Achieving Wire-Latency Storage Systems by Exploiting Hardware ACKs

Qing Wang, Jiwu Shu, Jing Wang, and Yuhao Zhang, *Tsinghua University*

https://www.usenix.org/conference/nsdi25/presentation/wang-qing

This paper is included in the
Proceedings of the 22nd USENIX Symposium on
Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Open access to the Proceedings of the
22nd USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

Achieving Wire-Latency Storage Systems by Exploiting Hardware ACKs

Qing Wang, Jiwu Shu*, Jing Wang, and Yuhao Zhang

Tsinghua University

Abstract

We present JUNEBERRY, a low-latency communication framework for storage systems. Different from existing RPC frameworks, JUNEBERRY provides a fast path for storage requests: they can be committed with *a single round trip* and *server CPU bypass*, thus delivering extremely low latency; the execution of these requests is performed asynchronously on the server CPU. JUNEBERRY achieves it by relying on our proposed Ordered Queue abstraction, which exploits NICs' hardware ACKs as commit signals of requests while ensuring linearizability of the whole system. JUNEBERRY also supports durability by placing requests in persistent memory (PM). We implement JUNEBERRY using commodity RDMA NICs and integrate it into two storage systems: Memcached (a widely used in-memory caching system) and PMemKV (a PM-based persistent key-value store). Evaluation shows that compared with RPC, JUNEBERRY can significantly lower their latency under write-intensive workloads.

1 Introduction

Fast Remote Direct Memory Access (RDMA) networks are driving us to build low-latency storage systems [1, 10, 18, 19, 42, 59]. First, RDMA NICs (RNICs) offload network protocol tasks, such as segmentation and checksum, freeing CPU cycles and thus reducing software latency. Second, the RDMA software stack bypasses the OS kernel, thus eschewing latency overhead from context switch and data copying. Finally, the latency can be further reduced by leveraging RDMA one-sided verbs (i.e., RDMA `WRITE`/`READ`) to access remote memory directly; it bypasses the remote CPU, thus detaching the remote CPU's software overhead from the critical path.

Protocol offloads and kernel bypass can bring latency reduction for *every* storage request. However, this is not the case for remote CPU bypass: it is non-trivial to commit a request using one-sided verbs *without increasing the number of round trips*. For simplicity, we term the latency of a single round trip with remote CPU bypass as **wire latency**. Currently, only two use cases can achieve wire latency: simple data access and data replication. In the first case, clients use one-sided verbs to read/write a data block whose remote address is known [18]. However, due to the limited semantics of one-sided verbs, this approach cannot be extended to complex scenarios, such as those containing index traversal and concurrency control, since it will induce excessive round trips [53, 55], offsetting the latency benefits of bypassing remote CPUs. In the data replication case, the leader appends replication requests to followers' replication logs via RDMA `WRITE`, and commits upon the completion of RDMA `WRITE` [9, 18, 58]. Followers execute replication requests lazily in the background. By doing so, the software overhead of followers is removed from the replication critical path.

In this paper, we explore how to achieve wire latency for more general storage requests (beyond simple data access and data replication mentioned before). We begin by identifying what types of storage requests semantically support wire latency. By analyzing, we find that *nil-externalizing* (nilext) requests defined by recent work [21] can meet the requirement of wire latency. Nilext requests do not return execution results or execution errors, i.e., they do not externalize its effects or system state. Hence, the execution of nilext requests can be deferred, making it possible to commit them with server CPU bypass (one requirement of wire latency). Nilext requests are common: for example, `set` requests in Memcached, and `put`/`delete`/`merge` requests in RocksDB [21]. So our goal becomes to make nilext requests wire-latency.

The challenge of achieving wire-latency nilext requests is *maintaining linearizability in the presence of multiple clients*. Specifically, we need to order (nilext and non-nilext) requests issued by different clients for linearizability, and ensure that nilext requests are committed with server CPU bypass. However, RDMA one-sided verbs lack the ability of inter-client coordination, since clients must specify the destination addresses of sent data. This restricts RDMA `WRITE` to achieve wire latency only for single-client scenarios (e.g., replication case where only one leader writes log entries to followers).

We propose Ordered Queue (OQ) abstraction to address the challenge. It is based on our new insight: *the property of remote CPU bypass results from the ability of RNICs to generate hardware ACKs, independent of the type of RDMA verbs*. Specifically, as long as the RNIC supports reliable data transfer, it will generate hardware ACKs, which we can use as the commit signals of nilext requests, to achieve remote CPU bypass. Inspired by the insight, OQ abstraction leverages receive queue, a key structure of RDMA two-sided verbs, to achieve wire-latency nilext requests. It specifies how the CPU and RNIC separately interact with the receive queue using two rules : 1) NIC rule: the server-side RNIC allocates buffers from the receive queue *in order*, accommodating incoming requests, as well as generates hardware ACKs; 2) CPU rule: the server CPU executes requests in the receive queue *in order*

*Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn)

and returns software responses for non-nilext requests. By using the OQ abstraction, storage systems will generate a linearizable history that follows the positions of buffers in the receive queue, while enabling wire-latency nilext requests.

We build JUNEBERRY, a communication framework that implements the OQ abstraction using commodity RDMA NICs. In JUNEBERRY, clients can mark a request as nilext and use the hardware ACK from server-side RNIC as its commit signal, to make nilext requests wire-latency. The OQ abstraction in JUNEBERRY is centered on an RDMA hardware component called *shared receive queue* (SRQ) [12]. SRQ inherently enforces the NIC rule: RNICs place requests from different clients in an SRQ in order [12]. To enforce the CPU rule and support multiple CPU cores, JUNEBERRY adopts a bitmap-assisted, in-order execution mechanism, and partitions requests into multiple SRQs according to data affinity.

The implementation of the OQ in JUNEBERRY brings two performance issues. First, work queue elements or WQEs (i.e., descriptors) in an SRQ are connected using a link list, not as an array as in a standard receive queue. As a result, the RNIC must fetch them in a pointer-chasing manner, inducing excessive DMA operations. Second, the JUNEBERRY server creates a large number of connections and the RNIC cannot cache their states due to its size-limited SRAM, thus leading to performance degradation. To address the first issue, JUNEBERRY allows a single buffer to accommodate multiple requests by leveraging the multi-packet features in RNICs [56]. Hence, a WQE fetched by one DMA operation can serve multiple requests. To address the second issue, we design a client-side delegation mechanism that supports connection sharing between client threads, reducing the number of connections.

To enable JUNEBERRY to be integrated into storage systems that are crash-consistent, we further add durability support for JUNEBERRY. This is achieved by leveraging emerging persistent memory (PM) technologies. Specifically, JUNEBERRY places SRQ buffers in PM and makes the hardware ACK have persistence guarantees. Moreover, we design a recovery procedure that scans PM-resident SRQs to identify valid requests and then restores storage systems to a consistent state.

We augment two storage systems with JUNEBERRY: Memcached [6], a widely used in-memory caching system, and PMemKV [7], a PM-based persistent key-value store. We evaluate them in a cluster of 9 machines. For Memcached, we run representative traces from Twitter's workloads [63]. Compared with RPC, JUNEBERRY reduces median latency and P99 latency by up to 90.70% and 45.65%, respectively. We also compare JUNEBERRY with a software approach that returns responses of nilext requests before execution: JUNEBERRY still reduces median/P99 latency by 83.77%/42.86%. This is because by leveraging hardware ACKs, JUNEBERRY eliminates any server-side software delay from the critical path of nilext requests, including software queueing. For PMemKV, we run YCSB-like workloads. Under write-intensive workloads (i.e., 50% `put`), JUNEBERRY reduces median latency by 40.83%. Under read-intensive workloads (i.e., 5% `put`), JUNEBERRY has 1.19× (1.6μs) higher median latency, which results from the overhead of supporting durability, including disabled DDIO and PM-resident SRQs (Optane PM [5] in our evaluation).

In summary, we make the following contributions.

- Ordered Queue (OQ) abstraction, which leverages RNIC hardware ACKs to enable wire-latency nilext requests while maintaining linearizability.
- JUNEBERRY, a communication framework that implements the OQ abstraction with commodity RNICs, optimizes performance using hardware/software techniques, and supports durability with persistent memory.
- A set of evaluations on Memcached and PMemKV that demonstrate the performance benefits of JUNEBERRY.

2 Background

In this section, we first provide the background of RDMA (§2.1). Then, we show how existing work utilizes RDMA one-sided verbs to achieve extremely low latency (i.e., wire latency) for storage systems (§2.2).

2.1 RDMA

As the CPU performance stagnates but the network bandwidth continues to grow, it is prevalent to offload network-related processing tasks to NICs [11, 22, 46, 48, 50], freeing valuable CPU cycles. RDMA is a typical example, which is widely deployed in data centers due to high performance and high CPU efficiency [1, 22, 23].

