INFOCOM 2024 1570936659

RB^2: Narrow the Gap between RDMA Abstraction and Performance via a Middle Layer

Haifeng Sun[1], Yixuan Tan[1], Yongtong Wu[1], Jiaqi Zhu[1], Qun Huang[1], Xin Yao[2], Gong Zhang[2]
[1]National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University
[2]Huawei Theory Department

***Abstract*—Although the native RDMA interface allows for high throughput and low latency, its low-level abstraction raises significant programming challenges. Consequently, numerous systems encapsulate the RDMA interface into more user-friendly high-level abstractions such as Socket, MPI, and RPC. However, this ease of development often incurs considerable performance degradation. To address this trade-off, this paper introduces RB^2, a high-performance RDMA-based Distributed Ring Buffer (DRB). RB^2 serves as a middle layer that effectively conceals the low-level details of the RDMA interface while also facilitating extension to other high-level abstractions.**

Nonetheless, it is non-trivial for DRBs to preserve the RDMA performance. We optimize the performance of RB^2 in three aspects. First, we perform micro-benchmarks to identify the pointer synchronization methods that are seemingly counter-intuitive but offer optimal performance improvements. Second, we propose an adaptive batching mechanism to alleviate the limitations of conventional fixed batching. Finally, we build an efficient memory subsystem using various optimization techniques. RB^2 outperforms SOTA designs by achieving 2.5$\times$ to 7.5$\times$ throughput while maintaining comparable tail latency for small messages.

I. Introduction

Remote Direct Memory Access (RDMA) has become an essential part of data center infrastructure for high performance. It enables direct access the memory at remote hosts, which bypasses the host CPU, achieving high throughput and low latency. However, there is a fundamental mismatch between the native RDMA abstraction and the requirements of contemporary data center applications. On the one hand, native RDMA provides a low-level abstraction close to hardware primitives. On the other hand, modern data center applications are constructed upon high-level abstraction, such as remote procedure call (RPC) library, message passing interface (MPI) and socket abstraction. As a result, most systems [1]–[24] further encapsulate the RDMA interface, which prevents them from fully exploiting the high throughput and low latency capabilities of RDMA hardware.

We observe that each common high-level abstraction has one or more systems that use a distributed ring buffer (DRB) as the internal implementation. This finding inspires us to design a high-performance RDMA-based DRB, namely RB^2, serving as an infrastructure to support different types of high-level abstraction. For abstraction, DRB is high-level enough to hide the details of native RDMA APIs. At the same time, DRB can be easily re-encapsulated to support more complex abstractions (e.g. socket-like API [1]–[4], RPC library [5]–[13], and MPI library [14]–[20]).

To ensure the high performance of RB^2, we explore the design space of DRBs to integrate the most performance-beneficial design choices. However, compared to classical ring buffers [25], the distributed operations (i.e., pointer synchronization and message transmission) of DRBs raise challenges to preserve the RDMA performance. (1) Pointer synchronization plays a crucial role in DRBs, preventing the sender from overwriting unprocessed messages or the receiver from reading incomplete messages. There are numerous combinations of `head` and `tail` synchronization methods, and the batching mechanism is often used to minimize synchronization overhead. Nevertheless, the intricate interplay between these factors creates a complex influence on overall performance, posing a challenge in designing a high-performance DRB. (2) Although the batching mechanism can also increase the throughput of message transmission, it simultaneously results in idle bandwidth during waiting periods. Also, conventional fixed batching lacks the adaptability to dynamic network scenarios. (3) For memory operations within the message transmission, the introduction of the middle layer inevitably amplifies memory access and cache misses, necessitating efficient memory management to ensure optimal performance.

To this end, we make the following efforts to address the above challenges.

- For pointer synchronization (§IV-A), we conduct a micro-benchmark to compare various combinations and discern the optimal design choices. In particular, RB^2 synchronizes `head` and `tail` via *lazy push* and *write twice*, respectively. For lazy push, the receiver pushes the latest `head` to the sender after it processes a certain number of messages. Write twice assures that the advancement of the `tail` (2^{nd} `WRITE`) only happens after the completion of the message transmission (1^{st} `WRITE`). This design choice appears counterintuitive, requiring two RDMA requests to transfer one message, but it offers optimal performance benefits.
- For batching mechanism (§IV-B), RB^2 proposes *adaptive batching*, consisting of *synchronization-ahead transmission* and *elastic threshold*, to alleviate the limitations of common fixed batching. For synchronization-ahead transmission, RB^2 transmits several small groups of batched mes-

*Qun Huang is the corresponding author.

sages before the pointer synchronization, which exploits idle bandwidth during batching waiting time. For elastic threshold, RB^2 delays the pointer advancement until the completion of the previous pointer synchronization operation, helping stagger busy network links.
- For memory management (§IV-C), RB^2 builds an efficient memory subsystem using many classical optimizations to achieve higher performance. Specifically, RB^2 integrates: (1) a zero-copy mechanism to eliminate the additional memory copy introduced by the middle layer; (2) cache-friendly structures to preserve cache efficiency; (3) huge pages to reduce TLB misses; and (4) NUMA-aware memory allocation to mitigate cross-core overheads.

We build a prototype of RB^2 and compare it with three SOTA RDMA-based DRBs. For small messages (64 B), RB^2 achieves 2.5×-7.5× the throughput of existing DRBs and maintains comparable tail latency. For large messages (1 MB), the throughput of RB^2 is not only 1.8×-8.5× that of classical DRBs, but its tail latency is only 6%-23% of theirs. We further illustrate the ease-of-use benefits of the DRB abstraction by integrating RB^2 with three real applications: a network function, a database system, and an HTTP server. Our experiments demonstrate that with the aid of RB^2, different encapsulations can concentrate on high-level communication semantics and application-related features, thus reducing the burden of performance optimization.

II. Problem

Basic concepts of RDMA. RDMA allows applications to directly access memory on remote hosts. This provides low latency, high throughput, and efficient CPU consumption. To achieve these performance gains, RDMA offloads several layers of the network stack to the RDMA NIC (RNIC). That is, RDMA requests are sent directly to the RNIC bypassing the kernel and are served by the remote RNIC without interrupting the CPU. To utilize the benefits, the host must first register a *memory region* (MR) to execute RDMA requests. Then, the host issues an RDMA request over a *queue pair*, consisting of a *send queue* (SQ) and a *receive queue* (RQ). Each queue pair is associated with a *completion queue* (CQ). By polling the CQ for the presence of the corresponding CQ element (CQE), the host can determine the completion status of an RDMA request. The RNIC receives the RDMA requests via PCIe.

Motivation. The low-level abstraction of RDMA hinders the migration of existing applications to the RDMA environment, despite its potential for superior performance. Specifically, modern distributed applications are developed atop high-level communication abstractions like socket, RPC, and others. Conversely, RDMA provides a low-level abstraction that is close to hardware devices, with various configuration options bewildering to software developers [26]. Therefore, developers have to properly select different RDMA requests and carefully tune configurations to exploit the performance improvement within their specific scenarios.

