toward the PhD degree with The Ohio State University. His research interests include in parallel systems and algorithms, especially for using new hardware to accelerate data-intensive applications.

Sami Ben-Romdhane received the BS degree from Ecole Nationale des Ingénieurs de Tunis, and the master's degree from Ecole Nationale des sciences de l'informatique. He is a vice president and technical fellow with eBay Inc. He has long experience in designing and implementing high-performance/reliable, highly scalable, highly available distributed systems.

Hari Kadayam Pichumani received the BEng degree from the University of Madras. He is a distinguished engineer with eBay, Inc. He has more than 20 years of experience in technical leadership, building architecture from ground up, prototyping, designing, and implementing end-to-end storage systems software.

Vinay Phegade is a principal engineer with eBay, Inc. He is a technical leader with expertise in building AI/ML platforms with security and privacy foundations, leading eBay's AI/ML platform architecture team to deliver advanced capabilities to data scientists at optimized cost. He also has applied hands-on experience in building scalable systems with software/hardware co-design and practical knowledge of training and deploying deep learning models.



Xiaodong Zhang (Fellow, IEEE) received the PhD degree in computer science from the University of Colorado at Boulder, where he was honored with a Distinguished Engineering Alumni Award, in 2011. He is a University distinguished scholar and the Robert M. Critchfield professor in engineering with the Ohio State University. His research interests include on data management in computer and distributed systems. He received the Education Leadership Award from the Lutron Foundation for chairing the Department of Computer Science and Engineering with Ohio State from 2006 to 2018. He is also a fellow of the ACM.