RDMA NICs (RNICs) execute RDMA protocol in hardware and provide queue pair (QP) abstraction to applications. A QP consists of a send queue (SQ) and a receive queue (RQ), each being associated with a completion queue (CQ). A sender submits RDMA commands (verbs), i.e., `WRITE`/`READ`/`SEND`, to the SQ. `WRITE` and `READ` are one-sided, which means that the sender can directly access remote memory without involving the receiver's CPUs. In contrast, `SEND` is two-sided, since the receiver needs to push `RECV` commands to the RQ to prepare receive buffers, which accommodate incoming `SEND` messages. Upon the completion of RDMA commands, the RNIC generates completion entries (CEs) to the associated CQs. RDMA offers several transport modes, of which the *reliable connection (RC)* mode is the most commonly used. RC mode supports 1) reliable data transfer, 2) one-sided `READ`/`WRITE`, and 3) larger-than-MTU message delivery, but it needs one-to-one QP connections.

2.2 Achieving Wire-latency via One-sided Verbs

RDMA one-sided verbs (i.e., `READ`/`WRITE`) are capable of accessing remote memory with remote CPU bypass. This enables extremely low-latency data transfer by avoiding the receiver's CPU software overhead. For simplicity, we term the latency of a single one-sided `READ`/`WRITE` (i.e., *a single round trip with remote CPU bypass*) as **wire latency**.

Researchers have leveraged one-sided verbs to achieve wire-latency replication for storage systems [9, 18, 19, 58].

For example, Mu [9] redesigns Paxos-based state machine replication (SMR): the leader appends a log entry to followers' replication logs via one-sided `WRITE` commands; once the majority of `WRITE` commands have been completed, the leader can ensure the log entry is committed. Followers execute log entries asynchronously in the background. As with the traditional RPC-based approach, Mu completes replication in a single round trip. The difference is that Mu removes the CPU overhead of followers from the replication critical path. Therefore, Mu achieves wire-latency for replication requests (i.e., 1.3μs for leader replicating small data [9]).

3 Motivation

Motivated by the one-sided replication case, we explore a more fundamental question in this work: *Can we achieve wire latency for general storage requests (not just replication requests)?* In this section, we analyze the feasibility and locate the associated technique challenges.

What types of storage requests semantically support wire latency? If a storage request can be realized in a wire-latency manner, it must meet certain requirements, such as not returning an execution result. This is because a wire-latency request is committed without waiting for the server CPU to execute. We find the *nil-externality* property defined by a recent work [21] perfectly describes the requirements.

Formally, a storage interface is nil-externalizing (nilext) if it does not externalize its effects or system state immediately [21]: it just returns an acknowledgment indicating committed, rather than returning an execution result or error. Nilext interfaces are common in storage systems: for example, in Memcached, `set` is nilext; in RocksDB, `put`, `delete`, and `merge` are nilext [21]. By exploiting nil-externality, the execution can be performed *asynchronously* on the server CPU, which makes wire-latency nilext requests possible.

What are the benefits of wire-latency nilext requests? Accelerating nilext requests will notably reduce overall latency (including tail and median) that clients perceive. First, the nilext requests are typically writes and read-modify-writes; executing them is more complicated and thus slower than executing read requests. For example, Memcached's `set` involves memory allocation and object eviction; LevelDB/RocksDB's `put` appends log entries to a write-ahead-log (WAL) file. Making them wire-latency will lower the tail in latency distribution. Second, according to recent studies, many workloads have a non-negligible write ratio [15, 54, 63]. For example, in Twitter, more than 35% cache clusters generate more than 30% `set` requests (nilext) [63]; in Meta's large-scale AI/ML services UP2X, which uses RocksDB for storage backend, about 92.53% of request are RocksDB's `merge` (nilext) [15]. Under these write-heavy workloads, wire-latency nilext requests will reduce the median latency.

Can wire-latency nilext requests be achieved via RDMA `WRITE`? Unfortunately, we give a negative answer by analyzing the limited semantics of RDMA `WRITE`. To guarantee linearizability [24] of storage systems (e.g., after a `set(k,v1)` is committed, the subsequent `get(k)` can return `v1`), we need to produce a linearizable order for requests from *different clients*. However, RDMA `WRITE` cannot achieve coordination between clients: a client must specify the destination address of the server when issuing an RDMA `WRITE`. If we use an external component like sequencer [13, 16, 38] for inter-client coordination, i.e., obtaining an increasing address from the sequencer before issuing `WRITE` and using addresses as the linearizable order, it will induce an extra round trip [13], violating the definition of wire latency, or require programmable switches [16, 38], hindering wide deployment. We also cannot use the server CPU to order requests, since wire-latency requests need to be committed with server CPU bypass. Now, we revisit why RDMA `WRITE` can achieve wire latency for replication. The reason behind it is simple: in the replication case, the server (i.e., a follower) has only a *single* client (i.e., the leader), so the leader can easily specify the order for replication requests by generating increasing destination addresses *locally*; but it does not work for the multi-sender scenario.

In summary, due to interface semantics of nilext requests, making them wire-latency is possible. If realized, we can obtain latency benefits for the upper storage systems. Yet, the challenge is how to guarantee linearizability in the presence of multiple clients, which RDMA `WRITE` cannot tackle.

4 Ordered Queue Abstraction

According to the analysis in §3, RDMA one-sided verbs are incapable of realizing wire-latency nilext requests, since they do not support inter-client coordination inherently. It seems to be in a predicament: the common view is that only one-sided verbs can transfer data in a remote-CPU-bypass manner, the requirement of wire latency.

We break the common view by reexamining what it means to bypass a remote CPU. For an RDMA `WRITE`, when the client gets a completion entry (CE) by polling the CQ, it can guarantee that the sent data will reliably reach the server memory; the entire process does not involve the server CPU. How does the client obtain the guarantee in a remote-CPU-bypass manner? By further looking at the hardware mechanism of RNICs, the answer is explicit: RNICs can generate hardware ACKs due to offloading reliability. Specifically, to support reliable data transfer (RC mode), upon receiving a packet, the receiver-side RNIC generates a hardware ACK to confirm safe delivery, or a negative ACK (NACK) to indicate out-of-order delivery[1]. The sender-side RNIC handles ACKs by generating CEs to the associated CQs; it handles NACKs by performing go-back-N retransmission or Selective Repeat (SR) retransmission (e.g., ConnectX-6 RNIC). This reveals a new insight: *the property of bypassing the remote CPU results from the ability of RNICs to generate hardware ACKs, independent of the type of RDMA verbs*.

[1] If a hardware ACK is lost, the RNIC will resend it.

Inspired by the new insight, we find that it has the potential to achieve wire-latency nilext requests by leveraging two-sided verbs (i.e., RDMA `SEND`). First, RNICs also generate hardware ACKs for RDMA `SEND`, which we can use as the commit signals of nilext requests, ensuring the property of bypassing the remote CPU. Second, unlike RDMA `WRITE`, clients do not need to specify destination addresses for RDMA `SEND`: the server-side RNIC writes the data into buffers in the receive queue (RQ); this gives the server-side RNIC a chance to order requests from different clients.

We therefore propose Ordered Queue (OQ) abstraction, which enables wire-latency nilext requests while maintaining linearizability of the whole storage system. In principle, in OQ abstraction, the server-side RNIC produces a linearizable order for requests from clients, and the server CPU executes requests according to the order. Concretely, OQ abstraction consists of two rules that define how the server-side RNIC and server CPU interact with the RQ, respectively:

Rule 1 *(NIC rule)*. The RNIC allocates receive buffers *in order*: ① the RNIC always allocates receive buffers from the head of RQ, and ② returns hardware ACKs *after* allocation.

Rule 2 *(CPU rule)*. The CPU executes requests *in order*. In other words, the CPU always fetches the first receive buffer not yet executed in the RQ, executes the associated request, and finally returns software responses for non-nilext requests.

A request is committed when the client receives hardware ACKs (for nilext requests) or software responses (for non-nilext requests). Using the above two rules, storage systems will generate a linearizable order that follows the positions of receive buffers in the RQ, even using hardware ACKs as responses of nilext requests. Here, we prove that a request (R_2) sees the effects of any request (R_1) that is committed before it invokes. There are three cases:

Case 1: R_2 is nilext. In this case, according to the definition of nilext requests, R_2 does not need to return any state of storage systems, including effects of committed R_1.

Case 2: R_1 and R_2 are both non-nilext. In this case, R_2 is invoked after the client receives the software response of R_1. According to **Rule 2**, server CPU executes R_1 before returning the software response. Thus, R_2 will reflect the result of R_1.

Case 3: R_1 is nilext but R_2 is non-nilext. In this case, R_2 is invoked after the client receives the hardware ACK of R_1. According to **Rule 1**, at the server side, R_1 has a more advanced position in RQ than R_2. Further, according to **Rule 2**, the server CPU executes R_1 before R_2, so the software response of R_2 will see the effects of R_1.