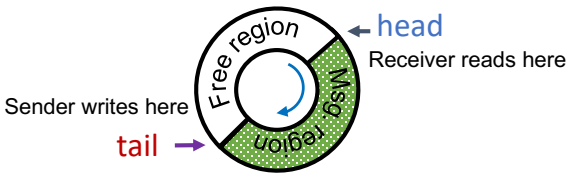


Fig. 1. Logical view of DRBs.

Recent years have witnessed numerous efforts to narrow the gap between the RDMA abstraction and what most applications desire. They encapsulate the RDMA abstraction into high-level ones, including socket-like API [1]–[4], RPC library [5]–[13], MPI library [14]–[20], and other abstractions [21]–[24]. However, there is no one-size-fits-all best approach, as different applications favor distinct abstractions.

Middle layer: distributed ring buffer (DRB). Observing that many high-level encapsulations (e.g., [1], [14], [21], [27], [28]) are built atop the distributed ring buffer (DRB) inspires us to design a high-performance RDMA-based DRB. This DRB serves as a middle layer between high-level abstractions and underlying low-level RDMA interfaces, effectively narrowing this gap. In a nutshell, a DRB is a circular space that can be accessed by a sender and a receiver, as shown in Fig. 1. The receiver maintains a `head` pointer that indicates the address of the next message to be read within the *message region*, and the sender keeps a `tail` pointer that denotes the address where the next message will be written into the *free region*. The advantages of DRBs are threefold. First, the DRB abstraction sufficiently hides the details of native RDMA APIs. Second, it can easily be encapsulated into many application-layer systems, such as socket, MPI library, and RPC library. Finally, the inherent simplicity of DRB enables the easy application of numerous performance optimization techniques. Here, we disregard DRBs with multiple senders and receivers due to their excessive pointer synchronization overheads. Such scenarios can easily be constructed using multiple DRBs with a single sender and receiver.

III. RB^2 Overview

In this paper, we propose an RDMA-based DRB, namely RB^2. To reach optimal performance, we explore the design space of RDMA-based DRBs (see §III-A) to combine the most performance-beneficial design choices for RB^2 (see §III-B). However, it is non-trivial to preserve RDMA performance when applying RDMA to the distributed operations of DRBs. We identify three design challenges during the integration (see §III-C) and present our solutions to address them in §IV.

A. Basic Design Space of RDMA-based DRB

Table I compares different design choices of state-of-the-art RDMA-based DRBs (i.e., [1], [14], [21], [27]). We first discuss the basic design choices (C1-C5) to constitute the fundamental structure of RB^2 in this section.

C1: RDMA verbs. RDMA provides two primary categories of communication verbs (i.e., one-sided and two-sided verbs), bringing multiple different combinations. Specifically, two-sided verbs include `SEND` and `RECV`, which require CPU

Table I. Design space of RDMA-based DRBs.

RDMA-based DRBs	MVAPICH [14]	FaRM [21]	SocksDirect [1]	L5 [27]	**RB^2**
Basic Design Choice					
C1: RDMA verb	`WRITE`	`WRITE`	`WRITE`	`WRITE`	`WRITE`
C2: Transport mode	RC	RC	RC	RC	RC
C3: Data structure	linked list	big array	big array	big array	big array
C4: Data structure maintenance	both	receiver only	both	both	both
C5: Message layout	slot-to-slot	back-to-back	back-to-back	back-to-back	slot-to-slot
Pointer Synchronization and Message Transmission					
Head synchronization	active push	lazy push	lazy push	lazy push	lazy push
Tail synchronization	poll message	poll message	poll RDMA CQ (CQE generated by `WRITE_IMM`)	write twice	write twice
Batching of sending messages			✔		adaptive batching
Memory Management					
Zero copy			socket-level		✔
Cache-friendly structure					✔
NUMA-aware allocation					✔
Huge page		✔			✔

involvement at both the sender and receiver. Two-sided verbs require flow control to ensure that the `RECV` verbs are always posted when a message comes. One-sided verbs include `WRITE`, `READ`, and atomic operations, which are executed by the RNIC bypassing the remote CPU. Thus, one-sided verbs offer lower latency and higher throughput than two-sided verbs [24]. Some systems employ both verb types for different messages to combine their advantages (e.g., [9], [11], [13], [14]), although this also introduces the shortcomings of both verb types. Alternatively, some systems only use one type of RDMA verbs for communication (e.g., [1], [5], [6], [12], [21], [29]), simplifying the communication logic.

C2: RDMA transport modes. RDMA provides different transport modes for reliability and scalability concerns, including reliable connection (RC), unreliable connection (UC), and unreliable datagram (UD). RC assures reliable packet delivery but suffers poor scalability when the number of connections increases. UC only supports one-to-one unreliable connections and does not support the `READ` verb. Conversely, UD can communicate with multiple remote hosts using a single queue pair, reducing the memory overhead of RNICs and thereby enhancing scalability. However, UD demands applications to (i) only use two-sided verbs with high CPU cost and (ii) manage re-transmissions and congestion control. Most previous studies select RDMA transport modes between RC (e.g., [1], [10], [12], [21]) and UD (e.g., [6], [11], [18], [22]) based on their scalability requirements.

C3: Data structure. The data structures for building DRBs include linked lists and arrays. For linked list implementation, its benefit is that there is no need to handle boundary cases in array-based realization (i.e. a message crosses separate chunks at both the tail and head of the array). However, in contrast to the array-based realization, the spatial locality of a linked list is poor, which can easily cause TLB misses. Therefore, most DRBs (e.g., [1], [21], [27]) adopt a big array to realize the ring buffer, and only MVAPICH [14] uses a linked list.

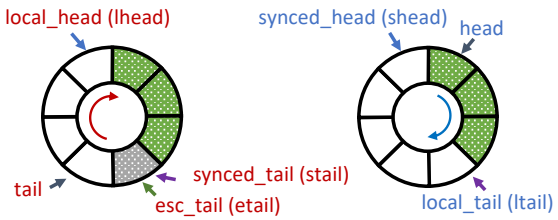
C4: Data structure maintenance. Since DRB is a distributed abstraction, it is significant to decide the location of the DRB structure. In particular, the DRB structure can be maintained (1) solely on the receiver side or (ii) on both the sender and receiver sides. In the first case (e.g., [21]), the sender directly `WRITE`s the message to the remote buffer. This blocks the sender when writing messages to the remote buffer and precludes the batching of multiple small messages for performance enhancement. In the second case (e.g., [1], [14], [27]), the sender maintains an additional copy of the ring buffer. This enables the execution of non-blocking operations and facilitates higher throughput by leveraging batching.

C5: Message layout. The layout of messages within the DRB tradeoffs between memory utilization and cache-friendly access. There are two types of message layouts: *back-to-back* and *slot-to-slot*. For back-to-back layout (e.g., [1], [21], [27]), messages are stored continuously, which maximizes memory utilization. In contrast, for slot-to-slot layout (e.g., [14]), the ring buffer memory is segmented into several fixed-length slots. Each slot can store only one message, but one message may span multiple continuous slots. In this way, the ring buffer can ensure cache-friendly access at the expense of lower memory utilization due to memory fragmentation.

B. Basic Design Choices of RB^2

RDMA-related design choices. We adopt the one-sided verb `WRITE` (C1) over RC (C2) for the following three reasons. First, considering performance, we select one-sided verbs (unsupported in UD) for better performance. Second, in terms of abstraction, our objective is to preserve the simplicity of the DRB abstraction and avoid introducing additional modules (e.g., reliability guarantees). Finally, for scalability, recent advancements in both software [5], [29] and hardware [30] have demonstrated that RC can also achieve high scalability.

Basic structure. We prioritize performance factors when making design decisions for the basic structure. Fig. 2 shows the layout of RB^2. RB^2 maintains two big circular arrays (C3) to maintain spatial locality, one for the receiver and the other for the sender (C4). This arrangement conveniently facilitates the application of a batching mechanism for enhanced throughput. We divide the buffers into N continuous slots of equal size (C5) to align the cache line. Both buffers are registered as MR for remote RDMA operations.



(a) sender (b) receiver

Fig. 2. RB^2 overview.

Pointers. Fig. 2 shows that the sender manages four pointers and the receiver maintains three pointers in RB^2. Every pointer points to a concrete slot, which can only be advanced. The definitions of `head` and `tail` pointers remain consistent with those in the logical view (see §II), pointing to the next slot to be read and written, respectively. We define "`head=tail`" as the empty status and "`(tail+1)%`N`=head`" as the full status. The `local_head` and `local_tail` pointers are caches of `head` and `tail` on the opposite side. Here, the pointer synchronization in RB^2 indicates updating the `local_head`/`local_tail` by the remote latest `head`/`tail`. They can avoid frequent pointer synchronization every time the sender writes or the receiver reads a message. The slots between the `tail` and `local_head` are the free region of the sender. The slots between the `head` and `local_tail` are the message region of the receiver.

Basic batching. The `synced_head` and `synced_tail` pointers are used to apply batching mechanism in RB^2. The slots between `synced_tail` and `tail` are batched messages. When the number of batched messages meets the threshold, the sender transfers all the batched messages to the receiver, synchronizes the latest `tail` to the remote `local_tail`, and moves the `synced_tail` to the current `tail`. Likewise, the slots between `synced_head` and `head` are processed messages. After processing a certain number of messages, RB^2 synchronizes the latest `head` to the remote `local_head` and points the `synced_head` to the current `head`. We discuss the usage of `esc_tail` in the adaptive batching of RB^2 (see §IV-B).

C. Challenges

Although we make basic design choices for RB^2 that are optimal for performance in §III-B, the most significant performance bottleneck of DRBs stems from distributed operations (i.e., pointer synchronization and message transmission). We discuss three challenges to preserve the performance benefits of RDMA as much as possible (one for pointer synchronization and two for message transmission).

Challenge 1: Pointer synchronization. Pointer synchronization is essential for DRBs to function properly and has profound implications for performance. In particular, `head` synchronization releases the processed message region to the free region, which is critical to avoid blocking the sender. Moreover, `tail` synchronization must occur promptly after message transmission, assuring that the receiver can read the complete message immediately. There are various methods to synchronize pointers (details in §IV-A). However, the influence of different combinations of `head` and `tail` synchronization methods on the overall performance of the DRB is unclear. In addition, RB^2 utilizes the batching mechanism to avoid frequent pointer synchronization (see §III-B). The integration with different synchronization combinations and the batching mechanism makes this design even more challenging.

Challenge 2: Batching mechanism. Although batching can increase throughput and reduce pointer synchronization frequency, the common fixed batching mechanism still imposes adverse effects on performance. First, the bandwidth resource between the sender and receiver is underutilized due to the waiting time for the number of batched messages to reach the threshold. Second, the fixed batching threshold cannot adapt to dynamic conditions (e.g., network congestion).

Challenge 3: Memory management. RB^2 raises challenges for memory management as a high-performance middleware. That is, the lifecycle of a message may span multiple threads, including RB^2, upper-layer encapsulation, and the application. On the one hand, RB^2 incurs frequent inter-thread data transfer, which rarely occurs in previous network stacks. On the other hand, the multi-core execution amplifies the cache misses and complicates NUMA allocation.

IV. RB^2 Design

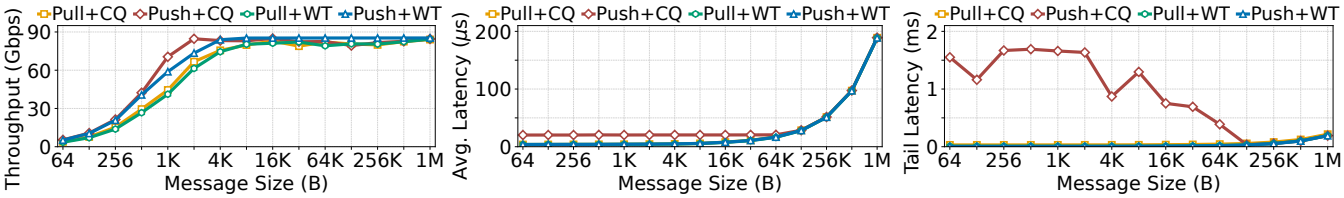
We elaborate on the key designs of RB^2 in this section. For Challenge 1, we perform a micro-benchmark, comparing different combinations of pointer synchronization methods to discern the optimal design in §IV-A. For Challenge 2, we propose an adaptive batching mechanism in §IV-B. For Challenge 3, we adopt many optimizations to build an efficient memory subsystem in §IV-C.

A. Micro-benchmark for Pointer Synchronization

Revisiting Challenge 1, the integration of different pointer synchronization combinations along with the batching mechanism makes it extremely difficult to identify the optimal solution for the overall performance. To tackle it, we first elaborate on potential combinations and then proceed with a micro-benchmark to figure out the best design.

Pointer synchronization methods. Table I summarizes existing methods to synchronize `head` and `tail`. Here, we detail these methods and explore their integration with batching.

- **Head synchronization.** There are two methods to synchronize the `head`: (1) the sender *pulls* the latest `head` from the receiver; and (2) the receiver *pushes* its `head` to the sender. For batching integration, both methods delay the synchronization operation according to a threshold (e.g., the number of released messages, a time interval, etc.), namely lazy synchronization (e.g., [1], [21], [27]). On the contrary, we refer the basic implementation without batching to active synchronization (e.g., [14]).
- **Tail synchronization.** There are three methods based on the one-sided verb to synchronized `tail`.
  (1) *Poll CQ*: The sender employs the `WRITE_IMM` verb (a variant of `WRITE` that can generate CQE on the receiver)



(a) Throughput. (b) Average latency. (c) 99.9% tail latency.