Here, we give an example of OQ using Figure 1. Suppose the back-end storage system is Memcached. **(a)** Initially, at the server side, the RQ is empty and the Memcached instance contains a KV pair ⟨k, v0⟩. **(b)** Then, a nilext request `set(k,v1)` arrives at the first buffer on the RQ; and the RNIC returns a hardware ACK indicating committed, achieving wire latency. **(c)** After that, a nilext request `set(k,v2)` experiences the same process with the second receive buffer; at the same time, the CPU asynchronously executes requests in RQ in order (i.e., executing `set(k,v1)` first). **(d)** Then, the RNIC receives a non-nilext request `get(k)` and places it in the third buffer in RQ. **(e)** After the CPU has finished executing the first two requests, it executes the `get(k)` and returns a software response containing `v2`. Receive buffers consumed by the CPU can be pushed into the RQ again.

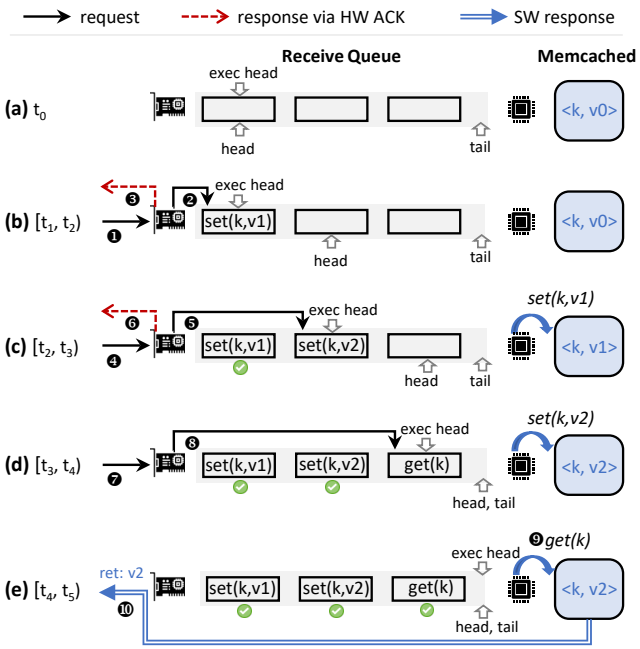

Figure 1: Memcached example with OQ. *This figure shows components at the server side, including RNIC, receive queue, CPU, and a Memcached instance.* `head` *points to the first buffer with no request;* `exec head` *points to the first buffer that is not executed by the CPU.*

Comparison with the pure software solution. One may wonder whether we can achieve the same low latency using a pure software solution that adopts deferred execution: when the server CPU meets a nilext request, it first returns a software ACK to commit, and then executes the request. However, we argue that the OQ abstraction (which achieves wire-latency nilext requests) has remarkable advantages over the pure software approach. For nilext requests, by leveraging the hardware ACKs, the OQ abstraction eliminates the server-side software delay from the critical path, not only for request execution *but also for request responses and software queuing*. In contrast, in the software approach, when the CPU is busy executing requests, the subsequent nilext requests in the RQ cannot be committed immediately and suffer from latency caused by the software queuing. We will conduct the detailed experimental comparison in §6.

Analysis of pathological access patterns. There is a pathological access pattern that makes OQ perform poorly: a client thread sends a non-nilext request *immediately after* receiving the hardware ACK of a nilext request, and both requests reach

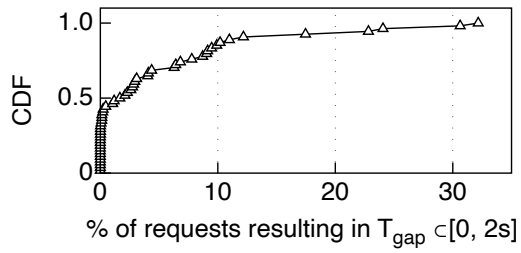

Figure 2: Analysis of 54 clusters in Twitter workloads. *For example, for a point (10, 0.852) in the figure, it means that 85.2% clusters out of 54 have less than 10% requests resulting in $T_{gap} \subset [0, 2s]$.*

the *same* RQ. In this case, the server has no chance of hiding the nilext request's execution latency to the client thread, making the non-nilext request suffer high latency.

For example, suppose the three requests in Figure 1 come from the same client thread, and the thread *immediately* sends a request when receiving the response of the previous one. We assume that the network RTT is 2μs (i.e., 1μs one-way delay), executing `set` needs 10μs, and executing `get` needs 5μs. Without the OQ, the latency of three requests are 12μs, 12μs, and 7μs, respectively. With the OQ, the latency are 2μs, 2μs, and 23μs, respectively: although the two `set` requests are fast, the `get` request has a much higher latency.

However, we argue that the above pathological access pattern is uncommon in real-world workloads. First, there are many storage servers (and each server may have multiple RQs; see §5.1), so two adjacent requests from a client often have different destinations. Second, client threads need to do computational tasks, e.g., webpage rendering (instead of sending storage requests all the time), creating time intervals between requests. As a result, client threads can easily hide the latency of executing nilext requests. Further, we analyze production traces from in-memory caching workloads of Twitter [63]. These traces contain unsampled requests that are sent to two instances (i.e., servers). We chose 1 million consecutive requests to measure the time intervals between a nilext request and the next non-nilext request issued by the same clients (called T_{gap}). Since the traces record timestamps in seconds, we can only coarsely count how many requests result in $T_{gap} \subset [0, 2s]$. Figure 2 shows the cumulative distribution across 54 clusters. In 85.2% (46) clusters, requests leading to $T_{gap} \subset [0, 2s]$ are less than 10%, which gives the OQ abstraction a great opportunity to reduce system latency.

5 Juneberry Design

Juneberry is a communication framework that implements the OQ abstraction using commodity RNICs. Juneberry provides a similar usage as RPC frameworks: The server registers request handler functions, each having a unique request type; Clients issue a request with a specified request type and parameters, to call the associated function.

Juneberry is unique in that it supports wire latency for nilext requests. Clients can mark a request as nilext. For such a request, the server CPU does not return a software response, and the client treats the hardware ACK generated by the server-side RNIC as the commit signal. In this way, the nilext request is committed with extremely low latency, removing the server CPU's overhead from the critical path.

However, when designing Juneberry, there are two obstacles that we must overcome:

- **Correct semantic with high performance**. Despite the simplicity of OQ abstraction, mapping it to the RNIC is not so straightforward. We need to carefully select hardware primitives and scrutinize them for correctness (§5.1). Moreover, while ensuring correctness, we also need to achieve high performance (§5.2).
- **Durability support with crash consistency**. The OQ abstraction lacks a description of how to handle server crashes, which is indispensable for storage systems that contain persistent states (e.g., RocksDB). Therefore, we need to augment the OQ abstraction to support durability (§5.3), including 1) making the hardware ACKs have the durability guarantee, and 2) ensuring that storage systems can recover correctly upon server crashes.

5.1 Implementing OQ Abstraction

Juneberry server leverages shared receive queues (SRQs) to implement the OQ abstraction. SRQ is a standard RDMA component supported by almost all RNICs [12]: it is proposed to reduce memory footprint by supporting multiple QPs sharing a single RQ. Juneberry exploits SRQs for another purpose: by binding multiple QPs to an SRQ, requests from different clients (i.e., QP connections) can be placed in the same queue structure, which is a basic requirement of the OQ abstraction. Juneberry runs in reliable connection (RC) mode, to enable the generation of hardware ACKs. The server pushes receive buffers to the SRQ via RDMA `RECV` commands; client threads send requests to the server's SRQ by issuing RDMA `SEND` commands.

Next, we describe how Juneberry follows the two rules of OQ (§4). In brief, SRQ naturally follows Rule 1 (i.e., NIC rule), but Juneberry relies on some software designs to follow Rule 2 (i.e., CPU rule).

According to RDMA specification [12], the receiver-side RNIC always allocates receive buffers from the head of SRQ[2]. However, if the RNIC returns hardware ACKs *before* allocation due to some radical optimizations (e.g., delayed allocation), the Rule 1 will be violated. Fortunately, by revisiting the hardware semantics of RDMA, we determine that returning ACKs before allocation is *impossible*. Concretely, in RDMA, all memory areas (including receive buffers) that can be accessed by the RNIC are registered as *memory regions* (MRs); each MR is assigned several permissions (e.g., remotely writable). Therefore, for an incoming `SEND` request,

[2] In Section 10.2.9.1 of [12]: "*When an incoming Receive Message arrives on any QP that is associated with an SRQ, the HCA uses **the next available SRQ WQE** to receive the incoming data*".

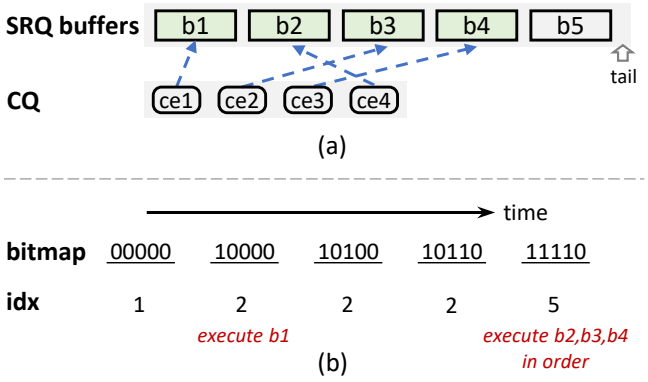

Figure 3: JUNEBERRY enforces Rule 2. ***(a)*** *The order of CEs in the CQ may not match the order of receive buffers in the SRQ.* ***(b)*** *Executing receive buffers in order with the help of the CQ and a bitmap.*

the server-side RNIC must check the permissions of the destination address before returning hardware ACKs, to enforce access control. Checking permissions indicates that the associated receive buffers have been allocated.