Fig. 3. Micro-Benchmark for different pointer synchronization combinations with batching mechanism.

to transfer messages, so that the receiver can poll CQ to detect the transmission completion and then updates its `local_tail` (e.g., [1]), referred to "*poll CQ*". For batching integration, the sender sets the immediate value in `WRITE_IMM` to the number of batched messages, allowing the receiver to adjust `local_tail` accordingly. In this method, the receiver needs to read the CQE from CQ for each incoming message and then update its `local_tail`, taking up extra CPU load on the receiver.

(2) *Write twice*: The sender transmits the message content by the first `WRITE` and then synchronizes the latest `tail` to the remote `local_tail` by the second `WRITE`, referred to "*write twice*" (e.g., [27]). The ordered execution of RDMA verbs [31] guarantees that the message transmission ends before the pointer synchronization. For batching integration, the sender transfers multiple batched messages via the first `WRITE` and advances the `tail` via the second `WRITE`. In this method, two `WRITE` requests are issued for each message, which incurs significant bandwidth overhead.

(3) *Poll message*: The receiver confirms the completion of the message transmission by polling a non-zero value at the last byte of the message and then updating its `local_tail`, referred to "*poll message*" (e.g., [14], [21]). Here, the receiver locates the last byte according to the message length which resides at the beginning address of the next arriving message. Note that the receiver must explicitly zero out the memory of processed messages in this method. However, this method relies on a front-to-back write order of the `WRITE` verb, which is not guaranteed in all RNICs. For example, RNICs in [11], [14], [21] use sequential writing, while RNICs in [1], [32], [33] take out-of-order writing. Thus, we do not consider this method in RB^2.

Setup. In summary, there are two methods for synchronizing `head` (lazy push and lazy pull) and `tail` (poll CQ and write twice), respectively. To evaluate the overall influence on the performance, we combine these design choices pairwise, yielding combinations denoted as Pull+CQ, Push+CQ, Pull+WT, and Push+WT. We use an identical testbed as in §V. We divide the ring buffer into 128 slots, in which the slot size is equal to the message size. We vary the message sizes from 64 B to 1 MB and set the batching threshold to 16. We push and pull the `head` via the `WRITE` and `READ` verbs, respectively. For throughput, we send 40 GB of data for stress testing. For latency, we use a ping-pong application to measure the round-trip time as the latency metric. We turn off batching for transferring data messages to achieve the best latency for latency-sensitive scenarios, but we still reserve lazy push/pull for `head` synchronization (no impact on data transmission).

Results for throughput. Fig. 3(a) shows the throughput of different combinations. The Push+WT and Push+CQ achieve higher throughput than the combinations that pull `head`, particularly evident for small messages (256 B∼4 KB). It should be noted that the throughput for 64 B messages using Push-based methods exceeds that of Pull-based methods by 35.9% to 51.7% (not apparent in the figure). The reason is that the Pull-based methods introduce more control logic to the sender, increasing both CPU overhead and outgoing RDMA requests. Push+CQ and Push+WT achieve roughly equivalent throughput except for 1 KB and 2 KB messages. Thus, we tend to select the combinations that use the push method so far.

Results for latency. Fig. 3(b) and 3(c) illustrate the average latency and 99.9% tail latency, respectively. For average latencies of messages smaller than 32 KB, Push+CQ is slightly higher (10 μs to 16 μs) than the other combinations. For tail latency, Push+CQ is significantly higher and more volatile than the remaining combinations for messages smaller than 128 KB. This is a counterintuitive result as we expect that poll CQ that uses a single verb should outperform write twice which issues two `WRITE` verbs. However, the actual results contradict this expectation. The key reason behind this outcome lies in the extra CPU consumption of the receiver. In particular, given the high PPS (packets per second) rate for small message transmissions, the receiver has to process numerous CQEs per second and must ensure that there are sufficient elements in RQ to process the arriving `WRITE_IMM` requests. As a result, the CQ polling and RQ replenishing incur high CPU overhead on the critical path of message transmission and pointer synchronization, which results in erratic tail latency.

Design choices of RB^2. According to the micro-benchmark, RB^2 selects the lazy push and write twice to synchronize the `head` and `tail`, respectively.

B. Adaptive Batching

To address Challenge 2, RB^2 proposes *adaptive batching*, which consists of *synchronization-ahead transmission* and *elastic threshold*.

Synchronization-ahead transmission. To fulfill the idle bandwidth brought by the batching mechanism, RB^2 proposes synchronization-ahead transmission. Its key idea is to decouple the two `WRITE` verbs of "batched write twice" into *slot transmission* (1^{st} `WRITE`) and *pointer advancement* (2^{nd} `WRITE`), which are guided by different batching thresholds, respectively. Here, we denote the thresholds of pointer advancement and

Algorithm 1 RB^2 workflow.

Buffer: RB_S for the sender, RB_R for the receiver
Sender Pointers: `tail`, `synced_tail`, `esc_tail`, `local_head`.
Receiver Pointers : `head`, `synced_head`, `local_tail`.
Thresholds pointer advancement: α, slot transmission: β, lazy push: γ
Notes:
I. All pointer arithmetics are performed in the context of modulo slot count.
II. RB[a:b] means all slots from a to b, inclusive and clockwise circularly.

```
1: function WRITEMSG(msg)
2:     if (tail+1) % N == local_tail then
3:         return Error-Full-RB
4:     RB_S[tail] = msg
5:     tail++
6:     if tail-synced_tail ≥ α then                    ▷ Pointer Advancement
7:         WRITE RB_S[esc_tail:tail-1] to the same slots in RB_R
8:         esc_tail = tail
9:         synced_tail = tail
10:        Try to poll WRITE CQE of last tail advancement
11:        if Polling Success then                      ▷ elastic threshold
12:            Synchronize tail to local_tail
13:    else if tail-esc_tail ≥ β then                   ▷ Slot transmission
14:        WRITE RB_S[esc_tail:tail-1] to the same slots in RB_R
15:        esc_tail = tail
16:    return WriteSlot SUCCESS

17: function READMSG(void* msg)
18:    if head == local_tail then
19:        return Error-Empty-RB
20:    *msg = RB_R[head]
21:    head++
22:    if head-synced_head ≥ γ then                     ▷ Lazy push
23:        Synchronize head to local_head
24:        synced_head = head
25:    return ReadSlot SUCCESS
```

slot transmission as α and β, respectively. By setting $\alpha > \beta$, RB^2 transfers several groups of small batched messages prior to pointer advancement, effectively utilizing idle bandwidth.

- **Slot transmission:** RB^2 treats the slots between `esc_tail` and `tail-1` as a group of batched messages to be transported. Each time the sender writes a new message, RB^2 checks whether the number of batched slots (`tail` $-$ `esc_tail` $+$ `1`) reaches β. If the condition is met, RB^2 issues a `WRITE` request to transfer the batched slots and subsequently moves `esc_tail` to the current `tail`.
- **Pointer advancement:** RB^2 utilizes `synced_tail` to track the number of transferred messages since last pointer synchronization. Each time the sender writes a new message to RB^2, it checks `tail` $-$ `synced_tail` $\geq \beta$. If the condition holds true, the sender (i) issues a slot transmission to flush the untransmitted batched messages; (ii) `WRITE`s its `tail` to the remote `local_tail`; and (iii) advances `synced_tail` to the slot pointed by the `tail`.