In existing RPC frameworks using two-sided verbs [31], the order of executing requests is the same as the order of completion entries (CEs): the server-side CPU polls the CQ to obtain CEs, then locates receive buffers using the CEs, and finally executes requests in the receive buffers. However, such a workflow (i.e., executing requests according to the order of CEs) can not enforce the Rule 2. This is because RDMA only guarantees in-order delivery for CEs generated by the *same* QP [12], but in JUNEBERRY, the SRQ's CQ accommodates CEs from different RC QPs. Hence, RDMA `RECV` commands issued to the SRQ can be completed out-of-order; in other words, when two CEs are generated by different QPs, their order in the CQ may not match the order of receive buffers (e.g., `ce3` and `ce4` in Figure 3(a)).

JUNEBERRY adopts a simple solution to enforce Rule 2, making requests in the SRQ be executed in order. As shown in Figure 3(b), JUNEBERRY maintains a bitmap and a variable `idx` for the SRQ. The bitmap tracks which receive buffers have requests; the `idx` points to the first buffer that is not executed by the CPU. CPU executes requests with the following three steps: ❶ Polls the CQ to obtain CEs and sets associated bits in the bitmap; ❷ If the `idx`-th bit is set, executes the request and advances `idx`; ❸ Repeats the step ❷ until the `idx`-th bit is clear. For example, in Figure 3(b), upon obtaining `ce4` from the CQ, the CPU executes the request in `b2` (since `idx` equals 2), and then executes the next contiguous sequence of requests in the bitmap in order (i.e., `b3` and `b4`).

Scaling to multiple CPU cores. Until now, we have assumed that there is only a single SRQ at server side. However, it is impractical when considering the multi-core architecture of servers. First, when multiple threads operate the same SRQ, the scalability issue happens. Second, when multiple threads execute requests concurrently, producing the same results as sequential execution (for Rule 2) is intricate and low-performance: it involves dependency tracking and resolution, thus inducing expensive cross-thread synchronization.

JUNEBERRY uses data partitioning to support multi-core scaling. Specifically, JUNEBERRY creates multiple SRQs at the server side, each being associated with a worker thread. Each worker thread manages an *exclusive* set of datasets. Such data partitioning is easily achieved in many storage systems [32, 33, 37, 39] such as key-value stores, since they adopt a shared-nothing architecture to improve performance. Clients in JUNEBERRY need to send requests (via RDMA `SEND`) to correct server-side worker threads according to partition scheme. For example, in the Memcached case, clients hash the target key in a `set`/`get` request, and send the request to the SRQ managed by the i-th worker thread, where i = `hash(key) % #(worker threads)`.

5.2 Optimizing Performance

JUNEBERRY materializes the OQ abstraction by leveraging SRQ and adopting software designs (§5.1). However, it poses two performance issues: 1) operating SRQs is inefficient for RNIC; 2) a large number of RC QPs harm system scalability. For each issue, this subsection first explains its causes and negative effects, and then introduces our solutions.

5.2.1 Streamlining RNIC Prefetching on SRQ

The lifecycle of work queue elements (WQEs). When creating an RQ/SRQ, the RNIC driver allocates a contiguous memory region that contains multiple WQEs, which can be accessed by RNICs (via DMA). When pushing a receive buffer to the RQ/SRQ via `RECV` commands, the CPU fills in a *free* WQE with metadata (including the address of the receive buffer), making it *valid*. A valid WQE becomes free when the CPU obtains the associated CE from CQs, which indicates that the WQE has been consumed by a `SEND` request.

The RNIC prefetches valid `RECV` WQEs via DMA and caches them in its SRAM, to accelerate the processing of incoming `SEND` requests. Such prefetching is efficient for standard RQs, since logically contiguous WQEs are also physically contiguous at memory locations, making a single prefetch operation (i.e., a DMA operation) be capable of fetching multiple valid WQEs at a time. For example, as shown in Figure 4(a), receive buffers `b1`, `b2`, `b3` are pushed into the RQ in order, and three WQEs allocated for them are memory-contiguous.

However, SRQ counteracts the benefit of RNIC prefetching due to its structure feature. Specifically, as described in §5.1, the valid WQEs in an SRQ can be completed out-of-order, thus creating fragmented free WQEs like holes. Subsequent `RECV` commands often fail to allocate WQEs in a memory-contiguous manner. As a result, valid WQEs in an SRQ are connected using a link list. For example, as shown in Figure 4(b), the SRQ organizes valid WQEs using the link list `wqe3` → `wqe1` → `wqe5`. Due to the link list structure of WQEs, the RNIC is more susceptible to cache misses (thus stalling in-

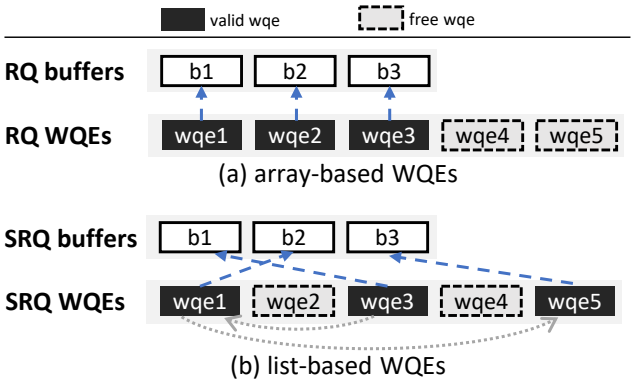

Figure 4: Array-based WQEs vs. list-based WQEs. *In these two figures, receive buffers b1, b2, and b3 are pushed into the RQ/SRQ in order, and the RNIC driver allocates WQEs for them.*

coming traffic), and needs more DMA operations to prefetch WQEs with the pointer-chasing access pattern, both degrading the network performance. We validate it using a simple experiment, where a client launches 24 threads to send 64B messages to a remote server. The remote server uses one CPU core to receive messages and respond with 64B messages; each client thread allows 4 outstanding requests. The server can deliver 2.91Mops/s when using RQ, but the performance drops to 2.24Mops/s when using SRQ. A recent work observes a similar performance degradation on Ethernet NIC that adopts list-based receive (Rx) rings [49].

JUNEBERRY improves the performance of SRQ by allowing a single WQE to accommodate multiple messages, so that a prefetch operation can serve multiple `SEND` requests. This is achieved by exploiting the multi-packet (MP) feature in advanced RNICs [20,56]. For an MP RQ/SRQ, multiple `SEND` requests can be sequentially written to the same receive buffer, and the address of each request is aligned to a k (e.g., 64) byte offset, where k is pre-defined[3]. The MP feature is proposed to reduce memory footprint, but JUNEBERRY uses it to streamline RNIC prefetching and mitigate network traffic stall. In addition, with MP SRQ, JUNEBERRY server can post a large receive buffer (e.g., 64KB) to amortize the overhead of filling in WQEs and ringing doorbells, saving CPU cycles.

5.2.2 Reducing the Number of QPs

Recall that in JUNEBERRY, a client thread can send requests to the MP SRQ of any worker thread according to partition scheme (§5.1), which indicates that there are RC connections between every client thread and every worker thread (RC QPs only support one-to-one connections). Concretely, the server maintains $C{\times}Ct{\times}St$ RC QPs in total, where `C` is the number of client machines, `Ct` is the number of client threads per client machine, and `St` is the number of worker threads in the server. However, it is well-known that RNIC cannot cache states for a large number of QPs due to its size-limited SRAM [18,31,45,57]. Upon cache misses, RNIC issues DMA operations to fetch QP states from host memory, stalling the incoming network traffic and thus degrading performance.

We reduce the number of QPs in the JUNEBERRY server to $C{\times}St$ by employing a client-side delegation mechanism. Specifically, a client machine creates St QPs, each connecting to a remote worker thread. Note that each QP uses a separate context to avoid false synchronization between client threads [58]. Each QP is managed by one client thread, which is called *owner* of the QP. When a client thread is about to send a request to a worker thread, it identifies the owner of the target QP and publishes the request in a reserved slot. The owner scans slots and emits associated requests (including requests generated by itself) to the network using the dedicated QP. The owner is also responsible for processing responses from the QP and delivering these responses to initiators.

5.2.3 Alternative Optimization Techniques

We discuss alternative optimization techniques that can address the above two performance issues.

Array-based SRQ. An alternative approach to streamlining RNIC prefetching is to make RNIC support array-based SRQ (by modifying the firmware of RNIC ASIC). This approach also requires us to modify the driver of RNIC. Compared with array-based SRQ, MP SRQ used by JUNEBERRY has an additional advantage: mitigating CPU overhead by reducing the number of doorbells.

eXtended RC (XRC). XRC [4] is an Infiniband transport service supported by Mellanox RNIC. When using XRC, an RC QP can deliver messages to multiple SRQs in a remote machine. If we adopt XRC in JUNEBERRY, the number of QPs in the server will decrease to $C{\times}Ct$, the total number of client threads in the cluster. Compared with the client-side delegation mechanism (which has $C{\times}St$ QPs at the server side), XRC has a scalability advantage when the server launches a large number of threads.

5.3 Supporting Durability

JUNEBERRY supports durability by leveraging emerging persistent memory (PM) technologies. This subsection first gives a brief background on PM, and then describes how JUNEBERRY uses it for durability and crash consistency. Here, we assume that the back-end storage systems of JUNEBERRY are crash-consistent (unlike Memcached).

A primer on persistent memory (PM). PM is a new class of storage devices that can guarantee persistence with DRAM-like features. Specifically, PM can expose its storage space as physical addresses to CPUs and other devices (e.g., RNICs). Therefore, it can be accessed by CPUs via `load/store` instructions with low latency, or other devices via DMA engines. There are two typical PM devices for datacenters: Intel's Optane PM [5] and Samsung's Memory-Semantic SSD [8]. The former sits in DIMM slots, while the latter is connected by the CXL link. The performance of Optane PM is well-known [62]: it can deliver 2GB/s writes and 6GB/s reads per DIMM, and

[3] We set k to 64, which is the minimum value that our RNICs support, thus reducing internal fragmentation caused by data alignment.

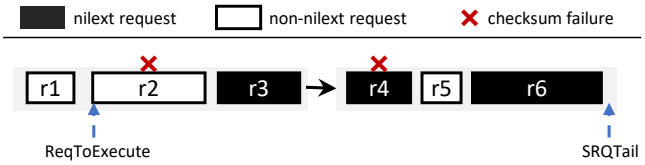

Figure 5: An example of scanning an SRQ upon recovery.

it has the same write latency and 3× higher read latency compared to DRAM. Memory-Semantic SSD is not currently available. Although the CPU cache is volatile, recent CPUs put it into the persistent domain via eADR [3,26,65] or Global Persistent Flush (GPF) in CXL [2], which simplifies the use of PM. With eADR or GPF, upon server crash, the data in the CPU cache will be flushed to PM safely.

Persistent hardware ACKs. We first place SRQ receive buffers in PM, so that requests in these buffers can survive power outages. Recall that JUNEBERRY uses hardware ACKs as commit signals of the nilext requests, to achieve wire latency. Yet, when a client receives the hardware ACK for an RDMA `SEND` command, it only ensures that the `SEND` is allocated a destination address in the SRQ; the DMA write operation in the server-side RNIC may still be in progress, i.e., the `SEND` data may not reach PM successfully. To make the hardware ACK have the persistence guarantee, JUNEBERRY drains DMA operations of `SEND` commands in the server-side RNIC. It does so by letting client threads issue an extra RDMA `READ` (with 1-byte size) to the same QP after every `SEND` command [28, 60]: the PCIe read triggered by the RDMA `READ` will flush the previous PCIe writes. In this way, the hardware ACKs of `READ` ensure the persistence of nilext requests.

In addition, JUNEBERRY performs two optimizations. First, client threads combine the `SEND` command and `READ` command in one round trip, and remove CEs of `SEND`, since the receiver-side RNIC executes these two commands in order [12]. Second, the server disables Data Direct I/O (DDIO), to let the RNIC bypass the CPU cache when performing DMA operations. It can reduce PM write amplification caused by granularity mismatch between a cache line (64B) and a PM internal write (e.g., 256B in Optane PM) [28].

Recoverable SRQ. Simply putting receive buffers in PM does not imply that we can recover requests from the SRQ after power failure. JUNEBERRY devises an identifiable request format to address it. Specifically, each request begins with a 64-bit `checksum` and a 32-bit `size`; `checksum` covers the whole request and `size` is the request length, both being filled by client threads. Moreover, each worker thread persistently maintains two variables for its SRQ in PM: `ReqToExecute` and `SRQTail`. `ReqToExecute` points to the first request not yet executed: it is updated after the worker thread executes a request. `SRQTail` records the last receive buffer in the SRQ; it is updated after the worker thread issues a `RECV` command.

Upon recovery, the server scans SRQs to identify valid requests. Recall that JUNEBERRY leverages the multi-packet feature of RNICs, so a receive buffer may contain multiple requests. Each request has a start address that is aligned to a pre-defined value (§5.2), which simplifies the scanning. For each SRQ, a recovery thread scans receive buffers from `ReqToExecute` to `SRQTail`. When an address results in checksum failure, the thread jumps to the next aligned address in the current receive buffer or the next buffer, continuing to identify valid requests. Finally, as shown in Figure 5, all valid requests (i.e., `r3`, `r5`, and `r6`) are marked. Among them, nilext requests may have returned hardware ACKs as commit signals; therefore, for crash consistency, the recovery thread executes these nilext requests in order (i.e., `r3` and `r6`). Note that we do not need to execute non-nilext request `r5`, because it is uncommitted and thus is concurrent with the subsequent requests in the SRQ (i.e., `r6` in this example).

There is a corner case: `ReqToExecute` points to a valid request that is nilext and non-idempotent (e.g., fetch-and-add without returning values). The recovery thread *cannot* determine whether this request has returned a hardware ACK. It also *cannot* determine whether this request has been executed, since executing the request and updating the `ReqToExecute` are not atomic. As a result, the recovery thread *cannot* decide whether to execute the request (may cause duplicated execution) or skip it (may miss a committed request).

To address the above issue, we make all non-idempotent requests *software controllable* (i.e., have the same workflow as non-nilext requests): the worker thread ❶ executes a request, ❷ updates `ReqToExecute`, and ❸ finally returns a software response; client threads use the software response as the commit signal. Upon recovery, the recovery thread can skip the non-idempotent (or non-nilext) request pointed by `ReqToExecute`, since the client thread has not received the request's response, according to the order of ❶ ❷ ❸.

6 Evaluation

We seek to answer the following questions in our evaluation:

- How do optimization techniques proposed in §5.2 affect JUNEBERRY's throughput? (§6.2)
- How does JUNEBERRY perform when applying to in-memory storage systems like Memcached? (§6.3)
- How does JUNEBERRY perform when applying to persistent storage systems like PMemKV? (§6.4)

6.1 Experimental Setup

We conduct experiments in a cluster having one server machine and 8 client machines. The server machine is equipped with two 18-core Intel Xeon Gold 6240M CPUs, 192GB DRAM, and 1.5TB Optane PM (three Optane DIMMs per CPU). Each client machine is equipped with two 12-core Intel Xeon E5-2650 CPUs and 128GB DRAM. Each machine installs a 100Gbps Mellanox ConnectX-5 RNIC; all RNICs are connected to a Mellanox 100Gbps IB Switch. To avoid the severe NUMA impacts on PM [35, 66], we only use one

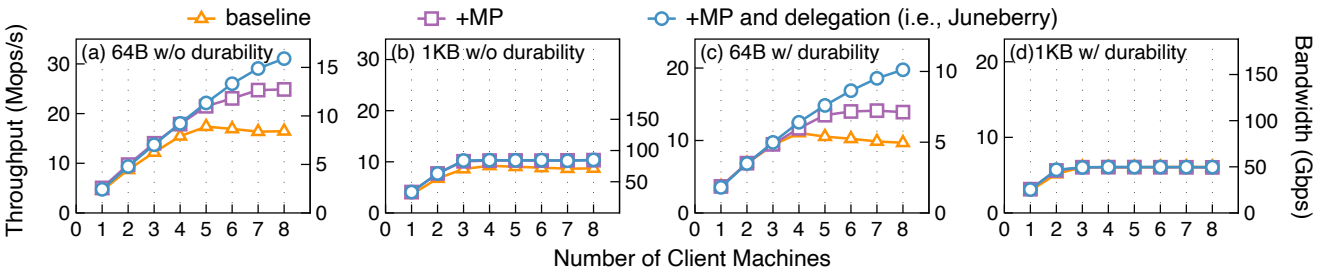

Figure 6: Throughput/Bandwidth with the varying number of client machines.

CPU and the associated DRAM/PM resources in the server.

There are two versions of our system: JUNEBERRY and JUNEBERRY-D. The latter supports durability (§5.3), so it places the server-side receive buffers in PM and disables DDIO. Without explicit mention, each MP SRQ consists of 48 receive buffers of 16KB.