Elastic threshold. RB^2 proposes the *elastic threshold* to adapt to the busy network. Suppose a scenario where RB^2 reaches the threshold of pointer advancement again when the previous `WRITE` to advance `local_tail` is still in flight. This implies that the `WRITE` requests of slot transmission prior to the newly triggered pointer advancement are not yet completed. There are two common reasons for this scenario: (i) the message generation rate is extremely high; and (ii) the network is under congestion. Hence, we utilize this scenario as an indicator of network busyness. RB^2 defers the pointer advancement to continue the slot transmission until the previous pointer advancement is completed. In essence, we temporarily augment the batching threshold of pointer advancement. Our approach, which only generates CQE for `WRITE` of pointer advancement, is more efficient compared to the adaptive batching of SocksDirect [1], where the sender generates CQE for every issued `WRITE` of sending messages to track the number of requests in flight.

Putting it together. RB^2 integrates the above designs into WRITEMSG and READMSG functions (Algorithm 1), which are provided to the sender and receiver, respectively.

- **Sender.** The sender invokes WRITEMSG to write a message. If the ring buffer is full, the write operation fails, and the function instantly returns (lines 2-3). Otherwise, RB^2 copies the message into RB_S[`tail`] (line 4) and advances the `tail` (line 5). Next, RB^2 checks whether RB^2 reaches the pointer advancement threshold (line 6). If so, the sender first triggers a slot transmission for untransmitted messages (lines 7,8), points `synced_tail` to the current `tail` (line 9), and then tries to synchronize the `tail` (lines 10-12). Here, the sender only `WRITE`s its `tail` to the remote `local_tail` when the previous `tail` synchronization is done (i.e., elastic threshold). Otherwise, the sender continues to check whether RB^2 reaches the slot transmission threshold (line 13). If so, the sender `WRITE`s the messages between `esc_tail` and `tail` to the receiver and advances `esc_tail` to the current `tail` (lines 14,15), which is the synchronization-ahead transmission. At last, we return a success signal (line 16).
- **Receiver.** The receiver invokes READMSG to read a message. If the ring buffer is empty, the read operation fails, and the function returns immediately (lines 18,19). Otherwise, we copy the message in RB_R[`head`] to the user buffer (line 20) and then advance `head` (line 21). If the increment from `head` to `synced_head` reaches γ (i.e., the threshold of lazy push), the receiver pushes its `head` pointer to the remote `local_head` and points its `synced_head` to the position of the `head` (lines 22-24). Finally, we return a success signal (line 25).

Example. We provide an example to illustrate the key designs in RB^2. We configure RB^2 as follows: $N = 8$, $\alpha = 4$, $\beta = 2$, $\gamma = 2$. Fig. 4 shows the status of RB^2 after executing a sequence of operations, i.e., write one message, read two messages, and write two messages (see Fig. 2 for the shorthand of the pointer names). We show stable statuses when the current execution of WRITEMSG and READMSG is finished.

- *Slot transmission* (Fig. 4(a) $\rightarrow$ 4(b)): The sender writes a message and advances its `tail`. Then slot transmission is triggered, i.e., two blue slots are transmitted to the receiver ahead of the pointer synchronization.
- *Lazy push* (Fig. 4(b) $\rightarrow$ 4(c)): The receiver reads two messages. The first READMSG triggers lazy push to synchronize the `head`, while the second one only advances the `head`.

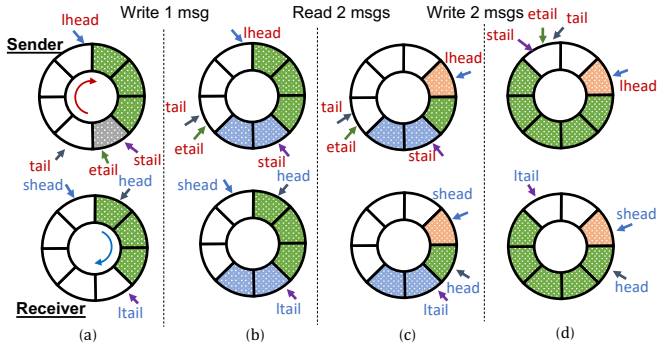


Fig. 4. Examples. (Colored slots are non-empty. Grey slots haven't been sent to the receiver. Blue slots are sent but can't be read due to unsynchronized `local_tail`. Orange slots are processed, but unsynchronized to `local_head`.)

- *Pointer advancement* (Fig. 4(c) $\rightarrow$ 4(d)): The sender writes two messages, which triggers pointer advancement to synchronize the `tail`. Two new slots are sent to the receiver, and `local_tail` is advanced by four. Also, `synced_tail` and `esc_tail` are advanced to the slot pointed by the current `tail`.

C. Memory Management

Zero-copy. RB^2 incorporates zero-copy to eliminate the additional memory copy introduced by the middle layer. For the receiver, users provide a callback function to RB^2 to handle the received data. Each time the receiver successfully reads a new message, this function is automatically invoked. RB^2 facilitates this process by passing the address of the received message to this function as an argument. For the sender, the upper layer application furnishes a function that enables direct insertion of the data into the RB^2 buffer. RB^2 calls this function and passes the address of slots in the local buffer as an argument.

Cache-friendly structure. We carefully tune the structure and variables of RB^2 to maximize the benefits offered by the CPU cache. First, RB^2 holds each pointer in a dedicated cacheline and frequently accesses them to keep them in the cache instead of evicting them to memory. Second, we align the starting address of each message to the cache-line boundaries, by setting the slot length to a multiple of cache-line length (64 B by default). Here, cache-line alignment is significant to the performance. This also explains why we take a slot-to-slot layout to store messages (C5), which can further prevent false cache sharing. For example, if the ring buffer structure is unaligned, a 128 B message crosses three cache lines. When the RNIC receives the `WRITE` request for this message, it launches a DMA to copy this message to the main memory, which invalidates the three cache lines. This inevitably affects the cache performance for the adjacent slots.

NUMA-aware allocation. RB^2 implements NUMA-aware allocation to further mitigate the performance impact of middleware. In particular, our NUMA-aware allocation ensures that (1) the RB^2 process is bound to the CPU core of the NUMA node to which the RNIC belongs; (2) the RB^2 buffer is allocated in the physical memory of the same NUMA node. RB^2 also prompts users to bind the high-level application processes to CPU cores of the same NUMA node. In this way, RB^2 can reduce some memory access latency and eliminate performance fluctuations.