Target comparisons. In §6.3 and §6.4, we will compare JUNEBERRY with three systems:

- **RawRPC.** RawRPC uses the same codebase as JUNEBERRY, including optimizations such as MP features and delegation (§5.2). Like traditional RPC frameworks, in RawRPC, the server CPU handles requests and then returns responses.
- **DeferredExec.** DeferredExec also uses the same codebase as JUNEBERRY. It optimizes nilext requests using a pure software solution: when processing a nilext request, the server CPU first returns a software ACK to commit and then executes it.
- **eRPC.** eRPC [29] is a state-of-the-art RPC framework. It uses unreliable datagrams for network communication and implements retransmission and congestion in software.

6.2 Microbenchmarks

In this experiment, we use microbenchmarks to demonstrate the efficacy of two techniques proposed to improve JUNEBERRY's performance: MP features and client-side delegation. Each client machine launches 24 threads: each client thread keeps one request in flight, and sends the request to a randomly-chosen remote worker thread. The server machine launches 18 threads: when a worker thread receives a request, it replies using the same size message. Figure 6 shows the system throughput with the varying number of client machines, under different settings (i.e., small requests vs. large requests, w/o durability vs. durability).

Small requests without durability. We make the following observations from Figure 6(a). First, with 8 client machines, MP improves the throughput of 64B requests from 16.45Mops/s to 24.86Mops/s (i.e., 1.51×). This is because by accommodating multiple requests in a single receive buffer, MP greatly mitigates the overhead of RNIC prefetching on list-based SRQ WQEs and thus lowers the probability of RNIC cache misses. Second, client-side delegation further increases throughput by 1.25×, since it reduces the number of RC QPs from 3456 (i.e., 8×24×18) to 144 (i.e., 8×18); in this way, the QP states become smaller, alleviating the cache thrash caused by RNICs' size-limited SRAM. Third, these two techniques make JUNEBERRY scale well; without them, JUNEBERRY can not scale to more than 5 client machines.

Large requests without durability. From Figure 6(b) we can see that even with large requests (i.e., 1KB), the baseline can only achieve 71Gbps due to inefficient RNIC prefetching on SRQ WQEs. When using multi-packet (MP) SRQ, the bandwidth increases to 85Gbps. What needs to be explained here is why JUNEBERRY does not reach the raw performance of RNICs (i.e., 100Gbps). We suspect the cause is the hardware limitation of the SRQ, because when we replace SRQs with standard RQs, the bandwidth can approach 100Gbps.

Small requests with durability. We make two observations from Figure 6(c). First, as with DRAM, the two techniques significantly boost the throughput of small requests when using PM: in the case of 8 client machines, MP and delegation improve throughput by 1.44× and 1.42×, respectively. Second, supporting durability brings 36.40% throughput degradation (compared with Figure 6(a)): from 31.07Mops/s to 19.76Mops/s. This is because 1) every RDMA SEND command is followed by an RDMA READ command for durability, consuming the IOPS of the server-side RNIC; 2) handling PM-resident requests consumes more CPU cycles, since PM has 3× higher read latency than DRAM.

Large requests with durability. As shown in Figure 6(d), for large requests, the limited bandwidth of PM (6GB/s for 3 Optane DIMMs) becomes the performance bottleneck.

6.3 In-Memory Caching: Memcached

In this experiment, we use Memcached (version: 1.6.19) as the back-end storage system. We remove network-related parts in Memcached's codebase and produce a local library that can be linked to JUNEBERRY server. We spawn 24 threads per client machine (8 client machines in total); client threads generate requests at a given rate with Poisson arrivals. The server machine launches 16 threads to handle requests and reserves 2 threads for background tasks (e.g., LRU maintainer).

We select four representative in-memory caching work-

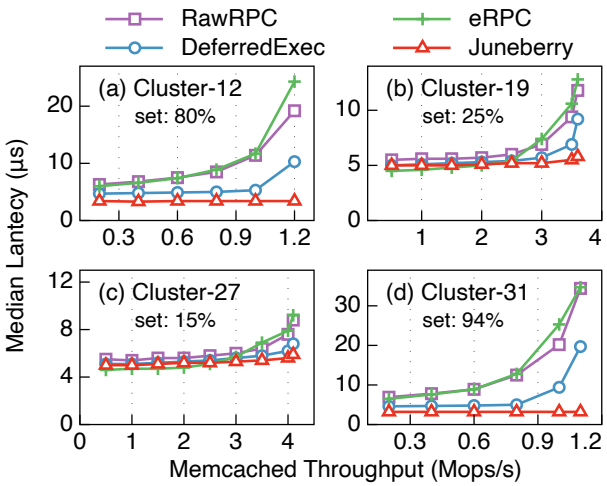

Figure 7: Median latency vs. throughput. *We use Memcached (version: 1.6.19) as the back-end storage system.*

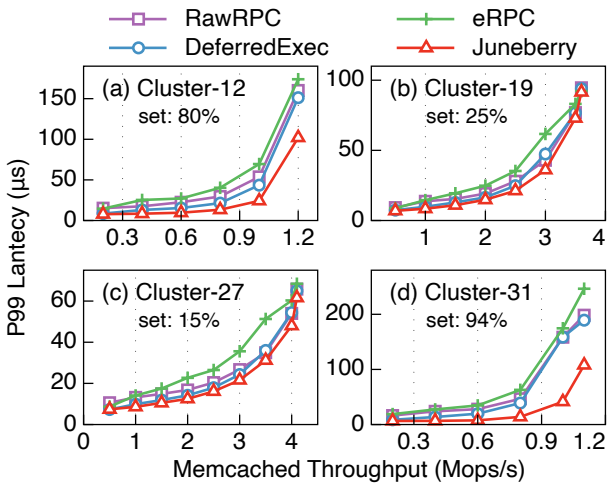

Figure 8: P99 latency vs. throughput. *We use Memcached (version: 1.6.19) as the back-end storage system.*

	ratio of `set`	key size	value size	Zipf alpha
Cluster-12	80%	44B	1030B	0.3048
Cluster-19	25%	42B	101B	0.735
Cluster-27	15%	66B	8B	1.065
Cluster-31	94%	41B	15B	0

Table 1: Settings of in-memory caching workloads.

loads from Twitter [63], as shown in Table 1. Among them, Cluster-12 and Cluster-31 are write-intensive; Cluster-19 and Cluster-27 are read-intensive.

Figure 7 shows median latency under varying throughput. All systems achieve the same peak throughput since the back-end storage engine is the throughput bottleneck. We can make the following observations.

First, under peak throughput, compared with `RawRPC`, JUNEBERRY achieves 82.30%/50.74%/32.89%/90.70% lower median latency in Cluster-12/19/27/31 workloads. This is because JUNEBERRY makes nilext requests (i.e., `set` requests) wire-latency by leveraging hardware ACKs as commit signals. For workloads containing more nilext requests (e.g., Cluster-12 and Cluster-31), the latency reduction is more pronounced.

Second, by deferring execution of nilext requests until after returning responses, `DeferredExec` reduces median latency of `RawRPC`. However, `DeferredExec` is still outperformed by JUNEBERRY: for example, when approaching peak throughput under Cluster-12, `DeferredExec` has 3.03× higher median latency compared with JUNEBERRY. This is because when worker threads of `DeferredExec` are executing requests, they cannot immediately return software responses for nilext requests residing in RQs. In contrast, JUNEBERRY offloads the task of returning nilext requests' ACKs to the RNIC hardware, avoiding software queuing caused by the busy CPU. As a result, the median latency of JUNEBERRY is not only low but also stable under different degrees of loads. This demonstrates that in the microsecond era, hardware/software co-design (e.g., the OQ abstraction) is necessary to further realize latency reduction for storage systems. It is also worth noting that compared to `DeferredExec`, JUNEBERRY reduces the number of network packets used for returning responses.

Third, `eRPC` suffers much higher latency than JUNEBERRY at peak throughput, but when workloads are read-intensive and not heavy, it median latency is slightly lower. For example, as shown in Figure 7(c), when the throughput is less than 2.5Mops/s under Cluster-27, `eRPC` has 100~400ns lower median latency than JUNEBERRY. This is because at this point, software queuing does not affect latency, but the client-side delegation in JUNEBERRY incurs extra software overhead such as cross-thread communication.

Figure 8 shows 99th percentile (P99) tail latency under varying throughput, from which we can make the following observations. First, under write-intensive workloads Cluster-12 and Cluster-31, JUNEBERRY yields a P99 latency that is up to 32.43% and 42.86% lower than the second best system (i.e., `DeferredExec`), respectively. This is because the OQ abstraction enables fast commit of nilext requests, which represent a large fraction of these two workloads.

Second, under read-intensive workloads Cluster-19 and Cluster-27, the P99 latency reduction achieved by JUNEBERRY is limited. Take Cluster-19 for example, compared with `RawRPC`, JUNEBERRY decreases P99 latency from 10.6μs to 7.4μs with 0.5Mops/s throughput, and from 65.8μs to 61.8μs with 4.1Mops/s throughput (peak throughput). This is because a small fraction of `get` requests produce pathological access patterns (recall §4), i.e., the time interval between them and previous `set` requests from the same client threads is too short to hide execution latency of `set` requests, leading to high tail latency. Therefore, the improvement from the OQ abstraction is overshadowed. This also explains why JUNEBERRY is less effective in reducing tail latency than median latency.

6.4 Persistent KV Store: PMemKV

This experiment explores the performance of persistent storage systems with JUNEBERRY-D. We select PMemKV [7] as

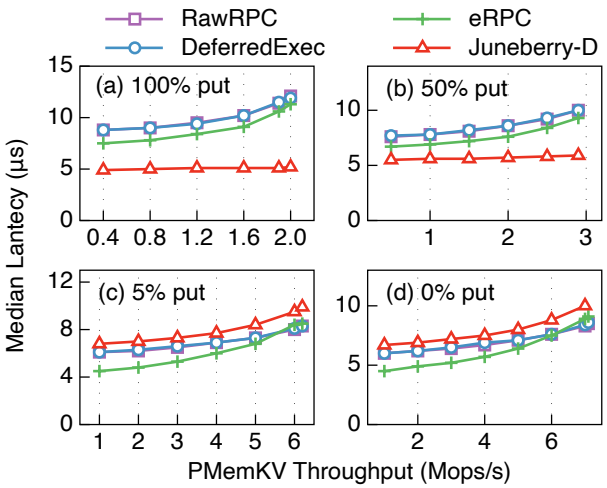

Figure 9: Median latency vs. throughput. *We use PMemKV as the back-end storage system.*

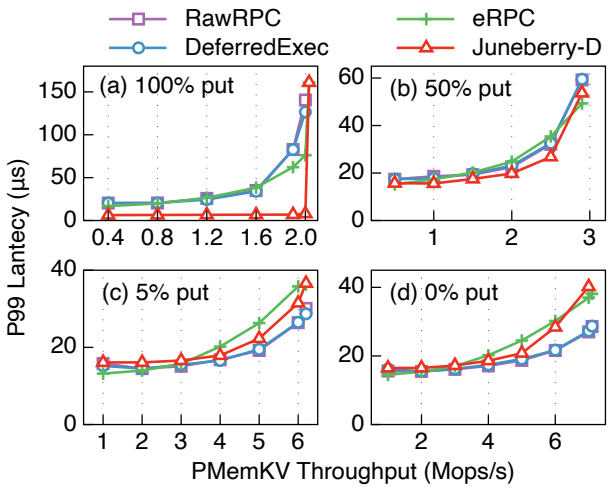

Figure 10: P99 latency vs. throughput. *We use PMemKV as the back-end storage system.*

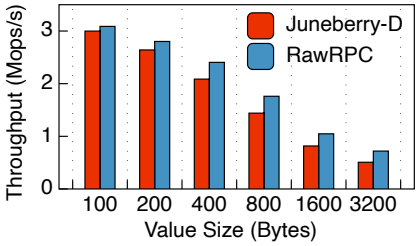

Figure 11: Throughput with different value size. *We use PMemKV as the back-end storage system. The* `put` *ratio is 50%.*

the backend; it is a persistent KV store that maintains objects in PM. The experiment setup is the same as in §6.3, except that the server launches 18 threads. We use YCSB-like workloads with four types of `put`/`get` ratio: write-only (100% `put`), read-only (100% `get`), write-intensive (50% `put` and 50% `get`), read-intensive (5% `put` and 95% `get`). We set the object size to 91-byte, which is the average object size in Meta's largest KV system, i.e., ZippyDB [15]. Workloads follow a Zipf 0.99 access distribution.

Figure 9 depicts median latency with increasing throughput, from which we make two observations. First, Juneberry-D reduces median latency by 57.08%/56.33%/53.92% against `RawRPC`/`DeferredExec`/`eRPC` under write-only workloads; 40.83%/40.83%/36.71% under write-intensive workloads (i.e., 50% `put`). By leveraging persistent hardware ACKs along with PM-resident recoverable SRQ, Juneberry-D can quickly commit a nilext request without sacrificing crash consistency.

Second, Juneberry-D is inferior to other systems under read-intensive (i.e., 5% `put`) and read-only workloads. Specifically, compared with `RawRPC`, Juneberry-D has 1.19× (1.6μs) and 1.20× (1.7μs) higher median latency. This is because in Juneberry-D, the server ① disables DDIO, harming the communication latency between RNICs and host memory, and ② uses PM-resident receive buffers, leading to higher latency spent by the CPU in fetching requests. In addition, under read-only workloads, Juneberry-D has about 1.5% lower peak throughput compared with other systems.

Figure 10 further shows P99 latency of PMemKV. Under write-only workloads (Figure 10(a)), when the throughput is 2.0Mops/s, Juneberry-D exhibits 94.38%/93.76%/89.61% lower P99 latency against `RawRPC`/`DeferredExec`/`eRPC`. However, when the throughput offered is slightly higher than Juneberry-D can handle, its tail latency increases dramatically. This is because when all requests are nilext, the server-side SRQ can easily have no free buffers, causing the server-side RNIC to generate the receive not ready (RNR) error.

When workloads are read-intensive (5% `put`) and read-only, Juneberry-D has up to 1.28× and 1.49× higher P99 latency due to PM receive buffers and disabled DDIO. Combining Figure 9 and Figure 10, we can summarize that when supporting durability, the OQ abstraction brings latency benefits under write-intensive workloads, but incurs latency overhead under read-intensive workloads.

In Juneberry-D, PM-resident receive buffers consume PM write bandwidth, which inevitably affects the performance of back-end storage systems using PM (recall that PM write bandwidth is limited). We conduct an experiment to show it. We use 8-byte keys and vary the value size; the ratio of `put` requests is 50%. Figure 11 presents the result. When the value size is small, the throughput degradation of Juneberry-D is acceptable: only 3.0% with 100B values. However, when values become larger, the size of `put` requests increases and thus more PM write bandwidth is consumed on server-side SRQ, reducing the available PM bandwidth for the back-end storage system. As a result, with 3200B values, `RawRPC` delivers 1.42× higher throughput against Juneberry-D. Fortunately, small objects are very common in real-world workloads [15, 43, 63], so Juneberry-D can achieve wire-latency nilext requests at a small cost in throughput in many cases. In addition, if the back-end storage system uses a storage device that is not a PM (e.g., SSD or HDD), Juneberry will not induce throughput degradation.

7 Discussion

Supporting durability without Optane PM. In our experiments, we use Optane PM to store requests for durability. However, Intel has fully killed off its Optane Memory business. Here, we provide two alternative solutions:

- **CXL-based SSD.** By attaching SSD devices on CXL links, CXL-based SSD can expose load/store interfaces to RNICs. Thus, we can place SRQ buffers on CXL-SSD and the RNIC will DMA requests to it, guaranteeing data durability. Some CXL-based SSD products (e.g., Samsung's Memory-Semantic SSD [8]) will enter the market in the near future.
- **UPS-backed DRAM.** Uninterruptible power supply (UPS) is a type of power system that makes DRAM durable. Specifically, with the help of UPS, the contents of DRAM will be written to SSD when the power fails [19]. Compared with Optane PM, USP-backed DRAM will not induce extra latency overhead to JUNEBERRY. Moreover, since the size of the SRQ buffers is small, the battery provisioning costs of UPS are limited for JUNEBERRY.

Implementing JUNEBERRY with different technologies. JUNEBERRY exploits hardware ACKs generated by RNICs to enable fast commit. It can also be implemented on other NICs supporting hardware ACKs [46, 50], such as FlexTOE [50], a TCP offload engine (TOE) on SmartNICs. Moreover, the idea of fast commits can be applied to other levels of the network stack. For example, for Linux socket-based applications, we can return a software ACK in the Linux kernel by implementing an in-kernel eBPF hook via XDP. By doing so, the latency overhead caused by the context switch can be avoided.

8 Related Work

RDMA communication. Existing works use RDMA for inter-server communication in two ways: 1) implementing RPC upon RDMA verbs [27, 29–31, 52] and 2) directly accessing remote memory via one-sided verbs [9, 18, 55, 59, 64]. RPC enables complicated server-side execution (e.g., index traversal) in one round trip. One-sided access can bypass remote CPUs and thus achieve low latency, but is restricted to simple read and write operations. Thus, retrieving data using one-sided verbs in a storage system typically involves multiple round trips due to indexing and concurrency control (e.g., Sherman [55] and CliqueMap [51]). JUNEBERRY based on the OQ abstraction provides a new way for inter-server RDMA communication: the server CPU can execute requests like RPC, supporting general operations within one round trip, and nilext requests are committed with remote CPU bypass, achieving extremely low latency.

NIC assisted systems. Recent works leverage SmartNICs and customized NICs to accelerate system software [14, 17, 25, 34, 36, 40, 41]. Among them, RPCValet [17], Dagger [36], and nanoPU [25] targets RPC systems: they offload tasks of the RPC stack to NICs, such as request scheduling and (de)serialization, and use memory interconnects (instead of PCIe) for CPU-NIC communication to lower latency. JUNEBERRY can also be viewed as leveraging NICs to accelerate RPC systems: it relies on NICs for providing a fast path to commit nilext requests. However, different from the above works, JUNEBERRY does not require hardware modification; it repurposes hardware ACKs that exist on commodity NICs supporting reliable data transfer (e.g., RNICs).