Huge page. RB^2 employs huge pages and allocates buffers on contiguous pinned memory. On the one hand, we aim to reduce the TLB misses. Although the memory access pattern of RB^2 is regular, the patterns of upper-layer applications are unpredictable, which is easy to cause numerous TLB misses. On the other hand, we aim to reduce the cache misses on the RNIC. Here, the RNIC caches the virtual to physical page mappings of the registered MR in its memory translation table. As the on-chip memory of RNIC is small, it easily causes cache misses in the memory translation table. Thus, we use huge pages to reduce the number of memory mapping entries.

V. Evaluation

A. Setup

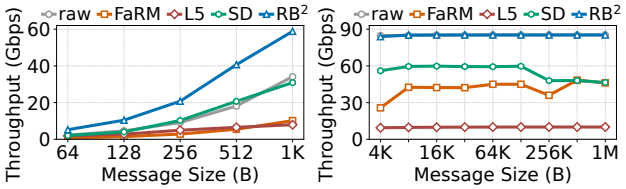
Testbed. We evaluate RB^2 on two servers (one sender and one receiver) with dual 12-core 2.3 GHz Intel Xeon Gold 5118 CPUs, 128 GB memory and a 100 Gbps Mellanox ConnectX-5 NIC. Each server runs on Ubuntu 18.04 with a 4.4.0 kernel. The NICs are supported by driver MLNX OFED 5.1-0.6.6.0 and firmware 16.28.2006. The servers are interconnected with a Mellanox MSB7890-ES2F InfiniBand switch.

Configurations. We compare RB^2 with FaRM [21], SocksDirect [1], and L5 [27]. We set the size of ring buffers as 128 times the message size. For FaRM, its original design cannot work on our testbed because it requires the RNIC with front-to-back write order, which our RNICs cannot support. We realize a modified version to address the issue by combining "poll message" and "write twice", where the first writes the message data and the second writes the message length. For L5, we use its open-source version [34]. For SocksDirect (SD), we implement its DRB based on its published design, in which the maximum number of inflight messages is ten. For RB^2, we use the following configuration: $\alpha = 32, \beta = 16, \gamma = 32$. In addition, we also present the performance of raw `WRITE` by using `perftest` (without other processing logic) to observe the additional processing overhead introduced by RB^2.

Methodology. We measure both throughput and latency. For throughput, we send 40 GB of data for stress testing. For latency, we build a single-thread ping-pong application to avoid overload impacts. We track round-trip time as the latency metric using software timestamps, including average latency and 99.9% tail latency. We perform 100 K rounds of ping-pong to warm up and then 300 K rounds of ping-pong to collect data.

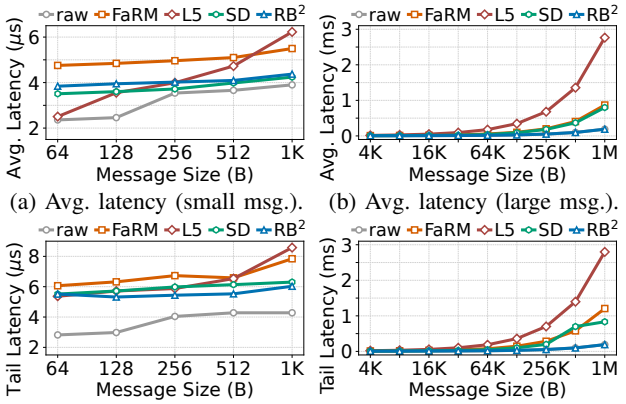
B. Experiments

(Exp#1) Throughput. Fig. 5 compares the throughput under different message sizes (64 B to 1 MB). The throughput of RB^2 is much higher than other DRBs. Compared to raw `WRITE`, RB^2 achieves more than twice throughput for small messages, since our batching for small messages significantly improves



(a) Throughput (small msg.). (b) Throughput (large msg.).

Fig. 5. (Exp#1) Throughput.



(a) Avg. latency (small msg.). (b) Avg. latency (large msg.).

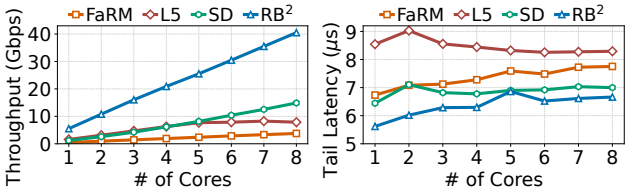
(c) Tail latency (small msg.). (d) Tail latency (large msg.).

Fig. 6. (Exp#2) Latency.

throughput. For large messages, the throughput results of RB^2 and raw `WRITE` are almost the same (close to 90 Gbps), because our zero-copy eliminates the biggest performance bottleneck. SocksDirect achieves similar throughput to raw `WRITE` for small messages due to its batching policy, but its memory copy overhead becomes a bottleneck for large messages. Since L5 and FaRM are not optimized for throughput, they have the lowest throughput for small messages. Even for large messages, the throughput results of L5 are always around 10 Gbps, which is caused by the additional bandwidth overhead of writing twice without batching.

(Exp#2) Latency. Fig. 6 compares the latency under different message sizes (64 B to 1 MB). For small messages, the average latency results of RB^2 acceptably low, which are around 4 μs and very close to that of SocksDirect (Fig. 6(a)). Here, our tail latency results for small messages are 6 μs lower than SocksDirect (Fig. 6(c)). For large messages, RB^2 achieves comparably low latency to the raw `WRITE` and much lower latency than other DRBs due to our zero-copy optimization (Fig. 6(b) and 6(d)). Since L5 is designed to optimize the latency for messages no larger than 64 B, it achieves the best average latency (2.5 μs) for 64 B messages, close to raw `WRITE`. However, its tail latency gets extremely high for larger messages, e.g., 4 ms for messages of 1 MB.

(Exp#3) Multi-core scalability. Fig. 7 evaluates the scalability of RB^2. We bind each thread to a single core and transfer 40 GB of data with 64 B messages. We vary the number of threads from one to eight. Among the four DRBs, RB^2 has the best scalability. As the number of threads increases, the throughput can increase almost linearly, and the tail latency maintains the lowest (smaller than 7 μs). This is because RB^2 adopts NUMA-aware allocation and a cache-friendly structure



(a) Throughput. (b) 99.9% tail latency.

Fig. 7. (Exp#3) Multi-core scalability.

Table II. (Exp#4) Impact of optimizations.

Optimization	Throughput (Gbps)		Latency (μs)	
	small	large	small	large
M1 = Full-optimized RB^2	39.1	85.3	4.0	189
M2 = M1 - all batching policy	**12.8**	85.3	4.1	189
M3 = M1 - sync-ahead transmission	**33.2**	85.3	4.1	189
M4 = M1 - elastic threshold	**24.4**	85.3	4.1	189
M5 = M1 - lazy push	**33.1**	85.3	4.1	189
M6 = M1 - zero-copy	**33.9**	**46.8**	4.0	**713**
M7 = M1 - cache friendliness	**36.0**	**60.3**	4.1	**265**
M8 = M1 - huge page	**39.0**	85.3	4.1	189
M9 = M1 - NUMA-aware allocation	**36.1**	85.3	**4.8**	189

to alleviate the interaction between different ring buffers.

(Exp#4) Impact of optimizations. This experiment examines the benefits of different design choices and optimizations of RB^2. We measure the performance with both small messages (512 B) and large messages (1 MB). We consider RB^2 with full optimizations as the baseline (M1). Then we disable one optimization from the baseline to generate new versions denoted by M2-M8. For the test without NUMA-aware allocation (M8), we manually bind the testing thread to the wrong NUMA node to simulate the worst case.

Table II shows the impact of each optimization. Without all the batching mechanisms (M2), the throughput of small messages drops by 67%. Among each single batching policy (M3-M5), the elastic threshold (M4) contributes the highest throughput benefits. Zero-copy (M6) can eliminate the cost of copying, which is expensive, especially for large messages. Cache-friendly structure (M7) has significant impacts on the throughput. Without it, the throughput of small and large messages decreased by 8% and 29%, respectively. For huge pages (M8), since we do not allocate too many pages in our experiment, its performance benefits are not evident in this experiment. Note that it can benefit upper-layer applications with large memory consumption. NUMA-aware allocation (M9) can improve the performance for small messages because the PPS for small messages is much higher, incurring more memory accesses. In addition, we observe that the main performance bottleneck for transmitting large messages is message copying (M6 and M7). Considering the performance gains and the ample memory resources of modern servers, the increased memory footprint of RB^2, attributed to slot-to-slot layout and cacheline alignment, is regarded as acceptable.

C. Case Study

We evaluate RB^2 under three different applications, covering the most common RDMA scenarios.

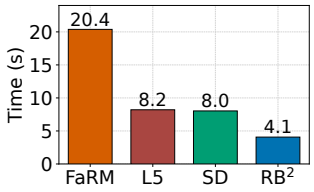


Fig. 8. (Case#1) NF.

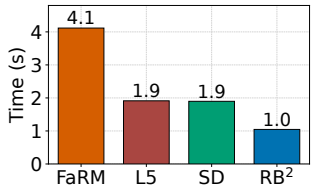
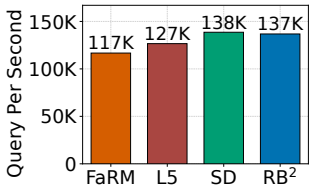
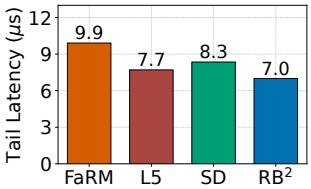


Fig. 9. (Case#2) Database.



(a) Throughput.



(b) 99% Tail latency.

Fig. 10. (Case#3) HTTP server

(Case#1) Network Function. We evaluate the performance of network function (NF) realized by DRB. We can easily implement different NFs based on a DRB because the FIFO behavior of NFs matches that of DRBs. In this case, we implement a measurement NF which counts the traffic. We use public traffic traces CAIDA 2019 [35] for testing. For each packet, we send its flow key, timestamp, sequence, and size (40 bytes in total) to NF using DRB. We replay one-minute traffic (30 M packets) and record the processing time. Fig. 8 shows that RB^2 only requires 4.1 s, which is half the time of L5 and SocksDirect and one-fifth the time of FaRM. The benefits come from the high throughput of RB^2 (see Exp#1).

(Case#2) Database. Fig. 9 compares the performance of database systems realized by different DRBs. We use the YCSB-C (read requests) benchmark [36] for testing. The sizes of the request and response are 16 B and 128 B, respectively. We generate 3 M requests and record the time cost. RB^2 only takes 1 second, which is half the time of L5 and SocksDirect and a quarter of FaRM. Since both sizes of the request and response are small, the results are consistent with Case#1. That is, our batching mechanism contributes the most benefits.

(Case#3) HTTP server. We compare the performance of HTTP servers implemented by different DRBs. A client sends 1 M requests using the `GET`. The server sends back a 1 KB file for each request. The client waits for the file before sending the next request. Since the HTTP server is latency-sensitive, we turn off the batching mechanism. Fig. 10 shows the results. RB^2 achieves a similar QPS (138 K) to SocksDirect (137 K) and the lowest tail latency (7 μs), which is 1.3 μs lower than that of SocksDirect. This is because (1) the socket-level zero copy of SocksDirect improves its QPS, and (2) RB^2 does not batch the generated requests.

VI. Related Works

According to existing systems (see Table I), RB^2 can be further encapsulated into the following high-level abstractions.

Socket-like API. Socket is the most widely used communication abstraction in modern applications. Many encapsulations for socket-like API have been proposed since the inception of RDMA. Sockets Direct Protocol (SDP) [2], [37]–[39] is the first attempt in the industry, which implements the socket semantics over the InfiniBand fabric. Both SDP and UNH EXS [3], [40] adopt a similar design, where control messages are transferred by two-sided RDMA verbs and data messages are transferred via one-sided verbs. RSocket [4] and SocksDirect [1] simplify the above design, which uses one-sided verbs to transfer both data messages and control messages.

MPI library. MPI is the dominant parallel programming model in the high-performance computing area. Since the inception of the first RDMA-based MPI library MVAPICH [14], numerous works emerge to address its scalability issue. ACM [15] proposes adaptive connection management to dynamically control the establishment of RC. MVAPICH-SRQ [16] utilizes shared RQs to achieve scalable buffer management. MVAPICH-UD [18] uses UD to reduce the queue pair memory usage and supports zero-copy protocol in [20]. Open MPI [19] integrates the above optimizations to provide a new open-source library. MVAPICH-Aptus [17] takes a multi-transport design that uses both the RC and UD for messages with different sizes or types.

RPC library. RPC is the cornerstone of modern distributed systems. Existing systems adopt different combinations of RDMA verbs to transmit RPC requests and responses. First, DaRPC [10] and FaSST [6] take two-sided verbs over RC and UD for communication, respectively. Second, ScaleRPC [29] and FLOCK [5] use the one-sided `WRITE`, while RFP [12] applies `WRITE` and `READ` to send requests and fetch the results, respectively. Finally, many systems propose hybrid approaches. HERD [11] issues the requests via `WRITE` over UC and returns the response using two-sided verbs over UD. AR-gRPC [9] and X-RDMA [13] use two-sided verbs for small messages. But for large messages, AR-gRPC takes `READ` while X-RDMA first wakes up the receiver via two-sided verbs and then transmits data messages via one-sided verbs. HatRPC [7] supports five modes to RDMA communication modes, which are selected by its hierarchical hint scheme. FaRM [21] provides an event-based programming model over a distributed shared memory abstraction.

VII. Conclusion

RB^2 is a high-performance RDMA-based DRB that serves as a middle layer between the low-level RDMA interface and the high-level encapsulations or applications. RB^2 explores the design space of RDMA-based DRBs to integrate the design choices that are seemingly counterintuitive but offer optimal performance improvements. RB^2 proposes adaptive batching and builds an efficient memory subsystem to optimize performance. Experiments show the high performance of RB^2 over three state-of-the-art DRBs.