Combining RDMA and PM. RDMA and PM can offer low-latency data transfer and storage, respectively, so researchers are actively exploring how to combine them efficiently [10, 28, 42, 53, 53, 60, 61]. Kalia et al. [28] and Wei et al. [60] present a set of guidelines to optimize the interaction between RDMA and PM. Octopus [42], Orion [61], and Assise [10] are distributed file systems powered by RDMA and PM. They support one-sided RDMA access to PM-resident file data, thus mitigating CPU overhead and avoiding unnecessary data copying. Clover [53] is a PM-based key-value store that uses RDMA networks. Clients access objects in PM via RDMA `WRITE` and `READ`, and resolve concurrent conflicts via RDMA atomic verbs. JUNEBERRY adopts a different approach to combine RDMA and PM: it places receive buffers in PM to persistently store incoming RDMA `SEND` requests, rather than directly accessing PM-resident data via one-sided RDMA.

Nil-externality. Ganesan et al. identify nil-externality [21], a property for interfaces if they do not externalize effects. By exploiting nil-externality, they propose nilext-aware replication, which improves replication performance by lazily ordering and executing nilext updates. The design of OQ abstraction gets inspiration from their work. Several works on storage systems also perform early commit and asynchronous execution. Xsyncfs [47] commits a file system operation without performing disk I/O, and defers execution until external output is generated (e.g., send data to the screen or network). ScaleDB [44] commits a transaction without updating the global range index, and introduces a concurrency control mechanism for transaction serializability. JUNEBERRY is designed for networked environments and can be used for many storage systems. Moreover, JUNEBERRY's early commit is achieved by exploiting hardware ACKs, thus eliminating software delay from the critical path.

9 Conclusion

In this paper, we propose Ordered Queue (OQ) abstraction, which accelerates nilext requests by leveraging NICs's hardware ACKs for fast commit. By doing so, a nilext request can be committed in a single round trip with remote CPU bypass, achieving extremely low latency. We materialize OQ abstraction by designing JUNEBERRY, a communication framework running on commodity RDMA NICs. JUNEBERRY uses a set of techniques to improve performance and incorporates persistent memory to support durability. JUNEBERRY significantly lowers the latency of storage systems under write-intensive workloads. This work demonstrates that making storage software have visibility into network-level knowledge can bring performance benefits.

Acknowledgements

We sincerely thank our shepherd Mariano Scazzariello for helping us improve the paper. We also thank the anonymous reviewers for their feedback, Junru Li for sharing his experience on networking. This work is supported the National Natural Science Foundation of China (U22B2023, 62472242, 62402204), Young Elite Scientists Sponsorship Program by CAST (2023QNRC001), and Postdoctoral Fellowship Program of CPSF (GZC20231296).

References

[1] Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'23, pages 49–67, Boston, MA, April 2023. USENIX Association.

[2] Compute Express Link: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/download-the-specification, 2024.

[3] eADR. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html, 2024.

[4] eXtended Reliable Connection (XRC). https://downloads.openfabrics.org/Media/SC07/2007_SC_Nov_XRC.pdf, 2024.

[5] Intel's Optane PM. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html, 2024.

[6] Memcached - a distributed memory object caching system. https://memcached.org/, 2024.

[7] PMemKV - Key/Value Datastore for Persistent Memory. https://github.com/pmem/pmemkv, 2024.

[8] Samsung's Memory-Semantic SSD. https://samsungmsl.com/cmmh/, 2024.

[9] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond Consensus for Microsecond Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'20, pages 599–616. USENIX Association, November 2020.

[10] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027. USENIX Association, November 2020.

[11] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling Programmable Transport Protocols in High-Speed NICs. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, pages 93–110, USA, 2020. USENIX Association.

[12] InfiniBand Trade Association et al. InfiniBandTM Architecture Specification Volume 1 Release 1.3 (General Specifications), 2015.

[13] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 1–14, San Jose, CA, April 2012. USENIX Association.

[14] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 228–242, New York, NY, USA, 2021. ACM.

[15] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies*, FAST'20, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[16] Inho Choi, Ellis Michael, Yunfan Li, Dan R. K. Ports, and Jialin Li. Hydra: Serialization-Free Network Ordering for Strongly Consistent Distributed Applications. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 293–320, Boston, MA, April 2023. USENIX Association.

[17] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. RPCValet: NI-Driven Tail-Aware Balancing of us-Scale RPCs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 35–48, New York, NY, USA, 2019. ACM.

[18] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[19] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 54–70, New York, NY, USA, 2015. ACM.

[20] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making Kernel Bypass Practical for the Cloud with Junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 55–73, Santa Clara, CA, April 2024. USENIX Association.

[21] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting Nil-Externality for Fast Replicated Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP'21, pages 440–456, New York, NY, USA, 2021. ACM.

[22] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When Cloud Storage Meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'21, pages 519–533. USENIX Association, April 2021.

[23] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM'16, pages 202–215, New York, NY, USA, 2016. ACM.

[24] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990.

[25] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256. USENIX Association, July 2021.

[26] Zhicheng Ji, Kang Chen, Leping Wang, Mingxing Zhang, and Yongwei Wu. Falcon: Fast OLTP Engine for Persistent Cache and Non-Volatile Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 531–544, New York, NY, USA, 2023. ACM.

[27] Tianyang Jiang, Guangyan Zhang, Zhiyue Li, and Weimin Zheng. Aurogon: Taming Aborts in All Phases for Distributed In-Memory Transactions. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 217–232, Santa Clara, CA, February 2022. USENIX Association.

[28] Anuj Kalia, David Andersen, and Michael Kaminsky. Challenges and Solutions for Fast Remote Persistent Memory Access. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 105–119, New York, NY, USA, 2020. ACM.

[29] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[30] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[31] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 185–201, USA, 2016. USENIX Association.

[32] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):1496–1499, aug 2008.

[33] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. ACM.

[34] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP'21, pages 756–771, New York, NY, USA, 2021. ACM.

[35] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 424–439, New York, NY, USA, 2021. ACM.

[36] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 36–51, New York, NY, USA, 2021. ACM.

[37] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 447–461, New York, NY, USA, 2019. ACM.

[38] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 467–483, Savannah, GA, November 2016. USENIX Association.

[39] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[40] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E. Stephens, Hassan Wassel, and Aditya Akella. RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1293–1308, Boston, MA, April 2023. USENIX Association.

[41] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259. USENIX Association, November 2020.

[42] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, July 2017. USENIX Association.

[43] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 243–262, New York, NY, USA, 2021. ACM.

[44] Syed Akbar Mehdi, Deukyeon Hwang, Simon Peter, and Lorenzo Alvisi. ScaleDB: A Scalable, Asynchronous In-Memory Database. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 361–376, Boston, MA, July 2023. USENIX Association.

[45] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 212–227, New York, NY, USA, 2021. ACM.

[46] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, pages 77–92, USA, 2020. USENIX Association.

[47] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 1–14, USA, 2006. USENIX Association.

[48] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. Autonomous NIC Offloads. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'21, pages 18–35, New York, NY, USA, 2021. ACM.

[49] Boris Pismenny, Adam Morrison, and Dan Tsafrir. ShRing: Networking with Shared Receive Rings. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 949–968, Boston, MA, July 2023. USENIX Association.

[50] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'22, pages 87–102, Renton, WA, April 2022. USENIX Association.

[51] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K.

Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 93–105, New York, NY, USA, 2021. Association for Computing Machinery.

[52] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is Faster than Server-Bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 1–15, New York, NY, USA, 2017. ACM.

[53] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48. USENIX Association, July 2020.

[54] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'20, pages 449–462, Santa Clara, CA, February 2020. USENIX Association.

[55] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pages 1033–1048, New York, NY, USA, 2022. Association for Computing Machinery.

[56] Qing Wang, Youyou Lu, Jing Wang, and Jiwu Shu. Replicating Persistent Memory Key-Value Stores with Efficient RDMA Abstraction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 441–459, Boston, MA, July 2023. USENIX Association.

[57] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, and Chuanxiong Guo. SRNIC: A Scalable Architecture for RDMA NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1–14, Boston, MA, April 2023. USENIX Association.

[58] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid is Better. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 233–251, USA, 2018. USENIX Association.

[59] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[60] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. Characterizing and Optimizing Remote Persistent Memory with RDMA and NVM. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 523–536. USENIX Association, July 2021.

[61] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 221–234, Boston, MA, February 2019. USENIX Association.

[62] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[63] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'20, pages 191–208. USENIX Association, November 2020.

[64] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.*, 10(6):685–696, feb 2017.

[65] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. NBTree: A Lock-Free PM-Friendly Persistent B+-Tree for EADR-Enabled PM Systems. *Proc. VLDB Endow.*, 15(6):1187–1200, feb 2022.

[66] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 179–193, Carlsbad, CA, July 2022. USENIX Association.