Acknowledgement: We sincerely thank our TPC reviewers and chairs for their constructive comments and insightful suggestions. This work is supported by the National Key R&D Program of China (2023YFB2904600), National Natural Science Foundation of China (62172007), and Beijing Natural Science Foundation (QY23044).

REFERENCES

[1] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang, "Socksdirect: Datacenter sockets can be fast and compatible," in *Proc. of ACM SIGCOMM*, 2019.

[2] D. Goldenberg, M. Kagan, R. Ravid, and M. S. Tsirkin, "Transparently achieving superior socket performance using zero copy socket direct protocol over 20gb/s infiniband links," in *Proc. of IEEE CLUSTER*, 2005.

[3] R. Robert, "The extended sockets interface for accessing rdma hardware," in *Proc. of PDCS*, 2008.

[4] linux man page, "rsocket - rdma socket api," https://linux.die.net/man/7/rsocket, 2022.

[5] S. K. Monga, S. Kashyap, and C. Min, "Birds of a feather flock together: Scaling rdma rpcs with flock," in *Proc. of ACM SOSP*, 2021.

[6] A. Kalia, M. Kaminsky, and D. G. Andersen, "Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs," in *Proc. of USENIX OSDI*, 2016.

[7] T. Li, H. Shi, and X. Lu, "Hatrpc: Hint-accelerated thrift rpc over rdma," in *Proc. of ACM SC*, 2021.

[8] S.-Y. Tsai and Y. Zhang, "Lite kernel rdma support for datacenter applications," in *Proc. of ACM SOSP*, 2017.

[9] R. Biswas, X. Lu, and D. K. Panda, "Accelerating tensorflow with adaptive rdma-based grpc," in *Proc. of IEEE HiPC*, 2018.

[10] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle, "Darpc: Data center rpc," in *Proc. of ACM SOCC*, 2014.

[11] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using rdma efficiently for key-value services," in *Proc. of ACM SIGCOMM*, 2014.

[12] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu, "Rfp: When rpc is faster than server-bypass with rdma," in *Proc. of ACM EuroSys*, 2017.

[13] T. Ma, T. Ma, Z. Song, J. Li, H. Chang, K. Chen, H. Jiang, and Y. Wu, "X-rdma: Effective rdma middleware in large-scale production environments," in *Proc. of IEEE CLUSTER*, 2019.

[14] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, "High performance rdma-based mpi implementation over infiniband," in *Proc. of ACM ICS*, 2003.

[15] W. Yu, Q. Gao, and D. K. Panda, "Adaptive connection management for scalable mpi over infiniband," in *Proc. of IEEE IPDPS*, 2006.

[16] S. Sur, L. Chai, H.-W. Jin, and D. Panda, "Shared receive queue based scalable mpi design for infiniband clusters," in *Proc. of IEEE IPDPS*, 2006.

[17] M. J. Koop, T. Jones, and D. K. Panda, "Mvapich-aptus: Scalable high-performance multi-transport mpi over infiniband," in *Proc. of IEEE IPDPS*, 2008.

[18] M. J. Koop, S. Sur, Q. Gao, and D. K. Panda, "High performance mpi design using unreliable datagram for ultra-scale infiniband clusters," in *Proc. of ACM ICS*, 2007.

[19] G. Shipman, T. Woodall, R. Graham, A. Maccabe, and P. Bridges, "Infiniband scalability in open mpi," in *Proc. of IEEE IPDPS*, 2006.

[20] M. J. Koop, S. Sur, and D. K. Panda, "Zero-copy protocol for mpi using infiniband unreliable datagram," in *Proc. of IEEE CLUSTER*, 2007.

[21] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *Proc. of USENIX NSDI*, 2014.

[22] B. Li, G. Zuo, W. Bai, and L. Zhang, "1pipe: Scalable total order communication in data center networks," in *Proc. of ACM SIGCOMM*, 2021.

[23] E. Amaro, Z. Luo, A. Ousterhout, A. Krishnamurthy, A. Panda, S. Ratnasamy, and S. Shenker, "Remote memory calls," in *Proc. of ACM HotNets*, 2020.

[24] M. Burke, S. Dharanipragada, S. Joyner, A. Szekeres, J. Nelson, I. Zhang, and D. R. K. Ports, "Prism: Rethinking the rdma interface for distributed systems," in *Proc. of ACM SOSP*, 2021.

[25] P. P. C. Lee, T. Bu, and G. Chandranmenon, "A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring," in *Proc. of IEEE IPDPS*, 2010.

[26] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance rdma systems," in *Proc. of USENIX ATC*, 2016.

[27] P. Fent, A. v. Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper, "Low-latency communication for fast dbms using rdma and shared memory," in *Proc. of IEEE ICDE*, 2020.

[28] 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. of IEEE ICDCS*, 2019.

[29] Y. Chen, Y. Lu, and J. Shu, "Scalable rdma rpc on reliable connection with efficient resource sharing," in *Proc. of ACM EuroSys*, 2019.

[30] Z. Wang, L. Luo, Q. Ning, C. Zeng, W. Li, X. Wan, P. Xie, T. Feng, K. Cheng, X. Geng, T. Wang, W. Ling, K. Huo, P. An, K. Ji, S. Zhang, B. Xu, R. Feng, T. Ding, K. Chen, and C. Guo, "SRNIC: A scalable architecture for RDMA NICs," in *Proc. of USENIX NSDI*, 2023.

[31] I. T. Association, "The infiniband architecture specification," https://www.infinibandta.org/ibta-specification/, 2000.

[32] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker, "Revisiting network support for rdma," in *Proc. of ACM SIGCOMM*, 2018.

[33] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen, and T. Moscibroda, "Multi-path transport for rdma in datacenters," in *Proc. of USENIX NSDI*, 2018.

[34] P. Fent and A. v. Renen, "L5rdma," https://github.com/pfent/L5RDMA, 2020.

[35] CAIDA, "The caida anonymized internet traces dataset," https://www.caida.org/catalog/datasets/passive_dataset/, 2019.

[36] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. of ACM SOCC*, 2010.

[37] I. T. Association, "Infiniband architecture specification volume 1, release 1.2: Annex a4: Sockets direct protocol (sdp)," 2004.

[38] D. Goldenberg, M. Kagan, R. Ravid, and M. S. Tsirkin, "Zero copy sockets direct protocol over infiniband-preliminary implementation and performance analysis," in *Proc. of IEEE HOTI*, 2005.

[39] P. Balaji, S. Bhagvat, H.-W. Jin, and D. K. Panda, "Asynchronous zero-copy communication for synchronous sockets in the sockets direct protocol (sdp) over infiniband," in *Proc. of IEEE IPDPS*, 2006.

[40] R. D. Russell, "A general-purpose api for iwarp and infiniband," in *in the First Workshop on DC-CAVES*, 2009.